百炼成钢之Android经典畅销书

Android Studio 开发实战

从零基础到App上线（第3版）

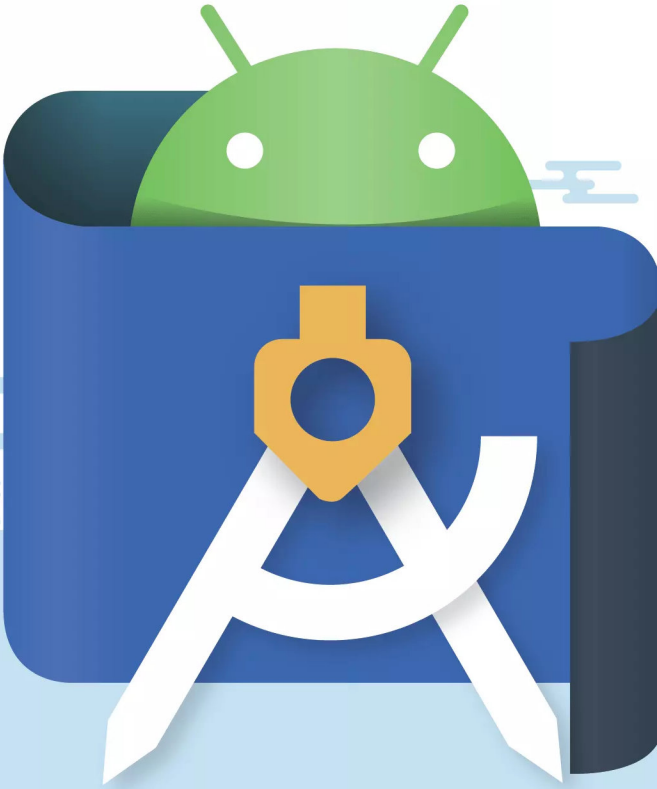
基于Android Studio Dolphin / Android 12版本

欧阳燊 / 著

PPT课件

程序源码

思维导图



移动开发丛书

17款流行App全程详解，每一行代码都有注释

清华大学出版社

移动开发丛书

开发实战 Android Studio

从零基础到App上线（第3版）

欧阳燊 / 著



清華大學出版社
北京

内 容 简 介

本书是一部 Android 开发的实战教程，由浅入深、由基础到高级，带领读者一步一步走进 App 开发的神奇世界。全书共分为 20 章。其中，前 10 章是基础部分，主要讲解 Android Studio 的环境搭建、App 开发的各种常用控件、App 开发的四大组件、App 的数据存储方式、如何打造 App 的安装包；后 10 章是进阶部分，主要讲解 App 开发的事件交互、动画特效、网络通信、多媒体、手机阅读、感知定位、物联网、智能语音、机器视觉、在线直播等。书中在讲解知识点的同时给出了大量实战范例，方便读者迅速将所学的知识运用到实际开发中。通过本书的学习，读者不仅能够掌握 3 类主流 App 的基本开发技术，包括购物 App（电子商务）、聊天 App（即时通信）、娱乐 App（短视频分享），还能够学会开发一些趣味应用，包括计算器、记账本、抠图工具、动感影集、电子阅读、附近交友、遥控小车、语音问答、人脸识别、视频通话等。

本书适用于 Android 开发的广大从业者、有志于转型 App 开发的程序员、App 开发的业余爱好者，也可作为大中专院校与培训机构的 Android 课程教材。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。举报：010-62782989，beiqinquan@tup.tsinghua.edu.cn。

图书在版编目（CIP）数据

Android Studio 开发实战：从零基础到 App 上线/欧阳燊著. —3 版. —北京：清华大学出版社，2022.9
（移动开发丛书）
ISBN 978-7-302-61782-2

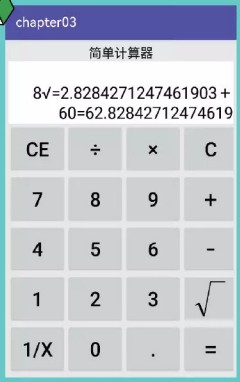
Ⅰ. ①A… Ⅱ. ①欧… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2022）第 161868 号

责任编辑：王金柱
封面设计：王　翔
责任校对：闫秀华
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-83470000　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市君旺印务有限公司
经　　销：全国新华书店
开　　本：190mm×260mm　　**彩　　插：**2　　**印　　张：**40.75　　**字　　数：**1099 千字
版　　次：2017 年 6 月第 1 版　2022 年 10 月第 3 版　　**印　　次：**2022 年 10 月第 1 次印刷
定　　价：148.80 元

产品编号：096100-01



第3章 计算器效果图



第6章 购物车效果图1



第6章 购物车效果图2



第5章 找回密码效果图1



第8章 广告轮播效果图1



第8章 广告轮播效果图2



第9章 电商首页效果图1



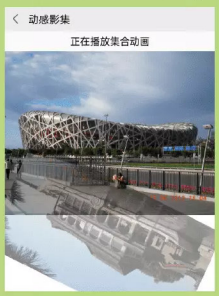
第9章 电商首页效果图2



第11章 抠图工具效果图



第12章 动感影集效果图



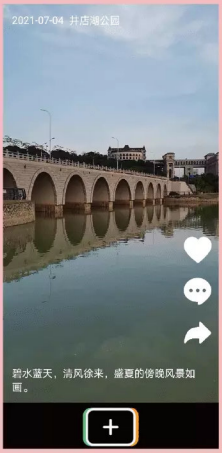


第13章 即时通信效果图1



第13章 即时通信效果图2





第14章 短视频分享效果图1



第14章 短视频分享效果图2

第15章 电子书架效果图1

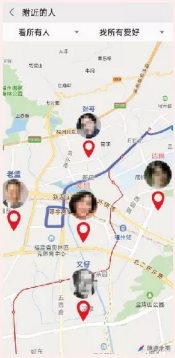


第15章 电子书架效果图2

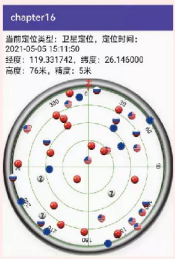




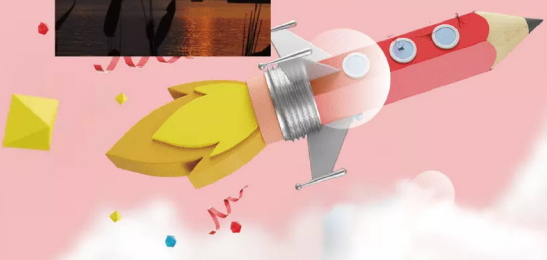
第16章 附近的人效果图1



第16章 附近的人效果图2



第16章 卫星浑天仪效果图





第17章 智能小车效果图



第18章 问答机器人效果图



第19章 智慧天眼效果图



第20章 视频通话效果图1



第20章 视频通话效果图2



第20章 直播带货效果图

推 荐 序

计算机的发展是以信息智能化与小型化为进化路线，从 IBM 庞大的巨型机到比尔盖茨的个人电脑，信息无所不在。乔布斯的伟大之处在于“用一个手指头改变世界”。当全世界的粉丝用苹果手机的时候，移动开发领域开始全面地封闭在 iOS 的体系里。安卓作为移动手机和设备开放象征的另一级，更具有活力和前途。

欧阳先生是一位具有丰富程序开发经验的架构师和项目管理者，平时常常思考和总结 21 世纪以来我国软件开发者，特别是移动开发工程师的困惑。社会从“一支笔的科学家时代”发展到“一个键盘开发 App 改变世界”，对程序员来说，用自己的智慧进行移动应用开发是创业的捷径。读者遵循书中的指引，很快能够登堂入室，成为当前安卓应用开发的精英人才。

本书对所有有志于进行安卓系统开发的人员而言具有非常重要的意义。

杭州海适云承科技有限公司

董事长兼首席架构师

沈英桓

前　言

本书第2版面世已近四年，在此期间信息科技继续快速发展，5G网络大规模建成，智能化应用越发普及。万物互联、智慧城市、见屏如见面等场景已从早些年的概念变成如今的生活常态，这些都离不开移动互联网、物联网、人工智能等技术的助力。

单就App开发而言，安卓的系统版本从2018年的Android 9升级到2021年的Android 12，Android Studio的版本也从2018年的3.2更新到2021年的4.2，然后开始以字母命名，同时操作系统又多了个兼容Android的国产鸿蒙。近几年还诞生了一些新框架和新标准，诸如谷歌公司的JetPack套件、华为公司的HMS套件，以及臻于完善的WebRTC、低功耗蓝牙（简称BLE）等，这些技术都在App开发中得到越来越广泛的运用。

有鉴于此，本书急需补充这期间风起云涌的新技术新知识，以跟上时代发展的滚滚浪潮。种种因缘际会，加上众多读者的热忱建议，因此便有了重新修订之后的第3版问世。第3版的图书不是第2版的简单修补，而是脱胎换骨的全面升级。与第2版相比，第3版的图书主要有以下五处重要的增补变化。

1. 工具更新颖

第3版的App开发全部基于Android 12环境，使用的开发工具为2022年9月发布的Android Studio Dolphin（小海豚版本），JNI用到的NDK则为2021年8月发布的r23。相关的功能点都根据上述最新版本的工具展开论述，比如Android 12新增支持AVIF图像、强制使用64位应用，JetPack库内含的DataStore、Room、ViewPager2、CameraX等流行组件。

2. 技术更先进

移动互联网的后继发展方向如物联网、人工智能、实时音视频等开展得如火如荼，第3版为此投入了大量笔墨深入描述相关的技术细节，例如物联网涉及的NFC、红外、低功耗蓝牙、快速扫码等，人工智能涉及的语音识别、语音合成、人脸识别、活体检测等，实时音视频涉及的WebSocket、WebRTC、原始音频、视频通话等。

3. 案例更实用

本书的一大特色是突出实战，每章末尾都给出了技术精炼的实战项目，第3版更是将这个优良传统发扬光大。一方面给部分现有项目补充新功能，例如聊天App原来只支持私聊，新版本增加了群聊功能；阅读App除了支持现有格式外，改造后还支持Office文档（含Word、Excel、PPT）。另一方面新增了若干新潮实用的实战项目，包括无线遥控的智能小车、你问我答的智能语音机器人、自动寻人的智慧天眼、爱心助农的手机直播等。

4. 自主更可控

每个App都凝聚着开发者的心血，持续稳定运行是每个开发者的期望。然而有时由于外部原因导致App无法使用，例如国外服务遭到断供、不在服务范围之内等。为此第3版推荐使用稳定可靠的国内自主服务，包括HMS（Huawei Mobile Service，华为移动服务）、TBS（Tencent Browser Service，腾讯浏览服务）。另外，语音服务和地图服务也选用了方便快速集成的第三方SDK。

5. 讲解更精细

作为一部软件开发方面的专著，要让读者看得懂、学得会，循序渐进的重要性毋庸置疑。第3版在这方面进行了大力改善，首先增加了单独的第2章作为入门，先讲解App开发必备的基础知识，有助于读者打好基础往下学习；其次，对于App开发的四大组件（Activity、Broadcast、Service、ContentProvider），补充了更加全面的描述以及各组件在不同Android版本中需要注意的事项；最后，大幅精简示例源码，加了注释的每段源码基本不超过一百行，避免代码过长引起的疲惫感。

综上所述，经过精心修订的第3版图书，无论是广度还是深度，从数量到质量都比第2版有了质的提升。全书的写作目的，不仅是教会读者怎么快速开发一个好玩、好看、好用的App，更是让读者领略行业前沿的移动互联网技术。深度解密流行App背后的手机开发技术，展示移动信息科技的最新工程实践，这才是本书一直想要呈献给读者的知识盛宴。

第3版的主要代码基于Java 8编写，其中客户端的App代码基于Android Studio Dolphin（小海豚版本）开发，并使用API31的SDK（Android 12）编译与调试通过，同时兼容鸿蒙2.0设备。配套的服务端源码采用Java Web框架，结合MySQL数据库，并基于IDEA开发。

读者可以扫描下述二维码获取本书的源码、思维导图和PPT课件：







如果下载有问题，请发送邮件到booksaga@126.com，邮件主题为“Android Studio开发实战：从零基础到App上线（第3版）”。

最后，感谢王金柱编辑的热情指点，感谢出版社同仁的辛勤工作，感谢我的家人一直以来的支持，感谢各位师长的谆谆教导，没有他们的鼎力相助，本书就无法顺利完成。

欧阳燊

2022 年 7 月

目　录

第 1 章　Android 开发环境搭建 …… 1

1.1 Android 开发简介 …… 1
1.1.1 Android的发展历程 …… 1
1.1.2 Android Studio的发展历程 …… 2
1.2 搭建 Android Studio 开发环境 …… 2
1.2.1 计算机配置要求 …… 2
1.2.2 安装Android Studio …… 3
1.2.3 下载Android的SDK …… 6
1.3 创建并编译 App 工程 …… 7
1.3.1 创建新项目 …… 7
1.3.2 导入已有的工程 …… 9
1.3.3 编译App工程 …… 10
1.4 运行和调试 App …… 11
1.4.1 创建内置模拟器 …… 11
1.4.2 在模拟器上运行App …… 14
1.4.3 观察App的运行日志 …… 15
1.5 小结 …… 16
1.6 动手练习 …… 16

第 2 章　Android App 开发基础 …… 17

2.1 App 的开发特点 …… 17
2.1.1 App的运行环境 …… 17
2.1.2 App的开发语言 …… 19
2.1.3 App连接的数据库 …… 22
2.2 App 的工程结构 …… 23
2.2.1 App工程目录结构 …… 23
2.2.2 编译配置文件build.gradle …… 24
2.2.3 运行配置文件 AndroidManifest.xml …… 26
2.3 App 的设计规范 …… 27
2.3.1 界面设计与代码逻辑 …… 28
2.3.2 利用XML标记描绘应用界面 …… 29
2.3.3 使用Java代码书写程序逻辑 …… 30
2.4 App 的活动页面 …… 31
2.4.1 创建新的App页面 …… 31
2.4.2 快速生成页面源码 …… 33
2.4.3 跳转到另一个页面 …… 34
2.5 小结 …… 35
2.6 动手练习 …… 36

第 3 章　简单控件 …… 37

3.1 文本显示 …… 37
3.1.1 设置文本的内容 …… 37
3.1.2 设置文本的大小 …… 38
3.1.3 设置文本的颜色 …… 40
3.2 视图基础 …… 42
3.2.1 设置视图的宽和高 …… 42
3.2.2 设置视图的间距 …… 44
3.2.3 设置视图的对齐方式 …… 45
3.3 常用布局 …… 46
3.3.1 线性布局LinearLayout …… 46
3.3.2 相对布局RelativeLayout …… 49
3.3.3 网格布局GridLayout …… 51
3.3.4 滚动视图ScrollView …… 52
3.4 按钮触控 …… 53
3.4.1 按钮控件Button …… 53
3.4.2 点击事件和长按事件 …… 55
3.4.3 禁用与恢复按钮 …… 56
3.5 图像显示 …… 58
3.5.1 图像视图ImageView …… 58
3.5.2 图像按钮ImageButton …… 60
3.5.3 同时展示文本与图像 …… 61
3.6 实战项目：计算器 …… 62
3.6.1 需求描述 …… 62
3.6.2 界面设计 …… 62
3.6.3 关键代码 …… 63
3.7 小结 …… 65
3.8 动手练习 …… 65

第 4 章　常用组件 …… 66

4.1 启停活动页面 …… 66
4.1.1 Activity的启动和结束 …… 66
4.1.2 Activity的生命周期 …… 67

4.1.3 Activity的启动模式……70
4.2 在活动之间传递消息 …… 73
4.2.1 显式Intent和隐式Intent ……73
4.2.2 普通的活动数据交互 ……75
4.2.3 改进后的活动数据交互 ……78
4.3 收发应用广播…… 80
4.3.1 收发标准广播 ……80
4.3.2 收发有序广播 ……83
4.3.3 收发静态广播 ……84
4.3.4 定时管理器AlarmManager……86
4.4 操作后台服务…… 89
4.4.1 服务的启动和停止 ……89
4.4.2 服务的绑定与解绑 ……92
4.4.3 活动与服务之间的交互 ……94
4.5 小结…… 96
4.6 动手练习…… 96

第 5 章 中级控件……97

5.1 图形定制…… 97
5.1.1 图形Drawable……97
5.1.2 形状图形 ……98
5.1.3 九宫格图片 …… 101
5.1.4 状态列表图形 …… 102
5.2 选择按钮…… 103
5.2.1 复选框CheckBox…… 103
5.2.2 开关按钮Switch …… 105
5.2.3 单选按钮RadioButton …… 106
5.3 文本输入…… 108
5.3.1 编辑框EditText…… 108
5.3.2 焦点变更监听器 …… 110
5.3.3 文本变化监听器 …… 111
5.4 对话框…… 113
5.4.1 提醒对话框AlertDialog …… 113
5.4.2 日期对话框DatePickerDialog…… 115
5.4.3 时间对话框TimePickerDialog …… 116
5.5 实战项目：找回密码 …… 117
5.5.1 需求描述 …… 117
5.5.2 界面设计 …… 118
5.5.3 关键代码 …… 119
5.6 小结…… 121
5.7 动手练习…… 121

第 6 章 数据存储……122

6.1 键值对 ……122
6.1.1 共享参数的用法 ……122
6.1.2 实现记住密码功能 ……123
6.1.3 更安全的数据仓库 ……124
6.2 数据库 ……126
6.2.1 SQL的基本语法 ……127
6.2.2 数据库管理器 SQLiteDatabase ……129
6.2.3 数据库帮助器 SQLiteOpenHelper……130
6.2.4 优化记住密码功能 ……133
6.3 存储卡 ……135
6.3.1 私有存储空间与公共存储空间 ……135
6.3.2 在存储卡上读写文件 ……136
6.3.3 运行时动态申请权限 ……139
6.4 应用组件 Application……142
6.4.1 Application的生命周期 ……142
6.4.2 利用Application操作全局变量 ……143
6.4.3 避免方法数过多的问题 ……144
6.4.4 利用Room简化数据库操作……145
6.5 共享数据 ……148
6.5.1 通过ContentProvider封装数据 ……148
6.5.2 通过ContentResolver访问数据 ……151
6.5.3 利用ContentResolver读写联系人 ……153
6.5.4 利用ContentObserver监听短信 ……154
6.6 实战项目：购物车……156
6.6.1 需求描述 ……156
6.6.2 界面设计 ……158
6.6.3 关键代码 ……159
6.7 小结……163
6.8 动手练习 ……163

第 7 章 高级控件……164

7.1 下拉框 ……164

7.1.1 下拉框控件Spinner …… 164
7.1.2 数组适配器ArrayAdapter …… 166
7.1.3 简单适配器SimpleAdapter …… 166
7.2 列表类视图 …… 168
7.2.1 基本适配器BaseAdapter …… 168
7.2.2 列表视图ListView …… 170
7.2.3 网格视图GridView …… 174
7.3 翻页类视图 …… 176
7.3.1 翻页视图ViewPager …… 176
7.3.2 翻页标签栏PagerTabStrip …… 179
7.3.3 简单的启动引导页 …… 180
7.4 碎片 Fragment …… 183
7.4.1 碎片的静态注册 …… 183
7.4.2 碎片的动态注册 …… 185
7.4.3 改进的启动引导页 …… 188
7.5 实战项目：记账本 …… 190
7.5.1 需求描述 …… 190
7.5.2 界面设计 …… 191
7.5.3 关键代码 …… 192
7.6 小结 …… 195
7.7 动手练习 …… 195

第 8 章 自定义控件 …… 196

8.1 视图的构建过程 …… 196
8.1.1 视图的构造方法 …… 196
8.1.2 视图的测量方法 …… 199
8.1.3 视图的绘制方法 …… 202
8.2 改造已有的控件 …… 205
8.2.1 自定义月份选择器 …… 205
8.2.2 给翻页标签栏添加新属性 …… 206
8.2.3 不滚动的列表视图 …… 208
8.3 推送消息通知 …… 210
8.3.1 通知推送Notification …… 210
8.3.2 通知渠道NotificationChannel …… 213
8.3.3 推送服务到前台 …… 215
8.3.4 仿微信的悬浮通知 …… 216
8.4 通过持续绘制实现简单动画 …… 221
8.4.1 Handler的延迟机制 …… 221
8.4.2 重新绘制视图界面 …… 222
8.4.3 自定义饼图动画 …… 224
8.5 实战项目：广告轮播 …… 227
8.5.1 需求描述 …… 227
8.5.2 界面设计 …… 227
8.5.3 关键代码 …… 228
8.6 小结 …… 231
8.7 动手练习 …… 231

第 9 章 组合控件 …… 232

9.1 底部标签栏 …… 232
9.1.1 利用BottomNavigationView实现底部标签栏 …… 232
9.1.2 自定义标签按钮 …… 236
9.1.3 结合RadioGroup和ViewPager自定义底部标签栏 …… 238
9.2 顶部导航栏 …… 241
9.2.1 工具栏Toolbar …… 241
9.2.2 溢出菜单OverflowMenu …… 242
9.2.3 标签布局TabLayout …… 244
9.3 增强型列表 …… 246
9.3.1 循环视图RecyclerView …… 247
9.3.2 布局管理器LayoutManager …… 249
9.3.3 动态更新循环视图 …… 252
9.4 升级版翻页 …… 254
9.4.1 下拉刷新布局SwipeRefreshLayout …… 254
9.4.2 第二代翻页视图ViewPager2 …… 256
9.4.3 给ViewPager2集成标签布局 …… 259
9.5 实战项目：电商首页 …… 261
9.5.1 需求描述 …… 261
9.5.2 界面设计 …… 262
9.5.3 关键代码 …… 263
9.6 小结 …… 264
9.7 动手练习 …… 264

第 10 章 打造安装包 …… 265

10.1 应用打包 …… 265
10.1.1 导出APK安装包 …… 265
10.1.2 制作App图标 …… 267
10.1.3 给APK瘦身 …… 268
10.2 规范处理 …… 271
10.2.1 版本设置 …… 271
10.2.2 发布模式 …… 272
10.2.3 多渠道打包 …… 274
10.3 安全加固 …… 276

10.3.1 反编译 ····· 276
10.3.2 代码混淆 ····· 278
10.3.3 第三方加固及重签名 ····· 279
10.4 小结 ····· 280
10.5 动手练习 ····· 280

第 11 章 事件交互 ····· 281

11.1 按键事件 ····· 281
11.1.1 检测软键盘 ····· 281
11.1.2 检测物理按键 ····· 283
11.1.3 接管返回按键 ····· 284
11.2 触摸事件 ····· 285
11.2.1 手势事件的分发流程 ····· 285
11.2.2 接管手势事件处理 ····· 289
11.2.3 跟踪滑动轨迹实现手写签名 ····· 292
11.3 根据触摸行为辨别手势动作 ····· 293
11.3.1 区分点击和长按动作 ····· 293
11.3.2 识别手势滑动的方向 ····· 295
11.3.3 辨别缩放与旋转手势 ····· 297
11.4 手势冲突处理 ····· 300
11.4.1 上下滚动与左右滑动的冲突处理 ····· 300
11.4.2 内部滑动与翻页滑动的冲突处理 ····· 302
11.4.3 正常下拉与下拉刷新的冲突处理 ····· 306
11.5 实战项目：仿美图秀秀的抠图工具 ····· 309
11.5.1 需求描述 ····· 309
11.5.2 功能分析 ····· 309
11.5.3 效果展示 ····· 310
11.6 小结 ····· 311
11.7 动手练习 ····· 311

第 12 章 动画特效 ····· 312

12.1 帧动画 ····· 312
12.1.1 帧动画的实现 ····· 312
12.1.2 显示动图特效 ····· 314
12.1.3 淡入淡出动画 ····· 317
12.2 补间动画 ····· 318
12.2.1 补间动画的种类 ····· 318
12.2.2 补间动画的原理 ····· 322
12.2.3 集合动画 ····· 324
12.3 属性动画 ····· 325
12.3.1 常规的属性动画 ····· 325
12.3.2 属性动画组合 ····· 328
12.3.3 插值器和估值器 ····· 329
12.3.4 利用估值器实现打赏动画 ····· 332
12.4 遮罩动画及滚动器 ····· 335
12.4.1 画布的绘图层次 ····· 335
12.4.2 实现百叶窗动画 ····· 338
12.4.3 利用滚动器实现平滑翻页 ····· 342
12.5 实战项目：仿手机 QQ 的动感影集 ····· 344
12.5.1 需求描述 ····· 345
12.5.2 功能分析 ····· 346
12.5.3 效果展示 ····· 349
12.6 小结 ····· 350
12.7 动手练习 ····· 350

第 13 章 网络通信 ····· 351

13.1 多线程 ····· 351
13.1.1 分线程通过Handler操作界面 ····· 351
13.1.2 通过runOnUiThread快速操纵界面 ····· 354
13.1.3 工作管理器WorkManager ····· 356
13.2 HTTP 访问 ····· 358
13.2.1 移动数据格式JSON ····· 359
13.2.2 通过okhttp调用HTTP接口 ····· 361
13.2.3 使用okhttp下载和上传文件 ····· 365
13.3 图片加载 ····· 368
13.3.1 使用Glide加载网络图片 ····· 369
13.3.2 利用Glide实现图片的三级缓存 ····· 371
13.3.3 使用Glide加载特殊图像 ····· 373
13.4 即时通信 ····· 375
13.4.1 通过SocketIO传输文本消息 ····· 375
13.4.2 通过SocketIO传输图片消息 ····· 378
13.4.3 利用WebSocket传输消息 ····· 380
13.5 实战项目：仿微信的私聊和群聊 ····· 383
13.5.1 需求描述 ····· 383
13.5.2 功能分析 ····· 384
13.5.3 效果展示 ····· 390

13.6 小结……393
13.7 动手练习……393

第 14 章 多媒体……394

14.1 音频录播……394
14.1.1 使用系统录音机录制音频……394
14.1.2 利用MediaPlayer播放音频……395
14.1.3 利用MediaRecorder录制音频……398
14.2 传统摄制……399
14.2.1 使用系统相机拍摄照片……399
14.2.2 使用系统摄像机录制视频……401
14.2.3 利用视频视图与媒体控制条播放视频……403
14.2.4 截取视频的某帧画面……405
14.3 增强摄制……407
14.3.1 使用CameraX拍照……407
14.3.2 使用CameraX录像……411
14.3.3 新型播放器ExoPlayer……413
14.4 实战项目：仿抖音的短视频分享……416
14.4.1 需求描述……417
14.4.2 功能分析……418
14.4.3 效果展示……421
14.5 小结……422
14.6 动手练习……422

第 15 章 手机阅读……423

15.1 浏览网页……423
15.1.1 打开本地网页……423
15.1.2 网页浏览器……425
15.1.3 浏览EPUB文件……432
15.2 浏览 PDF 文件……435
15.2.1 PDF文件渲染器……435
15.2.2 实现平滑翻书效果……437
15.2.3 实现卷曲翻书动画……439
15.3 浏览 Office 文档……444
15.3.1 利用POI库浏览Word和Excel文档……445
15.3.2 结合前后端浏览PPT文档……447
15.3.3 利用TBS浏览各种文档……449
15.4 JNI 开发……451
15.4.1 NDK环境搭建……451
15.4.2 创建JNI接口……454
15.4.3 JNI实现加解密……457
15.4.4 采取CMake编译方式……459
15.5 实战项目：笔墨飘香之电子书架……462
15.5.1 需求描述……462
15.5.2 功能分析……463
15.5.3 效果展示……466
15.6 小结……467
15.7 动手练习……468

第 16 章 感知定位……469

16.1 传感器……469
16.1.1 传感器的种类……469
16.1.2 摇一摇——加速度传感器……471
16.1.3 指南针——磁场传感器……472
16.1.4 计步器、感光器和陀螺仪……474
16.2 基础定位……476
16.2.1 开启定位功能……476
16.2.2 获取定位信息……478
16.2.3 根据经纬度查找详细地址……482
16.2.4 全球卫星导航系统……483
16.3 地图导航……486
16.3.1 集成腾讯地图……486
16.3.2 显示地图面板……488
16.3.3 获取地点信息……490
16.3.4 规划导航路线……492
16.4 实战项目：仿微信的附近的人……496
16.4.1 需求描述……496
16.4.2 功能分析……497
16.4.3 效果展示……500
16.5 小结……501
16.6 动手练习……502

第 17 章 物联网……503

17.1 短距离通信……503
17.1.1 WiFi管理器……503
17.1.2 NFC近场通信……506
17.1.3 红外遥控……511
17.2 传统蓝牙……515
17.2.1 蓝牙技术的发展历程……515
17.2.2 蓝牙设备配对……515
17.2.3 点对点蓝牙通信……521

17.3 低功耗蓝牙 ····· 526
17.3.1 扫描BLE设备 ····· 526
17.3.2 发送BLE广播 ····· 531
17.3.3 通过主从BLE实现聊天应用 ····· 533
17.4 实战项目：自动驾驶的智能小车 ····· 538
17.4.1 需求描述 ····· 539
17.4.2 功能分析 ····· 539
17.4.3 效果展示 ····· 543
17.5 小结 ····· 544
17.6 动手练习 ····· 544

第 18 章 智能语音 ····· 545

18.1 原生语音处理 ····· 545
18.1.1 系统自带的语音引擎 ····· 545
18.1.2 文字转语音 ····· 547
18.1.3 中文转拼音 ····· 549
18.2 在线语音处理 ····· 551
18.2.1 原始音频录播 ····· 552
18.2.2 在线语音合成 ····· 554
18.2.3 在线语音识别 ····· 558
18.3 基于机器学习的语音推断 ····· 561
18.3.1 TensorFlow简介 ····· 561
18.3.2 TensorFlow Lite ····· 563
18.3.3 从语音中识别指令 ····· 564
18.4 实战项目：你问我答之小小机器人 ····· 566
18.4.1 需求描述 ····· 566
18.4.2 功能分析 ····· 567
18.4.3 效果展示 ····· 572
18.5 小结 ····· 574
18.6 动手练习 ····· 574

第 19 章 机器视觉 ····· 575

19.1 简单图像识别 ····· 575
19.1.1 扫描二维码 ····· 575
19.1.2 扫描识别文本 ····· 578
19.1.3 识别图像中的文档 ····· 582
19.2 人脸识别 ····· 583
19.2.1 人脸检测 ····· 584
19.2.2 人脸比对 ····· 585
19.2.3 活体检测 ····· 588
19.3 人像进阶 ····· 590
19.3.1 人像抠图 ····· 590
19.3.2 背景替换 ····· 592
19.3.3 笑脸捕捉 ····· 594
19.4 实战项目：寻人神器之智慧天眼 ····· 596
19.4.1 需求描述 ····· 597
19.4.2 功能分析 ····· 598
19.4.3 效果展示 ····· 599
19.5 小结 ····· 601
19.6 动手练习 ····· 601

第 20 章 在线直播 ····· 602

20.1 搭建 WebRTC 的服务端 ····· 602
20.1.1 WebRTC的系统架构 ····· 602
20.1.2 搭建信令服务器 ····· 604
20.1.3 搭建穿透服务器 ····· 606
20.2 给 App 集成 WebRTC ····· 608
20.2.1 引入WebRTC开源库 ····· 608
20.2.2 实现WebRTC的发起方 ····· 611
20.2.3 实现WebRTC的接收方 ····· 614
20.3 实战项目：仿微信的视频通话 ····· 616
20.3.1 需求描述 ····· 616
20.3.2 功能分析 ····· 617
20.3.3 效果展示 ····· 621
20.4 实战项目：仿拼多多的直播带货 ····· 621
20.4.1 需求描述 ····· 622
20.4.2 功能分析 ····· 622
20.4.3 效果展示 ····· 627
20.5 小结 ····· 629
20.6 动手练习 ····· 629

附录 A 移动互联网行业的新技术发展简表 ····· 630

附录 B Android 各版本的新增功能简表 ····· 632

附录 C Android 常用开发库说明简表 ····· 634

附录 D 移动开发专业术语索引 ····· 636

附录 E 本书的服务端程序说明 ····· 639

第 1 章

Android 开发环境搭建

本章主要介绍如何在个人计算机上搭建Android开发环境，包括：Android开发的发展历史是怎样的、Android Studio的开发环境是如何搭建的、如何创建并编译App工程、如何运行和调试App。

1.1 Android 开发简介

本节介绍Android开发的历史沿革，包括Android的发展历程和Android Studio的发展历程两个方面。

1.1.1 Android 的发展历程

Android是一款基于Linux的移动端开源操作系统，中文名为安卓，它不仅能用于智能手机，还可用于平板计算机、智能电视、车载大屏、智能家居等设备，已然成为人们日常生活中不可或缺的系统软件。Android的首个正式版本Android 1.0于2008年9月由谷歌公司发布，而第一部Android手机则由HTC公司制造。从此Android与苹果公司的iOS系统成为智能手机的两大操作系统，将功能机时代的霸主诺基亚拉下马来。因为Android的开源特性，各家手机厂商纷纷对其加以定制优化，所以Android阵营愈发壮大，带动Android手机的市场份额水涨船高。同时Android几乎每年都要发布一个大版本，技术的更新迭代速度非常快，表1-1展示了Android几个主要版本的发布时间。

表 1-1 Android 主要版本的发布时间

Android 版本号	对应的 API	发布时间
Android 12	31	2021 年 10 月
Android 11	30	2020 年 9 月
Android 10	29	2019 年 8 月
Android 9	28	2018 年 8 月
Android 8	26/27	2017 年 8 月
Android 7	24/25	2016 年 8 月
Android 6	23	2015 年 9 月
Android 5	21/22	2014 年 6 月
Android 4.4	19/20	2013 年 9 月

1.1.2 Android Studio 的发展历程

虽然Android是基于Linux内核，但是Android手机的应用App却主要是采用Java语言开发。为了吸引众多的Java程序员，早期的App开发工具是Eclipse，通过给Eclipse安装ADT插件，使之支持开发和调试App。然而Eclipse毕竟不是专门的App开发环境，运行速度也偏慢，因此谷歌公司在2013年5月推出了全新的Android开发环境——Android Studio。Android Studio基于IntelliJ IDEA演变而来，既保持了IDEA方便快捷的特点，又增加了Android开发的环境支持。自2015年之后，谷歌公司便停止了ADT的版本更新，转而重点打造自家的Android Studio，数年升级换代下来，Android Studio的功能愈加丰富，性能也愈高效，使得它逐步成为主流的App开发环境。表1-2展示了Android Studio几个主要版本的发布时间。

表 1-2 Android Studio 主要版本的发布时间

Android Studio 版本号	发布时间
Android Studio Dolphin	2022 年 9 月
Android Studio 4.0	2020 年 5 月
Android Studio 3.0	2017 年 10 月
Android Studio 2.0	2016 年 4 月
Android Studio 1.0	2013 年 5 月

1.2 搭建 Android Studio 开发环境

本节介绍在计算机上搭建Android Studio开发环境的过程和步骤：首先说明用作开发机的计算机应当具备哪些基本配置，然后描述Android Studio的安装和配置的详细过程，最后叙述如何下载Android开发需要的SDK组件及相关工具。

1.2.1 计算机配置要求

工欲善其事，必先利其器。要想保证Android Studio的运行速度，开发用的计算机配置就要跟上（现在一般使用笔记本计算机开发App）。下面是对计算机硬件的基本要求：

（1）内存要求至少8GB，越大越好。
（2）CPU要求1.5GHz以上，越快越好。
（3）硬盘要求系统盘剩余空间10GB以上，越大越好。
（4）要求带无线网卡与USB插槽。

下面是对操作系统的基本要求（以Windows为例）：

（1）必须是64位系统，不能是32位系统。
（2）Windows系统至少为Windows 7，推荐Windows 10，不支持Windows XP。
（3）屏幕分辨率至少为1280×800。

下面是对网络的基本要求：

（1）最好连接公众网，因为校园网可能无法访问国外的网站。

（2）下载速度至少每秒1MB，越快越好。因为Android Studio安装包大小为1GB左右，还需要另外下载几百MB的SDK，所以网络带宽一定要够大，否则下载文件都要等很久。

1.2.2　安装 Android Studio

Android Studio的官方下载页面是https://developer.android.google.cn/studio/index.html，单击网页中央的Download Android Studio按钮即可下载Android Studio的安装包。或者下拉网页找到Android Studio downloads区域，选择指定操作系统对应的Android Studio安装包。Android Studio的安装步骤如下：

01 双击下载完成的Android Studio安装程序，弹出安装向导对话框，如图1-1所示。直接单击Next按钮，进入下一页的组件选择对话框，如图1-2所示。



图 1-1　Android Studio 的安装向导对话框

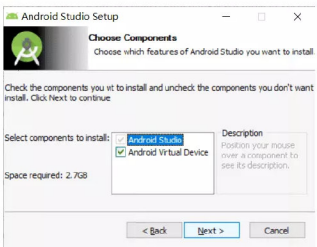


图 1-2　Android Studio 的组件选择对话框

02 勾选Android Studio和Android Virtual Device两个选项，然后单击Next按钮，进入下一页的安装路径对话框，如图1-3所示。建议将Android Studio安装在除系统盘外的其他磁盘（比如E盘），然后单击Next按钮，进入下一页的开始菜单设置对话框，如图1-4所示。

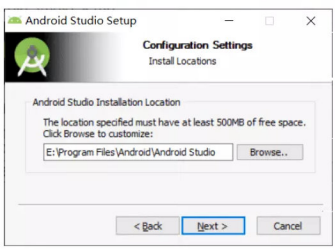


图 1-3　Android Studio 的安装路径对话框

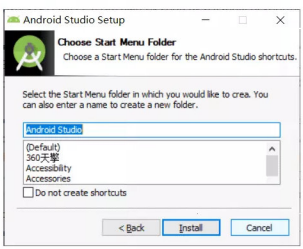


图 1-4　Android Studio 的开始菜单设置对话框

03 单击右下角的Install按钮，跳到下一页的安装过程对话框（见图1-5），耐心等待安装操作。待安装进度条填满后，单击Next按钮进入安装完成对话框，如图1-6所示。

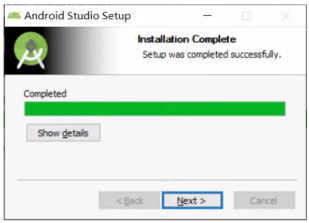


图 1-5 Android Studio 的安装过程对话框



图 1-6 Android Studio 的安装完成对话框

04 勾选安装完成对话框中的Start Android Studio选项，再单击右下角的Finish按钮，在结束安装操作的同时启动Android Studio。待Android Studio启动之后会打开如图1-7所示的配置向导对话框。单击Next按钮进入下一页的安装类型对话框，如图1-8所示。



图 1-7 Android Studio 的配置向导对话框

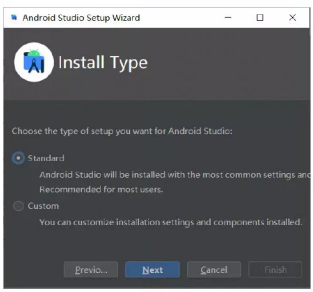


图 1-8 Android Studio 的安装类型对话框

05 这里保持Standard选项，单击Next按钮，跳到下一页的界面，如图1-9所示。选中右边的Light主题，表示开发界面采取白底黑字。然后单击Next按钮，跳到下一页的设置确认对话框，如图1-10所示。

06 设置确认对话框列出了需要下载哪些工具及其安装包大小，确认完毕后继续单击Next按钮，跳到下一页的许可授权对话框，如图1-11所示。该对话框默认选中intel-android-extra-license，单击右下角的Accept选项，表示接受以上条款；接着依次选中左上角的android-sdk-license和android-sdk-preview-license，分别在对应条款页面单击右下角的Accept选项，三种条款都接受之后的对话框如图1-12所示。

07 单击许可授权对话框右下角的Finish按钮，跳到组件下载对话框，如图1-13所示。耐心等待组件下载，全部下载完成后，调到更新完成对话框，该对话框提示更新成功，如图1-14所示。

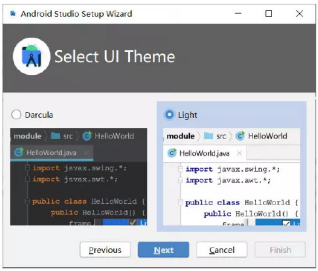


图 1-9　Android Studio 的对话框

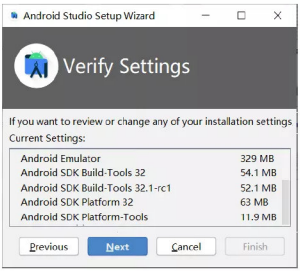


图 1-10　Android Studio 的设置确认对话框

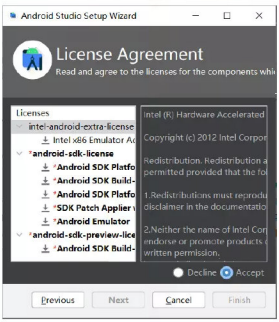


图 1-11　Android Studio 的许可授权对话框

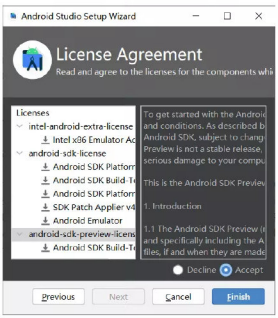


图 1-12　Android Studio 的许可授权对话框
（三种条款都接受）

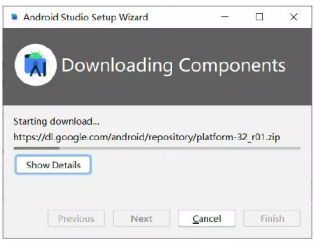


图 1-13　Android Studio 的组件下载对话框

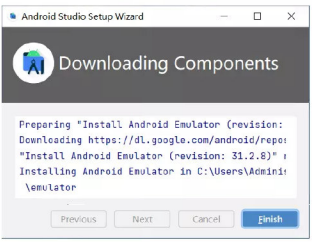


图 1-14　Android Studio 的更新完成对话框

08 单击对话框右下角的Finish按钮，完成安装配置工作，同时打开Android Studio的欢迎界面，如图1-15所示。单击中央的New Project 按钮即可开始Android开发之旅。

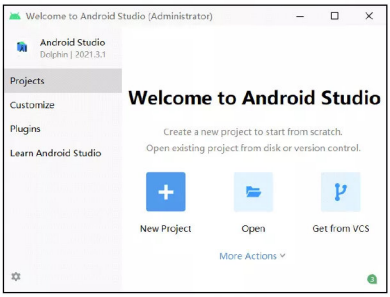


图 1-15 Android Studio 的欢迎界面

注 意

在配置过程中可能出现如下错误提示：

（1）第一次打开 Android Studio 可能会报 Unable to access Android SDK add-on list 错误信息，这个错误不用理会，单击 Cancel 按钮即可。进入 Android Studio 主界面后，依次选择 File→Project Structure→SDK Location，在弹出的对话框中设置 SDK 的路径。设置完毕后再打开 Android Studio 就不会报错了。

（2）已经按照安装步骤正确安装 Android Studio，但运行时却总是打不开。这时请检查计算机上是否开启了防火墙，建议关闭系统防火墙及所有杀毒软件的防火墙。关闭了防火墙后再重新打开 Android Studio 试试。

1.2.3 下载 Android 的 SDK

Android Studio只提供了App的开发环境界面，编译App源码还需另外下载Android官方的SDK，1.2.2节中的图1-10便展示了初始下载安装的SDK工具包。SDK全称为Software Development Kit，即软件开发工具包，它可将App源码编译为可执行的App应用。随着Android版本的更新换代，SDK也需时常在线升级，接下来介绍如何下载最新的SDK。具体步骤如下：

01 在 Android Studio 主界面，依次选择 Tools → SDK Manager，或者在Android Studio右上角中单击 图标，如图1-16所示。

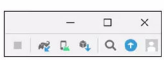


图 1-16 打开 SDK Manager 的图标栏

02 此时弹出SDK Manager的管理界面，界面右边是SDK安装配置区域，列出了各个SDK平台，初始画面如图1-17所示。注意Android SDK Location一栏，可单击右侧的Edit链接，进而选择SDK下载后的保存路径。其下的三个选项卡默认显示SDK Platforms，也就是各个SDK平台的版本列表，勾选每个列表项左边的复选框，表示需要下载该

版本的SDK平台，然后单击OK按钮即可自动下载并安装对应版本的SDK。也可单击中间的SDK Tools选项卡，此时会切换到SDK工具的管理列表，如图1-18所示。在这个工具管理界面，能够在线升级编译工具Build Tools、平台工具Platform Tools，以及开发者需要的其他工具。

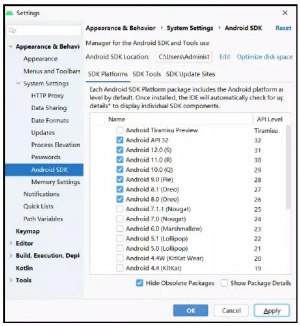


图 1-17　SDK 平台的管理列表

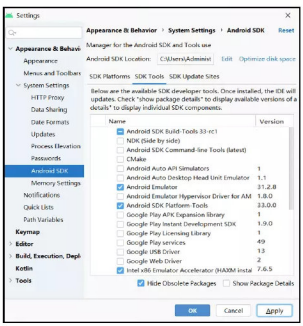


图 1-18　SDK 工具的管理列表

03 SDK下载完成后，可以到“我的计算机”中打开Android SDK Location指定的SDK保存路径，发现下面还有十几个目录，其中比较重要的几个目录说明如下：

- build-tools**目录：存放各版本**Android**的编译工具**。
- emulator**目录：存放模拟器的管理工具**。
- platforms**目录：存放各版本**Android**的资源文件与内核**JAR**包**android.jar。
- platform-tools**目录：存放常用的开发辅助工具，包括客户端驱动程序**adb.exe、**数据库管理工具**sqlite3.exe，**等等**。
- sources**目录：存放各版本**Android**的SDK源码**。

1.3　创建并编译 App 工程

本节介绍使用Android Studio创建并编译App工程的过程和步骤：首先叙述如何通过Android Studio创建新的App项目，然后描述如何导入已有的App工程（包括导入项目和导入模块两种方式），最后阐述如何手工编译App工程。

1.3.1　创建新项目

创建新项目的具体步骤如下：

01 在“1.2.2　安装Android Studio”节最后一步出来的图1-15中，单击第一项New Project会创建初始的新项目。如果要创建另外的新项目，也可在打开Android Studio之后，依次选择菜单File→New→New Project。以上两种创建方式都会弹出如图1-19所示的项目创建对话框，在该对话框

中保持默认的“Empty Activity”，单击Next按钮跳转到下一个配置对话框指定目标设备，如图1-20所示。

02 在配置对话框的Name栏输入应用名称，在Package Name栏输入应用的包名，在Save Location栏输入或者选择项目工程的保存目录，在Language下拉框中选择编码语言为Java，在Minimun SDK下拉框中选择最低支持到API20:Android 50(Lollipop)，Minimun SDK下方的文字提示当前版本支持设备的市场份额为98.0%。下面有个User legacy android.support libraries复选框，如果勾选表示采用旧的support支持库，如果不勾选表示采用新的androidx库，因为Android官方不再更新旧的support库，所以此处无须勾选，默认采用新的androidx库就可以了。

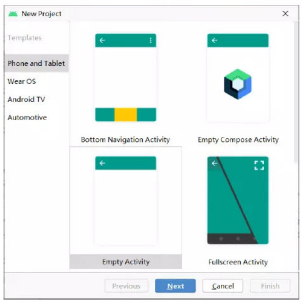


图 1-19 创建新项目

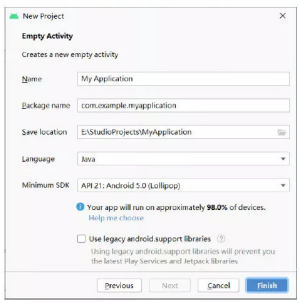


图 1-20 指定目标设备

03 单击Finish按钮完成配置操作，Android Studio便自动创建规定配置的新项目了。稍等片刻，Android Studio将呈现刚刚创建好的项目页面，如图1-21所示。

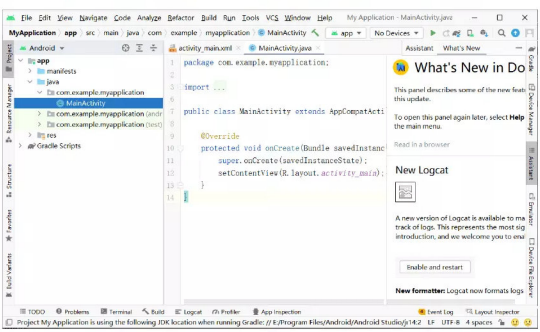


图 1-21 刚刚创建的项目页面

工程创建完毕后，Android Studio自动打开activity_main.xml与MainActivity.java，并默认展示MainActivity.java的源码。MainActivity.java上方的标签表示该文件的路径结构，注意源码左侧有一列标签，从上到下依次是Project、Resource Manager、Structure、Favorites、Build Variants。单击Project标签，左侧会展开小窗口表示该项目的工程结构，如图1-22所示。单击Structure标签，左侧会展开小窗口表示该代码的内部方法结构，如图1-23所示。

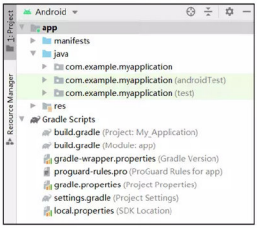


图 1-22　新项目的工程结构

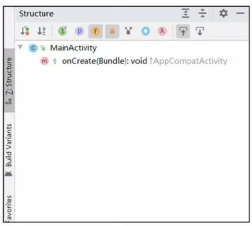


图 1-23　MainActivity 的方法结构

1.3.2　导入已有的工程

本书提供了所有章节的示例源码，为方便学习，读者可将本书源码直接导入Android Studio。根据App工程的组织形式，有两种源码导入方式，分别是导入整个项目，以及导入某个模块。这两种导入方式的简要说明如下。

1. 导入整个项目

以本书源码android3为例，依次选择菜单File→Open，或者依次选择菜单File→New→Import Project，均会弹出如图1-24所示的文件对话框。

在文件对话框中选中待导入的项目路径，再单击对话框下方的OK按钮。此时文件对话框关闭，弹出另一个如图1-25所示的确认对话框。

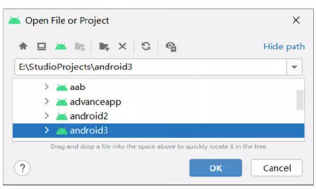


图 1-24　打开 App 项目的文件对话框

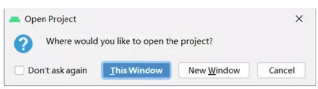


图 1-25　是否开启新窗口的确认对话框

确认对话框右下角有3个按钮，分别是This Window、New Window和Cancel，其中This Window按钮表示在当前窗口打开该项目，New Window按钮表示在新窗口打开该项目，Cancel按钮表示取消打开操作。此处建议单击New Window按钮，在新窗口打开App项目。

2. 导入某个模块

如果读者已经创建了自己的项目，想在当前项目中导入某章的源码，应当通过Module方式导入模块源码。依次选择菜单File→New→Import Module，弹出如图1-26所示的导入模块的对话框。

单击Source Directory输入框右侧的文件夹图标，弹出如图1-27所示的选择模块的文件对话框。

在文件对话框中选择待导入的模块路径，再单击对话框下方的OK按钮，回到如图1-28所示的导入对话框，导入对话框已经自动填上了待导入模块的完整路径。

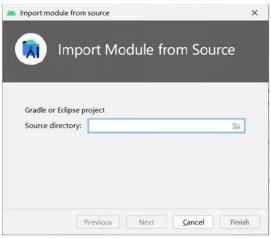


图 1-26　导入模块的对话框

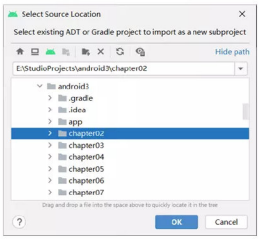


图 1-27　选择模块的文件对话框

单击对话框右下角的Finish按钮完成导入操作。然后Android Studio自动开始模块的导入和编译动作，等待导入结束即可在Android Studio左上角的项目结构图中看到导入的chapter02模块，如图1-29所示。

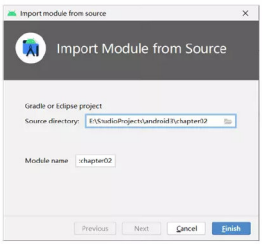


图 1-28　自动填写模块路径的导入对话框

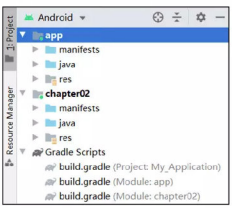


图 1-29　成功导入模块之后的项目结构图

1.3.3　编译 App 工程

Android Studio跟IDEA一样，被改动的文件会自动保存，无须开发者手工保存。它还会自动编译最新的代码，如果代码有误，编辑界面会标红提示出错了。但是有时候可能因为异常关闭的缘故，造成Android Studio的编译文件发生损坏，此时需要开发者手动重新编译。手动编译有以下3种途径：

（1）依次选择菜单Build→Make Project，该方式会编译整个项目下的所有模块。

（2）依次选择菜单Build→Make Module ***，该方式会编译指定名称的模块。

（3）先选择菜单Build→Clean Project，再选择菜单Build→Rebuild Project，表示先清理当前项目，再对整个项目进行重新编译。

不管是编译项目还是编译模块，编译结果都展示在Android Studio主界面下方的Build窗口中，如图1-30所示。

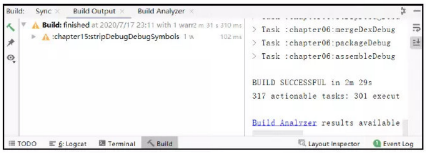


图 1-30　App 工程的编译结果窗口 Build

由编译结果可知，当前项目编译耗时2分29秒，共发现了1个警告，未发现错误。

1.4　运行和调试 App

本节介绍使用Android Studio运行和调试App的过程：首先叙述如何创建Android Studio内置的模拟器，然后描述如何在刚创建的模拟器上运行测试App，最后阐述如何在Android Studio中查看App的运行日志。

1.4.1　创建内置模拟器

所谓模拟器，指的是在计算机上构造一个演示窗口，模拟手机屏幕的App运行效果。App通过编译之后，只说明代码没有语法错误，若想验证App能否正确运行，还得让它在Android设备上跑起来。这个设备可以是真实手机，也可以是计算机里的模拟器。依次选择菜单Run→Run '***'（也可按快捷键Shift+F10），或者选择菜单Run→Run...，在弹出的小窗口中选择待运行的模块名称，Android Studio会判断当前是否存在已经连接的设备，如果已有连接上的设备就在该设备上安装测试App。

如果一开始没有任何已连上的设备，运行App就会报错“Error running '***': No target device found.”，意思是未找到任何目标设备。此时要先创建一个模拟器，具体操作步骤如下：

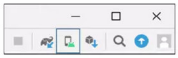


图 1-31　打开 Device Manager 的图标栏

01 依次选择菜单Tools→Device Manager，或者在Android Studio右上角的按钮中单击图标，如图1-31所示。

02 此时Android Studio打开模拟器的创建窗口，如图1-32所示。单击创建窗口中的Create device按钮，弹出如图1-33所示的硬件选择对话框。

03 在对话框的左边列表中单击Phone表示选择手机，在中间列表中选择某个手机型号如Pixel 2，然后单击对话框右下角的Next按钮，跳转到下一页的系统镜像选择对话框，如图1-34所示。从图中看到镜像列表顶端的发布名称叫作S，对应的API级别为31，它正是Android 12的系统镜像。

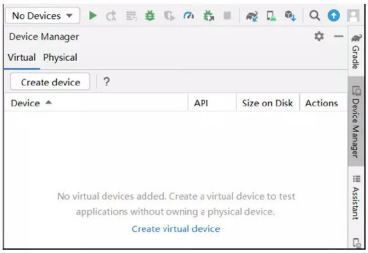


图 1-32　模拟器的创建窗口

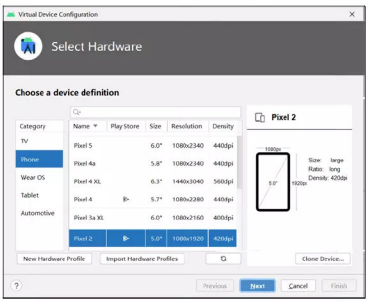


图 1-33　硬件选择对话框

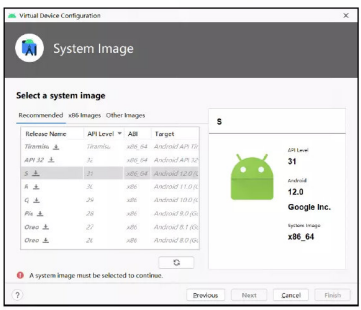


图 1-34　系统镜像选择对话框

04 单击S右边的下载图标，弹出如图1-35所示的许可授权对话框。

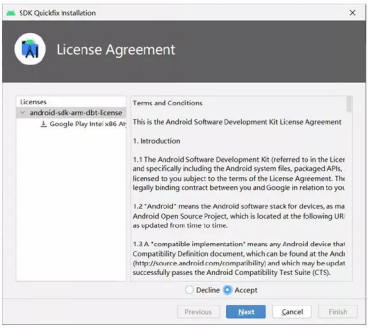


图 1-35　许可授权对话框

05 单击许可授权对话框的Accept选项，表示接受上述条款，再单击Next按钮跳转到下一页的镜像下载对话框，如图1-36所示。

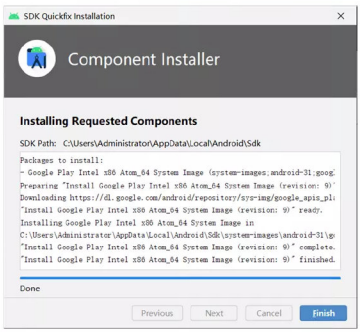


图 1-36　镜像下载对话框

06 等待镜像下载完成，单击右下角的Finish按钮，返回到如图1-37所示的系统镜像选择对话框。

07 此时S右边的下载图标消失，说明计算机中已经存在该版本的Android镜像。于是选中S这行，再单击Next按钮，跳转到模拟器的配置对话框，如图1-38所示。

08 配置对话框左上方的AVD Name用于填写模拟器的名称，这里保持默认名称不动，单击对话框右下角的Finish按钮完成创建操作。一会儿对话框关闭，回到如图1-39所示的模拟器的列表对话框，该对话框中多了个名为Pixel 2 API 31的模拟器，且该模拟器基于Android 12（API 31）。

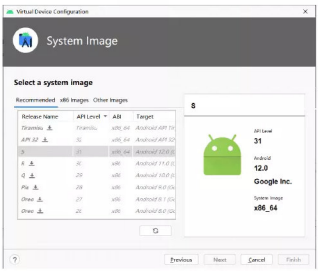


图 1-37　系统镜像选择对话框

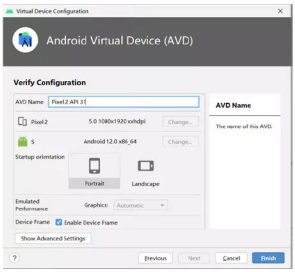


图 1-38　模拟器的配置对话框

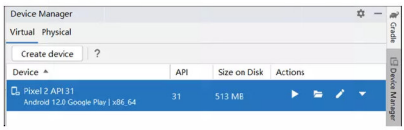


图 1-39　模拟器的列表对话框

1.4.2　在模拟器上运行 App

模拟器创建完成后，回到Android Studio的主界面，即可在顶部工具栏的下拉框中发现多了个“Pixel 2 API 31”，它正是上一小节创建好的模拟器，如图1-40所示。

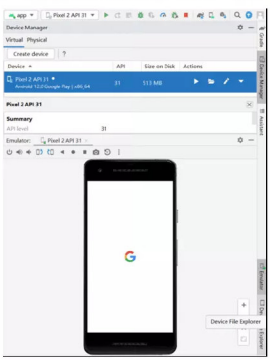


图 1-40　在顶部工具栏中出现刚创建的模拟器

重新选择菜单Run→Run 'app'，也可以直接单击Pixel 2 API 31右侧的三角运行按钮，Android Studio便开始启动名为“Pixel 2 API 31”的模拟器，如图1-41所示。等待模拟器启动完毕，出现如图1-42所示的模拟器的开机画面。再过一会儿，模拟器自动打开如图1-43所示的App界面。

在图1-43中可以看见在模拟器屏幕左上角显示应用名称为“My Application”，在屏幕中间显示页面内容为“Hello World！”。它正是刚才想要运行的测试App，说明已经在模拟器上成功运行App了。



图 1-41　模拟器正在启动



图 1-42　模拟器的开机画面



图 1-43　模拟器运行 App

1.4.3　观察 App 的运行日志

虽然在模拟器上能够看到App的运行，却无法看到App的调试信息。以前写Java代码的时候，通过System.out.println可以很方便地向IDEA的控制台输出日志，当然Android Studio也允许查看App的运行日志，只是Android不使用System.out.println，而是采用Log工具打印日志。

有别于System.out.println，Log工具将各类日志划分为5个等级，每个等级的重要性是不一样的，这些日志等级按照从高到低的顺序依次说明如下：

- Log.e：**表示错误信息，比如可能导致程序崩溃的异常。**
- Log.w：**表示警告信息。**
- Log.i：**表示一般消息。**
- Log.d：**表示调试信息，可把程序运行时的变量值打印出来，方便跟踪调试。**
- Log.v：**表示冗余信息。**

一般而言，日常开发使用Log.d即可。下面是给App添加日志信息的代码的例子：

```
(完整代码见app\src\main\java\com\example\app\MainActivity.java)
import android.util.Log;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d("MainActivity", "我看到你了");  // 添加一行日志信息
    }
}
```

重新运行测试App，等模拟器刷新App界面后，单击Android Studio底部的Logcat标签，此时主界面下方弹出一排日志窗口，如图1-44所示。

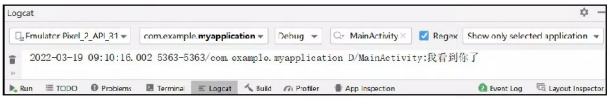


图 1-44 Android Studio 的日志查看窗口

日志窗口的顶部是一排条件筛选框，从左到右依次为：测试设备的名称（如“Pixel_2_API_31”）、测试App的包名（例如只显示com.example.myapplication的日志）、查看日志的级别（例如只显示级别不低于Debug即Log.d的日志）、日志包含的字符串（例如只显示包含MainActivity的日志），还有最后一个是筛选控制选项（其中“Show only selected application”表示只显示选中的应用日志，而“No Filters”则表示不过滤任何条件）。经过一排条件筛选之后，Logcat窗口只显示一行文字“D/MainActivity: 我看到你了”，说明成功捕获前面代码调用Log.d的日志信息。

1.5 小 结

本章主要介绍了Android开发环境的搭建过程，包括：Android开发简介（Android的发展历程、Android Studio的发展历程）、搭建Android Studio开发环境（开发机配置要求、安装Android Studio、下载Android的SDK）、创建并编译App工程（创建新项目、导入已有的工程、编译App工程）、运行和调试App（创建内置模拟器、在模拟器上运行App、观察App的运行日志）。

通过本章的学习，读者应该掌握Android Studio的基本操作技能，能够使用自己搭建的Android Studio环境创建简单的App工程，并在模拟器上成功运行测试App。

1.6 动 手 练 习

请上机实验搭建App的开发环境，主要步骤说明如下：

1. 下载并安装Android Studio的最新版本。
2. 创建一个新的App项目“Hello World”。
3. 使用Android Studio创建一个模拟器。
4. 在模拟器上安装并运行第二步创建的App，观察能否看到“Hello World”字样。

第 2 章

Android App 开发基础

本章介绍基于Android系统的App开发常识，包括以下几个方面：App开发与其他软件开发有什么不一样，App工程是怎样的组织结构又是怎样配置的，App开发的前后端分离设计是如何运作实现的，App的活动页面是如何创建又是如何跳转的。

2.1 App 的开发特点

本节介绍App开发与其他软件开发不一样的特点，例如：App能在哪些操作系统上运行、App开发用到了哪些编程语言、App能操作哪些数据库等，搞清楚了App的开发运行环境，才能有的放矢，不走弯路。

2.1.1 App 的运行环境

App 是在手机上运行的一类应用软件，而应用软件依附于操作系统，无论计算机还是手机，在开机时都会显示桌面，这个桌面便是操作系统的工作台。个人计算机的操作系统主要有微软的Windows和苹果的MacOS，智能手机流行的操作系统也有两种，分别是安卓手机的Android和苹果手机的iOS。本书讲述的App开发为Android上的应用开发，Android系统基于Linux内核，但不等于Linux系统，故App应用无法在Linux系统上运行。

Android Studio是谷歌官方推出的App开发环境，它提供了三种操作系统的安装包，分别是Windows、Mac和Linux。这就产生了一个问题：开发者可以在计算机上安装Android Studio，并使用Android Studio开发App项目，但是编译出来的App在计算机上跑不起来。这种情况真是令人匪夷所思，通常学习C语言、Java或者Python，都能在计算机的开发环境中直接观看程序运行过程，就算是J2EE开发，也能在浏览器中通过网页观察程序的运行结果。可是安卓的App应用竟然没法在计算机上直接运行，那该怎样验证App的界面展示及其业务逻辑是否正确呢？

为了提供App开发的功能测试环境，一种办法是利用Android Studio创建内置的模拟器，然后启动内置模拟器，再在模拟器上运行App应用，详细步骤参见第1章的“1.4.2　在模拟器上运行App”。

另一种办法是使用真实手机测试App，该办法在实际开发中更为常见。由于模拟器本身运行在计算机上面，占用计算机的CPU和内存，会拖累计算机的运行速度；况且模拟器仅仅是模拟而已，无法完全验证App的所有功能，因此最终都得通过真机测试才行。

利用真机调试要求具备以下5个条件：

1. 使用数据线把手机连接到计算机上

手机的电源线拔掉插头就是数据线。数据线长方形的一端接到计算机的USB接口，即可完成手机与计算机的连接。

2. 在计算机上安装手机的驱动程序

一般计算机会把手机当作USB存储设备一样安装驱动，大多数情况会自动安装成功。如果遇到少数情况安装失败，需要先安装**手机助手，由助手软件下载并安装对应的手机驱动。

3. 打开手机的开发者选项并启用USB调试

手机出厂后默认关闭开发者选项，需要开启开发者选项才能调试App。打开手机的设置菜单，进入“系统”→“关于手机”→“版本信息”页面，这里有好几个版本项，每个版本项都使劲点击七八下，总会有某个版本在点击后出现“你将开启开发者模式”的提示。继续点击该版本开启开发者模式，然后退出并重新进入设置页面，此时就能在“系统”菜单下找到“开发者选项”或“开发人员选项”了。进入“开发者选项”页面，启用“开发者选项”和“USB调试”两处开关，允许手机通过USB接口安装调试应用。

4. 将连接的手机设为文件传输模式，并允许计算机进行USB调试

手机通过USB数据线连接计算机后，在屏幕上弹出如图2-1所示的USB连接方式选择列表，请求选择某种USB连接方式。这里记得选中“传输文件”，因为充电模式不支持调试App。

选完之后手机桌面弹出如图2-2所示的确认对话框，提示开发者是否允许当前计算机进行USB调试。这里勾选“始终允许使用这台计算机进行调试”选项，再点击右下角的“确定”按钮，允许计算机在手机上调试App。



图 2-1　USB 连接方式选择列表



图 2-2　USB 调试的确认对话框

5. 手机要能正常使用

在锁屏状态下，Android Studio向手机安装App的行为可能会被拦截，所以要保证手机处于解锁状态，才能顺利通过计算机安装App到手机上。

有的手机还要求插入SIM卡才能调试App，还有的手机要求登录会员才能调试App，总之如果遇到无法安装的问题，各种情况都尝试一遍才好。

经过以上操作，总算具备通过计算机在手机上安装App的条件了。马上启动Android Studio，在顶部中央的执行区域找到已连接的手机信息，如图2-3所示。此时的设备信息提示这是一台华为手机，单击手机名称右边的三角运行按钮，接下来就是等待Android Studio往手机上安装App了。

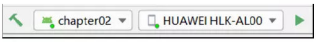


图 2-3　找到已连接的真机设备

2.1.2　App 的开发语言

基于安卓系统的App开发主要有两大技术路线，分别是原生开发和混合开发。原生开发指的是在移动平台上利用官方提供的编程语言（例如Java、Kotlin等）、开发工具包（SDK）、开发环境（Android Studio）进行App开发；混合开发指的是结合原生与H5技术开发混合应用，也就是将部分App页面改成内嵌的网页，这样无须升级App、只要覆盖服务器上的网页，即可动态更新App页面。

不管是原生开发还是混合开发，都要求掌握Android Studio的开发技能，因为混合开发本质上依赖于原生开发，如果没有原生开发的皮，哪里还有混合开发的毛呢？单就原生开发而言，又涉及多种编程语言，包括Java、Kotlin、C/C++、XML等，详细说明如下。

1. Java

Java是Android开发的主要编程语言，在创建新项目时，弹出如图2-4所示的项目配置对话框，Language栏默认选择了Java，表示该项目采用Java编码。

虽然Android开发需要Java环境，但没要求计算机上必须事先安装JDK，因为Android Studio已经自带了JRE。依次选择菜单File→Project Structure，弹出如图2-5所示的项目结构对话框。

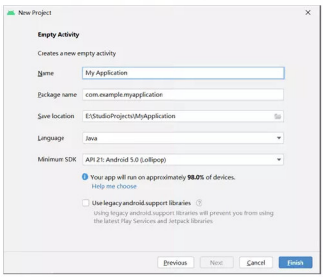


图 2-4　创建新项目时的项目配置对话框（Java）

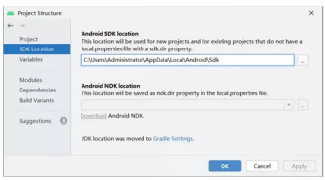


图 2-5　项目结构对话框

单击项目结构对话框左侧的SDK Location，对话框右边从上到下依次排列着Android SDK location、Android NDK location、JDK location was moved to Gradle Settings。单击JDK location was moved to Gradle Settings中的Gradle Settings，打开Gradle的设置窗口，如图2-6所示。

其中下方的Gradle JDK提示JDK路径位于Android Studio安装路径的JRE目录下，它正是Android Studio自带的Java运行环境。

可是Android Studio自带的JRE看不出来基于Java哪个版本，它支不支持最新的Java版本呢？其实Android Studio自带的JRE默认采用Java 7编译，如果在代码里直接书写Java 8语句就会报错，比如Java 8引入了Lambda表达式，下面代码通过Lambda表达式给整型数组排序：

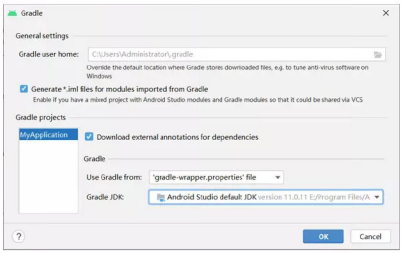


图 2-6　Gradle 的设置窗口

```
Integer[] intArray = { 89, 3, 67, 12, 45 };
Arrays.sort(intArray, (o1, o2) -> Integer.compare(o2, o1));
```

倘若由Android Studio编译上面代码，结果提示出错“Lambda expressions are not supported at language level '7'”，意思是Java 7不支持Lambda表达式，错误信息如图2-7所示。

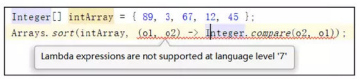


图 2-7　不支持 Lambda 表达式的出错提示

看来Android Studio果真默认支持Java 7而非Java 8，但Java 8增添了诸多新特性，其拥趸与日俱增，有的用户已经习惯了Java 8，能否想办法让Android Studio也支持Java 8呢？当然可以，只要略施小计便可，依次选择菜单File→Project Structure，在弹出的项目结构对话框左侧单击Modules，此时模块的属性设置对话框如图2-8所示。

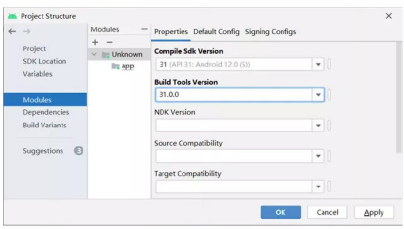


图 2-8　模块的属性设置对话框

对话框右侧的Properties选项卡，从上到下依次排列着Compile Sdk Version、Build Tools Version、NDK Version、Source Compatibility、Target Compatibility，这5项分别代表：编译的SDK版本、构建工具的版本、编译C/C++代码的NDK版本、源码兼容性、目标兼容性，其中后面两项用来设置Java代码的兼容版本。单击Source Compatibility右边的下拉箭头按钮，弹出如图2-9所示的下拉列表。

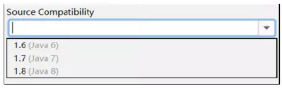


图 2-9　源码兼容性的 Java 版本选择列表

从下拉列表中看到，Android Studio自带的JRE支持Java 6、Java 7、Java 8三种版本。单击选中列表项的“1.8（Java 8）”，并在Target Compatibility栏也选择“1.8（Java 8）”，然后单击窗口下方的OK按钮，就能将编译模块的Java版本改成Java 8了。

2. Kotlin

Kotlin是谷歌官方力推的又一种编程语言，它与Java同样基于JVM（Java Virtual Machine，即Java虚拟机），且完全兼容Java语言。创建新项目时，在Language栏下拉可选择Kotlin，此时项目结构对话框如图2-10所示。

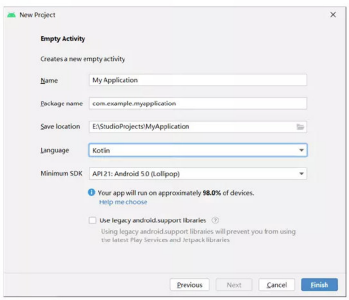


图 2-10　创建新项目时的项目配置对话框（Kotlin）

一旦在创建新项目时选定Kotlin，该项目就会自动加载Kotlin插件，并将Kotlin作为默认的编程语言。不过本书讲述的App开发采用Java编程，未涉及Kotlin编程，如果读者对Kotlin开发感兴趣，可参考笔者的另一部图书《Kotlin从零到精通Android开发》。

3. C/C++

不管是Java还是Kotlin，它们都属于解释型语言，这类语言在运行之时才将程序翻译成机器语言，故而执行效率偏低。虽然现在手机配置越来越高，大多数场景的App运行都很流畅，但是涉及图像与音视频处理等复杂运算的场合，解释型语言的性能瓶颈便暴露了出来。

编译型语言在首次编译时就将代码编译为机器语言，后续运行无须重新编译，直接使用之前的编译文件即可，因此执行效率比解释型语言高。C/C++正是编译型语言的代表，它能够有效弥补解释型语言的性能缺憾，借助于JNI（Java Native Interface，即Java原生接口）技术，Java代码允许调用C/C++编写的程序。事实上，Android的SDK开发包内部定义了许多JNI，包括图像读写在内的底层代码均由C/C++编写，再由外部通过封装好的Java方法调用。

4. XML

XML全称为Extensible Markup Language，即可扩展标记语言，严格地说，XML并非编程语言，只是一种标记语言。它类似于HTML，利用各种标签表达页面元素，以及各元素之间的层级关系及其排列组合。每个XML标签都是独立的控件对象，标签内部的属性以“android:”打头，表示这是标准的安卓属性，各属性分别代表控件的某种规格。比如下面是以XML书写的文本控件：

```
<TextView
    android:id="@+id/tv_hello"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!" />
```

上面的标签名称为TextView，翻译过来叫文本视图，该标签携带4个属性，说明如下：

- id：**控件的编号**。
- layout_width：**控件的布局宽度**，wrap_content**表示刚好包住该控件的内容**。
- layout_height：**控件的布局高度**，wrap_content**表示刚好包住该控件的内容**。
- text：**控件的文本，也就是文本视图要显示的文字**。

综合起来，以上XML代码所表达的意思为：这是一个名为tv_hello的文本视图，显示的文字内容是“Hello World!”，它的宽度和高度都要刚好包住这些文字。

以上就是Android开发常见的几种编程语言，本书选择了Java路线而非Kotlin路线，并且定位是安卓初学者教程，因此读者需要具备Java和XML基础。如果你尚未掌握Java编程，建议先学习笔者的Java专著《好好学Java：从零基础到项目实战》，等打好Java基础再学Android开发也不迟。

2.1.3 App 连接的数据库

在学习Java编程的时候，基本会学到数据库操作，通过JDBC连接数据库进行记录的增删改查，这个数据库可能是MySQL，也可能是Oracle，还可能是SQL Server。然而手机应用不能直接操作上述几种数据库，因为数据库软件也得像应用软件那样安装到操作系统上，比如MySQL提供了Windows系统的安装包，也提供了Linux系统的安装包，可是它没有提供Android系统的安装包，所以MySQL无法在Android系统上安装，手机里面的App也就不能直接连接MySQL。

既然MySQL、Oracle这些企业数据库无法在手机上安装，那么App怎样管理业务方面的数据记录呢？其实Android早已内置了专门的数据库，名为SQLite，它遵循关系数据库的设计理念，SQL语法类似于MySQL。不同之处在于，SQLite无须单独安装，因为它内嵌到应用进程当中，所以App无须配置连接信息，即可直接对其增删改查。由于SQLite嵌入到应用程序中，省去了配置数据库服务器的开销，因此它又被归类为嵌入式数据库。

可是SQLite的数据库文件保存在手机上，开发者拿不到用户的手机，又该如何获取App存储的业务数据呢？比如用户的注册信息、用户的购物记录，等等。如果像Java Web那样，业务数据统一保存在后端的数据库服务器中，开发者只要登录数据库服务器，就能方便地查询并导出需要的记录信息。

手机端的App，连同程序代码及其内置的嵌入式数据库，其实是个又独立又完整的程序实体，它只负责手机上的用户交互与信息处理，该实体被称作客户端。而后端的Java Web服务，包括Web代码和数据库服务器，同样构成另一个单独运行的程序实体，它只负责后台的业务逻辑与数据库操

作，该实体被称作服务端。客户端与服务端之前通过HTTP接口通信，每当客户端觉得需要把信息发给服务端，或者需要从服务端获取信息时，客户端便向服务端发起HTTP请求，服务端收到客户端的请求之后，根据规则完成数据处理，并将处理结果返回给客户端。这样客户端经由HTTP接口并借服务端之手，方能间接读写后端的数据库服务器（如MySQL），具体的信息交互过程如图2-11所示。

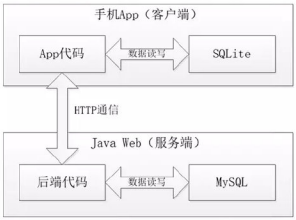


图 2-11　客户端与服务端分别操作的数据库

由此看来，一个具备用户管理功能的App系统，实际上并不单单只是手机上的一个应用，还包括与其对应的Java Web服务。手机里的客户端App，面向的是手机用户，App与用户之间通过手机屏幕交互；而后端的服务程序，面向的是手机App，客户端与服务端之间通过HTTP接口交互。客户端和服务端这种多对一的架构关系如图2-12所示。

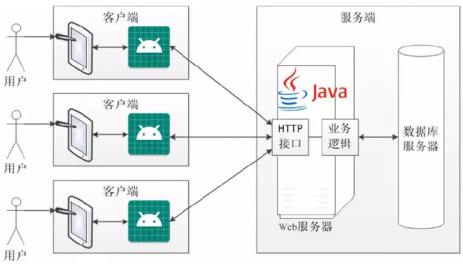


图 2-12　客户端与服务端的多对一架构关系图

总结一下，手机App能够直接操作内置的SQLite数据库，但不能直接操作MySQL这种企业数据库。必须事先搭建好服务端程序（如Java Web），然后客户端与服务端通过HTTP接口通信，再由服务端去操作MySQL这样的数据库服务器。

2.2　App 的工程结构

本节介绍App工程的基本结构及其常用配置：首先描述项目和模块的区别，以及工程内部各目录与配置文件的用途说明；其次阐述两种级别的编译配置文件build.gradle，以及它们内部的配置信息说明；最后讲述运行配置文件AndroidManifest.xml的节点信息及其属性说明。

2.2.1　App 工程目录结构

App工程分为两个层次：第一个层次是项目，依次选择菜单File→New→New Project即可创建

新项目；另一个层次是模块，模块依附于项目，每个项目至少有一个模块，也能拥有多个模块，依次选择菜单File→New→New Module即可在当前项目创建新模块。一般所言的“编译运行App”，指的是运行某个模块，而非运行某个项目，因为模块才对应实际的App。单击Android Studio左上角竖排的Project标签，可见App工程的项目结构如图2-13所示。

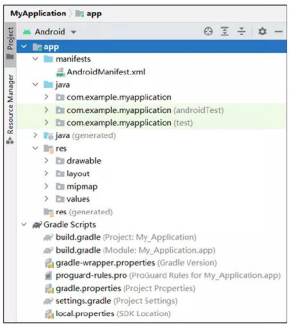


图 2-13　App 工程的项目结构图

从图2-13中可以看到，该项目下面有两个分类：一个是app（代表app模块），另一个是Gradle Scripts。其中，app下面又有3个子目录，其功能说明如下：

（1）manifests子目录，下面只有一个XML文件，即AndroidManifest.xml，它是App的运行配置文件。

（2）java子目录，下面有3个com.example.myapplication包，其中第一个包存放当前模块的Java源码，后面两个包存放测试用的Java代码。

（3）res子目录，存放当前模块的资源文件。res下面又有4个子目录：

- drawable**目录：存放图形描述文件与图片文件。**
- layout**目录：存放**App**页面的布局文件。**
- mipmap**目录：存放**App**的启动图标。**
- values**目录：存放一些常量定义文件，例如字符串常量**strings.xml**、像素常量**dimens.xml**、颜色常量**colors.xml**、样式风格定义**styles.xml**等。**

Gradle Scripts下面主要是工程的编译配置文件，主要有：

（1）build.gradle，该文件分为项目级与模块级两种，用于描述App工程的编译规则。

（2）proguard-rules.pro，该文件用于描述Java代码的混淆规则。

（3）gradle.properties，该文件用于配置编译工程的命令行参数，一般无须改动。

（4）settings.gradle，该文件配置了需要编译哪些模块。初始内容为include ':app'，表示只编译app模块。

（5）local.properties，项目的本地配置文件，它在工程编译时自动生成，用于描述开发者计算机的环境配置，包括SDK的本地路径、NDK的本地路径等。

2.2.2　编译配置文件 build.gradle

新创建的App项目默认有两个build.gradle，一个是Project（项目）级别的build.gradle；另一个是Module（模块）级别的build.gradle。

在Android Studio 4.2和之前的版本中，项目级别的build.gradle指定了当前项目的总体编译规则，打开该文件在buildscript下面找到repositories和dependencies两个节点，其中repositories节点用于设置Android Studio插件的网络仓库地址，而dependencies节点用于设置gradle插件的版本号。由于官方的谷歌仓库位于国外，下载速度相对较慢，因此可在repositories节点添加阿里云的仓库地址，方便国内开发者下载相关插件。修改之后的buildscript节点内容如下：

```
buildscript {
    repositories {
        // 以下4行添加阿里云的仓库地址，方便国内开发者下载相关插件
        maven { url 'https://maven.aliyun.com/repository/jcenter' }
        maven { url 'https://maven.aliyun.com/repository/google'}
        maven { url 'https://maven.aliyun.com/repository/gradle-plugin'}
        maven { url 'https://maven.aliyun.com/repository/public'}
        google()
        mavenCentral()
    }
    dependencies {
        // 配置gradle插件版本，下面的版本号就是Android Studio的版本号
        classpath 'com.android.tools.build:gradle:4.1.0'
    }
}
```

但从Android Studio Bumblebee开始，项目级别的build.gradle不再存放仓库地址，而是将仓库地址的repositories节点配置到了settings.gradle。因此要在settings.gradle中补充如下的阿里云仓库地址：

```
pluginManagement {
    repositories {
        gradlePluginPortal()
        google()
        mavenCentral()
        // 以下4行添加阿里云的仓库地址，方便国内开发者下载相关插件
        maven { url 'https://maven.aliyun.com/repository/jcenter' }
        maven { url 'https://maven.aliyun.com/repository/google'}
        maven { url 'https://maven.aliyun.com/repository/gradle-plugin'}
        maven { url 'https://maven.aliyun.com/repository/public'}
    }
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
        // 以下4行添加阿里云的仓库地址，方便国内开发者下载相关插件
        maven { url 'https://maven.aliyun.com/repository/jcenter' }
        maven { url 'https://maven.aliyun.com/repository/google'}
        maven { url 'https://maven.aliyun.com/repository/gradle-plugin'}
        maven { url 'https://maven.aliyun.com/repository/public'}
    }
}
```

至于模块级别的build.gradle则分别对应具体模块，每个模块都有自己的build.gradle，它指定了当前模块的详细编译规则。下面给chapter02模块的build.gradle补充文字注释，方便读者更好地理解每个参数的用途。

```
(完整代码见chapter02\build.gradle)
plugins {
    id 'com.android.application'
}

android {
    // 指定编译用的SDK版本号。比如31表示使用Android 12编译
    compileSdk 31

    defaultConfig {
        // 指定该模块的应用编号，也就是App的包名
        applicationId "com.example.chapter02"
        // 指定App适合运行的最小SDK版本号。比如21表示至少要在Android 5.0上运行
```

```
            minSdk 21
            // 指定目标设备的SDK版本号。表示App最希望在哪个版本的Android上运行
            targetSdk 31
            // 指定App的应用版本号
            versionCode 1
            // 指定App的应用版本名称
            versionName "1.0"
            testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
        }

        buildTypes {
            release {
                minifyEnabled false
                proguardFiles getDefaultProguardFile('proguard-android-optimize. txt'),
'proguard-rules.pro'
            }
        }
    }

    // 指定App编译的依赖信息
    dependencies {
        // 指定引用JAR包的路径
        implementation fileTree(dir: 'libs', include: ['*.jar'])
        // 指定编译Android的高版本支持库。如AppCompatActivity必须指定编译appcompat库
        implementation 'androidx.appcompat:appcompat:1.4.1'
        // 指定单元测试编译用的junit版本号
        testImplementation 'junit:junit:4.13.2'
        androidTestImplementation 'androidx.test.ext:junit:1.1.3'
        androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    }
```

为什么这两种编译配置文件的扩展名都是gradle呢？这是因为它们采用了Gradle工具完成编译构建操作。Gradle工具的版本配置在gradle\wrapper\gradle-wrapper.properties，也可以依次选择菜单File→Project Structure→Project，在弹出的设置页面中修改Gradle Version。注意每个版本的Android Studio都有对应的Gradle版本，只有二者的版本正确对应，App工程才能成功编译。比如Android Studio Dolphin对应的Gradle版本为7.4，更多的版本对应关系见https://developer.android.google.cn/studio/releases/gradle-plugin#updating-plugin。

2.2.3 运行配置文件 AndroidManifest.xml

AndroidManifest.xml指定了App的运行配置信息，它是一个XML描述文件，初始内容如下：

```
(完整代码见chapter02\src\main\AndroidManifest.xml)
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.chapter02">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic launcher round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
```

```
            </intent-filter>
        </activity>
    </application>
</manifest>
```

可见AndroidManifest.xml的根节点为manifest，它的package属性指定了该App的包名。manifest下面有个application节点，它的各属性说明如下：

- android:allowBackup：是否允许应用备份。为true表示允许，为false则表示不允许。
- android:icon：指定App在手机屏幕上显示的图标。
- android:label：指定App在手机屏幕上显示的名称。
- android:roundIcon：指定App的圆角图标。
- android:supportsRtl：是否支持阿拉伯语/波斯语这种从右往左的文字排列顺序。为true表示支持，为false则表示不支持。
- android:theme：指定App的显示风格。

注意application下面还有个activity节点，它是活动页面的注册声明，只有在AndroidManifest. xml中正确配置了activity节点，才能在运行时访问对应的活动页面。初始配置的MainActivity正是App的默认主页，之所以说该页面是App主页，是因为它的activity节点内部还配置了以下的过滤信息：

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

其中action节点设置的android.intent.action.MAIN表示该页面是App的入口页面，启动App时会最先打开该页面。而category节点设置的android.intent.category.LAUNCHER决定了是否在手机屏幕上显示App图标，如果同时有两个activity节点内部都设置了android.intent.category.LAUNCHER，那么桌面就会显示两个App图标。以上的两种节点规则可能一开始不太好理解，读者只需记住默认主页必须同时配置这两种过滤规则即可。

注　意

从 Android12 开始，任意组件节点（包括 activity、receiver、service）一旦配置了 intent-filter，那么该组件节点就必须声明 android:exported 属性，以便设定是否支持其他应用调用当前组件。

2.3　App 的设计规范

本节介绍App工程的源码设计规范：首先App将看得见的界面设计与看不见的代码逻辑区分开，然后利用XML标记描绘应用界面，同时使用Java代码书写程序逻辑，从而形成App前后端分离的设计规约，有利于提高App集成的灵活性。

2.3.1　界面设计与代码逻辑

手机的功能越来越强大，在某种意义上相当于微型计算机。比如在手机上打开一个电商App，仿佛是在计算机上浏览网站。网站分为用户看得到的网页，以及用户看不到的Web后台；App也分为用户看得到的界面，以及用户看不到的App后台。虽然Android允许使用Java代码描绘界面，但不提倡这么做，推荐的做法是将界面设计从Java代码剥离出来，通过单独的XML文件定义界面布局，就像网站使用HTML文件定义网页那样。直观地看，网站的前后端分离设计如图2-14所示，App的前后端分离设计如图2-15所示。

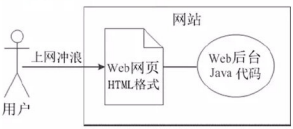


图 2-14　网站的前后端分离设计

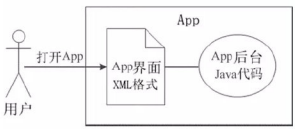


图 2-15　App 的前后端分离设计

把界面设计与代码逻辑分开，不仅能做到App的前后端分离，还有下列几点好处。

（1）使用XML文件描述App界面，可以很方便地在Android Studio上预览界面效果。比如新创建的App项目，默认首页布局为activity_main.xml，单击界面右上角的Design按钮，即可看到如图2-16所示的预览界面。

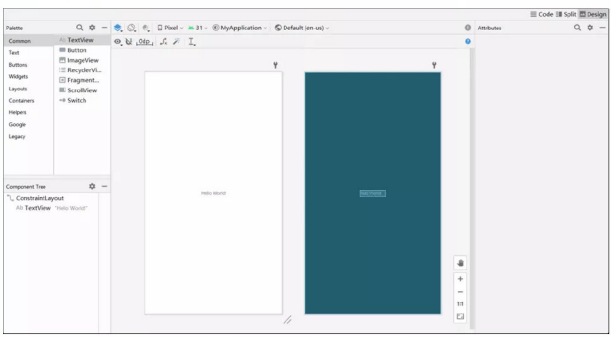


图 2-16　XML 文件的预览界面

如果XML文件修改了Hello World的文字内容，立刻就能在预览区域观看最新界面。倘若使用Java代码描绘界面，那么必须运行App才能看到App界面，无疑费时许多。

（2）一个界面布局可以被多处代码复用，比如看图界面，既能通过商城购物代码浏览商品图片，也能通过商品评价代码浏览买家晒单。

（3）反过来，一段Java代码也可能适配多个界面布局，比如手机有竖屏与横屏两种模式，默认App采用同一套布局，然而在竖屏时很紧凑的界面布局（见图2-17），切换到横屏往往变得松垮乃至变形（见图2-18）。



图 2-17　竖屏时候的界面布局

鉴于竖屏与横屏遵照一样的业务逻辑，仅仅是屏幕方向不同，若要调整的话，只需分别给出竖屏时候的界面布局，以及横屏时候的界面布局。因为多数用户习惯竖屏浏览，所以res/layout目录下放置的XML文件默认为竖屏规格。要设置横屏的页面布局，首先在res下面新建名为layout-land的目录，用来存放横屏规格的XML文件。land是landscape的缩写，意思是横向，Android把layout-land作为横屏XML的专用布局目录。然后在layout-land目录创建与原XML同名的XML文件，并重新编排界面控件的展示方位。调整后的横屏界面如图2-19所示，有效适配了屏幕的水平方向。



图 2-18　横屏时候的界面布局



图 2-19　采用另一个 XML 文件的横屏布局

总的来说，界面设计与代码逻辑分离的好处多多，后续的例程都由XML布局与Java代码两部分组成。

2.3.2　利用 XML 标记描绘应用界面

在前面“2.1.2　App的开发语言”末尾，给出了安卓控件的XML定义例子，如下：

```
<TextView
    android:id="@+id/tv hello"
    android:layout width="wrap content"
    android:layout height="wrap content"
    android:text="Hello World!" />
```

注意到TextView标签以“<”开头，以“/>”结尾，为何尾巴多了个斜杆呢？要是没有斜杆，以左右尖括号包裹标签名称，岂不更好？其实这是XML的标记规范，凡是XML标签都由标签头与标签尾组成，标签头以左右尖括号包裹标签名称，形如“<TextView>”；标签尾在左尖括号后面插入斜杆，以此同标签头区分开，形如“</TextView>”。标签头允许在标签名称后面添加各种属性取值，而标签尾不允许添加任何属性，因此上述TextView标签的完整XML定义是下面这样的：

```
<TextView
    android:id="@+id/tv hello"
    android:layout width="wrap content"
    android:layout_height="wrap_content"
    android:text="Hello World!" >
</TextView>
```

考虑到TextView仅仅是个文本视图，其标签头和标签尾之间不会插入其他标记，所以合并它的标签头和标签尾，也就是让TextView标签以“/>”结尾，表示该标签到此为止。

然而不是所有情况都能采取简化写法，简写只适用于TextView控件这种末梢节点。好比一棵大树，大树先有树干，树干分岔出树枝，一些大树枝又分出小树枝，树枝再长出末端的树叶。一个界面也是先有根节点（相当于树干），根节点下面挂着若干布局节点（相当于树枝），布局节点下面再挂着控件节点（相当于树叶）。因为树叶已经是末梢了，不会再包含其他节点，所以末梢节点允许采用“/>”这种简写方式。

譬如下面是个XML文件的布局内容，里面包含了根节点、布局节点，以及控件节点：

```
(完整代码见chapter02\src\main\res\layout\activity_main.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout height="match parent">
    <!-- 这是个线性布局， match_parent意思是与上级视图保持一致-->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout height="match parent">
        <!-- 这是个文本视图，名称为tv hello，显示的文字内容为“Hello World!” -->
        <TextView
            android:id="@+id/tv_hello"
            android:layout_width="wrap_content"
            android:layout height="wrap content"
            android:text="Hello World!" />
    </LinearLayout>
</LinearLayout>
```

上面的XML内容，最外层的LinearLayout标签为该界面的根节点，中间的LinearLayout标签为布局节点，最内层的TextView为控件节点。由于根节点和布局节点都存在下级节点，因此它们要有配对的标签头与标签尾，才能将下级节点包裹起来。根节点其实是特殊的布局节点，它的标签名称可以跟布局节点一样，区别之处在于下列两点：

（1）每个界面只有一个根节点，却可能有多个布局节点，也可能没有中间的布局节点，此时所有控件节点都挂在根节点下面。

（2）根节点必须配备“xmlns:android="http://schemas.android.com/apk/res/android"”，表示指定XML内部的命名空间，有了这个命名空间，Android Studio会自动检查各节点的属性名称是否合法，如果不合法就提示报错。至于布局节点就不能再指定命名空间了。

有了根节点、布局节点、控件节点之后，XML内容即可表达丰富多彩的界面布局，因为每个界面都能划分为若干豆腐块，每个豆腐块再细分为若干控件罢了。三种节点之外，尚有“<!—说明文字 -->”这类注释标记，它的作用是包裹注释性质的说明文字，方便其他开发者理解此处的XML含义。

2.3.3 使用 Java 代码书写程序逻辑

在XML文件中定义界面布局，已经明确是可行的了，然而这只是静态界面，倘若要求在App运行时修改文字内容，该如何是好？倘若是动态变更网页内容，还能在HTML文件中嵌入JavaScript代码，由JS片段操作Web控件。但Android的XML文件仅仅是布局标记，不能再嵌入其他语言的代码了，也就是说，只靠XML文件自身无法动态刷新某个控件。

即然XML表达不了复杂的业务逻辑，那么这副重担就得交给App后台的Java代码了。Android Studio每次创建新项目，除了生成默认的首页布局activity_main.xml之外，还会生成与其对应的代码文件MainActivity.java。该Java文件中MainActivity类的内容如下：

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

可见MainActivity.java的代码内容很简单，只有一个MainActivity类，该类下面只有一个onCreate方法。注意onCreate内部的setContentView方法直接引用了布局文件的名字activity_main，该方法的意思是往当前活动界面填充activity_main.xml的布局内容。现在准备在这里改动，把文字内容改成中文：首先打开activity_main.xml，在TextView节点下方补充一行android:id="@+id/tv_hello"，表示给它起个名字编号；然后回到MainActivity.java，在setContentView方法下面补充几行代码，具体如下：

```
(完整代码见chapter02\src\main\java\com\example\chapter02\MainActivity.java)
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // 当前的页面布局采用的是res/layout/activity_main.xml
        setContentView(R.layout.activity_main);
        // 获取名为tv_hello的TextView控件，注意添加导包语句import android.widget.TextView;
        TextView tv_hello = findViewById(R.id.tv_hello);
        // 设置TextView控件的文字内容
        tv_hello.setText("你好，世界");
```

新增的两行代码主要做了这些事情：先调用findViewById方法，从布局文件中取出名为tv_hello的TextView控件；再调用控件对象的setText方法，为其设置新的文字内容。

代码补充完毕，重新运行测试App，发现应用界面变成了如图2-20所示的样子。

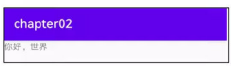


图 2-20　修改控件文本后的界面效果

可见使用Java代码成功修改了界面控件的文字内容。

2.4　App 的活动页面

本节介绍App活动页面的基本操作：首先分三步创建新的App页面，然后通过活动创建菜单快速生成页面源码，最后说明如何在代码中跳转到新的活动页面。

2.4.1　创建新的 App 页面

每次创建新的项目，都会生成默认的activity_main.xml和MainActivity.java，它们正是App首页对应的XML文件和Java代码。若要增加新的页面，就得由开发者自行操作了，完整的页面创建过程包括：创建XML文件、创建Java代码、注册页面配置。分别介绍如下：

1. 创建XML文件

创建XML文件的具体步骤如下：

01 在Android Studio左上方找到项目结构图，右击res目录下面的layout，在右键菜单中依次选择New→XML→Layout XML File，如图2-21所示。

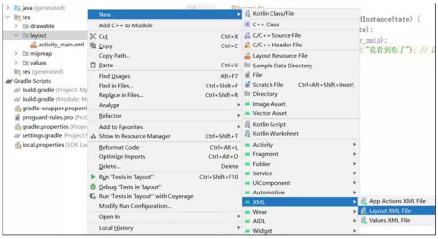


图 2-21　通过右键菜单创建 XML 文件

02 弹出如图2-22所示的XML创建对话框。在XML创建对话框的Layout File Name输入框中填写XML文件名，例如activity_main2。

03 单击窗口右下角的Finish按钮，之后便会在layout目录下面看到新创建的XML文件activity_main2.xml。双击它即可打开该XML的编辑窗口，再往其中填写详细的布局内容。

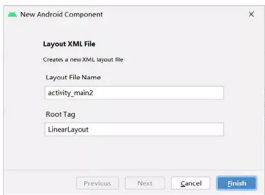


图 2-22　XML 创建对话框

2. 创建Java代码

创建Java代码的具体步骤如下：

01 同样在Android Studio左上方找到项目结构图，右击java目录下面的第一个包，在右键菜单中依次选择New→Java Class，如图2-23所示。

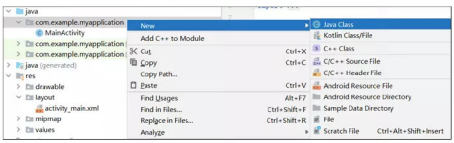


图 2-23　通过右键菜单创建 Java 代码

02 弹出如图2-24所示的Java代码创建窗口。在代码创建窗口的Name输入框中填写Java类名，例如Main2Activity。

03 单击窗口下方的OK按钮，之后便会在Java包下面看到新创建的代码文件Main2Activity.java。双击它即可打开代码编辑窗口，再往其中填写如下代码，表示加载来自activity_main2的页面布局。

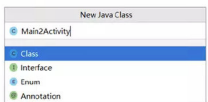


图 2-24　Java 代码的创建窗口

```
(完整代码见chapter02\src\main\java\com\example\chapter02\Main2Activity.java)
public class Main2Activity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
    }
}
```

3. 注册页面配置

创建好了页面的XML文件及其Java代码，还得在项目中注册该页面，打开AndroidManifest.xml，在application节点内部补充如下一行配置：

```
<activity android:name=".Main2Activity"></activity>
```

添加了上面这行配置，表示给该页面注册身份，否则App运行时打开页面会提示错误“activity not found”。如果activity的标记头与标记尾中间没有其他内容，则节点配置也可省略为下面这样：

```
<activity android:name=".Main2Activity /">
```

至此，才算创建了一个合法的新页面。

2.4.2　快速生成页面源码

上一小节经过创建XML文件、创建Java代码、注册页面配置后，就算创建好了一个新页面。没想到区区一个页面也这么费事，怎样才能提高开发效率呢？其实Android Studio早已集成了快速创建页面的功能，只要一个对话框就能完成所有操作。具体步骤如下：

01 仍旧在项目结构图中，右击java目录下面的包名，在右键菜单中依次选择New→Activity→Empty Activity，如图2-25所示。

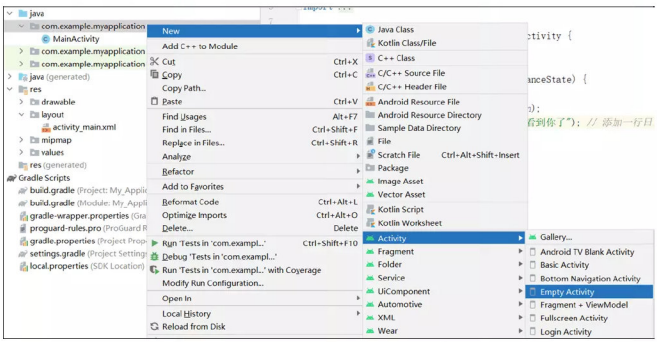


图 2-25　通过右键菜单创建活动页面

02 弹出如图2-26所示的页面创建对话框。在页面创建对话框的Activity Name输入框中填写页面的Java类名（例如Main2Activity），此时下方的Layout Name输入框会自动填写对应的XML文件名（例如activity_main2）。

03 单击对话框右下角的Finish按钮，完成新页面的创建动作。

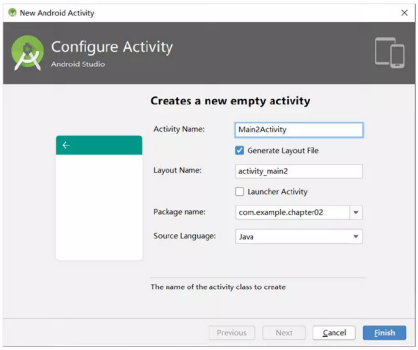


图 2-26 页面创建对话框

回到Android Studio左上方的项目结构图，发现res的layout目录下多了个activity_main2.xml，同时java目录下多了个Main2Activity，并且Main2Activity代码已经设定了加载activity_main2布局。接着打开AndroidManifest.xml，找到application节点发现多了下面这行配置：

```
<activity android:name=".Main2Activity"></activity>
```

检查结果说明，只要填写一个创建页面对话框，即可实现页面的创建。

2.4.3 跳转到另一个页面

一旦创建好新页面，就得在合适的时候跳转到该页面。假设出发页面为A，到达页面为B，那么跳转动作是从A跳转到B。由于启动App会自动打开默认主页MainActivity，因此跳跃的起点理所当然在MainActivity，跳跃的终点则为目标页面的Activity。这种跳转动作翻译为Android代码，格式形如“startActivity(new Intent(源页面.this，目标页面.class));”。如果目标页面名为Main2Activity，跳转代码便是下面这样的：

```
// 活动页面跳转，从MainActivity跳转到Main2Activity
startActivity(new Intent(MainActivity.this, Main2Activity.class));
```

因为跳转动作通常发生在当前页面，也就是从当前页面跳转到其他页面，所以不产生歧义的话，可以使用this指代当前页面。简化后的跳转代码如下：

```
startActivity(new Intent(this, Main2Activity.class));
```

接下来做个实验，准备让App启动后在首页停留3秒，3秒之后跳转到新页面Main2Activity。此

处的延迟处理功能，用到了Handler工具的postDelayed方法，该方法的第一个参数为待处理的Runnable任务对象，第二个参数为延迟间隔（单位为毫秒）。为此在MainActivity.java中补充以下的跳转处理代码：

```
(完整代码见chapter02\src\main\java\com\example\chapter02\MainActivity.java)
    @Override
    protected void onResume() {
        super.onResume();
        goNextPage();  // 跳转到下个页面
    }
    // 跳转到下个页面
    private void goNextPage() {
        TextView tv_hello = findViewById(R.id.tv_hello);
        tv_hello.setText("3秒后进入下个页面");
        // 延迟3秒（3000毫秒）后启动任务mGoNext
        new Handler().postDelayed(mGoNext, 3000);
    }
    private Runnable mGoNext = new Runnable() {
        @Override
        public void run() {
            // 活动页面跳转，从MainActivity跳转到Main2Activity
            startActivity(new Intent(MainActivity.this, Main2Activity.class));
        }
    };
```

运行测试App，刚打开的App界面如图2-27所示，过了3秒发生跳转事件，跳转后的App界面如图2-28所示，可见成功跳转到了新页面。

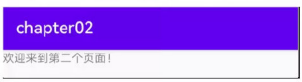


图 2-27　跳转之前的 App 界面



图 2-28　跳转之后的 App 界面

当然，以上的跳转代码有些复杂，比如：Intent究竟是什么？为何在onResume方法中执行跳转动作？Handler工具的处理机制是怎样的？对于这些疑问，后续章节将会逐渐展开，一层一层拨开Android开发的迷雾。

2.5　小　　结

本章主要介绍了App开发必须事先掌握的基础知识，包括App的开发特点（App的运行环境、App的开发语言、App访问的数据库）、App的工程结构（App工程的目录结构、编译配置文件build.gradle、运行配置文件AndroidManifest.xml）、App的设计规范（界面设计与代码逻辑、利用XML标记描绘应用界面、使用Java代码书写程序逻辑）、App的活动页面（创建新的App页面、快速生成页面源码、跳转到另一个页面）。

通过本章的学习，读者应该了解App开发的基本概念，并且熟悉App工程的组织形式，同时能使用Android Studio完成一些简单操作。

2.6 动手练习

请上机实验修改App工程的XML文件和Java代码，并使用真机调试App，主要步骤说明如下：

1. 创建一个新的App项目。
2. 修改项目级别的build.gradle，添加阿里云的仓库地址。
3. 创建一个名为Main2Activity的新页面（含XML文件与Java代码）。
4. 在该页面的XML文件中添加一个TextView标签，文本内容为“你好，世界！”。
5. 在MainActivity的Java代码中添加页面跳转代码，从当前页面跳转到Main2Activity。
6. 把App安装到手机上并运行，观察能否看到“你好，世界！”字样。

第 3 章

简 单 控 件

本章介绍App开发常见的几类简单控件的用法，主要包括：显示文字的文本视图、容纳视图的常用布局、响应点击的按钮控件、显示图片的图像视图等。然后结合本章所学的知识，演示一个实战项目“简单计算器”的设计与实现。

3.1 文 本 显 示

本节介绍如何在文本视图TextView上显示规定的文本，包括：怎样在XML文件和Java代码中设置文本内容，尺寸的大小有哪些单位、又该怎样设置文本的大小，颜色的色值是如何表达的、又该怎样设置文本的颜色。

3.1.1 设置文本的内容

在前一章的“2.3.3 使用Java代码书写程序逻辑”节，给出了设置文本内容的两种方式，一种是在XML文件中通过属性android:text设置文本，比如下面这样：

```
(完整代码见chapter03\src\main\res\layout\activity text view.xml)
<TextView
    android:id="@+id/tv_hello"
    android:layout_width="wrap_content"
    android:layout height="wrap content"
    android:text="你好，世界" />
```

另一种是在Java代码中调用文本视图对象的setText方法设置文本，比如下面这样：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\TextViewActivity.java)
        // 获取名为tv hello的文本视图
        TextView tv_hello = findViewById(R.id.tv_hello);
        tv_hello.setText("你好，世界");  // 设置tv_hello的文字内容
```

在XML文件中设置文本的话，把鼠标移到“你好，世界”上方时，Android Studio会弹出如图3-1所示的提示框。



图 3-1 XML 文件提示字符串硬编码

提示内容为“Hardcoded string "你好，世界", should use @string resouce”，意思是说这几个字是硬编码的字符串，建议使用来自@string的资源。Android Studio不推荐在XML布局文件里直接写字符串，因为可能有好几个页面都显示“你好，世界”，若想把这句话换成“你吃饭了吗？”，就得一个一个XML文件改过去，无疑费时费力。故而Android Studio推荐把字符串放到专门的地方进行管理，这个名为@string的地方位于res/values目录下的strings.xml，打开该文件发现它的初始内容如下：

```
<resources>
    <string name="app name">chapter03</string>
</resources>
```

strings.xml定义了一个名为“app_name”的字符串常量，其值为“chapter03”。在此添加新的字符串定义，字符串名为“hello”，字符串值为“你好，世界”，添加之后的strings.xml内容如下：

```
<resources>
    <string name="app name">chapter03</string>
    <string name="hello">你好，世界</string>
</resources>
```

添加完新的字符串定义，回到XML布局文件，将android:text属性值改为“@string/字符串名”这般，也就是“@string/hello”，修改之后的TextView标签示例如下：

```
<TextView
    android:id="@+id/tv hello"
    android:layout width="wrap content"
    android:layout_height="wrap_content"
    android:text="@string/hello" />
```

然后把鼠标移到“你好，世界”上方，此时Android Studio不再弹出任何提示了。

若要在Java代码中引用字符串资源，则需要在调用setText方法时填写形如“R.string.字符串名”的参数，就本例而言填入“R.string.hello”，修改之后的Java代码示例如下：

```
// 获取名为tv hello的文本视图
TextView tv_hello = findViewById(R.id.tv_hello);
tv_hello.setText(R.string.hello);  // 设置tv_hello的文字资源
```

至此不管是XML文件还是Java代码都从strings.xml引用字符串资源，以后想把“你好，世界”改为其他文字的话，只需改动strings.xml一个地方即可。

3.1.2 设置文本的大小

TextView允许设置文本内容，也允许设置文本大小，在Java代码中调用setTextSize方法，即可指定文本大小，就像以下代码这样：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\TextSizeActivity.java)
// 从布局文件中获取名为tv sp的文本视图
TextView tv_sp = findViewById(R.id.tv_sp);
tv_sp.setTextSize(30);  // 设置tv_sp的文本大小
```

这里的大小数值越大，则看到的文本也越大；大小数值越小，则看到的文本也越小。在XML文件中则通过属性android:textSize指定文本大小，可是如果给TextView标签添加“android:textSize="30"”，数字马上变成红色，鼠标移过去还会提示错误“Cannot resolve symbol '30'”，意思是无法解析“30”这个符号，如图3-2所示。

原来文本大小存在不同的字号单位，XML文件要求在字号数字后面写明单位类型，常见的字号单位主要有px、dp、sp 3种，分别介绍如下。



图 3-2 textSize 属性值只填数字时报错

1. px

px是手机屏幕的最小显示单位，它与设备的显示屏有关。一般来说，同样尺寸的屏幕（比如6英寸手机），如果看起来越清晰，则表示像素密度越高，以px计量的分辨率也越大。

2. dp

dp有时也写作dip，指的是与设备无关的显示单位，它只与屏幕的尺寸有关。一般来说，同样尺寸的屏幕以dp计量的分辨率是相同的，比如同样是6英寸手机，无论它由哪个厂家生产，其分辨率换算成dp单位都是一个大小。

3. sp

sp的原理跟dp差不多，但它专门用来设置字体大小，也是Android推荐的字号单位。手机在系统设置里可以调整字体的大小（小、标准、大、超大）。设置普通字体时，同数值dp和sp的文字看起来一样大；如果设置为大字体，用dp设置的文字没有变化，用sp设置的文字就变大了。

字体大小采用不同单位的话，显示的文字大小各不相同。例如，30px、30dp、30sp这3个字号，在不同手机上的显示大小有所差异。有的手机像素密度较低，1个dp相当于2个px，此时30px等同于15dp；有的手机像素密度较高，1个dp相当于3个px，此时30px等同于10dp。假设某个App的内部文本使用字号30px，则该App安装到前一部手机的字体大小为15dp，安装到后一部手机的字体大小为10dp，显然后一部手机显示的文本会更小。

至于dp与sp之间的区别，可通过以下实验加以观察。首先创建测试活动页面，该页面的XML文件分别声明30px、30dp、30sp这3个字号的TextView控件，布局内容如下：

```
（完整代码见chapter03\src\main\res\layout\activity_text_size.xml）
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout height="match parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="wrap_content"
        android:layout height="wrap content"
        android:text="你好，世界（px大小）"
        android:textSize="30px" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="你好，世界（dp大小）"
        android:textSize="30dp" />
    <TextView
        android:layout_width="wrap_content"
        android:layout height="wrap content"
        android:text="你好，世界（sp大小）"
        android:textSize="30sp" />
</LinearLayout>
```

接着打开手机的设置菜单，依次选择“显示”→“字体与显示大小”，确认当前的字体为标

准大小，如图3-3所示。然后在手机上运行测试App，进入测试页面看到的文字效果如图3-4所示。

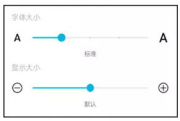


图 3-3　系统默认字体是标准大小



图 3-4　标准字体时的演示界面

回到设置菜单的字体页面，将系统字体大小调整为大号，如图3-5所示。再次进入测试页面看到的文字效果如图3-6所示。

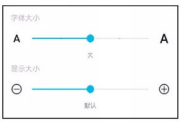


图 3-5　把系统字体改为大号

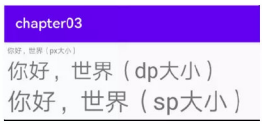


图 3-6　大号字体时的演示界面

对照图3-4和图3-6，发现字号单位30px和30dp的文字大小不变，而30sp的文字随着系统字体一起变大了。

既然XML文件要求android:textSize必须指定字号单位，为什么Java代码调用setTextSize只填数字不填单位呢？查看SDK源码，找到setTextSize方法的实现代码如下：

```
public void setTextSize(float size) {
    setTextSize(TypedValue.COMPLEX_UNIT_SP, size);
}
```

原来纯数字的setTextSize方法，内部默认字号单位为sp（COMPLEX_UNIT_SP），这也从侧面印证了之前的说法：sp是Android推荐的字号单位。

3.1.3　设置文本的颜色

除了文字大小外，文字颜色也经常需要修改，毕竟Android默认的灰色文字不够醒目。在Java代码中调用setTextColor方法即可设置文本颜色，具体在Color类中定义了12种颜色，详细的取值说明见表3-1。

表 3-1　颜色类型的取值说明

Color 类中的颜色类型	说　明	Color 类中的颜色类型	说　明
BLACK	黑色	GREEN	绿色
DKGRAY	深灰	BLUE	蓝色
GRAY	灰色	YELLOW	黄色
LTGRAY	浅灰	CYAN	青色
WHITE	白色	MAGENTA	玫红
RED	红色	TRANSPARENT	透明

比如以下代码便将文本视图的文字颜色改成了绿色：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\TextColorActivity.java)
// 从布局文件中获取名为tv_code_system的文本视图
TextView tv_code_system = findViewById(R.id.tv_code_system);
// 将tv_code_system的文字颜色设置为系统自带的绿色
tv_code_system.setTextColor(Color.GREEN);
```

由于XML文件无法引用Color类的颜色常量，为此Android制定了一套规范的编码标准，将色值交由透明度alpha和RGB三原色（红色red、绿色green、蓝色blue）联合定义。该标准又有8位十六进制数与6位十六进制数两种表达方式，例如8位编码FFEEDDCC中，FF表示透明度，EE表示红色的浓度，DD表示绿色的浓度，CC表示蓝色的浓度。透明度为FF表示完全不透明，为00表示完全透明。RGB三色的数值越大，表示颜色越浓，也就越暗；数值越小，表示颜色越淡，也就越亮。RGB亮到极致就是白色，暗到极致就是黑色。

至于6位十六进制编码，则有两种情况：它在XML文件中默认不透明（等价于透明度为FF），在代码中默认透明（等价于透明度为00）。以下代码给两个文本视图分别设置6位色值与8位色值，注意添加0x前缀表示十六进制数：

```
// 从布局文件中获取名为tv_code_six的文本视图
TextView tv_code_six = findViewById(R.id.tv_code_six);
// 将tv_code_six的文字颜色设置为透明的绿色，透明就是看不到
tv_code_six.setTextColor(0x00ff00);
// 从布局文件中获取名为tv_code_eight的文本视图
TextView tv_code_eight = findViewById(R.id.tv_code_eight);
// 将tv_code_eight的文字颜色设置为不透明的绿色，即正常的绿色
tv_code_eight.setTextColor(0xff00ff00);
```

运行测试App，发现tv_code_six控件的文本不见了（其实是变透明了），而tv_code_eight控件的文本显示正常的绿色。

在XML文件中可通过属性android:textColor设置文本颜色，但要给色值添加井号前缀（#），设定文本颜色的TextView标签示例如下：

```
(完整代码见chapter03\src\main\res\layout\activity_text_color.xml)
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="布局文件设置6位文字颜色"
    android:textColor="#00ff00" />
```

就像字符串资源那样，Android把颜色也当作一种资源，打开res/values目录下的colors.xml，发现里面已经定义了3种颜色：

```
<resources>
    <color name="colorPrimary">#008577</color>
    <color name="colorPrimaryDark">#00574B</color>
    <color name="colorAccent">#D81B60</color>
</resources>
```

那么先在resources节点内部补充如下的绿色常量定义：

```
<color name="green">#00ff00</color>
```

然后回到XML布局文件，把android:textColor的属性值改为“@color/颜色名称”，也就是android:textColor="@color/green"，修改之后的TextView标签如下：

```
<TextView
    android:layout width="wrap content"
    android:layout height="wrap content"
    android:text="资源文件引用6位文字颜色"
    android:textColor="@color/green" />
```

不仅文本颜色，还有背景颜色也会用到上述的色值定义，在XML文件中通过属性android:background设置控件的背景颜色。Java代码则有两种方式设置背景颜色：倘若色值来自Color类或十六进制数，则调用setBackgroundColor方法设置背景颜色；倘若色值来自colors.xml中的颜色资源，则调用setBackgroundResource方法，以“R.color.颜色名称”的格式设置背景颜色。下面是两种方式的背景颜色设定代码例子：

```
// 从布局文件中获取名为tv code background的文本视图
TextView tv_code_background = findViewById(R.id.tv_code_background);
// 将tv_code_background的背景颜色设置为绿色
tv code background.setBackgroundColor(Color.GREEN);          // 在代码中定义的色值
tv_code_background.setBackgroundResource(R.color.green);     // 颜色来自资源文件
```

注意属性android:background和setBackgroundResource方法，它俩用来设置控件的背景，不单单是背景颜色，还包括背景图片。在设置背景图片之前，先将图片文件放到res/drawable***目录（以drawable开头的目录，不仅仅是drawable目录），然后把android:background的属性值改为“@drawable/不含扩展名的图片名称”，或者调用setBackgroundResource方法填入“R.drawable.不含扩展名的图片名称”。

3.2 视图基础

本节介绍视图的几个基本概念及其用法，包括：如何设置视图的宽度和高度，如何设置视图的外部间距和内部间距，如何设置视图的外部对齐方式和内部对齐方式，等等。

3.2.1 设置视图的宽和高

手机屏幕是块长方形区域，较短的那条边叫作宽，较长的那条边叫作高。App控件通常也是长方形形状，控件宽度通过属性android:layout_width表达，控件高度通过属性android:layout_height表达，宽和高的取值主要有下列3种：

（1）match_parent：表示与上级视图保持一致。上级视图的尺寸有多大，当前视图的尺寸就有多大。

（2）wrap_content：表示与内容自适应。对于文本视图来说，内部文字需要多大的显示空间，当前视图就要占据多大的尺寸。但最宽不能超过上级视图的宽度，一旦超过就要换行；最高不能超过上级视图的高度，一旦超过就会被隐藏。

（3）以dp为单位的具体尺寸，比如300dp，表示宽度或者高度就是这么大。

在XML文件中采用以上任一方式均可设置视图的宽和高，但在Java代码中设置宽和高就有点复杂了，首先确保XML中的宽、高属性值为wrap_content，这样才允许在代码中修改宽和高。接着打开该页面对应的Java代码，依序执行以下3个步骤：

01 调用控件对象的getLayoutParams方法获取布局参数，参数类型为ViewGroup.LayoutParams。
02 布局参数的width属性表示宽度，height属性表示高度，修改这两个属性值，即可调整控件的宽和高。
03 调用控件对象的setLayoutParams方法，填入修改后的布局参数使之生效。

不过布局参数的width和height两个数值默认是px单位，需要将dp单位的数值转换为px单位的数值，然后才能赋值给width属性和height属性。下面是把dp大小转为px大小的方法代码：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\util\Utils.java)
// 根据手机的分辨率从 dp 的单位转成为 px(像素)
public static int dip2px(Context context, float dpValue) {
    // 获取当前手机的像素密度（1个dp对应几个px）
    float scale = context.getResources().getDisplayMetrics().density;
    return (int) (dpValue * scale + 0.5f);  // 四舍五入取整
}
```

有了上面定义的公共方法dip2px，就能将某个dp数值转换成px数值，比如准备把文本视图的宽度改为300dp，那么调整宽度的Java代码示例如下：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\ViewBorderActivity.java)
    // 获取名为tv_code的文本视图
    TextView tv_code = findViewById(R.id.tv_code);
    // 获取tv_code的布局参数（含宽度和高度）
    ViewGroup.LayoutParams params = tv_code.getLayoutParams();
    // 修改布局参数中的宽度数值，注意默认是px单位，需要把dp数值转成px数值
    params.width = Utils.dip2px(this, 300);
    tv_code.setLayoutParams(params);  // 设置tv_code的布局参数
```

接下来通过演示页面并观察几种尺寸设置方式的界面效果，主要通过背景色区分当前视图的宽高范围，详细的XML文件内容如下：

```
(完整代码见chapter03\src\main\res\layout\activity_view_border.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="5dp"
        android:background="#00ffff"
        android:text="视图宽度采用wrap_content定义" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="5dp"
        android:background="#00ffff"
        android:text="视图宽度采用match_parent定义" />
    <TextView
        android:layout_width="300dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="5dp"
        android:background="#00ffff"
        android:text="视图宽度采用固定大小" />
    <TextView
        android:id="@+id/tv_code"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:layout_marginTop="5dp"
        android:background="#00ffff"
        android:text="通过代码指定视图宽度" />
</LinearLayout>
```

运行测试App，打开的演示界面如图3-7所示，依据背景色判断文本视图的边界，可见wrap_content方式刚好包住了文本内容，match_parent方式扩展到了与屏幕等宽，而300dp的宽度介于前两者之间（安卓手机的屏幕宽度基本为360dp）。



图 3-7　设置控件宽度的几种方式的效果

3.2.2　设置视图的间距

在上一小节末尾的XML文件中，每个TextView标签都携带新的属性android:layout_marginTop="5dp"，该属性的作用是让当前视图与上方间隔一段距离。同理，android:layout_marginLeft让当前视图与左边间隔一段距离，android:layout_marginRight让当前视图与右边间隔一段距离，android:layout_marginBottom让当前视图与下方间隔一段距离。如果上下左右都间隔同样的距离，还能使用android:layout_margin一次性设置四周的间距。

layout_margin不单单用于文本视图，还可用于所有视图，包括各类布局和各类控件。因为不管布局还是控件，它们统统由视图基类View派生而来，而layout_margin正是View的一个通用属性，所以View的子子孙孙都能使用layout_margin。在View的大家族中，视图组ViewGroup尤为特殊，它既是View的子类，又是各类布局的基类。布局下面能容纳其他视图，而控件却不行，这正源自ViewGroup的组装特性。View、ViewGroup、控件、布局四者的继承关系如图3-8所示。



图 3-8　视图家族的依赖继承关系

除了layout_margin之外，padding也是View的一个通用属性，它用来设置视图的内部间距，并且padding也提供了paddingTop、paddingBottom、paddingLeft、paddingRight四个方向的距离属性。同样是设置间距，layout_margin指的是当前视图与外部视图（包括上级视图和平级视图）之间的距离，而padding指的是当前视图与内部视图（包括下级视图和内部文本）之间的距离。为了观察外部间距和内部间距的差异，接下来做个实验，看看layout_margin与padding究竟有什么区别。

首先创建新的活动页面，并给该页面的XML文件填入以下的布局内容：

```
(完整代码见chapter03\src\main\res\layout\activity_view_margin.xml)
<!-- 最外层的布局背景为蓝色 -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="300dp"
    android:background="#00aaff"
    android:orientation="vertical">
    <!-- 中间层的布局背景为黄色 -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_margin="20dp"
```

```
        android:background="#ffff99"
        android:padding="60dp">
        <!-- 最内层的视图背景为红色 -->
        <View
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="#ff0000" />
    </LinearLayout>
</LinearLayout>
```

上面的XML文件有两层视图嵌套，第一层是蓝色背景布局里面放黄色背景布局，第二层是黄色背景布局里面放红色背景视图。中间层的黄色背景布局，同时设置了20dp的layout_margin，以及60dp的padding，其中padding是layout_margin的三倍宽（60/20=3）。接着运行测试App，看到的演示界面如图3-9所示。

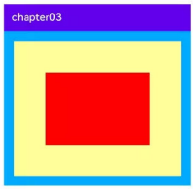


图 3-9 两种间距方式的演示效果

从效果图可见，外面一圈间隔较窄，里面一圈间隔较宽，表示20dp的layout_margin位于外圈，而60dp的padding位于内圈。这种情况印证了：layout_margin指的是当前图层与外部图层的距离，而padding指的是当前图层与内部图层的距离。

3.2.3 设置视图的对齐方式

App界面上的视图排列，默认靠左朝上对齐，这也符合日常的书写格式。然而页面的排版不是一成不变的，有时出于美观或者其他原因，要将视图排列改为朝下或靠右对齐，为此需要另外指定视图的对齐方式。在XML文件中通过属性android:layout_gravity可以指定当前视图的对齐方向，当属性值为top时表示视图朝上对齐，为bottom时表示视图朝下对齐，为left时表示视图靠左对齐，为right时表示视图靠右对齐。如果希望视图既朝上又靠左，则用竖线连接top与left，此时属性标记为android:layout_gravity="top|left"；如果希望视图既朝下又靠右，则用竖线连接bottom与right，此时属性标记为android:layout_gravity="bottom|right"。

注意layout_gravity规定的对齐方式，指的是当前视图往上级视图的哪个方向对齐，并非当前视图的内部对齐。若想设置内部视图的对齐方向，则需由当前视图的属性android:gravity指定，该属性一样拥有top、bottom、left、right 4种取值及其组合。它与layout_gravity的不同之处在于：layout_gravity设定了当前视图相对于上级视图的对齐方式，而gravity设定了下级视图相对于当前视图的对齐方式；前者决定了当前视图的位置，而后者决定了下级视图的位置。

为了进一步分辨layout_gravity与gravity的区别，接下来做个实验，对某个布局视图同时设置android:layout_gravity和android:gravity属性，再观察内外视图的对齐情况。下面便是实验用的XML文件例子：

```
(完整代码见chapter03\src\main\res\layout\activity_view_gravity.xml)
<!-- 最外层的布局背景为橙色，它的下级视图在水平方向排列 -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="300dp"
    android:background="#ffff99"
    android:padding="5dp">
    <!-- 第一个子布局背景为红色，它在上级视图中朝下对齐，它的下级视图则靠左对齐 -->
    <LinearLayout
        android:layout_width="0dp"
```

```
            android:layout_height="200dp"
            android:layout weight="1"
            android:layout gravity="bottom"
            android:gravity="left"
            android:background="#ff0000"
            android:layout_margin="10dp"
            android:padding="10dp">
            <!-- 内部视图的宽度和高度都是100dp，且背景色为青色 -->
            <View
                android:layout_width="100dp"
                android:layout_height="100dp"
                android:background="#00ffff" />
        </LinearLayout>
        <!-- 第二个子布局背景为红色，它在上级视图中朝上对齐，它的下级视图则靠右对齐 -->
        <LinearLayout
            android:layout_width="0dp"
            android:layout_height="200dp"
            android:layout weight="1"
            android:layout gravity="top"
            android:gravity="right"
            android:background="#ff0000"
            android:layout_margin="10dp"
            android:padding="10dp">
            <!-- 内部视图的宽度和高度都是100dp，且背景色为青色 -->
            <View
                android:layout_width="100dp"
                android:layout_height="100dp"
                android:background="#00ffff" />
        </LinearLayout>
    </LinearLayout>
```

运行测试App，打开演示界面如图3-10所示。

由效果图可见，第一个子布局朝下，并且它的内部视图靠左；而第二个子布局朝上，并且它的内部视图靠右。对比XML文件中的layout_gravity和gravity取值，证明了二者的对齐情况正如之前所言：layout_gravity决定当前视图位于上级视图的哪个方位，而gravity决定了下级视图位于当前视图的哪个方位。

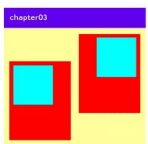


图 3-10　两种对齐方式的演示效果

3.3　常 用 布 局

本节介绍常见的几种布局用法，包括：在某个方向上顺序排列的线性布局，参照其他视图的位置相对排列的相对布局，像表格那样分行分列显示的网格布局，以及支持通过滑动操作拉出更多内容的滚动视图。

3.3.1　线性布局 LinearLayout

前几个小节的例程中，XML文件用到了LinearLayout布局，它的学名为线性布局。顾名思义，线性布局像是用一根线把它的内部视图串起来，故而内部视图之间的排列顺序是固定的，要么从左到右排列，要么从上到下排列。在XML文件中，LinearLayout通过属性android:orientation区分两种

方向，其中从左到右排列叫作水平方向，属性值为horizontal；从上到下排列叫作垂直方向，属性值为vertical。如果LinearLayout标签不指定具体方向，则系统默认该布局为水平方向排列，也就是默认android:orientation="horizontal"。

下面做个实验，让XML文件的根节点挂着两个线性布局，第一个线性布局采取水平方向，第二个线性布局采取垂直方向。然后每个线性布局内部各有两个文本视图，通过观察这些文本视图的排列情况，从而检验线性布局的显示效果。详细的XML文件内容如下：

```
(完整代码见chapter03\src\main\res\layout\activity_linear_layout.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout height="match parent"
    android:orientation="vertical">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout height="wrap content"
        android:orientation="horizontal">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="横排第一个" />
        <TextView
            android:layout width="wrap content"
            android:layout_height="wrap_content"
            android:text="横排第二个" />
    </LinearLayout>
    <LinearLayout
        android:layout width="match parent"
        android:layout height="wrap content"
        android:orientation="vertical">
        <TextView
            android:layout width="wrap content"
            android:layout height="wrap content"
            android:text="竖排第一个" />
        <TextView
            android:layout_width="wrap_content"
            android:layout height="wrap content"
            android:text="竖排第二个" />
    </LinearLayout>
</LinearLayout>
```

运行测试App，进入如图3-11所示的演示页面，可见horizontal为横向排列，vertical为纵向排列，说明android:orientation的方向属性确实奏效了。



图 3-11 线性布局的方向排列

除了方向之外，线性布局还有一个权重概念。所谓权重，指的是线性布局的下级视图各自拥有多大比例的宽和高。比如一块蛋糕分给两个人吃，可能两人平均分，也可能甲分三分之一，乙分三分之二。两人平均分的话，先把蛋糕平均切成两半，然后甲分到一半，乙分到另一半，此时甲、乙的权重比为1:1。甲分三分之一、乙分三分之二的话，先把蛋糕平均切成三块，然后甲分到一块，乙分到两块，此时甲、乙的权重比为1:2。就线性布局而言，它自身的尺寸相当于一整块蛋糕，它的下级视图们一起来分这一整块蛋糕，有的视图分得多，有的视图分得少。分多分少全凭每个视图分到了多大的权重而定，这个权重在XML文件中通过属性android:layout_weight来表达。

把线性布局看作蛋糕的话，分蛋糕的甲、乙两人就相当于线性布局的下级视图。假设线性布

局平均分为左、右两块，则甲视图和乙视图的权重比为1:1，意味着两个下级视图的layout_weight属性都是1。不过视图有宽、高两个方向，系统怎知layout_weight表示哪个方向的权重呢？所以这里有个规定，一旦设置了layout_weight属性值，便要求layout_width填0dp或者layout_height填0dp。如果layout_width填0dp，则layout_weight表示水平方向的权重，下级视图会从左往右分割线性布局；如果layout_height填0dp，则layout_weight表示垂直方向的权重，下级视图会从上往下分割线性布局。

按照左右均分的话，线性布局设置水平方向horizontal，且甲、乙两视图的layout_width都填0dp，layout_weight都填1，此时横排的XML片段示例如下：

```
(完整代码见chapter03\src\main\res\layout\activity_linear_layout.xml)
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#ff0000"
        android:orientation="horizontal">
        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:gravity="center"
            android:text="横排第一个" />
        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:gravity="center"
            android:text="横排第二个" />
    </LinearLayout>
```

按照上下均分的话，线性布局设置垂直方向vertical，且甲乙两视图的layout_height都填0dp，layout_weight都填1，此时竖排的XML片段示例如下：

```
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="100dp"
        android:background="#00ffff"
        android:orientation="vertical">
        <TextView
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:gravity="center"
            android:text="竖排第一个" />
        <TextView
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:gravity="center"
            android:text="竖排第二个" />
    </LinearLayout>
```

把上面两个片段放到新页面的XML文件中，其中第一个是横排区域，采用红色背景（色值为ff0000），第二个是竖排区域，采用青色背景（色值为00ffff）。重新运行测试App，打开的演示界面如图3-12所示，可见横排区域平均分为左、右两块，竖排区域平均分为上、下两块。

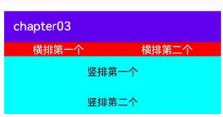


图 3-12　线性布局的权重分割

3.3.2　相对布局 RelativeLayout

线性布局的下级视图是顺序排列着的，另一种相对布局的下级视图位置则由其他视图决定。相对布局名为RelativeLayout，因为下级视图的位置是相对位置，所以得有具体的参照物才能确定最终位置。如果不设定下级视图的参照物，那么下级视图默认显示在RelativeLayout内部的左上角。

用于确定下级视图位置的参照物分两种：一种是与该视图自身平级的视图；另一种是该视图的上级视图（也就是它归属的RelativeLayout）。综合两种参照物，相对位置在XML文件中的属性取值说明见表3-2。

表 3-2　相对位置的属性取值说明

相对位置的属性取值	相对位置说明
layout_toLeftOf	当前视图在指定视图的左边
layout_toRightOf	当前视图在指定视图的右边
layout_above	当前视图在指定视图的上方
layout_below	当前视图在指定视图的下方
layout_alignLeft	当前视图与指定视图的左侧对齐
layout_alignRight	当前视图与指定视图的右侧对齐
layout_alignTop	当前视图与指定视图的顶部对齐
layout_alignBottom	当前视图与指定视图的底部对齐
layout_centerInParent	当前视图在上级视图中间
layout_centerHorizontal	当前视图在上级视图的水平方向居中
layout_centerVertical	当前视图在上级视图的垂直方向居中
layout_alignParentLeft	当前视图与上级视图的左侧对齐
layout_alignParentRight	当前视图与上级视图的右侧对齐
layout_alignParentTop	当前视图与上级视图的顶部对齐
layout_alignParentBottom	当前视图与上级视图的底部对齐

为了更好地理解上述相对属性的含义，接下来使用RelativeLayout及其下级视图进行布局来看看实际效果。下面是演示相对布局的XML文件例子：

```
(完整代码见chapter03\src\main\res\layout\activity_relative_layout.xml)
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="150dp" >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:background="#eeeeee"
        android:text="我在中间" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:background="#eeeeee"
        android:text="我在水平中间" />
    <TextView
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:layout centerVertical="true"
        android:background="#eeeeee"
        android:text="我在垂直中间" />
    <TextView
        android:layout_width="wrap_content"
        android:layout height="wrap content"
        android:layout alignParentLeft="true"
        android:background="#eeeeee"
        android:text="我跟上级左边对齐" />
    <TextView
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout alignParentRight="true"
        android:background="#eeeeee"
        android:text="我跟上级右边对齐" />
    <TextView
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout alignParentTop="true"
        android:background="#eeeeee"
        android:text="我跟上级顶部对齐" />
    <TextView
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout_alignParentBottom="true"
        android:background="#eeeeee"
        android:text="我跟上级底部对齐" />
    <TextView
        android:layout width="wrap content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@+id/tv_center"
        android:layout alignTop="@+id/tv center"
        android:background="#eeeeee"
        android:text="我在中间左边" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout toRightOf="@+id/tv center"
        android:layout alignBottom="@+id/tv center"
        android:background="#eeeeee"
        android:text="我在中间右边" />
    <TextView
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout above="@+id/tv center"
        android:layout_alignLeft="@+id/tv_center"
        android:background="#eeeeee"
        android:text="我在中间上面" />
    <TextView
        android:layout width="wrap content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/tv_center"
        android:layout alignRight="@+id/tv center"
        android:background="#eeeeee"
        android:text="我在中间下面" />
</RelativeLayout>
```

上述XML文件的布局效果如图3-13所示，RelativeLayout的下级视图都是文本视图，控件上的文字说明了所处的相对位置，具体的控件显示方位正如XML属性中描述的那样。

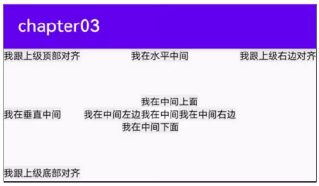


图 3-13 相对布局的相对位置效果

3.3.3 网格布局 GridLayout

虽然线性布局既能在水平方向排列，也能在垂直方向排列，但它不支持多行多列的布局方式，只支持单行（水平排列）或单列（垂直排列）的布局方式。若要实现类似表格那样的多行多列形式，可采用网格布局GridLayout。

网格布局默认从左往右、从上到下排列，它先从第一行从左往右放置下级视图，塞满之后另起一行放置其余的下级视图，如此循环往复直至所有下级视图都放置完毕。为了判断能够容纳几行几列，网格布局新增了android:columnCount与android:rowCount两个属性，其中columnCount指定了网格的列数，即每行能放多少个视图；rowCount指定了网格的行数，即每列能放多少个视图。

下面是运用网格布局的XML布局样例，它规定了一个两行两列的网格布局，且内部容纳四个文本视图。XML文件内容如下：

```
(完整代码见chapter03\src\main\res\layout\activity_grid_layout.xml)
<!-- 根布局为两行两列的网格布局，其中列数由columnCount指定，行数由rowCount指定 -->
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="2"
    android:rowCount="2">
    <TextView
        android:layout_width="180dp"
        android:layout_height="60dp"
        android:gravity="center"
        android:background="#ffcccc"
        android:text="浅红色" />
    <TextView
        android:layout_width="180dp"
        android:layout_height="60dp"
        android:gravity="center"
        android:background="#ffaa00"
        android:text="橙色" />
    <TextView
        android:layout_width="180dp"
        android:layout_height="60dp"
        android:gravity="center"
        android:background="#00ff00"
        android:text="绿色" />
    <TextView
        android:layout_width="180dp"
        android:layout_height="60dp"
        android:gravity="center"
        android:background="#660066"
```

```
        android:text="深紫色" />
</GridLayout>
```

在一个新建的活动页面加载上述布局，运行App观察到的界面如图3-14所示。



图 3-14　网格布局的视图分布情况

由图3-14可见，App界面的第一行分布着浅红色背景与橙色背景的文本视图，第二行分布着绿色背景与深紫色背景的文本视图，说明利用网格布局实现了多行多列的效果。

3.3.4　滚动视图 ScrollView

手机屏幕的显示空间有限，常常需要上下滑动或左右滑动才能拉出其余页面内容，可惜一般的布局节点都不支持自行滚动，这时就要借助滚动视图了。与线性布局类似，滚动视图也分为垂直方向和水平方向两类，其中垂直滚动视图名为ScrollView，水平滚动视图名为HorizontalScrollView。这两个滚动视图的使用并不复杂，主要注意以下3点：

（1）在垂直方向滚动时，layout_width属性值设置为match_parent，layout_height属性值设置为wrap_content。

（2）在水平方向滚动时，layout_width属性值设置为wrap_content，layout_height属性值设置为match_parent。

（3）滚动视图节点下面必须且只能挂着一个子布局节点，否则会在运行时报错Caused by：java.lang.IllegalStateException：ScrollView can host only one direct child。

下面是ScrollView和HorizontalScrollView的XML例子：

```
(完整代码见chapter03\src\main\res\layout\activity_scroll_view.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout height="match parent"
    android:orientation="vertical">
    <!-- HorizontalScrollView是水平方向的滚动视图，当前高度为200dp -->
    <HorizontalScrollView
        android:layout_width="wrap_content"
        android:layout height="200dp">
        <!-- 水平方向的线性布局，两个子视图的颜色分别为青色和黄色 -->
        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:orientation="horizontal">
            <View
                android:layout width="300dp"
                android:layout height="match parent"
                android:background="#aaffff" />
            <View
                android:layout width="300dp"
                android:layout height="match parent"
                android:background="#ffff00" />
        </LinearLayout>
    </HorizontalScrollView>
    <!-- ScrollView是垂直方向的滚动视图，当前高度为自适应 -->
    <ScrollView
        android:layout width="match parent"
        android:layout_height="wrap_content">
```

```
        <!-- 垂直方向的线性布局，两个子视图的颜色分别为绿色和橙色 -->
        <LinearLayout
            android:layout width="match parent"
            android:layout height="wrap content"
            android:orientation="vertical">
            <View
                android:layout width="match parent"
                android:layout height="400dp"
                android:background="#00ff00" />
            <View
                android:layout_width="match_parent"
                android:layout height="400dp"
                android:background="#ffffaa" />
        </LinearLayout>
    </ScrollView>
</LinearLayout>
```

运行测试App，可知ScrollView在纵向滚动，而HorizontalScrollView在横向滚动。

有时ScrollView的实际内容不够，又想让它充满屏幕，怎么办呢？如果把layout_height属性赋值为match_parent，结果还是不会充满，正确的做法是再增加一行属性android:fillViewport（该属性值为true表示允许填满视图窗口），属性片段举例如下：

```
android:layout_height="match_parent"
android:fillViewport="true"
```

3.4 按钮触控

本节介绍按钮控件的常见用法，包括：如何设置大小写属性与点击属性，如何响应按钮的点击事件和长按事件，如何禁用按钮又该如何启用按钮，等等。

3.4.1 按钮控件 Button

除了文本视图之外，按钮也是一种基础控件。因为Button是由TextView派生而来，所以文本视图拥有的属性和方法，包括文本内容、文本大小、文本颜色等，按钮控件均能使用。不同的是，Button拥有默认的按钮背景，而TextView默认无背景；Button的内部文本默认居中对齐，而TextView的内部文本默认靠左对齐。此外，按钮还要额外注意textAllCaps与onClick两个属性，这两个属性分别介绍如下：

1. textAllCaps属性

对于TextView来说，text属性设置了什么文本，文本视图就显示什么文本。但对于Button来说，不管text属性设置的是大写字母还是小写字母，按钮控件都默认转成大写字母显示。比如在XML文件中加入下面的Button标签：

```
<Button
    android:layout width="match parent"
    android:layout height="wrap content"
    android:text="Hello World" />
```

编译运行后的App界面，按钮上显示全大写的“HELLO WORLD”，而非原来大小写混合的“Hello World”。显然这个效果不符合预期，为此需要给Button标签补充textAllCaps属性，该属性

的值默认为true，表示全部转为大写，如果设置为false，则表示不转为大写。于是在布局文件添加新的Button标签，该标签补充了android:textAllCaps="false"，具体内容如下：

```
(完整代码见chapter03\src\main\res\layout\activity_button_style.xml)
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Hello World"
    android:textAllCaps="false" />
```

再次运行App，此时包含新、旧按钮的界面如图3-15所示，可见textAllCaps属性果然能够控制大小写转换。



图 3-15　textAllCaps 属性的效果

2. onClick属性

按钮之所以成为按钮，是因为它会响应按下动作，就手机而言，按下动作等同于点击操作，即手指轻触屏幕然后马上松开。每当点击按钮之时，就表示用户确认了某个事项，接下来轮到App进行处理了。onClick属性便用来接管用户的点击动作，该属性的值是个方法名，也就是当前页面的Java代码存在这么一个方法：当用户点击按钮时，就自动调用该方法。

例如下面的Button标签指定了onClick属性值为doClick，表示点击该按钮会触发Java代码中的doClick方法：

```
(完整代码见chapter03\src\main\res\layout\activity button style.xml)
<Button
    android:id="@+id/btn_click_xml"
    android:layout_width="match_parent"
    android:layout height="wrap content"
    android:onClick="doClick"
    android:text="直接指定点击方法" />
<TextView
    android:id="@+id/tv result"
    android:layout width="match parent"
    android:layout height="wrap content"
    android:text="这里查看按钮的点击结果" />
```

与之相对应，页面所在的Java代码需要增加doClick方法，方法代码示例如下：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\ButtonStyleActivity.java)
// activity_button_style.xml中给btn_click_xml指定了点击方法doClick
public void doClick(View view) {
    String desc = String.format("%s 您点击了按钮：%s",
            DateUtil.getNowTime(), ((Button) view).getText());
    tv result.setText(desc);  // 设置文本视图的文本内容
}
```

然后编译运行，并在App界面上点击新加的按钮，点击前、后的界面分别如图3-16和图3-17所示。

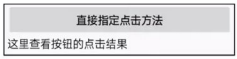


图 3-16　按钮点击之前的界面

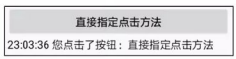


图 3-17　按钮点击之后的界面

比较图3-16和图3-17的文字差异，可见点击按钮之后确实调用了doClick方法。

3.4.2 点击事件和长按事件

虽然按钮控件能够在XML文件中通过onClick属性指定点击方法，但是方法的名称可以随便叫，既能叫doClick也能叫doTouch，甚至叫它doA或doB都没问题，这样很不利于规范化代码，倘若以后换了别人接手，就不知道doA或doB是干什么用的。因此在实际开发中，不推荐使用Button标签的onClick属性，而是在代码中给按钮对象注册点击监听器。

所谓监听器，意思是专门监听控件的动作行为，它平时无事可做，只有控件发生了指定的动作，监听器才会触发开关去执行对应的代码逻辑。点击监听器需要实现接口View.OnClickListener，并重写onClick方法补充点击事件的处理代码，再由按钮调用setOnClickListener方法设置监听器对象。比如下面的代码给按钮控件btn_click_single设置了一个点击监听器：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\ButtonClickActivity.java)
// 从布局文件中获取名为btn_click_single的按钮控件
Button btn_click_single = findViewById(R.id.btn_click_single);
// 给btn_click_single设置点击监听器，一旦用户点击按钮，就触发监听器的onClick方法
btn_click_single.setOnClickListener(new MyOnClickListener());
```

上面的点击监听器名为MyOnClickListener，它的定义代码示例如下：

```
// 定义一个点击监听器，它实现了接口View.OnClickListener
class MyOnClickListener implements View.OnClickListener {
    @Override
    public void onClick(View v) {          // 点击事件的处理方法
        String desc = String.format("%s 您点击了按钮：%s",
                DateUtil.getNowTime(), ((Button) v).getText());
        tv_result.setText(desc);           // 设置文本视图的文本内容
    }
}
```

接着运行App，点击按钮之后的界面如图3-18所示，可见点击动作的确触发了监听器的onClick方法。

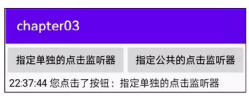


图 3-18 点击了单独的点击监听器

如果一个页面只有一个按钮，单独定义新的监听器倒也无妨，可是如果存在许多按钮，给每个按钮都定义自己的监听器，那就劳民伤财了。对于同时监听多个按钮的情况，更好的办法是注册统一的监听器，也就是让当前页面实现接口View.OnClickListener，如此一来，onClick方法便写在了页面代码之内。因为是统一的监听器，所以onClick内部需要判断是哪个按钮被点击了，也就是利用视图对象的getId方法检查控件编号，完整的onClick代码举例如下：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\ButtonClickActivity.java)
@Override
public void onClick(View v) {                          // 点击事件的处理方法
    if (v.getId() == R.id.btn_click_public) {     // 来自按钮btn_click_public
        String desc = String.format("%s 您点击了按钮：%s",
                DateUtil.getNowTime(), ((Button) v).getText());
        tv_result.setText(desc);                       // 设置文本视图的文本内容
    }
}
```

当然在该页面的onCreate内部别忘了调用按钮对象的setOnClickListener方法，把按钮的点击监听器设置成当前页面，设置代码如下：

```
// 从布局文件中获取名为btn_click_public的按钮控件
Button btn_click_public = findViewById(R.id.btn_click_public);
// 设置点击监听器，一旦用户点击按钮，就触发监听器的onClick方法
btn_click_public.setOnClickListener(this);
```

重新运行App，点击第二个按钮之后的界面如图3-19所示，可见当前页面的onClick方法也正确执行了。

除了点击事件外，Android还设计了另外一种长按事件，每当控件被按住超过500毫秒之后，就会触发该控件的长按事件。若要捕捉按钮的长按事件，可调用按钮对象的setOnLongClickListener方法设置长按监听器。具体的设置代码示例如下：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\ButtonLongclickActivity.java)
// 从布局文件中获取名为btn_click_public的按钮控件
Button btn_longclick_public = findViewById(R.id.btn_longclick_public);
// 设置长按监听器，一旦用户长按按钮，就触发监听器的onLongClick方法
btn_longclick_public.setOnLongClickListener(this);
```

以上代码把长按监听器设置到当前页面，意味着该页面需要实现对应的长按接口View.OnLongClickListener，并重写长按方法onLongClick，下面便是重写后的onLongClick代码例子：

```
@Override
public boolean onLongClick(View v) {                    // 长按事件的处理方法
    if (v.getId() == R.id.btn_longclick_public) {       // 来自btn_longclick_public
        String desc = String.format("%s 您长按了按钮：%s",
                DateUtil.getNowTime(), ((Button) v).getText());
        tv_result.setText(desc);                        // 设置文本视图的文本内容
    }
    return true;
}
```

再次运行App，长按按钮之后的界面如图3-20所示，说明长按事件果然触发了onLongClick方法。

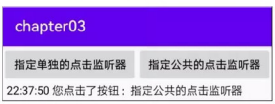


图 3-19　点击了公共的点击监听器

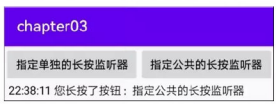


图 3-20　长按了公共的长按监听器

值得注意的是，点击监听器和长按监听器不局限于按钮控件，其实它们都来自视图基类View，凡是从View派生而来的各类控件，均可注册点击监听器和长按监听器。譬如文本视图TextView，其对象也能调用setOnClickListener方法与setOnLongClickListener方法，此时TextView控件就会响应点击动作和长按动作。因为按钮存在按下和松开两种背景，便于提示用户该控件允许点击，但文本视图默认没有按压背景，不方便判断是否被点击，所以一般不会让文本视图处理点击事件和长按事件。

3.4.3　禁用与恢复按钮

尽管按钮控件生来就是给人点击的，可是某些情况仍然希望暂时禁止点击操作，譬如用户在注册的时候，有的网站要求用户必须同意指定条款，而且至少浏览10秒之后才能点击注册按钮。那么在10秒之前，注册按钮应当置灰且不能点击，等过了10秒之后，注册按钮才恢复正常。在这样的业务场景中，按钮先后拥有两种状态，即不可用状态与可用状态，它们在外观和功能上的区别如下：

（1）不可用按钮：按钮不允许点击，即使点击也没反应，同时按钮文字为灰色。

（2）可用按钮：按钮允许点击，点击按钮会触发点击事件，同时按钮文字为正常的黑色。

从上述的区别说明可知，不可用与可用状态主要有两点差异：其一，是否允许点击；其二，按钮文字的颜色。就文字颜色而言，可在布局文件中使用textColor属性设置颜色，也可在Java代码中调用setTextColor方法设置颜色。至于是否允许点击，则需引入新属性android:enabled，该属性值为true时表示启用按钮，即允许点击按钮；该属性值为false时表示禁用按钮，即不允许点击按钮。在Java代码中，则可通过setEnabled方法设置按钮的可用状态（true表示启用，false表示禁用）。

接下来通过一个例子演示按钮的启用和禁用操作。为了改变测试按钮的可用状态，需要额外添加两个控制按钮，分别是“启用测试按钮”和“禁用测试按钮”，加起来一共3个按钮控件，注意“测试按钮”默认是灰色文本。“测试按钮”尚未启用时的界面效果如图3-21所示。



图 3-21 “测试按钮”尚未启用时的界面效果

与图3-21对应的布局文件内容如下：

```
(完整代码见chapter03\src\main\res\layout\activity button enable.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <LinearLayout
        android:layout width="match parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
        <Button
            android:id="@+id/btn enable"
            android:layout width="0dp"
            android:layout height="wrap content"
            android:layout_weight="1"
            android:text="启用测试按钮" />
        <Button
            android:id="@+id/btn disable"
            android:layout width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="禁用测试按钮" />
    </LinearLayout>
    <Button
        android:id="@+id/btn test"
        android:layout_width="match_parent"
        android:layout height="wrap content"
        android:enabled="false"
        android:text="测试按钮" />
    <TextView
        android:id="@+id/tv_result"
        android:layout_width="match_parent"
        android:layout height="wrap content"
        android:paddingLeft="5dp"
        android:text="这里查看测试按钮的点击结果" />
</LinearLayout>
```

然后在Java代码中给3个按钮分别注册点击监听器，注册代码如下：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\ButtonEnableActivity.java)
    // 因为按钮控件的setOnClickListener方法来自View基类，所以也可对findViewById得到的视图直接设置点击监听器
    findViewById(R.id.btn_enable).setOnClickListener(this);
    findViewById(R.id.btn_disable).setOnClickListener(this);
    btn_test = findViewById(R.id.btn_test);        // 获取名为btn_test的按钮控件
    btn_test.setOnClickListener(this);             // 设置btn_test的点击监听器
```

同时重写页面的onClick方法，分别处理3个按钮的点击事件，修改之后的onClick代码示例如下：

```
@Override
public void onClick(View v) {                      // 点击事件的处理方法
    // 由于多个控件都把点击监听器设置到了当前页面，因此onClick方法需要区分来自哪个按钮
    if (v.getId() == R.id.btn_enable) {            // 点击了按钮“启用测试按钮”
        btn_test.setTextColor(Color.BLACK);        // 设置按钮的文字颜色
        btn_test.setEnabled(true);                 // 启用当前控件
    } else if (v.getId() == R.id.btn_disable) {    // 点击了按钮“禁用测试按钮”
        btn_test.setTextColor(Color.GRAY);         // 设置按钮的文字颜色
        btn_test.setEnabled(false);                // 禁用当前控件
    } else if (v.getId() == R.id.btn_test) {       // 点击了按钮“测试按钮”
        String desc = String.format("%s 您点击了按钮：%s",
                DateUtil.getNowTime(), ((Button) v).getText());
        tv_result.setText(desc);                   // 设置文本视图的文本内容
    }
}
```

最后编译运行App，点击了“启用测试按钮”之后，原本置灰的测试按钮btn_test恢复正常的黑色文本，点击该按钮发现界面有了反应，具体效果如图3-22所示。

对比图3-21和图3-22，观察按钮启用前后的外观及其是否响应点击动作，即可知晓禁用按钮和启用按钮两种模式的差别。



图 3-22 “测试按钮”已经启用后的界面

3.5 图像显示

本节介绍与图像显示有关的几种控件及其用法，包括：专门用于显示图片的图像视图以及若干缩放类型效果，支持显示图片的按钮控件——图像按钮，如何在按钮控件上同时显示文本和图标等。

3.5.1 图像视图 ImageView

显示文本用到了文本视图TextView，显示图像则用到图像视图ImageView。由于图像通常保存为单独的图片文件，因此需要先把图片放到res/drawable目录中，然后再去引用该图片的资源名称。比如现在有张名为apple.png的苹果图片，那么XML文件通过属性android:src设置图片资源，属性值格式形如“@drawable/不含扩展名的图片名称”。添加了src属性的ImageView标签示例如下：

```
(完整代码见chapter03\src\main\res\layout\activity_image_scale.xml)
<ImageView
    android:id="@+id/iv_scale"
    android:layout_width="match_parent"
    android:layout_height="220dp"
    android:src="@drawable/apple" />
```

若想在Java代码中设置图像视图的图片资源，可调用ImageView控件的setImageResource方法，方法参数格式形如“R.drawable.不含扩展名的图片名称”。仍以上述的苹果图片为例，给图像视图设置图片资源的代码例子如下：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\ImageScaleActivity.java)
// 从布局文件中获取名为iv_scale的图像视图
ImageView iv_scale = findViewById(R.id.iv_scale);
iv_scale.setImageResource(R.drawable.apple);  // 设置图像视图的图片资源
```

运行测试App，展示图片的界面效果如图3-23所示。

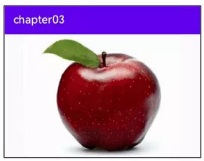


图 3-23 图像视图显示苹果图片

观察效果图发现苹果图片居中显示，而非像文本视图里的文字那样默认靠左显示，这是怎么回事？原来ImageView本身默认图片居中显示，不管图片有多大抑或有多小，图像视图都会自动缩放图片，使之刚好够着ImageView的边界，并且缩放后的图片保持原始的宽高比例，看起来图片很完美地占据了视图中央。这种缩放类型在XML文件中通过属性android:scaleType定义，即使图像视图未明确指定该属性，系统也会默认其值为fitCenter，表示让图像缩放后居中显示。添加了缩放属性的ImageView标签如下：

```
<ImageView
    android:id="@+id/iv_scale"
    android:layout_width="match_parent"
    android:layout_height="220dp"
    android:src="@drawable/apple"
    android:scaleType="fitCenter" />
```

在Java代码中可调用setScaleType方法设置图像视图的缩放类型，其中fitCenter对应的类型为ScaleType.FIT_CENTER，设置代码示例如下：

```
    // 将缩放类型设置为“保持宽高比例，缩放图片使其位于视图中间”
    iv_scale.setScaleType(ImageView.ScaleType.FIT_CENTER);
```

除了居中显示，图像视图还提供了其他缩放类型，详细的缩放类型取值说明见表3-3。

表 3-3 缩放类型的取值说明

XML 中的缩放类型	ScaleType 类中的缩放类型	说　明
fitCenter	FIT_CENTER	保持宽高比例，缩放图片使其位于视图中间
centerCrop	CENTER_CROP	缩放图片使其充满视图（超出部分会被裁剪），并位于视图中间
centerInside	CENTER_INSIDE	保持宽高比例，缩小图片使之位于视图中间（只缩小不放大）
center	CENTER	保持图片原始尺寸，并使其位于视图中间
fitXY	FIT_XY	缩放图片使其正好填满视图（图片可能被拉伸变形）
fitStart	FIT_START	保持宽高比例，缩放图片使其位于视图上方或左侧
fitEnd	FIT_END	保持宽高比例，缩放图片使其位于视图下方或右侧

注意，fitCenter是默认的缩放类型，它的图像效果如之前的图3-23所示。其余缩放类型的图像显示效果分别如图3-24～图3-29所示，其中图3-24为centerCrop的效果图，图3-25为centerInside的效果图，图3-26为center的效果图，图3-27为fitXY的效果图，图3-28为fitStart的效果图，图3-29为fitEnd的效果图。

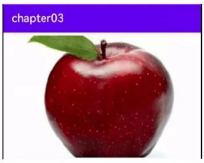


图 3-24 缩放类型为 centerCrop 的效果图

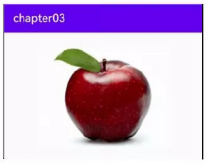


图 3-25 缩放类型为 centerInside 的效果图



图 3-26 缩放类型为 center 的效果图

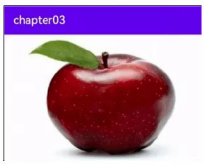


图 3-27 缩放类型为 fitXY 的效果图

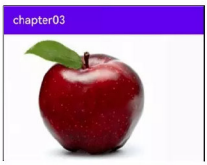


图 3-28 缩放类型为 fitStart 的效果图

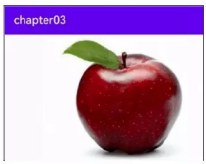


图 3-29 缩放类型为 fitEnd 的效果图

注意到centerInside和center的显示效果居然一模一样，这缘于它们的缩放规则设定。表面上fitCenter、centerInside、center这三个类型都是居中显示，且均不越过图像视图的边界。它们之间的区别在于：fitCenter既允许缩小图片、也允许放大图片，centerInside只允许缩小图片、不允许放大图片，而center自始至终保持原始尺寸（既不允许缩小图片、也不允许放大图片）。因此，当图片尺寸大于视图宽高，centerInside与fitCenter都会缩小图片，此时它俩的显示效果相同；当图片尺寸小于视图宽高，centerInside与center都保持图片大小不变，此时它俩的显示效果相同。

3.5.2 图像按钮 ImageButton

常见的按钮控件Button其实是文本按钮，因为按钮上面只能显示文字，不能显示图片，ImageButton才是显示图片的图像按钮。虽然ImageButton号称图像按钮，但它并非继承自Button，而是继承了ImageView。所以凡是ImageView拥有的属性和方法，ImageButton统统拿了过来，区别在于ImageButton有个按钮背景。

尽管ImageButton源自ImageView，但它毕竟是个按钮，按钮家族常用的点击事件和长按事件，ImageButton全都没落下。不过ImageButton和Button之间除了名称不同外，还有下列差异：

- Button**既可显示文本也可显示图片（通过**setBackgroundResource**方法设置背景图片），而**ImageButton**只能显示图片不能显示文本。**
- ImageButton**上的图像可按比例缩放，而**Button**通过背景设置的图像会拉伸变形，因为背景图采取**fitXY**方式，无法按比例缩放。**
- Button**只能靠背景显示一张图片，而**ImageButton**可分别在前景和背景显示图片，从而实现两张图片叠加的效果。**

从上面可以看出，Button与ImageButton各有千秋，通常情况使用Button就够用了。但在某些场

合，比如输入法打不出来的字符，以及特殊字体显示的字符串，就适合先切图再放到ImageButton。举个例子，数学常见的开方运算，由输入法打出来的运算符号为“√”，但该符号缺少右上角的一横，正确的开方符号是带横线的$\sqrt{\ }$，此时便需要通过ImageButton显示这个开方图片。

不过使用ImageButton得注意，图像按钮默认的缩放类型为center（保持原始尺寸不缩放图片），而非图像视图默认的fitCenter，倘若图片尺寸较大，那么图像按钮将无法显示整个图片。为避免显示不完整的情况，XML文件中的ImageButton标签必须指定fitCenter的缩放类型，详细的标签内容示例如下：

```
(完整代码见chapter03\src\main\res\layout\activity_image_button.xml)
<ImageButton
    android:layout_width="match_parent"
    android:layout_height="80dp"
    android:src="@drawable/sqrt"
    android:scaleType="fitCenter" />
```

运行测试App，打开的演示界面如图3-30所示，可见图像按钮正确展示了开方符号。

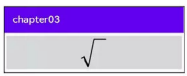


图 3-30 显示开方符号的图像按钮

3.5.3 同时展示文本与图像

现在有了Button可在按钮上显示文字，又有ImageButton可在按钮上显示图像，照理说绝大多数场合都够用了。然而现实项目中的需求往往捉摸不定，例如客户要求在按钮文字的左边加一个图标，这样按钮内部既有文字又有图片，乍看之下Button和ImageButton都没法直接使用。若用LinearLayout对ImageView和TextView组合布局，虽然可行，但XML文件却变得冗长许多。

其实有个既简单又灵活的办法，要想在文字周围放置图片，使用Button就能实现。Button提供了几个与图标有关的属性，通过这些属性即可指定文字旁边的图标。相关的图标属性说明如下：

- drawableTop：**指定文字上方的图片。**
- drawableBottom：**指定文字下方的图片。**
- drawableLeft：**指定文字左边的图片。**
- drawableRight：**指定文字右边的图片。**
- drawablePadding：**指定图片与文字的间距。**

譬如下面是个既有文字又有图标的Button标签例子：

```
(完整代码见chapter03\src\main\res\layout\activity_image_text.xml)
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:drawableTop="@drawable/ic_about"
    android:drawablePadding="5dp"
    android:text="图标在上" />
```

以上的Button标签通过属性android:drawableTop设置了文字上边的图标，若想变更图标所处的位置，只要把drawableTop换成对应方向的属性即可。各方向的图文混排按钮效果分别如图3-31～图3-34所示，其中图3-31为指定了drawableTop的按钮界面，图3-32为指定了drawableBottom的按钮界面，图3-33为指定了drawableLeft的按钮界面，图3-34为指定了drawableRight的按钮界面。



图 3-31 指定了 drawableTop 的按钮界面



图 3-32 指定了 drawableBottom 的按钮界面

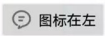


图 3-33 指定了 drawableLeft 的按钮界面

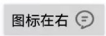


图 3-34 指定了 drawableRight 的按钮界面

3.6 实战项目：计算器

本章虽然只学了一些Android的简单控件，但是只要活学善用这些布局和控件，也能够做出实用的App。接下来让我们尝试设计并实现一个简单计算器。

3.6.1 需求描述

计算器是人们日常生活中最常用的工具之一，无论是在计算机上还是在手机上，都少不了计算器的身影。以Windows系统自带的计算器为例，它的界面简洁且十分实用，如图3-35所示。

计算器的界面分为两大部分：第一部分是上方的计算表达式，既包括用户的按键输入，也包括计算结果数字；第二部分是下方的各个按键，包括从0到9的数字按键、加减乘除与等号、正负号按键、小数点按键、求倒数按键、平方按键、开方按键，以及退格、清空、取消等控制按键。通过这些按键操作，能够实现整数和小数的四则运算，以及求倒数、求平方、求开方等简单运算。

3.6.2 界面设计

上一小节介绍的Windows计算器，它主要由上半部分的计算结果与下半部分的计算按键两块区域组成，据此可创建一个界面相似的计算器App，同样由计算结果和计算按键两部分组成，如图3-36所示。



图 3-35 Windwos 系统自带的计算器



图 3-36 计算器 App 的效果图

按照计算器App的效果图，大致分布着下列Android控件：

- **线性布局**LinearLayout：**因为计算器界面整体从上往下布局，所以需要垂直方向的**LinearLayout。
- **网格布局**GridLayout：**计算器下半部分的几排按键，正好成五行四列表格分布，适合采用**GridLayout。
- **滚动视图**ScrollView：**虽然计算器界面不宽也不高，但是以防万一，最好还是加个垂直方向的**ScrollView。
- **文本视图**TextView：**很明显顶部标题“简单计算器”就是**TextView，**且文字居中显示；标题下面的计算结果也需要使用**TextView，**且文字靠右靠下显示。**
- **按钮**Button：**几乎所有的数字与运算符按键都采用了**Button**控件。**
- **图像按钮**ImageButton：**开方的运算符“√”虽然能够打出来，但是右上角少了数学课本上的一横，所以该按钮要显示一张标准的开方符号图片，需要用到**ImageButton。

3.6.3 关键代码

App在同用户交互的过程中，时常要向用户反馈一些信息，例如：点错了按钮、输入了非法字符，诸如此类。对于这些一句话的提示，Android设计了Toast控件，用于展示短暂的提示文字。Toast的用法很简单，只需以下一行代码即可弹出提示窗口：

```
Toast.makeText(MainActivity.this, "提示文字", Toast.LENGTH_SHORT).show();
```

上面代码用到了两个方法，分别是makeText和show，其中show方法用来展示提示窗口，而makeText方法用来构建提示文字的模板。makeText的第一个参数为当前页面的实例，倘若当前页面名为MainActivity的话，这里就填MainActivity.this，当然如果不引发歧义的话，直接填this也可以；第二个参数为准备显示的提示文本；第三个参数规定了提示窗的驻留时长，为Toast.LENGTH_SHORT表示停留2秒后消失，为Toast.LENGTH_LONG表示停留3.5秒后消失。

对于计算器来说，有好几种情况需要提示用户，比如“除数不能为零”“开方的数值不能小于零”“不能对零求倒数”等，这时就能通过Toast控件弹窗提醒用户。Toast弹窗的展示效果如图3-37所示，此时App发现了除数为零的情况。



图 3-37 Toast 弹窗效果

对于简单计算来说，每次运算至少需要两个操作数，比如加减乘除四则运算就要求有两个操作数，求倒数、求平方、求开方只要求一个操作数；并且每次运算过程有且仅有一个运算符（等号不计在内），故而计算器App得事先声明下列几个字符串变量：

```
private String operator = "";        // 运算符
private String firstNum = "";        // 第一个操作数
private String secondNum = "";       // 第二个操作数
private String result = "";          // 当前的计算结果
```

用户在计算器界面每输入一个按键，App都要进行下列两项操作：

1. 输入按键的合法性校验

在开展计算之前，务必检查用户输入的按键是否合法，因为非法按键将导致不能正常运算。合法的按键输入包括但不限于下列情况：

（1）除数不能为零。
（2）开方的数值不能小于零。
（3）不能对零求倒数。
（4）一个数字不能有两个小数点。
（5）如果没输入运算符，就不能点击等号按钮。
（6）如果没输入操作数，也不能点击等号按钮。

比如点击等号按钮之时，App的逻辑校验代码示例如下：

```
(完整代码见chapter03\src\main\java\com\example\chapter03\CalculatorActivity.java)
if (v.getId() == R.id.btn_equal) {          // 点击了等号按钮
    if (operator.equals("")) {              // 无运算符
        Toast.makeText(this, "请输入运算符", Toast.LENGTH_SHORT).show();
        return false;
    }
    if (firstNum.equals("") || secondNum.equals("")) {  // 无操作数
        Toast.makeText(this, "请输入数字", Toast.LENGTH_SHORT).show();
        return false;
    }
    // 除数为零
    if (operator.equals("÷") && Double.parseDouble(secondNum) == 0) {
        Toast.makeText(this, "除数不能为零", Toast.LENGTH_SHORT).show();
        return false;
    }
}
```

2. 执行运算并显示计算结果

合法性校验通过，方能继续接下来的业务逻辑，倘若用户本次未输入与计算有关的按钮（例如等号、求倒数、求平方、求开方），则计算器只需拼接操作数或者运算符；倘若用户本次输入了与计算有关的按钮（例如等号、求倒数、求平方、求开方），则计算器立即执行运算操作并显示计算结果。以加减乘除四则运算为例，它们的计算代码例子如下：

```
// 加减乘除四则运算，返回计算结果
private double caculateFour() {
    double caculate_result = 0;
    if (operator.equals("＋")) {                    // 当前是相加运算
        caculate_result = Double.parseDouble(firstNum) + Double.parseDouble(secondNum);
    } else if (operator.equals("－")) {             // 当前是相减运算
        caculate_result = Double.parseDouble(firstNum) - Double.parseDouble(secondNum);
    } else if (operator.equals("×")) {              // 当前是相乘运算
        caculate_result = Double.parseDouble(firstNum) *
                Double.parseDouble(secondNum);
    } else if (operator.equals("÷")) {              // 当前是相除运算
        caculate_result = Double.parseDouble(firstNum) /
                Double.parseDouble(secondNum);
    }
    return caculate_result;
}
```

完成合法性校验与运算处理之后，计算器App的编码基本结束了。运行计算器App，执行各种运算的界面效果如图3-38和图3-39所示。其中图3-38为执行乘法运算8×9=?的计算器界面，图3-39为先对8做开方再给开方结果加上60的计算器界面。



图 3-38 执行乘法运算的计算器界面

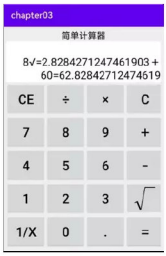


图 3-39 执行先开方再相加运算的计算器界面

3.7 小 结

本章主要介绍了App开发中常见的简单控件的用法，包括：在文本视图上显示文本（设置文本的内容、大小和颜色）、修改视图的基本属性（设置视图的宽和高、间距和对齐方式）、运用各种布局排列控件（线性布局、相对布局、网格布局、滚动视图）、处理按钮的触控事件（按钮控件的点击、长按、禁用与恢复）、在图像控件上显示图片（图像视图、图像按钮、同时展示文本与图像）。最后设计了一个实战项目“简单计算器”，在该项目的App编码中用到了前面介绍的大部分控件和布局，从而加深了对所学知识的理解。

通过本章的学习，读者应该能掌握以下4种开发技能：

（1）学会在文本控件上正确展示文字。
（2）学会在图像控件上正确展示图片。
（3）学会正确处理按钮的点击和长按事件。
（4）学会在常见布局上排列组合多个控件。

3.8 动手练习

请上机实验本章的计算器项目，要求实现加、减、乘、除、求倒数、求开方等简单运算。

第 4 章

常 用 组 件

本章介绍Android三个常用组件的基本概念和常见用法。主要包括活动组件——Activity、广播组件——Broadcast、服务组件——Service，以及如何在组件之间传递消息数据。

4.1 启停活动页面

本节介绍如何正确地启动和停止活动页面：首先描述活动页面的启动方法与结束方法，用户看到的页面就是开发者塑造的活动；然后详细分析活动的完整生命周期，以及每个周期方法的发生场景和流转过程；最后描述活动的几种启动模式，以及如何在代码中通过启动标志控制活动的跳转行为。

4.1.1 Activity 的启动和结束

在第2章的“2.4.3 跳转到另一个页面”一节中，提到通过startActivity方法可以从当前页面跳转到新页面，具体格式如“startActivity(new Intent(源页面.this, 目标页面.class));”。由于当时尚未介绍按钮控件，因此只好延迟3秒后才自动调用startActivity方法。现在有了按钮控件，就能利用按钮的点击事件去触发页面跳转，譬如以下代码便在重写后的点击方法onClick中执行页面跳转动作。

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActStartActivity.java)
// 页面类直接实现点击监听器的接口View.OnClickListener
public class ActStartActivity extends AppCompatActivity implements View.OnClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_act_start);
        // setOnClickListener来自View，故而允许直接给View对象注册点击监听器
        findViewById(R.id.btn_act_next).setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {  // 点击事件的处理方法
        if (v.getId() == R.id.btn_act_next) {
            // 从当前页面跳转到指定的新页面
            startActivity(new Intent(this, ActFinishActivity.class));
        }
    }
}
```

以上代码中的startActivity方法，清楚标明了从当前页面跳转到新的ActFinishActivity页面。之所以给新页面取名ActFinishActivity，是为了在新页面中演示如何关闭页面。众所周知，若要从当前页面回到上一个页面，点击屏幕底部的返回键即可实现，但不是所有场景都使用返回键。比如页面左上角的箭头图标经常代表着返回动作，况且有时页面上会出现“完成”按钮，无论点击箭头图标还是点击完成按钮，都要求马上回到上一个页面。包含箭头图标与“完成”按钮的演示界面如图4-1所示。



图 4-1 箭头图标与完成按钮

既然点击某个图标或者点击某个按钮均可能触发返回动作，就需要App支持在某个事件发生时主动返回上一页。回到上一个页面其实相当于关闭当前页面，因为最开始是由A页面跳转到B页面，一旦关闭了B页面，App应该展示哪个页面呢？当然是展示跳转之前的A页面了。在Java代码中，调用finish方法即可关闭当前页面，前述场景要求点击箭头图标或“完成”按钮都返回上一页面，则需给箭头图标和“完成”按钮分别注册点击监听器，然后在onClick方法中调用finish方法。下面便是添加了finish方法的新页面代码例子：

(完整代码见chapter04\src\main\java\com\example\chapter04\ActFinishActivity.java)
```
    // 活动类直接实现点击监听器的接口View.OnClickListener
    public class ActFinishActivity extends AppCompatActivity implements
View.OnClickListener {

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_act_finish);
            // 给箭头图标注册点击监听器，ImageView由View类派生而来
            findViewById(R.id.iv_back).setOnClickListener(this);
            // 给“完成”按钮注册点击监听器，Button也由View类派生而来
            findViewById(R.id.btn_finish).setOnClickListener(this);
        }

        @Override
        public void onClick(View v) {          // 点击事件的处理方法
            if (v.getId() == R.id.iv_back || v.getId() == R.id.btn_finish) {
                finish();                      // 结束当前的活动页面
            }
        }
    }
```

另外，所谓“打开页面”或“关闭页面”沿用了浏览网页的叫法，对于App而言，页面的真实名称是“活动”——Activity。打开某个页面其实是启动某个活动，所以有startActivity方法却无openActivity方法；关闭某个页面其实是结束某个活动，所以有finish方法却无close方法。

4.1.2 Activity 的生命周期

App引入活动的概念而非传统的页面概念，这是有原因的，单从字面意思理解，页面更像是静态的，而活动更像是动态的。犹如花开花落那般，活动也有从含苞待放到盛开再到凋零的生命过程。每次创建新的活动页面，自动生成的Java代码都给出了onCreate方法，该方法用于执行活动创建的相关操作，包括加载XML布局、设置文本视图的初始文字、注册按钮控件的点击监听，等等。onCreate方法所代表的创建动作，正是一个活动最开始的行为。除了onCreate，活动还有其他几种生命周期行为，它们对应的方法说明如下：

- onCreate：创建活动。此时会把页面布局加载进内存，进入了初始状态。
- onStart：开启活动。此时会把活动页面显示在屏幕上，进入了就绪状态。
- onResume：恢复活动。此时活动页面进入活跃状态，能够与用户正常交互，例如允许响应用户的点击动作、允许用户输入文字等。
- onPause：暂停活动。此时活动页面进入暂停状态（也就是退回就绪状态），无法与用户正常交互。
- onStop：停止活动。此时活动页面将不在屏幕上显示。
- onDestroy：销毁活动。此时回收活动占用的系统资源，把页面从内存中清除掉。
- onRestart：重启活动。处于停止状态的活动，若想重新开启的话，无须经历onCreate的重复创建过程，而是走onRestart的重启过程。
- onNewIntent：重用已有的活动实例。

上述的生命周期方法，涉及复杂的App运行状态，更直观的活动状态切换过程如图4-2所示。



图 4-2　活动的状态变迁

由图4-2可知，打开新活动页面的方法调用顺序为onCreate→onStart→onResume，关闭旧活动页面的方法调用顺序为onPause→onStop→onDestroy。为了弄清楚这些方法的调用时机。接下来通过一个实验加以观察。

首先分别创建两个活动页面，它们的Java代码都重写了下列7个生命周期方法：onCreate、onStart、onResume、onPause、onStop、onDestroy、onRestart，每个方法内部均调用新写的refreshLife方法打印日志。其中第一个活动页面的Java代码示例如下，第二个活动页面的Java代码可如法炮制：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActLifeActivity.java)
public class ActLifeActivity extends AppCompatActivity implements View.OnClickListener {
    private final static String TAG = "ActLifeActivity";
    private TextView tv_life;                    // 声明一个文本视图对象
    private String mStr = "";

    private void refreshLife(String desc) {      // 刷新生命周期的日志信息
        mStr = String.format("%s%s %s %s\n", mStr, DateUtil.getNowTimeDetail(), TAG,
desc);
        tv_life.setText(mStr);
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {  // 创建活动页面
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_act_life);
        findViewById(R.id.btn_act_next).setOnClickListener(this);
        tv_life = findViewById(R.id.tv_life);  // 获取名为tv_life的文本视图
        refreshLife("onCreate");     // 刷新生命周期的日志信息
    }

    @Override
    protected void onStart() {       // 开始活动
        super.onStart();
        refreshLife("onStart");      // 刷新生命周期的日志信息
    }
```

```
    @Override
    protected void onStop() {          // 停止活动
        super.onStop();
        refreshLife("onStop");         // 刷新生命周期的日志信息
    }

    @Override
    protected void onResume() {        // 恢复活动
        super.onResume();
        refreshLife("onResume");       // 刷新生命周期的日志信息
    }

    @Override
    protected void onPause() {         // 暂停活动
        super.onPause();
        refreshLife("onPause");        // 刷新生命周期的日志信息
    }

    @Override
    protected void onRestart() {       // 重启活动
        super.onRestart();
        refreshLife("onRestart");      // 刷新生命周期的日志信息
    }

    @Override
    protected void onDestroy() {       // 销毁活动
        super.onDestroy();
        refreshLife("onDestroy");      // 刷新生命周期的日志信息
    }

    @Override
    public void onClick(View v) {
        if (v.getId() == R.id.btn_act_next) {
            // 从当前页面跳转到指定的活动页面
            startActivity(new Intent(this, ActNextActivity.class));
        }
    }
}
```

运行测试App，依次打开两个活动页面，分别如图4-3～图4-5所示，其中图4-3为刚进入第一个活动的界面，图4-4为从第一个活动跳转到第二个活动的界面，图4-5为从第二个活动返回第一个活动的界面。

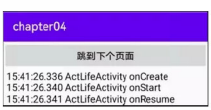


图 4-3 进入第一个活动

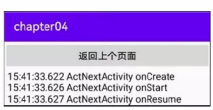


图 4-4 跳转到第二个活动



图 4-5 从第二个活动回到第一个活动

根据以上3幅图示的日志时间，梳理出完整的生命周期时间线如图4-6所示，从而验证了之前所说的生命周期过程。

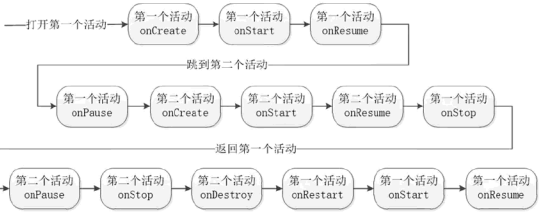


图 4-6　活动跳转时的生命周期

4.1.3　Activity 的启动模式

上一小节提到，从第一个活动跳转到第二个活动，接着结束第二个活动就能返回第一个活动，可是为什么不直接返回桌面呢？这要从Android的内核设计说起了。系统给每个正在运行的App都分配了活动栈，栈里面容纳着已经创建且尚未销毁的活动信息。鉴于栈是一种先进后出、后进先出的数据结构，故而后面入栈的活动总是先出栈，假设3个活动的入栈顺序为活动A→活动B→活动C，则它们的出栈顺序将变为活动C→活动B→活动A，可见活动C结束之后会返回活动B，而不是返回活动A或者别的地方。

假定某个App分配到的活动栈大小为3，该App先后打开两个活动，此时活动栈的变动情况如图4-7所示。

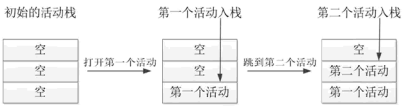


图 4-7　两个活动先后入栈

然后按下返回键，依次结束已打开的两个活动，此时活动栈的变动情况如图4-8所示。

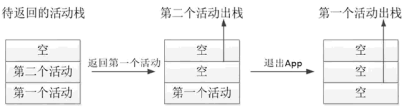


图 4-8　两个活动依次出栈

结合图4-7与图4-8的入栈与出栈流程，即可验证结束活动之时的返回逻辑了。

不过前述的出入栈情况仅是默认的标准模式，实际上Android允许在创建活动时指定该活动的启动模式，通过启动模式控制活动的出入栈行为。App提供了两种办法用于设置活动页面的启动模式：其一是修改AndroidManifest.xml，在指定的activity节点添加属性android:launchMode，表示本活动以哪个启动模式运行；其二是在代码中调用Intent对象的setFlags方法，表明后续打开的活动页面采用该启动标志。下面分别予以详细说明。

1. 在配置文件中指定启动模式

打开AndroidManifest.xml，给activity节点添加属性android:launchMode，属性值填入standard表示采取标准模式，当然不添加属性的话默认就是标准模式。具体的activity节点配置内容示例如下：

```
<activity android:name=".JumpFirstActivity" android:launchMode="standard" />
```

其中launchMode属性的几种取值说明见表4-1。

表 4-1　launchMode 属性的取值说明

launchMode 属性值	说　明
standard	标准模式，无论何时启动哪个活动，都是重新创建该页面的实例并放入栈顶。如果不指定 launchMode 属性，则默认为标准模式
singleTop	启动新活动时，判断如果栈顶正好就是该活动的实例，则重用该实例；否则创建新的实例并放入栈顶，也就是按照 standard 模式处理
singleTask	启动新活动时，判断如果栈中存在该活动的实例，则重用该实例，并清除位于该实例上面的所有实例；否则按照 standard 模式处理
singleInstance	启动新活动时，将该活动的实例放入一个新栈中，原栈的实例列表保持不变

2. 在代码里面设置启动标志

打开Java代码，先调用Intent对象的setFlags方法设置启动标志，再将该Intent对象传给startActivity方法。具体的方法调用代码示例如下：

```
// 创建一个意图对象，准备跳转到指定的活动页面
Intent intent = new Intent(this, JumpSecondActivity.class);
// 设置启动标志。Intent.FLAG_ACTIVITY_NEW_TASK表示创建新的任务栈
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(intent);  // 跳转到意图对象指定的活动页面
```

之所以要在代码中动态指定活动页面的启动模式，是因为AndroidManifest.xml对每个活动只能指定唯一的启动模式。若想在不同时候对同一个活动运用不同的启动模式，显然固定的launchMode属性无法满足这个要求。于是Android允许在代码中手动设置启动标志，这样在不同时候调用startActivity方法就能运用不一样的启动模式。

适用于setFlags方法的几种启动标志取值说明见表4-2。

表 4-2　代码中的启动标志取值说明

Intent 类的启动标志	说　明
Intent.FLAG_ACTIVITY_NEW_TASK	开辟一个新的任务栈，该值类似于 launchMode="standard"；不同之处在于，如果原来不存在活动栈，则 FLAG_ACTIVITY_NEW_TASK 会创建一个新栈
Intent.FLAG_ACTIVITY_SINGLE_TOP	当栈顶为待跳转的活动实例之时，则重用栈顶的实例。该值等同于 launchMode="singleTop"
Intent.FLAG_ACTIVITY_CLEAR_TOP	当栈中存在待跳转的活动实例时，则重新创建一个新实例，并清除原实例上方的所有实例。该值与 launchMode="singleTask" 类似，但 singleTask 采取 onNewIntent 方法启用原任务，而 FLAG_ACTIVITY_CLEAR_TOP 采取先调用 onDestroy 再调用 onCreate 来创建新任务

（续表）

Intent 类的启动标志	说　明
Intent.FLAG_ACTIVITY_NO_HISTORY	该标志与 launchMode="standard"情况类似，但栈中不保存新启动的活动实例。这样下次无论以何种方式再启动该实例，也要走 standard 模式的完整流程
Intent.FLAG_ACTIVITY_CLEAR_TASK	该标志非常暴力，跳转到新页面时，栈中的原有实例都被清空。注意该标志需要结合 FLAG_ACTIVITY_NEW_TASK 使用，即 setFlags 方法的参数为“Intent.FLAG_ACTIVITY_CLEAR_TASK | Intent.FLAG_ACTIVITY_NEW_TASK”

接下来举两个例子阐述启动模式的实际应用：在两个活动之间交替跳转，登录成功后不再返回登录页面，分别介绍如下。

1. 在两个活动之间交替跳转

假设活动A有个按钮，点击该按钮会跳转到活动B；同时活动B也有个按钮，点击按钮会跳转到活动A；从首页打开活动A之后，就点击按钮在活动A与活动B之间轮流跳转。此时活动页面的跳转流程为：首页→活动A→活动B→活动A→活动B→活动A→活动B→……多次跳转之后想回到首页，正常的话返回流程是这样的：……→活动B→活动A→活动B→活动A→活动B→活动A→首页。注意每个箭头都代表按一次返回键，可见要按下许多次返回键才能返回首页。其实在活动A和活动B之间本不应该重复返回，因为回来回去总是这两个页面有什么意义呢？照理说每个活动返回一次足矣，同一个地方返回两次已经是多余的了，再返回应当回到首页才是。也就是说，不管过去的时候怎么跳转，回来的时候应该按照……→活动B→活动A→首页这个流程，或者按照……→活动A→活动B→首页这个流程，总之已经返回了的页面，决不再返回第二次。

对于不允许重复返回的情况，可以设置启动标志FLAG_ACTIVITY_CLEAR_TOP，即使活动栈里面存在待跳转的活动实例，也会重新创建该活动的实例，并清除原实例上方的所有实例，保证栈中最多只有该活动的唯一实例，从而避免了无谓的重复返回。于是活动A内部的跳转代码就改成了下面这般：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\JumpFirstActivity.java)
// 创建一个意图对象，准备跳转到指定的活动页面
Intent intent = new Intent(this, JumpSecondActivity.class);
// 当栈中存在待跳转的活动实例时，则重新创建该活动的实例，并清除原实例上方的所有实例
intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);  // 设置启动标志
startActivity(intent);  // 跳转到意图对象指定的活动页面
```

当然活动B内部的跳转代码也要设置同样的启动标志：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\JumpSecondActivity.java)
// 创建一个意图对象，准备跳转到指定的活动页面
Intent intent = new Intent(this, JumpFirstActivity.class);
// 当栈中存在待跳转的活动实例时，则重新创建该活动的实例，并清除原实例上方的所有实例
intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);  // 设置启动标志
startActivity(intent);  // 跳转到意图指定的活动页面
```

这下两个活动的跳转代码都设置了FLAG_ACTIVITY_CLEAR_TOP，运行测试App发现多次跳转之后，每个活动仅会返回一次。

2. 登录成功后不再返回登录页面

很多App第一次打开都要求用户登录，登录成功再进入App首页，如果这时按下返回键，发现并没有回到上一个登录页面，而是直接退出App了，这又是什么缘故呢？原来用户登录成功后，App便记下用户的登录信息，接下来默认该用户是登录状态，自然不必重新输入用户名和密码了。既然默认用户已经登录，哪里还需要回到登录页面？不光登录页面，登录之前的其他页面包括获取验证码、找回密码等页面都不应回去，每次登录成功之后，整个App就焕然一新，仿佛忘记了有登录页面这回事。

对于回不去的登录页面情况，可以设置启动标志FLAG_ACTIVITY_CLEAR_TASK，该标志会清空当前活动栈里的所有实例。不过全部清空之后，就意味着当前栈没法用了，必须另外找个活动栈才行，也就是同时设置启动标志FLAG_ACTIVITY_NEW_TASK，该标志用于开辟新任务的活动栈。于是离开登录页面的跳转代码变成下面这样：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\LoginInputActivity.java)
// 创建一个意图对象，准备跳转到指定的活动页面
Intent intent = new Intent(this, LoginSuccessActivity.class);
// 设置启动标志：跳转到新页面时，栈中的原有实例都被清空，同时开辟新任务的活动栈
intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK |
                Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(intent);  // 跳转到意图指定的活动页面
```

运行测试App，登录成功进入首页之后，点击返回键果然没回到登录页面。

4.2 在活动之间传递消息

本节介绍如何在两个活动之间传递各类消息：首先描述Intent的用途和组成部分，以及显式Intent和隐式Intent的区别；然后阐述结合Intent和Bundle向下一个活动页面发送数据，再由下一个活动页面返回应答数据给上一个页面；最后叙述如何使用新的registerForActivityResult方法简化活动交互过程。

4.2.1 显式 Intent 和隐式 Intent

上一小节的Java代码通过Intent对象设置活动的启动标志，这个Intent究竟是什么呢？Intent的中文名是意图，意思是我想让你干什么，简单地说，就是传递消息。Intent是各个组件之间信息沟通的桥梁，既能在Activity之间沟通，又能在Activity与Service之间沟通，也能在Activity与Broadcast之间沟通。总而言之，Intent用于Android各组件之间的通信，它主要完成下列3部分工作：

（1）标明本次通信请求从哪里来、到哪里去、要怎么走。

（2）发起方携带本次通信需要的数据内容，接收方从收到的意图中解析数据。

（3）发起方若想判断接收方的处理结果，意图就要负责让接收方传回应答的数据内容。

为了做好以上工作，就要给意图配上必需的装备，Intent的组成部分见表4-3。

表 4-3　Intent 组成元素的列表说明

元素名称	设置方法	说明与用途
Component	setComponent	组件，它指定意图的来源与目标
Action	setAction	动作，它指定意图的动作行为
Data	setData	即 Uri，它指定动作要操纵的数据路径
Category	addCategory	类别，它指定意图的操作类别
Type	setType	数据类型，它指定消息的数据类型
Extras	putExtras	扩展信息，它指定装载的包裹信息
Flags	setFlags	标志位，它指定活动的启动标志

指定意图对象的目标有两种表达方式，一种是显式Intent，另一种是隐式Intent。

1. 显式Intent，直接指定来源活动与目标活动，属于精确匹配

在构建一个意图对象时，需要指定两个参数，第一个参数表示跳转的来源页面，即“来源Activity.this”；第二个参数表示待跳转的页面，即“目标Activity.class”。具体的意图构建方式有如下3种：

（1）在Intent的构造函数中指定，示例代码如下：

```
// 创建一个目标确定的意图
Intent intent = new Intent(this, ActNextActivity.class);
```

（2）调用意图对象的setClass方法指定，示例代码如下：

```
Intent intent = new Intent();                    // 创建一个新意图
intent.setClass(this, ActNextActivity.class);    // 设置意图要跳转的目标活动
```

（3）调用意图对象的setComponent方法指定，示例代码如下：

```
Intent intent = new Intent();                    // 创建一个新意图
// 创建包含目标活动在内的组件名称对象
ComponentName component = new ComponentName(this, ActNextActivity.class);
intent.setComponent(component);                  // 设置意图携带的组件信息
```

2. 隐式Intent，没有明确指定要跳转的目标活动，只给出一个动作字符串让系统自动匹配，属于模糊匹配

通常App不希望向外部暴露活动名称，只给出一个事先定义好的标记串，这样大家约定俗成、按图索骥就好，隐式Intent便起到了标记过滤作用。这个动作名称标记串，可以是自己定义的动作，也可以是已有的系统动作。常见系统动作的取值说明见表4-4。

表 4-4　常见系统动作的取值说明

Intent 类的系统动作常量名	系统动作的常量值	说　　明
ACTION_MAIN	android.intent.action.MAIN	App 启动时的入口
ACTION_VIEW	android.intent.action.VIEW	向用户显示数据
ACTION_SEND	android.intent.action.SEND	分享内容
ACTION_CALL	android.intent.action.CALL	直接拨号
ACITON_DIAL	android.intent.action.DIAL	准备拨号
ACTION_SENDTO	android.intent.action.SENDTO	发送短信
ACTION_ANSWER	android.intent.action.ANSWER	接听电话

动作名称既可以通过setAction方法指定，也可以通过构造函数Intent(String action)直接生成意图对象。当然，由于动作是模糊匹配，因此有时需要更详细的路径，比如仅知道某人住在天通苑小区，并不能直接找到他家，还得说明他住在天通苑的哪一期、哪栋楼、哪一层、哪一个单元。Uri和Category便是这样的路径与门类信息，Uri数据可以通过构造函数Intent(String action, Uri uri)在生成对象时一起指定，也可以通过setData方法指定（setData这个名字有歧义，实际相当于setUri）；Category可以通过addCategory方法指定，之所以用add而不用set方法，是因为一个意图允许设置多个Category，方便一起过滤。

下面是一个调用系统拨号程序的代码例子，其中就用到了Uri：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActionUriActivity.java)
String phoneNo = "12345";
Intent intent = new Intent();                   // 创建一个新意图
intent.setAction(Intent.ACTION_DIAL);           // 设置意图动作为准备拨号
Uri uri = Uri.parse("tel:" + phoneNo);          // 声明一个拨号的Uri
intent.setData(uri);                            // 设置意图前往的路径
startActivity(intent);                          // 启动意图通往的活动页面
```

隐式Intent还用到了过滤器的概念，把不符合匹配条件的过滤掉，剩下符合条件的按照优先顺序调用。譬如创建一个App模块，AndroidManifest.xml里的intent-filter就是配置文件中的过滤器。像最常见的首页活动MainAcitivity，它的activity节点下面便设置了action和category的过滤条件。其中android.intent.action.MAIN表示App的入口动作，而android.intent.category.LAUNCHER表示在桌面上显示App图标，配置样例如下：

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

4.2.2 普通的活动数据交互

上一小节提到，Intent对象的setData方法只指定到达目标的路径，并非本次通信所携带的参数信息，真正的参数信息存放在Extras中。Intent重载了很多种putExtra方法传递各种类型的参数，包括整型、双精度型、字符串等基本数据类型，甚至Serializable这样的序列化结构。只是调用putExtra方法显然不好管理，像送快递一样大小包裹随便扔，不但找起来不方便，丢了也难以知道。所以Android引入了Bundle概念，可以把Bundle理解为超市的寄包柜或快递收件柜，大小包裹由Bundle统一存取，方便又安全。

Bundle内部用于存放消息的数据结构是Map映射，既可添加或删除元素，还可判断元素是否存在。开发者若要把Bundle数据全部打包好，只需调用一次意图对象的putExtras方法；若要把Bundle数据全部取出来，也只需调用一次意图对象的getExtras方法。Bundle对象操作各类型数据的读写方法说明见表4-5。

表 4-5 Bundle 对各类型数据的读写方法说明

数据类型	读 方 法	写 方 法
整型数	getInt	putInt
浮点数	getFloat	putFloat
双精度数	getDouble	putDouble
布尔值	getBoolean	putBoolean
字符串	getString	putString
字符串数组	getStringArray	putStringArray
字符串列表	getStringArrayList	putStringArrayList
可序列化结构	getSerializable	putSerializable

接下来举个在活动之间传递数据的例子，首先在上一个活动使用包裹封装好数据，把包裹交给意图对象，再调用startActivity方法跳转到意图指定的目标活动。完整的活动跳转代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActSendActivity.java)
// 创建一个意图对象，准备跳转到指定的活动页面
Intent intent = new Intent(this, ActReceiveActivity.class);
Bundle bundle = new Bundle();              // 创建一个新包裹
// 往包裹存入名为request time的字符串
bundle.putString("request time", DateUtil.getNowTime());
// 往包裹存入名为request_content的字符串
bundle.putString("request content", tv_send.getText().toString());
intent.putExtras(bundle);                  // 把包裹交给意图
startActivity(intent);                     // 跳转到意图指定的活动页面
```

然后在下一个活动中获取意图携带的包裹，从包裹取出各参数信息，并将传来的数据显示到文本视图。下面便是目标活动获取并展示包裹数据的代码例子：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActReceiveActivity.java)
// 从布局文件中获取名为tv_receive的文本视图
TextView tv_receive = findViewById(R.id.tv_receive);
// 从上一个活动页面传来的意图中获取包裹
Bundle bundle = getIntent().getExtras();
// 从包裹中取出名为request time的字符串
String request_time = bundle.getString("request time");
// 从包裹中取出名为request content的字符串
String request_content = bundle.getString("request_content");
String desc = String.format("收到请求消息：\n请求时间为%s\n请求内容为%s",
        request_time, request_content);
tv_receive.setText(desc);  // 把请求消息的详情显示在文本视图上
```

代码编写完毕，运行测试App，打开上一个活动页面，如图4-9所示。单击页面上的“发送以上文字”按钮跳转到下一个活动页面，如图4-10所示，根据展示文本可知正确获得了传来的数据。



图 4-9 上一个活动将要发送数据

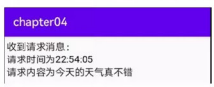


图 4-10 下一个活动收到传来的数据

数据传递经常是相互的，上一个活动不但把请求数据发送到下一个活动，有时候还要处理下一个活动的应答数据，所谓应答发生在下一个活动返回到上一个活动之际。如果只把请求数据发送

到下一个活动，上一个活动调用startActivity方法即可；如果还要处理下一个活动的应答数据，此时就得分多步处理，详细步骤说明如下：

01 上一个活动打包好请求数据，调用startActivityForResult方法执行跳转动作，表示需要处理下一个活动的应答数据，该方法的第二个参数表示请求代码，它用于标识每个跳转的唯一性。跳转代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActRequestActivity.java)
String request = "你吃饭了吗？来我家吃吧";
// 创建一个意图对象，准备跳转到指定的活动页面
Intent intent = new Intent(this, ActResponseActivity.class);
Bundle bundle = new Bundle();           // 创建一个新包裹
// 往包裹存入名为request time的字符串
bundle.putString("request_time", DateUtil.getNowTime());
// 往包裹存入名为request_content的字符串
bundle.putString("request content", request);
intent.putExtras(bundle);               // 把包裹交给意图
// 期望接收下一个活动的返回数据。第二个参数为本次请求代码
startActivityForResult(intent, 0);
```

02 下一个活动接收并解析请求数据，进行相应处理。接收代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActResponseActivity.java)
// 从上一个活动传来的意图中获取包裹
Bundle bundle = getIntent().getExtras();
// 从包裹中取出名为request time的字符串
String request time = bundle.getString("request time");
// 从包裹中取出名为request content的字符串
String request_content = bundle.getString("request_content");
String desc = String.format("收到请求消息：\n请求时间为%s\n请求内容为%s",
       request time, request content);
tv_request.setText(desc);  // 把请求消息的详情显示在文本视图上
```

03 下一个活动在返回上一个活动时，打包应答数据并调用setResult方法返回数据包裹。setResult方法的第一个参数表示应答代码（成功还是失败），第二个参数为携带包裹的意图对象。返回代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActResponseActivity.java)
String response = "我吃过了，还是你来我家吃";
Intent intent = new Intent();  // 创建一个新意图
Bundle bundle = new Bundle();  // 创建一个新包裹
// 往包裹存入名为response time的字符串
bundle.putString("response_time", DateUtil.getNowTime());
// 往包裹存入名为response_content的字符串
bundle.putString("response content", response);
intent.putExtras(bundle);  // 把包裹交给意图
// 携带意图返回上一个活动。RESULT OK表示处理成功
setResult(Activity.RESULT_OK, intent);
finish();  // 结束当前的活动页面
```

04 上一个活动重写方法onActivityResult，该方法的参数包含请求代码和结果代码，其中请求代码用于判断这次返回对应哪个跳转，结果代码用于判断下一个活动是否处理成功。如果下一个活动处理成功，再对返回数据进行解包操作，处理返回数据的代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ActRequestActivity.java)
// 从下一个活动携带参数返回当前活动时触发
@Override
```

```
    protected void onActivityResult(int requestCode, int resultCode, Intent intent) {  //
接收返回数据
        super.onActivityResult(requestCode, resultCode, intent);
        // 意图非空，且请求代码为之前传的0，结果代码也为成功
        if (intent!=null && requestCode==0 && resultCode== Activity.RESULT_OK) {
            Bundle bundle = intent.getExtras();  // 从返回的意图中获取包裹
            // 从包裹中取出名为response time的字符串
            String response time = bundle.getString("response time");
            // 从包裹中取出名为response content的字符串
            String response_content = bundle.getString("response_content");
            String desc = String.format("收到返回消息：\n应答时间为%s\n应答内容为%s",
                  response time, response content);
            tv response.setText(desc);  // 把返回消息的详情显示在文本视图上
        }
    }
```

结合上述的活动消息交互步骤，运行测试App，打开的第一个活动页面如图4-11所示。

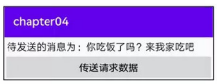


图 4-11　跳转之前的第一个活动页面

点击“传送请求数据”按钮跳转到第二个活动页面，如图4-12所示，可见第二个活动页面收到了请求数据。然后点击第二个活动页面的“返回应答数据”按钮，回到第一个活动页面，如图4-13所示，可见第一个活动页面成功收到了第二个活动页面的应答数据。

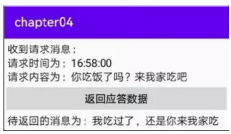


图 4-12　跳转到第二个活动页面

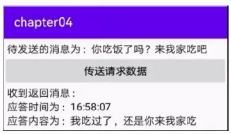


图 4-13　返回到第一个活动页面

4.2.3　改进后的活动数据交互

在两个活动之间传递消息，正常使用startActivityForResult搭配onActivityResult就好了，可是从appcompat1.3.0 开始，startActivityForResult 方法被标记为已废弃，官方建议改用registerForActivityResult方法。具体的使用步骤说明如下：

01 先声明一个活动结果启动器对象ActivityResultLauncher，举例如下：

```
    private ActivityResultLauncher mLauncher;  // 声明一个活动结果启动器对象
```

02 调用registerForActivityResult方法注册一个善后工作的活动结果启动器，并指定对活动返回数据的处理过程，也就是第一个参数传入ActivityResultContracts.StartActivityForResult对象，第二个参数填入onActivityResult要做的事情，示例如下：

```
    (完整代码见chapter04\src\main\java\com\example\chapter04\RegisterResultActivity.java)
    // 注册一个善后工作的活动结果启动器
    mLauncher = registerForActivityResult(new
ActivityResultContracts.StartActivityForResult(), result -> {
        if (result.getResultCode()==RESULT_OK && result.getData()!=null) {
            Bundle bundle = result.getData().getExtras();          // 从返回意图中获取包裹
            // 从包裹中取出名叫response_time的字符串
```

```
            String response_time = bundle.getString("response_time");
            // 从包裹中取出名叫response content的字符串
            String response_content = bundle.getString("response_content");
            String desc = String.format("收到返回消息：\n应答时间为：%s\n应答内容为：%s",
                    response_time, response_content);
            tv_response.setText(desc);        // 把返回消息的详情显示在文本视图上
        }
    });
```

08 调用启动器对象的launch方法，传入封装了参数信息的意图对象，开始执行启动器的跳转与回调处理。代码如下：

```
// 创建一个意图对象，准备跳转到指定的活动页面
Intent intent = new Intent(this, ActResponseActivity.class);
Bundle bundle = new Bundle();  // 创建一个新包裹
// 往包裹存入名叫request time的字符串
bundle.putString("request_time", DateUtil.getNowTime());
// 往包裹存入名叫request_content的字符串
bundle.putString("request content", mRrequest);
intent.putExtras(bundle);  // 把包裹交给意图
mLauncher.launch(intent);  // 活动结果启动器开动了
```

以上的活动结果启动器使用步骤，看起来并未简化多少代码，不见得带来什么方便。不过在某些特殊场合，运用活动结果启动器倒是能收到奇效。比如到系统相册挑选某张图片，调用startActivityForResult方法的话，活动跳转代码是下面这样的：

```
findViewById(R.id.btn_choose_common).setOnClickListener(v -> {
    // 创建一个内容获取动作的意图（准备跳转到系统相册）
    Intent intent = new Intent(Intent.ACTION_PICK,
            MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
    intent.setDataAndType(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, "image/*");
    startActivityForResult(intent, CHOOSE_CODE);
});
```

然后重写onActivityResult方法，依次判断resultCode和requestCode，校验通过后再展示图片，回调代码示例如下：

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    super.onActivityResult(requestCode, resultCode, intent);
    if (resultCode==RESULT_OK && requestCode==CHOOSE_CODE) {  // 从相册回来
        if (intent.getData() != null) {
            Uri uri = intent.getData();             // 获得已选择照片的路径对象
            // 根据指定图片的uri，获得自动缩小后的位图对象
            Bitmap bitmap = BitmapUtil.getAutoZoomImage(this, uri);
            iv_photo.setImageBitmap(bitmap);       // 设置图像视图的位图对象
        }
    }
}
```

此时利用活动结果启动器加以改造，可在调用registerForActivityResult方法时，第一个参数传入ActivityResultContracts.GetContent对象，第二个参数填入图片展示过程。之后调用启动器的launch方法传入文件类型“image/*”，即可完成从相册挑选并展示图片的功能。详细代码示例如下：

```
    (完整代码见chapter04\src\main\java\com\example\chapter04\ChoosePhotoActivity.java)
    // 注册一个善后工作的活动结果启动器，获取指定类型的内容
    ActivityResultLauncher launcher = registerForActivityResult(new
ActivityResultContracts.GetContent(), uri -> {
        if (uri != null) {
```

```
            // 根据指定图片的uri，获得自动缩小后的位图对象
            Bitmap bitmap = BitmapUtil.getAutoZoomImage(this, uri);
            iv_photo.setImageBitmap(bitmap);  // 设置图像视图的位图对象
        }
    });
    // 点击按钮时触发活动结果启动器，传入待获取内容的文件类型
    findViewById(R.id.btn_choose_register).setOnClickListener(
                v -> launcher.launch("image/*"));
```

从系统相册挑选图片的过程如图4-14和图4-15所示，其中图4-14为打开系统相册选择图片的界面，图4-15为从相册返回展示图片的界面。

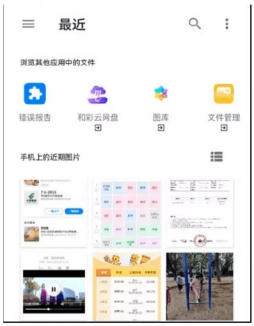


图 4-14　打开系统相册选择图片的界面



图 4-15　从相册返回展示图片的界面

4.3　收发应用广播

本节介绍应用广播的几种收发形式，包括如何收发标准广播、如何收发有序广播、如何收发静态广播、如何监听定时管理器发出的系统闹钟广播等。

4.3.1　收发标准广播

App在运行的时候有各种各样的数据流转，有的数据从上一个活动流向下一个活动，此时可通过意图在活动之间传递包裹；有的数据从应用内存流向存储卡，此时可进行文件读写操作。

随着App工程的代码量日益增长，承载数据流通的管道会越发不够用，好比在装修房子的时候，给每个房间都预留了网线插口，只有插上网线才能上网。可是现在联网设备越来越多，除了计算机之外，电视也要联网，平板也要联网，乃至空调都要联网，如此一来网口就不够用了。那怎样解决众多设备的联网问题呢？原来家家户户都配了无线路由器，路由器向四周发射WiFi信号，各设备只要安装了无线网卡，就能接收WiFi信号从而连接上网。于是“发射器+接收器”的模式另辟蹊径，比起网线这种固定管道要灵活得多，无须拉线即可随时随地传输数据。

Android的广播机制正是借鉴了WiFi的通信原理，不必搭建专门的通路，就能在发送方与接收方之间建立连接。同时广播（Broadcast）也是Android的四大组件之一，它用于Android各组件之间的灵活通信，与活动的区别在于：

（1）活动只能一对一通信，而广播可以一对多，一人发送广播，多人接收处理。

（2）对于发送方来说，广播不需要考虑接收方有没有在工作，接收方在工作就接收广播，不在工作就丢弃广播。

（3）对于接收方来说，因为可能会收到各式各样的广播，所以接收方要自行过滤符合条件的广播，之后再解包处理。

与广播有关的方法主要有以下3个。

- sendBroadcast：**发送广播**。
- registerReceiver：**注册广播的接收器，可在**onStart**或**onResume**方法中注册接收器。**
- unregisterReceiver：**注销广播的接收器，可在**onStop**或**onPause**方法中注销接收器。**

具体到编码实现上，广播的收发过程可分为3个步骤：发送标准广播、定义广播接收器、开关广播接收器，分别说明如下。

1. 发送标准广播

广播的发送操作很简单，一共只有两步：先创建意图对象，再调用sendBroadcast方法发送广播即可。不过要注意，意图对象需要指定广播的动作名称，如同每个路由器都得给自己的WiFi起个名称一般，这样接收方才能根据动作名称来判断来的是李逵还是李鬼。下面是通过点击按钮发送广播的活动页面代码：

```
    (完整代码见chapter04\src\main\java\com\example\chapter04\BroadStandardActivity.java)
    public class BroadStandardActivity extends AppCompatActivity implements
View.OnClickListener {
        // 这是广播的动作名称，发送广播和接收广播都以它作为接头暗号
        private final static String STANDARD_ACTION =
                        "com.example.chapter04.standard";
        private TextView tv_standard;  // 声明一个文本视图对象

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_broad_standard);
            tv_standard = findViewById(R.id.tv_standard);
            findViewById(R.id.btn_send_standard).setOnClickListener(this);
        }

        @Override
        public void onClick(View v) {
            if (v.getId() == R.id.btn_send_standard) {
                Intent intent = new Intent(STANDARD_ACTION);  // 创建指定动作的意图
                sendBroadcast(intent);  // 发送标准广播
            }
        }
    }
```

2. 定义广播接收器

广播发出来之后，还得有设备去接收广播，也就是需要广播接收器。接收器主要规定两个事

情：一个是接收什么样的广播，另一个是收到广播以后要做什么。由于接收器的处理逻辑大同小异，因此Android提供了抽象之后的接收器基类BroadcastReceiver，开发者自定义的接收器都从BroadcastReceiver派生而来。新定义的接收器需要重写onReceive方法，方法内部先判断当前广播是否符合待接收的广播名称，校验通过后再开展后续的业务逻辑。下面是广播接收器的一个定义代码例子：

```
private String mDesc = "这里查看标准广播的收听信息";
// 定义一个标准广播的接收器
private class StandardReceiver extends BroadcastReceiver {
    // 一旦接收到标准广播，马上触发接收器的onReceive方法
    @Override
    public void onReceive(Context context, Intent intent) {
        // 广播意图非空，且接头暗号正确
        if (intent != null && intent.getAction().equals(STANDARD_ACTION)) {
            mDesc = String.format("%s\n%s 收到一个标准广播", mDesc,
                                  DateUtil.getNowTime());
            tv_standard.setText(mDesc);
        }
    }
}
```

3. 开关广播接收器

为了避免资源浪费，还要求合理使用接收器。就像WiFi上网，需要上网时才打开WiFi，不需要上网时就关闭WiFi。广播接收器也是如此，活动页面启动之后才注册接收器，活动页面停止之际就注销接收器。在注册接收器的时候，允许事先指定只接收某种类型的广播，即通过意图过滤器挑选动作名称一致的广播。接收器的注册与注销代码示例如下：

```
private StandardReceiver standardReceiver;      // 声明一个标准广播的接收器实例
@Override
protected void onStart() {
    super.onStart();
    standardReceiver = new StandardReceiver();   // 创建一个标准广播的接收器
    // 创建一个意图过滤器，只处理STANDARD_ACTION的广播
    IntentFilter filter = new IntentFilter(STANDARD_ACTION);
    // 注册接收器，注册之后才能正常接收广播
    registerReceiver(standardReceiver, filter);
}

@Override
protected void onStop() {
    super.onStop();
    unregisterReceiver(standardReceiver);        // 注销接收器，注销之后就不再接收广播
}
```

完成上述3个步骤后，便构建了广播从发送到接收的完整流程。运行测试App，初始的广播界面如图4-16所示，点击“发送标准广播”按钮触发广播，界面下方立刻刷新广播日志，如图4-17所示，可见接收器正确收到广播并成功打印日志。

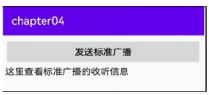


图 4-16　准备接收标准广播

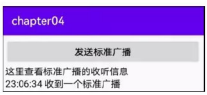


图 4-17　收听到了标准广播

4.3.2 收发有序广播

由于广播没指定唯一的接收者，因此可能存在多个接收器，每个接收器都拥有自己的处理逻辑。这种机制固然灵活，却不够严谨，因为不同接收器之间也许有矛盾。

比如只要办了借书证，大家都能借阅图书馆的藏书，不过一本书被读者甲借出去之后，读者乙就不能再借这本书了，必须等到读者甲归还了该书之后，读者乙方可继续借阅此书。这个借书场景体现了一种有序性，即图书是轮流借阅着的，且同时刻仅能借给一位读者，只有前面的读者借完归还，才轮到后面的读者借阅。另外，读者甲一定会归还此书吗？可能读者甲对该书爱不释手，从图书馆高价购买了这本书；也可能读者甲粗心大意，不小心弄丢了这本书。不管是哪种情况，读者甲都无法还书，导致正在排队的读者乙无书可借。这种借不到书的场景体现了一种依赖关系，即使读者乙迫不及待地想借到书，也得看读者甲的心情，要是读者甲因为各种理由没能还书，那么读者乙就白白排队了。上述的借书业务对应到广播的接收功能，则要求实现下列的处理逻辑：

（1）一个广播存在多个接收器，这些接收器需要排队收听广播，这意味着该广播是条有序广播。

（2）先收到广播的接收器A，既可以让其他接收器继续收听广播，也可以中断广播不让其他接收器收听。

至于如何实现有序广播的收发，则需完成以下3个编码步骤：

1. 发送广播时要注明这是个有序广播

之前发送标准广播用到了sendBroadcast方法，可是该方法发出来的广播是无序的。只有调用sendOrderedBroadcast方法才能发送有序广播，具体的发送代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\BroadOrderActivity.java)
Intent intent = new Intent(ORDER_ACTION);   // 创建一个指定动作的意图
sendOrderedBroadcast(intent, null);         // 发送有序广播
```

2. 定义有序广播的接收器

接收器的定义代码基本不变，也要从BroadcastReceiver继承而来，唯一的区别是有序广播的接收器允许中断广播。倘若在接收器的内部调用abortBroadcast方法，就会中断有序广播，使得后面的接收器不能再接收该广播。下面是有序广播的两个接收器代码例子：

```
private OrderAReceiver orderAReceiver;  // 声明有序广播接收器A的实例
// 定义一个有序广播的接收器A
private class OrderAReceiver extends BroadcastReceiver {
    // 一旦接收到有序广播，马上触发接收器的onReceive方法
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent != null && intent.getAction().equals(ORDER_ACTION)) {
            String desc = String.format("%s%s 接收器A收到一个有序广播\n",
                    tv_order.getText().toString(), DateUtil.getNowTime());
            tv_order.setText(desc);
            if (ck_abort.isChecked()) {
                abortBroadcast();  // 中断广播，此时后面的接收器无法收到该广播
            }
        }
    }
}
```

```
private OrderBReceiver orderBReceiver;  // 声明有序广播接收器B的实例
// 定义一个有序广播的接收器B
private class OrderBReceiver extends BroadcastReceiver {
    // 一旦接收到有序广播B，马上触发接收器的onReceive方法
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent != null && intent.getAction().equals(ORDER_ACTION)) {
            String desc = String.format("%s%s 接收器B收到一个有序广播\n",
                    tv_order.getText().toString(), DateUtil.getNowTime());
            tv_order.setText(desc);
            if (ck_abort.isChecked()) {
                abortBroadcast();  // 中断广播，此时后面的接收器无法收到该广播
            }
        }
    }
}
```

3. 注册有序广播的多个接收器

接收器的注册操作同样调用registerReceiver方法，为了给接收器排队，还需调用意图过滤器的setPriority方法设置优先级，优先级越大的接收器，越先收到有序广播。如果不设置优先级，或者两个接收器的优先级相等，那么越早注册的接收器，会越先收到有序广播。譬如以下的广播注册代码，尽管接收器A更早注册，但接收器B的优先级更高，结果先收到广播的应当是接收器B。

```
orderAReceiver = new OrderAReceiver();      // 创建一个有序广播的接收器A
// 创建一个意图过滤器A，只处理ORDER ACTION的广播
IntentFilter filterA = new IntentFilter(ORDER_ACTION);
filterA.setPriority(8);                     // 设置过滤器A的优先级，数值越大优先级越高
// 注册接收器A，注册之后才能正常接收广播
registerReceiver(orderAReceiver, filterA);
orderBReceiver = new OrderBReceiver();      // 创建一个有序广播的接收器B
// 创建一个意图过滤器B，只处理ORDER ACTION的广播
IntentFilter filterB = new IntentFilter(ORDER_ACTION);
filterB.setPriority(10);                    // 设置过滤器B的优先级，数值越大优先级越高
// 注册接收器B，注册之后才能正常接收广播
registerReceiver(orderBReceiver, filterB);
```

接下来通过测试页面演示有序广播的收发，如果没要求中断广播，则有序广播的接收界面如图4-18所示，此时接收器B和接收器A依次收到了广播；如果要求中断广播，则有序广播的接收界面如图4-19所示，此时只有接收器B收到了广播。

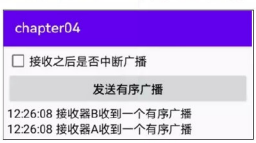


图 4-18　依次接收有序广播

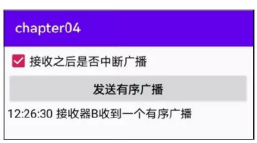


图 4-19　中途打断有序广播

4.3.3　收发静态广播

前面几节在使用广播之时，无一例外在代码中注册了接收器。可是同为四大组件，活动（activity）、服务（service）、内容提供器（provider）都能在AndroidManifest.xml注册，为什么广

播只能在代码中注册呢？其实广播接收器也能在AndroidManifest.xml中注册，并且注册时的节点名为receiver，一旦接收器在AndroidManifest.xml中注册，就无须在代码中注册了。

在AndroidManifest.xml中注册接收器，该方式被称作静态注册；在代码中注册接收器，该方式被称作动态注册。之所以罕见静态注册，是因为静态注册容易导致安全问题，故而Android 8.0之后废弃了大多数静态注册。话虽如此，Android倒也没有彻底禁止静态注册，只要满足特定的编码条件，那么依然能够通过静态方式注册接收器。具体注册步骤说明如下。

01 右击当前模块的默认包，依次选择右键菜单的New→Package，创建名为receiver的新包，用于存放静态注册的接收器代码。

02 右击刚创建的receiver包，依次选择右键菜单的New → Other → Broadcast Receiver，弹出如图4-20所示的广播组件的创建对话框。

03 在组件创建对话框的Class Name一栏填写接收器的类名，比如ShockReceiver，再单击对话框右下角的Finish按钮。之后Android Studio自动在receiver包内创建代码文件ShockReceiver.java，且接收器的默认代码如下：

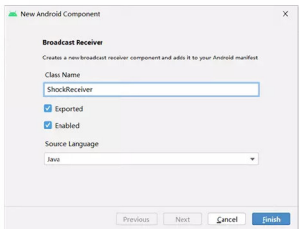


图 4-20 广播组件的创建对话框

```
public class ShockReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

同时AndroidManifest.xml自动添加接收器的节点配置，默认的receiver配置如下：

```
<receiver
    android:name=".receiver.ShockReceiver"
    android:enabled="true"
    android:exported="true"></receiver>
```

然而自动生成的接收器不仅什么都没干，还丢出一个异常UnsupportedOperationException。明显这个接收器没法用，为了感知到接收器正在工作，可以考虑在onReceive方法中记录日志，也可以在该方法中震动手机。因为ShockReceiver未依附于任何活动，自然无法直接操作界面控件，所以只能观察程序日志，或者干脆让手机摇晃起来。实现手机震动，要调用getSystemService方法，先从系统服务VIBRATOR_SERVICE获取震动管理器Vibrator，再调用震动管理器的vibrate方法震动手机。包含手机震动功能的接收器代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\receiver\ShockReceiver.java)
public class ShockReceiver extends BroadcastReceiver {
    // 静态注册时的action、发送广播时的action、接收广播时的action，三者需要保持一致
    public static final String SHOCK_ACTION = "com.example.chapter04.shock";

    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals(ShockReceiver.SHOCK_ACTION)){
            // 从系统服务中获取震动管理器
            Vibrator vb = (Vibrator) context.getSystemService(
                                Context.VIBRATOR_SERVICE);
```

```
                vb.vibrate(500);  // 命令震动器震动若干秒，这里的500表示500毫秒
            }
        }
    }
```

由于震动手机需要申请对应的权限，因此打开AndroidManifest.xml添加以下的权限申请配置：

```
<!-- 震动 -->
<uses-permission android:name="android.permission.VIBRATE" />
```

此外，接收器代码定义了一个动作名称，其值为“com.example.chapter04.shock”，表示onReceive方法只处理过滤该动作之后的广播，从而提高接收效率。除了在代码中过滤之外，还能修改AndroidManifest.xml，在receiver节点内部增加intent-filter标签加以过滤，添加过滤配置后的receiver节点信息如下：

```
<receiver
    android:name=".receiver.ShockReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action android:name="com.example.chapter04.shock" />
    </intent-filter>
</receiver>
```

终于到了发送广播这步，由于Android 8.0之后删除了大部分静态注册，防止App退出后仍在收听广播，因此为了让应用能够继续接收静态广播，需要给静态广播指定包名，也就是调用意图对象的setComponent方法设置组件路径。详细的静态广播发送代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\BroadStaticActivity.java)
String receiverPath = "com.example.chapter04.receiver.ShockReceiver";
// 创建一个指定动作的意图
Intent intent = new Intent(ShockReceiver.SHOCK_ACTION);
// 发送静态广播之时，需要通过setComponent方法指定接收器的完整路径
ComponentName componentName  = new ComponentName(this, receiverPath);
intent.setComponent(componentName);     // 设置意图的组件信息
sendBroadcast(intent);                  // 发送静态广播
```

经过上述的编码以及配置工作，总算完成了静态广播的发送与接收流程。

注　意

经过整改的静态注册只适用于接收 App 自身的广播，不能接收系统广播，也不能接收其他应用的广播。

运行测试App，初始的广播发送界面如图4-21所示，点击“发送震动广播”按钮触发静态广播，接着接收器收到广播信息，手机随之震动了若干时间，说明静态注册的接收器奏效了。

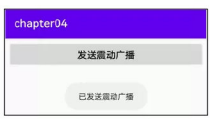


图 4-21　初始的广播发送界面

4.3.4　定时管理器 AlarmManager

Android提供了专门的定时管理器AlarmManager，它利用系统闹钟定时发送广播，能够让App实现定时功能。由于闹钟与震动器同属系统服务，且闹钟的服务名称为ALARM_SERVICE，因此依然调用getSystemService方法获取闹钟管理器的实例，下面是从系统服务中获取闹钟管理器的代码：

```
// 从系统服务中获取闹钟管理器
AlarmManager alarmMgr = (AlarmManager) getSystemService(ALARM_SERVICE);
```

得到闹钟实例后，即可调用它的各种方法设置闹钟规则了，AlarmManager的常见方法说明如下：

- set：**设置一次性定时器。第一个参数为定时器类型，通常填**larmManager.RTC_WAKEUP**；第二个参数为期望的执行时刻（单位为毫秒）；第三个参数为待执行的延迟意图**（PendingIntent**类型）**。**
- setAndAllowWhileIdle：**设置一次性定时器，参数说明同**set**方法，不同之处在于：即使设备处于空闲状态，也会保证执行定时器。因为从**Android 6.0**开始，**set**方法在暗屏时不保证发送广播，必须调用**setAndAllowWhileIdle**方法才能保证发送广播。**
- setRepeating：**设置重复定时器。第一个参数为定时器类型；第二个参数为首次执行时间（单位为毫秒）；第三个参数为下次执行的间隔时间（单位为毫秒）；第四个参数为待执行的延迟意图（**PendingIntent**类型）。然而**setRepeating**方法不保证按时发送广播，只能通过**setAndAllowWhileIdle**方法间接实现重复定时功能。**
- cancel：**取消指定延迟意图的定时器。**

以上的方法说明出现了新名词——延迟意图，它是PendingIntent类型，顾名思义，延迟意图不是马上执行的意图，而是延迟若干时间才执行的意图。像之前的活动页面跳转，调用startActivity方法跳转到下个活动页面，此时跳转动作是立刻发生的，所以要传入Intent对象。由于定时器的广播不是立刻发送的，而是时刻到达了才发送广播，因此不能传Intent对象，只能传PendingIntent对象。当然意图与延迟意图不止这一处区别，它们的差异主要有下列3点：

（1）PendingIntent代表延迟的意图，它指向的组件不会马上被激活；而Intent代表实时的意图，一旦被启动，它指向的组件就会马上被激活。

（2）PendingIntent是一类消息的组合，不但包含目标的Intent对象，还包含请求代码、请求方式等信息。

（3）PendingIntent对象在创建之时便已知晓将要用于活动还是广播，例如调用getActivity方法得到的是活动跳转的延迟意图，调用getBroadcast方法得到的是广播发送的延迟意图。

就闹钟广播的收发过程而言，需要实现3个编码步骤：定义定时器的广播接收器、开关定时器的广播接收器、设置定时器的播报规则，分别叙述如下。

1．定义定时器的广播接收器

闹钟广播的接收器采用动态注册方式，它的实现途径与标准广播类似，都要从BroadcastReceiver派生新的接收器，并重写onReceive方法。闹钟广播接收器的定义代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\AlarmActivity.java)
// 声明一个闹钟广播事件的标识串
private String ALARM_ACTION = "com.example.chapter04.alarm";
private String mDesc = "";  // 闹钟时间到达的描述
// 定义一个闹钟广播的接收器
public class AlarmReceiver extends BroadcastReceiver {
    // 一旦接收到闹钟时间到达的广播，马上触发接收器的onReceive方法
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent != null) {
            mDesc = String.format("%s\n%s 闹钟时间到达", mDesc, DateUtil.getNowTime());
```

```
            tv_alarm.setText(mDesc);
            // 从系统服务中获取震动管理器
            Vibrator vb = (Vibrator) context.getSystemService(Context.VIBRATOR_SERVICE);
            vb.vibrate(500);  // 命令震动器震动若干秒
        }
    }
}
```

2．开关定时器的广播接收器

定时接收器的开关流程参照标准广播，可以在活动页面的onStart方法中注册接收器，在活动页面的onStop方法中注销接收器。相应的接收器开关代码如下：

```
private AlarmReceiver alarmReceiver;                    // 声明一个闹钟的广播接收器
@Override
public void onStart() {
    super.onStart();
    alarmReceiver = new AlarmReceiver();                // 创建一个闹钟的广播接收器
    // 创建一个意图过滤器，只处理指定事件来源的广播
    IntentFilter filter = new IntentFilter(ALARM_ACTION);
    // 注册接收器，注册之后才能正常接收广播
    registerReceiver(alarmReceiver, filter);
}

@Override
public void onStop() {
    super.onStop();
    unregisterReceiver(alarmReceiver);                  // 注销接收器，注销之后就不再接收广播
}
```

3．设置定时器的播报规则

首先从系统服务中获取闹钟管理器，然后调用管理器的set***方法，把事先创建的延迟意图填到播报规则当中。下面是发送闹钟广播的代码例子:

```
// 发送闹钟广播
private void sendAlarm() {
    Intent intent = new Intent(ALARM_ACTION);    // 创建一个广播事件的意图
    // 创建一个用于广播的延迟意图
    PendingIntent pIntent = PendingIntent.getBroadcast(this, 0,
            intent, PendingIntent.FLAG_UPDATE_CURRENT);
    // 从系统服务中获取闹钟管理器
    AlarmManager alarmMgr = (AlarmManager) getSystemService(ALARM_SERVICE);
    // 给当前时间加上若干秒
    long delayTime = System.currentTimeMillis() + mDelay*1000;
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        // 允许在空闲时发送广播，Android 6.0之后新增的方法
        alarmMgr.setAndAllowWhileIdle(AlarmManager.RTC_WAKEUP,  delayTime, pIntent);
    } else {
        // 设置一次性闹钟，延迟若干秒后，携带延迟意图发送闹钟广播
        alarmMgr.set(AlarmManager.RTC_WAKEUP, delayTime, pIntent);
    }
}
```

完成上述的3个步骤之后，运行测试App，点击“设置闹钟”按钮，界面下方回显闹钟的设置信息，如图4-22所示。稍等片刻，发现回显文本多了一行日志，如图4-23所示，同时手机也嗡嗡震动了一会，对比日志时间可知，闹钟广播果然在设定的时刻触发且收听了。



图 4-22 刚刚设置闹钟

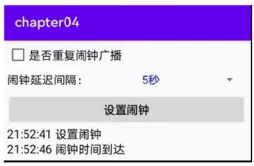


图 4-23 收到闹钟广播

至于闹钟的重复播报问题，因为setRepeating方法不再可靠，所以要修改闹钟的收听逻辑，在onReceive末尾补充调用sendAlarm方法，确保每次收到广播之后立即准备下一个广播。调整以后的onReceive方法代码示例如下：

```
public void onReceive(Context context, Intent intent) {
    if (intent != null) {
        // 这里省略现有的广播处理代码
        if (ck_repeate.isChecked()) {  // 需要重复闹钟广播
            sendAlarm();  // 发送闹钟广播
        }
    }
}
```

4.4 操作后台服务

本节介绍Android四大组件之一Service的基本概念和常见用法。包括服务的生命周期及其两种启停方式——普通方式和绑定方式（含立即绑定和延迟绑定），还介绍了如何在活动和服务之间交互数据。

4.4.1 服务的启动和停止

Service是Android的四大组件之一，它常用于看不见页面的高级场合，例如“4.3.3 收发静态广播”提到了系统的震动服务、“4.3.4 定时器AlarmManager”提到了系统的闹钟服务，等等。这些系统服务平时几乎感觉不到它们的存在，却是系统不可或缺的重要组成部分。

既然Android自带了系统服务， App也可以拥有自己的服务。Service与Activity相比，不同之处在于没有对应的页面，相同之处在于都有生命周期。要想用好服务，就要弄清楚它的生命周期。

Service与生命周期有关的方法说明如下：

- onCreate：**创建服务**。
- onStart：**开始服务，Android 2.0以下版本使用，现已废弃**。
- onStartCommand：**开始服务，Android 2.0及以上版本使用。该方法的返回值说明见表**4-6。

表 4-6　服务启动的返回值说明

返回值类型	返回值说明
START_STICKY	黏性的服务。如果服务进程被杀掉，就保留服务的状态为开始状态，但不保留传送的 Intent 对象。随后系统尝试重新创建服务，由于服务状态为开始状态，因此创建服务后一定会调用 onStartCommand 方法。如果在此期间没有任何启动命令传送给服务，参数 Intent 就为空值
START_NOT_STICKY	非黏性的服务。使用这个返回值时，如果服务被异常杀掉，系统就不会自动重启该服务
START_REDELIVER_INTENT	重传 Intent 的服务。使用这个返回值时，如果服务被异常杀掉，系统就会自动重启该服务，并传入 Intent 的原值
START_STICKY_COMPATIBILITY	START_STICKY 的兼容版本，但不保证服务被杀掉后一定能重启

- onDestroy：**销毁服务**。
- onBind：**绑定服务**。
- onUnbind：**解除绑定。返回值为true表示允许再次绑定，之后再绑定服务时，不会调用onBind方法而是调用onRebind方法；返回值为false表示只能绑定一次，不能再次绑定。**
- onRebind：**重新绑定。只有上次的onUnbind方法返回true时，再次绑定服务才会调用onRebind方法。**

看来服务的生命周期也存在好几个环节，除了必须的onCreate方法和onDestroy方法，还有其他几种生命周期方法。接下来以普通服务的启停为例，讲解服务的生命周期过程。

首先在Java代码包下面创建名为service的新包，右击该包并在右键菜单中依次选择New→Service→Service，弹出如图4-24所示的服务创建对话框。

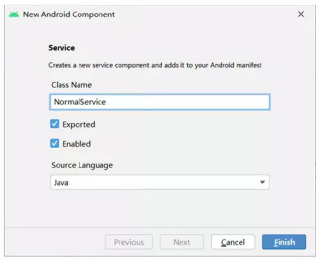


图 4-24　服务创建对话框

在服务创建对话框的Class Name一栏填写服务名称，比如NormalService，再单击对话框右下角的Finish按钮，Android Studio便自动在service包下生成NormalService.java，同时在AndroidManifest.xml的application节点内部添加如下的服务注册配置：

```
<service android:name=".service.NormalService"
    android:enabled="true" android:exported="true"></service>
```

打开NormalService.java发现里面只有几行代码，为了方便观察服务的生命周期过程，需要重写该服务的所有周期方法，给每个方法都打印相应的运行日志，修改之后的服务代码如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\service\NormalService.java)
public class NormalService extends Service {
    private void refresh(String text) {
        ServiceNormalActivity.showText(text);
    }

    @Override
```

```
    public void onCreate() {                    // 创建服务
        super.onCreate();
        refresh("onCreate");
    }
    // 启动服务
    @Override
    public int onStartCommand(Intent intent, int flags, int startid) {
        refresh("onStartCommand. flags=" + flags);
        return START_STICKY;
    }
    @Override
    public void onDestroy() {                       // 销毁服务
        super.onDestroy();
        refresh("onDestroy");
    }
    // 绑定服务。普通服务不存在绑定和解绑流程
    @Override
    public IBinder onBind(Intent intent) {
        refresh("onBind");
        return null;
    }
    @Override
    public void onRebind(Intent intent) {           // 重新绑定服务
        super.onRebind(intent);
        refresh("onRebind");
    }
    @Override
    public boolean onUnbind(Intent intent) {        // 解绑服务
        refresh("onUnbind");
        return true;  // 返回false表示只能绑定一次，返回true表示允许多次绑定
    }
}
```

启停普通服务很简单，只要创建一个指向服务的意图，然后调用startService方法即可启动服务，若要停止服务，调用stopService方法即可停止指定意图的服务。具体的服务启停代码示例如下：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ServiceNormalActivity.java)
// 创建一个通往普通服务的意图
Intent intent = new Intent(this, NormalService.class);
startService(intent);           // 启动指定意图的服务
//stopService(mIntent);         // 停止指定意图的服务
```

运行测试App，点击“启动服务”按钮，监听器调用了startService方法，此时测试界面如图4-25所示，可见服务的启动操作依次触发了onCreate和onStartCommand方法。接着点击“停止服务”按钮，监听器调用了stopService方法，此时测试界面如图4-26所示，可见服务的停止操作触发了onDestroy方法。



图 4-25 启动服务的界面日志



图 4-26 停止服务的界面日志

4.4.2 服务的绑定与解绑

服务启停除了上一小节介绍的普通方式外，Android还提供了另一种启停方式，也就是绑定服务和解绑服务。因为服务可能由组件甲创建却被组件乙所使用，也可能服务由进程A创建却由进程B使用，好比一块土地被它的主人租给其他人使用那样，所有者与使用者并非同一个人。既然所有者与使用者不是同一个人，就需要两人之间订立租约，规定土地的租赁关系。

对于服务来说，便要求提供黏合剂Binder指定服务的绑定关系，同时黏合剂还负责在两个组件或者在两个进程之间交流通信。此时增加了黏合剂的服务代码示例如下：

```
    （完整代码见chapter04\src\main\java\com\example\chapter04\service\
BindImmediateService.java）
    public class BindImmediateService extends Service {
        private final IBinder mBinder = new LocalBinder();  // 创建一个黏合剂对象
        // 定义一个当前服务的黏合剂，用于将该服务黏到活动页面的进程中
        public class LocalBinder extends Binder {
            public BindImmediateService getService() {
                return BindImmediateService.this;
            }
        }
        private void refresh(String text) {
            BindImmediateActivity.showText(text);
        }
        @Override
        public void onCreate() {                        // 创建服务
            super.onCreate();
            refresh("onCreate");
        }
        @Override
        public void onDestroy() {                       // 销毁服务
            super.onDestroy();
            refresh("onDestroy");
        }
        @Override
        public IBinder onBind(Intent intent) {          // 绑定服务。返回该服务的黏合剂对象
            refresh("onBind");
            return mBinder;
        }
        @Override
        public void onRebind(Intent intent) {           // 重新绑定服务
            super.onRebind(intent);
            refresh("onRebind");
        }
        @Override
        public boolean onUnbind(Intent intent) {        // 解绑服务
            refresh("onUnbind");
            return true;                  // 返回false表示只能绑定一次，返回true表示允许多次绑定
        }
    }
```

对于绑定了黏合剂的服务，它的绑定和解绑操作与普通方式不同：首先要定义一个ServiceConnection的服务连接对象，然后调用bindService方法绑定服务，绑定之后再择机调用unbindService方法解绑服务，具体的活动代码示例如下：

```
    (完整代码见chapter04\src\main\java\com\example\chapter04\BindImmediateActivity.java)
    public class BindImmediateActivity extends AppCompatActivity implements
View.OnClickListener {
        private static TextView tv immediate;          // 声明一个文本视图对象
        private Intent mIntent;                        // 声明一个意图对象
        private static String mDesc;                   // 日志描述
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity bind immediate);
            tv immediate = findViewById(R.id.tv immediate);
            findViewById(R.id.btn_start_bind).setOnClickListener(this);
            findViewById(R.id.btn_unbind).setOnClickListener(this);
            mDesc = "";
            // 创建一个通往立即绑定服务的意图
            mIntent = new Intent(this, BindImmediateService.class);
        }
        @Override
        public void onClick(View v) {
            if (v.getId() == R.id.btn_start_bind) {  // 点击了绑定服务按钮
                // 绑定服务。如果服务未启动，则系统先启动该服务再进行绑定
                boolean bindFlag = bindService(mIntent, mFirstConn,
                                    Context.BIND AUTO CREATE);
            } else if (v.getId() == R.id.btn_unbind) {  // 点击了解绑服务按钮
                if (mBindService != null) {
                    // 解绑服务。如果先前服务立即绑定，则此时解绑之后自动停止服务
                    unbindService(mFirstConn);
                    mBindService = null;
                }
            }
        }
        public static void showText(String desc) {
            if (tv immediate != null) {
                mDesc = String.format("%s%s %s\n", mDesc,
                            DateUtil.getNowDateTime("HH:mm:ss"), desc);
                tv immediate.setText(mDesc);
            }
        }
        private BindImmediateService mBindService;  // 声明一个服务对象
        private ServiceConnection mFirstConn = new ServiceConnection() {
            // 获取到服务对象时的操作
            public void onServiceConnected(ComponentName name, IBinder service) {
                // 如果服务运行于另外一个进程，则不能直接强制转换类型，否则会报错
                mBindService = ((BindImmediateService.LocalBinder) service).getService();
            }
            // 无法获取到服务对象时的操作
            public void onServiceDisconnected(ComponentName name) {
                mBindService = null;
            }
        };
    }
```

运行测试App，点击“启动并绑定服务”按钮之后，观察到日志界面如图4-27所示，可见此时依次调用了onCreate和onBind方法。然后点击“解绑并停止服务”按钮，观察到日志界面如图4-28所示，可见此时依次调用了onUnbind和onDestroy方法。

上述的服务绑定与解绑操作，其实并不纯粹，因为调用bindService方法时先后触发了onCreate和onBind，也就是创建服务后紧接着绑定服务；调用unbindService方法时先后触发了onUnbind和

onDestroy，也就是解绑服务后紧接着销毁服务。既然服务的创建操作后面紧跟着绑定操作，它们的时空关系近似于普通启停，又何必另外设计绑定流程呢？



图 4-27　立即绑定的日志界面



图 4-28　立即解绑的日志界面

诚然这种立即绑定体现不了差异化情况，为了更好地说明绑定方式的优势，Android还提供了另一种延迟绑定。延迟绑定与立即绑定的区别在于：延迟绑定要先通过startService方法启动服务，再通过bindService方法绑定已存在的服务；同理，延迟解绑要先通过unbindService方法解绑服务，再通过stopService方法停止服务。这样一来，因为启动操作在先、绑定操作在后，所以解绑操作只能撤销绑定操作，而不能撤销启动操作。由于解绑操作不能销毁服务，因此存在再次绑定服务的可能。

接下来做个实验，观察一下延迟绑定是否允许重新绑定操作，演示代码路径为chapter04\src\main\java\com\example\chapter04\BindDelayActivity.java，测试页面上提供了4个按钮：“启动服务”“绑定服务”“解绑服务”“停止服务”，分别对应startService、bindService、unbindService、stopService 4个方法。第一个实验依次点击“启动服务”→“绑定服务”→“解绑服务”→“停止服务”按钮，此时日志界面如图4-29所示。第二个实验依次点击“启动服务”→“绑定服务”→“解绑服务”→“绑定服务”按钮，此时日志界面如图4-30所示。



图 4-29　延迟绑定的日志界面



图 4-30　再次绑定的日志界面

从上面两个日志界面可知，延迟绑定与立即绑定这两种方式的生命周期区别在于：

（1）延迟绑定的首次绑定操作只触发onBind方法，再次绑定操作只触发onRebind方法（是否允许再次绑定要看上次onUnbind方法的返回值）。

（2）延迟绑定的解绑操作只触发onUnbind方法。

4.4.3　活动与服务之间的交互

不管是startService方法，还是bindService方法，都支持将意图对象作为参数，这意味着在启动服务或绑定服务之时能够向服务传递信息。可是服务跑起来以后就一直在运行，活动代码怎么知道服务跑得快还是跑得慢，须知服务并不提供回调机制，活动守株待兔是等不到结果的。若想及时获取服务的运行情况，活动就得主动打探消息，此时需要有个信使承担消息传输的任务，这个信使便是绑定方式用到的服务黏合剂——IBinder。

注意看服务代码的onBind方法，它的返回值类型正是IBinder，表示绑定成功后会返回服务的黏合剂对象。只要活动代码拿到了服务的黏合剂对象，就能通过黏合剂与服务进行数据交互。由于IBinder是个接口，它的实现类名叫Binder，因此每个服务的黏合剂都得从Binder派生而来。除了定义getService方法返回当前的服务对象之外，黏合剂还可以定义一般的数据交互方法，用于同活动代码往来通信。下面是与黏合剂有关的服务定义代码片段：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\service\DataService.java)
private final IBinder mBinder = new LocalBinder();  // 创建一个黏合剂对象

// 定义一个当前服务的黏合剂，用于将该服务粘到活动页面的进程中
public class LocalBinder extends Binder {
    public DataService getService() {
        return DataService.this;
    }

    // 获取数字描述
    public String getNumber(int number) {
        return "我收到了数字"+number;
    }
}

@Override
public IBinder onBind(Intent intent) {  // 绑定服务。返回该服务的黏合剂对象
    return mBinder;
}
```

活动代码在调用bindService方法时，第二个参数为ServiceConnection类型，表示绑定结果的连接对象。这个连接对象来自接口ServiceConnection，它的onServiceConnected方法在连接成功时回调，onServiceDisconnected方法在连接断开时回调。重写ServiceConnection的onServiceConnected方法，即可拿到已绑定服务的黏合剂对象。有了服务的黏合剂，才能通过黏合剂获取服务的内部情况。比如以下代码演示了如何利用黏合剂与服务通信：

```
(完整代码见chapter04\src\main\java\com\example\chapter04\ServiceDataActivity.java)
private DataService.LocalBinder mBinder;  // 声明一个黏合剂对象

private ServiceConnection mServiceConn = new ServiceConnection() {
    // 获取到服务对象时的操作
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        mBinder = (DataService.LocalBinder) service;
        // 活动代码通过黏合剂与服务代码通信
        String response = mBinder.getNumber(new Random().nextInt(100));
        tv_result.setText(DateUtil.getNowTime()+" 绑定服务应答："+response);
    }

    // 无法获取到服务对象时的操作
    @Override
    public void onServiceDisconnected(ComponentName name) {
        mBinder = null;
    }
};
```

运行测试App，点击“启动并绑定服务”按钮，观察到通信结果如图4-31所示，可见成功实现了活动与服务之间的数据交互。

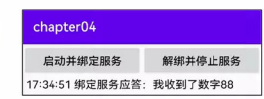


图 4-31 活动与服务的数据交互结果

4.5 小　结

本章主要介绍了Android三大常用组件：活动组件Activity、广播组件Broadcast、服务组件Service的常见用法，包括：启停活动页面（Activity的启动和结束、Activity的生命周期、Activity的启动模式）、在活动之间传递消息（显式Intent和隐式Intent、普通的活动数据交互、改进后的活动数据交互）、收发应用广播（收发标准广播、收发有序广播、收发静态广播、定时管理器AlarmManager）、操作后台服务（服务的启动和停止、服务的绑定与解绑、活动与服务之间的交互）。

通过本章的学习，读者应该能掌握以下4种开发技能：

（1）理解活动的生命周期过程，并学会正确启动和结束活动。

（2）理解意图的组成结构，并学会利用意图在活动之间传递消息。

（3）了解广播的应用场景，并学会正确收发应用广播。

（4）了解服务的生命周期，并学会服务的两种启停方式（普通方式和绑定方式）。

4.6 动手练习

请上机实验下列3项练习：

1. 创建两个活动页面，分别模拟注册页面和完成页面，先从注册页面跳转到完成页面，但是在完成页面按返回键，不能回到注册页面（因为注册成功之后无须重新注册）。

2. 创建两个活动页面，从A页面携带请求数据跳转到B页面，B页面应当展示A页面传来的信息；然后B页面向A页面返回应答数据，A页面也要展示B页面返回的信息。

3. 通过设置不同的优先级，实现有序广播的正确收发。

第 5 章

中 级 控 件

本章介绍App开发常见的几类中级控件的用法，主要包括：如何定制几种简单的图形、如何使用几种选择按钮、如何高效地输入文本、如何利用对话框获取交互信息等，然后结合本章所学的知识，演示了一个实战项目“找回密码”的设计与实现。

5.1 图 形 定 制

本节介绍Android图形的基本概念和几种常见图形的使用办法，包括：形状图形的组成结构及其具体用法、九宫格图片（点九图片）的制作过程及其适用场景、状态列表图形的产生背景及其具体用法。

5.1.1 图形 Drawable

Android把所有能够显示的图形都抽象为Drawable类（可绘制的）。这里的图形不止是图片，还包括色块、画板、背景等。

包含图片在内的图形文件放在res目录的各个drawable目录下，其中drawable目录一般保存描述性的XML文件，而图片文件一般放在具体分辨率的drawable目录下。例如：

- drawable-ldpi**里面存放低分辨率的图片（如**240×320**），现在基本没有这样的智能手机了。**
- drawable-mdpi**里面存放中等分辨率的图片（如**320×480**），这样的智能手机已经很少了。**
- drawable-hdpi**里面存放高分辨率的图片（如**480×800**），一般对应**4**英寸～**4.5**英寸的手机（但不绝对，同尺寸的手机有可能分辨率不同，手机分辨率就高不就低，因为分辨率低了屏幕会有模糊的感觉）。**
- drawable-xhdpi**里面存放加高分辨率的图片（如**720×1280**），一般对应**5**英寸～**5.5**英寸的手机。**
- drawable-xxhdpi**里面存放超高分辨率的图片（如**1080×1920**），一般对应**6**英寸～**6.5**英寸的手机。**
- drawable-xxxhdpi**里面存放超超高分辨率的图片（如**1440×2560**），一般对应**7**英寸以上的平板计算机。**

基本上，分辨率每加大一级，宽度和高度就要增加二分之一或三分之一像素。如果各目录存

在同名图片，Android就会根据手机的分辨率分别适配对应文件夹里的图片。在开发App时，为了兼容不同的手机屏幕，在各目录存放不同分辨率的图片，才能达到最合适的显示效果。例如，在drawable-hdpi里放了一张背景图片bg.png（分辨率为480×800），其他目录没放，使用分辨率为480×800的手机查看该App界面没有问题，但是使用分辨率为720×1280的手机查看该App会发现背景图片有点模糊，原因是Android为了让bg.png适配高分辨率的屏幕，强行把bg.png拉伸到了720×1280，拉伸的后果是图片变模糊了。

在XML布局文件中引用图形文件可使用“@drawable/不含扩展名的文件名称”这种形式，如各视图的background属性、ImageView和ImageButton的src属性、TextView和Button四个方向的drawable***系列属性都可以引用图形文件。

5.1.2 形状图形

Shape图形又称形状图形，用来描述常见的几何形状，包括矩形、圆角矩形、圆形、椭圆等。用好形状图形可以让App页面不再呆板，还可以节省美工不少工作量。

形状图形的定义文件放在drawable目录下，它是以shape标签为根节点的XML描述文件。根节点下定义了6个节点，分别是size（尺寸）、stroke（描边）、corners（圆角）、solid（填充）、padding（间隔）、gradient（渐变），各节点的属性值主要是宽和高、半径、角度以及颜色等。下面是形状图形各个节点及其属性的简要说明。

1. shape（形状）

shape是形状图形文件的根节点，它描述了当前是哪种几何图形。下面是shape节点的常用属性说明。

- shape：**字符串类型，表示图形的形状。形状类型的取值说明见表**5-1。

表 5-1 形状类型的取值说明

形状类型	说　明
rectangle	矩形。默认值
oval	椭圆。此时 corners 节点会失效
line	直线。此时必须设置 stroke 节点，不然会报错
ring	圆环

2. size（尺寸）

size是shape的下级节点，它描述了形状图形的宽高尺寸。若无size节点，则表示宽高与宿主视图一样大小。下面是size节点的常用属性说明。

- height：**像素类型，图形高度**。
- width：**像素类型，图形宽度**。

3. stroke（描边）

stroke是shape的下级节点，它描述了形状图形的描边规格。若无stroke节点，则表示不存在描边。下面是stroke节点的常用属性说明。

- color：颜色类型，描边的颜色。
- dashGap：像素类型，每段虚线之间的间隔。
- dashWidth：像素类型，每段虚线的宽度。若dashGap和dashWidth有一个值为0，则描边为实线。
- width：像素类型，描边的厚度。

4. corners（圆角）

corners是shape的下级节点，它描述了形状图形的圆角大小。若无corners节点，则表示没有圆角。下面是corners节点的常用属性说明。

- bottomLeftRadius：像素类型，左下圆角的半径。
- bottomRightRadius：像素类型，右下圆角的半径。
- topLeftRadius：像素类型，左上圆角的半径。
- topRightRadius：像素类型，右上圆角的半径。
- radius：像素类型，4个圆角的半径（若有上面4个圆角半径的定义，则不需要radius定义）。

5. solid（填充）

solid是shape的下级节点，它描述了形状图形的填充色彩。若无solid节点，则表示无填充颜色。下面是solid节点的常用属性说明。

- color：颜色类型，内部填充的颜色。

6. padding（间隔）

padding是shape的下级节点，它描述了形状图形与周围边界的间隔。若无padding节点，则表示四周不设间隔。下面是padding节点的常用属性说明。

- top：像素类型，与上方的间隔。
- bottom：像素类型，与下方的间隔。
- left：像素类型，与左边的间隔。
- right：像素类型，与右边的间隔。

7. gradient（渐变）

gradient是shape的下级节点，它描述了形状图形的颜色渐变。若无gradient节点，则表示没有渐变效果。下面是gradient节点的常用属性说明。

- angle：整型，渐变的起始角度。为0时表示时钟的9点位置，值增大表示往逆时针方向旋转。例如，值为90表示6点位置，值为180表示3点位置，值为270表示0点/12点位置。
- type：字符串类型，渐变类型。渐变类型的取值说明见表5-2。

表 5-2　渐变类型的取值说明

渐变类型	说　明
linear	线性渐变，默认值
radial	放射渐变，起始颜色就是圆心颜色
sweep	滚动渐变，即一个线段以某个端点为圆心做 360° 旋转

- centerX：浮点型，圆心的X坐标。当android:type="linear"时不可用。
- centerY：浮点型，圆心的Y坐标。当android:type="linear"时不可用。
- gradientRadius：整型，渐变的半径。当android:type="radial"时需要设置该属性。
- centerColor：颜色类型，渐变的中间颜色。
- startColor：颜色类型，渐变的起始颜色。
- endColor：颜色类型，渐变的终止颜色。
- useLevel：布尔类型，设置为true为无渐变色、false为有渐变色。

在实际开发中，形状图形主要使用3个节点：stroke（描边）、corners（圆角）和solid（填充）。至于shape根节点的属性一般不用设置（默认矩形即可）。

接下来演示一下形状图形的界面效果，首先右击drawable目录，并依次选择右键菜单的New→Drawable resource file，在弹窗中输入文件名称再单击OK按钮，即可自动生成一个XML描述文件。往该文件填入下面的圆角矩形内容定义：

```
(完整代码见chapter05\src\main\res\drawable\shape_rect_gold.xml)
<shape xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- 指定了形状内部的填充颜色 -->
    <solid android:color="#ffdd66" />
    <!-- 指定了形状轮廓的粗细与颜色 -->
    <stroke
        android:width="1dp"
        android:color="#aaaaaa" />
    <!-- 指定了形状4个圆角的半径 -->
    <corners android:radius="10dp" />
</shape>
```

接着创建一个测试页面，并在页面的XML文件中添加名为v_content的View标签，再给Java代码补充以下的视图背景设置代码：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\DrawableShapeActivity.java)
// 从布局文件中获取名为v_content的视图
View v_content = findViewById(R.id.v_content);
// v_content的背景设置为圆角矩形
v_content.setBackgroundResource(R.drawable.shape_rect_gold);
```

然后运行测试App，观察到对应的形状图形如图5-1所示。该形状为一个圆角矩形，内部填充色为土黄色，边缘线为灰色。

再来一个椭圆的XML描述文件，示例代码如下：

```
(完整代码见chapter05\src\main\res\drawable\shape_oval_rose.xml)
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval" >
    <!-- 指定了形状内部的填充颜色 -->
    <solid android:color="#ff66aa" />
    <!-- 指定了形状轮廓的粗细与颜色 -->
    <stroke
        android:width="1dp"
        android:color="#aaaaaa" />
</shape>
```

把前述的视图对象v_content背景改为R.drawable.shape_oval_rose，运行App观察到对应的形状图形如图5-2所示。该形状为一个椭圆，内部填充色为玫红色，边缘线为灰色。



图 5-1　圆角矩形效果

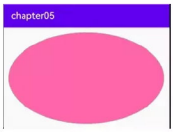


图 5-2　椭圆图形效果

5.1.3　九宫格图片

将某张图片设置成视图背景时，如果图片尺寸太小，则系统会自动拉伸图片使之填满背景。可是一旦图片拉得过大，其画面容易变得模糊，如图5-3所示，上面按钮的背景图片被拉得很宽，此时左右两边的边缘线既变宽又变模糊了。



图 5-3　普通图片与九宫格图片的拉伸效果对比

为了解决这个问题，Android专门设计了点九图片。点九图片的扩展名是.png，文件名后面常带有“.9”字样。因为该图片划分了3×3的九宫格区域，所以得名点九图片，也叫作九宫格图片。如果背景是一个形状图形，其stroke节点的width属性已经设置了固定数值（如1dp），那么无论该图形被拉到多大，描边宽度始终是1dp。九宫格图片的实现原理与之类似，即拉伸图形时，只拉伸内部区域，不拉伸边缘线条。

为了演示九宫格图片的展示效果，利用Android Studio制作一张点九图片。首先在drawable目录下找到待加工的原始图片button_pressed_orig.png，右击它弹出右键菜单，如图5-4所示。

选择右键菜单下面的Create 9-Patch files…，并在随后弹出的对话框中单击OK按钮。接着drawable目录自动生成一个名为“button_pressed_orig.9.png”的图片，双击该文件，主界面右侧弹出如图5-5所示的点九图片的加工窗口。

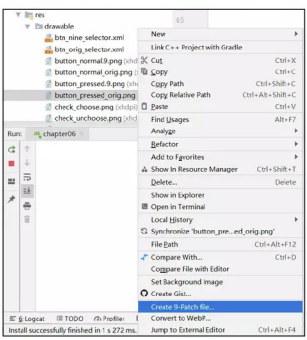


图 5-4　点九图片的制作菜单路径

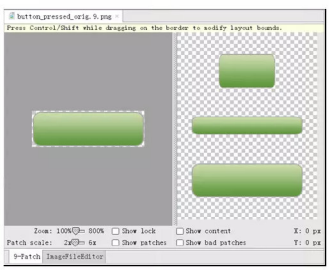


图 5-5　点九图片的加工窗口界面

注意图5-5的左侧窗口是图片加工区域，右侧窗口是图片预览区域，从上到下依次是纵向拉伸预览、横向拉伸预览、两方向同时拉伸预览。在左侧窗口图片四周的马赛克处单击会出现一个黑点，把黑点左右或上下拖动会拖出一段黑线，不同方向上的黑线表示不同的效果。

如图5-6所示，界面上边的黑线指的是水平方向的拉伸区域。在水平方向拉伸图片时，只有黑线区域内的图像会被拉伸，黑线以外的图像保持原状，从而保证左右两侧的边框厚度不变。

如图5-7所示，界面左边的黑线指的是垂直方向的拉伸区域。在垂直方向拉伸图片时，只有黑线区域内的图像会被拉伸，黑线以外的图像保持原状，从而保证上下两侧的边框厚度不变。



图 5-6　点九图片上边的黑线



图 5-7　点九图片左边的黑线

如图5-8所示，界面下边的黑线指的是该图片作为控件背景时，控件内部的文字左右边界只能放在黑线区域内。这里Horizontal Padding的效果就相当于android:paddingLeft与android:paddingRight。

如图5-9所示，界面右边的黑线指的是该图片作为控件背景时，控件内部的文字上下边界只能放在黑线区域内。这里Vertical Padding的效果就相当于android:paddingTop与android:paddingBottom。



图 5-8　点九图片下边的黑线



图 5-9　点九图片右边的黑线

注　意

如果点九图片被设置为视图背景，且该图片指定了 Horizontal Padding 和 Vertical Padding，那么视图内部将一直与视图边缘保持固定间距，无论怎么调整 XML 文件和 Java 代码都无法缩小间隔，缘由是点九图片早已在水平和垂直方向都设置了 padding。

5.1.4　状态列表图形

常见的图形文件一般为静态图形，但有时会用到动态图形，比如按钮控件的背景在正常情况下是凸起的，在按下时是凹陷的，从按下到弹起的过程，用户便知道点击了该按钮。根据不同的触摸情况变更图形状态，这种情况用到了Drawable的一个子类StateListDrawable（状态列表图形），它在XML文件中规定了不同状态所呈现的图形列表。

接下来演示一下状态列表图形的界面效果，右击drawable目录，并依次选择右键菜单的New→Drawable resource file，在弹窗中输入文件名称再单击OK按钮，即可自动生成一个XML描述文件。往该文件填入下面的状态列表图形定义：

```
（完整代码见chapter05\src\main\res\drawable\btn_nine_selector.xml）
```

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:drawable="@drawable/button_pressed" />
    <item android:drawable="@drawable/button_normal" />
</selector>
```

上述XML文件的关键点是state_pressed属性，该属性表示按下状态，值为true表示按下时显示button_pressed图像，其余情况显示button_normal图像。

为方便理解，接下来做个实验，首先将“定制样式的按钮”这一按钮控件的background属性设置为@drawable/btn_nine_selector，然后在屏幕上点击该按钮，观察发现按下按钮时的界面如图5-10所示，而松开按钮时的界面如图5-11所示，可见按下与松开时果然显示不同的图片。

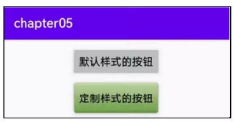


图 5-10　按下按钮时的背景样式

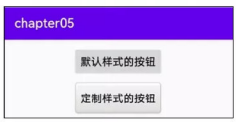


图 5-11　松开按钮时的背景样式

状态列表图形不仅用于按钮控件，还可用于其他拥有多种状态的控件，这取决于开发者在XML文件中指定了哪种状态类型。各种状态类型的取值说明详见表5-3。

表 5-3　状态类型的取值说明

状态类型的属性名称	说　明	适用的控件
state_pressed	是否按下	按钮 Button
state_checked	是否勾选	复选框 CheckBox、单选按钮 RadioButton
state_focused	是否获取焦点	文本编辑框 EditText
state_selected	是否选中	各控件通用

5.2　选 择 按 钮

本节介绍几个常用的特殊控制按钮，包括：如何使用复选框CheckBox及其勾选监听器、如何使用开关按钮Switch、如何借助状态列表图形实现仿iOS的开关按钮、如何使用单选按钮RadioButton和单选组RadioGroup及其选中监听器。

5.2.1　复选框 CheckBox

在学习复选框之前，先了解一下CompoundButton类。在Android体系中，CompoundButton类是抽象的复合按钮，因为是抽象类，所以它不能直接使用。在实际开发中用的是CompoundButton的几个派生类，主要有复选框CheckBox、单选按钮RadioButton以及开关按钮Switch，这些派生类均可使用CompoundButton的属性和方法。加之CompoundButton本身继承了Button类，故以上几种按钮同时具备Button的属性和方法，它们之间的继承关系如图5-12所示。

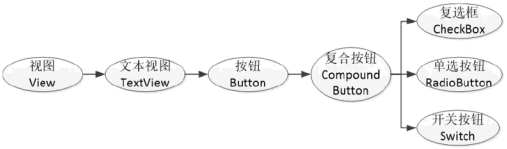


图 5-12　复合按钮的继承关系

CompoundButton在XML文件中主要使用下面两个属性。

- checked：**指定按钮的勾选状态，true表示勾选，false则表示未勾选。默认为未勾选。**
- button：**指定左侧勾选图标的图形资源，如果不指定就使用系统的默认图标。**

CompoundButton在Java代码中主要使用下列4个方法。

- setChecked：**设置按钮的勾选状态。**
- setButtonDrawable：**设置左侧勾选图标的图形资源。**
- setOnCheckedChangeListener：**设置勾选状态变化的监听器。**
- isChecked：**判断按钮是否勾选。**

复选框CheckBox是CompoundButton一个最简单的实现控件，点击复选框将它勾选，再次点击取消勾选。复选框对象调用setOnCheckedChangeListener方法设置勾选监听器，这样在勾选和取消勾选时就会触发监听器的勾选事件。

接下来演示复选框的操作过程，首先编写活动页面的XML文件，代码如下：

```
(完整代码见chapter05\src\main\res\layout\activity_check_box.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <CheckBox
        android:id="@+id/ck_system"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:checked="false"
        android:text="这是系统的CheckBox" />
</LinearLayout>
```

接着编写对应的Java代码，主要是如何处理勾选监听器，具体代码如下：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\CheckBoxActivity.java)
// 该页面实现了接口OnCheckedChangeListener，要重写监听器的onCheckedChanged方法
public class CheckBoxActivity extends AppCompatActivity
        implements CompoundButton.OnCheckedChangeListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_check_box);
        // 从布局文件中获取名为ck_system的复选框
        CheckBox ck_system = findViewById(R.id.ck_system);
        // 设置勾选监听器，一旦点击复选框，就触发监听器的onCheckedChanged方法
        ck_system.setOnCheckedChangeListener(this);
```

```
    }

    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        String desc = String.format("您%s了这个CheckBox",
                                    isChecked ? "勾选" : "取消勾选");
        buttonView.setText(desc);
    }
}
```

然后运行测试App，一开始的演示界面如图5-13所示，此时复选框默认未勾选。首次点击复选框，此时复选框的图标及文字均发生变化，如图5-14所示；再次点击复选框，此时复选框的图标及文字又发生变化，如图5-15所示；可见先后触发了勾选与取消勾选事件。



图 5-13　初始的复选框界面

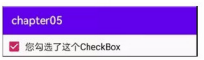


图 5-14　首次点击后的复选框



图 5-15　再次点击后的复选框

5.2.2　开关按钮 Switch

Switch是开关按钮，它像一个高级版本的CheckBox，在选中与取消选中时可展现的界面元素比复选框丰富。Switch控件新添加的XML属性说明如下：

- textOn：**设置右侧开启时的文本。**
- textOff：**设置左侧关闭时的文本。**
- track：**设置开关轨道的背景。**
- thumb：**设置开关标识的图标。**

虽然开关按钮是升级版的复选框，但它在实际开发中用得不多。原因之一是大家觉得Switch的默认界面不够大气，如图5-16和图5-17所示，小巧的开关图标显得有些拘谨；原因之二是大家觉得iPhone的界面很漂亮，无论用户还是客户，都希望App实现iOS那样的控件风格，于是iOS的开关按钮UISwitch就成了安卓开发者仿照的对象。



图 5-16　Switch 控件的“关”状态



图 5-17　Switch 控件的“开”状态

现在要让Android实现类似iOS的开关按钮，主要思路是借助状态列表图形，首先创建一个图形专用的XML文件，给状态列表指定选中与未选中时的开关图标，代码如下：

```
(完整代码见chapter05\src\main\res\drawable\switch selector.xml)
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_checked="true"
          android:drawable="@drawable/switch_on" />
    <item android:drawable="@drawable/switch off" />
</selector>
```

然后把CheckBox标签的background属性设置为@drawable/switch_selector，同时将button属性设置为@null。完整的CheckBox标签内容示例如下：

```
(完整代码见chapter05\src\main\res\layout\activity_switch_ios.xml)
<CheckBox
    android:id="@+id/ck_status"
    android:layout_width="60dp"
    android:layout_height="30dp"
    android:background="@drawable/switch_selector"
    android:button="@null" />
```

为什么这里修改background属性，而不直接修改button属性呢？因为button属性有局限，无论多大的图片，都只显示一个小小的图标，可是小小的图标一点都不大气，所以这里必须使用background属性，要它有多大就能有多大，这样才够炫够酷。

最后看看这个仿iOS开关按钮的效果，分别如图5-18和图5-19所示。这下开关按钮脱胎换骨，又圆又鲜艳，比原来的Switch好看了很多。



图 5-18　仿 iOS 按钮的“关”状态



图 5-19　仿 iOS 按钮的“开”状态

5.2.3　单选按钮 RadioButton

所谓单选按钮，指的是在一组按钮中选择其中一项，并且不能多选，这要求有个容器确定这组按钮的范围，这个容器便是单选组RadioGroup。单选组实质上是个布局，同一组RadioButton都要放在同一个RadioGroup节点下。RadioGroup提供了orientation属性指定下级控件的排列方向，该属性为horizontal时，单选按钮在水平方向排列；该属性为vertical时，单选按钮在垂直方向排列。RadioGroup下面除了RadioButton，还可以挂载其他子控件（如TextView、ImageView等）。如此看来，单选组相当于特殊的线性布局，它们主要有以下两个区别：

（1）单选组多了管理单选按钮的功能，而线性布局不具备该功能。

（2）如果不指定orientation属性，那么单选组默认垂直排列，而线性布局默认水平排列。

下面是RadioGroup在Java代码中的3个常用方法。

- check：**选中指定资源编号的单选按钮。**
- getCheckedRadioButtonId：**获取已选中单选按钮的资源编号。**
- setOnCheckedChangeListener：**设置单选按钮勾选变化的监听器。**

与CheckBox不同的是，RadioButton默认未选中，点击后显示选中，但是再次点击不会取消选中。只有点击同组的其他单选按钮时，原来选中的单选按钮才会取消选中。另需注意，单选按钮的选中事件不是由RadioButton处理，而是由RadioGroup处理。

接下来演示单选按钮的操作过程，首先编写活动页面的XML文件，代码如下：

```
(完整代码见chapter05\src\main\res\layout\ activity_radio_horizontal.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:text="请选择您的性别" />
    <RadioGroup
        android:id="@+id/rg_sex"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <RadioButton
            android:id="@+id/rb_male"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="男" />
        <RadioButton
            android:id="@+id/rb_female"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="女" />
    </RadioGroup>
    <TextView
        android:id="@+id/tv_sex"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

接着编写对应的Java代码，主要是如何处理选中监听器，具体代码如下：

```
// 该页面实现了接口OnCheckedChangeListener，要重写监听器的onCheckedChanged方法
public class RadioHorizontalActivity extends AppCompatActivity
        implements RadioGroup.OnCheckedChangeListener {
    private TextView tv_sex;  // 声明一个文本视图对象

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_radio_horizontal);
        // 从布局文件中获取名为tv_sex的文本视图
        tv_sex = findViewById(R.id.tv_sex);
        // 从布局文件中获取名为rg_sex的单选组
        RadioGroup rg_sex = findViewById(R.id.rg_sex);
        // 设置单选监听器，一旦点击组内的单选按钮，就触发监听器的onCheckedChanged方法
        rg_sex.setOnCheckedChangeListener(this);
    }

    // 在用户点击组内的单选按钮时触发
    @Override
    public void onCheckedChanged(RadioGroup group, int checkedId) {
        if (checkedId == R.id.rb_male) {
            tv_sex.setText("哇哦，你是个帅气的男孩");
        } else if (checkedId == R.id.rb_female) {
            tv_sex.setText("哇哦，你是个漂亮的女孩");
        }
    }
}
```

然后运行测试App，一开始的演示界面如图5-20所示，此时两个单选按钮均未选中。先点击左边的单选按钮，此时左边按钮显示选中状态，如图5-21所示；再点击右边的单选按钮，此时右边按钮显示选中状态，同时左边按钮取消选中，如图5-22所示；可见果然实现了组内只能选中唯一按钮的单选功能。

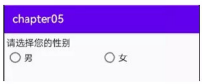


图 5-20　初始的单选按钮界面



图 5-21　选中左边按钮的单选界面



图 5-22　选中右边按钮的单选界面

5.3　文本输入

本节介绍如何在编辑框EditText上高效地输入文本，包括：如何改变编辑框的控件外观，如何利用焦点变更监听器提前校验输入位数，如何利用文本变化监听器自动关闭软键盘。

5.3.1　编辑框 EditText

编辑框EditText用于接收软键盘输入的文字，例如用户名、密码、评价内容等，它由文本视图派生而来，除了TextView已有的各种属性和方法之外，EditText还支持下列XML属性。

- inputType：**指定输入的文本类型。输入类型的取值说明见表5-4，若同时使用多种文本类型，则可使用竖线（|）把多种文本类型拼接起来。**
- maxLength：**指定文本允许输入的最大长度。**
- hint：**指定提示文本的内容。**
- textColorHint：**指定提示文本的颜色。**

表 5-4　输入类型的取值说明

输入类型	说　明
text	文本
textPassword	文本密码。显示时用圆点（•）代替
number	整型数
numberSigned	带符号的数字。允许在开头带负号（－）
numberDecimal	带小数点的数字
numberPassword	数字密码。显示时用圆点（•）代替
datetime	时间日期格式。除了数字外，还允许输入横线（-）、斜杆（/）、空格（　）、冒号（:）
date	日期格式。除了数字外，还允许输入横线（-）和斜杆（/）
time	时间格式。除了数字外，还允许输入冒号（:）

接下来通过XML布局观看编辑框界面效果，演示用的XML文件内容如下：

```
(完整代码见chapter05\src\main\res\layout\activity edit simple.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:text="下面是登录信息" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text"
        android:maxLength="10"
        android:hint="请输入用户名" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPassword"
        android:maxLength="8"
        android:hint="请输入密码" />
</LinearLayout>
```

运行测试App，进入初始的编辑框页面，如图5-23所示。然后往用户名编辑框输入文字，输满10个字后发现不能再输入，于是切换到密码框继续输入，直到输满8位密码，此时编辑框页面如图5-24所示。

根据以上图示可知编辑框的各属性正常工作。不过编辑框有根下划线，未输入时显示灰色，正在输入时显示红色，这种效果是怎么实现的呢？其实下划线没用到新属性，而用了已有的背景属性background；至于未输入与正在输入两种情况的颜色差异，乃是因为使用了状态列表图形，编辑框获得焦点时（正在输入）显示红色的下划线，其余时候显示灰色下划线。当然EditText默认的下划线背景不甚好看，下面将利用状态列表图形将编辑框背景改为更加美观的圆角矩形。



图 5-23　初始的编辑框样式



图 5-24　输入文字的编辑框样式

首先编写圆角矩形的形状图形文件，它的XML定义文件示例如下：

```
(完整代码见chapter05\src\main\res\drawable\shape_edit_normal.xml)
<shape xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- 指定了形状内部的填充颜色 -->
    <solid android:color="#ffffff" />
    <!-- 指定了形状轮廓的粗细与颜色 -->
    <stroke
        android:width="1dp"
        android:color="#aaaaaa" />
    <!-- 指定了形状4个圆角的半径 -->
    <corners android:radius="5dp" />
    <!-- 指定了形状4个方向的间距 -->
    <padding
        android:bottom="2dp"
        android:left="2dp"
        android:right="2dp"
        android:top="2dp" />
</shape>
```

上述的shape_edit_normal.xml定义了一个灰色的圆角矩形，可在未输入时展示该形状。正在输入时候的形状要改为蓝色的圆角矩形，其中轮廓线条的色值从aaaaaa（灰色）改成0000ff（蓝色），具体定义放在shape_edit_focus.xml。

接着编写编辑框背景的状态列表图形文件，主要在selector节点下添加两个item：一个item设置了获得焦点时（android:state_focused="true"）的图形为@drawable/shape_edit_focus；另一个item设置了图形@drawable/shape_edit_normal但未指定任何状态，表示其他情况都展示该图形。完整的状态列表图形定义示例如下：

```
(完整代码见chapter05\src\main\res\drawable\editext selector.xml)
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_focused="true"
          android:drawable="@drawable/shape_edit_focus" />
    <item android:drawable="@drawable/shape edit normal" />
</selector>
```

然后编写测试页面的XML布局文件，一共添加3个EditText标签：第一个EditText采用默认的编辑框背景；第二个EditText将background属性值设为@null，此时编辑框不显示任何背景；第三个EditText将background属性值设为@drawable/editext_selector，其背景由editext_selector.xml所定义的状态列表图形决定。详细的XML文件内容如下：

```
(完整代码见chapter05\src\res\layout\activity edit border.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout height="match parent"
    android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text"
        android:hint="这是默认边框" />
    <EditText
        android:layout width="match parent"
        android:layout_height="wrap_content"
        android:inputType="text"
        android:hint="我的边框不见了"
        android:background="@null" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text"
        android:hint="我的边框是圆角"
        android:background="@drawable/editext selector" />
</LinearLayout>
```

最后运行测试App，更换背景之后的编辑框界面如图5-25所示，可见第三个编辑框的背景成功变为了圆角矩形边框。



图 5-25　更换背景后的编辑框样式

5.3.2　焦点变更监听器

虽然编辑框EditText提供了maxLength属性，用来设置可输入文本的最大长度，但是它没提供对应的minLength属性，也就无法设置可输入文本的最小长度。譬如手机号码为固定的11位数字，用户必须输满11位才是合法的，然而编辑框不会自动检查手机号码是否达到11位，即使用户少输入一位，编辑框依然认为这是合法的手机号。比如图5-26所示的登录页面，有手机号码编辑框，有密码编辑框，还有登录按钮。



图 5-26　简单的登录界面

既然编辑框不会自动校验手机号是否达到11位，势必要求代码另行检查。一种想法是在用户点击“登录”按钮时再判断，不过通常此时已经输完手机号与密码了。为什么不能在输入密码之前就判断手机号码的位数呢？早点检查可以帮助用户早点发现错误，特别是表单元素较多的时候，更能改善用户的使用体验。就上面的登录例子而言，手机号编辑框下方为密码框，那么能否给密码框注册点击事件，以便在用户准备输入密码时就校验手机号的位数呢？

然而实际运行App却发现，先输入手机号码再输入密码，一开始并不会触发密码框的点击事件，再次点击密码框才会触发点击事件。缘由是编辑框比较特殊，要点击两次后才会触发点击事件，因为第一次点击只触发焦点变更事件，第二次点击才触发点击事件。编辑框的焦点，直观上就看那个闪动的光标，哪个编辑框有光标，焦点就落在哪里。光标在编辑框之间切换，便产生了焦点变更事件，所以对于编辑框来说，应当注册焦点变更监听器，而非注册点击监听器。

焦点变更监听器来自接口View.OnFocusChangeListener，若想注册该监听器，就要调用编辑框对象的setOnFocusChangeListener方法，即可在光标切换之时（获得光标和失去光标）触发焦点变更事件。下面是给密码框注册焦点变更监听器的代码例子：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\EditFocusActivity.java)
// 从布局文件中获取名为et password的编辑框
EditText et_password = findViewById(R.id.et_password);
// 给编辑框注册一个焦点变化监听器，一旦焦点发生变化，就触发监听器的onFocusChange方法
 et_password.setOnFocusChangeListener(this);
```

以上代码把焦点变更监听器设置到当前页面，则还需让活动页面实现接口View.OnFocusChangeListener，并重写该接口定义的onFocusChange方法，判断如果是密码框获得焦点，就检查输入的手机号码是否达到11位。具体的焦点变更处理方法如下：

```
// 焦点变更事件的处理方法，hasFocus表示当前控件是否获得焦点
// 为什么光标进入事件不选onClick？因为要点两下才会触发onClick（第一下是切换焦点动作）
@Override
public void onFocusChange(View v, boolean hasFocus) {
    // 判断密码编辑框是否获得焦点。hasFocus为true表示获得焦点，为false表示失去焦点
    if (v.getId()==R.id.et_password && hasFocus) {
        String phone = et_phone.getText().toString();
        if (TextUtils.isEmpty(phone) || phone.length()<11) {  // 手机号不足11位
            // 手机号码编辑框请求焦点，也就是把光标移回手机号码编辑框
            et_phone.requestFocus();
            Toast.makeText(this, "请输入11位手机号码", Toast.LENGTH_SHORT).show();
        }
    }
}
```

改好代码重新运行App，当手机号不足11位时点击密码框，界面底部果然弹出了相应的提示文字，如图5-27所示，并且光标仍然留在手机号码编辑框，说明首次点击密码框的确触发了焦点变更事件。



图 5-27 编辑框触发了焦点变更监听器

5.3.3 文本变化监听器

输入法的软键盘往往会遮住页面下半部分，使得“登录”“确认”“下一步”等按钮看不到了，用户若想点击这些按钮还得再点一次返回键才能关闭软键盘。为了方便用户操作，最好在满足特定条件时自动关闭软键

盘，比如手机号码输入满11位后自动关闭软键盘，又如密码输入满6位后自动关闭软键盘，等等。达到指定位数便自动关闭键盘的功能，可以再分解为两个独立的功能点：一个是如何关闭软键盘，另一个是如何判断已输入的文字是否达到指定位数。分别说明如下。

1. 如何关闭软键盘

诚然按下返回键就会关闭软键盘，但这是系统自己关闭的，而非开发者在代码中关闭的。因为输入法软键盘由系统服务INPUT_METHOD_SERVICE管理，所以关闭软键盘也要由该服务处理，下面是使用系统服务关闭软键盘的代码例子：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\util\ViewUtil.java)
public static void hideOneInputMethod(Activity act, View v) {
    // 从系统服务中获取输入法管理器
    InputMethodManager imm = (InputMethodManager)
            act.getSystemService(Context.INPUT METHOD SERVICE);
    // 关闭屏幕上的输入法软键盘
    imm.hideSoftInputFromWindow(v.getWindowToken(), 0);
}
```

注意上述代码里面的视图对象v，虽然控件类型为View，但它必须是EditText类型才能正常关闭软键盘。

2. 如何判断已输入的文字是否达到指定位数

该功能点要求实时监控当前已输入的文本长度，这个监控操作用到文本监听器接口TextWatcher，该接口提供了3个监控方法，具体说明如下：

- beforeTextChanged：**在文本改变之前触发**。
- onTextChanged：**在文本改变过程中触发**。
- afterTextChanged：**在文本改变之后触发**。

具体到编码实现，需要自己写个监听器实现接口TextWatcher，再调用编辑框对象的addTextChangedListener方法注册文本监听器。监听操作建议在afterTextChanged方法中完成，如果同时监听11位的手机号码和6位的密码，一旦输入文字达到指定长度就关闭软键盘，则详细的监听器代码如下：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\EditHideActivity.java)
// 定义一个编辑框监听器，在输入文本达到指定长度时自动隐藏输入法软键盘
private class HideTextWatcher implements TextWatcher {
    private EditText mView;  // 声明一个编辑框对象
    private int mMaxLength;  // 声明一个最大长度变量

    public HideTextWatcher(EditText v, int maxLength) {
        super();
        mView = v;
        mMaxLength = maxLength;
    }

    // 在编辑框的输入文本变化前触发
    public void beforeTextChanged(CharSequence s, int start, int count, int after) {}

    // 在编辑框的输入文本变化时触发
    public void onTextChanged(CharSequence s, int start, int before, int count) {}
```

```
        // 在编辑框的输入文本变化后触发
        public void afterTextChanged(Editable s) {
            String str = s.toString();  // 获得已输入的文本字符串
            // 输入文本达到11位（如手机号码），或者达到6位（如登录密码）时关闭输入法软键盘
            if ((str.length() == 11 && mMaxLength == 11)
                || (str.length() == 6 && mMaxLength == 6)) {
                // 隐藏输入法软键盘
                ViewUtil.hideOneInputMethod(EditHideActivity.this, mView);
            }
        }
    }
```

写好文本监听器代码，还要给手机号码编辑框和密码编辑框分别注册监听器，注册代码示例如下：

```
// 从布局文件中获取名为et phone的手机号码编辑框
EditText et_phone = findViewById(R.id.et_phone);
// 从布局文件中获取名为et_password的密码编辑框
EditText et password = findViewById(R.id.et password);
// 给手机号码编辑框添加文本变化监听器
et phone.addTextChangedListener(new HideTextWatcher(et phone, 11));
// 给密码编辑框添加文本变化监听器
et_password.addTextChangedListener(new HideTextWatcher(et_password, 6));
```

然后运行测试App，先输入手机号码的前10位，因为还没达到11位，所以软键盘依然展示，如图5-28所示。接着输入最后一位手机号，总长度达到11位，于是软键盘自动关闭，如图5-29所示。



图 5-28　输入 10 位手机号码



图 5-29　输入 11 位手机号码

5.4　对　话　框

本节介绍几种常用的对话框控件，包括：如何使用提醒对话框处理不同的选项，如何使用日期对话框获取用户选择的日期，如何使用时间对话框获取用户选择的时间。

5.4.1　提醒对话框 AlertDialog

AlertDialog名为提醒对话框，它是Android中最常用的对话框，可以完成常见的交互操作，例如提示、确认、选择等功能。由于AlertDialog没有公开的构造方法，因此必须借助建造器

AlertDialog.Builder才能完成参数设置，AlertDialog.Builder的常用方法说明如下。

- setIcon：**设置对话框的标题图标。**
- setTitle：**设置对话框的标题文本。**
- setMessage：**设置对话框的内容文本。**
- setPositiveButton：**设置肯定按钮的信息，包括按钮文本和点击监听器。**
- setNegativeButton：**设置否定按钮的信息，包括按钮文本和点击监听器。**
- setNeutralButton：**设置中性按钮的信息，包括按钮文本和点击监听器，该方法比较少用。**

通过AlertDialog.Builder设置完对话框参数，还需调用建造器的create方法才能生成对话框实例。最后调用对话框实例的show方法，在页面上弹出提醒对话框。

下面是构建并显示提醒对话框的Java代码例子：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\AlertDialogActivity.java)
// 创建提醒对话框的建造器
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("尊敬的用户");                    // 设置对话框的标题文本
builder.setMessage("你真的要卸载我吗？");          // 设置对话框的内容文本
// 设置对话框的肯定按钮文本及其点击监听器
builder.setPositiveButton("残忍卸载", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        tv_alert.setText("虽然依依不舍，但是只能离开了");
    }
});
// 设置对话框的否定按钮文本及其点击监听器
builder.setNegativeButton("我再想想", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        tv_alert.setText("让我再陪你三百六十五个日夜");
    }
});
AlertDialog alert = builder.create();              // 根据建造器构建提醒对话框对象
alert.show();                                      // 显示提醒对话框
```

提醒对话框的弹窗效果如图5-30所示，可见该对话框有标题和内容，还有两个按钮。

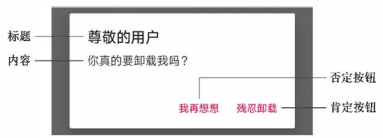


图 5-30　提醒对话框的弹窗效果

点击不同的对话框按钮会触发不同的处理逻辑。例如，图5-31为点击“我再想想”按钮后的页面，图5-32为点击“残忍卸载”按钮后的页面。

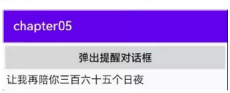


图 5-31　点击“我再想想”按钮后的页面

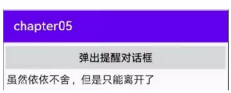


图 5-32　点击“残忍卸载”按钮后的页面

5.4.2　日期对话框 DatePickerDialog

虽然EditText提供了inputType="date"的日期输入，但是很少有人会手工输入完整日期，况且EditText还不支持“****年**月**日”这样的中文日期，所以系统提供了专门的日期选择器DatePicker，供用户选择具体的年、月、日。不过，DatePicker并非弹窗模式，而是在当前页面占据一块区域，并且不会自动关闭。按习惯来说，日期控件应该弹出对话框，选择完日期就要自动关闭对话框。因此，很少直接在界面上显示DatePicker，而是利用已经封装好的日期选择对话框DatePickerDialog。

DatePickerDialog相当于在AlertDialog上装载了DatePicker，编码时只需调用构造方法设置当前的年、月、日，然后调用show方法即可弹出日期对话框。日期选择事件则由监听器OnDateSetListener负责响应，在该监听器的onDateSet方法中，开发者获取用户选择的具体日期，再做后续处理。特别注意onDateSet的月份参数，它的起始值不是1而是0。也就是说，一月份对应的参数值为0，十二月份对应的参数值为11，中间月份的数值以此类推。

在界面上内嵌显示DatePicker的效果如图5-33所示，其中，年、月、日通过上下滑动选择。单独弹出日期对话框的效果如图5-34所示，其中年、月、日按照日历风格展示。



图 5-33　日期选择器的截图



图 5-34　日期对话框的截图

下面是使用日期对话框的Java代码例子，包括弹出日期对话框和处理日期监听事件：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\DatePickerActivity.java)
// 该页面类实现了接口OnDateSetListener，意味着要重写日期监听器的onDateSet方法
public class DatePickerActivity extends AppCompatActivity implements
        View.OnClickListener, DatePickerDialog.OnDateSetListener {
    private TextView tv_date;  // 声明一个文本视图对象

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_date_picker);
        tv_date = findViewById(R.id.tv_date);
        findViewById(R.id.btn_date).setOnClickListener(this);
    }
```

```
    @Override
    public void onClick(View v) {
        if (v.getId() == R.id.btn_date) {
            // 获取日历的一个实例，里面包含了当前的年月日
            Calendar calendar = Calendar.getInstance();
            // 构建一个日期对话框，该对话框已经集成了日期选择器。
            // DatePickerDialog的第二个构造参数指定了日期监听器
            DatePickerDialog dialog = new DatePickerDialog(this, this,
                    calendar.get(Calendar.YEAR),            // 年份
                    calendar.get(Calendar.MONTH),           // 月份
                    calendar.get(Calendar.DAY_OF_MONTH));   // 日期
            dialog.show();                                  // 显示日期对话框
        }
    }

    // 一旦点击日期对话框上的确定按钮，就会触发监听器的onDateSet方法
    @Override
    public void onDateSet(DatePicker view, int year, int monthOfYear, int dayOfMonth)
{
        // 获取日期对话框设定的年份、月份
        String desc = String.format("您选择的日期是%d年%d月%d日",
                year, monthOfYear + 1, dayOfMonth);
        tv_date.setText(desc);
    }
}
```

5.4.3　时间对话框 TimePickerDialog

既然有了日期选择器，还得有对应的时间选择器。同样，在实际开发中也很少直接用TimePicker，而是用封装好的时间选择对话框TimePickerDialog。该对话框的用法类似DatePickerDialog，不同之处主要有两个：

（1）构造方法传入的是当前的小时与分钟，最后一个参数表示是否采取24小时制，一般为true表示小时的数值范围为0～23；若为false则表示采取12小时制。

（2）时间选择监听器为OnTimeSetListener，对应需要实现onTimeSet方法，在该方法中可获得用户选择的小时和分钟。

在界面上内嵌显示TimePicker的效果如图5-35所示，其中，小时与分钟可通过上下滑动选择。单独弹出时间对话框的效果如图5-36所示，其中小时与分钟按照钟表风格展示。



图 5-35　时间选择器的截图

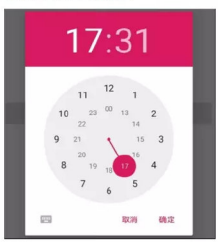


图 5-36　时间对话框的截图

下面是使用时间对话框的Java代码例子，包括弹出时间对话框和处理时间监听事件：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\TimePickerActivity.java)
// 该页面类实现了接口OnTimeSetListener，意味着要重写时间监听器的onTimeSet方法
public class TimePickerActivity extends AppCompatActivity implements
        View.OnClickListener, TimePickerDialog.OnTimeSetListener {
    private TextView tv_time;  // 声明一个文本视图对象

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_time_picker);
        tv_time = findViewById(R.id.tv_time);
        findViewById(R.id.btn_time).setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        if (v.getId() == R.id.btn_time) {
            // 获取日历的一个实例，里面包含了当前的时、分、秒
            Calendar calendar = Calendar.getInstance();
            // 构建一个时间对话框，该对话框已经集成了时间选择器
            // TimePickerDialog的第二个构造参数指定了时间监听器
            TimePickerDialog dialog = new TimePickerDialog(this, this,
                    calendar.get(Calendar.HOUR_OF_DAY),          // 小时
                    calendar.get(Calendar.MINUTE),               // 分钟
                    true);                              // true表示24小时制，false表示12小时制
            dialog.show();                              // 显示时间对话框
        }
    }

    // 一旦点击时间对话框上的确定按钮，就会触发监听器的onTimeSet方法
    @Override
    public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
        // 获取时间对话框设定的小时和分钟
        String desc = String.format("您选择的时间是%d时%d分", hourOfDay, minute);
        tv_time.setText(desc);
    }
}
```

5.5 实战项目：找回密码

在移动互联网时代，用户是每家IT企业最宝贵的资源，对于App而言，吸引用户注册并登录是万分紧要之事，因为用户登录之后才有机会产生商品交易。登录校验通常是用户名+密码组合，可是每天总有部分用户忘记密码，为此要求App提供找回密码的功能。如何简化密码找回步骤，同时兼顾安全性，就是一个值得认真思考的问题。

5.5.1 需求描述

各家电商App的登录页面大同小异，要么是用户名与密码组合登录，要么是手机号码与验证码组合登录，若是做好一点的，还会提供找回密码与记住密码等功能。先来看一下登录页面是什么样，因为有两种组合登录方式，所以登录页面也分成两个效果图。如图5-37所示，这是选中密码登录时的界面；如图5-38所示，这是选中验证码登录时的界面。



图 5-37　选中密码登录方式时的界面



图 5-38　选中验证码登录时的界面

从以上两个登录效果图可以看到，密码登录与验证码登录的界面主要存在以下几点区别：

（1）密码输入框和验证码输入框的左侧标题以及输入框内部的提示语各不相同。

（2）如果是密码登录，则需要支持找回密码；如果是验证码登录，则需要支持向用户手机发送验证码。

（3）密码登录可以提供记住密码功能，而验证码的数值每次都不一样，无须也没法记住验证码。

对于找回密码功能，一般直接跳转到找回密码页面，在该页面输入和确认新密码，并校验找回密码的合法性（通过短信验证码检查），据此勾勒出密码找回页面的轮廓概貌，如图5-39所示。



图 5-39　找回密码的界面效果

在找回密码的操作过程当中，为了更好地增强用户体验，有必要在几个关键节点处提醒用户。比如成功发送验证码之后，要及时提示用户注意查收短信，这里暂且做成提醒对话框的形式，如图5-40所示。又比如密码登录成功之后，也要告知用户已经修改成功登录，注意继续后面的操作，登录成功的提示弹窗如图5-41所示。

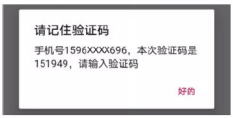


图 5-40　发送验证码的提醒对话框

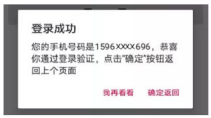


图 5-41　登录成功后的提醒对话框

真是想不到，原来简简单单的一个登录功能，就得考虑这么多的需求场景。可是仔细想想，这些需求场景都是必要的，其目的是为了让用户能够更加便捷地顺利登录。正所谓“台上十分钟，台下十年功”，每个好用的App背后，都离不开开发者十年如一日的辛勤工作。

5.5.2　界面设计

用户登录与找回密码界面看似简单，用到的控件却不少。按照之前的界面效果图，大致从上到下、从左到右分布着下列Android控件：

- **单选按钮**RadioButton：**用来区分是密码登录还是验证码登录。**
- **文本视图**TextView：**输入框左侧要显示此处应该输入什么信息。**

- **编辑框**EditText：**用来输入手机号码、密码和验证码。**
- **复选框**CheckBox：**用于判断是否记住密码。**
- **按钮**Button：**除了“登录”按钮，还有“忘记密码”和“获取验证码”两个按钮。**
- **线性布局**LinearLayout：**整体界面从上往下排列，用到了垂直方向的线性布局。**
- **相对布局**RelativeLayout：**忘记密码的按钮与密码输入框是叠加的，且“忘记密码”与上级视图右对齐。**
- **单选组**RadioGroup：**密码登录和验证码登录这两个单选按钮，需要放在单选组之中。**
- **提醒对话框**AlertDialog：**为了演示方便，获取验证码与登录成功都通过提醒对话框向用户反馈结果。**

另外，由于整个登录模块由登录页面和找回密码页面组成，因此这两个页面之间需要进行数据交互，也就是在页面跳转之时传递参数。譬如，从登录页面跳转到找回密码页面，要携带唯一标识的手机号码作为请求参数，不然密码找回页面不知道要给哪个手机号码修改密码。同时，从找回密码页面回到登录页面，也要将修改之后的新密码作为应答参数传回去，否则登录页面不知道密码被改成什么了。

5.5.3　关键代码

为了方便读者更好更快地完成登录页面与找回密码页面，下面列举几个重要功能的代码片段。

1. 关于自动清空错误的密码

这里有个细微的用户体验问题：用户会去找回密码，肯定是发现输入的密码不对；那么修改密码后回到登录页面，如果密码框里还是刚才的错误密码，用户只能先清空错误密码，然后才能输入新密码。一个App要想让用户觉得好用，就得急用户之所急，想用户之所想，像刚才那个错误密码的情况，应当由App在返回登录页面时自动清空原来的错误密码。

自动清空密码框的操作，放在onActivityResult方法中处理是个办法，但这样有个问题，如果用户直接按返回键回到登录页面，那么onActivityResult方法发现数据为空便不做处理。因此应该这么处理：判断当前是否为返回页面动作，只要是从找回密码页面返回到当前页面，则不管是否携带应答参数，都要自动清空密码输入框。对应的Java代码则为重写登录页面的onRestart方法，在该方法中强制清空密码。这样一来，不管用户是修改密码完成回到登录页面，还是点击返回键回到登录页面，App都会自动清空密码框了。

下面是重写onRestart方法之后的代码例子：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\LoginMainActivity.java)
// 从修改密码页面返回登录页面，要清空密码的输入框
@Override
protected void onRestart() {
    super.onRestart();
    et_password.setText("");
}
```

2. 关于自动隐藏输入法面板

在输入手机号码或者密码的时候，屏幕下方都会弹出输入法面板，供用户按键输入数字和字母。但是输入法面板往往占据屏幕下方大块空间，很是碍手碍脚，用户输入完11位的手机号码时，

还得再按一下返回键来关闭输入法面板，接着才能继续输入密码。理想的做法是：一旦用户输完11位手机号码，App就要自动隐藏输入法面板。同理，一旦用户输完6位密码或者6位验证码，App也要自动隐藏输入法面板。要想让App具备这种智能的判断功能，就得给文本编辑框添加监听器，只要当前编辑框输入文本长度达到11位或者和6位，App就自动隐藏输入法面板。

下面是实现自动隐藏软键盘的监听器代码例子：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\LoginMainActivity.java)
// 定义一个编辑框监听器，在输入文本达到指定长度时自动隐藏输入法软键盘
private class HideTextWatcher implements TextWatcher {
    private EditText mView;  // 声明一个编辑框对象
    private int mMaxLength;  // 声明一个最大长度变量

    public HideTextWatcher(EditText v, int maxLength) {
        super();
        mView = v;
        mMaxLength = maxLength;
    }

    // 在编辑框的输入文本变化前触发
    public void beforeTextChanged(CharSequence s, int start, int count, int after) {}

    // 在编辑框的输入文本变化时触发
    public void onTextChanged(CharSequence s, int start, int before, int count) {}

    // 在编辑框的输入文本变化后触发
    public void afterTextChanged(Editable s) {
        String str = s.toString();  // 获得已输入的文本字符串
        // 输入文本达到11位（如手机号码），或者达到6位（如登录密码）时关闭输入法软键盘
        if ((str.length() == 11 && mMaxLength == 11)
                || (str.length() == 6 && mMaxLength == 6)) {
            // 隐藏输入法软键盘
            ViewUtil.hideOneInputMethod(LoginMainActivity.this, mView);
        }
    }
}
```

3. 关于密码修改的校验操作

由于密码对于用户来说是很重要的信息，因此必须认真校验新密码的合法性，务必做到万无一失。具体的密码修改校验可分作下列4个步骤：

（1）新密码和确认输入的新密码都要是6位数字。

（2）新密码和确认输入的新密码必须保持一致。

（3）用户输入的验证码必须和系统下发的验证码一致。

（4）密码修改成功，携带修改后的新密码返回登录页面。

根据以上的校验步骤，对应的代码逻辑示例如下：

```
(完整代码见chapter05\src\main\java\com\example\chapter05\LoginForgetActivity.java)
String password_first = et_password_first.getText().toString();
String password_second = et_password_second.getText().toString();
if (password_first.length() < 6 || password_second.length() < 6) {
    Toast.makeText(this, "请输入正确的新密码", Toast.LENGTH_SHORT).show();
    return;
}
if (!password_first.equals(password_second)) {
```

```
        Toast.makeText(this, "两次输入的新密码不一致", Toast.LENGTH_SHORT).show();
        return;
    }
    if (!et_verifycode.getText().toString().equals(mVerifyCode)) {
        Toast.makeText(this, "请输入正确的验证码", Toast.LENGTH_SHORT).show();
    } else {
        Toast.makeText(this, "密码修改成功", Toast.LENGTH_SHORT).show();
        // 以下把修改好的新密码返回给上一个页面
        Intent intent = new Intent();  // 创建一个新意图
        intent.putExtra("new_password", password_first);  // 存入新密码
        setResult(Activity.RESULT_OK, intent);  // 携带意图返回上一个页面
        finish();  // 结束当前的活动页面
    }
```

5.6 小 结

本章主要介绍了App开发的中级控件的相关知识，包括：定制简单的图形（图形的基本概念、形状图形、九宫格图片、状态列表图形）、操纵几种选择按钮（复选框CheckBox、开关按钮Switch、单选按钮RadioButton）、高效地输入文本（编辑框EditText、焦点变更监听器、文本变化监听器）、获取对话框的选择结果（提醒对话框AlertDialog、日期对话框DatePickerDialog、时间对话框TimePickerDialog）。最后设计了一个实战项目“找回密码”，在该项目的App编码中用到了前面介绍的大部分控件，从而加深了对所学知识的理解。

通过本章的学习，读者应该能掌握以下4种开发技能：

（1）学会定制几种简单的图形。

（2）学会操纵常见的选择按钮。

（3）学会高效且合法地输入文本。

（4）学会通过对话框获取用户选项。

5.7 动手练习

请上机实验本章的找回密码项目，其中登录操作支持“用户名+密码”和“手机号+验证码”两种方式，同时支持通过验证码重置密码。

第 6 章

数 据 存 储

本章介绍Android的4种存储方式的用法，包括键值对方式（共享参数和数据仓库）、数据库SQLite、存储卡文件、App的全局内存；另外介绍Android重要组件——Application的基本概念与常见用法，以及如何利用内容组件在应用之间共享数据。最后，结合本章所学的知识演示实战项目“购物车”的设计与实现。

6.1 键 值 对

本节介绍Android的键值对存储方式的使用方法，包括：如何将数据保存到共享参数，如何从共享参数读取数据，如何使用共享参数实现登录页面的记住密码功能，如何使用Jetpack集成的数据仓库。

6.1.1 共享参数的用法

SharedPreferences是Android的一个轻量级存储工具，它采用的存储结构是Key-Value的键值对方式，类似于Java的Properties，二者都是把Key-Value的键值对保存在配置文件中。不同的是，Properties的文件内容形如Key=Value，而SharedPreferences的存储介质是XML文件，且以XML标记保存键值对。保存共享参数键值对信息的文件路径为：/data/data/应用包名/shared_prefs/文件名.xml。下面是一个共享参数的XML文件例子：

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="name">Mr Lee</string>
    <int name="age" value="30" />
    <boolean name="married" value="true" />
    <float name="weight" value="100.0" />
</map>
```

基于XML格式的特点，共享参数主要用于如下场合：

（1）简单且孤立的数据。若是复杂且相互关联的数据，则要保存在关系数据库中。

（2）文本形式的数据。若是二进制数据，则要保存至文件。

（3）需要持久化存储的数据。App退出后再次启动时，之前保存的数据仍然有效。

在实际开发中，共享参数经常存储的数据包括：App的个性化配置信息、用户使用App的行为信息、临时需要保存的片段信息等。

共享参数对数据的存储和读取操作类似于Map，也有存储数据的put方法，以及读取数据的get方法。调用getSharedPreferences方法可以获得共享参数实例，获取代码示例如下：

```
// 从share.xml获取共享参数实例
SharedPreferences shared = getSharedPreferences("share", MODE_PRIVATE);
```

由以上代码可知，getSharedPreferences方法的第一个参数是文件名，填share表示共享参数的文件名是share.xml；第二个参数是操作模式，填MODE_PRIVATE表示私有模式。

往共享参数存储数据要借助于Editor类，保存数据的代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\ShareWriteActivity.java)
SharedPreferences.Editor editor = shared.edit();     // 获得编辑器的对象
editor.putString("name", "Mr Lee");                  // 添加一个名为name的字符串参数
editor.putInt("age", 30);                            // 添加一个名为age的整型参数
editor.putBoolean("married", true);                  // 添加一个名为married的布尔型参数
editor.putFloat("weight", 100f);                     // 添加一个名为weight的浮点数参数
editor.commit();                                     // 提交编辑器中的修改
```

注意上述代码采用了commit方法提交修改，该方法会把数据直接写入磁盘。如果想要更好的性能，可将commit方法改为apply方法，该方法的提交操作会先将数据写入内存，然后异步把数据写入磁盘。

从共享参数读取数据相对简单，直接调用共享参数实例的get***方法即可读取键值，注意get***方法的第二个参数表示默认值。读取数据的代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\ShareReadActivity.java)
String name = shared.getString("name", "");      // 从共享参数获取名为name的字符串
int age = shared.getInt("age", 0);               // 从共享参数获取名为age的整型数
// 从共享参数获取名为married的布尔值
boolean married = shared.getBoolean("married", false);
// 从共享参数获取名为weight的浮点数
float weight = shared.getFloat("weight", 0);
```

下面通过测试页面演示共享参数的存取过程：先在编辑页面录入用户注册信息，点击保存按钮把数据提交至共享参数，如图6-1所示；再到查看页面浏览用户注册信息，App从共享参数中读取各项数据，并将注册信息显示在页面上，如图6-2所示。



图 6-1 把注册信息写入共享参数



图 6-2 从共享参数读取注册信息

6.1.2 实现记住密码功能

在上一章末尾的实战项目中，登录页面下方有一个“记住密码”复选框，当时只是为了演示

控件的用法，并未真正记住密码。因为用户退出后重新进入登录页面，App没有回忆起上次的登录密码。现在利用共享参数改造该项目，使之实现记住密码的功能。

改造的内容主要有下列3处：

（1）声明一个共享参数对象，并在onCreate中调用getSharedPreferences方法获取共享参数的实例。

（2）登录成功时，如果用户勾选了“记住密码”复选框，就使用共享参数保存手机号码与密码。也就是在loginSuccess方法中增加以下代码：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\LoginShareActivity.java)
// 如果勾选了“记住密码”复选框，就把手机号码和密码都保存到共享参数中
if (bRemember) {
    SharedPreferences.Editor editor = mShared.edit();  // 获得编辑器的对象
    // 添加名为phone的手机号码
    editor.putString("phone", et_phone.getText().toString());
    // 添加名为password的密码
    editor.putString("password", et_password.getText().toString());
    editor.commit();  // 提交编辑器中的修改
}
```

（3）再次打开登录页面时，App从共享参数中读取手机号码与密码，并自动填入编辑框。也就是在onCreate方法中增加以下代码：

```
// 从share_login.xml获取共享参数实例
mShared = getSharedPreferences("share_login", MODE_PRIVATE);
String phone = mShared.getString("phone", "");            // 获取共享参数保存的手机号码
String password = mShared.getString("password", "");      // 获取共享参数保存的密码
et_phone.setText(phone);            // 在手机号码编辑框中填写上次保存的手机号
et_password.setText(password);      // 在密码编辑框中填写上次保存的密码
```

代码修改完毕，只要用户上次登录成功时勾选了“记住密码”复选框，下次进入登录页面后App就会自动填写上次登录的手机号码与密码。具体的效果如图6-3和图6-4所示。其中，图6-3为用户首次登录成功的界面，此时勾选了“记住密码”复选框；图6-4为用户再次进入登录的界面，因为上次登录成功时已经记住密码，所以这次页面会自动填充保存的登录信息。



图 6-3　首次登录成功的界面



图 6-4　再次进入登录的界面

6.1.3　更安全的数据仓库

虽然SharedPreferences用起来比较方便，但是在一些特殊场景会产生问题。比如共享参数保存的数据较多时，初始化共享参数会把整个文件加载进内存，加载耗时可能导致主线程堵塞。又如在调用apply方法保存数据时，频繁apply容易导致线程等待超时。为此Android官方推出了数据仓库DataStore，并将其作为Jetpack库的基础组件。DataStore提供了两种实现方式，分别是Preferences

DataStore和Proto DataStore，前者采用键值对存储数据，后者采用自定义类型存储数据。其中Preferences DataStore可以直接替代SharedPreferences。

由于DataStore并未集成到SDK中，而是作为第三方框架提供，因此首先要修改模块的build.gradle文件，往dependencies节点添加下面两行配置，表示导入指定版本的DataStore库：

```
implementation "androidx.datastore:datastore-preferences:1.0.0"
implementation "androidx.datastore:datastore-preferences-rxjava2:1.0.0"
```

数据仓库的用法类似于共享参数，首先要指定仓库名称，并创建仓库实例，示例代码如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\util\DatastoreUtil.java)
private RxDataStore<Preferences> mDataStore;  // 声明一个数据仓库实例

private DatastoreUtil(Context context) {
    mDataStore = new RxPreferenceDataStoreBuilder(
        context.getApplicationContext(), "datastore").build();
}

// 获取数据仓库工具的实例
public static DatastoreUtil getInstance(Context context) {
    if (instance == null) {
        instance = new DatastoreUtil(context);
    }
    return instance;
}
```

其次从仓库实例中获取指定键名的数据，下面的代码模板演示了如何从数据仓库中读取字符串值：

```
// 获取指定名称的字符串值
public String getStringValue(String key) {
    Preferences.Key<String> keyId = PreferencesKeys.stringKey(key);
    Flowable<String> flow = mDataStore.data().map(prefs -> prefs.get(keyId));
    try {
        return flow.blockingFirst();
    } catch (Exception e) {
        return "";
    }
}
```

最后往仓库实例写入指定键值，下面的代码模板演示了如何将字符串值写入数据仓库：

```
// 设置指定名称的字符串值
public void setStringValue(String key, String value) {
    Preferences.Key<String> keyId = PreferencesKeys.stringKey(key);
    Single<Preferences> result = mDataStore.updateDataAsync(prefs -> {
        MutablePreferences mutablePrefs = prefs.toMutablePreferences();
        mutablePrefs.set(keyId, value);
        return Single.just(mutablePrefs);
    });
}
```

前面把数据仓库的初始化以及读写操作封装在DatastoreUtil中，接下来通过该工具类即可方便地访问数据仓库了。往数据仓库保存数据的代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\DatastoreReadActivity.java)
// 从数据仓库中读取信息
private void readDatastore() {
    // 获取数据仓库工具的实例
    DatastoreUtil datastore = DatastoreUtil.getInstance(this);
```

```
        String desc = "数据仓库中保存的信息如下：";
        desc = String.format("%s\n  %s为%s", desc, "姓名",
                datastore.getStringValue("name"));
        desc = String.format("%s\n  %s为%d", desc, "年龄",
                datastore.getIntValue("age"));
        desc = String.format("%s\n  %s为%d", desc, "身高",
                datastore.getIntValue("height"));
        desc = String.format("%s\n  %s为%.2f", desc, "体重",
                datastore.getDoubleValue("weight"));
        desc = String.format("%s\n  %s为%b", desc, "婚否",
                datastore.getBooleanValue("married"));
        desc = String.format("%s\n  %s为%s", desc, "更新时间",
                datastore.getStringValue("update time"));
        tv_data.setText(desc);
    }
```

从数据仓库获取数据的代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\DatastoreWriteActivity.java)
// 获取数据仓库工具的实例
DatastoreUtil datastore = DatastoreUtil.getInstance(this);
datastore.setStringValue("name", name);  // 添加一个名叫name的字符串
// 添加一个名叫age的整数
datastore.setIntValue("age", Integer.parseInt(age));
// 添加一个名叫height的整数
datastore.setIntValue("height", Integer.parseInt(height));
// 添加一个名叫weight的双精度数
datastore.setDoubleValue("weight", Double.parseDouble(weight));
// 添加一个名叫married的布尔值
datastore.setBooleanValue("married", isMarried);
datastore.setStringValue("update_time",
            DateUtil.getNowDateTime("yyyy-MM-dd HH:mm:ss"));
```

运行测试App，先打开记录保存页面，填写数据后点击“保存到数据仓库”按钮，此时保存界面如图6-5所示。然后打开记录获取界面，看到记录获取结果如图6-6所示，说明成功实现了数据仓库的读写功能。



图 6-5　将登录信息保存到数据仓库

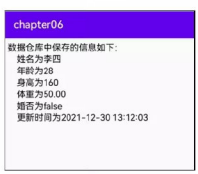


图 6-6　从数据仓库读取登录信息

6.2　数　据　库

本节介绍Android的数据库存储方式——SQLite的使用方法，包括：SQLite用到了哪些SQL语法，如何使用数据库管理器操纵SQLite，如何使用数据库帮助器简化数据库操作，以及如何利用SQLite改进登录页面的记住密码功能。

6.2.1 SQL的基本语法

SQL本质上是一种编程语言，它的学名叫作“结构化查询语言”（全称为Structured Query Language，简称SQL）。不过SQL语言并非通用的编程语言，它专用于数据库的访问和处理，更像是一种操作命令，所以常说SQL语句而不说SQL代码。标准的SQL语句分为3类：数据定义、数据操纵和数据控制。但不同的数据库往往有自己的实现。

SQLite是一种小巧的嵌入式数据库，使用方便、开发简单。如同MySQL、Oracle那样，SQLite也采用SQL语句管理数据，由于它属于轻型数据库，不涉及复杂的数据控制操作，因此App开发只用到数据定义和数据操纵两类SQL语句。此外，SQLite的SQL语法与通用的SQL语法略有不同，接下来介绍的两类SQL语法全部基于SQLite。

1. 数据定义语言

数据定义语言（全称Data Definition Language，简称DDL）描述了怎样变更数据实体的框架结构。就SQLite而言，DDL语言主要包括3种操作：创建表格、删除表格、修改表结构，分别说明如下。

（1）创建表格

表格的创建动作由create命令完成，格式为“CREATE TABLE IF NOT EXISTS 表格名称 (以逗号分隔的各字段定义);”。以用户信息表为例，它的建表语句如下：

```
CREATE TABLE IF NOT EXISTS user_info (
    _id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    name VARCHAR NOT NULL, age INTEGER NOT NULL,
    height LONG NOT NULL, weight FLOAT NOT NULL,
    married INTEGER NOT NULL, update_time VARCHAR NOT NULL);
```

上面的SQL语法与其他数据库的SQL语法有所出入，相关的注意点说明见下：

① SQL语句不区分大小写，无论是create与table这类关键词，还是表格名称、字段名称，都不区分大小写。唯一区分大小写的是被单引号括起来的字符串值。

② 为避免重复建表，应加上IF NOT EXISTS关键词，例如CREATE TABLE IF NOT EXISTS 表格名称 ……

③ SQLite支持整型INTEGER、长整型LONG、字符串VARCHAR、浮点数FLOAT，但不支持布尔类型。布尔类型的数据要使用整型保存，如果直接保存布尔数据，在入库时SQLite会自动将它转为0或1，其中0表示false，1表示true。

④ 建表时需要唯一标识字段，它的字段名为_id。创建新表都要加上该字段定义，例如_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL。

（2）删除表格

表格的删除动作由drop命令完成，格式为“DROP TABLE IF EXISTS 表格名称;”。下面是删除用户信息表的SQL语句例子：

```
DROP TABLE IF EXISTS user_info;
```

（3）修改表结构

表格的修改动作由alter命令完成，格式为“ALTER TABLE 表格名称 修改操作;”。不过SQLite只支持增加字段，不支持修改字段，也不支持删除字段。对于字段增加操作，需要在alter之

后补充add命令，具体格式如“ALTER TABLE 表格名称 ADD COLUMN 字段名称 字段类型;”。下面是给用户信息表增加手机号字段的SQL语句例子：

```
ALTER TABLE user_info ADD COLUMN phone VARCHAR;
```

注 意

SQLite 的 alter 命令每次只能添加一列字段，若要添加多列，就得分多次添加。

2. 数据操纵语言

数据操纵语言（全称Data Manipulation Language，简称DML）描述了怎样处理数据实体的内部记录。表格记录的操作类型包括添加、删除、修改、查询4类，分别说明如下：

（1）添加记录

记录的添加动作由insert命令完成，格式为“INSERT INTO 表格名称 (以逗号分隔的字段名列表) VALUES (以逗号分隔的字段值列表);”。下面是往用户信息表插入一条记录的SQL语句例子：

```
INSERT INTO user info (name,age,height,weight,married,update time)
VALUES ('张三',20,170,50,0,'20200504');
```

（2）删除记录

记录的删除动作由delete命令完成，格式为“DELETE FROM 表格名称 WHERE 查询条件;”，其中查询条件的表达式形如“字段名=字段值”，多个字段的条件交集通过“AND”连接，条件并集通过“OR”连接。下面是从用户信息表删除指定记录的SQL语句例子：

```
DELETE FROM user_info WHERE name='张三';
```

（3）修改记录

记录的修改动作由update命令完成，格式为“UPDATE 表格名称 SET 字段名=字段值 WHERE 查询条件;”。下面是对用户信息表更新指定记录的SQL语句例子：

```
UPDATE user_info SET married=1 WHERE name='张三';
```

（4）查询记录

记录的查询动作由select命令完成，格式为“SELECT 以逗号分隔的字段名列表 FROM 表格名称 WHERE 查询条件;”。如果字段名列表填星号（*），则表示查询该表的所有字段。下面是从用户信息表查询指定记录的SQL语句例子：

```
SELECT name FROM user_info WHERE name='张三';
```

查询操作除了比较字段值条件之外，常常需要对查询结果排序，此时要在查询条件后面添加排序条件，对应的表达式为“ORDER BY 字段名 ASC或者DESC”，意指对查询结果按照某个字段排序，其中ASC代表升序，DESC代表降序。下面是查询记录并对结果排序的SQL语句例子：

```
SELECT * FROM user_info ORDER BY age ASC;
```

如果读者之前不熟悉SQL语法，建议下载一个SQLite管理软件，譬如SQLiteStudio，先在计算机上多加练习SQLite的常见操作语句。

6.2.2 数据库管理器 SQLiteDatabase

SQL语句毕竟只是SQL命令，若要在Java代码中操纵SQLite，还需专门的工具类。SQLiteDatabase便是Android提供的SQLite数据库管理器，开发者可以在活动页面代码中调用openOrCreateDatabase方法获取数据库实例，参考代码如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\DatabaseActivity.java)
// 创建名为test.db的数据库。数据库如果不存在就创建它，如果存在就打开它
SQLiteDatabase db = openOrCreateDatabase(getFilesDir() + "/test.db",
                    Context.MODE_PRIVATE, null);
String desc = String.format("数据库%s创建%s", db.getPath(),
                    (db!=null)?"成功":"失败");
tv_database.setText(desc);
// deleteDatabase(getFilesDir() + "/test.db");  // 删除名为test.db的数据库
```

首次运行测试App，调用openOrCreateDatabase方法会自动创建数据库，并返回该数据库的管理器实例，创建结果如图6-7所示。

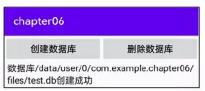


图 6-7 数据库的创建结果

获得数据库实例之后，就能对该数据库开展各项操作了。数据库管理器SQLiteDatabase提供了若干操作数据表的API，常用的方法有3类，列举如下：

1. 管理类，用于数据库层面的操作

- openDatabase：打开指定路径的数据库。
- isOpen：判断数据库是否已打开。
- close：关闭数据库。
- getVersion：获取数据库的版本号。
- setVersion：设置数据库的版本号。

2. 事务类，用于事务层面的操作

- beginTransaction：开始事务。
- setTransactionSuccessful：设置事务的成功标志。
- endTransaction：结束事务。执行本方法时，系统会判断之前是否调用了setTransactionSuccessful方法，如果之前已调用该方法就提交事务，如果没有调用该方法就回滚事务。

3. 数据处理类，用于数据表层面的操作

- execSQL：执行拼接好的SQL控制语句。一般用于建表、删表、变更表结构。
- delete：删除符合条件的记录。
- update：更新符合条件的记录信息。
- insert：插入一条记录。
- query：执行查询操作，并返回结果集的游标。
- rawQuery：执行拼接好的SQL查询语句，并返回结果集的游标。

在实际开发中，经常用到的是查询语句，建议先写好查询操作的select语句，再调用rawQuery方法执行查询语句。

6.2.3 数据库帮助器 SQLiteOpenHelper

由于SQLiteDatabase存在局限性，一不小心就会重复打开数据库，处理数据库的升级也不方便，因此Android提供了数据库帮助器SQLiteOpenHelper，帮助开发者合理使用SQLite。

SQLiteOpenHelper的具体使用步骤如下：

01 新建一个继承自SQLiteOpenHelper的数据库操作类，按提示重写onCreate和onUpgrade两个方法。其中，onCreate方法只在第一次打开数据库时执行，在此可以创建表结构；而onUpgrade方法在数据库版本升高时执行，在此可以根据新旧版本号变更表结构。

02 为保证数据库的安全使用，需要封装几个必要方法，包括获取单例对象、打开数据库连接、关闭数据库连接，说明如下：

- 获取单例对象：确保在 App 运行过程中数据库只会打开一次，避免重复打开引起错误。
- 打开数据库连接：SQLite 有锁机制，即读锁和写锁的处理，故而数据库连接也分两种，读连接可调用 getReadableDatabase 方法获得，写连接可调用 getWritableDatabase 方法获得。
- 关闭数据库连接：数据库操作完毕，调用数据库实例的 close 方法关闭连接。

03 提供对表记录增加、删除、修改、查询的操作方法。

能被SQLite直接使用的数据结构是ContentValues类，它类似于映射Map，也提供了put和get方法存取键值对。区别之处在于：ContentValues的键只能是字符串，不能是其他类型。ContentValues主要用于增加记录和更新记录，对应数据库的insert和update方法。

记录的查询操作用到了游标类Cursor，调用query和rawQuery方法返回的都是Cursor对象，若要获取全部的查询结果，则需根据游标的指示一条一条遍历结果集合。Cursor的常用方法可分为3类，说明如下：

1. 游标控制类方法，用于指定游标的状态

- close：**关闭游标**。
- isClosed：**判断游标是否关闭**。
- isFirst：**判断游标是否在开头**。
- isLast：**判断游标是否在末尾**。

2. 游标移动类方法，把游标移动到指定位置

- moveToFirst：**移动游标到开头**。
- moveToLast：**移动游标到末尾**。
- moveToNext：**移动游标到下一条记录**。
- moveToPrevious：**移动游标到上一条记录**。
- move：**往后移动游标若干条记录**。
- moveToPosition：**移动游标到指定位置的记录**。

3. 获取记录类方法，可获取记录的数量、类型以及取值

- getCount：**获取结果记录的数量**。

- getInt：**获取指定字段的整型值**。
- getLong：**获取指定字段的长整型值**。
- getFloat：**获取指定字段的浮点数值**。
- getString：**获取指定字段的字符串值**。
- getType：**获取指定字段的字段类型**。

鉴于数据库操作的特殊性，不方便单独演示某个功能，接下来从创建数据库开始介绍，完整演示一下数据库的读写操作。用户注册信息的演示页面包括两个，分别是记录保存页面和记录读取页面，其中记录保存页面通过insert方法向数据库添加用户信息，完整代码见chapter06\src\main\java\com\example\chapter06\SQLiteWriteActivity.java；而记录读取页面通过query方法从数据库读取用户信息，完整代码见chapter06\src\main\java\com\example\chapter06\ SQLiteReadActivity.java。

运行测试App，先打开记录保存页面，依次录入信息并将两个用户的注册信息保存至数据库，如图6-8和图6-9所示。再打开记录读取页面，从数据库读取用户注册信息并展示在页面上，如图6-10所示。



图 6-8　第一条注册信息保存到数据库



图 6-9　第二条注册信息保存到数据库



图 6-10　从数据库读取了两条注册信息

上述演示页面主要用到了数据库记录的添加、查询和删除操作，对应的数据库帮助器关键代码如下所示，尤其关注里面的insert、delete、update和query方法：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\database\UserDBHelper.java)
public class UserDBHelper extends SQLiteOpenHelper {
    private static final String DB_NAME = "user.db";              // 数据库的名称
    private static UserDBHelper mHelper = null;                   // 数据库帮助器的实例
    private SQLiteDatabase mDB = null;                            // 数据库的实例
    public static final String TABLE_NAME = "user_info";          // 表的名称
    private UserDBHelper(Context context, int version) {
        super(context, DB_NAME, null, version);
    }
    // 利用单例模式获取数据库帮助器的唯一实例
    public static UserDBHelper getInstance(Context context, int version) {
        if (version > 0 && mHelper == null) {
            mHelper = new UserDBHelper(context, version);
        }
        return mHelper;
    }
    // 打开数据库的读连接
    public SQLiteDatabase openReadLink() {
        if (mDB == null || !mDB.isOpen()) {
            mDB = mHelper.getReadableDatabase();
```

```
        }
        return mDB;
    }
    // 打开数据库的写连接
    public SQLiteDatabase openWriteLink() {
        if (mDB == null || !mDB.isOpen()) {
            mDB = mHelper.getWritableDatabase();
        }
        return mDB;
    }
    // 关闭数据库连接
    public void closeLink() {
        if (mDB != null && mDB.isOpen()) {
            mDB.close();
            mDB = null;
        }
    }
    // 创建数据库，执行建表语句
    public void onCreate(SQLiteDatabase db) {
        String create_sql = "CREATE TABLE IF NOT EXISTS " + TABLE_NAME + " ("
                + " id INTEGER PRIMARY KEY  AUTOINCREMENT NOT NULL,"
                + "name VARCHAR NOT NULL," + "age INTEGER NOT NULL,"
                + "height INTEGER NOT NULL," + "weight FLOAT NOT NULL,"
                + "married INTEGER NOT NULL," + "update time VARCHAR NOT NULL"
                + ",phone VARCHAR" + ",password VARCHAR" + ");";
        db.execSQL(create_sql);  // 执行完整的SQL语句
    }
    // 升级数据库，执行表结构变更语句
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {}
    // 根据指定条件删除表记录
    public int delete(String condition) {
        // 执行删除记录动作，该语句返回删除记录的数目
        return mDB.delete(TABLE_NAME, condition, null);
    }
    // 往该表添加多条记录
    public long insert(List<UserInfo> infoList) {
        long result = -1;
        for (int i = 0; i < infoList.size(); i++) {
            UserInfo info = infoList.get(i);
            // 不存在唯一性重复的记录，则插入新记录
            ContentValues cv = new ContentValues();
            cv.put("name", info.name);
            cv.put("age", info.age);
            cv.put("height", info.height);
            cv.put("weight", info.weight);
            cv.put("married", info.married);
            cv.put("update time", info.update time);
            cv.put("phone", info.phone);
            cv.put("password", info.password);
            // 执行插入记录动作，该语句返回插入记录的行号
            result = mDB.insert(TABLE_NAME, "", cv);
            if (result == -1) {  // 添加成功则返回行号，添加失败则返回-1
                return result;
            }
        }
        return result;
    }
    // 根据条件更新指定的表记录
```

```
    public int update(UserInfo info, String condition) {
        ContentValues cv = new ContentValues();
        cv.put("name", info.name);
        cv.put("age", info.age);
        cv.put("height", info.height);
        cv.put("weight", info.weight);
        cv.put("married", info.married);
        cv.put("update time", info.update time);
        cv.put("phone", info.phone);
        cv.put("password", info.password);
        // 执行更新记录动作，该语句返回更新的记录数量
        return mDB.update(TABLE NAME, cv, condition, null);
    }
    // 根据指定条件查询记录，并返回结果数据列表
    public List<UserInfo> query(String condition) {
        String sql = String.format("select rowid,_id,name,age,height," +
                    "weight,married,update time,phone,password " +
                    "from %s where %s;", TABLE NAME, condition);
        List<UserInfo> infoList = new ArrayList<UserInfo>();
        // 执行记录查询动作，该语句返回结果集的游标
        Cursor cursor = mDB.rawQuery(sql, null);
        // 循环取出游标指向的每条记录
        while (cursor.moveToNext()) {
            UserInfo info = new UserInfo();
            info.rowid = cursor.getLong(0);             // 取出长整型数
            info.xuhao = cursor.getInt(1);              // 取出整型数
            info.name = cursor.getString(2);            // 取出字符串
            info.age = cursor.getInt(3);                // 取出整型数
            info.height = cursor.getLong(4);            // 取出长整型数
            info.weight = cursor.getFloat(5);           // 取出浮点数
            //SQLite没有布尔型，用0表示false，用1表示true
            info.married = (cursor.getInt(6) == 0) ? false : true;
            info.update time = cursor.getString(7);     // 取出字符串
            info.phone = cursor.getString(8);           // 取出字符串
            info.password = cursor.getString(9);        // 取出字符串
            infoList.add(info);
        }
        cursor.close();                                 // 查询完毕，关闭数据库游标
        return infoList;
    }
}
```

6.2.4 优化记住密码功能

在“6.1.2 实现记住密码功能”中，虽然使用共享参数实现了记住密码功能，但是该方案只能记住一个用户的登录信息，并且手机号码跟密码没有对应关系，如果换个手机号码登录，前一个用户的登录信息就被覆盖了。真正的记住密码功能应当是这样的：先输入手机号码，然后根据手机号码匹配保存的密码，一个手机号码对应一个密码，从而实现具体手机号码的密码记忆功能。

现在运用数据库技术分条存储各用户的登录信息，并支持根据手机号查找登录信息，从而同时记住多个手机号的密码。具体的改造主要有下列3点：

（1）声明一个数据库的帮助器对象，然后在活动页面的onResume方法中打开数据库连接，在onPasue方法中关闭数据库连接，示例代码如下：

```
（完整代码见chapter06\src\main\java\com\example\chapter06\LoginSQLiteActivity.java）
    private UserDBHelper mHelper;                       // 声明一个用户数据库的帮助器对象
```

```
@Override
protected void onResume() {
    super.onResume();
    mHelper = UserDBHelper.getInstance(this, 1);  // 获得用户数据帮助器的实例
    mHelper.openWriteLink();                      // 恢复页面，则打开数据库连接
}
@Override
protected void onPause() {
    super.onPause();
    mHelper.closeLink();                          // 暂停页面，则关闭数据库连接
}
```

（2）登录成功时，如果用户勾选了“记住密码”复选框，就将手机号码及其密码保存至数据库。也就是在loginSuccess方法中增加如下代码：

```
// 如果勾选了“记住密码”复选框，则把手机号码和密码保存为数据库的用户表记录
if (bRemember) {
    UserInfo info = new UserInfo();           // 创建一个用户信息对象
    info.phone = et_phone.getText().toString();
    info.password = et_password.getText().toString();
    info.update_time = DateUtil.getNowDateTime("yyyy-MM-dd HH:mm:ss");
    mHelper.insert(info);                     // 往用户数据库中添加登录成功的用户信息
}
```

（3）再次打开登录页面，用户输入手机号后点击密码框时，App根据手机号到数据库查找登录信息，并将记录结果中的密码填入密码框。其中根据手机号码查找登录信息，要求在帮助器代码中添加以下方法，用于找到指定手机的登录密码：

```
// 根据手机号码查询指定记录
public UserInfo queryByPhone(String phone) {
    UserInfo info = null;
    List<UserInfo> infoList = query(String.format("phone='%s'", phone));
    if (infoList.size() > 0) {                // 存在该号码的登录信息
        info = infoList.get(0);
    }
    return info;
}
```

此外，上面第3点的点击密码框触发查询操作，用到了编辑框的焦点变更事件，有关焦点变更监听器的详细用法参见第5章的“5.3.2　焦点变更监听器”。就本案例而言，光标切到密码框触发焦点变更事件，具体处理逻辑要求重写监听器的onFocusChange方法，重写后的方法代码如下：

```
@Override
public void onFocusChange(View v, boolean hasFocus) {
    String phone = et_phone.getText().toString();
    // 判断是否是密码编辑框发生焦点变化
    if (v.getId() == R.id.et_password) {
        // 用户已输入手机号码，且密码框获得焦点
        if (phone.length() > 0 && hasFocus) {
            // 根据手机号码到数据库中查询用户记录
            UserInfo info = mHelper.queryByPhone(phone);
            if (info != null) {
                // 找到用户记录，则自动在密码框中填写该用户的密码
                et_password.setText(info.password);
            }
        }
    }
}
```

重新运行测试App，先打开登录页面，勾选“记住密码”复选框，并确保本次登录成功。然后再次进入登录页面，输入手机号码后光标还停留在手机框，如图6-11所示。接着点击密码框，光标随之跳转到密码框，此时密码框自动填入了该号码对应的密码串，如图6-12所示。由效果图可见，这次实现了真正意义上的记住密码功能。



图 6-11 光标在手机号码框



图 6-12 光标在密码输入框

6.3 存 储 卡

本节介绍Android的文件存储方式——在存储卡上读写文件，包括：公有存储空间与私有存储空间有什么区别、如何利用存储卡读写文本文件、如何利用存储卡读写图片文件、如何在App运行的时候动态申请权限等。

6.3.1 私有存储空间与公共存储空间

为了更规范地管理手机存储空间，Android从7.0开始将存储卡划分为私有存储和公共存储两大部分，也就是分区存储方式，系统给每个App都分配了默认的私有存储空间。App在私有空间上读写文件无须任何授权，但是若想在公共空间读写文件，则要在AndroidManifest.xml里面添加下述的权限配置。

```
<!-- 存储卡读写 -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

但是即使App声明了完整的存储卡操作权限，系统仍然默认禁止该App访问公共空间。打开手机的系统设置界面，进入到具体应用的管理页面，会发现该应用的存储访问权限被禁止了，如图6-13所示。



图 6-13 系统设置页面里的存储访问权限开关

当然图示的禁止访问只是不让访问存储卡的公共空间，App自身的私有空间依旧可以正常读写。这缘于Android把存储卡分成了两块区域，一块是所有应用均可访问的公共空间，另一块是只有应用自己才可访问的专享空间。虽然Android给每个应用都分配了单独的安装目录，但是安装目录的空间很紧张，所以Android在存储卡的“Android/data”目录下给每个应用又单独建了一个文件目录，用来保存应用自己需要处理的临时文件。这个目录只有当前应用才能够读写文件，其他应

用是不允许读写的。由于私有空间本身已经加了访问权限控制，因此它不受系统禁止访问的影响，应用操作自己的文件目录自然不成问题。因为私有的文件目录只有属主应用才能访问，所以一旦属主应用被卸载，那么对应的目录也会被删掉。

既然存储卡分为公共空间和私有空间两部分，它们的空间路径获取方法自然也就有所不同。若想获取公共空间的存储路径，调用的是Environment.getExternalStoragePublicDirectory方法；若想获取应用私有空间的存储路径，调用的是getExternalFilesDir方法。下面是分别获取两个空间路径的代码例子：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\FilePathActivity.java)
// 获取系统的公共存储路径
String publicPath = Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_DOWNLOADS).toString();
// 获取当前App的私有存储路径
String privatePath = getExternalFilesDir(Environment.DIRECTORY_DOWNLOADS).toString();
TextView tv_file_path = findViewById(R.id.tv_file_path);
String desc = "系统的公共存储路径位于" + publicPath +
              "\n\n当前App的私有存储路径位于" + privatePath +
              "\n\nAndroid 7.0之后默认禁止访问公共存储目录";
tv_file_path.setText(desc);
```

该例子运行之后获得的路径信息如图6-14所示，可见应用的私有空间路径位于“存储卡根目录/Android/data/应用包名/files/Download”这个目录中。

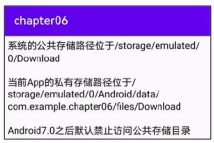


图 6-14 公共存储与私有存储的目录路径

6.3.2 在存储卡上读写文件

文本文件的读写借助于文件IO流FileOutputStream和FileInputStream。其中，FileOutputStream用于写文件，FileInputStream用于读文件，它们读写文件的代码例子如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\util\FileUtil.java)
// 把字符串保存到指定路径的文本文件
public static void saveText(String path, String txt) {
    // 根据指定的文件路径构建文件输出流对象
    try (FileOutputStream fos = new FileOutputStream(path)) {
        fos.write(txt.getBytes());  // 把字符串写入文件输出流
    } catch (Exception e) {
        e.printStackTrace();
    }
}
// 从指定路径的文本文件中读取内容字符串
public static String openText(String path) {
    String readStr = "";
    // 根据指定的文件路径构建文件输入流对象
    try (FileInputStream fis = new FileInputStream(path)) {
        byte[] b = new byte[fis.available()];
        fis.read(b);  // 从文件输入流读取字节数组
        readStr = new String(b);  // 把字节数组转换为字符串
    } catch (Exception e) {
        e.printStackTrace();
    }
    return readStr;  // 返回文本文件中的文本字符串
}
```

接着分别创建写文件页面和读文件页面，其中写文件页面调用saveText方法保存文本，完整代码见chapter06\src\main\java\com\example\chapter06\FileWriteActivity.java；而读文件页面调用readText方法从指定路径的文件中读取文本内容，完整代码见chapter06\src\main\java\com\example\chapter06\FileReadActivity.java。

然后运行测试App，先打开文本写入页面，录入注册信息后保存为私有目录里的文本文件，此时写入界面如图6-15所示。再打开文本读取页面，App自动在私有目录下找到文本文件列表，并展示其中一个文件的文本内容，此时读取界面如图6-16所示。



图 6-15 将注册信息保存到文本文件



图 6-16 从文本文件读取注册信息

文本文件读写可以转换为对字符串的读写，而图片文件保存的是图像数据，需要专门的位图工具Bitmap处理。位图对象依据来源不同又分成3种获取方式，分别对应位图工厂BitmapFactory的下列3个方法：

- decodeResource：**从指定的资源文件中获取位图数据。例如下面代码表示从资源文件huawei.png获取位图对象：**

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.huawei);
```

- decodeFile：**从指定路径的文件中获取位图数据。注意从Android 10开始，该方法只适用于私有目录下的图片，不适用公共空间下的图片。**
- decodeStream：**从指定的输入流中获取位图数据。比如使用IO流打开图片文件，此时文件输入流对象即可作为decodeStream方法的入参，相应的图片读取代码如下：**

```
(完整代码见chapter06\src\main\java\com\example\chapter06\util\FileUtil.java)
// 从指定路径的图片文件中读取位图数据
public static Bitmap openImage(String path) {
    Bitmap bitmap = null;                                    // 声明一个位图对象
    // 根据指定的文件路径构建文件输入流对象
    try (FileInputStream fis = new FileInputStream(path)) {
        bitmap = BitmapFactory.decodeStream(fis);      //从文件输入流中解码位图数据
    } catch (Exception e) {
        e.printStackTrace();
    }
    return bitmap;                                           // 返回图片文件中的位图数据
}
```

得到位图对象之后，就能在图像视图上显示位图。图像视图ImageView提供了下列方法显示各种来源的图片：

- setImageResource：设置图像视图的图片资源，该方法的入参为资源图片的编号，形如“R.drawable.去掉扩展名的图片名称”。
- setImageBitmap：设置图像视图的位图对象，该方法的入参为Bitmap类型。
- setImageURI：设置图像视图的路径对象，该方法的入参为Uri类型。字符串格式的文件路径可通过代码“Uri.parse(file_path)”转换成路径对象。

读取图片文件的方法很多，把位图数据写入图片文件却只有一个，即通过位图对象的compress方法将位图数据压缩到文件输出流。具体的图片写入代码如下：

```
// 把位图数据保存到指定路径的图片文件
public static void saveImage(String path, Bitmap bitmap) {
    // 根据指定的文件路径构建文件输出流对象
    try (FileOutputStream fos = new FileOutputStream(path)) {
        // 把位图数据压缩到文件输出流中
        bitmap.compress(Bitmap.CompressFormat.JPEG, 80, fos);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

接下来完整演示一遍图片文件的读写操作。首先创建图片写入页面，从某个资源图片读取位图数据，再把位图数据保存为私有目录的图片文件，相关代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\ImageWriteActivity.java)
// 获取当前App的私有下载目录
String path = getExternalFilesDir(Environment.DIRECTORY_DOWNLOADS).toString() + "/";
// 从指定的资源文件中获取位图对象
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.huawei);
String file_path = path + DateUtil.getNowDateTime("") + ".jpeg";
FileUtil.saveImage(file_path, bitmap);  // 把位图对象保存为图片文件
tv_path.setText("图片文件的保存路径为：\n" + file_path);
```

然后创建图片读取页面，从私有目录找到图片文件，并挑出一张在图像视图上显示，相关代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\ImageReadActivity.java)
// 获取当前App的私有下载目录
String path = getExternalFilesDir(Environment.DIRECTORY_DOWNLOADS).toString() + "/";
// 获得指定目录下面的所有图片文件
List<File> fileList = FileUtil.getfileList(path, new String[]{".jpeg"});
if (fileList.size() > 0) {
    // 打开并显示选中的图片文件内容
    String file_path = fileList.get(0).getAbsolutePath();
    tv_content.setText("找到最新的图片文件，路径为"+file_path);
    // 显示存储卡图片文件的第一种方式：直接调用setImageURI方法
    //iv_content.setImageURI(Uri.parse(file_path));  // 设置图像视图的路径对象
    // 第二种方式：先调用decodeFile方法获得位图，再调用setImageBitmap方法
    //Bitmap bitmap = BitmapFactory.decodeFile(file_path);
    //iv_content.setImageBitmap(bitmap);                // 设置图像视图的位图对象
    // 第三种方式：先调用FileUtil.openImage获得位图，再调用setImageBitmap方法
    Bitmap bitmap = FileUtil.openImage(file_path);
    iv_content.setImageBitmap(bitmap);                // 设置图像视图的位图对象
}
```

运行测试App，先打开图片写入页面，点击“把资源图片保存到存储卡”按钮，此时写入界面如图6-17所示。再打开图片读取页面，App自动在私有目录下找到图片文件列表，并展示其中一张图片，此时读取界面如图6-18所示。



图 6-17 把资源图片保存到存储卡



图 6-18 从存储卡读取图片文件

6.3.3 运行时动态申请权限

前面的“6.3.1 公共存储空间与私有存储空间”提到，App若想访问存储卡的公共空间，就要在AndroidManifest.xml里面添加下述的权限配置。

```
<!-- 存储卡读写 -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

然而即使App声明了完整的存储卡操作权限，从Android 7.0开始，系统仍然默认禁止该App访问公共空间，必须到设置界面手动开启应用的存储卡权限才行。尽管此举是为用户隐私着想，可是用户怎么知道要手工开权限呢？就算用户知道，去设置界面找到权限开关也颇费周折。为此Android支持在Java代码中处理权限，处理过程分为3个步骤，详述如下：

1. 检查 App 是否开启了指定权限

权限检查需要调用ContextCompat的checkSelfPermission方法，该方法的第一个参数为活动实例，第二个参数为待检查的权限名称，例如存储卡的写权限名为Manifest.permission. WRITE_EXTERNAL_STORAGE。注意checkSelfPermission方法的返回值，当它为PackageManager. PERMISSION_GRANTED时表示已经授权，否则就是未获授权。

2. 请求系统弹窗，以便用户选择是否开启权限

一旦发现某个权限尚未开启，就得弹窗提示用户手工开启，这个弹窗不是开发者自己写的提醒对话框，而是系统专门用于权限申请的对话框。调用ActivityCompat的requestPermissions方法，即可命令系统自动弹出权限申请窗口，该方法的第一个参数为活动实例，第二个参数为待申请的权限名称数组，第三个参数为本次操作的请求代码。

3. 判断用户的权限选择结果

然而上面第二步的requestPermissions方法没有返回值，那怎么判断用户到底选了开启权限还是拒绝权限呢？其实活动页面提供了权限选择的回调方法onRequestPermissionsResult，如果当前页面请求弹出权限申请窗口，那么该页面的Java代码必须重写onRequestPermissionsResult方法，并在该方法内部处理用户的权限选择结果。

具体到编码实现上，前两步的权限校验和请求弹窗可以合并到一块，先调用checkSelfPermission

方法检查某个权限是否已经开启，如果没有开启再调用requestPermissions方法请求系统弹窗。合并之后的检查方法代码示例如下，此处代码支持一次检查一个权限，也支持一次检查多个权限：

```
    (完整代码见chapter06\src\main\java\com\example\chapter06\util\PermissionUtil.java)
    // 检查某个权限。返回true表示已启用该权限，返回false表示未启用该权限
    public static boolean checkPermission(Activity act, String permission, int requestCode) {
        return checkPermission(act, new String[]{permission}, requestCode);
    }

    // 检查多个权限。返回true表示已完全启用权限，返回false表示未完全启用权限
    public static boolean checkPermission(Activity act, String[] permissions, int
requestCode) {
        boolean result = true;
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            int check = PackageManager.PERMISSION_GRANTED;
            // 通过权限数组检查是否都开启了这些权限
            for (String permission : permissions) {
                check = ContextCompat.checkSelfPermission(act, permission);
                if (check != PackageManager.PERMISSION_GRANTED) {
                    break;  // 有个权限没有开启，就跳出循环
                }
            }
            if (check != PackageManager.PERMISSION_GRANTED) {
                // 未开启该权限，则请求系统弹窗，好让用户选择是否立即开启权限
                ActivityCompat.requestPermissions(act, permissions, requestCode);
                result = false;
            }
        }
        return result;
    }
```

注意到上面代码有判断安卓版本号，只有系统版本大于Android 6.0（版本代号为M），才执行后续的权限校验操作。这是因为从Android 6.0开始引入了运行时权限机制，在Android 6.0之前，只要App在AndroidManifest.xml中添加了权限配置，则系统会自动给App开启相关权限；但在Android 6.0之后，即便事先添加了权限配置，系统也不会自动开启权限，而要开发者在App运行时判断权限的开关情况，再据此动态申请未获授权的权限。

回到活动页面代码，一方面增加权限校验入口，比如点击某个按钮后触发权限检查操作，其中Manifest.permission.WRITE_EXTERNAL_STORAGE表示存储卡权限，入口代码如下：

```
    (完整代码见chapter06\src\main\java\com\example\chapter06\MainActivity.java)
    if (v.getId() == R.id.btn_file_write) {  // 点击了按钮btn_file_write
        if (PermissionUtil.checkPermission(this,
                            Manifest.permission.WRITE_EXTERNAL_STORAGE,
                            R.id.btn_file_write % 65536)) {
            // 已获授权，则直接跳转到下个页面
            startActivity(new Intent(this, FileWriteActivity.class));
        }
    }
```

另一方面还要重写活动的onRequestPermissionsResult方法，在方法内部校验用户的选择结果，若用户同意授权，就执行后续业务；若用户拒绝授权，只能提示用户无法开展后续业务了。重写后的方法代码如下：

```
    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions, int[]
grantResults) {
        // requestCode不能为负数，也不能大于2的16次方即65536
```

```
        if (requestCode == R.id.btn_file_write % 65536) {
            if (PermissionUtil.checkGrant(grantResults)) {  // 用户选择了同意授权
                startActivity(new Intent(this, FileWriteActivity.class));
            } else {
                ToastUtil.show(this, "需要允许存储卡权限才能写入公共空间噢");
            }
        }
    }
```

以上代码为了简化逻辑，将结果校验操作封装为PermissionUtil的checkGrant方法，该方法遍历授权结果数组，依次检查每个权限是否都得到授权了。详细的方法代码如下：

```
// 检查权限结果数组，返回true表示都已经获得授权。返回false表示至少有一个未获得授权
public static boolean checkGrant(int[] grantResults) {
    boolean result = true;
    if (grantResults != null) {
        for (int grant : grantResults) {  // 遍历权限结果数组中的每条选择结果
            if (grant != PackageManager.PERMISSION_GRANTED) {  // 未获得授权
                result = false;
            }
        }
    } else {
        result = false;
    }
    return result;
}
```

代码都改好后，运行测试App，由于一开始App默认未开启存储卡权限，因此点击按钮btn_file_write触发了权限校验操作，弹出如图6-19所示的存储卡权限申请窗口。

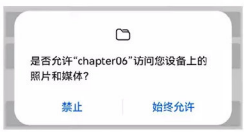


图6-19 App运行时弹出权限申请窗口

点击弹窗上的“始终允许”按钮，表示同意赋予存储卡读写权限，然后系统自动给App开启了存储卡权限，并执行后续处理逻辑，也就是跳转到了FileWriteActivity页面，在该页面即可访问公共空间的文件了。但在Android 10系统中，即使授权通过，App仍然无法访问公共空间，这是因为Android 10默认开启沙箱模式，不允许直接使用公共空间的文件路径，此时要修改AndroidManifest.xml，给application节点添加如下的requestLegacyExternalStorage属性：

```
android:requestLegacyExternalStorage="true"
```

从Android 11开始，为了让应用在升级时也能正常访问公共空间，还得修改AndroidManifest.xml，给application节点添加如下的preserveLegacyExternalStorage属性，表示暂时关闭沙箱模式：

```
android:preserveLegacyExternalStorage="true"
```

除了存储卡的读写权限，还有部分权限也要求在运行时动态申请，这些权限名称的取值说明见表6-1。

表6-1 权限名称的取值说明

代码中的权限名称	权限说明
Manifest.permission.READ_EXTERNAL_STORAGE	读存储卡
Manifest.permission.WRITE_EXTERNAL_STORAGE	写存储卡
Manifest.permission.READ_CONTACTS	读联系人

（续表）

代码中的权限名称	权限说明
Manifest.permission.WRITE_CONTACTS	写联系人
Manifest.permission.SEND_SMS	发送短信
Manifest.permission.RECEIVE_SMS	接收短信
Manifest.permission.READ_SMS	读短信
Manifest.permission.READ_CALL_LOG	读通话记录
Manifest.permission.WRITE_CALL_LOG	写通话记录
Manifest.permission.CAMERA	相机
Manifest.permission.RECORD_AUDIO	录音
Manifest.permission.ACCESS_FINE_LOCATION	精确定位

6.4　应用组件 Application

本节介绍Android的重要组件Application的基本概念和常见用法：首先说明Application的生命周期贯穿了App的整个运行过程，然后利用Application实现App全局变量的读写，以及如何避免方法数过多的问题，最后阐述如何借助App实例来操作Room数据库框架。

6.4.1　Application 的生命周期

Application是Android的一大组件，在App运行过程中有且仅有一个Application对象贯穿应用的整个生命周期。打开AndroidManifest.xml，发现activity节点的上级正是application节点，不过该节点并未指定name属性，此时App采用默认的Application实例。

注意到每个activity节点都指定了name属性，譬如常见的name属性值为.MainActivity，让人知晓该activity的入口代码是MainActivity.java。现在尝试给application节点加上name属性，看看其庐山真面目，具体步骤说明如下：

01 打开AndroidManifest.xml，给application节点加上name属性，表示application的入口代码是MainApplication.java。修改后的application节点示例如下：

```
(完整代码见chapter06\src\main\AndroidManifest.xml)
<application
    android:name=".MainApplication"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
```

02 在Java代码的包名目录下创建MainApplication.java，要求该类继承Application，继承之后可供重写的方法主要有以下3个。

- onCreate：在 App 启动时调用。
- onTerminate：在 App 终止时调用（按字面意思）。
- onConfigurationChanged：在配置改变时调用，例如从竖屏变为横屏。

光看字面意思的话，与生命周期有关的方法是onCreate和onTerminate，那么重写这两个方法，并在重写后的方法中打印日志，修改后的Java代码如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\MainApplication.java)
public class MainApplication extends Application {
    private final static String TAG = "MainApplication";

    @Override
    public void onCreate() {
        super.onCreate();
        Log.d(TAG, "onCreate");
    }
    @Override
    public void onTerminate() {
        super.onTerminate();
        Log.d(TAG, "onTerminate");
    }
}
```

08 运行测试App，在Logcat窗口观察应用日志。但是只在启动一开始看到MainApplication的onCreate日志（该日志先于MainActivity的onCreate日志），却始终无法看到它的onTerminate日志，无论是自行退出App还是强行杀掉App，日志都不会打印onTerminate。

无论你怎么折腾，这个onTerminate日志都不会出来。Android明明提供了这个方法，同时提供了关于该方法的解释，说明文字如下：This method is for use in emulated process environments. It will never be called on a production Android device, where processes are removed by simply killing them; no user code (including this callback) is executed when doing so。这段话的意思是：该方法供模拟环境使用，它在真机上永远不会被调用，无论是直接杀进程还是代码退出；执行该操作时，不会执行任何用户代码。

现在很明确了，onTerminate方法就是个摆设，中看不中用。如果读者想在App退出前回收系统资源，就不能指望onTerminate方法的回调了。

6.4.2 利用 Application 操作全局变量

C/C++有全局变量的概念，因为全局变量保存在内存中，所以操作全局变量就是操作内存，显然内存的读写速度远比读写数据库或读写文件快得多。所谓全局，指的是其他代码都可以引用该变量，因此全局变量是共享数据和消息传递的好帮手。不过Java没有全局变量的概念，与之比较接近的是类里面的静态成员变量，该变量不但能被外部直接引用，而且它在不同地方引用的值是一样的（前提是在引用期间不能改动变量值），所以借助静态成员变量也能实现类似全局变量的功能。

根据上一小节的介绍可知，Application的生命周期覆盖了App运行的全过程。不像短暂的Activity生命周期，一旦退出该页面，Activity实例就被销毁。因此，利用Application的全生命特性，能够在Application实例中保存全局变量。

适合在Application中保存的全局变量主要有下面3类数据：

- **会频繁读取的信息，例如用户名、手机号码等。**
- **不方便由意图传递的数据，例如位图对象、非字符串类型的集合对象等。**
- **容易因频繁分配内存而导致内存泄漏的对象，例如Handler处理器实例等。**

要想通过Application实现全局内存的读写，得完成以下3项工作：

（1）编写一个继承自Application的新类MainApplication。该类采用单例模式，内部先声明自身类的一个静态成员对象，在创建App时把自身赋值给这个静态对象，然后提供该对象的获取方法getInstance。具体实现代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\MainApplication.java)
public class MainApplication extends Application {
    private static MainApplication mApp;  // 声明一个当前应用的静态实例
    // 声明一个公共的信息映射，可当作全局变量使用
    public HashMap<String, String> infoMap = new HashMap<String, String>();
    // 利用单例模式获取当前应用的唯一实例
    public static MainApplication getInstance() {
        return mApp;
    }
    @Override
    public void onCreate() {
        super.onCreate();
        mApp = this;  // 在打开应用时对静态的应用实例赋值
    }
}
```

（2）在活动页面代码中调用MainApplication的getInstance方法，获得它的一个静态对象，再通过该对象访问MainApplication的公共变量和公共方法。

（3）不要忘了在AndroidManifest.xml中注册新定义的Application类名，也就是给application节点增加android:name属性，其值为.MainApplication。

接下来演示如何读写内存中的全局变量。首先分别创建写内存页面和读内存页面，其中写内存页面把用户的注册信息保存到全局变量infoMap，完整代码见chapter06\src\main\java\com\example\chapter06\AppWriteActivity.java；而读内存页面从全局变量infoMap读取用户的注册信息，完整代码见chapter06\src\main\java\com\example\chapter06\AppReadActivity.java。

然后运行测试App，先打开内存写入页面，录入注册信息后保存至全局变量，此时写入界面如图6-20所示。再打开内存读取页面，App自动从全局变量获取注册信息，并展示拼接后的信息文本，此时读取界面如图6-21所示。



图 6-20　注册信息保存到全局内存



图 6-21　从全局内存读取注册信息

6.4.3　避免方法数过多的问题

一个大规模的App工程，往往引入了数量繁多的第三方开发库，其中既有官方的Jetpack库，也有第三方厂商的开源包。有时候运行这种App会报错“Cannot fit requested classes in a single dex file (# methods: 65894 > 65536)”，意思是App内部引用的方法数量超过了65536个，导致App异常退出。

原来Android的每个App代码都放在一个dex文件中，系统会把内部方法的索引保存在一个链表结构里，由于这个链表的长度变量是short类型（short类型的数字占两个字节共16位），使得链表的最大长度不能超过65536（2的16次方），因此若App方法数超过65536的话，链表索引溢出就报错了。为了解决方法数过多的问题，Android推出了名叫MultiDex的解决方案，也就是在打包时把应用分成多个dex文件，每个dex文件中的方法数量均不超过65536个，由此规避了方法数过多的限制。

若想让App工程支持MultiDex，需要对其略加改造，具体改造步骤说明如下：

01 修改模块的build.gradle文件，往dependencies节点添加下面一行配置，表示导入指定版本的MultiDex库：

```
implementation 'androidx.multidex:multidex:2.0.1'
```

02 在defaultConfig节点添加以下配置，表示开启多个dex功能：

```
    multiDexEnabled true // 避免方法数最多65536的问题
```

03 编写自定义的Application，注意该Application类必须继承MultiDexApplication，代码如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\MainApplication.java)
public class MainApplication extends MultiDexApplication {
    // 此处省略内部的方法与属性代码
}
```

04 打开AndroidManifest.xml，给application节点的android:name属性设置自定义的Application，示例如下：

```
    android:name=".MainApplication"
```

05 重新编译App工程，之后运行的App就不会再出现方法数过多的问题了。

6.4.4 利用 Room 简化数据库操作

虽然Android提供了数据库帮助器，但是开发者在进行数据库编程时仍有诸多不便，比如每次增加一张新表，开发者都得手工实现以下代码逻辑：

（1）重写数据库帮助器的onCreate方法，添加该表的建表语句。

（2）在插入记录之时，必须将数据实例的属性值逐一赋给该表的各字段。

（3）在查询记录之时，必须遍历结果集游标，把各字段值逐一赋给数据实例。

（4）每次读写操作之前，都要先开启数据库连接；读写操作之后，又要关闭数据库连接。

上述的处理操作无疑存在不少重复劳动，数年来引得开发者叫苦连连。为此各类数据库处理框架纷纷涌现，包括GreenDao、OrmLite、Realm等，可谓百花齐放。眼见SQLite渐渐乏人问津，谷歌公司干脆整了个自己的数据库框架——Room，该框架同样基于SQLite，但它通过注解技术极大地简化了数据库操作，减少了原来相当大一部分编码工作量。

由于Room并未集成到SDK中，而是作为第三方框架提供，因此要修改模块的build.gradle文件，往dependencies节点添加下面两行配置，表示导入指定版本的Room库：

```
implementation 'androidx.room:room-runtime:2.4.2'
annotationProcessor 'androidx.room:room-compiler:2.4.2'
```

导入Room库之后，还要编写若干对应的代码文件。以录入图书信息为例，此时要对图书信息表进行增删改查，则具体的编码过程分为下列5个步骤：

1. 编写图书信息表对应的实体类

假设图书信息类名为BookInfo，且它的各属性与图书信息表的各字段一一对应，那么要给该类添加“@Entity”注解，表示该类是Room专用的数据类型，对应的表名称也叫作BookInfo。如果BookInfo表的name字段是该表的主键，则需给BookInfo类的name属性添加“@PrimaryKey”与“@NonNull”两个注解，表示该字段是个非空的主键。下面是BookInfo类的定义代码例子：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\entity\BookInfo.java)
@Entity
public class BookInfo {
    @PrimaryKey                     // 该字段是主键，不能重复
    @NonNull                        // 主键必须是非空字段
    private String name;            // 图书名称
    private String author;          // 作者
    private String press;           // 出版社
    private double price;           // 价格
    // 以下省略各属性的set***方法和get***方法
}
```

2. 编写图书信息表对应的持久化类

所谓持久化，指的是将数据保存到磁盘而非内存，其实等同于增删改等SQL语句。假设图书信息表的持久化类名叫作BookDao，那么该类必须添加“@Dao”注解，内部的记录查询方法必须添加“@Query”注解，记录插入方法必须添加“@Insert”注解，记录更新方法必须添加“@Update”注解，记录删除方法必须添加“@Delete”注解（带条件的删除方法除外）。对于记录查询方法，允许在@Query之后补充具体的查询语句以及查询条件；对于记录插入方法与记录更新方法，需明确出现重复记录时要采取哪种处理策略。下面是BookDao类的定义代码例子：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\dao\BookDao.java)
@Dao
public interface BookDao {
    @Query("SELECT * FROM BookInfo")                        // 设置查询语句
    List<BookInfo> getAllBook();                            // 加载所有图书信息

    @Query("SELECT * FROM BookInfo WHERE name = :name")     // 设置带条件的查询语句
    BookInfo getBookByName(String name);                    // 根据名字加载图书

    @Insert(onConflict = OnConflictStrategy.REPLACE)        // 记录重复时替换原记录
    void insertOneBook(BookInfo book);                      // 插入一条图书信息

    @Insert
    void insertBookList(List<BookInfo> bookList);           // 插入多条图书信息

    @Update(onConflict = OnConflictStrategy.REPLACE)        // 出现重复记录时替换原记录
    int updateBook(BookInfo book);                          // 更新图书信息

    @Delete
    void deleteBook(BookInfo book);                         // 删除图书信息

    @Query("DELETE FROM BookInfo WHERE 1=1")                // 设置删除语句
    void deleteAllBook();                                   // 删除所有图书信息
}
```

3. 编写图书信息表对应的数据库类

因为先有数据库然后才有表，所以图书信息表还得放到某个数据库里，这个默认的图书数据库要从RoomDatabase派生而来，并添加“@Database”注解。下面是数据库类BookDatabase的定义代码例子：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\database\BookDatabase.java)
//entities表示该数据库有哪些表，version表示数据库的版本号
//exportSchema表示是否导出数据库信息的JSON串，建议设为false，若设为true还需在build.gradle中指定JSON文件的保存路径
@Database(entities = {BookInfo.class},version = 1, exportSchema = false)
public abstract class BookDatabase extends RoomDatabase {
    // 获取该数据库中某张表的持久化对象
    public abstract BookDao bookDao();
}
```

4. 在自定义的Application类中声明图书数据库的唯一实例

为了避免重复打开数据库造成的内存泄漏问题，每个数据库在App运行过程中理应只有一个实例，此时要求开发者自定义新的Application类，在该类中声明并获取图书数据库的实例，并将自定义的Application类设为单例模式，保证App在运行之时有且仅有一个应用实例。下面是自定义Application类的代码例子：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\MainApplication.java)
public class MainApplication extends Application {
    private static MainApplication mApp;          // 声明一个当前应用的静态实例
    private BookDatabase bookDatabase;            // 声明一个图书数据库对象

    // 利用单例模式获取当前应用的唯一实例
    public static MainApplication getInstance() {
        return mApp;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        mApp = this;  // 在打开应用时对静态的应用实例赋值
        // 构建图书数据库的实例
        bookDatabase = Room.databaseBuilder(mApp,
                BookDatabase.class,"BookInfo")
                .addMigrations()          // 允许迁移数据库（发生数据库变更时，Room默认删除原数据库再创建新数据库。如此一来原来的记录会丢失，故而要改为迁移方式以便保存原有记录）
                .allowMainThreadQueries() // 允许在主线程中操作数据库（Room默认不能在主线程中操作数据库）
                .build();
    }

    // 获取图书数据库的实例
    public BookDatabase getBookDB(){
        return bookDatabase;
    }
}
```

5. 在操作图书信息表的地方获取数据表的持久化对象

持久化对象的获取代码很简单，只需下面一行代码就够了：

```
// 从App实例中获取唯一的图书持久化对象
BookDao bookDao = MainApplication.getInstance().getBookDB().bookDao();
```

完成以上5个编码步骤之后，接着调用持久化对象的query***、insert***、update***、delete***等方法，就能实现图书信息的增删改查操作了。例程的图书信息演示页面有两个，分别是记录保存页面和记录读取页面。其中记录保存页面通过insertOneBook方法向数据库添加图书信息，完整代码见chapter06\src\main\java\com\example\chapter06\RoomWriteActivity.java；而记录读取页面通过

queryAllBook方法从数据库读取图书信息，完整代码见chapter06\src\main\java\com\example\chapter06\RoomReadActivity.java。

运行测试App，先打开记录保存页面，依次录入两本图书信息并保存至数据库，如图6-22和图6-23所示。再打开记录读取页面，从数据库读取图书信息并展示在页面上，如图6-24所示。

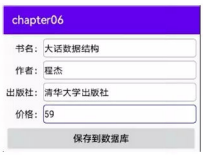


图 6-22　将第一本图书信息保存到数据库



图 6-23　将第二本图书信息保存到数据库

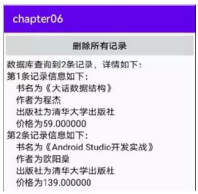


图 6-24　从数据库读取了两本图书信息

6.5　共享数据

本节介绍Android的四大组件之一ContentProvider的基本概念和常见用法：首先说明如何使用内容提供器封装内部数据的外部访问接口，然后阐述如何使用内容解析器通过外部接口操作内部数据，最后叙述如何利用内容解析器读写联系人信息，以及如何利用内容观察器监听收到的短信内容。

6.5.1　通过 ContentProvider 封装数据

Android提供了四大组件，分别是活动Activity、广播Broadcast、服务Service和内容提供器ContentProvider。其中内容提供器涵盖与内部数据存取有关的一系列组件，完整的内容组件由内容提供器ContentProvider、内容解析器ContentResolver、内容观察器ContentObserver三部分组成。

ContentProvider给App存取内部数据提供了统一的外部接口，让不同的应用之间得以互相共享数据。像上一章提到的SQLite可操作应用自身的内部数据库，上传和下载功能可操作后端服务器的文件，而ContentProvider可操作当前设备其他应用的内部数据，它是一种中间层次的数据存储形式。

在实际编码中，ContentProvider只是服务端App存取数据的抽象类，开发者需要在其基础上实现一个完整的内容提供器，并重写下列数据库管理方法。

- onCreate：**创建数据库并获得数据库连接。**
- insert：**插入数据。**
- delete：**删除数据。**
- update：**更新数据。**
- query：**查询数据，并返回结果集的游标。**
- getType：**获取内容提供器支持的数据类型。**

这些方法看起来是不是很像SQLite？没错，ContentProvider作为中间接口，本身并不直接保存

数据，而是通过SQLiteOpenHelper与SQLiteDatabase间接操作底层的数据库。所以要想使用ContentProvider，首先得实现SQLite的数据库帮助器，然后由ContentProvider封装对外的接口。以封装用户信息为例，具体步骤主要分成以下3步。

1. 编写用户信息表的数据库帮助器

这个数据库帮助器就是常规的SQLite操作代码，实现过程参见本章的“6.2.3 数据库帮助器SQLiteOpenHelper”，完整代码参见chapter06\src\main\java\com\example\chapter06\database\UserDBHelper.java。

2. 编写内容提供器的基础字段类

该类需要实现接口BaseColumns，同时加入几个常量定义。详细代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\provider\UserInfoContent.java)
public class UserInfoContent implements BaseColumns {
    // 这里的名称必须与AndroidManifest.xml里的android:authorities保持一致
    public static final String AUTHORITIES = "com.example.chapter06.provider.
                                              UserInfoProvider";
    // 内容提供器的外部表名
    public static final String TABLE_NAME = UserDBHelper.TABLE_NAME;
    // 访问内容提供器的URI
    public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITIES + "/user");
    // 下面是该表的各个字段名称
    public static final String USER_NAME = "name";
    public static final String USER_AGE = "age";
    public static final String USER_HEIGHT = "height";
    public static final String USER_WEIGHT = "weight";
}
```

3. 通过右键菜单创建内容提供器

右击App模块的包名目录，在弹出的右键菜单中依次选择New→Other→Content Provider，打开如图6-25所示的组件创建对话框。

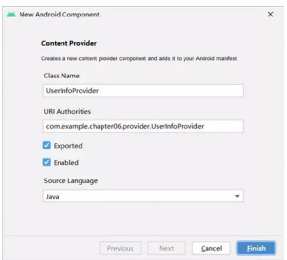


图 6-25 内容提供器的组件创建对话框

在创建对话框的Class Name一栏填写内容提供器的名称，比如UserInfoProvider；在URI Authorities一栏填写URI的授权串，比如“com.example.chapter06.provider.UserInfoProvider”；然后单击对话框右下角的Finish按钮，完成提供器的创建操作。

上述创建过程会自动修改App模块的两处地方，一处是往AndroidManifest.xml添加内容提供器的注册配置，配置信息示例如下：

```
<!-- provider的authorities属性值需要与Java代码的AUTHORITIES保持一致 -->
<provider
    android:name=".provider.UserInfoProvider"
    android:authorities="com.example.chapter06.provider.UserInfoProvider"
    android:enabled="true"
    android:exported="true" />
```

另一处是在包名目录下生成名为UserInfoProvider.java的代码文件，打开一看发现该类继承了

ContentProvider，并且提示重写onCreate、insert、delete、query、update、getType等方法，以便对数据进行增删改查等操作。这个提供器代码显然只有一个框架，还需补充详细的实现代码，为此重写onCreate方法，在此获取用户信息表的数据库帮助器实例，其他insert、delete、query等方法也要加入对应的数据库操作代码，修改之后的内容提供器代码如下：

（完整代码见chapter06\src\main\java\com\example\chapter06\provider\UserInfoProvider.java）

```
public class UserInfoProvider extends ContentProvider {
    private UserDBHelper userDB;                    // 声明一个用户数据库的帮助器对象
    public static final int USER_INFO = 1;          // Uri匹配时的代号
    public static final UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    static {                                        // 往Uri匹配器中添加指定的数据路径
        uriMatcher.addURI(UserInfoContent.AUTHORITIES, "/user", USER_INFO);
    }

    // 创建ContentProvider时调用，可在此获取具体的数据库帮助器实例
    public boolean onCreate() {
        userDB = UserDBHelper.getInstance(getContext(), 1);
        return true;
    }

    // 插入数据
    public Uri insert(Uri uri, ContentValues values) {
        if (uriMatcher.match(uri) == USER_INFO) {     // 匹配到了用户信息表
            // 获取SQLite数据库的写连接
            SQLiteDatabase db = userDB.getWritableDatabase();
            // 向指定的表插入数据，返回记录的行号
            long rowId = db.insert(UserInfoContent.TABLE_NAME, null, values);
            if (rowId > 0) {                          // 判断插入是否执行成功
                // 如果添加成功，就利用新记录的行号生成新的地址
                Uri newUri = ContentUris.withAppendedId(UserInfoContent. CONTENT_URI,
                        rowId);
                // 通知监听器，数据已经改变
                getContext().getContentResolver().notifyChange(newUri, null);
            }
            db.close();                               // 关闭SQLite数据库连接
        }
        return uri;
    }

    // 根据指定条件删除数据
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        int count = 0;
        if (uriMatcher.match(uri) == USER_INFO) {     // 匹配到了用户信息表
            // 获取SQLite数据库的写连接
            SQLiteDatabase db = userDB.getWritableDatabase();
            // 执行SQLite的删除操作，并返回删除记录的数目
            count = db.delete(UserInfoContent.TABLE_NAME, selection, selectionArgs);
            db.close();                               // 关闭SQLite数据库连接
        }
        return count;
    }

    // 根据指定条件查询数据库
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {
        Cursor cursor = null;
        if (uriMatcher.match(uri) == USER_INFO) {     // 匹配到了用户信息表
            // 获取SQLite数据库的读连接
            SQLiteDatabase db = userDB.getReadableDatabase();
            // 执行SQLite的查询操作
            cursor = db.query(UserInfoContent.TABLE_NAME,
```

```
                projection, selection, selectionArgs, null, null, sortOrder);
            // 设置内容解析器的监听
            cursor.setNotificationUri(getContext().getContentResolver(), uri);
        }
        return cursor;                                    // 返回查询结果集的游标
    }

    // 获取Uri支持的数据类型，暂未实现
    public String getType(Uri uri) {}

    // 更新数据，暂未实现
    public int update(Uri uri, ContentValues values, String selection, String[]
selectionArgs) {}
}
```

经过以上3个步骤之后，便完成了服务端App的接口封装工作，接下来再由其他App去访问服务端App的数据。

6.5.2 通过 ContentResolver 访问数据

上一小节提到了利用ContentProvider封装服务端App的数据，如果客户端App想访问对方的内部数据，就要借助内容解析器ContentResolver。内容解析器是客户端App操作服务端数据的工具，与之对应的内容提供器则是服务端的数据接口。在活动代码中调用getContentResolver方法，即可获取内容解析器的实例。

ContentResolver提供的方法与ContentProvider一一对应，比如insert、delete、query、update、getType等，甚至连方法的参数类型都雷同。以添加操作为例，针对前面UserInfoProvider提供的数据接口，下面由内容解析器调用insert方法，使之往内容提供器中插入一条用户信息，记录添加代码如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\ContentWriteActivity.java)
// 添加一条用户记录
private void addUser(UserInfo user) {
    ContentValues name = new ContentValues();
    name.put("name", user.name);
    name.put("age", user.age);
    name.put("height", user.height);
    name.put("weight", user.weight);
    name.put("married", 0);
    name.put("update_time", DateUtil.getNowDateTime(""));
    // 通过内容解析器往指定Uri添加用户信息
    getContentResolver().insert(UserInfoContent.CONTENT_URI, name);
```

至于删除操作就更简单了，只要下面一行代码就删除了所有记录：

```
getContentResolver().delete(UserInfoContent.CONTENT_URI, "1=1", null);
```

查询操作稍微复杂一些，调用query方法会返回游标对象，这个游标正是SQLite的游标Cursor，详细用法参见本章的“6.2.3 数据库帮助器SQLiteOpenHelper”。query方法的参数有好几个，具体说明如下（依参数顺序排列）：

- uri：Uri类型，指定本次操作的数据表路径。
- projection：字符串数组类型，指定将要查询的字段名称列表。
- selection：字符串类型，指定查询条件。
- selectionArgs：字符串数组类型，指定查询条件中的参数取值列表。

- sortOrder：字符串类型，指定排序条件。

下面是调用query方法从内容提供器查询所有用户信息的代码例子：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\ContentReadActivity.java)
// 显示所有的用户记录
private void showAllUser() {
    List<UserInfo> userList = new ArrayList<UserInfo>();
    // 通过内容解析器从指定Uri中获取用户记录的游标
    Cursor cursor = getContentResolver().query(
            UserInfoContent.CONTENT URI, null, null, null, null);
    // 循环取出游标指向的每条用户记录
    while (cursor.moveToNext()) {
        UserInfo user = new UserInfo();
        user.name = cursor.getString(
                cursor.getColumnIndex(UserInfoContent.USER NAME));
        user.age = cursor.getInt(
                cursor.getColumnIndex(UserInfoContent.USER_AGE));
        user.height = cursor.getInt(
                cursor.getColumnIndex(UserInfoContent.USER HEIGHT));
        user.weight = cursor.getFloat(
                cursor.getColumnIndex(UserInfoContent.USER WEIGHT));
        userList.add(user);                                  // 添加到用户信息列表
    }
    cursor.close();                                          // 关闭数据库游标
    String contactCount = String.format("当前共找到%d个用户", userList.size());
    tv_desc.setText(contactCount);
    ll_list.removeAllViews();                                // 移除线性布局下面的所有下级视图
    for (UserInfo user : userList) {                         // 遍历用户信息列表
        String contactDesc = String.format("姓名为%s，年龄为%d，身高为%d，体重为%f\n",
user.name, user.age, user.height, user.weight);
        TextView tv contact = new TextView(this);            // 创建一个文本视图
        tv_contact.setText(contactDesc);
        ll_list.addView(tv_contact);                         // 把文本视图添加至线性布局
    }
}
```

接下来分别演示通过内容解析器添加和查询用户信息的过程，其中记录添加页面为ContentWriteActivity.java，记录查询页面为ContentReadActivity.java。运行测试App，先打开记录添加页面，输入用户信息后点击“添加用户信息”按钮，由内容解析器执行插入操作，此时添加界面如图6-26所示。接着打开记录查询页面，内容解析器自动执行查询操作，并将查到的用户信息一一显示出来，此时查询界面如图6-27所示。



图 6-26 通过内容解析器添加用户信息



图 6-27 通过内容解析器查询用户信息

对比添加页面和查询页面的用户信息，可知成功查到了新增的用户记录。

6.5.3 利用 ContentResolver 读写联系人

在实际开发中，普通App很少会开放数据接口给其他应用访问，作为服务端接口的ContentProvider基本用不到。内容组件能够派上用场的情况，往往是App想要访问系统应用的通讯数据，比如查看联系人、短信、通话记录，以及对这些通讯数据进行增删改查。

在访问系统的通讯数据之前，得先在AndroidManifest.xml中添加相应的权限配置，常见的通讯权限配置主要有下面几个：

```
<!-- 联系人/通讯录。包括读联系人、写联系人 -->
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.WRITE_CONTACTS" />
<!-- 短信。包括发送短信、接收短信、读短信-->
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.READ_SMS" />
<!-- 通话记录。包括读通话记录、写通话记录 -->
<uses-permission android:name="android.permission.READ_CALL_LOG" />
<uses-permission android:name="android.permission.WRITE_CALL_LOG" />
```

当然，从Android 6.0开始，上述的通讯权限默认是关闭的，必须在运行App的时候动态申请相关权限，详细的权限申请过程参见本章的“6.3.3 运行时动态申请权限”。

尽管系统允许App通过内容解析器修改联系人列表，但操作过程比较烦琐，因为一个联系人可能有多个电话号码，还可能有多个邮箱。所以系统通讯录将其设计为3张表，分别是联系人基本信息表、联系号码表、联系邮箱表，于是每添加一位联系人，就要调用至少三次insert方法。下面是往手机通讯录中添加联系人信息的代码例子：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\util\CommunicationUtil.java)
// 往手机通讯录中添加一个联系人信息（包括姓名、电话号码、电子邮箱）
public static void addContacts(ContentResolver resolver, Contact contact) {
    // 构建一个指向系统联系人提供器的Uri对象
    Uri raw_uri = Uri.parse("content://com.android.contacts/raw_contacts");
    ContentValues values = new ContentValues();         // 创建新的配对
    // 往 raw contacts中添加联系人记录，并获取添加后的联系人编号
    long contactId = ContentUris.parseId(resolver.insert(raw uri, values));
    // 构建一个指向系统联系人数据的Uri对象
    Uri uri = Uri.parse("content:                       //com.android.contacts/data");
    ContentValues name = new ContentValues();           // 创建新的配对
    name.put("raw contact id", contactId);              // 往配对中添加联系人编号
    // 往配对中添加“姓名”的数据类型
    name.put("mimetype", "vnd.android.cursor.item/name");
    name.put("data2", contact.name);                    // 往配对中添加联系人的姓名
    resolver.insert(uri, name);                         // 往提供器中添加联系人的姓名记录
    ContentValues phone = new ContentValues();          // 创建新的配对
    phone.put("raw contact id", contactId);             // 往配对中添加联系人编号
    // 往配对中添加“电话号码”的数据类型
    phone.put("mimetype", "vnd.android.cursor.item/phone_v2");
    phone.put("data1", contact.phone);                  // 往配对中添加联系人的电话号码
    phone.put("data2", "2");                            // 联系类型。1表示家庭，2表示工作
    resolver.insert(uri, phone);                        // 往提供器中添加联系人的号码记录
    ContentValues email = new ContentValues();          // 创建新的配对
    email.put("raw_contact_id", contactId);             // 往配对中添加联系人编号
    // 往配对中添加“电子邮箱”的数据类型
    email.put("mimetype", "vnd.android.cursor.item/email_v2");
    email.put("data1", contact.email);                  // 往配对中添加联系人的电子邮箱
    email.put("data2", "2");                            // 联系类型。1表示家庭，2表示工作
```

```
        resolver.insert(uri, email);                    // 往提供器中添加联系人的邮箱记录
    }
```

同理，联系人读取代码也分成3个步骤，先查出联系人的基本信息，再查询联系人号码，最后查询联系人邮箱，详细代码参见CommunicationUtil.java的readAllContacts方法。

接下来演示联系人信息的访问过程。分别创建联系人的添加页面和查询页面，其中添加页面的完整代码见chapter06\src\main\java\com\example\chapter06\ContactAddActivity.java，查询页面的完整代码见chapter06\src\main\java\com\example\chapter06\ContactReadActivity.java。首先在添加页面输入联系人信息，点击“添加联系人”按钮调用addContacts方法写入联系人数据，此时添加界面如图6-28所示。然后打开联系人查询页面，App自动调用readAllContacts方法查出所有的联系人，并显示联系人列表，如图6-29所示，可见刚才添加的联系人已经成功写入系统的联系人列表，而且也能正确读取最新的联系人信息。



图 6-28　联系人的添加界面



图 6-29　联系人的查询界面

6.5.4　利用 ContentObserver 监听短信

ContentResolver获取数据采用的是主动查询方式，有查询就有数据，没查询就没数据。然而有时不但要获取以往的数据，还要实时获取新增的数据，最常见的业务场景是短信验证码。电商App经常在用户注册或付款时发送验证码短信，为了替用户省事，App通常会监控手机刚收到的短信验证码，并自动填写验证码输入框。这时就用到了内容观察器ContentObserver，事先给目标内容注册一个观察器，目标内容的数据一旦发生变化，就马上触发观察器的监听事件，从而执行开发者预先定义的代码。

内容观察器的用法与内容提供器类似，也要从ContentObserver派生一个新的观察器，然后通过ContentResolver对象调用相应的方法注册或注销观察器。下面是内容解析器与内容观察器之间的交互方法说明。

- registerContentObserver：**内容解析器要注册内容观察器。**
- unregisterContentObserver：**内容解析器要注销内容观察器。**
- notifyChange：**通知内容观察器发生了数据变化，此时会触发观察器的**onChange**方法。**notifyChange**的调用时机参见“**6.5.1　通过**ContentProvider**封装数据”的**insert**代码。**

为了让读者更好理解，下面举一个实际应用的例子。手机号码的每月流量限额由移动运营商指定，以中国移动为例，只要将流量校准短信发给运营商客服号码（如发送18到10086），运营商就会回复用户本月的流量数据，包括月流量额度、已使用流量、未使用流量等信息。手机App只需监控10086发来的短信内容，即可自动获取当前号码的流量详情。

下面是利用内容观察器实现流量校准的关键代码片段：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\MonitorSmsActivity.java)
private Handler mHandler = new Handler();       // 声明一个处理器对象
private SmsGetObserver mObserver;               // 声明一个短信获取的观察器对象
private static Uri mSmsUri;                     // 声明一个系统短信提供器的Uri对象
private static String[] mSmsColumn;             // 声明一个短信记录的字段数组

// 初始化短信观察器
private void initSmsObserver() {
    mSmsUri = Uri.parse("content://sms");       // 短信数据的提供器路径
    // 短信记录的字段数组
    mSmsColumn = new String[]{"address", "body", "date"};
    // 创建一个短信观察器对象
    mObserver = new SmsGetObserver(this, mHandler);
    // 给指定Uri注册内容观察器，一旦发生数据变化，就触发观察器的onChange方法
    getContentResolver().registerContentObserver(mSmsUri, true, mObserver);
}

// 在页面销毁时触发
protected void onDestroy() {
    super.onDestroy();
    getContentResolver().unregisterContentObserver(mObserver);       //注销内容观察器
}

// 定义一个短信获取的观察器
private static class SmsGetObserver extends ContentObserver {
    private Context mContext;                                                  // 声明一个上下文对象
    public SmsGetObserver(Context context, Handler handler) {
       super(handler);
        mContext = context;
    }

    // 观察到短信的内容提供器发生变化时触发
    public void onChange(boolean selfChange) {
        String sender = "", content = "";
        // 构建一个查询短信的条件语句，移动号码要查找10086发来的短信
        // 查找最近一分钟的短信
        String selection = String.format("address='10086' and date>%d",
                System.currentTimeMillis() - 1000 * 60 * 1);
        // 通过内容解析器获取符合条件的结果集游标
        Cursor cursor = mContext.getContentResolver().query(
                mSmsUri, mSmsColumn, selection, null, " date desc");
        // 循环取出游标所指向的所有短信记录
        while (cursor.moveToNext()) {
            sender = cursor.getString(0);               // 短信的发送号码
            content = cursor.getString(1);              // 短信内容
            break;
        }
        cursor.close();                                 // 关闭数据库游标
        mCheckResult = String.format("发送号码：%s\n短信内容：%s", sender, content);
        // 依次解析流量校准短信里面的各项流量数值，并拼接流量校准的结果字符串
        String flow = String.format("流量校准结果如下：总流量为：%s；已使用：%s" +"；剩余流量：%s",
                    findFlow(content, "总流量为"),
                findFlow(content, "已使用"), findFlow(content, "剩余"));
        if (tv_check_flow != null) {                    // 离开该页面后就不再显示流量信息
            tv_check_flow.setText(flow);                // 在文本视图中显示流量校准结果
        }
        super.onChange(selfChange);
    }
}
```

运行测试App，点击校准按钮发送流量校准短信，接着收到如图6-30所示的短信内容。同时App监听刚收到的流量短信，从中解析得到当前的流量数值，并展示在界面上，如图6-31所示。可见通过内容观察器实时获取了最新的短信记录。

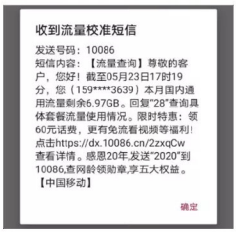


图 6-30　用户收到的短信内容



图 6-31　内容观察器监听短信并解析出流量信息

总结一下系统开放给普通应用访问的常用URI，详细的URI取值说明见表6-2。

表 6-2　常用的系统 URI 取值说明

内容名称	URI 常量名	实际路径
联系人基本信息	ContactsContract.Contacts.CONTENT_URI	content://com.android.contacts/contacts
联系人电话号码	ContactsContract.CommonDataKinds.Phone.CONTENT_URI	content://com.android.contacts/data/phones
联系人邮箱	ContactsContract.CommonDataKinds.Email.CONTENT_URI	content://com.android.contacts/data/emails
短信	Telephony.Sms.CONTENT_URI	content://sms
彩信	Telephony.Mms.CONTENT_URI	content://mms
通话记录	CallLog.Calls.CONTENT_URI	content://call_log/calls

6.6　实战项目：购物车

购物车的应用面很广，凡是电商App都可以看到它的身影，之所以选择购物车作为本章的实战项目，除了它使用广泛的特点之外，更因为它用到了多种存储方式。现在就让我们开启电商购物车的体验之旅吧。

6.6.1　需求描述

电商App的购物车可谓是司空见惯了，以京东商城的购物车为例，一开始没有添加任何商品，此时空购物车如图6-32所示，而且提示去逛秒杀商场；加入几件商品之后，购物车页面如图6-33所示。



图 6-32 京东 App 购物车的初始页面



图 6-33 在京东 App 购物车中加了几件商品

可见购物车除了底部有个结算行，其余部分主要是已加入购物车的商品列表，然后每个商品行左边是商品小图，右边是商品名称及其价格。

据此仿照出本项目的购物车功能，第一次进入购物车页面，购物车里面是空的，同时提示去逛手机商场，如图6-34所示。接着去商场页面选购手机，随便挑了几部手机加入购物车，再返回购物车页面，即可看到购物车的商品列表，如图6-35所示，有商品图片、名称、数量、单价、总价等信息。当然购物车并不仅仅只是展示待购买的商品，还要支持最终购买的结算操作、支持清空购物车等功能。



图 6-34 首次打开购物车页面



图 6-35 选购商品后的购物车

购物车的存在感很强，不仅仅在购物车页面才能看到购物车，往往在商场页面，甚至商品详情页面，都会看到某个角落冒出的购物车图标。一旦有新商品加入购物车，购物车图标上的商品数量立马加1。当然，用户也能点击购物车图标直接跳转到购物车页面。商场页面除了商品列表之外，页面右上角还有一个购物车图标，如图6-36所示，有时这个图标会在页面右下角。商品详情页面通

常也有购物车图标，如图6-37所示，倘使用户在详情页面把商品加入购物车，那么图标上的数字也会加1。



图 6-36 手机商场页面



图 6-37 手机详情页面

至此大概过了一遍购物车需要实现的基本功能，提需求总是很简单的，真正落到实处还得开发者发挥想象力，把购物车做成一个功能完备的模块。

6.6.2 界面设计

首先找找看，购物车使用了哪些Android控件：

- **线性布局**LinearLayout：**购物车界面从上往下排列，用到了垂直方向的线性布局。**
- **网格布局**GridLayout：**商场页面的陈列橱柜，允许分行分列展示商品。**
- **相对布局**RelativeLayout：**页面右上角的购物车图标，图标右上角又有数字标记，按照指定方位排列控件正是相对布局的拿手好戏。**
- **其他常见控件尚有文本视图**TextView、**图像视图**ImageView、**按钮控件**Button**等。**

然后考虑一下购物车的存储功能，到底采取了哪些存储方式：

- **数据库**SQLite：**最直观的肯定是数据库了，购物车里的商品列表一定是放在**SQLite**中，增删改查都少不了它。**
- **全局内存：购物车图标右上角的数字表示购物车中的商品数量，该数值建议保存在全局内存中，这样不必每次都到数据库中执行**count**操作。**
- **存储卡文件：通常商品图片来自电商平台的服务器，此时往往引入图片缓存机制，也就是首次访问时先将网络图片保存到存储卡中，下次访问时直接从存储卡中获取缓存图片，从而提高图片的加载速度。**
- **共享参数**SharedPreferences：**是否首次访问网络图片，这个标志位推荐放在共享参数中，因为它需要持久化存储，并且只有一个参数信息。**

真是想不到，一个小小的购物车，竟然用到了好几种存储方式。

6.6.3 关键代码

为了让读者更好更快地完成购物车项目，下面列举几个重要功能的代码片段。

1. 关于页面跳转

因为购物车页面允许直接跳转到商场页面，并且商场页面也允许跳转到购物车页面，所以如果用户在这两个页面之间来回跳转，然后再按返回键，结果发现返回的时候也是在这两个页面间往返跳转。出现问题的缘由在于：每次启动活动页面都往活动栈中加入一个新活动，那么返回出栈之时，也只好一个一个活动依次退出了。

解决该问题的办法参见第4章的“4.1.3 Activity的启动模式”，对于购物车的活动跳转需要指定启动标志FLAG_ACTIVITY_CLEAR_TOP，表示活动栈有且仅有该页面的唯一实例，如此即可避免多次返回同一页面的情况。比如从购物车页面跳转到商场页面，此时活动跳转的代码示例如下：

```
// 从购物车页面跳转到商场页面
Intent intent = new Intent(this, ShoppingChannelActivity.class);
intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);          // 设置启动标志
startActivity(intent);                                    // 跳转到手机商场页面
```

又如从商场页面跳转到购物车页面，此时活动跳转的代码示例如下：

```
// 从商场页面跳转到购物车页面
Intent intent = new Intent(this, ShoppingCartActivity.class);
intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);          // 设置启动标志
startActivity(intent);                                    // 跳转到购物车页面
```

2. 关于商品图片的缓存

通常商品图片由后端服务器提供，App打开页面时再从服务器下载所需的商品图。可是购物车模块的多个页面都会展示商品图片，如果每次都到服务器请求图片，显然既耗时间又耗流量，非常不经济。因此App都会缓存常用的图片，一旦从服务器成功下载图片，便在手机存储卡上保存图片文件。然后下次界面需要加载商品图片时，就先从存储卡中寻找该图片，如果找到就读取图片的位图信息，如果没找到就再到服务器下载图片。

以上的缓存逻辑是最简单的二级图片缓存，实际开发往往使用更高级的三级缓存机制，即“运行内存→存储卡→网络下载”。当然就初学者而言，先从掌握最简单的二级缓存开始，也就是“存储卡→网络下载”。按照二级缓存机制，可以设计以下的缓存处理逻辑：

（1）先判断是否为首次访问网络图片。

（2）如果是首次访问网络图片，就先从网络服务器下载图片。

（3）把下载完的图片数据保存到手机的存储卡中。

（4）往数据库中写入商品记录，以及商品图片的本地存储路径。

（5）更新共享参数中的首次访问标志。

按照上述的处理逻辑，编写的图片加载代码示例如下：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\ShoppingCartActivity.java)
private String mFirst = "true";  // 是否首次打开
// 模拟网络数据，初始化数据库中的商品信息
private void downloadGoods() {
    // 获取共享参数保存的是否首次打开参数
```

```
        mFirst = SharedUtil.getIntance(this).readString("first", "true");
        // 获取当前App的私有下载路径
        String path = getExternalFilesDir(Environment.DIRECTORY_DOWNLOADS).toString()+"/";
        if (mFirst.equals("true")) {                    // 如果是首次打开
            // 模拟网络图片下载
            ArrayList<GoodsInfo> goodsList = GoodsInfo.getDefaultList();
            for (int i = 0; i < goodsList.size(); i++) {
                GoodsInfo info = goodsList.get(i);
                // 往商品数据库中插入一条该商品的记录
                long rowid = mGoodsHelper.insert(info);
                info.rowid = rowid;
                Bitmap pic = BitmapFactory.decodeResource(getResources(), info.pic);
                String pic_path = path + rowid + ".jpg";
                FileUtil.saveImage(pic_path, pic);      // 往存储卡中保存商品图片
                pic.recycle();                          // 回收位图对象
                info.pic_path = pic_path;
                mGoodsHelper.update(info);              // 更新商品数据库中该商品记录的图片路径
            }
        }
        // 把是否首次打开写入共享参数
        SharedUtil.getIntance(this).writeString("first", "false");
    }
```

3. 关于各页面共同的标题栏

注意到购物车、手机商场、手机详情三个页面顶部都有标题栏，而且这三个标题栏风格统一，既然如此，能否把它做成公共的标题栏呢？当然App界面支持局部的公共布局，以购物车的标题栏为例，公共布局的实现过程包括以下两个步骤：

01 定义标题栏专用的布局文件，包含返回箭头、文字标题、购物车图标、商品数量表等，具体内容如下：

```
(完整代码见chapter06\src\main\res\layout\title_shopping.xml)
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout height="50dp"
    android:background="#aaaaff" >
    <ImageView
        android:id="@+id/iv_back"
        android:layout_width="50dp"
        android:layout height="match parent"
        android:layout alignParentLeft="true"
        android:padding="10dp"
        android:scaleType="fitCenter"
        android:src="@drawable/ic_back" />
    <TextView
        android:id="@+id/tv title"
        android:layout width="wrap content"
        android:layout height="match parent"
        android:layout_centerInParent="true"
        android:gravity="center" />
    <ImageView
        android:id="@+id/iv cart"
        android:layout_width="50dp"
        android:layout_height="match_parent"
        android:layout_alignParentRight="true"
        android:src="@drawable/cart" />
    <TextView
        android:id="@+id/tv count"
        android:layout_width="20dp"
```

```
        android:layout_height="20dp"
        android:layout alignParentTop="true"
        android:layout toRightOf="@+id/iv cart"
        android:layout marginLeft="-20dp"
        android:gravity="center"
        android:background="@drawable/shape_oval_red"
        android:text="0" />
</RelativeLayout>
```

02 在购物车页面的布局文件中添加如下一行include标签，表示引入title_shopping.xml的布局内容：

```
(完整代码见chapter06\src\main\res\layout\activity shopping cart.xml)
<include layout="@layout/title_shopping" />
```

之后重新运行测试App，即可发现购物车页面的顶部果然出现了公共标题栏，商场页面、详情页面的公共标题栏可参考购物车页面的include标签。

4. 关于商品网格的单元布局

商场页面的商品列表，呈现三行二列的表格布局，每个表格单元的界面布局雷同，都是商品名称在上、商品图片居中、商品价格与添加按钮在下，看起来跟公共标题栏的处理有些类似。但后者为多个页面引用同一个标题栏，是多对一的关系；而前者为一个商场页面引用了多个商品网格，是一对多的关系。因此二者的实现过程不尽相同，就商场网格而言，它的单元复用分为下列3个步骤：

01 在商场页面的布局文件中添加GridLayout节点，示例代码如下：

```
(完整代码见chapter06\src\main\res\layout\activity shopping channel.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <include layout="@layout/title_shopping" />
    <ScrollView
        android:layout width="match parent"
        android:layout height="wrap content" >
        <GridLayout
            android:id="@+id/gl channel"
            android:layout width="match parent"
            android:layout height="wrap content"
            android:columnCount="2" />
    </ScrollView>
</LinearLayout>
```

02 为商场网格编写统一的商品信息布局，XML文件内容示例如下：

```
(完整代码见chapter06\src\main\res\layout\item_goods.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/ll item"
    android:layout width="match parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:id="@+id/tv name"
        android:layout width="match parent"
        android:layout height="wrap content"
        android:gravity="center" />
    <ImageView
        android:id="@+id/iv thumb"
        android:layout_width="180dp"
```

```
        android:layout_height="150dp" />
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="45dp"
        android:orientation="horizontal">
        <TextView
            android:id="@+id/tv_price"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="2"
            android:gravity="center" />
        <Button
            android:id="@+id/btn_add"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="3"
            android:gravity="center"
            android:text="加入购物车" />
    </LinearLayout>
</LinearLayout>
```

08 在商场页面的Java代码中，先利用下面代码获取布局文件item_goods.xml的根视图：

```
View view = LayoutInflater.from(this).inflate(R.layout.item_goods, null);
```

再从根视图中依据控件ID分别取出网格单元的各控件对象：

```
ImageView iv_thumb = view.findViewById(R.id.iv_thumb);
TextView tv_name = view.findViewById(R.id.tv_name);
TextView tv_price = view.findViewById(R.id.tv_price);
Button btn_add = view.findViewById(R.id.btn_add);
```

然后就能按照寻常方式操纵这些控件对象了，下面便是给网格布局加载商品的代码例子：

```
(完整代码见chapter06\src\main\java\com\example\chapter06\ShoppingChannelActivity.java)
private void showGoods() {
    gl_channel.removeAllViews();  // 移除下面的所有子视图
    // 查询商品数据库中的所有商品记录
    List<GoodsInfo> goodsArray = mGoodsHelper.query("1=1");
    for (final GoodsInfo info : goodsArray) {
        // 获取布局文件item_goods.xml的根视图
        View view = LayoutInflater.from(this).inflate(R.layout.item_goods, null);
        ImageView iv_thumb = view.findViewById(R.id.iv_thumb);
        TextView tv_name = view.findViewById(R.id.tv_name);
        TextView tv_price = view.findViewById(R.id.tv_price);
        Button btn_add = view.findViewById(R.id.btn_add);
        tv_name.setText(info.name);                          // 设置商品名称
        iv_thumb.setImageURI(Uri.parse(info.pic_path));    // 设置商品图片
        iv_thumb.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(
                    ShoppingChannelActivity.this,
                    ShoppingDetailActivity.class);
                intent.putExtra("goods_id", info.rowid);
                startActivity(intent);                       // 跳转到商品详情页面
            }
        });
        tv_price.setText("" + (int)info.price);              // 设置商品价格
        btn_add.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
```

```
                addToCart(info.rowid, info.name);          // 添加到购物车
            }
        });
        gl_channel.addView(view);                          // 把商品视图添加到网格布局
    }
}
```

弄好了商场页面的网格单元，购物车页面的商品行也可照此办理，不同之处在于购物车页面的商品行使用线性布局而非网格布局，其余实现过程依然分成上述3个步骤。

6.7 小 结

本章主要介绍了Android常用的4种数据存储方式，包括键值对存取的两种使用方式（共享参数和数据仓库）、数据库SQLite的关系型数据存取、存储卡的文件读写操作（含文本文件读写和图片文件读写）、App全局内存的读写，以及组件Application的生命周期及其用法，还有内容组件ContentProvider的常见用法（通过ContentProvider封装数据、通过ContentResolver访问数据、利用ContentResolver读写联系人、利用ContentObserver监听短信）。最后设计了一个实战项目“购物车”，通过该项目的编码进一步复习并巩固4种存储方式的使用。

通过本章的学习，读者应该能够掌握以下6种开发技能：

（1）学会分别使用共享参数和数据仓库存取键值对数据。
（2）学会使用SQLite存取数据库记录。
（3）学会使用存储卡读写文本文件和图片文件。
（4）学会在App运行过程中动态申请权限。
（5）学会应用组件Application的用法。
（6）学会利用ContentProvider在应用之间共享数据。

6.8 动手练习

1. 请上机实验完善找回密码项目的记住密码功能，分别采用以下两种存储方式：

（1）使用共享参数记住上次登录成功时输入的用户名和密码。
（2）使用SQLite数据库记住用户名对应的密码，也就是根据用户名自动填写密码。

2. 请上机实验本章的购物车项目，要求实现下列功能：

（1）往购物车中添加商品。
（2）自动计算购物车中所有商品的总金额。
（3）移除购物车里的某个商品。
（4）清空购物车。

第 7 章

高 级 控 件

本章介绍App开发常用的一些高级控件用法，主要包括：如何使用下拉框及其适配器、如何使用列表类视图及其适配器、如何使用翻页类视图及其适配器、如何使用碎片及其适配器等。然后结合本章所学的知识，演示了一个实战项目“记账本”的设计与实现。

7.1 下 拉 框

本节介绍下拉框的用法以及适配器的基本概念，结合对下拉框Spinner的使用说明分别阐述数组适配器ArrayAdapter、简单适配器SimpleAdapter的具体用法与展示效果。

7.1.1 下拉框控件 Spinner

Spinner是下拉框控件，它用于从一串列表中选择某项，其功能类似于单选按钮的组合。下拉列表的展示方式有两种，一种是在当前下拉框的正下方弹出列表框，此时要把spinnerMode属性设置为dropdown，下面是XML文件中采取下拉模式的Spinner标签例子：

```
<Spinner
    android:id="@+id/sp_dropdown"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:spinnerMode="dropdown" />"
```

另一种是在页面中部弹出列表对话框，此时要把spinnerMode属性设置为dialog，下面是XML文件中采取对话框模式的Spinner标签例子：

```
<Spinner
    android:id="@+id/sp_dialog"
    android:layout width="match parent"
    android:layout height="wrap content"
    android:spinnerMode="dialog" />"
```

此外，在Java代码中，Spinner还可以调用下列4个方法。

- setPrompt：**设置标题文字。注意对话框模式才显示标题，下拉模式不显示标题。**
- setAdapter：**设置列表项的数据适配器。**
- setSelection：**设置当前选中哪项。注意该方法要在**setAdapter**方法后调用。**

- setOnItemSelectedListener：设置下拉列表的选择监听器，该监听器要实现接口OnItemSelectedListener。

下面是初始化下拉框，并设置选择监听器的代码例子：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\SpinnerDropdownActivity.java)
// 初始化下拉模式的列表框
private void initSpinnerForDropdown() {
    // 声明一个下拉框的数组适配器
    ArrayAdapter<String> starAdapter = new ArrayAdapter<String>(this,
            R.layout.item select, starArray);
    // 从布局文件中获取名为sp dropdown的下拉框
    Spinner sp dropdown = findViewById(R.id.sp dropdown);
    // 设置下拉框的标题。对话框模式才显示标题，下拉模式不显示标题
    sp_dropdown.setPrompt("请选择行星");
    sp dropdown.setAdapter(starAdapter);     // 设置下拉框的数组适配器
    sp dropdown.setSelection(0);             // 设置下拉框默认显示第一项
    // 给下拉框设置选择监听器，一旦用户选中某一项，就触发监听器的onItemSelected方法
    sp_dropdown.setOnItemSelectedListener(new MySelectedListener());
}

// 定义下拉列表需要显示的文本数组
private String[] starArray = {"水星", "金星", "地球", "火星", "木星", "土星"};
// 定义一个选择监听器，它实现了接口OnItemSelectedListener
class MySelectedListener implements OnItemSelectedListener {
    // 选择事件的处理方法，其中arg2代表选择项的序号
    public void onItemSelected(AdapterView<?> arg0, View arg1, int arg2, long arg3) {
            Toast.makeText(SpinnerDropdownActivity.this,
            "您选择的是" + starArray[arg2], Toast.LENGTH_LONG).show();
    }
    // 未选择时的处理方法，通常无须关注
    public void onNothingSelected(AdapterView<?> arg0) {}
}
```

接下来观察两种下拉列表的界面效果，运行测试App，一开始的下拉框如图7-1所示。

在下拉模式页面（SpinnerDropdownActivity.java）单击下拉框，六大行星的列表框在下拉框正下方展开，如图7-2所示。点击某项后，列表框消失，同时下拉框中的文字变为刚选中的行星名称。再打开对话框模式页面（SpinnerDialogActivity），单击下拉框会在页面中央弹出六大行星的列表对话框，如图7-3所示。点击某项后，对话框消失，同时下拉框中的文字也变为刚选中的行星名称。



图 7-1　下拉框控件的初始界面

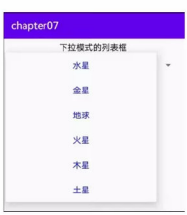


图 7-2　下拉模式的列表框

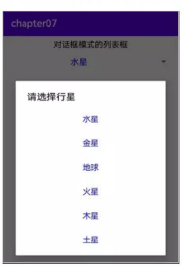


图 7-3　对话框模式的列表框

7.1.2 数组适配器 ArrayAdapter

上一小节在演示下拉框控件时，调用了setAdapter方法设置列表适配器。这个适配器好比一组数据的加工流水线，你丢给它一大把糖果（六大行星的原始数据），适配器先按顺序排列糖果（对应行星数组starArray），然后拿来制作好的包装盒（对应每个列表项的布局文件item_select.xml），把糖果往里面一塞，出来的便是一个个精美的糖果盒（界面上排布整齐的列表框）。这个流水线可以做得很复杂，也可以做得简单一些，最简单的流水线就是之前演示用到的数组适配器ArrayAdapter。

ArrayAdapter主要用于每行列表只展示文本的情况，实现过程分成下列3个步骤：

01 编写列表项的XML文件，内部布局只有一个TextView标签，示例如下：

```
(完整代码见chapter07\src\main\res\layout\item select.xml)
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout height="50dp"
    android:gravity="center" />
```

02 调用ArrayAdapter的构造方法，填入待展现的字符串数组，以及列表项的包装盒，即XML文件R.layout.item_select。构造方法的调用代码示例如下：

```
// 声明一个下拉框的数组适配器
ArrayAdapter<String> starAdapter = new ArrayAdapter<String>(this,
                        R.layout.item_select, starArray);
```

03 调用下拉框控件的setAdapter方法，传入步骤2得到的适配器实例，代码如下：

```
sp_dropdown.setAdapter(starAdapter);  // 设置下拉框的数组适配器
```

经过以上3个步骤，先由ArrayAdapter明确原料糖果的分拣过程与包装方式，再由下拉框调用setAdapter方法发出开工指令，适配器便会把一个个包装好的糖果盒输出到界面。

7.1.3 简单适配器 SimpleAdapter

ArrayAdapter只能显示文本列表，显然不够美观，有时还想给列表加上图标，比如希望显示六大行星的天文影像。这时简单适配器SimpleAdapter就派上用场了，它允许在列表项中同时展示文本与图片。

SimpleAdapter的实现过程略微复杂，因为它的原料需要更多信息。例如，原料不但有糖果，还有贺卡，这样就得把一大袋糖果和一大袋贺卡送进流水线，适配器每次拿一颗糖果和一张贺卡，把糖果与贺卡按规定塞进包装盒。对于SimpleAdapter的构造方法来说，第2个参数Map容器放的是原料糖果与贺卡，第3个参数放的是包装盒，第4个参数放的是糖果袋与贺卡袋的名称，第5个参数放的是包装盒里塞糖果的位置与塞贺卡的位置。

下面是下拉框控件使用简单适配器的示例代码：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\SpinnerIconActivity.java)
// 初始化下拉框，演示简单适配器
private void initSpinnerForSimpleAdapter() {
    // 声明一个映射对象的列表，用于保存行星的图标与名称配对信息
    List<Map<String, Object>> list = new ArrayList<Map<String, Object>>();
    // iconArray是行星的图标数组，starArray是行星的名称数组
    for (int i = 0; i < iconArray.length; i++) {
        Map<String, Object> item = new HashMap<String, Object>();
```

```
            item.put("icon", iconArray[i]);
            item.put("name", starArray[i]);
            list.add(item);                          // 把行星图标与名称的配对映射添加到列表
        }
        // 声明一个下拉框的简单适配器，其中指定了图标与文本两组数据
        SimpleAdapter starAdapter = new SimpleAdapter(this, list, R.layout.item_simple,
                new String[]{"icon", "name"},new int[]{R.id.iv icon, R.id.tv name});
        // 设置简单适配器的布局样式
        starAdapter.setDropDownViewResource(R.layout.item simple);
        // 从布局文件中获取名为sp_icon的下拉框
        Spinner sp_icon = findViewById(R.id.sp_icon);
        sp icon.setPrompt("请选择行星");              // 设置下拉框的标题
        sp icon.setAdapter(starAdapter);             // 设置下拉框的简单适配器
        sp icon.setSelection(0);                     // 设置下拉框默认显示第一项
        // 给下拉框设置选择监听器，一旦用户选中某一项，就触发监听器的onItemSelected方法
        sp_icon.setOnItemSelectedListener(new MySelectedListener());
    }
```

以上代码中，简单适配器使用的包装盒名为R.layout.item_simple，它的布局内容如下：

```
(完整代码见chapter07\src\main\res\layout\item_simple.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout height="wrap content"
    android:orientation="horizontal">
    <!-- 这是展示行星图标的ImageView -->
    <ImageView
        android:id="@+id/iv icon"
        android:layout width="0dp"
        android:layout height="50dp"
        android:layout_weight="1" />
    <!-- 这是展示行星名称的TextView -->
    <TextView
        android:id="@+id/tv name"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3"
        android:gravity="center" />
</LinearLayout>
```

运行测试App，一开始的下拉框如图7-4所示，可见默认选项既有图标又有文字。然后单击下拉框，页面中央弹出六大行星的列表对话框，如图7-5所示，可见列表框的各项也一齐展示了行星的图标及其名称。

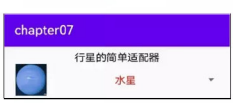


图 7-4　采用简单适配器的初始下拉框

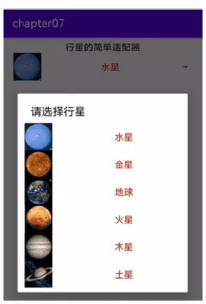


图 7-5　采用简单适配器的列表对话框

7.2 列表类视图

本节介绍列表类视图怎样结合基本适配器展示视图阵列，包括：基本适配器BaseAdapter的用法、列表视图ListView的用法及其常见问题的解决、网格视图GridView的用法及其拉伸模式说明。

7.2.1 基本适配器 BaseAdapter

由上一节的介绍可知，数组适配器适用于纯文本的列表数据，简单适配器适用于带图标的列表数据。然而在实际应用中常常有更复杂的列表，比如每个列表项存在3个以上的控件，这种情况即便是简单适配器也很吃力，而且不易扩展。为此Android提供了一种适应性更强的基本适配器BaseAdapter，该适配器允许开发者在别的代码文件中编写操作代码，大大提高了代码的可读性和可维护性。

从BaseAdapter派生的数据适配器主要实现下面5个方法。

- **构造方法：指定适配器需要处理的数据集合。**
- **getCount：获取列表项的个数。**
- **getItem：获取列表项的数据。**
- **getItemId：获取列表项的编号。**
- **getView：获取每项的展示视图，并对每项的内部控件进行业务处理。**

下面以下拉框控件为载体，演示如何操作BaseAdapter，具体的编码过程分为3步：

01 编写列表项的布局文件，示例代码如下：

```
(完整代码见chapter07\src\main\res\layout\item_list.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <!-- 这是显示行星图片的图像视图 -->
    <ImageView
        android:id="@+id/iv_icon"
        android:layout_width="0dp"
        android:layout_height="80dp"
        android:layout_weight="1"
        android:scaleType="fitCenter" />
    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3"
        android:orientation="vertical">
        <!-- 这是显示行星名称的文本视图 -->
        <TextView
            android:id="@+id/tv_name"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />
        <!-- 这是显示行星描述的文本视图 -->
        <TextView
            android:id="@+id/tv_desc"
```

```
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="2" />
    </LinearLayout>
</LinearLayout>
```

02 写个新的适配器继承BaseAdapter，实现对列表项的管理操作，示例代码如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\adapter\PlanetBaseAdapter.java)
public class PlanetBaseAdapter extends BaseAdapter {
    private Context mContext;                       // 声明一个上下文对象
    private List<Planet> mPlanetList;               // 声明一个行星信息列表
    // 行星适配器的构造方法，传入上下文与行星列表
    public PlanetBaseAdapter(Context context, List<Planet> planet list) {
        mContext = context;
        mPlanetList = planet_list;
    }
    // 获取列表项的个数
    public int getCount() {
        return mPlanetList.size();
    }
    // 获取列表项的数据
    public Object getItem(int arg0) {
        return mPlanetList.get(arg0);
    }
    // 获取列表项的编号
    public long getItemId(int arg0) {
        return arg0;
    }
    // 获取指定位置的列表项视图
    public View getView(final int position, View convertView, ViewGroup parent){
        ViewHolder holder;
        if (convertView == null) {                  // 转换视图为空
            holder = new ViewHolder();              // 创建一个新的视图持有者
            // 根据布局文件item_list.xml生成转换视图对象
            convertView = LayoutInflater.from(mContext).inflate(
                          R.layout.item list, null);
            holder.iv icon = convertView.findViewById(R.id.iv icon);
            holder.tv_name = convertView.findViewById(R.id.tv_name);
            holder.tv_desc = convertView.findViewById(R.id.tv_desc);
            convertView.setTag(holder);             // 将视图持有者保存到转换视图当中
        } else {                                    // 转换视图非空
            // 从转换视图中获取之前保存的视图持有者
            holder = (ViewHolder) convertView.getTag();
        }
        Planet planet = mPlanetList.get(position);
        holder.iv icon.setImageResource(planet.image);     // 显示行星的图片
        holder.tv name.setText(planet.name);               // 显示行星的名称
        holder.tv desc.setText(planet.desc);               // 显示行星的描述
        holder.iv_icon.requestFocus();
        return convertView;
    }
    // 定义一个视图持有者，以便重用列表项的视图资源
    public final class ViewHolder {
        public ImageView iv icon;                          // 声明行星图片的图像视图对象
        public TextView tv_name;                           // 声明行星名称的文本视图对象
        public TextView tv_desc;                           // 声明行星描述的文本视图对象
    }
}
```

08 在页面代码中创建该适配器实例，并交给下拉框设置，示例代码如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\BaseAdapterActivity.java)
// 初始化行星列表的下拉框
private void initPlanetSpinner() {
    // 获取默认的行星列表，即水星、金星、地球、火星、木星、土星
    planetList = Planet.getDefaultList();
    // 构建一个行星列表的适配器
    PlanetBaseAdapter adapter = new PlanetBaseAdapter(this, planetList);
    // 从布局文件中获取名为sp_planet的下拉框
    Spinner sp_planet = findViewById(R.id.sp_planet);
    sp_planet.setPrompt("请选择行星");          // 设置下拉框的标题
    sp_planet.setAdapter(adapter);              // 设置下拉框的列表适配器
    sp_planet.setSelection(0);                  // 设置下拉框默认显示第一项
    // 给下拉框设置选择监听器，一旦用户选中某一项，就触发监听器的onItemSelected方法
    sp_planet.setOnItemSelectedListener(new MySelectedListener());
}
```

运行测试App，一开始的下拉框如图7-6所示，可见默认选项有图标有标题还有内容。然后单击下拉框，页面中央弹出六大行星的列表对话框，如图7-7所示，可见列表框的各项也一齐展示了行星的图标、名称及其详细描述。因为对列表项布局item_list.xml使用了单独的适配器代码PlanetBaseAdapter，所以即使多加几个控件也不怕麻烦了。

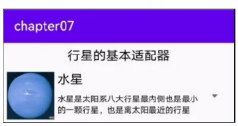


图 7-6 采用基本适配器的初始下拉框



图 7-7 采用基本适配器的列表对话框

7.2.2 列表视图 ListView

上一小节给下拉框控件设置了基本适配器，然而列表效果只在弹出对话框中展示，一旦选中某项，回到页面时又只显示选中的内容。这么丰富的列表信息没展示在页面上实在是可惜，也许用户对好几项内容都感兴趣。若想在页面上直接显示全部列表信息，就要引入新的列表视图ListView。列表视图允许在页面上分行展示相似的数据列表，例如新闻列表、商品列表、图书列表等，方便用户浏览与操作。

ListView同样通过setAdapter方法设置列表项的数据适配器，但在操作列表项时，它不使用setOnItemSelectedListener方法，而是调用setOnItemClickListener方法设置列表项的点击监听器OnItemClickListener，有时也调用setOnItemLongClickListener方法设置列表项的长按监听器OnItemLongClickListener。在点击列表项或者长按列表项之时，即可触发监听器对应的事件处理方法。除此之外，列表视图还新增了几个属性与方法，详细说明见表7-1。

表 7-1　列表视图新增的属性与方法说明

XML 中的属性	ListView 类的设置方法	说　明
divider	setDivider	指定分隔线的图形。如需取消分隔线，可将该属性值设为@null
dividerHeight	setDividerHeight	指定分隔线的高度
listSelector	setSelector	指定列表项的按压背景（状态图形格式）

在XML文件中添加ListView很简单，只要以下几行代码就声明了一个列表视图：

```
<ListView
    android:id="@+id/lv_planet"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

往列表视图中填充数据也很容易，先利用基本适配器实现列表适配器，再调用setAdapter方法设置适配器对象。下面是使用列表视图在界面上展示行星列表的代码例子：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\ListViewActivity.java)
List<Planet> planetList = Planet.getDefaultList();   // 获得默认的行星列表
// 构建一个行星列表的列表适配器
PlanetListAdapter adapter = new PlanetListAdapter(this, planetList);
// 从布局视图中获取名为lv_planet的列表视图
ListView lv_planet = findViewById(R.id.lv_planet);
lv_planet.setAdapter(adapter);                          // 设置列表视图的适配器
lv_planet.setOnItemClickListener(adapter);              // 设置列表视图的点击监听器
lv_planet.setOnItemLongClickListener(adapter);          // 设置列表视图的长按监听器
```

其中列表项的点击事件和长按事件的处理方法代码如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\adapter\PlanetListAdapter.java)
// 处理列表项的点击事件，由接口OnItemClickListener触发
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
    String desc = String.format("您点击了第%d个行星，它的名字是%s", position+1,
            mPlanetList.get(position).name);
    Toast.makeText(mContext, desc, Toast.LENGTH_LONG).show();
}
// 处理列表项的长按事件，由接口OnItemLongClickListener触发
public boolean onItemLongClick(AdapterView<?> parent, View view, int position,
long id) {
    String desc = String.format("您长按了第%d个行星，它的名字是%s", position+1,
            mPlanetList.get(position).name);
    Toast.makeText(mContext, desc, Toast.LENGTH_LONG).show();
    return true;
}
```

运行App后打开包含列表视图的测试页面，行星列表的界面效果如图7-8所示。

从图7-8可见，列表视图在各项之间默认展示灰色的分隔线，点击或长按某项时会显示默认的灰色水波背景。若想修改分隔线样式或按压背景，则需调整ListView的对应属性，调整时候的注意点说明如下：

1. 修改列表视图的分隔线样式

修改分隔线样式要在XML文件中同时设置divider（分隔图片）与dividerHeight（分隔高度）两个属性，并且遵循下列两条规则：

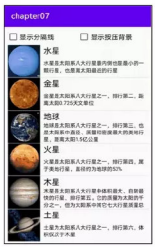


图 7-8　采用基本适配器的列表视图

（1）divider属性设置为@null时，不能再将dividerHeight属性设置为大于0的数值，因为这会导致最后一项没法完全显示，底部有一部分被掩盖了。原因是列表高度为wrap_content时，系统已按照没有分隔线的情况计算列表高度，此时dividerHeight占用了n−1块空白分隔区域，使得最后一项被挤到背影里面去了。

（2）通过代码设置的话，务必先调用setDivider方法再调用setDividerHeight方法。如果先调用setDividerHeight后调用setDivider，分隔线高度就会变成分隔图片的高度，而不是setDividerHeight设置的高度。XML布局文件则不存在divider属性和dividerHeight属性的先后顺序问题。

下面的代码示范了如何在代码中正确设置分隔线，以及如何正确去掉分隔线：

```
(完整代码见ListViewActivity.java的refreshListView方法)
if (ck_divider.isChecked()) {                                  // 显示分隔线
    // 从资源文件获得图形对象
    Drawable drawable = getResources().getDrawable(R.color.red);
    lv_planet.setDivider(drawable);                            // 设置列表视图的分隔线
    lv_planet.setDividerHeight(Utils.dip2px(this, 5));        // 设置分隔线的高度
} else {  // 不显示分隔线
    lv_planet.setDivider(null);                                // 设置列表视图的分隔线
    lv_planet.setDividerHeight(0);                             // 设置列表视图的分隔线高度
}
```

2. 修改列表项的按压背景

若想取消按压列表项之时默认的水波背景，可在布局文件中设置，也可在代码中设置，两种设置方式的注意点说明如下：

（1）在布局文件中取消按压背景的话，直接将listSelector属性设置为@null并不合适，因为尽管设为@null，在按压列表项时仍出现橙色背景。只有把listSelector属性设置为透明色才算真正取消背景，此时listSelector的属性值如下（事先在colors.xml中定义好透明色）：

```
android:listSelector="@color/transparent"
```

（2）在代码中取消按压背景的话，调用setSelector方法不能设置null值，因为null值会在运行时报空指针异常。正确的做法是先从资源文件中获得透明色的图形对象，再调用setSelector方法设置列表项的按压状态图形，设置按压背景的代码如下：

```
// 从资源文件获得图形对象
Drawable drawable = getResources().getDrawable(R.color.transparent);
lv_planet.setSelector(drawable);  // 设置列表项的按压状态图形
```

列表视图除了以上两处属性修改，在实际开发中还有两种用法要特别小心：一种是列表视图的高度问题，另一种是列表项的点击问题，分别叙述如下。

1. 列表视图的高度问题

在XML文件中，如果ListView后面还有其他平级的控件，就要将ListView的高度设置为0dp，同时权重设置为1，确保列表视图扩展到剩余的页面区域；如果ListView的高度设置为wrap_content，系统就只给列表视图预留一行高度，如此一来只有列表的第一项会显示，其他项不显示，这显然不是我们所期望的。因此建议列表视图的尺寸参数按照如下方式设置：

```
<ListView
    android:id="@+id/lv_planet"
    android:layout_width="match_parent"
```

```
    android:layout_height="0dp"
    android:layout_weight="1" />
```

2. 列表项的点击问题

通常只要调用setOnItemClickListener方法设置点击监听器，点击列表项即可触发列表项的点击事件，但是如果列表项中存在编辑框或按钮（含Button、ImageButton、Checkbox等），点击列表项就无法触发点击事件了。缘由在于编辑框和按钮这类控件会抢占焦点，因为它们要么等待用户输入、要么等待用户点击，按道理用户点击按钮确实应该触发按钮的点击事件，而非触发列表项的点击事件，可问题是用户点击列表项的其余区域，也由于焦点被强占的缘故导致触发不了列表项的点击事件。

为了规避焦点抢占的问题，列表视图允许开发者自行设置内部视图的焦点抢占方式，该方式在XML文件中由descendantFocusability属性指定，在代码中由setDescendantFocusability方法设置，详细的焦点抢占方式说明见表7-2。

表 7-2　列表视图的焦点抢占方式

焦点抢占方式说明	代码中的焦点抢占类型	XML文件中的焦点抢占属性
在子控件之前处理	ViewGroup.FOCUS_BEFORE_DESCENDANTS	beforeDescendants
在子控件之后处理	ViewGroup.FOCUS_AFTER_DESCENDANTS	afterDescendants
不让子控件处理	ViewGroup.FOCUS_BLOCK_DESCENDANTS	blocksDescendants

注意焦点抢占方式不是由ListView设置，而是由列表项的根布局设置，也就是item_***.xml的根节点。完整的演示代码见本章源码中的ListFocusActivity.java、PlanetListWithButtonAdapter. java，以及列表项的布局文件item_list_with_button.xml。自行指定焦点抢占方式的界面效果如图7-9所示。

在图7-9所示的界面上选择焦点抢占方式“不让子控件处理（FOCUS_BLOCK_DESCENDANTS）”，之后点击列表项除按钮之外的区域，才会弹出列表项点击事件的提示。

接下来读者不妨改写第6章实战项目的购物车页面，将商品列表改为列表视图实现，从而把列表项的相关操作剥离到单独的适配器代码中，有利于界面代码的合理解耦。改造完毕的购物车效果如图7-10所示（完整代码见chapter07\src\main\java\com\example\chapter07\ShoppingCartActivity.java）。

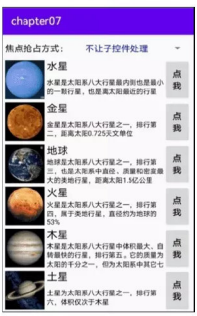


图 7-9　自行指定焦点抢占方式的界面效果



图 7-10　利用列表视图改造购物车界面

7.2.3　网格视图 GridView

除了列表视图之外，网格视图GridView也是常见的列表类视图，它用于分行分列显示表格信息，比列表视图更适合展示物品清单。除了沿用列表视图的3个方法setAdapter、setOnItemClickListener、setOnItemLongClickListener，网格视图还新增了部分属性与方法，新属性与新方法的说明见表7-3。

表 7-3　网格视图新增的属性与方法说明

XML 中的属性	代码中的设置方法	说　明
horizontalSpacing	setHorizontalSpacing	指定网格项在水平方向的间距
verticalSpacing	setVerticalSpacing	指定网格项在垂直方向的间距
numColumns	setNumColumns	指定列的数目
stretchMode	setStretchMode	指定剩余空间的拉伸模式。拉伸模式的取值说明见表 7-4
columnWidth	setColumnWidth	指定每列的宽度。拉伸模式为 spacingWidth、spacingWidthUniform 时，必须指定列宽

表 7-4　网格视图拉伸模式的取值说明

XML 中的拉伸模式	GridView 类的拉伸模式	说　明
none	NO_STRETCH	不拉伸
columnWidth	STRETCH_COLUMN_WIDTH	若有剩余空间，则拉伸列宽挤掉空隙
spacingWidth	STRETCH_SPACING	若有剩余空间，则列宽不变，把空间分配到每列间的空隙
spacingWidthUniform	STRETCH_SPACING_UNIFORM	若有剩余空间，则列宽不变，把空间分配到每列左右的空隙

在XML文件中添加GridView需要指定列的数目，以及空隙的拉伸模式，示例如下：

```
<GridView
    android:id="@+id/gv_planet"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:numColumns="2"
    android:stretchMode="columnWidth" />
```

网格视图的按压背景与焦点抢占问题类似于列表视图，此外还需注意网格项的拉伸模式，因为同一行的网格项可能占不满该行空间，多出来的空间就由拉伸模式决定怎么分配。接下来做个实验，看看各种拉伸模式分别呈现什么样的界面效果。实验之前先给网格视图设置青色背景，通过观察背景的覆盖区域，即可知晓网格项之间的空隙分布。

下面是演示网格视图拉伸模式的代码片段：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\GridViewActivity.java)
int dividerPad = Utils.dip2px(GridViewActivity.this, 2);   // 定义间隔宽度为2dp
gv_planet.setBackgroundColor(Color.CYAN);                  // 设置背景颜色
gv_planet.setHorizontalSpacing(dividerPad);                // 设置列表项在水平方向的间距
gv_planet.setVerticalSpacing(dividerPad);                  // 设置列表项在垂直方向的间距
gv_planet.setStretchMode(GridView.STRETCH_COLUMN_WIDTH);   // 设置拉伸模式
// 设置每列宽度为120dp
gv_planet.setColumnWidth(Utils.dip2px(GridViewActivity.this, 120));
gv_planet.setPadding(0, 0, 0, 0);                          // 设置网格视图的四周间距
```

```
if (arg2 == 0) {                                        // 不显示分隔线
    gv_planet.setBackgroundColor(Color.WHITE);
    gv_planet.setHorizontalSpacing(0);
    gv_planet.setVerticalSpacing(0);
} else if (arg2 == 1) {                                 // 不拉伸(NO_STRETCH)
    gv_planet.setStretchMode(GridView.NO_STRETCH);
} else if (arg2 == 2) {                                 // 拉伸列宽(COLUMN WIDTH)
    gv_planet.setStretchMode(GridView.STRETCH COLUMN WIDTH);
} else if (arg2 == 3) {                                 // 列间空隙(STRETCH SPACING)
    gv_planet.setStretchMode(GridView.STRETCH_SPACING);
} else if (arg2 == 4) {                                 // 左右空隙(SPACING_UNIFORM)
    gv_planet.setStretchMode(GridView.STRETCH SPACING UNIFORM);
} else if (arg2 == 5) {                                 // 使用padding显示全部分隔线
    gv_planet.setPadding(dividerPad, dividerPad, dividerPad, dividerPad);
}
```

运行测试App，一开始的行星网格界面如图7-11所示，此时网格视图没有分隔线。点击界面顶部的“拉伸模式”下拉框，并选择“不拉伸NO_STRETCH”，此时每行的网格项紧挨着，多出来的空隙排在当前行的右边，如图7-12所示。

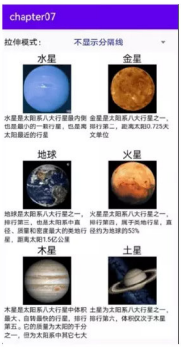


图 7-11　没有分隔线的效果

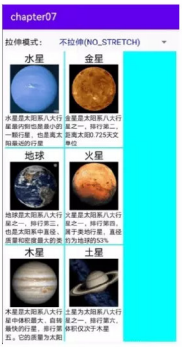


图 7-12　拉伸模式为 NO_STRETCH

拉伸模式选择“拉伸列宽（COLUMN_WIDTH）”，此时行星网格界面如图7-13所示，可见每个网格的宽度都变宽了。拉伸模式选择“列间空隙（STRETCH_SPACING）”，此时行星网格界面如图7-14所示，可见多出来的空隙位于网格项中间。

拉伸模式选择“左右空隙（SPACING_UNIFORM）”，此时行星网格界面如图7-15所示，可见空隙同时出现在网格项的左右两边。拉伸模式选择“使用padding显示全部分隔线”，此时行星网格界面如图7-16所示，可见网格视图的内外边界都显示了分隔线。

接下来继续在实战中运用网格视图，上一节的列表视图已经成功改造了购物车的商品列表，现在使用网格视图改造商品频道页面，六部手机正好做成三行两列的GridView。采用网格视图改造的商品频道页面效果如图7-17所示（完整代码见chapter07\src\main\ java\com\example\chapter07\ShoppingChannelActivity. java）。

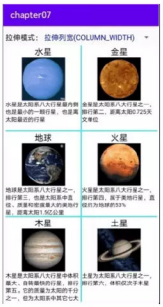


图 7-13　拉伸模式为 COLUMN_WIDTH

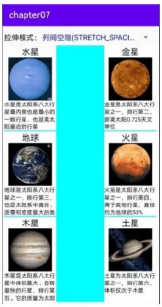


图 7-14　拉伸模式为 STRETCH_SPACING

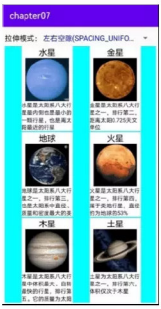


图 7-15　拉伸模式为 SPACING_UNIFORM

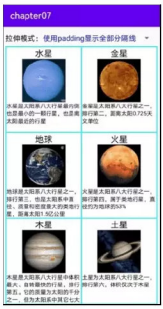


图 7-16　使用 padding 显示全部分隔线



图 7-17　使用网格视图改造后的商品频道页面

7.3　翻页类视图

本节介绍翻页类视图的相关用法，包括翻页视图ViewPager如何搭配翻页适配器PagerAdapter、如何搭配翻页标签栏PagerTabStrip，最后结合实战演示如何使用翻页视图实现简单的启动引导页。

7.3.1　翻页视图 ViewPager

上一节介绍的列表视图与网格视图，一个分行展示，另一个分行又分列，其实都是在垂直方向上下滑动。有没有一种控件允许页面在水平方向左右滑动，就像翻书、翻报纸一样呢？为了实现

左右滑动的翻页功能，Android提供了相应的控件——翻页视图ViewPager。对于ViewPager来说，一个页面就是一个项（相当于ListView的一个列表项），许多个页面组成了ViewPager的页面项。

既然明确了翻页视图的原理类似列表视图和网格视图，那它们的用法自然也很类似。例如，列表视图和网格视图使用基本适配器BaseAdapter，翻页视图则使用翻页适配器PagerAdapter；列表视图和网格视图使用列表项的点击监听器OnItemClickListener，翻页视图则使用页面变更监听器OnPageChangeListener监听页面切换事件。

下面是翻页视图3个常用方法的说明。

- setAdapter：**设置页面项的适配器。适配器用的是**PagerAdapter**及其子类。**
- setCurrentItem：**设置当前页码，也就是要显示哪个页面。**
- addOnPageChangeListener：**添加翻页视图的页面变更监听器。该监听器需实现接口**OnPageChangeListener**下的3个方法，具体说明如下：**
  - onPageScrollStateChanged：*在页面滑动状态变化时触发。*
  - onPageScrolled：*在页面滑动过程中触发。*
  - onPageSelected：*在选中页面时，即滑动结束后触发。*

在XML文件中添加ViewPager时注意指定完整路径的节点名称，示例如下：

```
<!-- 注意翻页视图ViewPager的节点名称要填全路径 -->
<androidx.viewpager.widget.ViewPager
    android:id="@+id/vp_content"
    android:layout_width="match_parent"
    android:layout_height="370dp" />
```

由于翻页视图包含了多个页面项，因此要借助翻页适配器展示每个页面。翻页适配器的实现原理与基本适配器类似，从PagerAdapter派生的翻页适配器主要实现下面6个方法。

- **构造方法：指定适配器需要处理的数据集合。**
- getCount：**获取页面项的个数。**
- isViewFromObject：**判断当前视图是否来自指定对象，返回**view == object**即可。**
- instantiateItem：**实例化指定位置的页面，并将其添加到容器中。**
- destroyItem：**从容器中销毁指定位置的页面。**
- getPageTitle：**获得指定页面的标题文本，有搭配翻页标签栏时才要实现该方法。**

以商品信息为例，翻页适配器需要通过构造方法传入商品列表，再由instantiateItem方法实例化视图对象并添加至容器，详细的翻页适配器代码示例如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\adapter\ImagePagerAdapater.java)
public class ImagePagerAdapater extends PagerAdapter {
    // 声明一个图像视图列表
    private List<ImageView> mViewList = new ArrayList<ImageView>();
    // 声明一个商品信息列表
    private List<GoodsInfo> mGoodsList = new ArrayList<GoodsInfo>();

    // 图像翻页适配器的构造方法，传入上下文与商品信息列表
    public ImagePagerAdapater(Context context, List<GoodsInfo> goodsList) {
        mGoodsList = goodsList;
        // 给每个商品分配一个专用的图像视图
        for (int i = 0; i < mGoodsList.size(); i++) {
            ImageView view = new ImageView(context);   // 创建一个图像视图对象
            view.setLayoutParams(new LayoutParams(
```

```
                    LayoutParams.MATCH_PARENT, LayoutParams.WRAP_CONTENT));
            view.setImageResource(mGoodsList.get(i).pic);
            mViewList.add(view);                    // 把该商品的图像视图添加到图像视图列表
        }
    }

    // 获取页面项的个数
    public int getCount() {
        return mViewList.size();
    }

    // 判断当前视图是否来自指定对象
    public boolean isViewFromObject(View view, Object object) {
        return view == object;
    }

    // 从容器中销毁指定位置的页面
    public void destroyItem(ViewGroup container, int position, Object object) {
        container.removeView(mViewList.get(position));
    }

    // 实例化指定位置的页面，并将其添加到容器中
    public Object instantiateItem(ViewGroup container, int position) {
        container.addView(mViewList.get(position));
        return mViewList.get(position);
    }
}
```

接着回到活动页面代码，给翻页视图设置上述的翻页适配器，代码如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\ViewPagerActivity.java)
public class ViewPagerActivity extends AppCompatActivity implements
OnPageChangeListener {
    private List<GoodsInfo> mGoodsList;  // 手机商品列表

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_view_pager);
        mGoodsList = GoodsInfo.getDefaultList();
        // 构建一个商品图片的翻页适配器
        ImagePagerAdapater adapter = new ImagePagerAdapater(this, mGoodsList);
        // 从布局视图中获取名为vp_content的翻页视图
        ViewPager vp_content = findViewById(R.id.vp_content);
        vp_content.setAdapter(adapter);          // 设置翻页视图的适配器
        vp_content.setCurrentItem(0);            // 设置翻页视图显示第一页
        vp_content.addOnPageChangeListener(this);// 给翻页视图添加页面变更监听器
    }

    // 翻页状态改变时触发。state取值说明：0表示静止，1表示正在滑动，2表示滑动完毕
    // 在翻页过程中，状态值变化依次为：正在滑动→滑动完毕→静止
    public void onPageScrollStateChanged(int state) {}

    // 在翻页过程中触发。该方法的3个参数取值说明：第一个参数表示当前页面的序号
    // 第二个参数表示页面偏移的百分比，取值为0～1；第三个参数表示页面的偏移距离
    public void onPageScrolled(int position, float ratio, int offset) {}

    // 在翻页结束后触发。position表示当前滑到了哪一个页面
    public void onPageSelected(int position) {
        Toast.makeText(this, "您翻到的手机品牌是：" +
                mGoodsList.get(position).name, Toast.LENGTH_SHORT).show();
    }
}
```

由于监听器 OnPageChangeListener 多数情况只用到 onPageSelected 方法，很少用到

onPageScrollStateChanged和onPageScrolled两个方法，因此Android又提供了简化版的页面变更监听器，名为SimpleOnPageChangeListener，新的监听器仅需实现onPageSelected方法。给翻页视图添加简化版监听器的代码示例如下：

```
// 给翻页视图添加简化版的页面变更监听器
vp_content.addOnPageChangeListener(
        new ViewPager.SimpleOnPageChangeListener() {
    @Override
    public void onPageSelected(int position) {
        Toast.makeText(ViewPagerActivity.this, "您翻到的手机品牌是："
                + mGoodsList.get(position).name, Toast.LENGTH_SHORT).show();
    }
});
```

然后运行测试App，初始的翻页视图如图7-18所示，此时整个页面只显示第一部手机。用手指从右向左活动页面，滑到一半的翻页视图如图7-19所示，可见第一部手机逐渐向左隐去，而第二部手机逐渐从右边拉出。继续向左活动一段距离再松开手指，此时滑动结束的翻页视图如图7-20所示，可见整个页面完全显示第二部手机了。



图 7-18　初始的翻页视图



图 7-19　滑到一半的翻页视图



图 7-20　滑动结束的翻页视图

7.3.2　翻页标签栏 PagerTabStrip

尽管翻页视图实现了左右滑动，可是没滑动的时候看不出这是个翻页视图，而且也不知道当前滑到了哪个页面。为此Android提供了翻页标签栏PagerTabStrip，它能够在翻页视图上方显示页面标题，从而方便用户的浏览操作。PagerTabStrip类似选项卡效果，文本下面有横线，点击左右选项卡即可切换到对应页面。给翻页视图引入翻页标签栏只需下列两个步骤：

01 在XML文件的ViewPager节点内部添加PagerTabStrip节点，示例如下：

```
(完整代码见chapter07\src\main\res\layout\activity_pager_tab.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <!-- 注意翻页视图ViewPager的节点名称要填全路径 -->
    <androidx.viewpager.widget.ViewPager
        android:id="@+id/vp_content"
        android:layout_width="match_parent"
        android:layout_height="400dp">
        <!-- 注意翻页标签栏PagerTabStrip的节点名称要填全路径 -->
        <androidx.viewpager.widget.PagerTabStrip
            android:id="@+id/pts_tab"
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </androidx.viewpager.widget.ViewPager>
</LinearLayout>
```

02 在翻页适配器的代码中重写getPageTitle方法，在不同位置返回对应的标题文本，示例代码如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\adapter\ImagePagerAdapater.java)
    // 获得指定页面的标题文本
    public CharSequence getPageTitle(int position) {
        return mGoodsList.get(position).name;
    }
```

完成上述两步骤之后，重新运行测试App，即可观察翻页标签栏的界面效果。如图7-21和图7-22所示，这是翻到不同页面的翻页视图，可见界面正上方是当前页面的标题，左上方文字是左边页面的标题，右上方文字是右边页面的标题。



图 7-21 翻页标签栏的界面效果 1



图 7-22 翻页标签栏的界面效果 2

另外，若想修改翻页标签栏的文本样式，必须在Java代码中调用setTextSize和setTextColor方法才行，因为PagerTabStrip不支持在XML文件中设置文本大小和文本颜色，只能在代码中设置文本样式，具体的设置代码如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\PagerTabActivity.java)
// 初始化翻页标签栏
private void initPagerStrip() {
    // 从布局视图中获取名为pts tab的翻页标签栏
    PagerTabStrip pts tab = findViewById(R.id.pts tab);
    // 设置翻页标签栏的文本大小
    pts_tab.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20);
    pts tab.setTextColor(Color.BLACK);  // 设置翻页标签栏的文本颜色
}
```

7.3.3 简单的启动引导页

翻页视图的使用范围很广，当用户安装一个新应用时，首次启动大多出现欢迎页面，这个引导页要往右翻好几页，才会进入应用主页。这种启动引导页就是通过翻页视图实现的。

下面就来动手打造你的第一个App启动欢迎页吧！翻页技术的核心在于页面项的XML布局及其适配器，因此首先要设计页面项的布局。一般来说，引导页由两部分组成，一部分是背景图；另一部分是页面下方的一排圆点，其中高亮的圆点表示当前位于第几页。启动引导页的界面效果如

图7-23与图7-24所示。其中，图7-23为欢迎页面的第一页，此时第一个圆点高亮显示；图7-24为右翻到了第二页，此时第二个圆点高亮显示。



图 7-23　欢迎页的第一页



图 7-24　欢迎页的第二页

除了背景图与一排圆点之外，最后一页往往有个按钮，它便是进入应用主页的入口。于是页面项的XML文件至少包含3个控件：引导页的背景图（采用ImageView）、底部的一排圆点（采用RadioGroup）、最后一页的入口按钮（采用Button），XML内容示例如下：

```
（完整代码见chapter07\src\main\res\layout\item_launch.xml）
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout height="match parent">
    <!-- 这是引导图片的图像视图 -->
    <ImageView
        android:id="@+id/iv launch"
        android:layout width="match parent"
        android:layout height="match parent"
        android:scaleType="fitXY" />
    <!-- 这里容纳引导页底部的一排圆点 -->
    <RadioGroup
        android:id="@+id/rg indicate"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:orientation="horizontal"
        android:paddingBottom="20dp" />
    <!-- 这是最后一页的入口按钮 -->
    <Button
        android:id="@+id/btn_start"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout centerInParent="true"
        android:text="立即开始美好生活"
        android:visibility="gone" />"
</RelativeLayout>
```

根据上面的XML文件，引导页的最后两页如图7-25与图7-26所示。其中，图7-25是第三页，此时第三个圆点高亮显示；图7-26是最后一页，只有该页才会显示入口按钮。



图 7-25 欢迎页的第三页



图 7-26 欢迎页的最后一页

写好了页面项的XML布局，还得编写启动引导页的适配器代码，主要完成3项工作：

（1）根据页面项的XML文件构造每页的视图。

（2）让当前页码的圆点高亮显示。

（3）如果翻到了最后一页，就显示中间的入口按钮。

启动引导页对应的翻页适配器代码示例如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\adapter\LaunchSimpleAdapter.java)
public class LaunchSimpleAdapter extends PagerAdapter {
    // 声明一个引导页的视图列表
    private List<View> mViewList = new ArrayList<View>();

    // 引导页适配器的构造方法，传入上下文与图片数组
    public LaunchSimpleAdapter(final Context context, int[] imageArray) {
        for (int i = 0; i < imageArray.length; i++) {
            // 根据布局文件item launch.xml生成视图对象
            View view = LayoutInflater.from(context).inflate(
                            R.layout.item launch, null);
            ImageView iv launch = view.findViewById(R.id.iv launch);
            RadioGroup rg indicate = view.findViewById(R.id.rg indicate);
            Button btn_start = view.findViewById(R.id.btn_start);
            iv_launch.setImageResource(imageArray[i]);          // 设置引导页的全屏图片
            // 每个页面都分配一个对应的单选按钮
            for (int j = 0; j < imageArray.length; j++) {
                RadioButton radio = new RadioButton(context);   //创建一个单选按钮
                radio.setLayoutParams(new LayoutParams(
                        LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT));
                // 设置单选按钮的图标
                radio.setButtonDrawable(R.drawable.launch guide);
                radio.setPadding(10, 10, 10, 10);       // 设置单选按钮的四周间距
                rg indicate.addView(radio);             // 把单选按钮添加到页面底部的单选组
            }
            // 当前位置的单选按钮要高亮显示，比如是第二个引导页就高亮第二个单选按钮
            ((RadioButton) rg indicate.getChildAt(i)).setChecked(true);
            // 如果是最后一个引导页，则显示入口按钮，以便用户点击按钮进入主页
            if (i == imageArray.length - 1) {
                btn start.setVisibility(View.VISIBLE);
                btn_start.setOnClickListener(v -> {
                    // 这里要跳转到应用主页
                });
            }
```

```
            mViewList.add(view);            // 把该图片对应的页面添加到引导页的视图列表
        }
    }
    // 获取页面项的个数
    public int getCount() {
        return mViewList.size();
    }
    // 判断当前视图是否来自指定对象
    public boolean isViewFromObject(View view, Object object) {
        return view == object;
    }
    // 从容器中销毁指定位置的页面
    public void destroyItem(ViewGroup container, int position, Object object){
        container.removeView(mViewList.get(position));
    }
    // 实例化指定位置的页面，并将其添加到容器中
    public Object instantiateItem(ViewGroup container, int position) {
        container.addView(mViewList.get(position));
        return mViewList.get(position);
    }
}
```

7.4 碎片 Fragment

本节介绍碎片的概念及其用法，包括通过静态注册方式使用碎片、通过动态注册方式使用碎片（需要配合碎片适配器FragmentPagerAdapter），并分析两种注册方式的碎片生命周期，最后结合实战演示如何使用碎片改进启动引导页。

7.4.1 碎片的静态注册

碎片Fragment是个特别的存在，它有点像报纸上的专栏，看起来只占据页面的一小块区域，但是这一区域有自己的生命周期，可以自食其力，仿佛独立王国；并且该区域只占据空间不扰乱业务，添加之后不影响宿主页面的其他区域，去除之后也不影响宿主页面的其他区域。

每个碎片都有对应的XML布局文件，依据其使用方式可分为静态注册与动态注册两类。静态注册指的是在XML文件中直接放置fragment节点，类似于一个普通控件，可被多个布局文件同时引用。静态注册一般用于某个通用的页面部件（如Logo条、广告条等），每个活动页面均可直接引用该部件。

下面是碎片页对应的XML文件内容，看起来跟列表项与网格项的布局文件差不多。

```
(完整代码见chapter07\src\main\res\layout\fragment_static.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:background="#bbffbb">
    <TextView
        android:id="@+id/tv_adv"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
```

```
            android:gravity="center"
            android:text="广告图片" />
        <ImageView
            android:id="@+id/iv_adv"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="4"
            android:src="@drawable/adv" />
    </LinearLayout>
```

下面是与上述XML布局对应的碎片代码，除了继承自Fragment与入口方法onCreateView两点外，其他地方类似活动页面代码。

```
    (完整代码见chapter07\src\main\java\com\example\chapter07\fragment\StaticFragment.java)
    public class StaticFragment extends Fragment implements OnClickListener {
        protected View mView;                       // 声明一个视图对象
        protected Context mContext;                 // 声明一个上下文对象
        // 创建碎片视图
        @Override
        public View onCreateView(LayoutInflater inflater,ViewGroup container,Bundle
savedInstanceState) {
            mContext = getActivity();               // 获取活动页面的上下文
            // 根据布局文件fragment_static.xml生成视图对象
            mView = inflater.inflate(R.layout.fragment_static, container, false);
            TextView tv_adv = mView.findViewById(R.id.tv_adv);
            ImageView iv_adv = mView.findViewById(R.id.iv_adv);
            tv_adv.setOnClickListener(this);        // 设置点击监听器
            iv_adv.setOnClickListener(this);        // 设置点击监听器
            return mView;                           // 返回该碎片的视图对象
        }
        @Override
        public void onClick(View v) {
            if (v.getId() == R.id.tv_adv) {
                Toast.makeText(mContext, "您点击了广告文本", Toast.LENGTH_LONG).show();
            } else if (v.getId() == R.id.iv_adv) {
                Toast.makeText(mContext, "您点击了广告图片",  Toast.LENGTH_LONG).show();
            }
        }
    }
```

若想在活动页面的XML文件中引用上面定义的StaticFragment，可以直接添加一个fragment节点，但需注意下列两点：

（1）fragment节点必须指定id属性，否则App运行会报错。

（2）fragment节点必须通过name属性指定碎片类的完整路径。

在布局文件中引用碎片的XML例子如下：

```
    (完整代码见chapter07\src\main\res\layout\activity_fragment_static.xml)
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <!-- 把碎片当作一个控件使用，其中android:name指明了碎片来源 -->
        <fragment
            android:id="@+id/fragment_static"
            android:name="com.example.chapter07.fragment.StaticFragment"
            android:layout_width="match_parent"
            android:layout_height="60dp" />
```

```
    <TextView
        android:layout width="match parent"
        android:layout height="wrap content"
        android:gravity="center"
        android:text="这里是每个页面的具体内容" />
</LinearLayout>
```

运行测试App，可见碎片所在界面如图7-27所示。此时碎片区域仿佛一个视图，其内部控件同样可以接收点击事件。

另外，介绍一下碎片在静态注册时的生命周期，像活动的基本生命周期方法onCreate、onStart、onResume、onPause、onStop、onDestroy，碎片同样也有，而且还多出了下面5个生命周期方法。



图 7-27　静态注册的碎片效果

- onAttach：**与活动页面结合。**
- onCreateView：**创建碎片视图。**
- onActivityCreated：**在活动页面创建完毕后调用。**
- onDestroyView：**回收碎片视图。**
- onDetach：**与活动页面分离。**

至于这些周期方法的先后调用顺序，观察日志最简单明了。下面是打开活动页面时的日志信息，此时碎片的onCreate方法先于活动的onCreate方法，而碎片的onStart与onResume均在活动的同名方法之后。

```
12:26:11.506: D/StaticFragment: onAttach
12:26:11.506: D/StaticFragment: onCreate
12:26:11.530: D/StaticFragment: onCreateView
12:26:11.530: D/FragmentStaticActivity: onCreate
12:26:11.530: D/StaticFragment: onActivityCreated
12:26:11.530: D/FragmentStaticActivity: onStart
12:26:11.530: D/StaticFragment: onStart
12:26:11.530: D/FragmentStaticActivity: onResume
12:26:11.530: D/StaticFragment: onResume
```

下面是退出活动页面时的日志信息，此时碎片的onPause、onStop、onDestroy都在活动的同名方法之前。

```
12:26:36.586: D/StaticFragment: onPause
12:26:36.586: D/FragmentStaticActivity: onPause
12:26:36.990: D/StaticFragment: onStop
12:26:36.990: D/FragmentStaticActivity: onStop
12:26:36.990: D/StaticFragment: onDestroyView
12:26:36.990: D/StaticFragment: onDestroy
12:26:36.990: D/StaticFragment: onDetach
12:26:36.990: D/FragmentStaticActivity: onDestroy
```

总结一下，在静态注册时，除了碎片的创建操作在页面创建之前，其他操作没有僭越页面范围。就像老实本分的下级，上级开口后才能说话，上级要做总结性发言前赶紧闭上嘴巴。

7.4.2　碎片的动态注册

碎片拥有两种使用方式，也就是静态注册和动态注册。相比静态注册，在实际开发中动态注册用得更多。静态注册是在XML文件中直接添加fragment节点，而动态注册迟至代码执行时才动态添加碎片。动态生成的碎片基本给翻页视图使用，要知道ViewPager和Fragment可是一对好搭档。

要想在翻页视图中使用动态碎片，关键在于适配器。在“7.3.1 翻页视图ViewPager”节演示翻页功能时，用到了翻页适配器PagerAdapter。如果结合使用碎片，翻页视图的适配器就要改用碎片适配器FragmentPagerAdapter。与翻页适配器相比，碎片适配器增加了getItem方法用于获取指定位置的碎片，同时去掉了isViewFromObject、instantiateItem、destroyItem三个方法，用起来更加容易。下面是一个碎片适配器的实现代码例子。

```
(完整代码见chapter07\src\main\java\com\example\chapter07\adapter\MobilePagerAdapter.java)
public class MobilePagerAdapter extends FragmentPagerAdapter {
    // 声明一个商品列表
    private List<GoodsInfo> mGoodsList = new ArrayList<GoodsInfo>();
    // 碎片页适配器的构造方法，传入碎片管理器与商品信息列表
    public MobilePagerAdapter(FragmentManager fm, List<GoodsInfo> goodsList) {
        super(fm, BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT);
        mGoodsList = goodsList;
    }
    // 获取碎片Fragment的个数
    public int getCount() {
        return mGoodsList.size();
    }
    // 获取指定位置的碎片Fragment
    public Fragment getItem(int position) {
        return DynamicFragment.newInstance(position,
                mGoodsList.get(position).pic, mGoodsList.get(position).desc);
    }
    // 获得指定碎片页的标题文本
    public CharSequence getPageTitle(int position) {
        return mGoodsList.get(position).name;
    }
}
```

上面的适配器代码在getItem方法中不调用碎片的构造方法，却调用了newInstance方法，目的是给碎片对象传递参数信息。由newInstance方法内部先调用构造方法创建碎片对象，再调用setArguments方法塞进请求参数，然后在onCreateView中调用getArguments方法才能取出请求参数。下面是在动态注册时传递请求参数的碎片代码例子：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\fragment\DynamicFragment.java)
public class DynamicFragment extends Fragment {
    protected View mView;                                   // 声明一个视图对象
    protected Context mContext;                             // 声明一个上下文对象
    private int mPosition;                                  // 位置序号
    private int mImageId;                                   // 图片的资源编号
    private String mDesc;                                   // 商品的文字描述
    // 获取该碎片的一个实例
    public static DynamicFragment newInstance(int position, int image id, String desc) {
        DynamicFragment fragment = new DynamicFragment();       // 创建该碎片的一个实例
        Bundle bundle = new Bundle();                           // 创建一个新包裹
        bundle.putInt("position", position);                // 往包裹里存入位置序号
        bundle.putInt("image id", image id);                // 往包裹里存入图片的资源编号
        bundle.putString("desc", desc);                     // 往包裹里存入商品的文字描述
        fragment.setArguments(bundle);                      // 把包裹塞给碎片
        return fragment;  // 返回碎片实例
    }
    // 创建碎片视图
    public View onCreateView(LayoutInflater inflater,ViewGroup container,Bundle
savedInstanceState) {
```

```
        mContext = getActivity();                    // 获取活动页面的上下文
        if (getArguments() != null) {                // 如果碎片携带有包裹，就打开包裹获取参数信息
            // 从包裹里取出位置序号
            mPosition = getArguments().getInt("position", 0);
            // 从包裹里取出图片的资源编号
            mImageId = getArguments().getInt("image_id", 0);
            mDesc = getArguments().getString("desc");       // 从包裹里取出商品的文字描述
        }
        // 根据布局文件fragment_dynamic.xml生成视图对象
        mView = inflater.inflate(R.layout.fragment_dynamic, container, false);
        ImageView iv_pic = mView.findViewById(R.id.iv_pic);
        TextView tv_desc = mView.findViewById(R.id.tv_desc);
        iv_pic.setImageResource(mImageId);
        tv_desc.setText(mDesc);
        return mView;                                        // 返回该碎片的视图对象
    }
}
```

现在有了适用于动态注册的适配器与碎片对象，还需要一个活动页面展示翻页视图及其搭配的碎片适配器。下面便是动态注册用到的活动页面代码。

(完整代码见chapter07\src\main\java\com\example\chapter07\FragmentDynamicActivity.java)

```
public class FragmentDynamicActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment_dynamic);
        List<GoodsInfo> goodsList = GoodsInfo.getDefaultList();
        // 构建一个手机商品的碎片翻页适配器
        MobilePagerAdapter adapter = new MobilePagerAdapter(
                getSupportFragmentManager(), goodsList);
        // 从布局视图中获取名为vp_content的翻页视图
        ViewPager vp_content = findViewById(R.id.vp_content);
        vp_content.setAdapter(adapter);              // 设置翻页视图的适配器
        vp_content.setCurrentItem(0);                // 设置翻页视图显示第一页
    }
}
```

运行测试App，初始的碎片界面如图7-28所示，此时默认展示第一个碎片，包含商品图片和商品描述。接着一路滑到最后一页如图7-29所示，此时展示了最后一个碎片，可见总体界面效果类似于“7.3.2　翻页标签栏PagerTabStrip”那样。



图 7-28　翻到第一个碎片界面



图 7-29　翻到最后一个碎片界面

接下来观察动态注册时的碎片生命周期。按惯例分别在活动代码与碎片代码内部补充生命周期的日志，然后观察App运行日志。下面是打开活动页面时的日志信息：

```
12:28:28.074: D/FragmentDynamicActivity: onCreate
12:28:28.074: D/FragmentDynamicActivity: onStart
12:28:28.074: D/FragmentDynamicActivity: onResume
12:28:28.086: D/DynamicFragment: onAttach position=0
12:28:28.086: D/DynamicFragment: onCreate position=0
12:28:28.114: D/DynamicFragment: onCreateView position=0
12:28:28.114: D/DynamicFragment: onActivityCreated position=0
12:28:28.114: D/DynamicFragment: onStart position=0
12:28:28.114: D/DynamicFragment: onResume position=0
12:28:28.114: D/DynamicFragment: onAttach position=0
12:28:28.114: D/DynamicFragment: onCreate position=0
12:28:28.146: D/DynamicFragment: onCreateView position=1
12:28:28.146: D/DynamicFragment: onStart position=1
12:28:28.146: D/DynamicFragment: onResume position=1
```

下面是退出活动页面时的日志信息：

```
12:28:57.994: D/DynamicFragment: onPause position=0
12:28:57.994: D/DynamicFragment: onPause position=1
12:28:57.994: D/FragmentDynamicActivity: onPause
12:28:58.402: D/DynamicFragment: onStop position=0
12:28:58.402: D/DynamicFragment: onStop position=1
12:28:58.402: D/FragmentDynamicActivity: onStop
12:28:58.402: D/DynamicFragment: onDestroyView position=0
12:28:58.402: D/DynamicFragment: onDestroy position=0
12:28:58.402: D/DynamicFragment: onDetach position=0
12:28:58.402: D/DynamicFragment: onDestroyView position=1
12:28:58.402: D/DynamicFragment: onDestroy position=1
12:28:58.402: D/DynamicFragment: onDetach position=1
12:28:58.402: D/FragmentDynamicActivity: onDestroy
```

日志搜集完毕，分析其中的奥妙，总结一下主要有以下3点：

（1）动态注册时，碎片的onCreate方法在活动的onCreate方法之后，其余方法的先后顺序与静态注册时保持一致。

（2）注意onActivityCreated方法，无论是静态注册还是动态注册，该方法都在活动的onCreate方法之后，可见该方法的确在页面创建之后才调用。

（3）最重要的一点，进入第一个碎片之际，实际只加载了第一页和第二页，并没有加载所有碎片页，这正是碎片动态注册的优点。无论当前位于哪一页，系统都只会加载当前页及相邻的左右两页，总共加载不超过3页。一旦发生页面切换，相邻页面就被加载，非相邻页面就被回收。这么做的好处是节省了宝贵的系统资源，只有用户正在浏览与将要浏览的碎片页才会加载，避免所有碎片页一起加载造成资源浪费，而这正是普通翻页视图的缺点。

7.4.3 改进的启动引导页

接下来将碎片用于实战，对“7.3.3 简单的启动引导页”加以改进。与之前相比，XML文件不变，改动的都是Java代码。用于启动引导页的碎片适配器代码如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\adapter\LaunchImproveAdapter.java)
public class LaunchImproveAdapter extends FragmentPagerAdapter {
    private int[] mImageArray;  // 声明一个图片数组
    // 碎片页适配器的构造方法，传入碎片管理器与图片数组
    public LaunchImproveAdapter(FragmentManager fm, int[] imageArray) {
        super(fm, BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT);
        mImageArray = imageArray;
```

```
        }
        // 获取碎片Fragment的个数
        public int getCount() {
            return mImageArray.length;
        }
        // 获取指定位置的碎片Fragment
        public Fragment getItem(int position) {
            return LaunchFragment.newInstance(position, mImageArray[position]);
        }
    }
```

以上的碎片适配器代码倒是简单，原来与视图控件有关的操作都挪到碎片代码当中了，下面是每个启动页的碎片代码例子：

```
    (完整代码见chapter07\src\main\java\com\example\chapter07\fragment\LaunchFragment.java)
    public class LaunchFragment extends Fragment {
        protected View mView;                                   // 声明一个视图对象
        protected Context mContext;                             // 声明一个上下文对象
        private int mPosition;                                  // 位置序号
        private int mImageId;                                   // 图片的资源编号
        private int mCount = 4;                                 // 引导页的数量
        // 获取该碎片的一个实例
        public static LaunchFragment newInstance(int position, int image_id) {
            LaunchFragment fragment = new LaunchFragment();     // 创建该碎片的一个实例
            Bundle bundle = new Bundle();                       // 创建一个新包裹
            bundle.putInt("position", position);                // 往包裹里存入位置序号
            bundle.putInt("image_id", image_id);                // 往包裹里存入图片的资源编号
            fragment.setArguments(bundle);                      // 把包裹塞给碎片
            return fragment;                                    // 返回碎片实例
        }
        // 创建碎片视图
        public View onCreateView(LayoutInflater inflater,ViewGroup container,Bundle
savedInstanceState) {
            mContext = getActivity();               // 获取活动页面的上下文
            if (getArguments() != null) {           // 如果碎片携带有包裹，就打开包裹获取参数信息
                // 从包裹里获取位置序号
                mPosition = getArguments().getInt("position", 0);
                // 从包裹里获取图片的资源编号
                mImageId = getArguments().getInt("image id", 0);
            }
            // 根据布局文件item_launch.xml生成视图对象
            mView = inflater.inflate(R.layout.item_launch, container, false);
            ImageView iv_launch = mView.findViewById(R.id.iv_launch);
            RadioGroup rg_indicate = mView.findViewById(R.id.rg_indicate);
            Button btn_start = mView.findViewById(R.id.btn_start);
            iv_launch.setImageResource(mImageId);           // 设置引导页的全屏图片
            // 每个页面都分配一个对应的单选按钮
            for (int j = 0; j < mCount; j++) {
                RadioButton radio = new RadioButton(mContext);
                radio.setLayoutParams(new LayoutParams(
                        LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT));
                // 设置单选按钮的图标
                radio.setButtonDrawable(R.drawable.launch_guide);
                radio.setPadding(10, 10, 10, 10);           // 设置单选按钮的四周间距
                rg_indicate.addView(radio);                 // 把单选按钮添加到页面底部的单选组
            }
            // 当前位置的单选按钮要高亮显示，比如是第二个引导页就高亮第二个单选按钮
            ((RadioButton) rg_indicate.getChildAt(mPosition)).setChecked(true);
            // 如果是最后一个引导页，则显示入口按钮，以便用户点击按钮进入首页
```

```
            if (mPosition == mCount - 1) {
                btn_start.setVisibility(View.VISIBLE);
                btn_start.setOnClickListener(v -> {
                    // 这里要跳转到应用主页
                });
            }
            return mView;  // 返回该碎片的视图对象
        }
    }
```

经过碎片改造后的启动引导页，其界面效果跟“7.3.3　简单的启动引导页”是一样的。尽管看不出界面上的差异，但引入碎片之后至少有以下两个好处。

（1）加快启动速度。因为动态注册的碎片一开始只会加载前两个启动页，对比原来加载所有启动页（至少4页），无疑大幅减少了加载页的数量，从而提升了启动速度。

（2）降低代码耦合。把视图操作剥离到单独的碎片代码，不与适配器代码混合在一起，方便后继的代码维护工作。

7.5　实战项目：记账本

人云：你不理财，财不理你。从工作开始，年轻人就要好好管理自己的个人收支。每年的收入减去支出，剩下的结余才是进一步发展的积累资金。记账本便是管理日常收支的好帮手，一个易用的记账本App有助于合理安排个人资金。

7.5.1　需求描述

好用的记账本必须具备两项基本功能：一项是记录新账单，另一项是查看账单列表。其中账单的记录操作要求用户输入账单的明细要素，包括账单的发生时间、账单的收支类型（收入还是支出）、账单的交易金额、账单的事由描述等，据此勾勒出的简易的账单添加界面如图7-30所示。账单列表页通常分月展示，每页显示单个月份的账单数据，还要支持在不同月份之间切换。每月的账单数据按照时间从上往下排列，每行的账单明细则需依次展示账单日期、事由描述、交易金额等信息，然后列表末尾展示当月的账单合计情况（总共收入多少、总共支出多少）。根据这些要求描绘的账单列表界面原型如图7-31所示。



图 7-30　账单填写页面



图 7-31　账单列表页面

账单的填写功能对应数据库记录的添加操作，账单的展示功能对应数据库记录的查询操作，

数据库记录还有修改和删除操作，分别对应账单的编辑功能和删除功能。账单的编辑页面原型如图7-32所示，至于删除操作则由如图7-33所示的提示窗控制，点击“是”按钮表示确定删除，点击“否”按钮表示取消删除。



图 7-32　账单编辑页面

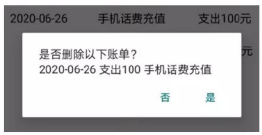


图 7-33　删除账单的提示窗

7.5.2　界面设计

除了文本视图、按钮、编辑框、单选按钮等简单控件之外，记账本还用到了下列控件以及相关的适配器：

- **翻页视图ViewPager：每页一个月份，一年12个月，支持左右滑动，用到了ViewPager。**
- **翻页标签栏PagerTabStrip：每个账单页上方的月份标题来自PagerTabStrip。**
- **碎片适配器FragmentPagerAdapter：把12个月份的Fragment组装到ViewPager中，用到了碎片适配器。**
- **碎片Fragment：12个月份对应12个账单页，每页都是一个Fragment。**
- **列表视图ListView：每月的账单明细从上往下排列，采用了ListView。**
- **基本适配器BaseAdapter：每行的账单项依次展示账单日期、事由描述、交易金额等信息，需要列表视图搭档基本适配器。**
- **提醒对话框AlertDialog：删除账单项的提示窗用到了AlertDialog。**
- **日期选择对话框DatePickerDialog：在填写账单信息时，要通过DatePickerDialog选择账单日期。**

记账本的几个页面当中，账单列表页面使用了好几种高级控件，又有翻页视图又有列表视图，以及它们各自的数据适配器，看起来颇为复杂。为方便读者理清该页面的控件联系，图7-34列出了从活动页面开始直到账单行为止的依赖嵌套关系（账单总体页面→每个月份的账单页→每月账单的明细列表→每行的账单信息）。

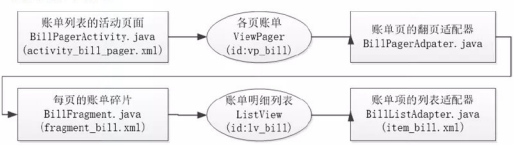


图 7-34　账单列表页面的控件嵌套关系

7.5.3 关键代码

为了方便读者顺利完成记账本的编码开发，下面罗列几处关键的代码实现逻辑。

1. 如何实现日期下拉框

在填写账单时间的时候，输入界面默认展示当天日期，用户若想修改账单时间，就要点击日期文本，此时界面弹出日期选择对话框，待用户选定具体日期，再回到主界面展示选定日期的文本。这种实现方式类似于下拉框控件Spinner，可是点击Spinner会弹出文本列表对话框，而非日期选择对话框。尽管Android未提供现成的日期下拉框，但是结合文本视图与日期选择对话框，也能实现类似Spinner的日期下拉框效果。具体步骤说明如下：

01 在账单填写页面的XML文件中添加名为tv_date的TextView，并给它指定drawableRight属性，属性值为一个向下三角形的资源图片，也就是让该控件看起来像个下拉框。包含tv_date在内的账单时间布局片段示例如下：

```
(完整代码见chapter07\src\main\res\layout\activity_bill_add.xml)
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="40dp"
    android:orientation="horizontal">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:gravity="center|right"
        android:text="账单日期：" />
    <TextView
        android:id="@+id/tv_date"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="2"
        android:drawableRight="@drawable/arrow_down"
        android:gravity="center" />
</LinearLayout>
```

02 回到该页面对应的Java代码，给文本视图tv_date注册点击监听器，一旦发现用户点击了该视图，就弹出日期选择对话框DatePickerDialog。下面是控件tv_date的点击响应代码例子：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\BillAddActivity.java)
@Override
public void onClick(View v) {
    if (v.getId() == R.id.tv_date) {
        // 构建一个日期对话框，构造方法的第二个构造参数指定了日期监听器
        DatePickerDialog dialog = new DatePickerDialog(this, this,
                calendar.get(Calendar.YEAR),                // 年份
                calendar.get(Calendar.MONTH),               // 月份
                calendar.get(Calendar.DAY_OF_MONTH));       // 日期
        dialog.show();  // 显示日期选择对话框
    }
}
```

03 注意到在步骤2构建日期对话框时，将日期监听器设在了当前页面，于是令活动代码实现日期变更监听接口DatePickerDialog.OnDateSetListener，同时还要重写该接口的onDateSet方法，一旦发现用户选择了某个日期，就将文本视图tv_date设为该日期文本。重写后的onDateSet方法代码示例如下：

```
    @Override
    public void onDateSet(DatePicker view, int year, int month, int dayOfMonth) {
        calendar.set(Calendar.YEAR, year);
        calendar.set(Calendar.MONTH, month);
        calendar.set(Calendar.DAY_OF_MONTH, dayOfMonth);
        tv_date.setText(DateUtil.getDate(calendar));
    }
```

2. 如何编辑与删除账单项

需求描述提到既要支持账单的编辑功能，又要支持账单的删除功能，因为账单明细位于列表视图当中，且列表视图允许同时设置列表项的点击监听器和长按监听器，所以可以考虑将列表项的点击监听器映射到账单的编辑功能，将列表项的长按监听器映射到账单的删除功能，也就是点击账单项时跳转到账单的编辑页面，长按账单项时弹出删除账单的提醒对话框。为此需要在账单的列表页实现下列两个步骤：

01 给每月账单的列表视图分别注册列表项的点击监听器和长按监听器，注册代码如下：

```
    (完整代码见chapter07\src\main\java\com\example\chapter07\fragment\BillFragment.java)
    // 构建一个当月账单的列表适配器
    BillListAdapter listAdapter = new BillListAdapter(mContext, mBillList);
    lv_bill.setAdapter(listAdapter);                        // 设置列表视图的适配器
    lv_bill.setOnItemClickListener(listAdapter);            // 设置列表视图的点击监听器
    lv_bill.setOnItemLongClickListener(listAdapter);        // 设置列表视图的长按监听器
```

02 由于步骤1将点击监听器和长按监听器设到了列表适配器，因此令BillListAdapter分别实现AdapterView.OnItemClickListener和AdapterView.OnItemLongClickListener，并且重写对应的点击方法onItemClick与长按方法onItemLongClick，其中onItemClick内部补充页面的跳转逻辑，而onItemLongClick内部补充提示窗的处理逻辑。重写之后的点击方法与长按方法代码如下：

```
    (完整代码见chapter07\src\main\java\com\example\chapter07\adapter\BillListAdapter.java)
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        BillInfo bill = mBillList.get(position);     // 获得当前位置的账单信息
        // 以下跳转到账单填写页面
        Intent intent = new Intent(mContext, BillAddActivity.class);
        intent.putExtra("xuhao", bill.xuhao);    // 携带账单序号，表示已存在该账单
        mContext.startActivity(intent);          // 因为已存在该账单，所以跳过去实际会编辑账单
    }

    @Override
    public boolean onItemLongClick(AdapterView<?> parent, View view, final int position, long
id) {
        BillInfo bill = mBillList.get(position);     // 获得当前位置的账单信息
        AlertDialog.Builder builder = new AlertDialog.Builder(mContext);
        String desc = String.format("是否删除以下账单？\n%s %s%d %s", bill.date,
                bill.type==0?"收入":"支出", (int) bill.amount, bill.desc);
        builder.setMessage(desc);                    // 设置提醒对话框的消息文本
        builder.setPositiveButton("是", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                deleteBill(position);                // 删除该账单
            }
        });
        builder.setNegativeButton("否", null);
        builder.create().show();                     // 显示提醒对话框
        return true;
    }
```

3. 合并账单的添加与编辑功能

上述第二点提到账单编辑页面仍然跳转到了BillAddActivity，然而该页面原本用作账单填写，若想让它同时支持账单编辑功能，则需从意图包裹中取出名为xuhao的字段，得到上个页面传来的序号数值，通过判断该字段是否为－1，再分别对应处理，后续的处理过程分成以下两个步骤：

01 若xuhao字段的值为－1，则表示不存在原账单的序号，此时应进入账单添加逻辑；若值不为－1，则表示已存在该账单序号，此时应进入账单编辑处理，也就是将数据库中查到的原账单信息展示在各输入框中，再由用户酌情修改详细的账单信息。相应的代码逻辑如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\BillAddActivity.java)
private int xuhao;                                      // 如果序号有值，说明已存在该账单
// 获取日历实例，里面包含了当前的年月日
private Calendar calendar = Calendar.getInstance();
private BillDBHelper mBillHelper;                       // 声明一个账单数据库的帮助器对象

@Override
protected void onResume() {
    super.onResume();
    xuhao = getIntent().getIntExtra("xuhao", -1);
    mBillHelper = BillDBHelper.getInstance(this);       // 获取账单数据库的帮助器对象
    if (xuhao != -1) {                                  // 序号有值，就展示数据库里的账单详情
        List<BillInfo> bill_list = (List<BillInfo>) mBillHelper.queryById(xuhao);
        if (bill_list.size() > 0) {                     // 已存在该账单
            BillInfo bill = bill_list.get(0);           // 获取账单信息
            Date date = DateUtil.formatString(bill.date);
            calendar.set(Calendar.YEAR, date.getYear()+1900);
            calendar.set(Calendar.MONTH, date.getMonth());
            calendar.set(Calendar.DAY_OF_MONTH, date.getDate());
            if (bill.type == 0) {                       // 收入
                rb_income.setChecked(true);
            } else {                                    // 支出
                rb_expand.setChecked(true);
            }
            et_desc.setText(bill.desc);                 // 设置账单的描述文本
            et_amount.setText(""+bill.amount);          // 设置账单的交易金额
        }
    }
    tv_date.setText(DateUtil.getDate(calendar));        // 设置账单的发生时间
}
```

02 在保存账单记录时，也要先判断数据库中是否已经存在对应账单，如果有找到对应的账单记录，那么执行记录更新操作，否则执行记录添加操作。对应的数据库的操作代码示例如下：

```
(完整代码见chapter07\src\main\java\com\example\chapter07\database\BillDBHelper.java)
public void save(BillInfo bill) {
    // 根据序号寻找对应的账单记录
    List<BillInfo> bill_list = (List<BillInfo>) queryById(bill.xuhao);
    BillInfo info = null;
    if (bill_list.size() > 0) {          // 有找到账单记录
        info = bill_list.get(0);
    }
    if (info != null) {                  // 已存在该账单信息，则更新账单
        bill.rowid = info.rowid;
        bill.create_time = info.create_time;
        bill.update_time = DateUtil.getNowDateTime("");
        update(bill);                    // 更新数据库记录
    } else {                             // 未存在该账单信息，则添加账单
```

```
            bill.create_time = DateUtil.getNowDateTime("");
            insert(bill);                    // 添加数据库记录
        }
    }
```

7.6 小 结

本章主要介绍了App开发的高级控件相关知识，包括：下拉框的用法（下拉框控件Spinner、数组适配器ArrayAdapter、简单适配器SimpleAdapter）、列表类视图的用法（基本适配器BaseAdapter、列表视图ListView、网格视图GridView）、翻页类视图的基本用法（翻页视图ViewPager、翻页适配器PagerAdapter、翻页标签栏PagerTabStrip）、碎片的两种用法（静态注册方式、动态注册方式、碎片适配器FragmentPagerAdapter）。中间穿插了实战模块的运用，如改进后的购物车、改进后的启动引导页等。最后设计了一个实战项目“记账本”，在该项目的App编码中用到了前面介绍的大部分控件，从而加深了对所学知识的理解。

通过本章的学习，读者应该能够掌握以下4种开发技能：

（1）学会使用下拉框控件。

（2）学会使用列表视图和网格视图。

（3）学会使用翻页视图与翻页标签栏。

（4）学会通过两种注册方式分别使用碎片。

7.7 动手练习

请上机实验下列3项练习：

1. 将第6章的购物车界面中的商品列表改造为列表视图，将商城界面中的商品列表改造为网格视图。

2. 联合运用翻页视图与碎片，实现App启动之时的欢迎引导页面。

3. 实践本章的记账本项目，要求实现账单的增加、删除、修改、查看功能，并支持账单的列表展示与分月浏览。

第 8 章

自定义控件

本章介绍App开发中的一些自定义控件技术，主要包括：视图是如何从无到有构建出来的、如何改造已有的控件变出新控件、如何将消息主动推送到通知栏、如何通过持续绘制实现简单动画。然后结合本章所学的知识，演示一个实战项目“广告轮播”的设计与实现。

8.1 视图的构建过程

本节介绍一个视图的构建过程，包括：如何编写视图的构造方法，4个构造方法之间有什么区别；如何测量实体的实际尺寸，包含文本、图像、线性视图的测量办法；如何利用画笔绘制视图的界面，并说明onDraw方法与dispatchDraw方法的先后执行顺序。

8.1.1 视图的构造方法

Android自带的控件往往外观欠佳，开发者常常需要修改某些属性，比如按钮控件Button就有好几个问题，其一字号太小，其二文字颜色太浅，其三字母默认大写。于是XML文件中的每个Button节点都得添加textSize、textColor、textAllCaps 3个属性，以便定制按钮的字号、文字颜色和大小写开关，就像下面这样：

```
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Hello World"
    android:textAllCaps="false"
    android:textColor="#000000"
    android:textSize="20sp" />
```

如果只是一两个按钮控件倒还好办，倘若App的许多页面都有很多Button，为了统一按钮风格，就得给全部Button节点都加上这些属性。要是哪天产品大姐心血来潮，命令将所有按钮统统换成另一种风格，如此多的Button节点只好逐个修改过去，令人苦不堪言。为此可以考虑把按钮样式提炼出来，将统一的按钮风格定义在某个地方，每个Button节点引用统一样式便行。为此打开res/values目录下的styles.xml，在resources节点内部补充如下所示的风格配置定义：

```
<style name="CommonButton">
    <item name="android:textAllCaps">false</item>
    <item name="android:textColor">#000000</item>
```

```
    <item name="android:textSize">20sp</item>
</style>
```

接着回到XML布局文件中，给Button节点添加形如“style="@style/样式名称"”的引用说明，表示当前控件将覆盖指定的属性样式，添加样式引用后的Button节点内容如下：

```
(完整代码见chapter08\src\main\res\layout\activity_custom_button.xml)
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="这是来自style的Button"
    style="@style/CommonButton" />
```

运行测试App，打开的按钮界面如图8-1所示，对比默认的按钮控件，可见通过style引用的按钮果然变了个模样。以后若要统一更换所有按钮的样式，只需修改styles.xml中的样式配置即可。



图 8-1　通过 style 属性设置样式的按钮界面

然而样式引用仍有不足之处，因为只有Button节点添加了style属性才奏效，要是忘了添加style属性就不管用了，而且样式引用只能修改已有的属性，不能添加新属性，也不能添加新方法。若想更灵活地定制控件外观，就要通过自定义控件实现了。

自定义控件听起来很复杂的样子，其实并不高深，不管控件还是布局，它们本质上都是一个Java类，也拥有自身的构造方法。以视图基类View为例，它有4个构造方法，分别介绍如下：

（1）带1个参数的构造方法public View(Context context)，在Java代码中通过new关键字创建视图对象时，会调用这个构造方法。

（2）带2个参数的构造方法public View(Context context, AttributeSet attrs)，在XML文件中添加视图节点时，会调用这个构造方法。

（3）带3个参数的构造方法public View(Context context, AttributeSet attrs, int defStyleAttr)，在采取默认的样式属性时，会调用这个构造方法。如果defStyleAttr填0，则表示没有默认的样式。

（4）带4个参数的构造方法public View(Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes)，在采取默认的样式资源时，会调用这个构造方法。如果defStyleRes填0，则表示无样式资源。

以上的4个构造方法中，前两个必须实现，否则要么不能在代码中创建视图对象，要么不能在XML文件中添加视图节点；至于后两个构造方法，则与styles.xml中的样式配置有关。先看带3个参数的构造方法，第3个参数defStyleAttr的意思是指定默认的样式属性，这个样式属性在res/values下面的attrs.xml中配置，如果values目录下没有attrs.xml就创建该文件，并填入以下的样式属性配置：

```
<resources>
    <declare-styleable name="CustomButton">
        <attr name="customButtonStyle" format="reference" />
    </declare-styleable>
</resources>
```

以上的配置内容表明了属性名称为customButtonStyle，属性格式为引用类型reference，也就是实际样式在别的地方定义，这个地方便是styles.xml中定义的样式配置。可是customButtonStyle怎样与styles.xml里的CommonButton样式关联起来呢？每当开发者创建新项目时，AndroidManifest. xml

的application节点都设置了主题属性，通常为android:theme="@style/AppTheme"，这个默认主题来自styles.xml的AppTheme，打开styles.xml发现文件开头的AppTheme配置定义如下：

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
    <item name="customButtonStyle">@style/CommonButton</item>
</style>
```

原来App的默认主题源自Theme.AppCompat.Light.DarkActionBar，其中的Light表示这是亮色主题，DarkActionBar表示顶部标题栏是暗色的，内部的3个color项指定了该主题采用的部分颜色。现在给AppTheme添加一项customButtonStyle，并指定该项的样式为@style/CommonButton，修改后的AppTheme配置示例如下：

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
    <item name="customButtonStyle">@style/CommonButton</item>
</style>
```

接着到Java代码包中编写自定义的按钮控件，控件代码如下所示，注意在defStyleAttr处填上默认的样式属性R.attr.customButtonStyle。

```
(完整代码见chapter08\src\main\java\com\example\chapter08\widget\CustomButton.java)
public class CustomButton extends Button {
    public CustomButton(Context context) {
        super(context);
    }
    public CustomButton(Context context, AttributeSet attrs) {
        this(context, attrs, R.attr.customButtonStyle);  // 设置默认的样式属性
    }
    public CustomButton(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }
}
```

然后打开测试界面的XML布局文件activity_custom_button.xml，添加如下所示的自定义控件节点CustomButton：

```
(完整代码见chapter08\src\main\res\layout\activity_custom_button.xml)
<!-- 注意自定义控件需要指定该控件的完整路径 -->
<com.example.chapter08.widget.CustomButton
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="这是自定义的Button"
    android:background="#ffff00" />
```

运行测试App，此时按钮界面如图8-2所示，可见第三个按钮也就是自定义的按钮控件字号变大、文字变黑，同时按钮的默认背景不见了，文字也不居中对齐了。

查看系统自带的按钮Button源码，发现它的构造方法是下面这样的：



图 8-2　通过自定义控件的按钮界面

```
    public Button(Context context, AttributeSet attrs) {
        this(context, attrs, com.android.internal.R.attr.buttonStyle);
    }
```

可见按钮控件的默认样式都写在系统内核的com.android.internal.R.attr.buttonStyle之中了，难怪Button与TextView的外观有所差异，原来是默认的样式属性造成的。

不过defStyleAttr的实现过程稍显烦琐，既要在styles.xml中配置好样式，又要在attrs.xml中添加样式属性定义，末了还得在App的当前主题中关联样式属性与样式配置。为简化操作，视图对象带4个参数的构造方法便派上用场了，第4个参数defStyleRes允许直接传入样式配置的资源名称，例如R.style.CommonButton就能直接指定当前视图的样式风格，于是defStyleAttr的3个步骤简化为defStyleRes的1个步骤，也就是只需在styles.xml中配置样式风格。此时自定义控件的代码就要将后两个构造方法改成下面这样：

```
    public CustomButton(Context context, AttributeSet attrs, int defStyleAttr) {
        // 下面不使用defStyleAttr，直接使用R.style.CommonButton定义的样式
        this(context, attrs, 0, R.style.CommonButton);
    }
    @SuppressLint("NewApi")
    public CustomButton(Context context, AttributeSet attrs, int defStyleAttr, int
defStyleRes) {
        super(context, attrs, defStyleAttr, defStyleRes);
    }
```

由于styles.xml定义的样式风格允许用在多个地方，包括XML文件中的style属性、构造方法中的defStyleAttr（对应当前主题）、构造方法中的defStyleRes，如果这三处地方分别引用了不同的样式，控件又该呈现什么样的风格呢？对于不同来源的样式配置，Android给每个来源都分配了优先级，优先级越大的来源，其样式会优先展示。至于上述的三处来源，它们之间的优先级顺序为：style属性>defStyleAttr>defStyleRes，也就是说，XML文件的style属性所引用的样式资源优先级最高，而defStyleRes所引用的样式资源优先级最低。

8.1.2　视图的测量方法

构造方法只是自定义控件的第一步，自定义控件的第二步是测量尺寸，也就是重写onMeasure方法。要想把自定义的控件画到界面上，首先得知道这个控件的宽高尺寸，而控件的宽和高在XML文件中分别由layout_width属性和layout_height属性规定，它们有3种赋值方式，具体说明见表8-1。

表 8-1　宽高尺寸的 3 种赋值方式

XML 中的尺寸类型	LayoutParams 类的尺寸类型	说　明
match_parent	MATCH_PARENT	与上级视图大小一样
wrap_content	WRAP_CONTENT	按照自身尺寸进行适配
**dp	整型数	具体的尺寸数值

方式1和方式3都较简单，要么取上级视图的数值，要么取具体数值。难办的是方式2，这个尺寸究竟要如何度量，总不可能让开发者拿着尺子在屏幕上比划吧。当然，Android提供了相关度量方法，支持在不同情况下测量尺寸。需要测量的实体主要有3种，分别是文本尺寸、图形尺寸和布局尺寸，依次说明如下。

1. 文本尺寸测量

文本尺寸分为文本的宽度和高度，需根据文本大小分别计算。其中，文本宽度使用Paint类的measureText方法测量，具体代码如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\util\MeasureUtil.java)
// 获取指定文本的宽度（其实就是长度）
public static float getTextWidth(String text, float textSize) {
    if (TextUtils.isEmpty(text)) {
        return 0;
    }
    Paint paint = new Paint();                  // 创建一个画笔对象
    paint.setTextSize(textSize);                // 设置画笔的文本大小
    return paint.measureText(text);             // 利用画笔丈量指定文本的宽度
}
```

至于文本高度的计算用到了FontMetrics类，该类提供了5个与高度相关的属性，详细说明见表8-2。

表 8-2　FontMetrics 类的距离属性说明

FontMetrics 类的距离属性	说　明
top	行的顶部与基线的距离
ascent	字符的顶部与基线的距离
descent	字符的底部与基线的距离
bottom	行的底部与基线的距离
leading	行间距

之所以区分这些属性，是为了计算不同规格的高度。如果要得到文本自身的高度，则高度值=descent－ascent；如果要得到文本所在行的行高，则高度值=bottom－top+leading。以计算文本高度为例，具体的计算代码如下：

```
// 获取指定文本的高度
public static float getTextHeight(String text, float textSize) {
    Paint paint = new Paint();                      // 创建一个画笔对象
    paint.setTextSize(textSize);                    // 设置画笔的文本大小
    FontMetrics fm = paint.getFontMetrics();        // 获取画笔默认字体的度量衡
    return fm.descent - fm.ascent;                  // 返回文本自身的高度
    //return fm.bottom - fm.top + fm.leading;       // 返回文本所在行的行高
}
```

下面观察文本尺寸的度量结果，当字体大小为17sp时，示例文本的宽度为119、高度为19，如图8-3所示；当字体大小为25sp时，示例文本的宽度为175、高度为29，如图8-4所示。

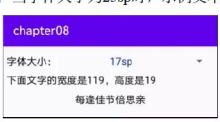


图 8-3　字体大小为 17sp 时的尺寸

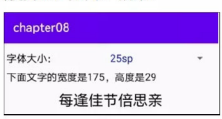


图 8-4　字体大小为 25sp 时的尺寸

2. 图形尺寸测量

相对于文本尺寸，图形尺寸的计算反而简单些，因为Android提供了现成的宽、高获取方法。

如果图形是Bitmap格式，就通过getWidth方法获取位图对象的宽度，通过getHeight方法获取位图对象的高度；如果图形是Drawable格式，就通过getIntrinsicWidth方法获取图形对象的宽度，通过getIntrinsicHeight方法获取图形对象的高度。

3. 布局尺寸测量

文本尺寸测量主要用于TextView、Button等文本控件，图形尺寸测量主要用于ImageView、ImageButton等图像控件。在实际开发中，有更多场合需要测量布局视图的尺寸。由于布局视图内部可能有文本控件、图像控件，还可能有padding和margin，因此，逐个测量布局的内部控件是不现实的。幸而View类提供了一种测量整体布局的思路，对应layout_width和layout_height的3种赋值方式，Android的视图基类同样提供了3种测量模式，具体取值说明见表8-3。

表 8-3　测量模式的取值说明

MeasureSpec 类的测量模式	视图宽、高的赋值方式	说　明
AT_MOST	MATCH_PARENT	达到最大
UNSPECIFIED	WRAP_CONTENT	未指定（实际就是自适应）
EXACTLY	具体 dp 值	精确尺寸

围绕这3种测量模式衍生了相关度量方法，如ViewGroup类的getChildMeasureSpec方法（获取下级视图的测量规格）、MeasureSpec类的makeMeasureSpec方法（根据指定参数制定测量规格）、View类的measure方法（按照测量规格进行测量操作）等。以线性布局为例，详细的布局高度测量代码如下：

```
// 计算指定线性布局的实际高度
public static float getRealHeight(View child) {
    LinearLayout llayout = (LinearLayout) child;
    // 获得线性布局的布局参数
    ViewGroup.LayoutParams params = llayout.getLayoutParams();
    if (params == null) {
        params = new ViewGroup.LayoutParams(
                LayoutParams.MATCH_PARENT, LayoutParams.WRAP_CONTENT);
    }
    // 获得布局参数里面的宽度规格
    int wdSpec = ViewGroup.getChildMeasureSpec(0, 0, params.width);
    int htSpec;
    if (params.height > 0) {            // 高度大于0，说明这是明确的dp数值
        // 按照精确数值的情况计算高度规格
        htSpec = View.MeasureSpec.makeMeasureSpec(params.height, MeasureSpec.EXACTLY);
    } else {  // MATCH_PARENT=-1，WRAP_CONTENT=-2，所以二者都进入该分支
        // 按照不确定的情况计算高度规则
        htSpec = View.MeasureSpec.makeMeasureSpec(0, MeasureSpec.UNSPECIFIED);
    }
    llayout.measure(wdSpec, htSpec);    // 重新丈量线性布局的宽、高
    // 获得并返回线性布局丈量之后的高度。调用getMeasuredWidth方法可获得宽度
    return llayout.getMeasuredHeight();
}
```

现在很多App页面都提供了下拉刷新功能，这需要计算下拉刷新的头部高度，以便在下拉时判断整个页面要拉动多少距离。比如图8-5所示的下拉刷新头部，对应的XML源码路径为chapter08\src\main\res\layout\drag_drop_header.xml，其中包含图像、文字和间隔，调用getRealHeight方法计算得到的布局高度为342。

以上的几种尺寸测量办法看似复杂，其实相关的测量逻辑早已封装在View和ViewGroup之中，开发者自定义的视图一般无须重写onMeasure方法；就算重写了onMeasure方法，也可调用getMeasuredWidth方法获得测量完成的宽度，调用getMeasuredHeight方法获得测量完成的高度。



图 8-5　布局视图的高度测量结果

8.1.3　视图的绘制方法

测量完控件的宽和高，接下来就要绘制控件图案了，此时可以重写两个视图绘制方法，分别是onDraw和dispatchDraw，它们的区别主要有下列两点：

（1）onDraw既可用于普通控件，也可用于布局类视图；而dispatchDraw专门用于布局类视图，像线性布局LinearLayout、相对布局RelativeLayout都属于布局类视图。

（2）onDraw方法先执行，dispatchDraw方法后执行，这两个方法中间再执行下级视图的绘制方法。比如App界面有个线性布局A，且线性布局内部有个相对布局B，同时相对布局B内部又有个文本视图C，则它们的绘制方法执行顺序为：线性布局A的onDraw方法→相对布局B的onDraw方法→文本视图C的onDraw方法→相对布局B的dispatchDraw方法→线性布局A的dispatchDraw方法，更直观的绘图顺序参见图8-6。

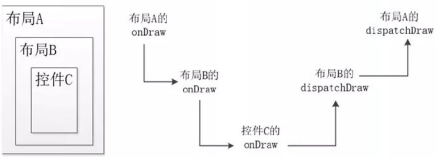


图 8-6　多个视图嵌套时的绘图次序

不管是onDraw方法还是dispatchDraw方法，它们的入参都是Canvas画布对象，在画布上绘图相当于在屏幕上绘图。绘图本身是个很大的课题，画布的用法也多种多样，单单Canvas便提供了3类方法：划定可绘制的区域、在区域内部绘制图形、画布的控制操作，分别说明如下。

1. 划定可绘制的区域

虽然视图内部的所有区域都允许绘制，但是有时候开发者只想在某个矩形区域内部画画，这时就得事先指定允许绘图的区域界限，相关方法说明如下：

- clipPath：**裁剪不规则曲线区域**。
- clipRect：**裁剪矩形区域**。
- clipRegion：**裁剪一块组合区域**。

2. 在区域内部绘制图形

该类方法用来绘制各种基本的几何图形，相关方法说明如下：

- drawArc：**绘制扇形或弧形。第4个参数为true时画扇形，为false时画弧形**。

- drawBitmap：**绘制位图**。
- drawCircle：**绘制圆形**。
- drawLine：**绘制直线**。
- drawOval：**绘制椭圆**。
- drawPath：**绘制路径，即不规则曲线**。
- drawPoint：**绘制点**。
- drawRect：**绘制矩形**。
- drawRoundRect：**绘制圆角矩形**。
- drawText：**绘制文本**。

3. 画布的控制操作

控制操作包括画布的旋转、缩放、平移以及存取画布状态的操作，相关方法说明如下：

- rotate：**旋转画布**。
- scale：**缩放画布**。
- translate：**平移画布**。
- save：**保存画布状态**。
- restore：**恢复画布状态**。

上述第二大点提到的draw***方法只是准备绘制某种几何图形，真正的细节描绘还要靠画笔工具Paint实现。Paint类定义了画笔的颜色、样式、粗细、阴影等，常用方法说明如下：

- setAntiAlias：**设置是否使用抗锯齿功能。主要用于画圆圈等曲线**。
- setDither：**设置是否使用防抖动功能**。
- setColor：**设置画笔的颜色**。
- setShadowLayer：**设置画笔的阴影区域与颜色**。
- setStyle：**设置画笔的样式**。Style.STROKE**表示线条**，Style.FILL**表示填充**。
- setStrokeWidth：**设置画笔线条的宽度**。

接下来演示如何通过画布和画笔描绘不同的几何图形，以绘制圆角矩形与绘制椭圆为例，重写后的onDraw方法示例如下：

(完整代码见chapter08\src\main\java\com\example\chapter08\widget\DrawRelativeLayout.java)

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    int width = getMeasuredWidth();                          // 获得布局的实际宽度
    int height = getMeasuredHeight();                        // 获得布局的实际高度
    if (width > 0 && height > 0) {
        if (mDrawType == 2) {                                // 绘制圆角矩形
            RectF rectF = new RectF(0, 0, width, height);
            canvas.drawRoundRect(rectF, 30, 30, mPaint);     // 在画布上绘制圆角矩形
        } else if (mDrawType == 4) {                         // 绘制椭圆
            RectF oval = new RectF(0, 0, width, height);
            canvas.drawOval(oval, mPaint);                   // 在画布上绘制椭圆
        }
    }
}
```

运行测试App，即可观察实际的绘图效果，其中调用drawRoundRect方法绘制圆角矩形的界面如图8-7所示，调用drawOval方法绘制椭圆的界面如图8-8所示。



图 8-7　绘制圆角矩形

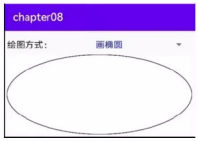


图 8-8　绘制椭圆

由于onDraw方法的调用在绘制下级视图之前，而dispatchDraw方法的调用在绘制下级视图之后，因此如果希望当前视图不被下级视图覆盖，就只能在dispatchDraw方法中绘图。下面是分别在onDraw和dispatchDraw两个方法中绘制矩形及其对角线的代码例子：

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    int width = getMeasuredWidth();                 // 获得布局的实际宽度
    int height = getMeasuredHeight();               // 获得布局的实际高度
    if (width > 0 && height > 0) {
        if (mDrawType == 5) {                       // 绘制矩形及其对角线
            Rect rect = new Rect(0, 0, width, height);
            canvas.drawRect(rect, mPaint);          // 绘制矩形
            // 绘制左上角到右下角的线段
            canvas.drawLine(0, 0, width, height, mPaint);
            // 绘制左下角到右上角的线段
            canvas.drawLine(0, height, width, 0, mPaint);
        }
    }
}

@Override
protected void dispatchDraw(Canvas canvas) {
    super.dispatchDraw(canvas);
    int width = getMeasuredWidth();                 // 获得布局的实际宽度
    int height = getMeasuredHeight();               // 获得布局的实际高度
    if (width > 0 && height > 0) {
        if (mDrawType == 6) {                       // 绘制矩形及其对角线
            Rect rect = new Rect(0, 0, width, height);
            canvas.drawRect(rect, mPaint);          // 绘制矩形
            // 绘制左上角到右下角的线段
            canvas.drawLine(0, 0, width, height, mPaint);
            // 绘制左下角到右上角的线段
            canvas.drawLine(0, height, width, 0, mPaint);
        }
    }
}
```

实验用的界面布局片段示例如下，主要观察对角线是否遮住内部的按钮控件：

```
(完整代码见chapter08\src\main\res\layout\activity_show_draw.xml)
<!-- 自定义的绘画视图，需要使用全路径 -->
<com.example.chapter08.widget.DrawRelativeLayout
    android:id="@+id/drl_content"
    android:layout_width="match_parent"
```

```
    android:layout_height="150dp" >
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="我在中间" />
</com.example.chapter08.widget.DrawRelativeLayout>
```

运行测试App，发现使用onDraw绘图的界面如图8-9所示，使用dispatchDraw绘图的界面如图8-10所示。

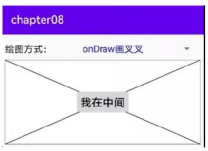


图 8-9　重写 onDraw 方法

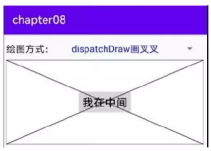


图 8-10　重写 dispatchDraw 方法

对比可见，调用onDraw方法绘制对角线时，中间的按钮遮住了对角线；调用dispatchDraw方法绘制对角线时，对角线没被按钮遮住，依然显示在视图中央。

8.2　改造已有的控件

本节介绍如何对现有控件加以改造，使之变成具备不同功能的新控件，包括：如何基于日期选择器实现月份选择器，如何给翻页标签栏添加文字样式属性，如何在滚动视图中展示完整的列表视图。

8.2.1　自定义月份选择器

虽然Android提供了许多控件，但是仍然不够用，比如系统自带日期选择器DatePicker和时间选择器TimePicker，却没有月份选择器MonthPicker，倘若希望选择某个月份，一时之间叫人不知如何是好。不过为什么支付宝的账单查询支持选择月份呢？就像图8-11所示的支付宝查询账单页面，分明可以单独选择年月。



图 8-11　支付宝的账单月份选择界面

看上去，支付宝的年月控件仿佛系统自带的日期选择器，区别在于去掉了右侧的日子列表。二者之间如此相似，这可不是偶然撞衫，而是它们本来系出一源。只要把日期选择器稍加修改，想办法隐藏右边多余的日子列，即可实现移花接木的效果。下面是将日期选择器修改之后变成月份选择器的代码例子：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\widget\MonthPicker.java)
// 由日期选择器派生出月份选择器
public class MonthPicker extends DatePicker {
    public MonthPicker(Context context, AttributeSet attrs) {
```

```
            super(context, attrs);
            // 获取年月日的下拉列表项
            ViewGroup vg = ((ViewGroup) ((ViewGroup) getChildAt(0)).getChildAt(0));
            if (vg.getChildCount() == 3) {  // 拥有3个下级视图
                // 有的机型显示格式为“年月日”，此时隐藏第3个控件
                vg.getChildAt(2).setVisibility(View.GONE);
            } else if (vg.getChildCount() == 5) {  // 拥有5个下级视图
                // 有的机型显示格式为“年|月|日”，此时隐藏第4个和第5个控件（即“|日”）
                vg.getChildAt(3).setVisibility(View.GONE);
                vg.getChildAt(4).setVisibility(View.GONE);
            }
        }
    }
```

由于日期选择器有日历和下拉框两种展示形式，上面的月份选择器代码只对下拉框生效，因此在布局文件中添加月份选择器之时，要特别注意添加属性“android:datePickerMode="spinner"”，表示该控件采取下拉列表显示；并添加属性“android:calendarViewShown="false"”，表示不显示日历视图。月份选择器在布局文件中的定义例子如下：

```
(完整代码见chapter08\src\main\res\layout\activity_month_picker.xml)
<!-- 自定义的月份选择器，需要使用全路径 -->
<com.example.chapter08.widget.MonthPicker
    android:id="@+id/mp_month"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:calendarViewShown="false"
    android:datePickerMode="spinner" />
```

这下大功告成，重新包装后的月份选择器俨然也是日期时间控件家族的一员，不但继承了日期选择器的所有方法，而且控件界面与支付宝的几乎一样。月份选择器的界面效果如图8-12所示，果然只展示年份和月份了。



图 8-12 月份选择器的界面效果

8.2.2 给翻页标签栏添加新属性

前面介绍的月份选择器，是以日期选择器为基础，只保留年月两项同时屏蔽日子而成，这属于在现有控件上做减法。反过来，也允许在现有控件上做加法，也就是给控件增加新的属性或者新的方法。例如第7章的“7.3.2 翻页标签栏PagerTabStrip”，提到PagerTabStrip无法在XML文件中设置文本大小和文本颜色，只能在Java代码中调用setTextSize和setTextColor方法。这让人很不习惯，最好能够在XML文件中直接指定textSize和textColor属性。接下来通过自定义属性来扩展PagerTabStrip，以便在布局文件中指定文字大小和文字颜色的属性。具体步骤说明如下：

01 在res\values目录下创建attrs.xml。其中，declare-styleable的name属性值表示新控件名为CustomPagerTab，两个attr节点表示新增的两个属性分别是textColor和textSize。文件内容如下：

```
<resources>
    <declare-styleable name="CustomPagerTab">
        <attr name="textColor" format="color" />
        <attr name="textSize" format="dimension" />
    </declare-styleable>
</resources>
```

02 在Java代码的widget目录下创建CustomPagerTab.java，填入以下代码：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\widget\CustomPagerTab.java)
public class CustomPagerTab extends PagerTabStrip {
    private int textColor = Color.BLACK;     // 文本颜色
    private int textSize = 15;               // 文本大小

    public CustomPagerTab(Context context) {
        super(context);
    }

    public CustomPagerTab(Context context, AttributeSet attrs) {
        super(context, attrs);
        if (attrs != null) {
            // 根据CustomPagerTab的属性定义，从XML文件中获取属性数组描述
            TypedArray attrArray = context.obtainStyledAttributes(
                    attrs, R.styleable.CustomPagerTab);
            // 根据属性描述定义，获取XML文件中的文本颜色
            textColor = attrArray.getColor(
                    R.styleable.CustomPagerTab textColor, textColor);
            // 根据属性描述定义，获取XML文件中的文本大小
            // getDimension得到的是px值，需要转换为sp值
            textSize = Utils.px2sp(context, attrArray.getDimension(
                  R.styleable.CustomPagerTab_textSize, textSize));
            attrArray.recycle();  // 回收属性数组描述
        }
    }

    @Override
    protected void onDraw(Canvas canvas) {       // 绘制方法
        setTextColor(textColor);                 // 设置标题文字的文本颜色
        // 设置标题文字的文本大小
        setTextSize(TypedValue.COMPLEX_UNIT_SP, textSize);
        super.onDraw(canvas);
    }
}
```

03 给演示页面的XML文件根节点增加命名空间声明xmlns:app="http://schemas.android.com/apk/res-auto"，再把PagerTabStrip的节点名称改为自定义控件的全路径名称（如com.example.chapter08.widget.CustomPagerTab），同时在该节点下添加两个新属性——app:textColor与app:textSize，也就是在XML文件中指定标签文本的颜色与大小。修改后的XML文件如下：

```
(完整代码见chapter08\src\main\res\layout\activity_custom_tab.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <androidx.viewpager.widget.ViewPager
        android:id="@+id/vp_content"
        android:layout_width="match_parent"
        android:layout_height="360dp" >
        <!-- 这里使用自定义控件的全路径名称，其中textColor和textSize为自定义的属性 -->
        <com.example.chapter08.widget.CustomPagerTab
            android:id="@+id/pts_tab"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:textColor="#ff0000"
            app:textSize="17sp" />
    </androidx.viewpager.widget.ViewPager>
</LinearLayout>
```

完成以上3个步骤的修改之后，运行测试App，打开的翻页界面如图8-13所示，可见此时翻页标签栏的标题文字变为红色，字体也变大了。

注意上述自定义控件的步骤1，attrs.xml里面attr节点的name表示新属性的名称，format表示新属性的数据格式；而在步骤2中，调用getColor方法获取颜色值，调用getDimensionPixelSize方法获取文字大小，不同的数据格式需要调用不同的获取方法。有关属性格式及其获取方法的对应说明见表8-4。



图 8-13　使用自定义属性的翻页标签栏

表 8-4　属性格式的取值说明

属性格式的名称	Java 代码的获取方法	XML 布局文件中的属性值说明
boolean	getBoolean	布尔值。取值为 true 或 false
integer	getInt	整型值
float	getFloat	浮点值
string	getString	字符串
color	getColor	颜色值。取值为开头带#的 6 位或 8 位十六进制数
dimension	getDimensionPixelSize	尺寸值。单位为 px
reference	getResourceId	参考某一资源。取值如@drawable/ic_launcher
enum	getInt	枚举值
flag	getInt	标志位

8.2.3　不滚动的列表视图

一个视图的宽和高，其实在页面布局的时候就决定了，视图节点的android:layout_width属性指定了该视图的宽度，而android:layout_height属性指定了该视图的高度。这两个属性又有3种取值方式，分别是：取值match_parent表示与上级视图一样尺寸，取值wrap_content表示按照自身内容的实际尺寸，最后一种则直接指定了具体的dp数值。在多数情况之下，系统按照这3种取值方式，完全能够自动计算正确的视图宽度和视图高度。

当然也有例外，像列表视图ListView就是个另类，尽管ListView在多数场合的高度计算不会出错，但是把它放到ScrollView之中便出现问题了。ScrollView本身叫作滚动视图，而列表视图ListView也是允许滚动的，于是一个滚动视图嵌套另一个也能滚动的视图，那么在双方的重叠区域，上下滑动的手势究竟表示要滚动哪个视图？这个滚动冲突的问题，不仅令开发者糊里糊涂，便是Android系统也得神经错乱。所以Android目前的处理对策是：如果ListView的高度被设置为wrap_content，则此时列表视图只显示一行的高度，然后整个界面只支持滚动ScrollView。

如此虽然滚动冲突的问题暂时解决，但是又带来一个新问题，好好的列表视图仅仅显示一行内容，这让出不了头的剩余列表情何以堪？按照用户正常的思维逻辑，列表视图应该显示所有行，并且列表内容要跟着整个页面一齐向上或者向下滚动。显然此时系统对ListView的默认处理方式并不符合用户习惯，只能对其改造使之满足用户的使用习惯。改造列表视图的一个可行方案，便是重写它的测量方法onMeasure，不管布局文件中设定的视图高度为何，都把列表视图的高度改为最大高度，即所有列表项高度加起来的总高度。

根据以上思路，编写一个扩展自ListView的不滚动列表视图NoScrollListView，它的实现代码如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\widget\NoScrollListView.java)
public class NoScrollListView extends ListView {
    public NoScrollListView(Context context) {
        super(context);
    }
    public NoScrollListView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    public NoScrollListView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }

    // 重写onMeasure方法，以便自行设定视图的高度
    @Override
    public void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        // 将高度设为最大值，即所有项加起来的总高度
        int expandSpec = MeasureSpec.makeMeasureSpec(
                Integer.MAX_VALUE >> 2, MeasureSpec.AT_MOST);
        // 按照新的高度规格重新测量视图尺寸
        super.onMeasure(widthMeasureSpec, expandSpec);
    }
}
```

接下来演示改造前后的列表视图界面效果，先在测试页面的XML文件中添加ScrollView节点，再在该节点下挂ListView节点，以及自定义的NoScrollListView节点。修改后的XML文件内容如下：

```
(完整代码见chapter08\src\main\res\layout\activity_noscroll_list.xml)
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="下面是系统自带的列表视图" />
        <ListView
            android:id="@+id/lv_planet"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="下面是自定义的列表视图" />
        <!-- 自定义的不滚动列表视图，需要使用全路径 -->
        <com.example.chapter08.widget.NoScrollListView
            android:id="@+id/nslv_planet"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</ScrollView>
```

回到该页面的活动代码，给ListView和NoScrollListView两个控件设置一模一样的行星列表，具体的Java代码如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\NoscrollListActivity.java)
public class NoscrollListActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_noscroll_list);
        PlanetListAdapter adapter1 = new PlanetListAdapter(
                this, Planet.getDefaultList());
        // 从布局文件中获取名为lv_planet的列表视图
        // lv_planet是系统自带的ListView，被ScrollView嵌套只能显示一行
        ListView lv_planet = findViewById(R.id.lv_planet);
        lv_planet.setAdapter(adapter1);                // 设置列表视图的行星适配器
        PlanetListAdapter adapter2 = new PlanetListAdapter(
                this, Planet.getDefaultList());
        // 从布局文件中获取名为nslv_planet的不滚动列表视图
        // nslv_planet是自定义控件NoScrollListView，会显示所有行
        NoScrollListView nslv_planet = findViewById(R.id.nslv_planet);
        nslv_planet.setAdapter(adapter2);              // 设置不滚动列表视图的行星适配器
    }
}
```

重新运行测试App，打开的行星列表界面如图8-14所示，可见系统自带的列表视图仅仅显示一条行星记录，而自定义的不滚动列表视图把所有行星记录都展示出来了。

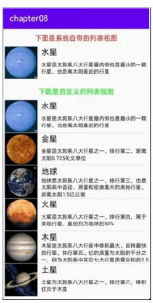


图 8-14　不滚动列表视图的界面效果

8.3　推送消息通知

本节介绍消息通知的推送过程及其具体用法，包括：通知由哪几个部分组成，如何构建并推送通知，如何区分各种通知渠道及其重要性，如何让服务呈现在前台运行，也就是利用通知管理器把服务推送到系统通知栏，以及如何使用悬浮窗技术模拟屏幕顶端的悬浮消息通知。

8.3.1　通知推送 Notification

在App的运行过程中，用户想购买哪件商品，想浏览哪条新闻，通常都由自己主动寻找并打开

对应的页面。当然用户不可避免地会漏掉部分有用的信息，例如购物车里的某件商品降价了，又如刚刚报道了某条突发新闻，这些很可能正是用户关注的信息。为了让用户及时收到此类信息，有必要由App主动向用户推送消息通知，以免错过有价值的信息。

在手机屏幕的顶端下拉会弹出通知栏，里面存放的便是App主动推送给用户的提醒消息，消息通知的组成内容由Notification类所描述。每条消息通知都有消息图标、消息标题、消息内容等基本元素，偶尔还有附加文本、进度条、计时器等额外元素，这些元素由通知建造器Notification.Builder所设定。下面是通知建造器的常用方法说明。

- setSmallIcon：**设置应用名称左边的小图标。这是必要方法，否则不会显示通知消息。**
- setLargeIcon：**设置通知栏右边的大图标。**
- setContentTitle：**设置通知栏的标题文本。**
- setContentText：**设置通知栏的内容文本。**
- setSubText：**设置通知栏的附加文本，它位于应用名称右边。**
- setProgress：**设置进度条并显示当前进度。进度条位于标题文本与内容文本下方。**
- setUsesChronometer：**设置是否显示计时器，计时器位于应用名称右边，它会动态显示从通知被推送到当前的时间间隔，计时器格式为“分钟：秒钟”。**
- setContentIntent：**设置通知内容的延迟意图**PendingIntent，**点击通知时触发该意图。调用**PendingIntent**的**getActivity**方法获得延迟意图对象，触发该意图等同于跳转到**getActivity**设定的活动页面。**
- setDeleteIntent：**设置删除通知的延迟意图**PendingIntent，**滑掉通知时触发该意图。**
- setAutoCancel：**设置是否自动清除通知。若为**true，**则点击通知后，通知会自动消失；若为**false，**则点击通知后，通知不会消失。**
- build：**构建通知。以上参数都设置完毕后，调用该方法返回**Notification**对象。**

注　意

Notification 仅仅描述了消息通知的组成内容，实际推送动作还需由通知管理器 NotificationManager 执行。NotificationManager 是系统通知服务的管理工具，要调用 getSystemService 方法，先从系统服务 NOTIFICATION_SERVICE 获取通知管理器，再调用管理器对象的消息操作方法。通知管理器的常用方法说明如下。

- notify：**把指定消息推送到通知栏。**
- cancel：**取消指定的消息通知。调用该方法后，通知栏中的指定消息将消失。**
- cancelAll：**取消所有的消息通知。**
- createNotificationChannel：**创建指定的通知渠道。**
- getNotificationChannel：**获取指定编号的通知渠道。**

以发送简单消息为例，它包括消息标题、消息内容、小图标、大图标等基本信息，则对应的通知推送代码示例如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\NotifySimpleActivity.java)
// 发送简单的通知消息（包括消息标题和消息内容）
private void sendSimpleNotify(String title, String message) {
    // 发送消息之前要先创建通知渠道，创建代码见MainApplication.java
```

```
        // 创建一个跳转到活动页面的意图
        Intent clickIntent = new Intent(this, MainActivity.class);
        // 创建一个用于页面跳转的延迟意图
        PendingIntent contentIntent = PendingIntent.getActivity(this,
                R.string.app_name, clickIntent,
                PendingIntent.FLAG_UPDATE_CURRENT);
        // 创建一个通知消息的建造器
        Notification.Builder builder = new Notification.Builder(this);
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            // 从Android 8.0开始必须给每个通知分配对应的渠道
            builder = new Notification.Builder(this,
                        getString(R.string.app_name));
        }
        builder.setContentIntent(contentIntent)       // 设置内容的点击意图
                .setAutoCancel(true)                  // 点击通知栏后是否自动清除该通知
                .setSmallIcon(R.mipmap.ic_launcher)   // 设置应用名称左边的小图标
                .setSubText("这里是副本")              // 设置通知栏里面的附加文本
                .setLargeIcon(BitmapFactory.decodeResource(getResources(),
                        R.drawable.ic_app))           // 设置通知栏右边的大图标
                .setContentTitle(title)               // 设置通知栏里面的标题文本
                .setContentText(message);             // 设置通知栏里面的内容文本
        Notification notify = builder.build();        // 根据通知建造器构建一个通知对象
        // 从系统服务中获取通知管理器
        NotificationManager notifyMgr = (NotificationManager)
                getSystemService(Context.NOTIFICATION_SERVICE);
        // 使用通知管理器推送通知，然后在手机的通知栏就会看到该消息
        notifyMgr.notify(R.string.app_name, notify);
    }
```

运行测试App，在点击发送按钮时触发sendSimpleNotify方法，手机的通知栏马上收到推送的简单消息，如图8-15所示。根据图示的文字标记，即可得知每种消息元素的位置。



图 8-15　简单消息的通知栏效果

如果消息通知包含计时器与进度条，则需调用消息建造器的setUsesChronometer与setProgress方法，计时消息的通知推送代码示例如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\NotifyCounterActivity.java)
// 发送计时的通知消息
private void sendCounterNotify(String title, String message) {
    // 发送消息之前要先创建通知渠道，创建代码见MainApplication.java
    // 创建一个跳转到活动页面的意图
    Intent cancelIntent = new Intent(this, MainActivity.class);
    // 创建一个用于页面跳转的延迟意图
    PendingIntent deleteIntent = PendingIntent.getActivity(this,
            R.string.app_name, cancelIntent,
            PendingIntent.FLAG_UPDATE_CURRENT);
    // 创建一个通知消息的建造器
    Notification.Builder builder = new Notification.Builder(this);
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        // 从Android 8.0开始必须给每个通知分配对应的渠道
        builder = new Notification.Builder(this,
                    getString(R.string.app_name));
    }
    builder.setDeleteIntent(deleteIntent)         // 设置内容的清除意图
            .setSmallIcon(R.mipmap.ic_launcher)   // 设置应用名称左边的小图标
            .setLargeIcon(BitmapFactory.decodeResource(getResources(),
                    R.drawable.ic_app))           // 设置通知栏右边的大图标
```

```
            .setProgress(100, 60, false)          // 设置进度条及其具体进度
            .setUsesChronometer(true)             // 设置是否显示计时器
            .setContentTitle(title)               // 设置通知栏里面的标题文本
            .setContentText(message);             // 设置通知栏里面的内容文本
        Notification notify = builder.build();    // 根据通知建造器构建一个通知对象
        // 从系统服务中获取通知管理器
        NotificationManager notifyMgr = (NotificationManager)
                getSystemService(Context.NOTIFICATION_SERVICE);
        // 使用通知管理器推送通知，然后在手机的通知栏中就会看到该消息
        notifyMgr.notify(R.string.app_name, notify);
    }
```

运行测试App，在点击发送按钮时触发sendCounterNotify方法，手机通知栏马上收到推送的计时消息，如图8-16所示。根据图示的文字标记，即可得知计时器和进度条的位置。

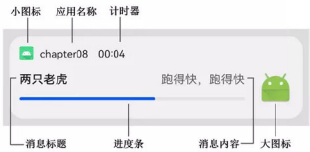


图 8-16　计时消息的通知栏效果

8.3.2　通知渠道 NotificationChannel

为了分清消息通知的轻重缓急，从Android 8.0开始新增了通知渠道，并且必须指定通知渠道才能正常推送消息。一个应用允许拥有多个通知渠道，每个渠道的重要性各不相同，有的渠道消息在通知栏被折叠成小行，有的渠道消息在通知栏展示完整的大行，有的渠道消息甚至会短暂悬浮于屏幕顶部，有的渠道消息在推送时会震动手机，有的渠道消息在推送时会发出铃声，有的渠道消息则完全静默推送，这些提示差别都有赖于通知渠道的特征设置。如果不考虑定制渠道特性，仅仅弄个默认渠道就去推送消息，那么只需以下3行代码即可创建默认的通知渠道：

```
// 从系统服务中获取通知管理器
NotificationManager notifyMgr = (NotificationManager)
        ctx.getSystemService(Context.NOTIFICATION_SERVICE);
// 创建指定编号、指定名称、指定级别的通知渠道
NotificationChannel channel = new NotificationChannel(channelId,
        channelName, NotificationManager.IMPORTANCE_DEFAULT);
notifyMgr.createNotificationChannel(channel);  // 创建指定的通知渠道
```

有了通知渠道之后，在推送消息之前使用该渠道创建对应的通知建造器，接着就能按照原方式推送消息了。使用通知渠道创建通知建造器的代码示例如下：

```
// 创建一个通知消息的建造器
Notification.Builder builder = new Notification.Builder(this);
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    // 从Android 8.0开始必须给每个通知分配对应的渠道
    builder = new Notification.Builder(this, channelId);
}
```

当然以上代码没有指定通知渠道的具体特征，消息通知的展示情况与提示方式完全由系统默认。若要个性化定制不同渠道的详细特征，就得单独设置渠道对象的各种特征属性。下面便是NotificationChannel提供的属性设置方法说明。

- setSound：**设置推送通知之时的铃声，若设为null表示静音推送。**
- enableLights：**推送消息时是否让呼吸灯闪烁。**
- enableVibration：**推送消息时是否让手机震动。**

- setShowBadge：**是否在应用图标的右上角展示小红点。**
- setLockscreenVisibility：**设置锁屏时候的可见性，可见性的取值说明见表**8-5。

表 8-5　锁屏时候的通知可见性取值说明

Notification 类的通知可见性	说　明
VISIBILITY_PUBLIC	显示所有通知信息
VISIBILITY_PRIVATE	只显示通知标题不显示通知内容
VISIBILITY_SECRET	不显示任何通知信息

- setImportance：**设置通知渠道的重要性，其实**NotificationChannel**的构造方法已经传入了重要性，所以该方法只在变更重要性时调用。重要性的取值说明见表**8-6。

表 8-6　通知重要性的取值说明

NotificationManagerr 类的通知重要性	说　明
IMPORTANCE_NONE	不重要。此时不显示通知
IMPORTANCE_MIN	最小级别。此时通知栏折叠，无提示声音，无锁屏通知
IMPORTANCE_LOW	有点重要。此时通知栏展开，无提示声音，有锁屏通知
IMPORTANCE_DEFAULT	一般重要。此时通知栏展开，有提示声音，有锁屏通知
IMPORTANCE_HIGH	非常重要。此时通知栏展开，有提示声音，有锁屏通知，在屏幕顶部短暂悬浮（有的手机需要在设置页面开启横幅）
IMPORTANCE_MAX	最高级别。具体行为同 IMPORTANCE_HIGH

特别注意：每个通知渠道一经创建，就不可重复创建，即使创建也是做无用功。因此在创建渠道之前，最好先调用通知管理器的getNotificationChannel方法，判断是否存在该编号的通知渠道，只有不存在的情况才要创建通知渠道。下面是通知渠道的创建代码例子：

```
    (完整代码见chapter08\src\main\java\com\example\chapter08\util\NotifyUtil.java)
    // 创建通知渠道。从Android 8.0开始必须给每个通知分配对应的渠道
    public static void createNotifyChannel(Context ctx, String channelId, String channelName,
int importance) {
        // 从系统服务中获取通知管理器
        NotificationManager notifyMgr = (NotificationManager)
                ctx.getSystemService(Context.NOTIFICATION_SERVICE);
        // 已经存在指定编号的通知渠道
        if (notifyMgr.getNotificationChannel(channelId) == null) {
            // 创建指定编号、指定名称、指定级别的通知渠道
            NotificationChannel channel = new NotificationChannel(channelId,
                    channelName, importance);
            channel.setSound(null, null);        // 设置推送通知时的铃声。null表示静音推送
            channel.enableLights(true);          // 通知渠道是否让呼吸灯闪烁
            channel.enableVibration(true);       // 通知渠道是否让手机震动
            channel.setShowBadge(true);          // 通知渠道是否在应用图标的右上角展示小红点
            // 设置锁屏时候的可见性，VISIBILITY_PRIVATE表明只显示通知标题不显示通知内容
            channel.setLockscreenVisibility(Notification.VISIBILITY_PRIVATE);
            channel.setImportance(importance);   // 设置通知渠道的重要性级别
            notifyMgr.createNotificationChannel(channel);          // 创建指定的通知渠道
        }
    }
```

尽管通知渠道提供了多种属性设置方法，但真正常用的莫过于重要性这个特征，它的演示代码参见chapter08\src\main\java\com\example\chapter08\NotifyChannelActivity.java。在测试页面推送各

重要性的消息的外观分别如图8-17～图8-19所示，其中图8-17为IMPORTANCE_MIN最小级别时的通知栏，可见该通知被折叠了，只显示消息标题不显示消息内容；图8-18为IMPORTANCE_DEFAULT默认重要性时的通知栏，可见该通知正常显示消息标题和消息内容；图8-19为IMPORTANCE_HIGH高重要性时的顶部悬浮通知。



图 8-17　最小级别时的通知栏



图 8-18　默认重要性时的通知栏



图 8-19　高重要性时的顶部悬浮通知

8.3.3　推送服务到前台

服务没有自己的布局文件，意味着无法直接在页面上展示服务信息，要想了解服务的运行情况，要么通过打印日志观察，要么通过某个页面的静态控件显示运行结果。然而活动页面有自身的生命周期，极有可能发生服务尚在运行但页面早已退出的情况，所以该方式不可靠。为此Android设计了一个让服务在前台运行的机制，也就是在手机的通知栏展示服务的画像，同时允许服务控制自己是否需要在通知栏显示，这类控制操作包括下列两个启停方法：

- startForeground：**把当前服务切换到前台运行，即展示到通知栏。第一个参数表示通知的编号，第二个参数表示**Notification**对象。**
- stopForeground：**停止前台运行，即取消通知栏上的展示。参数为**true**时表示清除通知，参数为**false**时表示不清除通知。**

注　意

从 Android 9.0 开始，要想在服务中正常调用 startForeground 方法，还需修改 AndroidManifest.xml，添加如下所示的前台服务权限配置：

```
<!-- 允许前台服务（Android 9.0之后需要） -->
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

音乐播放器是前台服务的一个常见应用，即使用户离开了播放器页面，手机仍然在后台继续播放音乐，同时还能在通知栏查看播放进度。接下来模拟音乐播放器的前台服务功能。首先创建名为MusicService的音乐服务，该服务的通知推送代码示例如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\service\MusicService.java)
// 发送前台通知
private void sendNotify(Context ctx, String song, boolean isPlaying, int progress) {
    String message = String.format("歌曲%s", isPlaying?"正在播放":"暂停播放");
    // 创建一个跳转到活动页面的意图
    Intent intent = new Intent(ctx, MainActivity.class);
    // 创建一个用于页面跳转的延迟意图
    PendingIntent clickIntent = PendingIntent.getActivity(ctx,
            R.string.app_name, intent, PendingIntent.FLAG_UPDATE_CURRENT);
    // 创建一个通知消息的建造器
```

```
    Notification.Builder builder = new Notification.Builder(ctx);
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        // 从Android 8.0开始必须给每个通知分配对应的渠道
        builder = new Notification.Builder(ctx, getString(R.string.app_name));
    }
    builder.setContentIntent(clickIntent)          // 设置内容的点击意图
           .setSmallIcon(R.drawable.tt_s)          // 设置应用名称左边的小图标
           // 设置通知栏右边的大图标
           .setLargeIcon(BitmapFactory.decodeResource(getResources(), R.drawable.tt))
           .setProgress(100, progress, false)      // 设置进度条与当前进度
           .setContentTitle(song)                  // 设置通知栏里面的标题文本
           .setContentText(message);               // 设置通知栏里面的内容文本
    Notification notify = builder.build();         // 根据通知建造器构建一个通知对象
    startForeground(2, notify);                    // 把服务推送到前台的通知栏
}
```

接着通过活动页面的播放按钮控制音乐服务，不管是开始播放还是暂停播放都调用startService方法，区别在于传给服务的isPlaying参数不同（开始播放传true，暂停播放传false），再由音乐服务根据isPlaying来刷新消息通知。活动页面的播放控制代码如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\ForegroundServiceActivity.java)
// 创建一个通往音乐服务的意图
Intent intent = new Intent(this, MusicService.class);
intent.putExtra("is play", isPlaying);  // 是否正在播放音乐
intent.putExtra("song", et_song.getText().toString());
btn_send_service.setText(isPlaying?"暂停播放音乐":"开始播放音乐");
startService(intent);  // 启动音乐播放服务
```

运行测试App，先输入歌曲名称，活动页面如图8-20所示，点击“开始播放音乐”按钮，启动音乐服务并推送到前台，此时通知栏如图8-21所示。

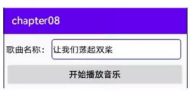


图 8-20　准备播放音乐



图 8-21　正在播放的通知栏

回到活动页面，如图8-22所示，点击“暂停播放音乐”按钮，音乐服务根据收到的isPlaying更新通知栏，此时通知栏如图8-23所示。

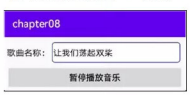


图 8-22　准备暂停音乐



图 8-23　暂停播放的通知栏

8.3.4　仿微信的悬浮通知

每个活动页面都是一个窗口，许多窗口对象需要一个管家来打理，这个管家被称作窗口管理器（WindowManager）。在手机屏幕上新增或删除页面窗口都可以归结为WindowManager的操作，下面是该管理类的常用方法：

- getDefaultDisplay：**获取默认的显示屏信息。通常可用该方法获取屏幕分辨率。**

- addView：往窗口中添加视图，第二个参数为WindowManager.LayoutParams对象。
- updateViewLayout：更新指定视图的布局参数，第二个参数为WindowManager. LayoutParams对象。
- removeView：从窗口中移除指定视图。

下面是窗口布局参数WindowManager.LayoutParams的常用属性：

- alpha：窗口的透明度，取值为0.0～1.0（0.0表示全透明，1.0表示不透明）。
- gravity：内部视图的对齐方式。取值说明同View类的setGravity方法。
- x和y：分别表示窗口左上角的横坐标和纵坐标。
- width和height：分别表示窗口的宽度和高度。
- format：窗口的像素点格式。取值见PixelFormat类中的常量定义，一般取值为PixelFormat.RGBA_8888。
- type：窗口的显示类型，常用的显示类型的取值说明见表8-7。

表 8-7　窗口显示类型的取值说明

WindowManager 类的窗口显示类型	说　明
TYPE_APPLICATION_OVERLAY	悬浮窗（覆盖于应用之上）
TYPE_SYSTEM_ALERT	系统警告提示，该类型从 Android 8.0 开始被废弃
TYPE_SYSTEM_ERROR	系统错误提示
TYPE_SYSTEM_OVERLAY	页面顶层提示
TYPE_SYSTEM_DIALOG	系统对话框
TYPE_STATUS_BAR	状态栏
TYPE_TOAST	短暂提示

- flags：窗口的行为准则，对于悬浮窗来说，一般设置为FLAG_NOT_FOCUSABLE。常用的窗口标志位的取值说明见表8-8。

表 8-8　窗口标志位的取值说明

WindowManager 类的窗口标志位	说　明
FLAG_NOT_FOCUSABLE	不能抢占焦点，即不接受任何按键或按钮事件
FLAG_NOT_TOUCHABLE	不接受触摸屏事件。悬浮窗一般不设置该标志，因为一旦设置该标志就将无法拖动
FLAG_NOT_TOUCH_MODAL	当窗口允许获得焦点时（没有设置 FLAG_NOT_FOCUSALBE 标志），仍然将窗口之外的按键事件发送给后面的窗口处理，否则它将独占所有的按键事件，而不管它们是不是发生在窗口范围之内
FLAG_LAYOUT_IN_SCREEN	允许窗口占满整个屏幕
FLAG_LAYOUT_NO_LIMITS	允许窗口扩展到屏幕之外
FLAG_WATCH_OUTSIDE_TOUCH	设置了 FLAG_NOT_TOUCH_MODAL 标志后，当按键动作发生在窗口之外时，将接收到一个 MotionEvent.ACTION_OUTSIDE 事件

自定义的悬浮窗有点类似于对话框，它们都是独立于活动页面的窗口，但是悬浮窗又有一些与众不同的特性，例如：

（1）悬浮窗允许拖动，对话框不允许拖动。

（2）悬浮窗不妨碍用户触摸窗外的区域，对话框不让用户操作窗外的控件。

（3）悬浮窗独立于活动页面，当页面退出后，悬浮窗仍停留在屏幕上；对话框与活动页面是共存关系，一旦退出页面那么对话框就消失了。

基于悬浮窗的以上特性，若要实现窗口的悬浮效果，就不能仅仅调用WindowManager的addView方法，而要做一系列的自定义处理，具体步骤说明如下：

01 在AndroidManifest.xml中声明系统窗口权限，即增加下面这行权限配置：

```
<!-- 悬浮窗 -->
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />
```

02 自定义的悬浮窗控件需要设置触摸监听器，根据用户的手势动作相应调整窗口位置，以实现悬浮窗的拖动功能。

03 合理设置悬浮窗的窗口参数，主要是把窗口参数的显示类型设置为TYPE_APPLICATION_OVERLAY。另外，还要设置标志位为FLAG_NOT_FOCUSABLE。

04 在构造悬浮窗实例时，要传入应用实例Application的上下文对象，这是为了保证即使退出活动页面，也不会关闭悬浮窗。因为应用对象在App运行过程中始终存在，而活动对象只在打开页面时有效；一旦退出页面，那么活动对象的上下文就会立刻被回收（这导致依赖于该上下文的悬浮窗也一块被回收了）。

下面是一个悬浮窗控件的自定义代码片段：

```
(完整代码见video\src\main\java\com\example\video\widget\FloatWindow.java)
private Context mContext;                                   // 声明一个上下文对象
private WindowManager wm;                                   // 声明一个窗口管理器对象
private static WindowManager.LayoutParams wmParams;  // 悬浮窗的布局参数
public View mContentView;                                   // 声明一个内容视图对象
private float mScreenX, mScreenY;                           // 触摸点在屏幕上的横纵坐标
private float mLastX, mLastY;                               // 上次触摸点的横纵坐标
private float mDownX, mDownY;                               // 按下点的横纵坐标
private boolean isShowing = false;                          // 是否正在显示

public FloatWindow(Context context) {
    super(context);
    // 从系统服务中获取窗口管理器，后续将通过该管理器添加悬浮窗
    wm = (WindowManager)
                context.getSystemService(Context.WINDOW_SERVICE);
    if (wmParams == null) {
        wmParams = new WindowManager.LayoutParams();
    }
    mContext = context;
}

// 设置悬浮窗的内容布局
public void setLayout(int layoutId) {
    // 从指定资源编号的布局文件中获取内容视图对象
    mContentView = LayoutInflater.from(mContext).inflate(layoutId, null);
    // 接管悬浮窗的触摸事件，使之既可随手势拖动又可处理点击动作
    mContentView.setOnTouchListener((v, event) -> {
        mScreenX = event.getRawX();
        mScreenY = event.getRawY();
```

```
        if (event.getAction() == MotionEvent.ACTION_DOWN) {         // 手指按下
            mDownX = mScreenX;
            mDownY = mScreenY;
        } else if (event.getAction() == MotionEvent.ACTION_MOVE) { // 移动
            updateViewPosition();                                   // 更新视图的位置
        } else if (event.getAction() == MotionEvent.ACTION_UP) {    // 手指松开
            updateViewPosition();                                   // 更新视图的位置
            if (Math.abs(mScreenX-mDownX)<3 && Math.abs(mScreenY-mDownY)<3) {
                if (mListener != null) {                        // 响应悬浮窗的点击事件
                    mListener.onFloatClick(v);
                }
            }
        }
        mLastX = mScreenX;
        mLastY = mScreenY;
        return true;
    });
}

// 更新悬浮窗的视图位置
private void updateViewPosition() {
    // 此处不能直接转为整数类型，因为小数部分会被截掉，重复多次后就会造成偏移越来越大
    wmParams.x = Math.round(wmParams.x + mScreenX - mLastX);
    wmParams.y = Math.round(wmParams.y + mScreenY - mLastY);
    wm.updateViewLayout(mContentView, wmParams);              // 更新内容视图的布局参数
}

// 显示悬浮窗
public void show(int gravity) {
    if (mContentView != null) {
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.O) {
            // 注意TYPE_SYSTEM_ALERT从Android 8.0开始被舍弃了
            wmParams.type = WindowManager.LayoutParams.TYPE_SYSTEM_ALERT;
        } else {  // 从Android 8.0开始悬浮窗要使用TYPE_APPLICATION_OVERLAY
            wmParams.type =
                    WindowManager.LayoutParams.TYPE APPLICATION OVERLAY;
        }
        wmParams.format = PixelFormat.RGBA_8888;
        wmParams.flags = WindowManager.LayoutParams.FLAG NOT FOCUSABLE;
        wmParams.alpha = 1.0f;            // 1.0为完全不透明，0.0为完全透明
        wmParams.gravity = gravity;       // 指定悬浮窗的对齐方式
        wmParams.x = 0;
        wmParams.y = 0;
        // 设置悬浮窗的宽度和高度为自适应
        wmParams.width = WindowManager.LayoutParams.WRAP_CONTENT;
        wmParams.height = WindowManager.LayoutParams.WRAP_CONTENT;
        // 添加自定义的窗口布局，然后在屏幕上就能看到悬浮窗了
        wm.addView(mContentView, wmParams);
        isShowing = true;
    }
}

// 关闭悬浮窗
public void close() {
    if (mContentView != null) {
        wm.removeView(mContentView);          // 移除自定义的窗口布局
        isShowing = false;
```

```
        }
    }
    private FloatClickListener mListener;       // 声明一个悬浮窗的点击监听器对象
    // 设置悬浮窗的点击监听器
    public void setOnFloatListener(FloatClickListener listener) {
        mListener = listener;
    }
    // 定义一个悬浮窗的点击监听器接口，用于触发点击动作
    public interface FloatClickListener {
        void onFloatClick(View v);
    }
```

有了悬浮窗以后，就能很方便地在手机屏幕上弹出动态小窗，例如时钟、天气、实时流量、股市指数等。还有微信的新消息通知，每当好友发了一条新消息，屏幕顶部便弹出微信的悬浮通知栏，点击这个悬浮栏会打开好友的聊天界面。这些类似的功能，都能通过悬浮窗控件实现。

悬浮窗的常见操作有打开、关闭和点击三种，前面定义的悬浮窗控件正好提供了对应的方法，比如调用show方法可以显示悬浮窗，调用close方法可以关闭悬浮窗，调用setOnFloatListener可以设置悬浮窗的点击监听器。下面是在活动页面操作悬浮窗的代码例子：

```
    (完整代码见chapter08\src\main\java\com\example\chapter08\FloatNoticeActivity.java)
    private static FloatWindow mFloatWindow;        // 声明一个悬浮窗对象
    // 打开悬浮窗
    private void openFloatWindow() {
        if (mFloatWindow == null) {
            // 创建一个新的悬浮窗
            mFloatWindow = new FloatWindow(MainApplication.getInstance());
            // 设置悬浮窗的布局内容
            mFloatWindow.setLayout(R.layout.float_notice);
            tv_content = mFloatWindow.mContentView.findViewById(R.id.tv_content);
            LinearLayout ll_float = mFloatWindow.mContentView.findViewById(R.id.ll_float);
            int margin = Utils.dip2px(this, 5);
            LinearLayout.LayoutParams params = (LinearLayout.LayoutParams)
ll_float.getLayoutParams();
            params.width = Utils.getScreenWidth(this) - 2*margin;
            // 在悬浮窗四周留白
            params.setMargins(margin, margin, margin, margin);
            ll_float.setLayoutParams(params);
            // 设置悬浮窗的点击监听器
            mFloatWindow.setOnFloatListener(v -> mFloatWindow.close());
        }
        if (mFloatWindow != null && !mFloatWindow.isShow()) {
            tv_content.setText(et_content.getText());
            mFloatWindow.show(Gravity.LEFT | Gravity.TOP);     // 显示悬浮窗
        }
    }
    // 关闭悬浮窗
    private void closeFloatWindow() {
        if (mFloatWindow != null && mFloatWindow.isShow()) {
            mFloatWindow.close();                              // 关闭悬浮窗
        }
    }
```

运行测试App，观察到仿微信悬浮窗的展示效果如图8-24和图8-25所示。其中图8-24为准备弹出悬浮窗的编辑页面，图8-25为已经弹出悬浮窗的屏幕画面。

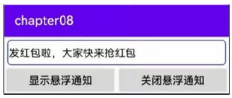


图 8-24　准备弹出悬浮窗的编辑页面

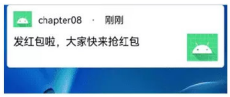


图 8-25　已经弹出悬浮窗的屏幕画面

若想实时弹出悬浮窗，需要通过服务Service来实现，此时要改造成在服务中创建并显示悬浮窗，有兴趣的读者不妨一试。

8.4　通过持续绘制实现简单动画

本节介绍如何通过持续绘制实现动画效果：首先阐述Handler的延迟机制以及简单计时器的实现，然后描述刷新视图的两种方式以及它们之间的区别，最后叙述如何结合Handler的延迟机制与视图刷新办法实现饼图动画。

8.4.1　Handler 的延迟机制

活动页面的Java代码通常是串行工作的，而且App界面很快就加载完成容不得半点迟延，不过偶尔也需要某些控件时不时地动一下，好让界面呈现动画效果显得更加活泼。这种简单动画基于视图的延迟处理机制，即间隔若干时间后刷新视图界面。在第2章的“2.4.3　跳转到另一个页面”中，当时为了演示3秒后自动跳转到另一个活动页面，便用到了Handler+Runnable组合，调用Handler对象的postDelayed方法，延迟若干时间再执行指定的Runnable任务。

Runnable接口用于声明某项任务，它定义了接下来要做的事情。简单地说，Runnable接口就是一个代码片段。编写任务代码需要实现Runnable接口，此时必须重写接口的run方法，在该方法内部存放待运行的代码逻辑。run方法无须显式调用，因为在启动Runnable实例时就会调用任务对象的run方法。

尽管视图基类View同样提供了post与postDelayed方法，但在实际开发中一般利用处理器Handler启动任务实例。Handler操作任务的常见方法说明如下：

- post：**立即启动指定的任务。参数为Runnable对象。**
- postDelayed：**延迟若干时间后启动指定的任务。第一个参数为Runnable对象；第二个参数为延迟的时间间隔，单位为毫秒。**
- postAtTime：**在设定的时间点启动指定的任务。第一个参数为Runnable对象；第二个参数为任务的启动时间点，单位为毫秒。**
- removeCallbacks：**移除指定的任务。参数为Runnable对象。**

计时器是Handler+Runnable组合的简单应用，每隔若干时间就刷新当前的计数值，使得界面上的数字持续跳跃。下面是一个简单计时器的活动代码例子：

```
    (完整代码见chapter08\src\main\java\com\example\chapter08\HandlerPostActivity.java)
    public class HandlerPostActivity extends AppCompatActivity implements
View.OnClickListener {
        private Button btn_count;                          // 声明一个按钮对象
        private TextView tv_result;                        // 声明一个文本视图对象

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_handler_post);
            btn_count = findViewById(R.id.btn_count);
            tv_result = findViewById(R.id.tv_result);
            btn_count.setOnClickListener(this);            // 设置按钮的点击监听器
        }

        @Override
        public void onClick(View v) {
            if (v.getId() == R.id.btn_count) {
                if (!isStarted) {                          // 不在计数，则开始计数
                    btn_count.setText("停止计数");
                    mHandler.post(mCounter);               // 立即启动计数任务
                } else {                                   // 已在计数，则停止计数
                    btn_count.setText("开始计数");
                    mHandler.removeCallbacks(mCounter);    // 立即取消计数任务
                }
                isStarted = !isStarted;
            }
        }

        private boolean isStarted = false;                 // 是否开始计数
        private Handler mHandler = new Handler();          // 声明一个处理器对象
        private int mCount = 0;                            // 计数值
        // 定义一个计数任务
        private Runnable mCounter = new Runnable() {
            @Override
            public void run() {
                mCount++;
                tv_result.setText("当前计数值为：" + mCount);
                mHandler.postDelayed(this, 1000);          // 延迟一秒后重复计数任务
            }
        };
    }
```

运行测试App，观察到计时器的计数效果如图8-26和图8-27所示。其中，图8-26表示当前正在计数；图8-27表示当前停止计数，终止的计数值为15。



图 8-26　计时器开始计数



图 8-27　计时器结束计数

8.4.2　重新绘制视图界面

控件的内容一旦发生变化，就得通知界面刷新它的外观，例如文本视图修改了文字，图像视图更换了图片等。然而，之前只听说TextView提供了setText方法，ImageView提供了setImageBitmap方法，这两个方法调用之后便能直接呈现最新的控件界面，好像并不需要刷新动作。虽然表面上看不出刷新操作，但仔细分析setText与setImageBitmap的源码，会发现它们内部都调用了invalidate方

法，该方法便用来刷新控件界面。只要调用了invalidate方法，系统就会重新执行该控件的onDraw方法和dispatchDraw方法，从而实现重新绘制界面，也就是界面刷新的功能。

除了invalidate方法，另一种postInvalidate方法也能刷新界面，它们之间的区别主要有下列两点：

（1）invalidate不是线程安全的，它只保证在主线程（UI线程）中能够正常刷新视图；而postInvalidate是线程安全的，即使在分线程中调用也能正常刷新视图。

（2）invalidate只能立即刷新视图，而post方式还提供了postInvalidateDelayed方法，允许延迟一段时间后再刷新视图。

为了演示invalidate、postInvalidate、postInvalidateDelayed这3种用法，并验证分线程内部的视图刷新情况，下面先定义一个椭圆视图OvalView，每次刷新该视图都将绘制更大角度的扇形。椭圆视图的定义代码示例如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\widget\OvalView.java)
public class OvalView extends View {
    private Paint mPaint = new Paint();                     // 创建一个画笔对象
    private int mDrawingAngle = 0;                          // 当前绘制的角度

    public OvalView(Context context) {
        this(context, null);
    }

    public OvalView(Context context, AttributeSet attrs) {
        super(context, attrs);
        mPaint.setColor(Color.RED);                         // 设置画笔的颜色
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        mDrawingAngle += 30;                                // 绘制角度增加30°
        int width = getMeasuredWidth();                     // 获得布局的实际宽度
        int height = getMeasuredHeight();                   // 获得布局的实际高度
        RectF rectf = new RectF(0, 0, width, height);       // 创建扇形的矩形边界
        // 在画布上绘制指定角度的扇形。第4个参数为true表示绘制扇形，为false表示绘制圆弧
        canvas.drawArc(rectf, 0, mDrawingAngle, true, mPaint);
    }
}
```

接着在演示用的布局文件中加入自定义的椭圆视图节点，具体的OvalView标签代码如下：

```
(完整代码见chapter08\src\main\res\layout\activity view invalidate.xml)
<!-- 自定义的椭圆视图，需要使用全路径 -->
<com.example.chapter08.widget.OvalView
    android:id="@+id/ov_validate"
    android:layout_width="match_parent"
    android:layout_height="150dp" />
```

然后在对应的活动代码中依据不同的选项，分别调用invalidate、postInvalidate、postInvalidateDelayed三个方法之一，加上分线程内部的两个方法调用，总共五种刷新选项。下面是这五种选项的方法调用代码片段：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\ViewInvalidateActivity.java)
public void onItemSelected(AdapterView<?> arg0, View arg1, int arg2, long arg3) {
    if (arg2 == 0) {                                    // 主线程调用invalidate
        ov_validate.invalidate();                       // 刷新视图（用于主线程）
    } else if (arg2 == 1) {                             // 主线程调用postInvalidate
        ov_validate.postInvalidate();                   // 刷新视图（主线程和分线程均可使用）
```

```
        } else if (arg2 == 2) {                          // 延迟3秒后刷新
            ov validate.postInvalidateDelayed(3000);     // 延迟若干时间后再刷新视图
        } else if (arg2 == 3) {                          // 分线程调用invalidate
            // invalidate不是线程安全的，在分线程中调用invalidate在复杂场合可能出错
            new Thread(new Runnable() {
                @Override
                public void run() {
                    ov validate.invalidate();            // 刷新视图（用于主线程）
                }
            }).start();
        } else if (arg2 == 4) {                          // 分线程调用postInvalidate
            // postInvalidate是线程安全的，在分线程中建议调用postInvalidate方法来刷新视图
            new Thread(new Runnable() {
                @Override
                public void run() {
                    ov_validate.postInvalidate();        // 刷新视图（主线程和分线程均可使用）
                }
            }).start();
        }
    }
```

运行测试App，分别选择上述选项中的4种，视图刷新后的界面效果如图8-28～图8-31所示。其中图8-28为主线程调用invalidate的刷新结果，图8-29为分线程调用invalidate的刷新结果，图8-30为主线程调用postInvalidate的刷新结果，图8-31为分线程调用postInvalidate的刷新结果。



图 8-28　主线程调用 invalidate 的刷新结果



图 8-29　分线程调用 invalidate 的刷新结果



图 8-30　主线程调用 postInvalidate 的刷新结果



图 8-31　分线程调用 postInvalidate 的刷新结果

观察发现，不管是在主线程中调用刷新方法，还是在分线程中调用刷新方法，界面都能正常显示角度渐增的椭圆视图。从实验结果可知，尽管invalidate不是线程安全的方法，但它仍然能够在简单的分线程中刷新视图。不过考虑到实际的业务场景较复杂，建议还是遵循安卓的开发规范，在主线程中使用invalidate方法刷新视图，在分线程中使用postInvalidate方法刷新视图。

8.4.3　自定义饼图动画

掌握了Handler的延迟机制，加上视图对象的刷新方法，就能间隔固定时间不断渲染控件界面，

从而实现简单的动画效果。接下来通过饼图动画的实现过程，进一步加深对自定义控件技术的熟练运用。自定义饼图动画的具体实现步骤说明如下：

01 在Java代码的widget目录下创建PieAnimation.java，该类继承了视图基类View，并重写onDraw方法，在onDraw方法中使用画笔对象绘制指定角度的扇形。

02 在PieAnimation内部定义一个视图刷新任务，每次刷新操作都增大一点绘图角度，然后调用invalidate方法刷新视图界面。如果动画尚未播放完毕，就调用处理器对象的postDelayed方法，间隔几十毫秒后重新执行刷新任务。

03 给PieAnimation补充一个start方法，用于控制饼图动画的播放操作。start方法内部先初始化绘图角度，再调用处理器对象的post方法立即启动刷新任务。

按照上述3个步骤，编写自定义的饼图动画控件代码示例如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\widget\PieAnimation.java)
public class PieAnimation extends View {
    private Paint mPaint = new Paint();             // 创建一个画笔对象
    private int mDrawingAngle = 0;                  // 当前绘制的角度
    private Handler mHandler = new Handler();       // 声明一个处理器对象
    private boolean isRunning = false;              // 是否正在播放动画

    public PieAnimation(Context context) {
        this(context, null);
    }

    public PieAnimation(Context context, AttributeSet attrs) {
        super(context, attrs);
        mPaint.setColor(Color.GREEN);               // 设置画笔的颜色
    }

    // 开始播放动画
    public void start() {
        mDrawingAngle = 0;                          // 绘制角度清零
        isRunning = true;
        mHandler.post(mRefresh);                    // 立即启动绘图刷新任务
    }

    // 定义一个绘图刷新任务
    private Runnable mRefresh = new Runnable() {
        @Override
        public void run() {
            mDrawingAngle += 3;                     // 每次绘制时角度增加3°
            if (mDrawingAngle <= 270) {             // 未绘制完成，最大绘制到270°
                invalidate();  // 立即刷新视图
                mHandler.postDelayed(this, 70);     // 延迟若干时间后再次启动刷新任务
            } else {                                // 已绘制完成
                isRunning = false;
            }
        }
    };

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        if (isRunning) {                                    // 正在播放饼图动画
            int width = getMeasuredWidth();                 // 获得已测量的宽度
            int height = getMeasuredHeight();               // 获得已测量的高度
            // 视图的宽和高取较小的那个作为扇形的直径
            int diameter = Math.min(width, height);
            // 创建扇形的矩形边界
```

```
            RectF rectf = new RectF((width - diameter)/2,(height - diameter)/2,
                    (width + diameter) / 2, (height + diameter) / 2);
            // 在画布上绘制指定角度的图形。第4个参数为true绘制扇形，为false绘制圆弧
            canvas.drawArc(rectf, 0, mDrawingAngle, true, mPaint);
        }
    }
}
```

接着创建演示用的活动页面，在该页面的XML文件中放置新控件PieAnimation，完整的XML文件内容示例如下：

```
(完整代码见chapter08\src\main\res\layout\activity_pie_animation.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout height="match parent"
    android:orientation="vertical">
    <!-- 自定义的饼图动画，需要使用全路径 -->
    <com.example.chapter08.widget.PieAnimation
        android:id="@+id/pa circle"
        android:layout width="match parent"
        android:layout height="350dp" />
</LinearLayout>
```

然后在该页面的Java代码中获取饼图控件，并调用饼图对象的start方法开始播放动画，相应的活动代码如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\PieAnimationActivity.java)
public class PieAnimationActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity pie animation);
        // 从布局文件中获取名为pa circle的饼图动画
        PieAnimation pa_circle = findViewById(R.id.pa_circle);
        pa_circle.start();  // 开始播放饼图动画
    }
}
```

最后运行测试App。观察到饼图动画的播放效果如图8-32和图8-33所示。其中，图8-32为饼图动画开始播放不久的画面，图8-33为饼图动画播放结束时的画面。

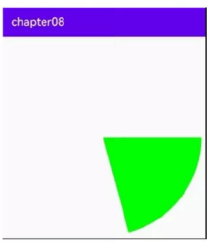


图 8-32　饼图动画播放开始

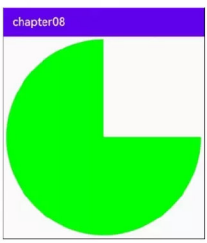


图 8-33　饼图动画播放结束

8.5　实战项目：广告轮播

电商App的首页上方，都在明显位置放了一栏广告条，并且广告条会自动轮播，着实吸引眼球。这种广告轮播的功能为推广热门事物出力甚大，早已成为电商App、视频App、新闻App的标配。本节就来探讨如何实现类似的广告轮播效果。

8.5.1　需求描述

作为App首页的常客，广告轮播特效早就为人所熟知，它的界面也司空见惯，比如图8-34～图8-37便是一组简单的轮播广告。其中图8-34为展示第一幅广告的效果，图8-35为轮播第二幅广告的效果，图8-36为轮播第四幅广告的效果，图8-37为轮播第五幅广告的效果。

由图8-34～图8-37可知，广告条除了广告图片之外，底部还有一排圆点，这些圆点被称作指示器，每当轮播到第几副广告，指示器就高亮显示第几个圆点，其余圆点显示白色，如此一来，用户便知晓当前播放到了第几个广告。

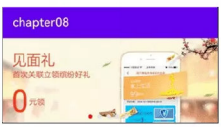


图 8-34　展示第一幅广告



图 8-35　轮播第二幅广告



图 8-36　轮播第四幅广告

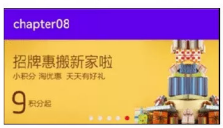


图 8-37　轮播第五幅广告

正常情况下，间隔固定时间（2～3秒），广告条从左往右依次轮播，一直轮播到最后一幅广告，其右边再无别的广告，此时广告条自动切回左边第一幅广告，从头开始新一轮的广告播放，这样才具备了广告轮播条的基础功能。

8.5.2　界面设计

依据广告轮播条的界面效果，轮播功能应当引入下列几个控件：

- **相对布局**RelativeLayout：**指示器圆点位于广告条的底部中央，这种靠下且居中的相对位置用到了相对布局。**
- **单选组**RadioGroup：**指示器同一时刻只有一个圆点高亮显示，符合单选组内部只能选择唯一单选按钮的规则。**

- **翻页视图ViewPager：广告轮播采取翻页方式，自然用到了翻页视图。**
- **翻页适配器PagerAdapter：翻页视图需要搭配翻页适配器，对于广告来说，就是基于PagerAdapter实现一个展示图像的翻页适配器。**

此外，广告条每隔两三秒就轮播下一幅广告，这种自动轮播可采用Handler+Runnable的延迟机制，即先定义一个广告滚动任务，再由Handler间隔固定时间启动广告滚动任务。

考虑到广告条并非只在App首页出现，App的其他页面也有可能使用广告轮播功能，因此有必要将广告条封装为单独的控件，以便随时随地在各页面中添加，这就需要运用自定义控件的相关技术，把广告条做成通用的自定义控件。

8.5.3 关键代码

自定义广告条感觉不那么容易实现，接下来详细阐述广告轮播的具体实现步骤，方便读者更好地理解和运用自定义控件技术。具体步骤说明如下：

1. 定义广告条的XML布局文件

首先通过XML文件描述广告条的界面布局，其外层采用相对布局，内部嵌套容纳广告图片的翻页视图，以及用作指示器的单选组，其中单选组的位置相对上级布局靠下且居中。详细的广告条布局示例如下：

```
(完整代码见chapter08\src\main\res\layout\banner_pager.xml)
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <androidx.viewpager.widget.ViewPager
        android:id="@+id/vp_banner"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <RadioGroup
        android:id="@+id/rg_indicator"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingBottom="2dp"
        android:orientation="horizontal"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true" />
</RelativeLayout>
```

2. 编写广告条的Java定义代码

接着编写广告条的自定义控件代码，该控件由RelativeLayout派生而来，通过LayoutInflater工具从第一步定义的布局文件banner_pager.xml中获取视图对象，并调用addView方法将视图对象添加至当前控件。简要的自定义控件代码示例如下：

```
(完整代码见chapter08\src\main\java\com\example\chapter08\widget\BannerPager.java)
public class BannerPager extends RelativeLayout {
    private Context mContext;                       // 声明一个上下文对象
    private ViewPager vp_banner;                    // 声明一个翻页视图对象
    private RadioGroup rg_indicator;                // 声明一个单选组对象

    public BannerPager(Context context) {
        this(context, null);
    }
```

```
    public BannerPager(Context context, AttributeSet attrs) {
        super(context, attrs);
        mContext = context;
        initView();                    // 初始化视图
    }
    // 初始化视图
    private void initView() {
        // 根据布局文件banner pager.xml生成视图对象
        View view = LayoutInflater.from(mContext).inflate(R.layout.banner_pager, null);
        // 从布局文件中获取名为vp_banner的翻页视图
        vp_banner = view.findViewById(R.id.vp_banner);
        // 从布局文件中获取名为rg indicator的单选组
        rg_indicator = view.findViewById(R.id.rg_indicator);
        addView(view);                 // 将该布局视图添加到广告轮播条
    }
}
```

3. 为广告条添加图片清单

第二步的自定义控件代码，仅仅获得广告条的布局框架，尚未指定广告图片的来源及数量。为此需要给BannerPager类添加setImage方法，通过该方法传入准备轮播的图片清单。同时，根据轮播图片的数量，也给单选组添加同等数量的单选按钮，以便指示器标记当前轮播的图片序号。setImage方法的实现代码示例如下：

```
// 声明一个图像视图列表
private List<ImageView> mViewList = new ArrayList<ImageView>();
// 设置广告图片列表
public void setImage(List<Integer> imageList) {
    int dip_15 = Utils.dip2px(mContext, 15);
    // 根据图片列表生成图像视图列表
    for (int i = 0; i < imageList.size(); i++) {
        Integer imageResId = imageList.get(i);          // 获取图片的资源编号
        ImageView iv = new ImageView(mContext);         // 创建一个图像视图对象
        iv.setLayoutParams(new LayoutParams(
                LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT));
        iv.setScaleType(ImageView.ScaleType.FIT_XY);
        iv.setImageResource(imageResId);                // 设置图像视图的资源图片
        mViewList.add(iv);                              // 往视图列表中添加新的图像视图
    }
    // 设置翻页视图的图像适配器
    vp_banner.setAdapter(new ImageAdapater());
    // 给翻页视图添加简单的页面变更监听器，此时只需重写onPageSelected方法
    vp_banner.addOnPageChangeListener(new SimpleOnPageChangeListener() {
        @Override
        public void onPageSelected(int position) {
            setSelectedButton(position);                // 高亮显示该位置的指示按钮
        }
    });
    // 根据图片列表生成指示按钮列表
    for (int i = 0; i < imageList.size(); i++) {
        RadioButton radio = new RadioButton(mContext);  // 创建一个单选按钮对象
        radio.setLayoutParams(new RadioGroup.LayoutParams(dip_15, dip_15));
        // 设置单选按钮的资源图片
        radio.setButtonDrawable(R.drawable.indicator_selector);
        rg_indicator.addView(radio);                    // 往单选组中添加新的单选按钮
    }
    vp_banner.setCurrentItem(0);                        // 设置翻页视图显示第一页
    setSelectedButton(0);                               // 默认高亮显示第一个指示按钮
```

```
    }
    // 设置选中单选组内部的哪个单选按钮
    private void setSelectedButton(int position) {
        ((RadioButton) rg_indicator.getChildAt(position)).setChecked(true);
    }
```

4. 实现广告条的自动轮播功能

给广告条添加图片清单之后，还得设置具体的轮播规则，比如轮播的方向、轮播的间隔、是否重复轮播等，这些轮播规则都在广告滚动任务中制定。另外，再给BannerPager类提供一个start方法，外部调用该方法即可启动滚动任务。广告滚动任务以及启动方法的代码例子如下：

```
    private int mInterval = 2000;                           // 轮播的时间间隔，单位为毫秒
    // 开始广告轮播
    public void start() {
        mHandler.postDelayed(mScroll, mInterval);           // 延迟若干秒后启动滚动任务
    }
    private Handler mHandler = new Handler();               // 声明一个处理器对象
    // 定义一个广告滚动任务
    private Runnable mScroll = new Runnable() {
        @Override
        public void run() {
            int index = vp_banner.getCurrentItem() + 1;     // 获得下一张广告图的位置
            if (index >= mViewList.size()) {                // 已经到末尾了，准备重头开始
                index = 0;
            }
            vp_banner.setCurrentItem(index);                // 设置翻页视图显示第几页
            mHandler.postDelayed(this, mInterval);          // 延迟若干秒后继续启动滚动任务
        }
    };
```

5. 在活动页面中使用广告条控件

前面4步总算定义好了新控件——广告条BannerPager，打开活动页面的XML文件，添加新节点BannerPager，注意自定义控件需要使用全路径。XML文件内容如下：

```
    (完整代码见chapter08\src\main\res\layout\activity_banner_pager.xml)
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <!-- 自定义的广告轮播条，需要使用全路径 -->
        <com.example.chapter08.widget.BannerPager
            android:id="@+id/banner_pager"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
```

回到活动页面的Java代码，取出XML文件声明的广告条对象，先调用该对象的setImage方法添加图片清单，再调用start方法启动轮播操作。包含广告条的活动代码例子如下：

```
    (完整代码见chapter08\src\main\java\com\example\chapter08\NoscrollListActivity.java)
    public class BannerPagerActivity extends AppCompatActivity {
        private List<Integer> getImageList() {
            ArrayList<Integer> imageList = new ArrayList<Integer>();
            imageList.add(R.drawable.banner_1);
            imageList.add(R.drawable.banner_2);
            imageList.add(R.drawable.banner_3);
```

```
        imageList.add(R.drawable.banner_4);
        imageList.add(R.drawable.banner_5);
        return imageList;  // 返回默认的广告图片列表
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_banner_pager);
        tv_pager = findViewById(R.id.tv_pager);
        // 从布局文件中获取名为banner_pager的广告轮播条
        BannerPager banner = findViewById(R.id.banner_pager);
        // 获取广告轮播条的布局参数
        LayoutParams params = (LayoutParams) banner.getLayoutParams();
        params.height = (int) (Utils.getScreenWidth(this) * 250f / 640f);
        banner.setLayoutParams(params);           // 设置广告轮播条的布局参数
        banner.setImage(getImageList());          // 设置广告轮播条的广告图片列表
        banner.start();                           // 开始广告图片的轮播滚动
    }
}
```

8.6 小　　结

本章主要介绍了App开发的自定义控件相关知识，包括：视图的构建过程（视图的构造方法、视图的测量方法、视图的绘制方法）、改造已有的控件（自定义月份选择器、给翻页标签栏添加新属性、不滚动的列表视图）、推送消息通知（通知推送Notification、通知渠道NotificationChannel、推送服务到前台、仿微信的悬浮通知）、通过持续绘制实现简单动画（Handler的延迟机制、重新绘制视图界面、自定义饼图动画）。最后设计了一个实战项目“广告轮播”，在该项目的App编码中采用了本章介绍的大部分自定义控件技术，从而加深了对所学知识的理解。

通过本章的学习，读者应该能够掌握以下4种开发技能。

（1）学会实现视图的构造、测量和绘图方法。

（2）学会通过重写某个方法改造已有的控件。

（3）了解通知的应用场景，并学会正确推送消息通知。

（4）学会结合Handler的延迟机制与视图刷新办法实现简单动画。

8.7 动 手 练 习

请上机实验本章的广告轮播项目，注意运用自定义控件技术并实现简单动画。

第 9 章

组 合 控 件

本章介绍App开发常用的一些组合控件用法，主要包括：如何实现底部标签栏、如何运用顶部导航栏、如何利用循环视图实现3种增强型列表、如何使用二代翻页视图实现更炫的翻页效果。然后结合本章所学的知识，演示一个实战项目“电商首页”的设计与实现。

9.1 底部标签栏

本节介绍底部标签栏的两种实现方式：首先说明如何通过Android Studio菜单自动创建基于BottomNavigationView的导航活动，然后描述如何利用状态图形与风格样式实现自定义的标签按钮，最后阐述怎样结合RadioGroup和ViewPager制作自定义的底部标签栏。

9.1.1 利用 BottomNavigationView 实现底部标签栏

不管是微信还是 QQ，淘宝抑或京东，它们的首屏都在底部展开一排标签，每个标签对应着一个频道，从而方便用户迅速切换到对应频道。这种底部标签栏原本是苹果手机的标配，原生安卓最开始并不提供屏幕底部的快捷方式，倒是众多国产 App 纷纷山寨苹果的风格，八仙过海各显神通整出了底部标签栏。后来谷歌一看这种风格还颇受用户欢迎，于是顺势在 Android Studio 中集成了该风格的快捷标签。

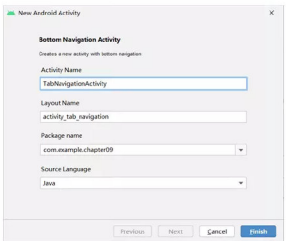


图 9-1 导航活动的创建对话框

如今在Android Studio上创建官方默认的首屏标签页面已经很方便了，具体步骤如下：

01 首先右击需要添加标签栏的模块，在弹出的右键菜单中依次选择 New → Activity → Bottom Navigation Activity，弹出如图9-1所示的活动创建对话框。

02 在创建对话框的Activity Name一栏填写新活动的名称，再单击对话框右下角的Finish按钮，Android Studio就会自动创建该活动的Java代码及其XML文件。

03 编译运行App，进入刚创建的活动页面，其界面效果如图9-2所示。可见测试页面的底部默认提供了3个导航标签，分别是Home、Dashboard和Notifications。

注意到初始页面的Home标签从文字到图片均为高亮显示，说明当前处于Home频道。接着点击Dashboard标签，此时界面如图9-3所示，可见切换到了Dashboard频道。继续点击Notifications标签，此时界面如图9-4所示，可见切换到了Notifications频道。至此不费丝毫功夫，Android Studio已然实现了简单的标签导航功能。



图 9-2 导航活动的默认界面



图 9-3 点击了 Dashboard 标签

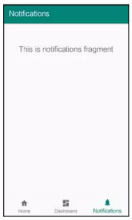


图 9-4 点击了 Notifications 标签

不过为了定制页面的详细内容，开发者仍需修改相关代码，譬如将标签文字从英文改成中文，将频道上方的描述说明从英文改成中文，给频道页面添加图像视图等其他控件，等等，故而还得梳理标签栏框架的实现方式。

首先打开模块的build.gradle，在dependencies节点内部发现多了下面两行依赖库配置，表示引用了标签导航的navigation库：

```
implementation 'androidx.navigation:navigation-fragment:2.4.1'
implementation 'androidx.navigation:navigation-ui:2.4.1'
```

再来查看标签页面的XML文件，它的关键内容如下：

```
(完整代码见chapter09\src\main\res\layout\activity_tab_navigation.xml)
<com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/nav view"
    android:layout width="0dp"
    android:layout_height="wrap_content"
    android:background="?android:attr/windowBackground"
    app:layout constraintBottom toBottomOf="parent"
    app:menu="@menu/bottom nav menu" />
<fragment
    android:id="@+id/nav host fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout width="match parent"
    android:layout height="match parent"
    app:defaultNavHost="true"
    app:layout_constraintTop_toTopOf="parent"
    app:navGraph="@navigation/mobile_navigation" />
```

从上面的布局内容可知，标签页面主要包含两个组成部分：一个是位于底部的BottomNavigationView（底部导航视图），另一个是位于其上占据剩余屏幕的碎片fragment。底部导航视图又由一排标签菜单组成，具体菜单在@menu/bottom_nav_menu中定义；而碎片为各频道的主体部分，具体内容在app:navGraph="@navigation/mobile_navigation中定义。哟，原来奥妙就在这两个文件当中，赶紧打开menu目录之下的bottom_nav_menu.xml看看：

```
(完整代码见chapter09\src\main\res\menu\bottom_nav_menu.xml)
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/navigation home"
        android:icon="@drawable/ic_home_black_24dp"
        android:title="@string/title_home" />
    <item
        android:id="@+id/navigation dashboard"
        android:icon="@drawable/ic dashboard black 24dp"
        android:title="@string/title_dashboard" />
    <item
        android:id="@+id/navigation notifications"
        android:icon="@drawable/ic notifications black 24dp"
        android:title="@string/title notifications" />
</menu>
```

上面的菜单定义文件以menu为根节点，内部容纳3个item节点，分别对应屏幕底部的3个标签。每个item节点都拥有id、icon、title 3个属性，其中id指定该菜单项的编号，icon指定该菜单项的图标，title指定该菜单项的文本。顺藤摸瓜查看values目录之下的strings.xml，果然找到了下面的3个标签文本定义：

```
<string name="title home">Home</string>
<string name="title dashboard">Dashboard</string>
<string name="title_notifications">Notifications</string>
```

搞清楚了底部标签栏的资源情况，接着打开navigation目录之下的mobile_navigation.xml，究竟里面是怎么定义各个频道的呢？

```
(完整代码见chapter09\src\main\res\navigation\mobile navigation.xml)
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/mobile navigation"
    app:startDestination="@+id/navigation home">
    <fragment
        android:id="@+id/navigation_home"
        android:name="com.example.chapter09.ui.home.HomeFragment"
        android:label="@string/title home"
        tools:layout="@layout/fragment home" />
    <fragment
        android:id="@+id/navigation_dashboard"
        android:name="com.example.chapter09.ui.dashboard.DashboardFragment"
        android:label="@string/title dashboard"
        tools:layout="@layout/fragment dashboard" />
    <fragment
        android:id="@+id/navigation_notifications"
        android:name="com.example.chapter09.ui.notifications.NotificationsFragment"
        android:label="@string/title notifications"
        tools:layout="@layout/fragment notifications" />
</navigation>
```

上述的导航定义文件以navigation为根节点，内部依旧分布着3个fragment节点，显然正好对应3个频道。每个fragment节点拥有id、name、label、layout 4个属性，各属性的用途说明如下：

- id：**指定当前碎片的编号**。
- name：**指定当前碎片的完整类名路径**。
- label：**指定当前碎片的标题文本**。
- layout：**指定当前碎片的布局文件**。

这些默认的碎片代码到底有何不同，打开其中一个HomeFragment.java研究研究，它的关键代码如下：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\ui\home\HomeFragment.java)
public View onCreateView(@NonNull LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    homeViewModel = ViewModelProviders.of(this).get(HomeViewModel.class);
    View root = inflater.inflate(R.layout.fragment_home, container, false);
    final TextView textView = root.findViewById(R.id.text_home);
    homeViewModel.getText().observe(this, new Observer<String>() {
        @Override
        public void onChanged(@Nullable String s) {
            textView.setText(s);
        }
    });
    return root;
}
```

看来频道用到的碎片代码仍然在onCreateView方法中根据XML布局文件生成页面元素，这样频道界面的修改操作就交给碎片编码了。总算理清了这种底部导航的实现方式，接下来准备修理修理默认的标签及其频道。先打开values目录之下的strings.xml，把3个标签的文字从英文改成中文，修改内容示例如下：

```
<string name="title_home">首页</string>
<string name="title_dashboard">仪表盘</string>
<string name="title_notifications">消息</string>
```

再打开3个频道的碎片代码，给文本视图填上中文描述，首页频道HomeFragment.java修改之后的代码示例如下：

```
public View onCreateView(@NonNull LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    View root = inflater.inflate(R.layout.fragment_home, container, false);
    final TextView textView = root.findViewById(R.id.text_home);
    textView.setText("这是首页页面");
    return root;
}
```

重新编译运行App，改过的各频道界面如图9-5～图9-7所示，其中图9-5为首页频道的页面效果，图9-6为仪表盘频道的页面效果，图9-7为消息频道的页面效果，可见3个频道的标签文本和说明描述都改成了中文。



图 9-5 首页频道界面



图 9-6 仪表盘频道界面



图 9-7 消息频道界面

9.1.2 自定义标签按钮

按钮控件种类繁多，有文本按钮 Button、图像按钮 ImageButton、单选按钮 RadioButton、复选框 CheckBox、开关按钮 Switch 等，支持展现的形式有文本、图像、文本+图标，如此丰富的展现形式，已经能够满足大部分需求。但总有少数场合比较特殊，一般的按钮样式满足不了，比如图 9-8 所示的微信底部标签栏，一排有 4 个标签按钮，每个按钮的图标和文字都会随着选中而高亮显示。



图 9-8 微信的底部标签栏

这样的标签栏是各大主流 App 的标配，无论是淘宝、京东，还是微信、手机 QQ，首屏底部是清一色的标签栏，而且在选中标签按钮时该标签对应的文字、图标、背景一起高亮显示。虽然上一小节使用 Android Studio 自动生成了底部标签栏，但是整个标签栏都封装进了 BottomNavigationView，看不到标签按钮的具体实现，令人不知其所以然。像这种标签按钮，Android 似乎没有对应的专门控件，如果要自定义控件，就得设计一个布局容器，里面放入一个文本控件和图像控件，然后注册选中事件的监听器，一旦监听到选中事件，就高亮显示它的文字、图标与布局背景。

自定义控件固然是一个不错的思路，不过无须如此大动干戈。早在第5章的“5.2.2 开关按钮 Switch”中，介绍了结合状态图形与复选框实现仿iOS开关按钮的例子，通过状态图形自动展示选中与未选中两种状态的图像，使得复选框在外观上就像一个新控件。标签控件也是如此，要想高亮显示背景，可以给background属性设置状态图形；要想高亮显示图标，可以给drawableTop属性设置状态图形；要想高亮显示文本，可以给textColor属性设置状态图形。既然背景、图标、文字都能通过状态图形控制是否高亮显示，接下来的事情就好办了，具体的实现步骤如下：

01 定义一个状态图形的XML描述文件，当状态为选中时展示高亮图标，其余情况展示普通图标，于是状态图形的XML内容示例如下：

```
(完整代码见chapter09\src\main\res\drawable\tab_bg_selector.xml)
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_checked="true" android:drawable="@drawable/tab_bg_selected" />
    <item android:drawable="@drawable/tab_bg_normal" />
</selector>
```

上面定义的tab_bg_selector.xml用于控制标签背景的状态显示，控制文本状态的tab_text_selector.xml和控制图标状态的tab_first_selector.xml可如法炮制。

02 在活动页面的XML文件中添加CheckBox节点，并给该节点的background、drawableTop、textColor 3个属性分别设置对应的状态图形，修改后的XML文件内容如下：

```
(完整代码见chapter09\src\main\res\layout\activity_tab_button.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <!-- 复选框的背景、文字颜色和顶部图标都采用了状态图形，看起来像个崭新的标签控件 -->
    <CheckBox
        android:id="@+id/ck_tab"
        android:layout_width="100dp"
        android:layout_height="60dp"
        android:padding="5dp"
```

```
        android:gravity="center"
        android:button="@null"
        android:background="@drawable/tab_bg_selector"
        android:text="点我"
        android:textSize="12sp"
        android:textColor="@drawable/tab_text_selector"
        android:drawableTop="@drawable/tab_first_selector" />
    <TextView
        android:id="@+id/tv_select"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="这里查看标签选择结果" />
</LinearLayout>
```

08 活动页面的Java代码给复选框ck_tab设置勾选监听器，用来监听复选框的选中事件和取消选中事件，活动代码如下：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\TabButtonActivity.java)
public class TabButtonActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_tab_button);
        final TextView tv_select = findViewById(R.id.tv_select);
        CheckBox ck_tab = findViewById(R.id.ck_tab);
        // 给复选框设置勾选监听器
        ck_tab.setOnCheckedChangeListener((buttonView, isChecked) -> {
            if (buttonView.getId() == R.id.ck_tab) {
                String desc = String.format("标签按钮被%s了",
                                            isChecked?"选中":"取消选中");
                tv_select.setText(desc);
            }
        });
    }
}
```

运行测试App，一开始的标签按钮界面如图9-9所示。首次点击标签控件，复选框变为选中状态，它的文字、图标、背景同时高亮显示，如图9-10所示；再次点击标签控件，复选框变为取消选中状态，它的文字、图标、背景同时恢复原状，如图9-11所示。



图 9-9 初始的标签按钮界面

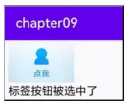


图 9-10 首次点击的标签按钮

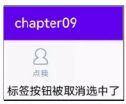


图 9-11 再次点击的标签按钮

是不是很神奇？接下来不妨把该控件的共同属性挑出来，因为底部标签栏通常有4、5个标签按钮，如果每个按钮节点都添加重复的属性，就太啰嗦了，所以把它们之间通用的属性挑出来，然后在values/styles.xml中定义名为TabButton的新风格，具体的风格内容如下：

```
(完整代码见chapter09\src\main\res\values\styles.xml)
<style name="TabButton">
    <item name="android:layout_width">0dp</item>
    <item name="android:layout_height">match_parent</item>
    <item name="android:layout_weight">1</item>
    <item name="android:padding">5dp</item>
```

```
    <item name="android:gravity">center</item>
    <item name="android:background">@drawable/tab bg selector</item>
    <item name="android:textSize">12sp</item>
    <item name="android:textColor">@drawable/tab text selector</item>
    <item name="android:button">@null</item>
</style>
```

然后，XML文件只要给CheckBox节点添加一行style="@style/TabButton"，即可将其变为标签按钮。直接在styles.xml中定义风格，无须另外编写自定义控件的代码，这是自定义控件的另一种途径。

回到前述活动页面的XML文件，补充以下的布局内容，表示添加一行3个标签控件，也就是3个CheckBox节点都声明了style="@style/TabButton"，同时每个CheckBox另外指定自己的标签文字和标签图标。

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="60dp"
    android:orientation="horizontal">
    <CheckBox
        style="@style/TabButton"
        android:checked="true"
        android:drawableTop="@drawable/tab_first_selector"
        android:text="首页" />
    <CheckBox
        style="@style/TabButton"
        android:drawableTop="@drawable/tab second selector"
        android:text="分类" />
    <CheckBox
        style="@style/TabButton"
        android:drawableTop="@drawable/tab third selector"
        android:text="购物车" />
</LinearLayout>
```

重新运行测试App，发现标签控件界面多了一排标签按钮，分别是“首页”“分类”“购物车”，如图9-12所示。



图 9-12 整合了 3 个标签按钮的标签栏

多次点击 3 个按钮，它们的外观都遵循一种样式状态，可见统一的风格定义果然奏效了。

9.1.3 结合 RadioGroup 和 ViewPager 自定义底部标签栏

尽管使用 Android Studio 很容易生成自带底部标签栏的活动页面，可是该标签栏基于BottomNavigationView，标签的样式风格不方便另行调整，况且它也不支持通过左右滑动切换标签。因此，开发者若想实现拥有更多花样的标签栏，就得自己定义专门的底部标签栏了。

话说翻页视图ViewPager搭配翻页标签栏PagerTabStrip，本来已经实现了带标签的翻页功能，不过这个标签位于翻页视图上方而非下方，而且只有标签文字没有标签图标，比起BottomNavigationView更不友好，所以用不了PagerTabStrip。鉴于标签栏每次只能选中一项标签，这种排他性与单选按钮类似，理论上采用一排单选按钮也能实现标签栏的单选功能。只是单选按钮的外观不满足要求，中用却不中看。这点小瑕疵倒也无妨，把它的样式改成上一小节介绍的标签按钮就行，也就是给RadioButton标签添加样式属性style="@style/TabButton"。然后用单选组

RadioGroup容纳这几个单选按钮，再把单选组放在页面底部，把翻页视图放在单选组上方，于是整个页面的XML文件变成下面这样：

```
(完整代码见chapter09\src\main\res\layout\activity_tab_pager.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <androidx.viewpager.widget.ViewPager
        android:id="@+id/vp_content"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />
    <RadioGroup
        android:id="@+id/rg_tabbar"
        android:layout_width="match_parent"
        android:layout_height="60dp"
        android:orientation="horizontal">
        <RadioButton
            android:id="@+id/rb_home"
            style="@style/TabButton"
            android:checked="true"
            android:text="首页"
            android:drawableTop="@drawable/tab_first_selector" />
        <RadioButton
            android:id="@+id/rb_class"
            style="@style/TabButton"
            android:text="分类"
            android:drawableTop="@drawable/tab_second_selector" />
        <RadioButton
            android:id="@+id/rb_cart"
            style="@style/TabButton"
            android:text="购物车"
            android:drawableTop="@drawable/tab_third_selector" />
    </RadioGroup>
</LinearLayout>
```

该页面对应的Java代码主要实现以下两个切换逻辑：

（1）左右滑动翻页视图的时候，每当页面滚动结束，就自动选择对应位置的单选按钮。

（2）点击某个单选按钮的时候，先判断当前选择的是第几个按钮，再将翻页视图翻到第几个页面。

具体到编码实现，则要给翻页视图添加页面变更监听器，并补充翻页完成的处理操作；还要给单选组注册选择监听器，并补充选中之后的处理操作。详细的活动代码如下：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\TabPagerActivity.java)
public class TabPagerActivity extends AppCompatActivity {
    private ViewPager vp_content;                           // 声明一个翻页视图对象
    private RadioGroup rg_tabbar;                           // 声明一个单选组对象
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_tab_pager);
        vp_content = findViewById(R.id.vp_content);         // 从布局文件获取翻页视图
        // 构建一个翻页适配器
        TabPagerAdapter adapter = new TabPagerAdapter(getSupportFragmentManager());
        vp_content.setAdapter(adapter);                     // 设置翻页视图的适配器
        // 给翻页视图添加页面变更监听器
```

```
        vp_content.addOnPageChangeListener(new ViewPager.SimpleOnPageChangeListener() {
            @Override
            public void onPageSelected(int position) {
                // 选中指定位置的单选按钮
                rg_tabbar.check(rg_tabbar.getChildAt(position).getId());
            }
        });
        rg tabbar = findViewById(R.id.rg tabbar);          // 从布局文件获取单选组
        // 设置单选组的选中监听器
        rg_tabbar.setOnCheckedChangeListener((group, checkedId) -> {
            for (int pos=0; pos<rg_tabbar.getChildCount(); pos++) {
                // 获得指定位置的单选按钮
                RadioButton tab = (RadioButton) rg tabbar.getChildAt(pos);
                if (tab.getId() == checkedId) {            // 正是当前选中的按钮
                    vp content.setCurrentItem(pos);        // 设置翻页视图显示第几页
                }
            }
        });
    }
}
```

由于翻页视图需要搭配翻页适配器，因此以上代码给出了一个适配器TabPagerAdapter，该适配器的代码很简单，仅仅返回3个碎片而已，下面是翻页适配器的代码例子：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\adapter\TabPagerAdapter.java)
public class TabPagerAdapter extends FragmentPagerAdapter {
    // 翻页适配器的构造方法，传入碎片管理器
    public TabPagerAdapter(FragmentManager fm) {
        super(fm, BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT);
    }
    // 获取指定位置的碎片Fragment
    @Override
    public Fragment getItem(int position) {
        if (position == 0) {
            return new TabFirstFragment();         // 返回第一个碎片
        } else if (position == 1) {
            return new TabSecondFragment();        // 返回第二个碎片
        } else if (position == 2) {
            return new TabThirdFragment();         // 返回第三个碎片
        } else {
            return null;
        }
    }
    // 获取碎片Fragment的个数
    @Override
    public int getCount() {
        return 3;
    }
}
```

若要说3个碎片TabFirstFragment、TabSecondFragment、TabThirdFragment分别干了什么，其实他们内部都只放了一个文本视图，嫌麻烦的话可以参考前面“9.1.1　利用BottomNavigationView实现底部标签栏”修改之后的碎片代码。

至此从活动页面到适配器再到碎片全部重写了一遍，运行测试 App，打开该活动的初始界面，如图 9-13 所示，可见此时显示第一个碎片内容，下方的标签栏默认选择第一项的“首页”按钮；点击第二项的“分类”按钮，发现上方切换到了第二个碎片，如图 9-14 所示；在空白处从右向左滑动，拉出第三个碎片内容，此时标签栏自动选择了第三项的“购物车”按钮，如图 9-15 所示。



图 9-13 默认的首页界面



图 9-14 点击了“分类”标签



图 9-15 自动选择了“购物车”标签

根据上述的点击标签和滑动翻页结果，可知结合 RadioGroup 和 ViewPager 完美实现了自定义的底部标签栏。

9.2 顶部导航栏

本节介绍顶部导航栏的组成控件：首先描述工具栏Toolbar的基本用法，然后叙述溢出菜单OverflowMenu的格式及其用法，最后讲解标签布局TabLayout的相关属性和方法用途。

9.2.1 工具栏 Toolbar

主流App除了底部有一排标签栏外，通常顶部还有一排导航栏。在Android 5.0之前，这个顶部导航栏用的是ActionBar控件，但ActionBar存在不灵活、难以扩展等毛病，所以Android 5.0之后推出了Toolbar工具栏，意在取代ActionBar。

不过为了兼容之前的系统版本，ActionBar仍然保留。当然，由于Toolbar与ActionBar都占着顶部导航栏的位置，二者肯定不能共存，因此要想引入Toolbar就得先关闭ActionBar。具体的操作步骤如下：

01 在styles.xml中定义一个不包含ActionBar的风格样式，代码如下：

```
<style name="AppCompatTheme" parent="Theme.AppCompat.Light.NoActionBar" />
```

02 修改AndroidManifest.xml，给activity节点添加android:theme属性，并将属性值设为步骤1定义的风格，如android:theme="@style/AppCompatTheme"。

03 将活动页面的XML文件根节点改成LinearLayout，且为vertical（垂直方向）；然后增加一个Toolbar节点，因为Toolbar本质是一个ViewGroup，所以允许在内部添加其他控件。下面是Toolbar节点的XML例子：

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/tl_head"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

04 打开活动页面的Java代码，在onCreate方法中获取布局文件中的Toolbar对象，并调用setSupportActionBar方法设置当前的Toolbar对象。对应代码如下：

```
// 从布局文件中获取名为tl_head的工具栏
Toolbar tl_head = findViewById(R.id.tl_head);
setSupportActionBar(tl_head);  // 使用tl_head替换系统自带的ActionBar
```

Toolbar之所以比ActionBar灵活，原因之一是Toolbar提供了多个属性，方便定制各种控件风格。它的常用属性及其设置方法见表9-1。

表 9-1　Toolbar 的常用属性及设置方法说明

XML 中的属性	Toolbar 类的设置方法	说　明
logo	setLogo	设置工具栏图标
title	setTitle	设置标题文字
titleTextColor	setTitleTextColor	设置标题的文字颜色
subtitle	setSubtitle	设置副标题文字。副标题在标题下方
subtitleTextColor	setSubtitleTextColor	设置副标题的文字颜色
navigationIcon	setNavigationIcon	设置左侧的箭头导航图标
无	setNavigationOnClickListener	设置导航图标的点击监听器

结合表9-1提到的设置方法，下面是给Toolbar设置风格的代码例子：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\ToolbarActivity.java)
// 从布局文件中获取名为tl_head的工具栏
Toolbar tl_head = findViewById(R.id.tl_head);
tl_head.setTitle("工具栏页面");                    // 设置工具栏的标题文本
setSupportActionBar(tl_head);                     // 使用tl_head替换系统自带的ActionBar
tl_head.setTitleTextColor(Color.RED);             // 设置工具栏的标题文字颜色
tl_head.setLogo(R.drawable.ic_app);               // 设置工具栏的标志图片
tl_head.setSubtitle("Toolbar");                   // 设置工具栏的副标题文本
tl_head.setSubtitleTextColor(Color.YELLOW);       // 设置工具栏的副标题文字颜色
tl_head.setBackgroundResource(R.color.blue_light);   // 设置工具栏的背景
tl_head.setNavigationIcon(R.drawable.ic_back);       //设置工具栏左边的导航图标
// 给tl_head设置导航图标的点击监听器
// setNavigationOnClickListener必须放到setSupportActionBar之后，不然不起作用
tl_head.setNavigationOnClickListener(view -> {
    finish();                                             // 结束当前页面
});
```

运行测试App，观察到工具栏效果如图9-16所示，可见该工具栏包括导航箭头图标、工具栏图标、标题、副标题。

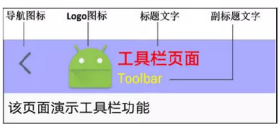


图 9-16　简单设置后的工具栏效果

9.2.2　溢出菜单 OverflowMenu

导航栏右边往往有个3点图标，点击后会在界面右上角弹出菜单。这个菜单名为溢出菜单OverflowMenu，意思是导航栏不够放、溢出来了。溢出菜单的格式同“9.1.1　利用BottomNavigationView实现底部标签栏”介绍的导航菜单，它们都在res\menu下面的XML文件中定义，不过溢出菜单多了个app:showAsAction属性，该属性用来控制菜单项在导航栏上的展示位置，位置类型的取值说明见表9-2。

表 9-2 菜单项展示位置类型的取值说明

展示位置类型	说 明
always	总是在导航栏上显示菜单图标
ifRoom	如果导航栏右侧有空间，该项就直接显示在导航栏上，不再放入溢出菜单
never	从不在导航栏上直接显示，总是放在溢出菜单列表里面
withText	如果能在导航栏上显示，除了显示图标，还要显示该菜单项的文字说明

注意，因为showAsAction是菜单的自定义属性，所以要先在菜单XML的menu根节点增加命名空间声明xmlns:app="http://schemas.android.com/apk/res-auto"，这样showAsAction指定的位置类型才会生效。下面是一个包含3个菜单项的溢出菜单XML例子：

```
(完整代码见chapter09\src\main\res\menu\menu overflow.xml)
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto" >
    <item
        android:id="@+id/menu refresh"
        android:icon="@drawable/ic refresh"
        app:showAsAction="ifRoom"
        android:title="刷新" />
    <item
        android:id="@+id/menu about"
        android:icon="@drawable/ic about"
        app:showAsAction="never"
        android:title="关于" />
    <item
        android:id="@+id/menu quit"
        android:icon="@drawable/ic quit"
        app:showAsAction="never"
        android:title="退出" />
</menu>
```

有了上面的菜单文件menu_overflow.xml，还得在活动代码中增加对菜单的处理逻辑。下面是在活动页面中操作溢出菜单的代码片段：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\OverflowMenuActivity.java)
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // 从menu overflow.xml中构建菜单界面布局
    getMenuInflater().inflate(R.menu.menu overflow, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();                      // 获取菜单项的编号
    if (id == android.R.id.home) {                  // 点击了工具栏左边的返回箭头
        finish();  // 结束当前页面
    } else if (id == R.id.menu refresh) {           // 点击了刷新图标
        tv desc.setText("当前刷新时间: " + DateUtil. getNowTime());
    } else if (id == R.id.menu about) {             // 点击了关于菜单项
        Toast.makeText(this, "这个是工具栏的演示demo", Toast.LENGTH_LONG).show();
    } else if (id == R.id.menu_quit) {              // 点击了退出菜单项
        finish();  // 结束当前页面
    }
    return super.onOptionsItemSelected(item);
}
```

运行测试App，打开添加了溢出菜单的导航栏页面，可见初始界面如图9-17所示，此时导航栏右侧有刷新按钮和3点图标；点击3点图标，弹出剩余的菜单项列表，如图9-18所示，点击某个菜单项即可触发对应的菜单事件。

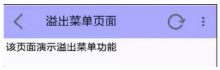


图9-17　溢出菜单初始界面

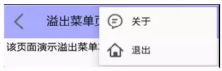


图9-18　点击3点图标弹出菜单列表

9.2.3　标签布局 TabLayout

Toolbar作为ActionBar的升级版，它不仅允许设置内部控件的样式，还允许添加其他外部控件。例如京东App的商品页面，既有图9-19所示的商品页，又有图9-20所示的详情页。可见这个导航栏拥有一排文字标签，类似于翻页视图附属的翻页标题栏PagerTabStrip，商品页和详情页之间通过点击标签进行切换。

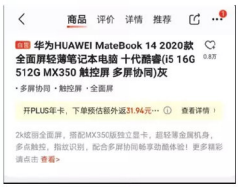


图9-19　京东的商品页面



图9-20　京东的详情页面

通过导航栏集成文字切换标签，有效提高了页面空间的利用效率，该功能用到了design库中的标签布局TabLayout。使用该控件前要先修改build.gradle，在dependencies节点中加入以下配置表示导入design库：

```
implementation 'com.google.android.material:material:1.5.0'
```

TabLayout的展现形式类似于PagerTabStrip，同样是文字标签带下划线，不同的是TabLayout允许定制更丰富的样式，它新增的样式属性主要有下列6种。

- tabBackground：**指定标签的背景**。
- tabIndicatorColor：**指定下划线的颜色**。
- tabIndicatorHeight：**指定下划线的高度**。
- tabTextColor：**指定标签文字的颜色**。
- tabTextAppearance：**指定标签文字的风格。样式风格来自**styles.xml**中的定义**。
- tabSelectedTextColor：**指定选中文字的颜色**。

下面是在XML文件中通过Toolbar 集成TabLayout的内容片段：

```
(完整代码见chapter09\src\main\res\layout\activity_tab_layout.xml)
<androidx.appcompat.widget.Toolbar
```

```
        android:id="@+id/tl_head"
        android:layout width="match parent"
        android:layout height="50dp"
        app:navigationIcon="@drawable/ic back">
        <!-- 注意TabLayout节点需要使用完整路径 -->
        <com.google.android.material.tabs.TabLayout
            android:id="@+id/tab title"
            android:layout width="wrap content"
            android:layout height="match parent"
            android:layout_centerInParent="true"
            app:tabIndicatorColor="@color/red"
            app:tabIndicatorHeight="2dp"
            app:tabSelectedTextColor="@color/red"
            app:tabTextColor="@color/grey"
            app:tabTextAppearance="@style/TabText" />
    </androidx.appcompat.widget.Toolbar>
```

在Java代码中，TabLayout通过以下4个方法操作文字标签。

- newTab：**创建新标签。**
- addTab：**添加一个标签。**
- getTabAt：**获取指定位置的标签。**
- setOnTabSelectedListener：**设置标签的选中监听器。该监听器需要实现OnTabSelectedListener接口的3个方法。**
  - onTabSelected：标签被选中时触发。
  - onTabUnselected：标签被取消选中时触发。
  - onTabReselected：标签被重新选中时触发。

把TabLayout与ViewPager结合起来就是一个固定的套路，二者各自通过选中监听器或者翻页监听器控制页面切换，使用时直接套框架就行。下面是两者联合使用的代码例子：

```
    (完整代码见chapter09\src\main\java\com\example\chapter09\TabLayoutActivity.java)
    public class TabLayoutActivity extends AppCompatActivity implements
OnTabSelectedListener {
        private ViewPager vp_content;                       // 声明一个翻页视图对象
        private TabLayout tab title;                        // 声明一个标签布局对象
        private String[] mTitleArray = {"商品", "详情"};    // 标题文字数组

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_tab_layout);
            // 从布局文件中获取名为tl head的工具栏
            Toolbar tl head = findViewById(R.id.tl head);
            tl head.setTitle("");                           // 设置工具栏的标题文本
            setSupportActionBar(tl_head);                   // 使用tl_head替换系统自带的ActionBar
            initTabLayout();                                // 初始化标签布局
            initTabViewPager();                             // 初始化标签翻页
        }
        // 初始化标签布局
        private void initTabLayout() {
            // 从布局文件中获取名为tab_title的标签布局
            tab title = findViewById(R.id.tab title);
            // 给标签布局添加一个文字标签
            tab title.addTab(tab title.newTab().setText(mTitleArray[0]));
            // 给标签布局添加一个文字标签
            tab_title.addTab(tab_title.newTab().setText(mTitleArray[1]));
```

```
        tab_title.addOnTabSelectedListener(this);          // 给标签布局添加标签选中监听器
    }
    // 初始化标签翻页
    private void initTabViewPager() {
        // 从布局文件中获取名为vp_content的翻页视图
        vp_content = findViewById(R.id.vp_content);
        // 构建一个商品信息的翻页适配器
        GoodsPagerAdapter adapter = new GoodsPagerAdapter(
                getSupportFragmentManager(), mTitleArray);
        vp_content.setAdapter(adapter);                    // 设置翻页视图的适配器
        // 给vp_content添加页面变更监听器
        vp_content.addOnPageChangeListener(new SimpleOnPageChangeListener() {
            public void onPageSelected(int position) {
                tab_title.getTabAt(position).select();     // 选中指定位置的标签
            }
        });
    }
    // 在标签被重复选中时触发
    public void onTabReselected(Tab tab) {}
    // 在标签被选中时触发
    public void onTabSelected(Tab tab) {
        vp_content.setCurrentItem(tab.getPosition());      // 设置翻页视图显示第几页
    }
    // 在标签被取消选中时触发
    public void onTabUnselected(Tab tab) {}
}
```

运行测试App，准备观察标签布局的界面效果。先点击“商品”标签，此时页面显示商品的图片概览，如图9-21所示；再点击“详情”标签，切换到商品的详情页面，如图9-22所示。感觉不错吧，赶快动手实践一下，你也可以实现京东App的标签导航栏。

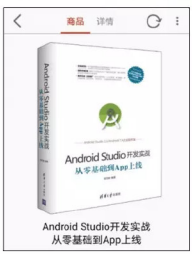


图9-21　点击了“商品”标签

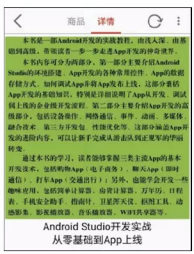


图9-22　点击了“详情”标签

9.3　增强型列表

本节介绍如何利用循环视图RecyclerView实现3种增强型列表，包括线性列表布局、普通网格布局、瀑布流网格布局等，以及如何动态更新循环视图内部的列表项数据。

9.3.1 循环视图 RecyclerView

尽管ListView和GridView分别实现了多行单列和多行多列的列表，使用也很简单，可是它们缺少变化，风格也比较呆板。为此Android推出了更灵活多变的循环视图RecyclerView，它的功能非常强大，不但足以囊括列表视图和网格视图，还能实现高度错开的瀑布流网格效果。总之，只要学会了RecyclerView，就相当于同时掌握了ListView、GridView，再加上瀑布流一共3种列表界面。

由于RecyclerView来自recyclerview库，因此在使用RecyclerView前要修改build.gradle，在dependencies节点中加入以下配置表示导入recyclerview库：

```
implementation 'androidx.recyclerview:recyclerview:1.2.0'
```

下面是RecyclerView的常用方法说明。

- setAdapter：**设置列表项的循环适配器。适配器采用**RecyclerView.Adapter。
- setLayoutManager：**设置列表项的布局管理器。管理器一共**3**种，包括线性布局管理器**LinearLayoutManager、**网格布局管理器**GridLayoutManager、**瀑布流网格布局管理器**StaggeredGridLayoutManager。
- addItemDecoration：**添加列表项的分割线**。
- setItemAnimator：**设置列表项的变更动画。默认动画为**DefaultItemAnimator。
- scrollToPosition：**滚动到指定位置**。

循环视图有专门的循环适配器RecyclerView.Adapter，在setAdapter方法之前，得先实现一个从RecyclerView.Adapter派生而来的适配器，用来定义列表项的界面布局及其控件操作。下面是实现循环适配器时有待重写的方法说明。

- getItemCount：**获得列表项的数目**。
- onCreateViewHolder：**创建整个布局的视图持有者，可在该方法中指定列表项的布局文件。第二个参数为视图类型**viewType，**根据视图类型加载不同的布局，从而实现带头部的列表布局。**
- onBindViewHolder：**绑定列表项的视图持有者。可在该方法中操纵列表项的控件。**

以上3个方法是必需的，每个自定义的循环适配器都要重写这3个方法。

- getItemViewType：**返回每项的视图类型。这里的类型与**onCreateViewHolder**方法的**viewType**参数保持一致。**
- getItemId：**获得每个列表项的编号**。

以上2个方法不是必需的，可以重写也可以不重写。

举个公众号消息列表的例子，若要通过循环适配器实现的话，需要让自定义的适配器完成下列步骤：

01 在构造方法中传入消息列表。
02 重写getItemCount方法，返回列表项的个数。
03 定义一个由RecyclerView.ViewHolder派生而来的内部类，用作列表项的视图持有者。

04 重写onCreateViewHolder方法，根据指定的布局文件生成视图对象，并返回该视图对象对应的视图持有者。

05 重写onBindViewHolder方法，从参数中的视图持有者获取各个控件实例，再操纵这些控件（设置文字、设置图片、设置点击监听器等）。

依据上述步骤编写而成的循环适配器代码示例如下：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\adapter\
RecyclerLinearAdapter.java)
public class RecyclerLinearAdapter extends RecyclerView.Adapter<ViewHolder> {
    private Context mContext;                // 声明一个上下文对象
    private List<NewsInfo> mPublicList;      // 公众号列表
    public RecyclerLinearAdapter(Context context, List<NewsInfo> publicList) {
        mContext = context;
        mPublicList = publicList;
    }
    // 获取列表项的个数
    public int getItemCount() {
        return mPublicList.size();
    }
    // 创建列表项的视图持有者
    public ViewHolder onCreateViewHolder(ViewGroup vg, int viewType) {
        // 根据布局文件item_linear.xml生成视图对象
        View v = LayoutInflater.from(mContext).inflate(R.layout.item_linear,vg, false);
        return new ItemHolder(v);
    }
    // 绑定列表项的视图持有者
    public void onBindViewHolder(ViewHolder vh, final int position) {
        ItemHolder holder = (ItemHolder) vh;
        holder.iv pic.setImageResource(mPublicList.get(position).pic id);
        holder.tv title.setText(mPublicList.get(position).title);
        holder.tv desc.setText(mPublicList.get(position).desc);
    }
    // 定义列表项的视图持有者
    public class ItemHolder extends RecyclerView.ViewHolder {
        public ImageView iv pic;             // 声明列表项图标的图像视图
        public TextView tv title;            // 声明列表项标题的文本视图
        public TextView tv_desc;             // 声明列表项描述的文本视图
        public ItemHolder(View v) {
            super(v);
            iv pic = v.findViewById(R.id.iv pic);
            tv title = v.findViewById(R.id.tv title);
            tv_desc = v.findViewById(R.id.tv_desc);
        }
    }
}
```

回到活动页面，由循环视图对象调用setAdapter方法设置适配器，具体的调用代码如下：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\RecyclerLinearActivity.java)
// 初始化线性布局的循环视图
private void initRecyclerLinear() {
    // 从布局文件中获取名为rv_linear的循环视图
    RecyclerView rv_linear = findViewById(R.id.rv_linear);
    // 创建一个垂直方向的线性布局管理器
    LinearLayoutManager manager = new LinearLayoutManager( this, RecyclerView.VERTICAL,
                                                           false);
    rv_linear.setLayoutManager(manager);          // 设置循环视图的布局管理器
```

```
        // 构建一个公众号列表的线性适配器
        RecyclerLinearAdapter adapter = new RecyclerLinearAdapter( this,
                                        NewsInfo.getDefaultList());
        rv_linear.setAdapter(adapter);                  // 设置循环视图的线性适配器
    }
```

运行测试App，观察到公众号消息界面如图9-23所示。可见该效果仿照微信公众号的消息列表，看起来像是用ListView实现的。

注 意

循环视图并未提供点击监听器和长按监听器，若想让列表项能够响应点击事件，则需在适配器的onBindViewHolder方法中给列表项的根布局注册点击监听器，代码示例如下：

```
// 列表项的点击事件需要自己实现。ll_item为列表项的根布局
holder.ll_item.setOnClickListener(v -> {
    // 这里补充点击事件的处理代码
});
```



图 9-23 循环视图的简单实现

9.3.2 布局管理器 LayoutManager

循环视图之所以能够变身为3种列表，是因为它允许指定不同的列表布局，这正是布局管理器LayoutManager的拿手好戏。LayoutManager不但提供了3类布局管理，分别实现类似列表视图、网格视图、瀑布流网格的效果，而且可由循环视图对象随时调用setLayoutManager方法设置新布局。一旦调用了setLayoutManager方法，界面就会根据新布局刷新列表项。此特性特别适用于手机在竖屏与横屏之间的显示切换（如竖屏时展示列表，横屏时展示网格），也适用于在不同屏幕尺寸（如手机与平板）之间的显示切换（如在手机上展示列表，在平板上展示网格）。接下来分别介绍循环视图的3类布局管理器。

1. 线性布局管理器LinearLayoutManager

LinearLayoutManager可看作是线性布局LinearLayout，它在垂直方向布局时，展示效果类似于列表视图ListView；在水平方向布局时，展示效果类似于水平方向的列表视图。

下面是LinearLayoutManager的常用方法。

- **构造方法：第二个参数指定了布局方向，RecyclerView.HORIZONTAL表示水平，RecyclerView.VERTICAL表示垂直；第三个参数指定了是否从相反方向开始布局。**
- setOrientation：**设置布局的方向，RecyclerView.HORIZONTAL表示水平方向，RecyclerView.VERTICAL表示垂直方向。**
- setReverseLayout：**设置是否从相反方向开始布局，默认为**false。**如果设置为true，那么垂直方向将从下往上开始布局，水平方向将从右往左开始布局。**

前面在介绍循环视图时，采用了最简单的线性布局管理器，虽然调用addItemDecoration方法能够添加列表项的分隔线，但是RecyclerView并未提供默认的分隔线，需要先由开发者自定义分隔线的样式，再调用addItemDecoration方法设置分隔线样式。下面是个允许设置线宽的分隔线实现代码：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\widget\SpacesDecoration.java)
public class SpacesDecoration extends RecyclerView.ItemDecoration {
```

```
        private int space;                    // 空白间隔
        public SpacesDecoration(int space) {
            this.space = space;
        }
        @Override
        public void getItemOffsets(Rect outRect, View v, RecyclerView parent,
RecyclerView.State state) {
            outRect.left = space;             // 左边空白间隔
            outRect.right = space;            // 右边空白间隔
            outRect.bottom = space;           // 上方空白间隔
            outRect.top = space;              // 下方空白间隔
        }
    }
```

2. 网格布局管理器GridLayoutManager

GridLayoutManager可看作是网格布局GridLayout，从展示效果来看，GridLayoutManager类似于网格视图GridView。不管是GridLayout还是GridView，抑或GridLayoutManager，都呈现多行多列的网格界面。

下面是GridLayoutManager的常用方法。

- **构造方法：第二个参数指定了网格的列数。**
- **setSpanCount：设置网格的列数。**
- **setSpanSizeLookup：设置网格项的占位规则。默认一个网格项占一列，若想某个网格项占多列，就可在此设置占位规则。**

下面是在活动页面中给循环视图设置网格布局管理器的代码例子：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\RecyclerGridActivity.java)
// 初始化网格布局的循环视图
private void initRecyclerGrid() {
    // 从布局文件中获取名为rv_grid的循环视图
    RecyclerView rv_grid = findViewById(R.id.rv_grid);
    // 创建一个网格布局管理器（每行5列）
    GridLayoutManager manager = new GridLayoutManager(this, 5);
    rv_grid.setLayoutManager(manager);        // 设置循环视图的布局管理器
    // 构建一个市场列表的网格适配器
    RecyclerGridAdapter adapter = new RecyclerGridAdapter( this,
                                  NewsInfo.getDefaultGrid());
    rv_grid.setAdapter(adapter);              // 设置循环视图的网格适配器
}
```

运行测试App，观察到网格布局管理器的循环视图界面如图9-24所示，看起来跟GridView的展示效果没什么区别。



图 9-24 循环视图的网格布局

但GridLayoutManager绝非GridView可比，因为它还提供了setSpanSizeLookup方法，该方法允许一个网格占据多列，展示更加灵活。下面是使用占位规则的网格管理器代码例子：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\RecyclerCombineActivity.java)
private void initRecyclerCombine() {
    // 从布局文件中获取名为rv_combine的循环视图
    RecyclerView rv_combine = findViewById(R.id.rv_combine);
    // 创建一个4列的网格布局管理器
    GridLayoutManager manager = new GridLayoutManager(this, 4);
    // 设置网格布局管理器的占位规则。以下规则为：第一项和第二项占两列，其他项占一列；
    // 如果网格的列数为4，那么第一项和第二项平分第一行，第二行开始每行有4项
```

```
        manager.setSpanSizeLookup(new GridLayoutManager.SpanSizeLookup() {
            public int getSpanSize(int position) {
                if (position == 0 || position == 1) {       // 为第一项或者第二项
                    return 2;                               // 占据两列
                } else {                                    // 为其他项
                    return 1;                               // 占据一列
                }
            }
        });
        rv_combine.setLayoutManager(manager);               // 设置循环视图的布局管理器
        // 构建一个猜你喜欢的网格适配器
        RecyclerCombineAdapter adapter = new RecyclerCombineAdapter(
                                        this, NewsInfo.getDefaultCombine());
        rv_combine.setAdapter(adapter);                     // 设置循环视图的网格适配器
    }
```

运行测试App，观察到占位规则的界面效果如图9-25所示。可见第一行只有两个网格，而第二行有4个网格，这意味着第一行的每个网格都占据了两列位置。



图 9-25 循环视图的合并网格布局效果

3. 瀑布流网格布局管理器StaggeredGridLayoutManager

电商App在展示众多商品信息时，往往通过高矮不一的格子展示。因为不同商品的外观尺寸不一样，比如冰箱在纵向比较长，空调在横向比较长，所以若用一样规格的网格展示，必然导致有的商品图片会被压缩得很小。像这种根据不同的商品形状展示不同高度的图片，就是瀑布流网格的应用场合。自从有了瀑布流网格布局管理器StaggeredGridLayoutManager，瀑布流效果的开发过程便大大简化了，只要在循环适配器中动态设置每个网格的高度，系统就会在界面上自动排列瀑布流网格。

下面是StaggeredGridLayoutManager的常用方法。

- **构造方法：第一个参数指定了瀑布流网格每行的列数；第二个参数指定了瀑布流布局的方向，取值说明同**LinearLayoutManager。
- setSpanCount：**设置瀑布流网格每行的列数。**
- setOrientation：**设置瀑布流布局的方向。取值说明同**LinearLayoutManager。
- setReverseLayout：**设置是否从相反方向开始布局，默认为**false。**如果设置为**true，**那么垂直方向将从下往上开始布局，水平方向将从右往左开始布局。**

下面是在活动页面中操作瀑布流网格布局管理器的代码例子：

(完整代码见chapter09\src\main\java\com\example\chapter09\RecyclerStaggeredActivity.java)

```
// 初始化瀑布流布局的循环视图
private void initRecyclerStaggered() {
    // 从布局文件中获取名为rv staggered的循环视图
    RecyclerView rv_staggered = findViewById(R.id.rv_staggered);
    // 创建一个垂直方向的瀑布流布局管理器（每行3列）
    StaggeredGridLayoutManager manager = new StaggeredGridLayoutManager(
            3, RecyclerView.VERTICAL);
    rv_staggered.setLayoutManager(manager);         // 设置循环视图的布局管理器
    // 构建一个服装列表的瀑布流适配器
    RecyclerStagAdapter adapter = new RecyclerStagAdapter(
                                this, NewsInfo.getDefaultStag());
    rv_staggered.setAdapter(adapter);               // 设置循环视图的瀑布流适配器
}
```

运行测试App，观察到瀑布流网格布局的效果如图9-26和图9-27所示，每个网格的高度依照具体图片的高度变化而变化，使得整个页面更加生动活泼。读者可以打开淘宝App，在顶部导航栏搜索“连衣裙”，看看搜索结果页面是不是如瀑布流网格这般交错显示。



图9-26　循环视图的瀑布流效果1



图9-27　循环视图的瀑布流效果2

9.3.3　动态更新循环视图

循环视图不但支持多种布局，而且更新内部数据也很方便。原先列表视图或者网格视图若想更新列表项，只能调用setAdapter方法重新设置适配器，或者由适配器对象调用notifyDataSetChanged刷新整个列表界面，可是这两种更新方式都得重新加载全部列表项，非常低效。相比之下，循环视图允许动态更新局部记录，既能对一条列表项单独添加/修改/删除，也能更新全部列表项。这种动态更新功能用到了循环适配器对象的下列方法：

- notifyItemInserted：**通知适配器在指定位置插入了新项。**
- notifyItemRemoved：**通知适配器在指定位置删除了原有项。**
- notifyItemChanged：**通知适配器在指定位置发生了数据变化。此时循环视图会刷新指定位置的列表项。**
- notifyDataSetChanged：**通知适配器整个列表的数据发生了变化。此时循环视图会刷新整个列表。**

动态更新列表项不仅只是功能上的增强，在更新之时还能展示变更动画，这是循环视图在用户体验上的优化。自从有了变更动画，列表项的增删动作看起来更加柔和，不再像列表视图或者网格视图那么呆板了。总之，只要用上了循环视图，你一定会对它爱不释手。

以公众号消息列表的更新操作为例，往循环视图顶部添加一条消息的步骤如下：

01 在适配器的数据列表头部添加新的消息数据。
02 调用适配器对象的notifyItemInserted方法，通知适配器在指定位置插入了新项。
03 调用循环视图的scrollToPosition方法，让它滚动到指定的列表项。

下面是在活动页面中动态添加公众号消息的代码例子：

```
    (完整代码见chapter09\src\main\java\com\example\chapter09\RecyclerDynamicActivity.java)
    public class RecyclerDynamicActivity extends AppCompatActivity implements
View.OnClickListener{
```

```
    private RecyclerView rv_dynamic;                    // 声明一个循环视图对象
    private LinearDynamicAdapter mAdapter;              // 声明一个线性适配器对象
    // 当前的公众号信息列表
    private List<NewsInfo> mPublicList = NewsInfo.getDefaultList();
    // 原始的公众号信息列表
    private List<NewsInfo> mOriginList = NewsInfo.getDefaultList();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_recycler_dynamic);
        findViewById(R.id.btn_recycler_add).setOnClickListener(this);
        initRecyclerDynamic();                          // 初始化动态线性布局的循环视图
    }
    // 初始化动态线性布局的循环视图
    private void initRecyclerDynamic() {
        // 从布局文件中获取名为rv_dynamic的循环视图
        rv_dynamic = findViewById(R.id.rv_dynamic);
        // 创建一个垂直方向的线性布局管理器
        LinearLayoutManager manager = new LinearLayoutManager(
                this, RecyclerView.VERTICAL, false);
        rv_dynamic.setLayoutManager(manager);           // 设置循环视图的布局管理器
        // 构建一个公众号列表的线性适配器
        mAdapter = new LinearDynamicAdapter(this, mPublicList);
        rv_dynamic.setAdapter(mAdapter);                // 设置循环视图的线性适配器
    }
    @Override
    public void onClick(View v) {
        if (v.getId() == R.id.btn_recycler_add) {
            // 获取一个随机位置
            int position = new Random().nextInt(mOriginList.size()-1);
            NewsInfo old_item = mOriginList.get(position);
            NewsInfo new_item = new NewsInfo(
                    old_item.pic_id, old_item.title, old_item.desc);
            mPublicList.add(0, new_item);               // 在顶部添加一条公众号消息
            mAdapter.notifyItemInserted(0);             // 通知适配器列表在第一项插入数据
            rv_dynamic.scrollToPosition(0);             // 让循环视图滚动到第一项所在的位置
        }
    }
}
```

运行测试App，观察到动态添加消息的界面效果如图9-28和图9-29所示。其中，图9-28为公众号消息列表的初始界面；点击“增加新聊天”按钮，在列表顶部新增一条公众号消息，注意在添加消息的时候会显示变更动画，图9-29为动画结束之后的公众号界面。



图9-28 消息的初始页面



图9-29 新增了一条消息

9.4 升级版翻页

本节介绍如何使用循环视图的扩展功能：首先引入下拉刷新布局 SwipeRefreshLayout，并说明如何通过下拉刷新动态更新循环视图；然后描述第二代翻页视图 ViewPager2 的基本用法，以及如何给 ViewPager2 搭档循环适配器；最后阐述如何给 ViewPager2 搭档专门的翻页适配器，以及如何集成标签布局。

9.4.1 下拉刷新布局 SwipeRefreshLayout

电商App在商品列表页面提供了下拉刷新功能，在屏幕顶端向下滑动即可触发页面的刷新操作，该功能用到了下拉刷新布局SwipeRefreshLayout。在使用SwipeRefreshLayout前要修改build.gradle，在dependencies节点中加入以下配置表示导入swiperefreshlayout库：

```
implementation 'androidx.swiperefreshlayout:swiperefreshlayout:1.1.0'
```

下面是SwipeRefreshLayout的常用方法说明。

- setOnRefreshListener：**设置刷新监听器。需要重写监听器**OnRefreshListener**的**onRefresh**方法，该方法在下拉松开时触发。**
- setRefreshing：**设置刷新的状态。**true**表示正在刷新，**false**表示结束刷新。**
- isRefreshing：**判断是否正在刷新。**
- setColorSchemeColors：**设置进度圆圈的圆环颜色。**

在XML文件中，SwipeRefreshLayout节点内部有且仅有一个直接子视图。如果存在多个直接子视图，那么只会展示第一个子视图，后面的子视图将不予展示。并且直接子视图必须允许滚动，包括滚动视图ScrollView、列表视图ListView、网格视图GridView、循环视图RecyclerView等。如果不是这些视图，当前界面就不支持滚动，更不支持下拉刷新。以循环视图为例，通过下拉刷新动态添加列表记录，从而省去一个控制按钮，避免按钮太多显得界面凌乱。

下面是结合SwipeRefreshLayout与RecyclerView的XML文件例子：

```
(完整代码见chapter09\src\main\res\layout\activity_swipe_recycler.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="match parent"
    android:layout height="match parent"
    android:orientation="vertical">
    <!-- 注意SwipeRefreshLayout要使用全路径 -->
    <androidx.swiperefreshlayout.widget.SwipeRefreshLayout
        android:id="@+id/srl dynamic"
        android:layout width="match parent"
        android:layout_height="wrap_content">
        <!-- 注意RecyclerView要使用全路径 -->
        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/rv dynamic"
            android:layout width="match parent"
            android:layout height="wrap content" />
    </androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
</LinearLayout>
```

与上面的XML文件对应的活动页面代码示例如下：

```
    (完整代码见chapter09\src\main\java\com\example\chapter09\SwipeRecyclerActivity.java)
    public class SwipeRecyclerActivity extends AppCompatActivity implements
OnRefreshListener {
        private SwipeRefreshLayout srl_dynamic;          // 声明一个下拉刷新布局对象
        private RecyclerView rv_dynamic;                 // 声明一个循环视图对象
        private LinearDynamicAdapter mAdapter;           // 声明一个线性适配器对象
        // 当前的公众号信息列表
        private List<NewsInfo> mPublicList = NewsInfo.getDefaultList();
        // 原始的公众号信息列表
        private List<NewsInfo> mOriginList = NewsInfo.getDefaultList();

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_swipe_recycler);
            // 从布局文件中获取名为srl_dynamic的下拉刷新布局对象
            srl_dynamic = findViewById(R.id.srl_dynamic);
            srl_dynamic.setOnRefreshListener(this);      // 设置下拉布局的下拉刷新监听器
            // 设置下拉刷新布局的进度圆圈颜色
            srl_dynamic.setColorSchemeResources(
                    R.color.red, R.color.orange, R.color.green, R.color.blue);
            initRecyclerDynamic();                       // 初始化动态线性布局的循环视图
        }

        // 初始化动态线性布局的循环视图
        private void initRecyclerDynamic() {
            // 从布局文件中获取名为rv_dynamic的循环视图
            rv_dynamic = findViewById(R.id.rv_dynamic);
            // 创建一个垂直方向的线性布局管理器
            LinearLayoutManager manager = new LinearLayoutManager(
                    this, RecyclerView.VERTICAL, false);
            rv_dynamic.setLayoutManager(manager);        // 设置循环视图的布局管理器
            // 构建一个公众号列表的线性适配器
            mAdapter = new LinearDynamicAdapter(this, mPublicList);
            rv_dynamic.setAdapter(mAdapter);             // 设置循环视图的线性适配器
        }

        // 一旦在下拉刷新布局内部往下拉动页面，就触发下拉监听器的onRefresh方法
        public void onRefresh() {
            mHandler.postDelayed(mRefresh, 2000);    // 模拟网络耗时，延迟若干秒后启动刷新任务
        }

        private Handler mHandler = new Handler();        // 声明一个处理器对象
        // 定义一个刷新任务
        private Runnable mRefresh = new Runnable() {
            public void run() {
                srl_dynamic.setRefreshing(false);        // 结束下拉刷新布局的刷新动作
                // 获取一个随机位置
                int position = new Random().nextInt(mOriginList.size()-1);
                NewsInfo old_item = mOriginList.get(position);
                NewsInfo new_item = new NewsInfo(
                        old_item.pic_id, old_item.title, old_item.desc);
                mPublicList.add(0, new_item);            // 在顶部添加一条公众号消息
                mAdapter.notifyItemInserted(0);          // 通知适配器列表在第一项插入数据
                rv_dynamic.scrollToPosition(0);          // 让循环视图滚动到第一项所在的位置
            }
        };
    }
```

运行测试App，打开公众号列表界面，在屏幕顶端下拉再松手，此时页面上方弹出转圈提示正在刷新，如图9-30所示；稍等片刻结束刷新，此时列表顶端增加了一条新消息，如图9-31所示。



图9-30 刷新中的消息列表



图9-31 刷新完成的消息列表

9.4.2 第二代翻页视图 ViewPager2

正如RecyclerView横空出世取代ListView和GridView那样，Android也推出了二代翻页视图ViewPager2，打算替换原来的翻页视图ViewPager。与ViewPager相比，ViewPager2支持更丰富的界面特效，包括但不限于下列几点：

（1）不但支持水平方向翻页，还支持垂直方向翻页。

（2）支持 RecyclerView.Adapter，也允许调用适配器对象的 notifyItem***方法，从而动态刷新某个页面项。

（3）除了展示当前页，也支持展示左、右两页的部分区域。

（4）支持在翻页过程中展示自定义的切换动画。

虽然ViewPager2增加了这么棒的功能，但它用起来很简单，掌握下面几个方法就够了：

- setAdapter：**设置二代翻页视图的页面适配器。**
- setOrientation：**设置二代翻页视图的翻页方向。其中**ORIENTATION_HORIZONTAL**表示水平方向，**ORIENTATION_VERTICAL**表示垂直方向。**
- setPageTransformer：**设置二代翻页视图的页面转换器，以便展示切换动画。**

接下来利用循环适配器搭档二代翻页视图，演示看看ViewPager2的界面效果。注意，因为RecyclerView与ViewPager2拥有各自的依赖库，所以需要修改模块的build.gradle，在dependencies节点内部补充以下两行依赖配置：

```
implementation 'androidx.recyclerview:recyclerview:1.1.0'
implementation 'androidx.viewpager2:viewpager2:1.0.0'
```

接着新建一个活动页面，往该页面的XML文件中添加如下所示的ViewPager2标签：

```
(完整代码见chapter09\src\main\res\layout\activity view pager2 recycler.xml)
<androidx.viewpager2.widget.ViewPager2
    android:id="@+id/vp2_content"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1" />
```

由于ViewPager2仍然需要适配器，因此首先编写每个页面项的布局文件，下面便是一个页面项的XML例子，页面上方是图像视图，下方是文本视图。

```
    (完整代码见chapter09\src\main\res\layout\item_mobile.xml)
    <!-- ViewPager2要求每页的宽高都必须是match parent -->
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <ImageView
            android:id="@+id/iv_pic"
            android:layout_width="match_parent"
            android:layout_height="360dp"
            android:scaleType="fitCenter" />
        <TextView
            android:id="@+id/tv_desc"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
```

然后给上面的页面项补充对应的循环适配器代码，在适配器的构造方法中传入一个商品列表，再展示每个商品的图片与文字描述。循环适配器的代码示例如下：

```
    (完整代码见chapter09\src\main\java\com\example\chapter09\adapter\
MobileRecyclerAdapter.java)
    public class MobileRecyclerAdapter extends RecyclerView.Adapter
<RecyclerView.ViewHolder> {
        private Context mContext;           // 声明一个上下文对象
        // 声明一个商品列表
        private List<GoodsInfo> mGoodsList = new ArrayList<GoodsInfo>();
        public MobileRecyclerAdapter(Context context, List<GoodsInfo> goodsList) {
            mContext = context;
            mGoodsList = goodsList;
        }
        // 创建列表项的视图持有者
        public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup vg, int viewType) {
            // 根据布局文件item_mobile.xml生成视图对象
            View v = LayoutInflater.from(mContext).inflate(R.layout.item_mobile, vg, false);
            return new ItemHolder(v);
        }
        // 绑定列表项的视图持有者
        public void onBindViewHolder(RecyclerView.ViewHolder vh, final int position) {
            ItemHolder holder = (ItemHolder) vh;
            holder.iv_pic.setImageResource(mGoodsList.get(position).pic);
            holder.tv_desc.setText(mGoodsList.get(position).desc);
        }
        // 定义列表项的视图持有者
        public class ItemHolder extends RecyclerView.ViewHolder {
            public ImageView iv_pic;            // 声明列表项图标的图像视图
            public TextView tv_desc;            // 声明列表项描述的文本视图
            public ItemHolder(View v) {
                super(v);
                iv_pic = v.findViewById(R.id.iv_pic);
                tv_desc = v.findViewById(R.id.tv_desc);
            }
        }
    }
```

回到测试页面的Java代码，把二代翻页视图的排列方向设为水平方向，并将它的适配器设置为上述的循环适配器。只要以下几行代码就搞定了：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\ViewPager2RecyclerActivity.java)
// 从布局文件中获取名为vp2 content的二代翻页视图
ViewPager2 vp2 content = findViewById(R.id.vp2 content);
// 设置二代翻页视图的排列方向为水平方向
vp2_content.setOrientation(ViewPager2.ORIENTATION_HORIZONTAL);
// 构建一个商品信息列表的循环适配器
MobileRecyclerAdapter adapter = new MobileRecyclerAdapter(
                                this, GoodsInfo.getDefaultList());
vp2_content.setAdapter(adapter);  // 设置二代翻页视图的适配器
```

运行测试App，观察二代翻页视图的展示效果，其中水平方向的翻页过程如图9-32所示；如果把翻页方向改为垂直方向，那么翻页之时的界面如图9-33所示。

以上的效果图看起来仅仅多了垂直翻页，稍等片刻给它加上其他特效。先在测试页面的活动代码中补充下面几行代码：

```
// ViewPager2支持展示左、右两页的部分区域
RecyclerView cv_content = (RecyclerView) vp2_content.getChildAt(0);
cv_content.setPadding(Utils.dip2px(this, 60), 0, Utils.dip2px(this, 60), 0);
cv_content.setClipToPadding(false);  // false表示不裁剪下级视图
```

重新运行测试App，此时页面效果如图9-34所示，可见除了显示当前商品之外，左、右两页也呈现了边缘区域。



图9-32 水平方向的二代翻页视图



图9-33 垂直方向的二代翻页视图



图9-34 左右两边呈现边缘的二代翻页视图

撤销刚加的边缘特效代码，再给测试页面的活动代码中补充下面几行代码：

```
// ViewPager2支持在翻页时展示切换动画，通过页面转换器计算切换动画的各项参数
ViewPager2.PageTransformer animator = new ViewPager2.PageTransformer() {
    @Override
    public void transformPage(@NonNull View page, float position) {
        page.setRotation(position * 360);  // 设置页面的旋转角度
    }
};
vp2_content.setPageTransformer(animator);// 设置二代翻页视图的页面转换器
```

重新运行测试App，此时翻页过程如图9-35和图9-36所示，其中图9-35为开始翻页不久的界面效果，图9-36为翻页即将结束的界面效果，从中可见翻页时展示了旋转动画。



图9-35　开始翻页不久的二代翻页视图



图9-36　翻页即将结束的二代翻页视图

9.4.3　给 ViewPager2 集成标签布局

ViewPager2 不仅支持循环适配器，同样支持翻页适配器，还是新的哦！原先 ViewPager 采用的翻页适配器叫作 FragmentPagerAdapter，而 ViewPager2 采用了 FragmentStateAdapter，看起来仅仅差了个“Pager”，实际上差得远了，因为新适配器 FragmentStateAdapter 继承了循环适配器 RecyclerView.Adapter。尽管它们都支持碎片 Fragment，但具体的方法就不一样了。新、旧适配器的实现方法对比见表 9-3。

表 9-3　新、旧翻页适配器的实现方法对比

旧翻页适配器的方法来自 FragmentPagerAdapter	新翻页适配器的方法来自 FragmentStateAdapter	说　明
FragmentPagerAdapter(FragmentManager fm)	FragmentStateAdapter(FragmentActivity fa)	构造方法
getCount	getItemCount	获取碎片个数
getItem	createFragment	创建指定位置的碎片
getPageTitle	无	获得指定位置的标题

比如下面是采用FragmentStateAdapter的翻页适配器代码例子：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\adapter\MobilePagerAdapter.java)
public class MobilePagerAdapter extends FragmentStateAdapter {
    // 声明一个商品列表
    private List<GoodsInfo> mGoodsList = new ArrayList<GoodsInfo>();
    // 碎片页适配器的构造方法，传入碎片管理器与商品信息列表
    public MobilePagerAdapter(FragmentActivity fa, List<GoodsInfo> goodsList) {
        super(fa);
        mGoodsList = goodsList;
    }
    // 创建指定位置的碎片Fragment
    @Override
    public Fragment createFragment(int position) {
        return MobileFragment.newInstance(position,mGoodsList.get(position).pic,
```

```
                                                mGoodsList.get(position).desc);
        }
        // 获取碎片Fragment的个数
        @Override
        public int getItemCount() {
            return mGoodsList.size();
        }
    }
```

新适配器集成的碎片MobileFragment代码参见chapter09\fragment\MobileFragment.java，该碎片的功能同上一小节，依旧传入商品列表，然后展示每个商品的图片与文字描述。运行测试App观察到的界面效果跟循环适配器差不多，因为展示商品信息的场景比较简单，所以循环适配器和翻页适配器看不出区别。就实际开发而言，简单的业务场景适合采用循环适配器，复杂的业务场景适合采用翻页适配器。

ViewPager 有个标签栏搭档 PagerTabStrip，然而 ViewPager2 抛弃了 PagerTabStrip，直接跟 TabLayout 搭配了。可是在“9.2.3 标签布局 TabLayout”中，为了让 ViewPager 联动 TabLayout，着实费了不少功夫。先给 ViewPager 添加页面变更监听器，一旦监听到翻页事件就切换对应的标签；再给 TabLayout 注册标签选中监听器，一旦监听到标签事件就翻到对应的页面。现在有了 ViewPager2，搭配 TabLayout 便轻松多了，只要一行代码即可绑定 ViewPager2 与 TabLayout。下面是将二者联结起来的操作步骤。

01 创建测试页面，并往页面的XML文件先后加入TabLayout标签和ViewPager2标签，具体代码如下：

```
(完整代码见chapter09\src\main\res\layout\activity_view_pager2_fragment.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <!-- 标签布局TabLayout节点需要使用完整路径 -->
    <com.google.android.material.tabs.TabLayout
        android:id="@+id/tab_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    <!-- 二代翻页视图ViewPager2节点也需要使用完整路径 -->
    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/vp2_content"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />
</LinearLayout>
```

02 打开该页面的Java代码，分别获取TabLayout和ViewPager2的视图对象，再利用TabLayoutMediator把标签布局跟翻页视图连为一体，完整代码示例如下：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\ViewPager2FragmentActivity.java)
public class ViewPager2FragmentActivity extends AppCompatActivity {
    // 声明一个商品信息列表
    private List<GoodsInfo> mGoodsList = GoodsInfo.getDefaultList();
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_view_pager2_fragment);
        // 从布局文件中获取名为tab_title的标签布局
        TabLayout tab_title = findViewById(R.id.tab_title);
```

```
        // 从布局文件中获取名为vp2_content的二代翻页视图
        ViewPager2 vp2 content = findViewById(R.id.vp2 content);
        // 构建一个商品信息的翻页适配器
        MobilePagerAdapter adapter = new MobilePagerAdapter(this, mGoodsList);
        vp2_content.setAdapter(adapter);                    // 设置二代翻页视图的适配器
        // 把标签布局跟翻页视图通过指定策略连为一体，二者在页面切换时一起联动
        new TabLayoutMediator(tab title, vp2 content, new TabConfigurationStrategy() {
            @Override
            public void onConfigureTab(TabLayout.Tab tab, int position) {
                tab.setText(mGoodsList.get(position).name);      //设置每页的标签文字
            }
        }).attach();
    }
}
```

重新运行测试App，初始的演示页面如图9-37所示；接着点击上方标签栏的第二个标签，此时页面下方翻到了第二页商品，如图9-38所示。

然后手指在商品处向左滑动，此时翻到了第三页商品，同时标签栏也切换到了第三个标签，如图 9-39 所示。通过点击标签与滑动翻页的结果对比，从而验证了标签布局与翻页视图的确绑定到了一块。



图9-37 集成标签布局的二代翻页视图



图9-38 通过点击标签切换的二代翻页视图



图9-39 通过滑动切换的二代翻页视图

9.5 实战项目：电商首页

各家电商App的首页都是动感十足，页面元素丰富，令人眼花缭乱，其中运用了Android的多种组合控件，可谓App界面开发的集大成之作。到目前为止，本章的知识点已经涵盖了电商首页的大部分技术，所以仿照着做个山寨的App首页也不是什么难事，接下来好好分析一下如何实现电商首页。

9.5.1 需求描述

首先看看大家熟悉的京东首页长什么模样，如图9-40所示。是不是很眼熟呢？其实该界面几乎是各电商App首页的通用模板。除了京东之外，还有淘宝、拼多多、当当、苏宁易购、美团等，这

些电商App的首页大同小异，所以只要吃透了京东首页采用的App技术，其他电商App也能依葫芦画瓢。

因为本节的实战项目只是模仿京东首页，不是完全一模一样，所以页面只要大致相似就行。图9-41和图9-42是两张山寨后的页面效果，其中图9-41为首页页面的效果图，图9-42为分类页面的效果图。两个页面均通过底部的标签栏切换，顶部又各有标题栏，其中首页上方拥有轮播着的广告条，以及布局多样的栏目列表和商品列表，至于分类页面的商品列表，则高低不一呈现瀑布流效果。



图 9-40　手机京东的 App 首页



图 9-41　仿电商 App 的首页页面



图 9-42　仿电商 App 的分类页面

9.5.2　界面设计

认真观察上一小节的两张效果图，找找看它们分别运用了本章的哪些知识点？仔细琢磨首页与分类页面，基本由本章介绍的各种组合控件拼接而成，分解起来也不难。

- **底部标签栏：界面底部的一排标签按钮，用于控制切换到哪个页面，标签栏既可基于BottomNavigationView实现，也可结合RadioGroup与ViewPager自定义实现。**
- **广告条：页面上方的广告条有轮播功能，图片底部还有指示器，这个广告轮播控件参见第8章的实战项目。**
- **循环视图RecyclerView的网格布局：广告条下方的两排市场图标是标准的网格布局，再下面的推荐商品是合并单元后的网格布局。**
- **工具栏Toolbar：页面顶部的标题栏采用了工具栏Toolbar。**
- **标签布局TabLayout：分类页面顶部的“服装”和“电器”标签用到了标签布局。**
- **第二代翻页视图ViewPager2：“服装”和“电器”标签对应的两个碎片通过二代翻页视图组装起来。**
- **循环视图RecyclerView的瀑布流布局：电器商品的交错展示运用了瀑布流网格布局。**
- **下拉刷新布局SwipeRefreshLayout：在电器商品页，下拉手势会触发商品列表的刷新动作。**

另外，这个电商首页项目还使用了前几章学过的控件，包括翻页视图ViewPager、碎片Fragment等，正好一起复习。除了首页和分类页，购物车页面的具体实现参见第6章的实战项目，有兴趣的读者可以将其整合进来，形成一个电商App的演示程序。

9.5.3 关键代码

尽管电商首页的控件技术在之前章节已经提到，可是多种控件组合起来仍需十分小心，因为复杂的界面布局时常产生意想不到的结果，下面列举了几处需要额外注意的关键代码。

1. 在ScrollView内部添加RecyclerView

由于电商首页往往很长，手机屏幕无法显示首页的全部内容，必须经常上拉页面，才能看到首页下方的商品界面，因此首页XML通常在外层加个ScrollView，在ScrollView内部再添加各种栏目布局。这些栏目布局采用的控件种类各不相同，既有自定义的广告条，又有样式丰富的循环视图。然而循环视图以列表形式展现，如果它的列表项越过了屏幕底部，那么循环视图将只显示第一行，后面部分要向上滑动才会出现。此时只有RecyclerView的列表项在滑动，而非整个ScrollView在滑动。若要解决RecyclerView与ScrollView的滑动冲突问题，得在RecyclerView外面嵌套一层RelativeLayout，举例如下：

```
(完整代码见chapter09\src\main\res\layout\fragment_department_home.xml)
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:descendantFocusability="blocksDescendants">
    <androidx.recyclerview.widget.RecyclerView
       android:id="@+id/rv_combine"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</RelativeLayout>
```

2. 关于ViewPager+Fragment的多重嵌套

通过翻页适配器组装各碎片时，翻页适配器的构造方法需要传入FragmentManager实例，此时在外部创建适配器对象一般会传入getSupportFragmentManager()，比如下面的代码：

```
(完整代码见chapter09\src\main\java\com\example\chapter09\DepartmentStoreActivity.java)
ViewPager vp_content = findViewById(R.id.vp_content);
// 构建一个翻页适配器
DepartmentPagerAdapter adapter = new DepartmentPagerAdapter(getSupportFragmentManager());
vp_content.setAdapter(adapter);  // 设置翻页视图的适配器
```

可是一旦出现ViewPager+Fragment嵌套的情况，构造参数就不能这么传了。譬如电商首页采取RadioGroup与ViewPager组合的话，每个页面都是一个Fragment，如果其中一个页面还想引入ViewPager+Fragment，假设分类页面的TabLayout搭配ViewPager，那么下级适配器就不能再传getSupportFragmentManager()了，而要传getChildFragmentManager()，否则编译会报错，因为getSupportFragmentManager方法来自Activity，而getChildFragmentManager方法来自Fragment。下面是在Fragment内部使用翻页适配器的代码例子：

```
// 构建一个分类信息的翻页适配器
// 注意ViewPager+Fragment嵌套时要传入getChildFragmentManager
ClassPagerAdapter adapter = new ClassPagerAdapter(
                    getChildFragmentManager(), mTitleList);
vp_content.setAdapter(adapter);  // 设置翻页视图的适配器
```

为了避免ViewPager+Fragment嵌套导致的适配器构造问题，ViewPager2御用的FragmentStateAdapter在构造方法中使用FragmentActivity实例，因为它是个活动实例，不会依赖于碎片，所以无所谓是

否嵌套，只要给FragmentStateAdapter传入当前页面的活动实例即可。

3. 电商首页项目的源码之间的关系

接下来简单介绍一下本章模块源码中，与电商App有关的主要代码之间的关系。

（1）DepartmentStoreActivity.java：这是电商App首页的入口代码，采用RadioGroup+ViewPager方式的底部标签栏，挂载了“首页”“分类”和“购物车”3个标签及其对应的3个碎片。

（2）DepartmentPagerAdapter.java：这是电商首页集成3个碎片页的翻页适配器代码。

（3）DepartmentHomeFragment.java：这是“首页”标签对应的碎片代码，从上到下依次分布着工具栏、广告条、市场网格列表、猜你喜欢的合并网格列表、手机网格列表，主要运用了Toolbar、BannerPager、RecyclerView等组合控件。

（4）DepartmentClassFragment.java：这是“分类”标签对应的碎片代码，该页面顶端的工具栏通过集成标签布局TabLayout，加载了服装与电器两个瀑布流列表碎片，从而形成Fragment再次嵌套Fragment的页面结构。

（5）DepartmentCartFragment.java：这是“购物车”标签对应的碎片代码，具体实现可参考第6章的实战项目“购物车”。

9.6 小　　结

本章主要介绍了App开发的组合控件的相关知识，包括：底部标签栏（利用BottomNavigationView实现底部标签栏、自定义标签按钮、结合RadioGroup和ViewPager自定义底部标签栏）、顶部导航栏（工具栏Toolbar、溢出菜单OverflowMenu、标签布局TabLayout）、增强型列表（循环视图RecyclerView、布局管理器LayoutManager、动态更新循环视图）、升级版翻页（下拉刷新布局SwipeRefreshLayout、第二代翻页视图ViewPager2、给ViewPager2集成标签栏）。最后设计了一个实战项目“电商首页”，在该项目的App编码中用到了前面介绍的大部分控件，从而加深了对所学知识的理解。

通过本章的学习，读者应该能够掌握以下4种开发技能：

（1）学会使用底部标签栏及其切换操作。

（2）学会使用顶部导航栏及其导航操作。

（3）学会使用循环视图实现3种界面布局（线性布局、网格布局、瀑布流网格布局）。

（4）学会使用第二代翻页视图及其常见搭配。

9.7 动 手 练 习

请上机实验本章的电商首页项目，要求实现底部标签栏、顶部导航栏、瀑布流布局的商品列表等界面效果，并支持通过第二代翻页视图切换标签布局。

第 10 章

打造安装包

本章介绍应用安装包的基本制作规范，主要包括：如何导出既美观又精简的APK文件、如何按照上线规范调整App的相关设置、如何对APK文件进行安全加固以防止安装包被破解。

10.1 应 用 打 包

本节介绍 APK 安装包的打包过程，包括：如何利用 Android Studio 导出 APK 格式的安装包、如何利用 Android Studio 制作 App 的个性化图标、如何通过各种瘦身手段压缩 APK 文件的大小。

10.1.1 导出 APK 安装包

前面章节在运行App的时候，都是先由数据线连接手机和计算机，再通过Android Studio的Run菜单把App安装到手机上。这种方式只能在自己手机上调试应用，如果想在别人手机上安装应用，就得把App打包成APK文件（该文件就是App的安装包），然后把APK传给其他人安装。

下面是使用Android Studio打包APK的具体步骤说明：

01 依次选择菜单Build→Generate Signed Bundle / APK...，弹出对话框，如图10-1所示。

02 选中该窗口左下方的APK选项，再单击Next按钮，进入APK签名对话框，如图10-2所示。

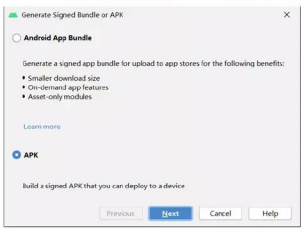


图10-1 生成安装包的对话框

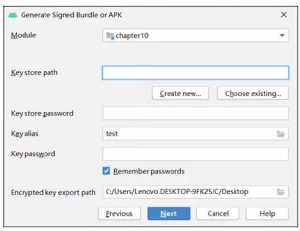


图10-2 APK签名对话框

03 在该窗口选择待打包的模块名（如chapter10），以及密钥文件的路径，如果原来有密钥文件，

就单击Choose existing...按钮，在弹出的文件对话框中选择密钥文件。如果首次打包没有密钥文件，就单击Create new...按钮，弹出密钥创建对话框，如图10-3所示。

04 单击该对话框右上角的按钮，弹出文件对话框，如图10-4所示，在此可选择密钥文件的保存路径。

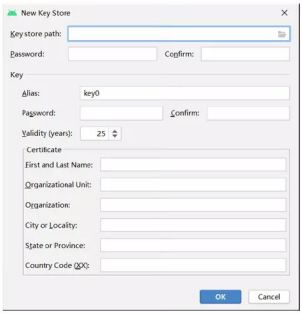


图10-3 密钥文件的创建对话框

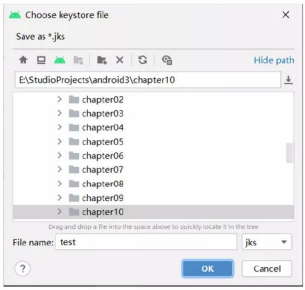


图10-4 密钥文件的文件对话框

05 在文件对话框中选择文件保存路径，并在下方的File name输入框中填写密钥文件的名称，然后单击OK按钮回到密钥创建对话框。在该对话框依次填写密码（Password）、确认密码（Confirm）、别名（Alias）、别名密码（Password）、别名的确认密码（Confirm），修改密钥文件的有效期限（Validity）。对话框下半部分的输入框只有姓名（First and Last Name）是必填的，填完后的对话框如图10-5所示。

06 单击OK按钮回到APK签名对话框，此时Android Studio自动把密码和别名都填上了，如图10-6所示。如果一开始选择已存在的密钥文件，这里就要手工输入密码和别名。

07 单击Next按钮进入下一个打包对话框，如图10-7所示。对话框上方可选择APK文件的保存路径，对话框中部可选择编译变量（Build Variants），如果是调试用，则编译变量选择debug；如果是发布用，则编译变量选择release。最后单击Finish按钮，等待Android Studio生成APK安装包。

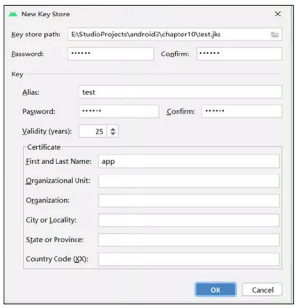


图 10-5 填写完成的密钥创建对话框

若无编译问题，片刻之后会在APK保存路径下看到release目录，打开该目录找到名为chapter10-release.apk的安装包文件。把该文件通过QQ或者微信传给其他人，对方在手机上收到APK文件，点击APK即可安装应用。

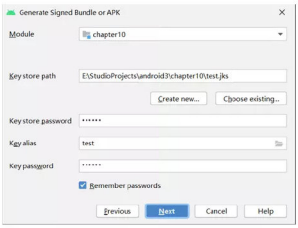


图10-6　填上签名信息的签名对话框

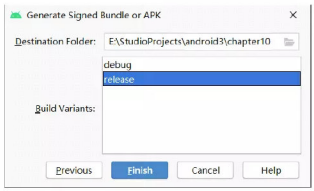


图10-7　导出APK的打包对话框

如果APK文件安装失败，则可能是以下原因导致的：

（1）App只能升级不能降级，假如安装包的版本号小于已安装App的版本号，就无法正常安装。版本号在build.gradle中的versionCode节点配置。

（2）倘若新、旧App的签名不一致，也会造成安装失败。比如该手机之前安装了debug类型的App，现在又要安装release类型的版本，就会出现签名冲突。

10.1.2　制作 App 图标

新建一个App工程，默认的应用图标都是机器人，如果要发布正规的App，肯定得更换醒目的专享图标。可是res目录下有好几种分辨率的mipmap-***目录，每种分辨率又有圆角矩形和圆形两类图标，加起来要做十几个图标，倘若每个图标都手工制作，实在要累得够呛。幸好Android Studio早早提供了专门的图标制作插件，只要简简单单几个步骤，即可自动生成所有规格的应用图标。该插件的具体使用步骤如下。

01 右击项目结构图的模块名称，在右键菜单中依次选择菜单New→Image Asset，弹出如图10-8所示的图标制作对话框。

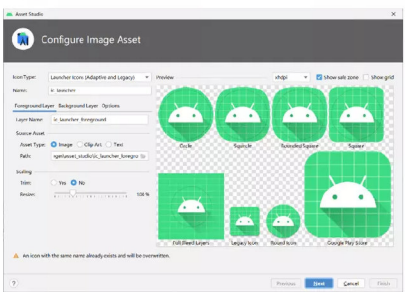


图 10-8　Android Studio 的图标制作对话框

02 图10-8所示的对话框左侧是图标的配置选项，右侧是各规格图标的展示效果。在对话框左侧中间找到Path区域，单击路径输入框右边的文件夹图标，在弹出的文件窗口中选择新图标的素材图片，再回到图标制作窗口，此时该对话框的界面如图10-9所示。

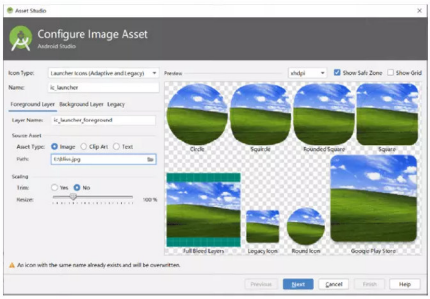


图 10-9 选择新图片后的图标制作对话框

03 由图10-9可见，对话框右侧的展示区域一下子全部换成了新的图标，完全自动加工好了。接着单击窗口下方的Next按钮，跳转到如图10-10所示的下一页对话框。

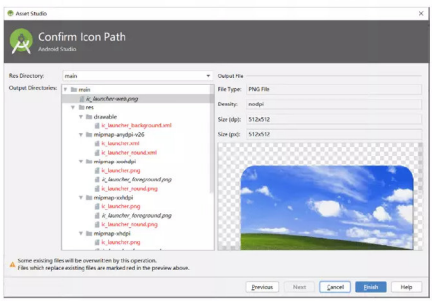


图 10-10 图标制作的下一页对话框

04 单击下一页窗口中的Finish按钮，结束图标制作操作，然后在mipmap-***目录下就能看到各种规格的新图标了。

10.1.3 给 APK 瘦身

App 不但要求功能完善，其他方面也得综合考虑，比如 APK 安装包的文件大小就是很重要的参考因素。具备同样功能的两个安装包，一个很大很占用空间，另一个较小不怎么占空间，用户的

选择结果自然不言而喻。如何压缩打包后的 APK 文件大小，也就是所谓的给 APK 瘦身，这涉及很多技术手段，最常用的主要有 3 种：去除冗余功能、精简无用资源、压缩图片大小。分别介绍如下：

1. 去除冗余功能

每当开发者创建新的Android项目，打开模块的AndroidManifest.xml，看到默认的application节点是下面这样的：

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
```

注意application节点有两个属性allowBackup和supportsRtl，且都被设置为true，它俩到底是干什么用的呢？

首先看 allowBackup，该属性若设置为 true，则允许用户备份 APK 安装包和应用数据，以便在刷机或者数据丢失后恢复应用。这里其实隐含着高危漏洞，因为备份后的应用数据可能被人复制到其他设备，如此一来用户的隐私就会泄露出去，账号密码、聊天记录等均可遭窃。所以还是赶紧关闭这个鸡肋功能，把 allowBackup 属性值由默认的 true 改为 false。

然后看 supportsRtl，该属性名称当中的 Rtl 为“Right-to-Left”（从右到左）的缩写，像中东的阿拉伯语、希伯来文等都是从右到左书写，supportsRtl 属性值为 true 时表示支持这种从右向左的文字系统。可是常用的中文、英文等都是从左往右书写，根本用不着从右到左的倒排功能，因此若无特殊情况可把 supportsRtl 属性值由默认的 true 改为 false。

关闭备份与倒排功能之后，application节点变成了下面这样：

```
<application
    android:allowBackup="false"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app name"
    android:roundIcon="@mipmap/ic launcher round"
    android:supportsRtl="false"
    android:theme="@style/AppTheme">
```

2. 精简无用资源

同样打开新项目中模块级别的build.gradle，发现buildTypes节点是下面这样的：

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize. txt'),
'proguard-rules.pro'

    }
}
```

可见有个minifyEnabled属性，默认值为false，该属性的字面意思为是否启用最小化，如果将它设置为true，则Android Studio在打包APK时会进行以下代码处理：

（1）压缩代码，移除各种无用的实体，包括类、接口、方法、属性、临时变量等。

（2）混淆代码，把类名、属性名、方法名、实例名、变量名替换为简短且无意义的名称，例如 Student 类的名称可能改为 a，方法 getName 的名称可能改为 b 等。

App的Java代码经过压缩和混淆之后，打包生成的APK文件会随之变小。除了代码之外，应用项目还包括各种资源文件，若想移除无用的资源文件（包括XML布局和图片），就要引入新属性shrinkResources，并将该属性值设置为true，这样Android Studio在打包APK时会自动移除无用的资源文件。同时开启代码压缩和资源压缩的buildTypes节点示例如下：

```
    buildTypes {
        release {
            minifyEnabled true
            shrinkResources true
            proguardFiles getDefaultProguardFile('proguard-android-optimize. txt'),
'proguard-rules.pro'
        }
    }
```

3. 压缩图片大小

由于手机屏幕的尺寸有限，原始质量的高清图片与有损压缩后的图片在视觉上没有太大差别，因此适当压缩图片质量也是减小 APK 体积的一个重要途径。App 传统的资源图片主要有 JPG 和 PNG 两种格式，对于 JPG 图片来说，利用看图软件 ACDSee 即可快速压缩图片大小，先使用 ACDSee 打开 JPG 文件，然后依次选择菜单“文件”→“另存为”，弹出如图 10-11 所示的“图像另存为”对话框。

单击“图像另存为”对话框右下角的“选项”按钮，弹出如图 10-12 所示的“JPEG 选项”对话框，把窗口上方“图像质量”区域的拖动条往左拖到 60 处，表示有损压缩保持 60%的图像质量。单击窗口下方的“确定”按钮，回到前一步骤的“图像另存为”对话框，再单击对话框右侧的“保存”按钮，完成 JPG 图片的压缩操作。

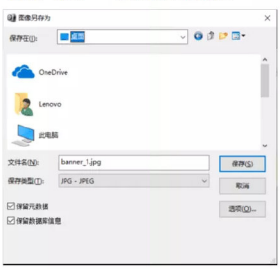


图10-11　ACDSee的“图像另存为”对话框

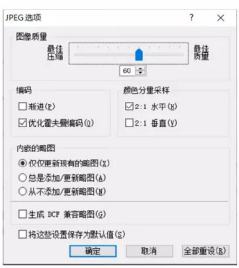


图10-12　“JPEG选项”对话框

对于 PNG 图片来说，利用 PhotoShop 即可快速压缩图片大小，先使用 PhotoShop 打开 PNG 文件，然后依次选择菜单“文件”→“存储为 Web 所用格式”，弹出如图 10-13 所示的图片转换对话框。

单击对话框右上角的“预设”下拉框，并选中最后一项“PNG-8”，再单击对话框下方的“存储”按钮，完成 PNG 图片的压缩操作。

当然，专业的图像处理软件毕竟存在操作门槛，初学者不易掌握使用技巧。此时也可借助第三方的图像压缩网站，自动完成图片文件的压缩处理，就 PNG 格式而言，常见的压缩站点包括 https://tinypng.com/和 https://compresspng.com/zh/，感兴趣的读者不妨一试。

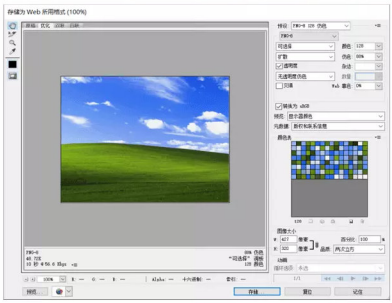


图 10-13　PhotoShop 的 PNG 图片转换对话框

10.2　规 范 处 理

本节介绍App上线前必做的准备工作，包括：如何正确设置App的版本编号和版本名称、如何把App从调试模式切换到发布模式、如何给多个渠道同时打包APK文件。

10.2.1　版本设置

每个App都有3个基础信息：第一个是App的图标，图标文件为res/mipmap-***目录下的ic_launcher.png；第二个是App的名称，名称文字保存在res/values/strings.xml的app_name当中；第三个是App的版本号，版本信息包括build.gradle的versionCode与versionName两个参数，其中versionCode为纯数字的版本编号，versionName为带点号的字符串，格式形如“数字.数字.数字”。

App图标和App名称都好理解，在手机桌面上也能看到App的图标和名称，那么为什么App还需要版本编号与版本名称这样的版本信息呢？这是因为App需要经常升级，但不允许App降级，也就是说，一旦安装了某个版本的App，那么之后只能安装版本更新的同名App，不能安装版本更旧的同名App。这种只能升级不能降级的判断，就依赖于每个APK文件设定的版本号versionCode，versionCode的数值越大，表示该安装包的版本越高；versionCode的数值越小，表示该安装包的版本越低。依据当前App的版本号与待安装APK的版本号，系统方能比较得知是否允许升级App。

至于版本名称versionName，则用来标识每次App升级的改动程度，按照通常的版本名称格式“数字.数字.数字”，第一个数字为大版本号，每当有页面改版或代码重构等重大升级时，大版本号要加1，后面两个数字清零；第二个数字为中版本号，每当要更新局部页面或添加新功能时，中版本号加1，第三个数字清零；第三个数字为小版本号，每当有界面微调或问题修复时，小版本号加1。

每次App升级重新导出APK的时候，versionCode与versionName都要一起更改，不能只改其中一个。并且升级后的versionCode与versionName只能比原来大，不能比原来小。如果没有按照规范修改版本号，就会产生以下问题：

（1）版本号比已安装的版本号小，在安装时系统直接提示失败，因为App只能做升级操作，不能做降级操作。

（2）在升级系统应用（手机厂商内置的应用，非普通应用）时，如果只修改versionName，没修改versionCode，重启手机后会发现更新丢失，该应用被还原到升级前的版本。这是因为：对于系统应用，Android会检查versionCode的数值，如果versionCode不大于当前已安装的版本号，本次更新就被忽略了。

除了系统要求检查应用的基础信息，App有时也需要获取自身信息，比如应用图标可从资源图片获取，应用名称可调用getString方法获取。其他像应用包名、应用版本等信息，可从编译配置工具BuildConfig获取，该类提供的几个配置属性说明如下：

- APPLICATION_ID：**应用包名**。
- BUILD_TYPE：**编译类型。为debug表示这是调试包，为release表示这是发布包**。
- VERSION_CODE：**应用的版本编号**。
- VERSION_NAME：**应用的版本名称**。

下面是获取App基础信息的代码例子：

```
(完整代码见chapter10\src\main\java\com\example\chapter10\AppVersionActivity.java)
public class AppVersionActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity app version);
        ImageView iv icon = findViewById(R.id.iv icon);
        // 应用图标取自ic launcher
        iv icon.setImageResource(R.mipmap.ic launcher);
        TextView tv_desc = findViewById(R.id.tv_desc);
        // 应用名称取自app_name，应用包名、版本号、版本名称均来自BuildConfig
        String desc = String.format("App名称为：%s\nApp包名为：%s\n" +
                    "App版本号为：%d\nApp版本名称为：%s",
              getString(R.string.app name), BuildConfig.APPLICATION ID,
              BuildConfig.VERSION_CODE, BuildConfig.VERSION_NAME);
        tv_desc.setText(desc);
    }
}
```

运行测试App，看到App版本信息的获取页面如图10-14所示，可见分别展示了测试App的图标、名称、包名，以及版本编号和版本名称。

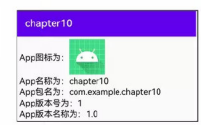


图 10-14　App 版本信息的获取页面

10.2.2　发布模式

为了编码调试方便，开发者经常在代码里添加日志，还在页面上弹出各种提示。这样固然有利于发现 bug、提高软件质量，不过调试信息过多往往容易泄露敏感信息，例如用户的账号密码、业务流程的逻辑等。从保密角度考虑，App 在上线前必须去掉多

余的调试信息，也就是生成发布模式的安装包，与之相对的是开发阶段的调试模式。

建立发布模式拥有下列两点优势：

（1）保护用户的敏感账户信息不被泄露。

（2）保护业务逻辑与流程处理的交互数据不被泄露。

发布模式与调试模式的安装包很好区分，通过菜单Generate Signed Bundle / APK...导出安装包的打包对话框，在Build Variants一栏即可选择安装包类型。选中release时表示生成发布模式的安装包，如图10-15所示；选中debug时表示生成调试模式的安装包，如图10-16所示。

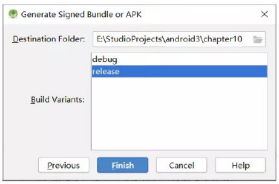


图10-15　导出APK时选择发布模式

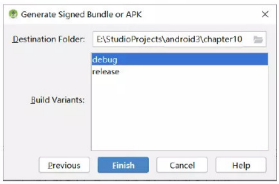


图10-16　导出APK时选择调试模式

发布模式不是直接删掉调试代码，而是通过某个开关控制是否显示调试信息，因为App后续还得修改、更新、重新发布，这个迭代过程要不断调试，从而实现并验证新功能。App代码可通过BuildConfig.DEBUG判断当前是发布模式还是调试模式，BuildConfig.DEBUG值为false表示处于发布模式，为true表示处于调试模式。于是利用BuildConfig.DEBUG能够控制是否打开日志，在开发阶段导出调试包，在上架阶段导出发布包，这样日志只会在调试包中打印，不会在发布包中打印。

控制调试信息的工具类主要有两种，分别对Log工具和Toast工具加以封装，说明如下：

1. 日志Log

Log工具用于打印调试日志。在App运行过程中，日志信息会输出到Logcat窗口。因为最终用户不关心App日志，所以除非特殊情况，发布上线的App应屏蔽所有日志信息。下面是封装了调试模式的Log工具代码：

```
(完整代码见chapter10\src\main\java\com\example\chapter10\util\LogUtil.java)
public class LogUtil {
    // 调试模式来自BuildConfig.DEBUG，false表示发布模式，true表示调试模式
    public static boolean isDebug = BuildConfig.DEBUG;

    public static void v(String tag, String msg) {
        if (isDebug) {
            Log.v(tag, msg);  // 打印冗余日志
        }
    }
    public static void d(String tag, String msg) {
        if (isDebug) {
            Log.d(tag, msg);  // 打印调试日志
        }
    }
    public static void i(String tag, String msg) {
        if (isDebug) {
            Log.i(tag, msg);  // 打印一般日志
```

```
        }
    }
    public static void w(String tag, String msg) {
        if (isDebug) {
            Log.w(tag, msg);  // 打印警告日志
        }
    }
    public static void e(String tag, String msg) {
        if (isDebug) {
            Log.e(tag, msg);  // 打印错误日志
        }
    }
}
```

2. 提示Toast

Toast工具在界面下方弹出小窗，给用户一两句话的提示，小窗短暂停留一会儿后消失。由于Toast窗口无交互动作，样式也基本固定，因此除了少数弹窗在发布时予以保留，其他弹窗都应在发布时屏蔽。下面是封装了调试模式的Toast工具代码：

```
(完整代码见chapter10\src\main\java\com\example\chapter10\util\ToastUtil.java)
public class ToastUtil {
    // 调试模式来自BuildConfig.DEBUG，false表示发布模式，true表示调试模式
    public static boolean isDebug = BuildConfig.DEBUG;
    // 不管发布模式还是调试模式，都弹出提示文字
    public static void show(Context ctx, String desc) {
        Toast.makeText(ctx, desc, Toast.LENGTH_SHORT).show();
    }
    // 调试模式下弹出短暂提示
    public static void showShort(Context ctx, String desc) {
        if (isDebug) {
            Toast.makeText(ctx, desc, Toast.LENGTH_SHORT).show();
        }
    }
    // 调试模式下弹出长久提示
    public static void showLong(Context ctx, String desc) {
        if (isDebug) {
            Toast.makeText(ctx, desc, Toast.LENGTH_LONG).show();
        }
    }
}
```

除此以外，AndroidManifest.xml也要区分发布模式与调试模式。应用上架之后，若无特殊情况，开发者都不希望activity和service对外部应用开放，所以要在activity和service标签下分别添加属性android:exported="false"，表示该组件不允许对外开放。

10.2.3 多渠道打包

对于很多大型App来说，针对不同渠道进行精细化运营是必不可少的，并且客观上也要求对App分渠道管理。这里所谓的渠道，指的是提供App下载的各大应用商店，尤其是各大手机厂商预装的自家应用商店，包括华为、荣耀、小米、OPPO、vivio等品牌。根据不同渠道打造对应的App安装包，带来的好处包括但不限于以下几点：

（1）各厂商的底层系统有着不同的适配要求，需要分别加以定制。
（2）有助于统计各家渠道的App下载量、用户数量以及业务交易量。
（3）有助于对各家厂商的App用户分别开展精准营销活动。

那么应该如何对App分渠道打包呢？倘若每打一个安装包，就要手工改配置手工导出APK，无疑费时费力费神。其实略施小计，通过修改build.gradle，即可实现自动化分渠道打包的功能。

以常见的友盟统计为例，按照友盟SDK的打包规定，需要在AndroidManifest.xml里面补充如下的配置信息，表示定义一个名叫UMENG_CHANNEL的元数据变量：

```
<meta-data
    android:name="UMENG_CHANNEL"
    android:value="${UMENG_CHANNEL_VALUE}" />
```

有了这个元数据UMENG_CHANNEL，App代码才能获取具体的渠道字符串，再分支处理各个渠道的统计数据。

那么UMENG_CHANNEL的字符串值又该如何传达呢？此时要修改build.gradle，在该文件中编写多渠道打包配置，具体的修改步骤说明如下：

01 给android节点添加flavorDimensions与productFlavors配置，指定风味维度和产品风味，表示开启多渠道打包功能。详细的配置信息如下：

```
(完整代码见chapter10\build.gradle)
flavorDimensions "versionCode" // 指定风味维度。多渠道打包需要
// 指定产品风味（下面一共设定了五种风味：华为、荣耀、小米、OPPO、vivo）
productFlavors {
    huawei {}
    rongyao {}
    xiaomi {}
    oppo {}
    vivo {}
}
// 遍历每种风味，分别对AndroidManifest.xml的UMENG CHANNEL VALUE赋值
productFlavors.all {
    flavor -> flavor.manifestPlaceholders = [UMENG CHANNEL VALUE: name]
}
```

02 在buildTypes下面的release节点内部补充以下的配置内容，表示给每个渠道的安装包文件分别命名。

```
// 自定义输出配置
applicationVariants.all { variant ->
    variant.outputs.all { output ->
        def outputFile = output.outputFile
        if (outputFile != null && outputFile.name.endsWith('.apk')) {
            // 输出的apk名称类似test v1.0 huawei.apk
            def fileName = "test v${defaultConfig.versionName}
_${variant.productFlavors[0].name}.apk"
            outputFileName = fileName
        }
    }
}
```

然后依次选择菜单Build→Generate Signed Bundle / APK...，在最后一页的打包对话框中看到多个渠道的名称列表，如图10-17所示。

选中列表的所有渠道，注意勾选下方的V1和V2两个选项，单击窗口底部的Finish按钮开始打包。稍等片刻，即可在设置的目标路径中看到打包好的各渠道安装包了，安装包文件名形如test_v1.0_huawei.apk这样。

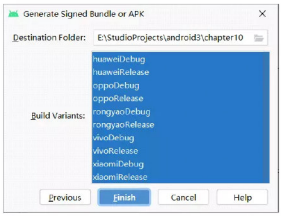


图 10-17　在 APK 打包对话框中看到多个渠道

10.3　安 全 加 固

本节介绍如何对APK安装包进行安全加固：首先通过反编译工具成功破解App源码，从而表明对APK实施安全防护的必要性；然后说明代码混淆的开关配置，并演示代码混淆如何加大源码破译的难度；最后描述怎样利用第三方加固网站对APK进行加固，以及如何对加固包进行重签名。

10.3.1　反编译

编译是把代码编译为程序，反编译是把程序破解为代码。

谁都不想自己的劳动成果被别人窃取，何况是辛辛苦苦敲出来的App代码，然而由于Java语言的特性，Java写的程序往往很容易被破解，只要获得App的安装包，就能通过反编译工具破解出该App的完整源码。开发者绞尽脑汁上架一个App，结果这个App却被他人从界面到代码都“山寨”了，那可真是欲哭无泪。为了说明代码安全的重要性，下面详细介绍反编译的完整过程，警醒开发者防火、防盗、防破解。

首先准备反编译的3个工具，分别是apktool、dex2jar、jd-gui，注意下载它们的最新版本。下面是这3个工具的简要说明。

- apktool：**对APK文件解包，主要用来解析**res**资源和**AndroidManifest.xml。
- dex2jar：**将**APK**包中的**classes.dex**转为**JAR**包，**JAR**包就是**Java**代码的编译文件**。
- jd-gui：**将**JAR**包反编译为**Java**源码**。

以Windows环境为例，下面是反编译APK的具体步骤。

01 依次选择开始菜单→Windows系统→命令提示符，打开命令行窗口，进入apktool所在的目录，运行命令“apktool.bat d -f解包后的保存目录名 待处理的APK文件名”，等待反编译过程，如图10-18所示。

反编译完毕，即可在apktool目录下看到破解目录。apktool的用途是解析出res资源，包括AndroidManifest.xml和res/layout、res/values、res/drawable等目录下的资源文件。

02 用压缩软件（如WinRAR）打开APK文件，发现APK安装包其实是一个压缩文件，使用WinRAR打开的APK文件的目录结构如图10-19所示。

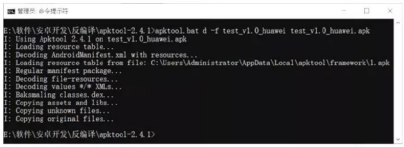


图 10-18　反编译工具 apktool 的运行截图



图 10-19　APK 解压后的内部目录结构

先从APK包中解压出classes.dex文件，再进入dex2jar所在的目录，运行命令“d2j-dex2jar.bat classes.dex”，等待转换过程，如图10-20所示。



图 10-20　转换工具 dex2jar 的运行截图

转换完毕，即可在dex2jar目录下看到新文件classes_dex2jar.jar，该JAR包即为Java源码的编译文件。

03 双击打开jd-gui.exe，用鼠标把第二步生成的classes_dex2jar.jar拖到jd-gui界面中，程序就会自动将JAR包反编译为Java源码，反编译后的Java源码目录结构如图10-21所示。

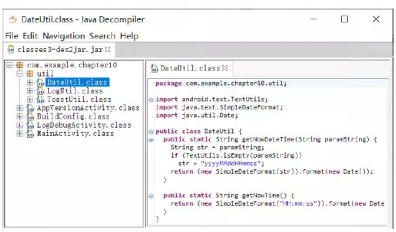


图 10-21　反编译后的 Java 源码目录结构

在jd-gui界面依次选择菜单File→Save All Sources，输入保存路径再单击保存按钮，即可在指定目录下生成ZIP文件，解压ZIP文件就能看到反编译后的全部Java代码了。

由此可见，反编译过程不但破解了Java代码，而且res目录下的资源文件也被一起破解了，所以，如果App不采取一些保护措施，整个工程源码就会暴露在大庭广众之下。

10.3.2　代码混淆

前面讲到反编译能够破解App的工程源码，因此有必要对App源码采取防护措施，代码混淆就是保护代码安全的措施之一。Android Studio已经自带了代码混淆器ProGuard，它的用途主要有下列两点：

（1）压缩APK包的大小，删除无用代码，并简化部分类名和方法名。
（2）加大破解源码的难度，部分类名和方法名被重命名使得程序逻辑变得难以理解。

代码混淆的配置文件其实一直都存在，每次在Android Studio新建一个模块，该模块的根目录下会自动生成文件proguard-rules.pro。打开build.gradle，在android→buildTypes→release节点下可以看到两行编译配置，其中便用到了proguard-rules.pro：

```
    minifyEnabled false
    proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
```

由于Android Studio默认不做代码混淆，因此上面第一行的minifyEnabled为false，表示关闭混淆功能，要把该参数改为true才能开启混淆功能。上面第二行指定proguard-rules.pro作为本模块的混淆规则文件，该文件保存着各种详细的代码混淆规则。

对于初学者来说，采用Android Studio默认的混淆规则即可，所以无须改动proguard-rules.pro，只要把build.gradle里的minifyEnabled改为true，Android Studio就会按照默认的混淆规则对App代码进行混淆处理。注意默认规则保存在proguard-android-optimize.txt中，该文件位于SDK安装目录的tools\proguard\proguard-android-optimize.txt。

经过代码混淆后重新生成APK安装包，再用反编译工具破解APK文件，反编译后的Java源码结构如图10-22所示。

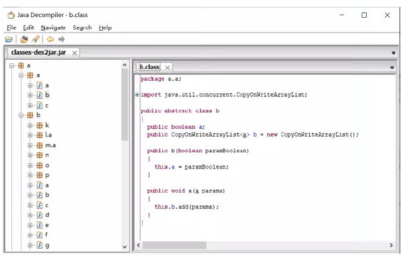


图 10-22　经过代码混淆再破解后的 Java 源码目录结构

从图中可以看到，混淆后的包名与类名都变成了a、b、c、d这样的名称，无疑加大了黑客理解源码的难度。试想当黑客面对这些天书般的a、b、c、d，还会绞尽脑汁地尝试破译吗？

10.3.3　第三方加固及重签名

App经过代码混淆后初步结束了裸奔的状态，但代码混淆只能加大源码破译的难度，并不能完全阻止被破解。除了代码破解外，App还存在其他安全风险，比如二次打包、篡改内存、漏洞暴露等情况。对于这些安全风险，Android Studio基本无能为力。因此，鉴于术业有专攻，不妨把APK文件交给专业网站进行加固处理。举个做得比较好的第三方加固例子——360加固保，其网址是http://jiagu.360.cn/。开发者要先在该网站注册新用户，然后进入“管理中心”→“应用安全”，打开的在线加固页面如图10-23所示。



图 10-23　360 加固保的在线加固页面

单击该页面的“上传应用”按钮，在新页面中选中本地的APK文件后，点击“确定”按钮，接着页面开始加固操作，如图10-24所示。



图 10-24　正在加固的应用页面

稍等片刻，加固完成的页面如图10-25所示。



图 10-25　加固完成的应用页面

可见应用右侧的加固状态为“加固成功”，单击文字链接或右边的下载图标，把加固好的安装包下载到本地，下载后的文件名形如chapter10-release_enc.apk。然后利用反编译工具尝试破解这个加固包，会发现该安装包变得无法破译。

不过加固后的APK破坏了原来的签名，也就无法在手机上安装，此时要对该文件进行重签名，才能成为合法的APK安装包。重签名可使用专门的签名软件，比如爱加密的APKSign，先下载该软件，解压后打开APKSign.exe，它的软件界面如图10-26所示。单击界面右上角的“浏览”按钮，选择待签名的APK（如chapter10-release_enc.apk），再选择签名文件的路径，依次输入密码、别名、别名

的密码、APK签名后的保存路径，输入各项信息的签名界面如图10-27所示，最后单击“开始签名”按钮开始签名操作。

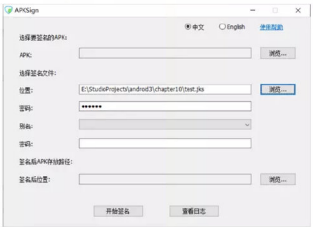


图10-26 爱加密的重签名工具界面

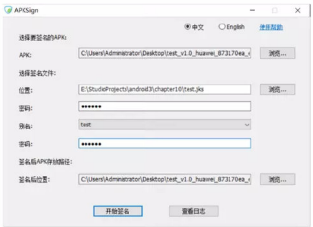


图10-27 填好信息的重签名工具界面

等待签名结束，即可在“签名后位置”一栏指定的路径下找到重签名的安装包文件chapter10-release_enc_signed.apk，该 APK 文件即可直接在手机上安装了。

10.4 小 结

本章主要介绍了应用安装包的相关制作规范，包括：应用打包（导出 APK 安装包、制作 App 图标、给 APK 瘦身）、规范处理（版本设置、发布模式、多渠道打包）、安全加固（反编译、代码混淆、第三方加固及重签名）。经过这一系列应用制作流程，完成了 App 从开发阶段的代码到用户手机应用的华丽转变，实现 App 从开发→调试→加固→发布的完整过程。

通过本章的学习，读者应该能够掌握以下3种开发技能：

（1）学会从 App 工程导出 APK 安装包。

（2）学会把 App 工程从调试模式转为发布模式。

（3）学会对 APK 文件进行安全加固和重签名。

10.5 动 手 练 习

请上机实验制作并导出测试应用的安装包文件，要求 App 图标醒目，APK 文件适当瘦身，采取发布模式，并且经过代码混淆、安全加固处理。