精通 Android 5 多媒体开发

王石磊◆编著

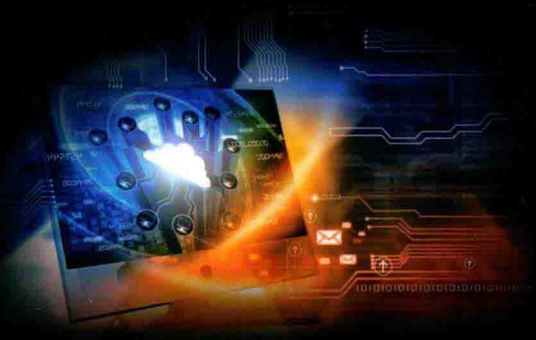
详细地讲解了 Android 多媒体开发的主要应用

Android 多媒体框架详解、音频系统框架详解、视频系统框架详解、照相机系统详解、Alarm 时钟系统详解、振动器系统详解、二维图像渲染详解、绘制二维图像、二维动画应用、渲染二维图像、开发音频应用程序、开发视频应用程序、OpenGL ES 系统基本应用、纹理映射、绘制不同的三维形状、坐标变换和混合、开发一个屏保系统、开发一个音乐播放器、开发一个闹钟系统等。

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

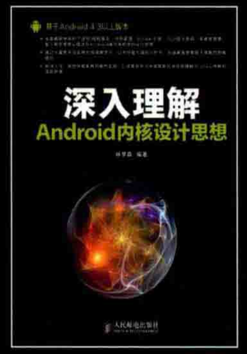
精通 Android 5 多媒体开发



作者简介

王石磊，通信工程硕士，计算机工程硕士，资深Android开发工程师和架构师，现在专门从事Andorid移动客户端的通信研发工作。曾经在谷歌市场中发布过多款应用，这些应用软件在谷歌市场上取得了好的销售成绩。另外，还精通C#、Java、iOS等开发技术，并且精通Linux底层嵌入式开发技术，曾经独立开发过一款通信产品。业余期间，曾经在国内主流期刊中发表过多篇通信领域的论文。

畅销书推荐



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

封面设计：董志桢

分类建议：计算机／程序设计／移动开发
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-38758-5
定价:89.00 元

精通Android 5多媒体开发

王石磊◆编著

人民邮电出版社
北京

图书在版编目（CIP）数据

精通Android 5多媒体开发 / 王石磊编著. -- 北京 : 人民邮电出版社, 2015.11
ISBN 978-7-115-38758-5

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

中国版本图书馆CIP数据核字(2015)第073608号

内 容 提 要

在整个Android应用开发体系中，图像、音频、视频、渲染和3D处理一直是其核心应用。本书分为5篇，共计24章，循序渐进地讲解Android多媒体应用开发的基本知识，遵循从底层原理开始到顶层应用结束的开发过程，全程剖析Android多媒体应用开发的所有核心知识点。本书从获取并编译Android源码开始讲起，依次讲解基本技术、系统分析、典型应用、三维技术、综合实战这5大部分的知识。在讲解每一个知识点时，都从基础理论开始入手，遵循由浅入深的写作方法，按照运作流程逐步分析Android多媒体应用的方方面面。本书几乎涵盖Android多媒体系统的所有主要内容。

本书适合Android爱好者、Android初学者、Android应用开发者、Android视频/音频开发者、Android游戏开发者，也可以作为相关培训学校和大专院校相关专业的教学用书。

◆ 编　　著　王石磊
责任编辑　张　涛
责任印制　张佳莹　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京中新伟业印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：36.5
字数：1016千字　　　　2015年11月第1版
印数：1－2 500册　　　2015年11月北京第1次印刷

定价：89.00元

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

前 言

Android 系统是一款于 2007 年问世的基于 Linux 平台的开源手机操作系统，该平台由操作系统、中间件、用户界面和应用软件组成，是首个为移动终端打造的真正开放和完整的移动软件。

本书的内容

本书依次讲解了 Android 技术概述、Android 技术核心框架分析、获取并分析 Android 源码、Android 多媒体框架、音频系统框架、视频系统框架、照相机系统、Alarm 时钟系统、振动器系统、二维图像渲染、绘制二维图像、二维动画应用、渲染二维图像、开发音频应用程序、开发视频应用程序、OpenGL ES 系统初步、OpenGL ES 基本应用、纹理映射、绘制不同的三维形状、坐标变换和混合、OpenGL ES 进阶、开发一个屏保系统、开发一个音乐播放器、开发一个闹钟系统等知识。

本书版本

Android 系统自 2008 年 9 月发布第一个版本 1.1 以来，截至 2014 年 10 月发布的最新版本 5.0，一共存在十多个版本。由此可见，Android 系统升级频率较快，一年之中最少有两个新版本诞生。如果过于追求新版本，会造成力不从心的结果。在此建议广大读者不必追求最新的版本，只需关注最流行的版本即可。据官方统计，截至 2014 年 12 月 15 日，占据前 3 位的版本分别是 Android 4.4、Android 4.3 和 Android 4.2，其实这 3 个版本的区别并不是很大，只是在某领域的细节上进行了更新。

本书的内容以笔者撰稿时的最新版本 Android 5.0 为基础，并且兼容了 Android L 及其以前的版本。

本书特色

本书内容丰富、全面。我们的目标是通过一本图书提供多本图书的价值，读者可以根据自己的需要有选择地阅读。在内容的编写上，本书具有以下特色。

（1）结构合理。

从用户的实际需要出发，合理安排知识结构，内容由浅入深，详细地讲解了和 Android 多媒体应用开发有关的知识。

（2）遵循“理论介绍—演示实例—综合演练”主线。

为了使广大读者彻底弄清楚 Android 多媒体应用开发的每一个知识点，在讲解时依次剖析了基本理论、演示实例分析、综合实战演练等内容，遵循了从理论到实践的原则，实现了实践教学这一目标。

（3）易学易懂。

本书内容条理清晰，语言简洁，读者既可以按照本书编排的章节顺序进行学习，也可以根据自己的需求对某一章节进行有针对性的学习。

（4）实用性强。

本书彻底摒弃枯燥的理论和简单的操作，注重实用性和可操作性，详细讲解了各个知识点的实战知识。

读者对象

初学 Android 编程的自学者；
多媒体开发人员；
游戏开发人员；
大中专院校的老师和学生；
做毕业设计的学生；
Android 编程爱好者；
相关培训机构的老师和学员；
从事 Android 开发的程序员。

在编写本书的过程中，得到了人民邮电出版社工作人员的大力支持，正是各位编辑的求实、耐心和效率，才使得本书在这么短的时间内出版。另外，也十分感谢我的家人在我写作的时候给予的巨大支持。本人毕竟水平有限，本书如有纰漏和不尽如人意之处，诚请读者提出意见或建议，以便修订并使之更臻完善。另外，我们提供了答疑和源程序下载支持网站 http://www.toppr.net/，读者如有疑问可以在此提出，一定会得到满意的答复。

编　者

目　录

第一篇　基础技术篇

第1章　Android技术概述……2

1.1　智能手机系统介绍……2
1.1.1　何谓智能手机……2
1.1.2　看当前主流的智能手机系统……2
1.1.3　Android 5.0的突出变化……3
1.2　搭建Android应用开发环境……4
1.2.1　安装Android SDK的系统要求……4
1.2.2　安装JDK……5
1.2.3　获取并安装Eclipse和Android SDK……8
1.2.4　安装ADT……10
1.2.5　设定Android SDK Home……12
1.2.6　验证开发环境……13
1.2.7　创建Android虚拟设备（AVD）……13
1.2.8　启动AVD模拟器……16
1.2.9　解决搭建环境过程中的常见问题……18

第2章　Android技术核心框架分析……21

2.1　简析Android安装文件……21
2.1.1　Android SDK目录结构……21
2.1.2　android.jar及内部结构……22
2.1.3　阅读SDK帮助文档……22
2.1.4　常用的SDK工具……23
2.2　演示官方实例……24
2.3　剖析Android系统架构……28
2.3.1　Android体系结构介绍……28
2.3.2　Android应用工程文件组成……30
2.4　简述五大组件……32
2.4.1　用Activity来表现界面……32
2.4.2　用Intent和IntentFilter实现切换……33
2.4.3　Service为你服务……33
2.4.4　用BroadcastIntentReceiver发送广播……34
2.4.5　用ContentProvider存储数据……34
2.5　进程和线程……34
2.5.1　先看进程……34
2.5.2　再看线程……35
2.5.3　应用程序的生命周期……35
2.6　第一段Android程序……37

第3章　获取并分析Android源码……42

3.1　获取Android源码……42
3.1.1　在Linux系统中获取Android源码……42
3.1.2　在Windows系统中获取Android源码……43
3.2　分析Android源码结构……45
3.3　编译Android源码……46
3.3.1　搭建编译环境……47
3.3.2　开始编译……48
3.3.3　在模拟器中运行……49
3.3.4　常见的错误分析……49
3.3.5　实践演练——演示两种编译Android程序的方法……50

第二篇　系统分析篇

第4章　Android多媒体框架……56

4.1　Android多媒体系统介绍……56
4.2　OpenMax框架详解……57
4.2.1　分析OpenMax框架构成……58
4.2.2　实现OpenMax IL层接口……62
4.3　分析OpenCore框架……68
4.3.1　OpenCore层次结构……69
4.3.2　OpenCore代码结构……70
4.3.3　OpenCore编译结构……71
4.3.4　操作系统兼容库……74
4.3.5　实现OpenCore中的OpenMax部分……75
4.3.6　OpenCore扩展详解……85
4.4　Stagefright框架详解……91
4.4.1　Stagefright代码结构……91
4.4.2　Stagefright实现OpenMax接口……91
4.4.3　分析Video Buffer传输流程……94

第 5 章 音频系统框架……99

5.1 音频系统基础……99
5.2 分析音频系统的层次……100
5.2.1 层次说明……100
5.2.2 Media 库中的 Audio 框架……101
5.2.3 本地代码……103
5.2.4 分析 JNI 代码……105
5.2.5 Java 层代码简介……106
5.3 Audio 系统的硬件抽象层……106
5.3.1 Audio 硬件抽象层基础……107
5.3.2 分析 AudioFlinger 中的 Audio 硬件抽象层的实现……108
5.3.3 真正实现 Audio 硬件抽象层……113
5.4 分析编码/解码过程……113
5.4.1 AMR 编码……114
5.4.2 AMR 解码……117
5.4.3 解码 MP3……120

第 6 章 视频系统框架……122

6.1 视频系统结构……122
6.2 分析硬件抽象层……123
6.2.1 Overlay 系统硬件抽象层的接口……123
6.2.2 实现 Overlay 系统的硬件抽象层……125
6.2.3 实现接口……126
6.3 实现 Overlay 硬件抽象层……127

第 7 章 照相机系统……129

7.1 Camera 系统的结构……129
7.2 Camera 驱动层实现详解……131
7.2.1 V4L2 驱动程序……131
7.2.2 硬件抽象层……137
7.3 实现 Camera 系统的硬件抽象层……141
7.3.1 Java 程序部分……141
7.3.2 Camera 的 Java 本地调用部分……142
7.3.3 Camera 的本地库 libui.so……142
7.3.4 Camera 服务 libcameraservice.so……143

第 8 章 Alarm 时钟系统……147

8.1 Alarm 系统基础……147
8.2 分析 RTC 驱动程序……148
8.3 Alarm 驱动程序详解……149
8.3.1 分析文件 android_alarm.h……149
8.3.2 分析文件 alarm.c……151
8.3.3 分析文件 alarm-dev.c……160
8.4 JNI 层详解……167
8.5 Java 层详解……168
8.5.1 分析 AlarmManagerService 类……168
8.5.2 分析 AlarmManager 类……176

第 9 章 振动器系统……178

9.1 振动器系统结构……178
9.2 硬件抽象层实现详解……179
9.3 分析 Java 层部分……181
9.4 实现硬件抽象层……184

第三篇 典型应用篇

第 10 章 二维图像渲染……186

10.1 SurfaceFlinger 渲染管理器……186
10.1.1 SurfaceFlinger 基础……186
10.1.2 Surface 和 Canvas……188
10.2 Surface 渲染详解……189
10.2.1 渲染类 Surface 详解……189
10.2.2 分析 Layer 和 LayerBuffer……191
10.3 Skia 渲染引擎详解……196
10.3.1 Skia 基础……196
10.3.2 Android 中的 Skia……197
10.3.3 使用 Skia 绘图……205
10.3.4 Skia 的其他功能……206

第 11 章 绘制二维图像……208

11.1 绘图界面布局详解……208
11.1.1 View 视图组件……208
11.1.2 ViewGroup 容器……208
11.1.3 Layout 规划布局……209
11.2 Android 绘图基础……212
11.2.1 使用 Canvas 画布……212
11.2.2 使用 Paint 类……214
11.2.3 位图操作类 Bitmap……217
11.3 使用其他的绘图类……222
11.3.1 使用设置文本颜色类 Color……223
11.3.2 使用矩形类 Rect 和 RectF……224
11.3.3 非矢量图形拉伸类 NinePatch……228
11.3.4 使用变换处理类 Matrix……228
11.3.5 使用 BitmapFactory 类……231

11.3.6 使用 Region 类……233
11.3.7 使用类 Typeface……234

第 12 章 二维动画应用……235

12.1 使用 Drawable 实现动画效果……235
12.1.1 Drawable 基础……235
12.1.2 使用 Drawable 实现动画效果……236
12.2 Tween Animation 动画详解……237
12.2.1 Tween 动画基础……237
12.2.2 Tween 动画类详解……240
12.2.3 Tween 应用实战……243
12.3 实现 Frame Animation 动画效果……246
12.3.1 Frame 动画基础……246
12.3.2 使用 Frame 动画……246
12.4 Property Animation 动画……248
12.4.1 Property Animation（属性）动画基础……248
12.4.2 使用 Property Animation……250
12.5 实现动画效果的其他方法……253
12.5.1 播放 GIF 动画……254
12.5.2 实现 EditText 动画特效……256

第 13 章 渲染二维图像……257

13.1 使用渲染类 Shader……257
13.2 使用 SurfaceFlinger 渲染器……261
13.2.1 SurfaceFlinger 基础……261
13.2.2 渲染 Android 手机屏幕中的图形……263
13.3 使用 Skia 渲染引擎……265
13.3.1 Skia 基础……265
13.3.2 使用 Skia 绘图……268
13.4 通过 Skia 绘制文字……275

第 14 章 开发音频应用程序……277

14.1 音频应用接口类介绍……277
14.2 AudioManager 类……278
14.2.1 AudioManager 基础……278
14.2.2 AudioManager 基本应用——设置短信提示铃声……280
14.2.3 AudioManager 基本应用——调节手机音量的大小……284
14.3 录音处理……287
14.3.1 使用 MediaRecorder 接口录制音频……287
14.3.2 使用 AudioRecord 接口录制音频……292
14.4 播放音频……297
14.4.1 使用 AudioTrack 播放音频……297
14.4.2 使用 MediaPlayer 播放音频……300
14.4.3 使用 SoundPool 播放音频……311
14.4.4 使用 Ringtone 播放铃声……316
14.4.5 使用 JetPlayer 播放音频……318
14.4.6 使用 AudioEffect 处理音效……319
14.5 语音识别技术……321
14.5.1 Text-To-Speech 技术……322
14.5.2 谷歌的 Voice Recognition 技术……324
14.6 实现振动效果……326
14.6.1 Vibrator 类基础……326
14.6.2 使用 Vibrator 实现振动效果……327
14.7 设置闹钟……332
14.7.1 AlarmManage 基础……332
14.7.2 开发一个闹钟程序……333

第 15 章 开发视频应用程序……338

15.1 使用 MediaPlayer 播放视频……338
15.2 使用 VideoView 播放视频……344
15.2.1 VideoView 基础……344
15.2.2 使用 VideoView 播放手机中的影片……346
15.2.3 使用 VideoView 播放手机中的 MP4……348
15.2.4 开发一个网络视频播放器……350
15.3 使用 Camera 拍照……356
15.3.1 Camera 基础……356
15.3.2 总结 Camera 拍照的流程……360
15.3.3 使用 Camera 预览并拍照……362
15.3.4 使用 Camera API 方式拍照……366

第四篇 三维技术篇

第 16 章 OpenGL ES 系统初步……372

16.1 OpenGL ES 介绍……372
16.1.1 OpenGL ES 3.0 介绍……372
16.1.2 Android 全面支持 OpenGL ES 3.0……373
16.2 OpenGL ES 3.0 系统初步分析……373
16.3 分析下层的包裹库……374
16.3.1 libGLESv1_CM.so 包裹库详解……374
16.3.2 libGLESv2 包裹库详解……379

16.3.3 libEGL 包裹库详解 ……381
16.4 加载并解析 OpenGL 库 ……383
16.4.1 开始加载并解析 ……383
16.4.2 库加载器 Loader 详解 ……384
16.5 EGL 实现详解 ……388
16.5.1 分析 EGL 的数据结构 ……388
16.5.2 分析 EGL 的 API ……392
第 17 章 OpenGL ES 基本应用 ……401
17.1 OpenGL ES 的基本应用 ……401
17.1.1 使用点线法绘制三角形 ……401
17.1.2 使用索引法绘制三角形 ……405
17.1.3 使用顶点法绘制三角形 ……409
17.2 实现投影效果 ……411
17.2.1 正交投影 ……411
17.2.2 透视投影 ……411
17.2.3 正交投影和透视投影的区别 ……412
17.2.4 实现投影效果 ……412
17.3 实现光照效果 ……415
17.3.1 光源的类型 ……416
17.3.2 光源的颜色 ……416
17.3.3 开启/关闭光照 ……417
第 18 章 纹理映射 ……422
18.1 纹理映射基础 ……422
18.1.1 纹理贴图和纹理拉伸 ……422
18.1.2 Texture Filter 纹理过滤 ……423
18.2 实现三角形纹理贴图效果 ……424
18.3 实现地月模型效果 ……427
18.4 实现纹理拉伸效果 ……434
第 19 章 绘制不同的三维形状 ……438
19.1 绘制一个圆柱体 ……438
19.2 绘制一个圆环 ……444
19.3 绘制一个抛物面效果 ……448
19.4 绘制一个螺旋面效果 ……450
第 20 章 坐标变换和混合 ……454
20.1 实现坐标变换 ……454
20.1.1 坐标变换基础 ……454
20.1.2 实现缩放变换 ……454
20.1.3 实现平移变换 ……458
20.2 使用 Alpha 混合技术 ……460
20.2.1 基本知识 ……460
20.2.2 实现简单混合 ……461
20.2.3 实现光晕和云层效果 ……465
20.2.4 实现滤光器效果 ……469
第 21 章 OpenGL ES 进阶 ……474
21.1 实现摄像机和雾特效功能 ……474
21.1.1 摄像机基础 ……474
21.1.2 雾特效基础 ……475
21.1.3 实现雾特效和摄像机效果 ……475
21.2 粒子系统 ……484
21.2.1 粒子系统基础 ……484
21.2.2 实现粒子系统效果 ……484
21.3 镜像技术 ……487
21.4 实现旗帜飘扬效果 ……491

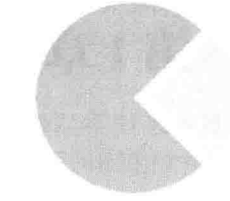
第五篇 综合实战篇

第 22 章 开发一个屏保系统 ……494
22.1 屏幕保护程序介绍 ……494
22.1.1 屏幕保护程序的作用 ……494
22.1.2 手机中的屏幕保护程序 ……494
22.2 开发屏保程序的原理 ……495
22.3 开发一个屏保程序 ……496
22.3.1 准备素材图片 ……496
22.3.2 编写布局文件 ……496
22.3.3 编写主程序文件 ……497
第 23 章 开发一个音乐播放器 ……507
23.1 项目介绍 ……507
23.1.1 项目背景介绍 ……507
23.1.2 项目的目的 ……508
23.2 系统需求分析 ……508
23.2.1 构成模块 ……508
23.2.2 系统流程 ……512
23.2.3 功能结构图 ……513
23.2.4 系统功能说明 ……514
23.2.5 系统需求 ……514
23.3 数据库设计 ……515
23.3.1 字段设计 ……515
23.3.2 E-R 图设计 ……515
23.3.3 数据库连接 ……515
23.3.4 创建数据库 ……516
23.3.5 操作数据库 ……517
23.3.6 数据显示 ……518
23.4 具体编码 ……518
23.4.1 设置服务信息 ……518
23.4.2 播放器主界面 ……519
23.4.3 播放列表功能 ……529
23.4.4 菜单功能模块 ……531

23.4.5 播放设置界面……533
23.4.6 设置显示歌词……535
23.4.7 文件浏览器模块……536
23.4.8 数据存储……539

第24章 开发一个闹钟系统……541

24.1 项目介绍……541
24.1.1 系统需求分析……541
24.1.2 构成模块……541
24.2 系统主界面……542
24.2.1 布局文件……542
24.2.2 程序文件……543
24.3 闹钟列表模块……552
24.3.1 设置主界面……552
24.3.2 设置闹钟界面……557
24.3.3 闹钟提醒模块……564
24.3.4 重复设置……569
24.3.5 闹钟数据操作……570
24.4 选择铃声音乐……573



第一篇

基础技术篇

第 1 章　Android 技术概述

第 2 章　Android 技术核心框架分析

第 3 章　获取并分析 Android 源码

第1章　Android 技术概述

Android 是一种智能手机操作系统，是建立在 Linux 开源系统基础之上的，能够迅速建立手机软件的解决方案。虽然 Android 外形比较简单，但是其功能十分强大，已经成为当前软件行业的一股新兴力量。从 2011 年开始到现在，Android 一直占据全球智能手机市场占有率第一的宝座。在本章的内容中，将简单介绍 Android 的发展历程和背景，并介绍搭建 Android 应用开发环境的基本知识，为进入本书后面知识的学习打下基础。

1.1 智能手机系统介绍

在 Android 系统诞生之前，智能手机这个新鲜事物大大丰富了人们的生活，得到了广大手机用户的青睐，各大手机厂商纷纷建立了各种智能手机操作系统来抢占市场份额。Android 系统就是在这个风起云涌的历史背景下诞生的。

1.1.1　何谓智能手机

智能手机具有像个人电脑那样强大的功能，拥有独立的操作系统，允许用户自行安装应用软件、游戏等第三方服务商提供的程序，并且通过移动通信网络接入到互联网络中。在 Android 系统诞生之前已经有很多优秀的智能手机产品，例如家喻户晓的 Symbian 系列和微软的 Windows Mobile 系列等。

1.1.2　看当前主流的智能手机系统

在当今市面中最主流的智能手机系统当属微软的 Windows Mobile、Symbian、Palm、BlackBerry、iOS 和本书的主角 Android。

1. 微软的 Windows Mobile

Windows Mobile 是微软公司的一款接触产品，Windows Mobile 将熟悉的 Windows 桌面扩展到了个人设备中。使用 Windows Mobile 操作系统的设备主要有 PC 手机、PDA、随身音乐播放器等。Windows Mobile 操作系统有 3 种，分别是 Windows Mobile Standard、Windows Mobile Professional、Windows Mobile Classic。当前的最新版本是 Windows Phone 7 和 Windows Phone 8。

2. iOS

iOS 作为苹果移动设备 iPhone 和 iPad 的操作系统，在 App Store 的推动之下，成为了世界上引领潮流的操作系统之一。原本这个系统名为“iPhone OS”，2010 年 6 月 7 日在 WWDC 大会上宣布改名为“iOS”。iOS 的用户界面的概念基础是能够使用多点触控直接操作。控制方法包括滑动、轻触开关及按键。与系统交互包括滑动（Swiping）、轻按（Tapping）、挤压（Pinching，通常

用于缩小）及反向挤压（Reverse Pinching or Unpinching，通常用于放大）。此外通过其自带的加速器，可以令其旋转设备改变其 *y* 轴以令屏幕改变方向，这样的设计令 iPhone 更便于使用。

3. Android

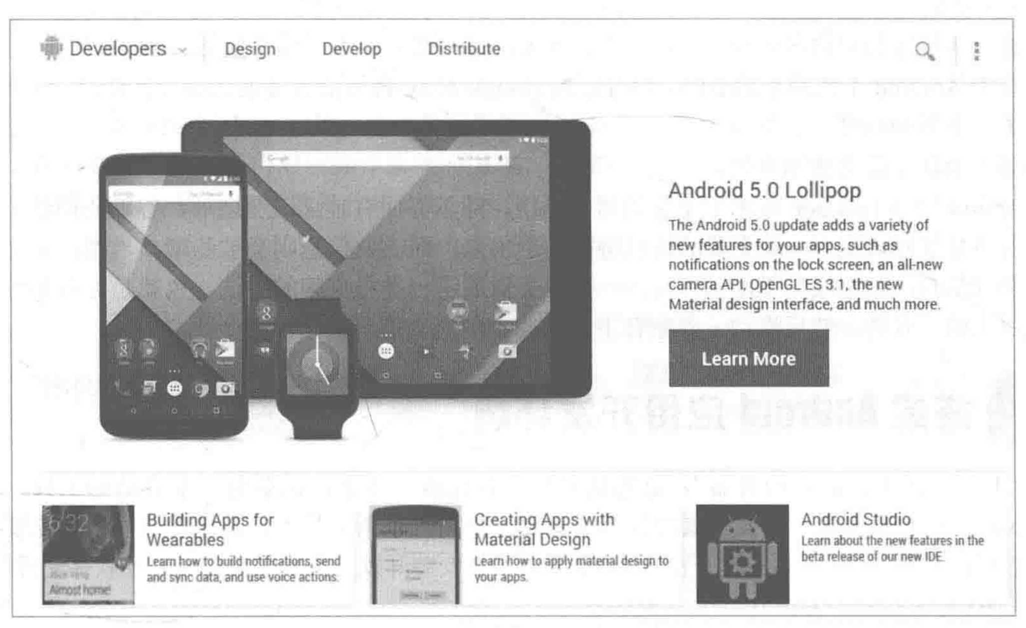
Android 是我们本书的主角，是 2007 年 11 月 5 日宣布的基于 Linux 平台的开源手机操作系统，该平台由操作系统、中间件、用户界面和应用软件组成，号称是首个为移动终端打造的真正开放和完整的移动软件。

根据国际数据公司（IDC）公布的数据，在 2013 年第一季度，Android 和 iOS 的装机量占所有智能手机出货量的 92.3%。在 2013 年前 3 个月，安装 Android 系统的新智能手机数量跃升至 1.621 亿部，大大超过去年同期的 9 030 万部。这意味着，在运往世界各地的所有新智能手机中，谷歌的移动操作系统的市场占有率已经达到 75%，比 2012 年第一季度的 59.1%有显著提高。

到本书截稿之时，Android 的最新版本是 Android L。

1.1.3 Android 5.0 的突出变化

2014 年 10 月 15 日，谷歌发布了下一代系统 Android 5.0，并将在 26 日晚些时候提供给开发者下载，如图 1-1 所示。



▲图 1-1 谷歌推出 Android 5.0

和以往版本相比，Android 5.0 版本的最突出特性如下所示。

（1）“Material”主题。

Dave Burke 表示，开发者在开发新应用时，可选择一个被称为“Material”的主题。该主题支持新的动画效果、实时 3D 阴影显示以及其他多项新功能，在 Demo 中，他使用了拨号界面做介绍，所有的操作都十分流畅。随后 Dave Burke 介绍了新的强化的通知中心，通过下滑操作，可以看到所有的通知。然后 Dave Burke 开始利用游戏介绍新的消息系统 heads up，玩游戏时，如果有电话拨打进来，屏幕顶端会出现一个通知框。如果向左右滑动手指，则可以忽略来电。这样的设计能尽量保证不打断用户的当前操作。

（2）新 Android Wear 发布。

Android 工程部总监 David Singleton 登台介绍穿戴设备的相关开发。David Singleton 通过 LG G Watch 智能手表展示 Android Wear 系统，智能手表通过振动提醒穿戴者有消息、来电。用户可上下滑动屏幕来翻页通知内容。

完整的 Android Wear SDK 将会发布，其 API 与标准版 Android API 基本一致。开发人员移植应用不存在难度。Android Wear SDK 会自动同步通知到 Android。开发者可以开发语音回复和页面回复的应用程序。

（3）Android TV 发布。

Android TV 是一套可用于电视机顶盒的系统，有相应 SDK，从这里看出谷歌对它的重视程度不亚于智能手机和平板。Android TV 需要一个 D Pad 进行语音控制，其支持 HDMI 和接收器等视频信号输入。系统本身是覆盖在视频之上的，如搜索菜单、控制菜单等。Android TV 的核心优势是搜索（基于 Google Now）。用户可以用 Android Wear 智能手表设备来控制 Android TV。Android TV 支持谷歌 Cast 技术，也就是说用户可以通过这个系统把电视当作 ChromeCast 电视棒使用。谷歌 Play 也专门开辟了 Android TV 应用类别。

（4）Android Auto 系统发布：导航、通信和音乐成核心。

谷歌发布的 Android Auto 系统面向未来汽车市场。Android Auto 的核心将是导航、通信和音乐。当 Android 智能手机与 Android Auto 系统连接时，手机屏幕能投射到车载屏幕上。Android Auto 可以进行环境感知和语音控制，它的主界面跟 Google Now 并无二致。虽然 Android Auto 可以被看作基于 Android 系统的车载 GPS，但考虑到 Google Now 自然语言搜索的强大性能，“人车对话”达到了一个新的高度。

（5）全新设计的通知系统。

Android 5.0 Lollipop 带来了全新的通知系统。除了界面有较大改变之外，谷歌还调整了通知中心的信息展示规则——最重要的信息将被显示出来，而次要信息则会被隐藏。当然，如果需要查看全部信息，则继续向下滑动即可——有些类似展示一叠扑克牌的手法，也就是你首先看到的是表面上的一张牌，然后滑动，隐藏在下方的扑克牌就会展示出来。

1.2 搭建 Android 应用开发环境

“工欲善其事，必先利其器”，意思是要想高效完成一件事，需要有一个合适的工具。对于 Android 开发人员来说，开发工具同样至关重要。作为一项新兴技术，在进行开发前首先要搭建一个对应的开发环境。而在搭建开发环境前，需要了解安装开发工具所需要的硬件和软件配置条件。

> **注意**
>
> Android 开发包括底层开发和应用开发。底层开发一般是指和硬件相关的开发，并且是基于 Linux 环境的，例如开发驱动程序。应用开发是指开发能在 Android 系统上运行的程序，如游戏、地图等程序。本书的重点是讲解多媒体应用开发，即使讲一些底层的知识，也是为上层的应用服务的。
>
> 因为开发 Android 应用程序最合适的系统是 Windows，所以本书只介绍在 Windows 下配置 Eclipse+ADT 的过程。

1.2.1 安装 Android SDK 的系统要求

在搭建之前，一定先确定安装 Android SDK 时对系统的要求，具体如表 1-1 所示。

表 1-1 安装 Android SDK 的系统要求

项 目	版本要求	说 明	备 注
操作系统	Windows XP 或 Vista；Mac OS X 10.4.8+Linux Ubuntu Drapper	根据自己的电脑自行选择	选择自己最熟悉的操作系统
软件开发包	Android SDK	选择最新版本的 SDK	截止到目前，最新手机版本是 5.0
IDE	Eclipse IDE+ADT	Eclipse3.3 (Europa), 3.4 (Ganymede)ADT(Android Development Tools)开发插件	选择“for Java Developer”
其他	JDKApache Ant	Java SE Development Kit 5 或 6 Linux 和 Mac 上使用 Apache Ant 1.6.5+, Windows 上使用 1.7+版本	单独的 JRE 是不可以的，必须要有 JDK，不兼容 Gnu Java 编译器（gcj）

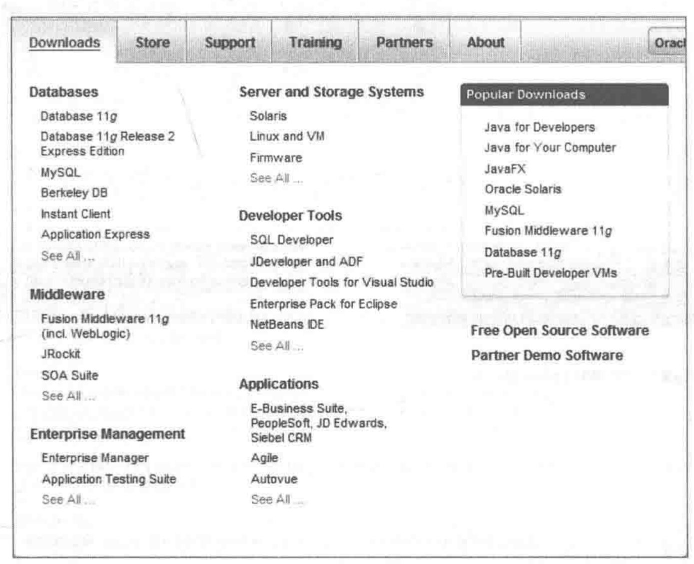
Android 工具是由多个开发包组成的，具体说明如下。

- JDK：可以到网址 http://java.sun.com/javase/downloads/index.jsp 下载。
- Eclipse（Europa）：可以到网址 http://www.eclipse.org/downloads/下载 Eclipse IDE for Java Developers。
- Android SDK：可以到网址为 http://developer.android.com 的网站下载。
- 还有对应的开发插件。

1.2.2 安装 JDK

JDK（Java Development Kit）是整个 Java 的核心，包括了 Java 运行环境、Java 工具和 Java 基础的类库。JDK 是学好 Java 的第一步，是开发和运行 Java 环境的基础，当用户要对 Java 程序进行编译的时候，必须先获得对应操作系统的 JDK，否则将无法编译 Java 程序。在安装 JDK 之前需要先获得 JDK，获得 JDK 的操作流程如下所示。

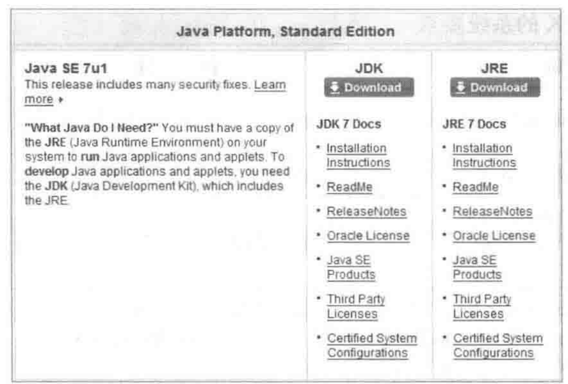
（1）登录 Oracle 官方网站，网址为 http://developers.sun.com/downloads/，如图 1-2 所示。



▲图 1-2 Oracle 官方下载页面

（2）在图 1-2 中可以看到有很多版本，例如在此选择 Java 7 版本，下载页面如图 1-3 所示。

（3）在图 1-2 中单击 JDK 下方的“Download”按钮，在弹出的新界面中选择将要下载的 JDK，笔者在此选择的是 Windows x86 版本。如图 1-4 所示。

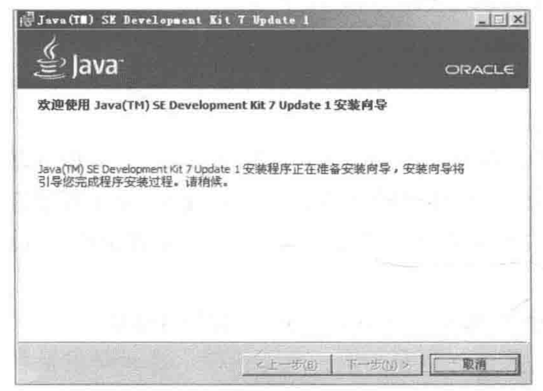


▲图 1-3　JDK 下载页面

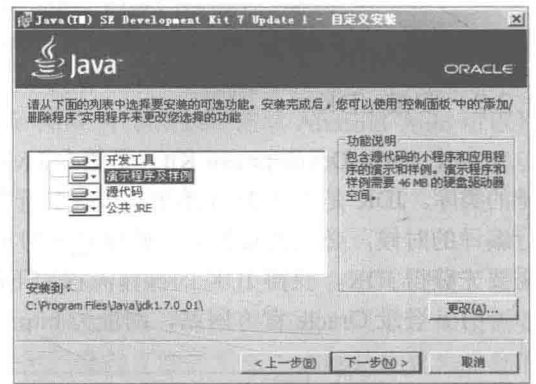
▲图 1-4　选择 Windows x86 版本

（4）下载完成后双击下载的“.exe”文件进行安装，将弹出“安装向导”对话框，在此单击“下一步”按钮。如图 1-5 所示。

（5）弹出“安装路径”对话框，在此选择文件的安装路径。如图 1-6 所示。



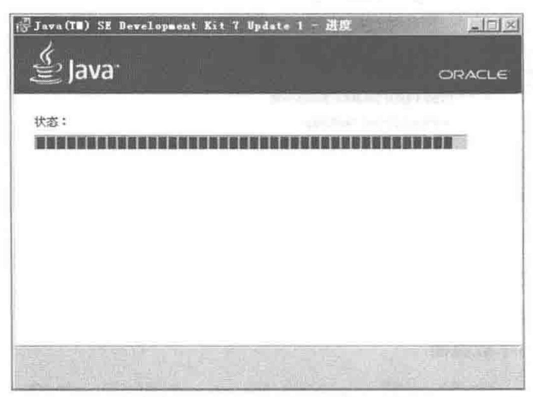
▲图 1-5　“安装向导”对话框



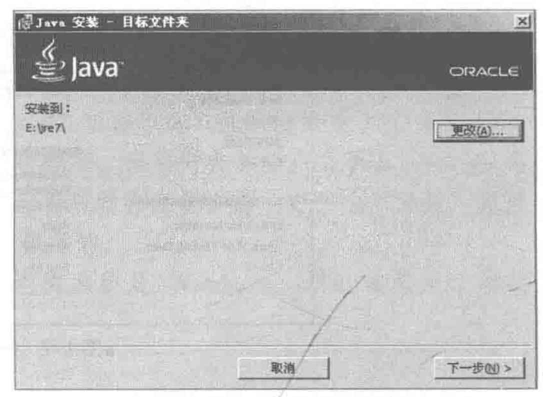
▲图 1-6　“安装路径”对话框

（6）在此设置安装路径是“E:\jdk1.7.0_01\”，然后单击“下一步”按钮开始在安装路径解压缩下载的文件。如图 1-7 所示。

（7）完成后弹出“目标文件夹”对话框，在此选择要安装的位置。如图 1-8 所示。



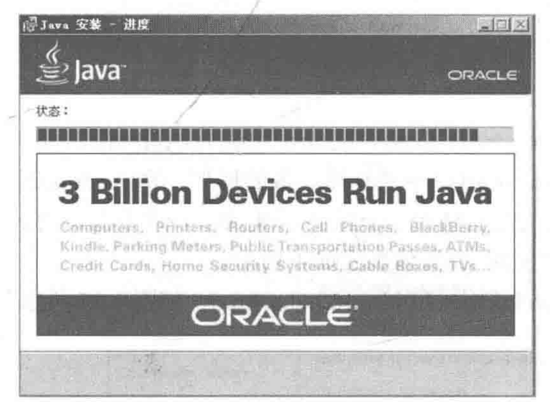
▲图 1-7　解压缩下载的文件



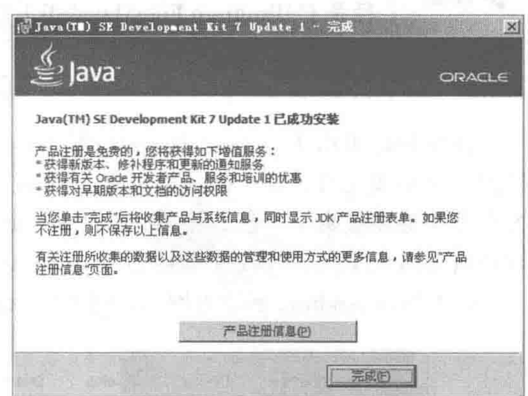
▲图 1-8　“目标文件夹”对话框

（8）单击“下一步”按钮后开始正式安装，如图 1-9 所示。

（9）完成后弹出“完成”对话框，单击“完成”按钮后完成整个安装过程。如图 1-10 所示。



▲图 1-9　正式安装



▲图 1-10　“完成”对话框

完成安装后可以检测是否安装成功，检测方法是依次单击【开始】|【运行】，在运行框中输入“cmd”并按下回车键，在打开的 CMD 窗口中输入 java –version，如果显示图 1-11 所示的提示信息，则说明安装成功。

如果检测没有安装成功，需要将其目录的绝对路径添加到系统的 PATH 中。具体做法如下所示。

（1）右键依次单击【我的电脑】|【属性】|【高级】，单击下面的“环境变量”，在下面的“系统变量”处选择“新建”，在变量名处输入 JAVA_HOME，变量值中输入刚才的目录，比如设置为“C:\Program Files\Java\jdk1.7.0_01”。如图 1-12 所示。

▲图 1-11　CMD 窗口中检测是否安装成功

▲图 1-12　设置系统变量

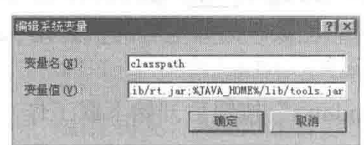
（2）再次新建一个变量名为 classpath，其变量值如下所示。

```
.;%JAVA_HOME%/lib/rt.jar;%JAVA_HOME%/lib/tools.jar
```

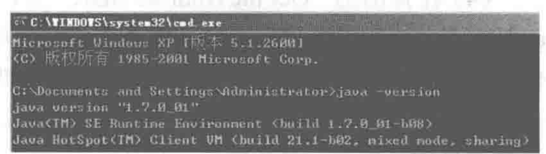
具体如图 1-13 所示。单击“确定”按钮找到 PATH 的变量，双击或单击编辑，在变量值最前面添加如下值。

```
%JAVA_HOME%/bin;
```

（3）再依次单击【开始】|【运行】，在运行框中输入“cmd”并按下回车键，在打开的 CMD 窗口中输入 java –version，如果显示图 1-14 所示的提示信息，则说明安装成功。



▲图 1-13　设置系统变量



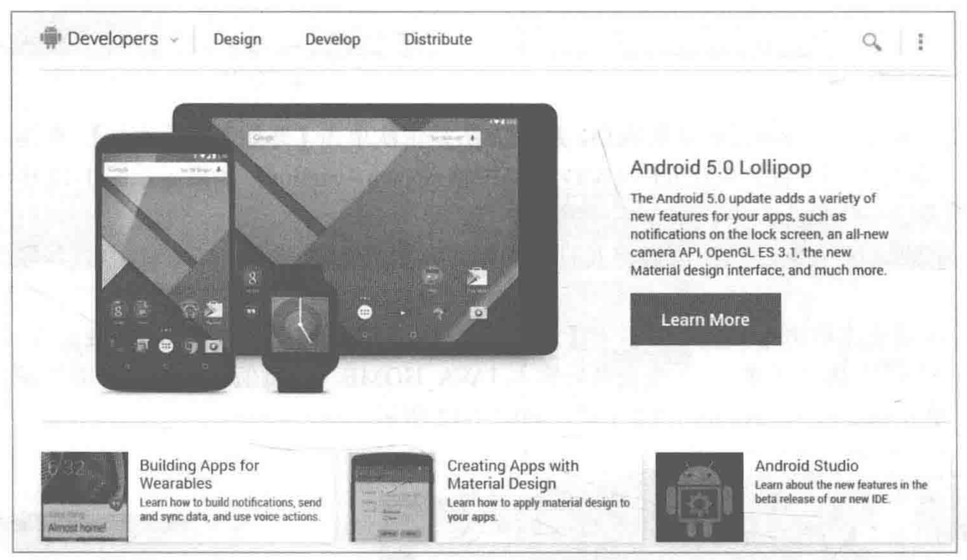
▲图 1-14　CMD 界面

> **注意**　上述变量设置中，是按照笔者本人的安装路径设置的，笔者安装的 JDK 的路径是 C:\Program Files\Java\jdk1.7.0_01。

1.2.3　获取并安装 Eclipse 和 Android SDK

在安装好 JDK 后，接下来需要安装 Eclipse 和 Android SDK。Eclipse 是进行 Android 应用开发的一个集成工具，而 Android SDK 是开发 Android 应用程序锁必须具备的框架。在 Android 官方公布的最新版本中，已经将 Eclipse 和 Android SDK 这两个工具进行了集成，一次下载即可同时获得这两个工具。获取并安装 Eclipse 和 Android SDK 的具体步骤如下所示。

（1）登录 Android 的官方网站，网址是 http://developer.android.com/index.html，如图 1-15 所示。



▲图 1-15　Android 的官方网站

（2）单击图 1-15 左上方“Developers”右边的 ⌄ 符号，在弹出的界面中单击“Tools”链接。如图 1-16 所示。

Developers	Design	Develop	Distribute
About	Get Started	Training	Google Play
Wear	Devices	API Guides	Essentials
TV	Style	Reference	Get Users
Auto	Patterns	Tools	Engage & Retain
	Building Blocks	Google Services	Monetize
	Downloads	Samples	Tools & Reference
	Videos		Developer Stories

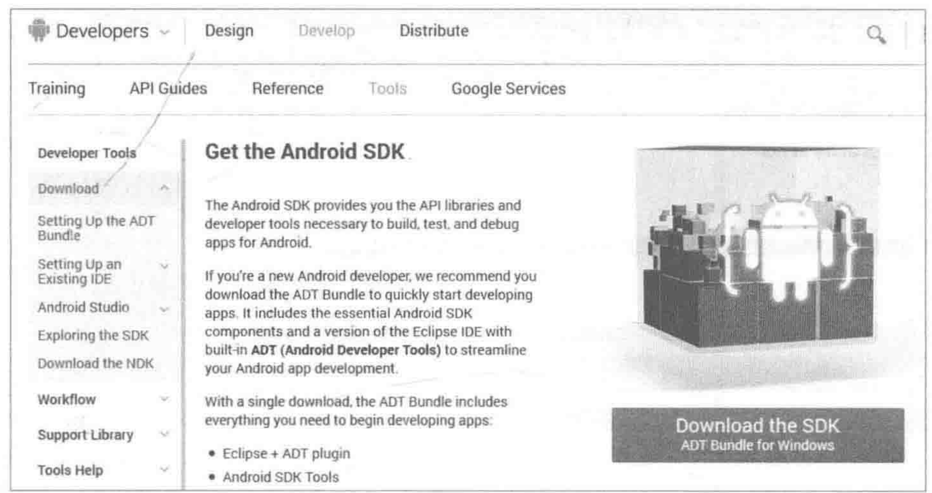
▲图 1-16　“Tools”链接

（3）在弹出的新页面中单击“Download the SDK”按钮，如图 1-17 所示。

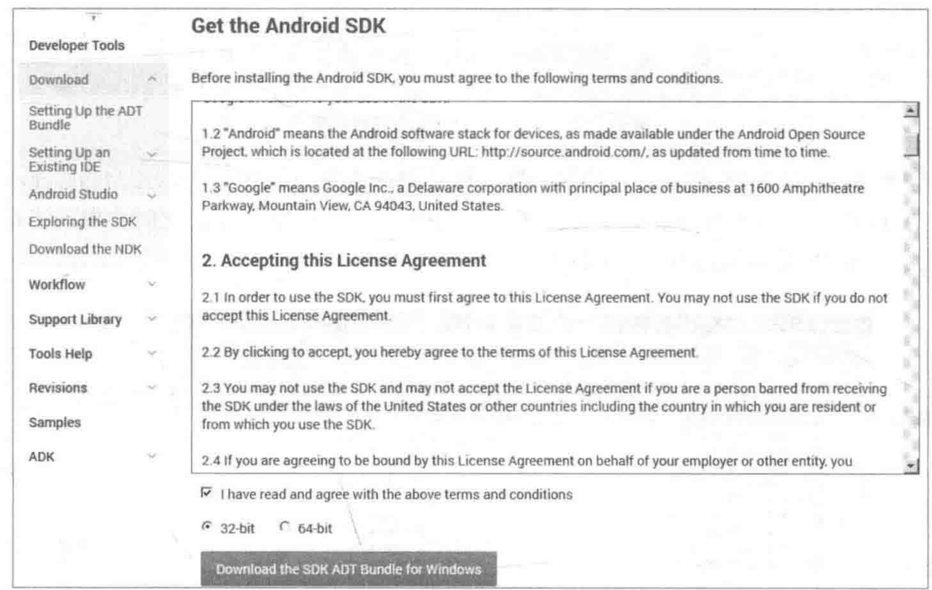
（4）在弹出的“Get the Android SDK”界面中勾选“I have read and agree with the above terms and conditions”前面的复选框，然后在下面的单选按钮中选择系统的位数。例如笔者的机器是 32 位的，所以勾选“32-bit”前面的单选按钮。如图 1-18 所示。

（5）单击图 1-18 中的“Download the SDK ADT Bundle for Windows”按钮后开始下载工作，下载的目标文件是一个压缩包。如图 1-19 所示。

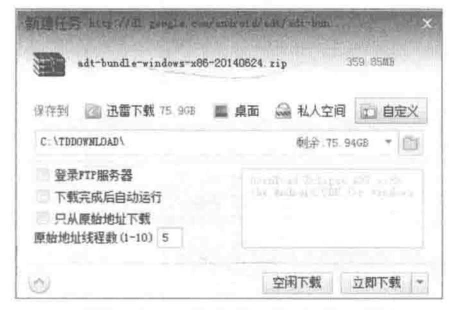
（6）将下载得到的压缩包进行解压，解压后的目录结构如图 1-20 所示。



▲图 1-17 单击“Download the SDK”按钮



▲图 1-18 “Get the Android SDK”界面

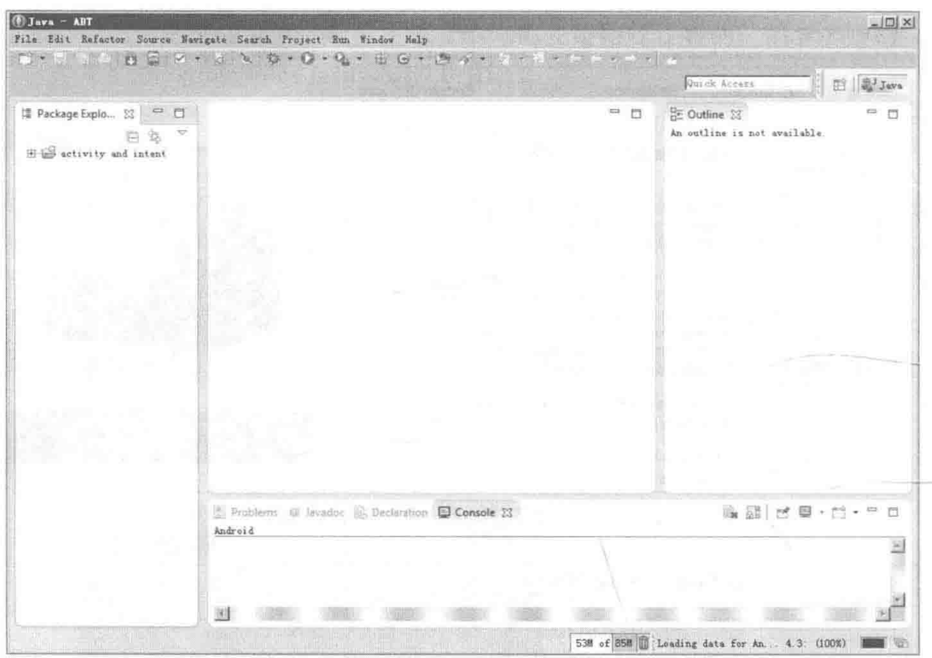


▲图 1-19 开始下载目标文件压缩包

eclipse	2014/10/14 8:51	文件夹	
sdk	2014/10/18 16:28	文件夹	
SDK Manager.exe	2014/7/3 3:24	应用程序	216 KB

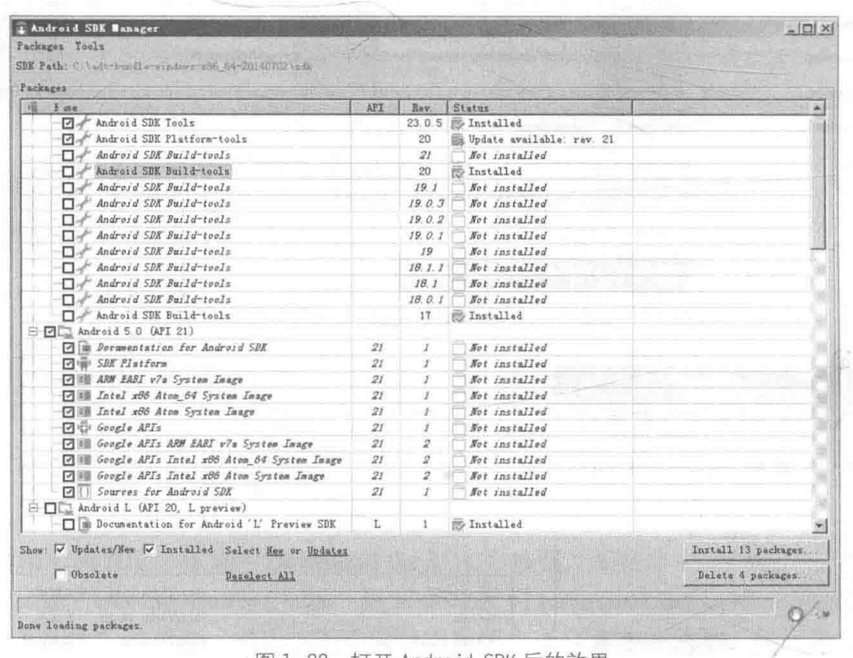
▲图 1-20 解压后的目录结构

由此可见，Android 官方已经将 Eclipse 和 Android SDK 实现了集成。双击“eclipse”目录中的“eclipse.exe”可以打开 Eclipse，界面效果如图 1-21 所示。



▲图 1-21　打开 Eclipse 后的界面效果

（7）打开 Android SDK 的方法有两种，第一种是双击下载目录中的“SDK Manager.exe”文件，第二种是在 Eclipse 工具栏中单击图标。打开后的效果如图 1-22 所示，此时会发现当前 Android SDK 的最新版本是 Android 5.0（API 21）。



▲图 1-22　打开 Android SDK 后的效果

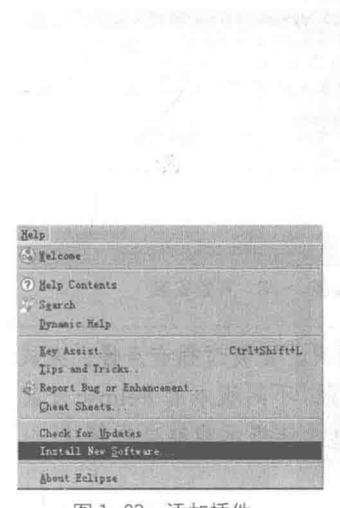
1.2.4　安装 ADT

Android 为 Eclipse 定制了一个专用插件 Android Development Tools（ADT），此插件为用户提供了一个强大的开发 Android 应用程序的综合环境。ADT 扩展了 Eclipse 的功能，可以让用户快

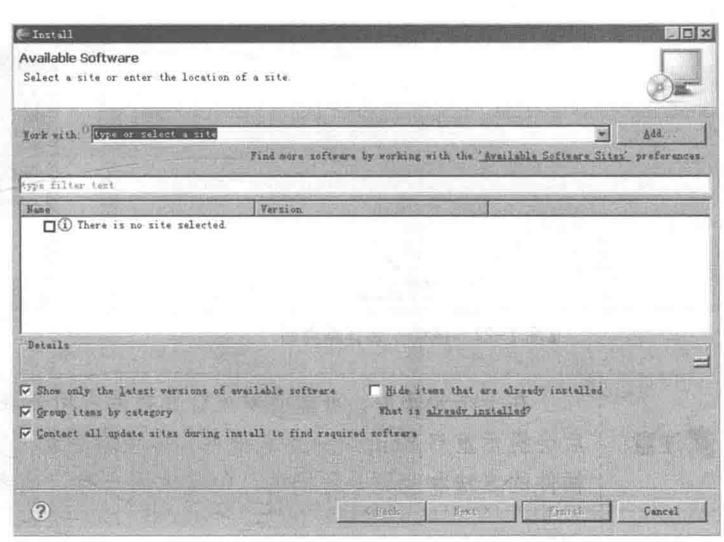
速地建立 Android 项目，创建应用程序界面。要安装 Android Development Tools plug-in，需要首先打开 Eclipse IDE。然后进行如下操作。

（1）打开 Eclipse 后，依次单击菜单栏中的【Help】|【Install New Software...】选项，如图 1-23 所示。

（2）在弹出的对话框中单击“Add”按钮，如图 1-24 所示。



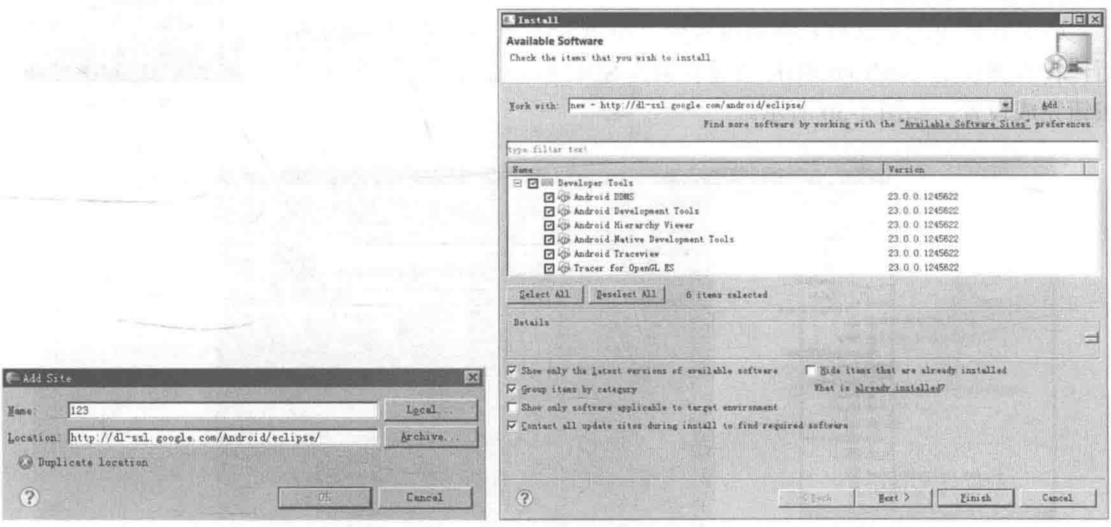
▲图 1-23 添加插件



▲图 1-24 添加插件

（3）在弹出的“Add Site”对话框中分别输入名字和地址，名字可以自己命名，例如“123”，但是在 Location 中必须输入插件的网络地址 http://dl-ssl.google.com/Android/eclipse/。如图 1-25 所示。

（4）单击“OK”按钮，此时在“Install”对话框中将会显示系统中可用的插件。如图 1-26 所示。

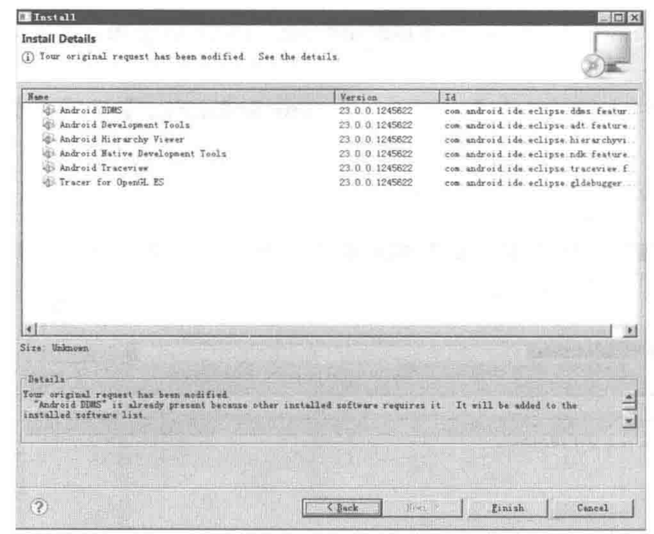


▲图 1-25 设置地址

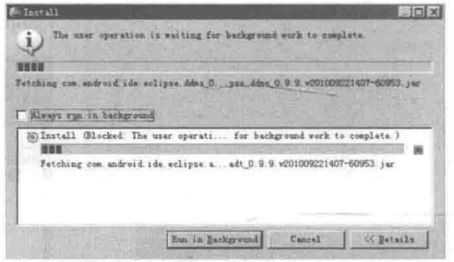
▲图 1-26 选择插件

（5）选中“Android DDMS”和“Android Development Tools”，然后单击“Next”按钮来到安装详情界面。如图 1-27 所示。

（6）单击“Finish”按钮，开始进行安装，安装进度对话框如图 1-28 所示。



▲图 1-27　插件安装详情界面



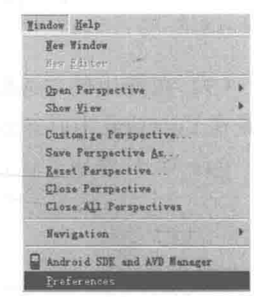
▲图 1-28　开始安装

注意　在上个步骤中，可能会发生计算插件占用资源的情况，安装过程有点慢。完成后会提示重启 Eclipse 来加载插件，等重启后就可以用了。不同版本的 Eclipse 安装插件的方法和步骤是不同的，但是大同小异，读者根据操作提示能够自行解决。

1.2.5　设定 Android SDK Home

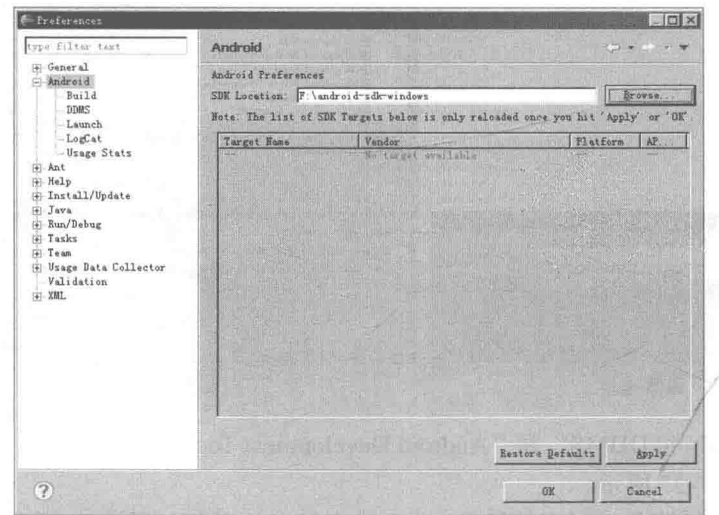
当完成上述插件装备工作后，此时还不能使用 Eclipse 创建 Android 项目，我们还需要在 Eclipse 中设置 Android SDK 的主目录。

（1）打开 Eclipse，在菜单中依次单击【Windows】|【Preferences】选项，如图 1-29 所示。



▲图 1-29 “Preferences” 选项

（2）在弹出的对话框左侧可以看到“Android”选项，选中 Android 后，在右侧设定 Android SDK 所在目录为 SDK Location，单击“OK”按钮完成设置。如图 1-30 所示。

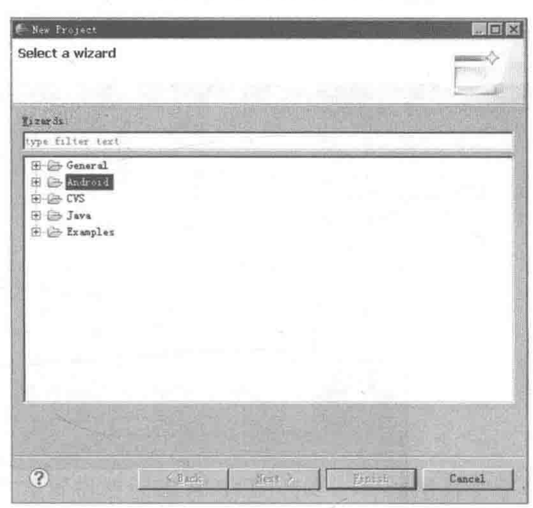


▲图 1-30 “Preferences” 对话框

1.2.6 验证开发环境

经过前面步骤的操作，一个基本的 Android 开发环境就搭建完成了。“实践是检验真理的唯一标准”，下面通过新建一个项目来验证当前的环境是否可以正常工作。

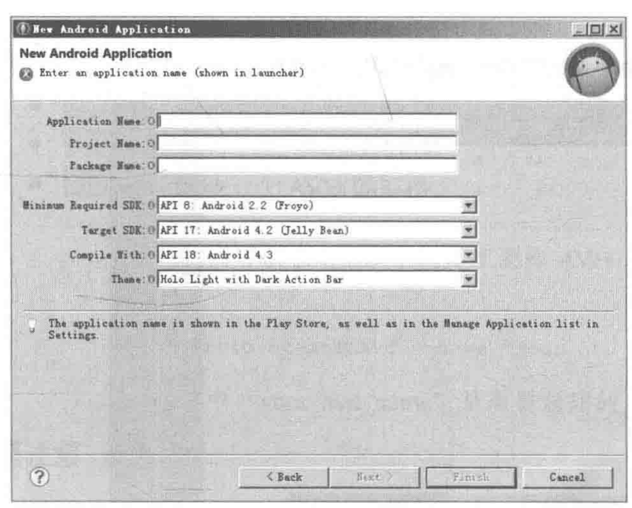
（1）打开 Eclipse，在菜单中依次选择【File】|【New】|【Project】选项，在弹出的对话框中可以看到 Android 类型的选项，如图 1-31 所示。



▲图 1-31 “New Project” 对话框

（2）在图 1-31 所示的对话框中选择“Android”，单击“Next”按钮后打开“New Android Application”对话框，在对应的文本框中输入必要的信息，如图 1-32 所示。

（3）单击“Finish”按钮后 Eclipse 会自动完成项目的创建工作，最后会看到图 1-33 所示的项目结构。



▲图 1-32 “New Android Application” 对话框

▲图 1-33 项目结构

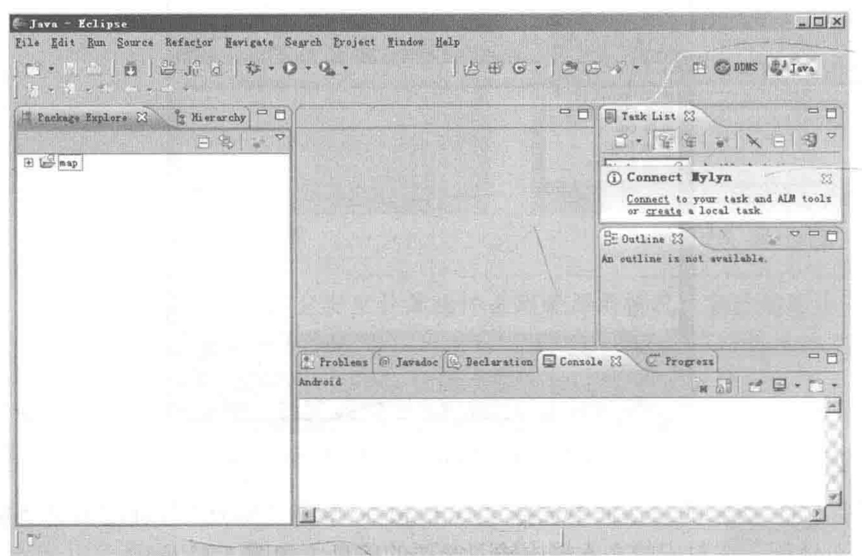
1.2.7 创建 Android 虚拟设备（AVD）

我们都知道程序开发需要调试，只有经过调试才能知道我们的程序是否正确运行。作为一个手机系统，我们怎么样在电脑平台上调试 Android 程序呢？不用担心，谷歌提供了模拟器来解决

我们担心的问题。所谓模拟器，就是指在电脑上模拟 Android 系统，可以用这个模拟器来调试并运行开发的 Android 程序。开发人员不需要一个真实的 Android 手机，只要通过电脑即可模拟运行一个手机，并开发出应用在手机上面程序。

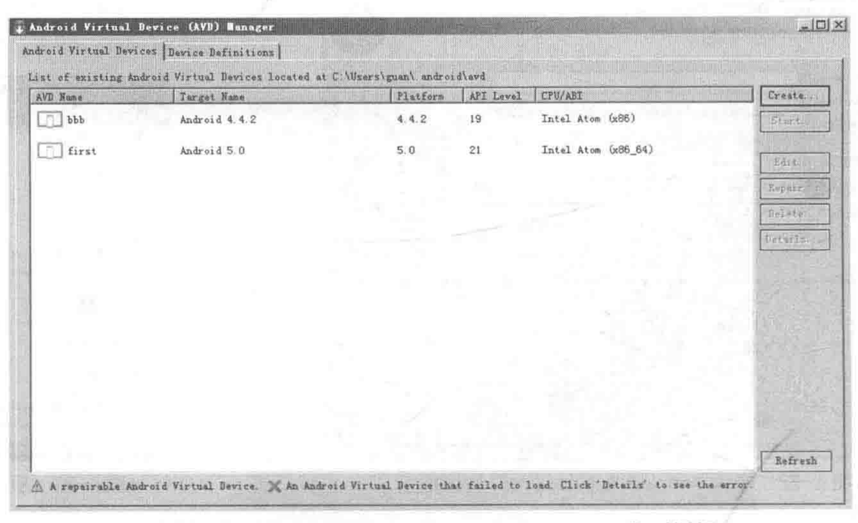
AVD 全称为 Android 虚拟设备（Android Virtual Device），每个 AVD 模拟了一套虚拟设备来运行 Android 平台，这个平台至少要有自己的内核、系统图像和数据分区，还可以有自己的 SD 卡和用户数据以及外观显示等。创建 AVD 的基本步骤如下所示。

（1）单击 Eclips 菜单中的图标，如图 1-34 所示。



▲图 1-34　Eclipse 界面

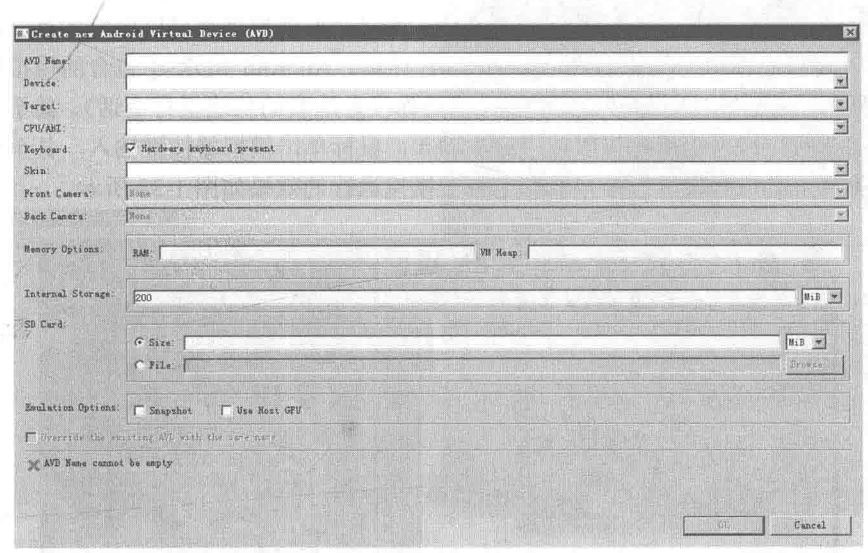
（2）在弹出的“Android Virtual Device (AVD) Manager”对话框的左侧导航中选择“Android Virtual Device”选项，如图 1-35 所示。



▲图 1-35　“Android Virtual Device (AVD) Manager”对话框

在“Android Virtual Device”列表中列出了当前已经安装的 AVD 版本，我们可以通过右侧的按钮来创建、删除或修改 AVD。主要按钮的具体说明如下所示。

● New...：创建新的 AVD，单击此按钮在弹出的对话框中可以创建一个新 AVD，如图 1-36 所示。



▲图 1-36　新建 AVD 对话框

➢ AVD Name：在此设置将要创建 AVD 的名字，可以用英文字符命名。

➢ Device：在此设置将要创建 AVD 的屏幕分辨率大小。

➢ Target：在此设置将要创建 AVD 的 API 版本，例如 Android 2.3、Android 4.0、Android 5.0 等。

➢ CPU/ABI：用于设置当前机器的 CPU。在开发低 Android SDK 版本应用程序时，使用的 Android 模拟器模拟的是 ARM 的体系结构（arm-eabi），这个模拟器并不是运行在 x86 上，而是模拟的 ARM，所以在调试程序的时候经常感觉到非常慢。针对这个问题，Intel 推出了支持 x86 的 Android 模拟器，这大大提高了启动速度和程序的运行速度，并使 Android 模拟器能够以原始速度（真机运行速度）运行在使用 Intel x86 处理器的电脑中。对于使用 Intel x86 电脑开发 Android 应用程序的开发者来说，建议在"CPU/ABI"中选择有"Intel"标识符的选项。

● Edit...：修改已经存在的 AVD。

● Delete...：删除已经存在的 AVD。

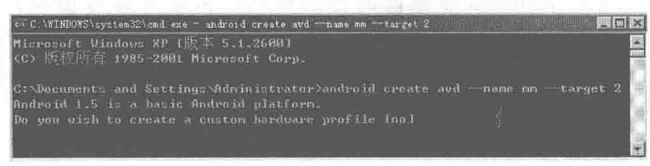
● Start...：启动一个 AVD 模拟器。

注意

我们可以在 CMD 中创建或删除 AVD，例如可以按照如下 CMD 命令创建一个 AVD。

```
android create avd --name <your_avd_name> --target <targetID>
```

其中"your_avd_name"是需要创建的 AVD 的名字，在 CMD 窗口中如图 1-37 所示。



▲图 1-37　CMD 界面

1.2.8　启动 AVD 模拟器

对于 Android 程序的开发者来说，模拟器的推出给开发者的开发和测试带来了很大的便利。无论是在 Windows 系统中还是在 Linux 系统中，Android 模拟器都可以顺利运行。官方提供了 Eclipse 插件，可以将模拟器集成到 Eclipse 的 IDE 环境。Android SDK 中包含的模拟器的功能非常齐全，电话本、通话等功能都可正常使用（当然你没办法真的从这里打电话），甚至其内置的浏览器和 Maps 都可以联网。用户可以使用键盘输入，鼠标单击模拟器按键输入，甚至还可以使用鼠标单击、拖动屏幕进行操纵。模拟器在电脑上模拟运行的效果如图 1-38 所示。

▲图 1-38　AVD 模拟器

注意

模拟器和真机究竟有何区别

当然 Android 模拟器不能完全替代真机，具体有如下差异。

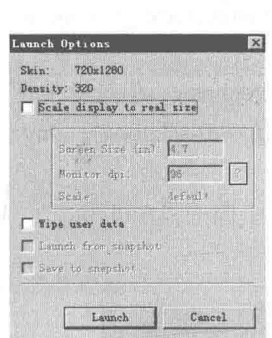
- 模拟器不支持呼叫和接听实际来电，但可以通过控制台模拟电话呼叫（呼入和呼出）;
- 模拟器不支持 USB 连接;
- 模拟器不支持相机/视频捕捉;
- 模拟器不支持音频输入（捕捉），但支持输出（重放）;
- 模拟器不支持扩展耳机;
- 模拟器不能确定连接状态;
- 模拟器不能确定电池电量水平和交流充电状态;
- 模拟器不能确定 SD 卡的插入/弹出;
- 模拟器不支持蓝牙。

有关 Andorid 模拟器的详细知识，将在本章后面的内容中进行详细介绍。

在调试的时候我们需要启动 AVD 模拟器，启动 AVD 模拟器的基本流程如下所示。

（1）选择图 1-35 列表中名为“first”的 AVD，单击 Start... 按钮后弹出“Launch Options”对话框，如图 1-39 所示。

（2）单击“Launch”按钮后将会运行名为“first”的模拟器，运行界面效果如图 1-40 所示。



▲图 1-39 "Launch Options" 对话框

▲图 1-40 Android 5.0 模拟器运行成功

注意

注意技巧——快速安装 SDK 的方法

通过 Android SDK Manager 在线安装的速度非常慢，而且有时容易挂掉。其实我们可以先从网络中寻找到 SDK 资源，用迅雷等工具下载后，将其放到指定目录后就可以完成安装。具体方法是先下载可以更新的 android-sdk-windows，然后在 android-sdk-windows 下双击 setup.exe，在更新的过程中会发现安装 Android SDK 的速率是 1kbit/s，此时打开迅雷，分别输入下面的地址。

```
https://dl-ssl.google.com/android/repository/platform-tools_r05-
windows.zip
https://dl-ssl.google.com/android/repository/docs-3.1_r01-linux.zip
https://dl-ssl.google.com/android/repository/android-2.2_r02-windows.zip
https://dl-ssl.google.com/android/repository/android-2.3.3_r01-linux.zip
https://dl-ssl.google.com/android/repository/android-2.1_r02-windows.zip
https://dl-ssl.google.com/android/repository/samples-2.3.3_r01-linux.zip
https://dl-ssl.google.com/android/repository/samples-2.2_r01-linux.zip
https://dl-ssl.google.com/android/repository/samples-2.1_r01-linux.zip
https://dl-ssl.google.com/android/repository/compatibility_r02.zip
https://dl-ssl.google.com/android/repository/tools_r11-windows.zip
https://dl-ssl.google.com/android/repository/google_apis-10_r02.zip
https://dl-ssl.google.com/android/repository/android-2.3.1_r02-linux.zip
https://dl-ssl.google.com/android/repository/usb_driver_r04-windows.zip
https://dl-ssl.google.com/android/repository/googleadmobadssd
kandroid-4.1.0.zip
https://dl-ssl.google.com/android/repository/market_licensing-r01.zip
https://dl-ssl.google.com/android/repository/market_billing_r01.zip
https://dl-ssl.google.com/android/repository/google_apis-8_r02.zip
https://dl-ssl.google.com/android/repository/google_apis-7_r01.zip
https://dl-ssl.google.com/android/repository/google_apis-9_r02.zip
………
可以继续根据自己开发要求选择不同版本的 API
```

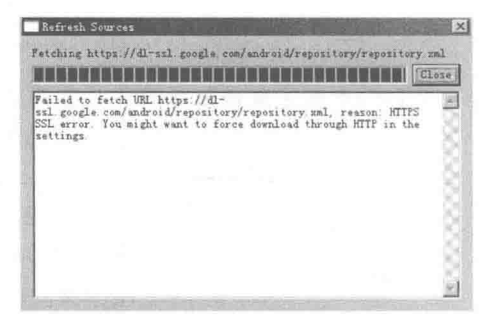
下载完后将它们复制到"android-sdk-windows/Temp"目录下，然后再运行 setup.exe，勾选需要的 API 选项，会发现马上就安装好了。记得把原始文件保留好，因为放在 temp 目录下的文件装好后立刻就消失。

1.2.9　解决搭建环境过程中的常见问题

下面将总结在搭建 Android SDK 环境时出现过的问题。

1. 不能在线更新

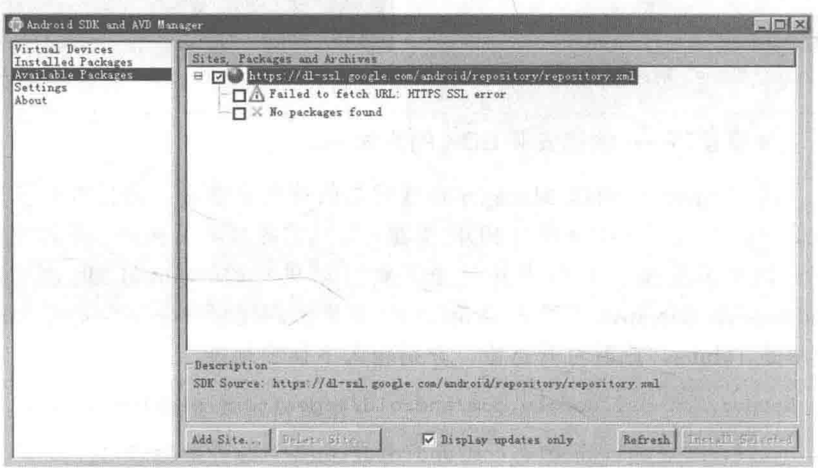
在安装 Android 后，需要更新为最新的资源和配置。但是在启动 Android 后，经常会不能更新，弹出图 1-41 所示的提示。



▲图 1-41　不能更新的提示

Android 默认的在线更新网址是：https://dl-ssl.google.com/android/eclipse/，但是经常会出现错误。如果此网址不能更新，可以自行设置更新地址，修改为：http://dl-ssl.google.com/android/repository/repository.xml。具体操作方法如下。

（1）单击 Android 左侧的“Available Packages”选项，然后单击下面的“Add Site...”按钮。如图 1-42 所示。

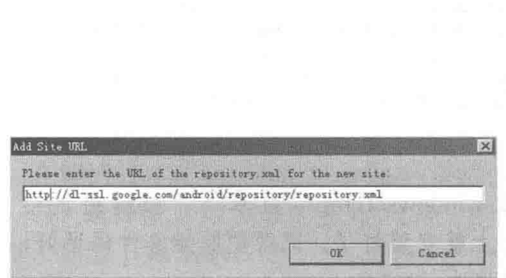


▲图 1-42　“Available Packages”界面

（2）在弹出的“Add Site URL”对话框中输入下面修改后的地址，如图 1-43 所示。

```
http://dl-ssl.google.com/android/repository/repository.xml
```

（3）单击“OK”按钮后完成设置工作，此时就可以使用更新功能了。如图 1-44 所示。



▲图 1-43　“Add Site URL”对话框

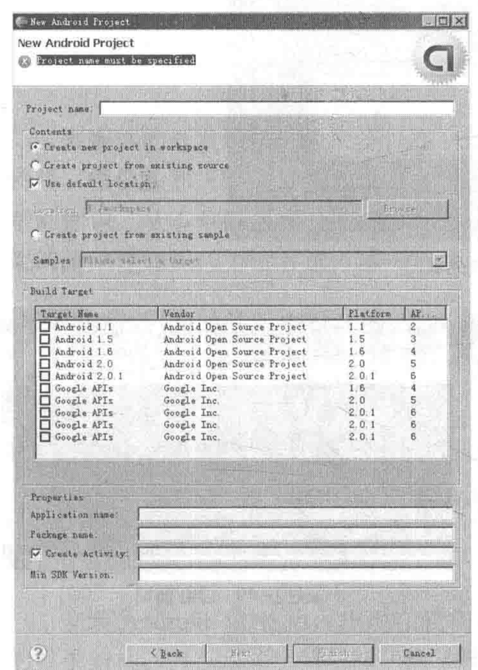
▲图 1-44　更新地址后的“Available Packages”界面

2. 显示“Project name must be specified”提示

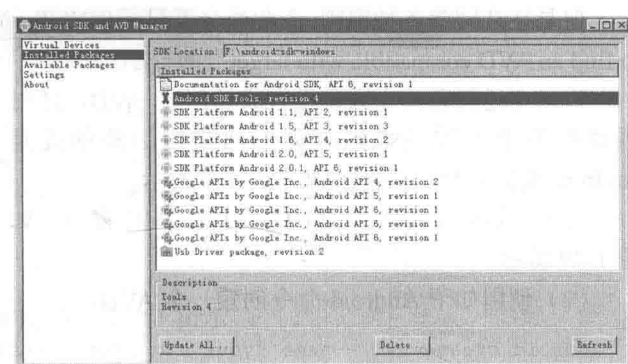
很多初学者在 Eclipse 中新创建 Android 工程时，经常会遇到“Project name must be specified”提示的问题，如图 1-45 所示。

造成上述问题的原因是 Android 没有更新完成，需要进行完全更新。具体方法如下所示。

（1）打开 Android，选择左侧的“Installed Packages”，如图 1-46 所示。

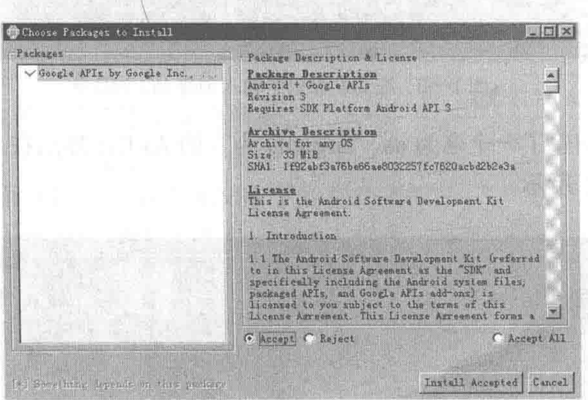


▲图 1-45 “Project name must be specified”提示



▲图 1-46 “Installed Packages”界面

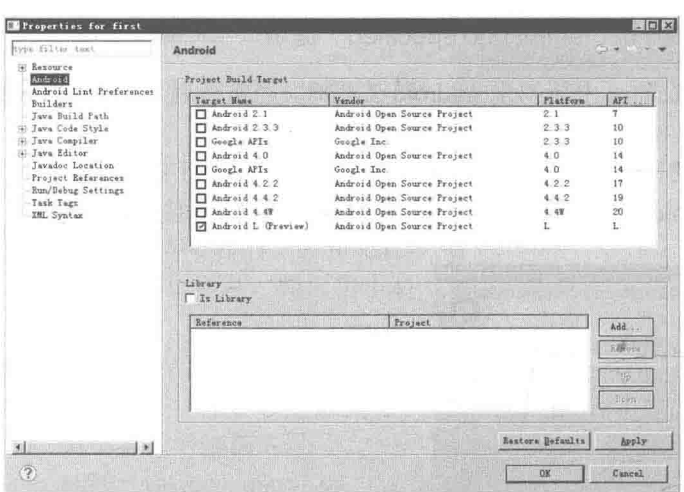
（2）在图 1-46 中的右侧列表中选择“Android SDK Tools, revision4”，在弹出的对话框中选择“Accept”，最后单击“Install Accepted”按钮开始更新。如图 1-47 所示。



▲图 1-47 “Choose Packages to Install”对话框

3. Target 列表中没有 Target 选项

通常来说，当 Android 开发环境搭建完毕后，在 Eclipse 工具栏中依次单击【Window】|【Preference】，单击左侧的“Android”选项后会显示存在的 SDK Targets。如图 1-48 所示。



▲图 1-48　SDK Targets 列表

但是往往因为各种原因，在此处会不显示 SDK Targets 列表，图 1-45 中就是如此，并输出“Failed to find an AVD compatible with target”的提示。

造成上述问题的原因是没有成功创建 AVD，此时需要我们手工安装来解决这个问题，当然前提是 Android 更新完毕。具体解决方法如下所示。

（1）在运行中键入“CMD”，打开 CMD 窗口。如图 1-49 所示。

▲图 1-49　CMD 窗口

（2）使用如下 Android 命令创建一个 AVD。

```
android create avd --name <your_avd_name> --target <targetID>
```

其中“your_avd_name”是需要创建的 AVD 的名字，在 CMD 窗口中如图 1-50 所示。

▲图 1-50　在 CMD 窗口中输入创建 AVD 的命令

图 1-50 的窗口中创建了一个名为 aa，targetID 为 3 的 AVD，然后在 CMD 界面中输入“n”，即完成操作。如图 1-51 所示。

▲图 1-51　在 CMD 窗口中完成创建 AVD 的操作

第 2 章　Android 技术核心框架分析

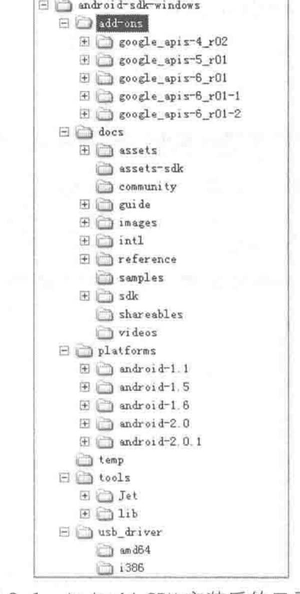
多媒体的开发领域比较广泛，在 Android 系统中的多媒体主要包括图形图像、音频、视频、铃声、摄像头和三维渲染等知识。在学习这些知识之前，读者需要了解一些和多媒体开发相关的基础知识。从本章的内容开始，将简要讲解 Android 体系的组成，介绍底层接口和驱动的应用框架的基础性知识，为读者进入本书后面知识的学习打下基础。

2.1 简析 Android 安装文件

当我们下载并安装 Android SDK 后，会在安装目录中看到一些安装文件。这些文件究竟是干什么用的呢？带着疑问和好奇之心，我们一起走上学习 Android 安装文件的旅程。

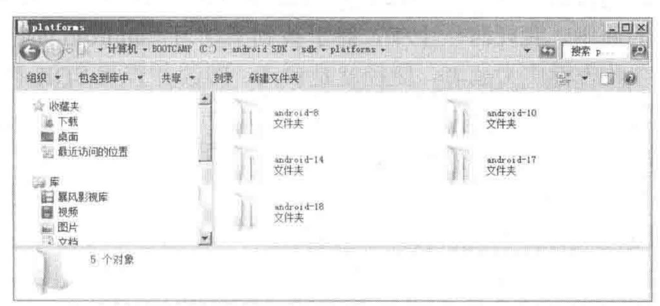
2.1.1 Android SDK 目录结构

安装 Android SDK 后，出现在我们面前的是图 2-1 所示的目录结构。



▲图 2-1　Android SDK 安装后的目录结构

- add-ons：里面包含了官方提供的 API 包，例如常用的 Google Map API（谷歌地图）。
- docs：里面包含了帮助文档和说明文档。
- platforms：里面包含了针对每个版本的 SDK 版本，提供了和其对应的 API 包以及一些示例文件，其中包含了各个版本的 Android，如图 2-2 所示。
- temp：里面包含了一些常用的文件模板。
- tools：里面包含了一些通用的工具文件。
- usb_driver：里面包含了 amd64 和 x86 下的驱动文件。
- SDK Setup.exe：Android 的启动文件。



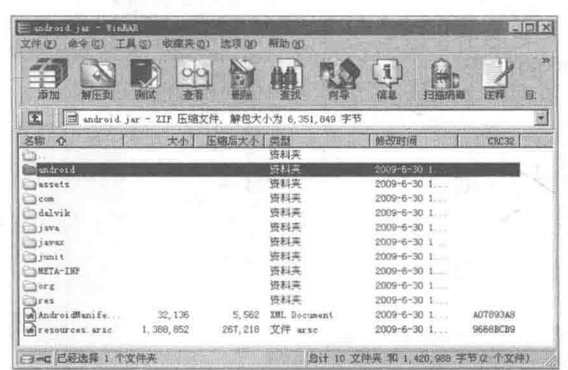
▲图 2-2　platforms 目录项

2.1.2　android.jar 及内部结构

在“platforms”目录下的每个 Android 版本中，都有一个名为“android.jar”的压缩包。例如在笔者机器中，“platforms\android-18”目录中的内容如图 2-3 所示。

▲图 2-3　android.jar 文件所在目录

“android.jar”里面包含了编译后的压缩文件和所有有用的 API。另外，在它强大的外表下却有一颗温暖的心，我们只需使用 Windows 系统上的解压缩工具即可打开它。为了探其究竟，决定打开“android.jar”压缩包，打开后的内部结构分别如图 2-4 和图 2-5 所示。



▲图 2-4　android.jar 文件结构（1）

▲图 2-5　android.jar 文件结构（2）

注意　上述各个文件，对于我们研究 Android 应用开发没有多大帮助。但是对大家了解 Android 运行机制和内核却有很大帮助。

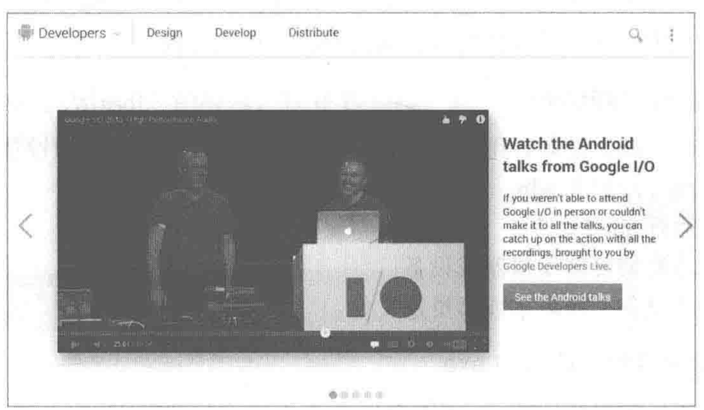
2.1.3　阅读 SDK 帮助文档

在我们解压缩文件“android.jar”之后，就可以了解其内部 API 的包结构和组织方式。如果要深入理解各个文件包内包含的 API 和对应的具体用法，我们必须花费一定的精力和时间来研究它。本着学习编程要执着的原则，我耗时一上午，终于找到了学习“android.jar”的捷径。本着乐于分享学习经验的原则，在此告诉广大读者，捷径就是参考、阅读并查找 SDK 帮助文档。希望大家也具备我的分享精神，尽情与外人道出。

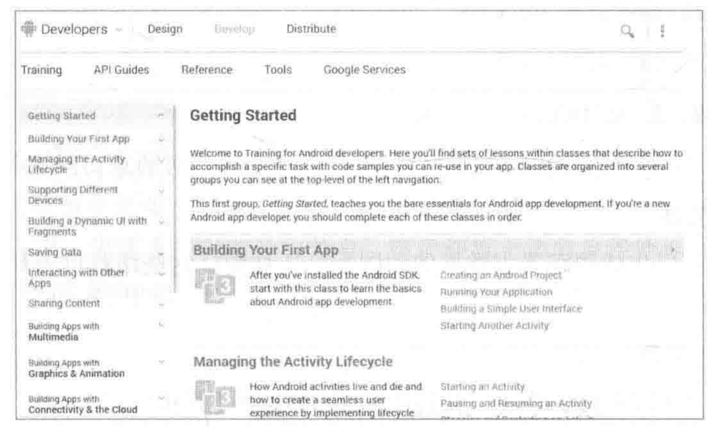
打开 SDK 帮助文档的方法非常简单，我们可以使用浏览器打开“docs”目录下的文件 index.html，如图 2-6 所示。然后单击顶部“Developers”中的“Training”链接可以来到一个新页面，这个网页就是 SDK 帮助文档的学习主页，界面效果如图 2-7 所示。

在图 2-7 所示的页面中，介绍了 Android 基本概念和当前常用版本。此 SDK 文件对于初学者来说十分重要，可以帮助读者解决很多常见的问题，是一个很好的学习文档和帮助文档。在图 2-7

所示页面中，左侧是目录索引链接，单击某个链接后可以在右侧界面中显示对应的说明信息。如果要想迅速地理解一个问题或知识点，可以在搜索对话框中通过输入关键字的方式进行快速检索。也许有很多读者存在“英语水平有限，看不懂帮助文档”的疑问，其实大家不必担心，因为有很多热心的程序员和学者对这个帮助文档进行了翻译，大家可以从网络中获取免费的中文版帮助文档。



▲图 2-6　SDK 文档主页



▲图 2-7　SDK 帮助文档的学习界面

2.1.4　常用的 SDK 工具

都说“相爱没有那么简单”，其实 Android SDK 也同样不简单。在前面搭建 Android 开发环境时，我们已经接触到了 Android SDK 中的一些开发工具，例如 AVD 模拟器。但是 SDK 很不一般，它集成了很多其他有用的开发工具，这些工具能够帮助我们在 Android 平台上开发出有用的应用程序。接下来将和大家一起领略 Android SDK 中这些有用的开发工具，请大家记住它们的名字和长相吧，因为它们会在以后的开发生涯中帮助我们完成很多任务。

1. Android 模拟器

模拟器是运行在计算机上的虚拟移动设备，有关模拟器的基本知识已经在本书的第 1 章中进行了详细介绍，在此不再讲解。

2. 集成开发插件 ADT

Android 为 Eclipse 定制了一个插件，即 Android Development Tools（ADT），这个插件为用户提供一个强大的综合环境用于开发 Android 应用程序。ADT 扩展了 Eclipse 的功能，可以让用户快速地建立 Android 项目，创建应用程序界面，在基于 Android 框架 API 的基础上添加组件，以

及用 SDK 工具集调试应用程序，甚至导出签名（或未签名）的 APKs 以便发行应用程序。

3. 调试监视服务 ddms.bat

调试监视服务 ddms.bat 集成在 Dalvik（Android 平台的虚拟机）中，用于管理运行在模拟器或设备上的进程，并协助调试工作。它可以去除一些进程，选择一个特定的程序来调试，生成跟踪数据，查看堆和线程数据，对模拟器或设备进行屏幕快照等操作。

4. Android 调试桥 adb.exe

Android 调试桥（adb）是多种用途的工具，该工具可以帮助我们管理设备或模拟器的状态。可以通过下面的几种方法加入 adb。

（1）在设备上运行 shell 命令；

（2）通过端口转发来管理模拟器或设备；

（3）从模拟器或设备上拷贝来或拷贝走文件。

5. Android 资源打包工具 aapt.exe

此工具可以创建 apk 文件，在 apk 文件中包含了 Android 应用程序的二进制文件和资源文件。

6. Android 接口描述语言 aidl.exe

此工具用于生成进程间接口代码。

7. SQLite3 数据库 sqlite3.exe

Android 可以创建和使用 SQLite 数据文件。开发人员和用户可随意访问这些 SQLite 数据文件。

8. 跟踪显示工具

跟踪显示工具可以生成跟踪日志数据的的图形分析视图，这些跟踪日志数据由 Android 应用程序产生。

9. 创建 SD 卡工具

创建 SD 卡工具用于创建磁盘镜像，此镜像可以在模拟器上模拟外部存储卡，例如常见的 SD 卡。

10. DX 工具（dx.bat）

DX 工具将 class 字节码重写为 Android 字节码（被存储在 dex 文件中）。

11. 生成 Ant 构建文件工具（activitycreator.bat）

activitycreator.bat 是一个脚本，用于生成 Ant 构建文件。Ant 构建文件用于编译 Android 应用程序，如果在安装 ADT 插件的 Eclipse 环境下开发，则就不需要这个脚本了。

12. Android 虚拟设备

在 Android SDK1.5 版以后的 Android 开发中，必须创建至少一个 AVD。AVD 全称为 Android 虚拟设备（Android Virtual Device），每个 AVD 模拟了一套虚拟设备来运行 Android 平台。这个平台至少要有自己的内核、系统图像和数据分区，还可以有自己的 SD 卡和用户数据以及外观显示等。

2.2 演示官方实例

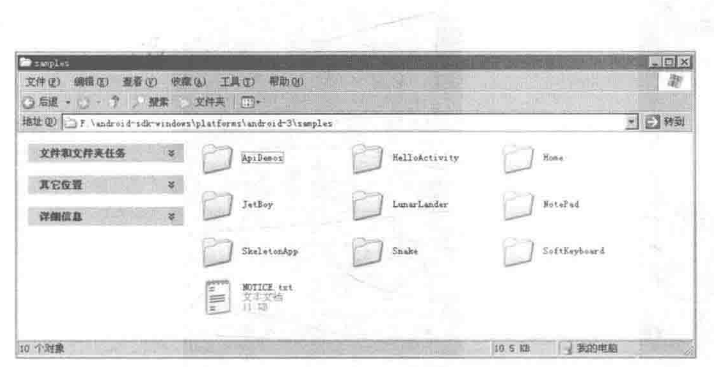
Android 官方为学习人员和开发者提供了大量的演示实例。在 Android 安装后的目录中有一个名为“samples”的子目录，在里面保存了 SDK 中的几个演示实例。这些实例从不同的方面展示

了 SDK 的特性，例如“android-3”目录中的实例文件结构如图 2-8 所示。

在接下来的内容中，将带领大家浏览图 2-8 中各个实例的效果，目的是让大家体会 Android 的功能的强大，相信大家肯定会发出“原来 Android 可以实现这么牛的效果”的感叹。

1. HelloActivity

这和编程语言中的 Hello World 程序类似，是一个在 Android 平台上的最简单程序，运行后将在手机上显示出“Hello,World!”的提示。打开 Eclipse，将“HelloActivity”导入，然后查看执行后的效果，效果如图 2-9 所示。



▲图 2-8 演示实例结构

Hello, Activity!

Hello, World!

2:30 PM

▲图 2-9 “HelloActivity”的执行效果

注意

在查看安装目录中的“samples”实例时，不能使用“Import”将实例导入到 Eclipse 中。要查看实例的运行效果，需要按照下面的步骤操作。

（1）在 Eclipse 中依次单击【file】|【new】|【android project】后弹出“New Android Project”对话框。在里面选择“Create project from existing source”选项，然后单击“Browse”按钮，并选择对应的实例文件夹即可。如图 2-10 所示。

▲图 2-10 “New Android Project”对话框

（2）单击“Finish”按钮完成操作，这样就可以将实例程序成功导入到 Eclipse 中。

2. SkeletonApp

本实例展示了如何在 Android 中使用提供的视图组件的方法，例如常见的 EditRext、Button、ImageView 和菜单等，并且还演示了如何操作这些组件。其执行后的效果如图 2-11 所示。

3. ApiDemos

ApiDemos 演示了很多 API 的使用方法，包括 app、content、graphic、media 等，如图 2-12 所示。

在图 2-12 中可以选择查看具体的分类，从而进一步了解 API 的强大功能。

▲图 2-11　SkeletonApp 的执行效果

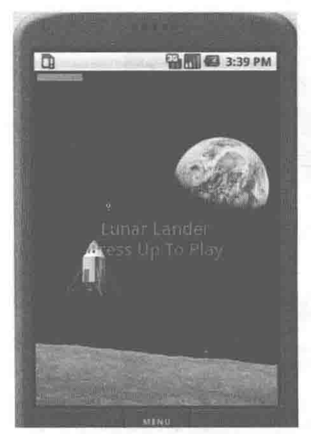
▲图 2-12　ApiDemos 的执行效果

4. LunarLander

这是一个登月游戏实例，演示了一个类似于登陆月球的小游戏，可以通过方向键和点火时机控制画面上的飞船，其执行效果如图 2-13 所示。

5. NotePad

NotePad 是一个记事本程序，此程序可以实现新建、编辑和删除等文档操作。本实例应用了 SQLite 的数据存储和编辑，并使用了 ContentProvider 等方面的信息。其执行后效果如图 2-14 所示。



▲图 2-13　LunarLander 的执行效果

▲图 2-14　NotePad 的执行效果

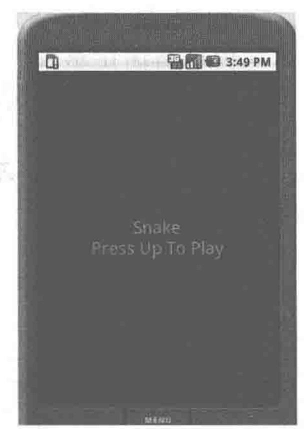
6. Snake

Snake 是贪吃蛇演示实例。这是一款经典的游戏，使用手机方向键可以对游戏进行控制。其

执行后效果如图 2-15 所示。

7. Home

Home 是一款主题类软件实现的实例，实现了一套新的主题界面。此实例演示了如何开发主题类应用，读者通过这个实例可以轻松掌握主题类开发的步骤和一些注意事项。其执行后的效果如图 2-16 所示。



▲图 2-15　Snake 的执行效果

▲图 2-16　Home 的执行效果

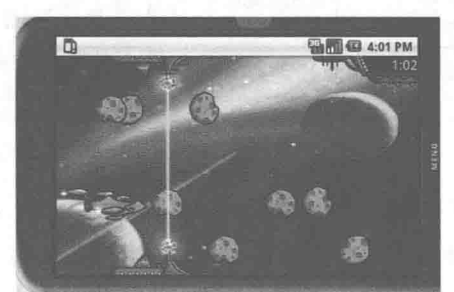
8. SoftKeyboard

SoftKeyboard 是一个软键盘实例。此实例演示了如何将软键盘绑定到输入框输入事件上。当焦点到输入框上时会自动显示软键盘。其执行后效果如图 2-17 所示。

9. JetBoy

JetBoy 是一款具备声音支持的游戏实例。它模拟演示了如何在游戏中集成 SONiVOX 的 audioINSIDE 技术的方法，此技术是 SONiVOX 捐赠给手机联盟的。此实例可以完美地播放背景音乐和场景，实现子弹击碎飞来障碍物等效果。其执行后效果如图 2-18 所示。

▲图 2-17　SoftKeyboard 的执行效果



▲图 2-18　JetBoy 的执行效果

到此为止，在 Android 安装目录中自带的实例文件就介绍完毕了。希望大家仔细品味每个演示实例的具体效果。如果具备了一定的编程基础，特别是 Java 基础，可以尝试阅读每个实例的具体实现代码，为进入本书后面知识的学习打下基础。

2.3 剖析Android系统架构

为了更加深入理解Android系统的精髓，初学者很有必要了解Android系统的整体架构，了解它的具体组成。只有这样才能知道Android究竟能干什么，我们所要学的是什么。

2.3.1 Android体系结构介绍

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

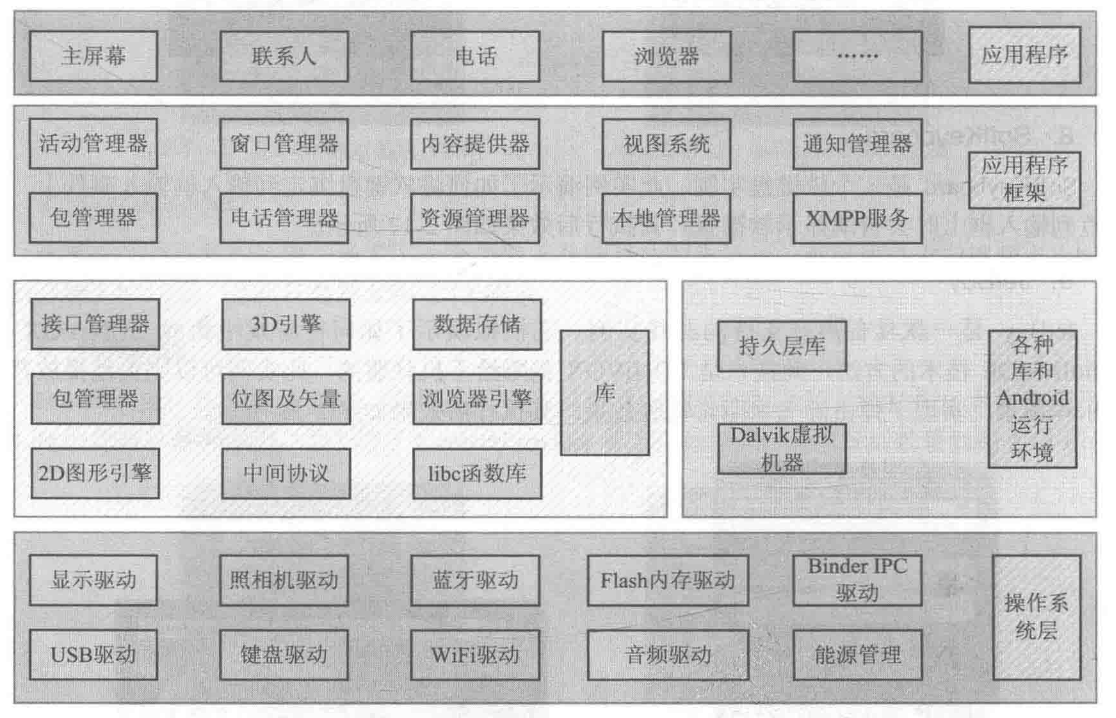
（1）操作系统层（OS）；

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

（3）应用程序框架（Application Framework）；

（4）应用程序（Application）。

上述各个层的具体结构如图2-19所示。



▲图2-19 Android操作系统的组件结构图

1. 操作系统层（OS）——最底层

因为Android源于Linux，所以使用Linux内核作为底层操作系统。Linux是一种标准的技术，也是一个开放的操作系统。Android对操作系统的使用包括核心和驱动程序两部分。Android的Linux核心为标准的Linux内核。Android更多的是需要一些与移动设备相关的驱动程序，主要的驱动如下所示。

- 显示驱动（Display Driver）：常用基于Linux的帧缓冲（Frame Buffer）驱动。
- Flash内存驱动（Flash Memory Driver）：是基于MTD的Flash驱动程序。

- 照相机驱动（Camera Driver）：常用基于 Linux 的 v4l（Video for）驱动。
- 音频驱动（Audio Driver）：常用基于 ALSA（Advanced Linux Sound Architecture，高级 Linux 声音体系）驱动。
- WiFi 驱动（WiFi Driver）：基于 IEEE 802.11 标准的驱动程序。
- 键盘驱动（KeyBoard Driver）：作为输入设备的键盘驱动。
- 蓝牙驱动（Bluetooth Driver）：基于 IEEE 802.15.1 标准的无线传输技术。
- Binder IPC 驱动：是 Android 一个特殊的驱动程序，具有单独的设备节点，提供进程间通信的功能。
- 能源管理（Power Management）：管理电池电量等信息。

2. 各种库（Libraries）和 Android 运行环境（RunTime）——中间层

本层对应一般嵌入式系统，相当于中间件层。本层分成两个部分：一个是各种库，另一个是 Android 运行环境。本层的内容大多是使用 C 语言实现的。其中包含的各种库如下所示。

- C 库：C 语言的标准库，也是系统中一个最为底层的库，C 库通过 Linux 的系统调用来实现。
- 多媒体框架（MediaFrameword）：这部分内容是 Android 多媒体的核心部分，基于 Packet Video（即 PV）的 OpenCORE。本库从功能上一共分为两大部分，一部分是音频、视频的回放（PlayBack），另一部分则是音频、视频的记录（Recorder）。
- SGL：2D 图像引擎。
- SSL：即 Secure Socket Layer，位于 TCP/IP 协议与各种应用层协议之间，为数据通信提供安全支持。
- OpenGL ES：提供了对 3D 的支持。
- 界面管理工具（Surface Management）：提供了管理显示子系统等功能。
- SQLite：一个通用的嵌入式数据库。
- WebKit：网络浏览器的核心。
- FreeType：位图和矢量字体的功能。

Android 的各种库一般是以系统中间件的形式提供的，它们均有的一个显著特点就是与移动设备的平台的应用密切相关。

Android 运行环境主要是指的 Dalvik 虚拟机。Dalvik 虚拟机和一般 Java 虚拟机（Java VM）不同，它执行的不是 Java 标准的字节码（Bytecode），而是 Dalvik 可执行格式（.dex）中的执行文件。在执行的过程中，每一个应用程序即一个进程（Linux 的一个 Process）。二者最大的区别在于 Java VM 是基于栈（Stack-based）的虚拟机，而 Dalvik 是基于寄存器（Register-based）的虚拟机。显然，后者最大的好处在于可以根据硬件实现更大的优化，这更适合移动设备的特点。

3. 应用程序框架（Application Framework）

Android 的应用程序框架为应用程序层的开发者提供 API，它实际上是一个应用程序的框架。由于上层的应用程序是以 Java 构建的，因此本层首先包含了 UI 程序中所需要的各种控件，例如 Views（视图系统），其中又包括了 List（列表）、Grid（栅格）、Text Box（文本框）、Button（按钮）等，甚至一个嵌入式的 Web 浏览器。

作为一个基本的 Andoid 应用程序，可以利用应用程序框架中的以下 5 个部分来构建。

- Activity（活动）；
- Broadcast Intent Receiver（广播意图接收者）；
- Service（服务）；

- Content Provider（内容提供者）；
- Intent and Intent Filter（意图和意图过滤器）。

本书的目的是讲解 Android 网络应用开发的知识，这方面的内容在结构图中和应用程序（Application）相对应，所以读者们需要重点关注应用程序框架（Application Framework）的知识，这些都是用 Java 开发的。当然也需要掌握一些其他层的相关知识，例如底层的内核和驱动等知识。

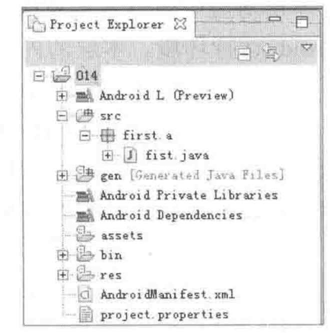
4. 应用程序（Application）

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

2.3.2　Android 应用工程文件组成

讲解完 Android 的整体结构之后，接下来讲解 Android 应用工程文件的组成。因为学习本书的目的就是开发 Android 网络应用项目，而每个 Android 应用项目是用 Eclipse 创建的工程，所以有必要了解一个 Android 应用工程文件的结构。

在 Eclipse 中，一个基本的 Android 应用工程文件组成如图 2-20 所示。



▲图 2-20　Android 应用工程文件组成

1. src

src 里面保存了开发人员编写的程序文件。和一般的 Java 项目一样，“src”目录下保存的是项目的所有包及源文件（.java），“res”目录下包含了项目中的所有资源。例如程序图标（drawable）、布局文件（layout）和常量（values）等。不同的是，在 Java 项目中没有“gen”目录，也没有每个 Android 项目都必须有的 AndroidManfest.xml 文件。

“.java”格式文件是在建立项目时自动生成的，这个文件是只读模式，不能更改。R.java 文件是定义该项目所有资源的索引文件。例如下面是某项目中 R.java 文件的代码。

```
package com.yarin.Android.HelloAndroid;
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

在上述代码中定义了很多常量，这些常量的名字都与 res 文件夹中的文件名相同，这再次证明.java 文件中所存储的是该项目所有资源的索引。有了这个文件，在程序中使用资源将变得更加方便。由于这个文件不能被手动编辑，所以当我们在项目中加入了新的资源时，只需要刷新一下该项目，.java 文件便自动生成了所有资源的索引。

2. AndroidManfest.xml

文件 AndroidManfest.xml 是一个控制文件，在里面包含了该项目中所使用的 Activity、Service 和 Receiver。例如下面是某项目中文件 AndroidManfest.xml 的代码。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
     package="com.yarin.Android.HelloAndroid"
     android:versionCode="1"
     android:versionName="1.0">
   <application android:icon="@drawable/icon"
android:label="@string/app_name">
       <activity android:name=".HelloAndroid"
                 android:label="@string/app_name">
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER" />
           </intent-filter>
       </activity>
   </application>
   <uses-sdk android:minSdkVersion="9" />
</manifest>
```

在上述代码中，intent-filter 描述了 Activity 启动的位置和时间。每当一个 Activity（或者操作系统）要执行一个操作时，它将创建出一个 Intent 的对象，这个 Intent 对象可以描述你想做什么，想处理什么数据，数据的类型，以及一些其他信息。Android 会和每个 Application 所暴露的 intent-filter 的数据进行比较，找到最合适的 Activity 来处理调用者所指定的数据和操作。下面我们来仔细分析 AndroidManfest.xml 文件，如表 2-1 所示。

表 2-1　AndroidManfest.xml 分析

参　数	说　明
manifest	根节点，描述了 package 中所有的内容
xmlns:android	包含命名空间的声明 xmlns:android=http://schemas.android.com/apk/res/android，使得 Android 中各种标准属性能在文件中使用，提供了大部分元素中的数据
Package	声明应用程序包
application	包含 package 中 application 级别组件声明的根节点。此元素也可包含 application 的一些全局和默认的属性，如标签、icon、主题、必要的权限等。1 个 manifest 能包含 0 个或 1 个此元素（不能大于 1 个）
android:icon	应用程序图标
android:label	应用程序名字
activity	Activity 是与用户交互的主要工具，是用户打开一个应用程序的初始页面。大部分被使用到的其他页面也由不同的 Activity 所实现，并声明在另外的 Activity 标记中。注意，每一个 Activity 必须有一个<activity>标记对应，无论它给外部使用或是只用于自己的 package 中。如果一个 Activity 没有对应的标记，你将不能运行它。另外，为了支持运行时查找 Activity，可包含一个或多个<intent-filter>元素来描述 Activity 所支持的操作
android:name	应用程序默认启动的 Activity
intent-filter	声明了指定的一组组件支持的 Intent 值，从而形成了 Intent Filter。除了能在此元素下指定不同类型的值，属性也能放在这里来描述一个操作所需的唯一的标签、icon 和其他信息
action	组件支持的 Intent action
category	组件支持的 Intent Category。这里指定了应用程序默认启动的 Activity
uses-sdk	该应用程序所使用的 SDK 版本相关

3. 常量定义文件

下面我们看看在资源文件中对常量的定义，例如文件 String.xml 的代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
 <string name="hello">Hello World, HelloAndroid!</string>
   <string name="app_name">HelloAndroid</string>
</resources>
```

上述常量定义文件的代码非常简单，只定义了两个字符串资源。请不要小看上面的几行代码，它们的内容很“露脸”，里面的字符直接显示在手机屏幕中，就像动态网站中的 HTML 一样。

4. 布局文件

布局（layout）文件一般位于“res\layout\main.xml”目录中，通过其代码能够生成一个显示界面。例如下面的代码。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
</LinearLayout>
```

在上述代码中，有以下几个布局和参数。

- <LinearLayout>、</LinearLayout>：在这个标签中，所有元件都是由上到下排队排成的。
- android:orientation：表示这个介质的版面配置方式是从上到下垂直地排列其内部的视图。
- android:layout_width：定义当前视图在屏幕上所占的宽度，fill_parent 即填充整个屏幕。
- android:layout_height：定义当前视图在屏幕上所占的高度，fill_parent 即填充整个屏幕。
- wrap_content：随着文字栏位的不同而改变这个视图的宽度或高度。

在上述布局代码中，使用了一个 TextView 来配置文本标签 Widget（构件），其中设置的属性 android:layout_width 为整个屏幕的宽度，android:layout_height 可以根据文字来改变高度，而 android:text 则设置了这个 TextView 要显示的文字内容，这里引用了@string 中的 hello 字符串，即 String.xml 文件中的 hello 所代表的字符串资源。hello 字符串的内容“Hello World, HelloAndroid!”就是我们在 HelloAndroid 项目运行时看到的字符串。

> **注意**　上面介绍的文件只是主要文件，在项目中需要我们自行编写。在项目中还有很多其他文件，那些文件很少是需要我们编写的，所以在此就不进行讲解了。

2.4 简述五大组件

一个典型的 Android 应用程序通常由 5 个组件组成，这 5 个组件构成了 Android 的核心功能。在本节的内容中，将一一讲解这五大组件的基本知识，为读者进入本书后面知识的学习打下基础。

2.4.1 用 Activity 来表现界面

Activity 是这 5 个组件中最常用的一个组件。程序中 Activity 的表现形式通常是一个单独的界

面（screen）。每个 Aactivity 都是一个单独的类，它扩展实现了 Activity 基础类。这个类显示为一个由 Views 组成的用户界面，并响应事件。大多数程序有多个 Activity。例如一个文本信息程序有以下几个界面：显示联系人列表界面，写信息界面，查看信息界面或者设置界面等。每个界面都是一个 Activity。切换到另一个界面就是载入一个新的 Activity。某些情况下，一个 Activity 可能会给前一个 Activity 返回值，例如一个让用户选择相片的 Activity 会把选择的相片返回给调用者。

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

2.4.2 用 Intent 和 IntentFilter 实现切换

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

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

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

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

2.4.3 Service 为你服务

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

1. 如何使用服务

在 Android 系统中有以下两种使用服务的方法。

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

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

2. Service 的生命周期

在 startService()后，即使调用 startService()的进程结束了，Service 仍然存在，一直到有进程调用 stoptService()或者 Service 自己灭亡（stopSelf()）为止。

在调用 bindService()后，Service 就和调用 bindService()的进程同生共死，也就是说如果调用 bindService()的进程死了，那么它绑定的 Service 也要跟着结束，当然期间也可以调用 unbindService()让 Service 结束。

当混合使用上述两种方式时，例如同时调用 startService()和 bindService()，那么只有同时调用 stoptService()和 unbindService()，这个 Service 才会结束。

3. 进程生命周期

Android 系统将会尝试保留那些启动了的或者绑定了的服务进程，具体说明如下所示。

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

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

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

2.4.4 用 BroadcastIntentReceiver 发送广播

当要执行一些与外部事件相关的代码时，如来电响铃时或者半夜时就可能用到 IntentReceiver。尽管 IntentReceivers 使用 NotificationManager 来通知用户一些好玩的事情发生，但是没有 UI。IntentReceivers 可以在文件 AndroidManifest.xml 中声明，也可以使用 Context.registerReceiver()来声明。当一个 IntentReceiver 被触发时，如果需要系统自然会自动启动程序。程序也可以通过 Context.broadcastIntent()给其他程序发送自己的 Intent 广播。

2.4.5 用 ContentProvider 存储数据

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

2.5 进程和线程

进程和线程很容易理解。我们电脑中有一个进程管理器，打开它后，会显示当前运行的所有程序。同样，在 Android 中也有进程。当某个组件第一次运行的时候，Android 会启动一个进程，在默认情况下，所有的组件和程序运行在这个进程和线程中，也可以安排组件在其他的进程或者线程中运行。

2.5.1 先看进程

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

当更加常用的进程无法获取足够内存时，Android 会智能地关闭不常用的进程，当下次启动程序的时候会重新启动这些进程。当决定哪个进程需要被关闭的时候，Android 会考虑哪个对用

户更加有用。例如 Android 会倾向于关闭一个长期不显示在界面的进程来支持一个经常显示在界面的进程。是否关闭一个进程决定于组件在进程中的状态。

2.5.2 再看线程

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

线程是通过 Java 的标准对象 Thread 来创建的，在 Android 中提供了以下方便地管理线程的方法。

（1）Looper 在线程中运行一个消息循环；

（2）Handler 传递一个消息；

（3）HandlerThread 创建一个带有消息循环的线程；

（4）Android 让一个应用程序在单独的线程中，指导它创建自己的线程；

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

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

（7）可以创建一个新的线程来执行长期操作。

2.5.3 应用程序的生命周期

自然界的事物都有自己的生命周期，例如人的生、老、病、死。作为一个 Android 应用程序也与自然界的生物一样，有自己的生命周期。我们开发一个程序的目的是为了完成一个功能，例如银行计算加息的软件，每当一个用户去柜台办理取款业务时，银行工作人员便启动了我们这个程序的生命，当用这个软件完成利息计算时，这个软件当前的任务就完成了，此时就需要结束自己的使命。肯定有人会提出疑问：生生死死多么麻烦，就让这个程序一直是“活着”的状态，一个用户办理完取款业务后，继续等着下一个用户办理取款业务，这样这个程序就“长生不老”了。其实谁都想自己的程序“长生不老”，但是很不幸，我们不能这样做。原因是计算机的处理性能是一定的，针对一个人、二个人、三个人，计算机可以处理这个任务。但是一个安装这个软件的机器一天会处理成千上万个取款业务，如果它们都一直活着，一台配置有限的计算机能承受得了吗？

由此可见，应用程序的生命周期就是一个程序的存活时间，即在啥时间内有效。Android 是一个构建在 Linux 之上的开源移动开发平台，在 Android 中，多数情况下每个程序都是在各自独立的 Linux 进程中运行的。当一个程序或其某些部分被请求时，它的进程就“出生”了；当这个程序没有必要再运行下去且系统需要收回这个进程的内存用于其他程序时，这个进程就“死亡”了。可以看出，Android 程序的生命周期是由系统控制而非程序自身直接控制。这和我们编写桌面应用程序时的思维有一些不同，一个桌面应用程序的进程也是在其他进程或用户请求时被创建，但是往往是在程序自身收到关闭请求后执行一个特定的动作（比如从 main 函数中返回）而导致进程结束的。要想做好某种类型的程序或者某种平台下的程序的开发，最关键的就是要弄清楚这种类型的程序或整个平台下的程序的一般工作模式，并将其熟记在心。在 Android 中，程序的生命周期控制就是属于这个范畴。

开发者必须理解不同的应用程序组件，尤其是 Activity、Service 和 Intent Receiver，需要了解这些组件是如何影响应用程序的生命周期的。如果不正确地使用这些组件，可能会导致系统终止正在执行重要任务的应用程序进程。

一个常见的进程生命周期漏洞的例子是 Intent Receiver（意图接收器），当 Intent Receiver 在 onReceive 方法中接收到一个 Intent（意图）时，它会启动一个线程，然后返回。一旦返回，系统

将认为 Intent Receiver 不再处于活动状态，因而 Intent Receiver 所在的进程也就不再有用了（除非该进程中还有其他的组件处于活动状态）。因此，系统可能会在任意时刻终止该进程以收回占有的内存。这样进程中创建出的那个线程也将被终止。解决这个问题的方法是从 Intent Receiver 中启动一个服务，让系统知道进程中还有处于活动状态的工作。为了使系统能够正确决定在内存不足时应该终止哪个进程，Android 根据每个进程中运行的组件及组件的状态把进程放入一个“Importance Hierarchy”（重要性分级）中。

进程的类型有多种多样，按照重要的程度主要包括如下几类进程。

（1）前台进程（Foreground）。

前台进程是看得见的，与用户当前正在做的事情密切相关，不同的应用程序组件能够通过不同的方法将它的宿主进程移到前台。在如下的任何一个条件下系统会把进程移动到前台。

- 进程正在屏幕的最前端运行一个与用户交互的活动（Activity），它的 onResume 方法被调用；
- 进程有一个正在运行的 Intent Receiver（它的 IntentReceiver.onReceive 方法正在执行）；
- 进程有一个服务（Service），并且在服务的某个回调函数（Service.onCreate、Service.onStart 或 Service.onDestroy）内有正在执行的代码。

（2）可见进程（Visible）。

可见进程也是可见的，它有一个可以被用户从屏幕上看到的活动，但不在前台（它的 onPause 方法被调用）。假如前台的活动是一个对话框，以前的活动就隐藏在对话框之后，就会显现这种进程。可见进程非常重要，一般不允许被终止，除非是为了保证前台进程的运行而不得不终止它。

（3）服务进程（Srvice）。

服务进程是无法看见的，拥有一个已经用 startService()方法启动的服务。虽然用户无法直接看到这些进程，但它们做的事情却是用户所关心的（如后台 MP3 回放或后台网络数据的上传、下载）。所以系统将一直运行这些进程，除非内存不足以维持所有的前台进程和可见进程。

（4）后台进程（Background）。

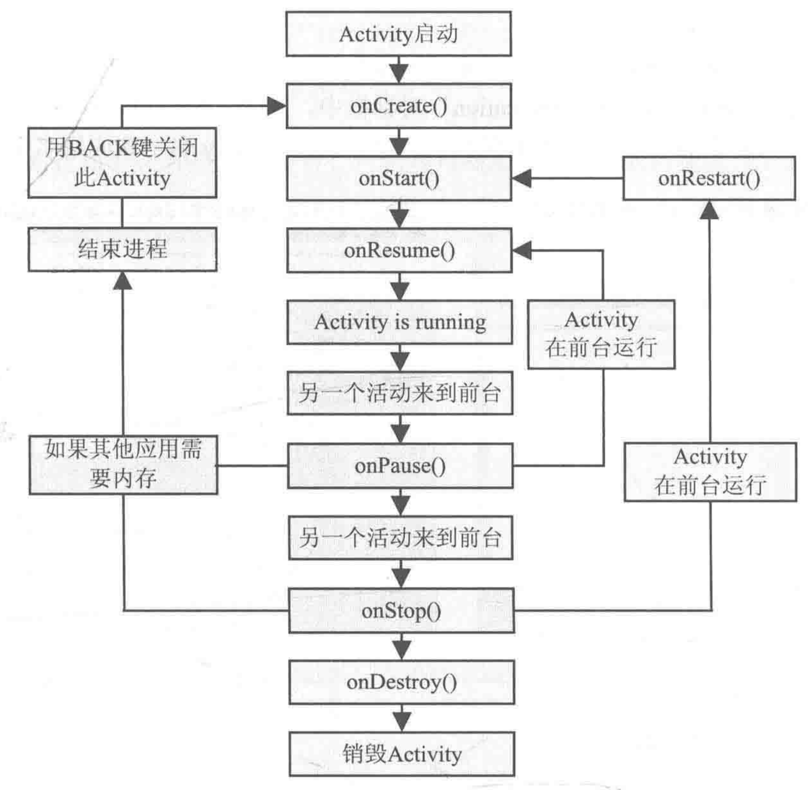
后台进程也是看不见的，只有打开之后才能看见。例如迅雷下载，我们可以将其最小化，虽然在桌面上看不见了，但是它一直在进行下载的工作。这些进程拥有一个当前用户看不到的活动（它的 onStop()方法被调用），对用户体验没有直接的影响。如果它们正确执行了活动生命周期，系统可以在任意时刻终止该进程以收回内存，并提供给前面 3 种类型的进程使用。系统中通常有很多这样的进程在运行，因此要将这些进程保存在 LRU 列表中，以确保当内存不足时用户最近看到的进程被最后终止。

（5）空进程（Empty）。

空进程是指不拥有任何活动的应用程序组件的进程。保留这种进程的唯一原因是在下次应用程序的某个组件需要运行时，不需要重新创建进程，这样可以提高启动速度。系统将以进程中当前处于活动状态组件的重要程度为基础对进程进行分类。进程的优先级可能也会根据该进程与其他进程的依赖关系而增长。假如进程 A 通过在进程 B 中设置 Context.BIND_AUTO_CREATE 标记或使用 ContentProvider 被绑定到一个服务（Service），那么进程 B 在分类时至少要被看成与进程 A 同等重要。

例如 Activity 的状态转换图如图 2-21 所示。

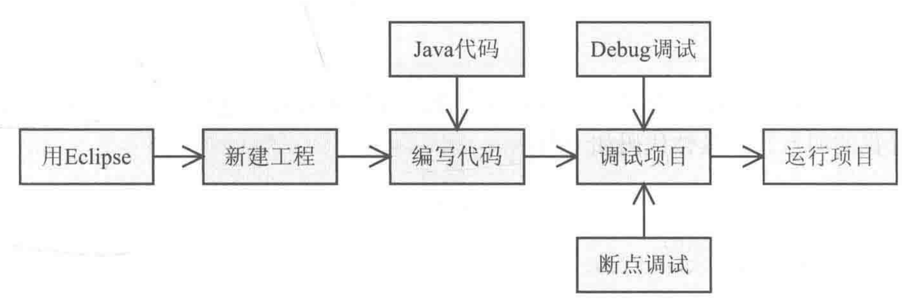
图 2-21 所示的状态的变化是由 Android 内存管理器决定的，Android 会首先关闭那些包含 Inactive Activity 的应用程序，然后关闭 Stopped 状态的程序，只有在极端情况下才会移除 Paused 状态的程序。



▲图 2-21 Activity 状态转换图

2.6 第一段 Android 程序

本实例的功能是在手机屏幕中显示问候语“你好我的朋友！”，在具体开始之前先做一个简单的流程规划，如图 2-22 所示。



▲图 2-22 第一段 Android 程序规划流程图

题目	目的	源码路径
实例 2-1	在手机屏幕中显示问候语	\codes\2\first

本实例的具体实现流程如下所示。

1. 新建 Android 工程

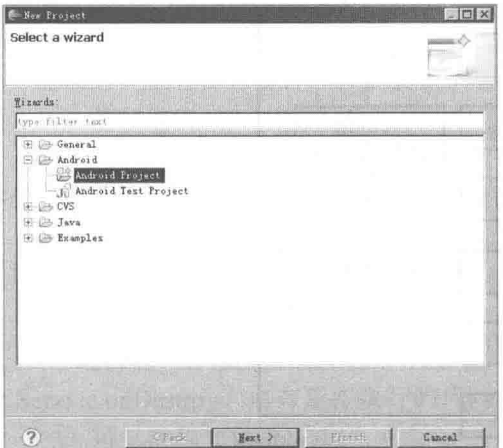
（1）在 Eclipse 中依次单击【File】｜【New】｜【Project】新创建一个工程文件。如图 2-23

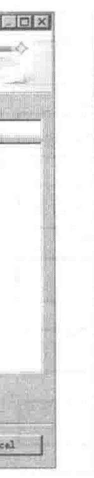
所示。

（2）选择“Android Project”选项，单击“Next”按钮。

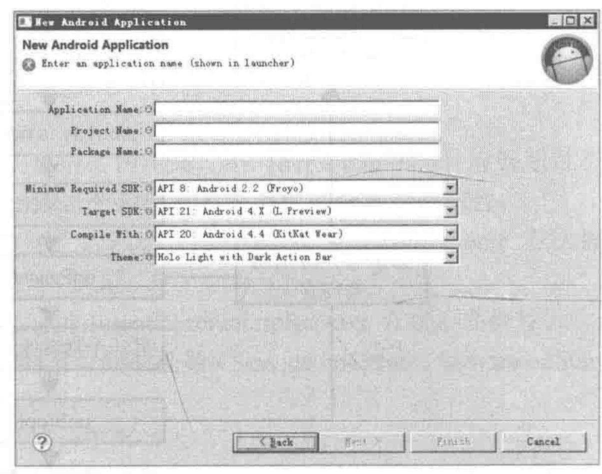
（3）在弹出的“New Android Application”对话框中，设置工程信息。如图 2-24 所示。

在图 2-24 所示的界面中依次设置工程名字、包名字、Activity 名字和应用名字。





▲图 2-23　新建工程文件



▲图 2-24　设置工程

2. 编写代码和代码分析

现在已经创建了一个名为“first”的工程文件，打开文件 first.java，会显示自动生成的如下代码。

```
package first.a;
import android.app.Activity;
import android.os.Bundle;
public class fistMM extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

如果此时运行程序，将不会显示任何东西。此时我们可以对上述代码稍微进行修改，让程序输出“你好我的朋友!”。具体代码如下所示。

```
package first.a;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class fistMM extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView tv = new TextView(this);
        tv.setText("你好我的朋友! ");
        setContentView(tv);
    }
}
```

经过上述改写后，就可以在屏幕中输出“你好我的朋友!”完全符合预期的要求。

3. 调试

Android 调试一般分为 3 个步骤，分别是设置断点、Debug 调试和断点调试。

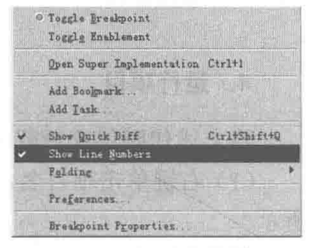
（1）设置断点。

此处的设置断点和 Java 中的方法一样，可以通过双击代码左边的区域进行。如图 2-25 所示。

```
first Manifest    first.java
1  package first.a;
2  import android.app.Activity;
5
6  public class first extends Activity {
7      /** Called when the activity is first created. */
8      @Override
9      public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.main);
12         TextView tv = new TextView(this);
13         tv.setText("你好我的朋友！");
14         setContentView(tv);
15
16     }
17 }
18
19
20
21
```

▲图 2-25　设置断点

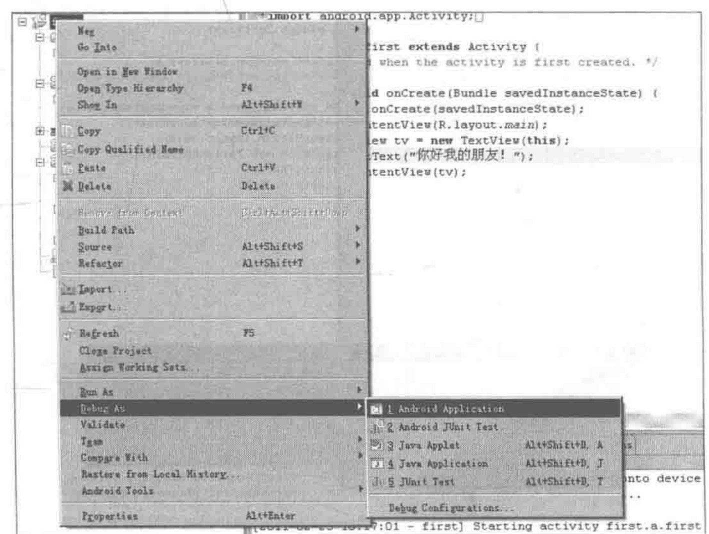
为了调试方便，可以设置显示代码的行数。只需在代码左侧的空白部分单击右键，在弹出的命令中选择“Show Line Numbers”即可，如图 2-26 所示。



▲图 2-26　显示行数

（2）Debug 调试。

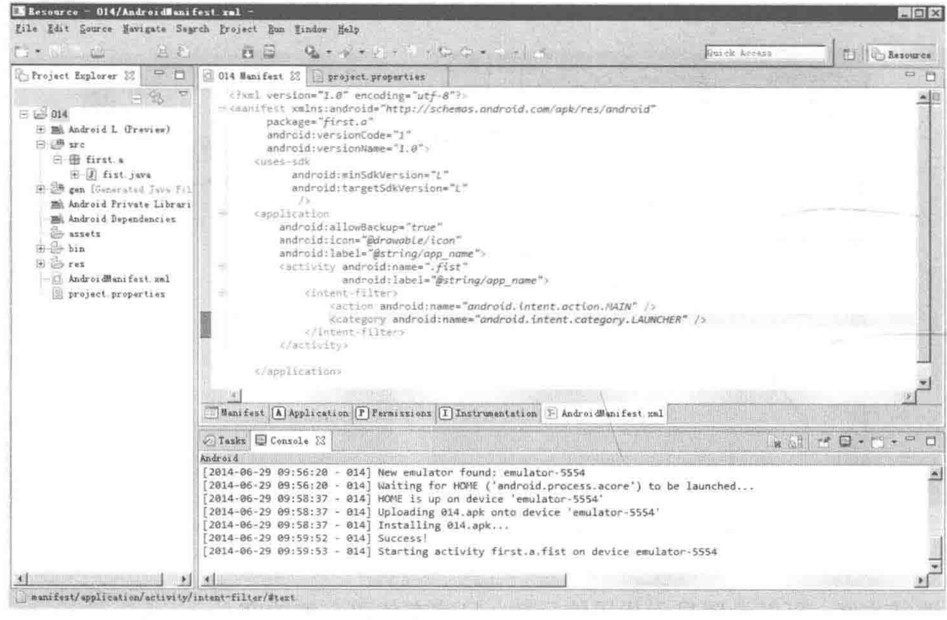
Debug Android 调试项目的方法和普通 Debug Java 调试项目的方法类似，唯一的不同是在调试项目时选择“Android Application”命令。具体方法是右键单击项目名，在弹出的命令中依次选择【Debug As】|【Android Application】命令。如图 2-27 所示。



▲图 2-27　Debug 调试

（3）断点调试。

可以进行单步调试，具体调试方法和调试普通 Java 程序的方法类似，调试界面如图 2-28 所示。

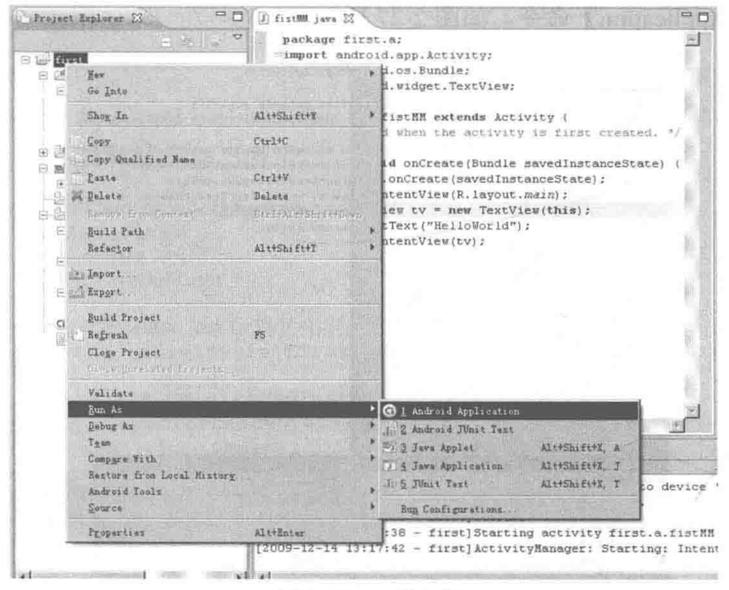


▲图 2-28　断点调试

4. 运行项目

将上述代码保存后就可运行这段程序了，具体过程如下所示。

（1）右键单击项目名，在弹出命令中依次选择【Run As】|【Android Application】。如图 2-29 所示。



▲图 2-29　开始运行

（2）此时工程开始运行，运行完成后在屏幕中输出“你好我的朋友！”这段文字。如图 2-30 所示。

▲图 2-30 运行结果

第 3 章　获取并分析 Android 源码

经过前面的介绍，我们简要了解了 Android 系统的的基本知识，并掌握了 Android 系统的基本架构知识。在本章的内容中，将详细讲解获取并分析 Android 源码的基本知识，介绍各个目录中主要文件的功能，为读者进入本书后面知识的学习打下基础。

3.1 获取 Android 源码

要想研究 Android 系统的源码，需要先获取源码。目前市面上主要是 Windows、Linux、Mac OS 的操作系统。由于 Mac OS 属于类 Linux 系统，所以本书将详细讲解在 Windows 系统和 Linux 系统中获取 Android 源码的知识。

3.1.1 在 Linux 系统中获取 Android 源码

在 Linux 系统中，通常使用 Ubuntu 来下载和编译 Android 源码。由于 Android 的源码内容很多，Google 采用了 git 的版本控制工具，并对不同的模块设置不同的 git 服务器。我们可以用 repo 自动化脚本下载 Android 源码，下面介绍获取 Android 源码的过程。

（1）下载 repo。

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

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

下载 repo 的脚本用于执行 repo，命令如下。

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

设置可执行权限，命令如下。

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

（2）初始化一个 repo 的客户端。

在用户目录下，创建一个空目录用于存放 Android 源码，命令如下。

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

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

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

下载其他分支，正式发布的版本可以通过添加-b 参数下载，命令如下。

```
$ repo init -u https://android.googlesource.com/platform/manifest -b
                                                   android-4.3_r1
```

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

```
$ repo sync
```

经过上述步骤后，便开始下载并同步 Android 源码了，界面效果如图 3-1 所示。

```
Checking out files: 100% (2497/2497), done. out files:  31% (790/2497)
Checking out files: 100% (1654/1654), done. out files:  39% (649/1654)
Checking out files: 100% (3471/3471), done.
Checking out files: 100% (24607/24607), done.ut files:  21% (5269/24607)
Checking out files: 100% (2431/2431), done. out files:  32% (800/2431)
Checking out files: 100% (18696/18696), done.
Checking out files: 100% (4276/4276), done. out files:  48% (2058/4276)
Checking out files: 100% (2216/2216), done. out files:  49% (1093/2216)
Checking out files: 100% (857/857), done.ng out files:  13% (115/857)
Checking out files: 100% (1141/1141), done. out files:   2% (28/1141)
Checking out files: 100% (431/431), done.ng out files:  10% (46/431)
Checking out files: 100% (175/175), done.ng out files:  10% (19/175)
Checking out files: 100% (135/135), done.
Checking out files: 100% (378/378), done.
Checking out files: 100% (433/433), done.
Checking out files: 100% (2407/2407), done. out files:  30% (724/2407)
Checking out files: 100% (2489/2489), done.
Checking out files: 100% (2493/2493), done.
Checking out files: 100% (177/177), done.ng out files:  15% (27/177)
Checking out files: 100% (137/137), done.
Checking out files: 100% (40775/40775), done.ut files:   5% (2199/40775)
Checking out files: 100% (93/93), done.
Checking out files: 100% (450/450), done.
Checking out files: 100% (5265/5265), done. out files:  41% (2167/5265)
Syncing work tree: 100% (329/329), done.
```

▲图 3-1 下载并同步 Android 源码

3.1.2 在 Windows 系统中获取 Android 源码

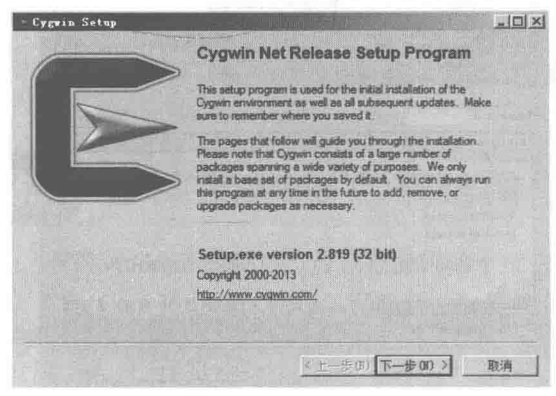
在 Windows 系统中获取源码和在 Linux 系统中原理相同，但是需要预先在 Windows 系统中搭建一个 Linux 环境，此处需要用到 Cygwin 工具。Cygwin 的作用是构建一套在 Windows 系统中的 Linux 模拟环境，下载 Cygwin 工具的地址如下。

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

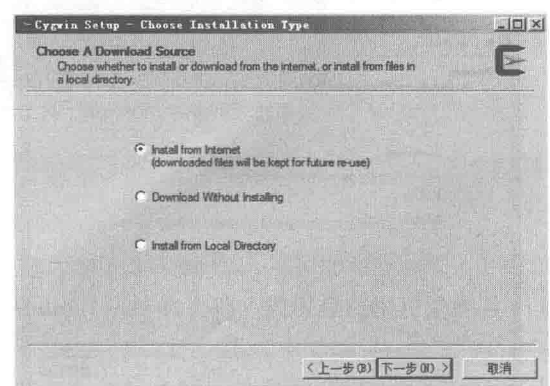
下载成功后会得到一个名为“setup.exe”的可执行文件，通过此文件可以更新和下载最新的工具版本，具体流程如下所示。

（1）启动 Cygwin，如图 3-2 所示。

（2）单击“下一步”按钮，选择第一个选项：Install from Internet（从网络下载安装），如图 3-3 所示。



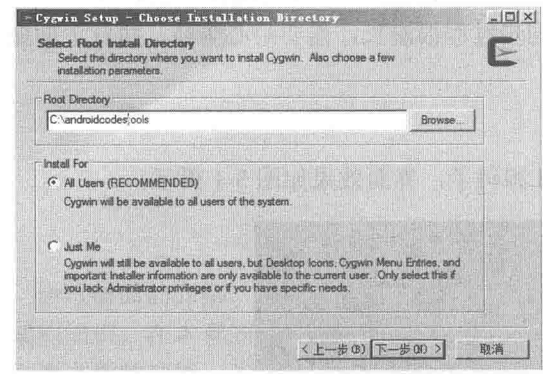
▲图 3-2 启动 Cygwin



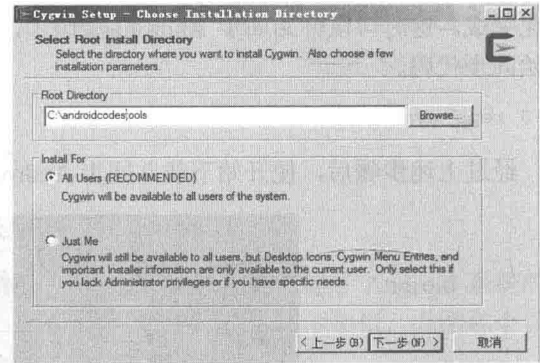
▲图 3-3 选择从网络下载安装

（3）单击“下一步”按钮，选择安装根目录，如图 3-4 所示。

（4）单击“下一步”按钮，选择临时文件目录，如图 3-5 所示。



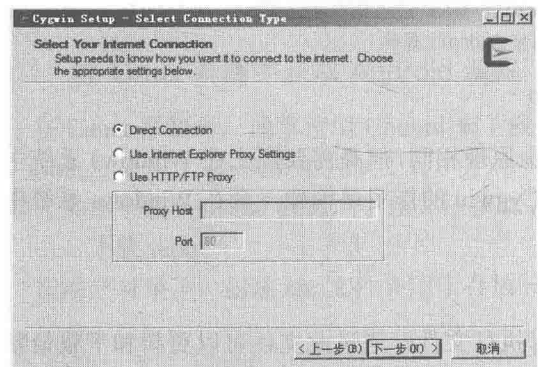
▲图 3-4　选择安装根目录



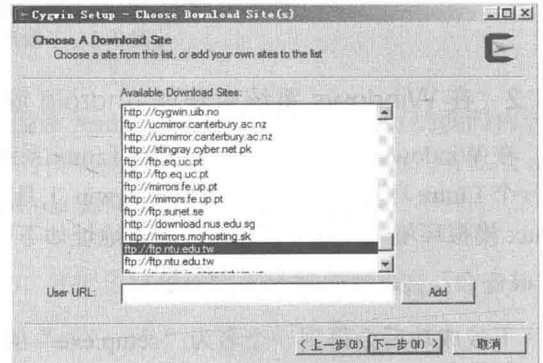
▲图 3-5　选择临时文件目录

（5）单击“下一步”按钮，设置网络代理。如果所在网络需要代理，则在这一步进行设置；如果不用代理，则选择直接下载，如图 3-6 所示。

（6）单击“下一步”按钮，选择下载站点。一般选择离我们比较近的站点，速度会比较快，如图 3-7 所示。



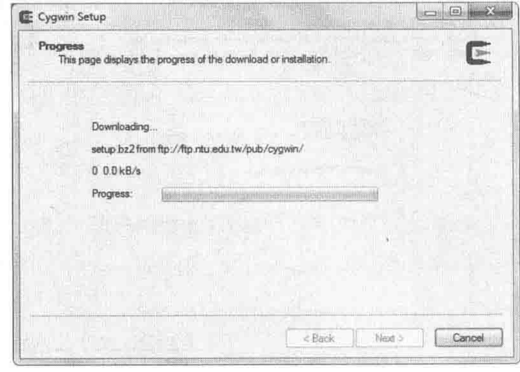
▲图 3-6　设置网络代理



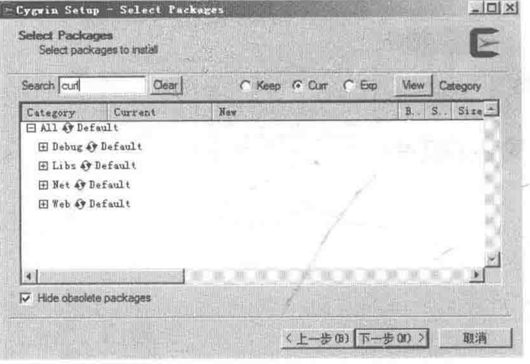
▲图 3-7　选择下载站点

（7）单击“下一步”按钮，开始更新工具列表，如图 3-8 所示。

（8）单击“下一步”按钮，选择需要下载的工具包。在此我们需要依次下载 curl、git、python 这些工具包。如图 3-9 所示。



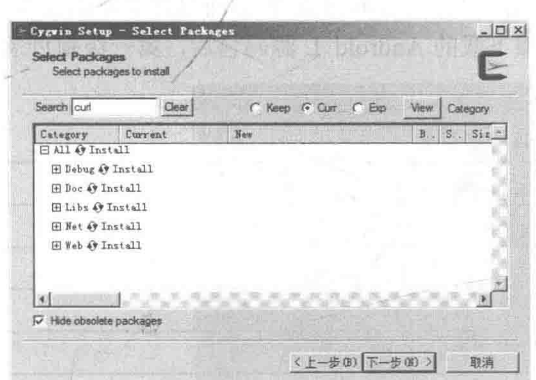
▲图 3-8　更新工具列表



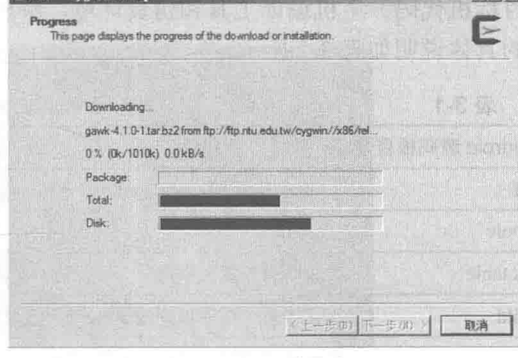
▲图 3-9　依次下载工具

为了确保能够安装上述工具，一定要用鼠标双击使其变为 Install 形式。如图 3-10 所示。

（9）单击“下一步”按钮，这之后要经过漫长的等待过程。如图 3-11 所示。



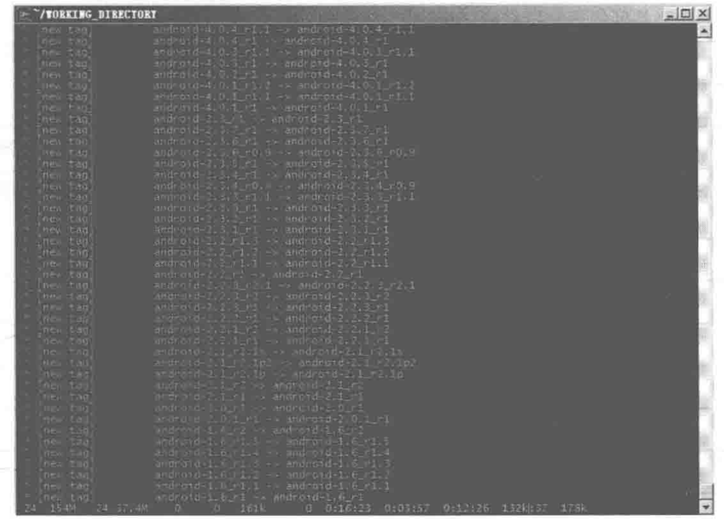
▲图 3-10　务必设置为 Install 形式



▲图 3-11　下载进度

如果下载安装成功会出现提示信息，单击“完成”按钮即完成安装。打开安装好的 Cygwin 后，会模拟一个 Linux 的工作环境，然后就可以按照 Linux 平台的源码下载方法下载 Android 源码了。

建议读者在下载 Android 源码时，严格按照官方提供的步骤进行，网址是 http://source.android.com/source/downloading.html，这一点对初学者来说尤为重要。另外，整个下载过程比较漫长，需要大家耐心等待。图 3-12 所示为笔者机器的命令截图。



▲图 3-12　在 Windows 中用 Cygwin 工具下载 Android 源码的截图

3.2 分析 Android 源码结构

获得 Android 源码后，可以把源码的全部工程分为如下 3 个部分。

（1）Core Project：核心工程部分，这是建立 Android 系统的基础，被保存在根目录的各个文件夹中。

（2）External Project：扩展工程部分，可以使其他开源项目具有扩展功能，被保存在“external”文件夹中。

（3）Package：包部分，提供了 Android 的应用程序、内容提供者、输入法和服务，被保存在“package”文件夹中。

无论是 Android 1.5 还是 Android L，各个版本的源码目录基本类似。在里面包含了原始 Android 的目标机代码、主机编译工具和仿真环境。解压缩下载的 Android L 源码包后，第一级别目录结构的具体说明如表 3-1 所示。

表 3-1　Android L 源码的根目录

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

3.3 编译 Android 源码

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

```
$: cd ~/AndroidL（这里的“AndroidL”就是我们下载源码的保存目录）
$: make
```

编译 Android 源码可以得到“~/project/android/cupcake/out”目录，笔者的截图界面如图 3-13 所示。

▲图 3-13　编译过程的界面截图

整个编译过程也非常漫长，需要读者耐心等待。

3.3.1　搭建编译环境

在编译 Android 源码之前，需要先进行环境搭建工作。在接下来的内容中，以 Ubuntu 系统为例讲解搭建编译环境以及编译 Android 源码的方法。具体流程如下所示。

（1）安装 JDK。编译 Android L 的源码需要 JDK1.6，下载 jdk-6u21-linux-i586.bin 后进行安装，对应命令如下。

```
$ cd /usr
$ mkdir java
$ cd java
$ sudo cp jdk-6u21-linux-i586.bin 所在目录 ./
$ sudo chmod 755 jdk-6u21-linux-i586.bin
$ sudo sh jdk-6u21-linux-i586.bin
```

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

```
export JAVA_HOME=/usr/java/jdk1.6.0_23
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$PATH:$JAVA_HOME/bin:$JAVA_HOME/bin/tools.jar:$JRE_HOME/bin
export ANDROID_JAVA_HOME=$JAVA_HOME
```

（3）安装需要的包。读者可以根据编译过程中的提示进行选择，可能需要的包的安装命令如下。

```
$ sudo apt-get install git-core bison zlib1g-dev flex libx11-dev gperf sudo aptitude
install git-core gnupg flex bison gperf libsdl-dev libesd0-dev libwxgtk2.6-dev
build-essential zip curl libncurses5-dev zlib1g-dev
```

3.3.2　开始编译

当安装所依赖的包工作完成之后，就可以开始编译 Android 源码了，具体步骤如下所示。

（1）首先进行编译初始化工作，在终端中执行下面的命令。

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

或：

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

执行后将会输出：

```
source build/envsetup.sh
including device/asus/grouper/vendorsetup.sh
including device/asus/tilapia/vendorsetup.sh
including device/generic/armv7-a-neon/vendorsetup.sh
including device/generic/armv7-a/vendorsetup.sh
including device/generic/mips/vendorsetup.sh
including device/generic/x86/vendorsetup.sh
including device/samsung/maguro/vendorsetup.sh
including device/samsung/manta/vendorsetup.sh
including device/samsung/toroplus/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
```

（2）然后选择编译目标，具体命令是：

```
lunch full-eng
```

执行后会输出如下所示的提示信息。

```
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=L
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-2.6.33-45-generic-x86_63-with-Ubuntu-10.03-lucid
HOST_BUILD_TYPE=release
BUILD_ID=JOP40C
OUT_DIR=out
============================================
```

（3）接下来开始编译代码，在终端中执行下面的命令。

```
make -j4
```

其中"-j4"表示用 4 个线程进行编译。整个编译进度根据不同机器的配置而需要不同的时间。例如笔者电脑为 Intel i5-2300 四核 2.8、4GB 内存，经过近 4 小时才编译完成。当出现下面的信息时表示编译完成。

```
target Java: ContactsTests (out/target/common/obj/APPS/ContactsTests_intermediates/
classes)
target Dex: Contacts
Done!
Install: out/target/product/generic/system/app/Browser.odex
Install: out/target/product/generic/system/app/Browser.apk
Note: Some input files use or override a deprecated API.
```

```
Note: Recompile with -Xlint:deprecation for details.
Copying: out/target/common/obj/APPS/Contacts_intermediates/noproguard.classes.dex
target Package: Contacts (out/target/product/generic/obj/APPS/Contacts_intermediates/
package.apk)
 'out/target/common/obj/APPS/Contacts_intermediates/classes.dex' as 'classes.dex'...
Processing target/product/generic/obj/APPS/Contacts_intermediates/package.apk
Done!
Install: out/target/product/generic/system/app/Contacts.odex
Install: out/target/product/generic/system/app/Contacts.apk
build/tools/generate-notice-files.py  out/target/product/generic/obj/NOTICE.txt  out/
target/product/generic/obj/NOTICE.html "Notices for files contained in the filesystem
images in this directory:" out/target/product/generic/obj/NOTICE_FILES/src
Combining NOTICE files into HTML
Combining NOTICE files into text
Installed file list: out/target/product/generic/installed-files.txt
Target system fs image: out/target/product/generic/obj/PACKAGING/systemimage_
intermediates/system.img
Running:  mkyaffs2image -f out/target/product/generic/system out/target/product/
generic/obj/PACKAGING/systemimage_intermediates/system.img
Install system fs image: out/target/product/generic/system.img
DroidDoc took 5331 sec. to write docs to out/target/common/docs/doc-comment-check
```

3.3.3 在模拟器中运行

在模拟器中运行的步骤就比较简单了，只需在终端中执行下面的命令即可。

```
emulator
```

运行成功后的效果如图 3-14 所示。

▲图 3-14 在模拟器中的编译执行效果

3.3.4 常见的错误分析

虽然编译方法非常简单，但是作为初学者来说很容易出错，下面列出其中常见的编译错误类型。

（1）缺少必要的软件。

进入到 Android 目录下，使用 make 命令进行编译，可能会出现如下所示的错误提示。

```
host C: libneo_cgi <= external/clearsilver/cgi/cgi.c
external/clearsilver/cgi/cgi.c:22:18: error: zlib.h: No such file or directory
```

上述错误是因为缺少 zlib1g-dev，需要使用 apt-get 命令从软件仓库中安装 zlib1g-dev，具体命令如下所示。

```
sudo apt-get install zlib1g-dev
```

同理需要安装下面的软件，否则也会出现上述类似的错误。

```
sudo apt-get install flex
sudo apt-get install bison
sudo apt-get install gperf
sudo apt-get install libsdl-dev
sudo apt-get install libesd0-dev
sudo apt-get install libncurses5-dev
sudo apt-get install libx11-dev
```

（2）没有安装 Java 环境 JDK。

当安装所有上述软件后，运行 make 命令再次编译 Android 源码。如果在之前忘记安装 Java 环境 JDK，则此时会出现很多 Java 文件无法编译的错误。如果打开 Android 的源码，可以看到在如下目录中发现很多 Java 源文件。

```
android/dalvik/libcore/dom/src/test/java/org/w3c/domts
```

这充分说明在编译 Android 之前必须先安装 Java 环境 JDK，安装流程如下所示。

- 从 Oracle 官方网站下载 jdk-6u16-linux-i586.bin 文件，然后安装。

在 Ubuntu 8.04 中，“/etc/profile”文件是全局的环境变量配置文件，它适用于所有的 shell。在登录 Linux 系统时应该先启动“/etc/profile”文件，然后再启动用户目录下的“~/.bash_profile”、“~/.bash_login”或“~/.profile”文件中的其中一个，执行的顺序和上面的排序一样。如果“~/.bash_profile”文件存在，则还会执行“~/.bashrc”文件。在此只需要把 JDK 的目录放到“/etc/profile”目录下即可。

```
JAVA_HOME=/usr/local/src/jdk1.6.0_16
PATH=$PATH:$JAVA_HOME/bin:/usr/local/src/android-sdk-linux_x86-1.1_r1/tools:~/bin
```

- 重新启动机器，输入 java –version 命令，如果输出下面的信息则表示配置成功。

```
ava version "1.6.0_16"
Java(TM) SE Runtime Environment (build 1.6.0_16-b01)
Java HotSpot(TM) Client VM (build 13.3-b01, mixed mode, sharing)
```

当成功编译 Android 源码后，在终端会输出如下提示。

```
Target system fs image: out/target/product/generic/obj/PACKAGING/systemimage_unopt_
intermediates/system.img
Install system fs image: out/target/product/generic/system.img
Target ram disk: out/target/product/generic/ramdisk.img
Target userdata fs image: out/target/product/generic/userdata.img
Installed file list: out/target/product/generic/installed-files.txt
root@dfsun2009-desktop:/bin/android#
```

3.3.5 实践演练——演示两种编译 Android 程序的方法

Android 编译环境本身比较复杂，并且不像普通的编译环境那样只有顶层目录下才有 Makefile 文件，而其他的每个 component 都使用统一标准的 Android.mk. Android.mk 文件。不过这并不是我们熟悉的 Makefile，而是经过 Android 自身编译系统的很多处理，所以其中的联系比较复杂。不过这种方式的好处在于，编写一个新的 Android.mk 给 Android 增加一个新的 Component 会变得比较简单。为了使读者更加深入地理解在 Linux 环境下编译 Android 程序的方法，在接下来的内容中，将分别演示两种编译 Android 程序的方法。

1. 编译 Native C 的 helloworld 模块

编译 Java 程序可以直接采用 Eclipse 的集成环境来完成，实现方法非常简单，在这里就不再重复了。接下来将通过一个例子来说明如何在 Android 中增加一个 C 程序的 Hello World。

（1）在“$(YOUR_ANDROID)/development”目录下创建一个名为“hello”的目录，并用

“$(YOUR_ANDROID)”指向 Android 源代码所在的目录。

```
- # mkdir $(YOUR_ANDROID)/development/hello
```

（2）在目录“$(YOUR_ANDROID)/development/hello/”下编写一个名为“hello.c”的 C 语言文件，文件 hello.c 的代码如下所示。

```
#include <stdio.h>
int main()
{
   printf("Hello World!\n");//输出 Hello World
return 0;
}
```

（3）在目录“$(YOUR_ANDROID)/development/hello/”下编写 Android.mk 文件。这是 Android Makefile 的标准命名，不能更改。文件 Android.mk 的格式和内容可以参考其他已有的 Android.mk 文件的写法，针对 helloworld 程序的 Android.mk 文件内容如下所示。

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
   hello.c
LOCAL_MODULE := helloworld
include $(BUILD_EXECUTABLE)
```

- LOCAL_SRC_FILES：用来指定源文件；
- LOCAL_MODULE：指定要编译的模块的名字，在下一步骤编译时将会用到；
- include $(BUILD_EXECUTABLE)：表示要编译成一个可执行文件，如果想编译成动态库则可用 BUILD_SHARED_LIBRARY，这些具体用法可以在“$(YOUR_ANDROID)/build/core/config.mk”中查到。

（4）回到 Android 源代码顶层目录进行编译。

```
# cd $(YOUR_ANDROID) && make helloworld
```

在此需要注意，make helloworld 中的目标名 helloworld 就是上面 Android.mk 文件中由 LOCAL_MODULE 指定的模块名。最终的编译结果如下所示。

```
target thumb C: helloworld <= development/hello/hello.c
target Executable: helloworld (out/target/product/generic/obj/EXECUTABLES/helloworld_
intermediates/LINKED/helloworld)
target  Non-prelinked:  helloworld  (out/target/product/generic/symbols/system/bin/
helloworld)
target  Strip:  helloworld  (out/target/product/generic/obj/EXECUTABLES/helloworld_
intermediates/helloworld)
Install: out/target/product/generic/system/bin/helloworld
```

（5）如果和上述编译结果相同，则编译后的可执行文件存放在如下目录。

```
out/target/product/generic/system/bin/helloworld
```

通过“adb push”将它传送到模拟器上，再通过“adb shell”登录到模拟器终端后就可以执行了。

2. 手工编译 C 模块

在前面讲解了通过标准的 Android.mk 文件来编译 C 模块的具体流程。其实我们可以直接运用 gcc 命令行来编译 C 程序，这样可以更好地了解 Android 编译环境的细节。具体流程如下所示。

（1）在 Android 编译环境中，提供了”showcommands”选项来显示编译命令行，我们可以通过打开这个选项来查看一些编译时的细节。

（2）在具体操作之前需要使用如下命令把前面中的 helloworld 模块清除。

```
# make clean-helloworld
```

上面的“make clean-$(LOCAL_MODULE)”命令是 Android 编译环境提供的 make clean 的方式。

（3）使用 showcommands 选项重新编译 helloworld，具体命令如下所示。

```
# make helloworld showcommands
build/core/product_config.mk:229: WARNING: adding test OTA key
target thumb C: helloworld <= development/hello/hello.c
prebuilt/linux-x86/toolchain/arm-eabi-L/bin/arm-eabi-gcc -I system/core/include -I hardware/libhardware/include -I hardware/ril/include -I dalvik/libnativehelper/include -I frameworks/base/include -I external/skia/include -I out/target/product/generic/obj/include -I bionic/libc/arch-arm/include -I bionic/libc/include -I bionic/libstdc++/include -I bionic/libc/kernel/common -I bionic/libc/kernel/arch-arm -I bionic/libm/include -I bionic/libm/include/arch/arm -I bionic/libthread_db/include -I development/hello -I out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates -c -fno-exceptions -Wno-multichar -march=armv5te -mtune=xscale -msoft-float -fpic -mthumb-interwork -ffunction-sections -funwind-tables -fstack-protector -D__ARM_ARCH_5__ -D__ARM_ARCH_5T__ -D__ARM_ARCH_5E__ -D__ARM_ARCH_5TE__ -include system/core/include/arch/linux-arm/AndroidConfig.h -DANDROID -fmessage-length=0 -W -Wall -Wno-unused -DSK_RELEASE -DNDEBUG -O2 -g -Wstrict-aliasing=2 -finline-functions -fno-inline-functions-called-once -fgcse-after-reload -frerun-cse-after-loop -frename-registers -DNDEBUG -UDEBUG -mthumb -Os -fomit-frame-pointer -fno-strict-aliasing -finline-limit=64 -MD -o out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/hello.o development/hello/hello.c

target Executable: helloworld (out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/LINKED/helloworld)

prebuilt/linux-x86/toolchain/arm-eabi-L/bin/arm-eabi-g++ -nostdlib -Bdynamic -Wl,-T,build/core/armelf.x -Wl,-dynamic-linker,/system/bin/linker -Wl,--gc-sections -Wl,-z,nocopyreloc -o out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/LINKED/helloworld -Lout/target/product/generic/obj/lib -Wl,-rpath-link=out/target/product/generic/obj/lib -lc -lstdc++ -lm out/target/product/generic/obj/lib/crtbegin_dynamic.o out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/hello.o -Wl,--no-undefined prebuilt/linux-x86/toolchain/arm-eabi-L/bin/../lib/gcc/arm-eabi/4.3.1/interwork/libgcc.a out/target/product/generic/obj/lib/crtend_android.o

target Non-prelinked: helloworld (out/target/product/generic/symbols/system/bin/helloworld)

out/host/linux-x86/bin/acp -fpt out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/LINKED/helloworld out/target/product/generic/symbols/system/ bin/helloworld

target Strip: helloworld (out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/helloworld)

out/host/linux-x86/bin/soslim --strip --shady --quiet out/target/product/generic/symbols/system/bin/helloworld --outfile out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/helloworld

Install: out/target/product/generic/system/bin/helloworld

out/host/linux-x86/bin/acp -fpt out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/helloworld out/target/product/generic/system/bin/helloworld
```

从上述命令行可以看到，Android 编译环境所用的交叉编译工具链如下。

```
prebuilt/linux-x86/toolchain/arm-eabi-L/bin/arm-eabi-gcc
```

其中参数“-I”和“-L”分别指定了所用的 C 库头文件和动态库文件路径分别是“bionic/libc/include”和“out/target/product/generic/obj/lib”，其他还包括很多编译选项以及-D 所定义的预编译宏。

（4）此时就可以利用上面的编译命令来手工编译 helloworld 程序，首先手工删除上次编译得

到的 helloworld 程序。

```
# rm out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/hello.o
# rm out/target/product/generic/system/bin/helloworld
```

然后再用 gcc 编译以生成目标文件。

```
# prebuilt/linux-x86/toolchain/arm-eabi-L/bin/arm-eabi-gcc -I bionic/libc/arch-arm/
include -I bionic/libc/include -I bionic/libc/kernel/common  -I bionic/libc/kernel/
arch-arm -c -fno-exceptions -Wno-multichar -march=armv5te -mtune=xscale -msoft-float
-fpic  -mthumb-interwork  -ffunction-sections  -funwind-tables  -fstack-protector
-D__ARM_ARCH_5__  -D__ARM_ARCH_5T__  -D__ARM_ARCH_5E__  -D__ARM_ARCH_5TE__  -include
system/core/include/arch/linux-arm/AndroidConfig.h -DANDROID -fmessage-length=0 -W
-Wall -Wno-unused -DSK_RELEASE -DNDEBUG -O2 -g -Wstrict-aliasing=2 -finline-functions
-fno-inline-functions-called-once -fgcse-after-reload -frerun-cse-after-loop -frename-
registers  -DNDEBUG  -UDEBUG  -mthumb  -Os  -fomit-frame-pointer  -fno-strict-aliasing
-finline-limit=64     -MD -o out/target/product/generic/obj/EXECUTABLES/helloworld_
intermediates/hello.o development/hello/hello.c
```

如果此时与 Android.mk 编译参数进行比较，会发现上面主要减少了不必要的参数“-I”。

（5）接下来开始生成可执行文件。

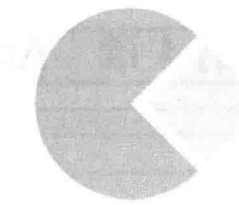
```
# prebuilt/linux-x86/toolchain/arm-eabi-L/bin/arm-eabi-gcc -nostdlib -Bdynamic -Wl,
-T,build/core/armelf.x -Wl,-dynamic-linker,/system/bin/linker -Wl,--gc-sections -Wl,-z,
nocopyreloc  -o  out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/
LINKED/helloworld  -Lout/target/product/generic/obj/lib  -Wl,-rpath-link=out/target/
product/generic/obj/lib -lc -lm  out/target/product/generic/obj/EXECUTABLES/helloworld_
intermediates/hello.o out/target/product/generic/obj/lib/crtbegin_dynamic.o -Wl,--no-
undefined  ./prebuilt/linux-x86/toolchain/arm-eabi-L/bin/../lib/gcc/arm-eabi/4.3.1/in
terwork/libgcc.a out/target/product/generic/obj/lib/crtend_android.o
```

在此需要特别注意的是参数“-Wl,-dynamic-linker,/system/bin/linker”，它指定了 Android 专用的动态链接器是“/system/bin/linker”，而不是平常使用的 ld.so。

（6）最后可以使用命令 file 和 readelf 来查看生成的可执行程序。

```
#  file  out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/LINKED/
helloworld
out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/LINKED/helloworl
d: ELF 33-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs),
not stripped
#   readelf  -d  out/target/product/generic/obj/EXECUTABLES/helloworld_intermediates/
LINKED/helloworld |grep NEEDED
 0x00000001 (NEEDED)                     Shared library: [libc.so]
 0x00000001 (NEEDED)                     Shared library: [libm.so]
```

这就是 ARM 格式的动态链接可执行文件，在运行时需要 libc.so 和 libm.so。当提示“not stripped”时表示它还没被 STRIP（剥离）。嵌入式系统中为节省空间通常将编译完成的可执行文件或动态库进行剥离，即去掉其中多余的符号表信息。在前面“make helloworld showcommands”命令的最后我们也可以看到，Android 编译环境中使用了“out/host/linux-x86/bin/soslim”工具进行 STRIP。



第二篇

系统分析篇

第 4 章　Android 多媒体框架

第 5 章　音频系统框架

第 6 章　视频系统框架

第 7 章　照相机系统

第 8 章　Alarm 时钟系统

第 9 章　振动器系统

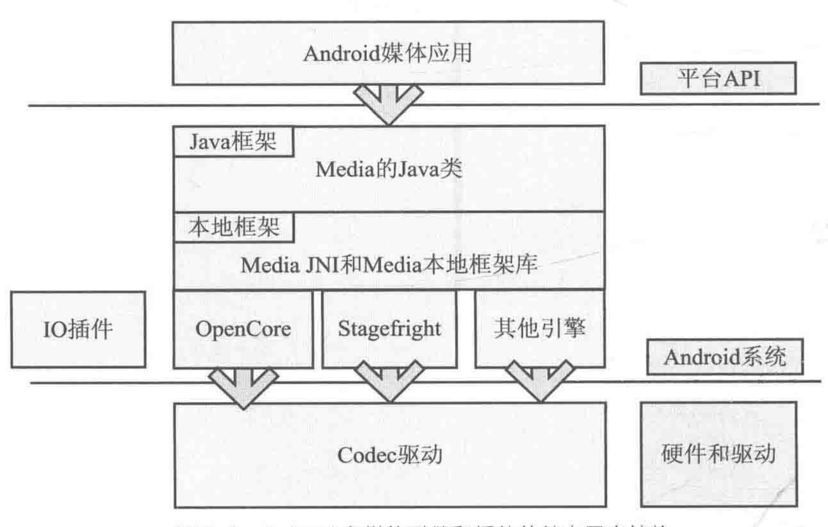
第 4 章　Android 多媒体框架

从 Android 2.2 版本以后，Android 对多媒体框架进行了很大的调整，抛弃了原来的 OpenCore 框架，改用 StageFright 框架，仅仅对 OpenCore 中的 omx-component 部分做了引用。和 OpenCore 框架相比，StageFright 框架更加易懂，并且封装也相对简单。在 Android 2.2 及以前版本，OpenCore 位于“extemal”目录下，在 Android 2.3 以后，多媒体的功能被放置到“frameworks/base/media”目录下。在本章将详细讲解 OpenCore 框架和 StageFright 框架的基本知识，为读者进入本书后面知识的学习打下基础。

4.1 Android 多媒体系统介绍

在 Android 的多媒体系统中，可以根据需要添加一些第三方插件，这样可以增强多媒体系统的功能。在 Android 系统的本地多媒体引擎上面，是 Android 的多媒体本地框架，而在多媒体本地框架上面是多媒体 JNI 和多媒体的 Java 框架部分。和多媒体相关的应用程序通过调用 Android Java 框架层，来提供标准的多媒体 API 进行构建。我们本章将要讲解的 OpenCore 引擎和 Stagefright 引擎是 Android 本地框架中定义接口的实现者，上层调用者不知道 Android 下层使用什么多媒体引擎。

Android 多媒体引擎和插件的基本层次结构如图 4-1 所示。



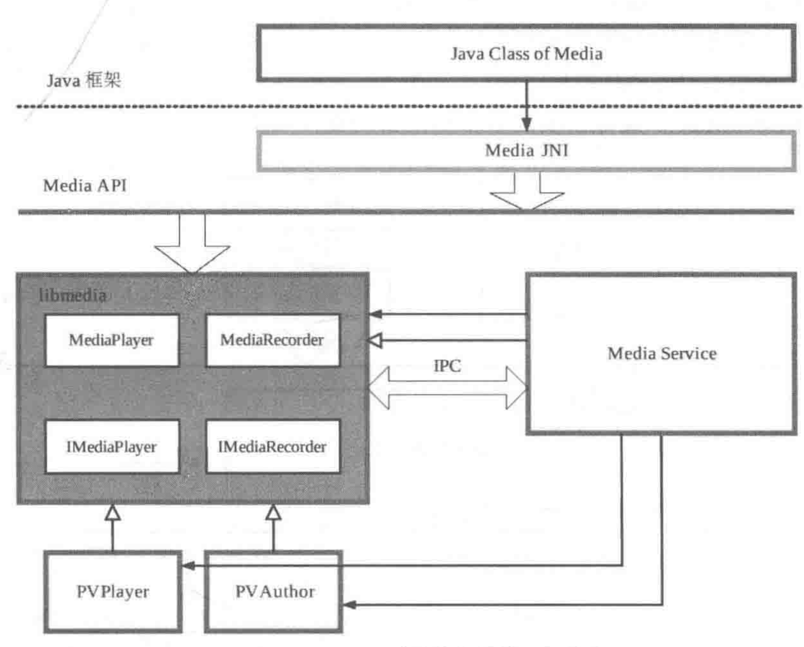
▲图 4-1　Android 多媒体引擎和插件的基本层次结构

Android 系统的多媒体框架结构如图 4-2 所示。

从多媒体应用的实现角度来看，多媒体系统主要包含如下两方面的内容。

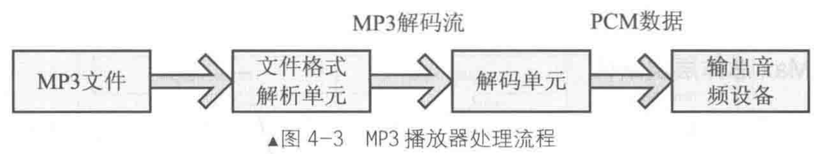
（1）输入、输出环节：音频、视频纯数据流的输入、输出系统；

（2）中间处理环节：包括文件格式处理和编码/解码环节处理。



▲图 4-2　Android 系统的多媒体框架结构

假如想要处理一个 MP3 文件，媒体播放器的处理流程是：将一个 MP3 格式的文件作为播放器输入，将声音从播放器设备输出。在具体实现上，MP3 播放器经过了 MP3 格式文件解析、MP3 码流解码和 PCM 输出播放的过程。整个过程如图 4-3 所示。



▲图 4-3　MP3 播放器处理流程

4.2 OpenMax 框架详解

2006 年，NVIDIA 公司和 Khronos 联合推出 OpenMax，这是多媒体应用程序的框架标准。OpenMax 是无授权费的、跨平台的应用程序接口（API）。OpenMax 通过使媒体加速组件，能够在开发、集成和编程环节中跨多操作系统和处理器硬件平台，并提供全面的流媒体编码/解码器和应用程序便携化。

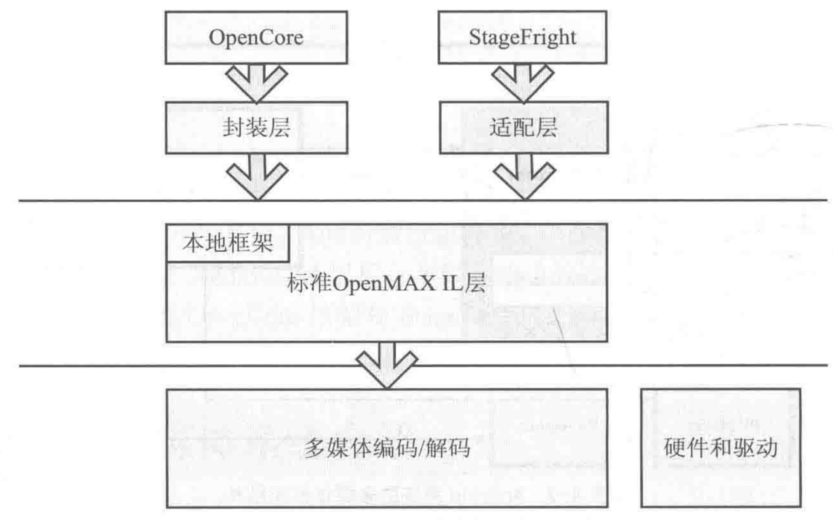
OpenMax 的官方网站地址是：http://www.khronos.org/openmax/

OpenMax 是一个多媒体应用程序的框架标准。在这个标准中，在集成层 OpenMax IL 中定义了媒体组件接口，通过这些接口可以在嵌入式器件的流媒体框架中，实现对加速编码器和解码器的快速集成。

Android 系统本身没有独立的多媒体系统，而是直接使用了市面中的现成的产品，OpenMax IL 便是其中之一。在 Android 结构中，OpenMax IL 通常被当作多媒体引擎插件来使用。Android 最早的多媒体引擎是 OpenCore，后续版本逐渐使用 StageFright 来代替。这两种引擎都可以使用 OpenMax 作为插件，主要实现编码和解码（Codec）处理。

在 Android 的框架层中定义了由 Android 封装的 OpenMax 接口，此接口和标准的接口类似。但是因为使用的是 C++类型接口，并且使用了 Android 的 Binder IPC 机制，所以处理速度会很快。后续引擎 StageFright 使用了封装的 OpenMax 接口，而早启引擎 OpenCore 并没有使用此接口，而是使用其他形式封装了 OpenMax IL 层接口。

Android 中 OpenMax 多媒体框架的基本层次结构如图 4-4 所示。



▲图 4-4　OpenMax 多媒体框架的层次结构

4.2.1　分析 OpenMax 框架构成

在图 4-4 中列出了 OpenMax 多媒体框架的层次结构，在接下来的内容中，将详细讲解各个层次结构的基本知识。

1. OpenMax 总体层次结构

OpenMax 分成 3 个层次，从上到下分别是 OpenMax DL（开发层）、OpenMax IL（集成层）和 OpenMax AL（应用层）。在实际的应用中，OpenMax 的 3 个层次中使用较多的是 OpenMax IL 集成层，由于操作系统到硬件的差异和多媒体应用的差异，OpenMax 的 DL 层和 AL 层使用相对较少。

接下来对上述 3 个层次结构进行具体说明。

（1）OpenMax DL（Development Layer，开发层）。

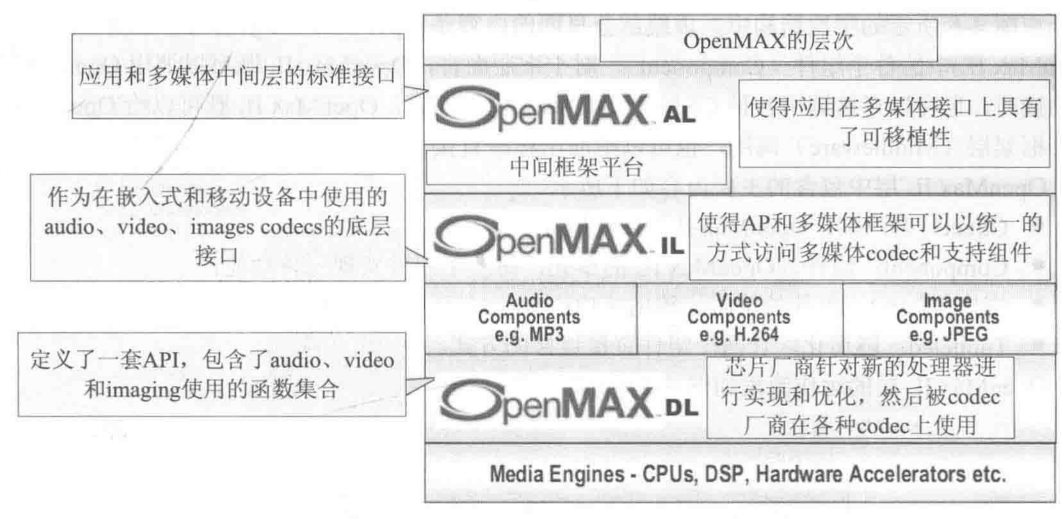
在 OpenMax DL 中定义了集音频、视频和图像功能的 API，这样供应商可以在一个新的处理器上实现并优化，然后编码/解码供应商使用它来编写更广泛的编码/解码器功能。OpenMax DL 可以处理 FFT 和 Filter 等音频信号，也可以实现颜色空间转换和处理原始视频，还可以实现对诸如 MPEG-4、H.264、MP3、AAC 和 JPEG 等编码/解码器的优化。

（2）OpenMax IL（Integration Layer，集成层）。

OpenMax IL 是一种音频、视频和图像编码/解码器，能够实现和多媒体编码/解码器的交互。OpenMax IL 的主要目的是使用特征集合为编码/解码器提供一个系统抽象，解决多个不同媒体系统之间的轻便性问题。

（3）OpenMax AL（Appliction Layer，应用层）。

OpenMax AL API 在应用程序和多媒体中间件之间提供了一个标准化接口，多媒体中间件提供服务以实现被期待的 API 功能。OpenMax 具有 3 个层次，具体如图 4-5 所示。

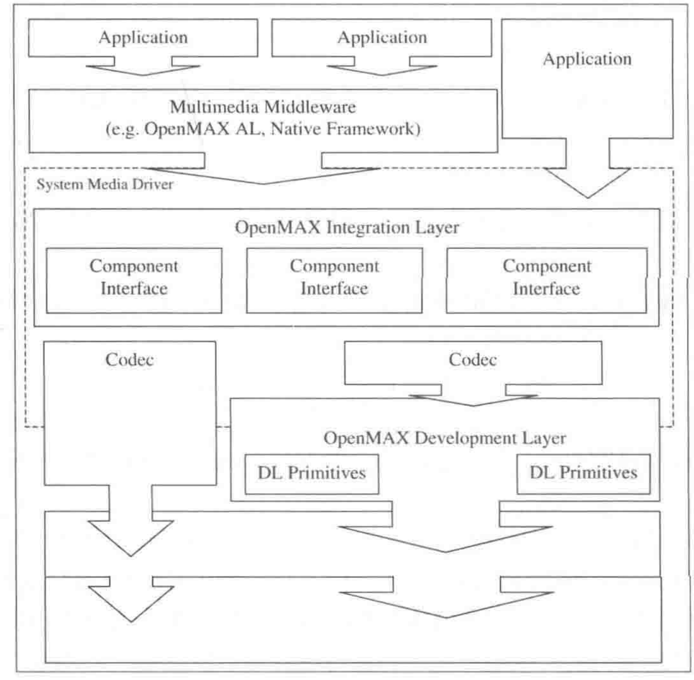


▲图 4-5 OpenMax 层次

2. OpenMax IL 层的结构

在当前多媒体领域，因为 OpenMax IL 的普及性，所以它实际上已经成为了多媒体框架标准。大多数嵌入式处理器或者多媒体编码/解码器模块的硬件生产者，都提供了标准的 OpenMax IL 层的软件接口，这样程序员就可以基于此层次的标准化接口进行多媒体程序的开发。

OpenMax IL 的接口层次结构比较科学，既不是硬件编码/解码器的接口，也不是应用程序层的接口，所以可以比较容易地实现标准化。OpenMax IL 的层次结构如图 4-6 所示。



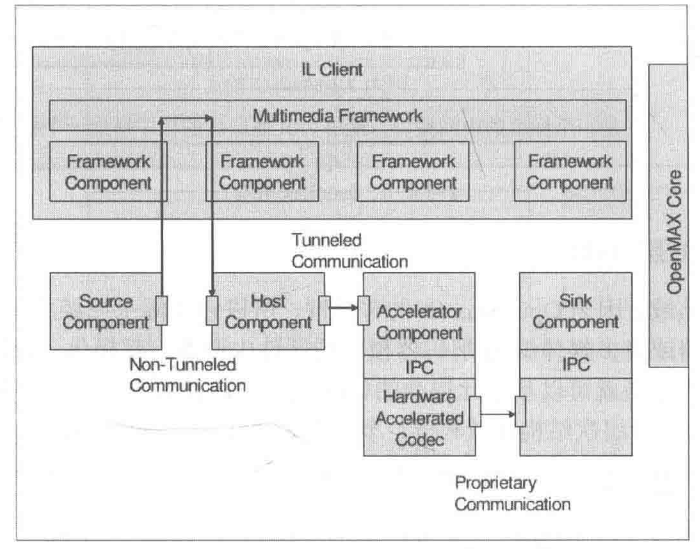
▲图 4-6 OpenMax IL 的层次结构

在图 4-6 所示的层次结构中，虚线部分里面的内容是 OpenMax IL 层的内容，其功能是实现 OpenMax IL 中的各个组件（Component）。对于下层而言，OpenMax IL 既可以调用 OpenMax DL 层的接口，也可以直接调用各种 Codec 实现。对于上层而言，OpenMax IL 既可以给 OpenMax AL 层等框架层（Middleware）调用，也可以给应用程序直接调用。

OpenMax IL 层中包含的主要内容如下所示。

- Client：客户端，OpenMax IL 的调用者；
- Component：组件，OpenMax IL 的单元，每一个组件实现一种功能；
- 端口（Port）：组件的输入、输出接口；
- Tunneled：隧道化，让两个组件直接连接的方式。

OpenMax IL 层的运作流程如图 4-7 所示。



▲图 4-7 OpenMax IL 层的运作流程

在图 4-7 中，OpenMAL IL 层的客户端通过调用如下 4 个 OpenMAL IL 组件来实现同一个功能。

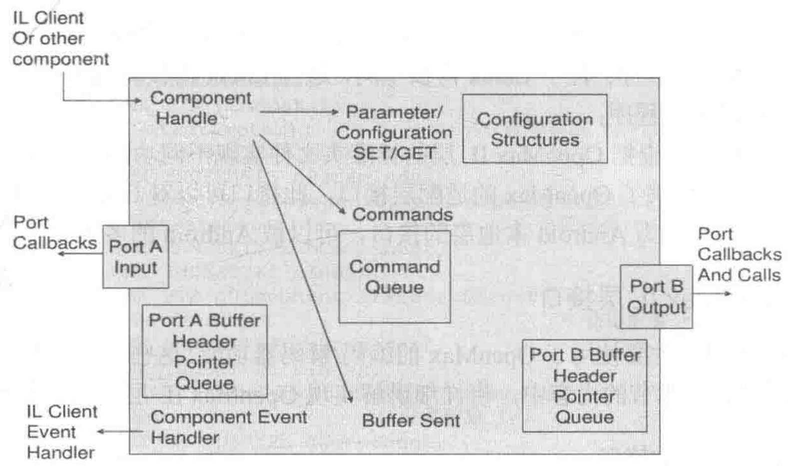
- Source 组件：只有一个输出端口；
- Host 组件：有一个输入端口和一个输出端口；
- Accelerator 组件：具有一个输入端口，调用了硬件的编码/解码器器，加速主要体现在此环节上；
- Sink 组件：Accelerator 组件和 Sink 组件通过私有通信方式在内部进行连接，没有经过明确的组件端口。

在使用 OpenMAL IL 的时候，既可以经由客户端处理数据流，也可以不经由客户端处理数据流。在图 4-7 中，Source 组件到 Host 组件的数据流就是经过客户端的；而 Host 组件到 Accelerator 组件的数据流就没有经过客户端，使用了隧道化的方式；Accelerator 组件和 Sink 组件甚至可以使用私有的通信方式。

OpenMax Core 是辅助组件正常运行的模块，它的任务是完成各个组件的初始化等工作。在具体运行时，需要重点初始化 OpenMax IL 组件，而不是初始化 OpenMax Core 组件。

在 OpenMax IL 层中，真正的核心内容是 OpenMAL IL 组件，此组件分别以输入端和输出端为接口，端口可以被连接到另一个组件上。外部对组件可以发送命令，还进行设置/获取参数、配置等。组件的端口可以包含缓冲区（Buffer）的队列。

在 OpenMax IL 层中，组件的处理的核心内容是通过输入端口来消耗 Buffer，通过输出端口来填充 Buffer，这样做的好处是通过多个组件的相互连接而构成流式处理。在 OpenMAL IL 层中，一个组件的基本结构如图 4-8 所示。



▲图 4-8 OpenMAL IL 层中的组件结构

组件的功能和定义端口的类型有着密切的联系，在大多数情况下的具体联系如下所示。

- 只有一个输出端口的是 Source 组件；
- 只有一个输入端口的是 Sink 组件；
- 有多个输入端口、一个输出端口的是 Mux 组件；
- 有一个输入端口、多个输出端口的是 DeMux 组件。输入和输出端口各一个组件的为中间处理环节，这是最常见的组件。

端口根据应用来支持不同的数据类型。假如在输入端和输出端各有一个组件，如果输入端口使用的是 MP3 格式数据，而在输出端口使用的是 PCM 格式数据，那么此组件就是一个 MP3 解码组件。

注意 上述组件连接的方式有一个专业术语——隧道化。通过隧道化（Tunneled）的方式可以将不同的组件的一个输入端口和一个输出端口连接到一起，此时会合并两个组件的处理过程并实现共同处理，合二为一。

3. Android 中的 OpenMax

在 Android 系统中，主要使用的是标准 OpenMax IL 层的接口，在里面只是进行了简单的封装。通过使用标准的 OpenMax IL 实现，可以很容易将 OpenMax IL 以插件的形式加入到 Android 系统中。

无论是 OpenCore 引擎，还是 StageFright 引擎，都可以使用 OpenMax 作为多媒体编码/解码器的插件，但是并没有直接使用 OpenMax IL 层提供的纯 C 的接口，而只是对其进行了简单的封装处理。

Android 系统对 OpenMax 支持的力度逐渐扩大，在 Android 2.x 版本之后，Android 的框架层开始封装定义 OpenMax IL 层接口，甚至使用 Android 中的 Binder IPC 机制来调用。在 Stagefright 中使用了 OpenMax IL 层接口，但是没有使用 OpenCore。OpenCore 使用在 OpenMax IL 层作为编码/解码器插件在前，Android 框架层封装 OpenMax 接口在后面的版本中才引入。

在 Android 系统中，主要使用了 OpenMax 的编码/解码器功能，而且使用最多的仍然是编码/

解码器组件，尽管 OpenMax 也可以生成输入、输出、文件解析/构建等组件。主要原因有如下两条。

（1）媒体输入/输出环节和系统有很大的关系，如果硬要使用 OpenMax 标准会比较麻烦；

（2）文件解析/构建环节一般不需要使用硬件加速。因为编码/解码器组件最能体现硬件加速环节，所以最常使用。

在 Android 系统中，当实现 OpenMax IL 层和标准的 OpenMax IL 层时需要实现如下两个环节。

- 编码/解码器驱动程序：位于 Linux 内核空间，通过 Linux 内核调用驱动程序，调用的驱动程序通常是非标准的驱动程序；
- OpenMax IL 层：根据 OpenMax IL 层的标准头文件实现不同功能的组件。

另外，Android 还提供了 OpenMax 的适配层接口，此接口可以对 OpenMax IL 的标准组件进行封装并适配。此接口作为 Android 本地层的接口，可以被 Android 的多媒体引擎随时调用。

4.2.2　实现 OpenMax IL 层接口

在 Android 系统中，主要使用了 OpenMax 的编码/解码器功能，这些功能主要是通过 OpenMax IL 层的接口实现的。在本节的内容中，将详细讲解实现 OpenMax IL 层接口的基本知识。

1. OpenMax IL 层的接口

（1）头文件。

在 OpenMax IL 层的接口中定义了由若干个头文件，被保存在“frameworks\native\include\media\openmax\”目录中。在这些文件中定义了实现 OpenMax IL 层接口的内容，这些头文件的具体说明如下所示。

- OMX_Types.h：OpenMax IL 的数据类型定义；
- OMX_Core.h：OpenMax IL 核心的 API；
- OMX_Component.h：OpenMax IL 组件相关的 API；
- OMX_Audio.h：音频相关的常量和数据结构；
- OMX_IVCommon.h：图像和视频公共的常量和数据结构；
- OMX_Image.h：图像相关的常量和数据结构；
- OMX_Video.h：视频相关的常量和数据结构；
- OMX_Other.h：其他数据结构（包括 A/V 同步）；
- OMX_Index.h：OpenMax IL 定义的数据结构索引；
- OMX_ContentPipe.h：内容的管道定义；

在 OpenMax 标准中只有头文件，没有标准的库。

（2）实现过程。

在具体实现 OpenMax IL 层的接口时，程序员主要实现包含函数指针的结构体，上述头文件中的实现流程如下。

- 在文件 frameworks\native\include\media\openmax\OMX_Component.h 中定义的 OMX_COMPONENTTYPE 结构体是 OpenMax IL 层的核心内容，表示一个组件，其实现代码如下所示。

```
typedef struct OMX_COMPONENTTYPE
{
    OMX_U32 nSize;                                          /* 定义此结构体的大小 */
    OMX_VERSIONTYPE nVersion;                               /* 版本号 */
    OMX_PTR pComponentPrivate;                              /* 此组件的私有数据指针 */
    /* 调用者（IL client）设置的指针，用于保存它的私有数据，传回给所有的回调函数 */
    OMX_PTR pApplicationPrivate;
    /* 下面的函数指针返回 OMX_core.h 中的对应内容 */
    OMX_ERRORTYPE (*GetComponentVersion)(
```

```
/* 获得组件的版本*/
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_OUT OMX_STRING pComponentName,
        OMX_OUT OMX_VERSIONTYPE* pComponentVersion,
        OMX_OUT OMX_VERSIONTYPE* pSpecVersion,
        OMX_OUT OMX_UUIDTYPE* pComponentUUID);
   OMX_ERRORTYPE (*SendCommand)(           /* 发送命令 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_COMMANDTYPE Cmd,
        OMX_IN  OMX_U32 nParam1,
        OMX_IN  OMX_PTR pCmdData);
   OMX_ERRORTYPE (*GetParameter)(           /* 获得参数 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_INDEXTYPE nParamIndex,
        OMX_INOUT OMX_PTR pComponentParameterStructure);
   OMX_ERRORTYPE (*SetParameter)(             /* 设置参数 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_INDEXTYPE nIndex,
        OMX_IN  OMX_PTR pComponentParameterStructure);
   OMX_ERRORTYPE (*GetConfig)(                       /* 获得配置 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_INDEXTYPE nIndex,
        OMX_INOUT OMX_PTR pComponentConfigStructure);
   OMX_ERRORTYPE (*SetConfig)(        /* 设置配置 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_INDEXTYPE nIndex,
        OMX_IN  OMX_PTR pComponentConfigStructure);
   OMX_ERRORTYPE (*GetExtensionIndex)(        /* 转换成 OMX 结构的索引 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_STRING cParameterName,
        OMX_OUT OMX_INDEXTYPE* pIndexType);
   OMX_ERRORTYPE (*GetState)(            /* 获得组件当前的状态 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_OUT OMX_STATETYPE* pState);
   OMX_ERRORTYPE (*ComponentTunnelRequest)(       /* 用于连接到另一个组件*/
        OMX_IN  OMX_HANDLETYPE hComp,
        OMX_IN  OMX_U32 nPort,
        OMX_IN  OMX_HANDLETYPE hTunneledComp,
        OMX_IN  OMX_U32 nTunneledPort,
        OMX_INOUT  OMX_TUNNELSETUPTYPE* pTunnelSetup);
   OMX_ERRORTYPE (*UseBuffer)(                 /* 为某个端口使用 Buffer */
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
        OMX_IN OMX_U32 nPortIndex,
        OMX_IN OMX_PTR pAppPrivate,
        OMX_IN OMX_U32 nSizeBytes,
        OMX_IN OMX_U8* pBuffer);
   OMX_ERRORTYPE (*AllocateBuffer)(          /* 在某个端口分配 Buffer */
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_INOUT OMX_BUFFERHEADERTYPE** ppBuffer,
        OMX_IN OMX_U32 nPortIndex,
        OMX_IN OMX_PTR pAppPrivate,
        OMX_IN OMX_U32 nSizeBytes);
   OMX_ERRORTYPE (*FreeBuffer)(         /*将某个端口 Buffer 释放*/
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_U32 nPortIndex,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
   OMX_ERRORTYPE (*EmptyThisBuffer)(          /* 让组件消耗此 Buffer */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
   OMX_ERRORTYPE (*FillThisBuffer)(             /* 让组件填充此 Buffer */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
   OMX_ERRORTYPE (*SetCallbacks)(            /* 设置回调函数 */
        OMX_IN  OMX_HANDLETYPE hComponent,
        OMX_IN  OMX_CALLBACKTYPE* pCallbacks,
        OMX_IN  OMX_PTR pAppData);
   OMX_ERRORTYPE (*ComponentDeInit)(             /* 反初始化组件 */
        OMX_IN  OMX_HANDLETYPE hComponent);
```

```
    OMX_ERRORTYPE (*UseEGLImage)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN void* eglImage);
    OMX_ERRORTYPE (*ComponentRoleEnum)(
            OMX_IN OMX_HANDLETYPE hComponent,
         OMX_OUT OMX_U8 *cRole,
         OMX_IN OMX_U32 nIndex);
} OMX_COMPONENTTYPE;
```

在实现上述 OMX_COMPONENTTYPE 结构体后，其中调用者可以使用的内容就是各个函数指针，而这些函数指针和文件 OMX_core.h 中定义的内容相对应。例如在文件 OMX_core.h 中定义 OMX_FreeBuffer 的代码如下所示。

```
#define OMX_FreeBuffer(
        hComponent,
        nPortIndex,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FreeBuffer(
        hComponent,
        nPortIndex,
        pBuffer)
```

在文件 OMX_core.h 中定义 OMX_FillThisBuffer 的代码如下所示。

```
#define OMX_FillThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FillThisBuffer(
        hComponent,
        pBuffer)
```

- 接下来需要定义组件运行机制。其中 EmptyThisBuffer 和 FillThisBuffer 是驱动组件运行的基本机制，前者表示让组件消耗缓冲区，表示对应组件输入的内容；后者表示让组件填充缓冲区，表示对应组件输出的内容。其中定义 OMX_EmptyThisBuffer 的代码如下所示。

```
#define OMX_EmptyThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->EmptyThisBuffer(
        hComponent,
        pBuffer)
```

其中定义 FillThisBuffer 的代码如下所示。

```
#define OMX_FillThisBuffer(
        hComponent,
        pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FillThisBuffer(
        hComponent,
        pBuffer)
```

- 然后开始定义和端口相关的缓冲区管理函数，这些函数分别是 UseBuffer、AllocateBuffer、FreeBuffer，对于组件的端口，有些可以自己分配缓冲区，有些可以使用外部的缓冲区，因此由不同的接口对其进行操作。
- 使用 SendCommand 向组件发送控制类的命令。接口 GetParameter、SetParameter、GetConfig、SetConfig 用于辅助的参数和配置的设置和获取。具体代码如下所示。

```
#define OMX_GetParameter(
        hComponent,
        nParamIndex,
```

```
        pComponentParameterStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->GetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
#define OMX_SetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->SetParameter(
        hComponent,
        nParamIndex,
        pComponentParameterStructure)
#define OMX_GetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->GetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
#define OMX_SetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
    ((OMX_COMPONENTTYPE*)hComponent)->SetConfig(
        hComponent,
        nConfigIndex,
        pComponentConfigStructure)
```

- 然后使用 ComponentTunnelRequest 实现组件之间的隧道化连接，在此需要指定两个组件及其相连的端口。
- 接下来使用 ComponentDeInit 来反初始化组件。在文件 OMX_Component.h 中定义的端口类型为 OMX_PORTDOMAINTYPE 枚举类型，此枚举的定义代码如下所示。

```
typedef enum OMX_PORTDOMAINTYPE {
    OMX_PortDomainAudio,        /* 音频类型端口 */
    OMX_PortDomainVideo,        /* 视频类型端口 */
    OMX_PortDomainImage,        /* 图像类型端口 */
    OMX_PortDomainOther,        /* 其他类型端口 */
    OMX_PortDomainKhronosExtensions = 0x6F000000,
    OMX_PortDomainVendorStartUnused = 0x7F000000
    OMX_PortDomainMax = 0x7ffffff
} OMX_PORTDOMAINTYPE;
```

在上述代码中，分别定义了音频类型、视频类型和图像类型，其他类型则是 OpenMax IL 层此所定义的 4 种端口的类型。

- 使用 OMX_PARAM_PORTDEFINITIONTYPE 类（也在 OMX_Component.h 中定义）来定义端口的具体内容，其实现代码如下所示。

```
typedef struct OMX_PARAM_PORTDEFINITIONTYPE {
    OMX_U32 nSize;                          /* 结构体大小 */
    OMX_VERSIONTYPE nVersion;               /* 版本*/
    OMX_U32 nPortIndex;                     /* 端口号 */
    OMX_DIRTYPE eDir;                       /* 端口的方向 */
    OMX_U32 nBufferCountActual;             /* 为此端口实际分配的 Buffer 的数目 */
    OMX_U32 nBufferCountMin;                /* 此端口最小 Buffer 的数目*/
    OMX_U32 nBufferSize;                    /* 缓冲区的字节数 */
    OMX_BOOL bEnabled;                      /* 是否使能 */
    OMX_BOOL bPopulated;                    /* 是否在填充 */
    OMX_PORTDOMAINTYPE eDomain;             /* 端口的类型 */
    union {                                 /* 端口实际的内容，由类型确定具体结构 */
        OMX_AUDIO_PORTDEFINITIONTYPE audio;
        OMX_VIDEO_PORTDEFINITIONTYPE video;
        OMX_IMAGE_PORTDEFINITIONTYPE image;
```

```
        OMX_OTHER_PORTDEFINITIONTYPE other;
    } format;
    OMX_BOOL bBuffersContiguous;
    OMX_U32 nBufferAlignment;
} OMX_PARAM_PORTDEFINITIONTYPE;
```

对于上述代码的具体说明如下所示。

- OMX_DIRTYPE：端口的方向，包含如下两种。

➢ OMX_DirInput：输入；

➢ OMX_DirOutput：输出。

- 端口格式的数据结构：使用 format 联合体来表示，具体由如下 4 种不同类型来表示，与端口的类型相对应。

➢ OMX_AUDIO_PORTDEFINITIONTYPE；

➢ OMX_VIDEO_PORTDEFINITIONTYPE；

➢ OMX_IMAGE_PORTDEFINITIONTYPE；

➢ OMX_OTHER_PORTDEFINITIONTYPE。

上述类型分别在头文件 OMX_Audio.h、OMX_Video.h、OMX_Image.h 和 OMX_Other.h 中定义。

- OMX_BUFFERHEADERTYPE：表示一个缓冲区的头部结构，在 OMX_Core.h 中定义。
- 使用在文件 OMX_Core.h 中定义的枚举类型 OMX_STATETYPE 来表示 OpenMax 的状态，主要代码如下所示。

```
typedef enum OMX_STATETYPE
{
    OMX_StateInvalid,                    /* 如果组件监测到内部的数据结构被破坏 */
    OMX_StateLoaded,                     /* 如果组件被加载但是没有完成初始化 */
    OMX_StateIdle,                       /* 如果组件初始化完成，准备开始 */
    OMX_StateExecuting,                  /* 如果组件接受了开始命令，正在树立数据 */
    OMX_StatePause,                      /* 如果组件接受暂停命令*/
    OMX_StateWaitForResources,           /* 如果组件正在等待资源 */
    OMX_StateKhronosExtensions = 0x6F000000, /* 保留 */
    OMX_StateVendorStartUnused = 0x7F000000, /* 保留 */
    OMX_StateMax = 0X7FFFFFFF
} OMX_STATETYPE;
```

- 在文件 OMX_Core.h 中定义的枚举类型还有 OMX_COMMANDTYPE，此枚举表示对组件的命令类型，主要代码如下所示。

```
typedef enum OMX_COMMANDTYPE
{
    OMX_CommandStateSet,                 /* 改变状态机器 */
    OMX_CommandFlush,                    /* 刷新数据队列 */
    OMX_CommandPortDisable,              /* 禁止端口 */
    OMX_CommandPortEnable,               /* 使能端口 */
    OMX_CommandMarkBuffer,               /* 标记组件或 Buffer 用于观察 */
    OMX_CommandKhronosExtensions = 0x6F000000, /* 保留 */
    OMX_CommandVendorStartUnused = 0x7F000000, /* 保留 */
    OMX_CommandMax = 0X7FFFFFFF
} OMX_COMMANDTYPE;
```

注意　在 OpenMax 的函数参数中，经常包含 OMX_IN 和 OMX_OUT 等宏，它们的实际内容为空，只是为了标记参数的方向是输入还是输出。

2. 在 OpenMax IL 层中工作

在实现 OpenMax IL 层时一般不调用 OpenMax DL 层，具体实现的内容是各个不同的组件。通常通过以下两个步骤来实现 OpenMax IL 组件。

（1）组件的初始化函数。

包括硬件和 OpenMax 数据结构的初始化，主要步骤如下所示。

- 初始化函数指针；
- 初始化私有数据结构；
- 初始化端口。

在实现上述步骤的过程中，可以使用其中的 pComponentPrivate 成员保留本组件的私有数据为上下文，在最后获得填充完成 OMX_COMPONENTTYPE 类型的结构体。

（2）OMX_COMPONENTTYPE 类型结构体的各个指针。

在此需要实现其中的各个函数指针，当需要用到私有数据的时候，先从 pComponentPrivate 中得到指针，然后转化成实际的数据结构。

因为在 OpenMax IL 层中，经常用到的组件大多数是一个输入端口和一个输出端口，所以端口定义的是 OpenMax IL 组件对外部的接口。对于最常用的编解码（Codec）组件来说，通常需要在每个组件的实现过程中，调用硬件的编解码接口来实现。在组件的内部处理中可以通过建立线程来处理。在 OpenMax 组件的端口中有默认参数，但也可以在运行时设置，因此一个端口也可以支持不同的编码格式。音频编码组件的输出和音频编码组件的输入通常是原始数据格式（PCM 格式），视频编码组件的输出和视频编码组件的输入通常也是原始数据格式（YUV 格式）。

3. OpenMax 适配层

Android 系统中的 OpenMax 适配层的接口在如下文件中定义。

```
frameworks/av/include/media/IOMX.h
```

文件 IOMX.h 的主要代码如下所示。

```
class IOMX : public IInterface {
public:
   DECLARE_META_INTERFACE(OMX);
   typedef void *buffer_id;
   typedef void *node_id;
   virtual bool livesLocally(pid_t pid) = 0;
   struct ComponentInfo {                        // 组件的信息
      String8 mName;
      List<String8> mRoles;
   };
   virtual status_t listNodes(List<ComponentInfo> *list) = 0;  // 节点列表
   virtual status_t allocateNode(
         const char *name, const sp<IOMXObserver> &observer,  // 分配节点
         node_id *node) = 0;
   virtual status_t freeNode(node_id node) = 0;  // 找到节点
   virtual status_t sendCommand(            // 发送命令
         node_id node, OMX_COMMANDTYPE cmd, OMX_S32 param) = 0;
   virtual status_t getParameter(           // 获得参数
         node_id node, OMX_INDEXTYPE index,
         void *params, size_t size) = 0;
   virtual status_t setParameter(          // 设置参数
         node_id node, OMX_INDEXTYPE index,
         const void *params, size_t size) = 0;
   virtual status_t getConfig(             // 获得配置
         node_id node, OMX_INDEXTYPE index,
         void *params, size_t size) = 0;
   virtual status_t setConfig(       // 设置配置
         node_id node, OMX_INDEXTYPE index,
         const void *params, size_t size) = 0;
   virtual status_t useBuffer(        // 使用缓冲区
         node_id node, OMX_U32 port_index, const
 sp<IMemory> &params,
```

```
            buffer_id *buffer) = 0;
    virtual status_t allocateBuffer(    // 分配缓冲区
            node_id node, OMX_U32 port_index, size_t size,
            buffer_id *buffer, void **buffer_data) = 0;
    virtual status_t allocateBufferWithBackup(      // 分配后备缓冲区
            node_id node, OMX_U32 port_index, const
sp<IMemory> &params,
            buffer_id *buffer) = 0;
    virtual status_t freeBuffer(        // 释放缓冲区
            node_id node, OMX_U32 port_index,buffer_id buffer) = 0;
    virtual status_t fillBuffer(node_id node,  buffer_id buffer) = 0; // 填充缓冲区
    virtual status_t emptyBuffer( // 消耗缓冲区
            node_id node,
            buffer_id buffer,
            OMX_U32 range_offset, OMX_U32 range_length,
            OMX_U32 flags, OMX_TICKS timestamp) = 0;
    virtual status_t getExtensionIndex(
            node_id node,
            const char *parameter_name,
            OMX_INDEXTYPE *index) = 0;
    virtual sp<IOMXRenderer> createRenderer( // 创建渲染器（从 ISurface）
            const sp<ISurface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t encodedWidth, size_t encodedHeight,
            size_t displayWidth, size_t displayHeight) = 0;
    sp<IOMXRenderer> createRenderer(    // 创建渲染器（从 Surface）
            const sp<Surface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t encodedWidth, size_t encodedHeight,
            size_t displayWidth, size_t displayHeight);
    sp<IOMXRenderer> createRendererFromJavaSurface(  // 从 Java 层创建渲染器
            JNIEnv *env, jobject javaSurface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t encodedWidth, size_t encodedHeight,
            size_t displayWidth, size_t displayHeight);
};
```

在 IOMX 中，只有第一个 createRenderer 函数是纯虚函数，第二个函数 createRenderer 和函数 createRendererFromJavaSurface 通过调用第一个 createRenderer 函数实现。

类 IOMXRenderer 表示了一个 OpenMax 的渲染器，定义此类的代码如下所示。

```
class IOMXRenderer : public IInterface {
public:
    DECLARE_META_INTERFACE(OMXRenderer);
    virtual void render(IOMX::buffer_id buffer) = 0;  // 渲染输出函数
};
```

在类 IOMXRenderer 中只包含了一个 Render 接口，其参数类型 IOMX::buffer_id 其实是 void*，可以根据不同的渲染器使用不同的类型。

在文件 IOMX.h 中还存在一个观察器类 IOMXObserver，此类表示 OpenMax 的观察者。在里面包含了函数 onMessage()，其参数是 omx_message 接口体，其中包含 Event 事件类型、FillThisBuffer 完成和 EmptyThisBuffer 完成等几种类型。

4.3 分析 OpenCore 框架

在本节的内容中，将详细讲解 OpenCore 框架的基本知识，分别介绍其结构和插件机制，为读者进入本书后面知识的学习打下基础。

4.3.1 OpenCore 层次结构

在 Android 系统中，OpenCore 的另外一个常用的称呼是 PacketVideo，它是 Android 多媒体系统的核心。其实 PacketVideo 是一家公司的名称，而 OpenCore 是这套多媒体框架的软件层的名称。在 Android 开发者的眼中，二者的含义基本相同。与其他 Android 程序库相比，OpenCore 的代码非常庞大，它是基于 C++实现的，并定义了全功能的操作系统移植层，各种基本功能均被封装成类的形式，各层次之间的接口使用继承等方式实现。

Android 系统中的 OpenCore 是一个多媒体的框架，从宏观上来看主要包含了如下两方面的内容。

（1）PVPlayer。

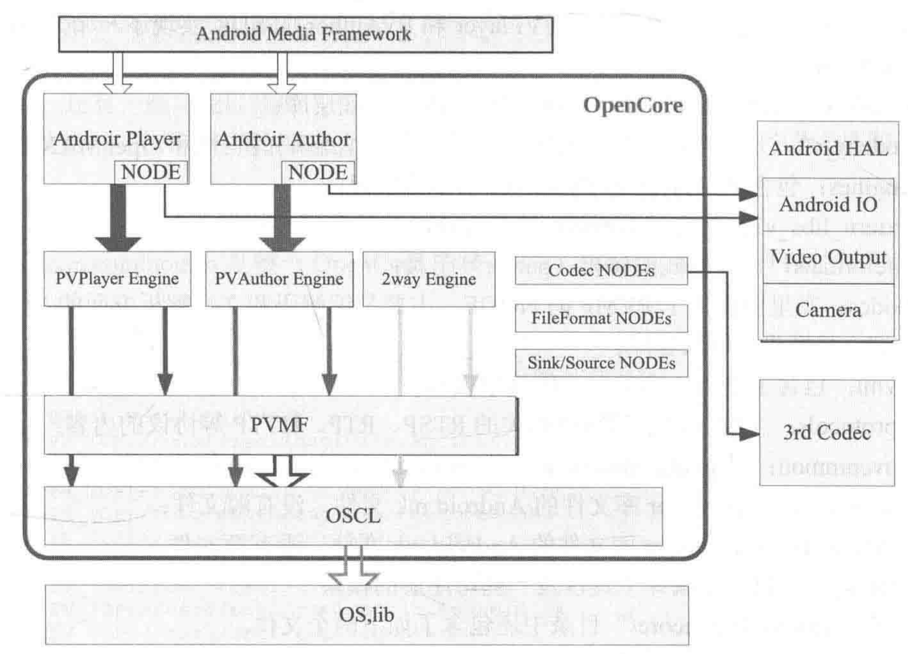
PVPlayer 提供了媒体播放器的功能，可以完成各种音频（Audio）、视频（Video）流的回放（Playback）功能。

（2）PVAuthor。

PVPlayer 提供了媒体流的记录功能，可以完成各种音频（Audio）、视频（Video）以及静态图像捕获功能。

PVPlayer 和 PVAuthor 以 SDK 的形式提供给开发者，可以在这个 SDK 之上构建多种应用程序和服务。在移动终端中常常使用的多媒体应用程序，例如媒体播放器、照相机、录像机、录音机等。

OpenCore 层次的基本结构如图 4-9 所示。



▲图 4-9　OpenCore 层次的基本结构

在图 4-9 所示的结构中，主要层次元素的具体说明如下所示。

（1）OSCL。

OSCL 是 Operating System Compatibility Library 的缩写，意为操作系统兼容库。在 OSCL 中包含了一些操作系统底层的操作，目的是更好地在不同操作系统移植。在 OSCL 中包含的系统底层操作有基本数据类型、配置、字符串工具、IO、错误处理、线程等内容，类似一个基础的 C++库。

（2）PVMF。

PVMF 是 PacketVideo Multimedia Framework 的缩写，意为 PV 多媒体框架。PVMF 可以在框架内实现一个文件解析（parser）和组成（composer）、编解码的 NODE，也可以继承其通用的接口，在用户层实现一些 NODE。

（3）PVPlayer Engine：是 PVPlayer 引擎。

（4）PVAuthor Engine：是 PVAuthor 引擎。

除了上述 4 个元素外，其实在 OpenCore 中包含的内容还有很多。从播放的角度看，PVPlayer 输入（Source）的是文件或者网络媒体流，输出（Sink）的是音频视频的输出设备，其基本功能包含了媒体流控制、文件解析、音频视频流的解码（Decode）等方面的内容。除了从文件中播放媒体文件之外，还包含了与网络相关的 RTSP（Real Time Stream Protocol，实时流协议）流。在媒体流记录的方面，PVAuthor 输入（Source）的是照相机、麦克风等设备，输出（Sink）的是各种文件，包含了流的同步、音频视频流的编码（Encode）以及文件的写入等功能。

在使用 OpenCore SDK 的时候，有可能需要在应用层实现一个适配器（Adaptor），然后在适配器之上实现具体的功能，对于 PVMF 的 NODE 也可以基于通用的接口，在上层实现，以插件的形式使用。

4.3.2　OpenCore 代码结构

在 Android 系统中，OpenCore 的代码保存在“external/opencore/”目录中，此目录是 OpenCore 的根目录，其中包含的各个子目录的具体说明如下所示。

（1）android：是一个上层库，基于 PVPlayer 和 PVAuthor 的 SDK 实现了一个为 Android 使用的 Player 和 Author；

（2）baselibs：包含了数据结构和线程安全等内容的底层库；

（3）codecs_v2：是一个内容较多的库，主要包含了编/解码的实现和 OpenMAX 的实现；

（4）engines：包含 PVPlayer 和 PVAuthor 引擎的实现；

（5）extern_libs_v2：包含了 khronos 的 OpenMAX 的头文件；

（6）fileformats：文件格式的解析（parser）工具；

（7）nodes：在里面提供了 PVMF 的 NODE，主要是编解码和文件解析方面的 NODE；

（8）oscl：是操作系统兼容库；

（9）pvmi：包含了输入、输出控制的抽象接口；

（10）protocols：主要包含了和网络相关的 RTSP、RTP、HTTP 等协议的内容；

（11）pvcommon：是 pvcommon 库文件的 Android.mk 文件，没有源文件；

（12）pvplayer：是 pvplayer 库文件的 Android.mk 文件，没有源文件；

（13）pvauthor：是 pvauthor 库文件的 Android.mk 文件，没有源文件。

（14）tools_v2：包含了编译工具以及一些可注册的模块。

另外，在“external/opencore/”目录中还包含了如下两个文件。

- Android.mk：全局的编译文件；
- pvplayer.conf：配置文件。

在“external/opencore/”的各个子文件夹中还包含了很多个 Android.mk 文件，在这些文件之间存在着“递归”的关系。例如在根目录下的 Android.mk 中包含了下面的内容片断。

- include $(PV_TOP)/pvcommon/Android.mk；
- include $(PV_TOP)/pvplayer/Android.mk；
- include $(PV_TOP)/pvauthor/Android.mk。

这表示要引用 pvcommon、pvplayer 和 pvauthor 等目录下面的 Android.mk 文件。“external/opencore/”目录中各个 Android.mk 文件可以按照排列组合进行使用，可以将几个 Android.mk 内容合并在一个库里面。

4.3.3　OpenCore 编译结构

在 Android 开源系统中，通过 OpenCore 编译出来的各个库的具体说明如下所示。

- libopencoreauthor.so：OpenCore 的 Author 库；
- libopencorecommon.so：OpenCore 底层的公共库；
- libopencoredownloadreg.so：下载注册库；
- libopencoredownload.so：下载功能实现库；
- libopencoremp4reg.so：MP4 注册库；
- libopencoremp4.so：MP4 功能实现库；
- libopencorenet_support.so：网络支持库；
- libopencoreplayer.so：OpenCore 的 Player 库；
- libopencorertspreg.so：RTSP 注册库；
- libopencorertsp.so：RTSP 功能实现库。

OpenCore 中的各个库之间的关系如下所示。

- libopencorecommon.so：是所有库的依赖库，提供了公共的功能；
- libopencoreplayer.so 和 libopencoreauthor.so：是两个并立的库，分别用于回放和记录，而且这两个库是 OpenCore 对外的接口库；
- libopencorenet_support.so：提供网络支持的功能。

除此之外，还有一些功能以插件（Plug-In）的方式放入 Player 中使用，每个功能使用两个库，一个实现具体功能，一个用于注册。在接下来的内容中，将简要介绍 OpenCore 中各个库的基本结构。

1. 库 libopencorecommon.so 的结构

库 libopencorecommon.so 是整个 OpenCore 的核心库，其编译控制的文件路径如下所示。

```
pvcommon/Android.mk
```

上述文件使用递归的方式寻找子文件，其主要内容如下所示。

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//oscl/oscl/osclbase/Android.mk
include $(PV_TOP)//oscl/oscl/osclerror/Android.mk
include $(PV_TOP)//oscl/oscl/osclmemory/Android.mk
include $(PV_TOP)//oscl/oscl/osclutil/Android.mk
include $(PV_TOP)//oscl/pvlogger/Android.mk
include $(PV_TOP)//oscl/oscl/osclproc/Android.mk
include $(PV_TOP)//oscl/oscl/osclio/Android.mk
include $(PV_TOP)//oscl/oscl/osclregcli/Android.mk
include $(PV_TOP)//oscl/oscl/osclregserv/Android.mk
include $(PV_TOP)//oscl/unit_test/Android.mk
include $(PV_TOP)//oscl/oscl/oscllib/Android.mk
include $(PV_TOP)//pvmi/pvmf/Android.mk
include $(PV_TOP)//baselibs/pv_mime_utils/Android.mk
include $(PV_TOP)//nodes/pvfileoutputnode/Android.mk
include $(PV_TOP)//baselibs/media_data_structures/Android.mk
include $(PV_TOP)//baselibs/threadsafe_callback_ao/Android.mk
include $(PV_TOP)//codecs_v2/utilities/colorconvert/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/common/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/common/Android.mk
```

这些被包含的 Android.mk 文件真正指定需要编译的文件，它们在 Android.mk 的目录及其子目录中。事实上，在 libopencorecommon.so 库中包含了以下内容。

- OSCL 的所有内容；
- Pvmf 框架部分的内容（pvmi/pvmf/Android.mk）；
- 基础库中的一些内容（baselibs）；
- 编解码的一些内容；
- 文件输出的 node（nodes/pvfileoutputnode /Android.mk）。

从库 libopencorecommon.so 的结构可以看出，最终生成库的结构与 OpenCore 的层次关系并非完全重合。在库 libopencorecommon.so 中已经包含了底层的 OSCL 的内容、PVMF 的框架以及 Node 和编解码的工具。

2. 库 libopencoreplayer.so 的结构

库 libopencoreplayer.so 是一个用于实现播放功能的库，其编译控制的文件的路径如下所示。

```
pvplayer/Android.mk
```

上述文件 Android.mk 的主要代码如下所示。

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//engines/player/Android.mk
include $(PV_TOP)//codecs_v2/audio/aac/dec/util/getactualaacconfig/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/aac/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_wb/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/common/dec/Android.mk
include $(PV_TOP)//codecs_v2/audio/mp3/dec/Android.mk
include $(PV_TOP)//codecs_v2/utilities/m4v_config_parser/Android.mk
include $(PV_TOP)//codecs_v2/utilities/pv_video_config_parser/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_common/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_queue/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_h264/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_aac/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_amr/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_mp3/Android.mk
include $(PV_TOP)//codecs_v2/omx/factories/omx_m4v_factory/Android.mk
include $(PV_TOP)//codecs_v2/omx/omx_proxy/Android.mk
include $(PV_TOP)//nodes/common/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/plugins/oma1/passthru/Android.mk
include $(PV_TOP)//pvmi/content_policy_manager/plugins/common/Android.mk
include $(PV_TOP)//pvmi/media_io/pvmiofileoutput/Android.mk
include $(PV_TOP)//fileformats/common/parser/Android.mk
include $(PV_TOP)//fileformats/id3parcom/Android.mk
include $(PV_TOP)//fileformats/rawgsmamr/parser/Android.mk
include $(PV_TOP)//fileformats/mp3/parser/Android.mk
include $(PV_TOP)//fileformats/mp4/parser/Android.mk
include $(PV_TOP)//fileformats/rawaac/parser/Android.mk
include $(PV_TOP)//fileformats/wav/parser/Android.mk
include $(PV_TOP)//nodes/pvaacffparsernode/Android.mk
include $(PV_TOP)//nodes/pvmp3ffparsernode/Android.mk
include $(PV_TOP)//nodes/pvamrffparsernode/Android.mk
include $(PV_TOP)//nodes/pvmediaoutputnode/Android.mk
include $(PV_TOP)//nodes/pvomxvideodecnode/Android.mk
include $(PV_TOP)//nodes/pvomxaudiodecnode/Android.mk
include $(PV_TOP)//nodes/pvwavffparsernode/Android.mk
include $(PV_TOP)//pvmi/recognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvamrffrecognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvmp3ffrecognizer/Android.mk
include $(PV_TOP)//pvmi/recognizer/plugins/pvwavffrecognizer/Android.mk
include $(PV_TOP)//engines/common/Android.mk
```

```
include $(PV_TOP)//engines/adapters/player/framemetadatautility/Android.mk
include $(PV_TOP)//protocols/rtp_payload_parser/util/Android.mk
include $(PV_TOP)//android/Android.mk
include $(PV_TOP)//android/drm/oma1/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_rtsp/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_rtsp/node_registry/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_net_support/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_download/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_download/node_registry/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_mp4/core/Android.mk
include $(PV_TOP)//tools_v2/build/modules/linux_mp4/node_registry/Android.mk
```

在库 libopencoreplayer.so 中包含了如下内容。

- 解码工具；
- 文件的解析器（MP4）；
- 解码工具对应的 Node；
- player 的引擎部分（路径是“engines/player/Android.mk”）；
- 为 Android 的 player 适配器（路径是“android/Android.mk”）；
- 识别工具（路径是“pvmi/recognizer”）；
- 编解码工具中的 OpenMax 部分（路径是“codecs_v2/omx”）；
- 对应几个插件 Node 的注册。

库 libopencoreplayer.so 中的内容较多，其中主要为各个文件解析器和解码器，PVPlayer 的核心功能在文件“engines/player/Android.mk”中。而文件“android/Android.mk”的内容比较特殊，它是在 PVPlayer 之上构建的一个为 Android 使用的播放器。

3. 库 libopencoreauthor.so 的结构

库 libopencoreauthor.so 是实现媒体流记录的功能库，其编译控制的文件的路径如下所示。

```
pvauthor/Android.mk
```

上述文件 Android.mk 的主要代码如下所示。

```
include $(BUILD_SHARED_LIBRARY)
include $(PV_TOP)//engines/author/Android.mk
include $(PV_TOP)//codecs_v2/video/m4v_h263/enc/Android.mk
include $(PV_TOP)//codecs_v2/audio/gsm_amr/amr_nb/enc/Android.mk
include $(PV_TOP)//codecs_v2/video/avc_h264/enc/Android.mk
include $(PV_TOP)//fileformats/mp4/composer/Android.mk
include $(PV_TOP)//nodes/pvamrencnode/Android.mk
include $(PV_TOP)//nodes/pvmp4ffcomposernode/Android.mk
include $(PV_TOP)//nodes/pvvideoencnode/Android.mk
include $(PV_TOP)//nodes/pvavcencnode/Android.mk
include $(PV_TOP)//nodes/pvmediainputnode/Android.mk
include $(PV_TOP)//android/author/Android.mk
```

在库 libopencoreauthor.so 中包含了如下内容。

- 编码工具，例如视频流 H263、H264，音频流 Amr；
- 文件的组成器，例如 MP4；
- 编码工具对应的 Node；
- 用于媒体输入的 Node（目录是“nodes/pvmediainputnode/Android.m”）；
- author 引擎（目录是“engines/author /Android.mk”）；
- Android 的 author 适配器（目录是“android/author/Android.mk”）。

在库 libopencoreauthor.so 中，其内容主要由各个文件编码器和文件组成器构成，其中 PVAuthor 的核心功能在“engines/author /Android.mk”目录中，而文件“android/author/Android.mk”是在

PVAuthor 之上构建的一个为 Android 使用的媒体记录器。

4. 其他库

除了前面介绍的 3 个库之外，在 OpenCore 中还有另外几个库，具体说明如下所示。

网络支持库 libopencorenet_support.so，对应的 Android.mk 文件的路径如下所示。

```
tools_v2/build/modules/linux_net_support/core/Android.mk
```

MP4 功能实现库 libopencoremp4.so 和注册库 libopencoremp4reg.so，对应的 Android.mk 文件的路径如下所示。

```
tools_v2/build/modules/linux_mp4/core/Android.mk
tools_v2/build/modules/linux_mp4/node_registry/Android.mk
```

RTSP 功能实现库 libopencorertsp.so 和注册库 libopencorertspreg.so，对应的 Android.mk 文件的路径如下所示。

```
tools_v2/build/modules/linux_rtsp/core/Android.mk
tools_v2/build/modules/linux_rtsp/node_registry/Android.mk
```

下载功能实现库 libopencoredownload.so 和注册库 libopencoredownloadreg.so，对应的 Android.mk 文件的路径如下所示。

```
tools_v2/build/modules/linux_download/core/Android.mk
tools_v2/build/modules/linux_download/node_registry/Android.mk
```

4.3.4　操作系统兼容库

OSCL 是 Operating System Compatibility Library（操作系统兼容库）的缩写，在里面包含了一些不同操作系统中移植层的功能，其代码结构如下所示。

```
oscl/oscl
|-- config：配置的宏
|-- makefile
|-- makefile.pv
|-- osclbase：包含基本类型、宏以及一些与 STL 容器类似的功能
|-- osclerror：错误处理的功能
|-- osclio：文件 IO 和 Socket 等功能
|-- oscllib：动态库接口等功能
|-- osclmemory：内存管理、自动指针等功能
|-- osclproc：线程、多任务通信等功能
|-- osclregcli：注册客户端的功能
|-- osclregserv：注册服务器的功能
`-- osclutil：字符串等基本功能
```

在目录“oscl”中，通常用一个目录表示一个模块。OSCL 对应的功能非常详细，几乎封装 C 语言中的每一个细节功能，并且提供了 C++接口为上层使用。其实在 OperCore 中的 PVMF 和 Engine 都在使用 OSCL，整个 OperCore 的调用者也需要使用 OSCL。

在实现 OSCL 时，简单封装了很多典型的 C 语言函数，例如 osclutil 中与数学相关的功能在 oscl_math.inl 中被定义成为了内嵌（inline）的函数，具体代码如下所示。

```
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_log(double value)
{
        return (double) log(value);
}
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_log10(double value)
{
        return (double) log10(value);
}
```

```
OSCL_COND_EXPORT_REF OSCL_INLINE double oscl_sqrt(double value)
{
        return (double) sqrt(value);
}
```

因为文件 oscl_math.inl 被 oscl_math.h 所包含，所以其结果是和函数 oscl_log 的功能等价的原始函数 log。

OSCL 的具体实现比较复杂，很多 C 语言标准库的句柄都被定义成 C++类的形式，实现起来会比较繁琐。尽管如此，OSCL 并不复杂。例如以 oscllib 为例，其代码结构如下所示。

```
oscl/oscl/oscllib/
|-- Android.mk
|-- build
|   `-- make
|       `-- makefile
`-- src
    |-- oscl_library_common.h
    |-- oscl_library_list.cpp
    |-- oscl_library_list.h
    |-- oscl_shared_lib_interface.h
    |-- oscl_shared_library.cpp
    `-- oscl_shared_library.h
```

其中文件 oscl_shared_library.h 是提供给上层使用的动态库的接口功能，定义的接口代码如下所示。

```
class OsclSharedLibrary {
public:
        OSCL_IMPORT_REF OsclSharedLibrary();
        OSCL_IMPORT_REF OsclSharedLibrary(const OSCL_String& aPath);
        OSCL_IMPORT_REF ~OsclSharedLibrary();
        OSCL_IMPORT_REF OsclLibStatus LoadLib(const OSCL_String& aPath);
        OSCL_IMPORT_REF OsclLibStatus LoadLib();
        OSCL_IMPORT_REF void SetLibPath(const OSCL_String& aPath);
        OSCL_IMPORT_REF  OsclLibStatus  QueryInterface(const  OsclUuid&  aInterfaceId,
OsclAny*& aInterfacePtr);
        OSCL_IMPORT_REF OsclLibStatus Close();
        OSCL_IMPORT_REF void AddRef();
        OSCL_IMPORT_REF void RemoveRef();
}
```

这些接口都与库的加载有关系，而在文件 oscl_shared_library.cpp 中，其具体的功能通过使用函数 dlopen 等来实现。

4.3.5 实现 OpenCore 中的 OpenMax 部分

在 OpenCore 框架中，OpenMax 是作为插件来实现的，只要封装了 OpenMax，就可以在 OpenCore 中使用标准的 OpenMax。

1. OpenMax 结构

在 OpenCore 中，在如下目录的头文件中包含标准的 OpenMax。

```
extern_libs_v2/khronos/openmax/include/
```

在文件“build_config/opencore_dynamic/Android_omx_aacdec_sharedlibrary.mk”中，声明了插件 OpenMax 的主要库是 libomx_sharedlibrary.so，主要代码如下所示。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_WHOLE_STATIC_LIBRARIES := \
```

```
      libomx_aac_component_lib \
      libpv_aac_dec
LOCAL_MODULE := libomx_aacdec_sharedlibrary
-include $(PV_TOP)/Android_platform_extras.mk
-include $(PV_TOP)/Android_system_extras.mk

LOCAL_SHARED_LIBRARIES +=  libomx_sharedlibrary libopencore_common

include $(BUILD_SHARED_LIBRARY)
include  $(PV_TOP)/codecs_v2/omx/omx_aac/Android.mk
include  $(PV_TOP)/codecs_v2/audio/aac/dec/Android.mk
```

库 libomx_sharedlibrary.so 为 omx 针对 OpenCore 的接口层库，也就是说在每个模拟器上 libomx_sharedlibrary.so 向外（即 OpenCore）提供的接口是一致的。此库可以动态打开各个 OpenMax 的编码/解码模块，各个编码/解码模块通过调用 codecs_v2 中 audio 和 video 目录中软件的编码/解码库来实现。

在“opencore”的根目录中，有一个名为 pvplayer.cfg 的文件，此文件用于实现 OpenCore 运行过程的动态配置，此文件的主要代码如下所示。

```
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_rtsp
reg.so"
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_down
loadreg.so"
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_mp4l
ocalreg.so"
(0x6d3413a0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_mp4l
ocalreg.so"
(0xa054369c,0x22c5,0x412e,0x19,0x17,0x87,0x4c,0x1a,0x19,0xd4,0x5f),"libomx_sharedlib
rary.so"
```

2. OpenMax 接口

在 OpenCore 中，OpenMax 接口是通过封装标准的 OpenMax IL 层来构建的。这些接口的基本内容相同，但是不同于标准的 OpenMax IL 层的 C 语言接口。在 OpenCore 中和 OpenMax 接口相关的头文件如下所示。

- opencore/codecs_v2/omx/omx_mastercore/include/omx_interface.h：定义插件接口。
- opencore/codecs_v2/omx/omx_common/include/pv_omxcore.h：核心定义。
- opencore/codecs_v2/omx/omx_baseclass/include/pv_omxcomponent.h：定义 PV 的 OpenMax 组件。

文件 omx_interface.h 定义了 OpenMax 接口的核心功能，在里面包含了各种函数指针的定义类型，具体实现代码如下所示。

```
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_Init)(void);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_Deinit)(void);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_ComponentNameEnum)(
    OMX_OUT OMX_STRING cComponentName,
    OMX_IN  OMX_U32 nNameLength,
    OMX_IN  OMX_U32 nIndex);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_GetHandle)(
    OMX_OUT OMX_HANDLETYPE* pHandle,
    OMX_IN  OMX_STRING cComponentName,
    OMX_IN  OMX_PTR pAppData,
    OMX_IN  OMX_CALLBACKTYPE* pCallBacks);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_FreeHandle)(
    OMX_IN  OMX_HANDLETYPE hComponent);
typedef OMX_ERRORTYPE(*tpOMX_GetComponentsOfRole)(
    OMX_IN      OMX_STRING role,
    OMX_INOUT   OMX_U32 *pNumComps,
    OMX_INOUT   OMX_U8  **compNames);
typedef OMX_ERRORTYPE(*tpOMX_GetRolesOfComponent)(
```

```
    OMX_IN      OMX_STRING compName,
    OMX_INOUT   OMX_U32 *pNumRoles,
    OMX_OUT     OMX_U8 **roles);
typedef OMX_ERRORTYPE OMX_APIENTRY(*tpOMX_SetupTunnel)(
    OMX_IN  OMX_HANDLETYPE hOutput,
    OMX_IN  OMX_U32 nPortOutput,
    OMX_IN  OMX_HANDLETYPE hInput,
    OMX_IN  OMX_U32 nPortInput);
typedef OMX_ERRORTYPE(*tpOMX_GetContentPipe)(
    OMX_OUT OMX_HANDLETYPE *hPipe,
    OMX_IN OMX_STRING szURI);

typedef OMX_BOOL(*tpOMXConfigParser)(
    OMX_PTR aInputParameters,
    OMX_PTR aOutputParameters);
```

上述函数指针是 OpenMax 的核心方法，这些指针类型需要使用继承来设置。

另外，在文件 omx_interface.h 中还定义了类 OMXInterface，在类中包含了一系列函数，这些函数返回的都是上面类型的函数指针。类 OMXInterface 是 OpenMax 直接实现 OpenCore 的接口。

```
class OMXInterface : public OsclSharedLibraryInterface
{
    public:
        OMXInterface()
        {
            pOMX_Init = NULL;
            pOMX_Deinit = NULL;
            pOMX_ComponentNameEnum = NULL;
            pOMX_GetHandle = NULL;
            pOMX_FreeHandle = NULL;
            pOMX_GetComponentsOfRole = NULL;
            pOMX_GetRolesOfComponent = NULL;
            pOMX_SetupTunnel = NULL;
            pOMX_GetContentPipe = NULL;
            pOMXConfigParser = NULL;
        };
        virtual bool UnloadWhenNotUsed(void) = 0;
        tpOMX_Init GetpOMX_Init()
        {
            return pOMX_Init;
        };
        tpOMX_Deinit GetpOMX_Deinit()
        {
            return  pOMX_Deinit;
        };
        tpOMX_ComponentNameEnum GetpOMX_ComponentNameEnum()
        {
            return pOMX_ComponentNameEnum;
        };
        tpOMX_GetHandle GetpOMX_GetHandle()
        {
            return pOMX_GetHandle;
        };
        tpOMX_FreeHandle GetpOMX_FreeHandle()
        {
            return pOMX_FreeHandle;
        };
        tpOMX_GetComponentsOfRole GetpOMX_GetComponentsOfRole()
        {
            return pOMX_GetComponentsOfRole;
        };
        tpOMX_GetRolesOfComponent GetpOMX_GetRolesOfComponent()
        {
            return pOMX_GetRolesOfComponent;
        };
        tpOMX_SetupTunnel GetpOMX_SetupTunnel()
        {
            return pOMX_SetupTunnel;
```

```
        };
        tpOMX_GetContentPipe GetpOMX_GetContentPipe()
        {
            return pOMX_GetContentPipe;
        };
        tpOMXConfigParser GetpOMXConfigParser()
        {
            return pOMXConfigParser;
        };
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_Init)(void);
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_Deinit)(void);
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_ComponentNameEnum)(
            OMX_OUT OMX_STRING cComponentName,
            OMX_IN  OMX_U32 nNameLength,
            OMX_IN  OMX_U32 nIndex);
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_GetHandle)(
            OMX_OUT OMX_HANDLETYPE* pHandle,
            OMX_IN  OMX_STRING cComponentName,
            OMX_IN  OMX_PTR pAppData,
            OMX_IN  OMX_CALLBACKTYPE* pCallBacks);
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_FreeHandle)(
            OMX_IN  OMX_HANDLETYPE hComponent);
        OMX_ERRORTYPE(*pOMX_GetComponentsOfRole)(
            OMX_IN      OMX_STRING role,
            OMX_INOUT   OMX_U32 *pNumComps,
            OMX_INOUT   OMX_U8  **compNames);
        OMX_ERRORTYPE(*pOMX_GetRolesOfComponent)(
            OMX_IN      OMX_STRING compName,
            OMX_INOUT   OMX_U32 *pNumRoles,
            OMX_OUT     OMX_U8 **roles);
        OMX_ERRORTYPE OMX_APIENTRY(*pOMX_SetupTunnel)(
            OMX_IN  OMX_HANDLETYPE hOutput,
            OMX_IN  OMX_U32 nPortOutput,
            OMX_IN  OMX_HANDLETYPE hInput,
            OMX_IN  OMX_U32 nPortInput);
        OMX_ERRORTYPE(*pOMX_GetContentPipe)(
            OMX_OUT OMX_HANDLETYPE *hPipe,
            OMX_IN OMX_STRING szURI);
        OMX_BOOL(*pOMXConfigParser)(
            OMX_PTR aInputParameters,
            OMX_PTR aOutputParameters);
};
```

3. OpenMax 组织结构

在文件“opencore/codecs_v2/omx/omx_sharedlibrary/interface/src/pv_omx_interface.cpp”中，通过实现类里面的函数指针方式实现类 OMXInterface。

在文件 pv_omx_interface.cpp 中，函数 PVGetInterface()和 PVReleaseInterface()是使用 C 语言导出的函数，这两个函数的实现代码如下所示。

```
extern "C"
{
    OSCL_EXPORT_REF OsclAny* PVGetInterface()
    {
        return PVOMXInterface::Instance();
    }
    OSCL_EXPORT_REF void PVReleaseInterface(void* interface)
    {
        PVOMXInterface* pInterface = (PVOMXInterface*)interface;
        if (pInterface)
        {
            OSCL_DELETE(pInterface);
        }
    }
}
```

在文件 pv_omx_interface.cpp 中，类 PVOMXInterface 继承了 OMXInterface，在此类的构造函数中设置了各个 OMXInterface 中的函数指针。构造函数 PVOMXInterface()的主要代码如下所示。

```
private:
    PVOMXInterface()
    {
        //设置指针 OMX 的核心方法
        pOMX_Init = OMX_Init;
        pOMX_Deinit = OMX_Deinit;
        pOMX_ComponentNameEnum = OMX_ComponentNameEnum;
        pOMX_GetHandle = OMX_GetHandle;
        pOMX_FreeHandle = OMX_FreeHandle;
        pOMX_GetComponentsOfRole = OMX_GetComponentsOfRole;
        pOMX_GetRolesOfComponent = OMX_GetRolesOfComponent;
        pOMX_SetupTunnel = OMX_SetupTunnel;
        pOMX_GetContentPipe = OMX_GetContentPipe;
        pOMXConfigParser = OMXConfigParser;
    };
```

我们介绍的上述构造函数，都是在文件“opencore/codecs_v2/omx/omx_common/src/pv_omxcore.cpp”中实现的，此文件实现了 OpenMax 的核心功能。

文件“opencore/codecs_v2/omx/omx_common/src/pv_omxregistry.cpp”的功能是注册 OpenMax 模块，其主要实现代码如下所示。

```
//注册 MP3 解码器
OMX_ERRORTYPE Mp3Register()
{
   ComponentRegistrationType *pCRT = (ComponentRegistrationType *) oscl_malloc(sizeof
(ComponentRegistrationType));
   if (pCRT)
   {
      pCRT->ComponentName = (OMX_STRING)"OMX.PV.mp3dec";//组件名
      pCRT->RoleString[0] = (OMX_STRING)"audio_decoder.mp3";
      pCRT->NumberOfRolesSupported = 1;
      pCRT->SharedLibraryOsclUuid = NULL;
#if USE_DYNAMIC_LOAD_OMX_COMPONENTS
      pCRT->FunctionPtrCreateComponent = &OmxComponentFactoryDynamicCreate;
      pCRT->FunctionPtrDestroyComponent = &OmxComponentFactoryDynamicDestructor;
      pCRT->SharedLibraryName = (OMX_STRING)"libomx_mp3dec_sharedlibrary.so";
      pCRT->SharedLibraryPtr = NULL;
      OsclUuid *temp = (OsclUuid *) oscl_malloc(sizeof(OsclUuid));
      if (temp == NULL)
      {
         oscl_free(pCRT); //释放内存
         return OMX_ErrorInsufficientResources;
      }
      OSCL_PLACEMENT_NEW(temp, PV_OMX_MP3DEC_UUID);
      pCRT->SharedLibraryOsclUuid = (OMX_PTR) temp;
      pCRT->SharedLibraryRefCounter = 0;
#endif
#if REGISTER_OMX_MP3_COMPONENT
#if (DYNAMIC_LOAD_OMX_MP3_COMPONENT == 0)
      pCRT->FunctionPtrCreateComponent = &Mp3OmxComponentFactory;
      pCRT->FunctionPtrDestroyComponent = &Mp3OmxComponentDestructor;
      pCRT->SharedLibraryName = NULL;
      pCRT->SharedLibraryPtr = NULL;
      if (pCRT->SharedLibraryOsclUuid)
         oscl_free(pCRT->SharedLibraryOsclUuid);
      pCRT->SharedLibraryOsclUuid = NULL;
      pCRT->SharedLibraryRefCounter = 0;
#endif
#endif
   }
   else
   {
      return OMX_ErrorInsufficientResources;
   }
```

```
    return  ComponentRegister(pCRT);
}
//WMA 格式解码
OMX_ERRORTYPE WmaRegister()
{
    ComponentRegistrationType *pCRT = (ComponentRegistrationType *) oscl_malloc(sizeof
(ComponentRegistrationType));
    if (pCRT)
    {
        pCRT->ComponentName = (OMX_STRING)"OMX.PV.wmadec";
        pCRT->RoleString[0] = (OMX_STRING)"audio_decoder.wma";
        pCRT->NumberOfRolesSupported = 1;
        pCRT->SharedLibraryOsclUuid = NULL;
#if USE_DYNAMIC_LOAD_OMX_COMPONENTS
        pCRT->FunctionPtrCreateComponent = &OmxComponentFactoryDynamicCreate;
        pCRT->FunctionPtrDestroyComponent = &OmxComponentFactoryDynamicDestructor;
        pCRT->SharedLibraryName = (OMX_STRING)"libomx_wmadec_sharedlibrary.so";
        pCRT->SharedLibraryPtr = NULL;
        OsclUuid *temp = (OsclUuid *) oscl_malloc(sizeof(OsclUuid));
        if (temp == NULL)
        {
            oscl_free(pCRT); // free allocated memory
            return OMX_ErrorInsufficientResources;
        }
        OSCL_PLACEMENT_NEW(temp, PV_OMX_WMADEC_UUID);
        pCRT->SharedLibraryOsclUuid = (OMX_PTR) temp;
        pCRT->SharedLibraryRefCounter = 0;
#endif
#if REGISTER_OMX_WMA_COMPONENT
#if (DYNAMIC_LOAD_OMX_WMA_COMPONENT == 0)

        pCRT->FunctionPtrCreateComponent = &WmaOmxComponentFactory;
        pCRT->FunctionPtrDestroyComponent = &WmaOmxComponentDestructor;
        pCRT->SharedLibraryName = NULL;
        pCRT->SharedLibraryPtr = NULL;
        if (pCRT->SharedLibraryOsclUuid)
            oscl_free(pCRT->SharedLibraryOsclUuid);
        pCRT->SharedLibraryOsclUuid = NULL;
        pCRT->SharedLibraryRefCounter = 0;
#endif
#endif
    }
    else
    {
        return OMX_ErrorInsufficientResources;
    }
    return  ComponentRegister(pCRT);
}
```

4. 实现 OpenMax 编码/解码组件

OpenMax 的主要功能是通过解码/编码组件实现的。各个组件的基本结构类似，它们的实现内容实际上就是文件“opencore/codecs_v2/omx/omx_baseclass/include/pv_omxcomponent.h”中定义的类 OmxComponentBase。假如要实现 MP3 格式文件的解码处理，则在如下目录中实现了 MP3 的解码功能。

```
opencore/codecs_v2/omx/mp3
```

在上述目录中，文件 Android.mk 生成了名为 libomx_mp3_component_lib.so 的库，此静态库将被连接并生成动态库 libomx_mp3dec_sharedlibrary_lib。此 Android.mk 文件的主要代码如下所示。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES := \
    src/mp3_dec.cpp \
    src/omx_mp3_component.cpp \
    src/mp3_timestamp.cpp
```

```
LOCAL_MODULE := libomx_mp3_component_lib
LOCAL_CFLAGS :=  $(PV_CFLAGS)
LOCAL_ARM_MODE := arm
LOCAL_STATIC_LIBRARIES :=
LOCAL_SHARED_LIBRARIES :=
LOCAL_C_INCLUDES := \
   $(PV_TOP)/codecs_v2/omx/omx_mp3/src \
   $(PV_TOP)/codecs_v2/omx/omx_mp3/include \
   $(PV_TOP)/extern_libs_v2/khronos/openmax/include \
   $(PV_TOP)/codecs_v2/omx/omx_baseclass/include \
   $(PV_TOP)/codecs_v2/audio/mp3/dec/src \
   $(PV_TOP)/codecs_v2/audio/mp3/dec/include \
   $(PV_INCLUDES)
LOCAL_COPY_HEADERS_TO := $(PV_COPY_HEADERS_TO)
LOCAL_COPY_HEADERS := \
   include/mp3_dec.h \
   include/omx_mp3_component.h \
   include/mp3_timestamp.h
include $(BUILD_STATIC_LIBRARY)
```

在目录“opencore/codecs_v2/omx/omx_mp3/src/”中存在如下 3 个文件。

（1）mp3_dec.cpp：能够调用 MP3 解码器组件；

（2）mp3_timestamp.cpp：能够实现时间戳功能；

（3）omx_mp3_component.cpp：定义了 MP3 解码器组件；

在文件“opencore/codecs_v2/omx/omx_mp3/include/omx_mp3_component.h”中定义了类 OpenmaxMp3AO，此类继承了 OmxComponentAudio，主要代码如下所示。

```
class OpenmaxMp3AO : public OmxComponentAudio {
   public:
      OpenmaxMp3AO();
      ~OpenmaxMp3AO();
      OMX_ERRORTYPE ConstructComponent(OMX_PTR pAppData, OMX_PTR pProxy);
      OMX_ERRORTYPE DestroyComponent();
      OMX_ERRORTYPE ComponentInit();
      OMX_ERRORTYPE ComponentDeInit();
      static void ComponentGetRolesOfComponent(OMX_STRING* aRoleString);
      void ProcessData();
      void SyncWithInputTimestamp();
      void ProcessInBufferFlag();
      void ResetComponent();
      OMX_ERRORTYPE GetConfig(
         OMX_IN  OMX_HANDLETYPE hComponent,
         OMX_IN  OMX_INDEXTYPE nIndex,
         OMX_INOUT OMX_PTR pComponentConfigStructure);
   private:
      void CheckForSilenceInsertion();
      void DoSilenceInsertion();
      Mp3Decoder*     ipMp3Dec;
      Mp3TimeStampCalc iCurrentFrameTS;
};
```

在文件 omx_mp3_component.cpp 中定义了 MP3 解码器组件，通过函数 ProcessData()实现 MP3 文件的解码处理。函数 ProcessData()的实现代码如下所示。

```
void OpenmaxMp3AO::ProcessData()
{
   PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0, "OpenmaxMp3AO : ProcessData IN"));

   QueueType* pInputQueue  = ipPorts[OMX_PORT_INPUTPORT_INDEX]->pBufferQueue;
   QueueType* pOutputQueue = ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->pBufferQueue;

   ComponentPortType* pInPort = (ComponentPortType*) ipPorts[OMX_PORT_INPUTPORT_INDEX];
   ComponentPortType* pOutPort = ipPorts[OMX_PORT_OUTPUTPORT_INDEX];
   OMX_COMPONENTTYPE* pHandle = &iOmxComponent;
```

```
    OMX_U8* pOutBuffer;//输出缓冲区的指针
    OMX_U32 OutputLength; //输出缓冲区的长度
    OMX_S32 DecodeReturn;
    OMX_BOOL ResizeNeeded = OMX_FALSE;

    OMX_U32 TempInputBufferSize = (2 * sizeof(uint8) * (ipPorts[OMX_PORT_INPUTPORT_
INDEX]->PortParam.nBufferSize));

    if ((!iIsInputBufferEnded) || iEndofStream)
    {
        if (OMX_TRUE == iSilenceInsertionInProgress)
        {
            DoSilenceInsertion();
            //If the flag is still true, come back to this routine again
            if (OMX_TRUE == iSilenceInsertionInProgress)
            {
                return;
            }
        }
        //证实 prev 是否发布了 buffer
        if (OMX_TRUE == iNewOutBufRequired)
        {
            //证实一个新的输出缓冲区是否为可利用的
            if (0 == (GetQueueNumElem(pOutputQueue)))
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
                "OpenmaxMp3AO : ProcessData OUT output buffer unavailable"));
                return;
            }
            ipOutputBuffer = (OMX_BUFFERHEADERTYPE*) DeQueue(pOutputQueue);
            if (NULL == ipOutputBuffer)
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG,  iLogger,  PVLOGMSG_NOTICE,  (0,
"OpenmaxMp3AO : ProcessData Error, Output Buffer Dequeue returned NULL, OUT"));
                return;
            }
            ipOutputBuffer->nFilledLen = 0;
            iNewOutBufRequired = OMX_FALSE;
            //设置当前时间戳对输出缓冲区时间戳
            ipOutputBuffer->nTimeStamp = iCurrentFrameTS.GetConvertedTs();
            //复制在动态港重组之前当地被存放的输出缓冲区
            //被接受的新的 OMX 缓冲
            if (OMX_TRUE == iSendOutBufferAfterPortReconfigFlag)
            {
                if ((ipTempOutBufferForPortReconfig)
                        && (iSizeOutBufferForPortReconfig <= ipOutputBuffer->nAllocLen))
                {
                    oscl_memcpy(ipOutputBuffer->pBuffer, ipTempOutBufferForPortReconfig,
                    iSizeOutBufferForPortReconfig);
                    ipOutputBuffer->nFilledLen = iSizeOutBufferForPortReconfig;
                    ipOutputBuffer->nTimeStamp = iTimestampOutBufferForPortReconfig;
                }
                iSendOutBufferAfterPortReconfigFlag = OMX_FALSE;
                //当充满时退还输出缓冲区
                if ((ipOutputBuffer->nAllocLen - ipOutputBuffer->nFilledLen) <
                iOutputFrameLength)
                {
                    ReturnOutputBuffer(ipOutputBuffer, pOutPort);
                }
                //释放临时输出缓冲区
                if (ipTempOutBufferForPortReconfig)
                {
                    oscl_free(ipTempOutBufferForPortReconfig);
                    ipTempOutBufferForPortReconfig = NULL;
                    iSizeOutBufferForPortReconfig = 0;
                }
                //Dequeue new output buffer if required to continue decoding the next frame
                if (OMX_TRUE == iNewOutBufRequired)
                {
                    if (0 == (GetQueueNumElem(pOutputQueue)))
                    {
```

```
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
"OpenmaxMp3AO : ProcessData OUT, output buffer unavailable"));
                return;
            }
            ipOutputBuffer = (OMX_BUFFERHEADERTYPE*) DeQueue(pOutputQueue);
            if (NULL == ipOutputBuffer)
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
"OpenmaxMp3AO : ProcessData Error, Output Buffer Dequeue returned NULL, OUT"));
                return;
            }
            ipOutputBuffer->nFilledLen = 0;
            iNewOutBufRequired = OMX_FALSE;
            ipOutputBuffer->nTimeStamp = iCurrentFrameTS.GetConvertedTs();
        }
    }
}
/*标号缓冲的代码
 * 根据 hMarkTargetComponent 设置规格
 */
if (ipMark != NULL)
{
    ipOutputBuffer->hMarkTargetComponent = ipMark->hMarkTargetComponent;
    ipOutputBuffer->pMarkData = ipMark->pMarkData;
    ipMark = NULL;
}
if (ipTargetComponent != NULL)
{
    ipOutputBuffer->hMarkTargetComponent = ipTargetComponent;
    ipOutputBuffer->pMarkData = iTargetMarkData;
    ipTargetComponent = NULL;
}
//在此标记缓冲代码末端
pOutBuffer = &ipOutputBuffer->pBuffer[ipOutputBuffer->nFilledLen];
OutputLength = 0;
/*复制临时被存放的前个输入缓冲区的残余数据
 *缓冲接踵而来的数据流
 */
if (iTempInputBufferLength > 0 &&
        ((iInputCurrLength + iTempInputBufferLength) < TempInputBufferSize))
{
    oscl_memcpy(&ipTempInputBuffer[iTempInputBufferLength], ipFrameDecodeBuffer,
    iInputCurrLength);
    iInputCurrLength += iTempInputBufferLength;
    iTempInputBufferLength = 0;
    ipFrameDecodeBuffer = ipTempInputBuffer;
}
//将输出缓冲区作为指针
DecodeReturn = ipMp3Dec->Mp3DecodeAudio(//设置 ipMp3Dec 的类型是 Mp3Decode
(OMX_S16*) pOutBuffer,//输出缓冲区的指针
 (OMX_U32*) & OutputLength,//输出缓冲区的长度
                                        &(ipFrameDecodeBuffer),
                                        &iInputCurrLength,
                                        &iFrameCount,

&(ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->AudioPcmMode),

&(ipPorts[OMX_PORT_INPUTPORT_INDEX]->AudioMp3Param),
                                        iEndOfFrameFlag,
                                        &ResizeNeeded);
if (ResizeNeeded == OMX_TRUE)
{
    if (0 != OutputLength)
    {
        iOutputFrameLength = OutputLength * 2;
        //更新时间戳
        iSamplesPerFrame = OutputLength / ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->
        AudioPcmMode.nChannels;
        iCurrentFrameTS.SetParameters(ipPorts[OMX_PORT_OUTPUTPORT_INDEX]->
        AudioPcmMode.nSamplingRate, iSamplesPerFrame);
        iOutputMilliSecPerFrame = iCurrentFrameTS.GetFrameDuration();
```

```
        }
        iResizePending = OMX_TRUE;
        /*不要退回引起的输出缓冲区，在当地存放它
         *并且等待动态接口重新构造完成*/
        if ((NULL == ipTempOutBufferForPortReconfig))
        {
            ipTempOutBufferForPortReconfig = (OMX_U8*) oscl_malloc(sizeof(uint8) *
            OutputLength * 2);
            if (NULL == ipTempOutBufferForPortReconfig)
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
                "OpenmaxMp3AO : ProcessData error, insufficient resources"));
                return;
            }
        }
        //复制 OMX 输出缓冲区对临时内部缓冲
        oscl_memcpy(ipTempOutBufferForPortReconfig, pOutBuffer, OutputLength * 2);
        iSizeOutBufferForPortReconfig = OutputLength * 2;
        //设置当前时间戳对第一个产品框架的输出缓冲区时间戳
        //以后将取消
        iTimestampOutBufferForPortReconfig = iCurrentFrameTS.GetConvertedTs();
        iCurrentFrameTS.UpdateTimestamp(iSamplesPerFrame);
           OutputLength = 0;
        OMX_COMPONENTTYPE* pHandle = (OMX_COMPONENTTYPE*) ipAppPriv->CompHandle;
        (*(ipCallbacks->EventHandler))
        (pHandle,
         iCallbackData,
         OMX_EventPortSettingsChanged, //The command was completed
         OMX_PORT_OUTPUTPORT_INDEX,
         0,
         NULL);
    }
    ipOutputBuffer->nFilledLen += OutputLength * 2;
    ipOutputBuffer->nOffset = 0;
    if (OutputLength > 0)
    {
        iCurrentFrameTS.UpdateTimestamp(iSamplesPerFrame);
    }
    if (OMX_TRUE == iEndofStream)
    {
     if (MP3DEC_SUCCESS != DecodeReturn)
        {
            PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
            "OpenmaxMp3AO : ProcessData EOS callback send"));
            (*(ipCallbacks->EventHandler))
            (pHandle,
             iCallbackData,
             OMX_EventBufferFlag,
             1,
             OMX_BUFFERFLAG_EOS,
             NULL);
            iEndofStream = OMX_FALSE;
            ipOutputBuffer->nFlags |= OMX_BUFFERFLAG_EOS;
            ReturnOutputBuffer(ipOutputBuffer, pOutPort);
            ipOutputBuffer = NULL;
            PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
            "OpenmaxMp3AO : ProcessData OUT"));
            return;
        }
    }
    if (MP3DEC_SUCCESS == DecodeReturn)
    {
        ipInputBuffer->nFilledLen = iInputCurrLength;
    }
    else if (MP3DEC_INCOMPLETE_FRAME == DecodeReturn)
    {
        oscl_memcpy(ipTempInputBuffer, ipFrameDecodeBuffer, iInputCurrLength);
        iTempInputBufferLength = iInputCurrLength;
        ipInputBuffer->nFilledLen = 0;
        iInputCurrLength = 0;
    }
```

```
        else
        {
            ipInputBuffer->nFilledLen = 0;
            iInputCurrLength = 0;
            PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0,
            "OpenmaxMp3AO : ProcessData ErrorStreamCorrupt callback send"));
            (*(ipCallbacks->EventHandler))
            (pHandle,
             iCallbackData,
             OMX_EventError,
             OMX_ErrorStreamCorrupt,
             0,
             NULL);
        }
        //如果它经过译码器处理，则会得到充分地消耗，并退回到输入缓冲区
        if (0 == ipInputBuffer->nFilledLen)
        {
            ReturnInputBuffer(ipInputBuffer, pInPort);
            ipInputBuffer = NULL;
            iIsInputBufferEnded = OMX_TRUE;
            iInputCurrLength = 0;
        }
        //当充满时送回输出缓冲区
        if ((ipOutputBuffer->nAllocLen - ipOutputBuffer->nFilledLen) <
        (iOutputFrameLength))
        {
            ReturnOutputBuffer(ipOutputBuffer, pOutPort);
            ipOutputBuffer = NULL;
        }
        /*如果有些处理在当前缓冲中，则重新编排 AO*/
        if (((iInputCurrLength != 0 || GetQueueNumElem(pInputQueue) > 0)
                && (GetQueueNumElem(pOutputQueue) > 0) && (ResizeNeeded == OMX_FALSE))
                || (OMX_TRUE == iEndofStream))
        {
            RunIfNotReady();
        }
    }
    PVLOGGER_LOGMSG(PVLOGMSG_INST_HLDBG, iLogger, PVLOGMSG_NOTICE, (0, "OpenmaxMp3AO :
ProcessData OUT"));
    return;
}
```

4.3.6 OpenCore 扩展详解

在 Android 系统中，除了可以使用 OpenCore 本身提供的强大功能外，还可以对 OpenCore 进行扩展以实现更加强大的功能。

1. OpenCore Node

在扩展 OpenCore 时，一般基于 OpenCore 的框架为其增加固定的插件，插件主要做成 Node 的形式。其中和编解码相关的 Node 如下所示。

- pvomxbasedecnode;
- pvomxaudiodecnode;
- pvomxvideod encode;
- pvomxencnode。

在扩展 OpenCore 时，和文件格式相关的 Node 如下所示。

- pvwavffparsernode;
- Pvaacffparsernode;
- Pvamrffparsernode;
- pvmp3ffparsern ode;

- pvmp4ffparsernode；
- pvvideoparsernode；
- pvmp4f fcomposernode。

在扩展 OpenCore 时，和输入、输出相关的 Node 如下所示。

- pvmediainputnode；
- pvmediaoutputnode；
- pvdummyinput node；
- pvdummyoutputnode；
- pvfileoutputnode；
- pvdownloadmanagernode。

除了上述 Node 之外，还包括一些其他功能的常用 Node，例如 pvsocketnode 和 pvdownloadmanagernode 等。

2. MediaIO

MediaIO 的缩写是 MIO，在“opencore/pvmi/pvmf/include/”目录中由如下头文件定义。

- pvmiMIOControl.h；
- pvmi_media_transfer.h。

在实现的过程中只需要继承和构建其中的接口，然后由框架最终实现成为 Node，并在 OpenCore 系统中使用。其实 MediaIO 是对 Node 的一种封装，将其封装成多媒体的输入、输出环节。

在文件 pvmi_mio_control.h 中，定义类 PvmiMIOControl 来表示 MIO 的控制类接口。定义此类的代码如下所示。

```
class PvmiMIOControl{
   public:
      virtual ~PvmiMIOControl() {}
      virtual PVMFStatus connect(PvmiMIOSession& aSession,
                         PvmiMIOObserver* aObserver) = 0;
      virtual PVMFStatus disconnect(PvmiMIOSession aSession) = 0;
      virtual PvmiMediaTransfer* createMediaTransfer(
         PvmiMIOSession& aSession,
         PvmiKvp* read_formats = NULL, int32 read_flags = 0,
         PvmiKvp* write_formats = NULL, int32 write_flags = 0) = 0;
      virtual void deleteMediaTransfer(
         PvmiMIOSession& aSession,
         PvmiMediaTransfer* media_transfer) = 0;
      virtual PVMFCommandId QueryUUID(const PvmfMimeString& aMimeType,
                            Oscl_Vector<PVUuid, OsclMemAllocator>& aUuids,
                            bool aExactUuidsOnly = false,
                            const OsclAny* aContext = NULL) = 0;
      virtual PVMFCommandId QueryInterface(const PVUuid& aUuid,
                               PVInterface*& aInterfacePtr,
                               const OsclAny* aContext = NULL) = 0;
      virtual PVMFCommandId Init(const OsclAny* aContext = NULL) = 0;
      virtual PVMFCommandId Reset(const OsclAny* aContext = NULL) = 0;
      virtual PVMFCommandId Start(const OsclAny* aContext = NULL) = 0;
      virtual PVMFCommandId Pause(const OsclAny* aContext = NULL) = 0;
      virtual PVMFCommandId Flush(const OsclAny* aContext = NULL) = 0;
      virtual PVMFCommandId DiscardData(const OsclAny* aContext = NULL) = 0;
      virtual  PVMFCommandId  DiscardData(PVMFTimestamp  aTimestamp,  const  OsclAny*
aContext = NULL) = 0;
      virtual PVMFCommandId Stop(const OsclAny* aContext = NULL) = 0;
      virtual PVMFCommandId CancelCommand(PVMFCommandId aCmd, const OsclAny* aContext
= NULL) = 0;
      virtual PVMFCommandId CancelAllCommands(const OsclAny* aContext = NULL) = 0;
      virtual void ThreadLogon() = 0;
```

```
        virtual void ThreadLogoff() = 0;
};
```

在上述代码中，有很多函数使用 OsclAny 类型的指针作为参数，这样的好处是可以使用所有数据结构。其中接口 Init()、Reset()、Start()、Pause()、Flush()和 Stop()实现流控制，而函数 createMediaTransfer 用于得到类 PvmiMediaTransfer。

而在文件 pvmi_media_transfer.h 中，定义类 PvmiMediaTransfer 来表示 MIO 的数据接口。定义此类的代码如下所示。

```
class PvmiMediaTransfer
{
    public:
        virtual ~PvmiMediaTransfer() {}
        virtual void setPeer(PvmiMediaTransfer* aPeer) = 0;
        virtual void useMemoryAllocators(OsclMemAllocator* read_write_alloc) = 0;
        virtual PVMFCommandId writeAsync(
            uint8 format_type, int32 format_index,
            uint8* data, uint32 data_len,
            const PvmiMediaXferHeader& data_header_info,
            OsclAny* aContext = NULL) = 0;
        virtual void writeComplete(
            PVMFStatus aStatus,
            PVMFCommandId write_cmd_id,
            OsclAny* aContext) = 0;
        virtual PVMFCommandId readAsync(
            uint8* data, uint32 max_data_len,
            OsclAny* aContext = NULL,
            int32* formats = NULL, uint16 num_formats = 0) = 0;
        virtual void readComplete(
            PVMFStatus aStatus,
            PVMFCommandId read_cmd_id,
            int32 format_index,
            const PvmiMediaXferHeader& data_header_info,
            OsclAny* aContext) = 0;
        virtual void statusUpdate(uint32 status_flags) = 0;
        virtual void cancelCommand(PVMFCommandId command_id) = 0;
        virtual void cancelAllCommands() = 0;
};
```

3. OpenCore Player

OpenCore Player 的编译文件是“pvplayer/Android.mk”，编译后将生成动态库文件 libopencoreplayer.so，在此库中包含了如下两方面的内容。

（1）Player 的 Engine（引擎）；

（2）为 Android 构建的 Player 是一个适配器（Adapter），Engine 的路径是“engine/player”，Adapter 的路径是“android”。

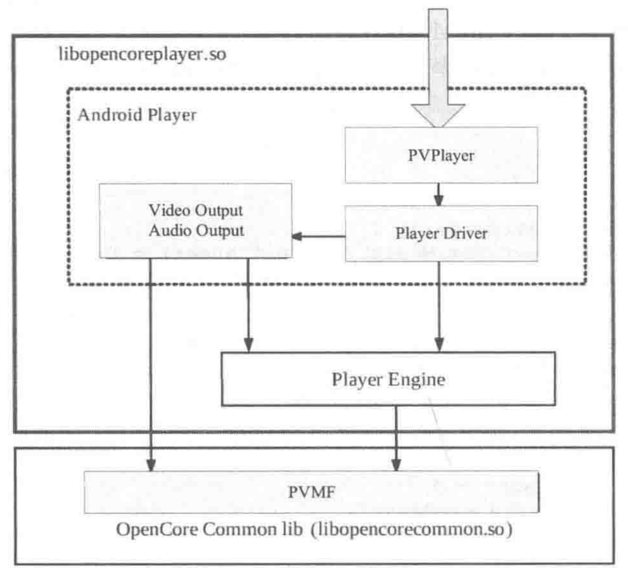
在库 libopencoreplayer.so 中包含了下面的内容。

- 解码工具；
- 文件的解析器；
- 解码工具对应的 Node；
- Player 的引擎部分，编译文件是“engines/player/Android.mk”；
- 为 Android 构建的 Player 适配器，编译文件是“android/Android.mk”；
- 识别工具，目录是“pvmi/recognizer”；
- 编解码工具中的 OpenMAX 部分，目录是“codecs_v2/omx”；
- 对应插件 Node 的注册。

由此可见，库 libopencoreplayer.so 中的内容较多，其中主要功能是作为各个文件的解析器和

解码器。PVPlayer 的核心功能在文件“engines/player/Android.mk”中。而文件“android/Android.mk”的内容比较特殊，其功能是在 PVPlayer 之上构建的一个可以供 Android 使用的播放器。

库 libopencoreplayer.so 的具体结构如图 4-10 所示。

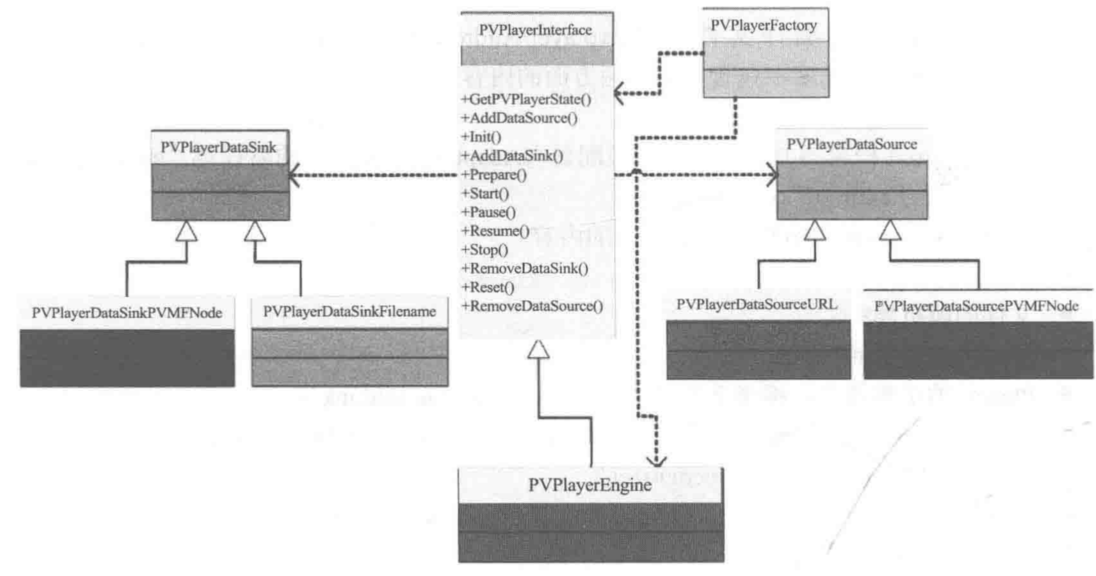


▲图 4-10 库 libopencoreplayer.so 的结构

在接下来的内容中，将详细讲解图 4-10 中列出的各个构成部分。

（1）Player Engine。

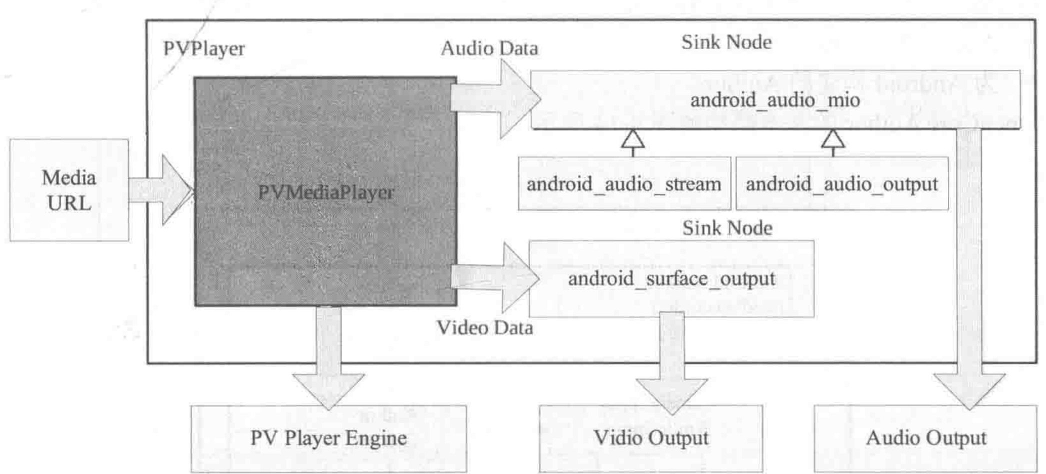
OpenCore 中的 Player Engine 具有清晰、明确的接口，在此接口中不同的系统可以根据具体情况实现不同的 Player。Player Engine 位于 OpenCore 中的“engines/player/”目录下，其中在“engines/player/include”目录中保存的是接口头文件，在“engines/player/src”目录中保存的是源文件和私有头文件。Player Engine 的类结构如图 4-11 所示。



▲图 4-11 Player Engine 的类结构

（2）PVPlayer。

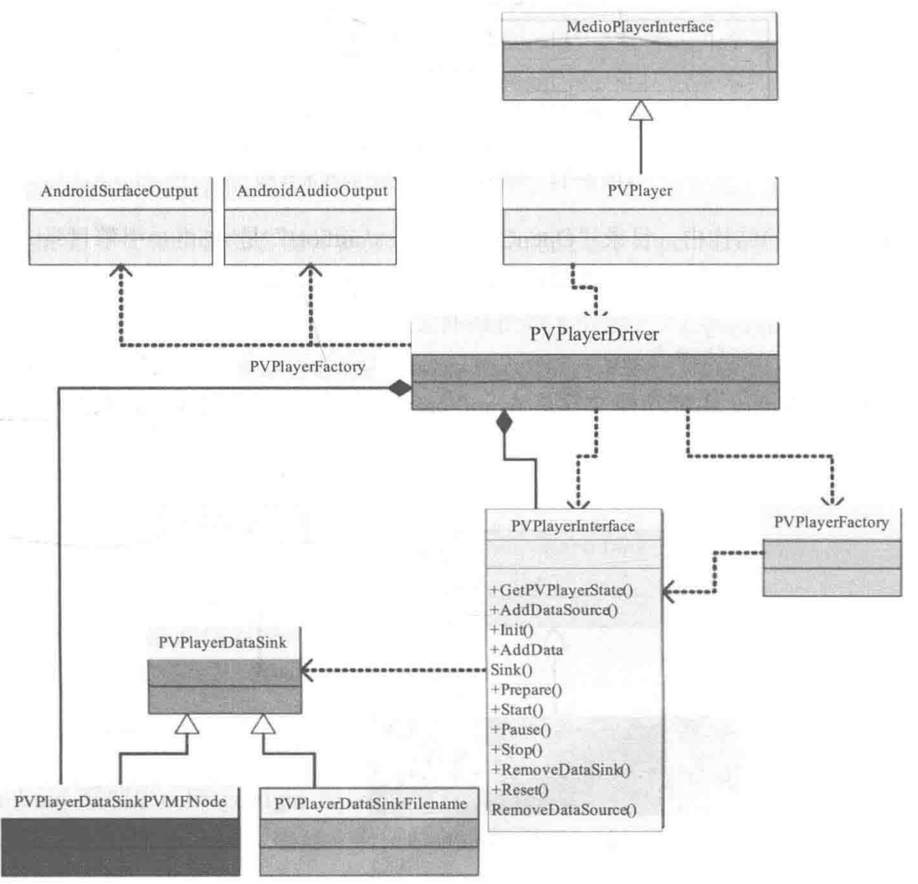
PVPlayer 的结构如图 4-12 所示。



▲图 4-12　PVPlayer 的结构

在图 4-12 中，Sink Node 会接受上一个 Node 写的动作。

VPlayer 的类结构如图 4-13 所示。



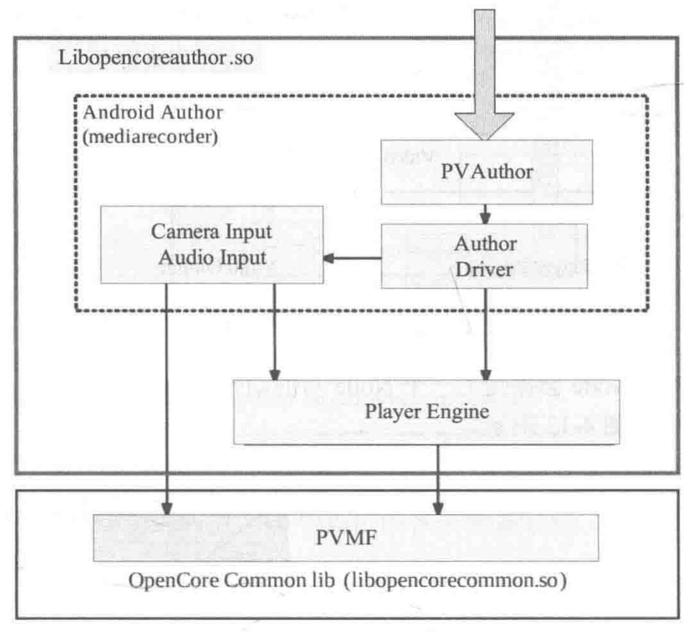
▲图 4-13　VPlayer 的类结构

（3）Author。

OpenCore 的 Author 的编译文件是“pvauthor/Android.mk”，编译后将生成动态库文件 libopencoreauthor.so，这个库和 Player 类似，在里面包含了如下两方面的内容。

- Author 的 Engine;
- 为 Android 构建的 Author。

OpenCore Author 的基本结构如图 4-14 所示。

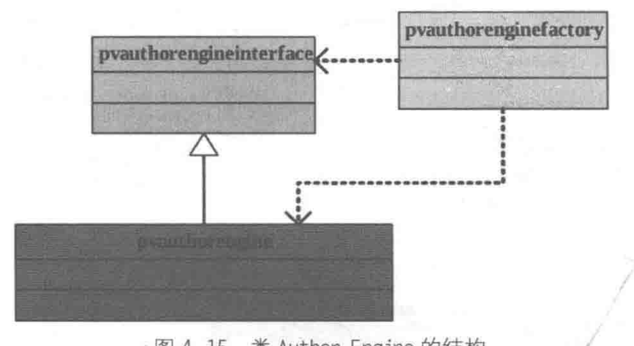


▲图 4-14　OpenCore Author 的基本结构

在图 4-14 所示的结构中，目录“OpenCore/engines/author/”是 Author 引擎目录，在里面主要包含了“include”和“src”两个子目录，其中如下两个头文件是接口。

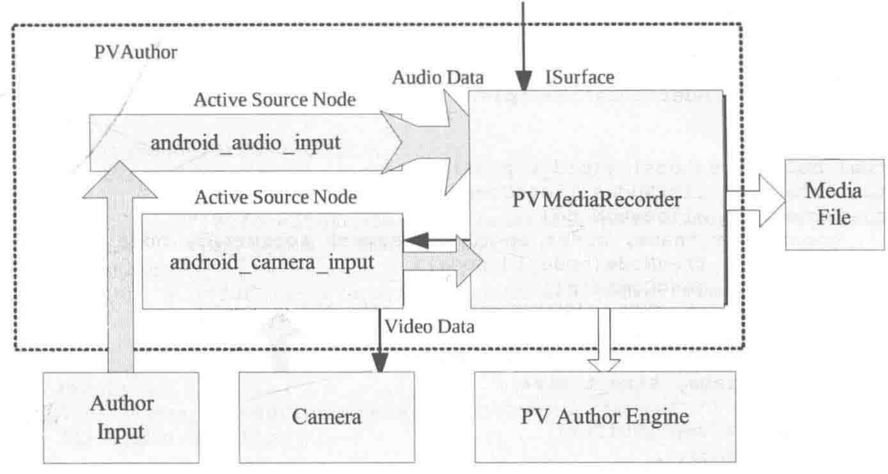
```
pvauthorenginefactory.h
pvauthorengineinterface.h
```

OpenCore 的 Author 主要功能文件是 pvauthorengine.cpp，类 Author Engine 的结构如图 4-15 所示。



▲图 4-15　类 Author Engine 的结构

PVAuthor 的结构如图 4-16 所示。



▲图 4-16 PVAuthor 的结构

4.4 Stagefright 框架详解

在 Android 系统中，预设的多媒体框架（Multimedia Framework）是 OpenCore。OpenCore 的特点是兼顾了跨平台的移植性，而且已经过多方验证，所以相对来说比较稳定；但是其缺点是庞大、复杂，需要耗费相当多的时间去维护。从 Android 2.0 开始，Google 引入了架构稍微简单的 Stagefright，并且有逐渐取代 OpenCore 的趋势。从 Android 2.2 开始，几乎完全放弃了 OpenCore，而主推 Stagefright。在本节的内容中，将详细讲解 Stagefright 框架的基本知识，为读者进入本书后面知识的学习打下基础。

4.4.1 Stagefright 代码结构

Stagefright 是一个轻量级的多媒体框架，其主要功能是基于 OpenMax 实现的。在 Stagefright 中提供了媒体播放等接口，这些接口可以为 Android 框架层所使用。

在 Android 开源代码中，Stagefright 的头文件路径如下所示。

```
frameworks/av/include/media/stagefright/
```

实现 Stagefright 功能的文件路径如下所示。

```
frameworks/av/media/libstagefright/
```

实现 Stagefright 播放器和录音器功能的文件路径如下所示。

```
frameworks/av/media/libmediaplayerservice/
```

测试 Stagefrigh 功能的代码路径如下所示。

```
frameworks/av/cmds/stagefright/
```

4.4.2 Stagefright 实现 OpenMax 接口

在 Android 系统中，Stagefright 可以实现 OpenMax 的接口，并可以让 Stagefright 引擎内的 OMXCode 调用实现的 OpenMax 接口，最终目的是使用 OpenMax IL 实现“编码/解码”功能。

在 Android 系统中通过 Stagefrigh 来定义 OpenMax 接口，具体实现内容保存在“omx”目录中。在头文件“frameworks/av/media/libstagefright/include/OMX.h”中实现了 Android 标准的 IOMX

类，此文件的主要代码如下所示。

```
class OMX : public BnOMX,
            public IBinder::DeathRecipient {
public:
    OMX();
    virtual bool livesLocally(pid_t pid);
    virtual status_t listNodes(List<ComponentInfo> *list);
    virtual status_t allocateNode(
            const char *name, const sp<IOMXObserver> &observer, node_id *node);
    virtual status_t freeNode(node_id node);
    virtual status_t sendCommand(
            node_id node, OMX_COMMANDTYPE cmd, OMX_S32 param);
    virtual status_t getParameter(
            node_id node, OMX_INDEXTYPE index,
            void *params, size_t size);
..........................................
    virtual status_t emptyBuffer(
            node_id node,
            buffer_id buffer,
            OMX_U32 range_offset, OMX_U32 range_length,
            OMX_U32 flags, OMX_TICKS timestamp);
    virtual status_t getExtensionIndex(
            node_id node,
            const char *parameter_name,
            OMX_INDEXTYPE *index);
    virtual sp<IOMXRenderer> createRenderer(
            const sp<ISurface> &surface,
            const char *componentName,
            OMX_COLOR_FORMATTYPE colorFormat,
            size_t encodedWidth, size_t encodedHeight,
            size_t displayWidth, size_t displayHeight,
            int32_t rotationDegrees);
```

文件“frameworks/av/media/libstagefright/omx/OMX.cpp”是上述 OMX.h 的实现文件，首先定义函数 createRenderer()来创建映射，然后建立一个“hardware renderer”，如果失败则建立“software renderer”。此函数的主要代码如下所示。

```
sp<IOMXRenderer> OMX::createRenderer(
        const sp<ISurface> &surface,
        const char *componentName,
        OMX_COLOR_FORMATTYPE colorFormat,
        size_t encodedWidth, size_t encodedHeight,
        size_t displayWidth, size_t displayHeight,
        int32_t rotationDegrees) {
    Mutex::Autolock autoLock(mLock);
    VideoRenderer *impl = NULL;
    void *libHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
    if (libHandle) {
        typedef VideoRenderer *(*CreateRendererWithRotationFunc)(
                const sp<ISurface> &surface,
                const char *componentName,
                OMX_COLOR_FORMATTYPE colorFormat,
                size_t displayWidth, size_t displayHeight,
                size_t decodedWidth, size_t decodedHeight,
                int32_t rotationDegrees);
        typedef VideoRenderer *(*CreateRendererFunc)(
                const sp<ISurface> &surface,
                const char *componentName,
                OMX_COLOR_FORMATTYPE colorFormat,
                size_t displayWidth, size_t displayHeight,
                size_t decodedWidth, size_t decodedHeight);
        CreateRendererWithRotationFunc funcWithRotation =
            (CreateRendererWithRotationFunc)dlsym(
                libHandle,
                "_Z26createRendererWithRotationRKN7android2spINS_8"
                "ISurfaceEEEPKc20OMX_COLOR_FORMATTYPEjjjji");
```

```
        if (funcWithRotation) {
            impl = (*funcWithRotation)(
                    surface, componentName, colorFormat,
                    displayWidth, displayHeight, encodedWidth, encodedHeight,
                    rotationDegrees);
        } else {
            CreateRendererFunc func =
                (CreateRendererFunc)dlsym(
                        libHandle,
                        "_Z14createRendererRKN7android2spINS_8ISurfaceEEEPKc20"
                        "OMX_COLOR_FORMATTYPEjjjj");
            if (func) {
                impl = (*func)(surface, componentName, colorFormat,
                        displayWidth, displayHeight, encodedWidth, encodedHeight);
            }
        }
        if (impl) {
            impl = new SharedVideoRenderer(libHandle, impl);
            libHandle = NULL;
        }
        if (libHandle) {
            dlclose(libHandle);
            libHandle = NULL;
        }
    }
    if (!impl) {
        LOGW("Using software renderer.");
        impl = new SoftwareRenderer(
                colorFormat,
                surface,
                displayWidth, displayHeight,
                encodedWidth, encodedHeight);
        if (((SoftwareRenderer *)impl)->initCheck() != OK) {
            delete impl;
            impl = NULL;
            return NULL;
        }
    }
    return new OMXRenderer(impl);
}
```

由此可见，OMXMaster 是 OMX.cpp 的真正实现者，并且能够管理 OpenMax 插件的类，这些功能是通过头文件 OMXMaster.h 和源码文件 OMXMaster.cpp 实现的。其中在文件"frameworks/av/include/media/stagefright/OMXMaster.h"中定义了类 OMXMaster，主要代码如下所示。

```
struct OMXCodec : public MediaSource,
                  public MediaBufferObserver {
    enum CreationFlags {
        kPreferSoftwareCodecs    = 1,
        kIgnoreCodecSpecificData = 2,
        kClientNeedsFramebuffer  = 4,
    };
    static sp<MediaSource> Create(      //创建类 MediaSource
            const sp<IOMX> &omx,
            const sp<MetaData> &meta, bool createEncoder,
            const sp<MediaSource> &source,
            const char *matchComponentName = NULL,
            uint32_t flags = 0);
    static void setComponentRole(      //设置组件的职责
            const sp<IOMX> &omx, IOMX::node_id node, bool isEncoder,
            const char *mime);
    virtual status_t start(MetaData *params = NULL);
    virtual status_t stop();
    virtual sp<MetaData> getFormat();
//省略声明函数代码
......................................................
```

在文件“frameworks/av/media/libstagefright/OMXMaster.cpp”中，定义静态函数 Create 将 MediaSource 作为 IOMX 插件给 OMXCode。函数 Create 的主要实现代码如下所示。

```
sp<MediaSource> OMXCodec::Create(
        const sp<IOMX> &omx,
        const sp<MetaData> &meta, bool createEncoder,
        const sp<MediaSource> &source,
        const char *matchComponentName,
        uint32_t flags) {
    const char *mime;
    bool success = meta->findCString(kKeyMIMEType, &mime);//获取 mime 信息
    CHECK(success);
    Vector<String8> matchingCodecs;
    findMatchingCodecs(
            mime, createEncoder, matchComponentName, flags, &matchingCodecs);
    if (matchingCodecs.isEmpty()) {
        return NULL;
    }
    sp<OMXCodecObserver> observer = new OMXCodecObserver;
    IOMX::node_id node = 0;
    const char *componentName;
    for (size_t i = 0; i < matchingCodecs.size(); ++i) {//使用 for 循环循环查找插件
        componentName = matchingCodecs[i].string();
        sp<MediaSource> softwareCodec = createEncoder?
            InstantiateSoftwareEncoder(componentName, source, meta):
            InstantiateSoftwareCodec(componentName, source);
        if (softwareCodec != NULL) {
            LOGV("Successfully allocated software codec '%s'", componentName);
            return softwareCodec;
        }
        LOGV("Attempting to allocate OMX node '%s'", componentName);
        uint32_t quirks = getComponentQuirks(componentName, createEncoder);
        if (!createEncoder
                && (quirks & kOutputBuffersAreUnreadable)
                && (flags & kClientNeedsFramebuffer)) {
            if (strncmp(componentName, "OMX.SEC.", 8)) {
                LOGW("Component '%s' does not give the client access to "
                     "the framebuffer contents. Skipping.",
                     componentName);
                continue;
            }
        }
        status_t err = omx->allocateNode(componentName, observer, &node);
        if (err == OK) {
            LOGV("Successfully allocated OMX node '%s'", componentName);
            sp<OMXCodec> codec = new OMXCodec(           //新建类 OMXCodec
                    omx, node, quirks,
                    createEncoder, mime, componentName,
                    source);
            observer->setCodec(codec);               //设置编码/解码器
            err = codec->configureCodec(meta, flags);
            if (err == OK) {
                return codec;
            }
            LOGV("Failed to configure codec '%s'", componentName);
        }
    }
    return NULL;
}
```

4.4.3　分析 Video Buffer 传输流程

视频播放的过程是处理 Video Buffer 的过程，在 Stagefright 框架中需要使用 VideoRenderer 插件来来实现处理功能。接下来的内容中将详细讲解在 Stagefright 框架中使用插件传输 Video Buffer 的具体流程。

（1）OMXCodec 会在开始时通过函数 read 来传送未解码的 data 数据给 decoder，并要求 decoder 将解码后的 data 传回来。对应的实现代码如下所示。

```
status_t OMXCodec::read(...)
{
  if (mInitialBufferSubmit)
  {
   mInitialBufferSubmit = false;
   drainInputBuffers(); <----- OMX_EmptyThisBuffer
   fillOutputBuffers(); <----- OMX_FillThisBuffer
  }
  ...
}
void OMXCodec::drainInputBuffers()
{
  Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexInput];

  for (i = 0; i < buffers->size(); ++i)
  {
   drainInputBuffer(&buffers->editItemAt(i));
  }
}
void OMXCodec::drainInputBuffer(BufferInfo *info)
{
  mOMX->emptyBuffer(...);
}
void OMXCodec::fillOutputBuffers()
{
  Vector<BufferInfo> *buffers = &mPortBuffers[kPortIndexOutput];
  for (i = 0; i < buffers->size(); ++i)
  {
   fillOutputBuffer(&buffers->editItemAt(i));
  }
}
void OMXCodec::fillOutputBuffer(BufferInfo *info)
{
  mOMX->fillBuffer(...);
}
```

（2）Decoder 从 input port（输入端）获取资料，然后进行解码处理，并回传 EmptyBufferDone 以通知 OMXCodec 当前的工作。对应的实现代码如下所示。

```
void OMXCodec::on_message(const omx_message &msg)
{
  switch (msg.type)
  {
   case omx_message::EMPTY_BUFFER_DONE:
   {
    IOMX::buffer_id buffer = msg.u.extended_buffer_data.buffer;
    drainInputBuffer(&buffers->editItemAt(i));
   }
  }
}
```

（3）当 OMXCodec 接收到 EMPTY_BUFFER_DONE 之后，继续传送下一个未解码的资料给 Decoder。Decoder 解码后的资料送到 output port（输出端），并回传 FillBufferDone 以通知 OMXCodec。对应的实现代码如下所示。

```
void OMXCodec::on_message(const omx_message &msg)
{
  switch (msg.type)
  {
   case omx_message::FILL_BUFFER_DONE:
   {
    IOMX::buffer_id buffer = msg.u.extended_buffer_data.buffer;
    fillOutputBuffer(info);
```

```
        mFilledBuffers.push_back(i);
        mBufferFilled.signal();
      }
    }
  }
```

当 OMXCodec 收到 FILL_BUFFER_DONE 后，将解码后的资料放入 mFilledBuffers，然后发出 mBufferFilled 信号，并要求 decoder 继续发出资料。

（4）使用函数 read 等待 mBufferFilled 信号。当 mFilledBuffers 被填入资料后，函数 read 将其指定给 buffer，并回传给 AwesomePlayer。对应的实现代码如下所示。

```
status_t OMXCodec::read(MediaBuffer **buffer, ...)
{
  ...
  while (mFilledBuffers.empty())
  {
    mBufferFilled.wait(mLock);
  }
  BufferInfo *info = &mPortBuffers[kPortIndexOutput].editItemAt(index);
  info->mMediaBuffer->add_ref();
  *buffer = info->mMediaBuffer;
}
```

函数 AwesomePlayer::onVideoEvent 除了通过 OMXCodec::read 取得解码后的资料外，还需要将这些资料（mVideoBuffer）传给 video renderer，以便在屏幕上显示。此功能的实现过程如下所示。

（1）在将 mVideoBuffer 中的资料输出之前，必须先建立 mVideoRenderer。对应的实现代码如下所示。

```
void AwesomePlayer::onVideoEvent()
{
  ...
  if (mVideoRenderer == NULL)
  {
    initRenderer_l();
  }
  ...
}
void AwesomePlayer::initRenderer_l()
{
  if (!strncmp("OMX.", component, 4))
  {
    mVideoRenderer = new AwesomeRemoteRenderer(
                        mClient.interface()->createRenderer(
                                    mISurface,
                                    component,
                                    ...));
  }
  else
  {
    mVideoRenderer = new AwesomeLocalRenderer(
                        ...,
                        component,
                        mISurface);
  }
}
```

（2）如果 video decoder 是 OMX component，则需要建立一个 AwesomeRemoteRenderer 作为 mVideoRenderer。从步骤（1）中的代码看，AwesomeRemoteRenderer 的核心功能是由函数 OMX::createRenderer 实现的。函数 createRenderer()先建立一个“hardware renderer”流程，如果失败则建立“software renderer”流程。对应的实现代码如下所示。

```
sp<IOMXRenderer> OMX::createRenderer(...)
```

```
{
  VideoRenderer *impl = NULL;
  libHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
  if (libHandle)
  {
    CreateRendererFunc func = dlsym(libHandle, ...);
    impl = (*func)(...); <----------------- Hardware Renderer
  }
  if (!impl)
  {
    impl = new SoftwareRenderer(...); <---- Software Renderer
  }
}
```

（3）如果 video decoder 是 software component，则需要建立一个 AwesomeLocalRenderer 作为 mVideoRenderer。AwesomeLocalRenderer 的 constructor 会呼叫本身的函数 init，其具体功能和函数 OMX::createRenderer 的功能相同。对应的实现代码如下所示。

```
void AwesomeLocalRenderer::init(...)
{
  mLibHandle = dlopen("libstagefrighthw.so", RTLD_NOW);
  if (mLibHandle)
  {
    CreateRendererFunc func = dlsym(...);
    mTarget = (*func)(...); <---------------- Hardware Renderer
  }
  if (mTarget == NULL)
  {
    mTarget = new SoftwareRenderer(...); <--- Software Renderer
  }
}
```

（4）建立 mVideoRenderer 后就可以开始将解码后的资料回传给它，对应的实现代码如下所示。

```
void AwesomePlayer::onVideoEvent()
{
  if (!mVideoBuffer)
  {
    mVideoSource->read(&mVideoBuffer, ...);
  }
  [Check Timestamp]
  if (mVideoRenderer == NULL)
  {
    initRenderer_l();
  }
  mVideoRenderer->render(mVideoBuffer); <----- Render Data
}
```

经过上述操作之后，Renderer 的处理过程介绍完毕。在播放多媒体的时候，需要使用 audio 来实现处理功能。在 Stagefright 框架中，audio 的部分内容是由 AudioPlayer 来处理的，在函数 AwesomePlayer::play_l 中被建立。在接下来的内容中，介绍使用 audio 的基本流程。

（1）当要求播放影音时，会同时建立并启动 AudioPlayer。对应的实现代码如下所示。

```
status_t AwesomePlayer::play_l()
{
  ...
  mAudioPlayer = new AudioPlayer(mAudioSink, ...);
  mAudioPlayer->start(...);
  ...
}
```

（2）在启动 AudioPlayer 在过程中会先读取第一笔解码后的资料，并开启 Audio Output。对应的实现代码如下所示。

```
status_t AudioPlayer::start(...)
{
  mSource->read(&mFirstBuffer);
  if (mAudioSink.get() != NULL)
  {
    mAudioSink->open(..., &AudioPlayer::AudioSinkCallback, ...);
    mAudioSink->start();
  }
  else
  {
    mAudioTrack = new AudioTrack(..., &AudioPlayer::AudioCallback, ...);
    mAudioTrack->start();
  }
}
```

在上述代码中，AudioPlayer 并没有将 mFirstBuffer 传给 Audio Output。

（3）在开启 Audio Output 的同时，AudioPlayer 将启用函数 callback()，这样每当函数 callback 被呼叫时，AudioPlayer 就会去 audio decoder 读取解码后的资料。对应的实现代码如下所示。

```
size_t AudioPlayer::AudioSinkCallback(audioSink, buffer, size, ...)
{
  return fillBuffer(buffer, size);
}
void AudioPlayer::AudioCallback(..., info)
{
  buffer = info;
  fillBuffer(buffer->raw, buffer->size);
}
size_t AudioPlayer::fillBuffer(data, size)
{
  mSource->read(&mInputBuffer, ...);
  memcpy(data, mInputBuffer->data(), ...);
}
```

由上述代码可以知道，读取解码后的 Audio 资料工作是由函数 callback 所驱动的，fillBuffer 将资料（mInputBuffer）复制到数据 data 后，Audio Output 回去取 data。

第 5 章　音频系统框架

手机离不开美妙的声音来点缀，无论是手机铃声、闹铃声音、来电铃声、彩铃还是播放音乐。在 Android 系统中，提供了功能强大的框架来实现音频处理功能。在本章的内容中，将详细讲解 Android 系统中各个音频系统框架的基本知识，为读者进入本书后面知识的学习打下基础。

5.1 音频系统基础

在 Android 音频系统中，对应的硬件设备有两部分：音频输入部分和音频输出部分。手机中的输入设备通常是话筒，输出设备通常是耳机和扬声器。Android 音频系统的核心是 Audio 系统，它在 Android 中负责音频方面的数据流传输和控制功能，也负责音频设备的管理。Audio 部分作为 Android 的 Audio 系统的输入/输出层次，一般负责播放 PCM 声音输出和从外部获取 PCM 声音，以及管理声音设备和设置。

在 Audio 系统中，整个音频管理模块主要分成以下 4 个层次。

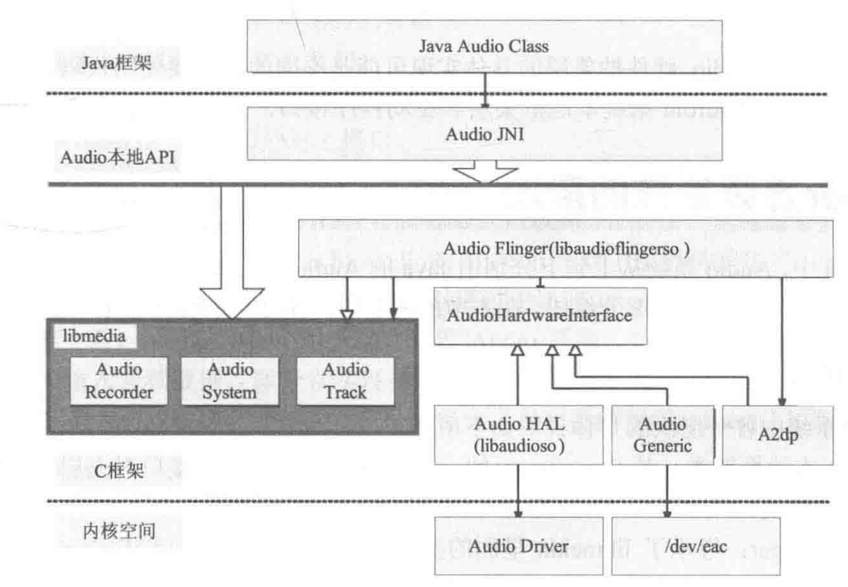
（1）Media 库提供的 Audio 系统本地部分接口。

（2）AudioFlinger 作为 Audio 系统的中间层。

（3）Audio 的硬件抽象层提供底层支持。

（4）Audio 接口通过 JNI 和 Java 框架提供给上层。

Android 音频系统的基本层次结构如图 5-1 所示。



▲图 5-1　Android 音频系统的基本层次结构

图 5-1 中各个构成部分的具体说明如下所示。

（1）Audio 的 Java 部分。

Java 部分的代码路径是：

```
frameworks/base/media/java/android/media
```

与 Audio 系统相关的 Java 包是“android.media”，在里面主要包含了与 AudioManager 和 Audio 系统等相关的类。

（2）Audio 的 JNI 部分。

JNI 部分的代码路径是：

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

Audio 的 JNI 部分的生成库是“libandroid_runtime.so”，Audio 的 JNI 是其中的一个部分。

（3）Audio 的框架部分。

框架部分的头文件路径是：

```
frameworks/base/include/media/
```

具体实现源代码路径是：

```
frameworks/base/media/libmedia/
```

Audio 本地框架是 Media 库的一部分，本部分内容被编译成库 libmedia.so，提供 Audio 部分的接口（包括基于 Binder 的 IPC 机制）。

（4）Audio Flinger。

Flinger 部分的代码路径是：

```
frameworks/base/libs/audioflinger
```

Flinger 部分的内容被编译成库 libaudioflinger.so，这是 Audio 系统的本地服务部分。

（5）Audio 的硬件抽象层接口。

硬件抽象层接口的头文件路径是：

```
hardware/libhardware_legacy/include/hardware/
```

在各个系统中，Audio 硬件抽象层的具体实现可能是不同的，需要使用代码去继承相应的类并实现它们，并作为 Android 系统本地框架层和驱动程序接口。

5.2 分析音频系统的层次

在 Android 中，Audio 系统从上到下分别由 Java 的 Audio 类、Audio 本地框架类、AudioFlinger 和 Audio 的硬件抽象层等几个部分组成。在本节的内容中，将简要介绍上述几个层次的基本知识。

5.2.1 层次说明

在 Audio 系统中各个层次的具体说明如下所示。

（1）Audio 本地框架类：是 libmedia.so 的一个部分，这些 Audio 接口对上层提供接口，由下层的本地代码去实现。

（2）AudioFlinger：继承了 libmeida 里面的接口，提供实现库 libaudiofilnger.so。这部分内容没有自己的对外头文件，上层调用的只是 libmedia 本部分的接口，但实际调用的内容是

libaudioflinger.so。

（3）JNI：在 Audio 系统中，使用 JNI 和 Java 对上层提供接口，JNI 部分通过调用 libmedia 库提供的接口来实现。

（4）Audio 硬件抽象层：提供到硬件的接口，供 AudioFlinger 调用。Audio 的硬件抽象层实际上是各个平台开发过程中需要主要关注和独立完成的部分。

因为 Android 中的 Audio 系统不涉及编解码环节，只负责上层系统和底层 Audio 硬件的交互，所以通常以 PCM 作为输入/输出格式。

在 Android 的 Audio 系统中，无论上层还是下层，都使用一个管理类和“输出/输入”类来表示整个 Audio 系统，“输出/输入”类负责数据通道。Audio 系统在各个层次之间的对应关系如表 5-1 所示。

表 5-1 Android 各个层次的对应关系

层次说明	Audio 管理环节	Audio 输出	Audio 输入
Java 层	android.media AudioSystem	android.media AudioTrack	android.media AudioRecorder
本地框架层	AudioSystem	AudioTrack	AudioRecorder
AudioFlinger	IAudioFlinger	IAudioTrack	IAudioRecorder
硬件抽象层	AudioHardwareInterface	AudioStreamOut	AudioStreamIn

5.2.2 Media 库中的 Audio 框架

在 Media 库中提供了 Audio 系统的核心框架，在库中实现了 AudioSystem、AudioTrack 和 AudioRecorder 三个类。另外还提供了 IAudioFlinger 类接口，通过此类可以获得 IAudioTrack 和 IAudioRecorder 两个接口，分别用于声音的播放和录制功能。AudioTrack 和 AudioRecorder 分别通过调用 IAudioTrack 和 IAudioRecorder 来实现。

Audio 系统的头文件被保存在“frameworks\av\include\media”目录中，其中包含的主要的头文件如下所示。

- AudioSystem.h：Media 库的 Audio 部分对上层的总管接口。
- IAudioFlinger.h：需要下层实现的总管接口。
- AudioTrack.h：放音部分对上接口。
- IAudioTrack.h：放音部分需要下层实现的接口。
- AudioRecorder.h：录音部分对上接口。
- IAudioRecorder.h：录音部分需要下层实现的接口。

其中文件 IAudioFlinger.h、IAudioTrack.h 和 IAudioRecorder.h 的接口是通过下层的继承来实现的。文件 AudioFlinger.h、AudioTrack.h 和 AudioRecorder.h 是对上层提供的接口，它们既供本地程序调用（例如声音的播放器、录制器等），也可以通过 JNI 向 Java 层提供接口。

从具体功能上看，AudioSystem 用于综合管理 Audio 系统，而 AudioTrack 和 AudioRecorder 分别负责输出和输入音频数据，即分别实现播放和录制功能。

AudioTrack 是 Audio 输出环节的类，在里面包含了最重要的接口 write()，主要代码如下所示。

```
class AudioTrack : virtual public RefBase
{
public:
    enum channel_index {
        MONO   = 0,
        LEFT   = 0,
```

```
        RIGHT  = 1
    };

    /* Events used by AudioTrack callback function (audio_track_cblk_t).
     * Keep in sync with frameworks/base/media/java/android/media/AudioTrack.java
NATIVE_EVENT_*.
     */
    enum event_type {
        EVENT_MORE_DATA = 0,        // Request to write more data to buffer
                                    // If this event is delivered but the callback handler
                                    // does not want to write more data, the handler must
explicitly
                                    // ignore the event by setting frameCount to zero
        EVENT_UNDERRUN = 1,         // Buffer underrun occurred
        EVENT_LOOP_END = 2,         // Sample loop end was reached; playback restarted from
                                    // loop start if loop count was not 0
        EVENT_MARKER = 3,           // Playback head is at the specified marker position
                                    // (See setMarkerPosition())
        EVENT_NEW_POS = 4,          // Playback head is at a new position
                                    // (See setPositionUpdatePeriod())
        EVENT_BUFFER_END = 5        // Playback head is at the end of the buffer
    };
{
    typedef void (*callback_t)(int event,
void* user, void *info);
    AudioTrack( int streamType,
                uint32_t sampleRate  = 0,    // 音频的采样律
                int format           = 0,    // 音频的格式（例如8位或者16位的PCM）
                int channelCount      = 0,    // 音频的通道数
                int frameCount        = 0,    // 音频的帧数
                uint32_t flags        = 0,
                callback_t cbf        = 0,
                void* user           = 0,
                int notificationFrames = 0);
    void        start();
    void        stop();
    void        flush();
    void        pause();
    void        mute(bool);
    ssize_t      write(const void* buffer, size_t size);
……………………………………………………
        enum {
            NO_MORE_BUFFERS = 0x80000001,   // same name in AudioFlinger.h, ok to be
different value
            STOPPED = 1
        };
……………………………………………………
```

类AudioRecord是用于实现和Audio录制相关的功能，主要实现代码如下所示。

```
   class AudioRecord
    {
    enum event_type {
        EVENT_MORE_DATA = 0,        // Request to read more data from PCM buffer
        EVENT_OVERRUN = 1,          // PCM buffer overrun occurred
        EVENT_MARKER = 2,           // Record head is at the specified marker position
                                    // (See setMarkerPosition())
        EVENT_NEW_POS = 3,          // Record head is at a new position
                                    // (See setPositionUpdatePeriod())
    };
    class Buffer
    {
    public:
        size_t      frameCount;     // number of sample frames corresponding to size
                                    // on input it is the number of frames available
                                    // on output is the number of frames actually drained

        size_t      size;           // total size in bytes == frameCount * frameSize
        union {
```

```
        void*      raw;
        short*     i16;      // signed 16-bit
        int8_t*    i8;       // unsigned 8-bit, offset by 0x80
    };
};
typedef void (*callback_t)(int event, void* user, void *info);
static status_t getMinFrameCount(size_t* frameCount,
                                 uint32_t sampleRate,
                                 audio_format_t format,
                                 audio_channel_mask_t channelMask);
```

在类 AudioTrack 和 AudioRecord 中，函数 read 和 write 的参数都是内存的指针及其大小，内存中的内容一般表示的是 Audio 的原始数据（PCM 数据）。这两个类还涉及 Auido 数据格式、通道数、帧数目等参数，可以在建立时指定，也可以在建立之后使用 set()函数进行设置。

另外，在 libmedia 库中提供的只是一个 Audio 系统框架，其中类 AudioSystem、AudioTrack 和 AudioRecord 分别调用下层的接口 IAudioFlinger、IAudioTrack 和 IAudioRecord 来实现。另外的一个接口是 IAudioFlingerClient，它作为向 IAudioFlinger 中注册的监听器，相当于使用回调函数获取 IAudioFlinger 运行时的信息。

5.2.3 本地代码

在 Android 系统中，AudioFlinger 是 Audio 音频系统的中间层，能够作为 libmedia 提供的 Audio 部分接口的实现。这部分本地代码的路径如下所示。

```
frameworks/base/libs/audioflinger
```

文件 AudioFlinger.h 和 AudioFlinger.cpp 是实现 AudioFlinger 的核心文件，在里面提供了类 AudioFlinger，此类是一个 IAudioFlinger 的实现，其接口代码如下所示。

```
class AudioFlinger : public BnAudioFlinger,
public IBinder::DeathRecipient
{
public:
    static void instantiate();
    virtual    status_t    dump(int fd, const Vector<String16>& args);
    virtual sp<IAudioTrack> createTrack(
// 获得音频输出接口（Track）
                              audio_stream_type_t streamType,
                              uint32_t sampleRate,
                              audio_format_t format,
                              audio_channel_mask_t channelMask,
                              size_t frameCount,
                              track_flags_t *flags,
                              const sp<IMemory>& sharedBuffer,
                              audio_io_handle_t output,
                              pid_t tid,  // -1 means unused, otherwise must be valid non-0
                              int *sessionId,
                              status_t *status) = 0;
// 获得音频输出接口（Record）
    virtual sp<IAudioRecord> openRecord(
                              audio_io_handle_t input,
                              uint32_t sampleRate,
                              audio_format_t format,
                              audio_channel_mask_t channelMask,
                              size_t frameCount,
                              track_flags_t flags,
                              pid_t tid,  // -1 means unused, otherwise must be valid non-0
                              int *sessionId,
                              status_t *status) = 0;
```

由上述代码可以看出，AudioFlinger 使用函数 createTrack()来创建音频的输出设备 IAudioTrack，使用函数 openRecord()来创建音频的输入设备 IAudioRecord，并且还使用接口“get/set”来实现控

制功能。

构造函数 AudioFlinger()的代码如下所示。

```
AudioFlinger::AudioFlinger()
{
    mHardwareStatus = AUDIO_HW_IDLE;
    mAudioHardware = AudioHardwareInterface::create();
    mHardwareStatus = AUDIO_HW_INIT;
    if (mAudioHardware->initCheck() == NO_ERROR) {
        mHardwareStatus = AUDIO_HW_OUTPUT_OPEN;
        status_t status;
        AudioStreamOut *hwOutput =
            mAudioHardware->openOutputStream (AudioSystem::PCM_16_BIT, 0, 0, &status);
        mHardwareStatus = AUDIO_HW_IDLE;
        if (hwOutput) {
            mHardwareMixerThread =
            new MixerThread(this, hwOutput, AudioSystem::AUDIO_OUTPUT_HARDWARE);
        } else {
            LOGE("Failed to initialize hardware  output stream, status: %d", status);
        }
    #ifdef WITH_A2DP
        mA2dpAudioInterface = new A2dpAudioInterface();
        AudioStreamOut *a2dpOutput = mA2dpAudioInterface->openOutputStream(AudioSystem::
PCM_16_BIT, 0, 0, &status);
        if (a2dpOutput) {
            mA2dpMixerThread = new MixerThread(this, a2dpOutput, AudioSystem::AUDIO_
OUTPUT_A2DP);
            if (hwOutput) {
                uint32_t frameCount = ((a2dpOutput->bufferSize()/a2dpOutput->frameSize())
* hwOutput->sampleRate()) / a2dpOutput->sampleRate();
                MixerThread::OutputTrack *a2dpOutTrack = new MixerThread::OutputTrack
(mA2dpMixerThread,
                hwOutput->sampleRate(),
                AudioSystem::PCM_16_BIT,
                hwOutput->channelCount(),
                frameCount);
                mHardwareMixerThread->setOuputTrack(a2dpOutTrack);
            }
        } else {
            LOGE("Failed to initialize A2DP output stream, status: %d", status);
        }
#endif
        setRouting(AudioSystem::MODE_NORMAL,              AudioSystem::ROUTE_SPEAKER,
AudioSystem::ROUTE_ALL);
        setRouting(AudioSystem::MODE_RINGTONE,            AudioSystem::ROUTE_SPEAKER,
AudioSystem::ROUTE_ALL);
        setRouting(AudioSystem::MODE_IN_CALL,            AudioSystem::ROUTE_EARPIECE,
AudioSystem::ROUTE_ALL);
        setMode(AudioSystem::MODE_NORMAL);
        setMasterVolume(1.0f);
        setMasterMute(false);
        mAudioRecordThread = new  AudioRecordThread(mAudioHardware, this);
        if (mAudioRecordThread != 0) {
            mAudioRecordThread->run("AudioRecordThread", PRIORITY_URGENT_AUDIO);
        }
    } else {
        LOGE("Couldn't even initialize the stubbed audio hardware!");
    }
}
```

由上述代码可以看出，在初始化 AudioFlinger 之后，会首先获得放音设备，然后为混音器（Mixer）建立线程并建立放音设备线程，最后在线程中获得放音设备。

在文件 frameworks\av\services\audioflinger\AudioResampler.h 中定义了类 AudioResampler，此类是一个音频重取样器的工具类，定义代码如下所示。

```
class AudioResampler {
public:
```

```
    enum src_quality {
        DEFAULT=0,
        LOW_QUALITY=1,              // 线性差值算法
        MED_QUALITY=2,              // 立方差值算法
        HIGH_QUALITY=3              // fixed multi-tap FIR 算法
        VERY_HIGH_QUALITY=4,
    };
    static AudioResampler* create(int bitDepth, int inChannelCount,
            int32_t sampleRate, src_quality quality=DEFAULT_QUALITY);
    virtual ~AudioResampler();
    virtual void init() = 0;
    virtual void setSampleRate(int32_t inSampleRate);
    virtual void setVolume(int16_t left, int16_t right);
    virtual void setLocalTimeFreq(uint64_t freq);

    // set the PTS of the next buffer output by the resampler
    virtual void setPTS(int64_t pts);

    virtual void resample(int32_t* out, size_t outFrameCount,
            AudioBufferProvider* provider) = 0;

    virtual void reset();
    virtual size_t getUnreleasedFrames() const { return mInputIndex; }

    // called from destructor, so must not be virtual
    src_quality getQuality() const { return mQuality; }
```

在上述音频重取样工具类中，包含了如下4种质量。

- 低等质量（LOW_QUALITY）：使用线性差值算法实现；
- 中等质量（MED_QUALITY）：使用立方差值算法实现；
- 高等质量（HIGH_ QUALITY）：使用FIR（有限阶滤波器）实现；
- 非常高质量（VERY_HIGH_QUALITY）。

在类AudioResampler中，AudioResamplerOrder1是线性实现，AudioResamplerCubic.*文件提供立方实现方式，AudioResamplerSinc.*提供FIR实现。

通过文件AudioMixer.h和AudioMixer.cpp实现了一个Audio系统混音器，它被AudioFlinger调用，一般用于在声音输出之前的处理，提供多通道处理、声音缩放、重取样。AudioMixer调用了AudioResampler。

5.2.4 分析JNI代码

在Android中的Audio系统中，通过JNI向Java层提供功能强大的接口，这样就可以在Java层通过JNI接口完成Audio系统的大部分操作。

Audio JNI的实现代码保存在“frameworks/base/core/jni”目录下，在目录中主要有3个核心文件，它们分别对应了Android Java框架中的3个类的支持，具体说明如下所示。

- android.media.AudioSystem：负责Audio系统的总体控制。
- android.media.AudioTrack：负责Audio系统的输出环节。
- android.media.AudioRecorder：负责Audio系统的输入环节。

在Android系统的Java层中，可以对Audio系统进行控制和数据流操作，其中控制操作和底层的处理基本一致；但是对于数据流操作，由于Java不支持指针，因此接口被封装成了另外的形式。例如在音频输出功能中，通过文件android_media_AudioTrack.cpp提供了写字节和写短整型的接口类型。对应代码如下所示。

```
static jint android_media_AudioTrack_native_
write(JNIEnv *env, jobject thiz,
jbyteArray javaAudioData,
```

```
jint offsetInBytes, jint sizeInBytes,
jint javaAudioFormat) {
    jbyte* cAudioData = NULL;
    AudioTrack *lpTrack = NULL;
    lpTrack = (AudioTrack *)env->GetIntField(
                thiz, javaAudioTrackFields. Native TrackInJavaObj);
    ssize_t written = 0;
    if (lpTrack->sharedBuffer() == 0) {
    //进行写操作
        written = lpTrack->write(cAudioData +
offsetInBytes, sizeInBytes);
    } else {
        if (javaAudioFormat == javaAudioTrackFields.PCM16) {
            memcpy(lpTrack->sharedBuffer()->pointer(),
                    cAudioData+offsetInBytes, sizeInBytes);
            written = sizeInBytes;
        } else if (javaAudioFormat == javaAudioTrackFields.PCM8) {
            int count = sizeInBytes;
            int16_t *dst = (int16_t *)lpTrack->sharedBuffer()->pointer();
            const int8_t *src = (const int8_t *)
(cAudioData + offsetInBytes);
            while(count--) {
                *dst++ = (int16_t)(*src++^0x80) << 8;
            }
            written = sizeInBytes;
        }
    }
    env->ReleasePrimitiveArrayCritical(javaAudioData, cAudioData, 0);
    return (int)written;
}
```

5.2.5 Java 层代码简介

在 Android 的 Audio 系统中，和 Java 相关的类定义在 android.media 包中，Java 部分的代码保存在“frameworks/base/media/java/android/media”目录中，在里面主要实现了如下所示的类：

- android.media.AudioSystem
- android.media. Audio Track
- android.media.AudioRecorder
- android.media.AudioFormat

其中前 3 个类和本地代码是对应的，在 AudioFormat 中提供了一些和 Audio 相关的枚举值。在此需要注意的是在 Audio 系统的 Java 代码中，虽然可以通过 AudioTrack 和 AudioRecorder 的 write()和 read()接口在 Java 层对 Audio 的数据流进行操作，但是，更多的时候并不需要这样做，而是在本地代码中直接调用接口进行数据流的“输入/输出”，在 Java 层只进行控制类方面的操作，不处理具体的数据流。

5.3 Audio 系统的硬件抽象层

Audio 硬件抽象层是 Audio 驱动程序和 Audio 本地框架类 AudioFlinger 的接口。根据 Android 系统对接口的定义，Audio 硬件抽象层是 C++类的接口，需要在继承接口中定义 3 个类来实现 Audio 硬件抽象层，这 3 个类分别实现总控、输入和输出功能。要想实现一个 Android 的硬件抽象层，则需要实现 AudioHardwareInterface、AudioStream Out 和 AudioStreamIn 这 3 个类，并将代码编译成动态库 libauido.so。AudioFlinger 会连接这个动态库，并调用其中的 createAudioHardware 函数来获取接口。在本节的内容中，将详细讲解 Audio 系统的硬件抽象层的知识。

5.3.1 Audio 硬件抽象层基础

Audio 系统的硬件抽象层是 AudioFlinger 和 Audio 硬件之间的接口，在不同系统的移植过程中可以有不同的实现方式。其中 Audio 硬件抽象层的接口路径如下所示。

```
hardware/libhardware_legacy/include/hardware/
```

在上述路径的核心文件是 AudioHardwareBase.h 和 AudioHardwareInterface.h。

作为 Android 系统的 Audio 硬件抽象层，既可以基于 Linux 标准的 ALSA 或 OSS 音频驱动来实现，也可以基于私有的 Audio 驱动接口来实现。

在文件 AudioHardwareInterface.h 中分别定义了类 AudioStreamOut、AudioStreamIn 和 AudioHardwareInterface。类 AudioStreamOut 和 AudioStreamIn 分别描述了音频输出设备和音频输入设备，其中负责数据流的接口分别是函数 wirte 和 read，其参数是表示一块内存的指针和长度；另外还有一些设置和获取接口。类 AudioStreamOut 和 AudioStreamIn 的实现代码如下所示。

```
class AudioStreamOut {
public:
    virtual             ~AudioStreamOut() = 0;
    virtual status_t     setVolume(float volume) = 0;
    virtual ssize_t      write(const void* buffer, size_t bytes) = 0;
    virtual int         channelCount() const = 0;
virtual int         format() const = 0;
virtual status_t     setVolume(float volume) = 0;
virtual ssize_t      write(const void* buffer, size_t bytes) = 0;
virtual status_t dump(int fd, const Vector<String16>& args) = 0;
};
class AudioStreamIn {
public:
    virtual             ~AudioStreamIn() = 0;
    virtual status_t     setGain(float gain) = 0;
    virtual ssize_t      read(void* buffer, ssize_t bytes) = 0;
    virtual int         channelCount() const = 0;
     virtual int         format() const = 0;
    virtual status_t     setGain(float gain) = 0;
    virtual ssize_t      read(void* buffer, ssize_t bytes) = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) = 0;
};
```

由此可见，类 AudioStreamOut 和 AudioStreamIn 是两个对应的接口类，分别实现输出和输入环节。类 AudioStreamOut 和 AudioStreamIn 都需要通过 Audio 硬件抽象层的核心 AudioHardwareInterface 接口类来获取。接口类 AudioHardwareInterface 的实现代码如下所示。

```
class AudioHardwareInterface {
public:
                    AudioHardwareInterface();
    virtual            ~AudioHardwareInterface() { }
    virtual status_t    initCheck() = 0;
    virtual status_t    standby() = 0;
    virtual status_t    setVoiceVolume(float volume) = 0;
    virtual status_t    setMasterVolume(float volume) = 0;
    virtual status_t    setRouting(int mode, uint32_t routes);
    virtual status_t    getRouting(int mode, uint32_t* routes);
    virtual status_t    setMode(int mode);
    virtual status_t    getMode(int* mode);
    virtual status_t    setMicMute(bool state) = 0;
    virtual status_t    getMicMute(bool* state) = 0;
    virtual status_t    setParameter(const char* key, const char* value);
    virtual AudioStreamOut* openOutputStream(                // 打开输出流
                            int format=0,
                            int channelCount=0,
                            uint32_t sampleRate=0) = 0;
    virtual AudioStreamIn* openInputStream(     // 打开输入流
```

```
                                    int format,
                                    int channelCount,
                                    uint32_t sampleRate) = 0;
    virtual status_t dumpState(int fd, const Vector<String16>& args);
    static AudioHardwareInterface* create();
```

在上述 AudioHardwareInterface 接口的实现代码中，分别使用函数 openOutputStream 和 openInputStream 来获取类 AudioStreamOut 和类 AudioStreamIn，将它们分别作为音频输入设备和输出设备来使用。

除此之外，在文件 AudioHardwareInterface.h 中还定义了 C 语言的接口来获取一个 AudioHardware Interface 类型的指针，具体的定义代码如下所示。

```
extern "C" AudioHardwareInterface* createAudioHardware(void);
```

5.3.2　分析 AudioFlinger 中的 Audio 硬件抽象层的实现

在 Android 系统的 AudioFlinger 中，可以通过编译宏的方式来选择到底用哪一个 Audio 硬件抽象层。可选择的 Audio 硬件抽象层既可以作为参考设计，也可以在没有实际的 Audio 硬件抽象层时使用，目的是保证系统的正常运行。

1. 编译文件

文件 Android.mk 是 AudioFlinger 的编译文件，定义代码如下所示。

```
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
 LOCAL_STATIC_LIBRARIES += libaudiointerface
else
 LOCAL_SHARED_LIBRARIES += libaudio
endif
LOCAL_MODULE:= libaudioflinger
include $(BUILD_SHARED_LIBRARY)
```

在上述代码中，当 BOARD_USES_GENERIC_AUDIO 为 True 时连接 libaudiointer face.a 静态库；当 BOARD_USES_GENERIC_AUDIO 为 False 时连接 libaudiointerface.so 动态库，在大多数的情况下使用后者。

另外，在文件 Android.mk 中也生成了 libaudiointerface.a，具体代码如下所示。

```
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
    AudioHardwareGeneric.cpp \
    AudioHardwareStub.cpp \
    AudioDumpInterface.cpp \
    AudioHardwareInterface.cpp
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libmedia \
    libhardware_legacy
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
 LOCAL_CFLAGS += -DGENERIC_AUDIO
endif
LOCAL_MODULE:= libaudiointerface
include $(BUILD_STATIC_LIBRARY)
```

在上述代码中，分别编译 4 个源文件来生成 libaudiointerface.a 静态库。其中文件 AudioHardwareInterface.cpp 用于实现基础类和管理；文件 AudioHardwareGeneric.cpp、AudioHard wareStub.cpp 和 AudioDumpInterface.cpp 分别代表一种 Auido 硬件抽象层的实现，具体说明如下所示。

- AudioHardwareGeneric.cpp：实现基于特定驱动的通用 Audio 硬件抽象层。
- AudioHardwareStub.cpp：实现 Audio 硬件抽象层的一个桩。
- AudioDumpInterface.cpp：实现输出到文件的 Audio 硬件抽象层。

在文件 AudioHardwareInterface.cpp 中定义了 AudioHardwareInterface::create 函数，此函数是 Audio 硬件抽象层的创建函数，主要代码如下所示。

```
AudioHardwareInterface* AudioHardwareInterface::create()
{
   AudioHardwareInterface* hw = 0;
   char value[PROPERTY_VALUE_MAX];
#ifdef GENERIC_AUDIO
   hw = new AudioHardwareGeneric();
// 此处用通用的 Audio 硬件抽象层
#else
   if (property_get("ro.kernel.qemu", value, 0)) {
      LOGD("Running in emulation - using generic audio driver");
      hw = new AudioHardwareGeneric();
   }
   else {
      LOGV("Creating Vendor Specific AudioHardware");
      hw = createAudioHardware();
// 此处用实际的 Audio 硬件抽象层
   }
#endif
   if (hw->initCheck() != NO_ERROR) {
      LOGW("Using stubbed audio hardware.No sound will be produced.");
      delete hw;
      hw = new AudioHardwareStub();
// 此处用实际的 Audio 硬件抽象层的桩实现
   }
#ifdef DUMP_FLINGER_OUT
   hw = new AudioDumpInterface(hw);
// 此处用实际的 Audio 的 Dump 接口实现
#endif
   return hw;
}
```

2. 桩方式实现

在文件 AudioHardwareStub.h 和 AudioHardwareStub.cpp 中，通过桩方式实现了一个 Android 硬件抽象层。桩方式不操作实际的硬件和文件，只是进行了空操作。当系统没有实际的 Audio 设备时才使用桩方式实现，目的是保证系统的正常工作。如果使用这个硬件抽象层，实际上 Audio 系统的输入和输出都将为空。

在文件 AudioHardwareStub.h 中定义了类 AudioStreamOutStub 和 AudioStreamInStub，分别实现输出和输入。主要实现代码如下所示。

```
class AudioStreamOutStub : public AudioStreamOut {
public:
   virtual status_t    set(int format, int
channelCount, uint32_t sampleRate);
   virtual uint32_t    sampleRate() const { return 44100; }
   virtual size_t      bufferSize() const { return 4096; }
   virtual int         channelCount() const { return 2; }
   virtual int         format() const { return
 AudioSystem::PCM_16_BIT; }
   virtual uint32_t    latency() const { return 0; }
   virtual status_t    setVolume(float volume) { return NO_ERROR; }
   virtual ssize_t     write(const void* buffer, size_t bytes);
   virtual status_t    standby();
   virtual status_t    dump(int fd, const Vector<String16>& args);
};
class AudioStreamInStub : public AudioStreamIn {
```

```
public:
   virtual status_t set(int format, int
 channelCount, uint32_t sampleRate, AudioSystem::
audio_in_acoustics acoustics);
   virtual uint32_t    sampleRate() const { return 8000; }
   virtual size_t      bufferSize() const { return 320; }
   virtual int         channelCount() const { return 1; }
   virtual int         format() const { return
 AudioSystem::PCM_16_BIT; }
   virtual status_t    setGain(float gain) { return NO_ERROR; }
   virtual ssize_t     read(void* buffer, ssize_t bytes);
   virtual status_t    dump(int fd, const Vector<String16>& args);
   virtual status_t    standby() { return NO_ERROR; }
};
```

在上述代码中，只用缓冲区大小、采样率和通道数这 3 个固定的参数将一些函数直接无错误返回。

然后需要使用类 AudioHardwareStub 来继承类 AudioHardwareBase，也就是继承类 AudioHardwareInterface。主要实现代码如下所示。

```
 class AudioHardwareStub : public  AudioHardwareBase
 {
 public:
                        AudioHardwareStub();
    virtual             ~AudioHardwareStub();
    virtual status_t    initCheck();
    virtual status_t    setVoiceVolume(float volume);
    virtual status_t    setMasterVolume(float volume);
    virtual status_t    setMicMute(bool state)
{ mMicMute = state;  return  NO_ERROR; }
    virtual status_t    getMicMute(bool* state)
{ *state = mMicMute ; return NO_ERROR; }
    virtual status_t    setParameter(const
char* key, const char* value)
            { return NO_ERROR; }
    virtual AudioStreamOut* openOutputStream(       //打开输出流
                              int format=0,
                              int channelCount=0,
                              uint32_t sampleRate=0,
                              status_t *status=0);

    virtual AudioStreamIn* openInputStream(     //打开输入流
                              int format,
                              int channelCount,
                              uint32_t sampleRate,
                              status_t *status,
                              AudioSystem::audio_in_acoustics acoustics);
 ......
```

为了保证可以输入和输出声音，桩实现的主要内容是实现类 AudioStreamOutStub 和类 AudioStreamInStub 的“读/写”函数。主要实现代码如下所示。

```
ssize_t AudioStreamOutStub::write(const void* buffer, size_t bytes)
{
   usleep(bytes * 1000000 / sizeof(int16_t) /
channelCount() / sampleRate());
   return bytes;
}
ssize_t AudioStreamInStub::read(void* buffer, ssize_t bytes)
{
   usleep(bytes * 1000000 / sizeof(int16_t) /
channelCount() / sampleRate());
   memset(buffer, 0, bytes);
   return bytes;
}
```

当使用这个接口来输入和输出音频时，和真实的设备并没有任何关系，输出和输入都使用延时来完成。在输出时不会播出声音，但是返回值表示全部内容已经输出完成；在输入时会返回全部为 0 的数据。

3. 通用 Audio 硬件抽象层

在 Android 系统中，文件 AudioHardwareGeneric.h 和 AudioHardwareGeneric.cpp 实现了通用的 Audio 硬件抽象层。与前面介绍的桩实现方式不同，这是一个真正能够使用的 Audio 硬件抽象层，但是它需要 Android 的一种特殊的声音驱动程序的支持。

在通用硬件抽象层中，类 AudioStreamOutGeneric、AudioStreamInGeneric 和 AudioHardwareGeneric 分别继承 Audio 硬件抽象层的 3 个接口。对应代码如下所示。

```
class AudioStreamOutGeneric : public AudioStreamOut {
   //通用 Audio 输出类的接口
};
class AudioStreamInGeneric : public AudioStreamIn {
   //通用 Audio 输入类的接口
};
class AudioHardwareGeneric : public AudioHardwareBase
{
   //通用 Audio 控制类的接口
};
```

在文件 AudioHardwareGeneric.cpp 中使用的驱动程序是“/dev/eac”，这是一个非标准程序，定义设备路径的代码如下所示。

```
static char const * const kAudioDeviceName = "/dev/eac";
```

> **注意**　eac 是 Linux 中的一个 misc 驱动程序，作为 Android 的通用音频驱动，写设备表示放音，读设备表示录音。

在 Linux 操作系统中，“/dev/eac”驱动程序在文件系统中的节点主设备号为 10，是次设备号自动生成的。通过构造函数 AudioHardwareGeneric()可以打开这个驱动程序的设备节点。对应代码如下所示。

```
AudioHardwareGeneric::AudioHardwareGeneric()
   : mOutput(0), mInput(0),  mFd(-1), mMicMute(false)
{
   mFd = ::open(kAudioDeviceName, O_RDWR); //打开通用音频设备的节点
}
```

此音频设备是一个比较简单的驱动程序，在里面并没有很多设置接口，只是用写设备来表示录音，用读设备来表示放音。放音和录音支持的都是 16 位的 PCM。对应的实现代码如下所示。

```
ssize_t AudioStreamOutGeneric::write(const void* buffer, size_t bytes)
{
   Mutex::Autolock _l(mLock);
   return ssize_t(::write(mFd, buffer, bytes));    //写入硬件设备
}
ssize_t AudioStreamInGeneric::read(void* buffer, ssize_t bytes)
{
   AutoMutex lock(mLock);
   if (mFd < 0) {
      return NO_INIT;
   }
   return ::read(mFd, buffer, bytes);              // 读取硬件设备
}
```

尽管 AudioHardwareGeneric 是一个可以真正工作的 Audio 硬件抽象层，但是这种实现方式非常简单，不支持各种设置，参数也只能使用默认的。而且，这种驱动程序需要在 Linux 核心加入 eac 驱动程序的支持。

4. 具备 Dump 功能的 Audio 硬件抽象层

在文件 AudioDumpInterface.h 和 AudioDumpInterface.cpp 中，提供了具备 Dump 功能的 Audio 硬件抽象层，目的是将输出的 Audio 数据写入到文件中。

其实 AudioDumpInterface 本身支持 Audio 输出功能，但是不支持输入功能。在文件 AudioDumpInterface.h 中定义类的代码如下所示。

```
class AudioStreamOutDump : public AudioStreamOut {
public:
                    AudioStreamOutDump( AudioStreamOut* FinalStream);
                    ~AudioStreamOutDump();
                    virtual ssize_t    write(const void* buffer, size_t bytes);
   virtual uint32_t    sampleRate() const { return mFinalStream->sampleRate(); }
   virtual size_t     bufferSize() const { return mFinalStream->bufferSize(); }
   virtual int        channelCount() const { return
 mFinalStream->channelCount(); }
   virtual int        format() const { return mFinalStream->format(); }
   virtual uint32_t    latency() const { return  mFinalStream->latency(); }
   virtual status_t    setVolume(float volume)
    { return  mFinalStream->setVolume(volume); }
   virtual status_t    standby();
};
class AudioDumpInterface : public AudioHardwareBase
{
   virtual AudioStreamOut* openOutputStream(
                           int format=0,
                           int channelCount=0,
                           uint32_t sampleRate=0,
                           status_t *status=0);
}
```

在上述代码中，只实现了 AudioStreamOut 输出，而没有实现 AudioStreamIn 输入。由此可见，此 Audio 硬件抽象层只支持输出功能，不支持输入功能。其中输出文件的名称被定义为如下格式。

```
#define FLINGER_DUMP_NAME "/data/FlingerOut.pcm"
```

在文件 AudioDumpInterface.cpp 中，通过函数 AudioStreamOut 实现写操作，写入的对象就是这个文件。对应的实现代码如下所示。

```
ssize_t AudioStreamOutDump::write(const void* buffer, size_t bytes)
{
    ssize_t ret;
    ret = mFinalStream->write(buffer, bytes);
    if(!mOutFile && gFirst) {
        gFirst = false;
        mOutFile = fopen(FLINGER_DUMP_NAME, "r");
        if(mOutFile) {
            fclose(mOutFile);
            mOutFile = fopen(FLINGER_DUMP_NAME, "ab");
// 打开输出文件
        }
    }
    if (mOutFile) {
        fwrite(buffer, bytes, 1, mOutFile);
// 写文件输出内容
    }
    return ret;
}
```

如果文件是打开的，则可以使用追加方式写入。当使用这个 Audio 硬件抽象层时，播放的内容（PCM）将全部被写入到文件。而且这个类支持各种格式的输出，具体什么格式将取决于调用者的设置。

使用 AudioDumpInterface 的目的并不是为了实际的应用，而是为了调试使用的类。当使用播放器调试音频时，有时无法确认是解码器的问题还是 Audio 输出单元的问题，这时就可以用这个类来替换实际的 Audio 硬件抽象层，将解码器输出的 Audio 的 PCM 数据写入文件中，由此可以判断解码器的输出是否正确。

5.3.3 真正实现 Audio 硬件抽象层

想要实现一个真正的 Audio 硬件抽象层，需要完成和 5.2 节中实现硬件抽象层类似的工作。例如可以基于 Linux 标准的音频驱动 OSS（Open Sound System）或 ALSA（Advanced Linux Sound Architecture）驱动程序来实现。

（1）基于 OSS 驱动程序实现。

对于 OSS 驱动程序来说，实现方式和前面的 AudioHardwareGeneric 方式类似，数据流的读/写操作通过对“/dev/dsp”设备的“读/写”来完成；区别在于 OSS 支持了更多的 ioctl 进行设置，还涉及通过“/dev/mixer”设备进行控制，并支持更多不同的参数。

（2）基于 ALSA 驱动程序实现。

对于 ALSA 驱动程序来说，实现方式一般不是直接调用驱动程序的设备节点，而是先实现用户空间的 alsa-lib，然后 Audio 硬件抽象层通过调用 alsa-lib 来实现。

在实现 Audio 硬件抽象层时，如果系统中有多个 Audio 设备，此时可由硬件抽象层自行处理 setRouting()函数设定。例如可以选择支持多个设备的同时输出，或者有优先级输出。对于这种情况，数据流一般来自函数 AudioStreamOut::write()，可由硬件抽象层确定输出方法。对于某种特殊的情况，也有可能采用硬件直接连接的方式，此时数据流可能并不来自于上面的 write()，这样就没有数据通道，只有控制接口。Audio 硬件抽象层也是可以处理这种情况的。

5.4 分析编码/解码过程

为了减小传输过程中的数据流量和存储空间，传输的媒体文件必须进行压缩处理。在目前的主流移动计算平台上，支持的音频记录格式主要有 AAC 和 AMR-NB 两种。另外，部分厂商也提供了对元数据 PCM 的记录支持。

其中 AAC（Adpative Audio Coding）是在 MP3 基础上开发出来的，其主要算法在 1997 年研发完成。与 MP3 相比，AAC 采用了修正离散余弦变换（Modified Discrete Cosiine Transform，MDCT）算法，具有更高的压缩率，能够支持最多 48 个全音域声道，最高支持 8～96kHz 的采样速率，具有更高的解码效率，占用的解码资源更少。AAC 有效地解决了 MP3 的压缩率较低、音质尤其是在低码率下不够理想、仅有两个声道等问题。目前支持 AAC 的厂家主要有 Nokia、Apple、Qalcomm、Panasonic 等。

> **注意** 因为目前 OpenCORE 不支持 AAC 编码，所以本书中不再对 AAC 的编解码进行过多的介绍。

AMR 根据带宽的不同可以分为自适应多速率宽带编码（AMR WideBand，AMR-WB）和自适应多速率窄带编码（AMR NarrowBand，AMR-NB）。其中 AMR-WB 的音频带宽为 50～7000Hz，

采样速率为 16kHz，而 AMR-NB 的音频带宽为 300～3400Hz，采样速率为 8kHz。AMR-WB 同时被 ITU-T 和 3GPP 采用，也称 G722.2 标准。AMR-WB 抗扰度优于 AMR-NB。

在 Android 系统中，在上层框架中提供了 MediaRecorder 类等来支持音频内容的编码。

5.4.1　AMR 编码

无论是在 2G 还是 3G 通信中，AMR 都是最常用的一种音频编码格式。根据制定组织的不同，帧结构略有不同。目前业界采用的 AMR 的帧结构主要由 ETS、WMF、IETF 和 3GPP 等制定。其中 GDM&&GPRS 采用的是 ETS 制定的帧结构，WCDMA&&TD-SCDMA 采用的是 3GPP 制定的帧结构。

在 OpenCore 中，相关的编码过程保存在"external\opencore\codecs_ v2\audio\ gsm_amr\amr_nb\enc"目录下。支持的帧结构包括 AMR_TX_WMF、AMR_TX_IF2、AMR_TX_ETS、AMR_TX_IETF 等。其中 AMR_TX_WMF 表示的是无线多媒体论坛（Wireless Multimedia Forum，WMF）制定的帧结构；AMR_TX_IF2 表示的是 3GPP 制定的帧结构；AMR_TX_ETS 表示的是欧洲电信标准（European Telecommunication Standard，ETS）制定的帧结构；AMR_TX_IETF 表示是 IETF 制定的帧结构。

编码的入口 AMREncode()函数位于文件"opencore/codecs_v2/audio/gsm_amr/amr_nb/enc/src/amrencode.cpp"中，函数 AMREncode()首先调用 GSM EFR 编码器进行编码，然后根据指定的输出格式参数 output_format 的值，将 GSM EFR 编码器的输出转换为相应的帧结构。下面代码是 AMR 的编码过程。

```
Word16 AMREncode(
    void *pEncState,
    void *pSidSyncState,
    enum Mode mode,
    Word16 *pEncInput,
    UWord8 *pEncOutput,
    enum Frame_Type_3GPP *p3gpp_frame_type,
    Word16 output_format
)
{
    Word16 ets_output_bfr[MAX_SERIAL_SIZE+2];
    UWord8 *ets_output_ptr;
    Word16 num_enc_bytes = -1;
    Word16 i;
    enum TXFrameType tx_frame_type;
    enum Mode usedMode = MR475;
    /* WMF 或 IF2 编码*/
    if ((output_format == AMR_TX_WMF) | (output_format == AMR_TX_IF2))
    {
        /* 通话帧编码速度(20 ms) */
#ifndef CONSOLE_ENCODER_REF
        /* GSM EFR 编码器的 PV 实现*/
        GSMEncodeFrame(pEncState, mode, pEncInput, ets_output_bfr, &usedMode);
#else
        /* GSM EFR 编码器的 ETS 实现*/
        Speech_Encode_Frame(pEncState, mode, pEncInput, ets_output_bfr, &usedMode);
#endif
        /* 判断帧类型 */
        sid_sync(pSidSyncState, usedMode, &tx_frame_type);
        if (tx_frame_type != TX_NO_DATA)
        {
            /* There is data to transmit */
            *p3gpp_frame_type = (enum Frame_Type_3GPP) usedMode;
            /* 为 SID 帧添加 SID 类型和模式信息*/
            if (*p3gpp_frame_type == AMR_SID)
            {
                /* Add SID type to encoder output buffer */
```

```
                if (tx_frame_type == TX_SID_FIRST)
                {
                    ets_output_bfr[AMRSID_TXTYPE_BIT_OFFSET] &= 0x0000;
                }
                else if (tx_frame_type == TX_SID_UPDATE)
                {
                    ets_output_bfr[AMRSID_TXTYPE_BIT_OFFSET] |= 0x0001;
                }
                for (i = 0; i < NUM_AMRSID_TXMODE_BITS; i++)
                {
                    ets_output_bfr[AMRSID_TXMODE_BIT_OFFSET+i] =
                        (mode >> i) & 0x0001;
                }
            }
        }
        else
        {
            /* 无数据传递*/
            *p3gpp_frame_type = (enum Frame_Type_3GPP)AMR_NO_DATA;
        }
        /* 判断输出帧类型 */
        if (output_format == AMR_TX_WMF)
        {
            /* 转换为 IETF 帧结构 */
            ets_to_wmf(*p3gpp_frame_type, ets_output_bfr, pEncOutput);
            /* Set up the number of encoded WMF bytes */
            num_enc_bytes = WmfEncBytesPerFrame[(Word16) *p3gpp_frame_type];
        }
        else if (output_format == AMR_TX_IF2)
        {
            /* 转换为 AMR IF2 帧结构*/
            ets_to_if2(*p3gpp_frame_type, ets_output_bfr, pEncOutput);
            num_enc_bytes = If2EncBytesPerFrame[(Word16) *p3gpp_frame_type];
        }
    }
    /* ETS 帧编码*/
    else if (output_format == AMR_TX_ETS)
    {
#ifndef CONSOLE_ENCODER_REF
        GSMEncodeFrame(pEncState, mode, pEncInput, &ets_output_bfr[1], &usedMode);
#else
        Speech_Encode_Frame(pEncState, mode, pEncInput, &ets_output_bfr[1], &usedMode);
#endif
        *p3gpp_frame_type = (enum Frame_Type_3GPP) usedMode;
        sid_sync(pSidSyncState, usedMode, &tx_frame_type);
        ets_output_bfr[0] = tx_frame_type;

        if (tx_frame_type != TX_NO_DATA)
        {
            ets_output_bfr[1+MAX_SERIAL_SIZE] = (Word16) mode;
        }
        else
        {
            ets_output_bfr[1+MAX_SERIAL_SIZE] = -1;
        }
        ets_output_ptr = (UWord8 *) & ets_output_bfr[0];
        for (i = 0; i < 2*(MAX_SERIAL_SIZE + 2); i++)
        {
            *(pEncOutput + i) = *ets_output_ptr;
            ets_output_ptr += 1;
        }
        /* Set up the number of encoded bytes */
        num_enc_bytes = 2 * (MAX_SERIAL_SIZE + 2);
    }
    /* 无效的帧格式 */
    else
    {
        /* Invalid output format, set up error code */
        num_enc_bytes = -1;
```

```
    }
    return(num_enc_bytes);
}
```

接下来需要将 GSM ETS 帧结构转换为 AMR IF2 帧结构。在文件“opencore/codecs_v2/audio/gsm_amr/amr_nb/enc/src/ets_to_if2.cpp”中，将 GSM ETS 帧结构转换为 AMR IF2 帧结构的实现代码如下所示。

```
void ets_to_if2(
    enum Frame_Type_3GPP frame_type_3gpp,
    Word16 *ets_input_ptr,
    UWord8 *if2_output_ptr)
{
    Word16  i;
    Word16  k;
    Word16  j = 0;
    Word16 *ptr_temp;
    Word16  bits_left;
    UWord8  accum;
    if (frame_type_3gpp < AMR_SID)
    {
        if2_output_ptr[j++] = (UWord8)(frame_type_3gpp) |
                              (ets_input_ptr[reorderBits[frame_type_3gpp][0]] << 4) |
                              (ets_input_ptr[reorderBits[frame_type_3gpp][1]] << 5) |
                              (ets_input_ptr[reorderBits[frame_type_3gpp][2]] << 6) |
                              (ets_input_ptr[reorderBits[frame_type_3gpp][3]] << 7);
        for (i = 4; i < numOfBits[frame_type_3gpp] - 7;)
        {
            if2_output_ptr[j]  =
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]];
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 1;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 2;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 3;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 4;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 5;
            if2_output_ptr[j] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 6;
            if2_output_ptr[j++] |=
                (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << 7;
        }
        bits_left = 4 + numOfBits[frame_type_3gpp] -
                    ((4 + numOfBits[frame_type_3gpp]) & 0xFFF8);
        if (bits_left != 0)
        {
            if2_output_ptr[j] = 0;
            for (k = 0; k < bits_left; k++)
            {
                if2_output_ptr[j] |=
                    (UWord8) ets_input_ptr[reorderBits[frame_type_3gpp][i++]] << k;
            }
        }
    }
    else
    {
        if (frame_type_3gpp != AMR_NO_DATA)
        {
            if2_output_ptr[j++] = (UWord8)(frame_type_3gpp) |
                                  (ets_input_ptr[0] << 4) | (ets_input_ptr[1] << 5) |
                                  (ets_input_ptr[2] << 6) | (ets_input_ptr[3] << 7);
            ptr_temp = &ets_input_ptr[4];
            bits_left = ((4 + numOfBits[frame_type_3gpp]) & 0xFFF8);
            for (i = (bits_left - 7) >> 3; i > 0; i--)
            {
```

```
            accum  = (UWord8) * (ptr_temp++);
            accum |= (UWord8) * (ptr_temp++) << 1;
            accum |= (UWord8) * (ptr_temp++) << 2;
            accum |= (UWord8) * (ptr_temp++) << 3;
            accum |= (UWord8) * (ptr_temp++) << 4;
            accum |= (UWord8) * (ptr_temp++) << 5;
            accum |= (UWord8) * (ptr_temp++) << 6;
            accum |= (UWord8) * (ptr_temp++) << 7;
            if2_output_ptr[j++] = accum;
        }
        bits_left = 4 + numOfBits[frame_type_3gpp] - bits_left;
        if (bits_left != 0)
        {
            if2_output_ptr[j] = 0;
            for (i = 0; i < bits_left; i++)
            {
                if2_output_ptr[j] |= (ptr_temp[i] << i);
            }
        }
    }
    else
    {
        /* When there is no data, LSnibble of first octet */
        /* is the 3GPP frame type, MSnibble is zeroed out */
        if2_output_ptr[j++] = (UWord8)(frame_type_3gpp);
    }
  }
  return;
}
```

另外，函数 ets_to_ietf 和函数 ets_to_wmf 能够分别针对 ETS 帧结构转换为 IETF 帧结构和 WMF 帧结构，其具体实现分别位于文件 ets_to_if2.cpp 和 ets_to_wmf.cpp 中，读者可以查看具体实现代码，在本书不再进行介绍。

5.4.2 AMR 解码

在 AMR 解码过程中，OpenCore 定义了 ETS (AMR-WB、AMR-NB)、ITU(AMR-WB)、MIME_IETF(AMR-WB)、WMF (AMR-NB)、IF2 (AMR-NB)这 5 种帧结构，帧结构的定义和编码实现没有统一。在目前仅使用了 ETS、WMF、IF2 这 3 种帧结构。

AMR 的解码过程和其编码过程相反，首先需要根据输入格式参数 input_format 确定当前要解码的帧结构。如果是 IETF 或者 AMRIF2，则首先将其帧结构转换为 ETS 帧结构，然后调用函数 GSMFrameDecode()进行解码；如果是 ETS 帧结构，直接调用函数 GSMFrameDecode()进行解码。在文件"opencore/codecs_v2/audio/gsm_amr/amr_nb/dec/src/amrdecode.cpp"中实现了 AMR 的解码过程，主要实现代码如下所示。

```
Word16 AMRDecode(
    void                    *state_data,
    enum Frame_Type_3GPP      frame_type,
    UWord8                  *speech_bits_ptr,
    Word16                  *raw_pcm_buffer,
    bitstream_format          input_format
)
{
    Word16 *ets_word_ptr;
    enum Mode mode = (enum Mode)MR475;
    int modeStore;
    int tempInt;
    enum RXFrameType rx_type = RX_NO_DATA;
    Word16 dec_ets_input_bfr[MAX_SERIAL_SIZE];
    Word16 i;
    Word16 byte_offset = -1;
    Speech_Decode_FrameState *decoder_state
```

```
    = (Speech_Decode_FrameState *) state_data;
    if ((input_format == MIME_IETF) | (input_format == IF2))
    {
        if (input_format == MIME_IETF)
        {
            /* 转换编码 */
            wmf_to_ets(frame_type, speech_bits_ptr, dec_ets_input_bfr, &(decoder_state->
decoder_amrState.common_amr_tbls));

            /*下个框架开始的地址垂距*/
            byte_offset = WmfDecBytesPerFrame[frame_type];
        }
        else   /* 必须输入 IF2 帧 */
        {
            /*将接踵而来的 packetized 未加工的 IF2 数据转换成 ETS 格式*/
            if2_to_ets(frame_type, speech_bits_ptr, dec_ets_input_bfr, &(decoder_state->
decoder_amrState.common_amr_tbls));

            /* Address offset of the start of next frame */
            byte_offset = If2DecBytesPerFrame[frame_type];
        }
        /*这时，以 ETS 格式输入数据*/
        /*确定 AMR 编解码器方式和 AMR RX 框架类型*/
        if (frame_type <= AMR_122)
        {
            mode = (enum Mode) frame_type;
            rx_type = RX_SPEECH_GOOD;
        }
        else if (frame_type == AMR_SID)
        {
            /*在输入缓冲区前以可读方式方式清除*/
            modeStore = 0;
            for (i = 0; i < NUM_AMRSID_RXMODE_BITS; i++)
            {
                tempInt = dec_ets_input_bfr[AMRSID_RXMODE_BIT_OFFSET+i] << i;
                modeStore |= tempInt;
            }
            mode = (enum Mode) modeStore;

            /*得到 RX 框架类型*/
            if (dec_ets_input_bfr[AMRSID_RXTYPE_BIT_OFFSET] == 0)
            {
                rx_type = RX_SID_FIRST;
            }
            else
            {
                rx_type = RX_SID_UPDATE;
            }
        }
        else if (frame_type < AMR_NO_DATA)
        {
            /*无效 frame_type，返回误差编码*/
            byte_offset = -1;   /*  !!! */
        }
        else
        {
            mode = decoder_state->prev_mode;
            rx_type = RX_NO_DATA;
        }
    }

    /* ETS 帧 */
    else if (input_format == ETS)
    {
        /*转换未加工的 ETS 数据*/
        ets_word_ptr = (Word16 *) speech_bits_ptr;
        /* 获取 RX 帧的类型*/
        rx_type = (enum RXFrameType) * ets_word_ptr;
        ets_word_ptr++;
```

```
        for (i = 0; i < MAX_SERIAL_SIZE; i++)
        {
            dec_ets_input_bfr[i] = *ets_word_ptr;
            ets_word_ptr++;
        }
        /*得到编解码器方式*/
        if (rx_type != RX_NO_DATA)
        {
            /* Get mode from input bitstream */
            mode = (enum Mode) * ets_word_ptr;
        }
        else
        {
            /*如果没有收到数据*/
            mode = decoder_state->prev_mode;
        }
        byte_offset = 2 * (MAX_SERIAL_SIZE + 2);
    }
    else
    {/
        byte_offset = -1;
    }
    if (byte_offset != -1)
    {
#ifndef CONSOLE_DECODER_REF
        GSMFrameDecode(decoder_state, mode, dec_ets_input_bfr, rx_type,
                       raw_pcm_buffer);
#else
        Speech_Decode_Frame(decoder_state, mode, dec_ets_input_bfr, rx_type,
                            raw_pcm_buffer);
#endif
        decoder_state->prev_mode = mode;
    }
    return (byte_offset);
}
```

在文件“codecs_v2/audio/gsm_amr/amr_nb/dec/src/if2_to_ets.cpp”中定义函数 if2_to_ets，用于将 AMR IF2 帧结构转换为 ETS 帧结构。函数 if2_to_ets 的主要实现代码如下所示。

```
void if2_to_ets(
    enum Frame_Type_3GPP frame_type_3gpp,
    UWord8   *if2_input_ptr,
    Word16   *ets_output_ptr,
    CommonAmrTbls* common_amr_tbls)
{

    Word16 i;
    Word16 j;
    Word16 x = 0;
    const Word16* numCompressedBytes_ptr = common_amr_tbls->numCompressedBytes_ptr;
    const Word16* numOfBits_ptr = common_amr_tbls->numOfBits_ptr;
    const Word16* const* reorderBits_ptr = common_amr_tbls->reorderBits_ptr;
    if (frame_type_3gpp < AMR_SID)
    {
        for (j = 4; j < 8; j++)
        {
            ets_output_ptr[reorderBits_ptr[frame_type_3gpp][x++]] =
                (if2_input_ptr[0] >> j) & 0x01;
        }
        for (i = 1; i < numCompressedBytes_ptr[frame_type_3gpp]; i++)
        {
            for (j = 0; j < 8; j++)
            {
                if (x >= numOfBits_ptr[frame_type_3gpp])
                {
                    break;
                }
                ets_output_ptr[reorderBits_ptr[frame_type_3gpp][x++]] =
```

```
                    (if2_input_ptr[i] >> j) & 0x01;
            }
        }
    }
    else
    {
        for (j = 4; j < 8; j++)
        {
            ets_output_ptr[x++] =
                (if2_input_ptr[0] >> j) & 0x01;
        }
        for (i = 1; i < numCompressedBytes_ptr[frame_type_3gpp]; i++)
        {
            for (j = 0; j < 8; j++)
            {
                ets_output_ptr[x++] =
                    (if2_input_ptr[i] >> j) & 0x01;
            }
        }
    }
    return;
}
```

在目前的解码的实现上，仅仅支持 AMR IF2、ETS 帧结构，并不支持 IETF 帧结构。另外，函数 wmf_to_ets()的具体实现位于文件 amrdecode.cpp 文件中，在本书中将不再介绍此函数，读者课后看一下其具体实现代码。

5.4.3　解码 MP3

MP3（Moving Picture Experts Group Audio Layer III）是目前最流行的音频编码格式，MP3 的解码需要经过同步及检错、哈夫曼解码、逆量化、立体声解码、反锯齿、IMDCT 和子带合成等运算，其中 IMDCT 过程的运算量占到了整个解码运算总量的 19%。

在文件“opencore/codecs_v2/omx/omx_mp3/src/mp3_dec.cpp”中实现了对 MP3 文件的解码，主要实现代码如下所示。

```
int Mp3Decoder::Mp3DecodeAudio(OMX_S16* aOutBuff,
                          OMX_U32* aOutputLength, OMX_U8** aInputBuf,
                          OMX_U32* aInBufSize, OMX_S32* aIsFirstBuffer,
                          OMX_AUDIO_PARAM_PCMMODETYPE* aAudioPcmParam,
                          OMX_AUDIO_PARAM_MP3TYPE* aAudioMp3Param,
                          OMX_BOOL aMarkerFlag,
                          OMX_BOOL* aResizeFlag)
{
    int32 Status = MP3DEC_SUCCESS;
    *aResizeFlag = OMX_FALSE;
    if (iInitFlag == 0)
    {
        if (*aIsFirstBuffer != 0)
        {
            e_equalization EqualizType = iMP3DecExt->equalizerType;
            iMP3DecExt->inputBufferCurrentLength = 0;
            iInputUsedLength = 0;
            iAudioMp3Decoder->StartL(iMP3DecExt, false, false, false, EqualizType);
        }
        iInitFlag = 1;
    }
    iMP3DecExt->pInputBuffer = *aInputBuf + iInputUsedLength;
    iMP3DecExt->pOutputBuffer = &aOutBuff[0];
    iMP3DecExt->inputBufferCurrentLength = *aInBufSize;
    iMP3DecExt->inputBufferUsedLength = 0;
    if (OMX_FALSE == aMarkerFlag)
    {
        //如果没有标志位，则检测帧的边界
        Status = iAudioMp3Decoder->SeekMp3Synchronization(iMP3DecExt);
```

```
        if (1 == Status)
        {
            if (0 == iMP3DecExt->inputBufferCurrentLength)
            {
                *aInBufSize -= iMP3DecExt->inputBufferMaxLength;
                iInputUsedLength += iMP3DecExt->inputBufferMaxLength;
                iMP3DecExt->inputBufferUsedLength += iMP3DecExt->inputBufferMaxLength;;
                return MP3DEC_SUCCESS;
            }
            else
            {
                *aInputBuf += iInputUsedLength;
                iMP3DecExt->inputBufferUsedLength = 0;
                iInputUsedLength = 0;
                return MP3DEC_INCOMPLETE_FRAME;
            }
        }
    }
    Status = iAudioMp3Decoder->ExecuteL(iMP3DecExt);
    if (MP3DEC_SUCCESS == Status)
    {
        *aInBufSize -= iMP3DecExt->inputBufferUsedLength;
        if (0 == *aInBufSize)
        {
            iInputUsedLength = 0;
        }
        else
        {
            iInputUsedLength += iMP3DecExt->inputBufferUsedLength;
        }
        *aOutputLength = iMP3DecExt->outputFrameSize * iMP3DecExt->num_channels;
        if (0 == *aIsFirstBuffer)
        {
            (*aIsFirstBuffer)++;
            aAudioPcmParam->nSamplingRate = iMP3DecExt->samplingRate;
            aAudioPcmParam->nChannels = iMP3DecExt->num_channels;
            *aResizeFlag = OMX_TRUE;
        }
        return Status;
    }
    else if (Status == MP3DEC_INVALID_FRAME)
    {
        *aInBufSize = 0;
        iInputUsedLength = 0;
    }
    else if (Status == MP3DEC_INCOMPLETE_FRAME)
    {
        *aInputBuf += iInputUsedLength;
        iMP3DecExt->inputBufferUsedLength = 0;
        iInputUsedLength = 0;
    }
    else
    {
        *aInputBuf += iInputUsedLength;
        iInputUsedLength = 0;
    }
    return Status;
}
```

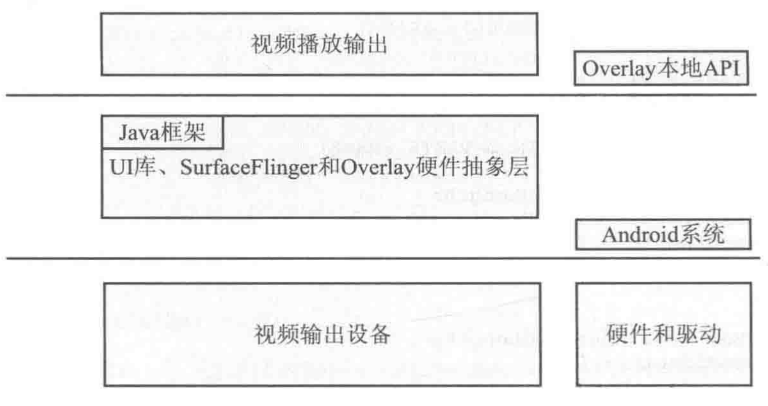
第 6 章　视频系统框架

在当前的智能手机系统应用中，多媒体视频应用比较常见。我们通常在手机中播放各种各样的视频文件，也通常用手机在线观看精美大片。在 Android 系统中，为开发人员提供了功能强大的视频系统框架。在本章的内容中，将详细讲解 Android 系统中各个视频框架的基本知识，为读者进入本书后面知识的学习打下基础。

6.1 视频系统结构

在 Android 系统中，视频输出系统对应的是 Overlay 子系统，此系统是 Android 的一个可选系统，用于加速显示输出视频数据。视频输出系统的硬件通常叠加在主显示区之上的额外的叠加显示区。这个额外的叠加显示区和主显示区使用独立的显示内存。在通常情况下，主显示区用于输出图形系统，通常是 RGB 颜色空间。额外显示区用于输出视频，通常是 YUV 颜色空间。主显示区和叠加显示区通过 Blending（硬件混淆）自动显示在屏幕上。在软件部分我们无需关心叠加的实现过程，但是可以控制叠加的层次顺序和叠加层的大小等内容。

Overlay 系统的基本层次结构如图 6-1 所示。

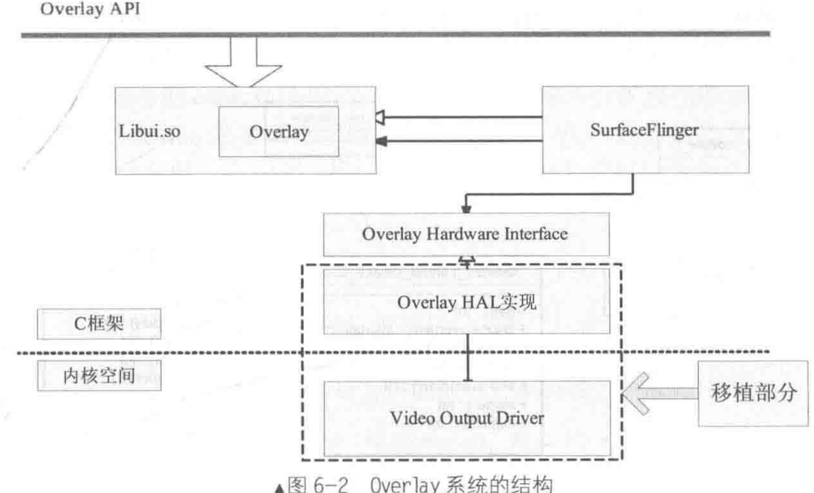


▲图 6-1　Overlay 的基本层次结构

Android 中的 Overlay 系统没有 Java 部分，在里面只包含了视频输出的驱动程序、硬件抽象层和本地框架等。Overlay 系统的结构如图 6-2 所示。

在图 6-2 所示的系统结构中，各个构成部分的具体说明如下所示。

（1）Overlay 驱动程序：通常是基于 FrameBuffer 或 V4L2 的驱动程序。在此文件中主要定义了两种结构，分别是 data device 和 control device，这两个结构体分别针对 data device 和 control device 的函数 open()和函数 close()。这两个函数是注册到 device_module 里面的函数。



▲图 6-2 Overlay 系统的结构

（2）Overlay 硬件抽象层：代码路径如下所示。

```
hardware/qcom/display/liboverlay/overlay.h
```

Overlay 硬件抽象层是一个 Android 中标准的硬件模块，其接口只有一个头文件。

（3）Overlay 服务部分：代码路径如下所示。

```
frameworks/native/services/surfaceflinger/
```

由此可见，Overlay 系统的服务部分包含在 SurfaceFlinger 中，此层次的内容比较简单，主要功能是通过类 LayerBuffer 实现的。首先要明确的是 SurfaceFlinger 只是负责控制 merge Surface，比如说计算出两个 Surface 重叠的区域，至于 Surface 需要显示的内容，则通过 Skia、Opengl 和 Pixflinger 来计算。所以我们在介绍 SurfaceFlinger 之前先忽略里面存储的内容究竟是什么，先弄清楚它对 merge 的一系列控制的过程，然后再结合 2D、3D 引擎来看它的处理过程。

（4）本地框架代码。

在 Overlay 系统中，本地框架的头文件路径如下所示。

```
frameworks/native/include/ui
```

源代码路径如下所示。

```
frameworks/native/libs/ui
```

Overlay 系统只是整个框架的一部分，主要功能是通过类 Ioverlay 和 Overlay 实现的，源代码被编译成 libui.so，它提供的 API 主要在视频输出和照相机取景模块中使用。

6.2 分析硬件抽象层

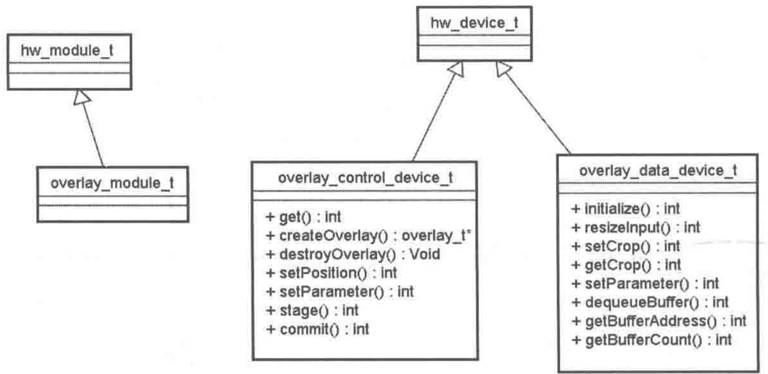
Overlay 系统的硬件抽象层是一个硬件模块。在本节的内容中，将简要介绍 Overlay 系统的硬件抽象层的基本知识，为后面的知识做好铺垫。

6.2.1 Overlay 系统硬件抽象层的接口

在如下文件中定义 Overlay 系统硬件抽象层的接口。

```
hardware/qcom/display/liboverlay/overlay.h
```

在文件 overlay.h 中，主要定义了 data device 和 control device 两个结构。并提供针对 data device 和 control device 的函数 open()和函数 close()。文件 overlay.h 的代码结构如图 6-3 所示。



▲图 6-3　文件 overlay.h 的代码结构

（1）定义 Overlay 控制设备和 Overlay 数据设备，它们的名称被定义为如下两个字符串。

```
#define OVERLAY_HARDWARE_CONTROL    "control"
#define OVERLAY_HARDWARE_DATA       "data"
```

（2）定义一个枚举 enum，也定义了所有支援的 Format，FrameBuffer 会根据 Format 和 width、height 来决定 Buffer（FrameBuffer 里面用来显示的 Buffer）的大小。定义 enum 的代码如下所示。

```
enum {
   OVERLAY_FORMAT_RGBA_8888    = HAL_PIXEL_FORMAT_RGBA_8888,
   OVERLAY_FORMAT_RGB_565      = HAL_PIXEL_FORMAT_RGB_565,
   OVERLAY_FORMAT_BGRA_8888    = HAL_PIXEL_FORMAT_BGRA_8888,
   OVERLAY_FORMAT_YCbCr_422_SP = HAL_PIXEL_FORMAT_YCbCr_422_SP,
   OVERLAY_FORMAT_YCbCr_420_SP = HAL_PIXEL_FORMAT_YCbCr_420_SP,
   OVERLAY_FORMAT_YCrCb_420_SP = HAL_PIXEL_FORMAT_YCrCb_420_SP,
   OVERLAY_FORMAT_YCbYCr_422_I = HAL_PIXEL_FORMAT_YCbCr_422_I,
   OVERLAY_FORMAT_YCbYCr_420_I = HAL_PIXEL_FORMAT_YCbCr_420_I,
   OVERLAY_FORMAT_CbYCrY_422_I = HAL_PIXEL_FORMAT_CbYCrY_422_I,
   OVERLAY_FORMAT_CbYCrY_420_I = HAL_PIXEL_FORMAT_CbYCrY_420_I,
   OVERLAY_FORMAT_DEFAULT      = 99    // The actual color format is determined
                                       // by the overlay
};
```

（3）定义和 Overlay 系统相关的结构体。

在文件 overlay.h 中和 Overlay 系统相关的结构体是 overlay_t 和 overlay_handle_t，主要代码如下所示。

```
typedef struct overlay_t {
   uint32_t           w;                         //宽
   uint32_t           h;                         //高
   int32_t            format;                    //颜色格式
   uint32_t           w_stride;                  //一行的内容
   uint32_t           h_stride;                  //一列的内容
   uint32_t           reserved[3];
   /* returns a reference to this overlay's handle (the caller doesn't
    * take ownership) */
   overlay_handle_t     (*getHandleRef)(struct overlay_t* overlay);
   uint32_t           reserved_procs[7];
} overlay_t;
```

结构体 overlay_handle_t 是在内部使用的结构体，用于保存 Overlay 硬件设备的句柄。在使用的过程中，需要从 overlay_t 获取 overlay_handle_t。其中上一层的使用只实现结构体 overlay_handle_t 指针的传递，具体的操作是在 Overlay 的硬件抽象层中完成的。

（4）定义结构体 overlay_control_device_t。此结构体定义了一个 control device，里面的成员除了 common 都是函数，这些函数就是我们需要去实现的，在实现的时候我们会基于这个结构体扩展出一个关于 control device 的 context 结构体，context 结构体内部会扩充一些信息并且包含 control device。common 每一个 device 都必须有，而且必须放到第一位，目的只是为了 overlay_control_device_t 和 hw_device_t 做匹配。overlay_control_device_t 的定义代码如下所示。

```
struct overlay_control_device_t {
    struct hw_device_t common;
    int (*get)(struct overlay_control_device_t *dev, int name);
//建立设备
    overlay_t* (*createOverlay)(struct overlay_control_device_t *dev,
            uint32_t w, uint32_t h, int32_t format);
//释放资源，分配的 handle 和 control device 的内存
    void (*destroyOverlay)(struct overlay_control_device_t *dev,
            overlay_t* overlay);
//设置 overlay 的显示范围。(如果是 camera 的 preview，那么 h、w 要和 preview h、w 一致。)
    int (*setPosition)(struct overlay_control_device_t *dev,
            overlay_t* overlay,
            int x, int y, uint32_t w, uint32_t h);
//获取 overlay 的显示范围
    int (*getPosition)(struct overlay_control_device_t *dev,
            overlay_t* overlay,
            int* x, int* y, uint32_t* w, uint32_t* h);
    int (*setParameter)(struct overlay_control_device_t *dev,
            overlay_t* overlay, int param, int value);
    int (*stage)(struct overlay_control_device_t *dev, overlay_t* overlay);
    int (*commit)(struct overlay_control_device_t *dev, overlay_t* overlay);
};
```

（5）定义结构 overlay_data_device_t。此结构和 overlay_control_device_t 类似。在具体使用上，overlay_control_device_t 负责初始化、销毁和控制类的操作，overlay_data_device_t 用于显示内存输出的数据操作。结构 overlay_data_device_t 的定义代码如下所示。

```
struct overlay_data_device_t {
    struct hw_device_t common;
//通过参数 handle 来初始化 data device
    int (*initialize)(struct overlay_data_device_t *dev,
            overlay_handle_t handle);
//重新配置显示参数 w、h。使这两个参数生效我这里需要 close，然后重新 open
    int (*resizeInput)(struct overlay_data_device_t *dev,
            uint32_t w, uint32_t h);
//下面两个分别设置显示的区域和获取显示的区域，当播放的时候，需要根据坐标和宽高来定义如何显示这些数据
    int (*setCrop)(struct overlay_data_device_t *dev,
            uint32_t x, uint32_t y, uint32_t w, uint32_t h) ;
    int (*getCrop)(struct overlay_data_device_t *dev,
        uint32_t* x, uint32_t* y, uint32_t* w, uint32_t* h) ;

    int (*setParameter)(struct overlay_data_device_t *dev,
            int param, int value);

    int (*dequeueBuffer)(struct overlay_data_device_t *dev,
                overlay_buffer_t *buf);
    int (*queueBuffer)(struct overlay_data_device_t *dev,
            overlay_buffer_t buffer);
    void* (*getBufferAddress)(struct overlay_data_device_t *dev,
            overlay_buffer_t buffer);
    int (*getBufferCount)(struct overlay_data_device_t *dev);
    int (*setFd)(struct overlay_data_device_t *dev, int fd);
};
```

6.2.2 实现 Overlay 系统的硬件抽象层

在实现 Overlay 系统的硬件抽象层时，具体实现方法取决于硬件和驱动程序，根据设备需要进行处理。具体来说分为如下两种情况。

（1）FrameBuffer 驱动程序方式。

在此方式下，需要先实现函数 getBufferAddress()，再返回通过 mmap 获得 FrameBuffer 的指针即可。如果没有双缓冲的问题，不需要真正实现函数 dequeueBuffer()和 queueBuffer()。上述函数的实现文件是 overlay.cpp，此文件被保存在如下目录中。

```
Hardware/qcom/display/liboverlay/overlay.cpp
```

函数 getBufferAddress()用于返回 FrameBuffer 内部显示的内存，通过 mmap 获取内存地址。函数的实现代码如下所示。

```
void* Overlay::getBufferAddress(overlay_buffer_t buffer)
{
    if (mStatus != NO_ERROR) return NULL;
    return mOverlayData->getBufferAddress(mOverlayData, buffer);
}
```

函数 dequeueBuffer()和 queueBuffer()的实现代码如下所示。

```
status_t Overlay::dequeueBuffer(overlay_buffer_t* buffer)
{
    if (mStatus != NO_ERROR) return mStatus;
    return  mOverlayData->dequeueBuffer(mOverlayData, buffer);
}

status_t Overlay::queueBuffer(overlay_buffer_t buffer)
{
    if (mStatus != NO_ERROR) return mStatus;
    return mOverlayData->queueBuffer(mOverlayData, buffer);
}
```

（2）Video for Linux 2 方式。

如果使用 Video for Linux 2 的输出驱动，函数 dequeueBuffer()和 queueBuffer()和调用驱动时主要 ioctl 是一致的，即分别调用 VIDIOC_QBUF 和 VIDIOC_DQBUF 即可直接实现。至于其他的初始化工作，可以在 initialize 中进行处理。因为存在视频数据队列，所以此处处理的内容比一般的帧缓冲区要复杂，但是可以实现更高的性能。

由此可见，在某一个硬件系统中，Overlay 的硬件层和 Overlay 系统的调用者都是特定实现的，所以只需匹配上下层代码即可实现，并不要一一满足每一个要求，各个接口可以根据具体情况来灵活使用。

6.2.3 实现接口

在 Android 系统中，Overlay 系统提供了接口 overlay，此接口用于叠加在主显示层上面的另外一个显示层。此叠加的显示层经常用作视频的输出或相机取景器的预览界面。文件 Overlay.h 的主要内部实现类是 Overlay 和 OverlayRef。OverlayRef 需要和 Surface 配合使用，通过 Isurface 可以创建出 OverlayRef。RefBase 的主要代码如下所示。

```
class Overlay : public virtual RefBase
{
public:
    Overlay(const sp<OverlayRef>& overlayRef);
    void destroy();
    //获取 overlay handle，可以根据自己的需要扩展，扩展之后就有很多数据了
    overlay_handle_t getHandleRef() const;
    //获取 framebuffer  用于显示的内存地址
    status_t dequeueBuffer(overlay_buffer_t* buffer);
    status_t queueBuffer(overlay_buffer_t buffer);
    status_t resizeInput(uint32_t width, uint32_t height);
    status_t setCrop(uint32_t x, uint32_t y, uint32_t w, uint32_t h) ;
    status_t getCrop(uint32_t* x, uint32_t* y, uint32_t* w, uint32_t* h) ;
```

```
    status_t setParameter(int param, int value);
    void* getBufferAddress(overlay_buffer_t buffer);

    /*获取属性的信息*/
    uint32_t getWidth() const;
    uint32_t getHeight() const;
    int32_t getFormat() const;
    int32_t getWidthStride() const;
    int32_t getHeightStride() const;
    int32_t getBufferCount() const;
    status_t getStatus() const;

private:
    virtual ~Overlay();

    sp<OverlayRef> mOverlayRef;
    overlay_data_device_t *mOverlayData;
    status_t mStatus;
};

Overlay(const sp<OverlayRef>& overlayRef);
```

在上述代码中，通过 surface 来控制 Overlay，也可以在不使用 Overlay 的情况下统一进行管理。此处是通过 overlayRef 来创建 Overlay，一旦获取了 Overlay 就可以通过这个 Overlay 来获取用来显示的 Address 地址，向 Address 中写入数据后就可以显示图像了。

6.3 实现 Overlay 硬件抽象层

在前面的内容中，了解了 Overlay 系统的基本知识和硬件抽象层的原理。在接下来的内容中，将详细讲解实现 Overlay 硬件抽象层的框架的基本知识，为进入本书后面知识的学习打下基础。

在 Android 系统中，提供了一个 Overlay 硬件抽象层的框架实现，在里面有完整的实现代码，我们可以将其作为使用 Overlay 硬件抽象层的方法。但是在里面没有使用具体硬件，所以不会有实际的现实效果。上述框架实现的的源码目录如下所示。

```
hardware/libhardware/modules/overlay/
```

在上述目录中，主要包含了文件 Android.mk 和 overlay.cpp，其中文件 Android.mk 的主要代码如下所示。

```
LOCAL_PATH := $(call my-dir)

# HAL module implemenation, not prelinked and stored in
# hw/<OVERLAY_HARDWARE_MODULE_ID>.<ro.product.board>.so
include $(CLEAR_VARS)
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := overlay.cpp
LOCAL_MODULE := overlay.trout
include $(BUILD_SHARED_LIBRARY)
```

Overlay 库是一个 C 语言库，没有被其他库所链接，在使用时是被动打开的。所以它必须被放置在目标文件系统的“system/lib/hw”目录中。

文件 overlay.cpp 的主要代码如下所示。

```
//此结构体用于扩充 overlay_control_device_t 结构体
struct overlay_control_context_t {
    struct overlay_control_device_t device;
    /* our private state goes below here */
};
//此结构体用于扩充 overlay_data_device_t 结构体
```

```
struct overlay_data_context_t {
    struct overlay_data_device_t device;
    /* our private state goes below here */
};

//定义打开函数
static int overlay_device_open(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device);

static struct hw_module_methods_t overlay_module_methods = {
    open: overlay_device_open
};
struct overlay_module_t HAL_MODULE_INFO_SYM = {
    common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 0,
        id: OVERLAY_HARDWARE_MODULE_ID,
        name: "Sample Overlay module",
        author: "The Android Open Source Project",
        methods: &overlay_module_methods,
    }
static int overlay_device_open(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device)
{
    int status = -EINVAL;
    if (!strcmp(name, OVERLAY_HARDWARE_CONTROL)) {          //Overlay的控制设备
        struct overlay_control_context_t *dev;
        dev = (overlay_control_context_t*)malloc(sizeof(*dev));

        /* initialize our state here */
        memset(dev, 0, sizeof(*dev));                       //初始化结构体

        /* initialize the procs */
        dev->device.common.tag = HARDWARE_DEVICE_TAG;
        dev->device.common.version = 0;
        dev->device.common.module = const_cast<hw_module_t*>(module);
        dev->device.common.close = overlay_control_close;

        dev->device.get = overlay_get;
        dev->device.createOverlay = overlay_createOverlay;
        dev->device.destroyOverlay = overlay_destroyOverlay;
        dev->device.setPosition = overlay_setPosition;
        dev->device.getPosition = overlay_getPosition;
        dev->device.setParameter = overlay_setParameter;

        *device = &dev->device.common;
        status = 0;
    } else if (!strcmp(name, OVERLAY_HARDWARE_DATA)) { //Overlay的数据设备
        struct overlay_data_context_t *dev;
        dev = (overlay_data_context_t*)malloc(sizeof(*dev));

        /* initialize our state here */
        memset(dev, 0, sizeof(*dev));                   //初始化结构体

        /* initialize the procs */
        dev->device.common.tag = HARDWARE_DEVICE_TAG;
        dev->device.common.version = 0;
        dev->device.common.module = const_cast<hw_module_t*>(module);
        dev->device.common.close = overlay_data_close;

        dev->device.initialize = overlay_initialize;
        dev->device.dequeueBuffer = overlay_dequeueBuffer;
        dev->device.queueBuffer = overlay_queueBuffer;
        dev->device.getBufferAddress = overlay_getBufferAddress;

        *device = &dev->device.common;
        status = 0;
    }
    return status;
}
```

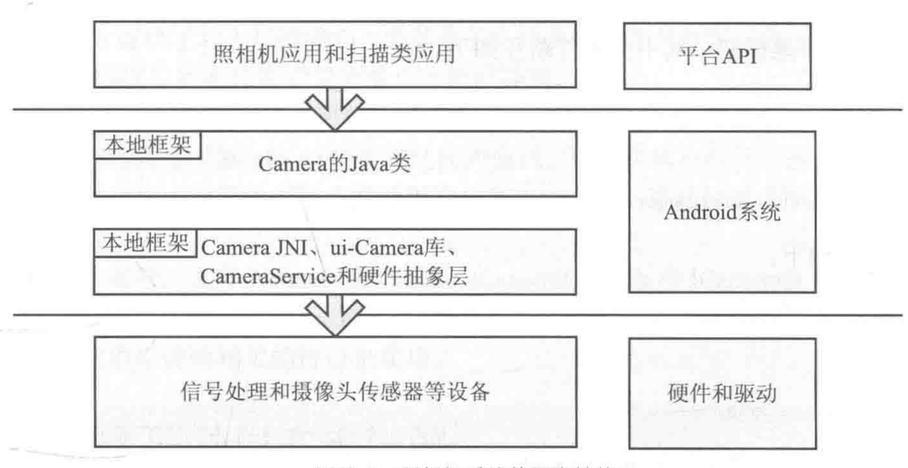
第 7 章　照相机系统

在当前的智能手机应用中，几乎所有的手机都具有手机拍照和录制视频功能。在 Android 系统中，照相机功能是通过 Camera 系统实现的。在本章的内容中，将详细讲解 Android 平台中 Camera 系统的基本知识，为进入本书后面知识的学习打下基础。

7.1 Camera 系统的结构

在 Android 系统中，Camera 照相机系统提供了取景器、视频录制和拍摄相片等功能，并且还具有各种控制类的接口。另外，在 Camera 系统中还提供了 Java 层的接口和本地接口。其中 Java 框架中的类 Camera 实现了 Java 层相机接口，为照相机类和扫描类使用。而 Camera 的本地接口可以给本地程序调用，作为视频输入环节应用于摄像机和视频电话领域。

Android 照相机系统的基本层次结构如图 7-1 所示。



▲图 7-1　照相机系统的层次结构

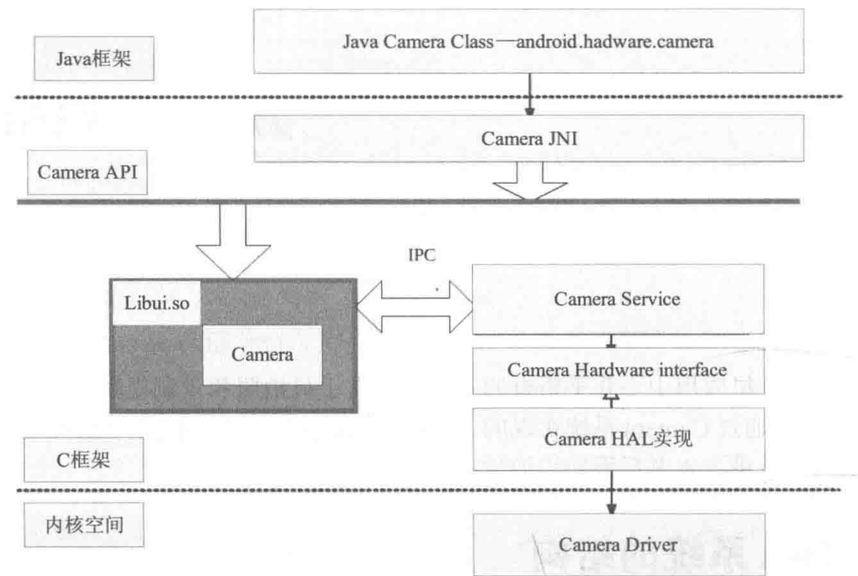
Android 系统中的 Camera 系统包括了 Camera 驱动程序层、Camera 硬件抽象层、AudioService、Camera 本地库、Camera 的 Java 框架类和 Java 应用层对 Camera 系统的调用。Camera 系统的具体结构如图 7-2 所示。

图 7-2 中各个构成层次的具体说明如下所示。

（1）Camera 系统的 Java 层，代码路径如下。

```
frameworks/base/core/java/android/hardware/
```

其中文件 Camera.java 是主要实现的文件，对应的 Java 层次的类是 android.hardware.Camera，这个类是和 JNI 中定义的类是同一个，有些方法通过 JNI 的方式调用本地代码得到，有些方法自己实现。



▲图 7-2　Camera 系统的结构

（2）Camera 系统的 Java 本地调用部分（JNI），代码路径如下。

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

这部分内容编译成为目标文件 libandroid_runtime.so，主要的头文件在以下的目录中。

```
frameworks/base/include/ui/
```

（3）Camera 本地框架，其中头文件路径如下。

```
frameworks/native/include/ui
```

或：

```
frameworks/av/include/camera/
```

源代码路径如下。

```
frameworks/native/libs/ui
```

或：

```
frameworks/av/camera/
```

这部分的内容被编译成库 libui.so 或 libcamera_client.so。

（4）Camera 服务部分，代码路径如下。

```
frameworks/av/services/camera/libcameraservice/
```

这部分内容被编译成库 libcameraservice.so。

为了实现一个具体功能的 Camera 驱动程序，在最底层还需要一个硬件相关的 Camer 库（例如通过调用 video for linux 驱动程序和 Jpeg 编码程序实现）。这个库将被 Camera 的服务库 libcameraservice.so 调用。

（5）摄像头驱动程序。

摄像头驱动程序部分是基于 Linux 的 Video for Linux 视频驱动框架。

（6）硬件抽象层。

硬件抽象层中的接口代码路径如下。

```
frameworks/base/include/ui/
```

或：

```
frameworks/av/include/camera/
```

其中的核心文件是 CameraHardwareInterface.h。

在 Camera 系统的各个库中，库 libui.so 位于核心的位置，它对上层提供的接口主要是 Camera 类，类 libandroid_runtime.so 通过调用 Camera 类提供对 Java 的接口，并且实现了 android.hardware.camera 类。

库 libcameraservice.so 是 Camera 的服务器程序，它通过继承 libui.so 的类实现服务器的功能，并且与 libui.so 中的另外一部分内容则通过进程间通讯（即 Binder 机制）的方式进行通讯。

库 libandroid_runtime.so 和 libui.so 是公用库，在里面除了 Camera 外还有其他方面的功能。

Camera 部分的头文件被保存在"frameworks/base/include/ui/"目录中，此目录是和库 libmedia.so 的源文件目录"frameworks/base/libs/ui/"相对应的。

在 Camera 中主要包含如下头文件。

- ICameraClient.h；
- Camera.h；
- ICamera.h；
- ICameraService.h；
- CameraHardwareInterface.h。

文件 Camera.h 提供了对上层的接口，而其他的几个头文件都是提供一些接口类（即包含了纯虚函数的类），这些接口类必须被实现类继承才能够使用。

当整个 Camera 在运行的时候，可以大致上分成 Client 和 Server 两个部分，它们分别在两个进程中运行，它们之间使用 Binder 机制实现进程间通讯。这样在客户端调用接口，在服务器中实现功能，但是在客户端中调用就好像直接调用服务器中的功能，进程间通讯的部分对上层程序不可见。

从框架结构上来看，文件 ICameraService.h、ICameraClient.h 和 ICamera.h 中的 3 个类定义了 Camera 的接口和架构，ICameraService.cpp 和 Camera.cpp 两个文件用于实现 Camera 架构，Camera 的具体功能在下层调用硬件相关的接口来实现。

7.2 Camera 驱动层实现详解

经过本章前面内容的讲解，已经了解了 Android 系统中 Camera 驱动系统的基本结构，并分析了在移植工作中需要移植的任务。在本节的内容中，将详细讲解在 Android 平台中 Camera 系统驱动层的具体实现过程。

7.2.1 V4L2 驱动程序

在 Linux 系统中，Camera 驱动程序使用了 Linux 标准的 Video for Linux 2（V4L2）驱动程序。无论是内核空间还是用户空间，都使用 V4L2 驱动程序框架来定义数据类和控制类。所以在移植 Android 中的 Camera 系统时，也是用标准的 V4L2 驱动程序作为 Camera 的驱动程序。在 Camera

系统中，V4L2 驱动程序的任务是获得 Video 数据。

1. V4L2 API

V4L2 是 V4L 的升级版本，为 linux 下视频设备程序提供了一套接口规范，包括一套数据结构和底层 V4L2 驱动接口。V4L2 驱动程序向用户空间提供字符设备，主设备号是 81。对于视频设备来说，次设备号是 0～63。如果次设备号在 64～127 之间的是 Radio 设备，次设备号在 192～223 之间的是 Teletext 设备，次设备号在 27～255 之间的是 VBI 设备。

V4L2 中常用的结构体在内核文件“include/linux/videodev2.h”中定义，代码如下所示。

```
struct v4l2_requestbuffers          //申请帧缓冲，对应命令 VIDIOC_REQBUFS
       struct v4l2_capability       //视频设备的功能，对应命令 VIDIOC_QUERYCAP
       struct v4l2_input            //视频输入信息，对应命令 VIDIOC_ENUMINPUT
       struct v4l2_standard         //视频的制式，比如 PAL，NTSC，对应命令 VIDIOC_ENUMSTD
       struct v4l2_format           //帧的格式，对应命令 VIDIOC_G_FMT、VIDIOC_S_FMT 等
       struct v4l2_buffer           //驱动中的一帧图像缓存，对应命令 VIDIOC_QUERYBUF
       struct v4l2_crop             //视频信号矩形边框
       v4l2_std_id                  //视频制式
```

常用的 ioctl 接口命令也在文件“include/linux/videodev2.h”中定义，代码如下所示。

```
VIDIOC_REQBUFS //分配内存
       VIDIOC_QUERYBUF          //把 VIDIOC_REQBUFS 中分配的数据缓存转换成物理地址
       VIDIOC_QUERYCAP          //查询驱动功能
       VIDIOC_ENUM_FMT          //获取当前驱动支持的视频格式
       VIDIOC_S_FMT             //设置当前驱动的频捕获格式
       VIDIOC_G_FMT             //读取当前驱动的频捕获格式
       VIDIOC_TRY_FMT           //验证当前驱动的显示格式
       VIDIOC_CROPCAP           //查询驱动的修剪能力
       VIDIOC_S_CROP            //设置视频信号的矩形边框
       VIDIOC_G_CROP            //读取视频信号的矩形边框
       VIDIOC_QBUF              //把数据从缓存中读取出来
       VIDIOC_DQBUF             //把数据放回缓存队列
       VIDIOC_STREAMON          //开始视频显示函数
       VIDIOC_STREAMOFF         //结束视频显示函数
       VIDIOC_QUERYSTD          //检查当前视频设备支持的标准，例如 PAL 或 NTSC
```

2. 操作 V4L2 的流程

在 V4L2 中提供了很多访问接口，我们可以根据具体需要选择操作方法。不过需要注意的是，很少有驱动完全实现了所有的接口功能。所以在使用时需要参考驱动源码，或仔细阅读驱动提供者的使用说明。接下来简单列举出一种 V4L2 的操作流程供读者朋友们参考。

（1）打开设备文件，具体代码如下所示。

```
int fd = open(Devicename,mode);
Devicename：/dev/video0、/dev/video1 ……
Mode：O_RDWR [| O_NONBLOCK]
```

如果需要使用非阻塞模式调用视频设备，当没有可用的视频数据时不会阻塞，而会立刻返回。

（2）获取设备的 capability，具体代码如下所示。

```
struct v4l2_capability capability;
int ret = ioctl(fd, VIDIOC_QUERYCAP, &capability);
```

在此需要查看设备具有什么功能，如是否具有视频输入特性。

（3）选择视频输入，代码如下。

```
struct v4l2_input input;
//……开始初始化 input
int ret = ioctl(fd, VIDIOC_QUERYCAP, &input);
```

每一个视频设备可以有多个视频输入，如果只有一路输入，则可以没有这个功能。

（4）检测视频支持的制式，具体代码如下所示。

```
v4l2_std_id std;
do {
            ret = ioctl(fd, VIDIOC_QUERYSTD, &std);
 } while (ret == -1 && errno == EAGAIN);
              switch (std) {
              case V4L2_STD_NTSC:
              case V4L2_STD_PAL:
              }
```

（5）设置视频捕获格式，具体代码如下所示。

```
struct v4l2_format fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = height;
fmt.fmt.pix.width = width;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if(ret) {
       perror("VIDIOC_S_FMT\n");
       close(fd);
       return -1;
}
```

（6）向驱动申请帧缓存，具体代码如下所示。

```
struct v4l2_requestbuffers req;
 if (ioctl(fd, VIDIOC_REQBUFS, &req) == -1) {
       return -1;
 }
```

在结构 v4l2_requestbuffers 中定义了缓存的数量，驱动会根据这个申请对应数量的视频缓存。通过多个缓存可以建立 FIFO，这样可以提高视频采集的效率。

（7）获取每个缓存的信息，并 mmap 到用户空间。具体代码如下所示。

```
typedef struct VideoBuffer {
       void *start;
       size_t length;
} VideoBuffer;
         VideoBuffer* buffers = calloc( req.count, sizeof(*buffers) );
struct v4l2_buffer buf;
         for (numBufs = 0; numBufs < req.count; numBufs++) {//映射所有的缓存
       memset( &buf, 0, sizeof(buf) );
       buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
       buf.memory = V4L2_MEMORY_MMAP;
       buf.index = numBufs;
       if (ioctl(fd, VIDIOC_QUERYBUF, &buf) == -1) {//获取到对应 index 的缓存信息，此处主要
利用 length 信息及 offset 信息来完成后面的 mmap 操作。
              return -1;
       }
buffers[numBufs].length = buf.length;
       // 转换成相对地址
       buffers[numBufs].start = mmap(NULL, buf.length,
              PROT_READ | PROT_WRITE,
              MAP_SHARED,
              fd, buf.m.offset);
 if (buffers[numBufs].start == MAP_FAILED) {
              return -1;
       }
```

（8）开始采集视频，具体代码如下所示。

```
int buf_type= V4L2_BUF_TYPE_VIDEO_CAPTURE;
int ret = ioctl(fd, VIDIOC_STREAMON, &buf_type);
```

（9）取出 FIFO 缓存中已经采样的帧缓存，具体代码如下所示。

```
struct v4l2_buffer buf;
memset(&buf,0,sizeof(buf));
buf.type=V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory=V4L2_MEMORY_MMAP;
buf.index=0;//此值由下面的 ioctl 返回
if (ioctl(fd, VIDIOC_DQBUF, &buf) == -1)
{
       return -1;
}
```

通过上述代码，可以根据返回的 buf.index 找到对应的 mmap 映射好的缓存，实现取出视频数据的功能。

（10）将刚刚处理完的缓冲重新入队列尾，这样可以循环采集，具体代码如下所示。

```
if (ioctl(fd, VIDIOC_QBUF, &buf) == -1) {
       return -1;
}
```

（11）停止视频的采集，具体代码如下所示。

```
int ret = ioctl(fd, VIDIOC_STREAMOFF, &buf_type);
```

（12）关闭视频设备，具体代码如下所示。

```
close(fd);
```

3. V4L2 驱动框架

在上述使用 V4L2 的流程中，各个操作都需要有底层 V4L2 驱动的支持。在内核中有一些非常完善的例子。例如在 Linux-2.6.26 内核目录“/drivers/media/video//zc301/”中，文件 zc301_core.c 实现了 ZC301 视频驱动代码。

（1）V4L2 驱动注册、注销函数。

在 Video 核心层文件“drivers/media/video/videodev.c”中提供了注册函数，具体代码如下所示。

```
int video_register_device(struct video_device *vfd, int type, int nr)
```

- video_device：要构建的核心数据结构；
- Type：表示设备类型，此设备号的基地址受此变量的影响；
- Nr：如果 end-base>nr>0，次设备号=base（基准值，受 type 影响）+nr；否则将系统自动分配合适的次设备号。

我们具体需要的驱动只需构建 video_device 结构，然后调用注册函数即可。例如在文件 zc301_core.c 中的实现代码如下所示。

```
err = video_register_device(cam->v4ldev, VFL_TYPE_GRABBER,
                            video_nr[dev_nr]);
```

在 Video 核心层文件“drivers/media/video/videodev.c”中提供了如下注销函数。

```
void video_unregister_device(struct video_device *vfd)
```

（2）构建 struct video_device。

在结构 video_device 中包含了视频设备的属性和操作方法，具体可以参考文件 zc301_core.c，代码如下所示。

```
strcpy(cam->v4ldev->name, "ZC0301[P] PC Camera");
```

```
cam->v4ldev->owner = THIS_MODULE;
cam->v4ldev->type = VID_TYPE_CAPTURE | VID_TYPE_SCALES;
cam->v4ldev->fops = &zc0301_fops;
cam->v4ldev->minor = video_nr[dev_nr];
cam->v4ldev->release = video_device_release;
video_set_drvdata(cam->v4ldev, cam);
```

在上述 zc301 的驱动中并没有实现 struct video_device 中的很多操作函数，例如 vidioc_querycap、vidioc_g_fmt_cap，这是因为在 struct file_operations zc0301_fops 中的 zc0301_ioctl 实现了前面的所有 ioctl 操作，所以无需在 struct video_device 再次实现 struct video_device 中的操作。

另外也可以使用下面的代码来构建 struct video_device。

```
static struct video_device camif_dev =
  {
    .name = "s3c2440 camif",
    .type = VID_TYPE_CAPTURE|VID_TYPE_SCALES|VID_TYPE_SUBCAPTURE,
    .fops = &camif_fops,
    .minor = -1,
    .release = camif_dev_release,
    .vidioc_querycap = vidioc_querycap,
    .vidioc_enum_fmt_cap = vidioc_enum_fmt_cap,
    .vidioc_g_fmt_cap = vidioc_g_fmt_cap,
    .vidioc_s_fmt_cap = vidioc_s_fmt_cap,
    .vidioc_queryctrl = vidioc_queryctrl,
    .vidioc_g_ctrl = vidioc_g_ctrl,
    .vidioc_s_ctrl = vidioc_s_ctrl,
  };
  static struct file_operations camif_fops =
  {
    .owner = THIS_MODULE,
    .open = camif_open,
    .release = camif_release,
    .read = camif_read,
    .poll = camif_poll,
    .ioctl = video_ioctl2, /* V4L2 ioctl handler */
    .mmap = camif_mmap,
    .llseek = no_llseek,
  };
```

结构 video_ioctl2 是在文件 videodev.c 中是实现的，video_ioctl2 中会根据 ioctl 不同的 cmd 来调用 video_device 中的操作方法。

4. 实现 Video 核心层

具体实现代码请参考内核文件“/drivers/media/videodev.c”，实现流程如下所示。

（1）注册 256 个视频设备，代码如下所示。

```
static int __init videodev_init(void)
       {
               int ret;
               if (register_chrdev(VIDEO_MAJOR, VIDEO_NAME, &video_fops)) {
                       return -EIO;
               }
               ret = class_register(&video_class);
               ……
       }
```

在上述代码中注册了 256 个视频设备和 video_class 类，video_fops 为这 256 个设备共同的操作方法。

（2）实现 V4L2 驱动的注册函数，具体代码如下所示。

```
int video_register_device(struct video_device *vfd, int type, int nr)
       {
```

```
            int i=0;
            int base;
            int end;
            int ret;
            char *name_base;
      switch(type) //根据不同的 type 确定设备名称、次设备号
            {
                   case VFL_TYPE_GRABBER:
                          base=MINOR_VFL_TYPE_GRABBER_MIN;
                          end=MINOR_VFL_TYPE_GRABBER_MAX+1;
                          name_base = "video";
                          break;
                   case VFL_TYPE_VTX:
                          base=MINOR_VFL_TYPE_VTX_MIN;
                          end=MINOR_VFL_TYPE_VTX_MAX+1;
                          name_base = "vtx";
                          break;
                   case VFL_TYPE_VBI:
                          base=MINOR_VFL_TYPE_VBI_MIN;
                          end=MINOR_VFL_TYPE_VBI_MAX+1;
                          name_base = "vbi";
                          break;
                   case VFL_TYPE_RADIO:
                          base=MINOR_VFL_TYPE_RADIO_MIN;
                          end=MINOR_VFL_TYPE_RADIO_MAX+1;
                          name_base = "radio";
                          break;
                   default:
                          printk(KERN_ERR "%s called with unknown type: %d\n",
                                 __func__, type);
                          return -1;
            }
      /* 计算出次设备号 */
            mutex_lock(&videodev_lock);
            if (nr >= 0 && nr < end-base) {
                   /* use the one the driver asked for */
                   i = base+nr;
                   if (NULL != video_device[i]) {
                          mutex_unlock(&videodev_lock);
                          return -ENFILE;
                   }
            } else {
                   /* use first free */
                   for(i=base;i<end;i++)
                          if (NULL == video_device[i])
                                 break;
                   if (i == end) {
                          mutex_unlock(&videodev_lock);
                          return -ENFILE;
                   }
            }
          video_device[i]=vfd; //保存 video_device 结构指针到系统的结构数组中，最终的次设备
号和 i 相关
          vfd->minor=i;
          mutex_unlock(&videodev_lock);
          mutex_init(&vfd->lock);
      /* sysfs class */
            memset(&vfd->class_dev, 0x00, sizeof(vfd->class_dev));
            if (vfd->dev)
                   vfd->class_dev.parent = vfd->dev;
         vfd->class_dev.class = &video_class;
         vfd->class_dev.devt = MKDEV(VIDEO_MAJOR, vfd->minor);
         sprintf(vfd->class_dev.bus_id, "%s%d", name_base, i - base);//最后在/dev 目
录下的名称
    ret = device_register(&vfd->class_dev);//结合 udev 或 mdev 可以实现自动在/dev 下创建设备
节点
            ......
      }
```

从上面的注册函数代码中可以看出，注册 V4L2 驱动的过程只是创建了设备节点，例如“/dev/video0”。并且保存了 video_device 结构指针。

（3）打开视频驱动。

使用下面的代码在用户空间调用 open()函数打开对应的视频文件。

```
int fd = open(/dev/video0, O_RDWR);
```

此时，对应“/dev/video0”目录的文件操作结构是在文件“/drivers/media/videodev.c”中定义的 video_fops。代码如下所示。

```
static const struct file_operations video_fops=
       {
              .owner = THIS_MODULE,
              .llseek = no_llseek,
              .open = video_open,
       };
```

上述代码只是实现了 open 操作，后面的其他操作需要使用 video_open()来实现，具体代码如下所示。

```
static int video_open(struct inode *inode, struct file *file)
       {
              unsigned int minor = iminor(inode);
              int err = 0;
              struct video_device *vfl;
              const struct file_operations *old_fops;
       if(minor>=VIDEO_NUM_DEVICES)
                     return -ENODEV;
              mutex_lock(&videodev_lock);
              vfl=video_device[minor];
              if(vfl==NULL) {
                     mutex_unlock(&videodev_lock);
                     request_module("char-major-%d-%d", VIDEO_MAJOR, minor);
                     mutex_lock(&videodev_lock);
                     vfl=video_device[minor]; //根据次设备号取出 video_device 结构
                     if (vfl==NULL) {
                            mutex_unlock(&videodev_lock);
                            return -ENODEV;
                     }
              }
              old_fops = file->f_op;
              file->f_op = fops_get(vfl->fops);//替换此打开文件的 file_operation 结构。后面
的其他针对此文件的操作都由新的结构来负责。也就是由每个具体的 video_device 的 fops 负责
              if(file->f_op->open)
                     err = file->f_op->open(inode,file);
              if (err) {
                     fops_put(file->f_op);
                     file->f_op = fops_get(old_fops);
              }
              ......
       }
```

7.2.2 硬件抽象层

在 Andorid 2.1 及其以前的版本中，Camera 系统的硬件抽象层的头文件保存在如下目录中。

```
frameworks/base/include/ui/
```

在 Andorid 2.2 及其以后的版本中，Camera 系统的硬件抽象层的头文件保存在如下目录中。

```
frameworks/av/include/camera/
```

在上述目录中主要包含了如下头文件。

- CameraHardwareInterface.h：在里面定义了 C++接口类，此类需要根据系统的情况来实现继承。
- CameraParameters.h：在里面定义了 Camera 系统的参数，可以在本地系统的各个层次中使用这些参数；
- Camera.h：在里面提供了 Camera 系统本地对上层的接口。

1. Andorid 2.1 及其以前的版本

在 Andorid 2.1 及其以前的版本中，在文件 CameraHardwareInterface.h 中首先定义了硬件抽象层接口的回调函数类型，对应代码如下所示。

```
/** startPreview()使用的回调函数*/
typedef void (*preview_callback)(const sp<IMemory>& mem, void* user);

/** startRecord()使用的回调函数*/
typedef void (*recording_callback)(const sp<IMemory>& mem, void* user);

/** takePicture()使用的回调函数*/
typedef void (*shutter_callback)(void* user);

/** takePicture()使用的回调函数*/
typedef void (*raw_callback)(const sp<IMemory>& mem, void* user);

/** takePicture()使用的回调函数*/
typedef void (*jpeg_callback)(const sp<IMemory>& mem, void* user);

/** autoFocus()使用的回调函数*/
typedef void (*autofocus_callback)(bool focused, void* user);
```

然后定义类 CameraHardwareInterface，并在类中定义了各个接口函数。具体代码如下所示。

```
class CameraHardwareInterface : public virtual RefBase {
public:
    virtual ~CameraHardwareInterface() { }
    virtual sp<IMemoryHeap>         getPreviewHeap() const = 0;
    virtual sp<IMemoryHeap>         getRawHeap() const = 0;
    virtual status_t    startPreview(preview_callback cb, void* user) = 0;
    virtual bool useOverlay() {return false;}
    virtual status_t setOverlay(const sp<Overlay> &overlay) {return BAD_VALUE;}
    virtual void        stopPreview() = 0;
    virtual bool        previewEnabled() = 0;
    virtual status_t    startRecording(recording_callback cb, void* user) = 0;
    virtual void        stopRecording() = 0;
    virtual bool        recordingEnabled() = 0;
    virtual void        releaseRecordingFrame(const sp<IMemory>& mem) = 0;
    virtual status_t    autoFocus(autofocus_callback,
                            void* user) = 0;
    virtual status_t    takePicture(shutter_callback,
                              raw_callback,
                              jpeg_callback,
                              void* user) = 0;
    virtual status_t    cancelPicture(bool cancel_shutter,
                                bool cancel_raw,
                                bool cancel_jpeg) = 0;
    /** Return the camera parameters. */
    virtual CameraParameters  getParameters() const = 0;
    virtual void release() = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) const = 0;
};
extern "C" sp<CameraHardwareInterface> openCameraHardware();
};
```

可以将上述代码中的接口分为如下几类。

- 取景预览：startPreview、stopPreview、useOverlay 和 setOverlay；
- 录制视频：startRecording、stopRecording、recordingEnabled 和 releaseRecordingFrame；
- 拍摄照片：takePicture 和 cancelPicture；
- 辅助功能：autoFocus（自动对焦）、setParameters 和 getParameters。

2. Andorid 2.2 及其以后的版本

在 Andorid 2.2 及其以前的版本中，在文件 Camera.h 中首先定义了通知信息的枚举值，对应代码如下所示。

```
enum {
   CAMERA_MSG_ERROR              = 0x001,   //错误信息
   CAMERA_MSG_SHUTTER             = 0x002,  //快门信息
   CAMERA_MSG_FOCUS              = 0x004,  //聚焦信息
   CAMERA_MSG_ZOOM              = 0x008,  //缩放信息
   CAMERA_MSG_PREVIEW_FRAME       = 0x010,  //帧预览信息
   CAMERA_MSG_VIDEO_FRAME        = 0x020,  //视频帧信息
   CAMERA_MSG_POSTVIEW_FRAME      = 0x040,  //拍照后停止帧信息
   CAMERA_MSG_RAW_IMAGE          = 0x080,  //原始数据格式照片信息
   CAMERA_MSG_COMPRESSED_IMAGE = 0x100,  //压缩格式照片信息
   CAMERA_MSG_ALL_MSGS           = 0x1FF  //所有信息
};
```

然后在文件 CameraHardwareInterface.h 中定义如下 3 个回调函数。

```
//通知回调
typedef void (*notify_callback)(int32_t msgType,
                                int32_t ext1,
                                int32_t ext2,
                                void* user);
//数据回调
typedef void (*data_callback)(int32_t msgType,
                              const sp<IMemory>& dataPtr,
                              void* user);
//带有时间戳的数据回调
typedef void (*data_callback_timestamp)(nsecs_t timestamp,
                                        int32_t msgType,
                                        const sp<IMemory>& dataPtr,
                                        void* user);
```

然后定义类 CameraHardwareInterface。在类中的各个函数的具体实现和其他 Android 版本中的相同，区别是回调函数不再由各个函数分别设置，所以在 startPreview 和 startRecording 缺少了回调函数的指针和 void*类型的附加参数。其主要实现代码如下所示。

```
class CameraHardwareInterface : public virtual RefBase {
public:
   virtual ~CameraHardwareInterface() { }
   virtual sp<IMemoryHeap>        getPreviewHeap() const = 0;
   virtual sp<IMemoryHeap>        getRawHeap() const = 0;
   virtual void setCallbacks(notify_callback notify_cb,
                             data_callback data_cb,
                             data_callback_timestamp data_cb_timestamp,
   virtual void        enableMsgType(int32_t msgType) = 0;
   virtual void        disableMsgType(int32_t msgType) = 0;
   virtual bool        msgTypeEnabled(int32_t msgType) = 0;
   virtual status_t     startPreview() = 0;
   virtual status_t     getBufferInfo(sp<IMemory>& Frame, size_t *alignedSize) = 0;
   virtual bool         useOverlay() {return false;}
   virtual status_t      setOverlay(const sp<Overlay> &overlay) {return BAD_VALUE;}
   virtual void        stopPreview() = 0;
   virtual bool        previewEnabled() = 0;
   virtual status_t     startRecording() = 0;
   virtual void        stopRecording() = 0;
```

```
    virtual bool       recordingEnabled() = 0;
    virtual void       releaseRecordingFrame(const sp<IMemory>& mem) = 0;
    virtual status_t    autoFocus() = 0;
    virtual status_t    cancelAutoFocus() = 0;
    virtual status_t    takePicture() = 0;
    virtual status_t    cancelPicture() = 0;
    virtual CameraParameters  getParameters() const = 0;
    virtual status_t sendCommand(int32_t cmd, int32_t arg1, int32_t arg2) = 0;
    virtual void release() = 0;
    virtual status_t d
}
```

因为在新版本的 Camera 系统中增加了函数 sendCommand()，所以需要在文件 Camera.h 中增加新命令和返回值。具体实现代码如下所示。

```
// 函数 sendCommand()使用的命令类型
enum {
    CAMERA_CMD_START_SMOOTH_ZOOM     = 1,
    CAMERA_CMD_STOP_SMOOTH_ZOOM      = 2,
    CAMERA_CMD_SET_DISPLAY_ORIENTATION = 3,
};

// 错误类型
enum {
    CAMERA_ERROR_UKNOWN  = 1,
    CAMERA_ERROR_SERVER_DIED = 100
};
```

3. 实现 Camera 硬件抽象层

在函数 startPreview 的实现过程中，保存预览回调函数并建立预览线程。在预览线程的循环中，等待视频数据的到达；视频帧到达后调用预览回调函数，将视频帧送出。

取景器预览的主要步骤如下所示。

（1）在初始化的过程中，建立预览数据的内存队列（多种方式）。

（2）在函数 startPreview 中建立预览线程。

（3）在预览线程的循环中，等待视频数据到达。

（4）视频到达后使用预览回调机制将视频向上传送。

在此过程中不需要使用预览回调函数，可以直接将视频数据输入到 Overlay 上。如果使用 Overlay 实现取景器，则需要有以下两个变化。

- 在函数 setOverlay()中，从 ISurface 接口中取得 Overlay 类。
- 在预览线程的循环中，不是用预览回调函数直接将数据输入到 Overlay 上。

录制视频的主要步骤如下所示。

（1）在函数 startRecording 的实现（或者在 setCallbacks）中保存录制视频回调函数。

（2）录制视频可以使用自己的线程，也可以使用预览线程。

（3）通过调用录制回调函数的方式将视频帧送出。

当调用函数 releaseRecordingFrame 后，表示上层通知 Camera 硬件抽象层这一帧的内存已经用完，可以进行下一次的处理。如果在 V4L2 驱动程序中使用原始数据（RAW），则视频录制的数据和取景器预览的数据为同一数据。当调用 releaseRecordingFrame()时，通常表示编码器已经完成了对当前视频帧的编码，并对这块内存进行释放。在这个函数的实现中，可以设置标志位，标记帧内存可以再次使用。

由此可见，对于 Linux 系统来说，摄像头驱动部分大多使用 Video for Linux 2（V4L2）驱动程序，在此处主要的处理流程可以如下所示。

（1）如果使用映射内核内存的方式（V4L2_MEMORY_MMAP），构建预览的内存 MemoryHeapBase

需要从 V4L2 驱动程序中得到内存指针。

（2）如果使用用户空间内存的方式（V4L2_MEMORY_USERPTR），MemoryHeapBase 中开辟的内存是在用户空间建立的。

（3）在预览的线程中，使用 VIDIOC_DQBUF 调用阻塞等待视频帧的到来，处理完成后使用 VIDIOC_QBUF 调用将帧内存再次压入队列，然后等待下一帧的到来。

7.3 实现 Camera 系统的硬件抽象层

在 Android 系统中已经实现了一个 Camera 硬件抽象层的“桩”，这样可以根据“宏”来配置。此“桩”使用假的方式实现取景器预览和照片拍摄功能。在 Camera 系统的“桩”实现中使用黑白格子代替来自硬件的视频流，这样可以在不接触硬件的情况下让 Camera 系统运行。因为没有视频输出设备，所以不会使用 Overlay 来实现 Camera 硬件抽象层的“桩”。在本节的内容中，将详细讲解实现 Camera 系统的硬件抽象层的具体过程。

7.3.1　Java 程序部分

在文件“packages/apps/Camera/src/com/android/camera/Camera.java”中，已经包含了对 Camera 的调用。在文件 Camera.java 中包含的对包的引用代码如下所示。

```
import android.hardware.Camera.PictureCallback;
import android.hardware.Camera.Size;
```

然后定义类 Camera，此类继承了 Activity 类，在它的内部包含了一个 android.hardware.Camera。对应代码如下所示。

```
public class Camera extends Activity implements View.OnClickListener, SurfaceHolder.
Callback{
android.hardware.Camera mCameraDevice;
}
```

调用 Camera 功能的代码如下所示。

```
mCameraDevice.takePicture(mShutterCallback, mRawPictureCallback, mJpegPicture
Callback);
mCameraDevice.startPreview();
mCameraDevice.stopPreview();
//startPreview、stopPreview 和 takePicture 等接口就是通过 JAVA 本地调用（JNI）来实现的
frameworks/base/core/java/android/hardware/
//目录中的 Camera.java 文件提供了一个 JAVA 类：Camera
public class Camera {
}
```

在类 Camera 中，大部分代码使用 JNI 调用下层得到，例如下面的代码。

```
public void setParameters(Parameters params) {
      Log.e(TAG, "setParameters()");
      //params.dump();
      native_setParameters(params.flatten());
}
```

还有下面的代码。

```
public final void setPreviewDisplay(SurfaceHolder holder) {
      setPreviewDisplay(holder.getSurface());
}
private native final void setPreviewDisplay(Surface surface);
```

在上面的两端代码中，两个 setPreviewDisplay 参数不同，后一个是本地方法，参数为 Surface 类型；前一个通过调用后一个实现，但自己的参数以 SurfaceHolder 为类型。

7.3.2　Camera 的 Java 本地调用部分

在 Android 系统中，Camera 驱动的 Java 本地调用（JNI）部分在如下文件中实现。

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

在文件 android_hardware_Camera.cpp 中定义了一个 JNINativeMethod（Java 本地调用方法）类型的数组 gMethods，具体代码如下所示。

```
static JNINativeMethod camMethods[] = {
{"native_setup","(Ljava/lang/Object;)V",(void*)android_hardware_Camera_native_setup
},
  {"native_release","()V",(void*)android_hardware_Camera_release },
  {"setPreviewDisplay","(Landroid/view/Surface;)V",(void
*)android_hardware_Camera_setPreviewDisplay },
  {"startPreview","()V",(void *)android_hardware_Camera_startPreview },
  {"stopPreview", "()V", (void *)android_hardware_Camera_stopPreview },
  {"setHasPreviewCallback","(Z)V",(void
*)android_hardware_Camera_setHasPreviewCallback },
  {"native_autoFocus","()V",(void *)android_hardware_Camera_autoFocus },
  {"native_takePicture", "()V", (void *)android_hardware_Camera_takePicture },
  {"native_setParameters","(Ljava/lang/String;)V",(void
*)android_hardware_Camera_setParameters },
  {"native_getParameters",  "()Ljava/lang/String;",(void  *)android_hardware_Camera_
getParameters }
};
```

JNINativeMethod 的第一个成员是一个字符串，表示 Java 本地调用方法的名称，此名称是在 Java 程序中调用的名称；第二个成员也是一个字符串，表示 Java 本地调用方法的参数和返回值；第三个成员是 Java 本地调用方法对应的 C 语言函数。

通过函数 register_android_hardware_Camera() 将 gMethods 注册为的类“android/media/Camera”，其主要实现如下所示。

```
int register_android_hardware_Camera(JNIEnv *env)
{
// Register native functions
return AndroidRuntime::registerNativeMethods(env, "android/hardware/Camera",
camMethods, NELEM(camMethods));
}
```

其中类“android/hardware/Camera”和 Java 类 android.hardware.Camera 相对应。

7.3.3　Camera 的本地库 libui.so

文件“frameworks/base/libs/ui/Camera.cpp”用于实现文件 Camera.h 中提供的接口，其中最重要的代码片段如下所示。

```
sp<Camera> Camera::create(const sp<ICamera>& camera)
{
     ALOGV("create");
     if (camera == 0) {
         ALOGE("camera remote is a NULL pointer");
         return 0;
     }

    sp<Camera> c = new Camera(-1);
    if (camera->connect(c) == NO_ERROR) {
        c->mStatus = NO_ERROR;
```

```
        c->mCamera = camera;
        camera->asBinder()->linkToDeath(c);
        return c;
    }
    return 0;
}
```

函数 connect()的实现代码如下所示。

```
sp<Camera> Camera::connect(int cameraId, const String16& clientPackageName,
        int clientUid)
{
    return CameraBaseT::connect(cameraId, clientPackageName, clientUid);
}
```

函数 connect()通过调用 getCameraService 得到一个 ICameraService，再通过 ICameraService 的 cs->connect(c)得到一个 ICamera 类型的指针。调用 connect()函数会得到一个 Camera 类型的指针。在正常情况下，已经初始化完成了 Camera 的成员 mCamera。

函数 startPreview()的实现代码如下所示。

```
status_t Camera::startPreview()
{
    ALOGV("startPreview");
    sp <ICamera> c = mCamera;
    if (c == 0) return NO_INIT;
    return c->startPreview();
}
```

其他函数的实现过程也与函数 setDataSource 类似。在库 libmedia.so 中的其他一些文件与头文件的名称相同，分别如下所示。

```
frameworks/base/libs/ui/ICameraClient.cpp
frameworks/base/libs/ui/ICamera.cpp
frameworks/base/libs/ui/ICameraService.cpp
```

此处的类 BnCameraClient 和 BnCameraService 虽然实现了 onTransact()函数，但是由于还有纯虚函数没有实现，所以不能实例化这个类。

7.3.4 Camera 服务 libcameraservice.so

目录“frameworks/av/services/camera/libcameraservice/”实现一个 Camera 的服务，此服务是继承 ICameraService 的具体实现。在此目录下和硬件抽象层“桩”实现相关的文件说明如下所示。

- CameraHardwareStub.cpp：Camera 硬件抽象层“桩”实现；
- CameraHardwareStub.h：Camera 硬件抽象层“桩”实现的接口；
- CannedJpeg.h：包含一块 Jpeg 数据，在拍照片时作为 Jpeg 数据；
- FakeCamera.h 和 FakeCamera.cpp：实现假的 Camera 黑白格取景器效果。

在文件 Android.mk 中，使用宏 USE_CAMERA_STUB 决定是否使用真的 Camera，如果宏为真，则使用 CameraHardwareStub.cpp 和 FakeCamera.cpp 构造一个假的 Camera；如果为假，则使用 CameraService.cpp 构造一个实际上的 Camera 服务。文件 Android.mk 的主要代码如下所示。

```
LOCAL_MODULE:= libcamerastub
LOCAL_SHARED_LIBRARIES:= libui
include $(BUILD_STATIC_LIBRARY)
endif # USE_CAMERA_STUB
#
# libcameraservice
#
```

```
include $(CLEAR_VARS)
LOCAL_SRC_FILES:=               \
   CameraService.cpp
LOCAL_SHARED_LIBRARIES:= \
   libui \
   libutils \
   libcutils \
   libmedia
LOCAL_MODULE:= libcameraservice
LOCAL_CFLAGS+=-DLOG_TAG=\"CameraService\"
ifeq ($(USE_CAMERA_STUB), true)
LOCAL_STATIC_LIBRARIES += libcamerastub
LOCAL_CFLAGS += -include CameraHardwareStub.h
else
LOCAL_SHARED_LIBRARIES += libcamera
endif
include $(BUILD_SHARED_LIBRARY)
```

文件 CameraService.cpp 继承了 BnCameraService 的实现，在此类内部又定义了类 Client，CameraService::Client 继承了 BnCamera。在运作的过程中，函数 CameraService::connect()用于得到一个 CameraService::Client。在使用过程中，主要是通过调用这个类的接口来实现完成 Camera 的功能。因为 CameraService::Client 本身继承了 BnCamera 类，而 BnCamera 类继承了 ICamera，所以可以将此类当成 ICamera 来使用。

类 CameraService 和 CameraService::Client 的结果如下所示。

```
class CameraService : public BnCameraService
{
class Client : public BnCamera {};
wp<Client>              mClient;
}
```

在 CameraService 中，静态函数 instantiate()用于初始化一个 Camera 服务，此函数的代码如下所示。

```
void CameraService::instantiate() {
defaultServiceManager()->addService( String16("media.camera"), new CameraService());
}
```

其实函数 CameraService::instantiate()注册了一个名称为“media.camera”的服务，此服务和文件 Camera.cpp 中调用的名称相对应。

Camera 整个运作机制是：在文件 Camera.cpp 中调用 ICameraService 的接口，此时实际上调用的是 BpCameraService。而 BpCameraService 通过 Binder 机制和 BnCameraService 实现两个进程的通讯。因为 BpCameraService 的实现就是此处的 CameraService，所以 Camera.cpp 虽然是在另外一个进程中运行，但是调用 ICameraService 的接口就像直接调用一样，从函数 connect()中可以得到一个 ICamera 类型的指针，整个指针的实现实际上是 CameraService::Client。

上述 Camera 功能就是 CameraService::Client 所实现的，其构造函数如下所示。

```
CameraService::Client::Client(const sp<CameraService>& cameraService,
      const sp<ICameraClient>& cameraClient) :
mCameraService(cameraService), mCameraClient(cameraClient), mHardware(0)
{
mHardware = openCameraHardware();
mHasFrameCallback = false;
}
```

在构造函数中，通过调用 openCameraHardware()得到一个 CameraHardwareInterface 类型的指针，并作为其成员 mHardware。以后对实际的 Camera 的操作都通过对这个指针进行，这是一个简单的直接调用关系。

其实真正的 Camera 功能已经通过实现 CameraHardwareInterface 类来完成。在这个库中，文件 CameraHardwareStub.h 和 CameraHardwareStub.cpp 定义了一个“桩”模块的接口，可以在没有 Camera 硬件的情况下使用。例如在仿真器的情况下使用就是文件 CameraHardwareStub.cpp 和它依赖的文件 FakeCamera.cpp。

类 CameraHardwareStub 的结构如下所示。

```
class CameraHardwareStub : public CameraHardwareInterface {
class PreviewThread : public Thread {
};
};
```

在类 CameraHardwareStub 中包含了线程类 PreviewThread，此线程可以处理 PreView，即负责刷新取景器的内容。实际的 Camera 硬件接口通常可以通过对 V4L2 捕获驱动的调用来实现，同时还需要一个 JPEG 编码程序将从驱动中取出的数据编码成 JPEG 文件。

在文件 FakeCamera.h 和 FakeCamera.cpp 中实现了类 FakeCamera，用于实现一个假的摄像头输入数据的内存。定义代码如下所示。

```
class FakeCamera {
public:
    FakeCamera(int width, int height);
    ~FakeCamera();

    void setSize(int width, int height);
    void getNextFrameAsRgb565(uint16_t *buffer);//获取 RGB565 格式的预览帧
    void getNextFrameAsYuv422(uint8_t *buffer);//获取 Yuv422 格式的预览帧
    status_t dump(int fd, const Vector<String16>& args);

private:
    void drawSquare(uint16_t *buffer, int x, int y, int size, int color, int shadow);
    void drawCheckerboard(uint16_t *buffer, int size);

    static const int kRed = 0xf800;
    static const int kGreen = 0x07c0;
    static const int kBlue = 0x003e;

    int         mWidth, mHeight;
    int         mCounter;
    int         mCheckX, mCheckY;
    uint16_t    *mTmpRgb16Buffer;
};
```

当在 CameraHardwareStub 中设置参数后会调用函数 initHeapLocked()，此函数的实现代码如下所示。

```
void CameraHardwareStub::initHeapLocked()
{
    int picture_width, picture_height;
    mParameters.getPictureSize(&picture_width, &picture_height);
    //建立内存堆栈，创建两块内存
    mRawHeap = new MemoryHeapBase(picture_width * 2 * picture_height);

    int preview_width, preview_height;
    mParameters.getPreviewSize(&preview_width, &preview_height);
    LOGD("initHeapLocked: preview size=%dx%d", preview_width, preview_height);

    // 从参数中获取信息
    int how_big = preview_width * preview_height * 2;

    // If we are being reinitialized to the same size as before, no
    // work needs to be done.
    if (how_big == mPreviewFrameSize)
        return;
```

```
    mPreviewFrameSize = how_big;

    // Make a new mmap'ed heap that can be shared across processes.
    // use code below to test with pmem
    mPreviewHeap = new MemoryHeapBase(mPreviewFrameSize * kBufferCount);
    // 建立内存队列
    for (int i = 0; i < kBufferCount; i++) {
        mBuffers[i] = new MemoryBase(mPreviewHeap, i * mPreviewFrameSize,
mPreviewFrameSize);
    }

    // Recreate the fake camera to reflect the current size.
    delete mFakeCamera;
    mFakeCamera = new FakeCamera(preview_width, preview_height);
}
```

定义函数startPrevie()来创建一个线程，此函数的实现代码如下所示。

```
status_t CameraHardwareStub::startPreview(preview_callback cb, void* user)
{
    Mutex::Autolock lock(mLock);
    if (mPreviewThread != 0) {
        // already running
        return INVALID_OPERATION;
    }
    mPreviewCallback = cb;
    mPreviewCallbackCookie = user;
    mPreviewThread = new PreviewThread(this);//建立视频预览线程
    return NO_ERROR;
}
```

通过上面建立的线程可以调用预览回调机制，将预览的数据传递给上层的CameraService。

创建预览线程函数previewThread，建立一个循环以得到假的摄像头输入数据的来源，并通过预览回调函数将输出传到上层中。函数previewThread的主要实现代码如下所示。

```
int CameraHardwareStub::previewThread()
{
    mLock.lock();
        int previewFrameRate = mParameters.getPreviewFrameRate();
         //发现在当前缓冲的堆的之内垂距
         ssize_t offset = mCurrentPreviewFrame * mPreviewFrameSize;
         sp<MemoryHeapBase> heap = mPreviewHeap;
             // 假设假照相机内部状态没有变化
         // (or is thread safe)
         FakeCamera* fakeCamera = mFakeCamera;
         sp<MemoryBase> buffer = mBuffers[mCurrentPreviewFrame];
             mLock.unlock();
    if (buffer != 0) {
        //多久计算等待在框架之间
        int delay = (int)(1000000.0f / float(previewFrameRate));
            //这总是合法的，即使内存消亡，仍然在我们的过程中被映射
           void *base = heap->base();
           //用假照相机填装当前框架.
        uint8_t *frame = ((uint8_t *)base) + offset;
        fakeCamera->getNextFrameAsYuv422(frame);

                // Notify the client of a new frame.
        mPreviewCallback(buffer, mPreviewCallbackCookie);
            //推进缓冲尖
        mCurrentPreviewFrame = (mCurrentPreviewFrame + 1) % kBufferCount;
        //等待它
        usleep(delay);
    }
    return NO_ERROR;
}
```

在上述文件中还定义了其他的函数，函数的功能一看便知，在此为节省篇幅将不再进行详细讲解，请读者参考开源的代码文件。

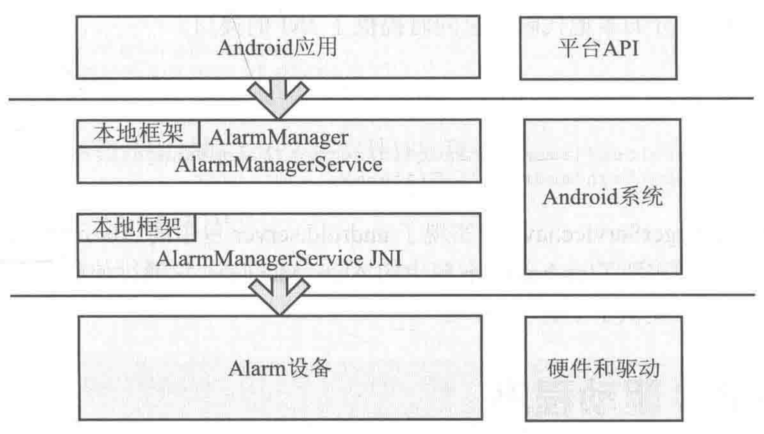
第8章　Alarm时钟系统

在 Android 系统中，Alarm 是一个硬件时钟，也被称为警报器系统，它提供了一个定时器，用于把设备从睡眠状态唤醒，同时它也提供了一个在设备睡眠时仍然会运行的时钟基准。在本章的内容中，将和大家一起探讨 Android 系统中的时钟系统驱动 Alarm 的基本知识，分析其具体原理和实现源码，为读者进入本书后面知识的学习打下基础。

8.1 Alarm 系统基础

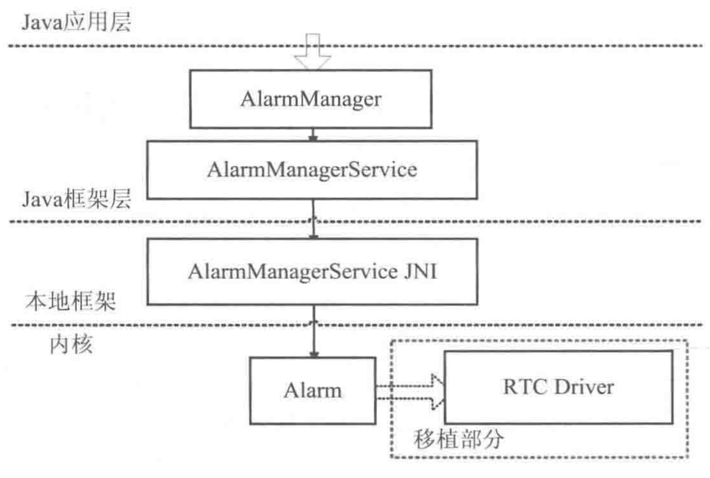
在 Android 系统中，警报器系统又叫时钟系统或闹钟系统。Alarm 闹钟是 Android 系统中在标准 RTC 驱动上开发的一个新的驱动，提供了一个定时器，用于把设备从睡眠状态唤醒，因为它是依赖 RTC 驱动的，所以它同时为系统提供一个断电状态下还能运行的实时时钟。当系统断电时，主板上的 RTC 芯片将继续维持系统的时间，这样保证再次开机后系统的时间不会错误。当系统开始时，内核从 RTC 中读取时间来初始化系统时间，关机时便又将系统时间写回到 RTC 中，关机阶段将由主板上另外的电池来供应 RTC 计时。Android 中的 Alarm 在设备处于睡眠模式时仍保持活跃，它可以设置来唤醒设备。在本节的内容中，将详细讲解 Android 系统中 Alarm 报警系统的基本知识。

在 Android 平台中，Alarm 系统的基本层次结构如图 8-1 所示。



▲图 8-1　Alarm 系统的基本层次结构

由图 8-1 可知，Android 平台中 Wi-Fi 系统从上到下主要包括：AlarmManager、AlarmManagerService、AlarmManagerService JNI、Alarm 驱动程序和实时时钟（RTC）驱动程序，这几部分的系统结构如图 8-2 所示。



▲图 8-2　Alarm 的系统结构

图 8-2 中各个部分的具体说明如下所示。

（1）RTC 驱动程序。

Linux 的 Alarm 驱动程序代码路径在内核的“drivers/rtc/”目录中，各个硬件的具体实现不同。

（2）Alarm 驱动程序。

这是 Android 特定内核的组件，能够调用 RTC 系统的功能，但是其本身和硬件无关。Alarm 驱动程序的实现文件如下所示。

```
drivers/staging/android/alarm.c
drivers/staging/android/android_alarm.h
drivers/staging/android/alarm-dev.c
```

（3）本地 JNI 部分。

代码路径如下所示。

```
frameworks/base/services/jni/com_android_server_AlarmManagerService.cpp
```

此文件是 Alarm 部分的本地代码，也同时提供了 JNI 的接口。

（4）Java 部分。

此部分的代码路径如下所示。

```
frameworks/base/services/java/com/android/server/AlarmManagerService.java
frameworks/base/core/java/android/app/AlarmManager.java
```

在文件 AlarmManagerService.java 中实现了 android.server 包中的 AlarmManagerService。在文件 AlarmManager.java 中实现了 android.app 包中的 AlarmManager，它通过使用 AlarmManagerService 服务实现，并对 Java 层提供了平台 API。

8.2 分析 RTC 驱动程序

RTC 驱动程序是 Linux 中标准的 Alarm 驱动程序框架，此驱动程序的框架内容在内核文件“inlcude/linux/rtc.h”中定义。首先在此文件中定义了如下两个函数，功能是分别实现注册和注销 RTC 设备。具体代码如下所示。

```
extern struct rtc_device *rtc_device_register(const char *name,
                                    struct device *dev,
```

```
                                   const struct rtc_class_ops *ops,
                                   struct module *owner);
extern void rtc_device_unregister(struct rtc_device *rtc);
```

然后定义结构体 rtc_class_ops，具体实现代码如下所示。

```
struct rtc_class_ops {
       int (*open)(struct device *);
       void (*release)(struct device *);
       int (*ioctl)(struct device *, unsigned int, unsigned long);
       int (*read_time)(struct device *, struct rtc_time *);
       int (*set_time)(struct device *, struct rtc_time *);
       int (*read_alarm)(struct device *, struct rtc_wkalrm *);
       int (*set_alarm)(struct device *, struct rtc_wkalrm *);
       int (*proc)(struct device *, struct seq_file *);
       int (*set_mmss)(struct device *, unsigned long secs);
       int (*read_callback)(struct device *, int data);
       int (*alarm_irq_enable)(struct device *, unsigned int enabled);
};
```

结构体 struct rtc_device 是在 RTC 驱动程序中使用的，是对 struct device 的扩展，其中也包含了 rtc_class_ops 结构。RTC 驱动程序实际上就是实现了 rtc_class_ops 中的函数指针，主要包括时间和警报器这两方面的内容。

在用户空间中，可以通过 RTC 驱动程序的设备节点对其进行调试，调试的方法是通过 ioctl 命令实现的。这些命令在文件 rtc.h 中定义，以 RTC 开头。例如下面就是 4 个命令。

```
#define RTC_ALM_SET    _IOW('p', 0x07, struct rtc_time)       /* 设置警报器时间 */
#define RTC_ALM_READ   _IOR('p', 0x08, struct rtc_time)       /* 读取警报器时间 */
#define RTC_RD_TIME    _IOR('p', 0x09, struct rtc_time)       /* 读取 RTC 时间 */
#define RTC_SET_TIME   _IOW('p', 0x0a, struct rtc_time)       /* 设置 RTC 时间 */
```

8.3 Alarm 驱动程序详解

在 Android 系统中，Alarm 驱动程序为用户空间提供了设备节点“/dev/alarm”，这是一个主设备号为 10 的 Misc 字符设备，其次设备号是动态生成的。Alarm 驱动程序由内核代码中的如下文件实现的。

```
drivers/staging/android/alarm.c
drivers/staging/android/android_alarm.h
drivers/staging/android/alarm-dev.c
```

在本节的内容中，将详细讲解上述文件的具体实现流程。

8.3.1 分析文件 android_alarm.h

头文件 android_alarm.h 的功能是提供了到用户空间的各 ioctl 命令接口，具体实现代码如下所示。

```
#ifndef _LINUX_ANDROID_ALARM_H
#define _LINUX_ANDROID_ALARM_H

#include <linux/ioctl.h>
#include <linux/time.h>
#include <linux/compat.h>

enum android_alarm_type {
      /*返回码的比特数或设置报警参数*/
      ANDROID_ALARM_RTC_WAKEUP,
      ANDROID_ALARM_RTC,
      ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP,
```

```
        ANDROID_ALARM_ELAPSED_REALTIME,
        ANDROID_ALARM_SYSTEMTIME,

        ANDROID_ALARM_TYPE_COUNT,

        /*返回码的比特数

        /* ANDROID_ALARM_TIME_CHANGE = 16 */
};

enum android_alarm_return_flags {
        ANDROID_ALARM_RTC_WAKEUP_MASK = 1U << ANDROID_ALARM_RTC_WAKEUP,
        ANDROID_ALARM_RTC_MASK = 1U << ANDROID_ALARM_RTC,
       ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP_MASK =
                        1U << ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP,
       ANDROID_ALARM_ELAPSED_REALTIME_MASK =
                        1U << ANDROID_ALARM_ELAPSED_REALTIME,
       ANDROID_ALARM_SYSTEMTIME_MASK = 1U << ANDROID_ALARM_SYSTEMTIME,
       ANDROID_ALARM_TIME_CHANGE_MASK = 1U << 16
;

/*禁用报警*/
#define ANDROID_ALARM_CLEAR(type)            _IO('a', 0 | ((type) <<4))

/*确认最后报警并等待下一个*/
#define ANDROID_ALARM_WAIT                  _IO('a', 1)

#define ALARM_IOW(c, type, size)             _IOW('a', (c) | ((type) <<4), size)
/*设置报警*/
#define ANDROID_ALARM_SET(type)             ALARM_IOW(2, type, struct timespec)
#define ANDROID_ALARM_SET_AND_WAIT(type)     ALARM_IOW(3, type, struct timespec)
#define ANDROID_ALARM_GET_TIME(type)        ALARM_IOW(4, type, struct timespec)
#define ANDROID_ALARM_SET_RTC               _IOW('a',5, struct timespec)
#define ANDROID_ALARM_BASE_CMD(cmd)         (cmd & ~(_IOC(0, 0, 0xf0, 0)))
#define ANDROID_ALARM_IOCTL_TO_TYPE(cmd)     (_IOC_NR(cmd) >>4)


#ifdef CONFIG_COMPAT
#define ANDROID_ALARM_SET_COMPAT(type)          ALARM_IOW(2, type, \
                                                 struct compat_timespec)
#define ANDROID_ALARM_SET_AND_WAIT_COMPAT(type) ALARM_IOW(3, type, \
                                                 struct compat_timespec)
#define ANDROID_ALARM_GET_TIME_COMPAT(type)      ALARM_IOW(4, type, \
                                                 struct compat_timespec)
#define ANDROID_ALARM_SET_RTC_COMPAT             _IOW('a',5, \
                                                 struct compat_timespec)
#define ANDROID_ALARM_IOCTL_NR(cmd)              (_IOC_NR(cmd) & ((1<<4)-1))
#define ANDROID_ALARM_COMPAT_TO_NORM(cmd)  \
                        ALARM_IOW(ANDROID_ALARM_IOCTL_NR(cmd), \
                               ANDROID_ALARM_IOCTL_TO_TYPE(cmd), \
                               struct timespec)

#endif

#endif
```

上述代码的核心是枚举 android_alarm_type，在里面定义了一些和 Alarm 相关的信息，主要包括如下 5 种类型的 Alarm。

- ANDROID_RTC_WAKEUP 类型：表示在触发 Alarm 时需要唤醒设备，反之则不需要唤醒设备；
- ANDROID_ALARM_RTC_WAKEUP 类型：表示在指定的某一时刻触发 Alarm；
- ANDROID_ALARM_ELAPSED_REALTIME：表示在设备启动后，流逝的时间达到总时间之后触发 Alarm；
- ANDROID_ALARM_SYSTEMTIME 类型：表示系统时间；

● ANDROID_ALARM_TYPE_COUN：表示 Alram 类型的计数。

Alarm 返回标记随着 Alarm 的类型而改变。通过定义的宏实现禁用 Alarm、Alarm 等待、设置 Alarm 等功能。

8.3.2 分析文件 alarm.c

Android 系统的 Alarm 模块提供了如下所示的两个设备。

● Alarm 的 Platform Driver，实现文件是 alarm.c。

● 暴露给用户使用的接口 MISC 的 Alarm 接口，实现文件是 alarm-dev.c。

文件 alarm.c 的功能是定义一系列的 ioctl 函数，来操纵 Platform Alarm Driver 提供的功能。在本节的内容中，将首先讲解文件 alarm.c 的具体实现源码。

（1）首先进行初始化处理。函数 alarm_driver_init 的功能是初始化 5 个 Alarm Device 相关联的 hrtimer 定时器，设置 hrtimer 定时器的回调函数为 alarm_timer_triggered。然后再注册一个 Plateform Driver 和 class interface。函数 alarm_driver_init 的具体实现代码如下所示。

```
static int __init alarm_driver_init(void)
{
	int err;
	int i;

	for (i = 0; i < ANDROID_ALARM_SYSTEMTIME; i++) {
		hrtimer_init(&alarms[i].timer,
				CLOCK_REALTIME, HRTIMER_MODE_ABS);
		alarms[i].timer.function = alarm_timer_triggered;
	}
	hrtimer_init(&alarms[ANDROID_ALARM_SYSTEMTIME].timer,
			CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
	alarms[ANDROID_ALARM_SYSTEMTIME].timer.function = alarm_timer_triggered;
	err = platform_driver_register(&alarm_driver);
	if (err < 0)
		goto err1;
	wake_lock_init(&alarm_rtc_wake_lock, WAKE_LOCK_SUSPEND, "alarm_rtc");
	rtc_alarm_interface.class = rtc_class;
	err = class_interface_register(&rtc_alarm_interface);
	if (err < 0)
		goto err2;

	return 0;

err2:
	wake_lock_destroy(&alarm_rtc_wake_lock);
	platform_driver_unregister(&alarm_driver);
err1:
	return err;
}
```

在上述代码中用到了 platform_driver 类型的 alarm_driver 结构体，具体实现代码如下所示。

```
static struct platform_driver alarm_driver = {
	.suspend = alarm_suspend,
	.resume = alarm_resume,
	.driver = {
		.name = "alarm"
	}
};
```

在上述代码中，指定了当系统挂起（suspend）和唤醒（resume）时所需要的实现，分别是 alarm_suspend 和 alarm_resume，同时将 Alarm 设备驱动的名称设置为了“alarm”。

函数 alarm_suspend 的具体实现代码如下所示。

```
static int alarm_suspend(struct platform_device *pdev, pm_message_t state)
{
	int                 err = 0;
	unsigned long        flags;
	struct rtc_wkalrm    rtc_alarm;
	struct rtc_time      rtc_current_rtc_time;
	unsigned long        rtc_current_time;
	unsigned long        rtc_alarm_time;
	struct timespec      rtc_delta;
	struct timespec      wall_time;
	struct alarm_queue *wakeup_queue = NULL;
	struct alarm_queue *tmp_queue = NULL;

	pr_alarm(SUSPEND, "alarm_suspend(%p, %d)\n", pdev, state.event);

	spin_lock_irqsave(&alarm_slock, flags);
	suspended = true;
	spin_unlock_irqrestore(&alarm_slock, flags);

	hrtimer_cancel(&alarms[ANDROID_ALARM_RTC_WAKEUP].timer);
	hrtimer_cancel(&alarms[
			ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP].timer);

	tmp_queue = &alarms[ANDROID_ALARM_RTC_WAKEUP];
	if (tmp_queue->first)
		wakeup_queue = tmp_queue;
	tmp_queue = &alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP];
	if (tmp_queue->first && (!wakeup_queue ||
				hrtimer_get_expires(&tmp_queue->timer).tv64 <
				hrtimer_get_expires(&wakeup_queue->timer).tv64))
		wakeup_queue = tmp_queue;
	if (wakeup_queue) {
		rtc_read_time(alarm_rtc_dev, &rtc_current_rtc_time);
		getnstimeofday(&wall_time);
		rtc_tm_to_time(&rtc_current_rtc_time, &rtc_current_time);
		set_normalized_timespec(&rtc_delta,
					wall_time.tv_sec - rtc_current_time,
					wall_time.tv_nsec);

		rtc_alarm_time = timespec_sub(ktime_to_timespec(
			hrtimer_get_expires(&wakeup_queue->timer)),
			rtc_delta).tv_sec;

		rtc_time_to_tm(rtc_alarm_time, &rtc_alarm.time);
		rtc_alarm.enabled = 1;
		rtc_set_alarm(alarm_rtc_dev, &rtc_alarm);
		rtc_read_time(alarm_rtc_dev, &rtc_current_rtc_time);
		rtc_tm_to_time(&rtc_current_rtc_time, &rtc_current_time);
		pr_alarm(SUSPEND,
			"rtc alarm set at %ld, now %ld, rtc delta %ld.%09ld\n",
			rtc_alarm_time, rtc_current_time,
			rtc_delta.tv_sec, rtc_delta.tv_nsec);
		if (rtc_current_time + 1 >= rtc_alarm_time) {
			pr_alarm(SUSPEND, "alarm about to go off\n");
			memset(&rtc_alarm, 0, sizeof(rtc_alarm));
			rtc_alarm.enabled = 0;
			rtc_set_alarm(alarm_rtc_dev, &rtc_alarm);

			spin_lock_irqsave(&alarm_slock, flags);
			suspended = false;
			wake_lock_timeout(&alarm_rtc_wake_lock, 2 * HZ);
			update_timer_locked(&alarms[ANDROID_ALARM_RTC_WAKEUP],
								false);
			update_timer_locked(&alarms[
				ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP], false);
			err = -EBUSY;
			spin_unlock_irqrestore(&alarm_slock, flags);
		}
	}
```

```
        return err;
}
```

函数 alarm_resume 的具体实现代码如下所示。

```
static int alarm_resume(struct platform_device *pdev)
{
        struct rtc_wkalrm alarm;
        unsigned long      flags;

        pr_alarm(SUSPEND, "alarm_resume(%p)\n", pdev);

        memset(&alarm, 0, sizeof(alarm));
        alarm.enabled = 0;
        rtc_set_alarm(alarm_rtc_dev, &alarm);

        spin_lock_irqsave(&alarm_slock, flags);
        suspended = false;
        update_timer_locked(&alarms[ANDROID_ALARM_RTC_WAKEUP], false);
        update_timer_locked(&alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP],
                                                        false);
        spin_unlock_irqrestore(&alarm_slock, flags);

        return 0;
}
```

另外，在函数 alarm_driver_init 中还用到了类 class_interface 的接口 rtc_alarm_interface，具体实现代码如下所示。

```
static struct class_interface rtc_alarm_interface = {
        .add_dev = &rtc_alarm_add_device,
        .remove_dev = &rtc_alarm_remove_device,
};
```

（2）再看 hrtimer 定时器回调函数 alarm_timer_triggered，其功能是轮询红黑树中所有的 Alarm 节点符合条件的节点，如果有则执行 alarm.functionalarm 指向文件 alarm_dev.c 中的 alarm_triggered 函数。这是因为在执行文件 alarm_dev.c 中的 alarm_init 函数时，会把每个 Alarm 节点的 function 设置成 alarm_triggered。具体实现代码如下所示。

```
static enum hrtimer_restart alarm_timer_triggered(struct hrtimer *timer)
{
        struct alarm_queue *base;
        struct android_alarm *alarm;
        unsigned long flags;
        ktime_t now;

        spin_lock_irqsave(&alarm_slock, flags);

        base = container_of(timer, struct alarm_queue, timer);
        now = base->stopped ? base->stopped_time : hrtimer_cb_get_time(timer);
        now = ktime_sub(now, base->delta);

        pr_alarm(INT, "alarm_timer_triggered type %ld at %lld\n",
                base - alarms, ktime_to_ns(now));

        while (base->first) {
                alarm = container_of(base->first, struct android_alarm, node);
                if (alarm->softexpires.tv64 > now.tv64) {
                        pr_alarm(FLOW, "don't call alarm, %pF, %lld (s %lld)\n",
                                alarm->function, ktime_to_ns(alarm->expires),
                                ktime_to_ns(alarm->softexpires));
                        break;
                }
                base->first = rb_next(&alarm->node);
                rb_erase(&alarm->node, &base->alarms);
                RB_CLEAR_NODE(&alarm->node);
```

```
            pr_alarm(CALL, "call alarm, type %d, func %pF, %lld (s %lld)\n",
                   alarm->type, alarm->function,
                   ktime_to_ns(alarm->expires),
                   ktime_to_ns(alarm->softexpires));
            spin_unlock_irqrestore(&alarm_slock, flags);
            alarm->function(alarm);
            spin_lock_irqsave(&alarm_slock, flags);
        }
        if (!base->first)
            pr_alarm(FLOW, "no more alarms of type %ld\n", base - alarms);
        update_timer_locked(base, true);
        spin_unlock_irqrestore(&alarm_slock, flags);
        return HRTIMER_NORESTART;
}
```

在上述代码中，用到了函数 alarm_triggered 和 alarm_timer_triggered。其中前者是 rtc 芯片的 alarm 中断的回调函数，后者是 android alarm_queue_gttimer 到时的回调函数。

（3）再看函数 rtc_alarm_add_device，其功能是注册了一个 rtc 中断 rtc_irq_register，具体实现代码如下所示。

```
static int rtc_alarm_add_device(struct device *dev,
                struct class_interface *class_intf)
{
        int err;
        struct rtc_device *rtc = to_rtc_device(dev);

        mutex_lock(&alarm_setrtc_mutex);

        if (alarm_rtc_dev) {
                err = -EBUSY;
                goto err1;
        }

        alarm_platform_dev =
                platform_device_register_simple("alarm", -1, NULL, 0);
        if (IS_ERR(alarm_platform_dev)) {
                err = PTR_ERR(alarm_platform_dev);
                goto err2;
        }
        err = rtc_irq_register(rtc, &alarm_rtc_task);
        if (err)
                goto err3;
        alarm_rtc_dev = rtc;
        pr_alarm(INIT_STATUS, "using rtc device, %s, for alarms", rtc->name);
        mutex_unlock(&alarm_setrtc_mutex);

        return 0;

err3:
        platform_device_unregister(alarm_platform_dev);
err2:
err1:
        mutex_unlock(&alarm_setrtc_mutex);
        return err;
}
```

（4）再看中断的回调函数 alarm_triggered_func，具体实现代码如下所示。

```
static void alarm_triggered_func(void *p)
{
        struct rtc_device *rtc = alarm_rtc_dev;
        if (!(rtc->irq_data & RTC_AF))
                return;
        pr_alarm(INT, "rtc alarm triggered\n");
        wake_lock_timeout(&alarm_rtc_wake_lock, 1 * HZ);
}
```

通过上述实现代码可知，当硬件 rtc chip 中的 Alarm 发生中断时，系统会调用函数 alarm_triggered_func。函数 alarm_triggered_func 的功能是调用函数 wake_lock_timeout 锁住 alarm_rtc_wake_lock 1 秒。因为这时 Alarm 会进入 alarm_resumelock 锁住 alarm_rtc_wake_lock，这样可以防止 Alarm 在此时进入 suspend 状态。

（5）再来分析唤醒和休眠。wakelock 有加锁和解锁两种操作，而加锁又可以分为如下两种方式。

- 永久加锁（wake_lock），这种锁必须手动的解锁。
- 超时锁（wake_lock_timeout），这种锁在过去指定时间后，会自动解锁。

永久加锁（wake_lock）和超时锁（wake_lock_timeout）的实现代码如下所示。

```
void wake_lock(struct wake_lock *lock)
{
wake_lock_internal(lock, 0, 0);
}
void wake_lock_timeout(struct wake_lock *lock, long timeout)
{
wake_lock_internal(lock, timeout, 1);
}
```

对于 wakelock 来说，如果“timeout = has_timeout = 0”则直接加锁后退出。在上述代码中用到了函数 wake_lock_internal，具体实现代码如下所示。

```
static void wake_lock_internal(
struct wake_lock *lock, long timeout, int has_timeout)
{
int type;
unsigned long irqflags;
long expire_in;
spin_lock_irqsave(&list_lock, irqflags);
type = lock->flags & WAKE_LOCK_TYPE_MASK;
BUG_ON(type >= WAKE_LOCK_TYPE_COUNT);
BUG_ON(!(lock->flags & WAKE_LOCK_INITIALIZED));
#ifdef CONFIG_WAKELOCK_STAT
if (type == WAKE_LOCK_SUSPEND && wait_for_wakeup) {
if (debug_mask & DEBUG_WAKEUP)
pr_info("wakeup wake lock: %s\n", lock->name);
wait_for_wakeup = 0;
lock->stat.wakeup_count++;
}
if ((lock->flags & WAKE_LOCK_AUTO_EXPIRE) &&
   (long)(lock->expires - jiffies) <= 0) {
wake_unlock_stat_locked(lock, 0);
lock->stat.last_time = ktime_get();
}
#endif
if (!(lock->flags & WAKE_LOCK_ACTIVE)) {
lock->flags |= WAKE_LOCK_ACTIVE;
#ifdef CONFIG_WAKELOCK_STAT
lock->stat.last_time = ktime_get();
#endif
}
list_del(&lock->link);
if (has_timeout) {
if (debug_mask & DEBUG_WAKE_LOCK)
pr_info("wake_lock: %s, type %d, timeout %ld.%03lu\n",
lock->name, type, timeout / HZ,
(timeout % HZ) * MSEC_PER_SEC / HZ);
lock->expires = jiffies + timeout;
lock->flags |= WAKE_LOCK_AUTO_EXPIRE;
list_add_tail(&lock->link, &active_wake_locks[type]);
} else {
if (debug_mask & DEBUG_WAKE_LOCK)
pr_info("wake_lock: %s, type %d\n", lock->name, type);
```

```
lock->expires = LONG_MAX;
lock->flags &= ~WAKE_LOCK_AUTO_EXPIRE;
list_add(&lock->link, &active_wake_locks[type]);
}
if (type == WAKE_LOCK_SUSPEND) {
current_event_num++;
#ifdef CONFIG_WAKELOCK_STAT
if (lock == &main_wake_lock)
update_sleep_wait_stats_locked(1);
else if (!wake_lock_active(&main_wake_lock))
update_sleep_wait_stats_locked(0);
#endif
if (has_timeout)
expire_in = has_wake_lock_locked(type);
else
expire_in = -1;
if (expire_in > 0) {
if (debug_mask & DEBUG_EXPIRE)
pr_info("wake_lock: %s, start expire timer, "
"%ld\n", lock->name, expire_in);
mod_timer(&expire_timer, jiffies + expire_in);
} else {
if (del_timer(&expire_timer))
if (debug_mask & DEBUG_EXPIRE)
pr_info("wake_lock: %s, stop expire timer\n",
lock->name);
if (expire_in == 0)
queue_work(suspend_work_queue, &suspend_work);
}
}
spin_unlock_irqrestore(&list_lock, irqflags);
}
```

而对于 wake_lock_timeout 来说，在经过 timeout 时间后才可以加锁。当判断当前持有 wakelock 的时候启动另一个定时器，然后在 expire_timer 的回调函数中再次判断是否持有 wakelock。函数 expire_wake_locks 的具体实现代码如下所示。

```
static void expire_wake_locks(unsigned long data)
{
long has_lock;
unsigned long irqflags;
if (debug_mask & DEBUG_EXPIRE)
pr_info("expire_wake_locks: start\n");
spin_lock_irqsave(&list_lock, irqflags);
if (debug_mask & DEBUG_SUSPEND)
print_active_locks(WAKE_LOCK_SUSPEND);
has_lock = has_wake_lock_locked(WAKE_LOCK_SUSPEND);
if (debug_mask & DEBUG_EXPIRE)
pr_info("expire_wake_locks: done, has_lock %ld\n", has_lock);
if (has_lock == 0)
queue_work(suspend_work_queue, &suspend_work);
spin_unlock_irqrestore(&list_lock, irqflags);
}
```

在 wakelock 中，有如下两个地方可以让系统从 early_suspend 进入 suspend 状态。

- 在 wake_unlock 中，解锁之后，若没有其他的 wakelock，则进入 suspend。
- 在超时锁的定时器超时后，定时器的回调函数会判断有没有其他的 wakelock，没有则进入 suspend。

（6）再看函数 alarm_late_init。当启动 Alarm 后需要读取当前的 RCT 和系统时间，由于需要确保在这个操作过程中不被中断，或者在中断之后能告诉其他进程该过程没有读取完成，不能被请求，因此这里需要通过 spin_lock_irqsave 和 spin_unlock_irqrestore 来对其执行锁定和解锁操作。函数 alarm_late_init 的具体实现代码如下所示。

```
static int __init alarm_late_init(void)
{
      unsigned long  flags;
      struct timespec tmp_time, system_time;

      /* this needs to run after the rtc is read at boot */
      spin_lock_irqsave(&alarm_slock, flags);
      /* We read the current rtc and system time so we can later calulate
       * elasped realtime to be (boot_systemtime + rtc - boot_rtc) ==
       * (rtc - (boot_rtc - boot_systemtime))
       */
      getnstimeofday(&tmp_time);
      ktime_get_ts(&system_time);
      alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP].delta =
             alarms[ANDROID_ALARM_ELAPSED_REALTIME].delta =
                    timespec_to_ktime(timespec_sub(tmp_time, system_time));

      spin_unlock_irqrestore(&alarm_slock, flags);
      return 0;
}
```

（7）再看函数 alarm_exit。当 Alarm 退出时需要通过函数 class_interface_unregister 卸载在初始化时注册的 Alarm 接口，通过 wake_lock_destroy()函数销毁 SUSPEND lock，并通过函数 platform_driver_unregister()卸载 Alarm 驱动。函数 alarm_exit 的具体实现代码如下所示。

```
static void __exit alarm_exit(void) {
class_interface_unregister(&rtc_alarm_interface);
wake_lock_destroy(&alarm_rtc_wake_lock);
wake_lock_destroy(&alarm_wake_lock);
platform_driver_unregister(&alarm_driver);
}
```

（8）接下来讲解函数 rtc_alarm_add_device 和函数 rtc_alarm_remove_device 的具体实现。在添加设备时，首先将设备转换成 rtc_device 类型，然后通过函数 misc_register 将自己注册成为一个 Misc 设备。在此功能阶段的主要对应代码如下所示。

```
static struct file_operations alarm_fops = {
.owner = THIS_MODULE,
.unlocked_ioctl = alarm_ioctl,
.open = alarm_open,
.release = alarm_release,
};
static struct miscdevice alarm_device = {
.minor = MISC_DYNAMIC_MINOR,
.name = "alarm",
.fops = &alarm_fops,
};
```

在上述代码中，alarm_device 中的“.name”表示设备文件名称，alarm_fops 定义了 Alarm 的常用操作，包括打开、释放和 I/O 控制。另外，还需要通过 rtc_irq_register()函数注册一个 rtc_task，用来处理 Alarm 触发的方法。

（9）再看函数 android_alarm_set_rtc。其功能是处理在指定的某一时刻触发 Alarm 的事件，具体实现代码如下所示。

```
int android_alarm_set_rtc(struct timespec new_time)
{
       int i;
       int ret;
       unsigned long flags;
       struct rtc_time rtc_new_rtc_time;
       struct timespec tmp_time;

       rtc_time_to_tm(new_time.tv_sec, &rtc_new_rtc_time);
```

```
	pr_alarm(TSET, "set rtc %ld %ld - rtc %02d:%02d:%02d %02d/%02d/%04d\n",
		new_time.tv_sec, new_time.tv_nsec,
		rtc_new_rtc_time.tm_hour, rtc_new_rtc_time.tm_min,
		rtc_new_rtc_time.tm_sec, rtc_new_rtc_time.tm_mon + 1,
		rtc_new_rtc_time.tm_mday,
		rtc_new_rtc_time.tm_year + 1900);

	mutex_lock(&alarm_setrtc_mutex);
	spin_lock_irqsave(&alarm_slock, flags);
	wake_lock(&alarm_rtc_wake_lock);
	getnstimeofday(&tmp_time);
	for (i = 0; i < ANDROID_ALARM_SYSTEMTIME; i++) {
		hrtimer_try_to_cancel(&alarms[i].timer);
		alarms[i].stopped = true;
		alarms[i].stopped_time = timespec_to_ktime(tmp_time);
	}
	alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP].delta =
		alarms[ANDROID_ALARM_ELAPSED_REALTIME].delta =
		ktime_sub(alarms[ANDROID_ALARM_ELAPSED_REALTIME].delta,
			timespec_to_ktime(timespec_sub(tmp_time, new_time)));
	spin_unlock_irqrestore(&alarm_slock, flags);
	ret = do_settimeofday(&new_time);
	spin_lock_irqsave(&alarm_slock, flags);
	for (i = 0; i < ANDROID_ALARM_SYSTEMTIME; i++) {
		alarms[i].stopped = false;
		update_timer_locked(&alarms[i], false);
	}
	spin_unlock_irqrestore(&alarm_slock, flags);
	if (ret < 0) {
		pr_alarm(ERROR, "alarm_set_rtc: Failed to set time\n");
		goto err;
	}
	if (!alarm_rtc_dev) {
		pr_alarm(ERROR,
			"alarm_set_rtc: no RTC, time will be lost on reboot\n");
		goto err;
	}
	ret = rtc_set_time(alarm_rtc_dev, &rtc_new_rtc_time);
	if (ret < 0)
		pr_alarm(ERROR, "alarm_set_rtc: "
			"Failed to set RTC, time will be lost on reboot\n");
err:
	wake_unlock(&alarm_rtc_wake_lock);
	mutex_unlock(&alarm_setrtc_mutex);
	return ret;
}
```

（10）再看函数 android_alarm_cancel。其功能是取消报警功能并等待处理完成，返回 0 表示报警是不活跃的，返回 1 表示报警是活跃的。函数 android_alarm_cancel 的具体实现代码如下所示。

```
int android_alarm_cancel(struct android_alarm *alarm)
{
	for (;;) {
		int ret = android_alarm_try_to_cancel(alarm);
		if (ret >= 0)
			return ret;
		cpu_relax();
	}
}
```

（11）再看函数 android_alarm_try_to_cancel。其功能是停止报警功能，具体实现代码如下所示。

```
int android_alarm_try_to_cancel(struct android_alarm *alarm)
{
	struct alarm_queue *base = &alarms[alarm->type];
	unsigned long flags;
	bool first = false;
```

```
	int ret = 0;

	spin_lock_irqsave(&alarm_slock, flags);
	if (!RB_EMPTY_NODE(&alarm->node)) {
		pr_alarm(FLOW, "canceled alarm, type %d, func %pF at %lld\n",
			alarm->type, alarm->function,
			ktime_to_ns(alarm->expires));
		ret = 1;
		if (base->first == &alarm->node) {
			base->first = rb_next(&alarm->node);
			first = true;
		}
		rb_erase(&alarm->node, &base->alarms);
		RB_CLEAR_NODE(&alarm->node);
		if (first)
			update_timer_locked(base, true);
	} else
		pr_alarm(FLOW, "tried to cancel alarm, type %d, func %pF\n",
			alarm->type, alarm->function);
	spin_unlock_irqrestore(&alarm_slock, flags);
	if (!ret && hrtimer_callback_running(&base->timer))
		ret = -1;
	return ret;
}
```

（12）再看函数 alarm_enqueue_locked。在 Alarm 初始化函数中注册了 hrtimer 定时器码，它的回调函数是 alarm_timer_triggered，但是 hrtimer 定时器和 alarm 只是进行了初始化工作，并没有使用它，只有在 hrtimer_start 后才能使用 hrtimer 定时器。而函数 update_timer_locked 是在函数 alarm_enqueue_locked 中被调用的，此函数的具体实现代码如下所示。

```
static void alarm_enqueue_locked(struct android_alarm *alarm)
{
	struct alarm_queue *base = &alarms[alarm->type];
	struct rb_node **link = &base->alarms.rb_node;
	struct rb_node *parent = NULL;
	struct android_alarm *entry;
	int leftmost = 1;
	bool was_first = false;

	pr_alarm(FLOW, "added alarm, type %d, func %pF at %lld\n",
		alarm->type, alarm->function, ktime_to_ns(alarm->expires));

	if (base->first == &alarm->node) {
		base->first = rb_next(&alarm->node);
		was_first = true;
	}
	if (!RB_EMPTY_NODE(&alarm->node)) {
		rb_erase(&alarm->node, &base->alarms);
		RB_CLEAR_NODE(&alarm->node);
	}

	while (*link) {
		parent = *link;
		entry = rb_entry(parent, struct android_alarm, node);
		/*
		* We dont care about collisions. Nodes with
		* the same expiry time stay together.
		*/
		if (alarm->expires.tv64 < entry->expires.tv64) {
			link = &(*link)->rb_left;
		} else {
			link = &(*link)->rb_right;
			leftmost = 0;
		}
	}
	if (leftmost)
		base->first = &alarm->node;
```

```
        if (leftmost || was_first)
                update_timer_locked(base, was_first);

        rb_link_node(&alarm->node, parent, link);
        rb_insert_color(&alarm->node, &base->alarms);
}
```

（13）再看函数 update_timer_locked。其功能是实现报警修改处理，具体实现代码如下所示。

```
static void update_timer_locked(struct alarm_queue *base, bool head_removed)
{
        struct android_alarm *alarm;
        bool is_wakeup = base == &alarms[ANDROID_ALARM_RTC_WAKEUP] ||
                        base == &alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP];

        if (base->stopped) {
                pr_alarm(FLOW, "changed alarm while setting the wall time\n");
                return;
        }

        if (is_wakeup && !suspended && head_removed)
                wake_unlock(&alarm_rtc_wake_lock);

        if (!base->first)
                return;

        alarm = container_of(base->first, struct android_alarm, node);

        pr_alarm(FLOW, "selected alarm, type %d, func %pF at %lld\n",
                alarm->type, alarm->function, ktime_to_ns(alarm->expires));

        if (is_wakeup && suspended) {
                pr_alarm(FLOW, "changed alarm while suspened\n");
                wake_lock_timeout(&alarm_rtc_wake_lock, 1 * HZ);
                return;
        }

        hrtimer_try_to_cancel(&base->timer);
        base->timer.node.expires = ktime_add(base->delta, alarm->expires);
        base->timer._softexpires = ktime_add(base->delta, alarm->softexpires);
        hrtimer_start_expires(&base->timer, HRTIMER_MODE_ABS);
}
```

8.3.3　分析文件 alarm-dev.c

在 Android 系统中，文件 alarm-dev.c 以文件 alarm.c 为基础，实现了与应用层的交互功能，即暴露了 miscdevice 的设备接口。在本节的内容中，将详细讲解文件 alarm-dev.c 的具体实现源码。

（1）首先定义如下所示的全局变量。

```
//标志位，表示 Alarm 设备是否被打开
45static int alarm_opened;
46static DEFINE_SPINLOCK(alarm_slock);
47static struct wakeup_source alarm_wake_lock;
48static DECLARE_WAIT_QUEUE_HEAD(alarm_wait_queue);
//表示设备是否就绪
49static uint32_t alarm_pending;
50static uint32_t alarm_enabled;
51static uint32_t wait_pending;
//每种类型一个 Alarm 设备，Android 目前创建了 5 个 Alarm 设备
61static struct devalarm alarms[ANDROID_ALARM_TYPE_COUNT];
```

（2）分别定义模块初始化函数 alarm_dev_init 和 exit 函数 alarm_dev_exit，具体实现代码如下所示。

```
static int __init alarm_dev_init(void)
{
```

```
        int err;
        int i;

        err = misc_register(&alarm_device);
        if (err)
              return err;

        alarm_init(&alarms[ANDROID_ALARM_RTC_WAKEUP].u.alrm,
                   ALARM_REALTIME, devalarm_alarmhandler);
        hrtimer_init(&alarms[ANDROID_ALARM_RTC].u.hrt,
                   CLOCK_REALTIME, HRTIMER_MODE_ABS);
        alarm_init(&alarms[ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP].u.alrm,
                   ALARM_BOOTTIME, devalarm_alarmhandler);
        hrtimer_init(&alarms[ANDROID_ALARM_ELAPSED_REALTIME].u.hrt,
                   CLOCK_BOOTTIME, HRTIMER_MODE_ABS);
        hrtimer_init(&alarms[ANDROID_ALARM_SYSTEMTIME].u.hrt,
                   CLOCK_MONOTONIC, HRTIMER_MODE_ABS);

        for (i = 0; i < ANDROID_ALARM_TYPE_COUNT; i++) {
              alarms[i].type = i;
              if (!is_wakeup(i))
                    alarms[i].u.hrt.function = devalarm_hrthandler;
        }

        wakeup_source_init(&alarm_wake_lock, "alarm");
        return 0;
}

static void __exit alarm_dev_exit(void)
{
        misc_deregister(&alarm_device);
        wakeup_source_trash(&alarm_wake_lock);
}

module_init(alarm_dev_init);
module_exit(alarm_dev_exit);
```

通过上述实现代码可知，初始化函数 alarm_dev_init 的功能是调用函数 misc_register 注册一个 miscdevice。其中定义报警器设备的结构体的代码如下所示。

```
static struct miscdevice alarm_device = {
      .minor = MISC_DYNAMIC_MINOR,
      .name = "alarm",
      .fops = &alarm_fops,
};
```

对应的 File Operations 为 alarm_fops，具体实现代码如下所示。

```
static const struct file_operations alarm_fops = {
      .owner = THIS_MODULE,
      .unlocked_ioctl = alarm_ioctl,
      .open = alarm_open,
      .release = alarm_release,
#ifdef CONFIG_COMPAT
      .compat_ioctl = alarm_compat_ioctl,
#endif
};
```

接下来就需要为每个 alarm device 调用初始化函数 alarm_init，此函数的代码在文件 alarm.c 中实现。

（3）再看模块 Misc Device 的标准接口函数 alarm_open、alarm_release 和 alarm_ioctl，具体实现代码如下所示。

```
static int alarm_open(struct inode *inode, struct file *file)
{
      file->private_data = NULL;
```

```
        return 0;
}

static int alarm_release(struct inode *inode, struct file *file)
{
        int i;
        unsigned long flags;

        spin_lock_irqsave(&alarm_slock, flags);
        if (file->private_data) {
                for (i = 0; i < ANDROID_ALARM_TYPE_COUNT; i++) {
                        uint32_t alarm_type_mask = 1U << i;
                        if (alarm_enabled & alarm_type_mask) {
                                alarm_dbg(INFO,
                                        "%s: clear alarm, pending %d\n",
                                        __func__,
                                        !!(alarm_pending & alarm_type_mask));
                                alarm_enabled &= ~alarm_type_mask;
                        }
                        spin_unlock_irqrestore(&alarm_slock, flags);
                        devalarm_cancel(&alarms[i]);
                        spin_lock_irqsave(&alarm_slock, flags);
                }
                if (alarm_pending | wait_pending) {
                        if (alarm_pending)
                                alarm_dbg(INFO, "%s: clear pending alarms %x\n",
                                        __func__, alarm_pending);
                        __pm_relax(&alarm_wake_lock);
                        wait_pending = 0;
                        alarm_pending = 0;
                }
                alarm_opened = 0;
        }
        spin_unlock_irqrestore(&alarm_slock, flags);
        return 0;
}
static long alarm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{

        struct timespec ts;
        int rv;

        switch (ANDROID_ALARM_BASE_CMD(cmd)) {
        case ANDROID_ALARM_SET_AND_WAIT(0):
        case ANDROID_ALARM_SET(0):
        case ANDROID_ALARM_SET_RTC:
                if (copy_from_user(&ts, (void __user *)arg, sizeof(ts)))
                        return -EFAULT;
                break;
        }

        rv = alarm_do_ioctl(file, cmd, &ts);
        if (rv)
                return rv;

        switch (ANDROID_ALARM_BASE_CMD(cmd)) {
        case ANDROID_ALARM_GET_TIME(0):
                if (copy_to_user((void __user *)arg, &ts, sizeof(ts)))
                        return -EFAULT;
                break;
        }

        return 0;
}
```

（4）再看 Alarm 定时时间到时候的回调函数 devalarm_triggered，当定时闹铃的时间到了后，函数 alarm_timer_triggered 会调用该回调函数。函数 devalarm_triggered 的具体实现代码如下所示。

```
static void devalarm_triggered(struct devalarm *alarm)
{
	unsigned long flags;
	uint32_t alarm_type_mask = 1U << alarm->type;

	alarm_dbg(INT, "%s: type %d\n", __func__, alarm->type);
	spin_lock_irqsave(&alarm_slock, flags);
	if (alarm_enabled & alarm_type_mask) {
		__pm_wakeup_event(&alarm_wake_lock, 5000); /* 5secs */
		alarm_enabled &= ~alarm_type_mask;
		alarm_pending |= alarm_type_mask;
		wake_up(&alarm_wait_queue);
	}
	spin_unlock_irqrestore(&alarm_slock, flags);
}
```

函数 alarm_timer_triggered 在文件 alarm.c 中定义。

（5）再看函数 alarm_ioctl。其功能是提供了如下所示的 ioctl 命令。

- ANDROID_ALARM_CLEAR：功能是清除 Alarm。
- ANDROID_ALARM_SET_OLD：功能是设置 Alarm 闹铃时间。
- ANDROID_ALARM_SET：功能同上。
- ANDROID_ALARM_SET_AND_WAIT_OLD：功能是设置 Alarm 闹铃时间并等待这个 Alarm。
- ANDROID_ALARM_SET_AND_WAIT：功能同上。
- ANDROID_ALARM_WAIT：功能是等待 Alarm。
- ANDROID_ALARM_SET_RTC：功能是设置 RTC 时间。
- ANDROID_ALARM_GET_TIME：功能是读取 Alarm 时间。

函数 alarm_ioctl 的具体实现代码如下所示。

```
static long alarm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{

	struct timespec ts;
	int rv;

	switch (ANDROID_ALARM_BASE_CMD(cmd)) {
	case ANDROID_ALARM_SET_AND_WAIT(0):
	case ANDROID_ALARM_SET(0):
	case ANDROID_ALARM_SET_RTC:
		if (copy_from_user(&ts, (void __user *)arg, sizeof(ts)))
			return -EFAULT;
		break;
	}

	rv = alarm_do_ioctl(file, cmd, &ts);
	if (rv)
		return rv;

	switch (ANDROID_ALARM_BASE_CMD(cmd)) {
	case ANDROID_ALARM_GET_TIME(0):
		if (copy_to_user((void __user *)arg, &ts, sizeof(ts)))
			return -EFAULT;
		break;
	}

	return 0;
}
```

（6）再看函数 alarm_do_ioctl。其功能是根据“switch…case”语句来执行对应的 iotc 命令，具体实现代码如下所示。

```
static long alarm_do_ioctl(struct file *file, unsigned int cmd,
                                                struct timespec *ts)
{
        int rv = 0;
        unsigned long flags;
        enum android_alarm_type alarm_type = ANDROID_ALARM_IOCTL_TO_TYPE(cmd);

        if (alarm_type >= ANDROID_ALARM_TYPE_COUNT)
                return -EINVAL;

        if (ANDROID_ALARM_BASE_CMD(cmd) != ANDROID_ALARM_GET_TIME(0)) {
                if ((file->f_flags & O_ACCMODE) == O_RDONLY)
                        return -EPERM;
                if (file->private_data == NULL &&
                    cmd != ANDROID_ALARM_SET_RTC) {
                        spin_lock_irqsave(&alarm_slock, flags);
                        if (alarm_opened) {
                                spin_unlock_irqrestore(&alarm_slock, flags);
                                return -EBUSY;
                        }
                        alarm_opened = 1;
                        file->private_data = (void *)1;
                        spin_unlock_irqrestore(&alarm_slock, flags);
                }
        }

        switch (ANDROID_ALARM_BASE_CMD(cmd)) {
        case ANDROID_ALARM_CLEAR(0):
                alarm_clear(alarm_type);
                break;
        case ANDROID_ALARM_SET(0):
                alarm_set(alarm_type, ts);
                break;
        case ANDROID_ALARM_SET_AND_WAIT(0):
                alarm_set(alarm_type, ts);
                /* fall though */
        case ANDROID_ALARM_WAIT:
                rv = alarm_wait();
                break;
        case ANDROID_ALARM_SET_RTC:
                rv = alarm_set_rtc(ts);
                break;
        case ANDROID_ALARM_GET_TIME(0):
                rv = alarm_get_time(alarm_type, ts);
                break;

        default:
                rv = -EINVAL;
        }
        return rv;
}
```

（7）再看函数 alarm_compat_ioctl。其功能是根据“switch...case”实现 iotc 指令的兼容性处理。具体实现代码如下所示。

```
static long alarm_compat_ioctl(struct file *file, unsigned int cmd,
                                                unsigned long arg)
{

        struct timespec ts;
        int rv;

        switch (ANDROID_ALARM_BASE_CMD(cmd)) {
        case ANDROID_ALARM_SET_AND_WAIT_COMPAT(0):
        case ANDROID_ALARM_SET_COMPAT(0):
        case ANDROID_ALARM_SET_RTC_COMPAT:
                if (compat_get_timespec(&ts, (void __user *)arg))
                        return -EFAULT;
```

```
		/* fall through */
	case ANDROID_ALARM_GET_TIME_COMPAT(0):
		cmd = ANDROID_ALARM_COMPAT_TO_NORM(cmd);
		break;
	}

	rv = alarm_do_ioctl(file, cmd, &ts);
	if (rv)
		return rv;

	switch (ANDROID_ALARM_BASE_CMD(cmd)) {
	case ANDROID_ALARM_GET_TIME(0): /* NOTE: we modified cmd above */
		if (compat_put_timespec(&ts, (void __user *)arg))
			return -EFAULT;
		break;
	}

	return 0;
}
```

（8）最后分析各个 iotc 指令对应的处理函数，具体实现代码如下所示。

```
static void devalarm_start(struct devalarm *alrm, ktime_t exp)
{
	if (is_wakeup(alrm->type))
		alarm_start(&alrm->u.alrm, exp);
	else
		hrtimer_start(&alrm->u.hrt, exp, HRTIMER_MODE_ABS);
}


static int devalarm_try_to_cancel(struct devalarm *alrm)
{
	if (is_wakeup(alrm->type))
		return alarm_try_to_cancel(&alrm->u.alrm);
	return hrtimer_try_to_cancel(&alrm->u.hrt);
}

static void devalarm_cancel(struct devalarm *alrm)
{
	if (is_wakeup(alrm->type))
		alarm_cancel(&alrm->u.alrm);
	else
		hrtimer_cancel(&alrm->u.hrt);
}

static void alarm_clear(enum android_alarm_type alarm_type)
{
	uint32_t alarm_type_mask = 1U << alarm_type;
	unsigned long flags;

	spin_lock_irqsave(&alarm_slock, flags);
	alarm_dbg(IO, "alarm %d clear\n", alarm_type);
	devalarm_try_to_cancel(&alarms[alarm_type]);
	if (alarm_pending) {
		alarm_pending &= ~alarm_type_mask;
		if (!alarm_pending && !wait_pending)
			__pm_relax(&alarm_wake_lock);
	}
	alarm_enabled &= ~alarm_type_mask;
	spin_unlock_irqrestore(&alarm_slock, flags);

}

static void alarm_set(enum android_alarm_type alarm_type,
							struct timespec *ts)
{
	uint32_t alarm_type_mask = 1U << alarm_type;
	unsigned long flags;
```

```
	spin_lock_irqsave(&alarm_slock, flags);
	alarm_dbg(IO, "alarm %d set %ld.%09ld\n",
			alarm_type, ts->tv_sec, ts->tv_nsec);
	alarm_enabled |= alarm_type_mask;
	devalarm_start(&alarms[alarm_type], timespec_to_ktime(*ts));
	spin_unlock_irqrestore(&alarm_slock, flags);
}

static int alarm_wait(void)
{
	unsigned long flags;
	int rv = 0;

	spin_lock_irqsave(&alarm_slock, flags);
	alarm_dbg(IO, "alarm wait\n");
	if (!alarm_pending && wait_pending) {
		__pm_relax(&alarm_wake_lock);
		wait_pending = 0;
	}
	spin_unlock_irqrestore(&alarm_slock, flags);

	rv = wait_event_interruptible(alarm_wait_queue, alarm_pending);
	if (rv)
		return rv;

	spin_lock_irqsave(&alarm_slock, flags);
	rv = alarm_pending;
	wait_pending = 1;
	alarm_pending = 0;
	spin_unlock_irqrestore(&alarm_slock, flags);

	return rv;
}
static int alarm_set_rtc(struct timespec *ts)
{
	struct rtc_time new_rtc_tm;
	struct rtc_device *rtc_dev;
	unsigned long flags;
	int rv = 0;

	rtc_time_to_tm(ts->tv_sec, &new_rtc_tm);
	rtc_dev = alarmtimer_get_rtcdev();
	rv = do_settimeofday(ts);
	if (rv < 0)
		return rv;
	if (rtc_dev)
		rv = rtc_set_time(rtc_dev, &new_rtc_tm);

	spin_lock_irqsave(&alarm_slock, flags);
	alarm_pending |= ANDROID_ALARM_TIME_CHANGE_MASK;
	wake_up(&alarm_wait_queue);
	spin_unlock_irqrestore(&alarm_slock, flags);

	return rv;
}

static int alarm_get_time(enum android_alarm_type alarm_type,
					struct timespec *ts)
{
	int rv = 0;

	switch (alarm_type) {
	case ANDROID_ALARM_RTC_WAKEUP:
	case ANDROID_ALARM_RTC:
		getnstimeofday(ts);
		break;
	case ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP:
	case ANDROID_ALARM_ELAPSED_REALTIME:
```

```
            get_monotonic_boottime(ts);
            break;
    case ANDROID_ALARM_SYSTEMTIME:
            ktime_get_ts(ts);
            break;
    default:
            rv = -EINVAL;
    }
    return rv;
}
```

8.4 JNI 层详解

在 Alarm 系统中，本地 JNI 部分的实现源码如下所示。

```
frameworks/base/services/jni/com_android_server_AlarmManagerService.cpp
```

文件 com_android_server_AlarmManagerService.cpp 是 Alarm 部分的本地代码，也同时提供了 JNI 的接口。JNI 层的功能是从底层调用驱动程序，向上面的应用层提供 JNI 接口。文件 com_android_server_AlarmManagerService.cpp 的具体实现代码如下所示。

```
//设置内核时区接口
static jint android_server_AlarmManagerService_setKernelTimezone(JNIEnv* env, jobject
obj, jint fd, jint minswest)
{
    struct timezone tz;

    tz.tz_minuteswest = minswest;
    tz.tz_dsttime = 0;

    int result = settimeofday(NULL, &tz);
    if (result < 0) {
        ALOGE("Unable to set kernel timezone to %d: %s\n", minswest, strerror(errno));
        return -1;
    } else {
        ALOGD("Kernel timezone updated to %d minutes west of GMT\n", minswest);
    }

    return 0;
}
//初始化接口
static jint android_server_AlarmManagerService_init(JNIEnv* env, jobject obj)
{
    return open("/dev/alarm", O_RDWR);
}
//关闭接口
static void android_server_AlarmManagerService_close(JNIEnv* env, jobject obj, jint fd)
{
    close(fd);
}

//设置闹钟接口
static void android_server_AlarmManagerService_set(JNIEnv* env, jobject obj, jint fd,
jint type, jlong seconds, jlong nanoseconds)
{
    struct timespec ts;
    ts.tv_sec = seconds;
    ts.tv_nsec = nanoseconds;

    int result = ioctl(fd, ANDROID_ALARM_SET(type), &ts);
    if (result < 0)
    {
        ALOGE("Unable to set alarm to %lld.%09lld: %s\n", seconds, nanoseconds,
strerror(errno));
    }
```

```
}
//等待时钟接口
static jint android_server_AlarmManagerService_waitForAlarm(JNIEnv* env, jobject obj,
jint fd)
{
   int result = 0;

   do
   {
      result = ioctl(fd, ANDROID_ALARM_WAIT);
   } while (result < 0 && errno == EINTR);

   if (result < 0)
   {
      ALOGE("Unable to wait on alarm: %s\n", strerror(errno));
      return 0;
   }

   return result;
}
//下面是定义接口的代码
static JNINativeMethod sMethods[] = {
     /* name, signature, funcPtr */
   {"init", "()I", (void*)android_server_AlarmManagerService_init},
   {"close", "(I)V", (void*)android_server_AlarmManagerService_close},
   {"set", "(IIJJ)V", (void*)android_server_AlarmManagerService_set},
   {"waitForAlarm", "(I)I", (void*)android_server_AlarmManagerService_waitForAlarm},
   {"setKernelTimezone",                                                    "(II)I",
(void*)android_server_AlarmManagerService_setKernelTimezone},
};
//注册闹钟服务
int register_android_server_AlarmManagerService(JNIEnv* env)
{
   return jniRegisterNativeMethods(env, "com/android/server/AlarmManagerService",
                                   sMethods, NELEM(sMethods));
}

} /* namespace android */
```

8.5 Java 层详解

在 Android 系统中，Alarm 系统的 Java 层的实现代码如下所示。

```
frameworks/base/services/java/com/android/server/AlarmManagerService.java
frameworks/base/core/java/android/app/AlarmManager.java
```

在本节的内容中，将详细讲解上述文件的具体实现流程。

8.5.1　分析 AlarmManagerService 类

在 Android 系统中，闹钟和唤醒功能都是由 Alarm Manager Service 控制并管理的。RTC 闹钟以及定时器都和它有莫大的关系。为了便于称呼，我常常把这个 Service 简称为 ALMS。ALMS 提供了一个 AlarmManager 辅助类。在实际的代码中，应用程序一般都是通过这个辅助类和 ALMS 打交道的。就具体的实现代码来说，辅助类只不过是把一些逻辑语义传递给 ALMS 服务端而已，具体怎么做则完全要看 ALMS 的实现代码了。

因为 ALMS 是服务端的内容，所以必须向外提供具体的接口才能被外界使用。在 Android 平台中，ALMS 的外部接口为 IAlarmManager，在如下所示的脚本中定义。

```
frameworks\base\core\java\android\app\IAlarmManager.aidl
```

具体的定义代码如下所示。

```
interface IAlarmManager {
    void set(int type, long triggerAtTime, in PendingIntent operation);
    void setRepeating(int type, long triggerAtTime, long interval, in PendingIntent
operation);
    void setInexactRepeating(int type, long triggerAtTime, long interval, in
PendingIntent operation);
    void setTime(long millis);
    void setTimeZone(String zone);
    void remove(in PendingIntent operation);
}
```

在大多数情况下，Service 的使用者会通过 Service Manager Service 接口，先获取感兴趣的 Service 对应的代理 I 接口，然后再调用 I 接口的成员函数向 Service 发出请求。所以应该先拿到一个 IAlarmManager 接口，然后再使用它。可是对于 Alarm Manager Service 来说，具体实现情况有所不同，最常见的调用方式如下。

```
manager = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
```

其中，函数 getSystemService 返回的不再是 IAlarmManager 接口，而是 AlarmManager 对象。

在文件 AlarmManagerService.java 中实现了 android.server 包中的 AlarmManagerService 类，具体实现流程如下所示。

（1）首先看类 Alarm，其功能是定义逻辑闹钟类，具体代码如下所示。

```
private static class Alarm {
    public int type;
    public int count;
    public long when;
    public long repeatInterval;
    public PendingIntent operation;
    public int uid;
    public int pid;
    …
```

由此可见，在其中记录了逻辑闹钟的一些关键信息，具体说明如下所示。

● type 域：记录着逻辑闹钟的闹钟类型，比如 RTC_WAKEUP、ELAPSED_REALTIME_WAKEUP 等。

● count 域：是个辅助域，它和 repeatInterval 域一起工作。当 repeatInterval 大于 0 时，这个域可被用于计算下一次重复激发 alarm 的时间。这是针对重复性闹钟的一个辅助域，重复性闹钟的实现原理是，如果当前时刻已经超过闹钟的激发时刻，那么 ALMS 会先从逻辑闹钟数组中摘取 Alarm 节点，并执行闹钟对应的逻辑动作；然后进一步比较“当前时刻”和 Alarm“理应激发的理想时刻”之间的时间跨度，从而计算出 Alarm 的“下一次理应激发的理想时刻”，并将这个激发时间记入 Alarm 节点，接着将该节点重新排入逻辑闹钟列表。这一点和普通 Alarm 不一样，普通 Alarm 节点被摘下后就不再归还逻辑闹钟列表。

● when 域：记录闹钟的激发时间，这个域和 type 域相关。

● repeatInterval 域：表示重复激发闹钟的时间间隔，如果闹钟只需激发一次，则此域为 0；如果闹钟需要重复激发，此域为以毫秒为单位的时间间隔。

● operation 域：记录闹钟激发时应该执行的动作。

● uid 域：记录设置闹钟的进程的 uid。

● pid 域：记录设置闹钟的进程的 pid。

（2）再看函数 set，其功能是设置闹钟，此函数用于设置一次性闹钟。具体实现代码如下所示。

```
public void set(int type, long triggerAtTime, PendingIntent operation) {
    setRepeating(type, triggerAtTime, 0, operation);
}
```

在上述代码中，第一个参数表示闹钟类型，第二个参数表示闹钟执行时间，第三个参数表示闹钟响应动作。

（3）再看函数 setRepeating，其功能是设置周期性的闹钟，具体实现代码如下所示。

```
public void setRepeating(int type, long triggerAtTime, long interval,
        PendingIntent operation) {
    if (operation == null) {
        Slog.w(TAG, "set/setRepeating ignored because there is no intent");
        return;
    }
    synchronized (mLock) {
        Alarm alarm = new Alarm();
        alarm.type = type;
        alarm.when = triggerAtTime;
        alarm.repeatInterval = interval;
        alarm.operation = operation;

        // Remove this alarm if already scheduled.
        removeLocked(operation);

        if (localLOGV) Slog.v(TAG, "set: " + alarm);

        int index = addAlarmLocked(alarm);
        if (index == 0) {
            setLocked(alarm);
        }
    }
}
```

通过上述代码可知，函数 setRepeating 可以设置重复闹钟，其中第一个参数表示闹钟类型，第二个参数表示闹钟首次执行时间，第三个参数表示闹钟两次执行的间隔时间，第四个参数表示闹钟响应动作。其实现原理类似 Java 的 Timer 里面的 scheduleAtFixedRate(TimerTask task, long delay, long period)，都是以近似固定的时间间隔（由指定的周期分隔）进行后续执行。在固定速率执行中，根据已安排的初始执行时间来安排每次执行。如果由于任何原因（如垃圾回收或其他后台活动）而延迟了某次执行，则将快速、连续地出现两次或更多的执行，从而使后续执行能够“追上来”。

（4）再看函数 setInexactRepeating，其功能也是设置重复闹钟，与函数 setRepeating 的功能相似，不过其两个闹钟执行的间隔时间不是固定的。函数 setInexactRepeating 相对而言更节能（power-efficient）一些，因为系统可能会将几个差不多的闹钟合并为一个来执行，从而减少设备的唤醒次数。这一点类似 Java 的 Timer 里面的 schedule(TimerTask task, Date firstTime, long period)，能够根据前一次执行的实际执行时间来安排每次执行。如果由于任何原因（如垃圾回收或其他后台活动）而延迟了某次执行，则后续执行也将被延迟。在长期运行中，执行的频率一般要稍慢于指定周期的倒数（假定 Object.wait(long) 所依靠的系统时钟是准确的）。函数 setInexactRepeating 的具体实现代码如下所示。

```
public void setInexactRepeating(int type, long triggerAtTime, long interval,
        PendingIntent operation) {
    if (operation == null) {
        Slog.w(TAG, "setInexactRepeating ignored because there is no intent");
        return;
    }

    if (interval <= 0) {
        Slog.w(TAG, "setInexactRepeating ignored because interval " + interval
                + " is invalid");
        return;
    }
```

```
        // If the requested interval isn't a multiple of 15 minutes, just treat it as exact
        if (interval % QUANTUM != 0) {
            if (localLOGV) Slog.v(TAG, "Interval " + interval + " not a quantum multiple");
            setRepeating(type, triggerAtTime, interval, operation);
            return;
        }

        // Translate times into the ELAPSED timebase for alignment purposes so that
        // alignment never tries to match against wall clock times.
        final   boolean   isRtc   =   (type   ==   AlarmManager.RTC   ||   type   ==
AlarmManager.RTC_WAKEUP);
        final long skew = (isRtc)
                ? System.currentTimeMillis() - SystemClock.elapsedRealtime()
                : 0;

        // Slip forward to the next ELAPSED-timebase quantum after the stated time.  If
        // we're *at* a quantum point, leave it alone.
        final long adjustedTriggerTime;
        long offset = (triggerAtTime - skew) % QUANTUM;
        if (offset != 0) {
            adjustedTriggerTime = triggerAtTime - offset + QUANTUM;
        } else {
            adjustedTriggerTime = triggerAtTime;
        }

        // Set the alarm based on the quantum-aligned start time
        if (localLOGV) Slog.v(TAG, "setInexactRepeating: type=" + type + " interval=" +
interval
                + " trigger=" + adjustedTriggerTime + " orig=" + triggerAtTime);
        setRepeating(type, adjustedTriggerTime, interval, operation);
    }
```

在上述代码中，参数 int type 表示闹钟的类型，常用的类型有如下所示的 5 个值。

- AlarmManager.ELAPSED_REALTIME：表示当系统进入睡眠状态时，这种类型的闹铃不会唤醒系统，直到系统下次被唤醒才传递它。该闹铃所用的时间是相对时间，是从系统启动后开始计时的,包括睡眠时间，可以通过调用 SystemClock.elapsedRealtime()获得。系统值是 3 (0x00000003)。
- AlarmManager.ELAPSED_REALTIME_WAKEUP：表示闹钟在睡眠状态下会唤醒系统并执行提示功能，该状态下闹钟也使用相对时间，用法同 ELAPSED_REALTIME，系统值是 2 (0x00000002)。
- AlarmManager.RTC：表示闹钟在睡眠状态下，这种类型的闹铃不会唤醒系统，直到系统下次被唤醒才传递它。该闹铃所用的时间是绝对时间，所用时间是 UTC 时间，可以通过调用 System.currentTimeMillis()获得。系统值是 1 (0x00000001)。
- AlarmManager.RTC_WAKEUP：表示闹钟在睡眠状态下会唤醒系统并执行提示功能，该状态下闹钟使用绝对时间。系统值为 0(0x00000000)。
- AlarmManager.POWER_OFF_WAKEUP：表示闹钟在手机关机状态下也能正常进行提示功能（关机闹钟），所以是 5 个状态中用的最多的状态之一。该状态下闹钟也是用绝对时间，系统值为 4(0x00000004)；不过本状态好像受 SDK 版本影响，注意某些版本并不支持此类型。

（5）函数 setTime 的功能是设置时间，具体实现代码如下所示。

```
public void setTime(long millis) {
    mContext.enforceCallingOrSelfPermission(
            "android.permission.SET_TIME",
            "setTime");

    SystemClock.setCurrentTimeMillis(millis);
}
```

（6）函数 setTimeZone 的功能是设置时区，此功能需要 android.permission.SET_TIME_ZONE 权限，具体实现代码如下所示。

```
public void setTimeZone(String tz) {
    mContext.enforceCallingOrSelfPermission(
            "android.permission.SET_TIME_ZONE",
            "setTimeZone");

    long oldId = Binder.clearCallingIdentity();
    try {
        if (TextUtils.isEmpty(tz)) return;
        TimeZone zone = TimeZone.getTimeZone(tz);
        // Prevent reentrant calls from stepping on each other when writing
        // the time zone property
        boolean timeZoneWasChanged = false;
        synchronized (this) {
            String current = SystemProperties.get(TIMEZONE_PROPERTY);
            if (current == null || !current.equals(zone.getID())) {
                if (localLOGV) {
                    Slog.v(TAG, "timezone changed: " + current + ", new=" + zone.getID());
                }
                timeZoneWasChanged = true;
                SystemProperties.set(TIMEZONE_PROPERTY, zone.getID());
            }

            // Update the kernel timezone information
            // Kernel tracks time offsets as 'minutes west of GMT'
            int gmtOffset = zone.getOffset(System.currentTimeMillis());
            setKernelTimezone(mDescriptor, -(gmtOffset / 60000));
        }

        TimeZone.setDefault(null);

        if (timeZoneWasChanged) {
            Intent intent = new Intent(Intent.ACTION_TIMEZONE_CHANGED);
            intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING);
            intent.putExtra("time-zone", zone.getID());
            mContext.sendBroadcastAsUser(intent, UserHandle.ALL);
        }
    } finally {
        Binder.restoreCallingIdentity(oldId);
    }
}
```

（7）函数 remove 的功能是取消某一个闹钟，在取消 alarm 时，是以一个 PendingIntent 对象作为参数进行传递的。这个 PendingIntent 对象正是当初设置 Alarm 时所传入的 operation 参数。不能随便创建一个新的 PendingIntent 对象来调用函数 remove，因为 remove 不会起任何作用。对象 PendingIntent 在 AMS（Activity Manager Service）端对应一个 PendingIntentRecord 实体，而 ALMS 在遍历逻辑闹钟列表时，会根据是否指代相同的 PendingIntentRecord 实体来判断 PendingIntent 是否相符。如果随便地创建一个 PendingIntent 对象并传入函数 remove，那么在 ALMS 端会找不到相符的 PendingIntent 对象，所以 remove 肯定不会起任何作用。函数 remove 的具体实现代码如下所示。

```
public void removeLocked(PendingIntent operation) {
    removeLocked(mRtcWakeupAlarms, operation);
    removeLocked(mRtcAlarms, operation);
    removeLocked(mElapsedRealtimeWakeupAlarms, operation);
    removeLocked(mElapsedRealtimeAlarms, operation);
}
```

在上述代码中，把 4 个逻辑闹钟数组都遍历一遍，删除其中所有和 operation 相符的 Alarm 节点。

（8）函数 removeLocked 的功能是取消闹钟列表中的某个闹钟，具体实现代码如下所示。

```
private void removeLocked(ArrayList<Alarm> alarmList,
        PendingIntent operation) {
    if (alarmList.size() <= 0) {
        return;
    }

    // iterator over the list removing any it where the intent match
    Iterator<Alarm> it = alarmList.iterator();

    while (it.hasNext()) {
        Alarm alarm = it.next();
        if (alarm.operation.equals(operation)) {
            it.remove();
        }
    }
}
```

（9）函数 removeUserLocked 的功能是取消用户设置的闹钟，具体实现代码如下所示。

```
private void removeUserLocked(ArrayList<Alarm> alarmList, int userHandle) {
    if (alarmList.size() <= 0) {
        return;
    }

    // iterator over the list removing any it where the intent match
    Iterator<Alarm> it = alarmList.iterator();

    while (it.hasNext()) {
        Alarm alarm = it.next();
        if (UserHandle.getUserId(alarm.operation.getCreatorUid()) == userHandle) {
            it.remove();
        }
    }
}
```

通过前面的取消闹钟函数的实现代码可以看出，所谓的取消工作只是删除了对应的逻辑 Alarm 节点而已，并不会和底层驱动打什么交道。也就是说，是不存在针对底层“实体闹钟”的删除动作的。所以在底层“实体闹钟”到时之时还是会被“激发”出来，只不过此时在 Frameworks 层会因为找不到符合要求的“逻辑闹钟”而不做进一步的激发动作而已。

（10）再看函数 addAlarmLocked，其功能是将逻辑闹钟添加到内部逻辑闹钟数组的某个合适位置。函数 addAlarmLocked 的具体实现代码如下所示。

```
private int addAlarmLocked(Alarm alarm) {
    ArrayList<Alarm> alarmList = getAlarmList(alarm.type);

    int index = Collections.binarySearch(alarmList, alarm, mIncreasingTimeOrder);
    if (index < 0) {
        index = 0 - index - 1;
    }
    if (localLOGV) Slog.v(TAG, "Adding alarm " + alarm + " at " + index);
    alarmList.add(index, alarm);

    if (localLOGV) {
        // Display the list of alarms for this alarm type
        Slog.v(TAG, "alarms: " + alarmList.size() + " type: " + alarm.type);
        int position = 0;
        for (Alarm a : alarmList) {
            Time time = new Time();
            time.set(a.when);
            String timeStr = time.format("%b %d %I:%M:%S %p");
            Slog.v(TAG, position + ": " + timeStr
                    + " " + a.operation.getTargetPackage());
            position += 1;
```

```
            }
        }

        return index;
    }
```

在 Android 系统中，逻辑闹钟列表是依据 Alarm 的激发时间实现排序的。因为越早被激发的 Alarm 越靠近第 0 位。所以函数 addAlarmLocked 在添加新逻辑闹钟时，需要先用二分查找法快速找到列表中合适的位置，然后再把 Alarm 对象插入此处。

（11）再看类 AlarmThread，这是在 AlarmManagerService 中的一个继承于 Thread 的内嵌类。定义代码如下所示。

```
private class AlarmThread extends Thread
```

类 AlarmThread 的核心是函数 run，具体实现流程如下所示。

- 在 AlarmThread 线程中，在一个 while(true)循环中不断调用函数 waitForAlarm 来等待底层 Alarm 的激发动作。当 AlarmThread 调用到函数 ioctl 时，线程会被阻塞住，直到底层激发 Alarm。而且所激发的 alarm 的类型会记录到 ioctl()的返回值中。这个返回值对外界来说非常重要，外界用它来判断应该遍历哪个逻辑闹钟列表。
- 一旦等到底层驱动的激发动作，AlarmThread 会开始遍历相应的逻辑闹钟列表。AlarmThread 先创建了一个临时的数组列表 triggerList，然后根据 result 的值对相应的 Alarm 数组列表调用函数 triggerAlarmsLocked。如果发现 Alarm 数组列表中某个 Alarm 符合激发条件，则把它移到 triggerList 中。这样，4 条 Alarm 数组列表中需要激发的 Alarm 就汇总到 triggerList 数组列表中了。
- 遍历一遍 triggerList，每当在 while 循环中遍历到一个 Alarm 对象，就执行它的 alarm.operation.send()函数。在 Alarm 中记录的 operation，就是当初设置它时传来的那个 PendingIntent 对象。

函数 run 的具体实现代码如下所示。

```
public void run()
        {
            while (true)
            {
                int result = waitForAlarm(mDescriptor);

                ArrayList<Alarm> triggerList = new ArrayList<Alarm>();

                if ((result & TIME_CHANGED_MASK) != 0) {
                    remove(mTimeTickSender);
                    mClockReceiver.scheduleTimeTickEvent();
                    Intent intent = new Intent(Intent.ACTION_TIME_CHANGED);
                    intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING
                            | Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
                    mContext.sendBroadcastAsUser(intent, UserHandle.ALL);
                }

                synchronized (mLock) {
                    final long nowRTC = System.currentTimeMillis();
                    final long nowELAPSED = SystemClock.elapsedRealtime();
                    if (localLOGV) Slog.v(
                        TAG, "Checking for alarms... rtc=" + nowRTC
                        + ", elapsed=" + nowELAPSED);

                    if ((result & RTC_WAKEUP_MASK) != 0)
                        triggerAlarmsLocked(mRtcWakeupAlarms, triggerList, nowRTC);

                    if ((result & RTC_MASK) != 0)
```

```
                triggerAlarmsLocked(mRtcAlarms, triggerList, nowRTC);

            if ((result & ELAPSED_REALTIME_WAKEUP_MASK) != 0)
                triggerAlarmsLocked(mElapsedRealtimeWakeupAlarms,    triggerList,
nowELAPSED);

            if ((result & ELAPSED_REALTIME_MASK) != 0)
                triggerAlarmsLocked(mElapsedRealtimeAlarms,          triggerList,
nowELAPSED);

            // now trigger the alarms
            Iterator<Alarm> it = triggerList.iterator();
            while (it.hasNext()) {
                Alarm alarm = it.next();
                try {
                    if (localLOGV) Slog.v(TAG, "sending alarm " + alarm);
                    alarm.operation.send(mContext, 0,
                            mBackgroundIntent.putExtra(
                                    Intent.EXTRA_ALARM_COUNT, alarm.count),
                            mResultReceiver, mHandler);

                    // we have an active broadcast so stay awake.
                    if (mBroadcastRefCount == 0) {
                        setWakelockWorkSource(alarm.operation);
                        mWakeLock.acquire();
                    }
                    final InFlight inflight = new InFlight(AlarmManagerService.this,
                            alarm.operation);
                    mInFlight.add(inflight);
                    mBroadcastRefCount++;

                    final BroadcastStats bs = inflight.mBroadcastStats;
                    bs.count++;
                    if (bs.nesting == 0) {
                        bs.nesting = 1;
                        bs.startTime = nowELAPSED;
                    } else {
                        bs.nesting++;
                    }
                    final FilterStats fs = inflight.mFilterStats;
                    fs.count++;
                    if (fs.nesting == 0) {
                        fs.nesting = 1;
                        fs.startTime = nowELAPSED;
                    } else {
                        fs.nesting++;
                    }
                    if (alarm.type == AlarmManager.ELAPSED_REALTIME_WAKEUP
                            || alarm.type == AlarmManager.RTC_WAKEUP) {
                        bs.numWakeup++;
                        fs.numWakeup++;
                        ActivityManagerNative.noteWakeupAlarm(
                                alarm.operation);
                    }
                } catch (PendingIntent.CanceledException e) {
                    if (alarm.repeatInterval > 0) {
                        // This IntentSender is no longer valid, but this
                        // is a repeating alarm, so toss the hoser.
                        remove(alarm.operation);
                    }
                } catch (RuntimeException e) {
                    Slog.w(TAG, "Failure sending alarm.", e);
                }
            }
        }
    }
  }
}
```

在上述代码中调用了函数 noteWakeupAlarm，其功能是调用 BatteryStatsService 服务的相关动作并导致机器的唤醒。函数 noteWakeupAlarm 的具体实现代码如下所示。

```
public void noteWakeupAlarm(IIntentSender sender)
{
    if (!(sender instanceof PendingIntentRecord))
    {
        return;
    }

    BatteryStatsImpl stats = mBatteryStatsService.getActiveStatistics();
    synchronized (stats)
    {
        if (mBatteryStatsService.isOnBattery())
        {
            mBatteryStatsService.enforceCallingPermission();
            PendingIntentRecord rec = (PendingIntentRecord)sender;
            int MY_UID = Binder.getCallingUid();
            int uid = rec.uid == MY_UID ? Process.SYSTEM_UID : rec.uid;
            BatteryStatsImpl.Uid.Pkg pkg = stats.getPackageStatsLocked(uid, rec.key.
packageName);
            pkg.incWakeupsLocked();
        }
    }
}
```

8.5.2　分析 AlarmManager 类

在文件 AlarmManager.java 中实现了 android.app 包中的 AlarmManager 类，它通过使用 AlarmManagerService 服务实现，并对 Java 层提供了平台 API。在 AlarmManager 中提供了如下所示的成员函数，即 1 个构造函数和 6 个功能函数，基本上完全和 IAlarmManager 的成员函数一一对应。

```
AlarmManager(IAlarmManager service)
public void set(int type, long triggerAtTime, PendingIntent operation)
public void setRepeating(int type, long triggerAtTime, long interval,
PendingIntent operation)
public void setInexactRepeating(int type, long triggerAtTime, long interval,
                                PendingIntent operation)
public void cancel(PendingIntent operation)
public void setTime(long millis)
public void setTimeZone(String timeZone)
```

在接下来的内容中，将详细讲解文件 AlarmManager.java 的具体实现流程。

（1）定义常量和接口，具体实现代码如下所示。

```
public static final int RTC_WAKEUP = 0;
public static final int RTC = 1;
public static final int ELAPSED_REALTIME_WAKEUP = 2;
public static final int ELAPSED_REALTIME = 3;
private final IAlarmManager mService;
```

（2）在文件 AlarmManager.java 中提供了闹钟操作的各个接口函数，这个接口函数是通过调用在文件 AlarmManager.java 中对应的函数实现的。各个接口函数的具体实现代码如下所示。

```
    public void set(int type, long triggerAtMillis, PendingIntent operation) {
        try {
            mService.set(type, triggerAtMillis, operation);
        } catch (RemoteException ex) {
        }
    }
    public void setRepeating(int type, long triggerAtMillis,
            long intervalMillis, PendingIntent operation) {
```

```
        try {
            mService.setRepeating(type, triggerAtMillis, intervalMillis, operation);
        } catch (RemoteException ex) {
        }
    }
    public void setInexactRepeating(int type, long triggerAtMillis,
            long intervalMillis, PendingIntent operation) {
        try {
            mService.setInexactRepeating(type,     triggerAtMillis,     intervalMillis,
operation);
        } catch (RemoteException ex) {
        }
    }
    public void cancel(PendingIntent operation) {
        try {
            mService.remove(operation);
        } catch (RemoteException ex) {
        }
    }
    public void setTime(long millis) {
        try {
            mService.setTime(millis);
        } catch (RemoteException ex) {
        }
    }
    public void setTimeZone(String timeZone) {
        try {
            mService.setTimeZone(timeZone);
        } catch (RemoteException ex) {
        }
    }
}
```

因为在本章前面讲解 AlarmManagerService.java 文件时已经涉及了各个操作函数的具体实现，在此不再讲解 AlarmManager.java 文件中的接口函数，因为它们的功能是一样的。

第 9 章　振动器系统

在 Android 智能手机操作系统中，振动器是比较常见的功能之一，例如可以将来电设置为振动模式。在 Android 系统中也有振动系统模块，也能够实现上述来电铃声和闹钟的振动设置。在本章的内容中，将详细讲解 Android 振动器系统驱动的实现和移植内容，为读者进入本书后面知识的学习打下基础。

9.1 振动器系统结构

在 Android 系统中，振动器是负责控制启动或关闭电话振动功能的设备。Android 系统中的振动系统包括驱动程序、硬件抽象层、JNI 部分、Java 框架类等部分，并且向 Java 应用程序层提供了简单的 API 作为平台接口。Android 振动器系统的基本层次结构如图 9-1 所示。

▲图 9-1　Android 振动器系统的基本层次结构

Android 振动器系统自下而上包含了驱动程序、硬件抽象层、Java 框架类、Java 应用等几个部分，其结构如图 9-2 所示。

在图 9-2 中，各个构成元素的具体说明如下所示。

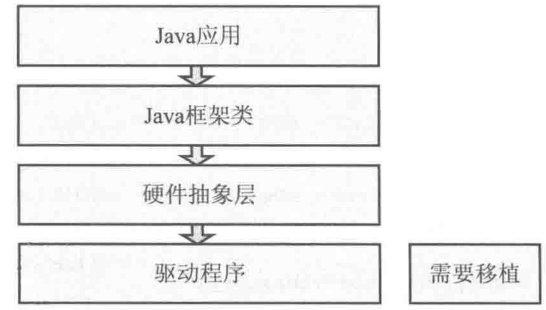
（1）驱动程序。

此处的驱动程序是某特定硬件平台振动器的驱动程序，通常基于 Android 的 Timed Output 驱动框架来实现。

（2）硬件抽象层。

振动器系统的硬件抽象层接口路径如下。

```
hardware/libhardware_legacy/include/hardware_legacy/vibrator.h
```



▲图 9-2 振动器系统结构元素

振动系统的硬件抽象层的默认代码路径如下。

```
hardware/libhardware_legacy/vibrator/vibrator.c
```

因为 Android 振动器的硬件抽象层是 libhardware_legacy.so 的一部分，所以通常并不需要重新实现。

（3）Java 框架类。

振动器系统的 Java 框架部分的代码路径如下。

```
frameworks/base/services/jni/com_android_server_VibratorService.cpp
```

在此文件中定义了振动器的 JNI 部分，通过调用硬件抽象层向上层提供接口。

（4）Java 应用部分。

振动器系统的 Java 部分的代码路径如下。

```
frameworks/base/services/java/com/android/server/VibratorService.java
frameworks/base/core/java/android/os/Vibrator.java
```

文件 VibratorService.java 通过调用 VibratorService JNI 来实现包 com.android.server 中的类 VibratorService。类 VibratorService 不是平台的 API，只被 Android 系统 Java 框架中的一小部分调用。

在文件 Vibrator.java 中实现了 android.os 包中的 Vibrator 类，这是向 Java 层提供的 API。

9.2 硬件抽象层实现详解

振动器系统的硬件抽象层接口的实现文件是 hardware/libhardware_legacy/include/hardware_legacy/vibrator.h，主要实现代码如下所示。

```
#ifndef _HARDWARE_VIBRATOR_H
#define _HARDWARE_VIBRATOR_H
 #if __cplusplus
 extern "C" {
 #endif

 /**
  * 开始振动
  *
  * @振动时间，单位毫秒
  *
  * @返回 0 表示成功,返回 1 表示出错
  */
 int vibrator_on(int timeout_ms);

 /**
```

```
 * 关闭 vibrator
 * @返回 0 表示成功,返回 1 表示出错
 */
int vibrator_off();
#if __cplusplus
} // extern "C"
#endif
#endif // _HARDWARE_VIBRATOR_H
```

振动系统的硬件抽象层在 Android 中的默认代码路径如下所示。

```
hardware/libhardware_legacy/vibrator/vibrator.c
```

文件 vibrator.c 的主要代码如下所示。

```
int vibrator_exists()
{
    int fd;

#ifdef QEMU_HARDWARE
    if (qemu_check()) {
        return 1;
    }
#endif

    fd = open(THE_DEVICE, O_RDWR);
    if(fd < 0)
        return 0;
    close(fd);
    return 1;
}

static int sendit(int timeout_ms)
{
    int nwr, ret, fd;
    char value[20];

#ifdef QEMU_HARDWARE
    if (qemu_check()) {
        return qemu_control_command( "vibrator:%d", timeout_ms );
    }
#endif

    fd = open(THE_DEVICE, O_RDWR);
    if(fd < 0)
        return errno;

    nwr = sprintf(value, "%d\n", timeout_ms);
    ret = write(fd, value, nwr);

    close(fd);

    return (ret == nwr) ? 0 : -1;
}

int vibrator_on(int timeout_ms)
{
    /* constant on, up to maximum allowed time */
    return sendit(timeout_ms);
}

int vibrator_off()
{
    return sendit(0);
}
```

在 Android 系统中，振动器系统的 JNI 框架部分的实现文件是 com_android_server_VibratorService.cpp，主要实现代码如下所示。

```
namespace android
{

static jboolean vibratorExists(JNIEnv *env, jobject clazz)
{
    return vibrator_exists() > 0 ? JNI_TRUE : JNI_FALSE;
}

static void vibratorOn(JNIEnv *env, jobject clazz, jlong timeout_ms)
{
    // ALOGI("vibratorOn\n");
    vibrator_on(timeout_ms);
}

static void vibratorOff(JNIEnv *env, jobject clazz)
{
    // ALOGI("vibratorOff\n");
    vibrator_off();
}

static JNINativeMethod method_table[] = {
    { "vibratorExists", "()Z", (void*)vibratorExists },
    { "vibratorOn", "(J)V", (void*)vibratorOn },//振动器开
    { "vibratorOff", "()V", (void*)vibratorOff }//振动器关
};

int register_android_server_VibratorService(JNIEnv *env)
{
    return jniRegisterNativeMethods(env, "com/android/server/VibratorService",
            method_table, NELEM(method_table));
}

};
```

在上述代码中，核心功能是通过 JNINativeMethod method_table[]和 register_android_server_VibratorService()实现的。

9.3 分析 Java 层部分

在 JNI 层文件 com_android_server_VibratorService.cpp 中，调用了 VibratorService JNI 来实现 com.android.server 包中的 VibratorService 类。类 VibratorService 在文件 frameworks/base/services/java/com/android/server/VibratorService.java 中定义，主要实现代码如下所示。

```
//系统准备
    public void systemReady() {
        mIm = (InputManager)mContext.getSystemService(Context.INPUT_SERVICE);

        mContext.getContentResolver().registerContentObserver(
                Settings.System.getUriFor(Settings.System.VIBRATE_INPUT_DEVICES), true,
                new ContentObserver(mH) {
                    @Override
                    public void onChange(boolean selfChange) {
                        updateInputDeviceVibrators();
                    }
                }, UserHandle.USER_ALL);

        mContext.registerReceiver(new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                updateInputDeviceVibrators();
            }
        }, new IntentFilter(Intent.ACTION_USER_SWITCHED), null, mH);

        updateInputDeviceVibrators();
    }
//已经开始振动
```

```
    public boolean hasVibrator() {
        return doVibratorExists();
    }
//验证传入的 UID
    private void verifyIncomingUid(int uid) {
        if (uid == Binder.getCallingUid()) {
            return;
        }
        if (Binder.getCallingPid() == Process.myPid()) {
            return;
        }
        mContext.enforcePermission(android.Manifest.permission.UPDATE_APP_OPS_STATS,
                Binder.getCallingPid(), Binder.getCallingUid(), null);
    }
//振动函数，不能超时
    public void vibrate(int uid, String packageName, long milliseconds, IBinder token)
{
        if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)
                != PackageManager.PERMISSION_GRANTED) {
            throw new SecurityException("Requires VIBRATE permission");
        }
        verifyIncomingUid(uid);
        // We're running in the system server so we cannot crash. Check for a
        // timeout of 0 or negative. This will ensure that a vibration has
        // either a timeout of > 0 or a non-null pattern.
        if (milliseconds <= 0 || (mCurrentVibration != null
                && mCurrentVibration.hasLongerTimeout(milliseconds))) {
            // Ignore this vibration since the current vibration will play for
            // longer than milliseconds.
            return;
        }

        Vibration vib = new Vibration(token, milliseconds, uid, packageName);

        final long ident = Binder.clearCallingIdentity();
        try {
            synchronized (mVibrations) {
                removeVibrationLocked(token);
                doCancelVibrateLocked();
                mCurrentVibration = vib;
                startVibrationLocked(vib);
            }
        } finally {
            Binder.restoreCallingIdentity(ident);
        }
    }
//设置振动模式函数，可以设置振动重复，也可以设置振动时长
    public void vibratePattern(int uid, String packageName, long[] pattern, int repeat,
            IBinder token) {
        if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)
                != PackageManager.PERMISSION_GRANTED) {
            throw new SecurityException("Requires VIBRATE permission");
        }
        verifyIncomingUid(uid);
        // so wakelock calls will succeed
        long identity = Binder.clearCallingIdentity();
        try {
            if (false) {
                String s = "";
                int N = pattern.length;
                for (int i=0; i<N; i++) {
                    s += " " + pattern[i];
                }
                Slog.i(TAG, "vibrating with pattern: " + s);
            }

            // we're running in the server so we can't fail
            if (pattern == null || pattern.length == 0
                    || isAll0(pattern)
                    || repeat >= pattern.length || token == null) {
                return;
```

```
        }

        Vibration vib = new Vibration(token, pattern, repeat, uid, packageName);
        try {
            token.linkToDeath(vib, 0);
        } catch (RemoteException e) {
            return;
        }

        synchronized (mVibrations) {
            removeVibrationLocked(token);
            doCancelVibrateLocked();
            if (repeat >= 0) {
                mVibrations.addFirst(vib);
                startNextVibrationLocked();
            } else {
                // A negative repeat means that this pattern is not meant
                // to repeat. Treat it like a simple vibration.
                mCurrentVibration = vib;
                startVibrationLocked(vib);
            }
        }
    }
    finally {
        Binder.restoreCallingIdentity(identity);
    }
}
//取消振动
public void cancelVibrate(IBinder token) {
    mContext.enforceCallingOrSelfPermission(
            android.Manifest.permission.VIBRATE,
            "cancelVibrate");

    // so wakelock calls will succeed
    long identity = Binder.clearCallingIdentity();
    try {
        synchronized (mVibrations) {
            final Vibration vib = removeVibrationLocked(token);
            if (vib == mCurrentVibration) {
                doCancelVibrateLocked();
                startNextVibrationLocked();
            }
        }
    }
    finally {
        Binder.restoreCallingIdentity(identity);
    }
}
```

然后通过文件“frameworks/base/core/java/android/os/Vibrator.java”实现 android.os 包中的 Vibrator 类。最后获得名称为 vibrator 的服务，并配合同目录中的 IVibratorService.aidl 文件向应用程序层提供 Vibrator 的相关 API。文件 Vibrator.java 的主要实现代码如下所示。

```
package android.os;
import android.content.Context;
public abstract class Vibrator {
    public Vibrator() {
    }
    /**
     *检查硬件是否有一个振动器
     */
    public abstract boolean hasVibrator();

    /**
     *在指定的时间内一直振动
     */
    public abstract void vibrate(long milliseconds);

    /**
```

```
    *对于一个给定的模式振动
    */
   public abstract void vibrate(long[] pattern, int repeat);
   public abstract void vibrate(int owningUid, String owningPackage, long milliseconds);

   public abstract void vibrate(int owningUid, String owningPackage, long[] pattern, int
repeat);

   /**
    * 关闭 vibrator
    */
   public abstract void cancel();
}
```

9.4 实现硬件抽象层

在接下来的内容中，将详细讲解实现硬件抽象层的具体流程。

1. 硬件抽象层的接口

由前面的知识可以了解到，Vibrator 硬件抽象层的接口在如下文件中。

```
hardware/libhardware_legacy/include/hardware_legacy/vibrator.h
```

文件 vibrator.h 的核心代码如下所示。

```
int vibrator_on(int timeout_ms);            // 开始振动
int vibrator_off();                         // 关闭振动
```

在文件 vibrator.h 中定义了两个接口，分别表示振动和关闭，振动以毫秒（ms）作为时间单位。

2. 实现标准硬件抽象层

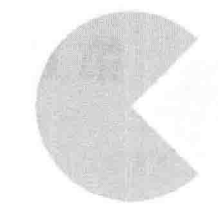
Vibrator 硬件抽象层是标准的实现代码，定义在如下文件中。

```
hardware/libhardware_legacy/vibrator/vibrator.c
```

文件 vibrator.c 的核心内容是函数 sendit()，此函数的实现代码如下所示。

```
#define THE_DEVICE "/sys/class/timed_output/vibrator/enable"
static int sendit(int timeout_ms)
{
    int nwr, ret, fd;
    char value[20];
#ifdef QEMU_HARDWARE                    // 使用 QEMU 的情况
    if (qemu_check()) {
        return qemu_control_command( "vibrator:%d", timeout_ms );
    }
#endif
    fd = open(THE_DEVICE, O_RDWR);
// 读取 sys 文件系统中的内容
    if(fd < 0) return errno;
    nwr = sprintf(value, "%d\n", timeout_ms);
    ret = write(fd, value, nwr);
    close(fd);
    return (ret == nwr) ? 0 : -1;
}
```

上述 sendit()函数的功能是根据时间进行“振动”，在真实的硬件中是通过 sys 文件系统的文件进行控制的，如果是模拟器环境则通过 QEMU 发送命令。其中 vibrator_on()调用 sendit()以时间作为参数，vibrator_on()调用 sendit()以 0 作为参数。



第三篇

典型应用篇

第 10 章　二维图像渲染

第 11 章　绘制二维图像

第 12 章　二维动画应用

第 13 章　渲染二维图像

第 14 章　开发音频应用程序

第 15 章　开发视频应用程序

第 10 章　二维图像渲染

渲染是指对事物的描写、形容和烘托处理，目的是烘染物像，增强艺术效果、质感和立体感。在 Android 多媒体开发应用中，经常因为项目需求需要渲染屏幕中的二维图形图像。在本章的内容中，将详细讲解在 Android 系统中渲染二维图像系统的基本知识，为读者进入本书后面知识的学习打下基础。

10.1 SurfaceFlinger 渲染管理器

在 Android 系统中，为了在缺少 Overlay 的显示设备上达到更好的渲染效果，采用了 SurfaceFlinger 作为系统的渲染管理器。SurfaceFlinger 的设计思想类似于 Windows Vista 和 Linux 下的 Compiz。在本书前面的内容中，已经从底层讲解了 Android GDI 系统中 SurfaceFlinger 的基本知识。在本节将简要介绍 SurfaceFlinger 在图形处理方面的基本应用知识，为读者进入本书后面知识的学习打下基础。

10.1.1 SurfaceFlinger 基础

SurfaceFlinger 按英文翻译过来就是 Surface 投递者，其主要功能如下所示。

（1）将 Layers（Surfaces）内容刷新到屏幕上。

（2）维持 Layer 的 Zorder 序列，并对 Layer 最终输出做出裁剪计算。

（3）响应 Client 要求，创建 Layer 与客户端的 Surface 建立连接。

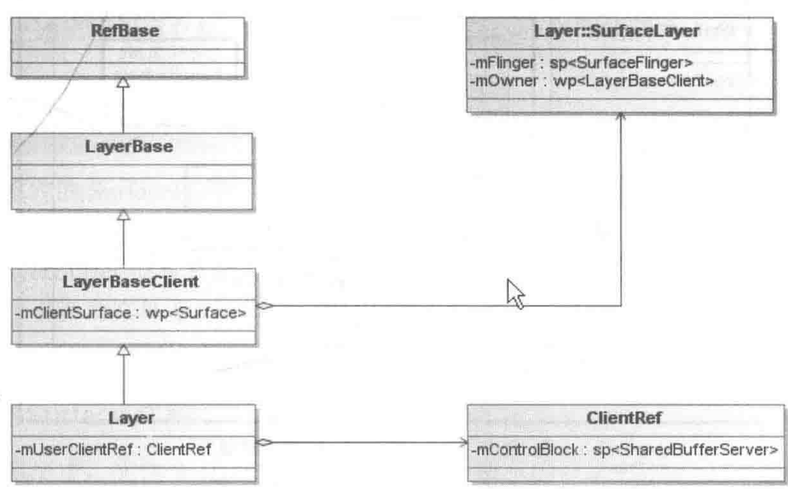
（4）接收 Client 要求，修改 Layer 属性，例如输出大小和 Alpha 等设定。

我们可以将 Surface 理解为一个绘图表面，Android 应用程序负责往这个绘图表面填内容，而 SurfaceFlinger 服务负责将这个绘图表面的内容取出来，并且渲染在显示屏上。在 SurfaceFlinger 服务端，使用 Layer 类来描述绘图表面，Layer 类的实现如图 10-1 所示。

由此可见，类 Layer 继承了类 LayerBaseClient，而类 LayerBaseClient 继承了类 LayerBase，类 LayerBase 继续了类 RefBase。从上述继承关系就可以看出，我们可以通过 Android 系统的智能指针来引用 Layer 对象，从而可以自动地维护它们的生命周期。

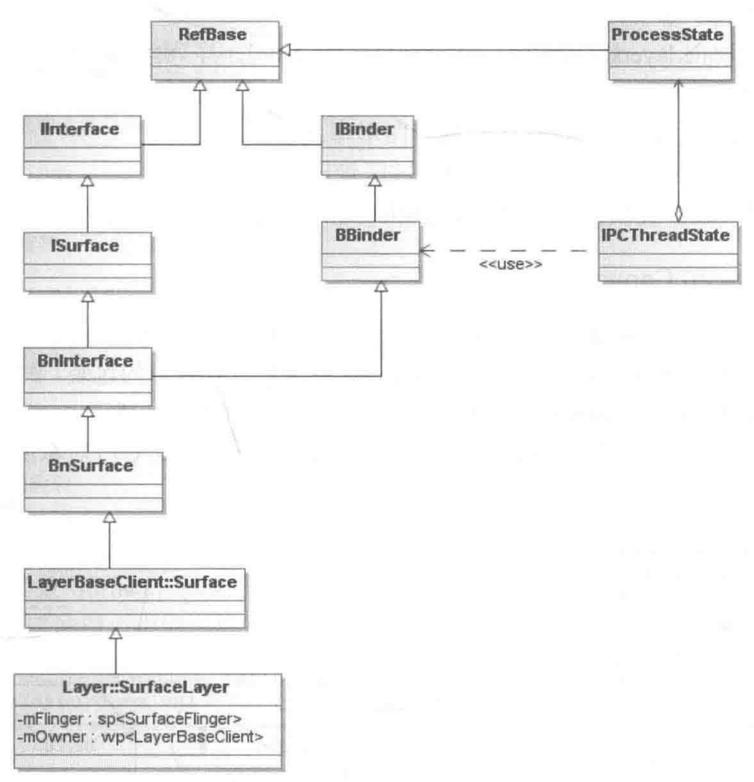
类 Layer 内部的成员变量 mUserClientRef 指向了一个 ClientRef 对象，这个 ClientRef 对象内部有一个成员变量 mControlBlock，它指向了一个 SharedBufferServer 对象。从前面 Android 应用程序与 SurfaceFlinger 服务之间的共享 UI 元数据（SharedClient）的创建过程可以知道，SharedBufferServer 类是用来在 SurfaceFlinger 服务这一侧描述一个 UI 元数据缓冲区堆栈的，即在 SurfaceFlinger 服务中，每一个绘图表面，即一个 Layer 对象，都关联一个 UI 元数据缓冲区堆栈。

在类 LayerBaseClient 的内部有一个类型为 LayerBaseClient::Surface 的弱指针，它引用了一个 Layer::SurfaceLayer 对象。此 Layer::SurfaceLayer 对象是一个 Binder 本地对象，是 SurfaceFlinger 服务用来与 Android 应用程序建立通信的，以便可以共同维护一个绘图表面。



▲图 10-1 Layer 类的实现

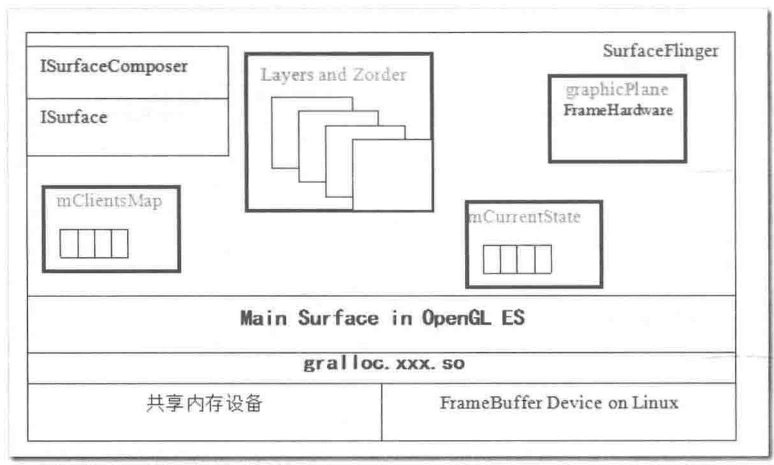
类 Layer::SurfaceLayer 继承于类 LayerBaseClient::Surface，其具体实现结构如图 10-2 所示。



▲图 10-2 类 Layer::SurfaceLayer 的实现

由图 10-2 所示的结构可以看出，类 Layer::SurfaceLayer 实现了 ISurface 接口，Android 应用程序正是通过这个接口和 SurfaceFlinger 服务共同维护一个绘图表面的。

在 Android 系统中，SufaceFlinger 的构成并不是太复杂，复杂的是它的客户端建构。SufaceFlinger 的构成框架如图 10-3 所示。



▲图 10-3 SufaceFlinger 的构成框架

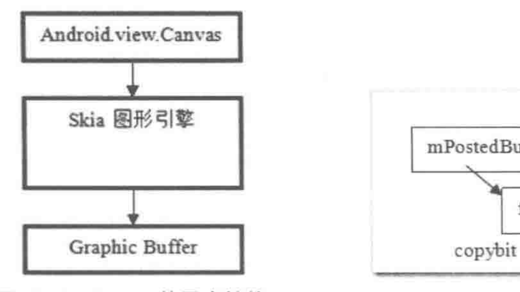
在 SurfaceFlinger 应用中，存在如下几个常用的管理对象。

- mClientsMap：管理客户端与服务端的连接。
- Isurface 和 IsurfaceComposer：AIDL 调用接口实例。
- mLayerMap：服务端的 Surface 的管理对象。
- mCurrentState.layersSortedByZ：以 Surface 的 Z-order 序列排列的 Layer 数组。
- graphicPlane：缓冲区输出管理。
- OpenGL ES：图形计算、图像合成等图形库。
- gralloc.xxx.so：跟平台相关的图形缓冲区管理器。
- pmem Device：提供共享内存，在 gralloc.xxx.so 可见，在上层被 gralloc.xxx.so 抽象了。

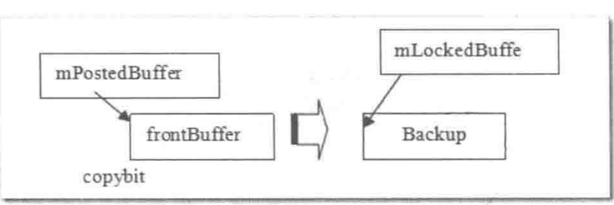
10.1.2 Surface 和 Canvas

在 Android 系统中，Surface 和 Canvas 密切相关。Canvas 是画布的意思，Android 上层的作图几乎都通过 Canvas 实例来完成。除此之外，Canvas 更多是一种接口的包装，例如接口 drawPaints、drawPoints、drawRect 和 drawBitmap。Canvas 的层次结构如图 10-4 所示。

Canvas（Java）在 C++的 Native 层，都有一个 Native Canvas 的 C++对象所对应，具体结构如图 10-5 所示。



▲图 10-4 Canvas 的层次结构



▲图 10-5 Native 层结构

通过 SurfaceLock 可以获取 Surface（mLockedBuffe）所对应的图形缓冲区地址，具体流程如下所示。

（1）建立与 SkCanvas 连接的位图设备，此位图使用上面取得的图形缓冲区地址做自己的位图内存。

（2）设置 SkCanvas 的作图目标设备为该位图。

这样，通过上述流程就建立起了 SurfaceControl 与 Canvas 之间的联系。

10.2 Surface 渲染详解

在 Android 系统的 SurfaceFlinger 渲染管理器中，提供了 4 种类型的 Layer 供用户选择。每个 Layer 对应一种类型的窗口，并且对应这种窗口相应的操作分别为 Layer、LayerBlur、LayerBuffer 和 LayerDim。LayerBuffer 很容易让人理解成为 Layer 的 Buffer，这实际上是一种 Layer 类型。各个 Layer 的效果可以参考文件 Surface.java 中的描述。在本节的内容中，将详细讲解 Surface 渲染的具体实现过程。

10.2.1 渲染类 Surface 详解

在 Android 系统中，文件 Surface.java 被保存在如下所示的目录中。

```
frameworks/base/core/java/android/view/
```

在接下来的内容中，将详细讲解文件 Surface.java 的具体实现过程。

（1）定义需要的常量，具体代码如下所示。

```
//一个矩阵的刻度，非兼容模式设置为空
private Matrix mCompatibleMatrix;
/**
 * 转动常数：0 度旋转（自然方向）
 */
public static final int ROTATION_0 = 0;

/**
 *转动常数：90 度旋转
 */
public static final int ROTATION_90 = 1;

/**
 *转动常数：180 度旋转
 */
public static final int ROTATION_180 = 2;

/**
 *转动常数：270 度旋转
 */
public static final int ROTATION_270 = 3;
```

（2）再看构造函数 Surface，其功能是新建一个 Canvas 画布，由此我们不难猜测，这块画布对应于 Surface 的创建过程。构造函数 Surface 的具体实现代码如下所示。

```
public Surface(SurfaceTexture surfaceTexture) {
    if (surfaceTexture == null) {
        throw new IllegalArgumentException("surfaceTexture must not be null");
    }

    synchronized (mLock) {
        mName = surfaceTexture.toString();
        try {
            setNativeObjectLocked(nativeCreateFromSurfaceTexture(surfaceTexture));
        } catch (OutOfResourcesException ex) {
            // We can't throw OutOfResourcesException because it would be an API change
            throw new RuntimeException(ex);
        }
    }
}
```

```
/* 从android_view_Surface_createFromIGraphicBufferProducer()执行操作 */
private Surface(int nativeObject) {
    synchronized (mLock) {
        setNativeObjectLocked(nativeObject);
    }
}
```

通过上述实现代码可知，Surface 的本质就是一块内存区。

（3）再看函数 release，其功能是释放本地 Surface，具体实现代码如下所示。

```
public void release() {
    synchronized (mLock) {
        if (mNativeSurface != 0) {
            nativeRelease(mNativeSurface);
            setNativeObjectLocked(0);
        }
    }
}
```

（4）再看函数 lockCanvas，其功能是获取 Canvas 对象，通过在 Canvas 上绘制内容修改 Surface 中的数据。函数 lockCanvas 的具体实现代码如下所示。

```
public Canvas lockCanvas(Rect inOutDirty)
        throws OutOfResourcesException, IllegalArgumentException {
    synchronized (mLock) {
        checkNotReleasedLocked();
        nativeLockCanvas(mNativeSurface, mCanvas, inOutDirty);
        return mCanvas;
    }
}
```

（5）再看函数 unlockCanvasAndPost，其功能是释放被锁定的 Canvas 对象，具体实现代码如下所示。

```
public void unlockCanvasAndPost(Canvas canvas) {
    if (canvas != mCanvas) {
        throw new IllegalArgumentException("canvas object must be the same instance that "
                + "was previously returned by lockCanvas");
    }

    synchronized (mLock) {
        checkNotReleasedLocked();
        nativeUnlockCanvasAndPost(mNativeSurface, canvas);
    }
}
```

在 Android 的处理机制中，当调用 lockCanvas 函数获取 Canvas 后，SurfaceView 会获取 Surface 的一个同步锁直到调用 unlockCanvasAndPost(Canvas canvas)函数才释放该锁，这里的同步机制保证在 Surface 绘制过程中不会被改变（被摧毁、修改）。

（6）再看函数 copyFrom，其功能是从 Android 系统中复制数组，并将内容拷贝到原生层的内容中来保存。函数 copyFrom 的具体实现代码如下所示。

```
public void copyFrom(SurfaceControl other) {
    if (other == null) {
        throw new IllegalArgumentException("other must not be null");
    }

    int surfaceControlPtr = other.mNativeObject;
    if (surfaceControlPtr == 0) {
        throw new NullPointerException(
                "SurfaceControl native object is null. Are you using a released
SurfaceControl?");
```

```
    }
    int newNativeObject = nativeCreateFromSurfaceControl(surfaceControlPtr);

    synchronized (mLock) {
        if (mNativeSurface != 0) {
            nativeRelease(mNativeSurface);
        }
        setNativeObjectLocked(newNativeObject);
    }
}
```

（7）再看函数 transferFrom，其功能是复制来替代 SurfaceView 中原来的 Surface。函数 transferFrom 的具体实现代码如下所示。

```
public void transferFrom(Surface other) {
    if (other == null) {
        throw new IllegalArgumentException("other must not be null");
    }
    if (other != this) {
        final int newPtr;
        synchronized (other.mLock) {
            newPtr = other.mNativeSurface;
            other.setNativeObjectLocked(0);
        }

        synchronized (mLock) {
            if (mNativeSurface != 0) {
                nativeRelease(mNativeSurface);
            }
            setNativeObjectLocked(newPtr);
        }
    }
}
```

（8）最后看函数 rotationToString，其功能是根据旋转角度进行对应的处理，具体实现代码如下所示。

```
public static String rotationToString(int rotation) {
    switch (rotation) {
        case Surface.ROTATION_0: {
            return "ROTATION_0";
        }
        case Surface.ROTATION_90: {
            return "ROATATION_90";
        }
        case Surface.ROTATION_180: {
            return "ROATATION_180";
        }
        case Surface.ROTATION_270: {
            return "ROATATION_270";
        }
        default: {
            throw new IllegalArgumentException("Invalid rotation: " + rotation);
        }
    }
}
```

10.2.2 分析 Layer 和 LayerBuffer

在 SurfaceFlinger 渲染管理器中提供了 4 种类型的 Layer，其中最为重要的是 Layer(Norm Layer) 和 LayerBuffer。

（1）Norm Layer。

Norm Layer 是在 Android 中使用最多的一种 Layer，一般的应用程序在创建 Surface 的时候都是采用的这样的 Layer，了解 Normal Layer 可以让我们知道 Android 进行 Display 过程中的一些基

础原理。Normal Layer 为每个 Surface 分配两个 Buffer：Front Buffer 和 Back Buffer，这个前后是相对的概念，它们是可以进行 Flip 的。Front Buffer 用于 SurfaceFlinger 进行显示，而 Back Buffer 用于应用程序进行画图。当 Back Buffer 填满数据（Dirty）以后，就会替换（Flip），Back Buffer 就变成了 Front Buffer 用于显示，而 Front Buffer 就变成了 Back Buffer 用来画图，这两个 Buffer 的大小是根据 Surface 的大小格式而动态变化的。这个动态变化在文件 frameworks/native/services/surfaceflinger/layer.cpp 的函数 setbuffers 中实现，具体实现代码如下所示。

```
status_t Layer::setBuffers( uint32_t w, uint32_t h,
                            PixelFormat format, uint32_t flags)
{
    // this surfaces pixel format
    PixelFormatInfo info;
    status_t err = getPixelFormatInfo(format, &info);
    if (err) {
        ALOGE("unsupported pixelformat %d", format);
        return err;
    }

    uint32_t const maxSurfaceDims = min(
            mFlinger->getMaxTextureSize(), mFlinger->getMaxViewportDims());

    // never allow a surface larger than what our underlying GL implementation
    // can handle
    if ((uint32_t(w)>maxSurfaceDims) || (uint32_t(h)>maxSurfaceDims)) {
        ALOGE("dimensions too large %u x %u", uint32_t(w), uint32_t(h));
        return BAD_VALUE;
    }

    mFormat = format;

    mSecure = (flags & ISurfaceComposerClient::eSecure) ? true : false;
    mProtectedByApp = (flags & ISurfaceComposerClient::eProtectedByApp) ? true : false;
    mOpaqueLayer = (flags & ISurfaceComposerClient::eOpaque);
    mCurrentOpacity = getOpacityForFormat(format);

    mSurfaceFlingerConsumer->setDefaultBufferSize(w, h);
    mSurfaceFlingerConsumer->setDefaultBufferFormat(format);
    mSurfaceFlingerConsumer->setConsumerUsageBits(getEffectiveUsage(0));

    return NO_ERROR;
}
```

在上述代码中，参数 format 是一个整数值，用来描述要创建的 Surface 的像素格式。函数 setbuffers 首先调用另外一个函数 getPixelFormatInfo 将它转换为一个 PixelFormatInfo 对象 info，以便可以获得更多的该种类型的像素格式的信息。例如一个像素点占多少字节，每个颜色分量又分别占多少位等。函数 getPixelFormatInfo 在文件 frameworks/native/libs/ui/PixelFormat.cpp 中定义，具体实现代码如下所示。

```
status_t getPixelFormatInfo(PixelFormat format, PixelFormatInfo* info)
{
    if (format <= 0)
        return BAD_VALUE;

    if (info->version != sizeof(PixelFormatInfo))
        return INVALID_OPERATION;

    // YUV format from the HAL are handled here
    switch (format) {
    case HAL_PIXEL_FORMAT_YCbCr_422_SP:
    case HAL_PIXEL_FORMAT_YCbCr_422_I:
        info->bitsPerPixel = 16;
        goto done;
    case HAL_PIXEL_FORMAT_YCrCb_420_SP:
```

```
    case HAL_PIXEL_FORMAT_YV12:
        info->bitsPerPixel = 12;
     done:
        info->format = format;
        info->components = COMPONENT_YUV;
        info->bytesPerPixel = 1;
        info->h_alpha = 0;
        info->l_alpha = 0;
        info->h_red = info->h_green = info->h_blue = 8;
        info->l_red = info->l_green = info->l_blue = 0;
        return NO_ERROR;
    }

    size_t numEntries;
    const Info *i = gGetPixelFormatTable(&numEntries) + format;
    bool valid = uint32_t(format) < numEntries;
    if (!valid) {
        return BAD_INDEX;
    }

    info->format = format;
    info->bytesPerPixel = i->size;
    info->bitsPerPixel  = i->bitsPerPixel;
    info->h_alpha       = i->ah;
    info->l_alpha       = i->al;
    info->h_red        = i->rh;
    info->l_red        = i->rl;
    info->h_green       = i->gh;
    info->l_green       = i->gl;
    info->h_blue        = i->bh;
    info->l_blue        = i->bl;
    info->components    = i->components;
    return NO_ERROR;
}
```

由此可见，Front Buffer 和 Back Buffer 这两个 Buffer Flip 方式是 Android 显示系统中的一个重要实现方式，不只是每个 Surface 都得这么实现，最后写入 FB 的 Main Surface 也是采用这种方式实现的。

（2）LayerBuffer。

在 Android 系统中，LayerBuffer 是将来必定会用到的一个 Layer，同时也是一个比较复杂的一个 Layer。LayerBuffer 不具备 Render Buffer 功能，主要用在 Camera Preview（照相预览）和 Video Playback（视频回放）上。LayerBuffer 提供了如下两种实现方式。

- 一种是 Post Buffer；
- 一种是 Overlay，Overlay 的接口实际上就是在这个 Layer 上实现的。

无论是 Overlay 方式还是 Post Buffer 方式，都是指这个 Layer 的数据来源于其他地方，只是 Post Buffer 是通过软件的方式实现的，而 Overlay 则是通过硬件 Merge 方式来实现。与此 Layer 紧密联系在一起的是 ISurface 接口，通过它来注册数据来源。

当窗口的内容或者状态发生变化时，Surface Flinger 就会监听到相关的时间，并进行如下所示的处理。

- 首先判断事件是不是控制台信号，如果是就调用函数 handleConsoleEvents 进行控制台信号的处理，获取渲染的区域。函数 handleConsoleEvents 在文件 frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp 中定义，主要实现代码如下所示。

```
void SurfaceFlinger::handleConsoleEvents()
{
    // something to do with the console
    const DisplayHardware& hw = graphicPlane(0).displayHardware();
```

```
    int what = android_atomic_and(0, &mConsoleSignals);
    ......

    if (mDeferReleaseConsole && hw.isScreenAcquired()) {
        // We got the release signal before the acquire signal
        mDeferReleaseConsole = false;
        hw.releaseScreen();
    }

    if (what & eConsoleReleased) {
        if (hw.isScreenAcquired()) {
            hw.releaseScreen();
        } else {
            mDeferReleaseConsole = true;
        }
    }

    mDirtyRegion.set(hw.bounds());
}
```

● 当交易处理完成时，SurfaceFlinger 会在函数 handleTransaction 中遍历 Z-order 上的 Layer，移去已经被销毁的 Layer。函数 handleTransaction 在文件 frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp 中定义，主要实现代码如下所示。

```
void SurfaceFlinger::handleTransaction(uint32_t transactionFlags)
{
    ATRACE_CALL();

    Mutex::Autolock _l(mStateLock);
    const nsecs_t now = systemTime();
    mDebugInTransaction = now;

    // Here we're guaranteed that some transaction flags are set
    // so we can call handleTransactionLocked() unconditionally
    // We call getTransactionFlags(), which will also clear the flags,
    // with mStateLock held to guarantee that mCurrentState won't change
    // until the transaction is committed

    transactionFlags = getTransactionFlags(eTransactionMask);
    handleTransactionLocked(transactionFlags);

    mLastTransactionTime = systemTime() - now;
    mDebugInTransaction = 0;
    invalidateHwcGeometry();
    // here the transaction has been committed
}
```

因为 Layer 的混合是在线程中进行的，在混合的过程中，应用程序或者系统可能会改变 Layer 的状态，例如屏幕旋转、增加/删除 Layer、某个 Layer 可见或不可见。为了使这些变动不会破坏当前正在进行的混合动作，SurfaceFlinger 维护着如下两个 Layer 列表。

➢ mCurrentState.layersSortedByZ：当前系统最新的 Layer 列表。

➢ mDrawingState.layersSortedByZ：本次混合操作使用的 Layer 列表。

函数 handleTransaction 就是根据 Layer 列表的这些状态的变化，计算是否有可见区域内需要更新，并设置状态变量 mVisibleRegionsDirty，然后把 mCurrentState 赋值给 mDrawingState，最后释放已经被丢弃（ditch）的 Layer。

● 接下来 SurfaceFlinger 会遍历 LayerVector，并计算每个 Layer 的可视区域。如果需要重绘，则在函数 handleRepaint 中进行 Layer 组合。也就是说，函数 handleRepaint 能够将各个 Surface 的图形缓冲区合成起来，以便接下来可以渲染到硬件帧缓冲区中去。函数 handleRepaint 的具体实现代码如下所示。

```
void SurfaceFlinger::handleRepaint()
{
    // compute the invalid region
    mInvalidRegion.orSelf(mDirtyRegion);
    // Skip this check for original resolution  and layerbuffer surfaces, since MDP is
    // used for display and we want to ensure UI updates
    if (mInvalidRegion.isEmpty() && mOrigResSurfAbsent && !mIsLayerBufferPresent) {
        // nothing to do
        return;
    }

    if (UNLIKELY(mDebugRegion)) {
        debugFlashRegions();
    }

    // set the frame buffer
    const DisplayHardware& hw(graphicPlane(0).displayHardware());
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    uint32_t flags = hw.getFlags();
    //Enter block only if original resolution surface absent
    //If present, ensures that the entire region is marked dirty
    if  ((flags  &  DisplayHardware::SWAP_RECTANGLE  &&  !mIsLayerBufferPresent  &&
mOrigResSurfAbsent) ||
        (flags & DisplayHardware::BUFFER_PRESERVED))
    {
        // we can redraw only what's dirty, but since SWAP_RECTANGLE only
        // takes a rectangle, we must make sure to update that whole
        // rectangle in that case
        if (flags & DisplayHardware::SWAP_RECTANGLE) {
            // TODO: we really should be able to pass a region to
            // SWAP_RECTANGLE so that we don't have to redraw all this
            mDirtyRegion.set(mInvalidRegion.bounds());
        } else {
            // in the BUFFER_PRESERVED case, obviously, we can update only
            // what's needed and nothing more
            // NOTE: this is NOT a common case, as preserving the backbuffer
            // is costly and usually involves copying the whole update back
        }
    } else {
        if (flags & DisplayHardware::PARTIAL_UPDATES) {
            // We need to redraw the rectangle that will be updated
            // (pushed to the framebuffer)
            // This is needed because PARTIAL_UPDATES only takes one
            // rectangle instead of a region (see DisplayHardware::flip())
            mDirtyRegion.set(mInvalidRegion.bounds());
        } else {
            // we need to redraw everything (the whole screen)
            mDirtyRegion.set(hw.bounds());
            mInvalidRegion = mDirtyRegion;
        }
    }

    // compose all surfaces
    composeSurfaces(mDirtyRegion);

    // clear the dirty regions
    mDirtyRegion.clear();
}
```

通过上述代码可知，函数 handleRepaint 首先重置了 openGL 的观察矩阵，然后遍历 mDrawingState.layersSortedByZ 中的 Layer 列表，调用每个 Layer 的 onDraw 方法。在 onDraw 方法中，会调用 drawWithOpenGL 将在 handlePageFlip 阶段生成的贴图混合到 OpenGL 的主表面。最后 handleRepaint 把需要刷新的区域清除。由此可见，函数 handleRepaint 真正实现了 Layer 混合的阶段。

● 当组合完成后会调用各个 Layer 的 bootFinished 函数，WindowManagerService 通过函数 bootFinished 来告诉 SurfaceFlinger 服务系统启动完成了，这时候 SurfaceFlinger 服务就会停止执行开机动画。函数 bootFinished 的具体实现代码如下所示。

```
void SurfaceFlinger::bootFinished()
{
    const nsecs_t now = systemTime();
    const nsecs_t duration = now - mBootTime;
    ALOGI("Boot is finished (%ld ms)", long(ns2ms(duration)) );
    mBootFinished = true;

    // wait patiently for the window manager death
    const String16 name("window");
    sp<IBinder> window(defaultServiceManager()->getService(name));
    if (window != 0) {
        window->linkToDeath(static_cast<IBinder::DeathRecipient*>(this));
    }

    // stop boot animation
    // formerly we would just kill the process, but we now ask it to exit so it
    // can choose where to stop the animation
    property_set("service.bootanim.exit", "1");
}
```

另外，在 Surface 中还有一个十分重要的概念：Layer_state_t，其功能是根据窗口状态的变化表示 Layer 处于不同的状态，在 layer_state_t 中包括如下所示的值。

- ePositionChanged；
- eLayerChanged；
- eSizeChanged；
- eAlphaChanged；
- eMatrixChanged；
- eTransparentRegionChanged；
- eVisibilityChanged。

当发生 eLayerChanged 变化时说明需要切换缓冲，并且无论 layer_state_t 值发生了何种变化，都需要重新遍历 Layer，并进行相应的重绘操作。

通常来说，屏幕的渲染过程需要重绘 Surface 的所有像素。但是出于提供效率和速率的需要，在具体设计过程中会先进行对比操作，只会重绘发生变化的区域。这部分的内容主要位于文件 framebuffer.cpp 的函数 fb_post 中，并且在 LayerBuffer 和 LayerDim 模式的 Layer 中也有所采用。

10.3 Skia 渲染引擎详解

在 Android 中的画图过程分为 2D 和 3D 两种，其中 2D 图形是由 Skia 来实现的，2D 图形的渲染功能也是由 Skia 实现的。在本节的内容中，将详细讲解 Skia 的基本知识，为读者进入本书后面知识的学习打下基础。

10.3.1 Skia 基础

Android 系统使用 Skia 作为其核心图形引擎，Google Chrome 的图形引擎也是 Skia。Skia 图形渲染引擎最初由 Skia 公司开发，该公司于 2005 年被 Google 收购。Skia 与 Openwave's（现在叫 Purple Labs）V7 Vector Graphics Engine 非常类似，它们都来自于 Mike Reed 的公司。要想了解 Skia 的重要性，需要先了解如下两个问题。

（1）为什么不用 OpenGL 或者 DirectX 来加速渲染？

● 数据从 Video Card 读出后，在另一个进程中再拷贝回 Video Card，这种情况下不值得用硬件加速渲染。

● 相对而言，Skia 实现图形绘制只用很少时间，大部分时间是计算页面元素的位置、风格、输出文本，即使用了 3D 加速也不会有明显改善。

（2）为什么不用别的图形库？

当前市面中，有如下 3 个最为常用的图形库。

● Windows GDI：Microsoft Windows 的底层图形 API，相对而言只具备基本的绘制功能，像 <canvas>和 SVG 需要单独实现。

● GDI+：Windows 上更高级的 API，GDI+ 使用的是设备独立的 metrics，这会使 Chrome 中的 text and letter spacing 看以来与别的 Windows 应用不同。而且微软当时也推荐开发人员使用新的图形 API，GDI+ 的技术支持和维护可能有限。

● Cairo：一个开源 2D 图形库，已经在 Firefox 3 中开始使用。

在 Android 系统中，选择 Skia 渲染的原因有如下 3 点。

● Skia 是一个跨平台的应用程序和 UI 框架。

● Skia 拥有优质的 WebKit 接口，使用它可以为 Android 浏览器提供高质量的效果。

● Skia 拥有机构内部的专门技术，这些技术都是在领域中的尖端技术。

10.3.2 Android 中的 Skia

熟悉 Windows 编程的读者应该知道 GDI+是 Windows 的一套图形绘制库。也就是说 Windows 系统下，所有图形图像绘制最终都是通过 GDI 来实现的。同样，在 Android 下也需要一套能绘制出点、线、面等复杂图形的工具，或者渲染界面等图像方面的一个可供开发者调用的绘图函数接口，Skia 满足了这些要求。

究竟 Skia 是什么？举个例子，假设现在让你来画一幅国画——山水画，需要哪些工具呢？需要一张纸，比如白纸，或者带有某些背景图的纸张，并且需要不同型号的毛笔，墨汁和颜料等。然后规定在纸张的哪个区域画图，用什么样的毛笔，用什么样的颜色，画什么样的图形，是点、线、面还是花草等。

Skia 是什么？Skia 就是类似上面画图所有需要的一系列设备、工具等。要在 Android 里画图或者渲染图像，都需要 Skia 提供的 API 接口，或者是间接提供。

在 Android 应用程序中不会看到或者用到 Skia 函数，这是因为我们不需要直接控制图形绘制，没有实现绘图类的应用，所以没有用到这方面的函数。但是应用中的所有 Activity、View 或者其他控件的显示，都是在底层通过 Skia 提供的函数进行显示的。

读者无需对 Skia 的实现原理进行深入了解，在此之所以介绍 Skia，目的是为了后面介绍 Android 下 OpenGL ES 方面的知识做准备。因为在 Android 系统下，通过 OpenGL ES 绘制的 3D 图形，最终还是被合并到 Skia 定义的显示缓存中进行显示的。

在 Android 系统中，Skia 引擎所处的位置如图 10-6 所示。

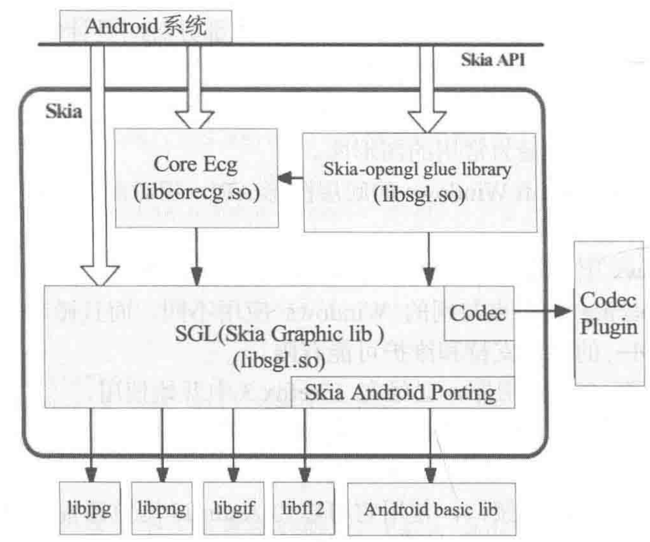
在 Android 系统的开源代码中，Skia 模块的目录位置如下所示。

（1）头文件：即 Internal API，位置是“android/external/skia/include”目录，在里面还包含 animator、core、effects、images 和 views 等几个子目录，其中最重要的就是“core”目录。

（2）源文件：位于“android/external/skia/src”目录，子目录结构和头文件目录相同。

（3）封装层：Android 对 Skia 引擎进行了封装，以便让 Java 代码方便地调用，对 Skia 封装的代码保存在“android/framework/base/core/jni”目录以及“android/framework/base/core/jni/android/

graphics”目录下面，主要功能是对 Canvas、Bitmap、Graphics 和 Picture 等进行封装，以及和 libui 库结合使用。



▲图 10-6　Skia 引擎所处的位置

1. 库

在 Skia 引擎中，主要包括如下所示的 3 个库。

- libcorecg.so：包含“/skia/src/core”目录的部分内容，比如其中的 Region。在 SurfaceFlinger 中，Rect 是用来计算可视区域的。
- libsgl.so：包含“/skia/src/core|effects|images|ports|utils”等目录中的部分和全部内容，在此实现了 Skia 大部分的图形效果，以及图形格式的编解码。
- libskiagl.so：包含“/skia/src/gl”目录的内容，功能是调用 OpenGL 实现三维效果。

2. Skia 对上层的接口（API）

Skia 的源文件及部分头文件都保存在“external/skia/src”目录下，导出的头文件保存在“external/skia/include”目录下。其中最主要的是类 SKCanvas，几乎整个 Android GUI 系统的底层绘制都是由这个类来完成的。其头文件和源代码文件的路径分别位于下面的文件中。

- external/skia/include/core/SKCanvas.h；
- external/skia/src/core/SKCanvas.cpp。

在类 SKCanvas 中，主要包含如下所示的 3 种绘制功能。

- 基本图形绘制：例如函数 drawARGB 和函数 drawLine。其中函数 drawARGB 在文件 SKCanvas.h 中的原型如下所示。

```
void drawARGB(U8CPU a, U8CPU r, U8CPU g, U8CPU b,
        SkXfermode::Mode mode = SkXfermode::kSrcOver_Mode);
```

函数 drawARGB 具体功能是在文件 SKCanvas.cpp 中实现的，功能是绘制一个指定填充样式的矩形。函数 drawARGB 的具体实现代码如下所示。

```
void SkCanvas::drawARGB(U8CPU a, U8CPU r, U8CPU g, U8CPU b,
                SkXfermode::Mode mode) {
```

```
    SkPaint paint;

    paint.setARGB(a, r, g, b);
    if (SkXfermode::kSrcOver_Mode != mode) {
        paint.setXfermodeMode(mode);
    }
    this->drawPaint(paint);
}
```

而函数 drawLine 的功能是绘制指定颜色的直线，在文件 SKCanvas.cpp 中的具体实现代码如下所示。

```
void SkCanvas::drawLine(SkScalar x0, SkScalar y0, SkScalar x1, SkScalar y1,
                        const SkPaint& paint) {
    SkPoint pts[2];
    pts[0].set(x0, y0);
    pts[1].set(x1, y1);
    this->drawPoints(kLines_PointMode, 2, pts, paint);
}
```

- 图像文件绘制：例如函数 drawBitmap。此函数的功能是在画布中绘制一幅图像，在文件 SKCanvas.cpp 中定义函数 drawBitmap 的具体实现代码如下所示。

```
void SkCanvas::drawBitmap(const SkBitmap& bitmap, SkScalar x, SkScalar y,
                          const SkPaint* paint) {
    SkDEBUGCODE(bitmap.validate();)

    if (NULL == paint || paint->canComputeFastBounds()) {
        SkRect bounds = {
            x, y,
            x + SkIntToScalar(bitmap.width()),
            y + SkIntToScalar(bitmap.height())
        };
        if (paint) {
            (void)paint->computeFastBounds(bounds, &bounds);
        }
        if (this->quickReject(bounds)) {
            return;
        }
    }

    SkMatrix matrix;
    matrix.setTranslate(x, y);
    this->internalDrawBitmap(bitmap, NULL, matrix, paint);
}
```

- 文本绘制：例如函数 drawText。此函数的功能是在画布中绘制指定的文本，在文件 SKCanvas.cpp 中定义函数 drawText 的具体实现代码如下所示。

```
void SkCanvas::drawText(const void* text, size_t byteLength,
                        SkScalar x, SkScalar y, const SkPaint& paint) {
    CHECK_SHADER_NOSETCONTEXT(paint);

    LOOPER_BEGIN(paint, SkDrawFilter::kText_Type)

    while (iter.next()) {
        SkDeviceFilteredPaint dfp(iter.fDevice, looper.paint());
        iter.fDevice->drawText(iter, text, byteLength, x, y, dfp.paint());
        DrawTextDecorations(iter, dfp.paint(),
                            static_cast<const char*>(text), byteLength, x, y);
    }

    LOOPER_END
}
```

3. Skia 图像编解码部分

在 Android 系统中，Skia 图像“编/解码”部分的接口功能是由如下两个文件实现的。

● external/include/image/SKImageDecoder.h：把图像文件或者流解码到 Skia 的内部内存 SKBitmap 中。在此文件中定义了和解码相关的函数，也定义了格式枚举。例如定义格式枚举的代码如下所示。

```
enum Format {
    kUnknown_Format,
    kBMP_Format,
    kGIF_Format,
    kICO_Format,
    kJPEG_Format,
    kPNG_Format,
    kWBMP_Format,
    kWEBP_Format,

    kLastKnownFormat = kWEBP_Format
};
```

● external/include/image/SKImageEncoder.h：把 Skia 内部内存 SKBitmap 编码成文件或流的形式。在此文件中定义了和编码相关的函数，也定义了格式枚举。例如定义格式枚举的代码如下所示。

```
enum Type {
    kJPEG_Type,
    kPNG_Type,
    kWEBP_Type
};
static SkImageEncoder* Create(Type);
```

上述接口需要具体的实现类的主要实现代码保存在“src/image”文件中。

4. Android 图形系统的 JNI 接口

在 Android 系统中，JNI 接口主要提供了从 Skia 底层库到 Java 上层的支持，具体实现代码主要保存在如下所示的文件中。

```
frameworks/base/core/jni/android/graphic/Canvas.cpp
```

在文件 Canvas.cpp 中定义了和画布操作有关的接口函数，例如以下绘制点和直线的接口函数。

```
static void drawPoints(JNIEnv* env, jobject jcanvas, jfloatArray jptsArray,
                       jint offset, jint count, jobject jpaint) {
    doPoints(env, jcanvas, jptsArray, offset, count, jpaint,
             SkCanvas::kPoints_PointMode);
}

static void drawLines(JNIEnv* env, jobject jcanvas, jfloatArray jptsArray,
                      jint offset, jint count, jobject jpaint) {
    doPoints(env, jcanvas, jptsArray, offset, count, jpaint,
             SkCanvas::kLines_PointMode);
}

static void drawPoint(JNIEnv* env, jobject jcanvas, float x, float y,
                      jobject jpaint) {
    NPE_CHECK_RETURN_VOID(env, jcanvas);
    NPE_CHECK_RETURN_VOID(env, jpaint);
    SkCanvas* canvas = GraphicsJNI::getNativeCanvas(env, jcanvas);
    const SkPaint& paint = *GraphicsJNI::getNativePaint(env, jpaint);

    canvas->drawPoint(SkFloatToScalar(x), SkFloatToScalar(y), paint);
}
```

5. Android 的图形包（Graphics）

Android 图形类的包是 android.graphics，它通过调用图形系统的 JNI 提供了对 Java 框架中图形系统的支持。在 Android 的 Java 框架和 Java 应用程序中，2D 绘制的功能（基本图形、图片文件、字）也是通过调用 Graphics 来实现的，其主要实现代码位于如下所示的目录中。

```
frameworks/base/graphics/java/android/graphics/
```

在上述目录中，在文件 Canvas.java 中实现了 Android 图形系统中最重要的类 android.graphic.canvas，定义此类的主要实现代码如下所示。

```
public class Canvas {
    int mNativeCanvas;
    private Bitmap mBitmap;
    private DrawFilter mDrawFilter;
```

6. Android 2D 图形硬件加速

在 Android 系统中，Android 2D 图形硬件加速主要是通过 Copybit 模块来实现。Copybit 是封装在 Android 中 OpenGL 软件实现库（libagl）的一部分，仅对 OpenGL ES 2D API 进行了封装，实现 OpenGL ES 2D API 到硬件的加速功能。

在 Android 系统中，Copybit 模块以 HAL 的形式实现，其具体实现文件如下所示。

```
hardware/msm7k/libcopybit/copybit.c
```

要想完全理解类 SkBitmap，可以从其实现源文件入手。类 SkBitmap 的实现源文件是 SkBitmap.h，主要实现代码如下所示。

```
class SkBitmap {
public:
    class Allocator;
    enum Config {
        kNo_Config,
        kA1_Config,
        kA8_Config,
        kIndex8_Config,
        kRGB_565_Config,
        kARGB_4444_Config,
        kARGB_8888_Config,
        kRLE_Index8_Config,
        kConfigCount
    };
    SkBitmap();
        but ownership of the pixels remains with the src bitmap.
    SkBitmap(const SkBitmap& src);
    ~SkBitmap();
        with the src bitmap.
    SkBitmap& operator=(const SkBitmap& src);
    void swap(SkBitmap& other);
    //把当前的 SkBitmap 和另外的 SkBitmap other 做交换
    bool empty() const { return 0 == fWidth || 0 == fHeight; }
    //是否有宽高
    bool isNull() const { return NULL == fPixels && NULL == fPixelRef; }
    //是否有像素
    Config  config() const { return (Config)fConfig; }
    Config  getConfig() const { return this->config(); }
    int width() const { return fWidth; }
    int height() const { return fHeight; }
    int rowBytes() const { return fRowBytes; }
    //位图的宽、高、格式，以及每行的字节数
    int shiftPerPixel() const { return fBytesPerPixel >> 1; }
    int bytesPerPixel() const { return fBytesPerPixel; }
```

```
    //每像素的字节数
       height). Note, for 1-byte per pixel configs like kA8_Config, this will
       return the same as rowBytes(). Is undefined for configs that are less
       than 1-byte per pixel (e.g. kA1_Config)
    int rowBytesAsPixels() const { return fRowBytes >> (fBytesPerPixel >> 1); }
    //一行的像素数
    void* getPixels() const { return fPixels; }
    //位图数据的起始地址
    //获得像素的字节空间大小，如果超过 32 位 ( 即 4294967296 ) 则会截断，也就是只取最后的 32 位
    //可以使用 getSize64()函数代替，不过会稍微慢一点
    Sk64 getSize64() const {
       Sk64 size;
       size.setMul(fHeight, fRowBytes);
       return size;
    }
    bool isOpaque() const;
    void setIsOpaque(bool);
    //根据当前的格式判断是否为不透明的
    owner of the pixels, that ownership is decremented.
    void reset();
    //清除当前位图状态，释放位图
    static int ComputeRowBytes(Config c, int width);
    static int ComputeBytesPerPixel(Config c);
     not at least 1-byte per pixel, return 0, including for kNo_Config.
    static int ComputeShiftPerPixel(Config c) {
       return ComputeBytesPerPixel(c) >> 1;
    }
    static Sk64 ComputeSize64(Config, int width, int height);
    static size_t ComputeSize(Config, int width, int height);
    //根据参数计算一些需要的值
    void setConfig(Config, int width, int height, int rowBytes = 0);
    //设置格式
    void setPixels(void* p, SkColorTable* ctable = NULL);
    //使用已有的图像数据作为当前位图的数据，这里要注意的是如何确保图像数据和宽、高、格式等的匹配
    bool allocPixels(SkColorTable* ctable = NULL) {
       return this->allocPixels(NULL, ctable);
    }
    bool allocPixels(Allocator* allocator, SkColorTable* ctable);
    //此函数很重要，创建位图数据参考计数器。因为只有有了参考计数器，才能对位图的多重引用产生作用
    //注意，这里也会分配位图的内存空间。空间的大小是当前图像配置的大小。这里同样可以看到，对位图数据
访问时需要 lock 锁定和 unlock 解锁
    SkPixelRef* pixelRef() const { return fPixelRef; }
    size_t pixelRefOffset() const { return fPixelRefOffset; }
    SkPixelRef* setPixelRef(SkPixelRef* pr, size_t offset = 0);
    void lockPixels() const;
    void unlockPixels() const;
    //在访问时对图像数据进行锁定
       it has non-null pixels, and if required by its config, it has a
       non-null colortable. Returns true if all of the above are met.
    bool readyToDraw() const {
      return this->getPixels() != NULL &&
             ((this->config()!= kIndex8_Config && this->config()!=kRLE_Index8_Config) ||
                    fColorTable != NULL);
    }
    //确保当前 SkBitmap 可以用来绘图了
       reference count.
    SkColorTable* getColorTable() const { return fColorTable; }
       pixelref, or 0 if we do not have a pixelref. Each time the pixels are
       changed (and notifyPixelsChanged is called), a different generation ID
       will be returned.
    uint32_t getGenerationID() const;
       turn cause a different generation ID value to be returned from
       getGenerationID().
    void notifyPixelsChanged() const;
    //通知图像有改变
       for the bitmap's config. If the config is kRGB_565_Config, then the alpha value
is ignored.
       If the config is kA8_Config, then the r,g,b parameters are ignored.
    void eraseARGB(U8CPU a, U8CPU r, U8CPU g, U8CPU b) const;
```

```
        for the bitmap's config. If the config is kRGB_565_Config, then the alpha value
is presumed
        to be 0xFF. If the config is kA8_Config, then the r,g,b parameters are ignored
and the
        pixels are all set to 0xFF.
    void eraseRGB(U8CPU r, U8CPU g, U8CPU b) const {
        this->eraseARGB(0xFF, r, g, b);
    }
        for the bitmap's config. If the config is kRGB_565_Config, then the color's alpha
value is presumed
        to be 0xFF. If the config is kA8_Config, then only the color's alpha value is used.
    void eraseColor(SkColor c) const {
        this->eraseARGB(SkColorGetA(c), SkColorGetR(c), SkColorGetG(c),
                SkColorGetB(c));
    }
    //用指定的颜色擦除
        no pixels allocated (i.e. getPixels() returns null) the method will
        still update the inval region (if present).
        @param subset The subset of the bitmap to scroll/move. To scroll the
                entire contents, specify [0, 0, width, height] or just
                pass null.
        @param dx The amount to scroll in X
        @param dy The amount to scroll in Y
        @param inval Optional (may be null). Returns the area of the bitmap that
                was scrolled away. E.g. if dx = dy = 0, then inval would
                be set to empty. If dx >= width or dy >= height, then
                inval would be set to the entire bounds of the bitmap.
        @return true if the scroll was doable. Will return false if the bitmap
                uses an unsupported config for scrolling (only kA8,
                kIndex8, kRGB_565, kARGB_4444, kARGB_8888 are supported).
                If no pixels are present (i.e. getPixels() returns false)
                inval will still be updated, and true will be returned.
    bool scrollRect(const SkIRect* subset, int dx, int dy,
            SkRegion* inval = NULL) const;
        check to know the size of the pixels, and will return the same answer
        as the corresponding size-specific method (e.g. getAddr16). Since the
        check happens at runtime, it is much slower than using a size-specific
        version. Unlike the size-specific methods, this routine also checks if
        getPixels() returns null, and returns that. The size-specific routines
        perform a debugging assert that getPixels() is not null, but they do
        not do any runtime checks.
    //获得指定点的地址
    inline uint32_t* getAddr32(int x, int y) const;
    inline uint8_t* getAddr8(int x, int y) const;
        for 1bit pixels.
    inline uint8_t* getAddr1(int x, int y) const;
        colortable based bitmaps.
    inline SkPMColor getIndex8Color(int x, int y) const;
        pixel memory, and just point into a subset of it. However, if the config
        does not support this, a local copy will be made and associated with
        the dst bitmap. If the subset rectangle, intersected with the bitmap's
        dimensions is empty, or if there is an unsupported config, false will be
        returned and dst will be untouched.
        @param dst  The bitmap that will be set to a subset of this bitmap
        @param subset The rectangle of pixels in this bitmap that dst will
        @return true if the subset copy was successfully made.
    bool extractSubset(SkBitmap* dst, const SkIRect& subset) const;
        setting the new bitmap's config to the one specified, and then copying
        this bitmap's pixels into the new bitmap. If the conversion is not
        supported, or the allocator fails, then this method returns false and
        dst is left unchanged.
        @param dst  The bitmap to be sized and allocated
        @param c The desired config for dst
        @param allocator Allocator used to allocate the pixelref for the dst
                bitmap. If this is null, the standard HeapAllocator
                will be used.
        @return true if the copy could be made.
    bool copyTo(SkBitmap* dst, Config c, Allocator* allocator = NULL) const;
    bool hasMipMap() const;
```

```
    void buildMipMap(bool forceRebuild = false);
    void freeMipMap();
    //在做绘图时使用了 MipMap 结构。这个结构可以防止拷贝
       bitmap, and return it (this is the amount to shift matrix iterators
       by). If dst is not null, it is set to the correct level.
    int extractMipLevel(SkBitmap* dst, SkFixed sx, SkFixed sy);
    void extractAlpha(SkBitmap* dst) const {
       this->extractAlpha(dst, NULL, NULL);
    }
    void extractAlpha(SkBitmap* dst, const SkPaint* paint,
                      SkIPoint* offset) const;
    void flatten(SkFlattenableWriteBuffer&) const;
    void unflatten(SkFlattenableReadBuffer&);
    SkDEBUGCODE(void validate() const;)
    class Allocator : public SkRefCnt {
    public:
          config. Return true on success, where success means either setPixels
          or setPixelRef was called. The pixels need not be locked when this
          returns. If the config requires a colortable, it also must be
          installed via setColorTable. If false is returned, the bitmap and
          colortable should be left unchanged.
       virtual bool allocPixelRef(SkBitmap*, SkColorTable*) = 0;
       //这里分配位图的空间
    };
       memory from the heap. This is the default Allocator invoked by
       allocPixels().
    class HeapAllocator : public Allocator {
    public:
       virtual bool allocPixelRef(SkBitmap*, SkColorTable*);
    };
    class RLEPixels {
    public:
       RLEPixels(int width, int height);
       virtual ~RLEPixels();
       uint8_t* packedAtY(int y) const {
          SkASSERT((unsigned)y < (unsigned)fHeight);
          return fYPtrs[y];
       }
       void setPackedAtY(int y, uint8_t* addr) {
          SkASSERT((unsigned)y < (unsigned)fHeight);
          fYPtrs[y] = addr;
       }
    private:
       uint8_t** fYPtrs;
       int       fHeight;
    };
private:
#ifdef SK_SUPPORT_MIPMAP
    struct MipMap;
    mutable MipMap* fMipMap;
#endif
    mutable SkPixelRef* fPixelRef;
    //位图的引用参考计数器
    mutable size_t     fPixelRefOffset;
    //位图的引用参考计数器的偏移
    mutable int        fPixelLockCount;
    //位图数据锁定计数器
    mutable void*      fPixels;
    //位图数据
    mutable SkColorTable* fColorTable;    // only meaningful for kIndex8
    enum Flags {
       kImageIsOpaque_Flag  = 0x01
    };
    uint32_t   fRowBytes;
    //位图一行的字节数
    uint16_t   fWidth, fHeight;
    //位图宽高
    uint8_t    fConfig;
    //位图格式
```

```
    uint8_t     fFlags;
    //图像是否为不透明的标签，有两种状况，一种是图像有一部分为透明的，一部分为不透明的；还有一种状况
    //是整个图像有同样的透明度
    uint8_t     fBytesPerPixel; // based on config
    //位图的一个像素占用的字节数
    /* Unreference any pixelrefs or colortables
    void freePixels();
    void updatePixelsFromRef() const;
    static SkFixed ComputeMipLevel(SkFixed sx, SkFixed dy);
};
```

由此可见，类 SkBitmap 是一个位图类。但它的内部有自己的一些逻辑，尤其是使用了参考计数器，可以避免多重拷贝，同样增加了锁机制。关键的是位图的空间可以由外部给定。

7. 字体类 SkTypeface 和 SkFontHost

在 Android 系统中，字体由 Android 2D 图形引擎 Skia 实现，并在 Zygote 的 Preloading classes 中对系统字体进行载入。在 Skia 中是通过类 SkTypeface 和 SkFontHost 实现字体功能的，对应的相关实现文件有 skTypeface.cpp 和 skFontHost_android.cpp，其中后者是 Skia 针对 Android 平台字体实现的 port，主要存在如下所示的变量中。

```
struct FontInitRec {
const char*          fFileName;
const char* const*  fNames;     // null-terminated list
};
struct FamilyRec {
FamilyRec*  fNext;
SkTypeface*  fFaces[5];
};
uint32_t gFallbackFonts[SK_ARRAY_COUNT(gSystemFonts)+1];
load_system_fonts()@skFontHost_android.cpp
```

在 load 系统中，会给所有的字体分配一个唯一的 ID，并将字体分为 FamilyFonts 和 FallbackFonts 两种。skPaint 通过应用程序设置的字体（Typeface）所对应的 ID 最终实现字符的显示。

在 Android 系统中，DroidSans 是默认字体，只包含西方字符，应用程序在默认情况下都会调用它。而 DroidSansFallback 包含了东亚字符，当需要显示的字符在 DroidSans 字体中不存在（如汉字）时，即没有对应编码的字符时，系统会到 DroidSansFallback 中去找相应编码的字符。如果找到，则使用 DroidSansFallback 字体来显示。如果找不到该编码对应的字符，则无法在屏幕上显示该字符。在 Android 系统中更换默认中文字体的步骤如下所示。

（1）将幼圆字体库 youyuan.ttf 重命名为 DroidSansFallback.ttf，覆盖 Android 源码中"frameworks/base/data/fonts"目录下的文件 DroidSansFallback.ttf。

（2）重新编译 Android 系统。

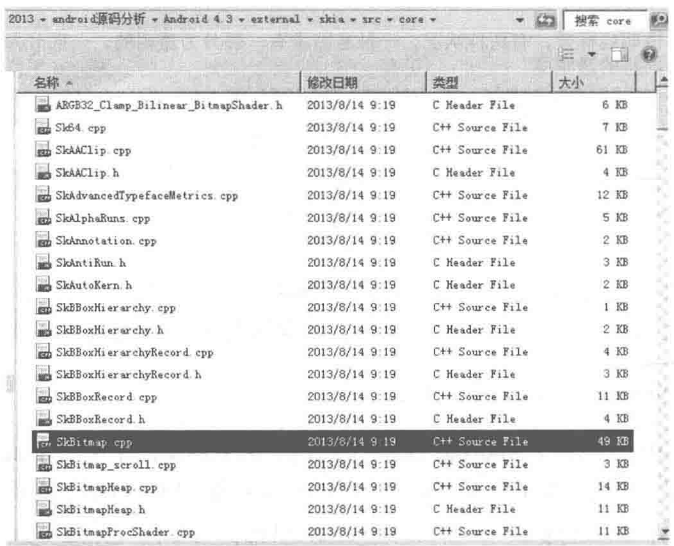
（3）编译 SDK，在生成的 SDK 中，Android 默认的中文字体已更换为幼圆字体。该方法的不足是删除了 Android 系统原来的中文字体。

10.3.3 使用 Skia 绘图

在 Android 多媒体开发应用中，使用 Skia API 进行图形绘制时会用到如下所示的类。

- SkBitmap 类：用来设置像素。
- SkCanvas 类：写入位图。
- SkPaint 类：设置颜色和样式。
- SkRect 类：用来绘制矩形。

上述实现代码主要保存在"external/skia/src/core"目录下，如图 10-7 所示。



▲图 10-7　Skia 绘图类的保存目录

根据图 10-7 中所示的类的名字可以很明确地找到这个类的实现文件，例如类 SkBitmap 的实现文件是 SkBitmap.cpp。

10.3.4　Skia 的其他功能

在 Android 系统中，Skia 不但具有绘图功能外，还有如下所示的常见功能。

（1）图形图像特效。

图形图像特效的实现文件被保存在“src/effects”目录中，主要用于实现一些图形图像的特效，包括遮罩、浮雕、模糊、滤镜、渐变色、离散、透明以及 PATH 的各种特效等。

（2）动画。

动画功能的实现文件被保存在“src/animator”目录中，主要实现了 Skia 的动画效果，但是 Android 不支持。

（3）界面 UI 库。

界面 UI 库的实现文件被保存在“src/view”目录中，这部分实现代码构建了一套完整的界面 UI 库，在里面主要包括 Window、Menu、TextBox、ListView、ProgressBar、Widget、ScrollBa、TagList 和 Image 等几个组件。

（4）3D 效果。

3D 效果的实现文件被保存在“src/gl”目录中，这部分文件用于调用 OpenGL 或 OpenGL ES 来实现 3D 效果。如果定义了 MAC，则使用 OpenGL。如果定义了 Android，则使用嵌入式系统上的 ESGL 三维图形库。

（5）处理图像。

处理图像的实现文件被保存在“src/images”目录中，主要是 SkImageDecoder 和 SkImageEncoder 以及 SkMovie，主要是用来处理 images 图像的，能处理的图像类型包括 BMP、JPEG/PVJPEG、PNG、ICO，而 SkMovie 是用来处理 GIF 动画的。

（6）性能优化。

性能优化的实现文件被保存在“src/opts”目录中。

（7）PDF 处理。

PDF 处理的实现文件被保存在“src/pdf”目录中，主要功能是使用 fpdfemb 库处理 PDF 文档。

（8）接口实现。

接口实现的实现文件被保存在“src/ports”目录中，这部分主要是 Skia 的一些接口在不同系统上的实现，包含了平台相关的代码，比如字体、线程、时间等。其主要包括 Font、Event、File、Thread、Time、XMLParser 等几个部分，这些与 Skia 的接口一样，需要针对不同的操作系统实现。

（9）实现矢量图像。

实现矢量图像的实现文件被保存在“src/svg”目录中，主要用于实现矢量图像，Android 不支持。其主要包括 SkSVGPath、SkSVGPolyline、SkSVGRect、SkSVGText、SkSVGLine、SkSVGImage 和 SkSVGEllipse 等文件。

（10）辅助工具类。

辅助工具类的实现文件被保存在“src/utils”目录中，这部分文件主要是一些辅助工具类，主要包括 SkCamera、SkColorMatrix、SkOSFile、SkProxyCanvas 和 SkInterpolator 等文件。

（11）处理 XML 数据。

处理 XML 数据的实现文件被保存在“src/xml”目录中，主要用于处理 XML 数据的部分。Skia 在这里只是对 XML 解析器做了一层包装，具体的 XML 解析器的实现需要根据不同的操作系统及宿主程序来实现。

通过分析 Skia 源程序，可以知道 Skia 主要使用下面的第三方库。

- Zlib：处理数据的压缩和解压缩；
- Jpeglib：处理 JPEG 图像的编码解码；
- Pnglib：处理 PNG 图像的编码解码；
- Giflib：处理 GIF 图像；
- Fpdfemb：处理 PDF 文档。

另外，Skia 还需要下面列出的 Linux/Unix 的头文件。

- stdint.h；
- unistd.h；
- features.h；
- cdefs.h；
- stubs.h；
- posix_opt.h；
- types.h；
- wordsize.h；
- typesizes.h；
- confname.h；
- getopt.h；
- mman.h。

第11章 绘制二维图像

在 Android 多媒体应用领域中，图像处理是永远的话题之一，这是因为绚丽的生活离不开精美的图片。无论是二维图像还是三维图像，都给手机用户带来了绚丽的视觉冲击。在本章的内容中，将首先讲解在 Android 系统中使用 Graphics 类处理二维图像的知识，为读者进入后面知识的学习打下基础。

11.1 绘图界面布局详解

在 Android 系统中，绘制图形图像时需要先对整个屏幕界面进行布局，为绘制的图形和图像寻找一个落脚点。在这个时候，就涉及了 UI 界面布局的问题。众所周知，手机屏幕的大小十分有限，究竟怎样摆放才能更加优美是 UI 布局的主要任务。在看似简单的手机界面中，不是随随便便、简单布置实现元素排列，而是使用 UI 组件实现界面布局。在本节的内容中，将详细讲解为 Android 图像绘制项目实现 UI 布局的基本知识。

11.1.1 View 视图组件

在 Android 系统中，View 类是一个最基本的 UI 类，几乎所有的 UI 组件都是继承 View 类而实现的。View 类的主要功能如下所示。

（1）为指定的屏幕矩形区域存储布局和内容；

（2）处理尺寸和布局，绘制图形，改变焦点，翻屏，按键、手势；

（3）widget 基类。

View 类的语法格式如下所示。

```
Android.view.View
```

在 Android 中的常用的 View 子类如表 11-1 所示。

表 11-1　View 子类

文本 TextView	输入框 EditText
输入法 InputMethod	活动方法 MovementMethod
复选框 Checkbox	滚动视图 ScrollView
按钮 Button	单选按钮 RadioButton

11.1.2 ViewGroup 容器

ViewGroup 仿佛是一个容器，我们可以对它里面的 View 进行布局处理。使用 ViewGroup 的语法格式如下所示。

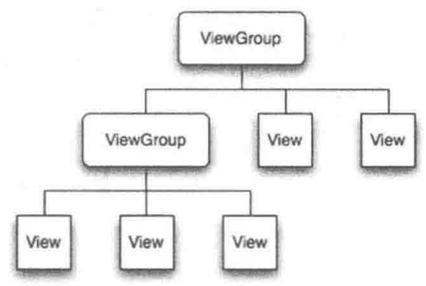
```
android.view.ViewGroup
```

ViewGroup 能够包含并管理下级系列的 Views 和其他 ViewGroup，是一个布局的基类。ViewGroup 好像一个 View 容器，负责对添加进来的 View 进行布局处理。在一个 ViewGroup 中可以看见另一个 ViewGroup 中的内容。各个 ViewGroup 类之间的继承关系如图 11-1 所示。

11.1.3 Layout 规划布局

在 ViewGroup 里面可以装很多控件，布局的作用就是对这些控件进行排列，以达到最实用的效果。在布局里面还可以套用其他的布局，这样可以实现界面多样化以及设计的灵活性。布局组件 Layout 的语法格式如下所示。

```
<LinearLayout xmlns:Android="http://schemas.
Android.com/apk/res/Android"
    Android:orientation="vertical"
    Android:layout_width="fill_parent"
    Android:layout_height="fill_parent"
>
```



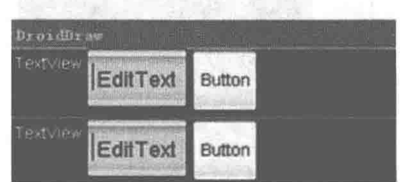
▲图 11-1 各 ViewGroup 类之间的继承关系

在一个布局容器里可以包括 0 个或多个布局容器，其中有如下 5 个最为常用的 Layout 实现类。

（1）AbsoluteLayout：可以让子元素指定准确的 *x/y* 坐标值，并显示在屏幕上。（0,0）表示左上角，当向下或向右移动时，坐标值将变大。AbsoluteLayout 没有页边框，允许元素之间互相重叠（尽管不推荐）。我们通常不推荐使用 AbsoluteLayout，除非你有正当理由，因为它使界面代码太过刚性，以至于在不同的设备上可能不能很好地工作。如图 11-2 所示。

（2）TableLayout：用于把子元素放入行与列中，不显示行、列或是单元格边界线，但是单元格不能横跨行，和 HTML 中一样。如图 11-3 所示。

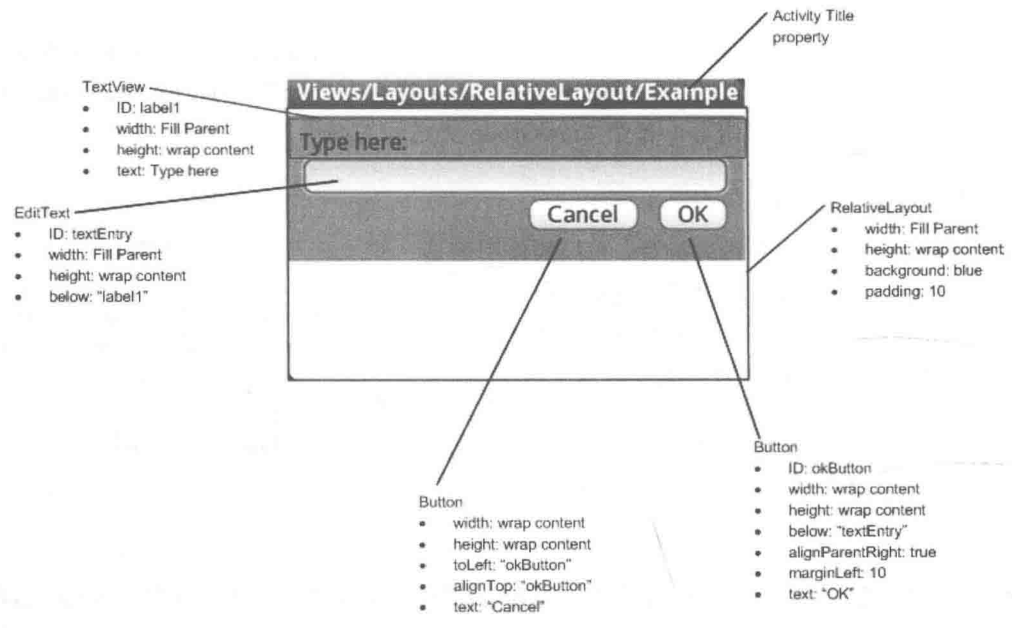
▲图 11-2 AbsoluteLayout 效果



▲图 11-3 TableLayout 效果

（3）FrameLayout：是最简单的一个布局对象，被定制为屏幕上的一个空白备用区域，这样可以在其中填充一个单一对象，例如添加一张要发布的图片。在 FrameLayout 中，所有的子元素将会固定在屏幕的左上角，我们不能为 FrameLayout 中的一个子元素指定一个具体位置。在默认情况下，后一个子元素将会直接在前一个子元素之上进行覆盖填充，把它们部分或全部挡住（除非后一个子元素是透明的）。

（4）RelativeLayout：允许子元素指定它们相对于其他元素或父元素的位置（通过 ID 指定）。我们可以以右对齐、上下或置于屏幕中央的形式来排列两个元素。在 RelativeLayout 中的元素是按顺序排列的，如果第一个元素在屏幕的中央，那么相对于这个元素的其他元素将以屏幕中央的相对位置来排列。如果使用 XML 来指定这个 layout，那么在定义它之前必须定义被关联的元素。其结构如图 11-4 所示。



▲图 11-4　RelativeLayout 结构

（5）LinearLayout：可以在一个方向上（垂直或水平）对齐所有子元素。在里面既可以将所有子元素罗列堆放，也可以一个垂直列表排列，每行将只有一个子元素（无论它们有多宽），如图 11-5 所示。另外也可以一个水平列表排列，只是一列的高度，如图 11-6 所示。

▲图 11-5　LinearLayout 的垂直布局

▲图 11-6　LinearLayout 的水平布局

注意

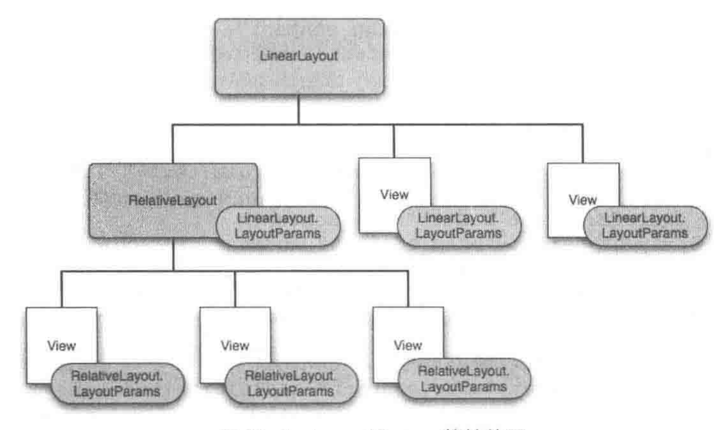
LayoutParams 参数的作用

当把一个 View 加入一个 ViewGroup 中后，例如加入 RelativeLayout 里面，我们如何能知道此时这个 View 在 RelativeLayout 里面是怎样显示的呢？答案其实很简单：当向里面加入 View 时，我们传递一组值，并将这组值封装在 LayoutParams 类中。这样当再显示这个 View 时，其容器会根据封装在 LayoutParams 的值确认此 View 的显示大小和位置。由此可以看出，LayoutParams 的功能如下所示。

（1）每一个 ViewGroup 类使用一个继承于 ViewGroup.LayoutParams 的嵌套类；

（2）包含定义了子节点 View 的尺寸和位置的属性类型。

LayoutParams 的具体结构如图 11-7 所示。



▲图 11-7 LayoutParams 的结构图

在接下来的内容中，将演示使用 TabLayout 布局标签实现一个简易计算器效果的方法。

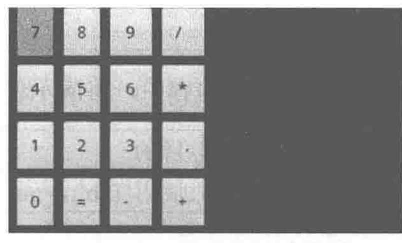
题目	目的	源码路径
实例 11-1	使用 TabLayout 布局实现一个简易计算器界面	\daima\11\biao

布局文件“res/layour/main.xml”的具体实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent" android:layout_height="fill_parent">
<TableRow>
   <Button android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text=" 7 " />
   <Button android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text=" 8 "
    />
   <Button android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text=" 9 "
    />
   <Button android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text=" /   "
    />
</TableRow>
<TableRow>
   <Button android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text=" 4 "
    />
   <Button android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text=" 5 "
    />
   <Button android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text=" 6 "
    />
   <Button android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text=" * "
    />
</TableRow>
<TableRow>
   <Button android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text=" 1 "
    />
   <Button android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content" android:text=" 2 "
        />
      <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text=" 3 "
        />
      <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="   . "
        />
    </TableRow>
    <TableRow>
      <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text=" 0 "
        />
      <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text=" = "
        />
      <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text=" -   "
        />
      <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text=" + "
        />
    </TableRow>
    </TableLayout>
```

通过上述代码可以看出，在布局界面中插入了多个实现按钮效果的 Button 控件，以表示计算器中的各个按键。执行后的效果如图 11-8 所示。



▲图 11-8　计算器执行效果

11.2 Android 绘图基础

有了 View 视图进行 UI 布局处理之后，就可以正式进行绘图工作了。在开始绘图工作之前，需要先了解和绘制 Android 二维图形图像有关的基本知识，这些内容将在本节一一为广大读者呈现。

11.2.1　使用 Canvas 画布

在绘制图形图像的时候，需要先准备一张画布，也就是一张白纸，我们将在这张白纸上绘制图形图像。在 Android 绘制二维图形应用中，类 Canvas 起了这张白纸的作用，也就是画布。在绘制过程中，所有产生的界面类都需要继承于该类。画布类 Canvas 可以看作是一种处理过程，能够使用各种方法来管理 Bitmap、GL 或者 Path 路径。同时 Canvas 可以配合 Matrix 矩阵类给图像做旋转、缩放等操作，并且也提供了裁剪、选取等操作。在类 Canvas 中提供了以下常用的方法。

- Canvas()：功能是创建一个空的画布，可以使用 setBitmap()方法设置绘制的具体画布。
- Canvas(Bitmap bitmap)：功能是以 bitmap 对象创建一个画布，并将内容都绘制在 bitmap 上，bitmap 不能为 null。
- Canvas(GL gl)：在绘制 3D 效果时使用，此方法与 OpenGL 有关。
- drawColor：功能是设置画布的背景色。
- setBitmap：功能是设置具体的画布。
- clipRect：功能是设置显示区域，即设置裁剪区。
- isOpaque：功能是检测是否支持透明。
- rotate：功能是旋转画布。
- canvas.drawRect(RectF,Paint)：功能是绘制矩形，其中第一个参数 Rect 是图形显示区域，第二个参数 Paint 是画笔，设置好图形显示区域和画笔后就可以画图。

● canvas.drawRoundRect(RectF, float, float, Paint)：功能是绘制圆角矩形，第一个参数表示图形显示区域，第二个参数和第三个参数分别表示水平圆角半径和垂直圆角半径。

● canvas.drawLine(startX, startY, stopX, stopY, paint)：前四个参数的类型均为 float，最后一个参数类型为 Paint。表示用画笔 paint 从点（startX,startY）到点（stopX,stopY）画一条直线。

● canvas.drawArc(oval, startAngle, sweepAngle, useCenter, paint)：第一个参数 oval 为 RectF 类型，即圆弧显示区域；startAngle 和 sweepAngle 均为 float 类型，分别表示圆弧起始角度和圆弧度数，3 点钟方向为 0 度；useCenter 设置是否显示圆心，boolean 类型；paint 表示画笔。

● canvas.drawCircle(float,float, float, Paint)：用于绘制圆，前两个参数代表圆心坐标，第三个参数为圆半径，第四个参数是画笔。

Canvas 画布比较重要，特别是在游戏开发应用中。例如对某个精灵执行旋转、缩放等操作时，需要通过旋转画布的方式实现。但是在旋转画布时会旋转画布上的所有对象，而我们只是需要旋转其中的一个，这时就需要用到 save 方法来锁定需要操作的对象，在操作之后通过 restore 方法来解除锁定。

在接下来的内容中，将详细讲解在 Android 中使用画布类 Canvas 的基本知识。

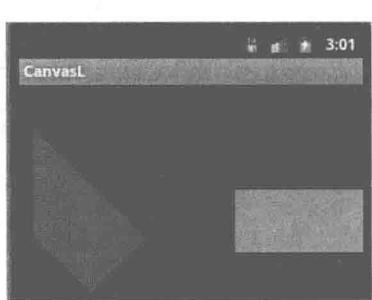
题目	目的	源码路径
实例 11-2	在 Android 中使用 Canvas 类	\daima\11\CanvasL

实例文件 CanvasL.java 的主要代码如下所示：

```
/* 声明 Paint 对象 */
private Paint mPaint = null;
public CanvasL(Context context)
{
    super(context);
    /* 构建对象 */
    mPaint = new Paint();
    /* 开启线程 */
    new Thread(this).start();
}
public void onDraw(Canvas canvas)
{
    super.onDraw(canvas);
    /* 设置画布的颜色 */
    canvas.drawColor(Color.BLACK);
    /* 设置取消锯齿效果 */
    mPaint.setAntiAlias(true);
    /* 设置裁剪区域 */
    canvas.clipRect(10, 10, 280, 260);
    /* 线锁定画布 */
    canvas.save();
    /* 旋转画布 */
    canvas.rotate(45.0f);
    /* 设置颜色及绘制矩形 */
    mPaint.setColor(Color.RED);
    canvas.drawRect(new Rect(15,15,140,70), mPaint);
    /* 解除画布的锁定 */
    canvas.restore();
    /* 设置颜色及绘制另一个矩形 */
    mPaint.setColor(Color.GREEN);
    canvas.drawRect(new Rect(150,75,260,120), mPaint);
}
// 触笔事件
public boolean onTouchEvent(MotionEvent event)
{
    return true;
}
// 按键按下事件
```

```
    public boolean onKeyDown(int keyCode, KeyEvent event)
    {
        return true;
    }
    // 按键弹起事件
    public boolean onKeyUp(int keyCode, KeyEvent event)
    {
        return false;
    }
    public boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event)
    {
        return true;
    }
    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
        {
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
            // 使用 postInvalidate 可以直接在线程中更新界面
            postInvalidate();
        }
    }
}
```

执行后的效果如图 11-9 所示。



▲图 11-9　实例 Canvas1.java 的执行效果

11.2.2　使用 Paint 类

有了画布之后，还需要用一支画笔来绘制图形图像。在 Android 系统中，绘制二维图形图像的画笔是类 Paint。类 Paint 的完整写法是 Android.Graphics.Paint，在里面定义了画笔和画刷的属性。在类 Paint 中的常用方法如下所示。

（1）void reset()：是功能实现重置。

（2）void setARGB(int a, int r, int g, int b)或 void setColor(int color)：功能是设置 Paint 对象的颜色。

（3）void setAntiAlias(boolean aa)：功能是设置是否抗锯齿，此方法需要配合 void setFlags (Paint.ANTI_ALIAS_FLAG)方法一起使用，来帮助消除锯齿使其边缘更平滑。

（4）Shader setShader(Shader shader)：功能是设置阴影效果，Shader 类是一个矩阵对象，如果为 null 则清除阴影。

（5）void setStyle(Paint.Style style)：功能是设置样式，一般为 Fill（填充），或者 STROKE（凹陷效果）。

（6）void setTextSize(float textSize)：功能是设置字体的大小。

（7）void setTextAlign(Paint.Align align)：功能是设置文本的对齐方式。

（8）Typeface setTypeface(Typeface typeface)：功能是设置具体的字体，通过 Typeface 可以加载 Android 内部的字体，对于中文来说一般为宋体，我们可以根据需要来自己添加部分字体，例如雅黑等。

（9）void setUnderlineText(boolean underlineText)：功能是设置是否需要下划线。

在接下来的内容中，将通过一个具体实例来讲解联合使用类 Color 和类 Paint 绘图的过程。

题目	目的	源码路径
实例 11-3	使用 Color 类和 Paint 类绘图	\daima\11\PaintL

本实例的具体实现流程如下所示。

（1）编写布局文件 main.xml，具体代码如下所示。

```
<LinearLayout xmlns:Android="http://schemas.Android.com/apk/res/Android"
    Android:orientation="vertical"
    Android:layout_width="fill_parent"
    Android:layout_height="fill_parent"
    >
<TextView
    Android:layout_width="fill_parent"
    Android:layout_height="wrap_content"
    Android:text="@string/hello"
    />
</LinearLayout>
```

（2）编写文件 Activity.java，通过代码语句“mGameView = new GameView(this)”，调用 Activity 类的 setContentView 方法来设置要显示的具体 View 类。文件 Activity.java 的主要代码如下所示。

```
public class Activity01 extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        mGameView = new GameView(this);

        setContentView(mGameView);
    }
}
```

（3）编写文件 draw.java 来绘制指定的图形，首先声明 Paint 对象 mPaint，定义 draw 分别用于构建对象和开启线程。其主要实现代码如下所示。

```
/* 声明 Paint 对象 */
private Paint mPaint = null;
public draw(Context context)
{
    super(context);
    /* 构建对象 */
    mPaint = new Paint();
    /* 开启线程 */
    new Thread(this).start();
}
```

然后定义方法 onDraw 实现具体的绘制操作，先设置 Paint 格式和颜色，并根据提取的颜色、尺寸、风格、字体和属性实现绘制处理。其主要实现代码如下所示。

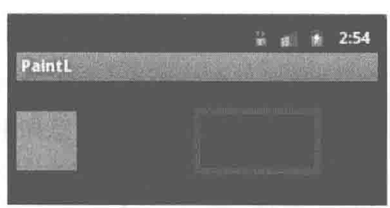
```
public void onDraw(Canvas canvas)
{
    super.onDraw(canvas);
    /* 设置 Paint 为无锯齿 */
    mPaint.setAntiAlias(true);
    /* 设置 Paint 的颜色 */
    mPaint.setColor(Color.WHITE);
    mPaint.setColor(Color.BLUE);
    mPaint.setColor(Color.YELLOW);
    mPaint.setColor(Color.GREEN);
    /* 同样是设置颜色 */
    mPaint.setColor(Color.rgb(255, 0, 0));
    /* 提取颜色 */
    Color.red(0xcccccc);
```

```
    Color.green(0xcccccc);
    /* 设置 paint 的颜色和 Alpha 值(a,r,g,b) */
    mPaint.setARGB(255, 255, 0, 0);
    /* 设置 paint 的 Alpha 值 */
    mPaint.setAlpha(220);
    /* 这里可以设置为另外一个 paint 对象 */
    // mPaint.set(new Paint());
    /* 设置字体的尺寸 */
    mPaint.setTextSize(14);
    // 设置 paint 的风格为“空心”
    // 当然也可以设置为“实心”(Paint.Style.FILL)
    mPaint.setStyle(Paint.Style.STROKE);
    // 设置“空心”的外框的宽度。
    mPaint.setStrokeWidth(5);
    /* 得到 Paint 的一些属性 */
    Log.i(TAG, "paint 的颜色: " + mPaint.getColor());
    Log.i(TAG, "paint 的 Alpha: " + mPaint.getAlpha());
    Log.i(TAG, "paint 的外框的宽度: " + mPaint.getStrokeWidth());
    Log.i(TAG, "paint 的字体尺寸: " + mPaint.getTextSize());
    /* 绘制一个矩形 */
    // 肯定是一个空心的矩形
    canvas.drawRect((320 - 80) / 2, 20, (320 - 80) / 2 + 80, 20 + 40, mPaint);
    /* 设置风格为实心 */
    mPaint.setStyle(Paint.Style.FILL);
    mPaint.setColor(Color.GREEN);
    /* 绘制绿色实心矩形 */
    canvas.drawRect(0, 20, 40, 20 + 40, mPaint);
}
```

最后定义触笔事件 onTouchEvent，定义按键按下事件 onKeyDown，定义按键弹起事件 onKeyUp。其主要实现代码如下所示。

```
    // 触笔事件
    public boolean onTouchEvent(MotionEvent event)
    {
        return true;
    }
    // 按键按下事件
    public boolean onKeyDown(int keyCode, KeyEvent event)
    {
        return true;
    }
    // 按键弹起事件
    public boolean onKeyUp(int keyCode, KeyEvent event)
    {
        return false;
    }
    public boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event)
    {
        return true;
    }
    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
        {
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
            // 使用 postInvalidate 可以直接在线程中更新界面
            postInvalidate();
        }
    }
}
```

执行后的效果如图 11-10 所示。



▲图 11-10 使用 Color 类和 Paint 类绘图的执行效果

11.2.3 位图操作类 Bitmap

准备好画布，并准备好指定颜色的画笔后，就可以在画布上创作自己的作品了。但是有的时候，需要更加细致的操作，例如和 PhotoShop 一样可以在画布中复制图像，可以精确地设置某一个像素的颜色。为了实现上述功能，在 Android 系统中推出了类 Bitmap。类 Bitmap 的完整写法是 Android.Graphics.Bitmap，这是一个位图操作类，能够实现对位图的基本操作。在类 Bitmap 中提供了很多实用的方法，其中最为常用的几种方法如下所示。

（1）boolean compress(Bitmap.CompressFormat format, int quality, OutputStream stream)：功能是压缩一个 Bitmap 对象，并根据相关的编码和画质将其保存到一个 OutputStream 中。目前的压缩格式有 JPG 和 PNG 两种。

（2）void copyPixelsFromBuffer(Buffer src)：功能是从一个 Buffer 缓冲区复制位图像素。

（3）void copyPixelsToBuffer(Buffer dst)：功能是将当前位图像素内容复制到一个 Buffer 缓冲区。

（4）final int getHeight()：功能是获取对象的高度。

（5）final int getWidth()：功能是获取对象的宽度。

（6）final boolean hasAlpha()：功能是设置是否有透明通道。

（7）void setPixel(int x, int y, int color)：功能是设置某像素的颜色。

（8）int getPixel(int x, int y)：功能是获取某像素的颜色。

在 Android 多媒体开发应用中，类 Bitmap 的功能主要体现在如下 3 个方面。

1. 从资源中获取位图

可以使用 BitmapDrawable 或者 BitmapFactory 获取资源中的位图，获取资源的代码如下所示。

```
Resources res=getResources();
```

在 Android 多媒体开发应用中，使用 BitmapDrawable 获取位图的基本流程如下所示。

（1）使用 BitmapDrawable (InputStream is)构造一个 BitmapDrawable。

（2）使用类 BitmapDrawable 中的方法 getBitmap 获得位图。例如通过下面的代码可以读取 InputStream 并得到位图。

```
InputStream is=res.openRawResource(R.drawable.pic180);
BitmapDrawable bmpDraw=new BitmapDrawable(is);
Bitmap bmp=bmpDraw.getBitmap();
```

也可以采用下面的方式实现。

```
BitmapDrawable bmpDraw=(BitmapDrawable)res.getDrawable(R.drawable.pic180);
Bitmap bmp=bmpDraw.getBitmap();
```

接下来需要使用类 BitmapFactory 中的方法 decodeStream(InputStream is)解码位图资源，然后获取位图。具体代码如下所示。

```
Bitmap bmp=BitmapFactory.decodeResource(res, R.drawable.pic180);
```

BitmapFactory 中的所有函数都是静态的，这个辅助类可以通过资源 ID、路径、文件、数据流等方式获取位图。

2. 获取位图的信息

要想获取位图信息，比如位图大小、像素、密度、透明度、颜色格式等，只需获得 Bitmap 即可，在具体实现时需要注意如下两点。

（1）在 Bitmap 中使用 Bitmap.Config 定义 RGB 颜色格式时，仅包括 ALPHA_8、ARGB_4444、ARGB_8888、RGB_565 等格式，缺少了诸如 RGB_555 等格式。

（2）Bitmap 提供了接口 compress 来压缩图片，但是 AndroidSAK 只支持 PNG、JPG 格式的压缩，其他格式的需要 Android 开发人员自己补充。

3. 显示位图

在 Android 多媒体开发应用中，可以使用核心类 Canvas 来显示位图，通过类 Canvas 中的方法 drawBirmap 显示位图，或借助 BitmapDrawable 将 Bitmap 绘制到 Canvas。当然，也可以通过 BitmapDrawable 将位图显示到 View 中。

（1）转换为 BitmapDrawable 对象显示位图，例如下面的代码。

```
// 获取位图
Bitmap bmp=BitmapFactory.decodeResource(res, R.drawable.pic180);
// 转换为 BitmapDrawable 对象
BitmapDrawable bmpDraw=new BitmapDrawable(bmp);
// 显示位图
ImageView iv2 = (ImageView)findViewById(R.id.ImageView02);
iv2.setImageDrawable(bmpDraw);
```

（2）使用 Canvas 类显示位图。

在此可以采用一个继承自 View 的子类 Panel，在子类的 OnDraw 中显示，具体代码如下所示。

```
public class MainActivity extends Activity {
@Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(new Panel(this));
 }
 class Panel extends View{
 public Panel(Context context) {
 super(context);
 }
 public void onDraw(Canvas canvas){
 Bitmap bmp = BitmapFactory.decodeResource(getResources(), R.drawable.pic180);
 canvas.drawColor(Color.BLACK);
 canvas.drawBitmap(bmp, 10, 10, null);
 }
 }
}
```

在接下来的内容中，将通过两个演示实例来讲解使用类 Bitmap 处理图像的基本方法。

题目	目的	源码路径
实例 11-4	使用 Bitmap 类实现模拟水纹效果	\daima\11\BitmapL1

实例文件 BitmapL1.java 的主要实现代码如下所示。

```
public class BitmapL1 extends View implements Runnable
{
    int BACKWIDTH;
    int BACKHEIGHT;
    short[] buf2;
    short[] buf1;
    int[] Bitmap2;
```

```
int[] Bitmap1;
public BitmapL1(Context context)
{
    super(context);
    /* 装载图片 */
    Bitmap image = BitmapFactory.decodeResource(this.getResources(),R.drawable.qq);
    BACKWIDTH = image.getWidth();
    BACKHEIGHT = image.getHeight();

    buf2 = new short[BACKWIDTH * BACKHEIGHT];
    buf1 = new short[BACKWIDTH * BACKHEIGHT];
    Bitmap2 = new int[BACKWIDTH * BACKHEIGHT];
    Bitmap1 = new int[BACKWIDTH * BACKHEIGHT];
    /* 加载图片的像素到数组中 */
    image.getPixels(Bitmap1, 0, BACKWIDTH, 0, 0, BACKWIDTH, BACKHEIGHT);
    new Thread(this).start();
}
void DropStone(int x,// x坐标
            int y,// y坐标
            int stonesize,// 波源半径
            int stoneweight)// 波源能量
{
    for (int posx = x - stonesize; posx < x + stonesize; posx++)
        for (int posy = y - stonesize; posy < y + stonesize; posy++)
            if ((posx - x) * (posx - x) + (posy - y) * (posy - y) < stonesize * stonesize)
                buf1[BACKWIDTH * posy + posx] = (short) -stoneweight;
}
void RippleSpread()
{
    for (int i = BACKWIDTH; i < BACKWIDTH * BACKHEIGHT - BACKWIDTH; i++)
    {
        // 波能扩散
        buf2[i] = (short) (((buf1[i - 1] + buf1[i + 1] + buf1[i - BACKWIDTH] + buf1[i
        + BACKWIDTH]) >> 1) - buf2[i]);
        // 波能衰减
        buf2[i] -= buf2[i] >> 5;
    }
    // 交换波能数据缓冲区
    short[] ptmp = buf1;
    buf1 = buf2;
    buf2 = ptmp;
}
/* 渲染水纹效果 */
void render()
{
    int xoff, yoff;
    int k = BACKWIDTH;
    for (int i = 1; i < BACKHEIGHT - 1; i++)
    {
        for (int j = 0; j < BACKWIDTH; j++)
        {
            // 计算偏移量
            xoff = buf1[k - 1] - buf1[k + 1];
            yoff = buf1[k - BACKWIDTH] - buf1[k + BACKWIDTH];
            // 判断坐标是否在窗口范围内
            if ((i + yoff) < 0)
            {
                k++;
                continue;
            }
            if ((i + yoff) > BACKHEIGHT)
            {
                k++;
                continue;
            }
            if ((j + xoff) < 0)
            {
                k++;
                continue;
```

```
                }
                if ((j + xoff) > BACKWIDTH)
                {
                    k++;
                    continue;
                }
                // 计算出偏移像素和原始像素的内存地址偏移量
                int pos1, pos2;
                pos1 = BACKWIDTH * (i + yoff) + (j + xoff);
                pos2 = BACKWIDTH * i + j;
                Bitmap2[pos2++] = Bitmap1[pos1++];
                k++;
            }
        }
    }

    public void onDraw(Canvas canvas)
    {
        super.onDraw(canvas);
        /* 绘制经过处理的图片效果 */
        canvas.drawBitmap(Bitmap2, 0, BACKWIDTH, 0, 0, BACKWIDTH, BACKHEIGHT, false, null);
    }
    // 触笔事件
    public boolean onTouchEvent(MotionEvent event)
    {
        return true;
    }
    // 按键按下事件
    public boolean onKeyDown(int keyCode, KeyEvent event)
    {
        return true;
    }
    // 按键弹起事件
    public boolean onKeyUp(int keyCode, KeyEvent event)
    {
        DropStone(BACKWIDTH/2, BACKHEIGHT/2, 10, 30);
        return false;
    }
    public boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event)
    {
        return true;
    }
    /*线程处理*/
    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
        {
            try
            {
                Thread.sleep(50);
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
            RippleSpread();
            render();
            //使用 postInvalidate 可以直接在线程中更新界面
            postInvalidate();
        }
    }
}
```

▲图 11-11　使用 Bitmap 类模拟水纹效果的执行效果

执行后将通过对图像像素的操作数来模拟水纹效果，如图 11-11 所示。

题目	目的	源码路径
实例 11-5	旋转屏幕中的一幅图片	\daima\11\BitmapL2

本实例的实现文件是 BitmapL2.java，分别实现了左旋转按钮事件 mButton1.setOnClickListener 和右旋转按钮事件 mButton2.setOnClickListener。文件 BitmapL2.java 的主要实现代码如下所示。

```
public class BitmapL2 extends Activity
{
  private Button mButton1;
  private Button mButton2;
  private TextView mTextView1;
  private ImageView mImageView1;
  private int ScaleTimes;
  private int ScaleAngle;
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mButton1 =(Button) findViewById(R.id.myButton1);
    mButton2 =(Button) findViewById(R.id.myButton2);
    mTextView1 = (TextView) findViewById(R.id.myTextView1);
    mImageView1 = (ImageView) findViewById(R.id.myImageView1);
    ScaleTimes = 1;
    ScaleAngle = 1;

    final Bitmap mySourceBmp =
    BitmapFactory.decodeResource(getResources(), R.drawable.hippo);

    final int widthOrig = mySourceBmp.getWidth();
    final int heightOrig = mySourceBmp.getHeight();

    /* 程序刚运行，加载默认的 Drawable */
    mImageView1.setImageBitmap(mySourceBmp);

    /* 向左旋转按钮 */
    mButton1.setOnClickListener(new Button.OnClickListener()
    {
      @Override
      public void onClick(View v)
      {
        // TODO Auto-generated method stub
        ScaleAngle--;
        if(ScaleAngle<-5)
        {
          ScaleAngle = -5;
        }

        /* ScaleTimes=1, 维持 1:1 的宽高比例*/
        int newWidth = widthOrig * ScaleTimes;
        int newHeight = heightOrig * ScaleTimes;

        float scaleWidth = ((float) newWidth) / widthOrig;
        float scaleHeight = ((float) newHeight) / heightOrig;

        Matrix matrix = new Matrix();
        /* 使用 Matrix.postScale 设置维度 */
        matrix.postScale(scaleWidth, scaleHeight);

        /* 使用 Matrix.postRotate 方法旋转 Bitmap*/
        //matrix.postRotate(5*ScaleAngle);
        matrix.setRotate(5*ScaleAngle);

        /* 创建新的 Bitmap 对象 */
        Bitmap resizedBitmap =
        Bitmap.createBitmap
        (mySourceBmp, 0, 0, widthOrig, heightOrig, matrix, true);
        BitmapDrawable myNewBitmapDrawable =
        new BitmapDrawable(resizedBitmap);
        mImageView1.setImageDrawable(myNewBitmapDrawable);
        mTextView1.setText(Integer.toString(5*ScaleAngle));
```

```
        }
    });
    /* 向右旋转按钮 */
    mButton2.setOnClickListener(new Button.OnClickListener()
    {
      @Override
      public void onClick(View v)
      {
        ScaleAngle++;
        if(ScaleAngle>5)
        {
          ScaleAngle = 5;
        }
        /* ScaleTimes=1，维持 1:1 的宽高比例*/
        int newWidth = widthOrig * ScaleTimes;
        int newHeight = heightOrig * ScaleTimes;

        /* 计算旋转的 Matrix 比例 */
        float scaleWidth = ((float) newWidth) / widthOrig;
        float scaleHeight = ((float) newHeight) / heightOrig;

        Matrix matrix = new Matrix();
        /* 使用 Matrix.postScale 设置维度 */
        matrix.postScale(scaleWidth, scaleHeight);
        /* 使用 Matrix.postRotate 方法旋转 Bitmap*/
        //matrix.postRotate(5*ScaleAngle);
        matrix.setRotate(5*ScaleAngle);

        /* 创建新的 Bitmap 对象 */
        Bitmap resizedBitmap =
        Bitmap.createBitmap
        (mySourceBmp, 0, 0, widthOrig, heightOrig, matrix, true);
        BitmapDrawable myNewBitmapDrawable =
        new BitmapDrawable(resizedBitmap);
        mImageView1.setImageDrawable(myNewBitmapDrawable);
        mTextView1.setText(Integer.toString(5*ScaleAngle));
      }
    });
  }
}
```

执行后将显示一幅图片和两个按钮，单击“左旋转”和“右旋转”按钮后会实现对图片的旋转处理。如图 11-12 所示。

▲图 11-12　旋转屏幕中一幅图片的执行效果

11.3 使用其他的绘图类

经过本章前面内容的学习，已经了解了画布类、画图类和位图操作类的基本知识，根据这 3 种技术可以在手机屏幕中绘制图形图像。另外，在 Android 多媒体开发应用中，还可以使用其他的绘图类来绘制二维图形图像。有关这些其他绘图类的具体用法，在本节的内容中将进行详细讲解。

11.3.1 使用设置文本颜色类 Color

在 Android 系统中，类 Color 的完整写法是 Android.Graphics.Color，通过此类可以很方便地绘制 2D 图像，并为这些图像填充不同的颜色。在 Android 平台上有很多种表示颜色的方法，在里面包含了如下 12 种最常用的颜色。

- Color.BLACK；
- Color.BLUE；
- Color.CYAN；
- Color.DKGRAY；
- Color.GRAY；
- Color.GREEN；
- Color.LTGRAY；
- Color.MAGENTA；
- Color.RED；
- Color.TRANSPARENT；
- Color.WHITE；
- Color.YELLOW。

在 r 类 Colo 中包含了如下 3 个常用的静态方法。

（1）static int argb(int alpha, int red, int green, int blue)：功能是构造一个包含透明对象的颜色。

（2）static int rgb(int red, int green, int blue)：功能是构造一个标准的颜色对象。

（3）static int parseColor(String colorString)：功能是解析一种颜色字符串的值，比如传入 Color.BLACK。

类 Color 中的静态方法返回的都是一个整形结果，例如返回 0xff00ff00 表示绿色，返回 0xffff0000 表示红色。我们可以将这个 DWORD 型看作 AARRGGBB，AA 代表 Aphla 透明色，后面的 RRGGBB 是具体颜色值，用 0～255 之间的数字表示。

在接下来的内容中，将通过一个具体的演示实例来讲解使用类 Color 更改文字颜色的基本方法。

题目	目的	源码路径
实例 11-6	使用类 Color 更改文字的颜色	\daima\11\yanse

1. 设计理念

在本实例中，预先在 Layout 中插入两个 TextView 控件，并通过两种程序的描述方法来实时更改原来 Layout 里 TextView 的背景色以及文字颜色，最后使用类 Android.Graphics.Color 来更改文字的前景色。

2. 具体实现

（1）编写主文件 yanse.java，功能是调用各个公用文件来实现具体的功能。主要实现代码如下所示。

```
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  mTextView01 = (TextView) findViewById(R.id.myTextView01);
  mTextView01.setText("使用的是 Drawable 背景色文本。");
  Resources resources = getBaseContext().getResources();
  Drawable HippoDrawable = resources.getDrawable(R.drawable.white);
  mTextView01.setBackgroundDrawable(HippoDrawable);
```

```
    mTextView02 = (TextView) findViewById(R.id.myTextView02);
    mTextView02.setTextColor(Color.MAGENTA);
  }
}
```

在上述代码中，分别新建了两个类成员变量 mTextView01 和 mTextView02。这两个变量在 onCreate 之初，使用 findViewById 方法将其初始化为 layout（main.xml）里的 TextView 对象。在当中使用了 Resource 类以及 Drawable 类，分别创建了 resources 对象以及 HippoDrawable 对象，并调用了 setBackgroundDrawable 来更改 mTextView01 的文字底纹。更改 TextView 里的文字，则使用了 setText 方法。

在 mTextView02 中，使用了类 Android.Graphics.Color 中的颜色常数，并使用 setTextColor 来更改文字的前景色。

（2）编写布局文件 main.xml，在里面使用了 2 个 TextView 对象，主要实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TextView
  android:id="@+id/myTextView01"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/str_textview01"
  />
  <TextView
  android:id="@+id/myTextView02"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/str_textview02"
  />
</LinearLayout>
```

经过上述操作设置，此实例的主要文件编程完毕。调试运行后的效果如图 11-13 所示。

▲图 11-13 使用类 Color 更改文字颜色的运行效果

11.3.2 使用矩形类 Rect 和 RectF

1. 类 Rect

在 Android 系统中，类 Rect 的完整形式是 Android.Graphics.Rect，表示矩形区域。类 Rect 除了能够表示一个矩形区域位置描述外，还可以帮助计算图形之间是否有碰撞（包含）关系，这一点对于 Android 游戏开发比较有用。在类 Rect 中的方法成员中，主要通过如下 3 种重载方法来判断包含关系。

```
boolean contains(int left, int top, int right, int bottom)
boolean contains(int x, int y)
boolean contains(Rect r)
```

在上述构造方法中包含了 4 个参数：left、top、right、bottom，分别代表左、上、右、下四个方向，具体说明如下所示。

- left：矩形区域中左边的 x 坐标。

- top：矩形区域中顶部的 y 坐标。
- right：矩形区域中右边的 x 坐标。
- bottom：矩形区域中底部的 y 坐标。

例如下面代码的含义是，左上角的坐标是（150,75），右下角的坐标是（260,120）。

```
Rect(150, 75, 260, 120)
```

2. 类 RectF

在 Android 系统中，另外一个矩形类是 RectF，此类和类 Rect 的用法几乎完全相同。两者的区别是精度不一样，Rect 是使用 int 类型作为数值，RectF 是使用 float 类型作为数值。在类 RectF 中包含了一个矩形的四个单精度浮点坐标，通过上、下、左、右 4 个边的坐标来表示一个矩形。这些坐标值属性可以被直接访问，使用 width 和 height 方法可以获取矩形的宽和高。

类 Rect 和类 RectF 提供的方法也不是完全一致，类 RectF 提供了如下所示的构造方法。

- RectF()：功能是构造一个无参的矩形。
- RectF(float left,float top,float right,float bottom)：功能是构造一个指定了 4 个参数的矩形。
- RectF(Rect F r)：功能是根据指定的 RectF 对象来构造一个 RectF 对象（对象的左边坐标不变）。
- RectF(Rect r)：功能是根据给定的 Rect 对象来构造一个 RectF 对象。

另外在类 RectF 中还提供了很多功能强大的方法，具体说明如下所示。

- Public Boolean contain(RectF r)：功能是判断一个矩形是否在此矩形内，如果在这个矩形内或者和这个矩形等价则返回 true，类似的方法还有 public Boolean contain(float left,float top,float right,float bottom)和 public Boolean contain(float x,float y)。
- Public void union(float x,float y)：功能是更新这个矩形，使它包含矩形自己和（x，y）这个点。

在接下来的内容中，将通过一个具体的演示实例讲解在 Android 中使用 Canvas 类的方法。

题目	目的	源码路径
实例 11-7	在 Android 中使用 Canvas 类	\daima\11\RectL

实例文件 RectL.java 的主要实现代码如下所示。

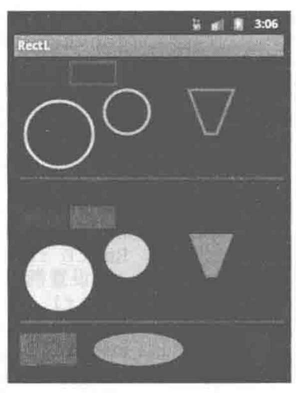
```
/* 声明 Paint 对象 */
private Paint mPaint = null;
    private RectL_1 mGameView2 = null;
public RectL(Context context)
{
    super(context);
    /* 构建对象 */
    mPaint = new Paint();

    mGameView2 = new RectL_1(context);

    /* 开启线程  */
    new Thread(this).start();
}

public void onDraw(Canvas canvas)
{
    super.onDraw(canvas);

    /* 设置画布为黑色背景 */
    canvas.drawColor(Color.BLACK);
    /* 取消锯齿 */
```

```
mPaint.setAntiAlias(true);

mPaint.setStyle(Paint.Style.STROKE);

{
    /* 定义矩形对象 */
    Rect rect1 = new Rect();
    /* 设置矩形大小 */
    rect1.left = 5;
    rect1.top = 5;
    rect1.bottom = 25;
    rect1.right = 45;

    mPaint.setColor(Color.BLUE);
    /* 绘制矩形 */
    canvas.drawRect(rect1, mPaint);

    mPaint.setColor(Color.RED);
    /* 绘制矩形 */
    canvas.drawRect(50, 5, 90, 25, mPaint);

    mPaint.setColor(Color.YELLOW);
    /* 绘制圆形(圆心 x,圆心 y,半径 r,p) */
    canvas.drawCircle(40, 70, 30, mPaint);

    /* 定义椭圆对象 */
    RectF rectf1 = new RectF();
    /* 设置椭圆大小 */
    rectf1.left = 80;
    rectf1.top = 30;
    rectf1.right = 120;
    rectf1.bottom = 70;

    mPaint.setColor(Color.LTGRAY);
    /* 绘制椭圆 */
    canvas.drawOval(rectf1, mPaint);

    /* 绘制多边形 */
    Path path1 = new Path();

    /*设置多边形的点*/
    path1.moveTo(150+5, 80-50);
    path1.lineTo(150+45, 80-50);
    path1.lineTo(150+30, 120-50);
    path1.lineTo(150+20, 120-50);
    /* 使这些点构成封闭的多边形 */
    path1.close();

    mPaint.setColor(Color.GRAY);
    /* 绘制这个多边形 */
    canvas.drawPath(path1, mPaint);

    mPaint.setColor(Color.RED);
    mPaint.setStrokeWidth(3);
    /* 绘制直线 */
    canvas.drawLine(5, 110, 315, 110, mPaint);
}
//
//下面绘制实心几何体
//
mPaint.setStyle(Paint.Style.FILL);
{
    /* 定义矩形对象 */
    Rect rect1 = new Rect();
    /* 设置矩形大小 */
    rect1.left = 5;
    rect1.top = 130+5;
    rect1.bottom = 130+25;
    rect1.right = 45;
    mPaint.setColor(Color.BLUE);
```

```
            /* 绘制矩形 */
            canvas.drawRect(rect1, mPaint);

            mPaint.setColor(Color.RED);
            /* 绘制矩形 */
            canvas.drawRect(50, 130+5, 90, 130+25, mPaint);
            mPaint.setColor(Color.YELLOW);
            /* 绘制圆形(圆心 x,圆心 y,半径 r,p) */
            canvas.drawCircle(40, 130+70, 30, mPaint);
            /* 定义椭圆对象 */
            RectF rectf1 = new RectF();
            /* 设置椭圆大小 */
            rectf1.left = 80;
            rectf1.top = 130+30;
            rectf1.right = 120;
            rectf1.bottom = 130+70;
            mPaint.setColor(Color.LTGRAY);
            /* 绘制椭圆 */
            canvas.drawOval(rectf1, mPaint);
            /* 绘制多边形 */
            Path path1 = new Path();
            /*设置多边形的点*/
            path1.moveTo(150+5, 130+80-50);
            path1.lineTo(150+45, 130+80-50);
            path1.lineTo(150+30, 130+120-50);
            path1.lineTo(150+20, 130+120-50);
            /* 使这些点构成封闭的多边形 */
            path1.close();
            mPaint.setColor(Color.GRAY);
            /* 绘制这个多边形 */
            canvas.drawPath(path1, mPaint);
            mPaint.setColor(Color.RED);
            mPaint.setStrokeWidth(3);
            /* 绘制直线 */
            canvas.drawLine(5, 130+110, 315, 130+110, mPaint);
        }
        /* 通过 ShapeDrawable 来绘制几何图形 */
        mGameView2.DrawShape(canvas);
    }
    // 触笔事件
    public boolean onTouchEvent(MotionEvent event)
    {
        return true;
    }
    // 按键按下事件
    public boolean onKeyDown(int keyCode, KeyEvent event)
    {
        return true;
    }
    // 按键弹起事件
    public boolean onKeyUp(int keyCode, KeyEvent event)
    {
        return false;
    }
    public boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event)
    {
        return true;
    }
    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
        {
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
```

```
            //使用 postInvalidate 可以直接在线程中更新界面
            postInvalidate();
        }
    }
}
```

执行后的效果如图 11-14 所示。



▲图 11-14　在 Android 中使用 Canvas 类的执行效果

11.3.3　非矢量图形拉伸类 NinePatch

在 Android 系统中，类 NinePatch 的完整形式是 Android.Graphics. NinePatch。类 NinePatch 是 Android 中特有的一种非矢量图形自然拉伸处理方法，可以帮助常规的图形在拉伸时不会缩放。在 Android SDK 中提供了一个名为“Draw 11-Patch”的工具，有关该工具的使用方法可以参考相关资料，因为这不是本书重点，所以不做详细讲解。由于该类提供了高质量支持透明的缩放方式，所以图形格式为 PNG，文件命名方式为.11.png 的后缀，比如 Android123.11.png。

采用 NinePatch 图片做背景，可使背景随着内容的拉伸（缩小）而拉伸（缩小）。那么如何将普通的 PNG 图片编辑为 NinePatch 图片呢？在 Android SDK 的“tools”目录下提供了编辑器 draw9patch.bat，双击即可打开，使用起来很简单，在里面主要有以下选项。

- Zoom：用来缩放左边编辑区域的大小。
- Patch scale：用来缩放右边预览区域的大小。
- Show lock：当鼠标在图片区域的时候显示不可编辑区域。
- Show patches：在编辑区域显示图片拉伸的区域，使用粉红色来标示。
- Show content：在预览区域显示图片的内容区域，使用浅紫色来标示。
- Show bad patches：在拉伸区域周围用红色边框显示可能会对拉伸后的图片产生变形的区域，如果完全消除该内容则图片拉伸后是没有变形的，也就是说不管如何缩放图片显示都是良好的。

11.3.4　使用变换处理类 Matrix

在 Android 系统中，类 Matrix 的完整形式是 Android.Graphics.Matrix，功能是实现图形图像的变换操作，例如常见的缩放和旋转处理。在类 Matrix 中提供了以下几种常用的方法。

（1）void reset()：功能是重置一个 matrix 对象。

（2）void set(Matrix src)：功能是复制一个源矩阵，和本类的构造方法 Matrix(Matrix src)一样。

（3）boolean isIdentity()：功能是返回这个矩阵是否定义（已经有意义）。

（4）void setRotate(float degrees)：功能是指定一个角度以点（0,0）为坐标进行旋转。

（5）void setRotate(float degrees, float px, float py)：功能是指定一个角度以点（px,py）为坐标进行旋转。

（6）void setScale(float sx, float sy)：功能是实现缩放处理。

（7）void setScale(float sx, float sy, float px, float py)：功能是以点（px,py）为坐标进行缩放。

（8）void setTranslate(float dx, float dy)：功能是实现平移处理。

（9）void setSkew (float kx, float ky, float px, float py)：功能是以点（px，py）为坐标进行倾斜。

（10）void setSkew (float kx, float ky)：功能是实现倾斜处理。

在接下来的内容中，将通过一个具体的演示实例来讲解用类 Matrix 实现图片缩放功能的方法。

题目	目的	源码路径
实例 11-8	使用 Matrix 类实现图片缩放功能	\daima\11\MatrixL

本实例的核心程序文件是 MatrixL.java，功能是实现图片缩放处理，分别定义缩小按钮响应 mButton01.setOnClickListener，放大按钮响应 mButton02.setOnClickListener。文件 MatrixL.java 的主要实现代码如下所示。

```
public class MatrixL extends Activity
{
  /* 相关变量声明 */
  private ImageView mImageView;
  private Button mButton01;
  private Button mButton02;
  private AbsoluteLayout layout1;
  private Bitmap bmp;
  private int id=0;
  private int displayWidth;
  private int displayHeight;
  private float scaleWidth=1;
  private float scaleHeight=1;
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    /* 载入 main.xml Layout */
    setContentView(R.layout.main);

    /* 取得屏幕分辨率大小 */
    DisplayMetrics dm=new DisplayMetrics();
    getWindowManager().getDefaultDisplay().getMetrics(dm);
    displayWidth=dm.widthPixels;
    /* 屏幕高度须扣除下方 Button 高度 */
    displayHeight=dm.heightPixels-80;
    /* 初始化相关变量 */
    bmp=BitmapFactory.decodeResource(getResources(),
                                R.drawable.suofang);
    mImageView = (ImageView)findViewById(R.id.myImageView);
    layout1 = (AbsoluteLayout)findViewById(R.id.layout1);
    mButton01 = (Button)findViewById(R.id.myButton1);
    mButton02 = (Button)findViewById(R.id.myButton2);
    /* 缩小按钮 onClickListener */
    mButton01.setOnClickListener(new Button.OnClickListener()
    {
      @Override
      public void onClick(View v)
      {
        small();
      }
    });
    /* 放大按钮 onClickListener */
    mButton02.setOnClickListener(new Button.OnClickListener()
    {
      @Override
      public void onClick(View v)
      {
        big();
      }
    });
  }
  /* 图片缩小的 method */
  private void small()
  {
    int bmpWidth=bmp.getWidth();
    int bmpHeight=bmp.getHeight();
    /* 设置图片缩小的比例 */
    double scale=0.8;
    /* 计算出这次要缩小的比例 */
    scaleWidth=(float) (scaleWidth*scale);
    scaleHeight=(float) (scaleHeight*scale);

    /* 产生 reSize 后的 Bitmap 对象 */
    Matrix matrix = new Matrix();
```

```
    matrix.postScale(scaleWidth, scaleHeight);
    Bitmap resizeBmp = Bitmap.createBitmap(bmp,0,0,bmpWidth,
                                   bmpHeight,matrix,true);
    if(id==0)
    {
      /* 如果是第一次按，就删除原来默认的 ImageView */
      layout1.removeView(mImageView);
    }
    else
    {
      /* 如果不是第一次按，就删除上次放大缩小所产生的 ImageView */
      layout1.removeView((ImageView)findViewById(id));
    }
    /* 产生新的 ImageView，放入 reSize 的 Bitmap 对象，再放入 Layout 中 */
    id++;
    ImageView imageView = new ImageView(suofang.this);
    imageView.setId(id);
    imageView.setImageBitmap(resizeBmp);
    layout1.addView(imageView);
    setContentView(layout1);

    /* 因为图片放到最大时放大按钮会 disable，所以在缩小时把它重设为 enable */
    mButton02.setEnabled(true);
  }
  /* 图片放大的 method */
  private void big()
  {
    int bmpWidth=bmp.getWidth();
    int bmpHeight=bmp.getHeight();
    /* 设置图片放大的比例 */
    double scale=1.25;
    /* 计算这次要放大的比例 */
    scaleWidth=(float)(scaleWidth*scale);
    scaleHeight=(float)(scaleHeight*scale);

    /* 产生 reSize 后的 Bitmap 对象 */
    Matrix matrix = new Matrix();
    matrix.postScale(scaleWidth, scaleHeight);
    Bitmap resizeBmp = Bitmap.createBitmap(bmp,0,0,bmpWidth,
                                   bmpHeight,matrix,true);

    if(id==0)
    {
      /* 如果是第一次按，就删除原来设置的 ImageView */
      layout1.removeView(mImageView);
    }
    else
    {
      /* 如果不是第一次按，就删除上次放大缩小所产生的 ImageView */
      layout1.removeView((ImageView)findViewById(id));
    }
    /* 产生新的 ImageView，放入 reSize 的 Bitmap 对象，再放入 Layout 中 */
    id++;
    ImageView imageView = new ImageView(suofang.this);
    imageView.setId(id);
    imageView.setImageBitmap(resizeBmp);
    layout1.addView(imageView);
    setContentView(layout1);

    /* 如果再放大会超过屏幕大小，就把 Button disable */
    if(scaleWidth*scale*bmpWidth>displayWidth||
        scaleHeight*scale*bmpHeight>displayHeight)
    {
      mButton02.setEnabled(false);
    }
  }
```

执行后将显示一幅图片和两个按钮，分别单击“缩小”和“放大”按钮后会实现对图片的缩小、放大处理。如图 11-15 所示。

▲图 11-15　使用 Matrix 类缩放图片的执行效果

11.3.5 使用 BitmapFactory 类

在 Android 系统中，类 BitmapFactory 的完整形式是 Android.Graphics.BitmapFactory。类 BitmapFactory 是 Bitmap 对象的 I/O 类，在里面提供了丰富的构造 Bitmap 对象的方法，比如从一个字节数组、文件系统、资源 ID 以及输入流中来创建一个 Bitmap 对象。在类 BitmapFactory 中提供了如下所示的成员。

（1）从字节数组中的创建方法。

- static Bitmap decodeByteArray(byte[] data, int offset, int length)；
- static Bitmap decodeByteArray(byte[] data, int offset, int length, BitmapFactory.Options opts)。

（2）从文件创建方法，在使用时要写全路径。

- static Bitmap decodeFile(String pathName, BitmapFactory.Options opts)；
- static Bitmap decodeFile(String pathName)。

（3）从输入流句柄中的创建方法。

- static Bitmap decodeFileDescriptor(FileDescriptor fd, Rect outPadding, BitmapFactory.Options opts)；
- static Bitmap decodeFileDescriptor(FileDescriptor fd)。

（4）从 Android 的 APK 文件资源中的创建方法。

- static Bitmap decodeResource(Resources res, int id)。
- static Bitmap decodeResource(Resources res, int id, BitmapFactory.Options opts)。
- static Bitmap decodeResourceStream(Resources res, TypedValue value, InputStream is, Rect pad, BitmapFactory.Options opts)。

（5）从一个输入流中的创建方法。

- static Bitmap decodeStream(InputStream is)；
- static Bitmap decodeStream(InputStream is, Rect outPadding, BitmapFactory.Options opts)。

在智能手机系统中，很有必要获取屏幕中某幅图片的宽和高。在接下来的内容中，将通过一个具体的演示实例来讲解使用类 BitmapFactory 获取指定图片的宽和高的方法。

题目	目的	源码路径
实例 11-9	使用 BitmapFactory 类获取指定图片的宽和高	\daima\11\BitmapFactoryL

在本实例中，通过 ListView 控件实现了一个操作选项效果，当用户单击一个选项后能够分别获取图片的宽和高。在具体实现时，通过 Bitmap 对象的 BitmapFactory.decodeResource 方法来获取预先设定的图片“m123.png”，然后再通过 Bitmap 对象的 getHeight 和 getWidth 来获取图片的宽和高。本实例的主程序文件是 BitmapFactoryL.java，具体实现流程如下所示。

（1）通过 findViewById 构造器来创建 TextView 和 ImageView 对象，然后将 Drawable 中的图片 m123.png 放入自定义的 ImageView 中。其主要实现代码如下所示。

```
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  /*通过 findViewById 构造器创建 TextView 与 ImageView 对象*/
  mTextView01 = (TextView)findViewById(R.id.myTextView1);
  mImageView01= (ImageView)findViewById(R.id.myImageView1);
  /*将 Drawable 中的图片 baby.png 放入自定义的 ImageView 中*/
  mImageView01.setImageDrawable(getResources().
          getDrawable(R.drawable.m123));
```

（2）设置 OnCreateContextMenuListener 监听给 TextView，这样图片上可以使用 ContextMenu，然后覆盖 OnCreateContextMenu 来创建 ContextMenu 的选项。其主要实现代码如下所示。

```
  /*设置 OnCreateContextMenuListener 给 TextView，让图片上可以使用 ContextMenu*/
  mImageView01.setOnCreateContextMenuListener
  (new ListView.OnCreateContextMenuListener()
  {
    /*覆盖 OnCreateContextMenu 来创建 ContextMenu 的选项*/
    public void onCreateContextMenu
    (ContextMenu menu, View v, ContextMenuInfo menuInfo)
    {
      menu.add(Menu.NONE, CONTEXT_ITEM1, 0, R.string.str_context1);
      menu.add(Menu.NONE, CONTEXT_ITEM2, 0, R.string.str_context2);
      menu.add(Menu.NONE, CONTEXT_ITEM3, 0, R.string.str_context3);
    }
  });
}
```

（3）覆盖 OnContextItemSelected 来定义用户单击 MENU 键后的动作，然后通过自定义 Bitmap 对象 BitmapFactory.decodeResource 来获取预设的图片资源。其主要实现代码如下所示。

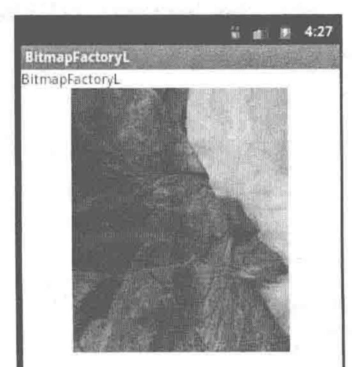
```
/*覆盖 OnContextItemSelected 来定义用户单击 MENU 键后的动作*/
public boolean onContextItemSelected(MenuItem item)
{
  /*自定义 Bitmap 对象并通过 BitmapFactory.decodeResource 取得
   *预先 Import 至 Drawable 的 baby.png 图档*/
  Bitmap myBmp = BitmapFactory.decodeResource
    (getResources(), R.drawable.baby);
   /*通过 Bitmap 对象的 getHight 与 getWidth 来取得图片宽和高*/
   int intHeight = myBmp.getHeight();
   int intWidth = myBmp.getWidth();
```

（4）根据用户选择的选项，分别通过方法 getHeight()和 getWidth()获取对应图片宽度和高度。其主要实现代码如下所示。

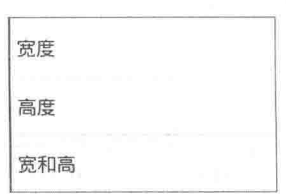
```
    try
    {
      /*菜单选项与动作*/
      switch(item.getItemId())
      {
        /*将图片宽度显示在 TextView 中*/
        case CONTEXT_ITEM1:
          String strOpt =
          getResources().getString(R.string.str_width)
          +"="+Integer.toString(intWidth);
          mTextView01.setText(strOpt);
          break;
        /*将图片高度显示在 TextView 中*/
        case CONTEXT_ITEM2:
          String strOpt2 =
          getResources().getString(R.string.str_height)
          +"="+Integer.toString(intHeight);
          mTextView01.setText(strOpt2);
          break;
        /*将图片宽高显示在 TextView 中*/
        case CONTEXT_ITEM3:
          String strOpt3 =
          getResources().getString(R.string.str_width)
          +"="+Integer.toString(intWidth)+"\n"
          +getResources().getString(R.string.str_height)
          +"="+Integer.toString(intHeight);
          mTextView01.setText(strOpt3);
          break;
      }
    }
    catch(Exception e)
```

```
      {
        e.printStackTrace();
      }
      return super.onContextItemSelected(item);
    }
  }
```

执行后的效果如图 11-16 所示，当长时间选中图片后会弹出用户选项，如图 11-17 所示。当选择一个选项后，会弹出对应的获取数值，如图 11-18 所示。



▲图 11-16　使用 BitmapFactory 类获取图片宽和高的初始效果



▲图 11-17　使用 BitmapFactory 类获取图片宽和高的弹出选项

▲图 11-18　使用 BitmapFactory 类获取图片宽和高的对应数值

11.3.6　使用 Region 类

在 Android 系统中，类 Region 的完整写法是 Android.Graphics.Region，此类在 Android 平台中表示的区域和 Rect 表示的不同。类 Region 表示的是一个不规则的样子，可以是椭圆或多边形等，而类 Rect 表示的仅仅是矩形。同样，类 Region 的 boolean contains(int x, int y)成员可以判断一个点是否在该区域内。

Region 的中文意思即区域的意思，它表示的是 canvas 图层上的某一块封闭的区域。为了更好地学习类 Region，在下面的代码中列出此类的所有 API。

```
/**构造方法*/
 public Region()  //创建一个空的区域
 public Region(Region region) //拷贝一个 region 的范围
 public Region(Rect r)  //创建一个矩形的区域
 public Region(int left, int top, int right, int bottom) //创建一个矩形的区域

/**一系列 set 方法，这些 set 方法和上面构造方法形式差不多*/
 public void setEmpty() {
 public boolean set(Region region)
 public boolean set(Rect r)
 public boolean set(int left, int top, int right, int bottom)
 /*往一个 Region 中添加一个 Path，只有这种方法，参数 clip 代表这个 Region 的区域，在里面裁剪出 path
范围的区域*/
 public boolean setPath(Path path, Region clip) //用指定的 Path 和裁剪范围构建一个区域

/**几个判断方法*/
public native boolean isEmpty();//判断该区域是否为空
public native boolean isRect(); //是否是一个矩阵
public native boolean isComplex();//是否是多个矩阵组合


/**一系列的 getBound 方法，返回一个 Region 的边界*/
public Rect getBounds()
public boolean getBounds(Rect r)
public Path getBoundaryPath()
public boolean getBoundaryPath(Path path)
```

```
/**一系列的判断是否包含某点和是否相交*/
public native boolean contains(int x, int y);//是否包含某点
public boolean quickContains(Rect r)   //是否包含某矩阵
public native boolean quickContains(int left, int top, int right,
              
              
            int bottom) //是否
没有包含某矩阵
 public boolean quickReject(Rect r) //是否没和该矩阵相交
 public native boolean quickReject(int left, int top, int right, int bottom); //是否没
和该矩阵相交
 public native boolean quickReject(Region rgn);  //是否没和该矩阵相交

/**几个平移变换的方法*/
public void translate(int dx, int dy)
public native void translate(int dx, int dy, Region dst);
public void scale(float scale) //hide
public native void scale(float scale, Region dst);//hide


/**一系列组合的方法*/
public final boolean union(Rect r)
public boolean op(Rect r, Op op) {
public boolean op(int left, int top, int right, int bottom, Op op)
public boolean op(Region region, Op op)
public boolean op(Rect rect, Region region, Op op)
```

11.3.7　使用类 Typeface

在 Android 系统中，类 Typeface 的完整写法是 Android.Graphics.Typeface。类 Typeface 能够帮助描述一个字体对象。在 TextView 中通过使用方法 setTypeface 可以指定一个输出文本的字体，通过调用直接构造成员方法 create 可以直接指定一个字体名称和样式，例如下面的代码。

```
static Typeface create(Typeface family, int style)
static Typeface create(String familyName, int style)
```

同时使用 isBold 和 isItalic 方法可以判断出是否包含粗体或斜体的字型。

```
final boolean  isBold()
final boolean  isItalic()
```

除此之外，在类 Typeface 中还有从某 APK 资源或向一个具体的文件写入文本的方法，其具体方法如下所示。

```
static Typeface  createFromAsset(AssetManager mgr, String path)
static Typeface  createFromFile(File path)
static Typeface  createFromFile(String path)
```

第 12 章　二维动画应用

在多媒体领域中，动画也是永远的话题之一。动画和简单的图像相比，更具有视觉冲击力。Android 平台为我们提供了一套完整的动画框架，使得开发者可以用它来开发各种动画效果。在本章的内容中，将详细讲解在 Android 系统中实现动画效果的基本知识，为读者进入后面知识的学习打下基础。

12.1 使用 Drawable 实现动画效果

在 Android 系统中，通过类 Drawable 可以实现动画效果，尽管这个类比较抽象。在本节的内容中，将详细讲解使用类 Drawable 实现动画效果的基本知识，为读者进入本书后面知识的学习打下基础。

12.1.1 Drawable 基础

下面先通过一个简单的例子程序来理解它。在这个例子中，使用类 Drawable 的子类 ShapeDrawable 来画图，具体代码如下所示。

```
public class testView extends View {
private ShapeDrawable mDrawable;
public testView(Context context) {
super(context);
int x = 10;
int y = 10;
int width = 300;
int height = 50;
mDrawable = new ShapeDrawable(new OvalShape());
mDrawable.getPaint().setColor(0xff74AC23);
mDrawable.setBounds(x, y, x + width, y + height);
}
protected void onDraw(Canvas canvas)
super.onDraw(canvas);
canvas.drawColor(Color.WHITE);//画白色背景
mDrawable.draw(canvas);
}
}
```

上述代码的实现流程如下所示。

（1）创建一个 OvalShape（椭圆）。

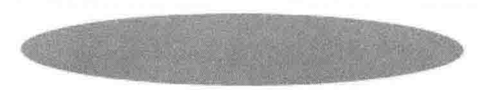
（2）使用刚创建的 OvalShape 构造一个 ShapeDrawable 对象 mDrawable。

（3）设置 mDrawable 的颜色。

（4）设置 mDrawable 的大小。

（5）将 mDrawable 绘制在 testView 的画布上。

上述代码的执行效果如图 12-1 所示。



▲图 12-1　使用类 ShapeDrawable 画图的执行效果

通过这个简单的例子可以帮我们理解什么是 Drawable。Drawable 就是一个可画的对象，可能是一张位图（BitmapDrawable），也可能是一个图形（ShapeDrawable），还有可能是一个图层（LayerDrawable）。在项目中可以根据画图的需求，创建相应的可画对象，就可以将这个可画对象当作一块"画布（Canvas）"，在其上面操作可画对象，并最终将这种可画对象显示在画布上，有点类似于"内存画布"。

12.1.2　使用 Drawable 实现动画效果

本章 12.1.1 节中只是一个简单的使用 Drawable 的例子，完全没有体现出 Drawable 的强大功能。Android SDK 中说明了 Drawable 主要的作用是在 XML 中定义各种动画，然后把 XML 当作 Drawable 资源来读取，通过 Drawable 显示动画。在接下来的内容中，将通过一个具体的演示实例来讲解 Drawable 实现动画效果的方法。

题目	目的	源码路径
实例 12-1	用 Drawable 实现动画效果	\daima\12\testView

本实例是在 12.1.1 节中实例的基础上实现的，具体修改过程如下所示。

（1）去掉文件"layout/main.xml"中的 TextView，增加 ImagView，主要实现代码如下所示。

```
<ImageView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:tint="#55ff0000"
android:src="@drawable/my_image"/>
```

（2）新建一个 XML 文件，命名为 expand_collapse.xml，主要实现代码如下所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<transition xmlns:android="http://schemas.android.com/apk/res/android">
<item android:drawable="@drawable/image_expand"/>
<item android:drawable="@drawable/image_collapse"/>
</transition>
```

准备 3 张 PNG 格式的素材图片，保存到"到 res\drawable"目录下，给 3 张图片分别命名如下。

- my_image.png;
- image_expand.png;
- image_collapse.png。

（3）修改 Activity 中的代码，主要实现代码如下所示。

```
LinearLayout mLinearLayout;
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
mLinearLayout = new LinearLayout(this);
ImageView i = new ImageView(this);
i.setAdjustViewBounds(true);
i.setLayoutParams(new  Gallery.LayoutParams(LayoutParams.WRAP_CONTENT,  LayoutParams.
WRAP_CONTENT));
mLinearLayout.addView(i);
setContentView(mLinearLayout);
Resources res = getResources();
```

```
TransitionDrawable transition =
(TransitionDrawable) res.getDrawable(R.drawable.expand_collapse);
i.setImageDrawable(transition);
transition.startTransition(10000);
}
```

执行后的效果如图 12-2 所示。

初始效果

过渡中效果

最后的效果

▲图 12-2 用 Drawable 实现动画的执行效果

由此可见，执行后在屏幕上的显示形式是：从图片 image_expand.png 过渡到 image_collapse.png，也就是我们在 expand_collapse.xml 中定义的一个 Transition 动画。

12.2 Tween Animation 动画详解

在上一节的内容中，讲解了使用 Drawable 实现动画效果的知识。其实 Drawable 的功能何止如此，Drawable 更加强大的功能是可以显示 Animation。Animation 是以 XML 格式定义的，由于 Tween Animation 与 Frame Animation 的定义、使用都有很大的差异，所以特意将定义好的 XML 文件存放在“res\anim”目录中。

在 Android SDK 中提供了如下两种 Animation。

- Tween Animation：通过对场景里的对象不断做图像变换（平移、缩放、旋转）产生动画效果。
- Frame Animation：顺序播放事先做好的图像，跟电影类似。

由此可见，在 Android 平台中提供了如下两类动画。

- Tween 动画：用于对场景里的对象不断进行图像变换来产生动画效果，Tween 可以把对象进行缩小、放大、旋转和渐变等操作。
- Frame 动画：用于顺序播放事先做好的图像。

在使用 Animation 前，需要先学习如何定义 Animation，这对我们使用 Animation 会有很大的帮助。

12.2.1 Tween 动画基础

在 Android 系统中，Tween 动画通过对 View 的内容实现了一系列的的图形变换操作，通过平移、缩放、旋转、改变透明度来实现动画效果。在 XML 文件中，Tween 动画主要包括以下 4 种动画效果。

- Alpha：渐变透明度动画效果。
- Scale：渐变尺寸伸缩动画效果。
- Translate：画面转移位置移动动画效果。
- Rotate：画面转移旋转动画效果。

在 Android 应用代码中，Tween 动画对应以下 4 种动画效果。

- AlphaAnimation：渐变透明度动画效果。
- ScaleAnimation：渐变尺寸伸缩动画效果。

- TranslateAnimation：画面转换位置移动动画效果。
- RotateAnimation：画面转移旋转动画效果。

在 Android 系统中，Tween 动画预先定义一组指令，这些指令指定了图形变换的类型、触发时间、持续时间。程序沿着时间线执行这些指令就可以实现动画效果。我们可以首先定义 Animation 动画对象，然后设置该动画的一些属性，最后通过方法 startAnimation 来开始实现动画效果。

在接下来的内容中，将通过一个具体的演示实例讲解实现 Tween 动画的 4 种效果的方法。

题目	目的	源码路径
实例 12-2	演示 Tween 动画的 4 种动画效果	\daima\12\myActionAnimation

本实例的具体实现流程如下所示。

（1）编写文件 my_alpha_action.xml，实现 Alpha 渐变透明度动画效果，主要实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android" >
<alpha
android:fromAlpha="0.1"
android:toAlpha="1.0"
android:duration="3000"
/>
<!-- 透明度控制动画效果 alpha
        浮点型值:
        fromAlpha 属性为动画起始时透明度
        toAlpha   属性为动画结束时透明度
        说明:
        0.0 表示完全透明
        1.0 表示完全不透明
        以上值取 0.0～1.0 之间的 float 数据类型的数字
        长整型值:
        duration  属性为动画持续时间
        说明:
                    时间以毫秒为单位
-->
</set>
```

（2）编写文件 my_rotate_action.xml，实现 Rotate 画面转移旋转动画效果，主要实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
<rotate
        android:interpolator="@android:anim/accelerate_decelerate_interpolator"
        android:fromDegrees="0"
        android:toDegrees="+350"
        android:pivotX="50%"
        android:pivotY="50%"
        android:duration="3000" />
<!-- rotate 旋转动画效果
       属性: interpolator 指定一个动画的插入器
        我在试验过程中，使用 android.res.anim 中的资源时候发现有 3 种动画插入器
        accelerate_decelerate_interpolator  加速-减速动画插入器
        accelerate_interpolator          加速-动画插入器
        decelerate_interpolator          减速-动画插入器
        浮点数型值:
        fromDegrees 属性为动画起始时物件的角度
        toDegrees   属性为动画结束时物件旋转的角度，可以大于 360 度
        当角度为负数——表示逆时针旋转
        当角度为正数——表示顺时针旋转
        (负数 from——to 正数:顺时针旋转)
        (负数 from——to 负数:逆时针旋转)
```

```
        (正数 from——to 正数:顺时针旋转)
        (正数 from——to 负数:逆时针旋转)
        pivotX     属性为动画相对于物件的 X 坐标的开始位置
        pivotY     属性为动画相对于物件的 Y 坐标的开始位置
        说明: 以上两个属性值 从 0%~100%中取值, 50%为物件的 X 或 Y 方向坐标上的中点位置
        长整型值: duration 属性为动画持续时间, 时间以毫秒为单位。
-->
</set>
```

（3）编写文件 my_scale_action.xml，实现 Scale 渐变尺寸伸缩动画效果，主要实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
  <scale android:interpolator="@android:anim/accelerate_decelerate_interpolator"
        android:fromXScale="0.0"
        android:toXScale="1.4"
        android:fromYScale="0.0"
        android:toYScale="1.4"
        android:pivotX="50%"
        android:pivotY="50%"
        android:fillAfter="false"
        android:duration="700" />
</set>
<!-- 尺寸伸缩动画效果 scale
      属性: interpolator 指定一个动画的插入器
      有 3 种动画插入器
      accelerate_decelerate_interpolator  加速-减速动画插入器
       accelerate_interpolator         加速-动画插入器
       decelerate_interpolator         减速-动画插入器
       fromXScale 属性为动画起始时 X 坐标上的伸缩尺寸
       toXScale   属性为动画结束时 X 坐标上的伸缩尺寸
       fromYScale 属性为动画起始时 Y 坐标上的伸缩尺寸
       toYScale   属性为动画结束时 Y 坐标上的伸缩尺寸
       以上 4 种属性值

       0.0 表示收缩到没有
       1.0 表示正常无伸缩
       值小于 1.0 表示收缩
       值大于 1.0 表示放大
       pivotX 属性为动画相对于物件的 X 坐标的开始位置
       pivotY 属性为动画相对于物件的 Y 坐标的开始位置
       以上两个属性值从 0%~100%中取值, 50%为物件的 X 或 Y 方向坐标上的中点位置
       duration  属性为动画持续时间, 时间以毫秒为单位
       fillAfter 属性 当设置为 true , 该动画转化在动画结束后被应用
-->
```

（4）编写文件 my_translate_action.xml，实现 Translate 画面转移位置移动动画效果，主要实现代码如下所示。

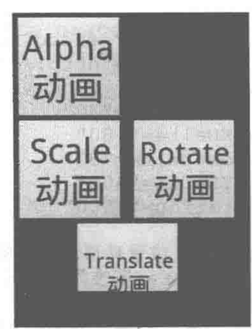
```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
<translate
android:fromXDelta="30"
android:toXDelta="-80"
android:fromYDelta="30"
android:toYDelta="300"
android:duration="2000"
/>
<!-- translate 位置转移动画效果
       fromXDelta 属性为动画起始时 X 坐标上的位置
       toXDelta   属性为动画结束时 X 坐标上的位置
       fromYDelta 属性为动画起始时 Y 坐标上的位置
       toYDelta   属性为动画结束时 Y 坐标上的位置
       没有指定 fromXType toXType fromYType toYType 的时候, 默认是以自己为相对参照物
       duration  属性为动画持续时间, 时间以毫秒为单位
-->
</set>
```

（5）编写文件 myActionAnimation.java，使用 case 语句根据用户的选择来显示对应的动画效果。其主要实现代码如下所示。

```
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);
   button_alpha = (Button) findViewById(R.id.button_Alpha);
   button_alpha.setOnClickListener(this);
   button_scale = (Button) findViewById(R.id.button_Scale);
   button_scale.setOnClickListener(this);
   button_translate = (Button) findViewById(R.id.button_Translate);
   button_translate.setOnClickListener(this);
   button_rotate = (Button) findViewById(R.id.button_Rotate);
   button_rotate.setOnClickListener(this);
}
public void onClick(View button) {
switch (button.getId()) {
   case R.id.button_Alpha: {
      myAnimation_Alpha = AnimationUtils.loadAnimation(this,R.anim.my_alpha_action);
      button_alpha.startAnimation(myAnimation_Alpha);
   }
      break;
   case R.id.button_Scale: {
      myAnimation_Scale= AnimationUtils.loadAnimation(this,R.anim.my_scale_action);
      button_scale.startAnimation(myAnimation_Scale);
   }
      break;
   case R.id.button_Translate: {
      myAnimation_Translate=
AnimationUtils.loadAnimation(this,R.anim.my_translate_action);
      button_translate.startAnimation(myAnimation_Translate);
   }
      break;
   case R.id.button_Rotate: {
      myAnimation_Rotate=
AnimationUtils.loadAnimation(this,R.anim.my_rotate_action);
      button_rotate.startAnimation(myAnimation_Rotate);
   }
      break;
   default:
      break;
   }
}
```

执行后的效果如图 12-3 所示。单击屏幕中的的选项卡会显示对应的动画效果，例如单击“Translate 动画”选项后的效果如图 12-4 所示。

▲图 12-3　Tween 动画的执行效果



▲图 12-4　单击“Translate”选项后的动画效果

12.2.2　Tween 动画类详解

在 Android 系统中的 Tween 动画应用中，存在了如下所示的应用类。

（1）类 AlphaAnimation。

类 AlphaAnimation 是 Android 系统中的透明度变化动画类，用于控制 View 对象的透明度变化，该类继承于类 Animation。类 AlphaAnimation 中的很多方法都与类 Animation 一致，在此类中最常用的方法便是 AlphaAnimation 构造方法，具体原型如下所示。

```
AlphaAnimation(float fromAlpha, float toAlpha)
```

方法 AlphaAnimation 的功能是构建一个渐变透明度动画，各个参数的具体说明如下所示。

- fromAlpha：表示动画起始透明度。
- toAlpha：表示动画结束透明度，其中 0.0 表示完全透明，1.0 表示完全不透明。

（2）尺寸变化动画类 ScaleAnimation。

在 Android 系统中，类 ScaleAnimation 是尺寸变化动画类，用于控制 View 对象的尺寸变化。类 ScaleAnimation 继承于类 Animation，此类中的很多方法都与 Animation 类一致。类 ScaleAnimation 中最常用的方法是构造方法 ScaleAnimation，具体原型如下所示。

```
ScaleAnimation(float fromX, float toX, float fromY, float toY, int pivotXType, float
pivotXValue, int pivotYType, float pivotYValue)
```

构造方法 ScaleAnimation 的功能是构建一个渐变尺寸伸缩动画，各个参数的具体说明如下所示。

- fromX 和 toX：分别表示起始和结束时 x 坐标上的伸缩尺寸。
- fromY 和 toY：分别表示起始和结束时 y 坐标上的伸缩尺寸。
- pivotXValue 和 pivotYValue：分别表示动画相对于物件的 x、y 坐标的开始位置。
- pivotYType 和 pivotXType：分别表示 x、y 的伸缩模式。

（3）位置变化类 TranslateAnimation。

在 Android 系统中，位置变化类 TranslateAnimation 用于控制 View 对象的位置变化。类 TranslateAnimation 继承于类 Animation，在此类中的很多方法都与类 Animation 一致。类 TranslateAnimation 中最常用的方法是构造方法 TranslateAnimation，具体原型如下所示。

```
TranslateAnimation(float fromXDelta, float toXDelta, float fromYDelta, float toYDelta)
```

构造方法 TranslateAnimation 的功能是构建一个画面转换位置移动动画，各个参数的具体说明如下所示。

- fromXDelta：表示起始坐标。
- toXDelta：表示结束坐标。

（4）旋转变化动画类 RotateAnimation。

在 Android 系统中，旋转变化动画类 RotateAnimation 用于控制 View 对象的旋转动作。类 RotateAnimation 继承于类 Animation，在此类中的很多方法都与类 Animation 一致，其中最常用的方法便是构造方法 RotateAnimation，具体原型如下所示。

```
RotateAnimation(float fromDegress, float toDegress, int pivotXType, float pivotXValue,
int pivotYType, float pivotYValue)
```

构造方法 RotateAnimation 的功能是构建一个旋转动画，各个参数的具体说明如下所示。

- fromDegress：表示开始的角度。
- toDegress：表示结束的角度。
- pivotXType 和 pivotYType：分别表示 x、y 的伸缩模式。
- pivotXValue 和 pivotYValue：分别表示伸缩动画相对于 x、y 的坐标的开始位置。

（5）动画抽象类 Animation。

在 Android 系统中，所有其他一些动画类都要继承类 Animation 中的实现方法。类 Animation 主要用于补间动画效果，提供了动画启动、停止、重复、持续时间等方法。在类 Animation 中的方法适用于任何一种补间动画对象，此类中的常用方法如下所示。

- setDuration。

方法 setDuration 用于设置动画的持续时间，以毫秒为单位，具体原型如下所示。

```
public void setDuration (long durationMillis)
```

其中，参数 durationMillis 为动画的持续时间，单位为毫秒（ms）。

方法 setDuration 是设置补间动画时间长度的主要方法，使用非常普遍。

- startNow。

在 Android 系统中，方法 startNow 用于启动执行一个动画。此方法是启动执行动画的主要方法，使用时需要先通过方法 setAnimation 为某一个 View 对象设置动画。另外，用户在程序中也可以使用 View 组件的 startAnimation 方法来启动执行动画。方法 startNow 的具体原型如下所示。

```
public void startNow ()
```

- start。

在 Android 系统中，方法 start 用于启动执行一个动画。方法 start 是启动执行动画的另一个主要方法，使用时需要先通过 setAnimation 方法为某一个 View 对象设置动画。start 方法区别于 startNow 方法的地方在于，方法 start 可以用于在 getTransformation 方法被调用时启动动画，具体原型如下所示。

```
public void start ()
```

方法 start 的执行效果类似于方法 startNow，在此不再赘述。

- cancel。

在 Android 系统中，方法 cancel 用于取消一个动画的执行。方法 cancel 是取消一个正在执行中的动画的主要方法，此方法和 startNow 方法结合可以实现对动画执行过程的控制。在此需要注意的是，当通过方法 cancel 取消动画时，必须使用 reset 方法或者 setAnimation 方法重新设置，才可以再次执行动画。

方法 cancel 的具体原型如下所示。

```
public void cancel ()
```

- setRepeatCount。

在 Android 系统中，方法 setRepeatCount 用于设置一个动画效果重复执行的次数。Android 系统默认每个动画仅执行一次，通过该方法可以设置动画执行多次。方法 setRepeatCount 的具体原型如下所示。

```
public void setRepeatCount (int repeatCount)
```

其中，参数 repeatCount 表示重复执行的次数。如果设置为 n，则动画将执行 n+1 次。

- setFillEnabled。

在 Android 系统中，方法 setFillEnabled 用于使程序能实现填充效果。当该方法设置为 true 时，将执行 setFillBefore 和 setFillAfter 方法进行填充，否则将忽略 setFillBefore 和 setFillAfter 方法。方法 setFillEnabled 的具体原型如下所示。

```
public void setFillEnabled (boolean fillEnabled)
```

其中，参数 fillEnabled 表示是否使能填充效果，true 表示使用该效果，false 表示禁用该效果。

- setFillBefore。

在 Android 系统中，方法 setFillBefore 用于设置一个动画效果执行完毕后，View 对象返回到起始的位置。方法 setFillBefore 的效果是系统默认的效果，在执行该方法时需要首先通过 setFillEnabled 方法设置能够实现填充效果，否则设置无效。方法 setFillBefore 的具体原型如下所示。

```
public void setFillBefore (boolean fillBefore)
```

其中，参数 fillBefore 为是否执行起始填充效果，true 表示使用该效果，false 表示禁用该效果。

- setFillAfter。

在 Android 系统中，方法 setFillAfter 用于设置一个动画效果执行完毕后，View 对象保留在终止的位置。在执行方法 setFillAfter 时，需要首先通过 setFillEnabled 方法实现填充效果，否则设置无效。方法 setFillAfter 的具体原型如下所示。

```
public void setFillAfter (boolean fillAfter)
```

其中，参数 fillAfter 为是否执行终止填充效果，true 表示使用该效果，false 表示禁用该效果。

- setRepeatMode。

在 Android 系统中，方法 setRepeatMode 用于设置一个动画效果执行的重复模式。在 Android 系统中提供了好几种重复模式，其中最主要的便是 RESTART 模式和 REVERSE 模式。方法 setRepeatMode 的具体原型如下所示。

```
public void setRepeatMode (int repeatMode)
```

其中，参数 repeatMode 为动画效果的重复模式，常用的取值如下。

- RESTART：表示重新从头开始执行。
- REVERSE：表示反方向执行。

- setStartOffset。

在 Android 系统中，方法 setStartOffset 用于设置一个动画执行的启动时间，单位为毫秒。系统默认当执行 start 方法后立刻执行动画，当使用该方法设置后，将延迟一定的时间再启动动画。方法 setStartOffset 的具体原型如下所示。

```
public void setStartOffset (long startOffset)
```

其中，参数 startOffset 表示动画的启动时间，单位是毫秒（ms）。

```
public void startAnimation (Animation animation)
```

12.2.3 Tween 应用实战

经过本节前面内容的学习，已经详细讲解了在 Android 系统中进行 Tween 动画开发的基本知识。在接下来的内容中，将通过两个具体的演示实例来讲解在 Android 系统中实现 Tween 动画效果的方法。

题目	目的	源码路径
实例 12-3	在 Android 中实现 Tween 动画效果	\daima\12\TweenL

实例文件 TweenL.java 的主要实现代码如下所示。

```
    /* 定义 Alpha 动画 */
    private Animation mAnimationAlpha = null;

    /* 定义 Scale 动画 */
    private Animation mAnimationScale = null;

    /* 定义 Translate 动画 */
    private Animation mAnimationTranslate = null;

    /* 定义 Rotate 动画 */
    private Animation mAnimationRotate = null;

    /* 定义 Bitmap 对象 */
    Bitmap              mBitQQ              = null;

    public example9(Context context)
    {
        super(context);

        /* 装载资源 */
        mBitQQ = ((BitmapDrawable) getResources().getDrawable(R.drawable.qq)).getBitmap();
    }

    public void onDraw(Canvas canvas)
    {
        super.onDraw(canvas);

        /* 绘制图片 */
        canvas.drawBitmap(mBitQQ, 0, 0, null);
    }

    public boolean onKeyUp(int keyCode, KeyEvent event)
    {
        switch ( keyCode )
        {
        case KeyEvent.KEYCODE_DPAD_UP:
            /* 创建 Alpha 动画 */
            mAnimationAlpha = new AlphaAnimation(0.1f, 1.0f);
            /* 设置动画的时间 */
            mAnimationAlpha.setDuration(3000);
            /* 开始播放动画 */
            this.startAnimation(mAnimationAlpha);
            break;
        case KeyEvent.KEYCODE_DPAD_DOWN:
            /* 创建 Scale 动画 */
            mAnimationScale =new ScaleAnimation(0.0f, 1.0f, 0.0f, 1.0f,
                                Animation.RELATIVE_TO_SELF, 0.5f,
                                Animation.RELATIVE_TO_SELF, 0.5f);
            /* 设置动画的时间 */
            mAnimationScale.setDuration(500);
            /* 开始播放动画 */
            this.startAnimation(mAnimationScale);
            break;
        case KeyEvent.KEYCODE_DPAD_LEFT:
            /* 创建 Translate 动画 */
            mAnimationTranslate = new TranslateAnimation(10, 100,10, 100);
            /* 设置动画的时间 */
            mAnimationTranslate.setDuration(1000);
            /* 开始播放动画 */
            this.startAnimation(mAnimationTranslate);
            break;
        case KeyEvent.KEYCODE_DPAD_RIGHT:
            /* 创建 Rotate 动画 */
            mAnimationRotate=new RotateAnimation(0.0f, +360.0f,
                                Animation.RELATIVE_TO_SELF,0.5f,
                                Animation.RELATIVE_TO_SELF, 0.5f);
            /* 设置动画的时间 */
            mAnimationRotate.setDuration(1000);
            /* 开始播放动画 */
```

```
            this.startAnimation(mAnimationRotate);
            break;
        }
        return true;
    }
}
```

执行后可以通过键盘的上、下、左、右键实现动画效果，如图 12-5 所示。

▲图 12-5 Tween 动画的执行效果

我们知道，在 Android 系统中提供了两种使用 Tween Animation 的方法，分别是直接从 XML 资源中读取 Animation 和使用 Animation 子类的构造函数来初始化 Animation 对象。本实例将演示从 XML 资源中读取 Animation，并实现 Tween 动画效果的过程。

题目	目的	源码路径
实例 12-4	从 XML 资源中读取 Animation	\daima\12\testDrawable

本实例具体实现流程如下所示。

（1）创建 Android 工程，然后导入一张图片资源。

（2）将“res\layout\main.xml”目录中的 TextView 转换为 ImageView。

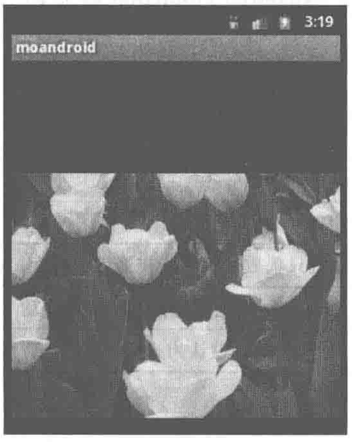
（3）在“res”目录下创建新的文件夹“anim”，并在此文件夹下面定义 Animation XML 文件。

（4）修改 OnCreate()中的代码，显示动画资源。

文件 testDrawable.java 的主要实现代码如下所示。

```
public class testDrawable extends Activity {
    LinearLayout mLinearLayout;
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ImageView spaceshipImage = (ImageView) findViewById(R.id.spaceshipImage);
        Animation    hyperspaceJumpAnimation    =    AnimationUtils.loadAnimation(this,
R.anim.hyperspace_jump);
        spaceshipImage.startAnimation(hyperspaceJumpAnimation);
    }
}
```

在上述代码中，AnimationUtils 提供了加载动画的函数，除了函数 loadAnimation()，读者可以到 Android SDK 中去详细了解其他函数。执行效果如图 12-6 所示。



▲图 12-6 从 XML 资源中读取 Animation 的执行效果

12.3 实现 Frame Animation 动画效果

在我们日常生活中见到的最多的可能就是 Frame 动画了，Android 中当然也少不了它。在本节的内容中，将简要介绍 Frame 动画的基本知识，并通过具体实例讲解实现 Frame 动画的流程，为读者进入本书后面知识的学习打下基础。

12.3.1　Frame 动画基础

在 Android SDK 中，可以通过类 AnimationDrawable 来定义并使用 Frame Animation，与此相关的 SDK 的位置如下所示。

- Tween animation：android.view.animation 包。
- Frame animation：android.graphics.drawable.AnimationDrawable 类。

在 Android SDK 中，AnimationDrawable 的功能是获取、设置动画的属性，其中最为常用的方法如下所示。

- int getDuration()：功能是获取动画的时长。
- int getNumberOfFrames()：功能是获取动画的帧数。
- boolean isOneShot()：功能是获取 oneshot 属性。
- Void setOneShot(boolean oneshot)：功能是设置 oneshot 的属性。
- void inflate(Resurce r,XmlPullParser p,AttributeSet attrs)：功能是增加、获取帧动画。
- Drawable getFrame(int index)：功能是获取某帧的 Drawable 资源。
- void addFrame(Drawable frame,int duration) ：功能是为当前动画增加帧（资源，持续时长）。
- void start()：表示开始动画。
- void run()：表示外界不能直接调用，使用 start()替代。
- boolean isRunning()：表示当前动画是否在运行。
- void stop()：表示停止当前动画。

在 Android 应用中，可以在 XML Resource 中定义 Frame Animation，此时也是存放到"res\anim"目录下。另外，也可以使用 AnimationDrawable 中的 API 来定义 Frame Animation。

因为 Tween Animation 与 Frame Animation 有着很大的不同，所以定义 XML 的格式也完全不一样。定义 Frame Animation 的格式是：首先是 animation-list 根节点，其中包含多个 item 子节点，每个 item 节点定义一帧动画，定义当前帧的 Drawable 资源和当前帧持续的时间。在表 12-1 中对节点中的元素进行了详细说明。

表 12-1　XML 属性元素说明

XML 属性	说　明
drawable	当前帧引用的 Drawable 资源
duration	当前帧显示的时间（毫秒为单位）
oneshot	如果为 true，表示动画只播放一次停止在最后一帧上；如果设置为 false，表示动画循环播放
variablePadding	如果为 true，允许 Drawable 根据被选择的现状而变动
visible	规定 Drawable 的初始可见性，默认为 flase

12.3.2　使用 Frame 动画

在 Android 多媒体开发应用中，使用 Frame 动画的方法十分简单，基本流程如下所示。

（1）首先创建一个 AnimationDrawabledF 对象来表示 Frame 动画。

（2）然后通过 addFrame 方法把每一帧要显示的内容添加进去。

（3）最后通过 start 方法就可以播放这个动画了，同时还可以通过 setOneShot 方法设置是否重复播放。

在 Android 多媒体开发应用中，Frame 动画主要是通过类 AnimationDrawable 来实现的，通过此类中的方法 start 和 stop 分别启动和停止动画。Frame 动画一般通过 XML 文件进行配置，可以在工程中的“res/anim”目录下创建一个 XML 配置文件，该配置文件有一个<animation-list>根元素和若干个<item>子元素。

在接下来的内容中，将通过一个具体的演示实例讲解在 Android 中实现 Frame 动画效果的方法。

题目	目的	源码路径
实例 12-5	在 Android 中实现 Frame 动画效果	\daima\12\FrameL

实例文件 FrameL.java 的主要代码如下所示。

```
/* 定义 AnimationDrawable 动画 */
private AnimationDrawable frameAnimation = null;
Context mContext = null;
/* 定义一个 Drawable 对象 */
Drawable mBitAnimation = null;
public FrameL(Context context)
{
   super(context);

   mContext = context;

   /* 实例化 AnimationDrawable 对象 */
   frameAnimation = new AnimationDrawable();

   /* 装载资源 */
   //这里用一个循环装载所有名字类似的资源
   //如 "a1.......12.png" 的图片
   //这个方法用处非常大
   for (int i = 1; i <= 15; i++)
   {
      int id = getResources().getIdentifier("a" + i, "drawable", mContext.
      getPackageName());
      mBitAnimation = getResources().getDrawable(id);
      /* 为动画添加一帧 */
      //参数 mBitAnimation 是该帧的图片
      //参数 500 是该帧显示的时间,按毫秒计算
      frameAnimation.addFrame(mBitAnimation, 500);
   }

   /* 设置播放模式是否循环，false 表示循环，而 true 表示不循环 */
   frameAnimation.setOneShot( false );

   /* 设置本类将要显示这个动画 */
   this.setBackgroundDrawable(frameAnimation);
}

public void onDraw(Canvas canvas)
{
   super.onDraw(canvas);

}

public boolean onKeyUp(int keyCode, KeyEvent event)
{
   switch ( keyCode )
```

```
        {
        case KeyEvent.KEYCODE_DPAD_UP:
            /* 开始播放动画 */
            frameAnimation.start();
            break;
        }
        return true;
    }
}
```

执行后可以通过按下键盘的上、下方向键的方式实现动画效果，执行效果如图 12-7 所示。

▲图 12-7 Frame 动画的执行效果

12.4 Property Animation 动画

Android 3.0 推出了一种全新的动画系统属性动画 Property Animation，这是一个全新的可伸缩的动画框架。Property Animation 允许我们将动画效果应用到任何对象的任意属性上，例如 View、Drawable、Fragment、Object 等。通常我们可以为对象的 int、float 和 16 进制的颜色值定义很多动画因素，例如持续时间、重复次数、插入器等。当一个对象有属性使用了这些类型后，就可以随时改变这些值以影响动画效果。在本节的内容中，将详细讲解属性动画 Property Animation 的基本知识，为读者进入本书后面知识的学习打下基础。

12.4.1 Property Animation（属性）动画基础

属性动画系统是一个功能强大的框架，无论是否将它绘制到屏幕上，我们都可以定义一个可以改变任何对象的属性的方法，以随着时间的推移而形成动画效果。通过属性动画，可以设置一个对象在屏幕中的位置、动画时间和动画之间的距值。

在 Android 系统中，通过属性动画框架可以定义如下特点的动画。

- Duration（时间）：可以指定动画的持续时间，默认长度是 300 毫秒。
- Time interpolation（时间插值）：定义了动画变化的频率。
- Repeat count and behavior（重复计数和行为）：可以指定是否有一个动画的重复，还可以指定是否要反向播放动画，也可以设置重复播放的次数。
- Animator Sets（动画设置）：可以按照一定的逻辑设置来组织动画，例如同时播放、按顺序播放或指定延迟播放。
- Frame refresh delay（帧刷新延迟）：可以指定如何经常刷新动画帧。默认设置每 10 毫秒刷新，但在应用程序中可以指定刷新帧的速度，这最终取决于系统整体的状态，提供多快服务的速度依据底层的定时器。

在 android.animation 里面保存了属性动画系统的大部分 API。因为视图的动画系统已经在 android.view.animation 中定义了许多值，所以在此可以使用属性动画系统的值。在类 Animator 中提供了用于创建动画的基本结构，但是通常不能直接使用这个类，因为它直接提供最基本的功能必须被扩展到完全支持的动画值。在表 12-2 中，列出了类 Animator 中的子类扩展。

表 12-2　类 Animator 中的子类扩展

类	描　述
ValueAnimator	属性动画时序引擎也计算属性动画的值，拥有所有的核心功能，能够计算动画值，并包含每个动画，可以设置动画是否重复，并设置自定义类型的功能。在类 ValueAnimator 中有两个重要属性：动画值计算和设置这些对象的属性动画值。因为 ValueAnimator 不进行二次处理，所以一定要更新计算 ValueAnimator 的值并修改想用自己的逻辑动画的对象
ObjectAnimator	ValueAnimator 的子类，允许你设置一个目标对象和对象属性的动画。当计算出一个新的动画值，本类更新相应的属性。大部分情况使用 ObjectAnimator，因为它使动画的目标对象的值更简单。有时直接使用 ValueAnimator，因为 ObjectAnimator 有一些限制，如对目标对象目前要求的具体 acessor 方法
AnimatorSet	提供机制，以组合动画并让它们关联性运行。可以设置动画一起播放，按播放顺序，或在指定的延迟之后播放

在属性动画系统中提供了如表 12-3 所示的 Evaluators。

表 12-3　Evaluators 中的接口

类/接口	描　述
IntEvaluator	默认的计算器来计算 int 属性值
FloatEvaluator	默认评估值来计算浮动属性
ArgbEvaluator	默认的计算器计算值表示为十六进制值的色彩属性
TypeEvaluator	允许创建自己的评估。如果是动画对象的属性，而不是一个整数、浮点数或颜色，则必须实现 TypeEvaluator 接口以指定如何计算对象属性的动画值。若想对比默认行为来处理这些类型的不同，也可以指定自定义的 TypeEvaluator 整数、浮点数、颜色值

在属性动画系统中，Interpolators（时间插补）定义了如何将一个动画的特定值作为时间函数进行计算的描述。例如可以指定在整个动画系统中实现线性动画，这意味着能够整个时间段内均匀地移动以实现动画效果。当然可以指定实现非线性时间动画效果，例如实现一个在开始时加速，在最后减速的动画效果。在下面的表 12-4 中列出了 android.view.animation 中内置的 Interpolators。

表 12-4　内置的 Interpolators

类/接口	描　述
AccelerateDecelerateInterpolator	慢慢开始和结束，在中间加速
AccelerateInterpolator	开始缓慢，然后加快
AnticipateInterpolator	开始时向后，然后向前甩
AnticipateOvershootInterpolator	开始的时候向后，然后向前甩一定值后返回最后的值
BounceInterpolator	动画结束的时候弹起
CycleInterpolator	动画循环播放特定的次数，速率改变沿着正弦曲线
DecelerateInterpolator	在动画开始的地方快，然后慢
LinearInterpolator	以常量速率改变
OvershootInterpolator	向前甩一定值后再回到原来位置
TimeInterpolator	这是一个接口，可以自定义实现 Interpolators

12.4.2　使用 Property Animation

经过本节前面内容的学习，已经了解了 Property Animation 的基本知识。在接下来的内容中，将详细讲解使用 Property Animation 的基本方法。

（1）使用 ValueAnimator 创建动画。

在 Android 系统中，可以通过调用工厂方法 ofInt()、ofFloat()和 ofObject()的方式获取 ValueAnimator 实例。例如下面的代码。

```
ValueAnimator animation = ValueAnimator.ofFloat(0f, 1f);
animation.setDuration(1000);
animation.start();
```

通过上述代码，实现了在 1 000ms 内值从 0～1 的变化。当然可以通过如下代码实现一个自定义的 Evaluator。

```
ValueAnimator animation = ValueAnimator.ofObject(new MyTypeEvaluator(),
startPropertyValue, endPropertyValue);
animation.setDuration(1000);
animation.start();
```

在上述代码中，ValueAnimator 只是计算了在动画过程中发生变化的值，而没有把这些计算出来的值应用到具体的对象上面，所以也不会显示出任何动画。要把计算出来的值应用到对象上，必须为 ValueAnimator 注册一个监听器 ValueAnimator.AnimatorUpdateListener，我们可以通过该监听器更新对象的属性值。在实现监听器 ValueAnimator.AnimatorUpdateListener 时，可以通过 getAnimatedValue()的方法获取当前帧的值。

（2）使用 ObjectAnimator 创建动画。

在 Android 系统中，ObjectAnimator 与 ValueAnimator 之间的区别如下所示。

- ObjectAnimator 可以直接将在动画过程中计算出来的的值应用到一个具体对象的属性上；
- ValueAnimator 需要先另外注册一个监听器，然后才可以将在动画过程中计算出来的的值应用到一个具体对象的属性上。

由此可见，当使用 ObjectAnimator 时不再需要实现 ValueAnimator.AnimatorUpdateListener。

具体实例化 ObjectAnimator 的过程跟 ValueAnimator 的过程类似，但是需要额外指定具体的对象和对象的属性名（字符串形式）。例如下面的代码。

```
ObjectAnimator anim = ObjectAnimator.ofFloat(foo, "alpha", 0f, 1f);
anim.setDuration(1000);
anim.start();
```

为了能够让 ObjectAnimator 正常运作，需要注意如下所示的 3 点。

- 要为对应的对象提供 setter 方法，例如在上面代码中需要为对象 foo 添加方法 setAlpha(float value)。在不能修改对象源码的情况下，不能先对对象进行封装（extends），或者使用 ValueAnimator。
- 如果 ObjectAnimator 的工厂方法中的参数 values 提供了一个值（需要提供起始值和结束值），那么该值会被认为是结束值，需要通过对象中的方法 getter 提供起始值。并且在这种情况下，需要提供对应属性的 getter 方法。例如下面的代码。

```
ObjectAnimator.ofFloat(targetObject, "propName", 1f)
```

- 如果动画的对象是 View，那么可能需要在回调函数 onAnimationUpdate()中调用方法 View.invalidate()来刷新屏幕，例如设置 Drawable 对象的属性 color。但是因为 View 中的所有 setter 方法（例如 setAlpha() and setTranslationX()）会自动地调用方法 invalidate()，所以不需要额外调用方法 invalidate()。

（3）使用 AnimatorSet 排列多个 Animator。

有时需要在动画的开始依赖于其他动画的开始或结束，这时可以使用 AnimatorSet 来绑定这些 Animator 了。例如下面的代码。

```
AnimatorSet bouncer = new AnimatorSet();
bouncer.play(bounceAnim).before(squashAnim1);
bouncer.play(squashAnim1).with(squashAnim2);
bouncer.play(squashAnim1).with(stretchAnim1);
bouncer.play(squashAnim1).with(stretchAnim2);
bouncer.play(bounceBackAnim).after(stretchAnim2);
ValueAnimator fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f);
fadeAnim.setDuration(250);
AnimatorSet animatorSet = new AnimatorSet();
animatorSet.play(bouncer).before(fadeAnim);
animatorSet.start();
```

在上述代码中，动画会按照如下所示的顺序执行。

- 播放 bounceAnim 动画。
- 同时播放 squashAnim1、squashAnim2、stretchAnim1 和 stretchAnim2。
- 播放 bounceBackAnim。
- 播放 fadeAnim。

（4）使用 Animation 监听器。

在 Android 开发应用中，可以使用下面的监听器来监听重要的事件。

- Animator.AnimatorListener。

➢ onAnimationStart()：在动画启动时调用。

➢ onAnimationEnd()：在动画结束时调用。

➢ onAnimationRepeat()：在动画重新播放时调用。

➢ onAnimationCancel()：在动画被 Cancel 时调用，一个被 Cancel 的动画也会调用 onAnimationEnd()。

- ValueAnimator.AnimatorUpdateListener。

➢ onAnimationUpdate()：在动画的每一帧上调用，在这个方法中可以使用 ValueAnimator 的 getAnimatedValue()方法来获取计算出来的值。此监听器一般只适用于 ValueAnimator，另外可能需要在这个方法中调用 View.invalidate()方法来刷新屏幕的显示。

在 Android 开发应用中，可以用继承适配器 AnimatorListenerAdapter 来代替对 Animator.AnimatorListener 的接口的实现，接下来只需要实现我们所关心的方法即可。例如下面的代码。

```
ValueAnimatorAnimator fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f);
fadeAnim.setDuration(250);
fadeAnim.addListener(new AnimatorListenerAdapter() {
public void onAnimationEnd(Animator animation) {
   balls.remove(((ObjectAnimator)animation).getTarget());
}
```

（5）使用 ViewPropertyAnimator 创建动画。

在 Android 开发应用中，使用 ViewPropertyAnimator 可以根据 View 的多个属性值来创建动画。其中用多个 ObjectAnimator 方式创建动画的代码如下所示。

```
ObjectAnimator animX = ObjectAnimator.ofFloat(myView, "x", 50f);
ObjectAnimator animY = ObjectAnimator.ofFloat(myView, "y", 100f);
AnimatorSet animSetXY = new AnimatorSet();
animSetXY.playTogether(animX, animY);
animSetXY.start();
```

使用单个 ObjectAnimator 方式创建动画的代码如下所示。

```
PropertyValuesHolder pvhX = PropertyValuesHolder.ofFloat("x", 50f);
PropertyValuesHolder pvhY = PropertyValuesHolder.ofFloat("y", 100f);
ObjectAnimator.ofPropertyValuesHolder(myView, pvhX, pvyY).start();
```

使用 ViewPropertyAnimator 方式创建 View 多属性变化动画的代码如下所示。

```
myView.animate().x(50f).y(100f);
```

（6）使用 Keyframes 方式创建动画。

在 Android 开发应用中，对象 Keyframe 由 elapsed fraction/value 组成，并且对象 Keyframe 可以使用插值器。例如下面的演示代码。

```
Keyframe kf0 = Keyframe.ofFloat(0f, 0f);
Keyframe kf1 = Keyframe.ofFloat(.5f, 360f);
Keyframe kf2 = Keyframe.ofFloat(1f, 0f);
PropertyValuesHolder pvhRotation = PropertyValuesHolder.ofKeyframe("rotation", kf0, kf1,
kf2);
ObjectAnimator rotationAnim = ObjectAnimator.ofPropertyValuesHolder(target,
pvhRotation)
rotationAnim.setDuration(5000ms);
```

（7）在 ViewGroup 布局改变时应用动画。

在 Android 开发应用中，当 ViewGroup 布局发生改变时可能想要用动画的形式来体现，例如 ViewGroup 中的 View 消失或显示的时候。ViewGroup 可以通过方法 setLayoutTransition(LayoutTransition) 来设置一个布局转换的动画。在 LayoutTransition 中可以通过调用方法 setAnimator()的方式来设置 Animator，并且还需要向这个方法传递一个 LayoutTransition 标志常量，通过这个常量来设置在什么时候执行这个 animator。在这个过程中，有如下所示可用的常量。

- APPEARING：功能是设置 Layout 中的 view 正要显示的时候运行动画。
- CHANGE_APPEARING：功能是设置 Layout 中因为有新的 view 加入而改变 layout 时运行动画。
- DISAPPEARING：功能是设置 Layout 中的 view 正要消失的时候运行动画。
- CHANGE_DISAPPEARING：功能是设置 Layout 中有 view 消失而改变 layout 时运行动画。

如果想要使用系统默认的 ViewGroup 布局改变时的动画，只需将属性 android:animateLayoutchanges 设置为 true 即可。例如下面的演示代码。

```
<LinearLayout
      android:orientation="vertical"
      android:layout_width="wrap_content"
      android:layout_height="match_parent"
      android:id="@+id/verticalContainer"
      android:animateLayoutChanges="true"
/>
```

在接下来的内容中，将通过一个具体的演示实例来讲解在 Android 系统中使用属性动画的方法。

题目	目的	源码路径
实例 12-6	在 Android 中使用属性动画	\daima\12\shuxing

本实例的功能比较简单，通过调用 ValueAnimator 中的方法 ofFloat 来实现动画效果。本实例的具体实现流程如下所示。

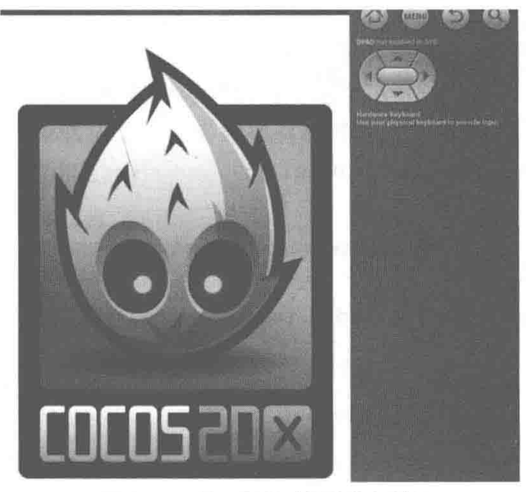
（1）编写布局文件 activity_main.xml，在界面中插入一副指定的图片，主要实现代码如下所示。

```
<ImageView
    android:id="@+id/imageView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_alignParentTop="true"
    android:src="@drawable/cool"
/>
```

（2）编写文件 MainActivity.java，主要实现代码如下所示。

```
public class MainActivity extends Activity {

    private Bitmap bm;
    private ValueAnimator animator;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        BitmapDrawable m=(BitmapDrawable)getResources().getDrawable(R.drawable.cool);
        bm=m.getBitmap();
        animator= ValueAnimator.ofFloat(0f, 1f);
        animator.setDuration(1000);
        animator.setTarget(bm);
        animator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
            public void onAnimationUpdate(ValueAnimator animation) {

            }
        });
        animator.start();
    }
```

执行后将在屏幕中实现简易的动画效果，如图 12-8 所示。



▲图 12-8　使用属性动画的执行效果

12.5 实现动画效果的其他方法

经过本章前面内容学习，已经讲解了在 Android 系统中实现动画效果的主要方法。其实在 Android 多媒体开发应用中，还可以通过其他方法实现动画效果。在本节的内容中，将介绍其他几种在 Android 系统中实现动画效果的方法。

12.5.1　播放 GIF 动画

在默认情况下，在 Android 平台上是不能播放 GIF 动画的。要想播放 GIF 动画，需要先对 GIF 图像进行解码，然后将 GIF 中的每一帧取出来并保存到一个容器中，然后根据需要连续绘制每一帧，这样就可以轻松地实现了 GIF 动画的播放。

在接下来的内容中，将通过一个具体的演示实例讲解在 Android 中播放 GIF 动画的方法。

题目	目的	源码路径
实例 12-7	在 Android 中播放 GIF 动画	\daima\12\GIFL

本实例的具体实现流程如下所示。

编写文件 GameView.java，此文件是本实例的核心，功能是解析 GIF 动画文件并设置显示效果。主要实现代码如下所示。

```
public class GameView extends View implements Runnable
{
    Context mContext = null;
    /* 声明 GifFrame 对象 */
    GifFrame mGifFrame = null;
    public GameView(Context context)
    {
        super(context);
        mContext = context;
        /* 解析 GIF 动画 */
    mGifFrame=GifFrame.CreateGifImage(fileConnect(this.getResources().
    openRawResource(R.drawable.gif1)));
        /* 开启线程 */
        new Thread(this).start();
    }
    public void onDraw(Canvas canvas)
    {
        super.onDraw(canvas);
        /* 下一帧 */
        mGifFrame.nextFrame();
        /* 得到当前帧的图片 */
        Bitmap b=mGifFrame.getImage();

        /* 绘制当前帧的图片 */
        if(b!=null)
            canvas.drawBitmap(b,10,10,null);
    }

    /*线程处理*/
    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
        {
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
            //使用 postInvalidate 可以直接在线程中更新界面
            postInvalidate();
        }
    }
    /* 读取文件 */
    public byte[] fileConnect(InputStream is)
    {
        try
```

```
        {ByteArrayOutputStream baos = new ByteArrayOutputStream();
           int ch = 0;
           while( (ch = is.read()) != -1)
           {
              baos.write(ch);
           }
           byte[] datas = baos.toByteArray();
           baos.close();
           baos = null;
           is.close();
           is = null;
           return datas;
        }
        catch(Exception e)
        {
           return null;
        }
     }
  }
```

在 Android 系统中，可以通过方法 AnimationDrawable 实现支持逐帧播放的功能。由此可见，要想在 Android 中播放 GIF 动画，需要先把 GIF 图片打散使之成为由单一帧构成的图片。在现实应用中，可以使用第三方软件帮助打散图片，例如 GIFSplitter。分割成功后会得到独立的帧文件，接下来就可以在 res 目录下新建 anim 动画文件夹，例如如下所示的代码。

```
<?xml version="1.0" encoding="UTF-8"?>
<animation-list android:oneshot="false"
   xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:duration="150" android:drawable="@drawable/xiu0" />
   <item android:duration="150" android:drawable="@drawable/xiu1" />
   <item android:duration="150" android:drawable="@drawable/xiu2" />
   <item android:duration="150" android:drawable="@drawable/xiu3" />
</animation-list>
```

在上述代码中，xiu0、xiu1、xiu2 和 xiu3 是用第三方软件分割后生成的独立帧文件名。通过上述代码，对应的 item 为顺序的图片从开始到结束，duration 用于为每张逐帧播放间隔。如果 oneshot 设置为 false 表示循环播放，设置为 true 表示播放一次后就停止。这样就可以使用 AnimationDrawable 对象获得图片的图片，然后指定这个 AnimationDrawable 开始播放 GIF 动画。但是此时还有一个问题：不会默认播放，必须要有事件触发才可播放动画，例如通过如下代码实现单击监听触发动画的播放功能。

```
import android.app.Activity;
import android.graphics.drawable.AnimationDrawable;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ImageView;
public class animActivity extends Activity implements OnClickListener {
   ImageView iv = null;
   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      iv = (ImageView) findViewById(R.id.ImageView01);
      iv.setOnClickListener(this);
   }
   @Override
   public void onClick(View v) {
      // TODO Auto-generated method stub
      AnimationDrawable anim = null;
      Object ob = iv.getBackground();
      anim = (AnimationDrawable) ob;
```

```
        anim.stop();
        anim.start();
    }
}
```

12.5.2　实现 EditText 动画特效

在接下来的内容中，将通过一个具体的演示实例来讲解实现 EditText 振动特效的方法。

题目	目的	源码路径
实例 12-8	在 Android 中实现 EditText 振动特效	\daima\12\GIFL

本实例的具体实现流程如下所示。

（1）编写文件 animation_1.xml，在里面分别插入一个 EditTex 输入框控件和一个 Button 按钮控件。主要实现代码如下所示。

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="10dip"
    android:text="@string/animation_1_instructions"
/>

<EditText android:id="@+id/pw"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:clickable="true"
    android:singleLine="true"
    android:password="true"
/>
<Button android:id="@+id/login"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/googlelogin_login"
/>
```

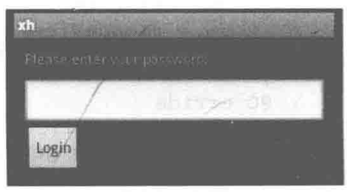
（2）编写文件 shake.xml，在里面设置特效的振动时间。其主要实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<translate xmlns:android="http://schemas.android.com/apk/res/android"
android:fromXDelta="0"
android:toXDelta="10"
android:duration="1000"
android:interpolator="@anim/cycle_7" />
```

（3）编写文件 Animation1.java，其主要实现代码如下所示。

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.animation_1);
    View loginButton = findViewById(R.id.login);
    loginButton.setOnClickListener(this);
}
public void onClick(View v) {
    Animation shake = AnimationUtils.loadAnimation(this,
R.anim.shake);
    findViewById(R.id.pw).startAnimation(shake);
}
```

执行后的效果如图 12-9 所示，单击“Login”按钮的时候 EditText 就会振动，具体怎么振动，振动多久，振动几次等都是由 XML 文件中的配置参数指定的。



▲图 12-9　EditText 振动的执行效果

第 13 章　渲染二维图像

渲染是指对事物的描写、形容和烘托处理，目的是烘染物像，增强艺术效果、质感和立体感。在 Android 多媒体开发应用中，经常因为项目需求需要渲染屏幕中的二维图形图像。在本章的内容中，将详细讲解在 Android 系统中渲染二维图像的基本知识，为读者进入本书后面知识的学习打下基础。

13.1 使用渲染类 Shader

在 Android 系统中，可以通过类 Shader 渲染屏幕中的图像以及几何图形。在类 Shader 中包含了如下所示的常用直接子类。

（1）类 BitmapShader：主要用来渲染图像。

通过使用渲染器类 BitmapShader 可以将一个位图着色为一个纹理，并且可以重复或设置位图的模式。类 BitmapShader 的核心方法如下所示。

```
public BitmapShader(Bitmap bitmap,Shader.TileMode tileX,Shader.TileMode tileY)
```

调用上述 BitmapShader 方法可以产生一个画有一个位图的渲染器（Shader），各个参数的具体说明如下所示。

- bitmap：表示在渲染器内使用的位图。
- tileX：表示在位图上 x 方向花砖模式。
- tileY：表示在位图上 y 方向花砖模式。
- TileMode：表示渲染模式，共有以下 3 种模式。

➢ CLAMP：如果渲染器超出原始边界范围，会复制范围内边缘染色。

➢ REPEAT：横向和纵向的重复渲染器图片，平铺。

➢ MIRROR：横向和纵向的重复渲染器图片，这个和 REPEAT 重复方式不一样，是以镜像方式平铺。

（2）类 LinearGradient：是线性渐变类，用来进行梯度渲染，其核心方法有两个，第一个核心方法如下所示。

```
public LinearGradient(float x0, float y0, float x1, float y1, int[] colors, float[]
positions, Shader.TileMode tile)
```

各个参数的具体说明如下所示。

- x0：渐变起初点坐标 x 位置。
- y0：渐变起初点坐标 y 位置。
- x1：渐变终点坐标 x 位置。
- y1：渐变终点坐标 y 位置。

- colors：渐变颜色数组。
- positions：是一个数组用来指定颜色数组的相对位置，如果为 null 则沿坡度线均匀分布。
- tile：平铺方式。

第二个核心方法如下所示。

```
public LinearGradient(float x0, float y0, float x1, float y1, int color0, int color1,
Shader.TileMode tile)
```

各个参数的具体说明如下所示。

- x0：渐变起初点坐标 x 位置。
- y0：渐变起初点坐标 y 位置。
- x1：渐变终点坐标 x 位置。
- y1：渐变终点坐标 y 位置。
- color0：渐变开始颜色。
- color1：渐变结束颜色。
- tile：平铺方式。

（3）类 RadialGradient：用来实现环形渲染。其核心方法有两个，第一个核心方法如下所示。

```
public RadialGradient(float x, float y, float radius, int[] colors, float[]
positions,Shader.TileMode tile)
```

各个参数的具体说明如下所示。

- float x：圆心 x 坐标。
- float y：圆心 y 坐标。
- float radius：半径。
- int[] colors：渲染颜色数组。
- floate[] positions：相对位置数组，可以为 null。如果为 null，则颜色沿渐变线均匀分布。
- Shader.TileMode tile：渲染器的平铺模式。

第二个核心方法如下所示。

```
public RadialGradient(float x, float y, float radius, int color0, int color1,
Shader.TileMode tile)
```

各个参数的具体说明如下所示。

- float x：圆心 x 坐标。
- float y：圆心 y 坐标。
- float radius：半径。
- int color0：圆心颜色。
- int color1：圆边缘颜色。
- Shader.TileMode tile：渲染器平铺模式。

（4）类 SweepGradient：这是一个扫描渲染类，用来实现梯度渲染，其核心方法有两个，第一个核心方法如下所示。

```
public SweepGradient(float cx, float cy, int[] colors, float[] positions)
```

各个参数的具体说明如下所示。

- cx：渲染中心 x 点坐标。
- cy：渲染中心 y 点坐标。

- colors：围绕中心渲染的颜色数组，至少要有两种颜色值。
- positions：相对位置的颜色数组,可为 null，若为 null，可为 null，颜色沿渐变线均匀分布。

第二个核心方法如下所示。

```
public SweepGradient(float cx, float cy, int color0, int color1)
```

各个参数的具体说明如下所示。

- cx：渲染中心点 x 坐标。
- cy：渲染中心点 y 坐标。
- color0：起始渲染颜色。
- color1：结束渲染颜色。

（5）类 ComposeShader：这是一个混合渲染类，可以和其他几个子类组合起来使用。

在使用 Shader 类时需要先构建 Shader 对象，然后通过 Paint 的 setShader 方法设置对象，接着设置渲染对象，最后在绘制时使用这个 Paint 对象即可。当然在使用不同的渲染时，需要构建不同的对象。

在接下来的内容中，将通过一个具体的演示实例讲解用类 Shader 来渲染不同的图像的方法。

题目	目的	源码路径
实例 13-1	用 Shader 类来渲染不同的图像	\daima\13\ShaderL

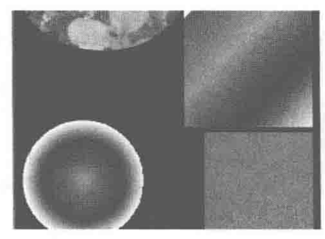
实例文件 ShaderL.java 的主要实现代码如下所示。

```
/* 声明 Bitmap 对象 */
Bitmap mBitQQ = null;
int    BitQQwidth = 0;
int    BitQQheight = 0;
Paint  mPaint = null;
/* Bitmap 渲染 */
Shader mBitmapShader = null;
/* 线性渐变渲染 */
Shader mLinearGradient = null;
/* 混合渲染 */
Shader mComposeShader = null;
/* 唤醒渐变渲染 */
Shader mRadialGradient = null;
/* 梯度渲染 */
Shader mSweepGradient = null;
ShapeDrawable mShapeDrawableQQ = null;
public example5(Context context)
{
    super(context);
    /* 装载资源 */
    mBitQQ = ((BitmapDrawable) getResources().getDrawable(R.drawable.qq)).getBitmap();
    /* 得到图片的宽度和高度 */
    BitQQwidth = mBitQQ.getWidth();
    BitQQheight = mBitQQ.getHeight();
    /* 创建 BitmapShader 对象 */
    mBitmapShader = new BitmapShader(mBitQQ,Shader.TileMode.REPEAT,Shader.TileMode.
    MIRROR);
    /* 创建 LinearGradient 并设置渐变的颜色数组 */
    mLinearGradient = new LinearGradient(0,0,100,100,
    new int[]{Color.RED,Color.GREEN,Color.BLUE,Color.WHITE},
    null,Shader.TileMode.REPEAT);
    /*混合渲染*/
    mComposeShader = new ComposeShader(mBitmapShader,mLinearGradient,PorterDuff.
    Mode.DARKEN);

    /* 构建 RadialGradient 对象，设置半径的属性 */
    //这里将 BitmapShader 和 LinearGradient 进行混合
```

```
        //当然也可以使用其他的组合
        //混合渲染的模式很多，可以根据自己需要来选择
        mRadialGradient = new RadialGradient(50,200,50,
        new int[]{Color.GREEN,Color.RED,Color.BLUE,Color.WHITE},
        null,Shader.TileMode.REPEAT);
        /* 构建 SweepGradient 对象 */
        mSweepGradient = new SweepGradient(30,30,new int[]{Color.GREEN,Color.RED,Color.
        BLUE,Color.WHITE},null);
        mPaint = new Paint();
        /* 开启线程 */
        new Thread(this).start();
    }
    public void onDraw(Canvas canvas)
    {
        super.onDraw(canvas);

        //将图片裁剪为椭圆形
        /* 构建 ShapeDrawable 对象并定义形状为椭圆 */
        mShapeDrawableQQ = new ShapeDrawable(new OvalShape());
        /* 设置要绘制的椭圆形的东西为 ShapeDrawable 图片 */
        mShapeDrawableQQ.getPaint().setShader(mBitmapShader);
        /* 设置显示区域 */
        mShapeDrawableQQ.setBounds(0,0, BitQQwidth, BitQQheight);
        /* 绘制 ShapeDrawableQQ */
        mShapeDrawableQQ.draw(canvas);
        //绘制渐变的矩形
        mPaint.setShader(mLinearGradient);
        canvas.drawRect(BitQQwidth, 0, 320, 156, mPaint);
        //显示混合渲染效果
        mPaint.setShader(mComposeShader);
        canvas.drawRect(0, 300, BitQQwidth, 300+BitQQheight, mPaint);
        //绘制环形渐变
        mPaint.setShader(mRadialGradient);
        canvas.drawCircle(50, 200, 50, mPaint);
        //绘制梯度渐变
        mPaint.setShader(mSweepGradient);
        canvas.drawRect(150, 160, 300, 300, mPaint);
    }
    // 触笔事件
    public boolean onTouchEvent(MotionEvent event)
    {
        return true;
    }
    // 按键按下事件
    public boolean onKeyDown(int keyCode, KeyEvent event)
    {
        return true;
    }
    // 按键弹起事件
    public boolean onKeyUp(int keyCode, KeyEvent event)
    {
        return false;
    }
    public boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event)
    {
        return true;
    }
    /*线程处理*/
    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
        {
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
```

```
            }
            //使用 postInvalidate 可以直接在线程中更新界面
            postInvalidate();
        }
    }
}
```

执行后的效果如图 13-1 所示。



▲图 13-1 用 Shader 类渲染不同图像的执行效果

13.2 使用 SurfaceFlinger 渲染器

Android 采用 SurfaceFlinger 作为系统的渲染管理器。在本书前面的内容中，已经从底层讲解了 Android 系统中 SurfaceFlinger 系统的实现过程。在本节将详细讲解使用 SurfaceFlinger 渲染二维图像的基本应用知识，为读者进入本书后面知识的学习打下基础。

13.2.1 SurfaceFlinger 基础

在 Android 系统，只要是在 UI 屏幕中显示的任何图形图像，都离不开 SurfaceFlinger 渲染器的渲染。甚至于一个简单的开机画面，也需要 SurfaceFlinger 渲染生成。也就是说，展现在用户面前的 Android 屏幕的内容，都是借助于 SurfaceFlinger 实现的。

在 Android 系统的 SurfaceFlinger 渲染器中，提供了以下几个常用的管理对象。

- mClientsMap：管理客户端与服务端的连接。
- Isurface 和 IsurfaceComposer：AIDL 调用接口实例。
- mLayerMap：服务端的 Surface 的管理对象。
- mCurrentState.layersSortedByZ：以 Surface 的 Z-order 序列排列的 Layer 数组。
- graphicPlane：缓冲区输出管理。
- OpenGL ES：图形计算、图像合成等图形库。
- gralloc.xxx.so：跟平台相关的图形缓冲区管理器。
- pmem Device：提供共享内存，在 gralloc.xxx.so 可见，在上层被 gralloc.xxx.so 抽象了。

Surface 渲染框架的功能十分强大，例如可以在 DataGrid 中利用多种组件实现 Column 渲染。下面是使用 Surface 渲染 Column 的 Actionscript 代码。

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:degrafa="com.degrafa.*"
    xmlns:paint="com.degrafa.paint.*"
    xmlns:geometry="com.degrafa.geometry.*"
    layout="absolute"
    creationComplete="generateData()"
    backgroundColor="#FFFFFF"
    backgroundGradientColors="[#999999, #FFFFFF]" viewSourceURL="srcview/index.html">
    <mx:Style>
        ApplicationControlBar {
            highlightAlphas: 0.7, 0.06;
            fillAlphas: 1, 1;
            fillColors: #999999, #666666;
            cornerRadius: 0;
            dropShadowEnabled: true;
            shadowDirection: center;
        }
    </mx:Style>
    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
```

```
            private var dataCollection : ArrayCollection;
            private function generateData() : void
            {
                if ( dataCollection == null )
                    dataCollection = new ArrayCollection();

                dataCollection.disableAutoUpdate();
                dataCollection.removeAll();

                for ( var x : int = 0; x < 100; x ++ )
                {
                    var o : Object = new Object();
                    o.capacity = 500;
                    o.used = Math.random() * o.capacity;
                    o.free = o.capacity - o.used;
                    o.name = "Disk: " + x;
                    dataCollection.addItem( o );
                }
                dataCollection.enableAutoUpdate();
                dataCollection.refresh();
            }
        ]]>
    </mx:Script>
    <mx:ApplicationControlBar dock="true">
        <mx:Button label="Generate New Data" click="generateData()" useHandCursor="true"
buttonMode="true" />
    </mx:ApplicationControlBar>
    <mx:DataGrid top="10" bottom="10" left="10" right="10"
        dataProvider="{ dataCollection }">
        <mx:columns>
            <mx:DataGridColumn headerText="Name" dataField="name"/>
            <mx:DataGridColumn headerText="Capacity" dataField="capacity"/>
            <mx:DataGridColumn headerText="Used" dataField="used"/>
            <mx:DataGridColumn headerText="Used" dataField="used">
                <mx:itemRenderer>
                    <mx:Component>
                        <mx:Canvas>
                            <degrafa:Surface>
                                    <degrafa:fills>
                                     <paint:LinearGradientFill id="redGradient" angle="90">
                                         <paint:GradientStop alpha="1"  color="#FF0000"/>
                                         <paint:GradientStop alpha="1" color="#333333"/>
                                     </paint:LinearGradientFill>
                                </degrafa:fills>

                                <degrafa:GeometryGroup>
                                    <geometry:RoundedRectangle
                                        fill="{redGradient}"
                                        cornerRadius="2"
                                        width="{ width * ( data.used / data.capacity ) }"
                                        height="{ height }" />
                                </degrafa:GeometryGroup>
                                <degrafa:filters>
                                    <mx:DropShadowFilter />
                                </degrafa:filters>
                            </degrafa:Surface>
                        </mx:Canvas>
                    </mx:Component>
                </mx:itemRenderer>
            </mx:DataGridColumn>
            <mx:DataGridColumn headerText="Free" dataField="free"/>
            <mx:DataGridColumn headerText="Free" dataField="free">
                <mx:itemRenderer>
                    <mx:Component>
                        <mx:Canvas>
                            <degrafa:Surface>
                                    <degrafa:fills>
                                     <paint:LinearGradientFill id="greenGradient" angle="90">
                                         <paint:GradientStop alpha="1"  color="#00FF00"/>
```

```
                        <paint:GradientStop alpha="1" color="#333333"/>
                     </paint:LinearGradientFill>
                  </degrafa:fills>
                  <degrafa:GeometryGroup>
                     <geometry:RoundedRectangle
                        fill="{greenGradient}"
                        cornerRadius="2"
                        width="{ width * ( data.free / data.capacity ) }"
                        height="{ height }" />
                  </degrafa:GeometryGroup>
                  <degrafa:filters>
                     <mx:DropShadowFilter />
                  </degrafa:filters>
               </degrafa:Surface>
            </mx:Canvas>
         </mx:Component>
      </mx:itemRenderer>
    </mx:DataGridColumn>
  </mx:columns>
 </mx:DataGrid>
</mx:Application>
```

13.2.2 渲染 Android 手机屏幕中的图形

在 Android 系统的屏幕中通过显示帧的方式来显示图形界面。而在 Android 系统帧缓冲区中分配的图形缓冲区是在 SurfaceFlinger 服务中使用的，在内存中分配的图形缓冲区既可以在 SurfaceFlinger 服务中使用，也可以在其他的应用程序中使用。当其他的应用程序需要使用图形缓冲区的时候，它们就会请求 SurfaceFlinger 服务为它们分配，因此，对于其他的应用程序来说，它们只需要将 SurfaceFlinger 服务返回来的图形缓冲区映射到自己的进程地址空间来使用就可以了，这就是后面我们所要分析的图形缓冲区的注册过程。

在接下来的内容中，将通过一个具体的演示实例讲解在手机屏幕中移动一个不断变换颜色的图形的方法。在本实例中，使用 SurfaceView 类技术，目的是在屏幕中实现一个不断变换颜色的图形，并且通过键盘上的方向按键可以移动这个图形。当我们需要开发一个复杂项目的时候，例如游戏，而且对程序的执行效率要求很高时，View 类就不能满足需求了，这时必须用 SurfaceView 类进行开发。对速度要求很高的游戏，View 类也不能满足需求，必须使用 SurfaceView 类进行开发。而对速度要求很高的游戏，可以使用双缓冲来显示。游戏中的背景、人物、动画等都需要绘制在一个画布（Canvas）上，而 SurfaceView 可以直接访问一个画布，SurfaceView 是提供给需要直接画像素而不是使用窗体部件的应用使用的。每个 Surface 创建一个 Canvas 对象（但属性时常改变），用来管理 View 和 Surface 上的绘图操作。本实例 UI 界面内容的显示过程，就是通过 SurfaceFlinger 服务中分配显示的帧缓冲区实现的。

题目	目的	源码路径
实例 13-2	在手机屏幕中移动一个不断变换颜色的图形	\daima\13\sur

本实例的具体实现流程如下所示。

（1）编写主程序文件，在屏幕中绘制一个图形，并且通过线程来实现颜色的不断闪烁。具体代码如下所示。

```
// 控制循环
boolean mbLoop = false;
// 定义 SurfaceHolder 对象
SurfaceHolder mSurfaceHolder = null;
int miCount = 0;
int y = 50;
public example155(Context context) {
```

```
        super(context);
        // 实例化 SurfaceHolder
        mSurfaceHolder = this.getHolder();
        // 添加回调函数
        // 注意 mSurfaceHolder.addCallback(this)这句执行完了之后
        // 马上就会回调 surfaceCreated 方法，然后开启线程执行绘图方法，这个执行顺序要搞清楚
        mSurfaceHolder.addCallback(this);
        this.setFocusable(true);
        mbLoop = true;
    }
    // 在 surface 的大小发生改变时激发
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width,
            int height) {
    }
    // surface 创建时激发，此方法在主线程总执行
    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        // 开启绘图线程
        new Thread(this).start();
    }
    // 在 surface 销毁时激发
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        // 停止循环
        mbLoop = false;
    }
    // 绘图循环
    @Override
    public void run() {
        while (mbLoop) {
            try {
                Thread.sleep(200);
            } catch (Exception e) {
            }
            // 为什么同步？这就像一块画布，你不能让两个人同时往上边画画
            synchronized (mSurfaceHolder) {
                Draw();
            }
        }
    }
    // 绘图方法，注意这里另起一个线程来执行绘图方法，不是在 UI 线程
    public void Draw() {
        // 锁定画布，用 SurfaceHolder 对象的 lockCanvas 方法得到 canvas
        Canvas canvas = mSurfaceHolder.lockCanvas();
        if (mSurfaceHolder == null || canvas == null) {
            return;
        }
        if (miCount < 100) {
            miCount++;
        } else {
            miCount = 0;
        }
        // 绘图
        Paint mPaint = new Paint();
        // 给 Paint 对象加上抗锯齿标志
        // [img]http://tech.techweb.com.cn/thread-459611-1-1.html[/img]
        // 详细说明可以看看这里
        mPaint.setAntiAlias(true);
        mPaint.setColor(Color.BLACK);
        // 绘制矩形——清屏作用
        canvas.drawRect(0, 0, 320, 480, mPaint);
        switch (miCount % 4) {
        case 0:
            mPaint.setColor(Color.BLUE);
            break;
        case 1:
            mPaint.setColor(Color.GREEN);
            break;
```

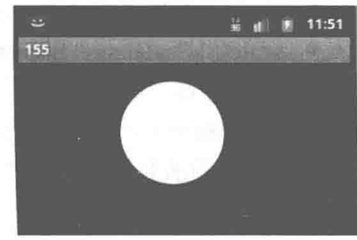
```
        case 2:
            mPaint.setColor(Color.RED);
        case 3:
            mPaint.setColor(Color.YELLOW);
        default:
            mPaint.setColor(Color.WHITE);
            break;
        }
        // 绘制矩形
        canvas.drawCircle((320 - 25) / 2, y, 50, mPaint);
        // 绘制后必须解锁才能显示
        mSurfaceHolder.unlockCanvasAndPost(canvas);
    }
}
```

（2）编写文件 Activity01.java 实现按键响应事件，主要代码如下所示。

```
    // 触笔事件，返回值为 true，父视图不做处理，以下返回值为 true 的都是不做处理的
    public boolean onTouchEvent(MotionEvent event) {
        return true;
    }
    // 按键按下事件
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        return true;
    }
    // 按键弹起事件
    public boolean onKeyUp(int keyCode, KeyEvent event) {
        switch (keyCode) {
        // 上方向键
        case KeyEvent.KEYCODE_DPAD_UP:
            mGameSurfaceView.y -= 3;
            break;
        // 下方向键
        case KeyEvent.KEYCODE_DPAD_DOWN:
            mGameSurfaceView.y += 3;
            break;
        }
        return false;
    }
    public boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event) {
        return true;
    }
}
```

执行后的效果如图 13-2 所示，可以用方向键来移动这个闪烁的图片。

13.3 使用 Skia 渲染引擎



▲图 13-2 移动不断变换颜色的图形的执行效果

在 Android 中的画图过程分为 2D 和 3D 两种，其中 2D 图形是由 Skia 来实现的，2D 图形的渲染功能也是由 Skia 实现的。在本节的内容中，将详细讲解 Skia 的基本知识，为读者进入本书后面知识的学习打下基础。

13.3.1 Skia 基础

Android 系统使用 Skia 作为其核心图形引擎，Google Chrome 的图形引擎也是 Skia。

（1）Skia 对上层的接口（API）。

Skia 的源文件及部分头文件都保存在"external/skia/src"目录下，导出的头文件保存在"external/skia/include"目录下。其中最主要的是 SKCanvas 类，几乎整个 Android GUI 系统的底层绘制都是由这个类来完成的。在类 SKCanvas 中主要具有 3 种绘制功能。

- 基本图形绘制，如 drawARGB 和 drawLine 函数。
- 图像文件绘制，如 drawBitmap 函数。
- 文本绘制，如 drawText 函数。

（2）Android 的图形包（Graphics）。

Android 图形类的包是 android.graphics，它通过调用图形系统的 JNI 提供了对 Java 框架中图形系统的支持。在 Android 的 Java 框架和 Java 应用程序中，2D 绘制的功能（基本图形、图片文件、字）也是通过调用 Graphics 来实现的。在 Android 图形系统中，最重要的一个类便是 android.graphic.canvas。

在 Skia 引擎中，存在如下所示的类。

1. SkCanvas

类 SkCanvas 是 Skia 引擎的一个核心类，在里面封装了所有对设备进行的画图操作。此类自身包含了一个设备的引用，以及一个矩阵和裁剪栈。所有的画图操作，都是在经过栈内存放的矩阵变幻之后才进行的（这点和 OpenGL 类似）。当然，最终显示给用户的图像，还必须经过裁剪堆栈的运算。

类 SkCanvas 记录着整个设备的绘画状态，而设备上面绘制的对象的状态又是由 SkPaint 类来记录的。类 SkPaint 作为参数，传递给不同 SkCanvas 类的成员函数 drawXXXX()，比如 drawPoints、drawLine、drawRect 和 drawCircle。在 SkPaint 类中记录着如颜色（color）、字体（typeface）、文字大小（textSize）、文字粗细（strokeWidth）和渐变（gradients, patterns）等属性。

在类 SkCanvas 中主要包含了如下所示的成员函数。

（1）通过下面的构造函数给定一个 Bitmap 或者 Device，在给定的对象上进行画图，Device 可以为空。

- SkCanvas(const SkBitmap& bitmap)。
- SkCanvas(SkDevice* device = NULL)。
- setViewport 和 getViewport：这 2 个函数只有在支持 OpenGL 视图时才有效。
- Save、saveLayer、saveLayerAlpha 和 restore：这 4 个函数用于保存和恢复显示矩阵，剪切，过滤堆栈，不同函数有不同的附加功能。

（2）下面的函数分别实现了移位、缩放、旋转和变形功能。

- translate(SkiaScalar dx，SkiaScalar dy)。
- scale(SkScalar sx，SkScalar sy)。
- rotate(SkScalar degrees)。
- skew(SkScalar sx，SkScalar sy)。

（3）下面的函数用于指定具体矩阵，进行相应的变换的函数、（2）中的 4 个函数都可以通过定义特定的矩阵，再调用下面的函数实现。

```
cancat(const SkMatrix& matrix);//图像剪辑，把指定的区域显示出来。
clipRect(SkRect&...);
clipPath(SkPath&...);
clipRegion(SkRegion&...);
```

（4）在当前画布内画图时，有以下多种画图方式。

- drawARGB(u8 a, u8 r, u8 g, u8 b....)：给定透明度以及红、绿、蓝 3 色，填充整个可绘制区域。
- drawColor(SkColor color...)：用给定颜色填充整个绘制区域。
- drawPaint(SkPaint& paint)：用指定的画笔填充整个区域。

- drawPoint(...)或 drawPoints(...)：根据各种不同参数绘制不同的点。
- drawLine(x0, y0, x1, y1, paint)：画线，起点($x0, y0$)，终点($x1, y1$)，使用 paint 作为画笔。
- drawRect(rect, paint) ：画矩形，矩形大小由 rect 指定，画笔由 paint 指定。
- drawRectCoords(left, top, right, bottom, paint)：给定 4 个边界画矩阵。
- drawOval(SkRect& oval, SkPaint& paint)：画椭圆，椭圆大小由 oval 矩形指定。
- drawCicle(cx，cy，radius, paint)：给定圆心坐标和半径画圆。
- drawArcSkRect& oval...)：画弧线，用法类似于画椭圆。
- drawRoundRect(rect, rx，ry，paint)：画圆角矩形，x、y 方向的弧度用 rx、ry 指定。
- drawPath(path, paint)：路径绘制，根据 path 指定的路径绘制路径。
- drawBitmap(SkBitmap& bitmap, left, top, paint = NULL)：绘制指定的位图，paint 可以为空。
- drawBitmapRect(bitmap, src, dest, paint=NULL)：绘制给定位图的一部分区域，此区域由 src 指定，然后把截取的部分位图绘制到 dest 指定的区域，可能进行缩放。
- drawBitmapMatrix(bitmap, matrix，paint=NULL)：功效同上，可以通过给定矩阵来进行裁剪和缩放变换。
- drawSprite(bitmap, left, top, paint=NULL)：绘制位图，不受当前变换矩阵影响。
- drawText(void* text, byteLength, x，y，paint) ：以(x，y)为起始点写文字，文字存储在 text 指针内，长度有 byteLength 指定。
- drawPosText(...)：功能同上，不过每个文字可以单独指定位置。
- drawPosTextH(...)：功能同上，不过由一个变量指定了当前所有文字的统一 y 坐标，即在同一条水平线上以不同的间隔写字。
- drawTextOnPathHV 和 drawTextOnPath, drawTextOnPath：以不同方式在给定点的 path 上面绘制文字。
- drawPicture(SkPicture& picture)：在画布上绘制图片，是比较高效的绘图函数。
- drawShape(SkShape*)：在画布上绘制指定形状的图像。
- drawVertices(...)：绘制点，可以有纹理、颜色等附加选项。

2. SkBitmap

类 SkBitmap 是 Skia 中很重要的一个类，很多画图动作涉及 SkBitmap，它封装了与位图相关的一系列操作。类 SkBitmap 是一个位图类，但它的内部有自己的一些逻辑，尤其是使用了参考计数器，可以避免多重拷贝，并且增加了锁机制，位图的空间可以由外部给定。

类 Allocator 是 SkBitmap 的内嵌类，它只有一个成员函数 allocPixelRef()，所以把它理解为一个接口更合适，SkBitmap 使用 Allocator 的派生类——HeapAllocator 作为它的默认分配器。其实现代码如下所示。

```
bool SkBitmap::HeapAllocator::allocPixelRef(SkBitmap* dst,  SkColorTable* ctable) {
    Sk64 size = dst->getSize64();
    if (size.isNeg() || !size.is32()) {
       return false;
    }
    void* addr = sk_malloc_flags(size.get32(), 0);  // returns NULL on failure
    if (NULL == addr) {
       return false;
    }
    dst->setPixelRef(new SkMallocPixelRef(addr, size.get32(), ctable))->unref();
    dst->lockPixels();
    return true;
}
```

我们也可以自己定义一个 Allocator，方法非常简单，只需使用 SkBitmap 的成员函数 allocPixels (Allocator* allocator, SkColorTable* ctable) 传入自定义的 Allocator 即可。如果传入 NULL，则使用默认的 HeapAllocator。

3. 字体类 SkTypeface 和 SkFontHost

Android 的字体由 Android 2D 图形引擎 Skia 实现，并在 Zygote 的 Preloading classes 中对系统字体进行载入。在 Skia 中是通过类 SkTypeface 和 SkFontHost 实现字体功能的。

13.3.2 使用 Skia 绘图

在 Android 多媒体开发应用中，使用 Skia API 进行图形绘制时会用到如下所示的类。

- SkBitmap：用来设置像素。
- SkCanvas：用来写入位图。
- SkPaint：用来设置颜色和样式。
- SkRect：用来绘制矩形。

在 Android 多媒体开发应用中，使用 Skia 进行绘图的基本步骤如下所示。

（1）定义一个位图 32 位像素并初始化，例如下面的代码。

```
SkBitmap bitmap;
bitmap.setConfig(SkBitmap::kARGB_8888_Config, 200, 200);
```

其中 setConfig 为设置位图的格式，其原型如下所示。

```
void setConfig(Config, int width, int height, int rowBytes = 0)
```

其中 Config 为一个数据结构，其格式如下所示。

```
enum Config {
kNo_Config, // 不确定的位图格式
kA1_Config, //1 位 ( 黑 , 白 ) 位图
kA8_Config, //8 位 ( 黑 , 白 ) 位图
kIndex8_Config, // 类似 windows 下的颜色索引表，具体请查看 SkColorTable 类结构
kRGB_565_Config, //16 位象素 565 格式位图，详情请查看 SkColorPriv.h 文件
kARGB_4444_Config, //16 位象素 4444 格式位图，详情请查看 SkColorPriv.h 文件
kARGB_8888_Config, //32 位象素 8888 格式位图，详情请查看 SkColorPriv.h 文件
kRLE_Index8_Config,
kConfigCount
};
```

（2）然后分配位图所占的空间，例如下面的代码。

```
bitmap.allocPixels()
```

其实 allocPixels 是一个重载函数，其原型如下所示。

```
bool allocPixels(SkColorTable* ctable = NULL)
```

其中参数 ctable 表示颜色索引表，在一般情况下为 NULL。

（3）指定输出设备，例如下面的代码。

```
SkCanvas canvas(new SkDevice(bitmap));
```

其中 canvas 为一个多构造函数，其原型如下所示。

```
explicit SkCanvas(const SkBitmap& bitmap) ,
explicit SkCanvas(SkDevice* device = NULL)
```

其中关键字 explicit 含有“不允许类型转换”之意，此处输出设备既可以是一个上下文 Device，也可以指定为一张位图。

（4）设置设备绘制的风格，例如下面的代码。

```
Paint paint;
SkRect r;
paint.setARGB(255, 255, 0, 0);
r.set(25, 25, 145, 145);
canvas.drawRect(r, paint);
```

（5）再次通过 paint 可以指定绘图的颜色、文本的大小、对齐方式和编码格式等，因为以前位图的格式设置为 kARGB_8888_Config，所以这里要设置绘制的颜色 setARGB(255, 255, 0, 0)。其中第一位参数表示透明颜色通道，其他 3 位分别为 R、G、B。r 表示设置要绘制的范围。

（6）最后通过 drawRect 绘制出指定区域的一个方形。

到此为止，一个红色的矩形就绘制成功了。

在接下来的内容中，将通过一个具体的演示实例讲解使用 Skia 在屏幕中绘图的方法。

题目	目的	源码路径
实例 13-3	在 Android 系统中使用 Skia 绘图	\daima\13\Skia

本实例的具体实现流程如下所示。

（1）新建一个工程，构建 Skia 绘图的外壳部分，具体实现代码如下所示。

```
public class SkiaView extends View {

    /** TAG 标识 */
    private static final String TAG = "SkiaView";

    /** 载入动态库 */
    static {
        try {
            System.loadLibrary("SkiaJni");
        } catch(UnsatisfiedLinkError e) {
            Log.e(TAG, "Couldn't load native libs");
            e.printStackTrace();
        }
    }

    public SkiaView(Context context) {
        super(context);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        Log.i(TAG, "==draw start==");
        // 调用本地方法
        native_renderCanvas(canvas);
        Log.i(TAG, "==draw end==");
    }

    /** 本地渲染画布方法 */
    private native void native_renderCanvas(Canvas canvas);

}
```

（2）开始进行 C/C++封装工作，准备好 Android 的源码，将创建的 Skia 工程保存到 JNI 目录，然后执行如下命令。

```
bash --login -c "cd $WORKSPACE/AndroidSkia && $NDKROOT/ndk-build"
```

其中$WORKSPACE 和$NDKROOT 分别表示工作空间和 NDK 路径，这些可以在 Cygwin 根目录\home\[your name]\ .bash_profile 文件内配置。

（3）使用 includes 命令包含 JNI 和 Skia 等需要的头文件，文件列表如图 13-3 所示。

```
\dalvik\libnativehelper\include\nativehelper
\system\core\include
\frameworks\base\include
\frameworks\base\native\include
\frameworks\base\core\jni\android\graphics
\external\skia\include\core
\external\skia\include\config
\external\skia\include\images
\prebuilt\ndk\android-ndk-r4\platforms\android-8\arch-arm\usr\include
```

▲图 13-3 需要使用 includes 命令包含的文件

（4）准备 Android.mk 文件，具体内容如下所示。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

MY_ANDROID_SOURCE =此处是电脑中 Android 源码的根目录
MY_ANDROID_SYSLIB = $(MY_ANDROID_SOURCE)/out/target/product/generic/system/lib
#也可以在 cygwin\home\Join\.bash_profile 文件内配置，如下：
#   export MY_ANDROID_SOURCE="/cygdrive/f/..."
#   export MY_ANDROID_SYSLIB="/cygdrive/f/..."

LOCAL_MODULE := libSkiaJni
LOCAL_SRC_FILES := \
    jniLoad.cpp \
    org_join_skia_SkiaView.cpp

LOCAL_C_INCLUDES := \
    $(MY_ANDROID_SOURCE)/dalvik/libnativehelper/include/nativehelper \
    $(MY_ANDROID_SOURCE)/frameworks/base/include \
    $(MY_ANDROID_SOURCE)/system/core/include \
    $(MY_ANDROID_SOURCE)/frameworks/base/native/include \
    $(MY_ANDROID_SOURCE)/frameworks/base/core/jni/android/graphics \
    $(MY_ANDROID_SOURCE)/external/skia/include/core \
    $(MY_ANDROID_SOURCE)/external/skia/include/config \
    $(MY_ANDROID_SOURCE)/external/skia/include/images
#同时在工程 Properties->C/C++ General->Paths and Symbols 属性内包含相应文件目录
#否则编程时找不到.h 文件，不便写代码，但不会影响编译。而 LOCAL_LDLIBS/LOCAL_LDFLAGS 必须添加

#LOCAL_LDLIBS := -L$(MY_ANDROID_SYSLIB) -llog -ljnigraphics  -lskia -landroid_runtime
#也可以用以下方式指定 so 库
LOCAL_LDFLAGS := \
    $(MY_ANDROID_SYSLIB)/liblog.so \
    $(MY_ANDROID_SYSLIB)/libjnigraphics.so \
    $(MY_ANDROID_SYSLIB)/libskia.so \
    $(MY_ANDROID_SYSLIB)/libskiagl.so \
    $(MY_ANDROID_SYSLIB)/libandroid_runtime.so

include $(BUILD_SHARED_LIBRARY)
```

在上述代码中，LOCAL_C_INCLUDES 的头文件路径，第一个是 JNI 的，最后三个是 Skia 的，倒数第四个是 ljnigraphics 的。

（5）准备文件 org_join_skia_SkiaView.cpp，具体实现代码如下所示。

```
#include "jniLoad.h"

#include <GraphicsJNI.h>
#include <SkCanvas.h>
#include <SkPaint.h>
```

```
#include <SkRect.h>
#include <SkColor.h>
#include <SkTypes.h>
#include <SkGraphics.h>

static void drawFlag(SkCanvas* canv);

static void native_renderCanvas(JNIEnv* env, jobject obj, jobject canvas) {
    MY_LOGI("==c method start==");

    SkCanvas* canv = GraphicsJNI::getNativeCanvas(env, canvas);
    if (!canv) {
        MY_LOGE("==canv is NULL==");
        return;
    }

    canv->save();
    canv->translate(100, 100);
    drawFlag(canv);
    canv->restore();

    MY_LOGI("==c method end==");
}

/** 画旗帜 */
static void drawFlag(SkCanvas* canv) {
    SkPaint* paint = new SkPaint();
    paint->setFlags(paint->kAntiAlias_Flag);

    SkRect* rect = new SkRect();
    rect->set(0, 0, 200, 100);
    paint->setColor(SK_ColorRED);
    canv->drawRect(*rect, *paint);

    paint->setColor(SK_ColorGRAY);
    paint->setStrokeWidth(10);
    canv->drawLine(5, 100, 5, 300, *paint);

    paint->setTextSize(30);
    paint->setColor(SK_ColorBLUE);
    paint->setTextAlign(paint->kCenter_Align);
    const char* text = "Hello World";
    canv->drawText(text, strlen(text), 100, 60, *paint);
}

/**
 * JNI registration.
 */
static JNINativeMethod methods[] = { { "native_renderCanvas",
      "(Landroid/graphics/Canvas;)V", (void*) native_renderCanvas } };

int register_org_join_skia_SkiaView(JNIEnv *env) {
    return jniRegisterNativeMethods(env, "org/join/skia/SkiaView", methods,
           sizeof(methods) / sizeof(methods[0]));
}
```

（6）准备文件 jniLoad.h，具体实现代码如下所示。

```
#ifndef JNILOAD_H_
#define JNILOAD_H_

#include <jni.h>
#include <utils/Log.h>

#define MY_LOG_TAG "JNI_LOG"
#define MY_LOGI(...) __android_log_print(ANDROID_LOG_INFO, MY_LOG_TAG, __VA_ARGS__)
#define MY_LOGE(...) __android_log_print(ANDROID_LOG_ERROR, MY_LOG_TAG, __VA_ARGS__)

#ifdef __cplusplus
extern "C" {
#endif
```

```
int jniRegisterNativeMethods(JNIEnv* env, const char* className,
        const JNINativeMethod* gMethods, int numMethods);

#ifdef __cplusplus
}
#endif

#endif /* JNILOAD_H_ */
```

（7）准备文件 jniLoad.cpp，具体实现代码如下所示。

```
#include "jniLoad.h"

#include <stdlib.h>

int register_org_join_skia_SkiaView(JNIEnv *env);

int jniRegisterNativeMethods(JNIEnv* env, const char* className,
        const JNINativeMethod* gMethods, int numMethods) {

    jclass clazz;
    MY_LOGI("Registering %s natives\n", className);
    clazz = env->FindClass(className);

    if (clazz == NULL) {
        MY_LOGE("Native registration unable to find class '%s'\n", className);
        return JNI_ERR;
    }
    if (env->RegisterNatives(clazz, gMethods, numMethods) < 0) {
        MY_LOGE("RegisterNatives failed for '%s'\n", className);
        return JNI_ERR;
    }
    return JNI_OK;
}

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

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        MY_LOGE("GetEnv failed!");
        return result;
    }

    MY_LOGI("loading . . .");

    if(register_org_join_skia_SkiaView(env) != JNI_OK) {
        MY_LOGE("can't load org_join_skia_SkiaView");
        goto end;
    }
    /**
     * register others
     */

    MY_LOGI("loaded");

    result = JNI_VERSION_1_4;
end:
    return result;
}
```

（8）最后使用 Java 编写测试文件 AndroidSkiaActivity.java，具体实现代码如下所示。

```
public class AndroidSkiaActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        SkiaView view = new SkiaView(this);
        setContentView(view);
```

```
    }
}
```

此时就可以在模拟器中运行测试，执行后的效果如图 13-4 所示。

▲图 13-4 使用 Skia 绘图的执行效果

注意

由此可以总结出，SkCanvas 主要可以完成如下 3 种绘制功能。

- 基本图形绘制，如函数 drawARG 和函数 drawLine。
- 图像文件绘制，如函数 drawBitmap 函数。
- 文本绘制，如函数 drawText 函数。

在 SkCanvas 中和绘制功能相关的 API 接口如下所示。

```
canvas.drawRect(rect, paint);
        canvas.drawOval(oval, paint);
        canvas.drawCircle(x, y, radius, paint);
        canvas.drawRoundRect(rect, rx, ry, paint);
        canvas.drawPath(path, paint);
        canvas.drawBitmap(bitmap, x, y, &paint);
        canvas.drawBitmapRect(bitmap, &srcRect, dstRect, &paint);
        canvas.drawBitmapMatrix(bitmap, matrix, &paint);
        canvas.drawText(text, length, x, y, paint);
        canvas.drawPosText(text, length, pos[], paint);
        canvas.drawTextOnPath(text, length, path, paint);
```

例如可以使用下面的代码实现分别绘制点、线、圆和文字的功能。

```
#include "SkBitmap.h"
#include "SkDevice.h"
#include "SkPaint.h"
#include "SkRect.h"
#include "SkImageEncoder.h"
#include "SkTypeface.h"
using namespace std;
int main(){
SkBitmap bitmap;
bitmap.setConfig(SkBitmap::kARGB_8888_Config,320,240);
bitmap.allocPixels();
SkCanvas canvas(new SkDevice(bitmap));
SkPaint paint;
paint.setARGB(255, 255, 0, 0);
paint.setStrokeWidth(4);
canvas.drawPoint(40,30, paint);
canvas.drawPoint(80,60, paint);
canvas.drawPoint(120,90, paint);
//绿线
paint.setARGB(255, 0, 255, 0);
paint.setStrokeWidth(4);

canvas.drawLine(160,10,320,110,paint);
//蓝色原型
```

```
paint.setARGB(255, 0, 0, 255);
canvas.drawCircle(80,180,50,paint);
//红色文字
SkTypeface *font = SkTypeface::CreateFromFile("simkai.ttf");
if ( font )
{
paint.setARGB(255, 255, 0, 0);
paint.setTypeface( font );
paint.setTextSize(24);
canvas.drawText("HELLO!:)", 8, 200, 180, paint);
}
SkImageEncoder::EncodeFile("snapshot.png", bitmap,SkImageEncoder::kPNG_Type,100);
return 0;
}
```

可以通过下面的代码实现对“.png”格式图片的编解码工作。

```
#include "SkBitmap.h"
#include "SkDevice.h"
#include "SkPaint.h"
#include "SkRect.h"
#include "SkImageEncoder.h"
#include "SkImageDecoder.h"
#include <iostream>
using namespace std;
int main(){
int ret = -1;
SkBitmap bitmap;
//解码
ret = SkImageDecoder::DecodeFile("./old.png", &bitmap);
cout<< "get the decode type = "<< bitmap.config() << endl;
//编码
ret = SkImageEncoder::EncodeFile("new1.png",bitmap,SkImageEncoder::kPNG_Type,100);
cout<< "encode data to png result = "<< ret<< endl;
return 0;
}
```

通过如下代码将 png 格式转换成位图格式，并将数据放到 bitmap 变量中。

```
SkImageDecoder::DecodeFile("./old.png", &bitmap);
```

再看下面的代码。

```
SkImageEncoder::EncodeFile("snapshot.png", bitmap,SkImageEncoder::kPNG_Type,/* Quality
ranges from 0..100 */ 100);
```

通过上述代码可以将 bitmap 中的数据编码输出为“.png”格式，第一位参数为 png 文件路径，第二位为指定的输出位图，第三位为文件的类型，第四位参数指定了输出位图的质量，范围为 0～100，默认是 80。

可以使用下面的代码实现对 JPEG 图片的解码操作。

```
SkStream.h,
SkImageDecoder.h
SkBitmap.h
.................................
SkBitmap bp;
SkImageDecoder::Format fmt;
property_set("hw.jpeg.path", "/data/test.jpg");//处理的 JPEG 图片
    char propBuf[PROPERTY_VALUE_MAX];
    property_get("hw.jpeg.path ", propBuf, "");
    LOGI("property_get: %s.", propBuf);
//将 propBuf 中存储的 JPEG 图片解码成 rgb565 文件
Bool result = SkImageDecoder::DecodeFile(propBuf,
 &bp,SkBitmap::kRGB_565_Config, SkImageDecoder::kDecodePixels_Mode, &fmt);
    if(!result){
        LOGI("decoder file fail!");
```

```
    }else{
        if(fmt!= SkImageDecoder::kJPEG_Format){
            LOGI("decoder file not jpeg!");
        }
        else{
LOGI("width %d,height %d,rowBytesAsPixels %d,config %d,
bytesPerPixel %d",bp.width(),bp.height(),bp.rowBytesAsPixels(),bp.config(),
bp.bytesPerPixel());
        FILE *f_rgb=fopen("/data/test_rgb565.raw","wb");
        short *pixl = (short *) bp.getPixels();
        for(int j=0;j<bp.height();j++){
            fwrite(pixl,1,bp.width()*bp.bytesPerPixel(),f_rgb);
            pixl += bp.rowBytesAsPixels();
        }
        fclose(f_rgb);
        }
}
```

对上述代码的具体说明如下。

- fmt：获取的图片的格式；
- bp.width()：图片的宽度；
- bp.height()：图片的高度；
- bp.rowBytesAsPixels()：一行的 pixel 数，有 pitch 问题；
- bp.config()：图片被解码后的文件格式；
- bp.bytesPerPixel()：每个 pixel 占用的 byte 数。

13.4 通过 Skia 绘制文字

在本章前面的内容中，讲解了 Skia 引擎的基本知识。本节的内容将通过 Skia 在屏幕中绘制“hello Skia”的实现过程来分析其使用原理。

背景介绍

虽然在 Android 的 NDK 文档中，Google 不推荐使用 Native C/C++来开发应用程序。但在实际开发中存在不得不使用 Native C/C++的情况，比如跨平台软件开发。在开发手机软件时，为了能够面向更广泛的用户群，避免不了要同时开发 Symbian、Windows mobile、Android、iPhone 等主流手机平台版本。一般手机软件采用 C/C++开发，所以为了避免重起炉灶，影响效率，在 Android 下一般选择使用 Native C/C++开发核心组件，再加上一个 Java 语言编写的外壳。

因为 Android 系统采用 Skia 作为其核心图形引擎，所以 Android 也全面支持 Skia 绘图功能，但是问题在于 Java 层如何和 JNI 层的代码相互访问。其实在 Android 中有很多绘图类（android.graphics 包下的类）都是对 Skia C++类的一个封装，例如 Canvas <-> SkCanvas、Paint <-> SkPaint，问题的关键是如何在它们之间架起一座桥梁。

我们首先看在 Java 端对类的定义代码。

```
public class SkiaView extends View
{
    private static final String TAG = "skiademo";
    static
    {
        System.loadLibrary("SkiaDemo");
    }
     public native void renderHello(Canvas canvas);

    public SkiaView(Context context)
    {
```

```
        super(context);
    }
    @Override
    public void onDraw(Canvas canvas)
    {
        super.onDraw(canvas);

        Log.d(TAG, "onDraw");
        renderHello(canvas);
    }
}
```

由此可见，类 SkiaView 继承自类 View，并定义了一个 Native（本地）函数 renderHello，其参数是 Canvas。JNI 端的代码如下所示。

```
void Java_com_whtr_example_skiademo_SkiaView_renderHello(JNIEnv *env, jobject thizz,
jobject canvas)
{
    SkCanvas* canv = GraphicsJNI::getNativeCanvas(env, canvas);
    if (!canv)
    {
        return;
    }
    SkPaint paint;
    paint.setColor(SK_ColorRED);
    canv->drawText("hello skia", 10, 20, 20, paint);
}
```

在函数的开始，调用函数 GraphicsJNI::getNativeCanvas 来处理 Java 端传递过来的 Canvas 对象，获取 SkCanvas 对象指针。这样有了 SkCanvas，我们就可以在上面进行绘制了。

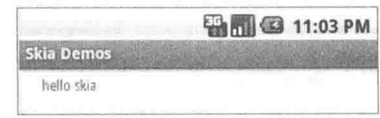
需要注意在 NDK 中并没有包含 Skia 相关头文件和库函数，因此在编译此程序时需要下载 Android 源代码并进行编译。但是在 Android 系统中已经有这些库文件，所以部署到手机上运行是没有问题的。Android.mk 文件的主要内容如下所示。

```
    LOCAL_PATH := $(call my-dir)
    include $(CLEAR_VARS)
    MY_ANDROID_SOURCE=$(HOME)/android/source/cupcake
    LOCAL_MODULE := libSkiaDemo
    LOCAL_CPP_EXTENSION := .cpp
    LOCAL_CXXFLAGS :=
    LOCAL_C_INCLUDES:=$(MY_ANDROID_SOURCE)/frameworks/base/core/jni/android/graphics \
                                     $(MY_ANDROID_SOURCE)/external/skia/include/core \
                                     $(MY_ANDROID_SOURCE)/external/skia/include/images\
                                     $(MY_ANDROID_SOURCE)/frameworks/base/include \
                                     $(MY_ANDROID_SOURCE)/system/core/include
    LOCAL_SRC_FILES := SkiaDemoJni.cpp
    LOCAL_LDLIBS := -llog -lsgl -landroid_runtime \-L$(MY_ANDROID_SOURCE)/out/target/
product/generic/ system/lib/
    include $(BUILD_SHARED_LIBRARY)
```

如果要想获取完整的开源代码和 Android.mk 文件，读者可以使用如下命令获取。

```
svn checkout http://androidcodes.googlecode.com/svn/trunk/androidcodes
```

执行后的效果如图 13-5 所示。



▲图 13-5 通过 Skia 绘制文字的执行效果

第 14 章　开发音频应用程序

在多媒体领域中，音频永远是最主流的应用之一。在本书前面的内容中，已经讲解了 Android 底层音频系统的基本知识。在顶层的 Java 应用中，可以通过底层提供的接口开发常见的音频应用。在本章的内容中，将详细讲解 Android 音频的开发应用基本知识，为读者进入后面知识的学习打下基础。

14.1 音频应用接口类介绍

Android 系统顶层的音频应用功能是通过专用接口实现的，在 Android 中，根据不同的场景，开发者会选择用不同的接口来播放音频资源。Android 提供了专门的接口类来实现音频应用功能，具体说明如下所示。

- 音乐类型的音频资源：通过 MediaPlayer 来播放。
- 音调：通过 ToneGenerator 来播放。
- 提示音：通过 Ringtone 来播放。
- 游戏中的音频资源：通过 SoundPool 来播放。
- 录音功能：通过 MediaRecorder 和 AudioRecord 等来记录音频。

除了上述音频处理类之外，Android 也提供了相关的类来处理音量调节和音频设备的管理等功能，具体说明如下所示。

- AudioManager：通过音频服务，为上层提供了音量和铃声模式控制的接口，铃声模式控制包括扬声器、耳机、蓝牙等是否打开，麦克风是否静音等。在开发多媒体应用时会经常用到 AudioManager。
- AudioSystem：提供了定义音频系统的基本类型和基本操作的接口，对应的 JNI 接口文件为 android_media_AudioSystem.cpp。在 Android 音频系统中主要包括如下所示的音频类型。
  - STREAM_VOICE_CALL；
  - STREAM_SYSTEM；
  - STREAM_ RING；
  - STREAM_MUSIC；
  - STREAM_ALARM；
  - STREAM_NOTIFICATION；
  - STREAM_BLUETOOTH_SCO；
  - STREAM_SYSTEM_ENFORCED；
  - STREAM_DTMF；
  - STREAM_TTS。
- AudioTrack：直接为 PCM 数据提供支持，对应的 JNI 接口文件为 android_media_AudioTrack.cpp。

- AudioRecord：这是音频系统的录音接口，默认的编码格式为 PCM_16_BIT，对应的 JNI 接口文件为 android media_AudioRecord.cpp。
- Ringtone 和 RingtoneManager：为铃声、提示音、闹钟等提供了快速播放以及管理的接口，实质是对媒体播放器提供了一个简单的封装。
- ToneGenerator：提供了对 DTMF 音（ITU-T Q.23）、呼叫监督音（3GPP TS 22.001）、专用音（3GPP TS 31.111）中规定的音频的支持，根据呼叫状态和漫游状态，该文件产生的音频路径为下行音频或者传输给扬声器或耳机。对应的 JNI 接口文件为 android_media_ToneGenerator.cpp，其中 DTMF 音为 WAV 格式，相关的音频类型定义位于文件 ToneGenerator.h 中。
- SoundPool：能够播放音频流的组合音，主要被应用在游戏领域。对应的 JNI 接口文件为 android_media_SoundPool.cpp。
- SoundPool：可以从 APK 包中的资源文件或者文件系统中的文件将音频资源加载到内存中。在底层的实现上，SoundPool 通过媒体播放服务可以将音频资源解码为一个 16bit 的单声道或者立体声的 PCM 流，这使得应用避免了在回放过程中进行解码造成的延迟。除了回放过程中延迟少这一优点外，SoundPool 还能够同时播放一定数量的音频流。当要播放的音频流数量超过 SoundPool 所设定的最大值时，SoundPool 将会停止已播放的一条低优先级的音频流。通过设置 SoundPool 最大播放音频流的数量，可以避免 CPU 过载和影响 UI 体验。
- android.media.audiofx 包：这是从 Android 2.3 开始新增的包，提供了对单曲和全局的音效的支持，包括重低音、环绕音、均衡器、混响和可视化等声音特效。

14.2 AudioManager 类

类 AudioManager 是 Android 系统中最常用的音量和铃声控制接口类。在本节的内容中，将详细介绍类 AudioManager 的基本知识，并通过对应的演示实例来讲解其使用方法，为读者进入本书后面知识的学习打下基础。

14.2.1 AudioManager 基础

类 AudioManager 位于 android.Media 包中，该类提供访问控制音量和铃声模式的操作。

1. 方法

在类 AudioManager 中是通过方法实现音频功能的，其中最为常用的方法如下所示。

- 方法：adjustVolume(int direction, int flags)。

解释：这个方法用来控制手机音量大小，当传入的第一个参数为 AudioManager.ADJUST_LOWER 时，可将音量调小 1 个单位；当传入 AudioManager.ADJUST_RAISE 时，则可以将音量调大 1 个单位。

- 方法：getMode()。

解释：返回当前音频模式。

- 方法：getRingerMode()。

解释：返回当前的铃声模式。

- 方法：getStreamVolume(int streamType)。

解释：取得当前手机的音量，最大值为 7，最小值为 0，当为 0 时，手机自动将模式调整为“振动模式”。

- 方法：setRingerMode(int ringerMode)。

解释：改变铃声模式。

2. 声音模式

Android 手机都有声音模式，例如声音、静音、振动、振动加声音兼备，这些都是手机的基本功能。在 Android 手机中，可以通过 Android SDK 提供的声音管理接口来管理手机声音模式以及调整声音大小，此功能通过类 AudioManager 来实现。

（1）设置声音模式，例如下面的演示代码。

```
//声音模式
AudioManager.setRingerMode(AudioManager.RINGER_MODE_NORMAL);
//静音模式
AudioManager.setRingerMode(AudioManager.RINGER_MODE_SILENT);
//振动模式
AudioManager.setRingerMode(AudioManager.RINGER_MODE_VIBRATE);
```

（2）调整声音大小，例如下面的演示代码。

```
//减少声音音量
AudioManager.adjustVolume(AudioManager.ADJUST_LOWER,  0);
//调大声音音量
AudioManager.adjustVolume(AudioManager.ADJUST_RAISE, 0);
```

3. 基本应用

AudioManager 类的常见应用如下所示。

（1）实现音量控制，例如下面的演示代码。

```
//音量控制,初始化定义
AudioManager mAudioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
//最大音量
int maxVolume = mAudioManager.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
//当前音量
int currentVolume = mAudioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
```

（2）控制音量大小，例如下面的演示代码。

```
if(isSilent){
     mAudioManager.setStreamVolume(AudioManager.STREAM_MUSIC, 0, 0);
   }else{
     mAudioManager.setStreamVolume(AudioManager.STREAM_MUSIC, tempVolume, 0);
     //tempVolume:音量绝对值
   }
```

（3）以步长控制音量的增减，并弹出系统默认音量控制条。例如下面的演示代码。

```
view sourceprint?
   //降低音量，调出系统音量控制
   if(flag == 0){
     mAudioManager.adjustStreamVolume(AudioManager.STREAM_MUSIC,AudioManager.
ADJUST_LOWER, AudioManager.FX_FOCUS_NAVIGATION_UP);
   }
   //增加音量，调出系统音量控制
   else if(flag == 1){
     mAudioManager.adjustStreamVolume(AudioManager.STREAM_MUSIC,AudioManager.
ADJUST_RAISE, AudioManager.FX_FOCUS_NAVIGATION_UP);
   }
```

4. 调节声音的基本步骤

在 Android 系统中，使用类 AudioManager 调节声音的基本步骤如下所示。

（1）通过系统服务获得声音管理器，例如下面的演示代码。

```
AudioManager audioManager = (AudioManager)getSystemService(Service.AUDIO_SERVICE);
```

（2）根据实际需要调用适当的方法，例如下面的演示代码。

```
audioManager.adjustStreamVolume(int streamType, int direction, int flags);
```

各个参数的具体说明如下所示。

- streamType：声音类型，可取下面的值。
  - STREAM_VOICE_CALL：打电话时的声音。
  - STREAM_SYSTEM：Android 系统声音。
  - STREAM_RING：电话铃响。
  - STREAM_MUSIC：音乐声音。
  - STREAM_ALARM：警告声音。
- direction：调整音量的方向，可取下面的值。
  - ADJUST_LOWER：调低音量。
  - ADJUST_RAISE：调高音量。
  - ADJUST_SAME：保持先前音量。
- flags：可选标志位。

（3）设置指定声音类型，例如下面的演示代码。

```
audioManager.setStreamMute(int streamType, boolean state)
```

通过上述方法设置指定声音类型（streamType）是否为静音。如果 state 为 true，则设置为静音；否则，不设置为静音。

（4）设置铃音模式，例如下面的演示代码。

```
audioManager.setRingerMode(int ringerMode);
```

通过上述方法设置铃音模式，可取的值如下所示。

- RINGER_MODE_NORMAL：铃音正常模式。
- RINGER_MODE_SILENT：铃音静音模式。
- RINGER_MODE_VIBRATE：铃音振动模式，即铃音为静音，启动振动。

（5）设置声音模式，例如下面的演示代码。

```
audioManager.setMode(int mode);
```

通过上述方法设置声音模式，可取的值如下所示。

- MODE_NORMAL：正常模式，即在没有铃音与电话的情况。
- MODE_RINGTONE：铃响模式。
- MODE_IN_CALL：接通电话模式。
- MODE_IN_COMMUNICATION：通话模式。

注意：声音的调节是没有权限要求的。

14.2.2 AudioManager 基本应用——设置短信提示铃声

在接下来的内容中，将通过一个具体演示实例讲解设置短信提示铃声的方法。

题目	目的	源码路径
实例 14-1	设置短信提示铃声	\daima\14\DotaBell

本实例的具体实现流程如下所示。

（1）编写文件 main.xml，在程序界面上放置 3 个按钮，分别用于启用、停止和设置间隔时间。主要代码如下所示。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:gravity="center"
>
   <Button
      android:id="@+id/startButton"
      android:text="@string/startButton"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content" />
   <Button
      android:id="@+id/endButton"
      android:text="@string/endButton"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content" />
   <Button
      android:id="@+id/configButton"
      android:text="@string/configButton"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content" />
</LinearLayout>
```

（2）编写文件 BellService.java，开启一个 Service 监听短信的事件，在短信到达后进行声音播放的处理，牵涉到的主要是 Service、Broadcast、MediaPlayer，还有为了设置间隔时间用了最简单的 Preference。在此包含了存放铃声的 Map 和播放铃声等逻辑处理，通过 AudioManager 暂时打开多媒体声音，播放完再关闭。文件 BellService.java 的主要代码如下所示。

```
public class BellService extends Service {
   //监听事件
   public static final String SMS_RECEIVED_ACTION = "android.provider.Telephony.SMS_
RECEIVED";
   //铃声序列
   public static final int ONE_SMS = 1;
   public static final int TWO_SMS = 2;
   public static final int THREE_SMS = 3;
   public static final int FOUR_SMS = 4;
   public static final int FIVE_SMS = 5;

   private HashMap<Integer,Integer> bellMap;//铃声 Map
   private Date lastSMSTime;//上条短信时间
   private int currentBell;//当前应当播放铃声
  //是否第一次启动，避免首次启动马上收到短信而导致立即播放第二条铃声的情况
   private boolean justStart=true;
   private AudioManager am;
   private int currentMediaStatus;
   private int currentMediaMax;

   public IBinder onBind(Intent intent) {
      return null;
   }
   @Override
   public void onCreate() {
      super.onCreate();
      IntentFilter filter = new IntentFilter();
      filter.addAction(SMS_RECEIVED_ACTION);
      Log.e("COOKIE", "Service start");
```

```
        //注册监听
        registerReceiver(messageReceiver, filter);
        // 初始化 Map，根据之后改进可以替换其中的铃声
        bellMap = new HashMap<Integer,Integer>();
        bellMap.put(ONE_SMS, R.raw.holyshit);
        bellMap.put(TWO_SMS, R.raw.holydouble);
        bellMap.put(THREE_SMS, R.raw.holytriple);
        bellMap.put(FOUR_SMS, R.raw.holyultra);
        bellMap.put(FIVE_SMS, R.raw.holyrampage);
        //当前时间
        lastSMSTime=new Date(System.currentTimeMillis());
        //当前应当播放的铃声，初始为 1
        //之后根据间隔判断，若为 5 分钟之内，则+1
        //若举例上一次超过 5 分钟，则重新置为 1
        currentBell=1;
    }

    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        //取消监听
        unregisterReceiver(messageReceiver);
        Log.e("COOKIE", "Service end");
    }
    // 设定广播
    private BroadcastReceiver messageReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            if (action.equals(SMS_RECEIVED_ACTION)) {
                playBell(context, 0);
            }
        }
    };
    //播放音效
    private void playBell(Context context, int num) {
        //为防止用户当前模式关闭了 media 音效，先将 media 打开
        am=(AudioManager)getSystemService(Context.AUDIO_SERVICE);//获取音量控制
        currentMediaStatus=am.getStreamVolume(AudioManager.STREAM_MUSIC);
        currentMediaMax=am.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
        am.setStreamVolume(AudioManager.STREAM_MUSIC, currentMediaMax, 0);
        //创建 MediaPlayer 进行播放
        MediaPlayer mp = MediaPlayer.create(context, getBellResource());
        mp.setOnCompletionListener(new musicCompletionListener());
        mp.start();
    }

    private class musicCompletionListener implements OnCompletionListener {
        @Override
        public void onCompletion(MediaPlayer mp) {
            //播放结束释放 mp 资源
            mp.release();
            //恢复用户之前的 media 模式
            am.setStreamVolume(AudioManager.STREAM_MUSIC, currentMediaStatus, 0);
        }
    }
    //获取当前应该播放的铃声
    private int getBellResource() {
        //判断时间间隔（毫秒）
        int preferenceInterval;
        long interval;
        Date curTime = new Date(System.currentTimeMillis());
        interval=curTime.getTime()-lastSMSTime.getTime();
        lastSMSTime=curTime;
        preferenceInterval=getPreferenceInterval();
        if(interval<preferenceInterval*60*1000&&!justStart){
```

```
                currentBell++;
                if(currentBell>5){
                    currentBell=5;
                }
            }else{
                currentBell=1;
            }
            justStart=false;
            return bellMap.get(currentBell);
        }
        //获取 Preference 设置
        private int getPreferenceInterval(){
            SharedPreferences settings = PreferenceManager.getDefaultSharedPreferences(this);
            int interval=Integer.valueOf(settings.getString("interval_config", "5"));
            return interval;
        }
    }
```

（3）编写文件 DotaBellActivity.java，在此为屏幕中的 3 个按钮设置了相应的处理事件。其主要代码如下所示。

```
public class DotaBellActivity extends Activity {
    private Button startButton;
    private Button endButton;
    private Button configButton;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        startButton=(Button)findViewById(R.id.startButton);
        endButton=(Button)findViewById(R.id.endButton);
        configButton=(Button)findViewById(R.id.configButton);
        startButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                Toast.makeText(DotaBellActivity.this, "start", Toast.LENGTH_SHORT).show();
                Intent serviceIntent=new Intent();
                serviceIntent.setClass(DotaBellActivity.this, BellService.class);
                startService(serviceIntent);
            }
        });
        endButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                Toast.makeText(DotaBellActivity.this, "end", Toast.LENGTH_SHORT).show();
                //停止服务
                Intent serviceIntent=new Intent();
                serviceIntent.setClass(DotaBellActivity.this, BellService.class);
                stopService(serviceIntent);
            }
        });
        configButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                Toast.makeText(DotaBellActivity.this, "config", Toast.LENGTH_SHORT).show();
                Intent preferenceIntent=new Intent();
                preferenceIntent.setClass(DotaBellActivity.this,
BellConfigPreference.class);
                startActivity(preferenceIntent);
            }
        });
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        android.os.Process.killProcess(android.os.Process.myPid());
    }
}
```

执行之后的效果如图 14-1 所示，单击屏幕中的按钮可以实现对应的铃声设置功能。

▲图 14-1　设置短信提示铃声的执行效果

14.2.3　AudioManager 基本应用——调节手机音量的大小

在使用移动手机时，我们经常需要调节音量的大小。在 Android API 中的 AudioManager 类中，提供了调节手机音量的相关方法。我们可以直接在程序中控制手机音量的大小，也可以切换声音的模式为振动或静音。

在接下来的内容中，将通过一个具体演示实例来讲解调节手机音量大小的方法。

题目	目的	源码路径
实例 14-2	调节手机音量的大小	\daima\14\tiao

在进行具体编码之前，需要预先准备素材图片，并将这些素材图片保存在“res\drawable”目录下。然后编写实例文件 example.java，下面开始讲解其具体实现代码。

（1）先声明系统需要的各个变量对象，具体实现代码如下所示。

```
public class example extends Activity
{
  /* 变量声明 */
  private ImageView myImage;
  private ImageButton downButton;
  private ImageButton upButton;
  private ImageButton normalButton;
  private ImageButton muteButton;
  private ImageButton vibrateButton;
  private ProgressBar myProgress;
  private AudioManager audioMa;
  private int volume=0;
```

（2）依次初始化 audioMa、myImage、myProgress、downButton、upButton、normalButton、muteButton、muteButton 和 vibrateButton 变量对象。其具体实现代码如下所示。

```
/* 对象初始化 */
audioMa = (AudioManager)getSystemService(Context.AUDIO_SERVICE);
myImage、 = (ImageView)findViewById(R.id.myImage);
myProgress = (ProgressBar)findViewById(R.id.myProgress);
downButton = (ImageButton)findViewById(R.id.downButton);
upButton = (ImageButton)findViewById(R.id.upButton);
normalButton = (ImageButton)findViewById(R.id.normalButton);
muteButton = (ImageButton)findViewById(R.id.muteButton);
vibrateButton = (ImageButton)findViewById(R.id.vibrateButton);
```

（3）分别设置初始的手机音量大小和初始的声音模式，具体实现代码如下所示。

```
/* 设置初始的手机音量 */
volume=audioMa.getStreamVolume(AudioManager.STREAM_RING);
myProgress.setProgress(volume);
/* 设置初始的声音模式 */
int mode=audioMa.getRingerMode();
if(mode==AudioManager.RINGER_MODE_NORMAL)
{
  myImage.setImageDrawable(getResources()
                         .getDrawable(R.drawable.normal));
}
else if(mode==AudioManager.RINGER_MODE_SILENT)
{
  myImage.setImageDrawable(getResources()
                         .getDrawable(R.drawable.mute));
}
else if(mode==AudioManager.RINGER_MODE_VIBRATE)
{
  myImage.setImageDrawable(getResources()
                         .getDrawable(R.drawable.vibrate));
}
```

（4）设置单击音量调小按钮 **downButton** 的处理事件，每单击依次设置音量调小一格，并设置调整后声音模式。其具体实现代码如下所示。

```
/* 音量调小声的 Button */
downButton.setOnClickListener(new Button.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    /* 设置音量调小一格 */
    audioMa.adjustVolume(AudioManager.ADJUST_LOWER, 0);
    volume=audioMa.getStreamVolume(AudioManager.STREAM_RING);
    myProgress.setProgress(volume);
    /* 设置调整后声音模式 */
    int mode=audioMa.getRingerMode();
    if(mode==AudioManager.RINGER_MODE_NORMAL)
    {
      myImage.setImageDrawable(getResources()
                             .getDrawable(R.drawable.normal));
    }
    else if(mode==AudioManager.RINGER_MODE_SILENT)
    {
      myImage.setImageDrawable(getResources()
                             .getDrawable(R.drawable.mute));
    }
    else if(mode==AudioManager.RINGER_MODE_VIBRATE)
    {
      myImage.setImageDrawable(getResources()
                            .getDrawable(R.drawable.vibrate));
    }
  }
});
```

（5）设置单击音量调小按钮 **upButton** 的处理事件，设置每次音量调大一格，然后设置调整后声音模式。其具体实现代码如下所示。

```
/* 音量调大声的 Button */
upButton.setOnClickListener(new Button.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    /* 设置音量调大一格 */
    audioMa.adjustVolume(AudioManager.ADJUST_RAISE, 0);
    volume=audioMa.getStreamVolume(AudioManager.STREAM_RING);
    myProgress.setProgress(volume);
```

```
    /* 设置调整后的声音模式 */
    int mode=audioMa.getRingerMode();
    if(mode==AudioManager.RINGER_MODE_NORMAL)
    {
      myImage.setImageDrawable(getResources()
                          .getDrawable(R.drawable.normal));
    }
    else if(mode==AudioManager.RINGER_MODE_SILENT)
    {
      myImage.setImageDrawable(getResources()
                          .getDrawable(R.drawable.mute));
    }
    else if(mode==AudioManager.RINGER_MODE_VIBRATE)
    {
      myImage.setImageDrawable(getResources()
                         .getDrawable(R.drawable.vibrate));
    }
  }
});
```

（6）设置单击调整正常铃声模式按钮 normalButton 的处理事件，单击后设置铃声模式为 NORMAL，并设置音量与声音模式。其具体实现代码如下所示。

```
/* 调整铃声模式为正常模式的 Button */
normalButton.setOnClickListener(new Button.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    /* 设置铃声模式为 NORMAL */
    audioMa.setRingerMode(AudioManager.RINGER_MODE_NORMAL);
    /* 设置音量与声音模式 */
    volume=audioMa.getStreamVolume(AudioManager.STREAM_RING);
    myProgress.setProgress(volume);
    myImage.setImageDrawable(getResources()
                        .getDrawable(R.drawable.normal));
  }
});
```

（7）设置单击调整静音铃声模式按钮 muteButton 的处理事件，首先设置铃声模式为 SILENT，然后设置音量与声音状态。其具体实现代码如下所示。

```
/* 调整铃声模式为静音模式的 Button */
muteButton.setOnClickListener(new Button.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    /* 设置铃声模式为 SILENT */
    audioMa.setRingerMode(AudioManager.RINGER_MODE_SILENT);
    /* 设置音量与声音状态 */
    volume=audioMa.getStreamVolume(AudioManager.STREAM_RING);
    myProgress.setProgress(volume);
    myImage.setImageDrawable(getResources()
                        .getDrawable(R.drawable.mute));
  }
});
```

（8）设置单击调整振动铃声模式按钮 vibrateButton 的处理事件，首先设置铃声模式为 VIBRATE，然后设置音量与声音状态。其具体实现代码如下所示。

```
    /* 调整铃声模式为振动模式的 Button */
    vibrateButton.setOnClickListener(new Button.OnClickListener()
    {
      @Override
      public void onClick(View arg0)
```

```
        {
          /* 设置铃声模式为 VIBRATE */
          audioMa.setRingerMode(AudioManager.RINGER_MODE_VIBRATE);
          /* 设置音量与声音状态 */
          volume=audioMa.getStreamVolume(AudioManager.STREAM_RING);
          myProgress.setProgress(volume);
          myImage.setImageDrawable(getResources()
                                   .getDrawable(R.drawable.vibrate));
        }
      });
    }
  }
```

执行后将会显示一个音量调节界面，我们既可以设置声音模式，也可以调整音量大小。执行效果如图 14-2 所示。

▲图 14-2 设置手机音量大小的执行效果

14.3 录音处理

在当今的智能手机中，几乎每一款手机都具备录音功能。在 Android 系统中，同样也可以实现录音处理。在本节的内容中，将详细讲解在 Android 系统中实现录音功能的方法，为读者进入本书后面知识的学习打下基础。

14.3.1 使用 MediaRecorder 接口录制音频

在 Android 系统中，通常采用 MediaRecorder 接口实现录制音频和视频功能。在录制音频文件之前，需要设置音频源、输出格式、录制时间和编码格式等。

类 AudioRecord 在 Android 顶层的 Java 应用程序中负责管理音频资源，通过 AudioRecord 对象来完成"pulling"（读取）数据，以记录从平台音频输入设备产生的数据。在 Android 应用中，可以通过以下方法从 AudioRecord 对象中读取音频数据。

- read(byte[], int, int);
- read(short[], int, int);
- read(ByteBuffer, int)。

在上述读取音频数据的方法中，使用 AudioRecord 是最方便的。

在 Android 系统中，当创建 AudioRecord 对象时会初始化 AudioRecord，并和音频缓冲区连接以缓冲新的音频数据。根据构造时指定的缓冲区大小，可以决定 AudioRecord 能够记录多长的数据。一般来说，从硬件设备读取的数据应小于整个记录缓冲区。

MediaRecorder 的内部类是 AudioRecord.OnRecordPositionUpdateListener，当 AudioRecord 收到一个由 setNotificationMarkerPosition(int)设置的通知标志，或由 setPositionNotificationPeriod(int)设置的周期更新记录的进度状态时，需要回调这个接口。

在类 MediaRecorder 中，包含了如表 14-1 中列出的常用方法。

表 14-1　　类 MediaRecorder 中的常用方法

方法名称	描　述
public void setAudioEncoder (int audio_encoder)	设置刻录的音频编码，其值可以通过 MediaRecoder 内部类的 MediaRecorder.AudioEncoder 的几个常量：AAC、AMR_NB、AMR_WB、DEFAULT
public void setAudioEncodingBitRate (int bitRate)	设置音频编码比特率

续表

方法名称	描　述
public void setAudioSource (int audio_source)	设置音频的来源，其值可以通过 MediaRecoder 内部类的 MediaRecorder. AudioSource 的几个常量来设置，通常设置的值 MIC 来源于麦克风
public void setCamera (Camera c)	设置摄像头用于刻录
public void setOutputFormat (int output_format)	设置输出文件的格式，其值可以通过 MediaRecoder 内部类 MediaRecorder. OutputFormat 的一些常量字段来设置。比如一些 3gp(THREE_GPP)、mp4(MPEG4)等
setOutputFile(String path)	设置输出文件的路径
setVideoEncoder(int video_encoder)	设置视频的编码格式。其值可以通过 MediaRecoder 内部类的 MediaRecorder. VideoEncoder 的几个常量：H263、H264、MPEG_4_SP
setVideoSource(int video_source)	设置刻录视频来源，其值可以通过 MediaRecorder 的内部类 MediaRecorder. VideoSource 来设置。比如可以设置刻录视频来源为摄像头：CAMERA
setVideoEncodingBitRate(int bitRate)	设置编码的比特率
setVideoSize(int width, int height)	设置视频的大尺寸
public void start()	开始刻录
public void prepare()	预期做准备
public void stop()	停止
public void release()	释放该对象资源

在 AudioRecord 中，有一个受保护的方法 protected void finalize()，用于通知 VM 收回此对象内存。方法 finalize ()只能用在运行的应用程序，没有任何线程再使用此对象，并告诉垃圾回收器回收此对象。方法 finalize ()用于释放系统资源，由垃圾回收器清除此对象。在执行期间，调用方法 finalize ()可能会立即抛出未定义异常，但是可以忽略。

注意　VM 保证对象可以一次或多次调用 finalize()，但并不保证 finalize()会马上执行。例如，对象 B 的 finalize()可能延迟执行，等待对象 A 的 finalize()延迟收回 A 的内存。为了安全起见，请看 ReferenceQueue，它提供了更多地控制 VM 的垃圾回收。

接下来将通过一个具体实例的实现过程，来讲解使用 MediaRecorder 实现音频录制的方法。

题目	目的	源码路径
实例 14-3	使用 MediaRecorder 录制音频	\daima\14\luzhi

在本实例中插入了 4 个按钮，分别实现录音、停止录音、播放录音和删除录音这 4 种操作。为了能够不限制录音的长度，先将录音暂时保存到存储卡，当录音完毕后，再将录音文件显示在 ListView 列表中。单击文件后，可以播放或删除录音文件。本实例的实现文件是 example.java，具体实现流程如下所示。

（1）分别构造 4 个按钮对象和 2 个文本对象，然后设置按钮状态不可选。其具体实现代码如下所示。

```
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
```

```
/* 设置 4 个按钮和 2 个文本 */
myButton1 = (ImageButton) findViewById(R.id.ImageButton01);
myButton2 = (ImageButton) findViewById(R.id.ImageButton02);
myButton3 = (ImageButton) findViewById(R.id.ImageButton03);
myButton4 = (ImageButton) findViewById(R.id.ImageButton04);
myListView1 = (ListView) findViewById(R.id.ListView01);
myTextView1 = (TextView) findViewById(R.id.TextView01);
/* 设置按钮状态不可选 */
myButton2.setEnabled(false);
myButton3.setEnabled(false);
myButton4.setEnabled(false);
```

（2）通过方法 sdCardExit 判断是否插入 SD 卡，然后将获取的 SD 卡路径作为录音的文件位置，并取得 SD 卡目录里的所有“.amr”格式的文件，最后将 ArrayAdapter 添加到 ListView 对象中以列表显示录音文件。其具体实现代码如下所示。

```
/* 判断是否插入 SD 卡 */
sdCardExit = Environment.getExternalStorageState().equals(
    android.os.Environment.MEDIA_MOUNTED);
/* 取得 SD 卡路径作为录音的文件位置 */
if (sdCardExit)
  myRecAudioDir = Environment.getExternalStorageDirectory();
/* 取得 SD 卡目录里的所有.amr 文件 */
getRecordFiles();
adapter = new ArrayAdapter<String>(this,
    R.layout.my_simple_list_item, recordFiles);
/* 将 ArrayAdapter 添加到 ListView 对象中 */
myListView1.setAdapter(adapter);
```

（3）编写单击录音按钮后的录音处理事件，先创建录音频文件，然后设置录音来源为麦克风，最后通过 myTextView1.setText(“录音中”)设置录音过程显示的提示文本。其具体实现代码如下所示。

```
/* 单击录音按钮的处理事件 */
myButton1.setOnClickListener(new ImageButton.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    try
    {
      if (!sdCardExit)
      {
        Toast.makeText(example.this, "请插入 SD Card",
            Toast.LENGTH_LONG).show();
        return;
      }
      /* 创建录音频文件 */
      myRecAudioFile = File.createTempFile(strTempFile, ".amr",
          myRecAudioDir);
      mMediaRecorder01 = new MediaRecorder();
      /* 设置录音来源为麦克风 */
      mMediaRecorder01
          .setAudioSource(MediaRecorder.AudioSource.MIC);
      mMediaRecorder01
          .setOutputFormat(MediaRecorder.OutputFormat.DEFAULT);
      mMediaRecorder01
          .setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);
      mMediaRecorder01.setOutputFile(myRecAudioFile
          .getAbsolutePath());
      mMediaRecorder01.prepare();
      mMediaRecorder01.start();
      myTextView1.setText("录音中");
      myButton2.setEnabled(true);
      myButton3.setEnabled(false);
      myButton4.setEnabled(false);
```

```
        isStopRecord = false;
      }
      catch (IOException e)
      {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
  });
```

（4）编写单击停止按钮的处理事件，首先使用方法 mMediaRecorder01.stop()停止录音，然后将录音文件名传递给 Adapter。其具体实现代码如下所示。

```
/* 停止 */
myButton2.setOnClickListener(new ImageButton.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    // TODO Auto-generated method stub
    if (myRecAudioFile != null)
    {
      /* 停止录音 */
      mMediaRecorder01.stop();
      mMediaRecorder01.release();
      mMediaRecorder01 = null;
      /* 将录音频文件名传递给 Adapter */
      adapter.add(myRecAudioFile.getName());
      myTextView1.setText("停止: " + myRecAudioFile.getName());
      myButton2.setEnabled(false);
      isStopRecord = true;
    }
  }
});
```

（5）编写单击播放按钮的处理事件，单击后将打开播放的程序。其主要代码如下所示。

```
/* 播放 */
myButton3.setOnClickListener(new ImageButton.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    // TODO Auto-generated method stub
    if (myPlayFile != null && myPlayFile.exists())
    {
      /* 打开播放的程序 */
      openFile(myPlayFile);
    }
  }
});
```

（6）编写单击删除按钮的处理事件，首先在 Adapter 删除录音文件名，然后删除录制的文件。其具体实现代码如下所示。

```
/* 删除事件 */
myButton4.setOnClickListener(new ImageButton.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    if (myPlayFile != null)
    {
      /* 先将 Adapter 删除文件名 */
      adapter.remove(myPlayFile.getName());
      /* 删除文件 */
      if (myPlayFile.exists())
```

```
        myPlayFile.delete();
      myTextView1.setText("完成删除");
    }
  }
});
```

（7）编写单击 Adapter 列表中某个录制文件的处理事件。如果有文件，单击后将删除及播放按钮设置为 Enable，然后输出选择提示语句。其具体实现代码如下所示。

```
  myListView1.setOnItemClickListener
  (new AdapterView.OnItemClickListener()
  {
    @Override
    public void onItemClick(AdapterView<?> arg0, View arg1,
        int arg2, long arg3)
    {
      /* 当有点击档名时将删除及播放按钮设置为 Enable */
      myButton3.setEnabled(true);
      myButton4.setEnabled(true);
      myPlayFile = new File(myRecAudioDir.getAbsolutePath()
          + File.separator
          + ((CheckedTextView) arg1).getText());
      myTextView1.setText("你选的是: "
          + ((CheckedTextView) arg1).getText());
    }
  });
}
```

（8）定义方法 onStop() 实现停止录音操作，具体实现代码如下所示。

```
protected void onStop()
{
  if (mMediaRecorder01 != null && !isStopRecord)
  {
    /* 停止录音 */
    mMediaRecorder01.stop();
    mMediaRecorder01.release();
    mMediaRecorder01 = null;
  }
  super.onStop();
}
```

（9）定义方法 getRecordFiles()来获取文件的长度，在此设置只能获取“.amr”格式的文件。其具体实现代码如下所示。

```
private void getRecordFiles()
{
  recordFiles = new ArrayList<String>();
  if (sdCardExit)
  {
    File files[] = myRecAudioDir.listFiles();
    if (files != null)
    {
      for (int i = 0; i < files.length; i++)
      {
        if (files[i].getName().indexOf(".") >= 0)
        {
          /* 只取.amr 文件 */
          String fileS = files[i].getName().substring(
              files[i].getName().indexOf("."));
          if (fileS.toLowerCase().equals(".amr"))
            recordFiles.add(files[i].getName());
        }
      }
    }
  }
}
```

（10）定义方法 openFile(File f)来打开播放指定的录音文件，主要代码如下所示。

```
/* 打开播放录音文件的程序 */
private void openFile(File f)
{
  Intent intent = new Intent();
  intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
  intent.setAction(android.content.Intent.ACTION_VIEW);
  String type = getMIMEType(f);
  intent.setDataAndType(Uri.fromFile(f), type);
  startActivity(intent);
}
```

（11）定义方法 getMIMEType(File f)设置系统可接受的文件类型，在此设置了 audio 类型、image 类型和其他类型。其具体实现代码如下所示。

```
  private String getMIMEType(File f)
  {
    String end = f.getName().substring(
        f.getName().lastIndexOf(".") + 1, f.getName().length())
        .toLowerCase();
    String type = "";
    if (end.equals("mp3") || end.equals("aac") || end.equals("aac")
        || end.equals("amr") || end.equals("mpeg")
        || end.equals("mp4"))
    {
      type = "audio";
    }
    else if (end.equals("jpg") || end.equals("gif")
        || end.equals("png") || end.equals("jpeg"))
    {
      type = "image";
    }
    else
    {
      type = "*";
    }
    type += "/*";
    return type;
  }
}
```

还需在文件 AndroidManifest.xml 中声明录音权限，具体实现代码如下所示。

```
<uses-permission android:name="android.permission.RECORD_AUDIO">
```

至此，整个实例介绍完毕，执行后的效果如图 14-3 所示。当单击“录音”按钮时开始录音处理，如图 14-4 所示。当单击“停止”按钮后停止录音处理，并在列表中显示录制的音频文件。当选中音频文件并单击“删除”按钮后会删除选中音频文件，单击“播放”按钮后会播放选中的音频文件。

▲图 14-3　录制音频初始效果　　▲图 14-4　正在录音

14.3.2　使用 AudioRecord 接口录制音频

类 AudioRecord 可以在 Java 应用程序中管理音频资源，通过 AudioRecord 对象来完成“pulling”（读取）数据的方法以记录从平台音频输入设备产生的数据。

1. 常量

AudioRecord 中包含的常量如下所示。

- public static final int ERROR：表示操作失败，常量值：-1 (0xffffffff)；
- public static final int ERROR_BAD_VALUE：表示使用了一个不合理的值导致的失败，常量值：-2 (0xfffffffe)；
- public static final int ERROR_INVALID_OPERATION：表示不恰当的方法导致的失败，常量值：-3 (0xfffffffd)；
- public static final int RECORDSTATE_RECORDING：指示 AudioRecord 录制状态为“正在录制”，常量值：3 (0x00000003)；
- public static final int RECORDSTATE_STOPPED：指示 AudioRecord 录制状态为“不在录制”，常量值：1 (0x00000001)；
- public static final int STATE_INITIALIZED：指示 AudioRecord 准备就绪，常量值：1 (0x00000001)；
- public static final int STATE_UNINITIALIZED：指示 AudioRecord 状态没有初始化成功，常量值：0 (0x00000000)；
- public static final int SUCCESS：表示操作成功，常量值：0 (0x00000000)。

2. 构造函数

AudioRecord 的构造函数是 AudioRecord，具体定义格式如下所示。

```
public AudioRecord (int audioSource, int sampleRateInHz, int channelConfig, int
audioFormat, int bufferSizeInBytes)
```

各个参数的具体说明如下所示。

- audioSource：录制源；
- sampleRateInHz：默认采样率，单位 Hz。44 100Hz 是当前唯一能保证在所有设备上工作的采样率，在一些设备上还有 22 050Hz，16 000Hz 或 11 025Hz；
- channelConfig：描述音频通道设置；
- audioFormat：音频数据保证支持此格式；
- bufferSizeInBytes：在录制过程中，音频数据写入缓冲区的总数（字节）。从缓冲区读取的新音频数据总会小于此值。用 getMinBufferSize(int, int, int)返回 AudioRecord 实例创建成功后的最小缓冲区，如果其设置的值比 getMinBufferSize()还小，则会导致初始化失败。

3. 公共方法

AudioRecord 中的公共方法如下所示。

（1）public int getAudioFormat ()：返回设置的音频数据格式。

（2）public int getAudioSource ()：返回音频录制源。

（3）public int getChannelConfiguration ()：返回设置的频道设置。

（4）public int getChannelCount ()：返回设置的频道数目。

（5）public static int getMinBufferSize (int sampleRateInHz, int channelConfig, int audioFormat)：返回成功创建 AudioRecord 对象所需要的最小缓冲区的值。

> 注意：这个值并不保证在此负荷下的流畅录制，应根据预期的频率选择更高的值，AudioRecord 实例在推送新数据时使用此值。

各个参数的具体说明如下。

- sampleRateInHz：默认采样率，单位是 Hz。
- channelConfig：描述音频通道设置。
- audioFormat：音频数据保证支持此格式。参见 ENCODING_PCM_16BIT。

如果硬件不支持录制参数，或输入了一个无效的参数，则返回 ERROR_BAD_VALUE，如果硬件查询到输出属性没有实现，或最小缓冲区用 byte 表示，则返回 ERROR。

（6）public int getNotificationMarkerPosition ()：返回通知，标记框架中的位置。

（7）public int getPositionNotificationPeriod ()：返回通知，更新框架中的时间位置。

（8）public int getRecordingState ()：返回 AudioRecord 实例的录制状态。

（9）public int getSampleRate ()：返回设置的音频数据样本采样率，单位是 Hz。

（10）public int getState ()：返回 AudioRecord 实例的状态。

（11）public int read (short[] audioData, int offsetInShorts, int sizeInShorts)：从音频硬件录制缓冲区读取数据。

各个参数的具体说明如下所示。

- audioData：写入的音频录制数据。
- offsetInShorts：目标数组 audioData 的起始偏移量。
- sizeInShorts：请求读取的数据大小。

返回值是返回 short 型数据，表示读取到的数据。如果对象属性没有初始化，则返回 ERROR_INVALID_OPERATION；如果参数不能解析成有效的数据或索引，则返回 ERROR_BAD_VALUE。返回数值不会超过 sizeInShorts。

（12）public int read (byte[] audioData, int offsetInBytes, int sizeInBytes)：从音频硬件录制缓冲区读取数据，读入缓冲区的总 byte 数。如果对象属性没有初始化，则返回 ERROR_INVALID_OPERATION；如果参数不能解析成有效的数据或索引，则返回 ERROR_BAD_VALUE。读取的总 byte 数不会超过 sizeInBytes。各个参数的具体说明如下所示。

- audioData：写入的音频录制数据。
- offsetInBytes：audioData 的起始偏移值，单位是 byte。
- sizeInBytes：读取的最大字节数。

（13）public int read (ByteBuffer audioBuffer, int sizeInBytes)：从音频硬件录制缓冲区读取数据，直接复制到指定缓冲区。如果 audioBuffer 不是直接的缓冲区，此方法总是返回 0。读入缓冲区的总 byte 数，如果对象属性没有初始化，则返回 ERROR_INVALID_OPERATION；如果参数不能解析成有效的数据或索引，则返回 ERROR_BAD_VALUE。 读取的总 byte 数不会超过 sizeInBytes。各个参数的具体说明如下所示。

- audioBuffer：存储写入音频录制数据的缓冲区。
- sizeInBytes：请求的最大字节数。

（14）public void release ()：释放本地 AudioRecord 资源。 对象不能经常使用此方法，而且在调用 release()后，必须设置引用为 null。

（15）public int setNotificationMarkerPosition (int markerInFrames)：如果设置了 setRecordPositionUpdateListener(OnRecordPositionUpdateListener)或 setRecordPositionUpdateListener(OnRecordPositionUpdateListener, Handler)，则通知监听者设置位置标记。

参数 markerInFrames 表示在框架中快速标记位置。

（16）public int setPositionNotificationPeriod (int periodInFrames)：如果设置了 setRecordPositionUpdateListener(OnRecordPositionUpdateListener)或 setRecordPositionUpdateListener(OnRecordPosition

UpdateListener, Handler)，则通知监听者设置时间标记。

参数 markerInFrames 表示在框架中快速更新时间标记。

（17）public void setRecordPositionUpdateListener (AudioRecord.OnRecordPositionUpdateListener listener, Handler handler)：当之前设置的标志已经成立，或者周期录制位置更新时，设置处理监听者。使用此方法将 Handler 和别的线程联系起来接收 AudioRecord 事件，比创建 AudioTrack 实例更好一些。

参数 handler 用来接收事件通知消息。

（18）public void setRecordPositionUpdateListener (AudioRecord.OnRecordPositionUpdateListener listener)：当之前设置的标志已经成立，或者周期录制位置更新时，设置处理监听者。

（19）public void startRecording ()：表示 AudioRecord 实例开始进行录制。

4. 受保护方法

在 AudioRecord 中的受保护方法是 protected void finalize ()，此方法用于通知 VM 回收此对象内存，只能被用在运行的应用程序，没有任何线程再使用此对象，并告诉垃圾回收器回收此对象。

方法 finalize ()用于释放系统资源，由垃圾回收器清除此对象。默认没有实现，由 VM 来决定，但子类根据需要可重写 finalize()。在执行期间，调用此方法可能会立即抛出未定义异常，但是可以忽略。

注意：VM 保证对象可以一次或多次调用 finalize()，但并不保证 finalize()会马上执行。例如，对象 B 的 finalize()可能延迟执行，等待对象 A 的 finalize()延迟收回 A 的内存。为了安全起见，请看 ReferenceQueue，它提供了更多地控制 VM 的垃圾回收。

另外，需要在 Activity 的线程里面创建 AudioRecord 对象，可以在独立的线程里面进行读取数据，否则像华为 U8800 之类的手机录音时会出错。

接下来将通过一个具体实例的实现过程讲解使用 AudioRecord 实现音频录制的方法。

题目	目的	源码路径
实例 14-4	使用 AudioRecord 录制音频	\daima\14\testRecord

在 Android 系统中，因为 MediaRecorder 可以直接把麦克风的数据存到文件，并且能够直接进行编码（如 AMR、MP3 等），而 AudioRecord 则是读取麦克风的音频流。本实例使用 AudioRecord 读取音频流，使用 AudioTrack 播放音频流，通过“边读边播放”以及增大音量的方式实现一个简单的助听器程序。

本实例的具体实现流程如下所示。

（1）编写布局文件 main.xml，主要代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <Button android:layout_height="wrap_content" android:id="@+id/btnRecord"
      android:layout_width="fill_parent" android:text="开始边录边放"></Button>
   <Button android:layout_height="wrap_content"
      android:layout_width="fill_parent"  android:text=" 停 止 "  android:id="@+id/
btnStop"></Button>
   <Button android:layout_height="wrap_content" android:id="@+id/btnExit"
      android:layout_width="fill_parent" android:text="退出"></Button>
   <TextView android:id="@+id/TextView01" android:layout_height="wrap_content"
      android:text="程序音量调节" android:layout_width="fill_parent"></TextView>
```

```
    <SeekBar android:layout_height="wrap_content" android:id="@+id/skbVolume"
        android:layout_width="fill_parent"></SeekBar>
</LinearLayout>
```

（2）在文件 AndroidManifest.xml 中声明权限，主要代码如下所示。

```
<uses-permission android:name="android.permission.RECORD_AUDIO"></uses-permission>
```

（3）编写文件 TestRecordActivity.java，主要代码如下所示。

```
public class TestRecordActivity extends Activity {
    Button btnRecord, btnStop, btnExit;
    SeekBar skbVolume;//调节音量
    boolean isRecording = false;//是否录放的标记
    static final int frequency = 44100;
    static final int channelConfiguration = AudioFormat.CHANNEL_CONFIGURATION_MONO;
    static final int audioEncoding = AudioFormat.ENCODING_PCM_16BIT;
    int recBufSize,playBufSize;
    AudioRecord audioRecord;
    AudioTrack audioTrack;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        setTitle("助听器");
        recBufSize = AudioRecord.getMinBufferSize(frequency,
                channelConfiguration, audioEncoding);
        playBufSize=AudioTrack.getMinBufferSize(frequency,
                channelConfiguration, audioEncoding);
        audioRecord = new AudioRecord(MediaRecorder.AudioSource.MIC, frequency,
                channelConfiguration, audioEncoding, recBufSize);
        audioTrack = new AudioTrack(AudioManager.STREAM_MUSIC, frequency,
                channelConfiguration, audioEncoding,
                playBufSize, AudioTrack.MODE_STREAM);
        btnRecord = (Button) this.findViewById(R.id.btnRecord);
        btnRecord.setOnClickListener(new ClickEvent());
        btnStop = (Button) this.findViewById(R.id.btnStop);
        btnStop.setOnClickListener(new ClickEvent());
        btnExit = (Button) this.findViewById(R.id.btnExit);
        btnExit.setOnClickListener(new ClickEvent());
        skbVolume=(SeekBar)this.findViewById(R.id.skbVolume);
        skbVolume.setMax(100);//音量调节的极限
        skbVolume.setProgress(70);//设置 seekbar 的位置值
        audioTrack.setStereoVolume(0.7f, 0.7f);//设置当前音量大小
        skbVolume.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {

            @Override
            public void onStopTrackingTouch(SeekBar seekBar) {
                float vol=(float)(seekBar.getProgress())/(float)(seekBar.getMax());
                audioTrack.setStereoVolume(vol, vol);//设置音量
            }

            @Override
            public void onStartTrackingTouch(SeekBar seekBar) {
            }

            public void onProgressChanged(SeekBar seekBar, int progress,
                    boolean fromUser) {
            }
        });
    }
    protected void onDestroy() {
        super.onDestroy();
        android.os.Process.killProcess(android.os.Process.myPid());
    }

    class ClickEvent implements View.OnClickListener {
```

```
        @Override
        public void onClick(View v) {
            if (v == btnRecord) {
                isRecording = true;
                new RecordPlayThread().start();// 开一条线程边录边放
            } else if (v == btnStop) {
                isRecording = false;
            } else if (v == btnExit) {
                isRecording = false;
                TestRecordActivity.this.finish();
            }
        }
    }

    class RecordPlayThread extends Thread {
        public void run() {
            try {
                byte[] buffer = new byte[recBufSize];
                audioRecord.startRecording();//开始录制
                audioTrack.play();//开始播放

                while (isRecording) {
                    //从MIC保存数据到缓冲区
                    int bufferReadResult = audioRecord.read(buffer, 0,
                            recBufSize);

                    byte[] tmpBuf = new byte[bufferReadResult];
                    System.arraycopy(buffer, 0, tmpBuf, 0, bufferReadResult);
                    //写入数据即播放
                    audioTrack.write(tmpBuf, 0, tmpBuf.length);
                }
                audioTrack.stop();
                audioRecord.stop();
            } catch (Throwable t) {
                Toast.makeText(TestRecordActivity.this, t.getMessage(), 1000);
            }
        }
    };
}
```

执行后可以实现录音功能，效果如图 14-5 所示。程序音量调节只是程序内部调节音量而已，要调到最大音量还需要手动设置系统音量。

▲图 14-5　录制音频的执行效果

14.4 播放音频

在当今的智能手机中，几乎每一款手机都具备音频播放功能，例如最常见的 MP3 音乐文件播放。在 Android 系统中，同样也可以播放常见的音频文件。在本节的内容中，将详细讲解在 Android 系统中实现音频播放功能的基本流程，为读者进入本书后面知识的学习打下基础。

14.4.1　使用 AudioTrack 播放音频

要想学好 AudioTrack API，读者可以从分析 Android 源码中的 Java 源码做起。

第一段 Java 代码如下所示。

```
//根据采样率、采样精度、单双声道来得到 frame 的大小。
int bufsize = AudioTrack.getMinBufferSize(8000,              //每秒 8 000 个点
AudioFormat.CHANNEL_CONFIGURATION_STEREO,                    //双声道
AudioFormat.ENCODING_PCM_16BIT);                             //一个采样点 16 比特～2 字节
//创建 AudioTrack
AudioTrack trackplayer = new AudioTrack(AudioManager.STREAM_MUSIC, 8000,
```

```
AudioFormat.CHANNEL_CONFIGURATION_ STEREO,
AudioFormat.ENCODING_PCM_16BIT,
bufsize,
AudioTrack.MODE_STREAM);
 trackplayer.play() ;//开始
trackplayer.write(bytes_pkg, 0, bytes_pkg.length) ;//往 track 中写数据
…
trackplayer.stop();//停止播放
trackplayer.release();//释放底层资源
```

针对上述代码有如下两点说明。

（1）AudioTrack.MODE_STREAM。

在 AudioTrack 中存在 MODE_STATIC 和 MODE_STREAM 两种分类。STREAM 的意思是由用户在应用程序通过 write 方式把数据一次一次地写到 audiotrack 中。这个和我们在 socket 中发送数据一样，应用层从某个地方获取数据，例如通过编解码得到 PCM 数据，然后写入 Audiotrack。这种方式的坏处就是总在 Java 层和 Native 层交互，效率损失较大。

而 STATIC 的意思是一开始创建的时候，就把音频数据放到一个固定的 buffer，然后直接传给 audiotrack，后续就不用一次一次地写了。AudioTrack 会自己播放这个 buffer 中的数据。这种方法对于铃声等内存占用较小、延时要求较高的声音来说很适用。

（2）StreamType。

这个参数在构造 AudioTrack 的第一个参数中使用。它和 Android 中的 AudioManager 有关系，涉及手机上的音频管理策略。

Android 将系统的声音分为以下几种常见的类。

- STREAM_ALARM：警告声；
- STREAM_MUSCI：音乐声，例如 music 等；
- STREAM_RING：铃声；
- STREAM_SYSTEM：系统声音；
- STREAM_VOCIE_CALL：电话声音；

其实系统将这几种声音的数据分开管理，所以，这个参数对 AudioTrack 的含义就是告诉系统现在想使用的是哪种类型的声音，这样系统就可以对应管理它们了。

接下来将通过一个具体实例的实现过程讲解使用 AudioTrack 播放音频的方法。

题目	目的	源码路径
实例 14-5	使用 AudioTrack 播放音频	\daima\14\AuTrack

此实例比较简单，主程序文件 AuTrack.java 的主要实现代码如下所示。

```
public class AuTrack extends Activity {
   public static final int MENU_START_ID = Menu.FIRST;
   public static final int MENU_STOP_ID = Menu.FIRST + 1;
   public static final int MENU_EXIT_ID = Menu.FIRST + 2;
   protected PCMAudioTrack m_player;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
   }
   public boolean onCreateOptionsMenu(Menu aMenu) {
      boolean res = super.onCreateOptionsMenu(aMenu);
      aMenu.add(0, MENU_START_ID, 0, "START");
      aMenu.add(0, MENU_STOP_ID, 0, "STOP");
      aMenu.add(0, MENU_EXIT_ID, 0, "EXIT");
      return res;
```

```
    }
    public boolean onOptionsItemSelected(MenuItem aMenuItem) {
        switch (aMenuItem.getItemId()) {
        case MENU_START_ID: {
            m_player = new PCMAudioTrack();
            m_player.init();
            m_player.start();
        }
            break;
        case MENU_STOP_ID: {
            m_player.free();
            m_player = null;
        }
            break;
        case MENU_EXIT_ID: {
            int pid = android.os.Process.myPid();
            android.os.Process.killProcess(pid);
        }
            break;
        default:
            break;
        }
        return super.onOptionsItemSelected(aMenuItem);
    }
}

class PCMAudioTrack extends Thread {
    protected AudioTrack m_out_trk;
    protected int m_out_buf_size;
    protected byte[] m_out_bytes;
    protected boolean m_keep_running;
    final String FILE_PATH = "/sdcard/";
    final String FILE_NAME = "test.wav";
    File file;
    FileInputStream in;

    public void init() {
        try {
            file = new File(FILE_PATH , FILE_NAME);
            file.createNewFile();
            in = new FileInputStream(file);
            m_keep_running = true;
            m_out_buf_size = AudioTrack.getMinBufferSize(44100,
                    AudioFormat.CHANNEL_CONFIGURATION_STEREO,// CHANNEL_CONFIGURATION_MONO,
                    AudioFormat.ENCODING_PCM_16BIT);
            m_out_trk = new AudioTrack(AudioManager.STREAM_MUSIC, 44100,
                    AudioFormat.CHANNEL_CONFIGURATION_STEREO,// CHANNEL_CONFIGURATION_MONO,
                    AudioFormat.ENCODING_PCM_16BIT,
                    m_out_buf_size,
                    AudioTrack.MODE_STREAM);
            m_out_bytes = new byte[m_out_buf_size];
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void free() {
        m_keep_running = false;
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            Log.d("sleep exceptions...\n", "");
        }
    }

    public void run() {
        byte[] bytes_pkg = null;
        m_out_trk.play();
        while (m_keep_running) {
```

```
            try {
                in.read(m_out_bytes);
                bytes_pkg = m_out_bytes.clone();
                m_out_trk.write(bytes_pkg, 0, bytes_pkg.length);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        m_out_trk.stop();
        m_out_trk = null;
        try {
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

▲图 14-6　播放音频的执行效果

执行之后将会播放根目录中的音频文件 test.wav，如图 14-6 所示。

14.4.2　使用 MediaPlayer 播放音频

MediaPlayer 的功能比较强大，既可以播放音频，也可以播放视频，还可以通过 VideoView 来播放视频。虽然 VideoView 比 MediaPlayer 简单、易用，但是定制性不如用 MediaPlayer，读者需要视具体情况来选择处理方式。MediaPlayer 播放音频比较简单，但是要播放视频就需要 SurfaceView。SurfaceView 比普通的自定义 View 更有绘图上的优势，它支持完全的 OpenGL ES 库。

MediaPlayer 能被用来控制音频/视频文件或流媒体的回放，可以在 VideoView 里找到关于使用该类中的这个方法的例子。使用 MediaPlayer 实现视音频播放的基本步骤如下所示。

（1）生成 MediaPlayer 对象，根据播放文件从不同的地方使用不同的生成方式（具体过程可以参考 MediaPlayer API）。

（2）得到 MediaPlayer 对象后，根据你的实际需要调用不同的方法，如 start()、storp()、pause()、release()等。

读者需要注意的是，在不需要播放的时候要及时释放掉与 MediaPlayer 对象相连接的播放文件，因为直接使用 MediaPlayer 对象一般都是进行音频播放。

1. MediaPlayer 的状态

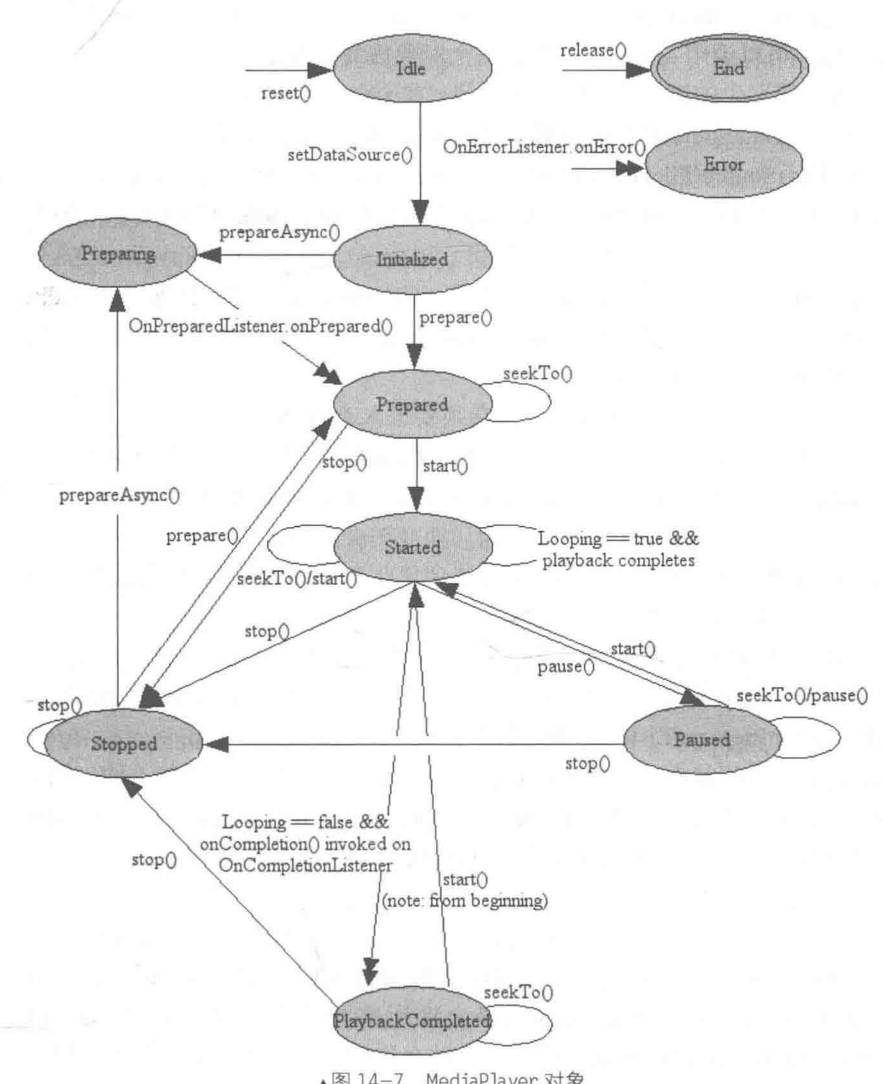
图 14-7 显示了一个 MediaPlayer 对象被支持的播放控制操作驱动的生命周期和状态。其中椭圆代表 MediaPlayer 对象可能驻留的状态，弧线表示驱动 MediaPlayer 在各个状态之间迁移的播放控制操作。这里有两种类型的弧线。由一个箭头开始的弧线代表同步的方法调用，而以双箭头开头的弧线代表异步方法调用。

通过图 14-7 可以知道一个 MediaPlayer 对象有如下的状态。

（1）当一个 MediaPlayer 对象被刚刚用 new 操作符创建或是调用了 reset()方法后，它就处于 Idle 状态。当调用了 release()方法后，它就处于 End 状态。这两种状态之间是 MediaPlayer 对象的生命周期。

在一个新构建的 MediaPlayer 对象和一个调用了 reset()方法的 MediaPlayer 对象之间有一个微小的但是十分重要的差别。在处于 Idle 状态时，调用 getCurrentPosition()、getDuration()、getVideoHeight()、getVideoWidth()、setAudioStreamType(int)、setLooping(boolean)、setVolume(float, float)、pause()、start()、stop()、seekTo(int)、prepare()或者 prepareAsync()方法都是编程错误。当一个 MediaPlayer 对象刚被构建的时候，内部的播放引擎和对象的状态都没有改变，在这个时候调

用以上的那些方法，框架将无法回调客户端程序注册的 OnErrorListener.onError()方法；但若这个 MediaPlayer 对象调用了 reset()方法之后，再调用以上的那些方法，内部的播放引擎就会回调客户端程序注册的 OnErrorListener.onError()方法了，并将错误的状态传入。



▲图 14-7 MediaPlayer 对象

笔者在此建议，一旦一个 MediaPlayer 对象不再被使用，应立即调用 release()方法释放在内部的播放引擎中与这个 MediaPlayer 对象关联的资源。资源可能包括如硬件加速组件的单态组件。若没有调用 release()方法可能会导致之后的 MediaPlayer 对象实例无法使用这种单态硬件资源，从而退回到软件实现或运行失败。一旦 MediaPlayer 对象进入了 End 状态，它不能再被使用，也没有办法再迁移到其他状态。

此外，在使用 new 操作符创建的 MediaPlayer 对象处于 Idle 状态，而那些通过重载的 create()便利方法创建的 MediaPlayer 对象却不是处于 Idle 状态。事实上，如果成功调用了重载的 create()方法，那么这些对象已经是 Prepare 状态了。

（2）在一般情况下，由于种种原因，一些播放控制操作可能会失败，如不支持的音频/视频格式、缺少隔行扫描的音频/视频、分辨率太高、流超时等原因等等。因此，错误报告和恢复在这种情况下是非常重要的。有时，由于编程错误，在处于无效状态的情况下调用了一个播放控制操作

可能发生。在所有这些错误条件下，内部的播放引擎会调用一个由客户端程序员提供的 OnErrorListener.onError()方法。客户端程序员可以通过调用 MediaPlayer.setOnErrorListener（android.media.MediaPlayer.OnErrorListener）方法来注册 OnErrorListener。

如一旦发生错误，MediaPlayer 对象会进入 Error 状态。为了重用一个处于 Error 状态的 MediaPlayer 对象，可以调用 reset()方法来把这个对象恢复成 Idle 状态。注册一个 OnErrorListener 来获知内部播放引擎发生的错误是好的编程习惯。在不合法的状态下调用一些方法，如 prepare()、prepareAsync()和 setDataSource()方法会抛出 IllegalStateException 异常。

（3）调用 setDataSource(FileDescriptor)方法、setDataSource(String)方法、setDataSource(Context、Uri)方法、setDataSource(FileDescriptor,long,long)方法会使处于 Idle 状态的对象迁移到 Initialized 状态。

当 MediaPlayer 对象处于其他的状态下，调用 setDataSource()方法，会抛出 IllegalStateException 异常。好的编程习惯是不要疏忽了调用 setDataSource()方法的时候可能会抛出的 IllegalArgument Exception 异常和 IOException 异常。

（4）在开始播放之前，MediaPlayer 对象必须要进入 Prepared 状态。

在此有如下两种方法（同步和异步）可以使 MediaPlayer 对象进入 Prepared 状态。

- 调用 prepare()方法（同步）：此方法返回就表示该 MediaPlayer 对象已经进入了 Prepared 状态；
- 调用 prepareAsync()方法（异步）：此方法会使此 MediaPlayer 对象进入 Preparing 状态并返回，而内部的播放引擎会继续未完成的准备工作。

当同步版本返回时或异步版本的准备工作全部完成时就会调用客户端程序员提供的 OnPreparedListener.onPrepared()监听方法。可以调用方法 MediaPlayer.setOnPreparedListener(android.media.MediaPlayer.OnPreparedListener)来注册 OnPreparedListener.

Preparing 是一个中间状态，如果在此状态下调用任何影响播放功能的方法，则最终的运行结果是未知的。在不合适的状态下调用 prepare()和 prepareAsync()方法会抛出 IllegalStateException 异常。当 MediaPlayer 对象处于 Prepared 状态的时候，可以调整音频/视频的属性，如音量、播放时是否一直亮屏、循环播放等。

（5）在要开始播放时必须调用 start()方法。当此方法成功返回时，MediaPlayer 的对象处于 Started 状态。isPlaying()方法可以被调用来测试某个 MediaPlayer 对象是否在 Started 状态。

当处于 Started 状态时，内部播放引擎会调用客户端程序员提供的 OnBufferingUpdate Listener.onBufferingUpdate()回调方法，此回调方法允许应用程序追踪流播放的缓冲的状态。对一个已经处于 Started 状态的 MediaPlayer 对象调用 start()方法没有影响。

（6）播放可以被暂停、停止以及调整当前播放位置。当调用 pause()方法并返回时，会使 MediaPlayer 对象进入 Paused 状态。注意 Started 与 Paused 状态的相互转换在内部的播放引擎中是异步的。所以可能需要一点时间在 isPlaying()方法中更新状态，若在播放流内容，这段时间可能会有几秒钟。

调用 start()方法会让一个处于 Paused 状态的 MediaPlayer 对象从之前暂停的地方恢复播放。当调用 start()方法返回的时候，MediaPlayer 对象的状态会又变成 Started 状态。对一个已经处于 Paused 状态的 MediaPlayer 对象 pause()方法没有影响。

（7）调用 stop()方法会停止播放，并且还会让一个处于 Started，Paused，Prepared 或 Playback Completed 状态的 MediaPlayer 进入 Stopped 状态。对一个已经处于 Stopped 状态的 MediaPlayer 对象 stop()方法没有影响。

（8）调用 seekTo()方法可以调整播放的位置。方法 seekTo(int)是异步执行的，所以它可以马

上返回，但是实际的定位播放操作可能需要一段时间才能完成，尤其是播放流形式的音频/视频。当实际的定位播放操作完成之后，内部的播放引擎会调用客户端程序员提供的 OnSeekComplete.onSeekComplete()回调方法。可以通过 setOnSeekCompleteListener(OnSeekCompleteListener)方法注册。

我们在此需要注意，seekTo(int)方法也可以在其他状态下调用，比如 Prepared、Paused 和 PlaybackCompleted 状态。此外，目前的播放位置，实际可以调用 getCurrentPosition()方法得到，它可以帮助如音乐播放器的应用程序不断更新播放进度。

（9）当播放到流的末尾时完成播放。如果调用了 setLooping(boolean)方法开启了循环模式，那么这个 MediaPlayer 对象会重新进入 Started 状态。如果没有开启循环模式，那么内部的播放引擎会调用客户端程序员提供的 OnCompletion.onCompletion()回调方法。可以通过调用 MediaPlayer.setOnCompletionListener(OnCompletionListener) 方法来设置。内部的播放引擎一旦调用了 OnCompletion.onCompletion()回调方法，说明这个 MediaPlayer 对象进入了 PlaybackCompleted 状态。当处于 PlaybackCompleted 状态的时候，可以再调用 start()方法让这个 MediaPlayer 对象再进入 Started 状态。

2. MediaPlayer 方法的有效状态和无效状态

- getCurrentPosition {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted} {Error}：在有效状态成功呼叫该方法不会改变此时的状态，在无效的状态呼叫该方法则会使该状态转换到错误状态中；
- getDuration {Prepared, Started, Paused, Stopped, PlaybackCompleted} {Idle, Initialized, Error}：在有效状态下成功呼叫该方法不会改变此时的状态，在无效的状态下呼叫该方法则会使该状态转换到错误状态中；
- getVideoHeight {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted} {Error}：在有效状态下成功呼叫该方法不会改变此时的状态，在无效的状态下呼叫该方法则会使该状态转换到错误状态中；
- getVideoWidth {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted} {Error}：在有效状态下成功呼叫该方法不会改变此时的状态，在无效的状态下呼叫该方法则会使该状态转换到错误状态中；
- isPlaying {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted} {Error}：在有效状态下成功呼叫该方法不会改变此时的状态，在无效的状态下呼叫该方法则会使该状态转换到错误状态中；
- pause {Started, Paused} {Idle, Initialized, Prepared, Stopped, PlaybackCompleted, Error}：在有效状态下成功呼叫该方法改变此时的状态到暂停状态，在无效的状态下呼叫该方法则会使该状态转换到错误状态中；
- prepare {Initialized, Stopped} {Idle, Prepared, Started, Paused, PlaybackCompleted, Error}：在有效状态下成功呼叫该方法改变此时的状态到准备状态，在无效的状态下呼叫该方法则会抛出错误状态异常；
- prepareAsync {Initialized, Stopped} {Idle, Prepared, Started, Paused, PlaybackCompleted, Error}：在有效状态下成功呼叫该方法改变此时的状态到准备状态，在无效的状态下呼叫该方法则会抛出错误状态异常；
- release any {}：在 release()后该对象不再是可用的；
- reset {Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted, Error} {}：在 reset()后该对象如刚创建的一样；

- seekTo {Prepared, Started, Paused, PlaybackCompleted} {Idle, Initialized, Stopped, Error}：在有效状态下成功呼叫该方法改变此时的状态到暂停状态，在无效的状态下呼叫该方法则会使该状态转换到错误状态中；
- setAudioStreamType {Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted} {Error} 在有效状态下成功呼叫该方法改变此时的状态到暂停状态；
- setDataSource {Idle} {Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted, Error}：在有效状态下成功呼叫该方法改变此时的状态到初始化状态，在无效的状态下呼叫该方法则会抛出错误状态异常；
- setDisplay any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- setLooping {Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted} {Error}：在有效状态下成功呼叫该方法不会改变此时的状态，在无效的状态下呼叫该方法则会使该状态转换到错误状态中；
- isLooping any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- setOnBufferingUpdateListener any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- setOnCompletionListener any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- setOnErrorListener any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- setOnPreparedListener any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- setOnSeekCompleteListener any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- setScreenOnWhilePlaying any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- setVolume {Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted} {Error}：成功调用该方法不会改变当前的状态；
- setWakeMode any {}：在任何状态下都可以呼叫该方法，且不会改变当前对象的状态；
- start {Prepared, Started, Paused, PlaybackCompleted} {Idle, Initialized, Stopped, Error}：在有效状态下成功呼叫该方法会改变此时的状态到开始状态，在无效的状态下呼叫该方法则会转换到错误状态；
- stop {Prepared, Started, Stopped, Paused, PlaybackCompleted} {Idle, Initialized, Error}：在有效状态下成功呼叫该方法会改变此时的状态到停止状态，在无效的状态下呼叫该方法则会转换到错误状态。

3. MediaPlayer 方法的接口

- 接口 MediaPlayer.OnBufferingUpdateListener：定义了唤起指明网络上的媒体资源以缓冲流的形式播放；
- 接口 MediaPlayer.OnCompletionListener：是为当媒体资源的播放完成后被唤起的回放定义的；
- 接口 MediaPlayer.OnErrorListener：定义了当在异步操作的时候（其他错误将会在呼叫方法的时候抛出异常）出现错误后唤起的回放操作；
- 接口 MediaPlayer.OnInfoListener：定义了与一些关于媒体或它的播放的信息以及/或者警告相关的被唤起的回放；
- 接口 MediaPlayer.OnPreparedListener：定义为媒体的资源准备播放的时候唤起回放准备的

操作；

- 接口 MediaPlayer.OnSeekCompleteListener：定义了指明查找操作完成后唤起的回放操作；
- 接口 MediaPlayer.OnVideoSizeChangedListener：定义了当视频大小被首次知晓或更新的时候唤起的回放。

4. MediaPlayer 方法的常量

- int MEDIA_ERROR_NOT_VALID_FOR_PROGRESSIVE_PLAYBACK：视频流及其容器是不支持连续的非处于播放文件内的播放视频序列；
- int MEDIA_ERROR_SERVER_DIED：媒体服务终止；
- int MEDIA_ERROR_UNKNOWN：未指明的媒体播放错误；
- int MEDIA_INFO_BAD_INTERLEAVING：不正确的交叉存储技术意味着媒体被不适当的交叉存储或者根本就没有交叉存储，例子里面有所有的视频和音频例子；
- int MEDIA_INFO_METADATA_UPDATE：一套新的可用的元数据；
- int MEDIA_INFO_NOT_SEEKABLE：媒体位置不可查找；
- int MEDIA_INFO_UNKNOWN：未指明的媒体播放信息；
- int MEDIA_INFO_VIDEO_TRACK_LAGGING：视频对于解码器太复杂以至于不能解码足够快的帧率。

5. MediaPlayer 方法的公共方法

- static MediaPlayer create(Context context, Uri uri)：根据给定的 uri 方便地创建 MediaPlayer 对象的方法。
- static MediaPlayer create(Context context, int resid)：根据给定的资源 id 方便地创建 MediaPlayer 对象的方法。
- static MediaPlayer create(Context context, Uri uri, SurfaceHolder holder)：根据给定的 uri 方便地创建 MediaPlayer 对象的方法。
- int getCurrentPosition()：获得当前播放的位置。
- int getDuration()：获得文件段。
- int getVideoHeight()：获得视频的高度。
- int getVideoWidth()：获得视频的宽度。
- boolean isLooping()：检查 MedioPlayer 处于循环与否。
- boolean isPlaying()：检查 MedioPlayer 是否在播放。
- void pause()：暂停播放。
- void prepare()：让播放器处于准备状态（同步的）。
- void prepareAsync()：让播放器处于准备状态（异步的）。
- void release()：释放与 MediaPlayer 相关的资源。
- void reset()：重置 MediaPlayer 到初始化状态。
- void seekTo(int msec)：搜寻指定的时间位置。
- void setAudioStreamType(int streamtype)：为 MediaPlayer 设定音频流类型。
- void setDataSource(String path)：从指定的装载路径所代表的文件。
- void setDataSource(FileDescriptor fd, long offset, long length)：指定装载 fd 所代表的文件中从 offset 开始、长度为 length 的文件内容。
- void setDataSource(FileDescriptor fd)：设定使用的数据源（filedescriptor）。

- void setDataSource(Context context, Uri uri)：设定一个如 Uri 内容的数据源。
- void setDisplay(SurfaceHolder sh)：设定播放该 Video 的媒体播放器的 SurfaceHolder。
- void setLooping(boolean looping)：设定播放器循环或是不循环。
- void setOnBufferingUpdateListener(MediaPlayer.OnBufferingUpdateListener listener)：注册一个当网络缓冲数据流变化时唤起的播放事件。
- void setOnCompletionListener(MediaPlayer.OnCompletionListener listener)：注册一个当媒体资源在播放的时候到达终点时唤起的播放事件。
- void setOnErrorListener(MediaPlayer.OnErrorListener listener)：注册一个当在异步操作过程中发生错误的时候唤起的播放事件。
- void setOnInfoListener(MediaPlayer.OnInfoListener listener)：注册一个当有信息/警告出现的时候唤起的播放事件。
- void setOnPreparedListener(MediaPlayer.OnPreparedListener listener)：注册一个当媒体资源准备播放时候唤起的播放事件。
- void setOnSeekCompleteListener(MediaPlayer.OnSeekCompleteListener listener)：注册一个当搜寻操作完成后唤起的播放事件。
- void setOnVideoSizeChangedListener(MediaPlayer.OnVideoSizeChangedListener listener)：注册一个当视频大小知晓或更新后唤起的播放事件。
- void setScreenOnWhilePlaying(boolean screenOn)：控制当视频播放发生时是否使用 SurfaceHolder 来保持屏幕。
- void setVolume(float leftVolume, float rightVolume)：设置播放器的音量。
- void setWakeMode(Context context, int mode)：为 MediaPlayer 设置低等级的电源管理状态。
- void start()：开始或恢复播放。
- void stop()：停止播放。

为了节约手机的存储空间，在听音乐时可以从网络中下载的方式播放 MP3。接下来将通过一个具体实例的实现过程，来讲解使用 MediaPlayer 播放网络中 MP3 音频的方法。

题目	目的	源码路径
实例 14-6	使用 MediaPlayer 播放网络中 MP3 音频	\daima\14\MediaPlayer

首先在本实例中插入 4 个按钮，分别用于播放、暂停、重新播放和停止处理。执行后，通过 Runnable 发起运行线程，在线程中远程下载指定的 MP3 文件，是通过网络传输方式下载的。下载完毕后，临时保存到 SD 卡中，这样可以通过 4 个按钮对其进行控制。当程序关闭后，删除 SD 卡中的临时性文件。本实例程序文件 example.java 的具体实现流程如下所示。

（1）定义 currentFilePath 用于记录当前正在播放 MP3 的 URL 地址，定义 currentTempFilePath 表示当前播放 MP3 的路径。其具体实现代码如下所示。

```
/*记录当前正在播放 MP3 的 URL*/
private String currentFilePath = "";
/*当前播放 MP3 的路径*/
private String currentTempFilePath = "";
private String strVideoURL = "";
```

（2）使用 strVideoURL 设置要播放 mp3 文件的网址，并设置透明度。其具体实现代码如下所示。

```
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
```

```
  setContentView(R.layout.main);
  /* mp3 文件不会被下载到 local*/
  strVideoURL = "http://www.lrn.cn/zywh/xyyy/yyxs/200805/W020080505536315331317.mp3";
  mTextView01 = (TextView)findViewById(R.id.myTextView1);
  /*设置透明度*/
  getWindow().setFormat(PixelFormat.TRANSPARENT);
  mPlay = (ImageButton)findViewById(R.id.play);
  mReset = (ImageButton)findViewById(R.id.reset);
  mPause = (ImageButton)findViewById(R.id.pause);
  mStop = (ImageButton)findViewById(R.id.stop);
```

（3）编写单击“播放”按钮所触发的处理事件，具体实现代码如下所示。

```
/* 播放按钮 */
mPlay.setOnClickListener(new ImageButton.OnClickListener()
{
  public void onClick(View view)
  {
    /* 调用播放影片 Function */
    playVideo(strVideoURL);
    mTextView01.setText
    (
      getResources().getText(R.string.str_play).toString()+
      "\n"+ strVideoURL
    );
  }
});
```

（4）编写单击“重播”按钮所触发的处理事件，具体实现代码如下所示。

```
/* 重新播放 */
mReset.setOnClickListener(new ImageButton.OnClickListener()
{
  public void onClick(View view)
  {
    if(bIsReleased == false)
    {
      if (mMediaPlayer01 != null)
      {
        mMediaPlayer01.seekTo(0);
        mTextView01.setText(R.string.str_play);
      }
    }
  }
});
```

（5）编写单击“暂停”按钮所触发的处理事件，具体实现代码如下所示。

```
/* 暂停播放 */
mPause.setOnClickListener(new ImageButton.OnClickListener()
{
  public void onClick(View view)
  {
    if (mMediaPlayer01 != null)
    {
      if(bIsReleased == false)
      {
        if(bIsPaused==false)
        {
          mMediaPlayer01.pause();
          bIsPaused = true;
          mTextView01.setText(R.string.str_pause);
        }
        else if(bIsPaused==true)
        {
          mMediaPlayer01.start();
          bIsPaused = false;
          mTextView01.setText(R.string.str_play);
```

```
                }
            }
        }
    }
});
```

（6）编写单击“停止”按钮所触发的处理事件，具体实现代码如下所示。

```
/*停止*/
mStop.setOnClickListener(new ImageButton.OnClickListener()
{
  public void onClick(View view)
  {
    try
    {
      if (mMediaPlayer01 != null)
      {
        if(bIsReleased==false)
        {
          mMediaPlayer01.seekTo(0);
          mMediaPlayer01.pause();
          //mMediaPlayer01.stop();
          //mMediaPlayer01.release();
          //bIsReleased = true;
          mTextView01.setText(R.string.str_stop);
        }
      }
    }
    catch(Exception e)
    {
      mTextView01.setText(e.toString());
      Log.e(TAG, e.toString());
      e.printStackTrace();
    }
  }
});
}
```

（7）定义方法 playVideo(final String strPath)来播放指定的 MP3，其播放的是存储卡中暂时保存的 MP3 文件，具体实现代码如下所示。

```
private void playVideo(final String strPath)
{
  try
  {
    if (strPath.equals(currentFilePath)&& mMediaPlayer01 != null)
    {
      mMediaPlayer01.start();
      return;
    }
    currentFilePath = strPath;
    mMediaPlayer01 = new MediaPlayer();
    mMediaPlayer01.setAudioStreamType(2);
```

（8）编写 setOnErrorListener 来监听错误处理，具体实现代码如下所示。

```
/*错误事件 */
mMediaPlayer01.setOnErrorListener(new MediaPlayer.OnErrorListener()
{
  @Override
  public boolean onError(MediaPlayer mp, int what, int extra)
  {
    //TODO Auto-generated method stub
    Log.i(TAG, "Error on Listener, what: " + what + "extra: " + extra);
    return false;
  }
});
```

（9）编写 setOnBufferingUpdateListener 来监听 MediaPlayer 缓冲区的更新事件，具体实现代码如下所示。

```
/* 监听 MediaPlayer 缓冲区的更新事件 */
mMediaPlayer01.setOnBufferingUpdateListener(new
MediaPlayer.OnBufferingUpdateListener()
{
  @Override
  public void onBufferingUpdate(MediaPlayer mp, int percent)
  {
    //TODO Auto-generated method stub
    Log.i(TAG, "Update buffer: " + Integer.toString(percent)+ "%");
  }
});
```

（10）编写 setOnCompletionListener 来监听播放完毕所触发的事件，具体实现代码如下所示。

```
/* 播放完毕所触发的事件 */
mMediaPlayer01.setOnCompletionListener(new MediaPlayer.OnCompletionListener()
{
  @Override
  public void onCompletion(MediaPlayer mp)
  {
    //TODO Auto-generated method stub
    //delFile(currentTempFilePath)
    Log.i(TAG,"mMediaPlayer01 Listener Completed");
  }
});
```

（11）编写 setOnPreparedListener 来监听开始阶段的事件，具体实现代码如下所示。

```
/* 监听开始阶段的事件 */
mMediaPlayer01.setOnPreparedListener(new MediaPlayer.OnPreparedListener()
{
  @Override
  public void onPrepared(MediaPlayer mp)
  {
    //TODO Auto-generated method stub
    Log.i(TAG,"Prepared Listener");
  }
});
```

（12）将文件存到 SD 卡后，通过方法 mMediaPlayer01.start()播放 MP3，具体实现代码如下所示。

```
  /* 用 Runnable 确保文件在存储完毕后才开始 */
  Runnable r = new Runnable()
  {
    public void run()
    {
      try
      {
        /* setDataSource 将文件存到 SD 卡   */
        setDataSource(strPath);
        /*  因为线程顺利进行，所以在 setDataSource 后运行 prepare() */
        mMediaPlayer01.prepare();
        Log.i(TAG, "Duration: " + mMediaPlayer01.getDuration());

        /* 开始播放 mp3 */
        mMediaPlayer01.start();
        bIsReleased = false;
      }
      catch (Exception e)
      {
        Log.e(TAG, e.getMessage(), e);
      }
    }
  };
  new Thread(r).start();
}
```

（13）如果有异常则输出提示，具体实现代码如下所示。

```
  catch(Exception e)
  {
    if (mMediaPlayer01 != null)
    {
      /* 线程发生异常则停止播放 */
      mMediaPlayer01.stop();
      mMediaPlayer01.release();
    }
    e.printStackTrace();
  }
}
```

（14）定义函数 setDataSource 用于存储 URL 的 MP3 文件到存储卡。首先判断传入的地址是否为 URL，然后创建 URL 对象和临时文件，具体实现代码如下所示。

```
/* 定义函数用于存储 URL 的 mp3 文件到存储卡 */
private void setDataSource(String strPath) throws Exception
{
  /* 判断传入的地址是否为 URL */
  if (!URLUtil.isNetworkUrl(strPath))
  {
    mMediaPlayer01.setDataSource(strPath);
  }
  else
  {
    if(bIsReleased == false)
    {
      /* 创建 URL 对象 */
      URL myURL = new URL(strPath);
      URLConnection conn = myURL.openConnection();
      conn.connect();

      /* 获取 URLConnection 的 InputStream */
      InputStream is = conn.getInputStream();
      if (is == null)
      {
        throw new RuntimeException("stream is null");
      }
      /* 创建临时文件 */
      File myTempFile = File.createTempFile("yinyue", "."+getFileExtension(strPath));
      currentTempFilePath = myTempFile.getAbsolutePath();
      FileOutputStream fos = new FileOutputStream(myTempFile);
      byte buf[] = new byte[128];
      do
      {
        int numread = is.read(buf);
        if (numread <= 0)
        {
          break;
        }
        fos.write(buf, 0, numread);
      }while (true);

      /*直到 fos 存储完毕，调用 MediaPlayer.setDataSource */
      mMediaPlayer01.setDataSource(currentTempFilePath);
      try
      {
        is.close();
      }
      catch (Exception ex)
      {
        Log.e(TAG, "error: " + ex.getMessage(), ex);
      }
    }
  }
}
```

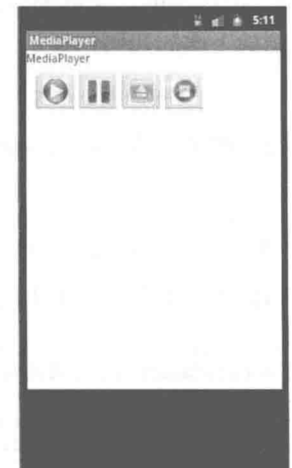
（15）定义方法 getFileExtension(String strFileName)来获取音乐文件的扩展名，如果无法顺利获取扩展名则默认为“.dat”，具体实现代码如下所示。

```
/* 获取音乐文件扩展名自定义函数 */
private String getFileExtension(String strFileName)
{
  File myFile = new File(strFileName);
  String strFileExtension=myFile.getName();
  strFileExtension=(strFileExtension.substring(strFileExtension.lastIndexOf
(".")+1)).toLowerCase();
  if(strFileExtension=="")
  {
    /* 如果无法顺利获取扩展名则默认为.dat */
    strFileExtension = "dat";
  }
  return strFileExtension;
}
```

（16）定义方法 delFile(String strFileName)来设置当离开程序时删除临时音乐文件，具体实现代码如下所示。

```
/* 离开程序时需要调用自定义函数删除临时音乐文件 */
private void delFile(String strFileName)
{
  File myFile = new File(strFileName);
  if(myFile.exists())
  {
    myFile.delete();
  }
}

@Override
protected void onPause()
{
  //TODO Auto-generated method stub

  /* 删除临时文件 */
  try
  {
    delFile(currentTempFilePath);
  }
  catch(Exception e)
  {
    e.printStackTrace();
  }
  super.onPause();
 }
}
```

执行后可以通过播放、暂停、重新播放和停止 4 个按钮来控制指定的 MP3 音乐，执行效果如图 14-8 所示。



▲图 14-8　使用 MediaPlayer 播放网络中的 MP3 音频的执行效果

14.4.3　使用 SoundPool 播放音频

在 Android 系统中，可以使用 SoundPool 播放一些短的反应速度要求高的声音，比如游戏中的爆破声，而 MediaPlayer 适合播放长点的音频。

1. 主要特点

在 Android 系统中，SoundPool 的主要特点如下所示。

（1）SoundPool 使用了独立的线程载入音乐文件，不会阻塞 UI 主线程的操作。但是在这里如果音效文件过大没有载入完成，我们调用 play 方法时可能产生严重的后果。Android SDK 提供了

一个 SoundPool.OnLoadCompleteListener 类来帮助我们了解媒体文件是否载入完成，我们重载 onLoadComplete(SoundPool soundPool, int sampleId, int status) 方法即可获得。

（2）从上面的 onLoadComplete 方法可以看出该类有很多参数，比如类似 id 的 SoundPool，在 load 时可以将多个媒体一次初始化并放入内存中，效率比 MediaPlayer 高了很多。

（3）SoundPool 类支持同时播放多个音效，这对于游戏来说是十分必要的，而 MediaPlayer 类是同步执行的，只能一个文件一个文件地播放。

2. 载入音效的方法

在 SoundPool 中包含了如下 4 个载入音效的方法。

- int load(Context context, int resId, int priority)：从 APK 资源载入。
- int load(FileDescriptor fd, long offset, long length, int priority)：从 FileDescriptor 对象载入。
- int load(AssetFileDescriptor afd, int priority)：从 Asset 对象载入。
- int load(String path, int priority)：从完整文件路径载入。

3. 使用流程

（1）使用 SoundPool 方法创建一个 SoundPool 对象，此方法的格式如下所示。

```
public SoundPool(int maxStream, int streamType, int srcQuality)
```

- maxStream：同时播放的流的最大数量。
- streamType：流的类型，一般为 STREAM_MUSIC，具体内容在 AudioManager 类中列出。
- srcQuality：采样率转化质量，当前无效果，使用 0 作为默认值。

例如下面的代码创建了一个最多支持 3 个流同时播放的、类型标记为音乐的 SoundPool。

```
SoundPool soundPool = new SoundPool(3, AudioManager.STREAM_MUSIC, 0);
```

（2）把多个声音放到 HashMap 中去，例如下面的代码。

```
soundPool = new SoundPool(4, AudioManager.STREAM_MUSIC, 100);
soundPoolMap = new HashMap();
soundPoolMap.put(1, soundPool.load(this, R.raw.dingdong, 1));
```

（3）接下来用下面的方法加载 SoundPool。

```
int load(Context context, int resId, int priority)              //从 APK 资源载入
//从 FileDescriptor 对象载入
int load(FileDescriptor fd, long offset, long length, int priority)
int load(AssetFileDescriptor afd, int priority)                 //从 Asset 对象载入
int load(String path, int priority)                             //从完整文件路径载入
```

其中参数 priority 表示优先级。

（4）使用方法 play()来播放音频，其语法格式如下所示。

```
play(int soundID, float leftVolume, float rightVolume, int priority, int loop, float rate)
```

- leftVolume 和 rightVolume：表示左、右音量。
- priority：表示优先级。
- loop：表示循环次数。
- rate：表示速率，速率最低 0.5，最高为 2，1 代表正常速度。

例如下面的代码。

```
sp.play(soundId, 1, 1, 0, 0, 1);
```

使用 pause(int streamID) 方法停止播放，这里的 streamID 和 soundID 均在构造 SoundPool 类的第一个参数中指明了总数量，而 id 从 0 开始。

注意

SoundPool 的缺陷

（1）SoundPool 最大只能申请 1M 的内存空间，这就意味着我们只能用一些很短的声音片段，而不是用它来播放歌曲或者游戏背景音乐。

（2）SoundPool 提供了 pause 和 stop 方法，但这些方法建议最好不要轻易使用，因为有些时候它们可能会使你的程序莫名其妙地终止。建议使用这两个方法的时候尽可能多做测试工作，还有些朋友反映它们不会立即中止播放声音，而是把缓冲区里的数据播放完才会停下来，也许会多播放 1 秒钟。

（3）SoundPool 的效率问题。其实 SoundPool 的效率在这些播放类中算是很好的了，但是有的朋友在 G1 中测试它还是有 100ms 左右的延迟，这可能会影响用户体验。也许这不能管 SoundPool 本身，因为到了性能比较好的 Droid 中这个延迟就可以让人接受了。

虽然在现阶段 SoundPool 有这些缺陷，但也有着不可替代的优点，基于这些我们建议在如下情况中多使用 SoundPool。

- 应用程序中的声效，例如按键提示音和消息等；
- 游戏中密集而短暂的声音，例如多个飞船同时爆炸。

4. 使用流程

根据上面的使用流程，我们总结出在使用 SoundPool 时的如下编码流程。

（1）在项目中的“res/raw”目录中放入音效素材文件；

（2）新建 SoundPool 对象，然后调用 SoundPool.load()加载音效，调用 SoundPool.play()方法播放指定音效文件。下面的代码是一段标准的格式。

```
public class AudioActivity extends Activity
{
    private SoundPool pool;
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        //指定声音池的最大音频流数目为 10，声音品质为 5
        pool = new SoundPool(10, AudioManager.STREAM_SYSTEM, 5);
        //载入音频流，返回在池中的 id
        final int sourceid = pool.load(this, R.raw.pj, 0);
        Button button = (Button)this.findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener()
        {
            public void onClick(View v)
            {   //播放音频，第二个参数为左声道音量;第三个参数为右声道音量;第四个参数为优先级
                //第五个参数为循环次数，0 不循环，-1 循环
                //第六个参数为速率，速率最低 0.5，最高为 2，1 代表正常速度
                pool.play(sourceid, 1, 1, 0, -1, 1);
            }
        });
    }
}
```

在接下来的内容中，将通过一个具体演示实例来讲解使用 SoundPool 播放指定音频的方法。

题目	目的	源码路径
实例 14-7	使用 SoundPool 播放指定的音频	\daima\14\SoundPoolL

本实例的具体实现流程如下所示。

（1）新建项目“SoundPool”，导入一长一短的两个音乐文件。

（2）在布局文件 main.xml 中设置显示两行文本，主要实现代码如下所示。

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
```

（3）编写主程序文件 MySurfaceView.java，实现界面元素内容的显示，创建 SurfaceView 视图，并监听设备中的按键事件。文件 MySurfaceView.java 的具体实现代码如下所示。

```
/**
 * SurfaceView 初始化函数
 */
    public MySurfaceView(Context context) {
        super(context);
        sfh = this.getHolder();
        sfh.addCallback(this);
        paint = new Paint();
        paint.setColor(Color.WHITE);
        paint.setAntiAlias(true);
        setFocusable(true);
        //实例 SoundPool 播放器
        sp = new SoundPool(4, AudioManager.STREAM_MUSIC, 100);
        //加载音乐文件获取其数据 ID
        soundId_long = sp.load(context, R.raw.sssong, 1);
        //加载音乐文件获取其数据 ID
        soundId_short = sp.load(context, R.raw.ssshort, 1);
    }

/**
 * SurfaceView 视图创建，响应此函数
 */
@Override
public void surfaceCreated(SurfaceHolder holder) {
    screenW = this.getWidth();
    screenH = this.getHeight();
    flag = true;
    //实例线程
    th = new Thread(this);
    //启动线程
    th.start();
}

/**
 * 游戏绘图
 */
public void myDraw() {
    try {
        canvas = sfh.lockCanvas();
        if (canvas != null) {
            canvas.drawColor(Color.WHITE);
            paint.setColor(Color.RED);
            paint.setTextSize(15);
            canvas.drawText("点击导航键的上键：播放短音效", 50, 50, paint);
            canvas.drawText("点击导航键的下键：播放长音效", 50, 80, paint);
        }
    } catch (Exception e) {
        // TODO: handle exception
    } finally {
        if (canvas != null)
            sfh.unlockCanvasAndPost(canvas);
    }
}
```

```
/**
 * 触屏事件监听
 */
@Override
public boolean onTouchEvent(MotionEvent event) {
   return true;
}

/**
 * 按键事件监听
 */
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
   if (keyCode == KeyEvent.KEYCODE_DPAD_UP)
      sp.play(soundId_long, 1f, 1f, 0, 0, 1);
   else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN)
      sp.play(soundId_short, 2, 2, 0, 0, 1);
   return super.onKeyDown(keyCode, event);
}

/**
 * 游戏逻辑
 */
private void logic() {
}

@Override
public void run() {
   while (flag) {
      long start = System.currentTimeMillis();
      myDraw();
      logic();
      long end = System.currentTimeMillis();
      try {
         if (end - start < 50) {
            Thread.sleep(50 - (end - start));
         }
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}

/**
 * SurfaceView 视图状态发生改变，响应此函数
 */
@Override
public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
}

/**
 * SurfaceView 视图消亡时，响应此函数
 */
@Override
public void surfaceDestroyed(SurfaceHolder holder) {
   flag = false;
}
```

执行后会先判断用户按下的按键，根据按键播放不同的音乐文件。执行效果如图 14-9 所示。

点击导航键的上键：播放短音效

点击导航键的下键：播放长音效

▲图 14-9　使用 SoundPool 播放指定音频的执行效果

14.4.4　使用 Ringtone 播放铃声

铃声是手机中的最重要应用之一，在 Android 系统中，通常配合使用 Ringtone 和 RingtoneManager 实现播放铃声、提示音。其中 RingtoneManager 的功能是维护铃声数据库，能够管理来电铃声（TYPE- RINGTONE）、提示音（TYPE NOTIFICATION）、闹钟铃声（TYPE—ALARM）等。在本质上，Ringtone 是对 MediaPlayer 的再一次封装。

在 Android 系统中，通过类 RingtoneManager 专门控制并管理各种铃声。例如常见的来电铃声、闹钟铃声和一些警告、信息通知。类 RingtoneManager 中的常用方法如下所示。

- getActualDefaultRingtoneUri：获取指定类型的当前默认铃声。
- getCursor：返回所有可用铃声的游标。
- getDefaultType：获取指定 URL 默认的铃声类型。
- getDefaultUri：返回指定类型默认铃声的 URL。
- getRingtoneUri：返回指定位置铃声的 URL。
- getRingtonePosition：获取指定铃声的位置。
- getValidRingtoneUri：获取一个可用铃声的位置。
- isDefault：获取指定 URL 是否是默认的铃声。
- setActualDefaultRingtoneUri：设置默认的铃声。

在 Android 系统中，默认的铃声被存储在“system/medio/audio”目录中，下载的铃声一般被保存在 SD 卡中。

接下来将通过一个具体实例的实现过程，讲解使用 RingtoneManager 设置手机铃声的方法。

题目	目的	源码路径
实例 14-8	使用 RingtoneManager 设置手机铃声	\daima\14\ling

编写程序文件 example.java，其具体实现流程如下所示。

（1）分别设置 3 个按钮对象、3 个自定义类型和 3 个铃声文件夹。其具体实现代码如下所示。

```
/* 3个按钮 */
private Button mButtonRingtone;
private Button mButtonAlarm;
private Button mButtonNotification;
/* 3个自定义的类型 */
public static final int ButtonRingtone          = 0;
public static final int ButtonAlarm             = 1;
public static final int ButtonNotification      = 2;
/* 3个铃声文件夹 */
private String strRingtoneFolder = "/sdcard/music/ringtone";
private String strAlarmFolder = "/sdcard/music/alarm";
private String strNotificationFolder = "/sdcard/music/notification";
```

（2）编写单击设置来电铃声按钮 mButtonRingtone 后的处理事件，先打开系统铃声设置，然后进行设置。其具体实现代码如下所示。

```
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    mButtonRingtone = (Button) findViewById(R.id.ButtonRingtone);
    mButtonAlarm = (Button) findViewById(R.id.ButtonAlarm);
    mButtonNotification = (Button) findViewById(R.id.ButtonNotification);
    /* 设置来电铃声 */
    mButtonRingtone.setOnClickListener(new Button.OnClickListener()
    {
        @Override
```

```
        public void onClick(View arg0)
        {
            if (bFolder(strRingtoneFolder))
            {
                //打开系统铃声设置
                Intent intent = new Intent(RingtoneManager.ACTION_RINGTONE_PICKER);
                //类型为来电 RINGTONE
                intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TYPE,
                RingtoneManager.TYPE_RINGTONE);
                //设置显示的 title
                intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TITLE, "设置来电铃声");
                //当设置完成之后返回到当前的 Activity
                startActivityForResult(intent, ButtonRingtone);
            }
        }
    });
```

（3）编写单击设置闹钟铃声按钮 mButtonAlarm 后的处理事件，先打开系统闹钟铃声设置，然后进行设置。其具体实现代码如下所示。

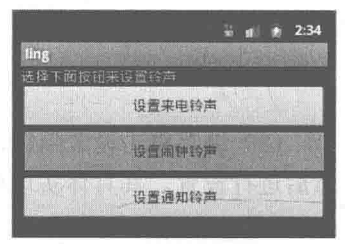
```
/* 设置闹钟铃声 */
mButtonAlarm.setOnClickListener(new Button.OnClickListener()
{
    @Override
    public void onClick(View arg0)
    {
        if (bFolder(strAlarmFolder))
        {
            //打开系统铃声设置
            Intent intent = new Intent(RingtoneManager.ACTION_RINGTONE_PICKER);
            //设置铃声类型和 title
            intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TYPE,
            RingtoneManager.TYPE_ALARM);
            intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TITLE, "设置闹铃铃声");
            //当设置完成之后返回到当前的 Activity
            startActivityForResult(intent, ButtonAlarm);
        }
    }
});
```

（4）编写单击通知铃声按钮 mButtonNotification 的处理事件，先打开系统铃声设置，然后进行设置。其具体实现代码如下所示。

```
    /* 设置通知铃声 */
    mButtonNotification.setOnClickListener(new Button.OnClickListener()
    {
        @Override
        public void onClick(View arg0)
        {
            if (bFolder(strNotificationFolder))
            {
                //打开系统铃声设置
                Intent intent = new Intent(RingtoneManager.ACTION_RINGTONE_PICKER);
                //设置铃声类型和 title
                intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TYPE,
                RingtoneManager.TYPE_NOTIFICATION);
                intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TITLE, "设置通知铃声");
                //当设置完成之后返回到当前的 Activity
                startActivityForResult(intent, ButtonNotification);
            }
        }
    });
}
```

（5）定义方法 boolean bFolder 来检测是否存在指定的文件夹，如果不存在则创建一个。其具体实现代码如下所示。

```
    private boolean bFolder(String strFolder)
    {
        boolean btmp = false;
        File f = new File(strFolder);
        if (!f.exists())
        {
            if (f.mkdirs())
            {
                btmp = true;
            }
            else
            {
                btmp = false;
            }
        }
        else
        {
            btmp = true;
        }
        return btmp;
    }
}
```



▲图 14-10　使用 RingtoneManager 设置手机铃声的执行效果

执行后的可以分别设置 3 种类型的铃声，效果如图 14-10 所示。

在本实例中，会根据音乐文件的位置不同有如下三种不同的声音。

```
/sdcard/ringtones/
/sdcard/media/ringtones/
/sdcard/music/ringtones/
```

14.4.5　使用 JetPlayer 播放音频

在 Android 系统中，还提供了对 Jet 播放的支持。Jet 是由 OHA 联盟成员 SONiVOX 开发的一个交互音乐引擎，包括 Jet 播放器和 Jet 引擎两部分。Jet 常用于控制游戏的声音特效，采用 MIDI（Musical Instrument Digital Interface）格式。

MIDI 数据由一套音乐符号构成，而非实际的音乐。这些音乐符号的一个序列称为 MIDI 消息，Jet 文件包含多个 Jet 段，而每个 Jet 段又包含多个轨迹，一个轨迹是 MIDI 消息的一个序列。

在类 JetPlayer 内部有一个存放 Jet 段的队列，类 JetPlayer 的主要作用是向队列中添加 Jet 段或者清空队列，其次就是控制 Jet 段的轨迹是否处于打开状态。在 Android 系统中，JetPlayer 是基于单子模式（Java 技术中的一种开发模式）实现的，在整个系统中仅存在一个 JetPlayer 的对象。

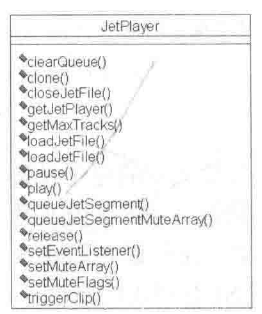
另外，类 JetPlayer 是一个单体类（a singleton class），使用 Static 函数 getJetPlayer()可以获取这个实例。在类 JetPlayer 内部有一个存放 segment 的队列，类 JetPlayer 的主要作用就是向队列中添加 segment 或者清空队列，其次就是控制 Segment 的 Track 是否处于打开状态。

类 JetPlayer 的具体结构如图 14-11 所示。

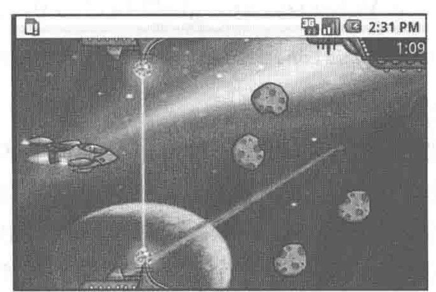
类 JetPlayer 中包含的常用方法如下所示。

- getJetPlayer()：获取 JetPlayer 句柄。
- clearQueue()：清空队列。
- setEventListener()：设置 JetPlayer.OnJetEventListener 监听器。
- loadJetFile：加载 Jet 文件。
- queueJetSegment：查询 Jet 段。
- play()：播放 Jet 文件。
- pause()：暂停播放。

有关类 JetPlayer 的具体用法，读者可以参考 Android SDK 中为我们提供的 JetBoy 游戏源码，游戏界面如图 14-12 所示。



▲图 14-11 类 JetPlayer 的结构



▲图 14-12 JetBoy 游戏界面

14.4.6 使用 AudioEffect 处理音效

从 Android 2.3 开始，对音频播放提供了更强大的音效支持，其实现位于 android.media.audiofx 包中。现在 Android 支持的音效包括：重低音（BassBoost）、环绕音（Virtualizer）、均衡器（Equalizer）、混响（EnvironmentalReverb）和可视化（Visualizer）。

1. AudioEffect 基础

AudioEffect 是 android audio framework（android 音频框架）提供的音频效果控制的基类。开发者不能直接使用此类，应该使用它的派生类。下面列出它的派生类。

- Equalizer；
- Virtualizer；
- BassBoost；
- PresetReverb；
- EnvironmentalReverb。

当创建 AudioEffect 时，如果音频效果应用到一个具体的 AudioTrack 和 MediaPlayer 的实例，应用程序必须指定该实例的音频 session ID，如果要应用 Global 音频输出混响的效果必须制定 Session 0。

要创建音频输出混响（音频 Session 0）要求有 MODIFY_AUDIO_SETTINGS 权限。如果要创建的效果在 audio framework 中不存在，那么直接创建该效果；如果已经存在那么直接使用此效果。如果优先级高的对象要在低级别的对象使用该效果，那么控制将转移到优先级高的对象上，否则继续停留在此对象上。在这种情况下，新的申请将被监听器通知。

（1）重低音。

重低音（BassBoost）通过放大音频中的低频音来实现，其具体细节由 OpenSL ES 定义。为了可以通过 AudioTrack、MediaPlayer 进行音频播放时实现重低音特效，在构建 BassBoost 实例时需要指明音频流的会话 ID。如果指定的会话 ID 为 0，则 BassBoost 作用于主要的音频输出混音器（mix）上，BassBoost 将会话 ID 指定为 0 并需要声明如下权限。

```
android.permission.MODIFY_AUDIO_SETTINGS
```

（2）环绕音。

环绕音（Virtualizer）依赖于输入和输出通道的数量和类型，需要打开立体声通道。通过放置音源于不同的位置，环绕音完美地再现了声音的质感和饱满感。在创建 Virtualizer 实例时，在音频框架层将会同时创建一个环绕音引擎。环绕音的细节由 OpenSL ES 1.0.1 规范定义。

为了在通过 AudioTrack 和 MediaPlayer 播放音频时实现环绕音特效，在构建 Virtualizer 实例时需要指明音频流的会话 ID。如果指定的会话 ID 为 0，则 Virtualizer 作用于主要的音频输出混音

器（mix）上，Virtualizer 将会话 ID 指定为 0，并声明如下所示的权限。

```
android.permission.MODIFY_AUDIO_SETTINGS
```

（3）均衡器。

均衡器（Equalizer）是一种可以分别调节各种频率成分电信号放大量的电子设备，通过对各种不同频率的电信号的调节来补偿扬声器和声场的缺陷，补偿和修饰各种声源及其他特殊作用。一般均衡器仅能对高频、中频、低频 3 段频率电信号分别进行调节。在创建 Equalizer 实例时，在音频框架层将会同时创建一个均衡器引擎。均衡器的细节由 OpenSL ES 1.0.1 规范定义。

为了在通过 AudioTrack、MediaPlayer 进行音频播放时实现均衡器特效，在构建 Equalizer 实例时指明音频流的会话 ID 即可。如果指定的会话 ID 为 0，则 Equalizer 作用于主要的音频输出混音器（mix）上，Equalizer 将会话 ID 指定为 0，需要声明如下所示的权限。

```
android.permission.MODIFY_AUDIO_SETTrNGS
```

（4）混响。

混响（EnvironmentalReverb）即通过声音在不同路径传播下造成的反射叠加产生声音特效。在 Android 平台中，Google 给出了两个实现：EnvironmentalReverb 和 PresetReverb，其中在游戏场景中推荐应用 EnvironmentalReverb，在音乐场景中应用 PresetReverb。在创建混响实例时，在音频框架层将会同时创建一个混响引擎。混响的细节由 OpenSL ES 1.0.1 规范定义。

为了在通过 AudioTrack、MediaPlayer 进行音频播放时实现混响特效，在构建混响实例时指明音频流的会话 ID 即可。如果指定的会话 ID 为 0，则混响作用于主要的音频输出混音器（mix）上，混响将会话 ID 指定为 0，需要声明如下所示的权限。

```
android.permission.MODIFY_AUDIO_SETTINGS
```

（5）可视化。

可视化（Visualizer）分为波形可视化和频率可视化两种情况，在使用可视化时要求声明如下权限。

```
android.permission.RECORD_AUDIO
```

在创建 Visualizer 实例时，同时会在音频框架层将创建一个可视化引擎。为了在通过 AudioTrack、MediaPlayer 进行音频播放时实现可视化特效，在构建 Visualizer 实例时需要指明音频流的会话 ID。如果指定的会话 ID 为 0，则 Visualizer 作用于主要的音频输出混音器（mix）上，Visualizer 将会话 ID 指定为 0，并声明如下权限。

```
android.permission.MODIFY_AUDIO_SETTINGS
```

2. AudioEffect 中的嵌套类

在 AudioEffect 中包含了如下所示的 3 个嵌套类。

- AudioEffect.Descriptor：效果描述符包含在音频框架内实现某种特定的效果的信息。
- AudioEffect.OnControlStatusChangeListener：此接口定义了当应用程序的音频效果的控制状态改变时由 AudioEffect 调用的方法。
- AudioEffect.OnEnableStatusChangeListener：此接口定义了当应用程序的音频效果的启用状态改变时由 AudioEffect 调用的方法。

3. AudioEffect 中的常量

在 AudioEffect 中包含了如下所示常量。

- String ACTION_CLOSE_AUDIO_EFFECT_CONTROL_SESSION：关闭音频效果。
- String ACTION_DISPLAY_AUDIO_EFFECT_CONTROL_PANEL：启动一个音频效果控制面板 UI。
- String ACTION_OPEN_AUDIO_EFFECT_CONTROL_SESSION：打开音频效果。
- int ALREADY_EXISTS：内部操作状态。
- int CONTENT_TYPE_GAME：当播放内容的类型是游戏音频时 EXTRA_CONTENT_TYPE 的值。
- int CONTENT_TYPE_MOVIE：当播放内容的类型是电影时 EXTRA_CONTENT_TYPE 的值。
- int CONTENT_TYPE_MUSIC：当播放内容的类型是音乐时 EXTRA_CONTENT_TYPE 的值。
- int CONTENT_TYPE_VOICE：当播放内容的类型是话音时 EXTRA_CONTENT_TYPE 的值。
- String EFFECT_AUXILIARY：表示 Effect connection mode 是 auxiliary。
- String EFFECT_INSERT：表示 Effect connection mode 是 insert。
- int ERROR：表示操作错误。
- int ERROR_BAD_VALUE：表示由于错误的参数导致的操作失败。
- int ERROR_DEAD_OBJECT：表示由于已关闭的远程对象导致的操作失败。
- int ERROR_INVALID_OPERATION：表示由于错误的请求状态导致的操作失败。
- int ERROR_NO_INIT：表示由于错误的对象初始化导致的操作失败。
- int ERROR_NO_MEMORY：表示由于内存不足导致的操作失败。
- String EXTRA_AUDIO_SESSION：包含使用效果的音频会话 ID。
- String EXTRA_CONTENT_TYPE：表示应用程序播放内容的类型。
- String EXTRA_PACKAGE_NAME：包含调用应用程序的包名。
- int SUCCESS：表示操作成功。

4. AudioEffect 中的公有方法

在 AudioEffect 中常用的公有方法如下所示。

- AudioEffect.Descriptor getDescriptor()：获取效果描述符。
- boolean getEnabled()：返回效果的启用状态。
- int getId()：返回效果的标识符。
- boolean hasControl()：检查该 AudioEffect 对象是否拥有效果引擎的控制。如果有，则返回 true。
- static Descriptor[] queryEffects()：查询平台上的所有有效的音频效果。
- void release()：释放本地 AudioEffect 资源。
- Void setControlStatusListener(AudioEffect.OnControlStatusChangeListener listener)：注册音频效果的控制状态监听器。当控制状态改变时 AudioEffect 发出通知。
- Void setEnableStatusListener(AudioEffect.OnEnableStatusChangeListener listener)：设置音频效果的启用状态监听器。当启用状态改变时 AudioEffect 发出通知。

14.5 语音识别技术

语音识别技术是 Android SDK 中比较重要且比较新颖的一项技术。在本节的内容中，将详细讲解 Android 中语音识别技术的基本知识，为读者进入本书后面知识的学习打下基础。

14.5.1　Text-To-Speech 技术

Text-To-Speech 简称 TTS，是 Android 1.6 版本中比较重要的新功能，其功能是将所指定的文本转成不同语言音频输出。它可以方便地嵌入游戏或者应用程序中，增强用户体验。在讲解 TTS API 及其实际应用方法之前，先初步了解 TTS 引擎。

1. Text-To-Speech 基础

TTS 引擎依托当前 AndroidPlatform 所支持的几种主要的语言：English、French、German、Italian 和 Spanish 五大语言（暂时没有我们伟大的中文，至少 Google 的科学家们还没有把中文玩到炉火纯青的地步，先易后难也是理所当然）。TTS 可以将文本随意转换成以上任意五种语言的语音输出。与此同时，对于个别的语言版本将取决于不同的时区，例如对于 English，在 TTS 中可以分别输出美式和英式两种不同的版本。

既然能支持如此庞大的数据量，TTS 引擎对于资源的优化采取预加载的方法。根据一系列的参数信息从库中提取相应的资源，并加载到当前系统中。尽管当前大部分加载有 Android 操作系统的设备都通过这套引擎来提供 TTS 功能，但由于一些设备的存储空间非常有限，而使 TTS 无法最大限度地发挥功能，这是当前的一个瓶颈。为此开发小组引入了检测模块，让利用这项技术的应用程序或者游戏针对不同的设备可以有相应的优化调整，从而避免由于此项功能的限制而影响到整个应用程序的使用。比较稳妥的做法是让用户自行选择是否有足够的空间或者需求来加载此项资源，下边给出了一个标准的检测方法。

```
Intent checkIntent = new Intent();
checkIntent.setAction(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
startActivityForResult(checkIntent, MY_DATA_CHECK_CODE);
```

如果当前系统允许创建一个“android.speech.tts.TextToSpeech”的 Object 对象，则说明已经提供 TTS 功能的支持，在检测返回结果中给出“CHECK_VOICE_DATA_PASS”的标记。如果系统不支持这项功能，那么用户可以选择是否加载这项功能，从而让设备支持输出多国语言的语音功能“Multi-lingual Talking”。“ACTION_INSTALL_TTS_DATA”将用户引入 Android market 中的 TTS 下载界面。下载完成后将自动完成安装，下边是实现上述过程的完整代码。

```
private TextToSpeech mTts;
protected void onActivityResult(
        int requestCode, int resultCode, Intent data) {
    if (requestCode == MY_DATA_CHECK_CODE) {
        if (resultCode == TextToSpeech.Engine.CHECK_VOICE_DATA_PASS) {
            // success, create the TTS instance
            mTts = new TextToSpeech(this, this);
        } else {
            // missing data, install it
            Intent installIntent = new Intent();
           installIntent.setAction(
                TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
            startActivity(installIntent);
        }
    }
}
```

TextToSpeech 实体和 OnInitListener 都需要引用当前 Activity 的 Context 作为构造参数。OnInitListener()的用处是通知系统当前 TTS 引擎已经加载完成，并处于可用状态。

2. Text-To-Speech 的实现流程

（1）首先检查 TTS 数据是否已经安装并且可用，例如下面的代码。

```
view plaincopy to clipboardprint?
  //检查 TTS 数据是否已经安装并且可用
        Intent checkIntent = new Intent();
        checkIntent.setAction(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
        startActivityForResult(checkIntent, REQ_TTS_STATUS_CHECK);
protected  void onActivityResult(int requestCode, int resultCode, Intent data) {
        if(requestCode == REQ_TTS_STATUS_CHECK)
        {
            switch (resultCode) {
            case TextToSpeech.Engine.CHECK_VOICE_DATA_PASS:
                //这个返回结果表明 TTS 引擎可以用
            {
                mTts = new TextToSpeech(this, this);
                Log.v(TAG, "TTS Engine is installed!");
            }
                break;
            case TextToSpeech.Engine.CHECK_VOICE_DATA_BAD_DATA:
                //需要的语音数据已损坏
            case TextToSpeech.Engine.CHECK_VOICE_DATA_MISSING_DATA:
                //缺少需要语言的语音数据
            case TextToSpeech.Engine.CHECK_VOICE_DATA_MISSING_VOLUME:
                //缺少需要语言的发音数据
            {
                //这三种情况都表明数据有错,重新下载安装需要的数据
                Log.v(TAG, "Need language stuff:"+resultCode);
                Intent dataIntent = new Intent();
                dataIntent.setAction(TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
                startActivity(dataIntent);

            }
                break;
            case TextToSpeech.Engine.CHECK_VOICE_DATA_FAIL:
                //检查失败
            default:
                Log.v(TAG, "Got a failure. TTS apparently not available");
                break;
            }
        }
        else
        {
            //其他 Intent 返回的结果
        }
        }
```

（2）然后初始化 TTS，例如下面的代码。

```
view plaincopy to clipboardprint?
//实现 TTS 初始化接口
    @Override
    public void onInit(int status) {
        // TODO Auto-generated method stub
        //TTS 引擎初始化完成
        if(status == TextToSpeech.SUCCESS)
        {
            int result = mTts.setLanguage(Locale.US);
            //设置发音语言
            if(result == TextToSpeech.LANG_MISSING_DATA || result == TextToSpeech.
            LANG_NOT_SUPPORTED)
            //判断语言是否可用
            {
                Log.v(TAG, "Language is not available");
                speakBtn.setEnabled(false);
            }
            else
            {
    mTts.speak("This is an example of speech synthesis.", TextToSpeech.QUEUE_ADD, null);
                speakBtn.setEnabled(true);
            }
        }
    }
```

（3）接下来需要设置发音语言，例如下面的代码。

```
view plaincopy to clipboardprint?
public void onItemSelected(AdapterView<?> parent, View view,
        int position, long id) {
    // TODO Auto-generated method stub
    int pos = langSelect.getSelectedItemPosition();
    int result = -1;
    switch (pos) {
    case 0:
    {
        inputText.setText("I love you");
        result = mTts.setLanguage(Locale.US);
    }
        break;
    case 1:
    {
        inputText.setText("Je t'aime");
        result = mTts.setLanguage(Locale.FRENCH);
    }
        break;
    case 2:
    {
        inputText.setText("Ich liebe dich");
        result = mTts.setLanguage(Locale.GERMAN);
    }
        break;
    case 3:
    {
        inputText.setText("Ti amo");
        result = mTts.setLanguage(Locale.ITALIAN);
    }
        break;
    case 4:
    {
        inputText.setText("Te quiero");
        result = mTts.setLanguage(new Locale("spa", "ESP"));
    }
        break;
    default:
        break;
    }
    //设置发音语言
    if(result == TextToSpeech.LANG_MISSING_DATA || result == TextToSpeech.LANG_NOT_
    SUPPORTED)
    //判断语言是否可用
    {
        Log.v(TAG, "Language is not available");
        speakBtn.setEnabled(false);
    }
    else
    {
        speakBtn.setEnabled(true);
    }
}
```

（4）最近设置单击按钮发出声音，例如下面的代码。

```
view plaincopy to clipboardprint?
public void onClick(View v) {
    //朗读输入框里的内容
    mTts.speak(inputText.getText().toString(), TextToSpeech.QUEUE_ADD, null);
}
```

14.5.2　谷歌的 Voice Recognition 技术

我们知道苹果的 iPhone 语音识别用的是 Google 的技术，作为 Google 力推的 Android 自然会将其核心技术植入到 Android 系统里面，并结合 Google 的云端技术将其发扬光大。所以 Google

Voice Recognition 在 Android 的实现就变得极其轻松，在它自带的 API 例子是通过一个 Intent 的 Action 来完成的。主要有以下 2 种模式。

- ACTION_RECOGNIZE_SPEECH：一般语音识别，在这种模式下我们可以捕捉到语音的处理后的文字列；
- ACTION_WEB_SEARCH：网络搜索。

在 Api Demo 源码中为我们提供了语音识别实例，具体实现代码如下所示。

```
package com.example.android.apis.app;

import com.example.android.apis.R;

import android.app.Activity;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.os.Bundle;
import android.speech.RecognizerIntent;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.ListView;

import java.util.ArrayList;
import java.util.List;

/**
 *用 API 开发的抽象语音识别代码
 */
public class VoiceRecognition extends Activity implements OnClickListener {

   private static final int VOICE_RECOGNITION_REQUEST_CODE = 1234;

   private ListView mList;

   /**
    *呼叫与活动首先被创造
    */
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      //从它的 XML 布局描述的 UI
      setContentView(R.layout.voice_recognition);

      //得到最新互作用的显示项目
      Button speakButton = (Button) findViewById(R.id.btn_speak);
      mList = (ListView) findViewById(R.id.list);

      //检查公认活动是否存在
      PackageManager pm = getPackageManager();
      List<ResolveInfo> activities = pm.queryIntentActivities(
            new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH), 0);
      if (activities.size() != 0) {
         speakButton.setOnClickListener(this);
      } else {
         speakButton.setEnabled(false);
         speakButton.setText("Recognizer not present");
      }
   }

   /**
    *单击“开始识别按钮”后的处理事件
    */
   public void onClick(View v) {
      if (v.getId() == R.id.btn_speak) {
         startVoiceRecognitionActivity();
```

```
        }
    }
    /**
     *发送开始语音识别信号
     */
    private void startVoiceRecognitionActivity() {
        Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
                RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
        intent.putExtra(RecognizerIntent.EXTRA_PROMPT, "Speech recognition demo");
        startActivityForResult(intent, VOICE_RECOGNITION_REQUEST_CODE);
    }

    /**
     *处理识别结果
     */
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data)
    {
        if (requestCode == VOICE_RECOGNITION_REQUEST_CODE && resultCode == RESULT_OK)
        {
            // Fill the list view with the strings the recognizer thought it could have heard
            ArrayList<String> matches = data.getStringArrayListExtra(
                    RecognizerIntent.EXTRA_RESULTS);
            mList.setAdapter(new  ArrayAdapter<String>(this,  android.R.layout.simple_
list_item_1,
                    matches));
        }
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

上述代码保存在 Google 的 API 开源文件中，原理和实现代码十分简单，感兴趣的读者可以学习一下。上述源码执行后，用户通过单击“Speak!”按钮显示界面如图 14-13 所示；用户说完话后将提交到云端搜索，如图 14-14 所示；在云端搜索完成后将返回打印数据，如图 14-15 所示。

▲图 14-13 单击按钮后

▲图 14-14 说完后

▲图 14-15 返回打印数据

14.6 实现振动效果

无论是智能手机还是普通手机，几乎每一款手机都具备振动功能。在 Android 系统中，同样也可以实现振动效果。在本节的内容中，将详细讲解在 Android 系统中实现振动功能的基本流程，为读者步入本书后面知识的学习打下基础。

14.6.1 Vibrator 类基础

Android 系统中的振动功能是通过类 Vibrator 实现的，读者可以在 SDK 中的 android.os.Vibrator 找到相关的描述。从 1.0 开始改进了一些声明方式，在实例化的同时去除了构造方法 new Vibrator()，调用时必需获取振动服务的实例句柄。我们选定一个 Vibrator 对象 mVibrator 变量，获取的方法很简单，具体代码如下所示。

```
mVibrator = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);
```

然后直接调用下面的方法。

```
vibrate(long[] pattern, int repeat)
```

- 第一个参数 long[] pattern：是一个节奏数组，比如{1, 200}；
- 第二个参数 repeat：是重复次数，-1 为不重复，而数字直接表示的是具体的数字，和一般-1 表示无限不同。

在使用振动功能之前，需要先在 manifest 中加入下面的权限。

```
<uses-permission android:name="android.permission.VIBRATE"/>
```

在设置振动（Vibration）事件时，必须要知道命令其振动的时间长短、振动事件的周期等。因为在 Android 里设置的数值，都是以毫秒（1 000 毫秒=1 秒）来做计算，所以在设置时，必须要注意设置时间的长短，如果设置的时间值太小的话，会感觉不出来。

要让手机乖乖地振动，需创建 Vibrator 对象，通过调用 vibrate 方法来达到振动的目的。在 Vibrator 的构造器中有 4 个参数，前 3 个的值是设置振动的大小，在这边可以把数值改成一大一小，这样就可以明显感觉出振动的差异，而最后一个值是设置振动的时间。

根据个人开发经验，笔者总结出在 Android 系统上开发振动系统的基本流程，如下所示。

（1）在 manifest 文件中声明振动权限。

（2）通过系统服务获得手机振动服务，例如下面的代码。

```
Vibrator vibrator = (Vibrator)getSystemService(VIBRATOR_SERVICE);
```

（3）得到振动服务后检测 vibrator 是否存在，例如下面的代码。

```
vibrator.hasVibrator();
```

通过上述代码可以检测当前硬件是否有 vibrator，如果有则返回 true，如果没有则返回 false。

（4）根据实际需要进行适当的调用，例如下面的代码。

```
vibrator.vibrate(long milliseconds);
```

通过上述代码开始启动 vibrator 持续 milliseconds 毫秒。

（5）编写下面的代码。

```
vibrator.vibrate(long[] pattern, int repeat);
```

这样以 pattern 方式重复 repeat 次启动 vibrator。pattern 的形式如下所示。

```
new  long[]{arg1,arg2,arg3,arg4......}
```

在上述格式中，以两个一组，如 arg1 和 arg2 为一组、arg3 和 arg4 为一组，每一组的前一个代表等待多少毫秒启动 vibrator，后一个代表 vibrator 持续多少毫秒停止，之后往复即可。repeat 表示重复次数，当其为-1 时，表示不重复，只以 pattern 的方式运行一次。

（6）停止振动，具体代码如下所示。

```
vibrator.cancel();
```

14.6.2 使用 Vibrator 实现振动效果

在接下来的内容中，将通过一个具体演示实例讲解使用 Vibrator 实现振动效果的方法。本实例的功能是，机背面朝上时自动启动振动模式。通过 Android 系统中的 API，可以判断手机倾斜、

旋转等模式。通过 BroadcastReceiver 对象来聆听系统广播短信或 PhoneState Listener 对象，聆听系统广播的电话事件等。Android 系统的 Sensor.Manager 事件是使用 Sensor 对象实现的。为了让 Activity 程序在 onCreate()后的第一个进入点（onResume()方法）就能监视手机状态，所以在 onResume()方法里创建 IntentFilter，使用 SensorListener.registerListener()注册一个自定义的 SensorListener，在 onPause()离开程序时取消系统注册（unregisterLister）SensorListener。

因为只判断手机的倾斜或旋转状态并不够实用，所以在本实例中联合使用了 SensorListener 和 AudioManager。当程序发现手机被翻成背面时，就会将铃声模式更改为振动模式。

在接下来的内容中，将通过一个具体实例来讲解使用 RingtoneManager 设置手机铃声的方法。

题目	目的	源码路径
实例 14-9	使用 RingtoneManager 设置手机铃声	\daima\14\lzhendong

本实例的具体实现流程如下所示。

（1）编写实现文件 example.java，在此文件中注册了 SensorListener 的 registerListener0 方法，使 Activity 程序能够捕捉到 Sensor 的变化。在捕捉变化时需要传入如下 3 个参数。

- mSensorListener：SensorListener 对象，为 Activity 类成员，通过覆盖 onSensorChanged()方法作为判断。
- SensorManager.SENSOR_ORIENTATION：欲捕捉的 Sensor 事件常数。
- SensorManager.SENSOR_DELAY NORMAL：状态更改的精准度常数。

文件 example.java 的主要实现代码如下所示。

```
public class example extends Activity
{
  /* 创建 SensorManager 对象 */
  private SensorManager mSensorManager01;
  private TextView mTextView01;

  /* 以私有类成员存储 AudioManager 模式 */
  private int strRingerMode;

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mTextView01 = (TextView)findViewById(R.id.myTextView1);

    /* 创建 SensorManager 对象，取得 SENSOR_SERVICE 服务 */
    mSensorManager01 =
    (SensorManager)getSystemService(Context.SENSOR_SERVICE);

    /* 取得现在的 AudioManager 模式 */
    GetAudioManagerMode();

    /* 依据现在的 AudioManager 模式，显示于 TextView 当中 */
    switch(strRingerMode)
    {
      /* 正常模式 */
      case AudioManager.RINGER_MODE_NORMAL:
        mTextView01.setText(R.string.str_normal_mode);
        break;
      /* 静音模式 */
      case AudioManager.RINGER_MODE_SILENT:
        mTextView01.setText(R.string.str_silent_mode);
        break;
      /* 振动模式 */
```

```
    case AudioManager.RINGER_MODE_VIBRATE:
      mTextView01.setText(R.string.str_vibrate_mode);
      break;
  }
}

/* 创建 SensorListener 捕捉 onSensorChanged 事件 */
private final SensorListener mSensorListener =
          new SensorListener()
{
  @Override
  public void onSensorChanged(int sensor, float[] values)
  {
    // TODO Auto-generated method stub

    //float fRollAngle = values[SensorManager.DATA_X];

    /* 取得 y 平面左右倾斜的 Pitch 角度 */
    float fPitchAngle = values[SensorManager.DATA_Y];

    /* 正面向下(y 轴旋转)，经实验结果为小于-120 为翻背面 */
    if(fPitchAngle<-120)
    {
      /* 先设置为静音模式 */
      ChangeToSilentMode();

      /* 再设置为振动模式 */
      ChangeToVibrateMode();

      /* 判断铃声模式 */
      switch(strRingerMode)
      {
        /* 正常模式 */
        case AudioManager.RINGER_MODE_NORMAL:
          mTextView01.setText(R.string.str_normal_mode);
          break;
        /* 静音模式 */
        case AudioManager.RINGER_MODE_SILENT:
          mTextView01.setText(R.string.str_silent_mode);
          break;
        /* 振动模式 */
        case AudioManager.RINGER_MODE_VIBRATE:
          mTextView01.setText(R.string.str_vibrate_mode);
          break;
      }
    }
    else
    {
      /* 正面向上(y 轴旋转)，经实验结果为大于-120 为翻正面 */
      /* 更改为正常模式 */
      ChangeToNormalMode();

      /* 调用更改模式后，再一次确认手机的模式 */
      switch(strRingerMode)
      {
        case AudioManager.RINGER_MODE_NORMAL:
          mTextView01.setText(R.string.str_normal_mode);
          break;
        case AudioManager.RINGER_MODE_SILENT:
          mTextView01.setText(R.string.str_silent_mode);
          break;
        case AudioManager.RINGER_MODE_VIBRATE:
          mTextView01.setText(R.string.str_vibrate_mode);
          break;
      }
    }
  }

  @Override
```

```
    public void onAccuracyChanged(int sensor, int values)
    {
      // TODO Auto-generated method stub

    }
  };

  /* 取得当下的 AudioManager 模式 */
  private void GetAudioManagerMode()
  {
    try
    {
      /* 创建 AudioManager 对象，取得 AUDIO_SERVICE */
      AudioManager audioManager =
      (AudioManager)getSystemService(Context.AUDIO_SERVICE);

      if (audioManager != null)
      {
        /* RINGER_MODE_NORMAL |
           RINGER_MODE_SILENT |
           RINGER_MODE_VIBRATE */

        strRingerMode = audioManager.getRingerMode();
      }
    }
    catch(Exception e)
    {
      mTextView01.setText(e.toString());
      e.printStackTrace();
    }
  }

  /* 更改为静音模式 */
  private void ChangeToSilentMode()
  {
    try
    {
      AudioManager audioManager =
      (AudioManager)getSystemService(Context.AUDIO_SERVICE);

      if (audioManager != null)
      {
        /* RINGER_MODE_NORMAL |
           RINGER_MODE_SILENT |
           RINGER_MODE_VIBRATE */

        audioManager.setRingerMode(AudioManager.RINGER_MODE_SILENT);
        strRingerMode = audioManager.getRingerMode();
      }
    }
    catch(Exception e)
    {
      mTextView01.setText(e.toString());
      e.printStackTrace();
    }
  }

  /* 更改为振动模式 */
  private void ChangeToVibrateMode()
  {
    try
    {
      AudioManager audioManager =
      (AudioManager)getSystemService(Context.AUDIO_SERVICE);

      if (audioManager != null)
      {
        /* 调用 setRingerMode 方法，设置模式 */
        audioManager.setRingerMode
```

```
        (
          AudioManager.RINGER_MODE_VIBRATE
        );
        /* RINGER_MODE_NORMAL |
           RINGER_MODE_SILENT |
           RINGER_MODE_VIBRATE */
        strRingerMode = audioManager.getRingerMode();
      }
    }
    catch(Exception e)
    {
      mTextView01.setText(e.toString());
      e.printStackTrace();
    }
  }

  /* 更改为正常模式 */
  private void ChangeToNormalMode()
  {
    try
    {
      AudioManager audioManager =
      (AudioManager)getSystemService(Context.AUDIO_SERVICE);

      if (audioManager != null)
      {
        /* RINGER_MODE_NORMAL |
           RINGER_MODE_SILENT |
           RINGER_MODE_VIBRATE */
        audioManager.setRingerMode(AudioManager.RINGER_MODE_NORMAL);
        strRingerMode = audioManager.getRingerMode();
      }
    }
    catch(Exception e)
    {
      mTextView01.setText(e.toString());
      e.printStackTrace();
    }
  }

  @Override
  protected void onResume()
  {
    // TODO Auto-generated method stub

    /* 注册一个 SensorListener 的 Listener */
    /* 传入 Sensor 模式与 rate */
    mSensorManager01.registerListener
    (
      mSensorListener,
      SensorManager.SENSOR_ORIENTATION,
      SensorManager.SENSOR_DELAY_NORMAL
    );
    super.onResume();
  }

  @Override
  protected void onPause()
  {
    // TODO Auto-generated method stub

    /* 覆盖 onPause 事件，取消 mSensorListener */
    mSensorManager01.unregisterListener(mSensorListener);
    super.onPause();
  }
}
```

（2）编写文件 AndroidManifest.xml，在此声明 Android.permission.VIBRATE 权限，主要代码

如下所示。

```
<uses-permission android:name="android.permission.VIBRATE"></uses-permission>
```

执行后的效果如图 14-16 所示，如果将手机反转则会自动进入振动模式。

▲图 14-16　使用 RingtoneManager 设置手机铃声的执行效果

14.7 设置闹钟

在 Android 系统中，可以使用 AlarmManage 来设置闹钟。在本节的内容中，将详细讲解在 Android 系统中设置闹钟的基本知识，为读者进入本书后面知识的学习打下基础。

14.7.1 AlarmManage 基础

在 Android 系统中，对应 AlarmManage 有一个 AlarmManagerServie 服务程序，该服务程序才是真正提供闹铃服务的，它主要维护应用程序注册下来的各类闹铃并适时的设置即将触发的闹铃给闹铃设备。在系统中，Linux 实现的设备名为"/dev/alarm"，并且一直监听闹铃设备，一旦有闹铃触发或者是闹铃事件发生，AlarmManagerServie 服务程序就会遍历闹铃列表找到相应的注册闹铃并发出广播。该服务程序在系统启动时被系统服务程序 System_service 启动并初始化闹铃设备(/dev/alarm)。当然，在 Java 层的 AlarmManagerService 与 Linux Alarm 驱动程序接口之间还有一层封装，那就是 JNI。

AlarmManager 将应用与服务分割开后，使得应用程序开发者不用关心具体的服务，而是直接通过 AlarmManager 使用这种服务。这也许就是客户/服务器模式的好处吧。AlarmManager 与 AlarmManagerServie 之间是通过 Binder 通信的，它们之间是多对一的关系。

在 Android 系统中，AlarmManage 提供了 3 个接口 5 种类型的闹铃服务。其中 3 个接口如下所示。

```
// 取消已经注册的与参数匹配的闹铃
void    cancel(PendingIntent operation)
  //注册一个新的闹铃
void    set( int  type,  long  triggerAtTime, PendingIntent operation)
  //注册一个重复类型的闹铃
void    setRepeating( int  type,  long  triggerAtTime,  long  interval, PendingIntent
operation)
//设置时区
void    setTimeZone(String timeZone)
```

5 个闹铃类型如下所示。

```
public static final int ELAPSED_REALTIME
//当系统进入睡眠状态时，这种类型的闹铃不会唤醒系统。直到系统下次被唤醒才传递它，该闹铃所用的时间是相
//对时间，是从系统启动后开始计时的,包括睡眠时间，可以通过调用 SystemClock.elapsedRealtime()获得
//系统值是 3(0x00000003)
        public static final int ELAPSED_REALTIME_WAKEUP
        //能唤醒系统，用法同 ELAPSED_REALTIME，系统值是 2(0x00000002)
        public static final int RTC

//当系统进入睡眠状态时，这种类型的闹铃不会唤醒系统。直到系统下次被唤醒才传递它，该闹铃所用的时间是绝
```

```
//对时间,所用时间是 UTC 时间,可以通过调用 System.currentTimeMillis()获得。系统值是 1(0x00000001)
      public static final int RTC_WAKEUP
      //能唤醒系统，用法同 RTC 类型，系统值为 0(0x00000000)
      Public static final int POWER_OFF_WAKEUP

//能唤醒系统，它是一种关机闹铃，就是说设备在关机状态下也可以唤醒系统，所以我们把它称之为关机闹铃
//使用方法同 RTC 类型，系统值为 4(0x00000004)
```

14.7.2 开发一个闹钟程序

在接下来的内容中，将通过一个具体演示实例讲解使用 AlarmManage 实现闹钟功能的方法。

题目	目的	源码路径
实例 14-10	使用 AlarmManage 实现闹钟功能	\daima\14\naozhong

本实例的具体实现流程如下所示。

（1）编写文件 example.java，其具体实现流程如下所示。

- 载入主布局文件 main.xml，单击 Button1 按钮后实现只响一次闹钟，通过 setTime1 对象实现只响一次的闹钟的设置，具体实现代码如下所示。

```
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  /* 载入 main.xml Layout */
  setContentView(R.layout.main);
  /* 以下为只响一次的闹钟的设置 */
  setTime1=(TextView) findViewById(R.id.setTime1);
  /* 只响一次的闹钟的设置 Button */
  mButton1=(Button)findViewById(R.id.mButton1);
  mButton1.setOnClickListener(new View.OnClickListener()
  {
    public void onClick(View v)
    {
      /* 取得单击按钮时的时间作为 TimePickerDialog 的默认值 */
      c.setTimeInMillis(System.currentTimeMillis());
      int mHour=c.get(Calendar.HOUR_OF_DAY);
      int mMinute=c.get(Calendar.MINUTE);
```

- 通过 TimePickerDialog 弹出一个对话框供用户来设置时间，具体实现代码如下所示。

```
    /* 跳出 TimePickerDialog 来设置时间 */
    new TimePickerDialog(example9.this,
      new TimePickerDialog.OnTimeSetListener() {
        public void onTimeSet(TimePicker view,int hourOfDay,int minute)
        {
          /* 取得设置后的时间，秒及毫秒设为 0 */
          c.setTimeInMillis(System.currentTimeMillis());
          c.set(Calendar.HOUR_OF_DAY,hourOfDay);
          c.set(Calendar.MINUTE,minute);
          c.set(Calendar.SECOND,0);
          c.set(Calendar.MILLISECOND,0);
          /* 指定闹钟设置时间到时要运行 CallAlarm.class */
          Intent intent = new Intent(example9.this, example_2.class);
          /* 创建 PendingIntent */
          PendingIntent sender=PendingIntent.getBroadcast(example9.this,0, intent, 0);
/* AlarmManager.RTC_WAKEUP 设置服务在系统休眠时同样会运行，以 set()设置的 PendingIntent 只会运
行一次* */
          AlarmManager am;
          am = (AlarmManager)getSystemService(ALARM_SERVICE);
          am.set(AlarmManager.RTC_WAKEUP,
                 c.getTimeInMillis(),
                 sender
                );
          /* 更新显示的设置闹钟时间 */
```

```
        String tmpS=format(hourOfDay)+": "+format(minute);
        setTime1.setText(tmpS);
        /* 以 Toast 提示设置已完成 */
        Toast.makeText(example9.this,"设置闹钟时间为"+tmpS,
          Toast.LENGTH_SHORT)
          .show();
      }
    },mHour,mMinute,true).show();
  }
});
```

- 单击 mButton2 按钮来删除只响一次的闹钟，具体实现代码如下所示。

```
mButton2=(Button) findViewById(R.id.mButton2);
mButton2.setOnClickListener(new View.OnClickListener()
{
  public void onClick(View v)
  {
    Intent intent = new Intent(example.this, example_2.class);
    PendingIntent sender=PendingIntent.getBroadcast(
                    Example.this,0, intent, 0);
    /* 在 AlarmManager 中删除 */
    AlarmManager am;
    am =(AlarmManager)getSystemService(ALARM_SERVICE);
    am.cancel(sender);
    /* 以 Toast 提示已删除设置，并更新显示的闹钟时间 */
    Toast.makeText(example.this,"闹钟时间解除",
              Toast.LENGTH_SHORT).show();
    setTime1.setText("目前无设置");
  }
});
```

（2）开始设置重复响起的闹钟，具体实现代码如下所示。

```
/* 以下为重复响起的闹钟的设置 */
setTime2=(TextView) findViewById(R.id.setTime2);
/* create 重复响起的闹钟的设置画面 */
/* 引用 timeset.xml 为 Layout */
LayoutInflater factory = LayoutInflater.from(this);
final View setView = factory.inflate(R.layout.timeset,null);
final TimePicker tPicker=(TimePicker)setView
                  .findViewById(R.id.tPicker);
tPicker.setIs24HourView(true);
/* create 重复响起闹钟的设置 Dialog */
final AlertDialog di=new AlertDialog.Builder(example.this)
     .setIcon(R.drawable.clock)
     .setTitle("设置")
     .setView(setView)
     .setPositiveButton("确定",
      new DialogInterface.OnClickListener()
      {
        public void onClick(DialogInterface dialog, int which)
        {
          /* 取得设置的间隔秒数 */
          EditText ed=(EditText)setView.findViewById(R.id.mEdit);
          int times=Integer.parseInt(ed.getText().toString())
                 *1000;
          /* 取得设置的开始时间，秒及毫秒设为 0 */
          c.setTimeInMillis(System.currentTimeMillis());
          c.set(Calendar.HOUR_OF_DAY,tPicker.getCurrentHour());
          c.set(Calendar.MINUTE,tPicker.getCurrentMinute());
          c.set(Calendar.SECOND,0);
          c.set(Calendar.MILLISECOND,0);

          /* 指定闹钟设置时间到时要运行 CallAlarm.class */
          Intent intent = new Intent(example.this,
                          Example_2.class);
          PendingIntent sender = PendingIntent.getBroadcast(
                         Example.this,1, intent, 0);
```

```
        /* setRepeating()可让闹钟重复运行 */
        AlarmManager am;
        am = (AlarmManager)getSystemService(ALARM_SERVICE);
        am.setRepeating(AlarmManager.RTC_WAKEUP,
                c.getTimeInMillis(),times,sender);
        /* 更新显示的设置闹钟时间 */
        String tmpS=format(tPicker.getCurrentHour())+": "+
                  format(tPicker.getCurrentMinute());
        setTime2.setText("设置闹钟时间为"+tmpS+
                      "开始，重复间隔为"+times/1000+"秒");
        /* 以 Toast 提示设置已完成 */
        Toast.makeText(example9.this,"设置闹钟时间为"+tmpS+
                     "开始，重复间隔为"+times/1000+"秒",
                     Toast.LENGTH_SHORT).show();
      }
    })
    .setNegativeButton("取消",
     new DialogInterface.OnClickListener()
    {
      public void onClick(DialogInterface dialog, int which)
      {
      }
    }).create();
```

上述代码的具体实现流程如下所示。

- 以 create 重复响起的闹钟的设置画面，并引用 timeset.xml 为布局文件。
- 以 create 重复响起闹钟的设置 Dialog 对话框。
- 获取设置的间隔秒数。
- 获取设置的开始时间，秒及毫秒都设为 0。
- 指定闹钟设置时间到时要运行 CallAlarm.class。
- 通过 setRepeating()可让闹钟重复运行。
- 通过 dmpS 更新显示的设置闹钟时间。
- 通过以 Toast 提示设置已完成。

- 单击 mButton3 按钮实现重复响起的闹钟，具体实现代码如下所示。

```
/* 重复响起的闹钟的设置 Button */
mButton3=(Button) findViewById(R.id.mButton3);
mButton3.setOnClickListener(new View.OnClickListener()
{
  public void onClick(View v)
  {
    /* 取得单击按钮时的时间作为 tPicker 的默认值 */
    c.setTimeInMillis(System.currentTimeMillis());
    tPicker.setCurrentHour(c.get(Calendar.HOUR_OF_DAY));
    tPicker.setCurrentMinute(c.get(Calendar.MINUTE));
    /* 跳出设置画面 di */
    di.show();
  }
});
```

- 单击 mButton4 按钮删除重复响起的闹钟，具体实现代码如下所示。

```
mButton4=(Button) findViewById(R.id.mButton4);
mButton4.setOnClickListener(new View.OnClickListener()
{
  public void onClick(View v)
  {
    Intent intent = new Intent(example9.this, example9_2.class);
    PendingIntent sender = PendingIntent.getBroadcast(
                     example9.this,1, intent, 0);
    /* 在 AlarmManager 中删除 */
    AlarmManager am;
    am = (AlarmManager)getSystemService(ALARM_SERVICE);
```

```
      am.cancel(sender);
      /* 以 Toast 提示已删除设置，并更新显示的闹钟时间 */
      Toast.makeText(example.this,"闹钟时间解除",
                    Toast.LENGTH_SHORT).show();
      setTime2.setText("目前无设置");
    }
  });
}
```

- 使用方法 format 来设置使用两位数的显示格式来表示日期时间，具体实现代码如下所示。

```
  /* 日期时间显示两位数的方法 */
  private String format(int x)
  {
    String s=""+x;
    if(s.length()==1) s="0"+s;
    return s;
  }
}
```

（3）编写文件 example_1.java，具体实现代码如下所示。

```
/* 实际跳出闹铃 Dialog 的 Activity */
public class example_1 extends Activity
{
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    /* 跳出的闹铃警示  */
    new AlertDialog.Builder(example9_1.this)
        .setIcon(R.drawable.clock)
        .setTitle("闹钟响了!!")
        .setMessage("赶快起床吧!!!")
        .setPositiveButton("关掉它",
         new DialogInterface.OnClickListener()
        {
          public void onClick(DialogInterface dialog, int whichButton)
          {
            /* 关闭 Activity */
            Example_1.this.finish();
          }
        })
        .show();
  }
}
```

（4）编写文件 example_2.java，具体实现代码如下所示。

```
/* 调用闹钟 Alert 的 Receiver */
public class example_2 extends BroadcastReceiver
{
  @Override
  public void onReceive(Context context, Intent intent)
  {
    /* 创建 Intent，调用 AlarmAlert.class */
    Intent i = new Intent(context, example9_1.class);
    Bundle bundleRet = new Bundle();
    bundleRet.putString("STR_CALLER", "");
    i.putExtras(bundleRet);
    i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    context.startActivity(i);
  }
}
```

（5）编写文件 AndroidManifest.xml，在里面添加对 CallAlarm 的 receiver 设置。具体实现代码如下所示。

```
<!—注册 receiver CallAlarm -->
<receiver android:name=".example_2" android:process=":remote" />
<activity android:name=".example_1" ndroid:label="@string/app_name">
</activity>
```

执行后的初始效果如图 14-17 所示。单击第一个“设置”按钮后弹出设置界面，在此可以设置闹钟时间，如图 14-18 所示。单击第二个“设置”按钮可以设置重复响起的时间，如图 14-19 所示。

▲图 14-17 使用 AlarmManager 设置闹钟的初始效果

▲图 14-18 响一次的设置界面

▲图 14-19 重复响起的设置界面

第 15 章　开发视频应用程序

在移动设备应用领域中，视频播放是最主流的多媒体应用之一，用户经常使用手机等移动设备来观看视频节目。在本书前面的内容中，已经讲解了 Android 系统中底层视频系统的知识。在高层的 Java 应用中，可以通过底层提供的接口开发常见的视频应用。在本章的内容中，将详细讲解开发 Android 视频应用的基本知识，为读者进入后面知识的学习打下基础。

15.1 使用 MediaPlayer 播放视频

在本书前面的内容中，已经讲解了 MediaPlayer 的基本知识。其实除了播放音频之外，它还是在 Android 系统中播放视频的主流方法之一。在本书的第 14 章中，已经详细讲解了 MediaPlayer 中的各个方法，在本节的内容中，将通过一个具体实例说明使用 MediaPlayer 播放视频的基本方法。

题目	目的	源码路径
实例 15-1	使用 MediaPlayer 播放网络中的视频	\daima\15\MediaBo

编写主程序文件 example.java，其具体实现流程如下所示。

（1）定义 bIsReleased 来标识 MediaPlayer 是否已被释放，识别 MediaPlayer 是否正处于暂停，并用 LogCat 输出 TAG filter。其具体实现代码如下所示。

```
/* 识别 MediaPlayer 是否已被释放*/
private boolean bIsReleased = false;
/* 识别 MediaPlayer 是否正处于暂停*/
private boolean bIsPaused = false;
/* LogCat 输出 TAG filter */
private static final String TAG = "HippoMediaPlayer";
private String currentFilePath = "";
private String currentTempFilePath = "";
private String strVideoURL = "";
```

（2）设置播放视频的 URL 地址，使用 mSurfaceView01 来绑定 Layout 上的 SurfaceView。然后设置 SurfaceHolder 为 Layout SurfaceView。其具体实现代码如下所示。

```
ublic void onCreate(Bundle savedInstanceState)

 super.onCreate(savedInstanceState);
 setContentView(R.layout.main);
 /* 将.3gp 图像文件存放 URL 网址*/
 strVideoURL =
 "http://new4.sz.3gp2.com//20100205xyy/喜羊羊与灰太狼%20 踩高跷(www.3gp2.com).3gp";
 //http://www.dubblogs.cc:8751/Android/Test/Media/3gp/test2.3gp

 mTextView01 = (TextView)findViewById(R.id.myTextView1);
 mEditText01 = (EditText)findViewById(R.id.myEditText1);
```

```
mEditText01.setText(strVideoURL);

/* 绑定 Layout 上的 SurfaceView */
mSurfaceView01 = (SurfaceView) findViewById(R.id.mSurfaceView1);

/* 设置 PixnelFormat */
getWindow().setFormat(PixelFormat.TRANSPARENT);
/* 设置 SurfaceHolder 为 Layout SurfaceView */
mSurfaceHolder01 = mSurfaceView01.getHolder();
mSurfaceHolder01.addCallback(this);
```

（3）为影片设置大小比例，并分别设置 mPlay、mReset、mPause 和 mStop 这 4 个控制按钮。其具体实现代码如下所示。

```
/* 由于原有的影片 Size 较小，故指定其为固定比例*/
mSurfaceHolder01.setFixedSize(160, 128);
mSurfaceHolder01.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
mPlay = (ImageButton) findViewById(R.id.play);
mReset = (ImageButton) findViewById(R.id.reset);
mPause = (ImageButton) findViewById(R.id.pause);
mStop = (ImageButton) findViewById(R.id.stop);
```

（4）编写单击“播放”按钮的处理事件，具体实现代码如下所示。

```
/* 播放按钮*/
mPlay.setOnClickListener(new ImageButton.OnClickListener()
{
  public void onClick(View view)
  {
    if(checkSDCard())
    {
      strVideoURL = mEditText01.getText().toString();
      playVideo(strVideoURL);
      mTextView01.setText(R.string.str_play);
    }
    else
    {
      mTextView01.setText(R.string.str_err_nosd);
    }
  }
});
```

（5）编写单击“重播”按钮的处理事件，具体实现代码如下所示。

```
* 重新播放按钮*/
Reset.setOnClickListener(new ImageButton.OnClickListener()

 public void onClick(View view)
 {
   if(checkSDCard())
   {
     if(bIsReleased == false)
     {
       if (mMediaPlayer01 != null)
       {
         mMediaPlayer01.seekTo(0);
         mTextView01.setText(R.string.str_play);
       }
     }
   }
   else
   {
     mTextView01.setText(R.string.str_err_nosd);
   }
 }
);
```

（6）编写单击“暂停”按钮的处理事件，具体实现代码如下所示。

```
/* 暂停按钮*/
mPause.setOnClickListener(new ImageButton.OnClickListener()
{
  public void onClick(View view)
  {
    if(checkSDCard())
    {
      if (mMediaPlayer01 != null)
      {
        if(bIsReleased == false)
        {
          if(bIsPaused==false)
          {
            mMediaPlayer01.pause();
            bIsPaused = true;
            mTextView01.setText(R.string.str_pause);
          }
          else if(bIsPaused==true)
          {
            mMediaPlayer01.start();
            bIsPaused = false;
            mTextView01.setText(R.string.str_play);
          }
        }
      }
    }
    else
    {
      mTextView01.setText(R.string.str_err_nosd);
    }
  }
});
```

（7）编写单击“停止”按钮的处理事件，具体实现代码如下所示。

```
/* 终止按钮*/
mStop.setOnClickListener(new ImageButton.OnClickListener()
{
  public void onClick(View view)
  {
    if(checkSDCard())
    {
      try
      {
        if (mMediaPlayer01 != null)
        {
          if(bIsReleased==false)
          {
            mMediaPlayer01.seekTo(0);
            mMediaPlayer01.pause();
            mTextView01.setText(R.string.str_stop);
          }
        }
      }
      catch(Exception e)
      {
        mTextView01.setText(e.toString());
        Log.e(TAG, e.toString());
        e.printStackTrace();
      }
    }
    else
    {
      mTextView01.setText(R.string.str_err_nosd);
    }
  }
});
}
```

（8）定义方法 playVideo 来下载指定 URL 地址的影片，并在下载后进行播放处理。其具体实现代码如下所示。

```
/* 自定义下载 URL 影片并播放*/
private void playVideo(final String strPath)
{
  try
  {
    /* 若传入的 strPath 为现有播放的连接，则直接播放*/
    if (strPath.equals(currentFilePath) && mMediaPlayer01 != null)
    {
      mMediaPlayer01.start();
      return;
    }
    else if(mMediaPlayer01 != null)
    {
      mMediaPlayer01.stop();
    }
    currentFilePath = strPath;
    /* 重新构建 MediaPlayer 对象*/
    mMediaPlayer01 = new MediaPlayer();
    /* 设置播放音量*/
    mMediaPlayer01.setAudioStreamType(2);
    /* 设置显示于 SurfaceHolder */
    mMediaPlayer01.setDisplay(mSurfaceHolder01);
    mMediaPlayer01.setOnErrorListener
    (new MediaPlayer.OnErrorListener()
    {
      @Override
      public boolean onError(MediaPlayer mp, int what, int extra)
      {
        // TODO Auto-generated method stub
        Log.i
        (
          TAG,
          "Error on Listener, what: " + what + "extra: " + extra
        );
        return false;
      }
    });
```

（9）定义 onBufferingUpdate 事件来监听缓冲进度，其具体实现代码如下所示。

```
mMediaPlayer01.setOnBufferingUpdateListener
(new MediaPlayer.OnBufferingUpdateListener()
{
  @Override
  public void onBufferingUpdate(MediaPlayer mp, int percent)
  {
    // TODO Auto-generated method stub
    Log.i
    (
      TAG, "Update buffer: " +
      Integer.toString(percent) + "%"
    );
  }
});
```

（10）定义方法 run()来接受连接并记录线程信息。先在运行线程时调用自定义函数来抓取下文，当下载完后调用 prepare，当有异常发生时输出错误信息。其具体实现代码如下所示。

```
Runnable r = new Runnable()
{
  public void run()
  {
    try
    {
```

```
        /* 在线程运行中，调用自定义函数抓取下文*/
        setDataSource(strPath);
         /* 下载完后才会调用 prepare */
         mMediaPlayer01.prepare();
         Log.i
         (
           TAG, "Duration: " + mMediaPlayer01.getDuration()
         );
         mMediaPlayer01.start();
         bIsReleased = false;
       }
       catch (Exception e)
       {
         Log.e(TAG, e.getMessage(), e);
       }
     }
   };
   new Thread(r).start();
 }
 catch(Exception e)
 {
   if (mMediaPlayer01 != null)
   {
     mMediaPlayer01.stop();
     mMediaPlayer01.release();
   }
 }
}
```

（11）定义方法 setDataSource，使用线程启动的方式来播放视频，具体实现代码如下所示。

```
* 自定义 setDataSource，由线程启动*/
rivate void setDataSource(String strPath) throws Exception

 if (!URLUtil.isNetworkUrl(strPath))
 {
   mMediaPlayer01.setDataSource(strPath);
 }
 else
 {
   if(bIsReleased == false)
   {
     URL myURL = new URL(strPath);
     URLConnection conn = myURL.openConnection();
     conn.connect();
     InputStream is = conn.getInputStream();
     if (is == null)
     {
       throw new RuntimeException("stream is null");
     }
     File myFileTemp = File.createTempFile
     ("hippoplayertmp", "."+getFileExtension(strPath));

     currentTempFilePath = myFileTemp.getAbsolutePath();

     /*currentTempFilePath = /sdcard/mediaplayertmp39327.dat */

     FileOutputStream fos = new FileOutputStream(myFileTemp);
     byte buf[] = new byte[128];
     do
     {
       int numread = is.read(buf);
       if (numread <= 0)
       {
         break;
       }
       fos.write(buf, 0, numread);
     }while (true);
     mMediaPlayer01.setDataSource(currentTempFilePath);
```

```
      try
      {
        is.close();
      }
      catch (Exception ex)
      {
        Log.e(TAG, "error: " + ex.getMessage(), ex);
      }
    }
  }
}
```

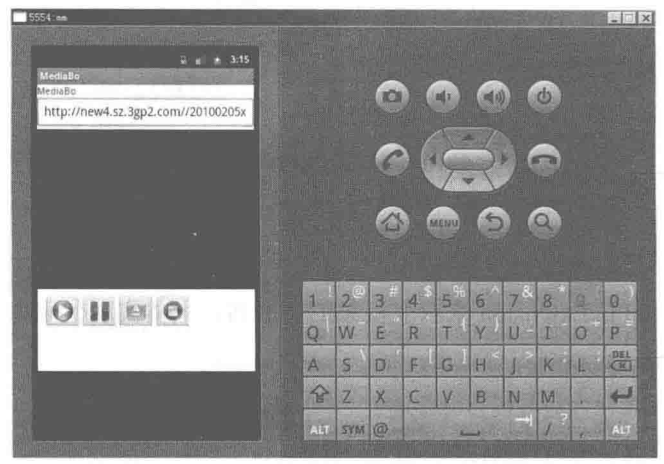
（12）定义方法 getFileExtension 来获取视频的扩展名，具体实现代码如下所示。

```
private String getFileExtension(String strFileName)
{
  File myFile = new File(strFileName);
  String strFileExtension=myFile.getName();
  strFileExtension=(strFileExtension.substring
  (strFileExtension.lastIndexOf(".")+1)).toLowerCase();

  if(strFileExtension=="")
  {
    /* 若无法顺利取得扩展名，默认为.dat */
    strFileExtension = "dat";
  }
  return strFileExtension;
}
```

（13）定义方法 checkSDCard()来判断存储卡是否存在，具体实现代码如下所示。

```
  private boolean checkSDCard()
  {
    /* 判断存储卡是否存在*/
    if(android.os.Environment.getExternalStorageState().equals
    (android.os.Environment.MEDIA_MOUNTED))
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  @Override
  public void surfaceChanged
  (SurfaceHolder surfaceholder, int format, int w, int h)
  {
    Log.i(TAG, "Surface Changed");
  }
  public void surfaceCreated(SurfaceHolder surfaceholder)
  {
    Log.i(TAG, "Surface Changed");
  }

  @Override
  public void surfaceDestroyed(SurfaceHolder surfaceholder)
  {
    Log.i(TAG, "Surface Changed");
  }
}
```

在上述代码中，通过 EditText 来获取远程视频的 URL，然后将此网址的视频下载到手机的存储卡中，并以暂存的方式保存。然后通过控制按钮来控制对视频的处理。在播放完毕并终止程序后，将暂存到 SD 卡中的临时视频删除。执行后在文本框中显示指定播放视频的 URL，当下载完毕后能实现播放处理。如图 15-1 所示。



▲图 15-1　使用 MediaPlayer 播放网络中的视频的执行效果

实例中的 MediaProvider 相当于一个数据中心，在里面记录了 SD 卡中的所有数据。而 Gallery 的什用就是展示和操作这个数据中心，每次用户启动 Gallery 时，Gallery 只是读取 MediaProvider 里面的记录并显示给用户。如果用户在 Gallery 里删除一个媒体时，Gallery 通过调用 MediaProvider 开放的接口实现。

15.2 使用 VideoView 播放视频

在 Android 系统中，内置了 VideoView Widget 作为多媒体视频播放器。在本节的内容中，将详细讲解使用 VideoView 播放视频的基本知识，为读者进入本书后面知识的学习打下基础。

15.2.1 VideoView 基础

在 Android 系统中，VideoView 的用法和其他 Widget 的使用方法类似。在使用 VideoView 时，必须先在 Layout XML 中定义 VideoView 属性，然后在程序中通过 findViewById()方法即可创建 VideoView 对象。

VideoView 的最大用处是播放视频文件，类 VideoView 可以从不同的来源（例如资源文件或内容提供器）读取图像，计算和维护视频的画面尺寸以使其适用于任何布局管理器，并提供一些诸如缩放、着色之类的显示选项。

1. 构造函数

在类 VideoView 中有 3 个构造函数，其中第一个的语法格式如下所示。

```
public VideoView (Context context)
```

通过上述函数可以创建一个默认属性的 VideoView 实例，参数 context 表示视图运行的应用程序上下文，通过它可以访问当前主题、资源等。

第二个构造函数的语法格式如下所示。

```
public VideoView (Context context, AttributeSet attrs)
```

通过上述函数可以创建一个带有 attrs 属性的 VideoView 实例，各个参数的具体说明如下所示。

- context：表示视图运行的应用程序上下文，通过它可以访问当前主题、资源等等。
- attrs：用于视图的 XML 标签属性集合。

第二个构造函数的语法格式如下所示。

```
public VideoView (Context context, AttributeSet attrs, int defStyle)
```

通过上述函数可以创建一个带有 attrs 属性，并且指定其默认样式的 VideoView 实例。各个参数的具体说明如下所示。

- context：视图运行的应用程序上下文，通过它可以访问当前主题、资源等。
- attrs：用于视图的 XML 标签属性集合。
- defStyle：应用到视图的默认风格。如果为 0 则不应用（包括当前主题中的）风格。该值可以是当前主题中的属性资源，或者是明确的风格资源 ID。

2. 公共方法

在类 VideoView 中，包含了如下所示的公共方法。

（1）public boolean canPause ()：判断是否能够暂停播放视频。

（2）public boolean canSeekBackward ()：判断是否能够倒退。

（3）public boolean canSeekForward ()：判断是否能够快进。

（4）public int getBufferPercentage ()：获得缓冲区的百分比。

（5）public int getCurrentPosition ()：获得当前的位置。

（6）public int getDuration ()：获得所播放视频的总时间。

（7）public boolean isPlaying ()：判断是否正在播放视频。

（8）public boolean onKeyDown（int keyCode,KeyEvent event）：是 KeyEvent.Callback.onKeyMultiple()的默认实现。如果视图可用并可按，当按下 KEYCODE_DPAD_CENTER 或 KEYCODE_ENTER 时执行视图的按下事件。如果处理了事件则返回 True，如果允许下一个事件接受器处理该事件则返回 false。

各个参数的具体说明如下所示。

- keyCode：表示按下的键的、在 KEYCODE_ENTER 中定义的键盘代码。
- event：KeyEvent 对象，定义了按钮动作。

（9）public boolean onTouchEvent (MotionEvent ev)：通过该方法来处理触屏事件，参数 event 表示触屏事件。如果事件已经处理返回 True，否则返回 false。

（10）public boolean onTrackballEvent (MotionEvent ev)：实现这个方法去处理轨迹球的动作事件，轨迹球相对于上次事件移动的位置能用 MotionEvent.getX()和 MotionEvent.getY()函数取回。当用户按下方向键时，将被作为一次移动操作来处理（为了表现来自轨迹球的更小粒度的移动信息，它们返回小数）。参数 ev 表示动作的事件。

（11）public void pause()：使得播放暂停。

（12）public int resolveAdjustedSize(int desiredSize, int measureSpec)：取得调整后的尺寸。如果 measureSpec 对象传入的模式是 UNSPECIFIED，那么返回的是 desiredSize。如果 measureSpec 对象传入的模式是 AT_MOST，返回的将是 desiredSize 和 measureSpec 对象的尺寸中最小的那个值。如果 measureSpec 对象传入的模式是 EXACTLY，那么返回的是 measureSpec 对象中的尺寸值。

> **注意** MeasureSpec 是一个 android.view.View 的内部类。它封装了从父类传送到子类的布局要求信息。每个 MeasureSpec 对象描述了控件的高度或者宽度。MeasureSpec 对象是由尺寸和模式组成的，有 3 个模式：UNSPECIFIED、EXACTLY、AT_MOST，这个对象由 MeasureSpec.makeMeasureSpec()函数创建。

（13）public void resume()：用于恢复挂起的播放器。

（14）public void seekTo (int msec)：设置播放位置。

（15）public void setMediaController (MediaController controller)：设置媒体控制器。

（16）public void setOnCompletionListener (MediaPlayer.OnCompletionListener l)：注册在媒体文件播放完毕时调用的回调函数。参数 l 表示要执行的回调函数。

（17）public void setOnErrorListener (MediaPlayer.OnErrorListener l)：注册在设置或播放过程中发生错误时调用的回调函数。如果未指定回调函数，或回调函数返回假，VideoView 会通知用户发生了错误。参数 l 表示要执行的回调函数。

（18）public void setOnPreparedListener (MediaPlayer.OnPreparedListener l)：用于注册在媒体文件加载完毕、可以播放时调用的回调函数。参数 l 表示要执行的回调函数。

（19）public void setVideoPath (String path)：用于设置视频文件的路径名。

（20）public void setVideoURI (Uri uri)：设置视频文件的统一资源标识符。

（21）public void start ()：开始播放视频文件。

（22）public void stopPlayback ()：停止回放视频文件。

（23）public void suspend ()：挂起视频文件的播放。

15.2.2 使用 VideoView 播放手机中的影片

经过本章 15.2.1 节中内容的介绍，我们已经了解了在 Android 系统中使用 VideoView 的基本知识。接下来将通过一个具体实例的实现过程，讲解在 Android 系统中使用 VideoView 播放手机中的影片的方法。

题目	目的	源码路径
实例 15-2	使用 MediaPlayer 播放 SD 卡中的视频	\daima\15\VideoViewBo

在本实例中，预先准备了两个“.3gp”格式的视频文件，然后将这两个文件上传到虚拟 SD 卡中。最后插入 2 个按钮，当单击按钮后分别实现对这两个视频文件的播放。

编写主程序文件 example.java，其具体实现流程如下所示。

（1）设置默认判别是否安装存储卡 flag 值为 false，然后设置全屏幕显示。其具体实现代码如下所示。

```
/* 默认判别是否安装存储卡 flag 为 false */
private boolean bIfSDExist = false;
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);

  /* 全屏幕 */
  getWindow().setFormat(PixelFormat.TRANSLUCENT);
  setContentView(R.layout.main);
```

（2）判断存储卡是否存在，不存在则通过 mMakeTextToast 输出提示。其具体实现代码如下所示。

```
* 判断存储卡是否存在 */
f(android.os.Environment.getExternalStorageState().equals
android.os.Environment.MEDIA_MOUNTED))

 bIfSDExist = true;

lse

 bIfSDExist = false;
```

```
 mMakeTextToast
 (
   getResources().getText(R.string.str_err_nosd).toString(),
   true
 );
```

（3）定义单击第一个按钮的处理事件，通过函数 playVideo(strVideoPath)来播放第一个影片。其具体实现代码如下所示。

```
Button01.setOnClickListener(new Button.OnClickListener()

 @Override
 public void onClick(View arg0)
 {
   // TODO Auto-generated method stub
   if(bIfSDExist)
   {
     /* 播放影片路径 1 */
     strVideoPath = "file:///sdcard/hello.3gp";
     playVideo(strVideoPath);
   }
 }
);
```

（4）定义单击第二个按钮的处理事件，通过函数 playVideo(strVideoPath)来播放第二个影片。其具体实现代码如下所示。

```
  mButton02.setOnClickListener(new Button.OnClickListener()
  {
    @Override
    public void onClick(View arg0)
    {
      // TODO Auto-generated method stub
      if(bIfSDExist)
      {
        /* 播放影片路径 2 */
        strVideoPath = "file:///sdcard/test.3gp";
        playVideo(strVideoPath);
      }
    }
  });
}
```

（5）定义方法 VideoView 来播放指定路径的影片，具体实现代码如下所示。

```
/* 自定义以 VideoView 播放影片 */
private void playVideo(String strPath)
{
  if(strPath!="")
  {
    /* 调用 VideoURI 方法，指定解析路径 */
    mVideoView01.setVideoURI(Uri.parse(strPath));

    /* 设置控制 Bar 显示于此 Context 中 */
    mVideoView01.setMediaController
    (new MediaController(example.this));

    mVideoView01.requestFocus();

    /* 调用 VideoView.start()自动播放 */
    mVideoView01.start();
    if(mVideoView01.isPlaying())
    {
      /*以下程序不会被运行，因 start()后尚需要 preparing() */
      mTextView01.setText("Now Playing:"+strPath);
      Log.i(TAG, strPath);
```

```
    }
  }
}
```

（6）定义方法 mMakeTextToast 来输出提醒语句，具体实现代码如下所示。

```
public void mMakeTextToast(String str, boolean isLong)
{
  if(isLong==true)
  {
    Toast.makeText(example.this, str, Toast.LENGTH_LONG).show();
  }
  else
  {
    Toast.makeText(example.this, str, Toast.LENGTH_SHORT).show();
  }
}
}
```

执行后的效果如图 15-2 所示。当单击按钮“播放影片 1”和“播放影片 2”后分别播放预设的影片。

▲图 15-2 使用 MediaPlayer 播放视频的执行效果

其实类 VideoView 的功能不止如此，它还可以从不同的来源（例如资源文件或内容提供器）读取图像，计算和维护视频的画面尺寸以使其适用于任何布局管理器，并提供一些诸如缩放、着色之类的显示选项。

15.2.3 使用 VideoView 播放手机中的 MP4

当前 Android 平台支持对 MP4 的 H.264、3GP 和 WMV 视频解析。使用 Android 内置的类 VideoView 可以快速制作一个系统播放器，VideoView 主要用来显示一个视频文件。在接下来的演示实例中，将讲解在 Android 系统中使用 VideoView 播放 MP4 视频的基本方法。

题目	目的	源码路径
实例 15-3	使用 VideoView 播放手机中 MP4	\daima\15\MP4Bo

本实例的具体实现流程如下所示。

（1）编写布局文件 main.xml，在里面设置装载、播放和暂停 3 个控制按钮。主要实现代码如下所示。

```
<TextView  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
    android:text="@string/hello"
    />
    <VideoView
    android:id="@+id/VideoView01"
    android:layout_width="320px"
   android:layout_height="240px"
     />
  <Button android:id="@+id/LoadButton"
    android:layout_width="80px"
    android:layout_height="wrap_content"
    android:text="装载"
  android:layout_x="30px"
  android:layout_y="300px"
    />
  <Button android:id="@+id/PlayButton"
    android:layout_width="80px"
    android:layout_height="wrap_content"
    android:text="播放"
  android:layout_x="120px"
  android:layout_y="300px"
    />
  <Button android:id="@+id/PauseButton"
    android:layout_width="80px"
    android:layout_height="wrap_content"
    android:text="暂停"
  android:layout_x="210px"
  android:layout_y="300px"
   />
```

（2）编写主程序文件 example.java，当单击“装载”按钮时将加载指定路径的视频文件，然后通过“播放”和“暂停”按钮来操作 MP4 文件。文件 example.java 的主要实现代码如下所示。

```
public class example extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        /* 创建 VideoView 对象 */
        final VideoView videoView = (VideoView) findViewById(R.id.VideoView01);
        /* 操作播放的 3 个按钮 */
        Button PauseButton = (Button) this.findViewById(R.id.PauseButton);
        Button LoadButton = (Button) this.findViewById(R.id.LoadButton);
        Button PlayButton = (Button) this.findViewById(R.id.PlayButton);
        /* 装载按钮事件 */
        LoadButton.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                /* 设置路径 */
                videoView.setVideoPath("/sdcard/test.mp4");
                /* 设置模式-播放进度条 */
                videoView.setMediaController(new MediaController(
                        example.this));
                videoView.requestFocus();
            }
        });
        /* 播放按钮事件 */
        PlayButton.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                /* 开始播放 */
                videoView.start();
            }
        });
        /* 暂停按钮 */
        PauseButton.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                /* 暂停 */
```

```
            videoView.pause();
        }
      });
    }
}
```

执行后的效果如图 15-3 所示。

15.2.4　开发一个网络视频播放器

当前智能手机都可以远程观看在线视频，通常视频都比较大，所以必须保证手机空间能够存储。另外，还要确保下载的视频能够被 MediaPlayer 所支持。在接下来的内容中，将通过一个具体实例的实现过程，介绍使用 MediaPlayer 播放网络中的视频的基本流程。

▲图 15-3　使用 VideoView 播放 MP4 的执行效果

题目	目的	源码路径
实例 15-4	使用 MediaPlayer 播放网络中的视频	\daima\15\bof

在本实例中，通过 EditText 来获取远程视频的 URL，然后将此网址的视频下载到手机的存储卡中，并以暂存的方式保存。接着通过控制按钮来控制对视频的处理。在播放完毕并终止程序后，将暂存到 SD 卡中的临时视频删除。本实例的主程序文件是 bof.java，其具体实现流程如下所示。

（1）定义 bIsReleased 来标识 MediaPlayer 是否已被释放，识别 MediaPlayer 是否正处于暂停，并用 LogCat 输出 TAG filter。其具体代码如下所示。

```
/* 识别 MediaPlayer 是否已被释放*/
private boolean bIsReleased = false;
/* 识别 MediaPlayer 是否正处于暂停*/
private boolean bIsPaused = false;
/* LogCat 输出 TAG filter */
private static final String TAG = "HippoMediaPlayer";
private String currentFilePath = "";
private String currentTempFilePath = "";
private String strVideoURL = "";
```

（2）设置播放视频的 URL 地址，使用 mSurfaceView01 绑定 Layout 上的 SurfaceView。然后设置 SurfaceHolder 为 Layout SurfaceView。其具体代码如下所示。

```
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  /* 将.3gp 图像文件存放入 URL 网址*/
  strVideoURL =
  "http://new4.sz.3gp2.com//20100205xyy/喜羊羊与灰太狼%20 踩高跷(www.3gp2.com).3gp";
  //http://www.dubblogs.cc:8751/Android/Test/Media/3gp/test2.3gp

  mTextView01 = (TextView)findViewById(R.id.myTextView1);
  mEditText01 = (EditText)findViewById(R.id.myEditText1);
  mEditText01.setText(strVideoURL);

  /* 绑定 Layout 上的 SurfaceView */
  mSurfaceView01 = (SurfaceView) findViewById(R.id.mSurfaceView1);

  /* 设置 PixnelFormat */
  getWindow().setFormat(PixelFormat.TRANSPARENT);
  /* 设置 SurfaceHolder 为 Layout SurfaceView */
  mSurfaceHolder01 = mSurfaceView01.getHolder();
  mSurfaceHolder01.addCallback(this);
```

（3）为影片设置大小比例，并分别设置 mPlay、mReset、mPause 和 mStop 这 4 个控制按钮。其具体代码如下所示。

```
/* 由于原有的影片 Size 较小，故指定其为固定比例*/
mSurfaceHolder01.setFixedSize(160, 128);
mSurfaceHolder01.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
mPlay = (ImageButton) findViewById(R.id.play);
mReset = (ImageButton) findViewById(R.id.reset);
mPause = (ImageButton) findViewById(R.id.pause);
mStop = (ImageButton) findViewById(R.id.stop);
```

（4）编写单击“播放”按钮的处理事件，具体代码如下所示。

```
* 播放按钮*/
Play.setOnClickListener(new ImageButton.OnClickListener()

 public void onClick(View view)
 {
   if(checkSDCard())
   {
     strVideoURL = mEditText01.getText().toString();
     playVideo(strVideoURL);
     mTextView01.setText(R.string.str_play);
   }
   else
   {
     mTextView01.setText(R.string.str_err_nosd);
   }
 }
);
```

（5）编写单击“重播”按钮的处理事件，具体代码如下所示。

```
* 重新播放按钮*/
mReset.setOnClickListener(new ImageButton.OnClickListener()

 public void onClick(View view)
 {
   if(checkSDCard())
   {
     if(bIsReleased == false)
     {
       if (mMediaPlayer01 != null)
       {
         mMediaPlayer01.seekTo(0);
         mTextView01.setText(R.string.str_play);
       }
     }
   }
   else
   {
     mTextView01.setText(R.string.str_err_nosd);
   }
 }
);
```

（6）编写单击“暂停”按钮的处理事件，具体代码如下所示。

```
* 暂停按钮*/
Pause.setOnClickListener(new ImageButton.OnClickListener()

 public void onClick(View view)
 {
   if(checkSDCard())
   {
     if (mMediaPlayer01 != null)
     {
       if(bIsReleased == false)
```

```
        {
          if(bIsPaused==false)
          {
            mMediaPlayer01.pause();
            bIsPaused = true;
            mTextView01.setText(R.string.str_pause);
          }
          else if(bIsPaused==true)
          {
            mMediaPlayer01.start();
            bIsPaused = false;
            mTextView01.setText(R.string.str_play);
          }
        }
      }
    }
    else
    {
      mTextView01.setText(R.string.str_err_nosd);
    }
  }
);
```

（7）编写单击“停止”按钮的处理事件，具体代码如下所示。

```
  /* 终止按钮*/
  mStop.setOnClickListener(new ImageButton.OnClickListener()
  {
   public void onClick(View view)
   {
     if(checkSDCard())
     {
       try
       {
         if (mMediaPlayer01 != null)
         {
           if(bIsReleased==false)
           {
             mMediaPlayer01.seekTo(0);
             mMediaPlayer01.pause();
             mTextView01.setText(R.string.str_stop);
           }
         }
       }
       catch(Exception e)
       {
         mTextView01.setText(e.toString());
         Log.e(TAG, e.toString());
         e.printStackTrace();
       }
     }
     else
     {
       mTextView01.setText(R.string.str_err_nosd);
     }
   }
  });
}
```

（8）定义方法 playVideo 来下载指定 URL 地址的影片，并在下载后进行播放处理。其具体代码如下所示。

```
/* 自定义下载 URL 影片并播放*/
private void playVideo(final String strPath)
{
  try
  {
    /* 若传入的 strPath 为现有播放的链接，则直接播放*/
```

```
    if (strPath.equals(currentFilePath) && mMediaPlayer01 != null)
    {
      mMediaPlayer01.start();
      return;
    }
    else if(mMediaPlayer01 != null)
    {
      mMediaPlayer01.stop();
    }
    currentFilePath = strPath;
    /* 重新构建 MediaPlayer 对象*/
    mMediaPlayer01 = new MediaPlayer();
    /* 设置播放音量*/
    mMediaPlayer01.setAudioStreamType(2);
    /* 设置显示于 SurfaceHolder */
    mMediaPlayer01.setDisplay(mSurfaceHolder01);
    mMediaPlayer01.setOnErrorListener
    (new MediaPlayer.OnErrorListener()
    {
      @Override
      public boolean onError(MediaPlayer mp, int what, int extra)
      {
        // TODO Auto-generated method stub
        Log.i
        (
          TAG,
          "Error on Listener, what: " + what + "extra: " + extra
        );
       return false;
      }
   });
```

（9）定义 onBufferingUpdate 事件来监听缓冲进度，具体代码如下所示。

```
mMediaPlayer01.setOnBufferingUpdateListener
(new MediaPlayer.OnBufferingUpdateListener()
{
  @Override
  public void onBufferingUpdate(MediaPlayer mp, int percent)
  {
    // TODO Auto-generated method stub
    Log.i
    (
      TAG, "Update buffer: " +
      Integer.toString(percent) + "%"
    );
  }
});
```

（10）定义方法 run()来接受连接并记录线程信息。先在运行线程时调用自定义函数抓取下文，当下载完后调用 prepare，当有异常发生时输出错误信息。其具体代码如下所示。

```
    Runnable r = new Runnable()
    {
      public void run()
      {
        try
        {
          /* 在线程运行中，调用自定义函数抓下文件*/
          setDataSource(strPath);
          /* 下载完后才会调用 prepare */
          mMediaPlayer01.prepare();
          Log.i
          (
            TAG, "Duration: " + mMediaPlayer01.getDuration()
          );
          mMediaPlayer01.start();
          bIsReleased = false;
        }
```

```
        catch (Exception e)
        {
          Log.e(TAG, e.getMessage(), e);
        }
      }
    };
    new Thread(r).start();
  }
  catch(Exception e)
  {
    if (mMediaPlayer01 != null)
    {
      mMediaPlayer01.stop();
      mMediaPlayer01.release();
    }
  }
}
```

（11）定义方法 setDataSource，使用线程启动的方式播放视频，具体代码如下所示。

```
/* 自定义 setDataSource，由线程启动*/
private void setDataSource(String strPath) throws Exception
{
  if (!URLUtil.isNetworkUrl(strPath))
  {
    mMediaPlayer01.setDataSource(strPath);
  }
  else
  {
    if(bIsReleased == false)
    {
      URL myURL = new URL(strPath);
       URLConnection conn = myURL.openConnection();
      conn.connect();
      InputStream is = conn.getInputStream();
      if (is == null)
      {
        throw new RuntimeException("stream is null");
      }
      File myFileTemp = File.createTempFile
      ("hippoplayertmp", "."+getFileExtension(strPath));

      currentTempFilePath = myFileTemp.getAbsolutePath();

      /*currentTempFilePath = /sdcard/mediaplayertmp39327.dat */

      FileOutputStream fos = new FileOutputStream(myFileTemp);
      byte buf[] = new byte[128];
      do
      {
        int numread = is.read(buf);
        if (numread <= 0)
        {
          break;
        }
        fos.write(buf, 0, numread);
      }while (true);
      mMediaPlayer01.setDataSource(currentTempFilePath);
      try
      {
        is.close();
      }
      catch (Exception ex)
      {
        Log.e(TAG, "error: " + ex.getMessage(), ex);
      }
    }
  }
}
```

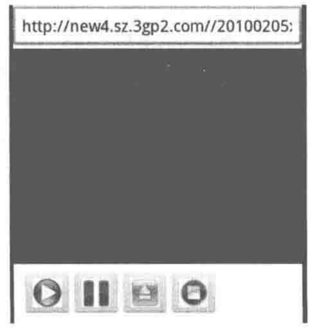
（12）定义方法 getFileExtension 来获取视频的扩展名，具体代码如下所示。

```
private String getFileExtension(String strFileName)
{
  File myFile = new File(strFileName);
  String strFileExtension=myFile.getName();
  strFileExtension=(strFileExtension.substring
  (strFileExtension.lastIndexOf(".")+1)).toLowerCase();

  if(strFileExtension=="")
  {
    /* 若无法顺利取得扩展名，默认为.dat */
    strFileExtension = "dat";
  }
  return strFileExtension;
}
```

（13）定义方法 checkSDCard()来判断存储卡是否存在，具体代码如下所示。

```
  private boolean checkSDCard()
  {
    /* 判断存储卡是否存在*/
    if(android.os.Environment.getExternalStorageState().equals
    (android.os.Environment.MEDIA_MOUNTED))
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  @Override
  public void surfaceChanged
  (SurfaceHolder surfaceholder, int format, int w, int h)
  {
    Log.i(TAG, "Surface Changed");
  }
  public void surfaceCreated(SurfaceHolder surfaceholder)
  {
    Log.i(TAG, "Surface Changed");
  }

  @Override
  public void surfaceDestroyed(SurfaceHolder surfaceholder)
  {
    Log.i(TAG, "Surface Changed");
  }
}
```

执行效果如图 15-4 所示。



▲图 15-4　使用 MediaPlayer 播放网络中的视频的执行效果

15.3 使用 Camera 拍照

拍照和录像已经成为当前手机的必备功能之一。在 Android 系统中，为我们提供了完整的相机拍照和录制视频接口，通过这些接口可以实现拍照和录制视频功能。在本节的内容中，将详细讲解在 Android 系统中开发拍照应用程序的方法，为读者进入本书后面知识的学习打下基础。

15.3.1　Camera 基础

从 Android 1.5 版本开始，在安全方面有诸多改进，其中之一与摄像头权限控制有关。在此之前，可以创建无需用户许可的拍照应用，但是现在该问题已被修复，如果想在自己的应用中使用摄像头，需要在 AndroidManifest.xml 中增加以下代码。

```
<uses-permission android:name="android.permission.CAMERA"></uses-permission>
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
```

在 Android 系统中，调用 Camera 最简单的办法是调用系统的功能，然后通过 onActivityResult 方法获得图像数据。如果读者不习惯使用 Android 的 XML 配置文件，为了代码简单起见可以先加一个 layout.xml 文件，例如下面的代码。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView android:text="Camera Demo" android:id="@+id/TextView01"
      android:layout_width="fill_parent"
android:layout_height="wrap_content"></TextView>
   <RelativeLayout android:id="@+id/FrameLayout01" android:layout_weight="1"
      android:layout_width="fill_parent"
android:layout_height="fill_parent"></RelativeLayout>
   <Button android:text="test" android:id="@+id/Button01"
      android:layout_width="wrap_content"       android:layout_height="wrap_content"
android:layout_gravity="center"></Button>
   </LinearLayout>
```

然后编写调用 Camera 的代码，具体实现代码如下所示。

```
android.media.action.IMAGE_CAPTURE
final int TAKE_PICTURE = 1;
ImageView iv;

private void test1(){
      iv = new ImageView(this);
      ((FrameLayout)findViewById(R.id.FrameLayout01)).addView(iv);
      Button buttonClick = (Button)findViewById(R.id.Button01);
      buttonClick.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View arg0) {
                startActivityForResult(new Intent("android.media.action.IMAGE_CAPTURE"),
TAKE_PICTURE);
            }

        });
}
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == TAKE_PICTURE) {
            if (resultCode == RESULT_OK) {
                Bitmap b = (Bitmap) data.getExtras().get("data");
                iv.setImageBitmap(b);
```

```
            }
        }
    }
```

上述方法是最简单的使用 Camera 的方法。通过对上述代码的分析，可以看出在 Camera 中用到的接口，具体说明如下所示。

（1）需要用 SurfaceHolder 类来显示图像，并将 SurfaceHolder 类传递给 Camera。之后 Camera 会通过该 Holder 对图像进行处理。所以程序中需要 SurfaceView 子类，并实现 SurfaceHolder.Callback 的如下接口。

```
public void surfaceChanged(SurfaceHolder holder, int format, int width,int height)
public void surfaceCreated(SurfaceHolder holder)
public void surfaceDestroyed(SurfaceHolder holder)
```

（2）在拍摄相片时主要用到下面的方法。

```
public final void takePicture(ShutterCallback shutter, PictureCallback raw, Picture
Callback jpeg)
```

上述方法中的参数是回调接口，具体说明如下所示。

- ShutterCallback。

void onShutter()：在拍照时调用该接口，用于按下拍摄按钮后播放声音等操作。

- PictureCallback。

void onPictureTaken(byte[] data, Camera camera)：在拍照时调用该接口，data 为拍摄照片数据，camera 为 Camera 类自身。

（3）预览方式接口。

void onPreviewFrame(byte[] data, Camera camera)：通过该接口可以获取摄像头每一帧的图像数据，此外还有如下几个辅助方法。

- startPreview()：开始预览；
- stopPreview()：停止预览；
- previewEnabled()：是否可以预览。

（4）设置 Camera 属性的接口。

- setPictureFormat(int pixel_format)：设置图片的格式，其取值为 PixelFormat YCbCr_420_SP、PixelFormatRGB_565 或者 PixelFormatJPEG。
- setPreviewFormat(int pixel_format)：设置图片的预览格式，取值如上。
- setPictureSize(int width,int height)：设置图片的高度和宽度，单位为像素。
- setPreviewSize(int width,int height)：设置预览的高度和宽度，取值如上。
- setPreviewFrameRate(int fps)：设置图片预览的帧速。在设置好 Camera 的参数后，可以通过函数 void startPreview()开始预览图像、void stopPreview()结束预览，通过 autoFocus(AutoFocusCallback cb)来自动对焦，最后可以通过 takePicture(ShutterCallback shutter, PictureCallback raw, PictureCallback jpeg)函数来拍照。

注意：函数 takePicture()有 3 个参数，分别为快门回调接口、原生图像数据接口和压缩格式图片数据接口。如果数据格式不存在，则数据流为空；如果不需要实现这些接口，则这些参数取值可以为 null。

（5）其他接口方法。

- 自动对焦 AutoFocusCallback：能够实现摄像头自动对焦，success 表示自动对焦是否成功。

原型是：

```
void onAutoFocus(boolean success, Camera camera);
```

- void onError（int error, Camera camera）：摄像头发生错误是调用该接口。
- CAMERA_ERROR_UNKNOWN：表示未知错误。
- CAMERA_ERROR_SERVER_DIED：表示媒体服务已经当掉，需要释放 Camera 重新启动。
- setParameters(Parameters params)：设置摄像头参数。

拍照流程

（1）设定摄像头布局。

这是开发工作的基础，也就是说我们希望在应用程序中增加多少辅助性元素，如摄像头各种功能按钮等。在本文中我们采取最简单方式，除了拍照外，没有多余摄像头功能。例如下面是布局文件 camera_surface.xml 的代码。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent" android:layout_height="fill_parent"
android:orientation="vertical">
<SurfaceView android:id="@+id/surface_camera"
android:layout_width="fill_parent" android:layout_height="10dip"
android:layout_weight="1">
</SurfaceView>
</LinearLayout>
```

上述代码的布局非常简单，只有一个 LinearLayout 视图组，在它下面只有一个 SurfaceView 视图，也就是摄像头屏幕。在上述过程中，不能在资源文件名称中使用大写字母，如果把该文件命名为“CameraSurface.xml”，就会带来不必要的麻烦。

（2）实现摄像。

创建一个名为“CameraView”的 Activity 类，然后实现 SurfaceHolder.Callback 接口。

```
public class CamaraView extends Activity implements SurfaceHolder.Callback
```

接口 SurfaceHolder.Callback 被用来接收摄像头预览界面变化的信息。它实现了 3 个方法。

- surfaceChanged：当预览界面的格式和大小发生改变时，该方法被调用。
- surfaceCreated：初次实例化，预览界面被创建时，该方法被调用。
- surfaceDestroyed：当预览界面被关闭时，该方法被调用。

下面我们一起看一下在摄像头应用中如何使用这个接口，首先看一下在 Activity 类中的方法 onCreate。

```
super.onCreate(icicle);
getWindow().setFormat(PixelFormat.TRANSLUCENT);
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
setContentView(R.layout.camera);
mSurfaceView = (SurfaceView) findViewById(R.id.surface_camera);
mSurfaceHolder = mSurfaceView.getHolder();
mSurfaceHolder.addCallback(this);
mSurfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
}
```

下面我们逐一对代码进行一下说明。

```
getWindow().setFormat(PixelFormat.TRANSLUCENT);
requestWindowFeature(Window.FEATURE_NO_TITLE);
```

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

通过上述代码，我们告诉屏幕如下两点信息。

（1）摄像头预览界面将通过全屏显示，没有“标题”(title)；

（2）屏幕格式为“半透明”。

```
setContentView(R.layout.camera_surface );
mSurfaceView = (SurfaceView) findViewById(R.id.surface_camera);
```

在以上代码中，我们为前面创建的 camera_surface 通过 setContentView 来设定 Activity 的布局，并创建一个 SurfaceView 对象，从 xml 文件中获得它。

```
mSurfaceHolder = mSurfaceView.getHolder();
mSurfaceHolder.addCallback(this);
mSurfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
```

通过以上代码，我们从 surfaceview 中获得了 holder，并增加 callback 功能到“this”。这意味着我们的操作（activity）将可以管理这个 surfaceview。

我们看一下 callback 功能是如何实现的。

```
public void surfaceCreated(SurfaceHolder holder) {
mCamera = Camera.open();
```

mCamera 是“Camera”类的一个对象。在 surfaceCreated 方法中“打开”摄像头，这就是启动它的方式。

```
public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
if (mPreviewRunning) {
mCamera.stopPreview();
}
Camera.Parameters p = mCamera.getParameters();
p.setPreviewSize(w, h);
mCamera.setParameters(p);
try {
mCamera.setPreviewDisplay(holder);
} catch (IOException e) {
e.printStackTrace();
}
mCamera.startPreview();
mPreviewRunning = true;
}
```

该方法让摄像头做好拍照准备，设定它的参数，并开始在 Android 屏幕中启动预览画面。我使用了一个“semaphore”参数来防止冲突：当 mPreviewRunning 为 true 时，意味着摄像头处于激活状态，并未被关闭，因此我们可以使用它。

```
public void surfaceDestroyed(SurfaceHolder holder) {
mCamera.stopPreview();
mPreviewRunning = false;
mCamera.release();
}
```

通过这个方法，停止摄像头并释放相关的资源。正如大家所看到的，在此设置 mPreview Running 为 false，以此来防止在 surfaceChanged 方法中的冲突。原因何在?因为这意味着我们已经关闭了摄像头，而且我们不能再设置其参数或在摄像头中启动图像预览。

最后我们看一下本例中最重要的方法。

```
Camera.PictureCallback mPictureCallback = new Camera.PictureCallback() {
public void onPictureTaken(byte[] imageData, Camera c) {
```

```
    }
    };
```

当拍照时，该方法被调用。举例来说，你可以在界面上创建一个 OnClickListener，当你单击屏幕时，调用 PictureCallBack 方法。这个方法会向你提供图像的字节数组，然后你可以使用 Android 提供的 Bitmap 和 BitmapFactory 类，将其从字节数组转换成你想要的图像格式。

15.3.2　总结 Camera 拍照的流程

经过本节前面内容的介绍，相信大家对 Camera 拍照的基本知识有了一个全新的了解。根据笔者的开发经验，以下总结出使用 Camera 实现拍照应用的基本流程，权当抛砖引玉。

（1）如果你想在自己的应用中使用摄像头，需要在 AndroidManifest.xml 中增加以下权限声明代码。

```
<uses-permissionandroid:name="android.permission.CAMERA"/>
```

（2）设定摄像头布局。

这一步是开发工作的基础，也就是说我们希望在应用程序中增加多少辅助性元素，如摄像头的各种功能按钮等。在本文中我们采取最简方式，除了拍照外，没有多余摄像头功能。下面我们一起看一下本文示例将要用到的布局文件“camera_surface.xml”。

```
<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"android:layout_height="fill_parent"
androidrientation="vertical">
<SurfaceViewandroid:id="@+id/surface_camera"
android:layout_width="fill_parent"android:layout_height="10dip"
android:layout_weight="1">
</SurfaceView>
</LinearLayout>
```

在此不要在资源文件名称中使用大写字母，如果把该文件命名为“CameraSurface.xml”，会给你带来不必要的麻烦。

上述该布局非常简单，只有一个 LinearLayout 视图组，在它下面只有一个 SurfaceView 视图，也就是我们的摄像头屏幕。

（3）编写摄像头实现代码。

在此创建一个名为“CameraView”的 Activity 类，实现 SurfaceHolder.Callback 接口。

```
publicclassCamaraViewextendsActivityimplementsSurfaceHolder.Callback
```

接口 SurfaceHolder.Callback 被用来接收摄像头预览界面变化的信息，它实现了如下 3 种方法。

- surfaceChanged：当预览界面的格式和大小发生改变时，该方法被调用。
- surfaceCreated：在初次实例化、预览界面被创建时，该方法被调用。
- surfaceDestroyed：当预览界面被关闭时，该方法被调用。

接下来看一下在摄像头应用中如何使用这个接口，首先看一下在 Activity 类中的 onCreate 方法，其中通过下面的代码设置摄像头预览界面将通过全屏显示，并且没有“标题”(title)，并设置屏幕格式为“半透明”。

```
getWindow().setFormat(PixelFormat.TRANSLUCENT);
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

然后通过 setContentView 为前面我们创建的 camera_surface 设定 Activity 的布局，并创建一个 SurfaceView 对象，从 xml 文件中获得它。其对应代码如下所示。

```
setContentView(R.layout.camera_surface);
mSurfaceView=(SurfaceView)findViewById(R.id.surface_camera);
mSurfaceHolder=mSurfaceView.getHolder();
mSurfaceHolder.addCallback(this);
mSurfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
```

通过以上代码从 surfaceview 中获得了 holder，并增加 callback 功能到“this”。这意味着我们的操作（Activity）将可以管理这个 surfaceview。

再看 Callback 功能的实现代码。

```
publicvoidsurfaceCreated(SurfaceHolderholder){
mCamera=Camera.open();
```

上面的 mCamera 是“Camera”类的一个对象。在 surfaceCreated 方法中我们“打开”摄像头。这就是启动它的方式。

接下来定义方法 surfaceChanged()让摄像头做好拍照准备，设定它的参数，并开始在 Android 屏幕中启动预览画面。在此使用了“semaphore”参数来防止冲突：当 mPreviewRunning 为 true 时，意味着摄像头处于激活状态，并未被关闭，因此我们可以使用它。

```
public void surfaceChanged(SurfaceHolderholder,intformat,intw,inth){
if(mPreviewRunning){
 mCamera.stopPreview();
}
Camera.Parametersp=mCamera.getParameters();
  p.setPreviewSize(w,h);
  mCamera.setParameters(p);
  try{
  mCamera.setPreviewDisplay(holder);
  }catch(IOExceptione){
  e.printStackTrace();
}
mCamera.startPreview();
mPreviewRunning=true;
  }
publicvoidsurfaceDestroyed(SurfaceHolderholder){
  mCamera.stopPreview();
  mPreviewRunning=false;
  mCamera.release();
}
```

通过上述方法代码停止了摄像头，并释放相关的资源。正如大家所看到的，在此设置 mPreviewRunning 为 false 以防止在 surfaceChanged 方法中的冲突。这是因为我们已经关闭了摄像头，而且我们不能再设置其参数或在摄像头中启动图像预览。

最后我们看下面的最重要的方法。

```
Camera.PictureCallbackmPictureCallback=newCamera.PictureCallback(){
  publicvoidonPictureTaken(byte[]imageData,Camerac){
}
};
```

在拍照时该方法被调用。例如，我们可以在界面上创建一个 OnClickListener，当单击屏幕时调用 PictureCallBack 方法，此方法会向你提供图像的字节数组，然后你可以使用 Android 提供的 Bitmap 和 BitmapFactory 类，将其从字节数组转换成你想要的图像格式。

到此为止，使用 Camera 拍照的基本流程介绍完毕。当然上述流程只是一种典型的解决方案，希望读者以此为基础，根据自己项目的要求扩展为自己的功能。

15.3.3　使用 Camera 预览并拍照

在接下来的内容中，将通过一个具体实例来讲解使用 Camera 实现预览和拍照功能的方法。

题目	目的	源码路径
实例 15-5	使用 Camera 预览并拍照	\daima\15\PaiZhao

本实例实现了一个简单的拍照功能，在实例中以 Activity 为基础，在 Layout 中配置了 3 个按钮，分别实现预览、关闭相机和拍照处理功能。当单击“拍照”按钮后会将屏幕中拍的画面截取下来并存储到 SD 卡中，然后将拍下来的图片显示在 Activity 中的 ImageView 控件中。为避免拍照相片造成的存储卡垃圾暂存堆栈，在离开程序前要删除临时文件。本实例的主程序文件是 example.java，其具体实现流程如下所示。

（1）引用 PictureCallback 作为取得拍照后的事件，具体实现代码如下所示。

```
/* 引用 Camera 类 */
import android.hardware.Camera;

/* 引用 PictureCallback 作为取得拍照后的事件 */
import android.hardware.Camera.PictureCallback;
import android.hardware.Camera.ShutterCallback;
import android.os.Bundle;
import android.util.DisplayMetrics;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.view.View;
import android.view.Window;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
```

（2）创建私有 Camera 对象，然后分别创建 mImageView01、mTextView01、TAG、mSurfaceView01、mSurfaceHolder01 和 intScreenY 作为预览相片之用，具体实现代码如下所示。

```
/* 使 Activity 实现 SurfaceHolder.Callback */
public class example10 extends Activity
implements SurfaceHolder.Callback
{
  /* 创建私有 Camera 对象 */
  private Camera mCamera01;
  private Button mButton01, mButton02, mButton03;

  /* 作为 review 照下来的相片之用 */
  private ImageView mImageView01;
  private TextView mTextView01;
  private String TAG = "HIPPO";
  private SurfaceView mSurfaceView01;
  private SurfaceHolder mSurfaceHolder01;
  //private int intScreenX, intScreenY;
```

（3）设置默认相机预览模式为 false，将照下来的图片存储在"/sdcard/camera_snap.jpg"目录下，具体实现代码如下所示。

```
/* 默认相机预览模式为 false */
private boolean bIfPreview = false;

/* 将照下来的图片存储在此 */
private String strCaptureFilePath = "/sdcard/camera_snap.jpg";
```

（4）使用 requestWindowFeature 设置全屏幕运行，然后判断存储卡是否存在，如果不存在则

提醒用户未安装存储卡，具体实现代码如下所示。

```
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);

  /* 使应用程序全屏幕运行，不使用 title bar */
  requestWindowFeature(Window.FEATURE_NO_TITLE);
  setContentView(R.layout.main);

  /* 判断存储卡是否存在 */
  if(!checkSDCard())
  {
    /* 提醒 User 未安装存储卡 */
    mMakeTextToast
    (
      getResources().getText(R.string.str_err_nosd).toString(),
      true
    );
  }
```

（5）通过 DisplayMetrics 对象 dm 获取屏幕解析像素，然后以 SurfaceView 作为相机预览之用，绑定 SurfaceView 后获取 SurfaceHolder 对象，通过 setFixedSize 可以设置预览大小，具体实现代码如下所示。

```
/* 取得屏幕解析像素 */
DisplayMetrics dm = new DisplayMetrics();
getWindowManager().getDefaultDisplay().getMetrics(dm);
mTextView01 = (TextView) findViewById(R.id.myTextView1);
mImageView01 = (ImageView) findViewById(R.id.myImageView1);
/* 以 SurfaceView 作为相机 Preview 之用 */
mSurfaceView01 = (SurfaceView) findViewById(R.id.mSurfaceView1);

/* 绑定 SurfaceView，取得 SurfaceHolder 对象 */
mSurfaceHolder01 = mSurfaceView01.getHolder();
/* Activity 必须实现 SurfaceHolder.Callback */
mSurfaceHolder01.addCallback(example10.this);

/* 额外的预览大小设置，在此不使用 */
/*以 SURFACE_TYPE_PUSH_BUFFERS(3)作为 SurfaceHolder 显示类型 */
mSurfaceHolder01.setType
(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);

mButton01 = (Button)findViewById(R.id.myButton1);
mButton02 = (Button)findViewById(R.id.myButton2);
mButton03 = (Button)findViewById(R.id.myButton3);
```

（6）编写打开相机和预览按钮事件，自定义初始化打开相机函数，具体实现代码如下所示。

```
/* 打开相机及 Preview */
mButton01.setOnClickListener(new Button.OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
    // TODO Auto-generated method stub

    /* 自定义初始化打开相机函数 */
    initCamera();
  }
});
```

（7）设置停止预览按钮事件，自定义重置相机并关闭相机预览函数，具体实现代码如下所示。

```
/* 停止 Preview 及相机 */
mButton02.setOnClickListener(new Button.OnClickListener()
{
```

```
  @Override
  public void onClick(View arg0)
  {
    /* 自定义重置相机，并关闭相机预览函数 */
    resetCamera();
  }
});
```

（8）设置停止拍照按钮事件，当存储卡存在才允许拍照，自定义函数 takePicture()实现拍照功能，具体实现代码如下所示。

```
  /* 拍照 */
  mButton03.setOnClickListener(new Button.OnClickListener()
  {
    @Override
    public void onClick(View arg0)
    {
      // TODO Auto-generated method stub

      /* 当存储卡存在才允许拍照，存储暂存图像文件 */
      if(checkSDCard())
      {
        /* 自定义拍照函数 */
        takePicture();
      }
      else
      {
        /* 存储卡不存在显示提示 */
        mTextView01.setText
        (
          getResources().getText(R.string.str_err_nosd).toString()
        );
      }
    }
  });
}
```

（9）定义方法 initCamera()，如果相机处于非预览模式则打开相机，具体实现代码如下所示。

```
/* 自定义初始相机函数 */
private void initCamera()
{
  if(!bIfPreview)
  {
    /* 若相机不在预览模式，则打开相机 */
    mCamera01 = Camera.open();
  }

  if (mCamera01 != null && !bIfPreview)
  {
    Log.i(TAG, "inside the camera");

    /* 创建 Camera.Parameters 对象 */
    Camera.Parameters parameters = mCamera01.getParameters();

    /* 设置相片格式为 JPEG */
    parameters.setPictureFormat(PixelFormat.JPEG);

    /* 指定 preview 的屏幕大小 */
    parameters.setPreviewSize(320, 240);

    /* 设置图片分辨率大小 */
    parameters.setPictureSize(320, 240);

    /* 为 Camera 设置 Camera.Parameters */
    mCamera01.setParameters(parameters);

    /* setPreviewDisplay 唯一的参数为 SurfaceHolder */
```

```
    mCamera01.setPreviewDisplay(mSurfaceHolder01);

    /* 立即运行 Preview */
    mCamera01.startPreview();
    bIfPreview = true;
  }
}
```

（10）定义方法 takePicture()来调用 takePicture()实现拍照并撷取图像，具体实现代码如下所示。

```
/* 拍照撷取图像 */
private void takePicture()
{
  if (mCamera01 != null && bIfPreview)
  {
    /* 调用 takePicture()方法拍照 */
    mCamera01.takePicture
    (shutterCallback, rawCallback, jpegCallback);
  }
}
```

（11）定义方法 resetCamera()来实现相机重置，具体实现代码如下所示。

```
/* 相机重置 */
private void resetCamera()
{
  if (mCamera01 != null && bIfPreview)
  {
    mCamera01.stopPreview();
    /* 扩展学习，释放 Camera 对象 */
    //mCamera01.release();
    mCamera01 = null;
    bIfPreview = false;
  }
}

private ShutterCallback shutterCallback = new ShutterCallback()
{
  public void onShutter()
  {
    // Shutter has closed
  }
};

private PictureCallback rawCallback = new PictureCallback()
{
  public void onPictureTaken(byte[] _data, Camera _camera)
  {
    // TODO Handle RAW image data
  }
};
```

（12）定义方法 delFile(String strFileName)来自定义删除临时文件，具体实现代码如下所示。

```
/* 自定义删除文件函数 */
private void delFile(String strFileName)
{
  try
  {
    File myFile = new File(strFileName);
    if(myFile.exists())
    {
      myFile.delete();
    }
  }
  catch (Exception e)
  {
    Log.e(TAG, e.toString());
```

```
        e.printStackTrace();
      }
  }
```

（13）定义方法 mMakeTextToast(String str, boolean isLong)来输出提示语句，具体实现代码如下所示。

```
public void mMakeTextToast(String str, boolean isLong)
{
  if(isLong==true)
  {
    Toast.makeText(example10.this, str, Toast.LENGTH_LONG).show();
  }
  else
  {
    Toast.makeText(example10.this, str, Toast.LENGTH_SHORT).show();
  }
}
```

（14）定义方法 checkSDCard()来检查是否有存储卡，具体实现代码如下所示。

```
private boolean checkSDCard()
{
  /* 判断存储卡是否存在 */
  if(android.os.Environment.getExternalStorageState().equals
  (android.os.Environment.MEDIA_MOUNTED))
  {
    return true;
  }
  else
  {
    return false;
  }
}
public void surfaceChanged
(SurfaceHolder surfaceholder, int format, int w, int h)
{
  Log.i(TAG, "Surface Changed");
}

public void surfaceCreated(SurfaceHolder surfaceholder)
{
  // TODO Auto-generated method stub
  Log.i(TAG, "Surface Changed");
}
```

在文件 AndroidManifest.xml 中声明 android.permission.CAMERA 权限，具体实现代码如下所示。

```
<uses-permission
android:name="android.permission.CAMERA">
```

执行后的效果如图 15-5 所示，分别单击“点击预览”、“拍照”和“关闭相机”按钮后可以实现对应的功能。

▲图 15-5　使用 Camera 预览并拍照的执行效果

15.3.4　使用 Camera API 方式拍照

在 Android 系统中，当通过 Camera API 方式实现拍照功能时，需要用到如下所示的类。

（1）Camera 类：最主要的类，用于管理 Camera 设备，常用的方法如下所示。

- open()：通过 open 方法获取 Camera 实例。
- setPreviewDisplay(SurfaceHolder)：设置预览拍照。
- startPreview()：开始预览。
- stopPreview()：停止预览。
- release()：释放 Camera 实例。
- takePicture（Camera.ShutterCallback shutter, Camera.PictureCallback raw, Camera.Picture Callback jpeg)：这是拍照要执行的方法，包含了 3 个回调参数。其中参数 Shutter 是快门按下时的回调，参数 raw 是获取拍照原始数据的回调，参数 jpeg 是获取压缩成 jpg 格式的图像数据。
- Camera.PictureCallback 接口：该回调接口包含了一个 onPictureTaken(byte[]data, Camera camera)方法。在这个方法中可以保存图像数据。

（2）SurfaceView 类：用于控制预览界面。其中 SurfaceHolder.Callback 接口用于处理预览的事件，需要实现如下所示的 3 个方法。

- surfaceCreated(SurfaceHolderholder)：预览界面创建时调用，每次界面改变后都会重新创建，需要获取相机资源并设置 SurfaceHolder。
- surfaceChanged(SurfaceHolderholder, int format, int width, int height)：在预览界面发生变化时调用，每次界面发生变化之后需要重新启动预览。
- surfaceDestroyed(SurfaceHolderholder)：预览销毁时调用，停止预览，释放相应资源。

在接下来的内容中，将通过一个具体实例来讲解使用 Camera API 方式拍照的方法。

题目	目的	源码路径
实例 15-6	使用 Camera API 方式拍照	\daima\15\AndroidCamera

本实例的具体实现流程如下所示。

（1）在布局文件 main.xml 中插入一个 Capture 按钮，具体实现代码如下所示。

```
<FrameLayout
 android:id="@+id/camera_preview"
 android:layout_width="fill_parent"
 android:layout_height="fill_parent"
 android:layout_weight="1"
 />

<Button
 android:id="@+id/button_capture"
 android:text="Capture"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 />
```

（2）在文件 AndroidManifest.xml 中添加使用 Camera 相关的声明，具体实现代码如下所示。

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

（3）编写 AndroidCameraActivity 类，具体实现代码如下所示。

```
public class AndroidCameraActivity extends Activity implements OnClickListener,
PictureCallback {
    private CameraSurfacePreview mCameraSurPreview = null;
    private Button mCaptureButton = null;
    private String TAG = "Dennis";
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Create our Preview view and set it as the content of our activity.
        FrameLayout preview = (FrameLayout) findViewById(R.id.camera_preview);
        mCameraSurPreview = new CameraSurfacePreview(this);
        preview.addView(mCameraSurPreview);

     // Add a listener to the Capture button
        mCaptureButton = (Button) findViewById(R.id.button_capture);
        mCaptureButton.setOnClickListener(this);
    }

    @Override
    public void onPictureTaken(byte[] data, Camera camera) {

        //save the picture to sdcard
        File pictureFile = getOutputMediaFile();
        if (pictureFile == null){
            Log.d(TAG, "Error creating media file, check storage permissions: ");
            return;
        }

        try {
            FileOutputStream fos = new FileOutputStream(pictureFile);
            fos.write(data);
            fos.close();

            Toast.makeText(this,"Image has been saved to "+pictureFile.getAbsolutePath(),
                    Toast.LENGTH_LONG).show();
        } catch (FileNotFoundException e) {
            Log.d(TAG, "File not found: " + e.getMessage());
        } catch (IOException e) {
            Log.d(TAG, "Error accessing file: " + e.getMessage());
        }

        // Restart the preview and re-enable the shutter button so that we can take another
picture camera.startPreview();

            //See if need to enable or not
            mCaptureButton.setEnabled(true);
    }

    @Override
    public void onClick(View v) {
        mCaptureButton.setEnabled(false);

        // get an image from the camera
        mCameraSurPreview.takePicture(this);
    }

    private File getOutputMediaFile(){
        //get the mobile Pictures directory
        File  picDir=Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_
PICTURES);

        //get the current time
        String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());

        return new File(picDir.getPath() + File.separator + "IMAGE_"+ timeStamp + ".jpg");
    }
}
```

通过上述实现代码，可以看出通过 Camera 方式实现拍照的如下基本流程。

（1）通过 Camera.open()来获取 Camera 实例。

（2）创建 Preview 类，需要继承 SurfaceView 类并实现 SurfaceHolder.Callback 接口。

（3）为相机设置 Preview。

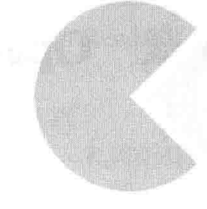
（4）构建一个 Preview 的 Layout 来预览相机。

（5）为拍照建立 Listener 以获取拍照后的回调。

（6）拍照并保存文件。

（7）释放 Camera。

本实例需要在有摄像头的真机上运行，拍完照之后可以在 SD 卡中的“Pictures”目录下找到保存的照片。



第四篇

三维技术篇

第 16 章　OpenGL ES 系统初步

第 17 章　OpenGL ES 基本应用

第 18 章　纹理映射

第 19 章　绘制不同的三维形状

第 20 章　坐标变换和混合

第 21 章　OpenGL ES 进阶

第16章　OpenGL ES 系统初步

OpenGL ES（OpenGL for Embedded Systems）是 OpenGL 三维图形 API 的子集，是专门针对手机、PDA 和游戏主机等嵌入式设备而设计的。在 Android 系统中，可以通过 OpenGL ES 提供的 API 实现三维效果功能。在本章的内容中，将详细讲解 OpenGL 基本知识，为读者进入后面知识的学习打下基础。

16.1　OpenGL ES 介绍

OpenGL ES API 由 Khronos 集团定义并推广，Khronos 是一个图形软硬件行业协会，该协会主要关注图形和多媒体方面的开放标准。OpenGL ES 是从 OpenGL 裁剪定制而来的，去除了 glBegin/glEnd、四边形（GL_QUADS）、多边形（GL_POLYGONS）等复杂图元的许多非绝对必要的特性。

16.1.1　OpenGL ES 3.0 介绍

在 SIGGRAPH 2012 大会上，OpenGL 背后的公益性组织科纳斯组织（Khronos Group）公布了如下所示新版本。

（1）面对移动领域的 OpenGL ES 版本更新到 3.0。

（2）面对桌面领域的 OpenGL 版本更新到 4.3。

（3）可运用在增强现实领域的图形接口 OpenVL。

在上述 3 个版本中，OpenGL ES 3.0 成为主角，因为它是 Android、iOS 等主流移动平台上的图形接口标准。OpenGL ES 3.0 带来很多新特性，具体说明如下所示。

（1）支持更多缓冲区对象。

在 OpenGL ES 2.0 时中，缓冲区对象的规范有模糊之处。名字一样的缓冲区对象，在实际渲染中对表现却有细微的差别。针对这个问题，OpenGL ES 3.0 制定了更详细的格式规范。新的 OpenGl ES 3.0 还增加了对 Uniform Buffer Object 的支持。

（2）GLSL ES 3.0 着色语言支持 32 位整数和浮点数据类型以及操作。

在之前版本中，着色语言只支持精度更低的数据类型。这样虽然能够加快计算的速度，所需的资源也更少，但当着色器的复杂度增加，出错也随之增加。同时，新版着色语言的语法更贴近 GLSL。

（3）支持遮挡查询（Occlusion Query）以及几何体实例化（Geometry Instancing）。

通过遮挡查询，能够让 GPU 知道 3D 场景中哪些物体被其他物体完全遮挡，这些完全被遮挡的物体不会被 GPU 渲染。几何体实例化是通过对具有相同顶点数据的几何体，赋予不同的空间位置、颜色或纹理等特征，从而创造出不同实例对象的技术。这两个特性都能够节省硬件资源，提高 3D 图形渲染的性能。

（4）增加多个纹理的支持，包括浮点纹理、深度纹理、顶点纹理等。

（5）通过多重渲染目标（Multiple Render Targets）可以让 GPU 一次性渲染多个纹理。

（6）通过多重采样抗锯齿（MSAA Render To Texture），可以让 3D 物体的边缘不出现毛刺，这样可以提升图像效果。

（7）使用统一的纹理压缩格式 ETC。

多年以来，阻碍 OpenGL 发展的一大顽疾是没有统一的纹理压缩格式，包括 S3TC、GPUs、PVPRTC、ETC 等。因为没有统一标准，所以开发者不得不根据不同的硬件环境而将纹理重复压缩多次。尤其对于 Android 开发者而言，这更是一个十分繁琐的过程。显然，通过统一纹理压缩格式能够提高开发者的开发效率。但现实中，统一标准这类事情的进展从来没有快过。ETC 的推广要看之后开发者与厂商会做出怎样的反应。

16.1.2 Android 全面支持 OpenGL ES 3.0

Android 系统从 4.3 版本开始，全面支持 OpenGL 最新的嵌入式移动版本 OpenGL ES 3.0。和旧版本的 OpenGL ES 2.0 相比，OpenGL ES 3.0 拥有更多的缓冲区对象，支持 GLSL ES 3.0 着色语言、32 位整数和浮点数据类型操作，统一了纹理压缩格式 ETC，实现了多重渲染目标和多重采样抗锯齿。这将为 Android 游戏带来更加出色的视觉效果，鼓舞开发商重视 Android 平台上的 3D 游戏业务，同时利好于谷歌游戏中心（Google Play Games）。

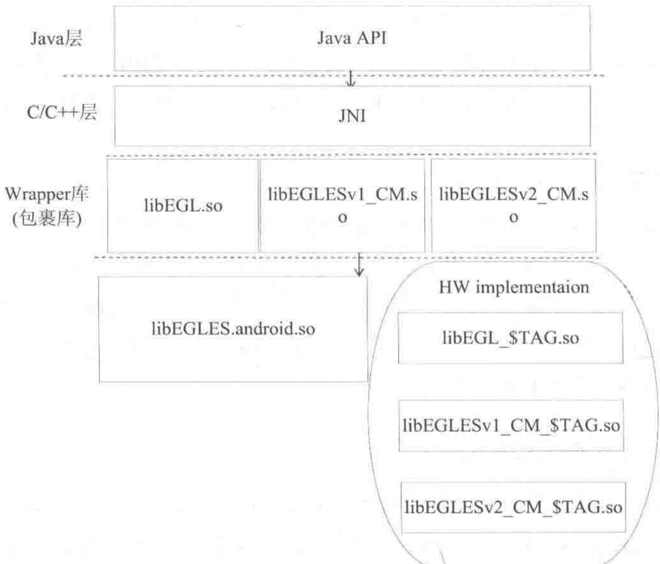
16.2 OpenGL ES 3.0 系统初步分析

在 Android 系统中，使用 OpenGL ES 的标准接口支持 3D 图形功能。根据 Android 系统的分层设计原则，OpenGL 的最上面是 Java 层，接着下面是 JNI 层，再调用下面的包裹（wrapper）层，包裹层下面则是 OpenGL 的软件实现或硬件实现。

在包 android.opengl 中的 API 类的调用层中，最上面有 4 个 OpenGL ES 的 API 类，包括 2.0 的和 3.0 的 API 类。这些类中的成员函数都是静态（static 关键字）的，且是 native 的（表示都由 JNI 层中的 C++代码实现）。而其中的 JNI 层代码被封装在 libandroid_runtime.so 库中，JNI 层中的 C++代码将调用路径 frameworks/native/opengl/libs（ICS 及以前的版本是 frameworks/base/opengl/libs）下面的代码生成的 wrapper（包裹）库。

包 javax.microedition.khronos.opengles 和包 javax.microedition.khronos.egl 实现了各个调用层堆栈功能。其中最顶层是包 javax.microedition.khronos.opengles，它用于支持 OpenGL ES API 接口规范。而另一个顶层包 javax.microedition.khronos.egl 则是 EGL 接口。虽然 OpenGL ES 为附加功能和可能的平台特性开发提供了扩展机制，但是仍然需要一个可以让 OpenGL ES 和本地视窗系统交互且与平台无关的层，此处的 EGL 就是 OpenGL ES 和底层 Native 平台视窗系统之间的接口。对于 OpenGL ES 和 EGL 两者之间的关系我们可以这样描述：OpenGL ES 本质上是一个图形渲染管线的状态机，而 EGL 则是用于监控这些状态以及维护 Frame buffer 和其他渲染 Surface 的外部层。在 EGL 中包含了一组数据类型，同时也提供了对一组平台相关的本地数据类型的支持。而处于第二层的包 com.google.android.gles_jni 用于实现其上面的接口类，本层依赖于 JNI 层的 C++实现，而 JNI 层则依赖于下面的包裹库。包裹库下面的层是 OpenGL 的硬件实现库或软件实现库，它们通过专门的图形处理器或 ARM 应用处理器去实现真正的绘制功能，这些才是真正的 OpenGL 实现。

在 Android 系统中，OpenGL ES 模块的具体架构如图 16-1 所示。



▲图 16-1　Android 系统中 OpenGL ES 模块的架构

通过图 16-1 可知，和 Android 系统中的其他功能模块一样，Android 的 3D 图形系统也被分为 Java 框架和本地代码两部分。具体说明如下所示。

（1）本地代码。

本地代码部分主要用于实现 OpenGL 接口的库。在 Java 框架层中，Java 标准的 OpenGL 包是 javax.microedition.khronos.opengles。包 android.opengl 提供了 OpenGL 系统和 Android GUI 系统之间的联系。

在 Android 源码系统中，OpenGL 的本地代码位于如下目录中。

```
frameworks/base/opengl
```

（2）JNI 接口。

在 Android 源码系统中，JNI 部分的代码位于如下所示的目录中。

```
frameworks/base/core/com_google_android_gles_jni_GLImpl.cpp
frameworks/base/core/com_google_android_gles_jni_EGLImpl.cpp
```

（3）本地测试代码。

在 Android 源码系统中，本地测试代码位于如下所示的目录中。

```
frameworks/base/opengl/tests
```

在本目录中包括 angeles、fillrate 等 14 个测试代码，这些代码都可以通过终端进行本地调用测试，也就是可以在模拟器中使用 adb shell 进行测试。

16.3 分析下层的包裹库

在 Android 系统源码中，OpenGL ES 系统有 3 个重要的包裹库，分别是 libGLESv1_CM.so、libGLESv2、libEGL。在本节的内容中，将详细讲解这 3 个包裹库的基本知识和具体运作原理。

16.3.1 libGLESv1_CM.so 包裹库详解

在 Android 系统源码中，在目录 frameworks/native/opengl/libs/GLES_CM/下共有 3 个包裹文

件:gl.cpp、gl_api.in 和 glext_api.in，如图 16-2 所示。

android源码分析 ▸ Android 4.3 ▸ frameworks ▸ native ▸ opengl ▸ libs ▸ GLES_CM

名称	修改日期	类型	大小
gl.cpp	2013/8/14 9:22	C++ Source File	8 KB
gl_api.in	2013/8/14 9:22	IN 文件	18 KB
glext_api.in	2013/8/14 9:22	IN 文件	21 KB

▲图 16-2　包裹文件（1）

上述文件将会生成 libGLESv1_CM.so 库文件，是 OpenGL ES 实现的包裹（Wrapper）库。其中在文件 gl.cpp 中包含着大量的函数，例如下面的代码。

```
void glColorPointerBounds(GLint size, GLenum type, GLsizei stride,
        const GLvoid *ptr, GLsizei count) {
    glColorPointer(size, type, stride, ptr);
}
void glNormalPointerBounds(GLenum type, GLsizei stride,
        const GLvoid *pointer, GLsizei count) {
    glNormalPointer(type, stride, pointer);
}
void glTexCoordPointerBounds(GLint size, GLenum type,
        GLsizei stride, const GLvoid *pointer, GLsizei count) {
    glTexCoordPointer(size, type, stride, pointer);
}
void glVertexPointerBounds(GLint size, GLenum type,
        GLsizei stride, const GLvoid *pointer, GLsizei count) {
    glVertexPointer(size, type, stride, pointer);
}

void GL_APIENTRY glPointSizePointerOESBounds(GLenum type,
        GLsizei stride, const GLvoid *pointer, GLsizei count) {
    glPointSizePointerOES(type, stride, pointer);
}
```

在 Android 的 OpenGL ES 系统中，上述生成的包裹库中的函数符号就来自于文件 gl.cpp 中的函数，而这些函数实现来自于两个.in 文件:gl_api.in 和 glext_api.in，然后在文件 gl.cpp 中通过“#include”语句将其包含进来。如图 16-3 所示。

gl_api.in

```
void API_ENTRY(glColor4f)(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha) {
    CALL_GL_API(glColor4f, red, green, blue, alpha);
}
void API_ENTRY(glDepthRangef)(GLclampf zNear, GLclampf zFar) {
    CALL_GL_API(glDepthRangef, zNear, zFar);
}
void API_ENTRY(glFogf)(GLenum pname, GLfloat param) {
    CALL_GL_API(glFogf, pname, param);
}
void API_ENTRY(glFogfv)(GLenum pname, const GLfloat *params) {
    CALL_GL_API(glFogfv, pname, params);
}
void API_ENTRY(glFrustumf)(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top,
    CALL_GL_API(glFrustumf, left, right, bottom, top, zNear, zFar);
}
void API_ENTRY(glGetClipPlanef)(GLenum pname, GLfloat eqn[4]) {
    CALL_GL_API(glGetClipPlanef, pname, eqn);
}
void API_ENTRY(glGetFloatv)(GLenum pname, GLfloat *params) {
    CALL_GL_API(glGetFloatv, pname, params);
}
void API_ENTRY(glGetLightfv)(GLenum light, GLenum pname, GLfloat *params) {
    CALL_GL_API(glGetLightfv, light, pname, params);
}
void API_ENTRY(glGetMaterialfv)(GLenum face, GLenum pname, GLfloat *params) {
    CALL_GL_API(glGetMaterialfv, face, pname, params);
}
void API_ENTRY(glGetTexEnvfv)(GLenum env, GLenum pname, GLfloat *params) {
    CALL_GL_API(glGetTexEnvfv, env, pname, params);
}
void API_ENTRY(glGetTexParameterfv)(GLenum target, GLenum pname, GLfloat *params) {
    CALL_GL_API(glGetTexParameterfv, target, pname, params);
```

▲图 16-3　.in 文件中的大量函数

上述做法的目的是，当需要增加新的 API 时，只需更改.in 文件即可。因为包裹库会通过统一的方式调用到真正的库中的函数，所以在.in 文件中采用宏方式来定义各个函数。

在 Android 系统中，从包裹库到真正的库的实现过程如下所示。

（1）首先解析真正的 OpenGL 实现的各个 API 函数的符号，将它们赋值给结构体（即钩子 hook）变量的函数指针成员。

（2）然后让包裹库调用这些结构体变量的成员。

（3）处理存放 API 函数指针的钩子结构体变量的地址，将它们放置到线程局部存储 TLS（Thread Local Storage）中。

为了提高上述过程的效率，当使用 TLS 和查找并调用钩子中的 API 函数指针时，特意划分为 ARM 汇编优化版本和普通的版本。其中前者使用内嵌 ARM 汇编代码完成，后者使用 C 语言完成。当后者获取和设置 TLS 中的钩子结构体变量时，函数 getGlThreadSpecific/setGlThreadSpecific 又分为两个版本进行处理，分别是 bionic 优化版本和 pthread 标准版本。

在接下来的内容中，将以 libGLESv1_CM.so 包裹库为例来详细分析上述实现过程。

在文件 gl_api.in 中，存在如下所示格式的函数。

```
void API_ENTRY(glAlphaFunc)(GLenum func, GLclampf ref) {
    CALL_GL_API(glAlphaFunc, func, ref);
}
void API_ENTRY(glClearColor)(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha) {
    CALL_GL_API(glClearColor, red, green, blue, alpha);
}
void API_ENTRY(glClearDepthf)(GLclampf depth) {
    CALL_GL_API(glClearDepthf, depth);
}
void API_ENTRY(glClipPlanef)(GLenum plane, const GLfloat *equation) {
    CALL_GL_API(glClipPlanef, plane, equation);
}
void API_ENTRY(glColor4f)(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha) {
    CALL_GL_API(glColor4f, red, green, blue, alpha);
}
```

在上述代码中，宏 API_ENTRY 和 CALL_GL_API 在文件 gl.cpp 中定义，然后又被文件 gl.cpp 包含进来。文件 gl.cpp 中的包含代码如下所示。

```
extern "C" {
#include "gl_api.in"
#include "glext_api.in"
}

#undef API_ENTRY
#undef CALL_GL_API
#undef CALL_GL_API_RETURN
```

这样文件 gl.cpp 就成为了 OpenGL ES API 的实现文件，包含了许多通过宏定义的函数。宏 API_ENTRY 和宏 CALL_GL_API 在文件 frameworks/native/opengl/libs/GLES_CM/gl.cpp 中定义，例如下面是当 USE_FAST_TLS_KEY 为 1 时的宏定义代码，此处使用汇编代码进行优化的方式来获取并调用钩子数组中的函数指针。

```
#undef API_ENTRY
#undef CALL_GL_API
#undef CALL_GL_API_RETURN

#if USE_FAST_TLS_KEY && !CHECK_FOR_GL_ERRORS

  #if defined(__arm__)
```

```
#define GET_TLS(reg) "mrc p15, 0, " #reg ", c13, c0, 3 \n"

#define API_ENTRY(_api) __attribute__((naked)) _api

#define CALL_GL_API(_api, ...)                            \
    asm volatile(                                         \
      GET_TLS(r12)                                        \
      "ldr   r12, [r12, %[tls]] \n"                       \
      "cmp   r12, #0            \n"                       \
      "ldrne pc,  [r12, %[api]] \n"                       \
      "mov   r0, #0             \n"                       \
      "bx    lr                 \n"                       \
      :                                                   \
      : [tls] "J"(TLS_SLOT_OPENGL_API*4),                 \
        [api] "J"(__builtin_offsetof(gl_hooks_t, gl._api))    \
      :                                                   \
      );
#define API_ENTRY(_api) __attribute__((noinline)) _api

#define CALL_GL_API(_api, ...)                              \
   register unsigned int _t0 asm("t0");                     \
   register unsigned int _fn asm("t1");                     \
   register unsigned int _tls asm("v1");                    \
   register unsigned int _v0 asm("v0");                     \
   asm volatile(                                            \
     ".set  push\n\t"                                       \
     ".set  noreorder\n\t"                                  \
     ".set  mips32r2\n\t"                                   \
     "rdhwr %[tls], $29\n\t"                                \
     "lw    %[t0], %[OPENGL_API](%[tls])\n\t"               \
     "beqz  %[t0], 1f\n\t"                                  \
     " move %[fn], $ra\n\t"                                 \
     "lw    %[fn], %[API](%[t0])\n\t"                       \
     "movz  %[fn], $ra, %[fn]\n\t"                          \
     "1:\n\t"                                               \
     "j     %[fn]\n\t"                                      \
     " move %[v0], $0\n\t"                                  \
     ".set  pop\n\t"                                        \
     : [fn] "=c"(_fn),                                      \
       [tls] "=&r"(_tls),                                   \
       [t0] "=&r"(_t0),                                     \
       [v0] "=&r"(_v0)                                      \
     : [OPENGL_API] "I"(TLS_SLOT_OPENGL_API*4),             \
       [API] "I"(__builtin_offsetof(gl_hooks_t, gl._api)) \
     :                                                      \
     );
```

在上述代码中，API_ENTRY 用于标识函数名称，CALL_GL_API 用于调用并返回钩子中的函数。未使用优化选项的宏，需要借助 C 库的实现来获取线程局部存储 TLS（Thread Local Storage）数据。

当展开上面的宏 CALL_GL_API 和 CALL_GL_API_RETURN 后，其中是各个 API 实现的函数体。在函数体中，首先使用 bionic 库中的函数 getGlThreadSpecific 调用获取线程局部存储中的数据。获取的数据类型是 void*类型的指针，实际上就是 gl_hooks_t 的结构体变量的地址。其对应代码如下所示。

```
#define API_ENTRY(_api) _api

#define CALL_GL_API(_api, ...)                                    \
   gl_hooks_t::gl_t const * const _c = &getGlThreadSpecific()->gl; \
   _c->_api(__VA_ARGS__);                                         \
   CHECK_GL_ERRORS(_api)

#define CALL_GL_API_RETURN(_api, ...)                             \
   gl_hooks_t::gl_t const * const _c = &getGlThreadSpecific()->gl; \
   return _c->_api(__VA_ARGS__)
```

由此可见，库 libGLESv1_CM 的实现会被最终转到调用 gl_hooks_t::gl_t（即 gl_hooks_t 结构体中的第一个结构体成员）中的结构体中的各个函数指针成员上。

如果定义了宏 USE_FAST_TLS_KEY，则需要借助 Android 系统自带的 C 库 bionic 优化的函数，来获取线程局部存储 TLS 中的变量以获取相关指针值。函数 getGlThreadSpecific 在文件 frameworks/native/opengl/libs/EGL/egl.cpp 中定义，具体实现代码如下所示。

```
void setGlThreadSpecific(gl_hooks_t const *value) {
    gl_hooks_t const * volatile * tls_hooks = get_tls_hooks();
    tls_hooks[TLS_SLOT_OPENGL_API] = value;
}

gl_hooks_t const* getGlThreadSpecific() {
    gl_hooks_t const * volatile * tls_hooks = get_tls_hooks();
    gl_hooks_t const* hooks = tls_hooks[TLS_SLOT_OPENGL_API];
    if (hooks) return hooks;
    return &gHooksNoContext;
}

#else

void setGlThreadSpecific(gl_hooks_t const *value) {
    pthread_setspecific(gGLWrapperKey, value);
}

gl_hooks_t const* getGlThreadSpecific() {
    gl_hooks_t const* hooks =static_cast<gl_hooks_t*>(pthread_getspecific(gGLWrapperKey));
    if (hooks) return hooks;
    return &gHooksNoContext;
}
```

通过上述实现代码可知，函数 getGlThreadSpecific 能够获取线程局部存储区的基址，并通过索引 TLS_SLOT_OPENGL_API 可以得到 gl_hooks_t 变量的指针。

在文件 bionic/libc/private/bionic_tls.h 中，为 OpenGL 定义了用于存储钩子指针值的 TLS 存储区数组，下面的代码是文件 bionic_tls.h 中的数组索引值。

```
enum {
  TLS_SLOT_SELF = 0, /* The kernel requires this specific slot for x86. */
  TLS_SLOT_THREAD_ID,
  TLS_SLOT_ERRNO,
  TLS_SLOT_OPENGL_API = 3,
  TLS_SLOT_OPENGL = 4,
  TLS_SLOT_BIONIC_PREINIT = TLS_SLOT_OPENGL_API,
  TLS_SLOT_STACK_GUARD = 5, /* GCC requires this specific slot for x86. */
  TLS_SLOT_DLERROR,
  TLS_SLOT_FIRST_USER_SLOT /* Must come last! */
};
```

通过上述实现代码可知，TLS_SLOT_OPENGL_API 的索引值为 3。如果不使用优化，则直接使用 pthread 库的函数调用，即下面的代码。

```
gl_hooks_t const* hooks =static_cast<gl_hooks_t*>(pthread_getspecific(gGLWrapperKey));
```

通过上述分析可以看出，函数 getGlThreadSpecific 使用的线程有 3 种，分别是局部存储、bionic 专门优化、pthread 库提供的。通过此函数获取钩子结构体 gl_hooks_t 变量的指针，就可以得到它里面的 gl 成员中的 API 函数指针。调用它们其实就是调用 OpenGL ES v1.x 的真正实现者。只要通过调用 setGlThreadSpecific 的方式为线程局部存储项指定不同的钩子结构体变量的地址，就可以使用不同的钩子，从而达到改变应用层的 OpenGL ES 调用到别的 OpenGL ES 实现的目的。

16.3.2 libGLESv2 包裹库详解

在 Android 系统源码中，frameworks/native/opengl/libs/GLES2/下有 5 个文件，分别是 gl2.cpp、gl2_api.in、gl2ext_api.in、gl3_api.in、gl3ext_api.in，这 5 个文件将生成库文件 libGLESv2.so，这是 OpenGL ES 2.0 和 3.0 的包裹（Wrapper）库。如图 16-4 所示。

名称	修改日期	类型	大小
gl2.cpp	2013/8/14 9:22	C++ Source File	5 KB
gl2_api.in	2013/8/14 9:22	IN 文件	19 KB
gl2ext_api.in	2013/8/14 9:22	IN 文件	21 KB
gl3_api.in	2013/8/14 9:22	IN 文件	36 KB
gl3ext_api.in	2013/8/14 9:22	IN 文件	0 KB

▲图 16-4　包裹文件（2）

libGLESv2 和 libGLESv1_CM 的实现方式类似，文件 gl2.cpp 通过包含的形式将 4 个.in 文件包含进来。当展开宏定义后，各个函数体会使用线程局部存储中的钩子结构体中的函数指针，于是 OpenGL ES 2.0 和 3.0 实现中的调用都转向调用到钩子指针所指向结构体中的成员函数。例如下面是在文件 gl2.cpp 中包含.in 文件的代码。

```
extern "C" {
#include "gl3_api.in"
#include "gl2ext_api.in"
#include "gl3ext_api.in"
}
#undef API_ENTRY
#undef CALL_GL_API
#undef CALL_GL_API_RETURN
```

当展开文件 gl2.cpp 中的函数后，会发现也是调用了钩子结构体，具体代码如下所示。

```
#undef API_ENTRY
#undef CALL_GL_API
#undef CALL_GL_API_RETURN

#if USE_FAST_TLS_KEY

  #if defined(__arm__)

    #define GET_TLS(reg) "mrc p15, 0, " #reg ", c13, c0, 3 \n"

    #define API_ENTRY(_api) __attribute__((naked)) _api

    #define CALL_GL_API(_api, ...)                              \
         asm volatile(                                      \
           GET_TLS(r12)                                     \
           "ldr   r12, [r12, %[tls]] \n"                      \
           "cmp   r12, #0          \n"                       \
           "ldrne pc,  [r12, %[api]] \n"                      \
           "mov   r0, #0           \n"                       \
           "bx    lr               \n"                      \
           :                                               \
           : [tls] "J"(TLS_SLOT_OPENGL_API*4),                 \
             [api] "J"(__builtin_offsetof(gl_hooks_t, gl._api))    \
           :                                               \
           );

  #elif defined(__mips__)

    #define API_ENTRY(_api) __attribute__((noinline)) _api

    #define CALL_GL_API(_api, ...)                              \
        register unsigned int _t0 asm("t0");                    \
        register unsigned int _fn asm("t1");                    \
```

```
        register unsigned int _tls asm("v1");                \
        register unsigned int _v0 asm("v0");                 \
        asm volatile(                                        \
            ".set  push\n\t"                                 \
            ".set  noreorder\n\t"                            \
            ".set mips32r2\n\t"                              \
            "rdhwr %[tls], $29\n\t"                          \
            "lw    %[t0], %[OPENGL_API](%[tls])\n\t"         \
            "beqz  %[t0], 1f\n\t"                            \
            " move %[fn],$ra\n\t"                            \
            "lw    %[fn], %[API](%[t0])\n\t"                 \
            "movz  %[fn], $ra, %[fn]\n\t"                    \
            "1:\n\t"                                         \
            "j     %[fn]\n\t"                                \
            " move %[v0], $0\n\t"                            \
            ".set  pop\n\t"                                  \
            : [fn] "=c"(_fn),                                \
              [tls] "=&r"(_tls),                             \
              [t0] "=&r"(_t0),                               \
              [v0] "=&r"(_v0)                                \
            : [OPENGL_API] "I"(TLS_SLOT_OPENGL_API*4),       \
              [API] "I"(__builtin_offsetof(gl_hooks_t, gl._api)) \
            :                                                \
            );

  #else

    #error Unsupported architecture

  #endif

    #define CALL_GL_API_RETURN(_api, ...) \
        CALL_GL_API(_api, __VA_ARGS__) \
        return 0; // placate gcc's warnings. never reached.

#else

    #define API_ENTRY(_api) _api

    #define CALL_GL_API(_api, ...)                                       \
        gl_hooks_t::gl_t const * const _c = &getGlThreadSpecific()->gl;  \
        _c->_api(__VA_ARGS__);

    #define CALL_GL_API_RETURN(_api, ...)                                \
        gl_hooks_t::gl_t const * const _c = &getGlThreadSpecific()->gl;  \
        return _c->_api(__VA_ARGS__)

#endif
```

由上述实现代码可知，OpenGL ES 2.0 和 3.0 的真正实现的 API 函数指针，也是被存放在 gl_hooks_t 的第一个成员 gl 中。

文件 gl3_api.in 和文件 gl3ext_api.in 是针对全新的 OpenGL ES 3.0 推出的，在里面定义了大量的功能函数，例如下面的代码。

```
void API_ENTRY(glActiveTexture)(GLenum texture) {
    CALL_GL_API(glActiveTexture, texture);
}
void API_ENTRY(glAttachShader)(GLuint program, GLuint shader) {
    CALL_GL_API(glAttachShader, program, shader);
}
void API_ENTRY(glBindAttribLocation)(GLuint program, GLuint index, const GLchar* name)
{
    CALL_GL_API(glBindAttribLocation, program, index, name);
}
void API_ENTRY(glBindBuffer)(GLenum target, GLuint buffer) {
    CALL_GL_API(glBindBuffer, target, buffer);
}
```

```
void API_ENTRY(glBindFramebuffer)(GLenum target, GLuint framebuffer) {
    CALL_GL_API(glBindFramebuffer, target, framebuffer);
}
void API_ENTRY(glBindRenderbuffer)(GLenum target, GLuint renderbuffer) {
    CALL_GL_API(glBindRenderbuffer, target, renderbuffer);
}
```

16.3.3　libEGL 包裹库详解

在 Android 系统源码中，在目录 frameworks/native/opengl/libs/EGL/下保存了多个.h 文件和.cpp 文件，如图 16-5 所示。

egl.cpp	2013/8/14 9:22	C++ Source File	12 KB
egl_cache.cpp	2013/8/14 9:22	C++ Source File	11 KB
egl_cache.h	2013/8/14 9:22	C Header File	6 KB
egl_display.cpp	2013/8/14 9:22	C++ Source File	14 KB
egl_display.h	2013/8/14 9:22	C Header File	9 KB
egl_entries.in	2013/8/14 9:22	IN 文件	5 KB
egl_object.cpp	2013/8/14 9:22	C++ Source File	4 KB
egl_object.h	2013/8/14 9:22	C Header File	5 KB
egl_tls.cpp	2013/8/14 9:22	C++ Source File	5 KB
egl_tls.h	2013/8/14 9:22	C Header File	3 KB
eglApi.cpp	2013/8/14 9:22	C++ Source File	45 KB
egldefs.h	2013/8/14 9:22	C Header File	2 KB
getProcAddress.cpp	2013/8/14 9:22	C++ Source File	8 KB
Loader.cpp	2013/8/14 9:22	C++ Source File	12 KB
Loader.h	2013/8/14 9:22	C Header File	3 KB
trace.cpp	2013/8/14 9:22	C++ Source File	17 KB

▲图 16-5　EGL 目录下的文件

通过上述代码生成的 libEGL.so 库也是一个 Wrapper 库。EGL 是 Khronos 的渲染 API（如 OpenGL ES 和 OpenVG）与底层 native 平台窗口系统（如桌面版 Linux 中的 X Window，MS-Windows 中的 GDI，Android 中的 Frame Buffer 等）之间进行交互的接口。EGL 负责处理图形上下文的管理、surface 与 buffer 的绑定、渲染同步机制和 2D/3D 混合渲染，并且也支持多媒体系统中的视频帧纹理贴图功能，具体实现流程如下所示。

（1）使用 EGL 创建图形上下文环境（graphics contexts）和创建用于渲染绘制的 surface.

（2）然后客户端 APIs（clients API，如 OpenGL Es 或 OpenVG）可以在 surface 上绘制图形。

（3）再使用 native 的平台渲染 API，将结果绘制到同步屏幕上。

EGL 底层真正实现的 API 函数指针将被存放在 egl_t 结构体中，上层的 EGL 调用都将通过结构体 egl_t 中的函数指针调用到下层的 EGL 实现。

结构体 egl_t 和 gl_hooks_t 在文件 frameworks/native/opengl/libs/hooks.h 中定义，具体代码如下所示。

```
struct egl_t {
    #include "EGL/egl_entries.in"
};
struct gl_hooks_t {
    struct gl_t {
        #include "entries.in"
    } gl;
    struct gl_ext_t {
        __eglMustCastToProperFunctionPointerType extensions[MAX_NUMBER_OF_GL_EXTENSIONS];
    } ext;
};
```

另外，在文件 hooks.h 中还定义了如下所示的宏。

```
#undef GL_ENTRY
#undef EGL_ENTRY
#define GL_ENTRY(_r, _api, ...) _r (*_api)(__VA_ARGS__);
#define EGL_ENTRY(_r, _api, ...) _r (*_api)(__VA_ARGS__);
```

在上述宏定义中，宏参数_r 表示返回类型，参数_api 表示函数指针，__VA_ARGS__表示函数参数，即函数返回值、函数名称、函数参数是通过宏的形式来定义的。在很多 entries.in 文件中保存的就是这些宏，它们在文件 hooks.h 中被包含在结构体中。所以当预处理之后，这些结构体就包含了大量的函数指针成员。结构体 egl_t 经过预处理后，文件 frameworks/native/opengl/libs/EGL/egl_entries.in 被包含进来，在里面定义的各项 EGL_ENTRY 宏被展开。在文件 egl_entries.in 中，和 EGL_ENTRY 宏相关的代码如下所示。

```
EGL_ENTRY(EGLDisplay, eglGetDisplay, NativeDisplayType)
EGL_ENTRY(EGLBoolean, eglInitialize, EGLDisplay, EGLint*, EGLint*)
EGL_ENTRY(EGLBoolean, eglTerminate, EGLDisplay)
EGL_ENTRY(EGLBoolean, eglGetConfigs, EGLDisplay, EGLConfig*, EGLint, EGLint*)
EGL_ENTRY(EGLBoolean, eglChooseConfig, EGLDisplay, const EGLint *, EGLConfig *, EGLint,
EGLint *)
```

其中，结构体 gl_hooks_t::gl_t 包含了文件 entries.in 中的宏 GL_ENTRY，展开后也包含了大量函数指针。这样，结构体 gl_hooks_t 由一个包含众多函数指针的 gl_t 结构体和包含函数指针数组的结构体 gl_ext_t 组成，其中__eglMustCastToProperFunctionPointerType 是一个函数类型的 typdef。通过为这几个结构体成员或结构体中的数组赋值的方法，就可以让 Wrapper 库里的.cpp 文件（如 GLES_CM/gl.cpp）中的各个函数调用指定的函数指针，也就实现了钩子（hook）功能，这相当于将调用“钩”到另外的地方。最后被统一放置到结构体 egl_connection_t。结构体 egl_connection_t 在文件 frameworks/native/opengl/libs/egl/egldefs.h 中定义，具体实现代码如下所示。

```
struct egl_connection_t {
   enum {
      GLESv1_INDEX = 0,
      GLESv2_INDEX = 1
   };

   inline egl_connection_t() : dso(0) { }
   void *              dso;
   gl_hooks_t *         hooks[2];
   EGLint              major;
   EGLint              minor;
   egl_t               egl;
};
```

在上述代码中，数组 hooks[2]被用来保存底层的 OpenGL ES API 函数指针，具体版本由如下索引代码进行区分。

```
LESv1_INDEX = 0,
LESv2_INDEX = 1
```

另外，在结构体 egl_connection_t 的定义代码中，egl_t 用于保存 EGL 库的 API 函数指针。

在文件 frameworks/native/opengl/libs/egl/egl.cpp 中，定义了结构体 egl_connection_t 的全局变量声明，具体代码如下所示。

```
egl_connection_t gEGLImpl;
gl_hooks_t gHooks[2];
gl_hooks_t gHooksNoContext;
pthread_key_t gGLWrapperKey = -1;
```

这样就为变量分配了栈上的内存空间，接下来 Loader 会进行库的装载工作，并将函数符号解析的结果赋值给结构体的成员变量。

注意

在 Android 系统中，每个使用 OpenGL 进行绘制和渲染的线程在操作时，必须与某一个 gl_hooks_t 相关联。当存在具体关联时，这些 hooks 指针存储在 TLS 中。当不存在具体关联时会使用全局 hook，这将导致在使用 openGLES 时会最终调用一个输出错误 log 信息的函数，具体过程如文件 egl.cpp 中的函数 gl_no_context。

```
static int gl_no_context() {
    if (egl_tls_t::logNoContextCall()) {
        char const* const error = "call to OpenGL ES API with "
                "no current context (logged once per thread)";
        if (LOG_NDEBUG) {
            ALOGE(error);
        } else {
            LOG_ALWAYS_FATAL(error);
        }
        char value[PROPERTY_VALUE_MAX];
        property_get("debug.egl.callstack", value, "0");
        if (atoi(value)) {
            CallStack stack(LOG_TAG);
        }
    }
    return 0;
}
```

16.4 加载并解析 OpenGL 库

在 Android 系统中进行 OpenGL 编程时，第一步是获取 Display，这一过程导致 eglGetDisplay 的调用。在函数 eglGetDisplay 中会调用 egl_init_drivers 实现驱动初始化的工作，并对装载的各个库并进行解析处理，将解析得到的符号放置于全局变量 egl_connection_t 所包含或指向的结构体的成员变量中。

16.4.1 开始加载并解析

函数 eglGetDisplay 的功能是在获取 display 时实现驱动初始化工作，此函数在文件 frameworks/native/opengl/libs/egl/eglApi.cpp 中定义，具体实现代码如下所示。

```
EGLDisplay eglGetDisplay(EGLNativeDisplayType display)
{
    clearError();

    uint32_t index = uint32_t(display);
    if (index >= NUM_DISPLAYS) {
        return setError(EGL_BAD_PARAMETER, EGL_NO_DISPLAY);
    }

    if (egl_init_drivers() == EGL_FALSE) {
        return setError(EGL_BAD_PARAMETER, EGL_NO_DISPLAY);
    }

    EGLDisplay dpy = egl_display_t::getFromNativeDisplay(display);
    return dpy;
}
```

在上述代码中调用了函数 egl_init_drivers，此函数使用 pthread_mutex 上锁保护调用了 egl_init_drivers_locked。而 egl_init_drivers_locked 使用 Loader 解析了 openGL 库中的 API 函数符号，并赋值给 egl_connection_t 全局变量 gEGLImpl 中的成员变量。

函数 egl_init_drivers 在文件 egl.cpp 中定义，具体实现代码如下所示。

```
EGLBoolean egl_init_drivers() {
```

```
    EGLBoolean res;
    pthread_mutex_lock(&sInitDriverMutex);
    res = egl_init_drivers_locked();
    pthread_mutex_unlock(&sInitDriverMutex);
    return res;
}
```

函数 egl_init_drivers_locked 在文件 egl.cpp 中定义，具体实现代码如下所示。

```
static EGLBoolean egl_init_drivers_locked() {
    if (sEarlyInitState) {
        // initialized by static ctor. should be set here.
        return EGL_FALSE;
    }

    // get our driver loader
    Loader& loader(Loader::getInstance());

    // dynamically load our EGL implementation
    egl_connection_t* cnx = &gEGLImpl;
    if (cnx->dso == 0) {
        cnx->hooks[egl_connection_t::GLESv1_INDEX] =
                &gHooks[egl_connection_t::GLESv1_INDEX];
        cnx->hooks[egl_connection_t::GLESv2_INDEX] =
                &gHooks[egl_connection_t::GLESv2_INDEX];
        cnx->dso = loader.open(cnx);
    }

    return cnx->dso ? EGL_TRUE : EGL_FALSE;
}
```

在上述实现代码中，在函数 egl_init_drivers_locked 中的第一行实现了判断处理，通过 if 语句判断是否成功进行了初始化工作。此处的初始化操作由 pthread_once 实现，具体实现代码如下所示。

```
static pthread_once_t once_control = PTHREAD_ONCE_INIT;
static int sEarlyInitState = pthread_once(&once_control, &early_egl_init);
```

这样做的目的是确保只调用初始化例程 early_egl_init 一次，如果没有成功则将 sEarlyInitState 设置为 1，直接返回 egl_init_drivers_locked。

除此之外，还可以设置属性"debug.egl.trace"的值，这样可以输出调用的 OpenGL 函数名称。此功能是通过调用 setGlThreadSpecific 为 gl_hooks_t 设置不同的钩子来实现的。具体代码在文件 egl.cpp 实现，如下所示。

```
void setGlThreadSpecific(gl_hooks_t const *value) {
    gl_hooks_t const * volatile * tls_hooks = get_tls_hooks();
    tls_hooks[TLS_SLOT_OPENGL_API] = value;
}
```

16.4.2　库加载器 Loader 详解

在 Android 系统中，OpenGL 库由库加载器 Loader 负责加载工作，其最终目的是为钩子赋值，让钩子指向真正的 OpenGL 实现。这将根据配置文件搜索特定路径下的库文件，解析出函数符号的地址，将这些函数地址赋值给钩子结构体中的成员。上述功能在文件 frameworks/native/opengl/libs/egl/Loader.cpp 中实现，可以加载如下所示的库。

- libGLES_$TAG.so。
- lib{[EGL|GLESv1_CM|GLESv2]}_$TAG.so。

其中，$TAG 由配置文件/system/lib/egl/egl.cfg 设置。如果该文件不存在，则使用"android"作为默认 TAG。

在接下来的内容中，将详细分析文件 Loader.cpp 的具体实现过程。

（1）构造函数 Loader()。

在文件 Loader.cpp 中，构造函数 Loader()能够根据配置文件读取配置项为后续中打开哪些库文件做好准备，具体实现代码如下所示。

```
Loader::Loader()
{
    char line[256];
    char tag[256];

    /* Special case for GLES emulation */
    if (checkGlesEmulationStatus() == 0) {
        ALOGD("Emulator without GPU support detected. "
              "Fallback to software renderer.");
        mDriverTag.setTo("android");
        return;
    }

    /* Otherwise, use egl.cfg */
    FILE* cfg = fopen("/system/lib/egl/egl.cfg", "r");
    if (cfg == NULL) {
        // default config
        ALOGD("egl.cfg not found, using default config");
        mDriverTag.setTo("android");
    } else {
        while (fgets(line, 256, cfg)) {
            int dpy, impl;
            if (sscanf(line, "%u %u %s", &dpy, &impl, tag) == 3) {
                //ALOGD(">>> %u %u %s", dpy, impl, tag);
                // We only load the h/w accelerated implementation
                if (tag != String8("android")) {
                    mDriverTag = tag;
                }
            }
        }
        fclose(cfg);
    }
}
```

在上述代码中，当用软件实现时，如果没有找到配置文件，则将"android"作为（0,0, "android"）被添加进去，这表示它是 0 号 Display（Android 默认的 Display）和 OpenGL 软件实现（值为 0）对应的 tag。也就是在文件 egl.cfg 中的第一个数字是 Display 号，都为 0。第二个是软硬件实现号，其中用 0 表示软件实现，用 1 表示硬件实现。例如，在 Nexus7 中的配置文件 device/asus/grouper/egl.cfg 的具体配置如下所示。

```
0 0 android
0 1 tegra
```

通过上述设置，表示软件实现的 tag 是 android，硬件实现的 tag 是 tegra。所以将加载解析如下所示的库文件。

```
/system/lib/egl/lib{[EGL|GLESv1_CM|GLESv2] }_tegra.so
```

（2）函数 open。

在文件 Loader.cpp 中，函数 open 的功能是调用 load_driver 加载解析对应的库，具体实现代码如下所示。

```
void* Loader::open(egl_connection_t* cnx)
{
    void* dso;
    driver_t* hnd = 0;

    char const* tag = mDriverTag.string();
```

```
    if (tag) {
        dso = load_driver("GLES", tag, cnx, EGL | GLESv1_CM | GLESv2);
        if (dso) {
            hnd = new driver_t(dso);
        } else {
            // Always load EGL first
            dso = load_driver("EGL", tag, cnx, EGL);
            if (dso) {
                hnd = new driver_t(dso);
                // TODO: make this more automated
                hnd->set( load_driver("GLESv1_CM", tag, cnx, GLESv1_CM), GLESv1_CM );
                hnd->set( load_driver("GLESv2",    tag, cnx, GLESv2),    GLESv2 );
            }
        }
    }
    LOG_FATAL_IF(!index && !hnd,
            "couldn't find the default OpenGL ES implementation "
            "for default display");

    return (void*)hnd;
}
```

通过上述代码可知，函数 open 会首先尝试解析形如 libGLES_$TAG.so 的库。如果失败（不存在这种方式命名的库），则尝试解析形如 libEGL_$TAG.so、libGLESv1_CM_$TAG.so 和 libGLESv2_$TAG.so 格式的库。

（3）函数 load_driver。

在函数 open 中使用函数 load_driver 装载并解析库，由此可见，函数 load_driver 的功能是解析某个库，具体说明如下所示。

- 将文件 entries.in 中的各个函数名称解析出来，得到 OpenGL ES 实现库中的函数指针。
- 将文件 egl_entries.in 里的函数名称解析出来，得到 EGL 实现库的函数指针。
- 对于库 EGL 来说，将解析后得到的函数符号赋给 egl_connection_t 中的 egl_t 结构体成员。
- 对于库 GLESv1_CM 和库 GLESv2 来说，将解析的函数符号赋给 egl_connection_t 中 gl_hooks_t *型数组 hooks[2]所指向的结构体的成员。

函数 load_driver 的具体实现代码如下所示。

```
void *Loader::load_driver(const char* kind, const char *tag,
        egl_connection_t* cnx, uint32_t mask)
{
    char driver_absolute_path[PATH_MAX];
    const char* const search1 = "/vendor/lib/egl/lib%s_%s.so";
    const char* const search2 = "/system/lib/egl/lib%s_%s.so";

    snprintf(driver_absolute_path, PATH_MAX, search1, kind, tag);
    if (access(driver_absolute_path, R_OK)) {
        snprintf(driver_absolute_path, PATH_MAX, search2, kind, tag);
        if (access(driver_absolute_path, R_OK)) {
            // this happens often, we don't want to log an error
            return 0;
        }
    }

    void* dso = dlopen(driver_absolute_path, RTLD_NOW | RTLD_LOCAL);
    if (dso == 0) {
        const char* err = dlerror();
        ALOGE("load_driver(%s): %s", driver_absolute_path, err?err:"unknown");
        return 0;
    }

    ALOGD("loaded %s", driver_absolute_path);

    if (mask & EGL) {
        getProcAddress = (getProcAddressType)dlsym(dso, "eglGetProcAddress");
```

```
        ALOGE_IF(!getProcAddress,
                "can't find eglGetProcAddress() in %s", driver_absolute_path);

#ifdef SYSTEMUI_PBSIZE_HACK
#warning "SYSTEMUI_PBSIZE_HACK enabled"
        /*
         * TODO: replace SYSTEMUI_PBSIZE_HACK by something less hackish
         *
         * Here we adjust the PB size from its default value to 512KB which
         * is the minimum acceptable for the systemui process.
         * We do this on low-end devices only because it allows us to enable
         * h/w acceleration in the systemui process while keeping the
         * memory usage down.
         *
         * Obviously, this is the wrong place and wrong way to make this
         * adjustment, but at the time of this writing this was the safest
         * solution.
         */
        const char *cmdline = getProcessCmdline();
        if (strstr(cmdline, "systemui")) {
            void *imgegl = dlopen("/vendor/lib/libIMGegl.so", RTLD_LAZY);
            if (imgegl) {
                unsigned int *PVRDefaultPBS =
                        (unsigned int *)dlsym(imgegl, "PVRDefaultPBS");
                if (PVRDefaultPBS) {
                    ALOGD("setting default PBS to 512KB, was %d KB", *PVRDefaultPBS / 1024);
                    *PVRDefaultPBS = 512*1024;
                }
            }
        }
#endif

        egl_t* egl = &cnx->egl;
        __eglMustCastToProperFunctionPointerType* curr =
            (__eglMustCastToProperFunctionPointerType*)egl;
        char const * const * api = egl_names;
        while (*api) {
            char const * name = *api;
            __eglMustCastToProperFunctionPointerType f =
                (__eglMustCastToProperFunctionPointerType)dlsym(dso, name);
            if (f == NULL) {
                // couldn't find the entry-point, use eglGetProcAddress()
                f = getProcAddress(name);
                if (f == NULL) {
                    f = (__eglMustCastToProperFunctionPointerType)0;
                }
            }
            *curr++ = f;
            api++;
        }
    }

    if (mask & GLESv1_CM) {
        init_api(dso, gl_names,
            (__eglMustCastToProperFunctionPointerType*)
                &cnx->hooks[egl_connection_t::GLESv1_INDEX]->gl,
            getProcAddress);
    }

    if (mask & GLESv2) {
      init_api(dso, gl_names,
            (__eglMustCastToProperFunctionPointerType*)
                &cnx->hooks[egl_connection_t::GLESv2_INDEX]->gl,
            getProcAddress);
    }

    return dso;
}
```

通过上述代码可知，函数 load_driver 会首先尝试在“/vendor/lib/egl/”目录下搜索库。如果没有找到，则尝试在“/system/lib/egl/”目录下进行搜索。并且函数 load_driver 还尝试解析出函数符号 eglGetProcAddress，这样可以在解析别的函数名未果的情况下，调用该函数尝试获得函数指针一次。

16.5 EGL 实现详解

在 Android 的 OpenGL ES 系统中，EGL 是用来管理绘图表面（Drawing surfaces），并且提供了如下所示的机制。

- 与本地窗口系统进行通信。
- 查找绘图表面可用的类型和配置信息。
- 创建绘图表面。
- 同步 OpenGL ES 和其他的渲染 API，包括 Open VG、本地窗口系统的绘图等命令。
- 管理渲染资源，比如材质。

在本节的内容中，将详细讲解在 Android 系统中 EGL 的具体实现过程。

16.5.1 分析 EGL 的数据结构

在文件 framework/base/opengl/libs/egl/egl_object.h 中定义了 EGL 的数据结构，具体实现代码如下所示。

```
namespace android {
// ----------------------------------------------------------------------------

struct egl_display_t;

class egl_object_t {
   egl_display_t *display;
   mutable volatile int32_t count;

protected:
   virtual ~egl_object_t();

public:
   egl_object_t(egl_display_t* display);
   void destroy();

   inline int32_t incRef() { return android_atomic_inc(&count); }
   inline int32_t decRef() { return android_atomic_dec(&count); }
   inline egl_display_t* getDisplay() const { return display; }

private:
   void terminate();
   static bool get(egl_display_t const* display, egl_object_t* object);

public:
   template <typename N, typename T>
   class LocalRef {
      egl_object_t* ref;
      LocalRef();
      LocalRef(const LocalRef* rhs);
   public:
      ~LocalRef();
      explicit LocalRef(egl_object_t* rhs);
      explicit LocalRef(egl_display_t const* display, T o) : ref(0) {
         egl_object_t* native = reinterpret_cast<N*>(o);
         if (o && egl_object_t::get(display, native)) {
            ref = native;
```

```
            }
        }
        inline N* get() {
            return static_cast<N*>(ref);
        }
        void acquire() const;
        void release() const;
        void terminate();
    };
    template <typename N, typename T>
    friend class LocalRef;
};

template<typename N, typename T>
egl_object_t::LocalRef<N, T>::LocalRef(egl_object_t* rhs) : ref(rhs) {
    if (ref) {
        ref->incRef();
    }
}

template <typename N, typename T>
egl_object_t::LocalRef<N,T>::~LocalRef() {
    if (ref) {
        ref->destroy();
    }
}

template <typename N, typename T>
void egl_object_t::LocalRef<N,T>::acquire() const {
    if (ref) {
        ref->incRef();
    }
}

template <typename N, typename T>
void egl_object_t::LocalRef<N,T>::release() const {
    if (ref) {
        if (ref->decRef() == 1) {
            // shouldn't happen because this is called from LocalRef
            ALOGE("LocalRef::release() removed the last reference!");
        }
    }
}

template <typename N, typename T>
void egl_object_t::LocalRef<N,T>::terminate() {
    if (ref) {
        ref->terminate();
    }
}

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

class egl_surface_t : public egl_object_t {
protected:
    ~egl_surface_t();
public:
    typedef egl_object_t::LocalRef<egl_surface_t, EGLSurface> Ref;

    egl_surface_t(egl_display_t* dpy, EGLConfig config,
            EGLNativeWindowType win, EGLSurface surface,
            egl_connection_t const* cnx);

    EGLSurface surface;
    EGLConfig config;
    sp<ANativeWindow> win;
    egl_connection_t const* cnx;
};
```

```
class egl_context_t: public egl_object_t {
protected:
    ~egl_context_t() {}
public:
    typedef egl_object_t::LocalRef<egl_context_t, EGLContext> Ref;

    egl_context_t(EGLDisplay dpy, EGLContext context, EGLConfig config,
            egl_connection_t const* cnx, int version);

    void onLooseCurrent();
    void onMakeCurrent(EGLSurface draw, EGLSurface read);

    EGLDisplay dpy;
    EGLContext context;
    EGLConfig config;
    EGLSurface read;
    EGLSurface draw;
    egl_connection_t const* cnx;
    int version;
    String8 gl_extensions;
};

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

typedef egl_surface_t::Ref  SurfaceRef;
typedef egl_context_t::Ref  ContextRef;

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

template<typename NATIVE, typename EGL>
static inline NATIVE* egl_to_native_cast(EGL arg) {
    return reinterpret_cast<NATIVE*>(arg);
}

static inline
egl_surface_t* get_surface(EGLSurface surface) {
    return egl_to_native_cast<egl_surface_t>(surface);
}

static inline
egl_context_t* get_context(EGLContext context) {
    return egl_to_native_cast<egl_context_t>(context);
}

// ----------------------------------------------------------------------------
}; // namespace android
// ----------------------------------------------------------------------------

#endif // ANDROID_EGL_OBJECT_H
```

在上述代码中，各个数据结构的具体说明如下所示。

- egl_object_t：用来描述没一个 elg 对象。
- egl_display_t：用来存储 get_display 函数获取的物理显示设备。
- egl_surface_t：用来存储 surface 对象，系统可以同时拥有多个 surface 对象。
- egl_context_t：用来存储 OpenGL 状态机信息。
- egl_image_t：用来存储 EGLImage 对象。

另外，在文件 framework/base/opengl/libagl/egl.cpp 中也定义了一些相关的结构体，具体实现代码如下所示。

```
struct egl_pixmap_surface_t : public egl_surface_t
{
    egl_pixmap_surface_t(
            EGLDisplay dpy, EGLConfig config,
            int32_t depthFormat,
```

```
        egl_native_pixmap_t const * pixmap);

    virtual ~egl_pixmap_surface_t() { }

    virtual     bool        initCheck() const { return !depth.format || depth.data!=0; }
    virtual     EGLBoolean  bindDrawSurface(ogles_context_t* gl);
    virtual     EGLBoolean  bindReadSurface(ogles_context_t* gl);
    virtual     EGLint      getWidth() const    { return nativePixmap.width;  }
    virtual     EGLint      getHeight() const   { return nativePixmap.height; }
private:
    egl_native_pixmap_t     nativePixmap;
};
struct egl_window_surface_v2_t : public egl_surface_t
{
    egl_window_surface_v2_t(
            EGLDisplay dpy, EGLConfig config,
            int32_t depthFormat,
            ANativeWindow* window);

    ~egl_window_surface_v2_t();

    virtual     bool        initCheck() const { return true; } // TODO: report failure if
ctor fails
    virtual     EGLBoolean  swapBuffers();
    virtual     EGLBoolean  bindDrawSurface(ogles_context_t* gl);
    virtual     EGLBoolean  bindReadSurface(ogles_context_t* gl);
    virtual     EGLBoolean  connect();
    virtual     void        disconnect();
    virtual     EGLint      getWidth() const    { return width;  }
    virtual     EGLint      getHeight() const   { return height; }
    virtual     EGLint      getHorizontalResolution() const;
    virtual     EGLint      getVerticalResolution() const;
    virtual     EGLint      getRefreshRate() const;
    virtual     EGLint      getSwapBehavior() const;
    virtual     EGLBoolean  setSwapRectangle(EGLint l, EGLint t, EGLint w, EGLint h);

private:
    status_t lock(ANativeWindowBuffer* buf, int usage, void** vaddr);
    status_t unlock(ANativeWindowBuffer* buf);
    ANativeWindow*   nativeWindow;
    ANativeWindowBuffer*   buffer;
    ANativeWindowBuffer*   previousBuffer;
    gralloc_module_t const*    module;
    int width;
    int height;
    void* bits;
    GGLFormat const* pixelFormatTable;

    struct Rect {
        inline Rect() { };
        inline Rect(int32_t w, int32_t h)
            : left(0), top(0), right(w), bottom(h) { }
        inline Rect(int32_t l, int32_t t, int32_t r, int32_t b)
            : left(l), top(t), right(r), bottom(b) { }
        Rect& andSelf(const Rect& r) {
            left   = max(left, r.left);
            top    = max(top, r.top);
            right  = min(right, r.right);
            bottom = min(bottom, r.bottom);
            return *this;
        }
        bool isEmpty() const {
            return (left>=right || top>=bottom);
        }
        void dump(char const* what) {
            ALOGD("%s { %5d, %5d, w=%5d, h=%5d }",
                    what, left, top, right-left, bottom-top);
        }
```

```
        int32_t left;
        int32_t top;
        int32_t right;
        int32_t bottom;
    };
struct egl_pbuffer_surface_t : public egl_surface_t
{
    egl_pbuffer_surface_t(
            EGLDisplay dpy, EGLConfig config, int32_t depthFormat,
            int32_t w, int32_t h, int32_t f);

    virtual ~egl_pbuffer_surface_t();

    virtual     bool        initCheck() const   { return pbuffer.data != 0; }
    virtual     EGLBoolean  bindDrawSurface(ogles_context_t* gl);
    virtual     EGLBoolean  bindReadSurface(ogles_context_t* gl);
    virtual     EGLint      getWidth() const    { return pbuffer.width;  }
    virtual     EGLint      getHeight() const   { return pbuffer.height; }
private:
    GGLSurface  pbuffer;
};
```

在上述代码中，各个结构体的具体说明如下所示。

● egl_window_surface_v2_t：继承于 egl_surface_t，提供了 egl_surface_t 功能的具体实现，属于可实际显示的 Surface。

● egl_pixmap_surface_t：用于存储保存在系统内存中的位图。

● egl_pbuffer_surface_t：用于存储保存在显存中的帧，以上两种位图属于不可显示的 Surface。

16.5.2　分析 EGL 的 API

在文件 framework/base/opengl/libagl/egl.cpp 中，定义了和 EGL 功能相关的功能函数，也就是我们在 Android 应用开发过程中用到的 API。各个功能函数的具体说明如下所示。

（1）EGLDisplay eglGetDisplay：功能是调用函数 egl_display_t::get_display(dpy)获取显示设备的句柄，具体实现代码如下所示。

```
EGLDisplay eglGetDisplay(NativeDisplayType display)
{
#ifndef HAVE_ANDROID_OS
    // this just needs to be done once
    if (gGLKey == -1) {
        pthread_mutex_lock(&gInitMutex);
        if (gGLKey == -1)
            pthread_key_create(&gGLKey, NULL);
        pthread_mutex_unlock(&gInitMutex);
    }
#endif
    if (display == EGL_DEFAULT_DISPLAY) {
        EGLDisplay dpy = (EGLDisplay)1;
        egl_display_t& d = egl_display_t::get_display(dpy);
        d.type = display;
        return dpy;
    }
    return EGL_NO_DISPLAY;
}
```

（2）EGLBoolean eglInitialize：功能是初始化 EGL，并获取 EGL 的版本号，具体实现代码如下所示。

```
EGLBoolean eglInitialize(EGLDisplay dpy, EGLint *major, EGLint *minor)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);
```

```
    EGLBoolean res = EGL_TRUE;
    egl_display_t& d = egl_display_t::get_display(dpy);

    if (android_atomic_inc(&d.initialized) == 0) {
        // initialize stuff here if needed
        //pthread_mutex_lock(&gInitMutex);
        //pthread_mutex_unlock(&gInitMutex);
    }

    if (res == EGL_TRUE) {
        if (major != NULL) *major = VERSION_MAJOR;
        if (minor != NULL) *minor = VERSION_MINOR;
    }
    return res;
}
```

（3）EGLBoolean eglTerminate：功能是结束一个 EGLDisplay，但是不结束 EGL 本身，具体实现代码如下所示。

```
EGLBoolean eglTerminate(EGLDisplay dpy)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);

    EGLBoolean res = EGL_TRUE;
    egl_display_t& d = egl_display_t::get_display(dpy);
    if (android_atomic_dec(&d.initialized) == 1) {
        // TODO: destroy all resources (surfaces, contexts, etc...)
        //pthread_mutex_lock(&gInitMutex);
        //pthread_mutex_unlock(&gInitMutex);
    }
    return res;
}
```

（4）EGLBoolean eglGetConfigs：功能是获取 EGL 配置参数，具体实现代码如下所示。

```
EGLBoolean eglGetConfigs(   EGLDisplay dpy,
                            EGLConfig *configs,
                            EGLint config_size, EGLint *num_config)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);

    GLint numConfigs = NELEM(gConfigs);
    if (!configs) {
        *num_config = numConfigs;
        return EGL_TRUE;
    }
    GLint i;
    for (i=0 ; i<numConfigs && i<config_size ; i++) {
        *configs++ = (EGLConfig)i;
    }
    *num_config = i;
    return EGL_TRUE;
}
```

（5）EGLBoolean eglChooseConfig：功能是选择 EGL 配置参数，配置一个期望并尽可能接近一个有效的系统配置，具体实现代码如下所示。

```
EGLBoolean eglChooseConfig( EGLDisplay dpy, const EGLint *attrib_list,
                            EGLConfig *configs, EGLint config_size,
                            EGLint *num_config)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);
```

```
    if (ggl_unlikely(num_config==0)) {
        return setError(EGL_BAD_PARAMETER, EGL_FALSE);
    }

    if (ggl_unlikely(attrib_list==0)) {
        /*
         * A NULL attrib_list should be treated as though it was an empty
         * one (terminated with EGL_NONE) as defined in
         * section 3.4.1 "Querying Configurations" in the EGL specification.
         */
        static const EGLint dummy = EGL_NONE;
        attrib_list = &dummy;
    }

    int numAttributes = 0;
    int numConfigs =  NELEM(gConfigs);
    uint32_t possibleMatch = (1<<numConfigs)-1;
    while(possibleMatch && *attrib_list != EGL_NONE) {
        numAttributes++;
        EGLint attr = *attrib_list++;
        EGLint val  = *attrib_list++;
        for (int i=0 ; possibleMatch && i<numConfigs ; i++) {
            if (!(possibleMatch & (1<<i)))
                continue;
            if (isAttributeMatching(i, attr, val) == 0) {
                possibleMatch &= ~(1<<i);
            }
        }
    }

    // now, handle the attributes which have a useful default value
    for (size_t j=0 ; possibleMatch && j<NELEM(config_defaults) ; j++) {
        // see if this attribute was specified, if not, apply its
        // default value
        if (binarySearch<config_pair_t>(
                (config_pair_t const*)attrib_list,
                0, numAttributes-1,
                config_defaults[j].key) < 0)
        {
            for (int i=0 ; possibleMatch && i<numConfigs ; i++) {
                if (!(possibleMatch & (1<<i)))
                    continue;
                if (isAttributeMatching(i,
                        config_defaults[j].key,
                        config_defaults[j].value) == 0)
                {
                    possibleMatch &= ~(1<<i);
                }
            }
        }
    }

    // return the configurations found
    int n=0;
    if (possibleMatch) {
        if (configs) {
            for (int i=0 ; config_size && i<numConfigs ; i++) {
                if (possibleMatch & (1<<i)) {
                    *configs++ = (EGLConfig)i;
                    config_size--;
                    n++;
                }
            }
        } else {
            for (int i=0 ; i<numConfigs ; i++) {
                if (possibleMatch & (1<<i)) {
                    n++;
                }
            }
```

```
        }
    }
    *num_config = n;
    return EGL_TRUE;
}
```

（6）EGLBoolean eglGetConfigAttrib：功能是获取 EGL 配置时需要参照的属性，具体实现代码如下所示。

```
EGLBoolean eglGetConfigAttrib(EGLDisplay dpy, EGLConfig config,
        EGLint attribute, EGLint *value)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);

    return getConfigAttrib(dpy, config, attribute, value);
}
```

（7）EGLSurface eglCreateWindowSurface：功能是创建一个 EGL surface，surface 是一个可以实际显示在屏幕上类型，具体实现代码如下所示。

```
EGLSurface eglCreateWindowSurface(  EGLDisplay dpy, EGLConfig config,
                                    NativeWindowType window,
                                    const EGLint *attrib_list)
{
    return createWindowSurface(dpy, config, window, attrib_list);
}
```

（8）EGLSurface eglCreatePixmapSurface：功能是创建 EGL PixmapSurface 和 PbufferSurface 类型，这两种类型不可直接显示与屏幕上，具体实现代码如下所示。

```
EGLSurface eglCreatePixmapSurface(  EGLDisplay dpy, EGLConfig config,
                                    NativePixmapType pixmap,
                                    const EGLint *attrib_list)
{
    return createPixmapSurface(dpy, config, pixmap, attrib_list);
}
```

（9）EGLBoolean eglDestroySurface：功能是销毁一个 EGL surface 对象，具体实现代码如下所示。

```
EGLBoolean eglDestroySurface(EGLDisplay dpy, EGLSurface eglSurface)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);
    if (eglSurface != EGL_NO_SURFACE) {
        egl_surface_t* surface( static_cast<egl_surface_t*>(eglSurface) );
        if (!surface->isValid())
            return setError(EGL_BAD_SURFACE, EGL_FALSE);
        if (surface->dpy != dpy)
            return setError(EGL_BAD_DISPLAY, EGL_FALSE);
        if (surface->ctx) {
            // defer disconnect/delete until no longer current
            surface->zombie = true;
        } else {
            surface->disconnect();
            delete surface;
        }
    }
    return EGL_TRUE;
}
```

（10）EGLBoolean eglQuerySurface：功能是查询 surface 的参数，具体实现代码如下所示。

```
EGLBoolean eglQuerySurface( EGLDisplay dpy, EGLSurface eglSurface,
                        EGLint attribute, EGLint *value)
{
```

```
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);
    egl_surface_t* surface = static_cast<egl_surface_t*>(eglSurface);
    if (!surface->isValid())
        return setError(EGL_BAD_SURFACE, EGL_FALSE);
    if (surface->dpy != dpy)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);

    EGLBoolean ret = EGL_TRUE;
    switch (attribute) {
        case EGL_CONFIG_ID:
            ret = getConfigAttrib(dpy, surface->config, EGL_CONFIG_ID, value);
            break;
        case EGL_WIDTH:
            *value = surface->getWidth();
            break;
        case EGL_HEIGHT:
            *value = surface->getHeight();
            break;
        case EGL_LARGEST_PBUFFER:
            // not modified for a window or pixmap surface
            break;
        case EGL_TEXTURE_FORMAT:
            *value = EGL_NO_TEXTURE;
            break;
        case EGL_TEXTURE_TARGET:
            *value = EGL_NO_TEXTURE;
            break;
        case EGL_MIPMAP_TEXTURE:
            *value = EGL_FALSE;
            break;
        case EGL_MIPMAP_LEVEL:
            *value = 0;
            break;
        case EGL_RENDER_BUFFER:
            // TODO: return the real RENDER_BUFFER here
            *value = EGL_BACK_BUFFER;
            break;
        case EGL_HORIZONTAL_RESOLUTION:
            // pixel/mm * EGL_DISPLAY_SCALING
            *value = surface->getHorizontalResolution();
            break;
        case EGL_VERTICAL_RESOLUTION:
            // pixel/mm * EGL_DISPLAY_SCALING
            *value = surface->getVerticalResolution();
            break;
        case EGL_PIXEL_ASPECT_RATIO: {
            // w/h * EGL_DISPLAY_SCALING
            int wr = surface->getHorizontalResolution();
            int hr = surface->getVerticalResolution();
            *value = (wr * EGL_DISPLAY_SCALING) / hr;
        } break;
        case EGL_SWAP_BEHAVIOR:
            *value = surface->getSwapBehavior();
            break;
        default:
            ret = setError(EGL_BAD_ATTRIBUTE, EGL_FALSE);
    }
    return ret;
}
```

（11）EGLContext eglCreateContext：功能是创建一个 OpenGL 状态，具体实现代码如下所示。

```
EGLContext eglCreateContext(EGLDisplay dpy, EGLConfig config,
                            EGLContext share_list, const EGLint *attrib_list)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_NO_SURFACE);

    ogles_context_t* gl = ogles_init(sizeof(egl_context_t));
```

```
    if (!gl) return setError(EGL_BAD_ALLOC, EGL_NO_CONTEXT);

    egl_context_t* c = static_cast<egl_context_t*>(gl->rasterizer.base);
    c->flags = egl_context_t::NEVER_CURRENT;
    c->dpy = dpy;
    c->config = config;
    c->read = 0;
    c->draw = 0;
    return (EGLContext)gl;
}
```

（12）EGLBoolean eglDestroyContext：功能是销毁一个 OpenGL 状态，具体实现代码如下所示。

```
EGLBoolean eglDestroyContext(EGLDisplay dpy, EGLContext ctx)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);
    egl_context_t* c = egl_context_t::context(ctx);
    if (c->flags & egl_context_t::IS_CURRENT)
        setGlThreadSpecific(0);
    ogles_uninit((ogles_context_t*)ctx);
    return EGL_TRUE;
}
```

（13）EGLBoolean eglMakeCurrent：功能是将 OpenGL context 与 surface 绑定，具体实现代码如下所示。

```
EGLBoolean eglMakeCurrent(  EGLDisplay dpy, EGLSurface draw,
                            EGLSurface read, EGLContext ctx)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);
    if (draw) {
        egl_surface_t* s = (egl_surface_t*)draw;
        if (!s->isValid())
            return setError(EGL_BAD_SURFACE, EGL_FALSE);
        if (s->dpy != dpy)
            return setError(EGL_BAD_DISPLAY, EGL_FALSE);
        // TODO: check that draw is compatible with the context
    }
    if (read && read!=draw) {
        egl_surface_t* s = (egl_surface_t*)read;
        if (!s->isValid())
            return setError(EGL_BAD_SURFACE, EGL_FALSE);
        if (s->dpy != dpy)
            return setError(EGL_BAD_DISPLAY, EGL_FALSE);
        // TODO: check that read is compatible with the context
    }

    EGLContext current_ctx = EGL_NO_CONTEXT;

    if ((read == EGL_NO_SURFACE && draw == EGL_NO_SURFACE) && (ctx != EGL_NO_CONTEXT))
        return setError(EGL_BAD_MATCH, EGL_FALSE);

    if ((read != EGL_NO_SURFACE || draw != EGL_NO_SURFACE) && (ctx == EGL_NO_CONTEXT))
        return setError(EGL_BAD_MATCH, EGL_FALSE);

    if (ctx == EGL_NO_CONTEXT) {
        // if we're detaching, we need the current context
        current_ctx = (EGLContext)getGlThreadSpecific();
    } else {
        egl_context_t* c = egl_context_t::context(ctx);
        egl_surface_t* d = (egl_surface_t*)draw;
        egl_surface_t* r = (egl_surface_t*)read;
        if ((d && d->ctx && d->ctx != ctx) ||
            (r && r->ctx && r->ctx != ctx)) {
            // one of the surface is bound to a context in another thread
            return setError(EGL_BAD_ACCESS, EGL_FALSE);
```

```
        }
    }

    ogles_context_t* gl = (ogles_context_t*)ctx;
    if (makeCurrent(gl) == 0) {
        if (ctx) {
            egl_context_t* c = egl_context_t::context(ctx);
            egl_surface_t* d = (egl_surface_t*)draw;
            egl_surface_t* r = (egl_surface_t*)read;

            if (c->draw) {
                egl_surface_t* s = reinterpret_cast<egl_surface_t*>(c->draw);
                s->disconnect();
                s->ctx = EGL_NO_CONTEXT;
                if (s->zombie)
                    delete s;
            }
            if (c->read) {
                // FIXME: unlock/disconnect the read surface too
            }

            c->draw = draw;
            c->read = read;

            if (c->flags & egl_context_t::NEVER_CURRENT) {
                c->flags &= ~egl_context_t::NEVER_CURRENT;
                GLint w = 0;
                GLint h = 0;
                if (draw) {
                    w = d->getWidth();
                    h = d->getHeight();
                }
                ogles_surfaceport(gl, 0, 0);
                ogles_viewport(gl, 0, 0, w, h);
                ogles_scissor(gl, 0, 0, w, h);
            }
            if (d) {
                if (d->connect() == EGL_FALSE) {
                    return EGL_FALSE;
                }
                d->ctx = ctx;
                d->bindDrawSurface(gl);
            }
            if (r) {
                // FIXME: lock/connect the read surface too
                r->ctx = ctx;
                r->bindReadSurface(gl);
            }
        } else {
            // if surfaces were bound to the context bound to this thread
            // mark then as unbound
            if (current_ctx) {
                egl_context_t* c = egl_context_t::context(current_ctx);
                egl_surface_t* d = (egl_surface_t*)c->draw;
                egl_surface_t* r = (egl_surface_t*)c->read;
                if (d) {
                    c->draw = 0;
                    d->disconnect();
                    d->ctx = EGL_NO_CONTEXT;
                    if (d->zombie)
                        delete d;
                }
                if (r) {
                    c->read = 0;
                    r->ctx = EGL_NO_CONTEXT;
                    // FIXME: unlock/disconnect the read surface too
                }
            }
        }
```

```
        return EGL_TRUE;
    }
    return setError(EGL_BAD_ACCESS, EGL_FALSE);
}
```

（14）EGLBoolean eglQueryContext：功能是查询 context 的参数，具体实现代码如下所示。

```
EGLBoolean eglQueryContext( EGLDisplay dpy, EGLContext ctx,
                        EGLint attribute, EGLint *value)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);
    egl_context_t* c = egl_context_t::context(ctx);
    switch (attribute) {
        case EGL_CONFIG_ID:
            // Returns the ID of the EGL frame buffer configuration with
            // respect to which the context was created
            return getConfigAttrib(dpy, c->config, EGL_CONFIG_ID, value);
    }
    return setError(EGL_BAD_ATTRIBUTE, EGL_FALSE);
}
```

（15）EGLBoolean eglSwapBuffers：功能是绘制完图形后用于显示的函数，具体实现代码如下所示。

```
EGLBoolean eglSwapBuffers(EGLDisplay dpy, EGLSurface draw)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);

    egl_surface_t* d = static_cast<egl_surface_t*>(draw);
    if (!d->isValid())
        return setError(EGL_BAD_SURFACE, EGL_FALSE);
    if (d->dpy != dpy)
        return setError(EGL_BAD_DISPLAY, EGL_FALSE);

    // post the surface
    d->swapBuffers();

    // if it's bound to a context, update the buffer
    if (d->ctx != EGL_NO_CONTEXT) {
        d->bindDrawSurface((ogles_context_t*)d->ctx);
        // if this surface is also the read surface of the context
        // it is bound to, make sure to update the read buffer as well
        // The EGL spec is a little unclear about this
        egl_context_t* c = egl_context_t::context(d->ctx);
        if (c->read == draw) {
            d->bindReadSurface((ogles_context_t*)d->ctx);
        }
    }

    return EGL_TRUE;
}
```

（16）const char* eglQueryString：功能是查询 EGL 参数字串，具体实现代码如下所示。

```
const char* eglQueryString(EGLDisplay dpy, EGLint name)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE)
        return setError(EGL_BAD_DISPLAY, (const char*)0);

    switch (name) {
        case EGL_VENDOR:
            return gVendorString;
        case EGL_VERSION:
            return gVersionString;
        case EGL_EXTENSIONS:
            return gExtensionsString;
```

```
        case EGL_CLIENT_APIS:
            return gClientApiString;
    }
    return setError(EGL_BAD_PARAMETER, (const char *)0);
}
```

（17）EGLImageKHR eglCreateImageKHR：功能是将目标处理对象封装成 EGLImageKHR，以便于实现纹理操作，具体实现代码如下所示。

```
EGLImageKHR eglCreateImageKHR(EGLDisplay dpy, EGLContext ctx, EGLenum target,
        EGLClientBuffer buffer, const EGLint *attrib_list)
{
    if (egl_display_t::is_valid(dpy) == EGL_FALSE) {
        return setError(EGL_BAD_DISPLAY, EGL_NO_IMAGE_KHR);
    }
    if (ctx != EGL_NO_CONTEXT) {
        return setError(EGL_BAD_CONTEXT, EGL_NO_IMAGE_KHR);
    }
    if (target != EGL_NATIVE_BUFFER_ANDROID) {
        return setError(EGL_BAD_PARAMETER, EGL_NO_IMAGE_KHR);
    }

    ANativeWindowBuffer* native_buffer = (ANativeWindowBuffer*)buffer;

    if (native_buffer->common.magic != ANDROID_NATIVE_BUFFER_MAGIC)
        return setError(EGL_BAD_PARAMETER, EGL_NO_IMAGE_KHR);

    if (native_buffer->common.version != sizeof(ANativeWindowBuffer))
        return setError(EGL_BAD_PARAMETER, EGL_NO_IMAGE_KHR);

    switch (native_buffer->format) {
        case HAL_PIXEL_FORMAT_RGBA_8888:
        case HAL_PIXEL_FORMAT_RGBX_8888:
        case HAL_PIXEL_FORMAT_RGB_888:
        case HAL_PIXEL_FORMAT_RGB_565:
        case HAL_PIXEL_FORMAT_BGRA_8888:
        case HAL_PIXEL_FORMAT_RGBA_5551:
        case HAL_PIXEL_FORMAT_RGBA_4444:
            break;
        default:
            return setError(EGL_BAD_PARAMETER, EGL_NO_IMAGE_KHR);
    }

    native_buffer->common.incRef(&native_buffer->common);
    return (EGLImageKHR)native_buffer;
}
```

第 17 章 OpenGL ES 基本应用

在 Android 系统中，使用 OpenGL ES 的目的主要是构建三维效果。在 OpenGL ES 中三维效果都是通过构建三角形实现的，并且结合使用投影功能使三维效果更加逼真。在本章的内容中，将详细讲解在 Android 系统中使用 OpenGL ES 实现投影和光照功能的基本知识，为读者进入后面知识的学习打下基础。

17.1 OpenGL ES 的基本应用

在 Android 系统中，使用 OpenGL ES 的目的主要是构建三维效果。在 OpenGL ES 开发应用中，三维效果都是通过构建三角形实现的。在本节的内容中，将详细介绍使用 OpenGL ES 技术绘制三角形的方法。

17.1.1 使用点线法绘制三角形

在 Android 系统中，使用 OpenGL ES 绘制三角形的方法有多种，其中最为常用的方法如下所示。

（1）GL_POINTS。

把每个顶点作为一个点进行处理，索引数组中的第 *z* 个顶点即定义了点 *z*，共绘制 *n* 个点。例如，索引数组{0，1，2，3，4}。

（2）GL_INES。

把每两个顶点作为一条独立的线段面，索引数组中的第 2*n* 和第 2*n*+1 个顶点定义了第 *n* 条线段，总共绘制了 *n*/2 条线段。如果 *n* 为奇数，则忽略最后一个顶点。例如，索引数组{0，3，2，1}。

（3）GL_LINE_STRIF。

绘制索引数组中从第 0 个顶点到最后一个顶点依次相连的一组线段，第 *n* 个和第 *n*+1 个顶点定义了线段，总共绘制 *n*-1 条线段。例如，索引数组{0，3，2，1}。

（4）GL_LINE_LOOP。

绘制索引数组中从第 0 个顶点到最后一个顶点依次相连的一组线段，最终最后一个顶点与第 0 个顶点相连。第 *n* 个和第 *n*+1 个顶点定义了线段 *n*，最后一条线段是由顶点 *n*–1 和 0 之间定义，总共绘制 *n* 条线段。例如，索引数组{0，3，2，1}。

（5）GL_TRIANGLES。

把索引数组中的每 3 个顶点作为一个独立三角形。索引数组中第 3*n* 个、3*n*+1 个和 3*n*+2 个顶点定义了第 *n* 个三角形，总共绘制 *n*/3 个三角形。例如，索引数组{0，1，2，2，1，3}。

（6）GL_TRIANGLE_STRIP。

此方式用于绘制一组相连的三角形。对于索引数组中的第 *n* 个点，如果行为奇数，则第 *n*+1 个、第 *n*+2 个顶点定义了第 *n* 个三角形；如果行为偶数，则第 *n* 个、第 *n*+1 个和第 *n*+2 个顶点定义了第 *n* 个三角形。总共绘制 *n*-2 个三角形。例如，索引数组{0，1，2，3，4}。

（7）GL_TRIANGLE_FAN\。

绘制一组相连的三角形。三角形是由索引数组中的第 0 个顶点及其后给定的顶点所确定。顶点 0、*n*+1 和 *n*+2 定义了第 *n* 个三角形，一总共绘制 *n*–2 个三角形。例如索引数组{0，1，2，3，4}。

在接下来的内容中，将通过一个具体的实例向读者演示使用点线法绘制三角形的方法。

题目	目的	源码路径
实例 17-1	使用点线方法绘制三角形	\daima\17\threeCH

本实例的实现流程如下所示。

（1）编写布局文件 main.xml，设置垂直方向布局和线型布局的 ID，主要实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:id="@+id/main_liner"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
</LinearLayout>
```

（2）编写文件 MyActivity.java，用于重写 onCreate()方法，在创建时为 Activity 设置布局，在暂停时保存 mSurfaceView，在恢复时恢复 mSurfaceView，主要实现代码如下所示。

```
public class MyActivity extends Activity {
   private MySurfaceView mSurfaceView;
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      mSurfaceView=new MySurfaceView(this);//创建 MySurfaceView 对象
      mSurfaceView.requestFocus();//获取焦点
      mSurfaceView.setFocusableInTouchMode(true);//设置可触控模式
      LinearLayout ll=(LinearLayout)this.findViewById(R.id.main_liner);//获得对线性布
局的引用
      ll.addView(mSurfaceView);
   }
   @Override
   protected void onPause() {
      super.onPause();
      mSurfaceView.onPause();
   }
   @Override
   protected void onResume() {
      super.onResume();
      mSurfaceView.onResume();
   }
}
```

（3）编写文件 MySurfaceView.java，首先引入相关类及自定义视图来加载图像，然后角度缩放比例，并重写触控事件的回调方法来计算在屏幕上滑动多少距离对应物体应该旋转多少度，最后定义渲染器类，实现其内部的相关方法来渲染场景。文件 MySurfaceView.java 的主要实现代码如下所示。

```
public class MySurfaceView extends GLSurfaceView {
   //设置角度缩放比例，即屏幕宽 320，从屏幕的一端滑到另一端，x 轴上的差距对应相应的需要旋转的角度
   private final float TOUCH_SCALE_FACTOR=180.0f/320;
   private SceneRenderer myRenderer;                  //设置场景渲染器
   private float myPreviousY;                         //屏幕触控位置的 y 坐标
   private float myPreviousX;                         //屏幕触控位置的 x 坐标
   public MySurfaceView(Context context) {
      super(context);
      myRenderer=new SceneRenderer();
```

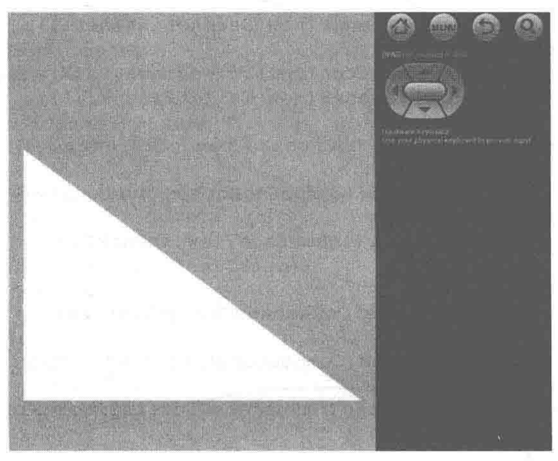
```
        this.setRenderer(myRenderer);
        this.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
//设置渲染模式为主动渲染
    }
    //触摸事件回调方法
    public boolean onTouchEvent(MotionEvent event) {
        // TODO Auto-generated method stub
        float y=event.getY();                          //获得当前触点的 y 坐标
        float x=event.getX();                          //获得当前触点的 x 坐标
        switch(event.getAction()){
        case MotionEvent.ACTION_MOVE:
            float dy=y-myPreviousY;                    //滑动距离在 y 轴方向上的垂直距离
            float dx=x-myPreviousX;                    //活动距离在 x 轴方向上的垂直距离
            myRenderer.tr.yAngle+=dx*TOUCH_SCALE_FACTOR;//设置沿 y 轴旋转角度
            myRenderer.tr.zAngle+=dy*TOUCH_SCALE_FACTOR;//设置沿 z 轴旋转角度
            requestRender();                                //渲染画面
        }
        myPreviousY=y;
        myPreviousX=x;
        return true;
    }
//内部类，实现 Renderer 接口，渲染器
    private class SceneRenderer  implements GLSurfaceView.Renderer{
        Triangle tr=new Triangle();
        public SceneRenderer(){
        }
        @Override
        public void onDrawFrame(GL10 gl) {
            gl.glEnable(GL10.GL_CULL_FACE);
            gl.glShadeModel(GL10.GL_SMOOTH);
            gl.glFrontFace(GL10.GL_CCW);
            //分别清除颜色缓存和深度缓存
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT|GL10.GL_DEPTH_BUFFER_BIT);
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
            gl.glTranslatef(0, 0, -2.0f);
            tr.drawSelf(gl);
        }
        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
             gl.glViewport(0, 0, width, height);
             gl.glMatrixMode(GL10.GL_PROJECTION);
             gl.glLoadIdentity();
             float ratio=(float)width/height;
             gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);
        }
        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            gl.glDisable(GL10.GL_DITHER);                  //关闭抗抖动
            gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
            gl.glClearColor(0, 255, 255, 0);               //设置屏幕背景色蓝色
            gl.glEnable(GL10.GL_DEPTH_TEST);               //启用深度检测
        }}}
```

（4）编写文件 threeCH.java，首先在此定义类 threeCH 来绘制图形，然后初始化三角形的顶点数据缓冲和颜色数据缓冲，并创建整型类型的顶点数据数组，最后定义应用程序中各个实现场景物体的绘制方法。文件 threeCH.java 的主要实现代码如下所示。

```
public class threeCH {
    private IntBuffer myVertexBuffer;
    private IntBuffer myColorBuffer;
    private ByteBuffer myIndexBuffer;
    int vCount=0;                               //初始顶点数量
    int iCount=0;                               //初始索引数量
    float yAngle=0;                             //初始绕 y 轴旋转的角度
    float zAngle=0;                             //初始绕 z 轴旋转的角度
    public threeCH(){
        vCount=3;                               //一个三角形，3 个顶点
```

```
        final int UNIT_SIZE=10000;          //缩放比例
        int []vertices=new int[]
           {
               -8*UNIT_SIZE,6*UNIT_SIZE,0,
               -8*UNIT_SIZE,-6*UNIT_SIZE,0,
               8*UNIT_SIZE,-6*UNIT_SIZE,0
           };
        //创建顶点坐标数据缓存，在此必须经过 ByteBuffer 转换
        ByteBuffer vbb=ByteBuffer.allocateDirect(vertices.length*4);
        vbb.order(ByteOrder.nativeOrder());
        myVertexBuffer=vbb.asIntBuffer();
        myVertexBuffer.put(vertices);
        myVertexBuffer.position(0);
        final int one=65535;
        int []colors=new int[]
        {
           one,one,one,0,
           one,one,one,0,
           one,one,one,0
        };
        ByteBuffer cbb=ByteBuffer.allocateDirect(colors.length*4);
        cbb.order(ByteOrder.nativeOrder());
        myColorBuffer=cbb.asIntBuffer();
        myColorBuffer.put(colors);
        myColorBuffer.position(0);
        //为三角形构造索引数据初始化
        iCount=3;
        byte []indices=new byte[]
            {
               0,1,2
            };
        //创建三角形构造索引数据缓冲
        myIndexBuffer=ByteBuffer.allocateDirect(indices.length);
        myIndexBuffer.put(indices);
        myIndexBuffer.position(0);
    }
//设置 GL10 表示实现接口 GL 的一个公共接口，在里面包含了一系列常量和抽象方法
    public void drawSelf(GL10 gl)
    {
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY); //启用顶点坐标数组
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);  //启用顶点颜色数组

        gl.glRotatef(yAngle,0,1,0);                   //根据 yAngle 的角度值，绕 y 轴旋转 yAngle
        gl.glRotatef(zAngle,0,0,1);

        gl.glVertexPointer                            //为画笔指定顶点坐标数据
        (
               3,
               GL10.GL_FIXED,
               0,
               myVertexBuffer
        );
        gl.glColorPointer                             //为画笔指定顶点 颜色数据
        (
           6,
           GL10.GL_FIXED,
           0,
           myColorBuffer
        );
        gl.glDrawElements//绘制图形
        (
           GL10.GL_TRIANGLES,                        //填充模式，这里是以三角形方式填充
           iCount,                                   //顶点数量
           GL10.GL_UNSIGNED_BYTE,                    //索引值的类型
           myIndexBuffer                             //索引值数据
        );
    }}
```

执行后将显示一个青色屏幕背景、颜色为白色的直角三角形。执行效果如图 17-1 所示。



▲图 17-1 使用点线法绘制三角形的执行效果

17.1.2 使用索引法绘制三角形

索引法是指通过调用 gl.glDrawElements()方法来绘制各种基本几何图形。在 OpenGL ES 中，方法 glDrawElements()的语法格式如下所示。

```
glDrawElements(int mode, int count, int type, Buffer indices)
```

- mode：定义画什么样的图元。
- count：定义一共有多少个索引值。
- type：定义索引数组使用的类型。
- indices：绘制顶点使用的索引缓存。

接下来将通过一个具体实例的实现流程，详细讲解使用索引法绘制三角形的方法。

题目	目的	源码路径
实例 17-2	使用索引法绘制三角形	\daima\17\suoyinCH

本实例的实现流程如下所示。

（1）编写文件 MyActivity.java，具体实现流程如下所示。

- 先引入相关包，并声明了 MySurfaceView 对象。
- 为布局文件中的按钮添加的监听器类，分别用于监听不同的 3 个按钮。
- 重写 onPause()继承父类的方法，并同时挂起或恢复 MySurfaceView 视图。

文件 MyActivity.java 的主要实现代码如下所示。

```
public class MyActivity extends Activity {
    private MySurfaceView mSurfaceView;     //声明 MySurfaceView 对象
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);                     //布局文件
        mSurfaceView=new MySurfaceView(this);
        mSurfaceView.requestFocus();                       //获取焦点
        mSurfaceView.setFocusableInTouchMode(true);        //设置为可触控
        //获得线性布局的引用
        LinearLayout ll=(LinearLayout)this.findViewById(R.id.main_liner);
        ll.addView(mSurfaceView);
        //获得第一个开关按钮的引用
        ToggleButton tb01=(ToggleButton)this.findViewById(R.id.ToggleButton01);
        tb01.setOnCheckedChangeListener(new FirstListener());
        //获得第二个开关按钮的引用
        ToggleButton tb02=(ToggleButton)this.findViewById(R.id.ToggleButton02);
```

```
        tb02.setOnCheckedChangeListener(new SecondListener());
        //获得第三个开关按钮的引用
        ToggleButton tb03=(ToggleButton)this.findViewById(R.id.ToggleButton03);
        tb03.setOnCheckedChangeListener(new ThirdListener());
    }
    class FirstListener implements OnCheckedChangeListener{
        @Override
        public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
            mSurfaceView.setBackFlag(!mSurfaceView.isBackFlag());
        }
    }
    class SecondListener implements OnCheckedChangeListener{     //声明第二个按钮的监听器
        @Override
        public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
            mSurfaceView.setSmoothFlag(!mSurfaceView.isSmoothFlag());
        }
    }
    class ThirdListener implements OnCheckedChangeListener{      //声明第三个按钮的监听器
        @Override
        public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
            mSurfaceView.setSelfCulling(!mSurfaceView.isSelfCulling());
        }
    }
    @Override
    protected void onPause() {
        super.onPause();
        mSurfaceView.onPause();
    }
    @Override
    protected void onResume() {
        super.onResume();
        mSurfaceView.onResume();
    }
}
```

（2）编写文件 MySurfaceView.java，具体实现流程如下所示。

- 在创建 MySurfaceView 对象的同时设置渲染器和渲染模式。
- 设置背面剪裁、平滑着色、自定义卷绕标志位的方法。
- 定义了触摸回调方法以实现屏幕触控，并在屏幕上滑动而使场景物体旋转的功能。
- 定义渲染器内部类以实现图像的渲染、屏幕横竖发生变化时的功能。
- 重写 onDrawFrame()方法，分别实现背面剪裁、平滑着色功能，并在屏幕横竖空间位置发生变化时自动调用。
- 当 MySurfaceView 创建时被调用以初始化屏幕背景颜色、绘制模式、是否深度检测等。

文件 MySurfaceView.java 的主要实现代码如下所示。

```
public class MySurfaceView extends GLSurfaceView {
    private final float TOUCH_SCALE_FACTOR=180.0f/320;      //设置角度缩放比例
    private SceneRenderer myRenderer;                        //场景渲染器
    private boolean backFlag=false;                          //设置是否打开背面剪裁标志
    private boolean smoothFlag=false;                        //设置是否打开平面着色标志
    private boolean selfCulling=false;
    private float myPreviousY;
    private float myPreviousX;
    public MySurfaceView(Context context) {
        super(context);
        myRenderer=new SceneRenderer();
        this.setRenderer(myRenderer);                        //设置渲染器
        this.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);//渲染模式为主动渲染
    }
    public void setBackFlag(boolean flag){
        this.backFlag=flag;
```

```
    }
    public boolean isBackFlag(){
        return backFlag;
    }
    public void setSmoothFlag(boolean flag){
        this.smoothFlag=flag;
    }
    public boolean isSmoothFlag(){
        return smoothFlag;
    }
    public void setSelfCulling(boolean flag){
        this.selfCulling=flag;
    }
    public boolean isSelfCulling(){
        return selfCulling;
    }
    //触摸事件回调方法
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        // TODO Auto-generated method stub
        float y=event.getY();
        float x=event.getX();
        switch(event.getAction()){
        case MotionEvent.ACTION_MOVE:
            float dy=y-myPreviousY;
            float dx=x-myPreviousX;
            myRenderer.tp.yAngle+=dx*TOUCH_SCALE_FACTOR;
            myRenderer.tp.zAngle+=dy*TOUCH_SCALE_FACTOR;
            requestRender();
        }
        myPreviousY=y;
        myPreviousX=x;
        return true;
    }
    private class SceneRenderer  implements GLSurfaceView.Renderer{
        suoyinCH tp=new suoyinCH();
        public SceneRenderer(){}
        @Override
        public void onDrawFrame(GL10 gl) {
            if(backFlag){
                gl.glEnable(GL10.GL_CULL_FACE);            //打开背面剪裁
            }
            else{
                gl.glDisable(GL10.GL_CULL_FACE);           //关闭背面剪裁
            }

            if(smoothFlag){
                gl.glShadeModel(GL10.GL_SMOOTH);           //平滑着色
            }
            else{
                gl.glShadeModel(GL10.GL_FLAT);             //不平滑着色
            }

            if(selfCulling){
                gl.glFrontFace(GL10.GL_CW);                //自定义卷绕顺序为顺时针，为正面
            }
            else{
                gl.glFrontFace(GL10.GL_CCW);               //自定义卷绕顺序为逆时针，为正面
            }
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT|GL10.GL_DEPTH_BUFFER_BIT);
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
            gl.glTranslatef(0, 0, -2.0f);
            tp.drawSelf(gl);
        }
        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
            // TODO Auto-generated method stub
             gl.glViewport(0, 0, width, height);
```

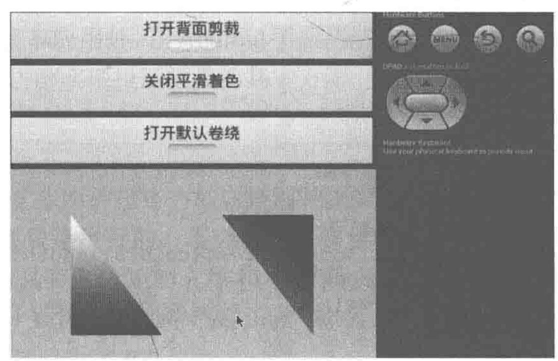
```
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        float ratio=(float)width/height;
        gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);
    }
    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
       gl.glDisable(GL10.GL_DITHER);                        //关闭抗抖动
       gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
       gl.glClearColor(0, 255, 255, 0);                     //设置屏幕背景色为青色
       gl.glEnable(GL10.GL_DEPTH_TEST);                     //启用深度检测机制
    }}}
```

（3）编写文件 suoyinCH.java，定义 suoyinCH 类的构造器来初始化相关数据，这些数据包括初始化三角形的顶点数据缓冲、颜色数据缓冲、索引数据缓冲。然后定义应用程序中具体实现场景物体的绘制方法，主要包括启用相应数组、旋转场景中物体、指定画笔的顶点坐标数据和顶点颜色数据，并用画笔实现绘图功能。文件 suoyinCH.java 的主要实现代码如下所示。

```
public class suoyinCH {
   private IntBuffer myVertexBuffer;
   private IntBuffer myColorBuffer;
   private ByteBuffer myIndexBuffer;
   int vCount=0;
   int iCount=0;
   float yAngle=0;                                   //绕 y 轴旋转的角度
   float zAngle=0;                                   //绕 z 轴旋转的角度
   public suoyinCH(){
      vCount=6;
      final int UNIT_SIZE=10000;                     //缩放比例
      int []vertices=new int[]{
             -8*UNIT_SIZE,10*UNIT_SIZE,0,
             -2*UNIT_SIZE,2*UNIT_SIZE,0,
             -8*UNIT_SIZE,2*UNIT_SIZE,0,
             8*UNIT_SIZE,2*UNIT_SIZE,0,
             8*UNIT_SIZE,10*UNIT_SIZE,0,
             2*UNIT_SIZE,10*UNIT_SIZE,0
         };
      //创建顶点坐标数据缓存
      ByteBuffer vbb=ByteBuffer.allocateDirect(vertices.length*4);//内存块
      vbb.order(ByteOrder.nativeOrder());
      myVertexBuffer=vbb.asIntBuffer();
      myVertexBuffer.put(vertices);
      myVertexBuffer.position(0);
      final int one=65535;
      int []colors=new int[]{
             one,one,one,0,
             0,0,one,0,
             0,0,one,0,
             one,0,one,0,
             0,0,0,0,
             one,0,0,0
      };
      ByteBuffer cbb=ByteBuffer.allocateDirect(colors.length*4);
      cbb.order(ByteOrder.nativeOrder());
      myColorBuffer=cbb.asIntBuffer();
      myColorBuffer.put(colors);
      myColorBuffer.position(0);                     //设置缓冲区的起始位置
      //为三角形构造索引数据初始化
      iCount=6;
      byte []indices=new byte[]{
            0,1,2,
            3,4,5
         };
      //创建三角形构造索引数据缓冲
      myIndexBuffer=ByteBuffer.allocateDirect(indices.length);
      myIndexBuffer.put(indices);                    //向缓冲区中放入顶点索引数据
      myIndexBuffer.position(0);                     //设置缓冲区的起始位置
```

```
    }
    public void drawSelf(GL10 gl){
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        gl.glRotatef(yAngle,0,1,0);
        gl.glRotatef(zAngle,0,0,1);
        gl.glVertexPointer(
                3,
                GL10.GL_FIXED,
                0,
                myVertexBuffer
        );
        gl.glColorPointer(
            4,
            GL10.GL_FIXED,
            0,
            myColorBuffer
        );
        gl.glDrawElements(
            GL10.GL_TRIANGLES,
            iCount,
            GL10.GL_UNSIGNED_BYTE,
            myIndexBuffer
        );
    }
}
```

到此为止，整个实例介绍完毕，执行后将显示青色背景的屏幕。在屏幕上方显示 3 个控制按钮，通过按钮可以设置屏幕下方的两个三角形的显示模式。执行效果如图 17-2 所示。



▲图 17-2　使用三角形绘制三角形的执行效果

17.1.3　使用顶点法绘制三角形

在 OpenGL ES 系统中，可以使用顶点法在 Android 屏幕中绘制三角形。顶点法的绘制过程与索引法的绘制过程比较类似，唯一的区别是顶点法没有建立索引缓冲，只是直接在建立顶点缓冲的同时将顶点数组中的顶点排好顺序，最后在调用绘制方法时使用顶点缓冲。

接下来将通过一个具体实例的实现流程，详细讲解使用顶点法绘制三角形的方法。

题目	目的	源码路径
实例 17-3	使用顶点法绘制三角形	\daima\17\dingCH

本实例的实现流程如下所示。

（1）分别编写文件 MyActivity.java 和 MySurfaceView.java，具体实现原理和实例 17-2 类似。

（2）编写文件 dingCH.java，具体实现流程如下所示。

- 先定义要绘制的 dingCH 类，并声明相关变量。

● 定义 dingCH 类的构造器来初始化相关数据，这些数据包括初始化三角形的顶点数据缓冲、颜色数据缓冲。

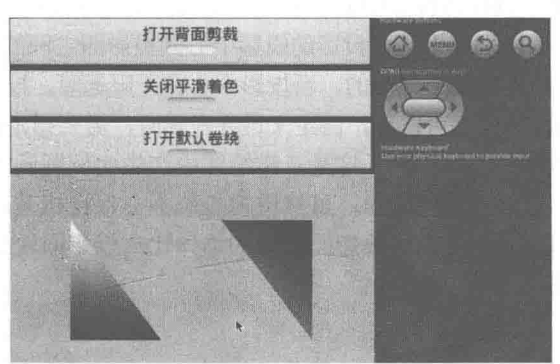
● 定义应用程序中具体实现场景物体的绘制方法，包括启用数组、旋转实现场景中的物体、为画笔指定顶点坐标数据、为画笔指定顶点颜色数据，并使用设置的画笔样式实现绘图功能。

文件 dingCH.java 的主要实现代码如下所示。

```
public class dingCH {
    private IntBuffer myVertexBuffer;
    private IntBuffer myColorBuffer;
    int vCount=0;
    float yAngle=0;
    float zAngle=0;
    public dingCH(){
        vCount=6;
        final int UNIT_SIZE=10000;
        int []vertices=new int[]{
            -8*UNIT_SIZE,10*UNIT_SIZE,0,
            -2*UNIT_SIZE,2*UNIT_SIZE,0,
            -8*UNIT_SIZE,2*UNIT_SIZE,0,
            8*UNIT_SIZE,2*UNIT_SIZE,0,
            8*UNIT_SIZE,10*UNIT_SIZE,0,
            2*UNIT_SIZE,10*UNIT_SIZE,0
        };
        //创建顶点坐标数据缓存
        ByteBuffer vbb=ByteBuffer.allocateDirect(vertices.length*4);
        vbb.order(ByteOrder.nativeOrder());
        myVertexBuffer=vbb.asIntBuffer();
        myVertexBuffer.put(vertices);
        myVertexBuffer.position(0);
        final int one=65535;
        int []colors=new int[]{
                one,one,one,0,
                0,0,one,0,
                0,0,one,0,
                one,one,one,0,
                one,0,0,0,
                one,0,0,0
        };
        ByteBuffer cbb=ByteBuffer.allocateDirect(colors.length*4);
        cbb.order(ByteOrder.nativeOrder());
        myColorBuffer=cbb.asIntBuffer();
        myColorBuffer.put(colors);
        myColorBuffer.position(0);
    }
    public void drawSelf(GL10 gl){
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        gl.glRotatef(yAngle,0,1,0);          //根据 yAngle 的角度值，绕 y 轴旋转 yAngle
        gl.glRotatef(zAngle,0,0,1);
        gl.glVertexPointer(
                3,                           //每个顶点的坐标数量为 3
                GL10.GL_FIXED,               //顶点坐标值的类型为 GL_FIXED,整型
                0,                           //连续顶点坐标数据之间的间隔
                myVertexBuffer               //顶点坐标数量
        );
        gl.glColorPointer(                   //为画笔指定顶点颜色数据
            4,
            GL10.GL_FIXED,
            0,
            myColorBuffer
        );
        gl.glDrawArrays(                     //绘制图形
            GL10.GL_TRIANGLES,
            0,                               //开始点编号
            vCount
        );
```

```
    }
}
```

本实例的执行效果如图 17-3 所示，和前面实例 17-2 的完全一样。



▲图 17-3 使用顶点法绘制三角形的执行效果

> 注意　在本实例中没有初始化三角形的索引数据缓冲，并且在此调用的是方法 glDrawElements()，也就是用顶点法来绘制图形。

17.2 实现投影效果

除了 17.1 中介绍的基本应用外，在 Android 手机屏幕中还可以使用 OpenGL ES 实现投影效果。在本节的内容中，将详细讲解实现投影效果的基本方法，为读者进入本书后面知识的学习打下基础。

17.2.1 正交投影

在 OpenGL ES 中只支持两种投影方式，分别是正交投影和透视投影。正交投影是平行投影的一种，特点是观察者的视线是平行的，不会产生真实世界中远大近小的透视效果。在此做一个假设：I 与 Z 是一个分别为具有二阶矩的 n 维和 m 维随机向量。如果存在一个与 I 同维的随机向量"&Icirc;"，如果满足下列 3 个条件则将"&Icirc;"称为是 I 在 Z 上的正交投影。

（1）线性表示，&Icirc; = $A + BZ$;

（2）无偏性，E（&Icirc;）= E（I）;

（3）I-&Icirc; 与 Z 正交，即 E[(I - &Icirc;)ZT]=0;

其中，ZT 是 Z 的转置。

17.2.2 透视投影

透视投影属于非平行投影，特点是观察者的视线在远处是相交的，当视线相交时表示灭点。因为通过透视投影可以产生现实世界中近大远小的效果，所以使用透视投影可以得到一个更加真实的 3D 感受。正因为如此，在现实游戏应用中一般采用透视投影方式。

透视投影是用中心投影法将形体投射到投影面上，从而获得的一种较为接近视觉效果的单面投影图。透视投影具有消失感、距离感、相同大小的形体呈现出有规律的变化等一系列的透视特性，能逼真地反映形体的空间形象。透视投影也称为透视图，简称透视。

除了在游戏领域比较受欢迎之外，透视图在建筑设计过程中通常用来表达设计对象的外貌，以帮助完成设计构思、研究、比较建筑物的空间造型和立面处理等工作，是建筑设计领域中最重要的辅助图样之一。

17.2.3 正交投影和透视投影的区别

在正交投影中，图形沿平行线变换到投影面上。对透视投影来说，图形沿收敛于某一点的直线变换到投影面上，这个点被称为投影中心，相当于观察点，也被称为视点。

正交投影和透视投影的区别在于透视投影的投影中心到投影面之间的距离是有限的，而正交投影的投影中心到投影面之间的距离是无限的。当投影中心在无限远时，投影线互相平行，所以定义正交投影时，给出投影线的方向就可以了，而定义透视投影时，需要指定投影中心的具体位置。

正交投影保持物体的有关比例不变，这是三维绘图中产生比例图画的方法，物体的各个面的精确视图可以由平行投影得到。另一方面，虽然透视投影不会保持相关比例，但是能够生成真实感的视图。对同样大小的物体来说，离投影面较远的物体比离投影面较近物体的投影图象要小，会产生近大远小的梦幻效果。

17.2.4 实现投影效果

在本节将通过一个具体实例的实现流程，详细讲解在 Android 屏幕中实现投影效果的方法。

题目	目的	源码路径
实例 17-4	在 Android 屏幕中实现投影效果	\daima\17\touCH

本实例的实现流程如下所示。

（1）编写文件 MyActivity.java，具体实现流程如下所示。

- 为布局文件中的按钮定义了监听器类，实现在两种投影之间切换，分别实现显示相应的效果。
- 重写 onPause 方法以继承父类的方法，并同时将 MySurfaceView 视图挂起或恢复。

文件 MyActivity.java 的主要实现代码如下所示。

```
public class MyActivity extends Activity {
    private MySurfaceView mSurfaceView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mSurfaceView = new MySurfaceView(this);
        mSurfaceView.requestFocus();
        mSurfaceView.setFocusableInTouchMode(true);    //设置为可触控
        LinearLayout ll=(LinearLayout)findViewById(R.id.main_liner);
        ll.addView(mSurfaceView);
        //控制是否打开背面剪裁的 ToggleButton
        ToggleButton tb=(ToggleButton)this.findViewById(R.id.ToggleButton01);
        tb.setOnCheckedChangeListener(new MyListener());
    }
    class MyListener implements OnCheckedChangeListener{
        @Override
        public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
                //在正交投影与透视投影之间切换
                mSurfaceView.isPerspective=!mSurfaceView.isPerspective;
                mSurfaceView.requestRender();        //重新绘制
        }
    }
```

（2）编写文件 MySurfaceView.java，具体实现流程如下所示。

- 定义 MySurfaceView 的构造器，以在创建 MySurfaceView 对象时设置渲染器和渲染模式。
- 定义触摸回调方法以实现屏幕触控功能，通过在屏幕上滑动实现旋转场景中物体的功能。
- 定义渲染器内部类，功能是实现对图像的渲染。

- 设置当屏幕横竖发生变化时的处理措施及创建 MySurfaceView 时的初始化功能。

文件 MySurfaceView.java 的主要实现代码如下所示。

```
class MySurfaceView extends GLSurfaceView {
    private final float TOUCH_SCALE_FACTOR = 180.0f/320;
    private SceneRenderer mRenderer;
    public boolean isPerspective=false;
    private float mPreviousY;                          //上次的触控位置 Y 坐标
    public float xAngle=0;                             //整体绕 x 轴旋转的角度
    public MySurfaceView(Context context) {
        super(context);
        mRenderer = new SceneRenderer();       setRenderer(mRenderer);
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
    }
    //触摸事件回调方法
    @Override
    public boolean onTouchEvent(MotionEvent e) {
        float y = e.getY();
        switch (e.getAction()) {
        case MotionEvent.ACTION_MOVE:                  //判断是否是滑动
            float dy = y - mPreviousY;                 //计算触控笔 Y 位移
            xAngle+= dy * TOUCH_SCALE_FACTOR;          //设置沿 x 轴旋转角度
            requestRender();
        }
        mPreviousY = y;
        return true;
    }
    private class SceneRenderer implements GLSurfaceView.Renderer {
        touCH[] ha=new touCH[]{                        //六边形数组
            new touCH(0),
            new touCH(-2),
            new touCH(-4),
            new touCH(-6),
            new touCH(-8),
            new touCH(-10),
            new touCH(-12),
        };
        public SceneRenderer(){}                       //渲染器构造类
        @Override
        public void onDrawFrame(GL10 gl) {
            gl.glMatrixMode(GL10.GL_PROJECTION);
            gl.glLoadIdentity();
            float ratio = (float) 320/480;
            if(isPerspective){
                gl.glFrustumf(-ratio, ratio, -1, 1, 1f, 10);
            }
            else{
                gl.glOrthof(-ratio, ratio, -1, 1, 1, 10);
            }
            gl.glEnable(GL10.GL_CULL_FACE);
            gl.glShadeModel(GL10.GL_SMOOTH);
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();

            gl.glTranslatef(0, 0f, -1.4f);
            gl.glRotatef(xAngle, 1, 0, 0);

            for(touCH th:ha){
                th.drawSelf(gl);
            }
        }
        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
            gl.glViewport(0, 0, width, height);
        }
        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
```

```
            gl.glDisable(GL10.GL_DITHER);                           //关闭抗抖动
            gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
            gl.glClearColor(0,255,255,0);                           //设置屏幕背景色青色
            gl.glEnable(GL10.GL_DEPTH_TEST);                        //启用深度测试
        }
    }
}
```

（3）编写文件 touCH.java，具体实现流程如下所示。

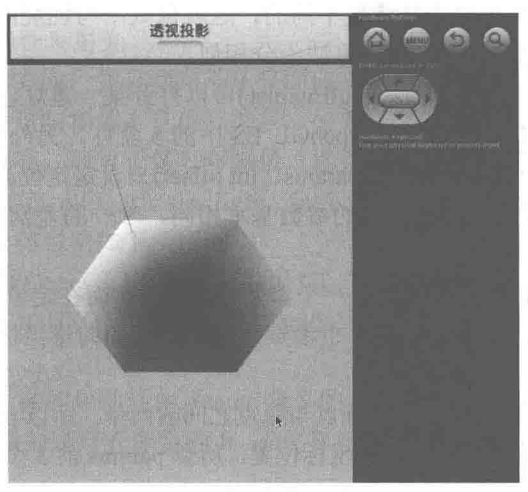
- 先声明顶点缓存、顶点颜色缓存、顶点索引缓存、顶点数、索引数等相关变量。
- 定义类 dingCH 的构造器来初始化相关数据，分别初始化六边形的顶点数据缓冲、颜色数据缓冲和索引数据缓冲。
- 定义应用程序中具体实现场景物体的绘制方法。

文件 touCH.java 的主要实现代码如下所示。

```
public class touCH {
private IntBuffer   mVertexBuffer;
private IntBuffer   mColorBuffer;
private ByteBuffer  mIndexBuffer;
int vCount=0;
int iCount=0;
public touCH(int zOffset){
    //顶点坐标数据的初始化
    vCount=7;
    final int UNIT_SIZE=10000;
    int vertices[]=new int[]{
        0*UNIT_SIZE,0*UNIT_SIZE,zOffset*UNIT_SIZE,
        2*UNIT_SIZE,3*UNIT_SIZE,zOffset*UNIT_SIZE,
        4*UNIT_SIZE,0*UNIT_SIZE,zOffset*UNIT_SIZE,
        2*UNIT_SIZE,-3*UNIT_SIZE,zOffset*UNIT_SIZE,
        -2*UNIT_SIZE,-3*UNIT_SIZE,zOffset*UNIT_SIZE,
        -4*UNIT_SIZE,0*UNIT_SIZE,zOffset*UNIT_SIZE,
        -2*UNIT_SIZE,3*UNIT_SIZE,zOffset*UNIT_SIZE
    };
    //创建顶点坐标数据缓冲
    ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
    vbb.order(ByteOrder.nativeOrder());
    mVertexBuffer = vbb.asIntBuffer();
    mVertexBuffer.put(vertices);
    mVertexBuffer.position(0);
    //顶点着色数据的初始化
    final int one = 65535;
    int colors[]=new int[]
    {
            0,0,one,0,
            0,one,0,0,
            0,one,one,0,

            one,0,0,0,
            one,0,one,0,
            one,one,0,0,
            one,one,one,0
    };
    //创建顶点着色数据缓冲
    ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length*4);
    cbb.order(ByteOrder.nativeOrder());
    mColorBuffer = cbb.asIntBuffer();
    mColorBuffer.put(colors);
    mColorBuffer.position(0);
    //三角形构造索引数据初始化
    iCount=18;
    byte indices[]=new byte[]{
        0,2,1,
        0,3,2,
        0,4,3,
        0,5,4,
```

```
            0,6,5,
            0,1,6
        };
        //创建三角形构造索引数据缓冲
        mIndexBuffer = ByteBuffer.allocateDirect(indices.length);
        mIndexBuffer.put(indices);
        mIndexBuffer.position(0);
    }
    public void drawSelf(GL10 gl){
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        gl.glVertexPointer(
                3,                                    //每个顶点的坐标数量为 3
                GL10.GL_FIXED,
                0,                                    //连续顶点坐标数据之间的间隔
                mVertexBuffer                         //顶点坐标数据
        );
        gl.glColorPointer(                            //为画笔指定顶点着色数据
                4,
                GL10.GL_FIXED,
                0,
                mColorBuffer
        );
        gl.glDrawElements(
                GL10.GL_TRIANGLES,
                iCount,
                GL10.GL_UNSIGNED_BYTE,                //索引值的尺寸
                mIndexBuffer                          //索引值数据
        );
    }
}
```

到此为止，整个实例的主要代码介绍完毕，执行后会显示一个青色背景屏幕，并在屏幕中分别显示正交投影和透视投影两种效果，如图 17-4 所示。



▲图 17-4　实现投影的执行效果

17.3 实现光照效果

除了本章前面介绍的基本应用外，在 Android 手机屏幕中通过 OpenGL ES 技术还可以实现光照效果。在本节的内容中，将详细讲解实现光照效果的基本方法，为读者进入本书后面知识的学习打下基础。

17.3.1 光源的类型

宇宙中的物体千姿百态，有的是发光的，有的是不发光的。我们把发光的物体叫作光源，例如太阳、电灯、燃烧着的蜡烛等都是光源。光也有能量。在 OpenGL ES 场景中至少包含 8 个光源，这些光源可以是不同的颜色。除 0 号灯之外的其他光源的颜色是黑色。

现实中的光源类型有多种，在日常生活中最常见的光源类型是定向光和定位光，具体说明如下所示。

（1）定向光。

我们日常所见的的光源有很多，比如太阳、灯泡、燃烧着的蜡烛等。像太阳这类被认为是从无穷远处发射的几乎平行的光被称为定向光。定向光对应的是光源在无穷远处的光，定向光在空间中的所有的位置方向都是相同的。

在 OpenGL ES 中通过方法 glLightfv (int light，int pname，float[] params，int offset)来设定定向光，上述方法中各个参数的具体说明如下所示。

- Light：该参数设定为 OpenGL ES 中的灯，用 GL_LIGHT0 到 GL_LIGHT7 分别表示 8 盏灯。如果该处设置的为 GL_LIGHT0，则表示方法 glLightfv 中其余的设置都是针对 GL_LIGHT0 的，即 0 号灯进行设置的。
- Pname：被设置的光源的属性是由 pname 定义的，它指定了一个命名参数，在设置定向光时应该设置成 GL POSITION。
- Params：此参数是一个 float 数组，该数组由 4 部分组成，前 3 个值组成表示定向光方向的向量，光的方向为从向量点处向原点处照射，如{0，1，0，0}表示沿 *Y* 轴负方向的光，最后的 0 表示此光源发出的是定向光。

（2）定位光。

在自然世界中定向光与定位光是截然不同的，这就像太阳与燃烧的蜡烛之间的区别。但是，在 OpenGL ES 中实现定向光与定位光的方法十分相似。

在 OpenGL ES 系统中，使用方法 gl.glEnable()可以打开某一盏灯，其参数 GL_LIGHTO、GL_LIGHT1……或 GL_LIGHT7 分别代表 OpenGL ES 中的 8 盏灯。另外，在 OpenGL ES 中通过方法 glLightfv (int light，int pname，float[] params，int offset)来设定定位光，其参数和前面介绍的定向光中的 glLightfv 方法类似，而且里面的参数基本相同，唯一的差别是 params 参数略有不同。具体差别如下所示。

- 在定向光中，参数 params 的最后一个参数设定为 0，而在定位光中，该参数设定为 1。
- 在定向光中，参数 params 的前 3 个参数为设定光源的向量坐标，而在定位光中，这 3 个参数是光源的位置。
- 在定向光中光的方向为给定的坐标点与原点之间的向量，所以 params 中的坐标不能设置为[0，0，0]；而在定位光中给出的是光源的坐标位置，所以 params 前 3 个参数可以设置为[0，0，0]。

在方法 glLightfv()中，设置其余参数的方法与前面介绍的方法 glLightfv 相同，在此不再赘述。

17.3.2 光源的颜色

颜色是光源的一种重要的属性，在 OpenGL ES 中允许把与颜色相关的 3 个不同参数 GL_AMBIENT、GL_DIFFUSE 和 GL_SPECULAR 与任何特定的光源相关联。

（1）GL_AMBIENT 环境光。

Ambient 表示环境光，表示一个特定的光源在场景中所添加的环境光的 RGBA 强度。在默认情况下是不存在环境光的，因为 GL_AMBIENT 的默认值是（0.0，0.0，0.0，1.0）。

在 OpenGL ES 中通过方法 glLightfv (int light，int pname，float[] params，int offset)来设定光源的环境光，各个参数的具体说明如下所示。

- light：该参数设定为 OpenGL ES 中的灯，用 GL_LIGHTO 到 GL_LIGHT7 分别来表示 8 盏灯。如果设置为 GL_LIGHTO 则表示 glLightfv 方法中其余的设置都是针对 GL_LIGHT0，即 0 号灯进行设置的。
- pname：被设置的光源的属性是由 pname 定义的，对于环境光设置为 GL_AMBIENT。
- params：此参数给出的是灯光颜色的 R、G、B、A 4 个色彩通道的值，一般环境光设置的值均较小。
- offset：偏移量，设置为 0，表示第 1 个色彩通道的值在数组中的偏移量。

（2）GL_DIFFUSE 散射光。

因为散射光是来自于某个方向的，所以如果散射光从正面照射物体表面，它看起来就显得更亮一些。反之如果它斜着从物体表面掠过，则看起来就显得暗一些。但是当散射光撞击物体表面时，它就会向四面八方均匀地发散。不管从哪个方向看，散射光看上去总是一样亮。来自某个特定位置或方向的任何光很可能具有散射成分。

在 OpenGL ES 平台中，可以通过方法 gILightfv (int light，int pname，float[] params，int offset)来设定光源的散射光，各个参数的具体说明如下所示。

- light：该参数设定为 OpenGL ES 中的灯，用 GL_LIGHTO 到 GL_LIGHT7 来表示 8 盏灯。如果设置为 GL_LIGHTO，则表示在 glLightfv()方法中其余的设置都是针对 GL_LIGHT0，即 0 号灯进行设置的。
- pname：被设置的光源的属性是由 pname 定义的，对于环境光设置为 GL_DIFFUSE。
- params：此参数给出的是灯光颜色的 R、G、B、A 四个色彩通道的值。
- offset：偏移量，设置为 0，表示第 1 个色彩通道的值在数组中的偏移量。

（3）GL_SPECULAR 镜面反射光。

镜面光来自一个特定的方向，并且倾向于从表面向某个特定的方向反射。镜面光肉眼看起来是物体上最亮的地方。当然这与物体本身也有关系，如果是类似镜子的、光泽的金属等，则光线为全反射，整个物体都很明亮，而对于像石膏雕像、地毯等则几乎不存在镜面成分。

在 OpenGL ES 中使用方法 glLightfv (int light，int pname，float[] params，int offset)来设定光源的镜面光，各个参数的具体说明如下所示。

- light：该参数设定为 OpenGL ES 中的灯，用 GL_LIGHT0 到 GL_LIGHT7 分别来表示 8 盏灯。如果该处设置的为 GL_LIGHTO，即表示在 gILightfv 方法中其余的设置都是针对 GL_LIGHT0，即 0 号灯进行设置的。
- pname：被设置的光源的属性是由 pname 定义的，对于环境光设置为 GL_SPECULARa。
- params：此参数给出的是灯光颜色的 R、G、B、A 4 个色彩通道的值。在镜面反射光中，该参数一般较大。
- offset：偏移量，设置为 0，表示第 1 个色彩通道的值在数组中的偏移量。

17.3.3 开启/关闭光照

在接下来的内容中，将通过一个具体实例来讲解在 Android 屏幕中分别开启、关闭光照效果的方法。

题目	目的	源码路径
实例 17-5	在 Android 屏幕中分别开启、关闭光照效果	\daima\17\kaiguanCH

本实例的实现流程如下所示。

（1）编写文件 MyActivity.java，具体实现流程如下所示。

- 实例化 MySurfaceView 对象，同时设置 Acitivity 的内容。
- 设置 MySurfaceView 为可触控。
- 当 Acitvity 调用了方法 onPause()和方法 onResume()时，GLSurfaceView 需要调用相应的操作，即分别调用方法 onPause()及方法 onResume()。

文件 MyActivity.java 的主要实现代码如下所示。

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        msv=new MySurfaceView(this);                    //实例化 MySurfaceView
        setContentView(R.layout.main);                  //设置 Acitivity 的内容
        msv.requestFocus();                             //获取焦点
        msv.setFocusableInTouchMode(true);              //设置为可触控
        LinearLayout lla=(LinearLayout)findViewById(R.id.lla);
        lla.addView(msv);                               //将 SurfaceView 加入 LinearLayout 中
     tb=(ToggleButton)findViewById(R.id.ToggleButton01);//添加监听器
        tb.setOnCheckedChangeListener(new OnCheckedChangeListener(){
            @Override
            public void onCheckedChanged(CompoundButton buttonView,
                    boolean isChecked) {
                msv.openLightFlag=!msv.openLightFlag;
        }});}
    @Override
    protected void onPause() {                         //在另一个 Acitvity 遮挡当前 Activity 时调用
        super.onPause();
        msv.onPause();
    }
    @Override
    protected void onResume() {                        //当 Acitvity 获得用户焦点时调用
        super.onResume();
        msv.onResume();
    }
}
```

（2）编写文件 MySurfaceView.java，具体实现流程如下所示。

- 使用方法 gl.glEnable(GLlO.GL_LIGHTING)打开灯光效果。
- 通过 gl.gILightfv()设定光照相关参数，分别实现关闭抗抖动、设置背景颜色、设置着色模式等操作。
- 初始化 0 号灯，分别设置 0 号灯的环境光、散射光、反射光。
- 设置物体的材质。

文件 MySurfaceView.java 的主要实现代码如下所示。

```
    private class SceneRenderer implements GLSurfaceView.Renderer
    {
        kaiguanCH ball=new kaiguanCH(4);
        public SceneRenderer(){
        }
        public void onDrawFrame(GL10 gl){
            gl.glShadeModel(GL10.GL_SMOOTH);
             if(openLightFlag){                                 //打开灯
                 gl.glEnable(GL10.GL_LIGHTING);                 //允许光照
                 initLight0(gl);                                //初始化绿色灯
                 initMaterialWhite(gl);                         //初始化材质为白色
               //设定 Light0 光源的位置
                 float[] positionParamsGreen={2,1,0,1};  //最后的 1 表示是定位光
                 gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_POSITION, positionParamsGreen,0);
            }else{//关灯
                gl.glDisable(GL10.GL_LIGHTING);
            }
            //清除颜色缓存
```

```
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        //设置为模式矩阵
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        //设置当前矩阵为单位矩阵
        gl.glLoadIdentity();
        gl.glTranslatef(0, 0f, -1.8f);
        ball.drawSelf(gl);
        gl.glLoadIdentity();
    }
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        //设置视窗大小及位置
        gl.glViewport(0, 0, width, height);
        //设置当前矩阵为投影矩阵
        gl.glMatrixMode(GL10.GL_PROJECTION);
        //设置当前矩阵为单位矩阵
        gl.glLoadIdentity();
        //计算透视投影的比例
        float ratio = (float) width / height;
        //计算产生透视投影矩阵
        gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);
    }
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        //关闭抗抖动
        gl.glDisable(GL10.GL_DITHER);
           //设置特定 Hint 项目的模式，这里设置为使用快速模式
        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
        //设置屏幕背景色黑色 RGBA
        gl.glClearColor(0,0,0,0);
        //设置着色模型为平滑着色
        gl.glShadeModel(GL10.GL_SMOOTH);//GL10.GL_SMOOTH  GL10.GL_FLAT
        //启用深度测试
        gl.glEnable(GL10.GL_DEPTH_TEST);
}}
private void initLight0(GL10 gl){
    gl.glEnable(GL10.GL_LIGHT0);//打开 0 号灯
    //环境光设置
    float[] ambientParams={0.1f,0.1f,0.1f,1.0f};
    gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_AMBIENT, ambientParams,0);
    //散射光设置
    float[] diffuseParams={0.5f,0.5f,0.5f,1.0f};
    gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_DIFFUSE, diffuseParams,0);
    //反射光设置
    float[] specularParams={1.0f,1.0f,1.0f,1.0f};//光参数 RGBA
    gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_SPECULAR, specularParams,0);
}
private void initMaterialWhite(GL10 gl){//设置材质为白色时的光照颜色
    //环境光为白色材质
    float ambientMaterial[] = {0.4f, 0.4f, 0.4f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT, ambientMaterial,0);
    //散射光为白色材质
    float diffuseMaterial[] = {0.8f, 0.8f, 0.8f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE, diffuseMaterial,0);
    //高光材质为白色
    float specularMaterial[] = {1.0f, 1.0f, 1.0f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SPECULAR, specularMaterial,0);
    //数越大，高亮区域越小越暗
    float shininessMaterial[] = {1.5f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,GL10.GL_SHININESS,shininessMaterial,0);
}
}
```

（3）编写文件 kaiguanCH.java，具体实现流程如下所示。

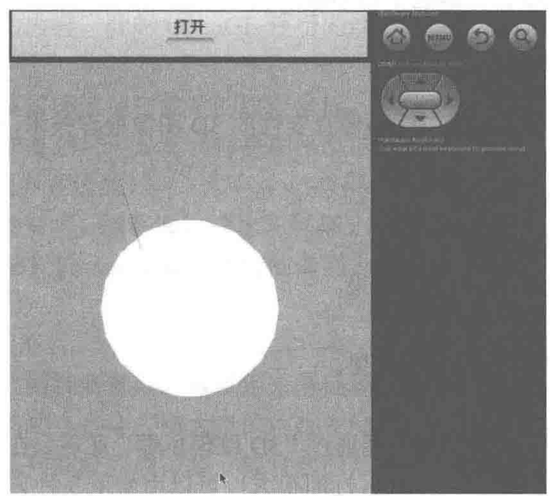
- 创建顶点坐标数据缓冲，并使用索引法为三角形构造初始化索引数据。
- 为画笔指定顶点坐标数据、顶点法向量数据，并同时绘制图形。
- 通过方法 glNormalPointer()为画笔指定顶点法向量数据，并分别计算球体的 x、y、z 坐标。
- 用中间行的两个相邻点与下一行的对应点构成三角形。
- 用中间行的两个相邻点与上一行的对应点构成三角形。

文件 kaiguanCH.java 的主要实现代码如下所示。

```
public class kaiguanCH {
   private IntBuffer  mVertexBuffer;                          //缓冲顶点坐标数据
   private IntBuffer  mNormalBuffer;                          //缓冲顶点法向量数据
   private ByteBuffer mIndexBuffer;                           //缓冲顶点构建索引数据
   public float mAngleX;                                      //x 轴旋转角度
   public float mAngleY;                                      //y 轴旋转角度
   public float mAngleZ;                                      //z 轴旋转角度
   int vCount=0;
   int iCount=0;
   public kaiguanCH(int scale){
       //初始化顶点坐标数据
       final int UNIT_SIZE=10000;
       //存放顶点坐标的 ArrayList
       ArrayList<Integer> alVertix=new ArrayList<Integer>();
       final int angleSpan=18;                                //将球进行单位切分的角度
       for(int vAngle=-90;vAngle<=90;vAngle=vAngle+angleSpan){
       for(int hAngle=0;hAngle<360;hAngle=hAngle+angleSpan)
       {    //纵向横向各到一个角度后开始计算对应的此点在球面上的坐标
            double xozLength=scale*UNIT_SIZE*Math.cos(Math.toRadians(vAngle));
            int x=(int)(xozLength*Math.cos(Math.toRadians(hAngle)));
            int z=(int)(xozLength*Math.sin(Math.toRadians(hAngle)));
            int y=(int)(scale*UNIT_SIZE*Math.sin(Math.toRadians(vAngle)));
            //将计算出来的 X、Y、Z 坐标存放到顶点坐标的 ArrayList
            alVertix.add(x);alVertix.add(y);alVertix.add(z);
       }
       }
       //因为一个顶点有 3 个坐标，所以顶点的数量为坐标值数量的 1/3
       vCount=alVertix.size()/3;
       //将 alVertix 中的坐标值转存到一个 int 数组中
       int vertices[]=new int[vCount*3];
       for(int i=0;i<alVertix.size();i++){
          vertices[i]=alVertix.get(i);
       }
       //创建顶点坐标数据缓冲
       ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
       vbb.order(ByteOrder.nativeOrder());                    //设置字节顺序
       mVertexBuffer = vbb.asIntBuffer();                     //转换为 int 型缓冲
       mVertexBuffer.put(vertices);                           //向缓冲区中放入顶点坐标数据
       mVertexBuffer.position(0);                             //设置缓冲区起始位置
       //创建顶点法向量数据缓冲
       //vertices.length*4 是因为一个 float4 字节
       ByteBuffer nbb = ByteBuffer.allocateDirect(vertices.length*4);
       nbb.order(ByteOrder.nativeOrder());                    //设置字节顺序
       mNormalBuffer = vbb.asIntBuffer();                     //转换为 int 型缓冲
       mNormalBuffer.put(vertices);                           //向缓冲区中放入顶点坐标数据
       mNormalBuffer.position(0);                             //设置缓冲区起始位置
       //顶点坐标数据的初始化
       //三角形构造索引数据初始化
       ArrayList<Integer> alIndex=new ArrayList<Integer>();
       int row=(180/angleSpan)+1;                             //切分球面的行数
       int col=360/angleSpan;                                 //切分球面的列数
       for(int i=0;i<row;i++){
       if(i>0&&i<row-1){
       //中间行
              for(int j=-1;j<col;j++){
              //中间行的两个相邻点与下一行的对应点构成三角形
                  int k=i*col+j;
                  alIndex.add(k+col);
                  alIndex.add(k+1);
                  alIndex.add(k);
              }
          for(int j=0;j<col+1;j++){
              //中间行的两个相邻点与上一行的对应点构成三角形
                  int k=i*col+j;
                  alIndex.add(k-col);
                  alIndex.add(k-1);
                  alIndex.add(k);
       }}}
       iCount=alIndex.size();
```

```
        byte indices[]=new byte[alIndex.size()];
        for(int i=0;i<alIndex.size();i++){
            indices[i]=alIndex.get(i).byteValue();
        }
        //创建三角形构造索引数据缓冲
        mIndexBuffer = ByteBuffer.allocateDirect(indices.length);
        mIndexBuffer.put(indices);                      //向缓冲区中放入三角形构造索引数据
        mIndexBuffer.position(0);                       //设置缓冲区起始位置
    }
    public void drawSelf(GL10 gl){
        gl.glRotatef(mAngleZ, 0, 0, 1);                 //Z 轴旋转
        gl.glRotatef(mAngleX, 1, 0, 0);                 //X 轴旋转
        gl.glRotatef(mAngleY, 0, 1, 0);                 //Y 轴旋转
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);
        //为画笔指定顶点坐标数据
        gl.glVertexPointer(
                3,                                      //每个顶点的坐标数量为 3
                GL10.GL_FIXED,                          //顶点坐标值的类型为 GL_FIXED
                0,                                      //连续顶点坐标数据之间的间隔
                mVertexBuffer                           //顶点坐标数据
        );
        //为画笔指定顶点法向量数据
        gl.glNormalPointer(GL10.GL_FIXED, 0, mNormalBuffer);
        //绘制图形
        gl.glDrawElements(
                GL10.GL_TRIANGLES,                      //以三角形方式填充
                iCount,                                 //一共(icount/3)个三角形，iCount 个顶点
                GL10.GL_UNSIGNED_BYTE,                  //索引值的尺寸
                mIndexBuffer                            //索引值数据
        );
    }
}
```

到此为止，整个实例介绍完毕，执行后的效果如图 17-5 所示。



▲图 17-5　开启和关闭光照的执行效果

第 18 章 纹理映射

纹理映射（Texture Mapping）是指将纹理空间中的纹理像素映射到屏幕空间中的像素的过程。在三维图形应用过程中使用最广泛的便是纹理映射，尤其在描述具有真实感的物体时更是最佳选择。在本章的内容中，将详细讲解在 Android 平台上实现纹理映射效果的知识，为读者进入本书后面知识的学习打下基础。

18.1 纹理映射基础

通过纹理映射能够制作出极具真实感的图形，而不必花过多时间来考虑物体的表面细节。但是当纹理图像非常大时，纹理加载的过程会影响程序运行速度。如何能够妥善地管理纹理，减少不必要的开销，是在做系统优化时必须考虑的一个问题。幸运的是，在 OpenGL 中提供的纹理对象管理技术可以帮助我们解决上述问题。跟传统的显示列表一样，可以通过一个单独的数字来标识纹理对象。这样可以允许 OpenGL 硬件能够在内存中保存多个纹理，而不是每次使用的时候再加载它们，从而减少了运算量，提高了处理速度。

18.1.1 纹理贴图和纹理拉伸

纹理贴图是一项能大幅度提高 3D 图像真实性的 3D 图像处理技术，使用这项技术的好处如下所示。

- 减少纹理衔接错误。
- 实时生成剖析截面显示图。
- 有更真实的雾、烟、火和动画效果。
- 提高变换视角看物体的真实性。
- 模拟移动光源产生的自然光影效果。
- 构成枪弹真实轨迹等。

在目前的显卡条件下，上述功能只能通过“3D 纹理压缩”实现。在具体实现时，可以把一幅纹理图拉伸或缩小贴到目标面上。如果目标面很大，可以用如下 3 种方案解决。

（1）将纹理拉大，这样做的缺点是纹理显得非常不清楚，失去了原来清晰的效果，甚至可能变形。

（2）将目标面分割为多个与纹理大小相似的矩形，再将纹理重复贴到被分割的目标上。这样做的缺点是浪费了内存（需要额外存储大量的顶点信息），也浪费了开发人员宝贵的精力。

（3）使用合理的纹理拉伸方式，使得纹理能够根据目标平面的大小自动重复，这样既不会失去纹理图的效果，也节省了内存，提高了开发效率。

通过比较上述 3 种解决方案可知，第 3 种方案是最好的解决方法，并且很容易实现，只需要做如下两方面的工作即可。

- 将纹理的 GL TEXTURE_WRAP_S 与 GL_TEXTURE_WRAP_T 属性值设置为 GL_REPEAT，而不是 GL CLAMP_TO_EDGE。
- 设置纹理坐标时纹理坐标的取值范围不再是 0～1，而是 0～*n*，*n* 为希望纹理重复的次数。

18.1.2 Texture Filter 纹理过滤

贴图层的纹理筛选过滤技术中，线性过滤能提升游戏纹理清晰度，而各项异性过滤甚至能降低附在三角形上的由于高几何精度下物体观察角度造成图形折叠的“纹理失真”（和锯齿失真一样附着于物体模型上的三角形纹理在非 90 度垂直观看时同样会损失效果，这就是 Xbox360、PS3 游戏机画面不仅物体边缘“抖”，而且与屏幕角度非平行的物体纹理更抖的原因——没有三维空间纹理过滤技术）。这使视觉效果更贴近真实物体表面，例如好莱坞大片《生化危机 5》支持如下线性或各项异性过滤技术。

（1）Linear（线性过滤）。

线性过滤分为具有纹理放大、缩小筛选器的双线性插补过滤和在 mipmap（纹理映射）级别间使用的三线性 mipmap 插补筛选器。双线性筛选后的纹理使用所需像素周围 2×2 区域内的“纹理像素”（单个像素纹理元素）的加权平均值。三线性筛选中，光栅化程序使用两个最近的 mipmap 纹理像素对像素颜色执行线性插补。显然三线性纹理过滤品质在 3D 游戏中高于双线性。

MAG_FILTER 采用的就是线性纹理过滤，实现方法为 glTexParameterf，具体语法格式如下所示。

```
gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_ MAG_FILTER,GL10.GL_LINEAR)
```

如果选择了 GL_LINEAR，那么 OpenGL 就会对靠近像素中心的一块 2×2 纹理矩形单元取加权平均值，用于实现放大和缩小处理。当纹理坐标靠近纹理图像的边缘时，最邻近 2×2 纹理单元可能包含了纹理图像之外的内容。此时 OpenGL 使用的纹理单元取决于当前生效的环绕模式，以及纹理是否有边框。

线性过滤先要经过计算目标图像中像素的纹理坐标，然后再从纹理图中提取像素实现。虽然这种计算方法比最近点采样复杂，但是能获得更加平滑的效果。

（2）Anisotropic（各项异性过滤）。

通过使用各项异性过滤技术，能够通过筛选与屏幕 *x*、*y* 轴平面之间的角度差异所造成的纹理模糊失真。Anisotropic 是 MT Framework 中支持的最强大的纹理过滤技术，按图形效果分为 2 倍到 16 倍筛选级别。

（3）MipMap 多重细节层。

MipMap 比前面介绍的两种要复杂，可以用来降低场景渲染的时间消耗，同时也提高了场景的真实感。但是 MipMap 的缺点是要占用大量的内存空间，在内存受限的场合不适合。

一个 MipMap 就是一系列的纹理图，每一幅纹理图都与前一幅具有相同的图样，但是分辨率要比前一幅有所降低。MipMap 中的每一幅或者每一级图像的宽和高都比之前一级小二分之一。需要注意的是，MipMap 并不一定是正方形的，为了使用 MipMap，必须提供全系列的大小为 2 的整数次方的纹理图像，其范围从最大值到 1×1 纹理单元。

高分辨率的 MipMap 图像用于接近观察者的物体，当物体逐渐远离观察者时，使用低分辨率的图像。MipMap 可以提高场景的渲染质量，但是其内存消耗十分大。这是因为此方法能够模拟纹理的头饰效果并能够减少处理时的计算量。与第一幅纹理用于不同的分辨率相比，这种方法的速度更快。

18.2 实现三角形纹理贴图效果

在接下来的内容中，将通过一个具体实例的实现过程来讲解在 Android 屏幕中实现三角形纹理贴图效果的方法。

题目	目的	源码路径
实例 18-1	在 Android 屏幕中实现三角形纹理贴图效果	\daima\18\wenCH

本实例的具体实现流程如下所示。

（1）编写文件 Dad.java，具体实现流程如下所示。

- 在 Dad 构造器中创建和设置场景渲染器为主动渲染，并设置重写触屏事件回调方法以记录触控笔坐标，改变三角形在坐标系的位置，使三角形能够在场景中转动。
- 为声明场景渲染类，在该类中首先设置场景属性，移动坐标系可以绘制三角形。
- 定义生成纹理 ID 的方法 initTexture，该方法通过接收图片 Id 和 gl 引用，将图片转换成 Bitmap。

文件 Dad.java 的主要实现代码如下所示。

```
public class Dad extends GLSurfaceView{
    private SceneRenderer mRenderer;                                //场景渲染器
    private float mPreviousY;
    private float mPreviousX;
    private float TOUCH_SCALE_FACTOR=180.0f/320;
    public int textureId;
    public Dad(Context context) {
        super(context);
        mRenderer = new SceneRenderer();
        setRenderer(mRenderer);
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);    //设置渲染模式为主动渲染
    }
    @Override
    public boolean onTouchEvent(MotionEvent e)
    {
        float y = e.getY();
        float x = e.getX();
        switch (e.getAction()) {
        case MotionEvent.ACTION_MOVE:
            float dy = y - mPreviousY;                              //计算触控笔 y 位移
            float dx = x - mPreviousX;                              //计算触控笔 y 位移
            mRenderer.texTri.jiaoY += dy * TOUCH_SCALE_FACTOR;    //设置沿 x 轴旋转角度
            mRenderer.texTri.mAngleZ += dx * TOUCH_SCALE_FACTOR; //设置沿 z 轴旋转角度
            requestRender();
        }
        mPreviousY = y;                                             //记录触控笔位置
        mPreviousX = x;                                             //记录触控笔位置
        return true;
    }
    private class SceneRenderer implements GLSurfaceView.Renderer
    {
        Texture texTri;
        int textureId;
        @Override
        public void onDrawFrame(GL10 gl) {
            // TODO Auto-generated method stub
            //清除颜色缓存
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
            //设置当前矩阵为模式矩阵
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            //设置当前矩阵为单位矩阵
            gl.glLoadIdentity();
```

```
            gl.glTranslatef(0, 0f, -2.5f);
            texTri.drawSelf(gl);
        }
        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
            // TODO Auto-generated method stub
             //设置视窗大小及位置
            gl.glViewport(0, 0, width, height);
            //设置当前矩阵为投影矩阵
            gl.glMatrixMode(GL10.GL_PROJECTION);
            //设置当前矩阵为单位矩阵
            gl.glLoadIdentity();
            //计算透视投影的比例
            float ratio = (float) width / height;
            //调用此方法计算产生透视投影矩阵
            gl.glFrustumf(-ratio, ratio, -1, 1, 1, 20);
        }
        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            // TODO Auto-generated method stub
            //关闭抗抖动
            gl.glDisable(GL10.GL_DITHER);
            //设置特定 Hint 项目的模式，这里为设置为使用快速模式
            gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
            //设置屏幕背景色黑色 RGBA
            gl.glClearColor(0,0,0,0);
            //打开背面剪裁
            //gl.glEnable(GL10.GL_CULL_FACE);
            //设置着色模型为平滑着色
            gl.glShadeModel(GL10.GL_SMOOTH);//GL10.GL_SMOOTH  GL10.GL_FLAT
            //启用深度测试
            gl.glEnable(GL10.GL_DEPTH_TEST);
            //初始化纹理
            textureId=initTexture(gl,R.drawable.su);
            texTri=new Texture(textureId);
        }
    }
    public int initTexture(GL10 gl,int textureId)//textureId
    {
        int[] textures = new int[1];
        gl.glGenTextures(1, textures, 0);
        int currTextureId=textures[0];
        gl.glBindTexture(GL10.GL_TEXTURE_2D, currTextureId);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_
NEAREST);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MAG_FILTER,GL10.GL_
LINEAR);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,GL10.GL_REPEAT);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,GL10.GL_REPEAT);

        InputStream is = this.getResources().openRawResource(textureId);
        Bitmap bitmapTmp;
        try
        {
        bitmapTmp = BitmapFactory.decodeStream(is);
        }
        finally
        {
            try
            {
                is.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmapTmp, 0);
        bitmapTmp.recycle();
```

```
        return currTextureId;
    }
}
```

（2）编写文件 yisuo.java，定义绘制三角形类 Texture，具体实现流程如下所示。

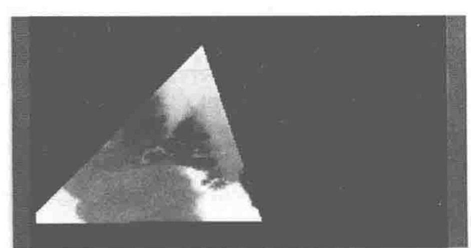
- 创建顶点数组，并将顶点数组放入顶点缓冲区内，为绘制三角形做好准备。
- 创建纹理坐标数组，并将纹理数组放入纹理坐标缓冲区内，为绘制三角形做好准备。
- 绘制三角形。

文件 yisuo.java 的主要实现代码如下所示。

```
    public Texture(int textureId)
    {
        this.textureId=textureId;
        //顶点坐标数据的初始化
        final int UNIT_SIZE=30000;
        vCount=3;                                       //顶点的数量
        int vertices[]=new int[]                        //顶点坐标数据数组
        {
            2*UNIT_SIZE,0,0,
            -2*UNIT_SIZE,0,0,
            0,4*UNIT_SIZE,0
        };
        //创建顶点坐标数据缓冲
        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
        vbb.order(ByteOrder.nativeOrder());             //设置字节顺序
        mVertexBuffer = vbb.asIntBuffer();
        mVertexBuffer.put(vertices);                    //向缓冲区中放入顶点坐标数据
        mVertexBuffer.position(0);                      //设置缓冲区起始位置

        float textureCoors[]=new float[]                //顶点纹理 s、t 坐标值数组
        {
            0,1,
            1,1,
            0.5f,0
        };

        //创建顶点纹理数据缓冲
        ByteBuffer cbb = ByteBuffer.allocateDirect(textureCoors.length*4);
        cbb.order(ByteOrder.nativeOrder());
        mTextureBuffer = cbb.asFloatBuffer();
        mTextureBuffer.put(textureCoors);
        mTextureBuffer.position(0);
    }

    public void drawSelf(GL10 gl)
    {
        gl.glRotatef(mAngleZ, 0, 0, 1);                 //沿 z 轴旋转
        gl.glRotatef(jiaoY, 0, 1, 0);                   //沿 y 轴旋转

        //允许使用顶点数组
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        //为画笔指定顶点坐标数据
        gl.glVertexPointer
        (
                3,
                GL10.GL_FIXED,
                0,
                mVertexBuffer
        );

        //开启纹理
        gl.glEnable(GL10.GL_TEXTURE_2D);
        //允许使用纹理数组
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        //为画笔指定纹理 u、v 坐标数据
        gl.glTexCoordPointer
```

```
        (
                2,
                GL10.GL_FLOAT,
                0,
                mTextureBuffer
        );
        //为画笔绑定指定名称 ID 纹理
        gl.glBindTexture(GL10.GL_TEXTURE_2D,textureId);
        //绘制图形
        gl.glDrawArrays
        (
                GL10.GL_TRIANGLES,
                0,
                vCount
        );
        gl.glDisable(GL10.GL_TEXTURE_2D);//关闭纹理
    }
}
```

到此为止，整个实例介绍完毕，执行后的效果如图 18-1 所示。



▲图 18-1　三角形纹理贴图的执行效果

18.3 实现地月模型效果

接下来的将通过一个具体实例的实现过程，来讲解在 Android 屏幕中使用纹理映射实现地月模型效果的方法。

题目	目的	源码路径
实例 18-2	使用纹理映射实现地月模型效果	\daima\18\moonCH

(1) 编写文件 Dad.java，具体实现流程如下所示。

在 Dad 构造器中创建和设置场景渲染器为主动渲染，并声明地球和月球的引用。

- 在绘制场景的方法中绘制地球、月球和浩瀚的星空。
- 通过线程控制月亮、地球和星星的转动。
- 创建纹理贴图。

文件 Dad.java 的主要实现代码如下所示。

```
private class SceneRenderer implements GLSurfaceView.Renderer
{
    Ball earth;
    Ball moon;
    Celestial celestialSmall;                          //小星星
    Celestial celestialBig;                            //大星，星星，空半球
            public void onDrawFrame(GL10 gl) {
    //清除颜色缓存
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
    //设置当前矩阵为模式矩阵
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    //设置当前矩阵为单位矩阵
```

```
        gl.glLoadIdentity();

        gl.glTranslatef(0, 0f, -3.6f);

        gl.glEnable(GL10.GL_LIGHTING);
        gl.glPushMatrix();
        gl.glMaterialf(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS, 3.5f);
        earth.drawSelf(gl);
        gl.glTranslatef(0, 0f, 1.5f);
        gl.glMaterialf(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS, 1.0f);
        moon.drawSelf(gl);
        gl.glPopMatrix();
        gl.glDisable(GL10.GL_LIGHTING);

        //绘制星空
        gl.glPushMatrix();
        gl.glTranslatef(0, -8.0f, 0.0f);
        celestialSmall.drawSelf(gl);
        celestialBig.drawSelf(gl);
        gl.glPopMatrix();

    }

    public void onSurfaceChanged(GL10 gl, int width, int height) {
        //设置视窗大小及位置
        gl.glViewport(0, 0, width, height);
        //设置当前矩阵为投影矩阵
        gl.glMatrixMode(GL10.GL_PROJECTION);
        //设置当前矩阵为单位矩阵
        gl.glLoadIdentity();
        //计算透视投影的比例
        float ratio = (float) width / height;
        //调用此方法计算产生透视投影矩阵
        gl.glFrustumf(-ratio*0.5f, ratio*0.5f, -0.5f, 0.5f, 1, 100);
    }

    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        //关闭抗抖动
        gl.glDisable(GL10.GL_DITHER);
          //设置特定 Hint 项目的模式，这里为设置为使用快速模式
        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
        //设置屏幕背景色黑色 RGBA
        gl.glClearColor(0,0,0,0);
        //设置着色模型为平滑着色
        gl.glShadeModel(GL10.GL_SMOOTH);
        //启用深度测试
        gl.glEnable(GL10.GL_DEPTH_TEST);
        //设置为打开背面剪裁
        gl.glEnable(GL10.GL_CULL_FACE);

        gl.glEnable(GL10.GL_LIGHTING);
        initSunLight(gl);                                   //太阳光源初始化
        initMaterial(gl);                                   //材质初始化

        //初始化纹理
        earthTextureId=initTexture(gl,R.drawable.di);
        moonTextureId=initTexture(gl,R.drawable.yue);

        earth=new Ball(6,earthTextureId);
        moon=new Ball(2,moonTextureId);
        //创建星空
        celestialSmall=new Celestial(0,0,1,0,750);
        celestialBig=new Celestial(0,0,2,0,200);
       //开启线程自动旋转地球与月球
        new Thread()
        {
            public void run()
            {
                while(true)
```

```
            {
           //设置球沿 y 轴转动
              mRenderer.earth.jiaoY+=2*TOUCH_SCALE_FACTOR;
              mRenderer.moon.jiaoY+=2*TOUCH_SCALE_FACTOR;
            requestRender();                        //重绘画面
            try
            {
               Thread.sleep(50);                    //休息 10ms 后继续重绘
            }
            catch(Exception e)
            {
               e.printStackTrace();
            }
            }
         }
      }.start();

      new Thread()//定时转动星空的线程
      {
         public void run()
         {
            while(true)
            {
              celestialSmall.yAngle+=0.5;
              if(celestialSmall.yAngle>=360)
              {
                  celestialSmall.yAngle=0;
              }
              celestialBig.yAngle+=0.5;
             if(celestialBig.yAngle>=360)
             {
                 celestialBig.yAngle=0;
             }
               try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                 }
            }
         }
      }.start();
   }
}

//初始化太阳光源
private void initSunLight(GL10 gl)
{
   gl.glEnable(GL10.GL_LIGHT0);                             //打开 0 号灯

   //环境光设置
   float[] ambientParams={0.05f,0.05f,0.025f,1.0f};
   gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_AMBIENT, ambientParams,0);

   //散射光设置
   float[] diffuseParams={1f,1f,0.5f,1.0f};
   gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_DIFFUSE, diffuseParams,0);
   //反射光设置
   float[] specularParams={1f,1f,0.5f,1.0f};
   gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_SPECULAR, specularParams,0);

   //设定光源的位置
   float[] positionParamsGreen={-14.14f,8.28f,6f,0};
   gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_POSITION, positionParamsGreen,0);
}


//初始化材质
private void initMaterial(GL10 gl)
{//材质为白色时什么颜色的光照在上面就将体现什么颜色
   //环境光为白色材质
```

```
        float ambientMaterial[] = {0.7f, 0.7f, 0.7f, 1.0f};
        gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT, ambientMaterial,0);
        //散射光为白色材质
        float diffuseMaterial[] = {1.0f, 1.0f, 1.0f, 1.0f};
        gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE, diffuseMaterial,0);
        //高光材质为白色
        float specularMaterial[] = {1f, 1f, 1f, 1.0f};
        gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SPECULAR, specularMaterial,0);
    }

    //初始化纹理
    public int initTexture(GL10 gl,int drawableId)
    {
        //生成纹理 ID
        int[] textures = new int[1];
        gl.glGenTextures(1, textures, 0);
        int textureId=textures[0];
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_
NEAREST);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MAG_FILTER,GL10.GL_LINEAR);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_WRAP_S,GL10.GL_CLAMP_TO_
EDGE);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_WRAP_T,GL10.GL_CLAMP_TO_
EDGE);

        InputStream is = this.getResources().openRawResource(drawableId);
        Bitmap bitmapTmp;
        try
        {
            bitmapTmp = BitmapFactory.decodeStream(is);
        }
        finally
        {
            try
            {
                is.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmapTmp, 0);
        bitmapTmp.recycle();
         return textureId;
    }
}
```

（2）编写文件 Ball.java，具体实现流程如下所示。

- 定义绘制类 Ball。
- 分别获得纹理图切分的行数和列数。
- 声明纹理的数组计数器和纹理数组的长度，以及存放纹理坐标的列表，为精确计算纹理坐标提前做好准备。
- 将获得的纹理坐标放入纹理坐标缓冲区内，并设置缓冲区的起始位置。
- 在开启纹理后，允许使用纹理 *s*、*t* 坐标缓冲，并为画笔指定纹理 *s*、*t* 坐标，绑定纹理后开始绘制图形。
- 定义方法 generateTexCoor，实现自动切分纹理产生纹理数组。

文件 Ball.java 的主要实现代码如下所示。

```
public class Ball {

    private IntBuffer   mVertexBuffer;
    private IntBuffer   mNormalBuffer;
```

```
private FloatBuffer mTextureBuffer;
public float mAngleX;
public float mAngleY;
public float mAngleZ;
int vCount=0;
int texId;

public Ball(int scale,int texId)
{
    this.texId=texId;
    final int UNIT_SIZE=10000;
    final float angleSpan=11.25f;//切分球的角度

    //定义获取切分图的纹理数组
    float[] texCoorArray=
     generateTexCoor
     (
            (int)(360/angleSpan),                             //列数
            (int)(180/angleSpan)                              //行数
    );
    int tc=0;//纹理数组计数器
    int ts=texCoorArray.length;//纹理数组长度

    ArrayList<Integer> alVertix=new ArrayList<Integer>();
    ArrayList<Float> alTexture=new ArrayList<Float>();

    for(float vAngle=90;vAngle>-90;vAngle=vAngle-angleSpan)
    {
        for(float hAngle=360;hAngle>0;hAngle=hAngle-angleSpan)
        {
            //构建三角形的顶点和坐标
            double xozLength=scale*UNIT_SIZE*Math.cos(Math.toRadians(vAngle));
            int x1=(int)(xozLength*Math.cos(Math.toRadians(hAngle)));
            int z1=(int)(xozLength*Math.sin(Math.toRadians(hAngle)));
            int y1=(int)(scale*UNIT_SIZE*Math.sin(Math.toRadians(vAngle)));

            xozLength=scale*UNIT_SIZE*Math.cos(Math.toRadians(vAngle-angleSpan));
            int x2=(int)(xozLength*Math.cos(Math.toRadians(hAngle)));
            int z2=(int)(xozLength*Math.sin(Math.toRadians(hAngle)));
            int y2=(int)(scale*UNIT_SIZE*Math.sin(Math.toRadians(vAngle-angleSpan)));

            xozLength=scale*UNIT_SIZE*Math.cos(Math.toRadians(vAngle-angleSpan));
            int x3=(int)(xozLength*Math.cos(Math.toRadians(hAngle-angleSpan)));
            int z3=(int)(xozLength*Math.sin(Math.toRadians(hAngle-angleSpan)));
            int y3=(int)(scale*UNIT_SIZE*Math.sin(Math.toRadians(vAngle-angleSpan)));

            xozLength=scale*UNIT_SIZE*Math.cos(Math.toRadians(vAngle));
            int x4=(int)(xozLength*Math.cos(Math.toRadians(hAngle-angleSpan)));
            int z4=(int)(xozLength*Math.sin(Math.toRadians(hAngle-angleSpan)));
            int y4=(int)(scale*UNIT_SIZE*Math.sin(Math.toRadians(vAngle)));

            //构建第一个三角形
            alVertix.add(x1);alVertix.add(y1);alVertix.add(z1);
            alVertix.add(x2);alVertix.add(y2);alVertix.add(z2);
            alVertix.add(x4);alVertix.add(y4);alVertix.add(z4);
            alVertix.add(x4);alVertix.add(y4);alVertix.add(z4);
            alVertix.add(x2);alVertix.add(y2);alVertix.add(z2);
            alVertix.add(x3);alVertix.add(y3);alVertix.add(z3);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
            alTexture.add(texCoorArray[tc++%ts]);
```

```
            alTexture.add(texCoorArray[tc++%ts]);
        }
    }
    vCount=alVertix.size()/3;//顶点的数量为坐标值数量的 1/3，因为一个顶点有 3 个坐标

    //将 alVertix 中的坐标值转存到一个 int 数组中
    int vertices[]=new int[vCount*3];
    for(int i=0;i<alVertix.size();i++)
    {
        vertices[i]=alVertix.get(i);
    }

    //创建绘制顶点数据缓冲
    ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
    vbb.order(ByteOrder.nativeOrder());
    mVertexBuffer = vbb.asIntBuffer();
    mVertexBuffer.put(vertices);
    mVertexBuffer.position(0);

    //创建顶点法向量数据缓冲
    ByteBuffer nbb = ByteBuffer.allocateDirect(vertices.length*4);
    nbb.order(ByteOrder.nativeOrder());
    mNormalBuffer = vbb.asIntBuffer();
    mNormalBuffer.put(vertices);
    mNormalBuffer.position(0);

    //创建纹理坐标缓冲
    float textureCoors[]=new float[alTexture.size()];
    for(int i=0;i<alTexture.size();i++)
    {
        textureCoors[i]=alTexture.get(i);
    }

    ByteBuffer tbb = ByteBuffer.allocateDirect(textureCoors.length*4);
    tbb.order(ByteOrder.nativeOrder());
    mTextureBuffer = tbb.asFloatBuffer();              //转换为 float 型缓冲
    mTextureBuffer.put(textureCoors);                  //向缓冲区中放入顶点纹理数据
    mTextureBuffer.position(0);                        //设置缓冲区起始位置
}

public void drawSelf(GL10 gl)
{
    gl.glRotatef(mAngleZ, 0, 0, 1);                    //沿 z 轴旋转
    gl.glRotatef(mAngleX, 1, 0, 0);                    //沿 x 轴旋转
    gl.glRotatef(mAngleY, 0, 1, 0);                    //沿 y 轴旋转

    //允许使用顶点数组
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    //为画笔指定顶点坐标数据
    gl.glVertexPointer
    (
            3,
            GL10.GL_FIXED,
            0,
            mVertexBuffer
    );

    //法向量数组
    gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);
    //为画笔指定顶点法向量数据
    gl.glNormalPointer(GL10.GL_FIXED, 0, mNormalBuffer);

    //打开纹理
    gl.glEnable(GL10.GL_TEXTURE_2D);
    //使用纹理 s、t 坐标缓冲
    gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
    //使用纹理 s、t 坐标缓冲
    gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mTextureBuffer);
    //绑定当前纹理
```

```
        gl.glBindTexture(GL10.GL_TEXTURE_2D, texId);

        //绘制图形
        gl.glDrawArrays
        (
                GL10.GL_TRIANGLES,
                0,
                vCount
        );
    }

    //自动切分纹理产生纹理数组的方法
    public float[] generateTexCoor(int bw,int bh)
    {
        float[] result=new float[bw*bh*6*2];
        float sizew=1.0f/bw;//列数
        float sizeh=1.0f/bh;//行数
        int c=0;
        for(int i=0;i<bh;i++)
        {
            for(int j=0;j<bw;j++)
            {
                //每行列 1 个矩形，由 2 个三角形构成，共 6 个点，12 个纹理坐标
                float s=j*sizew;
                float t=i*sizeh;

                result[c++]=s;
                result[c++]=t;

                result[c++]=s;
                result[c++]=t+sizeh;

                result[c++]=s+sizew;
                result[c++]=t;


                result[c++]=s+sizew;
                result[c++]=t;

                result[c++]=s;
                result[c++]=t+sizeh;

                result[c++]=s+sizew;
                result[c++]=t+sizeh;
            }
        }
        return result;
    }
}
```

到此为止，整个实例介绍完毕，执行后的效果如图 18-2 所示。

▲图 18-2　使用纹理映射实现地月模型的执行效果

18.4 实现纹理拉伸效果

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中使用纹理映射实现纹理拉伸效果的方法。

题目	目的	源码路径
实例 18-3	在手机屏幕中实现纹理拉伸效果	\daima\18\laCH

（1）编写文件 dad.java，具体实现流程如下所示。

- 声明 3 个矩形，分别贴 *s*、*t* 的最大值为 1×1、4×4、4×2 的纹理图，在场景中分别绘制 1×1、4×4、4×2 的纹理矩形。
- 设置视窗的大小、矩阵类型，并设置投影模式为透视投影。
- 定义封装方法 initTexture()以获取纹理 ID，该方法通过收取图片 ID，生成一个纹理 ID 并返回。

文件 dad.java 的主要实现代码如下所示。

```
public class dad extends GLSurfaceView{
   private SceneRenderer mRenderer;//定义场景渲染器

      public dad(Context context) {
         super(context);
         mRenderer = new SceneRenderer();                //创建渲染器
         setRenderer(mRenderer);                         //设置渲染器
         setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);//设置主动渲染模式
      }

      private class SceneRenderer implements GLSurfaceView.Renderer
      {
         int testTexId;//纹理

         laCH trOneXOne;//1×1 贴图矩形
         laCH trFourXFour;//4×4 贴图矩形
         laCH trFourXTwo;//4×2 贴图矩形

         public void onDrawFrame(GL10 gl) {
            //打开背面剪裁
            gl.glEnable(GL10.GL_CULL_FACE);
            //着色模型为平滑着色
            gl.glShadeModel(GL10.GL_SMOOTH);
            //清除颜色缓存
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
            //模式矩阵
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            //单位矩阵
            gl.glLoadIdentity();

            //推远场景
            gl.glTranslatef(0, 0, -3);
            //绘制 1×1 纹理矩形
            gl.glPushMatrix();
            gl.glTranslatef(-3.5f, 0, 0);
            trOneXOne.drawSelf(gl);
            gl.glPopMatrix();
            //绘制 4×4 纹理矩形
            gl.glPushMatrix();
            gl.glTranslatef(0, 0, 0);
            trFourXFour.drawSelf(gl);
            gl.glPopMatrix();
            //绘制 4×4 纹理矩形
            gl.glPushMatrix();
```

```
            gl.glTranslatef(3.5f, 0, 0);
            trFourXTwo.drawSelf(gl);
            gl.glPopMatrix();
        }

        public void onSurfaceChanged(GL10 gl, int width, int height) {
            //视窗大小及位置
            gl.glViewport(0, 0, width, height);
            //设置为为投影矩阵
            gl.glMatrixMode(GL10.GL_PROJECTION);
            //设置为单位矩阵
            gl.glLoadIdentity();
            //获取透视投影的比例
            float ratio = (float) width / height;
            //计算产生透视投影矩阵
            gl.glFrustumf(-ratio, ratio, -1, 1, 1, 100);
        }

        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            //抗抖动 关闭
            gl.glDisable(GL10.GL_DITHER);
            //设置使用快速模式
            gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
            //设置屏幕背景色为黑色 RGBA
            gl.glClearColor(0,0,0,0);
            //深度测试
            gl.glEnable(GL10.GL_DEPTH_TEST);
            //纹理初始化
            testTexId=initTexture(gl,R.drawable.lashen);
            //创建纹理矩形
            trOneXOne=new laCH(1.5f,1.5f,testTexId,1,1);//s-t  0-1
            trFourXFour=new laCH(1.5f,1.5f,testTexId,4,4);//s-t 4-4
            trFourXTwo=new laCH(1.5f,1.5f,testTexId,4,2);//s-t 4-2
        }
    }

    //初始化纹理
    public int initTexture(GL10 gl,int drawableId)//textureId
    {
        //生成纹理 ID
        int[] textures = new int[1];
        gl.glGenTextures(1, textures, 0);
        int currTextureId=textures[0];
        gl.glBindTexture(GL10.GL_TEXTURE_2D, currTextureId);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,
GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_NEAREST);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MAG_FILTER,GL10.
GL_LINEAR);
        //设置纹理拉伸方式为重复
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,GL10.GL_
REPEAT);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,GL10.GL_
REPEAT);

        InputStream is = this.getResources().openRawResource(drawableId);
        Bitmap bitmapTmp;
        try
        {
            bitmapTmp = BitmapFactory.decodeStream(is);
        }
        finally
        {
            try
            {
                is.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
```

```
            }
        }
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmapTmp, 0);
        bitmapTmp.recycle();

        return currTextureId;
    }
```

（2）编写文件 laCH.java，具体实现流程如下所示。

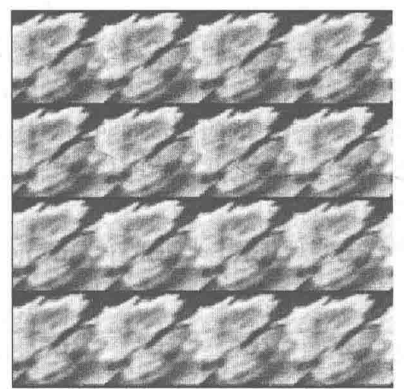
- 初始化顶点坐标数据，并创建顶点坐标数据缓冲。
- 定义方法 drawSelf()来开启纹理以绘制图形。

文件 laCH.java 的主要实现代码如下所示。

```
public class laCH {

   private FloatBuffer   mVertexBuffer;
   private FloatBuffer mTextureBuffer;
   int vCount=0;
   int texId;

   public laCH(float width,float height,int texId,float sRange,float tRange)
   {
      this.texId=texId;

      //开始初始化顶点坐标数据
      vCount=6;
      final float UNIT_SIZE=1.0f;
      float vertices[]=new float[]
      {
         width*UNIT_SIZE,height*UNIT_SIZE,0,
         -width*UNIT_SIZE,height*UNIT_SIZE,0,
         -width*UNIT_SIZE,-height*UNIT_SIZE,0,

         -width*UNIT_SIZE,-height*UNIT_SIZE,0,
         width*UNIT_SIZE,-height*UNIT_SIZE,0,
         width*UNIT_SIZE,height*UNIT_SIZE,0,
      };

      //创建顶点坐标数据缓冲
      ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
      vbb.order(ByteOrder.nativeOrder());
      mVertexBuffer = vbb.asFloatBuffer();
      mVertexBuffer.put(vertices);
      mVertexBuffer.position(0);
      //初始化纹理坐标
      float[] texST=
      {
         sRange,0,
         0,0,
         0,tRange,
         0,tRange,
         sRange,tRange,
         sRange,0
      };
      ByteBuffer tbb = ByteBuffer.allocateDirect(texST.length*4);
      tbb.order(ByteOrder.nativeOrder());
      mTextureBuffer = tbb.asFloatBuffer();
      mTextureBuffer.put(texST);
      mTextureBuffer.position(0);

   }
   public void drawSelf(GL10 gl)   {
      gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);//启用顶点坐标数组
      //设置画笔的顶点坐标
      gl.glVertexPointer
      (
            3,
```

```
                GL10.GL_FLOAT,
                0,
                mVertexBuffer
        );

        //打开纹理
        gl.glEnable(GL10.GL_TEXTURE_2D);
        //使用纹理 s、t 坐标缓冲
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        //用指定纹理设置画笔 s、t 坐标
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mTextureBuffer);
        //绑定当前纹理
        gl.glBindTexture(GL10.GL_TEXTURE_2D, texId);
        //绘制图形
        gl.glDrawArrays
        (
                GL10.GL_TRIANGLES,          //用三角形方式填充
                0,
                vCount
        );
    }
}
```

到此为止，整个实例介绍完毕，执行后的效果如图 18-3 所示。



▲图 18-3　纹理拉伸的执行效果

注意：当在实现海洋或草原等较大纹理贴图时，使用纹理的拉伸方式既可以节省内存，又可以保证画面的真实性。

第 19 章　绘制不同的三维形状

在三维世界中，所有的物体都是基于基本形状来构建的。无论多么复杂的物体，都是使用基本形状来建模的，在建模之后经过材质和渲染才成为我们眼中美轮美奂的角色和物体。在本章将通过具体实例的实现流程，详细介绍在 Android 平台中使用 OpenGL ES 技术构建基本三维形状的方法，为读者进入本书后面知识的学习打下基础。

19.1 绘制一个圆柱体

圆柱体是指在同一个平面内有一条定直线和一条动线，当这个平面绕着这条定直线旋转一周时，这条动线所成的面叫作旋转面，这条定直线叫作旋转面的轴，这条动线叫作旋转面的母线。如果母线是和轴平行的一条直线，那么所生成的旋转面叫作圆柱面。如果用垂直于轴的两个平面去截圆柱面，那么两个截面和圆柱面所围成的几何体叫作直圆柱，简称圆柱。圆柱又可以看作由一个矩形绕着它的一边旋转一周而得到的。

接下来的将通过一个具体实例的实现过程，来讲解在 Android 屏幕中绘制一个圆柱体的方法。

题目	目的	源码路径
实例 19-1	在屏幕中绘制一个圆柱体	\daima\19\zhuCH

本实例的实现流程如下所示。

（1）编写文件 Jiem.java，具体实现流程如下所示。

- 指定屏幕所要显示的界面，并对界面进行相关设置。
- 为 Activity 设置恢复处理，当 Activity 恢复设置时显示界面同样应该恢复。
- 当 Activity 暂停设置时，显示界面同样应该暂停。

文件 Jiem.java 的主要代码如下所示。

```
public class Jiem extends Activity {
    private MyGLSurfaceView mGLSurfaceView;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);

        mGLSurfaceView = new MyGLSurfaceView(this);
        setContentView(mGLSurfaceView);
        mGLSurfaceView.setFocusableInTouchMode(true);              //可触控
        mGLSurfaceView.requestFocus();                             //获取焦点
    }

    @Override
```

```
    protected void onResume() {
        super.onResume();
        mGLSurfaceView.onResume();
    }
    @Override
    protected void onPause() {
        super.onPause();
        mGLSurfaceView.onPause();
    }
}
```

（2）编写文件 MyGLSurfaceView.java，定义类 MyGLSurfaceView 实现场景加载和渲染功能，主要实现代码如下所示。

```
public class MyGLSurfaceView extends GLSurfaceView {
    private final float SUO = 180.0f/320;//缩放比例
    private SceneRenderer mRenderer;
    private float shangY;
    private float shangX;
    private int lightAngle=90;

    public MyGLSurfaceView(Context context) {
        super(context);
        mRenderer = new SceneRenderer();
        setRenderer(mRenderer);
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
    }

    //触摸事件回调方法
    @Override
    public boolean onTouchEvent(MotionEvent e) {
        float y = e.getY();
        float x = e.getX();
        switch (e.getAction()) {
        case MotionEvent.ACTION_MOVE:
            float dy = y - shangY;
            float dx = x - shangX;
            mRenderer.cylinder.AngleX += dy * SUO;
            mRenderer.cylinder.AngleZ += dx * SUO;
            requestRender();
        }
        shangY = y;
        shangX = x;
        return true;
    }
    private class SceneRenderer implements GLSurfaceView.Renderer
    {
        int textureId;        zhuCH cylinder;
        public SceneRenderer()
        {
        }
        public void onDrawFrame(GL10 gl) {
            //清除颜色缓存
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
             //设置为模式矩阵
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            //设置为单位矩阵
            gl.glLoadIdentity();
            gl.glPushMatrix();
            float lx=0;
            float ly=(float)(7*Math.cos(Math.toRadians(lightAngle)));
            float lz=(float)(7*Math.sin(Math.toRadians(lightAngle)));
            float[] positionParamsRed={lx,ly,lz,0};
            gl.glLightfv(GL10.GL_LIGHT1, GL10.GL_POSITION, positionParamsRed,0);

            initMaterial(gl);                               //纹理初始化
            gl.glTranslatef(0, 0, -10f);                    //平移
            initLight(gl);                                  //开灯
```

```
        cylinder.drawSelf(gl);                              //绘制
        closeLight(gl);                                     //关灯
        gl.glPopMatrix();                                   //恢复变换矩阵现场
    }
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        //设置视窗大小及位置
        gl.glViewport(0, 0, width, height);
        //设置当前矩阵为投影矩阵
        gl.glMatrixMode(GL10.GL_PROJECTION);
        //设置当前矩阵为单位矩阵
        gl.glLoadIdentity();
        //计算透视投影的比例
        float ratio = (float) width / height;
        //调用此方法计算产生透视投影矩阵
        gl.glFrustumf(-ratio, ratio, -1, 1, 1, 100);
    }
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
      //关闭抗抖动功能
        gl.glDisable(GL10.GL_DITHER);
        //设置 Hint 项目模式为快速模式
        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
        //设置屏幕背景色黑色 RGBA
        gl.glClearColor(0,0,0,0);
        //设置为平滑着色模型
        gl.glShadeModel(GL10.GL_SMOOTH);
        //用深度测试
        gl.glEnable(GL10.GL_DEPTH_TEST);
        textureId=initTexture(gl,R.drawable.stone);        //纹理 ID
        cylinder=new zhuCH(10f,2f,18f,textureId);           //创建圆柱体
        //开启线程来旋转光源
       new Thread()
    }
}
//白色灯初始化
private void initLight(GL10 gl)
{
    gl.glEnable(GL10.GL_LIGHTING);
    gl.glEnable(GL10.GL_LIGHT1);
    //环境光设置
    float[] ambientParams={0.2f,0.2f,0.2f,1.0f};
    gl.glLightfv(GL10.GL_LIGHT1, GL10.GL_AMBIENT, ambientParams,0);
    //散射光设置
    float[] diffuseParams={1f,1f,1f,1.0f};
    gl.glLightfv(GL10.GL_LIGHT1, GL10.GL_DIFFUSE, diffuseParams,0);
    //反射光设置
    float[] specularParams={1f,1f,1f,1.0f};
    gl.glLightfv(GL10.GL_LIGHT1, GL10.GL_SPECULAR, specularParams,0);
}
//关闭灯
private void closeLight(GL10 gl)
{
    gl.glDisable(GL10.GL_LIGHT1);
    gl.glDisable(GL10.GL_LIGHTING);
}
//材质初始化
private void initMaterial(GL10 gl)
{
    //环境光
    float ambientMaterial[] = {248f/255f, 242f/255f, 144f/255f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT, ambientMaterial,0);
    //散射光
    float diffuseMaterial[] = {248f/255f, 242f/255f, 144f/255f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE, diffuseMaterial,0);
    //高光材质
    float specularMaterial[] = {248f/255f, 242f/255f, 144f/255f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SPECULAR, specularMaterial,0);
    gl.glMaterialf(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS, 100.0f);
}
```

```
    //初始化纹理
    public int initTexture(GL10 gl,int drawableId)
    {
        //生成纹理 ID
        int[] textures = new int[1];
        gl.glGenTextures(1, textures, 0);
        int currTextureId=textures[0];
        gl.glBindTexture(GL10.GL_TEXTURE_2D, currTextureId);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,
GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_LINEAR_MIPMAP_NEAREST);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MAG_FILTER,GL10.GL_
LINEAR_MIPMAP_LINEAR);
        ((GL11)gl).glTexParameterf(GL10.GL_TEXTURE_2D,        GL11.GL_GENERATE_MIPMAP,
GL10.GL_TRUE);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,GL10.GL_REPEAT);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,GL10.GL_REPEAT);


        InputStream is = this.getResources().openRawResource(drawableId);
        Bitmap bitmapTmp;
        try
        {
            bitmapTmp = BitmapFactory.decodeStream(is);
        }
        finally
        {
            try
            {
                is.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmapTmp, 0);
        bitmapTmp.recycle();

        return currTextureId;
    }
}
```

（3）编写文件 zhuCH.java 定义圆柱类 zhuCH，在此实现了绘制三角形方法的构造器部分的代码，具体实现流程如下所示。

- 设置了圆柱体的控制属性，主要包括纹理、高度、截面半径、截面角度切分单位和高度切分单位，这些属性用于控制圆柱体的大小。
- 定义各个圆柱体绘制类的三角形绘制方法和工具方法。
- 实现圆柱体的线形绘制法，线形绘制法和三角形绘制法顶点的获取方法相同，只是采用的绘制顶点顺序和渲染方法不同，并且线形绘制没有光照和纹理贴图。

文件 zhuCH.java 的主要代码如下所示。

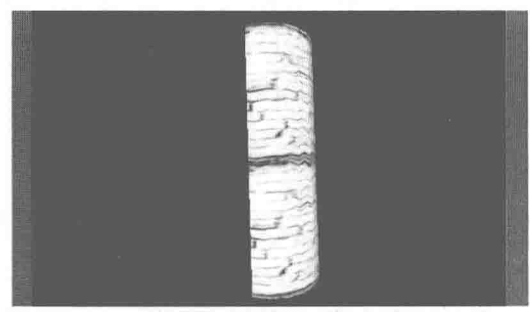
```
public class zhuCH
{
    private FloatBuffer dingBuffer;                    //缓冲顶点坐标
    private FloatBuffer myNormalBuffer;                //缓冲法向量
    private FloatBuffer weng;                          //缓冲纹理

    int textureId;
    int vCount;                                        //顶点数量
    float length;                                      //圆柱长度
    float circle_radius;                               //圆截环半径
    float degreespan;                                  //圆截环每一份的度数大小
```

```
    public float AngleX;
    public float AngleY;
    public float AngleZ;

    public zhuCH(float length,float circle_radius,float degreespan,int textureId)
    {
        this.circle_radius=circle_radius;
        this.length=length;
        this.degreespan=degreespan;
        this.textureId=textureId;

        float collength=(float)length;                    //每块圆柱所占的长度
        int spannum=(int)(360.0f/degreespan);

        ArrayList<Float> val=new ArrayList<Float>();      //顶点存放列表
        ArrayList<Float> ial=new ArrayList<Float>();      //法向量存放列表

        for(float circle_degree=180.0f;circle_degree>0.0f;circle_degree-=degreespan)
        {
                float x1 =(float)(-length/2);
                float y1=(float)(circle_radius*Math.sin(Math.toRadians(circle_degree)));
                float z1=(float)(circle_radius*Math.cos(Math.toRadians(circle_degree)));

                float a1=0;
                float b1=y1;
                float c1=z1;
                float l1=getVectorLength(a1, b1, c1);
                a1=a1/l1;                          // 规格化法向量
                b1=b1/l1;
                c1=c1/l1;

                float x2 =(float)(-length/2);
                float y2=(float)(circle_radius*Math.sin(Math.toRadians(circle_degree-
                degreespan)));
                float z2=(float) (circle_radius*Math.cos(Math.toRadians(circle_degree-
                degreespan)));

                float a2=0;
                float b2=y2;
                float c2=z2;
                float l2=getVectorLength(a2, b2, c2);
                a2=a2/l2;                              //规格化法向量
                b2=b2/l2;
                c2=c2/l2;
                float x3 =(float)(length/2);
                float y3=(float) (circle_radius*Math.sin(Math.toRadians(circle_degree-
                degreespan)));
                float z3=(float) (circle_radius*Math.cos(Math.toRadians(circle_degree-
                degreespan)));

                float a3=0;
                float b3=y3;
                float c3=z3;
                float l3=getVectorLength(a3, b3, c3);
                a3=a3/l3;                          //规格化法向量
                b3=b3/l3;
                c3=c3/l3;

                float x4 =(float)(length/2);
                float y4=(float)(circle_radius*Math.sin(Math.toRadians(circle_degree)));
                float z4=(float)(circle_radius*Math.cos(Math.toRadians(circle_degree)));

                float a4=0;
                float b4=y4;
                float c4=z4;
                float l4=getVectorLength(a4, b4, c4);
                a4=a4/l4;                          //规格化法向量
                b4=b4/l4;
                c4=c4/l4;
```

```
            val.add(x1);val.add(y1);val.add(z1);
            val.add(x2);val.add(y2);val.add(z2);
            val.add(x4);val.add(y4);val.add(z4);

            val.add(x2);val.add(y2);val.add(z2);
            val.add(x3);val.add(y3);val.add(z3);
            val.add(x4);val.add(y4);val.add(z4);

            ial.add(a1);ial.add(b1);ial.add(c1);          //顶点对应的法向量
            ial.add(a2);ial.add(b2);ial.add(c2);
            ial.add(a4);ial.add(b4);ial.add(c4);

            ial.add(a2);ial.add(b2);ial.add(c2);
            ial.add(a3);ial.add(b3);ial.add(c3);
            ial.add(a4);ial.add(b4);ial.add(c4);
        }
        vCount=val.size()/3;                              //确定顶点数量

        //顶点
        float[] vertexs=new float[vCount*3];
        for(int i=0;i<vCount*3;i++)
        {
            vertexs[i]=val.get(i);
        }
        ByteBuffer vbb=ByteBuffer.allocateDirect(vertexs.length*4);
        vbb.order(ByteOrder.nativeOrder());
        dingBuffer=vbb.asFloatBuffer();
        dingBuffer.put(vertexs);
        dingBuffer.position(0);
        //法向量
        float[] normals=new float[vCount*3];
        for(int i=0;i<vCount*3;i++)
        {
            normals[i]=ial.get(i);
        }
        ByteBuffer ibb=ByteBuffer.allocateDirect(normals.length*4);
        ibb.order(ByteOrder.nativeOrder());
        myNormalBuffer=ibb.asFloatBuffer();
        myNormalBuffer.put(normals);
        myNormalBuffer.position(0);
        //纹理
        float[] textures=generateTexCoor(spannum);
        ByteBuffer tbb=ByteBuffer.allocateDirect(textures.length*4);
        tbb.order(ByteOrder.nativeOrder());
        weng=tbb.asFloatBuffer();
        weng.put(textures);
        weng.position(0);
    }

    public void drawSelf(GL10 gl)
    {
        gl.glRotatef(AngleX, 1, 0, 0);                    //旋转处理
        gl.glRotatef(AngleY, 0, 1, 0);
        gl.glRotatef(AngleZ, 0, 0, 1);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);     //打开顶点缓冲
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, dingBuffer);

        gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);     //打开法向量缓冲
        gl.glNormalPointer(GL10.GL_FLOAT, 0, myNormalBuffer);

        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, weng);
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);

        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, vCount);    //绘制图像
        //关闭缓冲
```

```
        gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glDisableClientState(GL10.GL_NORMAL_ARRAY);
    }
    //此方法可以计算模长度
    public float getVectorLength(float x,float y,float z)
    {
        float pingfang=x*x+y*y+z*z;
        float length=(float) Math.sqrt(pingfang);
        return length;
    }
    //自动切分纹理
    public float[] generateTexCoor(int bh)
    {
        float[] result=new float[bh*6*2];
        float REPEAT=2;
        float sizeh=1.0f/bh;                          //行数
        int c=0;
        for(int i=0;i<bh;i++)
        {
                //设置每行列一个矩形
                float t=i*sizeh;
                result[c++]=0;
                result[c++]=t;
                result[c++]=0;
                result[c++]=t+sizeh;
                result[c++]=REPEAT;
                result[c++]=t;
                result[c++]=0;
                result[c++]=t+sizeh;
                result[c++]=REPEAT;
                result[c++]=t+sizeh;
                result[c++]=REPEAT;
                result[c++]=t;
        }
        return result;
    }
}
```

到此为止，整个实例介绍完毕，执行之后的效果如图 19-1 所示。



▲图 19-1　绘制圆柱体的执行效果

19.2 绘制一个圆环

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中绘制一个圆环的方法。

题目	目的	源码路径
实例 19-2	在屏幕中绘制一个圆环	\daima\19\huanCH

本实例的实现流程如下所示。

编写文件 HuanCH.java，具体实现流程如下所示。

- 设置圆锥曲面的控制属性，包括纹理、环半径、截面半径、环角度切分单位和截面角度切分单位。
- 通过上述属性可以控制圆环曲面的大小，并获取网格顶点坐标；最后设置顶点、纹理、法向量缓冲，并定义绘制方法 drawSelf()。

文件 HuanCH.java 的主要代码如下所示。

```
public class HuanCH
{
   private FloatBuffer ding;// 缓冲顶点
   private FloatBuffer weng;// 缓冲纹理
   private FloatBuffer myNormalBuffer;// 缓冲法向量

   int vcount;
   int textureid;

   float rSpan;
   float cSpan;

   float ring_Radius;
   float circle_Radius;

   public float AngleX;
   public float AngleY;
   public float AngleZ;

   public HuanCH(float rSpan,float cSpan,float ring_Radius,float circle_Radius,int
textureid)
   {
       this.rSpan=rSpan;
       this.cSpan=cSpan;
       this.circle_Radius=circle_Radius;
       this.ring_Radius=ring_Radius;
       this.textureid=textureid;

       ArrayList<Float> val=new ArrayList<Float>();
       ArrayList<Float> ial=new ArrayList<Float>();//法向量存放列表

       for(float circle_Degree=50f;circle_Degree<130f;circle_Degree+=cSpan)
       {
           for(float ring_Degree=-90f;ring_Degree<0f;ring_Degree+=rSpan)
           {
               float x1=(float)((ring_Radius+circle_Radius*Math.cos(Math.toRadians(circle_
Degree)))*Math.cos(Math.toRadians(ring_Degree)));
               float y1=(float)(circle_Radius*Math.sin(Math.toRadians(circle_Degree)));
               float z1=(float)((ring_Radius+circle_Radius*Math.cos(Math.toRadians(circle_
Degree))) *Math.sin(Math.toRadians(ring_Degree)));

               float x2=(float)((ring_Radius+circle_Radius*Math.cos(Math.toRadians
(circle_Degree)))*Math.cos(Math.toRadians(ring_Degree+rSpan)));
               float y2=(float)(circle_Radius*Math.sin(Math.toRadians(circle_Degree)));
               float z2=(float)((ring_Radius+circle_Radius*Math.cos(Math.toRadians
(circle_Degree)))*Math.sin(Math.toRadians(ring_Degree+rSpan)));
               float x3=(float)((ring_Radius+circle_Radius*Math.cos(Math.toRadians(circle_
Degree+cSpan)))*Math.cos(Math.toRadians(ring_Degree+rSpan)));
               float y3=(float)(circle_Radius*Math.sin(Math.toRadians(circle_Degree+cSpan)));
               float z3=(float)((ring_Radius+circle_Radius*Math.cos(Math.toRadians
(circle_Degree+cSpan)))*Math.sin(Math.toRadians(ring_Degree+rSpan)));

               float x4=(float)((ring_Radius+circle_Radius*Math.cos(Math.toRadians(circle_
Degree+cSpan)))*Math.cos(Math.toRadians(ring_Degree)));
               float y4=(float)(circle_Radius*Math.sin(Math.toRadians(circle_Degree+
cSpan)));
               float z4=(float)((ring_Radius+circle_Radius*Math.cos(Math.toRadians(circle_
```

```
Degree+cSpan)))*Math.sin(Math.toRadians(ring_Degree)));

          val.add(x1);val.add(y1);val.add(z1);
          val.add(x4);val.add(y4);val.add(z4);
          val.add(x2);val.add(y2);val.add(z2);

          val.add(x2);val.add(y2);val.add(z2);
          val.add(x4);val.add(y4);val.add(z4);
          val.add(x3);val.add(y3);val.add(z3);
          //顶点圆截面中心组成圆环上点的坐标
          float a1=(float)(x1-(ring_Radius*Math.cos(Math.toRadians(ring_Degree))));
          float b1=y1-0;
          float c1=(float)(z1-(ring_Radius*Math.sin(Math.toRadians(ring_Degree))));
          float l1=getVectorLength(a1, b1, c1);          //模长
          a1=a1/l1;                                        //规格化法向量
          b1=b1/l1;
          c1=c1/l1;

          float a2=(float)(x2-(ring_Radius*Math.cos(Math.toRadians(ring_Degree+rSpan))));
          float b2=y1-0;
          float c2=(float)(z2-(ring_Radius*Math.sin(Math.toRadians(ring_Degree+rSpan))));
          float l2=getVectorLength(a2, b2, c2);          //模长
          a2=a2/l2;                                        //规格化法向量
          b2=b2/l2;
          c2=c2/l2;

          float a3=(float)(x3-(ring_Radius*Math.cos(Math.toRadians(ring_Degree+rSpan))));
          float b3=y1-0;
          float c3=(float)(z3-(ring_Radius*Math.sin(Math.toRadians(ring_Degree+rSpan))));
          float l3=getVectorLength(a3, b3, c3);          //模长
          a3=a3/l3;
          b3=b3/l3;
          c3=c3/l3;

          float a4=(float)(x4-(ring_Radius*Math.cos(Math.toRadians(ring_Degree))));
          float b4=y1-0;
          float c4=(float)(z4-(ring_Radius*Math.sin(Math.toRadians(ring_Degree))));
          float l4=getVectorLength(a4, b4, c4);
          a4=a4/l4;                                        //规格化法向量
          b4=b4/l4;
          c4=c4/l4;

          ial.add(a1);ial.add(b1);ial.add(c1);           //顶点的法向量
          ial.add(a2);ial.add(b2);ial.add(c2);
          ial.add(a4);ial.add(b4);ial.add(c4);

          ial.add(a2);ial.add(b2);ial.add(c2);
          ial.add(a3);ial.add(b3);ial.add(c3);
          ial.add(a4);ial.add(b4);ial.add(c4);
      }
   }
   vcount=val.size()/3;
   float[] vertexs=new float[vcount*3];
   for(int i=0;i<vcount*3;i++)
   {
      vertexs[i]=val.get(i);
   }
   ByteBuffer vbb=ByteBuffer.allocateDirect(vertexs.length*4);
   vbb.order(ByteOrder.nativeOrder());
   ding=vbb.asFloatBuffer();
   ding.put(vertexs);
   ding.position(0);

   //法向量
   float[] normals=new float[vcount*3];
   for(int i=0;i<vcount*3;i++)
   {
      normals[i]=ial.get(i);
   }
```

```
    ByteBuffer ibb=ByteBuffer.allocateDirect(normals.length*4);
    ibb.order(ByteOrder.nativeOrder());
    myNormalBuffer=ibb.asFloatBuffer();
    myNormalBuffer.put(normals);
    myNormalBuffer.position(0);

    //纹理

    int row=(int) (360.0f/cSpan);
    int col=(int) (360.0f/rSpan);
    float[] textures=generateTexCoor(row,col);

    ByteBuffer tbb=ByteBuffer.allocateDirect(textures.length*4);
    tbb.order(ByteOrder.nativeOrder());
    weng=tbb.asFloatBuffer();
    weng.put(textures);
    weng.position(0);
}
//开始绘制
public void drawSelf(GL10 gl)
{
    gl.glRotatef(AngleX, 1, 0, 0);//旋转
    gl.glRotatef(AngleY, 0, 1, 0);
    gl.glRotatef(AngleZ, 0, 0, 1);

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, ding);

    gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);//打开法向量缓冲
    gl.glNormalPointer(GL10.GL_FLOAT, 0, myNormalBuffer);//设置法向量缓冲

    gl.glEnable(GL10.GL_TEXTURE_2D);
    gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
    gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, weng);
    gl.glBindTexture(GL10.GL_TEXTURE_2D, textureid);

    gl.glDrawArrays(GL10.GL_TRIANGLES, 0, vcount);

    gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);// 缓冲关闭
    gl.glEnable(GL10.GL_TEXTURE_2D);
    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glDisableClientState(GL10.GL_NORMAL_ARRAY);
}

//自切分纹理产生纹理数组
public float[] generateTexCoor(int bw,int bh)
{
    float[] result=new float[bw*bh*6*2];
    float sizew=1.0f/bw;                                    //列数
    float sizeh=1.0f/bh;                                    //行数
    int c=0;
    for(int i=0;i<bh;i++)
    {
        for(int j=0;j<bw;j++)
        {
            //每行列一个矩形，由两个三角形构成
            float s=j*sizew;
            float t=i*sizeh;

            result[c++]=s;
            result[c++]=t;

            result[c++]=s;
            result[c++]=t+sizeh;

            result[c++]=s+sizew;
            result[c++]=t;
            result[c++]=s+sizew;
            result[c++]=t;
```

```
                result[c++]=s;
                result[c++]=t+sizeh;
                result[c++]=s+sizew;
                result[c++]=t+sizeh;
            }
        }
        return result;
    }

    //法向量规格化，求模长度
    public float getVectorLength(float x,float y,float z)
    {
        float pingfang=x*x+y*y+z*z;
        float length=(float) Math.sqrt(pingfang);
        return length;
    }
}
```

在上述代码中，rSpan 表示环每一份多少度，cSpan 表示圆截环每一份多少度，ring_Radius 表示环半径，circle_Radius 表示圆截面半径。执行后的效果如图 19-2 所示。

▲图 19-2　绘制圆环的执行效果

19.3 绘制一个抛物面效果

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中绘制一个抛物面的方法。

题目	目的	源码路径
实例 19-3	在屏幕中绘制一个抛物面效果	\daima\19\paoCH

本实例的实现文件是 DrawpaoCH.java，具体实现流程如下所示。

- 设置抛物面的属性参数，通过这些参数来控制抛物面的形状和开口大小。
- 根据数学公式，运用双循环来获取抛物面网格上顶点的坐标，并将坐标存放到列表中，这样可以设置顶点缓冲。
- 获取顶点数组并设置顶点缓冲来绘制图像，并编写绘制图像的方法。

文件 DrawpaoCH.java 的主要代码如下所示。

```
    public DrawpaoCH(float height,float a,float b,int col,float angleSpan,int textureId)
    {
        this.height=height;
        this.a=a;
        this.b=b;
        this.col=col;
        this.angleSpan=angleSpan;
        this.textureId=textureId;

        float heightSpan=height/col;                              //每块抛物体的高度
        int spannum=(int)(360.0f/angleSpan);                      //切分截面份数

        ArrayList<Float> val=new ArrayList<Float>();              //存放顶点的列表
        //ArrayList<Float> ial=new ArrayList<Float>();            //存放法向量的列表

        for(float h=height;h>0;h-=heightSpan)                     //for 循环行
        {
            for(float angle=360;angle>0;angle-=angleSpan)         //for 循环列
            {
                float x1=(float) (a*Math.sqrt(2*h)*Math.cos(Math.toRadians(angle)));
                float y1=h;
                float z1=(float) (b*Math.sqrt(2*h)*Math.sin(Math.toRadians(angle)));

                float  x2=(float)(a*Math.sqrt(2*(h-heightSpan))*Math.cos(Math.toRadians
```

```
(angle)));
                float y2=h-heightSpan;
                float  z2=(float)(b*Math.sqrt(2*(h-heightSpan))*Math.sin(Math.toRadians
(angle)));

                float  x3=(float)(a*Math.sqrt(2*(h-heightSpan))*Math.cos(Math.toRadians
(angle-angleSpan)));
                float y3=h-heightSpan;
                float  z3=(float)(b*Math.sqrt(2*(h-heightSpan))*Math.sin(Math.toRadians
(angle-angleSpan)));

                float x4=(float)(a*Math.sqrt(2*h)*Math.cos(Math.toRadians(angle-angleSpan)));
                float y4=h;
                float z4=(float)(b*Math.sqrt(2*h)*Math.sin(Math.toRadians(angle-angleSpan)));

                val.add(x1);val.add(y1);val.add(z1);//2 个三角形需要 6 个坐标
                val.add(x2);val.add(y2);val.add(z2);
                val.add(x4);val.add(y4);val.add(z4);

                val.add(x2);val.add(y2);val.add(z2);
                val.add(x3);val.add(y3);val.add(z3);
                val.add(x4);val.add(y4);val.add(z4);
            }
        }

        vCount=val.size()/3;//顶点数量
        float[] vertexs=new float[vCount*3];
        for(int i=0;i<vCount*3;i++)
        {
            vertexs[i]=val.get(i);
        }
        ByteBuffer vbb=ByteBuffer.allocateDirect(vertexs.length*4);
        vbb.order(ByteOrder.nativeOrder());
        dingBuffer=vbb.asFloatBuffer();
        dingBuffer.put(vertexs);
        dingBuffer.position(0);
        //纹理
        float[] textures=generateTexCoor(col,spannum);
        ByteBuffer tbb=ByteBuffer.allocateDirect(textures.length*4);
        tbb.order(ByteOrder.nativeOrder());
        weng=tbb.asFloatBuffer();
        weng.put(textures);
        weng.position(0);
    }

    public void drawSelf(GL10 gl)
    {
        gl.glRotatef(AngleX, 1, 0, 0);                      //旋转
        gl.glRotatef(AngleY, 0, 1, 0);
        gl.glRotatef(AngleZ, 0, 0, 1);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, dingBuffer);
        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, weng);
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);

        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, vCount);

        gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);// 缓冲关闭
        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    }

    //求模长度
    public float getVectorLength(float x,float y,float z)
    {
        float pingfang=x*x+y*y+z*z;
```

```
        float length=(float) Math.sqrt(pingfang);
        return length;
    }

    //自动切分纹理数组
    public float[] generateTexCoor(int hang,int lie)
    {
        float[] result=new float[hang*lie*6*2];
        float sizeh=1.0f/hang;//行的大小单位
        float sizel=1.0f/lie;//列的大小单位
        int c=0;
        for(int i=0;i<hang;i++)
        {
            for(int j=0;j<lie;j++)
            {

                float h=i*sizeh;
                float l=j*sizel;
                result[c++]=l;
                result[c++]=h;
                result[c++]=l;
                result[c++]=h+sizeh;
                result[c++]=l+sizel;
                result[c++]=h;
                result[c++]=l;
                result[c++]=h+sizeh;
                result[c++]=l+sizel;
                result[c++]=h+sizeh;
                result[c++]=l+sizel;
                result[c++]=h;
            }
        }
        return result;
    }
```

在上述代码中，每行、列构成的矩形都是由 2 个三角形构成，一共 6 个点，12 个纹理坐标。执行后的效果如图 19-3 所示。

▲图 19-3 绘制抛物面的执行效果

19.4 绘制一个螺旋面效果

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中绘制一个螺旋面的方法。

题目	目的	源码路径
实例 19-4	在屏幕中绘制一个螺旋面效果	\daima\19\luoxuanCH

本实例的实现文件是 DrawluoxuanCH.java，具体实现流程如下所示。

- 设置螺绘制旋面的属性，通过控制这些属性可以控制螺旋面的形状和大小。
- 根据数学原理对组成螺旋面的网格顶点进行循环处理，并记录顶点的坐标和对应顶点的法向量。
- 分别获取纹理切分的行列数以切分纹理，并分别设置顶点、法向量和纹理的缓冲，并实现绘制方法的编码。

文件 DrawluoxuanCH.java 的主要代码如下所示。

```
    public DrawluoxuanCH
          (       float jie_R,
                  float Luo_R,
                  int textureid
          )
        {
        this.jie_R=jie_R;
```

```
        this.Luo_R=Luo_R;
        this.textureid=textureid;

        ArrayList<Float> val=new ArrayList<Float>();
        ArrayList<Float> ial=new ArrayList<Float>();

        for(float
h_angle=0,c_r=jie_R,h_r=Luo_R,length=0;h_angle<MAX_ANGLE;h_angle+=HEDICOID_ANGLE_SPA
N,c_r+=CIRCLE_R_SPAN,h_r+=HEDICOID_R_SPAN,length+=LENGTH_SPAN)
        {
            for(float
c_angle=CIECLE_ANGLE_BEGIN;c_angle<CIECLE_ANGLE_OVER;c_angle+=CIRCLE_ANGLE_SPAN)
            {
                float x1=(float)((h_r+c_r*Math.cos(Math.toRadians(c_angle)))*Math.cos(Math.
                toRadians(h_angle)));
                float y1=(float) (c_r*Math.sin(Math.toRadians(c_angle))+length);
                float z1=(float)((h_r+c_r*Math.cos(Math.toRadians(c_angle)))*Math.sin(Math.
                toRadians(h_angle)));

                float x2=(float)((h_r+c_r*Math.cos(Math.toRadians(c_angle+CIRCLE_ANGLE_
                SPAN)))*Math. cos(Math.toRadians(h_angle)));
                float y2=(float)(c_r*Math.sin(Math.toRadians(c_angle+CIRCLE_ANGLE_SPAN))
                +length);
                float z2=(float)((h_r+c_r*Math.cos(Math.toRadians(c_angle+CIRCLE_ANGLE_
                SPAN)))*Math.sin(Math.toRadians(h_angle)));

                float x3=(float)(((h_r+HEDICOID_R_SPAN)+(c_r+CIRCLE_R_SPAN)*Math.cos
(Math.toRadians(c_angle+CIRCLE_ANGLE_SPAN)))*Math.cos(Math.toRadians(h_angle+HEDICOID
_ANGLE_SPAN)));
                float y3=(float)((c_r+CIRCLE_R_SPAN)*Math.sin(Math.toRadians(c_angle+
                CIRCLE_ANGLE_SPAN))+(length+LENGTH_SPAN));
                float z3=(float)(((h_r+HEDICOID_R_SPAN)+(c_r+CIRCLE_R_SPAN)*Math.cos
(Math.toRadians(c_angle+CIRCLE_ANGLE_SPAN)))*Math.sin(Math.toRadians(h_angle+HEDICOID
_ANGLE_SPAN)));

                float x4=(float)(((h_r+HEDICOID_R_SPAN)+(c_r+CIRCLE_R_SPAN)*Math.
cos(Math.toRadians(c_angle)))*Math.cos(Math.toRadians(h_angle+HEDICOID_ANGLE_SPAN)));
                float y4=(float) ((c_r+CIRCLE_R_SPAN)*Math.sin(Math.toRadians(c_angle))+
                 (length+LENGTH_SPAN));
                float z4=(float)(((h_r+HEDICOID_R_SPAN)+(c_r+CIRCLE_R_SPAN)*Math.cos
(Math.toRadians(c_angle)))*Math.sin(Math.toRadians(h_angle+HEDICOID_ANGLE_SPAN)));

                val.add(x1);val.add(y1);val.add(z1);
                val.add(x2);val.add(y2);val.add(z2);
                val.add(x4);val.add(y4);val.add(z4);

                val.add(x2);val.add(y2);val.add(z2);
                val.add(x3);val.add(y3);val.add(z3);
                val.add(x4);val.add(y4);val.add(z4);

                //顶点圆截面中心组成圆环点坐标
                float a1=(float) (x1-(h_r*Math.cos(Math.toRadians(h_angle))));
                float b1=y1-length;
                float c1=(float) (z1-(h_r*Math.sin(Math.toRadians(h_angle))));
                float l1=getVectorLength(a1, b1, c1);
                a1=a1/l1;// 规格化法向量
                b1=b1/l1;
                c1=c1/l1;

                float a2=(float) (x2-(h_r*Math.cos(Math.toRadians(h_angle))));
                float b2=y1-length;
                float c2=(float) (z2-(h_r*Math.sin(Math.toRadians(h_angle))));
                float l2=getVectorLength(a2, b2, c2);
                a2=a2/l2;// 规格化法向量
                b2=b2/l2;
                c2=c2/l2;

                float a3=(float)(x3-(h_r*Math.cos(Math.toRadians(h_angle+HEDICOID_ANGLE_
                SPAN))));
```

```
                float b3=y1-(length+LENGTH_SPAN);
                float c3=(float)(z3-(h_r*Math.sin(Math.toRadians(h_angle+HEDICOID_ANGLE_
                SPAN))));
                float l3=getVectorLength(a3, b3, c3);
                a3=a3/l3;// 规格化法向量
                b3=b3/l3;
                c3=c3/l3;

                float a4=(float)(x4-(h_r*Math.cos(Math.toRadians(h_angle+HEDICOID_ANGLE_
                SPAN))));
                float b4=y1-(length+LENGTH_SPAN);
                float c4=(float)(z4-(h_r*Math.sin(Math.toRadians(h_angle+HEDICOID_ANGLE_
                SPAN))));
                float l4=getVectorLength(a4, b4, c4);
                a4=a4/l4;// 规格化法向量
                b4=b4/l4;
                c4=c4/l4;

                ial.add(a1);ial.add(b1);ial.add(c1);//顶点对应的法向量
                ial.add(a2);ial.add(b2);ial.add(c2);
                ial.add(a4);ial.add(b4);ial.add(c4);

                ial.add(a2);ial.add(b2);ial.add(c2);
                ial.add(a3);ial.add(b3);ial.add(c3);
                ial.add(a4);ial.add(b4);ial.add(c4);
            }
        }
        vcount=val.size()/3;
        float[] vertexs=new float[vcount*3];
        for(int i=0;i<vcount*3;i++)
        {
            vertexs[i]=val.get(i);
        }
        ByteBuffer vbb=ByteBuffer.allocateDirect(vertexs.length*4);
        vbb.order(ByteOrder.nativeOrder());
        ding=vbb.asFloatBuffer();
        ding.put(vertexs);
        ding.position(0);

        //法向量
        float[] normals=new float[vcount*3];
        for(int i=0;i<vcount*3;i++)
        {
            normals[i]=ial.get(i);
        }
        ByteBuffer ibb=ByteBuffer.allocateDirect(normals.length*4);
        ibb.order(ByteOrder.nativeOrder());
        myNormalBuffer=ibb.asFloatBuffer();
        myNormalBuffer.put(normals);
        myNormalBuffer.position(0);

        //纹理
        int col=(int) (MAX_ANGLE/HEDICOID_ANGLE_SPAN);
        int row=(int) ((CIECLE_ANGLE_OVER-CIECLE_ANGLE_BEGIN)/CIRCLE_ANGLE_SPAN);
        float[] textures=generateTexCoor(row,col);

        ByteBuffer tbb=ByteBuffer.allocateDirect(textures.length*4);
        tbb.order(ByteOrder.nativeOrder());
        weng=tbb.asFloatBuffer();
        weng.put(textures);
        weng.position(0);
    }

    public void drawSelf(GL10 gl)
    {
        gl.glRotatef(AngleX, 1, 0, 0);//旋转
        gl.glRotatef(AngleY, 0, 1, 0);
        gl.glRotatef(AngleZ, 0, 0, 1);
```

```
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, ding);

        gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);//打开法向量缓冲
        gl.glNormalPointer(GL10.GL_FLOAT, 0, myNormalBuffer);//设置法向量缓冲

        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, weng);
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textureid);

        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, vcount);

        gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);// 缓冲关闭
        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glDisableClientState(GL10.GL_NORMAL_ARRAY);
        gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    }

    //自动切分纹理的数组方法
    public float[] generateTexCoor(int hang,int lie)
    {
        float[] result=new float[hang*lie*6*2];
        float sizeh=1.0f/hang;//行的大小单位
        float sizel=1.0f/lie;//列的大小单位
        int c=0;
        for(int i=0;i<lie;i++)
        {
            for(int j=0;j<hang;j++)
            {
                float h=j*sizeh;
                float l=i*sizel;

                result[c++]=h;//1
                result[c++]=l;

                result[c++]=h+sizeh;//2
                result[c++]=l;

                result[c++]=h;//4
                result[c++]=l+sizel;

                result[c++]=h+sizeh;//2
                result[c++]=l;

                result[c++]=h+sizeh;//3
                result[c++]=l+sizel;

                result[c++]=h;//4
                result[c++]=l+sizel;
            }
        }
        return result;
    }
    //求模的长度
    public float getVectorLength(float x,float y,float z)
    {
        float pingfang=x*x+y*y+z*z;
        float length=(float) Math.sqrt(pingfang);
        return length;
    }

}
```

在上述代码中，每行、列一个矩形，由 2 个三角形构成，共 6 个点，12 个纹理坐标。执行后的效果如图 19-4 所示。

▲图 19-4　绘制螺旋面的执行效果

第 20 章　坐标变换和混合

在三维世界中，物体的移动是基于坐标变换来实现的。另外，使用混合技术可以实现雾蒙蒙的效果，使游戏效果更加显得真实。在本章的内容中，将详细介绍使用 OpenGL ES 技术实现坐标变换和混合处理的基本知识，为读者进入本书后面知识的学习打下基础。

20.1 实现坐标变换

坐标变换是指采用一定的数学方法将一种坐标系的坐标变换为另一种坐标系的坐标的过程。在本节的内容中，将简要介绍使用 OpenGL ES 实现坐标变换的基本知识。

20.1.1 坐标变换基础

在使用 OpenGL ES 绘制物体的时候，有时候需要在不同的位置绘制物体，有时候绘制的物体需要有不同的角度，这就需要平移或旋转技术。在平移或旋转的时候，会给观察者带来平移或旋转物体的感觉，但其实是平移或旋转了坐标系，物体相对于坐标系平移或旋转。坐标变换是以矩阵的形式存储的，要完成这种类型的操作，矩阵堆栈就是一种理想的机制。

在 OpenGL ES 中可以调用方法 glPushMatrix()和 glPopMatrix 来操作堆栈。glPushMatrix 表示复制一份当前矩阵，并把复制的矩阵添加到堆栈的顶部；glPopMatrix 表示丢弃堆栈顶部的那个矩阵。我们可以认为 glPushMatrix 表示记录下当前的坐标位置，经过一系列的平移、旋转变换之后，可以调用 glPopMatrix 以便回到原来的坐标位置。假如绘制一个游戏角色，就可以首先绘制机器人躯干，执行 glPushMatrix，记下自己的位置；然后移到角色左臂并绘制，执行 glPopMatrix，丢弃上次的平移变换；最后使自己回到角色的原点位置，执行 glPushMatrix，记住自己的位置，移动到机器人右臂。类似地，绘制每个部位都进行如上操作，就绘制好了游戏角色。

在三维世界中的坐标变换有两类，分别是缩放变换和平移变换。在本章的内容中，将通过具体实例的实现过程来讲解实现变换效果的流程。

20.1.2 实现缩放变换

通过缩放变换可以改变物体的大小，把当前矩阵与一个表示沿各个坐标轴对物体进行拉伸、收缩和反射的矩阵相乘。缩放的矩阵可以简单地如图 20-1 所示。

$$
\begin{matrix}
x & 0 & 0 & 0 \\
0 & y & 0 & 0 \\
0 & 0 & z & 0 \\
0 & 0 & 0 & 1
\end{matrix}
$$

▲图 20-1　缩放矩阵图

在图 20-1 所示的缩放矩阵图中，包含了 x、y 和 z 一共 3 个缩放因子，分别对应 x 轴、y 轴和 z 轴，缩放变换是关于原点的缩放。

在 OpenGL ES 中，通过方法 glScalex(int x，int y，int z)和 glScalef(float x，float y,float z)实现物体的缩放变换，表示把当前矩阵与一个表示沿各个轴对物体进行拉伸、收缩和放射的矩阵相乘，这个物体中的每个点的 x、y 和 z 坐标与对应的 x、y 和 z 参数相乘。

如果缩放值大于 1.0，它就拉伸物体；如果缩放值小于 1.0，它就收缩物体；如果缩放值为-1.0，它就反射这个物体。（1.0，1.0，1.0）是单位缩放值。

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中实现缩放变换效果的方法。

题目	目的	源码路径
实例 20-1	在 Android 手机屏幕中实现缩放变换效果	\daima\20\suoCH

本实例的实现流程如下所示。

（1）编写文件 ddd.java，首先开启一个自动缩放椭球体的大小的线程，并分别改变椭球体的缩放值和设置椭球缩放的范围；然后重写 onDrawFrame（GLlo gl）方法以绘制椭球体。文件 ddd.java 的主要实现代码如下所示。

```
class ddd extends GLSurfaceView {
   private SceneRenderer mRenderer;//渲染场景
   public float gai;
   public ddd(Context context) {
       super(context);
       mRenderer = new SceneRenderer();              //创建渲染器
       setRenderer(mRenderer);                       //设置渲染器
       setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);//设置为主动渲染
   }
   private class SceneRenderer implements GLSurfaceView.Renderer
   {
       suoCH ball=new suoCH(3);

       public SceneRenderer()
       {
          new Thread()
           {
           public void run()
               {

                   while(true)
                   {
                     ball.scaleX+=gai;
                         ball.scaleY+=gai;
                      if(ball.scaleX>1.5)
                      {
                         gai=-0.02f;
                      }
                      if(ball.scaleX<0.5)
                      {
                         gai+=0.02f;
                      }
                     try
                   {
                       Thread.sleep(50);//每隔 50ms 重绘一次
                   }
                   catch(Exception e)
                   {
                       e.printStackTrace();
                   }
                 }
               }
           }.start();
```

```
        }
        public void onDrawFrame(GL10 gl) {

            //清除颜色缓存
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
            //设置为模式矩阵
            gl.glMatrixMode(GL10.GL_MODELVCHIEW);
            //设置单位矩阵
            gl.glLoadIdentity();
            gl.glTranslatef(0, 0, -5);
            ball.drawSelf(gl);
        }

        public void onSurfaceChanged(GL10 gl, int width, int height) {
            //设置视窗大小及位置
            gl.glvCHiewport(0, 0, width, height);
            //设置当前矩阵为投影矩阵
            gl.glMatrixMode(GL10.GL_PROJECTION);
            //设置当前矩阵为单位矩阵
            gl.glLoadIdentity();
            gl.glShadeModel(GL10.GL_SMOOTH);
            //计算透视投影的比例
            float ratio = (float) width / height;
            //调用此方法计算产生透视投影矩阵
            gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);
        }

        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            //关闭抗抖动功能
            gl.glDisable(GL10.GL_DITHER);
            //设置特定 Hint 模式为快速模式
            gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
            //设置屏幕背景色黑色 RGBA
            gl.glClearColor(0,0,0,0);
            //设置为平滑着色
            gl.glShadeModel(GL10.GL_SMOOTH);//GL10.GL_SMOOTH  GL10.GL_FLAT
            //启用深度测试
            gl.glEnable(GL10.GL_DEPTH_TEST);
        }
    }
}
```

（2）编写文件 suoCH.java，在此定义绘制椭圆球的类 suoCH，主要实现代码如下所示。

```
    public suoCH(int scale)
    {
        //初始化顶点坐标数据
        final double a=2;
        final double b=1.5;
        final double c=1.5;
        final int UNIT_SIZE=10000;
        ArrayList<Integer> alvCHertix=new ArrayList<Integer>();
        final int angleSpan=18;//单位切分球的角度
        for(int vAngle=-90;vAngle<=90;vAngle=vAngle+angleSpan)
        {
            for(int hAngle=0;hAngle<360;hAngle=hAngle+angleSpan)
            {//纵向、横向分别计算此点在球面上的坐标
                int x=(int)(a*scale*UNIT_SIZE*Math.cos(Math.toRadians(vAngle))*Math.cos
(Math.toRadians(hAngle)));
                int y=(int)(b*scale*UNIT_SIZE*Math.cos(Math.toRadians(vAngle))*Math.sin
(Math.toRadians(hAngle)));
                int z=(int)(c*scale*UNIT_SIZE*Math.sin(Math.toRadians(vAngle)));
                //将计算结果 x、y、z 坐标加入存放顶点坐标的 ArrayList
                alvCHertix.add(x);alvCHertix.add(y);alvCHertix.add(z);
            }
        }
        vCount=alvCHertix.size()/3;

        //将 alvCHertix 坐标值转存到一个 int 数组中
```

```
    int vertices[]=new int[vCount*3];
    for(int i=0;i<alvCHertix.size();i++)
    {
        vertices[i]=alvCHertix.get(i);
    }
    //创建顶点坐标数据缓冲
    ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
    vbb.order(ByteOrder.nativeOrder());
    ding = vbb.asIntBuffer();
    ding.put(vertices);
    ding.position(0);

    //初始化顶点着色数据
    final int one = 65535;
    int colors[]=new int[vCount*4];//顶点颜色值数组
    for(int i=0;i<vCount;i++)
    {//随机生成每个顶点的颜色
        colors[i*4]=(int) (one*Math.random());
        colors[i*4+1]=(int) (one*Math.random());
        colors[i*4+2]=(int) (one*Math.random());
        colors[i*4+3]=0;//alpha 色彩
    }

    //创建顶点着色数据缓冲
    ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length*4);
    cbb.order(ByteOrder.nativeOrder());
    se = cbb.asIntBuffer();
    se.put(colors);
    se.position(0);

    //初始化三角形构造索引数据
    ArrayList<Integer> alIndex=new ArrayList<Integer>();
    int row=(180/angleSpan)+1;
    int col=360/angleSpan;
    for(int i=0;i<row;i++)
    {
        if(i>0&&i<row-1)
        {//中间行
            for(int j=-1;j<col;j++)
            {//中间行的两个相邻点与下一行的对应点构成三角形
                int k=i*col+j;
                alIndex.add(k+col);
                alIndex.add(k+1);
                alIndex.add(k);
            }
            for(int j=0;j<col+1;j++)
            {//中间行的两个相邻点与上一行的对应点构成三角形
                int k=i*col+j;
                alIndex.add(k-col);
                alIndex.add(k-1);
                alIndex.add(k);
            }
        }
    }
    iCount=alIndex.size();
    byte indices[]=new byte[alIndex.size()];
    for(int i=0;i<alIndex.size();i++)
    {
        indices[i]=alIndex.get(i).byteValue();
    }
    //创建三角形构造索引数据缓冲
    suo = ByteBuffer.allocateDirect(indices.length);
    suo.put(indices);
    suo.position(0);
}

public void drawSelf(GL10 gl)
{
    gl.glScalef(scaleX, scaleY, scaleZ);

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
```

```
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

        //为画笔指定顶点坐标数据
        gl.glvCHertexPointer
        (
                3,                      //每个顶点的坐标数量为 3
                GL10.GL_FIXED,          //设置顶点坐标值的类型为 GL_FIXED
                0,                      //连续顶点坐标数据之间的间隔
                ding                    //顶点坐标数据
        );

        //为画笔指定顶点着色数据
        gl.glColorPointer
        (
                4,
                GL10.GL_FIXED,
                0,
                se
        );
        //绘制图形，一共 icount/3 个三角形，iCount 个顶点
        gl.glDrawElements
        (
                GL10.GL_TRIANGLES,          //以三角形方式填充
                iCount,
                GL10.GL_UNSIGNED_BYTE,      //索引值的尺寸
                suo                         //索引值数据
        );
    }
}
```

执行后会在屏幕中显示一个自动缩放效果的椭圆，效果如图 20-2 所示。

▲图 20-2　自动缩放的执行效果

20.1.3　实现平移变换

由一个图形改变为另一个图形，在改变过程中，原图形上的所有的点都向同一个方向运动，并且运动相等的距离，这样的图形改变叫作图形的平移变换，简称平移。平移变换是把当前矩阵与一个表示移动物体的矩阵相乘，矩阵可以简单地如图 20-3 所示。

$$\begin{matrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{matrix}$$

▲图 20-3　平移矩阵图

在图 20-3 中所示的平移向量为（x,y,z）。在 OpenGL ES 中，我们可以使用方法 glTranslatex (int x，int y，int z)和 glTranslatef(float x，float y，float z)来实现平移变换效果，表示把当前矩阵与一个表示物体移动的矩阵相乘，3 个参数分别表示 3 个坐标上的位移值。这样通过平移变换就可以在场景中的不同位置绘制不同的物体，从而绘制出一个丰富多彩的 3D 场景。

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中实现平移变换效果的方法。

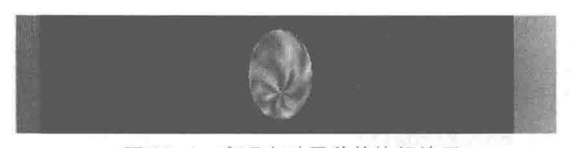
题目	目的	源码路径
实例 20-2	在 Android 手机屏幕中实现平移变换效果	\daima\20\pingCH

编写文件 ddd.java，在此定义方法 SceneRenderer()，通过新线程实现平移处理，主要实现代码如下所示。

```
private class SceneRenderer implements GLSurfaceView.Renderer
{
    pingCH ball=new pingCH(3);

    public SceneRenderer()
    {

        new Thread()
        {
            public void run()
            {

                while(true)
                {
                  ball.moveX+=gaiX;
                  if(ball.moveX>3)
                  {
                      gaiX=-0.1f;
                  }
                  if(ball.moveX<-3)
                  {
                      gaiX=0.1f;
                  }
                  try
                {
                    Thread.sleep(50);//休息 50ms 再重绘
                }
                catch(Exception e)
                {
                    e.printStackTrace();
                }
                }
            }
        }.start();
    }
    public void onDrawFrame(GL10 gl) {

        //清除颜色缓存
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        //设置当前矩阵为模式矩阵
        gl.glMatrixMode(GL10.GL_MODELVCHIEW);
        //设置当前矩阵为单位矩阵
        gl.glLoadIdentity();
        gl.glTranslatef(0, 0, -5);
        ball.drawSelf(gl);
    }
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        //设置视窗大小和位置
        gl.glvCHiewport(0, 0, width, height);
       //设置为投影矩阵
        gl.glMatrixMode(GL10.GL_PROJECTION);
        //设置为单位矩阵
        gl.glLoadIdentity();
        gl.glShadeModel(GL10.GL_SMOOTH);
        //获取透视投影比例
        float ratio = (float) width / height;
        //产生透视投影矩阵
        gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);
    }
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        //关闭抗抖动
        gl.glDisable(GL10.GL_DITHER);
        //设置特定 Hint 项目的模式，这里设置为使用快速模式
        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
        //设置屏幕背景色黑色 RGBA
```

```
            gl.glClearColor(0,0,0,0);
            //设置着色模型为平滑着色
            gl.glShadeModel(GL10.GL_SMOOTH);//GL10.GL_SMOOTH  GL10.GL_FLAT
            //启用深度测试
            gl.glEnable(GL10.GL_DEPTH_TEST);
        }
    }
}
```

本实例的其他实现文件和本书前面实例中的基本类似，在此不在介绍。执行后在屏幕中实现自动平移效果，如图 20-4 所示。



▲图 20-4　实现自动平移的执行效果

20.2 使用 Alpha 混合技术

无论是本书前面讲解的颜色绘制还是纹理绘制，它们都不是透明的。在很多真实的场景中，有非常多半透明的物体。要想在 OpenGL ES 中真实地再现半透明物体，这就需要 Alpha 混合技术来实现。从本节开始，将向读者详细介绍混合技术的基本知识。

20.2.1　基本知识

通过 Alpha 值在混合操作中可以控制新片元的颜色值与原有颜色值的合并权重。因此，通过 Alpha 混合可以创建半透明效果的片元。Alpha 颜色混合是诸如透明度、数字合成等技术的核心。

对于混合操作来说，最常见的是将 RGB 分量视为片元的颜色，而将 Alpha 分量视为不透明度。因此，透明或半透明表面的不透明度比不透明表面的低。例如，当透过绿色玻璃观察物体时，看到的颜色有几分玻璃的绿色，同时有几分物体的颜色。这两种颜色的比取决于玻璃的透射性质：如果照射在玻璃上的光有 80%透过（即不透明度为 20%），则看到的颜色是由 20%的玻璃颜色和 80%的物体颜色组合而成的。

在现实中有时会存在多个半透明面，例如在观察汽车时，汽车内部和视点之间有一片玻璃，如果透过两块车窗玻璃，可以看到汽车后面的物体。

（1）源因子和目标因子。

在混合过程中，分两步将输入片元（源）的颜色值同当前存储在帧缓存中的像素（目标）颜色值合并起来。首先，指定如何计算源因子和目标因子，这些因子是 RGBA 四元组，其分别与源和目标的 R、G、B、A 分量相乘：然后；将两个 RGBA 四元组中对应的分量相加。

在 OpenGL ES 系统中，通过调用方法 glBlendFunc()来选择源混合因子和目标混合因子，并指定两个混合因子，其中第 1 个参数为源 RGBA 的混合因子，第 2 个参数为目标 RGBA 的混合因子。

（2）处理方式。

在混合处理时，最常见的操作方式有如下 5 种。

- 均匀地混合两幅图像。

首先将源因子和目标因子分别设置为 GL_ONE 和 GL_ZERO，并绘制第 1 幅图像；然后将源因子设置为 GL_SRC_Alpha，目标因子设置为 GL_ONE_MINUS_SRC_ALPHA，并在绘制第 2 幅图像时设置 Alpha 的值为 0.5。

均匀地混合两幅图像是最常用的混合方式，如果要让第 1 幅图像占 75%，第 2 幅图像占 25%，可以按前面的方法绘制第 1 幅图像，然后在绘制第 2 幅图像使用 Alpha 的值为 0.25。

- 均匀地混合 3 幅图像。

将目标因子设置为 GL-ONE，将源因子设置为 GL_SEC_ALPHA，然后使用 Alpha 值 0.3333333 来绘制这些图像。这样每幅图像的亮度都只有原来的 1/3，如果图像之间重叠，将可以明显地观察到这一点。

- 逐渐加深图像。

假定编写绘图程序时，希望画笔能够逐渐地加深图像的颜色，使得每画一笔图像的颜色都将在原来的基础上加深一些。可将原混合因子和目标混合因子分别设置为 GL_SRC_ALPHA 和 GL_ONE_MINUS_SRC_ALPHA，并将画笔的 Alpha 值设置为 0.1。

- 模拟滤光器。

通过将源混合因子设置为 GL_DST_COLOR 或 GL_ONE_MJNUS_DST_COLOR，将目标混合因子设置为 GL_SRC_COLOR 或 GL_ONE_MINUS_SRC_COLOR，可以分别调整各个颜色分量。这样做相当于使用一个简单的滤光器。

例如，通过将红色分量乘以 0.8，绿色分量乘以 0.4，蓝色分量乘以 0.72，可以模拟通过这样的滤光器观察场景的情况：滤光器滤掉 20%的红光、60%的绿光和 28%的蓝光。

- 贴花法。

通过给图像中的片元指定不同的 Alpha 值，可以实现非矩阵光栅图像的效果。在大多数情况下会将透明片元的 Alpha 值设置为 0，每部透明片元的 Alpha 值设置为 1.0。例如可以绘制一个属性多边形，并应用树叶纹理。如果将 Alpha 值设置为 0，观察者将能够透过矩形纹理中不属于树的部分看到后面的东西。

20.2.2 实现简单混合

其实不管如何指定混合参数，总是需要启用混合功能，以便使其生效。启用方法是 gl.glEnable (GL_BLEND)，禁用混合的方法是 gl.glDisable(GL_BLEND)。

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中实现混合效果的方法。本实例有两种混合模式的组合，其中一个是参数 GL_ONE 和 GL_ONE_MINUS_DST_ALPHA 的混合效果；另一个是参数 GL_SRC_COLOR 和 GL_DST_ALPHA 的混合效果。

题目	目的	源码路径
实例 20-3	在屏幕中实现混合效果	\daima\20\hunheCH

本实例的实现流程如下所示。

（1）编写文件 ddd.java，在此实现场景绘制类。首先创建渲染器对象，并设置渲染器；然后通过场景绘制方法在在场景中绘制 4 个矩形。文件 Ddd.java 的主要实现代码如下所示。

```
private class SceneRenderer implements GLSurfaceView.Renderer
{
    final int one=65535;
    int diceng;
    int dingceng;
    ColorRect ju1;
    ColorRect ju2;
    TextureRect wen1;
    TextureRect wen2;

    public void onDrawFrame(GL10 gl) {
        //采用平滑着色
```

```
        gl.glShadeModel(GL10.GL_SMOOTH);

        //清除颜色缓存于深度缓存
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        //设置当前矩阵为模式矩阵
        gl.glMatrixMode(GL10.GL_MODELVCHIEW);
        //设置当前矩阵为单位矩阵
        gl.glLoadIdentity();

        //绘制底层纹理矩形
        gl.glPushMatrix();
        gl.glTranslatef(0, 0f, -2f);
        wen1.drawSelf(gl);
        gl.glPopMatrix();

        //绘制顶层纹理矩形
        gl.glPushMatrix();
        gl.glTranslatef(-0.7f, -0.3f, -1.9f);
        wen2.drawSelf(gl);
        gl.glPopMatrix();

        //绘制顶层颜色半透明矩形
        gl.glPushMatrix();
        gl.glTranslatef(0.7f, 0.4f, -1.8f);
        ju1.drawSelf(gl);
        gl.glPopMatrix();

        //绘制顶层颜色半透明矩形
        gl.glPushMatrix();
        gl.glTranslatef(-0.6f, 0.6f, -1.8f);
        ju2.drawSelf(gl);
        gl.glPopMatrix();
    }

    public void onSurfaceChanged(GL10 gl, int width, int height) {
        //设置窗口大小和位置
        gl.glvCHiewport(0, 0, width, height);
        //设置为投影矩阵
        gl.glMatrixMode(GL10.GL_PROJECTION);
        //设置为单位矩阵
        gl.glLoadIdentity();
        //计算透视投影比例
        float ratio = (float) width / height;
        //生成透视投影矩阵
        gl.glFrustumf(-ratio, ratio, -1, 1, 1, 100);
    }

    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        //关闭抗抖动
        gl.glDisable(GL10.GL_DITHER);
        //设置特定 Hint 项目的模式，这里设置为使用快速模式
        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
        //设置屏幕背景色黑色 RGBA
        gl.glClearColor(0,0,0,0);
        //启用深度测试
        gl.glEnable(GL10.GL_DEPTH_TEST);
        //打开混合
        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_SRC_COLOR, GL10.GL_DST_ALPHA);

        //纹理初始化
        diceng=initTexture(gl,R.drawable.base);
        dingceng=initTexture(gl,R.drawable.top);

        //创建矩形
        ju1=new ColorRect(one,0,0,one*3/4);
        ju2=new ColorRect(0,one,0,one/2);
        wen1=new TextureRect(diceng);
        wen2=new TextureRect(dingceng);
    }
```

```
    }

    //初始化纹理
    public int initTexture(GL10 gl,int drawableId)//textureId
    {
        //生成纹理 ID
        int[] textures = new int[1];
        gl.glGenTextures(1, textures, 0);
        int currTextureId=textures[0];
        gl.glBindTexture(GL10.GL_TEXTURE_2D, currTextureId);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_
NEAREST);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MAG_FILTER,GL10.GL_
LINEAR);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,GL10.GL_CLAMP_
TO_EDGE);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,GL10.GL_CLAMP_
TO_EDGE);

        InputStream is = this.getResources().openRawResource(drawableId);
        Bitmap bitmapTmp;
        try
        {
            bitmapTmp = BitmapFactory.decodeStream(is);
        }
        finally
        {
            try
            {
                is.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmapTmp, 0);
        bitmapTmp.recycle();

        return currTextureId;
    }

}
```

（2）编写文件 ColorRect.java，在此定义实现颜色矩形类 ColorRect，具体实现流程如下所示。

- 创建顶点坐标，并将顶点坐标数组放入缓冲区内。
- 为创建顶点着色数组，并将顶点着色数组放入缓冲区内。
- 定义绘制矩形方法，为画笔指定顶点坐标数据和顶点着色数据后，以三角形方式来绘制矩形。

文件 ColorRect.java 的主要实现代码如下所示。

```
    public ColorRect(int r,int g,int b,int alpha)
    {
        //初始化顶点坐标数据
        vCount=6;
        final int UNIT_SIZE=40000;
        int vertices[]=new int[]
        {
            -1*UNIT_SIZE,1*UNIT_SIZE,0,
            -1*UNIT_SIZE,-1*UNIT_SIZE,0,
            1*UNIT_SIZE,1*UNIT_SIZE,0,

            -1*UNIT_SIZE,-1*UNIT_SIZE,0,
            1*UNIT_SIZE,-1*UNIT_SIZE,0,
            1*UNIT_SIZE,1*UNIT_SIZE,0
        };

        //创建顶点坐标数据缓冲
        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
```

```
        vbb.order(ByteOrder.nativeOrder());
        ding = vbb.asIntBuffer();
        ding.put(vertices);
        ding.position(0);
        int colors[]=new int[]//顶点颜色值数组，每个顶点有 4 个色彩值
        {
                r,g,b,alpha,
                r,g,b,alpha,
                r,g,b,alpha,

                r,g,b,alpha,
                r,g,b,alpha,
                r,g,b,alpha,
        };
        //创建顶点着色数据缓冲
        ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length*4);
        cbb.order(ByteOrder.nativeOrder());
        se = cbb.asIntBuffer();
        se.put(colors);
        se.position(0);
    }
    public void drawSelf(GL10 gl)
    {
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

        //为画笔指定顶点坐标数据
        gl.glvCHertexPointer
        (
                3,
                GL10.GL_FIXED,
                0,
                ding
        );
        //为画笔指定顶点着色数据
        gl.glColorPointer
        (
                4,
                GL10.GL_FIXED,
                0,
                se
        );
        //绘制图形
        gl.glDrawArrays
        (
                GL10.GL_TRIANGLES,          //以三角形方式填充
                0,
                vCount
        );

        gl.glDisableClientState(GL10.GL_COLOR_ARRAY);//禁用顶点颜色数组
    }
}
```

本实例执行之后的效果如图 20-5 所示。

▲图 20-5　混合的执行效果

20.2.3 实现光晕和云层效果

通过混合可以方便地实现光晕及云层的效果。因为地球是有大气的，所以在太空中看，地球周围应该有光晕和云层，这样才会显得更加逼真。接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中实现光晕和云层效果的方法。

题目	目的	源码路径
实例 20-4	在屏幕中实现光晕和云层效果	\daima\20\yunCH

本实例的具体实现流程如下所示。

（1）编写文件 MyActivity.java，在此文件中定义类 MyActivity，然后声明场景界面的引用并设置场景为全屏，设置界面可触控和重写方法 onResume 和 onPause，具体实现代码如下所示。

```
public class MyActivity extends Activity {
  private ddd mGLSurfaceView;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      //设置全屏
      requestWindowFeature(Window.FEATURE_NO_TITLE);
      getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN ,
                 WindowManager.LayoutParams.FLAG_FULLSCREEN);

      mGLSurfaceView = new ddd(this);
      mGLSurfaceView.requestFocus();                               //获取焦点
      mGLSurfaceView.setFocusableInTouchMode(true);                //设为可触控

      setContentView(mGLSurfaceView);
  }

  @Override
  protected void onResume() {
      super.onResume();
      mGLSurfaceView.onResume();
  }
```

（2）编写文件 ddd.java，在此定义绘制场景类 ddd，具体实现流程如下所示。

- 设置渲染器，并重写触控方法完成转动地月系工作。
- 声明变量和绘制场景的方法，在绘制完地球后开启混合，设置源因子和目标因子以绘制云层。
- 关闭混合，在绘制完月球和星空后开启混合，再次设置源因子和目标因子开始绘制光晕。
- 设置场景，初始化光源、材质和纹理。

文件 ddd.java 的主要实现代码如下所示。

```
public void onDrawFrame(GL10 gl) {
//清除颜色缓存
gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
//设置当前矩阵为模式矩阵
gl.glMatrixMode(GL10.GL_MODELVCHIEW);
//设置当前矩阵为单位矩阵
gl.glLoadIdentity();

//将场景沿 z 轴推远
gl.glTranslatef(0, 0f, -3.6f);

gl.glEnable(GL10.GL_LIGHTING);
gl.glPushMatrix();
//设置高光聚光率
gl.glMaterialf(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS, 4.5f);
earth.drawSelf(gl);//绘制地球
```

```
//开启混合
gl.glEnable(GL10.GL_BLEND);
//设置源混合因子与目标混合因子
gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
earthCloud.drawSelf(gl);//绘制地球云
//关闭混合
gl.glDisable(GL10.GL_BLEND);

    //月球离地球 1.5f
    gl.glTranslatef(0, 0f, 1.5f);
    //设置高光聚光率
    gl.glMaterialf(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS, 0.5f);
    moon.drawSelf(gl);//绘制月球
    gl.glPopMatrix();
    gl.glDisable(GL10.GL_LIGHTING);

    //绘制星空
    gl.glPushMatrix();
    gl.glTranslatef(0, -20.0f, 0.0f);
    xiao.drawSelf(gl);
    da.drawSelf(gl);
    gl.glPopMatrix();

    //绘制光晕
    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_ONE, GL10.GL_ONE_MINUS_DST_ALPHA);
    halo.drawSelf(gl);
    gl.glDisable(GL10.GL_BLEND);

}
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    //关闭抗抖动
    gl.glDisable(GL10.GL_DITHER);
    //设置特定 Hint 项目的模式，这里设置为使用快速模式
    gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
    //设置屏幕背景色黑色 RGBA
    gl.glClearColor(0,0,0,0);
    //设置着色模型为平滑着色
    gl.glShadeModel(GL10.GL_SMOOTH);
    //启用深度测试
    gl.glEnable(GL10.GL_DEPTH_TEST);
    //设置为打开背面剪裁
    gl.glEnable(GL10.GL_CULL_FACE);

    gl.glEnable(GL10.GL_LIGHTING);
    initSunLight(gl);
    initMaterial(gl);

    //初始化纹理
    diwen=initTexture(gl,R.drawable.earth);
    yuewen=initTexture(gl,R.drawable.moon);
    haloTextureId=initTexture(gl,R.drawable.halo);
    yunwen=initTexture(gl,R.drawable.clouds);

    //初始化三个球
    earth=new yunCH1(6000,diwen);
    moon=new yunCH1(2000,yuewen);
    earthCloud=new yunCH1(6006,yunwen);

    //创建星空
    xiao=new yunCH2(0,0,1,0,750);
    da=new yunCH2(0,0,2,0,200);

    //创建地球光晕
    halo=new ttt(haloTextureId);

   //开启一个线程自动旋转地球与月球
    new Thread()
    {
       public void run()
```

```
            {
                while(true)
                {
                  xuan.earth.mAngleY+=2*TOUCH_SCALE_FACTOR;
                  xuan.moon.mAngleY+=2*TOUCH_SCALE_FACTOR;
                requestRender();//重绘画面
                try
                {
                    Thread.sleep(50);
                }
                catch(Exception e)
                {
                    e.printStackTrace();
                }
                }
            }
        }.start();

        new Thread()
        {//定时转动星空的线程
            public void run()
            {
                while(true)
                {
                   xiao.yAngle+=0.5;
                   if(xiao.yAngle>=360)
                   {
                       xiao.yAngle=0;
                   }
                   da.yAngle+=0.5;
                  if(da.yAngle>=360)
                  {
                      da.yAngle=0;
                  }
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
    }
}

//初始化太阳光源
private void initSunLight(GL10 gl)
{
    gl.glEnable(GL10.GL_LIGHT0);//打开 0 号灯

    //环境光设置
    float[] ambientParams={0.05f,0.05f,0.025f,1.0f};
    gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_AMBIENT, ambientParams,0);

    //散射光设置
    float[] diffuseParams={1f,1f,0.5f,1.0f};
    gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_DIFFUSE, diffuseParams,0);

    //反射光设置
    float[] specularParams={1f,1f,0.5f,1.0f};
    gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_SPECULAR, specularParams,0);

    //设定光源的位置
    float[] positionParamsGreen={-14.14f,20.28f,6f,0};
    gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_POSITION, positionParamsGreen,0);
}


//初始化材质
private void initMaterial(GL10 gl)
```

```
{//材质为白色时，什么颜色的光照在上面就将体现出什么颜色
    //环境光为白色材质
    float ambientMaterial[] = {0.6f, 0.6f, 0.6f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT, ambientMaterial,0);
    //散射光为白色材质
    float diffuseMaterial[] = {1.0f, 1.0f, 1.0f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE, diffuseMaterial,0);
    //高光材质为白色
    float specularMaterial[] = {1f, 1f, 1f, 1.0f};
    gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SPECULAR, specularMaterial,0);
}

//初始化纹理
public int initTexture(GL10 gl,int drawableId)
{
    //生成纹理 ID
    int[] textures = new int[1];
    gl.glGenTextures(1, textures, 0);
    int textureId=textures[0];
    gl.glBindTexture(GL10.GL_TEXTURE_2D, textureId);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_NEAREST);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MAG_FILTER,GL10.GL_LINEAR);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,GL10.GL_CLAMP_TO_EDGE);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,GL10.GL_CLAMP_TO_EDGE);

   InputStream is = this.getResources().openRawResource(drawableId);
   Bitmap bitmapTmp;
   try
   {
      bitmapTmp = BitmapFactory.decodeStream(is);
   }
   finally
   {
      try
      {
         is.close();
      }
      catch(IOException e)
      {
         e.printStackTrace();
      }
   }
   GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmapTmp, 0);
   bitmapTmp.recycle();

   return textureId;
}
```

到此为止，此实例的主要代码已经讲解完毕，执行后的效果如图 20-6 所示。

▲图 20-6　光晕和云层的执行效果

20.2.4 实现滤光器效果

滤光器是一种能限制光辐射通过，通常用来改变光谱的分布的器件，例如最常见的太阳镜，在烈日下带上墨镜可以减轻刺眼的感觉。通过滤光器可以使光的强度降低，并带上几分滤光器的颜色。接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中实现滤光器效果的方法。

题目	目的	源码路径
实例 20-5	在屏幕中实现滤光器效果	\daima\20\lvCH

本实例的实现流程如下所示。

（1）编写文件 ddd.java，在此实现场景类 ddd，具体实现流程如下所示。

- 声明摄像机坐标和目标点坐标，并在该类构造器内设置场景渲染器。
- 重写键按下方法，在每次按下按键时移动瞄准镜的位置。
- 编写场景的绘制方法，先设置摄像机，然后依次绘制场景中物体，在绘制瞄准镜前开启混合，设置源因子和目标因子，最后绘制瞄准镜。

文件 ddd.java 的主要实现代码如下所示。

```
public class ddd extends GLSurfaceView
{
    float sx=0;
    float sy=3;
    float sz=10;

    float mx=0;
    float my=0;
    float mz=0;
    private SceneRenderer mRenderer;
    public ddd(Context context) {
        super(context);
        // TODO Auto-generated constructor stub
        mRenderer = new SceneRenderer();
        setRenderer(mRenderer);
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
    }
    @Override
    public boolean onKeyDown(int keyCode, KeyEvent e)
    {
        switch(keyCode)
        {
        case 21://按下左键
            if(mRenderer.cover.xOffset>-2)
            {
                mRenderer.cover.xOffset-=0.2f;//每次左移 0.2f
            }
            break;
        case 22://按下右键
            if(mRenderer.cover.xOffset<2)
            {
                mRenderer.cover.xOffset+=0.2f;//每次左移 0.2f
            }
        }
        return true;
    }

    private class SceneRenderer implements GLSurfaceView.Renderer
    {
        int wenId;
        int diId;
        int jingId;
        lvCH wen;
        lvCHColor seli;
```

```
        ttt di;
        ttt cover;
        public void onDrawFrame(GL10 gl) {
           // TODO Auto-generated method stub
           //采用平滑着色
           gl.glShadeModel(GL10.GL_SMOOTH);
           //打开背面剪裁
           gl.glEnable(GL10.GL_CULL_FACE);
           //清除颜色缓存
           gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
           //设置为模式矩阵
           gl.glMatrixMode(GL10.GL_MODELVIEW);
           //设置为单位矩阵
           gl.glLoadIdentity();
          //设置 camera 位置
           GLU.gluLookAt
           (
                  gl,
                  sx,
                  sy,
                  sz,
                  mx,
                  my,
                  mz,
                  0,
                  1,
                  0
           );
           gl.glPushMatrix();
           gl.glTranslatef(1, 1, 0);
           wen.drawSelf(gl);
           gl.glPopMatrix();

           gl.glPushMatrix();
           gl.glTranslatef(-1, 0, 0);
           seli.drawSelf(gl);
           gl.glPopMatrix();

           gl.glPushMatrix();
           for(int i=0;i<10;i++)
           {
               for(int j=0;j<10;j++)
               {
                   gl.glPushMatrix();
                   gl.glTranslatef(-2.5f+i*0.5f, 0, -2.5f+j*0.5f);
                   di.drawSelf(gl);
                   gl.glPopMatrix();
               }
           }
           gl.glPopMatrix();

           gl.glPushMatrix();
           gl.glEnable(GL10.GL_BLEND);
           gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
           gl.glTranslatef(0, 1, 2);
           cover.drawSelf(gl);
           gl.glDisable(GL10.GL_BLEND);
           gl.glPopMatrix();
        }
        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
             //视窗大小及位置
           gl.glViewport(0, 0, width, height);
           //设置为投影矩阵
           gl.glMatrixMode(GL10.GL_PROJECTION);
           //设置为单位矩阵
           gl.glLoadIdentity();
           //计算透视投影的比例
           float ratio = (float) width / height;
```

```
        //计算产生透视的投影矩阵
        gl.glFrustumf(-ratio*0.5f, ratio*0.5f, -0.5f, 0.5f, 1, 100);
    }
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        // TODO Auto-generated method stub
         //关闭抗抖动
        gl.glDisable(GL10.GL_DITHER);
        //设置特定 Hint 模式为快速模式
        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
        //设置屏幕背景色
        gl.glClearColor(0,0,0,0);
        //设置为平滑着色
        gl.glShadeModel(GL10.GL_SMOOTH);
        //用深度测试
        gl.glEnable(GL10.GL_DEPTH_TEST);
        //打开背面剪裁
        gl.glEnable(GL10.GL_CULL_FACE);
        wenId=initTexture(gl,R.drawable.aaa);
        diId=initTexture(gl,R.drawable.ground);
        jingId=initTexture(gl,R.drawable.cover);
        wen=new lvCH(wenId,2,1,1);
        seli=new lvCHColor(1,2,2);
        di=new ttt//创建地板
        (
               1,0,1,diId,
               new float[]
                 {
                       0,0,0,1,1,1,
                       1,1,1,0,0,0
                 }
        );
        cover=new ttt//创建瞄准镜
        (
               1,1,0,jingId,
               new float[]
                 {
                       0,0,0,1,1,1,
                       1,1,1,0,0,0
                 }
        );

    }
}
```

（2）编写文件 lvCHColor.java，在此定义绘制着色长方体的类 lvCHColor，具体实现流程如下所示。

- 创建顶点数组，将顶点数组存入顶点缓冲内，并创建顶点颜色数组。因为各个顶点的 R、G、B、A 是通过随机数产生的，所以每次的运行结果可能有所不同。
- 编写绘制图形的方法，首先允许使用顶点数组、颜色数组，并为画笔指定顶点数组和颜色数组。
- 以顶点法中的三角形填充方式绘制图形。

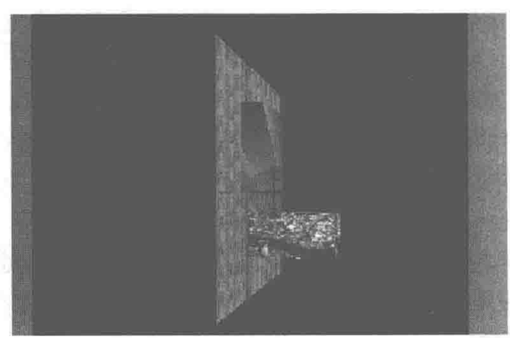
文件 lvCHColor.java 的主要实现代码如下所示。

```
public lvCHColor(float scale,float length,float width)
{
    this.scale=scale;
    vCount=36;
    float TABLE_UNIT_SIZE=0.5f;
    float TABLE_UNIT_HIGHT=0.5f;
    float[] verteices=
    {

           //顶面
           -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
```

```
            -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,

            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            //后面
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,

            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            //前面
            -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,

            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            //下面
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,

            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            //左面
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,

            -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            -TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            //右面
            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,-TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width,
            TABLE_UNIT_SIZE*length,-TABLE_UNIT_HIGHT*scale,TABLE_UNIT_SIZE*width
    };

    ByteBuffer vbb=ByteBuffer.allocateDirect(verteices.length*4);
    vbb.order(ByteOrder.nativeOrder());
    ding=vbb.asFloatBuffer();
    ding.put(verteices);
    ding.position(0);

    float[] colors=new float[vCount*4];
    for(int i=0;i<vCount;i++)
    {
        colors[i*4]=(float)Math.random();
        colors[i*4+1]=(float)Math.random();
        colors[i*4+2]=(float)Math.random();
        colors[i*4+3]=(float)Math.random();
    }
    ByteBuffer tbb=ByteBuffer.allocateDirect(colors.length*4);
    tbb.order(ByteOrder.nativeOrder());
    wen=tbb.asFloatBuffer();
    wen.put(colors);
    wen.position(0);
}
public void drawSelf(GL10 gl)
{
```

```
        gl.glRotatef(x, 1, 0, 0);
        gl.glRotatef(y, 0, 1, 0);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
        //为画笔指定顶点坐标数据
        gl.glVertexPointer
        (
                3,
                GL10.GL_FLOAT,
                0,
                ding
        );
        //为画笔指定顶点着色数据
        gl.glColorPointer
        (
                4,
                GL10.GL_FLOAT,
                0,
                wen
        );
        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, vCount);
        gl.glDisableClientState(GL10.GL_COLOR_ARRAY);
    }
}
```

执行后的效果如图 20-7 所示。



▲图 20-7　滤光器的执行效果

第21章　OpenGL ES进阶

经过本书前面4章内容的学习，我们了解了OpenGL ES技术的基本知识。其实OpenGL ES技术的功能十分强大，除了前面4章中介绍的功能外，还可以实现更加高级的功能。在本章的内容中，将详细讲解在Android系统中使用OpenGL ES技术实现高级功能的知识，为读者进入本书后面知识的学习打下基础。

21.1　实现摄像机和雾特效功能

摄像机和雾特效是三维世界中的常见效果，在OpenGL ES也可以实现摄像机和雾特效的功能。在本节的内容中，将向读者详细介绍在Android系统中实现摄像机和雾特效效果的基本知识。

21.1.1　摄像机基础

摄像机可将三维空间中的场景呈现在二维显示屏幕上，在3D游戏中玩家所看到的场景就是玩家通过摄像机观察到的游戏场景。摄像机在三维世界中至关重要，没有正确的设置，摄像机将会在屏幕上呈现错误的场景，甚至可能会出现黑屏的。

在没有摄像机的情况下，摄像机的默认位置是在原点（屏幕中心处），方向沿*z*轴负方向（沿屏幕向里）。但在很多实际应用中都需要根据程序运行情况来修改摄像机的位置、朝向和Up方向。这3个概念的具体说明如下所示。

- 摄像机的位置：是摄像机的*x*、*y*、*z*轴的坐标，也就是观察者眼睛的位置。在默认情况下，摄像机的位置是坐标原点。
- 摄像机的朝向：是观察者眼球目光的方向。在默认情况下，摄像机的朝向为沿*z*轴负方向。
- 摄像机的Up方向：是观察者头顶法线的指向。在默认情况下，摄像机的Up方向为沿*y*轴正方向。

上述摄像机的位置、朝向、Up方向可以有很多种组合，例如同样的位置可以有不同的朝向、Up方向，这与现实世界中人观察世界的情况非常相似。

为了获得场景中的某个想要的视图，我们开发人员可以把摄像机从默认位置移动，并让其指向特定的方向。在OpenGL ES系统中，可以使用GLU类的gluLookAt()方法来设置摄像机，此方法有10个参数，其语法格式如下所示。

```
gluLookAt ({arg0,argi,arg2,arg3,arg4,arg5,arg6,arg7,arg8,arg9);
```

各个参数的具体说明如下所示。

- arg0：表示画笔
- argl～arg3：依次表示摄像机位置的*x*、*r*、*z*坐标。
- arg4～arg6：依次表示摄像机朝向上某一指定点（在下面称之为目标点）的*x*、*y*、*z*坐标，

该指定点由开发人员自定。

- arg7～arg9：依次表示 Up 方向向量的 *x*、*y*、*z* 分量。

21.1.2 雾特效基础

雾特效是指使远处的物体看上去逐渐变得模糊。在自然界中，雾是用来描述自然界中的大气现象的。在三维世界中，“雾”用来描述一些类似的大气效果。雾可以用于模拟模糊、薄雾、烟或者污染。雾在其本质上是一种视觉模拟应用，用于模拟具有有限可视性的场合。

在三维世界中，有时候由于过于清晰和锐利，反而显得不太逼真。我们通过抗锯齿处理使物体的边缘显得更为平滑，增加了逼真感。另外，还可以通过添加雾特效，使三维世界变得更加逼真。计算机图像很多情况下轮廓过于鲜明，显得不够真实。在 3D 应用开发时可以加入雾效果，将物体融入背景中，使整个图像更为自然。

在开发 3D 应用时，通过加入雾效果，可以将物体融入背景中，使整个图像更为自然。当开启雾特效之后距离摄像机较远的物体开始融入雾的颜色中。在雾特效中还可以控制雾的浓度，它决定了物体随着距离的增加而融入雾颜色的速度。由于雾是在执行了矩阵变换、光照和纹理之后才应用的，因此它对经过变换、带光照和经过纹理贴图的物体产生影响。雾可以提高性能，因为它可以选择不绘制那些因为雾的影响而不可见的物体。雾的应用广泛，可以应用于所有类型的几何图元（包括点和直线）。

21.1.3 实现雾特效和摄像机效果

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中实现雾特效和摄像机效果的方法。

题目	目的	源码路径
实例 21-1	在 Android 手机屏幕中实现雾特效和摄像机效果	\daima\21\sheCH

本实例的实现流程如下所示。

（1）编写文件 jinzita.java，在此实现了一个金字塔场景的绘制类。首先声明数据缓冲，用于导入顶点坐标数据和纹理坐标数据；然后设定四面体顶点坐标数据来构造四面体，并分别设定四面体各个顶点对应的着色数据；最后设定四面体各个顶点对应的纹理坐标数据。文件 jinzita.java 的主要实现代码如下所示。

```
//金字塔类
public class jinzita {
    final float UNIT_SIZE=0.5f;
    private FloatBuffer  ding;
    private FloatBuffer  se;
    private FloatBuffer wen;
    int vCount=0;
    float yAngle;
    int x;
    int y;
    int wenID;
    public jinzita(int x,int y,float scale,float yAngle,int wenID)
    {
        this.x=x;
        this.y=y;
        this.yAngle=yAngle;
        this.wenID=wenID;

        //初始化顶点坐标数据
        vCount=12;//每个金字塔 4 个三角形面，共有 12 个顶点
```

```
    float vertices[]=new float[]
    {
        0,2*scale*UNIT_SIZE,0,
        UNIT_SIZE*scale,0,UNIT_SIZE*scale,
        UNIT_SIZE*scale,0,-UNIT_SIZE*scale,

        0,2*scale*UNIT_SIZE,0,
        UNIT_SIZE*scale,0,-UNIT_SIZE*scale,
        -UNIT_SIZE*scale,0,-UNIT_SIZE*scale,

        0,2*scale*UNIT_SIZE,0,
        -UNIT_SIZE*scale,0,-UNIT_SIZE*scale,
        -UNIT_SIZE*scale,0,UNIT_SIZE*scale,

        0,2*scale*UNIT_SIZE,0,
        -UNIT_SIZE*scale,0,UNIT_SIZE*scale,
        UNIT_SIZE*scale,0,UNIT_SIZE*scale,
    };

    //创建顶点坐标数据缓冲
    ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
    vbb.order(ByteOrder.nativeOrder());
    ding = vbb.asFloatBuffer();
    ding.put(vertices);
    ding.position(0);

    //初始化顶点的法向量数据
    float normals[]=new float[]
    {
        0.89443f,0.44721f,0f,
        0.89443f,0.44721f,0f,
        0.89443f,0.44721f,0f,

        0,0.44721f,-0.89443f,
        0,0.44721f,-0.89443f,
        0,0.44721f,-0.89443f,

        -0.89443f,0.44721f,0f,
        -0.89443f,0.44721f,0f,
        -0.89443f,0.44721f,0f,

        0,0.44721f,0.89443f,
        0,0.44721f,0.89443f,
        0,0.44721f,0.89443f,
    };

    ByteBuffer nbb = ByteBuffer.allocateDirect(normals.length*4);
    nbb.order(ByteOrder.nativeOrder());
    se = nbb.asFloatBuffer();
    se.put(normals);
    se.position(0);

    //初始化纹理坐标数据
    float[] texST=
    {
        0.5f,0.0f,0,1,1,1,
        0.5f,0.0f,0,1,1,1,
        0.5f,0.0f,0,1,1,1,
        0.5f,0.0f,0,1,1,1,
    };
    ByteBuffer tbb = ByteBuffer.allocateDirect(texST.length*4);
    tbb.order(ByteOrder.nativeOrder());
    wen = tbb.asFloatBuffer();
    wen.put(texST);
    wen.position(0);
}

public void drawSelf(GL10 gl)
{
```

```
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);

        gl.glPushMatrix();
        gl.glTranslatef(x*UNIT_SIZE, 0, 0);//平移 x
        gl.glTranslatef(0, 0, y*UNIT_SIZE);//平移 y
        gl.glRotatef(yAngle, 0, 1, 0);//绕 y 旋转

        //为画笔指定顶点坐标数据
        gl.glVertexPointer
        (
                3,              //每个顶点的坐标数量为 3
                GL10.GL_FLOAT,    //顶点坐标值类型为 GL_FIXED
                0,              //连续顶点坐标数据之间的间隔
                ding    //顶点坐标数据
        );

        //为画笔指定顶点法向量数据
        gl.glNormalPointer(GL10.GL_FLOAT, 0, se);

        //打开纹理
        gl.glEnable(GL10.GL_TEXTURE_2D);
        //使用纹理 s、t 坐标缓冲
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        //指定纹理 s、t 坐标缓冲
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, wen);
        //绑定当前纹理
        gl.glBindTexture(GL10.GL_TEXTURE_2D, wenID);
        //绘制图形
        gl.glDrawArrays
        (
                GL10.GL_TRIANGLES,
                0,
                vCount
        );
        gl.glPopMatrix();//恢复变换矩阵现场
    }
}
```

（2）编写文件 ddd.java，在此定义了一个 ddd 类，主要功能是在场景中实现太阳东升西落的效果。此文件的具体实现流程如下所示。

- 定义类 SceneRenderer，通过线程 Thread()实现旋转阳光效果，对应代码如下所示。

```
class ddd extends GLSurfaceView {

    static int qiangA;
    static int qiangB;
    static int qiangc;
    static int sha;

    private SceneRenderer mRenderer;//渲染器

    public ddd(Context context) {
        super(context);
        mRenderer = new SceneRenderer();
        setRenderer(mRenderer);
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
    }

    private class SceneRenderer implements GLSurfaceView.Renderer
    {
        jinzita[] pArray;
        shamo shamo;
        xingkong xiao;
        xingkong da;

        double lightAngle=120.0;
```

```
        public SceneRenderer()
        {
                new Thread()
                {//定时旋转阳光
                    public void run()
                    {
                        while(true)
                        {
                            lightAngle+=0.5;
                            if(lightAngle>=360)
                            {
                                lightAngle=0;
                            }
                            try {
                               Thread.sleep(10);
                           } catch (InterruptedException e) {
                               e.printStackTrace();
                           }
                        }
                    }
                }.start();
        }
```

● 定义方法 onDrawFrame()，设定光源后分别实现金字塔、沙漠和星空场景，对应代码如下所示。

```
        public void onDrawFrame(GL10 gl) {
            //使用平滑着色
            gl.glShadeModel(GL10.GL_SMOOTH);
            //Sun 光源的位置
            float lxSun=(float)(1*Math.cos(Math.toRadians(lightAngle)));
            float lySun=(float)(1*Math.sin(Math.toRadians(lightAngle)));
            float[] positionParamsGreen={lxSun,lySun,0.6f,0};
            gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_POSITION, positionParamsGreen,0);

            //清除颜色缓存
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
            //设置为模式矩阵
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            //设置为单位矩阵
            gl.glLoadIdentity();

            //绘制金字塔
            for(jinzita tp:pArray)
            {
               tp.drawSelf(gl);
            }
            //绘制沙漠
            shamo.drawSelf(gl);
            //绘制星空
            xiao.drawSelf(gl);
            da.drawSelf(gl);
        }
```

● 定义方法 onSurfaceChanged()，当窗口的大小发生改变时调用 onSurfaceChanged()方法。不过不管窗口的大小是否已经改变，它在程序开始时至少运行一次，所以在该方法中需要设置 OpenGL 场景的大小，对应代码如下所示。

```
public void onSurfaceChanged(GL10 gl, int width, int height) {
    //设置视窗大小及位置
    gl.glViewport(0, 0, width, height);
    //设置为投影矩阵
    gl.glMatrixMode(GL10.GL_PROJECTION);
    //设置为单位矩阵
    gl.glLoadIdentity();
    //计算透视投影比例
   float ratio = (float) width / height;
```

```
    //计算产生透视投影矩阵
    gl.glFrustumf(-ratio, ratio, -0.5f, 1.5f, 1, 100);

    //设置 camera 位置
    GLU.gluLookAt
    (
          gl,
          -1.0f,
          0.6f,
          3.0f,
          0,
          0.2f,
          0,
          0,
          1,
          0
    );
}
```

● 定义方法 onSurfaceCreate()，实现不同场景的光照、材质和雾化效果，对应代码如下所示。

```
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    //关闭抗抖动
    gl.glDisable(GL10.GL_DITHER);
    //设置 Hint 项目为快速模式
    gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
    //设置屏幕背景色黑色 RGBA
    gl.glClearColor(0,0,0,0);
    //打开深度测试
    gl.glEnable(GL10.GL_DEPTH_TEST);
    //打开背面剪裁
    gl.glEnable(GL10.GL_CULL_FACE);

    //纹理初始化
    qiangA=initTexture(gl,R.drawable.walla);
    qiangB=initTexture(gl,R.drawable.wallb);
    qiangc=initTexture(gl,R.drawable.wallc);
    sha=initTexture(gl,R.drawable.desert);

    //创建金字塔
    pArray=new jinzita[]
    {
          new jinzita(-2,-2,2.0f,30,qiangA),
          new jinzita(3,-7,2.0f,0,qiangB),
          new jinzita(6,-2,2.0f,0,qiangc),
    };
    //创建沙漠
    shamo=new shamo(-20,-20,4,0,sha,40,40);
    //创建星空
    xiao=new xingkong(0,0,1,0,250);
    da=new xingkong(0,0,2,0,50);

    gl.glEnable(GL10.GL_LIGHTING);
    initSunLight(gl);
    dx(gl);//材质初始化

    gl.glEnable(GL10.GL_FOG);
    initFog(gl);

    new Thread()
    {//定时转动星空
        public void run()
        {
            while(true)
            {
               xiao.yAngle+=0.5;
               if(xiao.yAngle>=360)
               {
                   xiao.yAngle=0;
```

```
                }
                da.yAngle+=0.5;
               if(da.yAngle>=360)
               {
                  da.yAngle=0;
               }
                 try {
                     Thread.sleep(100);
                 } catch (InterruptedException e) {
                    e.printStackTrace();
                 }
             }
         }
      }.start();
   }
}
```

● 定义方法 dx()，实现太阳东升西落的效果，对应代码如下所示。

```
private void dx(GL10 gl)
{
   //白色材质环境光
   float ambientMaterial[] = {0.4f, 0.4f, 0.4f, 1.0f};
   gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT, ambientMaterial,0);
   //白色材质散射光
   float diffuseMaterial[] = {0.8f, 0.8f, 0.8f, 1.0f};
   gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE, diffuseMaterial,0);
   //白色高光材质
   float specularMaterial[] = {0.6f, 0.6f, 0.6f, 1.0f};
   gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SPECULAR, specularMaterial,0);
   //高光反射区域
   float shininessMaterial[] = {1.5f};
   gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS, shininessMaterial,0);
}
```

● 定义方法 initFog()来初始化雾效果，分别设置雾的颜色、浓度、开始距离和结束距离等参数，对应代码如下所示。

```
//初始化雾
public void initFog(GL10 gl)
{
   float[] fogColor={1,0.91765f,0.66667f,0};
   gl.glFogfv(GL10.GL_FOG_COLOR, fogColor, 0);
   gl.glFogx(GL10.GL_FOG_MODE, GL10.GL_EXP2);
   gl.glFogf(GL10.GL_FOG_DENSITY, 1);
   gl.glFogf(GL10.GL_FOG_START, 0.5f);
   gl.glFogf(GL10.GL_FOG_END, 100.0f);
}

//纹理初始化
public int initTexture(GL10 gl,int drawableId)
{
   //生成纹理 ID
   int[] textures = new int[1];
   gl.glGenTextures(1, textures, 0);
   int currTextureId=textures[0];
   gl.glBindTexture(GL10.GL_TEXTURE_2D, currTextureId);
   gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_
NEAREST);
   gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MAG_FILTER,GL10.GL_LINEAR);
   gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,GL10.GL_CLAMP_TO_
EDGE);
   gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,GL10.GL_CLAMP_TO_
EDGE);

   InputStream is = this.getResources().openRawResource(drawableId);
   Bitmap bitmapTmp;
   try
```

```
        {
            bitmapTmp = BitmapFactory.decodeStream(is);
        }
        finally
        {
            try
            {
                is.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmapTmp, 0);
        bitmapTmp.recycle();

        return currTextureId;
    }
```

（3）编写文件 xingkong.java，在此定义一个表示星空天球的类 xingkong。原理是在地球的外围包裹一个半径远大于地球，并且在其上面绘制有不同大小的白色点的球作为星空天球，并用线程控制该天球逆时针缓慢地旋转，这样做的目的是在场景中观察到比较真实的效果。文件 xingkong.java 的主要实现代码如下所示。

```
//星空天球的类
public class xingkong {
    final float UNIT_SIZE=6.0f;
    private FloatBuffer   ding;
    private IntBuffer   se;
    int xing=0;
    float jiao;
    int x;
    int z;
    float scale;
    public xingkong(int x,int z,float scale,float jiao,int xing)
    {
        this.x=x;
        this.z=z;
        this.jiao=jiao;
        this.scale=scale;
        this.xing=xing;

        //初始化顶点坐标数据
        float vertices[]=new float[xing*3];
        for(int i=0;i<xing;i++)//随机产生位于球面上的点
        {
            //随机产生每个星星的 x、y、z 坐标
            double angleTempJD=Math.PI*2*Math.random();
            double angleTempWD=Math.PI/2*Math.random();
            vertices[i*3]=(float)(UNIT_SIZE*Math.cos(angleTempWD)*Math.sin(angleTempJD));
//通过球公式计算球面上对应经纬度上的点的坐标
            vertices[i*3+1]=(float)(UNIT_SIZE*Math.sin(angleTempWD));
            vertices[i*3+2]=(float)(UNIT_SIZE*Math.cos(angleTempWD)*Math.cos(angleTempJD));
        }

        //缓冲顶点坐标
        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
        vbb.order(ByteOrder.nativeOrder());
        ding = vbb.asFloatBuffer();
        ding.put(vertices);
        ding.position(0);

       //初始化顶点着色数据
        final int one = 65535;
        int colors[]=new int[xing*4];//顶点颜色值数组，每个顶点 4 个色彩值 RGBA
        for(int i=0;i<xing;i++)//将所有点的颜色设置成白色
```

```
        {
            colors[i*4]=one;
            colors[i*4+1]=one;
            colors[i*4+2]=one;
            colors[i*4+3]=0;
        }

        //创建顶点着色数据缓冲
        ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length*4);
        cbb.order(ByteOrder.nativeOrder());
        se = cbb.asIntBuffer();
        se.put(colors);
        se.position(0);
    }

    public void drawSelf(GL10 gl)
    {
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

        gl.glDisable(GL10.GL_LIGHTING);
        gl.glPointSize(scale);
        gl.glPushMatrix();
        gl.glTranslatef(x*UNIT_SIZE, 0, 0);
        gl.glTranslatef(0, 0, z*UNIT_SIZE);
        gl.glRotatef(jiao, 0, 1, 0);

        //为画笔指定顶点坐标数据
        gl.glVertexPointer
        (
                3,
                GL10.GL_FLOAT,
                0,
                ding    //顶点坐标数据
        );

        //为画笔指定顶点着色数据
        gl.glColorPointer
        (
                4,
                GL10.GL_FIXED,
                0,
                se
        );

        //绘制点
        gl.glDrawArrays
        (
                GL10.GL_POINTS,
                0,
                xing
        );

        gl.glPopMatrix();
        gl.glPointSize(1);
        gl.glEnable(GL10.GL_LIGHTING);
    }
}
```

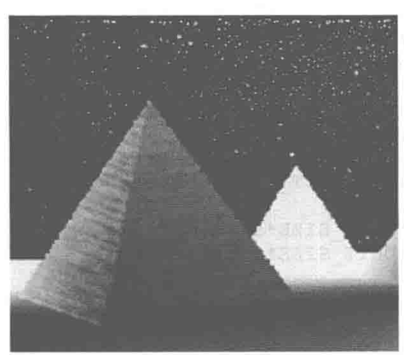
（4）编写文件 shamo.java，在此定义一个表示沙漠的类 shamo，其实现原理和实现金字塔的基本类似，主要实现代码如下所示。

```
    public shamo(int xOffset,int zOffset,float scale,float yAngle,int texId,int width,int
height)
    {
        this.xOffset=xOffset;
        this.zOffset=zOffset;
        this.yAngle=yAngle;
```

```
this.texId=texId;
this.width=width;
this.height=height;

vCount=width*height*6;                          //每个沙漠块 6 个顶点

float vertices[]=new float[vCount*3];
int k=0;
for(int i=0;i<width;i++)
    for(int j=0;j<height;j++)
    {
        vertices[k++]=i*UNIT_SIZE*scale;
        vertices[k++]=0;
        vertices[k++]=j*UNIT_SIZE*scale;
        vertices[k++]=i*UNIT_SIZE*scale;
        vertices[k++]=0;
        vertices[k++]=(j+1)*UNIT_SIZE*scale;
        vertices[k++]=(i+1)*UNIT_SIZE*scale;
        vertices[k++]=0;
        vertices[k++]=(j+1)*UNIT_SIZE*scale;

        vertices[k++]=(i+1)*UNIT_SIZE*scale;
        vertices[k++]=0;
        vertices[k++]=(j+1)*UNIT_SIZE*scale;
        vertices[k++]=(i+1)*UNIT_SIZE*scale;
        vertices[k++]=0;
        vertices[k++]=j*UNIT_SIZE*scale;
        vertices[k++]=i*UNIT_SIZE*scale;
        vertices[k++]=0;
        vertices[k++]=j*UNIT_SIZE*scale;
    };

//创建顶点坐标数据缓冲
//vertices.length*4 是因为一个 Float 四个字节
ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
vbb.order(ByteOrder.nativeOrder());              //设置字节顺序
mVertexBuffer = vbb.asFloatBuffer();
mVertexBuffer.put(vertices);
mVertexBuffer.position(0);

float normals[]=new float[vCount*3];
for(int i=0;i<vCount;i++)
{
    normals[i*3]=0;
    normals[i*3+1]=1;
    normals[i*3+2]=0;
}
ByteBuffer nbb = ByteBuffer.allocateDirect(normals.length*4);
nbb.order(ByteOrder.nativeOrder());
mNormalBuffer = nbb.asFloatBuffer();
mNormalBuffer.put(normals);
mNormalBuffer.position(0);

//实现纹理坐标数据初始化
float[] texST=new float[vCount*2];
for(int i=0;i<vCount*2/12;i++)
{
    texST[i*12]=0;
    texST[i*12+1]=0;

    texST[i*12+2]=0;
    texST[i*12+3]=1;

    texST[i*12+4]=1;
    texST[i*12+5]=1;

    texST[i*12+6]=1;
    texST[i*12+7]=1;
```

```
        texST[i*12+8]=1;
        texST[i*12+9]=0;

        texST[i*12+10]=0;
        texST[i*12+11]=0;
    };
    ByteBuffer tbb = ByteBuffer.allocateDirect (texST.length*4);
    tbb.order(ByteOrder.nativeOrder());
    mTextureBuffer = tbb.asFloatBuffer();
    mTextureBuffer.put(texST);
    mTextureBuffer.position(0);
}
```

到此为止，整个实例的主要实现代码介绍完毕，执行后将显示一个美轮美奂的、用摄像机和雾特效实现的场景效果，如图 21-1 所示。



▲图 21-1　摄像机和雾特效的执行效果

21.2 粒子系统

粒子系统也是三维世界中的最常见效果之一，在 OpenGL ES 也可以实现粒子系统特效的效果。在本节的内容中，将向读者详细介绍在 Android 系统中实现例子系统特效效果的基本知识。

21.2.1　粒子系统基础

粒子系统表示三维计算机图形学中模拟一些特定的模糊现象的技术，而这些现象是用其他传统的渲染技术难以实现的真实感的 Game Physics。经常使用粒子系统模拟的现象有火、爆炸、烟、水流、火花、落叶、云、雾、雪、尘、流星尾迹或者像发光轨迹这样的抽象视觉效果等。

实现粒子系统效果的方法与实现星星效果有相似之处，也是先创建一个类，这此类中包含了要创建原型的各类属性，然后再在 Renderer 中将各类属性赋予相应的值。实现粒子系统效果，先用一个循环初始化所有的 particles，然后在 onDrawFrame 中循环出每一个 particle，最后判断运行一段时间的 particle 是否还为激活状态，若为 false 则再初始化一次。

21.2.2　实现粒子系统效果

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中实现粒子系统效果的方法。

题目	目的	源码路径
实例 21-2	在 Android 手机屏幕中实现粒子系统效果	\daima\21\liziCH

本实例的实现流程如下所示。

（1）在布局文件 main.xml 中插入一个 TextView 控件。

（2）定义类 liziCH 来表示“点”，在里面定义了各个点的坐标变量，具体实现代码如下所示。

```
public class liziCH
{
      boolean active;
      float life;
      float fade;
      float r;
      float g;
      float b;
      float x;
      float y;
      float z;
      float xi;
      float yi;
      float zi;
      float xg;
      float yg;
      float zg;
}
```

（3）为了更好地操作和控制微粒，在文件 example152.java 中特意加入如下变量。

```
public final static int MAX_PARTICLES =1000;
boolean rainbow=true;
Random random = new Random();
float slowdown=0.5f;          /* 减速粒子*/
float xspeed=1;               /* x 方向的速度*/
float yspeed=3;               /* y 方向的速度*/
float zoom=-30.0f;            /* 沿 z 轴缩放*/
int loop;                     /* 循环变量*/
int col=0;                    /* 当前的颜色*/
int delay;                    /* 延迟彩虹效果*/
```

（4）定义数组 colors 用于存储 12 种不同的颜色，具体实现代码如下所示。

```
static float colors[][]=
{
   {1.0f,  0.5f,  0.5f},
   {1.0f,  0.75f, 0.5f},
   {1.0f,  1.0f,  0.5f},
   {0.75f, 1.0f,  0.5f},
   {0.5f,  1.0f,  0.5f},
   {0.5f,  1.0f,  0.75f},
   {0.5f,  1.0f,  1.0f},
   {0.5f,  0.75f, 1.0f},
   {0.5f,  0.5f,  1.0f},
   {0.75f, 0.5f,  1.0f},
   {1.0f,  0.5f,  1.0f},
   {1.0f,  0.5f,  0.75f}
};
```

（5）装载纹理贴图实现初始化处理，具体实现代码如下所示。

```
public void ResetParticle(int num, int color, float xDir, float yDir, float zDir)
{
    particle tmp = new particle();
    tmp.active=true;
    tmp.life=1.0f;
    tmp.fade=(float)(rand()%100)/1000.0f+0.003f;
    tmp.r=colors[color][0];
    tmp.g=colors[color][1];
    tmp.b=colors[color][2];
    tmp.x=0.0f;
    tmp.y=0.0f;
    tmp.z=0.0f;
    tmp.xi=xDir;
    tmp.yi=yDir;
```

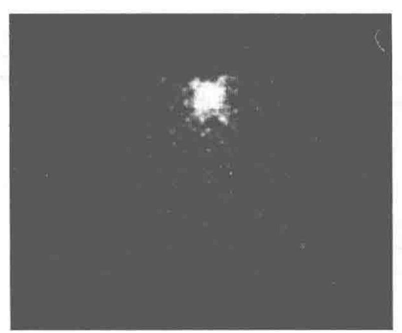
```
    tmp.zi=zDir;
    tmp.xg=0.0f;
    tmp.yg=-0.5f;
    tmp.zg=0.0f;
    particles[num] = tmp;
    return;
}
```

（6）给粒子分配一种颜色，通过方法 onDrawFrame()实现对粒子的绘制处理，具体实现代码如下所示。

```
public void onDrawFrame(GL10 gl)
{
    FloatBuffer vertices = FloatBuffer.wrap(new float[12]);
    FloatBuffer texcoords = FloatBuffer.wrap(new float[8]);
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertices);
    gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, texcoords);
    gl.glLoadIdentity();
    for (loop = 0; loop < MAX_PARTICLES; loop++)
    {
        if (particles[loop].active)
        {
            float x = particles[loop].x;
            float y = particles[loop].y;
            float z = particles[loop].z + zoom;
             gl.glColor4f(particles[loop].r,particles[loop].g,particles[loop].b,particles
[loop].life);
            texcoords.clear();
            vertices.clear();
            texcoords.put(1.0f);
            texcoords.put(1.0f);
            vertices.put(x + 0.5f);
            vertices.put(y + 0.5f);
            vertices.put(z);
            texcoords.put(0.0f);
            texcoords.put(1.0f);
            vertices.put(x - 0.5f);
            vertices.put(y + 0.5f);
            vertices.put(z);
            texcoords.put(1.0f);
            texcoords.put(0.0f);
            vertices.put(x + 0.5f);
            vertices.put(y - 0.5f);
            vertices.put(z);
            texcoords.put(0.0f);
            texcoords.put(0.0f);
            vertices.put(x - 0.5f);
            vertices.put(y - 0.5f);
            vertices.put(z);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4);
            particles[loop].x += particles[loop].xi / (slowdown * 1000);
            particles[loop].y += particles[loop].yi / (slowdown * 1000);
            particles[loop].z += particles[loop].zi / (slowdown * 1000);
            particles[loop].xi += particles[loop].xg;
            particles[loop].yi += particles[loop].yg;
            particles[loop].zi += particles[loop].zg;
            particles[loop].life -= particles[loop].fade;
            if (particles[loop].life < 0.0f)
            {
                float xi, yi, zi;
                xi = xspeed + (float) ((rand() % 60) - 32.0f);
                yi = yspeed + (float) ((rand() % 60) - 30.0f);
                zi = (float) ((rand() % 60) - 30.0f);
                ResetParticle(loop, col, xi, yi, zi);
            }
```

```
        }
    }
    gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glFinish();
}
```

执行后将在屏幕中显示一个粒子系统的效果，执行效果如图 21-2 所示。



▲图 21-2　粒子系统的执行效果

21.3 镜像技术

假如有一物体朝镜面运动，其镜像的运动轨迹与该物体的运动轨迹是一模一样的。所以如果在 3D 开发中实现镜像技术，首先需要确定物体的位置，然后根据物体位置计算出镜像的位置，最后在指定的位置绘制镜像。在实际开发过程中经常会遇到一个问题：在绘制镜像时总是会被反射面挡住，此时只要关闭深度检测即可解决该问题。因为开启深度检测会造成不再绘制被挡住的物体，所以才会出现以上问题。

接下来将通过一个具体实例的实现过程，来讲解在 Android 屏幕中实现镜像效果的方法。

题目	目的	源码路径
实例 21-3	在 Android 手机屏幕中实现镜像效果	\daima\21\jingCH

本实例的实现流程如下所示。

（1）编写文件 jingCHControl.java，在此定义一个篮球运动实现类 jingCHControl。首先声明成员变量，并为该类构造器创建一个控制球运动的线程，通过此线程可以控制球的运动，并检验球是否与地板相撞；然后实现绘制物体本身和镜像。文件 jingCHControl.java 的主要实现代码如下所示。

```
public jingCHControl(jingCHTextureByVertex btv,float gaol)
{
    this.btv=btv;
    this.gao=gaol;
    YD=gaol;
    new Thread()
    {//线程运动篮球
        public void run()
        {
            while(true)
            {
                if(isDown)
                {//下降中
                    if(YD>0)
                    {//没有碰到地板则继续下降
                        YD=gao-0.5f*G*zhou*zhou;
                        zhou+=TIME_SPAN;
```

```
                        }
                        else
                        {//如果碰到地板则反弹
                            su=G*zhou;
                            isDown=false;
                            zhou=0;
                            dangsu=su;
                            su=su*0.8f;
                        }
                    }
                    else
                    {//上升中
                        if(su<0.35f)
                        {//如果速度小于阈值则停止
                            YD=0;//球位移置 0
                            break;
                        }

                        if(dangsu>0)
                        {//如果速度大于零则继续上升
                            YD=su*zhou-0.5f*G*zhou*zhou;
                            zhou+=TIME_SPAN;
                            dangsu=su-G*zhou;
                        }
                        else
                        {//如果速度大于等于 0 则下降
                            zhou=0;
                            isDown=true;
                            gao=YD;
                        }
                    }

                    try
                    {
                        Thread.sleep(20);
                    }
                    catch(Exception e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        }.start();
    }

    public void drawSelf(GL10 gl)
    {//绘制物体自己
        gl.glPushMatrix();
        gl.glTranslatef(0, UNIT_SIZE*BALL_SCALE+YD, 0);
        btv.drawSelf(gl);
        gl.glPopMatrix();
    }
    public void drawSelfMirror(GL10 gl)
    {//绘制镜像体
        gl.glPushMatrix();
        gl.glTranslatef(0, -UNIT_SIZE*BALL_SCALE-YD, 0);
        btv.drawSelf(gl);
        gl.glPopMatrix();
    }
}
```

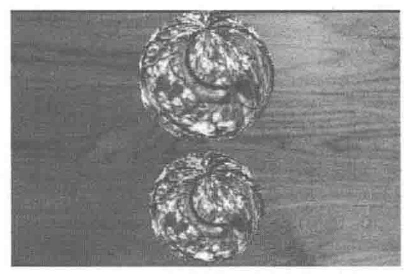
（2）编写文件 mmm.java，在此定义场景绘制类 mmm，并通过方法 initTextur()来初始化纹理。文件 mmm.java 的主要实现代码如下所示。

```
private SceneRenderer mRenderer;//渲染器

    public mm(Context context) {
        super(context);
```

```
        mRenderer = new SceneRenderer();
        setRenderer(mRenderer);
        setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
    }

    private class SceneRenderer implements GLSurfaceView.Renderer
    {
        int lan;
        int basketballTexId;
        ttt put;//普通反射面
        jingCHTextureByVertex qiu;//绘制的球
        jingCHControl kong;

        public void onDrawFrame(GL10 gl) {
            //打开背面剪裁
            gl.glEnable(GL10.GL_CULL_FACE);
            //设置为平滑着色
            gl.glShadeModel(GL10.GL_SMOOTH);
            //清除颜色缓存
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
            //设置为模式矩阵
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            //设置当前矩阵为单位矩阵
            gl.glLoadIdentity();
            //设置 camera 位置
            GLU.gluLookAt
            (
                    gl,
                    0.0f,
                    7.0f,
                    7.0f,
                    0,
                    0f,
                    0,
                    0,
                    1,
                    0
            );
            gl.glTranslatef(0, -2, 0);
            put.drawSelf(gl);
            kong.drawSelfMirror(gl);
            //tr.drawSelf(gl);
            kong.drawSelf(gl);
        }

        public void onSurfaceChanged(GL10 gl, int width, int height) {
            //设置视窗大小及位置
            gl.glViewport(0, 0, width, height);
            //设置当前矩阵为投影矩阵
            gl.glMatrixMode(GL10.GL_PROJECTION);
            //设置当前矩阵为单位矩阵
            gl.glLoadIdentity();
            //计算透视投影的比例
            float ratio = (float) width / height;
            //计算产生透视投影矩阵
            gl.glFrustumf(-ratio, ratio, -1, 1, 3, 100);
        }

        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            //关闭抗抖动
            gl.glDisable(GL10.GL_DITHER);
            //设置 Hint 项目为快速模式
            gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,GL10.GL_FASTEST);
            //设置屏幕背景色
            gl.glClearColor(0,0,0,0);
            //打开深度测试
            //gl.glEnable(GL10.GL_DEPTH_TEST);
            //打开混合
            gl.glEnable(GL10.GL_BLEND);
```

```
            //设置源混合因子与目标混合因子
            gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);

            //纹理初始化
            lan=chuwen(gl,R.drawable.aaa);
            basketballTexId=chuwen(gl,R.drawable.jingxiang);
            put=new ttt(4,2.568f,lan);
            //创建绘制球
            qiu=new jingCHTextureByVertex(BALL_SCALE,basketballTexId);
            //创建控制球
            kong=new jingCHControl(qiu,3f);
        }
    }

    //纹理初始化
    public int chuwen(GL10 gl,int drawableId)//textureId
    {
        //生成纹理 ID
        int[] textures = new int[1];
        gl.glGenTextures(1, textures, 0);
        int currTextureId=textures[0];
        gl.glBindTexture(GL10.GL_TEXTURE_2D, currTextureId);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MIN_FILTER,GL10.GL_NEAREST);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_MAG_FILTER,GL10.GL_LINEAR);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_WRAP_S,GL10.GL_CLAMP_
TO_EDGE);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D,GL10.GL_TEXTURE_WRAP_T,GL10.GL_CLAMP_
TO_EDGE);

        InputStream is = this.getResources().openRawResource(drawableId);
        Bitmap bitmapTmp;
        try
        {
            bitmapTmp = BitmapFactory.decodeStream(is);
        }
        finally
        {
            try
            {
                is.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D,0,bitmapTmp,0);
        bitmapTmp.recycle();

        return currTextureId;
    }
}
```

执行后的效果如图 21-3 所示。



▲图 21-3　镜像的执行效果

21.4 实现旗帜飘扬效果

在日常生活中我们经常会看到旗帜随风飘动的效果，在 3D 应用程序的开发中也可以实现这个效果。旗帜迎风飘动效果的实现并不是很难，总的策略是预先计算出旗帜飘动动画的每一帧，在程序运行时不断切换帧即可。

动画中的每一帧应该是旗帜飘动到某个时刻的一个快照，假定旗帜横向平行于 x 轴，纵向平行于 y 轴，面向 z 轴正方向。整个旗帜可以看作一个矩形，但由于需要制造迎风飘动的效果，因此旗帜的大矩形需要拆分成多个小矩形。要想产生迎风飘动效果，就需要使每一帧旗帜各顶点的 z 坐标按照一定的规律变化。假设旗帜按照正弦曲线飘动，并且包括 4 帧，我们可以将起始点不同的帧放入数组中，启动线程改变数组索引值，绘制出的效果就如同迎风飘扬一样。

在接下来的实例中，根据正弦波动原理实现了一个旗帜飘扬的效果。为了使图像看起来像浮动的样子，需要用单独的数组来存放网格各顶点独立的 x、y、z 坐标。

题目	目的	源码路径
实例 21-4	在 Android 手机屏幕中实现旗帜飘扬效果	\daima\21\qizhiCH

（1）在布局文件中设置插入一个 TextView。

（2）使用数组 vertex 来保存网格各顶点独立的 x、y、z 坐标，在此设置使用 45×45 点形成，具体实现代码如下所示。

```
float vertex[][][]=new float[45][45][3];        // Points 网格顶点数组(45,45,3)
int wiggle_count = 0;                           // 指定旗形波浪的运动速度
float hold;                                     // 临时变量
float xrot, yrot, zrot;
int texture = -1;
FloatBuffer texCoord = FloatBuffer.allocate(8);
FloatBuffer points = FloatBuffer.allocate(12);
```

（3）通过 2 个循环来初始化网格上的点，具体实现代码如下所示。

```
// 沿 x 平面循环
for(int x=0; x<45; x++)
{
   // 沿 y 平面循环
   for(int y=0; y<45; y++)
   {
      // 向表面添加波浪效果
      vertex[x][y][0]=((float)x/5.0f)-4.5f;
      vertex[x][y][1]=(((float)y/5.0f)-4.5f);
      vertex[x][y][2]=(float)(Math.sin(((((float)x/5.0f)*40.0f)/360.0f)*3.141592654
*2.0f));
   }
}
```

（4）开始绘制图形，通过 1 个双循环来处理每个面的顶点和纹理，具体实现代码如下所示。

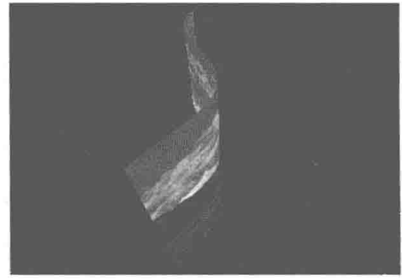
```
or( x = 0; x < 44; x++ ){
   for( y = 0; y < 44; y++) {
      float_x = (float)(x)/44.0f;               // 生成 x 浮点值
      float_y = (float)(y)/44.0f;               // 生成 y 浮点值
      float_xb = (float)(x+1)/44.0f;            // x 浮点值+0.0227f
      float_yb = (float)(y+1)/44.0f;            // y 浮点值+0.0227f

      texCoord.clear();
      texCoord.put(float_x);
      texCoord.put(float_y);
```

```
        texCoord.put(float_x);
        texCoord.put(float_yb);
        texCoord.put(float_xb);
        texCoord.put(float_yb);
        texCoord.put(float_xb);
        texCoord.put(float_y);

        points.clear();
        points.put(vertex[x][y][0]);
        points.put(vertex[x][y][1]);
        points.put(vertex[x][y][2]);

        points.put(vertex[x][y+1][0]);
        points.put(vertex[x][y+1][1]);
        points.put(vertex[x][y+1][2]);

        points.put(vertex[x+1][y+1][0]);
        points.put(vertex[x+1][y+1][1]);
        points.put(vertex[x+1][y+1][2]);

        points.put(vertex[x+1][y][0]);
        points.put(vertex[x+1][y][1]);
        points.put(vertex[x+1][y][2]);
        gl.glDrawArrays(GL10.GL_TRIANGLE_FAN, 0, 4);
```

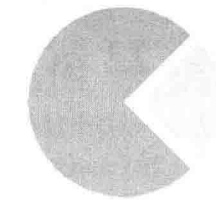
（5）因为要实现波浪效果，所以要绘制 2 次场景。生成波浪效果的实现代码如下所示、

```
if( wiggle_count == 2 )                    // 用来降低波浪速度(每隔 2 帧一次)
{
    for( y = 0; y < 45; y++ )              // 沿 y 平面循环
    {
        hold=vertex[0][y][2];              // 存储当前左侧波浪值
        for( x = 0; x < 44; x++)           // 沿 x 平面循环
        {
            // 当前波浪值等于其右侧的波浪值
            vertex[x][y][2] = vertex[x+1][y][2];
        }
        vertex[44][y][2]=hold;             // 刚才的值成为最左侧的波浪值
    }
    wiggle_count = 0;                      // 计数器清零
```

至此整个程序编写完毕，执行之后的效果如图 21-4 所示。



▲图 21-4　旗帜飘扬的执行效果



第五篇

综合实战篇

第 22 章　开发一个屏保系统
第 23 章　开发一个音乐播放器
第 24 章　开发一个闹钟系统

第 22 章　开发一个屏保系统

屏幕保护是为了保护显示器而设计的一种专门的程序。在计算机上系统中，屏保程序非常常见。例如当计算机处于休眠状态时，会在屏幕中显示预设的界面效果。因为手机也有自己的屏幕，所以很有必要为自己的手机开发一个屏保程序。在本章的内容中，将详细讲解开发一个 Android 手机屏保程序的方法，并详细讲解各个知识点的具体实现流程。

22.1 屏幕保护程序介绍

屏幕保护设计的初衷是为了防止电脑因无人操作而使显示器长时间显示同一个画面，导致老化而缩短显示器寿命。另外，虽然屏幕保护并不是专门为省电而设计的，但一般 Windows 下的屏幕保护程序都比较暗，大幅度降低屏幕亮度，有一定的省电作用。现行显示器分为两种——CRT 显示器和 LCD 显示器，屏幕保护程序对两种显示器有不同影响。

22.1.1 屏幕保护程序的作用

屏幕保护主要有如下 3 个作用。

（1）保护显象管。

由于长时间静止的 Windows 画面会让 CRT 显示器的电子束持续轰击屏幕的某一处，这样可能会造成对 CRT 显示器荧光粉的伤害，所以使用屏幕保护程序会阻止电子束过多地停留在一处，从而延长显示器的使用寿命。

（2）保护个人隐私。

如果你暂时离开电脑，为了防范别人偷窥存放在电脑上的一些隐私，可以在屏幕保护设置中，勾选“在恢复时使用密码保护”复选框，然后单击“电源”按钮，在“电源选项属性”对话框中选择“高级”选项卡，并勾选“在计算机从待机状态恢复时，提示输入密码”复选框。这样，当别人想用你的电脑时，会弹出密码输入框，密码不对的话，无法进入桌面，从而保护个人隐私。

（3）省电。

虽然屏幕保护并不是专门为省电而设计的，但一般 Windows 下的屏幕保护程序都比较暗，大幅度降低屏幕亮度，有一定的省电作用。

22.1.2 手机中的屏幕保护程序

因为手机拥有屏幕，所以也很有必要设置屏保程序。手机屏幕的大小和电脑屏幕的大小有所区别，所以在准备素材图片时必须提前设置图片的大小。在 Android 手机系统中，屏幕大小的高:宽是 480:320。

在当前的手机系统中，都为我们提供了设置屏保功能。我们可以选择喜欢的图片，并把这些图片设置为手机屏保，前提是这些图片被保存在手机中。

22.2 开发屏保程序的原理

在 Android 系统中开发一个屏保程序，其开发原理比较简单，具体说明如下所示。

（1）首先准备一个 Service，当然这个 Service 在主 Activity 中启动；然后在 Service 中注册一个 Receiver，该 Receiver 监听系统的 Screen Off（即屏幕关闭）事件。当然在 Service 中要关闭原有的屏保（关闭系统屏保需要再配置文件中获得权限）。

（2）然后在方法 onReceive 中启动自己的屏保 Activity。在此需要注意，事件 Screen off 不能在配置文件 AndroidManifest.xml 中注册，Receiver 必须在 Java 代码中声明和注册。

根据上述原理，我们开发出一个简单的屏保程序，具体实现流程如下所示。

（1）定义如下两个变量来关闭系统原有屏保。

```
KeyguardManager mKeyguardManager=null;
private KeyguardLock mKeyguardLock=null;
```

（2）通过如下代码关闭系统屏保。

```
mKeyguardManager= (KeyguardManager)getSystemService(Context.KEYGUARD_SERVICE);
mKeyguardLock= mKeyguardManager.newKeyguardLock("");
mKeyguardLock.disableKeyguard();
```

（3）通过如下代码注册 Receiver。

```
BroadcastReceiver mMasterResetReciever= new BroadcastReceiver() {
       public void onReceive(Context context, Intent intent){
          try{
            Intent i = new Intent();
            i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            i.setClass(context, MyScreenSaver.class);
            context.startActivity(i);
            //finish();

          }catch(Exception e){
            Log.i("Output:", e.toString());
          }
       }
     };
registerReceiver(mMasterResetReciever, new IntentFilter(Intent.ACTION_SCREEN_OFF));
```

在此可以看到，在 Receiver 的 onReceive()函数中启动了一个屏保 Activity。

（4）在配置文件中申请权限，具体代码如下所示。

```
<uses-permission android:name="android.permission.DISABLE_KEYGUARD">
</uses-permission>
```

经过上述操作之后，屏保 Activity 设计完成。

在此我们可以将 Activity 设置为全屏显示，具体代码如下所示。

```
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags
        (
          WindowManager.LayoutParams.FLAG_FULLSCREEN,
          WindowManager.LayoutParams.FLAG_FULLSCREEN
        );
setContentView(R.layout.main);
```

在此需要注意，必须将如下代码放在全屏代码之后，否则会无效。

```
setContentView(R.layout.main);
```

我们可以重写 onKeyDown()函数，目的是通过任意键来关闭屏保 Activity，具体代码如下所示。

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event){
    super.onKeyDown(keyCode, event);
    finish();
    return true;
}
```

22.3 开发一个屏保程序

了解了在 Android 系统中开发屏保程序的基本原理后，在本节的内容中，将通过一个具体实例的实现流程，来详细讲解开发 Android 屏保程序的基本流程。本实例的源代码保存在“\daima\22\pingbao”中，下面开始讲解本实例的具体实现流程。

22.3.1 准备素材图片

在本实例中，设置屏保程序轮换显示 5 幅图片，图片的大小是 320×480。本实例的素材图片保存在“res\drawable”目录下，效果如图 22-1 所示。

▲图 22-1　准备的素材图片

22.3.2 编写布局文件

本实例的布局文件是 main.xml，在里面分别插入了一个 ImageView 控件、一个 TextView 和一个 EditText，主要代码如下所示。

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

```
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <ImageView
    android:id="@+id/myImageView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:scaleType="fitCenter"
    android:layout_gravity="center" />
  <TextView
    android:id="@+id/myTextView1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="@drawable/blue"
    android:visible="true"
    android:text="@string/str_set_pwd"/>
  <EditText
    android:id="@+id/myEditText1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text=""
  />
</LinearLayout>
```

22.3.3 编写主程序文件

本实例的主程序文件是 example.java，其具体实现流程如下所示。

（1）先引入相关 class 类，然后设置 LayoutInflater 对象作为新建的 AlertDialog，具体代码如下所示：

```
package irdc.example;

import irdc.example.R;

import java.util.Date;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Matrix;
import android.graphics.drawable.BitmapDrawable;
import android.os.Bundle;
import android.os.Handler;
import android.util.DisplayMetrics;
import android.util.Log;
import android.view.KeyEvent;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuItem;
import android.view.MotionEvent;
import android.view.View;
import android.view.Window;
import android.view.WindowManager;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.TextView;

public class example extends Activity
{
  private TextView mTextView01;
  private ImageView mImageView01;

  /* LayoutInflater 对象作为新建 AlertDialog 之用 */
  private LayoutInflater mInflater01;
```

（2）定义 mView01，用于输入解锁的 View。通过 menu 选项 identifier，用以识别对应的事件，具体代码如下所示。

```
/* 输入解锁的 View */
private View mView01;
private EditText mEditText01,mEditText02;

/* menu 选项 identifier，用以识别事件 */
static final private int MENU_ABOUT = Menu.FIRST;
static final private int MENU_EXIT = Menu.FIRST+1;
private Handler mHandler01 = new Handler();
private Handler mHandler02 = new Handler();
private Handler mHandler03 = new Handler();
private Handler mHandler04 = new Handler();
```

（3）分别定义控制 User 静止与否的 Counter，控制 FadeIn 与 Fade Out 的 Counter，控制循序替换背景图 ID 的 Counter，具体代码如下所示。

```
/* 控制 User 静止与否的 Counter */
private int intCounter1, intCounter2;
/* 控制 FadeIn 与 Fade Out 的 Counter */
private int intCounter3, intCounter4;
/* 控制循序替换背景图 ID 的 Counter  */
private int intDrawable=0;
```

（4）设置 timePeriod，设置当静止超过 *n* 秒将自动进入屏幕保护，具体代码如下所示。

```
/* 上一次 User 有动作的 Time Stamp */
private Date lastUpdateTime;
/* 计算 User 共几秒没有动作 */
private long timePeriod;
/* 静止超过 n 秒将自动进入屏幕保护 */
private float fHoldStillSecond = (float) 5;
private boolean bIfRunScreenSaver;
private boolean bFadeFlagOut, bFadeFlagIn = false;
private long intervalScreenSaver = 1000;
private long intervalKeypadeSaver = 1000;
private long intervalFade = 100;
private int screenWidth, screenHeight;
```

（5）设置每 5 秒置换一次图片，并设置使用 Screen Saver 保存需要用到的背景图，具体代码如下所示。

```
/* 每 n 秒置换图片 */
private int intSecondsToChange = 5;

/* 设置 Screen Saver 需要用到的背景图 */
private static int[] screenDrawable = new int[]
{
  R.drawable.pingbao1,
  R.drawable.pingbao 2,
  R.drawable.pingbao 3,
  R.drawable.pingbao 4,
  R.drawable.pingbao 5
};
```

（6）设置在 setContentView 之前调用全屏幕显示，通过 lastUpdateTime 初始取得 User 用户触碰手机的时间，并用 recoverOriginalLayout()来初始化 Layout 屏幕上的 Widget 可见性，具体代码如下所示。

```
@Override
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
```

```
  /* 必须在 setContentView 之前调用全屏幕显示 */
  requestWindowFeature(Window.FEATURE_NO_TITLE);
  getWindow().setFlags
  (
    WindowManager.LayoutParams.FLAG_FULLSCREEN,
    WindowManager.LayoutParams.FLAG_FULLSCREEN
  );
  setContentView(R.layout.main);

  /* onCreate all Widget */
  mTextView01 = (TextView)findViewById(R.id.myTextView1);
  mImageView01 = (ImageView)findViewById(R.id.myImageView1);
  mEditText01 = (EditText)findViewById(R.id.myEditText1);

  /* 初始取得 User 触碰手机的时间 */
  lastUpdateTime = new Date(System.currentTimeMillis());

  /* 初始化 Layout 上的 Widget 可见性 */
  recoverOriginalLayout();
}
```

（7）设置 Menu 群组 ID，然后通过 menu.add 创建具有 SubMenu 的 Menu，最后创建退出 Menu，具体代码如下所示。

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
  // TODO Auto-generated method stub

  /* Menu 群组 ID */
  int idGroup1 = 0;

  /* The order position of the item */
  int orderMenuItem1 = Menu.NONE;
  int orderMenuItem2 = Menu.NONE+1;

  /* 创建具有 SubMenu 的 Menu */
  menu.add
  (
    idGroup1, MENU_ABOUT, orderMenuItem1, R.string.app_about
  );
  /* 创建退出 Menu */

  menu.add(idGroup1, MENU_EXIT, orderMenuItem2, R.string.str_exit);
  menu.setGroupCheckable(idGroup1, true, true);

  return super.onCreateOptionsMenu(menu);
}
```

（8）根据用户选择的 Menu，显示对应的 AlertDialog 提示框，具体代码如下所示。

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
  // TODO Auto-generated method stub
  switch(item.getItemId())
  {
    case (MENU_ABOUT):
      new AlertDialog.Builder
      (
        example.this
      ).setTitle(R.string.app_about).setIcon
      (
        R.drawable.hippo
      ).setMessage
      (
        R.string.app_about_msg
```

```
      ).setPositiveButton(R.string.str_ok,
      new DialogInterface.OnClickListener()
      {
        public void onClick
        (DialogInterface dialoginterface, int i)
        {
        }
      }).show();
      break;
    case (MENU_EXIT):
      /* 离开程序 */
      finish();
      break;
  }
  return super.onOptionsItemSelected(item);
}
```

（9）用 mTasks01 监控 User 没有动作的运行线程，通过 timePeriod 计算 User 静止不动的时间间距，如果静止不懂查过设置的 5 秒，则运行对应的线程，具体代码如下所示。

```
/* 监控 User 没有动作的运行线程 */
private Runnable mTasks01 = new Runnable()
{
  public void run()
  {
    intCounter1++;
    Date timeNow = new Date(System.currentTimeMillis());

    /* 计算 User 静止不动的时间间距 */
    timePeriod =
    (long)timeNow.getTime() - (long)lastUpdateTime.getTime();

    float timePeriodSecond = ((float)timePeriod/1000);

    /* 如果超过时间静止不动 */
    if(timePeriodSecond>fHoldStillSecond)
    {
      /* 静止超过时间第一次的标记 */
      if(bIfRunScreenSaver==false)
      {
        /* 启动运行线程 2 */
        mHandler02.postDelayed(mTasks02, intervalScreenSaver);

        /* Fade Out*/
        if(intCounter1%(intSecondsToChange)==0)
        {
          bFadeFlagOut=true;
          mHandler03.postDelayed(mTasks03, intervalFade);
        }
        else
        {
          /* 在 Fade Out 后立即 Fade In */
          if(bFadeFlagOut==true)
          {
            bFadeFlagIn=true;
            mHandler04.postDelayed(mTasks04, intervalFade);
          }
          else
          {
            bFadeFlagIn=false;
            intCounter4 = 0;
            mHandler04.removeCallbacks(mTasks04);
          }
          intCounter3 = 0;
          bFadeFlagOut = false;
        }
        bIfRunScreenSaver = true;
      }
```

```
      else
      {
        /* screen saver 正在运行中 */

        /* Fade Out*/
        if(intCounter1%(intSecondsToChange)==0)
        {
          bFadeFlagOut=true;
          mHandler03.postDelayed(mTasks03, intervalFade);
        }
        else
        {
          /* 在 Fade Out 后立即 Fade In */
          if(bFadeFlagOut==true)
          {
            bFadeFlagIn=true;
            mHandler04.postDelayed(mTasks04, intervalFade);
          }
          else
          {
            bFadeFlagIn=false;
            intCounter4 = 0;
            mHandler04.removeCallbacks(mTasks04);
          }
          intCounter3 = 0;
          bFadeFlagOut=false;
        }
      }
    }
    else
    {
      /* 当 User 没有动作的间距未超过时间 */
      bIfRunScreenSaver = false;
      /* 恢复原来的 Layout Visible*/
      recoverOriginalLayout();
    }

    /* 以 LogCat 监看 User 静止不动的时间间距 */
    Log.i
    (
      "HIPPO",
      "Counter1:"+Integer.toString(intCounter1)+
      "/"+
      Float.toString(timePeriodSecond));

    /* 反复运行运行线程 1 */
    mHandler01.postDelayed(mTasks01, intervalKeypadeSaver);
  }
};
```

（10）定义 mTasks02，设置每 1 秒运行一次屏保程序，并隐藏原有 Layout 上面的 Widget，并调用 ScreenSaver()加载图片，即轮换显示预设的 5 幅图片，具体代码如下所示。

```
/* Screen Saver Runnable */
private Runnable mTasks02 = new Runnable()
{
  public void run()
  {
    if(bIfRunScreenSaver==true)
    {
      intCounter2++;

      hideOriginalLayout();
      showScreenSaver();

      //Log.i("HIPPO", "Counter2:"+Integer.toString(intCounter2));
      mHandler02.postDelayed(mTasks02, intervalScreenSaver);
    }
```

```
    else
    {
      mHandler02.removeCallbacks(mTasks02);
    }
  }
};
```

（11）定义 mTasks03，通过 setAlpha 设置 ImageView 的透明度渐暗下去，具体代码如下所示。

```
/* Fade Out 特效 Runnable */
private Runnable mTasks03 = new Runnable()
{
  public void run()
  {
    if(bIfRunScreenSaver==true && bFadeFlagOut==true)
    {
      intCounter3++;

      /* 设置 ImageView 的透明度渐暗下去 */
      mImageView01.setAlpha(255-intCounter3*28);

      Log.i("HIPPO", "Fade out:"+Integer.toString(intCounter3));
      mHandler03.postDelayed(mTasks03, intervalFade);
    }
    else
    {
      mHandler03.removeCallbacks(mTasks03);
    }
  }
};
```

（12）定义 mTasks03，通过 setAlpha 设置设置 ImageView 的透明度渐亮起来，具体代码如下所示。

```
/* Fade In 特效 Runnable */
private Runnable mTasks04 = new Runnable()
{
  public void run()
  {
    if(bIfRunScreenSaver==true && bFadeFlagIn==true)
    {
      intCounter4++;

      /* 设置 ImageView 的透明度渐亮起来 */
      mImageView01.setAlpha(intCounter4*28);

      mHandler04.postDelayed(mTasks04, intervalFade);
      Log.i("HIPPO", "Fade In:"+Integer.toString(intCounter4));
    }
    else
    {
      mHandler04.removeCallbacks(mTasks04);
    }
  }
};
```

（13）先定义 recoverOriginalLayout()方法，用于恢复原有的 Layout 可视性；然后定义 hideOriginalLayout()方法，用于隐藏原有应用程序里的布局配置组件，具体代码如下所示。

```
/* 恢复原有的 Layout 可视性 */
private void recoverOriginalLayout()
{
  mTextView01.setVisibility(View.VISIBLE);
  mEditText01.setVisibility(View.VISIBLE);
  mImageView01.setVisibility(View.GONE);
}
```

```
/* 隐藏原有应用程序里的布局配置组件 */
private void hideOriginalLayout()
{
  /* 将欲隐藏的 Widget 写在此 */
  mTextView01.setVisibility(View.INVISIBLE);
  mEditText01.setVisibility(View.INVISIBLE);
}

/* 开始 ScreenSaver */
private void showScreenSaver()
{
  /* 屏幕保护之后要做的事件写在此*/

  if(intDrawable>4)
  {
    intDrawable = 0;
  }

  DisplayMetrics dm=new DisplayMetrics();
  getWindowManager().getDefaultDisplay().getMetrics(dm);
  screenWidth = dm.widthPixels;
  screenHeight = dm.heightPixels;
  Bitmap bmp=BitmapFactory.decodeResource(getResources(),screenDrawable[intDrawable]);
```

（14）通过 Matrix 设置比例，使用 Matrix.postScale 设置维度 ReSize，通过 resizedBitmap 对象设置图文件至屏幕分辨率，新建 Drawable 对象 myNewBitmapDrawable 用于放大图文件至全屏幕，通过 setVisibility(View.VISIBLE)使 ImageView 可见，具体代码如下所示。

```
  /* Matrix 比例 */
  float scaleWidth = ((float) screenWidth) / bmp.getWidth();
  float scaleHeight = ((float) screenHeight) / bmp.getHeight() ;

  Matrix matrix = new Matrix();
  /* 使用 Matrix.postScale 设置维度 ReSize */
  matrix.postScale(scaleWidth, scaleHeight);

  /* ReSize 图文件至屏幕分辨率 */
  Bitmap resizedBitmap = Bitmap.createBitmap
  (
    bmp,0,0,bmp.getWidth(),bmp.getHeight(),matrix,true
  );

  /* 新建 Drawable 放大图文件至全屏幕 */
  BitmapDrawable myNewBitmapDrawable =
      new BitmapDrawable(resizedBitmap);
  mImageView01.setImageDrawable(myNewBitmapDrawable);

  /* 使 ImageView 可见 */
  mImageView01.setVisibility(View.VISIBLE);

  /* 每间隔设置秒数置换图片 ID，于下一个 runnable2 才会生效 */
  if(intCounter2%intSecondsToChange==0)
  {
    intDrawable++;
  }
}
```

（15）定义方法 onUserWakeUpEvent()，实现解锁和加密处理，具体代码如下所示。

```
public void onUserWakeUpEvent()
{
  if(bIfRunScreenSaver==true)
  {
    try
    {
      /* LayoutInflater.from 取得此 Activity 的 context */
      mInflater01 = LayoutInflater.from(example.this);
```

```
    /* 创建解锁密码使用 View 的 Layout */
    mView01 = mInflater01.inflate(R.layout.securescreen, null);

    /* 于对话框中唯一的 EditText 等待输入解锁密码 */
    mEditText02 =
    (EditText) mView01.findViewById(R.id.myEditText2);

    /* 创建 AlertDialog */
    new AlertDialog.Builder(this)
    .setView(mView01)
    .setPositiveButton("OK",
    new DialogInterface.OnClickListener()
    {
      public void onClick(DialogInterface dialog, int whichButton)
      {
        /* 比较输入的密码与原 Activity 里的设置是否相符 */
        if(mEditText01.getText().toString().equals
          (mEditText02.getText().toString()))
        {
          /* 当密码正确才解锁屏幕保护装置 */
          resetScreenSaverListener();
        }
      }
    }).show();
  }
  catch(Exception e)
  {
    e.printStackTrace();
  }
 }
}
```

（16）定义方法 updateUserActionTime()，用于统计用户单击键盘或屏幕的时间间隔，具体实现流程如下所示。

- 第一步：取得单击按键事件时的系统 Time Millis。
- 第二步：重新计算单击按键距离上一次静止的时间间距。

方法 updateUserActionTime()的具体代码如下所示。

```
public void updateUserActionTime()
{
  /* 取得单击按键事件时的系统 Time Millis */
  Date timeNow = new Date(System.currentTimeMillis());

  /* 重新计算单击按键距离上一次静止的时间间距 */
  timePeriod =
  (long)timeNow.getTime() - (long)lastUpdateTime.getTime();
  lastUpdateTime.setTime(timeNow.getTime());
}
```

（17）定义方法 resetScreenSaverListener()来重新设置屏幕，具体实现流程如下所示。

- 第一步：删除现有的 Runnable。
- 第二步：取得单击按键事件时的系统 Time Millis。
- 第三步：重新计算单击按键距离上一次静止的时间间距。
- 第四步：通过 bIfRunScreenSaver 取消屏保。
- 第五步：恢复原来的 Layout Visible。

方法 resetScreenSaverListener()的具体代码如下所示。

```
public void resetScreenSaverListener()
{
  /* 删除现有的 Runnable */
  mHandler01.removeCallbacks(mTasks01);
```

```
    mHandler02.removeCallbacks(mTasks02);

    /* 取得单击按键事件时的系统 Time Millis */
    Date timeNow = new Date(System.currentTimeMillis());
    /* 重新计算单击按键距离上一次静止的时间间距 */
    timePeriod =
    (long)timeNow.getTime() - (long)lastUpdateTime.getTime();
    lastUpdateTime.setTime(timeNow.getTime());

    /* for Runnable2, 取消屏幕保护 */
    bIfRunScreenSaver = false;

    /* 重置 Runnable1 与 Runnable1 的 Counter */
    intCounter1 = 0;
    intCounter2 = 0;

    /* 恢复原来的 Layout Visible*/
    recoverOriginalLayout();

    /* 重置 postDelayed()的 Runnable */
    mHandler01.postDelayed(mTasks01, intervalKeypadeSaver);
  }
```

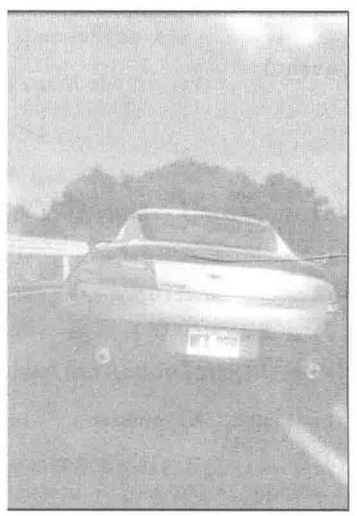
（18）定义 onKeyDown(int keyCode, KeyEvent event)，用于监听用户的触摸单击事件，具体代码如下所示。

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event)
{
  // TODO Auto-generated method stub
  if(bIfRunScreenSaver==true && keyCode!=4)
  {
    /* 当屏幕保护程序正在运行中，触动解除屏幕保护程序 */
    onUserWakeUpEvent();
  }
  else
  {
    /* 更新 User 未触动手机的时间戳记 */
    updateUserActionTime();
  }
  return super.onKeyDown(keyCode, event);
}
@Override
public boolean onTouchEvent(MotionEvent event)
{
  // TODO Auto-generated method stub
  if(bIfRunScreenSaver==true)
  {
    /* 当屏幕保护程序正在运行中，触动解除屏幕保护程序 */
    onUserWakeUpEvent();
  }
  else
  {
    /* 更新 User 未触动手机的时间戳记 */
    updateUserActionTime();
  }
  return super.onTouchEvent(event);
}

@Override
protected void onResume()
{
  // TODO Auto-generated method stub
  mHandler01.postDelayed(mTasks01, intervalKeypadeSaver);
  super.onResume();
}
```

（19）定义方法 onPause()来删除正在运行中的运行线程 mHandler01、mHandler02、mHandler03 和 mHandler01，具体代码如下所示。

```
  @Override
  protected void onPause()
  {
    // TODO Auto-generated method stub

    try
    {
      /* 删除运行中的运行线程 */
      mHandler01.removeCallbacks(mTasks01);
      mHandler02.removeCallbacks(mTasks02);
      mHandler03.removeCallbacks(mTasks03);
      mHandler04.removeCallbacks(mTasks04);
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
    super.onPause();
  }
}
```

至此，整个实例介绍完毕。执行后如果超过 5 秒不动键盘或屏幕，则会进入屏保状态，如图 22-2 所示。可以设置屏保密码，当输入正确的密码后才能解除屏保，如图 22-3 所示。



▲图 22-2　屏幕保护执行效果

▲图 22-3　屏幕保护解锁密码

在本实例的实现代码中，声明的 4 个 Runnable 是整个程序的重点，这 4 个 Runnable 的具体说明如下所示。

- mTasks01：设置每 1 秒检查一次 timePeriod，并监视是否超过 5 秒未触发。超过 5 秒则将 blRunScreenSaver 这个 flag 更改为 true，并启动 mTasks02。
- mTasks02：设置每 1 秒运行一次屏保程序，并隐藏原有 Layout 上面的 Widget，并调用 ScreenSaver()加载图片，即轮换显示预设的 5 幅图片。
- mTasks03：是 Fade-Out 特效使用的 Runable，每 0.1 秒运行一个 scale。
- mTasks04：是 Fade-In 特效使用的 Runable，每 0.1 秒运行一个 scale。

第 23 章　开发一个音乐播放器

当今社会的生活节奏越来越快，随着硬件移动设备越来越先进，人们对移动设备的要求也越来越高，从以前的追求技术到现在的追求视觉，因此，也逐步提高了对系统的要求。本章的音乐播放器实例采用了 Android 开源系统技术，利用 Java 语言和 Eclipse 编辑工具开发播放器，同时给出了详细的系统设计过程、部分界面图及主要功能运行流程图。在本章的内容中，还对开发过程中遇到的问题和解决方法进行了详细的讨论。这个音乐播放器实例集播放、暂停、停止、上一首、下一首、音量调节、歌词显示等功能于一体，性能良好，在 Android 系统中能独立运行。该播放器还拥有对手机文件浏览器的访问功能、歌曲播放模式以及歌词开闭状态的友好设置。“.MP3”的全名是 MPEG Audio Layer-3，是一种声音文件的压缩格式。因为本播放器只限于应用层程序的探讨，所以对具体的压缩算法不做深究。

23.1 项目介绍

本章播放器源码保存在本书附带源程序中的“\daima\23”目录下。在讲解具体编码之前，先简要介绍本项目的产生背景和项目的知识，为后面的具体编码打好理论基础。

23.1.1 项目背景介绍

当今社会的生活节奏越来越快，人们对手机的要求也越来越高。由于手机市场发展迅速，使得手机操作系统也出现了不同种类。现在的市场上主要有三种手机操作系统：Windows Mobile，Symbian，以及谷歌的 Android 操作系统，其中占有开放源代码优势的 Android 系统有最大的发展前景。那么能否在手机上拥有自己编写的个性音乐播放器呢？答案是完全可以！谷歌 Android 系统就能做到。本章讲解的的音乐播放器实例就是基于谷歌 Android 手机平台的播放器。

随着计算机的广泛运用，手机市场的迅速发展，各种音频、视频资源也在网上广为流传，这些资源看似平常，但已经渐渐成为人们生活中必不可少的一部分。于是各种手机播放器也紧跟着发展起来，但是很多播放器一味追求外观花哨，功能庞大，对用户的手机造成了资源浪费，比如 CPU、内存等的占用率过高，在用户需要多任务操作时，受到了不小的影响，带来了许多不便。而对于大多数普通用户，许多功能用不上，形同虚设。针对以上各种弊端，我们选择了开发多语种的音频、视频播放器，将各种性能优化，继承播放器的常用功能，满足一般用户的需求（如听歌，看电影）。除了能播放常见格式的语音视频文件，还有一些高级功能，如能播放 RMVB 格式的视频文件。此外，还有能支持中文、英文等语言界面。

要研究市场上流行的各种手机播放器，了解它们各自的插件及编码方式，还有各种播放器播放的特别格式文件，分析各种编码的优缺点以及各种播放器本身存在的缺陷和特点，这样才能编写出功能实用、使用方便快捷的播放器。目前已经实现的功能有播放常见音频文件，如 MP3 和 WAV 等，拥有播放菜单，能选择播放清单，具备一般播放器的功能，如快进、快退、音量调节等。

播放模式也比较完善，如有单曲播放、顺序播放、循环播放和随机播放等模式。

23.1.2　项目的目的

当今社会的生活节奏快，工作压力大，欣赏音乐是舒缓压力的最好方式之一。本项目的目的是开发一个可以播放主流音乐文件格式的播放器。这个播放器实例的主要功能是播放 MP3、WAV 等多种格式的音乐文件，并且能够控制播放、暂停、停止、上一曲、下一曲，也具备音量调节功能，并且具有好的视觉外观，还具有播放列表和歌曲文件的管理操作等多种播放控制功能，界面简明，操作简单。

本项目是一款基于 Android 手机平台的音乐播放器，使 Android 手机拥有个性的多媒体播放器，显得更生动、灵活，与人们更为接近，使人们的生活更加多样化，也使设计者更加熟练 Android 的技术及其在市场上的特点。

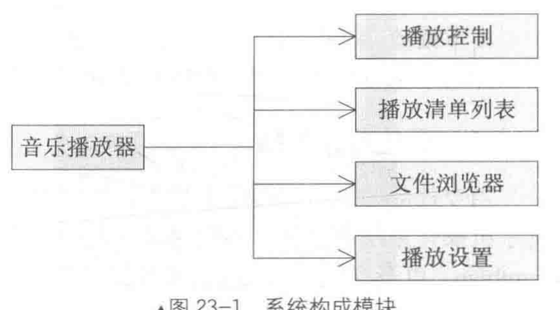
23.2 系统需求分析

根据项目的目标，我们可获得项目系统的基本需求。以下使用例图从不同角度来描述系统的需求分成 4 部分来概括，即播放器的基本控制需求、播放列表管理需求、播放器友好性需求和播放器扩展卡需求。

23.2.1　构成模块

本系统的构成模块如图 23-1 所示。

各个模块的具体说明如下所示。



▲图 23-1　系统构成模块

1. 播放控制模块

此模块的功能是控制播放的音乐文件。

（1）播放。

- 目标：使用户可以播放在播放列表中选中的歌曲。
- 前置条件：播放器正在运行。
- 基本事件流。
  - ➢ 用户单击“播放”按钮。
  - ➢ 播放器将播放列表中当前的歌曲。

（2）暂停播放。

- 目标：使用户可以暂停正在播放的歌曲。
- 前置条件：歌曲正在播放且未停止和暂停。
- 基本事件流。
  - ➢ 单击“暂停”按钮。
  - ➢ 播放器将暂停当前的歌曲。

（3）停止播放。

- 目标：使用户可以停止正在播放的歌曲。
- 前置条件：歌曲正在播放或暂停。
- 基本事件流。
  - ➢ 用户单击“停止”按钮。

➢ 播放器将停止当前播放的歌曲。

（4）上一首/下一首。

- 目标：使用户可以听上一首或下一首歌曲。
- 前置条件：歌曲正在播放或暂停。
- 基本事件流。

➢ 用户单击“上一首”或“下一首”按钮。

➢ 播放器将播放上一首或下一首歌曲。

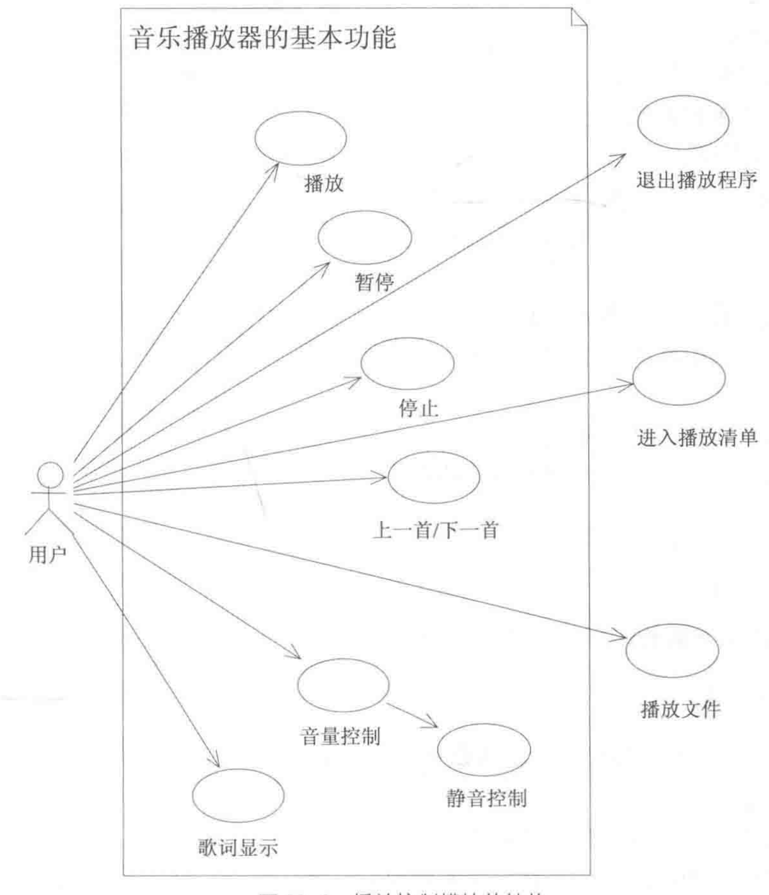
（5）播放清单。

- 目标：使用户可以进入播放清单。
- 前置条件：程序在运行。
- 基本事件流。

➢ 用户单击“清单”按钮。

➢ 播放器进入清单列表。

本实例播放控制界面的基本结构如图 23-2 所示。



▲图 23-2 播放控制模块的结构

2. 播放清单列表模块

当用户选中列表中的某一首歌曲后会显示播放清单，我们可以进行如下操作。

（1）播放。

- 目标：使程序播放选中的歌曲。
- 前置条件：程序运行在播放菜单选项中。
- 基本事件流。

➢ 用户单击“播放”按钮。

➢ 播放器进入播放状态。

（2）视频详情

- 目标：使程序显示歌曲详情。
- 前置条件：程序运行在播放菜单选项中。
- 基本事件流。

➢ 用户单击“详细”按钮。

➢ 显示歌曲详细状态。

（3）增加。

- 目标：使程序进入手机扩展 SD 卡。
- 前置条件：程序运行在播放菜单选项中。
- 基本事件流。

➢ 用户单击“增加”按钮。

➢ 播放器进入手机扩展 SD 卡。

（4）移除/全部移除。

- 目标：使选中的歌曲被移除。
- 前置条件：程序运行在播放菜单选项中。
- 基本事件流。

➢ 用户单击“移除/全部移除”按钮。

➢ 播放器移除选中歌曲/全部移除歌曲。

（5）设定。

- 目标：使程序进入播放器设定状态。
- 前置条件：程序运行在播放菜单选项中。
- 基本事件流。

➢ 用户单击“设定”按钮。

➢ 播放器进入设定界面。

本实例播放清单界面的结构如图 23-3 所示。

3. 播放设置模块

本模块用于设置音乐的播放方式，并设置是否显示歌词。

（1）播放模式。

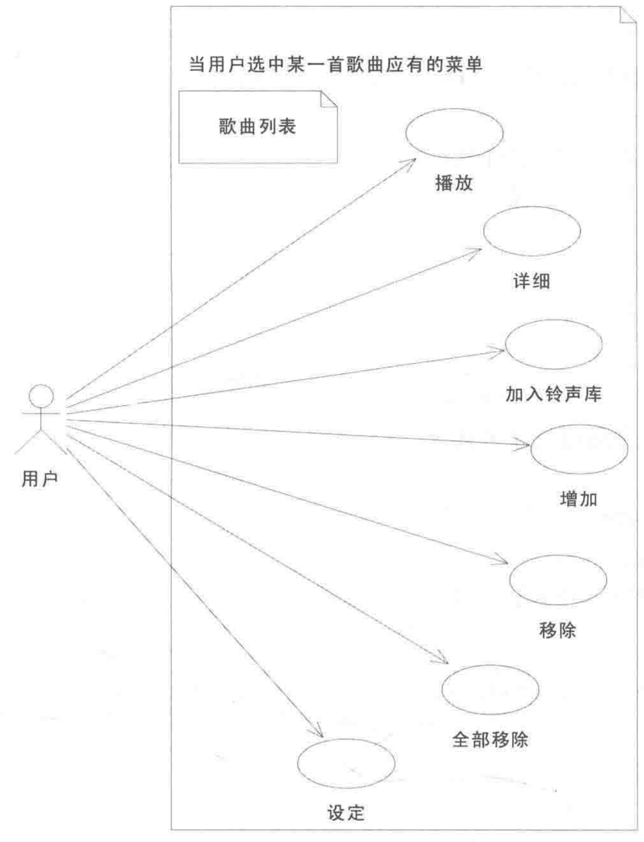
- 目标：使程序进入播放模式设定状态。
- 前置条件：程序运行在播放器设定界面中。
- 基本事件流。

➢ 用户单击“顺序”、“随机”、“单曲”按钮。

➢ 播放器进入选中模式播放状态。

（2）显示歌词。

- 目标：使程序进入播放器歌词设置状态。



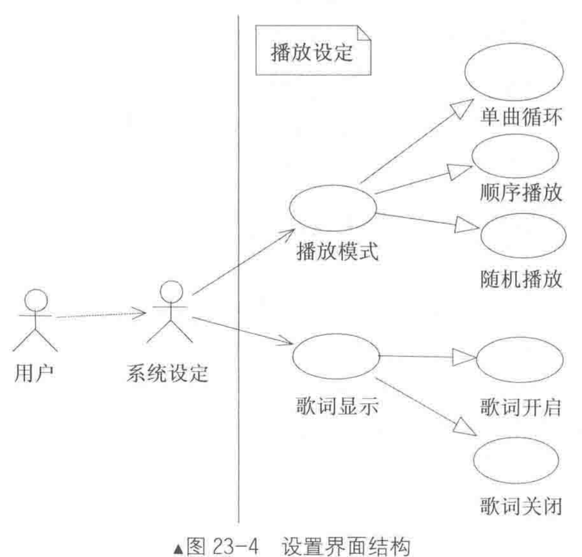
▲图 23-3 播放清单界面结构

- 前置条件：程序运行在播放器设定界面中。
- 基本事件流中。

➢ 用户单击“歌词开关按钮”按钮。

➢ 播放器显示或关闭歌词。

本实例设置界面的结构如图 23-4 所示。



▲图 23-4 设置界面结构

4. 文件浏览器模块

此模块的功能是浏览系统内或 SD 卡中的文件信息。

（1）SD 卡。

- 目标：使程序进入 SD 卡目录。
- 前置条件：程序运行目录界面。
- 基本事件流。

➢ 用户单击“Sdcard”按钮。

➢ 程序进入 Sdcard 目录。

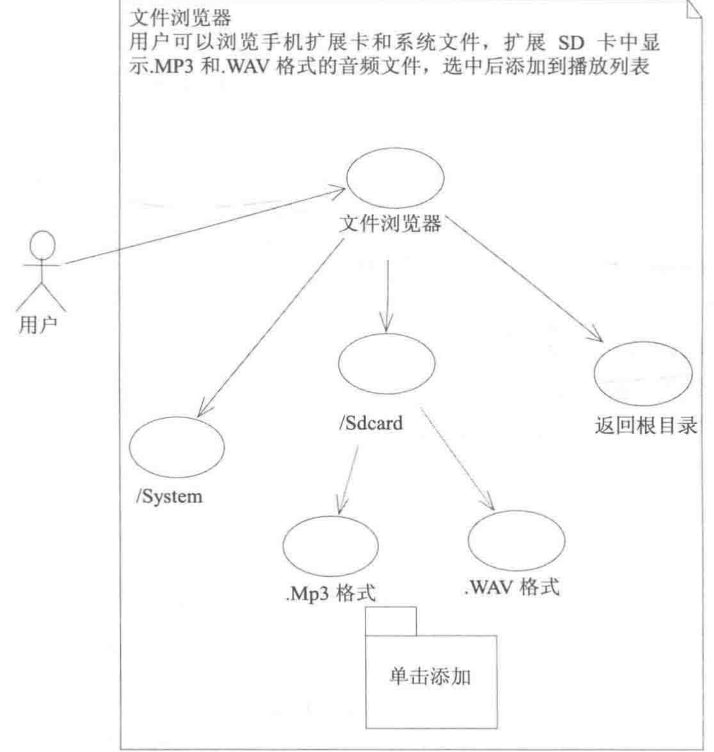
（2）系统

- 目标：使程序进入 System 目录。
- 前置条件：程序运行目录界面。
- 基本事件流。

➢ 用户单击“System”按钮。

➢ 程序进入 System 目录。

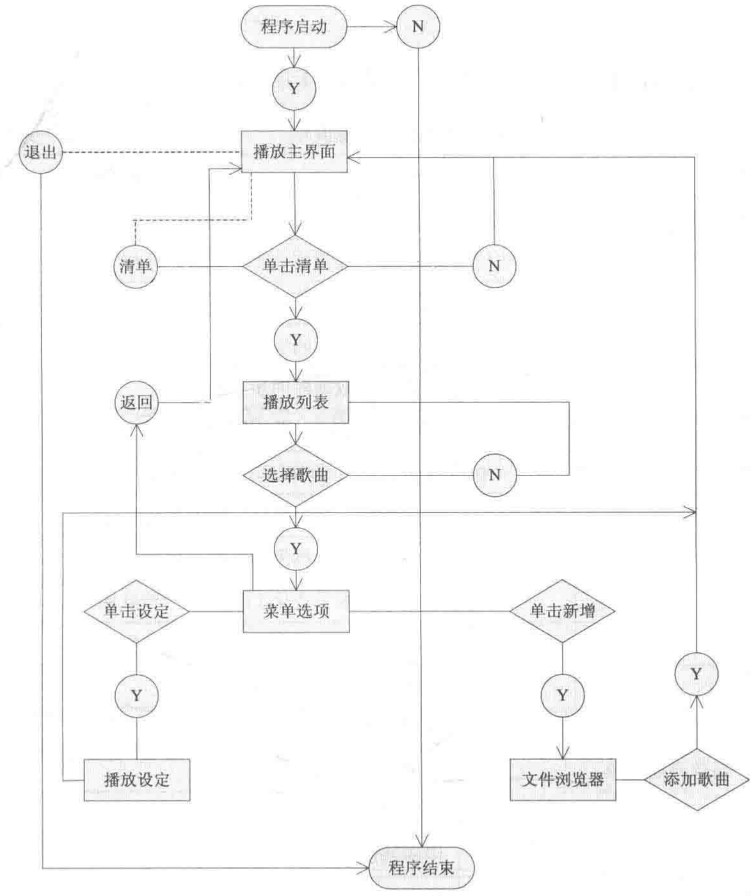
本实例文件浏览器模块的结构如图 23-5 所示。



▲图 23-5 文件浏览器模块结构

23.2.2 系统流程

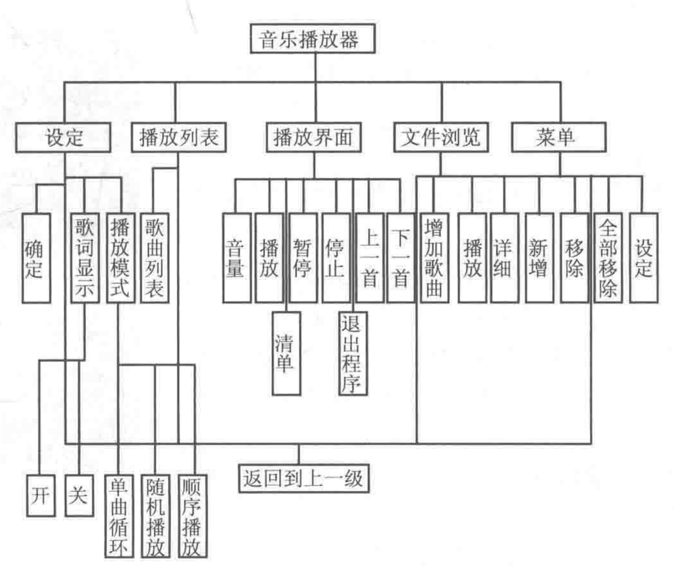
本章音乐播放器的系统流程如图 23-6 所示。



▲图 23-6 音乐播放器系统流程图

23.2.3 功能结构图

本章音乐播放器系统的完整功能结构如图 23-7 所示。



▲图 23-7 完整功能结构图

23.2.4　系统功能说明

本章音乐播放器系统各个模块功能的说明如表 23-1 所示。

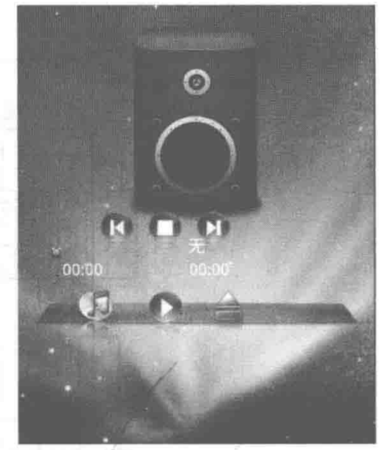
表 23-1　　模块结构功能说明

功能类别	子 功 能	子 功 能
播放列表	播放列表菜单	退出播放
		从扩展卡寻找歌曲
	歌曲菜单	播放　进入播放界面
		删除　数据库同步更新
		重命名　数据库同步更新
		向上、下移动　数据库同步更新
播放界面	播放	播放歌曲　线程启动　时间更新
	暂停	暂停歌曲　线程暂停　时间暂停
	停止	停止歌曲　线程停止　时间停止
	上一首	播放列表索引变化　寻找上一首歌曲
	下一首	播放列表索引变化　寻找下一首歌曲
	播放界面菜单	返回到播放列表
		返回到主菜单
		从扩展卡寻找歌曲
		退出播放器
		隐藏播放界面
主菜单	退出程序	程序退出
	进入播放列表	显示播放列表

23.2.5　系统需求

（1）系统界面需求。

播放器界面要求布局合理，颜色舒适，控制按钮友好。为了减少开发工程量，图片素材多数为公司项目素材。如图 23-8 所示。



▲图 23-8　播放器主界面

（2）系统性能需求。

根据 Android 手机系统要求无响应时间为 5 秒，所以就有如下性能要求。

- 当要求歌曲播放时，程序响应时间最长不能超过 5 秒；
- 当要求歌曲暂停时，程序响应时间最长不能超过 5 秒；
- 当要求歌曲停止时，程序响应时间最长不能超过 5 秒；
- 当要求歌曲上一首/下一首时，程序响应时间最长不能超过 5 秒；
- 当要求进行清单列表时，程序响应时间最长不能超过5秒。

（3）运行环境需求。

- 操作系统：Android 手机基于 Linux 操作系统。
- 支持环境：Android 1.5～2.0.1 版本。

- 开发环境：Eclipse 3.5 ADT 0.95。

23.3 数据库设计

数据库是存放数据的仓库。只不过这个仓库是在计算机存储设备上，而且数据是按一定的格式存放的。数据库中的数据按一定数据模型组织、描述和存储，具有较小的重复度、较高的数据独立性和易扩展性，并且可以被在一定范围内的各种用户共享。在涉及数据库的软件开发中，需要根据有待解决的问题性质、规模，以及所采用的前端程序创建工具等，选择合适的数据库类型。

23.3.1 字段设计

字段 file_table 用于保存歌曲的名字、类型和路径，具体说明如表 23-2 所示。

表 23-2 字段 file_table 说明

属　性	数据类型	说　明
_Id	INTEGER	id 号
fileName	TEXT	歌曲名字
filePath	TEXT	歌曲路径
sort	INTEGER	歌曲类型

SD 卡中保存歌曲的详情如表 23-3 所示。

表 23-3 歌曲详情表

属　性	数据类型	说　明
_ID	INTEGER	id 号
TITLE	TEXT	标题
ARTIST	TEXT	艺术家
ALBUM	TEXT	专辑
SIZE	LONG	大小

在 Android 系统中，通过自带的 MediaStore 封闭类来存储媒体信息，通过 Uri EXTERNAL_CONTENT_URI 来访问 SD 卡中的歌曲详细信息。

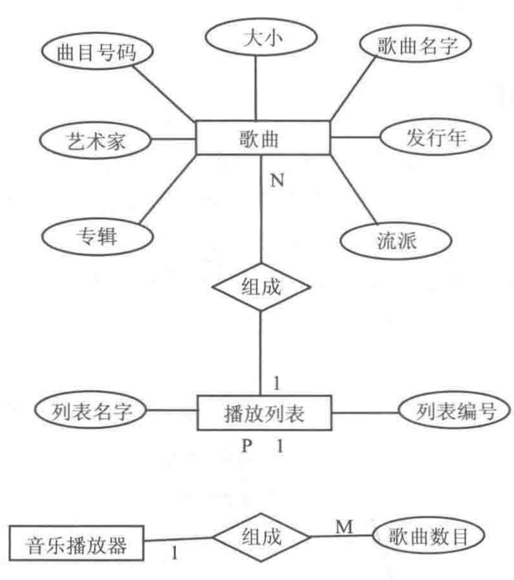
23.3.2 E-R 图设计

音乐播放器的 E-R 图如图 23-9 所示。

23.3.3 数据库连接

在 Android 系统中自带 iSQLite 数据库，这是一个十分小型的数据库，正适合 Android 这种移动平台使用。Android 数据库存储的位置在“data/data/<项目文件夹>/databases/”目录下。

Android 使用 ContentProvider 作为内容提供商，SQLiteOpenHelper 数据库帮助类进行对数据库的创建和操作。通过 Context.getContentResolver()方法直接对数据库进行操作。程序中数据库类为 DBHelper extends SQLiteOpenHelper（继承关系），内容提供类为 DBProvider extends ContentProvider（继承关系）。



▲图 23-9　音乐播放器的 E-R 图

23.3.4　创建数据库

Android 提供了标准的数据库创建方式，继承自 SQLiteOpenHelper，实现 onCreate 和 onUpgrade 两个方法，这样的好处是便于数据库版本的升级。编写文件 DBHelper.java，实现连接数据库的算法，具体代码如下所示。

```
public class DBHelper extends SQLiteOpenHelper{
    /**
     * 数据库名称常量
     */
    private static final String DATABASE_NAME = "MyMusic.db";
    /**
     *  数据库版本常量
     */
    private static final int DATABASE_VERSION = 1;
    /**
     *  表名称常量
     */
    public static final String TABLES_TABLE_NAME = "File_Table";
    private static final String DATABASE_CREATE = "CREATE TABLE " + FileColumn.TABLE +"
("
      + FileColumn.ID+" integer primary key autoincrement,"
      + FileColumn.NAME+" text,"
      + FileColumn.PATH+" text,"
      + FileColumn.SORT+" integer,"
      + FileColumn.TYPE+" text)";
    /**
     *  构造方法
     * @param context
     */
    public DBHelper(Context context) {
        // 创建数据库
        super(context, DATABASE_NAME,null, DATABASE_VERSION);
    }
    /**
     *  创建时调用
     */
    public void onCreate(SQLiteDatabase db) {
        /*Locale l = new Locale("zh", "CN");
        db.setLocale(l);*/
```

```
        db.execSQL(DATABASE_CREATE);
    }
    /**
     *  版本更新时调用
     */
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // 删除表
        db.execSQL("DROP TABLE IF EXISTS File_Table");
        onCreate(db);
    }
}
```

如果创建数据库不成功则抛出 FIleNotFoundException 异常。

23.3.5 操作数据库

Android 对数据库的操作主要有插入、删除、更新、查询操作，在进行任何操作时都必须指定一个 Uri，才能对相应的表数据进行操作。编写文件 DBProvider.java，在里面分别编写数据插入、修改、查询和删除操作的实现方法，具体代码如下所示。

```
public class DBProvider extends ContentProvider {
    private DBHelper dbOpenHelper;
    public static final String AUTHORITY = "MUSIC";
    public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY
            + "/" + FileColumn.TABLE);
    @Override
    public int delete(Uri arg0, String arg1, String[] arg2) {
        SQLiteDatabase db = dbOpenHelper.getWritableDatabase();

        try {
            db.delete(FileColumn.TABLE, arg1, arg2);
            Log.i("info", "delete");
        } catch (Exception ex) {
            ex.printStackTrace();
            Log.e("error", "delete");
        }
        return 1;
    }
    /**
     * 待实现
     */
    @Override
    public String getType(Uri uri) {
        return null;
    }
    /**
     * 插入
     */
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        SQLiteDatabase db = dbOpenHelper.getWritableDatabase();
        long count = 0;
        try {
            count = db.insert(FileColumn.TABLE, null, values);
        } catch (Exception ex) {
            ex.printStackTrace();
            Log.e("error", "insert");
        }
        if (count > 0)
            return uri;
        else
            return null;
    }
    @Override
    public boolean onCreate() {
        dbOpenHelper = new DBHelper(getContext());
        return true;
```

```
    }
    /**
     * 根据条件查询
     * @return 数据集
     */
    @Override
    public int update(Uri uri, ContentValues values, String selection,
            String[] selectionArgs) {
        SQLiteDatabase db = dbOpenHelper.getWritableDatabase();
        int i = 0;
        try {
            i = db.update(FileColumn.TABLE, values, selection, null);
            return i;
        } catch (Exception ex) {
            Log.e("error", "update");
        }
        return 0;
    }
}
```

23.3.6　数据显示

本项目在显示数据时，利用 Cursor 游标类指向数据表中的某一项，然后进行数据查询，并用 Log 日志显示出来。

```
//数据库查询操作
@Override
    public  Cursor  query(Uri  uri,  String[]  projection,  String  selection,String[]
selectionArgs, String sortOrder) {
        SQLiteDatabase db = dbOpenHelper.getWritableDatabase();
// 参数依次为：表名，查询字段，where 语句,替换
//group by(分组), having(分组条件),order by(排序)
        Cursor cur = db.query(FileColumn.TABLE, projection, selection,selectionArgs, null,
null, sortOrder);return cur;}
```

23.4　具体编码

经过前面的内容的讲解，本播放器实例项目的前期工作已经结束。在接下来的内容中，将详细讲解本项目的具体编码过程。

23.4.1　设置服务信息

编写文件 SystemService.java，在此设置项目的服务信息，主要代码如下所示。

```
public class SystemService {
    private Context context;
    private Cursor cursor;
    public SystemService(Context context) {
        this.context = context;
    }

    public Cursor allSongs() {
        if (cursor != null)
            return cursor;
        ContentResolver resolver = context.getContentResolver();
        cursor = resolver.query(MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,
                null, null, null, MediaStore.Audio.Media.DEFAULT_SORT_ORDER);
        return cursor;
    }

    /**
     * 读取正在播放歌曲的艺术家
     * @return
     */
```

```
    public String getArtist() {
        return cursor.getString(cursor
                .getColumnIndexOrThrow(MediaStore.Audio.Media.ARTIST));
    }
    /**
     * 读取正在播放歌曲的名字
     * @return 歌曲名字
     */
    public String getTitle() {
        String title = cursor.getString(cursor
                .getColumnIndexOrThrow(MediaStore.Audio.Media.TITLE));
        try {
            title=EncodingUtils.getString(title.getBytes(), "UTF-8");
        } catch (Exception e) {

            e.printStackTrace();
        }
        return title;
    }

    /**
     *  读取正在播放歌曲的专辑
     * @return 专辑名
     * @throws RemoteException
     */
    public String getAlbum() throws RemoteException {
        return cursor.getString(cursor
                .getColumnIndexOrThrow(MediaStore.Audio.Media.ALBUM));
    }

    /*public int getDuration() throws RemoteException {
        // 获得当前歌曲的时长
        return player.getDuration();
    }public int getTime() throws RemoteException {
        // 获得当前的媒体时间
        return player.getCurrentPosition();
    }
```

23.4.2 播放器主界面

Android 的每一个可视化界面，都有其唯一的布局配置文件。该文件里面有各种布局方式和各种资源文件，如图像、文字、颜色的引用。程序在运行时，可以通过代码对各配置文件进行读取。这样就可以形成不同的可视化界面和　丽的效果。

（1）本实例主界面的布局文件是 main.xml，主要代码如下所示。

```
<TextView
    android:id="@+id/current_music"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:textColor="#ffffff"
    android:padding="10dip"
    android:cacheColorHint="#00000000"
    android:text="this is TextView.."
    android:layout_y="320px"/>
 -->
<Gallery android:id="@+id/gallery"
       android:layout_width="fill_parent"
            android:layout_height="200dp"
            android:layout_alignParentLeft="true"
            android:spacing="16dp"
            android:cacheColorHint="#00000000"
            android:layout_centerVertical="true"
            android:layout_y="20px"/>
<SeekBar android:id="@+id/seekbar" android:layout_width="245px"
        android:layout_height="20px" android:layout_x="40px"
    android:progressDrawable="@drawable/seekbar_style"
```

```
    android:thumb="@drawable/thumb"
       android:paddingLeft="18px"
       android:paddingRight="15px"
       android:paddingTop="5px"
       android:paddingBottom="5px"
       android:progress="0"
android:max="100"
android:secondaryProgress="0"
android:layout_y="350px"/>
<TextView android:layout_x="60px" android:layout_height="wrap_content"
       android:text="00:00"                              android:layout_y="370px"
android:id="@+id/current_time_text"
       android:layout_width="wrap_content"></TextView>
   <TextView android:layout_x="230px" android:layout_height="wrap_content"
       android:text="00:00" android:layout_y="370px" android:id="@+id/end_Time_Text"
       android:layout_width="wrap_content"></TextView>
<LinearLayout android:orientation="horizontal"
        android:gravity="center"
        android:layout_y="423px"
        android:layout_height="wrap_content"
         android:layout_width="fill_parent"
         android:background="@drawable/buttonground" />
<!-- 建立第一个 ImageButton -->
  <ImageButton
  android:id="@+id/btStart"
   android:layout_height="70dp"
  android:layout_width="70dp"
  android:layout_x="145px"
  android:layout_y="390px"
  android:background="#00000000"
  android:src="@drawable/play"
  />
  <!-- 建立第二个 ImageButton  -->
  <ImageButton
  android:id="@+id/pause"
  android:layout_height="wrap_content"
  android:layout_width="wrap_content"
 android:background="#00000000"
  android:layout_x="141px"
  android:layout_y="50px"
  android:src="@drawable/pause"
  />
  <!-- 建立第三个 ImageButton -->
  <ImageButton
  android:id="@+id/before"
  android:layout_height="70dp"
  android:layout_width="70dp"
  android:background="#00000000"
  android:layout_x="80px"
  android:layout_y="280px"
  android:src="@drawable/backward"
  />
  <!-- 建立第四个 ImageButton -->
  <ImageButton
  android:id="@+id/next"
  android:layout_height="70dp"
  android:layout_width="70dp"
  android:background="#00000000"
  android:layout_x="210px"
  android:layout_y="280px"
  android:src="@drawable/forward"
  />
  <!-- 建立第五个 ImageButton -->
  <ImageButton
  android:id="@+id/btStop"
  android:layout_height="70dp"
  android:layout_width="70dp"
  android:layout_x="145px"
  android:layout_y="280px"
```

```
    android:background="#00000000"
  android:src="@drawable/stop"
  />

<ImageButton
  android:id="@+id/listplay"
  android:layout_height="70dp"
  android:layout_width="70dp"
 android:cacheColorHint="#00000000"
  android:layout_x="50px"
  android:layout_y="390px"
   android:background="#00000000"
  android:src="@drawable/itunes2" />
  <!--
  <ImageButton
  android:id="@+id/player"
  android:layout_height="70dp"
  android:layout_width="70dp"
 android:cacheColorHint="#00000000"
  android:layout_x="140px"
  android:layout_y="390px"
   android:background="#00000000"
  android:src="@drawable/wmp2" />
   -->
  <ImageButton
  android:id="@+id/returnBt"
  android:layout_height="70dp"
  android:layout_width="70dp"
 android:cacheColorHint="#00000000"
  android:layout_x="230px"
  android:layout_y="390px"
   android:background="#00000000"
  android:src="@drawable/white"
/>
```

（2）播放器主界面是一个 Activity。Android 工程在每个 activity 启动的时候会首先执行 Oncreate()方法。本实例主界面的程序文件是 MainPlayActivity.java，其具体实现流程如下所示。

- 实现界面初始化工作。如果有播放的歌曲则在播放器中显示歌曲名，并显示上次的播放进度。如果设置了显示歌词，则还会在界面中显示歌词。对应代码如下所示。

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.main);
        systemProvider=new SystemService(this);
        cursor=systemProvider.allSongs();
        SharedPreferences sp = getSharedPreferences("MUSIC",MODE_WORLD_READABLE);
        if (sp != null) {
            playingName = sp.getString("PLAYINGNAME", null);
            selectName = sp.getString("SELECTNAME", null);
            String s = sp.getString("MUSIC_LIST", null);
            if (s != null)
                music_List = StringHelper.spiltString(s);
        }

        init_Play_Rack();// 界面初始化
        if (playingName != null) {//
            int time1 = mplayer.getDuration();
            int time2 = mplayer.getCurrentPosition();
            seekBar.setMax(time1);
            seekBar.setProgress(time2);
            currently_Time.setText(getFileTime(time2));
            end_Time.setText(getFileTime(time1));
            currently_Music.setText(playingName);


            handler.removeCallbacks(thread_One);
```

```
            handler.postDelayed(thread_One, 1000);
            lrc_time = new ArrayList<String>();
            lrc_word = new ArrayList<String>();
            showLrc(playingName);// 歌词显示
        }
        if (selectName != null) {// 播放选中的歌曲
            play_bt.setImageBitmap(musicAdapter.getSuspend_Icon());  //默认暂停图标
            play_Music();
            lrc_time = new ArrayList<String>();
            lrc_word = new ArrayList<String>();
            showLrc(selectName);// 歌词显示
        }
        if (!(currently_Music.getText().toString()).equals("无")) {
            play_bt.setOnTouchListener(playListener);                    //播放监听器
            seekBar.setOnSeekBarChangeListener(seekBarListener);         //音轨监听器
            stop_bt.setOnTouchListener(stopListener);                    //停止监听器
            move_Down.setOnTouchListener(downListener);                  //下一首歌曲监听器
            move_Up.setOnTouchListener(upListener);                      //上一首歌曲监听器
        }
        list_bt.setOnTouchListener(list_bt_listener);                    //清单监听器
        back_bt.setOnTouchListener(return_bt_listener);
        mplayer.setOnCompletionListener(playerListener);                 //监听歌曲是否播放完

        mSwitcher = (ImageSwitcher) findViewById(R.id.switcher);
        mSwitcher.setFactory(this);
        mSwitcher.setInAnimation(AnimationUtils.loadAnimation(this,
                android.R.anim.fade_in));
        mSwitcher.setOutAnimation(AnimationUtils.loadAnimation(this,
                android.R.anim.fade_out));
        mSwitcher.setImageResource(R.drawable.background);
        Gallery g = (Gallery) findViewById(R.id.gallery);
        g.setAdapter(new ImageAdapter(this));
        g.setSelection(200);
        g.setOnItemSelectedListener(this);
    }
```

● 设置暂停和重置处理，对应代码如下所示。

```
protected void onPause() {
        super.onPause();
        SharedPreferences sp = getSharedPreferences("MUSIC",
                MODE_WORLD_WRITEABLE);
        SharedPreferences.Editor editor = sp.edit();
        playingName = currently_Music.getText().toString();
        if (!playingName.equals("无"))
            editor.putString("PLAYINGNAME", playingName);
        editor.putString("SELECTNAME", selectName);
        editor.putString("MUSIC_LIST", StringHelper.toStringAll(music_List));
        editor.commit();
        handler.removeCallbacks(thread_One);
    }
    @Override
    protected void onResume() {
        super.onResume();
        systemProvider=new SystemService(this);
        cursor=systemProvider.allSongs();
        SharedPreferences sp = getSharedPreferences("MUSIC",MODE_WORLD_READABLE);
        if (sp != null) {
            playingName = sp.getString("PLAYINGNAME", null);
            selectName = sp.getString("SELECTNAME", null);
            String s = sp.getString("MUSIC_LIST", null);
            if (s != null)
                music_List = StringHelper.spiltString(s);
        }
        if (mplayer.isPlaying()) {
            handler.removeCallbacks(thread_One);
            handler.postDelayed(thread_One, 1000);
        }
        else
```

```
        handler.removeCallbacks(thread_One);
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.i("info", "onDestroy");
        new File("/data/data/com.Rain.music.activity/shared_prefs/MUSIC.xml")
                .delete();
        new File("/data/data/com.Rain.music.activity/shared_prefs/SET_MSG.xml")
                .delete();
        System.exit(0);
    };
```

- 定义方法 init_Play_Rack()，实现主界面的初始化处理，对应代码如下所示。

```
private void init_Play_Rack() {
    list_bt = (ImageButton) findViewById(R.id.listplay);
    back_bt = (ImageButton) findViewById(R.id.returnBt);
    stop_bt = (ImageButton) findViewById(R.id.btStop);
    play_bt = (ImageButton) findViewById(R.id.btStart);
    move_Up = (ImageButton) findViewById(R.id.before);
    move_Down = (ImageButton) findViewById(R.id.next);
    end_Time = (TextView) findViewById(R.id.end_Time_Text);
    //title_Music = (TextView) findViewById(R.id.title_music);
    currently_Time = (TextView) findViewById(R.id.current_time_text);
    currently_Music = (TextView) findViewById(R.id.current_music);
    seekBar = (SeekBar) findViewById(R.id.seekbar);

    mplayer = MusicHelp.getMediaPlayer();
    musicAdapter = new MusicAdapter(this, music_List);
    handler = MusicHelp.getHandler();
    currently_Music.setText("无");
    currently_Music.setTextColor(Color.WHITE);
    currently_Time.setTextColor(Color.WHITE);
    end_Time.setTextColor(Color.WHITE);
    lrcTime = (TextView) findViewById(R.id.lrcText);
    SharedPreferences sp = getSharedPreferences("SET_MSG",
            MODE_WORLD_READABLE);
    if (sp != null) {
        if (sp.getString("sigle_Play", null) != null) {
            play_Mode = sp.getString("sigle_Play", null);
        }
        if (sp.getString("order_Play", null) != null) {
            play_Mode = sp.getString("order_Play", null);
        }
        if (sp.getString("random_Play", null) != null) {
            play_Mode = sp.getString("random_Play", null);
        }
        if (sp.getString("lyLrc", null) != null) {
            lrc_Show = sp.getString("lyLrc", null);
        }
        Log.i("info", "play_Mode=" + play_Mode);
        Log.i("info", "lrc_Show=" + lrc_Show);
    }
}
```

- 定义方法 onCompletion()，实现歌曲播放完的监听器功能，对应代码如下所示。

```
OnCompletionListener playerListener = new OnCompletionListener() {
    @Override
    public void onCompletion(MediaPlayer mp) {

        play_Mode();// 播放模式
        lrc_time = new ArrayList<String>();
        lrc_word = new ArrayList<String>();
        showLrc(selectName);
    }
};
```

- 定义 return_bt_listener，实现结束监听器处理，对应代码如下所示。

```
    OnTouchListener return_bt_listener = new OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            if (event.getAction() == MotionEvent.ACTION_DOWN) {
                back_bt.setImageResource(R.drawable.whitepress);
            } else if (event.getAction() == MotionEvent.ACTION_UP) {
                back_bt.setImageResource(R.drawable.white);
                finish();
                onDestroy();
//android.os.Process.killProcess(android.os.Process.myPid());
//获取 PID，目前获取自己的也只有该 API
//否则从/proc 中自己枚举其他进程，不过要说明的是，结束其他进程不一定有权限，不然就乱套了
            //    System.exit(0);
            }
            return false;
        }
    };
```

- 定义 list_bt_listener，实现清单监听器的处理，对应代码如下所示。

```
    OnTouchListener list_bt_listener = new OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            if (event.getAction() == MotionEvent.ACTION_DOWN) {
            //    v.setBackgroundResource(R.drawable.share_pressed);
                list_bt.setImageResource(R.drawable.itunespress);
            } else if (event.getAction() == MotionEvent.ACTION_UP) {
                //v.setBackgroundResource(R.drawable.share);
                list_bt.setImageResource(R.drawable.itunes2);

                Intent intent = new Intent(MainPlayActivity.this,
                        PlayListActivity.class);
                startActivityForResult(intent, 0);
            }
            return false;
        }
}
```

- 定义 OnSeekBarChangeListener seekBarListener，实现音轨监听器处理操作，对应代码如下所示。

```
private OnSeekBarChangeListener seekBarListener = new OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int progress,
            boolean fromUser) {
        if (fromUser) {
            mplayer.seekTo(progress);
            currently_Time.setText(getFileTime(progress));
        }
    }
    @Override
    public void onStopTrackingTouch(SeekBar seekBar) {
    }
    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {
    }
};
```

- 定义 playListener，实现播放监听器的处理，对应代码如下所示。

```
OnTouchListener playListener = new OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            if (event.getAction() == MotionEvent.ACTION_DOWN) {
                if (mplayer.isPlaying()) {
```

```
                play_bt.setImageResource(R.drawable.pausepress);
            }
            else {
                play_bt.setImageResource(R.drawable.playpress);
            }
        } else if (event.getAction() == MotionEvent.ACTION_UP) {
            if (mplayer.isPlaying()) {
                mplayer.pause();// 暂停
                currently_Time.setText(currently_Time.getText().toString());
                play_bt.setImageBitmap(musicAdapter.getPlay_Icon());
                handler.removeCallbacks(thread_One);
            }
            else {
                if (is_stopping) {
                    play_Music();
                    is_stopping = false;
                    play_bt.setImageBitmap(musicAdapter.getSuspend_Icon());
                } else {
                    mplayer.start();
                    handler.postDelayed(thread_One, 1000);
                    play_bt.setImageBitmap(musicAdapter.getSuspend_Icon());
                }
            }
        }
        return false;
    }
};
```

- 定义 stopListener，实现停止监听器的处理，对应代码如下所示。

```
OnTouchListener stopListener = new OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            if (event.getAction() == MotionEvent.ACTION_DOWN) {
                stop_bt.setImageResource(R.drawable.stoppress);
            } else if (event.getAction() == MotionEvent.ACTION_UP) {
                stop_bt.setImageResource(R.drawable.stop);
                mplayer.stop();
                currently_Time.setText("00:00");
                end_Time.setText("00:00");
                play_bt.setImageBitmap(musicAdapter.getPlay_Icon());
                handler.removeCallbacks(thread_One);
                seekBar.setProgress(1);
                is_stopping = true;
                lrcTime.setText("无");
            }
            return false;
        }
    };
```

- 定义 downListener，实现下一首歌曲监听器的操作处理，对应代码如下所示。

```
OnTouchListener downListener = new OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if (event.getAction() == MotionEvent.ACTION_DOWN) {
            move_Down.setImageResource(R.drawable.forwardpress);

        } else if (event.getAction() == MotionEvent.ACTION_UP) {
            move_Down.setImageResource(R.drawable.forward);
            move_Down(currently_Music.getText().toString());
            lrc_time = new ArrayList<String>();
            lrc_word = new ArrayList<String>();
            showLrc(currently_Music.getText().toString());// 歌词显示
        }
        return false;
    }
};
```

● 定义 upListener，实现上一首歌曲监听器的操作处理，对应代码如下所示。

```
OnTouchListener upListener = new OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if (event.getAction() == MotionEvent.ACTION_DOWN) {
            move_Up.setImageResource(R.drawable.backwardpress);
        } else if (event.getAction() == MotionEvent.ACTION_UP) {
            move_Up.setImageResource(R.drawable.backward);
            move_Up(currently_Music.getText().toString());
            lrc_time = new ArrayList<String>();
            lrc_word = new ArrayList<String>();
            showLrc(currently_Music.getText().toString());// 歌词显示
        }
        return false;
    }
};
```

● 定义方法 move_Down 和 move_Up，分别用于播放下一首歌曲和上一首歌曲，对应代码如下所示。

```
private void move_Down(String musicName) {
    for (int i = 0; i < music_List.size(); i++) {
        if (musicName.equals(music_List.get(i))) {
            if ((i + 1) < music_List.size()) {
                selectName = music_List.get(i + 1);
                play_Music();
                return;
            } else {
                selectName = music_List.get(0);
                play_Music();
                return;
            }
        }
    }
}

private void move_Up(String musicName) {
    for (int i = 0; i < music_List.size(); i++) {
        if (musicName.equals(music_List.get(i))) {
            if ((i - 1) >= 0) {
                selectName = music_List.get(i - 1);// 移动到上一首歌曲
                play_Music();
                return;
            } else {
                selectName = music_List.get(music_List.size() - 1);
                play_Music();
                return;
            }
        }
    }
}
```

● 定义方法 play_Mode()来设置系统的播放模式，本实例有单曲循环、顺序播放和随机播放 3 种模式，对应代码如下所示。

```
private void play_Mode() {
    if ("is_Sigle".equals(play_Mode)) {// 单曲循环
        play_Music();
    }
    if ("is_Order".equals(play_Mode)) {// 顺序播放
        move_Down(currently_Music.getText().toString());
    }
    if ("is_Random".equals(play_Mode)) {// 随机播放
        Random r = new Random();
        int idx = r.nextInt(music_List.size());// 随机生成 [0,music_List.size())
        // 的 INT 值
```

```
        selectName = music_List.get(idx);
        play_Music();
    }
}
```

- 定义方法 play_Music()来播放指定的音乐文件，对应代码如下所示。

```
private void play_Music() {
    try {

        mplayer.reset();
        mplayer.setDataSource(query());// 文件流中选择歌曲
        mplayer.prepare();
        mplayer.start();
        currently_Music.setText(selectName);
        /*
        if(cursor.moveToFirst()){
            String title=systemProvider.getArtist();
            currently_Music.setText(title);
        }*/

        seekBar.setMax(mplayer.getDuration());// 音频文件持续时间
        seekBar.setProgress(1);
        currently_Time.setText(getFileTime(mplayer.getCurrentPosition()));
    //    lrcTime.setText(systemProvider.getArtist());
        handler.removeCallbacks(thread_One);
        end_Time.setText(getFileTime(mplayer.getDuration()));
        handler.postDelayed(thread_One, 1000);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

- 定义方法 query()来查询歌曲路径，对应代码如下所示。

```
public String query() {
    ContentResolver cr = getContentResolver();
    Uri uri = DBProvider.CONTENT_URI;
    String[] projection = { "path" };
    String selection = "fileName=?";
    String[] selectionArgs = { selectName };
    Cursor c = cr.query(uri, projection, selection, selectionArgs, null);
    if (c.moveToFirst()) {
        String path = c.getString(0);
        return path;
    }
    return null;
}
```

- 定义方法 getFileTime()来获取音乐文件的播放持续时间长的格式化字符串，其返回值是一个格式化时间字符串，对应代码如下所示。

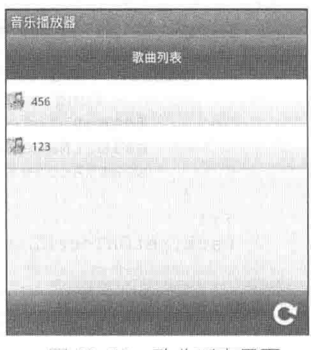
```
private String getFileTime(int timeMs) {
    int totalSeconds = timeMs / 1000;// 获取文件有多少秒
    StringBuilder mFormatBuilder = new StringBuilder();
    Formatter mFormatter = new Formatter(mFormatBuilder, Locale
            .getDefault());
    int seconds = totalSeconds % 60;
    int minutes = (totalSeconds / 60) % 60;
    int hours = totalSeconds / 3600;
    mFormatBuilder.setLength(0);

    if (hours > 0) {
        return mFormatter.format("%d:%02d:%02d", hours, minutes, seconds)
                .toString();// 格式化字符串
    } else {
        return mFormatter.format("%02d:%02d", minutes, seconds).toString();
    }
}
```

另外在文件 MainPlayActivityRoot.java 中，定义了主界面中控制按钮的响应方法，其主要实现代码如下所示。

```
/**
 * 清单按钮
 */
protected ImageButton list_bt;
/**
 * 返回按钮
 */
protected ImageButton back_bt;
/**
 * 停止按钮
 */
protected ImageButton stop_bt;
/**
 * 播放按钮
 */
protected ImageButton play_bt;
/**
 * 上一首歌曲
 */
protected ImageButton move_Up;
/**
 * 下一首歌曲
 */
protected ImageButton move_Down;
/**
 * 歌曲结束时间
 */
protected TextView end_Time;
/**
 * 当前播放时间
 */
protected TextView currently_Time;
/**
 * 当前播放歌曲
 */
protected TextView currently_Music;
/**
 * 音轨
 */
protected SeekBar seekBar;
/**
 * 歌词显示
 */
protected TextView lrcTime;
/**
 * 播放器是否停止
 */
protected boolean is_stopping = false;
/**
 * 系统自带播放器控件
 */
protected MediaPlayer mplayer;
/**
 * 选中的歌曲
 */
protected String selectName;
/**
 * 正在播放的歌曲
 */
protected String playingName;
/**
 *  播放模式 : 默认为随机播放模式
 */
protected String play_Mode = "is_Random";
/**
 * 歌词显示模式
 */
```

```
protected String lrc_Show;
/**
 * 歌曲列表
 */
protected List<String> music_List = new ArrayList<String>();
/**
 * 线程
 */
protected Handler handler;
```

23.4.3 播放列表功能



▲图 23-10 歌曲列表界面

系统的歌曲列表界面如图 23-10 所示。

（1）编写布局文件 play_list.xml，主要代码如下所示。

```
<LinearLayout android:layout_width="fill_parent"
   android:gravity="center" android:layout_height="wrap_content"
   android:background="@drawable/footer_bar">
   <TextView android:text="歌曲列表" android:id="@+id/music_list"
      android:textSize="@dimen/music_list_title" android:textStyle="bold"
      android:layout_width="wrap_content"
android:layout_height="wrap_content"></TextView>

</LinearLayout>

<ListView android:id="@+id/show_play_list"
   android:layout_width="fill_parent" android:layout_height="337px"></ListView>
<LinearLayout android:layout_width="fill_parent"
   android:gravity="right" android:layout_height="wrap_content"
   android:background="@drawable/footer_bar">
   <ImageButton android:id="@+id/back" android:background="@drawable/back"
      android:layout_width="wrap_content"
android:layout_height="wrap_content"></ImageButton>
</LinearLayout>
```

在 Android 系统中有一个名为 ListView 的视图，其特点是有一个 BaseAdapter 的属性，从上到下或从左到右的显示方式。系统默认的方式每一行只显示一个 TextView，本播放列表实现了自定义的方式，用 ListView 的每一行显示一个音乐图片和一个歌曲名字。在此定义了类 MusicAdapter 来继承 BaseAdapter，然后通过算法对这个适配器进行扩展，扩展成为第一行能显示一张图片和一个歌曲名字。由于 BaseAdapter 是一个抽象类，我们需要实现里面的抽象方法 getView()，该方法返回一个 View，即视图。视图可以显示在 Activity 上，所以就看到我们想要的歌曲列表界面。

ListView 同样有一个监听器方法 new onItemClickListener()，我们只要实现这个方法就可以监听鼠标的单击事件，当鼠标单击到每一行时，可以通过方法 ListView.getItemAtPositon(int position)得到该行上的信息。这样就可以通过 Intent 将数据传入到其他的 Activity。本程序的思路是当鼠标单击一行时，会跳转到另一个 Activity 里面，这个 Activity 和歌曲列表类似，也是一个 ListView，该界面将在下一节介绍。

歌曲列表是从播放主界面跳转过来的，能跳到该歌曲列表，前提是数据有歌曲列表的存在。因为每次歌曲列表显示时会查询数据库中的歌曲列表。如果不存在则会提示空列表，选择到 SD 卡中添加歌曲。如图 23-11 所示。

▲图 23-11 空列表时的提示

（2）编写程序文件 PlayListActivity.java。首先定义方法 setListener()来监听用户的选择操作，将用户选择的歌曲进行播放；然后定义方法 query()来查询系统库内的歌曲，如果为空则显示“播放列表为空”；最后定义方法 showDialog()来显示图 23-11 所示的提示框。文件 PlayListActivity.java 的主要代码如下所示。

```
private void setListener(){
    playlist.setOnItemClickListener(new OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> arg0, View arg1, int arg2,
                long arg3) {
            Intent intent = new Intent(PlayListActivity.this, Menu.class);
            selectName = playlist.getItemAtPosition(arg2).toString();
            startActivityForResult(intent, 2);
        }
    });
    back.setOnTouchListener(new OnTouchListener() {

        @Override
        public boolean onTouch(View v, MotionEvent event) {

            if (event.getAction() == MotionEvent.ACTION_DOWN) {
                v.setBackgroundResource(R.drawable.back_pressed);
            } else if (event.getAction() == MotionEvent.ACTION_UP) {
                v.setBackgroundResource(R.drawable.back);
                Intent intent = new Intent();
                intent.setClass(PlayListActivity.this, MainPlayActivity.class);
                setResult(0, intent);
                finish();
            }

            return false;
        }
    });
}

@Override
protected void onPause() {

    super.onPause();
    Log.v("log", "playListActivity is in pause state");
    SharedPreferences sp=getSharedPreferences("MUSIC", MODE_WORLD_WRITEABLE);
    SharedPreferences.Editor editor=sp.edit();


    editor.putString("SELECTNAME", selectName);
    String str=StringHelper.toStringAll(list);
    editor.putString("MUSIC_LIST", str);
    editor.commit();
}
public String[] query() {// 查询数据库
    cr = getContentResolver();
    uri = DBProvider.CONTENT_URI;
    list.clear();
    String[] projection = { "filename", "path" };
    Cursor c = cr.query(uri, projection, null, null, null);
    if (c.getCount() == 0) {
        showDialog("播放列表为空");
    }
    String[] music = new String[c.getCount()];
    if (c.moveToFirst()) {
        for (int i = 0; i < c.getCount(); i++) {
            c.moveToPosition(i);
            String filename = c.getString(0);
            music[i] = filename;
            list.add(filename);
        }
    }
    if (music.length > 0) {
        playlist.setAdapter(new MusicAdapter(this, list));
    }
    return music;
}

private void showDialog(String msg) {
```

```
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setMessage(msg).setCancelable(false).setPositiveButton("从 SD 卡",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int id) {
                    Intent intent = new Intent(PlayListActivity.this,
                            FileExplorerActivity.class);
                //    startActivity(intent);
                    startActivityForResult(intent, 2);
                }
            }).setNegativeButton("取消", new OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            Intent intent = new Intent();
            setResult(0, intent);
            finish();
        }
    });
    AlertDialog alert = builder.create();
    alert.show();
}
```

23.4.4 菜单功能模块

本系统实例的菜单功能界面如图 23-12 所示。

选项
播放
详细
新增
移除
全部移除
设置

▲图 23-12 菜单功能界面

（1）编写布局文件 menu.xml，主要代码如下所示。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/
res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent" android:background="@drawable/list_bg">
    <LinearLayout android:layout_width="fill_parent"
        android:gravity="center" android:layout_height="wrap_content"
        android:background="@drawable/footer_bar">
        <TextView android:text="选项" android:id="@+id/select_item"
            android:textSize="@dimen/music_list_title" android:textStyle="bold"
            android:layout_width="wrap_content"
android:layout_height="wrap_content"></TextView>
    </LinearLayout>
    <ListView android:id="@+id/menu" android:layout_width="wrap_content"
        android:background="@drawable/list_item_bg"
android:layout_height="wrap_content"></ListView>
    <TextView android:layout_width="wrap_content"
        android:layout_height="50px"></TextView>
    <LinearLayout android:layout_width="fill_parent" android:gravity="right"
        android:layout_height="wrap_content"
android:background="@drawable/footer_bar">
        <ImageButton android:id="@+id/back"
             android:background="@drawable/back"
            android:layout_width="wrap_content"
android:layout_height="wrap_content"></ImageButton>
    </LinearLayout>
```

（2）编写文件 menu.java，其具体实现流程如下所示。

- 使用 List<String>容器来保存 String 类型的字符，保存了“播放”、“新增”、“详细”、“移除”、“全部移除”和“设置”等选项。对应代码如下所示。

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.menu);
    menuLV = (ListView) findViewById(R.id.menu);
    back=(ImageButton)findViewById(R.id.back);
    select_Item = (TextView) findViewById(R.id.select_item);
    select_Item.setTextColor(Color.WHITE);
    SharedPreferences sp = getSharedPreferences("MUSIC",
            MODE_WORLD_READABLE);
```

```
    if (sp != null) {

        selectName=sp.getString("SELECTNAME", null);
    }

    List<String> seclect_items = new ArrayList<String>();
    seclect_items.add("播放");
    seclect_items.add("详细");
    seclect_items.add("新增");
    seclect_items.add("移除");
    seclect_items.add("全部移除");
    seclect_items.add("设置");
    menuLV.setAdapter(new MusicAdapter(this, seclect_items));

    back.setOnTouchListener(backListener)
    ;
```

- 使用 case 语句根据用户选择的选项转到对应的界面，对应代码如下所示。

```
    menuLV.setOnItemClickListener(new OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> arg0, View arg1, int arg2,
                long arg3) {
            SharedPreferences sp=getSharedPreferences("MUSIC", MODE_WORLD_WRITEABLE);
            SharedPreferences.Editor editor=sp.edit();
            switch (arg2) {
            case 0:// 播放
                Bundle bundle = new Bundle();
                bundle.putInt("operate", 0);
                Intent intent = new Intent();
                intent.putExtras(bundle);
                setResult(2, intent);
                finish();
                break;
            case 2:
                Intent intent_add = new Intent(Menu.this,
                        FileExplorerActivity.class);
                editor.putString("SELECTNAME", null);
                editor.commit();
            //    startActivity(intent_add);
                startActivityForResult(intent_add, 3);
                break;
            case 3:// 移除
                showDialog(selectName);
                break;
            case 4:// 全部移除
                showDialog("");
                break;
            case 5:// 设置
                Intent intent_set = new Intent(Menu.this, PlaySetting.class);
                editor.putString("SELECTNAME", null);
                editor.commit();
                startActivityForResult(intent_set, 3);
                break;
            default:
                break;
            }
        }
    });
}
```

- 定义方法 showDialog()，用于提示用户确认是否移除或全部移除列表中的音频文件。对应代码如下所示。

```
private void showDialog(final String msg) {
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    if (msg.equals(""))
        builder.setMessage("全部移除?");
```

```
    else
        builder.setMessage("是否移除?");
    builder.setCancelable(false).setPositiveButton("是",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int id) {
                    if (msg.equals(""))
                        del_All();
                    else
                        del_One(selectName);
                    Intent intent = new Intent();
                    setResult(6, intent);
                    finish();
                }
            }).setNegativeButton("否", new OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
        }
    });
    AlertDialog alert = builder.create();
    alert.show();
}
```

● 定义方法 del_One()来删除单首歌曲，定义方法 del_All()来删除全部歌曲。对应代码如下所示。

```
private void del_One(String musicName) {// 删除单首歌曲
    ContentResolver cr = getContentResolver();
    Uri uri = DBProvider.CONTENT_URI;
    String where = "fileName=?";
    String[] selectionArgs = { musicName };
    cr.delete(uri, where, selectionArgs);
}

private void del_All() {// 删除全部歌曲
    ContentResolver cr = getContentResolver();
    Uri uri = DBProvider.CONTENT_URI;
    cr.delete(uri, null, null);
}
```

23.4.5 播放设置界面

本系统的播放设置界面如图 23-13 所示。

▲图 23-13 播放设置界面

（1）编写布局文件 setting.xml，主要代码如下所示。

```
<LinearLayout android:layout_width="fill_parent"
    android:gravity="center" android:layout_height="wrap_content"
    android:background="@drawable/footer_bar">
    <TextView android:text="设定" android:id="@+id/setting"
        android:textSize="@dimen/music_list_title" android:layout_width="wrap_content"
        android:layout_height="wrap_content"></TextView>
</LinearLayout>
<LinearLayout android:orientation="horizontal"
    android:layout_width="wrap_content" android:gravity="center_vertical"
    android:layout_height="wrap_content">
    <TextView android:text="播放模式" android:id="@+id/setting"
        android:textSize="@dimen/text_size" android:layout_width="fill_parent"
        android:layout_height="wrap_content"></TextView>
    <RadioGroup android:id="@+id/RadioGroup"
        android:layout_width="wrap_content" android:layout_height="wrap_content">
        <RadioButton android:text="单曲循环" android:id="@+id/sigle_play"
            android:layout_width="wrap_content"
android:layout_height="wrap_content"></RadioButton>
        <RadioButton android:text="顺序播放" android:id="@+id/order_play"
            android:layout_width="wrap_content"
android:layout_height="wrap_content"></RadioButton>
        <RadioButton android:text="随机播放" android:id="@+id/random_play"
            android:checked="true" android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"></RadioButton>
      </RadioGroup>
   </LinearLayout>
   <LinearLayout android:orientation="horizontal"
      android:layout_width="wrap_content" android:gravity="center_vertical"
      android:layout_height="wrap_content">
      <TextView android:text="歌词显示" android:id="@+id/setting"
         android:textSize="@dimen/text_size" android:layout_width="wrap_content"
         android:layout_height="wrap_content" />
      <ToggleButton android:text="" android:id="@+id/ly_lrc"
         android:checked="false" android:layout_width="wrap_content"
         android:layout_height="wrap_content" />
   </LinearLayout>
   <TextView android:layout_width="fill_parent"
      android:layout_height="150px"></TextView>
   <AbsoluteLayout android:background="@drawable/footer_bar"
      android:layout_width="fill_parent" android:layout_height="wrap_content">
      <ImageButton android:id="@+id/make" android:background="@drawable/share"
         android:layout_x="270px" android:layout_width="wrap_content"
         android:layout_height="wrap_content" />
      <ImageButton android:id="@+id/cancel" android:layout_x="5px"
         android:background="@drawable/back" android:layout_width="wrap_content"
         android:layout_height="wrap_content" />
   </AbsoluteLayout>
```

（2）编写程序文件 PlaySetting.java，其主要功能如下所示。

- 设置播放模式。

在设置播放模式时用的是 RadioGroup 组件，这个组件有单项选择的功能，里面有 RadioButton 项，多个 RadioButton 项只能同时选中一个。该播放器播放模式有单曲循环、随机播放、顺序播放等功能。MediaPlayer 有一个监听器，它监听着歌曲是否正在播放或者是否播放完成。当歌曲播放完成时，会触发方法 OnCompletionListener()，该方法可以处理歌曲播放完成后的操作。RadioGroup 可以进行单项选择操作。

- 歌词设置。

歌词是否显示是由一个开关按钮 ToggleButton 实现的，有 ON 和 OFF 状态，当为 ON 时显示歌词，为 OFF 时关闭歌词。

ToogleButton 同样也有一个监听器，可以获得 ToogleButton 的不同状态。使用前对它进行实例化：（ToggleButton）View.findViewById（R.id.ly_lrc），并且用 ToggleButton.isChecked()方法获得开关状态。播放模式状态和歌词显示状态的操作结果都将以一个标志被写在一个配置文件中，这是关于 Android 的存储方式。

了解文件 PlaySetting.java 的实现原理和功能后，其实现代码就非常容易理解了，主要代码如下所示。

```
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.setting);
      set_textView = (TextView) findViewById(R.id.setting);
      set_textView.setTextColor(Color.WHITE);
      sigle_Play = (RadioButton) findViewById(R.id.sigle_play);
      order_Play = (RadioButton) findViewById(R.id.order_play);
      random_Play = (RadioButton) findViewById(R.id.random_play);
      lyLrc = (ToggleButton) findViewById(R.id.ly_lrc);
      set_bt = (ImageButton) findViewById(R.id.make);
      cancel_bt = (ImageButton) findViewById(R.id.cancel);
      set_bt.setOnTouchListener(setting_bt_Listener);
      cancel_bt.setOnTouchListener(cancel_bt_Listener);
   }

   OnTouchListener setting_bt_Listener = new OnTouchListener() {// 设置确定按钮监听器
      @Override
      public boolean onTouch(View v, MotionEvent event) {
```

```
        if (event.getAction() == MotionEvent.ACTION_DOWN) {
            v.setBackgroundResource(R.drawable.share_pressed);
            SharedPreferences sp = getSharedPreferences("SET_MSG",
                    MODE_WORLD_WRITEABLE);// 文件共享
            SharedPreferences.Editor editor = sp.edit();
            if (sigle_Play.isChecked()) {
                editor.putString("sigle_Play", "is_Sigle");
                editor.putString("order_Play", null);
                editor.putString("random_Play", null);
            }
            if (order_Play.isChecked()) {
                editor.putString("sigle_Play", null);
                editor.putString("order_Play", "is_Order");
                editor.putString("random_Play", null);
            }
            if (random_Play.isChecked()) {
                editor.putString("sigle_Play", null);
                editor.putString("order_Play", null);
                editor.putString("random_Play", "is_Random");
            }
            if (lyLrc.isChecked()) {
                editor.putString("lyLrc", "is_Show");
            }
            if (!lyLrc.isChecked()) {
                editor.putString("lyLrc", null);
            }
            editor.commit();// 提交
            Intent intent = new Intent();
            setResult(4, intent);
        finish();
        } else if (event.getAction() == MotionEvent.ACTION_UP) {
            v.setBackgroundResource(R.drawable.share);
        }
        return false;
    }
};

OnTouchListener cancel_bt_Listener = new OnTouchListener() {// 取消监听器
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if (event.getAction() == MotionEvent.ACTION_DOWN) {
            v.setBackgroundResource(R.drawable.back_pressed);
        } else if (event.getAction() == MotionEvent.ACTION_UP) {
            v.setBackgroundResource(R.drawable.back);
            Intent intent = new Intent();
            setResult(3, intent);
            finish();
        }
        return false;
    }
};
```

23.4.6 设置显示歌词

显示歌词功能比较重要，所以笔者决定单独讲解其实现原理。本播放器的歌词是“.lrc”格式文件，查看“.Lrc”文件中的歌词格式如下所示。

```
[00:16.18]我爱你 心爱的姑娘
```

歌词格式是以“时间+歌词”的格式存储。

接下来将介绍如何将.Lrc 中的歌词读取出来，并存储在 Android 的配置文件中。

（1）用 XML 配置文件的存储。

在 Android 系统中，SD 卡的目录结构如图 23-14 所示。

▲图 23-14 SD 卡的目录结构

从图中可以看到一个名为“sdcard”的目录，该目录即为扩展卡目录，里面预先存放着音频文件和“.lrc”歌词文件。我们定义如下代码来指定.lrc 文件存在的路径，并将文件读取到 BufferReader 中。

```
BufferedReader buffer=new BufferedReader(new FileReader(new File("/sdcard/"+ musicName
+ ".lrc")));
```

由于我们要分别存放时间和歌词，所以应该定义两个 List<String>容器来存放时间和歌词。在读取 lrc 时，每次读取一行，再用算法将时间和歌词分开后放到一个数组里面，并分别存放在两个 list 中。由于歌曲在播放时会存在界面之间的跳转，所以歌词必须固定存放在一个文件中，而不能作为一个对象。因此，我们将两个时间 List 和歌词 List 再写进一个配置文件中。

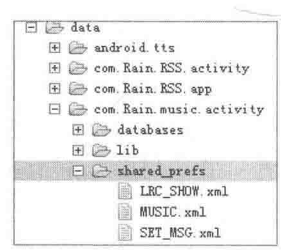
Android 为我们提供了共享文件类 SharedPreferences，它有一个方法 getSharedPreferences(参数 1，参数 2)，参数 1 表示写进时的标记，便于读取；参数 2 表示读取模式，有只写模式（MODE_WORLD_WRITEABLE）和只读模式（MODE_WORLD_READABLE），在写之前将其设置为编辑状态，下面是使用静态方法设置编辑状态的代码。

```
SharedPreferences.Editor editor = sp.edit();
```

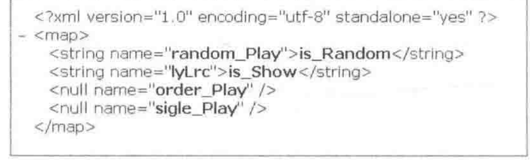
然后在对象 editor 中可以存入一个 HashMap<key, values>类型的键值，其格式是 putString（KEY, VALUES），这样就可以将 List 中的对象转化成一样长的字符中放进配置文件中。

当写入成功时，Android 系统会自动在目录“data/com.Rain.music.activity/shared_prefs/中”生成一个配置文件。如图 23-15 所示。

打开播放模式的 XML 配置文件，在此文件中是以 map 的形式存储的。键名是<string name="random_Play"></string>，其值是 is_Radom。如图 23-16 所示。



▲图 23-15　配置文件



▲图 23-16　XML 配置文件

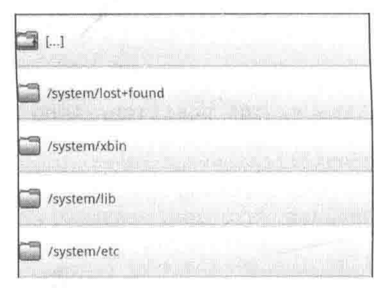
（2）读取 XML 配置文件。

我们仍以播放模式读取为例。当需要确定播放模式时将读取.xml 文件，同样用共享文件类 SharedPreferences，通过方法 getSharedPreferences（"SET_MSG",MODE_WORLD_READABLE）并且是只读方式获得.xml 的文件内容。SharedPreferences 的对象调用方法 getString("sigle_Play", null)，此方法会返回一个 String 类型的值，即我们以前存储的 String 值。当该标记不存在时，会默认返回一个 null 值。读取 XML 配置文件成功后，就可以根据配置的值对程序进行操作了。

23.4.7　文件浏览器模块

本项目程序实现了文件浏览器的功能，作为一个文件浏览器应该具有浏览功能。当程序运行到浏览界面时，会显示各文件的目录及图标。从文件浏览器中我们能看到各文件，而且能对其进行操作。本程序是专为播放器添加歌曲而设计的，因此功能仅限于对媒体文件的浏览，以及对含有媒体文件的目录的浏览，所以功能比较局限。

当显示菜单界面时，通过新增选项进入文件浏览器，或者当播放列表为空时，会提示进入文件浏览器进行歌曲新增操作。其中文件浏览器界面如图 23-17 所示，SD 卡目录界面如图 23-18 所示。



▲图 23-17　文件浏览器界面

▲图 23-18　SD 卡目录界面

文件浏览器界面布局格式类似前面介绍的菜单，只是在界面的第一行新增了一个返回根目录的功能。由于程序只关系到“目录/sdcard”下的文件，所以用程序屏　了其他的目录，这里只显示两个目录“/sdcard”和“/system”。播放器只需要用到媒体文件，所以代码也屏　了其他文件的子目录。当选中“sdcard”会进入到图 23-18 所示的目录，该目录下只显示媒体文件，如.Mp3 和 SD 卡下的子目录。选中“system”会进入到图 23-17 所示的目录，该目录下显示“system”下的各级子目录。当有媒体文件时才会出现添加 Dialog。

当要添加选中的歌曲时，程序有自动判断功能，首先弹出 Dialog。单击“确定”按钮后，程序会查询数据库中的歌曲，调用方法 query(fileName)，根据歌曲名字查询。如果歌曲名字不存在，则调用方法 insertMusic(file)；如果该歌曲名字已经存在，则弹出 Dialog。

（1）编写文件 directory_list.xml，实现文件浏览界面的布局，主要代码如下所示。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent" android:background="@drawable/list_bg">
    <LinearLayout android:layout_width="fill_parent"
        android:gravity="center" android:layout_height="wrap_content"
        android:background="@drawable/footer_bar">
        <TextView android:text="SD 卡" android:id="@+id/store_card" android:textStyle
="bold"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:textSize="@dimen/music_list_title"></TextView>
    </LinearLayout>

    <ListView android:id="@id/android:list" android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
        <TextView  android:layout_width="10px"
        android:layout_height="wrap_content" />
    <TextView android:id="@id/android:empty" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="@string/no_files"
/>
```

（2）编写文件 FileExplorerActivity.java。在此定义了文件浏览器类 FileExplorerActivity，此类继承了 ListActivity，此 Activity 是一个 ListView 界面。整个界面是一个 ListView 布局，而每一行是一个 LinearLayout 水平方式布局，上面将放置一个图片和一个文件全路径。该文件全路径被存放到数据库中，以便歌曲播放时能查询到歌曲路径源。文件 FileExplorerActivity.java 的主要代码如下所示。

```
public class FileExplorerActivity extends ListActivity {
    private List<String> items = null;
    private TextView store_Card;
```

```
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.directory_list);
        store_Card = (TextView) findViewById(R.id.store_card);
        store_Card.setTextColor(Color.WHITE);
        fill(new File("/").listFiles());
    }

    @Override
    protected void onListItemClick(ListView l, View v, int position, long id) {
        int selectionRowID = (int) position;
        File file = new File(items.get(selectionRowID));
        if (selectionRowID == 0) {
            fillWithRoot();
        } else {
            if (file.isDirectory()) {
                fill(file.listFiles());
            } else {
                Intent intent = this.getIntent();
                intent.putExtra("filePath", file);
                FileExplorerActivity.this.setResult(0, intent);
                showDialog("加入播放列表?", file);
            }
        }
    }

    private void fillWithRoot() {
        fill(new File("/").listFiles());
    }

    private void fill(File[] files) {
        items = new ArrayList<String>();
        items.add(getString(R.string.to_top));
        for (File file : files) {
            if (file.isDirectory()) {
                if ((file.getPath().indexOf("/sdcard")) != -1
                        || (file.getPath().indexOf("/system")) != -1)
                    items.add(file.getPath());
            }
            if ((file.getPath().indexOf(".mp3")) != -1) {
                items.add(file.getPath());
            }
        }
        setListAdapter(new MusicAdapter(this, items));
    }

    private void showDialog(String msg, final File file) {
        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setMessage(msg).setCancelable(false).setPositiveButton("确定",
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        String fileName = file.getName().substring(0,
                                file.getName().indexOf("."));
                        Log.i("info", fileName);
                        if (query(fileName)) {
                            insertMusic(file);// 添加音乐
                        }
                    }
                }).setNegativeButton("取消", null);
        AlertDialog alert = builder.create();
        alert.show();
    }

    // 添加音乐到播放列表
    private final void insertMusic(File file) {
        ContentResolver cr = getContentResolver();
        ContentValues values = new ContentValues();
        Uri uri = DBProvider.CONTENT_URI;
```

```
        String fileName = file.getName().substring(0,
                file.getName().indexOf("."));
        values.put(FileColumn.NAME, fileName);
        values.put(FileColumn.PATH, file.getAbsolutePath());
        values.put(FileColumn.TYPE, "Music");
        values.put(FileColumn.SORT, "popular");
        cr.insert(uri, values);
        Toast.makeText(FileExplorerActivity.this, "已加入", Toast.LENGTH_LONG)
                .show();
        Intent intent = new Intent();
        setResult(6, intent);
        finish();
    }

    private boolean query(String name) {
        ContentResolver cr = getContentResolver();
        Uri uri = DBProvider.CONTENT_URI;
        String[] projection = { "filename" };
        Cursor c = cr.query(uri, projection, null, null, null);
        if (c.moveToFirst()) {
            for (int i = 0; i < c.getCount(); i++) {
                c.moveToPosition(i);
                String filename_DB = c.getString(0);
                if (name.equals(filename_DB)) {// 判断播放列表中是否存在该歌曲
                    AlertDialog.Builder builder = new AlertDialog.Builder(this);
                    builder.setMessage("文件已存在").setCancelable(false)
                            .setPositiveButton("返回列表",
                                    new DialogInterface.OnClickListener() {
                                        public void onClick(
                                                DialogInterface dialog, int id) {
                                            Intent intent = new Intent(
                                                    FileExplorerActivity.this,
                                                    PlayListActivity.class);
                                            startActivity(intent);
                                        }
                                    }).setNegativeButton("重新添加", null);
                    AlertDialog alert = builder.create();
                    alert.show();
                    return false;
                }
            }
        }
        return true;
    }
```

上述 ListView 实现了自动判断的功能，即程序可以通过访问扩展卡中的文件属性而自动识别文件属性。当为一个 MP3 格式文件时，则前面图标显示 MP3 图标；当为一个文件目录时，则图标标识为一个文件。文件浏览器是用递归算法实现的，其中 fillWithRoot()方法用于返回根目录的列表。

23.4.8 数据存储

在播放器正常运行时，由于各界面存在相互跳转，为了避免数据在界面跳转的过程中丢失，我们需要将一些数据进行临时存储或者永久存储。

Android 作为一种手机操作系统，提供了 Preference（配置）、File（文件）、SQLite 数据和网络等存取数据的方式。

另外，在 Android 系统中各个应用程序组件之间是相互独立的， 此的数据不能共享。Android 系统提供了 Content Provider 组件来实现应用程序之间数据的共享。

（1）SharedPreferences。

Shared Preference 提供了一种轻量级的数据存取方法，一般数据比较少，如一些简单的配置信息。它以“键-值”（是一个 map）对的方式，将数据保存在一个 XML 配置文件中。

在本实例中用到了如下两种数据存储接口。

- android.content.SharedPreferences：提供了保存数据的方法。
- android.content.SharedPreferences.Editor：提供了获得数据的方法。

注意　有关上述 SharedPreferences 数据存储的知识，请读者参阅相关资料，这并不是本书的重点。

以播放器中的播放模式存取为例，实现流程如下所示。

- XML 配置文件的读取。

仍以播放模式读取为例，当需要确定播放模式时，我们将读取.xml 文件，同样用共享文件类 SharedPreferences，通过方法 getSharedPreferences("SET_MSG",MODE_WORLD_READABLE)并且是只读方式获得 XML 的文件内容。SharedPreferences 的对象调用方法 getString("sigle_Play", null)，其返回一个 String 类型的值，即我们以前存储的 String 值。此方法当该标记不存在时会默认返回一个 null 值。获得成功后我们就可以运用当前的值再对程序进行操作了。

- XML 配置文件的存储。

在类 SharedPreferences 中有一个方法 getSharedPreferences（参数 1，参数 2），参数 1 为写进时的标记，便于读取；参数 2 为读取模式，有只写模式（MODE_WORLD_WRITEABLE）和只读模式（MODE_WORLD_READABLE）。在写之前将其置入编辑状态，用静态方法 SharedPreferences.Editor editor = sp.edit()，然后对象 editor 可以存入一个 HashMap<key,values>类型的键值，即 putString（KEY, VALUES）。这样，我们可以将 List 中的对象转化成一样长的字符中放进配置文件。当写入成功时，Android 系统会自动在目录“data/工程包名/shared_prefs”下生成一个配置文件

（2）File 存储方式。

我们可以将一些数据直接以文件的形式保存在设备中。例如，一些文本文件、PDF 文件、音视频文件和图片等。Android 提供了如下文件读写的方法。

- Context.openFileInput()：获得标准 Java 文件输入流（FileInputStream）；
- Context.openFileOutput()：获得标准 Java 文件输出流（FileOutputStream）；
- Resources.openRawResource (R.raw.myDataFile)：返回 InputStream。

（3）SQLiteDatabase 数据库。

SQLite 是一个嵌入式数据库引擎，是针对内存等资源有限的设备（如手机、PDA、MP3）提供的一种高效的数据库引擎。SQLite 数据库不像其他的数据库（如 Oracle），它没有服务器进程，所有的内容包含在同一个单文件中。该文件是跨平台的，可以自由拷贝。基于其自身的先天优势，SQLite 在嵌入式领域得到了广泛应用。

- SQLiteDatabase 类：代表一个数据库对象，在里面提供了操作数据库的一些方法，具体方法请读者参考相关书　。
- SQLiteOpenHelper 类：是 SQLiteDatabase 的一个帮助类，用来管理数据库的创建和版本更新。一般的用法是定义一个类继承之，并实现其两个抽象方法 onCreate（SQLiteDatabase db）和 onUpgrade（SQLiteDatabase db, int oldVersion, int newVersion），以创建和更新数据库。

Android 系统的 3 种数据存储方式让我们可以轻松、方便地进行程序编写和数据访问，不会让不该消失的数据消失，这对我们进行程序书写有很大帮助。

第 24 章　开发一个闹钟系统

当今社会的生活节奏越来越快，硬件移动设备越来越先进，人们对时间的观念也越来越强。Android 作为一个开源的智能手机系统，具备闹钟功能。和传统的闹钟相比，Andorid 手机闹钟更加灵活、方便、准时。在本章的内容中，将详细讲解在 Android 系统中开发闹钟系统的方法，了解大型闹钟系统的具体开发流程。

24.1 项目介绍

本章播放器源码保存在本书附带源程序中的“\daima\24”目录下。在讲解具体编码之前，先简要介绍本项目的产生背景和项目目的，为后面的具体编码打好理论基础。

24.1.1　系统需求分析

随着社会生活的节奏越来越快，人们对时间观念的精确性要求也越来越高。在高节奏的生活中，闹钟是必不可少的。Android 作为一款强大的智能手机操作系统，当然内置了闹钟功能。但是为了满足人们的更高要求，开发一个个性化的、功能更强大的闹钟系统势在必行。

根据市场调研分析，一个典型的闹钟系统应该具备如下所示的功能。

（1）添加闹钟。

向系统中添加新的闹钟，设置什么时候闹钟会提醒我们，时间精确到几点几分。

（2）管理闹钟。

为了提高设置闹钟的效率，可以在原有已经设置的闹钟的基础上管理闹钟。具体来说，可以修改闹钟的时间和其他属性（如振动、铃声、重复次数）。

（3）删除闹钟。

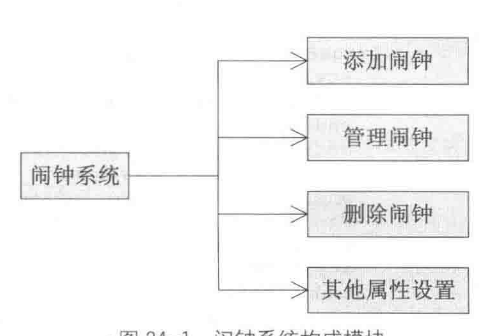
对于系统中不需要的闹钟，可以及时删除，节约系统空间。

（4）设置其他属性。

在设置一个闹钟时，除了设置一个具体时间外，还可以设置重复次数、闹钟开关、在哪一天重复、铃声、振动和标记。

24.1.2　构成模块

根据前面的系统需求分析，可以规划本系统的构成模块，如图 24-1 所示。



▲图 24-1　闹钟系统构成模块

24.2 系统主界面

本系统的主界面比较简单，执行后会在屏幕中间显示当前的时间，并显示最近的闹钟和天气情况，在屏幕下方显示一个操作导航。系统主界面效果如图 24-2 所示。

▲图 24-2　系统主界面效果

在本节的内容中，将详细讲解主界面的具体实现过程。

24.2.1　布局文件

主界面的核心布局功能是通过文件 desk_clock_buttons.xml 实现的，主要实现代码如下所示。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="horizontal"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:layout_weight="0"
   >
   <ImageButton android:id="@+id/alarm_button"
      style="@style/ButtonStripLeft"
      android:layout_height="wrap_content"
      android:layout_width="fill_parent"
      android:layout_weight=".25"
      android:src="@drawable/ic_clock_strip_alarm"
      android:contentDescription="@string/alarm_button_description"
      />
   <ImageButton android:id="@+id/gallery_button"
      style="@style/ButtonStripMiddle"
      android:layout_height="wrap_content"
      android:layout_width="fill_parent"
      android:layout_weight=".25"
      android:src="@drawable/ic_clock_strip_gallery"
      android:contentDescription="@string/gallery_button_description"
      />
   <ImageButton android:id="@+id/music_button"
      style="@style/ButtonStripMiddle"
      android:layout_height="wrap_content"
      android:layout_width="fill_parent"
      android:layout_weight=".25"
      android:src="@drawable/ic_clock_strip_music"
      android:contentDescription="@string/music_button_description"
```

```
        />
    <ImageButton android:id="@+id/home_button"
        style="@style/ButtonStripRight"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent"
        android:layout_weight=".25"
        android:src="@drawable/ic_clock_strip_home"
        android:contentDescription="@string/home_button_description"
        />
</LinearLayout>
```

执行代码后，展示在用户面前的便是图 24-2 所示的界面。

24.2.2 程序文件

系统主界面的程序文件是 DeskClock，具体实现流程如下所示。

（1）定义类 DeskClock，并定义所需常量，具体实现代码如下所示。

```
public class DeskClock extends Activity {
    private static final boolean DEBUG = false;

    private static final String LOG_TAG = "DeskClock";

    //午夜闹钟动作（可以更新日期显示）
    private static final String ACTION_MIDNIGHT = "com.android.superdeskclock.MIDNIGHT";

    // 设置天气和时间的间隔
    private final long QUERY_WEATHER_DELAY = 60 * 60 * 1000; // 1 hr

    //为闹钟设置广播
    private static final String DOCK_SETTINGS_ACTION="com.android.settings.DOCK_SETTINGS";

    // 设置闹钟延迟设置 5 分钟
    private final long SCREEN_SAVER_TIMEOUT = 5 * 60 * 1000; // 5 min

    //设置屏幕保护程序复位延迟
    private final long SCREEN_SAVER_MOVE_DELAY = 60 * 1000; // 1 min

    //设置在屏幕保护模式时的文本和图形的颜色
    private final int SCREEN_SAVER_COLOR = 0xFF308030;
    private final int SCREEN_SAVER_COLOR_DIM = 0xFF183018;

    // 设置时钟显示壁纸与黑色图层之间的不透明度
    private final float DIM_BEHIND_AMOUNT_NORMAL = 0.4f;
    // 高对比变暗
    private final float DIM_BEHIND_AMOUNT_DIMMED = 0.8f;

    //内部消息 ID
    private final int QUERY_WEATHER_DATA_MSG     = 0x1000;
    private final int UPDATE_WEATHER_DISPLAY_MSG = 0x1001;
    private final int SCREEN_SAVER_TIMEOUT_MSG   = 0x2000;
    private final int SCREEN_SAVER_MOVE_MSG      = 0x2001;

    //天气预报查询信息设置
    private  static  final  String  GENIE_PACKAGE_ID="com.google.android.apps.genie.
geniewidget";
    private  static  final  String  WEATHER_CONTENT_AUTHORITY  =  GENIE_PACKAGE_ID  +
".weather";
    private static final String WEATHER_CONTENT_PATH = "/weather/current";
    private static final String[] WEATHER_CONTENT_COLUMNS = new String[] {
            "location",
            "timestamp",
            "temperature",
            "highTemperature",
            "lowTemperature",
            "iconUrl",
            "iconResId",
```

```
        "description",
    };
```

（2）定义函数 moveScreenSaverTo，功能是移动系统的屏幕保护程序，其中参数 x 和 y 表示位置坐标。函数 moveScreenSaverTo 的具体实现代码如下所示。

```
private void moveScreenSaverTo(int x, int y) {
    if (!mScreenSaverMode) return;

    final View saver_view = findViewById(R.id.saver_view);

    DisplayMetrics metrics = new DisplayMetrics();
    getWindowManager().getDefaultDisplay().getMetrics(metrics);

    if (x < 0 || y < 0) {
        int myWidth = saver_view.getMeasuredWidth();
        int myHeight = saver_view.getMeasuredHeight();
        x = (int)(mRNG.nextFloat()*(metrics.widthPixels - myWidth));
        y = (int)(mRNG.nextFloat()*(metrics.heightPixels - myHeight));
    }

    if (DEBUG) Log.d(LOG_TAG, String.format("screen saver: %d: jumping to (%d,%d)",
            System.currentTimeMillis(), x, y));

    saver_view.setLayoutParams(new AbsoluteLayout.LayoutParams(
        ViewGroup.LayoutParams.WRAP_CONTENT,
        ViewGroup.LayoutParams.WRAP_CONTENT,
        x,
        y));

    // Synchronize our jumping so that it happens exactly on the second
    mHandy.sendEmptyMessageDelayed(SCREEN_SAVER_MOVE_MSG,
        SCREEN_SAVER_MOVE_DELAY +
        (1000 - (System.currentTimeMillis() % 1000)));
}
```

（3）定义函数 setWakeLock，功能是设置一个唤醒锁，到闹钟时间时会唤醒。函数 setWakeLock 的具体实现代码如下所示。

```
private void setWakeLock(boolean hold) {
    if (DEBUG) Log.d(LOG_TAG, (hold ? "hold" : " releas") + "ing wake lock");
    Window win = getWindow();
    WindowManager.LayoutParams winParams = win.getAttributes();
    winParams.flags |= (WindowManager.LayoutParams.FLAG_DISMISS_KEYGUARD
            | WindowManager.LayoutParams.FLAG_SHOW_WHEN_LOCKED
            | WindowManager.LayoutParams.FLAG_ALLOW_LOCK_WHILE_SCREEN_ON
            | WindowManager.LayoutParams.FLAG_TURN_SCREEN_ON);
    if (hold)
        winParams.flags |= WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON;
    else
        winParams.flags &= (~WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    win.setAttributes(winParams);
}
```

（4）定义函数 scheduleScreenSaver，功能是设置屏幕保护程序的时间表，即何时进入屏保状态。定义函数 restoreScreen，功能是恢复到正常的屏幕模式。定义函数 saveScreen，用于保存当前的屏幕状态，此函数能够处理 OLED 模式的屏幕保护程序。上述 3 个函数的具体实现代码如下所示。

```
private void scheduleScreenSaver() {
    // reschedule screen saver
    mHandy.removeMessages(SCREEN_SAVER_TIMEOUT_MSG);
    mHandy.sendMessageDelayed(
        Message.obtain(mHandy, SCREEN_SAVER_TIMEOUT_MSG),
        SCREEN_SAVER_TIMEOUT);
}
```

```
private void restoreScreen() {
    if (!mScreenSaverMode) return;
    if (DEBUG) Log.d(LOG_TAG, "restoreScreen");
    mScreenSaverMode = false;
    initViews();
    doDim(false); // restores previous dim mode
    // policy: update weather info when returning from screen saver
    if (mPluggedIn) requestWeatherDataFetch();

    scheduleScreenSaver();

    refreshAll();
}
private void saveScreen() {
    if (mScreenSaverMode) return;
    if (DEBUG) Log.d(LOG_TAG, "saveScreen");

    //快藏起来的 X/Y 的当前日期
    final View oldTimeDate = findViewById(R.id.time_date);
    int oldLoc[] = new int[2];
    oldTimeDate.getLocationOnScreen(oldLoc);

    mScreenSaverMode = true;
    Window win = getWindow();
    WindowManager.LayoutParams winParams = win.getAttributes();
    winParams.flags |= WindowManager.LayoutParams.FLAG_FULLSCREEN;
    win.setAttributes(winParams);

    //在切换布局屏幕时放弃任何内部聚焦
    final View focused = getCurrentFocus();
    if (focused != null) focused.clearFocus();

    setContentView(R.layout.desk_clock_saver);

    mTime = (DigitalClock) findViewById(R.id.time);
    mDate = (TextView) findViewById(R.id.date);
    mNextAlarm = (TextView) findViewById(R.id.nextAlarm);

    final int color = mDimmed ? SCREEN_SAVER_COLOR_DIM : SCREEN_SAVER_COLOR;

    ((TextView)findViewById(R.id.timeDisplay)).setTextColor(color);
    ((TextView)findViewById(R.id.am_pm)).setTextColor(color);
    mDate.setTextColor(color);
    mNextAlarm.setTextColor(color);
    mNextAlarm.setCompoundDrawablesWithIntrinsicBounds(
        getResources().getDrawable(mDimmed
            ? R.drawable.ic_lock_idle_alarm_saver_dim
            : R.drawable.ic_lock_idle_alarm_saver),
        null, null, null);

    mBatteryDisplay =
    mWeatherCurrentTemperature =
    mWeatherHighTemperature =
    mWeatherLowTemperature =
    mWeatherLocation = null;
    mWeatherIcon = null;

    refreshDate();
    refreshAlarm();

    moveScreenSaverTo(oldLoc[0], oldLoc[1]);
}
```

（5）在系统的主界面中显示了当日的天气情况，此功能通过如下所示的函数实现。

- 函数 requestWeatherDataFetch：功能是获取天气数据。
- 函数 scheduleWeatherQueryDelayed：功能是处理天气查询延迟时间。
- 函数 queryWeatherData：功能是查询天气数据。

- 函数 updateWeatherDisplay：功能是在界面中更新天气显示。

上述函数的具体实现代码如下所示。

```
//告诉 widget 部件从网络载入新的数据
private void requestWeatherDataFetch() {
    if (DEBUG) Log.d(LOG_TAG, "forcing the Genie widget to update weather now...");
    sendBroadcast(new Intent(ACTION_GENIE_REFRESH).putExtra("requestWeather", true));
    // we expect the result to show up in our content observer
}

private boolean supportsWeather() {
    return (mGenieResources != null);
}

private void scheduleWeatherQueryDelayed(long delay) {
    // cancel any existing scheduled queries
    unscheduleWeatherQuery();
    if (DEBUG) Log.d(LOG_TAG, "scheduling weather fetch message for " + delay + "ms from
    now");
    mHandy.sendEmptyMessageDelayed(QUERY_WEATHER_DATA_MSG, delay);
}
private void unscheduleWeatherQuery() {
    mHandy.removeMessages(QUERY_WEATHER_DATA_MSG);
}

private void queryWeatherData() {
    // if we couldn't load the weather widget's resources, we simply
    // assume it's not present on the device
    if (mGenieResources == null) return;

    Uri queryUri = new Uri.Builder()
        .scheme(android.content.ContentResolver.SCHEME_CONTENT)
        .authority(WEATHER_CONTENT_AUTHORITY)
        .path(WEATHER_CONTENT_PATH)
        .appendPath(new Long(System.currentTimeMillis()).toString())
        .build();

    if (DEBUG) Log.d(LOG_TAG, "querying genie: " + queryUri);

    Cursor cur;
    try {
        cur = getContentResolver().query(
            queryUri,
            WEATHER_CONTENT_COLUMNS,
            null,
            null,
            null);
    } catch (RuntimeException e) {
        Log.e(LOG_TAG, "Weather query failed", e);
        cur = null;
    }

    if (cur != null && cur.moveToFirst()) {
        if (DEBUG) {
            java.lang.StringBuilder sb =
                new java.lang.StringBuilder("Weather query result: {");
            for(int i=0; i<cur.getColumnCount(); i++) {
                if (i>0) sb.append(", ");
                sb.append(cur.getColumnName(i))
                    .append("=")
                    .append(cur.getString(i));
            }
            sb.append("}");
            Log.d(LOG_TAG, sb.toString());
        }

        mWeatherIconDrawable = mGenieResources.getDrawable(cur.getInt(
            cur.getColumnIndexOrThrow("iconResId")));

        mWeatherLocationString = cur.getString(
```

```
            cur.getColumnIndexOrThrow("location"));
\
         // any of these may be NULL
         final int colTemp = cur.getColumnIndexOrThrow("temperature");
         final int colHigh = cur.getColumnIndexOrThrow("highTemperature");
         final int colLow = cur.getColumnIndexOrThrow("lowTemperature");

         mWeatherCurrentTemperatureString =
            cur.isNull(colTemp)
               ? "\u2014"
               : String.format("%d\u00b0", cur.getInt(colTemp));
         mWeatherHighTemperatureString =
            cur.isNull(colHigh)
               ? "\u2014"
               : String.format("%d\u00b0", cur.getInt(colHigh));
         mWeatherLowTemperatureString =
            cur.isNull(colLow)
               ? "\u2014"
               : String.format("%d\u00b0", cur.getInt(colLow));
      } else {
         Log.w(LOG_TAG, "No weather information available (cur="
            + cur +")");
         mWeatherIconDrawable = null;
         mWeatherLocationString = getString(R.string.weather_fetch_failure);
         mWeatherCurrentTemperatureString =
            mWeatherHighTemperatureString =
            mWeatherLowTemperatureString = "";
      }

      if (cur != null) {
         // clean up cursor
         cur.close();
      }

      mHandy.sendEmptyMessage(UPDATE_WEATHER_DISPLAY_MSG);
}

private void refreshWeather() {
   if (supportsWeather())
      scheduleWeatherQueryDelayed(0);
   updateWeatherDisplay(); // in case we have it cached
}

private void updateWeatherDisplay() {
   if (mWeatherCurrentTemperature == null) return;

   mWeatherCurrentTemperature.setText(mWeatherCurrentTemperatureString);
   mWeatherHighTemperature.setText(mWeatherHighTemperatureString);
   mWeatherLowTemperature.setText(mWeatherLowTemperatureString);
   mWeatherLocation.setText(mWeatherLocationString);
   mWeatherIcon.setImageDrawable(mWeatherIconDrawable);
}
```

（6）在系统的主界面中显示了电量信息，此功能通过如下所示的函数实现。

- 函数 handleBatteryUpdate：功能是更新显示电量信息。
- 函数 refreshBattery：功能是刷新系统的当前电量。

上述函数的具体实现代码如下所示。

```
   private void handleBatteryUpdate(int plugStatus, int batteryLevel) {
      final boolean pluggedIn = (plugStatus == BATTERY_STATUS_CHARGING || plugStatus
== BATTERY_STATUS_FULL);
      if (pluggedIn != mPluggedIn) {
         setWakeLock(pluggedIn);

         if (pluggedIn) {
            // policy: update weather info when attaching to power
            requestWeatherDataFetch();
```

```
            }
        }
        if (pluggedIn != mPluggedIn || batteryLevel != mBatteryLevel) {
            mBatteryLevel = batteryLevel;
            mPluggedIn = pluggedIn;
            refreshBattery();
        }
    }

    private void refreshBattery() {
        if (mBatteryDisplay == null) return;

        if (mPluggedIn /* || mBatteryLevel < LOW_BATTERY_THRESHOLD */) {
            mBatteryDisplay.setCompoundDrawablesWithIntrinsicBounds(
                0, 0, android.R.drawable.ic_lock_idle_charging, 0);
            mBatteryDisplay.setText(
                getString(R.string.battery_charging_level, mBatteryLevel));
            mBatteryDisplay.setVisibility(View.VISIBLE);
        } else {
            mBatteryDisplay.setVisibility(View.INVISIBLE);
        }
    }
```

（7）为了及时更新系统主界面，特意提供了如下所示的刷新函数。

- 函数 refreshDate：功能是更新显示当前的时间。
- 函数 refreshAlarm：功能是刷新显示系统的闹钟。
- 函数 refreshAll：功能是分别调用时间、闹钟、电量和天气这 4 个刷新函数。

上述函数的具体实现代码如下所示。

```
private void refreshDate() {
    final Date now = new Date();
    if (DEBUG) Log.d(LOG_TAG, "refreshing date..." + now);
    mDate.setText(DateFormat.format(mDateFormat, now));
}

private void refreshAlarm() {
    if (mNextAlarm == null) return;

   String nextAlarm = Settings.System.getString(getContentResolver(),
          Settings.System.NEXT_ALARM_FORMATTED);
   if (!TextUtils.isEmpty(nextAlarm)) {
      mNextAlarm.setText(nextAlarm);
      mNextAlarm.setCompoundDrawablesWithIntrinsicBounds(
          android.R.drawable.ic_lock_idle_alarm, 0, 0, 0);
      mNextAlarm.setVisibility(View.VISIBLE);
   } else {
      mNextAlarm.setVisibility(View.INVISIBLE);
   }


rivate void refreshAll() {
   refreshDate();
   refreshAlarm();
   refreshBattery();
   refreshWeather();
```

（8）定义函数 doDim，设置系统进入模糊模式显示状态。在系统从休眠模式恢复到正常模式时，之间的模式就是模糊模式。函数 doDim 的具体实现代码如下所示。

```
private void doDim(boolean fade) {
    View tintView = findViewById(R.id.window_tint);
    if (tintView == null) return;

    Window win = getWindow();
```

```
    WindowManager.LayoutParams winParams = win.getAttributes();

    winParams.flags |= (WindowManager.LayoutParams.FLAG_LAYOUT_IN_SCREEN);
    winParams.flags |= (WindowManager.LayoutParams.FLAG_LAYOUT_NO_LIMITS);

    // dim the wallpaper somewhat (how much is determined below)
    winParams.flags |= (WindowManager.LayoutParams.FLAG_DIM_BEHIND);

    if (mDimmed) {
        winParams.flags |= WindowManager.LayoutParams.FLAG_FULLSCREEN;
        winParams.dimAmount = DIM_BEHIND_AMOUNT_DIMMED;
        winParams.buttonBrightness =WindowManager.LayoutParams.BRIGHTNESS_OVERRIDE_OFF;

        // show the window tint
        tintView.startAnimation(AnimationUtils.loadAnimation(this,
            fade ? R.anim.dim
                : R.anim.dim_instant));
    } else {
        winParams.flags &= (~WindowManager.LayoutParams.FLAG_FULLSCREEN);
        winParams.dimAmount = DIM_BEHIND_AMOUNT_NORMAL;
        winParams.buttonBrightness =WindowManager.LayoutParams.BRIGHTNESS_OVERRIDE_NONE;

        // hide the window tint
        tintView.startAnimation(AnimationUtils.loadAnimation(this,
            fade ? R.anim.undim
                : R.anim.undim_instant));
    }

    win.setAttributes(winParams);
}
```

（9）为了保证在系统主界面中实现不同模式的转换，特意提供了如下所示的转换函数。

- 函数 initViews：功能是初始化视图界面。
- 函数 onPause：功能是暂停当前的模式转换，关闭屏幕保护程序，并取消任何超时操作，但模糊模式除外。
- 函数 onStop：功能是停止当前的模式转换。
- 函数 onStart：功能是开启系统界面，分别调用对应的函数显示时间、闹钟、电量和天气。
- 函数 onNewIntent：功能是开启新的界面视图。
- 函数 onConfigurationChanged：功能是根据配置改变显示视图。

上述函数的具体实现代码如下所示。

```
public void onNewIntent(Intent newIntent) {
    super.onNewIntent(newIntent);
    if (DEBUG) Log.d(LOG_TAG, "onNewIntent with intent: " + newIntent);

    // update our intent so that we can consult it to determine whether or
    // not the most recent launch was via a dock event
    setIntent(newIntent);
}

@Override
public void onStart() {
    super.onStart();

    IntentFilter filter = new IntentFilter();
    filter.addAction(Intent.ACTION_DATE_CHANGED);
    filter.addAction(Intent.ACTION_BATTERY_CHANGED);
    filter.addAction(UiModeManager.ACTION_EXIT_DESK_MODE);
    filter.addAction(ACTION_MIDNIGHT);
    registerReceiver(mIntentReceiver, filter);
}

@Override
public void onStop() {
```

```
        super.onStop();

        unregisterReceiver(mIntentReceiver);
    }

    @Override
    public void onResume() {
        super.onResume();
        if (DEBUG) Log.d(LOG_TAG, "onResume with intent: " + getIntent());

        // reload the date format in case the user has changed settings
        // recently
        mDateFormat = getString(R.string.full_wday_month_day_no_year);

        // Listen for updates to weather data
        Uri weatherNotificationUri = new Uri.Builder()
            .scheme(android.content.ContentResolver.SCHEME_CONTENT)
            .authority(WEATHER_CONTENT_AUTHORITY)
            .path(WEATHER_CONTENT_PATH)
            .build();
        getContentResolver().registerContentObserver(
            weatherNotificationUri, true, mContentObserver);

        // Elaborate mechanism to find out when the day rolls over
        Calendar today = Calendar.getInstance();
        today.set(Calendar.HOUR_OF_DAY, 0);
        today.set(Calendar.MINUTE, 0);
        today.set(Calendar.SECOND, 0);
        today.add(Calendar.DATE, 1);
        long alarmTimeUTC = today.getTimeInMillis();

        mMidnightIntent = PendingIntent.getBroadcast(this, 0, new Intent(ACTION_MIDNIGHT), 0);
        AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
        am.setRepeating(AlarmManager.RTC, alarmTimeUTC, AlarmManager.INTERVAL_DAY,
        mMidnightIntent);
        if (DEBUG) Log.d(LOG_TAG, "set repeating midnight event at UTC: "
            + alarmTimeUTC + " ("
            + (alarmTimeUTC - System.currentTimeMillis())
            + " ms from now) repeating every "
            + AlarmManager.INTERVAL_DAY + " with intent: " + mMidnightIntent);

        // If we weren't previously visible but now we are, it's because we're
        // being started from another activity. So it's OK to un-dim.
        if (mTime != null && mTime.getWindowVisibility() != View.VISIBLE) {
            mDimmed = false;
        }

        // Adjust the display to reflect the currently chosen dim mode
        doDim(false);

        restoreScreen(); // disable screen saver
        refreshAll(); // will schedule periodic weather fetch

        setWakeLock(mPluggedIn);

        scheduleScreenSaver();

        final boolean launchedFromDock
            = getIntent().hasCategory(Intent.CATEGORY_DESK_DOCK);

        if (supportsWeather() && launchedFromDock && !mLaunchedFromDock) {
            // policy: fetch weather if launched via dock connection
            if (DEBUG) Log.d(LOG_TAG, "Device now docked; forcing weather to refresh right now");
            requestWeatherDataFetch();
        }

        mLaunchedFromDock = launchedFromDock;
    }

    @Override
    public void onPause() {
```

```
    if (DEBUG) Log.d(LOG_TAG, "onPause");

    // Turn off the screen saver and cancel any pending timeouts
    // (But don't un-dim.)
    mHandy.removeMessages(SCREEN_SAVER_TIMEOUT_MSG);
    restoreScreen();

    // Other things we don't want to be doing in the background
    // NB: we need to keep our broadcast receiver alive in case the dock
    // is disconnected while the screen is off
    getContentResolver().unregisterContentObserver(mContentObserver);

    AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
    am.cancel(mMidnightIntent);
    unscheduleWeatherQuery();

    super.onPause();
}

private void initViews() {
    //放弃任何在当前布局中的切换操作
    final View focused = getCurrentFocus();
    if (focused != null) focused.clearFocus();

    setContentView(R.layout.desk_clock);

    mTime = (DigitalClock) findViewById(R.id.time);
    mDate = (TextView) findViewById(R.id.date);
    mBatteryDisplay = (TextView) findViewById(R.id.battery);

    mTime.getRootView().requestFocus();

    mWeatherCurrentTemperature = (TextView) findViewById(R.id.weather_temperature);
    mWeatherHighTemperature = (TextView) findViewById(R.id.weather_high_temperature);
    mWeatherLowTemperature = (TextView) findViewById(R.id.weather_low_temperature);
    mWeatherLocation = (TextView) findViewById(R.id.weather_location);
    mWeatherIcon = (ImageView) findViewById(R.id.weather_icon);

    final View.OnClickListener alarmClickListener = new View.OnClickListener() {
        public void onClick(View v) {
            startActivity(new Intent(DeskClock.this, AlarmClock.class));
        }
    };
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    if (mScreenSaverMode) {
        moveScreenSaver();
    } else {
        initViews();
        doDim(false);
        refreshAll();
    }
}
```

（10）在主屏幕底部生成 4 个选项，单击不同的选项后会实现对应的操作，此功能的实现函数如下所示。

- 函数 onOptionsItemSelected：功能是根据用户选择的选项启动对应的视图界面。
- 函数 onCreateOptionsMenu：功能是创建屏幕底部的选项菜单。
- 函数 onCreate：功能是创建菜单视图。

上述函数的具体实现代码如下所示。

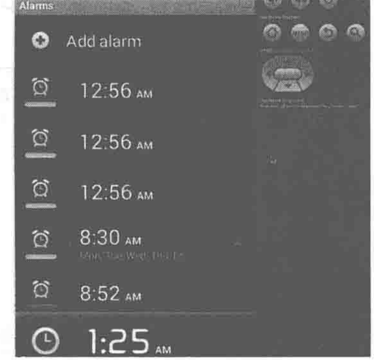
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_item_alarms:
            startActivity(new Intent(DeskClock.this, AlarmClock.class));
```

```
            return true;
        case R.id.menu_item_add_alarm:
            startActivity(new Intent(this, SetAlarm.class));
            return true;
        case R.id.menu_item_dock_settings:
            startActivity(new Intent(DOCK_SETTINGS_ACTION));
            return true;
        default:
            return false;
    }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.desk_clock_menu, menu);
    return true;
}

@Override
protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    mRNG = new Random();

    try {
        mGenieResources =getPackageManager().getResourcesForApplication(GENIE_PACKAGE_ID);
    } catch (PackageManager.NameNotFoundException e) {
        // no weather info available
        Log.w(LOG_TAG, "Can't find "+GENIE_PACKAGE_ID+". Weather forecast will not be 
available.");
    }

    initViews();
}
```

24.3 闹钟列表模块

当单击系统主界面底部导航菜单中的图标后，回来到闹钟列表界面。执行效果如图 24-3 所示。

在本节的内容中，将详细讲解本系统闹钟列表模块的具体实现过程。



▲图 24-3　闹钟列表界面执行效果

24.3.1　设置主界面

图 24-3 所示的闹钟列表界面中，在顶部显示一个添加按钮图标；在中间显示了一个两列列表，左侧列表显示了系统中的所有闹钟的“是否可用”开关，在右侧列表显示了各个闹钟的具体时间；在底部显示了系统的当前时间。设置主界面的布局文件是 alarm_clock.xml，具体实现代码如下所示。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/base_layout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <LinearLayout android:id="@+id/add_alarm"
        android:clickable="true"
        android:focusable="true"
        android:background="@android:drawable/list_selector_background"
```

```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
        style="@style/alarm_list_left_column"
        android:duplicateParentState="true"
        android:gravity="center"
        android:scaleType="center"
        android:src="@drawable/add_alarm" />

    <TextView
        android:duplicateParentState="true"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_gravity="center_vertical"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:textColor="?android:attr/textColorPrimary"
        android:text="@string/add_alarm" />

</LinearLayout>

<ImageView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:scaleType="fitXY"
    android:gravity="fill_horizontal"
    android:src="@android:drawable/divider_horizontal_dark" />

<ListView
    android:id="@+id/alarms_list"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1" />

<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >

    <ImageButton android:id="@+id/desk_clock_button"
        style="@style/ButtonStripLeft"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_clock_strip_desk_clock"
        android:contentDescription="@string/desk_clock_button_description"/>

    <com.android.superdeskclock.DigitalClock
        style="@style/ButtonStripRight"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

        <LinearLayout
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:baselineAligned="true">

            <TextView android:id="@+id/timeDisplay"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:paddingRight="6dip"
                android:textSize="48sp"
                android:textColor="?android:attr/textColorPrimary" />

            <TextView android:id="@+id/am_pm"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textAppearance="?android:attr/textAppearanceSmall"
                android:textStyle="bold"
```

```
                android:textColor="?android:attr/textColorPrimary" />
            </LinearLayout>
        </com.android.superdeskclock.DigitalClock>
    </LinearLayout>
</LinearLayout>
```

设置主界面的对应程序文件是 AlarmClock.java，具体实现流程如下所示。

（1）定义函数 bindView，根据 findViewById 获取系统内的闹钟，并将获取的数据绑定到 ListView 控件中，以列表样式显示出来。函数 bindView 的具体实现代码如下所示。

```
    public void bindView(View view, Context context, Cursor cursor) {
        final Alarm alarm = new Alarm(cursor);

        View indicator = view.findViewById(R.id.indicator);

        // Set the initial resource for the bar image
        final ImageView barOnOff = (ImageView) indicator.findViewById(R.id.bar_onoff);
        barOnOff.setImageResource(alarm.enabled ?R.drawable.ic_indicator_on :R.drawable.
        ic_indicator_off);

        // Set the initial state of the clock "checkbox"
        final CheckBox clockOnOff =(CheckBox) indicator.findViewById(R.id.clock_onoff);
        clockOnOff.setChecked(alarm.enabled);

        //新增
        this.context=context;
        // Clicking outside the "checkbox" should also change the state
        indicator.setOnClickListener(new OnClickListener() {
                public void onClick(View v) {
                    myDialog = ProgressDialog.show(AlarmTimeAdapter.this.context,"提示",
                    "请稍候",true);
                    new Thread(){
                        public void run(){
                            try{
                                sleep(800);
                            }catch (Exception e){
                                e.printStackTrace();
                            }finally{
                                // 卸载所创建的 myDialog 对象
                                myDialog.dismiss();
                            }
                        }
                    }.start();
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    clockOnOff.toggle();
                    updateIndicatorAndAlarm(clockOnOff.isChecked(),barOnOff, alarm);
                }
        });

        DigitalClock digitalClock =(DigitalClock) view.findViewById(R.id.digitalClock);

        // set the alarm text
        final Calendar c = Calendar.getInstance();
        c.set(Calendar.HOUR_OF_DAY, alarm.hour);
        c.set(Calendar.MINUTE, alarm.minutes);
        digitalClock.updateTime(c);
        digitalClock.setTypeface(Typeface.DEFAULT);

        // Set the repeat text or leave it blank if it does not repeat
        TextView daysOfWeekView = (TextView) digitalClock.findViewById(R.id.daysOfWeek);
        final String daysOfWeekStr = alarm.daysOfWeek.toString(AlarmClock.this, false);
        if (daysOfWeekStr != null && daysOfWeekStr.length() != 0) {
            daysOfWeekView.setText(daysOfWeekStr);
            daysOfWeekView.setVisibility(View.VISIBLE);
```

```
        } else {
            daysOfWeekView.setVisibility(View.GONE);
        }

        // Display the label
        TextView labelView = (TextView) view.findViewById(R.id.label);
        if (alarm.label != null && alarm.label.length() != 0) {
            labelView.setText(alarm.label);
            labelView.setVisibility(View.VISIBLE);
        } else {
            labelView.setVisibility(View.GONE);
        }
    }
};
```

（2）定义函数 onContextItemSelected，功能是根据用户选择确认是否列表中的闹钟处于可用状态，并根据用户选择来到不同状态的处理界面。函数 onContextItemSelected 的具体实现代码如下所示。

```
@Override
public boolean onContextItemSelected(final MenuItem item) {
    final AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
    final int id = (int) info.id;
    switch (item.getItemId()) {
        case R.id.delete_alarm:
            // 确认闹钟已被删除
            new AlertDialog.Builder(this)
                    .setTitle(getString(R.string.delete_alarm))
                    .setMessage(getString(R.string.delete_alarm_confirm))
                    .setPositiveButton(android.R.string.ok,
                            new DialogInterface.OnClickListener() {
                                public void onClick(DialogInterface d, int w) {
                                    Alarms.deleteAlarm(AlarmClock.this, id);
                                }
                            })
                    .setNegativeButton(android.R.string.cancel, null)
                    .show();
            return true;

        case R.id.enable_alarm:
            final Cursor c = (Cursor) mAlarmsList.getAdapter().getItem(info.position);
            final Alarm alarm = new Alarm(c);
            //修改
            Alarms.enableAlarm(this, alarm.id, !alarm.enabled);
            if (!alarm.enabled) {
                SetAlarm.popAlarmSetToast(this,alarm.hour,alarm.minutes,alarm.daysOfWeek);
            }
            return true;

        case R.id.edit_alarm:
            Intent intent = new Intent(this, SetAlarm.class);
            intent.putExtra(Alarms.ALARM_ID, id);
            startActivity(intent);
            return true;

        default:
            break;
    }
    return super.onContextItemSelected(item);
}
```

（3）定义函数 onCreate，功能是显示当前的界面。定义函数 updateLayout，根据用户选择更新显示为对应的界面。具体实现代码如下所示。

```
@Override
protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);
```

```
        mFactory = LayoutInflater.from(this);
        mPrefs = getSharedPreferences(PREFERENCES, 0);
        mCursor = Alarms.getAlarmsCursor(getContentResolver());

       updateLayout();
    }

    private void updateLayout() {
        setContentView(R.layout.alarm_clock);
        mAlarmsList = (ListView) findViewById(R.id.alarms_list);
        AlarmTimeAdapter adapter = new AlarmTimeAdapter(this, mCursor);
        mAlarmsList.setAdapter(adapter);
        mAlarmsList.setVerticalScrollBarEnabled(true);
        mAlarmsList.setOnItemClickListener(this);
        mAlarmsList.setOnCreateContextMenuListener(this);

        View addAlarm = findViewById(R.id.add_alarm);
        addAlarm.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                    addNewAlarm();
                }
            });
        // 当在屏幕中聚焦时选择这个选项
        addAlarm.setOnFocusChangeListener(new View.OnFocusChangeListener() {
                public void onFocusChange(View v, boolean hasFocus) {
                    v.setSelected(hasFocus);
                }
        });

        ImageButton deskClock =
                (ImageButton) findViewById(R.id.desk_clock_button);
        deskClock.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                    startActivity(new Intent(AlarmClock.this, DeskClock.class));
                }
        });
    }
```

（4）定义函数 addNewAlarm，功能是启动添加闹钟界面。定义函数 onCreateContextMenu，功能是创建一个上下文菜单。具体实现代码如下所示。

```
    private void addNewAlarm() {
        startActivity(new Intent(this, SetAlarm.class));
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        ToastMaster.cancelToast();
        mCursor.deactivate();
    }

    @Override
    public void onCreateContextMenu(ContextMenu menu, View view,
            ContextMenuInfo menuInfo) {
        // 从 xml 文件创建菜单.
        getMenuInflater().inflate(R.menu.context_menu, menu);

        //使用当前项目创建自定义视图的页眉
        final AdapterContextMenuInfo info = (AdapterContextMenuInfo) menuInfo;
        final Cursor c =
                (Cursor) mAlarmsList.getAdapter().getItem((int) info.position);
        final Alarm alarm = new Alarm(c);

        //构建日历计算时间
        final Calendar cal = Calendar.getInstance();
        cal.set(Calendar.HOUR_OF_DAY, alarm.hour);
```

```
    cal.set(Calendar.MINUTE, alarm.minutes);
    final String time = Alarms.formatTime(this, cal);

    //设置自定义视图和每个程序的文本
    final View v = mFactory.inflate(R.layout.context_menu_header, null);
    TextView textView = (TextView) v.findViewById(R.id.header_time);
    textView.setText(time);
    textView = (TextView) v.findViewById(R.id.header_label);
    textView.setText(alarm.label);

    //在菜单上设置自定义视图
    menu.setHeaderView(v);
    // Change the text based on the state of the alarm.
    if (alarm.enabled) {
        menu.findItem(R.id.enable_alarm).setTitle(R.string.disable_alarm);
    }
}
```

当按下“MENU”按钮后会在屏幕底部弹出这个菜单项，如图 24-4 所示。

在图 24-4 所示的菜单相中，包含了闹钟设置、添加闹钟和闹钟主界面 3 个选项。

（5）定义函数 onOptionsItemSelected，功能是根据用户在函数 onCreateContextMenu 中创建的上下文菜单上选择的项跳转到不同的界面。具体实现代码如下所示。

▲图 24-4　屏幕底部的上下文菜单项

```
Override
ublic boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      case R.id.menu_item_settings:
         startActivity(new Intent(this, SettingsActivity.class));
         return true;
      case R.id.menu_item_desk_clock:
         startActivity(new Intent(this, DeskClock.class));
         return true;
      case R.id.menu_item_add_alarm:
         addNewAlarm();
         return true;
      default:
         break;
   }
   return super.onOptionsItemSelected(item);


@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.alarm_list_menu, menu);
    return super.onCreateOptionsMenu(menu);
}

public void onItemClick(AdapterView parent, View v, int pos, long id) {
    Intent intent = new Intent(this, SetAlarm.class);
    intent.putExtra(Alarms.ALARM_ID, (int) id);
    startActivity(intent);
}
```

24.3.2 设置闹钟界面

当在图 24-3 所示的闹钟列表界面中选择一个闹钟时，会来到设置闹钟界面。在此界面中可以设置被选中闹钟的选项，如开关、时间、重复次数、铃声和振动等。设置闹钟界面的执行效果如图 24-5 所示。

▲图 24-5　设置闹钟界面的执行效果

图 24-5 所示的界面效果是由文件 set_alarm.xml 实现布局的，具体实现代码如下所示。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical">
   <ListView android:id="@android:id/list"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:drawSelectorOnTop="false"/>
   <LinearLayout
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal"
      style="@android:style/ButtonBar">
      <Button android:id="@+id/alarm_save"
         android:focusable="true"
         android:layout_width="fill_parent"
         android:layout_height="fill_parent"
         android:layout_weight="1"
         android:text="@string/done"/>
      <Button android:id="@+id/alarm_revert"
         android:focusable="true"
         android:layout_width="fill_parent"
         android:layout_height="fill_parent"
         android:layout_weight="1"
         android:text="@string/revert"/>
      <Button android:id="@+id/alarm_delete"
         android:focusable="true"
         android:layout_width="fill_parent"
         android:layout_height="fill_parent"
         android:layout_weight="1"
         android:text="@string/delete"/>
   </LinearLayout>
</LinearLayout>
```

设置闹钟界面的程序文件是 SetAlarm.java，具体实现流程如下所示。

（1）定义继承于 PreferenceActivity 的类 SetAlarm，用于管理系统中的每一个闹钟，具体实现代码如下所示。

```
public class SetAlarm extends PreferenceActivity
      implements TimePickerDialog.OnTimeSetListener,
      Preference.OnPreferenceChangeListener {
```

```
    private EditTextPreference mLabel;

    //新增
    private EditTextPreference times;
    private EditTextPreference interval;


    private CheckBoxPreference mEnabledPref;
    private Preference mTimePref;
    private SetBellPreference mAlarmPref;
    private RadioButton RadioButton;
    private CheckBoxPreference mVibratePref;
    private RepeatPreference mRepeatPref;
    private MenuItem mTestAlarmItem;

    private int     mId;
    private int     mHour;
    private int     mMinutes;
    private boolean mTimePickerCancelled;
    private Alarm   mOriginalAlarm;
```

（2）定义函数 onCreate 以创建此界面的显示视图，加载显示系统中原来对这个闹钟的设置。函数 onCreate 的具体实现代码如下所示。

```
/**
 *设置一个 Alarm，通过外部 Alarms.ALARM_ID 实现
 *诸如其他 Activity 之类的 Alarm 对象
 */
@Override
protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    // Override the default content view
    setContentView(R.layout.set_alarm);
    addPreferencesFromResource(R.xml.alarm_prefs);
    //新增
    times = (EditTextPreference) findPreference("times");
    times.setOnPreferenceChangeListener(
            new Preference.OnPreferenceChangeListener() {
                public boolean onPreferenceChange(Preference p,
                        Object newValue) {
                    String val = (String) newValue;
                    // Set the summary based on the new label.
                    p.setSummary(val);
                    if (val != null && !val.equals(times.getText())) {
                        // Call through to the generic listener.
                        return SetAlarm.this.onPreferenceChange(p,
                            newValue);
                    }
                    return true;
                }
            });
    // 检索值，获取每一个偏好值
    mLabel = (EditTextPreference) findPreference("label");
    mLabel.setOnPreferenceChangeListener(
            new Preference.OnPreferenceChangeListener() {
                public boolean onPreferenceChange(Preference p,
                        Object newValue) {
                    String val = (String) newValue;
                    // Set the summary based on the new label
                    p.setSummary(val);
                    if (val != null && !val.equals(mLabel.getText())) {
                        // Call through to the generic listener.
                        return SetAlarm.this.onPreferenceChange(p,
                            newValue);
                    }
                    return true;
                }
```

```
        });
mEnabledPref = (CheckBoxPreference) findPreference("enabled");
mEnabledPref.setOnPreferenceChangeListener(
        new Preference.OnPreferenceChangeListener() {
            public boolean onPreferenceChange(Preference p,
                    Object newValue) {
                // Pop a toast when enabling alarms
                if (!mEnabledPref.isChecked()) {
                    popAlarmSetToast(SetAlarm.this, mHour, mMinutes,
                        mRepeatPref.getDaysOfWeek());
                }
                return SetAlarm.this.onPreferenceChange(p, newValue);
            }
        });
mTimePref = findPreference("time");
mAlarmPref = (SetBellPreference) findPreference("alarm");
mAlarmPref.setOnPreferenceChangeListener(this);
mVibratePref = (CheckBoxPreference) findPreference("vibrate");
mVibratePref.setOnPreferenceChangeListener(this);
mRepeatPref = (RepeatPreference) findPreference("setRepeat");
mRepeatPref.setOnPreferenceChangeListener(this);

Intent i = getIntent();
mId = i.getIntExtra(Alarms.ALARM_ID, -1);
if (Log.LOGV) {
    Log.v("In SetAlarm, alarm id = " + mId);
}

Alarm alarm = null;
if (mId == -1) {
    // 没有闹钟则新建一个
    alarm = new Alarm();
} else {
    /* 从数据库中显示这个闹钟的详细信息 */
    alarm = Alarms.getAlarm(getContentResolver(), mId);
    // Bad alarm, bail to avoid a NPE
    if (alarm == null) {
        finish();
        return;
    }
}
mOriginalAlarm = alarm;

updatePrefs(mOriginalAlarm);

// 选中时高亮显示
getListView().setItemsCanFocus(true);

// 给每个按钮附加操作
Button b = (Button) findViewById(R.id.alarm_save);
b.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            saveAlarm();
            finish();
        }
});
final Button revert = (Button) findViewById(R.id.alarm_revert);
revert.setEnabled(false);
revert.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            int newId = mId;
            updatePrefs(mOriginalAlarm);
            // "Revert" on a newly created alarm should delete it
            if (mOriginalAlarm.id == -1) {
                Alarms.deleteAlarm(SetAlarm.this, newId);
            } else {
                saveAlarm();
            }
            revert.setEnabled(false);
```

```
        }
    });
    b = (Button) findViewById(R.id.alarm_delete);
    if (mId == -1) {
        b.setEnabled(false);
    } else {
        b.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                deleteAlarm();
            }
        });
    }

    // 如果改变了时间，则弹出时间选择器
    if (mId == -1) {
        // Assume the user hit cancel
        mTimePickerCancelled = true;
        showTimePicker();
    }
}
private static final Handler sHandler = new Handler();
```

（3）定义函数 onPreferenceChange 以处理偏好变化，具体实现代码如下所示。

```
public boolean onPreferenceChange(final Preference p, Object newValue) {
    // 定义异步方法保存偏好值的更改
    sHandler.post(new Runnable() {
        public void run() {
            // 编辑可设置的偏好
            if (p != mEnabledPref) {
                mEnabledPref.setChecked(true);
            }
            saveAlarmAndEnableRevert();
        }
    });
    return true;
}
```

（4）定义函数 updatePrefs 以更新对这个闹钟的设置，具体实现代码如下所示。

```
private void updatePrefs(Alarm alarm) {
    mId = alarm.id;
    mEnabledPref.setChecked(alarm.enabled);
    mLabel.setText(alarm.label);
    mLabel.setSummary(alarm.label);
    mHour = alarm.hour;
    mMinutes = alarm.minutes;
    mRepeatPref.setDaysOfWeek(alarm.daysOfWeek);
    mVibratePref.setChecked(alarm.vibrate);
    // Give the alert uri to the preference
    mAlarmPref.setAlert(alarm.alert);

    //新增
    times.setText(""+alarm.times);
    times.setSummary(""+alarm.times);


    updateTime();
}
```

（5）定义函数 onTimeSet 以设置新闹钟的时间，定义函数 onTimeSet 更改这个闹钟的时间，具体实现代码如下所示。

```
public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
    // onTimeSet is called when the user clicks "Set"
    mTimePickerCancelled = false;
    mHour = hourOfDay;
    mMinutes = minute;
```

```
        updateTime();
        // If the time has been changed, enable the alarm
        mEnabledPref.setChecked(true);
        // Save the alarm and pop a toast
        popAlarmSetToast(this, saveAlarmAndEnableRevert());
    }
    private void updateTime() {
        if (Log.LOGV) {
            Log.v("updateTime " + mId);
        }
        mTimePref.setSummary(Alarms.formatTime(this, mHour, mMinutes,
                mRepeatPref.getDaysOfWeek()));
    }
```

（6）定义函数 saveAlarmAndEnableRevert 以保存闹钟，并设置“Revert”按钮不可用，具体实现代码如下所示。

```
    private long saveAlarmAndEnableRevert() {
        // Enable "Revert" to go back to the original Alarm
        final Button revert = (Button) findViewById(R.id.alarm_revert);
        revert.setEnabled(true);
        return saveAlarm();
    }
```

（7）定义函数 saveAlarm 以保存对当前闹钟的设置，具体实现代码如下所示。

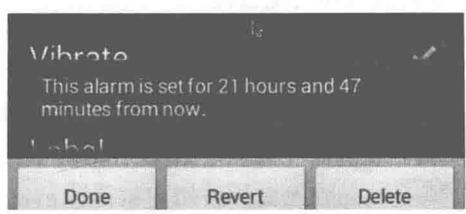
```
    private long saveAlarm() {
        Alarm alarm = new Alarm();
        alarm.id = mId;
        alarm.enabled = mEnabledPref.isChecked();
        alarm.hour = mHour;
        alarm.minutes = mMinutes;
        alarm.daysOfWeek = mRepeatPref.getDaysOfWeek();
        alarm.vibrate = mVibratePref.isChecked();
        alarm.label = mLabel.getText();
        alarm.alert = mAlarmPref.getAlert();

        //新增
        String timesText=times.getText();
        alarm.times=Integer.parseInt(timesText==null||"".equals(timesText)?"0":
    timesText);
        alarm.interval=0;

        long time;
        if (alarm.id == -1) {
            time = Alarms.addAlarm(this, alarm);
            // addAlarm populates the alarm with the new id. Update mId so that
            // changes to other preferences update the new alarm
            mId = alarm.id;
        } else {
            time = Alarms.setAlarm(this, alarm);
        }
        return time;
    }
```

（8）定义函数 deleteAlarm 以删除当前闹钟，在单击屏幕底部的“Delete”按钮时被触发。函数 deleteAlarm 的具体实现代码如下所示。

```
    private void deleteAlarm() {
        new AlertDialog.Builder(this)
                .setTitle(getString(R.string.delete_alarm))
                .setMessage(getString(R.string.delete_alarm_confirm))
                .setPositiveButton(android.R.string.ok,
                        new DialogInterface.OnClickListener() {
                            public void onClick(DialogInterface d, int w) {
                                Alarms.deleteAlarm(SetAlarm.this, mId);
                                finish();
```

```
                }
            })
        .setNegativeButton(android.R.string.cancel, null)
        .show();
}
```

（9）定义函数 popAlarmSetToast，功能是当设置完毕后单击屏幕底部的“Done”按钮后显示一个闹钟设置提醒信息，如图 24-6 所示。



▲图 24-6　闹钟提醒信息效果

函数 popAlarmSetToast 的具体实现代码如下所示。

```
static void popAlarmSetToast(Context context, int hour, int minute,
                        Alarm.DaysOfWeek daysOfWeek) {
    popAlarmSetToast(context,
            Alarms.calculateAlarm(hour, minute, daysOfWeek)
            .getTimeInMillis());
}

private static void popAlarmSetToast(Context context, long timeInMillis) {
    String toastText = formatToast(context, timeInMillis);
    Toast toast = Toast.makeText(context, toastText, Toast.LENGTH_LONG);
    ToastMaster.setToast(toast);
    toast.show();
}
```

（10）定义函数 formatToast，功能是设置提醒信息的内容，具体实现代码如下所示。

```
static String formatToast(Context context, long timeInMillis) {
    long delta = timeInMillis - System.currentTimeMillis();
    long hours = delta / (1000 * 60 * 60);
    long minutes = delta / (1000 * 60) % 60;
    long days = hours / 24;
    hours = hours % 24;

    String daySeq = (days == 0) ? "" :
            (days == 1) ? context.getString(R.string.day) :
            context.getString(R.string.days, Long.toString(days));

    String minSeq = (minutes == 0) ? "" :
            (minutes == 1) ? context.getString(R.string.minute) :
            context.getString(R.string.minutes, Long.toString(minutes));

    String hourSeq = (hours == 0) ? "" :
            (hours == 1) ? context.getString(R.string.hour) :
            context.getString(R.string.hours, Long.toString(hours));

    boolean dispDays = days > 0;
    boolean dispHour = hours > 0;
    boolean dispMinute = minutes > 0;

    int index = (dispDays ? 1 : 0) |
                (dispHour ? 2 : 0) |
                (dispMinute ? 4 : 0);

    String[] formats = context.getResources().getStringArray(R.array.alarm_set);
    return String.format(formats[index], daySeq, hourSeq, minSeq);
```

```
}
public void setInterval(EditTextPreference interval) {
    this.interval = interval;
}
```

24.3.3　闹钟提醒模块

在本实例中，涉及了多处闹钟提醒模块，如闹钟到时响起的提醒、闹钟全屏提醒。在本节的内容中，将详细讲解闹钟提醒模块的具体实现过程。

（1）界面布局文件是 alarm_alert.xml，具体实现代码如下所示。

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center">
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:gravity="center_horizontal"
        android:background="@drawable/dialog"
        android:orientation="vertical">
        <TextView android:id="@+id/alertTitle"
            style="?android:attr/textAppearanceLarge"
            android:padding="5dip"
            android:singleLine="true"
            android:ellipsize="end"
            android:gravity="center"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content" />
        <ImageView
            android:layout_width="fill_parent"
            android:layout_height="1dip"
            android:scaleType="fitXY"
            android:gravity="fill_horizontal"
            android:src="@drawable/dialog_divider_horizontal_light"
            android:layout_marginLeft="10dip"
            android:layout_marginRight="10dip"/>
        <com.android.superdeskclock.DigitalClock
            style="@style/clock"
            android:paddingTop="30dip"
            android:paddingBottom="30dip"
            android:baselineAligned="true"
            android:gravity="center_horizontal">
            <TextView android:id="@+id/timeDisplay"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="64sp"
                android:textColor="?android:attr/textColorPrimary"/>
            <TextView android:id="@+id/am_pm"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textStyle="bold"
                android:textAppearance="?android:attr/textAppearanceMedium"
                android:textColor="?android:attr/textColorPrimary"/>
        </com.android.superdeskclock.DigitalClock>
        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            style="@android:style/ButtonBar">
            <Button
                android:id="@+id/snooze"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_weight="3"
                android:text="@string/alarm_alert_snooze_text" />
```

```
        <!-- blank stretchable view -->
        <View
            android:layout_width="2dip"
            android:layout_height="2dip"
            android:layout_gravity="fill_horizontal"
            android:layout_weight="1"/>
        <Button
            android:id="@+id/dismiss"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="3"
            android:text="@string/alarm_alert_dismiss_text" />
      </LinearLayout>
   </LinearLayout>
</LinearLayout>
```

（2）文件 AlarmAlert.java 的功能是实现全屏幕的闹钟提示，使用持久性的可见指示器和指定音乐实现。文件 AlarmAlert.java 的具体实现代码如下所示。

```
public class AlarmAlert extends AlarmAlertFullScreen {
   //如果尝试操作键盘锁 5 次以上，则退出全屏幕活动
   private int mKeyguardRetryCount;
   private final int MAX_KEYGUARD_CHECKS = 5;

   private final Handler mHandler = new Handler() {
      @Override
      public void handleMessage(Message msg) {
         handleScreenOff((KeyguardManager) msg.obj);
      }
   };

   private final BroadcastReceiver mScreenOffReceiver =
         new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
               KeyguardManager km =
                     (KeyguardManager) context.getSystemService(
                     Context.KEYGUARD_SERVICE);
               handleScreenOff(km);
            }
         };

   @Override
   protected void onCreate(Bundle icicle) {
      super.onCreate(icicle);

      //当屏幕恢复时监听屏幕关闭，这样用户不需要解锁解除闹钟
      registerReceiver(mScreenOffReceiver,
            new IntentFilter(Intent.ACTION_SCREEN_OFF));
   }

   @Override
   public void onDestroy() {
      super.onDestroy();
      unregisterReceiver(mScreenOffReceiver);
      //删除任何键盘锁消息
      mHandler.removeMessages(0);
   }

   @Override
   public void onBackPressed() {
      finish();
   }

   private boolean checkRetryCount() {
      if (mKeyguardRetryCount++ >= MAX_KEYGUARD_CHECKS) {
         Log.e("Tried to read keyguard status too many times, bailing...");
         return false;
```

```
            }
            return true;
        }
        private void handleScreenOff(final KeyguardManager km) {
            if (!km.inKeyguardRestrictedInputMode() && checkRetryCount()) {
                if (checkRetryCount()) {
                    mHandler.sendMessageDelayed(mHandler.obtainMessage(0, km), 500);
                }
            } else {
                //启动全屏活动但不打开屏幕
                Intent i = new Intent(this, AlarmAlertFullScreen.class);
                i.putExtra(Alarms.ALARM_INTENT_EXTRA, mAlarm);
                i.putExtra(SCREEN_OFF, true);
                startActivity(i);
                finish();
            }
        }
    }
```

（3）文件 AlarmAlertFullScreen.java 的功能是实现有背景桌面的、被锁定的全屏幕闹钟提示效果，此类型闹钟使用持久性的可见指示器和指定音乐实现。文件 AlarmAlertFullScreen.java 的具体实现代码如下所示。

```
public class AlarmAlertFullScreen extends Activity {

    // 下面的值和文件"xml/settings.xml"中的相对应
    private static final String DEFAULT_SNOOZE = "10";
    private static final String DEFAULT_VOLUME_BEHAVIOR = "2";
    protected static final String SCREEN_OFF = "screen_off";

    protected Alarm mAlarm;
    private int mVolumeBehavior;

    // 从 AlarmKlaxon 接收 ALARM_KILLED 操作
    // 也从其他程序接收 ALARM_SNOOZE_ACTION / ALARM_DISMISS_ACTION
    private BroadcastReceiver mReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            if (action.equals(Alarms.ALARM_SNOOZE_ACTION)) {
                snooze();
            } else if (action.equals(Alarms.ALARM_DISMISS_ACTION)) {
                dismiss(false);
            } else {
                Alarm alarm = intent.getParcelableExtra(Alarms.ALARM_INTENT_EXTRA);
                if (alarm != null && mAlarm.id == alarm.id) {
                    dismiss(true);
                }
            }
        }
    };

    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        mAlarm = getIntent().getParcelableExtra(Alarms.ALARM_INTENT_EXTRA);

        // 获取声音和相机设置
        final String vol =
                PreferenceManager.getDefaultSharedPreferences(this)
                .getString(SettingsActivity.KEY_VOLUME_BEHAVIOR,
                        DEFAULT_VOLUME_BEHAVIOR);
        mVolumeBehavior = Integer.parseInt(vol);

        requestWindowFeature(android.view.Window.FEATURE_NO_TITLE);

        final Window win = getWindow();
```

```
        win.addFlags(WindowManager.LayoutParams.FLAG_SHOW_WHEN_LOCKED
                | WindowManager.LayoutParams.FLAG_DISMISS_KEYGUARD);
        // 打开屏幕
        if (!getIntent().getBooleanExtra(SCREEN_OFF, false)) {
            win.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON
                    | WindowManager.LayoutParams.FLAG_TURN_SCREEN_ON
                    | WindowManager.LayoutParams.FLAG_ALLOW_LOCK_WHILE_SCREEN_ON
                    );
        }

        updateLayout();

        //3 个闹钟按钮“关闭”、“睡眠”、“解散”的处理
        IntentFilter filter = new IntentFilter(Alarms.ALARM_KILLED);
        filter.addAction(Alarms.ALARM_SNOOZE_ACTION);
        filter.addAction(Alarms.ALARM_DISMISS_ACTION);
        registerReceiver(mReceiver, filter);
    }

    private void setTitle() {
        String label = mAlarm.getLabelOrDefault(this);
        TextView title = (TextView) findViewById(R.id.alertTitle);
        title.setText(label);
    }

    private void updateLayout() {
        LayoutInflater inflater = LayoutInflater.from(this);

        setContentView(inflater.inflate(R.layout.alarm_alert, null));

        /* snooze behavior: pop a snooze confirmation view, kick alarm
          manager. */
        Button snooze = (Button) findViewById(R.id.snooze);
        snooze.requestFocus();
        snooze.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                snooze();
            }
        });

        /* dismiss 按钮：关闭提醒*/
        findViewById(R.id.dismiss).setOnClickListener(
                new Button.OnClickListener() {
                    public void onClick(View v) {
                        dismiss(false);
                    }
                });

        /*在闹钟设置标题 */
        setTitle();
    }

    //暂停此警报
    private void snooze() {
        //不要贪睡，snooze 按钮被禁用
        if (!findViewById(R.id.snooze).isEnabled()) {
            dismiss(false);
            return;
        }
        final String snooze =
                PreferenceManager.getDefaultSharedPreferences(this)
                .getString(SettingsActivity.KEY_ALARM_SNOOZE, DEFAULT_SNOOZE);
        int snoozeMinutes = Integer.parseInt(snooze);

        final long snoozeTime = System.currentTimeMillis()
                + (1000 * 60 * snoozeMinutes);
        Alarms.saveSnoozeAlert(AlarmAlertFullScreen.this, mAlarm.id,
                snoozeTime);

        //根据 snooze 和 update 的设置得到显示时间
```

```
        final Calendar c = Calendar.getInstance();
        c.setTimeInMillis(snoozeTime);

        // Append (snoozed) to the label
        String label = mAlarm.getLabelOrDefault(this);
        label = getString(R.string.alarm_notify_snooze_label, label);

        //通知用户，闹钟被暂停
        Intent cancelSnooze = new Intent(this, AlarmReceiver.class);
        cancelSnooze.setAction(Alarms.CANCEL_SNOOZE);
        cancelSnooze.putExtra(Alarms.ALARM_ID, mAlarm.id);
        PendingIntent broadcast =
                PendingIntent.getBroadcast(this, mAlarm.id, cancelSnooze, 0);
        NotificationManager nm = getNotificationManager();
        Notification n = new Notification(R.drawable.stat_notify_alarm,
                label, 0);
        n.setLatestEventInfo(this, label,
                getString(R.string.alarm_notify_snooze_text,
                    Alarms.formatTime(this, c)), broadcast);
        n.flags |= Notification.FLAG_AUTO_CANCEL
                | Notification.FLAG_ONGOING_EVENT;
        nm.notify(mAlarm.id, n);

        String displayTime = getString(R.string.alarm_alert_snooze_set,
                snoozeMinutes);
        //故意延迟小睡的时间
        Log.v(displayTime);

        // 在提醒中显示小睡时间
        Toast.makeText(AlarmAlertFullScreen.this, displayTime,
                Toast.LENGTH_LONG).show();
        stopService(new Intent(Alarms.ALARM_ALERT_ACTION));
        finish();
    }

    private NotificationManager getNotificationManager() {
        return (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    }

    // Dismiss 闹钟
    private void dismiss(boolean killed) {
        Log.i(killed ? "Alarm killed" : "Alarm dismissed by user");
        // The service told us that the alarm has been killed, do not modify
        // the notification or stop the service
        if (!killed) {
            // Cancel the notification and stop playing the alarm
            NotificationManager nm = getNotificationManager();
            nm.cancel(mAlarm.id);
            stopService(new Intent(Alarms.ALARM_ALERT_ACTION));
        }
        finish();
    }
    @Override
    protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        if (Log.LOGV) Log.v("AlarmAlert.OnNewIntent()");
        mAlarm = intent.getParcelableExtra(Alarms.ALARM_INTENT_EXTRA);
        setTitle();
    }
    @Override
    protected void onResume() {
        super.onResume();
        // If the alarm was deleted at some point, disable snooze
        if (Alarms.getAlarm(getContentResolver(), mAlarm.id) == null) {
            Button snooze = (Button) findViewById(R.id.snooze);
            snooze.setEnabled(false);
        }
    }

    @Override
```

```
    public void onDestroy() {
        super.onDestroy();
        if (Log.LOGV) Log.v("AlarmAlert.onDestroy()");
        // No longer care about the alarm being killed
        unregisterReceiver(mReceiver);
    }

    @Override
    public boolean dispatchKeyEvent(KeyEvent event) {
        // Do this on key down to handle a few of the system keys
        boolean up = event.getAction() == KeyEvent.ACTION_UP;
        switch (event.getKeyCode()) {
            // Volume keys and camera keys dismiss the alarm
            case KeyEvent.KEYCODE_VOLUME_UP:
            case KeyEvent.KEYCODE_VOLUME_DOWN:
            case KeyEvent.KEYCODE_CAMERA:
            case KeyEvent.KEYCODE_FOCUS:
                if (up) {
                    switch (mVolumeBehavior) {
                        case 1:
                            snooze();
                            break;

                        case 2:
                            dismiss(false);
                            break;

                        default:
                            break;
                    }
                }
                return true;
            default:
                break;
        }
        return super.dispatchKeyEvent(event);
    }
```

24.3.4 重复设置

在系统中设置一个闹钟时，可以设置这个闹钟在其他时间也起作用，如每个周一或周二。此功能是通过文件 RepeatPreference.java 实现的，具体实现代码如下所示。

```
public class RepeatPreference extends ListPreference {
    // Initial value that can be set with the values saved in the database
    private Alarm.DaysOfWeek mDaysOfWeek = new Alarm.DaysOfWeek(0);
    // New value that will be set if a positive result comes back from the
    // dialog
    private Alarm.DaysOfWeek mNewDaysOfWeek = new Alarm.DaysOfWeek(0);

    public RepeatPreference(Context context, AttributeSet attrs) {
        super(context, attrs);

        String[] weekdays = new DateFormatSymbols().getWeekdays();
        String[] values = new String[] {
            weekdays[Calendar.MONDAY],
            weekdays[Calendar.TUESDAY],
            weekdays[Calendar.WEDNESDAY],
            weekdays[Calendar.THURSDAY],
            weekdays[Calendar.FRIDAY],
            weekdays[Calendar.SATURDAY],
            weekdays[Calendar.SUNDAY],
        };
        setEntries(values);
        setEntryValues(values);
    }

    @Override
```

```
    protected void onDialogClosed(boolean positiveResult) {
        if (positiveResult) {
            mDaysOfWeek.set(mNewDaysOfWeek);
            setSummary(mDaysOfWeek.toString(getContext(), true));
            callChangeListener(mDaysOfWeek);
        }
    }

    @Override
    protected void onPrepareDialogBuilder(Builder builder) {
        CharSequence[] entries = getEntries();
        @SuppressWarnings("unused")
        CharSequence[] entryValues = getEntryValues();


        builder.setMultiChoiceItems(
                entries, mDaysOfWeek.getBooleanArray(),
                new DialogInterface.OnMultiChoiceClickListener() {
                    public void onClick(DialogInterface dialog, int which,
                            boolean isChecked) {
                        mNewDaysOfWeek.set(which, isChecked);
                    }
                });
    }
    public void setDaysOfWeek(Alarm.DaysOfWeek dow) {
        mDaysOfWeek.set(dow);
        mNewDaysOfWeek.set(dow);
        setSummary(dow.toString(getContext(), true));
    }
    public Alarm.DaysOfWeek getDaysOfWeek() {
        return mDaysOfWeek;
    }
}
```

闹钟重复设置界面的执行效果如图 24-7 所示。

▲图 24-7　闹钟重复设置界面的执行效果

24.3.5　闹钟数据操作

在本系统中，采用了 SQLiteDatabase 数据库来保存系统中的数据，如每个闹钟的时间、重复、铃声和振动等信息。和数据操作相关的程序文件是 AlarmProvider.java，具体实现代码如下所示。

```
public class AlarmProvider extends ContentProvider
{
    private SQLiteOpenHelper mOpenHelper;

    private static final int ALARMS = 1;
    private static final int ALARMS_ID = 2;
    private static final UriMatcher sURLMatcher = new UriMatcher(
            UriMatcher.NO_MATCH);

    static {
        sURLMatcher.addURI("com.android.superdeskclock", "alarm", ALARMS);
        sURLMatcher.addURI("com.android.superdeskclock", "alarm/#", ALARMS_ID);
    }

    private static class DatabaseHelper extends SQLiteOpenHelper {
        private static final String DATABASE_NAME = "alarms.db";
        private static final int DATABASE_VERSION = 5;

        public DatabaseHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }
//创建保存数据库中的每一个选项
        @Override
```

```
        public void onCreate(SQLiteDatabase db) {
            db.execSQL("CREATE TABLE alarms (" +
                    "_id INTEGER PRIMARY KEY," +
                    "hour INTEGER, " +
                    "minutes INTEGER, " +
                    "daysofweek INTEGER, " +
                    "alarmtime INTEGER, " +
                    "enabled INTEGER, " +
                    "vibrate INTEGER, " +
                    "message TEXT, " +
                    "alert TEXT, " +
                    "times INTEGER, " +
                    "interval INTEGER);");

            // 插入闹钟数据
            String insertMe = "INSERT INTO alarms " +
                  "(hour, minutes, daysofweek, alarmtime, enabled, vibrate, message,
                  alert,times,interval) " + "VALUES ";
            db.execSQL(insertMe + "(8, 30, 31, 0, 0, 1, '', '',10,0);");
            db.execSQL(insertMe + "(9, 00, 96, 0, 0, 1, '', '',10,0);");
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int currentVersion) {
            if (Log.LOGV) Log.v(
                  "Upgrading alarms database from version " +
                  oldVersion + " to " + currentVersion +
                  ", which will destroy all old data");
            db.execSQL("DROP TABLE IF EXISTS alarms");
            onCreate(db);
        }
    }

    public AlarmProvider() {
    }

    @Override
    public boolean onCreate() {
        mOpenHelper = new DatabaseHelper(getContext());
        return true;
    }
//查询系统中的闹钟数据
    @Override
    public Cursor query(Uri url, String[] projectionIn, String selection,
          String[] selectionArgs, String sort) {
        SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

        // Generate the body of the query
        int match = sURLMatcher.match(url);
        switch (match) {
            case ALARMS:
                qb.setTables("alarms");
                break;
            case ALARMS_ID:
                qb.setTables("alarms");
                qb.appendWhere("_id=");
                qb.appendWhere(url.getPathSegments().get(1));
                break;
            default:
                throw new IllegalArgumentException("Unknown URL " + url);
        }

        SQLiteDatabase db = mOpenHelper.getReadableDatabase();
        Cursor ret = qb.query(db, projectionIn, selection, selectionArgs,
                     null, null, sort);

        if (ret == null) {
            if (Log.LOGV) Log.v("Alarms.query: failed");
        } else {
```

```
            ret.setNotificationUri(getContext().getContentResolver(), url);
        }

        return ret;
    }
//获取类型信息
    @Override
    public String getType(Uri url) {
        int match = sURLMatcher.match(url);
        switch (match) {
            case ALARMS:
                return "vnd.android.cursor.dir/alarms";
            case ALARMS_ID:
                return "vnd.android.cursor.item/alarms";
            default:
                throw new IllegalArgumentException("Unknown URL");
        }
    }
//更新系统中的闹钟信息
    @Override
    public int update(Uri url, ContentValues values, String where, String[] whereArgs)
{
        int count;
        long rowId = 0;
        int match = sURLMatcher.match(url);
        SQLiteDatabase db = mOpenHelper.getWritableDatabase();
        switch (match) {
            case ALARMS_ID: {
                String segment = url.getPathSegments().get(1);
                rowId = Long.parseLong(segment);
                count = db.update("alarms", values, "_id=" + rowId, null);
                break;
            }
            default: {
                throw new UnsupportedOperationException(
                        "Cannot update URL: " + url);
            }
        }
        if (Log.LOGV) Log.v("*** notifyChange() rowId: " + rowId + " url " + url);
        getContext().getContentResolver().notifyChange(url, null);
        return count;
    }

    @Override
    public Uri insert(Uri url, ContentValues initialValues) {
        if (sURLMatcher.match(url) != ALARMS) {
            throw new IllegalArgumentException("Cannot insert into URL: " + url);
        }

        ContentValues values = new ContentValues(initialValues);

        SQLiteDatabase db = mOpenHelper.getWritableDatabase();
        long rowId = db.insert("alarms", Alarm.Columns.MESSAGE, values);
        if (rowId < 0) {
            throw new SQLException("Failed to insert row into " + url);
        }
        if (Log.LOGV) Log.v("Added alarm rowId = " + rowId);

        Uri newUrl = ContentUris.withAppendedId(Alarm.Columns.CONTENT_URI, rowId);
        getContext().getContentResolver().notifyChange(newUrl, null);
        return newUrl;
    }
//删除信息
    public int delete(Uri url, String where, String[] whereArgs) {
        SQLiteDatabase db = mOpenHelper.getWritableDatabase();
        int count;
        long rowId = 0;
        switch (sURLMatcher.match(url)) {
            case ALARMS:
```

```
            count = db.delete("alarms", where, whereArgs);
            break;
        case ALARMS_ID:
            String segment = url.getPathSegments().get(1);
            rowId = Long.parseLong(segment);
            if (TextUtils.isEmpty(where)) {
                where = "_id=" + segment;
            } else {
                where = "_id=" + segment + " AND (" + where + ")";
            }
            count = db.delete("alarms", where, whereArgs);
            break;
        default:
            throw new IllegalArgumentException("Cannot delete from URL: " + url);
        }
        getContext().getContentResolver().notifyChange(url, null);
        return count;
    }
}
```

24.4 选择铃声音乐

在图 24-2 所示的界面中，如果按下♫图标则会获取 SD 卡中的音乐文件，在里面可以选择一个音乐文件作为闹钟铃声。此功能是通过文件 ChooseBellActivity.java 实现的，具体实现代码如下所示。

```
public class SetBellPreference extends ListPreference{
    private Uri mAlert;
    private PreferenceActivity preferenceActivity;
    private int mId;
    public SetBellPreference(Context context, AttributeSet attrs) {
        super(context, attrs);
        this.preferenceActivity=(PreferenceActivity)context;
        String[] values = context.getResources().getStringArray(R.array.choose_bell);
        mId=preferenceActivity.getIntent().getIntExtra(Alarms.ALARM_ID, -1);
        setEntries(values);
        setEntryValues(values);
    }


    @Override
    protected void onPrepareDialogBuilder(Builder builder) {
        CharSequence[] entries = getEntries();

        builder.setSingleChoiceItems(entries, -1, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                switch (which) {
                    case 0:
                        break;
                    case 1:
                        Intent intent=new Intent(preferenceActivity,ChooseBellActivity. class);
                        intent.putExtra(Alarms.ALARM_ID, mId);
                        intent.putExtra("TYPE", 1);
                        preferenceActivity.startActivity(intent);
                        preferenceActivity.finish();
                        break;
                    case 2:
                        Intent intent2=new Intent(preferenceActivity,ChooseBellActivity. class);
                        intent2.putExtra(Alarms.ALARM_ID, mId);
                        intent2.putExtra("TYPE", 2);
                        preferenceActivity.startActivity(intent2);
                        preferenceActivity.finish();
                        break;
                    default:
                        break;
                }
```