

移动平台
开发书库



Android 平台开发之旅

汪永松 编著

第2版



- ① 涵盖Android 2.2.x/2.3.x/3/4
- ② 横跨手机与平板设备
- ③ 案例丰富，涉及各类主流应用
- ④ 深入介绍平台新特性



机械工业出版社
CHINA MACHINE PRESS

移动平台开发书库

Android 平台开发之旅

第2版

汪永松 编著



机械工业出版社

PDG

本书涵盖 Android 3/4 的新特性,立足实际的开发案例,介绍了 Android 平台开发的基础概念、实用技术和应用模式。主要包括应用程序框架、高级界面、数据库应用、网络通信与 Web 开发、无线通信、多媒体应用、个人信息管理、电话系统管理、XML 应用、地图应用和系统信息管理。

本书精选案例 95 例,其中 70 例基于 Android 2.2.1 和 2.3.5 实机开发,25 例基于 Android 3.x 模拟器。

本书主要面向具有一定移动平台开发经验的开发者,以及有兴趣进行 Android 平台开发的程序员。

本书配有代码 CD 一张。

图书在版编目 (CIP) 数据

Android 平台开发之旅 / 汪永松编著. —2 版. —北京: 机械工业出版社, 2012.3

(移动平台开发书库)

ISBN 978-7-111-37276-9

I. ①A… II. ①汪… III. ①移动电话机-应用程序-程序设计
IV. ①TN929.53

中国版本图书馆 CIP 数据核字 (2012) 第 013723 号

机械工业出版社 (北京市百万庄大街 22 号 邮政编码 100037)

策划编辑: 车 忱

责任编辑: 车 忱 李 宁

责任印制: 乔 宇

三河市宏达印刷有限公司印刷

2012 年 3 月第 2 版·第 1 次印刷

184mm×260mm·29 印张·718 千字

0001-4000 册

标准书号: ISBN 978-7-111-37276-9

ISBN 978-7-89433-346-9 (光盘)

定价: 69.80 元 (含 1CD)

凡购本书,如有缺页、倒页、脱页,由本社发行部调换
电话服务 网络服务

社服 务 中 心: (010) 88361066

销 售 一 部: (010) 68326294

销 售 二 部: (010) 88379649

读者购书热线: (010) 88379203

门户网: <http://www.cmpbook.com>

教材网: <http://www.cmpedu.com>

封面无防伪标均为盗版

前 言

从 1.0 到 4.0, Android 平台走进我们的视野已经 3 年了。虽然在不断完善功能和提升性能,但 Android 平台还是没能摆脱分分合合的模式。为了满足以平板电脑和智能电视机为代表的大屏幕设备的应用需求,Android 3 平台另辟天地,开启了 Android 平台“合久必分”的局面:2.x 版本仍然面向手机设备,继续稳扎稳打;3.x 版本面向大屏幕设备,不断提升改进。

然而,为了解决平台兼容性、开发成本等问题,Android 4 来了!4.0 版本似乎充分诠释了“分久必合”:无论平板电脑还是智能手机都可以使用该系统。这对于 Android 平台开发者,似乎可以缓解一下不断更新的 SDK 带来的压力,但是 Android 4 又会把人们带向何方?根据最新统计数据,当前各版本 Android 设备中所占份额较大的是 2.2 和 2.3.3~2.3.7 版本;对于平板电脑,3.x 版本是当前最成熟的选择。由此看来,稳定和实用始终是 Android 设备用户所追求的目标。

出于此考虑,作者在选择以 Android 主流平台作为出发点的同时兼顾 4.0.x 和 3.x 平台的实机开发,内容兼顾实用性及新特性,给读者带来切实的帮助。

本书的特色

本书在内容编排方面具有三个特点。

第一点:内容全面,讲解细透。本书内容涵盖了 Android 主流平台主要的功能特性,对平台的底层机制进行了系统而全面的剖析,对主流的技术及应用模式进行完整介绍;其间还使用通俗易懂的比喻帮助读者理解概念;通过联想对比帮助读者加深对应用的认识。

第二点:案例丰富,领先实用。本书精选案例 95 例,其中 70 例基于 Android 2.2.1 和 2.3.5 实机开发,25 例基于 Android 3.x 模拟器开发。这些开发案例都是经过挑选和充分调试,其内容兼顾手机实机开发和最新 Android 特性,并具备较高的实用价值。

第三点:结构合理,循序渐进。本书内容的编排遵循“由表及里、从内而外,先专项、后综合”的思路,从使用功能到应用机制剖析;从前台界面设计到后台功能实现;从内部信息管理到外部连接通信;从单一功能到集成应用。步步为营地帮助读者加深对 Android 平台开发的认识。

本书内容

第 1 章从 Android 平台的起源和内容着手,重点介绍了平台的核心概念。第 2 章详细介绍了如何搭建开发环境和常用的开发技巧。在初步具备开发知识的基础上,第 3 章对应用程序组件以及平台中的重要机制进行深入剖析。这三章中,以理论为主,实践为辅,为后续开



发做好铺垫。

第4章先从界面组件使用模式着手，然后使用大量开发案例对常用布局和视图的具体使用进行详细说明，此外还重点介绍了若干界面框架的使用，其中包括 Android 平台的新特性，如操作栏、片段组件、拖放操作、动画效果等。

第5章介绍了用于底层用户界面控制的组件及其使用。

第6章介绍了各种类型文件的使用模式和文件系统监视。

第7章对较为特殊的文件——数据库的应用模式进行全面介绍，包括内容提供框架、SQLite 和 Db4o 数据库应用。

第8章介绍了网络连接管理、网页视图应用集成、浏览器信息管理和下载管理。

第9章对 Android 平台支持的无线通信方式——短消息、蓝牙通信和近距离通信进行实例介绍。

第10章全部使用实机案例对多媒体应用进行介绍，主要包括音频回放与录制、视频回放与录制及相机应用，此外还对媒体信息管理进行说明。最后通过一款音乐盒工具对多媒体的集成应用进行分析讨论。

第11章介绍了个人信息管理内容及模式，主要包括联系人信息、电话号码、公司信息等。

第12章全部使用实机案例对电话系统应用进行介绍，主要包括获取电话信息、侦听手机状态及手机网络基站定位，此外还介绍了拨号器的调用和获取呼叫日志。

第13章对平台中主要的 XML 应用进行介绍，包括 XML Pull API 的使用和 XML 资源解析。

第14章对地图 API 的功能进行详细说明，并结合地图视图对地图应用模式进行实例说明，主要包括基于地图的定位。

第15章对 Android 平台中重要的系统管理接口的使用进行了说明。

第16章对程序资源的定义及使用提供了详细的参考，此外还对 Android SDK 工具的使用进行说明。

附录对随书源代码的使用进行了说明。

本书中的一些约定

注意 提醒读者应该给予重视的内容。

提示 是对读者有所帮助的一些技巧。

源代码说明

本书中的代码分为 Java 代码和 XML 代码。囿于篇幅，部分非核心语句使用省略号代替。随书光盘包含完整代码。源代码的使用请参考附录。

术语中英文对照

表 Q-1 是本书中部分术语的中英文对照说明。

表 Q-1 术语中英文对照表

中 文	英 文	说 明
内容提供者	ContentProvider	内容提供框架中用于提供数据
内容解析器	ContentResolver	内容提供框架中用于读取数据
布局填充器	LayoutInflater	用于填充布局资源
小部件	Widget	不作为容器的显示组件
意向	Intent	用于描述操作的意向信息
动作	Action	用于描述意向发送方的动作
未决意向	PendingIntent	用于描述一个将要执行的动作
消息提示条	Toast	用于快速显示消息的可视条
片段	Fragment	用于表示 Activity 的一部分
插值器	Interpolator	用于计算属性的动画插值

编 者



目 录

前言

第 1 章 初识 Android 平台	1
1.1 Android 平台简介	1
1.1.1 Android 发展历史	1
1.1.2 平台内涵	3
1.2 Android 平台架构	5
1.2.1 架构图	5
1.2.2 架构内容	6
1.2.3 Android 应用程序内容	8
1.3 用户界面	10
1.3.1 视图结构层次	10
1.3.2 布局——设计图	10
1.3.3 视图——整体家居	10
1.3.4 显示部件——装饰品	11
1.3.5 用户界面事件	11
1.3.6 界面风格和主题	11
1.3.7 数据绑定	12
1.4 程序资源和资产	12
1.4.1 程序资源和资产概述	12
1.4.2 资源类型及内容	12
1.5 数据存储	13
1.5.1 首选项	13
1.5.2 文件	13
1.5.3 数据库	13
1.6 平台安全和许可	14
1.6.1 Android 平台安全结构体系	14
1.6.2 应用程序签名	14
1.6.3 用户 ID 和文件存取	14
1.6.4 许可	15
第 2 章 Android 平台开发之旅	16
2.1 搭建系统环境	16
2.2 Android 平台 SDK	17
2.2.1 安装配置 Android 平台 SDK	17
2.2.2 Android SDK 内容	17

2.2.3 Android SDK 附带工具简介	17
2.3 集成开发环境——Eclipse	19
2.4 Android 应用程序开发工具	20
2.4.1 获取 ADT	20
2.4.2 安装配置 ADT	20
2.5 创建 Android 虚拟设备	20
2.6 Android 应用程序开发	
环境验证	21
2.6.1 搭建工程	21
2.6.2 运行工程	25
2.7 应用程序开发过程	26
2.7.1 开发过程回顾	26
2.7.2 代码调试技巧	30
2.8 新手上路	32
第 3 章 Android 应用程序组件	34
3.1 应用程序组件	34
3.2 Android 应用程序组件	34
3.2.1 Activity 组件——形象大使	35
3.2.2 服务组件 (Service) ——	
老黄牛	39
3.2.3 广播接收器组件 (Broadcast	
Receiver) ——倾听者	42
3.2.4 内容提供者组件 (Content	
Provider) ——奉献者	46
3.2.5 Android 应用程序组件小结	50
3.3 组件应用机制	50
3.3.1 组件间的纽带——意向	50
3.3.2 组件间的预约——未决意向	56
3.3.3 与线程的交互——线程消息	
队列处理器	57
3.3.4 与服务组件的交互——AIDL	59
3.3.5 与本地服务组件交互	64

3.3.6 客户端与服务端的桥梁—— 信使	65	4.6.2 对话框 (Dialog)	133
3.4 Activity 组件关联对象	68	4.6.3 消息提示条 (Toast)	137
3.4.1 资源处理相关	69	4.6.4 片段组件 (Fragment)	137
3.4.2 用户界面框架相关	73	4.6.5 拖放操作	145
3.4.3 内容提供相关	74	4.6.6 动画效果	149
3.4.4 管理框架相关	75	4.6.7 定制 Activity 组件	152
3.4.5 环境信息相关	76	第 5 章 底层用户界面设计	154
3.4.6 数据存储相关	77	5.1 Android 底层用户界面	154
3.5 Android 应用程序组件小结	81	5.2 底层视图绘制	154
第 4 章 用户界面设计	82	5.2.1 表面视图 (Surface View)	155
4.1 Android 平台界面组件结构	82	5.2.2 底层视图的绘制接口	159
4.1.1 Android 界面组件结构层次	82	5.3 OpenGL 视图绘制	159
4.1.2 理解 Android 界面组件结构 层次	82	5.3.1 OpenGL ES 概述	159
4.1.3 布局的地位	83	5.3.2 Android 平台对 OpenGL ES 的支持	159
4.2 界面组件使用模式	84	5.3.3 OpenGL 表面视图	159
4.2.1 定义界面组件	84	5.3.4 渲染脚本表面视图	161
4.2.2 生成界面组件资源标识	85	5.3.5 Android 平台中 OpenGL 使用说明	166
4.2.3 组件属性和标识	85	5.4 视频视图 (VideoView)	167
4.2.4 引用界面组件	86	第 6 章 文件管理	168
4.2.5 界面设计器	87	6.1 Android 平台中的文件	168
4.3 布局组件 (Layouts)	87	6.2 资源文件和资产文件	168
4.3.1 线性布局 (Linear Layout)	88	6.2.1 资源文件	168
4.3.2 相对布局 (Relative Layout)	90	6.2.2 资产文件	170
4.3.3 框布局 (Frame Layout)	91	6.3 存储设备文件	171
4.3.4 表格布局 (Table Layout)	92	6.3.1 存储设备文件操作	171
4.3.5 绝对布局 (Absolute Layout)	93	6.3.2 文件浏览器	172
4.3.6 小结——布局的选择	94	6.4 应用程序文件	177
4.4 视图组件 (Views)	94	6.4.1 私有文件	178
4.4.1 视图的使用模式	94	6.4.2 共享首选项文件	179
4.4.2 常用视图	96	6.5 文件系统监视	181
4.4.3 定制视图	126	第 7 章 数据库应用	184
4.5 小部件 (Widgets)	127	7.1 Android 平台数据库应用概述	184
4.5.1 小部件的使用模式	128	7.2 嵌入式数据库 SQLite	185
4.5.2 常用小部件	128	7.2.1 SQLite 数据库介绍	185
4.6 界面框架	129	7.2.2 Android 平台对 SQLite 数据库的支持	186
4.6.1 菜单 (Menu) 和操作栏 (ActionBar)	129	7.3 SQLite 数据库应用模式	186



7.4 内容提供框架	187	9.3.1 Android 平台对蓝牙的支持	268
7.4.1 内容解析端	187	9.3.2 蓝牙通信模式	268
7.4.2 内容提供端	190	9.3.3 蓝牙通信	269
7.4.3 游标加载器应用	193	9.4 近距离通信 (NFC)	277
7.5 SQLite 数据库 API	195	9.4.1 近距离通信概述	277
7.5.1 SQLite 数据库应用	195	9.4.2 Android 平台对近距离通信的支持	278
7.5.2 基于 SQLite 数据库的 日记账工具	198	9.4.3 近距离通信的模式	278
7.6 嵌入式对象数据库 Db4o	215	9.4.4 标签调度系统	279
7.6.1 Db4o 对 Android 平台的支持	215	9.4.5 标签处理	281
7.6.2 Db4o API	215	第 10 章 多媒体应用	284
7.6.3 Db4o 数据库应用	216	10.1 Android 平台对多媒体应用的支持	284
7.6.4 基于 Db4o 数据库的 日记账工具	220	10.2 音频回放与录制	285
7.7 数据库开发小结	225	10.2.1 音频回放	285
第 8 章 网络通信与 Web 开发	226	10.2.2 录制音频	292
8.1 Android 平台网络通信	226	10.2.3 音频管理	299
8.2 Android 平台对网络 通信的支持	226	10.3 视频回放与录制	301
8.3 网络连接管理	227	10.3.1 视频回放 (表面视图)	301
8.3.1 连接管理	227	10.3.2 视频回放 (视频视图)	305
8.3.2 Wi-Fi 连接管理	231	10.3.3 录制视频 (代码控制)	308
8.4 网页浏览器	236	10.3.4 录制视频 (调用系统功能)	315
8.4.1 WebKit 介绍	236	10.4 相机应用	318
8.4.2 Android 平台对 WebKit 引擎的封装	236	10.4.1 拍摄照片 (代码控制)	318
8.4.3 网页视图 (WebView) 应用	237	10.4.2 拍摄照片 (调用系统功能)	321
8.5 浏览器信息管理	250	10.5 媒体信息管理	324
8.5.1 浏览书签信息	250	10.5.1 Android 平台对媒体信息 管理的支持	324
8.5.2 搜索历史记录	253	10.5.2 应用程序 Activity 框架	325
8.5.3 下载管理	254	10.5.3 应用程序配置信息接口	326
第 9 章 无线通信	263	10.5.4 扫描媒体文件	326
9.1 无线通信概述	263	10.5.5 获取媒体文件信息	330
9.2 短消息通信	263	10.6 音乐盒工具	331
9.2.1 Android 平台对短消息的 支持	263	第 11 章 个人信息管理	340
9.2.2 发送短消息	263	11.1 个人信息管理	340
9.2.3 接收短消息	265	11.2 Android 对个人信息管理 的支持	340
9.3 蓝牙通信	268	11.3 Android 平台个人信息 管理	341

11.3.1 管理工具	341	14.2 Android 平台对地图应用的支持	390
11.3.2 应用程序主 Activity 框架	341	14.3 地图视图 (MapView)	391
11.3.3 获取联系人信息	343	14.3.1 地图视图组件的定义	391
11.3.4 获取电话号码	344	14.3.2 获取地图 API 使用密钥	392
11.3.5 获取电子邮箱	346	14.3.3 地图应用工程设置	393
11.3.6 获取公司信息	348	14.3.4 地图应用程序 Activity 组件	394
11.4 Android 平台个人信息关联	349	14.3.5 引用地图库	394
11.4.1 联系数据库	350	14.3.6 地图使用许可	394
11.4.2 联系数据表关联	352	14.4 地图应用	394
第 12 章 电话系统管理	353	14.4.1 地图 Activity 组件框架	396
12.1 电话系统概述	353	14.4.2 获取地图当前位置	398
12.2 Android 平台对电话系统的支持	353	14.4.3 地图视图叠加图管理	399
12.3 电话系统管理	354	14.4.4 地图 API 使用小结	401
12.3.1 获取电话信息	354	14.5 地图定位	402
12.3.2 电话状态	358	14.5.1 位置管理	402
12.3.3 手机网络基站定位	363	14.5.2 手机基站定位	405
12.4 拨号及呼叫日志管理	368	第 15 章 系统信息管理	408
12.4.1 拨号功能	368	15.1 系统服务	408
12.4.2 日志	369	15.1.1 Android 系统服务介绍	408
第 13 章 XML 应用	373	15.1.2 Activity 管理	409
13.1 Android 平台对 XML 应用的支持	373	15.1.3 提醒管理	414
13.2 XML Pull API	373	15.1.4 剪贴板管理	416
13.2.1 Android 平台对 XML Pull API 的支持	373	15.1.5 通知管理	419
13.2.2 XML Pull API 使用模式	373	15.1.6 传感器管理	421
13.2.3 XML Pull API 应用示例	374	15.1.7 振动管理器	422
13.3 XML 资源解析	381	15.1.8 墙纸管理	423
13.3.1 应用程序主 Activity 框架	381	15.1.9 设备管理	424
13.3.2 解析菜单资源	382	15.2 Android 平台系统信息	427
13.3.3 解析 XML 布局资源	383	15.2.1 进程管理	428
13.3.4 解析 XML 资源	385	15.2.2 文件系统统计信息	429
13.3.5 解析 XML 原文件资源	387	15.2.3 环境信息	430
13.4 Android 平台 XML 使用小结	389	15.2.4 时间管理	430
第 14 章 地图应用	390	15.2.5 系统信息	434
14.1 地图概述	390	15.2.6 电池状态	436
		第 16 章 Android 资源及 SDK 工具	439
		16.1 资源类型及定义	439
		16.1.1 常量值资源	439



16.1.2 绘制用资源	441	16.2.3 XML 属性	448
16.1.3 布局资源	442	16.3 系统资源定义	448
16.1.4 动画资源	442	16.4 Android SDK 工具使用	448
16.1.5 菜单资源	444	16.4.1 adb 工具	448
16.1.6 文件资源	444	16.4.2 ddms 工具	449
16.1.7 备选资源	445	16.4.3 sqlite3 工具	450
16.2 资源的使用模式	447	16.4.4 keytool 工具	450
16.2.1 资源 ID	447	附录 随书源代码说明	451
16.2.2 引用资源	447	参考文献	452



第1章 初识 Android 平台

本章首先从多个视角对 Android 平台进行简要的入门介绍，然后详细地介绍 Android 平台的内容，包括其发展历史和内涵，特别是 Android 平台用到的一些开源项目。以此为基础全面介绍 Android 平台的架构组成、应用程序组件和 Android 平台所定义的一些核心概念。此外还将对用户界面、程序资源和字符串、数据存储以及平台安全和许可进行简要说明。

1.1 Android 平台简介

“Android”一词的本义是“机器人”。图 1-1 所示的是 Android 平台的各个版本的 Logo，这些形态各异、憨态可掬的机器人，有如 Android 给人们烙下的开放、灵活、活泼、新颖等印象。Android 的官方网站是 <http://www.android.com/>。



图 1-1 Android 平台的 Logo

作为一个手机平台项目，其丰富内涵和有效的市场运作，让“Android”的含义变得多样化。对于开发者而言，“Android”所指的更多是 Android 平台所提供的技术框架和开发包（SDK），Android 开发者的官方网站是 <http://developer.android.com/>；对于移动设备爱好者，“Android”指的却是时下流行的基于 Android 平台的移动设备。

1.1.1 Android 发展历史

2007 年 11 月，Google 公司宣布其基于 Linux 平台的开源手机操作系统的项目代号为“Android”。

2008 年 3 月，Android SDK 发布，代号为 m5-rc15；8 月，Android 0.9 SDK beta 版本发布，代号为 m5-0.9。

2008 年 9 月 23 日，美国运营商 T-Mobile 在纽约正式发布第一款 Android 手机——T-Mobile G1。该款手机由宏达电（HTC）代工制造，是世界上第一部使用 Android 操作系统的手机，支持 WCDMA 网络，并支持 Wi-Fi；当天，Android 1.0 SDK 发布。

2009 年 2 月，Android 1.1 SDK 发布；5 月，Android 1.5 SDK 发布，代号为 Cupcake；9 月，Android 1.6 SDK 发布，代号为 Donut。



Android 平台开发之旅 第 2 版

2010 年 1 月，Google 公司在美国加利福尼亚州山景城（Mountain View）总部的 Android 发布会上，正式发布了自有品牌手机 Nexus One，该机采用 Android 2.1 操作系统；5 月，Android 2.2 SDK 发布。

2011 年 1 月，第一款采用 Android 3 平台的平板电脑问世，如图 1-2 所示；2 月，Android 3 SDK 发布。

2011 年 10 月，Android 4.0 发布，其 Logo 如图 1-3 所示。



图 1-2 Android 3 平板电脑

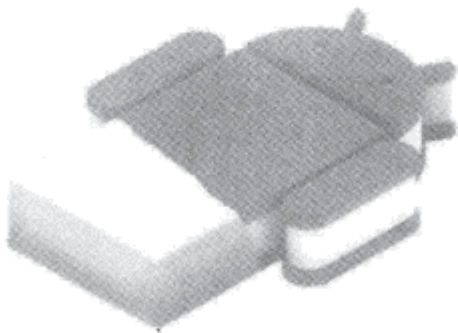


图 1-3 Android 4 平台 Logo

表 1-1 中是 Android 平台的版本发布信息，其中包括 SDK 的所有 API 级别。

表 1-1 Android 平台的版本发布信息

版 本	代 号	发 布 时 间	API 级 别	备 注
4.0.3	Ice Cream Sandwich	2011.12	15	第 1 次修订
4.0,4.0.x		2011.10	14	
3.2	Honeycomb	2011.7	13	
3.1		2011.5	12	
3.0		2011.2	11	
2.3.4	Gingerbread	2011.5	10	
2.3.3		2011.2	10	
2.3		2010.12	9	
2.2.x	Froyo	2010.5	8	第 2 次修订
2.1.x	Éclair	2010.1	7	第 2 次修订
2.0.1		2009.12	6	
2.0		2009.10	5	
1.6	Donut	2009.9	4	第 3 次修订
1.5	Cupcake	2009.4	3	第 4 次修订
1.1	Base	2009.2	2	
1.0		2008.9	1	
0.9	m5-0.9	2008.8		
	m5-rc15	2008.3		
	Android	2007.11		项目启动

从表 1-1 中不难看出，Android 平台一直保持升级的态势：1.x 平台在一年中连续进行 3

次较大的升级；Android 2 平台在一年中更是连续进行 4 次升级；2011 年，Android 平台借助 3.0 版本跨入了面向大屏幕设备应用的新天地；与此同时，基于 2.x 平台的手机设备也如雨后春笋般呈现在消费者面前，当前发布的最新版本为 2.3.7。

提示：根据最新的统计数据，各版本 Android 手机中所占份额较大的分别是 2.3.3 ~ 2.3.7 和 2.2 版本；对于平板电脑，主要还是以 3.x 版本为主。就功能而言，Android 3 平台兼容 2.x 平台并趋于稳定，新推出的 4.0 平台主要致力于平台兼容性及性能的提升。有关 Android 平台版本的统计信息发布网址为

<http://developer.android.com/resources/dashboard/platform-versions.html>

毋庸置疑，Android 平台的开放性决定了其发展不会停滞。

1.1.2 平台内涵

1. Android 平台的功能

(1) 提供应用程序框架 (Framework)，开发者可以遵照这些框架搭建应用程序

读者可以结合 J2SE 平台的 Applet 框架或 J2ME 平台的移动信息设备套件 (Mobile Information Device-let, MIDlet) 框架来理解 Android 平台的应用程序框架。

提示：每个开发者估计都纠结过平台和框架的概念，特别是对新手而言，平台和框架似乎总是前辈们口头上惯用的、玄而又玄的名词。实际上，读者可以把平台理解为舞台，其强调了事物的支持特性，有如舞台具有支撑舞者在其上进行表演的特性。同样，Android 平台具有支持 Android 应用程序运行的特性，具体表现在运行时 (Runtime) 环境和接口 (API)。常见的平台还有 Windows 平台、Linux 平台等。

框架可以理解为骨架，其强调了事物的可重用性。众所周知，人类无论高矮胖瘦、美丑强弱，其骨架都是相似的。反之，使用一个人体的骨架模型，可以塑造出不同的人体模型。同样，在软件开发过程中，使用框架可以开发出界面各异的、某一类应用程序。例如，使用微软公司的 MFC 框架可以快速地开发出一套运行于 Win32 平台的应用程序。框架的具体表现为一组协同工作的类，如界面组件类、事件处理类、网络通信类等。比较流行的框架有 .Net、Qt、MFC、VCL 等，借助这些框架，开发者可以高效地开发出应用程序。

简而言之，框架帮助应用程序的开发，平台支持应用程序的运行，框架建立在平台之上。

(2) 定制的 Dalvik 虚拟机

读者可以结合 J2SE 平台的 Java 虚拟机 (Java Virtual Machine, JVM) 和 J2ME 平台的千字节虚拟机 (Kilo-bytes Virtual Machine, KVM) 来理解 Dalvik 虚拟机。

无论是 JVM 还是 KVM 都是参照 Java 虚拟机的技术规范来进行设计的，而 Dalvik 虚拟机是 DalvikVM 公司 (<http://www.dalvikvm.com>) 开发的，其所遵照的技术规范可能与一般意义上的 Java 虚拟机不同。

Dalvik 虚拟机所支持的字节码 (Byte Code) 是 “dex” 文件 (Dalvik Executable)，也就是说 Dalvik 不支持通常的 Java 类文件 (class 文件) 字节码。



Android 平台开发之旅 第2版

(3) 集成了基于 WebKit 开源项目的浏览器

WebKit 是一个开源项目 (<http://webkit.org>), 其主要由 K 桌面环境 (K Desktop Environment, KDE) 的 KHTML 修改而来, 并且包含了一些来自苹果公司的一些组件。

传统上, WebKit 包含一个网页引擎 WebCore 和一个脚本引擎 JavaScriptCore, 它们分别对应的是 KDE 的 KHTML 和 KJS。不过, 随着 JavaScript 引擎的独立性越来越强, 现在 WebKit 和 WebCore 已经基本上混用不分。

Google 公司开发的网页浏览器产品 Google Chrome 就是基于 WebKit 开源代码, 并自行开发出称为“V8”的高性能 JavaScript 引擎。读者可以将 Android 平台的浏览器视为 Chrome 的移动设备版本。

(4) 2D 和 3D 图形引擎

2D 图形引擎基于 SGL; 3D 图形引擎基于 OpenGL ES 1.0 规范。

Skia 图形库 (Skia Graphics Library, SGL) 是一套用于绘制文本、几何图形和图片的完整的 2D 图形库。

OpenGL ES 1.0 是基于 OpenGL 1.3 规范来定义的, 同时增强了软件渲染和基本的硬件加速功能。读者可以从 <http://www.khronos.org/opengles/spec/> 获取 OpenGL ES 1.0 的规范。

(5) 提供 SQLite 数据库用于结构化数据存储

SQLite 是一个能够嵌入到进程内部的库, 同时它也是一个实现了独立性、无需服务器、零配置和事务处理的 SQL 数据库引擎。SQLite 官方网站为 <http://www.sqlite.org/>。

读者也可以把 SQLite 理解为一个嵌入式 SQL 数据库引擎, 其无需单独的服务器进程, 开发库小巧、可靠, 支持大多数的系统平台, 如 Linux、Mac OS X、Windows。

(6) 提供对音频、视频和图片等媒体的支持

Android 平台使用 PacketVideo 公司的移动多媒体框架 OpenCore 来支持各种媒体服务。PacketVideo 公司的官方网站是 <http://www.packetvideo.com/>。

在 PacketVideo 公司的官网上有对 OpenCore 框架在 Android 平台上的应用介绍, 有兴趣的读者可以参阅页面 <http://www.packetvideo.com/products/android/>。

Android 平台支持的媒体类型有 MPEG4 (mp4)、H.264、MP3、AAC、AMR、JPG、PNG、GIF 等。

(7) 提供 GSM 电话控制

(8) 支持蓝牙、EDGE、3G 和 Wi-Fi

增强型数据速率 GSM 演进 (Enhanced Data Rate for GSM Evolution, EDGE) 是一种 GSM 到 3G 的过渡技术。

(9) 支持摄像头、GPS、罗盘和加速计等设备

2. 与 Linux 平台的渊源

Android 平台是基于 Linux 2.6.25 版本的内核进行改造的。Linux 内核包括系统调用接口、进程管理、内存管理、虚拟文件系统、网络协议栈、设备驱动程序和架构, 如图 1-4 所示。

也就是说 Android 平台内核也会包含图 1-4 中这些内容。略有不同的是, Android 的目标环境是 ARM 平台, 而不是通常的 x86 平台。

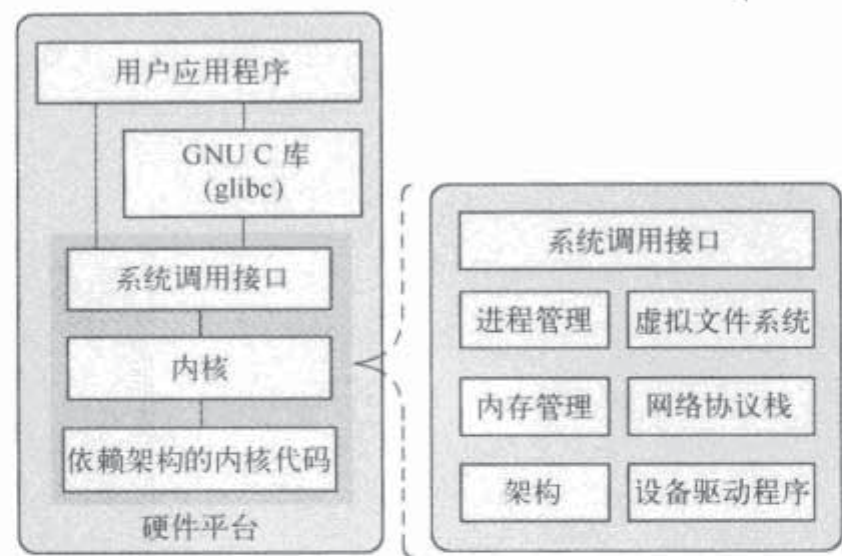


图 1-4 Linux 内核体系结构图

1.2 Android 平台架构

1.2.1 架构图

图 1-5 所示为 Android 平台架构图，按依赖关系层次，依次为 Linux 内核、支持库、Android 运行时、应用程序框架和应用程序。



图 1-5 Android 平台架构图



1.2.2 架构内容

1. 应用程序

包括 Android 平台配置的一套应用程序集，如短信程序、日历工具、地图浏览器、网页浏览器等工具，以及用户基于 Android 平台的应用程序框架，使用 Java 语言自行开发的程序。图 1-6 所示为 Android 平台的应用程序管理的实机界面。

2. 应用程序框架

开发者可以完全使用与那些内核应用程序相同的框架，这些框架用于简化和重用应用程序的组件。如果程序能够“暴露”其内容（如数据、功能模块），则其他的程序即可使用这些内容。

通过应用程序框架，用户自定义的程序可以执行框架的预设功能，这样可以极大减少用户程序的代码量。例如代码 1-1，用户 Activity 组件（HelloAndroidAct）在其重载的创建回调方法（onCreate）中使用 super 对象调用父类（Activity）的方法（onCreate），该回调方法在 Android 平台的应用程序框架中预定义。

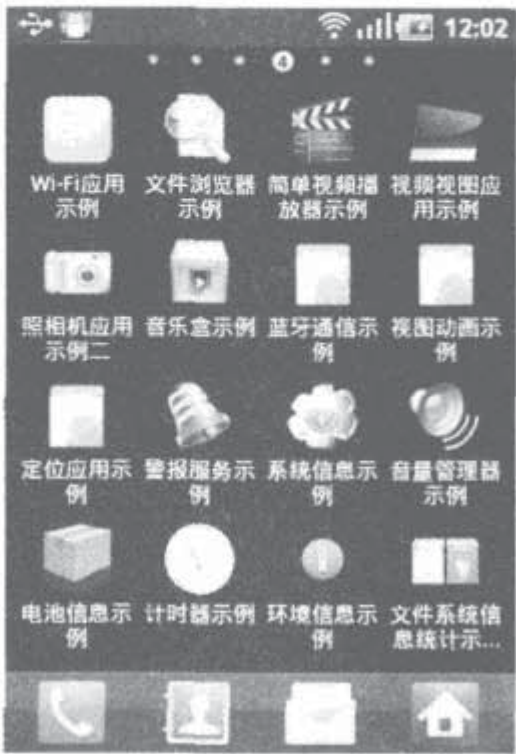


图 1-6 Android 应用程序

代码 1-1 Activity 组件的框架定义

文件名: HelloAndroidAct.java

```
1      public class HelloAndroidAct extends Activity {
2          @Override
3          public void onCreate(Bundle savedInstanceState) {
4              super.onCreate(savedInstanceState);    //调用父类的创建方法
5              setContentView(R.layout.main);      //设置内容视图
6          }
7      };
```

3. 系统开发库

Android 定义了一套 C/C++ 开发库供 Android 平台的其他组件使用。这些功能通过 Android 应用程序框架提供给开发者，开发者不能直接使用这些库。这些库包括以下几种。

- (1) 系统 C 开发库
源于 BSD 的标准 C 系统库（libc）。
- (2) 媒体开发库
基于 PacketVideo 公司的 OpenCore。
- (3) 屏幕管理库
管理对显示子系统的访问或无缝衔接多个应用程序的 2D 和 3D 图形层。
- (4) 网页浏览器引擎核心库
- (5) SGL 库——2D 图形引擎库

- (6) 基于 OpenGL ES 1.0 API 的 3D 开发库
- (7) 基于开源项目 FreeType 的字体引擎开发库

FreeType (<http://www.freetype.org>) 是一款免费的、高质量的、可移植的字体引擎，其设计小巧、高效、高度可定制，并且可以产生可移植的高品质输出内容（符号图像）。该引擎广泛用于图形库、展示服务、字体转换工具、文本图片生成工具和很多其他产品。

- (8) SQLite 开发库

提示：2009 年 6 月，Android 开发者网站 (<http://developer.android.com>) 发布了 Android 本机开发包（Android Native Development Kit, NDK），其核心内容包括一套本地系统的头文件（.h 文件）和库（.lib 文件）。尽管如此，Android 系统也不允许一个纯 C/C++ 的程序运行，而必须使用 Java 本机接口（Java Native Interface, JNI）方式来执行本地代码。

4. 运行时环境

Android 平台包括一套核心库和 Dalvik 虚拟机，该核心库提供了 Java 开发库的大多数功能。读者如果了解 JNI 技术，就很容易理解这个 Java 开发库实际上是对 Android 平台提供的 C/C++ 开发库进行的本地调用。

Dalvik 虚拟机用于执行 Android 应用程序。每一个 Android 应用程序都在它自己的进程中运行，每一个进程都拥有一个独立的 Dalvik 虚拟机实例，如图 1-7 所示。



图 1-7 Android 进程空间示意图

Dalvik 被设计成可以在一个设备上同时高效地运行多个实例的虚拟系统，Dalvik 虚拟机基于寄存器（Java 虚拟机基于栈，详情可参考 Java 虚拟机规范），所以其性能较 Java 虚拟机有较大提升。

Android 应用程序中的 Java 文件都需要先经过 Java 编译器编译成类文件，然后通过 SDK 中的“dex 文件转换”工具（参考第 2 章）转化成“dex 格式”的字节码文件，再由 Dalvik 虚拟机加载执行，如图 1-8 所示。



图 1-8 源文件编译运行示意图



Dalvik 虚拟机依赖于 Linux 内核的一些功能，如线程和底层内存管理机制。

5. Linux 内核

Android 的核心系统服务依赖于 Linux 2.6 内核，如安全性、内存管理、进程管理、网络协议栈和驱动模型。Linux 内核同时也是硬件和软件栈之间的抽象层。

1.2.3 Android 应用程序内容

1. Android 应用程序

Android 应用程序由 Java 语言编写，通过打包工具（第 2 章）将应用程序的字节码文件、资源文件和清单文件打包到一个以“apk”为扩展名的包文件中，如图 1-9 所示。该文件作为分发应用程序的载体，被应用程序安装到移动设备上。每个“apk”文件中的代码可以视为一个应用程序。

以下是 Android 应用程序进程的设计规则。

1) 默认的，每个应用程序运行在它自己的 Linux 进程空间。在需要执行该应用程序的时候 Android 将启动该进程，当不再需要该应用程序，并且系统资源不够分配时，系统将终止该进程。

2) 每个进程都有自己的 Java 虚拟机（Dalvik 虚拟机），所以任何应用程序的代码与其他的应用程序的代码是相互隔离的。

3) 默认的，每个应用程序被分配给一个唯一的用户 ID。所以任何应用程序的文件只能对该应用程序可见。

2. 应用程序组件

Android 平台的一个核心要点是一个应用程序能够利用其他应用程序的组件。例如，一个程序 A 用于显示指定文件夹中的全部数据库名，而另外一个程序 B 用于查看某一指定数据库的信息，如数据表名、数据表模式（Schema）、数据表内容等。在程序 A 中，当用户单击列表中的某一数据库名称项时，可以调用程序 B 中的模块去显示指定数据库的信息，而无需重复开发。

而为了做到这一点，系统必须能够在应用程序需要调用指定模块的时候找到并启动包含该模块的组件。因此，不像大多数系统的应用程序，Android 应用程序没有 main 方法，代码框架也必须遵照 Android 平台所定义的形式。所以 Android 应用程序需要包含系统能够识别并调用的一些基本组件。

(1) Activity 组件

Activity 表现为用于与用户进行交互的可视化界面。例如，一个 Activity 可以是系统登录界面（见图 1-10）；另外一个 Activity 可以是显示已登录用户信息的列表。

用户定义的每一个 Activity 都继承于父类：Activity（如代码 1-1 所示）。一个应用程序可能由一个或多个 Activity 组成，Android 平台通过 Activity 栈来对 Activity 进行管理。

每一个 Activity 都被分配一个用于绘制的窗体。一般来说，该窗体是全屏幕的，但也可能比全屏幕要小且浮于其他窗体之上，如弹出式对话框，如图 1-11 所示。

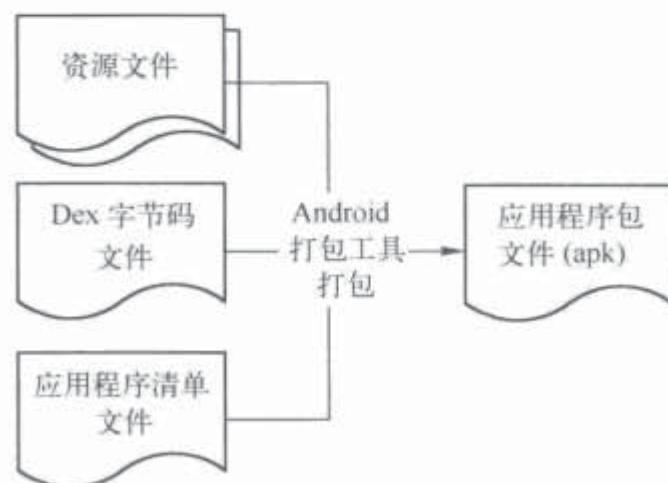


图 1-9 Android 应用程序打包

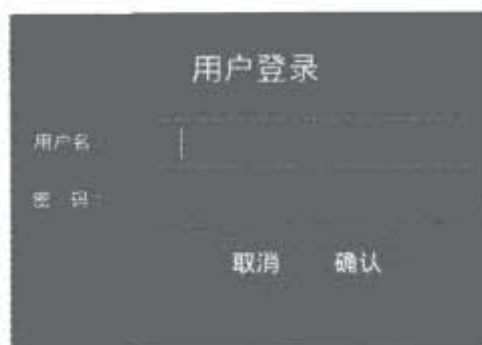


图 1-10 Activity 界面



图 1-11 弹出式对话框

窗体的可视内容由一组视图层次结构（见图 1-12）来提供，这些视图元素都继承于视图（View）类，每个视图元素控制窗体内一个常规的矩形框区域，父视图包含并组织其子视图的布局。

（2）Service（服务）

服务是一类无需可视用户界面，更适合在后台长期运行的应用程序，如背景音乐播放器或后台数据处理服务等。

同 Activity 一样，用户定义的每一个服务都继承于父类：Service。该父类由 Android 平台框架预定义。

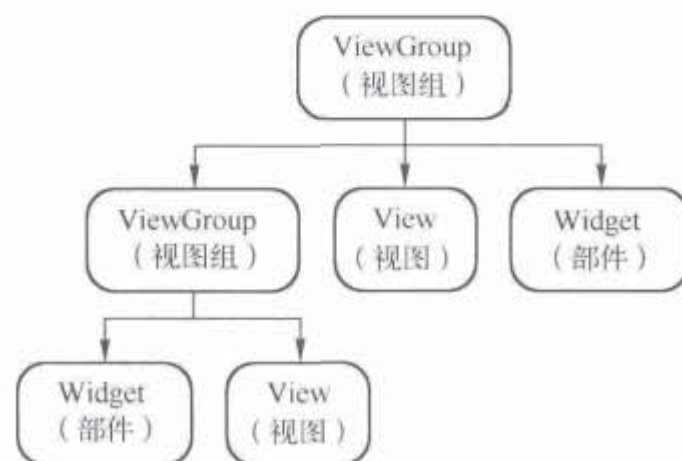


图 1-12 Android 界面元素的视图层次结构

（3）BroadcastReceiver（广播接收器）

广播接收器是一类只接受和处理广播消息的组件。广播接收器也没有显示用户界面，但是可以在响应其接受信息时启动一个 Activity，或者通过通知管理器来显示提示界面从而警示用户。

一个应用程序可能有任意数量的广播接收器来响应任何它认为重要的通告。所有用户定义的广播接收器都继承于父类：BroadcastReceiver。该父类由 Android 平台框架预定义。

（4）ContentProvider（内容提供者）

内容提供者可以将指定的一组应用程序数据让其他应用程序使用。这些数据可以存储于文件系统或者 SQLite 数据库。

用户定义的内容提供者都继承于父类：ContentProvider，并实现一套标准的方法来允许其他应用程序来获取并存储其所控制的数据类型。该父类由 Android 平台框架预定义。

一个内容提供者可以和其他内容提供者进行交互，甚至和其他内容提供者协作来管理任何进程内的通信。

3. 激活应用程序组件

通过前一节的介绍，相信读者对 Android 程序的主要组成部分应该有所认识。但是组件的调用或者组件与组件之间的切换又是如何实现的呢？例如，应用程序启动时如何调用指定的 Activity 或者服务，Activity 之间如何进行切换调用？

Android 平台定义了一种称为 Intent（意向）的异步消息，该消息可以用于激活 Activity、服务和广播接收器组件。一个意向是一个包含消息内容的对象，对于 Activity 和服务而言，意向就是以 Activity 或服务的名字作为执行请求并且指明其所要执行的数据的 URI



的组合消息。例如，请求一个名为“图片浏览器”的 Activity 和一个指定的文件夹（URI），其意向是启动该图片浏览器程序去执行显示指定文件夹中的图片的任务。

1.3 用户界面

应用程序的关键组件 Activity，表现为一个用户关注的、并用于执行用户行为的可视化界面。系统分配给 Activity 一个默认的窗体用来绘制界面，而该窗体中的内容是一套视图层次结构。下面将对用户界面的视图层次结构以及其组成进行介绍。

1.3.1 视图结构层次

图 1-12 所示是 Android 平台视图层次结构树图。结构树的根节点通常是一个视图组对象（ViewGroup），而根节点的子节点既可以是视图对象（View），也可以是视图组对象。

实际上，视图组对象也是继承于视图类，之所以要分开来定义，主要是为了开发者能够清晰地区分作为容器的视图（视图组）和作为显示的视图（普通可视视图）。

通过图 1-12 中的视图层次结构树，开发者可以任意设计简单或复杂的用户界面。既可以使用 Android 平台预定义的显示部件，也可以使用用户自定义的视图组件。

Activity 组件通过设置其内容视图的方法来装载并显示视图层次结构，并最终在设备屏幕上显示其界面内容。

1.3.2 布局——设计图

布局对象是用于指明可视组件的布置方式，如图 1-13 所示。



图 1-13 界面元素的布局

布局对象本身是不可见的，而是“隐含”于布局的结果之中。

Android 平台中布局的定义与 J2SE 平台是一致的，都是用来决定界面容器中所包含的可视组件的摆放，都是不可见的。但是不同的是，在 Android 平台，布局是作为视图组对象来定义的，而不像 J2SE 平台是作为显示容器的属性来设置；在用户界面设计中，Android 平台的布局往往需要显式地定义，而不像 J2SE 平台，不定义布局时显示容器将选用系统默认的布局。

读者可以联想到，布局就如同设计师最终绘制出的设计图。

1.3.3 视图——整体家居

为了区别布局和显示部件，下面将视图对象单独来介绍。如果说布局是从整体来设计用户界面，那么视图可以理解为集成度比较高的、内容范围介于全局和局部之间的组件。通过这些组件，开发者可以很快搭建某些类型的应用程序。例如，网页浏览器程序、视频播放器

程序，这些应用程序都用到了 Android 平台预定义的视图组件。

就像整体家具一样，出厂之前就已经由厂家进行了总体的设计和整合，除了提供产品的特定功能之外，在整体风格上的统一，更是能够达到良好的视觉效果。这种整体家具一般要比单独购置各个部件省时省力，而且能够保证风格的统一。例如，直接使用 Android 平台的网页浏览器视图要比开发者使用多个显示部件来自行设计浏览器界面要方便、美观得多。

1.3.4 显示部件——装饰品

显示部件（Widget）也是一个视图对象，主要提供与用户的交互界面。Android 平台预定义的显示部件中，简单的有按钮、文本框、复选框等；复杂的有日期选择面板、缩放控制面板等。

显示部件在用户界面中的视野应该是局部，是视图组件的最基本单元。开发者也可以自行设计显示部件。

显示部件与视图的关系，就像一只漂亮的花瓶加入到展示柜中，起到装饰的作用。

1.3.5 用户界面事件

通过前面几节对组成用户界面的视图层次结构的介绍，相信读者对 Android 程序的用户界面设计有了一个大致的框架性认识。和其他的开发平台一样，用户界面作为用户和程序沟通的桥梁，应用程序将通过界面组件来收集用户的请求事件，如单击按钮、触摸图片、改变列表内选择项等。

Android 平台通过两种方式来获取用户界面的请求事件。

1. 事件侦听器（Listener）

定义某一类事件的侦听器，然后将其设置给指定的组件。例如，定义了一个单击事件的侦听器对象，然后将该侦听器作为某一按钮的侦听器。这样发生在该按钮上的单击事件都将被该侦听器对象获取并处理。

该处理方式“即定义即用”，快捷直观，比较适用于简单的场合，如果要侦听的界面元素较多，则将会影响对事件处理的统一管理。

2. 事件回调方法

与使用侦听器绑定的方式不同，重载界面组件的事件回调方法的方式适用于父界面内的子组件。例如，一个父界面组件包含 4 个按钮组件，这 4 个按钮组件的单击事件的处理都设置为父界面的单击事件回调方法。在父界面的单击事件回调方法中，通过目标组件的 ID 来识别子按钮，并进行相应的事件处理。

这种方式将组件的事件处理进行统一的管理，结构清晰，但是在消息的分发方面需要注意。

1.3.6 界面风格和主题

虽然 Android 平台提供了标准的可视界面组件，但是某种情形下，界面风格还是无法满足用户需求。就像 HTML 规范中虽然定义了 H1 标记，但是用户为了突出页面的风格，使用样式表对 H1 标记的默认样式进行修改，使其变得更加有个性。

Android 平台所定义的样式，也是指一套或多套、应用于单个组件的格式属性。样式面



向的是单个的组件，而主题面向的是整个程序或某一个 Activity 的样式。就像我们常用的应用程序中的换肤功能，整体地控制整个系统的风格。

Android 平台提供了一些常用的默认风格和主题资源，用户也可以方便地定义自己喜欢的风格和主题。

1.3.7 数据绑定

从用户界面的交互到用户事件的响应，其背后的主导还是数据。例如，某一产品信息查询系统，通过网络或者访问本地数据库，获取所有产品的信息列表，使用列表组件来装载这些信息列表，用户通过浏览列表组件来查看产品的最新信息。

在一般的设计中，用户界面只负责数据的显示和事件的响应，而对于数据源的获取、编辑等处理都作为数据模型进行管理。这就是常说的 MVC（模型-视图-控制）模式。在 J2SE 平台中，就使用了模型、适配器等概念来对可视组件的数据模型进行管理。

在 Android 平台中，使用 and J2SE 平台同样的适配器的概念来对数据模型进行定义。

1.4 程序资源和资产

俗话说“巧妇难为无米之炊”，没有原料任何工作都没法开展，对于应用程序开发更是如此。同样是手机平台，有过 J2ME 平台游戏开发经验的读者应该都知道，编写一个游戏，除了代码之外，还需要很多额外的素材。例如，众多的人物、场景图像以及一些参数文件、音乐文件等，都是开发程序必备的“原料”。

Android 平台对这些“原料”的划分更为细致，总体上定义为两类：资源（Resource）和资产（Asset）。

资源和资产都是 Android 应用程序不可分割的一部分，都将和代码一样被打包合成到一个应用程序文件（扩展名为 apk）中。

1.4.1 程序资源和资产概述

Android 平台所定义的资源 and 资产是指需要包含并且在应用程序中参考的外部元素，如图片、音频、视频、XML 文件、数据文件等。每一个 Android 应用程序都包含一个资源文件夹（res/）和资产文件夹（assets/）。资源文件夹中一般存放的是 Android 平台可以识别的文件，如图片、XML 文件、音频、视频等。资产文件夹存放的是用户自定义的数据文件或者 Android 平台无法识别的文件，如用户自定义的配置文件，或 Android 平台不支持的音频或视频文件等。

从本质上来讲，Android 平台定义的资产 and 资源没有区别。例如，图片既可以放在资源文件夹也可以放在资产文件夹，虽然都可以存取该图片，但是存放于不同的文件夹其访问方式是不一致的。资源文件夹中的内容会经过 Android 平台的编译，通过其资源 ID 即可引用；而资产文件夹中的内容是无法方便地通过资源 ID 来引用的。

1.4.2 资源类型及内容

表 1-2 列举了 Android 平台常见的资源类型。

表 1-2 Android 平台常见的资源类型

序 号	资 源 类 型	内 容
1	常量值	<ul style="list-style-type: none">• 颜色值• 字符串和样式文本• 大小值• 数组
2	绘制用	<ul style="list-style-type: none">• 图片文件（JPG、PNG、GIF）
3	动画	XML 文件
4	菜单	<ul style="list-style-type: none">• 上下文菜单• 可选菜单
5	布局	XML 文件
6	样式和主题	XML 文件

从表 1-2 读者不难看出，Android 平台对于资源的定义范围是相当广泛的。这样，不仅可以方便地在代码中引用资源，甚至可以在资源的定义中引用有关的资源。Android 平台对资源的这种灵活的管理机制，不仅简化了资源的定义过程，而且有助于将创建资源的代码转移到资源的设计过程中，这对清晰代码结构和提高生产效率有很好的促进作用。

1.5 数据存储

无论是在桌面平台还是移动平台，应用程序都需要持久存储其数据，所以平台都提供了相应的数据存储机制。例如，Windows 平台提供了文件系统和基于文件系统的数据库管理系统用于持久存储用户数据；J2ME 平台提供了记录管理系统（Record Management System，RMS）机制来存储用户的记录数据。

Android 平台主要提供了 3 种数据存储方式：首选项、文件和数据库。

1.5.1 首选项

首选项是一种轻量级的、用于存储或获取简单数据类型的“键—值”项的机制。其典型的用法是存储应用程序的首选项，这些选项将在应用程序启动的时候被载入，大多数应用程序都提供了首选项设置的功能。

特别的，在 Android 平台，不能跨应用程序来共享首选项设置，除非显式地使用 Android 平台的内容提供机制。

1.5.2 文件

和桌面平台一样，Android 平台允许应用程序在移动设备或者移动存储设备上直接存储文件。不同的是，某一应用程序所存储的文件是不能被其他应用程序访问的。

1.5.3 数据库

数据库机制实际上也可以视为文件方式，Android 平台提供了创建和使用 SQLite 数据库的 API。SQLite 数据库是一款小巧、高效的嵌入式数据库，使用前景相当出色。

与文件存取机制一样，每个数据库是其创建程序私有的，并不像普通桌面平台，数据库



系统本身一般都是共享的，数据的访问权限才是通过数据库管理系统来管理的。

1.6 平台安全和许可

通过之前对 Android 应用程序组件和数据存储的介绍，相信读者应该已经“觉察”到 Android 平台与一般的桌面平台存在一定的区别，特别是在数据或文件共享方面。以下将对 Android 平台的安全机制进行简要的说明。

1.6.1 Android 平台安全结构体系

在 Android 平台的安全结构体系设计中是不包括应用程序的。默认的，在与其他应用程序或操作系统以及用户不发生冲突的前提下，应用程序可以执行任何操作。这些操作包括读写用户的私有数据、读写其他应用程序的文件等。

每个应用程序的进程可视为一个安全的沙盒（Sandbox），一般情况下，应用程序不会影响其他应用程序，但是当基本沙盒无法满足该应用程序的特定要求时，该应用程序会显式地要求使用某些系统功能，这种情形下可能会对系统应用程序产生一定的影响。

这些使用许可请求可以通过多种渠道进行处理，典型的是基于证书的自动许可，或通过用户界面提示来让用户选择允许或禁止该请求。例如，程序通过提示界面来供用户选择是否启动蓝牙功能。这些使用许可的请求必须在应用程序中静态地声明，这样 Android 平台才可以在安装应用程序之前预先知道，并且之后也不会改变这些许可声明。

提示：应用程序的使用许可（如访问互联网、读取联系信息、允许蓝牙功能等）都必须预先在应用程序清单文件（“AndroidManifest.xml”）中声明。

1.6.2 应用程序签名

所有的应用程序（apk 文件）必须使用那些私钥被开发者所持有的证书进行签名，这些证书标识了应用程序的作者。此外，这些证书不需要额外的认证授权来进行签名，因为对于 Android 应用程序，使用自签名的证书。这些证书只是用于建立应用程序之间的可信任关系，并不控制一个应用程序是否能够被安装。

1.6.3 用户 ID 和文件存取

在设备上的每个 Android 包文件（apk 文件）都被分配给一个唯一的 Linux 用户 ID，Android 平台为该应用程序创建一个沙盒并防止它接触其他应用程序，同时也防止其他应用程序接触它自身。用户 ID 在该应用程序被安装到设备上时进行分配，并且在该程序的使用期间内，该 ID 持久保证不变。

因为安全策略被强制应用在进程内部，所以任何两个不同用户 ID 的包中的代码都不能在同一个进程空间中正常地运行，这是由于它们属于不同的 Linux 用户。开发者可以改变应用程序的共享用户 ID 的属性来分配给不同的包以同样的用户 ID，这样才可以实现两个包在相同的进程空间中共存，并且具有同样的文件许可。这种方式是安全的，只不过是两个应用程序使用了相同的用户 ID 进行签名。

被一个应用程序存储的任何数据都可以被该应用程序的用户进行签名，并且通常不能被其他的包访问。当创建应用程序的参数文件或数据库时，可以使用读写标识来允许其他包读写该文件。当设置这些标识后，该文件还是被创建自身的应用程序所拥有，但是其全局的读取许可已经被设置，所以对于其他应用程序也可见。

1.6.4 许可

Android 平台定义了两大类许可：使用许可和强制许可。

1. 使用许可

作为一个基本的 Android 应用程序，它没有获取任何的许可，这也意味着它不能做任何对用户体验有反作用的事情或存取设备上的任何数据。当一个应用被安装时，其请求的许可将由包安装管理器来允许。为了使用设备上的受保护功能，开发者必须在 Android 应用程序的清单文件上包含一个或多个使用许可的标签（<uses-permission>）来声明该应用程序需要的许可。

Android 平台提供的所有许可都定义在许可包（Manifest.permission）中。任何应用程序也可以定义和强制它自己的许可。

2. 强制许可

为了强制自定义的许可，开发者必须使用一个或多个许可标签在应用程序清单文件上首先声明这些许可。

3. URI 许可

URI 许可通常与内容提供者一起使用。当一个内容提供者的直接客户端需要将指定 URI 交给其他应用程序来处理时，该内容提供者可能需要使用读写许可来保护自己。一个典型的例子是调用其他程序打开邮件程序的附件：该邮件的附件必须被许可所保护，因为这是敏感的用户数据。所以，如果将一个作为附件的压缩文件的 URI 提交给一个解压工具，那么该工具必须具有打开 URI 所描述的压缩文件的许可，否则解压工具将无法打开该附件。

第2章 Android 平台开发之旅

本章将带领读者开始 Android 平台开发之旅，从搭建系统环境、安装平台 SDK、安装 IDE 以及调试工具到示例工程的创建、运行和调试，通过详细的过程说明和经验提示，让读者能够快速对 Android 平台开发形成初步的理解，为后续的开发流程和常见问题介绍起到良好的过渡作用。

2.1 搭建系统环境

因为 Android 平台的应用程序是用 Java 语言编写的，所以最基本的还是需要 J2SE 平台提供的 Java 编译工具以及运行时环境（Java Runtime Environment, JRE）。本书中 J2SE 平台所用到的是 Java 开发工具包（Java Development Kit, JDK），版本是 1.6。

1. JDK 下载选项

甲骨文公司的下载中心 <http://www.oracle.com/technetwork/java/javase/downloads/> 提供了最新版本 JDK 的下载信息。读者下载时需要根据目标平台选择合适的版本，J2SE 平台的 JDK 提供了 Linux、Solaris 和 Windows 多个平台的版本。

注意：即使对于同样的操作系统，也可能因为 CPU 的类型、架构和字长不同而要选择不同的 JDK 版本。

2. JDK 安装配置

安装 JDK 的过程相信读者都比较熟悉，不再赘述。JDK 安装完毕，需要手动设置相关的环境变量。下面以 Windows 平台和 Linux 平台为例进行介绍。

(1) Windows 平台

1) 需要将 JDK 安装文件夹下的 bin 和 lib 子文件夹路径添加到系统的路径环境变量“PATH”中。

2) 创建“JAVA_HOME”环境变量，并将 JDK 的安装文件夹设置给该变量。

JDK 安装文件夹为“D:\J2SDK”，所以设置“JAVA_HOME”变量为“D:\J2SDK”，添加“PATH”变量的设置“D:\J2SDK\bin 和 D:\J2SDK\lib”（中间用英文分号分隔）。

(2) Linux 平台

1) 需要将 JDK 安装文件夹下的 bin 和 lib 子文件夹路径添加到系统的路径环境变量“PATH”中。

2) 创建“JAVA_HOME”环境变量，并将 JDK 的安装文件夹设置给该变量。

在 Ubuntu 平台下，可通过修改.bashrc 文件来实现“PATH”和“JAVA_HOME”环境变量的修改和设置。

3. 验证 JRE 环境

JDK 安装完毕，可以通过命令行执行 `java-version` 查看 JDK 的版本信息。

2.2 Android 平台 SDK

既然是开发 Android 平台的应用程序，所以仅有 JDK 是不够的，开发者还必须配置 Android 平台的 SDK。

Android 开发者网站提供了有关 Android SDK 资源的最新的下载信息，该网站是 Android 平台开发资源的官方网站。

2.2.1 安装配置 Android 平台 SDK

通过页面 <http://developer.android.com/sdk/index.html>，读者可以获取 Android SDK 管理器 (SDK Manager) 的下载信息。

Android SDK 管理器提供了 3 个平台的版本：Windows、Linux (i386) 和 Mac OS X (Intel)，读者可以根据目标平台选择相应的 SDK 进行下载。

Android SDK 管理器资源实际上是一个打包文件，其中 Android SDK 管理器工具 (SDK Manager.exe) 用于获取 SDK 的最新信息，并管理 SDK 的下载、安装和卸载。

2.2.2 Android SDK 内容

经过一定时间的等待，SDK 会安装完毕。读者可能会发现，Android SDK 文件夹中有非常丰富的开发资源，包括完备的开发参考和辅助工具。

其中主要的内容有：

- 1) docs 文件夹中包含的是完整的 Android SDK 参考文档，包括 SDK 发布历史信息（主要包括该版本的亮点和与上一版本的 API 差异报告）、开发引导和 API 参考。
- 2) tools 文件夹中存放的是 SDK 附带的 Android 平台的公用工具，如 Android 模拟器、SQLite 数据库工具等。
- 3) platform-tools 文件夹中存放的是与平台有关的工具，如调试桥接工具、Dalvik 字节码转换工具、Android 资源打包工具等。
- 4) samples 文件夹中存放的是相关的开发示例，初级开发者可以通过这些示例代码来更好地理解 Android 应用程序的实现过程。

提示：之所以对 SDK 文件夹目录进行详细介绍，目的只有一个：希望读者能够发现并用好已经拥有的资源，而不必舍近求远。当然，SDK 中提供的资源都是英文版本，而且有些概念也是非常“多元化”，对初学者来说可能会造成混淆。但是，这些资源是唯一官方发布的内容，网络中大多数中文资料都是以这些英文版为基础编写的。

2.2.3 Android SDK 附带工具简介

1. Android 模拟器——emulator

Android 模拟器用于模拟 Android 实机环境，是调试 Android 应用程序必不可少的工



Android 平台开发之旅 第2版

具。该工具一般由集成开发环境调用，也可以通过命令行启动。图 2-1 是 Android 模拟器的运行界面。

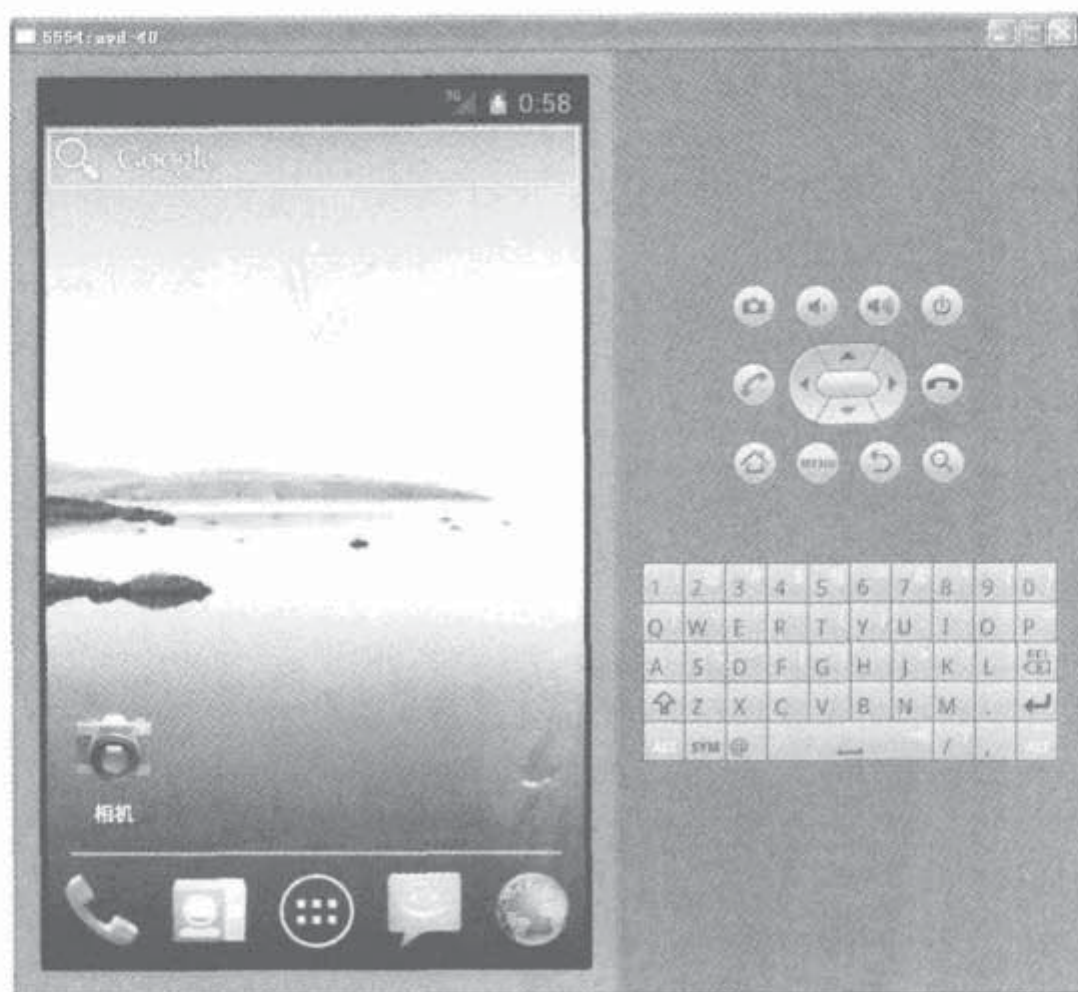


图 2-1 Android 模拟器的运行界面

2. Android 调试桥工具——adb

Android 调试桥（Android Debug Bridge, ADB）工具主要起开发调试的桥梁作用。例如，开发者可以通过调试桥工具访问模拟器或实机，向模拟器或实机上传文件，从模拟器或实机上下载文件等。

有关 adb 工具的使用参见第 16 章。

3. Dalvik 调试监视服务工具——ddms

Dalvik 调试监视服务（Dalvik Debug Monitor Server, DDMS）工具整合了 Android 平台的虚拟机，所以可以获取更多的应用程序运行信息，包括堆信息、线程信息、进程信息等。简而言之，该工具可以提供更加底层和完整的调试信息。

有关 ddms 工具的使用参见第 16 章。

4. SQLite 数据库工具——sqlite3

sqlite3 是支持第 3 版的 SQLite 数据库管理工具，其使用参见第 16 章。

5. 资产打包工具——aapt

Android 资产打包工具（Android Assets Packaging Tool, AAPT）用于将程序的字节码文件、资源文件和程序清单经过压缩和编码，合并到工程文件（apk 文件）中，如图 2-2 所示。

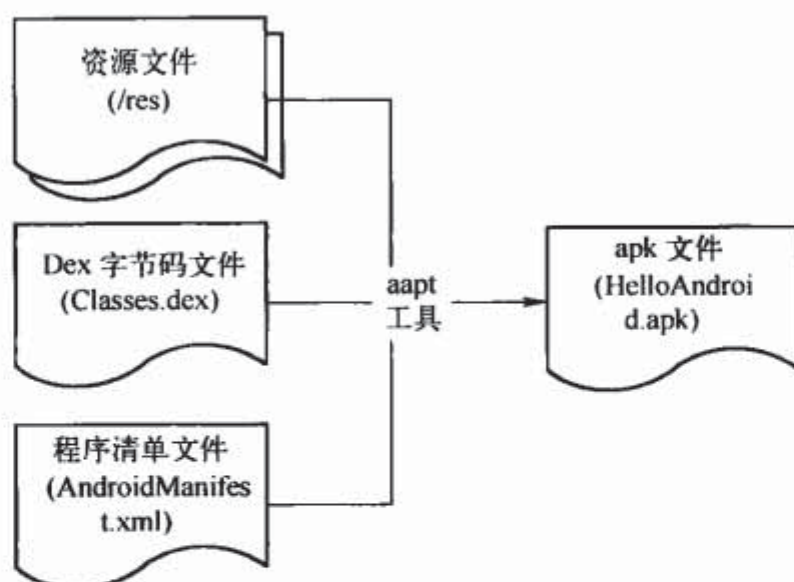


图 2-2 Android 应用程序打包

6. dex 文件转换工具——dx

Dalvik 虚拟机支持的字节码是一种 dex 文件，而通过 Java 编译器对源文件编译出来的却是通常的 Java 字节码文件（class 文件），那么这里就存在将 Java 字节码文件转换为 Dalvik 字节码文件的环节，如图 2-3 所示。



图 2-3 Dalvik 虚拟机字节码生成过程

dx 工具就是用于将一组 class 文件转换为 dex 文件，输出的文件的扩展名必须为 dex、jar、zip 或 apk 之一。

2.3 集成开发环境——Eclipse

看过前面介绍的 SDK 附带工具，读者心里可能还是有点发憊，要使用这些工具来创建、调试和编译 Android 应用程序似乎是很困难的。事实上，只有对这些工具的使用和工程结构非常熟悉的人才会使用这些工具以命令行或脚本文件的方式来开发 Android 应用程序。

所以，对于大多数开发者而言，最好的途径是选择集成开发环境（IDE），除非开发者觉得使用命令行进行开发的方式更具有成就感。对于 Java 开发工具，特别是需要引入 Android 开发插件（后续即将介绍的 Android 调试工具）的集成开发环境，Eclipse 集成开发环境似乎成了最为明智的选择。

最新的 Eclipse 集成开发环境可以从其官方网站 <http://www.eclipse.org/downloads/> 下载。其中，面向 Java 开发的 Eclipse 集成开发环境还提供了多个平台的版本：Windows、Mac 和 Linux，读者选择目标平台相应的版本即可。

Android 应用程序开发对 Eclipse 的版本要求是 3.4 及以上版本，当前 Eclipse 最新的版本号是 3.6，代号为 HELIOS（太阳神）。



至此, JDK、Android SDK、Eclipse 集成开发环境具备了, 那么, 如何将 Android SDK 融入到 Eclipse 集成开发环境中?

熟悉 Eclipse 开发的读者可能会马上想到以 Eclipse 插件 (Plug-in) 的方式将 Android SDK “嵌入” 到 Eclipse 集成开发环境中, 而事实上正是如此。Android 开发者网站提供了一套用于 Android 应用程序开发的 Eclipse 集成开发环境插件。开发者只需在 Eclipse 集成开发环境中安装该插件, 即可方便地开发 Android 应用程序了。

2.4 Android 应用程序开发工具

Android 开发工具 (Android Development Tools, ADT) 是为在 Eclipse 集成开发环境下开发 Android 应用程序提供的工具插件。

2.4.1 获取 ADT

ADT 插件的最新可用信息可以通过 Eclipse 工具的软件更新功能来获知。

2.4.2 安装配置 ADT

1. 安装 ADT 插件

1) 启动 Eclipse, 选择菜单 “Help (帮助)” → “Install New Software (安装新软件)”, 将会显示【可用软件 (Available Software)】窗口, 在文本框中输入 ADT 资源的获取链接 “<https://dl-ssl.google.com/android/eclipse/>”。

2) 单击 “Next (下一步)” 按钮完成 ADT 插件的安装。

2. 更新 ADT 插件

1) 启动 Eclipse, 选择菜单 “Help (帮助)” → “Check for Updates (检查更新)”, 将会显示【可用更新 (Available Updates)】窗口。

2) 单击 “Next (下一步)” 按钮完成 ADT 插件的更新安装。

3. 设置 ADT 插件选项

1) ADT 插件安装完毕后, 重启 Eclipse。选择菜单 “Window (窗体)” → “Preferences (首选项)”, 进入【首选项设置】窗口。

2) 选择左侧的 “Android” 项, 设置 “SDK Location (SDK 的位置)” 为 Android SDK 安装文件夹, 然后单击 “OK” 按钮即可。

2.5 创建 Android 虚拟设备

从 Android 1.5 开始, 启动应用程序之前必须至少建立一个 Android 虚拟设备 (Android Virtual Device, AVD)。

通过 Eclipse 的菜单 “Window (窗体)” → “Android SDK and AVD manager (Android SDK 和虚拟设备管理器)”, 会显示【Android 虚拟设备管理】窗口。

创建一个虚拟设备必须指定名称、目标平台、SD 卡和皮肤等选项。虚拟设备创建之后, 创建的虚拟设备信息将存放于当前用户文件夹下的 android 子文件夹中。

2.6 Android 应用程序开发环境验证

2.6.1 搭建工程

1. 第一个 Android 平台应用程序

1) 启动 Eclipse，选择菜单“File（文件）”→“Android Project（Android 工程）”进入【新建 Android 工程设置】窗口，如图 2-4 所示。

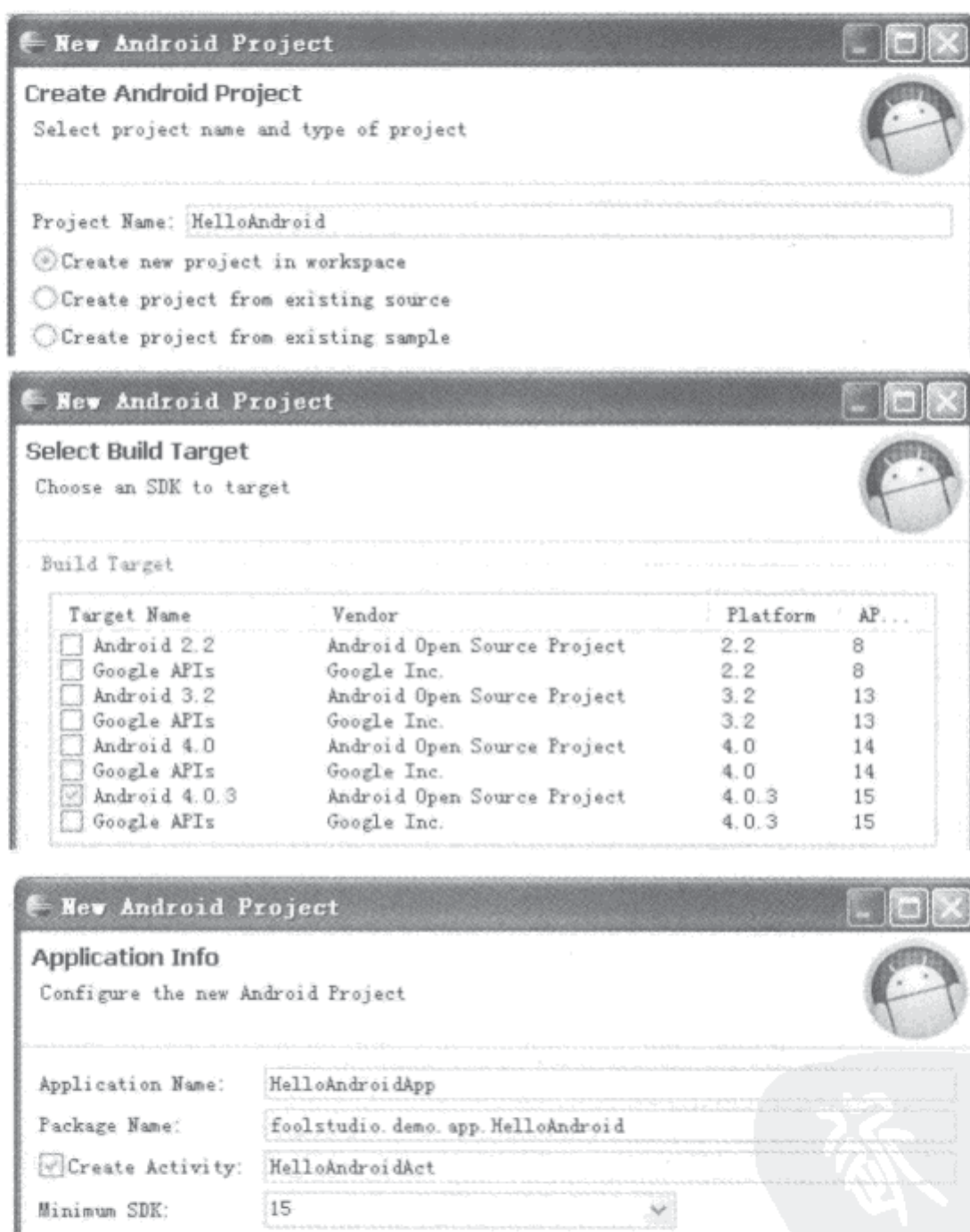


图 2-4 New Android Project 窗口

其中必须填写的项有：

- 工程名（Project name），填入 HelloAndroid。
- 编译平台（Build Target），选择 Android 4.0.3。
- 应用程序名（Application name），填入 HelloAndroidApp。
- 包名（Package name），填入 foolstudio.demo.app.HelloAndroid。
- Activity 名（Create Activity），填入 HelloAndroidAct。



Android 平台开发之旅 第2版

● 最小 SDK 版本 (Min SDK Version)，默认为编译平台的 API 级别 (API Level)。

2) 单击 “Finish (完成)” 按钮完成工程的创建，回到 Eclipse 代码编辑窗口。

注意：包名实际上也是应用程序名 (进程名)，其在系统中必须是唯一的。如果有两个应用程序的包名相同，那么包管理器在安装时会自动替换前面已安装的程序。

2. 工程结构内容

完成工程创建之后，在 Eclipse 左侧的 “Package Explorer (包浏览)” 窗体中将出现一个名为 HelloAndroid 的工程。图 2-5 中是 HelloAndroid 工程的结构内容。

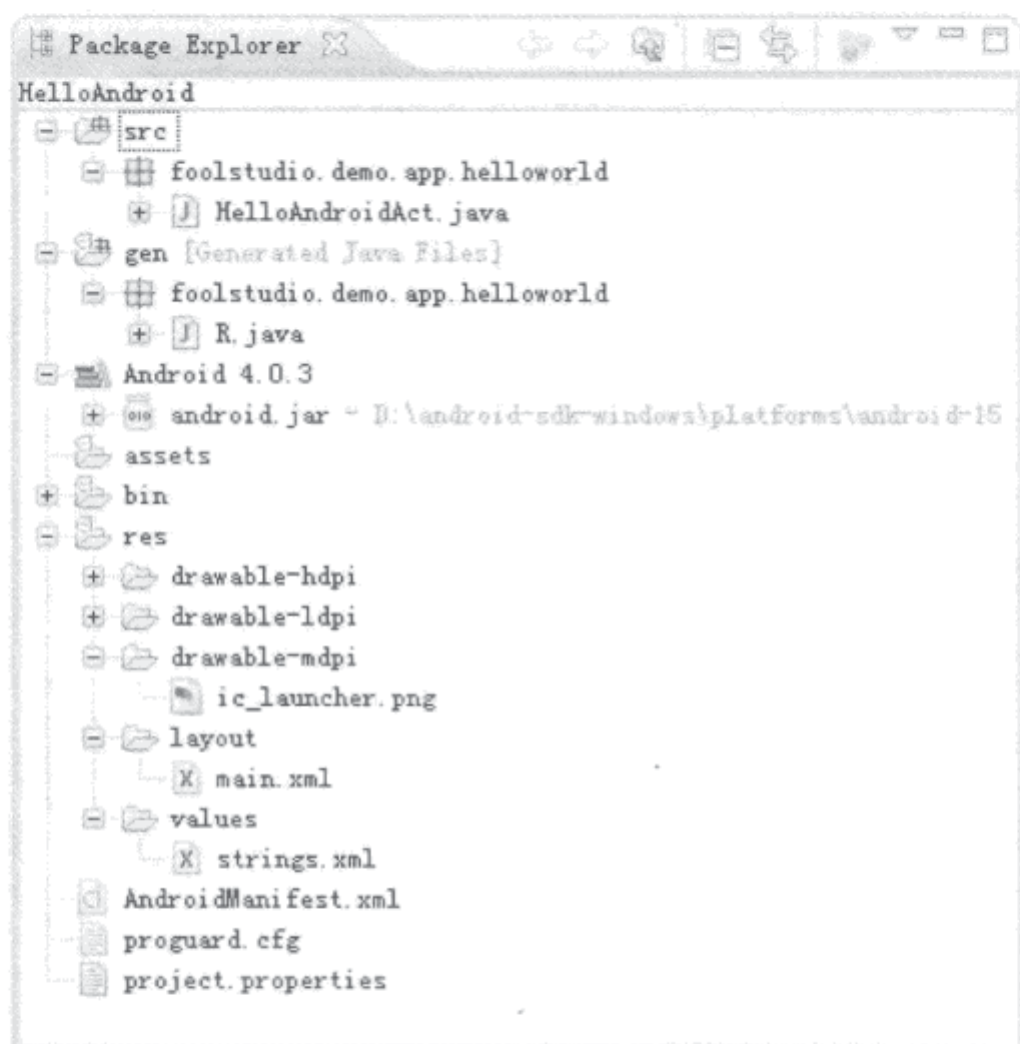


图 2-5 HelloAndroid 工程的结构内容

实际上，图 2-5 所示工程的结构内容是最基本的，用户还可以在此基础上添加其他需要用到或自定义的内容。下面将对 Android 工程结构的节点内容进行介绍。

(1) src——源代码节点

源代码节点用于管理由 ADT 自动生成的 Activity 框架代码以及用户自己创建的代码。代码 2-1 是图 2-5 中 HelloAndroidAct.java 的内容。

代码 2-1 Activity 组件的框架定义

文件名: HelloAndroidAct.java

```
1 public class HelloAndroidAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
```



```

5         setContentView(R.layout.main);
6     }
7 };

```

代码 2-1 的代码是由 ADT 自动生成的 Activity 框架代码，虽然只是寥寥几行，但已经是一段完整的代码。

(2) gen——自动生成内容节点

自动生成内容节点用于管理由 ADT 工具自动生成的内容，主要是资源定义文件。代码 2-2 是图 2-5 中资源定义文件 R.java 的内容。

代码 2-2 资源定义文件 R.java

文件名: R.java

```

1  public final class R {
2      public static final class attr {}
3      public static final class drawable { public static final int icon=0x7f020000; }
4      public static final class layout {     public static final int main=0x7f030000; }
5      public static final class string {     public static final int app_name=0x7f040001;
6                                          public static final int hello=0x7f040000;
7      }
8  };

```

如同该文件的头部说明，该文件是 aapt 工具（资源打包工具）通过探索资源内容而自动生成的。所以，其文件名“R”可以理解为资源（Resource），该文件中的内容是 res 节点中的内容（如 icon.png、main.xml）的资源 ID。

至此，读者应该可以大致猜想到，该文件的目的就是提供资源 ID 到具体资源的映射，方便应用程序对资源的引用。

(3) Android 4.0.3——参考库节点

参考库节点用于管理 Android 工程需要引入的外部参考库。如图 2-5 所示，该工程参考的是 Android 平台 4.0.3 版本的开发包。

(4) assets——工程资产节点

工程资产节点用于管理 Android 工程所引入的资产内容。

(5) res——资源节点

资源节点用于管理工程所引入的资源内容。在图 2-5 中，该节点又包含了几个子节点。

1) 子节点 drawable，用于管理可绘制的资源，主要是图片资源，且按照图片质量（dpi，每英寸点数）的高低，又分为 hdpi（高）、mdpi（中）、ldpi（低）3 个档次。

2) 子节点 layout，用于管理用户界面的设计布局。代码 2-3 是以内容查看方式查看 main.xml 文件的内容。

代码 2-3 main.xml 文件的界面布局

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"

```




Android 平台开发之旅 第2版

```

4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <TextView android:text="@string/hello"
6          android:layout_width="fill_parent" android:layout_height="wrap_content" />
7  </LinearLayout>

```

如果读者对 XML 文件格式略知一二，可以从代码 2-3 中大致了解该文件的内容含义：根节点为一个垂直方向、宽度和高度可以填充其父容器的线性布局（LinearLayout），该布局又包含一个宽度填充其父容器、高度为其内容本身的文本视图（TextView）。

3) 子节点 values，用于常量值的管理。代码 2-4 是资源文件 strings.xml 的内容。

代码 2-4 strings.xml 文件中的字符串常量

文件名: strings.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <string name="hello">Hello World, HelloAndroidAct!</string>
4      <string name="app_name">HelloAndroidApp</string>
5  </resources>

```

从代码 2-4 中，strings.xml 文件定义了两个字符串资源：hello 和 app_name，在代码 2-3 中第 5 行所引用的 hello 资源就是其中的 hello 字符串资源。

(6) AndroidManifest.xml——工程清单文件

工程清单文件（AndroidManifest.xml）为工程文件，其中包含了工程的基本信息、组件、使用许可等信息。代码 2-5 是图 2-5 中工程清单文件的内容。

代码 2-5 工程清单文件

文件名: AndroidManifest.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="foolstudio.demo.app.HelloAndroid"
4      android:versionCode="1"
5      android:versionName="1.0">
6      <uses-sdk android:minSdkVersion="15" />
7      <application android:icon="@drawable/ic_launcher" android:label="@string/app_name">
8          <activity android:name=".HelloAndroidAct" android:label="@string/app_name">
9              <intent-filter>
10                 <action android:name="android.intent.action.MAIN" />
11                 <category android:name="android.intent.category.LAUNCHER" />
12             </intent-filter>
13          </activity>
14      </application>
15 </manifest>

```

通过该工程清单文件，读者可以获取包名、应用程序名以及包含的 Activity 组件等信

息。其中的应用程序节点（**application**）描述的就是当前的应用程序，该应用程序包含一个 **Activity** 组件（第 8 行），系统的 **Activity** 管理器依据意向过滤器所指定的行为（第 10 行）和分类（第 11 行）来启动该 **Activity**。

（7）default.properties——属性文件

属性文件由 **Android** 工具自动生成，主要记录工程的目标平台的标识信息，无需手动编辑。目标平台的标识信息形如“**target=android-<API 级次>**”，如：**target=android-15**。

2.6.2 运行工程

1) 选择当前工程，通过菜单“**Run（运行）**”→“**Run Configurations（运行设置）**”进入【工程运行配置】对话框。

2) 双击对话框左侧的“**Android Application（Android 应用程序）**”选项，**Eclipse** 会为该工程创建一个新的配置，如图 2-6 所示。

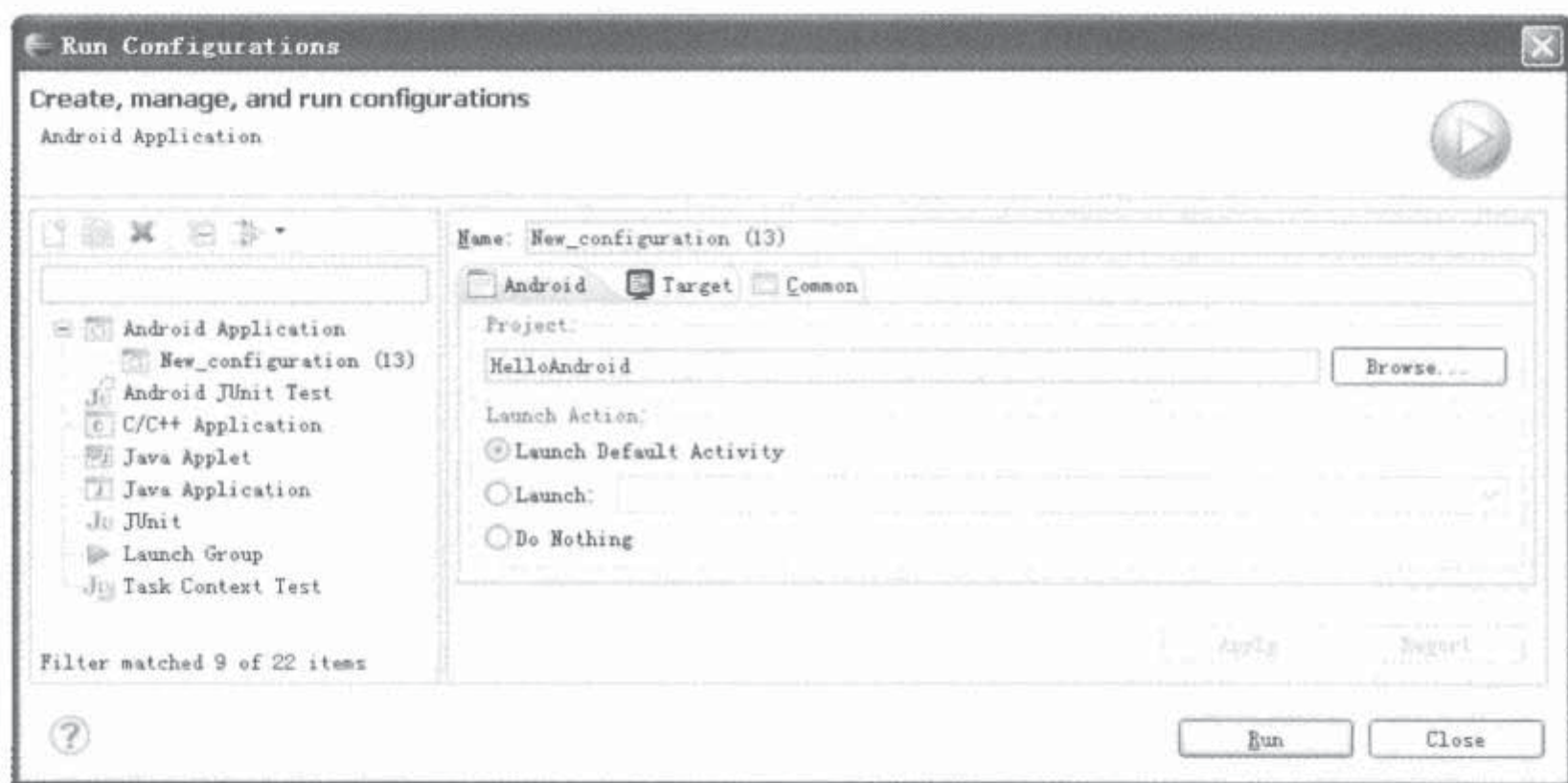


图 2-6 创建/管理运行配置

Android 工程的运行配置详细页包含 3 个选项卡：**Android**、**Target（目标）**和 **Common（公共）**，其中：

① **Android** 选项卡用于设置需要运行的工程以及启动的 **Activity**。

② **Target** 选项卡主要用于设置需要分发应用程序的虚拟设备。默认的，**ADT** 会根据当前环境自动地选择虚拟设备。

运行配置设置完毕，即可单击“**Run（运行）**”按钮来运行该应用程序。**ADT** 会先判断当前有无正在运行的虚拟设备，如果有则会连接到该虚拟设备，安装应用程序，再启动；反之，则会先启动一个新的虚拟设备，再执行连接、安装和启动的步骤。

图 2-7 是 **HelloAndroid** 工程在 **Android** 平台的模拟器上运行的画面，其模拟设备的显示标准为 **WXGA**，分辨率为 **1280×800** 像素（摩托罗拉公司的平板电脑 **XOOM** 采用该规格）。加上边框，模拟器工具的界面宽高为 **1339×887** 像素，而一般 19in 显示器的最高分



分辨率为 1440×900 像素，所以要想完全展示模拟器界面，显示器的屏幕规格最低是 19in。

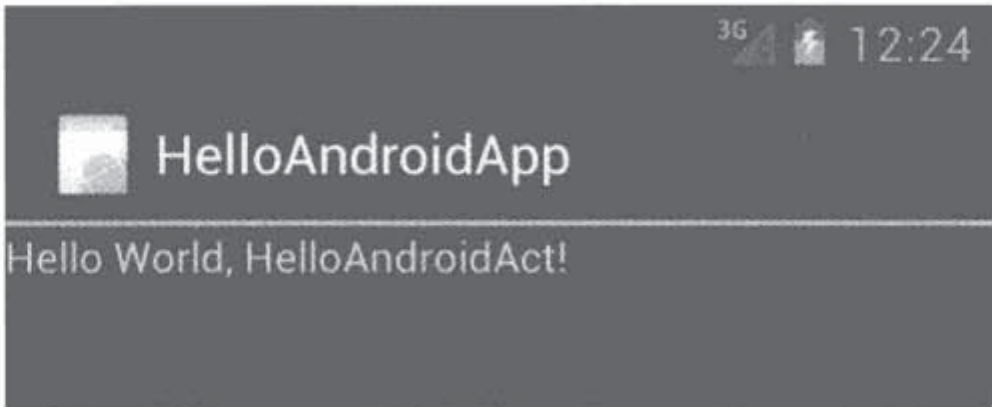


图 2-7 HelloAndroid 工程运行界面

HelloAndroid 是本书中第一个 Android 平台的应用程序，也是读者 Android 平台开发之旅的起点，希望读者能够保持兴趣，打开自身技术生涯的新局面。

2.7 应用程序开发过程

2.7.1 开发过程回顾

通过前一节对 HelloAndroid 工程从创建到调试的介绍，相信读者对 Android 应用程序的开发过程已经产生初步的印象，图 2-8 是 Android 应用程序开发的大致流程。

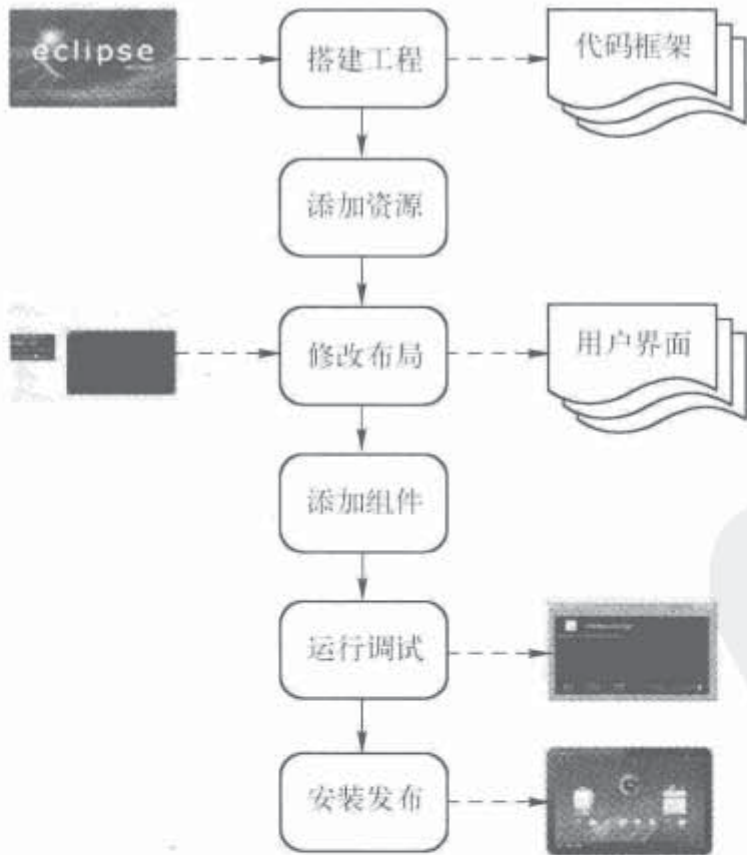


图 2-8 Android 应用程序开发流程

接下来，本书继续以 HelloAndroid 工程为例，对 Android 应用程序的开发过程进行简要的描述，例子中将改用图片来替换文本显示。

1. 搭建工程

在前一节中已经对如何搭建 Android 工程进行了详细介绍，不再重述。

2. 添加图片资源

将图片文件复制到工程文件夹下的 `res\drawable-mdpi` 子文件夹中。需要注意的是，图片类型必须符合 Android 平台的要求，图片文件名必须符合 ADT 对资源的命名规则。

3. 为文本组件设置 ID

编辑布局定义文件，给文本视图（TextView）设置 ID，并去掉其文本内容。修改后的布局定义内容如代码 2-6 所示。

代码 2-6 修改后的布局定义

文件名: main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent" android:layout_height="fill_parent">
5     <TextView android:id="@+id/text"
6         android:layout_width="wrap_content" android:layout_height="wrap_content"/>
7 </LinearLayout>
```

代码 2-6 中第 5 行，为文本视图添加了名为“text”的 ID，添加完毕，ADT 工具会自动为该 ID 在 R.java 文件中生成一个数值标识，如代码 2-7 所示。

代码 2-7 资源 ID 标识定义

文件名: R.java

```
1 public static final class id {
2     public static final int text=0x7f050000;
3 }
```

4. 添加组件

一个应用程序可以包含多个 Android 组件（Activity、服务、广播接收器、内容提供者）。一般，这些组件在定义之后还需要在应用程序清单文件（AndroidManifest.xml）中进行“登记”，以便 Android 平台进行调用。

5. 运行工程

在前一节中已经对如何运行 Android 工程进行了详细介绍，不再赘述。

6. 调试 Android 工程

代码调试方式一般可分为两种：一种是消极的调试，即以调试模式运行，直到遇到异常时才进行调试分析；另一种是主动的调试，即在可能出现异常的代码段添加调试断点。第二种方式可以在异常发生之前跟踪代码的执行状况，及早捕获异常原因。

本例中将结合使用这两种方式来进行代码调试。

(1) 修改 Activity 组件定义

编辑 Activity 组件定义，修改后的内容如代码 2-8 所示。



代码 2-8 修改后的 Activity 组件定义

文件名: HelloAndroidAct.java

```
1 public class HelloAndroidAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         //通过 Id 获取视图对象实例
6         TextView v = (TextView) findViewById(R.id.text);
7         //设置显示文本
8         v.setText("你好! ");
9         //设置内容视图
10        setContentView(R.layout.main);
11    }
12 };
```

在代码 2-8 中，首先通过 ID 获取文本视图实例（第 10 行），然后设置该视图的文本显示（第 12 行）。

资源及 Activity 定义编辑完毕之后，即可开始调试应用程序了。

(2) 消极调试

1) 选择菜单“(Run) 运行”→“Debug Configurations (调试配置)”进入到【工程调试配置】对话框（见图 2-9），该对话框与【工程运行配置】对话框大致相同，唯一区别在于其右下方为“Debug（调试）”按钮而后者为“Run（运行）”按钮。

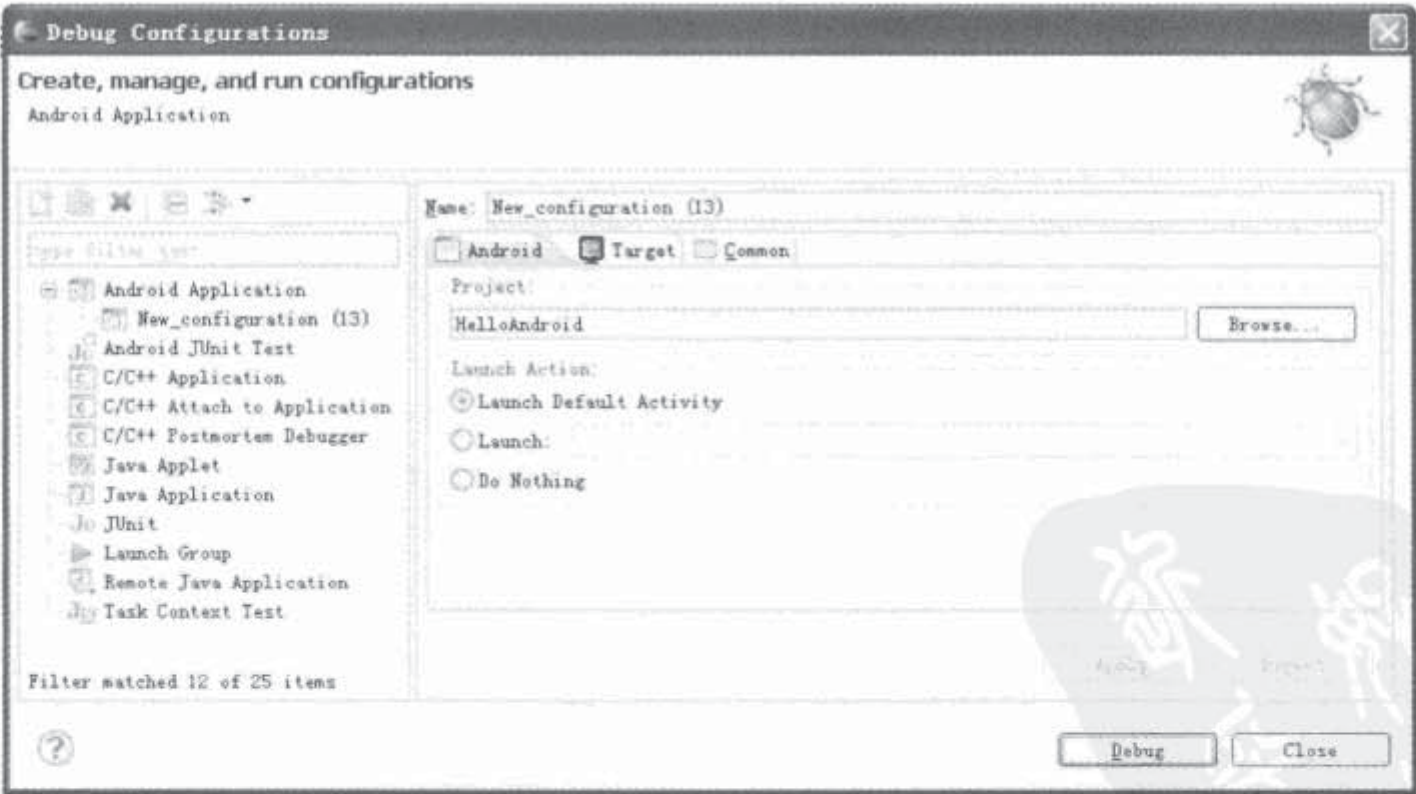


图 2-9 调试配置界面

2) 单击“调试”按钮，将以调试模式运行该工程。运行过程中 Android 平台可能会弹出等待调试器的提示（见图 2-10），开发者可以不予理会，当调试器绑定结束之后该提示会自动消失。

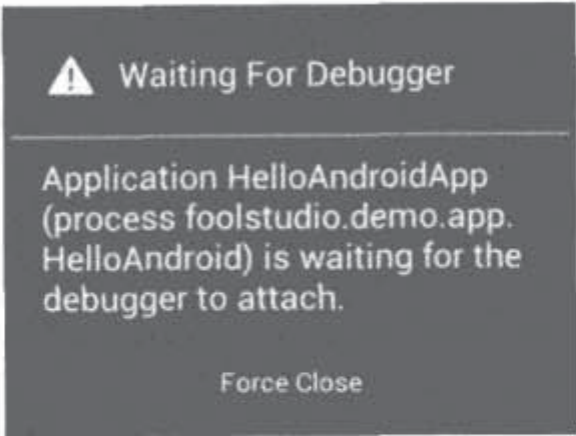


图 2-10 等待调试器的提示

3) 调试器绑定完毕，开始进入代码执行，其中遇到如图 2-11 所示的异常提示。



图 2-11 程序异常的提示

4) 选择菜单“Windows（窗体）”→“Show View（显示视窗）”→“LogCat（日志分类）”来显示【日志分类】窗体。通过其中的运行日志，开发者可以迅速找到异常的初步原因：

E/AndroidRuntime(991): Caused by: java.lang.NullPointerException

该异常起源于空指针异常，接下来另外一条错误信息指示了造成异常的代码位置：

at foolstudio.demo.app.HelloAndroid.HelloAndroidAct.onCreate(HelloAndroidAct.java:24)

双击该条信息可以直接定位到源文件 HelloAndroidAct.java 的第 24 行：

v.setText("你好！");

至此，引起该异常的原因可能是因 v 对象为 null（空）导致的空指针异常。接下来，开发者在可能造成异常的代码行前设置断点，跟踪问题变量的赋值情况。

(3) 主动调试

1) 在代码中设置断点，如图 2-12 所示。

```
17 public class HelloAndroidAct extends Activity {
18     @Override
19     public void onCreate(Bundle savedInstanceState) {
20         super.onCreate(savedInstanceState);
21         //通过id获取视图对象实例
22         TextView v = (TextView) findViewById(R.id.text);
23         //设置显示文本
24         v.setText("你好！");
25         //设置内容视图
26         setContentView(R.layout.main);
27     }
28 }
```

图 2-12 在代码中设置断点



2) 以调试模式运行该工程，当运行停留到设置断点的代码行，将鼠标移动到变量上，Eclipse 将会弹出【变量值查看】窗体，如图 2-13 所示。

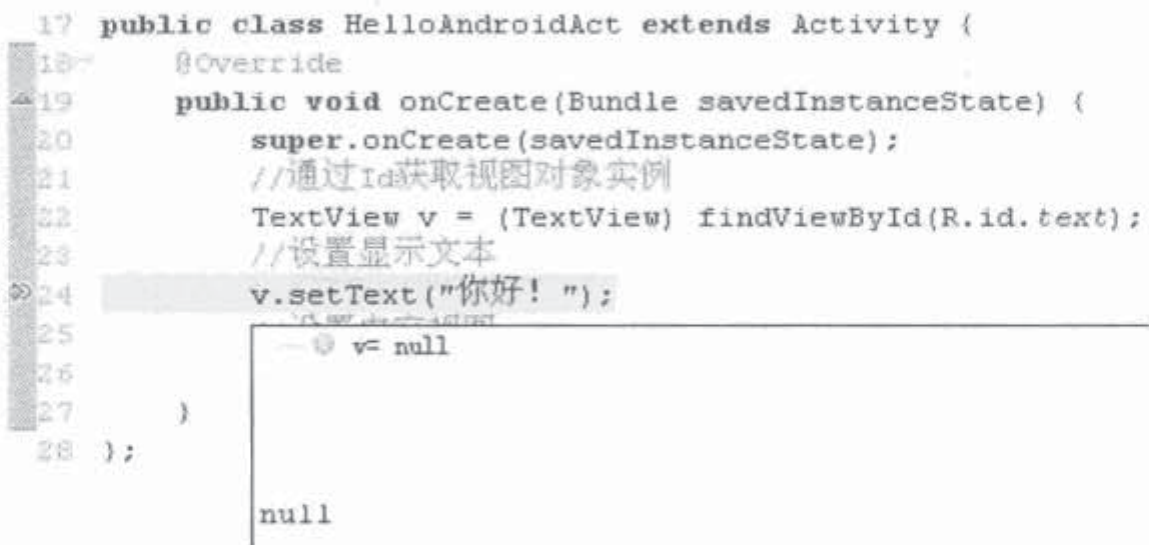


图 2-13 在调试模式下查看变量值

由此可以确定，图 2-11 所示的异常正是因为文本视图对象为空值所造成的。

7. 安装发布

Android 应用程序以包文件（apk）的形式进行发布。Android 平台通过包管理器对应用程序进行安装和卸载。

2.7.2 代码调试技巧

对于桌面应用程序的开发过程，开发和调试都在相同的环境；而对于 Android 平台，开发环境和调试环境是不同的，特别是需要在模拟器上运行调试。这些因素无疑增加了调试 Android 应用程序的难度，所以，掌握一定的调试技巧是相当有必要的。

(1) 在【日志分类】窗口中通过日志级次按钮快速查看不同级次的日志

通过日志级次按钮（V D I W E）可以按照级次查看日志信息，各日志级次及说明如表 2-1 所示。

表 2-1 日志级次及说明

级 次	说 明	输出颜色
V	详细级 (Verbose)	黑色
D	调试级 (Debug)	蓝色
I	信息级 (Information)	绿色
W	警告级 (Warning)	橙色
E	错误级 (Error)	红色

需要注意的是，不同级次的日志输出颜色是不同的，如图 2-14 所示。

(2) 养成在代码中输出日志的习惯

使用系统输出对象（System.out）的 println 方法或异常对象（Exception）的 printStack 方法都可以将调试信息输出到【日志分类】窗口，只是使用这两种方式输出的日志都显示为信息级。



图 2-14 【日志分类】窗口日志输出

推荐开发者使用 Android 工具包（android.util）中的 Log 类的日志输出方法，该方法可以设置输出日志的级次，代码如下所示。

```
Log.i(this.getPackageName(), "提示信息");
Log.d(this.getPackageName(), "调试信息");
Log.w(this.getPackageName(), "警告信息");
Log.e(this.getPackageName(), "错误信息");
```

图 2-15 是该代码在【日志分类】窗口的输出内容。

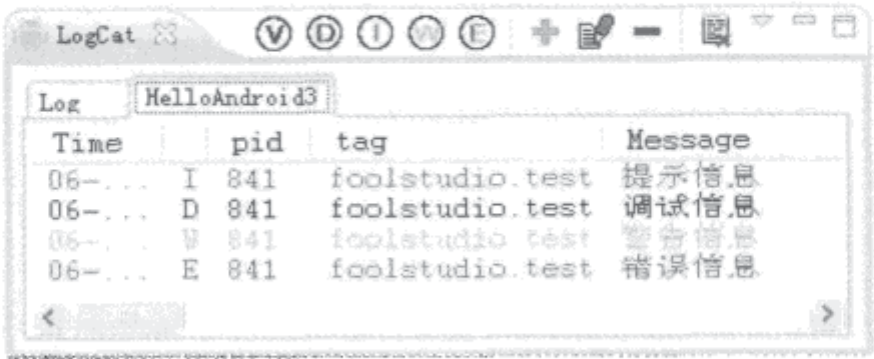


图 2-15 不同级次的日志输出

(3) 在【日志分类】窗口中创建日志过滤器来关注符合条件的日志
通过日志过滤器编辑按钮（+ -）可以建立或管理日志过滤器。通过日志过滤器可以只关注符合条件的日志。图 2-16 是创建日志过滤器的实例图。

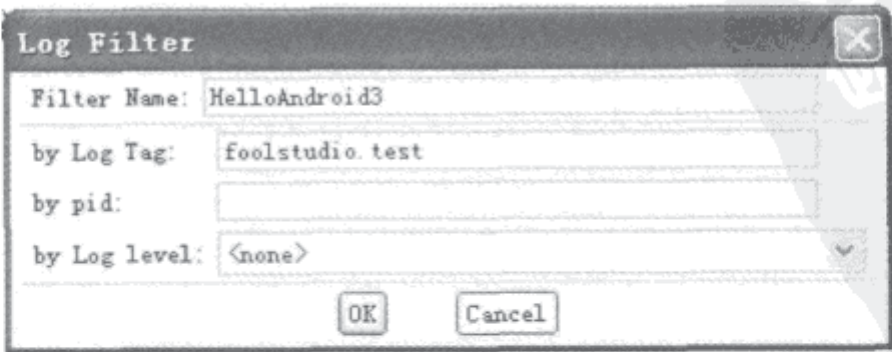


图 2-16 创建日志过滤器

图 2-16 中是按照日志的标签作为过滤条件，其输出内容如图 2-15 所示。

(4) 在调试运行过程中，Android 平台将会弹出等待调试器绑定的提示（见图 2-10）。



这时开发者不要单击“强制关闭”按钮，否则将无法进行后面的调试。

(5) 使用 Android 调试桥工具 (adb) 可以快速上传应用程序所需的资源，可以快速监测应用程序对模拟环境的文件系统的操作，如对文件的创建、删除等。

2.8 新手上路

Android 初学者在开发过程中可能会遇到一些莫名其妙的问题，如上一节中的文本视图对象为空值的问题。下面将结合该问题，对初学者在 Android 程序开发过程中可能遇到的一些问题进行探讨。

1. 无法通过资源 ID 获取实例

在前一节中，运行 HelloAndroid 工程遇到异常（见图 2-11），经调试确定是由于文本视图对象为空所造成的空指针异常。换言之，是因为 findViewById 方法所返回的对象实例为空（代码 2-8 第 6 行）。事实上，findViewById 方法所使用的 ID 为有效值，为什么使用有效 ID 无法获取到有效的对象？

在代码 2-8 中，本意是先设置布局中的文本视图，再将整个布局设置为 Activity 的内容视图，而问题正出在这两个设置的先后顺序。通过 Android SDK 文档，可以看出：对于 findViewById 方法的 ID 参数，必须是来自经过 onCreate 函数处理过的 XML 文件。也就是说，该 ID 不能仅仅经过定义就能使用，而是需要等 onCreate 函数对 XML 文件进行处理之后。而在 onCreate 函数中，需要先调用 setContentView 方法来填充 Activity 的用户界面。setContentView 方法的作用是：填充 ID 所指定的布局资源，并将该资源中定义的可视组件添加到 Activity 中。

所以，要使用一个布局内的组件，必须要先通过 setContentView 方法来填充该布局。也就是说，findViewById 方法必须在 setContentView 方法之后调用。

提示：这里的“填充”翻译于 SDK 参考中的单词 inflate，读者可以理解为 Android 平台解析资源定义文件（即 XML 文件），然后根据其中的内容生成资源对象实例的一个过程。

所以，代码 2-8 应该修改为代码 2-9 所示的内容，即 setContentView 方法的调用应该在所有 findViewById 方法之前。

代码 2-9 修改后的 Activity 组件定义

文件名：HelloAndroidAct.java

```
1 public class HelloAndroidAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         //设置内容视图
6         setContentView(R.layout.main);
7         //通过 Id 获取视图对象实例
8         TextView v = (TextView) findViewById(R.id.text);
```



```
9          v.setText("你好！");    //设置显示文本
10      }
11  };
```

修改后的代码在模拟器上的运行界面如图 2-17 所示。

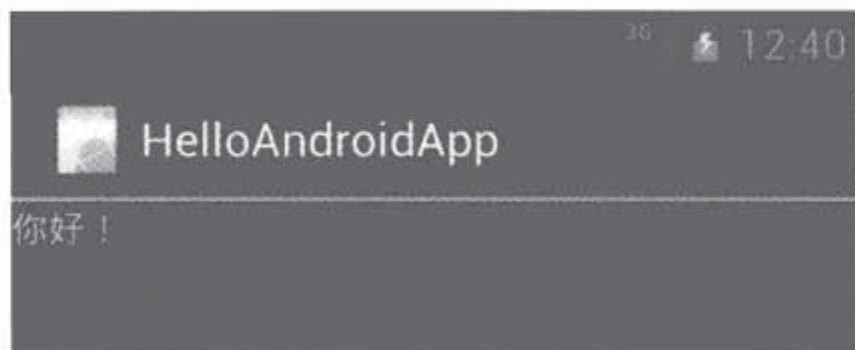


图 2-17 HelloAndroid 工程运行界面

2. 应用程序无响应错误

在 Android 平台中，应用程序的响应受到应用程序管理器的监视，如果在规定的时间内应用程序无响应，系统将弹出【应用程序无响应】（Application Not Responding, ANR）的对话框，并给出“确定关闭”和“等待”该应用程序的选择，如图 2-18 所示。



图 2-18 ANR 提示对话框

ANR 问题一般是由于在主线程中执行耗时较长的操作而造成，如通过互联网下载音乐或从数据库提取数据填充到内存模型中。由此可知，解决 ANR 问题的途径就是将耗时较长的操作改为异步的形式。例如，通过后台线程的方式下载音乐，而不阻塞前台用户界面的显示，下载完毕，再更新前台的可供播放的列表。

Android 平台提供了一组用于异步载入数据的类：加载器（Loader），使用加载器即可异步地进行数据加载，而不会造成主线程阻塞，从而避免 ANR 错误。

第 3 章 Android 应用程序组件

在进行深入开发之前，本章对开发的基本单元——程序组件进行详细的介绍，包括各组件的使用方式、框架及配置，希望读者真正了解各组件的特征和适用性，在此基础上能就具体应用进行各组件的集成设计。

此外，本章还对组件之间的一些交互机制和方式进行实例说明，通过实例希望读者能掌握这些机制的用法，为后面的应用集成奠定基础。

3.1 应用程序组件

有过软件项目开发经历的读者应该很了解，一个成型的项目往往需要由多个组件构成。在桌面平台的软件项目中，比较常见的组件形式有可执行代码、静态链接库和动态链接库。同为可执行程序，在用途方面可能存在较大的差异。有的可执行程序提供可视界面，与用户进行交互；有的不提供界面，而是作为后台服务。而对于动态链接库文件，有的包含资源定义，有的包含共享代码……其区分也主要体现在功用。

应用程序组件的多样化，其目的就是为了适应各种应用。例如，在 J2EE 平台，应用程序有 Applet、JSP、Servlet 等多种类型，各种组件形式有各自的适用场合。

3.2 Android 应用程序组件

同样，在 Android 平台也存在多种类型的应用组件。通过前两章的讲解，相信读者应该能列举一些 Android 平台的应用程序组件，如 Activity、服务、广播接收器和内容提供者。图 3-1 是这些重要的组件的类结构层次。



图 3-1 Android 应用程序组件的类结构层次

表 3-1 是对图 3-1 中应用程序组件类/接口的说明。

表 3-1 应用程序组件类/接口的说明

类/接口	说 明
android.app.Activity	Activity，通过用户界面提供与用户的交互
android.app.Service	服务，关注后台事务的操作
android.content.Context	上下文对象，代表应用程序所在环境的全局信息
android.content.ContentProvider	内容提供者，用于提供应用程序所需的数据
android.content.BroadcastReceiver	广播接收器，用于接收广播消息

3.2.1 Activity 组件——形象大使

Activity 组件提供一组可视界面来与用户进行交互，其主要用于处理前端事务，如 Applet 或者 Swing 程序（J2SE 平台）。将 Activity 组件比喻为形象大使的角色，是因为对于任何一款工具或游戏，其界面的体验效果将会直接影响用户对该软件的印象。如果某一款程序中有很多漂亮的“形象大使”，那么该程序一定会得到更多的用户关注。

图 3-2 是一个包含 Activity 组件的程序运行界面，其只展示了一张图片。



图 3-2 Activity 组件的程序运行界面

1. 应用程序主 Activity 框架

代码 3-1 是图 3-2 所示的程序中 Activity 组件的框架定义。

代码 3-1 Activity 组件的框架定义

文件名：HelloAndroidAct.java

```
1 public class HelloAndroidAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);           Log.i(this.getPackageName(), "创建");
6     }
7     @Override
8     protected void onRestart() {
9         super.onRestart();                       Log.d(this.getPackageName(), "重启");
10    }
11    @Override
12    protected void onStart() {
```




```
13         super.onStart();                                Log.w(this.getPackageName(), "启动");
14     }
15     @Override
16     protected void onPause() {
17         super.onPause();                                    Log.e(this.getPackageName(), "暂停");
18     }
19     @Override
20     protected void onResume() {
21         super.onResume();                                    Log.i(this.getPackageName(), "恢复");
22     }
23     @Override
24     protected void onStop() {
25         super.onStop();                                      Log.d(this.getPackageName(), "停止");
26     }
27     @Override
28     protected void onDestroy() {
29         super.onDestroy();                                  Log.w(this.getPackageName(), "销毁");
30     }
31 };
```

在代码 3-1 中，HelloAndroidAct 继承于父类 Activity，并重载父类的诸多方法，如 onCreate、onStart 等。其实这种创建模式与 J2SE 平台的 Applet 程序是一样的，开发者所编写的 Applet 组件必须继承于父类 Applet 或 JApplet，并重载父类的方法，如 init、start、stop、destroy 和 paint 等。需要注意的是，在 Activity 所重载的方法中，需要调用超类实例（super）的方法来完成平台的预设框架。

简而言之，Android 平台规定所有的 Activity 组件必须继承于其父类 Activity，这一规则由 Android 平台的应用程序框架来约定，读者可从 Applet 的程序框架获得启发。

2. Activity 程序生命周期




图 3-3 是代码 3-1 的日志输出（有关日志过滤器的创建可参考第 2 章）。



图 3-3 Activity 组件的程序输出日志

图 3-3 中的输入日志内容与右侧的按钮操作对应，而这些按钮实际上是模拟器左下方的控制按钮，其各自功能如表 3-2 所示。

表 3-2 控制按钮的说明

按 钮	说 明
	回退到前一界面或主页
	直接跳转到主页
	显示正在运行的或最近运行的 Activity 列表供用户选择

通过图 3-3，读者可初步了解 Activity 组件的生命主线：“创建”→“启动”→“恢复”→“暂停”→“停止”→“销毁”。Activity 组件的生命周期如图 3-4 所示。

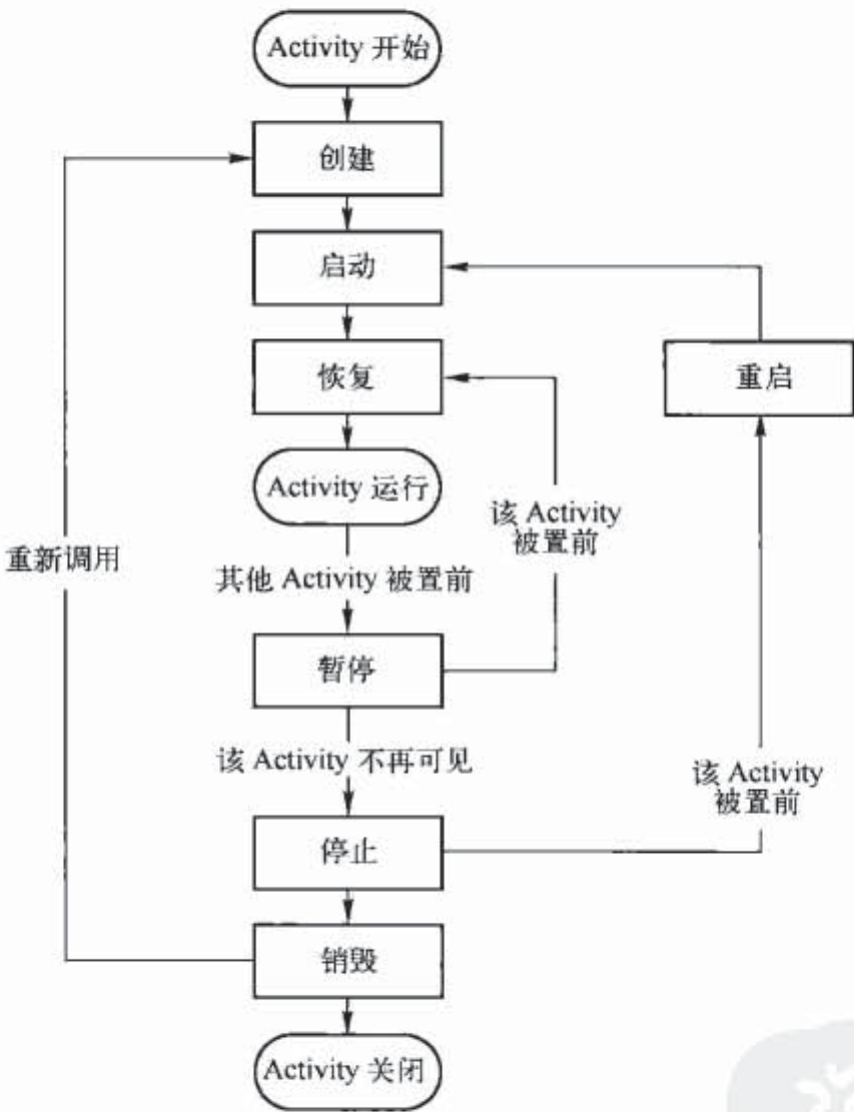


图 3-4 Activity 组件的生命周期

在图 3-4 中，Activity 被置前的方式有两种：第一种是该 Activity 被当前 Activity 所调用；第二种是在 Activity 列表中直接调用该 Activity。Activity 管理器（ActivityManager）会对当前正在运行的 Activity 和最近运行的 Activity 的信息进行管理。

3. Activity 程序界面展示

在代码 3-1 中的创建环节（onCreate 方法）中，Activity 使用 setContentView 方法来设置内容视图（第 5 行）。需要注意的是，setContentView 方法的参数是一个布局资源 ID，其对应布局资源文件夹（资源文件夹下 layout 子文件夹）中的 main.xml 文件，该文件定义了



一个布局资源，其内容如代码 3-2 所示。

代码 3-2 Activity 布局资源定义

文件名: main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent" android:layout_height="fill_parent">
5     <ImageView android:src="@drawable/gingerbread"
6         android:layout_width="wrap_content" android:layout_height="wrap_content"/>
7 </LinearLayout>
```

ADT 会根据布局资源定义自动生成 ID 到资源的映射（在 R.java 文件中），Activity 组件使用资源 ID 来访问资源内容，图 3-5 是 Activity 组件展示用户界面的实现机制。

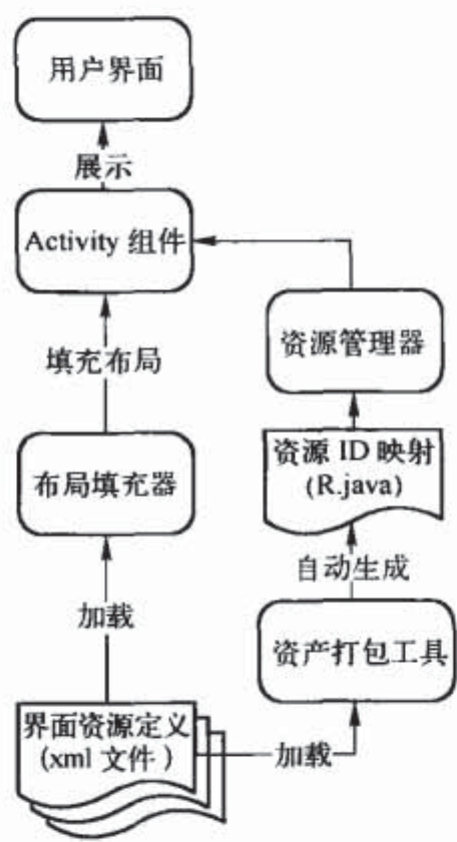


图 3-5 Activity 组件展示界面布局

4. 声明 Activity 组件

既然 Android 平台会对 Activity 程序进行管理，那么就必须知道 Activity 程序的相关信息，如同海关稽查员对箱子中的货品进行检查，首先要查看装箱单。Activity 程序也提供了与装箱单类似的内容——程序清单（Manifest）。

代码 3-3 是在应用程序清单中声明 Activity 的实例代码。

代码 3-3 在应用程序清单中声明 Activity

文件名: AndroidManifest.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
```



```

3      package="foolstudio.demo.app.helloworld2"
4      android:versionCode="1"
5      android:versionName="1.0">
6      <uses-sdk android:minSdkVersion="12" />
7      <application android:icon="@drawable/icon" android:label="@string/app_name">
8          <activity android:name=".HelloAndroidAct" android:label="@string/app_name">
9              <intent-filter>
10                 <action android:name="android.intent.action.MAIN" />
11                 <category android:name="android.intent.category.LAUNCHER" />
12             </intent-filter>
13         </activity>
14     </application>
15 </manifest>

```

通过代码 3-3，读者可获知该 Activity 程序的包名、版本代码、版本名和最低 SDK 版本等与安装有关的信息（第 3~6 行）；应用程序的信息在<application>节点中进行描述，包括程序图标和标签（第 7 行）；Activity 节点用于描述程序中每个 Activity 的信息，包括 Activity 类名和标签（第 8 行）；子节点<intent-filter>描述了该 Activity 的意向过滤信息，用于指明该 Activity 允许的动作和分类（第 10 行和第 11 行）。

提示：一个 Android 应用程序可包含多个 Activity，但只能有一个 Activity 用于启动，多个 Activity 之间可进行相互调用，形成一个 Activity 栈，Android 平台对 Activity 栈进行管理。

3.2.2 服务组件（Service）——老黄牛

Android 平台对服务的定义和读者熟知的平台所描述的基本是一致的，Android 平台中的服务组件不提供可视界面，主要用于后台任务，如从互联网下载文件、播放背景音乐等。作者将服务组件定义为老黄牛的角色，埋头苦干，总是很少抛头露面。服务组件与用户的交互需要通过 Activity 组件进行桥接，图 3-6 描述了服务组件的两种使用方式。

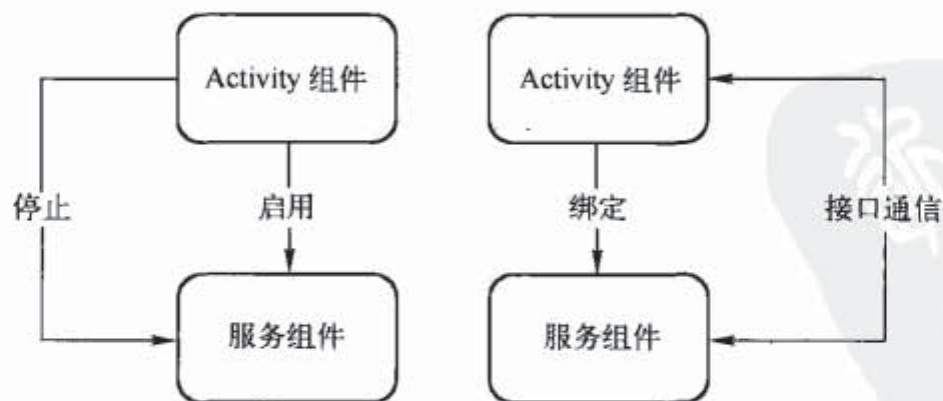


图 3-6 服务组件的使用方式

在图 3-6 中，第一种方式是 Activity 调用服务组件，除了启用和停止服务之外，Activity 无法与服务组件进行交互，如停止背景音乐的播放；第二种方式是 Activity 与服务组件使用绑定的方式进行连接，连接成功之后，Activity 可通过服务接口与服务组件进行通信。



图 3-7 是在 Activity 界面中启动和停止服务组件的实机界面。



图 3-7 启动和停止服务组件的实机界面

1. 应用程序主 Activity 框架

代码 3-4 是图 3-7 中 Activity 组件的框架定义，服务组件在 Activity 组件中启动和停止。

代码 3-4 Activity 组件的框架定义

文件名: ServiceDemoAct.java

```

1  public class ServiceDemoAct extends Activity implements OnClickListener {
2      //服务启动意向
3      private Intent mIntent = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main_view);
9          //获取按钮组件并设置单击侦听器
10         Button btnStart = (Button)findViewById(R.id.BTN_START);
11         Button btnStop = (Button)findViewById(R.id.BTN_STOP);
12         btnStart.setOnClickListener(this);
13         btnStop.setOnClickListener(this);
14     }
15     @Override
16     public void onClick(View v) {
17         switch(v.getId()) {
18             case R.id.BTN_START: { doStart(); break; } //启动服务
19             case R.id.BTN_STOP: { doStop(); break; } //停止服务
20         }
21     }
22     .....
23 };

```

在代码 3-4 中，Activity 组件定义了一个意向和两个按钮，其中意向是用于启动服务组件（意向的使用在后面介绍），而两个按钮分别用于启动和停止服务。

（1）启动服务组件

代码 3-5 是代码 3-4 中启动服务组件的代码，首先对意向进行初始化，意向包含服务组件的信息（第 3 行）。

代码 3-5 启动服务

文件名: ServiceDemoAct.java

```
1 private void doStart() { //启动服务
2     this.mIntent = new Intent(this, FooService.class);
3     this.startService(this.mIntent);
4 }
```

(2) 停止服务组件

代码 3-6 是代码 3-4 中停止服务组件的代码, 停止服务组件也需要意向。

代码 3-6 停止服务

文件名: ServiceDemoAct.java

```
1 private void doStop() { //停止服务
2     if(this.mIntent != null) { this.stopService(this.mIntent); }
3 }
```

2. 服务组件定义框架

代码 3-7 是代码 3-5 中提到的服务组件的完整定义。

代码 3-7 服务组件定义

文件名: FooService.java

```
1 public class FooService extends Service {
2     @Override
3     public IBinder onBind(Intent intent) { return null; }
4     @Override
5     public void onCreate() { super.onCreate();
6         Log.i(this.getPackageName(), "创建服务");
7     }
8     @Override
9     public void onStart(Intent intent, int startId) { super.onStart(intent, startId);
10        Log.i(this.getPackageName(), "启动服务");
11        //提示服务端时间戳
12        Toast.makeText(this, "服务端时间: "+FooSysUtil.getInstance().getTsp(),
13            Toast.LENGTH_LONG).show();
14    }
15    @Override
16    public void onDestroy() { super.onDestroy();
17        Log.w(this.getPackageName(), "销毁服务");
18    }
19 };
```

与 Activity 组件一样, 服务组件的定义也要遵循 Android 应用程序框架: 所有的服务组件必须继承于其父类: Service, 子类通过重载父类的方法来实现其特性。

该服务子类在启动后弹出提示条, 显示系统当前的时间戳。图 3-8 中的内容是单击“启



动”和“停止”按钮所输出的日志。



图 3-8 服务组件的输出日志

当单击“启动”按钮后，首先创建服务组件，然后是启动；当单击“停止”按钮后，该服务被销毁。

3. 声明服务组件

与 Activity 组件一样，服务组件也必须在程序清单中进行“登记”，否则 Android 平台将无法找到该服务。以下是声明服务组件的 XML 代码，该服务和主 Activity 都在应用程序标记（Application）中声明。

文件名：AndroidManifest.xml

```
<service android:name=".FooService" android:label="@string/service"/>
```

3.2.3 广播接收器组件（Broadcast Receiver）——倾听者

如果说 Activity 和服务都是实干派，那么将广播接收器组件定义为倾听者的角色是再恰当不过了。在 Android 平台中，广播接收器组件用于接收和响应系统广播的消息。与服务组件一样，广播接收器组件也需要通过 Activity 组件与用户进行交互。图 3-9 是广播接收器组件接收到广播并显示消息内容的界面。



图 3-9 接收广播消息并显示

1. 广播接收器组件的使用模式

广播接收器组件的使用可分为 Java 代码引用和 XML 代码引用两种方式。

(1) Java 代码引用

1) 应用程序主 Activity 框架。代码 3-8 是图 3-9 所示的界面所对应的 Activity 组件的定义。

代码 3-8 广播接收器示例程序 Activity 组件的定义

文件名：ReceiverAct.java

```
1 public class ReceiverAct extends Activity implements OnClickListener {
2     //广播接收器
3     private FooReceiver mReceiver = null;
4
5     @Override
6     public void onCreate(Bundle savedInstanceState) {
```



```
7      super.onCreate(savedInstanceState);
8      setContentView(R.layout.main_view);
9      //获取按钮组件并设置单击侦听器
10     Button btnRegister = (Button)findViewById(R.id.btn_reg);
11     Button btnSend = (Button)findViewById(R.id.btn_send);
12     btnRegister.setOnClickListener(this);
13     btnSend.setOnClickListener(this);
14     //初始化广播接收器
15     mReceiver = new FooReceiver();
16 }
17 @Override
18 public void onClick(View v) {
19     switch(v.getId() ) {
20         case R.id.btn_reg: { doRegister(); break; } //注册接收器
21         case R.id.btn_send: { doSend(); break; } //发送广播
22     }
23 }
24 .....
25 };
```

在代码 3-8 中，在 Activity 创建环节对广播接收器进行初始化（第 15 行），通过按钮来注册接收器以及发送广播（第 20 行和第 21 行）。

2) 注册广播接收器。代码 3-9 是代码 3-8 中注册广播接收器组件的代码。

代码 3-9 注册广播接收器

文件名: ReceiverAct.java

```
1 private void doRegister() { //注册接收器
2     IntentFilter filter = new IntentFilter(FooReceiver.class.getName() );
3     this.registerReceiver(mReceiver, filter);
4 }
```

从代码 3-9 可知，注册广播接收器不仅需要接收器组件，而且还需要意向过滤器。接收器组件用于明确谁来接收广播；意向过滤器用于明确接收哪些广播。广播接收器与过滤器是一对多的关系，即一个广播接收器可注册多个意向过滤器。

提示：几乎所有的系统状态信息都是以广播的形式发送，如手机状态改变、网络连接状态改变等，所以为了获取系统状态信息，大多数采用广播接收器，并按照系统为各个广播所定义的意向动作来接收系统广播。表 3-3 是 Android 平台所定义的标准广播动作。

表 3-3 标准广播动作

动 作	说 明
ACTION_BOOT_COMPLETED	系统启动完毕
ACTION_BATTERY_CHANGED	电池状态改变

**Android 平台开发之旅 第2版**

3) 注销广播接收器。Activity 组件既可注册广播接收器,也可对其进行注销;注销之后,广播接收器将不再接收广播。代码 3-10 是注销广播接收器的代码。

代码 3-10 注销广播接收器

文件名: ReceiverAct.java

```

1  @Override
2  protected void onDestroy() { super.onDestroy();
3      this.unregisterReceiver(mReceiver); //注销接收器
4  }

```

(2) XML 代码引用

1) 声明广播接收器。广播接收器可作为组件在应用程序清单中声明,该方式下无需再对广播接收器进行注册和注销。代码 3-11 是在应用程序清单中声明广播接收器的 XML 代码。

代码 3-11 在应用程序清单中声明广播接收器

文件名: AndroidManifest.xml

```

1  <receiver android:name=".FooReceiver" android:label="@string/receiver" >
2      <intent-filter>
3          <action android:name="foolstudio.demo.app.receiver2.FooReceiver" />
4      </intent-filter>
5  </receiver>

```

代码 3-11 中注册了一个广播接收器,其名称属性(android:name)指明了接收广播的接收器组件名称;其意向过滤器用于指明该广播接收器只能接收那些动作与指定动作(第 3 行)匹配的广播。

2) 应用程序主 Activity 框架。代码 3-12 使用 XML 代码引用广播接收器示例程序的主 Activity 组件的框架定义。

代码 3-12 广播接收器示例程序 Activity 组件的框架定义

文件名: Receiver2Act.java

```

1  public class Receiver2Act extends Activity implements OnClickListener {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main_view);
6          //获取按钮组件并设置单击侦听器
7          Button btnSend = (Button)findViewById(R.id.BTN_SEND);
8          btnSend.setOnClickListener(this);
9      }
10     @Override
11     public void onClick(View v) { //发送广播

```

```
12      .....  
13    }  
14  };
```

相比代码 3-8，代码 3-12 要精简不少，其只存在发送广播的操作，而无需对广播接收器进行注册和注销，因为该操作已经由应用程序框架自动完成。

(3) 发送广播

广播接收器注册完毕后就可“收听”广播，除了系统可发送广播外，用户定义的 Activity 组件也可发送广播。代码 3-13 是代码 3-8 中发送广播的主要代码。

代码 3-13 发送广播

文件名: ReceiverAct.java

```
1  private void doSend() { //发送广播  
2      Intent sendIntent = new Intent(FooReceiver.class.getName());  
3      sendIntent.putExtra(IConfig.EXTRA, FooSysUtil.getInstance().getTsp());  
4      this.sendBroadcast(sendIntent);  
5  }
```

在代码 3-13 中，使用 Activity 组件的 sendBroadcast 方法发送广播，其参数为意向，该对象中包含了广播接收器组件信息（第 2 行）和数据内容（第 3 行）。

2. 广播接收器组件定义框架

代码 3-14 是代码 3-8 中提到的广播接收器组件的定义。

代码 3-14 广播接收器组件的定义

文件名: FooReceiver.java

```
1  public class FooReceiver extends BroadcastReceiver {  
2      @Override  
3      public void onReceive(Context context, Intent intent) {  
4          String msg = intent.getStringExtra(IConfig.EXTRA);  
5          Toast.makeText(context, "广播内容: "+msg, Toast.LENGTH_LONG).show();  
6      }  
7  };
```

与 Activity 服务组件一样，广播接收器组件的定义要遵循 Android 应用程序框架：所有广播接收器组件的定义必须继承其父类 BroadcastReceiver，在其所重载的广播接收方法（第 3 行的 onReceive 方法）中实现对广播的过滤和接收。

提示：在代码中注册广播接收器和在程序清单中定义广播接收器这两种方式都可实现广播消息的接收。这两种使用方式的最大区别在于广播接收器的初始化方式：对于在代码中注册广播接收器的方式，用户可定制该接收器的初始化方式，如传递初始化参数等，典型的是传递主线程消息队列处理器（Handler）实例，由此实现将接收器接收内容传递到主线程界面中；而通过程序清单定义广播接收器的初始化过程由 Android 平台自动完成，用户无法干



预，但是该方式无需显式地对广播接收器进行注册及注销。

3.2.4 内容提供者组件（Content Provider）——奉献者

对于内容提供者，顾名思义，该组件用于给其他组件提供内容（数据），是当之无愧的奉献者。内容提供者组件无需可视控件，也无需与用户进行交互。通常情况下，需要数据的组件按照约定方式从内容提供者获取数据，继而利用可视控件显示这些数据。图 3-10 是内容提供者使用方式的示意图。

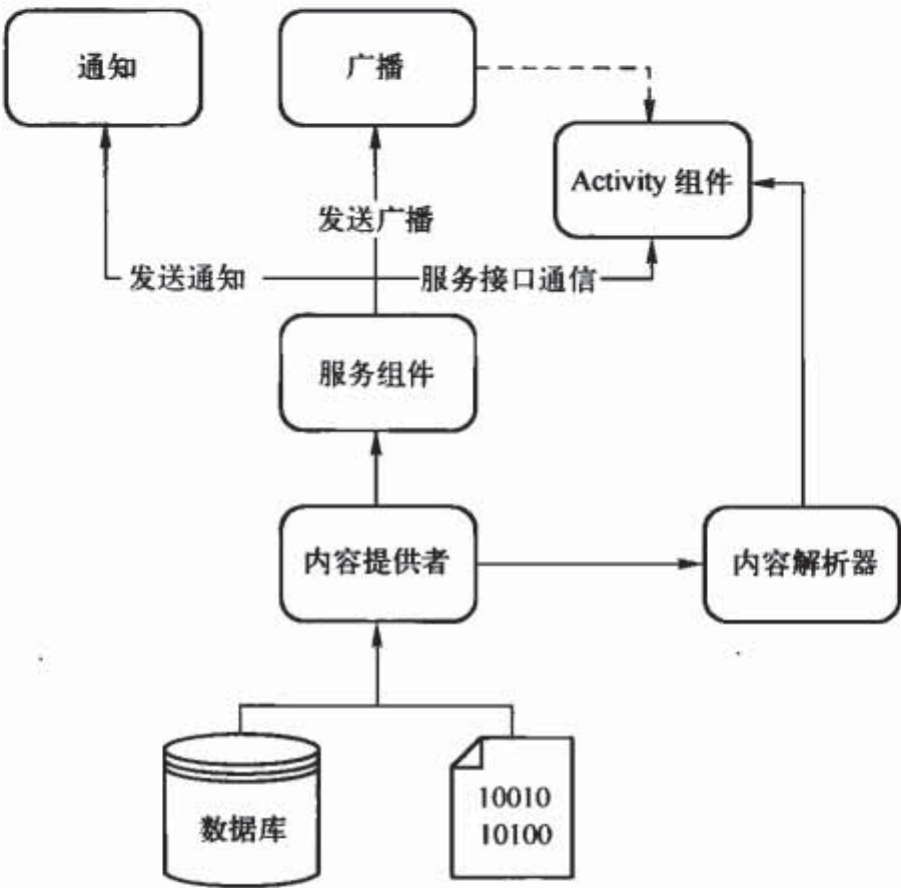


图 3-10 内容提供者使用方式示意图

图 3-11 是一个从内容提供者组件读取数据并显示内容的 Activity 组件界面。

1. 内容提供者的使用方式

代码 3-15 是图 3-11 中 Activity 组件的定义内容。在定义中，Activity 组件利用其关联对象内容解析器（ContentResolver）按内容提供者约定的资源标识执行查询请求，获得内容提供者所定义的数据内容。



图 3-11 内容提供者示例程序界面

代码 3-15 内容提供者应用程序 Activity 组件的定义

文件名: ProviderAct.java

```
1 //内容解析器
2 private ContentResolver mResolver = null;
3 private LinearLayout mLayout = null;
4
5 @Override
```



```

6 public class ProviderAct extends Activity implements OnClickListener{
7     public void onCreate(Bundle savedInstanceState) {
8         super.onCreate(savedInstanceState);
9         setContentView(R.layout.main_view);
10        //获取文本框对象
11        this.mLayout = (LinearLayout) findViewById(R.id.layoutPanel);
12        this.mLayout.setOrientation(LinearLayout.VERTICAL);
13        //获取按钮对象并设置单击事件回调
14        Button btnQuery = (Button)findViewById(R.id.BTN_QUERY);
15        btnQuery.setOnClickListener(this);
16        //初始化内容解析器
17        mResolver = this.getContentResolver();
18    }
19    @Override
20    public void onClick(View v) {
21        doQuery();
22    }
23    .....
24 };

```

在代码 3-15 中，Activity 组件使用 `getContentResolver` 方法获取内容解析器对象（第 17 行）。代码 3-16 是响应按钮单击事件的回调方法。

代码 3-16 按钮单击事件回调方法

文件名: ProviderAct.java

```

1 private void doQuery() { //执行查询
2     //生成资源全路径
3     Uri uri = ContentUris.withAppendedId(FooDataProvider.CONTENT_URI,
4                                           FooDataProvider.ALL_ROWS);
5     //不能进行类型转换，否则会抛出类转换异常
6     Cursor c = mResolver.query(uri, null, null, null, null);
7     if(c == null) {
8         Toast.makeText(this, "获取游标失败！", Toast.LENGTH_LONG).show();
9         return;
10    }
11    final double colWidths[] = { 0.2f, 0.5f, 0.3f};
12    FooTableViewController wrapper = new FooTableViewController(this, c, colWidths);
13    initColumnTitleMapping(wrapper);
14    //设置消息处理器（处理器由调用方创建）
15    //wrapper.setHandler(this.mHandler);
16    this.mLayout.addView(wrapper.getMainWidget());
17    //关闭游标
18    c.close();
19 }

```

在代码 3-16 中，使用内容解析器的 `query` 方法执行记录查询操作，用以获取记录游标（第 6 行），最后使用 `FooTableViewController` 组件将游标中的内容进行展示，即图 3-11 所示



的内容。

应该注意，内容解析器的 `query` 方法的第一个参数是数据源的统一资源标识（Uniform Resource Identifier, URI），而该统一资源标识由参考数据提供者（`FooDataProvider`）定义（第3行）。

2. 内容提供者组件框架

代码3-17是代码3-16中提到的内容提供者组件的框架定义。

代码3-17 内容提供者组件的框架定义

文件名: `FooDataProvider.java`

```

1  public class FooDataProvider extends ContentProvider {
2      public static final String URI_AUTHORITY = FooDataProvider.class.getName();
3      public static final String URI_PATH = "rs";
4      public static final String URI_PATH2 = "rs/#";
5      public static final int ALL_ROWS = 0;
6      public static final int SINGLE_ROW = 1;
7      //该 URI 的授权部分必须为有效类的全名
8      public static final Uri CONTENT_URI =
9          Uri.parse("content://" + URI_AUTHORITY + "/" + URI_PATH);
10     //构造一个矩阵游标作为数据源
11     public static MatrixCursor mCursor = new MatrixCursor(IDbSpec.COLUMN_NAMES);
12     //初始化 URI 匹配器
13     public static final UriMatcher uriMatcher;
14     static {
15         uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
16         uriMatcher.addURI(URI_AUTHORITY, URI_PATH, ALL_ROWS);
17         uriMatcher.addURI(URI_AUTHORITY, URI_PATH2, SINGLE_ROW);
18     }
19     @Override
20     public boolean onCreate() { //初始化数据内容
21         mCursor.addRow(new String[] { "1", "Paul", "Female" });
22         mCursor.addRow(new String[] { "2", "Leo", "Male" });
23         return true;
24     }
25     @Override
26     public Cursor query(Uri uri, String[] projection, String selection,
27         String[] selArgs, String sortOrder) { return (mCursor); }
28     @Override
29     public int delete(Uri uri, String selection, String[] selectionArgs) { return 0; }
30     @Override
31     public String getType(Uri uri) { return null; }
32     @Override
33     public Uri insert(Uri uri, ContentValues values) { return null; }
34     @Override
35     public int update(Uri uri, ContentValues values, String selection, String[] selArgs) {
36         return 0;

```



```

37     }
38 };

```

在代码 3-17 中，内容提供者组件的定义也必须遵照 Android 平台应用程序框架：所有内容提供者的定义必须继承父类 `ContentProvider`，在所重载的方法中实现对数据记录的操作，包括删除（第 29 行）、插入（第 33 行）、更新（第 35 行）和查询（第 26 行）等。

代码 3-17 中所定义的一些静态成员以及初始化（第 2~13 行）都是应用程序框架所要求的。

代码 3-18 是代码 3-17 中提到的数据规格接口（`IDbSpec`）的定义。

代码 3-18 数据规格接口的定义

文件名: `IDbSpec.java`

```

1  public interface IDbSpec {
2      public static final String _ID = "_id";
3      public static final String FIELD_NAME = "name";
4      public static final String FIELD_SEX = "sex";
5      //列编号
6      public static final int INDEX_ID = 0;
7      public static final int INDEX_NAME = 1;
8      public static final int INDEX_SEX = 2;
9      //列名数组
10     public static final String COLUMN_NAMES[] = new String[] {
11         _ID, FIELD_NAME, FIELD_SEX
12     };
13 };

```

3. 声明内容提供者

代码 3-19 是在应用程序清单中声明内容提供者的 XML 代码。

代码 3-19 在应用程序清单中声明内容提供者

文件名: `AndroidManifest.xml`

```

1  <provider android:name=".FooDataProvider" android:label="@string/provider"
2      android:authorities="@string/authorities"/>

```

在代码 3-19 中，内容提供者组件的定义除了需要指明类名（`android:name` 属性）之外，还需要授权标识（`android:authorities` 属性所指）。代码 3-20 是内容提供者组件的授权标识所引用的字符串资源的定义。

代码 3-20 字符串资源的定义

文件名: `strings.xml`

```

- 1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <string name="app_name">内容提供者示例</string>
4      <string name="authorities">foolstudio.demo.app.provider1.FooDataProvider</string>

```




```
5      <string name="provider">内容提供者示例</string>
6  </resources>
```

通过代码 3-20 可以看出，内容提供者组件的授权标识实际上就是代码 3-17 第 11 行所定义的 URI 授权标识常量，即为提供者类全名。

3.2.5 Android 应用程序组件小结

不同类型的应用程序组件造就了丰富多彩的 Android 平台应用，这些组件中，既有热衷于表现的“形象大使”（Activity 组件），也有习惯于埋头苦干的“老黄牛”（服务组件）；有冷静的“倾听者”（广播接收器组件），也有热情的“奉献者”（内容提供者组件）。各种组件各有所长，从而满足 Android 平台大部分的应用场合。

3.3 组件应用机制

在对 Android 平台 4 种应用程序组件的介绍中，涉及有关组件之间的相互调用、通信等应用。而这些应用机制正是关联所有组件的纽带，有了纽带，就能把各种不同类型、不同功能的应用组件“捆绑”在一起，从而帮助用户完成各种复合的应用。

在 Android 平台中，常用的应用机制有 Activity 组件与 Activity 组件、Activity 组件与线程以及 Activity 组件与服务组件之间的交互机制。

3.3.1 组件间的纽带——意向

在 Android 平台中，Activity 组件之间的交互通过意向（Intent）来实现。表 3-4 是对意向的说明。在 SDK 的参考中，Android 平台把意向归纳为激活组件，就是用于激活其他组件。这里把意向归集到应用机制进行讲解，主要目的是为了读者区分地理解组件特性和组件的使用方式。

表 3-4 应用程序组件类/接口的说明

类/接口	说 明
android.content.Intent	意向，用于描述将动作的执行

在图 3-12 所示的应用程序中，主 Activity 启动一个新的 Activity，并将数据记录传递给该 Activity 组件进行显示。

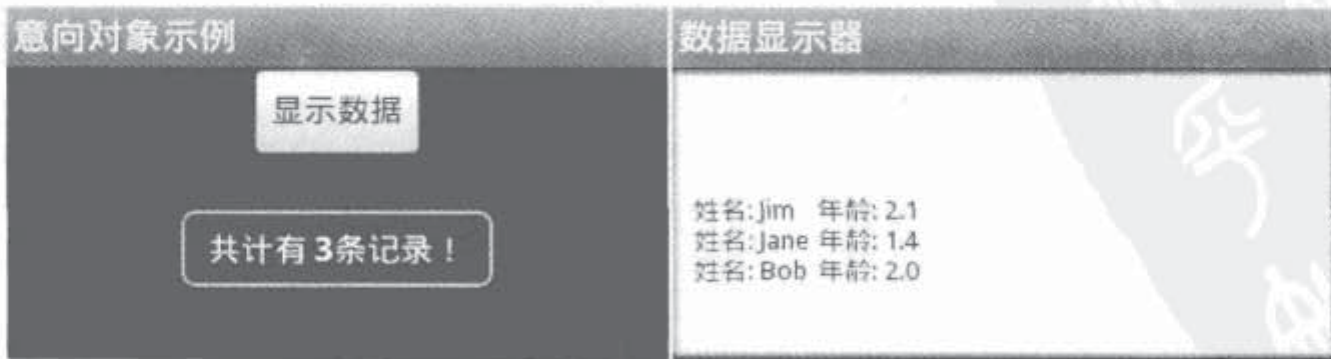


图 3-12 启动新的 Activity 组件

代码 3-21 是图 3-12 所对应的程序的主 Activity 定义，其主要功能是提供程序主界面，通过按钮启动另外一个 Activity 组件。

代码 3-21 主 Activity 组件的定义

文件名: IntentAct.java

```

1  public class IntentAct extends Activity implements OnClickListener {
2      /* Android 平台禁止访问 Activity 的构造函数，否则抛出异常
3      public static IntentDemoAct mInstance = new IntentDemoAct();
4      private IntentDemoAct() { }
5      public static IntentDemoAct getInstance() { return (mInstance); }
6      */
7
8      @Override
9      public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.main_view);
12         //获取按钮控件并设置单击事件回调函数
13         Button btnStart = (Button)findViewById(R.id.BTN_START);
14         btnStart.setOnClickListener(this);
15     }
16     .....
17 };

```

代码 3-21 中有一段注释了的代码（第 3~5 行），其本意是：主 Activity 以单例（Singleton）的形式向外部组件提供获取 Activity 实例的接口（第 3 行）；被调的 Activity 组件通过主 Activity 提供的接口获取主 Activity 实例，继而通过其数据访问接口获取数据并显示。而事实上，通过单例模式来实现 Activity 组件之间数据共享的想法在 Android 平台行不通，因为 Android 平台禁止应用程序组件访问 Activity 组件的构造函数，其异常输出如下所示。

```

java.lang.RuntimeException: Unable to instantiate activity ComponentInfo
java.lang.IllegalAccessException: access to constructor not allowed

```

实际上，Android 平台所定义的这一禁止规则，是用来规范应用程序组件之间的数据传递方式，保证组件间数据传递的统一和安全。

1. 调用子 Activity 组件

在 Android 平台中，Activity 组件使用 `startActivity` 或 `startActivityForResult` 方法来调用子 Activity 组件，两个方法的第一个参数都是意向，数据通过意向的扩展空间进行传递。代码 3-22 是按钮单击事件的回调方法，该方法执行对子 Activity 组件的调用。

代码 3-22 调用子 Activity 组件

文件名: IntentAct.java

```

1  @Override
2  public void onClick(View v) { //启动新的活动

```




```
3      Intent startNew = new Intent(this, DataViewerAct.class);
4      ArrayList<Kid> kids = initArrayList();
5      startNew.putParcelableArrayListExtra(IConfig.EXTRA1, kids);
6      //调用数据显示 Activity 并获取反馈结果
7      this.startActivityForResult(startNew, IConfig.REQ_CODE);
8  }
```

在代码 3-22 中，使用 `startActivityForResult` 方法调用子组件，因为调用方 `Activity` 还需要获取调用结果，该方法的第二个参数是请求码，用来识别反馈结果是否为预期（第 7 行）。代码 3-23 是代码 3-22 中初始化数据容器（第 4 行）的主要代码。

代码 3-23 初始化数据容器

文件名: `IntentAct.java`

```
1  private ArrayList<Kid> initArrayList() { //初始化列表
2      ArrayList<Kid> kids = new ArrayList<Kid>();
3      kids.add(addKid("Jim", 2.1f));
4      kids.add(addKid("Jane", 1.4f));
5      kids.add(addKid("Bob", 2.0f));
6
7      return kids;
8  }
9
10 private Kid addKid(String name, float age) { //添加小孩记录
11     Kid kid = new Kid(name, age);
12     return (kid);
13 }
```

2. 意向的内涵

既然 `Activity` 组件可通过意向来调用，那么在该意向中，应该包含以下内容：

- (1) 组件信息，“告诉”平台由谁去执行任务。
- (2) 数据的资源标识或数据容器，明确所要处理的内容。
- (3) 对内容的动作方式，如浏览、编辑、插入等。

图 3-13 是意向的内容结构示意图。

在代码 3-22 第 5 行，意向使用 `putParcelableArrayListExtra` 方法将一个包含多条小孩（`Kid`）记录的列表作为数据项添加到该意向的扩展数据中，该数据项以字符串（`IConfig.EXTRA1`）作为主键，由此可推断意向的扩展空间类似于散列表容器。

3. 扩展数据的定义

既然 `Activity` 组件之间无法简单地通过内存共享来实现数据的互访，而只能通过意向的扩展数据进行传递，那么扩展数据所包含的内容又是以一种什么机制实现传递的呢？代码 3-24 是代码 3-23 中小孩记录的定义。

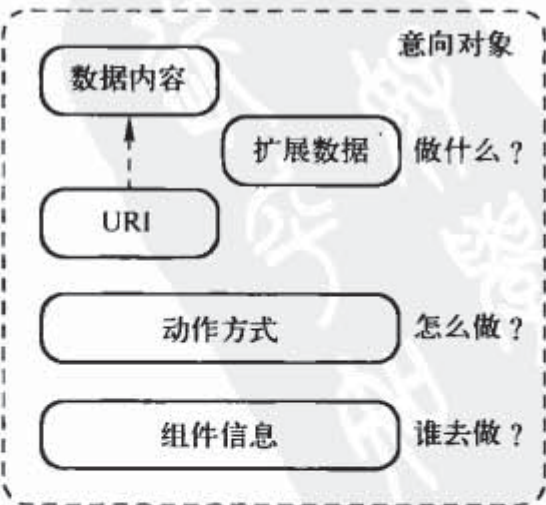


图 3-13 意向内容结构

代码 3-24 意向扩展数据项小孩记录的定义

文件名: Kid.java

```
1  public class Kid implements Parcelable {
2      private String mName = null;
3      private float mAge = 0.1f;
4
5      //必须有一个名为 CREATOR 的成员对象, 否则无法进行 Parcelable 对象通信
6      public static final Parcelable.Creator<Kid> CREATOR =
7          new Parcelable.Creator<Kid>() {
8              public Kid createFromParcel(Parcel in) { return new Kid(in); }
9              public Kid[] newArray(int size) { return new Kid[size]; }
10         };
11
12     public Kid(String name, float age) {      this.mName = name;
13                                                this.mAge = age;
14     }
15
16     public void setName(String name) { this.mName = name; }
17     public String getName() { return this.mName; }
18     public void setAge(float age) { this.mAge = age; }
19     public float getAge() { return this.mAge; }
20
21     public String toString() {
22         return ("姓名: "+this.getName()+"\t"+"年龄: "+this.getAge());
23     }
24
25     //实现 Parcelable 接口
26     public Kid(Parcel in) { this.mName = in.readString();
27                             this.mAge = in.readFloat();
28     }
29
30     @Override
31     public int describeContents() { return 0; }
32     @Override
33     public void writeToParcel(Parcel dest, int flags) {      dest.writeString(this.getName());
34                                                                dest.writeFloat(this.getAge());
35     }
36 };
```

在代码 3-24 中, 读者也许会觉得该定义过于烦琐, 特别是对其中 CREATOR 成员以及构造方法的定义形式有点不知所云。而恰恰是这些莫名其妙的属性和方法, 使数据在组件之间传递成为可能。

Android 平台对可通过进程间通信 (Inter-process Communication, IPC) 机制进行传递的数据定义进行约定: 这些数据类必须实现 Parcelable 接口, 且必须包含一个类型为 Parcelable.Creator 且名为 CREATOR 的公共静态成员。如果所定义的数据类不包含



CREATOR 成员，则会抛出如下的运行时异常信息：

android.os.BadParcelableException: Parcelable protocol requires a Parcelable.Creator object called CREATOR on class foolstudio.demo.Kid

代码 3-24 中的 Parcelable 接口就是代表可通过包裹 (Parcel) 进行数据传递的功能特性。只有实现 Parcelable 接口的类才能以意向的扩展数据进行传递，实际上，Android 平台中大部分的信息类都实现了 Parcelable 接口，包括意向类本身。

实现于 Parcelable 接口的 CREATOR 成员的 createFromParcel 方法用于“告诉”平台如何从包裹创建该类的实例；而 writeToParcel 方法用于“告诉”平台如何将该类的实例存储到包裹中。通过对接口和成员进行约定，Android 平台可获知该数据类的数据读取和写入的接口，从而可进行对象的实例化（从包裹中创建类实例）和持久化（将类实例存储到包裹中），如图 3-14 所示。

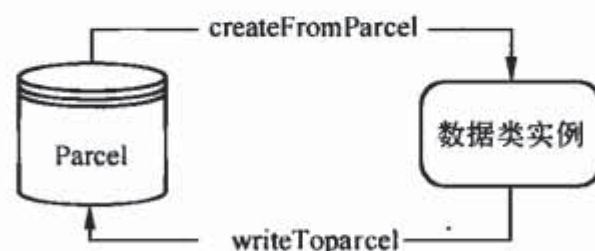


图 3-14 数据类实例与包裹之间的转换示意图

4. 子 Activity 组件接收数据

在代码 3-22 中，主 Activity 组件将小孩记录数组以意向的扩展数据传递给数据显示 Activity 组件。代码 3-25 是显示数据的 Activity 组件的框架定义。

代码 3-25 子 Activity 组件的框架定义

文件名：DataViewerAct.java

```

1  public class DataViewerAct extends Activity {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.data_viewer_view);
6          //获取显示组件
7          EditText txtData = (EditText)findViewById(R.id.TXT_DATA);
8          //获取父 Activity 所传递的附加数据
9          Intent in = this getIntent();
10         ArrayList<Kid> kids = in.getParcelableArrayListExtra(IConfig.EXTRA1);
11         //ArrayList<Kid> kids = IntentDemoAct.getInstance().getArrayList();
12
13         for(int i = 0; i < kids.size(); ++i) { //遍历记录
14             Kid kid = kids.get(i);
15             txtData.append(kid.toString()+"\n");
16         }
17
18         //返回结果
19         Intent reply = new Intent();
20         reply.putExtra(IConfig.EXTRA2, "共计有 " + kids.size() + "条记录！");
21         this.setResult(IConfig.REQ_CODE, reply);
22     }
23 };

```


在代码 3-25 中, 该 Activity 组件使用 `getIntent` 方法得到与之关联的意向实例 (第 9 行), 然后使用意向实例的 `getParcelableArrayListExtra` 方法从扩展数据中按主键名进行检索, 获取包含小孩记录的记录数组对象 (第 10 行), 继而进行记录展示。该代码中从意向读取记录与代码 3-22 中往意向填充记录的过程是互逆的。

5. 调用状态反馈与接收

在介绍 Activity 组件被调用的方式时谈到, 使用 `startActivityForResult` 方法调用 Activity 组件, 被调用的 Activity 可将结果反馈给调用方 Activity。

在代码 3-25 中, 第 19~21 行就是将结果反馈给调用方 Activity 的代码。子 Activity 反馈结果给调用方也需要使用意向 (第 19 行), 反馈信息项以“键—值”的方式添加到该意向的扩展数据中 (第 20 行), 进而使用 `setResult` 方法反馈结果, 其中反馈的结果码 (第 21 行) 即是调用该 Activity 时的请求码 (代码 3-22 中第 7 行)。

既然被调 Activity 有反馈, 那么主 Activity 组件则会对反馈结果进行捕获。代码 3-26 是在主 Activity 中处理子 Activity 组件调用反馈的回调方法。

代码 3-26 获取子 Activity 组件调用反馈的结果

文件名: IntentAct.java

```

1  @Override
2  protected void onActivityResult(int requestCode, int resultCode, Intent data) {
3      if(requestCode == resultCode) {
4          Bundle bundle = data.getExtras();
5          String extras = bundle.getString(IConfig.EXTRA2);
6          Toast.makeText(this, extras, Toast.LENGTH_LONG).show();
7      }
8      super.onActivityResult(requestCode, resultCode, data);
9  }
```

在代码 3-26 中, 调用方 Activity 组件 (主 Activity) 通过回调方法 `onActivityResult` 来获取所收到的反馈结果, 并进行数据项提取。

6. 程序清单

通过意向, 开发者可轻松地实现在 Activity 组件之间的交互。而实际上, 被调用 Activity 组件需要事先“告诉”Android 平台, 否则运行时将会抛出目标 Activity 组件没有找到的异常信息, 如下所示。

`android.content.ActivityNotFoundException: Unable to find explicit activity class have you declared this activity in your AndroidManifest.xml?`

Activity 组件通过在程序清单中进行“登记”来“告诉”Android 平台。代码 3-27 是该意向示例程序的程序清单内容。

代码 3-27 意向示例程序清单

文件名: AndroidManifest.xml

```

1  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2      package="foolstudio.demo.app.intent1"
```




```

3  <?xml version="1.0" encoding="utf-8"?>
4      android:versionCode="1"
5      android:versionName="1.0">
6      <application android:icon="@drawable/icon" android:label="@string/app_name">
7          <activity android:name=".IntentAct" android:label="@string/app_name">
8              <intent-filter>
9                  <action android:name="android.intent.action.MAIN" />
10                 <category android:name="android.intent.category.LAUNCHER" />
11             </intent-filter>
12         </activity>
13         <activity android:name=".DataViewerAct" android:label="@string/viewer"/>
14     </application>
15     <uses-sdk android:minSdkVersion="8" />
16 </manifest>

```

在代码 3-27 中，第 7 行和第 13 行共声明了两个 Activity 组件，但是使用<intent-filter>标记。读者可看出，第 7 行定义的 Activity 组件是主 Activity，作为启动用。

需要补充的是，无论是主 Activity 还是子 Activity 组件，都需要绑定可视界面，所以还需要为这些 Activity 组件分别定义界面布局。

3.3.2 组件间的预约——未决意向

如果说意向描述的是即将执行的动作（该动作会马上被执行），那么未决意向（PendingIntent）描述的却是可能将要执行的动作（该动作可能会被取消），如设定闹钟、发送短信、提醒通知等。

读者可把未决意向理解为带有条件的意向，未决意向实例需要依据意向来获取。使用 PendingIntent 类的静态方法 getActivity、getBroadcast 和 getService 可分别获取用于启动 Activity、发送广播和启动服务的未决意向实例。

未决意向与意向一样，本身不用于执行操作，而是必须由调用主体依照未决意向的描述启动操作。未决意向所定义的操作一般不会立即执行，往往需要满足一定条件，如在指定时刻或超过指定延时等。代码 3-28 是未决意向用于通知的示例代码。

代码 3-28 未决意向使用示例

文件名: NotificationsDemoAct.java

```

1  //创建通知（默认内容视图）
2  private Notification makeNotification1() {
3      Notification.Builder builder = new Notification.Builder(this);
4      builder = builder.setSmallIcon(R.drawable.info);
5      builder = builder.setTicker("通知");
6      //5s 后
7      builder = builder.setWhen(System.currentTimeMillis()+5000L);
8      //设置内容意向
9      Intent intent = new Intent(this, NotificationAct.class);
10     PendingIntent contentIntent = PendingIntent.getActivity(this,0,intent,0);

```



```
11     builder = builder.setContentIntent(contentIntent);
12     //返回通知对象
13     return (builder.getNotification());
14 }
```

在代码 3-28 中，消息创建器在设置内容意向时，先创建一个意向，并指定执行行动的 Activity 组件（第 9 行），然后再由意向创建一个未决意向（第 10 行），该未决意向用于表述当该通知被查看时的行动内容。

3.3.3 与线程的交互——线程消息队列处理器

有的开发者可能这样使用线程：通过可视控件（如按钮）启动线程，然后在线程的执行代码中将线程状态信息输出到用户界面控件（如文本框）。在 J2SE 平台中，这样的使用方式可能不会遇到什么问题，但是在 Android 平台中，将会抛出以下的异常信息。

android.view.ViewRoot\$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.

该异常的意思是，只有创建视图层次结构的原始线程才能存取该结构中的视图，其言外之意就是：不是创建界面的原始线程是不能存取界面元素的。那么，在不是创建界面的线程（主线程）中，如何将线程状态输出到用户界面中呢？

1. 线程消息队列处理器

Android 平台提供线程消息队列（Message Queue）的机制来解决上述遇到的问题。首先在主线程中创建一个消息队列接口，然后其他非主线程通过这个接口将消息发送到主线程的消息队列中，最后由主线程将消息内容展示到用户界面中。这个接口就是一个线程消息队列处理器（Handler），线程消息队列处理器就像一个嵌入线程消息队列的楔子，为线程之间的交互开启了方便之门，如图 3-15 所示。

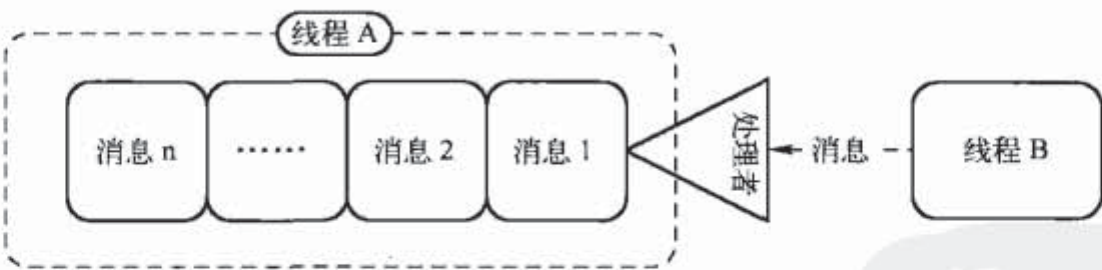


图 3-15 线程间通过消息队列处理器进行交互

图 3-16 是一个 Activity 主线程与外部线程交互的示例程序的运行界面。



图 3-16 Activity 主线程与外部线程交互的运行界面



代码 3-29 是图 3-16 中 Activity 主线程的框架定义，主要内容包括定义主线程消息队列处理器和启动外部线程。

代码 3-29 消息队列处理器示例程序 Activity 主线程的框架定义

文件名: HandlerDemoAct.java

```

1  public class HandlerAct extends Activity implements OnClickListener {
2      private EditText mTxtMsg = null;
3      private Handler mHandler = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main_view);
9          //文本输出框
10         mTxtMsg = (EditText)findViewById(R.id.TXT_MSG);
11         //获取按钮对象并设置单击事件回调
12         Button btnStart = (Button)findViewById(R.id.BTN_START);
13         btnStart.setOnClickListener(this);
14         //初始化线程消息处理器
15         mHandler = new Handler() {
16             @Override
17             public void handleMessage(Message msg) {
18                 //分解接收到的消息
19                 Bundle bundle = msg.getData();
20                 String from = bundle.getCharSequence(IConfig.KEY1).toString();
21                 String data = bundle.getString(IConfig.KEY2);
22                 //输出消息
23                 printLine(from+" | "+data);
24             }
25         };
26     }
27     @Override
28     public void onClick(View v) {
29         FooThread t = new FooThread(mHandler);
30         t.start();
31     }
32
33     public void printLine(String line) { mTxtMsg.append(line+"\n"); }
34 };

```

在代码 3-29 中，通过单击按钮控件来启动线程（第 30 行），并用文本框控件来显示外部线程所传递的消息（第 33 行）。

2. 线程消息队列的消息处理

在代码 3-29 中，定义了一个线程消息队列处理器（第 15~25 行），该处理器代表主线程的消息队列处理接口。在该处理器的定义中，`handleMessage` 方法是通过重载而来，用于

处理消息的接收。消息类（Message）代表线程消息队列中的消息，而消息中所包含的数据被存放在包容器（Bundle）中，这里的包容器类似于 Map，容器中的数据项以“键—值”的形式进行存放，以键来获取值。

3. 主线程消息的传递

在代码 3-29 第 29 行，Activity 组件创建了一个外部线程，并将消息队列处理器传递给该线程。代码 3-30 是该线程的定义。

代码 3-30 外部线程的定义

文件名：FooThread.java

```
1 public class FooThread extends Thread {
2     //主线程消息队列处理器
3     private Handler mHandler = null;
4
5     public FooThread(Handler handler) { this.mHandler = handler; }
6
7     @Override
8     public void run() {
9         //直接调用主线程方法
10        //mContext.println(FooSysUtil.getInstance().getTsp());
11        //将消息打包、发送
12        Bundle bundle = new Bundle();
13        bundle.putCharSequence(IConfig.KEY1, this.getName());
14        bundle.putString(IConfig.KEY2, FooSysUtil.getInstance().getTsp());
15        Message msg = new Message();
16        msg.setData(bundle);
17        mHandler.sendMessage(msg);
18    }
19 };
```

在线程的执行方法 run 中，首先创建一个包容器（第 12 行），并将消息内容以“键—值”的形式添加到该容器中（第 13 行和第 14 行）；再创建一个消息实例（第 15 行），并将填充好的包容器设置为该消息的数据内容（第 16 行）；最后通过消息队列处理器将该消息发送到主线程的消息队列中（第 17 行）。

代码 3-30 中的 run 方法与代码 3-29 中消息队列处理器的 handleMessage 方法的处理过程是互逆的，前者是将消息打包及发送，后者是将消息接收并分解。

3.3.4 与服务组件的交互——AIDL

对于在 Activity 组件中启动及关闭服务，Activity 与服务组件之间不存在交互，Activity 无法获取和控制服务组件的运行状态。而在大多数情况下，前台界面需要获取和控制后台服务组件的状态信息。例如，对于后台音乐播放服务，前端界面需要获知服务组件当前播放的内容信息，并控制其进行播放导航。基于这类应用需求，Android 平台定义了 Activity 与服务组件进行交互的机制。



1. AIDL 进程间通信机制

Android 平台提供了一套称为 AIDL 进程间通信的机制，可解决客户端组件与服务组件的接口访问。Android 接口定义语言（Android Interface Definition Language, AIDL）是经过扩展、适应 Android 平台的一种接口定义语言（IDL），ADT 插件可自动将 AIDL 所描述的内容生成对应的 Java 代码。

AIDL 进程间通信机制是一种基于接口、轻量级的机制，类似于 COM 或 CORBA。服务组件通过 AIDL 定义其向外界“暴露”的接口，而客户端通过绑定的方式可获取这些服务接口，通过调用这些服务接口而实现与服务组件进行交互，如图 3-17 所示。

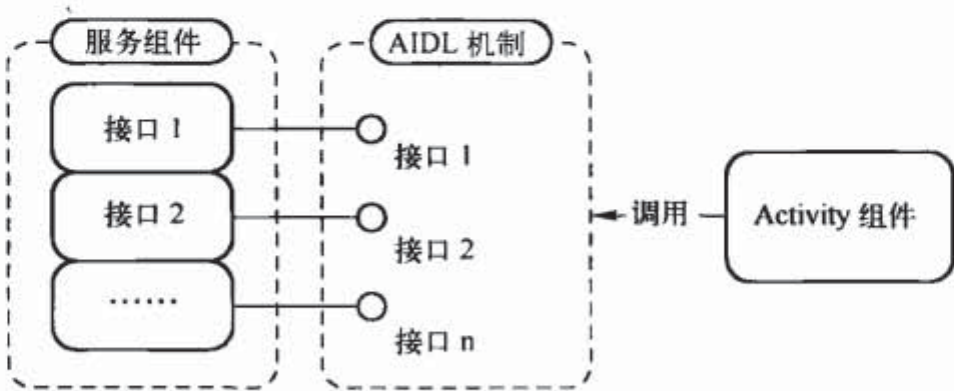


图 3-17 AIDL 机制示意图

提示：Android 平台底层提供了一种轻量级的、用于远程过程调用（Remote Procedure Calls, RPC）的机制。客户端组件在本地调用接口方法，但是该方法在远程组件中执行，并且将结果返回到客户端。详细解释请参考 SDK 文档 Processes and Threads 章节。实际上，Android 平台的这一机制与 J2SE 平台中的远程方法调用（Remote Method Invoke, RMI）机制类似，读者可结合 RMI 机制来了解 Android 平台的 RPC 应用模式。

2. 客户端 Activity 组件框架

图 3-18 是客户端 Activity 与后台服务组件通过连接的方式进行通信的示例界面。

代码 3-31 是客户端 Activity 组件的框架定义，其中主要包括绑定服务和取消绑定以及与服务组件进行通信。



图 3-18 Activity 与服务组件进行通信的界面

代码 3-31 客户端 Activity 组件的框架定义

文件名: AidlAct.java

```
1 public class AidlAct extends Activity implements OnClickListener {
2     .....
3     @Override
4     public void onCreate(Bundle savedInstanceState) {
5         super.onCreate(savedInstanceState);
6         setContentView(R.layout.main);
7         //获取按钮控件并设置单击侦听器
8         Button btnAction = (Button)findViewById(R.id.btn);
```



```

9      btnAction.setOnClickListener(this);
10     //绑定服务
11     this.bindService(new Intent(EchoService.class.getName() ),
12                     mConnection, Context.BIND_AUTO_CREATE);
13 }
14 @Override
15 protected void onDestroy() {
16     this.unbindService(mConnection); //解除绑定
17     super.onDestroy();
18 }
19 @Override
20 public void onClick(View v) {
21     switch(v.getId() ) {
22         case R.id.btn: { doShakeHand(); break; }
23     }
24 }
25 .....
26 };

```

在代码 3-31 中，当 Activity 创建时绑定服务（第 11 行）；在其销毁时取消绑定（第 16 行）。服务绑定后使用按钮触发与服务组件的通信。

使用 Activity 组件的 `bindService` 方法可让当前 Activity 组件连接到指定服务，该方法的第一个参数是服务组件的类全名，用于定位服务组件；其第二个参数是服务连接接口，该接口用于提供客户端的回调处理；第三个参数是绑定的操作选项，代码中的 `BIND_AUTO_CREATE` 标志是指自动创建服务组件并绑定。

使用 Activity 组件的 `unbindService` 方法可解除与服务组件的绑定，其参数有且仅有服务连接接口（第 16 行）。服务连接接口在客户端定义，代码 3-32 是该接口的定义。

代码 3-32 服务连接接口的定义

文件名: AidlAct.java

```

1  //服务接口
2  private IEchoService mService = null;
3  //服务连接接口
4  private ServiceConnection mConnection = new ServiceConnection() {
5      @Override
6      public void onServiceConnected(ComponentName name, IBinder service) {
7          //获取服务接口
8          mService = IEchoService.Stub.asInterface(service);
9      }
10     @Override
11     public void onServiceDisconnected(ComponentName name) {
12         //取消绑定
13         mService = null;
14     }
15 };

```




Android 平台开发之旅 第2版

从代码 3-32 可看出，服务连接接口主要用于提供对连接和断开事件的回调。当连接成功时，使用服务接口的存根可获取服务接口（第 8 行）；当断开时，释放服务接口。

提示：有关存根（Stub）的定义，在 J2SE 平台的 RMI 机制中有说明。存根是一种中间接口，用于对客户端与服务端之间的远程方法进行整理（包括方法的描述和编码）和传输。在传输方面要求客户端和服务端的数据内容是可使用“包裹”来传递的，即这些数据类都必须实现 Parcelable 接口。

在 J2SE RMI 机制中，远程接口定义中返回值的类型必须为支持序列化（Serializable）的实体类，用户自定义类型如果需要通过 RMI 机制进行传递，那么该类也必须实现 Serializable 接口。从通信的角度而言，无论是 Parcelable 还是 Serialization 接口，都是视为一种协议规范，通过这种协议规范，客户端才能与远程服务端进行信息交换。

与服务组件连接成功之后，使用服务接口调用接口方法即可实现与服务端的交互，如代码 3-33 所示。

代码 3-33 与服务组件进行通信

文件名: AidlAct.java

```

1 private void doShakeHand() { //进行交互
2     String echo = null;
3     final EditText editor = (EditText)findViewById(R.id.TXT_NAME);
4     try { echo = mService.getEcho(editor.getText().toString().trim());
5     } catch (RemoteException e) { e.printStackTrace(); }
6     //显示服务端的回复
7     Toast.makeText(this, echo, Toast.LENGTH_LONG).show();
8 }

```

在代码 3-33 中，Activity 组件使用服务接口调用 getEcho 方法来实现对服务功能的调用（第 4 行），该方法在服务端被作为服务接口进行定义。

3. 服务接口的定义

代码 3-34 是使用 AIDL 定义的服务接口，该接口中只定义一个方法 getEcho。需要注意的是，代码 3-34 不是 Java 代码，而是 aidl 代码，其文件以 aidl 作为扩展名。

代码 3-34 服务接口的定义

文件名: IEchoService.aidl

```

1 package foolstudio.demo.service;
2
3 interface IEchoService {
4     String getEcho(String caller);
5 }

```

ADT 插件会自动解释 aidl 文件，并生成一个与 aidl 文件名称相同且语言为 Java 的接口定义文件，该文件在工程文件结构的 gen 文件夹中，如图 3-19 所示。

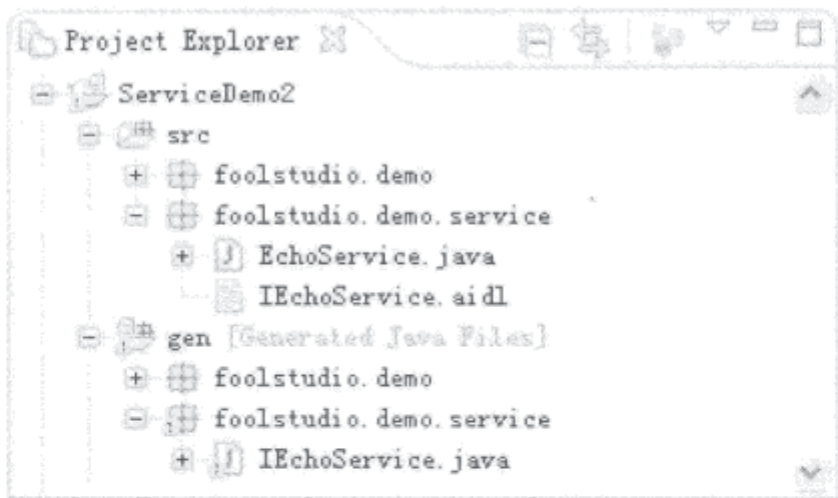


图 3-19 aidl 文件与自动生成文件

对于 AIDL 的语法，请参考有关 IDL 资料和 Android SDK 文档。

4. 服务组件

服务接口定义好了，还需由服务组件来实现才行。代码 3-35 是服务组件的定义。

代码 3-35 服务组件的定义

文件名: EchoService.java

```
1 public class EchoService extends Service {
2     //定义服务接口存根
3     private final IEchoService.Stub mBinder = new IEchoService.Stub() {
4         @Override
5         public String getEcho(String caller) throws RemoteException {
6             return ("你好, " + caller + "! ");
7         }
8     };
9
10    @Override
11    public IBinder onBind(Intent intent) {
12        if(EchoService.class.getName().equals(intent.getAction())) {
13            return (mBinder);
14        }
15        return null;
16    }
17 };
```

在代码 3-35 中，首先定义了一个服务接口存根（第 3 行），该存根在由 ADT 插件自动生成的文件中定义，其用于服务组件与客户端进行通信，将服务组件的接口方法以及返回值等通过协议规定的形式发送给客户端。

服务组件的 onBind 方法（第 11 行）为重载方法，其主要用于向客户端提供服务接口的存根。

提示：代码 3-35 中 onBind 方法的返回值是一个 IBinder 实例，IBinder 接口是一个远程对象的基本接口，描述了与远程对象进行交互的基本协议。但是 Android 平台建议不通过直接实现该接口来定义远程对象，而是通过继承 Binder 类。Binder 类是对 IBinder 接口的实



现，是远程对象的基类，该类是轻量级远程过程调用（RPC）机制的核心部分。大多数情况下，开发人员通过 `aidl` 来定义服务接口，然后由 `aidl` 工具生成对应的 `Binder` 子类。

3.3.5 与本地服务组件交互

有人可能认为 AIDL 机制过于烦琐，但是 AIDL 机制也是最为完整的解决方案，其适用于多线程和跨进程的场合。但事实上，作为移动应用，跨进程和多线程的场合并不算多，所以，Android 3/4 提供了两种与服务组件交互的简化方案：不跨进程的本地服务和单线程通信的信使服务。

图 3-20 是使用本地服务的方式下客户端 `Activity` 组件与服务组件进行交互的示例界面。

以本地服务组件的方式和以 AIDL 的方式获取服务接口的主要区别在于服务接口的获取，对于服务接口的使用框架基本相同，不再赘述。

1. 服务连接接口的定义

代码 3-36 是服务连接接口的定义。



图 3-20 与本地服务交互

代码 3-36 服务连接接口的定义

文件名: `BoundDemoAct.java`

```

1  //本地服务接口
2  protected FooService mService = null;
3  //服务连接接口
4  private ServiceConnection mConnection = new ServiceConnection() {
5      @Override
6      public void onServiceConnected(ComponentName className, IBinder service) {
7          //通过绑定器获取服务接口
8          mService = ((FooService.FooBinder)service).getService();
9      }
10     @Override
11     public void onServiceDisconnected(ComponentName className) {
12         mService = null;
13     }
14 };

```

代码 3-36 中没有使用服务接口的存根来获取服务接口，而是直接通过绑定器进行类型转换而获得本地服务绑定器，继而使用本地服务绑定器的 `getService` 方法获取服务接口（第 8 行）。

提示：因为不考虑跨进程，无需对远程方法的参数进行整理，所以本地服务通信无需使用存根。

2. 服务组件的定义

服务接口的获取取决于服务组件的定义。代码 3-37 是本地服务组件的定义。

代码 3-37 本地服务组件的定义

文件名: FooService.java

```

1  public class FooService extends Service {
2      //定义绑定器
3      class FooBinder extends Binder implements IBinder {
4          public FooService getService() { return (FooService.this); }
5      }
6
7      private final IBinder mFooBinder = new FooBinder();
8
9      @Override
10     public IBinder onBind(Intent intent) { return (mFooBinder); }
11
12     //获取服务端时间戳
13     public String getTsp() { return (FooSysUtil.getInstance().getTsp()); }
14 };

```

在代码 3-37 中, 读者可看出该服务组件通过绑定器来暴露其实例接口 (第 4 行), 而将绑定器使用 onBind 方法提供给客户端 (第 10 行), 所以才有代码 3-36 中先获取绑定器再通过其获取服务接口的操作。

3.3.6 客户端与服务端的桥梁——信使

如果服务接口需要跨进程通信, 那么使用本地服务组件的方式将无法适用。如果不考虑多线程的情况, 使用信使 (Messenger) 可满足与远程服务组件的交互。图 3-21 是使用信使来实现客户端 Activity 组件与服务组件交互的示例界面。

以信使的方式和以本地服务组件获取服务接口的主要区别在于服务接口的获取, 对于服务接口的使用框架基本相同, 不再赘述。

1. 服务连接接口的定义

代码 3-38 是服务连接接口的定义。



图 3-21 使用信使与服务交互的界面

代码 3-38 服务连接接口的定义

文件名: BoundDemo2Act.java

```

1  //本地服务接口
2  private Messenger mService = null;
3  //服务连接接口
4  private ServiceConnection mConnection = new ServiceConnection() {
5      @Override
6      public void onServiceConnected(ComponentName className, IBinder service) {
7          //获取服务接口 (信使)
8          mService = new Messenger(service);
9      }
10     @Override

```




Android 平台开发之旅 第2版

```

11      public void onServiceDisconnected(ComponentName className) {
12          mService = null;
13      }
14  };

```

在代码 3-38 中，通过绑定器初始化一个信使实例（第 5 行），客户端就是通过该信使与服务端进行通信。

代码 3-39 是按钮单击事件的回调处理，其主要内容是通过服务接口发送消息到服务端。

代码 3-39 按钮控件单击事件回调

文件名: BoundDemo2Act.java

```

1  @Override
2  public void onClick(View v) {
3      switch(v.getId()) {
4          case R.id.btnHello: { sendMessage(IMsgSpec.MSG_HELLO); break; }
5          case R.id.btnTsp: { sendMessage(IMsgSpec.MSG_TSP); break; }
6      }
7
8      private void sendMessage(int msgTsp) { //发送消息
9          Message msg = Message.obtain(null, msgTsp, 0, 0);
10         //设置消息的回复处理
11         msg.replyTo = this.mMessenger;
12         //发送消息
13         try { mService.send(msg); } catch (RemoteException e) { e.printStackTrace(); }
14     }

```

在代码 3-39 中，通过服务接口发送消息之前，对回复的信使进行了设置（第 11 行），服务端就是通过该信使与客户端进行交互。

2. 客户端信使的处理器定义

代码 3-40 是代码 3-39 中提到的客户端处理回复的信使的定义。

代码 3-40 客户端处理回复的信使的定义

文件名: BoundDemo2Act.java

```

1  //消息处理器
2  class FooHandler extends Handler {
3      @Override
4      public void handleMessage(Message msg) {
5          Bundle data = msg.getData();
6          switch(msg.what) {
7              case IMsgSpec.MSG_TSP: {
8                  println("时间戳: "+data.getString(IMsgSpec.EXTRA)); break;
9              }

```



```

10         case IMsgSpec.MSG_HELLO: {
11             printLine("问 候: "+data.getString(IMsgSpec.EXTRA)); break;
12         }
13     }
14 }
15 };
16
17 //信使
18 final Messenger mMessenger = new Messenger(new FooHandler());

```

至此，读者应该可初步猜测出如何使用信使来实现客户端与服务端的通信：客户端通过服务端的信使接口向服务端发送消息，并指明消息反馈的接收者为客户端信使；服务端处理完消息后，将结果再通过客户端的信使接口反馈给客户端，其机制如图 3-22 所示。

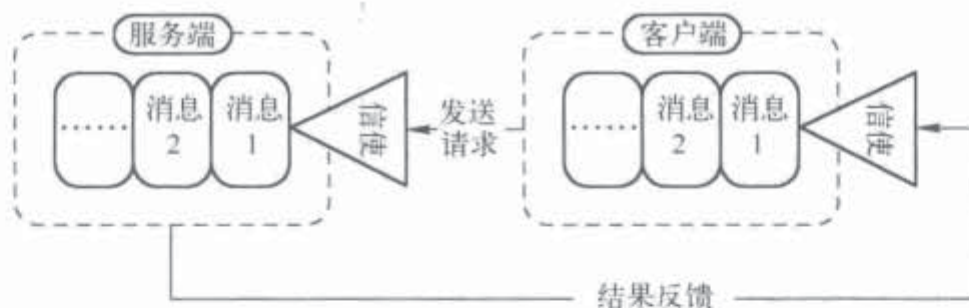


图 3-22 使用信使进行客户端与服务端通信

3. 服务组件的定义

代码 3-41 是服务端服务组件的定义。

代码 3-41 服务端服务组件的定义

文件名: FooService.java

```

1  public class FooService extends Service {
2      //消息处理器
3      class FooClientHandler extends Handler {
4          //处理客户端发过来的消息
5          @Override
6          public void handleMessage(Message msg) { respond(msg); }
7      };
8      //信使
9      final Messenger mMessenger = new Messenger(new FooClientHandler());
10
11      @Override
12      public IBinder onBind(Intent intent) { return (this.mMessenger.getBinder()); }
13
14      public void respond(Message msg) { //回复消息
15          Message reply = Message.obtain(null, msg.what, 0, 0);
16          Bundle data = new Bundle();
17          //根据所接收的消息标识组织反馈内容
18          switch(msg.what) {

```




```
19         case IMsgSpec.MSG_TSP: {
20             data.putString(IMsgSpec.EXTRA, FooSysUtil.getInstance().getTsp());
21             break;
22         }
23         case IMsgSpec.MSG_HELLO: {
24             data.putString(IMsgSpec.EXTRA, "你好! "); break;
25         }
26     }
27     //设置反馈消息内容
28     reply.setData(data);
29     //通过客户端信使接口发送消息
30     try { msg.replyTo.send(reply);
31         } catch (RemoteException e) { e.printStackTrace(); }
32 }
33 };
```

在代码 3-41 中，当服务端接收到客户端发送来的消息后，会依据消息的标识来准备返回的消息（第 18~26 行），最后通过客户端信使接口将反馈消息发送到客户端（第 30 行）。在 onBind 方法中，通过服务端信使接口的 getBinder 方法将其关联的绑定器“暴露”给客户端（第 12 行），当客户端成功连接到服务端后就可获取该接口（代码 3-38 第 8 行）。

3.4 Activity 组件关联对象

在前端展示方面，Activity 程序用于提供可视界面与用户进行交互；在后台功能方面，Activity 组件几乎与 Android 框架中的大部分重要对象存在关联。按照关联对象的特性，可分为资源处理、界面框架、内容提供、管理框架、环境信息和数据存储，如图 3-23 所示。



图 3-23 Activity 组件关联对象

3.4.1 资源处理相关

Android 平台所定义的资源类型相当丰富，无论是界面布局、菜单、图片、常量值等都可作为资源用 XML 进行定义；与此同时，Android 平台通过应用程序框架为 Activity 组件提供了一些内建对象用于帮助其引用所定义的资源，表 3-5 中是这些内建对象的说明。

表 3-5 资源相关对象的说明

对 象	说 明
布局填充器 (LayoutInflater)	用于填充布局资源定义，生成视图结构树
菜单填充器 (MenuInflater)	用于填充菜单资源定义，生成菜单对象实例
资源管理器 (Resources)	用于获取所定义的资源内容（如字符串、颜色值等）
资产管理器 (AssetManager)	用户获取所定义的资产内容（如数据文件等）

（有关“填充”的解释参见第 2 章）

1. 布局填充器 (LayoutInflater)

图 3-24 中就是使用布局填充器对布局资源进行填充后所展示的界面。



图 3-24 使用布局填充器填充的定制对话框界面

代码 3-42 是使用布局填充器填充布局资源的主要代码。

代码 3-42 使用布局填充器填充布局资源

文件名: ActivityCollectionsAct.java

```
1 private void showCustomDialog() {
2     //填充布局资源生成视图对象
3     View v = this.getLayoutInflater().inflate(R.layout.custom_dlg, null);
4     .....
5 }
```

在代码 3-42 中，Activity 组件使用其 `getLayoutInflater` 方法即可获得与之关联的布局填充器；使用布局管理器的 `inflate` 方法将布局资源定义填充为视图实例；`inflate` 方法的第一个参数是布局资源 ID，`R.layout.custom_dlg` 描述的是名为 `custom_dlg` 的布局资源定义文件，其存在于资源文件夹下的 `layout` 子文件夹中。代码 3-43 是该文件的内容。

代码 3-43 布局资源定义

文件名: custom_dlg.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
```




```
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="horizontal" android:layout_width="wrap_content"
4     android:layout_height="wrap_content">
5     <ImageView android:id="@+id/img" android:src="@drawable/icon"
6         android:layout_width="wrap_content" android:layout_height="wrap_content"/>
7     <TextView android:id="@+id/text" android:text="@string/about"
8         android:layout_width="240sp" android:layout_height="100sp"/>
9 </LinearLayout>
```

通过代码 3-43 中的内容以及图 3-24 的显示效果，读者应该可意会“填充”过程：根据定义布局资源的 XML 标记，生成对应的 Android 平台对象实例，如表 3-6 所示。

表 3-6 XML 标记及对应的 Android 平台对象

XML 标记	Android 平台对象
<LinearLayout>	线性布局（android.view.LinearLayout）
<ImageView>	图片视图（android.widget.ImageView）
<TextView>	文本视图（android.widget.TextView）

第 2 章中对于新手遇到的无法通过资源 ID 获取资源对象的问题，就涉及主布局资源的填充。代码 3-44 是 Activity 组件创建回调方法（onCreate）的定义。

代码 3-44 Activity 组件创建回调方法的定义

文件名：ActivityCollectionsAct.java

```
1 @Override
2 public void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.main);
5 }
```

第 4 行的 setContentView 方法实际上就是调用与该 Activity 组件相关的布局填充器去填充主布局资源（layout 文件夹下的 main.xml 资源定义文件），并按照布局定义生成视图对象的结构层次，继而通过图形界面进行展示。

2. 菜单填充器（MenuInflater）

菜单资源与布局资源的填充机制是相同的，只是在填充时机以及对填充后的对象（菜单或布局）的使用方式上存在一些差异。对于布局资源，开发者可自由决定其填充时机以及显示或隐藏；而对于菜单资源，必须在应用程序框架所定义的回调方法中填充，并由框架来控制其显示或隐藏。图 3-25 是选项菜单的实例界面；图 3-26 是上下文菜单的实例界面。

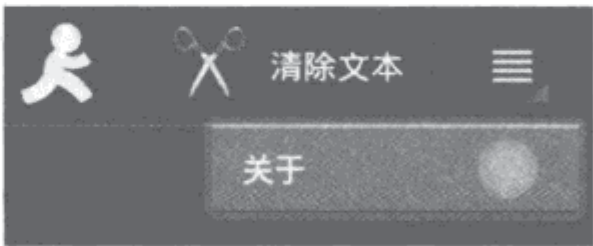


图 3-25 选项菜单的界面

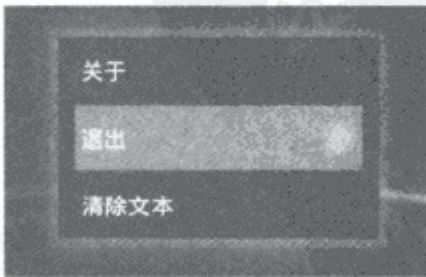



图 3-26 上下文菜单的界面

代码 3-45 是在 Activity 组件中使用菜单填充器填充上下文菜单和可选菜单的主要代码。

代码 3-45 使用菜单填充器填充菜单资源

文件名: ActivityCollectionsAct.java

```
1  @Override
2  public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo info) {
3      this.getMenuInflater().inflate(R.menu.ctx_menu, menu);
4      super.onCreateContextMenu(menu, v, info);
5  }
```

使用 Activity 组件的 `getMenuInflater` 方法可获取与之相关的菜单填充器（第 3 行），使用该接口的 `inflate` 方法可填充菜单资源。对于上下文菜单的填充时机，当用户长时间按压屏幕（相当于桌面平台单击鼠标的右键菜单），框架调用 Activity 组件的创建上下文菜单的回调方法 `onCreateContextMenu` 来填充上下文菜单资源；选项菜单的填充时机一般是单击菜单按钮（）后，框架调用 Activity 组件的创建选项菜单的回调方法 `onCreateOptionsMenu` 来填充选项菜单资源。填充后的菜单实例（即 `inflate` 方法中 `menu` 参数）由框架去控制其显示方式。

菜单资源标识 `R.menu.ctx_menu` 描述的是名为 `ctx_menu` 的菜单资源定义文件，其存在于资源文件夹下的 `menu` 子文件夹中。代码 3-46 是该文件内容。

代码 3-46 菜单资源定义

文件名: `ctx_menu.xml`

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/android">
3      <item android:id="@+id/mi_clear" android:title="清除文本"
4          android:icon="@drawable/clear" android:showAsAction="ifRoom|withText"/>
5  </menu>
```

需要注意的是，要为某个控件绑定上下文菜单，需要提前“告诉”Activity 组件，使用 Activity 组件实例的 `registerForContextMenu` 方法可指定需要绑定上下文菜单的控件，如代码 3-47 所示。

代码 3-47 为组件注册上下文菜单

文件名: ActivityCollectionsAct.java

```
1  @Override
2  public void onCreate(Bundle savedInstanceState) {
3      .....
4      //注册上下文菜单
5      EditText editor = (EditText)findViewById(R.id.text);
6      this.registerForContextMenu(editor);
7      .....
8  }
```




3. 资源管理器 (Resources)

对于布局或菜单，因为用于界面显示且以 XML 进行定义，所以需要先填充再使用；而对于那些简单的（如字符串、颜色值等）或不以 XML 定义的资源（如图片），那又该如何使用呢？Activity 组件内建的资源管理器用于对组件所定义的资源进行访问。代码 3-48 是在 Activity 组件中使用资源管理器获取应用程序资源的实例代码。

代码 3-48 使用资源管理器获取资源

文件名: ActivityCollectionsAct.java

```

1 //获取字符串资源
2 String appName= this.getResources().getString(R.string.app_name);
3 //获取可绘制用资源（图片）
4 Drawable wallpaper = this.getResources().getDrawable(R.drawable.wallpaper);

```

使用 Activity 组件的 getResources 方法可获取与之相关的资源管理器，使用该接口的 getXXX 方法，其中“XXX”为资源类型标识，如 String（字符串）、Drawable（绘制用）等，可获得资源 ID 所对应的资源内容。

4. 资产管理器 (AssetManager)

相比资源管理器，资产管理器所面向的是原数据文件，如用户自定义的 XML 文件、文本文件或数据文件。因为数据内容不一定遵照 Android 平台的约定，所以资产管理器无法直接对这些内容进行解析或“填充”，而只能通过底层的方式来读取。代码 3-49 是通过资产管理器读取原文件的实例代码。

代码 3-49 使用资产管理器读取原文件

文件名: ActivityCollectionsAct.java

```

1 private void showAssetContent() { //显示资产内容
2     String lists[] = null;
3     //遍历 Activity 组件相关的所有资产（文件夹或文件）
4     try { lists = this.getAssets().list("");
5     } catch (IOException e1) { e1.printStackTrace(); }
6
7     for(int i = 0; i < lists.length; ++i) {
8         //该资产不为文件时略过
9         if(lists[i].indexOf(".") == -1) { continue; }
10        //使用 I/O 读取原文件内容
11        try {
12            InputStream is = this.getAssets().open(lists[i]);
13            final int bufferSize = is.available();
14            byte data[] = new byte[bufferSize];
15            is.read(data, 0, bufferSize);
16            is.close();
17            //输出文件内容
18            showInfo(new String(data));
19        } catch (IOException e1) { e1.printStackTrace(); }
20    }
21 }

```




```
19     }
20 }
```

使用 Activity 组件的 `getAssets` 方法可获取与之相关的资产管理器，使用该接口的 `open` 方法可获得原文件的输出流（第 11 行），从而进行内容的读取。

3.4.2 用户界面框架相关

在 Android 3/4 中，Activity 组件的用户界面框架/接口有操作栏、片段管理器、窗体和窗体管理器。表 3-7 是用户界面框架/接口的说明。

表 3-7 用户界面框架/接口的说明

框架/接口	说 明
操作栏 (ActionBar)	类似于桌面应用程序的工具栏，为用户提供操作入口
片段管理器 (FragmentManager)	用于对 Activity 组件所包含的片段进行管理
窗体 (Window)	代表应用程序的顶层窗口
窗体管理器 (WindowManager)	用于获取系统默认显示信息

1. 操作栏 (ActionBar)

操作栏是 Android 3/4 的新特性，每一个 Android 程序都内建有操作栏。顾名思义，操作栏为用户提供操作入口，如图 3-27 所示。



图 3-27 操作栏

在图 3-27 中，操作栏从左到右依次放置有向上导航按钮、应用程序图标、标题、选项菜单项和选项菜单导航按钮。代码 3-50 是在操作栏上设置向上导航按钮和标题的代码。

代码 3-50 设置操作栏

文件名: ActivityCollectionsAct.java

```
1 private void setActionBar() { //设置操作栏
2     this.getActionBar().setDisplayHomeAsUpEnabled(true);
3     this.getActionBar().setTitle(mResources.getString(R.string.app_name));
4 }
```

使用 Activity 组件的 `getActionBar` 方法可获取与之相关的操作栏对象。

2. 片段管理器 (FragmentManager)

片段 (Fragment) 是 Android 3/4 的新特性，其表示 Activity 组件的一部分动作（后台功能）或界面（前台展现）。一个 Activity 组件可包含多个片段，Activity 虽然包含片段组件，但其不直接对片段进行管理，而是通过片段管理器。

Activity 组件使用 `getFragmentManager` 方法获取片段管理器；使用片段管理器的 `findFragmentById` 方法可获取指定 ID 的片段组件实例。



囿于篇幅，对片段管理器在此不予过多介绍，请参考第4章。

3. 窗体 (Window)

窗体代表 Activity 组件的顶层窗口，窗体可用于设置顶层窗口的外观（如背景和颜色等）和行为政策（如默认的按键处理等），如代码 3-51 所示。

代码 3-51 设置窗体内容

文件名: ActivityCollectionsAct.java

```
1 private void showWindowInfo() { //显示窗体内容
2     //设置窗体背景
3     this.getWindow().setBackgroundDrawable(
4         this.mResources.getDrawable(R.drawable.wallpaper));
5     //设置窗体标题
6     this.getWindow().setTitle("设置背景");
7 }
```

使用 Activity 的 getWindow 方法可获取与之相关的窗体对象，使用该接口的 setBackgroundDrawable 方法可将绘制用资源（如图片）设置为窗体背景。

4. 窗体管理器 (WindowManager)

窗体管理器可用于获取默认的显示信息，如图 3-28 所示。

代码 3-52 是使用窗体管理器获取默认显示信息的主要代码。

刷新率:60.0
屏幕高:752
屏幕宽:1280
像素格式:4

图 3-28 窗口显示信息

代码 3-52 使用窗体管理器获取默认显示信息

文件名: ActivityCollectionsAct.java

```
1 private void showWindowManagerInfo() { //窗体管理器
2     StringBuffer sb = new StringBuffer();
3     final WindowManager mgr = this.getWindowManager();
4     final Display screen = mgr.getDefaultDisplay();
5     sb.append("刷新率:" + screen.getRefreshRate() + "\n");
6     sb.append("屏幕高:" + screen.getHeight() + "\n");
7     sb.append("屏幕宽:" + screen.getWidth() + "\n");
8     sb.append("像素格式: " + screen.getPixelFormat());
9     //显示信息
10    showInfo(sb.toString());
11 }
```

使用 Activity 组件的 getWindowManager 方法可获取与之相关的窗口管理器，使用该接口的 getDefaultDisplay 方法可获取系统默认的显示对象（第4行），使用默认显示对象的相关方法即可获取系统默认的显示信息。

3.4.3 内容提供相关

经过前面小节中对 Android 平台组件的介绍，相信读者对 Activity 组件的内容提供对象已经有所了解，其中内容解析器主要用于存取内容提供者所提供的的数据；而意向和未决意向

用于组件之间的内容传递。表 3-8 是对内容提供对象的说明。

表 3-8 内容提供对象的说明

内 容	说 明
内容解析器 (ContentResolver)	为存取内容提供者所提供的的数据提供接口
意向 (Intent)	用于组件之间的内容传递
未决意向 (PendingIntent)	用于约定组件之间的调用关系

1. 内容解析器 (ContentResolver)
- 有关内容解析器已在前面小节中进行介绍，不再赘述。
2. 意向 (Intent)
- 有关意向已在前面小节中进行介绍，不再赘述。
3. 未决意向 (PendingIntent)
- 有关未决意向已在前面小节中进行介绍，不再赘述。

3.4.4 管理框架相关

与 Activity 组件相关的管理框架有系统服务、片段管理器和加载管理器。表 3-9 是对这些管理器的说明。

表 3-9 管理框架的说明

对 象	说 明
系统服务	用于调用系统服务
片段管理器 (FragmentManager)	用于对 Activity 组件所包含的片段进行管理
加载管理器 (LoaderManager)	用于对数据加载器的管理

1. 系统服务

在 Android 平台中，系统内核以服务的形式向应用程序提供服务访问接口；使用 Activity 组件的 getSystemService 方法可按照服务名获取对应的系统服务接口，然后通过服务接口访问系统功能。

Android 平台提供了丰富的系统服务，在后续章中将陆续对系统服务进行介绍。

2. 加载管理器 (LoaderManager)

对于 ANR 错误，该错误造成的主因是在主线程中“嵌入”了执行时间较长的操作（如从互联网下载文件或加载数据等），造成主线程阻塞，从而让系统“认为”该程序运行异常。为了解决因主线程阻塞造成的问题，Android 3/4 新增了异步加载数据的新特性，加载管理器就是其中最主要的机制之一。

对于 Activity 或片段组件，使用 getLoaderManager 方法可获取与之关联的加载管理器，再通过加载管理器对加载器进行初始化和 管理。

(1) 初始化加载器

使用加载管理器的 initLoader 方法可对加载器进行初始化。该初始化方法有 3 个参数。表 3-10 是对其中参数的说明。



表 3-10 初始化加载器中参数的说明

参 数	类 型	说 明
id	int	加载器 ID
args	Bundle	参数
callback	LoaderCallbacks<D>	加载器回调接口

读者可将加载管理器的初始化管理为“预约登记”过程，因为加载器并不由加载管理器直接创建，而是在加载器创建回调方法（onCreateLoader）中进行，而该回调方法使用加载器 ID 来区分所要创建的加载器。

对于数据加载，由于多数是从数据库加载，所以所创建的加载器一般都是游标加载器，该加载器负责从指定的内容提供者中“加载”数据，该加载过程是异步的，不会对主线程造成阻塞。

(2) 加载完毕处理

当加载过程结束，加载器会在其加载完毕回调方法（onLoadFinished）中“通知”Activity 组件，并将数据访问接口（一般是记录游标）提供给 Activity。加载完毕回调方法包含两个参数：第一个参数是加载器，用于区分是哪个加载器；第二个参数是记录游标接口，用于访问结果集。一旦获取记录游标，即可遍历记录内容。

有关加载器使用请参考第 7 章。

3.4.5 环境信息相关

对于任何应用程序，环境信息都是非常有用的。对于桌面应用程序，往往需要获取程序当前文件夹、传入参数、系统文件夹等环境信息；对于 Android 应用程序，环境信息更为繁多，如外部存储器的路径、应用程序信息、各种私有文件夹等。为了便于区分，作者将与 Activity 关联的环境信息分为上下文和应用程序信息。

提示：Activity 类继承于上下文类（Context），所以其本身关联了很多与运行时环境相关的信息；应用程序信息是程序包所包含的信息，其内容在应用程序清单中定义。Android 平台的系统环境信息不在此列，这些信息与 Activity 组件无关，可使用 Environment 类（android.os 包中）进行获取。

1. 上下文信息

表 3-11 中列举的是与 Activity 组件关联的上下文信息的获取方法。

表 3-11 与 Activity 组件关联的上下文信息的获取方法

对 象	说 明
getParent()	获取父组件（强调包含关系）
getCallingActivity()	获取调用方 Activity 组件（强调调用关系）
getComponentName()	获取 Activity 组件名称
getPackageName()	获取包名
getPackageCodePath()	获取包代码路径
getPackageResourcePath()	获取包资源路径
getFilesDir()	获取私有文件夹路径
getExternalFilesDir(String type)	获取外部文件夹路径

2. 应用程序信息

应用程序信息包括应用程序标签、包名（进程名）、最低 SDK 版本、许可、本地库文件夹、共享库文件列表等，如图 3-29 所示。

标签:Activity集锦示例
包名:foolstudio.test.core
资源路径:/data/app/foolstudio.test.core-1.apk
进程名:foolstudio.test.core
本地库路径:/data/data/foolstudio.test.core/lib
最低SDK版本:12
许可:null
文件资源文件夹:/data/data/foolstudio.test.core/files

图 3-29 应用程序信息

代码 3-53 是获取图 3-29 中信息的主要代码。

代码 3-53 获取应用程序信息

文件名: ActivityCollectionsAct.java

```
1 private void showApplicationInfo() { //显示应用程序信息
2     StringBuffer sb = new StringBuffer();
3     final ApplicationInfo info = this.getApplication().getApplicationInfo();
4     sb.append("标签:"+getResources().getString(info.labelRes) + "\n");
5     sb.append("包名:"+this.getApplication().getPackageName() + "\n");
6     sb.append("资源路径:"+this.getApplication().getPackageResourcePath() + "\n");
7     sb.append("进程名:"+ info.processName + "\n");
8     sb.append("本地库路径:"+ info.nativeLibraryDir + "\n");
9     sb.append("最低 SDK 版本:"+ info.targetSdkVersion + "\n");
10    sb.append("许可:"+ info.permission + "\n");
11    sb.append("文件资源文件夹:"+this.getFilesDir().getAbsolutePath());
12    showInfo(sb.toString());
13 }
```

使用 Activity 组件的 `getApplication` 方法可获取与之相关的应用程序接口，使用该接口的 `getApplicationInfo` 方法又可获取应用程序信息接口（第 3 行）。

3.4.6 数据存储相关

Activity 组件提供了 3 种方式进行数据存储，包括数据库、私有文件和共享首选项。表 3-12 是对这 3 种方式的说明。

表 3-12 Activity 组件数据存储方式的说明

内 容	说 明
数据库	使用 SQLite 数据进行数据存储
私有文件	只有创建的应用程序才能访问该文件
共享首选项 (SharedPreferences)	以首选项的方式（“键—值”）对数据进行存储



1. 数据库

图 3-30 所示的界面是使用数据库存储登录和登出的日志。

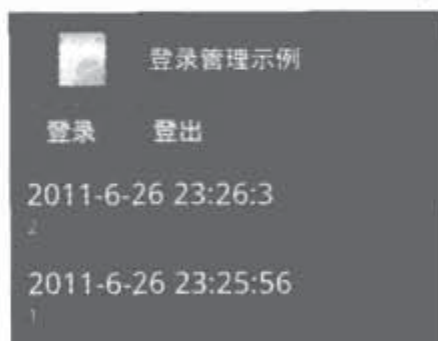


图 3-30 数据库存储

代码 3-54 是图 3-30 所示的程序使用数据库的主要代码。

代码 3-54 使用数据库存储数据

文件名: LoginMgrAct.java

```

1  private void init() {
2      //初始化数据库文件
3      File f = this.getDatabasePath(IDbSpec.DB_NAME);
4      boolean exists = f.exists();
5      this.mDb = this.openOrCreateDatabase(IDbSpec.DB_NAME, 0, null);
6      if(!exists) { //初始化数据表
7          mDb.execSQL(getResources().getString(R.string.sql_init_tbl));
8      }
9      //创建适配器
10     Cursor c = mDb.query(IDbSpec.TBL_NAME, null, null, null, null, null, null);
11     SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
12         android.R.layout.simple_list_item_2, c,
13         new String[] { IDbSpec.COL_TSP, IDbSpec.COL_TYPE },
14         new int[] { android.R.id.text1, android.R.id.text2 });
15     //设置适配器
16     this.setListAdapter(adapter);
17 }
18 .....
19 private void doLogin() { //添加登录日志
20     ContentValues values = new ContentValues(2);
21     values.put(IDbSpec.COL_TYPE, "1");
22     values.put(IDbSpec.COL_TSP, FooSysUtil.getInstance().getTsp());
23     //添加记录
24     mDb.insert(IDbSpec.TBL_NAME, null, values);
25     notifyDataSetChanged();
26 }

```

在代码 3-54 中, 使用 Activity 组件的 openOrCreateDatabase 方法可获取与之相关的数据库接口 (第 5 行), 使用该接口的 execSQL、query 等方法来执行数据库操作。

2. 私有文件

Activity 组件对私有文件的使用方式并没有什么特别之处，与读者熟知的文件读/写处理一样。图 3-31 是读取私有文件的内容。



Copyright(C) 2011 foolstudio.
All rights reserved.
邮箱: foolstudio@yahoo.com.cn
微博: <http://t.qq.com/foolstudio>

图 3-31 私有文件读取内容

(1) 私有数据写入

代码 3-55 是向私有文件中写入数据的主要代码。

代码 3-55 向私有文件中写入数据

文件名: ActivityCollectionsAct.java

```
1 private void init() {  
2     //打开私有文件用于输出  
3     try { FileOutputStream fos = this.openFileOutput(IConfig.PRIVATE_FILE, 0);  
4         DataOutputStream dos = new DataOutputStream(fos);  
5         //以 UTF-8 编码的方式写入  
6         dos.writeUTF(this.mResources.getString(R.string.about) );  
7         //关闭文件流  
8         dos.close();  
9         fos.close();  
10    } catch (IOException e) { e.printStackTrace(); }  
11 }
```

在代码 3-55 中，使用 Activity 组件的 `openFileOutput` 方法可获取与之相关的私有文件的数据写入接口（第 2 行），通过该接口往文件写入数据（第 6 行）。

(2) 私有数据读取

代码 3-56 是从私有文件中读取数据的主要代码。

代码 3-56 从私有文件中读取数据

文件名: ActivityCollectionsAct.java

```
1 private void doFileStream() { //打开私有文件用于输入  
2     try { FileInputStream fis = this.openFileInput(IConfig.PRIVATE_FILE);  
3         DataInputStream dis = new DataInputStream(fis);  
4         //按 UTF 的编码进行读取  
5         String data = dis.readUTF();  
6         //关闭文件流  
7         dis.close();  
8         fis.close();  
9         //显示私有文件内容  
10    showInfo(data);  
    }
```




```

11         } catch (IOException e) { e.printStackTrace(); }
12     }

```

在代码 3-56 中，使用 Activity 组件的 `openFileInput` 方法可获取与之相关的私有文件的数据读取接口（第 2 行），使用该接口从文件中读取数据（第 5 行）。

3. 共享首选项（SharedPreferences）

共享首选项使用方式和私有文件相同，其差异在于数据的组织方式。私有文件的数据内容可是任意方式；而共享首选项的数据项格式为“键—值”映射形式（见图 3-32），类似于 J2SE 平台的 `properties` 文件，或桌面平台的 `ini` 文件。

app_name=Activity 集锦示例

图 3-32 共享首选项读取内容

（1）设置首选项

代码 3-57 是设置共享首选项的主要代码。

代码 3-57 设置共享首选项

文件名：ActivityCollectionsAct.java

```

1  SharedPreferences mSharedPrefs = this.getSharedPreferences(IConfig.SHARED_PREFS,
2
3      Activity.MODE_WORLD_WRITEABLE|Activity.MODE_WORLD_READABLE);
4  .....
5  private void init() {
6      //设置共享首选项（往文件中写入“键—值”，并提交）
7      SharedPreferences.Editor editor = mSharedPrefs.edit();
8      editor.putString(IConfig.KEY_NAME, this.mResources.getString(R.string.app_name));
9      editor.commit();
10 }

```

在代码 3-57 中，使用 Activity 组件的 `getSharedPreferences` 方法可获取与之相关的共享首选项接口（第 1 行），使用该接口的 `edit` 方法可获取首选项的编辑接口（第 6 行），使用编辑接口的设置方法（形如“putXXXX”，如 `putString`）来设置指定键的值（第 7 行）。

在设置完毕后，必须使用编辑接口的提交方法 `commit` 对修改进行提交（第 8 行）。

（2）获取首选项

代码 3-58 是获取共享首选项的主要代码。

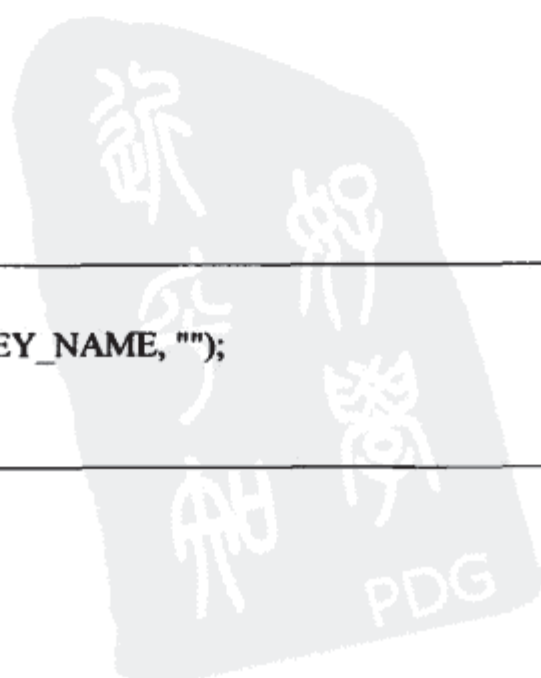
代码 3-58 获取共享首选项

文件名：ActivityCollectionsAct.java

```

1  private void doSharedPrefs() { //读取首选项
2      String about = mSharedPrefs.getString(IConfig.KEY_NAME, "");
3      this.showInfo(IConfig.KEY_NAME+"="+about);
4  }

```



在代码 3-58 中，直接使用共享首选项接口的获取方法（形如“getXXXX”，如 getString）来获取指定键的值（第 2 行）。

3.5 Android 应用程序组件小结

Android 平台中定义了 4 种重要的应用程序组件：Activity、服务、广播接收器和内容提供者。这些应用程序组件根据其适用场合的不同被作者赋予不同的角色：Activity 组件主要应用于提供用户界面，是非常注重界面表现的“形象大使”；而服务组件主要用于后台业务处理，是不习惯表现的“老黄牛”；广播接收器主要用于接收系统或用户程序所发送的广播消息并做出响应，是一个积极的“倾听者”；内容提供者组件主要用于使用约定的方式将本组件的数据共享给外部组件，是当之无愧的“奉献者”。

使用这 4 种基本组件的组合和集成，开发人员可开发出满足各种应用的程序。Android 平台还为 4 种组件之间的过程调用、数据共享提供了一些辅助的机制。使用意向组件桥接机制，可实现 Activity 组件与 Activity 组件以及 Activity 组件与服务组件之间的数据交互；使用线程消息队列的机制，Activity 组件可与外部线程进行消息传递。

AIDL IPC 和 RPC 机制是 Android 平台的核心机制，用于提供对远程对象的访问。Activity 组件可在本地调用远程服务组件所“暴露”的接口，该方法在远程对象中进行执行，使用 RPC 机制进行过程参数和执行结果的传递。



第4章 用户界面设计

本章首先对 Android 平台界面组件结构 and 应用模式进行系统阐述，再结合实际开发案例对布局、视图等常用界面组件的使用进行详细引导。在掌握组件使用的基础上，还对界面框架进行全面介绍。

4.1 Android 平台界面组件结构

4.1.1 Android 界面组件结构层次

在 Android 平台中，所有的可视组件都是视图（View）的子类；视图类有一个重要的直接子类是视图组（ViewGroup）；视图组类有一组重要的直接或间接子类是布局，如图 4-1 所示。Android 平台把这些布局类都归集到小部件（Widget）包中。

视图组类有一个重要的直接子类是适配器视图（AdapterView）。适配器视图又包含若干直接或间接的视图子类，如列表视图（ListView）和网格视图（GridView）。

视图类还有一个重要的直接子类是文本视图（TextView）。按钮组件（Button）是文本视图类直接子类之一；而复选框（CheckBox）和单选按钮（RadioButton）又是按钮的间接子类。



图 4-1 Android 可视组件架构示意图

4.1.2 理解 Android 界面组件结构层次

图 4-1 中的类在继承和归集上存在一定的交错。例如，视图组类是视图的子类，而视图组类又是某些视图的父类；布局类被归集到小部件包中。读者可能更关注的是，这些界面组件结构层次的设计思路是什么，如何方便地去理解。下面将借助 J2SE 平台的界面设计规则来介绍 Android 平台的界面组件的结构层次设计。

在 J2SE 平台中，组件类（Component）是所有抽象窗体工具（Abstract Window Toolkit, AWT）和 Swing 组件的父类。组件类的直接子类中既有控件（如按钮），也有容器

(如面板)。对于容器是组件的子类,可以将容器理解为组合组件。而容器类的子类除了一些高级容器类(如 Window、Panel 等),还包括 Swing 库中几乎所有组件的父类——组件类(JComponent)。也就是说,JComponent 的子类都具有容器的特性, JPanel 可以包含按钮。

J2SE 平台规定:每一个使用 Swing 组件的程序至少包含一个高级容器,JComponent 子类的最顶层组件必须放入到高级容器或其子类中。也就是说,可以把 JButton 添加到 JLabel 中,再把该 JLabel 添加到 JPanel 中……,但是 JButton 的最顶层组件必须添加到 JFrame 或 JDialog 这些高级容器中。这些组件的结构层次如图 4-2 所示。

需要提醒的是,JFrame 和 JDialog 都不是 JComponent 的子类,JPanel、JLabel 和 JButton 才是 JComponent 的子类。在图 4-2 中,JButton 的最顶层组件是 JPanel。

至此,通过参考 J2SE 平台的界面组件的结构层次,读者可以看出,容器可以是组件类的子类;而组件也可以是容器类的子类。因为容器本身也是组件,可以视为组合组件;组件本身也是容器,可以由多个子组件组成。界面组件的结构层次可以如图 4-3 所示。

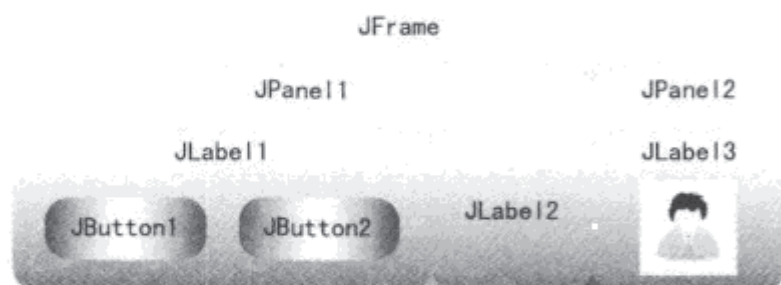


图 4-2 J2SE 平台界面组件的结构层次示意图

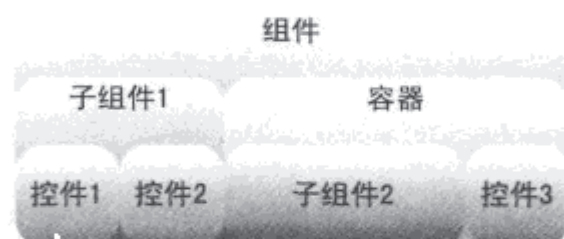


图 4-3 J2SE 平台的界面组件的结构层次示意图

切换到 Android 平台,这样的理解同样可以“移植”到视图与视图组的关系上:视图组作为容器可以包含视图,视图组本身也是视图,用于显示;而视图也可以作为容器,可以包含视图组(布局)。

4.1.3 布局的地位

在 J2SE 平台中,布局管理接口(LayoutManager)是所有布局的父类,从继承关系上,无论是组件还是容器都与布局没有直接关系。容器类通过 setLayout 方法来显式地设置其包含的所有组件的布局方式。如果不显式指定布局,各个容器将采用其默认的布局,如 JPanel 的默认布局为流布局(FlowLayout),内容面板的默认布局是边界布局(BorderLayout)。如果显式将容器的布局设置为空(null),那么该容器会按各子组件的绝对位置摆放子组件,如此一来,必须指定父窗体和各个组件的大小,并为组件设置不同的坐标。

由此看来,在 J2SE 平台中,布局与组件的关系是单方依存,布局依赖于组件,组件不依赖于布局。而对 Android 平台,布局被定义为视图组的直接或间接子类,并纳入到小部件包中。在功能上,布局既可以用于包含其他视图,同时又作为视图显示,也可以作为组件加入到其他的布局中。图 4-4 是一个简单的组合布局的实机界面。



图 4-4 组合布局的实机界面

在图 4-4 所示的界面中,根布局是垂直方向的线性布局,该布局包含一个表格布局和一个文本编辑框,其中表格布局又包含 3 个按钮控件。该界面的内容结构层次如图 4-5 所示。



至此，读者应该对 Android 平台中布局的“地位”有了清楚的认识：Android 平台中的布局就是可视组件，既可以作为容器来容纳其他可视组件，也可以作为组件加入到其他布局中。这样，通过布局包含视图，布局包含布局，从而形成繁茂的视图结构层次树，给用户展示的就是更加灵活和丰富的界面效果，如图 4-6 所示。

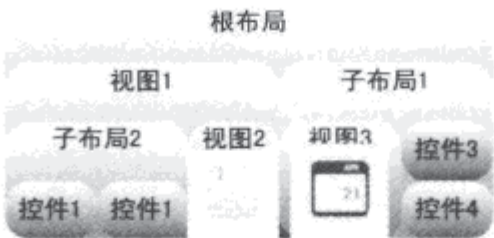
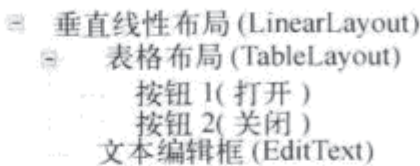


图 4-5 垂直线性布局的界面组件结构层次示意图

图 4-6 Android 平台界面元素结构层次示意图

4.2 界面组件使用模式

4.2.1 定义界面组件

在 Android 应用程序中，界面的定义最终将在 Activity 组件中进行呈现。在这个呈现过程之前，开发者必须定义界面组件，包括其属性和布局方式等。Android 组件大多数可以通过以下两种方式进行定义。

1. 使用 XML 代码定义界面组件

Android 平台为大多数视图及其子类提供了 XML 定义标记，也就是说通过 XML 标记即可定义该对应的视图或其子类。代码 4-1 是一个界面定义实例。

代码 4-1 界面组件定义

文件名: main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent" android:layout_height="fill_parent">
5     <Button android:id="@+id/btn1" android:text="确定"
6         android:layout_width="wrap_content" android:layout_height="wrap_content"
7         android:layout_gravity="center_horizontal"/>
8 </LinearLayout>
```

代码 4-1 包含一个按钮标记（第 5 行），其对应一个按钮控件。对于开发者而言，XML 代码与 Java 代码具有同等效果。

2. 使用 Java 代码定义组件对象

使用代码定义对象是开发中的常见方式，即在代码中通过 new 语句创建类实例，然后设置其属性，调用其方法。但这种方式几乎很少用于 Android 平台的组件定义。

实际上，对于使用 XML 定义组件的方式，同样需要通过 XML 标记内容来构建对应的类实例，只是该实例化过程被 Android 框架自动地实现了。当应用程序启动后，运行时库会

解析界面的 XML 定义，并根据 XML 标记和属性来实例化该标记对应的类实例，最后通过应用程序框架显示该组件。

4.2.2 生成界面组件资源标识

对象的引用，几乎都是遵循“先创建、再使用”的规则，即先实例化类定义，再使用对象实例来执行其方法。在 Android 平台中，对于 XML 定义的界面组件，其实例化过程由 Android 平台自动完成，这样就会遇到问题：如何引用这些实例化后的对象实例呢？

实际上，在 XML 资源的实例化过程中，资源打包工具（aapt）解析 XML 定义，并获取其中组件的定义标识，生成对应的资源标识，如此一来，就建立了组件定义标识与组件实例的对应关系，即可通过组件的定义标识获取对应的对象实例。

ADT 工具会根据界面定义生成相应的资源标识，并通过约定类（“R”）进行归集，代码 4-2 就是根据代码 4-1 中定义界面所生成的资源标识信息。

代码 4-2 资源定义类

文件名：R.java

```
1 public final class R {  
2     public static final class attr {}  
3     public static final class drawable { public static final int icon=0x7f020000; }  
4     public static final class id { public static final int btn1=0x7f050000; }  
5     public static final class layout { public static final int main=0x7f030000; }  
6     public static final class string { public static final int app_name=0x7f040000; }  
7 };
```

在代码 4-2 中，整个界面（main.xml）使用名为 main 的标识进行定义（第 5 行），那么使用 R.layout.main 标识即可访问整个界面组件（布局组件）；界面中标识的 btn1 的组件使用名为 btn1 的标识进行定义（第 4 行），那么使用 R.id.btn1 标识即可访问该按钮组件。

4.2.3 组件属性和标识

在 Android 平台中，几乎每一个视图类及其子类都有多个 XML 属性。这些 XML 属性就是用来定义该类的实例对象的成员内容。例如，文本组件的 android:text 属性用于描述该组件的显示文本。在 XML 中，设置该属性与 Java 代码中使用该组件对象实例的 setText 方法的效果相同。一般的，对于组件的 XML 属性，都有对应的运行时方法用于设置。

注意：在组件的 XML 定义中，给一个文本组件设置 android:text 或 text 属性都是合法的，但是 text 属性是不会被 Android 平台的资源解析功能所识别的，所以无论给 text 属性设置什么内容，都不会被显示出来；反之，android:text 是 Android 平台约定的属性名，能被资源解析功能所识别。

组件 ID 是一个特殊的属性，用于区分构成界面的各个组件。ID 的属性名为 android:id，其属性值的语法为



`android:id="@+id/<组件 ID>"`

其中，“@”、“+”和“/”符号是 Android 特有的资源定义符号。“/”为分隔符；“@”字符表示“/”符之后的内容是组件的 ID 字符串；“+”字符表示该资源必须创建并添加到资源中。代码 4-1 中第 5 行定义了一个 ID 为 `btn1` 的按钮组件。

在运行时 Java 代码中，该组件 ID 的格式为“`R.id.<组件 ID>`”；在 XML 代码中引用的格式为“`@id/<组件 ID>`”，没有“+”字符表示无需创建，仅仅是引用。

提示：ID 属性的创建并不一定都在对应组件定义语句内。在 Android 中存在这种情形：某些视图组的组件需要参考其所包含的子组件，但这时子组件还没有开始定义，无法进行引用。为了解决这种问题，Android 平台运行在引用组件 ID 的同时创建 ID，该组件的定义语句可以放在被引用的标记之后，在该组件的定义语句块中，无需再指示创建 ID，设置为引用时的 ID 即可，如代码 4-3 所示。

代码 4-3 引用组件 ID 并创建 ID

文件名：main.xml

```
1 <SlidingDrawer android:layout_width="fill_parent" android:layout_height="fill_parent"
2     android:handle="@+id/HANDLE_VIEW" android:content="@+id/CONTENT_VIEW">
3     <TextView android:id="@id/HANDLE_VIEW" android:gravity="center_horizontal"
4         android:layout_width="fill_parent" android:layout_height="wrap_content"
5         android:background="#111111" android:text="请选择项目"/>
6     <ListView android:id="@id/CONTENT_VIEW"
7         android:layout_width="fill_parent" android:layout_height="wrap_content"/>
8 </SlidingDrawer>
```

在代码 4-3 中，父组件在其属性中引用第 3 行和第 6 行才定义的子组件，但是由于子组件的定义在引用之后，所以采用引用 ID 并创建 ID 的方式。在其子组件定义语句中无需再创建 ID（第 3 行和第 6 行的 ID 定义中不再有“+”符号）。

4.2.4 引用界面组件

界面组件的引用也分为两种情形：在 XML 代码中引用组件（如代码 4-3 所示）和在 Java 代码中引用组件。

通过组件资源标识，使用 Activity 组件或者父组件的 `findViewById` 方法即可获取指定组件的对象实例。代码 4-4 是引用代码 4-1 中所定义的界面组件的示例代码。

代码 4-4 引用界面组件

```
1 @Override
2 public void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.main);
5     //获取按钮控件并设置单击事件侦听器
6     Button btn = (Button)findViewById(R.id.btn1);
```



```

7      btn.setOnClickListener(this);
8  }

```

在代码 4-4 中，使用 `R.layout.main` 来引用整个界面定义（第 4 行）；使用 `R.id.btn1` 来引用该界面中的按钮组件，并获取按钮对象实例（第 6 行）。

4.2.5 界面设计器

为了提高界面设计效率，ADT 也毫不例外地提供了界面设计器工具，而且不断进行完善。通过界面设计器，开发者可用拖曳的方式向界面中添加组件；以选择的方式设置整个界面属性、组件的属性。总而言之，可以非常简单而直观地进行界面设计，如图 4-7 所示。

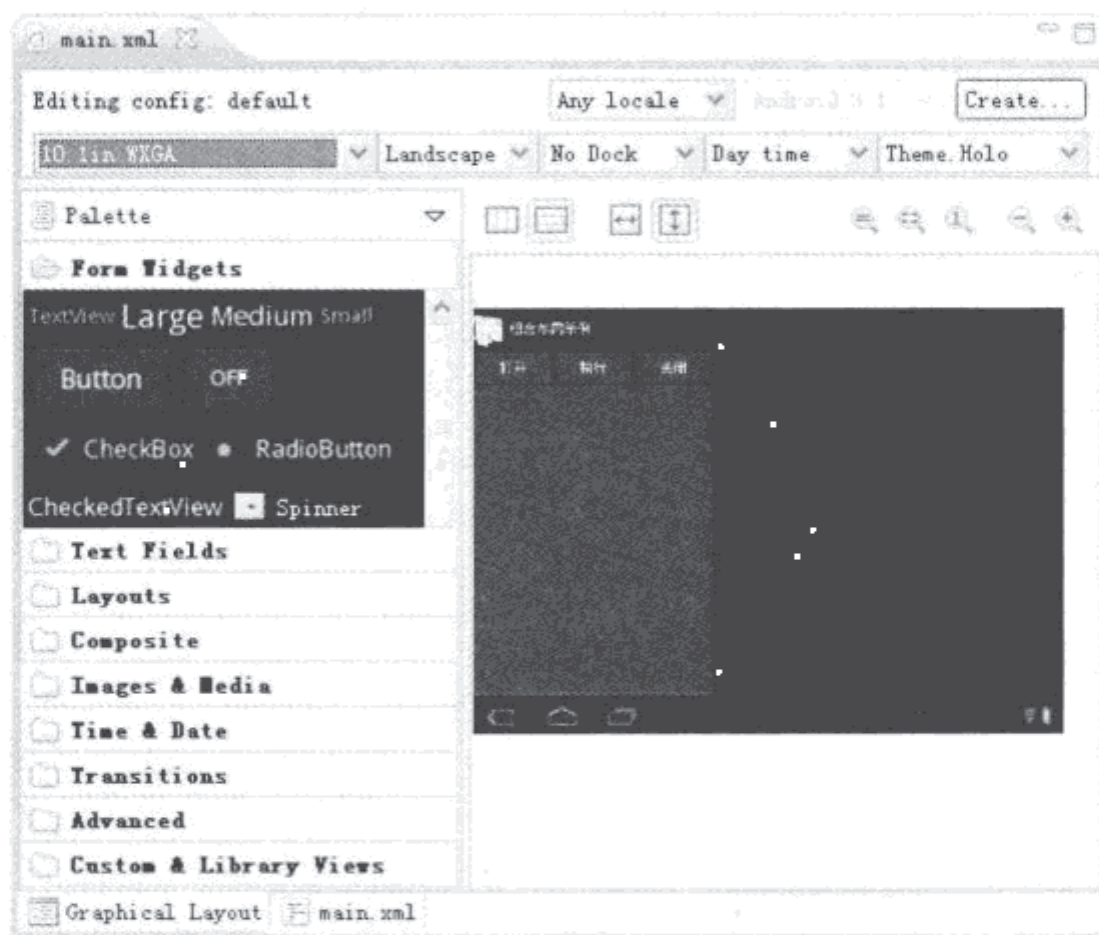


图 4-7 界面设计器

实际上，无论是否使用界面设计器，其输出形式还是 XML 代码，而这种通过 XML 定义界面的机制，不仅增强了定义的弹性，同时也减少了 Java 编码量。

4.3 布局组件 (Layouts)

相信在 J2SE 平台下进行界面设计的开发者对于布局 (Layout) 这个概念应该是耳熟能详，好的界面都必须在布局设计上大做文章。从边界布局 (Border Layout)、卡片布局 (Card Layout)、格子布局 (Grid Layout)……J2SE 平台提供的布局类型不下 20 种！

在 J2SE 平台中，布局与可视组件还是一种依存关系，布局只是描述了可视组件的摆放方式，也就是说，即使不要布局，可视组件也是可以摆放的（只是摆放得比较杂乱）。

特别的，Android 平台强化了布局的“地位”，可视组件必须放在布局或者与布局具有相

同“地位”的容器中，否则资源管理器认为该“摆放方式”非法！

Android 平台定义了线性布局、相对布局、框布局、表格布局和绝对布局等 5 种布局类型。虽然从数量上看，貌似比 J2SE 平台所提供的要少得多，但实际上，通过多种布局的组合，可以满足各种布局要求。图 4-8 是布局组件继承关系层次结构图。

图 4-8 中的这些布局类几乎都在小部件包（`android.widget`）中定义。

4.3.1 线性布局（Linear Layout）

线性布局就是将子组件按照直线进行摆放的一种格局，该布局与 J2SE 平台的流布局类似。图 4-9 是一个线性布局的实机界面，其中的组件按照垂直方向进行摆放。

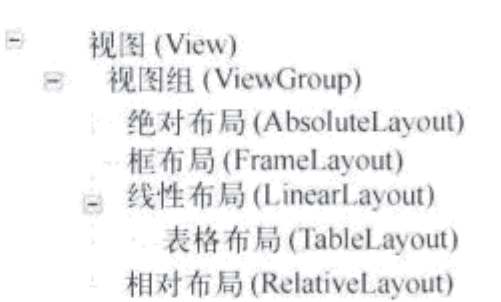


图 4-8 布局组件的层次结构示意图



图 4-9 线性布局的实机界面

无论是 Android 平台的线性布局，还是 J2SE 平台的流布局，它们都是有方向的。在流布局中定义的方向是“从左到右”（`LEFT_TO_RIGHT`）和“从右到左”（`RIGHT_TO_LEFT`）；在线性布局中定义的方式是“水平的”（`Horizontal`）和“垂直的”（`Vertical`）。

与流布局不同的是，线性布局是一个显示组件容器，有宽度和高度属性。

1. 定义线性布局

代码 4-5 是图 4-9 中界面布局的 XML 定义，其根节点是一个线性布局标记。

代码 4-5 线性布局定义

文件名：linear_layout_view.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <Button android:text="按钮"
6          android:layout_width="wrap_content" android:layout_height="wrap_content"/>
7      <CheckBox android:text="复选框"
8          android:layout_width="wrap_content" android:layout_height="wrap_content"/>
9      <RadioButton android:text="单选按钮"
10         android:layout_width="wrap_content" android:layout_height="wrap_content"/>
11  </LinearLayout>
```

在代码 4-5 中，标记 `LinearLayout` 表示定义了一个线性布局（第 2 行），该 XML 代码经过布局填充器填充后会创建一个线性布局类实例。该布局包含 3 个子组件：一个按钮组件（第 5 行）、一个复选框（第 7 行）和一个单选按钮（第 9 行），各组件又有各自的属性。

2. 线性布局的 XML 属性

在代码 4-5 中，线性布局包含若干个 XML 属性，介绍如下。

(1) 命名空间

其属性名为 `xmlns:android`，其值由 Android 平台定义，固定为

`http://schemas.android.com/apk/res/android`

(2) 方向属性

其属性名为 `android:orientation`，用于指定其中子组件的摆放方式（水平还是垂直）。表 4-1 是其取值类型及含义。

表 4-1 线性布局的方向属性值及含义

取值类型	含 义
<code>vertical</code>	垂直方向
<code>horizontal</code>	水平方向

(3) 宽度和高度属性

宽度属性名为 `android:layout_width`；高度属性名为 `android:layout_height`，用于指明自身的显示大小。表 4-2 是其取值类型及含义。

表 4-2 宽度和高度属性值及含义

取值类型	含 义
<code>fill_parent</code>	充满父组件
<code>wrap_content</code>	根据内容调整
<code>match_parent</code>	匹配父组件
具体大小值	指定值

有关大小值的定义参考第 16 章。

3. 引用线性布局

代码 4-6 是引用代码 4-5 所定义的线性布局的实例代码。

代码 4-6 引用线性布局

文件名: `LayoutsAct.java`

1	<code>public class LayoutsAct extends Activity{</code>
2	<code> @Override</code>
3	<code> public void onCreate(Bundle savedInstanceState) {</code>
4	<code> super.onCreate(savedInstanceState);</code>
5	<code> setContentView(R.layout.linear_layout_view);</code>
6	<code> }</code>
7	<code>};</code>

在代码 4-6 中，使用 Activity 组件的 `setContentView` 方法将线性布局设置为其内容视图即可。



4.3.2 相对布局（Relative Layout）

如果说线性布局是“直来直去”，那么相对布局就是“看菜吃饭”了。对于相对布局的子组件，后一个组件要参照前一个组件的位置进行摆放。图 4-10 是一个相对布局的实机界面。

1. 定义相对布局

代码 4-7 是图 4-10 中界面布局的 XML 定义，其根节点是一个相对布局标记。



图 4-10 相对布局的实机界面

代码 4-7 相对布局定义

文件名: relative_layout_view.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <Button android:id="@+id/btn1" android:text="按钮"
6          android:layout_width="wrap_content" android:layout_height="wrap_content"/>
7      <CheckBox android:id="@+id/chx1" android:text="复选框"
8          android:layout_width="wrap_content" android:layout_height="wrap_content"
9          android:layout_alignTop="@id/btn1" android:layout_toRightOf="@id/btn1"/>
10     <RadioButton android:id="@+id/rbtn1" android:text="单选按钮"
11         android:layout_width="wrap_content" android:layout_height="wrap_content"
12         android:layout_below="@id/chx1" android:layout_alignLeft="@id/chx1"/>
13 </RelativeLayout>
```

在代码 4-7 中，标记 RelativeLayout 表示定义了一个相对布局（第 2 行），该 XML 代码经过布局填充器填充后会创建一个相对布局类实例。该布局中包含 3 个子组件：按钮（第 5 行）、复选框（第 7 行）和单选按钮（第 10 行）。

2. 相对布局中子组件的 XML 属性

表 4-3 是相对布局中子组件用于描述其位置的属性及说明。

表 4-3 相对位置描述属性及说明

属 性	说 明
android:layout_below	在所参照组件的下方
android:layout_above	在所参照组件的上方
android:layout_toLeftOf	在所参照组件的左方
android:layout_toRightOf	在所参照组件的右方
android:layout_alignTop	与参照组件上对齐
android:layout_alignBottom	与参照组件下对齐
android:layout_alignLeft	与参照组件左对齐
android:layout_alignRight	与参照组件右对齐

根据属性说明可知，在代码 4-7 中，复选框在按钮的右边，而且顶部和按钮对齐；单选按钮在复选框的下方，且与复选框左边对齐。

相对布局在布局的灵活性方面比线性布局更强大，但是其定义也较为复杂，布局中的每个组件几乎都需要定义一个 ID，子组件之间通过 ID 进行参照。

3. 引用相对布局

相对布局的引用方式和线性布局相同，不再赘述。

4.3.3 框布局 (Frame Layout)

框布局就像一个电影屏幕，该屏幕用来显示电影数据中的每一幅画面，但同一时刻，屏幕上只能显示一幅画面。框布局中包含多个画面（组件），这些画面相互叠加，但只有一幅画面显示在屏幕的最前端。图 4-11 所示为一个框布局的实机界面，图片组件和按钮同时显示在该屏幕中，按钮主键覆盖了图片主键的一部分（如同视频的叠加效果）。

框布局特别适合切换显示的场合（如选项页组件，所有选项页都在屏幕的同一位置进行切换显示），正是由于框布局同时只能允许一个组件在屏幕的最上层进行显示，这样就会给那些还没有显示的组件一些“缓冲”的时间，这种特性与读者经常听到的屏幕缓冲机制同出一辙。

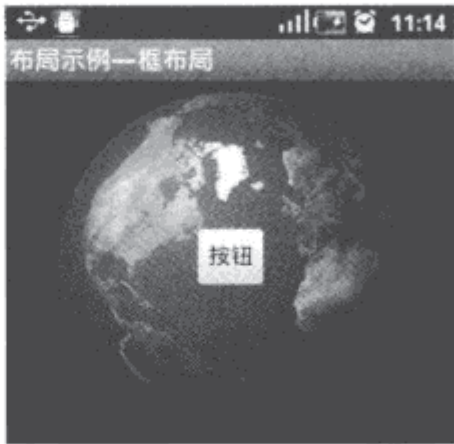


图 4-11 框布局的实机界面

提示：框布局来自对 FrameLayout 的翻译，有些书中翻译为框架布局。但是作者认为翻译成“框布局”更为明确一些，更能让读者联想到 FrameLayout 的使用方式：在一个方框的范围中，各种画面（组件）交替地“抛头露面”。

1. 定义框布局

代码 4-8 是图 4-11 中界面布局的 XML 定义，其根节点是一个框布局标记。

代码 4-8 框布局定义

文件名: layout/frame_layout_view.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <ImageView android:src="@drawable/bkg" android:scaleType="fitXY"
6          android:layout_height="wrap_content" android:layout_width="wrap_content"
7          android:layout_gravity="center"/>
8      <Button android:text="按钮"
9          android:layout_width="wrap_content" android:layout_height="wrap_content"
10         android:layout_gravity="center"/>
11  </FrameLayout>
```




在代码 4-8 中，标记<FrameLayout>表示定义了一个框布局（第 2 行），该 XML 代码经过布局填充器填充后会创建一个框布局类实例。该布局中包含两个子组件：图片视图（第 5 行）和按钮（第 8 行）。

2. 引用框布局

相对布局的引用方式和线性布局相同，不再赘述。

4.3.4 表格布局（Table Layout）

常见的表格都是按照固定的列，以行为单位进行排列。图 4-12 所示为一个表格布局的实机界面，该表格为两列两行。

1. 定义表格布局

代码 4-9 是图 4-12 中界面布局的 XML 定义，其根节点是一个表格布局标记。



图 4-12 表格布局的实机界面

代码 4-9 表格布局定义

文件名: table_layout_view.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent"
5      android:stretchColumns="0,1">
6      <TableRow>
7          <CheckBox android:text="复选框"
8              android:layout_width="wrap_content" android:layout_height="wrap_content"/>
9          <Button android:text="按钮"
10              android:layout_width="wrap_content" android:layout_height="wrap_content"/>
11      </TableRow>
12      <TableRow>
13          <RadioButton android:text="单选按钮"
14              android:layout_width="wrap_content" android:layout_height="wrap_content"/>
15          <Button android:text="按钮"
16              android:layout_width="wrap_content" android:layout_height="wrap_content"/>
17      </TableRow>
18  </TableLayout>
```

在代码 4-9 中，标记 TableLayout 表示定义了一个表格布局（第 2 行），该 XML 代码经过布局填充器填充后会创建一个表格布局类实例。标记<TableRow>用于定义表格中的行（第 6 行和第 12 行），行中的单元格（Cell）由子组件“填充”（如第 7 行和第 9 行）。

由此可知，该表格布局包含了两行内容，每行又包含两个子组件。

2. 表格布局的 XML 属性

(1) 扩展列集合

其属性名为 android:stretchColumns，用于指明哪些列可以扩展，其属性值为列编号集

合，列编号基于 0 开始，编号与编号之间用逗号分隔（代码 4-9 第 5 行）。

在代码 4-9 中，第 1 列和第 2 列都允许扩展。允许扩展的列将会自动“侵占”那些不允许扩展列的“地盘”。图 4-13 是只允许第 1 列扩展的实机界面；图 4-14 是只允许第 2 列扩展的实机界面。



图 4-13 设置第 1 列扩展

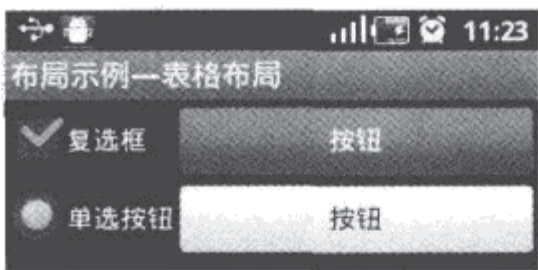


图 4-14 设置第 2 列扩展

(2) 紧缩列集合

其属性名为 `android:shrinkColumns`，用于指明哪些列需要紧缩，其属性值与扩展列属性相同。

(3) 关闭列集合

其属性名为 `android:collapseColumns`，用于指明哪些列关闭（可恢复），其属性值与扩展列属性相同。

4.3.5 绝对布局 (Absolute Layout)

绝对布局中的子组件就有点“各自为政”的味道了，相互之间在位置上没有任何关系，每个组件必须单独指定其位置信息。图 4-15 所示为一个绝对布局的实机界面。

从理论上来说，通过绝对布局可以随心所欲地放置组件，但是在实际操作中，可能会出现因为改变一个组件的位置而造成多个组件的位置都要随着调整的情况，界面的维护工作量要比之前的布局要大得多，所以 Android 平台已经开始废弃绝对布局。



图 4-15 绝对布局的实机界面

1. 定义绝对布局

代码 4-10 是图 4-15 中界面布局的 XML 定义，其根节点是一个绝对布局标记。

代码 4-10 绝对布局定义

文件名: `absolute_layout_view.xml`

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <Button android:text="按钮"
6          android:layout_width="wrap_content" android:layout_height="wrap_content"
7          android:layout_x="10dp" android:layout_y="10dp"/>
8      <CheckBox android:text="复选框"
```




```
9         android:layout_width="wrap_content" android:layout_height="wrap_content"
10        android:layout_x="100dp" android:layout_y="10dp"/>
11        <RadioButton android:text="单选按钮"
12        android:layout_width="wrap_content" android:layout_height="wrap_content"
13        android:layout_x="50dp" android:layout_y="50dp"/>
14    </AbsoluteLayout>
```

在代码 4-10 中，标记 `AbsoluteLayout` 表示定义了一个绝对布局（第 2 行），该 XML 代码经过布局填充器填充后会创建一个绝对布局类实例。该布局中包含 3 个子组件：按钮（第 5 行）、复选框（第 8 行）和单选按钮（第 11 行）。

2. 绝对布局中子组件的 XML 属性

表 4-4 是绝对布局中子组件用于描述其位置的属性及说明。

表 4-4 绝对位置描述属性及说明

属 性	说 明
<code>android:layout_x</code>	子组件的 x 轴位置
<code>android:layout_y</code>	子组件的 y 轴位置

4.3.6 小结——布局的选择

至此，对 Android 平台中的布局已经全部介绍完了。通过以上的详细介绍，相信读者应该对各个布局的特性已经逐渐清晰了：“直来直去”的线性布局、“看菜吃饭”的相对布局、“各自为政”的绝对布局，还有播放幻灯片的框布局和简洁直观的表格布局。

读者可以结合实际应用的特征来选择适合的界面布局，但是需要提醒读者的是：在 Android 平台中，布局类是归纳在小部件包中的，也就是说，从大的方面看，上述的这些基本布局可能就是屏幕中的某一小部分，一个屏幕内容可能是通过多个布局与布局、布局与组件进行嵌套、组合而成，这才真正是读者进行界面设计的目标。

4.4 视图组件（Views）

对于布局、视图和小部件，从本质而言，这三者都是显示组件；但是从角色定义上，如果说布局注重整体（好比设计图），那么小部件注重的是细节（好比装饰品），而视图注重的是组件的组合（就像组合家具）。

4.4.1 视图的使用模式

1. 视图组件的定义

与布局组件的定义一样，视图组件的定义也有在运行时创建和使用 XML 这两种途径；与布局定义不同的是，布局标记一般都是布局资源的根节点，而视图组件的标记既有根节点的（如画廊视图、网页视图等），也有包含于布局之内的（如列表视图、滚动视图等）。

2. 引用视图组件

对于视图组件的使用,在此主要介绍对 XML 所定义的视图组件的引用。由于视图组件既有根组件,也有子组件,所以会导致两种不同的引用方式:

- 1) 对于根组件,其引用方式和布局相同。
- 2) 对于子组件,需要通过其资源 ID 来获取对象实例。

3. 视图组件的事件响应

当用户通过组件与应用程序进行交互时(如单击按钮),需要组件对用户事件进行响应。与 J2SE 平台一样,Android 平台为视图组件定义了用于捕获各种事件的侦听器(Listener),视图组件使用这些侦听器捕获用户事件并响应。

有关对用户事件的侦听,可以通过两种形式来实现。

- (1) 单独定义事件侦听器,再与组件进行绑定

代码 4-11 是采用单独定义事件侦听器的方式响应用户事件的示例代码。

代码 4-11 使用事件侦听器响应用户事件

文件名: EventHandleAct.java

```

1  public class EventHandleAct extends Activity {
2      //初始化单击事件侦听器
3      private OnClickListener mListener = new OnClickListener() {
4          @Override
5          public void onClick(View v) {
6              if(v.getId() == R.id.btn1) {
7                  Toast.makeText(EventHandleAct.this,
8                      "已单击!", Toast.LENGTH_SHORT).show();
9              }
10         }
11     };
12
13     @Override
14     public void onCreate(Bundle savedInstanceState) {
15         super.onCreate(savedInstanceState);
16         setContentView(R.layout.main);
17         //获取按钮控件并设置单击事件侦听器
18         Button btn = (Button)findViewById(R.id.btn1);
19         btn.setOnClickListener(mListener);
20     }
21 };

```

在代码 4-11 中,第 3~11 行定义了一个单击事件的侦听器实例(mListener),在第 19 行中通过方法 setOnClickListener 将该侦听器注册给按钮组件。在该侦听器实例的定义体中,实现了单击事件的回调方法(onClick),在该函数体中,判断事件相关的组件是否按钮,是则弹出一个提示信息界面。图 4-16 是在屏幕中单



图 4-16 按钮单击事件响应的实机界面



击按钮后的实机界面。

Android 平台为组件定义了丰富的事件侦听器。表 4-5 是一些常用侦听器的说明。

表 4-5 View 组件的事件侦听器类型

事件侦听器类型	对应的事件	说 明
OnClickListener	onClick	单击事件
OnLongClickListener	onLongClick	长单击事件
OnFocusChangeListener	onFocusChange	焦点改变事件
OnKey	onKey	击键事件
OnTouch	onTouch	触摸事件
OnCreateContextMenu	onCreateContextMenu	创建上下文菜单事件

(2) 实现侦听器，重载其事件回调方法

代码 4-12 是通过 Activity 组件实现侦听器，并重载其事件回调方法的示例代码。

代码 4-12 重载侦听器事件回调方法

文件名: EventHandleAct.java

```
1 public class EventHandleAct extends Activity implements OnClickListener {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6         //获取按钮控件并设置单击事件侦听器
7         Button btn = (Button)findViewById(R.id.btn1);
8         btn.setOnClickListener(this);
9     }
10    @Override
11    public void onClick(View v) {
12        if(v.getId() == R.id.btn1) {
13            Toast.makeText(EventHandleAct.this,
14                           "已单击!", Toast.LENGTH_SHORT).show();
15        }
16    }
17 };
```

在代码 4-12 中，Activity 组件实现了单击事件侦听器（第 1 行），当把按钮的单击事件侦听器指定为 Activity 组件后（第 8 行），在 Activity 组件的单击事件回调方法中即可对用户单击动作进行响应（第 11 行）。

4.4.2 常用视图

图 4-17 是视图组件的继承关系层次结构图，其中包含了 Android 平台中几乎所有重要的视图组件。



图 4-17 视图组件的继承关系层次结构示意图

下面对几个常用视图的使用进行详细介绍，依次为适配器视图、列表视图、扩展列表视图、网格视图、画廊视图、栈视图、滚动视图、选项页控制器、视图切换器、视图翻转器、滑动抽屉、表面视图、地图视图和网页视图。

1. 适配器视图（AdapterView）

从图 4-17 中读者可能会发现一个比较重要的类：适配器视图，因为它及其子类占领了半壁江山。那么适配器视图究竟有什么独特之处呢？这恐怕要从适配器（Adapter）说起。

（1）适配器与适配器视图

适配器的概念来自 Java 平台中所用到的设计模式，适配器担当视图和该视图所依赖的数据之间的桥梁。例如，对于一个显示联系人信息的列表组件，该组件依赖联系人数据，当增加或删除联系人时，界面中需要及时更新数据的内容。出于设计的弹性，数据更新通知不是数据容器直接发送给界面组件，而是通过适配器这个“中间人”来完成。

适配器视图就是内建了适配器的视图组件，其主要用于展示数据，数据内容依赖于适配器所定义的数据形式（字符串、游标记录等），如图 4-18 所示。

图 4-19 是 Android 平台主要适配器对象的继承关系层次结构图。



图 4-18 适配器视图示意图



图 4-19 适配器继承关系层次结构示意图



(2) 适配器视图的使用

适配器视图类是一个抽象类，无法直接创建该类的实例，而只能通过继承来获取子类实例，或者直接使用 Android 平台中定义的适配器视图的子类。接下来要介绍的列表视图和网格视图都是适配器视图的子类。

适配器视图的使用无外乎对数据的管理和展示，其一般的使用模式如下。

- 1) 初始化数据接口（数组或游标）。
- 2) 使用数据容器初始化适配器（列表适配器或游标适配器）。
- 3) 使用已初始化的适配器作为适配器视图的适配器。
- 4) 通过适配器管理数据记录并通知适配器视图数据集发生改变。

后面即将介绍的列表视图、扩展列表视图、网格视图、画廊视图和栈视图都是界面设计中应用较为广泛的适配器视图。

2. 列表视图（List View）

列表视图几乎是所有开发平台都非常喜欢的组件，无论是作为标准的 Win32 组件，还是标准的 Java 基础类（Java Foundation Classes, JFC）组件，列表视图的功能都是高深莫测的。图 4-20 是使用列表视图显示人员职位信息的实机界面。

(1) 主界面布局定义

代码 4-13 是列表视图示例程序的主界面布局定义，该布局中只包含了一个列表视图。



图 4-20 列表视图的实机界面

代码 4-13 列表视图示例程序的主界面布局定义

文件名: main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent" android:layout_height="fill_parent"
5     android:paddingLeft="4sp" android:paddingRight="4sp">
6     <ListView android:id="@id/android:list"
7         android:layout_width="fill_parent" android:layout_height="fill_parent"/>
8 </LinearLayout>
```

在代码 4-13 中，该列表视图的 ID 属性为 @id/android:list，表示该组件参考的是系统列表视图。

提示：因为示例程序的 Activity 组件是列表 Activity，其内建了列表视图，完全可以无需再指定内容视图；而如果要指定内容视图，则必须保证该布局中有一个 ID 参考系统列表视图资源的列表视图组件，否则会抛出如下运行时错误。

```
java.lang.RuntimeException:
    Your content must have a ListView whose id attribute is 'android.R.id.list'
```

(2) 应用程序 Activity 组件框架

代码 4-14 是该列表视图示例程序的 Activity 组件的框架定义。

代码 4-14 列表视图程序的 Activity 组件的框架定义

文件名: ListViewAct.java

```
1  public class ListViewAct extends ListActivity {
2      //数据容器
3      private ArrayList<HashMap<String,String>> mItems = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //初始化列表项目
10         mItems = new ArrayList<HashMap<String,String>>();
11         addItem("张三", "市场总监");
12         addItem("李四", "技术总监");
13         addItem("王五", "首席执行官");
14         //创建列表数据适配器
15         ListAdapter adapter = new SimpleAdapter(ListViewAct.this,
16             mItems, R.layout.row_ui, IConfig.COLS, IConfig.IDS);
17         //设置页眉和页脚
18         this.getListView().addHeaderView(
19             this.getLayoutInflater().inflate(R.layout.header_view, null),
20             null, false);
21         this.getListView().addFooterView(
22             this.getLayoutInflater().inflate(R.layout.footer_view, null),
23             null, false);
24         //设置数据适配器, 绑定数据
25         setListAdapter(adapter);
26     }
27
28     //添加记录
29     private void addItem(String name, String pos) {
30         HashMap<String,String> item = new HashMap<String,String>();
31         item.put(IConfig.COLS[0], name);
32         item.put(IConfig.COLS[1], pos);
33         mItems.add(item);
34     }
35
36     @Override
37     protected void onListItemClick(ListView l, View v, int pos, long id) {
38         HashMap<String,String> item =
39             (HashMap<String,String>)(this.getListAdapter().getItem(pos-1));
40         //HashMap<String,String> item = (HashMap<String,String>)(mItems.get(pos-1));
41         //记录信息
42         String hint = item.get(IConfig.COLS[0])+"\t"+item.get(IConfig.COLS[1]);
43         //提示消息
```




```
44      Toast.makeText(this, hint, Toast.LENGTH_SHORT).show();
45      ((TextView)findViewById(R.id.text)).setText(hint);
46  }
47  };
```

在代码 4-14 中，首先是初始化列表项数据容器（第 10~13 行），然后是创建适配器（第 15 行），最后将该适配器设置为该 Activity（列表 Activity）的列表适配器（第 25 行）。在列表项单击事件回调方法中，根据列表项位置通过列表适配器或者直接从数据集获得该项对应的数据记录（第 38 行或第 40 行）。

适配器的初始化不仅需要数据容器，还需要指定行视图、记录容器、数据项列表和显示列内容的组件 ID 列表（第 16 行）。其中，行视图中必须包含显示列内容的组件（行视图可以包含其他组件，但必须首先满足包含 ID 列表中的组件）；数据项列表顺序应该与显示列的顺序一致（数据项与显示组件的对应按照其列表顺序而定）。这样一来，适配器才能明确，由哪个组件显示哪项数据，如图 4-21 所示。

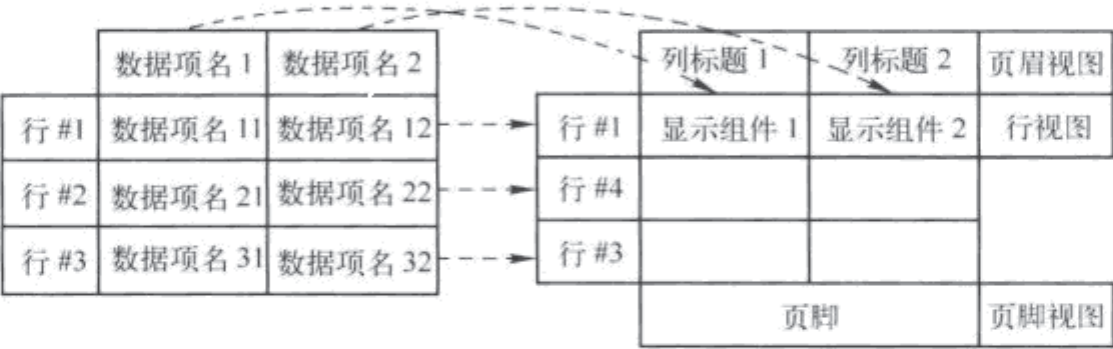


图 4-21 数据记录与行视图的对应示意图

(3) 应用程序配置信息接口

代码 4-15 是应用程序中有关配置信息接口的定义，包括记录的数据项数组和显示列内容的组件 ID 数组。

代码 4-15 应用程序配置信息接口的定义

文件名: IConfig.java

```
1  public interface IConfig {
2      public static final String[] COLS = { "_name", "_pos" };
3      public static final int[] IDS = { R.id.txtName, R.id.txtPos };
4  };
```

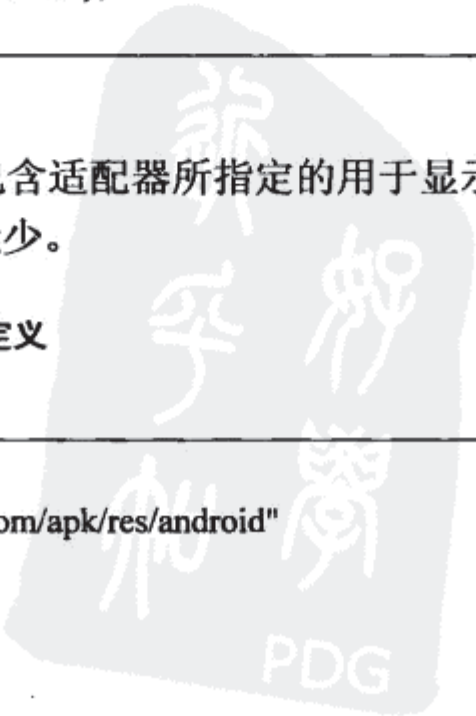
(4) 列表视图的行视图

代码 4-16 是该列表视图的行视图的定义，其中必须包含适配器所指定的用于显示数据项的组件，其组件数可以比数据记录的数据项多，但是不能少。

代码 4-16 列表视图的行视图的定义

文件名: row_ui.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
```




```

3      android:orientation="horizontal"
4      android:layout_width="fill_parent" android:layout_height="wrap_content"
5      android:stretchColumns="0,1">
6      <TableRow>
7          <TextView android:id="@+id/txtName" android:gravity="center"
8              android:layout_width="fill_parent" android:layout_height="40px"
9              android:text="姓名"/>
10         <TextView android:id="@+id/txtPos" android:gravity="center"
11             android:layout_width="fill_parent" android:layout_height="40px"
12             android:text="职位"/>
13     </TableRow>
14 </TableLayout>

```

(5) 设置列表视图的页眉页脚

在代码 4-14 中，使用内建列表视图的 `addHeaderView` 方法可添加列表视图的页眉；使用 `addFooterView` 方法可以添加列表视图的页脚。页眉页脚的添加必须在设置适配器之前，因为页眉和页脚视图会影响适配器对行视图的计算（这也是为什么代码 4-14 中第 39 行的列表项索引需要减 1 的原因，因为页眉视图也算做一行）；如果违背该规则，则会抛出如下运行时异常。

```

java.lang.IllegalStateException:
    Cannot add header view to list -- setAdapter has already been called

```

代码 4-17 是列表视图的页眉视图的定义。

代码 4-17 列表视图的页眉视图的定义

文件名: header_view.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="horizontal"
4      android:layout_width="fill_parent" android:layout_height="wrap_content"
5      android:stretchColumns="0,1">
6      <TableRow>
7          <TextView android:text="姓名" android:gravity="center"
8              android:layout_width="fill_parent" android:layout_height="50px"
9              android:textSize="8pt" android:textColor="#FF00FF00"/>
10         <TextView android:text="职位" android:gravity="center"
11             android:layout_width="fill_parent" android:layout_height="50px"
12             android:textSize="8pt" android:textColor="#FF00FF00"/>
13     </TableRow>
14     <TextView android:background="#FF00FF00"
15         android:layout_width="fill_parent" android:layout_height="1dp"/>
16 </TableLayout>

```

代码 4-18 是列表视图的页脚视图的定义。



代码 4-18 列表视图的页脚视图的定义

文件名: footer_view.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent" android:layout_height="wrap_content">
5     <TextView android:id="@+id/text" android:background="#FF333333"
6         android:layout_width="fill_parent" android:layout_height="40px"
7         android:gravity="center"/>
8 </LinearLayout>
```

(6) 列表视图的事件响应

表 4-6 是与列表视图相关的, 较为重要的事件侦听器。

表 4-6 列表视图的事件侦听器

属 性	说 明
OnItemClickListener	用于侦听列表项被单击
OnItemLongClickListener	用于侦听列表项被单击并长按
OnItemSelectedListener	用于侦听列表项被选择

使用列表视图设置侦听器的方法即可设置对表 4-6 中对应侦听器的绑定。

3. 扩展列表视图 (Expandable ListView)

顾名思义, 扩展列表视图是在列表视图的基础之上进行了扩展, 其组织形式要比列表视图更为复杂。图 4-22 所示为扩展列表视图的实机界面, 其列表项中又嵌套了列表。



图 4-22 扩展列表视图的实机界面

(1) 主界面布局定义

代码 4-19 是扩展列表视图示例程序的主界面布局定义, 该布局中只包含了一个扩展列表视图。

代码 4-19 扩展列表视图示例程序的主界面布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <ExpandableListView android:id="@id/android:list"
6          android:layout_width="fill_parent" android:layout_height="fill_parent"
7          android:background="#996633"/>
8  </LinearLayout>

```

在代码 4-19 中, 该列表视图的 ID 属性为 `@id/android:list`, 表示该组件参考的是系统扩展列表视图。

提示: 因为示例程序的 Activity 组件是扩展列表 Activity, 其内建了扩展列表视图, 完全可以无需再指定内容视图; 而如果要指定内容视图, 则必须保证该布局中有一个 ID 参考系统扩展列表视图资源的组件, 否则也会抛出运行时错误。

(2) 应用程序 Activity 组件框架

代码 4-20 是该列表视图示例程序的 Activity 组件的框架定义。

代码 4-20 扩展列表视图示例程序的 Activity 组件的框架定义

文件名: SimpleElvAct.xml

```

1  public class SimpleElvAct extends ExpandableListActivity {
2      //组数据容器
3      private ArrayList<HashMap<String,String>> mGroups =
4          new
5      ArrayList<HashMap<String,String>>();
6      //子项数据容器
7      private ArrayList<ArrayList<HashMap<String, String>>> mPeople =
8          new ArrayList<ArrayList<HashMap<String, String>>>();
9      @Override
10     public void onCreate(Bundle savedInstanceState) {
11         super.onCreate(savedInstanceState);
12         setContentView(R.layout.main);
13         //初始化数据容器
14         initChildren();
15         //创建扩展列表视图适配器
16         ExpandableListAdapter adapter = new SimpleExpandableListAdapter(this,
17             mGroups, android.R.layout.simple_expandable_list_item_1,
18             new String[] { IConfig.GR_KEY }, new int[] { android.R.id.text1 },
19             mPeople, android.R.layout.simple_expandable_list_item_2,
20             IConfig.COLS, new int[] { android.R.id.text1, android.R.id.text2 });
21         this.setAdapter(adapter);

```




```

22     }
23
24     @Override
25     public boolean onChildClick(ExpandableListView parent, View v,
26         int groupPos, int childPos, long id) {
27         //获取子项
28         People people = new People(mPeople.get(groupPos).get(childPos));
29         Toast.makeText(this, people.toString(), Toast.LENGTH_LONG).show();
30         return super.onChildClick(parent, v, groupPos, childPos, id);
31     }
32
33     private void initChildren() { //初始化子项
34         //同学组
35         Group group1 = new Group("同学");
36         mGroups.add(group1.getData());
37
38         PeopleGroup pg1 = new PeopleGroup();
39         //人员 1
40         People people1 = new People("张三", "139-1234-5678");
41         pg1.addPeople(people1);
42         //人员 2
43         People people2 = new People("李四", "135-6789-5432");
44         pg1.addPeople(people2);
45         mPeople.add(pg1.getData());
46
47         //朋友组
48         Group group2 = new Group("朋友");
49         mGroups.add(group2.getData());
50
51         PeopleGroup pg2 = new PeopleGroup();
52         //人员 3
53         People people3 = new People("王五", "134-2345-7890");
54         pg2.addPeople(people3);
55         //人员 4
56         People people4 = new People("赵六", "135-3456-6789");
57         pg2.addPeople(people4);
58         mPeople.add(pg2.getData());
59     }
};

```

在代码 4-20 中，首先是初始化列表项数据组和子项容器（initChildren 方法中），然后是创建适配器（第 15 行），最后将该适配器设置为该 Activity（扩展列表 Activity）的列表适配器（第 20 行）。在子项单击事件回调方法中，根据子项位置从数据集获得该项对应的数据记录（第 27 行）。相比列表视图，扩展列表视图的复杂点在其数据结构层次。

扩展列表视图的数据容器分为两部分：组数据和子项数据。每部分数据都需要指定布局、数据项和显示组件 ID（可以将扩展视图适配器的构造方法的参数分为两部分，每一部

分 4 个参数)。由此可知，扩展列表视图是列表视图中嵌套列表视图：当组列表项没展开时（图 4-22 中左图），其界面是一个列表视图；当展开其中某一组列表项后，其子项又是一个列表视图（图 4-22 中右图），所以其数据容器是一个列表嵌套列表（第 6 行），存在两层。

从扩展列表视图的适配器定义来看，组数据和子项数据在数据内容上可以不存在关联，那么如何确定哪些子项归属于某个组列表项呢？通过扩展列表视图的子项单击事件回调方法（第 24 行）可知：子项的定位需要同时使用组索引和子索引（第 27 行）。由此可知，子项与组列表项是通过存储位置来进行对应：组数据第 2 项对应子项数据中的第一层的第 2 项，而在第 2 项下的子项都属于组数据第 2 项的子项，其关系如图 4-23 所示。

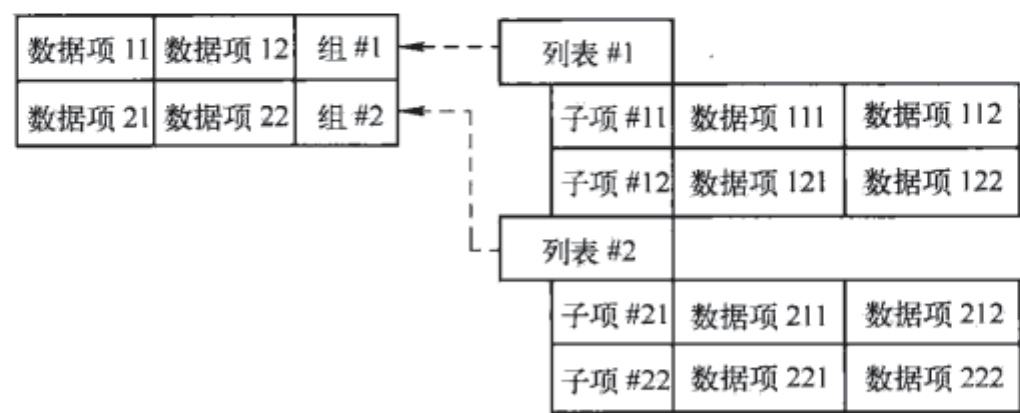


图 4-23 组数据与子项数据的归集关系

(3) 应用程序配置信息接口

代码 4-21 是应用程序中有关配置信息接口的定义，包括组列表项和子项的数据项。

代码 4-21 应用程序配置信息接口的定义

文件名: IConfig.java

```
1 public interface IConfig {
2     public static final String PP_KEY1 = "_name";
3     public static final String PP_KEY2 = "_number";
4     public static final String GR_KEY = "_group";
5     //子数据项列表
6     public static final String COLS[] = { PP_KEY1, PP_KEY2 };
7 };
```

(4) 扩展列表视图的事件响应

表 4-7 是与扩展列表视图相关的、较为重要的事件侦听器。

表 4-7 扩展列表视图的事件侦听器

属 性	说 明
OnChildClickListener	用于侦听子项被单击
OnGroupClickListener	用于侦听组列表项被单击
OnGroupCollapseListener	用于侦听组列表项关闭
OnGroupExpandListener	用于侦听组列表项展开

使用扩展列表视图设置侦听器的方法即可设置对表 4-7 中对应侦听器的绑定。



4. 网格视图 (Grid View)

顾名思义, 网格视图将其子组件以二维滚动网格的形式进行展现。图 4-24 所示为网格视图的实机界面。通过图 4-17 可知, 网格视图也是适配器视图, 而且还是抽象列表视图的子类 (和列表视图是兄弟), 但其形式不是表列, 而是二维网格。



图 4-24 网格视图的实机界面

(1) 主界面布局定义

代码 4-22 是网格视图示例程序的主界面布局定义, 其根组件为网格视图。

代码 4-22 网格视图示例程序的主界面布局定义

文件名: main_view.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <GridView xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@+id/gdview"
4      android:layout_width="fill_parent" android:layout_height="fill_parent"
5      android:columnWidth="150px" android:numColumns="auto_fit"
6      android:verticalSpacing="5px" android:horizontalSpacing="5px"
7      android:stretchMode="columnWidth" android:gravity="center" />

```

(2) 应用程序 Activity 组件框架

代码 4-23 是网格视图示例程序的 Activity 组件的框架定义。

代码 4-23 网格视图示例程序的 Activity 组件的框架定义

文件名: GridViewerAct.java

```

1  public class GridViewerAct extends Activity {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main_view);
6          //初始化适配器
7          FooImageAdapter adapter = new FooImageAdapter(this);
8          //设置网格视图适配器
9          GridView gridView = (GridView)findViewById(R.id.gdview);

```



```

10         gridView.setAdapter(adapter);
11     }
12 };

```

在代码 4-23 中，因为网格视图是适配器视图，所以其需要设置适配器（第 10 行）。与列表视图不同的是，网格视图的适配器所处理的数据既不是数组也不是游标，而是图片。

（3）网格视图的适配器

代码 4-24 是代码 4-23 中所提到的网格视图的适配器的定义。

代码 4-24 网格视图的适配器的定义

文件名: FooImageAdapter.java

```

1  public class FooImageAdapter extends BaseAdapter implements ListAdapter {
2      //上下文
3      private Context mContext = null;
4      public FooImageAdapter(Context c) { this.mContext = c; }
5
6      @Override
7      public int getCount() { return (IConfig.IMGS.length); }
8      @Override
9      public Object getItem(int position) { return null; }
10     @Override
11     public long getItemId(int position) { return 0; }
12     @Override
13     public View getView(int position, View convertView, ViewGroup parent) {
14         ImageView iv = null;
15
16         if(convertView == null) {
17             iv = new ImageView(mContext);
18             iv.setAdjustViewBounds(true);
19             iv.setLayoutParams(new GridView.LayoutParams(150,150));
20             iv.setScaleType(ImageView.ScaleType.CENTER_CROP);
21             iv.setPadding(8, 8, 8, 8);
22         } else { iv = (ImageView)convertView; }
23         //设置图片视图的资源
24         iv.setImageResource(IConfig.IMGS[position]);
25         return (iv);
26     }
27 };

```

在代码 4-24 中，读者通过第一行即可知该适配器的由来：继承于基本适配器，同时实现列表适配器。其主要处理方法在 `getView` 方法中，该适配器所管理的组件为图片视图（列表视图中通过行视图来定义），图片视图的数据源由数据容器提供（第 24 行）。

（4）网格视图的事件响应

网格视图与列表视图的适配器相同，其对用户事件的响应方式一致，只是其子项的类型不同而已。



5. 画廊视图 (Gallery)

相比网格视图，画廊视图的艺术性更高：其比网格视图的动态效果更强，其子组件以水平列表的形式进行摆放，且有滚动效果，犹如窗外经历的山水画廊。图 4-25 所示为画廊视图的实机界面。



图 4-25 画廊视图的实机界面

通过图 4-17 可知，画廊视图继承于抽象微调器 (AbsSpinner)，既有列表视图的基础，又有微调器的动态效果。画廊视图也是适配器视图的子类。

(1) 主界面布局定义

代码 4-25 是画廊视图示例程序的主界面布局定义，其根组件为画廊视图。

代码 4-25 画廊视图示例程序的主界面布局定义

文件名: main_view.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <Gallery xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@+id/gallery"
4     android:layout_width="fill_parent" android:layout_height="wrap_content"/>
```

(2) 应用程序 Activity 组件框架

代码 4-26 是画廊视图示例程序的 Activity 组件的框架定义。

代码 4-26 画廊视图示例程序的 Activity 组件的框架定义

文件名: GalleryViewerAct.java

```
1 public class GalleryViewerAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main_view);
6         //初始化适配器
7         FooImageAdapter adapter = new FooImageAdapter(this);
8         //设置画廊视图适配器
9         Gallery gallery = (Gallery)findViewById(R.id.gallery);
10        gallery.setAdapter(adapter);
```



```
11      }  
12  };
```

在代码 4-26 中，因为画廊视图是适配器视图，所以其需要设置适配器（第 10 行）。与网格视图相同，画廊视图的适配器所处理的类型也是图片。

(3) 画廊视图的适配器

画廊视图适配器与网格视图的区别仅是子组件的布局参数（代码 4-24 第 19 行），画廊视图子组件的布局参数如以下代码所示。

```
iv.setLayoutParams(new Gallery.LayoutParams(160,160) );
```

(4) 画廊视图的事件响应

画廊视图与网格视图的事件响应相同，在此不再赘述。

6. 栈视图（Stack View）

栈视图是 Android 平台的新特性，其子组件以栈元素的形式进行堆放，显示在最上层的组件为栈顶元素；当向后浏览时（出栈），次位的组件提升到栈顶，原栈顶组件的显示有两种情形：在循环模式下，其将在栈底显示（如图 4-26 所示）；在非循环模式下，其将不再可见。

栈视图的这种可视效果非常直观而且生动，非常适合选取界面的设计。

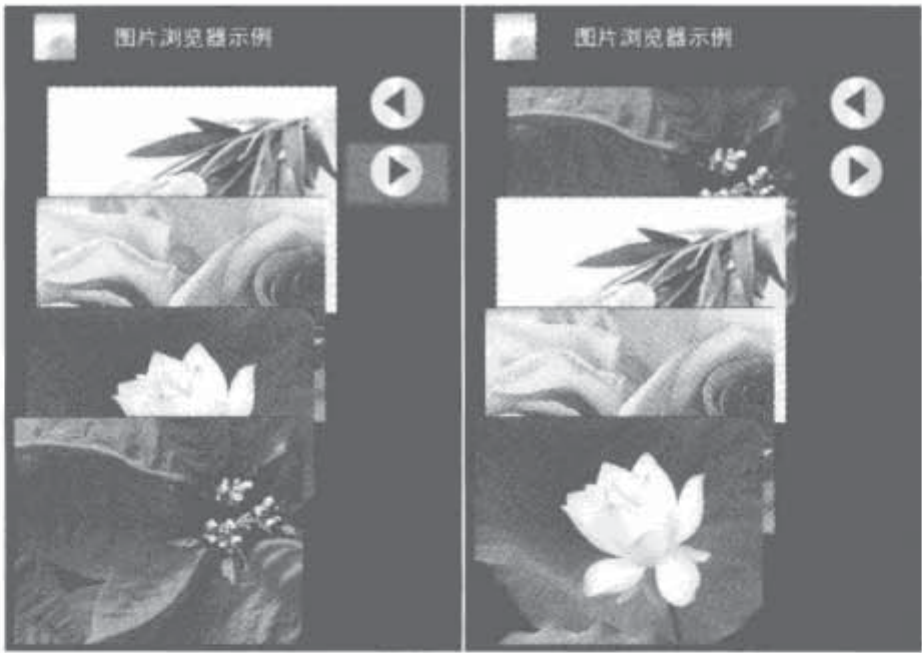


图 4-26 栈视图的实例界面（循环模式）

通过图 4-17 可知，栈视图继承于适配器视图动画（AdapterViewAnimator），其具备适配器视图的本质，又有动画效果。

(1) 主界面布局定义

代码 4-27 是栈视图示例程序的主界面布局定义，其主要组件是栈视图。

代码 4-27 栈视图示例程序的主界面布局定义

文件名: main.xml

```
1  <?xml version="1.0" encoding="utf-8"?>  
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```




Android 平台开发之旅 第2版

```

3      android:orientation="horizontal" android:layout_width="fill_parent"
4      android:layout_height="fill_parent">
5      <StackView android:id="@+id/sview" android:layout_width="fill_parent"
6          android:layout_height="wrap_content" android:loopViews="true"/>
7      <LinearLayout android:orientation="vertical" android:gravity="center"
8          android:layout_height="wrap_content" android:layout_width="wrap_content">
9          <ImageButton android:id="@+id/btn_prev" android:src="@drawable/prev"
10             android:layout_height="wrap_content" android:layout_width="wrap_content"/>
11          <ImageButton android:id="@+id/btn_next" android:src="@drawable/next"
12             android:layout_height="wrap_content" android:layout_width="wrap_content"/>
13      </LinearLayout>
14 </LinearLayout>

```

在代码 4-27 中，栈视图的 `android:loopViews` 属性用于描述其子组件是否为循环模式显示，取值为 `true` 或 `false`（默认值）。

(2) 应用程序 Activity 组件框架

代码 4-28 是栈视图示例程序的 Activity 组件的框架定义。

代码 4-28 栈视图示例程序的 Activity 组件的框架定义

文件名: `ImageViewer3Act.java`

```

1  public class ImageViewer3Act extends Activity implements OnClickListener {
2      //视图容器
3      private StackView mSvView = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //获取视图容器实例
10         this.mSvView = (StackView)findViewById(R.id.sview);
11         //获取按钮控件并设置单击侦听器
12         ImageButton btnPrev = (ImageButton)findViewById(R.id.btn_prev);
13         ImageButton btnNext = (ImageButton)findViewById(R.id.btn_next);
14         btnPrev.setOnClickListener(this);
15         btnNext.setOnClickListener(this);
16         //设置适配器
17         PanelAdapter adapter = new PanelAdapter(this);
18         this.mSvView.setAdapter(adapter);
19     }
20     @Override
21     public void onClick(View v) {
22         switch(v.getId()) {
23             case R.id.btn_prev: { this.mSvView.showPrevious(); break; }
24             case R.id.btn_next: { this.mSvView.showNext(); break; }
25         }

```



```

26      }
27  };

```

在代码 4-28 中，因为栈视图是适配器视图，所以其需要设置适配器，其适配器为定制类型（第 18 行）。

使用栈视图的 `showPrevious` 和 `showNext` 方法用于控制显示前一子组件和后一子组件（第 23 行和第 24 行），在循环模式和非循环模式下，其显示效果存在明显差异。图 4-27 是非循环模式下显示后一子组件的实例效果。

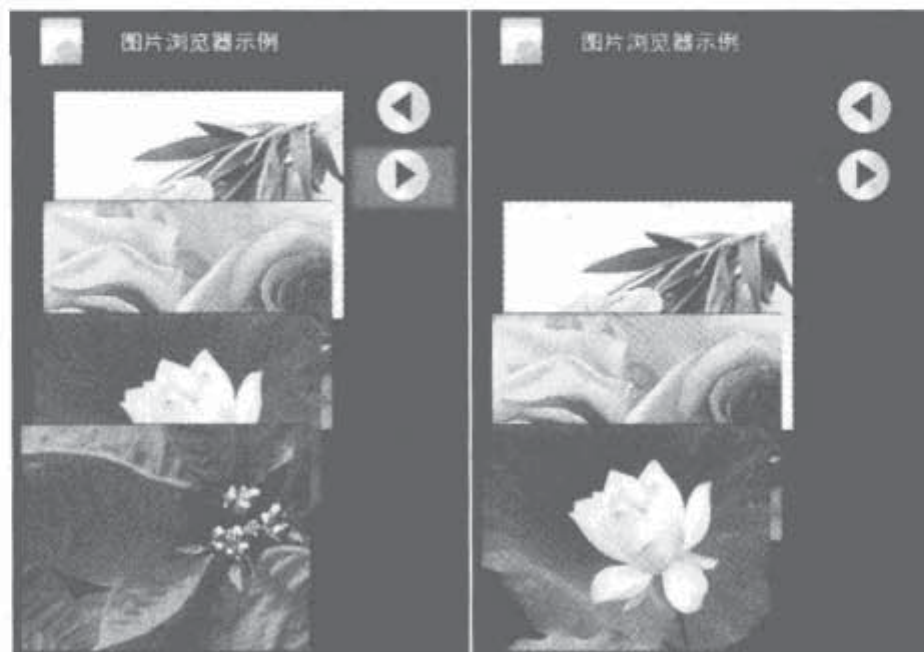


图 4-27 栈视图的实例界面（非循环模式）

（3）栈视图的适配器

代码 4-29 是代码 4-28 中所参考的栈视图的适配器的定义。

代码 4-29 栈视图的适配器的定义

文件名: `PanelAdapter.java`

```

1  public class PanelAdapter extends BaseAdapter {
2      //应用程序上下文
3      private Context mContext = null;
4      public PanelAdapter(Context context) { this.mContext = context; }
5
6      @Override
7      public int getCount() { return (IConfig.IMGS.length); }
8      @Override
9      public Object getItem(int pos) { return(IConfig.IMGS[pos]); }
10     @Override
11     public long getItemId(int pos) { return(pos); }
12     @Override
13     public View getView(int pos, View convertView, ViewGroup parent) {
14         ImageView iv = null;
15
16         if(convertView == null) {

```




```

17         iv = new ImageView(mContext);
18         iv.setAdjustViewBounds(true);
19         iv.setLayoutParams(new LinearLayout.LayoutParams(
20             ViewGroup.LayoutParams.WRAP_CONTENT,
21             ViewGroup.LayoutParams.WRAP_CONTENT));
22         iv.setScaleType(ImageView.ScaleType.MATRIX);
23         iv.setPadding(8, 8, 8, 8);
24     } else { iv = (ImageView)convertView; }
25     //设置图片视图的资源
26     iv.setImageResource(IConfig.IMGS[pos]);
27     return (iv);
28 }
29 };

```

在代码 4-29 中，栈视图的适配器只继承了基本适配器，其子组件的布局采用的是线性布局参数（第 19 行），其子组件类型与网格视图和画廊视图一样，也是图片。

（4）栈视图的事件响应

栈视图也是适配器视图，其事件响应与画廊视图及网格视图相同。

7. 滚动视图（Scroll View）

图 4-28 所示为一款类似于“手机报”工具的实机界面，该程序可以实现自动滚屏或根据选择直接滚动到指定的位置，其中用到的主要组件就是滚动视图（ScrollView）。

滚动视图继承于框布局，所以其具备框布局的使用特性：包含多个组件，组件相互压盖，同时只有一个组件可见。对于滚动视图中的内容，读者可以理解为多个内容片段。



图 4-28 滚动视图的实机界面

（1）主界面布局定义

代码 4-30 是滚动视图示例程序的主界面布局定义，其根组件为滚动视图。

代码 4-30 滚动视图示例程序的主界面布局定义

文件名: main.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@+id/sview" android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="wrap_content"
5      android:scrollbars="vertical" >
6      <LinearLayout android:orientation="vertical"
7          android:layout_width="fill_parent" android:layout_height="wrap_content">
8          <TextView android:text="@string/title" android:textSize="10pt"
9              android:layout_width="fill_parent" android:layout_height="wrap_content"
10             android:padding="8sp" android:gravity="center_horizontal"/>
11          <TextView android:text="@string/song_clarion"
12              android:layout_width="fill_parent" android:layout_height="wrap_content"/>
13          <ImageView android:src="@drawable/hulu" android:scaleType="matrix"
14              android:layout_width="wrap_content" android:layout_height="wrap_content"/>
15      </LinearLayout>
16  </ScrollView>
```

(2) 滚动视图的使用

顾名思义，滚动视图最大的特性就是“滚动”。当滚动视图中的内容超过屏幕大小时，通过滚动可以分屏地浏览其内容。滚动视图提供了两种滚动模式：相对滚动和绝对滚动。相对滚动是相对于视图中当前内容位置的滚动；而绝对滚动是直接滚动到视图中整个内容位置。前者适用于自动滚屏的场合，该场合不关注滚动到具体位置，只关注滚动的差分量；后者适用于内容跳转的场合，可以直接滚动到指定的刻度，如图 4-29 所示。

代码 4-31 是滚动视图自动滚屏的主要代码。

代码 4-31 滚动视图的自动滚屏

文件名: ScrollViewAct.java

```
1  private void doAutoScroll() { //自动滚动
2      mSview.smoothScrollBy(0, IConfig.AUTO_STEP);
3      if(mSview.getScrollY() >= mMaxScrollY) { stopAuto(); }
4  }
5
6  private void stopAuto() { //停止自动滚动
7      mTimer.cancel(); mTimer.purge(); mTimer = null;
8      mIsAuto = false;
9  }
```



图 4-29 滚动视图的滚动示意图

在代码 4-31 中，使用滚动视图的 smoothScrollBy 方法进行相对滚动，刻度差分量为自



动滚屏的步长（第2行）。如果当前刻度大于最大刻度时，则停止自动滚动。

自动滚动的机制是：使用计时器定时向主线程消息队列处理器发送消息，在处理器的回调方法中滚动视图，所以停止滚动实际上是停止计时器（第7行）。

代码 4-32 是滚动视图进行跳转的主要代码。

代码 4-32 滚动视图的跳转

文件名: ScrollViewAct.java

```
1 private void doJump() { //执行跳转
2     switch(group.getCheckedRadioButtonId() ) {
3         case R.id.btn_beg: { mSview.fullScroll(ScrollView.FOCUS_UP); break; }
4         case R.id.btn_end: { mSview.fullScroll(ScrollView.FOCUS_DOWN); break; }
5         case R.id.btn_mid: { double percent = (sbar.getProgress()*1.0f)/100.0f;
6             int pos = (int)(mMaxScrollY*percent);
7             mSview.scrollTo(0, pos);
8             break;
9         }
10    }
11 }
```

在代码 4-32 中，使用滚动视图的 fullScroll 方法可以完全滚动到视图头部和尾部（第3行和第4行）；使用 fullScroll 方法可以滚动到指定位置（第7行）。图 4-30 所示为滚动视图的跳转设置实机界面。



图 4-30 滚动视图的跳转设置实机界面

代码 4-33 是获取滚动视图的最大滚动位置的主要代码。

代码 4-33 获取滚动视图的最大滚动位置

文件名: ScrollViewAct.java

```
1 @Override
2 public boolean onCreateOptionsMenu(Menu menu) {
```



```

3      //计算最大滚动刻度
4      mMaxScrollY = getMaxScrollY(mSview);
5      //填充选项菜单资源
6      this.getMenuInflater().inflate(R.menu.ops_menu, menu);
7      return (super.onCreateOptionsMenu(menu) );
8  }
9
10     protected int getMaxScrollY(ScrollView v) { //获取最大 Y 轴滚动刻度
11         v.setVisibility(View.INVISIBLE);
12         v.fullScroll(View.FOCUS_DOWN);
13         v.fullScroll(View.FOCUS_UP);
14         //滚动到尾部
15         v.fullScroll(View.FOCUS_DOWN);
16         int end = v.getScrollY();
17         v.fullScroll(View.FOCUS_UP); //恢复视图的位置
18         v.setVisibility(View.VISIBLE);
19         return (end);
20     }

```

在代码 4-33 中，在创建选项菜单的回调方法中，获取滚动视图的最大滚动位置（第 4 行）；该过程中，先通过滚动到底端来预先计算最大位置（第 12 行），最后获取尾端的位置（第 16 行）。

除了滚动视图外，后面即将介绍的选项页控制器、视图切换器和视图翻转器都是框布局的子类，在子组件（视图）的切换显示方面各有风格。

8. 选项页控制器（TabHost）

选项页控制器属于标准界面组件，与滚动视图一样，都是框布局的直接子类。选项页控制器比滚动视图更为明显地表现了框布局的特性：通过选项页标签实现页面的切换，同一时间只能有一个页面可见。图 4-31 是选项页控制器的实机界面，该选项页控制器定义了两个选项页：第一个选项页显示文本；第二个选项页显示图片。

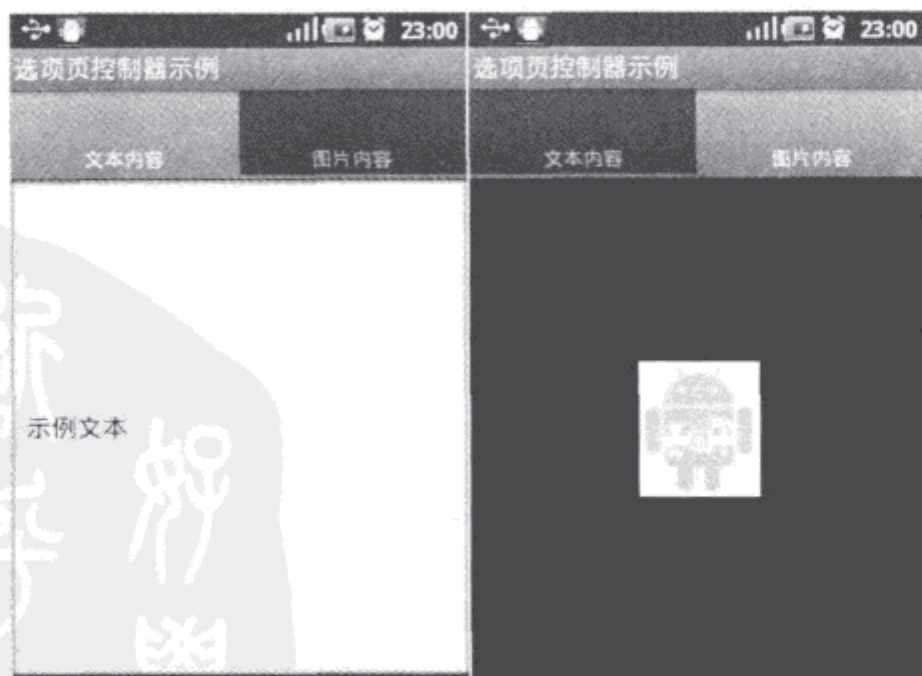


图 4-31 选项页控制器的实机界面



(1) 主界面布局定义

代码 4-34 是选项页控制器示例程序的主界面布局定义，其根组件为选项页控制器。

代码 4-34 选项页控制器示例程序的主界面布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <TabHost xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@android:id/tabhost"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <LinearLayout android:orientation="vertical"
6          android:layout_width="fill_parent" android:layout_height="fill_parent">
7          <TabWidget android:id="@android:id/tabs"
8              android:layout_width="fill_parent" android:layout_height="wrap_content"/>
9          <FrameLayout android:id="@android:id/tabcontent"
10              android:layout_width="fill_parent" android:layout_height="fill_parent">
11              <EditText android:id="@+id/text" android:text="示例文本"
12                  android:layout_width="fill_parent" android:layout_height="fill_parent"
13                  android:editable="false"/>
14              <ImageView android:id="@+id/img"
15                  android:layout_width="fill_parent" android:layout_height="fill_parent"
16                  android:src="@drawable/android" android:scaleType="center"/>
17          </FrameLayout>
18      </LinearLayout>
19  </TabHost>

```

(2) 应用程序 Activity 组件框架

代码 4-35 是选项页控制器示例程序的 Activity 组件的框架定义。

代码 4-35 选项页控制器示例程序的 Activity 组件的框架定义

文件名: TabHostAct.java

```

1  public class TabHostAct extends TabActivity {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main);
6          //获取 TabHost 组件对象实例
7          TabHost tabHost = getTabHost();
8          //初始化 TabHost 的选项页
9          tabHost.addTab(tabHost.newTabSpec("t1")
10              .setIndicator("文本内容").setContent(R.id.text) );
11          tabHost.addTab(tabHost.newTabSpec("t2")
12              .setIndicator("图片内容").setContent(R.id.img) );
13          //设置当前选项页
14          tabHost.setCurrentTab(1);
15      }

```



16 };

在代码 4-35 中，通过选项页控制器的 Activity 组件来搭建程序框架（第 1 行），通过其 `getTabHost` 方法可获取内建的选项页控制器（第 7 行）。通过控制器的 `addTab` 方法添加选项页（第 9 行和第 11 行）。

(3) 选项页控制器的事件响应

表 4-8 是与选项页控制相关的、较为重要的事件侦听器。

表 4-8 选项页控制器的事件侦听器

属 性	说 明
OnTabChangeListener	用于侦听选项页切换

使用选项页控制器设置侦听器的方法即可设置表 4-8 中对应侦听器的绑定。

9. 视图切换器 (ViewSwitcher)

相比选项页控制器，视图切换器更为专业：其主要用于对子视图（仅限于两个）进行快速切换的场合，同一时刻只显示一个子视图。图 4-32 是视图切换器示例程序的实机界面。



图 4-32 视图切换器示例程序的实机界面

(1) 主界面布局定义

代码 4-36 是视图切换器示例程序的主界面布局定义，其主要组件为视图切换器。

代码 4-36 视图切换器示例程序的主界面布局定义

文件名: main.xml

```
1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2     android:orientation="vertical" android:layout_width="fill_parent"
3     android:layout_height="fill_parent">
4     <ViewSwitcher android:id="@+id/vsview" android:layout_width="fill_parent"
```




Android 平台开发之旅 第2版

```

5  <?xml version="1.0" encoding="utf-8"?>
6      android:layout_height="300px" />
7      <LinearLayout android:gravity="center"
8          android:layout_height="wrap_content" android:layout_width="fill_parent">
9          <ImageButton android:id="@+id/btn_first" android:src="@drawable/first"
10             android:layout_height="wrap_content" android:layout_width="wrap_content"/>
11          <ImageButton android:id="@+id/btn_prev" android:src="@drawable/prev"
12             android:layout_height="wrap_content" android:layout_width="wrap_content"/>
13          <ImageButton android:id="@+id/btn_next" android:src="@drawable/next"
14             android:layout_height="wrap_content" android:layout_width="wrap_content"/>
15          <ImageButton android:id="@+id/btn_last" android:src="@drawable/last"
16             android:layout_height="wrap_content" android:layout_width="wrap_content"/>
17      </LinearLayout>
18 </LinearLayout>

```

代码 4-36 中定义的 4 个图片按钮用于控制图片播放，分别是首幅、前一幅、后一幅和末幅。

(2) 应用程序 Activity 组件框架

代码 4-37 是视图切换器示例程序的 Activity 组件的框架定义。

代码 4-37 视图切换器示例程序的 Activity 组件的框架定义

文件名: ImageViewerAct.java

```

1  public class ImageViewerAct extends Activity implements OnClickListener {
2      //视图切换器
3      private ViewSwitcher mVsView = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //获取视图切换器实例
10         this.mVsView = (ViewSwitcher)findViewById(R.id.vswitcher);
11         //添加子视图
12         this.mVsView.addView(new ImagePanel(this,1,1),0);
13         this.mVsView.addView(new ImagePanel(this,2,1),1);
14         //获取按钮控件并设置单击侦听器
15         .....
16     }
17     @Override
18     public boolean onCreateOptionsMenu(Menu menu) {
19         this.getMenuInflater().inflate(R.menu.ops_menu, menu);
20         return super.onCreateOptionsMenu(menu);
21     }
22     @Override
23     public boolean onOptionsItemSelected(MenuItem item) {

```



```

24         switch(item.getItemId()) {
25             case R.id.mi_single: { this.mVsView.showNext(); break; }
26             case R.id.mi_dual: { this.mVsView.showPrevious(); break; }
27         }
28         return super.onOptionsItemSelected(item);
29     }
30     @Override
31     public void onClick(View v) {
32         ImagePanel panel = (ImagePanel)this.mVsView.getCurrentView();
33         switch(v.getId()) {
34             case R.id.btn_first: { panel.showFirstPage(); break; }
35             case R.id.btn_prev: { panel.showPrevPage(); break; }
36             case R.id.btn_next: { panel.showNextPage(); break; }
37             case R.id.btn_last: { panel.showLastPage(); break; }
38         }
39     }
40 };

```

在代码 4-37 中，首先获取视图切换器实例（第 10 行），然后通过其 `addView` 方法添加子视图（第 12 行和第 13 行）。在代码 4-37 中，视图的切换使用选项菜单项来完成（第 32 行和第 33 行），使用视图切换器的 `showNext` 或 `showPrevious` 方法在两个子视图之间进行切换；或者使用切换器的 `setDisplayChild` 方法可以直接显示指定视图。

使用视图切换器的 `getCurrentView` 方法可获取当前显示的子视图（第 39 行），即可调用其子视图的相关方法（第 41~44 行）。

需要注意的是，视图切换器只支持包含两个子视图，如果添加超过两个子视图，则会抛出以下运行时异常。

Caused by: java.lang.IllegalStateException: Can't add more than 2 views to a ViewSwitcher

（3）子视图的定义

代码 4-38 是视图切换器示例程序中子视图的定义，其实际上是一个线性布局。

代码 4-38 视图切换器子视图

文件名: ImagePanel.java

```

1  public class ImagePanel extends LinearLayout {
2      private final int mColCount;
3      private final int mRowCount;
4      private final int mCountPerPage;
5      private final int mPageCount;
6      private Context mContext = null;
7      private ArrayList<ImageView> mImages = null;
8      private int mPageNo = 0;
9
10     public ImagePanel(Context context, int colCount, int rowCount) { super(context);
11         this.mContext = context;

```




```

12         this.mColCount = colCount; this.mRowCount = rowCount;
13         this.mCountPerPage = (rowCount*colCount);
14         this.mPageCount = (IConfig.IMGS.length%this.mCountPerPage==0) ?
15             (IConfig.IMGS.length/this.mCountPerPage) :
16             (IConfig.IMGS.length/this.mCountPerPage) + 1;
17         init();
18     }
19
20     private void init() { //初始化视图
21         this.setOrientation(LinearLayout.VERTICAL);
22         this.setLayoutParams(new LinearLayout.LayoutParams(
23             ViewGroup.LayoutParams.FILL_PARENT,
24             ViewGroup.LayoutParams.FILL_PARENT));
25         this.setGravity(Gravity.CENTER);
26         mImages = new ArrayList<ImageView>(mCountPerPage);
27         //初始化图片视图
28         for(int i = 0; i < this.mRowCount; ++i) {
29             LinearLayout row = new LinearLayout(this.mCtx);
30             row.setOrientation(LinearLayout.HORIZONTAL);
31             row.setLayoutParams(new LinearLayout.LayoutParams(
32                 ViewGroup.LayoutParams.FILL_PARENT,
33                 ViewGroup.LayoutParams.WRAP_CONTENT));
34             row.setGravity(Gravity.CENTER);
35             for(int j = 0; j < this.mColCount; ++j) {
36                 ImageView iv = new ImageView(this.mCtx);
37                 iv.setImageResource(IConfig.IMGS[mPageNo+j+(i*this.mRowCount)]);
38                 row.addView(iv);
39                 this.mImages.add(iv);
40             }
41             this.addView(row);
42         }
43     }
44     .....
45 };

```

在代码 4-38 中，切换器的子视图实际上是一个垂直方向的线性布局（第 21 行），其中包含指定列数和行数的图片视图（第 28~42 行），类似于网格视图。

10. 视图翻转器 (ViewFlipper)

从视图切换功能而言，视图翻转器是视图切换器的增强型组件，其可以支持多个子视图的切换；但从总体功能而言，将视图翻转器视为幻灯片播放器是再恰当不过了。图 4-33 是视图翻转器应用程序（浏览不同布局的图片）的实机界面，该界面中通过选项菜单项来实现。

这里的翻转类似于依次翻看扑克牌（手动），或自动浏览幻灯片（自动），无论如何，同一时刻只能看到面上的一个子视图。

(1) 主界面布局定义

代码 4-39 是视图翻转器示例程序的主界面布局定义，其主要组件为视图翻转器。

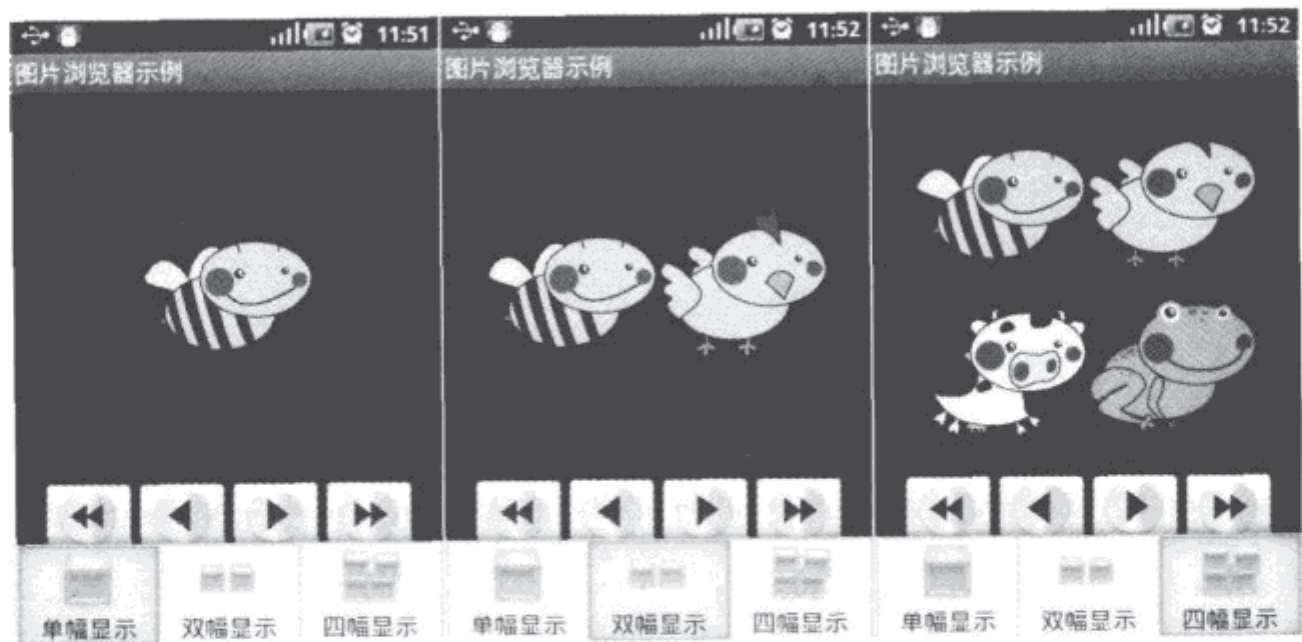


图 4-33 视图翻转器示例程序的实机界面

代码 4-39 视图翻转器示例程序的主界面布局定义

文件名: main.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical" android:layout_width="fill_parent"
4      android:layout_height="fill_parent">
5      <ViewFlipper android:id="@+id/vfview" android:layout_width="fill_parent"
6          android:layout_height="300px" />
7      <LinearLayout android:gravity="center"
8          ..... //与视图切换器相同
9  </LinearLayout>
```

(2) 应用程序 Activity 组件框架

代码 4-40 是视图翻转器示例程序的 Activity 组件的框架定义。

代码 4-40 视图翻转器示例程序的 Activity 组件的框架定义

文件名: ImageViewer2Act.java

```
1  public class ImageViewer2Act extends Activity implements OnClickListener {
2      //视图翻转器
3      private ViewFlipper mVfView = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //获取视图翻转器实例
10         this.mVfView = (ViewFlipper)findViewById(R.id.vfview);
11         //添加子视图
12         this.mVfView.addView(new ImagePanel(this,1,1),0);
13         this.mVfView.addView(new ImagePanel(this,2,1),1);
```




Android 平台开发之旅 第2版

```

14         this.mVfView.addView(new ImagePanel(this,2,2),2);
15         //获取按钮控件并设置单击侦听器
16         .....
17     }
18     @Override
19     public boolean onCreateOptionsMenu(Menu menu) {
20         this.getMenuInflater().inflate(R.menu.ops_menu, menu);
21         return super.onCreateOptionsMenu(menu);
22     }
23     @Override
24     public boolean onOptionsItemSelected(MenuItem item) {
25         switch(item.getItemId()) {
26             case R.id.mi_single: { this.mVfView.setDisplayedChild(0); break; }
27             case R.id.mi_dual: { this.mVfView.setDisplayedChild(1); break; }
28             case R.id.mi_four: { this.mVfView.setDisplayedChild(2); break; }
29         }
30         return super.onOptionsItemSelected(item);
31     }
32     @Override
33     public void onClick(View v) {
34         ImagePanel panel = (ImagePanel)this.mVfView.getCurrentView();
35         ..... //与视图切换器相同
36     }
37 };

```

在代码 4-40 中，视图翻转器添加了 3 个子视图（第 12~14 行）；翻转器使用 `setDisplayedChild` 方法直接显示指定子视图（第 26~28 行）。

（3）子视图的定义

视图翻转器示例程序中的翻转器子视图与视图切换器示例程序相同。

（4）播放子视图

除了手动切换子视图，视图翻转器主要用来播放子视图（自动切换）。图 4-34 所示为使用视图翻转器播放图片的实机界面，用户通过单击图来启动/暂停图片的播放。



图 4-34 视图翻转器播放图片的实机界面

对于这种全屏模式下的视图切换，其界面定义与使用选项菜单项有所不同。代码 4-41 是图 4-34 中示例程序的主界面布局定义，其根组件为视图翻转器。

代码 4-41 视图翻转器示例程序的主界面布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <ViewFlipper xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent"
5      android:id="@+id/vfview" android:flipInterval="1000"/>

```

代码 4-42 是图 4-34 中示例程序的 Activity 组件的框架定义。

代码 4-42 视图翻转器示例程序的 Activity 组件的框架定义

文件名: FlippingAct.java

```

1  public class FlippingAct extends Activity implements OnClickListener {
2      //视图翻转器
3      private ViewFlipper mFlipper = null;
4      private boolean mIsPlaying = false;
5
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.main);
10         //获取翻转器
11         mFlipper = (ViewFlipper)findViewById(R.id.vfview);
12         initFlipper();
13     }
14
15     private void initFlipper() { //初始化翻转器
16         for(int i = 0; i < IConfig.IMGS.length; ++i) {
17             ImageView iv = new ImageView(this);
18             iv.setImageResource(IConfig.IMGS[i]);
19             iv.setScaleType(ImageView.ScaleType.CENTER);
20             iv.setPadding(8, 8, 8, 8);
21             iv.setLayoutParams(new LayoutParams(
22                 LayoutParams.FILL_PARENT, LayoutParams.FILL_PARENT));
23             mFlipper.addView(iv);
24         }
25         //设置单击事件侦听器
26         mFlipper.setOnClickListener(this);
27     }
28
29     @Override
30     public void onClick(View v) {

```




```
31         if(mIsPlaying) { this.mFlipper.stopFlipping(); this.mIsPlaying = false;
32         } else { this.mFlipper.startFlipping(); this.mIsPlaying = true; }
33     }
34 };
```

在代码 4-42 中，视图适配器的子组件为图片视图（第 17 行），当其子组件初始化（添加）完毕后，可以使用翻转器的 `startFlipping` 或 `stopFlipping` 方法来启动/停止自动翻转（第 32 行和第 31 行）。

11. 滑动抽屉 (SlidingDrawer)

对于滑动抽屉界面，相信大多数读者不会陌生，Android 系统查看消息的界面就是使用的该界面效果：当用户触摸状态栏时，会弹出一个用于拖曳的按钮条；按住该按钮条并往下拖动即可打开查看状态的界面，其过程恰如打开抽屉。图 4-35 所示为滑动抽屉示例程序的实机界面。



图 4-35 滑动抽屉示例程序的实机界面

(1) 主界面布局定义

代码 4-43 是滑动抽屉示例程序的主界面布局定义，其根组件为滑动抽屉。

代码 4-43 滑动抽屉示例程序的主界面布局定义

文件名: `main.xml`

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <SlidingDrawer xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@+id/sdview"
4      android:layout_width="fill_parent" android:layout_height="fill_parent"
5      android:handle="@+id/tview" android:content="@+id/lview">
6      <TextView android:id="@+id/tview" android:text="请选择条目"
7          android:layout_width="fill_parent" android:layout_height="30px"
8          android:gravity="center" android:background="#111111"/>
9      <ListView android:id="@+id/lview"
10         android:layout_width="fill_parent" android:layout_height="fill_parent"/>
11 </SlidingDrawer>
```


在代码 4-43 中，滑动抽屉需要两个组件：手柄和内容视图（第 5 行），这两个组件可以自由定义：手柄主要用于定义抽屉的外观；内容视图定义抽屉打开后的内容。

提示：在滑动抽屉定义中，滑动抽屉需要引用手柄和内容视图，但这两个组件在滑动抽屉定义之后，所以不能按照普通的组件引用方式（@id/<组件 ID>），而是采用引用并创建的方式（@+id/<组件 ID>）。在这两个组件的定义中，其 ID 采用的是引用形式（第 6 行和第 9 行），因为在这之前该组件已经被创建了。

（2）应用程序 Activity 组件框架

代码 4-44 是滑动抽展示例程序的 Activity 组件的框架定义。

代码 4-44 滑动抽展示例程序的 Activity 组件的框架定义

文件名：SlidingDrawerAct.java

```
1 public class SlidingDrawerAct extends Activity implements OnItemClickListener {
2     //滑动抽屉视图
3     private SlidingDrawer mSdView = null;
4     //列表视图
5     private ListView mListView = null;
6
7     @Override
8     public void onCreate(Bundle savedInstanceState) {
9         super.onCreate(savedInstanceState);
10        setContentView(R.layout.main);
11        //获取滑动抽屉视图
12        mSdView = (SlidingDrawer)findViewById(R.id.sdview);
13        mListView = (ListView)findViewById(R.id.lview);
14        //初始化适配器
15        FooItemAdapter adapter = new FooItemAdapter(this);
16        mListView.setAdapter(adapter);
17        //设置选择侦听器
18        mListView.setOnItemClickListener(this);
19    }
20    @Override
21    public void onItemClick(AdapterView<?> parent, View view, int pos, long id) {
22        if(view == null) { return; }
23        TextView tv = (TextView)view.findViewById(R.id.title);
24        Toast.makeText(this, tv.getText().toString(), Toast.LENGTH_LONG).show();
25        //关闭抽屉
26        mSdView.close();
27    }
28 };
```

从代码 4-44 可知，滑动抽屉不对其子组件进行管理，而只对界面进行控制，如关闭抽屉（第 26 行）。除此之外，还有打开抽屉，锁定抽屉等。

（3）滑动抽屉的事件响应



表 4-9 是与滑动抽屉相关的、较为重要的事件侦听器。

表 4-9 滑动抽屉的事件侦听器

属 性	说 明
OnDrawerCloseListener	用于侦听抽屉的关闭
OnDrawerOpenListener	用于侦听抽屉的打开
OnDrawerScrollListener	用于侦听抽屉的滚动

使用滑动抽屉设置侦听器的方法即可设置表 4-9 中对应侦听器的绑定。

12. 表面视图（Surface View）

表面视图是底层渲染组件，代表了设备屏幕。通过表面视图，开发者可以更加灵活自由地对屏幕界面进行绘制，甚至可以引入 OpenGL 平台的绘制接口。在第 5 章会对该视图进行介绍。

13. 地图视图（Map View）

相信很多读者对地图应用相当熟悉，因为它给大家的生活带来了很多的便利。对于出差到陌生城市的用户，他们可以通过网页地图查询到住地附近的便利设施（如超市、银行、饭店、旅游景点等）。为了集成地图应用，Android 平台提供了地图视图，方便开发者将地图应用集成到应用程序中。在第 14 章会对该视图进行介绍。

14. 网页视图（Web View）

为了集成网页内容，Android 平台提供了强大并且易用的网页视图。通过网页视图，开发者可以轻松地将网页内容集成到应用程序中。在第 8 章会对该视图进行介绍。

4.4.3 定制视图

所谓定制视图，即不使用 Android 平台所预定义的视图，而是通过继承视图类（View）所自定义的视图。图 4-36 所示为在定制视图中绘制文本和图片的实机界面。

(1) 主界面布局定义

代码 4-45 是定制视图示例程序的主界面布局定义，其根组件为定制视图。

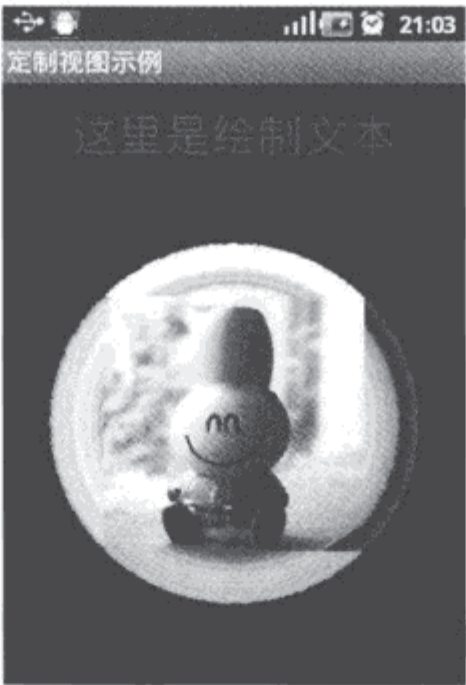


图 4-36 定制视图的实机界面

代码 4-45 定制视图示例程序的主界面布局定义

文件名: main2.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <foolstudio.demo.view.customview.FooCustomView
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     android:orientation="vertical"
5     android:layout_width="fill_parent" android:layout_height="fill_parent"/>
```

在代码 4-45 中，需要注意的是，定制视图的标签必须为类全名（包括路径），其目的是

为了保证资源解析器能够定位到该类，并实例化该类定义。

(2) 自定义视图

代码 4-46 是定制视图的 Activity 组件的框架定义，该视图直接继承于视图类。

代码 4-46 定制视图示例程序的 Activity 组件的框架定义

文件名: FooCustomView.java

```
1 public class FooCustomView extends View {
2     //屏幕宽高
3     private final int mWidth;
4     private final int mHeight;
5     private Bitmap mBkg = null;
6
7     public FooCustomView(Context ctx, AttributeSet attrs) { super(ctx, attrs);
8         final String sname = Context.WINDOW_SERVICE;
9         WindowManager mgr = (WindowManager)ctx.getSystemService(sname);
10        mWidth = mgr.getDefaultDisplay().getWidth();
11        mHeight = mgr.getDefaultDisplay().getHeight();
12        //载入背景图片
13        mBkg = BitmapFactory.decodeResource(ctx.getResources(), R.drawable.hulu);
14    }
15
16    @Override
17    public void draw(Canvas c) { //视图绘制回调
18        drawText(c);    //绘制文本
19        drawImage(c);   //绘制图片
20    }
21    .....
22 };
```

在代码 4-46 中，定制视图的绘制在重载方法 `draw` 中进行（第 17 行）。其中，需要注意的是，为了使定制视图类能用 XML 代码进行定义，该视图类必须拥有一个以属性集（`AttributeSet`）为参数的构造方法（第 7 行），否则在运行时会抛出以下异常。

`android.view.InflateException:`

Binary XML file line #6: Error inflating class foolstudio.demo.view.FoolCustomView

`java.lang.NoSuchMethodException: FoolCustomView(Context,AttributeSet)`

提示：View 类有 3 个构造方法：`View(Context)`、`View(Context, AttributeSet)`和 `View(Context, AttributeSet, int)`。其中，属性集（`AttributeSet`）所对应的内容就是 XML 属性标签。

4.5 小部件（Widgets）

从本质而言，小部件和视图组件的区分并不明显，都是显示组件。这里的小部件是指在



界面中不作为组件容器，功能比较独立的可视组件，如按钮组件、图片视图等。小部件类几乎都在小部件包（`android.widget`）中定义。

4.5.1 小部件的使用模式

1. 小部件的定义

与视图组件一样，小部件既可以使用 XML 代码也可以使用 Java 代码定义。使用 XML 代码定义的场合，小部件一般都是作为布局或者视图组件的子组件。

2. 引用小部件

对于小部件的使用，在此主要介绍对 XML 代码所定义的小部件的引用。由于小部件都是作为子组件进行定义，所以只需通过其资源 ID 来获取对象实例即可。

3. 用户事件响应

从设计的角度来说，小部件是用于与用户进行交互的，所以其对事件的响应最全面的，如按钮单击事件、文本框内容改变事件、组合框点选事件等。

小部件要响应指定类型的事件，必须先设置指定类型的侦听器，在侦听器的事件回调方法中对事件进行处理。

4.5.2 常用小部件

表 4-10 是 Android 平台所定义的常用小部件的说明。

表 4-10 Android 平台常用小部件的说明

分 类	小 部 件	说 明
文本	文本视图 (TextView)	用于显示文本内容
	文本编辑器 (EditText)	用于编辑文本内容
按钮	文字按钮 (Button)	用于单击触发
	图片按钮 (ImageButton)	用于单击触发
	复选框 (CheckBox)	用于勾选/取消
	单选按钮 (RadioButton)	用于互斥选择
	单选按钮组 (RadioGroup)	包含多个单选按钮
	切换按钮 (ToggleButton)	用于状态切换
	缩放控制按钮 (ZoomControls)	将缩、放按钮进行集成
图片	图片视图 (ImageView)	显示图片
显示条	进度条 (ProgressBar)	用于显示进度
	滑动条 (SeekBar)	用于显示或调整进度
选取器	日期选择器 (DatePicker)	用于选择日期
	时间选择器 (TimePicker)	用于选择时间
	微调控制器 (Spinner)	用于选择条目

囿于篇幅，在此不对小部件的使用进行过多介绍。实际上，在本书的开发案例中，基本涵盖了对大多数常用小部件的使用。

4.6 界面框架

这里所谓的界面框架是指 Android 平台为实现某些界面效果而定义的框架，通过这些框架可以方便地实现界面效果，如菜单、对话框、动画效果等。

4.6.1 菜单（Menu）和操作栏（ActionBar）

与桌面平台相同，Android 平台也定义了两种形式的菜单：选项菜单和上下文菜单。菜单包含多个菜单项，菜单项用于功能的触发。

1. 选项菜单

图 4-37 是选项菜单的实机界面（Android 2.2），图中的选项菜单包含两个菜单项，且其中一个用于功能切换（“自动滚屏”与“停止滚屏”的切换）。



图 4-37 选项菜单的实机界面（Android 2.2）

在 Android 3 平台中，对菜单的展现形式进行了增强，菜单项可以显示在操作栏中，如图 4-38 所示。



图 4-38 选项菜单的实机界面（Android 3.x）

(1) 选项菜单的定义

代码 4-47 是图 4-37 中选项菜单资源的定义，有关选项菜单资源的定义语法参考附录。

代码 4-47 选项菜单资源的定义（Android 2.x）

文件名：ops_menu.xml

```
1 <menu xmlns:android="http://schemas.android.com/apk/res/android">
2     <item android:id="@+id/mi_auto"
3           android:title="自动滚屏" android:icon="@drawable/auto"/>
4     <item android:id="@+id/mi_jump"
5           android:title="跳转" android:icon="@drawable/jump"/>
6 </menu>
```

代码 4-48 是图 4-38 中选项菜单资源的定义。

代码 4-48 选项菜单资源的定义（Android 3.x）

文件名：ops_menu.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
```



```
2 <menu xmlns:android="http://schemas.android.com/apk/res/android">
3     <item android:id="@+id/mi_add_mark" android:icon="@drawable/mark"
4         android:title="添加中心标注" android:showAsAction="ifRoom|withText" />
5     <item android:id="@+id/mi_get_loc" android:icon="@drawable/location"
6         android:title="获取中心位置" android:showAsAction="ifRoom|withText" />
7     <item android:id="@+id/mi_mode" android:icon="@drawable/satellite"
8         android:title="@string/satellite" android:showAsAction="ifRoom|withText"/>
9     <item android:id="@+id/mi_capture" android:icon="@drawable/capture"
10        android:title="截屏"/>
11 </menu>
```

在代码 4-48 中，`android:showAsAction` 是 Android 3 平台的新特性，用于设置菜单项的显示形式。表 4-11 是菜单项在操作栏中的显示风格的说明。

表 4-11 菜单项在操作栏中的显示风格的说明

取值类型	说 明
never	不显示在操作栏
ifRoom	有空间就显示
always	总是显示在操作栏
withText	显示文本

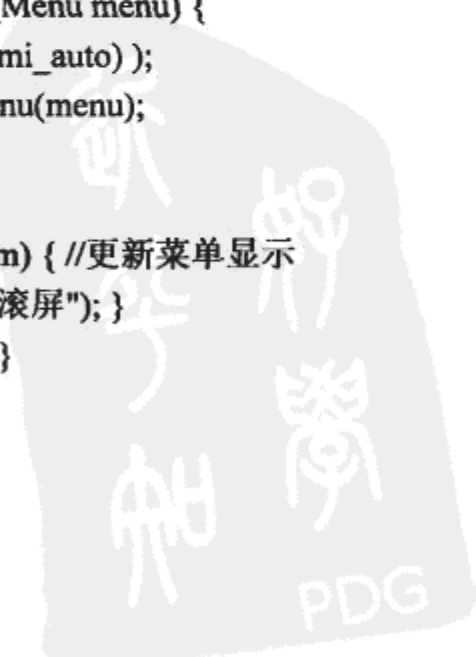
(2) 选项菜单使用框架

代码 4-49 是使用选项菜单资源的框架代码，其中包括选项菜单的创建和预处理以及选择菜单项的处理。

代码 4-49 选项菜单使用框架

文件名: `ScrollViewAct.java`

```
1 @Override
2 public boolean onCreateOptionsMenu(Menu menu) {
3     //填充选项菜单资源
4     this.getMenuInflater().inflate(R.menu.ops_menu, menu);
5     return (super.onCreateOptionsMenu(menu));
6 }
7 @Override
8 public boolean onPrepareOptionsMenu(Menu menu) {
9     updateMenu(menu.findItem(R.id.mi_auto));
10    return super.onPrepareOptionsMenu(menu);
11 }
12
13 private void updateMenu(MenuItem item) { //更新菜单显示
14     if(mIsAuto) { item.setTitle("停止滚屏"); }
15     else { item.setTitle("自动滚屏"); }
16 }
17
18 @Override
```




```
19 public boolean onOptionsItemSelected(MenuItem item) {
20     switch(item.getItemId() ) {
21         case R.id.mi_auto: { doAuto();  updateMenu(item); break; }
22         case R.id.mi_jump: { this.showDialog(IConfig.JUMP_DLG); break; }
23     }
24     return (super.onOptionsItemSelected(item) );
25 }
```

在代码 4-49 中，选项菜单的使用都在 Activity 组件提供的框架中完成。表 4-12 是与选项菜单相关的几个重要回调方法的说明。

表 4-12 选项菜单相关的回调方法的说明

回调方法	说 明
onCreateOptionsMenu(Menu)	用于创建选项菜单
onPrepareOptionsMenu(Menu)	用于选项菜单预处理
onOptionsItemSelected(MenuItem)	菜单项被选取后回调

在创建回调方法中，使用菜单填充器对菜单资源进行填充，生成菜单实例（第 4 行）；在预处理回调方法中，可以依据变量状态对菜单项进行修正（第 14 行或第 15 行），实现菜单状态的切换；在菜单项被选回调方法中，通过菜单项的 ID 即可获知由哪个菜单项触发的（第 20 行），从而调用相关的方法（第 21 行或第 22 行）。

对于菜单项的触发，还有一种简洁的方式：给菜单项绑定意向，如代码 4-50 所示。

代码 4-50 菜单项的意向绑定

文件名：JournalBookAct.java

```
1  @Override
2  public boolean onCreateOptionsMenu(Menu menu) { //创建选项菜单的回调方法
3      //填充菜单
4      this.getMenuInflater().inflate(R.menu.ops_menu, menu);
5      //设置菜单项意向
6      menu.findItem(R.id.mi_cfg).setIntent(new Intent(this,DBConfigAct.class));
7      menu.findItem(R.id.mi_add).setIntent(new Intent(this,AppendRecAct.class));
8      menu.findItem(R.id.mi_view).setIntent(new Intent(this,ReviewRecAct.class));
9      menu.findItem(R.id.mi_look).setIntent(new Intent(this,LookupRecAct.class));
10     //注意：必须要调用超类的方法，否则无法实现意图回调
11     return (super.onCreateOptionsMenu(menu) );
12 }
```

在代码 4-50 中，在选项菜单创建回调方法中，给每一个菜单项各绑定一个意向（第 6~9 行）；当选取菜单项后，无需在菜单项被选回调方法中进行处理，而是直接调用意向对象所包含的组件（代码中都是 Activity 组件）。

2. 上下文菜单

在桌面平台中，上下文菜单即右键菜单，其一般绑定到指定的可视组件；在手机设备中，长按屏幕（触摸屏）或按压指定的功能按钮也会触发上下文菜单。图 4-39 是上下文菜



单的实机界面，该菜单在长按网页视图组件时弹出，用于下载网页中的链接资源。

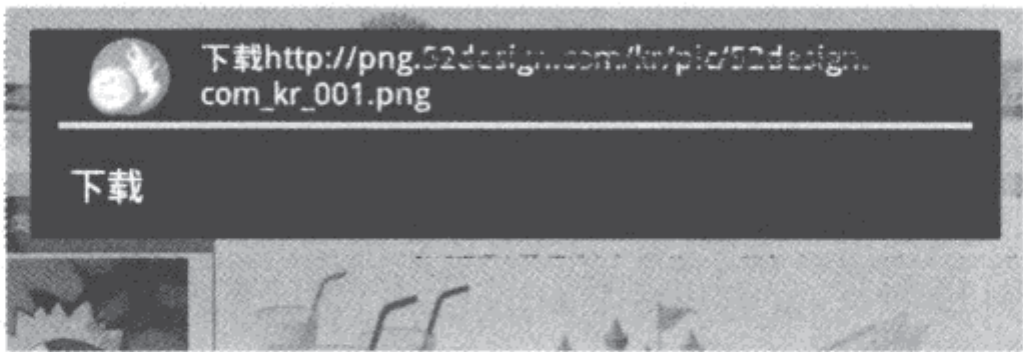


图 4-39 上下文菜单的实机界面

(1) 上下文菜单的定义

上下文菜单和选项菜单的定义方式相同。代码 4-51 是图 4-39 所示的上下文菜单的定义。

代码 4-51 上下文菜单的定义

文件名: context_menu.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <menu xmlns:android="http://schemas.android.com/apk/res/android">
3     <item android:id="@+id/mi_download" android:title="下载"/>
4 </menu>
```

(2) 上下文菜单使用框架

代码 4-52 是使用上下文菜单资源的框架代码，其中包括注册上下文菜单、创建上下文菜单以及上下文菜单项被选处理。

代码 4-52 上下文菜单使用框架

文件名: FooBrowserAct.java

```
1 @Override
2 @Override
3 public void onCreate(Bundle savedInstanceState) {
4     super.onCreate(savedInstanceState);
5     setContentView(R.layout.main);
6     .....
7     //设置网页视图的上下文菜单
8     this.registerForContextMenu(mWebView);
9     .....
10 }
11 @Override
12 public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo info) {
13     super.onCreateContextMenu(menu, v, info);
14     this.getMenuInflater().inflate(R.menu.context_menu, menu);
15     //设置菜单抬头按钮
16     menu.setHeaderIcon(R.drawable.download);
```



```
17      .....
18      //设置菜单标题
19      menu.setHeaderTitle("下载" + url);
20  }
21  @Override
22  public boolean onOptionsItemSelected(MenuItem item) {
23      switch(item.getItemId() ) { case R.id.mi_download: { ..... } }
24      return super.onOptionsItemSelected(item);
25  }
```

在代码 4-52 中，上下文菜单的使用都在 Activity 组件提供的框架中完成。表 4-13 是与上下文菜单相关的几个重要回调方法的说明。

表 4-13 上下文菜单相关的回调方法的说明

回 调 方 法	说 明
onCreateContextMenu(ContextMenu, View,ContextMenuInfo)	用于创建上下文菜单
onOptionsItemSelected(MenuItem)	菜单项被选取后回调

在 Activity 创建回调方法中，Activity 组件给网页视图注册上下文菜单（第 8 行）；在上下文菜单创建回调方法中，使用菜单填充器对菜单资源进行填充，生成菜单实例（第 14 行）；在菜单项被选回调方法中，通过菜单项的 ID 即可获知由哪个菜单项触发的，从而调用相关的方法（第 23 行）。

4.6.2 对话框（Dialog）

与菜单界面一样，对话框也是应用程序常用的一种界面方式。Android 平台在应用包（android.app）中定义了对对话框类及其子类。表 4-14 是对对话框类型的说明。

表 4-14 对话框类型的说明

对 话 框	说 明
Dialog	对话框基类
AlertDialog	提示对话框
ProgressDialog	进度对话框

1. 对话框的使用模式

Activity 框架为对话框的使用提供了框架，包括初始化对话框和显示对话框。代码 4-53 是使用对话框的实例代码。

代码 4-53 对话框的使用

文件名: DialogsAct.java

```
1  @Override
2  protected Dialog onCreateDialog(int id) { //初始化对话框
3      switch(id) {
4          case IConfig.DLG_ALERT: { return (initAlertDlg() ); }
```



```
5         case IConfig.DLG_CUSTOM: { return (initCustomDlg() ); }
6         default: { return (null); }
7     }
8 }
9 @Override
10 public void onClick(View v) {
11     switch(v.getId() ) {
12         case R.id.btnAlert: { this.showDialog(IConfig.DLG_ALERT); break; }
13         case R.id.btnCustom: { this.showDialog(IConfig.DLG_CUSTOM); break; }
14         case R.id.btnProgress: { initProgressDlg().show(); break; }
15     }
16 }
```

在代码 4-53 中，Activity 组件提供了 onCreateDialog 回调方法用于初始化对话框（第 4 行和第 5 行）；当单击按钮组件时，使用 showDialog 方法即可显示对话框（第 12~14 行）。其中，对话框的 ID 由用户定义，Activity 组件以此区分对话框。

图 4-40 是对话框显示与创建方法的关联示意图。

2. 对话框的初始化

(1) 提示对话框（Alert Dialog）

提示对话框用于提示用户给出答复。图 4-41 所示为提示对话框的实机界面。

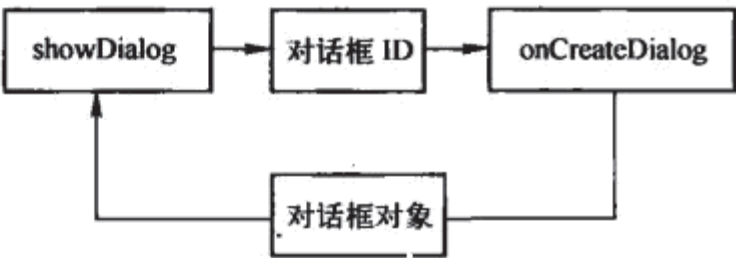


图 4-40 对话框显示与创建方法的关联示意图

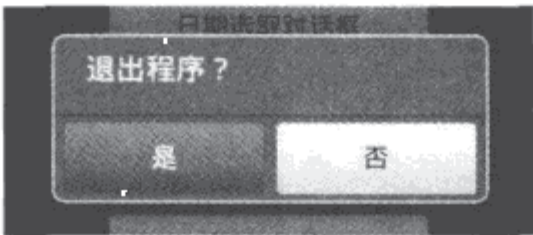


图 4-41 提示对话框的实机界面

代码 4-54 是初始化提示对话框的主要代码。

代码 4-54 初始化提示对话框

文件名: DialogsAct.java

```
1 private Dialog initAlertDlg() { //初始化提示对话框
2     AlertDialog.Builder builder = new AlertDialog.Builder(this);
3     //提示信息
4     builder.setMessage("退出程序? ");
5     builder.setCancelable(false);
6     //是按钮
7     builder.setPositiveButton("是", new DialogInterface.OnClickListener() {
8         @Override
9         public void onClick(DialogInterface dialog, int which) {
10             DialogsAct.this.finish();
11         }
12     });
```



```
13      //否按钮
14      builder.setNegativeButton("否", new DialogInterface.OnClickListener() {
15          @Override
16          public void onClick(DialogInterface dialog, int which) { dialog.cancel(); }
17      });
18      return (builder.create());
19  }
```

在代码 4-54 中，提示对话框使用其构建类（AlertDialog.Builder）进行构建，包括设置提示消息以及按钮的单击响应。

(2) 进度对话框（Progress Dialog）

进度对话框用于显示任务的执行进度。图 4-42 所示为进度对话框的实机界面。



图 4-42 进度对话框的实机界面

代码 4-55 是初始化进度对话框的主要代码。

代码 4-55 初始化进度对话框

文件名: DialogsAct.java

```
1  private Dialog initProgressDlg() { //初始化进度对话框
2      mProgressDlg = new ProgressDialog(this);
3      mProgressDlg.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
4      mProgressDlg.setMessage("正在加载...");
5      mProgressThd = new ProgressThread(mHandler);
6      mProgressThd.start();
7      return (mProgressDlg);
8  }
```

在代码 4-55 中，进度对话框通过启动进度条线程来执行模拟器任务（第 6 行），主 Activity 将主线程消息队列处理器传递给该线程（第 5 行），用于接收线程消息。

代码 4-56 是代码 4-55 中参考的执行任务的线程定义。

代码 4-56 执行任务的线程定义

文件名: DialogsAct.java

```
1  class ProgressThread extends Thread {
2      private Handler mHandler = null;
3      private int mState = IConfig.STATE_RUNNING;;
4      private int mCurVal = 0;
5
6      ProgressThread(Handler handler) { mHandler = handler; }
7
8      public void run() {
9          while (mState == IConfig.STATE_RUNNING) {
10             try { Thread.sleep(100); }
11             catch (InterruptedException e) { e.printStackTrace(); }
```



Android 平台开发之旅 第2版

```

12          //发送消息
13          FooSysUtil.getInstance().postMsg(mHandler, IConfig.EXTRA, mCurVal);
14          mCurVal++;
15      }
16  }
17
18  //改变状态
19  synchronized public void setState(int state) { mState = state; }
20  };

```

在代码 4-56 中，该线程模拟任务执行，并将进度信息通过主线程消息队列处理器发送给主线程（第 13 行）。代码 4-57 是代码 4-55 中参考的主线程消息队列处理器的定义。

代码 4-57 主线程消息队列处理器的定义

文件名: DialogsAct.java

```

1  private final Handler mHandler = new Handler() {
2      public void handleMessage(Message msg) {
3          //获取进度信息
4          int progress = msg.getData().getInt(IConfig.EXTRA);
5          mProgressDialog.setProgress(progress);
6          if (progress >= IConfig.MAX_VAL){
7              mProgressDialog.dismiss();
8              mProgressThd.setState(IConfig.STATE_DONE);
9          }
10     }
11 };

```

在代码 4-57 中，在主线程消息队列处理器的消息处理回调方法中，接收线程消息并从中提取进度信息（第 4 行），以此作为设置进度对话框进度和结束进程的依据（第 5 行和第 6 行）。

(3) 定制对话框 (Custom Dialog)

定制对话框就是由用户自己定义对话框界面。图 4-43 所示为定制对话框的实机界面。

代码 4-58 是初始化定制对话框的主要代码。

代码 4-58 初始化定制对话框

文件名: DialogsAct.java

```

1  private Dialog initCustomDlg() {
2      Dialog dlg = new Dialog(this);
3      //设置定制对话框内容视图和标题
4      dlg setContentView(R.layout.custom_dlg);
5      dlg.setTitle("定制对话框");
6      //返回对话框对象

```



图 4-43 定制对话框的实机界面


```
7         return (dlg);
8     }
```

在代码 4-58 中，该定制类使用对话框基类创建，且将其内容视图指定为某布局资源而实现定制（第 4 行）。代码 4-59 是该内容视图布局资源的定义。

代码 4-59 定制对话框内容视图布局资源的定义

文件名: custom_dlg.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="320sp" android:layout_height="120sp"
4      android:orientation="horizontal" android:padding="8px"
5      android:background="#336699">
6      <ImageView android:src="@drawable/cat" android:scaleType="matrix"
7          android:layout_width="wrap_content" android:layout_height="wrap_content"/>
8      <TextView  android:text="@string/about" android:paddingLeft="8px"
9          android:layout_width="fill_parent" android:layout_height="fill_parent"/>
10 </LinearLayout>
```

4.6.3 消息提示条（Toast）

之所以将 Toast 翻译为消息提示条，是因为其用于快速显示消息。图 4-44 所示为使用 Toast 显示列表视图中列表项的内容。相比对话框，消息提示条没有焦点，且其显示时长有限，显示后间隔一段时间会自动消失。

虽然消息提示条可以显示，且类似于定制对话框可以设置显示视图，但它和对话框一样，与视图没有直接关系，所以作者将其与对话框划分为界面使用模式。一般的，使用消息提示条的静态方法 `makeText` 来创建一个消息提示条实例，通过实例的 `show` 方法即可显示提示消息，其实例代码如下所示。



图 4-44 消息提示条的实机界面

```
Toast.makeText(this, "文件删除成功！", Toast.LENGTH_SHORT).show();
```

4.6.4 片段组件（Fragment）

1. 片段组件概述

片段是 Android 平台的新特性，其表示 Activity 组件的一部分动作或界面。一个 Activity 组件可以包含多个片段，Activity 通过片段管理器对其所包含的片段进行管理。读者可以将片段视为轻量级的 Activity，因为片段也需要与布局资源进行绑定，且用来与用户进行交互。图 4-45 是一个包含两个片段的界面，其中左边的一个片段用于展示列表，而右边的一个用于展示图片。

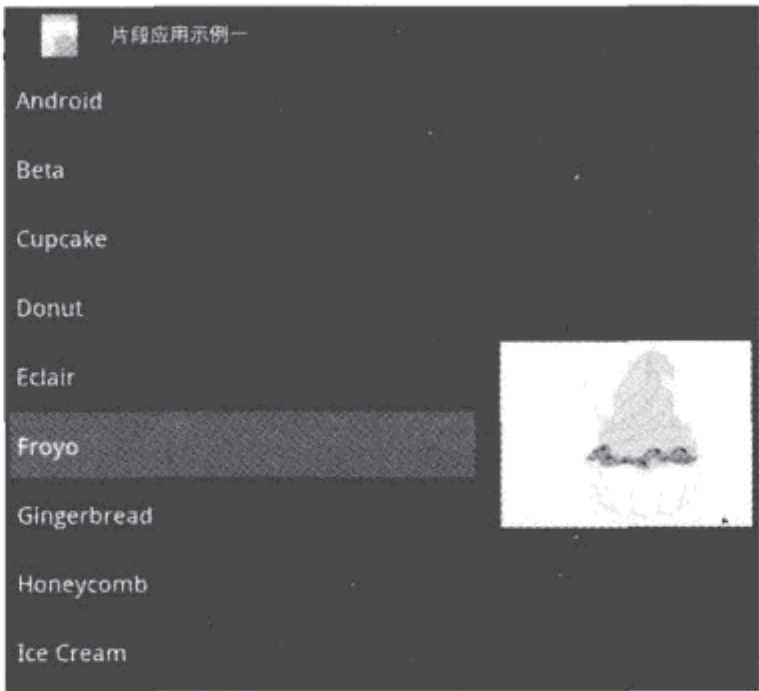


图 4-45 包含两个片段的实机界面

代码 4-60 是图 4-45 所示的应用程序的主 Activity 组件的框架定义。

代码 4-60 主 Activity 组件的框架定义

文件名: SimpleFmtAct.java

```
1 public class SimpleFmtAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6     }
7 };
```

读者对代码 4-60 可能会奇怪，因为其内容并没有特别之处，甚至只是基本的 Activity 组件的框架定义。而实际上，这正是片段的“魅力”之处，片段可以“分担”Activity 组件的工作，而且还可以像视图控件一样定义在布局资源中。代码 4-61 是主 Activity 组件所绑定的布局资源的定义。

代码 4-61 主 Activity 组件的布局资源的定义

文件名: main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="horizontal"
4     android:layout_width="fill_parent" android:layout_height="fill_parent">
5     <fragment android:name="foolstudio.demo.ui.simpltfmt.FooListFmt"
6         android:id="@+id/listfmt" android:layout_weight="1"
7         android:layout_width="0dp" android:layout_height="match_parent"/>
8     <fragment android:name="foolstudio.demo.ui.simpltfmt.DetailFmt"
9         android:id="@+id/detailfmt" android:layout_weight="2"
10        android:layout_width="0dp" android:layout_height="match_parent"/>
11 </LinearLayout>
```


至此应该水落石出，在代码 4-61 中，主 Activity 组件的布局资源包含了两个片段组件（第 5 行和第 8 行），这两个组件分担了主 Activity 组件与用户进行交互的任务，所以主 Activity 什么活儿都不用做，如图 4-46 所示。

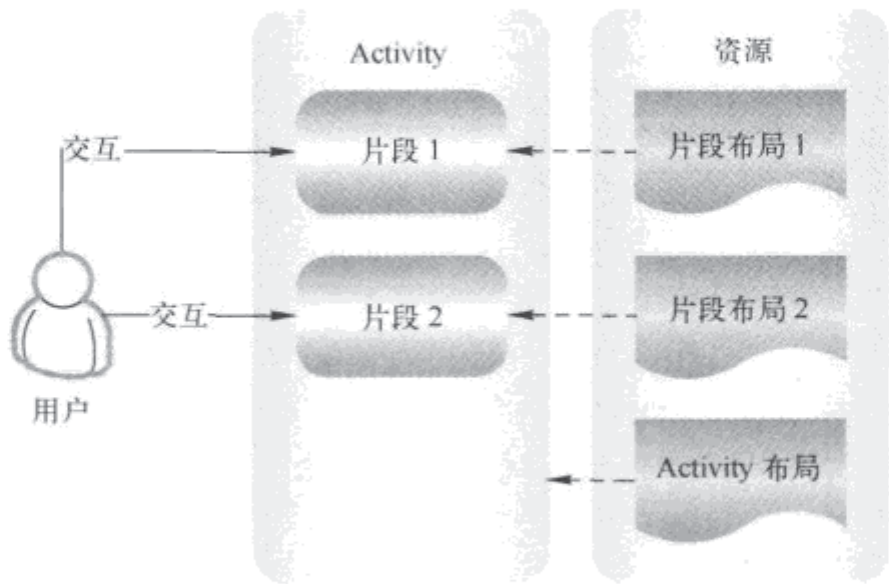


图 4-46 片段组件与用户进行交互的示意图

2. 片段组件框架

与 Activity 一样，Android 平台也为片段组件定义了框架。代码 4-62 是代码 4-61 中（第 5 行）所参考的列表片段组件的框架定义。

代码 4-62 列表片段组件的框架定义

文件名: FooListFmt.java

```
1 public class FooListFmt extends Fragment implements OnItemClickListener {
2     @Override //宿主 Activity 创建完毕后回调
3     public void onActivityCreated(Bundle savedInstanceState) {
4         super.onActivityCreated(savedInstanceState);
5         //获取列表视图
6         ListView v = (ListView)this.getActivity().findViewById(R.id.list);
7         if(v != null) { v.setOnItemClickListener(this); }
8     }
9     @Override //创建视图时回调
10    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle b) {
11        return inflater.inflate(R.layout.list_fmt, container, false);
12    }
13    //单击列表项回调
14    public void onItemClick(AdapterView<?> parent, View v, int pos, long id) {
15        //通过片段管理器获取详细片段
16        final FragmentManager mgr = this.getActivity().getFragmentManager();
17        DetailFmt detail = (DetailFmt)(mgr.findFragmentById(R.id.detailfmt));
18        //调用详细片段的方法
19        detail.updateImage(pos);
20    }
21    @Override //绑定时回调
22    public void onAttach(Activity activity) { super.onAttach(activity); }
23    };
```



在代码 4-62 中，列表片段组件通过继承父类片段（Fragment）进行定义（第 1 行），在重载方法中实现父类功能。表 4-15 是对片段组件的主要回调方法的说明。

表 4-15 片段组件回调方法及说明

回 调 方 法	说 明
onActivityCreated	当宿主 Activity 创建完毕后回调
onCreateView	当创建视图时回调
onAttach	当绑定到所宿主的 Activity 时回调

提示：代码 4-62 是在列表片段中调用详细片段，这种方式在片段数量较多的情况下会造成一定的混乱。通过绑定回调函数，在片段组件中可以获取宿主 Activity 接口，进而通过宿主 Activity 的方法可以实现片段的集中调用。其具体方式是：定义一个调用处理接口，主 Activity 实现该接口，在该接口方法中进行片段的调用处理；在片段组件中，定义一个该接口成员，在绑定回调方法中，将宿主 Activity 转换成该接口，在调用处理时，使用该接口进行调用（实际上调用的是宿主 Activity 的实现方法）。

代码 4-63 是片段调用接口定义的示例代码。

代码 4-63 片段调用接口的定义

文件名: ItemClickHandler.java

```
1 public interface ItemClickHandler {
2     public void doItemClick(int itemPos);
3 }
```

代码 4-64 是宿主 Activity（主 Activity）的框架定义。

代码 4-64 宿主 Activity 的框架定义

文件名: FmtDemo2Act.java

```
1 public class FmtDemo2Act extends Activity implements ItemClickHandler {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6     }
7
8     public void doItemClick(int itemPos) { //实现方法
9         //通过片段管理器获取详细片段
10        final FragmentManager mgr = this.getFragmentManager();
11        DetailFmt detail = (DetailFmt)(mgr.findFragmentById(R.id.detailfmt));
12        //调用详细片段的方法
13        detail.updateImage(itemPos);
14    }
15 };
```

代码 4-65 是调用方片段组件的框架定义。

代码 4-65 调用方片段组件的框架定义

文件名: FooListFmt.java

```

1  public class FooListFmt extends Fragment implements OnItemClickListener {
2      //列表项单击处理器
3      private ItemClickHandler mHandler = null;
4      .....
5      @Override //绑定到宿主 Activity 时回调
6      public void onAttach(Activity activity) {
7          super.onAttach(activity);
8          //获取宿主 Activity
9          try { mHandler = (ItemClickHandler)activity;
10             } catch(ClassCastException e) { e.printStackTrace(); }
11     }
12     @Override
13     public void onItemClick(AdapterView<?> parent, View v, int pos, long id) {
14         mHandler.doItemClick(pos);
15     }
16 };

```

3. 片段组件界面定义

通过代码 4-62 可以看出, 片段组件填充布局资源的时机与 Activity 组件有所不同: Activity 组件是在创建回调方法 (onCreate) 中, 而片段组件是在视图创建回调方法 (onCreateView) 中 (第 10 行)。代码 4-66 是片段组件所填充的布局资源的定义。

代码 4-66 片段组件所填充的布局资源的定义

文件名: list_fmt.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="match_parent" android:layout_height="match_parent"
4      android:orientation="horizontal">
5      <ListView android:id="@+id/list" android:entries="@array/android_codes"
6          android:layout_width="match_parent" android:layout_height="wrap_content"/>
7  </LinearLayout>

```

4. 片段组件的使用

(1) 引用界面组件

虽然在片段组件的框架中能够填充其界面布局, 但其无法直接引用界面组件, 而必须通过宿主 Activity 组件来进行资源管理或引用界面组件。使用片段实例的 getActivity 方法可获得宿主 Activity 组件的实例接口 (代码 4-62 中第 6 行); 再通过宿主 Activity 的方法可获得资源管理器或引用界面组件 (代码 4-62 中第 6 行的 findViewById 方法)。片段与宿主 Activity、资源管理器、片段管理等对象的关联如图 4-47 所示。

简而言之, 通过片段组件所绑定的宿主 Activity, 片段组件可以对应用程序中所定义的资源 (包括组件) 进行访问。

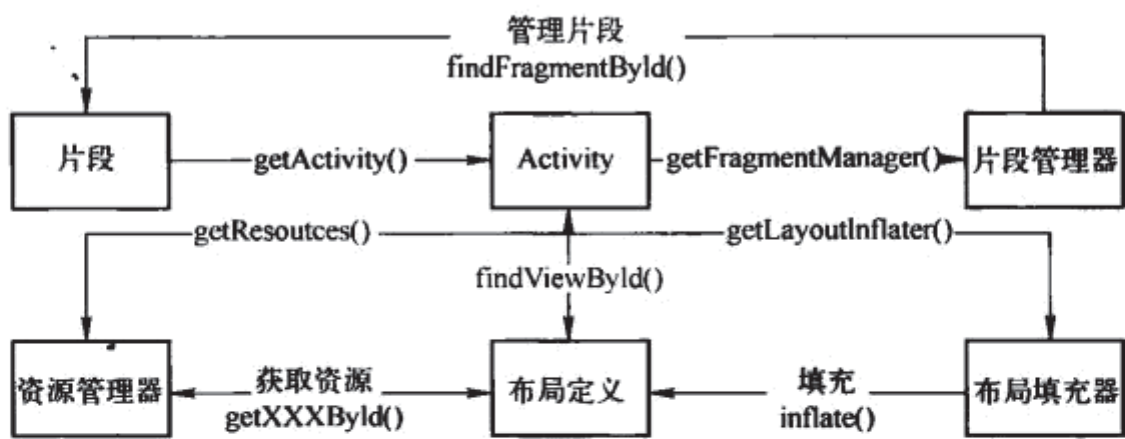


图 4-47 片段关联示意图

(2) 引用其他片段组件

在代码 4-62 中，列表片段通过宿主 Activity 的 `getFragmentManager` 方法获取片段管理器 (FragmentManager)，使用片段管理器的 `findFragmentById` 方法即可按照 ID 获取目标片段组件的实例。

简而言之，通过宿主 Activity 所关联的片段管理器，片段组件之间可以进行交互。

5. 常用片段组件

与列表视图和对话框框架一样，片段也定义了列表片段和对话框片段等组件，方便开发者使用。

(1) 列表片段 (ListFragment)

列表片段实际上就是内建了列表视图的片段组件，开发者甚至无需额外为该片段定义列表视图（这一点与列表 Activity 类似）。图 4-48 所示为使用列表片段组件的实机界面。

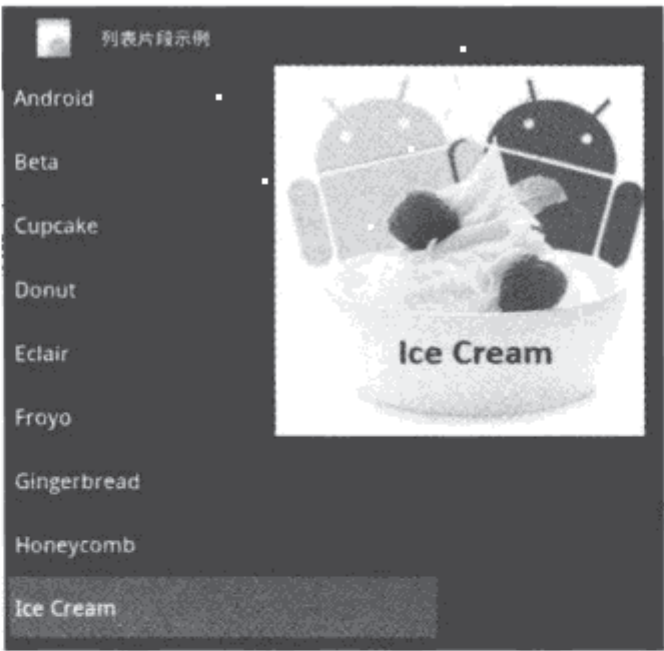


图 4-48 列表片段组件的实机界面

代码 4-67 是该列表片段组件的框架定义，其中主要内容是为内建的列表视图创建适配器，对其他片段组件的调用采用的是宿主 Activity 集中管理的方式。

代码 4-67 列表片段组件的框架定义

文件名: FooListFmt.java

```
1 public class FooListFmt extends ListFragment {
```



```

2      //列表项单击处理器
3      private ItemClickHandler mHandler = null;
4
5      @Override //创建片段时回调
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          //创建列表适配器
9          ArrayAdapter<String> adapter = new ArrayAdapter<String>(this.getActivity(),
10              android.R.layout.simple_list_item_1,
11              this.getResources().getStringArray(R.array.android_codes));
12          this.setAdapter(adapter);
13      }
14      @Override //绑定到宿主 Activity 时回调
15      public void onAttach(Activity activity) {
16          super.onAttach(activity);
17          //获取宿主 Activity
18          try { mHandler = (ItemClickHandler)activity;
19              } catch(ClassCastException e) { e.printStackTrace(); }
20      }
21      @Override
22      public void onItemClick(ListView l, View v, int pos, long id) {
23          mHandler.handleClick(pos);
24      }
25  };

```

(2) 对话框片段 (DialogFragment)

对话框片段以对话框的形式进行展示。图 4-49 是对话框片段的实机界面，当单击列表项后，将会弹出占满屏幕的对话框片段。

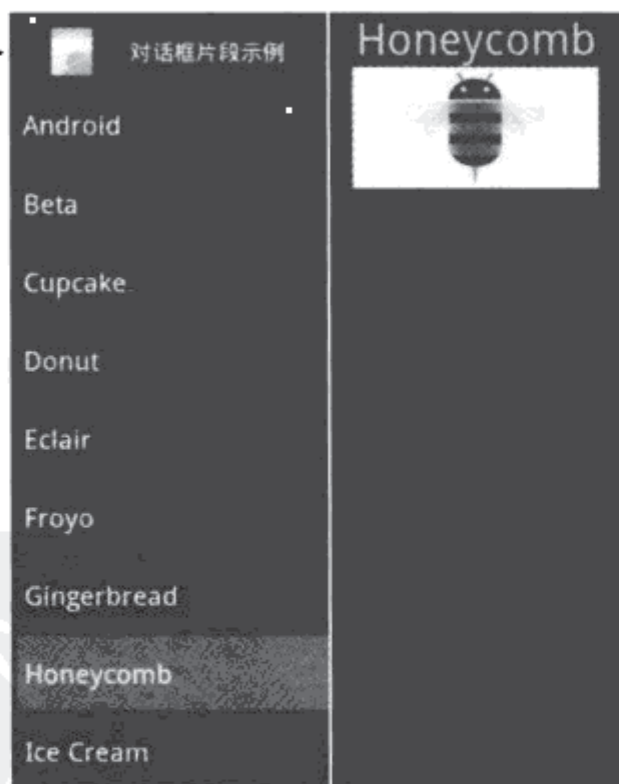


图 4-49 对话框片段的实机界面



代码 4-68 是对话框片段的定义，其中主要内容是设置对话框参数和显示风格。

代码 4-68 对话框片段的定义

文件名: DetailFmt.java

```

1  public class DetailFmt extends DialogFragment {
2      private int mPos = 0;
3      private TextView  mTitle = null;
4      private ImageView mImageView = null;
5
6      //创建对话框并设置参数
7      public static DetailFmt newInstance(int pos) {
8          DetailFmt f = new DetailFmt();
9          //设置代码参数
10         Bundle args = new Bundle();
11         args.putInt(IConfig.EXTRA, pos);
12         f.setArguments(args);
13         return f;
14     }
15
16     @Override
17     public void onCreate(Bundle savedInstanceState) {
18         super.onCreate(savedInstanceState);
19         //设置样式
20         this.setStyle(DialogFragment.STYLE_NO_TITLE, android.R.style.Theme);
21         mPos = this.getArguments().getInt(IConfig.EXTRA);
22     }
23     @Override
24     public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle b) {
25         View v = inflater.inflate(R.layout.detail_fmt, container, false);
26         mTitle = (TextView)v.findViewById(R.id.title);
27         mImageView = (ImageView)v.findViewById(R.id.img);
28         if(mImageView != null) { updateImage(mPos); }
29         return (v);
30     }
31
32     public void updateImage(int pos) {
33         final Resources res = this.getActivity().getResources();
34         mTitle.setText(res.getStringArray(R.array.android_codes)[pos]);
35         mImageView.setImageResource(IConfig.IDS[pos]);
36     }
37 };

```

代码 4-69 是宿主 Activity（主 Activity）组件的框架定义，其主要内容是通过片段管理器先移除之前的对话框，再创建对话框片段并显示。

代码 4-69 宿主 Activity 组件的框架定义

文件名: action_view.xml

```
1 public class DlgFmtAct extends Activity implements ItemClickHandler{
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6     }
7
8     public void handleItemClick(int itemPos) {
9         //通过片段管理器获取详细片段
10        final FragmentManager mgr = this.getFragmentManager();
11        DetailFmt detail = (DetailFmt)(mgr.findFragmentByTag(IConfig.TAG));
12        final FragmentTransaction ft = mgr.beginTransaction();
13        if(detail != null) { ft.remove(detail); }
14        ft.addToBackStack(null);
15        //创建对话框片段并显示
16        DetailFmt newFragment = DetailFmt.newInstance(itemPos);
17        newFragment.show(ft, IConfig.TAG);
18    }
19 };
```

4.6.5 拖放操作

拖放是 Android 平台的新特性。图 4-50 所示的界面是将列表项拖放到右侧的图片视图上进行显示的实机界面。与桌面平台一样，Android 平台为拖放操作定义了一套事件框架，通过事件回调方法处理拖放操作中的不同状态。

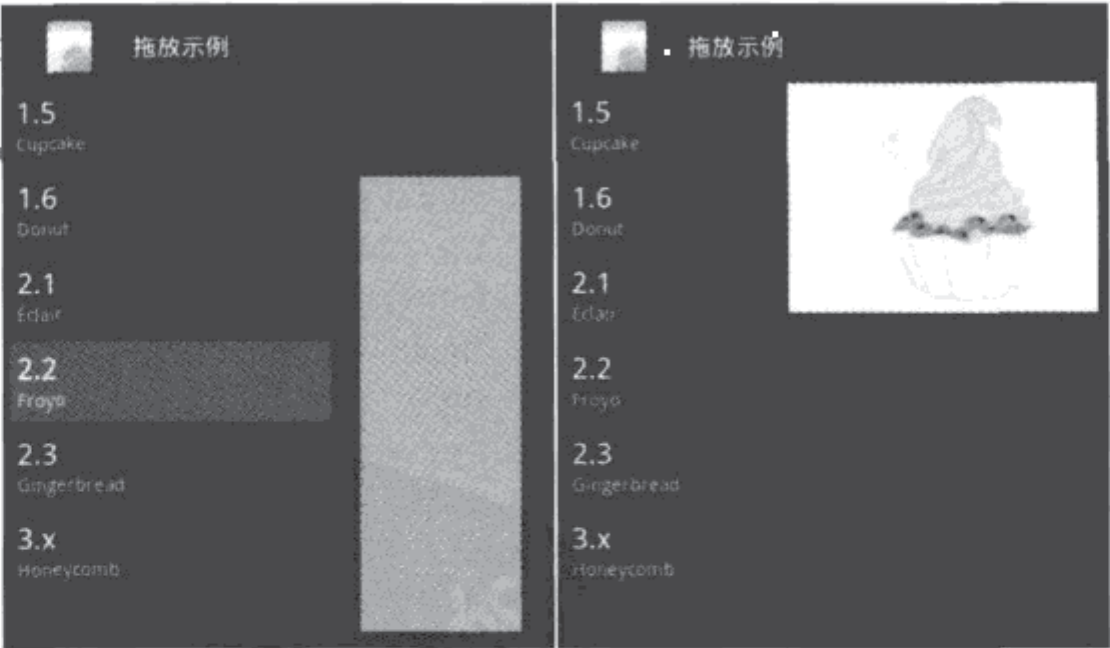


图 4-50 拖放的实机界面

1. 应用程序 Activity 框架

代码 4-70 是拖放示例程序的主 Activity 组件的框架定义。



代码 4-70 拖放示例程序的主 Activity 组件的框架定义

文件名: DragDropAct.java

```

1  public class DragDropAct extends Activity implements
2      OnDragListener, LoaderCallbacks<Cursor>, OnItemLongClickListener {
3      //列表视图
4      private ListView mListIndex = null;
5      private ImageView mImgDetail = null;
6
7      @Override
8      public void onCreate(Bundle savedInstanceState) {
9          super.onCreate(savedInstanceState);
10         setContentView(R.layout.main);
11         //获取列表视图实例
12         mListIndex = (ListView)findViewById(R.id.listLeft);
13         mImgDetail = (ImageView)findViewById(R.id.imgDetail);
14         //创建适配器
15         SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
16             android.R.layout.simple_list_item_2, null,
17             new String[] { IConfig.COL_VER, IConfig.COL_CODE },
18             new int[] { android.R.id.text1, android.R.id.text2 }, 0);
19         //设置适配器
20         mListIndex.setAdapter(adapter);
21         //初始化载入器
22         getLoaderManager().initLoader(0, null, this);
23         //设置拖曳侦听器
24         mImgDetail.setOnDragListener(this);
25         mListIndex.setOnItemLongClickListener(this);
26     }
27     @Override
28     public boolean onDrag(View v, DragEvent event) {
29         switch(event.getAction()) {
30             case DragEvent.ACTION_DRAG_STARTED: { return (true); }
31             case DragEvent.ACTION_DRAG_ENTERED: { return (true); }
32             case DragEvent.ACTION_DRAG_ENDED: { return (true); }
33             case DragEvent.ACTION_DRAG_EXITED: { return (true); }
34             case DragEvent.ACTION_DRAG_LOCATION: { return (true); }
35             case DragEvent.ACTION_DROP: {
36                 ClipData.Item item = event.getClipData().getItemAt(0);
37                 Intent dragIntent = item.getIntent();
38                 byte[] data = dragIntent.getByteArrayExtra(IConfig.EXTRA);
39                 Bitmap bmp = BitmapFactory.decodeByteArray(data, 0, data.length);
40                 this.mImgDetail.setImageBitmap(bmp);
41                 v.invalidate();
42                 return (true);
43             }

```



```

44         default: { break; }
45     }
46     return (false);
47 }
48 .....
49 };

```

在代码 4-70 中，为图片视图组件设置了拖曳侦听器（第 24 行），在其回调方法 `onDrag` 中可以获取拖放事件的状态并针对各种状态进行处理（第 28 行）；在代码 4-70 中只对放下操作进行了处理（第 35 行），当捕获到放下操作时，事件框架会获取剪贴数据项（第 36 行），并从中提取意向对象（第 37 行），从意向对象的扩展空间获取包含图片内容的字节数据（第 38 行），并转换成图片视图可以显示的位图（第 39 行）。

放下数据的解析与拖曳数据的组织是互逆的，读者应该可以猜出启动拖曳时的操作，即先获取列表项中图片数据，并通过意向对象进行封装，并以剪贴数据进行传递。此外，从代码 4-70 可知，列表视图的列表项记录的类型是记录游标，其数据内容都是通过数据加载器从数据库中加载。

提示：有关剪贴数据（`ClipData`）和剪贴项的使用可以参考系统剪贴板服务章节（第 15 章）；有关数据加载管理器的使用可以参考第 7 章。

2. 主界面布局定义

代码 4-71 是拖放示例程序的主界面布局的定义，其中包含一个列表视图和一个图片视图。

代码 4-71 拖放示例程序的主界面布局的定义

文件名：main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="fill_parent"
4      android:layout_height="fill_parent" android:orientation="horizontal">
5      <ListView android:id="@+id/listLeft"
6          android:layout_height="match_parent" android:layout_width="240sp" />
7      <ImageView android:id="@+id/imgDetail" android:scaleType="matrix"
8          android:layout_width="match_parent" android:layout_height="match_parent"/>
9  </LinearLayout>

```

3. 启动拖曳

拖曳操作是通过长按列表项进行触发。代码 4-72 是启动拖曳的关键代码。

代码 4-72 启动拖曳

文件名：DragDropAct.java

```

1  @Override
2  public boolean onItemLongClick(AdapterView<?> parent, View v, int pos, long id) {
3      //获取游标记录

```



Android 平台开发之旅 第2版

```

4      Cursor c = (Cursor)mListIndex.getAdapter().getItem(pos);
5      //创建意向
6      Intent intent = new Intent();
7      intent.putExtra(IConfig.EXTRA, c.getBlob(c.getColumnIndex(IConfig.COL_LOGO)));
8      //生成剪贴数据
9      ClipData.Item item = new ClipData.Item(intent);
10     ClipData dragData = new ClipData(IConfig.EXTRA,
11                                     new String[] { ClipDescription.MIMETYPE_TEXT_INTENT }, item);
12     View.DragShadowBuilder builder = new FooDragShadowBuilder(mListIndex);
13     //启动拖动
14     v.startDrag(dragData, builder, null, 0);
15
16     return false;
17 }

```

在代码 4-72 中，首先获取列表项对应的数据记录（记录游标，第 6 行）并从中获取 Logo 图片的数据内容，继而放入到意向对象的扩展空间中（第 7 行）；然后根据意向对象构造剪贴数据项（第 9 行），最后生成剪贴数据（第 10 行）。数据准备完毕，使用列表项视图的 `startDrag` 方法启动拖动（第 14 行）。对于该方法的调用，除了拖曳数据，还有一个重要的参数是拖曳的阴影构建器，其主要用于绘制拖曳的阴影效果（第 12 行）。

4. 拖曳阴影构建器

代码 4-73 是代码 4-72 所提到的拖曳阴影构建器的定义，其中主要内容是根据拖曳的位置信息绘制阴影效果。

代码 4-73 拖曳阴影构建器

文件名: FooDragShadowBuilder.java

```

1  public class FooDragShadowBuilder extends DragShadowBuilder {
2      //阴影
3      private static Drawable mShadow = null;
4
5      public FooDragShadowBuilder(View v) { super(v);
6          mShadow = new ColorDrawable(Color.LTGRAY);
7      }
8
9      @Override
10     public void onDrawShadow(Canvas c) { mShadow.draw(c); }
11     @Override
12     public void onProvideShadowMetrics(Point size, Point touch) {
13         int width = this.getView().getWidth()/2;
14         int height = this.getView().getHeight()/2;
15         mShadow.setBounds(0,0,width,height);
16         size.set(width, height);
17         touch.set(width/2, height/2);
18     }
19 };

```


4.6.6 动画效果

动画效果是增强手机应用程序的用户体验度的常用方式之一。Android 平台也为应用程序提供了多种动画效果的方式和样式。图 4-51 所示为两个图片视图，各自采用不同的动画方式进行动态效果的展现。

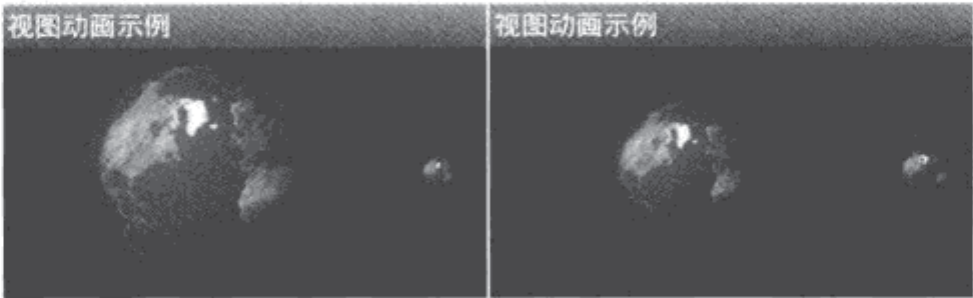


图 4-51 视图动画的实机界面

1. 视图动画

(1) 应用程序 Activity 框架

代码 4-74 是拖放示例程序的主 Activity 组件的框架定义。

代码 4-74 动画示例程序的主 Activity 组件的框架定义

文件名: ViewAnimDemoAct.java

```
1 public class ViewAnimDemoAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6     }
7     @Override
8     protected void onStart() { super.onStart();
9         ImageView iv1 = (ImageView) findViewById(R.id.img1);
10        ImageView iv2 = (ImageView) findViewById(R.id.img2);
11        Animation anim1 = AnimationUtils.loadAnimation(this, R.anim.foo_anim);
12        Animation anim2 = AnimationUtils.loadAnimation(this, R.anim.foo_anim2);
13        iv1.startAnimation(anim1);
14        iv2.startAnimation(anim2);
15    }
16 };
```

在代码 4-74 中，比较重要的是加载了两个动画资源并生成两个动画实例（第 11 行和第 12 行），然后通过图片视图的 startAnimation 方法即可按照动画实例所定义的动画方式启动动画效果（第 13 和第 14 行）。由此可见，Android 平台已经提供了一套完整的动画引擎，开发者只需定义动画资源即可，由此极大简化了视图动画效果的开发工作。

(2) 动画资源定义

代码 4-75 是代码 4-74 中引用的动画资源的定义，其中只包含缩放动画模式。



代码 4-75 动画资源的定义

文件名: foo_anim.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <set xmlns:android="http://schemas.android.com/apk/res/android"
3      android:interpolator="@android:anim/decelerate_interpolator">
4      <scale android:fromXScale="1.0" android:toXScale="0.0"
5          android:fromYScale="1.0" android:toYScale="0.0"
6          android:pivotX="50%" android:pivotY="50%"
7          android:startOffset="700" android:duration="400"
8          android:fillBefore="false"
9          android:repeatMode="reverse" android:repeatCount="-1"/>
10 </set>
```

2. 属性动画

Android 平台提供了一种新的动画方式：属性动画，即通过改变组织的属性值所产生的动画效果。表 4-16 是 Android 平台所支持的常用动画类型的说明。

表 4-16 常用动画类型的说明

类 型	说 明
透明度 (alpha)	渐变效果
缩放比例 (scale)	缩放效果
转变 (translate)	移动效果
旋转 (rotate)	旋转效果

以上动画类型还可以进行组合，形成动画集合（组合动画效果）。

(1) 单一属性值动画

图 4-52 所示为透明度属性的动画效果，其透明度属性值从不透明渐变到透明。

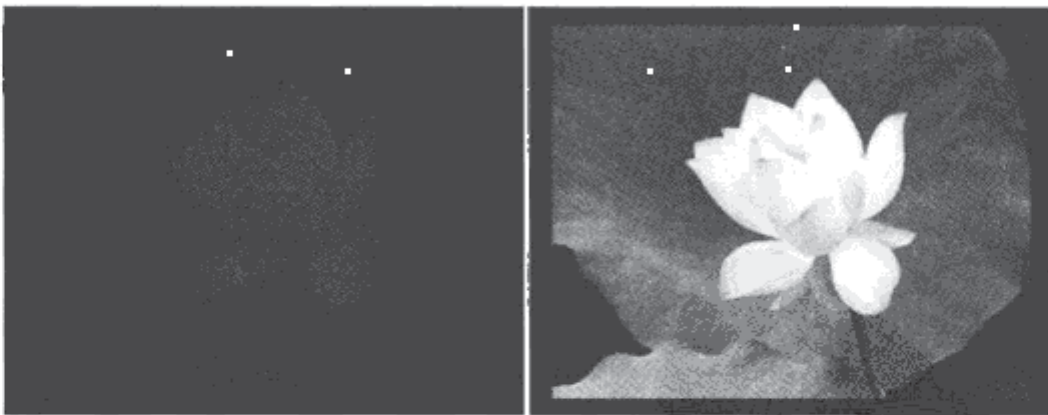


图 4-52 透明度属性的动画效果图

代码 4-76 是图 4-52 中所示的动画效果的实例代码。

代码 4-76 透明度属性动画（属性值动画）

文件名: PropertyAnimAct.java

```
1  private void onBtnValueClick() {
```



```
2      //构造属性值动画
3      ValueAnimator anim = ValueAnimator.ofFloat(0f, 1f);
4      //设置持续时长
5      anim.setDuration(5000L);
6      //设置更新侦听器
7      anim.addUpdateListener(this);
8      //启动动画
9      anim.start();
10 }
11
12 @Override
13 public void onAnimationUpdate(ValueAnimator a) {
14     //获取属性值的动画插值
15     Float value = (Float)(a.getAnimatedValue());
16     //设置透明度的值
17     mView.setAlpha(value.floatValue());
18 }
```

在代码 4-76 中，构造了一个属性值动画，其定义了属性值的渐变范围（第 3 行）和持续时长（第 5 行）。需要注意的是，当前该属性值动画仅可以用于计算插值（按值域和时间流逝进度），但是如何知道该值对应哪个属性？又如何将属性值设置给可视组件？为了解决该问题，属性动画框架采用侦听器的方式，为该属性值动画设置了一个动画更新侦听器（第 7 行）。当启动动画后，在动画更新回调方法中，可以获取动画属性的插值，并通过可视组件的属性设置方法设置给指定组件的指定属性（第 17 行）。

此外，动画框架还提供了一种不使用侦听器的方式，即直接指定组件和属性名。代码 4-77 是通过对象属性动画的方式定义和图 4-52 具有相同动画效果的实例代码。

代码 4-77 透明度属性动画（对象动画）

文件名: PropertyAnimAct.java

```
1 private void onBtnObjectClick() {
2     //构造对象属性动画
3     ObjectAnimator anim = ObjectAnimator.ofFloat(mView, "alpha", 0f, 1f);
4     //设置持续时长
5     anim.setDuration(5000L);
6     //启动动画
7     anim.start();
8 }
```

在代码 4-77 中，使用对象动画生成器可以在构造动画生成器的时候直接指定动画影响的组件、属性名及属性值域（第 3 行）。

（2）组合属性值动画

多个对象动画生成器进行组合，可以生成动画效果更为多样的动画集合。图 4-53 所示的界面中的动画效果就是组合了多个属性动画生成器所生成的组合动画效果。该效果融合了



渐变、缩放和旋转。

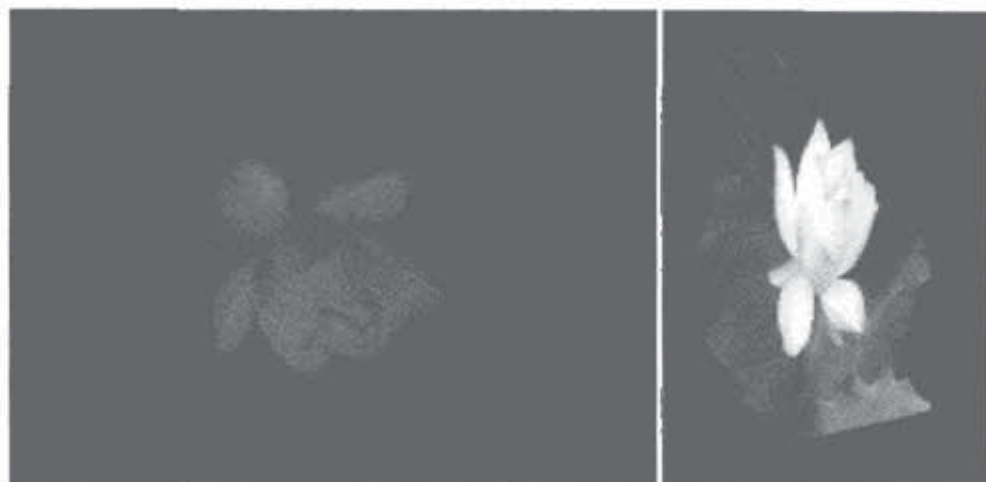


图 4-53 视图动画的界面

代码 4-78 是图 4-53 中所示的动画效果的实例代码。

代码 4-78 组合属性动画

文件名: PropertyAnimAct.java

```

1  private void onBtnCombineClick2() { //组合动画
2      //构造多个对象属性动画
3      ObjectAnimator anim = ObjectAnimator.ofFloat(mView, "alpha", 0f, 1f);
4      ObjectAnimator sx = ObjectAnimator.ofFloat(mView, "scaleX", 1f, 0.8f);
5      ObjectAnimator sy = ObjectAnimator.ofFloat(mView, "scaleY", 1f, 0.8f);
6      ObjectAnimator rx = ObjectAnimator.ofFloat(mView, "rotationX", 0f, 360f);
7      ObjectAnimator ry = ObjectAnimator.ofFloat(mView, "rotationY", 0f, 360f);
8      //构造动画集
9      AnimatorSet as = new AnimatorSet();
10     //规划动画播放次序
11     as.play(anim);
12     as.play(sx).with(sy).with(rx).before(ry);
13     //设置持续时长
14     as.setDuration(5000L);
15     //启动动画
16     as.start();
17 }

```

在代码 4-78 中, 通过动画集合构建器可以规划动画播放顺序 (第 11 行和第 12 行), 从而形成组合动画。

4.6.7 定制 Activity 组件

经过对 Android 平台中各种组件的介绍, 读者应该可以发现, 有些组件的使用是比较简单的, 但是有些组件的使用却是相当的复杂, 如列表视图、扩展类表视图、选项页控制器等。在这 3 个组件的使用中, 必须与相应的 Activity 组件配套使用。表 4-17 是 Android 平台所定义的特定 Activity 组件的说明。

表 4-17 特定的 Activity 组件的说明

类/接口	说 明
ListActivity	列表 Activity，归集于 android.app 包中
TabActivity	标签 Activity，归集于 android.app 包中
MapActivity	地图 Activity，归集于 com.google.android.maps 包中
PreferenceActivity	首选项 Activity，归集于 android.preference 包中

1. 列表 Activity（List Activity）

列表 Activity 组件中内建了列表视图组件，如果使用用户自定义的布局资源，则其资源中的列表视图组件 ID 必须参考系统的列表视图组件 ID（android:list）。列表 Activity 可以对包含列表视图组件定义的布局资源进行解析和实例化，并对列表视图的适配器进行管理。

列表 Activity 组件的使用可以参见列表视图的说明。

2. 扩展列表 Activity（ExpandableListActivity）

扩展列表 Activity 组件内建了扩展列表视图组件，如果使用用户自定义的布局资源，则其资源中的扩展列表视图组件 ID 必须参考系统的列表视图组件 ID（android:list）。扩展列表 Activity 可以对包含扩展列表视图组件定义的布局资源进行解析和实例化，并对扩展列表视图的适配器进行管理。

扩展列表 Activity 组件的使用可以参考扩展列表视图的说明。

3. 选项页 Activity（Tab Activity）

选项页 Activity 组件必须与选项页控制器（TabHost）配合使用。选项页 Activity 组件提供对选项页控制器定义的解析和实例化，并负责控制选项页的显示。

有关选项页 Activity 组件的使用可以参考选项页控制器的说明。

4. 地图 Activity（MapActivity）

地图 Activity 组件必须与地图视图（MapView）配合使用。地图 Activity 组件提供对地图视图定义的解析和实例化，并负责构建地图视图的配套组件（缩放控制组件、地图控制组件等）。

有关地图 Activity 组件的使用可以参见第 14 章。



第 5 章 底层用户界面设计

本章介绍底层用户界面控制的组件以及其使用方式。通过这些底层组件，开发者可以更多地使用底层界面的控制功能，而不拘泥于 Android 平台预定义的界面组件。在游戏开发、视频播放控制和 3D 效果展示等高级应用中会用到这些接口组件。

5.1 Android 底层用户界面

通过第 4 章的介绍，相信读者已经了解 Android 平台的大部分高级用户界面组件，并且使用这些组件来搭建自己的应用程序。但是随着应用的深入，读者会发现 Android 平台提供的这些高级组件无法满足一些对性能要求较高的应用，如游戏开发、3D 效果展现等，这些应用需要系统底层的“控制权”。

为了满足应用程序的这些高性能应用，Android 平台对屏幕设备的底层访问进行了封装，提供了代表屏幕设备的底层视图；同时，为了满足高性能绘制 3D 图形的要求，Android 平台提供对 OpenGL ES API 的支持并对其进行封装，打造出易用的 3D 图形绘制组件。

图 5-1 是 Android 平台提供的底层视图的定义的层次结构图。

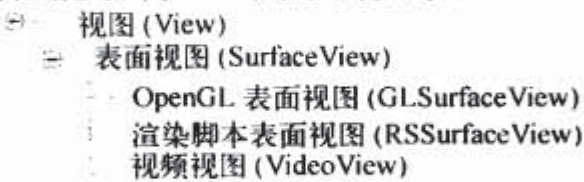


图 5-1 Android 平台底层视图层次结构图

表 5-1 是图 5-1 中底层用户界面类/接口的说明。

表 5-1 底层用户界面类/接口的说明

类/接口	说 明
SurfaceView	专用于绘制视图结构层次的表面，归集于 android.view 包
GLSurfaceView	实现于表面类，但其表面是专为显示 OpenGL 渲染，归集于 android.opengl 包
RSSurfaceView	支持渲染脚本（Render Script）的表面视图，归集于 android.renderscript 包
VideoView	用于播放视频文件，归集于 android.widget 包

5.2 底层视图绘制

Android 平台提供 SurfaceView（表面视图）组件用于对屏幕设备的底层访问。在这里，表面（Surface）与 DirectX 中表面缓冲区（DirectX Surface Buffer）的概念是一样的。为了提高绘制效率，无论是 Windows 还是 Android 平台都毫无疑问地使用了屏幕缓冲机制。

创建或销毁屏幕缓冲区的时机以及屏幕缓冲区的存取控制，是底层视图绘制尤为重要的两个环节。

5.2.1 表面视图 (Surface View)

表面视图是一个可以嵌入到应用程序的视图层次结构中、用于底层绘制的表面；而表面视图组件为表面输出到前端窗体提供了“绿色通道”，即可以将表面中绘制好的内容快速地显示到前端窗体中。开发者不能直接获取表面对象，只能通过其所绑定的表面视图组件来获取表面对象控制器 (SurfaceHolder)。

通过表面控制器的回调函数开发，可以方便地对表面对象进行初始化、绘制、销毁等控制。图 5-2 是通过表面视图绘制并旋转蝴蝶图片的实机界面。

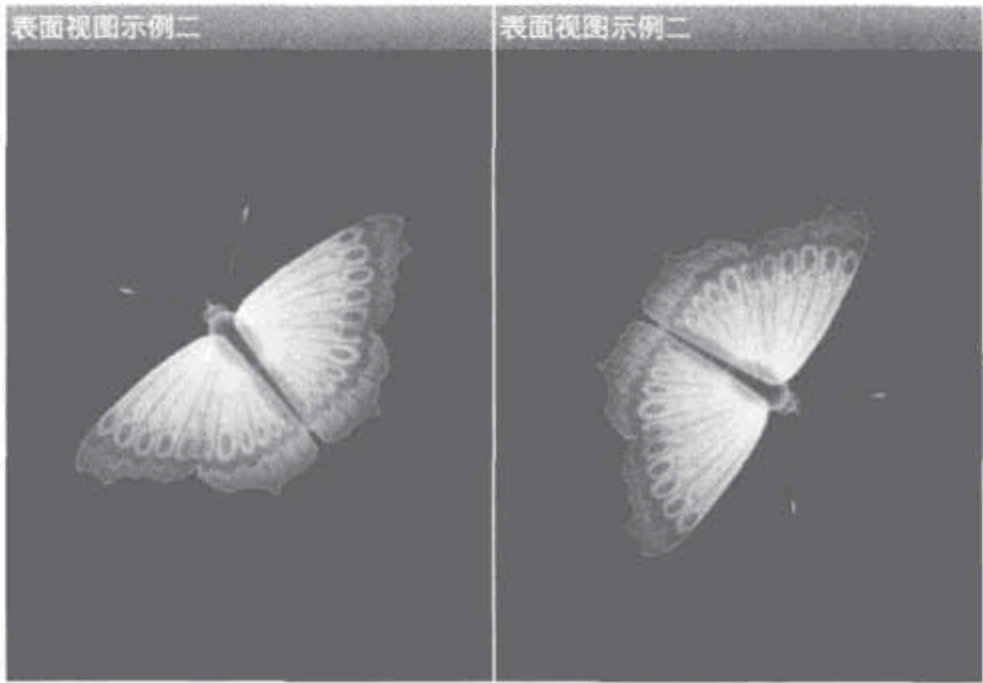


图 5-2 表面视图绘制实机界面

1. 表面视图的定义

表面视图与定制视图 (第 4 章) 的使用模式是一样的，需要使用其子类实例进行绘制。也就是说，必须先定义一个表面视图的子类，然后在子类重载的方法中进行绘制。

实际上，表面视图的子类除了继承父类之外，还必须实现表面控制器的回调管理接口 (Callback)。代码 5-1 是图 5-2 所示界面中的表面视图子类的定义。

代码 5-1 表面视图子类的定义

文件名: FoolSurfaceView.java

```
1 public class FooSurfaceView extends SurfaceView implements SurfaceHolder.Callback {
2     //绘制线程
3     private SurfaceViewThread mThread = null;
4
5     public FooSurfaceView(Context context, AttributeSet attrs) {
6         super(context, attrs);
7         SurfaceHolder holder = getHolder();
8         holder.addCallback(this);
9         //初始化绘制线程
10        mThread = new SurfaceViewThread(context, holder);
11    }
```




```
12
13      //获取绘制线程
14      public SurfaceViewThread getThread() { return (this.mThread); };
15
16      @Override
17      public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {}
18      @Override
19      public void surfaceCreated(SurfaceHolder holder) {
20          //在表面创建的同时启动绘制线程
21          mThread.setRunning(true);
22          mThread.start();
23      }
24      @Override
25      public void surfaceDestroyed(SurfaceHolder holder) {
26          mThread.setRunning(false);
27          //阻塞线程
28          try { mThread.join();
29              } catch (InterruptedException e) { e.printStackTrace(); }
30      }
31  };
```

在代码 5-1 中，当构造函数时，表面视图首先获得表面控制器（第 7 行），然后设置该表面控制器的回调控制器，指定的回调控制器正好是此表面视图类本身（第 8 行），因为该表面视图类实现了表面控制器回调管理器（第 1 行）。

对于表面的控制在重载于表面控制器回调管理器的方法中进行。表 5-2 是对表面控制器回调管理器所定义的回调函数的说明。

表 5-2 表面控制器回调管理器的说明

接 口	说 明
surfaceChanged(SurfaceHolder,int,int,int)	当表面发生改变（如屏幕旋转）后回调
surfaceCreated(SurfaceHolder)	当表面创建完毕后回调
surfaceDestroyed(SurfaceHolder)	当表面被销毁后回调

2. 表面视图的绘制控制

在代码 5-1 中，当表面视图内嵌的表面对象创建完毕之后，启动表面视图的绘制线程来实施绘制（第 22 行）。代码 5-2 是该表面视图绘制线程的定义。

代码 5-2 表面视图绘制线程的定义

文件名: SurfaceViewThread.java

```
1  class SurfaceViewThread extends Thread {
2      private Context mContext = null;
3      private SurfaceHolder mHolder = null;
4      private boolean mIsRunning = false;
5      private Bitmap mImage = null;
```



```
6      //
7      private int mRotate = 0;
8      private Matrix mMatrix = null;
9
10     public SurfaceViewThread(Context context, SurfaceHolder holder) {
11         this.mContext = context; this.mHolder = holder;
12         mImage = BitmapFactory.decodeResource(mContext.getResources(), R.drawable.hd);
13         mMatrix = new Matrix();
14     }
15
16     public void setRunning(boolean isRunning) { this.mIsRunning = isRunning; }
17
18     @Override
19     public void run() {
20         while (mIsRunning) {
21             Canvas c = null;
22             try { c = mHolder.lockCanvas(null);    //绘制前锁定
23                 synchronized (mHolder) { doRotate(c); }
24             } finally {
25                 if (c != null) { //绘制完毕后解锁
26                     mHolder.unlockCanvasAndPost(c);
27                 }
28             }
29         }
30     }
31
32     //执行画面旋转
33     private void doRotate(Canvas c) {
34         clearBkg(c);
35         //设置旋转
36         mMatrix.setRotate(getRotate() );
37         //创建临时位图
38         Bitmap bmpRotate = Bitmap.createBitmap(mImage, 0, 0,
39             mImage.getWidth(), mImage.getHeight(), mMatrix, true);
40         //绘制位图
41         c.drawBitmap(bmpRotate, (c.getWidth()-bmpRotate.getWidth())/2,
42             (c.getHeight()-bmpRotate.getHeight())/2, null);
43         //暂停
44         try { Thread.sleep(100L);
45             } catch (InterruptedException e) { e.printStackTrace(); }
46     }
47     .....
48     //清空背景
49     private void clearBkg(Canvas c) { c.drawColor(Color.BLACK, Mode.CLEAR);}
50 };
```




Android 平台开发之旅 第2版

在代码 5-2 中，当线程构造函数时，线程先获取通过表面视图传递过来的表面控制器（第 11 行），然后加载将要绘制的位图资源（第 12 行）。

关键的绘制操作是在重载的运行方法（run）中进行。在绘制之前，表面控制器必须获取表面视图的表面对象的画布接口（Canvas），并锁定表面对象（第 22 行）；然后通过画布接口来实施绘制操作（第 23 行）；绘制完毕后，表面控制器必须释放表面对象，并提交表面对象上绘制的内容（第 26 行），这样使绘制结果在前端窗体中显示。

注意：在绘制过程中，必须保证表面控制器实例的同步（代码 5-2 中第 23 行），因为表面对象的操作都是通过表面控制器实例来完成的；否则，就会造成绘制的混乱。

3. 在布局资源中定义表面视图

当表面视图子类定义完毕，开发者可以在程序的布局资源中引用该视图组件。代码 5-3 是使用表面视图组件定义主布局的代码。

代码 5-3 使用表面视图组件定义主布局

文件名：main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="fill_parent" android:layout_height="fill_parent" >
4      <foolstudio.demo.view.surfaceview2.FooSurfaceView android:id="@+id/sview"
5          android:layout_width="fill_parent" android:layout_height="fill_parent"/>
6  </FrameLayout>

```

代码 5-3 与定制视图的定义方式相同，其标签需要提供完整的类名（第 4 行）；否则，资源解析器将无法找到该标记所对应的组件类。

4. 在前端显示表面视图

在 Activity 组件中只需要将包含表示视图的布局资源设置为其内容视图即可。代码 5-4 是表面视图示例程序的 Activity 组件的框架定义。

代码 5-4 表面视图示例程序的 Activity 组件的框架定义

文件名：SurfaceView2Act.java

```

1  public class SurfaceView2Act extends Activity {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main);
6      }
7  };

```

5. 表面视图的使用场合

表面视图除了用于图形、图像的底层绘制之外，还可以用来播放视频文件。媒体播放器会指定一个表面视图实例作为渲染视频内容的载体。详细介绍参见多媒体开发章节。

5.2.2 底层视图的绘制接口

无论是定制视图还是表面视图，到了最后绘制视图内容时，都用到了画布接口。画布接口提供了丰富的绘制方法，开发者可以利用这些方法绘制出复杂的图形。

5.3 OpenGL 视图绘制

5.3.1 OpenGL ES 概述

嵌入式系统 OpenGL (OpenGL for Embedded System, OpenGL ES) 是专门针对嵌入设备 (如手机、PDA 和游戏机等) 的计算性能和应用需求, 对 OpenGL 3D 库进行裁剪和定制之后的子集, 其官方网站为 <http://www.khronos.org/opengles/>。

当前 OpenGL ES 主要有两个版本: OpenGL ES 1.x 版本针对固定管线硬件; OpenGL ES 2.x 版本针对可编程管线硬件。其中, OpenGL ES 1.0 是以 OpenGL 1.3 规范为基础; OpenGL ES 1.1 是以 OpenGL 1.5 规范为基础。

5.3.2 Android 平台对 OpenGL ES 的支持

Android 平台既提供了对 OpenGL ES 1.0 的支持也提供了对 OpenGL ES 1.1 的支持, 并且按 OpenGL ES 的使用模式封装成可视视图 GLSurfaceView (OpenGL 表面视图), 简化开发者对 OpenGL ES API 的使用。

5.3.3 OpenGL 表面视图

图 5-3 是使用 OpenGL 表面视图进行背景颜色随机切换的实机界面, 其中采用的颜色缓冲的机制使用了 OpenGL ES 1.0 API。



图 5-3 OpenGL 表面视图绘制界面

1. OpenGL 表面视图定义

OpenGL 表面视图和表面视图的使用模式相同, 开发者必须先定义一个继承于 OpenGL 表面视图的子类, 然后在子类重载的方法中进行绘制。

OpenGL 表面视图除了继承父类 GLSurfaceView 之外, 还必须实现 OpenGL 表面视图渲染器 (GLSurfaceView.Renderer)。使用 OpenGL ES 功能集进行绘制的操作都在重载于该渲染器的方法中。

代码 5-5 是图 5-3 所示的界面中 OpenGL 表面视图的定义。



代码 5-5 OpenGL 表面视图的定义

文件名: GLColorView.java

```
1 public class GLColorView extends GLSurfaceView implements GLSurfaceView.Renderer {
2     //随机数发生器
3     private Random mRandomizer = new Random(System.currentTimeMillis() );
4
5     public GLColorView(Context context, AttributeSet attrs) {
6         super(context, attrs);
7         //设置渲染器
8         this.setRenderer(this);
9     }
10
11     @Override
12     public void onDrawFrame(GL10 gl) {
13         //设置颜色
14         gl.glClearColor(mRandomizer.nextFloat(), //红色比重
15                         mRandomizer.nextFloat(), //绿色比重
16                         mRandomizer.nextFloat(), //蓝色比重
17                         mRandomizer.nextFloat() ); //透明比重
18         //暂停 1s
19         try { Thread.sleep(1000L);
20             } catch (InterruptedException e) { e.printStackTrace(); }
21         //清空背景
22         gl.glClear(GL10.GL_COLOR_BUFFER_BIT|GL10.GL_DEPTH_BUFFER_BIT);
23     }
24     @Override
25     public void onSurfaceChanged(GL10 gl, int width, int height) {}
26     @Override
27     public void onSurfaceCreated(GL10 gl, EGLConfig config) {
28         //初始化背景
29         gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
30         gl.glEnable(GL10.GL_LIGHT0);
31         gl.glEnable(GL10.GL_FOG);
32     }
33 };
```

在代码 5-5 中，当构造函数时，OpenGL 表面视图将其自身实例设置为渲染器（第 8 行），因为该视图实现了 OpenGL 表面视图渲染器（第 1 行）。

对于视图的绘制都在重载于渲染器的方法中进行。表 5-3 是对 OpenGL 表面视图渲染器所定义的接口的说明。

表 5-3 OpenGL 表面视图渲染器接口的说明

接 口	说 明
onDrawFrame(GL10)	当绘制当前帧时回调
onSurfaceChanged(GL10, int, int)	当视图大小发生改变时回调
onSurfaceCreated(GL10, EGLConfig)	当视图首次被创建后回调

在 OpenGL 表面视图渲染器的回调函数中，开发者可以通过 OpenGL ES 1.0 的功能接口来实现相应的 3D 绘制功能。

2. 引用 OpenGL 表面视图

当 OpenGL 表面视图子类定义完毕，开发者可以在布局资源中引用该视图组件。代码 5-6 是引用 OpenGL 表面视图组件定义主布局的代码。

代码 5-6 引用 OpenGL 表面视图组件定义主布局

文件名: main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent" android:layout_height="fill_parent">
5     <foolstudio.demo.opengl.colordemo.GLColorView
6         android:layout_width="fill_parent" android:layout_height="fill_parent"/>
7 </LinearLayout>
```

在代码 5-6 中，OpenGL 表面视图与表面视图的定义方式相同，其定义标签需要提供完整的类名（第 5 行）。

3. 展现 OpenGL 表面视图

在 Activity 组件中只需将包含表示 OpenGL 视图的布局资源设置为其内容视图即可。代码 5-7 是主 Activity 组件的定义。

代码 5-7 主 Activity 组件的定义

文件名: GLColorDemoAct.java

```
1 public class GLColorDemoAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6     }
7 };
```

5.3.4 渲染脚本表面视图

不熟悉 OpenGL 的读者可能会觉得代码 5-5 的内容晦涩难懂，而实际上，使用 OpenGL API 绘制图形或立方体的过程比代码 5-5 更为复杂。另外，OpenGL 的使用需要较深的数学功底和空间想象力，所以，并不是很容易上手。

1. 渲染脚本定义

为了简化 OpenGL 编程，Android 3 平台纳入了渲染脚本（Render Script）引擎，开发者只需编写脚本即可实现与 OpenGL 代码相同的效果。代码 5-8 就是一个渲染脚本文件的内容。



代码 5-8 渲染脚本的定义

文件名: foo.rs

```
1  #pragma version(1)
2  #pragma rs java_package_name(foolstudio.demo.rscript)
3
4  #include "rs_graphics.rsh"
5
6  int gTouchX;
7  int gTouchY;
8
9  void init() {
10     gTouchX = 50.0f;
11     gTouchY = 50.0f;
12 }
13
14 int root(int id) {
15     //清除背景色
16     rsgClearColor(0.0f, 0.0f, 0.0f, 0.0f);
17     //设置字体颜色
18     rsgFontColor(1.0f, 1.0f, 1.0f, 1.0f);
19     //绘制文本
20     rsgDrawText("Hello World!", gTouchX, gTouchY);
21     //返回重绘间隔（毫秒）
22     return 20;
23 }
```

代码 5-8 使用的是 C 语言语法格式，在 root 方法中（第 14 行），通过形如“rsgXXX”的方法（如 rsgClearColor、rsgFontColor 和 rsgDrawText）实现绘制。该脚本中所包含的 rs_graphics.rsh 头文件（第 4 行）存在于 SDK 安装文件夹下，如图 5-4 所示。

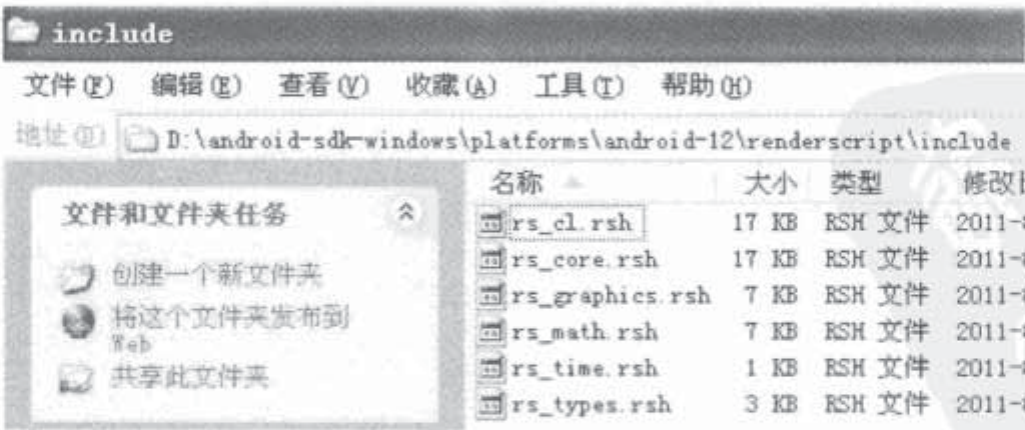


图 5-4 渲染脚本的头文件路径

当脚本文件创建完毕，ADT 会根据脚本文件自动生成相关的文件，如图 5-5 所示。

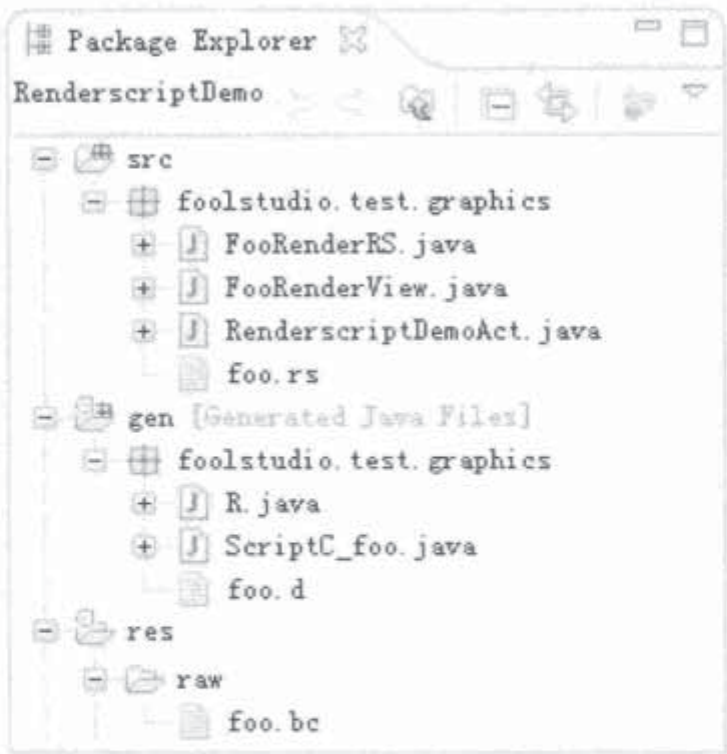


图 5-5 根据渲染脚本生成的文件

表 5-4 是对图 5-5 中由 ADT 自动生成的文件的说明。

表 5-4 渲染脚本自动生成文件的说明

内 容	说 明
foo.d	记录包含文件的路径信息
ScriptC_foo.java	脚本引擎实例定义文件
foo.bc	数据文件

2. 脚本引擎定义

代码 5-9 是脚本引擎的定义（即文件 ScriptC_foo.java 的内容），脚本引擎主要用于处理渲染请求。

代码 5-9 脚本引擎的定义

文件名：ScriptC_foo.java

```
1 public class ScriptC_foo extends ScriptC {
2     // Constructor
3     public ScriptC_foo(RenderScript rs, Resources resources, int id) {
4         super(rs, resources, id);
5     }
6
7     private final static int mExportVarIdx_gTouchX = 0;
8     private int mExportVar_gTouchX;
9     public void set_gTouchX(int v) {
10         mExportVar_gTouchX = v;
11         setVar(mExportVarIdx_gTouchX, v);
12     }
13 }
```



Android 平台开发之旅 第2版

```

14     public int get_gTouchX() { return mExportVar_gTouchX; }
15
16     private final static int mExportVarIdx_gTouchY = 1;
17     private int mExportVar_gTouchY;
18     public void set_gTouchY(int v) {
19         mExportVar_gTouchY = v;
20         setVar(mExportVarIdx_gTouchY, v);
21     }
22
23     public int get_gTouchY() { return mExportVar_gTouchY; }
24 }

```

3. 渲染脚本表面视图定义

代码 5-10 是渲染脚本表面视图的定义。

代码 5-10 渲染脚本表面视图的定义

文件名: FooRenderView.java

```

1  public class FooRenderView extends RSSurfaceView {
2      //渲染脚本引擎和渲染器
3      private RenderScriptGL mRs = null;
4      private FooRenderRS mRender = null;
5
6      public FooRenderView(Context context) { super(context);
7          initRenderScript();
8      }
9
10     private void initRenderScript() {
11         if(mRs != null) { return; }
12
13         RenderScriptGL.SurfaceConfig sc = new RenderScriptGL.SurfaceConfig();
14         mRs = this.createRenderScriptGL(sc);
15         mRender = new FooRenderRS();
16         mRender.init(mRs, getResources());
17     }
18
19     @Override
20     protected void onAttachedToWindow() { super.onAttachedToWindow();
21         initRenderScript();
22     }
23     @Override
24     protected void onDetachedFromWindow() {
25         mRender = null;
26         if (mRs != null) { mRs = null; destroyRenderScriptGL(); }
27     }
28     @Override
29     public boolean onTouchEvent(MotionEvent ev) {

```



```

30         if (ev.getAction() == MotionEvent.ACTION_DOWN) {
31             mRender.onActionDown((int)ev.getX(), (int)ev.getY());
32             return true;
33         }
34         return false;
35     }
36 };

```

在代码 5-10 中，渲染脚本表面视图有两个重要的成员：渲染脚本和渲染器（第 3 行和第 4 行），渲染器需要通过渲染脚本进行初始化（第 16 行）。在视图的触摸事件回调方法中，渲染器用于执行响应（第 31 行）。

4. 渲染器定义

代码 5-11 是代码 5-10 中提到的渲染器的定义。

代码 5-11 渲染器的定义

文件名：FooRenderRS.java

```

1  public class FooRenderRS {
2      private Resources mRes = null;
3      private RenderScriptGL mRs = null;
4      private ScriptC_foo mScript = null;
5
6      public FooRenderRS() {}
7      public void init(RenderScriptGL rs, Resources res) {
8          this.mRs = rs;
9          this.mRes = res;
10         initRS();
11     }
12
13     public void onActionDown(int x, int y) {
14         mScript.set_gTouchX(x);
15         mScript.set_gTouchY(y);
16     }
17
18     private void initRS() {
19         mScript = new ScriptC_foo(mRs, mRes, R.raw.foo);
20         mRs.bindRootScript(mScript);
21     }
22 };

```

在代码 5-11 中，渲染器包含两个重要成员：渲染脚本引擎和脚本（第 3 行和第 4 行），其中脚本需要渲染器引擎、资源和脚本进行初始化（第 19 行），最后将脚本与引擎进行绑定（第 20 行）。

至此可知，渲染脚本引擎是脚本和渲染器之间的“桥梁”，不仅用于从资源中获取脚本定义，还需要将渲染请求发送给嵌在视图中的渲染器，三者关系如图 5-6 所示。

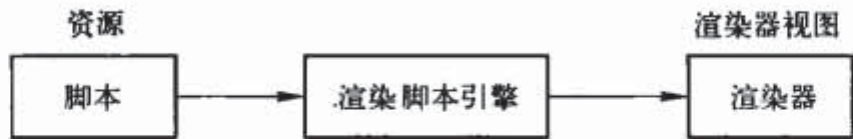


图 5-6 脚本、渲染脚本引擎和渲染器的关联示意图

5. 应用程序主 Activity 框架

代码 5-12 是主 Activity 组件的框架定义。

代码 5-12 主 Activity 组件的框架定义

文件名: RsDemoAct.java

```
1 public class RsDemoAct extends Activity {
2     //渲染视图
3     private FooRenderView mView = null;
4
5     @Override
6     public void onCreate(Bundle savedInstanceState) {
7         super.onCreate(savedInstanceState);
8         //设置内容视图
9         mView = new FooRenderView(this);
10        setContentView(mView);
11    }
12    @Override
13    protected void onResume() { super.onResume();
14        mView.resume();
15    }
16    @Override
17    protected void onPause() { super.onPause();
18        mView.pause();
19    }
20 };
```

在代码 5-12 中，主 Activity 组件将渲染脚本视图设置为其内容视图（第 10 行），如此一来即可在前端展现该视图的渲染效果。

5.3.5 Android 平台中 OpenGL 使用说明

实际上，读者可以把 OpenGL 表面视图理解为 OpenGL API 与 Android 平台的应用接口。通过这个接口，OpenGL 开发人员可以方便地将其工作平台转移到 Android 平台上来。而想要在 Android 平台中绘制 OpenGL 效果的图形，还必须夯实 OpenGL 的应用基础。

作为入门而言，读者可以通过 OpenGL 表面视图来实现 OpenGL 提供的简单功能；至于把 OpenGL 的功能发挥得淋漓尽致则已经不属于 Android 平台开发的范畴。所以，代码 5-5 的示例中，仅用到了 OpenGL 中颜色的功能，而真正的 3D 图形的绘制、立方体渲染、纹理

等 OpenGL 的特效功能在此不进行深入介绍。

5.4 视频视图 (VideoView)

视频视图的主要应用不是绘制，而是播放视频文件（3gp、mp4 等类型），它继承于表面视图，但其封装了渲染视频帧的过程，从而简化了视频的播放过程。读者可以参考第 10 章。



第6章 文件管理

本章对 Android 平台中的文件类型进行详细的说明，从系统和应用的角度介绍文件管理的模式。与桌面平台文件管理相比，Android 平台定义了多种文件类型，如资源文件、资产文件、存储设备文件、应用程序文件等。

6.1 Android 平台中的文件

作为存储数据的载体，文件无疑是各个平台不可缺少的组成部分。不仅如此，Android 平台还提供了多种形态的文件，如资源文件、存储设备文件和应用程序文件。

除此之外，Android 平台还提供了对文件进行监视的功能，通过监视接口，应用程序可以对指定路径下的文件操作（如创建、打开和删除等）进行监视。

6.2 资源文件和资产文件

相信读者对资源文件的概念已经不再陌生，在工程文件夹下的 `res` 文件夹中的文件都是资源文件，读者熟知的有界面布局定义文件、图片文件等。相比之下，读者对资产文件可能要生疏一些，这些文件存放于工程文件夹下的 `assets` 文件夹。

实际上，就文件内容结构而言，资源文件和资产文件并无太大的差异，两者的差异主要体现在使用模式方面。

6.2.1 资源文件

需要声明的是，这里所说的资源文件是指单一资源文件（即一个文件代表一个资源），并不包括那些组合资源文件（即一个文件包含多个资源）。资源文件按照类型存放于资源文件夹下的各个子文件夹中。

1. 资源文件的类型

表 6-1 是对 Android 平台资源文件类型的说明。

表 6-1 资源文件类型的说明

类 型	说 明
原文件	非标准格式文件，存放于 <code>raw</code> 子文件夹
图片	标准图片文件（png、jpg 等），存放于 <code>drawable</code> 子文件夹
布局定义文件	使用 <code>xml</code> 定义的界面布局，存放于 <code>layout</code> 子文件夹
xml 文件	标准 <code>xml</code> 文件，存放于 <code>xml</code> 子文件夹

当添加资源文件时，只需将文件放入到指定的文件夹即可。当资源打包工具（aapt）检测到有新的资源加入时，会重新为每个资源生成唯一的数值标识，并自动更新 R.java 文件，如图 6-1 所示。

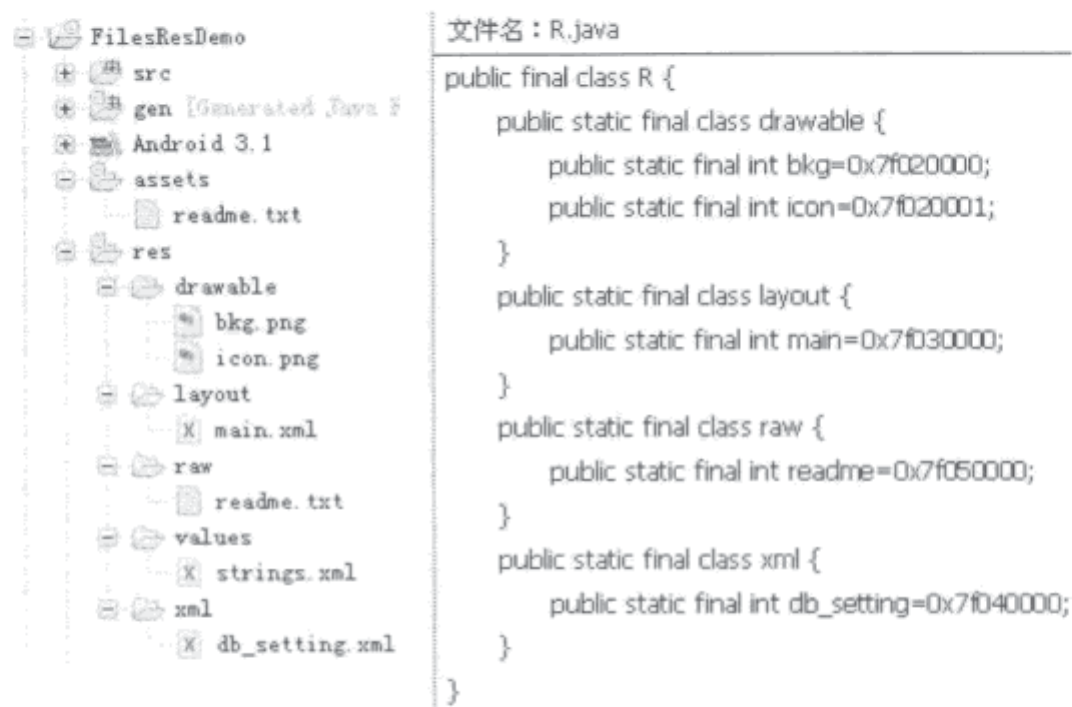


图 6-1 资源文件结构与资源标识对应

2. 资源文件的引用

资源文件的引用需要参考资源标识。在 Activity 组件中，资源管理器（Resources）通过指定的资源标识来获取资源对象。

(1) 原文件

在图 6-2 所示的界面中，应用程序通过资源标识来读取原文件资源的内容。

代码 6-1 是读取原文件资源的主要代码。



图 6-2 读取原文件资源

代码 6-1 读取原文件资源

文件名: FilesResAct.java	
1	private void readRaw(int resid) throws IOException { //读取原文件内容
2	//获取原文件输入流
3	InputStream is = this.getResources().openRawResource(resid);
4	DataInputStream dis = new DataInputStream(is);
5	byte[] data = new byte[is.available()];
6	dis.readFully(data);
7	dis.close();
8	is.close();
9	//显示文件内容
10	print(new String(data,"GBK"));
11	}



在代码 6-1 中，资源管理器使用 `openRawResource` 方法打开指定原文件资源的输入流（第 3 行），继而通过输入流读取原文件内容。其中，原文件的资源标识即 `R.raw.readme` 所定义的值。

(2) 图片资源

在图 6-3 所示的界面中，应用程序通过资源标识来读取图片文件，并设置图片为其背景。

代码 6-2 是引用图片文件资源并设置为文本视图背景的主要代码。



图 6-3 读取图片文件资源

代码 6-2 引用图片文件资源

文件名: FilesResAct.java

```
this.getWindow().setBackgroundDrawableResource(resid);
```

(3) 布局定义文件

一般来说，布局定义文件有两种使用模式：第一种是将布局设置为 `Activity` 组件的内容视图；第二种是使用布局填充器将布局定义文件填充成视图对象实例。

(4) XML 文件

有关 XML 文件的解析参见第 13 章。

6.2.2 资产文件

读者可能会注意到，图 6-1 中资源打包工具并没有为资产文件生成资源标识。仅此一点即可知资产文件与资源文件在使用方面存在一定的差异。

1. 资产文件的引用

既然资产文件没有资源标识，那又该如何引用其文件呢？资产文件可以通过文件名进行访问。代码 6-3 是读取资产文件内容的关键代码。

代码 6-3 读取资产文件内容

文件名: FilesResAct.java

```
1 private void readAsset(String fileName) {
2     try { //读取资产文件内容
3         InputStream is = this.getAssets().open(fileName);
4         int bufferSize = is.available();
5         byte data[] = new byte[bufferSize];
6         is.read(data, 0, bufferSize);
7         is.close();
8         //显示文件内容
9         print(new String(data,"GBK"));
10    } catch (IOException e) { e.printStackTrace(); }
11 }
```


在代码 6-3 中，使用资产管理器（AssetManager）的 open 方法打开指定文件名的资产文件的输入流（第 3 行），继而通过输入流接口来读取其内容。其中，资产文件名不包含路径，仅仅是基本名称 readme.txt。

2. 资产文件的遍历

资源的条目可以由资源打包工具生成的资源类“R”进行管理，而资产文件通过资产管理器进行管理。使用资产管理器的接口可以实现对应用程序的资产文件进行遍历。图 6-4 所示的微调控件中的前 4 个条目即为资产文件的遍历结果。

代码 6-4 是遍历资产文件的关键代码。



图 6-4 遍历资产文件

代码 6-4 遍历资产文件

文件名：FilesResAct.java

```
1 private void initSpinner() {
2     ArrayList<String> items = new ArrayList<String>();
3     try { //遍历资产文件
4         String[] assets = this.getAssets().list("");
5         for(int i=0; i<assets.length; ++i) {
6             items.add(assets[i]);
7         }
8     } catch (IOException e) { e.printStackTrace(); }
9     .....
10 }
```

在代码 6-4 中，使用资产管理器的 list 方法获取当前程序的所有资产文件名信息（包括文件夹和文件，第 4 行）。

6.3 存储设备文件

无论是资源文件还是资产文件，对于应用程序都是只读的，因为这些文件包含于安装文件（apk 文件）中；对于安装包文件，只能通过包管理器来进行管理，用户程序无法直接修改其中的文件。

相比之下，对于存储设备上的文件，应用程序不受这些限制，可以自由地对手机内存或外部存储设备上的文件进行读写操作。

6.3.1 存储设备文件操作

存储设备上的文件操作和桌面平台相同。在 J2SE 平台中，文件类（File）用于抽象地表示一个通过路径名来标识的文件系统实体（文件或文件夹）。Android 平台也定义了该类，通过该类的方法，开发者可以实现桌面平台的文件操作功能。



6.3.2 文件浏览器

智能手机都提供了浏览用户文件的功能，下面也将引领读者如何开发自己的文件浏览器。图 6-5 所示为该文件浏览器的实机界面。

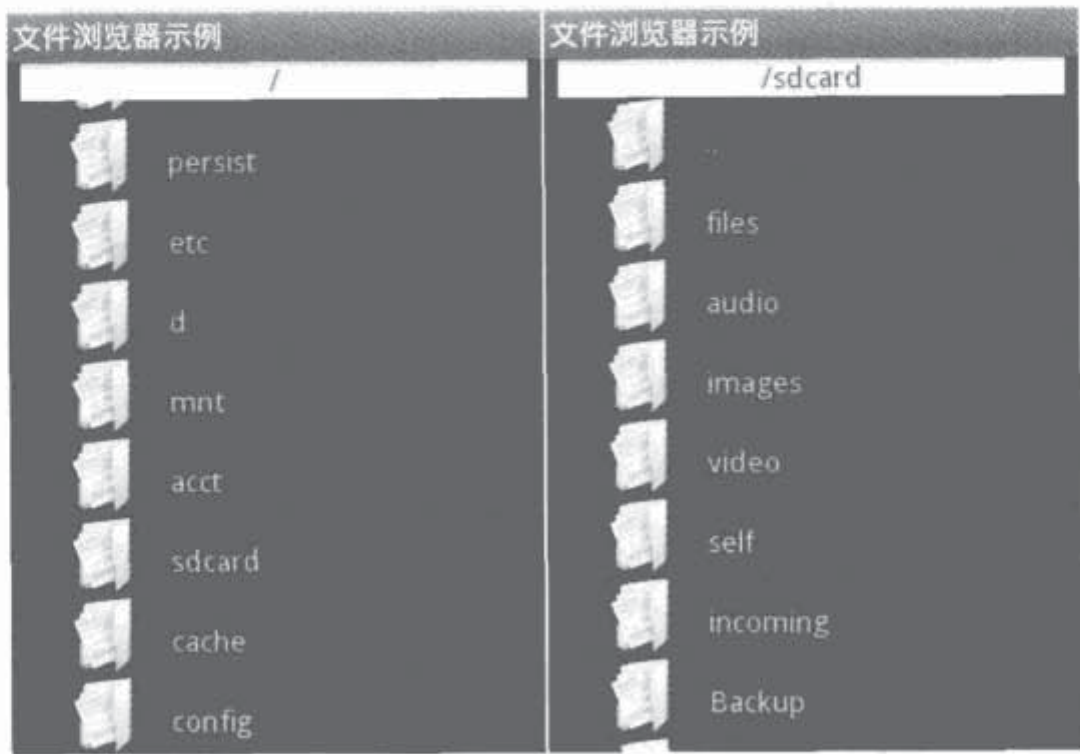


图 6-5 文件浏览器的实机界面

1. 文件浏览器界面

代码 6-5 是该文件浏览器的主布局定义，该布局使用列表视图（第 11 行）作为展现文件信息的重要组件。

代码 6-5 文件浏览器的主布局定义

文件名: main_view.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical" android:padding="8px"
4     android:layout_width="fill_parent" android:layout_height="fill_parent">
5     <TextView android:id="@+id/text_view"
6         android:layout_width="fill_parent" android:layout_height="wrap_content"
7         android:background="#EFFFFF" android:textColor="#FF0000"
8         android:gravity="center_horizontal" />
9     <ListView android:id="@id/android:list"
10         android:layout_width="fill_parent" android:layout_height="fill_parent"
11         android:layout_weight="1"/>
12 </LinearLayout>
```

2. 主 Activity 组件

代码 6-6 是文件浏览器的主 Activity 组件的框架定义，该 Activity 组件为列表 Activity 组件的子类，其以列表视图作为内容界面，以定制的适配器作为列表适配器。

代码 6-6 文件浏览器的主 Activity 组件的框架定义

文件名: FileSystemBrowserAct.java

```

1  public class FileSysBrowserAct extends ListActivity {
2      private TextView mTitle = null;
3      private String mCurDir = "";
4      private ArrayList<FileInfo> mItems = new ArrayList<FileInfo>();
5
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.main_view);
10         //抬头文本条
11         mTitle = (TextView)findViewById(R.id.text_view);
12         mTitle.setText(IConfig.DEFAULT);
13         //初始化数据集
14         mItems.add(new FileInfo(IConfig.ROOT, true) );
15         //设置适配器
16         ListAdapter adapter = new FileArrayAdapter(this,
17                                                     R.layout.item_view, R.id.label, mItems);
18         this.setAdapter(adapter);
19     }
20     .....
21 };

```

在代码 6-6 中, 定义了一个 FileArrayAdapter 类型的列表适配器 (第 16 行), 该适配器的数据集记录类型是 FileInfo (第 14 行), 其行视图的资源 ID 为 item_view。

在界面初始化过程中, 使用根节点信息初始化列表适配器所绑定的数据集 (第 14 行), 当单击根节点项, 列表视图就会显示根目录所包含的内容。代码 6-7 是列表项单击事件的回调处理方法。

代码 6-7 列表项单击事件回调处理

文件名: FileSystemBrowserAct.java

```

1  @Override
2  protected void onItemClick(ListView lv, View v, int pos, long id) {
3      String fName = lv.getAdapter().getItem(pos).toString();
4
5      if(fName.endsWith(IConfig.PARENT)) { // (1) 当前项是返回上层
6          if(mCurDir.endsWith(IConfig.ROOT)) { // (1.1) 当前文件夹已经是根目录
7              mCurDir = "";
8          } else { int index = mCurDir.lastIndexOf(File.separator);
9                  if(index == 0) { // (1.2) 当前文件夹的上一级为根目录
10                      mCurDir = IConfig.ROOT;
11                  } else { // (1.3) 当前文件夹 为普通文件夹
12                      mCurDir = mCurDir.substring(0, index);

```




```

13         }
14     }
15     notifyDataSetChanged(lv);
16     } else if(fName == IConfig.ROOT) { // (2) 当前项是根文件夹
17         mCurDir = (mCurDir + fName);
18         notifyDataSetChanged(lv);
19     } else { //当前项是普通文件夹或文件
20         File f = new File(mCurDir + File.separator + fName);
21         if(f.isDirectory()) { // (3) 为子文件夹
22             if(mCurDir.endsWith(IConfig.ROOT)) { // (3.1) 当前文件夹为根目录
23                 mCurDir = mCurDir + fName;
24             } else { mCurDir = mCurDir + File.separator + fName; }
25             notifyDataSetChanged(lv);
26         } else { // (4) 为文件
27             String filePath = null;
28             if(mCurDir.endsWith(IConfig.ROOT)) { // (4.1) 当前文件夹为根目录
29                 filePath = mCurDir+fName;
30             } else { filePath = mCurDir+File.separator+fName; }
31             Toast.makeText(this, filePath, Toast.LENGTH_LONG).show();
32         }
33     }
34 }

```

在代码 6-7 中，对点选项的判断规则存在以下 4 种情形。

1) 所选项为返回上层目录（第 5 行），又存在两种情形：

- 若当前已是根目录（第 6 行），则将目标文件夹设置为空（文件系统）。
- 若当前文件夹的上一级是根目录（第 8 行），则目标文件夹为根目录。

2) 所选项是根目录（第 16 行），将目标文件夹设置为根目录。

3) 所选项是子文件夹或文件（第 19 行），又存在两种情形：

- 若当前是根目录（第 22 行），则目标文件夹为根目录+所选文件夹。
- 若当前是普通目录，则目标文件夹为当前文件夹+分隔符+所选文件夹（第 24 行）。

4) 所选项是文件，又存在两种情形：

- 若当前是根目录，则目标文件为根目录+所选文件（第 29 行）。
- 若当前是普通目录，则目标文件为当前目录+分隔符+所选文件（30 行）。

当用户点选文件时，将会弹出文件信息的提示框（第 31 行）。

3. 列表视图记录集管理

在代码 6-7 中，当点选目标是文件夹时，只是更新了当前的路径（变量 mCurDir），然后通过调用 notifyDataSetChanged 方法通知列表视图对数据集的改变进行处理。代码 6-8 是对列表视图的记录集进行管理的主要代码。

代码 6-8 列表视图记录集管理

文件名：FileSystemBrowserAct.java

```
1 @SuppressWarnings("unchecked")
```



```

2 private void notifyDataSetChanged(ListView lv) {
3     if(mCurDir == "") { //文件系统
4         mItems.clear();
5         mItems.add(new FileInfo(IConfig.ROOT, true));
6     }
7     else { listFolder(mCurDir);
8         mItems.add(0, new FileInfo(IConfig.PARENT, true));
9     }
10    //数据集更新提示
11    ((ArrayAdapter<FileInfo>)lv.getAdapter()).notifyDataSetChanged();
12    mTitle.setText((mCurDir.length() < 1) ? "根目录" : mCurDir);
13 }
14
15 private void listFolder(String path) { //遍历指定目录中的内容（不包括子文件夹）
16     File f = new File(path);
17     if(f.exists() == false) { return; }
18     //初始化数据集
19     mItems.clear();
20     //文件夹
21     if(f.isDirectory() ) { String files[] = f.list();
22         if(files==null) { return; }
23         for(int i = 0; i < files.length; ++i) {
24             File child = null;
25             if(path.endsWith(IConfig.ROOT)) { //根文件夹
26                 child = new File(path + files[i]);
27             } else { //子文件夹
28                 child = new File(path + File.separator + files[i]);
29             }
30             mItems.add(new FileInfo(files[i], child.isDirectory()));
31         }
32     } else if(f.isFile()) { //文件
33         mItems.add(new FileInfo(f.getName(), false));
34     }
35 }

```

在代码 6-8 中，listFolder 方法（第 15 行）用于遍历指定文件夹中的内容（不递归），然后使用每一条文件的信息创建一个文件信息对象（FileInfo），并添加到列表适配器所绑定的数据集中（第 30 行和第 33 行）。

在代码 6-8 的第 11 行中，使用 getAdapter 方法获取该 Activity 组件的列表适配器，然后再通过该适配器的 notifyDataSetChanged 方法“告诉”视图组件，数据集已经改变了，通知其进行数据显示更新。

4. 列表视图的行视图

在图 6-5 中，行视图包含一个图片视图和一个文本视图，其中文本视图用于显示文件名，而图片视图中的图标用于指示当前项是文件还是文件夹。代码 6-9 是该行视图的定义。



代码 6-9 列表视图行视图的定义

文件名: item_view.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="fill_parent" android:layout_height="wrap_content"
4      android:orientation="vertical" >
5      <ImageView android:id="@+id/icon" android:paddingLeft="20dp"
6          android:layout_width="wrap_content" android:layout_height="wrap_content"
7          android:src="@drawable/folder" android:scaleType="matrix"/>
8      <TextView android:id="@+id/label" android:textSize="8pt"
9          android:layout_width="wrap_content" android:layout_height="wrap_content"
10         android:layout_toRightOf="@id/icon" android:layout_alignTop="@id/icon"/>
11 </RelativeLayout>

```

在代码 6-9 中, 该行视图采用的是相对布局 (第 2 行), 这里使用相对布局可以较好地控制图标和文本的对齐及间距。

5. 文件列表适配器

代码 6-10 是文件浏览器中列表适配器的定义, 该适配器继承于数组适配器。

代码 6-10 文件列表适配器的定义

文件名: FileArrayAdapter.java

```

1  public class FileArrayAdapter extends ArrayAdapter<FileInfo> {
2      //上下文
3      private Activity mContext = null;
4
5      public FileArrayAdapter(Activity context, int resource,
6          int textViewResourceId, List<FileInfo> objects) {
7          super(context, resource, textViewResourceId, objects);
8          this.mContext = context;
9      }
10
11     @Override
12     public View getView(int pos, View convertView, ViewGroup parent) {
13         final LayoutInflater inflater = mContext.getLayoutInflater();
14         //获取项目 View (每一行)
15         View itemView = inflater.inflate(R.layout.item_view, null, false);
16         //获取每一行子项
17         ImageView iv = (ImageView)itemView.findViewById(R.id.icon);
18         TextView tv = (TextView)itemView.findViewById(R.id.label);
19         FileInfo fileInfo = this.getItem(pos);
20         //按照文件类型设置图标
21         if(fileInfo.isFolder()) { iv.setImageResource(R.drawable.folder);
22         } else { iv.setImageResource(R.drawable.file); }
23         //设置文件名

```



```

24         tv.setText(fileInfo.getName());
25
26         return (itemView);
27     }
28 };

```

实际上, 如果只考虑显示文件名, 则在代码 6-6 中可以直接使用数组适配器来绑定列表视图, 因为数组适配器只能提供单个显示数据的组件, 恰好可以用来显示文件名。但这样会面临新的问题: 如何仅通过文件名的信息来进行文件浏览? 虽然通过文件的全路径可以获取文件的上层以及当前的路径信息, 但显示全路径会造成列表项冗长, 不便于查看。

所以, 如果既要根据文件信息来设置文件类型图标, 又要显示基本文件名, 那么就不能将列表项定义为简单的字符串类型, 而要定制列表项类型。在代码 6-10 中, 列表适配器的记录类型为 `FileInfo` (第 19 行), 通过该类型对象可以获取文件实体的文件类型和文件名 (第 21 行和第 24 行)。

6. 列表项记录

代码 6-11 是列表视图适配器所绑定的数据集的记录定义。

代码 6-11 数据集的记录定义

文件名: `FileInfo.java`

```

1  public class FileInfo {
2      private boolean isFolder;
3      private String name;
4
5      public FileInfo(String _name, boolean _isFolder) {
6          setName(_name);
7          setFolder(_isFolder);
8      }
9
10     public void setFolder(boolean isFolder) { this.isFolder = isFolder; }
11     public boolean isFolder() { return isFolder; }
12     public void setName(String name) { this.name = name; }
13     //必须重载 toString 方法
14     @Override
15     public String toString() { return (getName()); }
16     public String getName() { return name; }
17 };

```

在代码 6-11 中, 该记录类型保存了文件名和文件类型信息。实际上, 该记录所定义的成员与列表视图的行视图所定义的数据显示组件对应。

6.4 应用程序文件

除了资源文件, **Android** 平台还为应用程序提供了私有文件和共享首选项两种数据存储



方式；通过这两种方式，应用程序可以方便地加载和保存其状态或设置。

6.4.1 私有文件

顾名思义，私有文件为应用程序所私有，只有指定的应用程序对其拥有特定的读/写权限。图 6-6 是读/写私有文件的实机界面。

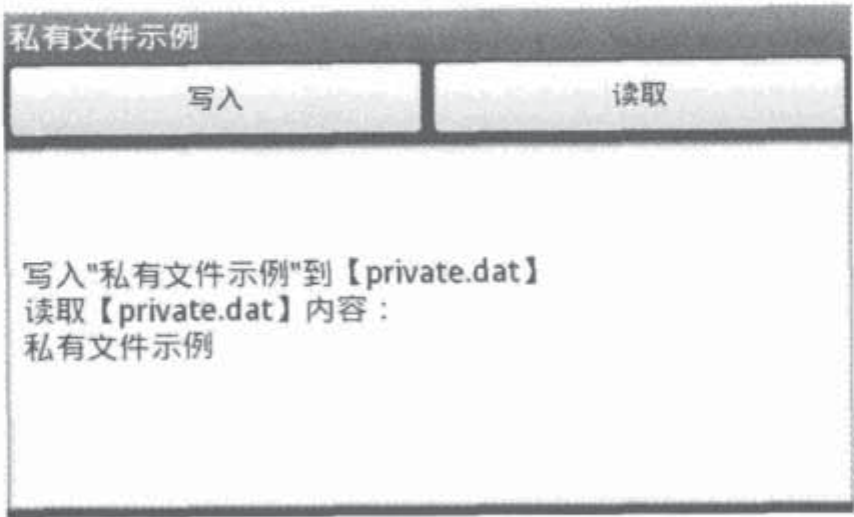


图 6-6 读/写私有文件的实机界面

1. 私有文件的创建及写入

代码 6-12 是创建私有文件并往其中写入内容的主要代码。

代码 6-12 写入内容到私有文件

文件名: ContextFileAct.java

```
1 private void doWrite() throws IOException { //写文件
2     //以私有的模式打开文件输出流
3     FileOutputStream fos = this.openFileOutput(IConfig.FNAME,
4                                             Context.MODE_PRIVATE);
5     PrintWriter pr = new PrintWriter(fos);
6     //写入内容
7     String contents = this.getResources().getString(R.string.app_name);
8     pr.println(contents);
9     print("写入\""+contents+"\"到【"+IConfig.FNAME+"】");
10    //关闭文件流
11    pr.close();
12    fos.close();
13 }
```

在代码 6-12 中，首先使用 Activity 组件的 openFileOutput 方法打开应用程序到私有文件的输出流（第 3 行），继而通过输出流接口对私有文件进行写操作。

当目标文件不存在时，openFileOutput 方法会自动创建指定文件，该方法有两个参数：第一个参数是私有文件名，不包含路径和路径分割符；第二个参数是访问模式。表 6-2 是对私有文件访问模式的说明。

表 6-2 对私有文件访问模式的说明

访问模式	说 明
MODE_APPEND	添加到已有文件
MODE_PRIVATE	私有模式（默认模式），只能被该应用程序存取
MODE_WORLD_READABLE	全局可读，其他应用程序也可以读取
MODE_WORLD_WRITEABLE	全局可写，其他应用程序也可以写入

私有文件会被创建在应用程序的数据文件夹中。

2. 私有文件的读取

代码 6-13 是读取私有文件的主要代码。

代码 6-13 读取私有文件

文件名: ContextFileAct.java

```
1 private void doRead() throws IOException { //读取文件
2     //打开文件输入流
3     FileInputStream fis = this.openFileInput(IConfig.FNAME);
4     BufferedReader br = new BufferedReader(new InputStreamReader(fis));
5     String line = null;
6     print("读取【" + IConfig.FNAME+"】内容: ");
7     //按行读取文件
8     while((line=br.readLine()) != null) {
9         print(line);
10    }
11    //关闭文件流
12    br.close();
13    fis.close();
14 }
```

在代码 6-13 中，首先通过 Activity 组件的 openFileInput 方法开打私有文件到应用程序的输入流（第 3 行），继而通过该输入流接口对私有文件进行读取。

提示：Android 平台对私有文件的访问控制与 J2ME 平台中的 RMS 文件有点相似。J2ME 平台中的 RMS 文件默认地只能被创建它的程序所访问，除非在创建时被设置成全局访问的标志，这样才能被其他程序所访问。

另外，有别于存储设备文件读/写，私有文件的读/写不必设置许可。

6.4.2 共享首选项文件

私有文件用于存储原数据（同原文件资源的内容形式），而共享首选项文件用于存放首选项。在共享首选项文件中，包含一些按照“键—值”的形式进行存储的条目。图 6-7 是读取首选项文件的实机界面。



图 6-7 读取首选项文件的实机界面

1. 首选项文件的创建和写入

代码 6-14 是创建首选项文件并向其添加选项条目的关键代码。

代码 6-14 创建首选项文件并添加选项条目

文件名: PrefsAct.java

```

1  private void doStore() { //存储首选项
2      //获取共享首选项接口
3      SharedPreferences sp = this.getSharedPreferences(IConfig.PREFS,
4                                                         Activity.MODE_PRIVATE);
5      //获取共享首选项编辑接口
6      SharedPreferences.Editor editor = sp.edit();
7      //添加设置
8      editor.putString(IConfig.KEY, this.getResources().getString(R.string.app_name));
9      //提交修改
10     editor.commit();
11 }

```

在代码 6-14 中, 首先通过 Activity 组件的 `getSharedPreferences` 方法获取与应用程序相关的共享选项接口 (第 3 行), 然后再使用共享选项接口的 `edit` 方法获取数据编辑接口 (第 6 行), 然后使用该编辑接口的 `putString` 方法实现字符串类型的选项设置的添加 (第 8 行), 最后通过编辑接口的 `commit` 方法提交添加 (第 10 行)。

当目标文件不存在时, `getSharedPreferences` 方法会自动创建指定文件, 该方法有两个参数: 第一个参数是首选项文件名, 不包含路径和路径分割符; 第二个参数是访问模式, 其模式与私有文件的相同。

首选项文件会被创建在应用程序的数据文件夹中。

2. 首选项文件的读取

代码 6-15 是读取首选项文件的关键代码。

代码 6-15 读取首选项文件

文件名: PrefsAct.java

```

1  private void doRead() { //读取首选项
2      //获取共享首选项接口

```



```

3      SharedPreferences sp = getSharedPreferences(IConfig.PREFS,
4                                                  Activity.MODE_PRIVATE);
5      //获取设置
6      String setting1 = sp.getString(IConfig.KEY, "");
7      Toast.makeText(this, setting1, Toast.LENGTH_LONG).show();
8  }

```

在代码 6-15 中，首先通过 Activity 组件的 `getSharedPreferences` 方法获取与应用程序相关的共享首选项接口（第 3 行），然后通过该接口的 `getString` 方法获取指定选项的字符串值（类似于 Bundle 容器的用法）。

6.5 文件系统监视

Android 系统提供了文件观察员类（FileObserver）用于监视指定文件或文件夹的操作。图 6-8 是文件系统监视程序的实机界面，该程序可以对 SD 卡上的文件操作进行监视。

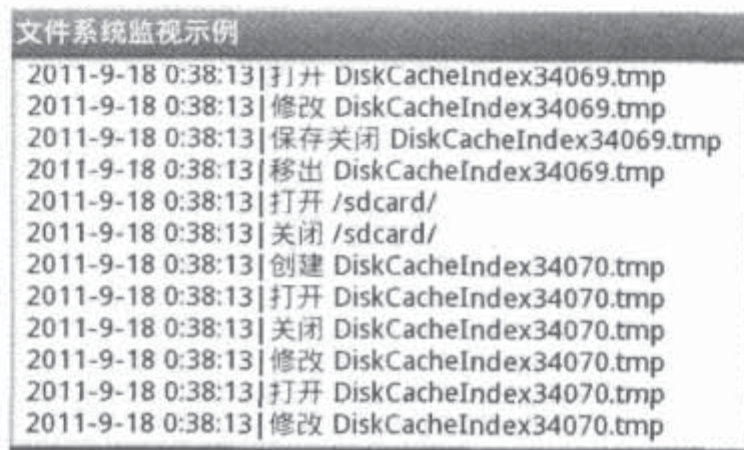


图 6-8 文件系统监视程序的实机界面

1. 启动文件系统监视

代码 6-16 是启动文件系统监视的关键代码。

代码 6-16 启动文件系统监视

文件名: FileObserverAct.java

```

1  public class FileObserverAct extends Activity {
2      //观察者
3      private FooFileObserver mObserver = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //初始化设置
10         mObserver = new FooFileObserver(IConfig.DEST);
11         mObserver.setHandler(new Handler() {
12             @Override

```



```
13         public void handleMessage(Message msg) {
14             Bundle bundle = msg.getData();
15             String extras = bundle.getString(IConfig.EXTRA);
16             print(FooSysUtil.getInstance().getTsp()+"|"+extras);
17         }
18     });
19     mObserver.startWatching();
20 }
21 @Override
22 protected void onDestroy() { super.onDestroy();
23     //停止监视
24     mObserver.stopWatching();
25 }
26 };
```

在代码 6-16 中，首先创建了一个定制的文件观察者（第 10 行），然后设置该观察者的消息队列处理器（第 11 行），最后使用观察者的 `startWatching` 方法启动监视（第 19 行）。

创建文件系统观察者需要两个参数：第一个参数是目标文件或文件夹的路径；第二个参数是所监视的事件类型掩码。代码 6-16 的第 5 行采用默认的事件类型掩码，监视所有事件。表 6-3 是文件监视事件类型掩码的说明。

表 6-3 文件监视事件类型掩码的说明

事件类型掩码	说 明
ACCESS	读取
ALL_EVENTS	所有事件
ATTRIB	元数据（权限、所有者、时间戳等）被修改
CLOSE_NOWRITE	关闭未写入
CLOSE_WRITE	关闭并写入
CREATE	创建
DELETE	删除
DELETE_SELF	所监视的文件或文件夹被删除，监视停止
MODIFY	修改
MOVED_FROM	从所监视的文件夹中移出
MOVED_TO	移动到所监视的文件夹中
OPEN	打开

当文件系统观察者启动后，即可对发生于目标文件夹中的文件操作行为进行监视。文件系统观察者监测到的消息通过消息队列处理器接口（Handler）发送给主线程，主线程对事件信息打印输出（第 16 行）。

使用文件观察者的 `stopWatching` 方法可以停止监视行为（第 24 行）。

2. 文件观察者

代码 6-17 是文件系统观察者的定义，该观察者继承于文件观察者抽象类。

代码 6-17 文件系统观察者的定义

文件名: FooFileObserver.java

```
1  public class FooFileObserver extends FileObserver {
2      //主线程消息队列处理器
3      private Handler mHandler = null;
4
5      public FooFileObserver(String path) { super(path); }
6      public void setHandler(Handler handler) { this.mHandler = handler; }
7
8      @Override
9      public void onEvent(int event, String path) {
10         //获取事件类型
11         int type = (event&FileObserver.ALL_EVENTS);
12         String realPath = (path==null)?IConfig.DEST:path;
13         String action = null;
14
15         switch(type) { //分派文件观察事件
16             case FileObserver.CREATE: { action = "创建"; break; }
17             case FileObserver.DELETE: { action = "删除"; break; }
18             case FileObserver.MODIFY: { action = "修改"; break; }
19             case FileObserver.OPEN: { action = "打开"; break; }
20             case FileObserver.CLOSE_WRITE: { action = "保存关闭"; break; }
21             case FileObserver.CLOSE_NOWRITE: { action = "关闭"; break; }
22             case FileObserver.MOVED_FROM: { action = "移出"; break; }
23             case FileObserver.MOVED_TO: { action = "移进"; break; }
24             case FileObserver.ATTRIB: { action = "属性修改"; break; }
25             default: { action = "Event: " + event; break; }
26         }
27         //发送消息给主线程消息队列
28         FooSysUtil.getInstance().postMsg(mHandler, IConfig.EXTRA,
29             action + " " + realPath);
30     }
31 };
```

在代码 6-17 中, 第 9 行的 `onEvent` 方法就是用于监视文件系统事件的回调方法。该方法包含两个参数: 第一个参数是所发生事件的类型标志; 第二个参数是事件相关的文件路径。

通过事件类型标识, 开发者可以获知对应的事件描述 (第 16~24 行)。需要注意的是, 对 `onEvent` 方法中的事件类型标志还需要与所有事件掩码 (第 11 行) 进行“与”操作, 才能得到文件观察者类所定义的监视事件类型代码。

第 7 章 数据库应用

本章对 Android 平台支持的数据库以及数据库应用模式进行全面介绍。数据库类型包括两款嵌入式数据库：SQLite 和 Db4o 数据库。SQLite 数据库应用模式有 3 种：内容提供框架、游标加载器和 SQLite 数据库 API。通过对两种数据库版本的记账工具的开发过程说明，期望读者在掌握数据库使用的同时，领悟各种数据库的适用性。

7.1 Android 平台数据库应用概述

从技术特征而言，Android 平台中的数据库应用基于嵌入式数据库（SQLite）；从应用模式而言，Android 平台中的数据库用于两个层面：一是作为系统内容提供机制中的持久层；二是作为定制应用中的持久层。

简而言之，Android 平台所使用的数据库是 SQLite 嵌入式数据库，其一般使用内容提供框架向应用程序提供数据存取接口，如联系信息数据库使用内容提供框架向通信录工具提供联系信息的管理；应用程序也可不使用框架而直接使用数据库 API 进行开发。该应用架构如图 7-1 所示。

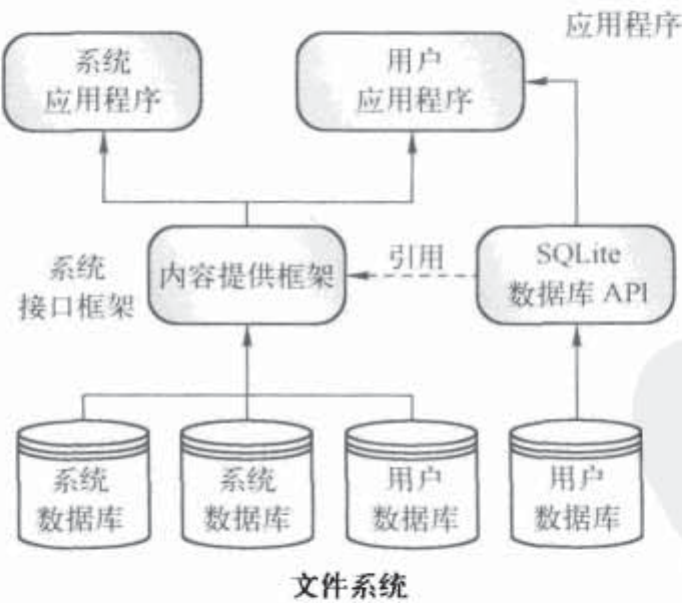


图 7-1 Android 平台数据库应用架构

在图 7-1 中，内容提供框架需要 SQLite 数据库 API 的支撑，框架底层将数据库的存取方法进行封装，以简化开发过程。Android 平台新增了数据加载机制来进行数据内容的异步加载（获取），其数据访问机制基于内容提供框架。

实际上，除了 SQLite 数据库，Android 平台还支持一款嵌入式对象数据库：Db4o。

7.2 嵌入式数据库 SQLite

7.2.1 SQLite 数据库介绍

顾名思义，SQLite 数据库是指一款精简（Lite）的 SQL 工具。SQLite 数据库的设计目标正是对系统资源的占用存在严格制约的嵌入式系统，已用于很多嵌入式产品中。除了占用较小的系统资源外，SQLite 数据库对 SQL 规范的支持很全面，它支持大多数 SQL 标准，摒弃了其中一些复杂功能（如左连接等），同时添加了一些自身的特性。

表 7-1 是 SQLite 数据库所支持的 SQL 语句类型。

表 7-1 SQLite 数据库提供的 SQL 支持

SQL 语句	说 明
CREATE/DROP TABLE	创建/删除数据表
CREATE/DROP VIEW	创建/删除视图
CREATE/DROP INDEX	创建/删除索引
CREATE/DROP TRIGGER	创建/删除触发器
INSERT / DELETE / UPDATE / SELECT	记录的增/删/改/查
BEGIN TRANSACTION	开始事务
END TRANSACTION	结束事务
COMMIT TRANSACTION	提交事务
ROLLBACK TRANSACTION	回滚事务
RELEASE SAVEPOINT	释放保存点
SAVEPOINT	创建保存点

通过表 7-1，读者应该可看出 SQLite 数据库对事务处理的支持比较完整，不仅包括事务的提交和回滚，还包括对保存点的创建和释放。而实际上，SQLite 数据库也是一款遵守 ACID 要求的关系型数据库引擎，其官方网站为 <http://www.sqlite.org/>。

表 7-2 是 SQLite 数据库（第 3 版）所定义的数据类型的说明。

表 7-2 SQLite 数据库所定义的数据类型的说明

类 型	说 明
NULL	空值
INTEGER	有符号整数
REAL	浮点数
TEXT	文本字符串
BLOB	数据块

与此同时，SQLite 数据库也支持 SQL 标准类型，如 VARCHAR、CHAR、BIGINT 等。此外，SQLite 数据库定义了 SQL 标准所约定的聚合函数、日期时间函数等，同时也定义了一套自身的功能函数（如获取系统版本）。



提示：ACID 是 Atomicity（原子性，即支持事务处理，每一个事务都是原子操作，不能再被切分）、Consistency（一致性，事务开始前后，数据库的完整性约束一致）、Isolation（隔离性，事务之间是隔离的，互不干扰）和 Durability（持久性，事务完成后，操作结果将持久地保存于数据库中）的简称。

7.2.2 Android 平台对 SQLite 数据库的支持

Android 平台提供了数据库（android.database）和 SQLite（android.database.sqlite）这两个包用于 SQLite 数据库应用。表 7-3 是数据库包中重要的类或接口的说明。

表 7-3 数据库包中重要的类/接口的说明

类/接口	说 明
Cursor	结果集游标
DatabaseUtils	处理数据库和游标的工具类
SQLException	异常定义接口

表 7-4 是 SQLite 包中重要的类或接口的说明。

表 7-4 SQLite 包中重要的类/接口的说明

类/接口	说 明
SQLiteDatabase	用于管理 SQLite 数据库的接口
SQLiteQueryBuilder	查询构建类
SQLiteOpenHelper	用于管理数据库创建和版本的工具类

7.3 SQLite 数据库应用模式

通过图 7-1 可知，Android 平台中的 SQLite 数据库应用有两种模式：一是使用内容提供框架向应用程序提供内容服务；二是直接使用 SQLite 数据库 API 进行数据存取。

实际上，内容提供框架是整个 Android 平台进行数据存取的基本框架，在系统数据文件夹（/data）中，其子文件夹 data 是所有应用程序的数据内容，各应用程序的数据内容以包名为子文件夹进行存放。图 7-2 所示为使用 adb 工具连接到虚拟设备后，列举数据文件夹中的内容。

```
root@android:/data/data # ls
ls
android.tts
com.android.browser
com.android.calculator2
com.android.camera
com.android.certinstaller
com.android.contacts
```

图 7-2 系统数据文件夹中的内容

在这些应用程序文件夹中，可以发现很多 SQLite 数据库文件（db 文件）。图 7-3 所示为列举浏览器应用程序的数据库文件夹中的内容。

```
root@android:/data/data # ls com.android.browser/databases
ls com.android.browser/databases
autofill.db
browser2.db
browser2.db-shm
browser2.db-wal
webview.db
```

图 7-3 系统数据库文件夹中的内容

在内容提供框架中，Activity 组件不直接进行数据操作，而是使用其内容解析器（Content Resolver）依据数据内容提供者（Content Provider）所定义的数据资源标识（URI）来存取数据，具体操作（增/删/改/查）在内容提供者中进行定义，内容解析器只需调用对应的接口即可实现数据的操作；对于使用数据库 API 的方式，应用程序组件使用 API 直接操纵数据库。其中，对于查询操作，无论使用内容提供框架还是数据库 API，其返回的内容都是记录游标，通过游标操作可以遍历记录内容，继而将内容通过可视组件展示。这两种应用模式的示意图如图 7-4 所示。



图 7-4 数据库应用模式示意图

在图 7-4 中，游标加载器（CursorLoader）的应用是 Android 平台的新特性，其数据访问基于内容提供框架。游标加载器只需提供数据内容的资源标识即可获取记录游标。

7.4 内容提供框架

7.4.1 内容解析端

在图 7-5 所示的实机界面中，该应用程序使用内容提供框架获取指定数据库中的记录，



并以列表的形式进行显示。

内容提供框架示例		
行号	Android版本	版本代码
1	1.5	Cupcake
2	1.6	Donut
3	2.1	Éclair
4	2.2	Froyo
5	2.3	Gingerbread
6	3.x	Honeycomb

图 7-5 使用内容提供框架获取记录的实机界面

1. 应用程序 Activity 组件框架

代码 7-1 是使用内容提供框架获取数据表内容的 Activity 组件的框架定义。

代码 7-1 数据加载 Activity 组件的框架定义

文件名: ContentsDemoAct.java

```
1 public class ContentsDemoAct extends Activity {
2     //布局容器和内容解析器
3     private LinearLayout mLayout = null;
4     private ContentResolver mResolver = null;
5
6     @Override
7     public void onCreate(Bundle savedInstanceState) {
8         super.onCreate(savedInstanceState);
9         setContentView(R.layout.main);
10        //获取布局容器对象
11        this.mLayout = (LinearLayout) findViewById(R.id.layoutPanel);
12        this.mLayout.setOrientation(LinearLayout.VERTICAL);
13        //获取内容解析器
14        this.mResolver = this.getContentResolver();
15        //初始化
16        init();
17    }
18
19    private void init() { //初始化数据显示
20        Cursor c = mResolver.query(IDataSpec.CONTENT_URI, IDataSpec.COLS_MINI,
21                                   null, null, null);
22        //使用表格视图显示数据内容
```



```

23         final double colWidths[] = { 0.2f, 0.4f, 0.4f};
24         FooTableWrapper wrapper = new FooTableWrapper(this, c, colWidths);
25         //初始化列抬头映射
26         initColumnTitleMapping(wrapper);
27         this.mLayout.addView(wrapper.getMainWidget());
28         //关闭游标
29         c.close();
30     }
31     .....
32 };

```

在代码 7-1 中，首先获取与 Activity 组件关联的内容解析器（第 14 行），然后使用内容解析器的 query 方法按照给定的 URI 获取记录游标（第 20 行），继而使用表格视图显示数据内容（第 24 行），最后关闭游标（第 29 行）。

2. 数据规格定义

代码 7-2 是代码 7-1 中所参考的数据内容规格的定义，主要包括内容资源的标识以及数据表的列顺序和列名等信息。

代码 7-2 数据内容规格的定义

文件名: IDataSpec.java

```

1  public interface IDataSpec {
2      //定义列号
3      public static final int COL_NO_ID = 0;
4      public static final int COL_NO_VER = 1;
5      public static final int COL_NO_CODE = 2;
6      public static final int COL_NO_LOGO = 3;
7      //定义列名
8      public static final String COL_ID = "_id";
9      public static final String COL_VER = "version";
10     public static final String COL_CODE = "code";
11     public static final String COL_LOGO = "logo";
12     //列名数组
13     public static final String[] COLS_MINI = { COL_ID, COL_VER, COL_CODE };
14     public static final String[] COLS = { COL_ID, COL_VER, COL_CODE, COL_LOGO };
15     //内容 URI
16     public static final Uri CONTENT_URI =
17         Uri.parse("content://foolstudio.demo.provider.FooTblProvider");
18     public static final String URI_AUTHORITY =
19         "foolstudio.demo.provider.FooTblProvider";
20 };

```

在代码 7-2 中，内容 URI 的模式名为 content，对于此类资源，Android 平台会在已注册的内容提供者列表中依据包及类名来查找相应的内容提供者。



7.4.2 内容提供端

1. 内容提供者框架

代码 7-3 是代码 7-2 中所引用的内容提供者的框架定义。

代码 7-3 内容提供者的框架定义

文件名: FooTblProvider.java

```

1  public class FooTblProvider extends ContentProvider {
2      //数据表
3      private static final String TBL_NAME = "tab_codes";
4      //URI 匹配器
5      private static final UriMatcher mUriMatcher;
6      private FooDbHelper mDbHelper = null;
7
8      static {      mUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
9                  mUriMatcher.addURI(IConfig.URI_AUTHORITY, null, 0);
10     }
11
12     @Override
13     public boolean onCreate() { mDbHelper = new FooDbHelper(this.getContext());
14         return (true);
15     }
16     .....
17 };

```

在代码 7-3 中,该内容提供者类继承于父类:内容提供者,其包含两个主要的成员,即资源标识匹配器和数据库助手(第 5 行和第 6 行)。

资源标识匹配器主要用于匹配内容解析端所提供的资源标识(是否合法及请求类型);数据库助手主要用于打开数据库,提供数据连接。

2. 记录操作接口

代码 7-4 是定制内容提供者重载的记录操作接口,包括查询、删除、插入和更新。

代码 7-4 记录操作接口

文件名: FooTblProvider.java

```

1  @Override
2  public Cursor query(Uri uri, String[] prj, String sel, String[] args, String sort) {
3      SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
4      builder.setTables(TBL_NAME);
5      //获取数据库句柄
6      SQLiteDatabase db = this.mDbHelper.getReadableDatabase();
7      Cursor c = builder.query(db, prj, sel, args, null, null, sort);
8      c.setNotificationUri(this.getContext().getContentResolver(), uri);
9      //返回记录游标
10     return (c);

```

```

11 }
12 @Override
13 public int delete(Uri uri, String where, String[] whereArgs) {
14     SQLiteDatabase db = this.mDbHelper.getWritableDatabase();
15     int rowCount = 0;
16     rowCount = db.delete(TBL_NAME, where, whereArgs);
17     return (rowCount);
18 }
19 @Override
20 public String getType(Uri uri) {
21     switch (mUriMatcher.match(uri)) {
22         case IConfig.COL_NO_ID: { return (IConfig.COL_ID); }
23         case IConfig.COL_NO_VER: { return (IConfig.COL_VER); }
24         case IConfig.COL_NO_CODE: { return (IConfig.COL_CODE); }
25         case IConfig.COL_NO_LOGO: { return (IConfig.COL_LOGO); }
26         default: { throw new IllegalArgumentException("无效 URI " + uri); }
27     }
28 }
29 @Override
30 public Uri insert(Uri uri, ContentValues initialValues) {
31     SQLiteDatabase db = this.mDbHelper.getWritableDatabase();
32     long rowId = -1;
33     //插入记录
34     rowId = db.insert(TBL_NAME, null, initialValues);
35     //返回已插入记录的 URI
36     Uri noteUri = ContentUris.withAppendedId(IConfig.CONTENT_URI, rowId);
37     getContext().getContentResolver().notifyChange(noteUri, null);
38     return noteUri;
39 }
40 @Override
41 public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {
42     SQLiteDatabase db = this.mDbHelper.getWritableDatabase();
43     int rowCount = 0;
44     rowCount = db.update(TBL_NAME, values, where, whereArgs);
45     return (rowCount);
46 }

```

在代码 7-4 中，数据库的操作接口实现中都使用到了数据库打开助手，用于获取可读（查询）或可写（删除、插入和更新）的数据库连接。

提示：由于涉及数据库的写入操作，所以可能需要有写入 SD 卡的许可，即必须在应用程序清单文件中添加该使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```




3. 数据库打开助手

代码 7-5 是代码 7-3 中所引用的数据库打开助手（第 13 行）的定义。

代码 7-5 数据库打开助手的定义

文件名: FooDbHelper.java

```
1 public class FooDbHelper extends SQLiteOpenHelper {
2     //数据库文件路径
3     private static String DB_PATH = "/sdcard/files/android.db";
4     private static int DB_VER = 1;
5
6     public FooDbHelper(Context ctx) { super(ctx, DB_PATH, null, DB_VER); }
7
8     @Override
9     public void onCreate(SQLiteDatabase db) {}
10    @Override
11    public void onUpgrade(SQLiteDatabase db, int oldVer, int newVer) {}
12 }
```

在代码 7-5 中，该数据库打开助手类继承于父类 SQLiteOpenHelper，其主要重载了构造函数，在构造函数中指定数据库路径（第 6 行）。

至此，读者应该可以大致了解 Android 平台的内容提供框架的机制：内容解析器使用数据操作接口依据内容 URI 向内容提供者发送请求；内容提供者使用 URI 匹配器对请求的 URI 进行匹配，并使用数据库打开助手对数据库进行对应的操作，最后将执行结果返回给内容解析器，如图 7-6 所示。

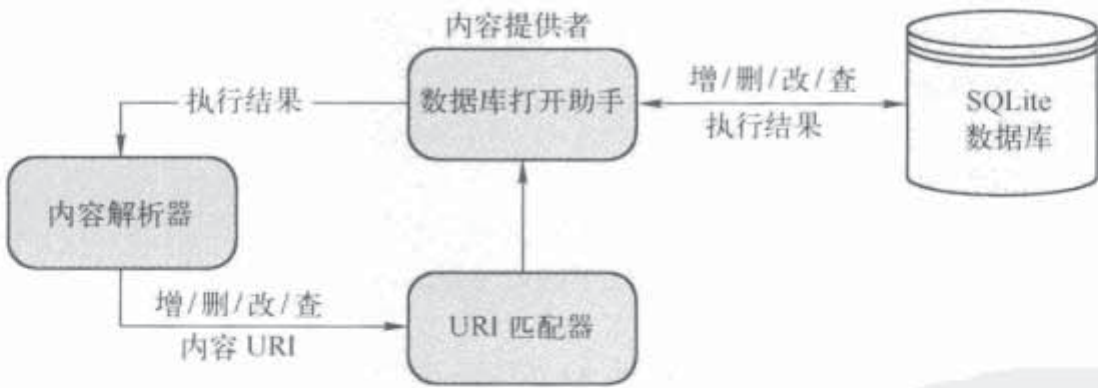


图 7-6 内容提供框架的机制

4. 声明内容提供者

内容解析器只需参照数据内容的 URI 即可建立与目标内容提供者的连接，其间需先对目标内容提供者进行定位。该定位过程由 Android 平台自动完成，其前提是目标内容提供者必须在程序清单中进行“登记”。代码 7-6 是在程序清单中声明数据提供者的 XML 代码。

代码 7-6 声明数据提供者

文件名: AndroidManifest.xml

```
1 <provider android:name="foolstudio.demo.provider.FooTblProvider"
2           android:authorities="foolstudio.demo.provider.FooTblProvider"
3           android:label="@string/provider"/>
```


7.4.3 游标加载器应用

在图 7-7 所示的实机界面中，该应用程序使用游标加载器加载指定内容提供者中的记录，并以列表的形式进行显示。



图 7-7 使用游标加载器加载数据记录的实机界面

1. 数据加载片段组件

代码 7-7 是使用游标加载器获取数据表内容的片段组件的定义。

代码 7-7 数据加载片段组件的定义

文件名: FooListFmt.java

```

1  public class FooListFmt extends ListFragment implements LoaderCallbacks<Cursor> {
2      //简单游标适配器（用于列表控件的适配器）
3      private ItemClickHandler mHandler = null;
4
5      @Override //宿主 Activity 创建完毕后回调
6      public void onActivityCreated(Bundle savedInstanceState) {
7          super.onActivityCreated(savedInstanceState);
8          //创建适配器
9          SimpleCursorAdapter adapter = new SimpleCursorAdapter(this.getActivity(),
10              android.R.layout.simple_list_item_2, null,
11              new String[] { IConfig.COL_VER, IConfig.COL_CODE },
12              new int[] { android.R.id.text1, android.R.id.text2 }, 0);
13          //设置适配器
14          this.setListAdapter(adapter);
15          //初始化加载器
16          getLoaderManager().initLoader(0, null, this);
17      }
18      @Override //创建视图时回调
19      public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle b) {

```



Android 平台开发之旅 第2版

```

20         return inflater.inflate(R.layout.list_fmt, container, false);
21     }
22     @Override //绑定视图时回调
23     public void onAttach(Activity act) { super.onAttach(act);
24         try { mHandler = (ItemClickListener)act;
25             } catch(ClassCastException e) { e.printStackTrace(); }
26     }
27
28     //创建载入器时回调
29     public Loader<Cursor> onCreateLoader(int id, Bundle args) {
30         Toast.makeText(this.getActivity(), "开始加载...", Toast.LENGTH_LONG).show();
31         //返回游标加载器
32         return new CursorLoader(this.getActivity(), IConfig.CONTENT_URI,
33                                 IConfig.COLS, null,null,null);
34     }
35     //载入完毕后回调
36     public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
37         //交换游标
38         ((SimpleCursorAdapter)this.getListAdapter()).swapCursor(c);
39         Toast.makeText(this.getActivity(), "载入完毕! ", Toast.LENGTH_SHORT). show();
40         //定位游标
41         c.moveToPosition(0);
42         //显示 Logo 图片
43         showLogo(c);
44     }
45     //载入器重置时回调
46     public void onLoaderReset(Loader<Cursor> loader) {
47         ((SimpleCursorAdapter)this.getListAdapter()).swapCursor(null);
48     }
49
50     @Override
51     public void onItemClick(ListView l, View v, int pos, long id) {
52         //定位游标
53         Cursor c = ((SimpleCursorAdapter)this.getListAdapter()).getCursor();
54         c.moveToPosition(pos);
55         //显示 Logo 图片
56         showLogo(c);
57     }
58
59     private void showLogo(Cursor c) { //显示 Logo 图片
60         //读取 logo 图片数据
61         byte[] data = c.getBlob(IConfig.COL_NO_LOGO);
62         //生成位图
63         Bitmap bmp = BitmapFactory.decodeByteArray(data, 0, data.length);
64         //回调 Activity 的方法
65         mHandler.handleClick(bmp);

```



```
66      }  
67  };
```

在代码 7-7 中，使用简单游标适配器作为列表视图的适配器（第 9 行），而该适配器与数据加载结果（记录游标）存在对接（第 38 行），从而实现通过数据加载器获取列表视图数据。数据加载通过加载管理器来进行调度（第 16 行），所管理的游标加载器按照内容提供者的资源标识获取数据内容（第 32 行）。当加载完毕后即可将包含数据内容的记录游标“连接”到列表视图的适配器（第 38 行），用于提供列表项内容。

简单游标适配器的记录项是记录游标，当选择列表项时，可通过列表项位置移动记录游标位置（第 54 行），再通过游标的读取方法即可获取列内容（第 61 行）。

2. 加载管理

加载管理就是对加载器的管理，Activity 或片段组件通过其关联的加载管理器即可对加载器进行管理。加载器通过框架所定义的回调方法来实现异步加载（代码 7-7 中第 1 行所实现的加载管理器回调接口）。表 7-5 是对加载器回调方法的说明。

表 7-5 加载器回调方法的说明

回调方法	说 明
onCreateLoader	当创建加载器时回调，用于加载器的定义
onLoadFinished	当加载完毕时回调
onLoaderReset	当加载器重置时回调

(1) 获取加载管理器

使用 Activity 组件或片段的 `getLoaderManager` 方法即可获得关联的加载管理器（代码 7-7 中第 16 行）。

(2) 初始化加载器

使用加载管理器的 `initLoader` 方法以“预约”加载器的初始化（第 16 行）；而实际上加载器的构造是在加载器创建回调方法中（第 32 行），在构造加载器的过程中，指明了加载目标的 URI 及查询约束，加载器即可启动加载过程。

(3) 加载完成处理

完成加载的处理在加载完成回调方法中进行（第 36 行），回调框架将加载所得的结果集以游标的方式提供给调用方 Activity 或片段，再由 Activity 或片段对数据进行展示。

7.5 SQLite 数据库 API

在对内容提供框架的机制中已经提到，内容提供者的底层操作是通过调用 SQLite 数据库 API 来实现的。可以说，SQLite 数据库 API 是整个 Android 平台进行数据库应用的基石。

7.5.1 SQLite 数据库应用

SQLite 数据库应用可分为 3 个层面：数据库管理、表模式管理和数据记录管理。



1. 数据库管理

SQLite 数据库是基于文件的数据库系统，所以对数据库的管理可视为对文件的管理，如库的删除、移动或复制等。

注意：创建库文件需要在 SD 卡上创建文件或文件夹，所以需要在工程清单文件中声明允许写外部存储器（SD 卡）的许可，其声明代码如下所示。

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

2. 表模式管理

表模式主要包括数据表的列定义、存储信息和本地化设置。SQLite 数据库 API 并没有提供管理表模式相关的接口，但 SQLite 数据库系统会在系统表中记录当前数据库中所有数据表的模式信息。表 7-6 是 SQLite 数据库系统表【sqlite_master】的模式信息。

表 7-6 系统表【sqlite_master】的模式信息

序 号	列 名	类 型	说 明
1	type	TEXT	对象类型，如表的类型是 table；索引的类型是 index
2	name	TEXT	对象名称，如表名或索引名
3	tbl_name	TEXT	该对象关联的表名
4	rootpage	INTEGER	根的页索引
5	sql	TEXT	对应的 SQL 定义（DDL）

通过系统表【sqlite_master】可以获取当前数据库中表对象列表，由此判断指定表是否存在，并通过其 sql 可以分解表的列信息。

3. 数据记录管理

数据记录的管理主要包括记录的增/删/改/查等操作。

4. SQLite 数据库工具类

为了统一对数据库的应用，作者将 SQLite 数据库的常用操作封装成工具类。代码 7-8 是该工具类的定义，其中功能主要包括打开/关闭数据库、执行 SQL 语句、执行查询并返回结果集、游标移动、获取列内容等。

代码 7-8 SQLite 工具类的定义

文件名：SQLiteUtil.java

```
1 public class SQLiteUtil {
2     private static final String MASTER_TBL = "sqlite_master";
3     //单例接口
4     private static SQLiteUtil mInstance = new SQLiteUtil();
5     private SQLiteUtil() {}
6     public static SQLiteUtil getInstance() { return (mInstance); }
7     //打开数据库
8     private SQLiteDatabase openDB(String dbName) { File file = new File(dbName);
9         if(file.exists() == true) { //库文件存在则打开数据库，如果不存在则初始化
10             return (SQLiteDatabase.openDatabase(dbName, null,
```



```

11                                     QLiteDatabase.OPEN_READWRITE) );
12         } else { return(SQLiteDatabase.openOrCreateDatabase(dbName, null) ); }
13     }
14
15     private void closeDB(SQLiteDatabase db) { db.close(); } //关闭数据库
16
17     public boolean deleteDB(String dbName) { File file = new File(dbName);
18         //库文件存在则删除
19         if(file.exists() == true) { return (file.delete() ); }
20         return (false);
21     }
22
23     public void execQuery(String dbName, String sql) { //执行 SQL 语句（无返回）
24         SQLiteDatabase db = openDB(dbName);
25         db.execSQL(sql);
26         closeDB(db);
27     }
28
29     //执行 SQL 语句并返回结果集
30     public Cursor openQuery(String dbName, String tblName, String cond) {
31         SQLiteDatabase db = openDB(dbName);
32         Cursor c = db.query(tblName, null, cond, null, null, null, null);
33         //游标复位
34         c.moveToFirst();
35         //关闭文件
36         closeDB(db);
37         return (c);
38     }
39
40     //获取记录集行数
41     public int getRowCount(Cursor c) { return(c.getCount() ); }
42     //获取列数
43     public int getColsCount(Cursor c) { return(c.getColumnCount()); }
44     //获取列名
45     public String getColNameBy(Cursor c, int i) { return(c.getColumnName(i) ); }
46     //判断游标的位置
47     public boolean isBOF(Cursor c) { return(c.isBeforeFirst()); }
48     public boolean isEOF(Cursor cursor) { return(cursor.isAfterLast() ); }
49     //移动游标
50     public boolean moveNext(Cursor c) { return(c.moveToNext() ); }
51     //获取当前游标指定列的内容
52     public String getCol(Cursor c, int i) { return(c.getString(i) ); }
53     //关闭游标
54     public void closeQuery(Cursor c) { c.close(); }
55
56     public boolean exists(String dbName, String tblName) { //判断指定表是否存在

```



```
57      Cursor c = openQuery(dbName, MASTER_TBL, "(tbl_name='"+tblName+"'");
58      int recordCount = c.getCount();
59      c.close();
60      return(recordCount > 0);
61  }
62  };
```

在代码 7-8 中，数据库的打开和文件访问相同，也需要指明打开标志（第 11 行）。表 7-7 是对 SQLite 数据库打开标志的说明。

表 7-7 SQLite 数据库打开标志的说明

类 型	说 明
CREATE_IF_NECESSARY	设置如果有需要（如数据库文件不存在）是否创建数据库
NO_LOCALIZED_COLLATORS	不使用本地化校对
OPEN_READONLY	以只读模式打开
OPEN_READWRITE	以读/写方式打开

注意：在使用完 SQLite 数据库后，一定要记得及时关闭数据库；否则，将会收到存在内存泄露的警告信息，其代码如下所示。

```
ERROR/Database(752): Leak found
ERROR/Database(752): java.lang.IllegalStateException: /sdcard/files/JournalBook.db
SQLiteDatabase created and never closed
```

7.5.2 基于 SQLite 数据库的日记账工具

下面将以一款基于 SQLite 数据库的日记账工具为读者介绍 SQLite 数据库 API 的开发过程。图 7-8 是该日记账工具（SQLite 版）的界面，该应用程序使用菜单项提供有关功能，依次为数据库管理和记录管理（添加、删除、浏览和查询）。

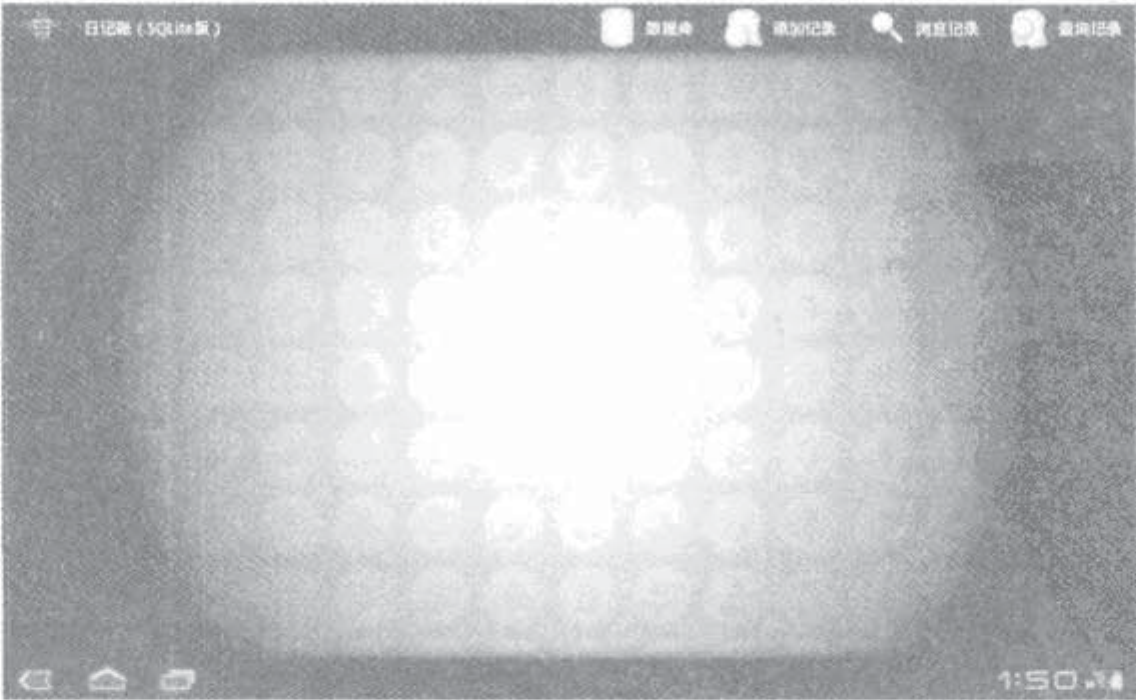


图 7-8 日记账工具（SQLite 版）的界面

1. 应用程序清单

代码 7-9 是该程序清单文件的内容。

代码 7-9 应用程序清单文件

文件名: AndroidManifest.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="foolstudio.demo.db.journalbook1"
4      android:versionCode="1"
5      android:versionName="1.0">
6      <uses-sdk android:minSdkVersion="12" />
7      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
8      <application android:icon="@drawable/icon" android:label="@string/app_name">
9          <activity android:name=".JournalBookAct" android:label="@string/app_name">
10             <intent-filter>
11                 <action android:name="android.intent.action.MAIN" />
12                 <category android:name="android.intent.category.LAUNCHER" />
13             </intent-filter>
14         </activity>
15         <activity android:name=".DBConfigAct" android:label="@string/act_cfg"/>
16         <activity android:name=".AppendRecAct" android:label="@string/act_add"/>
17         <activity android:name=".ReviewRecAct" android:label="@string/act_view"/>
18         <activity android:name=".LookupRecAct" android:label="@string/act_look"/>
19         <activity android:name=".ReportRecAct" android:label="@string/act_report"/>
20     </application>
21 </manifest>
```

在代码 7-9 中，除了主 Activity 组件外，应用程序还包含 5 个 Activity 组件（第 15~19 行）。图 7-9 是该日记账工具的组件关联示意图。



在图 7-9 中，主 Activity 与 4 个 Activity 组件之间是调用关系，通过调用子 Activity 组件对 SQLite 数据库进行相关操作，包括数据库管理、浏览记录、添加记录和查询记录。最终使用这些子 Activity 组件的界面对操作结果进行显示。



由于数据库的创建、删除以及记录的添加都是写入操作，需要应用程序具有写存储器的许可，所以在程序清单文件第 7 行中声明了写外部存储设备的使用许可。

2. 主界面定义

代码 7-10 是日记账工具主 Activity 组件的布局定义。

代码 7-10 主 Activity 组件的布局定义

文件名: main_view.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="match_parent" android:layout_height="match_parent">
5     <ImageView android:src="@drawable/login_bkg"
6         android:layout_width="match_parent" android:layout_height="match_parent"/>
7 </LinearLayout>
```

在代码 7-10 中，主布局中只放置了一张背景图片。代码 7-11 是应用程序选项菜单的定义。

代码 7-11 主 Activity 组件的选项菜单的定义

文件名: ops_menu.xml

```
1 <menu xmlns:android="http://schemas.android.com/apk/res/android">
2     <item android:title="数据库" android:icon="@drawable/db"
3         android:id="@+id/mi_cfg" android:showAsAction="ifRoom|withText"/>
4     <group android:checkableBehavior="single">
5         <item android:title="添加记录" android:icon="@drawable/append"
6             android:id="@+id/mi_add" android:showAsAction="ifRoom|withText"/>
7         <item android:title="浏览记录" android:icon="@drawable/preview"
8             android:id="@+id/mi_view" android:showAsAction="ifRoom|withText"/>
9         <item android:title="查询记录" android:icon="@drawable/lookup"
10            android:id="@+id/mi_look" android:showAsAction="ifRoom|withText"/>
11     </group>
12 </menu>
```

图 7-10 是该选项菜单的实机界面，前 4 项菜单将在操作栏中既显示图标，又显示文字。



图 7-10 选项菜单的实机界面

3. 应用程序主 Activity 框架

代码 7-12 是主 Activity 组件的框架定义。在该框架中，主 Activity 组件使用选项菜单项对子 Activity 组件进行调用。

代码 7-12 主 Activity 组件的框架定义

文件名: JournalBookAct.java

```

1  public class JournalBookAct extends Activity {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main_view);
6      }
7      @Override
8      public boolean onCreateOptionsMenu(Menu menu) { //创建选项菜单的回调方法
9          //填充菜单
10         this.getMenuInflater().inflate(R.menu.ops_menu, menu);
11         //设置菜单意向
12         menu.findItem(R.id.mi_cfg).setIntent(new Intent(this,DBConfigAct.class));
13         menu.findItem(R.id.mi_add).setIntent(new Intent(this,AppendRecAct.class));
14         menu.findItem(R.id.mi_view).setIntent(new Intent(this,ReviewRecAct.class));
15         menu.findItem(R.id.mi_look).setIntent(new Intent(this,LookupRecAct.class));
16         //注意: 必须要调用超类的方法, 否则无法实现意图回调
17         return (super.onCreateOptionsMenu(menu) );
18     }
19 };

```

在代码 7-12 中, 使用选项菜单项的 `setIntent` 方法将该菜单项与子 Activity 组件进行绑定 (第 12~15 行), 当选择其中的某一菜单项时则触发其所绑定的子 Activity 组件。

4. 程序配置信息接口

为了实现对配置信息的集中管理, 可以使用配置信息接口对程序中所有配置信息进行管理。代码 7-13 是该配置信息接口的定义。

代码 7-13 配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String DB_PATH = "/sdcard/files/JournalBook.db";
3      public static final String TBL_NAME = "Payout";
4      //数据项
5      public static final String EXTRA = "_rs";
6  };

```

5. 数据库管理 Activity

图 7-11 是数据库配置 Activity 组件的实机界面, 其中包含初始化和删除两个按钮。界面总体布局为垂直方向的线性布局。

代码 7-14 是数据库配置 Activity 组件的定义代码。



图 7-11 数据库配置的实现界面



代码 7-14 数据库配置 Activity 组件的定义

文件名: DBConfigAct.java

```

1  public class DBConfigAct extends Activity implements OnClickListener {
2      @Override
3      protected void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.config_view);
6          //获取按钮控件并设置单击事件侦听器
7          Button btnInit = (Button)findViewById(R.id.btn_init);
8          Button btnDrop = (Button)findViewById(R.id.btn_drop);
9          btnInit.setOnClickListener(this);
10         btnDrop.setOnClickListener(this);
11     }
12     @Override
13     public void onClick(View v) {
14         switch(v.getId()) {
15             case R.id.btn_init: { doInit(); break; } //初始化数据库
16             case R.id.btn_drop: { doDrop(); break; } //删除数据库
17         }
18     }
19
20     private void doInit() { //初始化数据库
21         if(SQLiteUtil.getInstance().exists(IConfig.DB_PATH,
22             IConfig.TBL_NAME) == false) {
23             String sql = "CREATE TABLE " + IConfig.TBL_NAME +
24                 "(timestamp TEXT primary key,comments TEXT,money REAL)";
25             //执行建表语句
26             SQLiteUtil.getInstance().execQuery(IConfig.DB_PATH, sql);
27             FoolUtil.getInstance().showMsg(this,"创建表成功!");
28         } else { FoolUtil.getInstance().showMsg(this, "表已存在!"); }
29     }
30
31     private void doDrop() { //删除数据库
32         if(SQLiteUtil.getInstance().deleteDB(IConfig.DB_PATH)) {
33             FoolUtil.getInstance().showMsg(this, "删除数据库成功!");
34         } else { FoolUtil.getInstance().showMsg(this, "数据库不存在!"); }
35     }
36 };

```

在代码 7-14 中，数据库的配置主要是初始化数据库（doInit）和删除数据库（doDrop），其中数据库的初始化主要是通过执行创建数据表的 SQL 语句来创建数据表（第 26 行）。

表 7-8 是所创建的记账数据表的模式信息。

表 7-8 日记账数据表的模式信息

序 号	列	列 类 型	描 述
1	timestamp	TEXT	事项发生时间
2	comments	TEXT	备注
3	money	REAL	金额

在数据库配置操作中都使用到了 SQLite 数据库工具类 (SQLiteUtil)，包括判断数据表是否存在 (第 21 行)、执行 SQL 语句 (第 26 行) 和删除数据库 (第 32 行)。

6. 添加记录 Activity

图 7-12 是在记录添加 Activity 组件中添加记录的实机界面，其中 3 个文本框用于接收用户输入；【确定】按钮用于提交添加，【取消】按钮用于返回到主 Activity 组件。界面总体布局为垂直方向的线性布局。

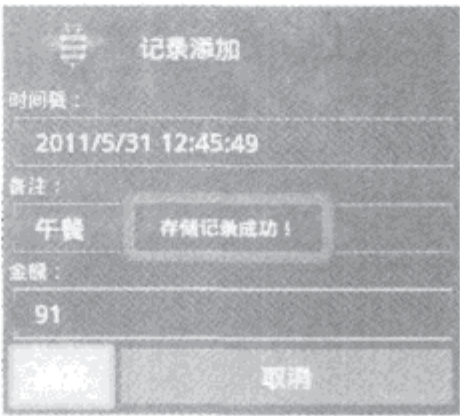


图 7-12 添加日记账记录的实机界面

代码 7-15 是记录添加 Activity 组件的定义。

代码 7-15 添加日记账记录的 Activity 组件的定义

文件名: AppendRecAct.java

```
1 public class AppendRecAct extends Activity implements OnClickListener {
2     private EditText mTxtTsp = null;
3     private EditText mTxtComments = null;
4     private EditText mTxtMoney = null;
5
6     @Override
7     public void onCreate(Bundle savedInstanceState) {
8         super.onCreate(savedInstanceState);
9         setContentView(R.layout.append_view);
10        //获取输入控制对象
11        mTxtTsp = (EditText)findViewById(R.id.etxt_tsp);
12        mTxtComments = (EditText)findViewById(R.id.etxt_comments);
13        mTxtMoney = (EditText)findViewById(R.id.etxt_money);
14        //设置按钮事件侦听
15        Button btnDiscard = (Button)findViewById(R.id.ebtn_discard);
16        Button btnSubmit = (Button)findViewById(R.id.ebtn_submit);
17        btnDiscard.setOnClickListener(this);
18        btnSubmit.setOnClickListener(this);
19        //初始化界面
20        mTxtTsp.setText(FooSysUtil.getInstance().getTsp());
21    }
22    @Override
23    public void onClick(View v) {
24        switch(v.getId()) {
25            case R.id.ebtn_discard: { this.finish(); break; } //取消添加
            case R.id.ebtn_submit: { doSubmit(); break; } //提交添加
```



```

26         }
27     }
28
29     private void doSubmit() { //添加记录
30         if(submitCheck() == false) { return; }
31         //拼凑插入语句
32         String sql = "INSERT INTO " + IConfig.TBL_NAME +
33             "(timestamp,comments,money) VALUES(" +
34             mTxtTsp.getText().toString().trim()+"','"+
35             mTxtComments.getText().toString().trim()+"','"+
36             mTxtMoney.getText().toString().trim()+")";
37         //执行插入语句
38         SQLiteUtil.getInstance().execQuery(IConfig.DB_PATH, sql);
39         FoolUtil.getInstance().showMsg(this, "存储记录成功!");
40         this.finish();
41     }
42
43     private boolean submitCheck() { //提交前检查
44         if(mTxtTsp.getText().toString().trim().length() < 1) {
45             FoolUtil.getInstance().showMsg(this, "时间戳不能为空!");
46             return (false);
47         } else if(mTxtComments.getText().toString().trim().length() < 1) {
48             FoolUtil.getInstance().showMsg(this, "备注不能为空!");
49             return (false);
50         } else if(mTxtMoney.getText().toString().trim().length() < 1) {
51             FoolUtil.getInstance().showMsg(this, "金额不能为空!");
52             return (false);
53         } else { double money = Double.parseDouble(mTxtMoney.getText().toString());
54             if(money==0.0D) { FoolUtil.getInstance().showMsg(this, "金额不能为零!");
55                 return (false);
56             }
57         }
58         return (true);
59     }
60 };

```

在代码 7-15 中，首先对用户的输入进行合法性检查（第 30 行），然后依据输入内容构建插入记录的 SQL 语句（INSERT INTO），继而使用 SQLite 工具类的 execQuery 方法来执行该语句，实现记录的插入（第 37 行）。

7. 记录浏览 Activity

图 7-13 是在记录浏览 Activity 组件中记录浏览的实机界面，其中 3 个文本框用于显示记录内容；【后一条】按钮用于向后导航，【前一条】按钮用于向前导航，【返回】按钮用于返回到主 Activity 组件。界面总体布局为垂直方向的线性布局。

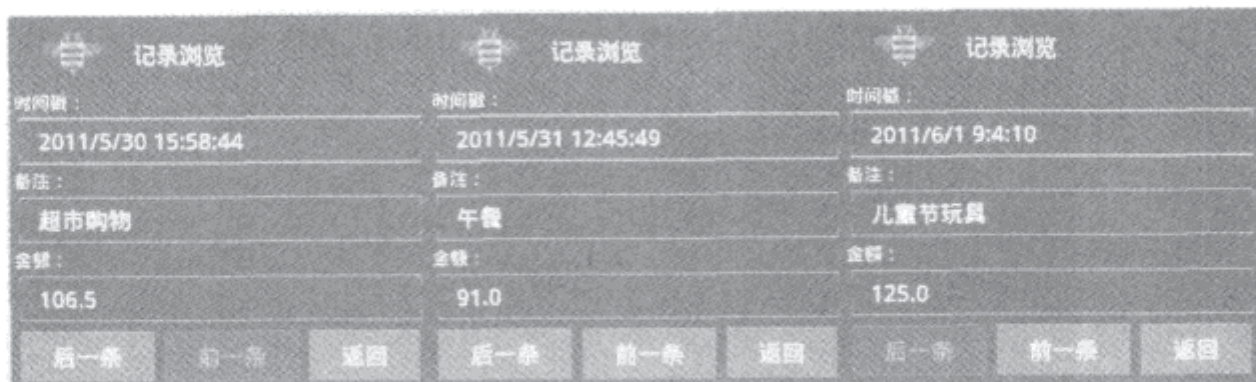


图 7-13 记录浏览的实机界面

代码 7-16 是记录浏览 Activity 组件的定义代码。

代码 7-16 记录浏览 Activity 组件的定义

文件名: ReviewRecAct.java

```

1  public class ReviewRecAct extends Activity implements OnClickListener {
2      //按钮
3      private Button mBtnNext = null;
4      private Button mBtnPrev = null;
5      //记录索引
6      private int mCurPos = 0;
7      private int mRecCount = 0;
8      //记录集
9      private ArrayList<Payout> mRs = null;
10
11     @Override
12     public void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.review_view);
15         //设置按钮事件侦听
16         mBtnNext = (Button)findViewById(R.id.BTN_NEXT);
17         mBtnPrev = (Button)findViewById(R.id.BTN_PREV);
18         Button btnBack = (Button)findViewById(R.id.BTN_BACK);
19         mBtnNext.setOnClickListener(this);
20         mBtnPrev.setOnClickListener(this);
21         btnBack.setOnClickListener(this);
22         //初始化数据集
23         initDataSet();
24         initShow();
25     }
26
27     private void initDataSet() { //初始化数据集
28         Cursor c = SQLiteUtil.getInstance().openQuery(IConfig.DB_PATH,
29                                                         IConfig.TBL_NAME, null);
30         mRecCount = c.getCount();
31         if(mRecCount > 0) { mRs = new ArrayList<Payout>(mRecCount);
32             while(!c.isAfterLast()) {

```



Android 平台开发之旅 第2版

```

33             Payout p = new Payout(c.getString(0),c.getString(1),c.getDouble(2));
34             mRs.add(p);
35             c.moveToNext();
36         }
37     }
38     c.close();
39 }
40 .....
41 };

```

在代码 7-16 中，首先使用 SQLite 工具类的 `openQuery` 方法获取日记账表中所有记录（第 28 行），再遍历结果集游标，获取各行的列内容，并使用列内容构造支出对象（第 33 行），这些结果集都会添加到记录对象容器中（第 34 行）。

当用户单击【后一条】按钮和【前一条】按钮改变当前记录的索引时，Activity 组件从支付对象容器中获取对应位置的条目，并显示该对象的内容。代码 7-17 是记录浏览导航控制的主要代码。

代码 7-17 记录浏览导航

文件名: ReviewRecAct.java

```

1  private void initShow() { //初始化显示
2      if(mRecCount>1) {      mBtnNext.setEnabled(true); mBtnPrev.setEnabled(false); }
3      else {                  mBtnNext.setEnabled(false);      mBtnPrev.setEnabled(false); }
4      //初始化显示
5      if(mRecCount>0) { showRecord(); }
6  }
7
8  @Override
9  public void onClick(View v) {
10     switch(v.getId() ) {      case R.id.BTN_BACK: { this.finish(); break; }
11                                case R.id.BTN_NEXT: { showRecord(1); break; }
12                                case R.id.BTN_PREV: { showRecord(-1); break; }
13     }
14 }
15
16 private void showRecord(int delta) { //按照顺序增量显示记录
17     if(delta > 0) { //向后浏览
18         mCurPos++;
19         //末条记录
20         if(mCurPos == (mRecCount-1) ) { mBtnNext.setEnabled(false); }
21         mBtnPrev.setEnabled(true);
22     } else { //向前浏览
23         mCurPos--;
24         //首条记录
25         if(mCurPos == 0) { mBtnPrev.setEnabled(false); }
26         mBtnNext.setEnabled(true);

```



```

27     }
28     showRecord();
29 }
30
31 //显示记录内容
32 private void showRecord() { Payout payout = mRs.get(mCurPos);
33     ((EditText)findViewById(R.id.txt_tsp)).setText(payout.getTsp() );
34     ((EditText)findViewById(R.id.txt_comments)).setText(payout.getComments() );
35     ((EditText)findViewById(R.id.txt_money)).setText(String.valueOf(payout.getMoney() ));
36 }

```

在代码 7-17 中，浏览导航过程除了对记录索引进行控制外，还对导航按钮的状态进行了调整。如果当前是末条记录，则【后一条】按钮将不可用（第 20 行）；如果当前是首条记录，则【前一条】按钮将不可用（第 25 行）；只有当记录在中间位置时，两个按钮都可用。

在第 33~35 行中，读者可以看出记录的显示和添加过程是互逆的：前者是将支付对象的成员内容展示到可视控件中；后者是将可视控件中的内容添加到支付对象所对应的数据库中，其间进行了记录到对象的转换。

8. 支付记录对象定义

代码 7-16 中第 33 行将记录游标中的每行记录与一个支付对象进行了对应，该过程可视为对象关系映射（Object Relational Mapping, ORM）的逆向过程，即通过数据库中的存储信息来构建对象实例。代码 7-18 是该支付记录对象的定义。

代码 7-18 支付记录对象的定义

文件名: Payout.java

```

1  public class Payout implements Parcelable {
2      private String mTsp = null;
3      private String mComments = null;
4      private double mMoney = 0.0D;
5
6      //必须要有一个名为 CREATOR 的成员对象，否则无法进行 Parcelable 对象通信
7      public static final Parcelable.Creator<Payout> CREATOR =
8          new Parcelable.Creator<Payout>() {
9              public Payout createFromParcel(Parcel in) { return new Payout(in); }
10             public Payout[] newArray(int size) { return new Payout[size]; }
11         };
12
13     public Payout(String tsp, String comments, double money) {
14         mTsp = tsp;      mComments = comments;      mMoney = money;
15     }
16
17     //设置/获取时间戳
18     public void setTsp(String tsp) { mTsp = tsp; }
19     public String getTsp() { return (mTsp); }
20     //设置/获取抬头

```



```

21     public void setComments(String comments) { mComments = comments; }
22     public String getComments() { return (mComments); }
23     //设置/获取金额
24     public void setMoney(double money) { mMoney = money; }
25     public double getMoney() { return (mMoney); }
26
27     public String toString() { return (mTsp+","+mComments+","+mMoney); }
28
29     //实现 Parcelable 接口
30     public Payout(Parcel in) {    this.mTsp = in.readString();
31                                this.mComments = in.readString();
32                                this.mMoney = in.readDouble();
33     }
34
35     @Override
36     public int describeContents() { return 0; }
37     @Override
38     public void writeToParcel(Parcel dest, int flags) { dest.writeString(this.mTsp);
39                                                         dest.writeString(this.mComments);
40                                                         dest.writeDouble(this.mMoney);
41     }
42 };

```

在代码 7-18 中，由于支付记录对象需要在 Activity 组件之间进行传递，所以该对象必须实现 Parcelable 接口，有关 Parcelable 接口的约定可参见第 3 章。

9. 记录查询 Activity

图 7-14 是在记录查询 Activity 组件中进行查询的实机界面，其中包含两个微调控件：第一个用于选择目标列，第二个用于选择查询操作符；文本框用于接收查询值，【查询】按钮用于提交查询，【取消】按钮用于返回到主 Activity 组件。界面总体布局为垂直方向的线性布局。

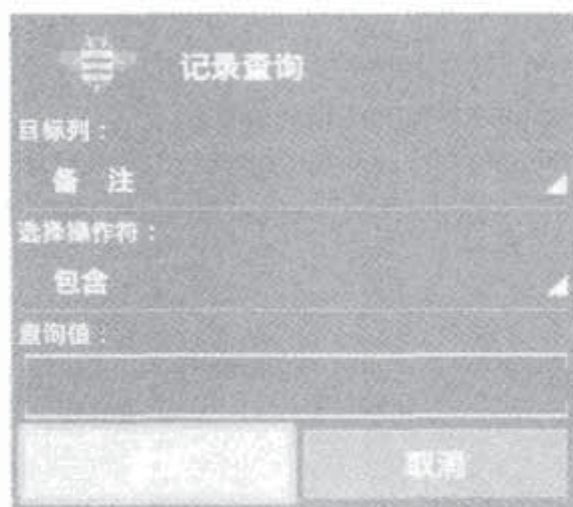


图 7-14 记录查询的实机界面

(1) 界面布局定义

代码 7-19 是记录查询 Activity 组件的布局定义。

代码 7-19 记录查询 Activity 组件的布局定义

文件名: lookup_view.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="320sp" android:layout_height="fill_parent" >
5      <TextView android:text="目标列: "
6          android:layout_width="fill_parent" android:layout_height="wrap_content"/>
7      <Spinner android:id="@+id/SPN_COLS" android:entries="@array/cols_title"
8          android:layout_width="fill_parent" android:layout_height="wrap_content"/>
9      <TextView android:text="选择操作符: "
10         android:layout_width="fill_parent" android:layout_height="wrap_content"/>
11     <Spinner android:id="@+id/SPN_OPS" android:entries="@array/ops_title"
12         android:layout_width="fill_parent" android:layout_height="wrap_content"/>
13     <TextView android:text="查询值: "
14         android:layout_width="fill_parent" android:layout_height="wrap_content"/>
15     <EditText android:id="@+id/txt_query"
16         android:layout_width="fill_parent" android:layout_height="wrap_content"/>
17     <TableLayout android:stretchColumns="0,1"
18         android:layout_width="fill_parent" android:layout_height="wrap_content">
19         <TableRow>
20             <Button android:id="@+id/btn_query" android:text="查询"
21                 android:layout_width="wrap_content" android:layout_height="wrap_content"/>
22             <Button android:id="@+id/qbtn_discard" android:text="取消"
23                 android:layout_width="wrap_content" android:layout_height="wrap_content"/>
24         </TableRow>
25     </TableLayout>
26 </LinearLayout>

```

(2) 记录查询 Activity

代码 7-20 是记录查询 Activity 组件的定义代码。

代码 7-20 记录查询 Activity 组件的定义

文件名: LookupRecAct.java

```

1  public class LookupRecAct extends Activity implements OnClickListener {
2      private Spinner mSpnColumns = null;
3      private Spinner mSpnOperators = null;
4      //列名列表和操作符列表
5      private String mCols[] = null;
6      private String mOps[] = null;
7
8      @Override
9      public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.lookup_view);

```



Android 平台开发之旅 第2版

```

12      //获取列名和查询操作符数组资源
13      mCols = getResources().getStringArray(R.array.cols);
14      mOps = getResources().getStringArray(R.array.ops);
15      //获取输入控制对象
16      mSpnColumns = (Spinner)findViewById(R.id.SPN_COLS);
17      mSpnOperators = (Spinner)findViewById(R.id.SPN_OPS);
18      //设置按钮事件侦听
19      Button btnQuery = (Button)findViewById(R.id.btn_query);
20      Button btnDiscard = (Button)findViewById(R.id.qbtn_discard);
21      btnQuery.setOnClickListener(this);
22      btnDiscard.setOnClickListener(this);
23  }
24  @Override
25  public void onClick(View v) {
26      switch(v.getId() ) {      case R.id.qbtn_discard: { this.finish(); break; }
27                               case R.id.btn_query: { doQuery(); break; }
28      }
29  }
30  .....
31  };

```

在代码 7-20 中，除了获取微调控件外（第 16 行和第 17 行），还获取了数组资源（第 13 行和第 14 行）。微调控件的条目所引用的数组是列标题和操作符标题（显示用），而此处引用的数组内容是数据库中列名和 SQL 操作符（生成 SQL 语句用）。

与记录添加 Activity 组件相似，记录查询 Activity 会根据用户的选择，拼凑成查询 SQL 语句，然后获取符合查询条件的记录。代码 7-21 是查询记录的主要代码。

代码 7-21 查询记录

文件名: LookupRecAct.java

```

1  private void doQuery() { //执行查询
2      String cond = getCondition();
3      //执行查询语句
4      Cursor c = SQLiteUtil.getInstance().openQuery(IConfig.DB_PATH,
5                                                     IConfig.TBL_NAME, cond);
6      int rc = c.getCount();
7      if(rc > 0) { ArrayList<Payout> rs = new ArrayList<Payout>(rc);
8          while(!c.isAfterLast()) {
9              Payout p = new Payout(c.getString(0),c.getString(1),c.getDouble(2));
10             rs.add(p);
11             c.moveToNext();
12         }
13         c.close();
14         //调用列表 Activity 组件进行显示
15         Intent reportRecIntent = new Intent(this, ReportRecAct.class);
16         reportRecIntent.putParcelableArrayListExtra(IConfig.EXTRA, rs);

```



```

17         this.startActivity(reportRecIntent);
18     } else { FoolUtil.getInstance().showMsg(this, "查询结果为空, 请重试! ");
19         return;
20     }
21 }
22
23 private String getCondition () { //生成查询语句条件子句
24     String colName = this.mCols[mSpnColumns.getSelectedItemPosition()];
25     String operator = this.mOps[mSpnOperators.getSelectedItemPosition()];
26     String val = ((EditText)findViewById(R.id.txt_query)).getText().toString();
27     StringBuffer sb = new StringBuffer(colName);
28     sb.append(' '+operator+' ');
29
30     if( (colName.compareToIgnoreCase("Timestamp") == 0) ||
31         (colName.compareToIgnoreCase("Comments") == 0) ) {
32         if(operator.compareToIgnoreCase("=") == 0) {
33             sb.append("\""+val+"\"");
34         } else if(operator.compareToIgnoreCase("LIKE") == 0) {
35             sb.append("\"%" + val + "%\"");
36         }
37     } else { sb.append(val); }
38     //返回条件子句
39     return (sb.toString() );
40 }

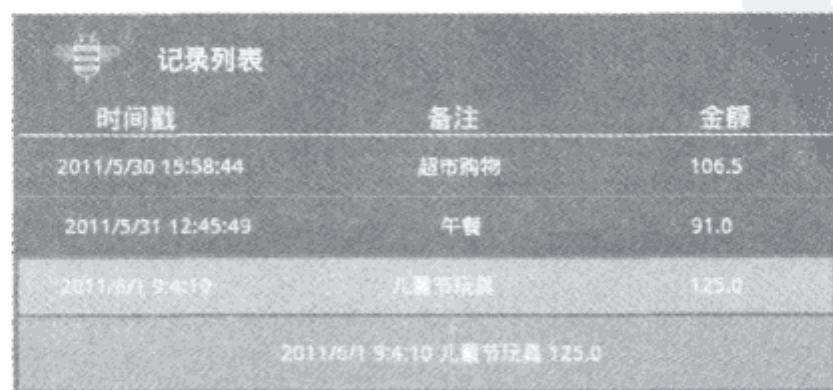
```

在代码 7-21 中, Activity 首先依据用户的选择生成 SQL 语句条件子句 (第 2 行)。在生成条件子句的方法中 (第 23 行的 getCondition), 通过列名、操作符和查询值这 3 个要素来构建条件子句。与记录浏览 Activity 中的查询 (代码 7-16 中第 28 行) 不同, 记录查询 Activity 中的查询带有条件 (第 5 行), 其只选择符合条件的记录。

查询结果的获取与记录浏览 Activity 相同, 都是先将记录转换成对象 (第 9 行), 并存入对象容器中 (第 10 行)。最后, 记录查询 Activity 借助意图对象扩展空间将支付对象容器传递给记录列表 Activity 组件 (第 15~17 行)。

10. 记录列表 Activity

图 7-15 是记录列表 Activity 的实机界面, 其中主要组件是列表视图, 该视图又包括 3 个部分: 页眉视图、行视图和页脚视图。



时间戳	备注	金额
2011/5/30 15:58:44	超市购物	106.5
2011/5/31 12:45:49	午餐	91.0
2011/6/1 9:4:19	儿童节玩具	125.0

图 7-15 记录列表 Activity 的实机界面



Android 平台开发之旅 第2版

(1) 界面布局定义

代码 7-22 是记录列表 Activity 组件的布局定义，其中引用的是系统资源中的列表视图和文本视图（第 5 行和第 8 行）。

代码 7-22 记录列表 Activity 组件的布局定义

文件名: report_view.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <ListView android:id="@id/android:list"
6          android:layout_width="fill_parent" android:layout_height="fill_parent"
7          android:drawSelectorOnTop="false" />
8  </LinearLayout>

```

(2) 行视图

代码 7-23 是记录列表 Activity 组件中列表视图的行视图的定义，其中主要包含 3 个文本视图，用于显示每条记录的 3 列数据内容。

代码 7-23 列表视图的行视图

文件名: row_view.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="fill_parent" android:layout_height="wrap_content"
4      android:orientation="vertical" android:stretchColumns="0,1,2">
5      <TableRow android:layout_gravity="center_horizontal">
6          <TextView android:id="@+id/lab_tsp"
7              android:layout_width="wrap_content" android:layout_height="40sp"
8              android:text="时间戳" android:gravity="center"/>
9          <TextView android:id="@+id/lab_comments"
10              android:layout_width="wrap_content" android:layout_height="fill_parent"
11              android:text="备注" android:gravity="center"/>
12          <TextView android:id="@+id/lab_money"
13              android:layout_width="100sp" android:layout_height="fill_parent"
14              android:text="金额" android:gravity="center"/>
15      </TableRow>
16  </TableLayout>

```

(3) 页眉视图

代码 7-24 是记录列表 Activity 组件中列表视图的页眉视图的定义，其中主要包含 3 个文本视图，用于显示 3 列的标题，另外一个文本视图用做隔离线（第 16 行）。

代码 7-24 列表视图的页眉视图

文件名: header.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="horizontal" android:stretchColumns="0,1,2"
4      android:layout_width="fill_parent" android:layout_height="50sp">
5      <TableRow>
6          <TextView android:text="时间戳" android:gravity="center"
7              android:layout_width="fill_parent" android:layout_height="fill_parent"
8              android:textColor="#FF00FF00" android:textSize="8pt"/>
9          <TextView android:text="备注" android:gravity="center"
10             android:layout_width="fill_parent" android:layout_height="fill_parent"
11             android:textColor="#FF00FF00" android:textSize="8pt"/>
12          <TextView android:text="金额" android:gravity="center"
13             android:layout_width="fill_parent" android:layout_height="fill_parent"
14             android:textColor="#FF00FF00" android:textSize="8pt"/>
15      </TableRow>
16      <TextView android:layout_width="fill_parent" android:layout_height="1dp"
17          android:background="#FF00FF00" />
18  </TableLayout>

```

(4) 页脚视图

代码 7-25 是记录列表 Activity 组件中列表视图的页脚视图的定义, 其中只有一个文本视图, 用于显示当前所选择项的内容。

代码 7-25 列表视图的页脚视图

文件名: footer.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="wrap_content">
5      <TextView android:id="@+id/txt_footer"
6          android:layout_width="fill_parent" android:layout_height="50sp"
7          android:background="#FF333333" android:gravity="center"/>
8  </LinearLayout>

```

(5) 记录列表 Activity 组件

代码 7-26 是记录列表 Activity 组件的定义代码, 该 Activity 继承于列表 Activity。

代码 7-26 记录列表 Activity 组件的定义

文件名: ReportRecAct.java

```

1  public class ReportRecAct extends ListActivity {
2      private String mCols[] = null;
3      private final int[] mViews = { R.id.lab_tsp,R.id.lab_comments,R.id.lab_money };

```




Android 平台开发之旅 第2版

```

4      private ArrayList<HashMap<String,String>> mItems =
5                                          new ArrayList<HashMap<String,String>>();
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.report_view);
10         //设置页眉和页脚
11         LayoutInflater inflater = this.getLayoutInflater();
12         this.getListView().addHeaderView(inflater.inflate(R.layout.header, null), null, false);
13         this.getListView().addFooterView(inflater.inflate(R.layout.footer, null), null, false);
14         //获取意向参数
15         ArrayList<Payout> rs = getIntent().getParcelableArrayListExtra(IConfig.EXTRA);
16         //提取记录
17         for(int i = 0; i < rs.size(); ++i) { addRecord(rs.get(i)); }
18         //获取列名数组资源
19         mCols = getResources().getStringArray(R.array.cols);
20         //创建列表数据适配器
21         ListAdapter adapter = new SimpleAdapter(this, mItems, R.layout.row_view,
22                                             new String[] { mCols[1],mCols[0],mCols[2] }, mViews);
23         //设置数据适配器, 绑定数据
24         setListAdapter(adapter);
25     }
26
27     private void addRecord(Payout payout) { //转储记录
28         HashMap<String,String> item = new HashMap<String,String>();
29         item.put(mCols[1], payout.getTsp() );
30         item.put(mCols[0], payout.getComments() );
31         item.put(mCols[2], String.valueOf(payout.getMoney()) );
32         mItems.add(item);
33     }
34
35     @Override
36     protected void onItemClick(ListView l, View v, int pos, long id) {
37         HashMap<String,String> item = (HashMap<String,String>)(mItems.get(pos-1) );
38         //在页脚视图中显示当前所选项
39         String hint = item.get(mCols[1]).toString()+" "+
40                     item.get(mCols[0]).toString()+" "+ item.get(mCols[2]).toString();
41         ((TextView)findViewById(R.id.txt_footer)).setText(hint);
42     }
43 };

```

在代码 7-26 中, 首先是从意向对象中获取所传递过来的对象容器 (第 15 行), 并转储成列表项形式 (第 17 行), 继而使用列表项容器构建列表视图的适配器 (第 21 行)。列表视图的适配器设置完毕后, 就可通过列表项的单击事件回调函数来响应单击动作 (第 36 行)。

需要注意的是, 对于列表视图, 页眉视图和页脚视图的添加 (第 12 行和第 13 行) 必须在设置适配器 (第 24 行) 之前, 因为页眉视图和页脚视图也会作为列表视图的一行。因

此，在第 37 行的行位置的基础上减 1，这是因为需要排除首行是页眉视图。

7.6 嵌入式对象数据库 Db4o

Db4o (<http://www.db4o.com/>) 是 Database for objects 的缩写，其内容是基于对象的数据库。Db4o 支持 Java 和 C#两种语言，通过其官方网站可获得最新的开发包。

Db4o 在嵌入式和移动平台中的应用十分广泛。例如，通过 Db4o 数据库来记忆车辆的状态和用户的喜好设置，在换购新的车辆时可导入 Db4o 数据库来迅速获取用户的偏好设置；通过 Db4o 数据库来实现移动公文包（Offline Briefcase）：当用户不在办公室的时候将数据直接保存到手机或 PDA 上的 Db4o 数据库中，回到办公室可连接到中心数据库时，再将移动设备中 Db4o 数据库中的记录导入到中心数据库。

Db4o 封装了对象的存储、检索和更新等细节，所以开发者不必再为这些操作而劳心费神，而只需要关注业务逻辑。另外，Db4o 提供了多个子集版本，方便应用系统的选择。例如，对 Java 平台的支持版本有 1.5、1.2（1.2~1.4）和 1.1 版本，对于 Android 平台可选择 1.5 版本，如图 7-16 所示。

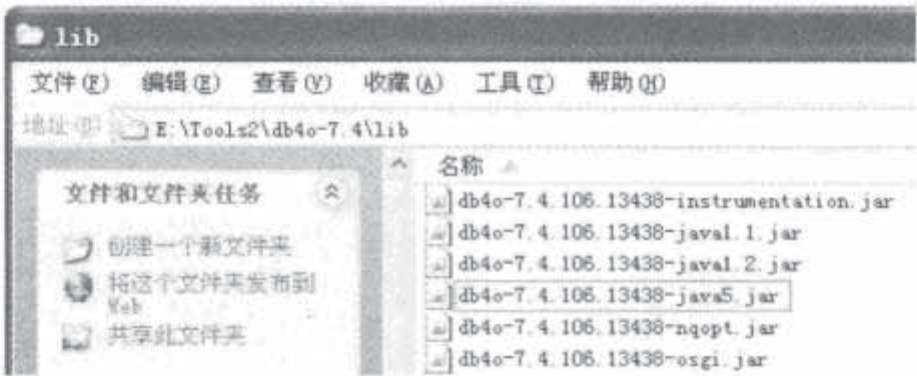


图 7-16 Db4o 对 Java 平台的支持版本

7.6.1 Db4o 对 Android 平台的支持

Db4o 提供的支持 Java 5 的 API 能够很好地兼容 Android 平台，本书使用的 Db4o 的开发包是 Db4o 7.4 版本。

7.6.2 Db4o API

Db4o API 提供多达 24 个包，这里介绍几个较为重要的包。表 7-9 是对核心包（com.db4o）中重要的类/接口的说明。

表 7-9 Db4o 包中重要的类/接口的说明

类/接口	说 明
Db4o	初始化 Db4o 数据库引擎的工厂类
ObjectContainer	代表一个 Db4o 数据库接口
ObjectSet	查询结果集接口

表 7-10 是对查询包（com.db4o.query）中重要的类/接口的说明。



表 7-10 查询包中重要的类/接口的说明

类/接口	说 明
Predicate	用于本地查询的谓词类

表 7-11 是对配置包（com.db4o.config）中重要的类/接口的说明。

表 7-11 配置包中重要的类/接口的说明

类/接口	说 明
Configuration	配置接口

详细的参考可查阅 Db4o 开发文档。

7.6.3 Db4o 数据库应用

1. 部署 Db4o 包文件

因为 Android 平台不包含 Db4o 运行环境，所以如果 Android 应用程序要使用 Db4o API，则需将 Db4o 包文件与应用程序一起分发到模拟器或实机中。在 Eclipse 开发环境中，将 Db4o 包文件以参考库的形式添加到工程中，包文件会被添加到应用程序安装文件（apk）中。

由于 Eclipse 工程一般以绝对路径引用包文件，所以为了防止由于包文件的移动造成参考路径的失效，建议在工程文件夹中额外创建一个名为 lib 的文件夹，并将 Db4o 包文件复制到其中，同时修改工程文件夹中的类路径配置文件（.classpath），如代码 7-27 所示。

代码 7-27 工程类路径配置文件

文件名: .classpath

1	<?xml version="1.0" encoding="UTF-8"?>
2	<classpath>
3	<classpathentry kind="src" path="src"/>
4	<classpathentry kind="src" path="gen"/>
5	<classpathentry kind="con" path="com.android.ide.eclipse.adt.ANDROID_FRAMEWORK"/>
6	<classpathentry kind="lib" path="lib/db4o-7.4.106.13438-java5.jar"/>
7	<classpathentry kind="output" path="bin"/>
8	</classpath>

因为代码 7-27 中引用参考库使用的是相对于工程文件夹的路径（第 6 行），所以不会存在当工程文件夹移动后造成参考路径无效的情形。

修改工程的类路径配置文件后，刷新工程，其结果如图 7-17 所示。

2. 数据库操作

与 SQLite 相同，Db4o 数据库的操作也包括打开、关闭和删除。



图 7-17 引入 Db4o 包文件

(1) 创建/打开数据库

使用 Db4o 类的 `openFile` 方法可创建或打开指定数据库，如果该数据库文件不存在则会自动创建。

注意：Db4o 类提供了两种 `openFile` 方法，其区别在于：第一种方法需要指定配置，而另一种无需指定配置。通过指定可引用本地化的配置，如果不指定，则采用 Db4o 的全球化的配置。对于 Db4o 7.4 版本，在 Android 平台中使用不指定配置的 `openFile` 方法打开已经存在的数据库时会抛出“转换到 `GenericObject` 类”的异常信息，代码如下所示。

java.lang.ClassCastException: com.db4o.reflect.generic.GenericObject

为了避免该异常的出现，程序中必须指定使用本地化的配置来打开数据库文件，并显式地设置该配置的反射器（`Reflector`）。代码 7-28 是打开 Db4o 数据库的方法定义。

代码 7-28 打开 Db4o 数据库的方法定义

文件名：OdbUtil.java

```
1  //打开数据库文件
2  private ObjectContainer openDBFile(final String odbName) {
3      //创建一个新的配置实例
4      Configuration config = Db4o.newConfiguration();
5      //为配置指定特定的反射
6      //config.reflectWith(new JdkReflector(ClassLoader.getSystemClassLoader() ));
7      config.reflectWith(new JdkReflector(this.getClass().getClassLoader() ));
8      //使用指定的配置打开数据库
9      //如不使用指定的配置，在第二次打开数据库文件时会提示有关反射的运行时错误
10     return (Db4o.openFile(config, odbName) );
11 }
```

在代码 7-28 中，创建了一个配置实例（第 4 行），并为其设置了一个以当前类的类载入器（`ClassLoader`）进行反射处理的反射器（第 7 行），最后在 `openFile` 方法中指定以该配置来打开指定数据库。

当数据库创建成功，程序会在指定的路径生成 Db4o 数据库的库文件，该文件的扩展名一般为“.yap”。

注意：同 SQLite 数据库，创建 Db4o 数据库文件需要在 SD 卡上创建文件或文件夹，所以也需要在工程清单文件中声明允许写外部存储器（SD 卡）的许可。

(2) 关闭数据库

通过 Db4o 类实例的 `close` 方法可关闭数据库。

(3) 删除数据库

数据库的删除与普通文件的删除没有什么区别，只需数据库在关闭状态下。

3. 对象集合操作

如果说 SQLite 数据库中存放的是记录集合，那么 Db4o 数据库中存放的就是对象集合。对象集合的操作包括对象的存储、遍历和查询。



(1) 存储对象

使用对象容器实例 (ObjectContainer) 的 store 方法可将内存对象存储到数据库中。

(2) 遍历对象

通过对象容器实例的查询方法可获取 Db4o 数据库中符合条件的对象的集合, 该集合类似于枚举接口 (Enumeration), 通过其 hasNext 方法和 next 方法可遍历该集合内所有对象。

(3) 查询对象

对象查询是 Db4o 数据库的特性。在 SQLite 的应用中, 记录的插入、查询和删除都可通过 SQL 语句来统一执行, 而 Db4o 却不是通过这种方式。

Db4o 提供了 3 种不同的查询模式: 按例查询 (Query-By-Example, QBE)、本地查询 (Native Query, NQ) 和 SODA API 方式, 这 3 种查询的使用技巧可参考 Db4o 文档。

4. Db4o 数据库工具类

为了统一对数据库的使用, 将 Db4o 数据库的常用操作封装成工具类。代码 7-29 就是该工具类的定义, 其主要功能包括 Db4o 数据库的打开、关闭和删除、对象的添加、遍历和查询等。

代码 7-29 Db4o 数据库工具类的定义

文件名: OdbUtil.java

```

1  public class OdbUtil {
2      //单例接口
3      private static OdbUtil mInstance = new OdbUtil();
4      private OdbUtil() {}
5      public static OdbUtil getInstance() { return (mInstance); }
6
7      //打开数据库
8      public ObjectContainer openDB(String odbName) { File file = new File(odbName);
9          //库文件存在则打开数据库, 如果不存在则初始化
10         if(file.exists() == true) { return (openDBFile(odbName)); }
11         } else { return (initDB(odbName)); }
12     }
13
14     private ObjectContainer openDBFile(final String odbName) { //打开数据库文件
15         //创建一个新的配置实例
16         Configuration config = Db4o.newConfiguration();
17         //为配置指定特定的反射
18         config.reflectWith(new JdkReflector(this.getClass().getClassLoader()));
19         //使用指定的配置打开数据库
20         //如不使用指定的配置, 在第二次打开数据库文件时会提示有关反射的运行时错误
21         return (Db4o.openFile(config, odbName));
22     }
23
24     private ObjectContainer initDB(String odbName) { //初始化数据库
25         //库文件已经存在则删除
26         if(deleteDB(odbName) == false) { return (null); }

```



```
27         return (openDBFile(odbName) );
28     }
29
30     //关闭数据库
31     public void closeDB(ObjectContainer odb) { odb.close(); }
32
33     //删除数据库
34     public boolean deleteDB(String odbName) { File file = new File(odbName);
35         //库文件存在则删除
36         if(file.exists() == true) { return (file.delete() ); }
37         return (true);
38     }
39
40     public void addObject(String odbName, Object obj) { //添加对象
41         //打开数据库
42         ObjectContainer odb = openDB(odbName);
43         //存储对象
44         odb.store(obj);
45         //关闭数据库
46         closeDB(odb);
47     }
48
49     //获取数据库中指定条件的所有对象(Query By Example)
50     public ArrayList<Payout> getObjects(String odbName, Object proto) {
51         //打开数据库
52         ObjectContainer odb = openDB(odbName);
53         //查询获得对象集
54         ObjectSet<Payout> objectSet = odb.queryByExample(proto);
55         //对象容器
56         ArrayList<Payout> objectDB = new ArrayList<Payout>();
57         //遍历对象集合
58         while(objectSet.hasNext() ) { objectDB.add(objectSet.next() ); }
59         //关闭数据库
60         closeDB(odb);
61         return (objectDB);
62     }
63
64     //获取数据库中指定条件的所有对象(Native Query)
65     public ArrayList<Payout> queryObjects(String odbName, Predicate<Payout> pre) {
66         //打开数据库
67         ObjectContainer odb = openDB(odbName);
68         //查询获得对象集
69         ObjectSet<Payout> objectSet = odb.query(pre);
70         //对象容器
71         ArrayList<Payout> objectDB = new ArrayList<Payout>();
72         //遍历对象集合
```



```
73         while(objectSet.hasNext() ) { objectDB.add(objectSet.next() ); }
74         //关闭数据库
75         closeDB(odb);
76         return (objectDB);
77     }
78 };
```

7.6.4 基于 Db4o 数据库的日记账工具

与 SQLite 数据库的介绍方式一样，下面将介绍一款基于 Db4o 数据库的日记账工具，让读者能够借此巩固对 Db4o 数据库的应用，并体会 Db4o 与 SQLite 的应用差异。该工具的界面定义与 SQLite 版本基本相同，如图 7-18 所示。

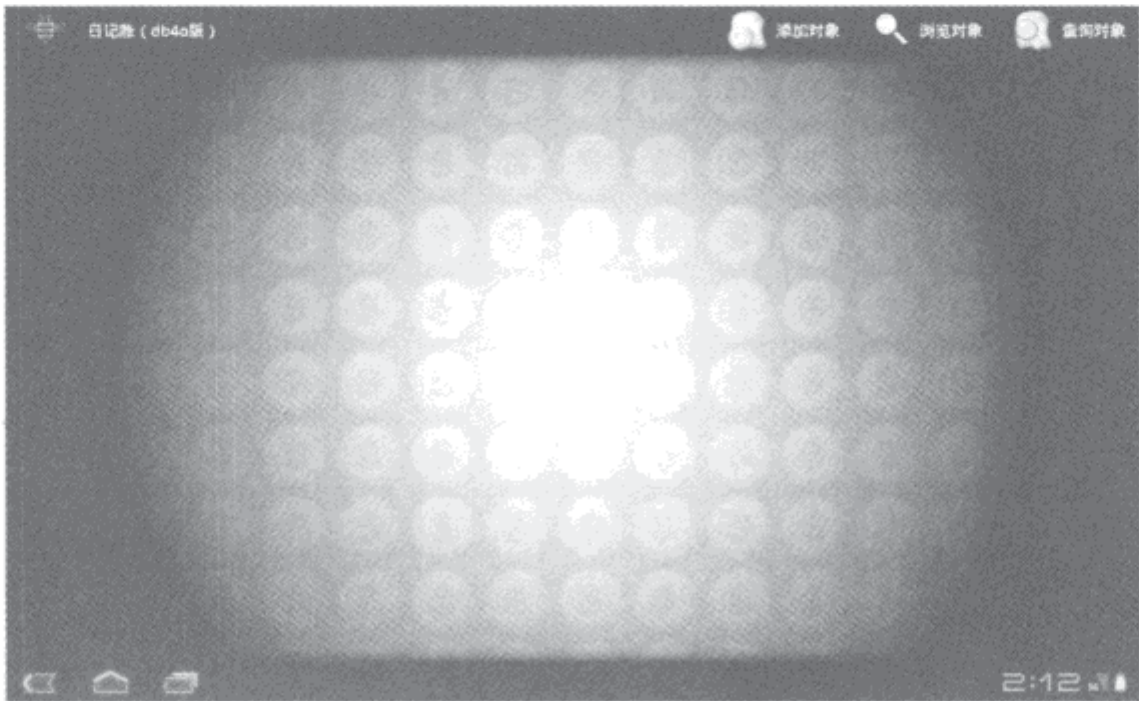


图 7-18 日记账工具 (Db4o 版)

提示：由于 Db4o 版的工具与 SQLite 版本在结构上基本相同，所以在对 Db4o 工具的介绍中，对于框架结构上的内容不再重复，而主要介绍在功能实现上与 SQLite 数据库不一样的地方。

1. 程序清单

代码 7-30 是该程序清单文件的内容。

代码 7-30 程序清单文件

文件名：AndroidManifest.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3          package="foolstudio.demo.db.journalbook2"
4          android:versionCode="1"
5          android:versionName="1.0">
6      <uses-sdk android:minSdkVersion="12"/>
7      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```



```

8      <application android:icon="@drawable/icon" android:label="@string/app_name">
9          <activity android:name=".JournalBook2Act" android:label="@string/app_name">
10              <intent-filter>
11                  <action android:name="android.intent.action.MAIN" />
12                  <category android:name="android.intent.category.LAUNCHER" />
13              </intent-filter>
14          </activity>
15          <activity android:name=".AppendRecAct" android:label="@string/act_add"/>
16          <activity android:name=".ReviewRecAct" android:label="@string/act_view"/>
17          <activity android:name=".LookupRecAct" android:label="@string/act_look"/>
18          <activity android:name=".ReportRecAct" android:label="@string/act_report"/>
19      </application>
20 </manifest>

```

在代码 7-30 中，除了主 Activity 组件外，应用程序还包含 4 个 Activity 组件（第 15~18 行）。图 7-19 是该日记账工具的组件关联示意图。



图 7-19 日记账工具的组件关联示意图

在图 7-19 中，主 Activity 与 3 个 Activity 组件之间是调用关系，通过调用子 Activity 组件对 Db4o 数据库进行相关操作，包括浏览对象、添加对象和查询对象。最终使用这些子 Activity 组件的界面对操作结果进行显示。

由于数据库的创建、删除以及对象的添加都是写入操作，需要应用程序具有写存储器的许可，所以在程序清单文件第 7 行声明了写外部存储设备的使用许可。

2. 选项菜单定义

代码 7-31 是应用程序选项菜单的定义。

代码 7-31 主 Activity 组件选项菜单的定义

文件名: ops_menu.xml

```

1 <menu xmlns:android="http://schemas.android.com/apk/res/android">
2     <item android:title="添加对象" android:icon="@drawable/append"
3         android:showAsAction="ifRoom|withText" android:id="@+id/mi_add"/>
4     <item android:title="浏览对象" android:icon="@drawable/preview"
5         android:showAsAction="ifRoom|withText" android:id="@+id/mi_view"/>
6     <item android:title="查询对象" android:icon="@drawable/lookup"

```



```

7         android:showAsAction="ifRoom|withText" android:id="@+id/mi_look"/>
8     </menu>

```

图 7-20 是该选项菜单的实机界面，前三项菜单将在操作栏中既显示图标，又显示文字。



图 7-20 选项菜单的实机界面

3. 应用程序配置信息接口

代码 7-32 是该配置信息接口的定义。

代码 7-32 配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String DB_PATH = "/sdcard/JournalBook.yap";
3      public static final String TBL_NAME = "Payout";
4      //数据项
5      public static final String EXTRA = "_rs";
6  };

```

4. 添加对象

代码 7-33 是添加对象的主要代码。

代码 7-33 添加对象

文件名: AppendRecAct.java

```

1  private void doSubmit() { //提交添加
2      if(submitCheck() == false) { return; }
3      //构建对象
4      Payout payout = new Payout(mTxtTsp.getText().toString().trim(),
5                                  mTxtComments.getText().toString().trim(),
6                                  Double.parseDouble(mTxtMoney.getText().toString().trim()));
7      //添加对象
8      OdbUtil.getInstance().addObject(IConfig.DB_PATH, payout);
9      FoolUtil.getInstance().showMsg(this, "对象存储成功!");
10     this.finish();
11 }

```

在代码 7-33 中，首先对用户的输入进行合法性检查（第 2 行），然后依据输入内容构建支付对象（第 4 行），继而使用 Db4o 数据库工具类的 addObject 方法直接将该对象添加到数据库中（第 8 行）。

5. 遍历对象

代码 7-34 是遍历对象并初始化对象集合的主要代码。

代码 7-34 初始化对象集合

文件名: ReviewRecAct.java

```
1 private void initDataSet() { //初始化数据集
2     Payout proto = new Payout(null, null, 0.0D);
3     mRs = OdbUtil.getInstance().getObjects(IConfig.DB_PATH, proto);
4     mRecCount = mRs.size();
5     if(mRecCount > 1) { mBtnNext.setEnabled(true); mBtnPrev.setEnabled(false);
6     } else { mBtnNext.setEnabled(false); mBtnPrev.setEnabled(false); }
7     //初始化显示
8     if(mRecCount > 0) { showRecord(); }
9 }
```

在代码 7-34 中, 使用对象原型 (第 2 行) 来获取数据库中所有的对象 (第 3 行), 继而对对象内容进行显示 (第 8 行)。

6. 查询对象

代码 7-35 是查询对象的主要代码。

代码 7-35 查询对象

文件名: LookupRecAct.java

```
1 private void doQuery() { //提交查询
2     String colName = this.mCols[mSpnColumns.getSelectedItemPosition()];
3     String operator = this.mOps[mSpnOperators.getSelectedItemPosition()];
4     String val = mTxtValue.getText().toString().trim();
5     //创建查询谓词实例
6     Predicate<Payout> pre = new PayoutPredicate(colName, operator, val);
7     //执行查询操作
8     ArrayList<Payout> rs = OdbUtil.getInstance().queryObjects(IConfig.DB_PATH, pre);
9     if(rs.size() > 0) {
10         Intent reportRecIntent = new Intent(this, ReportRecAct.class);
11         reportRecIntent.putParcelableArrayListExtra(IConfig.EXTRA, rs);
12         this.startActivity(reportRecIntent);
13     } else { FoolUtil.getInstance().showMsg(this, "查询结果为空, 请重试!");
14         return;
15     }
16 }
```

在代码 7-35 中, 对象查询 Activity 依据用户选择构建了一个用于查询的谓词对象 (第 6 行), 继而通过这个谓词对象来进行查询动作, 并获取对象记录集合 (第 8 行)。

提示: 如果读者对数据库有一定研究, 应该知道在数据库技术中有谓词的概念。例如, SQL 语法所定义的: EXISTS (是否存在)、IN (是否在之内) 和 LIKE (模式匹配) 就是开发者常用的 SQL 谓词。这些谓词包含了对集合中记录进行过滤的规则, 从这个角度而言, Db4o 所定义的谓词就是用于过滤对象的规则, 通过这些规则, Db4o 查询引擎可知道哪些对象记录符合条件, 哪些该过滤掉。



7. 查询用谓词定义

代码 7-36 是代码 7-35 中用于查询对象的谓词定义。

代码 7-36 Db4o 查询对象的谓词定义

文件名: PayoutPredicate.java

```

1  public class PayoutPredicate extends Predicate<Payout> {
2      private static final long serialVersionUID = 1L;
3
4      private String mCol = null;
5      private String mOperator = null;
6      private String mValue = null;
7
8      public PayoutPredicate(String col, String operator, String value) {
9          mValue = value; mOperator = operator; mCol = col;
10     }
11
12     @Override
13     public boolean match(Payout payout) {
14         if(mCol.compareToIgnoreCase("Timestamp") == 0) { //时间戳成员匹配
15             if(mOperator.compareToIgnoreCase("=") == 0) {
16                 return (payout.getTsp().equals(mValue));
17             } else if(mOperator.compareToIgnoreCase("LIKE") == 0) {
18                 return (payout.getTsp().indexOf(mValue) != -1);
19             }
20         } else if(mCol.compareToIgnoreCase("Comments") == 0) { //备注成员匹配
21             if(mOperator.compareToIgnoreCase("=") == 0) {
22                 return (payout.getComments().equals(mValue));
23             } else if(mOperator.compareToIgnoreCase("LIKE") == 0) {
24                 return (payout.getComments().indexOf(mValue) != -1);
25             }
26         } else { //金额成员匹配
27             if(mOperator.compareToIgnoreCase("=") == 0) {
28                 return (payout.getMoney() == Double.parseDouble(mValue));
29             } else if(mOperator.compareToIgnoreCase(">") == 0) {
30                 return (payout.getMoney() > Double.parseDouble(mValue));
31             } else if(mOperator.compareToIgnoreCase("<") == 0) {
32                 return (payout.getMoney() < Double.parseDouble(mValue));
33             }
34         }
35         return (false);
36     }
37 };

```

在代码 7-36 中, 通过重载于 Db4o 的谓词类 (Predicate) 的 match 方法 (第 13 行), 就是用于建立对象的匹配规则。在 Db4o 依据谓词对象进行对象查询的时候, 会将数据库中每

一个对象都使用该谓词类定义的匹配规制来进行检查，如果当前对象满足匹配条件则将其放入到对象集合中，否则将过滤。

7.7 数据库开发小结

从数据处理机制而言，SQLite 数据库基于 SQL 引擎，数据的最小单位是记录；而 Db4o 数据库是基于对象存储技术，数据的最小单位是对象。一般的，在内存中，数据记录多以对象容器的形式进行存放。

从数据库的组织形式而言，SQLite 和 Db4o 数据库都是单个文件，所以对于 SQLite 和 Db4o 数据库而言，数据库的管理就是对数据库文件的管理。

作为嵌入式平台应用，Android 平台对 SQLite 数据库进行了封装，用户不用过多考虑对数据库连接以及 SQL 语句的管理。但是 Db4o 数据库要比 SQLite 数据库更为简洁，对于对象存储的管理要比记录游标的管理更容易让开发者理解。

但是 Db4o 数据库在数据查询方面就稍显复杂，需要用户来定义对象的比较规则（谓词对象），而对于 SQL 方式，数据操作的模式较为统一。



第 8 章 网络通信与 Web 开发

本章对 Android 平台所支持的网络管理及应用进行详细的介绍,包括网络连接管理、网页内容集成、浏览器信息管理和系统下载管理等。通过这些介绍,可以帮助读者全面了解 Android 平台中的网络应用模式,这对于扩展应用程序功能大有益处。

8.1 Android 平台网络通信

Android 系统根目录下的“/dev”文件夹存放的是所有系统设备的描述符,其中包括套接字设备(socket)。套接字几乎是所有操作系统中进行网络通信的核心设备,无论是 TCP 通信还是 UDP 通信都基于套接字。

无线相容性认证(Wireless Fidelity, Wi-Fi)是 IEEE 802.11b 的别称,是由无线以太网兼容性联盟(Wireless Ethernet Compatibility Alliance, WECA)所发布的业界术语。它是一种短程无线传输技术,其最大优点就是数据传输速率较高,可达到 11Mbit/s;其有效距离也很长。

与蓝牙一样, Wi-Fi 也是在办公室和家庭中常用的短距离无线技术,虽然在数据安全性方面, Wi-Fi 比蓝牙要弱一些;但是在通信的效率和信号覆盖方面则比蓝牙有较大优势。Wi-Fi 的通信覆盖范围可达 100m,因此 Wi-Fi 一直是企业实现无线局域网所青睐的方式。

提示:随着 3G 网络和 3G 手机的逐步普及,读者可能越来越多地看到或听到产品介绍中提及的,某手机支持 WAPI 和 Wi-Fi 无线上网。那么 WAPI 又是如何与 Wi-Fi 扯上关系的呢?无线局域网鉴别和保密的基础结构(Wireless LAN Authentication and Privacy Infrastructure, WAPI)是一种无线局域网安全协议,也是中国无线局域网强制性标准中的安全机制。相比 Wi-Fi 技术, WAPI 的安全性要更胜一筹。特别是作为中国制定的标准, WAPI 标准可谓是历经艰难险阻,从 2004 年开始,直到 2009 年 6 月才获得国际标准组织 ISO/IEC JTC1/SC6 会议成员的一致同意。

8.2 Android 平台对网络通信的支持

Android 平台为网络通信提供了丰富的开发接口,不仅包含了 J2SE 平台中有关网络通信的功能,而且还将 Apache 的 HTTP 通信有关的包也纳入进来。在此基础上, Android 平台还提供了一些网络访问的帮助类。囿于篇幅,本书中主要介绍 Android 平台的相关内容。

表 8-1 是对 Android 平台中有关网络通信重要的类/接口的说明。

表 8-1 Android 平台网络通信重要的类/接口的说明

类/接口	说 明
ConnectivityManager	连接管理器
NetworkInfo	描述网络接口状态

Android 平台提供了 Wi-Fi 包（android.net.wifi）用于 Wi-Fi 应用。表 8-2 是对该包中重要的类/接口的说明。

表 8-2 Wi-Fi 包中重要的类/接口的说明

类/接口	说 明
WifiManager	Wi-Fi 连接管理器，用于管理 Wi-Fi 连接
WifiInfo	描述了 Wi-Fi 连接状态
WifiConfiguration	代表已配置的 Wi-Fi 网络的配置信息
ScanResult	用于描述检测到的 Wi-Fi 接入点的信息

8.3 网络连接管理

8.3.1 连接管理

连接管理器（ConnectivityManager）用于获取网络连接（包括 Wi-Fi、GPRS、UMTS 等）状态，其定义于网络包（android.net）中。图 8-1 所示为监视网络状态并通过连接管理器获取当前活跃连接的实机界面。

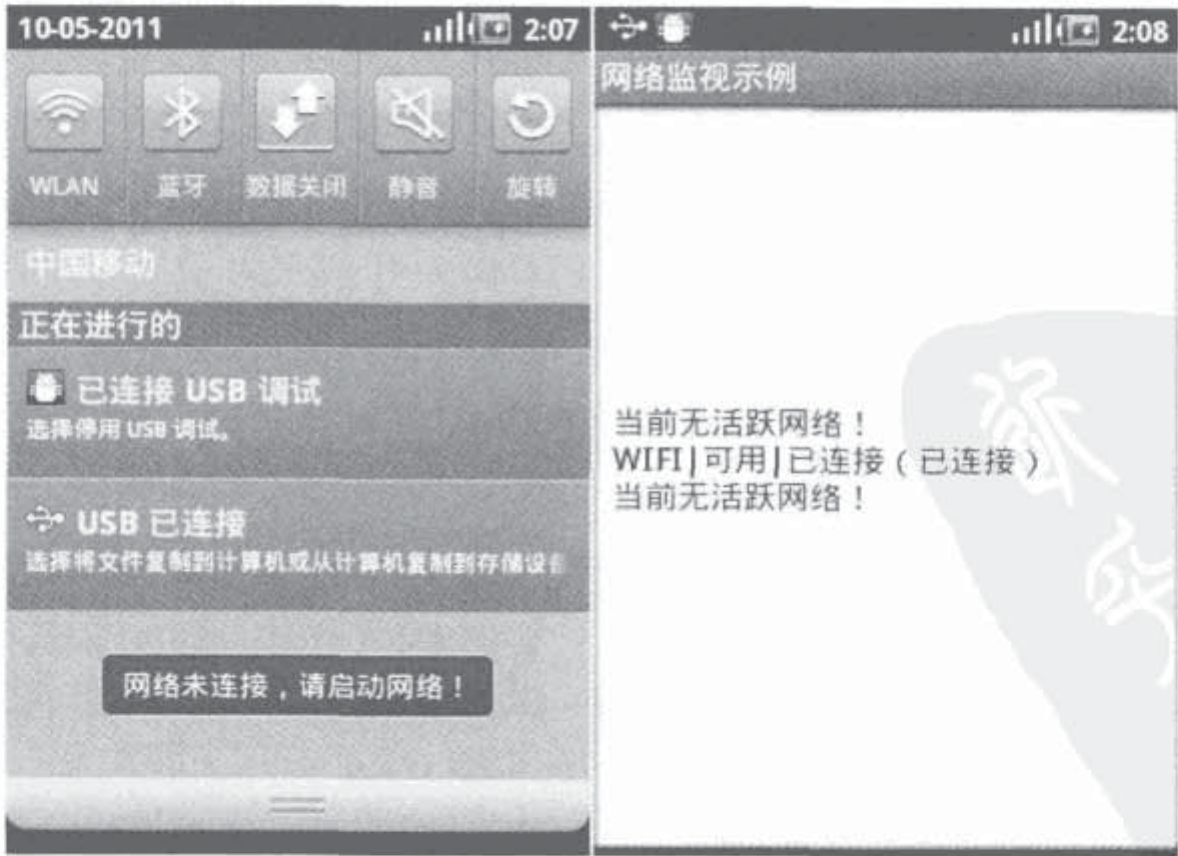


图 8-1 获取网络连接状态信息的实机界面



1. 应用程序主 Activity 框架

代码 8-1 是网络连接监视示例程序的主 Activity 组件的框架定义。

代码 8-1 网络连接监视示例程序的主 Activity 组件的框架定义

文件名: NetMonitorAct.java

```
1  public class NetMonitorAct extends Activity {
2      //连接管理器
3      private ConnectivityManager mMgr = null;
4      private FooNetMonitor mMonitor = null;
5
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.main);
10         //初始化
11         init();
12     }
13
14     private void init() {
15         //获取连接管理器实例
16         final String sname = Context.CONNECTIVITY_SERVICE;
17         mMgr = (ConnectivityManager)this.getSystemService(sname);
18         //初始化监视器和意向过滤器
19         mMonitor = new FooNetMonitor();
20         final String aname = ConnectivityManager.CONNECTIVITY_ACTION;
21         IntentFilter filter = new IntentFilter(aname);
22         //注册监视器
23         this.registerReceiver(mMonitor, filter);
24     }
25
26     @Override
27     protected void onDestroy() {
28         super.onDestroy();
29         //取消监视器的注册
30         this.unregisterReceiver(mMonitor);
31     }
32     .....
33 };
```

在代码 8-1 中, 初始化包括两个主要部分: 获取连接管理器实例和注册监视器。其中, 监视器将只接收网络状态改变意向 (第 20 行)。注册监视器后, 当网络状态改变时, 该监视器会获取网络状态改变意向。当应用程序退出时, 需要注销监视器 (第 30 行)。

2. 网络连接监视器

代码 8-2 是代码 8-1 中所参考的网络连接监视器的定义, 该监视器实际上是一个广播接收器组件, 用于接收网络状态改变意向。

代码 8-2 网络连接监视器的定义

文件名: NetMonitorAct.java

```

1  class FooNetMonitor extends BroadcastReceiver {
2      @Override
3      public void onReceive(Context context, Intent intent) {
4          Bundle data = intent.getExtras();
5          //获取当前连接是否可用
6          final String nextra = ConnectivityManager.EXTRA_NO_CONNECTIVITY;
7          boolean unavailable = data.getBoolean(nextra);
8          if(!unavailable) { showInfo(); }
9          else { print("当前无活跃网络！"); checkWifi(); }
10     }
11 };
12
13 private void showInfo() { //显示活跃连接信息
14     NetworkInfo info = mMgr.getActiveNetworkInfo();
15     if(info!=null) { showNetworkInfo(info); }
16 }
17
18 private void showNetworkInfo(NetworkInfo info) { //显示网络信息
19     String name = info.getTypeName();
20     NetworkInfo.State state = info.getState();
21     NetworkInfo.DetailedState dstate = info.getDetailedState();
22     String isAvailable = info.isAvailable()?"可用":"不可用";
23     print(name+"|"+isAvailable+"|"+
24         NetworkUtil.getInstance().getStateDesc(state)+" (" +
25         NetworkUtil.getInstance().getDetailStateDesc(dstate)+" ) ");
26 }
27
28 private boolean checkWifi () { //检查 Wi-Fi 网络
29     NetworkInfo info = mMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
30     if(info.getState() != NetworkInfo.State.CONNECTED) {
31         Toast.makeText(this,"网络未连接, 请启动网络!",Toast.LENGTH_LONG).show();
32         Intent i = new Intent(WifiManager.ACTION_PICK_WIFI_NETWORK);
33         this.startActivity(i);
34         return (false);
35     }
36     return (true);
37 }

```

在代码 8-2 中, 在重载的接收方法中分解意向的扩展数据, 获取网络状态改变信息 (第 4 行)。如果当前连接可用, 则显示活跃连接信息 (第 8 行); 否则将提示无活跃连接并检查 Wi-Fi 网络 (第 9 行)。

使用连接管理器的 `getActiveNetworkInfo` 方法可获取当前活跃的网络信息接口 (第 14 行), 继而使用信息接口的有关方法即可获取该网络的状态信息 (`showNetworkInfo`



方法)。

使用连接管理器的 `getNetworkInfo` 方法可获取指定网络类型的网络信息 (第 29 行是获取 Wi-Fi 网络的信息), 继而通过网络状态判断 Wi-Fi 是否连接 (第 30 行)。如果没有连接, 将启动选择 Wi-Fi 网络的意向, 供用户启动 Wi-Fi 网络 (第 32 行和第 33 行), 其实机界面如图 8-2 所示。



图 8-2 启动 Wi-Fi 网络的实机界面

表 8-3 是 Android 平台所定义的主要网络类型的说明, 这些类型在连接管理器类中定义。

表 8-3 主要网络类型的说明

类 型	说 明
TYPE_MOBILE	移动网络
TYPE_WIFI	Wi-Fi 网络

表 8-4 是 Android 平台所定义的网络状态类型的说明, 这些类型在连接管理器类中定义。

表 8-4 网络状态类型的说明

类 型	说 明
CONNECTED	已连接
CONNECTING	正在连接
DISCONNECTED	已断开
DISCONNECTING	正在断开
SUSPENDED	挂起
UNKNOWN	未知状态

3. 使用许可

对于获取网络连接状态, 需要拥有相应的使用许可, 必须在程序清单中声明获取网络连接状态的使用许可, 其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

8.3.2 Wi-Fi 连接管理

Wi-Fi 管理器 (WifiManager) 用于管理 Wi-Fi 连接, 包括扫描访问点和获取连接信息。图 8-3 所示为使用 Wi-Fi 管理器获取 Wi-Fi 连接信息的实机界面。



图 8-3 获取 Wi-Fi 连接信息的实机界面

1. 应用程序主 Activity 框架

代码 8-3 是 Wi-Fi 应用示例程序的主 Activity 组件的框架定义。

代码 8-3 Wi-Fi 应用示例程序的主 Activity 组件的框架定义

文件名: WifiInfoAct.java

```
1 public class WifiInfoAct extends Activity {
2     //Wi-Fi 管理器
3     private WifiManager mMgr = null;
4     private FooWifiMonitor mMonitor = null;
5
6     @Override
7     public void onCreate(Bundle savedInstanceState) {
8         super.onCreate(savedInstanceState);
9         setContentView(R.layout.main);
10        //初始化
11        init();
12    }
13
14    private void init() {
15        //获取 Wi-Fi 管理器
16        final String sname = Context.WIFI_SERVICE;
```



Android 平台开发之旅 第2版

```

17         mMgr = (WifiManager)(this.getSystemService(sname));
18         //是否启动 Wi-Fi, 如果没有则启动
19         if(!mMgr.isWifiEnabled()) { enableWifi(); }
20         //初始化监视器和意向过滤器
21         mMonitor = new FooWifiMonitor();
22         final String aname = WifiManager.SCAN_RESULTS_AVAILABLE_ACTION;
23         IntentFilter filter = new IntentFilter(aname);
24         //注册监视器
25         this.registerReceiver(mMonitor, filter);
26     }
27
28     private void enableWifi() { //启动 Wi-Fi
29         Intent i = new Intent(WifiManager.ACTION_PICK_WIFI_NETWORK);
30         this.startActivity(i);
31     }
32
33     @Override
34     protected void onDestroy() { super.onDestroy();
35         //取消监视器的注册
36         this.unregisterReceiver(mMonitor);
37     }
38     @Override
39     public boolean onOptionsItemSelected(MenuItem item) {
40         switch(item.getItemId()) {
41             case R.id.mi_scan: { mMgr.startScan(); break; }
42             case R.id.mi_info: { showInfo(); break; }
43         }
44         return super.onOptionsItemSelected(item);
45     }
46     .....
47 };

```

在代码 8-3 中, 初始化包括两个主要部分: 获取 Wi-Fi 管理器实例、启动 Wi-Fi 连接和注册监视器。监视器将只接收扫描结果可用的意向 (第 22 行, 其是系统广播意向)。注册监视器后, 当扫描完毕, 该监视器会获取扫描结果信息。当应用程序退出时, 需要注销监视器 (第 36 行)。

为了保证 Wi-Fi 功能可用, 首先使用 Wi-Fi 管理器的 `isWifiEnabled` 方法可获知 Wi-Fi 状态是否可用, 如果不可用则需要启动 Wi-Fi (第 19 行)。

该示例程序使用选项菜单项启动 Wi-Fi 扫描和获取 Wi-Fi 连接信息 (第 41 行和第 42 行)。其中, 使用 Wi-Fi 管理器的 `startScan` 方法即可启动 Wi-Fi 扫描。

提示: 除了定义扫描状态的广播动作, Wi-Fi 管理器还定义了其他 Wi-Fi 相关的广播动作, 用户程序可以通过广播接收器来接收 Wi-Fi 的相关状态。表 8-5 是 Wi-Fi 管理器所定义的广播动作的说明。

表 8-5 Wi-Fi 管理器所定义的广播动作的说明

广 播 动 作	说 明
NETWORK_IDS_CHANGED_ACTION	网络标识改变
NETWORK_STATE_CHANGED_ACTION	网络状态改变
RSSI_CHANGED_ACTION	接收信号强度改变
SCAN_RESULTS_AVAILABLE_ACTION	扫描结果可用（扫描完毕）
SUPPLICANT_CONNECTION_CHANGE_ACTION	请求连接状态改变（已连接或已断开）
SUPPLICANT_STATE_CHANGED_ACTION	请求状态改变
WIFI_STATE_CHANGED_ACTION	Wi-Fi 状态改变

2. Wi-Fi 扫描监视器

代码 8-4 是代码 8-3 中所参考的 Wi-Fi 扫描状态监视器的定义，该监视器实际上是一个广播接收器组件，用于接收 Wi-Fi 扫描状态意向。

代码 8-4 Wi-Fi 扫描状态监视器的定义

文件名: WifiInfoAct.java

```
1 class FooWifiMonitor extends BroadcastReceiver {
2     @Override
3     public void onReceive(Context context, Intent intent) {
4         Toast.makeText(WifiInfoAct.this, "扫描完毕！ ", Toast.LENGTH_LONG).show();
5         //获取扫描结果
6         List<ScanResult> results = mMgr.getScanResults();
7         if(results != null) {
8             for(int j = 0; j < results.size(); ++j) {
9                 print("==== 搜索信息#" + (j+1) + ": " + getScanResult(results.get(j)));
10            }
11        }
12    }
13
14    //获取搜索结果
15    private String getScanResult(ScanResult scanResult) {
16        StringBuffer sb = new StringBuffer("\nSSID: " + scanResult.SSID);
17        sb.append("\nBSSID: " + scanResult.BSSID);
18        sb.append("\n 性能: " + scanResult.capabilities);
19        sb.append("\n 频率: " + scanResult.frequency + " MHz");
20        sb.append("\n 信号等级: " + scanResult.level + " dBm");
21        return (sb.toString() );
22    }
23 };
```

在代码 8-4 中，当 Wi-Fi 扫描完毕后，使用 Wi-Fi 管理器的 getScanResults 方法即可获得扫描结果信息接口（第 9 行），再通过信息接口的有关方法获取结果内容，其实机界面如图 8-4 所示。

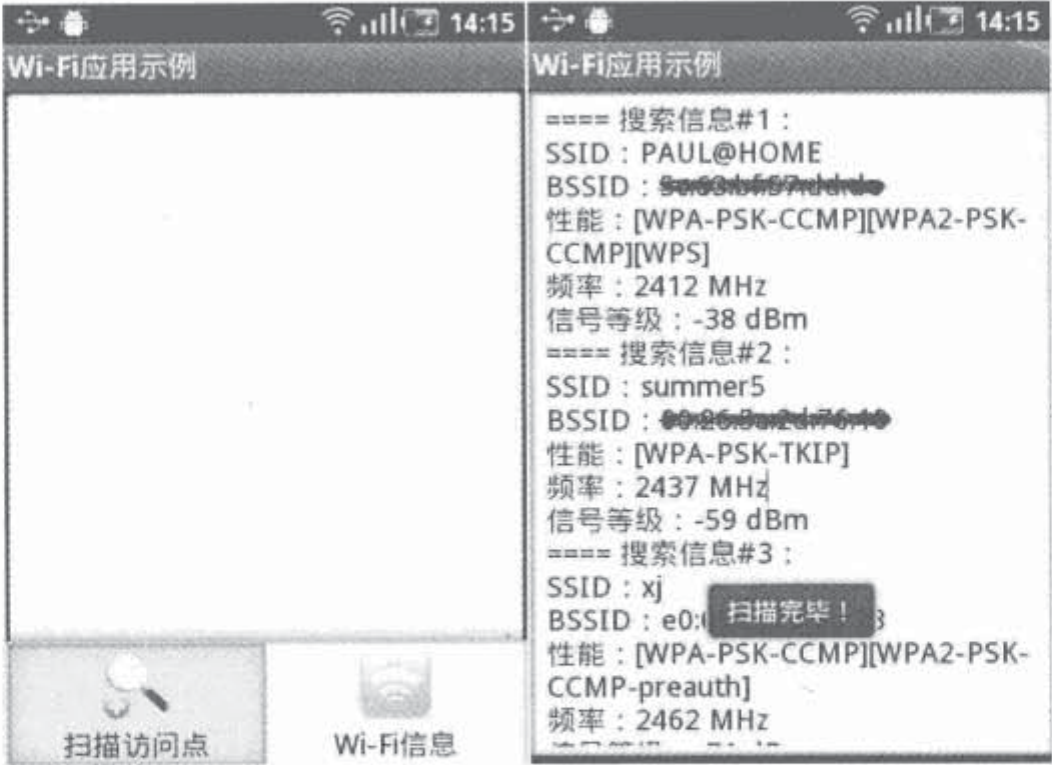


图 8-4 获取 Wi-Fi 扫描结果的实机界面

3. 获取 Wi-Fi 信息

Wi-Fi 信息主要包括配置信息、连接信息和 DHCP 信息，如代码 8-5 所示。

代码 8-5 获取 Wi-Fi 信息

文件名: WifiInfoUtil.java

```
1 public String getInfo(WifiManager mgr) { //获取 WiFi 服务信息
2     StringBuffer sb = new StringBuffer("状态: "+
3         FooTelUtil.getInstance().getWifiStateTypeDesc(mgr.getWifiState()));
4     //配置网络信息
5     List<WifiConfiguration> networks = mgr.getConfiguredNetworks();
6     for(int i = 0; i < networks.size(); ++i) {
7         sb.append("\n==== 配置信息#" + (i+1) + ": " + getConfig(networks.get(i)));
8     }
9     //连接信息
10    WifiInfo info = mgr.getConnectionInfo();
11    if(info != null) { sb.append("\n==== 连接信息 ===== " + getInfo(info)); }
12    //DHCP 信息
13    DhcpInfo info2 = mgr.getDhcpInfo();
14    if(info2 != null) { sb.append("\n==== DHCP 信息 ===== " + getInfo(info2)); }
15    return (sb.toString());
16 }
```

在代码 8-5 中，使用 Wi-Fi 管理器的 `getWifiState` 方法可获得 Wi-Fi 状态（第 3 行）；使用 `getConfiguredNetworks` 方法可获得已配置的网络信息；使用 `getConnectionInfo` 方法可获得 Wi-Fi 连接信息；使用 `getDhcpInfo` 可获得 DHCP 信息。

表 8-6 是 Android 平台所定义的 Wi-Fi 状态类型，这些类型在 Wi-Fi 管理器类中定义。

表 8-6 Wi-Fi 状态类型的说明

类 型	说 明
WIFI_STATE_DISABLED	不可用
WIFI_STATE_DISABLING	正在停用
WIFI_STATE_ENABLED	准备就绪
WIFI_STATE_ENABLING	正在启用
WIFI_STATE_UNKNOWN	未知状态

(1) 获取配置信息

代码 8-6 是使用 Wi-Fi 配置信息接口获取配置信息的实例代码。

代码 8-6 获取 Wi-Fi 配置信息

文件名: WifiInfoUtil.java

1	private String getConfig(WifiConfiguration cfg) {//获取配置信息
2	StringBuffer sb = new StringBuffer("\nSSID: " + cfg.SSID);
3	sb.append("\nBSSID: " + cfg.BSSID);
4	sb.append("\n 网络标识: " + cfg.networkId);
5	return (sb.toString());
6	}

提示：基本服务集（Basic Service Set, BSS）是一种特殊的 Ad-hoc（一种支持点对点访问的无线网络应用模式）局域网应用，一群计算机设定相同的 BSS 标识，就可形成一个组。

Wi-Fi 网络安全访问（Wi-Fi Protected Access, WPA）是一种保护无线网络（Wi-Fi）安全的系统标准；预先共享密钥（Pre-hared Key, PSK）是一种无线网络的安全认证模式。

服务集标识（Service Set Identifier, SSID）用于标识无线局域网，不同 SSID 的无线网络是无法进行互访的。

(2) 获取连接信息

代码 8-7 是使用 Wi-Fi 配置信息接口获取 Wi-Fi 连接信息的实例代码。

代码 8-7 获取 Wi-Fi 信息

文件名: WifiInfoUtil.java

1	private String getInfo(WifiInfo inf) {//获取 Wi-Fi 连接信息
2	StringBuffer sb = new StringBuffer("\nSSID: "+inf.getSSID());
3	sb.append("\nBSSID:"+inf.getBSSID());
4	sb.append("\n 网络标识: "+inf.getNetworkId());
5	sb.append("\nIP 地址: "+intToIP(inf.getIpAddress()));
6	sb.append("\n 链接速率: "+inf.getLinkSpeed()+" Mbps");
7	sb.append("\nMAC 地址: "+inf.getMacAddress());
8	sb.append("\n 接收信号强度: "+inf.getRssi()+" dBm");
9	return (sb.toString());
10	}



Android 平台开发之旅 第2版

提示：接收信号强度指示（Received Signal Strength Indication, RSSI）用来判定连接质量，以及是否增大无线发送强度。

动态主机配置协议（Dynamic Host Configuration Protocol, DHCP）主要用于给内部网络的主机自动分配 IP 地址（路由功能）。

（3）获取 DHCP 信息

代码 8-8 是使用 DHCP 信息接口获取 Wi-Fi DHCP 信息的实例代码。

代码 8-8 获取 DHCP 信息

文件名：WifiInfoUtil.java

```

1  private String getInfo(DhcpInfo inf) { //获取 DHCP 信息
2      StringBuffer sb = new StringBuffer("\nDNS1: "+intToIP(inf.dns1));
3      sb.append("\nDNS2: "+intToIP(inf.dns2));
4      sb.append("\n 网关: "+intToIP(inf.gateway));
5      sb.append("\nIP 地址: "+intToIP(inf.ipAddress));
6      sb.append("\n 服务地址: "+intToIP(inf.serverAddress));
7      return (sb.toString() );
8  }

```

4. 使用许可

为了获取 Wi-Fi 状态，需要拥有获取 Wi-Fi 状态的使用许可，即必须在程序清单中声明访问 Wi-Fi 状态的使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
```

8.4 网页浏览器

在 HTTP 通信中，3 种方式获取到的都仅仅是文本内容，而没办法做到像浏览器一样对网页的内容进行可视化展示。幸运的是，Android 平台将 WebKit 引擎植入系统，通过该引擎，应用程序可方便地将网页内容集成到用户界面中。

8.4.1 WebKit 介绍

WebKit (<http://webkit.org/>) 是一个开源网页浏览器引擎，同时也是 Mac OS X 的 Safari 网页浏览器的核心。WebKit 的 HTML 和 JavaScript 引擎分别源于 KDE 组织 (<http://kde.org/>) 的 KHTML 和 KJS 库。

在应用于 Mac OSX 上的 Safari 之后，WebKit 很快被广泛地移植到其他系统平台：Google Chrome 浏览器就是采用了 WebKit 引擎，Android 平台自带的浏览器也是基于 WebKit 内核。WebKit 以其开源的优势以及高速的网页加载效果赢得越来越多开发者的青睐。

8.4.2 Android 平台对 WebKit 引擎的封装

实际上，Android 平台对 WebKit 引擎进行了包装，读者无需对 WebKit 引擎的细节了解

更多就可使用引擎的功能。Android 平台定义了 android.webkit 包用于提供显示网页的工具。表 8-7 是该包中重要的类/接口的说明。

表 8-7 WebKit 包中重要的类/接口的说明

类/接口	说 明
DownloadListener	用于侦听下载事件
WebBackForwardList	用于包含网页视图向后/向前的列表
WebChromeClient	Chrome 浏览器客户端，代表网页处理的客户端
WebHistoryItem	代表向前/先后的列表项
WebSettings	管理一个网页视图的设置状态
WebView	网页视图，用于显示网页
WebView.HitTestResult	代表网页上锚点的单击测试结果
WebViewClient	常规浏览器客户端，代表网页处理的客户端
WebViewFragment	网页视图片段组件

8.4.3 网页视图（WebView）应用

图 8-5 是一款简单的浏览器，其在用户程序中嵌入网页视图并显示页面内容。



图 8-5 简单浏览器程序的实机界面

1. 主界面布局定义

代码 8-9 是图 8-5 所示的简单浏览器程序的实机界面布局定义，其主要组件是网页视图。

代码 8-9 网页视图界面布局定义

文件名：main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent" android:layout_height="fill_parent">
5     <TableLayout android:stretchColumns="1"
6         android:layout_width="fill_parent" android:layout_height="wrap_content">
7         <TableRow>
```




Android 平台开发之旅 第2版

```

8         <TextView android:text="地址"
9             android:layout_width="wrap_content" android:layout_height="wrap_content"/>
10        <EditText android:id="@+id/text_url" android:singleLine="true"
11            android:layout_width="wrap_content" android:layout_height="wrap_content"/>
12        <ImageView android:id="@+id/go" android:layout_width="wrap_content"
13            android:layout_height="wrap_content" android:layout_gravity="center_vertical"
14            android:src="@drawable/go" android:scaleType="matrix"/>
15    </TableRow>
16 </TableLayout>
17 <WebView android:id="@+id/web_view" android:layout_width="fill_parent"
18     android:layout_height="wrap_content" android:layout_weight="0.96" />
19 <TextView android:id="@+id/text_status" android:layout_width="fill_parent"
20     android:layout_height="wrap_content" android:text="就绪" />
21 </LinearLayout>

```

在代码 8-9 中，通过<WebView>标记对网页视图组件进行定义（第 17 行），其对应组件对象类型与其标记名相同。

2. 应用程序主 Activity 框架

作为视图组件，在主 Activity 组件对布局资源填充完毕之后，即可获取布局资源中所定义的网页视图组件实例；继而通过网页视图的有关接口进行功能设置或网页事件的回调。代码 8-10 是简单浏览器程序的主 Activity 组件的初始化代码。

代码 8-10 简单浏览器程序的主 Activity 组件的初始化

文件名: FooBrowserAct.java

```

1  public class FooBrowserAct extends Activity implements DownloadListener, OnClickListener {
2      //网页视图、主线程消息队列处理器
3      private WebView mWebView = null;
4      private Handler mHandler = null;
5      private EditText mTextUrl = null;
6      private TextView mTextStatus = null;
7
8      @Override
9      public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.main);
12         //获取布局中控件对象
13         mTextUrl = (EditText) findViewById(R.id.text_url);
14         mTextStatus = (TextView) findViewById(R.id.text_status);
15         ImageView btnGo = (ImageView) findViewById(R.id.go);
16         btnGo.setOnClickListener(this);
17         //获取网页视图控件
18         mWebView = (WebView) findViewById(R.id.web_view);
19         mWebView.getSettings().setJavaScriptEnabled(true);
20         mWebView.setWebChromeClient(new WebChromeClient() { //Chrome 客户端
21             @Override

```



```

22         public void onProgressChanged(WebView view, int progs) {
23             if(progs != 100) { mTextStatus.setText("加载: "+progs+"%"); }
24             else { mTextStatus.setText("加载完毕"); }
25         }
26     });
27     mWebView.setWebViewClient(new WebViewClient() { //客户端
28         @Override
29         public boolean shouldOverrideUrlLoading(WebView view, String url) {
30             view.loadUrl(url);
31             return (true);
32         }
33     });
34
35     //设置下载侦听器
36     mWebView.setDownloadListener(this);
37     //设置上下文菜单
38     this.registerForContextMenu(mWebView);
39
40     mHandler = new Handler() { //初始化界面线程消息处理器
41         public void handleMessage(Message msg) {
42             Bundle bundle = msg.getData();
43             String msgStr = bundle.getString(IConfig.EXTRA);
44             Toast.makeText(FooBrowserAct.this, msgStr, Toast.LENGTH_LONG).show();
45         }
46     };
47     //加载指定页面内容
48     mTextUrl.setText(IConfig.HOME_URL);
49     mWebView.loadUrl(IConfig.HOME_URL);
50 }
51 .....
52 };

```

在代码 8-10 中，首先使用网页视图的资源 ID 来获取网页视图实例（第 18 行），然后通过其设置接口来设置允许 JavaScript（第 19 行）。在第 20 行中，使用 `setWebChromeClient` 方法设置网页视图的 Chrome 浏览器客户端，用于对加载进度改变事件的回调（第 22 行）；在第 27 行中，使用 `setWebViewClient` 方法来设置网页视图的客户端，用于重载网页 URL 的加载方法（第 29 行）。

在第 36 行中，给网页视图指定下载侦听器，用于捕获网页视图的下载请求；在第 38 行中，给网页视图添加上下文菜单，用于下载操作区域的链接资源；在第 40 行中，为了获取下载线程的状态，给下载线程提供了主线程消息队列的处理器。

最后，使用网页视图的 `loadUrl` 方法（第 49 行）加载指定网页。由于需要网页，所以需要拥有访问互联网的使用许可，即需要在程序清单中声明该许可，其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.INTERNET"/>
```



3. 应用程序配置信息接口

代码 8-11 是应用程序中有关配置信息接口的定义, 包括主页 URL、截屏保存文件夹、下载文件夹和消息数据项。

代码 8-11 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String HOME_URL = "http://www.webkit.org/";
3      public static final String CAPTURE_PATH = "/sdcard/cap.data";
4      public static final String DOWNLOAD_DIR = "/sdcard/Download";
5      public static final String EXTRA = "_msg";
6      public static final String CAPTURE = "_capture";
7  };

```

4. 网页视图功能

通过代码 8-10 可看出, 网页视图的功能展示主要通过 3 种方式: 选项菜单、外部接口和上下文菜单。

(1) 选项菜单

图 8-6 是简单浏览器程序的选项菜单的实机界面, 按照从左到右, 从上到下的顺序, 所定义的选项分别是回退浏览、向前浏览、返回主页、刷新页面、截屏和收藏。



图 8-6 选项菜单的实机界面

在图 8-6 中, 前 4 个选项以图标形式显示在操作栏中, 而后两项以文字的形式显示在菜单下拉列表中。代码 8-12 是程序选项菜单的资源定义。

代码 8-12 选项菜单的资源定义

文件名: opt_menu.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/android">
3      <group>
4          <item android:id="@+id/mi_back" android:icon="@drawable/back"
5              android:title="回退" android:showAsAction="ifRoom" />
6          <item android:icon="@drawable/forward" android:id="@+id/mi_forward"
7              android:title="向前" android:showAsAction="ifRoom" />
8          <item android:icon="@drawable/home" android:id="@+id/mi_home"
9              android:title="主页" android:showAsAction="ifRoom" />
10         <item android:icon="@drawable/reload" android:id="@+id/mi_reload"
11             android:title="刷新" android:showAsAction="ifRoom" />

```



```
12      </group>
13      <item android:icon="@drawable/capture" android:id="@+id/mi_capture"
14            android:title="截屏" android:showAsAction="ifRoom|withText" />
15      <item android:icon="@drawable/favorite" android:id="@+id/mi_favorite"
16            android:title="收藏" android:showAsAction="ifRoom|withText" />
17  </menu>
```

在代码 8-12 中，将菜单项的前 4 项作为一个组，后两项为单独项。其中，前 4 项条目的 showAsAction 属性设置为 ifRoom（第 5 行），即可让其在操作栏占据立足之地；后两项条目的 showAsAction 属性设置为 ifRoom|withText（第 14 行），其本意是在操作栏上既显示图标又显示文本，但实际上该两项被挤入到选项菜单列表中。

根据作者的试验，在操作栏中最多只能显示 4 项菜单条目，并不是可任意设置。

表 8-8 是对选项菜单所调用的、网页视图的重要方法的说明。

表 8-8 网页视图的重要方法的说明

重 要 方 法	说 明
Picture capturePicture()	截取当前视图内容并保存到图片对象
void goBack()	回退并显示历史页面的内容
void goForward()	向前并显示历史页面的内容
void loadUrl(String)	加载指定 URL 的网页
void reload()	重载当前网页
void pageDown(boolean)	向下滚动并显示网页内容
void pageUp(boolean)	向上滚动并显示网页内容

代码 8-13 是选项菜单项选取事件的回调处理。

代码 8-13 选项菜单项选取回调处理

文件名: FooBrowserAct.java

```
1  @Override
2  public boolean onOptionsItemSelected(MenuItem item) {
3      switch(item.getItemId()) {
4          case R.id.mi_back: { mWebView.goBack(); //回退浏览
5                               mTextUrl.setText(mWebView.getUrl()); break; }
6          case R.id.mi_forward: { mWebView.goForward(); //前进浏览
7                                  mTextUrl.setText(mWebView.getUrl()); break; }
8          case R.id.mi_home: { //跳转到主页
9                               mTextUrl.setText(IConfig.HOME_URL);
10                              mWebView.loadUrl(IConfig.HOME_URL); break; }
11          case R.id.mi_reload: { mWebView.reload(); break; } //刷新
12          case R.id.mi_favorite: { //添加当前页到收藏夹
13                                  Browser.saveBookmark(this, mWebView.getTitle(), mWebView.getUrl());
14                                  break;
15          }
16          case R.id.mi_capture: { //保存网页缩略图
```



```
17         Picture picture = mWebView.capturePicture();
18         savePictureToSDCard(picture); break;
19     }
20 }
21 return super.onOptionsItemSelected(item);
22 }
```

(2) 外部接口

除了直接功能外，网页视图还通过一些关联接口对其功能状态进行管理、处理网页事件、处理下载请求以及数据管理等，如图 8-7 所示。

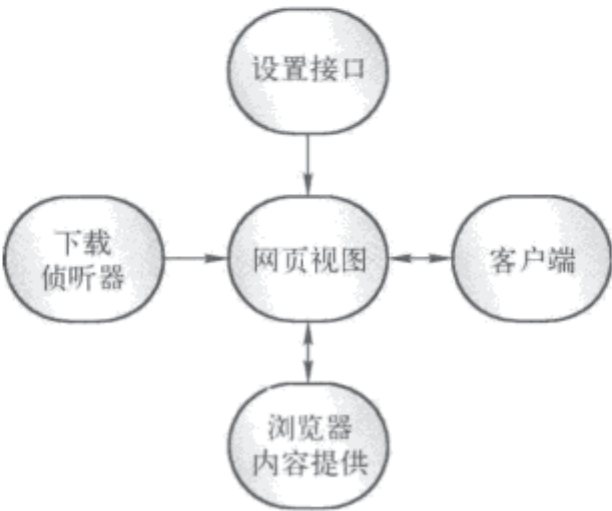


图 8-7 网页视图的关联接口

表 8-9 是网页视图进行接口关联的重要方法的说明。

表 8-9 网页视图接口关联的重要方法的说明

重要方法	说 明
WebSettings getSettings()	获取网页视图设置控制的接口
String getUrl()	获取网页视图当前的 URL
WebView.HitTestResult getHitTestResult()	获取网页视图当前单击测试的结果
void setWebChromeClient(WebChromeClient)	设置 Chrome 客户端
void setWebViewClient(WebViewClient)	设置常规客户端

(3) 上下文菜单

图 8-8 是当用户在页面图片区域触发上下文菜单时，弹出的下载该图片资源的实机界面。

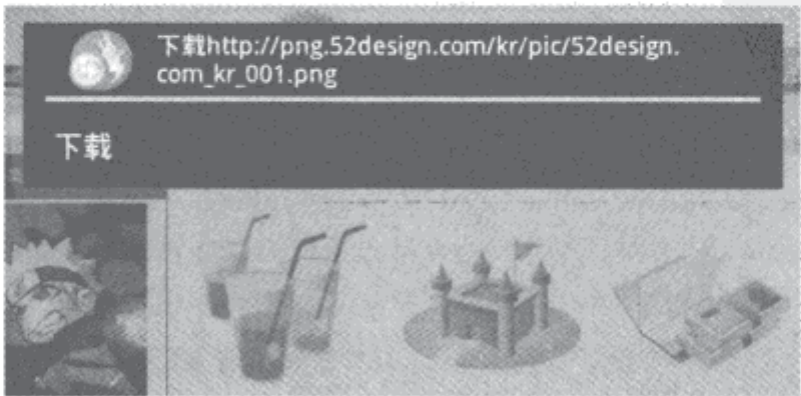


图 8-8 上下文菜单的实机界面

代码 8-14 是网页视图组件的上下文菜单的资源定义。

代码 8-14 网页视图组件的上下文菜单的资源定义

文件名: context_menu.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <menu xmlns:android="http://schemas.android.com/apk/res/android">
3     <item android:id="@+id/mi_download" android:title="下载"/>
4 </menu>
```

代码 8-15 是上下文菜单项选取事件的回调处理。

代码 8-15 上下文菜单项选取回调处理

文件名: FooBrowserAct.java

```
1 @Override
2 public boolean onContextItemSelected(MenuItem item) {
3     switch(item.getItemId()) {
4         case R.id.mi_download: {//获取单击测试结果
5             HitTestResult htr = this.mWebView.getHitTestResult();
6             //获取 URL 并下载资源
7             String url = htr.getExtra();
8             if(url != null) { downloadUrl(url); }
9             else { return(false); }
10        }
11    }
12    return super.onContextItemSelected(item);
13 }
```

在代码 8-15 中, 使用网页视图的 `getHitTestResult` 方法获取单击测试结果的接口 (第 5 行), 继而通过该接口的内容访问方法 `getExtra` 获取单击对象的 URL (第 7 行), 最后依据该 URL 启动下载过程 (第 8 行)。

5. 网页浏览控制

(1) 重载返回键事件处理

从表 8-8 可知, 通过网页视图的 `goBack`、`goForward`、`loadUrl`、`reload` 等方法可实现对网页内容的浏览控制。不仅如此, 为了防止用户单击返回键而退出程序, Activity 组件必须重载按键事件, 改变返回键的默认操作, 如代码 8-16 所示。

代码 8-16 重载返回键事件处理

文件名: FooBrowserAct.java

```
1 @Override
2 public boolean onKeyDown(int keyCode, KeyEvent event) {
3     if( (keyCode == KeyEvent.KEYCODE_BACK) && mWebView.canGoBack() ) {
4         mWebView.goBack();
5         return (true);
6     }
```




```
7         return super.onKeyDown(keyCode, event);
8     }
```

在代码 8-16 中，当 Activity 组件接收到返回键事件时，其将执行回退浏览网页的操作（第 4 行），而不是退出。

(2) 设置自动加载链接

在默认情况下，当用户单击网页链接时，网页视图不会自动载入链接内容，但网页视图会将该请求发给其“幕后主使”——视图客户端（Client）。在客户端的重载网页载入的方法（shouldOverrideUrlLoading，代码 8-10 中第 29 行）中，将所请求的链接信息分派给网页视图进行加载，从而实现网页浏览的连续。

6. 网页视图设置接口（WebSettings）

在代码 8-10 中，通过网页视图的 getSettings 方法即可获取其设置接口，通过该接口可管理网页视图的设置状态，如是否允许文件访问、是否内建缩放控件、是否支持 JavaScript 等。表 8-10 是网页视图设置接口的重要方法的说明。

表 8-10 网页视图设置接口的重要方法的说明

重要方法	说明
void setBlockNetworkImage(boolean)	是否阻塞网络图片的接收（网页上将不显示图片）
void setBuiltInZoomControls(boolean)	是否将缩放控件内建到网页视图中
void setDefaultFontSize(int)	设置默认的字体大小
void setDefaultTextEncodingName(String)	设置默认的文本编码名称
void setJavaScriptEnabled(boolean)	设置是否支持 JavaScript
void setStandardFontFamily(String)	设置标准的字体

7. 网页视图客户端（WebViewClient）

网页视图的客户端可为网页视图的宿主 Activity 组件提供网页事件处理的“后门”，如网页加载完毕、开始加载网页、重载按键事件、重载 URL 加载等。表 8-11 是网页视图客户端的重要接口的说明。

表 8-11 网页视图客户端的重要接口的说明

重要接口	说明
void onLoadResource(WebView, String)	通知宿主应用程序将载入指定资源
void onPageFinished(WebView, String)	通知宿主应用程序网页载入完成
void onPageStarted(WebView, String, Bitmap)	通知宿主应用程序开始加载指定页面
boolean shouldOverrideKeyEvent(WebView, KeyEvent)	提供重载按键事件的接口
boolean shouldOverrideUrlLoading(WebView, String)	提供重载加载 URL 的接口

从表 8-11 可知，代码 8-10 中第 29 行的 shouldOverrideUrlLoading 方法就是为 Activity 组件提供的、用于改变加载处理的“后门”。例如，用户在地址栏输入的地址为 foo，那么就可在该方法中利用域名的命名规则将原地址填补为 <http://www.foo.org/>。

8. 网页视图 Chrome 客户端 (WebChromeClient)

Chrome 客户端可视为对网页视图客户端的补充。网页视图客户端主要关注网页内容的展示，而网页视图 Chrome 客户端主要关注浏览器用户界面的发生，如加载进度更新或弹出 JavaScript 的警告。表 8-12 是网页视图 Chrome 客户端的重要接口的说明。

表 8-12 网页视图 Chrome 客户端的重要接口的说明

重 要 接 口	说 明
boolean onJsAlert(WebView, String, String, JsResult)	显示一个 JavaScript 警告对话框
boolean onJsConfirm(WebView, String, String, JsResult)	显示一个 JavaScript 确认对话框
void onProgressChanged(WebView, int)	载入页面的当前进度

从表 8-12 可知，代码 8-10 中第 22 行的 onProgressChanged 方法用于获取网页的加载进度，如图 8-9 所示。



图 8-9 更新网页加载进度信息的实机界面

9. 网页下载控制

在浏览网页的过程中，有两种情形可触发下载：第一种是单击特定文件的链接，如可执行程序或数据文件等；第二种是通过上下文菜单下载当前的资源链接，如图 8-8 中下载图片资源。第一种情形的下载信息可通过下载侦听器 (DownloadListener) 进行捕获；第二种情形的下载信息只能通过单击测试结果 (HitTestResult) 进行获取。

代码 8-17 是网页视图所绑定的下载侦听器的回调处理。

代码 8-17 下载侦听器回调处理

文件名: FooBrowserAct.java

```
1  @Override
2  public void onDownloadStart(String url, String userAgent,
3      String contentDisposition, String mimetype, long contentLength) {
4      Toast.makeText(this, "开始下载【"+url+"】", Toast.LENGTH_SHORT).show();
5      //下载文件
6      downloadUrl(url);
7  }
8
9  private void downloadUrl(String url) { //下载指定链接资源
10     DownloadThread t = new DownloadThread(url, mHandler);
11     t.start();
12 }
```




Android 平台开发之旅 第2版

在代码 8-17 中，实际下载过程通过线程来完成。代码 8-18 是该下载线程的定义。

代码 8-18 下载线程的定义

文件名: DownloadThread.java

```

1  public class DownloadThread extends Thread {
2      private String mUrl = null;
3      private Handler mHandler = null;
4
5      public DownloadThread(String url, Handler handler) {
6          this.mUrl = url; this.mHandler = handler;
7      }
8
9      @Override
10     public void run() {
11         try { //生成 URL 对象，用于建立 URL 连接
12             URL url = new URL(mUrl);
13             String fileName = url.getFile();
14             //将路径名替换成下划线（防止文件名重名）
15             fileName = fileName.replace(File.separatorChar, '_');
16             //建立 URL 连接
17             URLConnection conn = url.openConnection();
18             InputStream is = conn.getInputStream();
19             BufferedInputStream bis = new BufferedInputStream(is);
20             //创建文件输出流
21             FileOutputStream fos =
22             new FileOutputStream(IConfig.DOWNLOAD_DIR+File.separatorChar+fileName);
23             BufferedOutputStream bos = new BufferedOutputStream(fos);
24             //下载文件
25             int aByte = -1;
26             while( (aByte=bis.read()) != -1) { bos.write(aByte); }
27             //关闭输出流
28             bos.close(); fos.close();
29             //关闭输入流
30             bis.close(); is.close();
31             //反馈下载结果
32             FooSysUtil.getInstance().postMsg(mHandler, IConfig.EXTRA,
33             "保存"+IConfig.DOWNLOAD_DIR+File.separatorChar+fileName + "成功! ");
34         } catch (MalformedURLException e) { e.printStackTrace(); }
35         catch (IOException e) { e.printStackTrace(); }
36     }
37 };

```

在代码 8-18 中，下载线程通过 URL 建立 URL 连接（第 17 行），通过 I/O 流读取指定资源的字节并同时写入到本地存储器文件中，从而实现下载。当下载过程完成，通过主线程的消息队列处理器将下载状态发送给主线程（第 32 行）。

注意：下载资源需要在 SD 卡上创建文件或文件夹，所以在程序清单文件中必须声明允许写外部存储器（SD 卡）的许可，其声明代码如下所示。

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

当文件下载完毕，通过主线程的消息队列处理接口向主线程发送消息，再由主线程将该消息通过提示框的方式进行显示。

提示：Android 3 平台提供了下载管理器用于下载管理，开发者无需再通过下载线程以 URL 连接的方式来进行下载。下载管理器可以根据资源的 URI 执行下载动作，并对多个下载任务进行排队；当下载完成后，还会将下载信息记录到系统数据库中。有关下载管理器的介绍可以参考 8.5.3 节。

10. 网页视图截屏

在代码 8-13 中第 17 行，通过网页视图的 `capturePicture` 方法可获取当前网页视图内容的图像数据。代码 8-19 是将该图像数据保存到本地存储设备中并以缩略图的方式进行预览的主要代码。

代码 8-19 保存网页视图并预览

文件名：FooBrowserAct.java

```
1 private void savePictureToSDCard(Picture picture) { //保存图片到 SD 卡
2     try { File file = new File(IConfig.CAPTURE_PATH);
3         OutputStream stream = new FileOutputStream(file);
4         picture.writeToStream(stream);
5         stream.flush();
6         stream.close();
7         //将保存路径通过 intent 发送给截屏显示 Activity
8         Intent intent = new Intent(this, CaptureAct.class);
9         intent.putExtra(IConfig.CAPTURE, IConfig.CAPTURE_PATH);
10        this.startActivity(intent);
11    } catch(IOException e) { e.printStackTrace(); }
12 }
```

在代码 8-19 中，通过图片接口（Picture）的 `writeToStream` 方法将图像内容保存到 SD 卡中（第 4 行），同时启动图片显示 Activity 组件来显示图像内容。代码 8-20 是图片显示 Activity 组件的定义。

代码 8-20 图片显示 Activity 组件的定义

文件名：PictureViewerAct.java

```
1 public class PictureViewerAct extends Activity {
2     @Override
3     protected void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         //获取 intent 传来的文件存储路径信息
```




Android 平台开发之旅 第2版

```

6          Intent intent = this.getIntent();
7          Bundle bundle = intent.getExtras();
8          String filePath = bundle.getString(IConfig.CAPTURE);
9          //设置内容视图
10         ImageView view = new ImageView(this);
11         view.setSource(filePath);
12         this.setContentView(view);
13     }
14 };

```

在代码 8-20 中，图片显示 Activity 组件使用图片视图来展示图像文件（第 10 行）。代码 8-21 是该图片视图的定义。

代码 8-21 显示网页视图缩略图

文件名: `ImageView.java`

```

1  public class ImageView extends View {
2      //图片对象
3      private Picture mPic = null;
4
5      public ImageView(Context context) { super(context); }
6      public void setSource(String filePath) { //设置数据源
7          try { mPic = Picture.createFromStream(new FileInputStream(new File(filePath)));
8              } catch (FileNotFoundException e) { e.printStackTrace(); }
9      }
10
11     @Override
12     public void draw(Canvas canvas) { Rect rect = canvas.getClipBounds();
13         int width = rect.width();
14         int height = rect.height();
15         //非全屏显示缩略图（按 3/4 的比例）
16         rect.left += width/8;
17         rect.top += height/8;
18         rect.right = width*7/8;
19         rect.bottom = height*7/8;
20         //绘制图片
21         canvas.drawPicture(mPic, rect);
22         super.draw(canvas);
23     }
24 };

```

在代码 8-21 中，通过图片接口的 `createFromStream` 方法又将网页视图图片文件载入成图片实例（第 7 行），然后在 View 的绘制函数（第 12 行的 `draw` 函数）中对该图片对象进行绘制（第 21 行）。图 8-10 是显示网页缩略图的界面。



图 8-10 网页缩略图的界面

11. 添加网页到收藏夹

在代码 8-13 中第 13 行，使用浏览器类的 `saveBookmark` 方法可将当前页的地址添加到收藏夹中，其代码如下所示。

```
Browser.saveBookmark(this, mWebView.getTitle(), mWebView.getUrl());
```

图 8-11 是添加当前页到收藏夹所弹出的对话框。

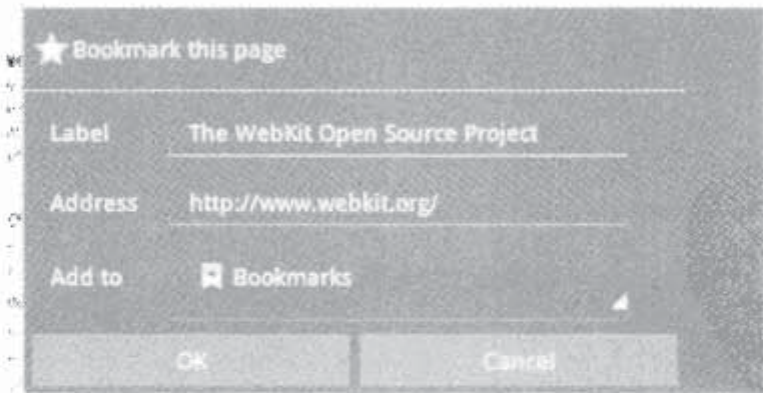


图 8-11 添加当前页到收藏夹

注意：添加当前页到收藏夹需要在程序清单文件中声明允许写历史书签的许可，其声明代码如下所示。

```
<uses-permission
    android:name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS"/>
```



8.5 浏览器信息管理

在 Android 平台的内容提供者包（`android.provider`）中提供了浏览器信息提供接口（`Browser`），通过该接口可获取浏览器的浏览书签和搜索历史信息。

代码 8-22 是浏览器信息的规格定义，包括书签和搜索历史数据列。

代码 8-22 浏览器信息的规格定义

文件名: `IDataSpec.java`

```
1 public interface IDataSpec {
2     public static final String[] BOOKMARK_COLS= { //书签信息的列集合
3         Browser.BookmarkColumns.TITLE, //主题
4         Browser.BookmarkColumns.URL, //地址
5         Browser.BookmarkColumns.DATE //添加书签的日期
6     };
7
8     public static final String[] SEARCHING_COLS= { //搜索信息的列集合
9         Browser.SearchColumns.DATE, //所搜日期
10        Browser.SearchColumns.SEARCH //搜索关键字内容
11    };
12 }
```

8.5.1 浏览书签信息

1. 获取浏览器书签

图 8-12 所示为浏览器中所添加的书签信息（被添加到收藏夹的网页信息）。



图 8-12 获取书签信息

代码 8-23 是图 8-12 所示程序的主 Activity 组件的定义。

代码 8-23 浏览器信息管理程序的主 Activity 组件的定义

文件名: `BrowserCollectorAct.java`

```
1 public class BrowserCollectorAct extends Activity {
2     //内容解析器
3     private LinearLayout mLayout = null;
4     private ContentResolver mResolver = null;
```



```

5
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.main);
10         //获取容器布局
11         this.mLayout = (LinearLayout) findViewById(R.id.layoutPanel);
12         this.mLayout.setOrientation(LinearLayout.VERTICAL);
13         //获取内容解析器
14         this.mResolver = this.getContentResolver();
15     }
16     @Override
17     public boolean onCreateOptionsMenu(Menu menu) {
18         this.getMenuInflater().inflate(R.menu.opt_menu, menu);
19         return super.onCreateOptionsMenu(menu);
20     }
21     @Override
22     public boolean onOptionsItemSelected(MenuItem item) {
23         switch(item.getItemId()) {
24             case R.id.mi_bookmark: { getBookmarks(); break; }
25             case R.id.mi_bookmark_add: { addBookmark(); break; }
26             case R.id.mi_searching: { getSearchings(); break; }
27             case R.id.mi_searching_add: { addSearching(); break; }
28         }
29         return super.onOptionsItemSelected(item);
30     }
31     .....
32 };

```

在代码 8-23 中, 使用 Activity 组件的内容解决者来获取浏览器信息内容 (第 14 行); 通过选项菜单来触发信息的获取。代码 8-24 是获取浏览器浏览书签的主要代码。

代码 8-24 获取浏览器浏览书签

文件名: BrowserCollectorAct.java

```

1      private void getBookmarks() { //获取浏览书签
2          Cursor c = mResolver.query(Browser.BOOKMARKS_URI,
3                                     IDataSpec.BOOKMARK_COLS,
4                                     "("+IDataSpec.BOOKMARK_COLS[2]+" not null)",
5                                     null, Browser.BookmarkColumns.DATE+" DESC");
6          //使用简单表格视图显示数据内容
7          FooTableViewController wrapper = new FooTableViewController(this, c);
8          //替换列显示内容
9          ArrayList<FooTableCellView> views = wrapper.getCellsViews();
10         for(int i = 2; i < views.size(); i += IDataSpec.BOOKMARK_COLS.length) {
11             long epoch = Long.parseLong(views.get(i).getText().toString());
12             views.get(i).setText(FooSysUtil.getInstance().unixTsp2Str(epoch));

```




```
13     }
14     //初始化列抬头文字与字段名映射
15     initColumnTitleMapping1(wrapper);
16     this.mLayout.addView(wrapper.getMainWidget());
17     //关闭游标
18     c.close();
19 }
```

在代码 8-24 中，使用内容解决器的 `query` 方法对书签信息的资源标识进行查询，获取记录游标（第 2 行），再通过简单表格视图组件（`FooTableViewWrapper`）将记录游标中的内容展现在可视组件中（第 7 行）。

在图 8-12 中，日期列所显示的数值实际上是该时点基于系统基准（1970-1-1）所经过的秒数。图 8-13 所示为将该数值转换成常规的日期时间形式的结果。

浏览器信息示例		
抬头	地址	日期
Android.com	http://www.android.com/	2011-07-18 21:57:32
The WebKit Open Source Project	http://www.webkit.org/	2011-07-18 21:56:14
我的主页_腾讯微博	http://t.qq.com/foolstudio	2011-07-18 21:52:35

图 8-13 将日期数值转换成常规的日期时间形式

代码 8-25 是将添加书签日期的数值形式转换成常规的日期时间形式的主要代码。

代码 8-25 书签日期转换

文件名: `FooSysUtil.java`

```
1 public String unixTsp2Str(long epoch) { //将 UNIX 时间戳转换成日期时间字符串
2     SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
3     sdf.setTimeZone(TimeZone.getTimeZone("GMT+08:00"));
4     Date theDate = new Date(epoch);
5     return (sdf.format(theDate));
6 }
```

2. 添加浏览书签

添加浏览书签已经在简单浏览器程序中进行了介绍，不再赘述。

3. 调用浏览器

通过意向对象（`Intent`）的查看动作（`ACTION_VIEW`）可调用系统的浏览器工具打开指定的网页，如代码 8-26 所示。

代码 8-26 浏览指定的网页

文件名: `BrowserCollectorAct.java`

```
1 private void addBookmark() { //浏览指定的网页
2     //(2)浏览指定网页
```



```

3      Uri uri = Uri.parse("http://t.qq.com/foolstudio");
4      Intent intent = new Intent(Intent.ACTION_VIEW, uri);
5      this.startActivity(intent);
6      //(1)添加浏览书签
7      //Browser.saveBookmark(this, "简单工作室", "http://t.qq.com/foolstudio");
8  }

```

4. 使用许可声明

对于读取/添加浏览书签信息，必须在程序清单中声明读取/写入书签信息的使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```

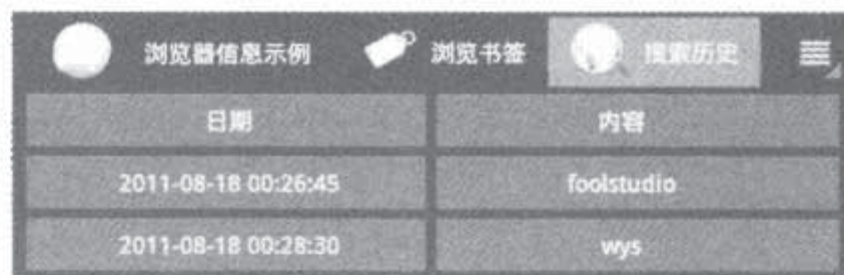
<uses-permission
    android:name="com.android.browser.permission.READ_HISTORY_BOOKMARKS"/>
<uses-permission
    android:name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS"/>

```

8.5.2 搜索历史记录

1. 获取搜索历史记录

图 8-14 所示为当前浏览器的搜索历史记录。



日期	内容
2011-08-18 00:26:45	foolstudio
2011-08-18 00:28:30	wys

图 8-14 浏览器的搜索历史记录

代码 8-27 是图 8-14 所示程序中获取当前浏览器搜索历史记录的主要代码。

代码 8-27 获取搜索历史记录

文件名：BrowserCollectorAct.java

```

1  private void getSearchings() { //获取搜索历史记录
2      Cursor c = mResolver.query(Browser.SEARCHES_URI,
3                                IDataSpec.SEARCHING_COLS,
4                                null, null,
5                                Browser.SearchColumns.DATE+" DESC");
6      //使用简单表格视图显示数据内容
7      FooTableWrapper wrapper = new FooTableWrapper(this, c);
8      //替换列内容
9      ArrayList<FooTableCellView> views = wrapper.getCellsViews();
10     for(int i = 0; i < views.size(); i += IDataSpec.SEARCHING_COLS.length) {
11         long epoch = Long.parseLong(views.get(i).getText().toString());
12         views.get(i).setText(FooSysUtil.getInstance().unixTsp2Str(epoch));

```



```

13     }
14     initColumnTitleMapping2(wrapper);
15     this.mLayout.addView(wrapper.getMainWidget());
16     //关闭游标
17     c.close();
18 }

```

在代码 8-27 中，使用内容解决器的 `query` 方法对搜索历史信息的资源标识进行查询，获取记录游标（第 2 行），再通过简单表格视图组件将记录游标中的内容展现在可视组件中（第 7 行）。

2. 添加搜索关键字

调用浏览器信息接口（`Browser`）的 `addSearchUrl` 方法可以将搜索关键字添加到系统的数据库中，代码如下所示。

文件名: `BrowserCollectorAct.java`

```
Browser.addSearchUrl(mResolver, "汪永松");
```

3. 调用浏览器搜索

通过意向对象（`Intent`）的网页搜索动作（`ACTION_WEB_SEARCH`）可调用系统的浏览器工具搜索指定的关键字，如代码 8-28 所示。

代码 8-28 调用浏览器搜索

文件名: `BrowserCollectorAct.java`

```

1  private void addSearching() {
2      //(3)调用搜索
3      Intent intent = new Intent(Intent.ACTION_SEARCH);
4      intent.putExtra(SearchManager.QUERY, "汪永松");
5      this.startActivity(intent);
6      //(2)调用网页自动搜索
7      //Intent intent = new Intent(Intent.ACTION_WEB_SEARCH);
8      //intent.putExtra(SearchManager.QUERY, "汪永松");
9      //this.startActivity(intent);
10     //(1)添加搜索关键字
11     //Browser.addSearchUrl(mResolver, "汪永松");
12 }

```

在代码 8-28 中，也可以通过意向对象的搜索动作（`ACTION_SEARCH`）来调用“搜索范围选择界面”来区分范围搜索指定的关键字。

8.5.3 下载管理

在网页视图应用小节中已经介绍过网页链接资源的下载过程（使用线程通过 `URL` 连接的方式获取资源流）。在 `Android 3` 平台中，为了简化下载管理（不仅是下载动作，还包括下载信息的管理），提供了下载管理器（`DownloadManager`）用于管理下载。图 8-15 所示为使用下载管理器进行下载管理的实机界面。



图 8-15 使用下载管理器进行下载管理的实机界面

1. 应用程序主 Activity 框架

代码 8-29 是下载器示例程序的主 Activity 组件的框架定义，包括选项页控件（界面主组件）、网页视图和客户端（集成网页内容）、下载列表视图（管理待下载信息）、下载信息列表（管理已下载信息）和下载管理器（用于下载和获取下载信息）。

代码 8-29 下载器示例程序的主 Activity 组件的框架定义

文件名: DownloaderAct.java

```

1  public class DownloaderAct extends TabActivity implements OnItemClickListener {
2      //网页视图及客户端
3      private WebView mWebView = null;
4      //下载列表视图
5      private ListView mDownloadView = null;
6      private ArrayList<String> mPaths = new ArrayList<String>();
7      //下载信息列表视图
8      private ListView mDownloadedView = null;
9      private SimpleCursorAdapter mAdapter = null;
10     //下载管理器
11     private DownloadManager mMgr = null;
12     private long mPrevDownloadId = -1;
13     //下载监视器
14     private FooDownloadMonitor mMonitor = null;
15
16     @Override
17     public void onCreate(Bundle savedInstanceState) {
18         super.onCreate(savedInstanceState);
19         setContentView(R.layout.main);
20         //布局定义组件（视频视图、列表视图和网页视图）
21         mDownloadView = (ListView)findViewById(R.id.download_list);
22         mWebView = (WebView)findViewById(R.id.web);
23         mDownloadedView = (ListView)findViewById(R.id.downloaded_list);
24         //初始化 TabHost
25         getTabHost().addTab(getTabHost().newTabSpec("t1")
26                             .setIndicator("推荐资源").setContent(R.id.web));
27         getTabHost().addTab(getTabHost().newTabSpec("t2")

```




Android 平台开发之旅 第2版

```

28             .setIndicator("下载管理").setContent(R.id.download_list));
29     getTabHost().addTab(getTabHost().newTabSpec("t3")
30             .setIndicator("本地下载").setContent(R.id.downloaded_list));
31     //初始化下载
32     initDownload();
33     initDownloadedList();
34     initWebView();
35 }
36 .....
37 };

```

在代码 8-29 中，使用选项页控件作为主界面组件，该组件包含 3 个选项页。

- 1) 推荐资源（第 25 行），用于显示网页，本页的主要组件是网页视图。
- 2) 下载管理（第 27 行），用于管理下载任务列表，本页的主要组件是列表视图。
- 3) 本地下载（第 29 行），用于管理已下载信息列表，本页的主要组件是列表视图。

2. 主界面布局定义

代码 8-30 是下载器示例程序的主界面布局定义，其使用选项页控件作为根组件，其中包含 3 个选项页。

代码 8-30 下载器示例程序的主界面布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <TabHost xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@android:id/tabhost"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <LinearLayout android:orientation="vertical"
6          android:layout_width="fill_parent" android:layout_height="fill_parent">
7          <TabWidget android:id="@android:id/tabs"
8              android:layout_width="fill_parent" android:layout_height="wrap_content" />
9          <FrameLayout android:id="@android:id/tabcontent"
10              android:layout_width="fill_parent" android:layout_height="fill_parent">
11              <WebView android:id="@+id/web"
12                  android:layout_width="fill_parent" android:layout_height="fill_parent" />
13              <ListView android:id="@+id/download_list"
14                  android:layout_width="fill_parent" android:layout_height="fill_parent" />
15              <ListView android:id="@+id/downloaded_list"
16                  android:layout_width="fill_parent" android:layout_height="fill_parent" />
17          </FrameLayout>
18      </LinearLayout>
19  </TabHost>

```

3. 应用程序配置信息接口

代码 8-31 是应用程序中有关配置信息接口的定义，包括主页 URL、下载文件夹和下载记录的列信息。

代码 8-31 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String HOME_URL = "http://mp3.sogou.com/";
3      public static final String SUB_DIR = "Download/";
4      //下载信息列
5      public static final String[] COLs = {
6          DownloadManager.COLUMN_URI,
7          DownloadManager.COLUMN_TOTAL_SIZE_BYTES,
8          DownloadManager.COLUMN_LAST_MODIFIED_TIMESTAMP };
9      public static final int COL_TSP = 2;
10     public static final int COL_SIZE = 1;
11     public static final int COL_URI = 0;
12 };

```

4. 初始化过程

在代码 8-29 中, 初始化内容包括下载管理、下载信息列表和网页视图。

(1) 初始化下载

代码 8-32 是初始化下载的主要代码, 其中主要是获取下载管理器示例并为下载列表视图设置适配器。其中, 下载列表视图列表项的内容是下载资源的 URL。

代码 8-32 初始化下载

文件名: DownloaderAct.java

```

1  private void initDownload() {
2      //获取下载管理器实例
3      final String sName = Service.DOWNLOAD_SERVICE;
4      this.mMgr = (DownloadManager) this.getSystemService(sName);
5      //初始化下载列表
6      ListAdapter adapter =
7          new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, mPaths);
8      mDownloadView.setAdapter(adapter);
9      mDownloadView.setOnItemClickListener(this);
10     //初始化下载监视器
11     initDownloadMonitor();
12 }

```

(2) 初始化下载信息列表视图

代码 8-33 是初始化下载信息列表视图的主要代码, 其中主要是为列表设置页眉视图和适配器。其中, 下载信息列表视图列表项的内容是下载记录, 包括 URL、资源大小和时间戳。

代码 8-33 初始化下载信息列表视图

文件名: DownloaderAct.java

```

1  private void initDownloadedList() { //初始化下载信息列表
2      mDownloadedView.addHeaderView(
3          this.getLayoutInflater().inflate(R.layout.header, null, false);

```




```
4      mDownloadedView.setHeaderDividersEnabled(true);
5      //创建适配器
6      mAdapter = new SimpleCursorAdapter(this, R.layout.row, null, IConfig.COLs,
7          new int[] { R.id.text1, R.id.text2, R.id.text3 }, 0);
8      mDownloadedView.setAdapter(mAdapter);
9  }
```

在代码 8-33 中，下载信息列表视图使用的是简单游标适配器，每个列表项对应一条记录游标。

(3) 初始化网页视图

代码 8-34 是初始化网页视图的主要代码，其中主要是设置网页视图客户端，此外还为网页视图设置了上下文菜单，用于下载网页链接。

代码 8-34 初始化网页视图

文件名: DownloaderAct.java

```
1  private void initWebView() { //初始化网页视图
2      mWebView.getSettings().setJavaScriptEnabled(true);
3      mWebView.setWebViewClient(new WebViewClient() {
4          @Override
5          public boolean shouldOverrideUrlLoading(WebView view, String url) {
6              view.loadUrl(url);
7              return(true);
8          }
9      });
10     mWebView.loadUrl(IConfig.HOME_URL);
11     //设置上下文菜单
12     this.registerForContextMenu(mWebView);
13 }
```

5. 下载控制

下载控制包括添加下载任务、启动下载和获取下载状态。

(1) 添加下载任务

下载任务的添加实质上是将目标资源的 URL 添加到下载任务列表视图中。目标资源的 URL 通过网页视图的单击测试结果来捕获；URL 的添加使用上下文菜单项进行触发，如图 8-16 所示。

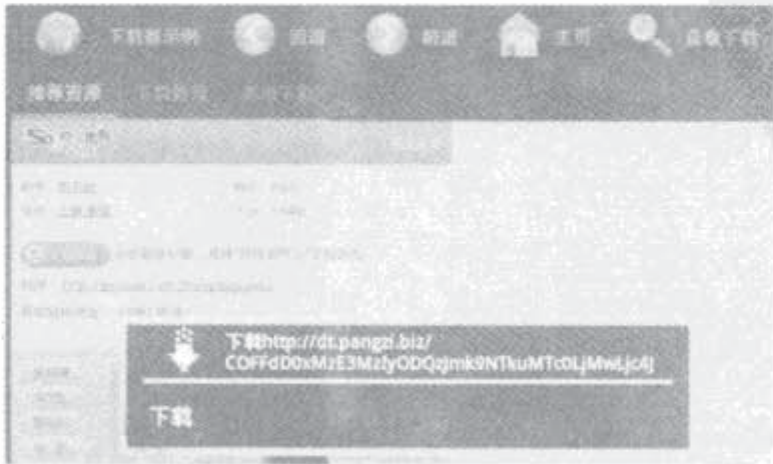


图 8-16 添加下载任务的实机界面

代码 8-35 是在网页视图中通过上下文菜单添加下载任务的主要代码。

代码 8-35 通过上下文菜单添加下载任务

文件名: DownloaderAct.java

```
1  @Override
2  public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo) {
3      super.onCreateContextMenu(menu, v, menuInfo);
4      this.getMenuInflater().inflate(R.menu.ctx_menu, menu);
5      //设置菜单抬头按钮
6      menu.setHeaderIcon(R.drawable.download);
7      //获取单击测试结果
8      HitTestResult htr = this.mWebView.getHitTestResult();
9      String url = htr.getExtra();
10     if(url != null) { menu.setHeaderTitle("下载" + url); }
11 }
12 @Override
13 public boolean onContextItemSelected(MenuItem item) {
14     switch(item.getItemId() ) {
15         case R.id.mi_download: { HitTestResult htr = this.mWebView.getHitTestResult();
16             String url = htr.getExtra();
17             //添加到下载任务列表
18             if(url != null) { addDownloadList(url); }
19             else { return(false); }
20         }
21     }
22     return super.onContextItemSelected(item);
23 }
24
25 @SuppressWarnings("unchecked")
26 protected void addDownloadList(String url) { //添加 URL 到下载任务列表
27     this.mPaths.add(url);
28     ((ArrayAdapter<String>)mDownloadView.getAdapter()).notifyDataSetChanged();
29     getTabHost().setCurrentTab(1);
30 }
```

在代码 8-35 中, 当将 URL 添加到下载列表视图所绑定的数据集后 (第 27 行), 还需要通过适配器发出数据集改变的通知 (第 28 行)。

(2) 启动下载

当下载任务添加到下载列表后, 用户通过单击列表中的项来启动对资源的下载, 如图 8-17 所示。

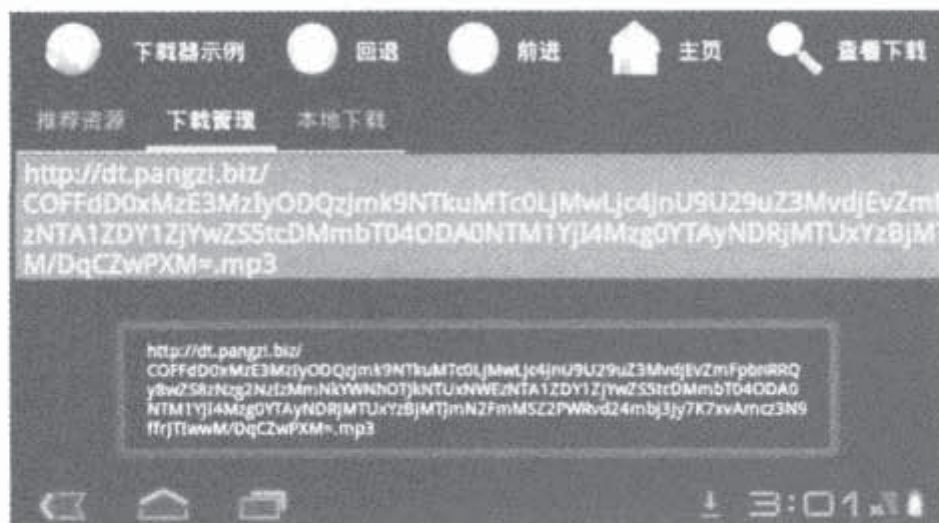


图 8-17 启动下载的实机界面

代码 8-36 是启动下载的主要代码。

代码 8-36 启动下载

文件名: DownloaderAct.java

```

1  @Override
2  public void onItemClick(AdapterView<?> parent, View v, int pos, long id) {
3      Toast.makeText(this, this.mPaths.get(pos), Toast.LENGTH_LONG).show();
4      doStart(this.mPaths.get(pos));
5  }
6
7  private void doStart(String url) { //启动下载
8      DownloadManager.Request request = new DownloadManager.Request(Uri.parse(url));
9      //构造目标文件路径
10     File file = new File(url);
11     final String fname = file.getName();
12     final String path = Environment.getExternalStorageDirectory().toURI().toString();
13     request.setDestinationUri(Uri.parse(path+IConfig.SUB_DIR+fname));
14     //设置下载界面的显示方式
15     request.setVisibleInDownloadsUi(true);
16     request.setNotificationVisibility(DownloadManager.Request.VISIBILITY_VISIBLE);
17     //取消前一次启动的下载
18     if(mPrevDownloadId!=-1) { this.mMgr.remove(mPrevDownloadId); }
19     //将请求排入队列
20     mPrevDownloadId = this.mMgr.enqueue(request);
21 }
22
23 @Override
24 protected void onDestroy() { super.onDestroy();
25     //取消前一次启动的下载
26     if(mPrevDownloadId!=-1) { this.mMgr.remove(mPrevDownloadId); }
27 }

```

在代码 8-36 中, 当用户单击下载列表中的条目时, 会获取该项对应的资源 URL, 从而

启动下载。对于下载过程，首先需要构建下载请求（DownloadManager.Request，第 8 行），再使用下载管理器将该请求添加到任务队列中（第 20 行），当轮到该请求时即启动下载。

下载请求用于明确下载的输入和输出以及界面显示方式，如是否以通知的形式显示下载状态界面。

注意：建议明确下载请求的目标路径（第 10~13 行）。在默认情况下，下载的内容将被保存到共享下载缓冲区，可能会受到系统缓冲区政策的影响（如对缓冲区大小的限制等），从而导致下载过程存在很多不可预测的问题。

（3）获取下载状态

当某一下载完成时系统将发出广播。用户程序可以通过广播接收器来获取该消息，从而获取所完成的下载信息。代码 8-37 是获取下载状态的主要代码。

代码 8-37 获取下载状态

文件名：DownloaderAct.java

```
1  private void initDownloadMonitor() { //初始化下载监视器
2      mMonitor = new FooDownloadMonitor();
3      final String action = DownloadManager.ACTION_DOWNLOAD_COMPLETE;
4      IntentFilter filter = new IntentFilter(action);
5      //注册监视器
6      this.registerReceiver(mMonitor, filter);
7  }
8
9  @Override
10 protected void onDestroy() { super.onDestroy();
11     //取消前一次启动的下载
12     if(mPrevDownloadId!=-1) { this.mMgr.remove(mPrevDownloadId); }
13     //取消下载监视器的注册
14     this.unregisterReceiver(mMonitor);
15 }
16
17 class FooDownloadMonitor extends BroadcastReceiver {
18     @Override
19     public void onReceive(Context context, Intent intent) {
20         Bundle data = intent.getExtras();
21         //获取完成的下载 ID
22         final long did = data.getLong(DownloadManager.EXTRA_DOWNLOAD_ID);
23         Uri uri = mMgr.getUriForDownloadedFile(did);
24         final String tip = "【"+uri.toString()+"】\n 下载完毕！ ";
25         Toast.makeText(DownloaderAct.this, tip, Toast.LENGTH_LONG).show();
26     }
27 };
```

在代码 8-37 中，下载监视器实际上是一个广播接收器（第 17 行），其用于接收系统所发送的下载完成的消息（第 3 行）。从意向的扩展空间中可获取所完成的下载 ID 信息（第



22 行), 然后使用下载管理器的 `getUriForDownloadedFile` 方法可获取指定下载 ID 的 URI (第 23 行)。

6. 下载信息管理

获取下载信息有两种方式: 调用系统的下载信息查看界面和使用下载管理器的查询接口 (`DownloadManager.Query`) 获取下载记录。

(1) 调用系统的下载信息查看界面

代码 8-38 是调用系统的下载信息查看界面的主要代码。

代码 8-38 调用系统的下载信息查看界面

文件名: `DownloaderAct.java`

```
1 //调用系统查看下载信息的 Activity 组件
2 Intent intent = new Intent(DownloadManager.ACTION_VIEW_DOWNLOADS);
3 this.startActivity(intent);
```

(2) 使用下载管理器的查询接口获取下载记录

代码 8-39 是使用下载管理器的查询接口获取下载记录的主要代码。

代码 8-39 获取下载记录

文件名: `DownloaderAct.java`

```
1 private void doQuery() {
2     //通过下载管理器的查询接口获取记录游标
3     Cursor c = this.mMgr.query(new DownloadManager.Query());
4     ..... //记录游标处理
5 }
```

在代码 8-39 中, 使用下载管理器的 `query` 方法可获取下载信息的记录游标 (第 3 行), 该方法的唯一参数是下载管理器查询接口, 通过该接口可以查询指定的下载 ID 和状态的记录。

7. 下载使用许可

为了从互联网获取下载资源并保存到本地存储器, 程序需要拥有访问互联网和写本地 SD 卡存储器的使用许可, 即必须在程序清单中声明这些使用许可, 其声明代码如下所示。

文件名: `AndroidManifest.xml`

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

第9章 无线通信

本章详细介绍 Android 平台支持的无线通信方式：短消息通信、蓝牙通信和近距离通信。希望借助实际的开发案例，帮助读者开启无线通信的奇妙世界。

9.1 无线通信概述

从电台广播到卫星电视，从蓝牙通信到 Wi-Fi 无线上网，这些不断丰富人们生活的通信方式所使用的核心技术就是无线通信。而无线通信技术也在不断地演进，越来越注重结合工业和医学方面的标准，数据的安全性也成为日益关注的内容。

Android 平台提供多种方式的无线通信，主要有短消息通信、蓝牙通信和近距离通信。

9.2 短消息通信

对于移动平台而言，与外界通信的最基本方式就是无线通信，而短消息通信是手机最常见的方式。随着多媒体技术的不断进步，短消息的内容从最简单的文本，发展到图像、音频甚至视频。图 9-1 是短消息通信示意图。

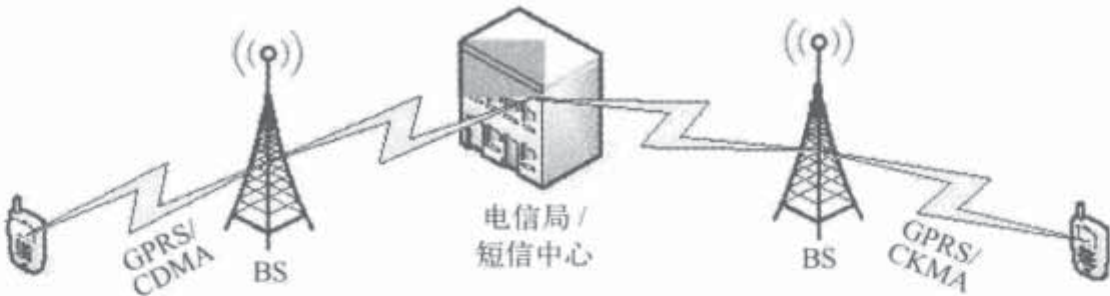


图 9-1 短消息通信示意图

9.2.1 Android 平台对短消息的支持

Android 平台提供了电话系统包（android.telephony）用于短消息的应用。表 9-1 是对该包中与短消息有关类/接口的说明。

表 9-1 电话系统包中短消息有关类/接口的说明

类/接口	说 明
SmsManager	短消息管理器
SmsMessage	代表一条短消息

9.2.2 发送短消息

图 9-2 所示的是发/收短消息的示例界面（短消息的发送方和接收方处于不同的设备



中), 其中左边为短消息发送程序界面, 右边是接收方设备的桌面。当接收方收到短消息后, 系统会以通知的方式提示用户 (在桌面上方的状态条中持续显示该通知)。

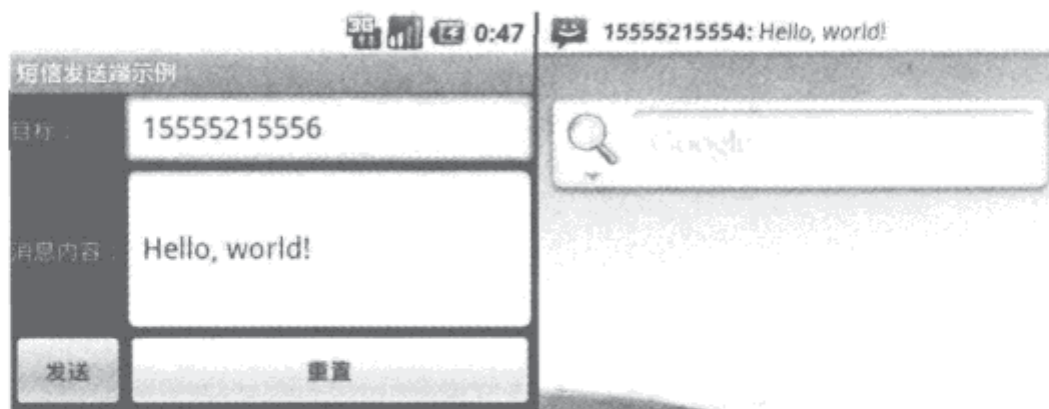


图 9-2 发/收短消息界面

1. 应用程序主 Activity 框架

代码 9-1 是短消息发送程序 Activity 组件的框架定义。

代码 9-1 发送短消息程序 Activity 组件的框架定义

文件名: SmsSenderAct.java

```

1  public class SmsSenderAct extends Activity implements OnClickListener {
2      //短消息管理器
3      private SmsManager mSmsMgr = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //获取按钮控件并设置单击侦听器
10         Button btnReset = (Button)findViewById(R.id.btn_reset);
11         Button btnSend = (Button)findViewById(R.id.btn_send);
12         btnReset.setOnClickListener(this);
13         btnSend.setOnClickListener(this);
14         //获取默认短消息管理器
15         this.mSmsMgr = SmsManager.getDefault();
16     }
17     @Override
18     public void onClick(View v) {
19         switch(v.getId()) { case R.id.btn_reset: { doReset(); break; }
20                             case R.id.btn_send: { doSend(); break; }
21         }
22     }
23     .....
24 };

```

在代码 9-1 中, 首先获取默认短消息管理器 (第 14 行), 继而通过按钮组件启动短消息发送 (第 20 行)。

2. 发送短消息

代码 9-2 是代码 9-1 中发送短消息的主要代码。

代码 9-2 发送短消息

文件名: SmsSenderAct.java

```
1 private void doSend() { //发送短消息
2     String dest = ((EditText)findViewById(R.id.txt_dest)).getText().toString();
3     String msg = ((EditText)findViewById(R.id.text)).getText().toString();
4     //获取未决意向对象
5     PendingIntent pi = PendingIntent.getBroadcast(
6         this, 0, new Intent(), PendingIntent.FLAG_ONE_SHOT);
7     //发送短消息
8     this.mSmsMgr.sendTextMessage(dest, null, msg, pi, null);
9 }
```

在代码 9-2 中，使用短消息管理器的 `sendTextMessage` 方法发送消息（第 8 行）。因为发送短消息是一个预期动作，并不保证会马上执行（在发送过程中可能会被取消），所以需要 通过未决意向对象来“预约”短消息的发送（第 5 行）。

3. 使用许可

为了能够发送短消息，应用程序需要具备发送短消息的使用许可，即必须在程序清单文件中声明该许可，其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

9.2.3 接收短消息

图 9-3 所示的是接收短消息工具的示例界面。



图 9-3 短消息接收程序界面

1. 应用程序主 Activity 框架

代码 9-3 是短消息接收程序 Activity 组件的框架定义。

代码 9-3 接收短消息程序 Activity 组件的框架定义

文件名: SmsReceiverAct.java

```
1 public class SmsReceiverAct extends Activity {
2     //短消息接收器
```




Android 平台开发之旅 第2版

```

3      private SmsReceiver mReceiver = null;
4      //主线程消息队列处理器
5      private Handler mHandler = null;
6
7      @Override
8      public void onCreate(Bundle savedInstanceState) {
9          super.onCreate(savedInstanceState);
10         setContentView(R.layout.main_view);
11         //初始化主线程消息队列处理器
12         this.mHandler = new Handler() {
13             @Override
14             public void handleMessage(Message msg) {
15                 Bundle data = msg.getData();
16                 if(data==null) { return; }
17                 final String dest = data.getString(IConfig.KEY1);
18                 final String text = data.getString(IConfig.KEY2);
19                 ((EditText)findViewById(R.id.text)).append(dest+": "+text+"\n");
20             }
21         };
22         //初始化短消息接收器
23         this.mReceiver = new SmsReceiver(this.mHandler);
24         //注册接收
25         IntentFilter filter1 = new IntentFilter(SmsReceiver.class.getName() );
26         IntentFilter filter2 = new IntentFilter(IConfig.ACTION);
27         this.registerReceiver(this.mReceiver, filter1);
28         this.registerReceiver(this.mReceiver, filter2);
29     }
30     @Override
31     protected void onDestroy() { //注销接收
32         this.unregisterReceiver(this.mReceiver);
33         super.onDestroy();
34     }
35 };

```

在代码 9-3 中，使用一个短消息接收器（第 23 行）来接收短消息，通过过滤器指定接收短消息的接收器组件（第 25 行和第 26 行）。当 Activity 组件注册接收器后，该接收器就会接收符合过滤器的短消息（第 27 行和第 28 行）；而当注销该接收器后，接收器就不会再接其短消息（第 32 行）。

代码中定义了一个主线程消息队列处理器（第 12 行），用于接收短消息接收器所转发的短消息，所以在接收器构造方法中将该处理器传入（第 23 行）。

2. 短消息接收器

代码 9-4 是代码 9-3 中所参考的短消息接收器的定义。

代码 9-4 短消息接收器的定义

文件名: SmsReceiver.java

```

1  public class SmsReceiver extends BroadcastReceiver {

```



```

2      //主线程消息队列
3      private Handler mHandler = null;
4
5      public SmsReceiver(Handler handler) { this.mHandler = handler; }
6
7      @Override
8      public void onReceive(Context context, Intent intent) {
9          if(!intent.getAction().equalsIgnoreCase(IConfig.ACTION)) { return; }
10         //获取意向对象扩展控件内容
11         Bundle bundle = intent.getExtras();
12         if (bundle == null) { return; }
13         //获取消息对象数组
14         Object[] pdus = (Object[]) bundle.get(IConfig.EXTRAS);
15         SmsMessage[] msgs = new SmsMessage[pdus.length];
16         for (int i = 0; i < pdus.length; i++) { //遍历消息群
17             msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
18             //获取发送源及消息内容
19             String from = msgs[i].getDisplayOriginatingAddress();
20             String msg = msgs[i].getDisplayMessageBody();
21             //传递消息给主线程
22             FooSysUtil.getInstance().postMsgs(mHandler,
23                 new String[] { IConfig.KEY1, IConfig.KEY2},
24                 new String[] { from, msg });
25         }
26     }
27 };

```

在代码 9-4 中，通过重载广播接收器的 `onReceive` 方法来处理系统的短消息接收事件（第 8 行）。在该方法中，首先通过数据内容生成短消息实例（第 17 行），继而对短消息中的信息（发送源地址和消息文本等）进行解析（第 19 行和第 20 行），最后将短消息内容转发给主线程（第 22 行）。

提示：代码 9-4 中短消息数据包（第 14 行）所包含的数据条目名为 `pdus`（即常量 `EXTRAS_NAME`），该名称源于协议数据单元（Protocol Data Unit, PDU），是指在网络的对等实体中用于传递的数据单元。

由于该接收器的构造方法并非默认（默认为无参数构造方法），所以该接收器只能通过 Java 代码构造，而不能在程序清单文件中进行声明。

3. 使用许可

为了能够正常接收短消息，需要在应用程序清单中添加接收短消息的使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
```




9.3 蓝牙通信

蓝牙技术基于无线技术，使用了号称符合工业的、科学的、医学的（Industrial，Scientific，Medical，ISM）频率的波段（2.45GHz），在无线设备的电气特性支持下，通过特定的通信协议栈进行通信。

如果说无线消息使用的是 GSM/CDMA 等广域无线网络，那么蓝牙技术使用的是手机与手机之间的局域无线网络，其私有化和个性化特征表现得尤为突出，这可能就是蓝牙设备风行十多年的原因所在。图 9-4 是移动设备间进行蓝牙通信的示意图。



图 9-4 蓝牙通信示意图

9.3.1 Android 平台对蓝牙的支持

Android 平台提供蓝牙包（android.bluetooth）用于蓝牙应用。表 9-2 是该包中重要类/接口的说明。

表 9-2 蓝牙包中重要类/接口的说明

类/接口	说 明
BluetoothAdapter	代表本地蓝牙设备的适配器
BluetoothDevice	表示一个远程蓝牙设备
BluetoothServerSocket	用于侦听的蓝牙服务套接字
BluetoothSocket	已连接或正连接的蓝牙套接字
BluetoothClass	用于描述一个蓝牙设备的特性和性能

9.3.2 蓝牙通信模式

对于手机设备而言，都是无线网络中的对等实体，不存在主从关系。但就某一次蓝牙通信过程而言，主动侦听连接的主机方被称为服务端，发起请求的主机方被称为客户端，无论客户端还是服务端，要想与其他主机进行蓝牙通信，那么通信双方都必须开启蓝牙模块。

1. 服务端

以下过程是蓝牙通信服务端的使用模式。

- 1) 获取本地蓝牙适配器（BluetoothAdapter）。
- 2) 开启蓝牙功能并允许本地设备被检测。
- 3) 使用本地适配器侦听服务记录，获取服务套接字（BluetoothServerSocket）。
- 4) 等待客户端的连接；当有客户端连接时，获取与客户端的套接字（BluetoothSocket）。
- 5) 通过套接字的输入/输出流与客户端进行通信。

6) 通信完毕, 关闭与客户端的套接字, 继续等待后续客户端的连接。

2. 客户端

以下过程是蓝牙通信客户端的使用模式。

- 1) 获取本地蓝牙适配器。
- 2) 开启蓝牙功能。
- 3) 检测远程设备并获取远程蓝牙设备 (BluetoothDevice)。
- 4) 通过远程蓝牙设备接口建立与服务端的套接字。
- 5) 通过套接字的输入/输出流与服务端进行通信。
- 6) 通信完毕, 关闭与服务端的套接字。

9.3.3 蓝牙通信

1. 服务端

图 9-5 是蓝牙服务端程序的实机界面。

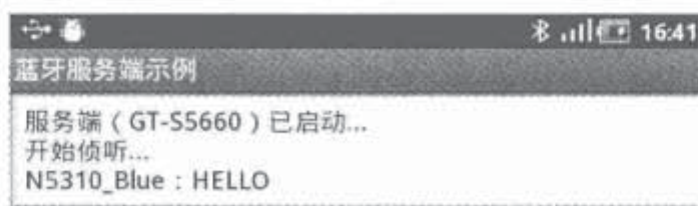


图 9-5 蓝牙服务端程序的实机界面

(1) 应用程序主 Activity 框架

代码 9-5 是蓝牙服务端程序 Activity 组件的框架定义。

代码 9-5 蓝牙服务端程序 Activity 组件的框架定义

文件名: BtServerDemoAct.java

```

1  public class BtServerAct extends Activity {
2      //蓝牙适配器
3      private BluetoothAdapter mAdapter = null;
4      //主线程消息队列处理器
5      private Handler mHandler = null;
6
7      @Override
8      public void onCreate(Bundle savedInstanceState) {
9          super.onCreate(savedInstanceState);
10         setContentView(R.layout.main);
11         //获取蓝牙适配器实例
12         mAdapter = BluetoothAdapter.getDefaultAdapter();
13         if(mAdapter == null) {
14             Toast.makeText(this, "获取蓝牙适配器失败!", Toast.LENGTH_LONG).show();
15             this.finish();
16             return;
17         }

```



Android 平台开发之旅 第2版

```

18         //启动蓝牙
19         enableBt();
20         //允许客户端检测
21         enableDiscover();
22         //初始化主线程消息队列处理器
23         mHandler = new Handler() {
24             @Override
25             public void handleMessage(Message msg) {Bundle data = msg.getData();
26                 printText(data.getString(IConfig.EXTRA_MSG));
27             }
28         };
29     }
30
31     private void enableBt() { //启动蓝牙
32         if(!mAdapter.isEnabled() ) {
33             Intent i = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
34             startActivity(i);
35         }
36     }
37
38     private void enableDiscover() { //允许被检测
39         Intent i = new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
40         i.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
41         startActivity(i);
42     }
43
44     @Override
45     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
46         if(requestCode == IConfig.REQ_ENABLE_DISCOVER) {
47             if(resultCode != Activity.RESULT_CANCELED) { startServer(); } //启动服务
48         }
49     }
50
51     private void startServer() { //启动服务线程
52         BtServerThread t = new BtServerThread(mAdapter, mHandler);
53         t.start();
54         printText("服务端 (" + this.mAdapter.getName() + ") 已启动...");
55     }
56 };

```

在代码 9-5 的 Activity 组件初始化环节中，使用蓝牙适配器的 `getDefaultAdapter` 方法来获取本地蓝牙适配器（第 12 行）。如果本地蓝牙适配器获取成功，则使用该适配器来创建服务线程（第 52 行）。

此外，为了保证蓝牙通信以及服务端设备能够被客户端所检测，代码中使用系统意向动作启动蓝牙（第 19 行）和允许设备被检测（第 21 行），如图 9-6 所示。

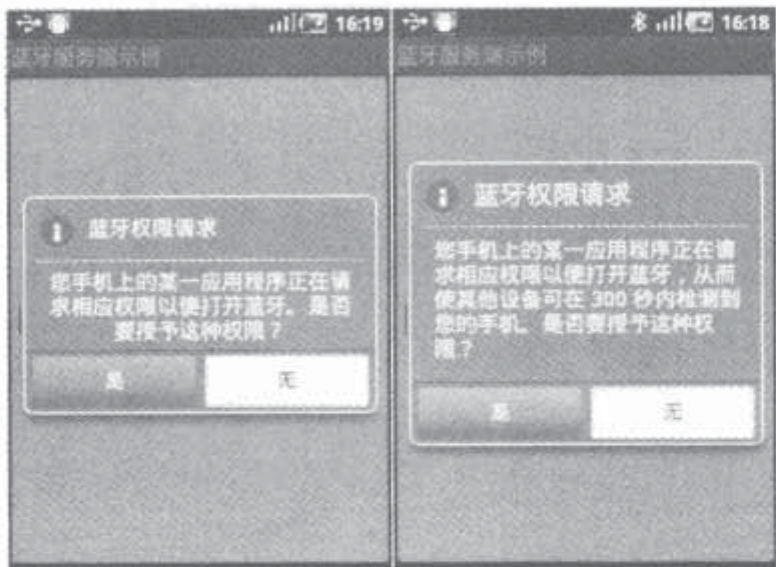


图 9-6 启动蓝牙和允许设备被检测的界面

(2) 蓝牙服务线程

代码 9-6 是代码 9-5 中提到的蓝牙服务线程的定义。

代码 9-6 蓝牙服务线程的定义

文件名: BtServerThread.java

```
1 public class BtServerThread extends Thread {
2     //主线程消息队列处理器
3     private Handler mHandler = null;
4     //蓝牙服务套接字
5     private BluetoothServerSocket mSSocket = null;
6
7     public BtServerThread(BluetoothAdapter adapter, Handler handler) {
8         this.mHandler = handler;
9         //获取蓝牙服务套接字
10        BluetoothServerSocket tmp = null;
11        try { tmp = adapter.listenUsingRfcommWithServiceRecord(IConfig.BT_SDP,
12                                                                IConfig.BT_UUID);
13            //发送消息给主线程
14            FooSysUtil.getInstance().postMsg(mHandler,IConfig.EXTRA_MSG,"开始侦听...");
15        } catch (IOException e) { e.printStackTrace(); }
16        this.mSSocket = tmp;
17    }
18
19    @Override
20    public void run() {
21        if(this.mSSocket == null) { return; }
22        try { while(true) {
23            //等待客户端的连接
24            BluetoothSocket csocket = this.mSSocket.accept();
25            //与客户端进行通信
26            talk(csocket);
27            //通信完毕，关闭连接
```



Android 平台开发之旅 第2版

```

28         csocket.close();
29     }
30     } catch (IOException e) { e.printStackTrace(); }
31     finally { if(mSSocket != null) {
32         try { mSSocket.close();
33         } catch (IOException e) { e.printStackTrace(); }
34     }
35     }
36 }
37
38 //与客户端进行通信
39 private void talk(BluetoothSocket csocket) throws IOException {
40     //获取套接字 I/O 流
41     InputStream is = csocket.getInputStream();
42     OutputStream os = csocket.getOutputStream();
43     //建立读取通道
44     BufferedReader br = new BufferedReader(new InputStreamReader(is));
45     StringBuffer sb = new StringBuffer();
46     String line = null;
47     //接收客户端输入
48     while( (line=br.readLine()) != null) { sb.append(line); }
49     //发送消息给主线程
50     final String remoteDevName = csocket.getRemoteDevice().getName();
51     final String reply = remoteDevName+": "+sb.toString();
52     FooSysUtil.getInstance().postMsg(mHandler, IConfig.EXTRA_MSG, reply);
53     //回复客户端
54     PrintWriter pw = new PrintWriter(os);
55     pw.println(sb.toString());
56     pw.flush();
57     //关闭 I/O 流
58     pw.close();
59     br.close();
60 }
61 };

```

在代码 9-6 中，使用蓝牙适配器来侦听指定的通用唯一识别码（UUID）和服务检测协议名的蓝牙服务，并获取服务套接字（第 11 行）。通过服务套接字等待客户端的连接，并与客户端建立套接字连接（第 24 行），服务端使用该套接字来与客户端进行通信（talk 方法，第 39 行）。

提示：无线频率通信（Radio frequency communication, RFCOMM）是一种类似于串口的简单传输协议，其目的是在不同设备的应用之间建立一条完整的通信路径并在彼此之间保持通信段（Communication Segment），如图 9-7 所示。



图 9-7 RFCOMM 通信示意图

(3) 使用许可

开启蓝牙功能并进行通信和允许本地设备被检测，需要拥有蓝牙和蓝牙管理的使用许可，即必须在程序清单文件中声明这些使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

2. 蓝牙通信配置信息接口

代码 9-7 是蓝牙通信示例程序中有关配置信息接口的定义，包括服务检测协议名、蓝牙服务 UUID 和消息数据项。

代码 9-7 应用程序配置信息接口的定义

文件名：IConfig.java

```
1 public interface IConfig {
2     //服务检测协议名
3     public static final String BT_SDP = "BtDemo";
4     //UUID(8-4-4-4-12)
5     public static final UUID BT_UUID =
6         UUID.fromString("12345678-90AB-CDEF-1234-567890ABCDEF");
7     //扩展数据项名
8     public static final String EXTRA_MSG = "msg";
9     //请求代码
10    public static final int REQ_ENABLE_DISCOVER = 2012;
11 };
```

在代码 9-7 中，蓝牙 UUID 采用的是“8-4-4-4-12”的形式，而不是“8-4-4-16”，读者可以借助 UUID 生成网页（<http://www.uuidgenerator.com>）来生成所需的 UUID。

3. 客户端

图 9-8 是蓝牙客户端程序的实机界面。客户端会先检测远程蓝牙设备，生成设备列表，然后由用户选择蓝牙设备并与之通信。



图 9-8 蓝牙客户端程序的实机界面



(1) 应用程序主 Activity 框架

代码 9-8 是蓝牙客户端程序 Activity 组件的框架定义。

代码 9-8 蓝牙客户端程序 Activity 组件的框架定义

文件名: BtClientAct.java

```

1  public class BtClientAct extends Activity implements OnItemClickListener {
2      //蓝牙适配器
3      private BluetoothAdapter mAdapter = null;
4      //检测蓝牙设备的广播接收器
5      private BroadcastReceiver mReceiver = null;
6      //设备列表视图
7      private ListView mDevListView = null;
8      private ArrayList<String> mNames = new ArrayList<String>();
9      private ArrayList<BluetoothDevice> mDevices = new ArrayList<BluetoothDevice>();
10
11     @Override
12     public void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.main);
15         //设备列表
16         mDevListView = ((ListView)findViewById(R.id.dev_list));
17         ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
18             android.R.layout.simple_list_item_1, mNames);
19         mDevListView.setAdapter(adapter);
20         mDevListView.setOnItemClickListener(this);
21         //获取蓝牙适配器
22         mAdapter = BluetoothAdapter.getDefaultAdapter();
23         if(mAdapter == null) {
24             Toast.makeText(this,"获取蓝牙适配器失败!",Toast.LENGTH_LONG).show();
25             this.finish();
26             return;
27         }
28         //启动蓝牙
29         enableBt();
30     }
31
32     private void enableBt() { //启动蓝牙
33         if(!mAdapter.isEnabled()) {
34             Intent i = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
35             startActivityForResult(i, IConfig.REQ_ENABLE_BT);
36         } else { initClient(); doFind(); }
37     }
38
39     @Override
40     protected void onActivityResult(int requestCode, int resultCode, Intent data) {

```



```

41         if(requestCode == IConfig.REQ_ENABLE_BT) {
42             if(resultCode == Activity.RESULT_OK) { initClient(); doFind(); }
43         }
44     }
45     .....
46 };

```

在代码 9-8 中，首先获取本地蓝牙设备适配器（第 22 行），然后启动蓝牙。若蓝牙已就绪或启动后就绪，则开始初始化客户端并开始检测远程蓝牙设备（第 36 行或第 42 行）。

（2）初始化客户端

代码 9-9 是代码 9-8 中初始化蓝牙客户端的主要代码，其中主要是定义接收器并注册，该接收器用于接收所检测到的远程蓝牙设备信息。

代码 9-9 初始化蓝牙客户端

文件名: BtClientAct.java

```

1  private void initClient() { //初始化客户端
2      mReceiver = new BroadcastReceiver() {
3          @Override
4          public void onReceive(Context context, Intent i) {
5              String action = i.getAction();
6              if(!BluetoothDevice.ACTION_FOUND.equals(action)) { return; }
7              //获取意向所包含的蓝牙设备实例
8              final String extra = BluetoothDevice.EXTRA_DEVICE;
9              BluetoothDevice dev = i.getParcelableExtra(extra);
10             String id = dev.getAddress()+"|"+dev.getName();
11             if(mNames.indexOf(id)==-1) { //重复判断，添加远程设备到列表
12                 mNames.add(id); mDevices.add(dev);
13             }
14         }
15     };
16     //注册接收检测到蓝牙设备消息的广播接收器
17     IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
18     this.registerReceiver(mReceiver, filter);
19 }
20 @Override
21 protected void onDestroy() { super.onDestroy();
22     //注销接收器
23     this.unregisterReceiver(mReceiver);
24 }

```

在代码 9-9 的接收器的接收方法中，从意向的扩展控件获取所包含的远程蓝牙设备信息（第 9 行）并添加到设备列表中（第 12 行）；当应用程序退出时需要注销该接收器。

（3）检测远程蓝牙设备

代码 9-10 是检测远程蓝牙设备的主要代码，其中主要包括启动检测、等待检测和停止检测。



代码 9-10 检测远程蓝牙设备

文件名: BtClientAct.java

```

1  @SuppressWarnings("unchecked")
2  private void doFind() { //检测远程蓝牙设备
3      mAdapter.startDiscovery(); //启动检测
4      //等待检测完毕
5      while(mAdapter.isDiscovering() ) {
6          try { Thread.sleep(100);
7              } catch (InterruptedException e) { e.printStackTrace(); }
8      }
9      //停止检测
10     mAdapter.cancelDiscovery();
11     //更新远程蓝牙设备列表
12     ListView devList = ((ListView)findViewById(R.id.dev_list));
13     ((ArrayAdapter<String>)devList.getAdapter()).notifyDataSetChanged();
14 }

```

在代码 9-10 中, 使用蓝牙设备适配器的 `startDiscovery` 方法启动对远程设备的检测 (第 3 行), 启动检测时还需要等待检测完毕 (第 5~8 行)。直到适配器不处于检测状态, 才能停止检测 (第 10 行), 并更新远程设备列表 (第 13 行)。

(4) 与远程蓝牙设备通信

代码 9-11 是客户端与远程蓝牙设备通信的主要代码。当用户单击设备列表中的某项时, 即发起与该远程蓝牙设备的通信。

代码 9-11 客户端与远程蓝牙设备通信

文件名: BtClientAct.java

```

1  @Override
2  public void onItemClick(AdapterView<?> parent, View v, int pos, long id) {
3      printText("设备总数: "+this.mDevices.size());
4      doSend(this.mDevices.get(pos));
5  }
6
7  private void doSend(BluetoothDevice dev) { //向服务端发送消息
8      try { BluetoothSocket socket =
9          dev.createRfcommSocketToServiceRecord(IConfig.BT_UUID);
10         printText("地址: "+socket.getRemoteDevice().getAddress());
11         printText("名称: "+socket.getRemoteDevice().getName());
12         talk(socket);
13         socket.close();
14     } catch (IOException e) { e.printStackTrace(); }
15 }
16
17 private void talk(BluetoothSocket socket) throws IOException { //与服务端通信

```



```

18      //获取套接字输入/输出流
19      OutputStream os = socket.getOutputStream();
20      InputStream is = socket.getInputStream();
21      //发送消息
22      PrintWriter pw = new PrintWriter(os);
23      pw.println(IConfig.HELLO);
24      pw.flush();
25      //读取服务端返回
26      BufferedReader br = new BufferedReader(new InputStreamReader(is));
27      String line = null;
28      while((line=br.readLine()) != null) { printText(line); }
29      //关闭输入/输出流
30      br.close();
31      pw.close();
32  }

```

在代码 9-11 中，客户端使用所检测到的远程蓝牙设备接口创建与服务端的 RFCOMM 通信套接字（第 8 行），客户端与服务端通过该套接字进行通信（talk 方法）。

（5）使用许可

对于开启蓝牙功能并进行通信，需要拥有蓝牙的使用许可，即必须在程序清单文件中声明该使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
```

9.4 近距离通信（NFC）

9.4.1 近距离通信概述

近距离通信（Near Field Communication, NFC）是一种短距离无线技术，通常要求小于或等于 4cm 的距离。近距离通信工作在 13.56MHz 频率，传输速度范围为 106~848kbit/s。近距离通信涉及一个发起方和一个目标，发起方生产射频场驱动目标。近距离通信的目标一般采取简单的形式，如电子标签或 IC 卡等这些无需电源的介质。如果两个设备都有电源供电，也可以通过近距离对等通信。

与其他无线技术（如蓝牙或 Wi-Fi）相比，近距离通信所提供的带宽和距离要低得多，但其实现成本低，目标无需电源，也无需检测或配对、无源目标。

对于手机设备而言，近距离通信设备通常作为发起方，也就是近距离读/写卡器。手机近距离通信设备会主动寻找标签，并对其信息进行处理。

近距离通信技术可以用于新兴的手机支付应用，其基本原理是将用户手机的 SIM 卡与本人的支付卡账号建立对应关系，业务系统通过 SIM 来识别用户，通过近距离通信来获取支付信息，通过无线通信（短消息方式）发送支付指令到网络运营商指定的服务点，再由网络运营商与银行机构之间完成请求扣款操作，并将结果以无线通信的方式反馈给用户，其过



程如图 9-9 所示。

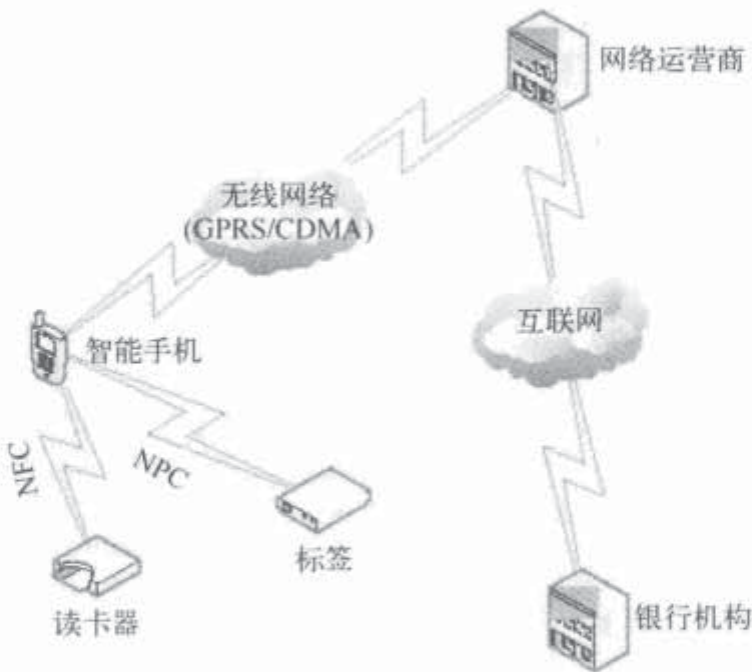


图 9-9 手机支付应用示意图

9.4.2 Android 平台对近距离通信的支持

Android 平台提供了近距离通信包（android.nfc）和相关技术包（android.nfc.tech）用于近距离通信应用。表 9-3 是对近距离通信相关包中重要类/接口的说明。

表 9-3 近距离通信相关包中重要类/接口及说明

类/接口	说 明
NfcManager	近距离通信管理器
NfcAdapter	近距离通信适配器
NdefMessage	近距离通信消息，一条消息可包含多条记录
NdefRecord	近距离通信数据记录
Tag	近距离通信标签
NdefFormatable	用于对标签时的格式化数据的访问

9.4.3 近距离通信的模式

近距离通信都是基于标签调度，存在两种应用模式。

1. 使用系统意向调度（后台）

该模式是通过在程序清单文件中设置主 Activity 组件的意向过滤器，预先设置好处理指定数据类型及技术规格的意向动作。当系统扫描到包含近距离通信数据的标签时，就会自动调用相应的处理意向。

2. 使用前台调度

该模式是在 Activity 组件中显式地定义近距离通信的意向过滤器，并允许前台调度。当系统扫描到包含近距离通信数据的标签时，就会调用该 Activity 组件的回调方法进行处理。

9.4.4 标签调度系统

1. 使用系统意向调度

使用系统意向调度主要是设置应用程序清单文件。代码 9-12 是示例程序的应用程序清单文件内容。

代码 9-12 应用程序清单文件

文件名: AndroidManifest.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="foolstudio.demo.wireless.nfcdemo"
4      android:versionCode="1"
5      android:versionName="1.0">
6      <uses-sdk android:minSdkVersion="12" />
7      <uses-permission android:name="android.permission.NFC"/>
8      <uses-feature android:name="android.hardware.nfc" android:required="true"/>
9      <application android:icon="@drawable/icon" android:label="@string/app_name">
10         <activity android:name=".SimpleNfcAct" android:label="@string/app_name">
11             <intent-filter>
12                 <action android:name="android.intent.action.MAIN" />
13                 <category android:name="android.intent.category.LAUNCHER" />
14             </intent-filter>
15             <intent-filter>
16                 <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
17                 <data android:mimeType="text/plain"/>
18             </intent-filter>
19             <intent-filter>
20                 <action android:name="android.nfc.action.TAG_DISCOVERED"/>
21             </intent-filter>
22             <intent-filter>
23                 <action android:name="android.nfc.action.TECH_DISCOVERED"/>
24                 <meta-data android:name="android.nfc.action.TECH_DISCOVERED"
25                     android:resource="@xml/nfc_tech_filter" />
26             </intent-filter>
27         </activity>
28     </application>
29 </manifest>

```

在代码 9-12 中, 使用近距离通信需要相应的使用许可, 即需要在程序清单文件中声明该许可 (第 7 行), 此外还需要声明使用特性 (第 8 行)。

代码 9-12 中定义了 3 个有关近距离通信的意向: 第一个是通信数据检测完毕 (第 16 行); 第二个是标签检测完毕 (第 20 行); 第三个是所支持技术检测完毕 (第 23 行)。其中, 数据检测还需要指明数据类型 (第 17 行), 通信模块不可能检测所有的数据类型; 技术检测同样需要指明技术列表, 该列表使用 XML 资源定义 (第 25 行), 如代码 9-13 所示。



代码 9-13 近距离通信技术过滤列表

文件名: nfc_tech_filter.xml

```

1  <resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
2      <tech-list>
3          <tech>android.nfc.tech.IsoDep</tech>
4          <tech>android.nfc.tech.NfcA</tech>
5          <tech>android.nfc.tech.NfcB</tech>
6          <tech>android.nfc.tech.NfcF</tech>
7          <tech>android.nfc.tech.NfcV</tech>
8          <tech>android.nfc.tech.Ndef</tech>
9          <tech>android.nfc.tech.NdefFormatable</tech>
10         <tech>android.nfc.tech.MifareClassic</tech>
11         <tech>android.nfc.tech.MifareUltralight</tech>
12     </tech-list>
13 </resources>

```

代码 9-13 中所定义的记录类型可以参考标签技术接口 (TagTechnology) 的说明文档, 这些类都在近距离通信技术包中定义。

2. 使用前台调度

前台调度主要是在 Activity 组件的框架中添加相关代码来实现对标签的检测及数据的处理。代码 9-14 是示例程序的 Activity 组件的框架定义。

代码 9-14 近距离通信示例程序 Activity 组件的框架定义

文件名: SimpleNfcAct.java

```

1  public class SimpleNfcAct extends Activity {
2      //本地适配器
3      private NfcAdapter mNfcAdapter = null;
4      private PendingIntent mPdIntent = null;
5      private IntentFilter[] mFilters = null;
6      private String[][] mTechLists = null;
7
8      @Override
9      public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.main);
12         //获取适配器实例
13         mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
14         if(mNfcAdapter==null) {
15             Toast.makeText(this,"获取 NFC 适配器失败! ", Toast.LENGTH_LONG).show();
16             return;
17         }
18         //未决意向
19         mPdIntent = PendingIntent.getActivity(this, 0,
20             new Intent(this, getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP),0);

```



```

21      //意向过滤器
22      IntentFilter ndef = new IntentFilter(NfcAdapter.ACTION_NDEF_DISCOVERED);
23      try { ndef.addDataType("*/*"); }
24      catch (MalformedMimeTypeException e) { e.printStackTrace(); }
25      mFilters = new IntentFilter[] { ndef };
26      //技术列表
27      mTechLists = new String[][] { new String[] { NfcF.class.getName() } };
28  }
29  @Override
30  protected void onResume() { super.onResume();
31      //启动前台调度
32      mNfcAdapter.enableForegroundDispatch(this, mPdIntent, mFilters, mTechLists);
33  }
34  @Override
35  protected void onPause() { super.onPause();
36      //取消前台调度
37      mNfcAdapter.disableForegroundDispatch(this);
38  }
39      .....
40  };

```

在代码 9-14 中，首先使用近距离通信适配器的 `getDefaultAdapter` 方法获取本地适配器（第 13 行），继而使用该适配器启动或取消前台调度（第 32 行或第 37 行）。

启动前台调度需要指明未决意向（用于指明处理组件）、意向过滤器（用于捕获系统有关近距离通信的消息意向）和技术列表。至此可知，前台调度所需的要素与系统意向调度相同，其区别在于前者使用 Java 代码定义，后者采用 XML 定义。

9.4.5 标签处理

1. 读取标签数据

Activity 组件对近距离通信数据的处理在回调方法 `onNewIntent` 中进行，其主要内容包括读取标签内容和写数据到标签中。代码 9-15 是读取标签中近距离通信消息的主要代码，该消息从所传入的意向进行抽取。

代码 9-15 获取近距离通信消息

文件名: SimpleNfcAct.java

```

1  @Override
2  protected void onNewIntent(Intent intent) {
3      //获取标签
4      Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
5      Toast.makeText(this, tag.toString(), Toast.LENGTH_LONG).show();
6      //获取消息
7      NdefMessage[] msgs = getNdefMsgs(intent);
8      for(int i = 0; i < msgs.length; ++i) {

```




Android 平台开发之旅 第2版

```

9          //获取记录
10         NdefRecord[] records = msgs[i].getRecords();
11         for(int j = 0; j < records.length; ++j) { showRecord(records[j]); }
12     }
13     //写标签
14     doWriteTag(tag);
15 }
16
17 //获取意向对象中的消息
18 private NdefMessage[] getNdefMsgs(Intent intent) {
19     NdefMessage[] msgs = null;
20     final String action = intent.getAction();
21     if(NfcAdapter.ACTION_TAG_DISCOVERED.equals(action)) {
22         final String extra = NfcAdapter.EXTRA_NDEF_MESSAGES;
23         Parcelable[] rawMsgs = intent.getParcelableArrayExtra(extra);
24         if(rawMsgs != null) { msgs = new NdefMessage[rawMsgs.length];
25             for (int i = 0; i < rawMsgs.length; i++) {
26                 msgs[i] = (NdefMessage) rawMsgs[i];
27             }
28         }
29     }
30     return msgs;
31 }
32
33 private void showRecord(NdefRecord record) { //显示记录内容
34     print("Id: "+record.getId().toString());
35     print("Payload: "+record.getPayload().toString());
36     print("TNF: "+record.getTnf() );
37     print("Type: "+record.getType().toString());
38 }

```

在代码 9-15 中，标签描述从意向中抽取（第 4 行），标签中所装载的数据（原消息）也从意向中抽取（第 23 行）。抽取出来的原消息还要经过拆分才能获得标准的消息内容（第 24~28 行）。一条消息中可以包含多条记录，记录是近距离通信中最小数据单元（第 10 行和第 11 行）。

2. 写数据到标签

代码 9-16 是写数据到标签的主要代码，该过程与读取标签数据是互逆的，其思路是将要写入的数据按照先构造记录再合成消息，然后写入标签中。

代码 9-16 写数据到标签

文件名: SimpleNfcAct.java

```

1 private void doWriteTag(Tag t) { //写标签
2     NdefFormatable tag = NdefFormatable.get(t);
3     Locale locale = Locale.US;
4     byte[] langBytes = locale.getLanguage().getBytes(Charset.forName("US-ASCII"));

```



```
5      String text = "Hello, NFC!";
6      byte[] textBytes = text.getBytes(Charset.forName("UTF-8"));
7      int utfBit = 0;
8      byte status = (byte) (utfBit + langBytes.length);
9      byte[] data = FooNfcUtil.getInstance().mergeNdefData(status, langBytes, textBytes);
10     final short tnf = NdefRecord.TNF_WELL_KNOWN;
11     NdefRecord record = new NdefRecord(tnf, NdefRecord.RTD_TEXT, new byte[0], data);
12     try { NdefRecord[] records = { record };
13         NdefMessage message = new NdefMessage(records);
14         tag.connect();
15         tag.format(message);
16     } catch (Exception e) { e.printStackTrace(); }
17 }
```

在代码 9-16 中，先构建数据记录（第 11 行），再构建数据消息（第 13 行），最后使用近距离通信格式化数据操作类（NdefFormatable）实例的 format 方法向标签中写入标准数据。



第 10 章 多媒体应用

本章对 Android 平台提供的多媒体应用进行实机示例，包括音频回放、录音、视频回放、录制视频、相机应用、媒体信息管理等。此外，通过一个音乐盒工具的开发案例，介绍了整合网页浏览器、视频回放和音频回放的集成应用。

10.1 Android 平台对多媒体应用的支持

Android 平台提供了媒体包（`android.media`）来管理各种音频和视频的媒体接口，该包中提供的 API 除了能够回放而且还能录制媒体文件。这些媒体资源包括音频（MP3 和其他音乐文件、响铃、游戏音效或 DTMF 响铃）和视频（从本地存储器中获取或经由网络的视频流）。表 10-1 是媒体包中重要的类/接口的说明。

表 10-1 媒体包中重要的类/接口的说明

类/接口	说 明
AudioManager	音频管理器用于管理音量和响应模式控制
AudioRecord	用于管理程序通过从音频输入设备所录制的音频信息
AsyncPlayer	异步播放器，用于播放一串音频资源标识
MediaPlayer	用于控制音频或视频文件和流的回放
MediaRecorder	用于录制音频和视频
RingtoneManager	用于访问响铃、通知和其他类型的声音
Ringtone	提供了一个快速播放响铃、通知或其他相同类型的声音
SoundPool	用于管理和播放应用程序的音频资源
ToneGenerator	用于播放 DTMF 响铃

在硬件包（`android.hardware`）中提供了用于访问相机服务的工具类，用于获取相机图片、控制照片拍摄过程等。表 10-2 是硬件包中多媒体相关的类/接口的说明。

表 10-2 硬件包中多媒体相关的类/接口的说明

类/接口	说 明
Camera	用于连接/断开摄像头服务
Camera.PictureCallback	获得照片时回调
Camera.PreviewCallback	预览时回调
Camera.ShutterCallback	快门关闭时回调

另外，在部件包（`android.widget`）中提供了视频视图组件（`VideoView`）用于显示视频文件。通过该组件，开发者可以将视频回放界面集成到自己的应用程序中。表 10-3 是对该

包中多媒体相关的类/接口的说明。

表 10-3 部件包中多媒体相关的类/接口的说明

类/接口	说 明
VideoView	视频视图，用于播放一个视频文件
MediaController	媒体播放控制器面板

10.2 音频回放与录制

10.2.1 音频回放

使用媒体播放器（MediaPlayer）可以回放音频媒体。图 10-1 是一款音乐播放器的实机界面。该界面提供了一个音乐资源列表，用户可以通过 3 种方式来控制播放：直接点选播放条目、通过播放按钮和拖动播放进度条。



图 10-1 音乐播放器的实机界面

1. 应用程序主 Activity 框架

代码 10-1 是音乐播放器示例程序的主 Activity 组件框架，该组件继承于列表 Activity，同时实现了按钮单击侦听器、媒体播放完成侦听器和滑动条改变侦听器。

代码 10-1 示例程序的主 Activity 组件的框架定义

文件名：MusicPlayerAct.java

```
1 public class MusicPlayerAct extends ListActivity implements OnClickListener,
2                                     OnCompletionListener, OnSeekBarChangeListener {
3     //播放器
4     private MediaPlayer mPlayer = null;
5     //播放控制按钮
6     private ImageButton mBtnPrev = null;
7     private ImageButton mBtnNext = null;
8     private ImageButton mBtnPlay = null;
9     //播放状态相关
```



Android 平台开发之旅 第2版

```

10     private SeekBar mPBar = null;
11     private Handler mHandler = null;
12     private int mStatus = IConfig.STATUS_PAUSED;
13     private int mCurPos = 0;
14     private Timer mPlayTimer = new Timer();
15     private int mPlayPos = -1;
16     //音乐文件路径列表
17     private final String[] mFileNames = FooFileUtil.getInstance().list(IConfig.PATH, ".mp3");
18
19     @Override
20     public void onCreate(Bundle savedInstanceState) {
21         super.onCreate(savedInstanceState); setContentView(R.layout.main);
22         //获取按钮控件并设置单击侦听
23         mBtnPrev = (ImageButton)findViewById(R.id.btn_prev);
24         mBtnNext = (ImageButton)findViewById(R.id.btn_next);
25         mBtnPlay = (ImageButton)findViewById(R.id.btn_play);
26         mBtnPrev.setOnClickListener(this); mBtnNext.setOnClickListener(this);
27         mBtnPlay.setOnClickListener(this);
28         //获取播放进度条并设置进度改变事件侦听器
29         mPBar = (SeekBar)findViewById(R.id.seekbar);
30         mPBar.setOnSeekBarChangeListener(this);
31         //设置列表视图适配器
32         ListAdapter adapter = new ArrayAdapter<String>(this,
33             android.R.layout.simple_list_item_1, mFileNames);
34         setListAdapter(adapter);
35         //初始化
36         init();
37         //设置当前所选条目
38         this.getListView().setSelected(true);
39         this.getListView().setSelection(this.mCurPos);
40     }
41
42     private void init() { //初始化播放器及主线程消息队列处理器
43         mPlayer = new MediaPlayer(); mPlayer.setOnCompletionListener(this);
44
45         mHandler = new Handler() { //初始化主线程消息队列处理器
46             @Override
47             public void handleMessage(Message msg) { Bundle data = msg.getData();
48                 mPBar.setProgress(data.getInt(IConfig.EXTRA));
49                 ((TextView)findViewById(R.id.progress)).setText(
50                     FooSysUtil.getInstance().toTimeStr2(mPBar.getProgress())+"/"+
51                     FooSysUtil.getInstance().toTimeStr2(mPBar.getMax()));
52             }
53         };
54     }
55

```



```
56      @Override
57      public void onClick(View v) { //播放控制按钮：前一首、播放/暂停、后一首
58          switch(v.getId()) {      case R.id.btn_prev: { doPrev(); break; }
59                                  case R.id.btn_play: { doPlay(); break; }
60                                  case R.id.btn_next: { doNext(); break; }
61          }
62      }
63      .....
64  };
```

在代码 10-1 中，Activity 组件定义包括播放控制按钮设置（第 23~27 行）、播放进度条设置（第 29 行和第 30 行）、播放列表初始化（第 32~34 行）和初始化播放器以及设置播放状态侦听器（第 43 行）。

表 10-4 是媒体播放相关的侦听器/回调接口的说明（按照常规的设置顺序）。

表 10-4 播放相关的侦听器/回调接口的说明

接 口	说 明
SurfaceHolder.Callback	表面控制器回调接口，用于获取表面创建的时机
MediaPlayer.OnPreparedListener	播放器就绪侦听器，用于响应播放器就绪事件
View.OnClickListener	按钮单击侦听器，用于响应按钮单击事件
SeekBar.OnSeekBarChangeListener	滑动条改变侦听器，用于绑定播放进度的改变
MediaPlayer.OnCompletionListener	播放器完成侦听器，用于响应播放结束事件

此外，代码中所定义的主线程消息队列处理器用于接收并显示播放进度（第 48~51 行）。

2. 主界面布局定义

代码 10-2 是音乐播放器示例程序的主界面布局定义，该布局总体为垂直方向的线性布局，从上到下依次包括播放控制按钮、播放进度条和播放列表。

代码 10-2 主界面布局定义

文件名：main.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <LinearLayout android:orientation="horizontal"
6          android:layout_width="fill_parent" android:layout_height="wrap_content">
7          <ImageButton android:src="@drawable/prev" android:id="@+id/btn_prev"
8              android:layout_width="wrap_content" android:layout_height="wrap_content"/>
9          <ImageButton android:src="@drawable/play" android:id="@+id/btn_play"
10             android:layout_width="wrap_content" android:layout_height="wrap_content"/>
11          <ImageButton android:src="@drawable/next" android:id="@+id/btn_next"
12             android:layout_width="wrap_content" android:layout_height="wrap_content"/>
13      <TextView android:id="@+id/text" android:layout_gravity="center_vertical"
```




```

14         android:layout_height="wrap_content" android:layout_width="wrap_content"
15         android:paddingLeft="2pt"/>
16     </LinearLayout>
17     <TableLayout android:stretchColumns="1"
18         android:layout_width="match_parent" android:layout_height="wrap_content">
19         <TableRow>
20             <TextView android:id="@+id/progress" android:text="00:00"
21                 android:layout_height="wrap_content" android:layout_width="match_parent"
22                 android:layout_gravity="center_vertical"/>
23             <SeekBar android:id="@+id/seekbar"
24                 android:layout_height="wrap_content" android:layout_width="match_parent"
25                 android:paddingLeft="4pt" android:paddingRight="4pt"/>
26         </TableRow>
27     </TableLayout>
28     <ListView android:id="@+id/android:list" android:layout_width="fill_parent"
29         android:layout_height="fill_parent"/>
30     <TextView android:id="@+id/android:empty" android:layout_width="fill_parent"
31         android:layout_height="fill_parent" android:text="无记录" />
32 </LinearLayout>

```

3. 应用程序配置信息接口

代码 10-3 是应用程序中有关配置信息接口的定义，包括音乐文件路径、播放状态类型等。

代码 10-3 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String PATH = "/sdcard/audio/";
3      //播放状态
4      public static final int STATUS_PLAYING = 0;
5      public static final int STATUS_PAUSED = 1;
6      //扩展数据项名
7      public static final String EXTRA = "progress";
8  };

```

4. 播放控制

在音乐播放器示例程序中，使用了 3 种播放控制方式：使用按钮进行播放导航、直接点播放列表中的条目进行播放切换和拖动滑动条来改变播放进度。

(1) 播放导航

示例程序中的播放导航按钮包括播放前一首和后一首以及播放/暂停切换，如图 10-2 所示。



图 10-2 播放导航按钮

代码 10-4 是进行播放导航的主要代码。

代码 10-4 播放导航

文件名: MusicPlayerAct.java

```
1 private void doPrev() { //播放前一条
2     if(mCurPos < 1) { mCurPos = (this.getListAdapter().getCount()-1); }
3     else { mCurPos--; }
4     this.getListView().setSelected(true); this.getListView().setSelection(mCurPos);
5     //启动播放
6     playMusic(mCurPos);
7     mPlayPos = -1;
8     updateButtons();
9 }
10
11 private void doNext() { //播放后一条
12     if(mCurPos >= (this.getListAdapter().getCount()-1)) { mCurPos = 0; }
13     else { mCurPos++; }
14     this.getListView().setSelected(true); this.getListView().setSelection(mCurPos);
15     //启动播放
16     playMusic(mCurPos);
17     mPlayPos = -1;
18     updateButtons();
19 }
20
21 private void doPlay() { //播放/暂停切换
22     if(mStatus == IConfig.STATUS_PAUSED) { //当前为暂停状态则恢复播放
23         playMusic(mCurPos);
24         if(mPlayPos != -1) { mPlayer.seekTo(mPlayPos); }
25         this.mStatus = IConfig.STATUS_PLAYING;
26         this.mBtnPlay.setImageResource(R.drawable.pause);
27     } else if(mStatus == IConfig.STATUS_PLAYING) { //当前为播放则暂停
28         this.mPlayer.pause();
29         mPlayPos = mPlayer.getCurrentPosition();
30         this.mStatus = IConfig.STATUS_PAUSED;
31         this.mBtnPlay.setImageResource(R.drawable.play);
32     }
33     //设置当前所选条目
34     this.getListView().setSelected(true); this.getListView().setSelection(mCurPos);
35 }
36
37 private void updateButtons() { //更新按钮图标及播放状态
38     this.mBtnPlay.setImageResource(R.drawable.pause);
39     this.mStatus = IConfig.STATUS_PLAYING;
40 }
```

在代码 10-4 中, 通过控制当前播放索引 (mCurPos) 和播放状态 (mStatus) 来进行播



放导航。在第 28 行中，使用播放器的 `pause` 方法暂停播放动作。

(2) 播放切换

代码 10-5 是在播放列表中通过单击列表项来实现播放切换的主要代码。

代码 10-5 在播放列表中实现播放切换

文件名: `MusicPlayerAct.java`

```
1  @Override
2  protected void onItemClick(ListView l, View v, int pos, long id) {
3      mCurPos = pos; mPlayer.stop();      playMusic(pos);
4  }
```

在代码 10-5 中，切换到另外一首音乐前需要先停止当前播放。使用播放器的 `stop` 方法停止当前的播放行为。

(3) 播放进度控制

使用按钮进行播放导航和使用列表进行播放切换都只是控制播放顺序，而播放进度是指在播放某一首音乐的过程中可以改变其播放的位置，如图 10-3 所示。



图 10-3 播放进度控制

代码 10-6 是改变播放进度的主要代码，当用户拖动滑动条（`SeekBar`）时，通过滑动条改变事件的回调方法可以获取拖动后的进度，从而调整播放位置。

代码 10-6 改变播放进度

文件名: `MusicPlayerAct.java`

```
1  @Override
2  public void onProgressChanged(SeekBar sb, int progress, boolean fromUser) {
3      if(fromUser) { mPlayer.seekTo(progress); }
4  }
5  @Override
6  public void onStartTrackingTouch(SeekBar sb) {}
7  @Override
8  public void onStopTrackingTouch(SeekBar sb) {}
```

在代码 10-6 中，在滑动条进度改变事件回调方法（`onProgressChanged`）中使用播放器的 `seekTo` 方法改变播放位置（第 3 行）。该回调方法的 `fromUser` 参数用于区分进度改变是否由用户触发，此处必须区分为用户触发。

需要说明的是，滑动条的最大值必须在播放开始之前设定，其为该音乐的播放时长。

5. 回放音频

代码 10-7 是回放音频媒体的主要代码。

代码 10-7 回放音频媒体

文件名: MusicPlayerAct.java

```
1 private boolean playMusic(int pos) { //播放指定路径的音乐
2     final String filename = this.getListAdapter().getItem(pos).toString();
3     return(playMusic(filename));
4 }
5
6 //播放指定路径的音乐
7 private boolean playMusic(String filename) {
8     try { mPlayer.reset();
9         mPlayer.setDataSource(IConfig.PATH+filename);
10        mPlayer.prepare();
11        mPlayer.start();
12        //设置当前播放
13        ((TextView)findViewById(R.id.text)).setText(filename);
14        startTimer();
15    } catch(IOException e) { return(false); }
16    return(true);
17 }
```

从代码 10-7 中可以看出，启动播放（第 11 行）前首先需要重置播放状态（第 8 行），再指定播放数据源（第 9 行），并进行预操作（第 10 行）。表 10-5 是回放音频媒体的主要步骤及说明。

表 10-5 回放音频媒体的主要步骤及说明

顺 序	操 作	说 明
1	setOnCompletionListener() setOnPreparedListener()	设置播放事件侦听器
2	reset()	重置播放器
3	setDataSource()	设置媒体源（可以是本地文件，也可以是网络资源）
4	prepare()	播放预备
5	start()	启动播放
6	pause()	暂停播放
7	stop()	停止播放
8	release()	释放播放器资源

启动播放之后，会显示当前播放的音乐名，并启动调整播放进度的计时器。因为播放器只能侦听播放完成事件（代码 10-1 第 43 行），而无法侦听播放进度改变事件，所以只能通过计时器去“保持”与播放进度同步。代码 10-8 是启动播放进度计时器的主要代码。

代码 10-8 启动播放进度计时器

文件名: MusicPlayerAct.java

```
1 private void startTimer() { //启动回放进度计时器
```




Android 平台开发之旅 第2版

```

2      TimerTask task = new TimerTask(){
3          public void run() {      Bundle data = new Bundle();
4              data.putInt(IConfig.EXTRA, mPlayer.getCurrentPosition());
5              Message msg = new Message();
6              msg.setData(data);
7              mHandler.sendMessage(msg);
8          }
9      };
10     this.mPBar.setMax(mPlayer.getDuration());
11     mPlayTimer.schedule(task, 0, 1000L);
12 }
13
14 @Override
15 public void onCompletion(MediaPlayer mp) { mp.stop(); doNext(); }

```

在代码 10-8 中，当启动播放后，首先获取当前播放时长并设置为进度条的最大值（第 10 行），然后启动播放计时器（第 11 行）。计时器会每隔一秒将当前播放进度发送给主线程消息队列处理器（第 4 行），再由该处理器去获取进度值（代码 10-1 中第 48 行）并显示。

代码 10-8 中第 15 行的 `onCompletion` 是播放完成侦听器的回调方法，该方法在当前播放正常完成后回调。

6. 释放播放器

因为播放器是共享资源，所以当不再进行播放时应该释放其相关资源。代码 10-9 是当 Activity 组件被销毁时进行的有关善后处理。

代码 10-9 释放播放器资源

文件名: MusicPlayerAct.java

```

1  @Override
2  protected void onDestroy() { super.onDestroy();
3      this.mPlayTimer.cancel(); this.mPlayTimer.purge();
4      this.mPlayer.release();
5  }

```

在代码 10-9 中，使用播放器的 `release` 方法可以释放播放器对象相关的资源（第 4 行）。此外，因为在播放进度控制计时器中会获取播放进度信息，所以必须在播放器释放之前停止播放进度计时器（第 3 行）。

10.2.2 录制音频

在 TOM 猫游戏中，TOM 猫会聆听玩家的话并复述出来。该功能实际上就是对玩家的语音进行录制并回放。使用媒体录制（`MediaRecorder`）可以录制音频输入设备（麦克风）所输入的音频。图 10-4 是一款录音应用示例程序的实机界面，该程序可以录制并回放音频。



图 10-4 录音应用示例程序的实机界面

1. 应用程序主 Activity 框架

代码 10-10 是该录音应用示例程序的主 Activity 组件的框架定义。

代码 10-10 示例程序的主 Activity 组件的框架定义

文件名: RecorderDemoAct.java

```

1  public class RecorderDemoAct extends Activity implements OnClickListener,
2                                     OnCompletionListener {
3      //录音机
4      private MediaRecorder mRecorder = null;
5      //播放器
6      private MediaPlayer mPlayer = null;
7      //录音/回放控制按钮
8      private ImageButton mBtnRec = null;
9      private ImageButton mBtnPlay = null;
10     //播放状态
11     private ProgressBar mPBar = null;
12     private Handler mHandler = null;
13     private int mStatus = IConfig.STATUS_PAUSED;
14     private Timer mPlayTimer = null;
15     private int mPlayPos = -1;
16     private int mCounter = 0;
17
18     @Override
19     public void onCreate(Bundle savedInstanceState) {
20         super.onCreate(savedInstanceState);
21         setContentView(R.layout.main);
22         //获取按钮控件并设置单击侦听
23         mBtnRec = (ImageButton)findViewById(R.id.btn_rec);
24         mBtnPlay = (ImageButton)findViewById(R.id.btn_play);
25         mBtnRec.setOnClickListener(this);
26         mBtnPlay.setOnClickListener(this);
27         //播放进度条
28         mPBar = (ProgressBar)findViewById(R.id.pbar);
29         //初始化录音机
30         mRecorder = new MediaRecorder();
31         //初始化播放器
32         mPlayer = new MediaPlayer();
33         mPlayer.setOnCompletionListener(this);
34         //初始化线程消息处理器

```



Android 平台开发之旅 第2版

```

35         mHandler = new Handler() {
36             @Override
37             public void handleMessage(Message msg) {
38                 super.handleMessage(msg);
39                 //分解接收到的消息
40                 Bundle data = msg.getData();
41                 switch(msg.what) {
42                     case IConfig.PLAYBACK: { //录音回放
43                         .....
44                     }
45                     case IConfig.RECORDING: { //录制音频
46                         int pos = data.getInt(IConfig.EXTRA);
47                         mPBar.setProgress(pos);
48                         ((TextView)findViewById(R.id.progress)).setText(
49                             FooSysUtil.getInstance().toTimeStr2(pos));
50                         if(pos>=IConfig.MAX_DURATION) { stopRecord(); }
51                         break;
52                     }
53                 }
54             }
55         };
56     }
57
58     @Override
59     public void onClick(View v) { //录制和回放按钮
60         switch(v.getId()) {
61             case R.id.btn_rec: { doRecord(); break; }
62             case R.id.btn_play: { doPlayback(); break; }
63         }
64         .....
65     };

```

在代码 10-10 中，录音机对象用于录制音频（第 30 行），播放器对象用于回放录音（第 32 行）。其中，主线程消息队列处理器用于接收并显示回放和录音进度（第 35~55 行）。

提示：由于录音应用示例程序中的音频回放部分与音乐播放器示例程序的相同，在此不予重复，读者可以参考音乐播放示例程序。

2. 主界面布局定义

代码 10-11 是录音应用示例程序的主界面布局定义，其中主要包括录制/回放控制按钮和进度条。

代码 10-11 录音应用示例程序的主界面布局定义

文件名：main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

```



```

3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <TableLayout android:stretchColumns="0,1"
6          android:layout_width="fill_parent" android:layout_height="wrap_content">
7          <TableRow>
8              <ImageButton android:src="@drawable/record" android:id="@+id/btn_rec"
9                  android:layout_width="wrap_content" android:layout_height="wrap_content"/>
10             <ImageButton android:src="@drawable/play" android:id="@+id/btn_play"
11                 android:layout_width="wrap_content" android:layout_height="wrap_content"/>
12             </TableRow>
13         </TableLayout>
14         <TableLayout android:stretchColumns="1"
15             android:layout_width="fill_parent" android:layout_height="wrap_content">
16             <TableRow>
17                 <TextView android:id="@+id/progress" android:layout_gravity="center_vertical"
18                     android:layout_width="wrap_content" android:layout_height="wrap_content"
19                     android:text="00'00"/00'00""/>
20                 <ProgressBar android:id="@+id/pbar"
21                     style="?android:attr/progressBarStyleHorizontal"
22                     android:layout_height="wrap_content" android:layout_width="match_parent"
23                     android:paddingLeft="2pt" android:paddingRight="2pt"/>
24             </TableRow>
25         </TableLayout>
26     </LinearLayout>

```

3. 应用程序配置信息接口

代码 10-12 是应用程序中有关配置信息接口的定义，包括录音文件路径、播放状态类型、最大录音时长等。

代码 10-12 应用程序配置信息接口的定义

文件名: IConfig.java

```

1      public interface IConfig {
2          public static final String REC_AUDIO_FILE = "/sdcard/audio/recording.3gp";
3          //消息类型
4          public static final int RECORDING= 1;
5          public static final int PLAYBACK= 2;
6          //播放状态
7          public static final int STATUS_RECORDING = 1;
8          public static final int STATUS_PAUSED = 2;
9          public static final int STATUS_PLAYING = 3;
10         //扩展数据项名
11         public static final String EXTRA = "progress";
12         //最长录制时间 (ms)
13         public static final int MAX_DURATION = 1*60*1000;
14     };

```



4. 录制音频

示例程序中的录音控制包括启动录音、录音进度显示和停止录音。

(1) 启动录音

代码 10-13 是代码 10-10 中启动录音的主要代码。

代码 10-13 启动录音

文件名: RecorderDemoAct.java

```

1  private void doRecord() {
2      if(this.mStatus == IConfig.STATUS_PAUSED) {
3          //录制准备
4          prepareRecord(mRecorder);
5          //开始录制
6          mRecorder.start();
7          mCounter = 0;
8          startTimer1();
9          //更新状态
10         mBtnRec.setImageResource(R.drawable.stop);
11         mStatus = IConfig.STATUS_RECORDING;
12     } else if(this.mStatus == IConfig.STATUS_RECORDING) {
13         stopRecord();
14     }
15 }
16
17 private void prepareRecord(MediaRecorder mr) {
18     //设置音频源
19     mr.setAudioSource(MediaRecorder.AudioSource.MIC);
20     //设置输出格式
21     mr.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
22     //设置编码格式
23     mr.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
24     //设置输出文件路径
25     mr.setOutputFile(IConfig.REC_AUDIO_FILE);
26     try { mr.prepare();
27     } catch (IllegalStateException e) {e.printStackTrace(); }
28     catch (IOException e) { e.printStackTrace(); }
29 }

```

在代码 10-13 中, 在录音开始 (第 6 行) 前需要进行预处理 (第 4 行)。其中, 过程包括设置音频源、输出格式、编码方式及输出路径 (从第 19~25 行), 且当这些设置完毕后还需通过录音机对象的 `prepare` 方法来执行预操作 (第 26 行)。表 10-6 是录制音频媒体的主要步骤及说明。

表 10-6 录制音频媒体的主要步骤及说明

顺 序	操 作	说 明
1	setAudioSource()	设置音频源 (MIC)
2	setOutputFormat()	设置输出格式
3	setAudioEncoder()	设置音频解码类型
4	setOutputFile()	设置输出文件
5	prepare()	录制预备
6	start()	启动录制
7	pause()	暂停录制
8	stop()	停止录制
9	release()	释放录音机资源

表 10-7 是录音机接口所支持的音频源类型及说明。

表 10-7 录音机接口所支持的音频源类型及说明

类 型	说 明
DEFAULT	系统音频源
MIC	麦克风

表 10-8 是录音机接口所支持的音频编码类型及说明

表 10-8 录音机接口所支持的编码方式及说明

类 型	说 明
AMR_NB	AMR 窄带
DEFAULT	默认编码

表 10-9 是录音机接口所支持的音频输入格式类型及说明。

表 10-9 录音接口所支持的输出格式类型及说明

类 型	说 明
DEFAULT	系统默认格式
MPEG_4	MPEG4 格式
RAW_AMR	原 AMR 格式文件
THREE_GPP	3GP 格式

(2) 录音进度显示

因为录音机无法侦听录音进度改变事件，所以只能通过计时器去“保持”与录音进度同步。代码 10-14 是启动录音进度计时器的主要代码。

代码 10-14 启动录音进度计时器

文件名: RecorderDemoAct.java

```
1 private void startTimer1() { //启动计时器
```



Android 平台开发之旅 第2版

```

2      mPlayTimer = new Timer();
3
4      TimerTask task = new TimerTask(){
5          public void run() {
6              Bundle data = new Bundle();
7              data.putInt(IConfig.EXTRA, 1000*(mCounter++));
8              Message msg = new Message();
9              msg.setData(data);
10             msg.what = IConfig.RECORDING;
11             mHandler.sendMessage(msg);
12         }
13     };
14     this.mPBar.setMax(IConfig.MAX_DURATION);
15     mPlayTimer.schedule(task, 0, 1000L);
16 }

```

在代码 10-14 中，当启动录音后，首先将最大录音时长设置为进度条的最大值（第 14 行），然后启动录音计时器（第 15 行）。计时器会每隔一秒将当前录音进度发送给主线程消息队列处理器（第 7 行），再由该处理器去获取进度值（代码 10-10 中第 46 行）并显示。

（3）停止录音

代码 10-15 是代码 10-13 中提到的停止录音的主要代码。

代码 10-15 停止录音

文件名: RecorderDemoAct.java

```

1  private void stopRecord() {
2      //停止录音计时器
3      this.mPlayTimer.cancel(); this.mPlayTimer.purge();
4      //停止录制并重置录音机（为下次录音做准备）
5      mRecorder.stop(); mRecorder.reset();
6      //初始化计时起点
7      mCounter = 0;
8      //更新状态
9      mBtnRec.setImageResource(R.drawable.record);
10     mStatus = IConfig.STATUS_PAUSED;
11
12     mPlayPos = -1;
13 }

```

在代码 10-15 中，使用录音机对象的 stop 方法可以停止录音；同时为了下一次的录音，还需要重置录音机（第 5 行）。

（4）释放录音机对象

因为录音机也是共享资源，所以当不再进行录音时应该释放相关资源。代码 10-16 是当应用程序被销毁时所进行的有关善后处理。

代码 10-16 释放录音机对象

文件名: RecorderDemoAct.java

```
1  @Override
2  protected void onDestroy() { super.onDestroy();
3      if(this.mPlayTimer!= null) { this.mPlayTimer.cancel(); this.mPlayTimer.purge(); }
4      //释放播放器和录音机对象相关资源
5      mRecorder.release();
6      mPlayer.release();
7  }
```

在代码 10-16 中，使用录音机的 release 方法可以释放播放器对象相关的资源（第 5 行）。

5. 录音使用许可

录制音频需要具备相应的许可，即需要在应用程序清单文件中声明录制音频的使用许可，其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

10.2.3 音频管理

音频管理器（AudioManager）用于管理系统音量和响铃模式。图 10-5 是通过音频管理器获取和设置音频流音量的实机界面。



图 10-5 音频管理器的实机界面

1. 音频管理器

获取音频管理器的代码如下所示。

```
AudioManager mMgr =
    (AudioManager)(this.getSystemService(Context.AUDIO_SERVICE));
```

2. 获取音量

获取音量包括两种情形：获取最大音量和获取当前音量。



(1) 获取最大音量

使用音频管理器的 `getStreamMaxVolume` 方法可以获得指定类型音频流的最大音量值。该方法仅有的一个参数是音频流类型。

表 10-10 是 Android 平台所定义的音频流类型及说明，这些类型在音频管理器类中定义。

表 10-10 音频流类型及说明

类 型	说 明
STREAM_ALARM	警报音频流
STREAM_DTMF	DTMF 音调
STREAM_MUSIC	音乐音频流
STREAM_NOTIFICATION	通知音频流
STREAM_RING	响铃音频流
STREAM_SYSTEM	系统音频流
STREAM_VOICE_CALL	电话呼叫音频流

(2) 获取当前音量

通过音频管理器的 `getStreamVolume` 方法可以获得指定类型音频流的当前音量值。代码 10-17 是获取呼叫音量最大值和设定值并通过滑动条进行显示的实例代码。

代码 10-17 获取音频流音量

文件名: `AudioServiceAct.java`

1	<code>int max = mMgr.getStreamMaxVolume(AudioManager.STREAM_VOICE_CALL);</code>
2	<code>int vol = mMgr.getStreamVolume(AudioManager.STREAM_VOICE_CALL);</code>
3	<code>mBarVol.setMax(max);</code>
4	<code>mBarVol.setProgress(vol);</code>
5	<code>printText("语音呼叫音量: " + vol + "/" + max);</code>

3. 调整音量

通过音频管理器的 `adjustStreamVolume` 方法可以调整指定类型音频流的音量值。该方法有 3 个参数：第一个参数是音频流的类型；第二个参数是音量调节的方向（增或减）；第三个参数是控制标志。

表 10-11 是 Android 平台所定义的音量调节方向的类型，这些类型在音频管理器类中定义。

表 10-11 音量调节方向的类型

类 型	说 明
ADJUST_LOWER	调低
ADJUST_RAISE	调高
ADJUST_SAME	不调整（主要用于调用系统音量调整条的显示）

表 10-12 是 Android 平台所定义的音量调整标志的类型，这些类型在音频管理器类中定义。

表 10-12 音量调整标志的类型及说明

类 型	说 明
FLAG_ALLOW_RINGER_MODES	是否包含响铃模式选项
FLAG_PLAY_SOUND	当改变音量的时候是否播放声音
FLAG_REMOVE_SOUND_AND_VIBRATE	是否移除队列中的任何声音或振动
FLAG_SHOW_UI	是否显示音量调节滑动条
FLAG_VIBRATE	是否进入振动响铃模式

代码 10-18 是调整音乐音量的示例代码。

代码 10-18 调整音乐音量

文件名: AudioServiceAct.java

```
1 //获取音量调整方向
2 if(dir) {dir2 = AudioManager.ADJUST_RAISE;
3 } else { dir2 = AudioManager.ADJUST_LOWER; }
4 //调整音量
5 mMgr.adjustStreamVolume(AudioManager.STREAM_MUSIC, dir2, 0);
```

注意：在用户程序中调整音量将会对系统音量产生影响。例如，在用户程序调高音乐流的音量，则正在后台运行的音乐程序的音量也将提升。所以，如果要不影响系统设置，用户程序可能需要将调整前的音量值进行保存，在程序退出时进行还原设置。

10.3 视频回放与录制

10.3.1 视频回放（表面视图）

使用媒体播放器（MediaPlayer）不仅可以回放音频，而且还能回放视频。图 10-6 所示的是通过媒体播放器播放 3GP 视频的实机界面。

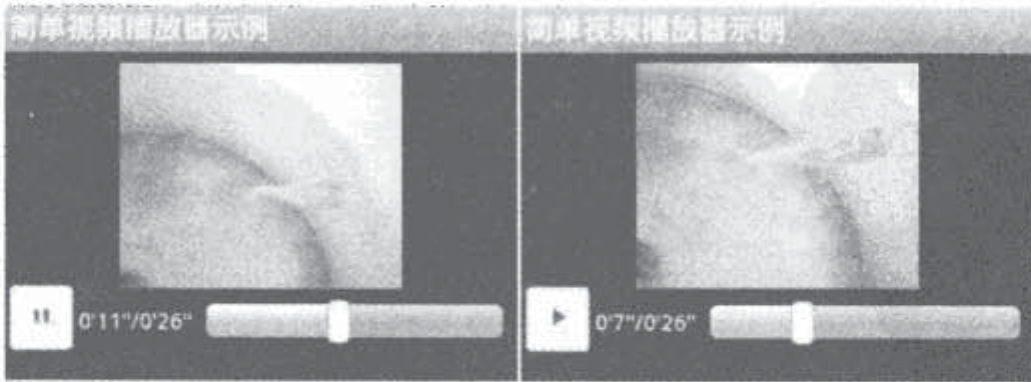


图 10-6 视频回放的实机界面

1. 应用程序主 Activity 框架

代码 10-19 是简单视频回放器示例程序的主 Activity 组件的框架定义，其与音乐播放器示例程序的框架结构基本相同，视频回放比音乐播放新增了视频帧渲染处理。



代码 10-19 视频回放器示例程序的主 Activity 组件的框架定义

文件名: FooViewPlayerAct.java

```

1  public class FooViewPlayerAct extends Activity implements OnSeekBarChangeListener,
2      OnPreparedListener, OnCompletionListener, OnClickListener, Callback {
3      private SurfaceView mRenderingView = null;
4      private SurfaceHolder mHolder = null;
5      private MediaPlayer mPlayer = null;
6      //播放状态控制
7      private ImageButton mBtnPlay = null;
8      private SeekBar mPBar = null;
9      private int mStatus = IConfig.STATUS_PAUSED;
10     private int mPlayPos = -1;
11     private Handler mHandler = null;
12     private Timer mPlayTimer = new Timer();
13
14     @Override
15     public void onCreate(Bundle savedInstanceState) {
16         super.onCreate(savedInstanceState);
17         setContentView(R.layout.main);
18         //获取按钮控件并设置单击事件侦听
19         this.mBtnPlay = (ImageButton)findViewById(R.id.btn_play);
20         this.mBtnPlay.setOnClickListener(this);
21         //获取渲染视频的视图
22         this.mRenderingView = (SurfaceView)findViewById(R.id.view);
23         this.mHolder = mRenderingView.getHolder();
24         this.mHolder.addCallback(this);
25         this.mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
26         //获取滑动条并设置滑动改变事件侦听器
27         mPBar = (SeekBar)findViewById(R.id.seekbar);
28         mPBar.setOnSeekBarChangeListener(this);
29         //初始化线程消息处理器
30         mHandler = new Handler() {
31             .....
32         };
33     }
34     @Override
35     public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {}
36     @Override
37     public void surfaceCreated(SurfaceHolder holder) { initPlayer(); }
38     @Override
39     public void surfaceDestroyed(SurfaceHolder holder) {}
40     .....
41 };

```

在代码 10-19 中，使用表面视图（SurfaceView）作为视频渲染的组件（第 22 行），在表

面视图控制器的回调方法中对播放器进行初始化（第 37 行）。

在该示例程序中，视频回放的主要顺序为渲染视图就绪→播放就绪→启动播放。所以，视频回放的思路为：在表面创建完毕的回调方法中初始化播放器；在播放器就绪的回调方法中打开播放开关。

2. 主界面布局定义

代码 10-20 是应用程序主界面的布局定义，其中主要包含表面视图、控制按钮和播放进度条等组件。

代码 10-20 视频回放器示例程序主界面布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <SurfaceView android:id="@+id/view"
6          android:layout_width="wrap_content" android:layout_height="wrap_content"/>
7      <TableLayout android:stretchColumns="2"
8          android:layout_height="wrap_content" android:layout_width="match_parent">
9          <TableRow>
10             <ImageButton android:id="@+id/btn_play" android:src="@drawable/play"
11                 android:layout_width="wrap_content" android:layout_height="wrap_content"/>
12             <TextView android:id="@+id/progress" android:layout_gravity="center_vertical"
13                 android:layout_width="wrap_content" android:layout_height="match_parent"
14                 android:gravity="center_vertical" android:text="0'00"/0'00""/>
15             <SeekBar android:id="@+id/seekbar" android:layout_gravity="center_vertical"
16                 android:layout_width="wrap_content" android:layout_height="wrap_content"
17                 android:paddingLeft="4pt" android:paddingRight="4pt"/>
18          </TableRow>
19      </TableLayout>
20 </LinearLayout>

```

3. 应用程序配置信息接口

代码 10-21 是应用程序中有关配置信息接口的定义，包括视频文件路径、播放状态类型等。

代码 10-21 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String PATH = "/sdcard/video/fish.3gp";
3      //播放状态
4      public static final int STATUS_PLAYING = 0;
5      public static final int STATUS_PAUSED = 1;
6      //扩展数据项名
7      public static final String EXTRA = "progress";
8  };

```




4. 回放视频

(1) 播放准备

代码 10-22 是初始化视频回放器的主要代码，其中主要包括创建播放器对象、设置数据源、设置显示控制器、设置音频输出类型、设置播放就绪和完成事件侦听器。

代码 10-22 初始化视频回放器

文件名: FooViewPlayerAct.java

```

1  private void initPlayer() { //初始化视频回放器
2      mPlayer = new MediaPlayer(); mPlayer.reset();
3      try { mPlayer.setDataSource(IConfig.PATH);
4          } catch (IllegalArgumentException e) { e.printStackTrace(); }
5          catch (IllegalStateException e) { e.printStackTrace(); }
6          catch (IOException e) { e.printStackTrace(); }
7      mPlayer.setDisplay(this.mHolder);
8      mPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
9      //设置音量
10     mPlayer.setVolume(80, 100);
11     //设置播放预备侦听器
12     mPlayer.setOnPreparedListener(this);
13     //设置播放完成侦听器
14     mPlayer.setOnCompletionListener(this);
15     preparePlay(mPlayer);
16     //更新视图大小
17     updateViewSize();
18
19     this.mBtnPlay.setEnabled(false);
20 }
21
22 private void preparePlay(MediaPlayer mp) { //播放预处理
23     try { mp.prepare(); } catch (IllegalStateException e) { e.printStackTrace(); }
24         catch (IOException e) { e.printStackTrace(); }
25 }
26
27 private void updateViewSize() { //更新视图大小
28     LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(
29         mPlayer.getVideoWidth(), mPlayer.getVideoHeight());
30     params.gravity = Gravity.CENTER;
31     params.topMargin = 4;
32     this.mRenderingView.setLayoutParams(params);
33 }
34
35 @Override //当播放已准备就绪后回调
36 public void onPrepared(MediaPlayer mp) { mBtnPlay.setEnabled(true); }
```

在代码 10-22 中，与音乐播放器不同的是，视频回放器的准备过程中需要设置显示控制器（第 7 行）并设置音频流类型（第 8 行）。只有当播放器就绪后才能开始播放（第 36 行）。表 10-13 是回放视频媒体的主要步骤及说明。

表 10-13 回放视频媒体的主要步骤及说明

顺 序	操 作	说 明
1	setOnCompletionListener() setOnPreparedListener()	设置播放事件侦听器
2	reset()	重置播放器
3	setDataSource()	设置媒体源
4	setDisplay()	设置渲染视频的表面
5	prepare()	播放预备
6	start()	启动播放
7	pause()	暂停播放
8	stop()	停止播放
9	release()	释放播放器资源

(2) 启动/暂停播放

代码 10-23 是通过播放控制按钮启动/暂停播放的主要代码。启动播放的时机是在播放器就绪之后。

代码 10-23 启动/暂停播放

文件名: FooViewPlayerAct.java

```
1  @Override
2  public void onClick(View v) {
3      if(this.mStatus == IConfig.STATUS_PAUSED) {
4          if(mPlayPos != -1) { mPlayer.seekTo(mPlayPos); }
5          this.mPlayer.start();
6          this.mStatus = IConfig.STATUS_PLAYING;
7          this.mBtnPlay.setImageResource(R.drawable.pause);
8          startTimer();
9      } else if(this.mStatus == IConfig.STATUS_PLAYING) {
10         mPlayer.pause();
11         this.mStatus = IConfig.STATUS_PAUSED;
12         this.mBtnPlay.setImageResource(R.drawable.play);
13         this.mPlayPos = mPlayer.getCurrentPosition();
14         this.mPlayTimer.cancel(); this.mPlayTimer.purge();
15     }
16 }
```

在代码 10-23 中，与音乐播放器示例程序一样，也使用播放进度计时器来显示播放进度（第 8 行），在此不予赘述。

10.3.2 视频回放（视频视图）

如果读者觉得使用“媒体播放器+表面视图”的模式回放视频比较复杂，那么选择视频视图是一个不错的主意。使用部件包中提供的视频视图组件（VideoView）可以简化对视频文件的播放。图 10-7 中就是使用视频视图播放 3GP 视频的实机界面。



图 10-7 使用视频视图回放视频的实机界面

1. 应用程序主 Activity 框架

代码 10-24 是视频视图应用示例程序的主 Activity 组件的框架定义。

代码 10-24 视频视图回放的主 Activity 组件的框架定义

文件名: FooVideoViewAct.java

```

1  public class FooVideoViewAct extends Activity implements OnPreparedListener,
2                                     OnCompletionListener {
3      //视频视图组件
4      private VideoView mVideoView = null;
5
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.main);
10         //获取视频视图
11         mVideoView = (VideoView)findViewById(R.id.video_view);
12         //初始化控制器
13         MediaController controller = new MediaController(this);
14         //设置视频视图与控制器的绑定
15         controller.setAnchorView(mVideoView);
16         mVideoView.setMediaController(controller);
17         //设置视频源
18         mVideoView.setVideoPath(IConfig.PATH);
19         mVideoView.setOnPreparedListener(this);
20         mVideoView.setOnCompletionListener(this);
21         //启动播放
22         mVideoView.start();
23     }
24     @Override
25     public void onPrepared(MediaPlayer mp) { updateViewSize(mp); }
26
27     private void updateViewSize(MediaPlayer mp) { //更新视图大小
28         LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(
29                                                     mp.getVideoWidth(), mp.getVideoHeight());
30         params.gravity = Gravity.CENTER;

```



```
31         params.topMargin = 4;
32         this.mVideoView.setLayoutParams(params);
33     }
34
35     @Override
36     public void onCompletion(MediaPlayer mp) {}
37     .....
38 };
```

在代码 10-24 中，首先根据资源标识获取视频视图组件（第 11 行），然后创建媒体控制器（第 13 行）并将视频视图与之进行互指：媒体控制器将视频视图设置为其绑定视图（第 15 行）；视频视图将该控制器设置为其媒体控制器（第 16 行）。两者分工相当明确：视频视图负责渲染视频，媒体控制器负责进行播放控制。

通过视频视图的 `setVideoPath` 方法设置将要播放的视频资源路径（第 18 行），即可启动播放（第 22 行），由此可见“视频视图+媒体控制器”的方式比较方便，开发者无需进行播放器的初始化及播放控制等。

2. 主界面布局定义

代码 10-25 是应用程序主界面布局定义，其中主要包含了视频视图组件。

代码 10-25 视频回放器示例程序主界面布局定义

文件名: main.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical" android:layout_width="320sp"
4      android:layout_height="fill_parent">
5      <VideoView android:id="@+id/video_view" android:layout_width="match_parent"
6          android:layout_height="match_parent" android:layout_gravity="center"/>
7  </LinearLayout>
```

3. 回放视频

作为视图组件，在 `Activity` 组件对布局资源填充完毕后，即可获取布局资源中所定义的视频视图组件实例，然后指定将要播放的视频资源定位信息（文件路径或网络 URI）即可启动播放行。表 10-14 是使用视频视图回放视频媒体的主要步骤及说明。

表 10-14 视频视图回放视频媒体的主要步骤及说明

顺 序	操 作	说 明
1	<code>setVideoPath()</code> <code>setVideoURI()</code>	设置视频路径或 URI
2	<code>setMediaController()</code>	设置媒体控制器
3	<code>start()</code>	启动播放
4	<code>pause()</code>	暂停播放
5	<code>stopPlayback()</code>	停止播放
6	<code>release()</code>	释放播放器资源



10.3.3 录制视频（代码控制）

对于手机而言，视频的录制往往通过相机设备进行视频采集。图 10-8 所示为摄像应用示例程序的实机界面，该程序使用手机相机设备录制视频，同时还可以对录制视频进行回放。

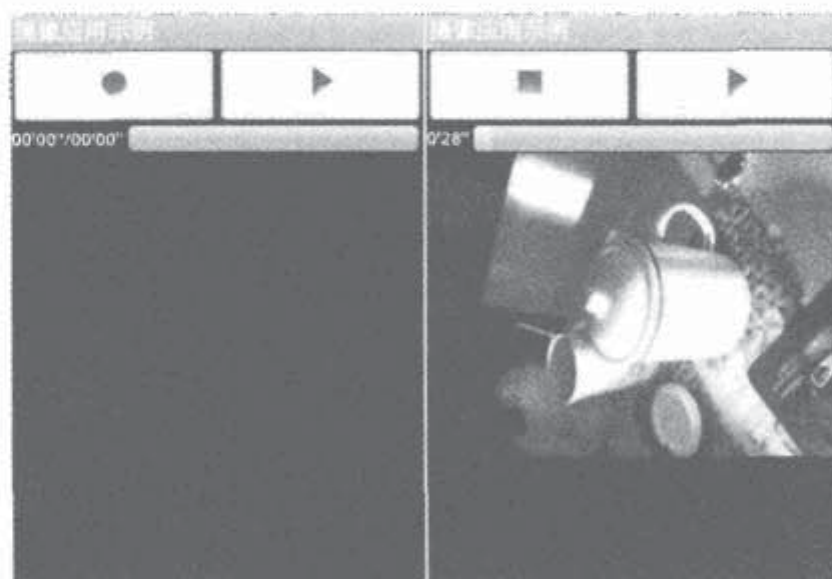


图 10-8 摄像应用示例程序的实机界面

1. 应用程序主 Activity 框架

代码 10-26 是该摄像应用示例程序的主 Activity 组件的框架定义，相比录音程序，摄像程序还需要使用相机硬件接口，同时还有用于渲染视频的视图。由于其中视频回放部分的实现与视频回放器示例程序相同，在此不予重复。

代码 10-26 摄像应用示例程序的主 Activity 组件的框架定义

文件名: CaptureDemoAct.java

```

1  public class CaptureDemoAct extends Activity implements OnClickListener,
2                                     OnCompletionListener, Callback {
3      //录音机
4      private MediaRecorder mRecorder = null;
5      //播放器
6      private MediaPlayer mPlayer = null;
7      //播放控制按钮
8      private ImageButton mBtnRec = null;
9      private ImageButton mBtnPlay = null;
10     //播放状态
11     private ProgressBar mPBar = null;
12     private Handler mHandler = null;
13     private int mStatus = IConfig.STATUS_PAUSED;
14     private Timer mPlayTimer = null;
15     private int mPlayPos = -1;
16     private int mCounter = 0;
17     //视频渲染视图（表面视图）和控制器

```



```

18     private SurfaceView mVideoView = null;
19     private SurfaceHolder mHolder = null;
20     //相机实例
21     private Camera mCamera = null;
22
23     @Override
24     public void onCreate(Bundle savedInstanceState) {
25         super.onCreate(savedInstanceState);
26         setContentView(R.layout.main);
27         //获取按钮控件并设置单击侦听
28         mBtnRec = (ImageButton)findViewById(R.id.btn_rec);
29         mBtnPlay = (ImageButton)findViewById(R.id.btn_play);
30         mBtnRec.setOnClickListener(this); mBtnPlay.setOnClickListener(this);
31         mBtnRec.setEnabled(false);
32         //获取表面视图及其控制器
33         this.mVideoView = (SurfaceView)findViewById(R.id.video_view);
34         this.mHolder = this.mVideoView.getHolder();
35         this.mHolder.addCallback(this);
36         this.mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
37         //获取播放进度条
38         mPBar = (ProgressBar)findViewById(R.id.pbar);
39         //初始化录音机
40         mRecorder = new MediaRecorder();
41         //初始化播放器
42         mPlayer = new MediaPlayer();
43         mPlayer.setDisplay(this.mHolder);
44         mPlayer.setOnCompletionListener(this);
45         //初始化线程消息处理器
46         mHandler = new Handler() {
47             .....
48         };
49     }
50     @Override
51     public void surfaceDestroyed(SurfaceHolder holder) {}
52     @Override
53     public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {}
54     @Override
55     public void surfaceCreated(SurfaceHolder holder) { mBtnRec.setEnabled(true); }
56     @Override
57     public void onClick(View v) {
58         switch(v.getId()) {
59             case R.id.btn_rec: { doCapture(); break; }
60             case R.id.btn_play: { doPlayback(); break; }
61         }
62     }
63     .....
64 };

```




Android 平台开发之旅 第2版

在代码 10-26 中，使用了一个表面视图来渲染视频（第 33 行），并在表面创建完毕的回调方法中打开摄像功能按钮（第 55 行）。

2. 主界面布局定义

代码 10-27 是摄像应用示例程序的主界面布局定义，其中主要包括录制/回放控制按钮（第 8 行和第 10 行）、录制/回放进度条（第 20 行）和用于渲染录制视频或回放视频的表面视图（第 26 行）。

代码 10-27 摄像应用示例程序的主界面布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <TableLayout android:stretchColumns="0,1"
6          android:layout_width="fill_parent" android:layout_height="wrap_content">
7          <TableRow>
8              <ImageButton android:src="@drawable/record" android:id="@+id/btn_rec"
9                  android:layout_width="wrap_content" android:layout_height="wrap_content"/>
10             <ImageButton android:src="@drawable/play" android:id="@+id/btn_play"
11                 android:layout_width="wrap_content" android:layout_height="wrap_content"/>
12             </TableRow>
13         </TableLayout>
14         <TableLayout android:stretchColumns="1"
15             android:layout_width="fill_parent" android:layout_height="wrap_content">
16             <TableRow>
17                 <TextView android:id="@+id/progress" android:layout_gravity="center_vertical"
18                     android:layout_width="wrap_content" android:layout_height="wrap_content"
19                     android:text="00'00"/00'00"/>
20                 <ProgressBar android:id="@+id/pbar"
21                     style="?android:attr/progressBarStyleHorizontal"
22                     android:layout_height="wrap_content" android:layout_width="match_parent"
23                     android:paddingLeft="2pt" android:paddingRight="2pt"/>
24                 </TableRow>
25             </TableLayout>
26             <SurfaceView android:id="@+id/video_view"
27                 android:layout_width="match_parent" android:layout_height="match_parent"/>
28         </TableLayout>

```

3. 应用程序配置信息接口

代码 10-28 是应用程序中有关配置信息接口的定义，包括录像文件路径、播放状态类型、最大录制时长和文件大小等。

代码 10-28 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {

```



```

2      public static final String REC_VIDEO_FILE = "/sdcard/video/capture.3gp";
3      //消息类型
4      public static final int RECORDING= 1;
5      public static final int PLAYBACK= 2;
6      //播放状态
7      public static final int STATUS_RECORDING = 1;
8      public static final int STATUS_PAUSED = 2;
9      public static final int STATUS_PLAYING = 3;
10     //扩展数据项名
11     public static final String EXTRA = "progress";
12     //最大录制时长和录制大小
13     public static final int MAX_DURATION = 10*60*1000; //10 分钟
14     public static final int MAX_SIZE = 10*1024*1024; //10MB
15 };

```

4. 录制视频

之前已经提到, 手机设备的视频录制需要使用相机设备。媒体录制类和相机类使用相机作为视频输入设备来录制视频。视频录制过程如图 10-9 所示。



图 10-9 视频录制过程示意图

(1) 视频录制过程

从图 10-9 可知, 视频录制的前提是获取相机接口并进行摄像内容预览。代码 10-29 是启动视频录制的主要代码, 其主要步骤为打开相机→启动预览→启动录制。

代码 10-29 启动视频录制

文件名: CaptureDemoAct.java

```

1      private void doCapture() {
2          if(this.mStatus == IConfig.STATUS_PAUSED) {
3              openCamera();
4              startPreview();
5              //录制准备
6              startRecord();
7              mCounter = 0;
8              startTimer1();
9              //更新状态
10             mBtnRec.setImageResource(R.drawable.stop);
11             mStatus = IConfig.STATUS_RECORDING;
12         } else if(this.mStatus == IConfig.STATUS_RECORDING) {
13             stopRecord();
14         }
15     }

```

(2) 打开相机

代码 10-30 是打开相机的主要代码。对于打开相机, 读者也可以理解为建立与相机服务



(硬件层)的连接。

代码 10-30 打开相机

文件名: CaptureDemoAct.java

```

1  private void openCamera() {
2      try { //确认相机处于关闭状态
3          closeCamera();
4          //打开相机
5          mCamera = Camera.open();
6          //获取参数, 设置参数
7          Camera.Parameters params = mCamera.getParameters();
8          params.setPreviewSize(320, 240);
9          //设置场景
10         params.setSceneMode(Camera.Parameters.SCENE_MODE_FIREWORKS);
11         //设置白平衡
12         params.setWhiteBalance(Camera.Parameters.WHITE_BALANCE_FLUORESCENT);
13         //设置预览帧率 (pfs)
14         //params.setPreviewFrameRate(25);
15         mCamera.setParameters(params);
16         mCamera.setPreviewDisplay(mHolder);
17     } catch (IOException e) { e.printStackTrace(); }
18 }
19
20 private void closeCamera() {
21     if (mCamera == null) { return; }
22     try { //使相机重新获取资源的控制
23         mCamera.reconnect();
24         //停止预览, 释放资源
25         mCamera.stopPreview();
26         mCamera.release();
27     } catch (IOException e) { e.printStackTrace(); }
28     catch (RuntimeException e) { e.printStackTrace(); }
29 }

```

在代码 10-30 中, 首先是获取相机实例 (第 5 行), 然后通过相机的参数设置器对相机的规格进行设置 (第 7~15 行), 此外还需指定相机的预览视图 (第 16 行)。

为了确保在打开之前相机处于关闭状态, 需要先关闭相机 (第 3 行)。

(3) 启动预览

代码 10-31 是打开相机预览功能的主要代码。

代码 10-31 打开相机预览功能

文件名: CaptureDemoAct.java

```

1  private void startPreview() {

```



```

2    //先使相机处于停止预览状态
3    mCamera.stopPreview();
4    //开始预览
5    mCamera.startPreview();
6    //相机释放资源，录像机才能使用相机进行视频采集
7    mCamera.unlock();
8 }

```

在代码 10-31 中，当相机启动预览后，开始录制视频前，需要对相机进行解锁（第 7 行的 unlock 方法），以便录像机对象能够访问并进行视频采集；反之，在对相机进行参数设置时，需要锁定访问（一般不使用 lock 方法而使用 reconnect 方法，代码 10-30 中第 23 行）。

（4）启动录制

代码 10-32 是启动视频录制的主要代码，其与音频录制的区别主要在于两点：需要设置相机和需要指定视频预览视图。

代码 10-32 启动视频录制

文件名：CaptureDemoAct.java

```

1    private void startRecord() {
2        try { //使相机先处于空闲状态
3            mRecorder.reset();
4            //指定用于录制的相机
5            mRecorder.setCamera(mCamera);
6            //设置视频源为相机
7            mRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
8            //设置音频源为 MIC
9            mRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
10           //设置输出文件的格式为 3GP
11           mRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
12           //mRecorder.setAudioChannels(2);
13           //最长录制时长
14           mRecorder.setMaxDuration(IConfig.MAX_DURATION);
15           //最大录制大小
16           mRecorder.setMaxFileSize(IConfig.MAX_SIZE);
17           //512KB/s
18           mRecorder.setVideoEncodingBitRate(512000);
19           mRecorder.setVideoFrameRate(15);
20           //mRecorder.setAudioSamplingRate(12);
21           //mRecorder.setAudioEncodingBitRate(1200);
22           //设置音频编码
23           mRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
24           //设置视频编码
25           mRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.H264);
26           //设置录制视频的分辨率
27           mRecorder.setVideoSize(320, 240);

```




```
28          //输出文件的路径和名称
29          mRecorder.setOutputFile(IConfig.REC_VIDEO_FILE);
30          //设置预览视图
31          mRecorder.setPreviewDisplay(mHolder.getSurface());
32          //准备, 开始, 视频录制
33          mRecorder.prepare();
34          mRecorder.start();
35      } catch (IllegalStateException e) { e.printStackTrace(); }
36      catch (IOException e) { e.printStackTrace(); }
37  }
```

在代码 10-32 中, 除了设置视频源为相机外 (第 7 行), 还必须设置相机 (第 5 行), 该方法可以实现在预览和摄像之间进行快速切换。为了进行视频预览, 需要为视频指定预览视图 (第 31 行)。表 10-15 是媒体录制接口所支持的视频源类型及说明。

表 10-15 媒体录制接口支持的视频源类型及说明

类 型	说 明
CAMERA	相机视频输入
DEFAULT	平台默认

表 10-16 是媒体录制接口所支持的视频编码方式。

表 10-16 媒体录制接口支持的编码方式

类/接口	说 明
DEFAULT	平台默认
H263	H.263 编码
H264	H.264 编码
MPEG_4_SP	MPEG4 编码

(5) 停止录制

代码 10-33 是停止视频录制的主要代码, 停止之前除了重置录像机之外, 还需要停止相机预览功能和释放相关资源 (需要重新打开相机进行再次录像)。

代码 10-33 停止视频录制

文件名: CaptureDemoAct.java

```
1  private void stopRecord() { //停止录制
2      //停止录音/回放计时器
3      if(this.mPlayTimer!=null) { mPlayTimer.cancel(); mPlayTimer.purge(); }
4      // 停止预览, 释放资源
5      mCamera.stopPreview(); mCamera.release();
6      // 设置 myRecorder 为空的状态, 为下次录制做准备
7      mRecorder.reset();
8      mPlayPos = -1;
9      //录音计时器
```



```
10      mCounter = 0;
11      //更新状态
12      mBtnRec.setImageResource(R.drawable.record);
13      mStatus = IConfig.STATUS_PAUSED;
14  }
```

5. 录像使用许可

录制视频需要具备相应的许可，即需要在应用程序清单文件中声明使用相机以及录制音频的使用许可及使用特性，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-feature android:name="android.hardware.camera"/>
<uses-feature android:name="android.hardware.camera.autofocus"/>
```

10.3.4 录制视频（调用系统功能）

对于上一节中的视频录制过程，可能会有读者觉得比较复杂，无论是相机的参数设置还是对预览视图的控制，都需要开发者进行控制，其定制性比较强。对于定制性要求不是很高的场合，可以通过捷径进行视频录制，即调用系统录制视频的功能，其实机界面如图 10-10 所示。

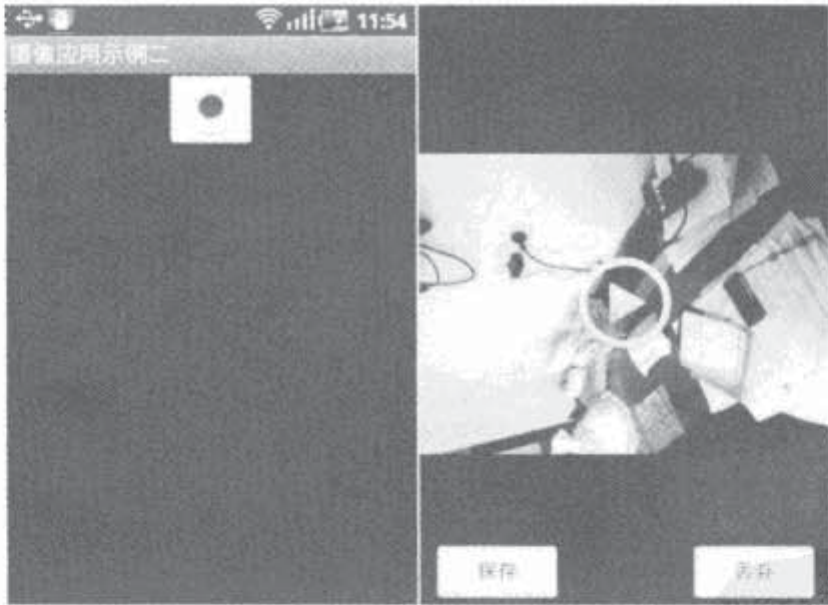


图 10-10 调用系统功能进行视频录制

1. 应用程序 Activity 框架

代码 10-34 是图 10-10 中示例程序的主 Activity 组件的框架定义，其界面只有一个按钮控件，用于调用系统摄像功能。

代码 10-34 应用示例二程序的主 Activity 组件的框架定义

文件名：CaptureDemo2Act.java

```
1  public class CaptureDemo2Act extends Activity implements OnClickListener {
2      @Override
```




```

3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main);
6          //获取按钮控件并设置单击侦听
7          ImageButton btnAction = (ImageButton)findViewById(R.id.btn_action);
8          btnAction.setOnClickListener(this);
9      }
10     @Override
11     public void onClick(View v) {
12         Intent i = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
13         //i.putExtra(MediaStore.EXTRA_OUTPUT,
14         //Uri.fromFile(new File(IConfig.REC_VIDEO_FILE)));
15         i.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1);
16         this.startActivityForResult(i, IConfig.REQ_CODE);
17     }
18     @Override
19     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
20         if((requestCode==IConfig.REQ_CODE)&&(resultCode==Activity.RESULT_OK)) {
21             if(data!=null) { sendNotification(data.getData()); }
22         }
23     }
24
25     void sendNotification(Uri uri) { //发送视频已经保存的通知
26         final String sn = Service.NOTIFICATION_SERVICE;
27         NotificationManager mgr = (NotificationManager)this.getSystemService(sn);
28         Notification n = new Notification(R.drawable.icon, "视频已保存!",
29             System.currentTimeMillis());
30         Intent i = new Intent(this, PlaybackAct.class);
31         i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
32         i.putExtra(IConfig.EXTRA, uri);
33         PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);
34         n.setLatestEventInfo(this, getApplicationContext(), "视频已保存!", "", pi);
35         //发送通知
36         mgr.notify(0, n);
37     }
38 };

```

在代码 10-34 中，使用按钮控件调用系统录制视频的功能（第 12~16 行），然后根据调用结果发出通知（第 20 行和第 21 行）。从代码 10-34 中可知，录制视频的 Activity 所返回的结果是所录制视频的内容 URI（第 25 行），还需要依据该 URI 在系统媒体库中进行查询才能获得详细的媒体信息。

注意：对于调用视频录制功能，存在两种形式：第一种是通过意向的扩展数据项指明录制的输出文件；第二种是不指明，由系统按规则生成输出文件。这两种方式的差异在于调用结果的获取：第一种方式无法在 Activity 调用结果的方法（onActivityResult）中获取所录制视频的内容 URI；第二种方式可以（第 21 行）。

2. 应用程序配置信息接口

代码 10-35 是应用程序中有关配置信息接口的定义, 包括 Activity 请求代码、扩展数据项名、视频媒体数据列 (该列内容为视频文件路径)。

代码 10-35 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String REC_VIDEO_FILE = "/sdcard/video/capture2.3gp";
3      public static final int REQ_CODE = 2012;
4      public static final String EXTRA = "uri";
5      //视频库信息数据列 (路径)
6      public static final String[] COLS = { MediaStore.Video.VideoColumns.DATA };
7  };

```

3. 获取录制结果

在代码 10-34 中, 示例程序以通知的形式告知用户录制结果。当用户查看该通知时, 将会触发指定的 Activity 来显示录制结果, 如图 10-11 所示。

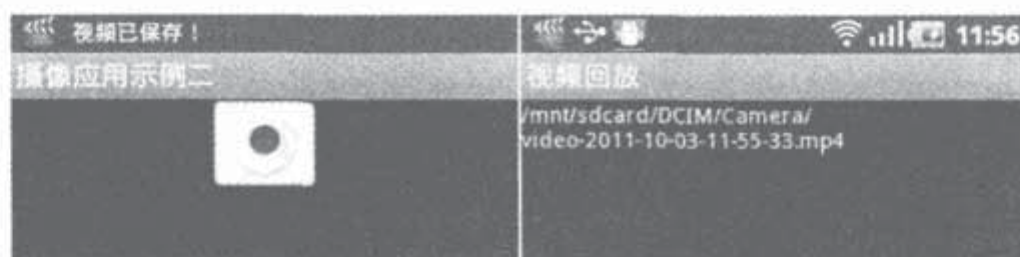


图 10-11 获取录制结果

代码 10-36 是通知所绑定的用于回放视频的 Activity 的定义, 实际上该 Activity 并没有进行回放操作, 而只是获取并显示所录制视频的路径信息。

代码 10-36 获取视频录制结果的 Activity 的定义

文件名: PlaybackAct.java

```

1  public class PlaybackAct extends Activity {
2      @Override
3      protected void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.viewer);
6          //获取主 Activity 传递过来的视频资源 URI 并回放视频
7          Uri uri = (Uri)(this getIntent().getExtras().get(IConfig.EXTRA));
8          Cursor c = this.getContentResolver().query(uri, IConfig.COLS, null, null, null);
9          c.moveToFirst();
10         String filepath = c.getString(0);
11         c.close();
12         //显示视频资源路径
13         ((TextView)findViewById(R.id.text)).setText(filepath);
14     }
15 };

```



通过代码 10-36 可知，主 Activity 所传递过来的内容实际上是所录制视频的内容 URI，而要获取视频文件的路径，需要按照内容 URI 进行查询，并获取包含视频文件路径的列内容（第 8 行）。

提示：由于该方式是调用视频录制功能，所以其无需录制视频相应的使用许可。

10.4 相机应用

在录像应用示例程序的介绍中已经提到过如何使用相机进行录像，而实际上，相机的主要功能是拍照。与录制视频的方式一样，相机拍照也存在两种方式：用户代码控制和调用系统功能。

10.4.1 拍摄照片（代码控制）

图 10-12 所示为相机应用示例程序的实机界面，拍照过程分为两个环节：拍摄预览和拍照。

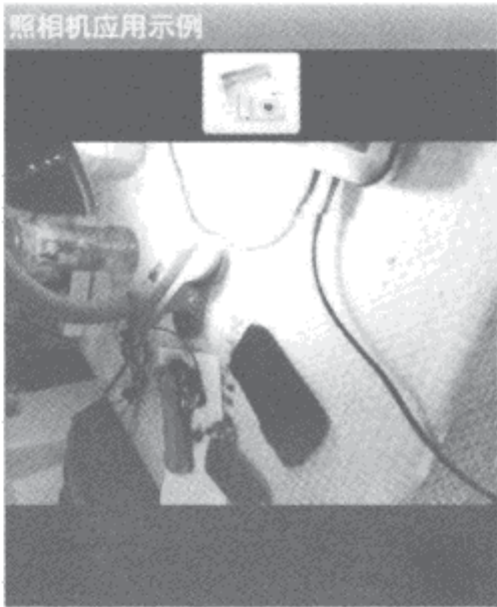


图 10-12 相机应用示例程序的实机界面

1. 应用程序 Activity 框架

代码 10-37 是该相机应用示例程序的主 Activity 组件的框架定义，相比录像程序，照相程序无需对相机解锁，通过照相方法（takePicture）就能进行照片捕获。

代码 10-37 应用示例程序的主 Activity 组件的框架定义

文件名：CameraDemoAct.java

```
1 public class CameraDemoAct extends Activity implements OnClickListener,
2                                     Callback, PictureCallback {
3     private SurfaceView mVideoView = null;
4     private SurfaceHolder mHolder = null;
5     private Camera mCamera = null;
6     //触发拍照的按钮
```



```

7      private ImageButton mBtnCapture = null;
8
9      @Override
10     public void onCreate(Bundle savedInstanceState) {
11         super.onCreate(savedInstanceState);
12         setContentView(R.layout.main);
13         //获取按钮控件并设置单击侦听
14         mBtnCapture = (ImageButton)findViewById(R.id.btn_capture);
15         mBtnCapture.setOnClickListener(this);
16         mBtnCapture.setEnabled(false);
17         //获取视频渲染视图
18         this.mVideoView = (SurfaceView)findViewById(R.id.video_view);
19         this.mHolder = this.mVideoView.getHolder();
20         this.mHolder.addCallback(this);
21         this.mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
22     }
23     @Override
24     public void surfaceDestroyed(SurfaceHolder holder) {}
25     @Override
26     public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {}
27     @Override
28     public void surfaceCreated(SurfaceHolder holder) {
29         //初始化相机
30         openCamera();      startPreview();  mBtnCapture.setEnabled(true);
31     }
32     .....
33     @Override
34     public void onClick(View v) {
35         if(mCamera!=null) { mCamera.takePicture(null, null, this); }
36     }
37     .....
38     @Override
39     protected void onDestroy() { super.onDestroy();
40         //停止预览，释放资源
41         mCamera.stopPreview();  mCamera.release();
42     }
43 };

```

在代码 10-37 中，使用了一个表面视图组件来进行照相预览（第 18 行），当表面视图创建完毕之后，打开相机并开始拍照预览（第 31 行）。在拍照按钮的单击事件回调方法中，使用相机实例的 `takePicture` 方法进行拍照（第 35 行）。

2. 主界面布局定义

代码 10-38 是照相应用示例程序的主界面布局定义，其中包含一个触发拍照的按钮（第 5 行）和一个用于照相预览的表面视图组件（第 8 行）。



代码 10-38 照相应用示例程序的主界面布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <ImageButton android:id="@+id/btn_capture" android:src="@drawable/picture"
6          android:layout_width="wrap_content" android:layout_height="wrap_content"
7          android:layout_gravity="center"/>
8      <SurfaceView android:id="@+id/video_view"
9          android:layout_width="match_parent" android:layout_height="match_parent"/>
10 </LinearLayout>

```

3. 应用程序配置信息接口

代码 10-39 是应用程序中有关配置信息接口的定义, 包括照片的存储路径。

代码 10-39 应用程序配置信息接口的定义

文件名: CameraDemoAct.java

```

1  public interface IConfig {
2      public static final String CAPTURE_PATH = "/sdcard/capture/";
3  };

```

4. 拍摄照片

使用相机拍照也需要先打开相机并启动预览, 相比录像程序, 两者初始化相机的差异主要在启动预览部分, 拍摄照片无需“插手”对相机的控制(代码 10-31 第 7 行)。代码 10-40 是启动拍照预览的主要代码。

代码 10-40 启动拍照预览

文件名: CameraDemoAct.java

```

1  private void startPreview() {
2      //先使相机处于停止预览状态
3      mCamera.stopPreview();
4      //开始预览
5      mCamera.startPreview();
6  }

```

提示: 由于相机程序与录像程序对相机的初始化处理相同(如代码 10-37 第 30 行), 所以在相机程序中对重复部分代码不再进行列举, 读者可以参考录像程序代码或随书代码。

启动拍照预览后, 即可拍摄照片(代码 10-37 第 35 行)。对于所拍摄的照片是使用回调的方式进行处理, 代码 10-37 中指定的回调接口为主 Activity 组件。代码 10-41 是获取并保存所拍照片的回调处理。

代码 10-41 保存拍摄照片

文件名: CameraDemoAct.java

```

1  @Override
2  public void onPictureTaken(byte[] data, Camera camera) {
3      Bitmap bmp = BitmapFactory.decodeByteArray(data, 0, data.length);
4      File f = new File(IConfig.CAPTURE_PATH+FooSysUtil.getInstance().getTsp2()+"jpg");
5      try {
6          BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(f));
7          bmp.compress(Bitmap.CompressFormat.JPEG, 80, bos);
8          bos.flush();
9          bos.close();
10         //拍完照之后会停止预览，需要恢复预览
11         startPreview();
12     } catch (Exception e) { e.printStackTrace(); }
13 }

```

在代码 10-41 中，所拍摄到的照片以原数据的形式存放在字节数组中，开发者可以将这些原数据压缩成指定格式的图片（第 7 行）。

5. 相机使用许可

与录像应用一样，使用相机需要具备相应的许可，即需要在应用程序清单文件中声明使用相机的使用许可和使用特性，其声明代码如下所示。

文件名: AndroidManifest.xml

```

<uses-permission android:name="android.permission.CAMERA"/>
<uses-feature android:name="android.hardware.camera"/>
<uses-feature android:name="android.hardware.camera.autofocus"/>

```

10.4.2 拍摄照片（调用系统功能）

与录制视频一样，拍摄照片也可以通过调用系统功能来完成。图 10-13 所示为调用系统功能进行拍照的实机界面。



图 10-13 调用系统功能进行拍照的实机界面



1. 应用程序 Activity 框架

代码 10-42 是该相机应用示例二程序的主 Activity 组件的框架定义，其界面只有一个按钮控件，用于调用系统拍照功能。

代码 10-42 应用示例二程序的主 Activity 组件的框架定义

文件名: CameraDemo2Act.java

```

1  public class CameraDemo2Act extends Activity implements OnClickListener {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main);
6          //获取按钮控件并设置单击侦听
7          ImageButton btnAction = (ImageButton)findViewById(R.id.btn_action);
8          btnAction.setOnClickListener(this);
9      }
10     @Override
11     public void onClick(View v) {
12         //String path = IConfig.CAPTURE_PATH+FooSysUtil.getInstance().getTsp2()+"jpg";
13         Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
14         //i.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(new File(path)));
15         this.startActivityForResult(i, IConfig.REQ_CODE);
16     }
17     @Override
18     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
19         if((requestCode==IConfig.REQ_CODE)&&(resultCode==Activity.RESULT_OK)) {
20             if(data!=null) { Bitmap bmp = (Bitmap)(data.getExtras().get("data"));
21                 sendNotification(bmp);
22             }
23         }
24     }
25
26     void sendNotification(Bitmap bmp) {
27         final String sn = Service.NOTIFICATION_SERVICE;
28         NotificationManager mgr = (NotificationManager)this.getSystemService(sn);
29         Notification n = new Notification(R.drawable.icon, "照片已保存!",
30             System.currentTimeMillis());
31         Intent i = new Intent(this, PhotoViewerAct.class);
32         i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
33         i.putExtra(IConfig.EXTRA, bmp);
34         PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);
35         n.setLatestEventInfo(this.getApplicationContext(), "照片已保存!", "", pi);
36         //发送通知
37         mgr.notify(0, n);
38     }
39 };

```


在代码 10-42 中，使用按钮控件调用系统拍照的功能（第 13~15 行），然后根据调用结果发出通知（第 21 行）。从代码 10-42 中可知，拍照的 Activity 所返回的结果是照片内容的位图对象（第 20 行）。

注意：对于调用系统拍照功能，也存在两种形式：第一种是通过意向的扩展数据项指明录制的输出文件；第二种是不指明，由系统按规则生成输出文件。这两种方式的差异在于调用结果的获取：第一种方式无法在 Activity 调用结果的方法（onActivityResult）中获取所拍照的内容；第二种方式却可以。

2. 应用程序配置信息接口

代码 10-43 是应用程序中有关配置信息接口的定义，包括 Activity 请求代码和扩展数据项名。

代码 10-43 应用程序配置信息接口的定义

文件名: IConfig.java

```
1 public interface IConfig {
2     public static final String CAPTURE_PATH = "/sdcard/capture/";
3     public static final int REQ_CODE = 2012;
4     public static final String EXTRA = "filepath";
5 };
```

3. 获取拍照结果

在代码 10-42 中，示例程序以通知的形式告知用户拍照结果。当用户查看该通知时，将会触发指定的 Activity 来显示拍照结果，如图 10-14 所示。



图 10-14 获取拍照结果

代码 10-44 是通知所绑定的用于显示照片的 Activity 组件的定义，该 Activity 通过图片视图来显示所传递的照片位图对象。

代码 10-44 获取拍照结果的 Activity 组件的定义

文件名: PhotoViewerAct.java

```
1 public class PhotoViewerAct extends Activity {
```




```
2      @Override
3      protected void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.viewer);
6          //获取主 Activity 传递过来的位图对象并显示
7          Intent i = this getIntent();
8          Bitmap bmp = (Bitmap)i.getExtras().get(IConfig.EXTRA));
9          ((ImageView)findViewById(R.id.iv)).setImageBitmap(bmp);
10     }
11 };
```

提示：由于该方式是调用系统拍照功能，所以其无需拍照相应的使用许可。

10.5 媒体信息管理



细心的用户会注意到，当更新安全数码（SD）卡中的内容时（下载或删除文件等），系统会启动对 SD 卡的扫描；当用户启动播放器，会获取手机中媒体文件的信息，而这些信息就是从系统媒体信息库中获取。图 10-15 所示的内容就是从系统媒体库获取的视频信息。

媒体库示例			
扫描	音频	视频	图片
行号	路径		时长
1	/mnt/sdcard/video/fish.3gp		26640
2	/mnt/sdcard/video/ocean-2.mp4		1252733
3	/mnt/sdcard/video/ocean_5.3gp		1219820

图 10-15 获取视频媒体信息的实机界面

10.5.1 Android 平台对媒体信息管理的支持

在 Android 平台中，媒体包（android.media）提供用于扫描媒体信息的功能。表 10-17 是媒体包中媒体扫描管理相关类/接口的说明。

表 10-17 媒体包中媒体扫描管理相关类/接口的说明

类/接口	说 明
MediaScannerConnection	媒体扫描连接器
MediaScannerConnection.OnScanCompletedListener	扫描完成事件侦听器

媒体扫描服务会将扫描结果存储到系统的媒体信息库中，通过媒体信息管理接口可以获取这些媒体信息。Android 平台在内容提供包（android.provider）中提供用于管理媒体信息的接口。表 10-18 是内容提供包中媒体信息管理相关的类/接口的说明。

表 10-18 内容提供包中媒体信息管理相关的类/接口的说明

类/接口	说 明
MediaStore	媒体库信息接口
MediaStore.Audio	音频媒体信息接口
MediaStore.Images	图片媒体信息接口
MediaStore.Video	视频媒体信息接口

10.5.2 应用程序 Activity 框架

代码 10-45 是该媒体库应用示例程序的主 Activity 组件的框架定义，对于媒体库信息获取采用的是内容提供机制，需要使用 Activity 组件关联的内容解析器。

代码 10-45 应用示例程序的主 Activity 组件的框架定义

文件名: MediaStoreAct.java

```
1 public class MediaStoreAct extends Activity implements OnClickListener,
2                                     OnScanCompletedListener {
3     //内容解析器
4     private ContentResolver mResolver = null;
5     private LinearLayout mLayout = null;
6
7     @Override
8     public void onCreate(Bundle savedInstanceState) {
9         super.onCreate(savedInstanceState);
10        setContentView(R.layout.main);
11        //获取按钮控件并设置单击事件侦听器
12        Button btnScanner = (Button)findViewById(R.id.btn_scanner);
13        Button btnAudio = (Button)findViewById(R.id.btn_audio);
14        Button btnVideo = (Button)findViewById(R.id.btn_video);
15        Button btnImage = (Button)findViewById(R.id.btn_image);
16        btnScanner.setOnClickListener(this);
17        btnAudio.setOnClickListener(this);
18        btnVideo.setOnClickListener(this);
19        btnImage.setOnClickListener(this);
20        //获取布局容器对象
21        this.mLayout = (LinearLayout) findViewById(R.id.layoutPanel);
22        this.mLayout.setOrientation(LinearLayout.VERTICAL);
23        //获取内容解析器
24        this.mResolver = this.getContentResolver();
25    }
26    @Override
27    public void onClick(View v) {
28        switch(v.getId()) { case R.id.btn_scanner: { scanMediaFiles(); break; }
29                           case R.id.btn_audio: { showAudioStore(); break; }
30                           case R.id.btn_video: { showVideoStore(); break; }
```




```

31                                     case R.id.btn_image: { showImageStore(); break; }
32                                     }
33                                     }
34                                     .....
35     };

```

在代码 10-45 中，在初始化代码中获取关联的内容解析器（第 24 行），使用按钮控件触发扫描媒体文件和显示音频、视频以及图片媒体信息（第 28~31 行）。

10.5.3 应用程序配置信息接口

代码 10-46 是应用程序中有关配置信息接口的定义，包括所扫描的路径、目标媒体的 MIME 类型以及各种媒体的信息列。

代码 10-46 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String[] PATHs= { "/sdcard/" };
3      public static final String[] MIMEs= { "audio/mpeg", "image/png", "video/mp4" };
4
5      public static final String[] COLS_AUDIO = { //音频媒体信息列
6          AudioColumns._ID, //行号
7          ImageColumns.DATA, //路径
8          AudioColumns.DURATION //播放时长
9      };
10
11     public static final String[] COLS_VIDEO = { //视频媒体信息列
12         VideoColumns._ID, //行号
13         ImageColumns.DATA, //路径
14         VideoColumns.DURATION //播放时长
15     };
16
17     public static final String[] COLS_IMAGE = { //图片媒体信息列
18         ImageColumns._ID, //行号
19         ImageColumns.DATA, //路径
20         ImageColumns.SIZE //图片大小
21     };
22 };

```

10.5.4 扫描媒体文件

使用媒体扫描连接器（MediaScannerConnection）可以对指定文件中的指定类型的媒体文件进行扫描。按照其扫描机制，将扫描操作分为无连接扫描和连接扫描。

1. 无连接扫描

无连接扫描就是客户端程序不与系统媒体扫描服务进行连接而直接执行扫描动作。

图 10-16 是进行媒体扫描的示例程序界面，当单击“扫描”按钮后，程序会调用系统的扫描

服务对指定的路径中的媒体进行扫描，扫描完毕会以通知的形式报告扫描结果。



图 10-16 进行媒体扫描的示例程序界面

(1) 启动媒体扫描

代码 10-47 是启动媒体扫描的主要代码。

代码 10-47 启动媒体扫描

文件名: MediaStoreAct.java

```
1 private void scanMediaFiles() {
2     MediaScannerConnection.scanFile(this, IConfig.PATHs, IConfig.MIMEs, this);
3 }
```

在代码 10-47 中，通过媒体扫描连接器的 scanFile 方法可以启动对指定路径和类型的媒体信息进行扫描，并指定扫描完成事件的侦听器为 Activity。Activity 必须实现该侦听接口以获取扫描状态（代码 10-45 第 2 行）。

(2) 获取扫描状态

代码 10-48 是通过实现扫描完成事件侦听器的接口获取扫描状态的主要代码。

代码 10-48 获取扫描状态

文件名: MediaStoreAct.java

```
1 @Override
2 public void onScanCompleted(String path, Uri uri) {
3     final String sn = Service.NOTIFICATION_SERVICE;
4     NotificationManager mgr = (NotificationManager)this.getSystemService(sn);
5     Notification n = new Notification(R.drawable.icon,
6         "【"+path+"】已扫描完毕!", System.currentTimeMillis());
7     PendingIntent pi = PendingIntent.getActivity(this, 0, null, 0);
8     n.setLatestEventInfo(this, "【"+path+"】已扫描完毕!", "", pi);
9     //发送通知
10    mgr.notify(0, n);
11 }
```



在代码 10-48 中,通过回调方法可以获取完成扫描的路径以及扫描结果的 URI,使用内容解析器即可查询 URI 所指向的结果集。

提示: 无连接扫描也需要与系统扫描服务进行连接,只是无法进行连接的管理。扫描动作需要在与扫描服务建立连接后进行,由于无法获知连接状态,扫描动作可能会受到不确定因素的影响(不能保证建立正常的连接)。

2. 连接扫描

连接扫描就是客户端与系统扫描服务之间建立连接,并通过连接来请求扫描动作并获取扫描状态和结果。图 10-17 所示就是通过连接扫描所获取的扫描结果。



图 10-17 获取的扫描结果

(1) 应用程序 Activity 框架

代码 10-49 是该媒体扫描器示例程序的主 Activity 组件的框架定义,其界面只有一个文本框来显示扫描状态和结果。

代码 10-49 媒体扫描器示例程序的主 Activity 组件的框架定义

文件名: ScannerAct.java

```

1  public class ScannerAct extends Activity implements MediaScannerConnectionClient {
2      //媒体扫描器连接
3      private MediaScannerConnection mConnection = null;
4      private Handler mHandler = null;
5
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.main);
10         //初始化主线程消息队列处理器
11         this.mHandler = new Handler() {
12             @Override
13             public void handleMessage(Message msg) {
14                 Bundle data = msg.getData();
15                 printText(data.getString(IConfig.EXTRA));
16             }
17         }
18     }
19 }
```



```

17         };
18         //初始化连接
19         initConnection();
20     }
21
22     private void initConnection() {
23         mConnection = new MediaScannerConnection(this, this);
24         mConnection.connect();
25     }
26
27     @Override
28     protected void onDestroy() { super.onDestroy();
29         mConnection.disconnect();
30     }
31     @Override
32     public void onMediaScannerConnected() {
33         for(int i = 0; i < IConfig.MIMEs.length; ++i) {
34             mConnection.scanFile(IConfig.PATHs[0], IConfig.MIMEs[i]);
35         }
36     }
37     @Override
38     public void onScanCompleted(String path, Uri uri) {
39         Message msg = new Message();
40         Bundle data = new Bundle();
41         data.putString("msg", "【"+path+"】扫描完毕! \n 结果 URI: "+uri);
42         msg.setData(data);
43         msg.setTarget(mHandler);
44         msg.sendToTarget();
45     }
46     .....
47 };

```

在代码 10-49 中，当 Activity 创建时建立与扫描服务的连接，当程序销毁时关闭该连接。在连接建立成功后客户端发起扫描请求（第 34 行），在扫描完成的回调方法中使用消息机制将扫描结果通过主线程消息队列处理器发送给主线程（第 44 行）；主线程获取该消息（扫描结果）并进行显示（第 15 行）。

（2）应用程序配置信息接口

代码 10-50 是应用程序中有关配置信息接口的定义，包括扫描路径、媒体类型和扩展数据项名。

代码 10-50 应用程序配置信息接口的定义

文件名: IConfig.java

```

1 public interface IConfig {
2     public static final String[] PATHs= { "/sdcard/" };
3     public static final String[] MIMEs= { "audio/mpeg", "image/png", "video/mp4" };

```



```
4      public static final String EXTRA = "msg";
5  };
```

10.5.5 获取媒体文件信息

媒体文件包括音频、视频和图片，不同类型的媒体其信息有所不同。

1. 音频媒体

图 10-18 所示为获取音频媒体信息的实机界面。

媒体库示例			
扫描	音频	视频	图片
行号	路径	时长	
3	/mnt/sdcard/Sounds/top100/002武装.mp3	226188	
6	/mnt/sdcard/Sounds/top100/005心为你而碎.mp3	246877	
7	/mnt/sdcard/Sounds/top100/006把幸福给你.mp3	232881	

图 10-18 获取音频媒体信息的实机界面

代码 10-51 是获取音频媒体信息的主要代码。

代码 10-51 获取音频媒体信息

文件名: MediaStoreAct.java

```
1  private void showAudioStore() {
2      Cursor c = mResolver.query(MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,
3                                IConfig.COLS_AUDIO, null, null, null);
4      //使用表格视图组件显示记录游标的遍历内容
5      final double colWidths[] = { 0.15f, 0.65f, 0.2f};
6      FooTableWrapper wrapper = new FooTableWrapper(this, c, colWidths);
7      initColumnTitleMapping1(wrapper);
8      //设置消息处理器（处理器由调用方创建）
9      //wrapper.setHandler(this.mHandler);
10     this.mLayout.addView(wrapper.getMainWidget());
11     //关闭游标
12     c.close();
13 }
```

在代码 10-51，使用内容解析器的 query 方法，依照音频媒体信息接口的内容 URI 即可获取系统媒体库中的有关音频媒体信息（第 2 行），继而对游标进行遍历，获取记录内容并显示在可视组件中（第 5~10 行）。

2. 视频媒体

图 10-15 所示为示例程序获取系统媒体库中视频媒体的实机界面。代码 10-52 是获取视频媒体信息的主要代码。

代码 10-52 获取视频媒体信息

文件名: MediaStoreAct.java

```
1 private void showVideoStore() {
2     Cursor c = mResolver.query(MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
3                               IConfig.COLS_VIDEO, null, null, null);
4     ..... //与音频媒体处理相同
5 }
```

在代码 10-52 中，与获取音频媒体信息所不同的仅仅是内容 URI 和信息列，其内容读取和显示与音频部分相同。

3. 图片媒体

图 10-19 所示为获取图片媒体信息的实机界面。

媒体库示例				
扫描		音频	视频	图片
行号	路径			大小
5	/mnt/sdcard/DCIM/贝贝/IMG_0030-d.JPG			510419
6	/mnt/sdcard/DCIM/贝贝/IMG_0001-d.JPG			796511
7	/mnt/sdcard/DCIM/贝贝/IMG_0009-d.JPG			744572

图 10-19 获取图片媒体信息的实机界面

代码 10-53 是获取图片媒体信息的主要代码。

代码 10-53 获取图片媒体信息

文件名: MediaStoreAct.java

```
1 private void showImageStore() {
2     Cursor c = mResolver.query(MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
3                               IConfig.COLS_IMAGE, null, null, null);
4     ..... //与音频媒体处理相同
5 }
```

在代码 10-53 中，与获取音频媒体信息所不同的仅仅是内容 URI 和信息列，其内容读取和显示与音频部分相同。

10.6 音乐盒工具

相信喜欢音乐的读者对音乐盒工具并不陌生，其通过网页显示组件，将浏览器/服务器模式（B/S）应用进行集成。当应用程序启动时，网页显示组件会使用 HTTP 方式访问其资源服务器，获取最新的资源信息并以网页的形式展示。当用户单击网页上的资源链接时，音乐盒工具会获取该链接信息并使用网络连接的方式获取媒体流实现播放。

同样，当进行媒体查询时，音乐盒工具向服务器发送请求后，服务器将查询结果以网页的形式返回给网页显示组件。图 10-20 所示为音乐盒示例程序的实机界面。



图 10-20 音乐盒示例程序的实机界面

图 10-21 所示为该音乐盒工具的功能示意图，通过载入服务端网页和扫描本地存储器来提供音乐或视频的资源列表。当用户选择网页链接时，程序会通过网页显示组件的后台获取该链接的资源信息，提供给媒体播放器进行播放；如果用户选择本地媒体列表时，媒体播放器会根据其路径直接播放。

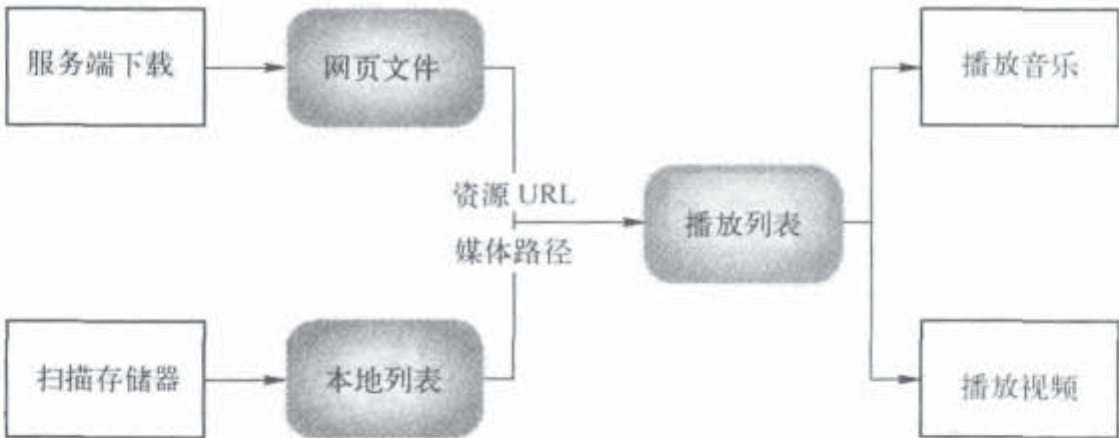


图 10-21 音乐盒工具的功能示意图

1. 应用程序 Activity 框架

代码 10-54 是该音乐盒示例程序的主 Activity 组件的框架定义，其包含选项页控件（界面主组件）、网页视图和客户端（集成网页内容）、播放列表视图（集中管理播放列表）、本地列表（管理本地媒体列表）、视频视图（渲染视频）和播放器对象（播放媒体用）。

代码 10-54 音乐盒示例程序的主 Activity 组件的框架定义

文件名: MusicBoxAct.java

```
1 public class MusicBoxAct extends TabActivity implements OnItemClickListener,
2                                     OnPreparedListener, OnCompletionListener {
```



```

3      private TabHost mTabHost = null;
4      //网页视图
5      private WebView mWebView = null;
6      //播放列表视图
7      private ListView mPlayListView = null;
8      private ArrayList<String> mItems = new ArrayList<String>();
9      private ArrayList<String> mPaths = new ArrayList<String>();
10     //本地列表视图
11     private ListView mLocalListView = null;
12     //视频视图
13     private VideoView mVideoView = null;
14     //音乐播放器
15     private MediaPlayer mPlayer = null;
16
17     @Override
18     public void onCreate(Bundle savedInstanceState) {
19         super.onCreate(savedInstanceState);
20         setContentView(R.layout.main);
21         //布局定义组件（视频视图、列表视图和网页视图）
22         mVideoView = (VideoView)findViewById(R.id.video);
23         mPlayListView = (ListView)findViewById(R.id.play_list);
24         mWebView = (WebView)findViewById(R.id.web);
25         mLocalListView = (ListView)findViewById(R.id.local_list);
26         //初始化 TabHost
27         mTabHost = getTabHost();
28         mTabHost.addTab(mTabHost.newTabSpec("t1")
29             .setIndicator("正在播放").setContent(R.id.video));
30         mTabHost.addTab(mTabHost.newTabSpec("t2")
31             .setIndicator("播放列表").setContent(R.id.play_list));
32         mTabHost.addTab(mTabHost.newTabSpec("t3")
33             .setIndicator("推荐资源").setContent(R.id.web));
34         mTabHost.addTab(mTabHost.newTabSpec("t4")
35             .setIndicator("本地资源").setContent(R.id.local_list));
36         //设置当前选项卡
37         mTabHost.setCurrentTab(2);
38         //初始化各个 tab
39         initPlayList();
40         preparePlay();
41         initWebView();
42         initLocalList();
43     }
44     .....
45 };

```

在代码 10-54 中，使用选项页控件作为主界面组件，该组件包含 4 个选项页。

1) 正在播放页（第 28 行），用于显示当前播放的视频，本页的主要组件是视频视图。

**Android 平台开发之旅 第2版**

2) 播放类表页 (第 30 行), 用于集中管理音乐播放列表, 本页的主要组件是列表视图。

3) 推荐资源页 (第 32 行), 模拟显示从服务端下载的网页, 本页的主要组件是网页视图。

4) 本地资源页 (第 34 行), 用于显示本地媒体列表, 本页的主要组件是列表视图。

2. 主界面布局定义

代码 10-55 是图 10-20 所示的界面布局的定义, 其使用选项页控件作为根组件, 其中包含 4 个选项页。

代码 10-55 主界面布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <TabHost xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@android:id/tabhost"
4      android:layout_width="fill_parent" android:layout_height="fill_parent" >
5      <LinearLayout android:orientation="vertical"
6          android:layout_width="fill_parent" android:layout_height="fill_parent">
7          <TabWidget android:id="@android:id/tabs"
8              android:layout_width="fill_parent" android:layout_height="wrap_content" />
9          <FrameLayout android:id="@android:id/tabcontent"
10             android:layout_width="fill_parent" android:layout_height="fill_parent">
11             <LinearLayout android:orientation="vertical"
12                 android:layout_width="fill_parent" android:layout_height="fill_parent" >
13                 <VideoView android:id="@+id/video"
14                     android:layout_width="fill_parent" android:layout_height="fill_parent"/>
15             </LinearLayout>
16             <ListView android:id="@+id/play_list"
17                 android:layout_width="fill_parent" android:layout_height="fill_parent" />
18             <WebView android:id="@+id/web"
19                 android:layout_width="fill_parent" android:layout_height="fill_parent" />
20             <ListView android:id="@+id/local_list"
21                 android:layout_width="fill_parent" android:layout_height="fill_parent" />
22             </FrameLayout>
23         </LinearLayout>
24     </TabHost>

```

3. 应用程序配置信息接口

代码 10-56 是应用程序中有关配置信息接口的定义, 包括主页 URI 和本地媒体路径。

代码 10-56 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String HOME_URI = "file:///sdcard/files/index.html";
3      public static final String PATH = "/sdcard/audio/";
4  };

```


4. 初始化过程

在代码 10-54 中, 初始化内容包括播放列表、播放器、网页视图和本地媒体列表。

(1) 初始化播放列表

代码 10-57 是初始化播放列表的主要代码, 其中主要是为播放器列表视图设置适配器以及列表项单击事件侦听器, 列表视图通过适配器与数据容器进行关联。

代码 10-57 初始化播放列表

文件名: MusicBoxAct.java

```
1 private void initPlayList() { //初始化播放列表
2     ListAdapter adapter = new ArrayAdapter<String>(this,
3                                     android.R.layout.simple_list_item_1, mItems);
4     mPlayListView.setAdapter(adapter);
5     mPlayListView.setOnItemClickListener(this);
6 }
```

(2) 初始化播放器

代码 10-58 是初始化播放器的主要代码, 其中主要是设置播放完成侦听器和媒体控制器。

代码 10-58 初始化播放器

文件名: MusicBoxAct.java

```
1 private void preparePlay() { //初始化播放器
2     mPlayer = new MediaPlayer();
3     mPlayer.setOnCompletionListener(this);
4     //设置媒体控制器与视频视图的绑定
5     MediaController controller = new MediaController(this);
6     controller.setAnchorView(this.mVideoView);
7     this.mVideoView.setMediaController(controller);
8 }
9
10 @Override
11 public void onCompletion(MediaPlayer mp) { //当播放结束后播放后一首
12     int pos = this.mPlayListView.getSelectedItemsPosition();
13     if(pos == ListView.INVALID_POSITION) { return; }
14     int newPos = (pos+1)%this.mPlayListView.getCount();
15     this.changeUrl(this.mPaths.get(newPos));
16 }
```

在代码 10-58 中, 当前播放结束后, 会继续播放列表中的后一首内容 (第 15 行)。

(3) 网页视图

代码 10-59 是初始化网页视图的主要代码, 其中主要是设置网页视图客户端, 而网页视图客户端主要用于获取所单击的网页中的链接信息。

代码 10-59 初始化网页视图

文件名: MusicBoxAct.java

```
1 private void initWebView() { //初始化网页视图
```




```
2      mWebView.getSettings().setJavaScriptEnabled(true);
3      mWebView.setWebViewClient(new WebViewClient() {
4          @Override
5          public boolean shouldOverrideUrlLoading(WebView view, String url) {
6              MusicBoxAct.this.changeUrl(url);
7              return (true);
8          }
9      });
10     //加载主页
11     mWebView.loadUrl(IConfig.HOME_URI);
12 }
```

在代码 10-59 中，网页视图所设置的客户端重载了加载方法，将网页视图所传过来的资源 URL “转交” 给 Activity 组件进行处理（第 6 行）。初始化时网页视图会加载主页内容（第 11 行），其实机界面如图 10-22a 所示。

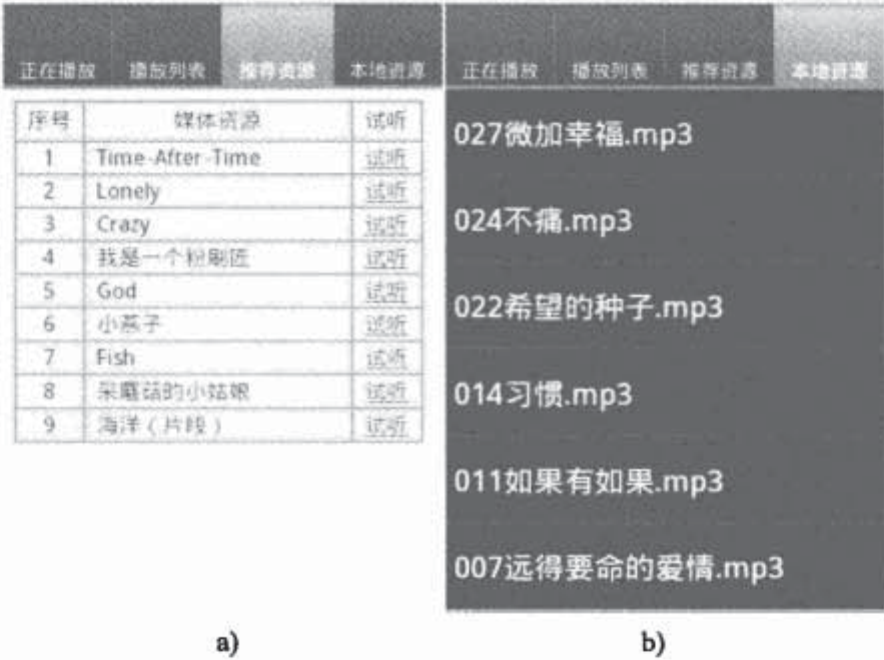


图 10-22 网页视图和本地媒体列表的实机界面

(4) 本地媒体列表

代码 10-60 是初始化本地媒体列表的主要代码，其中主要是遍历指定路径中的媒体文件，并将列表作为数据集提供给列表视图显示。

代码 10-60 初始化本地媒体列表

文件名: MusicBoxAct.java

```
1  private void initLocalList() {
2      //音乐文件路径列表
3      final String[] mFileNames = FooFileUtil.getInstance().list(IConfig.PATH, ".mp3");
4      ListAdapter adapter = new ArrayAdapter<String>(this,
5          android.R.layout.simple_list_item_1, mFileNames);
6      mLocalListView.setAdapter(adapter);
7      mLocalListView.setOnItemClickListener(this);
8  }
```


在代码 10-60 中, 首先会获取指定路径中的 MP3 媒体文件列表, 并与列表视图组件进行绑定, 其初始化界面如图 10-22b 所示。

5. 播放媒体资源

当用户单击网页中的链接时, 网页视图客户端会捕捉该链接信息, 并转交给 Activity 组件进行处理 (代码 10-59 第 7 行)。代码 10-61 是播放媒体资源的主要代码。

代码 10-61 播放媒体资源

文件名: MusicBoxAct.java

```

1  public void changeUrl(String url) { //当资源 URL 改变时 (回调函数)
2      //初始化播放
3      if(mVideoView.isPlaying() ) { mVideoView.stopPlayback(); }
4      if(mPlayer.isPlaying() ) { mPlayer.stop(); }
5
6      if(url.endsWith(".mp4") || url.endsWith(".3gp") ) { //视频资源
7          playVideo(url); mTabHost.setCurrentTab(0); //跳转到播放页
8      } else if(url.endsWith(".mp3") || url.endsWith(".wma") ) { //音频资源
9          playAudio(url); mTabHost.setCurrentTab(1); //跳转到播放列表页
10     }
11 }
12
13 //播放音频
14 @SuppressWarnings("unchecked")
15 private void playAudio(String url) {
16     //URL 检查, 去掉本地资源 URL 的模式("file://")
17     String url2 = url.replaceFirst("file://", "");
18
19     //更新数据绑定
20     if(mItems.size() > 0) {
21         //列表项去重
22         int index = mPaths.indexOf(url2);
23         if(index != -1) { mPaths.remove(index); mItems.remove(index); }
24         mPaths.add(0, url2); mItems.add(0, FooFileUtil.getInstance().getFilename(url2));
25     } else { mPaths.add(url2); mItems.add(FooFileUtil.getInstance().getFilename(url2)); }
26     //通知播放列表适配器的数据集改变
27     ((ArrayAdapter<String>)mPlayListView.getAdapter()).notifyDataSetChanged();
28     mPlayListView.setSelected(true); mPlayListView.setSelection(0);
29
30     //播放音频
31     try { mPlayer.reset();
32         mPlayer.setDataSource(url2);
33         mPlayer.prepare();
34         mPlayer.start();
35     } catch(IOException e) { e.printStackTrace(); }
36 }
37

```




```
38 private void playVideo(String url) { //播放视频
39     mVideoView.setVideoPath(url); //设置视频路径
40     mVideoView.setOnPreparedListener(this);
41     mVideoView.start();
42 }
```

在代码 10-61 中，根据资源 URL 来判断该资源是视频还是音频（第 6 行或第 8 行），进而调用不同的播放方式。对于视频媒体，将直接使用视频视图进行播放（第 39~41 行）；对于音频媒体，将使用媒体播放器对象进行播放（第 31~34 行），且在播放之前还需要将当前播放内容添加到播放列表中（第 19~24 行）。

对于播放列表的添加，需要注意两点：第一是列表项去重（第 22 行）；第二是资源路径与显示名称的位置同步（第 23 或第 24 行）。将资源信息拆分成显示名称和路径是为了显示的简洁（因为路径信息往往很长）：显示名称只用于显示，路径信息用于后台播放。

6. 播放列表管理

在音乐盒示例程序中，播放列表的调整有 3 种情形：前两种是在网页视图和在本地媒体列表选取音频媒体资源（添加调整）。对于添加的项目，播放列表首先会进行重复性判断，然后将该播放项置顶（代码 10-61 中第 23 行的“0”即首位），如图 10-23 所示。



图 10-23 最新播放项放置在列表首位

还有一种调整是在播放列表中点选列表项时，将该项置顶。代码 10-62 是列表视图中列表项单击事件的回调方法。

代码 10-62 添加播放列表

文件名: MusicBoxAct.java

```
1 @Override
2 public void onItemClick(AdapterView<?> parent, View v, int pos, long id) {
3     switch(parent.getId()) {
4         case R.id.play_list: { changeUrl(mPaths.get(pos)); break; }
5         case R.id.local_list: {
```



```
6          final String name = mLocalListView.getAdapter().getItem(pos).toString();
7          changeUrl(IConfig.PATH+name); break;
8      }
9  }
10 }
```

当播放列表改变后，需要通过适配器通知列表视图更新绘制（代码 10-61 中第 27 行）。

7. 应用许可

由于音乐盒工具需要通过互联网访问网页，所以需要在应用程序清单文件中声明访问互联网的使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.INTERNET"/>
```

第 11 章 个人信息管理

本章介绍 Android 支持的个人信息管理内容，通过实际的开发案例介绍如何获取联系人信息、电话号码、公司信息等与个人信息有关的内容。

11.1 个人信息管理

对于个人而言，手机存在一定的私密性。因为在手机平台，存在很多重要的个人信息，如联系人、电话号码、日程安排等。作为标准而言，个人信息管理（Personal Information Manager，PIM）中的信息一般包括联系人（Contact）、日程（Calendar）、任务（Task）、便签（Note）以及电子邮件（Email）等个人信息。其中，日程包括约会（Appointment）、会议（Meeting）和事件（Event）。

这些个人信息对于用户至关重要，一般不按照普通的文件访问，甚至不为用户所见，用户必须通过手机系统工具才能对这些信息进行管理，而这些工具使用手机系统的个人信息管理编程接口来进行信息的底层操作。

11.2 Android 对个人信息管理的支持

Android 平台提供 ContactsContract 类（android.provider 包中）用于对联系信息进行管理，该类包含多个子类。表 11-1 是个人信息管理相关的重要类/接口的说明。

表 11-1 个人信息管理相关的重要类/接口的说明

类/接口	说 明
ContactsContract	代表联系信息提供器与应用程序之间的接口
ContactsContract.Data	数据，包含各种个人信息，如电话号码、电子邮件地址等
ContactsContract.RawContacts	原联系表，包含每个同步源的基本联系信息
ContactsContract.Contacts	联系表，包含每个人的联系信息集合

表 11-2 是对表 11-1 中的联系信息所定义的数据类型说明。

表 11-2 个人信息数据类型说明

类/接口	说 明
ContactsContract.CommonDataKinds	数据类型
ContactsContract.CommonDataKinds.Email	电子邮件
ContactsContract.CommonDataKinds.Event	事件（生日、周年纪念等）

(续)

类/接口	说 明
ContactsContract.CommonDataKinds.Note	备注类型
ContactsContract.CommonDataKinds.Organization	组织（公司）
ContactsContract.CommonDataKinds.Phone	电话
ContactsContract.CommonDataKinds.Photo	图片
ContactsContract.CommonDataKinds.Relation	关系（父母、夫妻、兄弟、朋友等）
ContactsContract.CommonDataKinds.StructuredName	姓名
ContactsContract.CommonDataKinds.StructuredPostal	邮政编码
ContactsContract.CommonDataKinds.Website	网站信息

11.3 Android 平台个人信息管理

11.3.1 管理工具

在 Android 平台中，使用系统应用程序库中的通信录工具来管理联系信息，其主界面如图 11-1 所示。

在通信录工具中，通过操作栏中的“新建联系人”按钮可打开新建联系人界面。在该窗口中可设置联系人的标准信息，包括姓名、公司、电话、电子邮件、地址、个人网站等，如图 11-2 所示。



图 11-1 通信录工具的界面



图 11-2 新建联系人的界面

11.3.2 应用程序主 Activity 框架

Android 平台定义了多种类型的个人信息，下面将通过一个 PIM 示例程序来介绍如何在用户程序中获取联系人、电话号码、电子邮箱和公司信息，其框架定义如代码 11-1 所示。



代码 11-1 PIM 示例程序的主 Activity 的框架定义

文件名: PimDemoAct.java

```

1  public class PimDemoAct extends Activity {
2      //布局容器和内容解析器
3      private LinearLayout mLayout = null;
4      private ContentResolver mResolver = null;
5
6      @Override
7      public void onCreate(Bundle savedInstanceState) {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.main);
10         //获取布局容器对象
11         this.mLayout = (LinearLayout) findViewById(R.id.layoutPanel);
12         this.mLayout.setOrientation(LinearLayout.VERTICAL);
13         //获取内容解析器
14         this.mResolver = this.getContentResolver();
15     }
16     @Override
17     public boolean onCreateOptionsMenu(Menu menu) {
18         this.getMenuInflater().inflate(R.menu.opt_menu, menu);
19         return super.onCreateOptionsMenu(menu);
20     }
21     @Override
22     public boolean onOptionsItemSelected(MenuItem item) {
23         switch(item.getItemId()) {
24             case R.id.mi_people: { getPeople(); break; }
25             case R.id.mi_phone: { getPhones(); break; }
26             case R.id.mi_org: { getOrgs(); break; }
27             case R.id.mi_email: { getEmails(); break; }
28         }
29         return super.onOptionsItemSelected(item);
30     }
31     .....
32 };

```

在代码 11-1 中, 使用内容解析器 (第 14 行) 作为获取个人信息的接口; 以选项菜单的方式调用各个子功能 (第 24~27 行)。代码 11-2 是该选项菜单的定义。

代码 11-2 选项菜单定义

文件名: opt_menu.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/android">
3      <item android:id="@+id/mi_people" android:title="联系人"
4          android:showAsAction="ifRoom|withText" android:icon="@drawable/people"/>
5      <item android:id="@+id/mi_phone" android:title="电话号码"

```



```
6         android:showAsAction="ifRoom|withText" android:icon="@drawable/phone"/>
7     <item android:id="@+id/mi_org" android:title="公司信息"
8         android:showAsAction="ifRoom|withText" android:icon="@drawable/org"/>
9     <item android:id="@+id/mi_email" android:title="电邮信息"
10        android:showAsAction="ifRoom|withText" android:icon="@drawable/email"/>
11 </menu>
```

在代码 11-2 中，定义了 4 项菜单，且将以图标和文字共存的形式显示在操作栏中。

11.3.3 获取联系人信息

联系人信息主要包括姓、名等项。图 11-3 所示为当前设备中的联系人信息。



图 11-3 “联系人信息”的实机界面

1. 获取联系人信息

代码 11-3 是获取如图 11-3 中联系人信息的主要代码。

代码 11-3 获取联系人信息

文件名: PimDemoAct.java

```
1     private void getPeople() { //获取联系人信息
2         Cursor c = mResolver.query(ContactsContract.Data.CONTENT_URI,
3             IDataSpec.PEOPLE_COLS,
4             "("+Data.MIMETYPE+"="+StructuredName.CONTENT_ITEM_TYPE+")",
5             null, null);
6         //使用表格视图显示数据内容
7         final double colWidths[] = { 0.1f, 0.2f, 0.1f, 0.1f, 0.2f, 0.1f, 0.2f};
8         FooTableWrapper wrapper = new FooTableWrapper(this, c, colWidths);
9         //初始化列抬头映射
10        initColumnTitleMapping1(wrapper);
11        this.mLayout.addView(wrapper.getMainWidget());
12        //关闭游标
13        c.close();
14    }
```

在代码 11-3 中，通过内容解析器来查询联系信息接口 URI（CONTENT_URI）所描述的数据路径（第 2 行），并获取指定列（第 3 行）和过滤条件（第 4 行）的记录游标，继而通过表格视图组件（FooTableWrapper）展示记录游标中的内容（第 7 行）。

2. 联系人信息列

代码 11-4 是代码 11-3 中所指定的联系人信息列的定义。



代码 11-4 联系人信息列的定义

文件名: IDataSpec.java

```
1 public static final String[] PEOPLE_COLS = {
2     StructuredName._ID,           //行号
3     Data.CONTACT_ID,             //联系人编号
4     StructuredName.GIVEN_NAME,   //姓
5     StructuredName.FAMILY_NAME,  //名
6     StructuredName.DISPLAY_NAME, //显示名称
7     Contacts.TIMES_CONTACTED,    //联系次数
8     Contacts.LAST_TIME_CONTACTED //末次联系时间点
9 };
```

3. 联系信息类型

代码 11-3 中第 7 行使用 MIME 类型 (MIMETYPE) 作为记录过滤条件; 对于联系信息而言, MIME 类型是什么形式呢? 图 11-4 是联系数据库中 MIME 类型数据表的内容。

_id	mimetype
1	vnd.android.cursor.item/email_v2
2	vnd.android.cursor.item/im
3	vnd.android.cursor.item/sip_address
4	vnd.android.cursor.item/name
5	vnd.android.cursor.item/organization
6	vnd.android.cursor.item/nickname
7	vnd.android.cursor.item/phone_v2
8	vnd.android.cursor.item/postal-address_v2
9	vnd.android.cursor.item/photo
10	vnd.android.cursor.item/group_membership
11	vnd.android.cursor.item/website
12	vnd.android.cursor.item/note

图 11-4 MIME 类型数据表

实际上, 图 11-4 中的记录内容与表 11-2 中的数据类型是一一对应的。鉴于此, 读者可以将个人信息视同一个混合各类信息的大数据表, 不同类型的信息通过 MIME 类型区分。

4. 使用许可

为了读取联系信息, 应用程序必须具备读取联系信息的许可, 即需要在程序清单中声明该使用许可, 其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
```

11.3.4 获取电话号码

联系人的电话号码信息包括号码内容和号码类型, 如图 11-5 所示。



图 11-5 联系人的电话号码信息的实机界面

1. 获取电话号码信息

代码 11-5 是获取如图 11-5 中电话号码信息的主要代码。

代码 11-5 获取电话号码信息

文件名: PimDemoAct.java

```
1 private void getPhones() { //获取电话信息
2     Cursor c = mResolver.query(ContactsContract.Data.CONTENT_URI,
3                               IDataSpec.PHONE_COLS,
4                               "("+Data.MIMETYPE+"="+Phone.CONTENT_ITEM_TYPE+")",
5                               null,
6                               Data.CONTACT_ID);
7     //使用表格视图显示数据内容
8     final double colWidths[] = { 0.1f, 0.2f, 0.4f, 0.3f};
9     FooTableWrapper wrapper = new FooTableWrapper(this, c, colWidths);
10    //替换列内容
11    ArrayList<FooTableCellView> views = wrapper.getCellsViews();
12    int i = IDataSpec.INDEX_PHONE_TYPE;
13    for(; i < views.size(); i += IDataSpec.PHONE_COLS.length) {
14        int type = Integer.parseInt(views.get(i).getText().toString());
15        views.get(i).setText(PimUtil.getInstance().getPhoneTypeDesc(type));
16    }
17    //初始化列抬头映射
18    initColumnTitleMapping2(wrapper);
19    this.mLayout.addView(wrapper.getMainWidget());
20    //关闭游标
21    c.close();
22 }
```

与获取联系人信息不同，在代码 11-5 中，通过电话信息的 MIME 类型获取电话号码信息（第 4 行），此外还将电话号码的类型代码转换成描述（第 15 行）。代码 11-6 是获取电话号码类型描述的定义。

代码 11-6 获取电话号码类型描述的定义

文件名: PimUtil.java

```
1 public String getPhoneTypeDesc(int type) { //获取电话号码类型
2     switch(type) {
```



```
3      case Phone.TYPE_HOME: { return("家庭"); }
4      case Phone.TYPE_MOBILE: { return("手机"); }
5      case Phone.TYPE_WORK: { return("工作"); }
6      default: { return("未定义 (" + type + ")"); }
7    }
8  }
```

表 11-3 是 Android 平台定义的电话号码类型，这些类型定义在电话信息类中。

表 11-3 电话号码类型的说明

类 型	说 明
TYPE_CUSTOM	定制类型
TYPE_FAX_HOME	家庭传真
TYPE_FAX_WORK	工作传真
TYPE_HOME	家庭号码
TYPE_MOBILE	移动号码
TYPE_OTHER	其他号码
TYPE_PAGER	呼机号码
TYPE_WORK	工作号码

2. 电话号码信息列

代码 11-7 是代码 11-5 中所指定的电话信息列的定义。

代码 11-7 电话信息列的定义

文件名: IDataSpec.java

```
1  public static final String[] PHONE_COLS = {
2      Phone._ID,           //行号
3      Data.CONTACT_ID,     //联系人编号
4      Phone.NUMBER,        //号码
5      Phone.TYPE           //类型
6  };
7  public static final int INDEX_PHONE_TYPE = 3;
```

11.3.5 获取电子邮箱

联系人的电子邮箱信息包括邮箱地址和邮箱类型，如图 11-6 所示。



图 11-6 联系人的电子邮箱信息的实机界面

1. 获取电子邮箱信息

代码 11-8 是获取如图 11-6 中电子邮箱信息的主要代码。

代码 11-8 获取电子邮箱信息

文件名: PimDemoAct.java

```
1 private void getEmails() { //获取电子邮箱信息
2     Cursor c = mResolver.query(ContactsContract.Data.CONTENT_URI,
3         IDataSpec.EMAIL_COLS,
4         "("+Data.MIMETYPE+"="+Email.CONTENT_ITEM_TYPE+"")",
5         null, null);
6     //使用表格视图显示数据内容
7     final double colWidths[] = { 0.1f, 0.2f, 0.4f, 0.3f};
8     FooTableViewController wrapper = new FooTableViewController(this, c, colWidths);
9     //替换列内容
10    ArrayList<FooTableCellView> views = wrapper.getCellsViews();
11    int i = IDataSpec.INDEX_EMAIL_TYPE;
12    for(; i < views.size(); i += IDataSpec.EMAIL_COLS.length) {
13        int type = Integer.parseInt(views.get(i).getText().toString());
14        views.get(i).setText(PimUtil.getInstance().getEmailTypeDesc(type));
15    }
16    //初始化列抬头映射
17    initColumnTitleMapping4(wrapper);
18    this.mLayout.addView(wrapper.getMainWidget());
19    //关闭游标
20    c.close();
21 }
```

在代码 11-8 中，通过电子邮箱的 MIME 类型获取电子邮箱信息（第 4 行）；在第 13 行，将电子邮箱的类型代码转换成描述。代码 11-9 是获取电子邮箱类型描述的代码。

代码 11-9 获取电子邮箱类型描述

文件名: PimUtil.java

```
1 public String getEmailTypeDesc(int type) { //获取电子邮箱类型描述
2     switch(type) {
3         case Email.TYPE_HOME: { return ("家庭"); }
4         case Email.TYPE_MOBILE: { return ("手机"); }
5         case Email.TYPE_WORK: { return ("工作"); }
6         case Email.TYPE_OTHER: { return ("其他"); }
7         default: { return ("未定义"); }
8     }
9 }
```

表 11-4 是 Android 平台定义的电子邮箱类型，这些类型定义在电子邮箱信息类中。

表 11-4 电子邮箱类型的说明

类 型	说 明
TYPE_CUSTOM	定制邮箱
TYPE_HOME	家庭邮箱



(续)

类 型	说 明
TYPE_MOBILE	手机邮箱
TYPE_OTHER	其他邮箱
TYPE_WORK	工作邮箱

2. 电子邮箱信息列

代码 11-10 是代码 11-8 中所指定的电子邮箱信息列的定义。

代码 11-10 电子邮箱信息列的定义

文件名: IDataSpec.java

1	public static final String[] EMAIL_COLS = {
2	Email._ID, //行号
3	Data.CONTACT_ID, //联系人编号
4	Email.ADDRESS, //地址
5	Email.TYPE //类型
6	};
7	public static final int INDEX_EMAIL_TYPE = 3;

11.3.6 获取公司信息

联系人的公司信息包括公司名称、职位及公司类型等信息，如图 11-7 所示。



图 11-7 联系人的公司信息的实机界面

1. 获取公司信息

代码 11-11 是获取如图 11-7 中公司信息的主要代码。

代码 11-11 获取公司信息

文件名: PimDemoAct.java

1	private void getOrgs() {//获取公司信息
2	Cursor c = mResolver.query(ContactsContract.Data.CONTENT_URI,
3	IDataSpec.ORG_COLS,
4	"("+Data.MIMETYPE+"="+Organization.CONTENT_ITEM_TYPE+")",
5	null, null);
6	//使用表格视图显示数据内容
7	final double colWidths[] = { 0.1f, 0.2f, 0.3f, 0.2f, 0.2f};
8	FooTableViewController wrapper = new FooTableViewController(this, c, colWidths);
9	//初始化列抬头映射
10	initColumnTitleMapping3(wrapper);


```
11      this.mLayout.addView(wrapper.getMainWidget());
12      //关闭游标
13      c.close();
14  }
```

代码 11-11 中第 5 行，通过公司的 MIME 类型获取公司信息（第 4 行）。代码 11-12 是获取公司类型描述的代码。

代码 11-12 获取公司类型描述

文件名: PimUtil.java

```
1  public String getOrgTypeDesc(int type) { //获取公司类型
2      switch(type) {
3          case Organization.TYPE_CUSTOM: { return("自定义"); }
4          case Organization.TYPE_WORK: { return("工作"); }
5          case Organization.TYPE_OTHER: { return("其他"); }
6          default: { return("未定义 (" + type + ")"); }
7      }
8  }
```

表 11-5 是 Android 平台定义的公司类型，这些类型定义在公司信息类中。

表 11-5 公司类型及说明

类 型	说 明
TYPE_CUSTOM	定制公司
TYPE_OTHER	其他公司
TYPE_WORK	工作公司

2. 公司信息列

代码 11-13 是代码 11-11 中所指定的公司信息列的定义。

代码 11-13 公司信息列的定义

文件名: IDataSpec.java

```
1  public static final String[] ORG_COLS = {
2      Organization._ID,           //行号
3      Data.CONTACT_ID,           //联系人编号
4      Organization.COMPANY,       //公司
5      Organization.TITLE,         //职位
6      Organization.TYPE           //类型
7  };
8  public static final int INDEX_ORG_TYPE = 4;
```

11.4

Android 平台个人信息关联

介绍到这里，细心的读者可能会注意到，无论是联系人、电话号码、电子邮箱还是公司信息，它们有一个共同的纽带：联系 ID。通过联系 ID 可以将不同类型的个人信息串起来，



其间关联如图 11-8 所示。

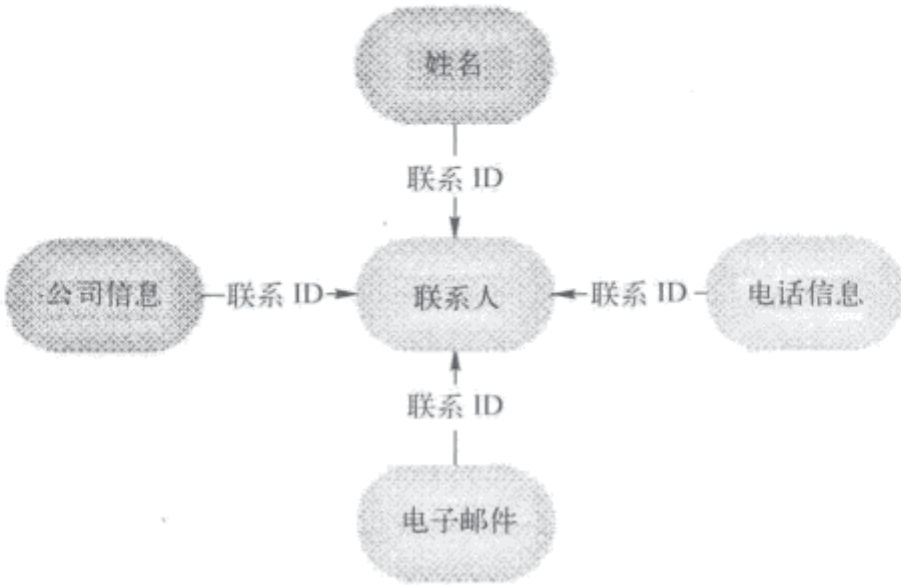


图 11-8 个人信息关联

11.4.1 联系数据库

为了分析 Android 平台中个人信息的关联，下面对系统的联系数据库进行分析，该库文件路径是 “/data/data/com.android.providers.contacts/databases/contacts2.db”。图 11-9 是使用 SQLite 数据库浏览工具对联系数据库中的内容进行浏览的界面。

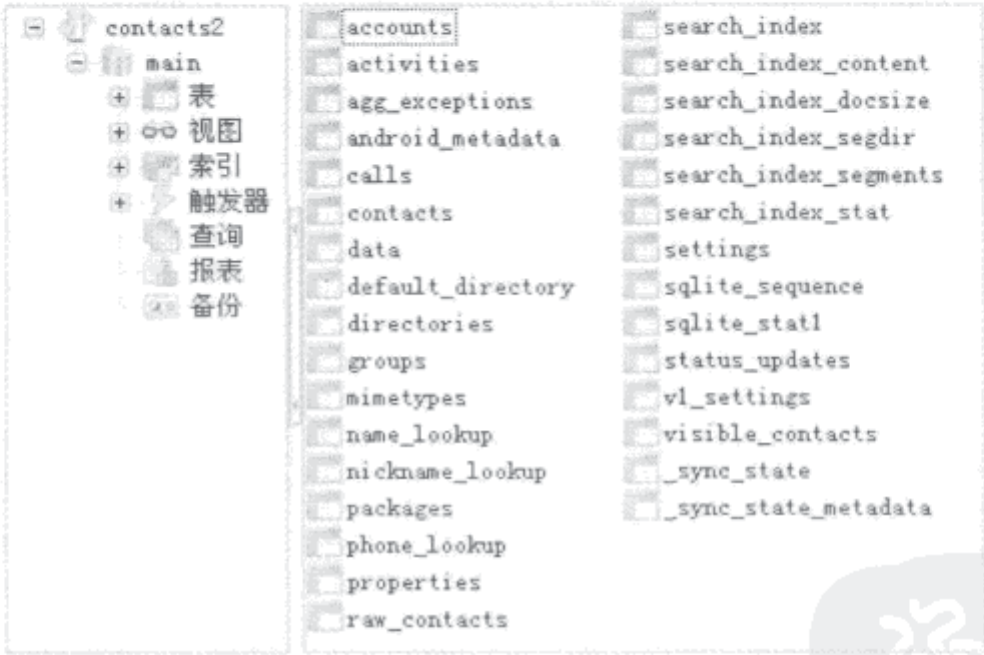


图 11-9 联系数据库中数据表

1. 联系表

图 11-4 所示的内容即为图 11-9 中 MIME 类型表（mimetypes）的记录内容；图 11-10 所示为联系表（contacts）中的记录内容。

id	name_raw_contact_id	photo_id	custom_ringtones
1	1		
3	3		

图 11-10 联系表中的记录内容

2. 姓名检索表

由图 11-10 中的列 name_raw_contact_id 推测，联系表与姓名表可能存在关联。图 11-11 所示为姓名检索表（name_lookup）中的记录内容。

3. 原联系表

由图 11-11 中的列 data_id 和 raw_contact_id 可知，姓名检索表与数据表及原联系表可能存在关联。图 11-12 所示为原联系表（raw_contacts）中的记录内容。

data_id	raw_contact_id	normalized_name	name_type
9	1	DF31CADF3E1CDF2CA0	0
9	1	DF31CADF3E1CDF2CA0	2
12	2	DF31CADF42B5DF0865	0
12	2	DF31CADF42B5DF0865	2
8	1	37494943515355333D49	4
30	3	DF0576DF3C4E	2
34	3	392D495D3D	4
35	3	5D3D392D49	4
36	1	CF2ADF31CA	3
38	3	DF0576DF46F1	3

图 11-11 姓名检索表中的记录内容

id	contact_id	display_name	sort_key
1	1	汪永松	WANG 汪 YONG 永 SONG 松
2	2	汪肇皓	WANG 汪 ZHAO 肇 HAN 皓
3	3	高益	GAO 高 YI 益

图 11-12 原联系表中的记录内容

通过对比图 11-10 和图 11-12 可知，联系表是原联系表的子集；原联系表中不仅包含有效的联系记录，而且还包含已被删除的记录（原联系表中第 2 条联系信息是创建后删除的）。

4. 数据表

图 11-13 所示为数据表（data）中的记录内容。

_id	mimetype_id	raw_contact_id	data1	data2	data3	data4
1	7	1	(555) 6	1		
2	7	1	1 555-521-5556	2		
5	1	1	foolstudio@yahoo.com.cn	1		
6	1	1	foolstudio@qq.com	2		
8	1	1	foolstudio@gmail.com	4		
9	4	1	汪永松	永松	汪	
10	5	1	简单工作室			程序员
11	11	1	http://t.qq.com/foolstudio	7		
12	4	2	汪肇皓	肇皓	汪	
13	7	2	(555) 9	1		
29	7	3	(555) 9	1		
30	4	3	高益	益	高	
31	7	3	1 555-521-5559	2		
32	5	3	ERP事业部			高级顾问
33	8	3	广东省东莞市白云路	1		广东省东莞
34	1	3	gaoyi@qq.com	1		
35	1	3	yigao@gmail.com	2		
36	6	1	阿汪	1		
37	7	1	(666) 6	3		
38	6	3	高总	1		
39	12	3	灌篮高手			
40	7	3	(777) 7	3		

图 11-13 数据表中的记录内容

由图 11-13 中的列名及记录内容可知，数据表中的记录是与原联系 ID 相关的各种类型信息（如电话号码、公司、姓名、电子邮箱等）。使用联系 ID 条件可获取指定联系人的信息合集；再添加信息类型条件可获取联系人信息子项（如电话号码、电子邮箱等）。



11.4.2 联系数据表关联

图 11-14 是根据上一节中对联系数据库中数据表的内容分析所整理的数据表之间的关联图，其中箭头指向的对象是被参考方，箭头文字是指所参考的列。

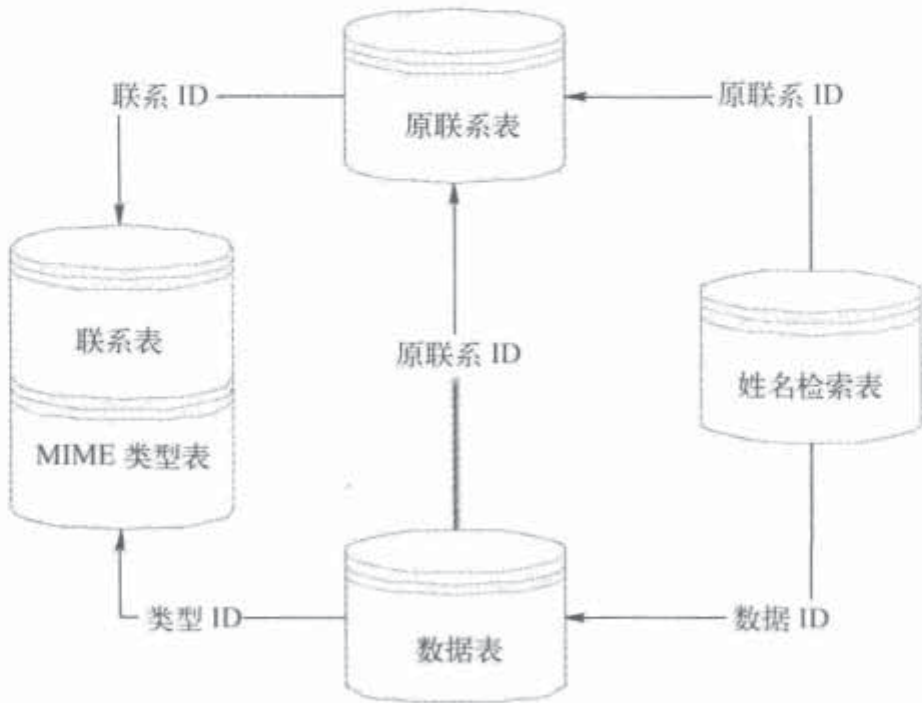


图 11-14 联系数据表关联图

在图 11-14 中，数据表需要同时参考 MIME 类型表和原联系表：参考 MIME 类型表可将全部记录按类分组，即将同类型信息归集为一组（如联系人信息、电话号码）；参考原联系信息表可将每组同类信息再按照联系人 ID 进行分组，即为每个联系人归集一组信息。其内容的结构层次如图 11-15 所示。

图 11-16 所示的界面就是将个人信息先分项再按照联系人进行归集的内容效果。

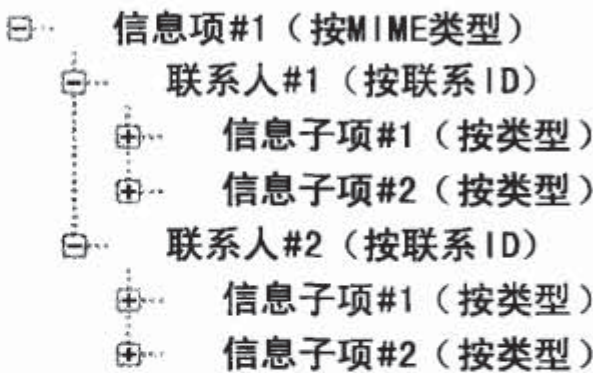


图 11-15 个人信息结构层次



图 11-16 个人信息层次管理的实机界面

在图 11-16 中，将个人信息分为多项（联系人、电话信息、电子邮件等），每一项使用一个选项卡进行展示。在选项卡视图中，又按联系人将同类信息进行归集，其核心可视组件为扩展列表视图。

第 12 章 电话系统管理

本章对 Android 平台的电话系统管理进行全面介绍，主要包括获取电话信息（设备信息、SIM 信息以及网络信息）、侦听电话状态（呼叫状态、服务状态、信号强度状态等）和手机基站定位。此外，还介绍了系统拨号器的调用和获取呼叫日志信息。

12.1 电话系统概述

不知读者是否还对购买手机的过程记忆犹新：首先要到卖场挑选一部自己喜欢的手机，在挑选过程中，需要关注两个基本要点：是否贴有入网许可标志以及是否适合自己需要的网络（如 GSM 或 CDMA）。买完手机后，还必须到网络运营商（如中国移动、中国联通或中国电信）柜台去申请一张客户识别模块（Subscriber Identity Module, SIM）卡，该 SIM 绑定了一个唯一的号码。当 SIM 正确安装完毕，用户即可利用手机通话了。

上述的这些步骤对于手机的系统而言都是必不可少的。当用户拨出电话时，手机会按照设备所支持的网络类型（GSM 或 CDMA）和频段（如 GSM 有 900MHz 和 1800MHz 等频段）来搜索运营商提供的网络；当连接到指定网络后，网络通过其入网标志信息判断该设备是否合法，继而会通过 SIM 卡中的验证信息来获取该号码相关的服务信息（如是否欠费、是否支持漫游）；当这些过程都正常时，手机网络就会根据目标号码来“告诉”目标手机有电话呼入，然后对方的手机就会以响铃或者振动的形式提醒用户。

由此可见，在电话的呼出过程中，手机必须提供很多的相关信息，而这些信息也正是手机电话系统不可缺少的内容。

提示：早在 J2SE 平台中就定义了电话功能的接口规范，即 JTAPI（Java Telephony API），该规范支持电话控制，被设计用于计算机和电话集成（CTI）的呼叫控制系统。呼叫中心（Call Center）的核心就是 CTI 技术。

通过 CTI 技术，专用分组交换机（Private Branch Exchange, PBX）可以自动将呼入的连接进行排队和分配，自动进行语音提示，在接通后还可以进行转接。

本章介绍的电话系统包括电话信息（如 SIM 卡信息、电话类型、网络信息等）和状态信息（如呼叫状态、服务状况、信号状况等）。另外，还会介绍如何使用系统定义的接口调用拨号功能以及获取通话记录。

12.2 Android 平台对电话系统的支持

Android 平台提供了电话包（android.telephony）用于对电话系统的支持。表 12-1 是该包中重要的类/接口的说明。



表 12-1 电话包中重要的类/接口的说明

类/接口	说 明
TelephonyManager	电话系统管理器，用于提供电话信息访问接口
PhoneStateListener	电话状态侦听器，用于获取电话状态信息
ServiceState	用于描述服务状态
SignalStrength	用于描述信号强度

12.3 电话系统管理

电话系统管理包括获取电话信息和状态，以及基于手机网络基站定位。

12.3.1 获取电话信息

电话信息主要包括 SIM 卡、设备和网络信息等。图 12-1 所示为实机界面。



图 12-1 获取电话信息的实机界面

1. 应用程序主 Activity 框架

代码 12-1 是获取电话信息应用程序的 Activity 组件的框架定义。

代码 12-1 获取电话信息的 Activity 组件的框架定义

文件名: TelInfoAct.java

```
1 public class TelInfoAct extends Activity {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6         //获取电话信息管理器
7         final String sname = Context.TELEPHONY_SERVICE;
8         TelephonyManager mgr = (TelephonyManager) getSystemService(sname);
9         FooTelUtil util = FooTelUtil.getInstance();
10        //获取电话信息
```



```

11         getTeleInfo(mgr, util);
12     }
13
14     private void getTeleInfo(TelephonyManager mgr, FooTelUtil util) {
15         //获取 SIM 卡信息
16         getSimInfo(mgr, util);
17         //获取设备信息
18         getDeviceInfo(mgr, util);
19         //获取网络信息
20         getNetworkInfo(mgr, util);
21     }
22     .....
23 };

```

在代码 12-1 中，Activity 组件首先获取电话信息管理器（第 8 行），继而通过该接口来获取电话的相关信息（第 16~20 行）。

2. 获取 SIM 卡信息

代码 12-2 是获取 SIM 卡信息的主要代码。SIM 卡信息包括国别码、提供商代码及名称、序号和状态信息。其中，序号又称为国际移动设备识别码（International Mobile Equipment Identity number, IMEI）。

代码 12-2 获取 SIM 卡信息

文件名: TelInfoAct.java

```

1     private void getSimInfo(TelephonyManager mgr, FooTelUtil util) { //获取 SIM 卡信息
2         printText("国别: " + mgr.getSimCountryIso() );
3         printText("服务提供商: " + mgr.getSimOperator() );
4         printText("服务商名: " + mgr.getSimOperatorName() );
5         printText("序号: " + mgr.getSimSerialNumber() );
6         printText("状态: " + util.getSimStateDesc(mgr.getSimState() ) );
7     }

```

在代码 12-2 中，使用电话系统管理器的 getSimState 方法（第 6 行）可获取 SIM 卡状态类型，继而通过电话系统工具类获取类型描述。代码 12-3 是获取 SIM 卡状态描述的主要代码。

代码 12-3 获取 SIM 卡状态描述

文件名: FooTelUtil.java

```

1     public String getSimStateDesc(int type) { //获取 SIM 卡状态描述
2         switch(type) {
3             case TelephonyManager.SIM_STATE_ABSENT: { return ("无 SIM 卡"); }
4             case TelephonyManager.SIM_STATE_NETWORK_LOCKED: { return ("网络锁定"); }
5             case TelephonyManager.SIM_STATE_PIN_REQUIRED: { return ("需要 PIN 码"); }
6             case TelephonyManager.SIM_STATE_PUK_REQUIRED: { return ("需要 PUK 码"); }
7             case TelephonyManager.SIM_STATE_READY: { return ("就绪"); }

```




```
8         case TelephonyManager.SIM_STATE_UNKNOWN: { return ("未知"); }
9     }
10    return ("未知类型");
11 }
```

表 12-2 是 Android 平台所定义的 SIM 卡状态类型的说明，这些类型在电话信息管理器类中定义。

表 12-2 SIM 卡状态类型的说明

类 型	说 明
SIM_STATE_ABSENT	未插卡
SIM_STATE_NETWORK_LOCKED	网络被锁定，需要网络 PIN 解锁
SIM_STATE_PIN_REQUIRED	需要 PIN 码，需要 SIM PIN 码解锁
SIM_STATE_PUK_REQUIRED	需要 PUK 码，需要 SIM 卡 PUK 码解锁
SIM_STATE_READY	准备就绪
SIM_STATE_UNKNOWN	未知状态

提示：个人识别码（Personal Identity Number，PIN）保存在 SIM 卡里，用于识别当前手机用户的身份。当由于 PIN 原因被锁定时，需要通过个人解锁码（Personal Unblocking Key，PUK）来解锁。PIN 和 PUK 的信息在用户所购买 SIM 卡的产品信息中被提供。

3. 获取设备信息

代码 12-4 是获取电话设备信息的主要代码。设备信息包括设备标识、软件版本、电话号码和电话类型。

代码 12-4 获取电话设备信息

文件名: TelInfoAct.java

```
1 private void getDeviceInfo(TelephonyManager mgr, FooTelUtil util) { //获取电话设备信息
2     printText("设备标识: " + mgr.getDeviceId() );
3     printText("软件版本: " + mgr.getDeviceSoftwareVersion() );
4     printText("线路 1 号码: " + mgr.getLine1Number() );
5     printText("电话类型: " + util.getPhoneTypeDesc(mgr.getPhoneType() ) );
6 }
```

在代码 12-4 中，使用电话信息管理器的 getPhoneType 方法（第 5 行）可获取电话类型，继而通过电话系统工具类获取类型描述。代码 12-5 是获取电话类型描述的主要代码。

代码 12-5 获取电话类型描述

文件名: FooTelUtil.java

```
1 public String getPhoneTypeDesc(int type) { //获取电话类型描述
2     switch(type) { case TelephonyManager.PHONE_TYPE_CDMA: { return ("CDMA"); }
3                   case TelephonyManager.PHONE_TYPE_GSM: { return ("GMS"); }
4                   case TelephonyManager.PHONE_TYPE_NONE: { return ("不可用"); }
```



```
5      }
6      return ("未知类型");
7  }
```

表 12-3 是 Android 平台所定义的电话类型的说明，这些类型在电话信息管理器类中定义。

表 12-3 电话类型的说明

类 型	说 明
PHONE_TYPE_GSM	GSM 手机
PHONE_TYPE_NONE	未知
PHONE_TYPE_CDMA	CDMA 手机

4. 获取网络信息

代码 12-6 是获取网络信息的主要代码。网络信息包括国别码、运营商代码及名称、网络类型。

代码 12-6 获取网络信息

文件名: TelInfoAct.java

```
1  private void getNetworkInfo(TelephonyManager mgr, FooTelUtil util) { //获取网络信息
2      printText("网络国别: " + mgr.getNetworkCountryIso() );
3      printText("网络运营商: " + mgr.getNetworkOperator() );
4      printText("运营商名: " + mgr.getNetworkOperatorName() );
5      printText("网络类型: " + util.getNetworkTypeDesc(mgr.getNetworkType() ) );
6  }
```

在代码 12-6 中，使用电话系统管理器的 getNetworkType 方法（第 5 行）可获取网络类型，继而通过电话系统工具类获取类型描述。代码 12-7 是获取网络类型描述的主要代码。

代码 12-7 获取网络类型描述

文件名: FooTelUtil.java

```
1  public String getNetworkTypeDesc(int type) { //获取网络类型描述
2      switch(type) {
3          case TelephonyManager.NETWORK_TYPE_EDGE: { return ("EDGE"); }
4          case TelephonyManager.NETWORK_TYPE_GPRS: { return ("GPRS"); }
5          case TelephonyManager.NETWORK_TYPE_UMTS: { return ("UMTS"); }
6          case TelephonyManager.NETWORK_TYPE_UNKNOWN: { return ("未知"); }
7      }
8      return ("未知类型");
9  }
```

表 12-4 是 Android 平台所定义的网络类型的说明，这些类型在电话信息管理器类中定义。

表 12-4 网络类型的说明

类 型	说 明
NETWORK_TYPE_EDGE	GSM 增强数据率演进 (2.75G)
NETWORK_TYPE_GPRS	通用分组无线服务 (2.5G)



(续)

类 型	说 明
NETWORK_TYPE_UMTS	全球移动通信系统 (3G)
NETWORK_TYPE_UNKNOWN	未知网络
NETWORK_TYPE_CDMA	CDMA 网络

提示: GSM 增强数据率演进 (Enhanced Data Rates for GSM Evolution, EDGE) 是一种数字移动电话技术, 作为一个 2G 和 2.5G (GPRS) 的延伸, 有时被称为 2.75G, 这项技术工作在 TDMA 和 GSM 网络中。通用分组无线服务 (General Packet Radio Service, GPRS) 是 GSM 移动电话用户使用的一种移动数据业务。全球通用移动通信系统 (Universal Mobile Telecommunications System, UMTS) 是当前最广泛采用的一种 3G 移动电话技术, 它的无线接口使用 WCDMA 技术。UMTS 分组交换系统是由 GPRS 系统演进而来的。

5. 使用许可

获取电话信息需要取得读取电话状态的许可, 即需要在应用程序清单文件中对该使用权限进行声明, 其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
```

12.3.2 电话状态

电话状态包括呼叫状态、服务状态、信号强度、数据连接状态等。图 12-2 所示为电话状态的侦听输出。

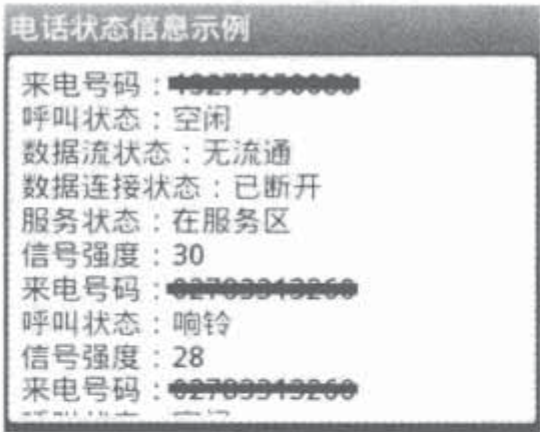


图 12-2 电话状态的侦听输出的实机界面

1. 应用程序主 Activity 框架

代码 12-8 是获取电话状态应用程序的 Activity 组件的框架定义。

代码 12-8 获取电话状态的 Activity 组件的框架定义

文件名: TelStateInfoAct.java

```
1 public class TelStateInfoAct extends Activity {
2     //电话系统工具类
```



```
3      private FooTelUtil mUtil = FooTelUtil.getInstance();
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //初始化电话服务
10         final String sname = Context.TELEPHONY_SERVICE;
11         TelephonyManager mgr = (TelephonyManager) getSystemService(sname);
12         //设置事件侦听
13         mgr.listen(new FooListener(), PhoneStateListener.LISTEN_CALL_STATE);
14         mgr.listen(new FooListener(), PhoneStateListener.LISTEN_DATA_ACTIVITY);
15         mgr.listen(new FooListener(),
16                     PhoneStateListener.LISTEN_DATA_CONNECTION_STATE);
17         mgr.listen(new FooListener(), PhoneStateListener.LISTEN_SERVICE_STATE);
18         mgr.listen(new FooListener(), PhoneStateListener.LISTEN_SIGNAL_STRENGTHS);
19     }
20     .....
21 };
```

在代码 12-8 中，Activity 组件首先获取电话信息管理器（第 11 行），继而通过该接口来绑定电话状态事件的侦听器（第 13~18 行）。该代码中绑定了对 5 种事件的侦听，即呼叫状态、数据活动、连接状态、服务状态和信号强度。

表 12-5 是 Android 平台所定义的电话侦听事件类型的说明，这些类型在电话状态侦听器类中定义。

表 12-5 电话侦听事件类型的说明

类 型	说 明
LISTEN_CALL_FORWARDING_INDICATOR	侦听呼叫转移指示器改变事件
LISTEN_CALL_STATE	侦听呼叫状态改变事件
LISTEN_CELL_LOCATION	侦听设备位置改变事件
LISTEN_DATA_ACTIVITY	侦听数据连接的流向改变事件
LISTEN_DATA_CONNECTION_STATE	侦听数据连接状态改变事件
LISTEN_MESSAGE_WAITING_INDICATOR	侦听消息等待指示器改变事件
LISTEN_NONE	停止侦听
LISTEN_SERVICE_STATE	侦听网络服务状态
LISTEN_SIGNAL_STRENGTHS	侦听网络信号强度

2. 电话状态侦听器

代码 12-9 是代码 12-8 中所参考的电话状态侦听器的定义。

代码 12-9 电话状态侦听器的定义

文件名：TelStateInfoAct.java

```
1      class FooListener extends PhoneStateListener { //电话状态侦听器
2          @Override
3          public void onDataActivity(int dir) {
4              printText("数据流状态: " + mUtil.getDataActTypeDesc(dir));
```




```
5      }
6      @Override
7      public void onDataConnectionStateChanged(int state) {
8          printText("数据连接状态: " + mUtil.getDataConnTypeDesc(state));
9      }
10     @Override
11     public void onCallStateChanged(int state, String incomingNumber) {
12         printText("来电号码: " + incomingNumber);
13         printText("呼叫状态: " + mUtil.getCallStateTypeDesc(state));
14     }
15     @Override
16     public void onServiceStateChanged(ServiceState state) {
17         printText("服务状态: " + mUtil.getServiceStateDesc(state));
18     }
19     @Override
20     public void onSignalStrengthsChanged(SignalStrength ss) {
21         printText("信号强度: "+ss.toString() );
22     }
23     };
```

在代码 12-9 中，使用重载电话状态侦听器的相关接口来侦听相应的状态内容，如接口 `onCallStateChanged` 用于侦听呼叫状态；接口 `onServiceStateChanged` 用于侦听服务状态等。

3. 侦听呼叫状态

代码 12-9 中的 `onCallStateChanged` 方法用于侦听电话的呼叫状态，其中包括来电号码。代码 12-10 是获取呼叫状态描述的主要代码。

代码 12-10 获取呼叫状态描述

文件名: FooTelUtil.java

```
1  public String getCallStateTypeDesc(int type) {
2      switch(type) {   case TelephonyManager.CALL_STATE_IDLE: { return ("空闲"); }
3                      case TelephonyManager.CALL_STATE_OFFHOOK: { return ("摘机"); }
4                      case TelephonyManager.CALL_STATE_RINGING: { return ("响铃"); }
5      }
6      return ("未知类型");
7  }
```

表 12-6 是 Android 平台所定义的电话呼叫状态类型的说明，这些类型在电话信息管理器类中定义。

表 12-6 电话呼叫状态类型的说明

类 型	说 明
CALL_STATE_IDLE	空闲（没呼入或已挂机）
CALL_STATE_RINGING	响铃（有呼入）
* CALL_STATE_OFFHOOK	摘机（接听中）

图 12-3 所示为电话 5554 与 5556 正在进行通话，则此时的呼叫状态为摘机。

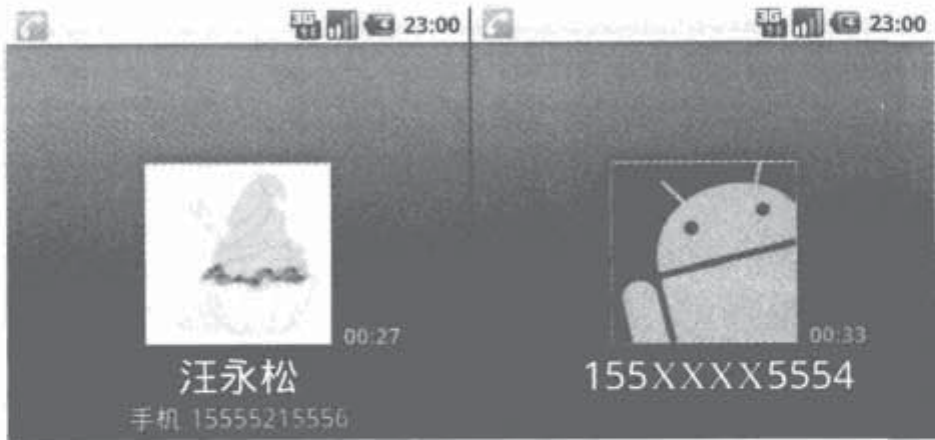


图 12-3 通话的实机界面

4. 侦听服务状态

代码 12-9 中的 `onServiceStateChanged` 方法用于侦听电话的服务状态。代码 12-11 是获取服务状态描述的主要代码。

代码 12-11 获取服务状态描述

文件名: FooTelUtil.java

```
1 public String getServiceStateDesc(ServiceState state) {
2     switch(state.getState()) {
3         case ServiceState.STATE_EMERGENCY_ONLY: { return ("只能应急呼叫"); }
4         case ServiceState.STATE_IN_SERVICE: { return ("在服务区"); }
5         case ServiceState.STATE_OUT_OF_SERVICE: { return ("不在服务区"); }
6         case ServiceState.STATE_POWER_OFF: { return ("已关机"); }
7     }
8     return ("未知状态");
9 }
```

表 12-7 中 Android 平台所定义的电话服务状态类型的说明，这些类型在服务状态类中定义。

表 12-7 电话服务状态类型的说明

类 型	说 明
STATE_EMERGENCY_ONLY	仅限紧急呼叫
STATE_IN_SERVICE	在服务区
STATE_OUT_OF_SERVICE	不在服务器
STATE_POWER_OFF	关机

此外，通过服务状态接口的 `getRoaming` 方法可以获知手机是否为漫游状态。

5. 侦听信号强度

代码 12-9 中的 `onSignalStrengthsChanged` 方法用于侦听电话的信号强度，通过信号强度接口的 `toString` 方法可以获取有关信号强度信息。



6. 侦听数据连接状态

代码 12-9 中的 `onDataConnectionStateChanged` 方法用于侦听数据连接状态。代码 12-12 是获取数据连接状态描述的主要方法。

代码 12-12 获取数据连接状态描述

文件名: FooTelUtil.java

```
1 public String getDataConnTypeDesc(int type) {
2     switch(type) { case TelephonyManager.DATA_CONNECTED: { return ("已连接"); }
3                   case TelephonyManager.DATA_CONNECTING: { return ("正在连接"); }
4                   case TelephonyManager.DATA_DISCONNECTED: { return ("已断开"); }
5                   case TelephonyManager.DATA_SUSPENDED: { return ("暂停"); }
6     }
7     return ("未知类型");
8 }
```

表 12-8 是 Android 平台所定义的电话数据连接状态类型的说明；这些类型在电话信息管理器类中定义。

表 12-8 电话数据连接状态类型及说明

类 型	说 明
DATA_DISCONNECTED	断开
DATA_CONNECTING	正在连接
DATA_CONNECTED	已连接
DATA_SUSPENDED	已暂停

7. 侦听数据活动状态

代码 12-9 中的 `onDataActivity` 方法用于侦听数据活动状态。代码 12-13 是获取数据活动状态描述的主要代码。

代码 12-13 获取数据活动状态描述

文件名: FooTelUtil.java

```
1 public String getDataActTypeDesc(int type) {
2     switch(type) {
3         case TelephonyManager.DATA_ACTIVITY_DORMANT: { return ("蛰伏"); }
4         case TelephonyManager.DATA_ACTIVITY_IN: { return ("流进"); }
5         case TelephonyManager.DATA_ACTIVITY_INOUT: { return ("进出"); }
6         case TelephonyManager.DATA_ACTIVITY_NONE: { return ("无流通"); }
7         case TelephonyManager.DATA_ACTIVITY_OUT: { return ("流出"); }
8     }
9     return ("未知类型");
10 }
```

表 12-9 是 Android 平台所定义的电话数据活动状态类型的说明，这些类型在电话信息

管理器类中定义。

表 12-9 电话数据活动状态类型的说明

类 型	说 明
DATA_ACTIVITY_NONE	无数据流动
DATA_ACTIVITY_IN	数据流入
DATA_ACTIVITY_OUT	数据流出
DATA_ACTIVITY_INOUT	数据交互
DATA_ACTIVITY_DORMANT	蛰伏，网络已连接，但物理链路无连接

8. 使用许可

获取电话状态需要取得读取电话状态的许可，即需要在应用程序清单文件中对该使用权限进行声明，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
```

12.3.3 手机网络基站定位

手机网络的基站除了提供信号连接之外，还可以用于定位。图 12-4 所示为通过手机网络基站获取手机定位信息的实机界面。

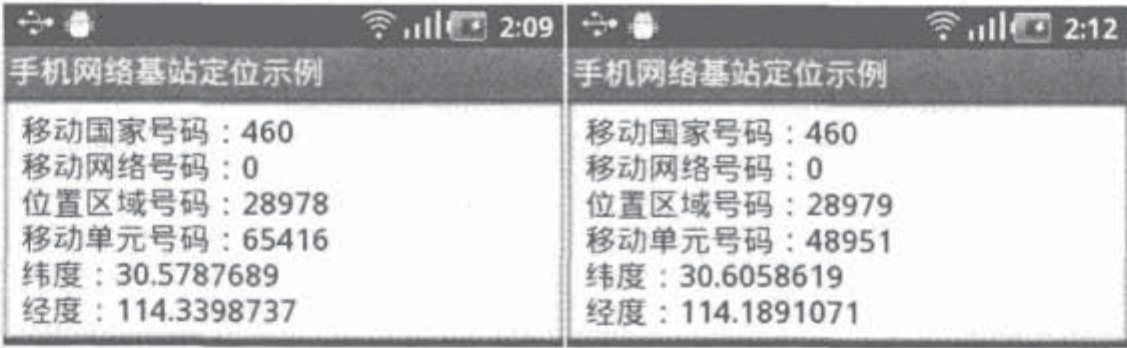


图 12-4 手机网络基站获取手机定位信息的实机界面

1. 应用程序主 Activity 框架

代码 12-14 是该基站定位示例程序的主 Activity 组件的框架定义，其界面只有一个文本框来显示定位信息。

代码 12-14 应用程序的主 Activity 组件的框架定义

文件名：CellLocationAct.java

```
1 public class CellLocationAct extends Activity {
2     //主线程消息队列处理器
3     private Handler mHandler = null;
4     private TelephonyManager mTelMgr = null;
5     private PhoneStateListener mListener = null;
6
7     @Override
```




Android 平台开发之旅 第2版

```

8      public void onCreate(Bundle savedInstanceState) {
9          super.onCreate(savedInstanceState);
10         setContentView(R.layout.main);
11         //获取电话信息服务管理器
12         final String sname = Context.TELEPHONY_SERVICE;
13         mTelMgr = (TelephonyManager)this.getSystemService(sname);
14         mListener = new FooPhoneStateListener();
15         //初始化主线程消息队列处理器
16         mHandler = new Handler() {
17             @Override
18             public void handleMessage(Message msg) { Bundle data = msg.getData();
19                 printText("纬度: "+data.getString(IConfig.EXTRA_LAT));
20                 printText("经度: "+data.getString(IConfig.EXTRA_LON));
21             }
22         };
23         //检查互联网是否连通, 如连通则开始侦听蜂窝单元位置事件
24         if(checkInternet()==true) {
25             mTelMgr.listen(mListener, PhoneStateListener.LISTEN_CELL_LOCATION);
26         }
27     }
28     @Override
29     protected void onDestroy() { super.onDestroy();
30         //取消侦听
31         mTelMgr.listen(mListener, PhoneStateListener.LISTEN_NONE);
32     };
33
34     private boolean checkInternet() {
35         final String sname = Context.CONNECTIVITY_SERVICE;
36         ConnectivityManager mgr = (ConnectivityManager)this.getSystemService(sname);
37         NetworkInfo info = mgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
38         if(info.getState() != NetworkInfo.State.CONNECTED) {
39             Toast.makeText(this,"网络未连接!",Toast.LENGTH_LONG).show();
40             Intent i = new Intent(WifiManager.ACTION_PICK_WIFI_NETWORK);
41             this.startActivityForResult(i, IConfig.REQ_CODE);
42             return (false);
43         }
44         return (true);
45     }
46
47     @Override
48     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
49         if(requestCode==IConfig.REQ_CODE) { //如启用网络连接则开始侦听
50             mTelMgr.listen(mListener, PhoneStateListener.LISTEN_CELL_LOCATION);
51         }
52     }
53     .....
54 };

```

在代码 12-14 中, 首先获取电话信息管理器实例 (第 13 行), 然后通过电话状态侦听器侦听蜂窝单元的定位事件 (第 25 行); 由于该程序需要通过互联网查询手机定位信息的经纬

度值，所以还要对网络状态进行检查，开启 Wi-Fi 连接（第 41 行）。

此外，还创建了主线程消息队列处理器，用于接收外部线程传递过来的经纬度信息（第 16~22 行）。

2. 应用程配置信息接口

代码 12-15 是应用程序中有关配置信息接口的定义，包括 Activity 调用请求代码和意向扩展数据项名。

代码 12-15 应用程序配置信息接口的定义

文件名: IConfig.java

```
1 public interface IConfig {
2     public static final String EXTRA_LAT = "latitude";
3     public static final String EXTRA_LON = "longitude";
4     //Activity 调用请求代码
5     public static final int REQ_CODE = 2011;
6 };
```

3. 获取手机位置信息

使用电话状态侦听器侦听蜂窝单位的定位事件即可获得手机的位置信息。代码 12-16 是代码 12-14 中所参考的电话状态侦听器的定义。

代码 12-16 电话状态侦听器的定义

文件名: CellLocationAct.java

```
1 class FooPhoneStateListener extends PhoneStateListener {
2     @Override
3     public void onCellLocationChanged(CellLocation loc) {
4         String operator = mTelMgr.getNetworkOperator();
5         //获取手机当前的位置区域识别信息
6         int mcc = Integer.parseInt(operator.substring(0,3));
7         int mnc = Integer.parseInt(operator.substring(3));
8         int lac = ((GsmCellLocation)loc).getLac();
9         int cid = ((GsmCellLocation)loc).getCid();
10        Lai lai = new Lai(mcc,mnc,lac,cid);
11        showLai(lai);
12        //启动查询线程
13        LocQueryThread t = new LocQueryThread(mHandler, lai);
14        t.start();
15    }
16 };
17
18 protected void showLai(Lai lai) { //显示位置区域识别信息
19     printText("移动国家号码: "+lai.getMcc());
20     printText("移动网络号码: "+lai.getMnc());
21     printText("位置区域号码: "+lai.getLac());
22     printText("移动单元号码: "+lai.getCid());
23 }
```




在代码 12-16 中, 使用电话信息管理器的 `getNetworkOperator` 方法可获取手机网络运营代码, 其中包括移动国家号码和移动网络号码信息 (第 4~7 行); 再使用蜂窝单元定位接口的方法可获取手机的位置区域号码和移动单元号码 (第 8 行和第 9 行)。

注意: 代码 12-16 中将单元定位接口 (`CellLocation`) 转换为 GSM 单元定位接口 (`GsmCellLocation`) 来获取 GSM 手机的位置信息; 对于 CDMA 手机而言, 将单元定位接口转换为 CDMA 单元定位接口 (`CdmaCellLocation`) 即可获取 CDMA 手机的位置信息。使用 CDMA 单元定位接口的 `getBaseStationLatitude` 和 `getBaseStationLongitude` 方法可直接获取基站的经纬度信息。

移动国家号码 (Mobile Country Code, MCC)、移动网络号码 (Mobile Network Code, MNC)、位置区域号码 (Location Area Code, LAC) 和移动单元号码 (Cellular ID, CID) 即构成了手机的位置区域标识 (Location Area Identity, LAI)。代码 12-17 是手机的位置区域标识的定义。

代码 12-17 位置区域标识的定义

文件名: Lai.java

```

1  public class Lai {
2      private int mcc;
3      private int mnc;
4      private int lac;
5      private int cid;
6
7      public Lai(int mcc, int mnc, int lac, int cid) {
8          this.mcc = mcc; this.mnc = mnc; this.lac = lac; this.cid = cid;
9      }
10
11     //移动国家号码
12     public int getMcc() { return mcc; }
13     public void setMcc(int mcc) { this.mcc = mcc; }
14     //移动网络号码
15     public int getMnc() { return mnc; }
16     public void setMnc(int mnc) { this.mnc = mnc; }
17     //位置区域号码
18     public int getLac() { return lac; }
19     public void setLac(int lac) { this.lac = lac; }
20     //移动单元号码
21     public int getCid() { return cid; }
22     public void setCid(int cid) { this.cid = cid; }
23 };

```

4. 获取经纬度信息

对于手机网络基站定位的功能, 实际上是网络运营商将手机网络服务区域按位置区域和单元编号的级次划分成一个个小单元, 为每个单元都分配一个唯一的标识, 通过该标识即可获取位置信息, 而这个标识就是位置区域标识。

位置区域标识是由运营商定义的，普通用户无法获知其中包含的内容，所以还需要将位置区域标识中包含的位置信息转换为常规的经纬度信息。代码 12-18 就是查询位置区域标识所对应的经纬度信息的查询线程的定义。

代码 12-18 手机位置信息查询线程的定义

文件名: LocQueryThread.java

```

1  public class LocQueryThread extends Thread {
2      //主线程消息队列处理器
3      private Handler mHandler = null;
4      private Lai mLai = null;
5
6      public LocQueryThread(Handler handler, Lai lai) {
7          this.mHandler = handler; this.mLai = lai;
8      }
9
10     @Override
11     public void run() {
12         try { queryLatitudeLongitude(mLai);
13             } catch (ClientProtocolException e) { e.printStackTrace(); }
14             catch (JSONException e) { e.printStackTrace(); }
15             catch (IOException e) { e.printStackTrace(); }
16     }
17
18     //查询位置区域识别码所对应的经纬度信息
19     private void queryLatitudeLongitude(Lai lai)
20         throws JSONException, ClientProtocolException, IOException {
21         //拼凑 JSON 查询字符串
22         JSONObject builder = new JSONObject();
23         builder.put("version", "1.1.0");
24         builder.put("host", "maps.google.com");
25         builder.put("request_address", true);
26         JSONArray array = new JSONArray();
27         JSONObject items = new JSONObject();
28         items.put("cell_id", lai.getCid() );
29         items.put("location_area_code", lai.getLac() );
30         items.put("mobile_country_code", lai.getMcc() );
31         items.put("mobile_network_code", lai.getMnc() );
32         array.put(items);
33         builder.put("cell_towers", array);
34         //创建连接，发送请求并接受回应
35         DefaultHttpClient client = new DefaultHttpClient();
36         HttpPost post = new HttpPost("http://www.google.com/loc/json");
37         StringEntity se = new StringEntity(builder.toString());
38         post.setEntity(se);
39         HttpResponse resp = client.execute(post);
40         //获取回应条目
41         HttpEntity entity = resp.getEntity();

```




```

42         BufferedReader br = new BufferedReader(
43             new InputStreamReader(entity.getContent()));
44         StringBuffer sb = new StringBuffer();
45         String line = br.readLine();
46         while (line != null) { sb.append(line); line = br.readLine(); }
47         //构建 JSON 对象
48         JSONObject all = new JSONObject(sb.toString());
49         JSONObject loc = new JSONObject(all.getString("location"));
50         br.close();
51         //获取经纬度值
52         String latitude = loc.getString("latitude");
53         String longitude = loc.getString("longitude");
54         //发送数据给主线程
55         FooSysUtil.getInstance().postMsgs(mHandler,
56             new String[] { IConfig.EXTRA_LAT, IConfig.EXTRA_LON },
57             new String[] { latitude, longitude });
58     }
59 };

```

在代码 12-18 中, 该线程将位置区域标识信息通过 JSON 格式进行组织 (第 22~33 行), 然后以 HTTP 的方式发送给位置查询服务器 (第 35~39 行), 然后获取服务器返回, 解析其中所包含的经纬度信息 (第 52 行和第 53 行), 并通过主线程消息队列处理器发送给主线程 (第 55 行)。

有关主线程对所收到的经纬度信息的处理参见代码 12-14 (第 16~22 行)。

5. 使用许可

获取手机定位信息不仅需要拥有读取电话状态的许可, 还需要访问位置信息的许可。此外, 对于位置区域标识的查询需要访问网络状态并访问互联网, 所以必须在应用程序清单文件对这些使用权限进行声明, 其代码如下所示。

文件名: AndroidManifest.xml

```

<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

```

12.4 拨号及呼叫日志管理

12.4.1 拨号功能

通过电话系统管理器虽然可以获取电话信息和侦听电话状态, 但是不能提供拨号功能。使用 Android 平台提供的意向组件可以实现在用户程序中对系统的拨号器进行调用。图 12-5 所示为一款调用系统拨号器的实机界面。



图 12-5 调用系统拨号器的实机界面

代码 12-19 是调用系统拨号器程序的 Activity 组件的框架定义。

代码 12-19 调用系统拨号器程序的 Activity 组件的框架定义

文件名: DialerDemoAct.java

```
1 public class DialerDemoAct extends Activity implements OnClickListener {
2     @Override
3     public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6         //获取按钮控件并设置侦听器
7         Button btnCall = (Button)findViewById(R.id.btn_call);
8         btnCall.setOnClickListener(this);
9     }
10    @Override
11    public void onClick(View v) {
12        String number = ((EditText)findViewById(R.id.txt_num)).getText().toString();
13        Uri uri = Uri.parse("tel:"+number);
14        Intent dialIntent = new Intent(Intent.ACTION_DIAL, uri);
15        startActivity(dialIntent);
16    }
17 };
```

在代码 12-19 中，Activity 使用系统所定义的标准拨号动作来调用系统的拨号器组件（第 14 行的 ACTION_DIAL），该动作的参数是以 tel 作为模式，目标号码为内容的资源标识符（第 13 行）。

12.4.2 日志

当电话接收呼入或呼出时，系统会将呼叫日志保存到系统数据库中，用户通过内容提供框架可以获取该日志。图 12-6 所示为呼叫日志的实机界面。



行号	号码	类型	日期	时长
680	*****	呼出	2011-09-29 10:01:35	0'12"
687	*****	呼入	2011-09-29 11:50:30	3'29"
688	*****	呼入	2011-09-29 13:42:58	0'37"
689	*****	呼入	2011-09-29 14:46:08	0'13"
690	*****	呼入	2011-09-29 14:46:08	0'13"

图 12-6 呼叫日志的实机界面

1. 应用程序主 Activity 框架

代码 12-20 是获取呼叫日志程序的 Activity 组件的框架定义。

代码 12-20 获取呼叫日志程序的 Activity 组件的框架定义

文件名: CallLogAct.java

```
1 public class CallLogAct extends Activity {
2     //布局容器及内容解析器
3     private LinearLayout mLayout = null;
4     private ContentResolver mResolver = null;
5
6     @Override
7     public void onCreate(Bundle savedInstanceState) {
8         super.onCreate(savedInstanceState);
9         setContentView(R.layout.main);
10        //获取布局容器对象
11        this.mLayout = (LinearLayout) findViewById(R.id.layoutPanel);
12        this.mLayout.setOrientation(LinearLayout.VERTICAL);
13        //获取内容解析器
14        this.mResolver = this.getContentResolver();
15        //显示呼叫记录
16        showCallLogs();
17    }
18
19    private void showCallLogs() {
20        Cursor c = mResolver.query(CallLog.Calls.CONTENT_URI,
21                                   IDataSpec.CALLLOG_COLS,
22                                   "("+CallLog.Calls.DURATION+">0)", null, null);
23        //使用表格视图显示数据内容
24        final double colWidths[] = { 0.15f, 0.25f, 0.15f, 0.3f, 0.15f};
25        FooTableWrapper wrapper = new FooTableWrapper(this, c, colWidths);
26        //调整各列内容
27        ArrayList<FooTableCellView> views = wrapper.getCellsViews();
```



```

28         for(int i = 1; i < views.size(); i += IDataSpec.CALLLOG_COLS.length) {
29             //电话号码列
30             views.get(i).setTextSize(10);
31             //呼入类型列
32             int type = Integer.parseInt(views.get(i+1).getText().toString());
33             views.get(i+1).setText(FooTelUtil.getInstance().getCallTypeDesc(type));
34             //日期列
35             long epoch = Long.parseLong(views.get(i+2).getText().toString());
36             views.get(i+2).setText(FooSysUtil.getInstance().unixTsp2Str(epoch));
37             views.get(i+2).setTextSize(9);
38             //时长列
39             int sec = Integer.parseInt(views.get(i+3).getText().toString());
40             views.get(i+3).setText(FooSysUtil.getInstance().toTimeStr(sec));
41         }
42         //初始化列抬头映射
43         initColumnTitleMapping(wrapper);
44         this.mLayout.addView(wrapper.getMainWidget());
45         //关闭游标
46         c.close();
47     }
48     .....
49 };

```

在代码 12-20 中，使用 Activity 组件的内容解析器（第 14 行）按照呼叫日志的内容资源标识符去获取呼叫日志的记录游标（第 20 行），然后使用表格视图显示记录游标中的内容（第 25 行）。

此外，在读取记录后，调整了部分列的内容显示，包括字体大小（第 30 行和第 37 行）以及对显示内容进行转换（第 33 行、第 36 行和第 40 行）。

2. 呼叫日志数据规格

代码 12-21 是代码 12-20 中所参考的呼叫日志数据规格的定义。

代码 12-21 呼叫日志数据规格的定义

文件名: IDataSpec.java

```

1     public interface IDataSpec {
2         public static final String[] CALLLOG_COLS = {
3             CallLog.Calls._ID,           //行号
4             CallLog.Calls.NUMBER,        //电话号码
5             CallLog.Calls.TYPE,          //呼叫类型
6             CallLog.Calls.DATE,          //呼叫时间（自基准时间以来的毫秒数）
7             CallLog.Calls.DURATION       //时长（秒）
8         };
9         public static final int COL_TYPE = 2;
10        public static final int COL_DATE = 3;
11        public static final int COL_DURATION = 4;
12    };

```




在代码 12-21 中，由于呼叫时间是 UNIX 格式（记录的是相对 1970-1-1 以来的毫秒数），所以需要将该数值转换成符合用户阅读习惯的“YYYY-MM-DD HH:MM:SS”时间格式。代码 12-22 是将 UNIX 时间戳转换成日期时间字符串的方法。

代码 12-22 将 UNIX 时间戳转换成日期时间字符串

文件名: FooSysUtil.java

```
1 public String unixTsp2Str(long epoch) { //将 UNIX 时间戳转换成日期时间字符串
2     SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
3     sdf.setTimeZone(TimeZone.getTimeZone("GMT+08:00"));
4     Date theDate = new Date(epoch);
5     return (sdf.format(theDate));
6 }
```

3. 获取呼叫类型描述

代码 12-23 是获取呼叫类型描述的主要代码。

代码 12-23 获取呼叫类型描述

文件名: FooTelUtil.java

```
1 public String getCallTypeDesc(int type) {
2     switch(type) { case CallLog.Calls.INCOMING_TYPE: { return ("呼入"); }
3                   case CallLog.Calls.MISSED_TYPE: { return ("未接"); }
4                   case CallLog.Calls.OUTGOING_TYPE: { return ("呼出"); }
5     }
6     return ("未知类型");
7 }
```

表 12-10 是 Android 平台所定义的电话呼叫类型的说明，这些类型在呼叫日志接口中进行定义。

表 12-10 呼叫类型的说明

类 型	说 明
INCOMING_TYPE	呼入类型
OUTGOING_TYPE	呼出类型
MISSED_TYPE	未接类型

4. 使用许可

为了获取呼叫日志，必须具备读取联系方式的权限，即要在程序清单中声明读取联系方式的使用许可，其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
```


第 13 章 XML 应用

本章对 Android 平台支持的 XML 应用方式进行重点说明，其主要内容包括 XML Pull API 以及资源解析（布局、菜单、常规 XML 资源和 XML 原文件）。

13.1 Android 平台对 XML 应用的支持

不知读者在编写第一个 Android 程序时，是否已经惊叹过布局资源文件定义的高深莫测？开发者只需在布局资源文件中添加一个“<Button>”标记块，程序界面中就会多出一个按钮组件。从寥寥几行的 XML 标记到一个“活生生”的按钮组件实例，读者可以想象出 Android 平台根据这几行 XML 标记做了多少事情！

如同资源是程序的基石，XML 技术是 Android 平台的应用基础。Android 平台中主要提供了 4 种使用 XML 的方式：XML Pull API、XML 资源解析、SAX 和 DOM。实际上，各种处理方式有各自的局限性和适用性，希望读者在下面的介绍中仔细体会。

由于 SAX 和 DOM 属于桌面平台的解析方式，在移动平台运用并不是很广泛，所以本书将不对 SAX 和 DOM 两种方式进行过多介绍。

13.2 XML Pull API

XML Pull 解析器是一款高效易用的 XML 解析器，其与 SAX 都是基于流（Stream）处理的解析器。SAX 采用的是“推”的解析方式（通过解析事件来推动解析动作），而 XML Pull 采用的是“拉”的方式，从 XML 流（文件流或字符流）中“拉”出标记内容。XML Pull API 的官方网站是 <http://www.xmlpull.org/>。

13.2.1 Android 平台对 XML Pull API 的支持

Android 平台提供了 XML Pull 包（org.xmlpull.v1）用于 XML Pull 解析应用。表 13-1 是该包中重要类/接口的说明。

表 13-1 XML Pull 包中重要类/接口的说明

类/接口	说 明
XmlPullParserFactory	解析器工厂，用于创建解析器
XmlPullParser	解析器，用于对 XML 文档进行解析

13.2.2 XML Pull API 使用模式

XML Pull API 的使用模式大致如下：

- 1) 获取 XML Pull 解析器工厂（XmlPullParserFactory）实例。



- 2) 借助工厂实例创建一个 XML Pull 解析器 (XmlPullParser)。
- 3) 设置 XML Pull 解析器的输入内容。
- 4) 通过 XML Pull 解析器的有关方法进行解析。
- 5) 将解析结果提供给可视界面进行展示。

13.2.3 XML Pull API 应用示例

在即将介绍的 XML Pull API 应用示例中，通过解析一个包含国家、省和城市信息的 XML 文档，将其中的记录通过微调控制器进行显示，如图 13-1 所示。



图 13-1 Xml Pull API 解析的实机界面

1. XML 文档内容结构

代码 13-1 是图 13-1 所示 XML 文档的内容片段。

代码 13-1 XML 文档的内容片段

文件名: cities.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <Country>
3      <Province Name="安徽" Code="34">
4          <City Name="安庆" Code="8"/>
5          <City Name="蚌埠" Code="3"/>
6          .....
7      </Province>
8      <Province Name="北京" Code="11"/>
9      .....
10 </Country>
```

在代码 13-1 中，该文档内容的节点层次为国家 (Country) → 省 (Province) → 城市 (City)。国家节点为根节点，其包含多个省节点；而省节点包含多个城市节点，城市为叶子节点。该文档中的内容结构是按照树状形式进行组织的。

在图 13-1 中，使用了两个微调控制器 (Spinner) 来分别显示省和城市的记录项。微调控制器属于适配器视图，其需要绑定适配器；而适配器需要与数据集关联 (有关适配器视图的应用可以参考第 4 章)。对于数据集的来源，可以考虑将 XML 中的节点映射为对应的记录对象，在解析过程中根据节点类型来创建记录对象并添加到记录集中。

2. XML 节点映射对象

(1) 国家节点对象

代码 13-2 是国家节点映射的对象定义，该对象中包含了一个省对象容器。

代码 13-2 国家节点对象的定义

文件名: Country.java

```

1 public class Country {
2     private ArrayList<Province> provinces = null;
3
4     public Country() { this.provinces = new ArrayList<Province>(); }
5
6     public void addProvince(Province p) { this.provinces.add(p); }
7     public ArrayList<Province> getProvinces() { return this.provinces; }
8 };

```

(2) 省节点对象

代码 13-3 是省节点映射的对象定义, 该对象中包含了一个城市对象容器。

代码 13-3 省节点对象的定义

文件名: Province.java

```

1 public class Province implements Parcelable {
2     public static final String TAG_NAME = "Province";
3
4     private String name = null;
5     private String code = null;
6     private ArrayList<City> cities = null;
7
8     //必须要有一个名为 CREATOR 的成员对象, 否则无法进行 Parcelable 对象通信
9     public static final Parcelable.Creator<Province> CREATOR =
10         new Parcelable.Creator<Province>() {
11             public Province createFromParcel(Parcel in) { return new Province(in); }
12             public Province[] newArray(int size) { return new Province[size]; }
13         };
14
15     //实现 Parcelable 接口
16     public Province(Parcel in) { this.name = in.readString();
17                               this.code = in.readString();
18     }
19
20     public Province(String name, String code) { this.name = name; this.code = code;
21         cities = new ArrayList<City>();
22     }
23
24     public String getName() { if(this.name == null) { return (" "); }
25         return (this.name);
26     }
27
28     public String getCode() { return (this.code); }
29     public void addCity(City city) { this.cities.add(city); }

```



Android 平台开发之旅 第2版

```

30     public ArrayList<City> getCities() { return this.cities; }
31
32     @Override
33     public String toString() { return (getName() ); }
34     @Override
35     public int describeContents() { return 0; }
36     @Override
37     public void writeToParcel(Parcel dest, int flags) {     dest.writeString(this.name);
38                                                         dest.writeString(this.code);
39     }
40 };

```

在代码 13-3 中，因为省节点对象会通过主 Activity 传递给显示城市详情的子 Activity，所以省节点对象必须实现 Parcelable 接口（有关 Parcelable 接口的定义约定参考第 3 章）。

（3）城市节点对象

代码 13-4 是城市节点映射对象的定义。

代码 13-4 城市节点对象的定义

文件名：City.java

```

1  public class City implements Parcelable {
2      public static final String TAG_NAME = "City";
3
4      private String name = null;
5      private String code = null;
6
7      //必须要有一个名为 CREATOR 的成员对象，否则无法进行 Parcelable 对象通信
8      public static final Parcelable.Creator<City> CREATOR =
9          new Parcelable.Creator<City>() {
10              public City createFromParcel(Parcel in) { return new City(in); }
11              public City[] newArray(int size) { return new City[size]; }
12          };
13
14      //实现 Parcelable 接口
15      public City(Parcel in) {     this.name = in.readString();
16                                  this.code = in.readString();
17      }
18
19      public City(String name, String code) {     this.name=name; this.code=code; }
20
21      //获取城市名称
22      public String getName() {
23          if(this.name == null) { return (" "); }
24          return (this.name);
25      }
26      //获取代码
27      public String getCode() { return (this.code); }

```



```

28
29     @Override
30     public String toString() { return (getName() ); }
31     @Override
32     public int describeContents() { return 0; }
33     @Override
34     public void writeToParcel(Parcel dest, int flags) {     dest.writeString(this.name);
35                                                         dest.writeString(this.code);
36     }
37 };

```

3. 应用程序主 Activity 框架

代码 13-5 是 XML Pull API 解析示例程序的 Activity 组件的框架定义。

代码 13-5 XML Pull API 解析示例程序的 Activity 组件的框架定义

文件名: XmlPullAct.java

```

1  public class XmlPullAct extends Activity implements OnItemSelectedListener {
2      private Spinner mSpnProvince = null;
3      private Spinner mSpnCity = null;
4      //适配器容器
5      private ArrayList<Province> mProvinces = new ArrayList<Province>();
6      private ArrayList<City> mCities = new ArrayList<City>();
7      private Province mCurProvince = null;
8
9      @Override
10     public void onCreate(Bundle savedInstanceState) {
11         super.onCreate(savedInstanceState);
12         setContentView(R.layout.main_view);
13         //获取显示组件
14         this.mSpnProvince = (Spinner)findViewById(R.id.spn_province);
15         this.mSpnCity = (Spinner)findViewById(R.id.spn_city);
16         //设置省/直辖市项微调控件
17         ArrayAdapter<Province> adapter1 = new ArrayAdapter<Province>(this,
18             android.R.layout.simple_spinner_item, mProvinces);
19         adapter1.setDropDownViewResource(
20             android.R.layout.simple_spinner_dropdown_item);
21         this.mSpnProvince.setAdapter(adapter1);
22         //城市项微调控件
23         ArrayAdapter<City> adapter2 = new ArrayAdapter<City>(this,
24             android.R.layout.simple_spinner_item, mCities);
25         adapter2.setDropDownViewResource(
26             android.R.layout.simple_spinner_dropdown_item);
27         this.mSpnCity.setAdapter(adapter2);
28         //设置侦听器
29         this.mSpnProvince.setOnItemSelectedListener(this);
30         this.mSpnCity.setOnItemSelectedListener(this);

```



Android 平台开发之旅 第2版

```

31      //初始化数据集
32      initDataSet();
33  }
34  .....
35  };

```

在代码 13-5 中, Activity 在初始化过程中为两个微调控制器设置适配器 (第 21 行和第 27 行), 并初始化适配器所关联的数据集, 其数据内容分别为省列表和省对应的城市列表。

4. XML Pull API 解析框架

代码 13-6 是代码 13-5 中使用 XML Pull API 解析 XML 文档的关键代码。

代码 13-6 使用 XML Pull API 解析 XML 文档

文件名: XmlPullAct.java

```

1  private void initDataSet() {
2      try {   XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
3              factory.setNamespaceAware(true);
4              XmlPullParser parser = factory.newPullParser();
5              //设置输入流
6              InputStream is = getResources().openRawResource(R.raw.cities);
7              parser.setInput(is, "utf-8");
8              doParse(parser);
9              //关闭输入流
10             is.close();
11         } catch (XmlPullParserException e) { e.printStackTrace(); }
12         catch (FileNotFoundException e) { e.printStackTrace(); }
13         catch (IOException e) { e.printStackTrace(); }
14     }

```

在代码 13-6 中, 首先获取一个新的 XML Pull 解析器工厂实例 (第 2 行) 并设置其对命名空间敏感 (第 3 行); 继而使用工厂示例的 newPullParser 方法获取一个新的解析器 (第 4 行); 最后使用解析器的 setInput 方法来指定输入流和编码方式 (第 7 行) 即可执行 XML 文档的解析。

5. 分派解析事件

代码 13-7 是 XML Pull 解析器分派解析事件的关键代码。

代码 13-7 分派解析事件

文件名: XmlPullAct.java

```

1  private void doParse(XmlPullParser parser) throws XmlPullParserException, IOException {
2      int eventType = parser.getEventType();
3      boolean finished = false;
4
5      while (!finished) {
6          switch(eventType) { //解析事件分派
7              case XmlPullParser.END_DOCUMENT: { finished=true; }
8              case XmlPullParser.END_TAG: { break; }
9              case XmlPullParser.START_DOCUMENT: { break; }

```



```

10         case XmlPullParser.START_TAG: { parseTag(parser); break; }
11         case XmlPullParser.COMMENT: { break; }
12         case XmlPullParser.TEXT: { break; }
13     }
14     eventType = parser.nextToken();
15 }
16 //设置城市数据集
17 mCities.addAll(mProvinces.get(0).getCities());
18 //通知数据集改变
19 notifyAllDataSet();
20 }
21
22 private void parseTag(XmlPullParser parser) { //解析标记
23     String tagName = parser.getName();
24
25     if(tagName.compareToIgnoreCase(Province.TAG_NAME)==0) {
26         Province pro = new Province(parser.getAttributeValue(null,"Name"),
27                                     parser.getAttributeValue(null,"Code"));
28         mProvinces.add(pro);
29         //记录当前省/直辖市
30         mCurProvince = pro;
31     } else if(tagName.equalsIgnoreCase(City.TAG_NAME)) { //城市节点
32         City c = new City(parser.getAttributeValue(null,"Name"),
33                           parser.getAttributeValue(null,"Code"));
34         mCurProvince.addCity(c);
35     }
36 }

```

在代码 13-7 中，首先使用解析器的 `getEventType` 方法获取当前事件类型（第 2 行），并以此进行分派处理（第 7~12 行）。当前一事件处理完毕后，再使用 `nextToken` 方法获取后一事件类型进行处理（第 14 行），直至文档结束事件（第 7 行）。

6. 列表项选择改变事件响应

当用户选择省微调控制器中的项时，对应的城市需要同步更新，如图 13-2 所示。



图 13-2 微调控制器中数据内容



Android 平台开发之旅 第2版

在代码 13-5 中, 微调控制器使用 `setOnItemSelectedListener` 方法设置对条目选择事件的侦听 (第 29 行和第 30 行)。代码 13-8 是微调控制器条目选择事件的回调处理。

代码 13-8 微调控制器条目选择事件的回调

文件名: XmlPullAct.java

```

1  @Override
2  public void onItemSelected(AdapterView<?> parent, View v, int pos, long id) {
3      if(pos==0) { return; }
4      switch(parent.getId() ) {
5          case R.id.spn_province: { doProvinceSelected(pos); break; }
6          case R.id.spn_city: { doCitySelected(pos); break; }
7      }
8  }
9
10 //选择省/直辖市
11 private void doProvinceSelected(int pos) {
12     //清除数据集
13     mCities.clear();
14     //设置数据集
15     mCities.addAll(mProvinces.get(pos).getCities() );
16     //通知数据集改变
17     ((ArrayAdapter)this.mSpnCity.getAdapter()).notifyDataSetChanged();
18 }
19
20 //选择城市
21 private void doCitySelected(int pos) {
22     int pPos = this.mSpnProvince.getSelectedItemId();
23     //获取当前省/直辖市、城市信息
24     Province province = mProvinces.get(pPos);
25     City city = mCities.get(pos);
26     //通过详情 Activity 组件显示
27     Intent intentStart = new Intent(this, CityInfoAct.class);
28     intentStart.putExtra(City.TAG_NAME, city);
29     intentStart.putExtra(Province.TAG_NAME, province);
30     this.startActivity(intentStart);
31 }

```

在代码 13-8 中, 按照目标组件的 ID 来分别处理 (第 4 行): 如果当前选择的是省条目 (第 5 行), 则获取该省所包含的城市对象集, 并填充到城市微调控制器的适配器数据集中 (第 15 行); 如果当前选择的是城市条目 (第 6 行), 则会获取该城市对象及所属的省对象 (第 24 行), 一起传递给城市详情 Activity 组件 (第 27~30 行)。

在查看城市详情的 Activity 组件中, 将会从意向对象的扩展空间中获取省和城市对象, 并显示其内容信息, 如图 13-1 所示。

13.3 XML 资源解析

Android 系统为 XML 资源提供了多种解析方式。常见的 XML 资源类型包括布局资源、菜单资源、常规 XML 资源和原文件资源。表 13-2 是对这些 XML 资源类型的说明。

表 13-2 XML 资源类型的说明

类 型	说 明
布局资源	用于定义界面布局，存放于“layout”子文件夹
菜单资源	用于定义程序菜单，存放于“menu”子文件夹
常规 XML 资源	定制资源文档，存放于“xml”子文件夹中
XML 原文件	定制资源文档，存放于“raw”子文件夹中

图 13-3 所示为工程中有 4 个相同名称但文件夹不同的 XML 资源文件（sample.xml）。



图 13-3 多种类型的 XML 资源文件

13.3.1 应用程序主 Activity 框架

代码 13-9 是该示例程序的主 Activity 组件的框架定义。

代码 13-9 示例程序的主 Activity 组件的框架定义

文件名: XmlResDemoAct.java

```
1 public class XmlResDemoAct extends Activity implements OnClickListener {
2     //主线程消息队列处理器
3     private Handler mHandler = null;
4
5     @Override
```



Android 平台开发之旅 第2版

```

6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //获取按钮控件并设置单击事件侦听器
10         Button btnGeneral = (Button)findViewById(R.id.btn_general);
11         Button btnRaw = (Button)findViewById(R.id.btn_raw);
12         btnGeneral.setOnClickListener(this);
13         btnRaw.setOnClickListener(this);
14         mHandler = new Handler() { //初始化线程消息队列处理器
15             @Override
16             public void handleMessage(Message msg) {
17                 Bundle bundle = msg.getData();
18                 String data = bundle.getString(IConfig.EXTRA);
19                 println(data);
20             }
21         };
22     }
23     @Override
24     public void onClick(View v) { //按钮单击事件回调
25         switch(v.getId()) {    case R.id.btn_raw: { parseRawXml(); break; }
26                               case R.id.btn_general: { parseGenericXml(); break; }
27         }
28     }
29     .....
30 };

```

在代码 13-9 中，该 Activity 组件初始化了一个主线程消息队列处理器（第 14 行），用于接收外部组件传递过来的消息。

代码 13-10 是该示例程序中配置信息接口的定义，该接口用于统一管理配置信息。

代码 13-10 配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String EXTRA = "_data";
3      public static final String RAW_TAG = "Player";
4      public static final String RAW_ATT1 = "Name";
5      public static final String RAW_ATT2 = "Score";
6      public static final String XML_TAG = "Language";
7      public static final String XML_ATT1 = "LCID";
8      public static final String XML_ATT2 = "Name";
9      public static final String XML_ATT3 = "Code";
10 };

```

13.3.2 解析菜单资源

图 13-4 所示为示例程序的选项菜单，该菜单项会触发显示关于对话框，如图 13-5 所示。

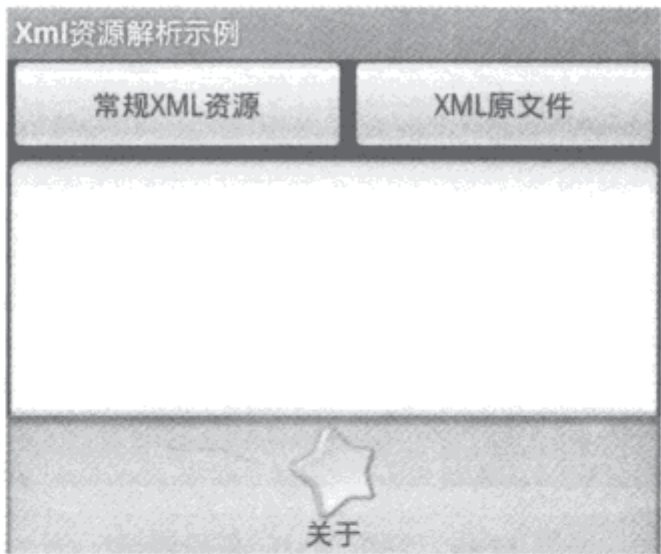


图 13-4 示例程序的选项菜单

1. 菜单资源定义

代码 13-11 是选项菜单资源的定义，该菜单只包含一个“关于”菜单项。

代码 13-11 选项菜单资源的定义

文件名：menu/sample.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/android">
3      <item android:id="@+id/mi_about" android:title="关于"
4          android:icon="@drawable/about"/>
5  </menu>
```

2. 解析菜单资源

代码 13-12 是对代码 13-11 所定义的菜单资源进行解析的主要代码。

代码 13-12 解析菜单资源

文件名：XmlResDemoAct.java

```
1  @Override
2  public boolean onCreateOptionsMenu(Menu menu) {
3      this.getMenuInflater().inflate(R.menu.sample, menu);
4      return super.onCreateOptionsMenu(menu);
5  }
6  @Override
7  public boolean onOptionsItemSelected(MenuItem item) {
8      if(item.getItemId()==R.id.mi_about) { inflateLayout(); }
9      return super.onOptionsItemSelected(item);
10 }
```

在代码 13-12 中，Activity 组件使用菜单填充器（MenuInflater）的 inflate 方法来填充选项菜单资源。当用户单击手机的“菜单”按钮时，Activity 组件会填充菜单资源并显示。

13.3.3 解析 XML 布局资源

图 13-5 所示为应用程序动态解析 XML 布局资源并以可视界面进行展示的效果。

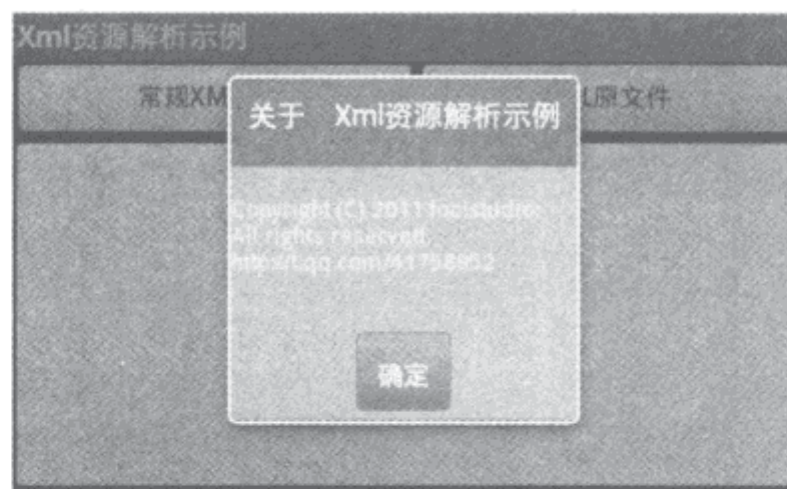


图 13-5 解析 XML 布局资源

1. XML 布局资源

代码 13-13 是该 XML 布局资源的定义，该布局为垂直方向的线性布局，只包含一个文本视图和一个按钮。

代码 13-13 XML 布局资源的定义

文件名: layout/sample.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical" android:background="#6699aa"
4      android:layout_width="fill_parent" android:layout_height="fill_parent">
5      <TextView android:text="@string/about"
6          android:layout_width="fill_parent" android:layout_height="100sp"/>
7      <Button android:id="@+id/btn_ok" android:text="确定"
8          android:layout_width="wrap_content" android:layout_height="wrap_content"
9          android:layout_gravity="center_horizontal" android:padding="16sp"/>
10 </LinearLayout>

```

2. 解析 XML 布局资源

代码 13-14 是对代码 13-13 所定义的布局进行填充的主要代码。

代码 13-14 解析 XML 布局资源并填充

文件名: XmlResDemoAct.java

```

1  private void inflateLayout() { //解析布局 XML 文件
2      //通过布局填充器来填充所定义的布局 XML
3      View v = this.getLayoutInflater().inflate(R.layout.sample, null);
4      Button btnOk = (Button)v.findViewById(R.id.btn_ok);
5      //创建对话框并设定其内容视图
6      final Dialog dlg = createDialogBy(v);
7      btnOk.setOnClickListener(new OnClickListener() {
8          @Override
9          public void onClick(View v) { dlg.dismiss(); }
10     });
11     dlg.show();

```



```

12 }
13
14 //使用指定视图来创建对话框
15 private Dialog createDialogBy(View v) { Dialog dlg = new Dialog(this);
16     dlg.setTitle("关于"+this.getResources().getString(R.string.app_name));
17     dlg.setCancelable(false);
18     //设置对话框内容视图
19     dlg setContentView(v);
20     return (dlg);
21 }

```

在代码 13-14 中，使用布局填充器（LayoutInflater）的 `inflate` 方法对布局资源进行填充，并获取布局资源所定义的根视图的对象实例（第 3 行），最后将该视图实例设置为对话框的内容视图（第 19 行）。当对话框显示时，其界面即为布局资源所定义的内容。

有关资源填充的解释可参考第 2 章，读者可以简单地理解为：对 XML 布局的解析和对 XML 标记的实例化。

13.3.4 解析 XML 资源

图 13-6 所示为应用程序解析 XML 资源并将解析内容输入到文本组件中。

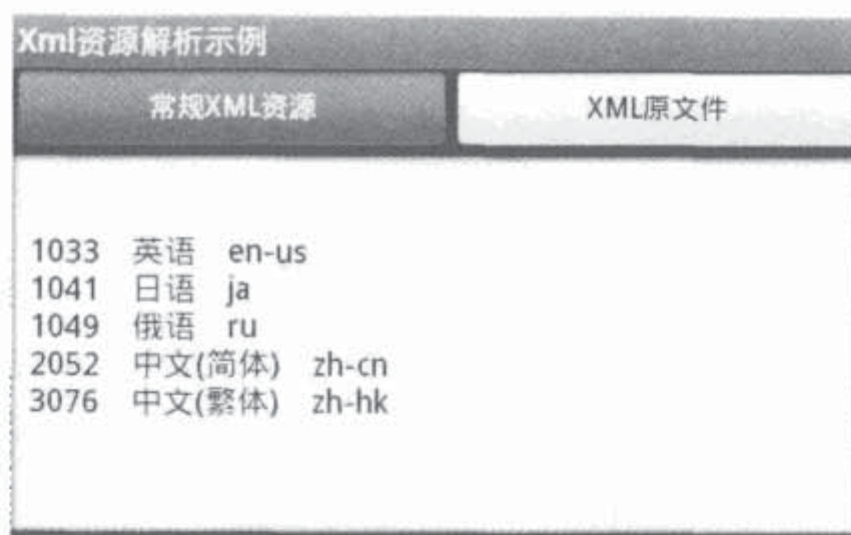


图 13-6 解析并输出 XML 资源文件

1. XML 资源文件

代码 13-15 是图 13-6 所示的 XML 资源文件的定义内容。

代码 13-15 XML 资源文件

文件名: xml/sample.xml

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <Languages>
3     <Language LCID="1033" Name="英语" Code="en-us"/>
4     <Language LCID="1041" Name="日语" Code="ja"/>
5     <Language LCID="1049" Name="俄语" Code="ru"/>
6     <Language LCID="2052" Name="中文(简体)" Code="zh-cn"/>
7     <Language LCID="3076" Name="中文(繁体)" Code="zh-hk"/>

```



2. 解析 XML 资源文件

代码 13-16 是解析代码 13-15 中的 XML 资源文件的主要代码。

代码 13-16 解析 XML 资源文件

文件名: XmlResDemoAct.java

```

1  private void parseGenericXml() { //解析常规 XML 资源
2      //获取 XML 资源解析器
3      XmlResourceParser parser = this.getResources().getXml(R.xml.sample);
4      //使用解析器解析
5      try { new FooXmlParseUtil(parser, mHandler).parse();
6          } catch (XmlPullParserException e) { e.printStackTrace(); }
7          catch (IOException e) { e.printStackTrace(); }
8  }

```

在代码 13-16 中, 使用资源管理器的 `getXml` 方法可获取 XML 资源解析器 (第 3 行), 再通过定制的工具类来包装解析器并完成解析过程 (第 5 行)。

代码 13-17 是定制的解析工具类 (`FooXmlParseUtil`) 的定义。

代码 13-17 XML 解析工具类的定义

文件名: FooXmlParseUtil.java

```

1  public class FooXmlParseUtil {
2      //资源解析器
3      private XmlResourceParser mParser = null;
4      //主线程消息队列处理器
5      private Handler mHandler = null;
6
7      public FooXmlParseUtil(XmlResourceParser parser, Handler handler) {
8          this.mHandler = handler; this.mParser = parser;
9      }
10
11     public void parse() throws XmlPullParserException, IOException {
12         int eventType = mParser.getEventType();
13         boolean finished = false;
14
15         while (!finished) {
16             switch(eventType) { //分派解析事件
17                 case XmlPullParser.END_DOCUMENT: { finished = true; break; }
18                 case XmlPullParser.END_TAG: { break; }
19                 case XmlPullParser.START_DOCUMENT: { break; }
20                 case XmlPullParser.START_TAG: { parserTag(mParser); break; }
21                 case XmlPullParser.COMMENT: { break; }
22                 case XmlPullParser.TEXT: { break; }
23             }

```

```

24         eventType = mParser.nextToken();
25     }
26 }
27
28 private void parserTag(XmlPullParser parser) { //解析标记
29     //获取标记标签
30     final String tagName = parser.getName();
31     if(tagName.compareToIgnoreCase(IConfig.XML_TAG)==0) { //标记判断
32         FooSysUtil.getInstance().sendMsg(mHandler,
33             parser.getAttributeValue(null, IConfig.XML_ATT1)+" "+
34             parser.getAttributeValue(null, IConfig.XML_ATT2)+" "+
35             parser.getAttributeValue(null, IConfig.XML_ATT3));
36     }
37 }
38 };

```

通过代码 13-17，读者可以发现 XML 资源解析器的用法与 XML Pull API 相同。实际上，XML 资源解析器是 XML Pull 解析器的直接子类。

13.3.5 解析 XML 原文件资源

图 13-7 所示为应用程序解析 XML 原文件资源并将其内容输出到文本框。



图 13-7 解析 XML 原文件资源

1. XML 原文件资源

代码 13-18 是图 13-7 中所用到的 XML 原文件资源的定义。

代码 13-18 XML 原文件资源的定义

文件名：raw/sample.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <Players>
3      <Player Name="Sam Zhang" Score="86"/>
4      <Player Name="Rose Li" Score="91"/>
5      <Player Name="Jack Chou" Score="80"/>
6      <Player Name="Andrew Wang" Score="76"/>

```



2. XML 原文件资源的解析

代码 13-19 是对代码 13-18 中的 XML 原文件进行解析的主要代码。

代码 13-19 对 XML 原文件进行解析

文件名: XmlResDemoAct.java

```

1 private void parseRawXml() { //解析 XML 原文件资源
2     InputStream is = this.getResources().openRawResource(R.raw.sample);
3     try { //通过指定的内容处理器来解析 XML
4         Xml.parse(is, Xml.Encoding.UTF_8, new FooContentHandler(mHandler));
5         is.close();
6     } catch (IOException e) { e.printStackTrace(); }
7     catch (SAXException e) { e.printStackTrace(); }
8 }

```

在代码 13-19 中, 首先使用资源管理器的 `openRawResource` 方法来打开 XML 原文件资源的输入流 (第 2 行), 然后通过 `Xml` 工具类的 `parse` 方法来指定内容处理器并启动解析 (第 4 行), 该解析方式类似于 SAX 方式。

3. XML 内容处理器

代码 13-20 是代码 13-19 中提到的 XML 内容处理器的定义。

代码 13-20 XML 内容处理器的定义

文件名: FooContentHandler.java

```

1 public class FooContentHandler implements ContentHandler {
2     //主线程消息队列处理器
3     private Handler mHandler = null;
4     public FooContentHandler(Handler handler) { this.mHandler = handler; }
5
6     @Override
7     public void characters(char[] ch, int start, int length) throws SAXException {}
8     @Override
9     public void endDocument() throws SAXException {}
10    @Override
11    public void endElement(String uri, String locName, String name) throws SAXException {}
12    @Override
13    public void endPrefixMapping(String prefix) throws SAXException {}
14    @Override
15    public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException {}
16    @Override
17    public void processingInstruction(String target, String data) throws SAXException {}
18    @Override
19    public void setDocumentLocator(Locator locator) {}
20    @Override
21    public void skippedEntity(String name) throws SAXException {}

```

```

22      @Override
23      public void startDocument() throws SAXException {}
24      @Override
25      public void startPrefixMapping(String prefix, String uri) throws SAXException {}
26      @Override
27      public void startElement(String uri, String localName, String name, Attributes atts)
28                               throws SAXException {
29          if(!localName.equalsIgnoreCase(IConfig.RAW_TAG)) { return; } //过滤节点
30          //输出分数信息
31          FooSysUtil util = FooSysUtil.getInstance();
32          util.sendMsg(mHandler, "玩家: "+atts.getValue(IConfig.RAW_ATT1)+" "+
33                      "最高得分: "+atts.getValue(IConfig.RAW_ATT2));
34      }
35  };

```

在代码 13-20 中，通过实现内容处理器（ContentHandler）来定制 XML 的内容处理器。在重载的 startElement 方法中对 XML 文档内容进行解析并输出（第 27 行）。

13.4 Android 平台 XML 使用小结

XML 技术是 Android 平台的应用基础，系统提供了多种 XML 的处理方式，不仅有遵循 W3C 规范的方式（DOM），也有业界主流方式（SAX 和 XML Pull API），而且 Android 平台本身还内建了多种对 XML 资源文件的解析方式。

其中，DOM 解析方式会根据 XML 文档的内容和层次结构自动生成面向树状结构的对象模型。该方式的主要特点是使用简单，开发者无需关注其解析过程，但最大的弊端是需要占用较大的内存，特别是当 XML 文档比较庞大时。所以，该方式一般不会对内存资源相对宝贵的嵌入式系统中应用（如 J2ME 平台就只定义了 SAX 而没有定义 DOM）。

SAX 方式和 XML Pull API 在解析过程中都不会占用较大的内存，它们依照文件的顺序进行扫描解析，因为无需保存标记内容和生成结构层次，所以它们的执行效率比 DOM 方式高。SAX 方式是通过解析事件的“推送”方式来进行解析，不断通过标记的解析事件的推送来完成全部的解析过程。SAX 方式需要定义定制的解析事件处理器（Handler），解析的处理在其定义的事件回调函数中进行，所以其使用方法相对 DOM 方式而言是比较复杂的。

XML Pull API 走的是“中庸之道”，它融合了 DOM 方式和 SAX 方式的使用优势。XML Pull API 使用的是基于流的“拔取”方式来进行解析，通过对标记内容的依次拔取来完成全部的解析过程。XML Pull API 无需指定解析事件处理器，其解析过程直接在调用端完成，所以其使用方式要较 SAX 方式直观和简单。在解析效率方面，XML Pull API 又比 DOM 方式高。

XML Pull API 也是 Android 平台进行 XML 解析的使用基础，Android 平台通过 XML Pull API 内建了一些 XML 解析模式，包括对 XML 布局资源的填充和对 XML 资源文件的解析，进一步简化开发代码量。

第14章 地图应用

本章对 Android 平台所提供的地图 API 的功能进行详细的阐述，并通过开发实例详细介绍如何控制地图以及添加地图叠加图等常用功能，同时还对基于地图的定位应用进行实机示例说明。

14.1 地图概述

相信很多读者对电子地图耳熟能详，用户通过电子地图可以很直观地定位自己所在的城市、街道和小区的位置，而且还可以检索周边的服务设施。图 14-1 所示为电子地图的界面。



图 14-1 电子地图的界面

电子地图可理解为一个基于地图数据的位置服务平台，该平台从地图服务器获取地图数据，并进行展现。用户通过该平台提供的视图模式可定位自己的住所以及附近兴趣点（Point of Interest, POI）的位置信息，并根据目标地点的位置进行路线规划。

14.2 Android 平台对地图应用的支持

Android 平台在地图包（com.google.android.maps）中提供了允许应用程序显示和控制地

图展示的接口。表 14-1 是该包中重要的类/接口的说明。

表 14-1 地图包中重要的类/接口的说明

类/接口	说 明
GeoPoint	表示地理位置坐标点（经度和纬度）
ItemizedOverlay	由一串叠加项目组成的叠加图
MapActivity	用于管理地图视图（MapView）显示的 Activity 组件
MapController	用于管理地图偏移后缩放的工具类
MapView	用于显示地图的视图组件
Overlay	表示一块可在地图上显示的叠加图
OverlayItem	重叠项目，是组成 ItemizedOverlay 的基本单元

14.3 地图视图（MapView）

Android 平台提供的地图显示组件是地图视图（MapView），该视图用于显示地图内容。当然，这些地图数据需要从地图服务器那里获取。

地图视图可通过自动获取用户的按键或触摸手势来水平移动和缩放地图，通过程序代码也可控制地图视图，而且还能在地图上绘制一些覆盖图。

图 14-2 所示为通过地图视图显示北美区域的地图信息。



图 14-2 地图视图界面

14.3.1 地图视图组件的定义

地图视图组件的定义方式与一般视图组件相同。代码 14-1 是图 14-2 所示界面的布局



定义。

代码 14-1 地图视图界面的布局定义

文件名: main.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <com.google.android.maps.MapView
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      android:id="@+id/map_view"
5      android:layout_width="fill_parent" android:layout_height="fill_parent"
6      android:clickable="true"
7      android:apiKey="0z_s1QxO7W0oO48ODd0t7DWQrZ3nu2FMbsbXFAG"/>

```

在代码 14-1 中, 使用<com.google.android.maps.MapView>标记来定义地图视图组件, 其形式与定制视图组件的定义一样, 必须指定该组件定义类的完整类名。

为了在地图视图组件中显示地图数据, 开发者必须通过在地图服务中心进行注册来获取地图的 API 的使用密钥。第 7 行的 android:apiKey 属性的内容就是当前地图视图用于访问地图服务的密钥。

14.3.2 获取地图 API 使用密钥

获取地图 API 使用密钥需要满足两个条件: 拥有一个 Google 账号和提供证书的 MD5 指纹。Google 账号可在 Google 公司的官方网站进行免费申请, 而证书的 MD5 指纹需要使用 keytool 工具从调试密钥库文件 (debug.keystore) 中抽取, 如图 14-3 所示。

```

C:\Documents and Settings\Administrator\.android>keytool -list
-keystore "debug.keystore" -storepass android -keypass android

Keystore 类型: JKS
Keystore 提供者: SUN

您的 keystore 包含 1 输入

androiddebugkey, 2011-7-27, PrivateKeyEntry,
认证指纹 (MD5): 68:B3:B9:40:BC:49:66:23:A4:AF:14:6C:45:9C:3C:67

```

图 14-3 提取调试用 MD5 指纹

提示: 当读者成功配置 Android 开发环境后 (参考第 2 章), 在当前用户的工作文件夹的 .android 子文件夹中有一个名为 debug.keystore 的文件, 该文件就是 ADT 插件所提供的调试密钥库文件。

keytool 工具的用法可参考第 16 章。

获取证书的 MD5 指纹后, 还需要登录地图 API 的签名页面 (<http://code.google.com/intl/zh-CN/android/add-ons/google-apis/maps-api-signup.html>), 使用 MD5 指纹获取 API 密钥, 如图 14-4 所示。

当用户单击生成 API 密钥 (Generate API Key) 按钮后, 会转入地图 API 密钥的显示页面, 如图 14-5 所示。

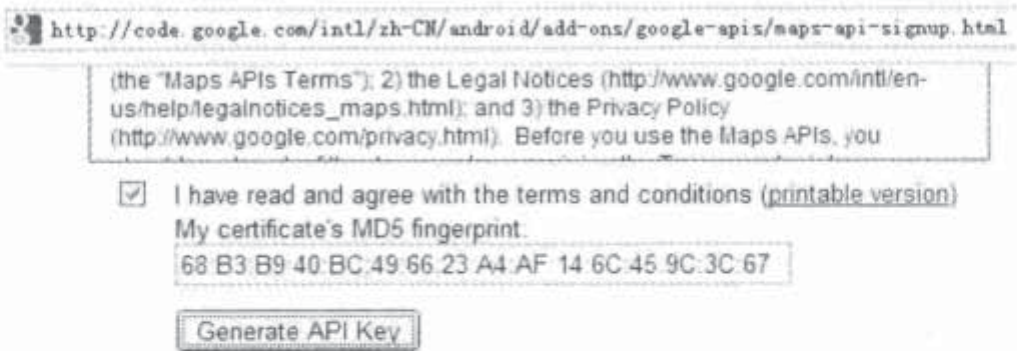


图 14-4 地图 API 的签名页面



图 14-5 地图 API 密钥的显示页面

14.3.3 地图应用工程设置

在创建地图应用有关的工程时，需要将 Google APIs 作为工程的构建目标，如图 14-6 所示。

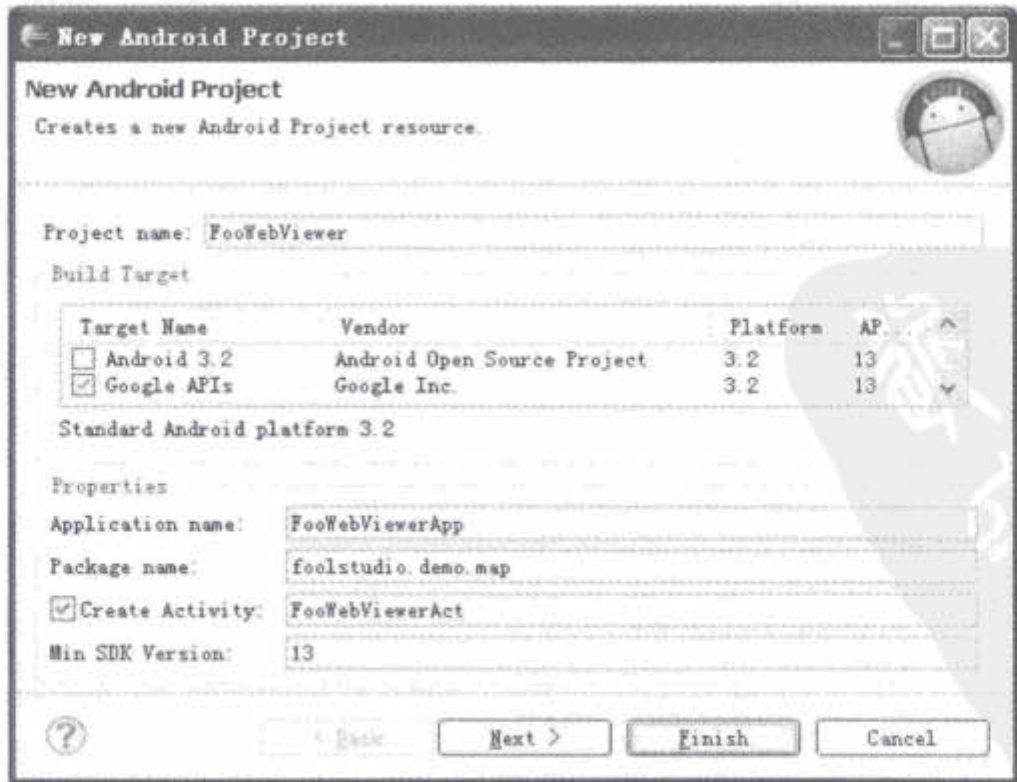


图 14-6 新建地图工程



14.3.4 地图应用程序 Activity 组件

为了正常显示地图视图，用户定义的 Activity 组件必须是一个 MapActivity 实例。因为 Android 平台在 MapActivity 组件中封装了用于访问网络（连接地图服务中心）和文件系统（文件缓存管理）的后台线程，这些线程的生命周期只能在 MapActivity 框架中进行管理。代码 14-2 是图 14-2 所示的程序中 MapActivity 组件的定义。

代码 14-2 MapActivity 组件的定义

文件名: FooMapView.java

```

1  public class FooMapView extends MapActivity {
2      //地图视图组件
3      private MapView mMapView = null;
4
5      @Override
6      public void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9          //获取地图视图组件并设置缩放控件
10         mMapView = (MapView) findViewById(R.id.map_view);
11         mMapView.setBuiltInZoomControls(true);
12     }
13     @Override
14     public boolean isRouteDisplayed() { return (false); }
15 };

```

14.3.5 引用地图库

地图 API 是作为 Android 开发包的附加功能。在默认情况下，应用程序不会自动地引用地图库，所以开发者必须通过程序清单显式地“告诉”系统安装程序引入地图库，其代码如下所示。

文件名: AndroidManifest.xml

```
<uses-library android:name="com.google.android.maps" />
```

14.3.6 地图使用许可

由于地图库中内嵌了网络访问，所以还必须在程序清单中声明允许访问互联网的使用许可，其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.INTERNET"/>
```

14.4 地图应用

正如之前所介绍的，Google 地图是一个基于地图数据的位置服务平台，所以地图的应用

应围绕位置服务的功能特性来开展。

这里所谓的位置服务就是当前流行的 LBS (Location Based Service) 应用。在常见的应用模式中，用户通过移动设备的定位模块 (GPS 模块或者蜂窝网络通信模块) 来获取设备所在的地理位置信息 (经纬度、海拔等信息)，再将这些地理位置信息与电子地图数据进行匹配，从而确定移动设备或用户的环境位置信息 (所在城市、街道等)。图 14-7 所示为手机定位方法的示意图。

在确定主体在电子地图中的位置后，就可
通过电子地图的空间相对信息来获取主体附近的 POI (超市、银行、饭店等) 等地理信息。

电子地图应用系统根据本体位置到目标位置的道路连通信息可为用户规划驾驶路线。在这种应用模式中，开发者不仅要处理电子地图数据的读取和展示，还要控制地图的平移和缩放。更为烦琐的，还要将定位设备获取到的位置信息在电子地图上匹配以及进行线路规划。

通过地图 API，开发者不用关注地图数据的访问和展示以及与地图的匹配，甚至都无需考虑线路规划。对于地图的平移及缩放控制，通过地图视图的内建控件可方便实现。开发者只需要“告诉”地图视图目标位置，视图就会自动地“前往”该位置。在图 14-8 所示的界面中，应用程序方便地将指定位置设置为当前视图的中间点。

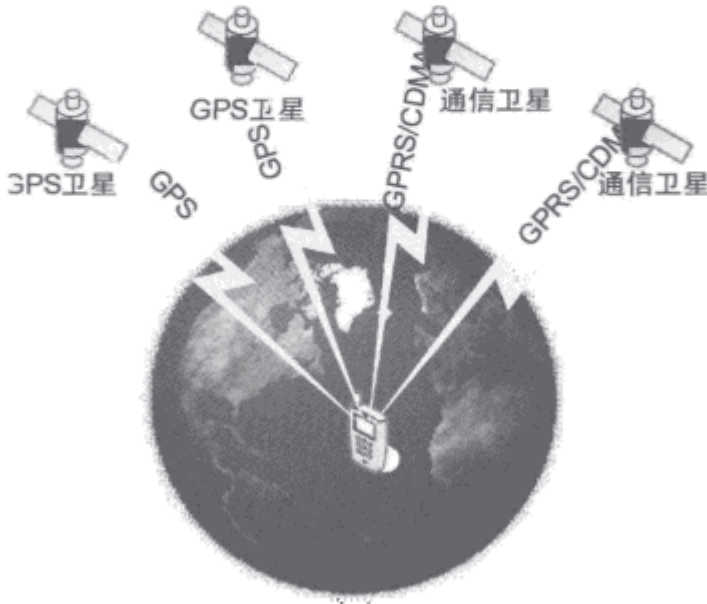


图 14-7 手机定位方法的示意图



图 14-8 地图查看示例



14.4.1 地图 Activity 组件框架

代码 14-3 是图 14-8 所示的地图查看器程序的地图 Activity 组件的框架定义。

代码 14-3 地图查看器程序的地图 Activity 组件的框架定义

文件名: MapViewDemoAct.java

```

1  public class MapViewDemoAct extends MapActivity {
2      //地图视图组件
3      private MapView mMapView = null;
4      private boolean mTrafficMode = true;
5      //叠加列表
6      private List<Overlay> mOverlays = null;
7      private Drawable mDrawable = null;
8
9      @Override
10     public void onCreate(Bundle savedInstanceState) {
11         super.onCreate(savedInstanceState);
12         setContentView(R.layout.main);
13         //设置地图视图
14         mMapView = (MapView) findViewById(R.id.map_view);
15         //设置缩放控件
16         mMapView.setBuiltInZoomControls(true);
17         mMapView.setTraffic(true);
18         //获取叠加管理接口
19         mOverlays = mMapView.getOverlays();
20         mDrawable = this.getResources().getDrawable(R.drawable.flag_red);
21         //移动到指定点
22         GeoPoint point = new GeoPoint(30574925,114404350);
23         mMapView.getController().animateTo(point);
24         //放大等级为【1, 21】
25         mMapView.getController().setZoom(14);
26     }
27     @Override
28     public boolean onCreateOptionsMenu(Menu menu) {
29         this.getMenuInflater().inflate(R.menu.opt_menu, menu);
30         return super.onCreateOptionsMenu(menu);
31     }
32     @Override
33     public boolean onOptionsItemSelected(MenuItem item) {
34         switch(item.getItemId() ) {
35             case R.id.mi_get_loc: { getLocation(); break; } //获取位置
36             case R.id.mi_add_mark: { addMark(); break; } //添加标注
37             case R.id.mi_mode: { setMode(item); break; } //切换模式
38         }
39         return super.onOptionsItemSelected(item);

```



```
40     }
41     .....
42  };
```

在代码 14-3 中，首先通过地图视图资源 ID 来获取地图视图组件实例（第 14 行），然后设置视图组件的缩放控件及地图模式（第 16 行和第 17 行）。在第 19 行，获取地图视图的叠加图管理接口，用于后续标注叠加图的添加；最后将地图视图移动到指定的地理位置（第 23 行），并设置其缩放等级（第 25 行）。

1. 设置缩放控件

在代码 14-3 中，使用地图视图的 `setBuiltInZoomControls` 方法可设置是否需要内建的缩放控件，如图 14-9 所示。

2. 设置地图模式

在代码 14-3 中，使用地图视图的 `setTraffic` 方法可设置地图是否为交通模式。地图视图还有另外一种模式：卫片模式，其显示效果如图 14-10 所示。



图 14-9 地图视图的缩放控件



图 14-10 卫片模式

代码 14-4 是切换地图模式的主要代码。

代码 14-4 地图模式切换

文件名: `MapViewDemoAct.java`

```
1 private void setMode(MenuItem item) { //设置地图模式
2     mTrafficMode = !mTrafficMode;
```




```
3
4     if(mTrafficMode) { mMapView.setSatellite(false); //切换至交通模式
5         mMapView.setTraffic(true);
6         item.setIcon(R.drawable.satellite);
7         item.setTitle(R.string.satellite);
8     } else { mMapView.setTraffic(false); //切换至卫片模式
9         mMapView.setSatellite(true);
10        item.setIcon(R.drawable.traffic);
11        item.setTitle(R.string.traffic);
12    }
13 }
```

在代码 14-4 中，使用地图视图的 `setSatellite` 方法即可设置地图是否为卫片模式。

3. 叠加图管理接口

叠加图（Overlay）表示可在视图表面进行叠加显示的图标。在代码 14-3 中，使用地图视图 `getOverlays` 方法可获其叠加图管理接口（第 19 行），叠加图通过该接口进行管理。对于叠加图的删减处理，只需对容器中列表项进行删减即可，无需与地图视图打交道。

4. 地图控制接口

在代码 14-3 中，地图的移动和缩放都不直接由地图视图来控制，而是通过地图控制器（MapController）来完成。使用地图视图的 `getController` 方法可获取该视图的地图控制器，通过该接口可实现对地图的移动和缩放等控制。

使用地图控制器的 `animateTo` 方法“指示”地图视图滚动到指定的地理位置；使用 `setZoom` 方法来设置地图显示的缩放等级。

14.4.2 获取地图当前位置

图 14-11 所示为显示地图视图中心地理位置信息。



图 14-11 显示地图视图中心地理位置信息

代码 14-5 是显示地图视图中心地理位置信息的主要代码。

代码 14-5 显示地图视图中心地理位置信息

文件名: MapViewDemoAct.java

```
1 //获取当前地图视图中心地理位置信息
2 private void getLocation() {
3     GeoPoint point = mMapView.getMapCenter();
4     Toast.makeText(this, "当前位置: 纬度: "+point.getLatitudeE6()+
5         "\n          经度: "+ point.getLongitudeE6(),
6         Toast.LENGTH_LONG).show();
7 }
```

在代码 14-5 中, 使用地图视图的 `getMapCenter` 方法可获取当前视图的中心点, 再通过该接口的 `getLatitudeE6` 和 `getLongitudeE6` 方法可获得该位置的纬度和经度信息 (该值应该是经过加密处理之后的数值)。

14.4.3 地图视图叠加图管理

前面已经提到, 地图视图的叠加图管理是通过叠加图管理器来完成的。地图视图并不直接使用叠加类来代表视图中的每个叠加图, 而是使用项目化叠加类 (`ItemizedOverlay`) 来表示。而项目化叠加类又可包含多个叠加项 (`OverlayItem`), 每一个叠加项具有详细的地理位置和说明信息。

将叠加图的管理分成项目化叠加类和叠加项可以更加详尽地描述地图平面中某一位置点的信息集合。例如, 项目化叠加类可以表示一栋大厦所有楼层的信息, 而叠加项表示每一层的信息。

1. 在指定地点添加叠加图

代码 14-6 是在指定地理位置添加叠加图的主要代码。

代码 14-6 在指定地理位置添加叠加图

文件名: MapViewDemoAct.java

```
1 //通过图标来创建一个项目化叠加实例
2 FooItemizedOverlay overlay = new FooItemizedOverlay(mDrawable);
3 GeoPoint point = new GeoPoint(30574925,114404350);
4 //通过地理位置点创建叠加条目
5 OverlayItem item = new OverlayItem(point, "东湖", "武汉东湖");
6 //添加叠加项
7 overlay.addItem(item);
8 mOverlays.add(overlay);
9 //发送重绘请求
10 this.mMapView.postInvalidate();
```

在代码 14-6 中, 通过指定图标 (读者可根据 POI 的类型来设置不同的图标) 来创建一个项目化叠加实例 (第 2 行); 再通过地理位置点及简介文字来创建一个叠加项 (第 5 行); 继而使用项目化叠加实例的 `addItem` 方法将叠加项添加到该实例中 (第 7 行), 再使用叠加



项管理接口的 `add` 方法将项目化叠加实例添加到地图视图的叠加项列表中（第 8 行）。最后，通过地图视图发送重绘请求，这样才能保证叠加图的改变能够及时显示出来。

2. 在当前地点添加叠加项

除了可在指定地理位置添加叠加项，还可在地图浏览过程中自由添加叠加项。图 14-12 所示为在当前地图的中心位置添加叠加项。



图 14-12 在当前地图的中心位置添加叠加项

代码 14-7 是在地图中心位置添加叠加项的主要代码。

代码 14-7 在地图中心位置添加叠加项

文件名: `MapViewDemoAct.java`

```
1 //添加叠加点
2 private void addMark() { GeoPoint point = mMapView.getMapCenter();
3     //创建叠加项
4     FooItemizedOverlay overlay = new FooItemizedOverlay(mDrawable);
5     OverlayItem item = new OverlayItem(point, "<title>", "<snippet>");
6     //添加叠加项
7     overlay.addItem(item);
8     mOverlays.add(overlay);
9     //发送重绘请求
10    this.mMapView.postInvalidate();
11 }
```

在代码 14-7 中，使用地图当前地理位置信息来创建叠加条目（第 5 行），并添加到项目化叠加实例中（第 7 行），再将该实例添加到地图视图的叠加项列表中（第 8 行），最后通知

地图视图重绘。

3. 叠加条目定义

代码 14-8 是代码 14-6 中所提到的项目化叠加类的定义。

代码 14-8 项目化叠加类的定义

文件名: FooItemizedOverlay.java

```
1 public class FooItemizedOverlay extends ItemizedOverlay<OverlayItem> {
2     //叠加项列表
3     private ArrayList<OverlayItem> mItems = new ArrayList<OverlayItem>();
4
5     public FooItemizedOverlay(Drawable defaultMarker) {
6         super(boundCenterBottom(defaultMarker));
7     }
8
9     @Override
10    protected OverlayItem createItem(int i) { return (mItems.get(i)); }
11    @Override
12    public int size() { return (mItems.size()); }
13    public void addItem(OverlayItem overlay) { mItems.add(overlay);
14        populate();
15    }
16 };
```

在代码 14-8 中，提供了自定义的公共方法 addItem（第 13 行）来添加叠加项到项目化叠加实例（第 13 行）；使用重载的 createItem 方法（第 10 行）向地图视图提供叠加项列表中的记录。

14.4.4 地图 API 使用小结

1. 地图视图的使用模式

对于地图视图的使用模式，读者可借助 MVC（Model—View—Controller）模式来理解：地图视图就是用于地图展示的视图对象；地图控制器用于对视图的控制；而叠加管理器用于管理叠加图，三者之间的关系如图 14-13 所示。

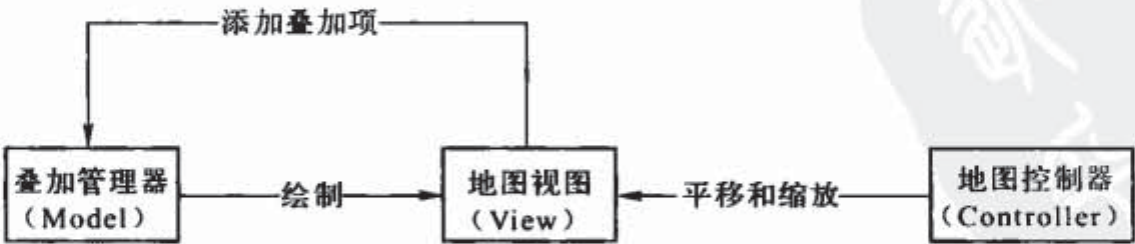


图 14-13 地图视图的使用模式

2. 地图缩放控制

对于地图的缩放，地图控制器定义了 21 个等级，其中等级 1 表示将地图缩小到极限，而等级 21 表示将地图放大到极限。当地图放大到极限时，缩放控件中的放大按钮将不再可



用；同样，当地图已缩小到极限时，缩小按钮也不再可用。

14.5 地图定位

地图呈现了客观世界的地理信息，但有时人们却不知道自己在地图中的哪个位置。当人们在旅途中的时候，经常会自问：我现在在哪里？实际上，一旦人们能够获取自己的位置信息（经纬度），再通过与地图进行匹配，即可知道自己在地图上的哪个点。

图 14-14 所示为通过获取手机当前位置并将地图移动到该位置的实机界面。



图 14-14 获取手机当前位置的实机界面

14.5.1 位置管理

位置管理器（LocationManager）用于访问系统的位置服务，该服务允许应用程序获取设备所在的实时地理位置，其定义在位置包（android.location）中。对于手机而言，常用的定位途径是：GPS 模块和手机网络基站定位。

图 14-15 所示为启动 GPS 并获取位置信息的实机界面。



图 14-15 启动 GPS 并获取位置信息的实机界面

1. 应用程序 Activity 组件框架

代码 14-9 是定位应用示例程序的 Activity 组件的框架定义。

代码 14-9 定位应用示例程序的 Activity 组件的框架定义

文件名: LocatorAct.java

```
1  public class LocatorAct extends Activity {
2      //位置管理器
3      private LocationManager mMgr = null;
4      //位置化侦听器
5      private FooLocationListener mListener = null;
6
7      @Override
8      public void onCreate(Bundle savedInstanceState) {
9          super.onCreate(savedInstanceState);
10         setContentView(R.layout.main);
11         //获取位置管理器实例
12         mMgr = (LocationManager)this.getSystemService(Context.LOCATION_SERVICE);
13         //初始化位置侦听器
14         mListener = new FooLocationListener();
15         enableGPS();
16     }
17
18     private void enableGPS() { //启用 GPS
19         if (!mMgr.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
20             Toast.makeText(this, "请开启 GPS!", Toast.LENGTH_SHORT).show();
21             Intent intent = new Intent(Settings.ACTION_SECURITY_SETTINGS);
22             startActivityForResult(intent, IConfig.REQ_CODE);
23         } else { initLocation(); }
24     }
25
26     @Override
27     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
28         if(requestCode==IConfig.REQ_CODE) { initLocation(); }
29     }
30
31     private void initLocation() { //初始化定位
32         //根据标准获取最佳位置提供者
33         Criteria criteria = new Criteria();
34         criteria.setAccuracy(Criteria.ACCURACY_FINE);
35         criteria.setAltitudeRequired(false);
36         criteria.setBearingRequired(false);
37         criteria.setCostAllowed(true);
38         criteria.setPowerRequirement(Criteria.POWER_MEDIUM);
39         String provider = mMgr.getBestProvider(criteria, true);
40         print("最佳位置提供者: "+provider);
```

**Android 平台开发之旅 第2版**

```

41         //获取最近位置信息
42         Location location = mMgr.getLastKnownLocation(provider);
43         showLocation(location);
44         //请求位置信息更新
45         mMgr.requestLocationUpdates(provider, 0L, 0.0f, mListener);
46     }
47
48     @Override
49     protected void onDestroy() { super.onDestroy();
50         //取消更新请求
51         mMgr.removeUpdates(mListener);
52     }
53     .....
54 };

```

在代码 14-9 中，首先获取位置管理器实例（第 12 行），同时初始化位置侦听器（第 13 行）。由于需要通过 GPS 获取定位信息，所以需要保证 GPS 开启（第 15 行）。

GPS 已经开启或等待 GPS 启动后，即可进行定位初始化：通过条件标准获取系统最佳定位提供者（第 33~39 行），然后请求该提供者的位置信息（第 45 行）。位置信息将通过位置侦听器的回调方法获取。

2. 应用程配置信息接口

代码 14-10 是应用程序中有关配置信息接口的定义，包括 Activity 请求代码。

代码 14-10 应用程序配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final int REQ_CODE = 20110;
3  };

```

3. 获取 GPS 位置信息

在位置侦听器的回调方法中即可获取 GPS 位置信息。代码 14-11 是代码 14-9 中提到的位置侦听器的定义。

代码 14-11 位置侦听器的定义

文件名: LocatorAct.java

```

1  class FooLocationListener implements LocationListener {
2      @Override
3      public void onLocationChanged(Location loc) { showLocation(loc); }
4      @Override
5      public void onProviderDisabled(String provider) {}
6      @Override
7      public void onProviderEnabled(String provider) {}
8      @Override
9      public void onStatusChanged(String provider, int status, Bundle extras) {}

```



```
10  };
11
12  private void showLocation(Location loc) { //显示位置信息
13      if(loc == null) { return; }
14      print("纬度: "+loc.getLatitude()+" "+"经度: "+loc.getLongitude());
15  }
```

在代码 14-11 中，在位置侦听器的 onLocationChanged 方法中即可获得改变后的位置信息（第 3 行），并通过该信息接口获取经纬度值（第 14 行）。

注意：GPS 定位依赖于 GPS 硬件模块对 Android 平台的支持程度，遗憾的是，作者的实机（三星 S5660，Android 2.2.1）无法通过 GPS 模块获取定位信息。

4. 使用许可

为了获取 GPS 的位置信息，需要拥有获取高精度位置信息的使用许可，即必须在程序清单中声明该使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

14.5.2 手机基站定位

有关通过手机网络基站进行定位已在第 12 章中进行介绍，本节主要介绍如何将手机定位与地图应用进行结合。图 14-16 所示为将地图视图转到手机当前位置的实机界面。



图 14-16 手机地图定位

1. 获取手机定位信息

示例程序中通过选项菜单项来获取手机定位信息，其所触发的操作仅仅是注册侦听蜂窝单元位置状态，如代码 14-12 所示。



代码 14-12 选项菜单项触发操作

文件名: MapView2Act.java

```

1 private void goCurPos() { //转入到手机当前位置
2     mTelMgr.listen(mListener, PhoneStateListener.LISTEN_NONE);
3     mTelMgr.listen(mListener, PhoneStateListener.LISTEN_CELL_LOCATION);
4 }

```

2. 主线程消息队列处理器

由手机位置侦听器所获取的位置信息最终将由主线程消息队列处理器发送给主线程。代码 14-13 是主线程消息队列处理器的定义。

代码 14-13 主线程消息队列处理器的定义

文件名: MapView2Act.java

```

1 mHandler = new Handler() { //初始化主线程消息队列处理器
2     @Override
3     public void handleMessage(Message msg) {
4         Bundle data = msg.getData();
5         goPosition(Double.parseDouble(data.getString(IConfig.EXTRA_LAT)),
6                     Double.parseDouble(data.getString(IConfig.EXTRA_LON)));
7     }
8 };

```

在代码 14-13 中, 主线程获取由手机位置查询线程(可参考第 12 章有关手机定位部分)所传入的手机经纬度信息, 并控制地图视图移动到该位置。需要注意的是, 查询线程所传递过来的手机经纬度信息为实际值, 并不是乘以 10^6 的形式。

3. 控制地图视图移动到手机位置

代码 14-14 是代码 14-13 提到的移动地图视图到手机位置的主要代码。

代码 14-14 移动地图视图到手机位置

文件名: MapView2Act.java

```

1 public void goPosition(double lat, double lon) { //移动到指定点
2     //构造地理地点
3     GeoPoint point = new GeoPoint((int)(lat*1000000), (int)(lon*1000000));
4     mMapView.getController().animateTo(point);
5     mMapView.getController().setZoom(17);
6     //取消位置侦听
7     mTelMgr.listen(mListener, PhoneStateListener.LISTEN_NONE);
8 }

```

在代码 14-14 中, 通过手机位置信息(需要乘以 10^6)创建地理地点(第 3 行), 然后通过地图视图的控制方法将地图移动到该地点。

4. 使用许可

获取手机定位信息不仅需要拥有读取电话状态的许可, 还需要访问位置信息的许可。此

外，对于位置区域标识的查询需要访问网络状态并访问互联网，所以必须在应用程序清单文件中对这些使用权限进行声明，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```



第 15 章 系统信息管理

本章对 Android 平台提供的系统信息管理接口进行全面介绍，包括主要的系统服务接口（Activity 管理、提醒管理、通知管理等）以及系统信息管理的应用模式。

15.1 系统服务

这里系统服务的概念和读者平常所理解的服务程序有所不同，这里的系统服务不仅指服务组件，还包括 Android 系统提供的服务功能。所以，它们的使用方式不能与服务组件的调用一概而论，而必须使用系统提供的特定方式来获取想要得到的服务功能接口，使用这些接口来与系统的核心组件“打交道”。这些系统的核心组件包括对所有 Activity 组件进行管理的 Activity 管理器、音量管理器、电源管理器、系统剪贴板管理器……

总而言之，使用上述系统服务接口，开发者可方便地获取系统信息，对系统功能进行应用集成。

15.1.1 Android 系统服务介绍

Android 平台将所有的服务接口都统一由上下文类（Context）来提供。使用上下文类的 getSystemService 方法依据指定的服务字符串标识来获取对应的服务接口。

图 15-1 是 Android 平台的系统服务。



图 15-1 Android 平台的系统服务

表 15-1 是 Android 平台所定义的服务类型的说明。

表 15-1 Android 平台所定义的服务类型的说明

服务类型	说 明
ACCESSIBILITY_SERVICE	辅助功能服务
ACCOUNT_SERVICE	账号服务
ACTIVITY_SERVICE	Activity 组件管理服务
ALARM_SERVICE	提醒服务
AUDIO_SERVICE	音量控制服务
CLIPBOARD_SERVICE	剪贴板服务
CONNECTIVITY_SERVICE	连接管理服务
DEVICE_POLICY_SERVICE	设备策略服务
DOWNLOAD_SERVICE	下载服务
DROPBOX_SERVICE	保管箱服务（调试用）
INPUT_METHOD_SERVICE	输入法服务
KEYGUARD_SERVICE	键盘锁定服务
LAYOUT_INFLATER_SERVICE	布局填充
LOCATION_SERVICE	定位服务
NFC_SERVICE	近距离通信服务
NOTIFICATION_SERVICE	通知服务
POWER_SERVICE	电源管理服务
SEARCH_SERVICE	搜索服务
SENSOR_SERVICE	传感器服务
STORAGE_SERVICE	存储服务
TELEPHONY_SERVICE	电话信息服务
UI_MODE_SERVICE	UI 模式控制服务
USB_SERVICE	USB 设备管理
VIBRATOR_SERVICE	振动管理器服务
WALLPAPER_SERVICE	墙纸服务
WIFI_SERVICE	Wi-Fi 服务
WINDOW_SERVICE	窗体管理服务

15.1.2 Activity 管理

Activity 管理器（ActivityManager）用于对所有运行中的 Activity 组件进行管理，其定义在 android.app 包中。使用 Activity 管理器可获取当前设备的配置信息、内存信息、进程错误状态、近期任务、运行中进程、运行中服务和运行中任务信息。

图 15-2 是使用 Activity 管理器获取 Activity 组件相关信息的实机界面。

1. Activity 管理器

获取 Activity 管理器的代码如下所示。

```
ActivityManager mgr =
    (ActivityManager)(this.getSystemService(Context.ACTIVITY_SERVICE));
```




系统服务应用示例		系统服务应用示例	
ACTIVITY_SERVICE	启动	ACTIVITY_SERVICE	启动
<p>====配置信息：</p> <p>输入方式：未知类型</p> <p>键盘类型：无键键盘</p> <p>导航方式：无导航</p> <p>触屏方式：手指触摸</p> <p>====内存信息：</p> <p>总内存：164872192</p> <p>是否低内存：否</p> <p>内存阈值：16777216</p> <p>====运行中进程信息：</p> <p>PID=10914 名称：foolstudio.demo.sys.collections 级别：前台</p> <p>PID=243 名称：com.diotek.ime.b2b.chn.pure 级别：可见</p> <p>PID=247 名称：com.android.phone 级别：前台</p> <p>PID=180 名称：system 级别：前台</p>		<p>====运行中进程信息：</p> <p>PID=10914 名称：foolstudio.demo.sys.collections 级别：前台</p> <p>PID=243 名称：com.diotek.ime.b2b.chn.pure 级别：可见</p> <p>PID=247 名称：com.android.phone 级别：前台</p> <p>PID=180 名称：system 级别：前台</p> <p>PID=415 名称：com.sec.android.app.samsungapps.una 级别：后台</p> <p>PID=8793 名称：com.svox.pico 级别：后台</p> <p>PID=8780 名称：com.uc.browser 级别：后台</p> <p>PID=7876 名称：com.wssyncmldm 级别：后台</p> <p>PID=7917 名称：com.sec.android.</p>	

图 15-2 Activity 管理信息的实机界面

2. 获取配置信息接口

使用 Activity 管理器的 `getDeviceConfigurationInfo` 方法可获取当前设备的配置信息接口 (`ConfigurationInfo`)，如代码 15-1 所示。

代码 15-1 获取配置信息接口

文件名：ActivityInfo.java

```
1 ConfigurationInfo cfgInfo = mgr.getDeviceConfigurationInfo();
2 .....
3 private String getInfo(ConfigurationInfo info) { //获取配置信息接口
4     StringBuffer sb = new StringBuffer("====配置信息：");
5     sb.append("\n 输入方式： "+
6         FooAndroidUtil.getInstance().getInputDesc(info.reqInputFeatures));
7     sb.append("\n 键盘类型： "+
8         FooAndroidUtil.getInstance().getKeyboardDesc(info.reqKeyboardType));
9     sb.append("\n 导航方式： "+
10        FooAndroidUtil.getInstance().getNaviDesc(info.reqNavigation));
11    sb.append("\n 触屏方式： "+
12        FooAndroidUtil.getInstance().getTouchScreenDesc(info.reqTouchScreen));
13    return (sb.toString() );
14 }
```

在代码 15-1 中，配置信息接口包括输入方式、键盘类型、导航方式和触屏方式。

(1) 输入方式

通过设备配置信息接口的 `reqInputFeatures` 属性可获取当前设备的输入方式。表 15-2 是 Android 平台所定义的输入方式类型的说明。

表 15-2 输入方式类型的说明

类 型	说 明
INPUT_FEATURE_FIVE_WAY_NAV	五向导航键输入
INPUT_FEATURE_HARD_KEYBOARD	硬键盘输入

(2) 键盘类型

通过设备配置信息接口的 reqKeyboardType 属性可获取当前设备的键盘类型。表 15-3 是 Android 平台所定义的键盘类型的说明。

表 15-3 键盘类型的说明

类 型	说 明
KEYBOARD_UNDEFINED	未定义键盘
KEYBOARD_NOKEYS	无键键盘
KEYBOARD_QWERTY	打字机键盘
KEYBOARD_12KEY	十二键盘

(3) 导航方式

通过设备配置信息接口的 reqNavigation 属性可获取当前设备的导航方式。表 15-4 是 Android 平台所定义的导航方式类型的说明。

表 15-4 导航方式类型的说明

类 型	说 明
NAVIGATION_UNDEFINED	未定义导航
NAVIGATION_DPAD	面板导航
NAVIGATION_TRACKBALL	定位球导航
NAVIGATION_WHEEL	滚轮导航
NAVIGATION_NONAV	无导航（触屏方式）

(4) 触屏方式

通过设备配置信息接口的 reqTouchScreen 属性可获取当前设备的触屏方式。表 15-5 是 Android 平台所定义的触屏方式类型的说明。

表 15-5 触屏方式类型的说明

类 型	说 明
TOUCHSCREEN_NOTOUCH	不支持触屏
TOUCHSCREEN_STYLUS	触摸笔
TOUCHSCREEN_FINGER	手指触摸

3. 获取内存信息

使用 Activity 管理器的 getMemoryInfo 方法可获取系统内存信息接口（MemoryInfo），再使用该接口的属性可获取当前内存信息，如代码 15-2 所示。



代码 15-2 获取内存信息

文件名: ActivityInfo.java

```
1  MemoryInfo memInfo = new MemoryInfo();
2  mgr.getMemoryInfo(memInfo);
3  .....
4  private String getInfo(MemoryInfo info) { //获取内存信息
5      StringBuffer sb = new StringBuffer("\n=====内存信息: ");
6      sb.append("\n 总内存: "+info.availMem);
7      sb.append("\n 是否低内存: "+(info.lowMemory?"是":"否"));
8      sb.append("\n 内存阈值: "+info.threshold);
9      return (sb.toString() );
10 }
```

4. 运行中应用程序进程信息

使用 Activity 管理器的 `getRunningAppProcesses` 方法可获取运行中应用程序的进程信息接口 (`RunningAppProcessInfo`), 再通过该接口的属性可获取当前运行中应用程序的进程信息, 如代码 15-3 所示。

代码 15-3 获取运行中应用程序的进程信息

文件名: ActivityInfo.java

```
1  List<ActivityManager.RunningAppProcessInfo> procs = mgr.getRunningAppProcesses();
2  .....
3  private getProcesses(List<RunningAppProcessInfo> procs) { //获取进程信息
4      StringBuffer sb = new StringBuffer("\n=====运行中进程信息: ");
5      for(int i = 0; i < procs.size(); ++i) {
6          RunningAppProcessInfo info = procs.get(i);
7          sb.append("\nPID="+info.pid);
8          sb.append(" 名称: "+info.processName);
9          sb.append(" 级别: "+
10              FooAndroidUtil.getInstance().getImportanceDesc(info.importance));
11      }
12      return (sb.toString() );
13 }
```

对于进程的重要性等级, 在 `ActivityManager.RunningAppProcessInfo` 接口中定义。表 15-6 是 Android 平台所定义的进程重要性等级的说明。

表 15-6 进程重要性等级的说明

标 识	说 明
IMPORTANCE_FOREGROUND	前台进程
IMPORTANCE_VISIBLE	可见进程
IMPORTANCE_SERVICE	服务进程
IMPORTANCE_BACKGROUND	后台进程
IMPORTANCE_EMPTY	空置

5. 运行中服务信息

图 15-3 是使用 Activity 管理器所获取到的运行中服务和任务信息的实机界面。



图 15-3 运行中服务和任务信息的实机界面

使用 Activity 管理器的 `getRunningServices` 方法可获取系统正在运行服务的信息接口 (`RunningServiceInfo`), 再通过该接口的属性可获取运行中服务信息, 如代码 15-4 所示。

代码 15-4 获取运行中服务信息

文件名: ActivityInfo.java

```
1 //获取最多 8 项服务信息
2 List<ActivityManager.RunningServiceInfo> servs = mgr.getRunningServices(8);
3 .....
4 private String getServices(List<RunningServiceInfo> servs) { //获取运行中服务信息
5     StringBuffer sb = new StringBuffer("\n====运行中服务信息: ");
6     for(int i = 0; i < servs.size(); ++i) {
7         RunningServiceInfo info = servs.get(i);
8         sb.append("\n 名称: "+info.service.getShortClassName());
9         sb.append(" PID="+info.pid);
10        sb.append(" 进程: "+info.process);
11    }
12    return (sb.toString());
13 }
```

6. 运行中任务信息

使用 Activity 管理器的 `getRunningTasks` 方法可获取正在运行任务的信息接口 (`RunningTaskInfo`), 再通过该接口的属性可获取当前运行中任务信息, 如代码 15-5 所示。



代码 15-5 获取运行中任务信息

文件名: ActivityInfo.java

```

1 //获取最多 8 项任务信息
2 List<ActivityManager.RunningTaskInfo> runningTasks = mgr.getRunningTasks(8);
3 .....
4 private String getRunningTasks(List<RunningTaskInfo> runningTasks) { //获取运行中任务信息
5     StringBuffer sb = new StringBuffer("\n=====运行中任务信息: \n");
6     for(int i = 0; i < runningTasks.size(); ++i) {
7         RunningTaskInfo info = runningTasks.get(i);
8         sb.append("TID: "+info.id);
9         sb.append(" 名称: "+info.baseActivity.getShortClassName()+"\n");
10    }
11    return (sb.toString());
12 }

```

7. 使用许可

对于获取任务信息, 需要拥有相应的使用许可, 即必须在程序清单中声明获取任务信息的使用许可, 其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.GET_TASKS"/>
```

15.1.3 提醒管理

提醒管理器 (AlarmManager) 用于存取系统提醒服务, 其允许用户自行规划应用程序的运行时点。图 15-4 所示为使用提醒管理器设置单次提醒和重复提醒的实机界面。

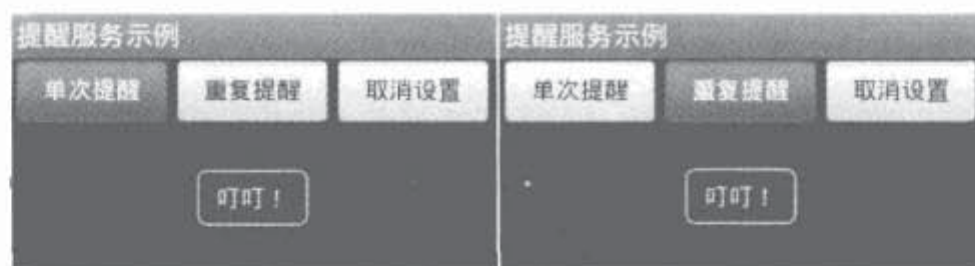


图 15-4 提醒服务应用示例的实机界面

1. 提醒管理器

获取提醒管理器的实例代码如下所示。

```

//获取系统提醒管理器
AlarmManager mMgr = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
//设置该提醒管理器的时区
mMgr.setTimeZone("GMT+08:00");

```

在以上代码中, 为了保证提醒计时的正确性, 需要设置时区信息。

2. 设置提醒任务

提醒管理器可以设置一次性提醒和重复提醒设置。

(1) 一次性提醒

顾名思义, 一次性提醒最多只会提醒一次。代码 15-6 是设置一次性提醒的主要代码。

代码 15-6 设置一次性提醒

文件名: AlarmServiceAct.java

```
1 private void doOnce() {
2     mMgr.set(AlarmManager.RTC_WAKEUP,
3             System.currentTimeMillis() + (5*1000), //触发事件（5 秒之后）
4             mPendingIntent);
5 }
```

在代码 15-6 中，使用提醒管理器的 set 方法来计划一次提醒。该方法包含 3 个参数：第一个参数是提醒时间类型，用以确定提醒的计时和提醒方式。表 15-7 是 Android 平台所定义的提醒时间类型的说明。

表 15-7 提醒时间类型的说明

类 型	说 明
ELAPSED_REALTIME	从系统启动开始计时（包括休眠时间）
ELAPSED_REALTIME_WAKEUP	从系统启动开始计时（包括休眠时间）并唤醒系统
RTC	以系统当前的时间戳计时（UTC 格式）
RTC_WAKEUP	以系统当前的时间戳计时（UTC 格式）并唤醒系统

第二个参数是提醒触发的时点。代码 15-6 中的提醒将在设定 5s 后触发；第三个参数是一个未决意向，用于指明提醒的处理。代码 15-7 是该未决意向的初始化代码。

代码 15-7 设置提醒处理意向

文件名: AlarmServiceAct.java

```
1 Intent intent = new Intent(this, AlarmListener.class);
2 mPendingIntent = PendingIntent.getBroadcast(this, 0, intent, 0);
```

在代码 15-7 中，该提醒的处理由未决意向所包含的意向指定 AlarmListener 来完成。该提醒将以广播的形式执行，所以该提醒的处理者应该是一个广播接收器组件。

(2) 重复性提醒

重复性提醒是指周期进行重复地提醒。使用提醒管理器的 setRepeating 方法可设置重复提醒。该方法包含了 4 个参数，比一次性提醒的 set 方法多了一个提醒的触发间隔参数（第 3 个参数）。以下是设置周期性提醒的实例代码。

```
private void doRepeat() {
    mMgr.setRepeating(AlarmManager.RTC_WAKEUP,
                      System.currentTimeMillis() + (5*1000),
                      (10*1000), //重复间隔
                      mPendingIntent);
}
```

在以上代码中，当该提醒设置后，间隔 5s 将触发第一次提醒，并且从第一次之后每隔 10s 进行一次提醒（除非取消提醒）。

3. 提醒事件侦听

代码 15-8 是代码 15-7 中所提到的提醒事件侦听器的定义。



代码 15-8 提醒事件侦听器的定义

文件名: AlarmListener.java

```
1 public class AlarmListener extends BroadcastReceiver {
2     @Override
3     public void onReceive(Context context, Intent intent) {
4         Toast.makeText(context, "叮叮!", Toast.LENGTH_LONG).show();
5     }
6 };
```

在代码 15-8 中, 所谓的提醒事件侦听器实际上是一个广播接收器。对于广播接收器组件的使用, 需要先进行注册, 这样才能让安装程序“知道”该安装包中有广播接收器组件。广播接收器组件的注册有两种情况 (可参考第 3 章), 在提醒事件侦听器中选择的是在程序清单文件中进行声明。代码 15-9 是在程序清单中声明提醒事件侦听器的主要代码。

代码 15-9 声明提醒事件侦听器

文件名: AndroidManifest.xml

```
<receiver android:label="@string/receiver" android:name=".AlarmListener"/>
```

4. 取消提醒

使用提醒管理器的 `cancel` 方法可取消提醒设置, 该方法仅有的一个参数是设置提醒时的未决意向对象。其代码如下所示。

```
mMgr.cancel(mPendingIntent);
```

5. 使用许可

对于设置提醒管理器的时区, 需要在程序清单中声明设置时区的使用许可, 其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.SET_TIME_ZONE"/>
```

15.1.4 剪贴板管理

剪贴板管理器 (ClipboardManager) 用于访问系统剪贴板, 主要包括设置或获取剪贴板中的内容。图 15-5 所示为获取剪贴板中的内容并输出到文本框中的实例界面。



图 15-5 获取剪贴板中的内容并输出

1. 应用程序配置信息接口

代码 15-10 是剪贴板示例程序中有关配置信息接口的定义。

代码 15-10 配置信息接口的定义

文件名: IConfig.java

```

1  public interface IConfig {
2      public static final String QUERY = "foolstudio";
3      public static final String TEXT = "示例文本";
4      public static final Uri CONTENT_URI = //内容 URI
5                                          Uri.parse("content://foolstudio.demo.FooTblProvider");
6  };

```

2. 剪贴板管理器

获取剪贴板管理器的代码如下所示。

```

ClipboardManager mMgr =
    (ClipboardManager) getSystemService(Activity.CLIPBOARD_SERVICE);

```

3. 设置剪贴板内容

在 Android 3 平台中, 系统剪贴板不仅支持文本, 还包括意向对象及资源定位符 (URI) 等复杂内容。设置剪贴板内容的模式是: 先创建一个剪贴数据 (ClipData), 再使用剪贴板管理器的 setPrimaryClip 方法将该数据设置为主剪贴数据。

(1) 文本内容

使用剪切数据的 newPlainText 方法可以创建一个文本剪贴数据。代码 15-11 是设置文本内容到剪贴板的主要代码。

代码 15-11 设置文本内容到剪贴板

文件名: ClipboardAct.java

```

1  private void doText() {
2      ClipData cd = ClipData.newPlainText("TEXT", IConfig.TEXT);
3      //设置主剪贴内容
4      mMgr.setPrimaryClip(cd);
5      Toast.makeText(this, "设置文本到剪贴板!", Toast.LENGTH_LONG).show();
6  }

```

(2) 资源定位符

使用剪切数据的 newUri 方法可以创建一个资源定位符剪贴数据。代码 15-12 是设置资源定位符到剪贴板的主要代码。

代码 15-12 设置资源定位符到剪贴板

文件名: ClipboardAct.java

```

1  private void doUri() {
2      ClipData cd = ClipData.newUri(this.mResolver, "URI", IConfig.CONTENT_URI);
3      //设置主剪贴内容
4      mMgr.setPrimaryClip(cd);
5      Toast.makeText(this, "设置 URI 到剪贴板!", Toast.LENGTH_LONG).show();
6  }

```

(3) 意向对象

使用剪切数据的 newIntent 方法可以创建一个意向剪贴数据。代码 15-13 是设置意向对



象到剪贴板的主要代码。

代码 15-13 设置意向对象到剪贴板

文件名: ClipboardAct.java

```

1  private void doIntent() {
2      Intent appIntent = new Intent(Intent.ACTION_SEARCH);
3      appIntent.putExtra(SearchManager.QUERY, IConfig.QUERY);
4      ClipData cd = ClipData.newIntent("INTENT", appIntent);
5      //设置主剪贴内容
6      mMgr.setPrimaryClip(cd);
7      Toast.makeText(this, "设置意向对象到剪贴板!", Toast.LENGTH_LONG).show();
8  }

```

在代码 15-13 中, 将一个查询意向设置到剪贴板 (第 2 行), 并设置其查询字符串 (第 3 行)。

4. 获取剪贴板内容

获取剪贴板内容的模式为: 先使用剪贴板管理器的 `getPrimaryClip` 方法获取剪贴板中的主剪贴数据项, 再使用数据项的类型来获取相应的数据内容。

代码 15-14 是获取剪贴板内容的主要代码。

代码 15-14 获取剪贴板内容

文件名: ClipboardAct.java

```

1  private void doPaste() {
2      if(!mMgr.hasPrimaryClip()) { return; }
3      //获取主剪贴数据项
4      ClipData.Item item = mMgr.getPrimaryClip().getItemAt(0);
5      //获取主剪贴数据的描述
6      ClipDescription desc = mMgr.getPrimaryClipDescription();
7      if(desc.hasMimeType(ClipDescription.MIMETYPE_TEXT_PLAIN)) {
8          CharSequence data = item.getText();
9          //粘贴剪贴内容
10         printText(data.toString());
11     } else if(desc.hasMimeType(ClipDescription.MIMETYPE_TEXT_URI_LIST)) {
12         Uri uri = item.getUri();
13         //获取 URI 所指向内容的记录游标
14         Cursor c = mResolver.query(uri, IDbSpec.COLS, null, null, null);
15         String rows = getCursor(c);
16         c.close();
17         //粘贴剪贴内容
18         printText(rows);
19     } else if(desc.hasMimeType(ClipDescription.MIMETYPE_TEXT_INTENT)) {
20         Intent intent = item.getIntent();
21         this.startActivity(intent);
22     }
23 }

```


在代码 15-14 中，先获取主剪贴数据项（第 4 行），再获取主剪贴的描述（第 6 行），再根据描述类型来获取不同的剪贴内容：使用 `getText` 方法来获取文本剪贴项（第 8 行）；使用 `getUri` 方法来获取资源定位符剪贴项（第 12 行）；使用 `getIntent` 方法来获取意向剪贴项（第 20 行）。

对于不同的剪贴内容，将使用不同的方式进行展现：文本内容将直接使用文本视图显示（第 10 行）；资源定位符内容需要使用内容解析器进行查询，获取对应的记录游标（第 14 行，详细内容可参考第 3 章）；意向内容可以直接启动该意向（第 21 行）。

15.1.5 通知管理

通知管理器（`NotificationManager`）用于通知用户有后台事件（如收到新短信、未接电话、备忘提醒）发生。当预期事件发生时，通知管理器会使用 3 种方式来通知用户。

- 1) 在状态栏中会出现持久的图标，用户可以单击该图标查看通知详情。
- 2) 屏幕开启或者闪烁。
- 3) 背景灯闪烁、播放声音或振动。

图 15-6 所示为使用状态栏的图标来通知用户的实机界面。



图 15-6 接收到后台事件的实机界面

1. 通知管理器

获取通知管理器的代码如下所示。

```
String sn = Context.NOTIFICATION_SERVICE;
NotificationManager mMgr = (NotificationManager)this.getSystemService(sn);
```

2. 发送通知

使用通知管理器的 `notify` 方法可发送通知，该方法有两个参数：第一个参数是通知标识，用于区分通知；第二个参数是通知内容，其中包含通知显示和方式。代码 15-15 是发送通知的实例代码。

代码 15-15 发送通知

文件名: `NotificationsAct.java`

```
1 private void doFirst() {
2     Notification notification = makeNotification();
3     notification.defaults |= Notification.DEFAULT_SOUND;
4     notification.defaults |= Notification.DEFAULT_LIGHTS;
5     //发送通知
6     this.mMgr.notify(NotificationManager.NOTIFICATION_ID1, notification);
7 }
```




在代码 15-15 中，首先构建通知内容（第 2 行），然后设置通知提醒的方式（第 3 行和第 4 行），最后使用通知管理器发送通知（第 6 行）。

3. 构建通知内容

从 Android 3 平台开始提供一个通知构建类（Notification.Builder），开发者使用该类可以简化通知构建过程（也可以不使用该类来构建通知）。根据所创建通知的展现方式，将通知的构建分为两种方式。

(1) 方式一（默认内容视图）

代码 15-16 是构建消息内容的主要代码。

代码 15-16 构建消息内容（方式一）

文件名：NotificationsAct.java

```
1 private Notification makeNotification1() { //创建通知（默认内容视图）
2     Notification.Builder builder = new Notification.Builder(this);
3     builder = builder.setSmallIcon(R.drawable.info);
4     builder = builder.setTicker("通知");
5     //5s 后
6     builder = builder.setWhen(System.currentTimeMillis()+5000L);
7     //设置内容意向
8     Intent intent = new Intent(this, ReminderAct.class);
9     PendingIntent contentIntent = PendingIntent.getActivity(this,0,intent,0);
10    builder = builder.setContentIntent(contentIntent);
11    //返回通知对象
12    return (builder.getNotification());
13 }
```

与提醒发送相比，在代码 15-16 中，通知对象的构建也使用了未决意向（第 9 行），因为通知的发送也是一种预期行为，所以也需要使用未决意向。此外，通知的发送也需要明示发送时点。两者之间的差异主要体现在对用户的展现方式：提醒将内容直接呈现给用户；通知内容只有当用户查看通知时才展现。所以，通知存在设置内容意向的环节（第 9 行和第 10 行），该意向为未决意向，当用户查看消息时才触发。

图 15-7 是按照方式一发送通知后系统状态栏的提示界面。

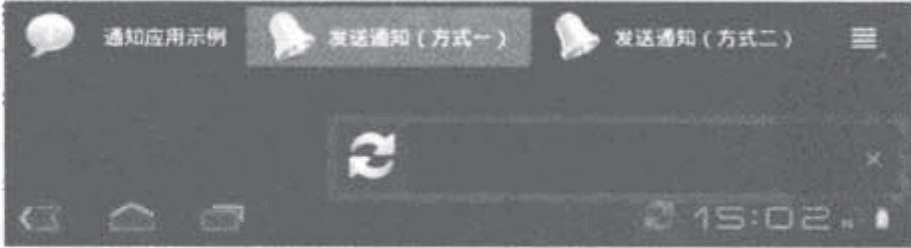


图 15-7 通知提示界面

在图 15-7 中，当用户单击通知图标时，会弹出提示浮动条；再单击浮动条就会触发通知的内容意向来调用 Activity 组件显示通知详情。

(2) 方式二（定制内容视图）

代码 15-17 是构建消息内容的主要代码，与代码 15-16 不同的是其采用指定的内容视图。

代码 15-17 构建消息内容（方式二）

文件名: NotificationsAct.java

```
1 private Notification makeNotification2() { //创建通知（定制内容视图）
2     Notification.Builder builder = new Notification.Builder(this);
3     builder.setSmallIcon(R.drawable.info);
4     builder.setTicker("通知");
5     //5s 后
6     builder.setWhen(System.currentTimeMillis()+5000L);
7     //设置内容意向
8     Intent intent = new Intent(this, ReminderAct.class);
9     PendingIntent contentIntent = PendingIntent.getActivity(this, 0, intent, 0);
10    builder.setContentIntent(contentIntent);
11    //设置内容视图
12    RemoteViews v = new RemoteViews(this.getPackageName(), R.layout.remote);
13    builder.setContent(v);
14    //返回通知对象
15    return (builder.getNotification());
16 }
```

相比方式一，代码 15-17 中主要增加了有关内容视图的设置（第 12 行和第 13 行）。这里所谓的内容视图就是当用户单击系统栏中的消息图标后所弹出的浮动条。图 15-6 所示的消息浮动条就是使用定制的布局界面（第 12 行中 remote 布局资源）。

4. 取消通知

使用通知管理器的 cancel 方法即可取消指定通知的发送，其代码如下所示。

```
this.mMgr.cancel(NOTIFICATION_ID);
```

15.1.6 传感器管理

传感器管理器（Sensormanager）用于访问设备的传感器。图 15-8 所示为使用传感器管理器获取传感器信息的实机界面。

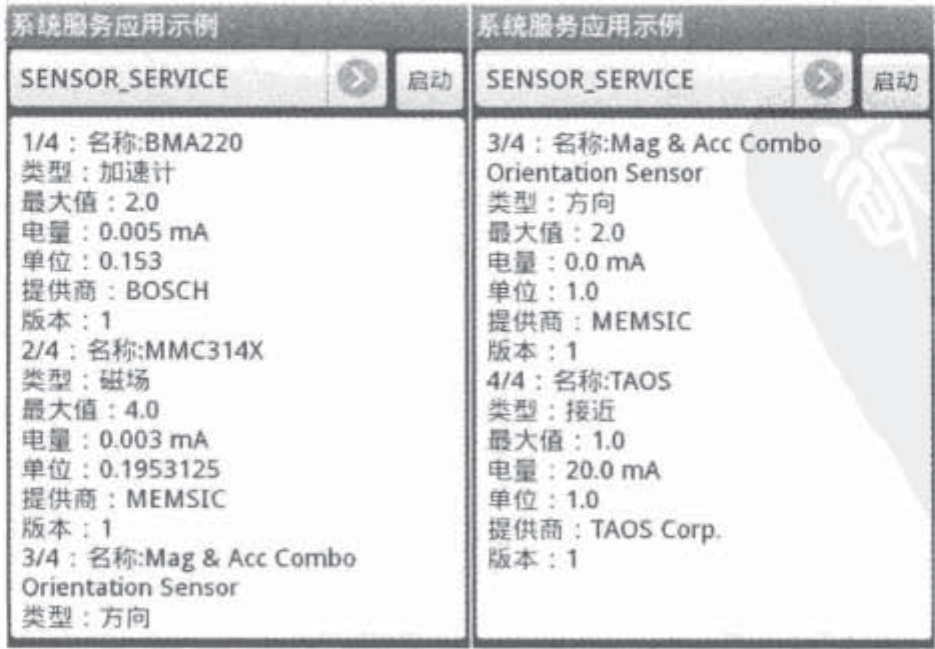


图 15-8 获取传感器信息的实机界面



1. 传感器管理器

获取传感器管理器的代码如下所示。

```
SensorManager mgr =
    (SensorManager)(this.getSystemService(Context.SENSOR_SERVICE));
```

2. 获取传感器

使用传感器管理器的 `getDefaultSensor` 方法可获取指定类型的系统默认传感器；使用 `getSensorList` 方法获取指定类型的所有传感器列表，其实例代码如下所示。

```
List<Sensor> sensors = mgr.getSensorList(Sensor.TYPE_ALL);
for(int i = 0; i < sensors.size(); ++i) {
    sb.append(""+(i+1)+"/"+sensors.size()+"： "+getInfo(sensors.get(i)));
}
```

传感器类型定义在传感器类中。表 15-8 是 Android 平台所定义的传感器类型的说明。

表 15-8 传感器类型的说明

类 型	说 明
TYPE_ACCELEROMETER	加速计
TYPE_ALL	所有传感器类型
TYPE_GYROSCOPE	陀螺仪
TYPE_LIGHT	光传感器
TYPE_MAGNETIC_FIELD	磁场传感器
TYPE_ORIENTATION	方位传感器
TYPE_PRESSURE	压力传感器
TYPE_PROXIMITY	接近传感器
TYPE_TEMPERATURE	温度传感器

3. 传感器信息

使用传感器对象的方法可获得该传感器的相关信息，如代码 15-18 所示。

代码 15-18 获取传感器信息

文件名: SensorInfo.java

```
1 private static String getInfo(Sensor sensor) { //获取传感器信息
2     StringBuffer sb = new StringBuffer("名称:" + sensor.getName());
3     sb.append("\n 类型: "+FooAndroidUtil.getInstance().getSensorDesc(sensor.getType()));
4     sb.append("\n 最大值: " + sensor.getMaximumRange() );
5     sb.append("\n 电量: " + sensor.getPower()+" mA");
6     sb.append("\n 单位: " + sensor.getResolution() );
7     sb.append("\n 提供商: " + sensor.getVendor() );
8     sb.append("\n 版本: " + sensor.getVersion()+"\n");
9     return (sb.toString() );
10 }
```

15.1.7 振动管理器

振动管理器（Vibrator）用于控制设备中的振动器。

1. 振动管理器

获取振动管理器的代码如下所示。

```
Vibrator mgr = (Vibrator)(this.getSystemService(Context.VIBRATOR_SERVICE));
```

2. 设置振动

使用振动管理器的 `vibrate` 方法可设置振动持续的时长并启动振动。

3. 取消振动

使用振动管理器的 `cancel` 方法可取消振动。当程序退出时，所有由该程序启动的振动都将停止。

4. 使用许可

使用振动管理器需要具备相应的使用许可，即必须在程序清单中声明使用振动管理器的使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.VIBRATE"/>
```

15.1.8 墙纸管理

墙纸管理器（`WallpaperManager`）用于访问系统墙纸。图 15-9 所示为在列表中选择图片并设置为系统墙纸的实机界面。

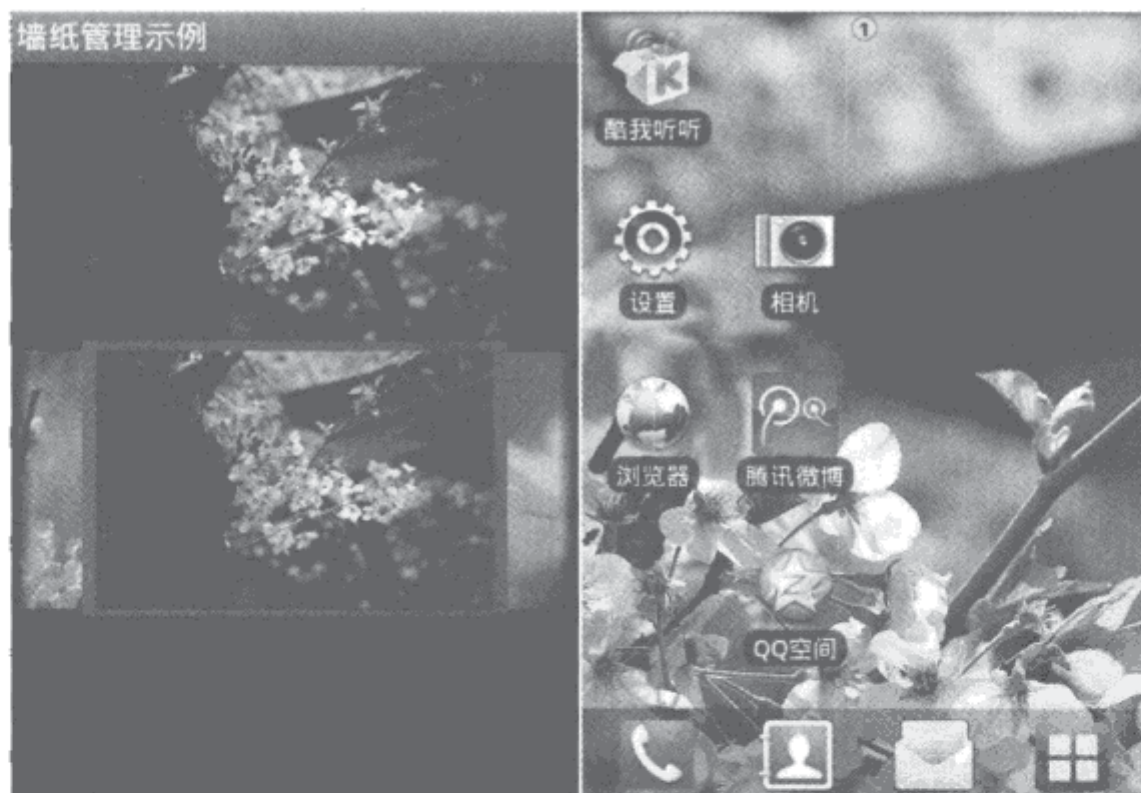


图 15-9 设置为系统墙纸的实机界面

1. 墙纸管理器

获取墙纸管理器的代码如下所示。

```
WallpaperManager mMgr = WallpaperManager.getInstance(this);
```

2. 设置墙纸

使用墙纸管理器的 `setBitmap`、`setResource` 和 `setStream` 方法可分别将位图、绘制用资源和文件流所对应的图片设置为系统墙纸。代码 15-19 是将绘制资源设置为系统墙纸



的实例代码。

代码 15-19 绘制资源设置为系统墙纸

文件名: WallpaperAct.java

```
1  @Override
2  public void onClick(View v) {
3      if(mCurPos == -1) { return; }
4      try { this.mMgr.setResource(FoolImageAdapter.mImageIds[mCurPos]);
5          } catch (IOException e) { e.printStackTrace(); }
6  }
```

3. 使用许可

设置墙纸需要拥有相应的使用许可，即必须在程序清单中声明设置墙纸的使用许可，其声明代码如下所示。

文件名: AndroidManifest.xml

```
<uses-permission android:name="android.permission.SET_WALLPAPER"/>
```

15.1.9 设备管理

设备策略管理器（DevicePolicyManager）用于管理设备上的执行策略，包括密码策略、锁定超时以及擦除用户数据等。图 15-10 所示为激活设备管理器的实机界面。



图 15-10 激活设备管理器的实机界面

1. 设备管理器

获取设备管理器的代码如下所示。

```
DevicePolicyManager mMgr =
    (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);
```

2. 激活设备管理器

使用设备管理器进行策略管理之前需要被激活。代码 15-20 是激活设备管理器的实

例代码。

代码 15-20 激活设备管理器

文件名: DevAdminAct.java

```

1  private void doEnableAdmin() {
2      Intent intent = new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
3      intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, mReceiverName);
4      startActivityForResult(intent, IConfig.REQ_CODE);
5  }
6
7  @Override
8  protected void onActivityResult(int requestCode, int resultCode, Intent data) {
9      if(requestCode==IConfig.REQ_CODE) {
10         if(resultCode == Activity.RESULT_OK) {
11             Toast.makeText(this, "允许设备管理!", Toast.LENGTH_LONG).show();
12         } else { Toast.makeText(this, "请求失败!", Toast.LENGTH_LONG).show(); }
13     }
14 }

```

在代码 15-20 中, 使用激活设备管理器的动作 (第 2 行) 来调用激活界面 (第 4 行)。其中, 必须使用意向对象的扩展空间来指明管理器组件 (第 3 行), 该管理器组件实际上是一个用以接收设备管理的状态的接收器 (DeviceAdminReceiver)。

3. 设备管理接收器

代码 15-21 是代码 15-20 所提到的设备管理接收器的定义, 其主要用于管理事件的回调处理。

代码 15-21 设备管理接收器的定义

文件名: DevAdminReceiver.java

```

1  public class DevAdminReceiver extends DeviceAdminReceiver {
2      @Override
3      public CharSequence onDisableRequested(Context context, Intent intent) {
4          return("是否取消设备管理!");
5      }
6      @Override
7      public void onDisabled(Context context, Intent intent) {
8          Toast.makeText(context, "取消设备管理!", Toast.LENGTH_LONG).show();
9      }
10     @Override
11     public void onEnabled(Context context, Intent intent) {
12         Toast.makeText(context, "允许设备管理!", Toast.LENGTH_LONG).show();
13     }
14     @Override
15     public void onPasswordChanged(Context context, Intent intent) {
16         Toast.makeText(context, "密码改变!", Toast.LENGTH_LONG).show();
17     }

```




Android 平台开发之旅 第 2 版

```

18     @Override
19     public void onPasswordExpiring(Context context, Intent intent) {
20         Toast.makeText(context, "密码过期!", Toast.LENGTH_LONG).show();
21     }
22     @Override
23     public void onPasswordFailed(Context context, Intent intent) {
24         Toast.makeText(context, "密码设置失败!", Toast.LENGTH_LONG).show();
25     }
26     @Override
27     public void onPasswordSucceeded(Context context, Intent intent) {
28         Toast.makeText(context, "密码设置成功!", Toast.LENGTH_LONG).show();
29     }
30 };

```

设备管理接收器还需要在应用程序清单文件中进行声明，并指明其所管理的策略等。代码 15-22 是程序清单文件中对设备管理接收器的声明实例代码。

代码 15-22 设备管理接收器的声明

文件名: AndroidManifest.xml

```

1  <receiver android:name=".DevAdminReceiver" android:label="@string/receiver"
2      android:permission="android.permission.BIND_DEVICE_ADMIN">
3      <meta-data android:resource="@xml/policies"
4          android:name="android.app.device_admin"/>
5      <intent-filter>
6          <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
7      </intent-filter>
8  </receiver>

```

在代码 15-22 中，不仅设置了使用许可（第 2 行）和行为（第 6 行），还指明了所管理策略的内容（第 3 行）。代码 15-23 是使用 XML 资源所定义的设备管理策略内容。

代码 15-23 设备管理策略

文件名: policies.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <device-admin xmlns:android="http://schemas.android.com/apk/res/android">
3      <uses-policies>
4          <limit-password />
5          <watch-login />
6          <reset-password />
7          <force-lock />
8          <wipe-data />
9      </uses-policies>
10 </device-admin>

```

在代码 15-23 中，第 4~8 行所定义的策略分别为设置密码规则、监视屏幕解锁尝试次数、更改屏幕解锁密码、锁定屏幕和清除所有数据。

4. 管理策略

一旦激活设备管理器，即可使用它来管理设备的策略（如设备密码的最小长度、设备锁定的最大时间等）。代码 15-24 是使用设备管理器来管理策略的实例代码。

代码 15-24 管理策略

文件名: DevAdminAct.java

```

1  private void doForceLock() { //强制锁定设备
2      if(mMgr.isAdminActive(mReceiverName)) { mMgr.lockNow(); }
3  }
4
5  private void doPassword() { //设置密码的最小长度策略
6      if(mMgr.isAdminActive(mReceiverName)) {
7          //设置密码最小长度
8          String password = ((EditText)findViewById(R.id.passwd)).getText().toString();
9          int minLength = Integer.parseInt(password);
10         mMgr.setPasswordMinimumLength(mReceiverName, minLength);
11         //设置新密码
12         Intent i = new Intent(DevicePolicyManager.ACTION_SET_NEW_PASSWORD);
13         startActivity(i);
14     }
15 }
16
17 private void doWipeData() { //清除设备数据
18     if(mMgr.isAdminActive(mReceiverName)) { mMgr.wipeData(0); }
19 }
```

在代码 15-24 中，先使用设备管理器的 `isAdminActive` 方法判断设备管理器是否激活，然后再使用设备管理器的相应方法来设置设备的策略。

5. 关闭设备管理器

为了系统安全，进行设备政策管理之后需要关闭设备管理器。代码 15-25 是关闭设备管理器的实例代码。

代码 15-25 关闭设备管理器

文件名: DevAdminAct.java

```

1  private void doDisableAdmin() {
2      if(mMgr.isAdminActive(mReceiverName)) {
3          mMgr.removeActiveAdmin(mReceiverName);
4      }
5  }
```

15.2 Android 平台系统信息

虽然开发者可以使用系统服务所提供的接口来获取一些常用的系统信息，但是对于有些应用，这些信息的粒度可能还不够，开发者需要一些更为精细的“专题信息”。



Android 平台对进程管理、文件系统、环境信息、系统时间、平台信息和电池管理等核心部分的访问进行了深层次地封装，从而让开发者能够获取到更多核心的系统信息。

15.2.1 进程管理

图 15-11 所示为显示当前应用程序的进程信息的实机界面。

1. 进程工具类

进程工具类（Process）提供了对操作系统进程的管理，包括获取当前程序的进程标识（PID）、线程标识（TID）和用户标识（UID）；获取/设置线程的优先级；结束指定进程或向指定进程发送信号等。该工具类在操作系统包（android.os）中定义。

2. 获取进程信息

代码 15-26 是获取进程信息的实例代码。

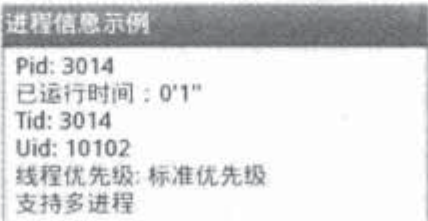


图 15-11 显示进程信息的实机界面

代码 15-26 获取进程信息

文件名: ProcessesDemoAct.java

```
1 private void showInfo() { //显示进程信息
2     printText("Pid: " + Process.myPid());
3     int elapsed = (int)(Process.getElapsedCpuTime());
4     printText("已运行时间: "+FooSysUtil.getInstance().toTimeStr2(elapsed));
5     int tid = Process.myTid();
6     printText("Tid: " + tid);
7     printText("Uid: " + Process.myUid());
8     int priority = Process.getThreadPriority(tid);
9     printText("线程优先级: "+FooProcUtil.getInstance().getPriorityDesc(priority));
10    printText(Process.supportsProcesses()?"支持多进程":"不支持多进程");
11 }
```

在代码 15-26 中，使用进程工具类的 `getThreadPriority` 方法可获取线程的优先级（第 8 行）。表 15-9 是 Android 平台所定义的线程优先级类型的说明。

表 15-9 线程优先级类型的说明

类 型	说 明
THREAD_PRIORITY_AUDIO	标准音频优先级
THREAD_PRIORITY_BACKGROUND	标准后台优先级
THREAD_PRIORITY_DEFAULT	标准优先级
THREAD_PRIORITY_DISPLAY	标准显示优先级
THREAD_PRIORITY_FOREGROUND	标准前台优先级
THREAD_PRIORITY_LESS_FAVORABLE	少喜好优先级
THREAD_PRIORITY_LOWEST	最低优先级
THREAD_PRIORITY_MORE_FAVORABLE	多喜好优先级
THREAD_PRIORITY_URGENT_AUDIO	重要音频优先级
THREAD_PRIORITY_DISPLAY	重要显示优先级

3. 进程管理

表 15-10 是进程工具类定义的用于进程管理方法的说明。

表 15-10 进程管理方法的说明

方 法	说 明
killProcess(int pid)	结束指定进程
sendSignal(int pid, int signal)	向指定进程发送信号

15.2.2 文件系统统计信息

图 15-12 所示为显示 SD 卡文件系统的统计信息的实机界面。

文件系统信息统计示例	文件系统信息统计示例
路径：/sdcard 可用块数：35779 总块数：60281 块大小（字节）：32768 空余块数：35779（59.35%） 已用块数：24502（40.65%）	路径：/sdcard 可用块数：26747 总块数：60281 块大小（字节）：32768 空余块数：26747（44.37%） 已用块数：33534（55.63%）

图 15-12 显示 SD 卡文件系统的统计信息的实机界面

1. 文件系统统计工具类

文件系统统计工具类（StatFs）用于获取与文件系统有关的统计信息，包括文件系统可用空间的块数、总块数、块大小和空余块数等信息。该工具类在操作系统包中提供。

2. 获取文件系统统计信息

代码 15-27 是获取指定路径下文件系统统计信息的实例代码。

代码 15-27 获取文件系统统计信息

文件名：ProcessesDemoAct.java

1	private void showStatInfo() { //显示统计信息
2	StatFs sfs = new StatFs(IConfig.DEST_PATH);
3	//获取统计信息
4	int total = sfs.getBlockCount();
5	int free = sfs.getFreeBlocks();
6	double usedRate = ((total-free)*100.0f)/total;
7	double freeRate = (free*100.0f)/total;
8	//显示统计信息
9	printText("路径: "+IConfig.DEST_PATH);
10	printText("可用块数: "+ sfs.getAvailableBlocks());
11	printText("总块数: "+ total);
12	printText("块大小（字节）: "+sfs.getBlockSize());
13	printText("空余块数: "+free+" (" +FooSysUtil.getInstance().round(freeRate) + "%) ");
14	printText("已用块数: "+(total-free)+" (" +
15	FooSysUtil.getInstance().round(usedRate) + "%) ");
16	}



15.2.3 环境信息

图 15-13 所示为显示当前设备的环境信息的实机界面。

1. 环境工具类

环境工具类（Environment）用于访问系统环境信息，包括数据文件夹、下载缓存文件夹、外部存储文件夹、外部存储状态和根文件夹等。该工具类在操作系统包中提供。

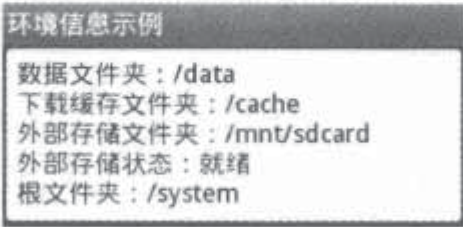


图 15-13 显示环境信息的实机界面

2. 获取环境信息

代码 15-28 是获取当前设备的环境信息的实例代码。

代码 15-28 获取环境信息

文件名：EnvInfoAct.java

```
1 private void showInfo() { //显示环境信息
2     printText("数据文件夹: "+Environment.getDataDirectory());
3     printText("下载缓存文件夹: "+Environment.getDownloadCacheDirectory());
4     printText("外部存储文件夹: "+Environment.getExternalStorageDirectory());
5     final String state = Environment.getExternalStorageState();
6     printText("外部存储状态: "+FooEnvUtil.getInstance().getStateDesc(state));
7     printText("根文件夹: "+Environment.getRootDirectory());
8 }
```

在代码 15-28 中，使用环境工具类的 `getExternalStorageState` 方法可获取当前外部存储器的状态（第 5 行）。表 15-11 是 Android 平台所定义的外部存储器状态类型的说明。

表 15-11 外部存储器状态类型的说明

类 型	说 明
MEDIA_BAD_REMOVAL	错误移除
MEDIA_CHECKING	正在检测
MEDIA_MOUNTED	就绪
MEDIA_MOUNTED_READ_ONLY	只读
MEDIA_NOFS	不支持的文件系统
MEDIA_REMOVED	已移除
MEDIA_SHARED	共享
MEDIA_UNMOUNTABLE	不能安装
MEDIA_UNMOUNTED	没有安装

15.2.4 时间管理

Android 平台提供了多种工具类用于时间的管理，包括计时器、计时器组件和倒计时器。

1. 计时器

图 15-14 所示为以 1s 为间隔进行计时的实机界面。

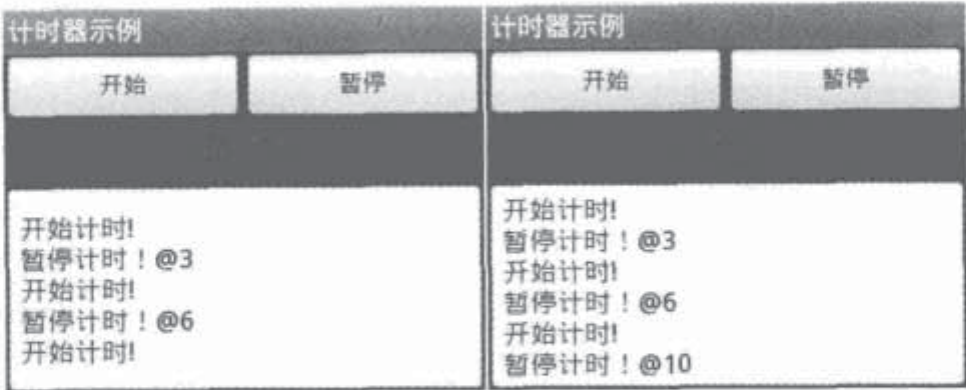


图 15-14 以 1s 为间隔计时的实机界面

(1) 计时器类

计时器类 (Timer) 用于规划在后台执行的任务，其定义在工具包 (java.util) 中。

(2) 计时器的使用

计时器所执行的周期性动作需要使用计时器任务 (TimerTask) 来定义。计时器任务实际上是一个单独线程 (其继承于 Runnable 接口)。如果要获取计时器任务的执行状态并使用可视控件进行显示，则涉及主线程与其他线程通信的问题，需要使用线程消息队列处理器 (Handler)。

(3) 计时器实例代码

代码 15-29 是使用计时器的实例代码，包括定义计时器和任务内容并启动计时器。

代码 15-29 计时器使用实例

文件名: TimerDemoAct.java

```
1  mHandler = new Handler() { //初始化主线程消息队列处理器
2      @Override
3      public void handleMessage(Message msg) {
4          if(msg.what == IConfig.MSG_ID) {
5              mCounter++;
6              mPanel.setText(FooSysUtil.getInstance().toTimeStr(mCounter));
7          }
8      }
9  };
10
11 private void doStart() { //开始计时
12     mTimer = new Timer();
13     mCounter = 0;
14     //定义任务
15     TimerTask task = new TimerTask() {
16         @Override
17         public void run() { //往主线程消息队列发送消息
18             mHandler.sendMessage(IConfig.MSG_ID);
19         }
20     };
21     //规划任务执行的时点和频率
22     mTimer.scheduleAtFixedRate(task, 0, 1000L);
```




```
23         printText("开始计时！");
24     }
25
```

在代码 15-29 中，使用计时器的 `scheduleAtFixedRate` 方法来规划任务的首次执行时点及频率并启动计时器（第 23 行）。

在任务定义体中，使用主线程的消息队列处理器来将消息周期性地发送给主线程（第 18 行）；在主线程中使用消息队列处理器的回调函数来接收外部线程所发来的消息并进行打印（第 7 行）。

（4）取消计时器

使用计时器的 `cancel` 方法可以取消计时器和所有的任务；使用 `purge` 方法将队列中所有被取消的任务清空。取消计时器的实例如代码 15-30 所示。

代码 15-30 取消计时器

文件名: `TimerDemoAct.java`

```
1 private void doPause() { //取消计时器
2     mTimer.cancel(); mTimer.purge(); mTimer = null;
3     printText("暂停计时！ @"+mCounter);
4 }
```

2. 计时器组件

对于计时器的使用，由于任务体是独立进程，所以任务与 `Activity` 组件的通信需要借助消息队列处理器。在小部件包中，`Android` 平台将计时器的使用封装成计时器组件（`Chronometer`）。图 15-15 所示为使用计时器组件进行计时的实机界面，当持续时长达到设定值时将自动停止计时。



图 15-15 计时器组件计时的实机界面

（1）声明计时器组件

作为可视组件，计时器组件可以在布局资源中进行声明。代码 15-31 是在程序主界面布局资源中声明计时器组件的实例代码。

代码 15-31 声明计时器组件

文件名: `main.xml`

```
1 <Chronometer android:id="@+id/chronometer" android:layout_gravity="center_horizontal"
```



```

2      android:layout_width="wrap_content" android:layout_height="wrap_content"
3      android:format="%s" android:padding="8sp"
4      android:textColor="#FF0000" android:textSize="32sp"/>

```

(2) 初始化计时器组件

与可视组件的初始化过程一样，在填充布局资源之后即可使用组件 ID 来获取组件对象。计时器组件的初始化如代码 15-32 所示。

代码 15-32 初始化计时器组件

文件名: ChronometerAct.java

```

1  @Override
2  public void onCreate(Bundle savedInstanceState) {
3      super.onCreate(savedInstanceState);
4      setContentView(R.layout.main);
5      //获取计时器控件并设置滴答事件侦听器
6      mChr = (Chronometer)findViewById(R.id.chronometer);
7      mChr.setOnChronometerTickListener(this);
8  }
9  @Override
10 public void onChronometerTick(Chronometer chr) {
11     if((SystemClock.elapsedRealtime()-chr.getBase())>=IConfig.LIMITED) {
12         chr.stop();
13         printText("时间到！");
14     }
15 }

```

在代码 15-32 中，为了获取计时状态，对计时器组件的滴答事件进行侦听（第 7 行），在其回调方法（第 10 行）中可执行规划任务。

(3) 启动计时

在启动计时之前，必须设置计时基准（用于获取计时增量），然后使用计时器组件的 start 方法启动计时，如代码 15-33 所示。

代码 15-33 启动计时

文件名: ChronometerAct.java

```

1  private void doStart() { //启动计时
2      mChr.setBase(SystemClock.elapsedRealtime());
3      mChr.start();
4      printText("开始计时！");
5  }

```

(4) 停止计时

使用计时器组件的 stop 方法即可停止计时。

3. 倒计时器

在计时应用中，很多场合需要倒计时，如某些操作要求时限；某些功能存在时效等。



Android 平台在操作系统包中提供倒计时类（CountDownTimer）用于倒计时应用。图 15-16 所示为使用倒计时器进行计时的实机界面。

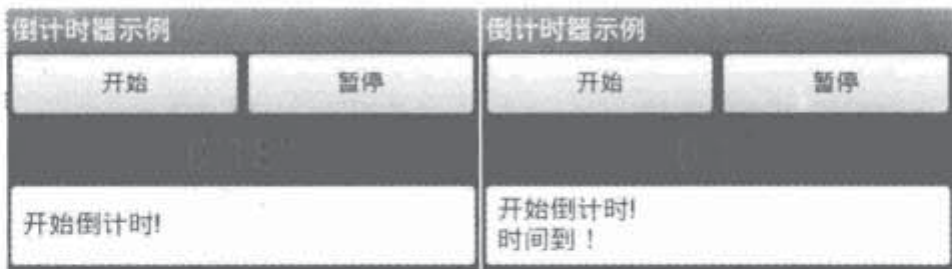


图 15-16 倒计时器计时的实机界面

(1) 应用程序配置信息接口

代码 15-34 是倒计时器示例程序的配置信息接口的定义，其中设定了计时时限和滴答间隔。

代码 15-34 配置信息接口的定义

文件名: IConfig.java

```
1 public interface IConfig {
2     public static final int LIMITED = 30*1000;           //倒计时限
3     public static final int INTERVAL = 1*1000;          //计时间隔
4 };
```

(2) 初始化倒计时器

构造倒计时器需要指明计时时限和滴答间隔。此外，为了响应计时结束和滴答事件，还需要重载倒计时器类。该初始化过程如代码 15-35 所示。

代码 15-35 初始化倒计时器

文件名: CountdownTimerAct.java

```
1 CountdownTimer mTimer = new CountdownTimer(IConfig.LIMITED, IConfig.INTERVAL) {
2     @Override
3     public void onFinish() { printText("时间到! "); }
4     @Override
5     public void onTick(long ms) {
6         mPanel.setText(FooSysUtil.getInstance().toTimeStr2((int)ms));
7     }
8 };
```

在代码 15-35 中，onFinish 和 onTick 分别是计时结束和计时滴答的回调方法。

(3) 启动/停止倒计时

使用计时器对象的 start 方法或 cancel 方法可启动或停止计时。

15.2.5 系统信息

图 15-17 所示为获取 Android 系统的调试信息和构建信息的实机界面。使用调试工具类（Debug）可获取系统的调试信息；使用构建工具类（Build）可获取系统的构建信息。这两

个工具类都在操作系统包中提供。



图 15-17 获取调试信息和构建信息的实机界面

1. 获取调试信息

使用调试工具类可以启动/停止跟踪调试并设置输出内容，还能获取内存调试信息，如代码 15-36 所示。

代码 15-36 获取系统调试信息

文件名: AndroidInfoAct.java

```
1 private void showDebugInfo() {
2     //启动调试
3     Debug.startMethodTracing(TRACE_NAME);
4     //Debug.printLoadedClasses(Debug.SHOW_FULL_DETAIL);
5     Debug.printLoadedClasses(Debug.SHOW_INITIALIZED);
6     //Debug.printLoadedClasses(Debug.SHOW_CLASSLOADER);
7     //获取内存信息
8     MemoryInfo memInfo = new MemoryInfo();
9     Debug.getMemoryInfo(memInfo);
10    showMemoryInfo(memInfo);
11 }
12
13 //显示内存有关信息
14 private void showMemoryInfo(MemoryInfo memInfo) {
15     StringBuffer sb = new StringBuffer();
16     sb.append("dalvik 虚拟机: "+memInfo.dalvikPss+" KB\n");
17     sb.append("本地堆: "+memInfo.nativePss+" KB\n");
18     sb.append("其他: "+memInfo.otherPss+" KB");
19     print(sb.toString() );
20 }
```

2. 获取构建信息

使用构建工具类可获取系统构建信息，如 SDK 版本、系统品牌、系统发布时间等，如



代码 15-37 所示。

代码 15-37 获取系统构建信息

文件名: AndroidInfoAct.java

```
1 private void showBuildInfo() { //显示构建信息
2     StringBuffer sb = new StringBuffer();
3     sb.append("SDK 版本: "+Build.VERSION.SDK+"\n");
4     sb.append("发布版本: "+Build.VERSION.RELEASE+"\n");
5     sb.append("内部版本: "+Build.VERSION.INCREMENTAL+"\n");
6     sb.append("包装: "+Build.BOARD+"\n");
7     sb.append("品牌: "+Build.BRAND+"\n");
8     sb.append("设备: "+Build.DEVICE+"\n");
9     sb.append("标签: "+Build.DISPLAY+"\n");
10    sb.append("指纹: "+Build.FINGERPRINT+"\n");
11    sb.append("主机: "+Build.HOST+"\n");
12    sb.append("标识: "+Build.ID+"\n");
13    sb.append("型号: "+Build.MODEL+"\n");
14    sb.append("产品: "+Build.PRODUCT+"\n");
15    sb.append("标记: "+Build.TAGS+"\n");
16    sb.append("时间戳: "+FooSysUtil.getInstance().unixTsp2Str(Build.TIME)+"\n");
17    sb.append("类型: "+Build.TYPE+"\n");
18    sb.append("用户: "+Build.USER);
19    print(sb.toString() );
20 }
```

15.2.6 电池状态

Android 平台提供电池管理类（BatteryManager）用于描述电池的状态，该类在操作系统中提供。图 15-18 所示为显示当前设备的电池状态的实机界面。

电池信息示例	电池信息示例
电量：49% 状态：充电中 温度：35 接入：USB电源 是否就绪：就绪 健康状况：良好 技术：Li-ion 电压：3	电量：49% 状态：没充电 温度：35 接入：电池供电 是否就绪：就绪 健康状况：良好 技术：Li-ion 电压：3

图 15-18 显示电池状态的实机界面

1. 获取电池状态

电池状态属于系统信息，用户程序只能“被动”接收该信息。当电池状态改变时，系统会发送广播消息，用户程序只能使用广播接收器来获取该消息，如代码 15-38 所示。

代码 15-38 初始化电池状态接收器

文件名: BatteryMonitorAct.java

```
1 private void init() {
```



```
2      mReceiver = new FooBatteryReceiver();
3      IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
4      this.registerReceiver(mReceiver, filter);
5  }
6
7  @Override
8  protected void onDestroy() { super.onDestroy();
9      this.unregisterReceiver(mReceiver);
10 }
```

在代码 15-38 中，程序以电池状态改变动作（第 3 行）作为过滤条件注册了电池状态接收器（第 4 行）。注册之后，当电池状态改变时，该接收器即可获取状态信息。当程序关闭时，需要注销该接收器（第 9 行）。

2. 电池状态接收器

代码 15-39 是代码 15-38 所提到的电池状态接收器的定义，该接收器在所重载的接收方法（onReceive）中获取电池状态。

代码 15-39 电池状态接收器的定义

文件名：BatteryMonitorAct.java

```
1  class FooBatteryReceiver extends BroadcastReceiver { // 电池状态接收器
2      @Override
3      public void onReceive(Context context, Intent intent) {
4          Bundle exts = intent.getExtras();
5          // 获取状态
6          int status = exts.getInt(BatteryManager.EXTRA_STATUS);
7          int plugged = exts.getInt(BatteryManager.EXTRA_PLUGGED);
8          boolean present = exts.getBoolean(BatteryManager.EXTRA_PRESENT);
9          int health = exts.getInt(BatteryManager.EXTRA_HEALTH);
10         // 显示信息
11         printText("电量: "+exts.getInt(BatteryManager.EXTRA_LEVEL)+"%");
12         printText("状态: "+FooBatteryUtil.getInstance().getStatusDesc(status));
13         printText("温度: "+(exts.getInt(BatteryManager.EXTRA_TEMPERATURE)/10));
14         printText("接入: "+FooBatteryUtil.getInstance().getPluggedDesc(plugged));
15         printText("是否就绪: "+(present?"就绪":"未就绪"));
16         printText("健康状况: "+FooBatteryUtil.getInstance().getHealthDesc(health));
17         printText("技术: "+exts.getString(BatteryManager.EXTRA_TECHNOLOGY));
18         printText("电压: "+exts.getInt(BatteryManager.EXTRA_VOLTAGE));
19     }
20 };
```

在代码 15-39 中，电池状态接收器在其重载的接收方法中获取电池状态信息，这些状态信息使用意向对象的扩展空间进行传递，需要对其扩展内容进行分解。使用电池管理器所定义的数据项即可获取对应的状态信息

表 15-12 是 Android 平台所定义的电池状态类型的说明。



表 15-12 电池状态类型的说明

类 型	说 明
BATTERY_STATUS_CHARGING	充电中
BATTERY_STATUS_DISCHARGING	放电中
BATTERY_STATUS_FULL	充满
BATTERY_STATUS_NOT_CHARGING	没充电
BATTERY_STATUS_UNKNOWN	未知

表 15-13 是 Android 平台所定义的电池健康状态类型的说明。

表 15-13 电池健康状态类型的说明

类 型	说 明
BATTERY_HEALTH_DEAD	需更换
BATTERY_HEALTH_GOOD	良好
BATTERY_HEALTH_OVER_VOLTAGE	电压过高
BATTERY_HEALTH_OVERHEAT	过热
BATTERY_HEALTH_UNKNOWN	未知状态
BATTERY_HEALTH_UNSPECIFIED_FAILURE	未定义失败

表 15-14 是 Android 平台所定义的电池接入类型的说明。

表 15-14 电池接入类型的说明

类 型	说 明
BATTERY_PLUGGED_AC	交流电源
BATTERY_PLUGGED_USB	USB 电源

3. 使用许可

获取电池状态需要拥有相应的使用许可，即必须在程序清单中声明获取电池状态的使用许可，其声明代码如下所示。

文件名：AndroidManifest.xml

```
<uses-permission android:name="android.permission.BATTERY_STATS"/>
```


第 16 章 Android 资源及 SDK 工具

本章对 Android 平台支持的资源类型及其定义、使用模式进行全面介绍。读者可以通过这些介绍来定义和使用程序所需的资源内容。本章还对 Android SDK 附带工具的常用方式进行简单说明。

16.1 资源类型及定义

俗话说“巧妇难为无米之炊”，对于 Android 程序而言，这些不可缺少的“米”就是各种可以编译到应用程序中的资源。Android 平台支持多种不同类型的资源文件，常见的包括 XML、PNG 和 JPG 等文件，其中 XML 文件还可以通过约定标记定义各种不同的资源元素，如字符串、颜色、大小值、数组、颜色、布局等。

资源定义通常都定义在工程文件夹下的资源文件夹（res）中。

16.1.1 常量值资源

常量值是指那些在程序中可以作为常量值（字符串、大小、数组）使用的资源。Android 平台定义了五类常量值资源：颜色值、字符串和格式化文本、大小值、数组（包括字符串数组和整数数组）、界面样式和主题。

1. 颜色值

一个颜色值包含了 RGB 值和透明度信息（Alpha），可以用于指定文本、背景的颜色。颜色值总是以字符“#”开头，然后是透明度和 RGB 值的字符串。一般有以下几种形式：

- #RGB。
- #ARGB。
- #RRGGBB。
- #AARRGGBB。

颜色值使用 XML 标记进行定义，一般存储在 values/colors.xml 文件中（实际上该文件名可以为任意，习惯上为 colors.xml）。该文件必须有一个 XML 的头部描述，并且有一个 <resources> 元素为根节点，该根节点可以包含一个或多个 <color> 标记。

XML 的头部描述为

```
<? xml version="1.0" encoding="utf-8" ?>
```

颜色值资源的定义语法为

```
<color name="颜色名 1">#颜色值 1</color>
<color name="颜色名 2">#颜色值 2</color>
```

在 Java 代码中使用 R.color.<颜色名> 对颜色值进行引用；在 XML 代码中使用 @color/<颜色名> 进行引用。



2. 字符串和格式化文本

字符串和格式化文本资源主要包括一些固定的文本内容，如按钮标题、提示文字、关于信息等。不同的是，格式化文本还可以包含一些标准的 HTML 标记，如****（Bold，粗体）、**<i>**（Italic，斜体）等。字符串和格式化文本也使用 XML 进行定义，一般存储在 `values/strings.xml` 文件中（实际上该文件名也可以为任意，习惯上为 `strings.xml`）。该文件也必须有一个 XML 的头部描述，并且有一个 `<resources>` 元素为根节点，该根节点可以包含一个或多个 `<string>` 标记。

字符串资源的定义语法为

```
<string name="字符串名 1">字符串 1</string>
<string name="字符串名 2">字符串 2</string>
```

在 Java 代码中使用 `R.string.<字符串名>` 对字符串进行引用；在 XML 代码中使用 `@string/<字符串名>` 进行引用。

3. 大小值

大小值一般用于指定各种各样的显示组件的大小（宽度和高度等）。Android 平台所定义的大小值支持以下 6 种单位。

1) 像素 (Pixel, px)，对应实际的屏幕像素（如 HVGA 的屏幕为 320×480 像素）。

2) 英寸 (Inches, in)，基于实际屏幕的物理大小。

3) 毫米 (Millimeter, mm)，基于实际屏幕的物理大小。

4) 点 (Point, pt)，1/72in，基于实际屏幕的物理大小。

5) 密度无关像素 (Density-independent Pixel, dp 或 dip)，抽象单位，基于屏幕的物理点阵密度。

6) 比例无关像素 (Scale-independent Pixel, sp)，与 dp 类似，不同的是，该单位可以根据用户的字体大小选择进行比例调节。

大小值使用 XML 进行定义，一般存储在 `values/dimens.xml` 文件中（实际上该文件名也可以为任意，习惯上为 `dimens.xml`）。该文件也必须有一个 XML 的头部描述，并且有一个 `<resources>` 元素为根节点，该根节点可以包含一个或多个 `<dimen>` 标记。

大小值资源的定义语法为

```
<dimen name="大小值名 1">大小值 1</dimen>
<dimen name="大小值名 2">大小值 2</dimen>
```

在 Java 代码中使用 `R.dimen.<大小值名>` 对大小值进行引用；在 XML 代码中使用 `@dimen/<大小值名>` 进行引用。

4. 数组

除了可以定义单个值，Android 平台还支持值集合——数组。数组在应用程序中一般用于提供列表内容。Android 平台支持字符串 (String) 和整形 (Integer) 两种数组。

数组使用 XML 进行定义，一般存储在 `values/arrays.xml` 文件中（实际上该文件名也可以为任意，习惯上为 `arrays.xml`）。该文件也必须有一个 XML 的头部描述，并且有一个 `<resources>` 元素为根节点，该根节点可以包含一个或多个数组类型节点（整形或字符串），这些数组节点又包含一个或多个 `<item>` 标记。

数组资源的定义语法为

```
<数组类型 name="数组名">
```



```

<item>数组项#1</item>
<item>数组项#2</item>
<item>..... </item>
<item>数组项#n</item>
</数组类型>

```

在 Java 代码中使用 `R.array.数组名` 对数组进行引用；在 XML 代码中使用 `@array/数组名` 进行引用。

5. 界面样式和主题

如果读者对界面样式和主题的概念还比较模糊，那么换成界面皮肤（Skin）的说法，相信大家就会很容易理解了。在一些常用的工具中，用户可以随心所欲地改变程序的皮肤界面，从而使程序的界面更加具有个性和绚丽。Android 平台也毫不例外地支持对应用程序界面的动态定义，这就是界面样式和主题。

读者可以把界面样式和主题视为界面组件的属性集。例如，对于文本组件，不再局限于像颜色值资源或大小值资源所表述的某一个属性的颜色值或大小值，而是可以包括字体颜色、背景色、前景色、高亮色、字体大小、字体样式等诸多的属性。

之所以要将样式和主题区分开，其主要因素是不同的作用域。界面样式一般作用于单个显示组件或某一个屏幕内的所有界面元素；而主题作用于整个应用程序，是全局性的。

界面样式和主题使用 XML 进行定义，一般存储在 `values/styles.xml` 文件中（实际上该文件名也可以为任意，习惯上为 `styles.xml`）。该文件也必须有一个 XML 的头部描述，并且有一个 `<resources>` 元素为根节点，该根节点可以包含一个或多个 `<style>` 节点，这些样式节点又包含一个或多个 `<item>` 标记。

界面样式和主题资源的定义语法为

```

<style name="样式名" parent="父样式">
    <item name="样式子项名#1">样式子项值#1</item>
    <item name="样式子项名#2">样式子项值#2</item>
    <item name=".....">.....</item>
    <item name="样式子项名#n">样式子项值#n</item>
</style>

```

(1) 界面样式

在 Java 代码中使用 `R.style.<样式名>` 对样式进行引用；在 XML 代码中使用 `@style/<样式名>` 进行引用。

(2) 界面主题

在 Java 代码中使用 `R.style.<主题名>` 对主题进行引用；在 XML 代码中使用 `@style/<主题名>` 进行引用。

16.1.2 绘制用资源

顾名思义，绘制用资源就是在程序中用于绘制的资源，常见的是图片。

Android 支持多种常见的图片资源，首选 PNG，次之是 JPG，最后才是 GIF。图片资源文件一般存放在 `drawable` 文件夹中。

在 Java 代码中通过 `R.drawable.<文件名>`（文件名不包括分隔符“.”和扩展名）对图片



进行引用；在 XML 代码中使用 `@drawable/<文件名>` 进行引用。

16.1.3 布局资源

布局资源的主要内容是应用程序的屏幕上可视组件的布局信息。Android 平台使用 XML 来定义布局资源，这些 XML 文件存放在 `layout` 文件夹中，ADT 自动创建的布局资源文件是 `layout/main.xml`。

该文件必须有一个 XML 的头部描述，并且有且仅有一个根节点，该根节点必须有一个 Android 的命名空间声明，该根节点可以包含一个或多个视图组件，从而形成用户界面组件的层次结构。

Android 命名空间格式为

```
xmlns:android="http://schemas.android.com/apk/res/android"
```

布局资源的定义语法为

```
<布局（视图组） xmlns:android="http://schemas.android.com/apk/res/android"
    “属性名#1”=属性值#1
    “属性名#2”=属性值#2
    .....
    “属性名#n”=属性值#n>
    <子视图#1 “属性名#1”=属性值#1 ...../>
    <子视图#2 “属性名#1”=属性值#1 ..... >
        <子视图#2.1 “属性名#1”=属性值#1 ...../>
        <子视图#2.2 “属性名#1”=属性值#1 ...../>
        .....
    </子视图#2>
    .....
</布局（视图组）>
```

在 Java 代码中使用 `R.layout.<文件名>`（文件名不包括分隔符“.”和扩展名）对布局进行引用；在 XML 代码中使用 `R.id.<布局组件 ID>` 来引用。

16.1.4 动画资源

动画资源可以算得上是 Android 平台的独特之处，它可以定义单张图片或序列图片的动态效果，这些效果包括旋转、淡入淡出、移动和伸缩。

动画资源也使用 XML 定义，每一个文件只能定义一个动画资源。这些文件一般存放在 `anim` 文件夹中。需要注意的是，动画资源文件对 XML 头部描述不作要求（虽然可有可无，但还是建议读者加上），有且只有一个根节点，根节点可以是以下 5 种标记之一。

- `<alpha>`（透明度）。
- `<scale>`（缩放比例）。
- `<translate>`（移动）。
- `<rotate>`（旋转）。
- `<interpolator>`（插值器）。

也可以通过 `<set>`（集合）标记块将以上 5 种标记中的一个或多个项目组合起来，形成一个动画效果集合。为了控制同步，在集合中的子标记可以通过属性 `startOffset` 来指定各自

出现的顺序。

提示：这里的插值器与“数值分析”课程中的插值是同一个概念，即在给定的模式（函数）下，根据因变量来获取中间插值。例如，淡入淡出插值器，其因变量是基于启动时点的时点流逝值，其插值内容为透明度。

动画资源的定义语法为

```
<set android:shareInterpolator="是否共享相同的插值器">
  <alpha
    android:fromAlpha=透明度开始值，取值【0.0，1.0】，0.0 是透明
    android:toAlpha=透明度终止值>
  <scale
    android:fromXScale=X 轴方向大小比例开始值，取值 1.0 为原大小值
    android:toXScale=X 轴方向大小比例终止值
    android:fromYScale=Y 轴方向大小比例开始值
    android:toYScale=Y 轴方向大小比例终止值
    android:pivotX=缩放固定点的 X 坐标
    android:pivotY=缩放固定点的 Y 坐标>
  <translate
    android:fromXDelta=X 轴位置开始值
    android:toXDelta=X 轴位置终止值
    android:fromYDelta=Y 轴位置开始值
    android:toYDelta=Y 轴位置终止值>
  <rotate
    android:fromDegrees=角度开始值
    android:toDegrees=角度终止值
    android:pivotX=旋转固定点的 X 坐标
    android:pivotY=旋转固定点的 Y 坐标 >
  <interpolator>
  <set>
</set>
```

对于缩放和旋转的固定点坐标以及移动的开始位置，可以使用以下 3 种格式。

- 取值“-100%”到“100%”，表示参考其自身大小的百分比。
- 取值“-100%p”到“100%p”，表示参考其屏幕大小的百分比。
- 直接的浮点数，表示是一个绝对值。

对于插值器标记，通常情况下标记名为插值器类的子类。

- CycleInterpolator，循环插值器。
- EaseInInterpolator，淡入插值器。
- EaseOutInterpolator，淡出插值器。

对<alpha>、<scale>、<translate>、<rotate>和<set>标记都支持的属性有：

- 1) duration（时长），动画效果持续的时长，单位为毫秒。
- 2) startOffset，该效果开始的时间偏移，单位为毫秒。
- 3) fillBefore，是否在动作开始前开始。
- 4) fillAfter，是否在动作结束后开始。



- 5) repeatCount, 该动作重复次数,“-1”为无限循环。
- 6) repeatMode, 重复模式,是重新开始(restart)还是倒回(reverse)。
- 7) zAdjustment, 图层的顺序,取值为正常(normal)、顶层(top)或底层(bottom)。
- 8) interpolator, 指定插值器。

在 Java 代码中使用 R.anim.文件名(文件名不包括分隔符“.”和扩展名)对动画进行引用;在 XML 代码中使用@anim/文件名进行引用。

16.1.5 菜单资源

Android 平台也有可选菜单、上下文菜单和子菜单之分,菜单的使用可以让应用程序更加符合用户的操作习惯,最重要的是可以有效地节约屏幕空间,这一点对于手机平台是相当重要的。

Android 平台的菜单资源可以使用 XML 定义,每一个文件只能定义一个菜单资源。这些文件一般存放在 menu 文件夹中。需要注意的是,菜单资源文件也对 XML 头部描述不作要求,有且只有一个根节点,该根节点必须有一个 Android 的命名空间声明。

菜单资源的定义语法为

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item //编者注:普通菜单项
        属性名#1=属性值#1
        属性名#2=属性值#2
        .....
        属性名#n=属性值#n />
    <group 属性名#1=属性值#1 .....> //编者注:菜单组
        <item 属性名#1=属性值#1 ...../>
        <item 属性名#1=属性值#1 ...../>
        .....
    </group>
    <item 属性名#1=属性值#1 .....>
        <menu> //编者注:子菜单
            <item 属性名#1=属性值#1 ...../>
            .....
        </menu>
    </item>
</menu>
```

在 Java 代码中使用 R.menu.<文件名>(文件名不包括分隔符“.”和扩展名)对菜单进行引用;菜单是一种特殊资源,一般不在 XML 中进行引用。

16.1.6 文件资源

在多数应用程序中,总少不了文件资源,如配置文件、声音文件以及其他的自定义数据文件。对于文件资源,虽然 Android 平台不再使用 XML 进行定义,但是 Android 平台还是提供了一些方法来方便用户对文件资源的使用。

Android 平台中常见的文件资源包括 XML 文件和原文件(Raw)。

1. XML 文件

与上述使用 XML 定义的资源文件不同，这里的 XML 文件不是按照 Android 所规定的语法，而是用户自定义的 XML 文件。如图 16-1 所示，该文件路径为 xml/db_setting.xml。

```

1 db_setting.xml
2
3 <?xml version="1.0" encoding="utf-8"?>
4 <Setting>
5   <Uri value="jdbc:oracle:thin:@localhost:1521/PIMS"/>
6   <User value="guest"/>
7   <Password value="*****"/>
8 </Setting>

```

图 16-1 用户自定义的 XML 文件

使用资源管理器的 `getXml` 方法可获取该 XML 文档的解析器，继而进行下一步的解析工作（详细可参考 XML 应用章节）。代码 16-1 是解析 XML 文件资源的入口代码。

代码 16-1 解析 XML 文件资源

```

1 private void parseGenericXml() { //解析常规 XML 资源
2     //获取 XML 资源解析器
3     XmlResourceParser parser = this.getResources().getXml(R.xml.db_setting);
4     //开始解析
5     try { new FooXmlParseUtil(parser, mHandler).parse();
6     } catch (XmlPullParserException e) { e.printStackTrace(); }
7     catch (IOException e) { e.printStackTrace(); }
8 }

```

在代码 16-1 中，引用了 xml 文件夹中的 db_setting.xml 文件（第 3 行）。

2. 原文件

原文件的范围要比 XML 文件更广泛，除了各种可以直接阅读的文本文件，还包括各种不可以直接阅读的二进制文件；既可以是 ASCII 字符，也可以是中文。

使用资源管理器 `openRawResource` 方法可获取到原文件的输入流，继而进行下一步的读取。代码 16-2 是读取原文件资源的主要代码。

代码 16-2 读取原文件资源

```

1 private void readRaw(int resid) throws IOException { //读取原文件内容
2     //获取原文件输入流
3     InputStream is = this.getResources().openRawResource(resid);
4     DataInputStream dis = new DataInputStream(is);
5     byte[] data = new byte[is.available()];
6     dis.readFully(data);
7     dis.close();
8     is.close();
9     //显示文件内容
10    this.mTxtContents.setText(new String(data, "GBK"));
11 }

```

16.1.7 备选资源

无论在任何平台，产品化的软件都必须考虑多语言、多场所（Locale）的问题。在移动



平台中，这种情况表现得尤为突出，除了界面语言、场所设置外，当屏幕方向改变或者针对不同机型的键盘，程序都必须进行相应地响应。为了适应检测到的配置变化，这些应用程序要预先准备多套不同的资源，对于不同的配置，载入不同的资源。

在 Android 平台中，无论是对配置的改变还是对备选资源的载入，这些过程都由平台自动完成，无需在应用程序中心进行额外的处理。开发者要做的，仅仅是按照 Android 平台的约定预先准备好若干套备选资源。

表 16-1 是 Android 平台所定义的与备选资源相关的配置类型的说明。

表 16-1 备选资源相关的配置类型的说明

类 型	说 明
语言	小写的、ISO 639-1 中两位语言代码，如 en、zh
区域	大写的、ISO 3166-1-alpha-2 中两位语言代码，小写“r”作为前缀，如 rUS、rCN
屏幕方向	<ul style="list-style-type: none">● port (portrait, 纵向的)● land (landscape, 横向的)● square (四方的)
屏幕像素密度	92dpi、108dpi 等
触摸屏类型	<ul style="list-style-type: none">● notouch (无触摸屏)● stylus (触摸笔)● finger (手指)
键盘是否可用	<ul style="list-style-type: none">● keysexposed (可用的)● keyshidden (隐藏的)
主文本输入法	<ul style="list-style-type: none">● nokeys, 无键盘● qwerty, 标准键盘● 12key, 12 键盘 (0~9 数组按键，再加上“*”和“#”)
主导航方式 (非触摸屏)	<ul style="list-style-type: none">● nonav, 无导航● dpad, 5 方向 (上、下、左、右和确认) 键盘● trackball, 轨迹球● wheel, 滚轮
屏幕大小	320×240, 640×480 等。大的尺寸必须放在前面

常见的，当屏幕方向发生改变（从横向变为纵向），相应的桌面背景也会发生变化（横向背景变为纵向背景）。这就是说，开发者必须预先准备两张背景图片，如图 16-2 和图 16-3 所示。

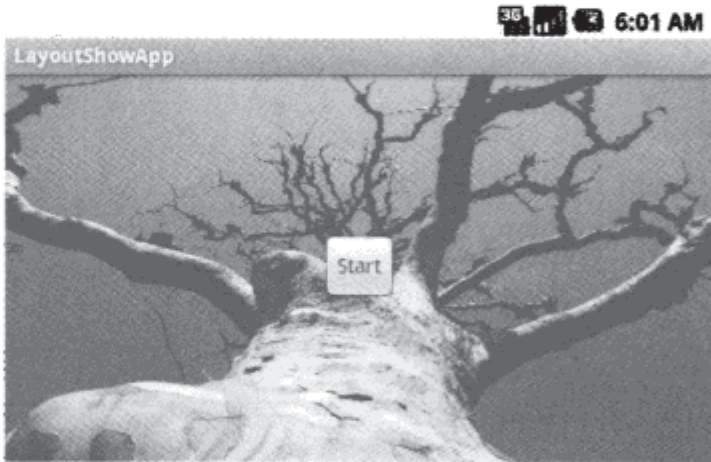


图 16-2 横向屏幕背景图片



图 16-3 纵向屏幕背景图片

对于这两张背景图片的区分，Android 平台约定：这两个不同图片文件必须使用相同的文件名，在其所存放的标准文件夹名称的基础上，分别以配置类型的值作为后缀（以连字符“-”分隔）新增文件夹，并将图片放入到对应的配置类型为后缀的文件夹中。

通常的，图片资源的标准存放位置是 `drawable` 文件夹。按照约定，对于屏幕的横向和纵向这两种不同配置，在 `drawable` 的基础上，分别以 `land` 和 `port` 作为后缀新增两个文件夹。横向的图片放入 `drawable-land` 文件夹；纵向的放入 `drawable-port` 文件夹。

不仅如此，Android 平台支持备选资源的多级后缀，即多个配置类型可以进行排列组合，以此来满足不同的配置变化。例如，`drawable-zh-rCN-land-160dpi-480x320` 文件夹存放的是支持中文语言、中国区域、横向屏幕、屏幕像素密度为 160dpi、屏幕大小为 480×320 像素的绘制用资源文件。

16.2 资源的使用模式

上述各种资源都是通过 XML 标记来进行定义的，Android 平台内嵌了对资源文件进行解析的 XML 解析引擎（基于 Xml Pull API）。通过该引擎，Android 平台会“识别”资源定义中的 XML 标记，并根据其属性信息，生成对应的组件对象实例。

16.2.1 资源 ID

Android 平台生成组件对象实例的行为对于开发者而言是不可见的，要想获取定义组件所对应的组件对象实例，只能在 Android 平台中通过资源 ID 来获取资源对应的对象实例。其实例代码如下所示。

```
Button btnSend = (Button)findViewById(R.id.btn_send);
```

资源 ID 是用于标识资源，用 `android:id` 属性来定义。代码 16-3 是一个按钮组件的 XML 定义。

代码 16-3 按钮组件的 XML 定义

```
1 <Button android:id="@+id/btn_send" android:text="发送"
2     android:layout_width="wrap_content" android:layout_height="wrap_content"
3     android:below="@id/text"/>
```

在代码 16-3 中，第 1 行的字符“@”、“+”和“/”都是 Android 平台约定的分隔符，用于“告诉”XML 解析器，该属性值中分隔字符“/”后是一个 ID 字符串，并且必须以此 ID 实例化该组件（这样才能通过资源 ID 找到对应的组件实例）。

16.2.2 引用资源

引用资源的情形有两种：在 XML 代码中引用资源和在 Java 代码中引用资源。

(1) XML 代码引用资源

在代码 16-3 中第 3 行，该按钮组件参考了一个组件（`text`），表示其在该组件的下方。这里“@”字符后不是“+”，XML 解析器就会识别出分隔字符“/”后的 ID 字符串是引用其他资源。这里的其他资源既可以是用户定义的资源，也可以是系统资源。



(2) Java 代码引用资源

在 Java 代码中主要通过 ID 来引用资源。Activity 和视图都定义了 `findViewById` 方法，该方法依据资源 ID 获取对应的组件实例。Java 代码中资源 ID 的形式为 `R.id.<组件 ID>`。其中“R”是一个由资源分析工具通过对资源文件的分析所映射的一个类定义，其中包含了所有资源的标识信息。

16.2.3 XML 属性

XML 属性是指用于描述资源特征的属性标记，如代码 16-3 中第 1~3 行都是该按钮组件的属性，这些属性用于设置该按钮的宽度、高度、位置和显示文本。

16.3 系统资源定义

系统资源是指由 Android 系统定义的一些资源，这些资源被用于 Android 的应用程序。这些资源包括动画、数组、颜色值、大小值、绘制用资源、ID、整数值、布局、字符串以及界面样式和主题。表 16-2 是 Android 平台所定义的资源类型的说明。

表 16-2 系统资源类型的说明

类 型	说 明
动画	定义在 <code>R.anim</code> 类中
数组	定义在 <code>R.array</code> 类中
属性值	定义在 <code>R.attr</code> 类中
颜色值	定义在 <code>R.color</code> 类中
大小值	定义在 <code>R.dimen</code> 类中
绘制用	定义在 <code>R.drawable</code> 类中
ID	定义在 <code>R.id</code> 类中
整数	定义在 <code>R.integer</code> 类中
布局	定义在 <code>R.layout</code> 类中
字符串	定义在 <code>R.string</code> 类中
样式和主题	定义在 <code>R.style</code> 类中

16.4 Android SDK 工具使用

16.4.1 adb 工具

adb 是 Android 平台的调试桥接工具 (Android Debug Bridge)，存放在 Android SDK 安装文件夹下的 `platform-tools` 文件夹中，其运行界面如图 16-4 所示。

使用 adb 工具之前必须先使用 USB 线连接实机或启动模拟器，以下是该工具常用命令。

(1) 连接到实机或模拟器

```
adb shell
```

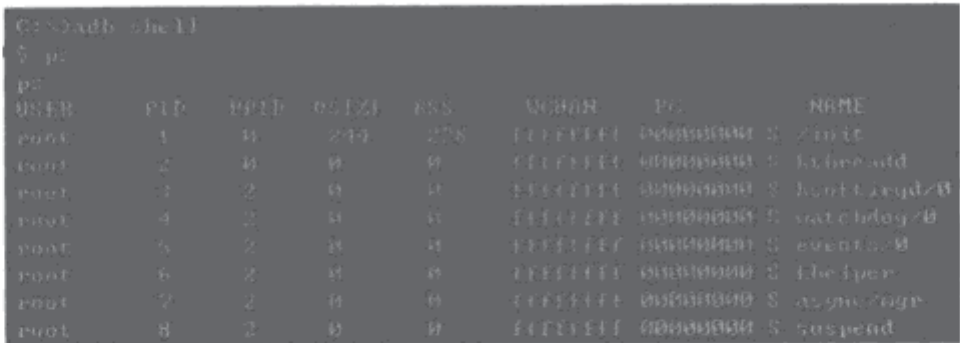



图 16-4 adb 工具运行界面

连接到实机或模拟器后，开发者可以通过命令行调用系统程序（如 ls、mkdir、ps 等标准 Linux 命令）来操作实机或模拟器系统。

(2) 上传本地（PC 端）文件到实机或模拟器中

```
adb push <本地路径> <远程文件路径>
```

(3) 从实机或模拟器下载文件到本地

```
adb pull <远程文件路径> <本地路径>
```

(4) 安装本地包文件到实机或模拟器中

```
adb install <本地包文件>
```

(5) 移除实机或模拟器中已安装的包

```
adb uninstall <包名>
```

提示：adb 工具对开发调试相当有用，如可以监测应用程序对文件的操作（创建、删除文件等），或者为应用程序提供试验数据。例如，可以先在桌面系统中创建数据文件，然后将该文件通过 adb 工具上传到模拟器或实机上用于调试。需要注意的是，adb 工具需要连接到模拟器或实机，所以必须保证模拟器或实机已经启动。

16.4.2 ddms 工具

ddms 是 Dalvik 虚拟机调试监控工具（Dalvik Debug Monitor），其存放在 Android SDK 安装文件夹的 tools 文件夹中，其运行界面如图 16-5 所示。

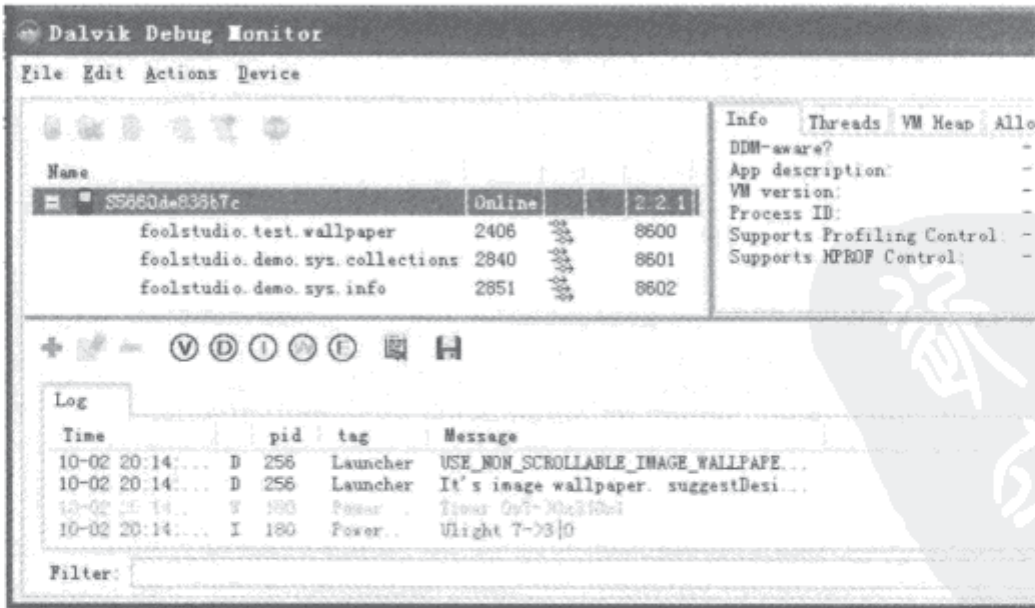


图 16-5 ddms 工具运行界面

ddms 工具除了可以监视虚拟机和应用程序的运行状况外，还可以截取设备当前屏幕，如图 16-6 所示。

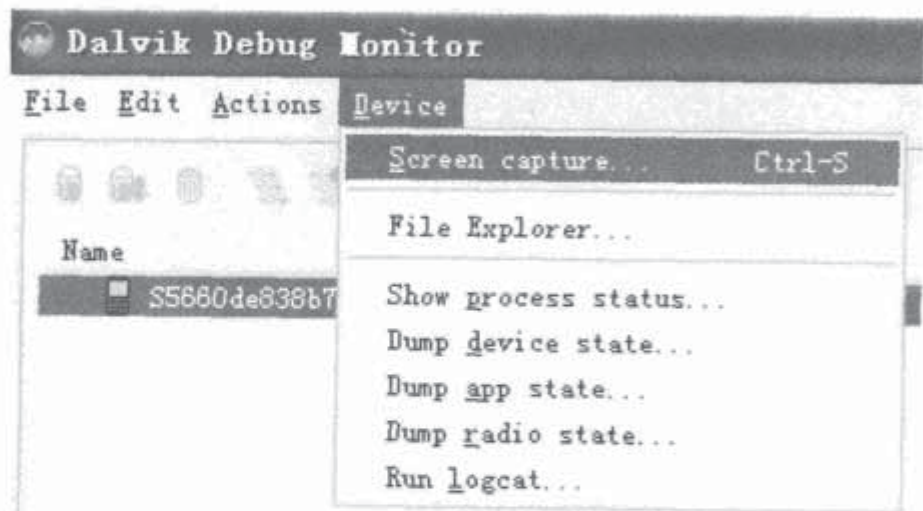


图 16-6 使用 ddms 工具截取设备屏幕

16.4.3 sqlite3 工具

sqlite3 工具用于操作 SQLite 数据库（第 3 版），存放在 Android SDK 安装文件夹的 Tools 文件夹中，其运行界面如图 16-7 所示。

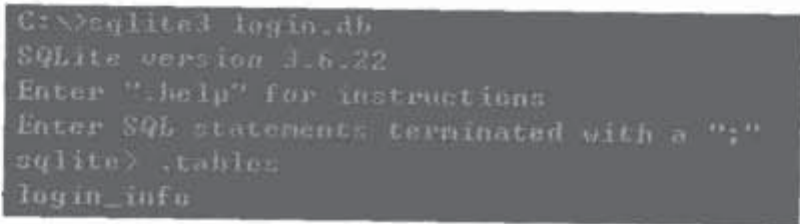


图 16-7 sqlite3 工具运行界面

以下是 sqlite3 工具的常用命令，开发者可以通过 `-help` 参数来获取使用帮助。

(1) 打开或创建 SQLite 数据库

```
sqlite3 <数据库路径>
```

(2) 查看版本信息

```
sqlite3 <version>
```

通过 sqlite3 工具打开并创建 SQLite 数据库，然后通过 sqlite3 工具提供的命令方式来使用 SQL 语句创建数据表、记录操作等。

16.4.4 keytool 工具

keytool 工具用于生成、查看、导入/导出密钥库，存放在 JDK 安装文件夹的 bin 文件夹中。开发者可以通过命令 `keytool-help` 来获取其使用帮助。以下是 keytool 工具的常用方式。

(1) 创建密钥库文件

```
keytool -genkey -keystore <密钥库路径> -alias <密钥库别名> -keyalg <密钥算法>
```

(2) 列举密钥库信息

```
keytool -list -keystore <密钥库路径>
```


附录 随书源代码说明

源代码在随书 CD 中，内容按照章划分文件夹（见图 A-1），各章文件夹下又按照工程文件夹为单位存放（见图 A-2）。

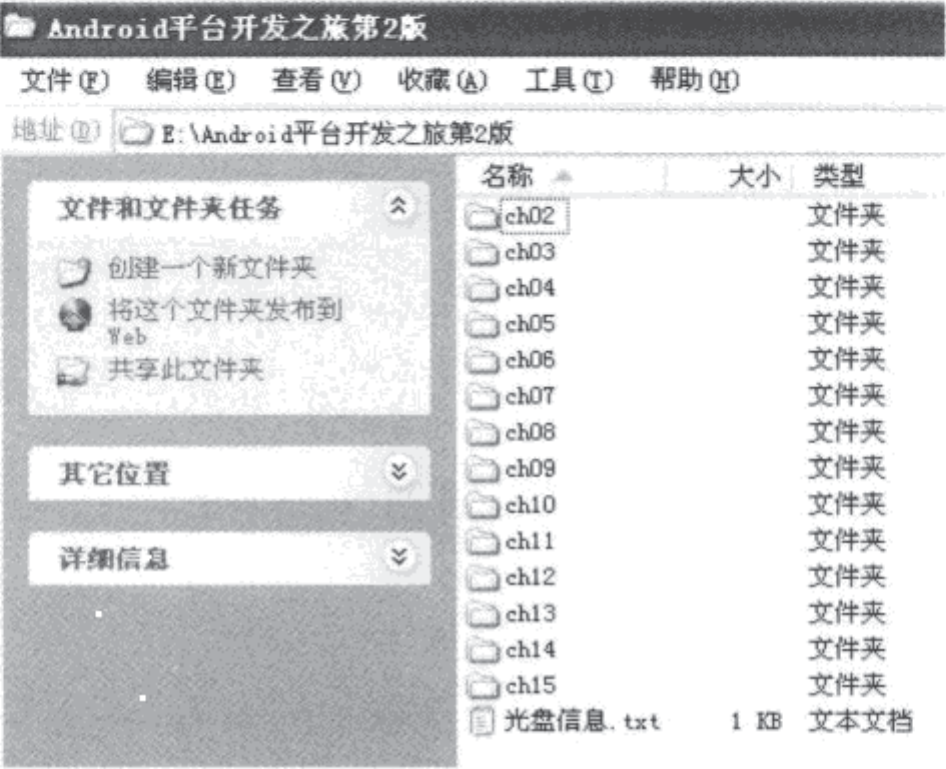


图 A-1 源代码根目录结构



图 A-2 第 10 章目录结构

本书中的源代码需要在 Eclipse（3.6 版本及以上）+ADT 的集成开发环境中通过导入工程来运行，详细指导可以参考第 2 章。

参 考 文 献

- [1] Android 开发者网站[OL]. <http://developer.android.com/>.
- [2] Android 开发资源网址[OL]. <http://code.google.com/android/>.
- [3] OpenGL ES 官方网[OL]. <http://www.khronos.org/opengles/>.
- [4] WebKit 开源项目官方网[OL]. <http://webkit.org/>.
- [5] SQLite 官方网[OL]. <http://www.sqlite.org/>.
- [6] Db4o 官方网[OL]. <http://www.db4o.com/>.
- [7] XML Pull API 官方网站[OL]. <http://www.xmlpull.org/>.
- [8] Ubuntu 门户网站[OL]. <http://www.ubuntu.com/>.



ISBN 978-7-111-37276-9

策划编辑：车 忱

封面设计： 子时文化
ZiShi Culture

本书涵盖Android的新特性，立足实际的开发案例，介绍了Android平台开发的基础概念、实用技术和应用模式。主要内容包括应用程序框架、高级界面、数据库应用、网络通信与Web开发、无线通信、多媒体应用、个人信息管理、电话系统管理、XML应用、地图应用和信息系统管理。

地址：北京市百万庄大街22号
电话服务
服务中心：(010)88361066
销售一部：(010)68326294
销售二部：(010)88379649
读者购书热线：(010)88379203

邮政编码：100037
网络服务
门户网：<http://www.cmpbook.com>
教材网：<http://www.cmpedu.com>
封面无防伪标均为盗版

定价：69.80元(含1CD)

上架建议 计算机/移动与嵌入式开发

ISBN 978-7-111-37276-9



9 787111 372769 >