

Android 进阶之光

刘望舒 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

Android 进阶之光

刘望舒 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书是一本 Android 进阶类书籍，书中各知识点由浅入深、环环相扣，最终这些知识点形成了一个体系结构。本书共分为 11 章。第 1 章介绍 Android 5.0 到 Android 7.0 的新特性。第 2 章介绍 Material Design。第 3 章介绍 View 体系，包括 View 的事件分发、工作流程、自定义 View 等知识点。第 4 章介绍多线程的知识。第 5 章介绍网络编程与网络框架的知识。第 6 章介绍常用的设计模式。第 7 章介绍事件总线。第 8 到第 10 章介绍架构设计所需要的知识点。第 11 章简单介绍 Android 系统框架与 MediaPlayer 框架。

本书详细并深入讲解 Android 开发者必备的和前沿的知识，适合有一定开发基础的开发者阅读，这有助于他们提高技术水平；同时，本书系统的知识体系结构也令高级开发者从中获益良多。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Android 进阶之光 / 刘望舒著. —北京：电子工业出版社，2017.7
ISBN 978-7-121-31530-5

I. ①A… II. ①刘… 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（2017）第 108511 号

策划编辑：付 睿

责任编辑：李云静

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×1092 1/16 印张：30.75 字数：764 千字

版 次：2017 年 7 月第 1 版

印 次：2017 年 7 月第 1 次印刷

定 价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

前言

为什么写这本书

从 2008 年 Android 系统发布以来，Android 已经发展了 9 年。在此期间，Android 开发也相当火热。这时，大量人员涌入 Android 开发职场，并导致 Android 开发人才市场相对饱和。如此一来，很多 Android 开发者会发现工作越来越难找，企业对开发者的要求也越来越高，企业需求最多的不再是初中级别的 Android 工程师，而是 Android 高级工程师。但是，Android 高级工程师有限。有些人在从事了几年开发工作后，对很多技术的理解却仍停留在会用的阶段。他们对于原理不求甚解，这导致他们进入技术瓶颈期并长期无法得到提高。很多开发者为了突破技术瓶颈，看了大量的网络视频和博客。尽管如此，他们仍旧无法突破自身的技术瓶颈。其主要原因是，他们没有将学到的知识点形成体系化。因此，这就需要有一本成体系的进阶书来帮助这些开发人员成为 Android 高级工程师并突破自身的技术瓶颈。纵观市面上 Android 开发相关的书籍，其中大部分书籍是入门级别类图书，还有一部分系统源码、逆向分析和系统移植类图书，而关于应用开发进阶的书籍则少得可怜。本书正是一本成体系的应用开发进阶图书，书中所要传达的不仅仅是知识，其同时还会告诉读者以下几点。

1. 要关注 Android 新技术；
2. Java 基础和设计模式很重要；
3. 学习框架要深入其原理；
4. 要学习架构设计；
5. 要了解和學習系統源碼。

本书内容

本书共分为 11 章，各章内容如下。

- 第 1 章介绍 Android 5.0、Android 6.0、Android 7.0 的新特性，包括 Android 5.0 的 RecyclerView、Android 6.0 的运行时权限机制和 Android 7.0 的多窗口模式等知识点。
- 第 2 章介绍 Material Design 以及 Design Support Library 常用的控件，并给出实例将 Design Support Library 中的常用控件结合在一起使用。
- 第 3 章介绍 View 相关的进阶知识，包括 View 的滑动、View 的事件分发和 View 的工作流程。最后结合以上知识点来介绍自定义 View。
- 第 4 章介绍多线程编程，本章不仅包括基础的线程知识，还会介绍线程同步和线程池等进阶知识点，最后结合这些知识点来分析 Android 7.0 的 AsyncTask 的源码。
- 第 5 章介绍网络编程的基础知识以及常用的网络框架：Volley、OkHttp 和 Retrofit 的使用方法和原理分析。
- 第 6 章将设计模式进行分类，并介绍每个分类中常用的设计模式。
- 第 7 章介绍事件总线 EventBus 和 otto 的使用方法和原理。
- 第 8 章介绍函数响应式编程 RxJava 的使用方法，包括 RxJava 的基本使用、操作符、使用场景和源码分析等知识点。
- 第 9 章介绍注解的知识点和依赖注入框架 ButterKnife 和 Dagger2 的使用方法以及原理。
- 第 10 章介绍 Android 应用架构设计，包括 MVP 框架以及 MVP 结合 RxJava 和 Dagger2，还有 MVVM 框架相关的 Data Binding 支持库。
- 第 11 章主要是指引导读者进行 Android 系统源码阅读并带其入门，介绍 Android 系统框架、源码目录和阅读源码工具，并以分析 MediaPlayer 框架的源码作为示例。

本书特色

本书主要有以下特点。

- 本书整体结构由浅入深，从最简单的第 1 章到难一些的第 11 章，其难度是逐步加深的。
- 本书为了分析一些框架的原理，会介绍一些知识点做铺垫，比如为了更好地介绍依赖注入框架，需要首先了解注解相关的知识点。再比如要分析 AsyncTask 的源码，则需要了解线程池和阻塞队列等知识点。
- 本书的知识点环环相扣，比如要介绍 MVP 框架的设计，就需要先学习 Retrofit、RxJava 和 Dagger2 的相关知识点。
- 本书对于很多知识点都有很深入的讲解。其中，对于常用的框架，比如 OkHttp、Retrofit、EventBus 和 RxJava 等不只是讲解了如何使用，而且更加深入地介绍了其原理。
- 本书是目前市场上详细介绍有关 Android 新特性、Material Design、网络框架、事件总线、RxJava、依赖注入框架和应用架构设计的难得一见的图书。

读者对象

本书的章节设计是由浅入深的，适合 Android 初、中、高级工程师阅读。本书的定位是学习 Android 的第二本书，其阅读前提是要有一定的 Android 基础。

致谢

感谢本书的策划编辑付睿，她在 CSDN 博客中发现了，并积极推动本书的出版进度，这才使得本书得以及时出版。感谢本书的责任编辑李云静，她审稿时很细致，这使得书中的一些错误能被及时发现并改正。感谢我的父母在写书过程中对我的不断鼓励，这样我才得以全力以赴地投入编写工作。感谢所有关注我的朋友们，你们的鼓励和认可为我写博客以及写书带来了不可或缺的动力。

勘误与互动

本人虽已竭尽全力，但书中难免会有错误，欢迎大家向我反馈，我也会在独立博客和 CSDN 博客中定期发布本书的勘误信息。

本书互动地址

独立博客：<http://liuwangshu.cn>

CSDN 博客：<http://blog.csdn.net/itachi85>

GitHub：<https://github.com/henrymorgen>

微信公众号：刘望舒

QQ 交流群：499174415

源码下载

<https://github.com/henrymorgen/android-advanced-light>

<http://www.broadview.com.cn/31530>

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在[下载资源处](#)下载。
- **提交勘误**：您对书中内容的修改意见可在[提交勘误处](#)提交，若被采纳，将获赠博文视点

社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

- **交流互动：**在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31530>



目录

第 1 章 Android 新特性	1
1.1 Android 5.0 新特性	1
1.1.1 Android 5.0 主要新特性概述	1
1.1.2 替换 ListView 和 GridView 的 RecyclerView	3
1.1.3 卡片 CardView	13
1.1.4 3 种 Notification	18
1.1.5 Toolbar 与 Palette	23
1.2 Android 6.0 新特性	29
1.2.1 Android 6.0 主要新特性概述	29
1.2.2 运行时权限机制	30
1.3 Android 7.0 新特性	43
1.3.1 Android 7.0 主要新特性概述	43
1.3.2 多窗口模式	44
1.4 本章小结	47
第 2 章 Material Design	48
2.1 Material Design 概述	48
2.1.1 核心理念	48
2.1.2 材质与空间	49
2.1.3 动画	49
2.1.4 样式	50
2.1.5 图标	51
2.1.6 图像	51

2.1.7 组件	51
2.2 Design Support Library 常用控件详解	54
2.2.1 Snackbar 的使用	54
2.2.2 用 TextInputLayout 实现登录界面	55
2.2.3 FloatingActionButton 的使用	60
2.2.4 用 TabLayout 实现类似网易选项卡的动态滑动效果	61
2.2.5 用 NavigationView 实现抽屉菜单界面	68
2.2.6 用 CoordinatorLayout 实现 Toolbar 隐藏和折叠	74
2.3 本章小结	86
第 3 章 View 体系与自定义 View	87
3.1 View 与 ViewGroup	87
3.2 坐标系	89
3.2.1 Android 坐标系	89
3.2.2 View 坐标系	90
3.3 View 的滑动	91
3.3.1 layout() 方法	92
3.3.2 offsetLeftAndRight() 与 offsetTopAndBottom()	94
3.3.3 LayoutParams (改变布局参数)	95
3.3.4 动画	95
3.3.5 scrollTo 与 scrollBy	96
3.3.6 Scroller	98
3.4 属性动画	99
3.5 解析 Scroller	105
3.6 View 的事件分发机制	108
3.6.1 源码解析 Activity 的构成	108
3.6.2 源码解析 View 的事件分发机制	112
3.7 View 的工作流程	119
3.7.1 View 的工作流程入口	119
3.7.2 理解 MeasureSpec	122
3.7.3 View 的 measure 流程	126
3.7.4 View 的 layout 流程	132
3.7.5 View 的 draw 流程	135
3.8 自定义 View	139
3.8.1 继承系统控件的自定义 View	140
3.8.2 继承 View 的自定义 View	141
3.8.3 自定义组合控件	147

3.8.4 自定义 ViewGroup	152
3.9 本章小结	164
第 4 章 多线程编程	165
4.1 线程基础	165
4.1.1 进程与线程	165
4.1.2 线程的状态	167
4.1.3 创建线程	168
4.1.4 理解中断	170
4.1.5 安全地终止线程	171
4.2 同步	173
4.2.1 重入锁与条件对象	173
4.2.2 同步方法	175
4.2.3 同步代码块	176
4.2.4 volatile	177
4.3 阻塞队列	183
4.3.1 阻塞队列简介	183
4.3.2 Java 中的阻塞队列	184
4.3.3 阻塞队列的实现原理	186
4.3.4 阻塞队列的使用场景	188
4.4 线程池	190
4.4.1 ThreadPoolExecutor	190
4.4.2 线程池的处理流程和原理	192
4.4.3 线程池的种类	193
4.5 AsyncTask 的原理	197
4.6 本章小结	203
第 5 章 网络编程与网络框架	204
5.1 网络分层	204
5.2 TCP 的三次握手与四次挥手	205
5.3 HTTP 协议原理	207
5.3.1 HTTP 简介	207
5.3.2 HTTP 请求报文	208
5.3.3 HTTP 响应报文	209
5.3.4 HTTP 的消息报头	210
5.3.5 抓包应用举例	211
5.4 HttpClient 与 HttpURLConnection	212

5.4.1	HttpClient.....	212
5.4.2	URLConnection.....	216
5.5	解析 Volley.....	218
5.5.1	Volley 基本用法.....	218
5.5.2	源码解析 Volley.....	223
5.6	解析 OkHttp.....	231
5.6.1	OkHttp 基本用法.....	232
5.6.2	源码解析 OkHttp.....	240
5.7	解析 Retrofit.....	255
5.7.1	Retrofit 基本用法.....	255
5.7.2	源码解析 Retrofit.....	261
5.8	本章小结.....	270
第 6 章 设计模式.....		271
6.1	设计模式六大原则.....	271
6.2	设计模式分类.....	273
6.3	创建型设计模式.....	273
6.3.1	单例模式.....	274
6.3.2	简单工厂模式.....	277
6.3.3	工厂方法模式.....	279
6.3.4	建造者模式.....	281
6.4	结构型设计模式.....	284
6.4.1	代理模式.....	285
6.4.2	装饰模式.....	288
6.4.3	外观模式.....	291
6.4.4	享元模式.....	295
6.5	行为型设计模式.....	298
6.5.1	策略模式.....	298
6.5.2	模板方法模式.....	301
6.5.3	观察者模式.....	304
6.6	本章小结.....	307
第 7 章 事件总线.....		308
7.1	解析 EventBus.....	308
7.1.1	使用 EventBus.....	308
7.1.2	源码解析 EventBus.....	314
7.2	解析 otto.....	324

7.2.1 使用 otto.....	324
7.2.2 源码解析 otto.....	327
第 8 章 函数响应式编程.....	333
8.1 RxJava 基本用法.....	333
8.1.1 RxJava 概述.....	333
8.1.2 RxJava 基本实现.....	334
8.1.3 RxJava 的不完整定义回调.....	336
8.2 RxJava 的 Subject.....	338
8.3 RxJava 操作符入门.....	339
8.3.1 创建操作符.....	339
8.3.2 变换操作符.....	340
8.3.3 过滤操作符.....	344
8.3.4 组合操作符.....	349
8.3.5 辅助操作符.....	352
8.3.6 错误处理操作符.....	355
8.3.7 条件操作符和布尔操作符.....	357
8.3.8 转换操作符.....	360
8.4 RxJava 的线程控制.....	362
8.5 RxJava 的使用场景.....	362
8.5.1 RxJava 结合 OkHttp 访问网络.....	362
8.5.2 RxJava 结合 Retrofit 访问网络.....	364
8.5.3 用 RxJava 实现 RxBus.....	368
8.6 RxJava 源码解析.....	370
8.6.1 RxJava 的订阅过程.....	371
8.6.2 RxJava 的变换过程.....	372
8.6.3 RxJava 的线程切换过程.....	376
8.7 本章小结.....	381
第 9 章 注解与依赖注入框架.....	382
9.1 注解.....	382
9.1.1 注解分类.....	382
9.1.2 定义注解.....	384
9.1.3 注解处理器.....	385
9.2 依赖注入的原理.....	392
9.2.1 控制反转与依赖注入.....	392
9.2.2 依赖注入的实现方式.....	393

9.3 依赖注入框架	395
9.3.1 为何使用依赖注入框架	395
9.3.2 解析 ButterKnife	395
9.3.3 解析 Dagger2	405
9.4 本章小结	421
第 10 章 应用架构设计	422
10.1 MVC 模式	422
10.2 MVP 模式	423
10.2.1 应用 MVP 模式	424
10.2.2 MVP 结合 RxJava 和 Dagger2	431
10.3 MVVM 模式	438
10.3.1 解析 Data Binding	439
10.3.2 应用 Data Binding	457
10.4 本章小结	459
第 11 章 系统架构与 MediaPlayer 框架	460
11.1 Android 系统架构	460
11.2 Android 系统源码目录	463
11.2.1 整体结构	463
11.2.2 应用层部分	464
11.2.3 应用框架层部分	465
11.2.4 C/C++ 程序库部分	465
11.3 Source Insights 使用	466
11.4 MediaPlayer 框架	467
11.4.1 Java Framework 层的 MediaPlayer 分析	467
11.4.2 JNI 层的 MediaPlayer 分析	469
11.4.3 Native 层的 MediaPlayer 分析	471
11.5 本章小结	478
后记	479

第 1 章 Android 新特性

作为本书的第 1 章，肯定是全书比较简单的内容，而本书面向中高级的开发者的，所以基础知识讲得较少。在本章笔者会介绍 Android 5.0、Android 6.0 以及 Android 7.0 的新特性，并会详细介绍 Android 5.0 和 Android 6.0 的新特性。虽然 Android 7.0 已经发布了，但是目前 Android 5.0 和 Android 6.0 的技术仍旧没有普及。笔者希望通过本章的学习，读者能够了解各版本有什么新的功能、带给用户怎样的新体验，且能够掌握 Android 的新特性并尝试运用到项目中。

1.1 Android 5.0 新特性

Android 5.0 Lollipop 是 Google 于 2014 年 10 月 15 日发布的 Android 操作系统。北京时间 2014 年 6 月 26 日，Google I/O 2014 开发者大会在旧金山正式召开，发布了 Android 5.0 的开发者预览版。下面我们先来看看 Android 5.0 给我们带来了什么。

1.1.1 Android 5.0 主要新特性概述

作为一个 Android 开发者，我们需要了解最近的 Android 版本带来了什么特性，这样更有利于开发。谷歌（Google）在 Android 5.0 中带给我们很多惊喜。

1. 全新的 Material Design 设计风格

Material Design 是一种大胆的平面化创新（见图 1-1）。换句话说，谷歌希望能够让 Material Design 给用户带来纸张化的体验。这种新的视觉语言，在基本元素的处理上，借鉴了传统的印刷设计，以及字体版式、网格系统、空间、比例、配色和图像使用等这些基础的平面设计规范。另外，Material Design 还推崇实体隐喻理念，利用实体的表面与边缘的质感打造出视觉线索，

让用户感受到真实性。熟悉的触感让用户可以快速地理解并认知。在设计中可以在符合物理规律的基础上灵活地运用物质，打造出不同的使用体验。为了吸引用户的注意力，Material Design 还带来了有意义而且更合理的动态效果，以及维持整个系统的连续性体验。需要注意的是 Material Design 虽然是在 Android 5.0 时被提出来的，但是它也是在不断更新的，所以关于 Material Design 的内容会在第 2 章专门介绍。



图 1-1 Material Design

2. 支持多种设备

Android 系统的身影早已出现在多种设备中，比如：智能手机、平板电脑、笔记本电脑、智能电视、汽车、智能手表甚至是各种家用电子产品等。

3. 全新的通知中心设计

谷歌在 Android 5.0 中加入了全新风格的通知系统。改进后的通知系统会优先显示对用户来说比较重要的信息，而将不太紧急的内容隐藏起来。用户只需要向下滑动就可以查看全部的通知内容，如图 1-2 所示。

4. 支持 64 位 ART 虚拟机

Android 5.0 内部的性能上也提升了不少，它放弃了之前一直使用的 Dalvik 虚拟机，改用了 ART 虚拟机，实现了真正的跨平台编译，在 ARM、X86、MIPS 等无处不在。

5. Overview

多任务视窗现在有了一个新的名字，Overview。在界面中，每一个 App 都是一张独立的卡片，拥有立体式的层叠效果，用户可以设定“最近应用程序”，通过滑动来快速切换 App，如图 1-3 所示。

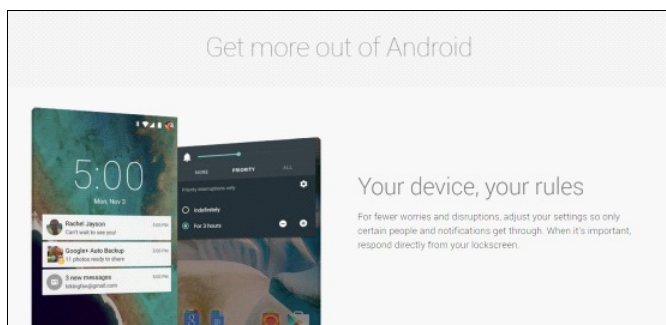


图 1-2 全新的通知中心设计

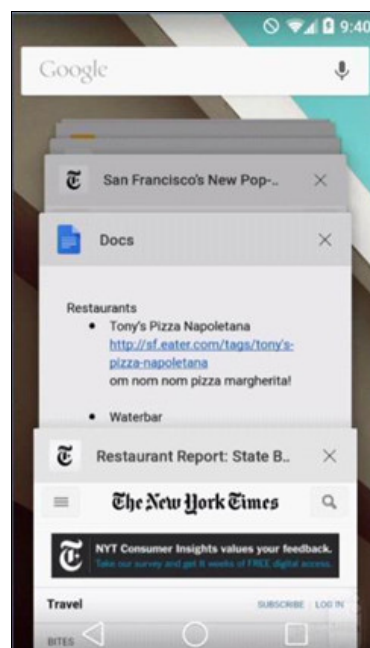


图 1-3 全新的“最近应用程序”

6. 设备识别解锁

现在个人识别解锁已经被普遍使用，比如当特定的智能手表出现在 Android 设备的附近时，就会直接绕过锁屏界面进行操作。而 Android 5.0 也增加了这种针对特定设备识别解锁的模式。换句话说，当设备没有检测到附近有可用的信任设备时，就会启动安全模式以防止未经授权访问。

7. Ok Google 语音指令

当手机处于待机状态时，对你的手机轻轻说声“Ok Google”，手机即刻被唤醒，只需说出简单的语言指令，如播放音乐、查询地点、拨打电话和设定闹钟等，一切只需“说说”而已。

8. Face unlock 面部解锁

在 Android 5.0 中，Google 花费大力气优化了面部解锁功能。当用户拿起手机处理锁屏界面上的消息通知时，面部解锁功能便自动被激活。随意浏览几条消息之后，手机已经默默地完成了面部识别。

1.1.2 替换 ListView 和 GridView 的 RecyclerView

有了 ListView、GridView，为什么还需要 RecyclerView 这样的控件呢？从整体上看，RecyclerView 架构提供了一种插拔式的体验，它具有高度的解耦、异常的灵活性和更高的效率，通过设置它提供的不同 LayoutManager、ItemDecoration、ItemAnimator 可实现更加丰富多样的效果。但是 RecyclerView 也有缺点和让人头疼的地方：设置列表的分割线时需要自定义，另外列表的点击事件需要自己去实现。

1. 配置 build.gradle

要想使用 RecyclerView，我们首先要导入 support-v7 包。因为我用的是 Android Studio（本书的所有例子均基于 Android Studio），所以在此需要在 build.gradle 中加入如下代码以自动导入 support-v7 包，记得配置完再重新 Build 一下工程。

```
dependencies {
    ...
    compile 'com.android.support:appcompat-v7:22.2.0'
    compile 'com.android.support:recyclerview-v7:22.1.0'
}
```

2. 使用 RecyclerView

```
RecyclerView mRecyclerView= (RecyclerView) this.findViewById(R.id.id_recyclerview);
//设置布局管理器
mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
// 设置 item 增加和删除时的动画
mRecyclerView.setItemAnimator(new DefaultItemAnimator());
mHomeAdaper=new HomeAdapter(this, mList);
mRecyclerView.setAdapter(mHomeAdaper);
```

与 ListView 不同的一点就是，需要设置布局管理器用于设置条目的排列样式，可以是垂直排列或者水平排列。这里我们设置 `setLayoutManager(new LinearLayoutManager(this))` 表示条目是线性排列的（默认是垂直排列的）。

```
public LinearLayoutManager(Context context) {
    this(context, VERTICAL, false);
}
```

如果想要设置为水平排列，可以按如下代码所示编写：

```
LinearLayoutManager linearLayoutManager=new LinearLayoutManager(this);
linearLayoutManager.setOrientation(LinearLayoutManager.HORIZONTAL);
mRecyclerView.setLayoutManager(linearLayoutManager);
```

此外，RecyclerView 比 ListView 的设置要复杂一些，主要是它需要自己去自定义分割线，设置动画和布局管理器，等等。布局文件 `activity_recycler_view.xml` 如下：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
```

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/id_recyclerview"
    android:divider="#FFB900"
    android:dividerHeight="1dp"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
</RelativeLayout>
```

Adapter 最大的改进就是对 ViewHolder 进行了封装定义，我们只需要自定义一个 ViewHolder 继承 RecyclerView.ViewHolder 就可以了。另外，Adapter 继承了 RecyclerView.Adapter，在 onCreateViewHolder 中加载条目布局，在 onBindViewHolder 中将视图与数据进行绑定。

```
class HomeAdapter extends RecyclerView.Adapter<HomeAdapter.MyViewHolder>
{
    private List<String> mList;
    private Context mContext;;
    public HomeAdapter(Context mContext,List<String>mList){
        this.mContext=mContext;
        this.mList=mList;
    }

    public void removeData(int position) {
        mList.remove(position);
        notifyItemRemoved(position);
    }
    @Override
    public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType)
    {
        MyViewHolder holder = new MyViewHolder(LayoutInflater.from(
            mContext).inflate(R.layout.item_recycler, parent,
            false));
        return holder;
    }

    @Override
    public void onBindViewHolder(final MyViewHolder holder, final int
    position)
    {
        holder.tv.setText(mList.get(position));
    }

    @Override
```

```

public int getItemCount()
{
    return mList.size();
}

class MyViewHolder extends RecyclerView.ViewHolder
{
    TextView tv;
    public MyViewHolder(View view)
    {
        super(view);
        tv = (TextView) view.findViewById(R.id.tv_item);
    }
}
}

```

在 HomeAdapter 的 onCreateViewHolder 方法中我们加载了条目的样式文件 item_recycler。它的布局很简单，就是显示一个文字。

```

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:background="@android:color/white"
    android:layout_height="wrap_content"
    >

    <TextView
        android:id="@+id/tv_item"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:gravity="center"
        android:text="moon" />

</FrameLayout>

```

运行程序效果如图 1-4 所示。从图 1-4 中可以看出，这里没有分割线，有些难看。下面我们来讲一讲分割线。

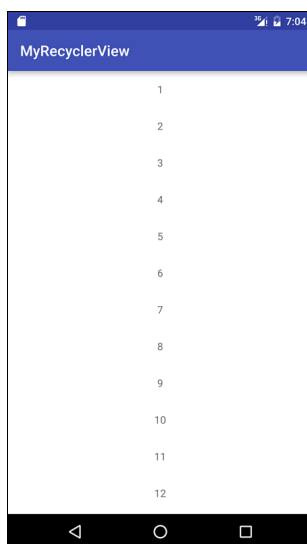


图 1-4 不带分割线的纵向排列列表

3. 设置分割线

我们可以使用 `mRecyclerView.addItemDecoration()` 来加入分割线。谷歌目前没有提供默认的分隔线，这就需要我们继承 `RecyclerView.ItemDecoration` 来自定义分隔线。

```
public class DividerItemDecoration extends RecyclerView.ItemDecoration {
    private static final int[] ATTRS = new int[]{
        android.R.attr.listDivider
    };
    public static final int HORIZONTAL_LIST = LinearLayoutManager.HORIZONTAL;
    public static final int VERTICAL_LIST = LinearLayoutManager.VERTICAL;
    private Drawable mDivider;
    private int mOrientation;
    public DividerItemDecoration(Context context, int orientation) {
        final TypedArray a = context.obtainStyledAttributes(ATTRS);
        mDivider = a.getDrawable(0);
        a.recycle();
        setOrientation(orientation);
    }

    public void setOrientation(int orientation) {
        if (orientation != HORIZONTAL_LIST && orientation != VERTICAL_LIST) {
            throw new IllegalArgumentException("invalid orientation");
        }
        mOrientation = orientation;
    }
}
```



```

@Override
public void onDraw(Canvas c, RecyclerView parent) {
    if (mOrientation == VERTICAL_LIST) {
        drawVertical(c, parent);
    } else {
        drawHorizontal(c, parent);
    }
}

public void drawVertical(Canvas c, RecyclerView parent) {
    final int left = parent.getPaddingLeft();
    final int right = parent.getWidth() - parent.getPaddingRight();
    final int childCount = parent.getChildCount();
    for (int i = 0; i < childCount; i++) {
        final View child = parent.getChildAt(i);
        android.support.v7.widget.RecyclerView v = new android.support.v7.widget.RecyclerView(parent.getContext());
        final RecyclerView.LayoutParams params = (RecyclerView.LayoutParams) child.getLayoutParams();
        final int top = child.getBottom() + params.bottomMargin;
        final int bottom = top + mDivider.getIntrinsicHeight();
        mDivider.setBounds(left, top, right, bottom);
        mDivider.draw(c);
    }
}

public void drawHorizontal(Canvas c, RecyclerView parent) {
    final int top = parent.getPaddingTop();
    final int bottom = parent.getHeight() - parent.getPaddingBottom();
    final int childCount = parent.getChildCount();
    for (int i = 0; i < childCount; i++) {
        final View child = parent.getChildAt(i);
        final RecyclerView.LayoutParams params = (RecyclerView.LayoutParams) child.getLayoutParams();
        final int left = child.getRight() + params.rightMargin;
        final int right = left + mDivider.getIntrinsicHeight();
        mDivider.setBounds(left, top, right, bottom);
        mDivider.draw(c);
    }
}

@Override
public void getItemOffsets(Rect outRect, int itemPosition, RecyclerView parent) {

```

```

        if (mOrientation == VERTICAL_LIST) {
            outRect.set(0, 0, 0, mDivider.getIntrinsicHeight());
        } else {
            outRect.set(0, 0, mDivider.getIntrinsicWidth(), 0);
        }
    }
}

```

这里核心的方法就是 `onDraw` 方法，它根据传进来的 `orientation` 来判断是绘制横向 `item` 的分割线还是纵向 `item` 的分割线。其中，`drawHorizontal` 用于绘制横向 `item` 的分割线，`drawVertical` 用于绘制纵向 `item` 的分割线。`getItemOffsets` 方法则用于设置 `item` 分割线的 `size`。虽然没有默认的分割线，但是好处我们也发现了：那就是我们可以更灵活地自定义分割线。实现自定义的分割线，我们只要在 `setAdapter` 之前加入如下代码便可加入分割线，如图 1-5 所示。

```

mRecyclerView.addItemDecoration(new DividerItemDecoration
    (RecyclerViewActivity.this, DividerItemDecoration.VERTICAL_LIST));

```



图 1-5 带分割线的纵向排列列表

4. 自定义点击事件

列表中条目的点击事件需要我们自己来定义，这是一个不尽如人意的地方。但是，自定义点击事件也并不是很难。在 `Adaper` 中定义接口并提供回调，我们在这里定义了条目的点击事件和长按点击事件。

```

public interface OnItemClickListener
{
    void onItemClick(View view, int position);
    void onItemLongClick(View view,int position);
}

```

```

    }

    public void setOnItemClickListener(OnItemClickListener mOnItemClickListener)
    {
        this.mOnItemClickListener = mOnItemClickListener;
    }

```

接下来对 **item** 中的控件进行点击事件监听并回调给我们自定义的监听，如下所示：

```

@Override
public void onBindViewHolder(final MyViewHolder holder, final int position)
{
    holder.tv.setText(mList.get(position));
    if (mOnItemClickListener != null)
    {
        holder.tv.setOnClickListener(new View.OnClickListener()
        {
            @Override
            public void onClick(View v)
            {
                int pos = holder.getLayoutPosition();
                mOnItemClickListener.onItemClick(holder.tv, pos);
            }
        });
    }
    holder.tv.setOnLongClickListener(new View.OnLongClickListener() {
        @Override
        public boolean onLongClick(View view) {
            int pos = holder.getLayoutPosition();
            mOnItemClickListener.onItemLongClick(holder.tv, pos);
            return false;
        }
    });
}
}

```

最后在 **Activity** 中进行监听：

```

mHomeAdaper.setOnItemClickListener(new HomeAdapter.OnItemClickListener() {
    @Override
    public void onItemClick(View view, int position) {
        Toast.makeText(RecyclerViewActivity.this, "点击第" + (position +
1) + "条", Toast.LENGTH_SHORT).show();
    }

    @Override

```

```

        public void onItemLongClick(View view, final int position) {
            new AlertDialog.Builder(RecyclerViewActivity.this)
                .setTitle("确认删除吗?")
                .setNegativeButton("取消", null)
                .setPositiveButton("确定", new DialogInterface.
                    OnClickListener() {
                        @Override
                        public void onClick(DialogInterface dialogInterface,
                            int i) {
                            mHomeAdaper.removeData(position);
                        }
                    })
                .show();
        }
    });
}

```

长按时会弹出对话框，删除时会有消失的动画，如图 1-6 所示。



图 1-6 自定义点击事件

5. 实现 GridView

只需要自定义横向的分割线，然后在代码中设置：

```

mRecyclerView.setLayoutManager(new StaggeredGridLayoutManager(4,
    StaggeredGridLayoutManager.VERTICAL));
mRecyclerView.addItemDecoration(new
    DividerGridItemDecoration(this));

```

很明显，这里设置一行显示条目为 4 个，其实现效果如图 1-7 所示。

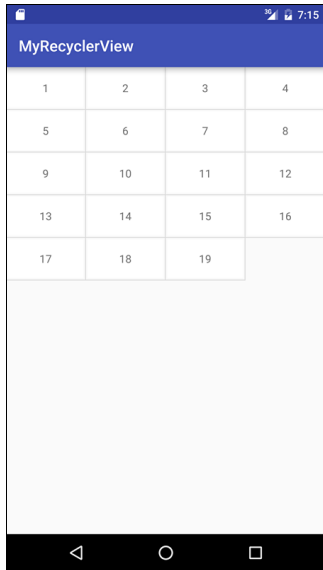


图 1-7 GridView

6. 实现瀑布流

虽然第三方实现的瀑布流已经很不错了，但是谷歌这次提供的 RecyclerView 支持瀑布流，我们没有理由不去用。因为它更稳定、效率更高、自定义能力更强。这里为了实现方便，我们可以不用写 `mRecyclerView.addItemDecoration(new DividerGridItemDecoration(this))` 来设置分割线，可以在 item 布局文件中定义分割距离 `android:layout_margin="2dp"`：

```
<?xml version="1-0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:background="@android:color/white"
    android:layout_height="wrap_content"
    android:layout_margin="2dp"
    >

    <TextView
        android:id="@+id/tv_item"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:gravity="center"
        android:text="moon" />

</FrameLayout>
```

实现瀑布流很简单，只要在 Adapter 写一个随机的高度来控制每个 item 的高度就可以了。

通常这个高度是由服务端返回的数据高度来控制的，在这里我们写一个随机的高度来控制每个 item 的高度：

```
mHeights = new ArrayList<Integer>();
for (int i = 0; i < mDataas.size(); i++)
{
    mHeights.add( (int) (100 + Math.random() * 300));
}
```

接着我们在 Adapter 的 onBindViewHolder 中设置每个 item 的高度，效果如图 1-8 所示。

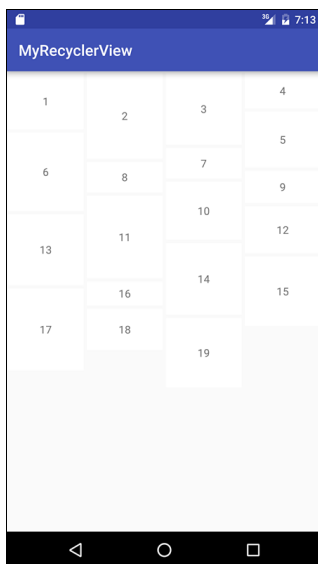


图 1-8 简单实现瀑布流

```
LayoutParams lp = holder.tv.getLayoutParams();
lp.height = mHeights.get(position);
holder.tv.setLayoutParams(lp);
```

1.1.3 卡片 CardView

Android 5.0 版本中新增了 CardView, CardView 继承自 FrameLayout 类，并且可以设置圆角和阴影，使得控件具有立体性，也可以包含其他的布局容器和控件。

1. 配置 build.gradle

如果 SDK 低于 5.0，我们仍旧要引入 v7 包。在 build.gradle 中加入如下代码以自动导入 support-v7 包。记得配置完再重新 Build 一下工程。

```
dependencies {
    ...
```

```

compile 'com.android.support:appcompat-v7:22.2.1'
compile 'com.android.support:cardview-v7:22.1.0'
}

```

2. 使用 CardView

先来看看布局，代码如下所示：

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width=
    "match_parent"
    android:layout_height="match_parent" android:paddingLeft="@dimen/activity_
    horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin" tools:context=
    ".CardViewActivity"
    android:orientation="vertical">
    <android.support.v7.widget.CardView
        android:id="@+id/tv_item"
        android:layout_width="match_parent"
        android:layout_height="250dp"
        android:layout_centerInParent="true"
        card_view:cardCornerRadius="20dp"
        card_view:cardElevation="20dp">

        <ImageView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="@drawable/cardview"
            android:scaleType="centerInside"
            />

    </android.support.v7.widget.CardView>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="30dp">
    <SeekBar
        android:id="@+id/sb_1"
        android:layout_width="200dp"
        android:layout_height="wrap_content"
        />

```

```

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="控制圆角大小"/>
    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="30dp">
        <SeekBar
            android:id="@+id/sb_2"
            android:layout_width="200dp"
            android:layout_height="wrap_content"/>

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="控制阴影大小"/>
    </LinearLayout>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="30dp">
        <SeekBar
            android:id="@+id/sb_3"
            android:layout_width="200dp"
            android:layout_height="wrap_content"/>

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="控制图片间距"/>

    </LinearLayout>
</LinearLayout>

```

这里有两个 `CardView` 的重要属性：`card_view:cardCornerRadius`，设置圆角的半径；`card_view:cardElevation`，设置阴影的半径。

除此之外，`CardView` 还有其他属性，在这里就不做实现了。其他属性如下所示。

- `CardView_cardBackgroundColor`：设置背景色。
- `CardView_cardElevation`：设置 Z 轴阴影。

- `CardView_cardMaxElevation`: 设置 Z 轴最大高度值。
- `CardView_cardUseCompatPadding`: 是否使用 `CompatPadding`。
- `CardView_cardPreventCornerOverlap`: 是否使用 `PreventCornerOverlap`。
- `CardView_contentPadding`: 内容的 padding。
- `CardView_contentPaddingLeft`: 内容的左 padding。
- `CardView_contentPaddingTop`: 内容的上 padding。
- `CardView_contentPaddingRight`: 内容的右 padding。
- `CardView_contentPaddingBottom`: 内容的底 padding。

接着来看看 Java 代码。

```
public class CardViewActivity extends AppCompatActivity {
    private CardView mCardView;
    private SeekBar sb1;
    private SeekBar sb2;
    private SeekBar sb3;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_card_view);
        assignViews();
    }
    private void assignViews() {
        mCardView = (CardView) findViewById(R.id.tv_item);
        sb1 = (SeekBar) findViewById(R.id.sb_1);
        sb2 = (SeekBar) findViewById(R.id.sb_2);
        sb3 = (SeekBar) findViewById(R.id.sb_3);
        sb1.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
            @Override
            public void onProgressChanged(SeekBar seekBar, int i, boolean b) {
                mCardView.setRadius(i);
            }

            @Override
            public void onStartTrackingTouch(SeekBar seekBar) {

            }

            @Override
            public void onStopTrackingTouch(SeekBar seekBar) {

            }
        });
    }
}
```

```
});

sb2.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int i, boolean b) {
        mCardView.setCardElevation(i);
    }

    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {

    }

    @Override
    public void onStopTrackingTouch(SeekBar seekBar) {

    }
});
sb3.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int i, boolean b) {
        mCardView.setContentPadding(i,i,i,i);
    }

    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {

    }

    @Override
    public void onStopTrackingTouch(SeekBar seekBar) {

    }
});
}
```

在这里设置了 3 个 seekBar 分别来设置 CardView: mCardView.setRadius()设置圆角的半径, mCardView.setCardElevation()设置阴影的半径, mCardView.setContentPadding()设置 CardView 中的子控件和父控件的距离。运行程序, 效果如图 1-9 所示。



图 1-9 CardView

1.1.4 3 种 Notification

Notification 可以让我们在获得消息的时候，在状态栏，锁屏界面来显示相应的信息。如果没有 Notification，那我们的 QQ 和微信以及其他应用没法主动通知我们，我们就需要时时地看手机来检查是否有新的信息和提醒，这着实让人烦心。这一点也体现出 Notification 的重要性。这里会介绍 3 种 Notification，分别是普通 Notification、折叠式 Notification 和悬挂式 Notification。

1. 普通 Notification

首先创建 Builder 对象，用 PendingIntent 控制跳转，这里跳转到网页。

```
Notification.Builder builder = new Notification.Builder(this);
Intent mIntent = new Intent(Intent.ACTION_VIEW, Uri.parse("http://blog.csdn.net/itachi85/"));
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, mIntent, 0);
```

有了 builder，我们就可以给 Notification 添加各种属性了：

```
builder.setContentIntent(pendingIntent);
builder.setSmallIcon(R.drawable.lanucher);
builder.setLargeIcon(BitmapFactory.decodeResource(getResources(), R.drawable.lanucher));
builder.setAutoCancel(true);
builder.setContentTitle("普通通知");
```

普通 Notification（普通通知）如图 1-10 所示。



图 1-10 普通 Notification

2. 折叠式 Notification

折叠式 Notification 是一种自定义视图的 Notification，用来显示长文本和一些自定义的布局场景。它有两种状态：一种是普通状态下的视图（如果不是自定义的话，和上面普通 Notification 的视图样式一样），另一种是展开状态下的视图。和普通 Notification 不同的是，我们需要自定义视图，而这个视图显示的进程和我们创建视图的进程不在一个进程，所以我们需要使用 RemoteViews。首先要使用 RemoteViews 来创建我们的自定义视图。

```
//用 RemoteViews 来创建自定义 Notification 视图
RemoteViews remoteViews = new RemoteViews(getPackageName(), R.layout.view_fold);
```

视图的布局文件：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="100dp"
    android:background="@drawable/fold"
    android:orientation="horizontal">
    <ImageView
        android:id="@+id/iv_image"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:src="@drawable/fold"
    />
```

```

<TextView
    android:id="@+id/tv_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="30dp"
    android:layout_marginLeft="150dp"
    android:text="展开后的自定义视图"
    android:textColor="@color/colorPrimaryDark"/>
</LinearLayout>

```

然后，我们需要把自定义的视图赋值给 Notification 的视图，如下代码是把自定义视图赋值给 Notification 展开时的视图：

```

//指定展开时的视图
notification.bigContentView = remoteViews;

```

当然，我们也可以把自定义视图赋值给 Notification 普通状态时的视图，如下所示：

```

//指定普通状态时的视图
notification.contentView = remoteViews;

```

其他的代码和普通 Notification 没什么区别，折叠式 Notification 的完整代码如下所示：

```

Notification.Builder builder = new Notification.Builder(this);
Intent mIntent = new Intent(Intent.ACTION_VIEW, Uri.parse("http://
blog.csdn.net/itachi85/"));
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
mIntent, 0);
builder.setContentIntent(pendingIntent);
builder.setSmallIcon(R.drawable.foldleft);
builder.setLargeIcon(BitmapFactory.decodeResource(getResources(),
R.drawable.lanucher));
builder.setAutoCancel(true);
builder.setContentTitle("折叠式通知");
//用 RemoteViews 来创建自定义 Notification 视图
RemoteViews remoteViews = new RemoteViews(getPackageName(),
R.layout.view_fold);
Notification notification = builder.build();
//指定展开时的视图
notification.bigContentView = remoteViews;
notificationManager.notify(1, notification);

```

如果不是自定义普通状态视图的话，折叠式 Notification（折叠式通知）普通状态下和普通 Notification（普通通知）没什么区别，如图 1-11 所示。我们接着往下拉，使折叠式 Notification

完全展开就会出现我们自定义的视图，如图 1-12 所示。



图 1-11 普通状态下的折叠式 Notification



图 1-12 展开状态下的折叠式 Notification

3. 悬挂式 Notification

悬挂式 Notification 是 Android 5.0 新增加的方式。和前两种显示方式不同的是，前两种需要下拉通知栏才能看到通知；而悬挂式 Notification 不需要下拉通知栏就直接显示出来悬挂在屏幕上方，并且焦点不变，仍在用户操作的界面，因此不会打断用户的操作。其过几秒就会自动消失。和前两种 Notification 不同的是，它需要调用 `setFullScreenIntent` 来将 Notification 变为悬挂式 Notification。

```
PendingIntent hangPendingIntent = PendingIntent.getActivity(this, 0,
    hangIntent, PendingIntent.FLAG_CANCEL_CURRENT);
builder.setFullScreenIntent(hangPendingIntent, true);
```

实现悬挂式 Notification 的完整代码：

```
Notification.Builder builder = new Notification.Builder(this);
Intent mIntent = new Intent(Intent.ACTION_VIEW, Uri.parse("http://
blog.csdn.net/itachi85/"));
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
    mIntent, 0);
builder.setContentIntent(pendingIntent);
builder.setSmallIcon(R.drawable.foldleft);
builder.setLargeIcon(BitmapFactory.decodeResource(getResources(),
    R.drawable.lanucher));
builder.setAutoCancel(true);
```

```
builder.setContentTitle("悬挂式通知");  
//设置点击跳转  
Intent hangIntent = new Intent();  
hangIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);  
hangIntent.setClass(this, MyNotificationActivity.class);  
PendingIntent hangPendingIntent = PendingIntent.getActivity(this, 0,  
hangIntent, PendingIntent.FLAG_CANCEL_CURRENT);  
builder.setFullScreenIntent(hangPendingIntent, true);  
notificationManager.notify(2, builder.build());
```

悬挂式 Notification（悬挂式通知）运行效果如图 1-13 所示。

4. Notification 的显示等级

Android 5.0 加入了一种新的模式 Notification 的显示等级，共有以下 3 种。

- VISIBILITY_PUBLIC：任何情况都会显示通知。
- VISIBILITY_PRIVATE：只有在没有锁屏时会显示通知。
- VISIBILITY_SECRET：在 pin、password 等安全锁和没有锁屏的情况下才能够显示通知。



图 1-13 悬挂式 Notification

设置非常简单，只要调用 `setVisibility` 方法就可以了：

```
builder.setVisibility(Notification.VISIBILITY_PUBLIC);
```

我在这里写了一个方法来设置 Notification 等级，用 `radioGroup` 来演示 Notification 的各个显示等级，详情请参照源码。

```

private void selectNotofovationLevel(Notification.Builder builder) {
    switch (radioGroup.getCheckedRadioButtonId()) {
        case R.id.rb_public:
            builder.setVisibility(Notification.VISIBILITY_PUBLIC);
            builder.setText("public");
            break;
        case R.id.rb_private:
            builder.setVisibility(Notification.VISIBILITY_PRIVATE);
            builder.setText("private");
            break;
        case R.id.rb_secret:
            builder.setVisibility(Notification.VISIBILITY_SECRET);
            builder.setText("secret");
            break;
        default:
            builder.setVisibility(Notification.VISIBILITY_PUBLIC);
            builder.setText("public");
            break;
    }
}

```

1.1.5 Toolbar 与 Palette

对于已经被不大好用的 ActionBar“折磨”的开发者来说, Toolbar 的出现确实是一个好消息。Toolbar 是应用内容的标准工具栏, 可以说是 ActionBar 的升级版。这两者不是独立关系, 要使用 Toolbar, 还是得跟 ActionBar 有关系的。相比于 ActionBar, Toolbar 最明显的一点就是变得很自由, 可随处放置, 其具体使用方法和 ActionBar 很类似。

1. 引入 Toolbar

我们首先还是要引入 v7 支持包, 在 build.gradle 中配置如下代码:

```

dependencies {
    ...
    compile 'com.android.support:appcompat-v7:23.0.1'
    compile 'com.android.support:palette-v7:23.0.1'
}

```

接下来为了显示 Toolbar 控件, 先要在 style 里把 ActionBar 去掉, 如下所示:

```

<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    </style>

```



```

        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>

```

设置各个部分属性，如图 1-14 所示。

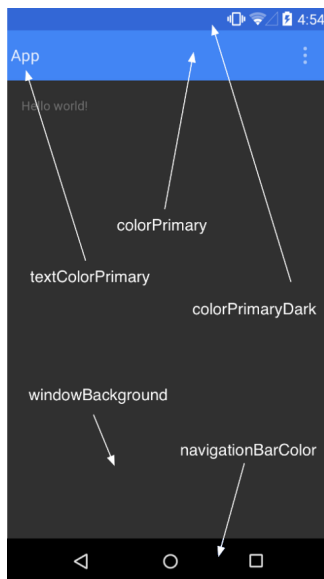


图 1-14 Toolbar 设置各个部分属性

接着写一个 mytoolbar.xml，引入 Toolbar 控件。这么写的目的是方便多次调用，我们可以在其他布局中用 include 引用此布局。

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.Toolbar xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    android:minHeight="?attr/actionBarSize"
    >
</android.support.v7.widget.Toolbar>

```

在主界面布局用 include 引用 mytoolbar.xml 中的 Toolbar，在主界面布局中我们用 DrawerLayout 来完成侧滑的效果。

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"

```

```

        android:layout_height="match_parent"
        android:orientation="vertical"
        tools:context=".ToolbarActivity">
<include layout="@layout/mytoolbar" />
<android.support.v4.widget.DrawerLayout
    android:id="@+id/id_drawerlayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
<!--内容界面-->
<LinearLayout
    android:id="@+id/ll_content"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="@drawable/bitmap">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:text="内容界面"
        android:textColor="@android:color/white"/>

</LinearLayout>
<LinearLayout
    android:id="@+id/ll_tabs"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/darker_gray"
    android:orientation="vertical"
    android:layout_gravity="start">

    <TextView
        android:id="@+id/tv_close"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:clickable="true"
        android:text="侧滑界面, 点击收回侧滑"
        android:textColor="@android:color/white"/>
</LinearLayout>
</android.support.v4.widget.DrawerLayout>
</LinearLayout>

```

接下来在 Java 代码中设定 Toolbar，代码如下所示：

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_toolbar);
    initView();
}

private void initView() {
    mToolbar= (Toolbar) this.findViewById(R.id.toolbar);
    setSupportActionBar(mToolbar);
}

```

是不是很容易？当然 Toolbar 能做的不只这些，接下来试试自定义 Toolbar。

2. 自定义 Toolbar

当然，我们还可以设置 Toolbar 的标题和图标以及 Menu Item 等属性。Menu Item 的设置和 ActionBar 类似，我们在 menu/main.xml 中去声明，代码如下所示：

```

<menu 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"
    tools:context=".ToolbarActivity" >

    <item
        android:id="@+id/ab_search"
        android:orderInCategory="80"
        android:title="搜索"
        app:actionViewClass="android.support.v7.widget.SearchView"
        app:showAsAction="ifRoom"/>

    <item
        android:id="@+id/action_share"
        android:orderInCategory="90"
        android:title="分享"
        app:actionProviderClass="android.support.v7.widget.ShareActionProvider"
        app:showAsAction="ifRoom"/>

    <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:title="设置"
        app:showAsAction="never"/>

</menu>

```

然后覆写 onCreateOptionsMenu 并在 toolbar.setOnMenuItemClickListener 实现点击 MenuItem 的回调，代码如下所示：

```

@Override

```

```

public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.main, menu);
    return true;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_toolbar);
        initView();
    }

    private void initView() {
        tv_close= (TextView) this.findViewById(R.id.tv_close);
        mToolbar= (Toolbar) this.findViewById(R.id.toolbar);
        mToolbar.setTitle("Toolbar");
        setSupportActionBar(mToolbar);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        getSupportActionBar().setLogo(R.drawable.ic_launcher);
        mToolbar.setOnMenuItemClickListener(new
        Toolbar.OnMenuItemClickListener() {
            @Override
            public boolean onOptionsItemSelected(MenuItem item) {
                switch (item.getItemId()) {
                    case R.id.action_settings:
                        Toast.makeText(ToolbarActivity.this, "action_settings",
                        Toast.LENGTH_SHORT).show();
                        break;
                    case R.id.action_share:
                        Toast.makeText(ToolbarActivity.this, "action_share",
                        Toast.LENGTH_SHORT).show();
                        break;
                    default:
                        break;
                }
                return true;
            }
        });
    }
}

```

3. 添加 DrawerLayout 实现侧滑

DrawerLayout 实现侧滑很简单，这里因为是介绍 Toolbar 的，所以就不实现稍微复杂的效果了。

```

//设置侧滑布局
mDrawerLayout= (DrawerLayout) this.findViewById(R.id.id_drawerlayout);

```

```
mDrawerToggle = new ActionBarDrawerToggle(this, mDrawerLayout,
mToolbar, R.string.drawer_open, R.string.drawer_close);
mDrawerToggle.syncState();
mDrawerLayout.setDrawerListener(mDrawerToggle);
tv_close.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        mDrawerLayout.closeDrawer(Gravity.LEFT);
    }
});
```

程序运行效果如图 1-15 所示，DrawerLayout 展开效果如图 1-16 所示。



图 1-15 DrawerLayout 未展开效果



图 1-16 DrawerLayout 展开效果

4. Palette 的应用

这次 Android 5.x 用 Palette 来提取颜色，从而让主题能够动态适应当前界面的色调，做到整个 App 颜色的基调和諧统一。Android 内置了几种提取色调的种类：

- Vibrant（充满活力的）
- Vibrant dark（充满活力的黑）
- Vibrant light（充满活力的亮）
- Muted（柔和的）
- Muted dark（柔和的黑）
- Muted light（柔和的亮）

要使用 Palette，我们需要引用 'com.android.support:palette-v7:23.0.1'。这在之前已经配置过了，实现提取颜色非常容易，只要将 bitmap 传递给 Palette，调用 generate 即可。在 onGenerated

回调中得到图片的色调，这里我们获取的是图片充满活力的色调，最后我们把 Toolbar 的背景设置为该图片的色调。

```
Bitmap bitmap= BitmapFactory.decodeResource(getResources(),R.drawable.bitmap);
Palette.from(bitmap).generate(new Palette.PaletteAsyncListener() {
    @Override
    public void onGenerated(Palette palette) {
        Palette.Swatch swatch=palette.getVibrantSwatch();
        getSupportActionBar().setBackgroundDrawable(new ColorDrawable
            (swatch.getRgb()));
    }
});
```

1.2 Android 6.0 新特性

Google I/O 2015 大会在 2015 年 5 月 28 日举行，在发布会上推出了 Android 6.0 Marshmallow(棉花糖)的新系统。新系统的整体设计风格依然保持扁平化的 Material Design 风格。Android 6.0 在对软件体验与运行性能上进行了大幅度的优化。据测试，Android 6.0 可使设备续航时间提升 30%。

1.2.1 Android 6.0 主要新特性概述

1. 应用权限管理

在 Android 6.0 中，应用许可提示可以自定义了。它允许对应用的权限进行高度管理，比如应用能否使用位置、相机、网络和通信录等，这些都开放给开发者和用户。此前的 Android 系统的应用权限管理只能靠第三方应用来实现，在 Android 6.0 中应用权限管理成为系统级的功能。

2. Android Pay

Android Pay 是 Android 支付统一标准。Android 6.0 系统中集成了 Android Pay，其特性在于简洁、安全和可选性。它是一个开放性平台，用户可以选择谷歌的服务或者使用银行的 App 来使用它。Android Pay 支持 Android 4.4 以后的系统设备并且可以使用指纹来进行支付。

3. 指纹支持

虽然很多厂商的 Android 手机实现了指纹的支持，但是这些手机都使用了非谷歌认证的技术。这一次谷歌提供的指纹识别支持，旨在统一指纹识别的技术方案。

4. Doze 电量管理

Android 6.0 自带 Doze 电量管理功能。手机静止不动一段时间后，会进入 Doze 电量管理模式。谷歌表示，当屏幕处于关闭状态时，平均续航时间可提高 30%。

5. App Links

Android 6.0 加强了软件间的关联，允许开发者将 App 和他们的 Web 域名关联。谷歌大会展示了 App Links 的应用场景，比如你的手机邮箱里收到一封邮件，内文里有一个 Twitter 链接，点击该链接可以直接跳转到 Twitter 应用，而不再是网页。

6. Now on Tap

在桌面或 App 的任意界面，长按 Home 键即可激活 Now on Tap，它会识别当前屏幕上的内容并创建 Now 卡片。比如，你和某人聊天时提到一起去一家餐馆吃饭，这时你长按 Home 键，Now on Tap 就会创建 Now 卡片提供这家餐馆的地址和评价等相关信息。如屏幕中出现电话号码，它就会提供一个拨号的标志，你可以直接拨打；而无须先复制，然后再粘贴到拨号界面。它还可以识别日历、地址、音乐、地标等信息。Now on Tap 能够快速便捷地帮助用户，完成用户需求，这在很大程度上解放了用户的嘴和双手。这可以说是自 Google Now 发布以来最为重大的一次升级。

1.2.2 运行时权限机制

在 Android 6.0 以前，我们安装 App 时会列出安装的 App 的访问权限，而且只有安装时会出现一次。一旦我们同意并安装了此 App，这个 App 就可以在用户毫不知晓的情况下访问权限内的所有东西，比如用户的通信信息、用户的位置等，这会侵犯用户的隐私。在 Android 6.0 时，将不会在安装的时候授予权限；取而代之的是，App 不得不在运行时一个一个询问用户来授予权限。对于 Android 用户来说这显然是一个好消息；但是对于开发者来说这显然是一场噩梦，因为开发者不能像以前一样随意地调用方法了，你需要在每个需要权限的地方检查权限，否则等待你的就是 App 崩溃。

1. Android 6.0 之前版本的应对之策

Android 6.0 系统默认为 targetSdkVersion 小于 23 的应用授予了所申请的所有权限，所以如果你以前的 App 设置的 targetSdkVersion 低于 23，在运行时也不会崩溃。为了验证这个问题，首先要在 build.gradle 中将 targetSdkVersion 设置为 23 以下。需要注意的是当前 compileSdkVersion 为 23，也就是 SDK 版本为 23，如下代码所示：

```
android {  
    compileSdkVersion 23  
    buildToolsVersion "23.0.3"  
  
    defaultConfig {  
        applicationId "com.example.liuwangshu.moonpermissions"  
        minSdkVersion 15  
        targetSdkVersion 22  
        versionCode 1  
        versionName "1.0"  
    }  
}
```

```

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

```

接下来定义一个按钮，点击按钮时拨打电话。

```

Intent intent = new Intent(Intent.ACTION_CALL);
Uri data = Uri.parse("tel:" + "10086");
intent.setData(data);
startActivity(intent);

```

运行程序会发现仍旧能打电话，这是在我们意料之中的。接下来我们取消授予的电话权限，这时系统会弹出提示框，提醒我们拒绝此权限可能会导致其无法正常运行。我们点击“拒绝”，仍要取消授予的电话权限。如图 1-17 所示。

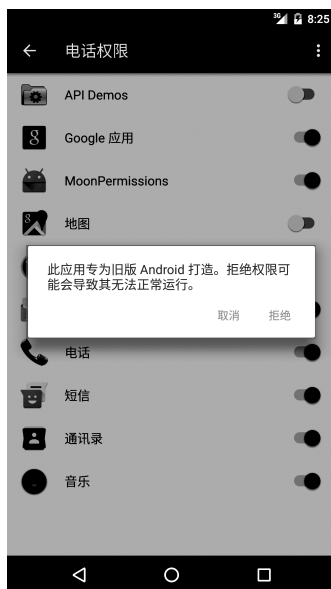


图 1-17 取消授予的电话权限

我们接下来再点击按钮拨打电话时，则发现无法拨打该电话。这样的用户体验显然不好。当然这只是电话权限。目前官方的做法是，如果用户取消该项授权，那么依赖该项授权的方法的返回值为 `null`，你的 App 可能会报空指针异常，所以我们需要尽快着手修改 App 支持最新的权限系统。

2. Normal Permissions 与 Dangerous Permission

Google 将权限分为两类：一类是 Normal Permissions，这类权限一般不涉及用户隐私，是无需用户进行授权的，比如手机振动、访问网络等，这些权限只需要在 `AndroidManifest.xml` 中简单声明就好，安装时就授权，无须每次使用时都检查权限，而且用户不能取消以上授权；另一类是 Dangerous Permission，一般会涉及用户隐私，需要用户进行授权，比如读取 `sdcard`、访问通信录等。

Normal Permissions 如表 1-1 所示。

表 1-1 Normal Permissions

android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
android.permission.ACCESS_NETWORK_STATE
android.permission.ACCESS_NOTIFICATION_POLICY
android.permission.ACCESS_WIFI_STATE
android.permission.ACCESS_WIMAX_STATE
android.permission.BLUETOOTH
android.permission.BLUETOOTH_ADMIN
android.permission.BROADCAST_STICKY
android.permission.CHANGE_NETWORK_STATE
android.permission.CHANGE_WIFI_MULTICAST_STATE
android.permission.CHANGE_WIFI_STATE
android.permission.KILL_BACKGROUND_PROCESSES
android.permission.MODIFY_AUDIO_SETTINGS
android.permission.NFC
android.permission.READ_SYNC_SETTINGS
android.permission.READ_SYNC_STATS
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.REORDER_TASKS
android.permission.REQUEST_INSTALL_PACKAGES
android.permission.SET_TIME_ZONE
android.permission.SET_WALLPAPER
android.permission.SET_WALLPAPER_HINTS
android.permission.TRANSMIT_IR
android.permission.USE_FINGERPRINT
android.permission.VIBRATE
android.permission.WAKE_LOCK

续表

android.permission.WRITE_SYNC_SETTINGS
com.android.alarm.permission.SET_ALARM
com.android.launcher.permission.INSTALL_SHORTCUT
com.android.launcher.permission.UNINSTALL_SHORTCUT

Dangerous Permission 则是以分组的形式给出的，如图 1-18 所示。

Permission Group	Permissions
android.permission-group.CALENDAR	• android.permission.READ_CALENDAR • android.permission.WRITE_CALENDAR
android.permission-group.CAMERA	• android.permission.CAMERA
android.permission-group.CONTACTS	• android.permission.READ_CONTACTS • android.permission.WRITE_CONTACTS • android.permission.GET_ACCOUNTS
android.permission-group.LOCATION	• android.permission.ACCESS_FINE_LOCATION • android.permission.ACCESS_COARSE_LOCATION
android.permission-group.MICROPHONE	• android.permission.RECORD_AUDIO
android.permission-group.PHONE	• android.permission.READ_PHONE_STATE • android.permission.CALL_PHONE • android.permission.READ_CALL_LOG • android.permission.WRITE_CALL_LOG • com.android.voicemail.permission.ADD_VOICEMAIL • android.permission.USE_SIP • android.permission.PROCESS_OUTGOING_CALLS
android.permission-group.SENSORS	• android.permission.BODY_SENSORS
android.permission-group.SMS	• android.permission.SEND_SMS • android.permission.RECEIVE_SMS • android.permission.READ_SMS • android.permission.RECEIVE_WAP_PUSH • android.permission.RECEIVE_MMS • android.permission.READ_CELL_BROADCASTS
android.permission-group.STORAGE	• android.permission.READ_EXTERNAL_STORAGE • android.permission.WRITE_EXTERNAL_STORAGE

图 1-18 Dangerous Permission

同一组的任何一个权限被授权了，其他权限也自动被授权。此外，对于申请时弹出的提示框上面的文本说明也是对整个权限组的说明，而不是单个权限的说明。

3. 实现支持运行时权限

下面仍旧通过一个打电话的例子来讲解如何支持运行时权限。首先我们要在 build.gradle 中将 targetSdkVersion 设置为 23。

```

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.3"

    defaultConfig {
        applicationId "com.example.liuwangshu.moonpermissions"
        minSdkVersion 15
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}

```

接下来是 Java 代码。因为比较简单，所以这里就不一步一步地讲解了，代码如下所示：

```

public class MainActivity extends AppCompatActivity {
    private Button bt_call;
    private static final int PERMISSIONS_REQUEST_CALL_PHONE = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bt_call = (Button) this.findViewById(R.id.bt_call);
        bt_call.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                call();
            }
        });
    }

    public void call() {
        //检查 App 是否有 permission.CALL_PHONE 的权限
        if (ActivityCompat.checkSelfPermission(this, Manifest.permission.CALL_PHONE) != PackageManager.PERMISSION_GRANTED) {
            //如果没有 permission.CALL_PHONE 的权限，就申请该权限
            ActivityCompat.requestPermissions(this,

```

```

        new String[]{Manifest.permission.CALL_PHONE},
        PERMISSIONS_REQUEST_CALL_PHONE);
    } else {
        callPhone();
    }
}

public void callPhone() {
    Intent intent = new Intent(Intent.ACTION_CALL);
    Uri data = Uri.parse("tel:" + "10086");
    intent.setData(data);
    try {
        startActivity(intent);
    } catch (SecurityException e) {
        e.printStackTrace();
    }
}

//申请权限的回调
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
int[] grantResults) {

    if (requestCode == PERMISSIONS_REQUEST_CALL_PHONE) {
        if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            callPhone();
        } else {
            Toast.makeText(MainActivity.this, "权限被拒绝", Toast.LENGTH_
SHORT).show();
        }
        return;
    }
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
}
}

```

这里定义一个按钮，点击按钮会调用 `call` 方法，`call` 方法首先判断当前 App 是否有 `permission.CALL_PHONE` 的权限。如果有，则直接调用 `callPhone` 方法来打电话；如果没有该权限，则弹出提示框进行权限申请，如图 1-19 所示。`onRequestPermissionsResult` 就是申请权限的回调，如果用户选择“允许”，则调用 `callPhone` 方法；如果用户选择“拒绝”，就弹出 Toast 显示权限被拒绝。需要注意的是，如果我们选择“允许”，下一次就不会弹出权限申请提示框了；如果我们点击“拒绝”，则下一次还会弹出权限申请提示框，只不过这一次会多出一个选

项，叫作“不再询问”，如图 1-20 所示。如果我们勾选了该选项，则下一次就不会弹出权限申请提示框，而直接调用 `onRequestPermissionsResult`，回调结果为最后一次用户的选择，也就会弹出我们定义的 `Toast`：“权限被拒绝”。



图 1-19 弹出权限申请提示框

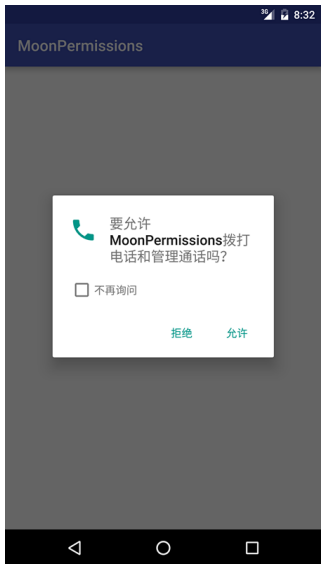


图 1-20 弹出权限申请提示框（带“不再询问”选项）

想要再次打开该权限，则需要在设置→应用→配置应用→应用权限→电话权限中操作，如图 1-21 所示。

如果这个程序运行在 5.0 版本的手机上会怎样呢？如果运行在 5.0 版本的手机上时，`ActivityCompat.checkSelfPermission (this, Manifest.permission.CALL_PHONE) = PackageManager.PERMISSION_GRANTED` 也就是授权过了，会调用 `callPhone` 方法直接拨打电话。

4. 处理“不再询问”选项

如果用户选择了“不再询问”，那么每次我们调用需要访问该权限的 API 时都会失效，这显然不会带来好的用户体验，所以我们需要做的就是给用户一个友好的提示。这时候需要使用 `shouldShowRequestPermissionRationale` 方法，这个方法用来帮助开发者向用户解释权限的情况。如果用户选择了“不再询问”选项，则 `shouldShowRequestPermissionRationale` 方法会返回 `false`，这时候我们可以弹出 `AlertDialog` 来提醒用户允许访问该权限的重要性，代码如下所示。弹出的 `AlertDialog` 效果如图 1-22 所示。

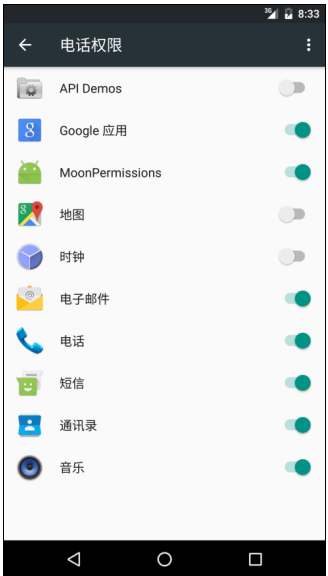


图 1-21 电话权限管理

```
@Override
public void onRequestPermissionsResult(int requestCode, String[]
```

```

permissions, int[] grantResults) {

    if (requestCode == PERMISSIONS_REQUEST_CALL_PHONE) {
        if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            callPhone();
        } else {
            if (!ActivityCompat.shouldShowRequestPermissionRationale
                (this, Manifest.permission.CALL_PHONE)) {
                AlertDialog dialog = new AlertDialog.Builder(this)
                    .setMessage("该功能需要访问电话的权限，不开启将无法正常工作!")
                    .setPositiveButton("确定", new DialogInterface.
                        OnClickListener() {
                            @Override
                            public void onClick(DialogInterface dialog, int which) {
                                dialog.dismiss();
                            }
                        })
                    .create();
                dialog.show();
            }
            return;
        }
    }
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
}
}

```



图 1-22 处理“不再询问”选项

5. PermissionsDispatcher 框架的使用

当然，每次在需要判断权限的地方均写如上面所示的一套方法，总是一件麻烦事。我们可以自己封装起来，因为流程都是一样的，变化的只是传进来的我们需要判断的权限就可以了。

不过目前已经有很多框架实现了这一点，这里拿 PermissionsDispatcher 来举个例子。首先，我们要在工程项目的 build.gradle 中添加如下代码：

```
dependencies {  
    ...  
    classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'  
}
```

在 app 模块的 build.gradle 中添加如下代码：

```
apply plugin: 'android-apt'  
dependencies {  
    ...  
    compile 'com.github.hotchemi:permissionsdispatcher:2.1.3'  
    apt 'com.github.hotchemi:permissionsdispatcher-processor:2.1.3'  
}
```

接下来我们需要了解几个注释。

- **RuntimePermissions**：必需的注释，它用来注册一个 Activity 或 Fragment，使它们可以处理权限。
- **NeedsPermission**：必需的注释，在需要获取权限的地方注释，用来获取权限。
- **OnShowRationale**：提示用户为何要开启此权限。在用户选择拒绝后，再次需要访问该权限时调用。
- **OnPermissionDenied**：用户选择拒绝时的提示。
- **OnNeverAskAgain**：用户选择不再询问后的提示。

好了，我们来编写代码。我们将上面打电话的例子应用在这里，如下所示：

```
@RuntimePermissions  
public class ThirdPartyActivity extends AppCompatActivity {  
    private Button bt_call;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_third_party);  
        bt_call = (Button) this.findViewById(R.id.bt_call);  
        bt_call.setOnClickListener(new View.OnClickListener() {  
            @Override  
            public void onClick(View v) {  
            }  
        });  
    }  
}
```

```
@NeedsPermission(Manifest.permission.CALL_PHONE)
//在需要获取权限的地方注释
void call() {
    Intent intent = new Intent(Intent.ACTION_CALL);
    Uri data = Uri.parse("tel:" + "10086");
    intent.setData(data);
    try {
        startActivity(intent);
    } catch (SecurityException e) {
        e.printStackTrace();
    }
}

@OnShowRationale(Manifest.permission.CALL_PHONE)
//提示用户为何要开启此权限
void showWhy(final PermissionRequest request) {
    new AlertDialog.Builder(this)
        .setMessage("提示用户为何要开启此权限")
        .setPositiveButton("知道了", new DialogInterface.
            OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    request.proceed(); //再次执行权限请求
                }
            })
        .show();
}

@OnPermissionDenied(Manifest.permission.CALL_PHONE)
//用户选择拒绝时的提示
void showDenied() {
    Toast.makeText(this, "用户选择拒绝时的提示", Toast.LENGTH_SHORT).
        show();
}

@OnNeverAskAgain(Manifest.permission.CALL_PHONE)
//用户选择不再询问后的提示
void showNotAsk() {
    new AlertDialog.Builder(this)
        .setMessage("该功能需要访问电话的权限，不开启将无法正常工作！")
        .setPositiveButton("确定", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
```



```

        }
    }).show();
}
}

```

这时我们要重新编译程序，会生成一个辅助类 `ThirdPartyActivityPermissionsDispatcher`，其余的事交给它就可以了。完整代码如下所示：

```

@RuntimePermissions
public class ThirdPartyActivity extends AppCompatActivity {
    private Button bt_call;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_third_party);
        bt_call = (Button) this.findViewById(R.id.bt_call);
        bt_call.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                ThirdPartyActivityPermissionsDispatcher.callWithCheck
                    (ThirdPartyActivity.this);
            }
        });
    }

    @NeedsPermission(Manifest.permission.CALL_PHONE)
    //在需要获取权限的地方注释
    void call() {
        Intent intent = new Intent(Intent.ACTION_CALL);
        Uri data = Uri.parse("tel:" + "10086");
        intent.setData(data);
        try {
            startActivity(intent);
        } catch (SecurityException e) {
            e.printStackTrace();
        }
    }

    @OnShowRationale(Manifest.permission.CALL_PHONE)
    //提示用户为何要开启此权限
    void showWhy(final PermissionRequest request) {
        new AlertDialog.Builder(this)

```

```

        .setMessage("提示用户为何要开启此权限")
        .setPositiveButton("知道了", new DialogInterface.
OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        request.proceed(); //再次执行权限请求
    }
})
        .show();
    }

    @OnPermissionDenied(Manifest.permission.CALL_PHONE)
    //用户选择拒绝时的提示
    void showDenied() {
        Toast.makeText(this, "用户选择拒绝时的提示", Toast.LENGTH_SHORT).
            show();
    }

    @OnNeverAskAgain(Manifest.permission.CALL_PHONE)
    //用户选择不再询问后的提示
    void showNotAsk() {
        new AlertDialog.Builder(this)
            .setMessage("该功能需要访问电话的权限，不开启将无法正常工作！")
            .setPositiveButton("确定", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                }
            }).show();
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, String[]
permissions, int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        ThirdPartyActivityPermissionsDispatcher.onRequestPermissionsResult
            (this, requestCode, grantResults);
    }
}

```

好了，运行此程序，点击打电话按钮，会弹出开启权限提示框，如图 1-23 所示。当我们点击“拒绝”时，会调用 `showDenied` 方法。当我们再次点击打电话按钮时，会调用 `showWhy` 方法，弹出 `AlertDialog`，如图 1-24 所示。我们点击“知道了”，会调用 `request.proceed()` 再次执行

权限请求，这时会再次弹出开启权限提示框，只不过这次的提示框带“不再询问”选项。当我们选择“不再询问”后点击“拒绝”时会调用 `showNotAsk` 方法，弹出自定义的 `AlertDialog`，如图 1-25 所示。



图 1-23 弹出开启权限提示框



图 1-24 提示用户为何要开启此权限

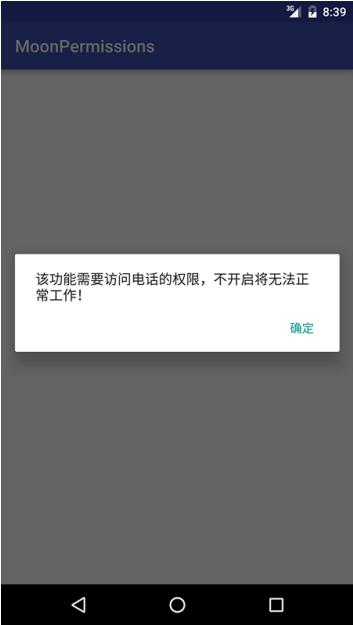


图 1-25 用户选择“不再询问”后的提示

1.3 Android 7.0 新特性

2016 年 8 月 22 日，谷歌正式推出 Android 7.0 Nougat（牛轧糖）正式版。Android 7.0 中包含一系列的新功能和改进，它们将会对 Android 系统的体验产生重大影响，例如捆绑通知和改进的多任务处理等。下面就让我们一起来看一下 Android 7.0 有哪些主要的新特性。

1.3.1 Android 7.0 主要新特性概述

1. 多窗口模式

Android 7.0 中支持多窗口多任务处理，你只要在一个应用程序中长按 Overview 按钮，就能进入多窗口模式。在大屏幕设备中，同时打开两个应用程序窗口显然可以提升执行效率，例如你可以一边网上冲浪一边发微信给自己的朋友。

2. Data Saver

Android 7.0 中引入了 Data Saver 模式，它是一种**流量保护机制**。启用 Data Saver 模式时，系统将拦截后台的数据使用，并在可能的情况下减少前台运行应用使用的数据量，例如限制流媒体服务的码率，下调画质，以及减少缓存等。而通过白名单设置，用户可以让应用避免受到 Data Saver 模式的影响。

3. 改进的 Java 8 语言支持

Android 7.0 可以支持 Java 8 语言平台，使得 Android 的 Jack 编译器现在能够有助于减少系统的冗余代码、降低占用和运行时间。开发者可以直接用 Lambda 表达式。

4. 自定义壁纸

在 Android 7.0 中，你可以为主屏幕设置壁纸，为锁屏设置另外一张壁纸。这个过程很简单，你只需要选择一张图像作为壁纸，然后就会弹出一个新的提示来让你选择是将它作为主屏壁纸，还是锁屏壁纸。

5. 快速回复

Android 7.0 还支持通知栏直接回复的功能，例如你收到一条新的 Facebook Messenger 或者来电，可以直接在通知栏进行输入操作或者接听操作。值得注意的是，这个功能不仅仅限于即时通信应用，它还适用于诸如 Twitter 这样的社交应用。

6. Daydream VR 支持

Android 7.0 内置谷歌的全新 VR 平台 Daydream。Daydream 是一个虚拟现实平台，由 Daydream 头盔、手柄和智能机构成，支持 Daydream 的智能手机要满足一定的硬件要求。

7. 后台省电

Android 7.0 在后台省电方面也做了不小的改进，屏幕关闭后所有的后台进程都将会被系统

限制活动，使这些应用不会在后台中持续唤醒，从而达到省电的目的。此外，Project Svelte 功能也在持续地改善，这最大限度地减少了 Android 设备中系统和应用所占用的内存。

8. 快速设置

下拉通知栏顶部可以展开快捷开关界面。在快捷开关界面右下角有一个“编辑”（EDIT）按钮，点击之后即可自定义添加/删除快捷开关，或拖动进行排序，如图 1-26 所示。

9. Unicode 9 支持和全新的 emoji 表情符号

Android 7.0 支持 Unicode 9，并且新增了大约 70 种 emoji 表情符号。这些表情符号大多数都是人形的，并且提供不同的肤色。

10. Google Assistant

Google Assistant 号称融合了谷歌搜索的深度学习技术以及 Google Now 的个人信息学习技术，它能够分辨用户的自然语言，并具备联系上下文的理解能力。这样你在向它发出指令时，就像和朋友聊天一样，无须迎合系统而说一些生硬且古板的语句。它能够按照你的谈话内容和习惯来调整自己的推荐建议，最终能够形成一种适合于用户本人的模式，为用户的日常生活提供帮助。

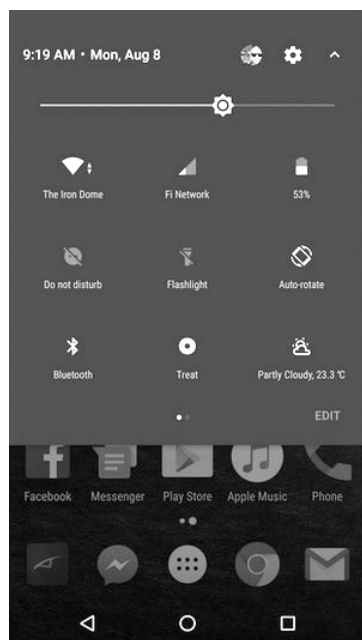


图 1-26 快速设置

1.3.2 多窗口模式

我们在操作 PC 或者是 Mac 时有时会一边看着电影，一边通过 QQ 和朋友聊天，这种多窗口操作的模式同样被引用到了 Android 系统中。在 Android 7.0 系统中加入了多窗口模式，这样我们就可以一边和朋友微信聊天，一边浏览网页了。

1. 进入多窗口模式

那么如何进入多窗口模式呢？有两种方式：

- 点击手机导航栏最右边的 Overview 按钮进入 Overview 列表，长按列表中的活动窗口并拖入到屏幕最上方的分屏提示区域。
- 打开一个程序，长按 Overview 按钮也可以进入多窗口模式。

多窗口模式的效果如图 1-27 所示。我们可以通过上下拖动两个窗口中间的白色小横线来改变两个窗口的显示区域大小。如果想要退出多窗口模式，长按 Overview 按钮就可以了。

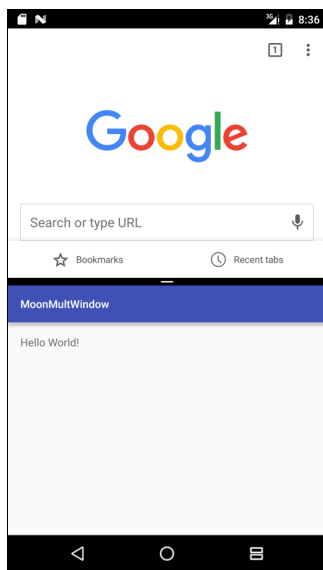


图 1-27 多窗口模式的效果

2. 多窗口模式的生命周期

我们创建 MoonMultWindow 项目，将 build.gradle 中的 targetSdkVersion 设置为 24，这时 MoonMultWindow 项目默认是支持多窗口模式的。接下来修改 MainActivity，代码如下所示：

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG="MultWindow";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d(TAG, "onCreate");
    }
    @Override
    protected void onStart() {
        super.onStart();
        Log.d(TAG, "onStart");
    }
    @Override
    protected void onResume() {
        super.onResume();
        Log.d(TAG, "onResume");
    }
    @Override
    protected void onPause() {
        super.onPause();
        Log.d(TAG, "onPause");
    }
}
```

```

@Override
protected void onStop() {
    super.onStop();
    Log.d(TAG, "onStop");
}

@Override
protected void onDestroy() {
    super.onDestroy();
    Log.d(TAG, "onDestroy");
}
}

```

在各个声明周期加上了 Log 日志。当我们长按 Overview 按钮进入多窗口模式时，会打印出如下日志：

```

D/MultWindow: onPause
D/MultWindow: onStop
D/MultWindow: onDestroy
D/MultWindow: onCreate
D/MultWindow: onStart
D/MultWindow: onResume
D/MultWindow: onPause

```

从上面可以看出 MainActivity 经历了一个重新创建的过程，最终会停留在 onPause 状态，当我们点击 MoonMultWindow 项目的窗口时，MainActivity 才会获取焦点进入 onResume 状态。接下来我们长按 Overview 按钮退出多窗口模式，查看 Log：

```

D/MultWindow: onStop
D/MultWindow: onDestroy
D/MultWindow: onCreate
D/MultWindow: onStart
D/MultWindow: onResume
D/MultWindow: onPause
D/MultWindow: onResume

```

MainActivity 经历了销毁的过程，然后是一个重新恢复的过程，最终会停留在 onResume 状态。通过上面的 Log 我们得知了多窗口模式生命周期的规则，那么此前编程的一些惯用方式也会相应地改变。比如，我们在播放视频时进入多窗口模式之际，这个视频应该还是播放的才对，因此不能在 onPause 方法中关闭视频，而是要在 onStop 方法中关闭视频，并在 onStart 方法中恢复视频的播放。

3. 禁用多窗口模式

多窗口模式未必适用于所有应用。如果想要禁用多窗口模式，只需要在 AndroidManifest.xml

中加入如下属性即可：

```
<application
    ...
    android:resizeableActivity="false"
    ...
</application>
```

如果不加此属性，则默认是支持多窗口模式的。我们将该属性设置为 `false`，这时应用就不支持多窗口模式。当我们长按 `Overview` 按钮想要进入多窗口模式时，会发现 `MoonMultWindow` 项目无法进入多窗口模式，并会弹出 `Toast` 来提醒用户当前应用不支持多窗口模式。但是需要注意的是在 `targetSdkVersion` 设置的值小于 24 时，`android:resizeableActivity` 这一属性不会起作用。面对这一情况，解决方案就是设置应用不支持横竖屏切换，比如将 `MainActivity` 设置为只支持竖屏，如下所示：

```
<activity
    android:name=".MainActivity"
    android:screenOrientation="portrait">
```

这时我们再长按 `Overview` 按钮，就会发现无法进入多窗口模式，并弹出了提醒的 `Toast`，说明上面方案的目的达到了。

1.4 本章小结

本章我们学习了 Android 5.0、Android 6.0 和 Android 7.0 的部分新特性。其中，Android 5.0 中的新特性讲得多一些。对于这些新特性，比如 `RecyclerView`，很多开发者还没有在项目中使用过它，我建议还是尽快使用它并熟悉它。另外，对于 Android 每个版本的新特性我们都要大概了解一下，并通过项目或者例子尽快地熟悉它们。本章应该算这本书中最简单的一章，后面的内容会越来越深。

第 2 章

Material Design

在第 1 章中，我们在讲到 Android 5.0 的新特性时谈到了 Material Design（质感设计），它是由 Google 推出的设计语言，旨在为手机、平板电脑等平台提供一致的设计和体验。作为一个开发者，不仅要有基本的审美能力，了解谷歌的设计语言也是十分必要的。本章我们就来学习什么是 Material Design，以及支持 Material Design 的 Design Support Library 的常用控件的使用方法。

2.1 Material Design 概述

这里只是大概地讲解 Material Design 的基本知识点，若想要了解 Material Design 的更多内容，可以查看其官方英文文档：<http://www.google.com/design/spec/material-design/>。

2.1.1 核心思想

Material Design 的核心思想，就是将物理世界中的体验带入屏幕，并且去掉物理世界中的杂质，再配合虚拟世界的灵活特性，达到最贴近真实的体验。到目前为止可以说，Material Design 是最重视跨平台体验的一套设计语言。它的规范严格细致，这样就保证了其在各个平台的使用体验高度一致，如图 2-1 所示。

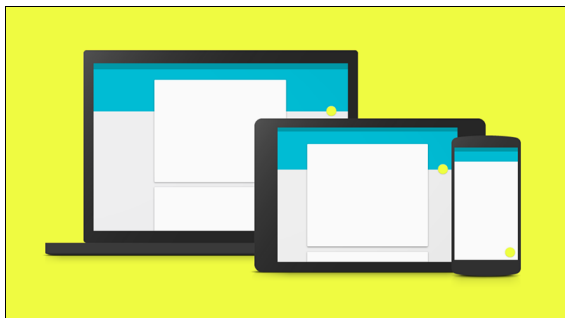


图 2-1 Material Design 的核心思想

2.1.2 材质与空间

魔法纸片是 Material Design 中最重要的信息载体。它拥有现实中的厚度、惯性和反馈，并且能够自由伸展变形。魔法纸片引入了 Z 轴的概念，Z 轴垂直于屏幕，用来表现元素的层叠关系。Z 值越高，元素离界面底层的距离越远，投影就越重，如图 2-2 所示。

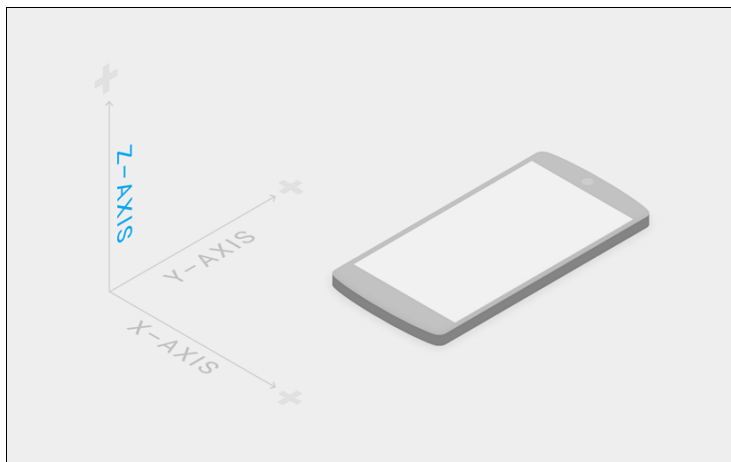


图 2-2 材质与空间

2.1.3 动画

1. 真实的动作

在物理世界中，通过观察一个物体的运动动作，我们可以感知到它的重量、惯性和大小。

在 Material Design 设计规范中，动作不仅仅要展示物体的运动轨迹，还需要展示其在空间中的关系、功能以及在整个系统中的趋势。所以在设计动画时，就需要先考虑它在现实世界中的运动规律。

2. 响应式交互

响应式交互能吸引很多用户，在用户操作一个既美观又符合常理的应用时，这会是一个很美好的体验，这会让用户产生愉悦感。响应式交互是一种有目的、非随机的、有些异想天开但不会让人分心的交互。响应式交互分为 3 种交互形式，分别是表层响应、元素响应和径向响应。

- **表层响应**——当用户点击屏幕时，系统会立即在交互的触点上绘制出一个可视化的图形让用户感知到：如在点击屏幕时会出现类似于墨水扩散那样的视觉效果形状。
- **元素响应**——元素本身也能做出交互响应，物体可以在触控或点击的时候浮起来，以表示该元素正处于激活可操作状态。比如我们长按一个应用图标时，这个应用图标会浮起来，我们可以拖动该应用图标完成位置更换或者卸载等操作。
- **径向响应**——所有的用户交互行为中都会有一个中心点，作为用户关注的中心点，当用户进行操作时应该绘制一个明显的视觉效果来让用户清晰地感知到自己的操作。

3. 转场动画

当一个界面跳转到另一个界面时，这一过程可以编排转场动画，这些动画不仅可以带来良好的视觉效果，更重要的是吸引用户的注意，比如我们可以利用转场动画引导用户做下一步的操作。转场动画不仅要提升用户体验的整体美感，还要为业务服务。

4. 细节动画

动画最基本的使用场景是过渡效果，但是最基本的动画，只要恰到好处并且足够出色，同样也能打动用户。例如一个播放器，我们根据播放状态将播放图标改为暂停图标，这种状态间的无缝切换虽说并不起眼，但也能让用户感知到。当无数个类似这样的细节和精湛的设计充满你的应用时，用户也会发现这些小细节，从而感受到设计者的诚意和用心。

2.1.4 样式

1. 色彩

色彩从当代建筑、路标、人行横道和运动场馆中获取灵感，采用了大胆的颜色表达，这与单调乏味的周边环境形成了鲜明的对比。其强调大胆的阴影和高光，从而产生了意想不到且充满活力的颜色。

2. 字体

自从 Android 4.0 发布以来,Roboto 一直都是 Android 系统的默认字体集。在 Material Design 中，为了适配更多的平台，将 Roboto 做了进一步的全面优化。宽度和圆度都有了轻微提高，从而提升了清晰度,并且看起来更令人赏心悦目。Roboto 字体如图 2-3 所示。中文字体则选用 Noto，如图 2-4 所示。

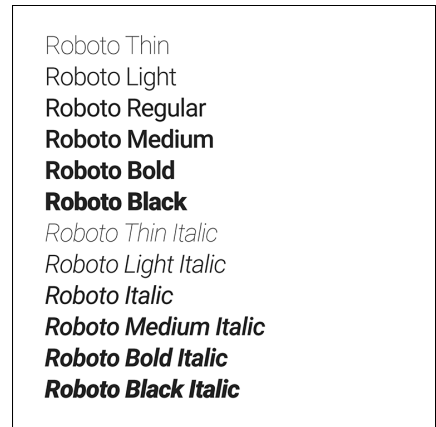


图 2-3 Roboto 字体



图 2-4 Noto 字体

3. 字体排版

一个优秀的布局不会使用过多的字体尺寸和样式。字体排版的缩放是包含了有限个字体尺

寸的集合，它们能够良好地适应布局结构。最基本的样式集合就是基于 12、14、16、20 和 34 号的字体排版缩放。这些字体尺寸是通过 SP 指定的，让大尺寸字体获得更好的可接受度。

2.1.5 图标

1. 桌面图标

桌面图标作为 App 的门面，传达了产品的核心理念与内涵。虽然每个产品的桌面图标都不相同，但对于一个特定的品牌，桌面图标应在理念和实践中统一设计。桌面图标建议模仿现实中的折纸效果，通过扁平色彩表现空间和光影，一般情况下请使用 $48\text{dp} \times 48\text{dp}$ 的尺寸，必要时可以放大到 $192\text{dp} \times 192\text{dp}$ 。

2. 小图标

优先使用 Material Design 默认图标。设计小图标时，使用最简练的图形来表达，图形不要带空间感。小图标尺寸是 $24\text{dp} \times 24\text{dp}$ ，图形限制在中央 $20\text{dp} \times 20\text{dp}$ 区域内。

2.1.6 图像

在 Material Design 中，图像都不应该是人为策划的，而是组建生成的，不要给用户一种过度制作的感觉。这种风格比较强调场景的实质性、质感和深度，让人意想不到的色彩运用，以及对环境背景的关注。这些原则都旨在创建目的性强、美观、具有深度的用户界面，如图 2-5 所示。

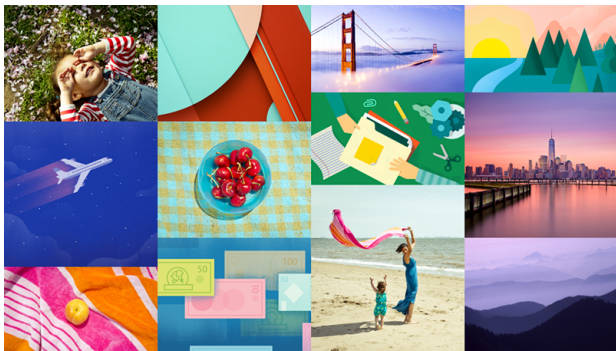


图 2-5 图像

布局

布局设计使用相同的视觉元素、结构网格和通用的行距规则，这样会使得 App 在不同平台与屏幕尺寸上都拥有一致的外观和体验。这样就创造了一个识别度高的跨平台产品的用户环境，这个环境会为用户提供高度的熟悉感和舒适性。

2.1.7 组件

1. 底部动作条 (Bottom Sheets)

底部动作条是一个从屏幕底部边缘向上滑出的面板，旨在向用户呈现一组功能。底部动作条呈现了简单、清晰的一组操作，如图 2-6 所示。

2. 卡片 (Cards)

卡片是包含一组特定数据集合的纸片，数据集合中包含了各种相关信息，例如，主题的照片、文本和链接。卡片通常用于展示一些基础和概述的信息，并且作为通往更详细信息的入口。卡片有固定的宽度和可变的高度，其最大高度限制于可适应平台上单一视图的内容；但如果需要它，可以临时扩展，如图 2-7 所示。

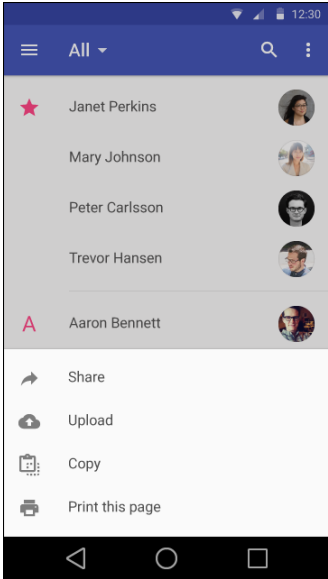


图 2-6 底部动作条

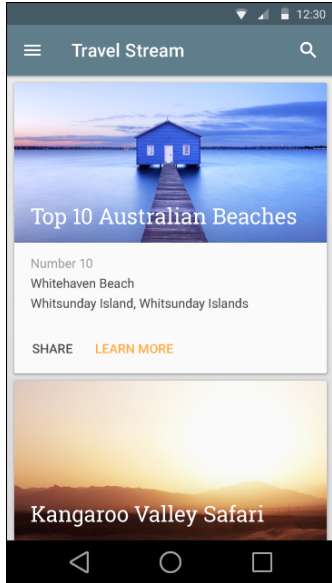


图 2-7 卡片

3. 提示框 (Dialogs)

提示框用于向用户提示一些信息，或者是一些决定的选择。提示框可以采用一种“取消/确定”的简单应答模式，也可以采用自定义布局的复杂模式，比如说增加一些文本设置或者是文本输入。

4. 菜单 (Menus)

菜单能够影响到应用、视图或者视图中选中的按钮。它由按钮、动作、点或者包含至少两个菜单项的其他控件触发。每一个菜单项是一个选项或者是动作，菜单不应该用作应用中主要的导航方法，如图 2-8 所示。

5. 选择器

选择器提供了一个简单的方法来从一个预定义的集合中选取单个值。最常用的例子就是用户要输入日期的场景，这时使用时间选择器可以帮你保证用户指定的日期是正确格式化的。

6. 滑块控件 (Sliders)

当我们调节音乐音量或者音乐的播放进度时会使用滑块控件，它让我们在一个区间内滑动锚点来选择一个合适的数值。区间最小值放在左边，最大值放在右边。

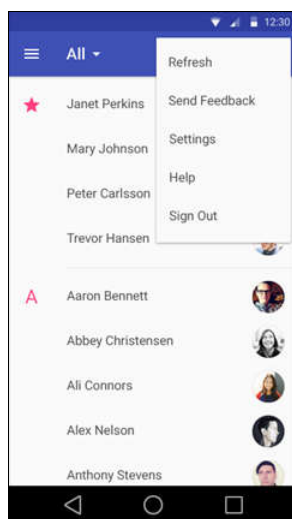


图 2-8 菜单

7. 进度和动态

在用户与内容进行交互之前，为了给用户展现出更完整的、变化更少的界面，需要使用进度指示器，并且要尽量使加载的过程令人愉快。每次加载操作只能由一个进度指示器呈现，例如，对于刷新操作，你不能既用刷新条，又用动态圆圈来指示。

8. Snackbar 与 Toast

Snackbar 是一种轻量级的弹出框，通常显示在屏幕的底部并在屏幕所有层的最上方，包括浮动操作按钮。它会在超时或者用户在屏幕其他地方触摸之后自动消失。**Snackbar** 可以在屏幕上滑动关闭，**Snackbar** 不会阻碍用户在屏幕上的输入，并且也不支持输入，屏幕上同时最多只能显示一个 **Snackbar**。Android 也提供了一种主要用于提示系统消息的胶囊状的提示框 **Toast**。**Toast** 同 **Snackbar** 非常相似，但是 **Toast** 并不包含操作，也不能从屏幕上滑动关闭。

9. Tab

在一个 App 中，**Tab** 使得在不同的视图和功能间切换更加简单。**Tab** 用来显示有关联的分组内容，**Tab** 标签则用来简要地描述 **Tab** 的内容，如图 2-9 所示。

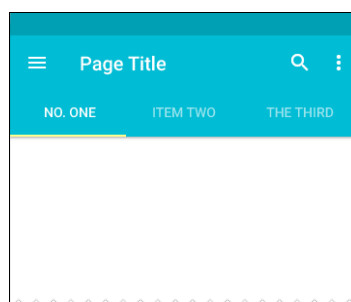


图 2-9 Tab

2.2 Design Support Library 常用控件详解

谷歌推出 Android 5.0 之际，在带来了更加详细的 Material Design 设计规范的同时也推出了全新的 Android Design Support Library，这个兼容库很容易和之前的 Android Support Library 22.1 混淆。它们都是兼容库，二者的区别是这个库多了一些 Material Design 设计风格的控件。本节我们就来学习一下如何使用这些 Material Design 设计风格的控件。

2.2.1 Snackbar 的使用

为一个操作提供轻量级的、快速的反馈是使用 Snackbar 的最好时机。Snackbar 显示在屏幕的底部，包含了文字信息与一个可选的操作按钮。它在指定时间结束之后自动消失。另外，配合 CoordinatorLayout 使用，还可以在超时之前将它滑动删除。

首先我们要配置 build.gradle:

```
dependencies {  
    . . .  
    compile 'com.android.support:design:23.2.0'  
    compile 'com.android.support:appcompat-v7:23.4.0'  
}
```

接下来我们定义一个按钮，点击的时候弹出 Snackbar，如下代码所示：

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    activity_main = (RelativeLayout) this.findViewById(R.id.activity_main);  
    bt_snackbar = (Button) this.findViewById(R.id.bt_snackbar);  
    bt_snackbar.setOnClickListener(new View.OnClickListener() {  
        @Override  
        public void onClick(View v) {  
            showSnackbar();  
        }  
    });  
}  
  
private void showSnackbar() {  
    Snackbar.make(activity_main, "标题", Snackbar.LENGTH_LONG)  
        .setAction("点击事件", new View.OnClickListener() {  
            @Override  
            public void onClick(View v) {  
                Toast.makeText(  

```

```

        MainActivity.this,
        "Toast",
        Toast.LENGTH_SHORT).show();
    }
    }).setDuration(Snackbar.LENGTH_LONG).show();
}
}

```

需要注意的是 `Snackbar` 的 `make` 方法的第一个参数是一个 `View` 类型的参数，这个 `View` 类型的参数是 `Snackbar` 的父控件，所以选择合适的父控件以确保 `Snackbar` 能显示在正确的位置上。另外，这个 `Snackbar` 界面包含了两个元素：一个用来显示文字，另一个用来设置点击事件。当我们点击“点击事件”按钮时会弹出 `Toast`，如图 2-10 所示。

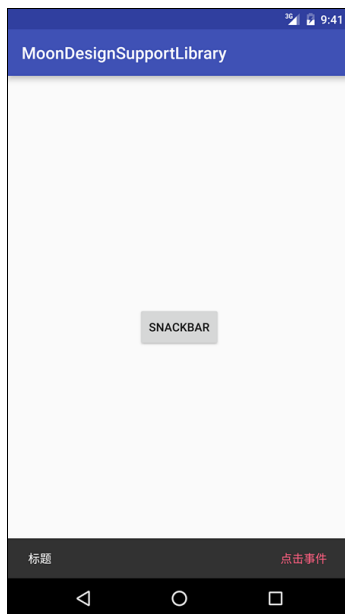


图 2-10 Snackbar

2.2.2 用 TextInputLayout 实现登录界面

1. 实现布局

现在要用 `TextInputLayout` 来实现一个简单的登录界面。我们首先用常规方法使用 `EditText` 来实现文本框。

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_text_input_layout"

```



```

        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin">

        <EditText
            android:id="@+id/et_username"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_centerInParent="true"
            android:hint="username"
            android:maxLength="25"
            android:maxLines="1" />

        <EditText
            android:id="@+id/et_password"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_below="@id/et_username"
            android:layout_centerHorizontal="true"
            android:layout_marginTop="20dp"
            android:hint="password"
            android:maxLength="25"
            android:maxLines="1" />

        <Button
            android:id="@+id/bt_login"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_below="@id/et_password"
            android:layout_centerHorizontal="true"
            android:layout_marginTop="4dp"
            android:text="登录" />
    </RelativeLayout>

```

运行程序效果如图 2-11 所示。很明显，按照常规的做法不会有什么不同，在这里需要留意一下，我们设置了 EditText 的 hint 值，其在我们未输入时会显示出来，输入的时候就会消失。接下来我们引入 TextInputLayout。

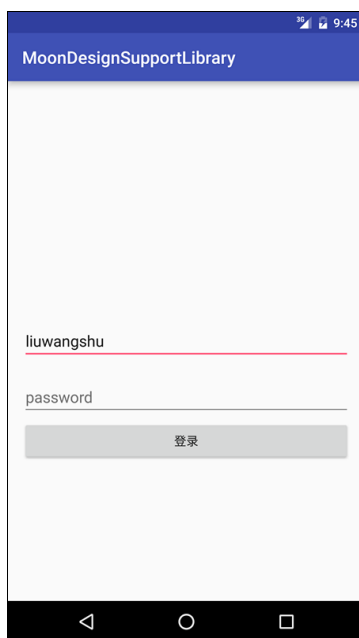


图 2-11 用 EditText 实现登录界面

2. 使用 TextInputLayout

TextInputLayout 控件是一个容器，它跟 ScrollView 一样只接受一个子元素，并且这个子元素是 EditText。代码如下所示：

```
<android.support.design.widget.TextInputLayout
    android:id="@+id/tl_username"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true">

    <EditText
        android:id="@+id/et_username"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="username"
        android:maxLength="25"
        android:maxLines="1" />
</android.support.design.widget.TextInputLayout>

<android.support.design.widget.TextInputLayout
    android:id="@+id/tl_password"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/tl_username">
```

```

        android:layout_centerHorizontal="true"
        android:layout_centerInParent="true"
        android:layout_marginTop="20dp">

        <EditText
            android:id="@+id/et_password"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_below="@id/tl_username"
            android:hint="password"
            android:maxLength="25"
            android:maxLines="1" />
    </android.support.design.widget.TextInputLayout>

```

运行程序发现我们设置的 hint 值不在原本的 EditText 中，而是浮在上方，如图 2-12 所示。当我们将焦点移动到输入 password 的输入框时惊喜出现了：username 的输入框上浮的文字回到原本的 EditText 中并伴有动画，而 password 的输入框中的 hint 值却上浮到了 EditText 之上并伴有动画。焦点移动到 password 输入框时的样式如图 2-13 所示。

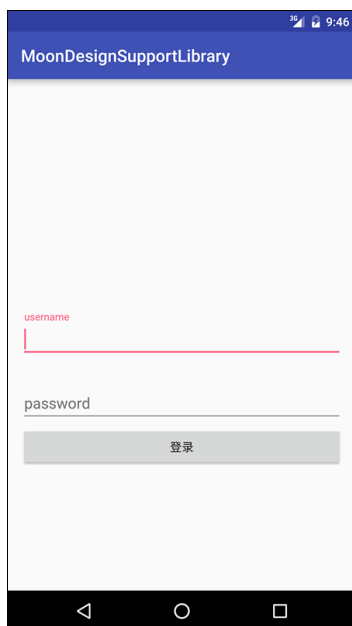


图 2-12 TextInputLayout 效果 1



图 2-13 TextInputLayout 效果 2

3. 显示错误信息

除了能更加友好地显示提示这一个优点之外，TextInputLayout 还有另一个优点，那就是可以友好地显示错误信息。在这个例子里，如果我们要设置对用户名和密码进行格式、字符数验证，若验证出错，则一般都会弹出 Toast 进行提示。TextInputLayout 则带给我们另一种方式。首

先我们要对用户名进行邮箱格式验证，对密码进行字符数验证。对代码字符数的验证如下所示：

```
private boolean validatePassword(String password) {
    return password.length() > 6;
}
```

为了验证效果，只是简单地验证了密码的长度。接下来对邮箱格式进行验证，用到了正则表达式。

```
Private static final String
EMAIL_PATTERN= "^[a-zA-Z0-9#_~!$&'()*+,-;=:.\\\"() , : ; < > @ \\ [ \\ ] \\ \\ \\ ] + @
[a-zA-Z0-9-]+ (\\. [a-zA-Z0-9-]+) * $";
private Pattern pattern = Pattern.compile(EMAIL_PATTERN);
private boolean validateUserName(String username) {
    matcher = pattern.matcher(username);
    return matcher.matches();
}
```

验证用户名和密码的方法写完后，我们就在点击登录的时候调用 login()方法，代码如下所示：

```
private void login() {
    String username=tl_username.getText().toString();
    String password=tl_password.getText().toString();
    if(!validateUserName(username)) {
        tl_username.setErrorEnabled(true);
        tl_username.setError("请输入正确的邮箱地址");
    } else if(!validatePassword(password)) {
        tl_password.setErrorEnabled(true);
        tl_password.setError("密码字数过少");
    } else {
        tl_username.setErrorEnabled(false);
        tl_password.setErrorEnabled(false);
        Toast.makeText(getApplicationContext(), "登录成功", Toast.LENGTH_SHORT).show();
    }
}
```

我们可以通过 tl_username.getText().toString()来获取用户输入的信息，然后用前面定义的 validateUserName()方法和 validatePassword()方法分别验证用户名和密码。需要注意的是，TextInputLayout 的 setErrorEnabled(boolean enabled)方法是用于显示和隐藏错误提示的，如果有错误则设置为 true，没有错误则设置为 false。我们运行程序，运行效果如图 2-14 所示。这个时候若输入了不符合 Email 地址格式的字符串，就会在输入框的下方显示错误提示。而这个错误提示可以通过 TextInputLayout.setError()来实现。

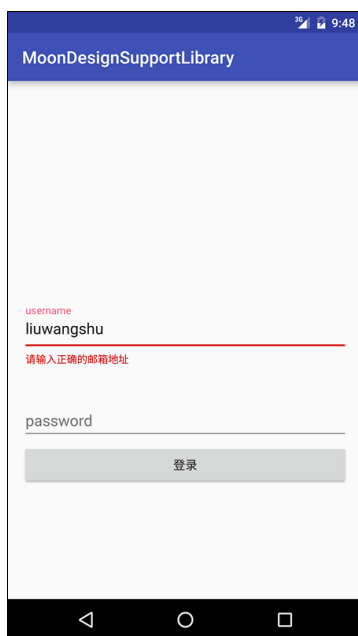


图 2-14 显示错误信息

4. 改变样式

如果对 `TextInputLayout` 输入框以及输入框上方的 hint 提示颜色不满意,则还可以在 `style.xml` 文件中对 `colorAccent` 属性进行修改。

```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

2.2.3 FloatingActionButton 的使用

它是一个负责显示界面基本操作的圆形按钮。其使用起来非常便捷,可以把它当作一个 `Button`, 代码如下所示:

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
```

```
android:layout_alignParentRight="true"
android:layout_margin="20dp"
android:src="@drawable/ic_record"
android:clickable="true"
app:backgroundTint="#3F51B5"
app:elevation="3dp"
app:pressedTranslationZ="6dp"
/>
```

由于父布局是 `RelativeLayout`, 所以这里用了 `layout_alignParentBottom` 和 `layout_alignParentRight` 两个属性。`FloatingActionButton` 的填充色默认也是 `style.xml` 文件 `colorAccent` 设定的颜色, 当然也可以通过 `app:backgroundTint` 来设定背景填充色。`app:elevation` 用来设置正常状态的阴影大小, `app:pressedTranslationZ` 用来设置点击时阴影的大小, 效果如图 2-15 所示。

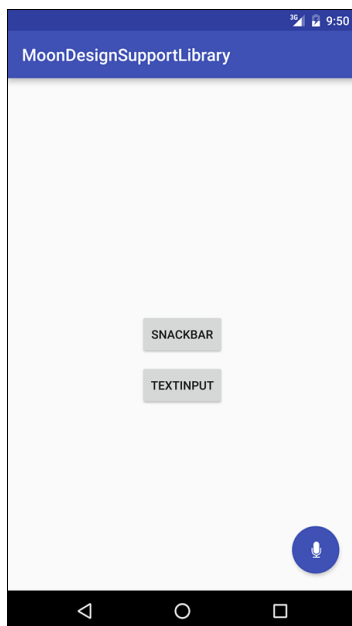


图 2-15 FloatingActionButton 的使用

2.2.4 用 TabLayout 实现类似网易选项卡的动态滑动效果

此前我们要实现类似网易选项卡的动态效果稍微有些麻烦, 还需要动态地加载布局等技术, 控制滑动的时候还需要用 `HorizontalScrollView`。这一次 `Design Support Library` 给我们带来了 `TabLayout`, 故此可以很轻松地实现这一效果。本节我们就来学习一下如何用 `TabLayout` 来实现类似网易选项卡的动态滑动效果。

1. 配置 build.gradle

这里要配置 `build.gradle`, 如下所示:

```
dependencies {
    . . .
    compile 'com.android.support:appcompat-v7:22.2.0'
    compile 'com.android.support:design:22.2.0'
    compile 'com.android.support:recyclerview-v7:22.2.0'
    compile 'com.android.support:cardview-v7:22.2.0'
}
```

在这个例子中需要使用 `recyclerview` 和 `cardview`，所以这里引用了 `com.android.support:recyclerview` 和 `com.android.support:cardview`。

2. 主界面的布局

下面看看主界面的布局是怎样的，代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".TabLayoutActivity"
    android:orientation="vertical">
    <android.support.design.widget.AppBarLayout
        android:id="@+id/appbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">
        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_scrollFlags="scroll|enterAlways"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>

        <android.support.design.widget.TabLayout
            android:id="@+id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:tabIndicatorColor="#ADBE107E"
            app:tabMode="scrollable"/>

    </android.support.design.widget.AppBarLayout>

    <android.support.v4.view.ViewPager
        android:id="@+id/viewpager"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
```

```
app:layout_behavior="@string/appbar_scrolling_view_behavior"/>

</LinearLayout>
```

这里用到了 `AppBarLayout` 和 `Toolbar`, `AppBarLayout` 是 `Android Design Support Library` 新加的控件, 继承自 `LinearLayout`, 它用来将 `Toolbar` 和 `TabLayout` 组合起来作为一个整体。`Toolbar` 在第 1 章我们已经讲过了, 这里就不做介绍了。该布局文件最关键的一点就是 `android.support.design.widget.TabLayout` 标签中的 `app:tabMode="scrollable"`, 其设置 `Tab` 的模式为“可滑动的”。另外还可以设置 `app:tabMode="fixed"`, 表示固定不可滑动的。如果我们将 `app:tabMode` 这一属性去掉, 或者设置 `app:tabMode="fixed"`, 运行程序就会如图 2-16 所示。`Tab` 的数量太多了 (13 个), 由于不能够滑动, 所以选项就会彼此压缩, 这样的效果显然是不行的。我们设置 `app:tabMode="scrollable"` 来看看效果, 如图 2-17 所示。这样 `Tab` 的样式就正常了, 而且还能够滑动。



图 2-16 压缩的 Tab 选项



图 2-17 正常的 Tab 选项

3. Java 代码

接下来我们在 `Activity` 中引用这个布局文件。

```
public class TabLayoutActivity extends AppCompatActivity {
    private DrawerLayout mDrawerLayout;
    private ViewPager mViewPager;
    private TabLayout mTabLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```



```

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_tab_layout);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        mViewPager = (ViewPager) findViewById(R.id.viewpager);
        initViewPager();
    }

```

initViewPager()方法的代码如下所示:

```

private void initViewPager() {
    mTabLayout = (TabLayout) findViewById(R.id.tabs);
    List<String> titles = new ArrayList<>();
    titles.add("精选");
    titles.add("体育");
    titles.add("巴萨");
    titles.add("购物");
    titles.add("明星");
    titles.add("视频");
    titles.add("健康");
    titles.add("励志");
    titles.add("图文");
    titles.add("本地");
    titles.add("动漫");
    titles.add("搞笑");
    titles.add("精选");

    for (int i = 0; i < titles.size(); i++) {
        mTabLayout.addTab(mTabLayout.newTab().setText(titles.get(i)));
    }
    List<Fragment> fragments = new ArrayList<>();
    for (int i = 0; i < titles.size(); i++) {
        fragments.add(new ListFragment());
    }
    FragmentAdapter mFragmentAdapteradapter =
        new FragmentAdapter(getSupportFragmentManager(), fragments,
            titles);
    //给 ViewPager 设置适配器
    mViewPager.setAdapter(mFragmentAdapteradapter);
    //将 TabLayout 和 ViewPager 关联起来
    mTabLayout.setupWithViewPager(mViewPager);
    //给 TabLayout 设置适配器
    mTabLayout.setTabsFromPagerAdapter(mFragmentAdapteradapter);
}

```

在这里我们设定了 13 个标题内容并创建了相应的 TabLayout 和 Fragment, 设置了 ViewPager 适配器和 TabLayout 适配器, 并将 TabLayout 和 ViewPager 关联起来。这里我们用自己创建的 ListFragment 作为每个 Tab 的界面, 接下来看看 ListFragment 是如何定义的, 代码如下所示:

```
public class ListFragment extends Fragment {
    private RecyclerView mRecyclerView;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        mRecyclerView =
            (RecyclerView) inflater.inflate(R.layout.list_fragment, container, false);
        return mRecyclerView;
    }
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        mRecyclerView.setLayoutManager(new
            LinearLayoutManager(mRecyclerView.getContext()));
        mRecyclerView.setAdapter(new RecyclerViewAdapter(getActivity()));
    }
}
```

这里用 RecyclerView 来代替 ListView, 来看看 RecyclerViewAdapter 的代码。

```
public class RecyclerViewAdapter extends RecyclerView.Adapter
    <RecyclerViewAdapter.ViewHolder> {
    private Context mContext;
    public RecyclerViewAdapter(Context mContext) {
        this.mContext = mContext;
    }
    @Override
    public RecyclerViewAdapter.ViewHolder onCreateViewHolder(ViewGroup
        parent, int viewType) {
        View view =
            LayoutInflater.from(parent.getContext()).inflate
            (R.layout.list_item_card_main, parent, false);
        return new ViewHolder(view);
    }
    @Override
    public void onBindViewHolder(final RecyclerViewAdapter.ViewHolder
        holder, int position) {
        final View view = holder.mView;
        view.setOnClickListener(new View.OnClickListener() {
```

```

        @Override
        public void onClick(View v) {
            Toast.makeText(mContext, "奔跑在孤傲的路上", Toast.LENGTH_
                SHORT).show();
        }
    });
}
@Override
public int getItemCount() {
    return 10;
}
public static class ViewHolder extends RecyclerView.ViewHolder {
    public final View mView;
    public ViewHolder(View view) {
        super(view);
        mView = view;
    }
}
}
}

```

在代码中为 item 选项设置了点击事件响应，并将 item 条目数设置为 10 个。最后来看看 FragmentAdapter 的代码。

```

public class FragmentAdapter extends FragmentStatePagerAdapter {
    private List<Fragment> mFragments;
    private List<String> mTitles;

    public FragmentAdapter(FragmentManager fm, List<Fragment> fragments,
        List<String> titles) {
        super(fm);
        mFragments = fragments;
        mTitles = titles;
    }

    @Override
    public Fragment getItem(int position) {
        return mFragments.get(position);
    }

    @Override
    public int getCount() {
        return mFragments.size();
    }
}

```

```

@Override
public CharSequence getPageTitle(int position) {
    return mTitles.get(position);
}
}

```

FragmentAdapter 主要处理了两件事：一件事就是根据不同的 position 来返回不同的 Fragment；另一件事就是根据不同的 position 来返回不同的 title。基本所有的代码都讲到了，当然这种稍微复杂的效果 TabLayout 能够实现，那简单的 3、4 个 Tab 滑动，TabLayout 实现起来就更不在话下了。修改 TabLayoutActivity 的 initViewPager 方法，代码如下所示：

```

private void initViewPager() {
    mTabLayout = (TabLayout) findViewById(R.id.tabs);
    List<String> titles = new ArrayList<>();
    titles.add("精选");
    titles.add("体育");
    titles.add("巴萨");
    titles.add("购物");
    for(int i=0;i<titles.size();i++){
        mTabLayout.addTab(mTabLayout.newTab().setText(titles.get(i)));
    }
    List<Fragment> fragments = new ArrayList<>();
    for(int i=0;i<titles.size();i++){
        fragments.add(new ListFragment());
    }
    FragmentAdapter mFragmentAdapteradapter =
        new FragmentAdapter(getSupportFragmentManager(), fragments,
            titles);
    //给 ViewPager 设置适配器
    mViewPager.setAdapter(mFragmentAdapteradapter);
    //将 TabLayout 和 ViewPager 关联起来
    mTabLayout.setupWithViewPager(mViewPager);
    //给 TabLayout 设置适配器
    mTabLayout.setTabsFromPagerAdapter(mFragmentAdapteradapter);
}

```

我们只保留了 4 个 Tab，然后设置 app:tabMode="fixed"，运行代码来看看效果，如图 2-18 所示。如果觉得底部指示器不符合心意，还可以用 tabIndicatorHeight 来设置底部指示器的高度，用 tabIndicatorColor 来设置底部指示器的颜色等。



图 2-18 4 个 Tab 选项

2.2.5 用 NavigationView 实现抽屉菜单界面

对于抽屉界面我们都不陌生，此前写这种抽屉界面都需要我们自定义。自己写的话，花点时间也能做出好看的侧拉菜单，但总是要耗费时间，于是 Design Support Library 提供了 `NavigationView` 来帮助我们实现抽屉菜单界面。和普通的侧拉菜单实现方式一样，所有的东西还是都放在一个 `DrawerLayout` 中，用 `NavigationView` 来替代我们此前自定义的控件。我们还是在 2.2.4 节代码的基础上进行修改。首先我们要修改主界面布局，代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.
    android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/dl_main_drawer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

    <LinearLayout
        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:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
```

```

tools:context=".TabLayoutActivity">

<android.support.design.widget.AppBarLayout
    android:id="@+id/appbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        app:layout_scrollFlags="scroll|enterAlways"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

    <android.support.design.widget.TabLayout
        android:id="@+id/tabs"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:tabIndicatorColor="#ADBE107E"
        app:tabMode="scrollable" />

</android.support.design.widget.AppBarLayout>

<android.support.v4.view.ViewPager
    android:id="@+id/viewpager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />

</LinearLayout>

<android.support.design.widget.NavigationView
    android:id="@+id/nv_main_navigation"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:headerLayout="@layout/navigation_header"
    app:menu="@menu/drawer_view" />
</android.support.v4.widget.DrawerLayout>

```

DrawerLayout 标签包含了主界面的布局以及抽屉的布局，这个抽屉界面就是 NavigationView，NavigationView 标签下的 app:headerLayout="" 可以引入头部文件，app:menu=""

则引入菜单的布局。我们查看引入的头部文件 `navigation_header.xml`。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="150dp"
    android:background="?attr/colorPrimaryDark"
    android:orientation="horizontal"
    android:theme="@style/ThemeOverlay.AppCompat.Dark">

    <ImageView
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_gravity="center_vertical"
        android:layout_marginLeft="50dp"
        android:background="@drawable/ic_user" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_marginLeft="10dp"
        android:text="Liuwangshu"
        android:textAppearance="@style/TextAppearance.AppCompat.Body1"
        android:textSize="20sp" />

</LinearLayout>
```

头部文件很简短，其包含了一个 `ImageView` 用来显示图片，以及一个 `TextView` 用来显示文字。接下来看看 `app:menu="@menu/drawer_view"`，引入菜单文件 `drawer_view.xml`，代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_home"
            android:icon="@drawable/ic_dashboard"
            android:title="首页" />
        <item
            android:id="@+id/nav_messages"
            android:icon="@drawable/ic_event"
            android:title="事项" />
    </group>
</menu>
```

```

        android:id="@+id/nav_friends"
        android:icon="@drawable/ic_headset"
        android:title="音乐" />
    <item
        android:id="@+id/nav_discussion"
        android:icon="@drawable/ic_forum"
        android:title="消息" />
    </group>
</menu>

```

菜单元素放在 `group` 标签之下，同时用 `<group android:checkableBehavior="single">` 声明每次只能有一个 `item` 被选中。在 `menu` 标签中包含了 4 个 `item`，每个 `item` 又包含了 `icon` 和 `title`。程序效果如图 2-19 所示。



图 2-19 没有分组的菜单

我们还可以通过为 `item` 添加子菜单来实现带有头部的分组效果，每添加一个分组，都会在该组的最上面自动添加一根分割线，以和普通的 `item` 进行区分。

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_home"
            android:icon="@drawable/ic_dashboard"
            android:title="首页" />
        <item
            android:id="@+id/nav_messages"
            android:icon="@drawable/ic_event"
            android:title="事项" />
        <item

```



```

        android:id="@+id/nav_friends"
        android:icon="@drawable/ic_headset"
        android:title="音乐" />
    <item
        android:id="@+id/nav_discussion"
        android:icon="@drawable/ic_forum"
        android:title="消息" />
</group>

<item android:title="其他">
    <menu>
        <item
            android:icon="@drawable/ic_dashboard"
            android:title="设置" />
        <item
            android:icon="@drawable/ic_dashboard"
            android:title="关于我们" />
    </menu>
</item>
</menu>

```

在原有代码的基础上，我们添加了分组标题“其他”，效果如图 2-20 所示。



图 2-20 加入分组的菜单

接下来在 Java 代码中引用布局。

```
@Override
```

```

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_tab_layout);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    final ActionBar ab = getSupportActionBar();
    ab.setHomeAsUpIndicator(R.drawable.ic_menu);
    ab.setDisplayHomeAsUpEnabled(true);
    mViewPager = (ViewPager) findViewById(R.id.viewpager);
    mDrawerLayout = (DrawerLayout) findViewById(R.id.dl_main_drawer);
    NavigationView navigationView =
        (NavigationView) findViewById(R.id.nv_main_navigation);
    if (navigationView != null) {
        navigationView.setNavigationItemSelectedListener(
            new NavigationView.OnNavigationItemSelectedListener() {
                @Override
                public boolean onNavigationItemSelected(MenuItem menuItem) {
                    menuItem.setChecked(true);
                    String title = menuItem.getTitle().toString();
                    Toast.makeText(getApplicationContext(), title, Toast.
                        LENGTH_SHORT).show();
                    //关闭导航菜单
                    mDrawerLayout.closeDrawers();
                    return true;
                }
            });
    }
    initViewPager();
}

```

我们用 `setNavigationItemSelectedListener()` 方法来设置当导航项被点击时的回调。在回调方法 `onNavigationItemSelected(MenuItem menuItem)` 中会向我们提供被选中的 `MenuItem`，我们得到这个 `MenuItem` 对象就可以处理 `MenuItem` 相关的功能，比如获取该 `MenuItem` 的标题等。另外，通过这个回调我们也可以改变 item 选中状态、关闭导航菜单，以及其他我们需要的操作。当然，别忘了对 `Toolbar` 的菜单选项进行监听回调，否则抽屉就出不来了。具体代码如下所示：

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case android.R.id.home:
            mDrawerLayout.openDrawer(GravityCompat.START);
            return true;
    }
}

```

```

        return super.onOptionsItemSelected(item);
    }

```

2.2.6 用 CoordinatorLayout 实现 Toolbar 隐藏和折叠

CoordinatorLayout 是 Android Design Support Library 中比较难的控件。顾名思义，它是用来组织其子 View 之间协作的一个父 View。CoordinatorLayout 默认情况下可被理解为一个 FrameLayout，它的布局方式默认是一层一层叠上去的，在此会介绍一下它最常用的两种情况。

1. CoordinatorLayout 实现 Toolbar 隐藏效果

本节的例子仍旧是在 2.2.4 节代码的基础上进行修改的。我们仍旧从布局文件讲起，最外层我们用 CoordinatorLayout 来做整体的布局，AppBarLayout 将 Toolbar 和 TabLayout 整合成了一个整体，代码如下所示：

```

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

<android.support.design.widget.CoordinatorLayout xmlns:android="http://
schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:id="@+id/main_content"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:fitsSystemWindows="true">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/appbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_scrollFlags="scroll|enterAlways"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

        <android.support.design.widget.TabLayout
            android:id="@+id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:tabIndicatorColor="#ADBE107E"
            app:tabMode="scrollable" />
    
```

```

</android.support.design.widget.AppBarLayout>

<android.support.v4.view.ViewPager
    android:id="@+id/viewpager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />

<android.support.design.widget.FloatingActionButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="10dp"
    android:clickable="true"
    android:onClick="checkin"
    android:src="@drawable/ic_discuss"
    app:layout_anchor="@id/main_content"
    app:layout_anchorGravity="bottom|right|end" />
</android.support.design.widget.CoordinatorLayout>

```

能隐藏的关键是 `app:layout_scrollFlags="scroll|enterAlways"` 这个属性，设置滚动事件，属性里面必须至少启用 `scroll` 这个 `flag`，这样这个 `View` 才会滚动出屏幕，否则它将一直固定在顶部。接下来看看 Java 代码的引用，如下所示：

```

public class CoordinatorLayoutActivity extends AppCompatActivity {
    private ViewPager mViewPager;
    private TabLayout mTabLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_tab_layout);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        final ActionBar ab = getSupportActionBar();
        ab.setDisplayHomeAsUpEnabled(true);
        mViewPager = (ViewPager) findViewById(R.id.viewpager);
        initViewPager();
    }

    private void initViewPager() {

        mTabLayout = (TabLayout) findViewById(R.id.tabs);
        List<String> titles = new ArrayList<>();
        titles.add("精选");
    }
}

```

```

        titles.add("体育");
        titles.add("巴萨");
        titles.add("购物");
        titles.add("明星");
        titles.add("视频");
        titles.add("健康");
        titles.add("励志");
        titles.add("图文");
        titles.add("本地");
        titles.add("动漫");
        titles.add("搞笑");
        titles.add("精选");
        for (int i = 0; i < titles.size(); i++) {
            mTabLayout.addTab(mTabLayout.newTab().setText(titles.get(i)));
        }
        List<Fragment> fragments = new ArrayList<>();
        for (int i = 0; i < titles.size(); i++) {
            fragments.add(new ListFragment());
        }
        FragmentAdapter mFragmentAdapteradapter =
            new FragmentAdapter(getSupportFragmentManager(), fragments,
                titles);
        //给 ViewPager 设置适配器
        mViewPager.setAdapter(mFragmentAdapteradapter);
        //将 TabLayout 和 ViewPager 关联起来
        mTabLayout.setupWithViewPager(mViewPager);
        //给 TabLayout 设置适配器
        mTabLayout.setTabsFromPagerAdapter(mFragmentAdapteradapter);

    }

    public void checkin(View view) {
        Snackbar.make(view, "点击成功", Snackbar.LENGTH_SHORT).show();
    }

}

```

主界面的 Java 代码和 2.2.4 节的主界面代码并没有什么区别，我们运行程序，默认的效果如图 2-21 所示。当我们的手指向上滑动屏幕时，随着手指的滑动 Toolbar 会逐渐消失滑进屏幕上方，如图 2-22 所示。我们在这里还用到了上面讲到的 Snackbar 和 FloatingActionButton，点击 FloatingActionButton 会触发 checkin() 方法，这就会弹出 Snackbar。至于为什么这里要用到 FloatingActionButton 和 Snackbar，是因为 FloatingActionButton 和 CoordinatorLayout 结合起来会有一个特殊的效果，如图 2-23 所示。当我们点击 FloatingActionButton 弹出 Snackbar 的时候，

为了给 Snackbar 留出空间，浮动的 FloatingActionButton 会向上移动。这是因为配合 CoordinatorLayout, FloatingActionButton 有一个默认的 Behavior 来检测 Snackbar 的添加并让按钮在 Snackbar 之上呈现上移与 Snackbar 等高的动画。关于 Behavior 属性，我们会在后面讲到。其他的代码请参照 2.2.4 节，这里就不赘述了。



图 2-21 默认展开的样式



图 2-22 向上滑动的样式



图 2-23 浮动的 FloatingActionButton

2. CoordinatorLayout 结合 CollapsingToolbarLayout 实现 Toolbar 折叠效果

要实现折叠效果，我们需要引入一个新的布局 CollapsingToolbarLayout，其作用是提供一个可以折叠的 Toolbar。CollapsingToolbarLayout 继承自 FrameLayout。CollapsingToolbarLayout 设置 layout_scrollFlag 属性，可以控制包含在 CollapsingToolbarLayout 中的控件，比如 imageView、Toolbar 在响应 layout_behavior 事件时做出相应的 scrollFlags 滚动事件。在布局文件中用 CollapsingToolbarLayout 将 ImageView 和 Toolbar 包含起来作为一个可折叠的 Toolbar，再用 AppBarLayout 包裹起来作为一个 AppBar 的整体。当然，AppBarLayout 目前必须是第一个嵌套在 CoordinatorLayout 里面的子 View。布局文件代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout xmlns:android="http://
schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:id="@+id/main_content"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:fitsSystemWindows="true">

<android.support.design.widget.AppBarLayout
    android:id="@+id/appbar"
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:fitsSystemWindows="true"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

<android.support.design.widget.CollapsingToolbarLayout
    android:id="@+id/collapsing_toolbar"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    app:contentScrim="?attr/colorPrimary"
    app:expandedTitleMarginEnd="64dp"
    app:expandedTitleMarginStart="48dp"
    app:layout_scrollFlags="scroll|exitUntilCollapsed">

<ImageView
    android:id="@+id/backdrop"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    android:scaleType="centerCrop"
    android:src="@drawable/mao" />
```

```

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_collapseMode="pin"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

    </android.support.design.widget.CollapsingToolbarLayout>

</android.support.design.widget.AppBarLayout>

<android.support.v7.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scrollbars="none"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />
</android.support.design.widget.CoordinatorLayout>

```

CollapsingToolbarLayout 有几个关键属性需要说明一下：

- `app:contentScrim=""`，用来设置 CollapsingToolbarLayout 收缩后最顶部的颜色。
- `app:expandedTitleGravity="left|bottom"`，表示将此 CollapsingToolbarLayout 完全展开后，title 所处的位置，默认的值 `left+bottom`。
- `app:collapsedTitleGravity="left"`，表示当头部的衬图 `ImageView` 消失后，此 title 将回归到 `Toolbar` 的位置，默认的值 `left`。
- `app:layout_scrollFlags=""`，这个属性我们上面讲过，它用来设置滚动事件，属性里面必须至少启用 `scroll` 这个 flag，这样这个 View 才会滚动出屏幕，否则它将一直固定在顶部。这里我们设置的是 `app:layout_scrollFlags="scroll|exitUntilCollapsed"`，这样能实现折叠效果。如果想要隐藏效果，我们可以设置 `app:layout_scrollFlags="scroll|enterAlways"`。

我们需要定义 `AppBarLayout` 与滚动视图之间的联系，Design Support Library 包含了一个特殊的字符串资源 `@string/appbar_scrolling_view_behavior`，它和 `AppBarLayout.ScrollingViewBehavior` 相匹配，用来通知 `AppBarLayout` 何时发生了滚动事件。这个 Behavior 需要设置在触发事件的 View 之上，所以我们应该在 `RecyclerView` 或者任意支持嵌套滚动的 View 比如 `NestedScrollView` 上添加 `app:layout_behavior="@string/appbar_scrolling_view_behavior"` 这个属性。当然，`AppBarLayout` 中的子 View 需要设置 `app:layout_scrollFlags` 这个属性；否则就算接收到 `RecyclerView` 滚动事件，`AppBarLayout` 也不会有什么变化。接下来看看 Java 代码的引用，代码如下所示：

```
public class CollapsingToolbarActivity extends AppCompatActivity {
```



```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_detail);
    Toolbar toolbar = (Toolbar) this.findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    CollapsingToolbarLayout collapsingToolbar =
        (CollapsingToolbarLayout)
            findViewById(R.id.collapsing_toolbar);
    collapsingToolbar.setTitle("哆啦 A 梦");
    RecyclerView mRecyclerView = (RecyclerView) findViewById(R.id.
        recyclerView);
    mRecyclerView.setLayoutManager(new StaggeredGridLayoutManager
        (1, StaggeredGridLayoutManager.VERTICAL));
    mRecyclerView.setItemAnimator(new DefaultItemAnimator());
    mRecyclerView.setAdapter(new
        RecyclerViewAdapter(CollapsingToolbarActivity.this));
}
}

```

这里列表的 Adapter 还是用到了此前定义的 RecyclerViewAdapter，另外设置了 CollapsingToolbarLayout 的标题，其运行效果如图 2-24 所示。这个时候我们向上滑动，哆啦 A 梦的图片就会逐渐消失，哆啦 A 梦这个标题也会逐渐移动到左上方，最终效果如图 2-25 所示。



图 2-24 展开的 CollapsingToolbarLayout



图 2-25 折叠的 CollapsingToolbarLayout

3. 自定义 Behavior

CoordinatorLayout 中最经典的设计就是 Behavior，在前面我们提到了 `app:layout_behavior="@string/appbar_scrolling_view_behavior"`，其实 `@string/appbar_scrolling_view_behavior` 对应着的是 `AppBarLayout.ScrollingViewBehavior`。我们也可以自定义 Behavior 来实现自己的组件和滑动交互。自定义 Behavior 可以分为两种方法：第一种是定义的 View 监听 CoordinatorLayout 里的滑动状态；第二种是定义的 View 监听另一个 View 的状态变化，例如 View 的大小、位置和显示状态等。对于第一种方法，我们需要注意 `onStartNestedScroll()` 和 `onNestedPreScroll` 方法；而对于第二种方法，则需要注意 `layoutDependsOn()` 和 `onDependentViewChanged()` 方法。我们需要明确自己需要实现什么，也就是我们定义一个底部提示条，当我们向上滚动的时候其就消失，向下滚动时就出现。我们用第一种方法来实现，下面查看布局。

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout xmlns:android="http://
schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:id="@+id/main_content"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:fitsSystemWindows="true">

<android.support.design.widget.AppBarLayout
    android:id="@+id/appbar"
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:fitsSystemWindows="true"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

<android.support.design.widget.CollapsingToolbarLayout
    android:id="@+id/collapsing_toolbar"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    app:contentScrim="?attr/colorPrimary"
    app:expandedTitleMarginEnd="64dp"
    app:expandedTitleMarginStart="48dp"
    app:layout_scrollFlags="scroll|exitUntilCollapsed">

<ImageView
    android:id="@+id/backdrop"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
```

```

        android:scaleType="centerCrop"
        android:src="@drawable/mao" />

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_collapseMode="pin"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

    </android.support.design.widget.CollapsingToolbarLayout>

</android.support.design.widget.AppBarLayout>

<android.support.v7.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scrollbars="none"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom"
    android:background="@android:color/holo_blue_dark"
    android:gravity="center_vertical"
    android:orientation="horizontal"
    android:padding="15dp"
    app:layout_behavior="com.example.liuwangshu.mooncoordinatorlayout.
FooterBehavior">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="自定义 Behavior"
        android:textColor="@android:color/white" />
</LinearLayout>

</android.support.design.widget.CoordinatorLayout>

```

我们在原有的布局基础上添加了一个 `LinearLayout`，里面有一个 `TextView`。需要注意的是我们在 `LinearLayout` 布局中添加了 `app:layout_behavior="com.example.liuwangshu.mooncoordinatorlayout.FooterBehavior"`。

FooterBehavior"这个属性，其中 FooterBehavior 就是我们自定义的 Behavior，代码如下所示：

```
public class FooterBehavior extends CoordinatorLayout.Behavior<View> {
    private int directionChange;
    public FooterBehavior(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    @Override
    public boolean onStartNestedScroll(CoordinatorLayout coordinatorLayout,
        View child, View directTargetChild, View target, int nestedScrollAxes) {
        return (nestedScrollAxes & ViewCompat.SCROLL_AXIS_VERTICAL) != 0;
    }

    @Override
    public void onNestedPreScroll(CoordinatorLayout coordinatorLayout, View
        child, View target, int dx, int dy, int[] consumed) {
        if (dy > 0 && directionChange < 0 || dy < 0 && directionChange > 0) {
            child.animate().cancel();
            directionChange = 0;
        }
        directionChange += dy;
        if (directionChange > child.getHeight() && child.getVisibility() ==
            View.VISIBLE) {
            hide(child);
        } else if (directionChange < 0 && child.getVisibility() == View.GONE) {
            show(child);
        }
    }

    ...
}
```

自定义的 Behavior 要继承 CoordinatorLayout.Behavior<View>。父类 Behavior 中有很多方法，我们在这里只关心 onStartNestedScroll()和 onNestedPreScroll 方法。onStartNestedScroll()方法的返回值表明这次滑动我们要不要关心，这里我们关心的是 Y 轴方向上的。onNestedPreScroll()方法则用于处理滑动。这个参数 child 就是我们定义的 LinearLayout；dy 则是我们水平滑动的距离，向上滑动为正值，向下滑动则为负值。如果滑动的距离累加值 directionChange 大于我们设置的 LinearLayout 的高度，并且该 LinearLayout 是 VISIBLE 状态，则隐藏 LinearLayout。如果 directionChange 小于 0，并且 LinearLayout 是 GONE 状态，则显示 LinearLayout。显示和隐藏的代码如下所示：

```
private void hide(final View view) {
```

```

        ViewPropertyAnimator animator = view.animate().translationY (view.
        getHeight()).setInterpolator(new
        FastOutSlowInInterpolator()).setDuration(200);
        animator.addListener(new AnimatorListenerAdapter() {
            @Override
            public void onAnimationEnd(Animator animator) {
                view.setVisibility(View.GONE);
            }
        });
        animator.start();
    }

    private void show(final View view) {
        ViewPropertyAnimator animator = view.animate().translationY(0).
        setInterpolator(new FastOutSlowInInterpolator()).setDuration(200);
        animator.addListener(new AnimatorListenerAdapter() {
            @Override
            public void onAnimationEnd(Animator animator) {
                view.setVisibility(View.VISIBLE);
            }
        });
        animator.start();
    }
}

```

show()方法和 hide()方法用来展示显示和隐藏的动画。这里用到了属性动画，关于属性动画会在第3章进行讲解，在这里就不介绍了。在这里需要注意的就是在动画结束时，也就是回调方法onAnimationEnd(Animator animator)中对我们添加的LinearLayout进行处理，show()方法在动画结束时调用view.setVisibility(View.VISIBLE)，而hide()方法在动画结束时调用view.setVisibility(View.GONE)。运行程序来查看效果，如图2-26所示。默认情况下这个控件是显示出来的，我们向上滑动时它就会消失，而我们再次向下滑动的时候它又会显示出来。到此，自定义Behavior的第一种方法就讲到这里了。接下来讲第二种方法，那就是我们定义的LinearLayout去监听另一个View的状态变化。使用第二种方法重新自定义Behavior代码如下所示。



图 2-26 自定义 Behavior

```

public class FooterBehaviorAppBar extends CoordinatorLayout.Behavior<View> {
    public FooterBehaviorAppBar(Context context, AttributeSet attrs) {

```

```

        super(context, attrs);
    }
    @Override
    public boolean layoutDependsOn(CoordinatorLayout parent, View child,
        View dependency) {
        return dependency instanceof AppBarLayout;
    }
    @Override
    public boolean onDependentViewChanged(CoordinatorLayout parent, View
        child, View dependency) {
        float translationY = Math.abs(dependency.getY());
        child.setTranslationY(translationY);
        return true;
    }
}

```

我们重新定义了 FooterBehaviorAppBar 继承 CoordinatorLayout.Behavior<View>。我们前面提到第二种方法要关心 layoutDependsOn() 和 onDependentViewChanged() 方法。layoutDependsOn() 方法用来返回我们需要关心的类，其第一个参数是当前的 CoordinatorLayout，第二个参数 child 是我们设置这个 Behavior 的 View，第三个参数 dependency 是我们关心的那个 View。而我们在 layoutDependsOn() 方法中返回了 dependency instanceof AppBarLayout，意思就是我们关心的那个 View 是 AppBarLayout。onDependentViewChanged() 方法根据我们关心的 View 的变化来对我们设置 Behavior 的 View 进行一系列处理。我们在这个方法中根据 dependency (AppBarLayout) 的垂直移动距离来设定 child (LinearLayout 的) 移动距离。接下来修改布局，将 app:layout_behavior 修改为我们重新设定的 FooterBehaviorAppBar，代码如下所示：

```

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom"
    android:background="@android:color/holo_blue_dark"
    android:gravity="center_vertical"
    android:orientation="horizontal"
    android:padding="15dp"
    app:layout_behavior="com.example.liuwangshu.mooncoordinatorlayout.
        FooterBehaviorAppbar">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="自定义 Behavior"
        android:textColor="@android:color/white" />
</LinearLayout>

```

运行程序，发现效果和第一种方法实现的效果不同，底部新加的 `LinearLayout` 并不完全随着我们的手指滑动而变化，而是随着 `AppBarLayout` 的显示而显示，随着 `AppBarLayout` 的隐藏而隐藏。而 `AppBarLayout` 的显示和隐藏又根据我们的手指滑动产生变化。所以我们定义的 `LinearLayout` 只有在滑到屏幕顶端的时候，会随着 `AppBarLayout` 的显示而逐渐显示出来。这个时候我们向上滑动，`LinearLayout` 也随着 `AppBarLayout` 的隐藏而隐藏，其余的场景 `LinearLayout` 并不会随着我们手指的手势而产生变化。

2.3 本章小结

本章主要介绍了 `Material Design` 的基本概念和 `Android Design Support Library` 库中主要控件的使用方法。开发者也需要对设计工作有一些了解。目前国内遵循 `Material Design` 的产品很少，很多设计师和产品经理也并不知道 `Android` 有自己的一套设计标准，他们更多的都是将 `iOS` 的设计风格硬搬到 `Android` 上，更有甚者就是想在多个平台下搞一套 `UI`。所以推广 `Material Design`，让设计师、产品经理更多地了解这些知识，对于我们开发者来说并没有坏处。

View 体系与自定义 View

本章将介绍 Android 中十分重要的 View。这里我用了 View 体系来概括它，可见它的庞大。对于一个 App 来说，与用户的交互、将内容展示给用户，既是十分重要的，也是十分必要的，而这些就是一个个 View 通过拓展实现的。View 就如同现实世界的原子一般，是实现界面展示和交互的最小“微粒”。在 Android 中，Activity 承担着大部分的展示任务，在本章中我们会了解 Activity 的组成。另外 View 的滑动也是非常重要的，对于手机这个屏幕较小的物品，如果想要展示更多的内容给用户，势必要进行滑动。本章会介绍 View 滑动的相关知识。Android 系统也提供了很多控件用于展示以及和用户交互，比如 TextView、Button、LinearLayout 等。虽然他们的功能十分强大，但有时我们为了追求更便利及更好的效果仍旧需要自己去写自定义 View，这就我们需要对 View 的事件分发及 View 的工作流程十分熟悉。所以在本章中也会着重讲解 View 的事件分发和 View 的工作流程，最后再讲解如何实现自定义 View。

3.1 View 与 ViewGroup

View 是 Android 所有控件的基类，我们平常所用的 TextView 和 ImageView 都是继承自 View 的，代码如下：

```
public class TextView extends View implements OnPreDrawListener {  
    . . .  
}  
public class ImageView extends View {  
    . . .  
}
```


接着看看我们平常用的布局控件 `LinearLayout`，它继承自 `ViewGroup`，代码如下所示。
`ViewGroup` 又是什么呢？`ViewGroup` 可以理解为 `View` 的组合，它可以包含很多 `View` 以及 `ViewGroup`，而它包含的 `ViewGroup` 又可以包含 `View` 和 `ViewGroup`，依此类推，形成一个 `View` 树，如图 3-1 所示。

```
public class LinearLayout extends ViewGroup {
    . . .
}
```

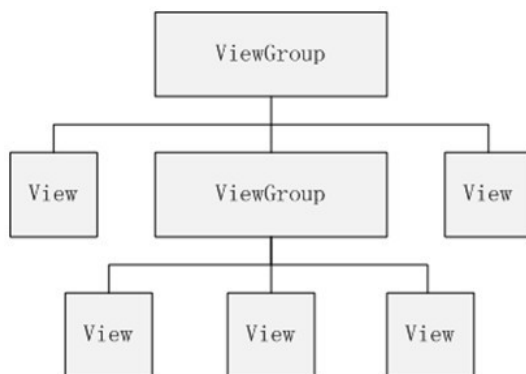


图 3-1 View 树

需要注意的是 `ViewGroup` 也继承自 `View`。`ViewGroup` 作为 `View` 或者 `ViewGroup` 这些组件的容器，派生了多种布局控件子类，比如 `LinearLayout`、`RelativeLayout` 等。一般来说，开发 Android 应用的 UI 界面都不会直接使用 `View` 和 `ViewGroup`，而是使用这两大基类的派生类。看图 3-2 我们就会有一个直观的了解。图 3-2 列举了 `View` 的部分继承关系，在这张图上我们看到 `ViewGroup`、`TextView` 等控件继承自 `View`，`LinearLayout`、`RelativeLayout` 等控件继承自 `ViewGroup`，`TableLayout`、`RadioGroup` 等控件继承自 `LinearLayout`，`EditText`、`Button` 等控件继承自 `TextView`，等等。

```
public abstract class ViewGroup extends View implements ViewParent,
ViewManager {
    . . .
}
```

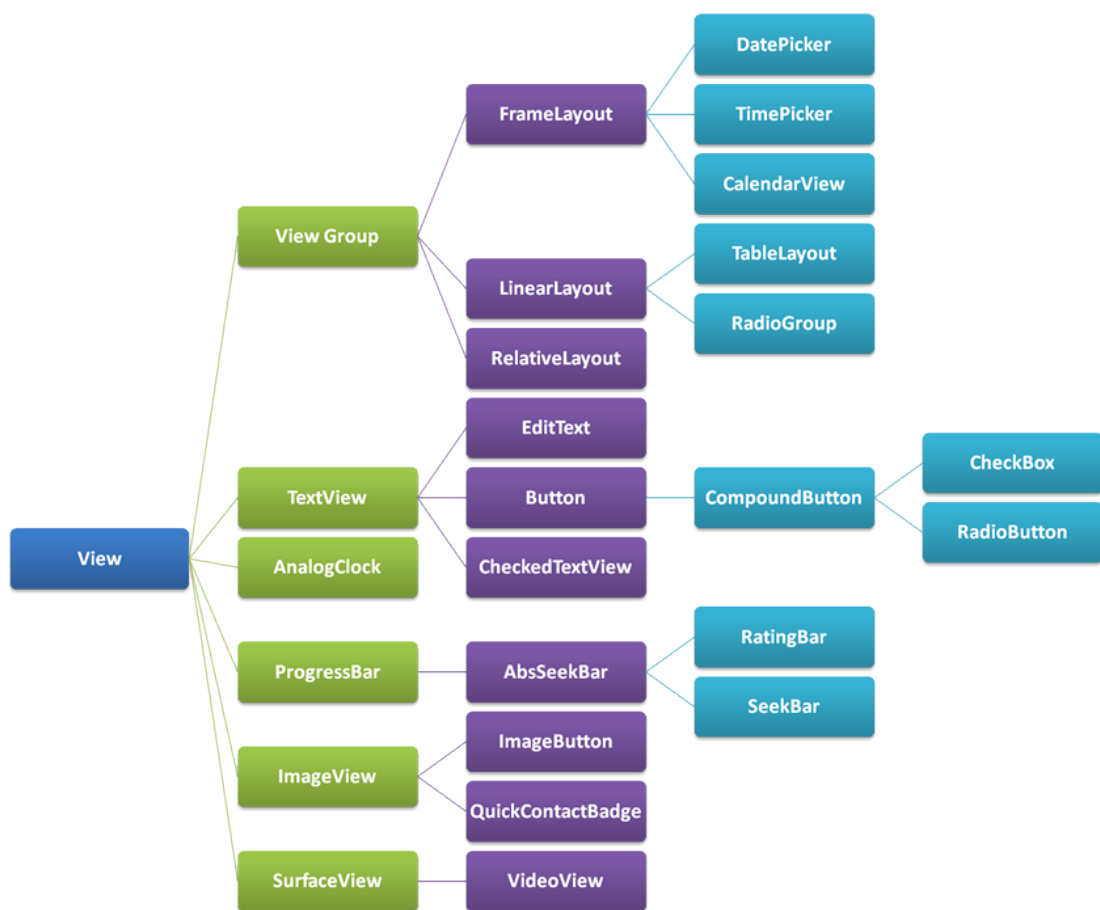


图 3-2 View 的部分继承关系

3.2 坐标系

Android 系统中有两种坐标系，分别为 Android 坐标系和 View 坐标系。了解这两种坐标系能够帮助我们实现 View 的各种操作，比如我们要实现 View 的滑动，你连这个 View 的位置都不知道，那如何去操作呢？首先我们来看看 Android 坐标系。

3.2.1 Android 坐标系

在 Android 中，将屏幕左上角的顶点作为 Android 坐标系的原点，这个原点向右是 X 轴正方向，向下是 Y 轴正方向，如图 3-3 所示。另外在触控事件中，使用 `getRawX()` 和 `getRawY()` 方法获得的坐标也是 Android 坐标系的坐标。

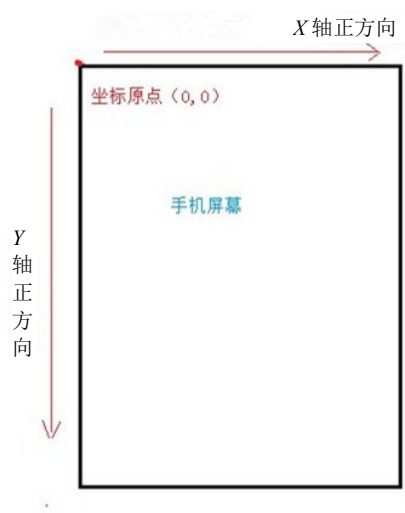


图 3-3 Android 坐标系

3.2.2 View 坐标系

除了 Android 坐标系，还有一个坐标系：View 坐标系，它与 Android 坐标系并不冲突，两者是共同存在的，它们一起来帮助开发者更好地控制 View。对于 View 坐标系，我们只需要搞明白图 3-4 中提供的信息就好了。

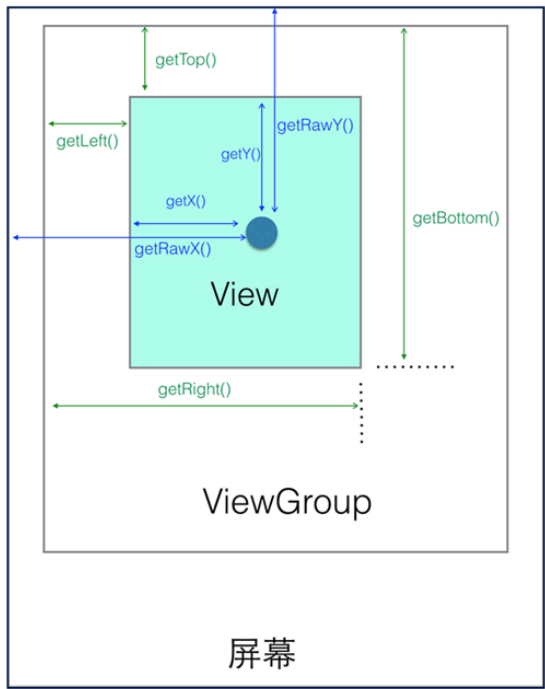


图 3-4 View 坐标系

1. View 获取自身的宽和高

根据图 3-4 可以得到很多结论，首先我们能算出 View 的宽和高：

```
width=getRight()-getLeft();  
height=getBottom()-getTop();
```

当然这样做显然有些麻烦，因为系统已经向我们提供了获取 View 宽和高的方法。`getHeight()` 用来获取 View 自身的高度，`getWidth()` 用来获取 View 自身的宽度。从 View 的源码来看，`getHeight()` 和 `getWidth()` 获取 View 自身的高度和宽度的算法与上面从图 3-4 中得出的结论是一致的。View 源码中的 `getHeight()` 方法和 `getWidth()` 方法如下所示：

```
public final int getHeight() {  
    return mBottom - mTop;  
}  
  
public final int getWidth() {  
    return mRight - mLeft;  
}
```

2. View 自身的坐标

通过如下方法可以获得 View 到其父控件（`ViewGroup`）的距离。

- `getTop()`：获取 View 自身顶边到其父布局顶边的距离。
- `getLeft()`：获取 View 自身左边到其父布局左边的距离。
- `getRight()`：获取 View 自身右边到其父布局左边的距离。
- `getBottom()`：获取 View 自身底边到其父布局顶边的距离。

3. MotionEvent 提供的方法

图 3-4 中间的那个圆点，假设就是我们触摸的点。我们知道无论是 View 还是 `ViewGroup`，最终的点击事件都会由 `onTouchEvent(MotionEvent event)` 方法来处理。`MotionEvent` 在用户交互中作用重大，其内部提供了很多事件常量，比如我们常用的 `ACTION_DOWN`、`ACTION_UP` 和 `ACTION_MOVE`。此外，`MotionEvent` 也提供了获取焦点坐标的各种方法。

- `getX()`：获取点击事件距离控件左边的距离，即视图坐标。
- `getY()`：获取点击事件距离控件顶边的距离，即视图坐标。
- `getRawX()`：获取点击事件距离整个屏幕左边的距离，即绝对坐标。
- `getRawY()`：获取点击事件距离整个屏幕顶边的距离，即绝对坐标。

3.3 View 的滑动

View 的滑动是 Android 实现自定义控件的基础，同时在开发中我们也难免会遇到 View 的

滑动处理。其实不管是哪种滑动方式，其基本思想都是类似的：当点击事件传到 View 时，系统记下触摸点的坐标，手指移动时系统记下移动后触摸的坐标并算出偏移量，并通过偏移量来修改 View 的坐标。实现 View 滑动有很多种方法，在这里主要讲解 6 种滑动方法，分别是 layout()、offsetLeftAndRight()与 offsetTopAndBottom()、LayoutParams、动画、scrollTo 与 scrollBy，以及 Scroller。

3.3.1 layout()方法

View 进行绘制的时候会调用 onLayout()方法来设置显示的位置，因此我们同样也可以通过修改 View 的 left、top、right、bottom 这 4 种属性来控制 View 的坐标。首先我们要自定义一个 View，在 onTouchEvent()方法中获取触摸点的坐标，代码如下所示：

```
public boolean onTouchEvent(MotionEvent event) {
    //获取手指触摸点的横坐标和纵坐标
    int x = (int) event.getX();
    int y = (int) event.getY();

    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            lastX = x;
            lastY = y;
            break;

        ...
    }
}
```

接下来我们在 ACTION_MOVE 事件中计算偏移量，再调用 layout()方法重新放置这个自定义 View 的位置即可。

```
case MotionEvent.ACTION_MOVE:
    //计算移动的距离
    int offsetX = x - lastX;
    int offsetY = y - lastY;
    //调用 layout 方法来重新放置它的位置
    layout(getLeft()+offsetX, getTop()+offsetY,
           getRight()+offsetX, getBottom()+offsetY);
    break;
```

在每次移动时都会调用 layout()方法对屏幕重新布局，从而达到移动 View 的效果。自定义 View，CustomView 的全部代码如下所示：

```
public class CustomView extends View {
    private int lastX;
    private int lastY;
```

```

    public CustomView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }
    public CustomView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public CustomView(Context context) {
        super(context);
    }

    public boolean onTouchEvent(MotionEvent event) {
        //获取手指触摸点的横坐标和纵坐标
        int x = (int) event.getX();
        int y = (int) event.getY();

        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                lastX = x;
                lastY = y;
                break;
            case MotionEvent.ACTION_MOVE:
                //计算移动的距离
                int offsetX = x - lastX;
                int offsetY = y - lastY;
                //调用 layout 方法来重新放置它的位置
                layout(getLeft()+offsetX, getTop()+offsetY,
                        getRight()+offsetX, getBottom()+offsetY);
                break;
        }
        return true;
    }
}

```

随后，我们在布局中引用自定义 View 就可以了。

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <com.example.liuwangshu.moonviewslide.CustomView
        android:id="@+id/customview"

```

```
        android:layout_width="80dp"
        android:layout_height="80dp"
        android:layout_margin="50dp"
        android:background="@android:color/holo_red_light" />
    </LinearLayout>
```

运行程序，效果如图 3-5 所示。图 3-5 中的方块就是我们自定义的 CustomView，它会随着我们手指的滑动改变自己的位置。

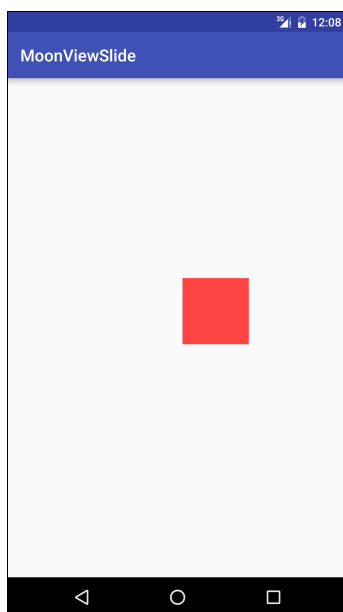


图 3-5 View 的滑动

3.3.2 offsetLeftAndRight()与 offsetTopAndBottom()

这两种方法和 layout()方法的效果差不多，其使用方式也差不多。我们将 ACTION_MOVE 中的代码替换成如下代码：

```
case MotionEvent.ACTION_MOVE:
    //计算移动的距离
    int offsetX = x - lastX;
    int offsetY = y - lastY;
    //对 left 和 right 进行偏移
    offsetLeftAndRight(offsetX);
    //对 top 和 bottom 进行偏移
    offsetTopAndBottom(offsetY);
    break;
```

3.3.3 LayoutParams (改变布局参数)

LayoutParams 主要保存了一个 View 的布局参数，因此我们可以通过 LayoutParams 来改变 View 的布局参数从而达到改变 View 位置的效果。同样，我们将 ACTION_MOVE 中的代码替换成如下代码：

```
LinearLayout.LayoutParams layoutParams= (LinearLayout.LayoutParams)
getLayoutParams();
        layoutParams.leftMargin = getLeft() + offsetX;
        layoutParams.topMargin = getTop() + offsetY;
        setLayoutParams(layoutParams);
```

因为父控件是 LinearLayout，所以我们用了 LinearLayout.LayoutParams。如果父控件是 RelativeLayout，则要使用 RelativeLayout.LayoutParams。除了使用布局的 LayoutParams 外，我们还可以用 ViewGroup.MarginLayoutParams 来实现：

```
ViewGroup.MarginLayoutParams layoutParams = (ViewGroup.MarginLayoutParams)
getLayoutParams();
        layoutParams.leftMargin = getLeft() + offsetX;
        layoutParams.topMargin = getTop() + offsetY;
        setLayoutParams(layoutParams);
```

3.3.4 动画

可以采用 View 动画来移动，在 res 目录新建 anim 文件夹并创建 translate.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <translate
        android:duration="1000"
        android:fromXDelta="0"
        android:toXDelta="300" />
</set>
```

接下来在 Java 代码中调用就好了，代码如下所示：

```
mCustomView.setAnimation(AnimationUtils.loadAnimation(this, R.anim.translate));
```

运行程序，我们设置的方块会向右平移 300 像素，然后又会回到原来的位置。为了解决这个问题，我们需要在 translate.xml 中加上 fillAfter="true"，代码如下所示。运行代码后会发现，方块向右平移 300 像素后就停留在当前位置了。

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true">
```



```

        <translate
            android:duration="1000"
            android:fromXDelta="0"
            android:toXDelta="300" />
    </set>

```

需要注意的是，View 动画并不能改变 View 的位置参数。如果对一个 Button 进行如上的平移动画操作，当 Button 平移 300 像素停留在当前位置时，我们点击这个 Button 并不会触发点击事件，但在我们点击这个 Button 的原始位置时却触发了点击事件。对于系统来说这个 Button 并没有改变原有的位置，所以我们点击其他位置当然不会触发这个 Button 的点击事件。在 Android 3.0 时出现的属性动画解决了上述问题，因为它不仅可以执行动画，还能够改变 View 的位置参数。当然，这里使用属性动画移动那就更简单了，我们让 CustomView 在 1000ms 内沿着 X 轴向右平移 300 像素，代码如下所示。关于属性动画，会在 3.4 节进行讲解。

```

ObjectAnimator.ofFloat(mCustomView, "translationX", 0, 300).setDuration(1000).start();

```

3.3.5 scrollTo 与 scrollBy

scrollTo(x,y)表示移动到一个具体的坐标点，而 scrollBy(dx,dy)则表示移动的增量为 dx、dy。其中，scrollBy 最终也是要调用 scrollTo 的。View.java 的 scrollBy 和 scrollTo 的源码如下所示：

```

public void scrollTo(int x, int y) {
    if (mScrollX != x || mScrollY != y) {
        int oldX = mScrollX;
        int oldY = mScrollY;
        mScrollX = x;
        mScrollY = y;
        invalidateParentCaches();
        onScrollChanged(mScrollX, mScrollY, oldX, oldY);
        if (!awakenScrollBars()) {
            postInvalidateOnAnimation();
        }
    }
}

public void scrollBy(int x, int y) {
    scrollTo(mScrollX + x, mScrollY + y);
}

```

scrollTo、scrollBy 移动的是 View 的内容，如果在 ViewGroup 中使用，则是移动其所有的子 View。我们将 ACTION_MOVE 中的代码替换成如下代码：

```

((View)getParent()).scrollBy(-offsetX, -offsetY);

```

这里若要实现 CustomView 随手指移动的效果，就需要将偏移量设置为负值。为什么要设置为负值呢？下面具体讲解一下。假设我们正用放大镜来看报纸，放大镜用来显示字的内容。同样我们可以把放大镜看作我们的手机屏幕，它们都是负责显示内容的；而报纸则可以被看作屏幕下的画布，它们都是用来提供内容的。放大镜外的内容，也就是报纸的内容不会随着放大镜的移动而消失，它一直存在。同样，我们的手机屏幕看不到的视图并不代表其不存在，如图 3-6 所示。画布上有 3 个控件，即 Button、EditText 和 SwitchButton。只有 Button 在手机屏幕中显示，它的 Android 坐标为 (60,60)。现在我们调用 `scrollBy(50,50)`，按照字面的意思，这个 Button 应该会在屏幕右下侧，可是事实并非如此。如果我们调用 `scrollBy(50,50)`，里面的参数都是正值，我们的手机屏幕向 *X* 轴正方向，也就是向右边平移 50，然后手机屏幕向 *Y* 轴正方向，也就是向下方平移 50，平移后的效果如图 3-7 所示。虽然我们设置的数值是正数并且在 *X* 轴和 *Y* 轴的正方向移动，但 Button 却向相反方向移动了，这是参考对象不同导致的差异。所以我们用 `scrollBy` 方法的时候要设置负数才会达到自己想要的效果。

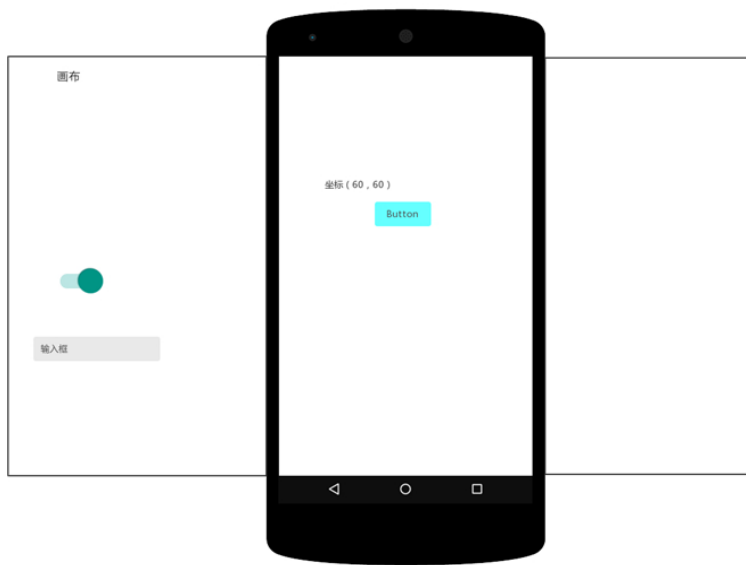
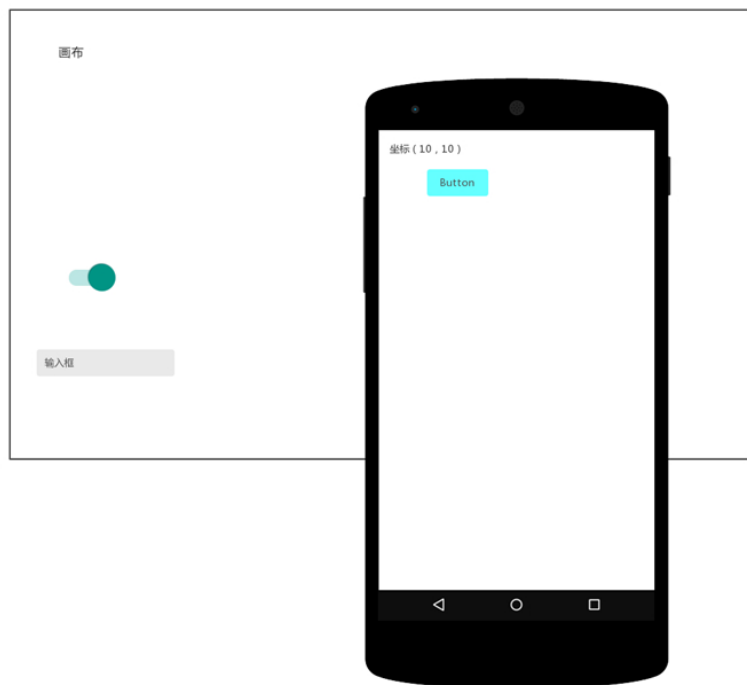


图 3-6 初始情况

图 3-7 调用 `scrollBy(50,50)`后

3.3.6 Scroller

我们在用 `scrollTo/scrollBy` 方法进行滑动时，这个过程是瞬间完成的，所以用户体验不大好。这里我们可以使用 `Scroller` 来实现有过渡效果的滑动，这个过程不是瞬间完成的，而是在一定的时间间隔内完成的。`Scroller` 本身是不能实现 `View` 的滑动的，它需要与 `View` 的 `computeScroll()` 方法配合才能实现弹性滑动的效果。在这里我们实现 `CustomView` 平滑地向右移动。首先我们要初始化 `Scroller`，代码如下所示：

```
public CustomView(Context context, AttributeSet attrs) {
    super(context, attrs);
    mScroller = new Scroller(context);
}
```

接下来重写 `computeScroll()` 方法，系统会在绘制 `View` 的时候在 `draw()` 方法中调用该方法。在这个方法中，我们调用父类的 `scrollTo()` 方法并通过 `Scroller` 来不断获取当前的滚动值，每滑动一小段距离我们就调用 `invalidate()` 方法不断地进行重绘，重绘就会调用 `computeScroll()` 方法，这样我们通过不断地移动一个小的距离并连贯起来就实现了平滑移动的效果。

```
@Override
public void computeScroll() {
    super.computeScroll();
    if(mScroller.computeScrollOffset()){
        ((View) getParent()).scrollTo(mScroller.getCurrX(),mScroller.
            getCurrY());
        invalidate();
    }
}
```

我们在 `CustomView` 中写一个 `smoothScrollTo` 方法，调用 `Scroller` 的 `startScroll()` 方法，在 2000ms 内沿 `X` 轴平移 `delta` 像素，代码如下所示：

```
public void smoothScrollTo(int destX,int destY){
    int scrollX=getScrollX();
    int delta=destX-scrollX;
    mScroller.startScroll(scrollX,0,delta,0,2000);
    invalidate();
}
```

最后我们在 `ViewSlideActivity.java` 中调用 `CustomView` 的 `smoothScrollTo()` 方法。这里我们设定 `CustomView` 沿着 `X` 轴向右平移 400 像素。

```
mCustomView.smoothScrollTo(-400,0);
```

3.4 属性动画

在 3.3.4 节中我们用属性动画实现了一个 View 的平移，但却并没有详细介绍它怎么用，本节就来学习一下如何使用属性动画。在属性动画出现之前，Android 系统提供的动画只有帧动画和 View 动画。View 动画我们都了解，它提供了 AlphaAnimation、RotateAnimation、TranslateAnimation、ScaleAnimation 这 4 种动画方式，并提供了 AnimationSet 动画集合来混合使用多种动画。随着 Android 3.0 属性动画的推出，View 动画不再风光。相比属性动画，View 动画一个非常大的缺陷突显，其不具有交互性。当某个元素发生 View 动画后，其响应事件的位置依然在动画进行前的地方，所以 View 动画只能做普通的动画效果，要避免涉及交互操作。但是它的优点也非常明显：效率比较高，使用也方便。由于 Android 3.0 之前已有的动画框架 Animation 存在一些局限性，也就是动画改变的只是显示，但 View 的位置没有发生变化，View 移动后并不能响应事件，所以在 Android 3.0 之后，谷歌就推出了新的动画框架，帮助开发者实现更加丰富的动画效果。在 Animator 框架中使用最多的就是 AnimatorSet 和 ObjectAnimator 配合：使用 ObjectAnimator 进行更精细化的控制，控制一个对象和一个属性值，而使用多个 ObjectAnimator 组合到 AnimatorSet 形成一个动画。属性动画通过调用属性 get、set 方法来真实地控制一个 View 的属性值，因此，强大的属性动画框架基本可以实现所有的动画效果。

1. ObjectAnimator

ObjectAnimator 是属性动画最重要的类，创建一个 ObjectAnimator 只需通过其静态工厂类直接返还一个 ObjectAnimator 对象。参数包括一个对象和对象的属性名字，但这个属性必须有 get 和 set 方法，其内部会通过 Java 反射机制来调用 set 方法修改对象的属性值。下面看看平移动画是如何实现的，代码如下所示：

```
ObjectAnimator mObjectAnimator=ObjectAnimator.ofFloat(view,"translationX",
200);
mObjectAnimator.setDuration(300);
mObjectAnimator.start();
```

通过 ObjectAnimator 的静态方法，创建一个 ObjectAnimator 对象，查看 ObjectAnimator.java 的静态方法 ofFloat()，代码如下所示：

```
public static ObjectAnimator ofFloat(Object target, String propertyName,
float... values) {
    ObjectAnimator anim = new ObjectAnimator(target, propertyName);
    anim.setFloatValues(values);
    return anim;
}
```

从源码可以看出第一个参数是要操作的 Object；第二个参数是要操作的属性；最后一个参数是一个可变的 float 类型数组，需要传进去该属性变化的取值过程，这里设置了一个参数，变化到 200。与 View 动画一样，也可以给属性动画设置显示时长、插值器等属性。下面就是一些

常用的可以直接使用的属性动画的属性值。

- **translationX** 和 **translationY**: 用来沿着 *X* 轴或者 *Y* 轴进行平移。
- **rotation**、**rotationX**、**rotationY**: 用来围绕 View 的支点进行旋转。
- **PivotX** 和 **PivotY**: 控制 View 对象的支点位置, 围绕这个支点进行旋转和缩放变换处理。默认该支点位置就是 View 对象的中心点。
- **alpha**: 透明度, 默认是 1 (不透明), 0 代表完全透明。
- **x** 和 **y**: 描述 View 对象在其容器中的最终位置。

需要注意的是, 在使用 **ObjectAnimator** 的时候, 要操作的属性必须要有 **get** 和 **set** 方法, 不然 **ObjectAnimator** 就无法生效。如果一个属性没有 **get**、**set** 方法, 也可以通过自定义一个属性类或包装类来间接地给这个属性增加 **get** 和 **set** 方法。现在来看看如何通过包装类的方法给一个属性增加 **get** 和 **set** 方法, 代码如下所示:

```
private static class MyView{
    private View mTarget;
    private MyView(View mTarget){
        this.mTarget=mTarget;
    }
    public int getWidth(){
        return mTarget.getLayoutParams().width;
    }
    public void setWidth(int width){
        mTarget.getLayoutParams().width=width;
        mTarget.requestLayout();
    }
}
```

使用时只需要操作包类就可以调用 **get**、**set** 方法了:

```
MyView mMyView=new MyView(mButton);
ObjectAnimator.ofInt(mMyView,"width",500).setDuration(500).start();
```

2. ValueAnimator

ValueAnimator 不提供任何动画效果, 它更像一个数值发生器, 用来产生有一定规律的数字, 从而让调用者控制动画的实现过程。通常情况下, 在 **ValueAnimator** 的 **AnimatorUpdateListener** 中监听数值的变化, 从而完成动画的变换, 代码如下所示:

```
ValueAnimator mValueAnimator=ValueAnimator.ofFloat(0,100);
mValueAnimator.setTarget(view);
mValueAnimator.setDuration(1000).start();
mValueAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    @Override
```

```

        public void onAnimationUpdate(ValueAnimator animation) {
            Float mFloat=(Float) animation.getAnimatedValue();
        }
    });
}

```

3. 动画的监听

完整的动画具有 start、Repeat、End、Cancel 这 4 个过程，代码如下所示：

```

ObjectAnimator animator=ObjectAnimator.ofFloat(view,"alpha",1.5f);
animator.addListener(new Animator.AnimatorListener() {
    @Override
    public void onAnimationStart(Animator animation) {
    }
    @Override
    public void onAnimationEnd(Animator animation) {
    }
    @Override
    public void onAnimationCancel(Animator animation) {
    }
    @Override
    public void onAnimationRepeat(Animator animation) {
    }
});
}

```

大部分时候我们只关心 onAnimationEnd 事件，Android 也提供了 AnimatorListenerAdapter 来让我们选择必要的事件进行监听。

```

ObjectAnimator animator=ObjectAnimator.ofFloat(view,"alpha",1.5f);
animator.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        super.onAnimationEnd(animation);
    }
});

```

4. 组合动画——AnimatorSet

AnimatorSet 类提供了一个 play()方法，如果我们向这个方法中传入一个 Animator 对象（ValueAnimator 或 ObjectAnimator），将会返回一个 AnimatorSet.Builder 的实例。AnimatorSet 的 play()方法源码如下所示：

```

public Builder play(Animator anim) {

```

```

        if (anim != null) {
            mNeedsSort = true;
            return new Builder(anim);
        }
        return null;
    }
}

```

很明显，在 `play()` 方法中创建了一个 `AnimatorSet.Builder` 类，这个 `Builder` 类是 `AnimatorSet` 的内部类。我们来看看这个 `Builder` 类中有什么，代码如下所示：

```

public class Builder {
    private Node mCurrentNode;
    Builder(Animator anim) {
        mCurrentNode = mNodeMap.get(anim);
        if (mCurrentNode == null) {
            mCurrentNode = new Node(anim);
            mNodeMap.put(anim, mCurrentNode);
            mNodes.add(mCurrentNode);
        }
    }

    public Builder with(Animator anim) {
        Node node = mNodeMap.get(anim);
        if (node == null) {
            node = new Node(anim);
            mNodeMap.put(anim, node);
            mNodes.add(node);
        }
        Dependency dependency = new Dependency(mCurrentNode, Dependency.WITH);
        node.addDependency(dependency);
        return this;
    }

    public Builder before(Animator anim) {
        mReversible = false;
        Node node = mNodeMap.get(anim);
        if (node == null) {
            node = new Node(anim);
            mNodeMap.put(anim, node);
            mNodes.add(node);
        }
        Dependency dependency = new Dependency(mCurrentNode, Dependency.AFTER);
        node.addDependency(dependency);
    }
}

```

```

        return this;
    }

    public Builder after(Animator anim) {
        mReversible = false;
        Node node = mNodeMap.get(anim);
        if (node == null) {
            node = new Node(anim);
            mNodeMap.put(anim, node);
            mNodes.add(node);
        }
        Dependency dependency = new Dependency(node, Dependency.AFTER);
        mCurrentNode.addDependency(dependency);
        return this;
    }

    public Builder after(long delay) {
        ValueAnimator anim = ValueAnimator.ofFloat(0f, 1f);
        anim.setDuration(delay);
        after(anim);
        return this;
    }
}

```

从源码中可以看出，**Builder** 类采用了建造者模式，每次调用方法时都返回 **Builder** 自身用于继续构建。**AnimatorSet.Builder** 中包括以下 4 个方法。

- **after(Animator anim)**: 将现有动画插入到传入的动画之后执行。
- **after(long delay)**: 将现有动画延迟指定毫秒后执行。
- **before(Animator anim)**: 将现有动画插入到传入的动画之前执行。
- **with(Animator anim)** : 将现有动画和传入的动画同时执行。

AnimatorSet 正是通过这几种方法来控制动画播放顺序的。很多读者可能还是一头雾水，这里再举一个例子，代码如下所示：

```

ObjectAnimator animator1 = ObjectAnimator.ofFloat(mCustomView, "translationX",
0.0f, 200.0f, 0f);
ObjectAnimator animator2 = ObjectAnimator.ofFloat(mCustomView, "scaleX",
1.0f, 2.0f);
ObjectAnimator animator3 = ObjectAnimator.ofFloat(mCustomView, "rotationX",
0.0f, 90.0f, 0.0f);
AnimatorSet set=new AnimatorSet();
set.setDuration(1000);
set.play(animator1).with(animator2).after(animator3);

```



```
set.start();
```

首先我们创建 3 个 `ObjectAnimator`，分别是 `animator1`、`animator2` 和 `animator3`，然后创建 `AnimatorSet`。在这里先执行 `animator3`，然后同时执行 `animator1` 和 `animator2`（也可以调用 `set.playTogether(animator1,animator2)`；来使这两种动画同时执行）。

5. 组合动画——PropertyValuesHolder

除了上面的 `AnimatorSet` 类，还可以使用 `PropertyValuesHolder` 类来实现组合动画。不过这个组合动画就没有上面的丰富了，使用 `PropertyValuesHolder` 类只能是多个动画一起执行。当然我们得结合 `ObjectAnimator.ofPropertyValuesHolder(Object target, PropertyValuesHolder... values)`；方法来使用。其第一个参数是动画的目标对象；之后的参数是 `PropertyValuesHolder` 类的实例，可以有多个这样的实例。具体代码如下所示：

```
PropertyValuesHolder valuesHolder1 = PropertyValuesHolder.ofFloat
("scaleX", 1.0f, 1.5f);
PropertyValuesHolder valuesHolder2 = PropertyValuesHolder.ofFloat
("rotationX", 0.0f, 90.0f, 0.0f);
PropertyValuesHolder valuesHolder3 = PropertyValuesHolder.ofFloat("alpha",
1.0f, 0.3f, 1.0f);
ObjectAnimator objectAnimator = ObjectAnimator.ofPropertyValuesHolder
(mCustomView, valuesHolder1, valuesHolder2, valuesHolder3);
objectAnimator.setDuration(2000).start();
```

6. 在 XML 中使用属性动画

和 `View` 动画一样，属性动画也可以直接写在 XML 中。在 `res` 文件中新建 `animator` 文件，在里面新建一个 `scale.xml`，其内容如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:duration="3000"
    android:propertyName="scaleX"
    android:valueFrom="1.0"
    android:valueTo="2.0"
    android:valueType="floatType"
    >
</objectAnimator>
```

在程序中引用 XML 定义的属性动画也很简单，代码如下所示：

```
Animator animator=AnimatorInflater.loadAnimator(this,R.animator.scale);
animator.setTarget(view);
```

```
animator.start();
```

3.5 解析 Scroller

在 3.3.6 节中我们介绍了如何使用 Scroller 进行滑动，但是其使用流程和一般的类的使用方式稍有不同。为了更好地理解 Scroller 的使用流程，我们有必要学习一下 Scroller 的源码。要想使用 Scroller，必须先调用 new Scroller()。下面先来看看 Scroller 的构造方法，代码如下所示：

```
public Scroller(Context context) {
    this(context, null);
}
public Scroller(Context context, Interpolator interpolator) {
    this(context, interpolator,
        context.getApplicationInfo().targetSdkVersion >= Build.VERSION_
        CODES.HONEYCOMB);
}
public Scroller(Context context, Interpolator interpolator, boolean
flywheel) {
    mFinished = true;
    if (interpolator == null) {
        mInterpolator = new ViscousFluidInterpolator();
    } else {
        mInterpolator = interpolator;
    }
    mPpi = context.getResources().getDisplayMetrics().density * 160.0f;
    mDeceleration = computeDeceleration(ViewConfiguration.
getScrollFriction());
    mFlywheel = flywheel;
    mPhysicalCoeff = computeDeceleration(0.84f);
}
```

从上面的代码我们得知，Scroller 有三个构造方法，通常情况下我们都用第一个；第二个需要传进去一个插值器 Interpolator，如果不传则采用默认的插值器 ViscousFluidInterpolator。接下来看看 Scroller 的 startScroll()方法，代码如下所示：

```
public void startScroll(int startX, int startY, int dx, int dy, int duration) {
    mMode = SCROLL_MODE;
    mFinished = false;
    mDuration = duration;
    mStartTime = AnimationUtils.currentAnimationTimeMillis();
    mStartX = startX;
    mStartY = startY;
    mFinalX = startX + dx;
```

```

        mFinalY = startY + dy;
        mDeltaX = dx;
        mDeltaY = dy;
        mDurationReciprocal = 1.0f / (float) mDuration;
    }

```

在 `startScroll()` 方法中并没有调用类似开启滑动的方法，而是保存了传进来的各种参数：`startX` 和 `startY` 表示滑动开始的起点，`dx` 和 `dy` 表示滑动的距离，`duration` 则表示滑动持续的时间。所以 `startScroll()` 方法只是用来做前期准备的，并不能使 `View` 进行滑动。关键是我们再 `startScroll()` 方法后调用了 `invalidate()` 方法，这个方法会导致 `View` 的重绘，而 `View` 的重绘会调用 `View` 的 `draw()` 方法，`draw()` 方法又会调用 `View` 的 `computeScroll()` 方法。我们重写 `computeScroll()` 方法如下：

```

@Override
public void computeScroll() {
    super.computeScroll();
    if (mScroller.computeScrollOffset()) {
        ((View) getParent()).scrollTo(mScroller.getCurrX(), mScroller.
            getCurrY());
        invalidate();
    }
}

```

我们在 `computeScroll()` 方法中通过 `Scroller` 来获取当前的 `ScrollX` 和 `ScrollY`，然后调用 `scrollTo()` 方法进行 `View` 的滑动，接着调用 `invalidate` 方法来让 `View` 进行重绘，重绘就会调用 `computeScroll()` 方法来实现 `View` 的滑动。这样通过不断地移动一个小的距离并连贯起来就实现了平滑移动的效果。但是在 `Scroller` 中如何获取当前位置的 `ScrollX` 和 `ScrollY` 呢？我们忘了一点，那就是在调用 `scrollTo()` 方法前会调用 `Scroller` 的 `computeScrollOffset()` 方法。接下来看看 `computeScrollOffset()` 方法：

```

public boolean computeScrollOffset() {
    if (mFinished) {
        return false;
    }
    int timePassed = (int) (AnimationUtils.currentAnimationTimeMillis()
        - mStartTime);
    if (timePassed < mDuration) {
        switch (mMode) {
            case SCROLL_MODE:
                final float x = mInterpolator.getInterpolation(timePassed *
                    mDurationReciprocal);
                mCurrX = mStartX + Math.round(x * mDeltaX);
                mCurrY = mStartY + Math.round(x * mDeltaY);

```

```

        break;
    case FLING_MODE:
        final float t = (float) timePassed / mDuration;
        final int index = (int) (NB_SAMPLES * t);
        float distanceCoef = 1.f;
        float velocityCoef = 0.f;
        if (index < NB_SAMPLES) {
            final float t_inf = (float) index / NB_SAMPLES;
            final float t_sup = (float) (index + 1) / NB_SAMPLES;
            final float d_inf = SPLINE_POSITION[index];
            final float d_sup = SPLINE_POSITION[index + 1];
            velocityCoef = (d_sup - d_inf) / (t_sup - t_inf);
            distanceCoef = d_inf + (t - t_inf) * velocityCoef;
        }

        mCurrVelocity = velocityCoef * mDistance / mDuration * 1000.0f;
        mCurrX = mStartX + Math.round(distanceCoef * (mFinalX - mStartX));
        mCurrX = Math.min(mCurrX, mMaxX);
        mCurrX = Math.max(mCurrX, mMinX);
        mCurrY = mStartY + Math.round(distanceCoef * (mFinalY -
            mStartY));
        mCurrY = Math.min(mCurrY, mMaxY);
        mCurrY = Math.max(mCurrY, mMinY);
        if (mCurrX == mFinalX && mCurrY == mFinalY) {
            mFinished = true;
        }
        break;
    }
}
else {
    mCurrX = mFinalX;
    mCurrY = mFinalY;
    mFinished = true;
}
return true;
}

```

首先会计算动画持续的时间 `timePassed`。如果动画持续时间小于我们设置的滑动持续时间 `mDuration`，则执行 `Switch` 语句。因为在 `startScroll()` 方法中的 `mMode` 值为 `SCROLL_MODE`，所以执行分支语句 `SCROLL_MODE`，然后根据插值器 `Interpolator` 来计算出在该时间段内移动的距离，赋值给 `mCurrX` 和 `mCurrY`，这样我们就能通过 `Scroller` 来获取当前的 `ScrollX` 和 `ScrollY` 了。另外，`computeScrollOffset()` 的返回值如果为 `true` 则表示滑动未结束，为 `false` 则表示滑动结

束。所以，如果滑动未结束，我们就得持续调用 `scrollTo()`方法和 `invalidate()`方法来进行 View 的滑动。讲到这里总结一下 Scroller 的原理：Scroller 并不能直接实现 View 的滑动，它需要配合 View 的 `computeScroll()`方法。在 `computeScroll()`中不断让 View 进行重绘，每次重绘都会计算滑动持续的时间，根据这个持续时间就能算出这次 View 滑动的位置，我们根据每次滑动的位置调用 `scrollTo()`方法进行滑动，这样不断地重复上述过程就形成了弹性滑动。

3.6 View 的事件分发机制

本章前面我们讲到了 View 的基础知识、View 的滑动、属性动画以及 Scroller 的相关知识，现在开始了解一下 View 体系中比较重要的知识点，即 View 的事件分发机制。在讲到 View 的事件分发机制之前要首先了解一下 Activity 的组成，然后从源码的角度来分析 View 的事件分发机制。

3.6.1 源码解析 Activity 的构成

点击事件用 `MotionEvent` 来表示，当一个点击事件产生后，事件最先传递给 Activity，所以我们首先要了解一下 Activity 的构成。当我们写 Activity 时会调用 `setContentView()`方法来加载布局。现在来看看 `setContentView()`方法是怎么实现的，代码如下所示：

```
public void setContentView(@LayoutRes int layoutResID) {
    getWindow().setContentView(layoutResID);
    initWindowDecorActionBar();
}
```

这里调用了 `getWindow().setContentView(layoutResID)`，`getWindow()`指的是什么呢？接着往下看，`getWindow()`返回 `mWindow`：

```
public Window getWindow() {
    return mWindow;
}
```

那这个 `mWindow` 又是什么呢？我们继续查看代码，最终在 Activity 的 `attach()`方法中发现了 `mWindow`，代码如下所示：

```
final void attach(Context context, ActivityThread aThread,
    Instrumentation instr, IBinder token, int ident,
    Application application, Intent intent, ActivityInfo info,
    CharSequence title, Activity parent, String id,
    NonConfigurationInstances lastNonConfigurationInstances,
    Configuration config, String referrer, IVoiceInteractor voiceInteractor) {
    attachBaseContext(context);
    mFragments.attachHost(null /*parent*/);
```

```

        mWindow = new PhoneWindow(this);
    ...
}

```

而 `getWindow()` 又指的是 `PhoneWindow`。所以来看看 `PhoneWindow` 的 `setContentView()` 方法，代码如下所示：

```

@Override
public void setContentView(View view, ViewGroup.LayoutParams params) {
    if (mContentParent == null) {
        installDecor();//1
    } else if (!hasFeature(FEATURE_CONTENT_TRANSITIONS)) {
        mContentParent.removeAllViews();
    }
    if (hasFeature(FEATURE_CONTENT_TRANSITIONS)) {
        view.setLayoutParams(params);
        final Scene newScene = new Scene(mContentParent, view);
        transitionTo(newScene);
    } else {
        mContentParent.addView(view, params);
    }
    final Callback cb = getCallback();
    if (cb != null && !isDestroyed()) {
        cb.onContentChanged();
    }
}

```

原来 `mWindow` 指的就是 `PhoneWindow`，`PhoneWindow` 是继承抽象类 `Window` 的，这样就知道了 `getWindow()` 得到的是一个 `PhoneWindow`，因为 `Activity` 中 `setContentView()` 方法调用的是 `getWindow().setContentView(layoutResID)`。

挑关键的接着看，看看上面代码注释 1 处 `installDecor()` 方法里面做了什么，代码如下所示：

```

private void installDecor() {
    if (mDecor == null) {
        mDecor = generateDecor();//1
        mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_DESCENDANTS);
        mDecor.setIsRootNamespace(true);
        if (!mInvalidatePanelMenuPosted && mInvalidatePanelMenuFeatures != 0) {
            mDecor.postOnAnimation(mInvalidatePanelMenuRunnable);
        }
    }
    if (mContentParent == null) {
        mContentParent = generateLayout(mDecor);
    }
}

```

```

        ...
    }
    ...
}
}

```

在前面的代码中没发现什么，紧接着查看上面代码注释 1 处的 `generateDecor` 方法里做了什么：

```

protected DecorView generateDecor() {
    return new DecorView(getContext(), -1);
}

```

这里创建了一个 `DecorView`，这个 `DecorView` 就是 `Activity` 中的根 `View`。接着查看 `DecorView` 的源码，发现 `DecorView` 是 `PhoneWindow` 类的内部类，并且继承了 `FrameLayout`。我们再回到 `installDecor()` 方法中，查看 `generateLayout(mDecor)` 做了什么：

```

protected ViewGroup generateLayout(DecorView decor) {
    ...
    //根据不同的情况加载不同的布局给 layoutResource
    int layoutResource;
    int features = getLocalFeatures();
    if ((features & (1 << FEATURE_SWIPE_TO_DISMISS)) != 0) {
        layoutResource = R.layout.screen_swipe_dismiss;
    } else if ((features & ((1 << FEATURE_LEFT_ICON) | (1 << FEATURE_RIGHT_
        ICON)))) != 0) {
        if (mIsFloating) {
            TypedValue res = new TypedValue();
            getContext().getTheme().resolveAttribute(
                R.attr.dialogTitleIconsDecorLayout, res, true);
            layoutResource = res.resourceId;
        } else {
            layoutResource = R.layout.screen_title_icons;
        }
        removeFeature(FEATURE_ACTION_BAR);
    } else if ((features & ((1 << FEATURE_PROGRESS) | (1 << FEATURE_
        INDETERMINATE_PROGRESS)))) != 0
        && (features & (1 << FEATURE_ACTION_BAR)) == 0) {
        layoutResource = R.layout.screen_progress;
    } else if ((features & (1 << FEATURE_CUSTOM_TITLE)) != 0) {
        if (mIsFloating) {
            TypedValue res = new TypedValue();
            getContext().getTheme().resolveAttribute(
                R.attr.dialogCustomTitleDecorLayout, res, true);
            layoutResource = res.resourceId;
        }
    }
}

```

```

        } else {
            layoutResource = R.layout.screen_custom_title;
        }
        removeFeature(FEATURE_ACTION_BAR);
    } else if ((features & (1 << FEATURE_NO_TITLE)) == 0) {
        if (mIsFloating) {
            TypedValue res = new TypedValue();
            getContext().getTheme().resolveAttribute(
                R.attr.dialogTitleDecorLayout, res, true);
            layoutResource = res.resourceId;
        } else if ((features & (1 << FEATURE_ACTION_BAR)) != 0) {
            layoutResource = a.getResourceId(
                R.styleable.Window_windowActionBarFullscreenDecorLayout,
                R.layout.screen_action_bar);
        } else {
            layoutResource = R.layout.screen_title;//1
        }
    } else if ((features & (1 << FEATURE_ACTION_MODE_OVERLAY)) != 0) {
        layoutResource = R.layout.screen_simple_overlay_action_mode;
    } else {
        layoutResource = R.layout.screen_simple;
    }
    ...
    return contentParent;
}

```

PhoneWindow 的 `generateLayout()` 方法比较长，这里只截取了一小部分关键的代码，其主要内容就是根据不同的情况加载不同的布局给 `layoutResource`。现在查看上面代码注释 1 处的布局 `R.layout.screen_title`，这个文件在 `frameworks`，它的代码如下所示：

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:fitsSystemWindows="true">
    <!-- Popout bar for action modes -->
    <ViewStub android:id="@+id/action_mode_bar_stub"
        android:inflatedId="@+id/action_mode_bar"
        android:layout="@layout/action_mode_bar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="?attr/actionBarTheme" />
    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="?android:attr/windowTitleSize"
        style="?android:attr/windowTitleBackgroundStyle">
        <TextView android:id="@android:id/title"
            style="?android:attr/windowTitleStyle"

```



```

        android:background="@null"
        android:fadingEdge="horizontal"
        android:gravity="center_vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    </FrameLayout>
    <FrameLayout android:id="@android:id/content"
        android:layout_width="match_parent"
        android:layout_height="0dip"
        android:layout_weight="1"
        android:foregroundGravity="fill_horizontal|top"
        android:foreground="?android:attr/windowContentOverlay" />
</LinearLayout>

```

上面的 ViewStub 是用来显示 ActionBar 的。下面的两个 FrameLayout：一个是 title，用来显示标题；另一个是 content，用来显示内容。看到上面的源码，大家就知道了一个 Activity 包含一个 Window 对象，这个对象是由 PhoneWindow 来实现的。PhoneWindow 将 DecorView 作为整个应用窗口的根 View，而这个 DecorView 又将屏幕划分为两个区域：一个是 TitleView，另一个是 ContentView，而我们平常做应用所写的布局正是展示在 ContentView 中的，如图 3-8 所示。

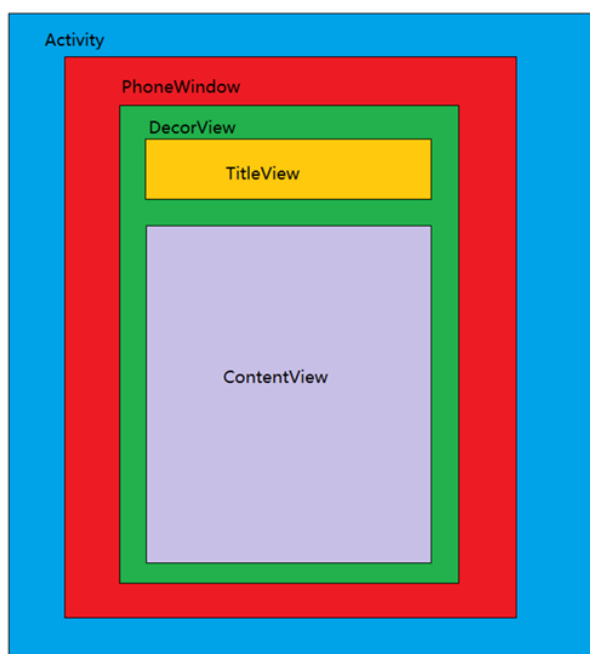


图 3-8 Activity 的构成

3.6.2 源码解析 View 的事件分发机制

当我们点击屏幕时，就产生了点击事件，这个事件被封装成了一个类：MotionEvent。而当

这个 MotionEvent 产生后，那么系统就会将这个 MotionEvent 传递给 View 的层级，MotionEvent 在 View 中的层级传递过程就是点击事件分发。在了解了什么是事件分发后，我们还需要了解事件分发的 3 个重要方法。点击事件有 3 个重要的方法，它们分别是：

- dispatchTouchEvent(MotionEvent ev)——用来进行事件的分发。
- onInterceptTouchEvent(MotionEvent ev)——用来进行事件的拦截，在 dispatchTouchEvent() 中调用，需要注意的是 View 没有提供该方法。
- onTouchEvent(MotionEvent ev)——用来处理点击事件，在 dispatchTouchEvent() 方法中进行调用。

1. View 的事件分发机制

当点击事件产生后，事件首先会传递给当前的 Activity，这会调用 Activity 的 dispatchTouchEvent() 方法，当然具体的事件处理工作都是交由 Activity 中的 PhoneWindow 来完成的，然后 PhoneWindow 再把事件处理工作交给 DecorView，之后再由 DecorView 将事件处理工作交给根 ViewGroup。所以，我们从 ViewGroup 的 dispatchTouchEvent() 方法开始分析，代码如下所示：

```
@Override
public boolean dispatchTouchEvent(MotionEvent ev) {
    ...
    if (actionMasked == MotionEvent.ACTION_DOWN) {
        cancelAndClearTouchTargets(ev);
        resetTouchState();
    }
    if (actionMasked == MotionEvent.ACTION_DOWN
        || mFirstTouchTarget != null) { //1
        final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_
            INTERCEPT) != 0; //2
        if (!disallowIntercept) {
            intercepted = onInterceptTouchEvent(ev);
            ev.setAction(action);
        } else {
            intercepted = false;
        }
    } else {
        intercepted = true;
    }
    ...
    return handled;
}
```

这里首先判断事件是否为 DOWN 事件，如果是，则进行初始化，resetTouchState 方法中会把 mFirstTouchTarget 的值置为 null。这里为什么要进行初始化呢？原因就是一个完整的事件序

列是以 DOWN 开始，以 UP 结束的。所以如果是 DOWN 事件，那么说明这是一个新的事件序列，故而需要初始化之前的状态。接着往下看，上面代码注释 1 处的条件如果满足，则执行下面的句子，mFirstTouchTarget 的意义是：当前 ViewGroup 是否拦截了事件，如果拦截了，mFirstTouchTarget=null；如果没有拦截并交由子 View 来处理，则 mFirstTouchTarget != null。假设当前的 ViewGroup 拦截了此事件，mFirstTouchTarget != null 则为 false，如果这时触发 ACTION_DOWN 事件，则会执行 onInterceptTouchEvent(ev) 方法；如果触发的是 ACTION_MOVE、ACTION_UP 事件，则不再执行 onInterceptTouchEvent(ev) 方法，而是直接设置 intercepted = true，此后的一个事件序列均由这个 ViewGroup 处理。再往下看，上面代码注释 2 处又出现了一个 FLAG_DISALLOW_INTERCEPT 标志位，它主要是禁止 ViewGroup 拦截除了 DOWN 之外的事件，一般通过子 View 的 requestDisallowInterceptTouchEvent 来设置。所以总结一下就是，当 ViewGroup 要拦截事件的时候，那么后续的事件序列都将交给它处理，而不用再调用 onInterceptTouchEvent() 方法了。所以，onInterceptTouchEvent() 方法并不是每次事件都会调用的。接下来查看 onInterceptTouchEvent() 方法：

```
public boolean onInterceptTouchEvent(MotionEvent ev) {
    return false;
}
```

onInterceptTouchEvent() 方法默认返回 false，不进行拦截。如果想要让 ViewGroup 拦截事件，那么应该在自定义的 ViewGroup 中重写这个方法。接着来看看 dispatchTouchEvent() 方法剩余的部分源码，如下所示：

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    ...
    final View[] children = mChildren;
    for (int i = childrenCount - 1; i >= 0; i--) { //1
        final int childIndex = customOrder
            ? getChildDrawingOrder(childrenCount, i) : i;
        final View child = (preorderedList == null)
            ? children[childIndex] : preorderedList.get (childIndex);
        if (childWithAccessibilityFocus != null) {
            if (childWithAccessibilityFocus != child) {
                continue;
            }

            childWithAccessibilityFocus = null;
            i = childrenCount - 1;
        }
        if (!canViewReceivePointerEvents(child)
            || !isTransformedTouchPointInView(x, y, child, null)) { //2
            ev.setTargetAccessibilityFocus(false);
            continue;
        }
    }
}
```

```

newTouchTarget = getTouchTarget(child);
if (newTouchTarget != null) {
    newTouchTarget.pointerIdBits |= idBitsToAssign;
    break;
}

resetCancelNextUpFlag(child);
if (dispatchTransformedTouchEvent(ev, false, child,
idBitsToAssign)) { //3
    mLastTouchDownTime = ev.getDownTime();
    if (preorderedList != null) {
        for (int j = 0; j < childrenCount; j++) {
            if (children[childIndex] == mChildren[j]) {
                mLastTouchDownIndex = j;
                break;
            }
        }
    } else {
        mLastTouchDownIndex = childIndex;
    }

    mLastTouchDownX = ev.getX();
    mLastTouchDownY = ev.getY();
    newTouchTarget = addTouchTarget(child,
idBitsToAssign);
    alreadyDispatchedToNewTouchTarget = true;
    break;
}

    ev.setTargetAccessibilityFocus(false);
}

...
}

```

在上面代码注释 1 处我们看到了 for 循环。首先遍历 **ViewGroup** 的子元素，判断子元素是否能够接收到点击事件，如果子元素能够接收到点击事件，则交由子元素来处理。需要注意这个 for 循环是倒序遍历的，即从最上层的子 **View** 开始往内层遍历。接着往下看注释 2 处的代码，其意思是判断触摸点位置是否在子 **View** 的范围内或者子 **View** 是否在播放动画。如果均不符合则执行 **continue** 语句，表示这个子 **View** 不符合条件，开始遍历下一个子 **View**。接下来查看注释 3 处的 **dispatchTransformedTouchEvent** 方法做了什么，代码如下所示：

```

private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean
cancel, View child, int desiredPointerIdBits) {

    final int oldAction = event.getAction();
    if (cancel || oldAction == MotionEvent.ACTION_CANCEL) {
        event.setAction(MotionEvent.ACTION_CANCEL);
    }
}

```

```

        if (child == null) {
            handled = super.dispatchTouchEvent(event);
        } else {
            handled = child.dispatchTouchEvent(event);
        }
        event.setAction(oldAction);
        return handled;
    }
    ...
}

```

如果有子 View，则调用子 View 的 `dispatchTouchEvent(event)` 方法。如果 ViewGroup 没有子 View，则调用 `super.dispatchTouchEvent(event)` 方法。ViewGroup 是继承 View 的，再来查看 View 的 `dispatchTouchEvent(event)`：

```

public boolean dispatchTouchEvent(MotionEvent event) {
    ...
    boolean result = false;
    if (onFilterTouchEventForSecurity(event)) {
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnTouchListener != null
            && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnTouchListener.onTouch(this, event)) {
            result = true;
        }
        if (!result && onTouchEvent(event)) {
            result = true;
        }
    }
    ...
    return result;
}

```

我们看到如果 `OnTouchListener` 不为 `null` 并且 `onTouch` 方法返回 `true`，则表示事件被消费，就不会执行 `onTouchEvent(event)`；否则就会执行 `onTouchEvent(event)`。可以看出 `OnTouchListener` 中的 `onTouch()` 方法优先级要高于 `onTouchEvent(event)` 方法。下面再来看看 `onTouchEvent()` 方法的部分源码：

```

public boolean onTouchEvent(MotionEvent event) {
    ...
    final int action = event.getAction();
    if (((viewFlags & CLICKABLE) == CLICKABLE ||
        (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE) ||
        (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE) {

```

```

        switch (action) {
            case MotionEvent.ACTION_UP:
                boolean prepressed = (mPrivateFlags & PFLAG_PREPRESSED) != 0;
                if ((mPrivateFlags & PFLAG_PRESSED) != 0 || prepressed) {
                    boolean focusTaken = false;
                    if (!mHasPerformedLongPress && !mIgnoreNextUpEvent) {
                        removeLongPressCallback();
                        if (!focusTaken) {
                            if (mPerformClick == null) {
                                mPerformClick = new PerformClick();
                            }
                            if (!post(mPerformClick)) {
                                performClick();
                            }
                        }
                    }
                }
            ...
        }
        return true;
    }
    return false;
}

```

从上面的代码可以看到，只要 View 的 CLICKABLE 和 LONG_CLICKABLE 有一个为 true，那么 onTouchEvent() 就会返回 true 消耗这个事件。CLICKABLE 和 LONG_CLICKABLE 代表 View 可以被点击和长按点击，可以通过 View 的 setClickable 和 setLongClickable 方法来设置，也可以通过 View 的 setOnClickListener 和 setOnLongClickListener 来设置，它们会自动将 View 设置为 CLICKABLE 和 LONG_CLICKABLE。接着在 ACTION_UP 事件中会调用 performClick() 方法：

```

public boolean performClick() {
    final boolean result;
    final ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnClickListener != null) { //1
        playSoundEffect(SoundEffectConstants.CLICK);
        li.mOnClickListener.onClick(this);
        result = true;
    } else {
        result = false;
    }
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);
    return result;
}

```

从上面代码注释 1 处可以看出，如果 View 设置了点击事件 OnClickListener，那么它的

onClick()方法就会被执行。View 事件分发机制的源码分析就讲到这里了，接下来介绍点击事件分发的传递规则。

2. 点击事件分发的传递规则

由前面事件分发机制的源码分析可知点击事件分发的这 3 个重要方法的关系，用伪代码来简单表示：

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    boolean result=false;
    if(onInterceptTouchEvent(ev)){
        result=super.onTouchEvent(ev);
    }else{
        result=child.dispatchTouchEvent(ev);
    }
    return result;
}
```

onInterceptTouchEvent 方法和 onTouchEvent 方法都在 dispatchTouchEvent 方法中调用。现在我们根据这段伪代码来分析一下点击事件分发的传递规则。

首先讲一下点击事件由上而下的传递规则，当点击事件产生后会由 Activity 来处理，传递给 PhoneWindow，再传递给 DecorView，最后传递给顶层的 ViewGroup。一般在事件传递中只考虑 ViewGroup 的 onInterceptTouchEvent 方法，因为一般情况下我们不会重写 dispatchTouchEvent()方法。对于根 ViewGroup，点击事件首先传递给它的 dispatchTouchEvent()方法，如果该 ViewGroup 的 onInterceptTouchEvent()方法返回 true，则表示它要拦截这个事件，这个事件就会交给它的 onTouchEvent()方法处理；如果 onInterceptTouchEvent()方法返回 false，则表示它不拦截这个事件，则这个事件会交给它的子元素的 dispatchTouchEvent()来处理，如此反复下去。如果传递给底层的 View，View 是没有子 View 的，就会调用 View 的 dispatchTouchEvent()方法，一般情况下最终会调用 View 的 onTouchEvent()方法。

举个例子，在金庸的《倚天屠龙记》中，武当派实力强劲，按照身份和实力区分，分别是武当掌门张三丰、武当七侠、武当弟子。这时突然有一个敌人来犯，这个消息首先会汇报给武当掌门张三丰。张三丰当然不会亲自出马，因此他就将应战的任务交给武当七侠之一的宋远桥（onInterceptTouchEvent()返回 false）；宋远桥威名远扬，也不会应战，因此他就把应战的任务交给武当弟子宋青书（onInterceptTouchEvent()返回 false）；武当弟子宋青书没有手下，他只能自己应战。在这里我们把武当掌门张三丰比作顶层 ViewGroup，将武当七侠之一的宋远桥比作中层 ViewGroup，武当弟子宋青书比作底层 View。那么事件的传递流程就是：武当掌门张三丰（顶层 ViewGroup）→武当七侠宋远桥（中层 ViewGroup）→武当弟子宋青书（底层 View）。因此得出结论，事件由上而下传递返回值的规则如下：为 true，则拦截，不继续向下传递；为 false，则不拦截，继续向下传递。

接下来讲解点击事件由下而上的传递。当点击事件传给底层的 View 时，如果其 onTouchEvent()方法返回 true，则事件由底层的 View 消耗并处理；如果返回 false 则表示该 View

不做处理，则传递给父 View 的 `onTouchEvent()` 处理；如果父 View 的 `onTouchEvent()` 仍旧返回 `false`，则继续传递给该父 View 的父 View 处理，如此反复下去。

再返回上面武侠的例子。武当弟子宋青书发现来犯的敌人是混元霹雳手成昆，他打不过成昆（`onTouchEvent()` 返回 `false`），于是就跑去找宋远桥，宋远桥来了，发现自己也打不过成昆（`onTouchEvent()` 返回 `false`），就去找武当掌门张三丰，张三丰用太极拳很轻松地打败了成昆（`onTouchEvent()` 返回 `true`）。因此得出结论，事件由下而上传递返回值的规则如下：为 `true`，则处理了，不继续向上传递；为 `false`，则不处理，继续向上传递。

3.7 View 的工作流程

View 的工作流程，指的就是 `measure`、`layout` 和 `draw`。其中，`measure` 用来测量 View 的宽和高，`layout` 用来确定 View 的位置，`draw` 则用来绘制 View。

3.7.1 View 的工作流程入口

在 3.6.1 节中我们讲到了 Activity 的构成，最后讲到了 `DecorView` 的创建以及它加载的资源。这个时候 `DecorView` 的内容还无法显示，因为它还没有被加载到 Window 中。接下来我们来看看 `DecorView` 如何被加载到 Window 中。

1. `DecorView` 被加载到 Window 中

当 `DecorView` 创建完毕，要加载到 Window 中时，我们需要先了解一下 Activity 的创建过程。当我们调用 Activity 的 `startActivity` 方法时，最终是调用 `ActivityThread` 的 `handleLaunchActivity` 方法来创建 Activity 的，代码如下所示：

```
private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {

    ...

    Activity a = performLaunchActivity(r, customIntent); //1
    if (a != null) {
        r.createdConfig = new Configuration(mConfiguration);
        Bundle oldState = r.state;
        handleResumeActivity(r.token, false, r.isForward,
            !r.activity.mFinished && !r.startsNotResumed); //2
    }

    ...
}
```

上面代码注释 1 处调用 `performLaunchActivity` 方法来创建 Activity，在这里面会调用到 Activity 的 `onCreate` 方法，从而完成 `DecorView` 的创建，这个内容在 3.6.1 节讲到过。接着在上面代码注释 2 处调用 `handleResumeActivity` 方法，代码如下所示：

```
final void handleResumeActivity(IBinder token,
```



```

        boolean clearHide, boolean isForward, boolean reallyResume) {
    unscheduleGcIdler();
    mSomeActivitiesChanged = true;
    ActivityClientRecord r = performResumeActivity(token, clearHide); //1
    if (r != null) {
        final Activity a = r.activity;
        ...
        if (r.window == null && !a.mFinished && willBeVisible) {
            r.window = r.activity.getWindow();
            View decor = r.window.getDecorView(); //2
            decor.setVisibility(View.INVISIBLE);
            ViewManager wm = a.getWindowManager(); //3
            WindowManager.LayoutParams l = r.window.getAttributes();
            a.mDecor = decor;
            l.type = WindowManager.LayoutParams.TYPE_BASE_APPLICATION;
            l.softInputMode |= forwardBit;
            if (a.mVisibleFromClient) {
                a.mWindowAdded = true;
                wm.addView(decor, l); //4
            }
        }
        ...
    }
}

```

在上面代码注释 1 处的 `performResumeActivity` 方法中会调用 `Activity` 的 `onResume` 方法。接着往下看，注释 2 处得到了 `DecorView`。注释 3 处得到了 `WindowManager`，`WindowManager` 是一个接口并且继承了接口 `ViewManager`。在注释 4 处调用 `WindowManager` 的 `addView` 方法，`WindowManager` 的实现类是 `WindowManagerImpl`，所以实际调用的是 `WindowManagerImpl` 的 `addView` 方法。具体代码如下所示：

```

public final class WindowManagerImpl implements WindowManager {
    private final WindowManagerGlobal mGlobal = WindowManagerGlobal.
        getInstance();
    ...
    @Override
    public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams
        params) {
        applyDefaultToken(params);
        mGlobal.addView(view, params, mDisplay, mParentWindow);
    }
    ...
}

```

在 `WindowManagerImpl` 的 `addView` 方法中，又调用了 `WindowManagerGlobal` 的 `addView` 方法，代码如下所示：

```

public void addView(View view, ViewGroup.LayoutParams params,
    Display display, Window parentWindow) {
    ...
    ViewRootImpl root;
    View panelParentView = null;
    synchronized (mLock) {
        ...
        root = new ViewRootImpl(view.getContext(), display);//1
        view.setLayoutParams(wparams);
        mViews.add(view);
        mRoots.add(root);
        mParams.add(wparams);
    }
    try {
        root.setView(view, wparams, panelParentView);//2
    } catch (RuntimeException e) {
        synchronized (mLock) {
            final int index = findViewLocked(view, false);
            if (index >= 0) {
                removeViewLocked(index, true);
            }
        }
        throw e;
    }
}

```

在上面代码注释 1 处创建了 `ViewRootImpl` 实例, 在注释 2 处调用了 `ViewRootImpl` 的 `setView` 方法并将 `DecorView` 作为参数传进去, 这样就把 `DecorView` 加载到了 `Window` 中。当然界面仍不会显示出什么来, 因为 `View` 的工作流程还没有执行完, 还需要经过 `measure`、`layout` 以及 `draw` 才会把 `View` 绘制出来。

2. ViewRootImpl 的 PerformTraveals 方法

前面讲到了将 `DecorView` 加载到 `Window` 中, 是通过 `ViewRootImpl` 的 `setView` 方法。`ViewRootImpl` 还有一个方法 `PerformTraveals`, 这个方法使得 `ViewTree` 开始 `View` 的工作流程, 代码如下所示:

```

private void performTraversals() {
    ...
    if (!mStopped) {
        int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);
        int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);
        performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
    }
}

```

```

    }

    if (didLayout) {
        performLayout(lp, desiredWindowWidth, desiredWindowHeight);
        ...
    }
    if (!cancelDraw && !newSurface) {
        if (!skipDraw || mReportNextDraw) {
            if (mPendingTransitions != null && mPendingTransitions.size() >
                0) {
                for (int i = 0; i < mPendingTransitions.size(); ++i) {
                    mPendingTransitions.get(i).startChangingAnimations();
                }
                mPendingTransitions.clear();
            }
            performDraw();
        }
    }
    ...
}

```

这里面主要执行了 3 个方法，分别是 `performMeasure`、`performLayout` 和 `performDraw`，在其方法的内部又会分别调用 `View` 的 `measure`、`layout` 和 `draw` 方法。需要注意的是，`performMeasure` 方法中需要传入两个参数，分别是 `childWidthMeasureSpec` 和 `childHeightMeasureSpec`。要了解这两个参数，需要了解 `MeasureSpec`。

3.7.2 理解 MeasureSpec

`MeasureSpec` 是 `View` 的内部类，其封装了一个 `View` 的规格尺寸，包括 `View` 的宽和高的信息，它的作用是在 `Measure` 流程中，系统会将 `View` 的 `LayoutParams` 根据父容器所施加的规则转换成对应的 `MeasureSpec`，然后在 `onMeasure` 方法中根据这个 `MeasureSpec` 来确定 `View` 的宽和高。`MeasureSpec` 的代码如下所示：

```

public static class MeasureSpec {
    private static final int MODE_SHIFT = 30;
    private static final int MODE_MASK = 0x3 << MODE_SHIFT;
    public static final int UNSPECIFIED = 0 << MODE_SHIFT;
    public static final int EXACTLY = 1 << MODE_SHIFT;
    public static final int AT_MOST = 2 << MODE_SHIFT;
    public static int makeMeasureSpec(int size, int mode) {
        if (sUseBrokenMakeMeasureSpec) {
            return size + mode;
        } else {

```

```

        return (size & ~MODE_MASK) | (mode & MODE_MASK);
    }
}

public static int makeMeasureSpec(int size, int mode) {
    if (sUseZeroUnspecifiedMeasureSpec && mode == UNSPECIFIED) {
        return 0;
    }
    return makeMeasureSpec(size, mode);
}

public static int getMode(int measureSpec) {
    return (measureSpec & MODE_MASK);
}

public static int getSize(int measureSpec) {
    return (measureSpec & ~MODE_MASK);
}

static int adjust(int measureSpec, int delta) {
    final int mode = getMode(measureSpec);
    int size = getSize(measureSpec);
    if (mode == UNSPECIFIED) {
        return makeMeasureSpec(size, UNSPECIFIED);
    }
    size += delta;
    if (size < 0) {
        Log.e(VIEW_LOG_TAG, "MeasureSpec.adjust: new size would be negative! (" + size + ") spec: " + toString(measureSpec) + " delta: " + delta);
        size = 0;
    }
    return makeMeasureSpec(size, mode);
}

public static String toString(int measureSpec) {
    int mode = getMode(measureSpec);
    int size = getSize(measureSpec);
    StringBuilder sb = new StringBuilder("MeasureSpec: ");
    if (mode == UNSPECIFIED)
        sb.append("UNSPECIFIED ");
    else if (mode == EXACTLY)
        sb.append("EXACTLY ");
    else if (mode == AT_MOST)
        sb.append("AT_MOST ");
    else
        sb.append(mode).append(" ");
    sb.append(size);
}

```

```

        return sb.toString();
    }
}

```

从 MeasureSpec 的常量可以看出，它代表了 32 位的 int 值，其中高 2 位代表了 SpecMode，低 30 位则代表 SpecSize。SpecMode 指的是测量模式，SpecSize 指的是测量大小。SpecMode 有 3 种模式，如下所示。

- UNSPECIFIED: 未指定模式，View 想多大就多大，父容器不做限制，一般用于系统内部的测量。
- AT_MOST: 最大模式，对应于 wrap_content 属性，子 View 的最终大小是父 View 指定的 SpecSize 值，并且子 View 的大小不能大于这个值。
- EXACTLY: 精确模式，对应于 match_parent 属性和具体的数值，父容器测量出 View 所需要的大小，也就是 SpecSize 的值。

对于每一个 View，都持有一个 MeasureSpec，而该 MeasureSpec 则保存了该 View 的尺寸规格。在 View 的测量流程中，通过 makeMeasureSpec 来保存宽和高的信息。通过 getMode 或 getSize 得到模式和宽、高。MeasureSpec 是受自身 LayoutParams 和父容器的 MeasureSpec 共同影响的。作为顶层 View 的 DecorView 来说，其并没有父容器，那么它的 MeasureSpec 是如何得来的呢？为了解决这个疑问，我们再回到 ViewRootImpl 的 PerformTraverse 方法，如下所示：

```

private void performTraversals() {
    ...
    if (!mStopped) {
        int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width); //1
        int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);
        performMeasure(childWidthMeasureSpec, childHeightMeasureSpec); //2
    }

    if (didLayout) {
        performLayout(lp, desiredWindowWidth, desiredWindowHeight);
        ...
    }

    if (!cancelDraw && !newSurface) {
        if (!skipDraw || mReportNextDraw) {
            if (mPendingTransitions != null && mPendingTransitions.size() > 0) {
                for (int i = 0; i < mPendingTransitions.size(); ++i) {
                    mPendingTransitions.get(i).startChangingAnimations();
                }
            }
            mPendingTransitions.clear();
        }
    }
}

```

```

        }
        performDraw();
    }
}
...
}

```

上面代码注释 1 处调用了 `getRootMeasureSpec(mWidth,lp.width)` 方法。下面来查看 `getRootMeasureSpec(mWidth,lp.width)` 方法做了什么：

```

private static int getRootMeasureSpec(int windowSize, int rootDimension) {
    int measureSpec;
    switch (rootDimension) {
        case ViewGroup.LayoutParams.MATCH_PARENT:
            measureSpec = MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.EXACTLY);
            break;
        case ViewGroup.LayoutParams.WRAP_CONTENT:
            measureSpec = MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.
                AT_MOST);
            break;
        default:
            measureSpec = MeasureSpec.makeMeasureSpec(rootDimension, MeasureSpec.
                EXACTLY);
            break;
    }
    return measureSpec;
}

```

`getRootMeasureSpec` 方法的第一个参数 `windowSize` 指的是窗口的尺寸，所以对于 `DecorView` 来说，它的 `MeasureSpec` 由自身的 `LayoutParams` 和窗口的尺寸决定，这一点和普通 `View` 是不同的。接着往下看，就会看到根据自身的 `LayoutParams` 来得到不同的 `MeasureSpec`。讲到这里，3.7.1 节最后遗留的问题：`performMeasure` 方法中需要传入两个参数，即 `childWidthMeasureSpec` 和 `childHeightMeasureSpec`，这代表什么我们也应该明白了。接着回到 `PerformTraveals` 方法，查看在注释 2 处的 `performMeasure` 方法内部做了什么，代码如下所示：

```

private void performMeasure(int childWidthMeasureSpec, int
    childHeightMeasureSpec) {
    Trace.traceBegin(Trace.TRACE_TAG_VIEW, "measure");
    try {
        mView.measure(childWidthMeasureSpec, childHeightMeasureSpec);
    } finally {

```

```

        Trace.traceEnd(Trace.TRACE_TAG_VIEW);
    }
}

```

其实，就算不看我们也应该知道里面调用了什么——是 View 的 measure 方法。3.7.3 节我们就来学习一下 View 的 measure 流程。

3.7.3 View 的 measure 流程

measure 用来测量 View 的宽和高，它的流程分为 View 的 measure 流程和 ViewGroup 的 measure 流程，只不过 ViewGroup 的 measure 流程除了要完成自己的测量，还要遍历地调用子元素的 measure() 方法。

1. View 的 measure 流程

首先来看一下 View 的 onMeasure 方法：

```

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(),
        widthMeasureSpec),
        getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
}

```

接着查看 setMeasuredDimension 方法，代码如下所示：

```

protected final void setMeasuredDimension(int measuredWidth, int
measuredHeight) {
    boolean optical = isLayoutModeOptical(this);
    if (optical != isLayoutModeOptical(mParent)) {
        Insets insets = getOpticalInsets();
        int opticalWidth = insets.left + insets.right;
        int opticalHeight = insets.top + insets.bottom;

        measuredWidth += optical ? opticalWidth : -opticalWidth;
        measuredHeight += optical ? opticalHeight : -opticalHeight;
    }
    setMeasuredDimensionRaw(measuredWidth, measuredHeight);
}

```

这很显然是用来设置 View 的宽、高的，再回头看看 getDefaultSize() 方法处理了什么：

```

public static int getDefaultSize(int size, int measureSpec) {
    int result = size;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);
}

```

```

switch (specMode) {
case MeasureSpec.UNSPECIFIED:
    result = size;
    break;
case MeasureSpec.AT_MOST:
case MeasureSpec.EXACTLY:
    result = specSize;
    break;
}
return result;
}

```

MeasureSpec 是 View 的测量模式，而 SpecSize 是 View 的测量大小，在 3.7.2 节中我们也了解了 MeasureSpec。这里很显然根据不同的 SpecMode 值来返回不同的 result 值，也就是 SpecSize。在 AT_MOST 和 EXACTLY 模式下，都返回 SpecSize 这个值，即 View 在这两种模式下的测量宽和高直接取决于 SpecSize。也就是说，对于一个直接继承自 View 的自定义 View 来说，它的 wrap_content 和 match_parent 属性的效果是一样的。因此如果要实现自定义 View 的 wrap_content，则要重写 onMeasure 方法，并对自定义 View 的 wrap_content 属性进行处理。而在 UNSPECIFIED 模式下返回的是 getDefaultSize 方法的第一个参数 size 的值，size 的值从 onMeasure 方法来看是 getSuggestedMinimumWidth 方法或者 getSuggestedMinimumHeight 方法得到的。我们来查看 getSuggestedMinimumWidth 方法做了什么。只需要弄懂 getSuggestedMinimumWidth 方法就可以了，因为这两个方法的原理是一样的。

```

protected int getSuggestedMinimumWidth() {
    return (mBackground == null) ? mMinWidth : max(mMinWidth, mBackground.
        getMinimumWidth());
}

```

如果 View 没有设置背景，则取值为 mMinWidth，mMinWidth 是可以设置的，它对应于 Android:minWidth 这个属性设置的值或者 View 的 setMinimumWidth 的值；如果不指定的话，则默认为 0。setMinimumWidth 方法如下所示：

```

public void setMinimumWidth(int minWidth) {
    mMinWidth = minWidth;
    requestLayout();
}

```

如果 View 设置了背景，则取值为 max(mMinWidth, mBackground.getMinimumWidth())，也就是取 mMinWidth 和 mBackground.getMinimumWidth() 之间的最大值。此前讲了 mMinWidth，下面看看 mBackground.getMinimumWidth()。这个 mBackground 是 Drawable 类型的，Drawable 类的 getMinimumWidth 方法如下所示：

```

public int getMinimumWidth() {

```



```

        final int intrinsicWidth = getIntrinsicWidth();
        return intrinsicWidth > 0 ? intrinsicWidth : 0;
    }

```

`intrinsicWidth` 得到的是这个 `Drawable` 的固有宽度，如果固有宽度大于 0 则返回固有宽度，否则返回 0。总结一下，`getSuggestedMinimumWidth` 方法就是：如果 `View` 没有设置背景，则返回 `mMinWidth`；如果设置了背景，就返回 `mMinWidth` 和 `Drawable` 的最小宽度之间的最大值。

2. ViewGroup 的 measure 流程

讲完了 `View` 的 `measure` 流程，接下来看看 `ViewGroup` 的 `measure` 流程。对于 `ViewGroup`，它不只要测量自身，还要遍历地调用子元素的 `measure()` 方法。`ViewGroup` 中没有定义 `onMeasure()` 方法，但却定义了 `measureChildren()` 方法：

```

protected void measureChildren(int widthMeasureSpec, int heightMeasureSpec) {
    final int size = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < size; ++i) {
        final View child = children[i];
        if ((child.mViewFlags & VISIBILITY_MASK) != GONE) {
            measureChild(child, widthMeasureSpec, heightMeasureSpec);
        }
    }
}

```

遍历子元素并调用 `measureChild` 方法，`measureChild` 方法如下所示：

```

protected void measureChild(View child, int parentWidthMeasureSpec,
    int parentHeightMeasureSpec) {
    final LayoutParams lp = child.getLayoutParams();

    final int childWidthMeasureSpec = getChildMeasureSpec(
        parentWidthMeasureSpec,
        mPaddingLeft + mPaddingRight, lp.width);
    final int childHeightMeasureSpec = getChildMeasureSpec(
        parentHeightMeasureSpec,
        mPaddingTop + mPaddingBottom, lp.height);

    child.measure(childWidthMeasureSpec, childHeightMeasureSpec);
}

```

调用 `child.getLayoutParams()` 方法来获得子元素的 `LayoutParams` 属性，获取子元素的 `MeasureSpec` 并调用子元素的 `measure()` 方法进行测量。`getChildMeasureSpec()` 方法里写了什么呢？其代码如下：

```
public static int getChildMeasureSpec(int spec, int padding, int
childDimension) {
    int specMode = MeasureSpec.getMode(spec);
    int specSize = MeasureSpec.getSize(spec);
    int size = Math.max(0, specSize - padding);
    int resultSize = 0;
    int resultMode = 0;
    switch (specMode) {
    case MeasureSpec.EXACTLY:
        if (childDimension >= 0) {
            resultSize = childDimension;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension == LayoutParams.MATCH_PARENT) {
            resultSize = size;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension == LayoutParams.WRAP_CONTENT) {
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;
        }
        break;
    case MeasureSpec.AT_MOST:
        if (childDimension >= 0) {
            resultSize = childDimension;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension == LayoutParams.MATCH_PARENT) {
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;
        } else if (childDimension == LayoutParams.WRAP_CONTENT) {
            resultSize = size;
            resultMode = MeasureSpec.AT_MOST;//1
        }
        break;
    case MeasureSpec.UNSPECIFIED:
        if (childDimension >= 0) {
            resultSize = childDimension;
            resultMode = MeasureSpec.EXACTLY;
        } else if (childDimension == LayoutParams.MATCH_PARENT) {
            resultSize = 0;
            resultMode = MeasureSpec.UNSPECIFIED;
        } else if (childDimension == LayoutParams.WRAP_CONTENT) {
            resultSize = 0;
            resultMode = MeasureSpec.UNSPECIFIED;
        }
    }
```

```

        break;
    }
    return MeasureSpec.makeMeasureSpec(resultSize, resultMode);
}

```

很显然，这是根据父容器的 `MeasureSpec` 模式再结合子元素的 `LayoutParams` 属性来得出的子元素的 `MeasureSpec` 属性。有一点需要注意的是，如果父容器的 `MeasureSpec` 属性为 `AT_MOST`，子元素的 `LayoutParams` 属性为 `WRAP_CONTENT`，那根据上面代码注释 1 处的代码，我们会发现子元素的 `MeasureSpec` 属性也为 `AT_MOST`，它的 `SpecSize` 值为父容器的 `SpecSize` 减去 `padding` 的值。换句话说，这和子元素设置 `LayoutParams` 属性为 `MATCH_PARENT` 效果是一样的。为了解决这个问题，需要在 `LayoutParams` 属性为 `WRAP_CONTENT` 时指定一下默认的宽和高。`ViewGroup` 并没有提供 `onMeasure` 方法，而是让其子类来各自实现测量的方法，究其原因就是 `ViewGroup` 有不同布局的需要，很难统一。接下来我们简单分析一下 `ViewGroup` 的子类 `LinearLayout` 的 `measure` 流程。现在先来看看它的 `onMeasure` 方法，代码如下所示：

```

@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    if (mOrientation == VERTICAL) {
        measureVertical(widthMeasureSpec, heightMeasureSpec);
    } else {
        measureHorizontal(widthMeasureSpec, heightMeasureSpec);
    }
}

```

这个方法的逻辑很简单，如果是垂直方向则调用 `measureVertical` 方法，否则就调用 `measureHorizontal` 方法。接着分析垂直 `measureVertical()` 方法的部分源码：

```

void measureVertical(int widthMeasureSpec, int heightMeasureSpec) {
    mTotalLength = 0;
    mTotalLength = 0;
    ...
    for (int i = 0; i < count; ++i) {
        final View child = getVirtualChildAt(i);
        if (child == null) {
            mTotalLength += measureNullChild(i);
            continue;
        }
        if (child.getVisibility() == View.GONE) {
            i += getChildrenSkipCount(child, i);
            continue;
        }
        if (hasDividerBeforeChildAt(i)) {
            mTotalLength += mDividerHeight;

```

```

    }
    LinearLayout.LayoutParams lp = (LinearLayout.LayoutParams)
        child.getLayoutParams();
    totalWeight += lp.weight;
    if (heightMode == MeasureSpec.EXACTLY && lp.height == 0 &&
        lp.weight > 0) {
        final int totalLength = mTotalLength;
        mTotalLength = Math.max(totalLength, totalLength + lp.topMargin
            + lp.bottomMargin);
        skippedMeasure = true;
    } else {
        int oldHeight = Integer.MIN_VALUE;
        if (lp.height == 0 && lp.weight > 0) {
            oldHeight = 0;
            lp.height = LayoutParams.WRAP_CONTENT;
        }
        measureChildBeforeLayout(
            child, i, widthMeasureSpec, 0, heightMeasureSpec,
            totalWeight == 0 ? mTotalLength : 0);
        if (oldHeight != Integer.MIN_VALUE) {
            lp.height = oldHeight;
        }

        final int childHeight = child.getMeasuredHeight();
        final int totalLength = mTotalLength;
        mTotalLength = Math.max(totalLength, totalLength + childHeight
            + lp.topMargin + lp.bottomMargin +
            getNextLocationOffset(child));
    }
    ...
    if (useLargestChild &&
        (heightMode == MeasureSpec.AT_MOST || heightMode ==
        MeasureSpec.UNSPECIFIED)) {
        mTotalLength = 0;
        for (int i = 0; i < count; ++i) {
            final View child = getVirtualChildAt(i);
            if (child == null) {
                mTotalLength += measureNullChild(i);
                continue;
            }
            if (child.getVisibility() == GONE) {
                i += getChildrenSkipCount(child, i);
                continue;
            }
        }
        final LinearLayout.LayoutParams lp = (LinearLayout.LayoutParams)

```

```

        child.getLayoutParams();
        final int totalLength = mTotalLength;
        mTotalLength = Math.max(totalLength, totalLength +
            largestChildHeight + lp.topMargin +
            lp.bottomMargin + getNextLocationOffset(child));
    }
}
mTotalLength += mPaddingTop + mPaddingBottom;
int heightSize = mTotalLength;
...
}

```

这里定义了 `mTotalLength` 用来存储 `LinearLayout` 在垂直方向的高度，然后遍历子元素，根据子元素的 `MeasureSpec` 模式分别计算每个子元素的高度。如果是 `WRAP_CONTENT`，则将每个子元素的高度和 `margin` 垂直高度等值相加并赋值给 `mTotalLength`。当然，最后还要加上垂直方向 `padding` 的值。如果布局高度设置为 `MATCH_PARENT` 或者具体数值，则和 `View` 的测量方法是一样的。`measure` 流程就讲到这里了，接下来讲解 `View` 的 `layout` 和 `draw` 流程。

3.7.4 View 的 layout 流程

`layout` 方法的作用是确定元素的位置。`ViewGroup` 中的 `layout` 方法用来确定子元素的位置，`View` 中的 `layout` 方法则用来确定自身的位置。首先我们看看 `View` 的 `layout` 方法：

```

public void layout(int l, int t, int r, int b) {
    if ((mPrivateFlags3 & PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT) != 0) {
        onMeasure(mOldWidthMeasureSpec, mOldHeightMeasureSpec);
        mPrivateFlags3 &= ~PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT;
    }
    int oldL = mLeft;
    int oldT = mTop;
    int oldB = mBottom;
    int oldR = mRight;
    boolean changed = isLayoutModeOptical(mParent) ?
        setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);
    if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) == PFLAG
        _LAYOUT_REQUIRED) {
        onLayout(changed, l, t, r, b);
        mPrivateFlags &= ~PFLAG_LAYOUT_REQUIRED;
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnLayoutChangeListeners != null) {
            ArrayList<OnLayoutChangeListener> listenersCopy =
                (ArrayList<OnLayoutChangeListener>)li.
                mOnLayoutChangeListeners.clone();

```

```

        int numListeners = listenersCopy.size();
        for (int i = 0; i < numListeners; ++i) {
            listenersCopy.get(i).onLayoutChange
                (this, l, t, r, b, oldL, oldT, oldR, oldB);
        }
    }
}

mPrivateFlags &= ~PFLAG_FORCE_LAYOUT;
mPrivateFlags3 |= PFLAG3_IS_LAID_OUT;
}

```

layout 方法的 4 个参数 l、t、r、b 分别是 View 从左、上、右、下相对于其父容器的距离。接着来查看 setFrame 方法里做了什么，代码如下所示：

```

protected boolean setFrame(int left, int top, int right, int bottom) {
    boolean changed = false;
    if (DBG) {
        Log.d("View", this + " View.setFrame(" + left + "," + top + ","
            + right + "," + bottom + ")");
    }
    if (mLeft != left || mRight != right || mTop != top || mBottom != bottom) {
        changed = true;
        int drawn = mPrivateFlags & PFLAG_DRAWN;
        int oldWidth = mRight - mLeft;
        int oldHeight = mBottom - mTop;
        int newWidth = right - left;
        int newHeight = bottom - top;
        boolean sizeChanged = (newWidth != oldWidth) || (newHeight !=
            oldHeight);
        invalidate(sizeChanged);
        mLeft = left;
        mTop = top;
        mRight = right;
        mBottom = bottom;
        mRenderNode.setLeftTopRightBottom(mLeft, mTop, mRight, mBottom);
        ...
    }
    return changed;
}

```

setFrame 方法用传进来的 l、t、r、b 这 4 个参数分别初始化 mLeft、mTop、mRight、mBottom 这 4 个值，这样就确定了该 View 在父容器中的位置。在调用 setFrame 方法后，会调用 onLayout 方法：

```
protected void onLayout(boolean changed, int left, int top, int right, int bottom)
{
}

```

`onLayout` 方法是一个空方法，这和 `onMeasure` 方法类似。确定位置时根据不同的控件有不同的实现，所以在 `View` 和 `ViewGroup` 中均没有实现 `onLayout` 方法。既然这样，我们下面就来查看 `LinearLayout` 的 `onLayout` 方法：

```
@Override
protected void onLayout(boolean changed, int l, int t, int r, int b) {
    if (mOrientation == VERTICAL) {
        layoutVertical(l, t, r, b);
    } else {
        layoutHorizontal(l, t, r, b);
    }
}

```

与 `onMeasure` 方法类似，根据方向来调用不同的方法。这里仍旧查看垂直方向的 `layoutVertical` 方法，如下所示：

```
void layoutVertical(int left, int top, int right, int bottom) {
    ...
    for (int i = 0; i < count; i++) {
        final View child = getVirtualChildAt(i);
        if (child == null) {
            childTop += measureNullChild(i);
        } else if (child.getVisibility() != GONE) {
            final int childWidth = child.getMeasuredWidth();
            final int childHeight = child.getMeasuredHeight();
            final LinearLayout.LayoutParams lp =
                (LinearLayout.LayoutParams) child.getLayoutParams();
            int gravity = lp.gravity;
            if (gravity < 0) {
                gravity = minorGravity;
            }
            final int layoutDirection = getLayoutDirection();
            final int absoluteGravity = Gravity.getAbsoluteGravity(
                gravity, layoutDirection);
            switch (absoluteGravity & Gravity.HORIZONTAL_GRAVITY_MASK) {
                case Gravity.CENTER_HORIZONTAL:
                    childLeft = paddingLeft + ((childSpace - childWidth) / 2)
                        + lp.leftMargin - lp.rightMargin;
                    break;
            }
        }
    }
}

```

```

        case Gravity.RIGHT:
            childLeft = childRight - childWidth - lp.rightMargin;
            break;

        case Gravity.LEFT:
        default:
            childLeft = paddingLeft + lp.leftMargin;
            break;
    }

    if (hasDividerBeforeChildAt(i)) {
        childTop += mDividerHeight;
    }
    childTop += lp.topMargin;
    setChildFrame(child, childLeft, childTop + getLocationOffset(
        child), childWidth, childHeight);
    childTop += childHeight + lp.bottomMargin +
        getNextLocationOffset (child);
    i += getChildrenSkipCount(child, i);
}
}
}

```

这个方法会遍历子元素并调用 `setChildFrame` 方法。其中 `childTop` 值是不断累加的，这样子元素才会依次按照垂直方向一个接一个排列下去而不会是重叠的。接着看 `setChildFrame` 方法：

```

private void setChildFrame(View child, int left, int top, int width, int
height) {
    child.layout(left, top, left + width, top + height);
}

```

在 `setChildFrame` 方法中调用子元素的 `layout` 方法来确定自己的位置。

3.7.5 View 的 draw 流程

View 的 draw 流程很简单，下面先来看看 View 的 draw 方法。

官方注释清楚地说明了每一步的做法，它们分别是：

- (1) 如果需要，则绘制背景。
- (2) 保存当前 canvas 层。
- (3) 绘制 View 的内容。
- (4) 绘制子 View。
- (5) 如果需要，则绘制 View 的褪色边缘，这类似于阴影效果。
- (6) 绘制装饰，比如滚动条。

其中第 2 步和第 5 步可以跳过，所以这里不做分析，重点分析其他步骤。

步骤 1：绘制背景

绘制背景调用了 View 的 drawBackground 方法，如下所示：

```
private void drawBackground(Canvas canvas) {
    final Drawable background = mBackground;
    if (background == null) {
        return;
    }
    setBackgroundBounds();
    ...
    final int scrollX = mScrollX;
    final int scrollY = mScrollY;
    if ((scrollX | scrollY) == 0) { //1
        background.draw(canvas);
    } else {
        canvas.translate(scrollX, scrollY);
        background.draw(canvas);
        canvas.translate(-scrollX, -scrollY);
    }
}
```

从上面代码注释 1 处可看出绘制背景考虑了偏移参数 scrollX 和 scrollY。如果有偏移值不为 0，则会在偏移后的 canvas 绘制背景。

步骤 3：绘制 View 的内容

步骤 3 调用了 View 的 onDraw 方法。这个方法是一个空实现，因为不同的 View 有着不同的内容，这需要我们自己去实现，即在自定义 View 中重写该方法来实现：

```
protected void onDraw(Canvas canvas) {
}
```

步骤 4：绘制子 View

步骤 4 调用了 dispatchDraw 方法，这个方法也是一个空实现，如下所示：

```
protected void dispatchDraw(Canvas canvas) {
}
```

ViewGroup 重写了这个方法，紧接着看看 ViewGroup 的 dispatchDraw 方法：

```
protected void dispatchDraw(Canvas canvas) {
    ...
    for (int i = 0; i < childrenCount; i++) {
```

```

while (transientIndex >= 0 && mTransientIndices.get(transientIndex)
== i) {
    final View transientChild = mTransientViews.get(transientIndex);
    if ((transientChild.mViewFlags & VISIBILITY_MASK) == VISIBLE ||
        transientChild.getAnimation() != null) {
        more |= drawChild(canvas, transientChild, drawingTime);
    }
    transientIndex++;
    if (transientIndex >= transientCount) {
        transientIndex = -1;
    }
}
...
}

```

源码很长，这里截取了关键的部分，在 `dispatchDraw` 方法中对子类 `View` 进行遍历，并调用 `drawChild` 方法：

```

protected boolean drawChild(Canvas canvas, View child, long drawingTime) {
    return child.draw(canvas, this, drawingTime);
} public void draw(Canvas canvas) {
    final int privateFlags = mPrivateFlags;
    final boolean dirtyOpaque = (privateFlags & PFLAG_DIRTY_MASK) ==
PFLAG_DIRTY_OPAQUE &&
        (mAttachInfo == null || !mAttachInfo.mIgnoreDirtyState);
    mPrivateFlags = (privateFlags & ~PFLAG_DIRTY_MASK) | PFLAG_DRAWN;
    int saveCount;
    if (!dirtyOpaque) {
        drawBackground(canvas);
    }
    final int viewFlags = mViewFlags;
    boolean horizontalEdges = (viewFlags & FADING_EDGE_HORIZONTAL) != 0;
    boolean verticalEdges = (viewFlags & FADING_EDGE_VERTICAL) != 0;
    if (!verticalEdges && !horizontalEdges) {
        if (!dirtyOpaque) onDraw(canvas);
        dispatchDraw(canvas);
        if (mOverlay != null && !mOverlay.isEmpty()) {
            mOverlay.getOverlayView().dispatchDraw(canvas);
        }
        onDrawForeground(canvas);
        return;
    }
}

```

这里主要调用了 `View` 的 `draw` 方法，代码如下所示：

```

boolean draw(Canvas canvas, ViewGroup parent, long drawingTime) {
    ...
    if (!drawingWithDrawingCache) { //1
        if (drawingWithRenderNode) {
            mPrivateFlags &= ~PFLAG_DIRTY_MASK;
            ((DisplayListCanvas) canvas).drawRenderNode(renderNode);
        } else {
            if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
                mPrivateFlags &= ~PFLAG_DIRTY_MASK;
                dispatchDraw(canvas);
            } else {
                draw(canvas);
            }
        }
    } else if (cache != null) {
        mPrivateFlags &= ~PFLAG_DIRTY_MASK;
        if (layerType == LAYER_TYPE_NONE) {
            Paint cachePaint = parent.mCachePaint;
            if (cachePaint == null) {
                cachePaint = new Paint();
                cachePaint.setDither(false);
                parent.mCachePaint = cachePaint;
            }
            cachePaint.setAlpha((int) (alpha * 255));
            canvas.drawBitmap(cache, 0.0f, 0.0f, cachePaint);
        } else {
            int layerPaintAlpha = mLayerPaint.getAlpha();
            mLayerPaint.setAlpha((int) (alpha * layerPaintAlpha));
            canvas.drawBitmap(cache, 0.0f, 0.0f, mLayerPaint);
            mLayerPaint.setAlpha(layerPaintAlpha);
        }
    }
}

```

源码很长，我们挑重点的看。在上面代码注释 1 处判断是否有缓存，如果没有则正常绘制，如果有则利用缓存显示。

步骤 6：绘制装饰

绘制装饰的方法为 View 的 onDrawForeground 方法：

```

public void onDrawForeground(Canvas canvas) {
    onDrawScrollIndicators(canvas);
    onDrawScrollBars(canvas);
    final Drawable foreground = mForegroundInfo != null ? mForegroundInfo.

```

```

mDrawable : null;
if (foreground != null) {
    if (mForegroundInfo.mBoundsChanged) {
        mForegroundInfo.mBoundsChanged = false;
        final Rect selfBounds = mForegroundInfo.mSelfBounds;
        final Rect overlayBounds = mForegroundInfo.mOverlayBounds;
        if (mForegroundInfo.mInsidePadding) {
            selfBounds.set(0, 0, getWidth(), getHeight());
        } else {
            selfBounds.set(getPaddingLeft(), getPaddingTop(),
                getWidth() - getPaddingRight(), getHeight() -
                getPaddingBottom());
        }

        final int ld = getLayoutDirection();
        Gravity.apply(mForegroundInfo.mGravity, foreground.
            getIntrinsicWidth(), foreground.getIntrinsicHeight(),
            selfBounds, overlayBounds, ld);
        foreground.setBounds(overlayBounds);
    }
    foreground.draw(canvas);
}
}

```

很明显这个方法用于绘制 `ScrollBar` 以及其他的装饰，并将它们绘制在视图内容的上层。

3.8 自定义 View

经过前面的铺垫，接下来讲讲自定义 `View`。自定义 `View` 一直被认为是高手掌握的技能，因为情况太多，想实现的效果又变化多端。但它也要遵循一定的规则，我们要讲的就是这个规则，至于那些变化多端的酷炫效果就由各位读者来慢慢发挥了。但是需要注意的是，凡事都要有度，自定义 `View` 毕竟不是规范的控件，如果设计不好、不考虑性能反而会适得其反，另外适配起来可能也会产生问题。笔者的建议是，如果能用系统控件的情况还是应尽量用系统控件。在阅读本节之前，希望读者能把本章前面的内容都读完，因为学习自定义 `View` 需要了解 `View` 的层次、`View` 的事件分发机制和 `View` 的工作流程。自定义 `View` 按照笔者的划分，分为三大类，第一种是自定义 `View`，第二种是自定义 `ViewGroup`，第三种是自定义组合控件。其中自定义 `View` 又分为继承系统控件（比如 `TextView`）和继承 `View` 两种。自定义 `ViewGroup` 也分为继承 `ViewGroup` 和继承系统特定的 `ViewGroup`（比如 `RelativeLayout`）。接下来就分别介绍自定义 `View` 的使用方法。

3.8.1 继承系统控件的自定义 View

这种自定义 View 在系统控件的基础上进行拓展，一般是添加新的功能或者修改显示的效果，一般情况下在 onDraw()方法中进行处理。这里举一个简单的例子，写一个自定义 View，继承自 TextView：

```
public class InvalidTextView extends TextView {
    private Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    public InvalidTextView(Context context) {
        super(context);
        initDraw();
    }
    public InvalidTextView(Context context, AttributeSet attrs) {
        super(context, attrs);
        initDraw();
    }
    public InvalidTextView(Context context, AttributeSet attrs, int
defStyleAttr) {
        super(context, attrs, defStyleAttr);
        initDraw();
    }
    private void initDraw() {
        mPaint.setColor(Color.RED);
        mPaint.setStrokeWidth((float) 1.5);
    }
    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        int width = getWidth();
        int height = getHeight();
        canvas.drawLine(0, height / 2, width, height / 2, mPaint);
    }
}
```

这个自定义 View 继承了 TextView，并且在 onDraw()方法中画了一条红色的横线。接下来在布局中引用这个 InvalidTextView，代码如下所示：

```
<com.example.liuwangshu.mooncustomview.InvalidTextView
    android:id="@+id/iv_text"
    android:layout_width="200dp"
    android:layout_height="100dp"
    android:background="@android:color/holo_blue_light"
    android:gravity="center"
    android:textSize="16sp"
    android:layout_centerHorizontal="true"
/>
```

运行程序，效果如图 3-9 所示。



图 3-9 继承系统控件的自定义 View

3.8.2 继承 View 的自定义 View

与上面的继承系统控件的自定义 View 不同，继承 View 的自定义 View 实现起来要稍微复杂一些。其不只是要实现 `onDraw()` 方法，而且在实现过程中还要考虑到 `wrap_content` 属性以及 `padding` 属性的设置；为了方便配置自己的自定义 View，还会对外提供自定义的属性。另外，如果要改变触控的逻辑，还要重写 `onTouchEvent()` 等触控事件的方法。按照上面的例子我们再写一个 `RectView` 类继承 View 来画一个正方形，代码如下所示。

1. 简单实现继承 View 的自定义 View

```
public class RectView extends View {
    private Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    private int mColor=Color.RED;
    public RectView(Context context) {
        super(context);
        initDraw();
    }
    public RectView(Context context, AttributeSet attrs) {
        super(context, attrs);
        initDraw();
    }
    public RectView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }
}
```

```

        initDraw();
    }
    private void initDraw() {
        mPaint.setColor(mColor);
        mPaint.setStrokeWidth((float) 1.5);
    }
    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        int width = getWidth();
        int height = getHeight();
        canvas.drawRect(0, 0, width, height, mPaint);
    }
}

```

最后在布局中引用 RectView，如下所示：

```

<com.example.liuwangshu.mooncustomview.RectView
    android:id="@+id/rv_rect"
    android:layout_width="200dp"
    android:layout_height="200dp"
    android:layout_below="@id/iv_text"
    android:layout_marginTop="50dp"
    android:layout_centerHorizontal="true"/>

```

运行程序查看效果，如图 3-10 所示。

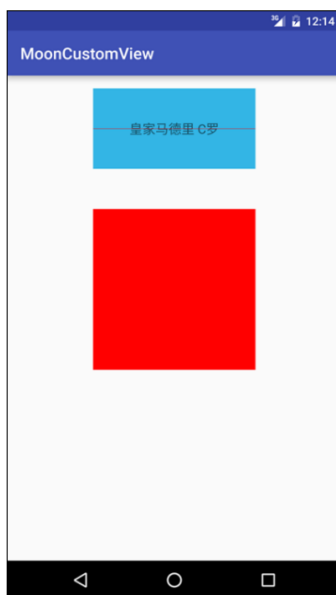


图 3-10 简单实现继承 View 的自定义 View

2. 对 padding 属性进行处理

修改布局文件，设置 padding 属性，如下所示：

```
<com.example.liuwangshu.mooncustomview.RectView
    android:id="@+id/rv_rect"
    android:layout_width="200dp"
    android:layout_height="200dp"
    android:layout_below="@id/iv_text"
    android:layout_marginTop="50dp"
    android:layout_centerHorizontal="true"
    android:padding="20dp"/>
```

运行程序，发现没有任何作用。看来还得对 padding 属性进行处理。只需要在 onDraw() 方法中稍加修改，在绘制正方形的时候考虑 padding 属性即可，代码如下所示：

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    int paddingLeft=getPaddingLeft();
    int paddingRight=getPaddingRight();
    int paddingTop=getPaddingTop();
    int paddingBottom=getPaddingBottom();
    int width = getWidth()-paddingLeft-paddingRight;
    int height = getHeight()-paddingTop-paddingBottom;
    canvas.drawRect(0+paddingLeft, 0+paddingTop, width+paddingRight,
        height+paddingBottom, mPaint);
}
```

运行程序，效果如图 3-11 所示。与图 3-10 对比一下，可以发现设置的 padding 属性确实生效了。

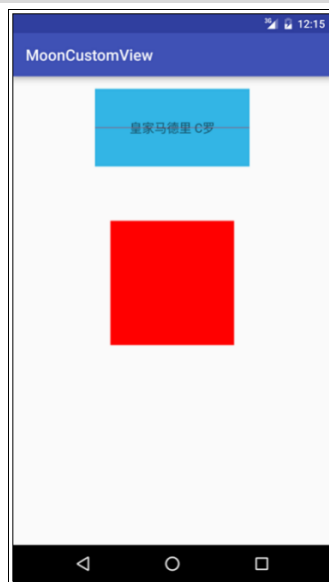


图 3-11 对 padding 属性进行处理

3. 对 wrap_content 属性进行处理

修改布局文件，让 RectView 的宽度分别为 wrap_content 和 match_parent 时的效果都是一样的，如图 3-12 所示。

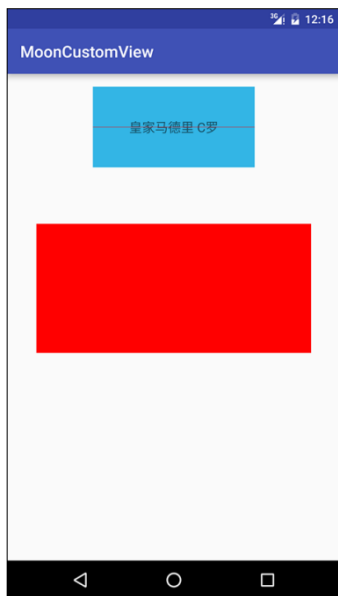


图 3-12 未对 wrap_content 属性进行处理的情况

至于为何会产生这种情况，这在 3.7.3 节中已经说得很清楚了，这里就不赘述了。对于这种情况需要我们在 onMeasure 方法中指定一个默认的宽和高，在设置 wrap_content 属性时设置此默认的宽和高就可以了：

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int widthSpecMode = MeasureSpec.getMode(widthMeasureSpec);
    int heightSpecMode = MeasureSpec.getMode(heightMeasureSpec);
    int widthSpecSize=MeasureSpec.getSize(widthMeasureSpec);
    int heightSpecSize=MeasureSpec.getSize(heightMeasureSpec);
    if (widthSpecMode==MeasureSpec.AT_MOST&&heightSpecMode==
    MeasureSpec.AT_MOST) {
        setMeasuredDimension(600,600);
    }else if (widthSpecMode==MeasureSpec.AT_MOST) {
        setMeasuredDimension(600,heightSpecSize);
    }else if (heightSpecMode==MeasureSpec.AT_MOST) {
        setMeasuredDimension(widthSpecSize, 600);
    }
}
```

需要注意的是 `setMeasuredDimension()` 方法接收的参数单位是 `px`，来看看效果，如图 3-13 所示。

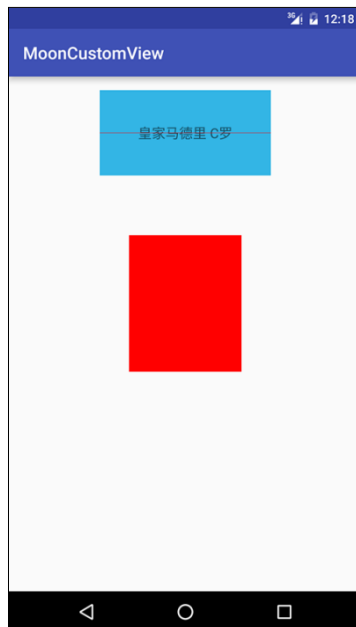


图 3-13 对 `wrap_content` 属性进行处理的情况

4. 自定义属性

Android 系统的控件以 `android` 开头的（比如 `android:layout_width`）都是系统自带的属性。为了方便配置 `RectView` 的属性，我们也可以自定义属性。首先在 `values` 目录下创建 `attrs.xml`：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="RectView">
        <attr name="rect_color" format="color" />
    </declare-styleable>
</resources>
```

这个配置文件定义了名为 `RectView` 的自定义属性组合。我们定义了 `rect_color` 属性，它的格式为 `color`。接下来在 `RectView` 的构造方法中解析自定义属性的值，如下所示：

```
public RectView(Context context, AttributeSet attrs) {
    super(context, attrs);
    TypedArray mTypedArray=context.obtainStyledAttributes(attrs,
        R.styleable.RectView);
    //提取 RectView 属性集合的 rect_color 属性。如果没设置，默认值为 Color.RED
    mColor=mTypedArray.getColor(R.styleable.RectView_rect_color,
        Color.RED);
}
```

```

        //获取资源后要及时回收
        mTypedArray.recycle();
        initDraw();
    }

```

用 `TypedArray` 来获取自定义的属性集 `R.styleable.RectView`，这个 `RectView` 就是我们在 XML 中定义的 `name` 的值，然后通过 `TypedArray` 的 `getColor` 方法来获取自定义的属性值。最后修改布局文件，如下所示：

```

<com.example.liuwangshu.mooncustomview.RectView
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/rv_rect"
    android:layout_width="wrap_content"
    android:layout_height="200dp"
    android:layout_below="@id/iv_text"
    android:layout_marginTop="50dp"
    android:layout_centerHorizontal="true"
    android:padding="10dp"
    app:rect_color="@android:color/holo_blue_light"
/>

```

使用自定义属性需要添加 `schemas`：`xmlns:app="http://schemas.android.com/apk/res-auto"`，其中 `app` 是我们自定义的名字。最后我们配置新定义的 `app:rect_color` 属性为 `android:color/holo_blue_light`。运行程序发现 `RectView` 的颜色变成了蓝色。`RectView` 的完整代码，如下所示：

```

public class RectView extends View {
    private Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    private int mColor=Color.RED;
    public RectView(Context context) {
        super(context);
        initDraw();
    }
    public RectView(Context context, AttributeSet attrs) {
        super(context, attrs);
        TypedArray mTypedArray=context.obtainStyledAttributes(attrs,
            R.styleable.RectView);
        //提取 RectView 属性集合的 rect_color 属性。如果没设置，默认值为 Color.RED
        mColor=mTypedArray.getColor(R.styleable.RectView_rect_color,
            Color.RED);
        //获取资源后要及时回收
        mTypedArray.recycle();
        initDraw();
    }
    public RectView(Context context, AttributeSet attrs, int defStyleAttr) {

```

```

        super(context, attrs, defStyleAttr);
        initDraw();
    }

    private void initDraw() {
        mPaint.setColor(mColor);
        mPaint.setStrokeWidth((float) 1.5);
    }
    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec);
        int widthSpecMode = MeasureSpec.getMode(widthMeasureSpec);
        int heightSpecMode = MeasureSpec.getMode(heightMeasureSpec);
        int widthSpecSize=MeasureSpec.getSize(widthMeasureSpec);
        int heightSpecSize=MeasureSpec.getSize(heightMeasureSpec);
        if (widthSpecMode==MeasureSpec.AT_MOST&&heightSpecMode==
            MeasureSpec.AT_MOST) {
            setMeasuredDimension(400,400);
        }else if (widthSpecMode==MeasureSpec.AT_MOST) {
            setMeasuredDimension(400,heightSpecSize);
        }else if (heightSpecMode==MeasureSpec.AT_MOST) {
            setMeasuredDimension(widthSpecSize,400);
        }
    }
    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        int paddingLeft = getPaddingLeft();
        int paddingRight = getPaddingRight();
        int paddingTop = getPaddingTop();
        int paddingBottom = getPaddingBottom();
        int width = getWidth() - paddingLeft - paddingRight;
        int height = getHeight() - paddingTop - paddingBottom;
        canvas.drawRect(0 + paddingLeft, 0 + paddingTop, width + paddingRight,
            height + paddingBottom, mPaint);
    }
}

```

3.8.3 自定义组合控件

前面讲到了自定义 View，下面接着讲常用的自定义组合控件。自定义组合控件就是多个控件组合起来成为一个新的控件，其主要用于解决多次重复地使用同一类型的布局。比如我们应用的顶部标题栏及弹出的固定样式的 Dialog，这些都是常用的，所以把它们所需要的控件组合

起来重新定义成一个新的控件。本节就来自定义一个顶部的标题栏。当然，实现标题栏有很多方法，下面来看看用自定义组合控件如何去实现。首先，我们定义组合控件的布局：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout_titlebar_rootlayout"
    android:layout_width="fill_parent"
    android:layout_height="45dp"
    >
    <ImageView
        android:id="@+id/iv_titlebar_left"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_alignParentLeft="true"
        android:layout_centerInParent="true"
        android:paddingLeft="15dp"
        android:paddingRight="15dp"
        android:src="@drawable/ico_return"
    />
    <TextView
        android:id="@+id/tv_titlebar_title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:maxEms="11"
        android:singleLine="true"
        android:ellipsize="end"
        android:textStyle="bold"/>
    <ImageView
        android:id="@+id/iv_titlebar_right"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_alignParentRight="true"
        android:layout_centerInParent="true"
        android:src="@drawable/title_right"
        android:gravity="center"
        android:padding="15dp"
    />
</RelativeLayout>
```

这是很简单的布局，左右边各一个图标，中间是标题文字。接下来编写 Java 代码。因为我

们的组合控件整体布局是 **RelativeLayout**，所以组合控件要继承 **RelativeLayout**，代码如下所示：

```
public class TitleBar extends RelativeLayout {
    private ImageView iv_titlebar_left;
    private ImageView iv_titlebar_right;
    private TextView tv_titlebar_title;
    private RelativeLayout layout_titlebar_rootlayout;
    private int mColor= Color.BLUE;
    private int mTextColor= Color.WHITE;
    public TitleBar(Context context) {
        super(context);
        initView(context);
    }

    public TitleBar(Context context, AttributeSet attrs) {
        super(context, attrs);
        initView(context);
    }

    public TitleBar(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        initView(context);
    }

    public void initView(Context context){
        LayoutInflater.from(context).inflate(R.layout.view_customtitle,
            this, true);
        iv_titlebar_left= (ImageView) findViewById(R.id.iv_titlebar_left);
        iv_titlebar_right= (ImageView) findViewById(R.id.iv_titlebar_right);
        tv_titlebar_title= (TextView) findViewById(R.id.tv_titlebar_title);
        layout_titlebar_rootlayout= (RelativeLayout) findViewById(R.id.layout_
            titlebar_rootlayout);
        //设置背景颜色
        layout_titlebar_rootlayout.setBackgroundColor(mColor);
        //设置标题文字颜色
        tv_titlebar_title.setTextColor(mTextColor);
    }
    public void setTitle(String titlename){
        if(!TextUtils.isEmpty(titlename)) {
            tv_titlebar_title.setText(titlename);
        }
    }

    public void setLeftListener(OnClickListener onClickListener){
```

```

        iv_titlebar_left.setOnClickListener(onClickListener);
    }
    public void setRightListener(OnClickListener onClickListener){
        iv_titlebar_right.setOnClickListener(onClickListener);
    }
}

```

这里重写了 3 个构造方法并在构造方法中加载布局文件，对外提供了 3 个方法，分别用来设置标题的名字，以及左右按钮的点击事件。前面讲到了自定义属性，这里同样使用自定义属性，在 `values` 目录下创建 `attrs.xml`，代码如下所示：

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="TitleBar">
        <attr name="title_text_color" format="color" />
        <attr name="title_bg" format="color" />
        <attr name="title_text" format="string" />
    </declare-styleable>
</resources>

```

我们定义了 3 个属性，分别用来设置顶部标题栏的背景颜色、标题文字颜色和标题文字。为了引入自定义属性，需要在 `TitleBar` 的构造方法中解析自定义属性的值，代码如下所示：

```

public TitleBar(Context context, AttributeSet attrs) {
    super(context, attrs);
    TypedArray mTypedArray=context.obtainStyledAttributes(attrs,
        R.styleable.TitleBar);
    mColor=mTypedArray.getColor(R.styleable.TitleBar_title_bg,
        Color.BLUE);
    mTextColor=mTypedArray.getColor(R.styleable.TitleBar_title_text_color,
        Color.WHITE);
    titlename=mTypedArray.getString(R.styleable.TitleBar_title_text);
    mTypedArray.recycle();
    initView(context);
}

```

接下来引用组合控件的布局。使用自定义属性需要添加 `schemas: xmlns:app="http://schemas.android.com/apk/res-auto"`，其中，`app` 是我们自定义的名字，当然也可以取其他的名字：

```

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

```

```

<com.example.liuwangshu.mooncustomgroup.TitleBar
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/title"
    android:layout_width="match_parent"
    android:layout_height="45dp"
    app:title_text="自定义组合控件"
    app:title_bg="@android:color/holo_orange_dark"
    app:title_text_color="@android:color/holo_blue_dark">
</com.example.liuwangshu.mooncustomgroup.TitleBar>
</LinearLayout>

```

最后，在主界面调用自定义的 TitleBar，并设置了左右两边按钮的点击事件：

```

public class MainActivity extends Activity {
    private TitleBar mTitleBar;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mTitleBar= (TitleBar) this.findViewById(R.id.title);
//        mTitleBar.setTitle("自定义组合控件");

        mTitleBar.setLeftListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(MainActivity.this, "点击左键", Toast.LENGTH_
                    SHORT).show();
            }
        });

        mTitleBar.setRightListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(MainActivity.this, "点击右键", Toast.LENGTH_
                    SHORT).show();
            }
        });
    }
}

```

运行程序，效果如图 3-14 所示。



图 3-14 自定义标题栏

3.8.4 自定义 ViewGroup

前面介绍过自定义 ViewGroup 又分为继承 ViewGroup 和继承系统特定的 ViewGroup（比如 RelativeLayout），其中继承系统特定的 ViewGroup 比较简单，在这里就不做介绍了，主要介绍继承 ViewGroup。本节的例子是一个自定义的 ViewGroup，左右滑动切换不同的页面，类似一个特别简化的 ViewPager。其会涉及本章很多节的内容，比如 View 的工作流程、View 的滑动等，所以，对 View 体系不太了解的读者先从头阅读本章，再来看本节会更好些。需要注意的是，要实现一个自定义的 ViewGroup 是很复杂的，这一点看 LineraLayout 等的源码就会知道，在此只实现主要的功能就好了。

1. 继承 ViewGroup

要实现自定义的 ViewGroup，首先要继承 ViewGroup 并调用父类的构造方法，实现抽象方法等：

```
public class HorizontalView extends ViewGroup{
    public HorizontalView(Context context) {
        super(context);
    }
    public HorizontalView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    public HorizontalView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    @Override
    protected void onLayout(boolean changed, int l, int t, int r, int b) {
    }
}
```

这里我们定义了名为 HorizontalView 的类并继承 ViewGroup，onLayout 这个抽象方法是必须要实现的，我们暂且什么都不做。

2. 对 wrap_content 属性进行处理

至于为什么要对 wrap_content 属性进行处理，这在 3.7.3 节中已经说得很清楚了，这里就不

赘述了。接下来的代码对 `wrap_content` 属性进行处理，其中省略了此前的构造方法代码：

```
public class HorizontalView extends ViewGroup {
    ...省略此前的构造方法代码
    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec);
        int widthMode = MeasureSpec.getMode(widthMeasureSpec);
        int widthSize = MeasureSpec.getSize(widthMeasureSpec);
        int heightMode = MeasureSpec.getMode(heightMeasureSpec);
        int heightSize = MeasureSpec.getSize(heightMeasureSpec);

        measureChildren(widthMeasureSpec, heightMeasureSpec);
        //如果没有子元素，就设置宽和高都为 0
        if (getChildCount() == 0) {
            setMeasuredDimension(0, 0);
        }
        //宽和高都是 AT_MOST，则宽度设置为所有子元素宽度的和，高度设置为第一个子元素的高度
        else if (widthMode == MeasureSpec.AT_MOST && heightMode ==
            MeasureSpec.AT_MOST) {
            View childOne = getChildAt(0);
            int childWidth = childOne.getMeasuredWidth();
            int childHeight = childOne.getMeasuredHeight();
            setMeasuredDimension(childWidth * getChildCount(), childHeight);
        }
        //宽度是 AT_MOST，则宽度为所有子元素宽度的和
        else if (widthMode == MeasureSpec.AT_MOST) {
            int childWidth = getChildAt(0).getMeasuredWidth();
            setMeasuredDimension(childWidth * getChildCount(), heightSize);
        }
        //高度是 AT_MOST，则高度为第一个子元素的高度
        else if (heightMode == MeasureSpec.AT_MOST) {
            int childHeight = getChildAt(0).getMeasuredHeight();
            setMeasuredDimension(widthSize, childHeight);
        }
    }
    @Override
    protected void onLayout(boolean changed, int l, int t, int r, int b) {
    }
}
```

这里如果没有子元素，则采用简化的写法，将宽和高直接设置为 0。正常的话，我们应该根据 `LayoutParams` 中的宽和高来做相应的处理。接着根据 `widthMode` 和 `heightMode` 来分别设置 `HorizontalView` 的宽和高。另外，我们在测量时没有考虑 `HorizontalView` 的 `padding` 和子元素的 `margin`。

3. 实现 onLayout

接下来实现 onLayout 来布局子元素。因为对每一种布局方式，子 View 的布局都是不同的，所以这是 ViewGroup 唯一一个抽象方法，需要我们自己去实现，代码如下所示：

```
public class HorizontalView extends ViewGroup {
    ... 省略构造方法的代码和 onMeasure 的代码
    @Override
    protected void onLayout(boolean changed, int l, int t, int r, int b) {
        int childCount = getChildCount();
        int left = 0;
        View child;
        for (int i = 0; i < childCount; i++) {
            child = getChildAt(i);
            if (child.getVisibility() != View.GONE) {
                int width = child.getMeasuredWidth();
                childWidth = width;
                child.layout(left, 0, left + width, child.getMeasuredHeight());
                left += width;
            }
        }
    }
}
```

遍历所有的子元素。如果子元素不是 GONE，则调用子元素的 layout 方法将其放置到合适的位置上。这相当于默认第一个子元素占满了屏幕，后面的子元素就是在第一个屏幕后面紧挨着和屏幕一样大小的后续元素。所以 left 是一直累加的，top 保持为 0，bottom 保持为第一个元素的高度，right 就是 left+元素的宽度。同样，这里没有处理 HorizontalView 的 padding 以及子元素的 margin。

4. 处理滑动冲突

这个自定义 ViewGroup 为水平滑动，如果里面是 ListView，ListView 则为垂直滑动，这样会导致滑动的冲突。解决的方法就是，如果我们检测到的滑动方向是水平的话，就让父 View 进行拦截，确保父 View 用来进行 View 的滑动切换。

```
public class HorizontalView extends ViewGroup {
    private int lastInterceptX;
    private int lastInterceptY;
    private int lastX;
    private int lastY;
    ... 省略了构造方法的代码

    @Override
    public boolean onInterceptTouchEvent(MotionEvent event) {
        boolean intercept = false;
```

```

        int x = (int) event.getX();//1
        int y = (int) event.getY();
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                break;
            case MotionEvent.ACTION_MOVE:
                int deltaX = x - lastInterceptX;
                int deltaY = y - lastInterceptY;
                if (Math.abs(deltaX) - Math.abs(deltaY) > 0) { //2
                    intercept = true;
                }
                break;
            case MotionEvent.ACTION_UP:
                break;
        }
        lastX = x;
        lastY = y;
        lastInterceptX = x;
        lastInterceptY = y;
        return intercept;
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        return super.onTouchEvent(event);
    }
    ... 省略了 onMeasure 和 onLayout 的代码
}

```

在上面代码注释 1 处，在刚进入 `onInterceptTouchEvent` 方法时就得到了点击事件的坐标，在 `MotionEvent.ACTION_MOVE` 中计算每次手指移动的距离，并在注释 2 处判断用户是水平滑动还是垂直滑动，如果是水平滑动则设置 `intercept = true` 来进行拦截，这样事件则由 `HorizontalView` 的 `onTouchEvent` 方法来处理。

5. 弹性滑动到其他页面

接着处理 `onTouchEvent` 事件，在 `onTouchEvent` 方法里需要进行滑动切换页面，这里就需要用到 `Scroller`。在本章的 3.3.6 节中介绍了如何使用 `Scroller`，在本章的 3.5 节中我们通过源码解析了 `Scroller` 为何能够进行滑动，不了解的读者可以去查看相关内容。

```

public class HorizontalView extends ViewGroup {
    ... 省略构造方法, init 方法, onInterceptTouchEvent
    int lastInterceptX;
    int lastInterceptY;
    int lastX;
}

```

```

int lastY;
int currentIndex = 0; //当前子元素
int childWidth = 0;
private Scroller scroller;
@Override
public boolean onTouchEvent(MotionEvent event) {
    int x = (int) event.getX();
    int y = (int) event.getY();
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            break;
        case MotionEvent.ACTION_MOVE:
            int deltaX = x - lastX;
            scrollBy(-deltaX, 0);
            break;
        case MotionEvent.ACTION_UP:
            int distance = getScrollX() - currentIndex * childWidth;
            if (Math.abs(distance) > childWidth / 2) { //1
                if (distance > 0) {
                    currentIndex++;
                } else {
                    currentIndex--;
                }
            }
            smoothScrollTo(currentIndex * childWidth, 0); //2
            break;
    }
    lastX = x;
    lastY = y;
    return super.onTouchEvent(event);
}
...省略 onMeasure 方法
@Override
public void computeScroll() {
    super.computeScroll();
    if (scroller.computeScrollOffset()) {
        scrollTo(scroller.getCurrX(), scroller.getCurrY());
        postInvalidate();
    }
}
//弹性滑动到指定位置
public void smoothScrollTo(int destX, int destY) {
    scroller.startScroll(getScrollX(), getScrollY(), destX - getScrollX(),
        destY - getScrollY(), 1000);
}

```

```

        invalidate();
    }
    ...省略 onLayout 方法
}

```

同样，在刚进入 `onTouchEvent` 方法时就得到点击事件的坐标，在 `MotionEvent.ACTION_MOVE` 中用 `scrollBy` 方法来处理 `HorizontalView` 控件随手指滑动的效果。在上面代码注释 1 处判断滑动的距离是否大于宽度的 $1/2$ ，如果大于则切换到其他页面，然后调用 `Scroller` 来进行弹性滑动。

6. 快速滑动到其他页面

通常情况下，只在滑动超过一半时才切换到上/下一个页面是不够的。如果滑动速度很快的话，我们也可以判定为用户想要滑动到其他页面，这样的体验也是好的。需要在 `onTouchEvent` 方法的 `ACTION_UP` 中对快速滑动进行处理。在这里又需要用到 `VelocityTracker`，它是用来测试滑动速度的。使用方法也很简单，首先在构造方法中进行初始化，也就是在前面的 `init` 方法中增加一条语句，代码如下所示：

```

...
private VelocityTracker tracker;
...
public void init() {
    scroller = new Scroller(getContext());
    tracker=VelocityTracker.obtain();
}
...

```

在 `init` 方法中对 `VelocityTracker` 进行初始化。接着开始改写 `onTouchEvent` 部分，如下所示：

```

@Override
public boolean onTouchEvent(MotionEvent event) {
    ...
    case MotionEvent.ACTION_UP:
        int distance = getScrollX() - currentIndex * childWidth;
        if (Math.abs(distance) > childWidth / 2) {
            if (distance > 0) {
                currentIndex++;
            } else {
                currentIndex--;
            }
        }
        else {
            tracker.computeCurrentVelocity(1000); //1
            float xV = tracker.getXVelocity();

```

```

        if (Math.abs(xV) > 50) { //2
            //切换到上一个页面
            if (xV > 0) {
                currentIndex--;
            } //切换到下一个页面
            else {
                currentIndex++;
            }
        }
        currentIndex = currentIndex < 0 ? 0 : currentIndex > getChildCount()
            - 1 ? getChildCount() - 1 : currentIndex;
        smoothScrollTo(currentIndex * childWidth, 0);
        tracker.clear(); //3
        break;
    }
}

```

在上面代码注释 1 处获取水平方向的速度；接着在注释 2 处，如果速度的绝对值大于 50 的话，就被认为是“快速滑动”，执行切换页面。不要忘了在注释 3 处重置速度计算器。

7. 再次触摸屏幕阻止页面继续滑动

如果有如下的场景：当我们快速向左滑动切换到下一个页面时，在手指释放以后，页面会弹性滑动到下一个页面，这可能需要 1 秒才完成滑动，在这个时间内，我们再次触摸屏幕，希望能拦截这次滑动，然后再次去操作页面。要实现在弹性滑动过程中再次触摸拦截，肯定要在 `onInterceptTouchEvent` 的 `ACTION_DOWN` 中去判断。如果在 `ACTION_DOWN` 的时候，`Scroller` 还没有执行完毕，说明上一次的滑动还正在进行中，则直接中断 `Scroller`，代码如下所示：

```

...
@Override
public boolean onInterceptTouchEvent(MotionEvent event) {
    boolean intercept = false;
    int x = (int) event.getX();
    int y = (int) event.getY();
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            intercept = false;
            if (!scroller.isFinished()) { //1
                scroller.abortAnimation();
            }
            break;
        case MotionEvent.ACTION_MOVE:
            int deltaX = x - lastInterceptX;
            int deltaY = y - lastInterceptY;
            if (Math.abs(deltaX) - Math.abs(deltaY) > 0) {

```

```

        intercept = true;
    } else {
        intercept = false;
    }
    break;
case MotionEvent.ACTION_UP:
    intercept = false;
    break;
}
lastX = x//2
lastY = y;
lastInterceptX = x;
lastInterceptY = y;
return intercept;
}
...

```

主要逻辑在上面代码注释1处,如果 Scroller 没有执行完成,则调用 Scroller 的 abortAnimation 方法来打断 Scroller。因为 onInterceptTouchEvent 方法的 ACTION_DOWN 返回 false,所以在 onTouchEvent 方法中无法获取 DOWN 事件,故而需要在注释2处设置 lastX 和 lastY 这两个参数。

8. 应用 HorizontalView

首先,我们在主布局中引用 HorizontalView,它作为父容器,里面有两个 ListView。布局文件如下所示:

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.liuwangshu.mooncustomviewgroup.MainActivity">
    <com.example.liuwangshu.mooncustomviewgroup.HorizontalView
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <ListView
            android:id="@+id/lv_one"
            android:layout_width="match_parent"
            android:layout_height="match_parent"></ListView>
        <ListView
            android:id="@+id/lv_two"
            android:layout_width="match_parent"
            android:layout_height="match_parent"></ListView>
    </com.example.liuwangshu.mooncustomviewgroup.HorizontalView>
</RelativeLayout>

```


接着，在代码中为 ListView 填加数据：

```
public class MainActivity extends AppCompatActivity {
    private ListView lv_one;
    private ListView lv_two;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        lv_one=(ListView)this.findViewById(R.id.lv_one);
        lv_two=(ListView)this.findViewById(R.id.lv_two);
        String[] strs1 = {"1","2","3","4","5","6","7","8","9","10","11",
            "12","13","14","15"};
        ArrayAdapter<String> adapter1=new ArrayAdapter<String>
            (this,android.R.layout.simple_expandable_list_item_1,strs1);
        lv_one.setAdapter(adapter1);
        String[] strs2 = {"A","B","C","D","E","F","G","H","I","J","K","L",
            "M","N","O"};
        ArrayAdapter<String> adapter2 = new ArrayAdapter<String>
            (this,android.R.layout.simple_expandable_list_item_1,strs2);
        lv_two.setAdapter(adapter2);
    }
}
```

运行程序，效果如图 3-15 所示。当我们向右滑动的时候，界面会滑动到第二个 ListView，如图 3-16 所示。



图 3-15 初始的第一页



图 3-16 滑动到第二页

最后贴上 HorizontalView 的整个源码:

```
public class HorizontalView extends ViewGroup {
    private int lastX;
    private int lastY;
    private int currentIndex = 0; //当前子元素
    private int childWidth = 0;
    private Scroller scroller;
    private VelocityTracker tracker;
    private int lastInterceptX=0;
    private int lastInterceptY=0;
    public HorizontalView(Context context) {
        super(context);
        init();
    }
    public HorizontalView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }
    public HorizontalView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        init();
    }

    public void init() {
        scroller = new Scroller(getContext());
        tracker = VelocityTracker.obtain();
    }

    @Override
    public boolean onInterceptTouchEvent(MotionEvent event) {
        boolean intercept = false;
        int x = (int) event.getX();
        int y = (int) event.getY();
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                intercept = false;
                if (!scroller.isFinished()) {
                    scroller.abortAnimation();
                }
                break;
            case MotionEvent.ACTION_MOVE:
                int deltaX = x - lastInterceptX;
                int deltaY = y - lastInterceptY;
```

```

        if (Math.abs(deltaX) - Math.abs(deltaY) > 0) {
            intercept = true;
            Log.i("wangshu", "intercept = true");
        } else {
            intercept = false;
            Log.i("wangshu", "intercept = false");
        }
        break;
    case MotionEvent.ACTION_UP:
        intercept = false;
        break;
}
lastX = x;
lastY = y;
lastInterceptX = x;
lastInterceptY = y;
return intercept;
}

@Override
public boolean onTouchEvent(MotionEvent event) {
    tracker.addMovement(event);
    int x = (int) event.getX();
    int y = (int) event.getY();
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            if (!scroller.isFinished()) {
                scroller.abortAnimation();
            }
            break;
        case MotionEvent.ACTION_MOVE:
            int deltaX = x - lastX;
            scrollBy(-deltaX, 0);
            break;
        case MotionEvent.ACTION_UP:
            int distance = getScrollX() - currentIndex * childWidth;
            if (Math.abs(distance) > childWidth / 2) {
                if (distance > 0) {
                    currentIndex++;
                } else {
                    currentIndex--;
                }
            }
            } else {
                tracker.computeCurrentVelocity(1000);
            }
    }
}

```

```

        float xV = tracker.getXVelocity();
        if (Math.abs(xV) > 50) {
            if (xV > 0) {
                currentIndex--;
            } else {
                currentIndex++;
            }
        }
    }

    currentIndex = currentIndex < 0 ? 0 : currentIndex >
    getChildCount() - 1 ? getChildCount() - 1 : currentIndex;
    smoothScrollTo(currentIndex * childWidth, 0);
    tracker.clear();
    break;
default:
    break;
}
lastX = x;
lastY = y;
return true;
}

@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int widthMode = MeasureSpec.getMode(widthMeasureSpec);
    int widthSize = MeasureSpec.getSize(widthMeasureSpec);
    int heightMode = MeasureSpec.getMode(heightMeasureSpec);
    int heightSize = MeasureSpec.getSize(heightMeasureSpec);
    measureChildren(widthMeasureSpec, heightMeasureSpec);
    if (getChildCount() == 0) {
        setMeasuredDimension(0, 0);
    } else if (widthMode == MeasureSpec.AT_MOST && heightMode ==
    MeasureSpec.AT_MOST) {
        View childOne = getChildAt(0);
        int childWidth = childOne.getMeasuredWidth();
        int childHeight = childOne.getMeasuredHeight();
        setMeasuredDimension(childWidth * getChildCount(), childHeight);
    } else if (widthMode == MeasureSpec.AT_MOST) {
        View childOne = getChildAt(0);
        int childWidth = childOne.getMeasuredWidth();
        setMeasuredDimension(childWidth * getChildCount(), heightSize);
    } else if (heightMode == MeasureSpec.AT_MOST) {
        int childHeight = getChildAt(0).getMeasuredHeight();
        setMeasuredDimension(widthSize, childHeight);
    }
}

```

```

    }
}
@Override
public void computeScroll() {
    super.computeScroll();
    if (scroller.computeScrollOffset()) {
        scrollTo(scroller.getCurrX(), scroller.getCurrY());
        postInvalidate();
    }
}
public void smoothScrollTo(int destX, int destY) {
    scroller.startScroll(getScrollX(), getScrollY(), destX - getScrollX(),
        destY - getScrollY(), 1000);
    invalidate();
}
@Override
protected void onLayout(boolean changed, int l, int t, int r, int b) {
    int childCount = getChildCount();
    int left = 0;
    View child;
    for (int i = 0; i < childCount; i++) {
        child = getChildAt(i);
        if (child.getVisibility() != View.GONE) {
            int width = child.getMeasuredWidth();
            childWidth = width;
            child.layout(left, 0, left + width, child.getMeasuredHeight());
            left += width;
        }
    }
}
}
}
}

```

3.9 本章小结

到这里自定义 View 就讲完了。读到这里，很多读者会发现前面各节做的铺垫十分有必要：从最基础的 View 与 ViewGroup，到 View 的滑动以及 View 的事件分发机制和 View 的工作流程，这些都是为了最终讲解自定义 View。当然，自定义 View 作为一个技术难点，它有着十分多变的处理方式。但是不管它如何多变，都遵循着一定的规则，本章就讲解了这一规则。其他的变化需要我们自己去处理，但前提是要掌握本章的全部内容，打好基本功。这样才能以不变应万变，开发出更实用、更绚丽的自定义 View。

第 4 章

多线程编程

Android 沿用了 Java 的线程模型，一个 Android 应用在创建的时候会开启一个线程，我们叫它主线程或者 UI 线程。如果我们想要访问网络或者数据库等耗时的操作时，都会开启子线程去处理，从 Android 3.0 开始，系统要求网络访问必须在子线程中进行，否则会抛出异常；也就是为了避免主线程被耗时操作阻塞从而产生 ANR，同时也说明了多线程在 Android 应用开发占据着十分重要的位置。本章将介绍 Android 的多线程编程。这里会从浅入深，从线程的基础讲到同步再讲到阻塞队列和线程池等知识，最后结合这些知识来分析 AsyncTask 是如何运作的。

4.1 线程基础

讲到线程，当然要先讲线程的基础知识。在本节中，读者会了解线程和进程的区别、线程的创建、线程的状态等知识点，了解了这些知识点才可以更好地阅读本章后面小节的内容。

4.1.1 进程与线程

1. 什么是进程

提到线程时，不得不提到进程。我们首先了解一下什么是进程。进程是操作系统结构的基础，是程序在一个数据集上运行的过程，是系统进行资源分配和调度的基本单位。进程可以被看作程序的实体，同样，它也是线程的容器。上面这段话，比较抽象，对于进程可以看图 4-1 进行理解。该图是 Windows 任务管理器，里面列表中的 exe 程序就是一个进程。再举一个例子，如图 4-2 所示，这是 Android Device Monitor 中的 Devices 窗口，里面就是 Android 手机中运行的进程。进程就是程序的实体，是受操作系统管理的基本运行单元。



图 4-1 Windows 任务管理器

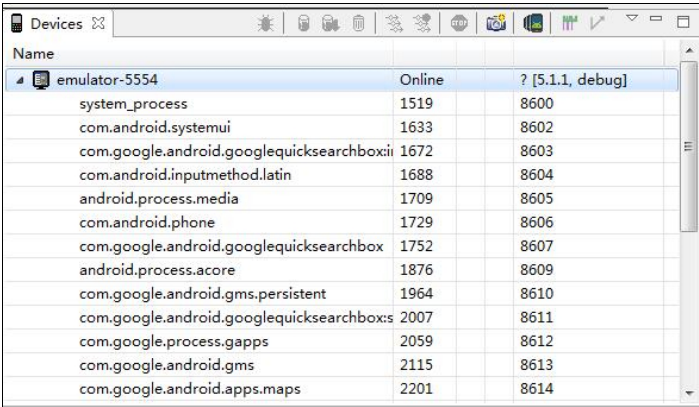


图 4-2 Devices 窗口

2. 什么是线程

例如图 4-1 中列表的第一项 qqbrowser.exe *32，这代表 QQ 浏览器的进程，它里面运行了很多子任务，这些子任务有的加载网页，有的处理缓存，有的进行下载，这些子任务就是线程，是操作系统调度的最小单元，也叫作轻量级进程。在一个进程中可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。

3. 为何要使用多线程

在操作系统级别上来看主要有以下几个方面：

- 使用多线程可以减少程序的响应时间。如果某个操作很耗时，或者陷入长时间的等待，此时程序将不会响应鼠标和键盘等的操作，使用多线程后可以把这个耗时的线程分配到一个单独的线程中去执行，从而使程序具备了更好的交互性。
- 与进程相比，线程创建和切换开销更小，同时多线程在数据共享方面效率非常高。
- 多 CPU 或者多核计算机本身就具备执行多线程的能力。如果使用单个进程，将无法重复利用计算机资源，这会造成资源的巨大浪费。在多 CPU 计算机中使用多线程能提高

CPU 的利用率。

- 使用多线程能简化程序的结构，使程序便于理解和维护。

4.1.2 线程的状态

Java 线程在运行的声明周期中可能会处于 6 种不同的状态，这 6 种线程状态分别为如下所示。

- **New:** 新创建状态。线程被创建，还没有调用 `start` 方法，在线程运行之前还有一些基础工作要做。
- **Runnable:** 可运行状态。一旦调用 `start` 方法，线程就处于 `Runnable` 状态。一个可运行的线程可能正在运行也可能没有运行，这取决于操作系统给线程提供运行的时间。
- **Blocked:** 阻塞状态。表示线程被锁阻塞，它暂时不活动。
- **Waiting:** 等待状态。线程暂时不活动，并且不运行任何代码，这消耗最少的资源，直到线程调度器重新激活它。
- **Timed waiting:** 超时等待状态。和等待状态不同的是，它是可以在指定的时间自行返回的。
- **Terminated:** 终止状态。表示当前线程已经执行完毕。导致线程终止有两种情况：第一种就是 `run` 方法执行完毕正常退出；第二种就是因为一个没有捕获的异常而终止了 `run` 方法，导致线程进入终止状态。

线程的状态如图 4-3 所示。

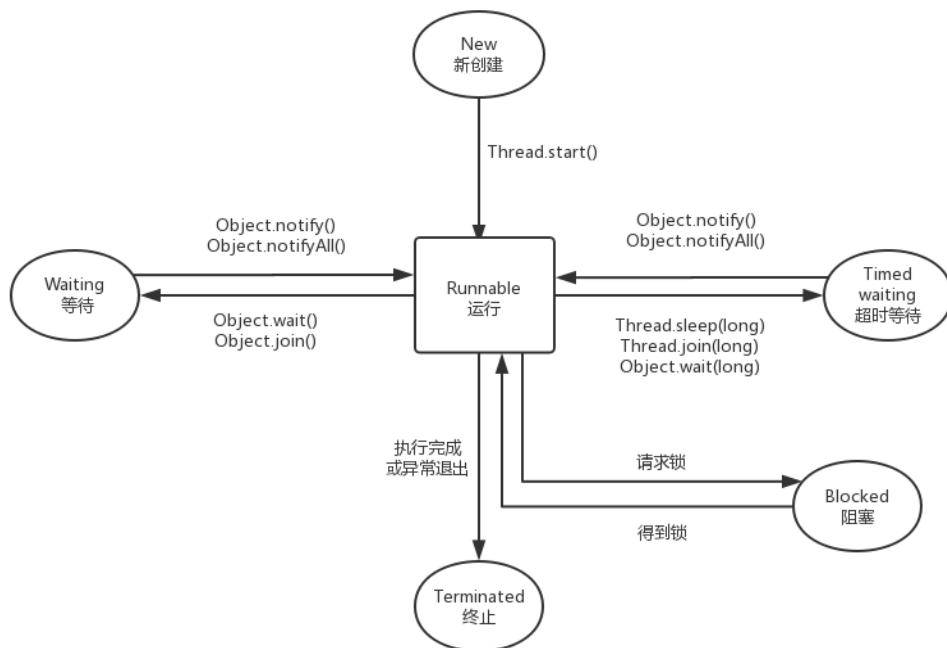


图 4-3 线程的状态

如图 4-3 所示，线程创建后，调用 `Thread` 的 `start` 方法，开始进入运行状态，当线程执行 `wait` 方法后，线程进入等待状态，进入等待状态的线程需要其他线程通知才能返回运行状态。超时等待相当于在等待状态加上了时间限制，如果超过时间限制，则线程返回运行状态。当线程调用到同步方法时，如果线程没有获得锁则进入阻塞状态，当阻塞状态的线程获取到锁时则重新回到运行状态。当线程执行完毕或者遇到意外异常终止时，都会进入终止状态。

4.1.3 创建线程

多线程的实现一般有以下 3 种方法，其中前两种为最常用的方法。

1. 继承 `Thread` 类，重写 `run()` 方法

`Thread` 本质上也是实现了 `Runnable` 接口的一个实例。需要注意的是调用 `start()` 方法后并不是立即地执行多线程的代码，而是使该线程变为可运行态，什么时候运行多线程代码是由操作系统决定的。以下是其主要步骤：

- (1) 定义 `Thread` 类的子类，并重写该类的 `run` 方法，该 `run` 方法的方法体就代表了线程要完成的任务。因此，`run()` 方法被称为执行体。
- (2) 创建 `Thread` 子类的实例，即创建了线程对象。
- (3) 调用线程对象的 `start()` 方法来启动该线程。

```
public class TestThread extends Thread{
    public void run() {
        System.out.println("Hello World");
    }
    public static void main(String[] args) {
        Thread mThread = new TestThread();
        mThread.start();
    }
}
```

2. 实现 `Runnable` 接口，并实现该接口的 `run()` 方法

以下是其主要步骤：

- (1) 自定义类并实现 `Runnable` 接口，实现 `run()` 方法。
- (2) 创建 `Thread` 子类的实例，用实现 `Runnable` 接口的对象作为参数实例化该 `Thread` 对象。
- (3) 调用 `Thread` 的 `start()` 方法来启动该线程。

```
public class TestRunnable implements Runnable {
    public void run() {
        System.out.println("Hello World");
    }
}
```

```
public class TestRunnable {
    public static void main(String[] args) {
        TestRunnable mTestRunnable = new TestRunnable();
        Thread mThread = new Thread(mTestRunnable);
        mThread.start();
    }
}
```

3. 实现 Callable 接口，重写 call()方法

Callable 接口实际是属于 Executor 框架中的功能类,Callable 接口与 Runnable 接口的功能类似,但提供了比 Runnable 更强大的功能,主要表现为以下 3 点:

- (1) Callable 可以在任务接受后提供一个返回值,Runnable 无法提供这个功能。
- (2) Callable 中的 call()方法可以抛出异常,而 Runnable 的 run()方法不能抛出异常。

(3) 运行 Callable 可以拿到一个 Future 对象,Future 对象表示异步计算的结果,它提供了检查计算是否完成的方法。由于线程属于异步计算模型,因此无法从别的线程中得到函数的返回值,在这种情况下就可以使用 Future 来监视目标线程调用 call()方法的情况。但调用 Future 的 get()方法以获取结果时,当前线程就会阻塞,直到 call()方法返回结果。

```
public class TestCallable {
    //创建线程类
    public static class MyTestCallable implements Callable {
        public String call() throws Exception {
            return "Hello World";
        }
    }

    public static void main(String[] args) {
        MyTestCallable mMyTestCallable= new MyTestCallable();
        ExecutorService mExecutorService = Executors.newSingleThreadPool();
        Future mfuture = mExecutorService.submit(mMyTestCallable);
        try {
            //等待线程结束,并返回结果
            System.out.println(mfuture.get());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在这 3 种方式中,一般推荐用实现 Runnable 接口的方式,其原因是,一个类应该在其需要加强或者修改时才会被继承。因此如果没有必要重写 Thread 类的其他方法,那么在这种情况下最好用实现 Runnable 接口的方式。

4.1.4 理解中断

当线程的 `run` 方法执行完毕，或者在方法中出现没有捕获的异常时，线程将终止。在 Java 早期版本中有一个 `stop` 方法，其他线程可以调用它终止线程，但是这个方法现在已经被弃用了。`interrupt` 方法可以用来请求中断线程。当一个线程调用 `interrupt` 方法时，线程的中断标识位将被置位（中断标识位为 `true`），线程会不时地检测这个中断标识位，以判断线程是否应该被中断。要想知道线程是否被置位，可以调用 `Thread.currentThread().isInterrupted()`，如下所示：

```
while(!Thread.currentThread().isInterrupted()){
    //do something
}
```

还可以调用 `Thread.interrupted()` 来对中断标识位进行复位。但是如果一个线程被阻塞，就无法检测中断状态。如果一个线程处于阻塞状态，线程在检查中断标识位时如果发现中断标识位为 `true`，则会在阻塞方法调用处抛出 `InterruptedException` 异常，并且在抛出异常前将线程的中断标识位复位，即重新设置为 `false`。需要注意的是被中断的线程不一定会终止，中断线程是为了引起线程的注意，被中断的线程可以决定如何去响应中断。如果是比较重要的线程则不会理会中断，而大部分情况则是线程会将中断作为一个终止的请求。另外，不要在底层代码里捕获 `InterruptedException` 异常后不做处理，如下所示：

```
void myTask() {
    ...
    try{
        sleep(50)
    }catch(InterruptedException e){

    }
    ...
}
```

如果你不知道抛出 `InterruptedException` 异常后如何处理，这里介绍两种合理的处理方式。

(1) 在 `catch` 子句中，调用 `Thread.currentThread().interrupt()` 来设置中断状态（因为抛出异常后中断标识位会复位），让外界通过判断 `Thread.currentThread().isInterrupted()` 来决定是否终止线程还是继续下去，应该这样做：

```
void myTask() {
    ...
    try{
        sleep(50)
    }catch(InterruptedException e){
        Thread.currentThread().interrupt();
    }
}
```

```
...
}
```

(2) 更好的做法就是，不使用 `try` 来捕获这样的异常，让方法直接抛出，这样调用者可以捕获这个异常，如下所示：

```
void myTask() throws InterruptedException {
    sleep(50)
}
```

4.1.5 安全地终止线程

前面我们讲到了中断，那么首先就用中断来终止线程，代码如下：

```
public class StopThread {
    public static void main(String[] args) throws InterruptedException {
        MoonRunner runnable = new MoonRunner();
        Thread thread = new Thread(runnable, "MoonThread");
        thread.start();
        TimeUnit.MILLISECONDS.sleep(10); //1
        thread.interrupt();
    }

    public static class MoonRunner implements Runnable {
        private long i;
        @Override
        public void run() {
            while (!Thread.currentThread().isInterrupted()) {
                i++;
                System.out.println("i=" + i);
            }
            System.out.println("stop");
        }
    }
}
```

在上面代码注释 1 处调用了 `sleep` 方法使得 `main` 线程睡眠 10ms，这是为了留给 `MoonThread` 线程时间来感知中断从而结束。除了中断，还可以采用 `boolean` 变量来控制是否需要停止线程，代码如下：

```
public class StopThread {
    public static void main(String[] args) throws InterruptedException {
        MoonRunner runnable = new MoonRunner();
        Thread thread = new Thread(runnable, "MoonThread");
```

```

        thread.start();
        TimeUnit.MILLISECONDS.sleep(10);
        runnable.cancel();
    }

    public static class MoonRunner implements Runnable {
        private long i;
        private volatile boolean on = true;//1

        @Override
        public void run() {
            while (on) {
                i++;
                System.out.println("i=" + i);
            }
            System.out.println("stop");
        }
        public void cancel() {
            on = false;
        }
    }
}

```

在上面代码注释 1 处用到了 `volatile` 关键字，这会在本章以后的小节中介绍。在这里我们只需要知道因为涉及多个线程对这个变量的访问，所以当我们在设置 `volatile boolean on` 的时候，当有其他线程改变其值时，所有的线程都会感知到它的变化。这两种方式输出的结果类似，如图 4-4 所示。这里只打出了部分信息，`i` 的值由 1 一直打印到 1390（输出的结果是变化的）。最后打印 `stop`，说明执行到了 `run` 方法的末尾，线程即将终止。

```

i=1381
i=1382
i=1383
i=1384
i=1385
i=1386
i=1387
i=1388
i=1389
i=1390
stop

```

图 4-4 打印结果

4.2 同步

同步一直是 Java 多线程的难点，而且在我们做 Android 开发时也很少会应用到它，但这并不是我们不熟悉同步的理由。本节我们就来学习一下同步的基础知识。在多线程应用中，两个或者两个以上的线程需要共享对同一个数据的存取。如果两个线程存取相同的对象，并且每一个线程都调用了修改该对象的方法，这种情况通常被称为竞争条件。竞争条件最容易理解的例子如下：比如车站售卖火车票，火车票是一定的，但卖火车票的窗口到处都有，每个窗口就相当于一个线程。这么多的线程共用所有的火车票资源，如果不使用同步是无法保证其原子性的。在一个时间点上，两个线程同时使用火车票资源，那其取出的火车票是一样的（座位号一样），这样就会给乘客造成麻烦。解决方法如下：当一个线程要使用火车票这个资源时，我们就交给它一把锁，等它把事情做完后再把锁给另一个要用这个资源的线程。这样就不会出现上述情况了。

4.2.1 重入锁与条件对象

`synchronized` 关键字自动提供了锁以及相关的条件。大多数需要显式锁的情况使用 `synchronized` 非常方便，但是等我们了解了重入锁和条件对象时，能更好地理解 `synchronized` 关键字。重入锁 `ReentrantLock` 是 Java SE 5.0 引入的，就是支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁。用 `ReentrantLock` 保护代码块的结构如下所示：

```
Lock mLock=new ReentrantLock();
mLock.lock();
try{
    ...
}
finally{
    mLock.unlock();
}
```

这一结构确保任何时刻只有一个线程进入临界区，临界区就是在同一时刻只能有一个任务访问的代码区。一旦一个线程封锁了锁对象，其他任何线程都无法进入 `Lock` 语句。把解锁的操作放在 `finally` 中是十分必要的。如果在临界区发生了异常，锁是必须要释放的，否则其他线程将会永远被阻塞。进入临界区时，却发现在某一个条件满足之后，它才能执行。这时可以使用一个条件对象来管理那些已经获得了一个锁但是却不能做有用工作的线程，条件对象又被称作条件变量。通过下面的例子来说明为何需要条件对象。假设一个场景需要用支付宝转账。我们首先写了支付宝的类，它的构造方法需要传入支付宝账户的数量和每个账户的账户金额。

```
public class Alipay {
    private double[] accounts;
    private Lock alipaylock;
    public Alipay(int n,double money){
        accounts=new double[n];
```

```

        alipaylock=new ReentrantLock();
        for (int i=0;i<accounts.length;i++){
            accounts[i]=money;
        }
    }
}

```

接下来我们要转账，写一个转账的方法，from 是转账方，to 是接收方，amount 是转账金额，如下所示：

```

public void transfer(int from,int to,int amount){
    alipaylock.lock();
    try{
        while (accounts[from]<amount){
            //wait
        }
    }finally {
        alipaylock.unlock();
    }
}

```

结果我们发现转账方余额不足；如果有其他线程给这个转账方再转足够的钱，就可以转账成功了。但是这个线程已经获取了锁，它具有排他性，别的线程无法获取锁来进行存款操作，这就是我们需要引入条件对象的原因。一个锁对象拥有多个相关的条件对象，可以用 `newCondition` 方法获得一个条件对象，我们得到条件对象后调用 `await` 方法，当前线程就被阻塞了并放弃了锁。整理以上代码，加入条件对象，代码如下所示：

```

public class Alipay {
    private double[] accounts;
    private Lock alipaylock;
    private Condition condition;
    public Alipay(int n,double money){
        accounts=new double[n];
        alipaylock=new ReentrantLock();
        //得到条件对象
        condition=alipaylock.newCondition();
        for (int i=0;i<accounts.length;i++){
            accounts[i]=money;
        }
    }
    public void transfer(int from,int to,int amount) throws
        InterruptedException {
        alipaylock.lock();
        try{

```

```

        while (accounts[from]<amount){
            //阻塞当前线程，并放弃锁
            condition.await();
        }
    }finally {
        alipaylock.unlock();
    }
}
}

```

一旦一个线程调用 `await` 方法，它就会进入该条件的等待集并处于阻塞状态，直到另一个线程调用了同一个条件的 `signalAll` 方法时为止。当另一个线程转账给我们此前的转账方时，只要调用 `condition.signalAll()`，就会重新激活因为这一条件而等待的所有线程。代码如下所示：

```

public void transfer(int from,int to,int amount) throws InterruptedException {
    alipaylock.lock();
    try{
        while (accounts[from]<amount){
            //阻塞当前线程，并放弃锁
            condition.await();
        }
        //转账的操作
        accounts[from]=accounts[from]-amount;
        accounts[to]=accounts[to]+amount;
        condition.signalAll();
    }finally {
        alipaylock.unlock();
    }
}
}

```

当调用 `signalAll` 方法时并不是立即激活一个等待线程，它仅仅解除了等待线程的阻塞，以便这些线程能够在当前线程退出同步方法后，通过竞争实现对对象的访问。还有一个方法是 `signal`，它则是随机解除某个线程的阻塞。如果该线程仍然不能运行，则再次被阻塞。如果没有其他线程再次调用 `signal`，那么系统就死锁了。

4.2.2 同步方法

`Lock` 和 `Condition` 接口为程序设计人员提供了高度的锁定控制，然而大多数情况下，并不需要那样的控制，并且可以使用一种嵌入到 Java 语言内部的机制。从 Java 1.0 版开始，Java 中的每一个对象都有一个内部锁。如果一个方法用 `synchronized` 关键字声明，那么对象的锁将保护整个方法。也就是说，要调用该方法，线程必须获得内部的对象锁。也就是如下代码

```
public synchronized void method(){
```



```
...
}
```

等价于：

```
Lock mLock=new ReentrantLock();
public void method(){
    mLock.lock();
    try{
        ...
    }finally{
        mLock.unlock();
    }
}
```

对于上面支付宝转账的例子，我们可以将 Alipay 类的 transfer 方法声明为 synchronized，而不是使用一个显式的锁。内部对象锁只有一个相关条件，wait 方法将一个线程添加到等待集中，notifyAll 或者 notify 方法解除等待线程的阻塞状态。也就是说 wait 相当于调用 condition.await()，notifyAll 等价于 condition.signalAll();

上面例子中的 transfer 方法也可以这样写：

```
public synchronized void transfer(int from,int to,int amount)throws
InterruptedException{
    while (accounts[from]<amount) {
        wait();
    }
    //转账的操作
    accounts[from]=accounts[from]-amount;
    accounts[to]=accounts[to]+amount;
    notifyAll();
}
```

可以看到使用 synchronized 关键字来编写代码要简洁很多。当然要理解这一代码，你必须了解每一个对象有一个内部锁，并且该锁有一个内部条件。由该锁来管理那些试图进入 synchronized 方法的线程，由该锁中的条件来管理那些调用 wait 的线程。

4.2.3 同步代码块

上面我们说过，每一个 Java 对象都有一个锁，线程可以调用同步方法来获得锁。还有另一种机制可以获得锁，那就是使用一个同步代码块，如下所示：

```
synchronized(obj){

}
```

其获得了 obj 的锁, obj 指的是一个对象。再来看看 Alipay 类, 我们用同步代码块进行改写。

```
public class Alipay {
    private double[] accounts;
    private Object lock=new Object();
    public Alipay(int n,double money){
        accounts=new double[n];
        for (int i=0;i<accounts.length;i++){
            accounts[i]=money;
        }
    }
    public void transfer(int from,int to,int amount){
        synchronized(lock){
            //转账的操作
            accounts[from]=accounts[from]-amount;
            accounts[to]=accounts[to]+amount;
        }
    }
}
```

在这里创建了一个名为 Lock 的 Object 类, 为的是使用 Object 类所持有的锁。同步代码块是非常脆弱的, 通常不推荐使用。一般实现同步最好用 `java.util.concurrent` 包下提供的类, 比如阻塞队列。如果同步方法适合你的程序, 那么请尽量使用同步方法, 这样可以减少编写代码的数量, 减少出错的概率。如果特别需要使用 Lock/Condition 结构提供的独有特性时, 才使用 Lock/Condition。

4.2.4 volatile

有时仅仅为了读写一个或者两个实例域就使用同步的话, 显得开销过大; 而 `volatile` 关键字为实例域的同步访问提供了免锁的机制。如果声明一个域为 `volatile`, 那么编译器和虚拟机就知道该域是可能被另一个线程并发更新的。再讲到 `volatile` 关键字之前, 我们需要了解一下内存模型的相关概念以及并发编程中的 3 个特性: 原子性、可见性和有序性。

1. Java 内存模型

Java 中的堆内存用来存储对象实例, 堆内存是被所有线程共享的运行时内存区域, 因此, 它存在内存可见性的问题。而局部变量、方法定义的参数则不会在线程之间共享, 它们不会有内存可见性问题, 也不受内存模型的影响。Java 内存模型定义了线程和主存之间的抽象关系: 线程之间的共享变量存储在主存中, 每个线程都有一个私有的本地内存, 本地内存中存储了该线程共享变量的副本。需要注意的是本地内存是 Java 内存模型的一个抽象概念, 其并不真实存在, 它涵盖了缓存、写缓冲区、寄存器等区域。Java 内存模型控制线程之间的通信,

它决定一个线程对主存共享变量的写入何时对另一个线程可见。Java 内存模型的抽象示意图如图 4-5 所示。

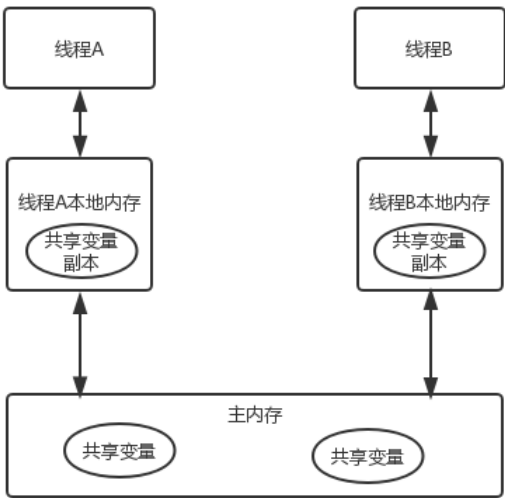


图 4-5 Java 内存模型的抽象示意图

线程 A 与线程 B 之间若要通信的话，必须要经历下面两个步骤：

(1) 线程 A 把线程 A 本地内存中更新过的共享变量刷新到主存中去。

(2) 线程 B 到主存中去读取线程 A 之前已更新过的共享变量。由此可见，如果我们执行下面的语句：

```
int i=3;
```

执行线程必须先在自己的工作线程中对变量 i 所在的缓存行进行赋值操作，然后再写入主存当中，而不是直接将数值 3 写入主存当中。

2. 原子性、可见性和有序性

那么 Java 语言本身对原子性、可见性以及有序性提供了哪些保证呢？首先我们要了解一下这 3 个特性。

(1) 原子性

对基本数据类型变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行完毕，要么就不执行。现在看一下下面的代码，如下所示：

```
x = 3;           //语句 1
y = x;           //语句 2
x++;             //语句 3
```

在上面 3 个语句中，只有语句 1 是原子性操作，其他两个语句都不是原子性操作。语句 2 虽说很短，但它包含了两个操作，它先读取 x 的值，再将 x 的值写入工作内存。读取 x 的值以

及将 `x` 的值写入工作内存这两个操作单拿出来都是原子性操作，但是合起来就不是原子性操作了。语句 3 包括 3 个操作：读取 `x` 的值、对 `x` 的值进行加 1、向工作内存写入新值。通过这 3 个语句我们得知，一个语句含有多个操作时，就不是原子性操作，只有简单地读取和赋值（将数字赋值给某个变量）才是原子性操作。`java.util.concurrent.atomic` 包中有很多类使用了很高效的机器级指令（而不是使用锁）来保证其他操作的原子性。例如 `AtomicInteger` 类提供了方法 `incrementAndGet` 和 `decrementAndGet`，它们分别以原子方式将一个整数自增和自减。可以安全地使用 `AtomicInteger` 类作为共享计数器而无须同步。另外这个包还包含 `AtomicBoolean`、`AtomicLong` 和 `AtomicReference` 这些原子类，这仅供开发并发工具的系统程序员使用，应用程序员不应该使用这些类。

（2）可见性

可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果，另一个线程马上就能看到。当一个共享变量被 `volatile` 修饰时，它会保证修改的值立即被更新到主存，所以对其他线程是可见的。当有其他线程需要读取该值时，其他线程会去主存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，并不会立即被写入主存，何时被写入主存也是不确定的。当其他线程去读取该值时，此时主存中可能还是原来的旧值，这样就无法保证可见性。

（3）有序性

Java 内存模型中允许编译器和处理器对指令进行重排序，虽然重排序过程不会影响到单线程执行的正确性，但是会影响到多线程并发执行的正确性。这时可以通过 `volatile` 来保证有序性，除了 `volatile`，也可以通过 `synchronized` 和 `Lock` 来保证有序性。我们知道，`synchronized` 和 `Lock` 保证每个时刻只有一个线程执行同步代码，这相当于是让线程顺序执行同步代码，从而保证了有序性。

3. `volatile` 关键字

当一个共享变量被 `volatile` 修饰之后，其就具备了两个含义，一个是线程修改了变量的值时，变量的新值对其他线程是立即可见的。换句话说，就是不同线程对这个变量进行操作时具有可见性。另一个含义是禁止使用指令重排序。

这里提到了重排序，那么什么是重排序呢？重排序通常是编译器或运行时环境为了优化程序性能而采取的对指令进行重新排序执行的一种手段。重排序分为两类：编译期重排序和运行期重排序，分别对应编译时和运行时环境。

下面我们来看一段代码，假设线程 1 先执行，线程 2 后执行，如下所示：

```
//线程 1
boolean stop = false;
while(!stop){
    //doSomething
}
```

```
//线程2
stop = true;
```

很多开发人员在中断线程时可能会采用这种方式。但是这段代码不一定会将线程中断。虽说无法中断线程这个情况出现的概率很小，但是一旦发生这种情况就会造成死循环。为何有可能无法中断线程？在前面我提到每个线程在运行时都有私有的工作内存，因此线程 1 在运行时会将 `stop` 变量的值复制一份放在私有的工作内存中。当线程 2 更改了 `stop` 变量的值之后，线程 2 突然需要去做其他的操作，这时就无法将更改的 `stop` 变量写入主存当中，这样线程 1 就不会知道线程 2 对 `stop` 变量进行了更改，因此线程 1 就会一直循环下去。当 `stop` 用 `volatile` 修饰之后，那么情况就变得不同了，当线程 2 进行修改时，会强制将修改的值立即写入主存，并且会导致线程 1 的工作内存中变量 `stop` 的缓存行无效，这样线程 1 再次读取变量 `stop` 的值时就会去主存读取。

volatile 不保证原子性

我们知道 `volatile` 保证了操作的可见性，下面我们来分析 `volatile` 是否能保证对变量的操作是原子性的。现在先阅读以下代码：

```
public class VolatileTest {
    public volatile int inc = 0;
    public void increase() {
        inc++;
    }

    public static void main(String[] args) {
        final VolatileTest test = new VolatileTest();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();

                };
            }.start();
        }
        //如果有子线程就让出资源，保证所有子线程都执行完
        while(Thread.activeCount()>2) {
            Thread.yield();
        }
        System.out.println(test.inc);
    }
}
```

这段代码每次运行，结果都不一致。在前面已经提到过，自增操作是不具备原子性的，它包括读取变量的原始值、进行加 1、写入工作内存。也就是说，自增操作的 3 个子操作可能会分割开执行。假如某个时刻变量 `inc` 的值为 9，线程 1 对变量进行自增操作，线程 1 先读取了变量 `inc` 的原始值，然后线程 1 被阻塞了。之后线程 2 对变量进行自增操作，线程 2 也去读取变量 `inc` 的原始值，然后进行加 1 操作，并把 10 写入工作内存，最后写入主存。随后线程 1 接着进行加 1 操作，因为线程 1 在此前已经读取了 `inc` 的值为 9，所以不会再去主存读取最新的数值，线程 1 对 `inc` 进行加 1 操作后 `inc` 的值为 10，然后将 10 写入工作内存，最后写入主存。两个线程分别对 `inc` 进行了一次自增操作后，`inc` 的值只增加了 1，因此自增操作不是原子性操作，`volatile` 也无法保证对变量的操作是原子性的。

volatile 保证有序性

`volatile` 关键字能禁止指令重排序，因此 `volatile` 能保证有序性。`volatile` 关键字禁止指令重排序有两个含义：一个是当程序执行到 `volatile` 变量的操作时，在其前面的操作已经全部执行完毕，并且结果会对后面的操作可见，在其后面的操作还没有进行；在进行指令优化时，在 `volatile` 变量之前的语句不能在 `volatile` 变量后面执行；同样，在 `volatile` 变量之后的语句也不能在 `volatile` 变量前面执行。

4. 正确使用 volatile 关键字

`synchronized` 关键字可防止多个线程同时执行一段代码，那么这就会很影响程序执行效率。而 `volatile` 关键字在某些情况下的性能要优于 `synchronized`。但是要注意 `volatile` 关键字是无法替代 `synchronized` 关键字的，因为 `volatile` 关键字无法保证操作的原子性。通常来说，使用 `volatile` 必须具备以下两个条件：

- (1) 对变量的写操作不会依赖于当前值。
- (2) 该变量没有包含在具有其他变量的不变式中。

第一个条件就是不能是自增、自减等操作，上文已经提到 `volatile` 不保证原子性。关于第二个条件，我们来举一个例子，它包含了一个不变式：下界总是小于或等于上界，代码如下所示：

```
public class NumberRange {
    private volatile int lower, upper;
    public int getLower() {
        return lower;
    }
    public int getUpper() {
        return upper;
    }
    public void setLower(int value) {
        if (value > upper)
            throw new IllegalArgumentException(...);
        lower = value;
    }
    public void setUpper(int value) {
```

```

        if (value < lower)
            throw new IllegalArgumentException(...);
        upper = value;
    }
}

```

这种方式将 `lower` 和 `upper` 字段定义为 `volatile` 类型不能够充分实现类的线程安全。如果当两个线程在同一时间使用不一致的值执行 `setLower` 和 `setUpper` 的话, 则会使范围处于不一致的状态。例如, 如果初始状态是(0, 5), 在同一时间内, 线程 A 调用 `setLower(4)` 并且线程 B 调用 `setUpper(3)`, 虽然这两个操作交叉存入的值是不符合条件的, 但是这两个线程都会通过用于保护不变式的检查, 使得最后的范围值是(4, 3)。这显然是不对的, 因此使用 `volatile` 无法实现 `setLower` 和 `setUpper` 操作的原子性。

使用 `volatile` 有很多种场景, 这里介绍其中的两种。

(1) 状态标志

```

volatile boolean shutdownRequested;
...
public void shutdown()
{
    shutdownRequested = true;
}
public void doWork() {
    while (!shutdownRequested) {
        ...
    }
}
}

```

如果在另一个线程中调用 `shutdown` 方法, 就需要执行某种同步来确保正确实现 `shutdownRequested` 变量的可见性。但是, 使用 `synchronized` 块编写循环要比使用 `volatile` 状态标志编写麻烦很多。在这里推荐使用 `volatile`, 状态标志 `shutdownRequested` 并不依赖于程序内的任何其他状态, 并且还能简化代码。因此, 此处适合使用 `volatile`。

(2) 双重检查模式 (DCL)

```

public class Singleton {
    private volatile static Singleton instance = null;
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(this) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
    }
}

```

```

    }
    return instance;
}
}

```

`getInstance` 方法中对 `Singleton` 进行了两次判空，第一次是为了不必要的同步，第二次是只有在 `Singleton` 等于 `null` 的情况下才创建实例。在这里用到了 `volatile` 关键字会或多或少地影响性能，但考虑到程序的正确性，牺牲这点性能还是值得的。`DCL` 的优点是资源利用率高，第一次执行 `getInstance` 方法时单例对象才被实例化，效率高。其缺点是第一次加载时反应稍慢一些，在高并发环境下也有一定的缺陷（虽然发生的概率很小）。

5. 小结

与锁相比，`volatile` 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性。如果严格遵循 `volatile` 的使用条件，即变量真正独立于其他变量和自己以前的值，在某些情况下可以使用 `volatile` 代替 `synchronized` 来简化代码。然而，使用 `volatile` 的代码往往比使用锁的代码更加容易出错。在前面的第 4 小节中介绍了可以使用 `volatile` 代替 `synchronized` 的最常见的两种用例，在其他情况下我们最好还是使用 `synchronized`。

4.3 阻塞队列

本节我们学习阻塞队列，学习它的目的就是为了更好地理解线程池。本节会介绍阻塞队列的定义、种类、实现原理以及应用。

4.3.1 阻塞队列简介

阻塞队列常用于生产者和消费者的场景，生产者是在队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

1. 常见阻塞场景

阻塞队列有两个常见的阻塞场景，它们分别是：

（1）当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列。

（2）当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。

支持以上两种阻塞场景的队列被称为阻塞队列。

2. `BlockingQueue` 的核心方法

放入数据：

- `offer(anObject)`: 表示如果可能的话，将 `anObject` 加到 `BlockingQueue` 里。即如果 `BlockingQueue`

可以容纳，则返回 `true`，否则返回 `false`。（本方法不阻塞当前执行方法的线程。）

- `offer(E o, long timeout, TimeUnit unit)`: 可以设定等待的时间。如果在指定的时间内还不能往队列中加入 `BlockingQueue`，则返回失败。
- `put(anObject)`: 将 `anObject` 加到 `BlockingQueue` 里。如果 `BlockingQueue` 没有空间，则调用此方法的线程被阻断，直到 `BlockingQueue` 里面有空间再继续。

获取数据:

- `poll(time)`: 取走 `BlockingQueue` 里排在首位的对象。若不能立即取出，则可以等 `time` 参数规定的时间。取不到时返回 `null`。
- `poll(long timeout, TimeUnit unit)`: 从 `BlockingQueue` 中取出一个队首的对象。如果在指定时间内，队列一旦有数据可取，则立即返回队列中的数据；否则直到时间超时还没有数据可取，返回失败。
- `take()`: 取走 `BlockingQueue` 里排在首位的对象。若 `BlockingQueue` 为空，则阻断进入等待状态，直到 `BlockingQueue` 有新的数据被加入。
- `drainTo()`: 一次性从 `BlockingQueue` 获取所有可用的数据对象（还可以指定获取数据的个数）。通过该方法，可以提升获取数据的效率；无须多次分批加锁或释放锁。

4.3.2 Java 中的阻塞队列

在 Java 中提供了 7 个阻塞队列，它们分别如下所示。

- `ArrayBlockingQueue`: 由数组结构组成的有界阻塞队列。
- `LinkedBlockingQueue`: 由链表结构组成的有界阻塞队列。
- `PriorityBlockingQueue`: 支持优先级排序的无界阻塞队列。
- `DelayQueue`: 使用优先级队列实现的无界阻塞队列。
- `SynchronousQueue`: 不存储元素的阻塞队列。
- `LinkedTransferQueue`: 由链表结构组成的无界阻塞队列。
- `LinkedBlockingDeque`: 由链表结构组成的双向阻塞队列。

下面分别介绍这些阻塞队列。

1. `ArrayBlockingQueue`

它是用数组实现的有界阻塞队列，并按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证线程公平地访问队列。公平访问队列就是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列。即先阻塞的生产者线程，可以先往队列里插入元素；先阻塞的消费者线程，可以先从队列里获取元素。通常情况下为了保证公平性会降低吞吐量。我们可以使用以下代码创建一个公平的阻塞队列，如下所示：

```
ArrayBlockingQueue fairQueue = new ArrayBlockingQueue(2000,true);
```

2. LinkedBlockingQueue

它是基于链表的阻塞队列，同 `ArrayListBlockingQueue` 类似，此队列按照先进先出（FIFO）的原则对元素进行排序，其内部也维持着一个数据缓冲队列（该队列由一个链表构成）。当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到缓存容量的最大值时（`LinkedBlockingQueue` 可以通过构造方法指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒。反之，对于消费者这端的处理也基于同样的原理。而 `LinkedBlockingQueue` 之所以能够高效地处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步。这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。作为开发者，我们需要注意的是，如果构造一个 `LinkedBlockingQueue` 对象，而没有指定其容量大小，`LinkedBlockingQueue` 会默认一个类似无限大小的容量（`Integer.MAX_VALUE`）。这样一来，如果生产者的速度一旦大于消费者的速度，也许还没有等到队列满阻塞产生，系统内存就有可能已被消耗殆尽了。`ArrayBlockingQueue` 和 `LinkedBlockingQueue` 是两个最普通也是最常用的阻塞队列。一般情况下，在处理多线程间的生产者-消费者问题时，使用这两个类足已。

3. PriorityBlockingQueue

它是一个支持优先级的无界队列。默认情况下元素采取自然顺序升序排列。这里可以自定义实现 `compareTo()` 方法来指定元素进行排序规则；或者初始化 `PriorityBlockingQueue` 时，指定构造参数 `Comparator` 来对元素进行排序。但其不能保证同优先级元素的顺序。

4. DelayQueue

它是一个支持延时获取元素的无界阻塞队列。队列使用 `PriorityQueue` 来实现。队列中的元素必须实现 `Delayed` 接口。创建元素时，可以指定元素到期的时间，只有在元素到期时才能从队列中取走。

5. SynchronousQueue

它是一个不存储元素的阻塞队列。每个插入操作必须等待另一个线程的移除操作，同样任何一个移除操作都等待另一个线程的插入操作。因此此队列内部其实没有任何一个元素，或者说容量是 0，严格来说它并不是一种容器。由于队列没有容量，因此不能调用 `peek` 操作（返回队列的头元素）。

6. LinkedTransferQueue

它是一个由链表结构组成的无界阻塞 `TransferQueue` 队列。`LinkedTransferQueue` 实现了一个重要的接口 `TransferQueue`。该接口含有 5 个方法，其中有 3 个重要的方法，它们分别如下所示。

（1）`transfer(E e)`：若当前存在一个正在等待获取的消费者线程，则立刻将元素传递给消费者；如果没有消费者在等待接收数据，就会将元素插入到队列尾部，并且等待进入阻塞状态，直到有消费者线程取走该元素。

(2) `tryTransfer(E e)`: 若当前存在一个正在等待获取的消费者线程, 则立刻将元素传递给消费者; 若不存在, 则返回 `false`, 并且不进入队列, 这是一个不阻塞的操作。与 `transfer` 方法不同的是, `tryTransfer` 方法无论消费者是否接收, 其都会立即返回; 而 `transfer` 方法则是消费者接收了才返回。

(3) `tryTransfer(E e, long timeout, TimeUnit unit)`: 若当前存在一个正在等待获取的消费者线程, 则立刻将元素传递给消费者; 若不存在则将元素插入到队列尾部, 并且等待消费者线程取走该元素。若在指定的超时时间内元素未被消费者线程获取, 则返回 `false`; 若在指定的超时时间内其被消费者线程获取, 则返回 `true`。

7. LinkedBlockingDeque

它是一个由链表结构组成的双向阻塞队列。双向队列可以从队列的两端插入和移出元素, 因此在多线程同时入队时, 也就减少了一半的竞争。由于是双向的, 因此 `LinkedBlockingDeque` 多了 `addFirst`、`addLast`、`offerFirst`、`offerLast`、`peekFirst`、`peekLast` 等方法。其中, 以 `First` 单词结尾的方法, 表示插入、获取或移除双端队列的第一个元素; 以 `Last` 单词结尾的方法, 表示插入、获取或移除双端队列的最后一个元素。

4.3.3 阻塞队列的实现原理

以 `ArrayBlockingQueue` 为例, 我们先来看看代码, 如下所示:

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    private static final long serialVersionUID = -817911632652898426L;
    final Object[] items;
    int takeIndex;
    int putIndex;
    int count;
    final ReentrantLock lock;
    private final Condition notEmpty;
    private final Condition notFull;
    ...
}
```

从上面的代码可以看出 `ArrayBlockingQueue` 是维护一个 `Object` 类型的数组, `takeIndex` 和 `putIndex` 分别表示队首元素和队尾元素的下标, `count` 表示队列中元素的个数, `lock` 则是一个可重入锁, `notEmpty` 和 `notFull` 是等待条件。接下来我们看看关键方法 `put`, 代码如下所示:

```
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
```

```

    try {
        while (count == items.length)
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}

```

从 `put` 方法的实现可以看出，它先获取了锁，并且获取的是可中断锁，然后判断当前元素个数是否等于数组的长度，如果相等，则调用 `notFull.await()` 进行等待。当此线程被其他线程唤醒时，通过 `enqueue(e)` 方法插入元素，最后解锁。接下来看看 `enqueue(e)` 方法，如下所示：

```

private void enqueue(E x) {
    final Object[] items = this.items;
    items[putIndex] = x;
    if (++putIndex == items.length)
        putIndex = 0;
    count++;
    notEmpty.signal();
}

```

插入成功后，通过 `notEmpty` 唤醒正在等待取元素的线程。再来看看 `take` 方法。

```

public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}

```

跟 `put` 方法实现类似，`put` 方法等待的是 `notFull` 信号，而 `take` 方法等待的是 `notEmpty` 信号。在 `take` 方法中，如果可以取元素，则通过 `dequeue` 方法取得元素。下面是 `dequeue` 方法的实现。

```

private E dequeue() {
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex];
    items[takeIndex] = null;
    if (++takeIndex == items.length) takeIndex = 0;
}

```

```

        count--;
        if (itrs != null)
            itrs.elementDequeued();
        notFull.signal();
        return x;
    }

```

跟 `enqueue` 方法类似，在获取元素后，通过 `notFull` 的 `signal` 方法来唤醒正在等待插入元素的线程。

4.3.4 阻塞队列的使用场景

除了线程池的实现使用阻塞队列外，我们还可以在生产者-消费者模式中使用阻塞队列：首先使用 `Object.wait()`、`Object.notify()` 和非阻塞队列实现生产者-消费者模式，代码如下所示：

```

public class Test {
    private int queueSize = 10;
    private PriorityQueue<Integer> queue = new PriorityQueue<Integer>
        (queueSize);
    public static void main(String[] args) {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();
        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{
        @Override
        public void run() {
            while(true){
                synchronized (queue) {
                    while(queue.size() == 0){
                        try {
                            System.out.println("队列空，等待数据");
                            queue.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                            queue.notify();
                        }
                    }
                }
                //每次移走队首元素
                queue.poll();
                queue.notify();
            }
        }
    }
}

```

```

        }
    }
}

class Producer extends Thread{
    @Override
    public void run() {
        while(true){
            synchronized (queue) {
                while(queue.size() == queueSize){
                    try {
                        System.out.println("队列满，等待有空余空间");
                        queue.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                        queue.notify();
                    }
                }
                //每次插入一个元素
                queue.offer(1);
                queue.notify();
            }
        }
    }
}
}
}
}

```

下面是使用阻塞队列实现的生产者-消费者模式。

```

public class Test {
    private int queueSize = 10;
    private ArrayBlockingQueue<Integer> queue =
        new ArrayBlockingQueue<Integer>(queueSize);
    public static void main(String[] args) {
        Test test = new Test();
        Producer producer = test.new Producer();
        Consumer consumer = test.new Consumer();
        producer.start();
        consumer.start();
    }

    class Consumer extends Thread{
        @Override

```

```

        public void run() {
            while(true){
                try {
                    queue.take();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    class Producer extends Thread{
        @Override
        public void run() {
            while(true){
                try {
                    queue.put(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

很显然，使用阻塞队列实现无须单独考虑同步和线程间通信的问题，其实现起来很简单。

4.4 线程池

在编程中经常会使用线程来异步处理任务，但是每个线程的创建和销毁都需要一定的开销。如果每次执行一个任务都需要开一个新线程去执行，则这些线程的创建和销毁将消耗大量的资源；并且线程都是“各自为政”的，很难对其进行控制，更何况有一堆的线程在执行。这时就需要线程池来对线程进行管理。在 Java 1.5 中提供了 `Executor` 框架用于把任务的提交和执行解耦，任务的提交交给 `Runnable` 或者 `Callable`，而 `Executor` 框架用来处理任务。`Executor` 框架中最核心的成员就是 `ThreadPoolExecutor`，它是线程池的核心实现类。本节就来着重讲解 `ThreadPoolExecutor`。

4.4.1 ThreadPoolExecutor

可以通过 `ThreadPoolExecutor` 来创建一个线程池，`ThreadPoolExecutor` 类一共有 4 个构造方

法。其中，拥有最多参数的构造方法如下所示：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    ...
}
```

这些参数的作用如下所示。

- **corePoolSize**: 核心线程数。默认情况下线程池是空的，只有任务提交时才会创建线程。如果当前运行的线程数少于 **corePoolSize**，则创建新线程来处理任务；如果等于或者多于 **corePoolSize**，则不再创建。如果调用线程池的 **prestartAllcoreThread** 方法，线程池会提前创建并启动所有的核心线程来等待任务。
- **maximumPoolSize**: 线程池允许创建的最大线程数。如果任务队列满了并且线程数小于 **maximumPoolSize** 时，则线程池仍旧会创建新的线程来处理任务。
- **keepAliveTime**: 非核心线程闲置的超时时间。超过这个时间则回收。如果任务很多，并且每个任务的执行事件很短，则可以调大 **keepAliveTime** 来提高线程的利用率。另外，如果设置 **allowCoreThreadTimeOut** 属性为 **true** 时，**keepAliveTime** 也会应用到核心线程上，
- **TimeUnit**: **keepAliveTime** 参数的时间单位。可选的单位有天（**DAYS**）、小时（**HOURS**）、分钟（**MINUTES**）、秒（**SECONDS**）、毫秒（**MILLISECONDS**）等。
- **workQueue**: 任务队列。如果当前线程数大于 **corePoolSize**，则将任务添加到此任务队列中。该任务队列是 **BlockingQueue** 类型的，也就是阻塞队列。这在前面已经介绍过了，这里就不赘述了。
- **ThreadFactory**: 线程工厂。可以用线程工厂给每个创建出来的线程设置名字。一般情况下无须设置该参数。
- **RejectedExecutionHandler**: 饱和策略。这是当任务队列和线程池都满了时所采取的应对策略，默认是 **AbordPolicy**，表示无法处理新任务，并抛出 **RejectedExecutionException** 异常。此外还有 3 种策略，它们分别如下。
 - (1) **CallerRunsPolicy**: 用调用者所在的线程来处理任务。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。
 - (2) **DiscardPolicy**: 不能执行的任务，并将该任务删除。
 - (3) **DiscardOldestPolicy**: 丢弃队列最近的任务，并执行当前的任务。

4.4.2 线程池的处理流程和原理

当向线程池提交任务时，线程池是如何处理的呢？本节就介绍线程池的原理。当提交一个新的任务到线程池时，线程池的处理流程如图 4-6 所示。

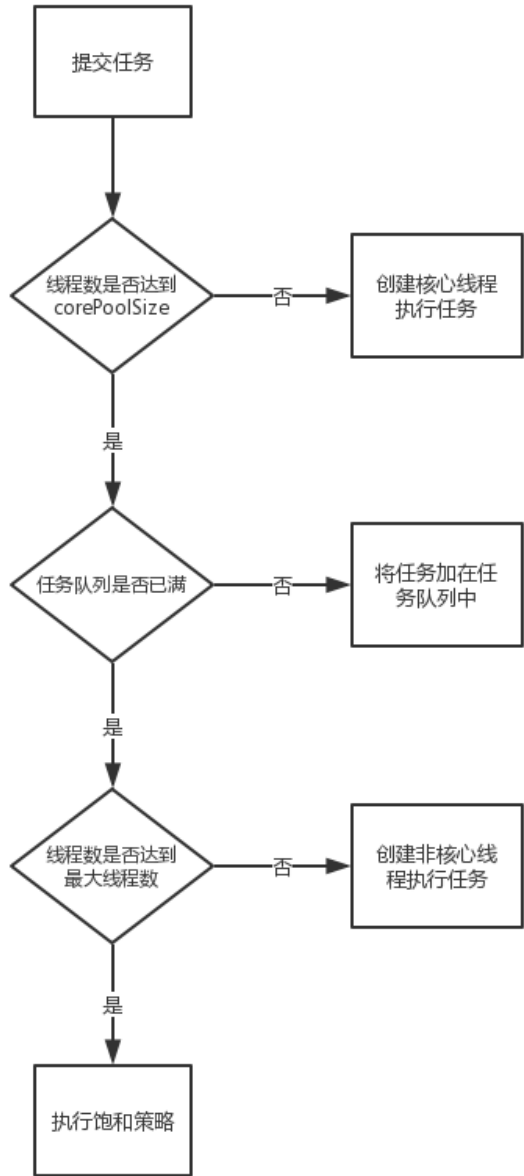


图 4-6 线程池的处理流程

从图 4-6 可以得知线程的处理流程主要分为 3 个步骤，如下所示。

(1) 提交任务后，线程池先判断线程数是否达到了核心线程数（corePoolSize）。如果未达到核心线程数，则创建核心线程处理任务；否则，就执行下一步操作。

(2) 接着线程池判断任务队列是否满了。如果没满，则将任务添加到任务队列中；否则，

就执行下一步操作。

(3) 接着因为任务队列满了，线程池就判断线程数是否达到了最大线程数。如果未达到，则创建非核心线程处理任务；否则，就执行饱和策略，默认会抛出 `RejectedExecutionException` 异常。

上面介绍了线程池的处理流程，但还不是很直观。下面结合图 4-7，我们就能更好地了解线程池的原理了。

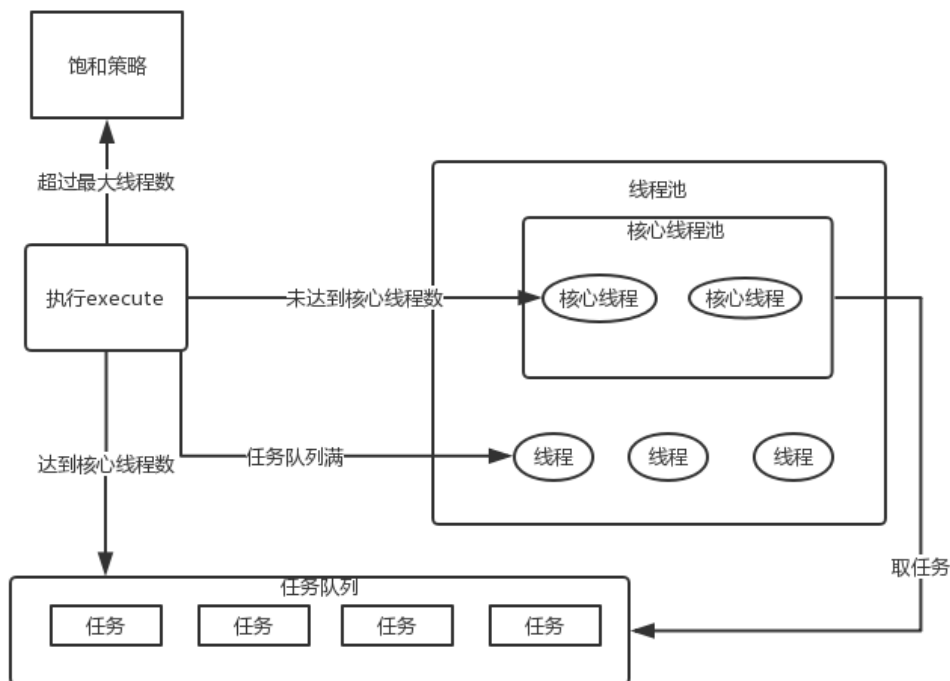


图 4-7 线程池执行示意图

从图 4-7 中可以看到，如果我们执行 `ThreadPoolExecutor` 的 `execute` 方法，会遇到各种情况：

- (1) 如果线程池中的线程数未达到核心线程数，则创建核心线程处理任务。
- (2) 如果线程数大于或者等于核心线程数，则将任务加入任务队列，线程池中的空闲线程会不断地从任务队列中取出任务进行处理。
- (3) 如果任务队列满了，并且线程数没有达到最大线程数，则创建非核心线程去处理任务。
- (4) 如果线程数超过了最大线程数，则执行饱和策略。

4.4.3 线程池的种类

通过直接或者间接地配置 `ThreadPoolExecutor` 的参数可以创建不同类型的 `ThreadPoolExecutor`，其中有 4 种线程池比较常用，它们分别是 `FixedThreadPool`、`CachedThreadPool`、`SingleThreadExecutor` 和 `ScheduledThreadPool`。下面分别介绍这 4 种线程池。

1. FixedThreadPool

FixedThreadPool 是可重用固定线程数的线程池。在 Executors 类中提供了创建 FixedThreadPool 的方法，如下所示：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

FixedThreadPool 的 corePoolSize 和 maximumPoolSize 都设置为创建 FixedThreadPool 指定的参数 nThreads，也就意味着 FixedThreadPool 只有核心线程，并且数量是固定的，没有非核心线程。keepAliveTime 设置为 0L 意味着多余的线程会被立即终止。因为不会产生多余的线程，所以 keepAliveTime 是无效的参数。另外，任务队列采用了无界的阻塞队列 LinkedBlockingQueue。FixedThreadPool 的 execute 方法的执行示意图如图 4-8 所示。

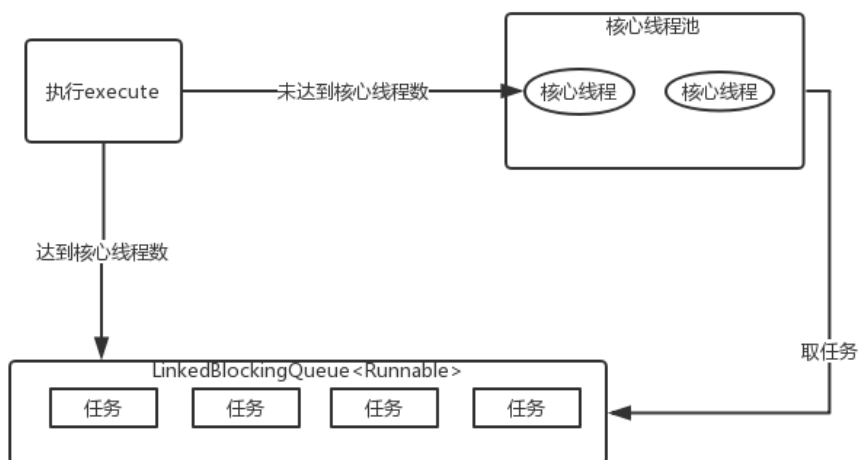


图 4-8 FixedThreadPool 的执行示意图

从图 4-8 中可以看出，当执行 execute 方法时，如果当前运行的线程未达到 corePoolSize（核心线程数）时就创建核心线程来处理任务，如果达到了核心线程数则将任务添加到 LinkedBlockingQueue 中。FixedThreadPool 就是一个有固定数量核心线程的线程池，并且这些核心线程不会被回收。当线程数超过 corePoolSize 时，就将任务存储在任务队列中；当线程池有空闲线程时，则从任务队列中去取任务执行。

2. CachedThreadPool

CachedThreadPool 是一个根据需要创建线程的线程池，创建 CachedThreadPool 的代码如下所示：

```
public static ExecutorService newCachedThreadPool() {
```

```

return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                              60L, TimeUnit.SECONDS,
                              new SynchronousQueue<Runnable>());
}

```

CachedThreadPool 的 corePoolSize 为 0，maximumPoolSize 设置为 Integer.MAX_VALUE，这意味着 CachedThreadPool 没有核心线程，非核心线程是无界的。keepAliveTime 设置为 60L，则空闲线程等待新任务的最长时间为 60s。在此用了阻塞队列 SynchronousQueue，它是一个不存储元素的阻塞队列，每个插入操作必须等待另一个线程的移除操作，同样任何一个移除操作都等待另一个线程的插入操作。CachedThreadPool 的 execute 方法的执行示意图如图 4-9 所示。

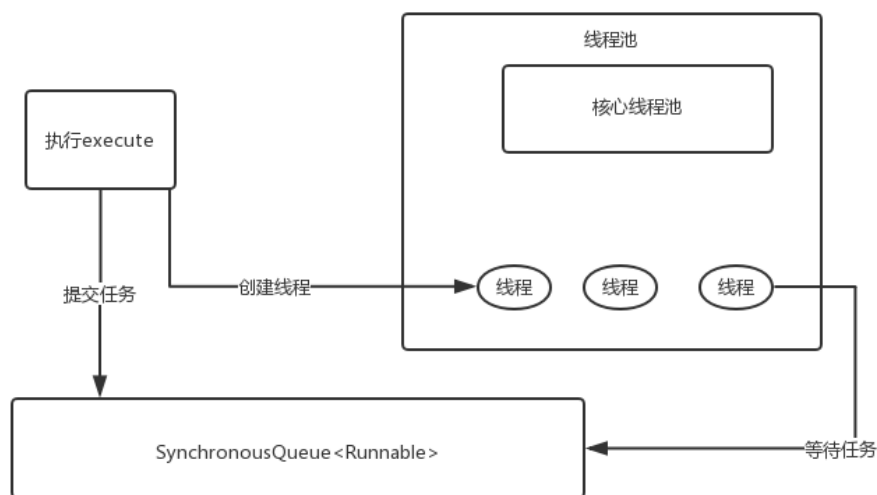


图 4-9 CachedThreadPool 的执行示意图

当执行 execute 方法时，首先会执行 SynchronousQueue 的 offer 方法来提交任务，并且查询线程池中是否有空闲的线程执行 SynchronousQueue 的 poll 方法来移除任务。如果有则配对成功，将任务交给这个空闲的线程处理；如果没有则配对失败，创建新的线程去处理任务。当线程池中的线程空闲时，它会执行 SynchronousQueue 的 poll 方法，等待 SynchronousQueue 中新提交的任务。如果超过 60s 没有新任务提交到 SynchronousQueue，则这个空闲线程将终止。因为 maximumPoolSize 是无界的，所以如果提交的任务大于线程池中线程处理任务的速度就会不断地创建新线程。另外，每次提交任务都会立即有线程去处理。所以，CachedThreadPool 比较适于大量的需要立即处理并且耗时较少的任务。

3. SingleThreadExecutor

SingleThreadExecutor 是使用单个工作线程的线程池，其创建源码如下所示：

```

public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,

```

```

        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());
    }

```

corePoolSize 和 maximumPoolSize 都为 1，意味着 SingleThreadExecutor 只有一个核心线程，其他的参数都和 FixedThreadPool 一样，这里就不赘述了。SingleThreadExecutor 的 execute 方法的执行示意图如图 4-10 所示。

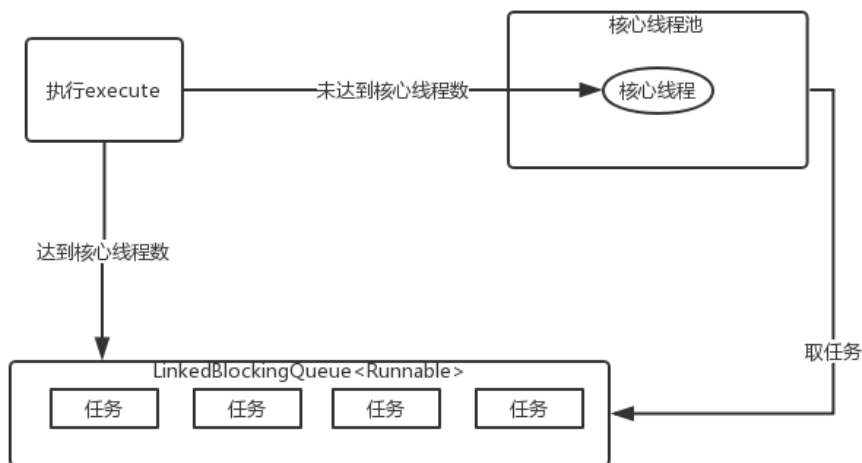


图 4-10 SingleThreadExecutor 的执行示意图

当执行 execute 方法时，如果当前运行的线程数未达到核心线程数，也就是当前没有运行的线程，则创建一个新线程来处理任务。如果当前有运行的线程，则将任务添加到阻塞队列 LinkedBlockingQueue 中。因此，SingleThreadExecutor 能确保所有的任务在一个线程中按照顺序逐一执行。

4. ScheduledThreadPool

ScheduledThreadPool 是一个能实现定时和周期性任务的线程池，它的创建源码如下所示：

```

public static ScheduledExecutorService newScheduledThreadPool(int
    corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

```

这里创建了 ScheduledThreadPoolExecutor，ScheduledThreadPoolExecutor 继承自 ThreadPoolExecutor，它主要用于给定延时之后的运行任务或者定期处理任务。ScheduledThreadPoolExecutor 的构造方法如下所示：

```

public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE,
        DEFAULT_KEEPLIVE_MILLIS, MILLISECONDS,

```

```
new DelayedWorkQueue();
}
```

从上面的代码可以看出，ScheduledThreadPoolExecutor 的构造方法最终调用的是 ThreadPoolExecutor 的构造方法。corePoolSize 是传进来的固定数值，maximumPoolSize 的值是 Integer.MAX_VALUE。因为采用的 DelayedWorkQueue 是无界的，所以 maximumPoolSize 这个参数是无效的。ScheduledThreadPoolExecutor 的 execute 方法的执行示意图如图 4-11 所示。

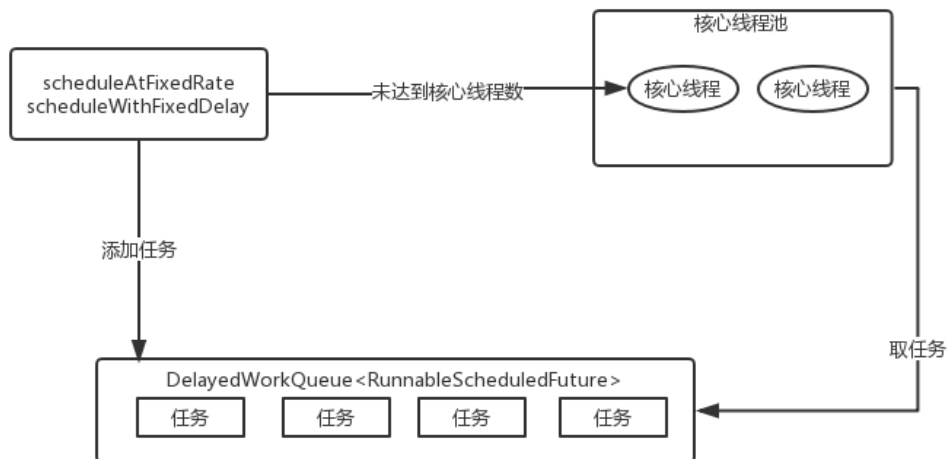


图 4-11 ScheduledThreadPoolExecutor 的执行示意图

当执行 ScheduledThreadPoolExecutor 的 scheduleAtFixedRate 或者 scheduleWithFixedDelay 方法时，会向 DelayedWorkQueue 添加一个实现 RunnableScheduledFuture 接口的 ScheduledFutureTask（任务的包装类），并会检查运行的线程是否达到 corePoolSize。如果没有则新建线程并启动它，但并不是立即去执行任务，而是去 DelayedWorkQueue 中取 ScheduledFutureTask，然后去执行任务。如果运行的线程达到了 corePoolSize 时，则将任务添加到 DelayedWorkQueue 中。DelayedWorkQueue 会将任务进行排序，先要执行的任务放在队列的前面。其跟此前介绍的线程池不同的是，当执行完任务后，会将 ScheduledFutureTask 中的 time 变量改为下次要执行的时间并放回到 DelayedWorkQueue 中。

4.5 AsyncTask 的原理

当我们通过线程去执行耗时的任务，并且在操作完之后可能还有更新 UI 时，通常还会用到 Handler 来更新 UI 线程。虽然其实现起来简单，但是如果有多多个任务同时执行时则会显得代码臃肿。Android 提供了 AsyncTask，它使得异步任务实现起来更加简单，代码更简洁。下面首先来看 AsyncTask 的定义，如下所示：

```
public abstract class AsyncTask<Params, Progress, Result> {
    ...
}
```

AsyncTask 是一个抽象的泛型类，它有 3 个泛型参数，分别为 Params、Progress 和 Result，其中 Params 为参数类型，Progress 为后台任务执行进度的类型，Result 为返回结果的类型。如果不需要某个参数，可以将其设置为 Void 类型。AsyncTask 中有 4 个核心方法，如下所示。

(1) onPreExecute(): 在主线程中执行。一般在任务执行前做准备工作，比如对 UI 做一些标记。

(2) doInBackground(Params... params): 在线程池中执行。在 onPreExecute 方法执行后运行，用来执行较为耗时的操作。在执行过程中可以调用 publishProgress(Progress... values)来更新进度信息。

(3) onProgressUpdate(Progress... values): 在主线程中执行。当调用 publishProgress(Progress... values)时，此方法会将进度更新到 UI 组件上。

(4) onPostExecute(Result result): 在主线程中执行。当后台任务执行完成后，它会被执行。doInBackground 方法得到的结果就是返回的 result 的值。此方法一般做任务执行后的收尾工作，比如更新 UI 和数据。

AsyncTask 源码分析

在了解了 AsyncTask 的基本原理后，接下来分析一下 AsyncTask 的源码。AsyncTask 在 3.0 版本之前和 3.0 及以后版本有着较大的改动，因此我们有必要知道主要的改动点在哪里。

1. Android 3.0 版本之前的 AsyncTask

下面是 Android 2.3.7 版本的 AsyncTask 的部分源码。

```
public abstract class AsyncTask<Params, Progress, Result> {
    private static final String LOG_TAG = "AsyncTask";
    private static final int CORE_POOL_SIZE = 5;
    private static final int MAXIMUM_POOL_SIZE = 128;
    private static final int KEEP_ALIVE = 1;
    private static final BlockingQueue<Runnable> sWorkQueue =
        new LinkedBlockingQueue<Runnable>(10);
    private static final ThreadFactory sThreadFactory = new ThreadFactory() {
        private final AtomicInteger mCount = new AtomicInteger(1);
        public Thread newThread(Runnable r) {
            return new Thread(r, "AsyncTask #" + mCount.getAndIncrement());
        }
    };
    private static final ThreadPoolExecutor sExecutor =
        new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE,
            TimeUnit.SECONDS, sWorkQueue, sThreadFactory);

    ...
}
```

在这里又看到了 `ThreadPoolExecutor`，它的核心线程数是 5 个，线程池允许创建的最大线程数为 128，非核心线程空闲等待新任务的最长时间为 1s。采用的阻塞队列是 `LinkedBlockingQueue`，它的容量为 10。3.0 版本之前的 `AsyncTask` 有一个缺点，就是线程池最大的线程数为 128，加上阻塞队列的 10 个任务，所以 `AsyncTask` 最多能同时容纳 138 个任务，当提交第 139 个任务时就会执行饱和策略，默认抛出 `RejectedExecutionException` 异常。

2. Android 7.0 版本的 AsyncTask

在这里采用 Android 7.0 版本的 `AsyncTask` 作为例子。首先来看 `AsyncTask` 的构造方法，代码如下所示：

```
public AsyncTask() {
    mWorker = new WorkerRunnable<Params, Result>() { //1
        public Result call() throws Exception {
            mTaskInvoked.set(true);
            Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
            Result result = doInBackground(mParams);
            Binder.flushPendingCommands();
            return postResult(result);
        }
    };

    mFuture = new FutureTask<Result>(mWorker) { //2
        @Override
        protected void done() {
            try {
                postResultIfNotInvoked(get());
            } catch (InterruptedException e) {
                android.util.Log.w(LOG_TAG, e);
            } catch (ExecutionException e) {
                throw new RuntimeException("An error occurred while executing doInBackground()",
                    e.getCause());
            } catch (CancellationException e) {
                postResultIfNotInvoked(null);
            }
        }
    };
}
```

从上面代码注释 1 处看这个 `WorkerRunnable` 实现了 `Callable` 接口，并实现了它的 `call` 方法，在 `call` 方法中调用了 `doInBackground(mParams)` 来处理任务并得到结果，并最终调用 `postResult` 将结果投递出去。注释 2 处的 `FutureTask` 是一个可管理的异步任务，它实现了 `Runnable` 和 `Futruce`

这两个接口。因此，它可以包装 `Runnable` 和 `Callable`，并提供给 `Executor` 执行。也可以调用线程直接执行 (`FutureTask.run()`)。在这里 `WorkerRunnable` 作为参数传递给了 `FutureTask`。这两个变量会暂时保存在内存中，稍后会用到它们。当要执行 `AsyncTask` 时，需要调用它的 `execute` 方法，代码如下所示：

```
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
    return executeOnExecutor(sDefaultExecutor, params);
}
```

`execute` 方法又调用了 `executeOnExecutor` 方法，代码如下所示：

```
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor
exec, Params... params) {
    if (mStatus != Status.PENDING) {
        switch (mStatus) {
            case RUNNING:
                throw new IllegalStateException("Cannot execute task:"
                    + " the task is already running.");
            case FINISHED:
                throw new IllegalStateException("Cannot execute task:"
                    + " the task has already been executed "
                    + "(a task can be executed only once)");
        }
    }
    mStatus = Status.RUNNING;
    onPreExecute();
    mWorker.mParams = params; //1
    exec.execute(mFuture);
    return this;
}
```

这里会首先调用 `onPreExecute` 方法，在上面代码注释 1 处将 `AsyncTask` 的参数传给 `WorkerRunnable`。从前面我们知道 `WorkerRunnable` 作为参数传递给了 `FutureTask`，因此，参数被封装到 `FutureTask` 中。接下来会调用 `exec` 的 `execute` 方法，并将 `mFuture` 也就是前面讲到的 `FutureTask` 传进去。这里 `exec` 是传进来的参数 `sDefaultExecutor`，它是一个串行的线程池 `SerialExecutor`，其代码如下所示：

```
private static class SerialExecutor implements Executor {
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
    Runnable mActive;
    public synchronized void execute(final Runnable r) {
        mTasks.offer(new Runnable() { //1
            public void run() {
                try {
```

```

        r.run();//2
    } finally {
        scheduleNext();
    }
}

});
if (mActive == null) {
    scheduleNext();
}
}

protected synchronized void scheduleNext() {
    if ((mActive = mTasks.poll()) != null) {
        THREAD_POOL_EXECUTOR.execute(mActive);
    }
}
}
}

```

从上面代码注释1处可以看出，当调用 `SerialExecutor` 的 `execute` 方法时，会将 `FutureTask` 加入到 `mTasks` 中。当任务执行完或者当前没有活动的任务时都会执行 `scheduleNext` 方法，它会从 `mTasks` 取出 `FutureTask` 任务并交由 `THREAD_POOL_EXECUTOR` 处理。关于 `THREAD_POOL_EXECUTOR` 的内容，后面会介绍。从这里可以看出 `SerialExecutor` 是串行执行的。在注释2处可以看到执行了 `FutureTask` 的 `run` 方法，它最终会调用 `WorkerRunnable` 的 `call` 方法。前面我们提到 `call` 方法最终会调用 `postResult` 方法将结果投递出去，`postResult` 方法的代码如下所示：

```

private Result postResult(Result result) {
    @SuppressWarnings("unchecked")
    Message message = getHandler().obtainMessage(MESSAGE_POST_RESULT,
        new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}

```

在 `postResult` 方法中会创建 `Message`，将结果赋值给这个 `Message`，通过 `getHandler` 方法得到 `Handler`，并通过这个 `Handler` 发送消息。`getHandler` 方法如下所示：

```

private static Handler getHandler() {
    synchronized (AsyncTask.class) {
        if (sHandler == null) {
            sHandler = new InternalHandler();
        }
        return sHandler;
    }
}
}

```

在 `getHandler` 方法中创建了 `InternalHandler`，`InternalHandler` 的定义如下所示：

```

private static class InternalHandler extends Handler {
    public InternalHandler() {
        super(Looper.getMainLooper());
    }

    @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
    @Override
    public void handleMessage(Message msg) {
        AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
        switch (msg.what) {
            case MESSAGE_POST_RESULT:
                // There is only one result
                result.mTask.finish(result.mData[0]);
                break;
            case MESSAGE_POST_PROGRESS:
                result.mTask.onProgressUpdate(result.mData);
                break;
        }
    }
}

```

在接收到 MESSAGE_POST_RESULT 消息后会调用 AsyncTask 的 finish 方法，代码如下所示：

```

private void finish(Result result) {
    if (isCancelled()) {
        onCancelled(result);
    } else {
        onPostExecute(result);
    }
    mStatus = Status.FINISHED;
}

```

如果 AsyncTask 任务被取消了，则执行 onCancelled 方法，否则就调用 onPostExecute 方法。正是通过 onPostExecute 方法，我们才能够得到异步任务执行后的结果。接着回头来看 SerialExecutor，线程池 SerialExecutor 主要用来处理排队，将任务串行处理。在 SerialExecutor 中调用 scheduleNext 方法时，将任务交给 THREAD_POOL_EXECUTOR。THREAD_POOL_EXECUTOR 同样是一个线程池，用来处理任务，代码如下所示：

```

private static final int CPU_COUNT = Runtime.getRuntime().availableProcessors();
private static final int CORE_POOL_SIZE = Math.max(2, Math.min(CPU_COUNT - 1, 4));
private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
private static final int KEEP_ALIVE_SECONDS = 30;
private static final BlockingQueue<Runnable> sPoolWorkQueue =

```

```

        new LinkedBlockingQueue<Runnable>(128);
    public static final Executor THREAD_POOL_EXECUTOR;
    static {
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor (
            CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE_SECONDS,
            TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);
        threadPoolExecutor.allowCoreThreadTimeOut(true);
        THREAD_POOL_EXECUTOR = threadPoolExecutor;
    }

```

THREAD_POOL_EXECUTOR 指的就是 `ThreadPoolExecutor`，其核心线程和线程池允许创建的最大线程数都是由 CPU 的核数来计算出来的。它采用的阻塞队列仍旧是 `LinkedBlockingQueue`，容量为 128。到此，Android 7.0 版本的 `AsyncTask` 源码就分析完了。在 `AsyncTask` 中用到了线程池，在线程池中运行线程，并且又用到了阻塞队列。因此，本章前面介绍的知识在本节中做了很好的铺垫。Android 3.0 及以上版本用 `SerialExecutor` 作为默认的线程，它将任务串行地处理，保证一个时间段只有一个任务执行；而 Android 3.0 之前的版本是并行处理的。Android 3.0 之前版本的缺点在 Android 3.0 之后的版本中也不会出现，因为线程是一个接一个执行的，不会出现超过任务数而执行饱和策略的情况。如果想要在 Android 3.0 及以上版本使用并行的线程处理，可以使用如下的代码：

```

asyncTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, "");

```

其中 `asyncTask` 是我们自定义的 `AsyncTask`，当然也可以传入 4.4.3 节中讲到的 4 种线程池，比如传入 `CachedThreadPool`。

```

asyncTask.executeOnExecutor(Executors.newCachedThreadPool(), "");

```

还可以传入自定义的线程池，如下所示：

```

Executor exec =new ThreadPoolExecutor(0, Integer.MAX_VALUE,
    0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
asyncTask.executeOnExecutor(exec, "");

```

4.6 本章小结

本章讲解了 Android 开发中比较容易被读者忽略的线程基础内容，并从线程基础讲起，一步步深入讲解了同步和线程池的知识。最后结合这些知识分析了 `AsyncTask` 的源码，这样读者才会豁然开朗：原来 `AsyncTask` 用到了线程池，而线程池用到了阻塞队列，阻塞队列的原理是……知识是成体系的，平常我们学习一个知识点时，就要将它进行分解，各个击破，然后再回头看这个知识点才会有更多、更深的理解，也更便于记忆。

第 5 章

网络编程与网络框架

本章学习 Android 的网络编程与网络框架。我们首先要了解网络分层、HTTP 协议原理、HttpClient 与 HttpURLConnection 的使用等这些网络编程的基础知识，然后学习 Volley、OkHttp 和 Retrofit 这些主流开源网络框架的使用以及原理分析。

5.1 网络分层

网络分层就是将网络节点所要完成的数据的发送或转发、打包或拆包，以及控制信息的加载或拆出等工作，分别由不同的硬件和软件模块来完成。这样可以将通信和网络互联这一复杂的问题变得较为简单。网络分层有不同的模型，有的模型分 7 层，有的模型分 5 层。这里介绍分 5 层的，因为它更好理解。网络分层的每一层都是为了完成一种功能而设的。为了实现这些功能，就需要遵守共同的规则，这个规则叫作“协议”。网络分层如图 5-1 所示。

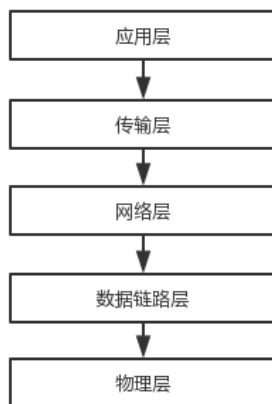


图 5-1 网络分层

如图 5-1 所示，网络分层从上到下分别是应用层、传输层、网络层、数据链路层和物理层。越靠下的层越接近硬件。接下来我们从下而上分别了解这些分层。

1. 物理层

该层负责比特流在节点间的传输，即负责物理传输。该层的协议既与链路有关，也与传输介质有关。其通俗来讲就是把计算机连接起来的物理手段。

2. 数据链路层

该层控制网络层与物理层之间的通信，其主要功能是如何在不可靠的物理线路上进行数据的可靠传递。为了保证传输，从网络层接收到的数据被分割成特定的可被物理层传输的帧。帧是用来移动数据的结构包，它不仅包括原始数据，还包括发送方和接收方的物理地址以及纠错和控制信息。其中的地址确定了帧将发送到何处，而纠错和控制信息则确保帧无差错到达。如果在传送数据时，接收点检测到所传数据中有差错，就要通知发送方重发这一帧。

3. 网络层

该层决定如何将数据从发送方路由到接收方。网络层通过综合考虑发送优先权、网络拥塞程度、服务质量以及可选路由的花费来决定从一个网络中的节点 A 到另一个网络中节点 B 的最佳路径。

4. 传输层

该层为两台主机上的应用程序提供端到端的通信。相比之下，网络层的功能是建立主机到主机的通信。传输层有两个传输协议：TCP（传输控制协议）和 UDP（用户数据报协议）。其中，TCP 是一个可靠的面向连接的协议，UDP 是不可靠的或者说无连接的协议。

5. 应用层

应用程序收到传输层的数据后，接下来就要进行解读。解读必须事先规定好格式，而应用层就是规定应用程序的数据格式的。它的主要协议有 HTTP、FTP、Telnet、SMTP、POP3 等。

5.2 TCP 的三次握手与四次挥手

为什么这里要插入 TCP 的三次握手与四次挥手的知识呢？因为在后面章节分析 OkHttp 源码的时候会涉及。通常我们进行 HTTP 连接网络的时候会进行 TCP 的三次握手，然后传输数据，之后再释放连接。TCP 传输如图 5-2 所示。

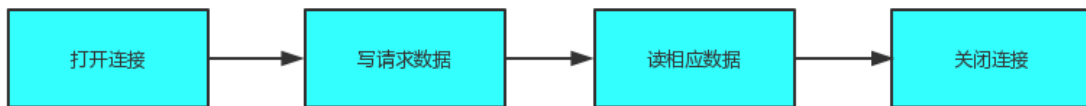


图 5-2 TCP 传输

TCP 三次握手的过程如下。

- 第一次握手：建立连接。客户端发送连接请求报文段，将 SYN 设置为 1、Sequence Number(seq)为 x ；接下来客户端进入 SYN_SENT 状态，等待服务端的确认。
- 第二次握手：服务器收到客户端的 SYN 报文段，对 SYN 报文段进行确认，设置 Acknowledgment Number(ACK)为 $x+1(seq+1)$ ；同时自己还要发送 SYN 请求信息，将 SYN 设置为 1、seq 为 y 。服务端将上述所有信息放到 SYN+ACK 报文段中，一并发送给客户端，此时服务端进入 SYN_RCVD 状态。
- 第三次握手：客户端收到服务端的 SYN+ACK 报文段；然后将 ACK 设置为 $y+1$ ，向服务端发送 ACK 报文段，这个报文段发送完毕后，客户端和服务端都进入 ESTABLISHED (TCP 连接成功) 状态，完成 TCP 的三次握手。

当客户端和服务端通过三次握手建立了 TCP 连接以后，当数据传送完毕，断开连接时就需要进行 TCP 的四次挥手。其四次挥手如下所示。

- 第一次挥手：客户端设置 seq 和 ACK，向服务端发送一个 FIN 报文段。此时，客户端进入 FIN_WAIT_1 状态，表示客户端没有数据要发送给服务端了。
- 第二次挥手：服务端收到了客户端发送的 FIN 报文段，向客户端回了一个 ACK 报文段。
- 第三次挥手：服务端向客户端发送 FIN 报文段，请求关闭连接，同时服务端进入 LAST_ACK 状态。
- 第四次挥手：客户端收到服务端发送的 FIN 报文段，向服务端发送 ACK 报文段，然后客户端进入 TIME_WAIT 状态。服务端收到客户端的 ACK 报文段以后，就关闭连接。此时，客户端等待 2MSL（最大报文段生存时间）后依然没有收到回复，则说明服务端已正常关闭，这样客户端也可以关闭连接了。

现在看图 5-3 来加强一下理解。

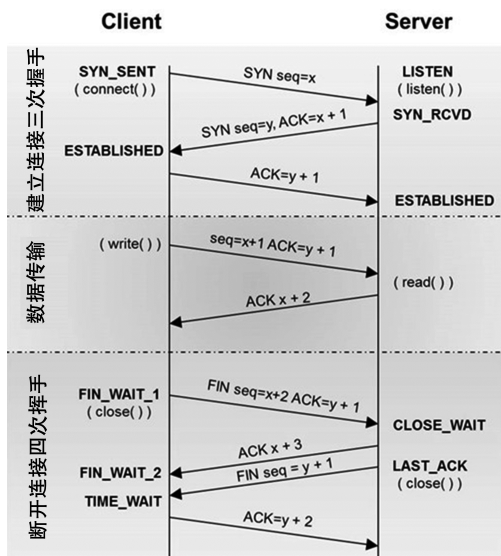


图 5-3 三次握手与四次挥手

如果有大量的连接，每次在连接、关闭时都要经历三次握手、四次挥手，这很显然会造成性能低下。因此，HTTP 有一种叫作 **keepalive connections** 的机制，它可以在传输数据后仍然保持连接，当客户端需要再次获取数据时，直接使用刚刚空闲下来的连接而无须再次握手，如图 5-4 所示。

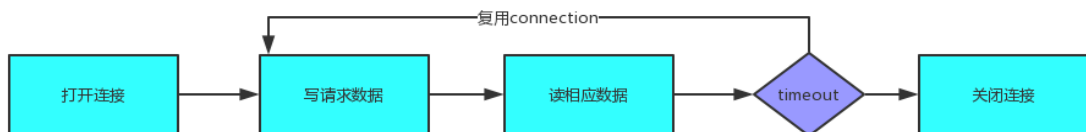


图 5-4 连接复用

5.3 HTTP 协议原理

作为移动开发者，开发的应用不免会对网络进行访问。虽然现在已经有很多开源库帮助我们轻而易举地访问网络，但是我们仍需要去了解网络访问的原理，这也是一个优秀开发人员所必备的知识点。本节我们就先来了解一下 HTTP 协议原理。

5.3.1 HTTP 简介

HTTP 是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于 1990 年被提出，经过几年的使用与发展，得到不断的完善和扩展。

1. HTTP 的历史版本

- HTTP 0.9: 1991 年发布的第一个版本，只有一个命令 GET，服务器只能回应 HTML 格式的字符串。
- HTTP 1.0: 1996 年发布的版本，内容容量大大增加。除了 GET 命令外，还引入了 POST 命令和 HEAD 命令。HTTP 请求和回应的格式除了数据部分，每次通信都必须包括头信息，用来描述一些元数据。
- HTTP 1.1: 1997 发布的版本，进一步完善了 HTTP 协议，直到现在还是最流行的版本。
- SPDY 协议: 2009 年谷歌为了解决 HTTP 1.1 效率不高的问题而自行研发的协议。
- HTTP 2: 2015 年新发布的版本，SPDY 协议的主要特性也在此版本中。

2. HTTP 协议的主要特点

HTTP 协议的主要特点如下。

- 支持 C/S（客户/服务器）模式。
- 简单快速: 客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST，每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。
- 灵活: HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

- 无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。
- 无状态：HTTP 协议是无状态协议，无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大；而另一方面，在服务器不需要先前信息时它的应答速度就较快。

HTTP URL 的格式如下所示：

```
http://host[":"port][abs_path]
```

http 表示要通过 HTTP 协议来定位网络资源；host 表示合法的 Internet 主机域名或者 IP 地址；port 指定一个端口号，为空则使用默认端口 80；abs_path 指定请求资源的 URI（Web 上任意的可用资源）。HTTP 有两种报文，分别是请求报文和响应报文，下面先来查看请求报文。

5.3.2 HTTP 请求报文

HTTP 报文是面向文本的，报文中的每一个字段都是一些 ASCII 码串，各个字段的长度是不确定的。一般一个 HTTP 请求报文由请求行、请求报头、空行和请求数据 4 个部分组成，如图 5-5 所示。



图 5-5 请求报文的一般格式

1. 请求行

请求行由请求方法、URL 字段和 HTTP 协议的版本组成，格式如下：

```
Method Request-URI HTTP-Version CRLF
```

其中 Method 表示请求方法；Request-URI 是一个统一资源标识符；HTTP-Version 表示请求的 HTTP 协议版本；CRLF 表示回车和换行（除了作为结尾的 CRLF 外，不允许出现单独的 CR 或 LF 字符）。

HTTP 请求方法有 8 种，分别是 GET、POST、HEAD、PUT、DELETE、TRACE、CONNECT、OPTIONS。对于移动开发最常用的就是 GET 和 POST 了。

- GET：请求获取 Request-URI 所标识的资源。
- POST：在 Request-URI 所标识的资源后附加新的数据。
- HEAD：请求获取由 Request-URI 所标识的资源的响应消息报头。
- PUT：请求服务器存储一个资源，并用 Request-URI 作为其标识。
- DELETE：请求服务器删除 Request-URI 所标识的资源。
- TRACE：请求服务器回送收到的请求信息，主要用于测试或诊断。
- CONNECT：HTTP 1.1 协议中预留给能够将连接改为管道方式的代理服务器。
- OPTIONS：请求查询服务器的性能，或者查询与资源相关的选项和需求。

例如，访问我的 CSDN 博客地址的请求行：

```
GET http://blog.csdn.net/itachi85 HTTP/1.1
```

2. 请求报头

在请求行之后会有 0 个或者多个请求报头，每个请求报头都包含一个名字和一个值，它们之间用英文冒号“:”分割。关于请求报头，我们会在后面做统一解释。

3. 请求数据

请求数据不在 GET 方法中使用，而在 POST 方法中使用。POST 方法适用于需要客户填写表单的场合，与请求数据相关的最常用的请求报头是 Content-Type 和 Content-Length。

5.3.3 HTTP 响应报文

响应报文的一般格式如图 5-6 所示。

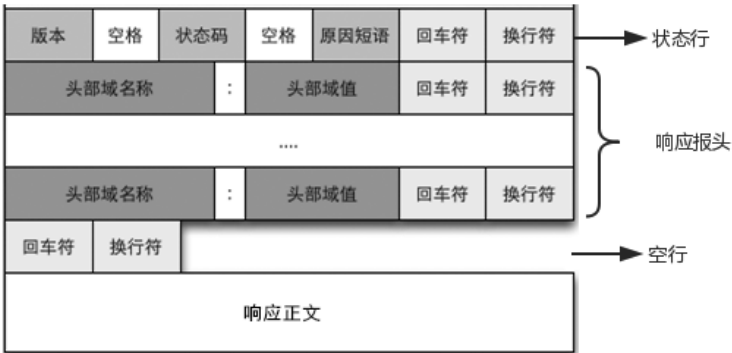


图 5-6 响应报文的一般格式

HTTP 的响应报文由状态行、响应报头、空行、响应正文组成。关于响应报头，我们会在后面做统一解释。响应正文是服务器返回的资源的内容。我们先来看看状态行。

状态行

状态行格式如下所示：

```
HTTP-Version Status-Code Reason-Phrase CRLF
```

其中，HTTP-Version 表示服务器 HTTP 协议的版本；Status-Code 表示服务器发回的响应状态码；Reason-Phrase 表示状态码的文本描述。状态码由 3 位数字组成，第一个数字定义了响应的类别，且有以下 5 种可能取值。

- 100~199：指示信息，收到请求，需要请求者继续执行操作。
- 200~299：请求成功，请求已被成功接收并处理。
- 300~399：重定向，要完成请求必须进行更进一步的操作。
- 400~499：客户端错误，请求有语法错误或请求无法实现。
- 500~599：服务器错误，服务器不能实现合法的请求。

常见的状态码如下。

- 200 OK：客户端请求成功。
- 400 Bad Request：客户端请求有语法错误，服务器无法理解。
- 401 Unauthorized：请求未经授权，这个状态码必须和 WWW-Authenticate 报头域一起使用。
- 403 Forbidden：服务器收到请求，但是拒绝提供服务。
- 500 Internal Server Error：服务器内部错误，无法完成请求。
- 503 Server Unavailable：服务器当前不能处理客户端的请求，一段时间后可能恢复正常。

例如，访问我的 CSDN 博客地址，响应的状态行如下所示：

```
HTTP/1.1 200 OK
```

5.3.4 HTTP 的消息报头

消息报头分为通用报头、请求报头、响应报头、实体报头等。消息报头由键值对组成，每行一对，关键字和值用英文冒号“:”分隔。

1. 通用报头

它既可以出现在请求报头，也可以出现在响应报头中，如下所示。

- Date：表示消息产生的日期和时间。
- Connection：允许发送指定连接的选项。例如指定连接是连续的；或者指定“close”选项，通知服务器，在响应完成后，关闭连接。
- Cache-Control：用于指定缓存指令，缓存指令是单向的（响应中出现的缓存指令在请求中未必会出现），且是独立的（一个消息的缓存指令不会影响另一个消息处理的缓存机制）。

2. 请求报头

请求报头通知服务器关于客户端请求的信息。典型的请求报头如下所示。

- **Host:** 请求的主机名，允许多个域名同处一个 IP 地址，即虚拟主机。
- **User-Agent:** 发送请求的浏览器类型、操作系统等信息。
- **Accept:** 客户端可识别的内容类型列表，用于指定客户端接收哪些类型的信息。
- **Accept-Encoding:** 客户端可识别的数据编码。
- **Accept-Language:** 表示浏览器所支持的语言类型。
- **Connection:** 允许客户端和服务器指定与请求/响应连接有关的选项。例如，这时为 **Keep-Alive** 则表示保持连接。
- **Transfer-Encoding:** 告知接收端为了保证报文的可靠传输，对报文采用了什么编码方式。

3. 响应报头

用于服务器传递自身信息的响应。常见的响应报头如下所示。

- **Location:** 用于重定向接收者到一个新的位置，常用在更换域名的时候。
- **Server:** 包含服务器用来处理请求的系统信息，与 **User-Agent** 请求报头是相对应的。

4. 实体报头

实体报头用来定义被传送资源的信息，其既可用于请求也可用于响应。请求和响应消息都可以传送一个实体。常见的实体报头如下所示。

- **Content-Type:** 发送给接收者的实体正文的媒体类型。
- **Content-Length:** 实体正文的长度。
- **Content-Language:** 描述资源所用的自然语言。
- **Content-Encoding:** 实体报头被用作媒体类型的修饰符。它的值指示了已经被应用到实体正文的附加内容的编码，因而要获得 **Content-Type** 报头域中所引用的媒体类型，必须采用相应的解码机制。
- **Last-Modified:** 实体报头用于指示资源的最后修改日期和时间。
- **Expires:** 实体报头给出响应过期的日期和时间。

5.3.5 抓包应用举例

要想查看网页或者手机请求网络的请求报文和响应报文有很多种方法，在 Windows 中推荐采用 Fiddler 来进行网络数据抓包，在 Mac 平台中则推荐使用 Charles。这里拿 Fiddler 来举例，其官方下载地址为 <https://www.telerik.com/download/fiddler>。打开 Fiddler，然后用浏览器访问我的 CSDN 博客网站。请求报文如图 5-7 所示。

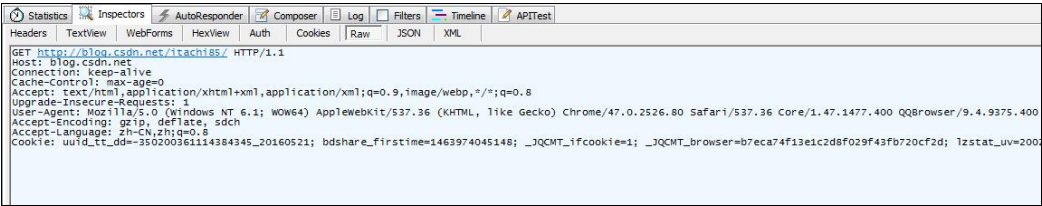


图 5-7 请求报文

从图 5-7 中很容易看出访问的是我的博客地址 `http://blog.csdn.net/itachi85`，请求的方法是 GET，HTTP 版本是 HTTP 1.1。因为请求的方法是 GET，所以并没有请求数据。接下来看响应报文，如图 5-8 所示。



图 5-8 响应报文

如图 5-8 所示，状态行显示请求成功了。这里截取了部分响应正文，是我博客地址的 HTML 数据。

5.4 HttpClient 与 HttpURLConnection

前面我们了解了 HTTP 协议原理，本节讲讲 Apache 的 HttpClient 和 Java 的 HttpURLConnection，它们都是我们平常请求网络会用到的。无论我们是自己封装的网络请求类还是第三方的网络请求框架，都离不开这两个类库。

5.4.1 HttpClient

Android SDK 中包含了 HttpClient。Android 6.0 版本直接删除了 HttpClient 类库。如果仍想使用它，解决方法就是在相应 module 下的 build.gradle 中加入如下代码：

```
android {
    useLibrary 'org.apache.http.legacy'
}
```

1. HttpClient 的 GET 请求

首先用 `DefaultHttpClient` 类来实例化一个 `HttpClient`，并配置好默认的请求参数，代码如下所示：

```
//创建 HttpClient
private HttpClient createHttpClient() {
    HttpParams mDefaultHttpParams = new BasicHttpParams();
    //设置连接超时
    HttpConnectionParams.setConnectionTimeout(mDefaultHttpParams,
        15000);
    //设置请求超时
    HttpConnectionParams.setSoTimeout(mDefaultHttpParams, 15000);
    HttpConnectionParams.setTcpNoDelay(mDefaultHttpParams, true);
    HttpProtocolParams.setVersion(mDefaultHttpParams,
        HttpVersion.HTTP_1_1);
    HttpProtocolParams.setContentCharset(mDefaultHttpParams,
        HTTP.UTF_8);
    //持续握手
    HttpProtocolParams.setUseExpectContinue(mDefaultHttpParams, true);
    HttpClient mHttpClient = new DefaultHttpClient(mDefaultHttpParams);
    return mHttpClient;
}
```

接下来创建 `HttpGet` 和 `HttpClient`，请求网络并得到 `HttpResponse`，并对 `HttpResponse` 进行处理：

```
private void useHttpClientGet(String url) {
    HttpGet mHttpGet = new HttpGet(url);
    mHttpGet.addHeader("Connection", "Keep-Alive");
    try {
        HttpClient mHttpClient = createHttpClient();
        HttpResponse mHttpResponse = mHttpClient.execute(mHttpGet);
        HttpEntity mHttpEntity = mHttpResponse.getEntity();
        int code = mHttpResponse.getStatusLine().getStatusCode();
        if (null != mHttpEntity) {
            InputStream mInputStream = mHttpEntity.getContent();
            String response = converStreamToString(mInputStream); //1
            Log.d(TAG, "请求状态码:" + code + "\n 请求结果:\n" + response);
            mInputStream.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

上面代码注释 1 处的 `converStreamToString` 方法将请求结果转换成 `String` 类型。

```
private String converStreamToString(InputStream is) throws IOException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(is));
    StringBuffer sb = new StringBuffer();
    String line = null;
    while ((line = reader.readLine()) != null) {
        sb.append(line + "\n");
    }
    String response = sb.toString();
    return response;
}
```

最后我们开启线程访问百度。

```
new Thread(new Runnable() {
    @Override
    public void run() {
        useHttpClientGet("http://www.baidu.com");
    }
}).start();
```

图 5-9 为请求的返回结果，请求的状态码为 200，请求结果就是一个 HTML 页，这里只截取了部分 HTML 代码。

```
D/HttpUrl: 请求状态码:200
请求结果:
<!DOCTYPE html><!--STATUS OK-->
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=Edge">
<link rel="dns-prefetch" href="//s1.bdstatic.com"/>
<link rel="dns-prefetch" href="//t1.baidu.com"/>
<link rel="dns-prefetch" href="//t2.baidu.com"/>
<link rel="dns-prefetch" href="//t3.baidu.com"/>
<link rel="dns-prefetch" href="//t10.baidu.com"/>
<link rel="dns-prefetch" href="//t11.baidu.com"/>
<link rel="dns-prefetch" href="//t12.baidu.com"/>
<link rel="dns-prefetch" href="//b1.bdstatic.com"/>
<title>百度一下，你就知道</title>
```

图 5-9 请求结果

GET 请求的参数暴露在 URL 中，这有些不大妥当，而且 URL 的长度也有限制：长度在 2048 字符之内，在 HTTP 1.1 后 URL 长度才没有了限制。一般情况下 POST 可以替代 GET。接下来学习 `HttpClient` 的 POST 请求。

2. HttpClient 的 POST 请求

POST 请求和 GET 请求类似，就是需要配置要传递的参数。这里访问淘宝 IP 库，它的接口说明在地址 <http://ip.taobao.com/instructions.php> 中，如图 5-10 所示。

接口说明
1. 请求接口 (GET) : /service/getIpInfo.php?ip=[ip地址字符串] 2. 响应信息 : (json格式的) 国家、省(自治区或直辖市)、市(县)、运营商 3. 返回数据格式 : <pre>{ "code":0, "data":{ "ip":"210.75.225.254", "country":"\u4e2d\u56fd", "area":"\u534e\u5317", "region":"\u5317\u4eac\u5e02", "city":"\u5317\u4eac\u5e02", "county":"", "isp":"\u7535\u4fe1", "country_id":"86", "area_id":"100000", "region_id":"110000", "city_id":"110000", "county_id":"-1", "isp_id":"100017" } }</pre> 其中code的值的含义为，0：成功，1：失败。

图 5-10 淘宝 IP 库接口说明

这个接口不仅支持 GET 请求，同时也支持 POST 请求，代码如下所示：

```
private void useHttpClientPost(String url) {
    HttpPost mHttpPost = new HttpPost(url);
    mHttpPost.addHeader("Connection", "Keep-Alive");
    try {
        HttpClient mHttpClient = createHttpClient();
        List<NameValuePair> postParams = new ArrayList<>();
        //要传递的参数
        postParams.add(new BasicNameValuePair("ip", "59.108.54.37"));
        mHttpPost.setEntity(new UrlEncodedFormEntity(postParams));
        HttpResponse mHttpResponse = mHttpClient.execute(mHttpPost);
        HttpEntity mHttpEntity = mHttpResponse.getEntity();
        int code = mHttpResponse.getStatusLine().getStatusCode();
        if (null != mHttpEntity) {
            InputStream mInputStream = mHttpEntity.getContent();
            String response = converStreamToString(mInputStream);
            Log.d(TAG, "请求状态码:" + code + "\n 请求结果:\n" + response);
            mInputStream.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

同样地，开启线程访问淘宝 IP 库：


```

new Thread(new Runnable() {
    @Override
    public void run() {
        useHttpClientPost("http://ip.taobao.com/service/getIpInfo.php");
    }
}).start();

```

Log 打印的部分结果如图 5-11 所示。

```

D/HttpUrl: 请求状态码:200
           请求结果:
           {"code":0,"data":{"country":"\u4e2d\u56fd","country_id":"CN","area":"\u534e\u5317","area_id":"100000",
           "ip":"59.108.54.37"}}

```

图 5-11 HttpClient 的 POST 请求结果

从图 5-11 中可以看出请求成功了，并且返回了传入 ip 的信息。

5.4.2 HttpURLConnection

在 Android 2.2 版本及其之前，HttpURLConnection 一直存在着一些令人厌烦的 bug。比如说对一个可读的 `InputStream` 调用 `close()` 方法时，就有可能导致连接池失效。我们通常的解决办法就是直接禁用连接池的功能，代码如下所示：

```

private void disableConnectionReuseIfNecessary() {
    // 这是 2.2 版本之前的一个 bug
    if (Integer.parseInt(Build.VERSION.SDK) < Build.VERSION_CODES.FROYO) {
        System.setProperty("http.keepAlive", "false");
    }
}

```

所以在 Android 2.2 版本及其之前的版本使用 `HttpClient` 是较好的选择；而在 Android 2.3 版本及其之后，`HttpURLConnection` 则是最佳的选择，它的 API 简单，体积较小，因而非常适用于 Android 项目。`HttpURLConnection` 的压缩和缓存机制可以有效地减少网络访问的流量，在提升速度和省电方面也起到了较大的作用。另外在 Android 6.0 版本中，`HttpClient` 库被移除了，如果不引用 `HttpClient`，`HttpURLConnection` 则是以后我们唯一的选择。

HttpURLConnection 的 POST 请求

因为学会了 `HttpURLConnection` 的 POST 请求，GET 请求也就学会了，所以我这里只举出 POST 的例子。首先我们创建一个 `UrlConnManager` 类，然后在里面提供 `getHttpURLConnection()` 方法用于配置默认的参数并返回 `HttpURLConnection`，代码如下所示：

```

public static HttpURLConnection getHttpURLConnection(String url){
    HttpURLConnection mHttpURLConnection=null;

```

```

try {
    URL mUrl=new URL(url);
    mHttpURLConnection=(HttpURLConnection)mUrl.openConnection();
    //设置连接超时时间
    mHttpURLConnection.setConnectTimeout(15000);
    //设置读取超时时间
    mHttpURLConnection.setReadTimeout(15000);
    //设置请求参数
    mHttpURLConnection.setRequestMethod("POST");
    //添加 Header
    mHttpURLConnection.setRequestProperty("Connection","Keep-Alive");
    //接收输入流
    mHttpURLConnection.setDoInput(true);
    //传递参数时需要开启
    mHttpURLConnection.setDoOutput(true);
} catch (IOException e) {
    e.printStackTrace();
}
return mHttpURLConnection ;
}

```

因为我们要发送 POST 请求，所以在 `UrlConnManager` 类中再写一个 `postParams` 方法，组织一下请求参数并将请求参数写入输出流，代码如下所示：

```

public static void postParams(OutputStream output,List<NameValuePair>paramsList)
throws IOException{
    StringBuilder mStringBuilder=new StringBuilder();
    for (NameValuePair pair:paramsList){
        if(!TextUtils.isEmpty(mStringBuilder)){
            mStringBuilder.append("&");
        }
        mStringBuilder.append(URLEncoder.encode(pair.getName(),"UTF-8"));
        mStringBuilder.append("=");
        mStringBuilder.append(URLEncoder.encode(pair.getValue(),"UTF-8"));
    }
    BufferedWriter writer=new BufferedWriter(new OutputStreamWriter
        (output,"UTF-8"));
    writer.write(mStringBuilder.toString());
    writer.flush();
    writer.close();
}

```

接下来我们添加请求参数，调用 `postParams()` 方法将请求的参数组织好并传给 `HttpURLConnection` 的输出流，请求连接并处理返回的结果。这里仍旧访问淘宝 IP 库，代码如下

下所示:

```
private void useHttpURLConnectionPost(String url) {
    InputStream mInputStream = null;
    HttpURLConnection mHttpURLConnection = UrlConnManager.
        getHttpURLConnection(url);
    try {
        List<NameValuePair> postParams = new ArrayList<>();
        postParams.add(new BasicNameValuePair("ip", "59.108.54.37"));
        UrlConnManager.postParams(mHttpURLConnection.getOutputStream(),
            postParams);
        mHttpURLConnection.connect();
        mInputStream = mHttpURLConnection.getInputStream();
        int code = mHttpURLConnection.getResponseCode();
        String respose = converStreamToString(mInputStream);
        Log.d(TAG, "请求状态码:" + code + "\n 请求结果:\n" + respose);
        mInputStream.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

最后开启线程请求淘宝 IP 库, 如下所示:

```
new Thread(new Runnable() {
    @Override
    public void run() {
        useHttpURLConnectionPost("http://ip.taobao.com/service/
            getIpInfo.php");
    }
}).start();
```

5.5 解析 Volley

在 2013 年 Google I/O 大会上推出了一个新的网络通信框架 Volley。Volley 既可以访问网络取得数据, 也可以加载图片, 并且在性能方面进行了大幅度的调整。它的设计目标就是适合进行数据量不大但通信频繁的网络操作。而对于大数据量的网络操作, 比如说下载文件等, Volley 的表现却非常糟糕。

5.5.1 Volley 基本用法

在使用 Volley 前请下载 Volley 库且放在 libs 目录下并 add 到工程中。其下载地址为 <http://central.maven.org/maven2/com/mcxiaoke/volley/library/>。

1. Volley 网络请求队列

Volley 请求网络都是基于请求队列的，开发者只要把请求放在请求队列中就可以了，请求队列会依次进行请求。一般情况下，一个应用程序如果网络请求不是特别频繁，完全可以只有一个请求队列（对应 Application）；如果网络请求非常多或有其他情况，则可以是一个 Activity 对应一个网络请求队列，这就要看具体情况了。首先创建队列，如下所示。

```
RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
```

2. StringRequest 的用法

StringRequest 返回的数据是 String 类型的。我们查看一下 StringRequest 的源码：

```
public class StringRequest extends Request<String> {
    private final Listener<String> mListener;
    public StringRequest(int method, String url, Listener<String> listener,
        ErrorListener errorListener) {
        super(method, url, errorListener);
        this.mListener = listener;
    }

    public StringRequest(String url, Listener<String> listener, ErrorListener
        errorListener) {
        this(0, url, listener, errorListener);
    }
    ...
}
```

这里有两个构造方法，其中第一个比第二个多了一个请求的方法，如果采用第二个则默认是 GET 请求。我们试着用 GET 方法来请求百度，代码如下所示：

```
RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
StringRequest mStringRequest = new StringRequest(Request.Method.GET,
    "http://www.baidu.com",
    new Response.Listener<String>() {
        @Override
        public void onResponse(String response) {
            Log.d(TAG, response);
        }
    }, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            Log.e(TAG, error.getMessage(), error);
        }
    });
```

```
//将请求添加到请求队列中
mQueue.add(mStringRequest);
```

请求结果不用说是百度界面的 HTML 文件。

3. JsonRequest 的用法

为了返回的数据是 JSON 格式的数据，这里仍旧访问 5.4 节用到的淘宝 IP 库，解析 JSON 数据，这里采用的是 Gson 库。针对这个接口返回的 JSON 数据格式，我们可以采用 <http://www.bejson.com/json2javapojo/> 这个网站来将 JSON 字符串转换成 Java 实体类。生成的实体类经过了简单修改，如下所示。

```
public class IpModel {
    private int code;
    private IpData data;
    public void setCode(int code) {
        this.code = code;
    }
    public int getCode() {
        return this.code;
    }
    public void setData(IpData data) {
        this.data = data;
    }
    public IpData getData() {
        return this.data;
    }
}
```

IpModel 中包含了 IpData，IpData 类如下所示：

```
public class IpData {
    private String country;
    private String country_id;
    private String area;
    private String area_id;
    private String region;
    private String region_id;
    private String city;
    private String city_id;
    private String county;
    private String county_id;
    private String isp;
    private String isp_id;
    private String ip;
```

```

    public void setCountry(String country) {
        this.country = country;
    }
    public String getCountry() {
        return this.country;
    }
    public void setCountry_id(String country_id) {
        this.country_id = country_id;
    }
    ...
}

```

JsonRequest 和 StringRequest 的使用方法类似，如下所示：

```

JsonObjectRequest mJsonObjectRequest = new JsonObjectRequest
(Request.Method.POST,
"http://ip.taobao.com/service/getIpInfo.php?ip=59.108.54.37",
    new Response.Listener<JSONObject>() {
        @Override
        public void onResponse(JSONObject response) {
            IpModel ipModel = new Gson().fromJson(response.
                toString(), IpModel.class);
            if (null != ipModel && null != ipModel.getData()) {
                String city = ipModel.getData().getCity();
                Log.d(TAG, city);
            }
        }
    }, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            Log.e(TAG, error.getMessage(), error);
        }
    });
mQueue.add(mJsonObjectRequest);

```

最终 Log 打印的结果是“北京”，很显然通过淘宝 IP 库查询 59.108.54.37 这个 IP 地址所在的地理位置为北京（这里的 59.108.54.37 为作者举例虚构的 IP 地址，如有雷同纯属巧合）。

4. 使用 ImageRequest 加载图片

ImageRequest 已经是过时的方法了，其和 StringRequest、JsonRequest 的用法类似，代码如下所示：

```
RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
```

```

ImageRequest imageRequest = new ImageRequest(
    "http://img.my.csdn.net/uploads/201603/26/1458988468_5804.jpg",
    new Response.Listener<Bitmap>() {
        @Override
        public void onResponse(Bitmap response) {
            iv_image.setImageBitmap(response);
        }
    }, 0, 0, Bitmap.Config.RGB_565, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            iv_image.setImageResource(R.drawable.ico_default);
        }
    });
mQueue.add(imageRequest);

```

查看 `ImageRequest` 的源码发现，它可以设置你想要的图片的最大宽度和高度。在加载图片时，如果图片超过期望的最大宽度和高度，则会进行压缩：

```

public ImageRequest(String url, Listener<Bitmap> listener, int maxWidth,
    int maxHeight, ScaleType scaleType, Config decodeConfig, ErrorListener
    errorListener) {
    super(0, url, errorListener);
    this.setRetryPolicy(new DefaultRetryPolicy(1000, 2, 2.0F));
    this.mListener = listener;
    this.mDecodeConfig = decodeConfig;
    this.mMaxWidth = maxWidth;
    this.mMaxHeight = maxHeight;
    this.mScaleType = scaleType;
}

```

5. 使用 ImageLoader 加载图片

`ImageLoader` 的内部使用 `ImageRequest` 来实现，它的构造方法可以传入一个 `ImageCache` 缓存形参，实现图片缓存的功能；同时还可以过滤重复连接，避免重复发送请求。与 `ImageRequest` 实现效果不同的是，`ImageLoader` 加载图片会先显示默认的图片，等待图片加载完成才会显示在 `ImageView` 上。

```

RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
ImageLoader imageLoader = new ImageLoader(mQueue, new BitmapCache());
ImageLoader.ImageListener listener = ImageLoader.getImageListener
imageLoader.get("http://img.my.csdn.net/uploads/201603/26/
1458988468_5804.jpg", listener);

```

`ImageLoader` 也提供了设置最大宽度和高度的方法，如下所示：

```
public ImageLoader.ImageContainer get(String requestUrl, ImageLoader.
    ImageListener imageListener, int maxWidth, int maxHeight) {
    return this.get(requestUrl, imageListener, maxWidth, maxHeight,
        ScaleType.CENTER_INSIDE);
}
```

6. 使用 NetworkImageView 加载图片

NetworkImageView 是一个自定义控件，继承自 ImageView，其封装了请求网络加载图片的功能。先在布局中引用：

```
<com.android.volley.toolbox.NetworkImageView
    android:id="@+id/nv_image"
    android:layout_width="200dp"
    android:layout_height="200dp"
    android:layout_centerHorizontal="true"
    android:layout_below="@id/iv_image"
    android:layout_marginTop="20dp">
</com.android.volley.toolbox.NetworkImageView>
```

接着在代码中调用，其和 ImageLoader 用法类似，如下所示：

```
iv_image = (ImageView) this.findViewById(R.id.iv_image);
RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
ImageLoader imageLoader = new ImageLoader(mQueue, new BitmapCache());
nv_image.setDefaultImageResId(R.drawable.ico_default);
nv_image.setErrorImageResId(R.drawable.ico_default);
nv_image.setImageUrl("http://img.my.csdn.net/uploads/201603/26/
1458988468_5804.jpg", imageLoader);
```

7. NetworkImageView

其并没有提供设置最大宽度和高度的方法，而是根据我们设置控件的宽和高，结合网络图片的宽和高内部会自动实现压缩。如果我们不想要压缩，也可以设置 NetworkImageView 控件的宽和高都为 wrap_content。

5.5.2 源码解析 Volley

1. 从 RequestQueue 入手

使用 Volley 之前首先要创建 RequestQueue，就从这里开始入手，如下所示：

```
RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
```

这也是 Volley 运作的入口。接着查看 newRequestQueue 做了什么：


```

public static RequestQueue newRequestQueue(Context context) {
    return newRequestQueue(context, (HttpStack)null);
}

public static RequestQueue newRequestQueue(Context context, HttpStack stack) {
    return newRequestQueue(context, stack, -1);
}

```

这里连续调用了两个重载函数，最终调用的是如下代码：

```

public static RequestQueue newRequestQueue(Context context, HttpStack
stack, int maxDiskCacheBytes) {
    File cacheDir = new File(context.getCacheDir(), "volley");
    String userAgent = "volley/0";
    try {
        String network = context.getPackageName();
        PackageInfo queue = context.getPackageManager().getPackageInfo
(network, 0);
        userAgent = network + "/" + queue.versionCode;
    } catch (NameNotFoundException var7) {
        ;
    }
    if(stack == null) {
        if(VERSION.SDK_INT >= 9) { //1
            stack = new HurlStack();
        } else {
            stack = new HttpClientStack(AndroidHttpClient.newInstance
(userAgent));
        }
    }
    BasicNetwork network1 = new BasicNetwork((HttpStack)stack);
    RequestQueue queue1;
    if(maxDiskCacheBytes <= -1) {
        queue1 = new RequestQueue(new DiskBasedCache(cacheDir), network1);
    } else {
        queue1 = new RequestQueue(new DiskBasedCache(cacheDir,
maxDiskCacheBytes), network1);
    }
    queue1.start();
    return queue1;
}

```

上面代码注释 1 处说明，如果 Android 版本大于或等于 2.3，则调用基于 `HttpURLConnection` 的 `HurlStack`，否则就调用基于 `HttpClient` 的 `HttpClientStack`。接下来创建 `RequestQueue`，并调用它的 `start()` 方法：

```

public void start() {
    this.stop();
    this.mCacheDispatcher = new CacheDispatcher(this.mCacheQueue,
        this.mNetworkQueue, this.mCache, this.mDelivery);
    this.mCacheDispatcher.start();

    for(int i = 0; i < this.mDispatchers.length; ++i) {
        NetworkDispatcher networkDispatcher = new NetworkDispatcher
            (this.mNetworkQueue, this.mNetwork, this.mCache, this.mDelivery);
        this.mDispatchers[i] = networkDispatcher;
        networkDispatcher.start();
    }
}

```

CacheDispatcher 是缓存调度线程，并调用了 start()方法。在循环中调用了 NetworkDispatcher 的 start()方法。NetworkDispatcher 是网络调度线程，默认情况下 mDispatchers.length 为 4，默认开启了 4 个网络调度线程，加上 1 个缓存调度线程，就有 5 个线程在后台运行并等待请求的到来。按照 Volley 的使用流程，接下来会创建各种 Request，并调用 RequestQueue 的 add()方法：

```

public <T> Request<T> add(Request<T> request) {
    request.setRequestQueue(this);
    Set var2 = this.mCurrentRequests;
    synchronized(this.mCurrentRequests) {
        this.mCurrentRequests.add(request);
    }
    request.setSequence(this.getSequenceNumber());
    request.addMarker("add-to-queue");
    //如果不能缓存，则将请求添加到网络请求队列中
    if(!request.shouldCache()) { //1
        this.mNetworkQueue.add(request);
        return request;
    } else {
        Map var8 = this.mWaitingRequests;
        synchronized(this.mWaitingRequests) {
            String cacheKey = request.getCacheKey();

            //如果此前有相同的请求且还没有返回结果的，就将此请求加入 mWaitingRequests 队列
            if(this.mWaitingRequests.containsKey(cacheKey)) {
                Object stagedRequests = (Queue)this.mWaitingRequests.get
                    (cacheKey);
                if(stagedRequests == null) {
                    stagedRequests = new LinkedList();

```

```

    }

    ((Queue) stagedRequests).add(request);
    this.mWaitingRequests.put(cacheKey, stagedRequests);
    if (VolleyLog.DEBUG) {
        VolleyLog.v("Request for cacheKey=%s is in flight,
            putting on hold.", new Object[]{cacheKey});
    }
} else {
    //没有的话就将请求加入缓存队列 mCacheQueue
    this.mWaitingRequests.put(cacheKey, (Object)null);
    this.mCacheQueue.add(request);
}
return request;
}
}
}

```

上面代码注释 1 处通过判断 `request.shouldCache()` 来判断是否可以缓存,默认是可以缓存的。如果不能缓存,则将请求添加到网络请求队列中。如果能缓存,就判断之前是否有执行相同的请求且还没有返回结果的,如果有的话将此请求加入 `mWaitingRequests` 队列,不再重复请求;没有的话就将请求加入缓存队列 `mCacheQueue`。`RequestQueue` 的 `add` 方法并没有请求网络或者对缓存进行操作。当将请求添加到网络请求队列或者缓存队列时,在后台的网络调度线程和缓存调度线程轮询各自的请求队列,若发现有请求任务则开始执行。下面先看看缓存调度线程。

2. CacheDispatcher 缓存调度线程

`CacheDispatcher` 的 `run` 方法代码如下所示:

```

public void run() {
    if (DEBUG) {
        VolleyLog.v("start new dispatcher", new Object[0]);
    }
    //线程优先级设置为最高级别
    Process.setThreadPriority(10);
    this.mCache.initialize();
    while (true) {
        while (true) {
            while (true) {
                try {
                    //获取缓存队列中的一个请求
                    final Request e = (Request) this.mCacheQueue.take();
                    e.addMarker("cache-queue-take");
                }
            }
        }
    }
}

```

```

        //如果请求取消了，则将请求停止
        if(e.isCanceled()) {
            e.finish("cache-discard-canceled");
        } else {
            //查看是否有缓存的响应
            Entry entry = this.mCache.get(e.getCacheKey());
            //如果缓存的响应为空，则将请求加入网络队列
            if(entry == null) {
                e.addMarker("cache-miss");
                this.mNetworkQueue.put(e);
                //判断缓存的响应是否过期
            } else if(!entry.isExpired()) {
                e.addMarker("cache-hit");
                //对数据进行解析并回调给主线程
                Response response = e.parseNetworkResponse
                    (new NetworkResponse(entry.data, entry.
                        responseHeaders));
            }
        }
    }
}

```

读者看到4个while循环估计会有些晕，下面挑重点进行解释。首先从缓存队列取出请求，判断请求是否被取消了，如果请求没有被取消则判断该请求是否有缓存的响应。如果有缓存的响应并且没有过期，则对缓存响应进行解析并回调给主线程；如果没有缓存的响应，则将请求加入网络调度线程。接下来看看网络调度线程。

3. NetworkDispatcher 网络调度线程

NetworkDispatcher 的 run 方法代码如下所示：

```

public void run() {
    Process.setThreadPriority(10);
    while(true) {
        long startTimeMs;
        Request request;
        while(true) {
            startTimeMs = SystemClock.elapsedRealtime();
            try {
                //从队列中取出请求
                request = (Request) this.mQueue.take();
                break;
            } catch (InterruptedException var6) {
                if(this.mQuit) {
                    return;
                }
            }
        }
    }
}

```

```

    }
    try {
        request.addMarker("network-queue-take");
        if(request.isCanceled()) {
            request.finish("network-discard-cancelled");
        } else {
            this.addTrafficStatsTag(request);
            //请求网络
            NetworkResponse e = this.mNetwork.performRequest(request);
            request.addMarker("network-http-complete");
            if(e.notModified && request.hasHadResponseDelivered()) {
                request.finish("not-modified");
            } else {
                Response volleyError1 = request.parseNetworkResponse(e);
                request.addMarker("network-parse-complete");
                if(request.shouldCache() && volleyError1.cacheEntry != null)
                {
                    //将响应存入缓存
                    this.mCache.put(request.getCacheKey(),
                        volleyError1.cacheEntry);
                    request.addMarker("network-cache-written");
                }
                request.markDelivered();
                this.mDelivery.postResponse(request, volleyError1);
            }
        }
        ...
    }
}

```

网络调度线程也是从队列中取出请求并且判断该请求是否被取消了。如果该请求没被取消，就去请求网络得到响应并回调给主线程。请求网络时调用 `this.mNetwork.performRequest(request)`，这个 `mNetwork` 是一个接口，实现它的类是 `BasicNetwork`。接下来查看 `BasicNetwork` 的 `performRequest()`方法：

```

public NetworkResponse performRequest(Request<?> request) throws
VolleyError {
    long requestStart = SystemClock.elapsedRealtime();
    while(true) {
        HttpResponse httpResponse = null;
        Object responseContents = null;
        Map responseHeaders = Collections.emptyMap();
        try {
            HashMap e = new HashMap();
            this.addCacheHeaders(e, request.getCacheEntry());

```

```

        httpResponse = this.mHttpStack.performRequest(request, e); //1
        StatusLine statusCode1 = httpResponse.getStatusLine();
        int networkResponse1 = statusCode1.getStatusCode();
        responseHeaders = convertHeaders(httpResponse.getAllHeaders());
        if(networkResponse1 == 304) {
            Entry requestLifetime2 = request.getCacheEntry();
            if(requestLifetime2 == null) {
                return new NetworkResponse(304, (byte[])null,
                    responseHeaders, true, SystemClock.elapsedRealtime()
                        - requestStart);
            }
            requestLifetime2.responseHeaders.putAll(responseHeaders);
            return new NetworkResponse(304, requestLifetime2.data,
                requestLifetime2.responseHeaders, true, SystemClock.
                    elapsedRealtime() - requestStart);
        }

        ...
    }

```

上面代码注释 1 处调用 `HttpStack` 的 `performRequest` 方法请求网络，接下来根据不同的响应状态码来返回不同的 `NetworkResponse`。另外 `HttpStack` 也是一个接口，实现它的两个类我们在前面已经提到了，就是 `HurlStack` 和 `HttpClientStack`。让我们再回到 `NetworkDispatcher`，请求网络后，会将响应结果存在缓存中，并调用 `this.mDelivery.postResponse(request, volleyError1)` 来回调给主线程。查看 `Delivery` 的 `postResponse` 方法，如下所示：

```

public void postResponse(Request<?> request, Response<?> response, Runnable
    runnable) {
    request.markDelivered();
    request.addMarker("post-response");
    this.mResponsePoster.execute(new ExecutorDelivery
        .ResponseDeliveryRunnable(request, response, runnable));
}

```

查看 `ResponseDeliveryRunnable` 里面做了什么：

```

private class ResponseDeliveryRunnable implements Runnable {
    private final Request mRequest;
    private final Response mResponse;
    private final Runnable mRunnable;
    public ResponseDeliveryRunnable(Request request, Response response,
        Runnable runnable) {
        this.mRequest = request;
    }
}

```

```

        this.mResponse = response;
        this.mRunnable = runnable;
    }
    public void run() {
        if(this.mRequest.isCanceled()) {
            this.mRequest.finish("canceled-at-delivery");
        } else {
            if(this.mResponse.isSuccess()) {
                this.mRequest.deliverResponse(this.mResponse.result);//1
            } else {
                this.mRequest.deliverError(this.mResponse.error);
            }
        }
    }
    ...
}

```

上面代码注释 1 处调用了 Request 的 deliverResponse 方法，假设这里我们使用的是 StringRequest，它继承自 Request，因此查看 StringRequest 的源码：

```

public class StringRequest extends Request<String> {
    private final Listener<String> mListener;
    public StringRequest(int method, String url, Listener<String> listener,
        ErrorListener errorListener) {
        super(method, url, errorListener);
        this.mListener = listener;
    }
    public StringRequest(String url, Listener<String> listener, ErrorListener
        errorListener) {
        this(0, url, listener, errorListener);
    }
    protected void deliverResponse(String response) {
        this.mListener.onResponse(response);
    }
    ...
}

```

在 deliverResponse 方法中调用了 this.mListener.onResponse(response)，最终将 response 回调给了 Response.Listener 的 onResponse()方法。我们用 StringRequest 请求网络的写法是这样的：

```

RequestQueue mQueue = Volley.newRequestQueue(getApplicationContext());
StringRequest mStringRequest = new StringRequest(Request.Method.GET,
    "http://www.baidu.com",
    new Response.Listener<String>() {
        @Override
        public void onResponse(String response) {//1

```

```

        Log.i("wangshu", response);
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
        Log.e("wangshu", error.getMessage(), error);
    }
});
//将请求添加到请求队列中
mQueue.add(mStringRequest);

```

上面代码注释 1 处将请求网络得到的 response 通过 Response.Listener 的 onResponse 方法回调回来，这样整个 Volley 的大致流程就走通了。再看看图 5-12，加深一下对 Volley 请求流程的理解。

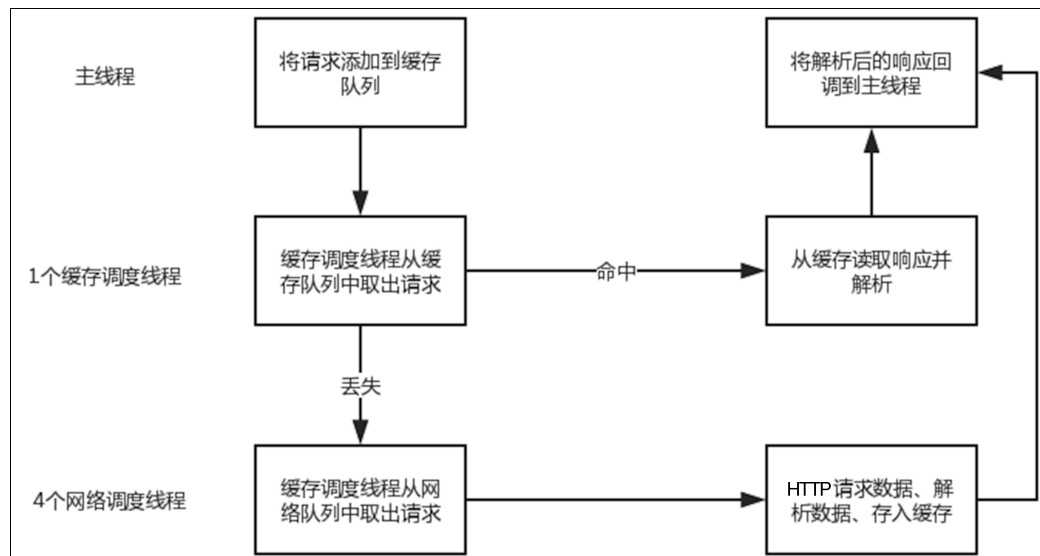


图 5-12 Volley 请求流程

从图 5-12 中可以看到 Volley 分为三类线程，分别是主线程、缓存调度线程和网络调度线程，其中网络调度线程默认开启 4 个。首先请求会加入缓存队列，缓存调度线程从缓存队列中取出请求。如果找到该请求的缓存响应就直接读取缓存的响应并解析，然后回调给主线程；如果没有找到缓存的响应，则将该条请求加入网络队列，然后网络调度线程会轮询取出网络队列中的请求，取出后发送 HTTP 请求，解析响应并将响应存入缓存，并回调给主线程。

5.6 解析 OkHttp

讲完了 Volley，接下来学习目前比较火的网络框架 OkHttp。它处理了很多网络疑难杂症，比如会从很多常用的连接问题中自动恢复。如果你的服务器配置了多个 IP 地址，当第一个 IP

连接失败的时候，OkHttp 会自动尝试下一个 IP。从 Android 4.4 版本开始，系统内置了 OkHttp，可见 OkHttp 功能的强大。

5.6.1 OkHttp 基本用法

1. 使用前的准备工作

首先配置 gradle，如下所示：

```
compile 'com.squareup.okhttp3:okhttp:3.2.0'
compile 'com.squareup.okio:okio:1.7.0'
```

okio 框架作为 OkHttp 的 io 组件，也是必须要引入的。当然，不要忘了在 manifest 中添加网络权限。

2. 异步 GET 请求

最简单的 GET 请求，请求我的 CSDN 博客地址，代码如下所示：

```
Request.Builder requestBuilder = new Request.Builder().url("http://
blog.csdn.net/itachi85");
requestBuilder.method("GET", null);
Request request = requestBuilder.build();
OkHttpClient mOkHttpClient = new OkHttpClient();
Call mcall = mOkHttpClient.newCall(request);
mcall.enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
    }
    @Override
    public void onResponse(Call call, Response response) throws IOException {
        String str = response.body().string();
        Log.d(TAG, str);
    }
});
```

其基本步骤就是创建 OkHttpClient、Request 和 Call，最后调用 Call 的 enqueue() 方法。但是每次这么写很麻烦，肯定是要进行封装的。需要注意的是 onResponse 回调并非在 UI 线程。如果想要调用同步 GET 请求，可以调用 Call 的 execute 方法。

3. 异步 POST 请求

OkHttp 3 异步 POST 请求和 OkHttp 2.x 有一些差别，就是没有 FormEncodingBuilder 这个类，替代它的是功能更加强大的 FormBody。这里访问淘宝 IP 库，代码如下所示：

```
RequestBody formBody = new FormBody.Builder()
    .add("ip", "59.108.54.37")
```

```

        .build();
Request request = new Request.Builder()
    .url("http://ip.taobao.com/service/getIpInfo.php")
    .post(formBody)
    .build();
OkHttpClient mOkHttpClient = new OkHttpClient();
Call call = mOkHttpClient.newCall(request);
call.enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
    }
    @Override
    public void onResponse(Call call, Response response) throws
        IOException {
        String str = response.body().string();
        Log.d(TAG, str);
    }
});

```

这与异步 GET 请求类似，只是多了用 `FormBody` 来封装请求的参数，并传递给 `Request`。

4. 异步上传文件

上传文件本身也是一个 POST 请求，首先定义上传文件类型：

```

public static final MediaType MEDIA_TYPE_MARKDOWN
    = MediaType.parse("text/x-markdown; charset=utf-8");

```

在 SD 卡根目录创建一个 `wangshu.txt` 文件，里面的内容为 "OKHttp"。

```

String filepath = "";
if (Environment.getExternalStorageState().equals(
    Environment.MEDIA_MOUNTED)) {
    filepath = Environment.getExternalStorageDirectory().getAbsolutePath();
} else {
    return;
}
File file = new File(filepath, "wangshu.txt");
Request request = new Request.Builder()
    .url("https://api.github.com/markdown/raw")
    .post(RequestBody.create(MEDIA_TYPE_MARKDOWN, file))
    .build();
mOkHttpClient.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {

```

```

    }
    @Override
    public void onResponse(Call call, Response response) throws
    IOException {
        Log.d(TAG, response.body().string());
    }
    });

```

当然，如果想要改为同步上传文件，只要调用 `OkHttpClient.newCall(request).execute()` 就可以了。最终请求网络返回的结果就是我们 txt 文件中的内容。

5. 异步下载文件

下载一张图片，得到 `Response` 后将流写进我们指定的图片文件中，代码如下所示：

```

String url = "http://img.my.csdn.net/uploads/201603/26/1458988468_5804.jpg";
Request request = new Request.Builder().url(url).build();
mOkHttpClient.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
    }
    @Override
    public void onResponse(Call call, Response response) {
        InputStream inputStream = response.body().byteStream();
        FileOutputStream outputStream = null;
        String filepath="";
        try {
            if (Environment.getExternalStorageState().equals(
                Environment.MEDIA_MOUNTED)) {
                filepath = Environment.getExternalStorageDirectory().
                    getAbsolutePath();
            } else {
                filepath=getFilesDir().getAbsolutePath();
            }
            File file=new File(filepath,"wangshu.jpg");
            if(null!=file) {
                outputStream = new FileOutputStream(file);
                byte[] buffer = new byte[2048];
                int len = 0;
                while ((len = inputStream.read(buffer)) != -1) {
                    outputStream.write(buffer, 0, len);
                }
                outputStream.flush();
            } catch (IOException e) {

```

```

        Log.d(TAG, "IOException");
        e.printStackTrace();
    }
}
});

```

6. 异步上传 Multipart 文件

有时上传文件，同时还需要传其他类型的字段。OkHttp 3 实现起来很简单。需要注意的是没有服务器接收我这个 **Multipart** 文件，所以这里只是举个例子，具体的应用还要结合实际工作中对应的服务器。

```

private static final MediaType MEDIA_TYPE_PNG = MediaType.parse("image/
png");
private void sendMultipart(){
    mOkHttpClient = new OkHttpClient();
    RequestBody requestBody = new MultipartBody.Builder()
        .setType(MultipartBody.FORM)
        .addFormDataPart("title", "wangshu")//1
        .addFormDataPart("image", "wangshu.jpg",
            RequestBody.create(MEDIA_TYPE_PNG,newFile("/sdcard/
            wangshu.jpg")))//2
        .build();

    Request request = new Request.Builder()
        .header("Authorization", "Client-ID " + "...")
        .url("https://api.imgur.com/3/image")
        .post(requestBody)
        .build();

    mOkHttpClient.newCall(request).enqueue(new Callback() {
        @Override
        public void onFailure(Call call, IOException e) {
        }
        @Override
        public void onResponse(Call call, Response response) throws IOException {
            Log.d(TAG, response.body().string());
        }
    });
}
}

```

首先我们要定义上传文件类型，上面代码注释 1 处是常见的 **key-value** 形式的参数上传；而注释 2 处则上传表单。**addFormDataPart** 方法的第一个参数是 **key** 值，第二个参数是上传文件的名称，第三个参数是需要上传的文件。

7. 设置超时时间和缓存

和 OkHttp 2.x 有区别的是 OkHttp 3 不能通过 OkHttpClient 直接设置超时时间和缓存了，而是通过 OkHttpClient.Builder 来设置。通过 builder 配置好 OkHttpClient 后用 builder.build() 来返回 OkHttpClient。所以我们通常不会调用 new OkHttpClient() 来得到 OkHttpClient，而是通过 builder.build() 得到 OkHttpClient。另外，OkHttp 3 支持设置连接、写入和读取的超时时间，如下所示：

```
File sdcache = getExternalCacheDir();
int cacheSize = 10 * 1024 * 1024;
OkHttpClient.Builder builder = new OkHttpClient.Builder()
    .connectTimeout(15, TimeUnit.SECONDS)
    .writeTimeout(20, TimeUnit.SECONDS)
    .readTimeout(20, TimeUnit.SECONDS)
    .cache(new Cache(sdcache.getAbsoluteFile(), cacheSize));
mOkHttpClient = builder.build();
```

8. 取消请求

使用 call.cancel() 可以立即停止一个正在执行的 call。当用户离开一个应用时或者跳到其他界面时，使用 call.cancel() 可以节约网络资源；另外，不管同步还是异步的 call 都可以取消，也可以通过 tag 来同时取消多个请求。当构建一个请求时，使用 Request.Builder.tag(Object tag) 来分配一个标签，之后你就可以用 OkHttpClient.cancel(Object tag) 来取消所有带有这个 tag 的 call。具体代码如下所示：

```
private ScheduledExecutorService executor =
    Executors.newScheduledThreadPool(1);
private void cancel() {
    final Request request = new Request.Builder()
        .url("http://www.baidu.com")
        .cacheControl(CacheControl.FORCE_NETWORK) //1
        .build();
    Call call = null;
    call = mOkHttpClient.newCall(request);
    final Call finalCall = call;
    //100ms 后取消 call
    executor.schedule(new Runnable() {
        @Override
        public void run() {
            finalCall.cancel();
        }
    }, 100, TimeUnit.MILLISECONDS);
    call.enqueue(new Callback() {
        @Override
```

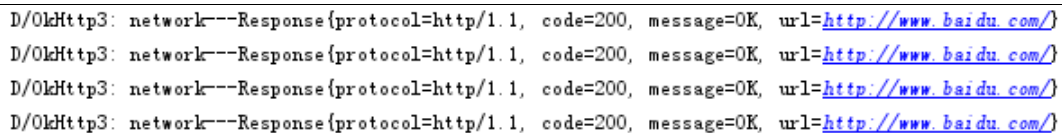
```

public void onFailure(Call call, IOException e) {
}

@Override
public void onResponse(Call call, Response response) throws
IOException {
    if (null != response.cacheResponse()) {
        String str = response.cacheResponse().toString();
        Log.d(TAG, "cache---" + str);
    } else {
        String str = response.networkResponse().toString();
        Log.d(TAG, "network---" + str);
    }
}
});
}

```

创建定时线程池，100 ms 后调用 `call.cancel()` 来取消请求。为了能让请求耗时，在上面代码注释 1 处设置每次请求都要请求网络，运行程序并且不断地调用 `cancel` 方法。Log 打印结果如图 5-13 所示。



```

D/OkHttp3: network---Response{protocol=http/1.1, code=200, message=OK, url=http://www.baidu.com/}
D/OkHttp3: network---Response{protocol=http/1.1, code=200, message=OK, url=http://www.baidu.com/}
D/OkHttp3: network---Response{protocol=http/1.1, code=200, message=OK, url=http://www.baidu.com/}
D/OkHttp3: network---Response{protocol=http/1.1, code=200, message=OK, url=http://www.baidu.com/}

```

图 5-13 Log 打印结果

很明显每次 `cancel()` 都失败了，也就是仍旧成功地访问了网络。每隔 100 ms 来调用 `call.cancel()` 显然时间间隔太长，我们设置为 1 ms，再运行程序并且不断地调用 `cancel` 方法，这时就会发现没有 Log 打印出来，那是因为每个请求都被取消了。

9. 关于封装

如果每次请求网络都需要写重复的代码，那绝对是令人头疼的。网上有很多对 `OkHttp` 进行封装的优秀开源项目，功能非常强大。封装的意义就在于更加方便地使用，且具有拓展性，但是对 `OkHttp` 进行封装最需要解决的是以下两点：

- (1) 避免重复代码调用；
- (2) 将请求结果回调改为 UI 线程。

根据以上两点，我们也简单封装一下，在此只是举个例子。如果想要使用 `OkHttp` 封装的开源库，笔者推荐使用 `OkHttpFinal`。

首先写一个抽象类用于请求回调：

```

public abstract class ResultCallback
{

```

```

    public abstract void onError(Request request, Exception e);
    public abstract void onResponse(Response response) throws IOException;
}

```

接下来封装 **OkHttp**，这里只实现了异步 GET 请求，如下所示：

```

public class OkHttpEngine {
    private static OkHttpEngine mInstance;
    private OkHttpClient mOkHttpClient;
    private Handler mHandler;

    public static OkHttpEngine getInstance(Context context) {
        if (mInstance == null) {
            synchronized (OkHttpEngine.class) {
                if (mInstance == null) {
                    mInstance = new OkHttpEngine(context);
                }
            }
        }
        return mInstance;
    }

    private OkHttpEngine(Context context) {
        File sdcache = context.getExternalCacheDir();
        int cacheSize = 10 * 1024 * 1024;
        OkHttpClient.Builder builder = new OkHttpClient.Builder()
            .connectTimeout(15, TimeUnit.SECONDS)
            .writeTimeout(20, TimeUnit.SECONDS)
            .readTimeout(20, TimeUnit.SECONDS)
            .cache(new Cache(sdcache.getAbsoluteFile(), cacheSize));
        mOkHttpClient=builder.build();
        mHandler = new Handler();
    }

    /**
     * 异步 GET 请求
     * @param url
     * @param callback
     */
    public void getAsynHttp(String url, ResultCallback callback) {

        final Request request = new Request.Builder()
            .url(url)

```

```

        .build();
        Call call = mOkHttpClient.newCall(request);
        dealResult(call, callback);
    }

    private void dealResult(Call call, final ResultCallback callback) {
        call.enqueue(new Callback() {
            @Override
            public void onFailure(Call call, IOException e) {
                sendFailedCallback(call.request(), e, callback);
            }
            @Override
            public void onResponse(Call call, Response response) throws
                IOException {
                sendSuccessCallback(response, callback);
            }
        });
        private void sendSuccessCallback(final Response object, final
            ResultCallback callback) {
            mHandler.post(new Runnable() {
                @Override
                public void run() {
                    if (callback != null) {
                        try {
                            callback.onResponse(object);
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }
            });
        }
        private void sendFailedCallback(final Request request, final
            Exception e, final ResultCallback callback) {
            mHandler.post(new Runnable() {
                @Override
                public void run() {
                    if (callback != null)
                        callback.onError(request, e);
                }
            });
        }
    }
}

```


其原理就是写一个双重检查模式的单例，在开始创建的时候配置好 `OkHttpClient`，并创建 `Handler`，在请求网络的时候用 `Handler` 将请求的结果回调给 UI 线程。当想要请求网路时就调用 `OkHttpEngine` 的 `getAsynHttp` 方法，如下所示：

```
OkHttpEngine.getInstance(MainActivity.this).getAsynHttp("http://www.
baidu.com", new ResultCallback() {
    @Override
    public void onError(Request request, Exception e) {

    }
    @Override
    public void onResponse(Response response) throws IOException{
        String str = response.body().string();
        Log.d(TAG, str);
        Toast.makeText(getApplicationContext(), "请求成功", Toast.
            LENGTH_SHORT).show();
    }
});
```

经过修改，达到了目的，无须每次都要写重复的代码，另外回调 `onResponse` 方法是运行在 UI 线程的。

5.6.2 源码解析 OkHttpClient

本节主要通过源码来了解 `OkHttp` 的请求网络流程和复用连接池。现在首先来学习 `OkHttp` 的请求网络流程。

1. OkHttpClient 的请求网络流程

(1) 从请求处理开始分析

当我们要请求网络的时候需要用 `OkHttpClient.newCall(request)` 进行 `execute` 或者 `enqueue` 操作；当调用 `newCall` 方法时，会调用如下代码：

```
@Override
public Call newCall(Request request) {
    return new RealCall(this, request);
}
```

其实际返回的是一个 `RealCall` 类。我们调用 `enqueue` 异步请求网络实际上是调用了 `RealCall` 的 `enqueue` 方法。查看 `RealCall` 的 `enqueue` 方法，如下所示：

```
void enqueue(Callback responseCallback, boolean forWebSocket) {
    synchronized (this) {
```

```

        if (executed) throw new IllegalStateException("Already Executed");
        executed = true;
    }
    client.dispatcher().enqueue(new AsyncCall(responseCallback, forWebSocket));
}

```

可以看到最终的请求是 dispatcher 来完成的，接下来就开始分析 dispatcher。

(2) Dispatcher 任务调度

Dispatcher 主要用于控制并发的请求，它主要维护了以下变量：

```

/** 最大并发请求数*/
private int maxRequests = 64;
/** 每个主机的最大请求数*/
private int maxRequestsPerHost = 5;
/** 消费者线程池 */
private ExecutorService executorService;
/** 将要运行的异步请求队列 */
private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();
/**正在运行的异步请求队列 */
private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>();
/** 正在运行的同步请求队列 */
private final Deque<RealCall> runningSyncCalls = new ArrayDeque<>();

```

接下来看看 Dispatcher 的构造方法，如下所示：

```

public Dispatcher(ExecutorService executorService) {
    this.executorService = executorService;
}
public Dispatcher() {
}
public synchronized ExecutorService executorService() {
    if (executorService == null) {
        executorService = new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60,
            TimeUnit.SECONDS, new SynchronousQueue<Runnable>(), Util.threadFactory(
                "OkHttp Dispatcher", false));
    }
    return executorService;
}

```

Dispatcher 有两个构造方法，可以使用自己设定的线程池。如果没有设定线程池，则会在请求网络前自己创建默认线程池。这个线程池类似于 `CachedThreadPool`，比较适合执行大量的耗时比较少的任务。前面讲过，当调用 `RealCall` 的 `enqueue` 方法时，实际上是调用了 Dispatcher 的 `enqueue` 方法，它的代码如下所示：

```
synchronized void enqueue(AsyncCall call) {
    if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call)
        < maxRequestsPerHost) {
        runningAsyncCalls.add(call);
        executorService().execute(call);
    } else {
        readyAsyncCalls.add(call);
    }
}
```

当正在运行的异步请求队列中的数量小于 64 并且正在运行的请求主机数小于 5 时, 把请求加载到 `runningAsyncCalls` 中并在线程池中执行, 否则就加入到 `readyAsyncCalls` 中进行缓存等待。线程池中传进来的参数是 `AsyncCall`, 它是 `RealCall` 的内部类, 其内部也实现了 `execute` 方法, 如下所示:

```
protected void execute() {
    boolean signalledCallback = false;
    try {
        ...
    } catch (IOException e) {
        ...
    } finally {
        client.dispatcher().finished(this); //1
    }
}
```

在上面代码注释 1 处, 无论这个请求的结果如何, 都会执行 `client.dispatcher().finished(this)`, `finished` 方法如下所示:

```
synchronized void finished(AsyncCall call) {
    if (!runningAsyncCalls.remove(call)) throw new AssertionError("AsyncCall
        wasn't running!");
    promoteCalls();
}
```

`finished` 方法将此次请求从 `runningAsyncCalls` 移除后还执行了 `promoteCalls` 方法:

```
private void promoteCalls() {
    if (runningAsyncCalls.size() >= maxRequests) return;
    if (readyAsyncCalls.isEmpty()) return;
    for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {
        AsyncCall call = i.next();

        if (runningCallsForHost(call) < maxRequestsPerHost) {
            i.remove();
        }
    }
}
```

```

        runningAsyncCalls.add(call);
        executorService().execute(call);
    }
    if (runningAsyncCalls.size() >= maxRequests) return;
}
}

```

最关键的一点就是从 `readyAsyncCalls` 取出下一个请求, 加入 `runningAsyncCalls` 中并交由线程池处理。好了, 让我们再回到上面 `AsyncCall` 的 `execute` 方法:

```

@Override
protected void execute() {
    boolean signalledCallback = false;
    try {
        Response response = getResponseWithInterceptorChain(forWebSocket); //1
        if (canceled) {
            signalledCallback = true;
            responseCallback.onFailure(RealCall.this, new IOException("Canceled"));
        } else {
            signalledCallback = true;
            responseCallback.onResponse(RealCall.this, response);
        }
    } catch (IOException e) {
        if (signalledCallback) {
            logger.log(Level.INFO, "Callback failure for " + toLoggableString(), e);
        } else {
            responseCallback.onFailure(RealCall.this, e);
        }
    } finally {
        client.dispatcher().finished(this);
    }
}
}

```

在上面代码注释 1 处, `getResponseWithInterceptorChain` 方法返回了 `Response`, 很明显这是在请求网络。

(3) Interceptor 拦截器

接下来查看 `getResponseWithInterceptorChain` 方法, 如下所示:

```

private Response getResponseWithInterceptorChain(boolean forWebSocket)
throws IOException {
    Interceptor.Chain chain = new ApplicationInterceptorChain(0, originalRequest,
        forWebSocket);
    return chain.proceed(originalRequest);
}

```

`getResponseWithInterceptorChain` 方法中创建了 `ApplicationInterceptorChain`，它是一个拦截器链。这个类也是 `RealCall` 的内部类，接下来执行了它的 `proceed` 方法：

```
public Response proceed(Request request) throws IOException {
    if (index < client.interceptors().size()) {
        Interceptor.Chain chain = new ApplicationInterceptorChain(index + 1,
            request, forWebSocket);
        //从拦截器列表中取出拦截器
        Interceptor interceptor = client.interceptors().get(index);
        Response interceptedResponse = interceptor.intercept(chain); //1
        if (interceptedResponse == null) {
            throw new NullPointerException("application interceptor " + interceptor
                + " returned null");
        }
        return interceptedResponse;
    }
    return getResponse(request, forWebSocket);
}
```

`proceed` 方法每次从拦截器列表中取出拦截器。当存在多个拦截器时都会在上面代码注释 1 处阻塞，并等待下一个拦截器的调用返回。下面分别以拦截器链中有一个、两个拦截器的场景加以模拟，如图 5-14 所示。

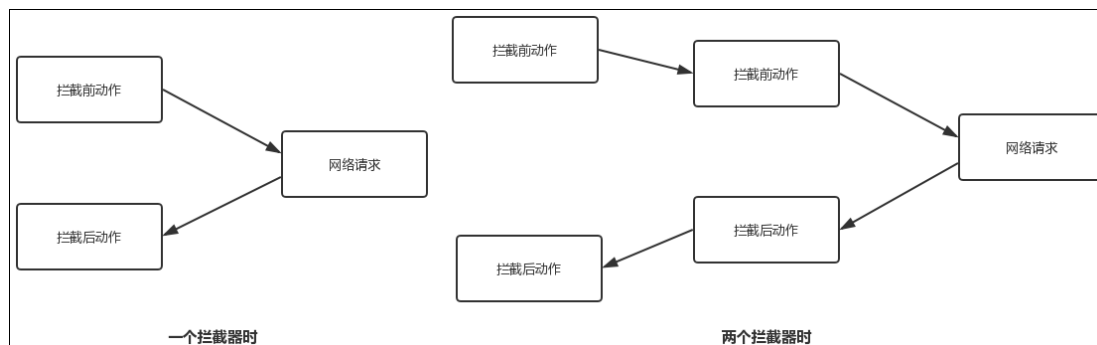


图 5-14 拦截器的场景模拟

拦截器是一种能够监控、重写、重试调用的机制。通常情况下，拦截器用来添加、移除、转换请求和响应的头部信息。比如将域名替换为 IP 地址，在请求头中添加 `host` 属性；也可以添加我们应用中的一些公共参数，比如设备 `id`、版本号，等等。回到代码上来，查看最后一行 `return getResponse(request, forWebSocket)`，如果没有更多拦截器的话，就会执行网络请求。现在查看 `getResponse` 方法做了什么，代码如下所示：

```
Response getResponse(Request request, boolean forWebSocket) throws
IOException {
    ...
    engine = new HttpEngine(client, request, false, false, forWebSocket,
```

```

    null, null, null);
    int followUpCount = 0;
    while (true) {
        if (canceled) {
            engine.releaseStreamAllocation();
            throw new IOException("Canceled");
        }
        boolean releaseConnection = true;
        try {
            engine.sendRequest();
            engine.readResponse();
            releaseConnection = false;
        } catch (RequestException e) {
            throw e.getCause();
        } catch (RouteException e) {
            ...
        }
    }
}

```

getResponse 方法比较长, 这里省略了一些代码, 在此可以看到创建了 HttpEngine 类并且调用了 HttpEngine 的 sendRequest 方法和 readResponse 方法。

(4) 缓存策略

现在先来查看 HttpEngine 的 sendRequest 方法, 如下所示:

```

public void sendRequest() throws RequestException, RouteException, IOException {
    if (cacheStrategy != null) return; // Already sent.
    if (httpStream != null) throw new IllegalStateException();
    Request request = networkRequest(userRequest);
    //获取 client 中的 Cache, 同时 Cache 在初始化时会读取缓存目录中曾经请求过的所有信息
    InternalCache responseCache = Internal.instance.internalCache(client);
    Response cacheCandidate = responseCache != null
        ? responseCache.get(request): null; //1
    long now = System.currentTimeMillis();
    cacheStrategy = new CacheStrategy.Factory(now, request, cacheCandidate).
        get();
    //网络请求
    networkRequest = cacheStrategy.networkRequest;
    //缓存的响应
    cacheResponse = cacheStrategy.cacheResponse;
    if (responseCache != null) {
        //记录当前请求是网络发起还是缓存发起
        responseCache.trackResponse(cacheStrategy);
    }
}

```

```

        if (cacheCandidate != null && cacheResponse == null) {
            closeQuietly(cacheCandidate.body());
        }
        // 不进行网络请求并且缓存不存在或者过期，则返回 504 错误
        if (networkRequest == null && cacheResponse == null) {
            userResponse = new Response.Builder()
                .request(userRequest)
                .priorResponse(stripBody(priorResponse))
                .protocol(Protocol.HTTP_1_1)
                .code(504)
                .message("Unsatisfiable Request (only-if-cached)")
                .body(EMPTY_BODY)
                .build();
            return;
        }
        // 不进行网络请求而且缓存可以使用，则直接返回缓存
        if (networkRequest == null) {
            userResponse = cacheResponse.newBuilder()
                .request(userRequest)
                .priorResponse(stripBody(priorResponse))
                .cacheResponse(stripBody(cacheResponse))
                .build();
            userResponse = unzip(userResponse);
            return;
        }
        // 需要访问网络时
        boolean success = false;
        try {
            httpStream = connect();
            httpStream.setHttpEngine(this);
            ...
        }
    }
}

```

上面的代码显然是在发送请求，但是最主要的是做了缓存的策略。上面代码注释 1 处的 `cacheCandidate` 是上次与服务器交互时缓存的 `Response`，这里的缓存均基于 `Map`。key 是请求中 url 的 md5，value 是在文件中查询到的缓存，页面置换基于 LRU 算法，我们现在只需要知道 `cacheCandidate` 是一个可以读取缓存 Header 的 `Response` 即可。根据 `cacheStrategy` 的处理得到了 `networkRequest` 和 `cacheResponse` 这两个值，根据这两个值的数据是否为 `null` 来进行进一步的处理。在 `networkRequest` 和 `cacheResponse` 都为 `null` 的情况下，也就是不进行网络请求并且缓存不存在或者过期，这时返回 504 错误；当 `networkRequest` 为 `null` 时也就是不进行网络请求，如果缓存可以使用时则直接返回缓存，其他情况则请求网络。

接下来查看 `HttpEngine` 的 `readResponse` 方法，如下所示：

```
public void readResponse() throws IOException {
    ...
    else{
        //读取网络响应
        networkResponse = readNetworkResponse();
    }
    receiveHeaders(networkResponse.headers());
    if (cacheResponse != null) {
        //检查缓存是否可用。如果可用，就用当前缓存的 Response，关闭网络连接，释放连接
        if (validate(cacheResponse, networkResponse)) { //1
            userResponse = cacheResponse.newBuilder()
                .request(userRequest)
                .priorResponse(stripBody(priorResponse))
                .headers(combine(cacheResponse.headers(), networkResponse.
                    headers()))
                .cacheResponse(stripBody(cacheResponse))
                .networkResponse(stripBody(networkResponse))
                .build();
            networkResponse.body().close();
            releaseStreamAllocation();
            InternalCache responseCache = Internal.instance.internalCache(client);
            responseCache.trackConditionalCacheHit();
            responseCache.update(cacheResponse, stripBody(userResponse));
            userResponse = unzip(userResponse);
            return;
        } else {
            closeQuietly(cacheResponse.body());
        }
    }
    userResponse = networkResponse.newBuilder()
        .request(userRequest)
        .priorResponse(stripBody(priorResponse))
        .cacheResponse(stripBody(cacheResponse))
        .networkResponse(stripBody(networkResponse))
        .build();

    if (hasBody(userResponse)) {
        maybeCache();
        userResponse = unzip(cacheWritingResponse(storeRequest, userResponse));
    }
}
```

这个方法主要用来解析 HTTP 响应报头。如果有缓存并且可用，则用缓存的数据并更新缓

存，否则就用网络请求返回的数据。再来查看上面代码注释 1 处的 `validate` 方法是如何判断缓存是否可用的：

```
private static boolean validate(Response cached, Response network) {
    //如果服务器返回 304，则缓存有效
    if (network.code() == HTTP_NOT_MODIFIED) {
        return true;
    }
    //通过缓存和网络请求响应中的 Last-Modified 来计算是否是最新数据。如果是，则缓存有效
    Date lastModified = cached.headers().getDate("Last-Modified");
    if (lastModified != null) {
        Date networkLastModified = network.headers().getDate("Last-Modified");
        if (networkLastModified != null
            && networkLastModified.getTime() < lastModified.getTime()) {
            return true;
        }
    }
    return false;
}
```

如果缓存是有效的，则返回 304 Not Modified，否则直接返回 body。如果缓存过期或者强制放弃缓存，则缓存策略全部交给服务器判断，客户端只需要发送条件 GET 请求即可。条件 GET 请求有两种方式：一种是 Last-Modified-Date，另一种是 ETag。这里采用了 Last-Modified-Date，通过缓存和网络请求响应中的 Last-Modified 来计算是否是最新数据。如果是，则缓存有效。

(5) 失败重连

最后我们再回到 `RealCall` 的 `getResponse` 方法，如下所示：

```
Response getResponse(Request request, boolean forWebSocket) throws
IOException {
    ...
    boolean releaseConnection = true;
    try {
        engine.sendRequest();
        engine.readResponse();
        releaseConnection = false;
    } catch (RequestException e) {
        throw e.getCause();
    } catch (RouteException e) {
        HttpEngine retryEngine = engine.recover(e.getLastConnectException(),
            null); //1
        if (retryEngine != null) {
            releaseConnection = false;
        }
    }
}
```

```

        engine = retryEngine;
        continue;
    }
    throw e.getLastConnectException();
} catch (IOException e) {
    HttpEngine retryEngine = engine.recover(e, null); //2
    if (retryEngine != null) {
        releaseConnection = false;
        engine = retryEngine;
        continue;
    }
    throw e;
} finally {
    if (releaseConnection) {
        StreamAllocation streamAllocation = engine.close();
        streamAllocation.release();
    }
}
...
engine = new HttpEngine(client, request, false, false, forWebSocket,
    streamAllocation, null, response);

}
}

```

在上面代码注释 1 和注释 2 处, 当发生 `IOException` 或者 `RouteException` 时都会执行 `HttpEngine` 的 `recover` 方法, 它的代码如下所示:

```

public HttpEngine recover(IOException e, Sink requestBodyOut) {
    if (!streamAllocation.recover(e, requestBodyOut)) {
        return null;
    }
    if (!client.retryOnConnectionFailure()) {
        return null;
    }
    StreamAllocation streamAllocation = close();
    return new HttpEngine(client, userRequest, bufferRequestBody,
        callerWritesRequestBody, forWebSocket, streamAllocation,
        (RetryableSink) requestBodyOut, priorResponse);
}

```

通过最后一行可以看到, 其就是重新创建了 `HttpEngine` 并返回, 用来完成重连。到这里 `OkHttp` 请求网络的流程基本上讲完了, 图 5-15 是 `OkHttp` 的请求流程图。

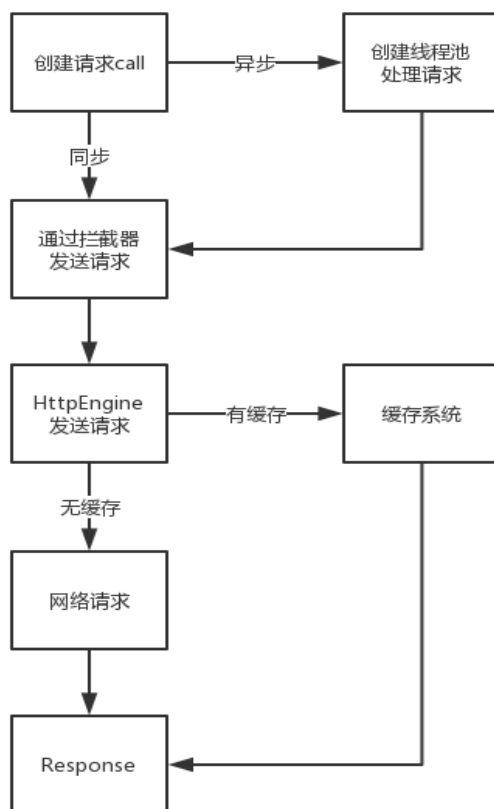


图 5-15 OkHttp 的请求流程图

2. OkHttp 的复用连接池

在 5.2 节中我们讲到了 TCP 的 3 次握手与 4 次挥手。为了解决 TCP 握手和挥手的效率问题，HTTP 有一种叫作 *keepalive connections* 的机制；而 OkHttp 支持 5 个并发 socket 连接，默认 *keepAlive* 时间为 5 分钟。接下来我们学习 OkHttp 是怎么复用连接的。

(1) 主要变量与构造方法

连接池的类位于 `okhttp3.ConnectionPool`，它的主要变量如下所示：

```

private static final Executor executor = new ThreadPoolExecutor(0 ,
    Integer.MAX_VALUE , 60L , TimeUnit.SECONDS,
    new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp
    ConnectionPool", true));
//空闲的 socket 最大连接数
private final int maxIdleConnections;
//socket 的 keepAlive 时间
private final long keepAliveDurationNs;
private final Deque<RealConnection> connections = new ArrayDeque<>();

```

```
final RouteDatabase routeDatabase = new RouteDatabase();
boolean cleanupRunning;
```

主要的变量有必要说明一下。

- **executor** 线程池，类似于 **CachedThreadPool**，需要注意的是这种线程池的工作队列采用了没有容量的 **SynchronousQueue**。
- **Deque**，双向队列，双端队列同时具有队列和栈性质，经常在缓存中被使用，里面维护了 **RealConnection** 也就是 **socket** 物理连接的包装。
- **RouteDatabase**，它用来记录连接失败的路线名单，当连接失败的时候就会把失败的线路加进去。
- **ConnectionPool** 的构造方法如下所示：

```
public ConnectionPool() {
    this(5, 5, TimeUnit.MINUTES);
}
public ConnectionPool(int maxIdleConnections, long keepAliveDuration,
    TimeUnit timeUnit) {
    this.maxIdleConnections = maxIdleConnections;
    this.keepAliveDurationNs = timeUnit.toNanos(keepAliveDuration);
    if (keepAliveDuration <= 0) {
        throw new IllegalArgumentException("keepAliveDuration <= 0: " +
            keepAliveDuration);
    }
}
```

通过构造方法可以看出 **ConnectionPool** 默认空闲的 **socket** 最大连接数为 5 个，**socket** 的 **keepAlive** 时间为 5 分钟。**ConnectionPool** 是在 **OkHttpClient** 实例化时创建的：

```
public OkHttpClient() {
    this(new Builder());
}
public Builder() {
    dispatcher = new Dispatcher();
    ...
    connectionPool = new ConnectionPool();
    ...
}
```

(2) 缓存操作

ConnectionPool 提供对 **Deque<RealConnection>** 进行操作的方法分别为 **put**、**get**、**connectionBecameIdle** 和 **evictAll** 这几个操作，分别对应放入连接、获取连接、移除连接和移除所有连接操作。这里我们举例说明 **put** 和 **get** 操作。

```

void put(RealConnection connection) {
    assert (Thread.holdsLock(this));
    if (!cleanupRunning) {
        cleanupRunning = true;
        executor.execute(cleanupRunnable);
    }
    connections.add(connection);
}

```

在添加到 Deque 之前首先要清理空闲的线程，这个后面会讲到。再来看 get 操作：

```

RealConnection get(Address address, StreamAllocation streamAllocation) {
    assert (Thread.holdsLock(this));
    for (RealConnection connection : connections) {
        if (connection.allocations.size() < connection.allocationLimit
            && address.equals(connection.route().address)
            && !connection.noNewStreams) {
            streamAllocation.acquire(connection);
            return connection;
        }
    }
    return null;
}

```

遍历 connections 缓存列表。当某个连接计数的次数小于限制的大小，并且 request 的地址和缓存列表中此连接的地址完全匹配时，则直接复用缓存列表中的 connection 作为 request 的连接。

(3) 自动回收连接

OkHttp 是根据 StreamAllocation 引用计数是否为 0 来实现自动回收连接的。我们在 put 操作前首先要调用 executor.execute(cleanupRunnable)来清理闲置的线程。我们来查看 cleanupRunnable 到底做了什么：

```

private final Runnable cleanupRunnable = new Runnable() {
    @Override
    public void run() {
        while (true) {
            long waitNanos = cleanup(System.nanoTime());
            if (waitNanos == -1) return;
            if (waitNanos > 0) {
                long waitMillis = waitNanos / 1000000L;
                waitNanos -= (waitMillis * 1000000L);
                synchronized (ConnectionPool.this) {

```

```

        try {
            ConnectionPool.this.wait(waitMillis, (int) waitNanos);
        } catch (InterruptedException ignored) {
        }
    }
}
}
}
};

```

线程不断地调用 `cleanup` 方法来进行清理，并返回下次需要清理的间隔时间，然后调用 `wait` 方法进行等待以释放锁与时间片。当等待时间到了后，再次进行清理，并返回下次要清理的间隔时间。如此循环下去。接下来看看 `cleanup` 方法，如下所示：

```

long cleanup(long now) {
    int inUseConnectionCount = 0;
    int idleConnectionCount = 0;
    RealConnection longestIdleConnection = null;
    long longestIdleDurationNs = Long.MIN_VALUE;
    synchronized (this) {
        for (Iterator<RealConnection> i = connections.iterator(); i.hasNext(); ) {
            RealConnection connection = i.next();
            if (pruneAndGetAllocationCount(connection, now) > 0) { //1
                inUseConnectionCount++;
                continue;
            }
            idleConnectionCount++;
            long idleDurationNs = now - connection.idleAtNanos;
            if (idleDurationNs > longestIdleDurationNs) {
                longestIdleDurationNs = idleDurationNs;
                longestIdleConnection = connection;
            }
        }

        if (longestIdleDurationNs >= this.keepAliveDurationNs
            || idleConnectionCount > this.maxIdleConnections) { //2
            connections.remove(longestIdleConnection);
        } else if (idleConnectionCount > 0) {
            return keepAliveDurationNs - longestIdleDurationNs;
        } else if (inUseConnectionCount > 0) {
            return keepAliveDurationNs;
        } else {
            cleanupRunning = false;
            return -1; //3
        }
    }
}

```

```

    }
    closeQuietly(longestIdleConnection.socket());
    return 0;
}

```

`cleanup` 方法所做的事情简单总结就是，根据连接中的引用计数来计算空闲连接数和活跃连接数，然后标记出空闲的连接。在上面代码注释 2 处，如果空闲连接 `keepAlive` 时间超过 5 分钟，或者空闲连接数超过 5 个，则从 `Deque` 中移除此连接。接下来根据空闲连接或者活跃连接来返回下次需要清理的时间数：如果空闲连接大于 0，则返回此连接即将到期的时间；如果都是活跃连接并且大于 0，则返回默认的 `keepAlive` 时间 5 分钟；在注释 3 处，如果没有任何连接，则跳出循环并返回 -1。在上述代码的注释 1 处，通过 `pruneAndGetAllocationCount` 方法来判断连接是否闲置。如果 `pruneAndGetAllocationCount` 方法的返回值大于 0 则是空闲连接，否则就是活跃连接。接下来查看 `pruneAndGetAllocationCount` 方法，如下所示：

```

private int pruneAndGetAllocationCount(RealConnection connection, long now) {
    List<Reference<StreamAllocation>> references = connection.allocations;
    for (int i = 0; i < references.size(); ) {
        Reference<StreamAllocation> reference = references.get(i);
        if (reference.get() != null) {
            i++;
            continue;
        }
        Internal.logger.warning("A connection to " + connection.route().address().url()
            + " was leaked. Did you forget to close a response body?");
        references.remove(i);
        connection.noNewStreams = true;
        if (references.isEmpty()) { // 1
            connection.idleAtNanos = now - keepAliveDurationNs;
            return 0;
        }
    }
    return references.size();
}

```

`pruneAndGetAllocationCount` 方法首先遍历传进来的 `RealConnection` 的 `StreamAllocation` 列表。如果 `StreamAllocation` 被使用，则接着遍历下一个 `StreamAllocation`；如果 `StreamAllocation` 未被使用，则从列表中移除。在上面代码注释 1 处，如果列表为空，则说明此连接没有引用了，这时返回 0，表示此连接是空闲连接；否则就返回非 0 的数，表示此连接是活跃连接。那么 `StreamAllocation` 是什么，怎样才能判断 `StreamAllocation` 使用与否？读者可接着往下看。

(4) 引用计数

在 `OkHttp` 的高层代码调用中，使用了类似于引用计数的方式跟踪 `socket` 流的调用。这里的

计数对象是 `StreamAllocation`，它被反复执行 `acquire` 与 `release` 操作，这两个方法其实是在改变 `RealConnection` 中的 `List<Reference<StreamAllocation>>` 的大小。`StreamAllocation` 的 `acquire` 方法和 `release` 方法如下所示：

```
public void acquire(RealConnection connection) {
    connection.allocations.add(new WeakReference<>(this));
}

private void release(RealConnection connection) {
    for (int i = 0, size = connection.allocations.size(); i < size; i++) {
        Reference<StreamAllocation> reference = connection.allocations.get(i);
        if (reference.get() == this) {
            connection.allocations.remove(i);
            return;
        }
    }
    throw new IllegalStateException();
}
```

`RealConnection` 是 socket 物理连接的包装，它里面维护了 `List<Reference<StreamAllocation>>` 的引用。`List` 中 `StreamAllocation` 的数量也就是 socket 被引用的计数。如果计数为 0，则说明此连接没有被使用，也就是空闲的，需要通过下文的算法实现回收；如果计数不为 0，则表示上层代码仍然在引用，就无须关闭连接。

(5) 小结

可以看出连接池复用的核心就是用 `Deque<RealConnection>` 来存储连接，通过 `put`、`get`、`connectionBecameIdle` 和 `evictAll` 几个操作来对 `Deque` 进行操作，另外通过判断连接中的计数对象 `StreamAllocation` 来进行自动回收连接。

5.7 解析 Retrofit

Retrofit 是 Square 公司开发的一款针对 Android 网络请求的框架，Retrofit 底层是基于 `OkHttp` 实现的。与其他网络框架不同的是，它更多使用运行时注解的方式提供功能。

5.7.1 Retrofit 基本用法

1. 使用前的准备工作

首先配置 `build.gradle`，如下所示：

```
dependencies {
    ...
    compile 'com.squareup.retrofit2:retrofit:2.1.0'
    compile 'com.squareup.retrofit2:converter-gson:2.1.0'
}
```


最后一行是为了增加支持返回值为 Gson 类型数据所需要添加的依赖包。如果想增加其他类型数据的支持，可以添加如下依赖包。

- Scalars (primitives, boxed, and String): com.squareup.retrofit2:converter-scalars;
- Jackson: com.squareup.retrofit2:converter-jackson;
- Moshi: com.squareup.retrofit2:converter-moshi;
- Protobuf: com.squareup.retrofit2:converter-protobuf;
- Wire: com.squareup.retrofit2:converter-wire;
- Simple XML: com.squareup.retrofit2:converter-simplexml。

当然，别忘了在 manifest 中加入访问网络的权限。

2. Retrofit 的注解分类

Retrofit 与其他请求框架不同的是，它使用了注解。Retrofit 的注解分为三大类，分别是 HTTP 请求方法注解、标记类注解和参数类注解。其中，HTTP 请求方法注解有 8 种，它们是 GET、POST、PUT、DELETE、HEAD、PATCH、OPTIONS 和 HTTP。其前 7 种分别对应 HTTP 的请求方法；HTTP 则可以替换以上 7 种，也可以扩展请求方法。标记类注解有 3 种，它们是 FormUrlEncoded、Multipart、Streaming。FormUrlEncoded 和 Multipart 后面会讲到；Streaming 代表响应的数据以流的形式返回，如果不使用它，则默认会把全部数据加载到内存，所以下载大文件时需要加上这个注解。参数类注解有 Header、Headers、Body、Path、Field、FieldMap、Part、PartMap、Query 和 QueryMap 等，下面会介绍几种参数类注解的用法。

3. GET 请求访问网络

首先实现用 GET 请求方式来访问网络，这里仍旧访问淘宝 IP 库。实体类的编写在 5.5.1 节中已经做过，这里就不赘述了。首先编写请求网络接口，如下所示：

```
public interface IpService {
    @GET("getIpInfo.php?ip=59.108.54.37")
    Call<IpModel> getIpMsg();
}
```

Retrofit 提供的请求方式注解有 @GET 和 @POST 等，分别代表 GET 请求和 POST 请求，我们在这里用的是 GET 请求，访问的地址是 "getIpInfo.php?ip=59.108.54.37"。另外定义了 getIpMsg 方法，这个方法返回 Call<IpModel> 类型的参数。接下来创建 Retrofit，并创建接口文件，代码如下所示：

```
String url = "http://ip.taobao.com/service/";
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl(url)
    //增加返回值为Json的支持
    .addConverterFactory(GsonConverterFactory.create())
```

```

        .build();
        IpService ipService = retrofit.create(IpService.class);
        Call<IpModel>call=ipService.getIpMsg();

```

Retrofit 是通过建造者模式构建出来的。请求 URL 是拼接而成的，它是由 baseUrl 传入的 URL 加上请求网络接口的 @GET("getIpInfo.php?ip=59.108.54.37") 中的 URL 拼接而成的。接下来用 Retrofit 动态代理获取到之前定义的接口，并调用该接口定义的 getIpMsg 方法得到 Call 对象。接下来用 Call 请求网络并处理回调，代码如下所示：

```

call.enqueue(new Callback<IpModel>() {
    @Override
    public void onResponse(Call<IpModel> call, Response<IpModel> response) {
        String country= response.body().getData().getCountry();
        Toast.makeText(getApplicationContext(),country,Toast.
            LENGTH_SHORT).show();
    }
    @Override
    public void onFailure(Call<IpModel> call, Throwable t) {
    }
});

```

这里是异步请求网络，回调的 Callback 是运行在 UI 线程的。得到返回的 Response 后将返回数据的 country 字段用 Toast 显示出来。如果想同步请求网络，请使用 call.execute()；如果想中断网络请求，则可以使用 call.cancel()。

动态配置 URL 地址：@Path

Retrofit 提供了很多请求参数注解，这使得请求网路时更加便捷。其中，@Path 用来动态地配置 URL 地址。请求网络接口代码如下所示：

```

public interface IpServiceForPath {
    @GET("{path}/getIpInfo.php?ip=59.108.54.37")
    Call<IpModel> getIpMsg(@Path("path") String path);
}

```

在 GET 注解中包含了 {path}，它对应着 @Path 注解中的 "path"，而用来替换 {path} 的正是需要传入的 "String path" 的值。请求网络的代码如下所示：

```

String url = "http://ip.taobao.com/";
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl(url)
    .addConverterFactory(GsonConverterFactory.create())
    .build();
IpServiceForPath ipService = retrofit.create(IpServiceForPath.class);
Call<IpModel>call=ipService.getIpMsg("service");//1
call.enqueue(new Callback<IpModel>() {

```

```

@Override
public void onResponse(Call<IpModel> call, Response<IpModel>
response) {
    String country= response.body().getData().getCountry();
    Toast.makeText(getApplicationContext(),country,Toast.
LENGTH_SHORT).show();
}
@Override
public void onFailure(Call<IpModel> call, Throwable t) {
}
});

```

在上面代码注释 1 处，传入 "service" 来替换 @GET 注解中的 {path} 的值。

动态指定查询条件：@Query

之前的例子就是为了查询 ip 的地址位置，每次查询更换不同的 ip 就可以了，可以用 @Query 来动态地指定 ip 的值。请求网络接口的代码如下所示：

```

public interface IpServiceForQuery{
    @GET("getIpInfo.php")
    Call<IpModel> getIpMsg(@Query("ip")String ip);
}

```

请求网络时，只需要传入想要查询的 ip 值就可以了。

动态指定查询条件组：@QueryMap

在网络请求中一般为了更精确地查找到我们所需要的数据，需要传入很多查询参数。如果用 @Query 会比较麻烦，这时我们可以采用 @QueryMap，将所有的参数集成在一个 Map 中统一传递，如下所示：

```

public interface IpServiceForQueryMap {
    @GET("getIpInfo.php")
    Call<IpModel> getIpMsg(@QueryMap Map<String, String> options);
}

```

4. POST 请求访问网络

传输数据类型为键值对：@Field

传输数据类型为键值对，这是我们最常用的 POST 请求数据类型，淘宝 IP 库支持数据类型为键值对的 POST 请求。请求网络接口的代码如下所示：

```

public interface IpServiceForPost {
    @FormUrlEncoded
    @POST("getIpInfo.php")
    Call<IpModel> getIpMsg(@Field("ip") String first);
}

```

首先用 `@FormUrlEncoded` 注解来标明这是一个表单请求，然后在 `getIpMsg` 方法中使用 `@Field` 注解来标示所对应的 `String` 类型数据的键，从而组成一组键值对进行传递。请求网络的代码如下所示：

```
String url = "http://ip.taobao.com/service/";
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl(url)
    .addConverterFactory(GsonConverterFactory.create())
    .build();
IpServiceForPost ipService = retrofit.create(IpServiceForPost.class);
Call<IpModel> call = ipService.getIpMsg("59.108.54.37");
call.enqueue(new Callback<IpModel>() {
    @Override
    public void onResponse(Call<IpModel> call, Response<IpModel> response) {
        String country = response.body().getData().getCountry();
        Toast.makeText(getApplicationContext(), country, Toast.LENGTH_SHORT).show();
    }
    @Override
    public void onFailure(Call<IpModel> call, Throwable t) {
    }
});
```

传输数据类型 JSON 字符串：@Body

我们也可以用 `POST` 方式将 `JSON` 字符串作为请求体发送到服务器，请求网络接口的代码如下所示：

```
public interface IpServiceForPostBody {
    @POST("getIpInfo.php")
    Call<IpModel> getIpMsg(@Body Ip ip);
}
```

用 `@Body` 这个注解标识参数对象即可，`Retrofit` 会将 `Ip` 对象转换为字符串：

```
public class Ip {
    private String ip;
    public Ip(String ip) {
        this.ip = ip;
    }
}
```

请求网络的代码基本上都是一致的：

...

```
IpServiceForPostBody ipService = retrofit.create (IpServiceForPostBody.class);
Call<IpModel>call=ipService.getIpMsg(new Ip(ip));
...
```

运行程序用 Fiddler 抓包，请求报文如图 5-16 所示。

```
POST http://ip.taobao.com/service/getIpInfo.php HTTP/1.1
Content-Type: application/json; charset=UTF-8
Content-Length: 21
Host: ip.taobao.com
Connection: Keep-Alive
Accept-Encoding: gzip
User-Agent: okhttp/3.3.0
{"ip": "59.108.54.37"}
```

图 5-16 请求报文

在此可以看到请求数据是一个 JSON 字符串，因为淘宝 IP 库并不支持此类型，所以不会返回我们需要的地理信息数据。

单个文件上传：@Part

```
public interface UploadFileForPart {
    @Multipart
    @POST("user/photo")
    Call<User> updateUser(@Part MultipartBody.Part photo, @Part("description")
        RequestBody description);
}
```

Multipart 注解表示允许多个 @Part。updateUser 方法的第一个参数是准备上传的图片文件，使用了 MultipartBody.Part 类型；另一个参数是 RequestBody 类型，它用来传递简单的键值对。请求网络代码如下所示：

```
...
File file = new File(Environment.getExternalStorageDirectory(), "wangshu.png");
RequestBody photoRequestBody = RequestBody.create(MediaType.parse("image/png"), file);
MultipartBody.Part photo = MultipartBody.Part.createFormData("photos", "wangshu.png", photoRequestBody);
UploadFileForPart uploadFile = retrofit.create(UploadFileForPart.class);
Call<User> call = uploadFile.updateUser(photo, RequestBody.create(null, "wangshu"));
...
```

多个文件上传：@PartMap

```
@Multipart
POST("user/photo")
```

```
Call<User> updateUser(@PartMap Map<String, RequestBody> photos, @Part
("description") RequestBody description);
```

这和单文件上传是类似的，只是使用 **Map** 封装了上传的文件，并用 **@PartMap** 注解来标示起来。其他的和单文件上传都一样，这里就不赘述了。

5. 消息报头 Header

在 HTTP 请求中，为了防止攻击或过滤掉不安全的访问，或者添加特殊加密的访问等，以便减轻服务器的压力和保证请求的安全，通常都会在消息报头中携带一些特殊的消息头处理。**Retrofit** 也提供了 **@Header** 来添加消息报头。添加消息报头有两种方式：一种是静态的，另一种是动态的。下面先来看静态的方式，如下所示：

```
interface SomeService {
    @GET("some/endpoint")
    @Headers("Accept-Encoding: application/json")
    Call<ResponseBody> getCarType();
}
```

使用 **@Headers** 注解添加消息报头。如果想要添加多个消息报头，则可以使用 **{}** 包含起来：

```
interface SomeService {
    @GET("some/endpoint")
    @Headers({
        "Accept-Encoding: application/json",
        "User-Agent: MoonRetrofit"
    })
    Call<ResponseBody> getCarType();
}
```

以动态的方式添加消息报头如下所示：

```
interface SomeService {
    @GET("some/endpoint")
    Call<ResponseBody> getCarType(
        @Header("Location") String location);
}
```

使用 **@Header** 注解，可以通过调用 **getCarType** 方法来动态地添加消息报头。

5.7.2 源码解析 Retrofit

1. Retrofit 的创建过程

当我们使用 **Retrofit** 请求网络时，首先要写请求接口：

```
public interface IpService {
```

```
@GET("getIpInfo.php?ip=59.108.54.37")
Call<IpModel> getIpMsg();
```

接着，我们通过调用如下代码来创建 Retrofit:

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl(url)
    .addConverterFactory(GsonConverterFactory.create())
    .build();
```

Retrofit 是通过建造者模式构建出来的。接下来查看 Builder 方法做了什么:

```
public Builder() {
    this(Platform.get());
}
```

很简短，查看 Platform 的 get 方法，如下所示:

```
private static final Platform PLATFORM = findPlatform();
static Platform get() {
    return PLATFORM;
}
private static Platform findPlatform() {
    try {
        Class.forName("android.os.Build");
        if (Build.VERSION.SDK_INT != 0) {
            return new Android();
        }
    } catch (ClassNotFoundException ignored) {
    }
    try {
        Class.forName("java.util.Optional");
        return new Java8();
    } catch (ClassNotFoundException ignored) {
    }
    try {
        Class.forName("org.robovm.apple.foundation.NSObject");
        return new IOS();
    } catch (ClassNotFoundException ignored) {
    }
    return new Platform();
}
```

Platform 的 get 方法最终调用的是 findPlatform 方法，根据不同的运行平台来提供不同的线程池。接下来查看 build 方法，代码如下所示:

```

public Retrofit build() {
    if (baseUrl == null) { //1
        throw new IllegalStateException("Base URL required.");
    }

    okhttp3.Call.Factory callFactory = this.callFactory; //2
    if (callFactory == null) {
        callFactory = new OkHttpClient(); //3
    }

    Executor callbackExecutor = this.callbackExecutor;
    if (callbackExecutor == null) {
        callbackExecutor = platform.defaultCallbackExecutor(); //4
    }

    List<CallAdapter.Factory> adapterFactories = new ArrayList<>(this.
        adapterFactories); //5
        adapterFactories.add(platform.defaultCallAdapterFactory
            (callbackExecutor));

    List<Converter.Factory> converterFactories = new ArrayList<>(this.
        converterFactories); //6
    return new Retrofit(callFactory, baseUrl, converterFactories, adapterFactories,
        callbackExecutor, validateEagerly);
}

```

从上面代码注释 1 处可以看出 `baseUrl` 是必须指定的。在注释 2 处, `callFactory` 默认为 `this.callFactory`。`this.callFactory` 就是我们在构建 `Retrofit` 时调用 `callFactory` 方法所传进来的, 如下所示:

```

public Builder callFactory(okhttp3.Call.Factory factory) {
    this.callFactory = checkNotNull(factory, "factory == null");
    return this;
}

```

因此, 如果需要对 `OkHttpClient` 进行设置, 则可以构建 `OkHttpClient` 对象, 然后调用 `callFactory` 方法将设置好的 `OkHttpClient` 传进去。在注释 3 处, 如果没有设置 `callFactory`, 则直接创建 `OkHttpClient`。注释 4 处的 `callbackExecutor` 用来将回调传递到 UI 线程。注释 5 处的 `adapterFactories` 主要用于存储对 `Call` 进行转化的对象, 后面在 `Call` 的创建过程会再次提到它。注释 6 处的 `converterFactories` 主要用于存储转化数据对象, 后面也会提及。此前在例子中调用的 `addConverterFactory(GsonConverterFactory.create())`, 就是设置返回的数据支持转换为 `Gson` 对象。其最终会返回配置好的 `Retrofit` 类。

2. Call 的创建过程

下面我们创建 `Retrofit` 实例并调用如下代码来生成接口的动态代理对象:

```

IpService ipService = retrofit.create(IpService.class);

```


接下来看 Retrofit 的 create 方法做了什么，代码如下所示：

```
public <T> T create(final Class<T> service) {
    Utils.validateServiceInterface(service);
    if (validateEagerly) {
        eagerlyValidateMethods(service);
    }
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[]
    { service },
        new InvocationHandler() {
            private final Platform platform = Platform.get();
            @Override
            public Object invoke(Object proxy, Method method, Object... args)
                throws Throwable {
                if (method.getDeclaringClass() == Object.class) {
                    return method.invoke(this, args);
                }
                if (platform.isDefaultMethod(method)) {
                    return platform.invokeDefaultMethod(method, service, proxy, args);
                }
                ServiceMethod serviceMethod = loadServiceMethod(method); //1
                OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);
                return serviceMethod.callAdapter.adapt(okHttpCall);
            }
        });
}
```

可以看到 create 方法返回了一个 Proxy.newProxyInstance 动态代理对象。当我们调用 IpService 的 getIpMsg 方法时，最终会调用 InvocationHandler 的 invoke 方法。它有三个参数：第一个是代理对象，第二个是调用的方法，第三个是方法的参数。在上面代码注释 1 处 loadServiceMethod(method) 中的 method 就是我们定义的 getIpMsg 方法。下面查看 loadServiceMethod 方法里做了什么：

```
private final Map<Method, ServiceMethod> serviceMethodCache = new
    LinkedHashMap<>();
ServiceMethod loadServiceMethod(Method method) {
    ServiceMethod result;
    synchronized (serviceMethodCache) {
        result = serviceMethodCache.get(method);
        if (result == null) {
            result = new ServiceMethod.Builder(this, method).build();
            serviceMethodCache.put(method, result);
        }
    }
}
```

```

    }
}
return result;
}

```

这里首先会从 `serviceMethodCache` 查询传入的方法是否有缓存。如果有，就用缓存的 `ServiceMethod`；如果没有，就创建一个，并加入 `serviceMethodCache` 缓存起来。下面看 `ServiceMethod` 是如何构建的，代码如下所示：

```

public ServiceMethod build() {
    callAdapter = createCallAdapter();//1
    responseType = callAdapter.responseType();//2
    if (responseType == Response.class || responseType == okhttp3.Response.
        class) {
        throw methodError("'"
            + Utils.getRawType(responseType).getName()
            + "' is not a valid response body type. Did you mean ResponseBody?");
    }
    responseConverter = createResponseConverter();//3
    for (Annotation annotation : methodAnnotations) {
        parseMethodAnnotation(annotation);//4
    }
    ...
    int parameterCount = parameterAnnotationsArray.length;
    parameterHandlers = new ParameterHandler<?>[parameterCount];
    for (int p = 0; p < parameterCount; p++) {
        Type parameterType = parameterTypes[p];
        if (Utils.hasUnresolvableType(parameterType)) {
            throw parameterError(p, "Parameter type must not include a type
                variable or wildcard: %s", parameterType);
        }
        Annotation[] parameterAnnotations = parameterAnnotationsArray[p];//5
        if (parameterAnnotations == null) {
            throw parameterError(p, "No Retrofit annotation found.");
        }
        parameterHandlers[p] = parseParameter(p, parameterType,
            parameterAnnotations);
    }
    ...
    return new ServiceMethod<>(this);
}

```

在上面代码注释 1 处调用了 `createCallAdapter` 方法，它最终会得到我们在构建 Retrofit 调用 `build` 方法时 `adapterFactories` 添加的对象的 `get` 方法。Retrofit 的 `build` 方法部分代码如下所示：

```
List<CallAdapter.Factory> adapterFactories = new ArrayList<>(this.adapterFactories);
adapterFactories.add(platform.defaultCallAdapterFactory(callbackExecutor));
```

`adapterFactories` 列表默认会添加 `defaultCallAdapterFactory`。`defaultCallAdapterFactory` 指的是 `ExecutorCallAdapterFactory`。`ExecutorCallAdapterFactory` 的 `get` 方法如下所示：

```
public CallAdapter<Call<?>> get(Type returnType, Annotation[] annotations,
    Retrofit retrofit) {
    if (getRawType(returnType) != Call.class) {
        return null;
    }
    final Type responseType = Utils.getCallResponseType(returnType);
    return new CallAdapter<Call<?>>() {
        @Override
        public Type responseType() {
            return responseType;
        }
        @Override
        public <R> Call<R> adapt(Call<R> call) {
            return new ExecutorCallbackCall<>(callbackExecutor, call);
        }
    };
}
```

`get` 方法会得到 `CallAdapter` 对象，`CallAdapter` 的 `responseType` 方法会返回数据的真实类型，比如 传入的是 `Call<IpModel>`，`responseType` 方法就会返回 `IpModel`。`adapt` 方法会创建 `ExecutorCallbackCall`，它会将 `call` 的回调转发至 UI 线程。

接着回到 `ServiceMethod` 的 `build` 方法。在那里的注释 2 处调用 `CallAdapter` 的 `responseType` 得到的是返回数据的真实类型。注释 3 处调用 `createResponseConverter` 方法来遍历 `converterFactories` 列表中存储的 `Converter.Factory`，并返回一个合适的 `Converter` 用来转换对象。此前我们在构建 `Retrofit` 时调用了 `addConverterFactory(GsonConverterFactory.create())`，这段代码将 `GsonConverterFactory`（`Converter.Factory` 的子类）添加到 `converterFactories` 列表中，表示返回的数据支持转换为 JSON 对象。注释 4 处遍历 `parseMethodAnnotation` 方法来对请求方式（比如 GET、POST）和请求地址进行解析。注释 5 处对方法中的参数注解进行解析（比如 `@Query`、`@Part`）。最后创建 `ServiceMethod` 类并返回。

接下来回过头来查看 `Retrofit` 的 `create` 方法（在第 264 页），在调用了 `loadServiceMethod` 方法后会创建 `OkHttpClient`，`OkHttpClient` 的构造方法只是进行了赋值操作。紧接着调用 `serviceMethod.callAdapter.adapt(okHttpClient)`。`callAdapter` 的 `adapt` 方法前面讲过，它会创建 `ExecutorCallbackCall`，并传入 `OkHttpClient`。`ExecutorCallbackCall` 的部分代码如下所示：

```
ExecutorCallbackCall(Executor callbackExecutor, Call<T> delegate) {
```



```

...
call.enqueue(new okhttp3.Callback() { //1
@Override
public void onResponse(okhttp3.Call call, okhttp3.Response rawResponse)
    throws IOException {
    Response<T> response;
    try {
        response = parseResponse(rawResponse); //2
    } catch (Throwable e) {
        callFailure(e);
        return;
    }
    callSuccess(response);
}
...
}

```

在上面代码注释 1 处调用了 `okhttp3.Call` 类型的 `call` 的 `enqueue` 方法。注释 2 处调用了 `parseResponse` 方法：

```

Response<T> parseResponse(okhttp3.Response rawResponse) throws IOException {
    ResponseBody rawBody = rawResponse.body();
    ...
    int code = rawResponse.code();
    if (code < 200 || code >= 300) {
        try {
            ResponseBody bufferedBody = Utils.buffer(rawBody);
            return Response.error(bufferedBody, rawResponse);
        } finally {
            rawBody.close();
        }
    }
    if (code == 204 || code == 205) {
        return Response.success(null, rawResponse);
    }
    ExceptionCatchingRequestBody catchingBody = new ExceptionCatchingRequestBody(
        rawBody);
    try {
        T body = serviceMethod.toResponse(catchingBody); //2
        return Response.success(body, rawResponse);
    } catch (RuntimeException e) {
        catchingBody.throwIfCaught();
        throw e;
    }
}

```

根据返回的不同状态码 `code` 值来做不同的操作。如果顺利，则会调用上面代码注释 2 处的代码。接下来看 `toResponse` 方法里做了什么：

```
T toResponse(ResponseBody body) throws IOException {
    return responseConverter.convert(body);
}
```

这个 `responseConverter` 就是此前讲过在 `ServiceMethod` 的 `build` 方法调用 `createResponseConverter` 方法返回的 `Converter`。在此前的例子中我们传入的是 `GsonConverterFactory`，因此可以查看 `GsonConverterFactory` 的代码，如下所示：

```
public final class GsonConverterFactory extends Converter.Factory {
    ...
    @Override
    public Converter<ResponseBody, ?> responseBodyConverter(Type type, Annotation[]
        annotations, Retrofit retrofit) {
        TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));
        return new GsonResponseBodyConverter<>(gson, adapter);
    }
    ...
}
```

在 `GsonConverterFactory` 中有一个方法 `responseBodyConverter`，它最终会创建 `GsonResponseBodyConverter`：

```
final class GsonResponseBodyConverter<T> implements Converter<ResponseBody,
    T> {
    private final Gson gson;
    private final TypeAdapter<T> adapter;
    GsonResponseBodyConverter(Gson gson, TypeAdapter<T> adapter) {
        this.gson = gson;
        this.adapter = adapter;
    }
    @Override
    public T convert(ResponseBody value) throws IOException {
        JsonReader jsonReader = gson.newJsonReader(value.charStream());
        try {
            return adapter.read(jsonReader);
        } finally {
            value.close();
        }
    }
}
```

在 `GsonResponseBodyConverter` 的 `convert` 方法里会将回调的数据转换为 JSON 格式。因此，我们也知道了此前调用 `responseConverter.convert` 是为了转换为特定的数据格式。`Call` 的 `enqueue` 方法主要做的就是用 `OkHttp` 来请求网络，将返回的 `Response` 进行数据转换并回调给 UI 线程。Retrofit 的源码就讲到这里了。

5.8 本章小结

本章的内容比较多，首先我们了解了网络编程的基础网络分层、TCP 的三次握手与四次挥手、HTTP 协议原理以及 `HttpClient` 与 `HttpURLConnection` 的用法。在这些内容的基础上，我们学习了网络编程中常用的框架，以及它们的原理，这也是要告诉读者：当我们使用框架的同时也需要了解框架的原理，这样不仅能学习框架开发者的设计理念，同时对使用框架也有很大的帮助。



第 6 章 设计模式

本章我们学习设计模式。设计模式的重要性不用我多说，如果你想成为优秀的 Android 工程师，设计模式是必须要掌握的。在本书加入设计模式这一章，其实并不突兀，因为本书的很多章节包括源码分析以及后文讲解的架构设计都是离不开设计模式的。如果读者对设计模式比较了解，在阅读本书时会吸收更多的知识点，并且有更多的领悟。

6.1 设计模式六大原则

在讲到常用的设计模式之前，首先介绍设计模式的六大原则，它们分别是单一职责原则、开放封闭原则、里氏替换原则、依赖倒置原则、迪米特原则和接口隔离原则。

单一职责原则 定义：就一个类而言，应该仅有一个引起它变化的原因。

从这句定义我们很难理解它的含义，这通俗地讲就是我们不要让一个类承担过多的职责。如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受到破坏。比如我们会看到一些 Android 开发者在 Activity 中写 Bean 文件、网络数据处理，如果有列表的话 Adapter 也写在 Activity 中。至于他们为什么这么做，除了好找也没什么理由了。把其拆分到其他类岂不是更好找？如果 Activity 过于臃肿、行数过多，显然不是好事。如果我们要修改 Bean 文件、网络处理和 Adapter 都需要上这个 Activity 来修改，就会导致引起该 Activity 变化的原因太多，我们在版本维护时也会比较头疼。这也就严重违背了定义：“就一个类而言，应该仅有一个引起它变化的原因。”单一职责的划分界限不是很清晰，很多时候需要靠个人经验来界定，因此它是一个饱受争议却又极其重要的原则。

开放封闭原则 定义：类、模块、函数等应该是可以拓展的，但是不可修改。

开放封闭有两个含义：一个是对于拓展是开放的，另一个是对于修改是封闭的。对于开发

来说，需求肯定是要变化的，但是有新需求，我们就要把类重新改一遍，这显然是令人头疼的。所以我们设计程序时，面对需求的改变要尽可能地保证相对稳定，尽量通过扩展的方式来实现变化，而不是通过修改原有的代码来实现。假设我们要实现一个列表，一开始只有查询的功能，后来产品又要新增“添加”功能，过几天又要增加“删除”功能。大多数人的做法是写一个方法，然后通过传入不同的值控制方法来实现不同的功能。但是如果又要新增功能，我们还得修改方法。用开发封闭原则解决就是增加一个抽象的功能类，让添加、删除和查询作为这个抽象功能类的子类。这样如果我们再新增功能，你就会发现自己无须修改原有的类，只需要添加一个功能类的子类实现功能类的方法就可以了。

里氏替换原则 定义：所有引用基类（父类）的地方必须能透明地使用其子类的对象。

里氏替换原则告诉我们，在软件中将一个基类对象替换成其子类对象，程序将不会产生任何错误和异常；反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。里氏替换原则是实现开放封闭原则的重要方式之一。由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。在使用里氏替换原则时需要注意以下几个问题。

子类的所有方法必须在父类中声明，或子类必须实现父类中声明的所有方法。根据里氏替换原则，为了保证系统的扩展性，在程序中通常使用父类来进行定义。如果一个方法只存在于子类中，在父类中不提供相应的声明，则无法在以父类定义的对象中使用该方法。

我们在运用里氏替换原则时，尽量把父类设计为抽象类或者接口，让子类继承父类或实现父接口，并实现在父类中声明的方法。运行时，子类实例替换父类实例，我们可以很方便地扩展系统的功能，同时无须修改原有子类的代码；增加新的功能可以通过增加一个新的子类来实现。里氏替换原则是开放封闭原则的具体实现手段之一。

在 Java 语言中，在编译阶段，Java 编译器会检查一个程序是否符合里氏替换原则。这是一个与实现无关的、纯语法意义上的检查，但 Java 编译器的检查是有局限性的。

依赖倒置原则 定义：高层模块不应该依赖低层模块，两者都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象。

在 Java 中，抽象指接口或者抽象类，两者都是不能被实例化的；细节就是实现类，实现接口或者继承抽象类而产生的就是细节，也就是可以加上一个关键字 `new` 产生的对象。高层模块就是调用端，低层模块就是具体实现类。依赖倒置原则在 Java 中的表现就是，模块间的依赖通过抽象发生，实现类之间不发生直接依赖关系，其依赖关系是通过接口或者抽象类产生的。如果类与类直接依赖细节，那么就会直接耦合。如此一来当修改时，就会同时修改依赖者代码，这样限制了可扩展性。

迪米特原则 定义：一个软件实体应当尽可能少地与其他实体发生相互作用。

这也被称为最少知识原则。如果一个系统符合迪米特原则，那么当其中某一个模块发生修改时，就会尽量少地影响其他模块。迪米特原则要求我们在设计系统时，应该尽量减少对象之间的交互。如果两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用。如果其中的一个对象需要调用另一个对象的某一个方法，则可以通过第三者转发这个

调用。简言之，就是通过引入一个合理的第三者来降低现有对象之间的耦合度。在将迪米特原则运用到系统设计中时，要注意下面几点：

- 在类的划分上，应当尽量创建松耦合的类。类之间的耦合度越低，就越有利于复用。一个处在松耦合中的类一旦被修改，则不会对关联的类造成太大波及。
- 在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限。
- 在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

接口隔离原则 定义：一个类对另一个类的依赖应该建立在最小的接口上。

建立单一接口，不要建立庞大臃肿的接口；尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图建立一个很庞大的接口供所有依赖它的类调用。采用接口隔离原则对接口进行约束时，要注意以下几点：

- 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计的灵活性；但是如果过小，则会造成接口数量过多，使设计复杂化。所以，一定要适度。
- 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
- 提高内聚，减少对外交互。接口方法尽量少用 `public` 修饰。接口是对外的承诺，承诺越少对系统的开发越有利，变更风险也会越少。

6.2 设计模式分类

GoF 提出的设计模式总共有 23 种，根据目的准则分类，分为三大类。

- 创建型设计模式，共 5 种：单例模式、工厂方法模式、抽象工厂模式、建造者模式、原型模式。
- 结构型设计模式，共 7 种：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
- 行为型设计模式，共 11 种：策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

另外，随着设计模式的发展也涌现出很多新的设计模式：它们分别是规格模式、对象池模式、雇工模式、黑板模式和空对象模式等。因为本书并不是专门讲设计模式的，所以本章仅会讲相对常用的设计模式，以便带领大家快速地掌握设计模式。如果读者想要了解更多设计模式知识，则需要阅读专门讲解设计模式的书籍。

6.3 创建型设计模式

创建型设计模式，顾名思义就是与对象创建有关，它包括单例模式、工厂方法模式、抽象工厂模式、建造者模式、原型模式。虽然简单工厂模式不是 GoF 提出的创建型设计模式，但是

它对理解抽象工厂模式有帮助，因此本节也会讲到简单工厂模式。下面分别学习单例模式、简单工厂模式、工厂方法模式和建造者模式。

6.3.1 单例模式

定义：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

单例模式的通用结构图如图 6-1 所示。

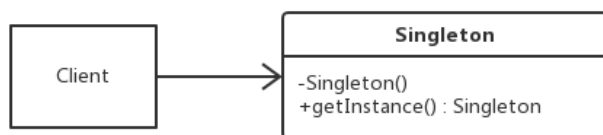


图 6-1 单例模式的通用结构图

在图 6-1 中，Client 为客户端，Singleton 是单例类，通过调用 Singleton.getInstance()来获取实例对象。

1. 单例模式的 6 种写法

单例模式有多种写法并且其各有利弊，现在我们来学习其中的 6 种写法。

(1) 饿汉模式

```

public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton () {
    }
    public static Singleton getInstance() {
        return instance;
    }
}
  
```

这种方式在类加载时就完成了初始化，所以类加载较慢，但获取对象的速度快。这种方式基于类加载机制，避免了多线程的同步问题。在类加载的时候就完成实例化，没有达到懒加载的效果。如果从始至终未使用过这个实例，则会造成内存的浪费。

(2) 懒汉模式（线程不安全）

```

public class Singleton {
    private static Singleton instance;
    private Singleton () {
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
    }
}
  
```

```

    }
    return instance;
}
}

```

懒汉模式声明了一个静态对象，在用户第一次调用时初始化。这虽然节约了资源，但第一次加载时需要实例化，反应稍慢一些，而且在多线程时不能正常工作。

（3）懒汉模式（线程安全）

```

public class Singleton {
    private static Singleton instance;
    private Singleton () {
    }
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

```

这种写法能够在多线程中很好地工作，但是每次调用 `getInstance` 方法时都需要进行同步。这会造成不必要的同步开销，而且大部分时候我们是用不到同步的。所以，不建议用这种模式。

（4）双重检查模式（DCL）

```

public class Singleton {
    private volatile static Singleton singleton;
    private Singleton () {
    }
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return singleton;
    }
}

```

这种写法在 `getSingleton` 方法中对 `Singleton` 进行了两次判空：第一次是为了不必要的同步，第二次是在 `Singleton` 等于 `null` 的情况下才创建实例。在这里使用 `volatile` 会或多或少地影响性

能，但考虑到程序的正确性，牺牲这点性能还是值得的。DCL 的优点是资源利用率高。第一次执行 `getInstance` 时单例对象才被实例化，效率高。其缺点是第一次加载时反应稍慢一些，在高并发环境下也有一定的缺陷。DCL 虽然在一定程度上解决了资源的消耗和多余的同步、线程安全等问题，但其还是在某些情况会出现失效的问题，也就是 DCL 失效。这里建议用静态内部类单例模式来替代 DCL。

(5) 静态内部类单例模式

```
public class Singleton {
    private Singleton() {
    }

    public static Singleton getInstance() {
        return SingletonHolder.sInstance;
    }

    private static class SingletonHolder {
        private static final Singleton sInstance = new Singleton();
    }
}
```

第一次加载 `Singleton` 类时并不会初始化 `sInstance`，只有第一次调用 `getInstance` 方法时虚拟机加载 `SingletonHolder` 并初始化 `sInstance`。这样不仅能确保线程安全，也能保证 `Singleton` 类的唯一性。所以，推荐使用静态内部类单例模式。

(6) 枚举单例

```
public enum Singleton {
    INSTANCE;
    public void doSomething() {
    }
}
```

默认枚举实例的创建是线程安全的，并且在任何情况下都是单例。在上面讲的几种单例模式实现中，有一种情况下其会重新创建对象，那就是反序列化：将一个单例实例对象写到磁盘再读回来，从而获得了一个实例。反序列化操作提供了 `readResolve` 方法，这个方法可以让开发人员控制对象的反序列化。在上述几个方法示例中，如果要杜绝单例对象被反序列化时重新生成对象，就必须加入如下方法：

```
private Object readResolve() throws ObjectStreamException {
    return singleton;
}
```

枚举单例的优点就是简单，但是大部分应用开发很少用枚举，其可读性并不是很高。到这里单例模式的 6 种写法都介绍完了。至于选择用哪种形式的单例模式，则取决于你的项目本身情况：是否为复杂的并发环境，或者是否需要控制单例对象的资源消耗。

2. 单例模式的使用场景

在一个系统中，要求一个类有且仅有一个对象，它的具体使用场景如下：

- 整个项目需要一个共享访问点或共享数据。
- 创建一个对象需要耗费的资源过多，比如访问 I/O 或者数据库等资源。
- 工具类对象。

6.3.2 简单工厂模式

简单工厂模式（又叫作静态工厂方法模式），其属于创建型设计模式，但是并不属于 23 种 GoF 设计模式之一。提到它是为了让大家能够更好地理解后面讲到的工厂方法模式。

定义：简单工厂模式属于创建型模式，其又被称为静态工厂方法模式，这是由一个工厂对象决定创建出哪一种产品类的实例。

简单工厂模式的结构图如图 6-2 所示。

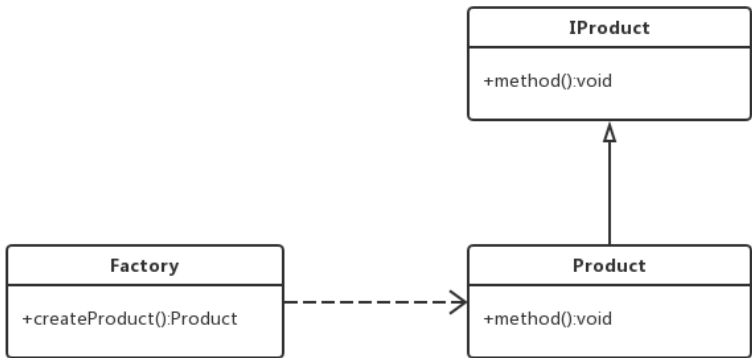


图 6-2 简单工厂模式的结构图

在简单工厂模式中有如下角色。

- **Factory**：工厂类，这是简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类的创建产品类的方法可以被外界直接调用，创建所需的产品对象。
- **IProduct**：抽象产品类，这是简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。
- **Product**：具体产品类，这是简单工厂模式的创建目标。

1. 简单工厂模式的简单实现

这里我们用生产计算机来举例，假设有一个计算机的代工生产商，它目前已经可以代工生产联想计算机了。随着业务的拓展，这个代工生产商还要生产惠普和华硕的计算机。这样我们就需要用一个单独的类来专门生产计算机，这就用到了简单工厂模式。下面我们来实现简单工厂模式。

(1) 抽象产品类

我们创建一个计算机的抽象产品类，其有一个抽象方法用于启动计算机，如下所示：

```
public abstract class Computer {  
    /**  
     * 产品的抽象方法，由具体的产品类实现  
     */  
    public abstract void start();  
}
```

(2) 具体产品类

接着我们创建各个品牌的计算机，其都继承了自己的父类 `Computer`，并实现了父类的 `start` 方法。具体的计算机产品分别是联想、惠普和华硕计算机：

```
public class LenovoComputer extends Computer{  
    @Override  
    public void start() {  
        System.out.println("联想计算机启动");  
    }  
}  
public class HpComputer extends Computer{  
    @Override  
    public void start() {  
        System.out.println("惠普计算机启动");  
    }  
}  
public class AsusComputer extends Computer {  
    @Override  
    public void start() {  
        System.out.println("华硕计算机启动");  
    }  
}
```

(3) 工厂类

接下来创建一个工厂类，它提供了一个静态方法 `createComputer` 用来生产计算机。你只需要传入自己想生产的计算机的品牌，它就会实例化相应品牌的计算机对象，代码如下所示：

```
public class ComputerFactory {  
    public static Computer createComputer(String type){  
        Computer mComputer=null;  
        switch (type) {  
            case "lenovo":  
                mComputer=new LenovoComputer();  
            // ... other cases ...  
        }  
    }  
}
```

```

        break;
    case "hp":
        mComputer=new HpComputer();
        break;
    case "asus":
        mComputer=new AsusComputer();
        break;
    }
    return mComputer;
}
}

```

(4) 客户端调用工厂类

客户端调用工厂类，传入"hp"生产出惠普计算机并调用该计算机对象的 start 方法，如下所示：

```

public class CreatComputer {
    public static void main(String[]args){
        ComputerFactory.createComputer("hp").start();
    }
}

```

2. 使用简单工厂模式的场景和优缺点

- 使用场景：
 - 工厂类负责创建的对象比较少。
 - 客户只需知道传入工厂类的参数，而无须关心创建对象的逻辑。
- 优点：使用户根据参数获得对应的类实例，避免了直接实例化类，降低了耦合性。
- 缺点：可实例化的类型在编译期间已经被确定。如果增加新类型，则需要修改工厂，这违背了开放封闭原则。简单工厂需要知道所有要生成的类型，其当子类过多或者子类层次过多时不适合使用。

6.3.3 工厂方法模式

定义：定义一个用于创建对象的接口，让子类决定实例化哪个类。工厂方法使一个类的实例化延迟到其子类。

工厂方法模式的结构图如图 6-3 所示。

在工厂方法模式中有如下角色。

- **Product：**抽象产品类。
- **ConcreteProduct：**具体产品类，实现 Product 接口。
- **Factory：**抽象工厂类，该方法返回一个 Product 类型的对象。

- ConcreteFactory: 具体工厂类，返回 ConcreteProduct 实例。

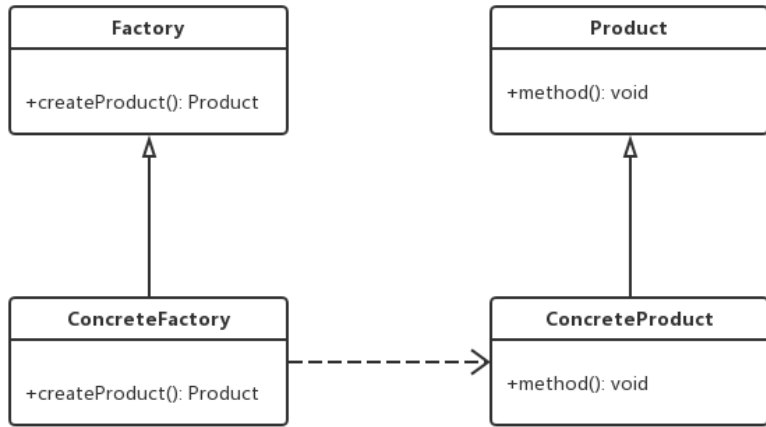


图 6-3 工厂方法模式的结构图

1. 工厂方法模式的简单实现

工厂方法模式的抽象产品类与具体产品类的创建和简单工厂模式是一样的，具体看 6.3.2 节，这里就不赘述了。

(1) 创建抽象工厂

抽象工厂里面有一个 `createComputer` 方法，想生产哪个品牌的计算机就生产哪个品牌的，如下所示。

```
public abstract class ComputerFactory {
    public abstract <T extends Computer> T createComputer(Class<T> clz);
}
```

(2) 具体工厂

广达代工厂是一个具体的工厂，其继承抽象工厂，通过反射来生产不同厂家的计算机：

```
public class GDComputerFactor extends ComputerFactory {
    @Override
    public <T extends Computer> T createComputer(Class<T> clz) {
        Computer computer=null;
        String classname=clz.getName();
        try {
            //通过反射来生产不同厂家的计算机
            computer= (Computer) Class.forName(classname).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        return (T) computer;
    }
}

```

(3) 客户端调用

客户端创建了 `GDComputerFactor`，并分别生产了联想、惠普和华硕计算机：

```

public class Client {
    public static void main(String[] args) {
        ComputerFactory computerFactory = new GDComputerFactor();
        LenovoComputer mLenovoComputer=computerFactory.createComputer
            (LenovoComputer.class);
        mLenovoComputer.start();
        HpComputer mHpComputer=computerFactory.createComputer(HpComputer.
            class);
        mHpComputer.start();
        AsusComputer mAsusComputerr=computerFactory.createComputer
            (AsusComputer.class);
        mAsusComputerr.start();
    }
}

```

2. 工厂方法与简单工厂

对于简单工厂模式，我们都知道其在工厂类中包含了必要的逻辑判断，根据不同的条件来动态实例化相关的类。对客户端来说，这去除了与具体产品的依赖；但与此同时也会带来一个问题：如果我们要增加产品，比如我们要生产苹果计算机，就需要在工厂类中添加一个 `Case` 分支条件，这违背了开放封闭原则，对修改也开放了。而工厂方法模式就没有违背这个开放封闭原则。如果我们需要生产苹果计算机，则无须修改工厂类，直接创建产品即可。

6.3.4 建造者模式

建造者模式也被称为生成器模式，它是创建一个复杂对象的创建型模式，其将构建复杂对象的过程和它的部件解耦，使得构建过程和部件的表示分离开来。例如我们要 DIY 一台台式计算机。我们找到 DIY 商家。这时我们可以要求这台计算机的 CPU、主板或者其他部件都是什么牌子的、什么配置的，这些部件是我们可以根据自己的需求来变化的。但是这些部件组装成计算机的过程是一样的，我们无须知道这些部件是怎样组装成计算机的，我们只需要提供相关部件的牌子和配置就可以了。对于这种情况我们就可以采用建造者模式，将部件和组装过程分离，使得构建过程和部件都可以自由拓展，两者之间的耦合也降到最低。

定义：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。建造者模式的结构图如图 6-4 所示。

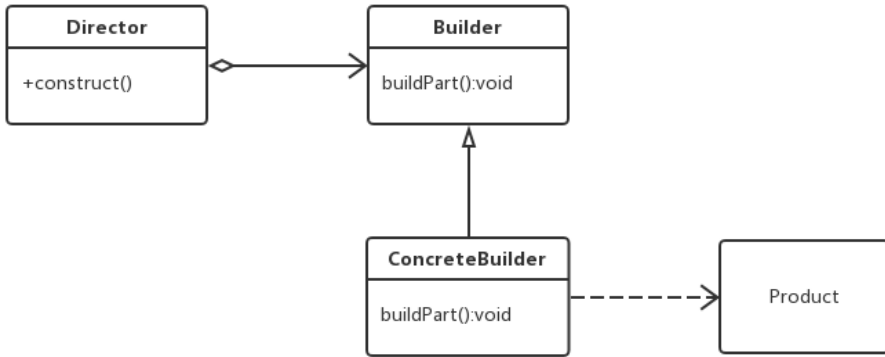


图 6-4 建造者模式的结构图

在建造者模式中有如下角色。

- **Director**: 导演类，负责安排已有模块的顺序，然后通知 **Builder** 开始建造。
- **Builder**: 抽象 **Builder** 类，规范产品的组建，一般由子类实现。
- **ConcreteBuilder**: 具体建造者，实现抽象 **Builder** 类定义的所有方法，并且返回一个组建好的对象。
- **Product**: 产品类。

1. 建造者模式的简单实现

这里我们就用 DIY 组装计算机的例子来实现一下建造者模式。

(1) 创建产品类

我要组装一台计算机，计算机被抽象为 **Computer** 类，它有 3 个部件：CPU、主板和内存，并在里面提供了 3 个方法分别用来设置 CPU、主板和内存：

```
public class Computer {
    private String mCpu;
    private String mMainboard;
    private String mRam;
    public void setmCpu(String mCpu) {
        this.mCpu = mCpu;
    }
    public void setmMainboard(String mMainboard) {
        this.mMainboard = mMainboard;
    }
    public void setmRam(String mRam) {
        this.mRam = mRam;
    }
}
```

（2）创建 Builder 类规范产品的组建

商家组装计算机有一套组装方法的模板，就是一个抽象的 Builder 类，其里面提供了安装 CPU、主板和内存的方法，以及组装成计算机的 create 方法，如下所示：

```
public abstract class Builder {
    public abstract void buildCpu(String cpu);
    public abstract void buildMainboard(String mainboard);
    public abstract void buildRam(String ram);
    public abstract Computer create();
}
```

商家实现了抽象的 Builder 类，MoonComputerBuilder 类用于组装计算机，代码如下所示：

```
public class MoonComputerBuilder extends Builder {
    private Computer mComputer = new Computer();
    @Override
    public void buildCpu(String cpu) {
        mComputer.setmCpu(cpu);
    }
    @Override
    public void buildMainboard(String mainboard) {
        mComputer.setmMainboard(mainboard);
    }
    @Override
    public void buildRam(String ram) {
        mComputer.setmRam(ram);
    }
    @Override
    public Computer create() {
        return mComputer;
    }
}
```

（3）用导演类来统一组装过程

商家的导演类用来规范组装计算机的流程规范，先安装主板，再安装 CPU，最后安装内存并组装成计算机：

```
public class Direcror {
    Builder mBuild=null;
    public Direcror(Builder build){
        this.mBuild=build;
    }
    public Computer CreateComputer(String cpu,String mainboard,String ram){
```

```

        //规范建造流程
        this.mBuild.buildMainboard(mainboard);
        this.mBuild.buildCpu(cpu);
        this.mBuild.buildRam(ram);
        return mBuild.create();
    }
}

```

(4) 客户端调用导演类

最后商家用导演类组装计算机。我们只需要提供自己想要的 CPU、主板和内存就可以了，至于商家怎样组装计算机我们无须知道。具体代码如下所示：

```

public class CreatComputer {
    public static void main(String[] args) {
        Builder mBuilder=new MoonComputerBuilder();
        Direcror mDirecror=new Direcror(mBuilder);
        //组装计算机
        mDirecror.CreateComputer("i7-6700","华擎玩家至尊","三星 DDR4");
    }
}

```

2. 使用建造者模式的场景和优缺点

- 使用场景：
 - 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。
 - 相同的方法，不同的执行顺序，产生不同的事件结果时。
 - 多个部件或零件都可以被装配到一个对象中，但是产生的运行结果又不相同时。
 - 产品类非常复杂，或者产品类中的调用顺序不同而产生了不同的效能。
 - 在创建一些复杂的对象时，这些对象的内部组成构件间的建造顺序是稳定的，但是对象的内部组成构件面临着复杂的变化。
- 优点：
 - 使用建造者模式可以使客户端不必知道产品内部组成的细节。
 - 具体的建造者类之间是相互独立的，容易扩展。
 - 由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响。
- 缺点：产生多余的 Build 对象以及导演类。

6.4 结构型设计模式

结构型设计模式是从程序的结构上解决模块之间的耦合问题，它包括适配器模式、代理模式、装饰模式、外观模式、桥接模式、组合模式和享元模式。本节会介绍代理模式、装饰模式、

外观模式和享元模式。

6.4.1 代理模式

代理模式也被称为委托模式，它是结构型设计模式的一种。在现实生活中我们用到类似代理模式的场景有很多，比如代理上网、打官司等。

定义：为其他对象提供一种代理以控制对这个对象的访问。

代理模式的结构图如图 6-5 所示。

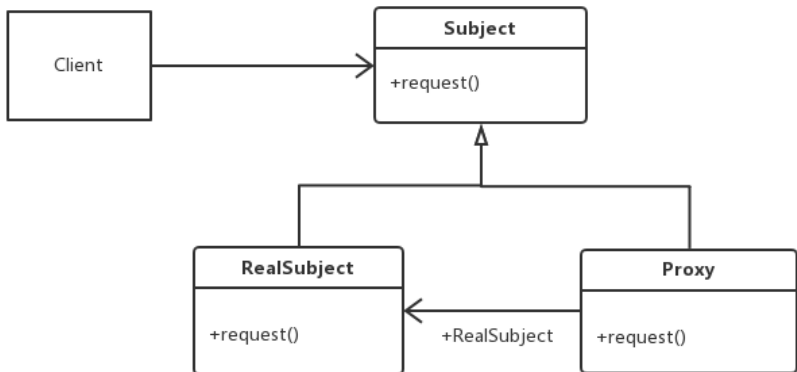


图 6-5 代理模式的结构图

在代理模式中有如下角色。

- **Subject:** 抽象主题类，声明真实主题与代理的共同接口方法。
- **RealSubject:** 真实主题类，代理类所代表的真实主题。客户端通过代理类间接地调用真实主题类的方法。
- **Proxy:** 代理类，持有对真实主题类的引用，在其所实现的接口方法中调用真实主题类中相应的接口方法执行。
- **Client:** 客户端类。

1. 代理模式的简单实现

我多年没有回过哈尔滨了，很想念哈尔滨秋林红肠的味道。但是本人工作一直很忙抽不开身，不能够亲自回哈尔滨购买，于是就托在哈尔滨的朋友帮我购买秋林红肠。

(1) 抽象主题类

抽象主题类具有真实主题类和代理的共同接口方法，共同的方法就是购买：

```

public interface IShop {
    void buy();
}
  
```

（2）真实主题类

这个购买者 LiuWangShu 也就是我，实现了 IShop 接口提供的 buy()方法，如下所示：

```
public class LiuWangShu implements IShop {
    @Override
    public void buy() {
        System.out.println("购买");
    }
}
```

（3）代理类

我找的代理类同样也要实现 IShop 接口，并且要持有被代理者，在 buy()方法中调用了被代理者的 buy()方法：

```
public class Purchasing implements IShop {
    private IShop mShop;
    public Purchasing(IShop shop){
        mShop=shop;
    }
    @Override
    public void buy() {
        mShop.buy();
    }
}
```

（4）客户端类

```
public class Client {
    public static void main(String[] args){
        IShop liuwangshu=new LiuWangShu();
        IShop purchasing=new Purchasing(liuwangshu);
        purchasing.buy();
    }
}
```

客户端类的代码就是代理类包含了真实主题类（被代理者），最终调用的都是真实主题类（被代理者）实现的方法。在上面的例子中就是 LiuWangShu 类的 buy()方法，所以运行的结果就是“购买”。

2. 动态代理的简单实现

从编码的角度来说，代理模式分为静态代理和动态代理。上面的例子是静态代理，在代码运行前就已经存在了代理类的 class 编译文件；而动态代理则是在代码运行时通过反射来动态地生成代理类的对象，并确定到底来代理谁。也就是我们在编码阶段无须知道代理谁，代理谁将

会在代码运行时决定。Java 提供了动态的代理接口 `InvocationHandler`，实现该接口需要重写 `invoke()` 方法。下面我们在上面静态代理的例子上做修改。首先创建动态代理类，代码如下所示：

```
public class DynamicPurchasing implements InvocationHandler{
    private Object obj;
    public DynamicPurchasing(Object obj){
        this.obj=obj;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable {
        Object result=method.invoke(obj, args);
        if(method.getName().equals("buy")){
            System.out.println("Liuwangshu 在买买买");
        }
        return result;
    }
}
```

在动态代理类中我们声明一个 `Object` 的引用，该引用指向被代理类，我们调用被代理类的具体方法在 `invoke()` 方法中执行。接下来我们修改客户端类代码：

```
public class Client {
    public static void main(String[] args){
        //创建 LiuWangShu
        IShop liuwangshu=new LiuWangShu();
        //创建动态代理
        DynamicPurchasing mDynamicPurchasing=new DynamicPurchasing(liuwangshu);
        //创建 LiuWangShu 的 ClassLoader
        ClassLoader loader=liuwangshu.getClass().getClassLoader();
        //动态创建代理类
        IShop purchasing= (IShop) Proxy.newProxyInstance(loader,new Class[]
        {IShop.class},mDynamicPurchasing);
        purchasing.buy();
    }
}
```

调用 `Proxy.newProxyInstance()` 来生成动态代理类，调用 `purchasing` 的 `buy` 方法会调用 `DynamicPurchasing` 的 `invoke` 方法。在 5.7.2 节 Call 的创建过程中也使用了动态代理。

3. 代理模式的类型和优点

代理模式从编码的角度来说可以分为静态代理和动态代理，而从适用范围来讲则可分为以下 4 种类型。

- 远程代理：为一个对象在不同的地址空间提供局部代表，这样系统可以将 Server 部分的实现隐藏。
- 虚拟代理：使用一个代理对象表示一个十分耗费资源的对象并在真正需要时才创建。
- 安全代理：用来控制真实对象访问时的权限。一般用于真实对象有不同的访问权限时。
- 智能指引：当调用真实的对象时，代理处理另外一些事，比如计算真实对象的引用计数，当该对象没有引用时，可以自动释放它；或者访问一个实际对象时，检查是否已经能够锁定它，以确保其他对象不能改变它。

代理模式的优点主要有以下几点：

- 真实主题类就是实现实际的业务逻辑，不用关心其他非本职的工作。
- 真实主题类随时都会发生变化；但是因为它实现了公共的接口，所以代理类可以不做任何修改就能够使用。

6.4.2 装饰模式

装饰模式是结构型设计模式之一，其在不必改变类文件和使用继承的情况下，动态地扩展一个对象的功能，是继承的替代方案之一。它通过创建一个包装对象，也就是装饰来包裹真实的对象。

定义：动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更为灵活。

装饰模式的结构图如图 6-6 所示。

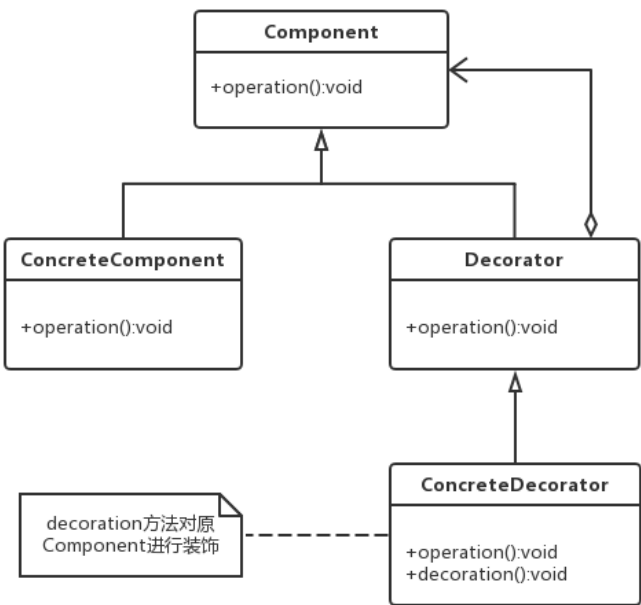


图 6-6 装饰模式的结构图

在装饰模式中有如下角色。

- **Component**: 抽象组件，可以是接口或是抽象类，被装饰的最原始的对象。
- **ConcreteComponent**: 组件具体实现类。**Component** 的具体实现类，被装饰的具体对象。
- **Decorator**: 抽象装饰者，从外类来拓展 **Component** 类的功能，但对于 **Component** 来说无须知道 **Decorator** 的存在。在它的属性中必然有一个 **private** 变量指向 **Component** 抽象组件。
- **ConcreteDecorator**: 装饰者的具体实现类。

1. 装饰模式的简单实现

装饰模式在现实生活中有很多例子，比如给一个人穿上各种衣服，给一幅画涂色、上框等。因为笔者喜欢武侠，所以在此就举一个武侠修炼武功的例子：杨过本身就会全真剑法，有两位武学前辈洪七公和欧阳锋分别向杨过传授过打狗棒法和蛤蟆功，这样杨过除了会全真剑法，还会打狗棒法和蛤蟆功。洪七公和欧阳锋就起到了“装饰”杨过的作用。

(1) 抽象组件

作为武侠，肯定要会使用武功。我们先定义一个武侠的抽象类，里面有使用武功的抽象方法：

```
public abstract class Swordsman {  
    /**  
     * Swordsman 武侠有使用武功的抽象方法  
     */  
    public abstract void attackMagic();  
}
```

(2) 组件具体实现类

被装饰的具体对象，在这里就是被教授武学的具体武侠，也就是杨过。杨过作为武侠当然也会武学（虽然不怎么厉害），代码如下所示：

```
public class YangGuo extends Swordsman {  
    @Override  
    public void attackMagic() {  
        //杨过初始的武学是全真剑法  
        System.out.println("杨过使用全真剑法");  
    }  
}
```

(3) 抽象装饰者

抽象装饰者保持了一个对抽象组件的引用，方便调用被装饰对象中的方法。在这个例子中就是武学前辈要持有武侠的引用，方便教授他武学并使他融会贯通：

```

public abstract class Master extends Swordsman{
    private Swordsman mSwordsman;
    public Master(Swordsman mSwordsman){
        this.mSwordsman=mSwordsman;
    }
    @Override
    public void attackMagic() {
        mSwordsman.attackMagic();
    }
}

```

(4) 装饰者具体实现类

这个例子中用两个装饰者具体实现类，分别是洪七公和欧阳锋，他们负责向杨过传授新的武功，如下所示：

```

public class HongQiGong extends Master {
    public HongQiGong(Swordsman mSwordsman) {
        super(mSwordsman);
    }
    public void teachAttackMagic(){
        System.out.println("洪七公教授打狗棒法");
        System.out.println("杨过使用打狗棒法");
    }
    @Override
    public void attackMagic() {
        super.attackMagic();
        teachAttackMagic();
    }
}

public class OuYangFeng extends Master {
    public OuYangFeng(Swordsman mSwordsman) {
        super(mSwordsman);
    }
    public void teachAttackMagic(){
        System.out.println("欧阳锋教授蛤蟆功");
        System.out.println("杨过使用蛤蟆功");
    }
    @Override
    public void attackMagic() {
        super.attackMagic();
        teachAttackMagic();
    }
}

```

(5) 客户端调用

经过洪七公和欧阳锋的教导，杨过除了初始武学全真剑法又学会了打狗棒法和蛤蟆功。客户端调用代码如下所示：

```
public class Client {  
    public static void main(String[] args){  
        //创建杨过  
        YangGuo mYangGuo=new YangGuo();  
        //洪七公向杨过传授打狗棒法，杨过学会了打狗棒法  
        HongQiGong mHongQiGong=new HongQiGong(mYangGuo);  
        mHongQiGong.attackMagic();  
        //欧阳锋向杨过传授蛤蟆功，杨过学会了蛤蟆功  
        OuYangFeng mOuYangFeng=new OuYangFeng(mYangGuo);  
        mOuYangFeng.attackMagic();  
    }  
}
```

2. 装饰模式的使用场景和优缺点

• 使用场景：

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 需要动态地给一个对象增加功能，这些功能可以动态地撤销。
- 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。

• 优点：

- 通过组合而非继承的方式，动态地扩展一个对象的功能，在运行时选择不同的装饰器，从而实现不同的行为。
- 有效避免了使用继承的方式扩展对象功能而带来的灵活性差、子类无限制扩张的问题。
- 具体组件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体组件类和具体装饰类，在使用时再对其进行组合，原有代码无须改变，符合“开放封闭原则”。

• 缺点：

- 因为所有对象均继承于 Component，所以如果 Component 内部结构发生改变，则不可避免地影响所有子类（装饰者和被装饰者）。如果基类改变，则势必影响对象的内部。
- 比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难。对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。所以，只在必要的时候使用装饰模式。
- 装饰层数不能过多，否则会影响效率。

6.4.3 外观模式

外观模式也被称为门面模式。当我们开发 Android 的时候，无论是做 SDK 还是封装 API，

大多都会用到外观模式，它通过一个外观类使得整个系统的结构只有一个统一的高层接口，这样能降低用户的使用成本。

定义：要求一个子系统的外部与内部的通信必须通过一个统一的对象进行。此模式提供一个高层的接口，使得子系统更易于使用。

外观模式的结构图如图 6-7 所示。

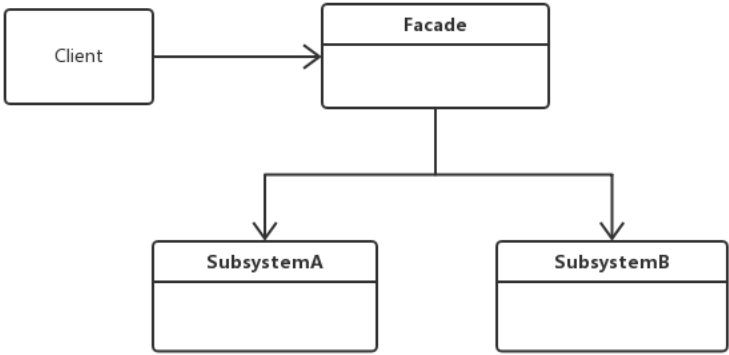


图 6-7 外观模式的结构图

在外观模式中有如下角色。

- **Facade:** 外观类，知道哪些子系统类负责处理请求，将客户端的请求代理给适当的子系统对象。
- **Subsystem:** 子系统类，可以有一个或者多个子系统。实现子系统的功能，处理外观类指派的任务，注意子系统类不含有外观类的引用。

1. 外观模式的简单实现

前面列举了武侠的例子，本节还举武侠的例子。首先，我们把武侠张无忌当作一个系统。张无忌作为一个武侠，他本身分为 3 个系统，分别是招式、内功和经脉。

(1) 子系统类

```
/**
 * 子系统招式
 */
public class ZhaoShi {
    public void TaiJiQuan() {
        System.out.print("使用招式太极拳");
    }
    public void QiShangQuan() {
        System.out.print("使用招式七伤拳");
    }
    public void ShengHuo() {
        System.out.print("使用招式圣火令");
    }
}
```

```

    }
}
/**
 * 子系统内功
 */
public class NeiGong {
    public void JiuYang(){
        System.out.print("使用九阳神功");
    }
    public void QianKun(){
        System.out.print("使用乾坤大挪移");
    }
}
/**
 * 子系统经脉
 */
public class JingMai {
    public void jingmai(){
        System.out.print("开启经脉");
    }
}

```

张无忌有很多武学和内功，怎么将它们搭配并对外界隐藏呢？接下来查看外观类。

(2) 外观类

这里的外观类就是张无忌，他负责将自己的招式、内功和经脉通过不同的情况合理地运用，代码如下所示：

```

/**
 * 外观类张无忌
 */
public class ZhangWuJi {
    private JingMai jingMai;
    private ZhaoShi zhaoShi;
    private NeiGong neiGong;
    public ZhangWuJi(){
        jingMai=new JingMai();
        zhaoShi=new ZhaoShi();
        neiGong=new NeiGong();
    }
    /**
     * 使用乾坤大挪移
     */
    public void Qiankun(){

```

```

        jingMai.jingmai(); //开启经脉
        neiGong.QianKun(); //使用内功乾坤大挪移
    }
    /**
     * 使用七伤拳
     */
    public void QiShang(){
        jingMai.jingmai(); //开启经脉
        neiGong.JiuYang(); //使用内功九阳神功
        zhaoShi.QiShangQuan(); //使用招式七伤拳
    }
}

```

初始化外观类的同时将各个子系统类创建好。很明显，张无忌很好地将自身的各个系统进行了搭配。如果其使用七伤拳，就需要开启经脉、使用内功九阳神功，接下来使用招式七伤拳；如果其不开启经脉或者不使用九阳神功，那么七伤拳的威力会大打折扣。

(3) 客户端调用

```

public class Client{
    public static void main(String[] args){
        ZhangWuJi zhangWuJi=new ZhangWuJi();
        //张无忌使用乾坤大挪移
        zhangWuJi.Qiankun();
        //张无忌使用七伤拳
        zhangWuJi.QiShang();
    }
}

```

当张无忌使用乾坤大挪移或者七伤拳的时候，比武的对手显然不知道张无忌本身运用了什么，同时张无忌也无须重新计划使用七伤拳的时候要怎么做（他已经早就计划好了）。如果每次使用七伤拳或者乾坤大挪移时都要计划怎么做很显然会增加成本并贻误战机。另外，张无忌也可以改变自己的内功、招式和经脉，这些都是对比武的对手有所隐藏的。外观模式本身就是将子系统的逻辑和交互隐藏起来，为用户提供一个高层次的接口，使得系统更加易用，同时也隐藏了具体的实现。这样即使具体的子系统发生了变化，用户也不会感知到。

2. 外观模式的使用场景和优缺点

• 使用场景：

- 构建一个有层次结构的子系统时，使用外观模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，则可以让其通过外观接口进行通信，减少子系统之间的依赖关系。
- 子系统往往会因为不断地重构演化而变得越来越复杂，大多数的模式使用时也会产生很多很小的类，这给外部调用它们的用户程序带来了使用上的困难。我们可以使

用外观类提供一个简单的接口，对外隐藏子系统的具体实现并隔离变化。

- 当维护一个遗留的大型系统时，可能这个系统已经非常难以维护和拓展；但因为它含有重要的功能，所以新的需求必须依赖于它，这时可以使用外观类，为设计粗糙或者复杂的遗留代码提供一个简单的接口，让新系统和外观类交互，而外观类负责与遗留的代码进行交互。
- **优点：**
 - 减少系统的相互依赖，所有的依赖都是对外观类的依赖，与子系统无关。
 - 对用户隐藏了子系统的具体实现，减少用户对子系统的耦合。这样即使具体的子系统发生了变化，用户也不会感知到。
 - 加强了安全性，子系统的方法如果不在外观类中开通，就无法访问到子系统的方法。
- **缺点：**不符合开放封闭原则。如果业务出现变更，则可能要直接修改外观类。

6.4.4 享元模式

享元模式是池技术的重要实现方式，它可以减少应用程序创建的对象，降低程序内存的占用，提高程序的性能。

定义：使用共享对象有效地支持大量细粒度的对象

要求细粒度对象，那么不可避免地使得对象数量多且性质相近。这些对象分为两个部分：内部状态和外部状态。内部状态是对象可共享出来的信息，存储在享元对象内部并且不会随环境的改变而改变；而外部状态是对象依赖的一个标记，它是随环境改变而改变的并且不可共享的状态。享元模式的结构图如图 6-8 所示。

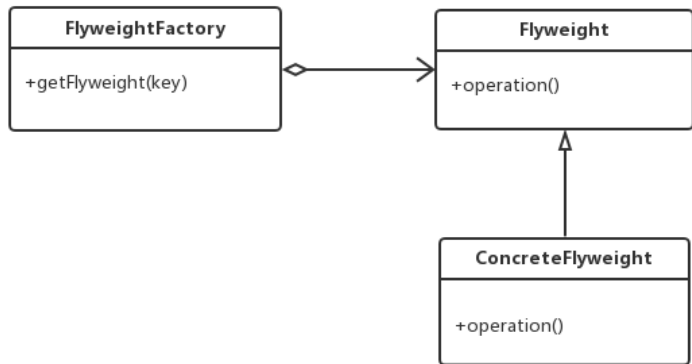


图 6-8 享元模式的结构图

在享元模式中有如下角色。

- **Flyweight:** 抽象享元角色，同时定义出对象的外部状态和内部状态的接口或者实现。
- **ConcreteFlyweight:** 具体享元角色，实现抽象享元角色定义的业务。
- **FlyweightFactory:** 享元工厂，负责管理对象池和创建享元对象。

1. 享元模式的简单实现

某著名网上商城卖商品，如果每个用户下单都生成商品对象，这显然会耗费很多资源。如果赶上“双 11”，那恐怖的订单量会生成很多商品对象，更何况商城卖的商品种类繁多，这样就极易产生“Out Of Memory”。因此，我们采用享元模式来对商品的创建进行优化。

(1) 抽象享元角色

抽象享元角色是一个商品接口，它定义了 `showGoodsPrice` 方法来展示商品的价格：

```
public interface IGoods {
    public void showGoodsPrice(String name);
}
```

(2) 具体享元角色

定义类 `Goods`，它实现 `IGoods` 接口，并实现了 `showGoodsPrice` 方法，如下所示：

```
public class Goods implements IGoods{
    private String name;//名称
    private String version;//版本
    Goods(String name){
        this.name=name;
    }
    @Override
    public void showGoodsPrice(String version) {
        if(version.equals("32G")){
            System.out.println("价格为 5199 元");
        }else if(version.equals("128G")){
            System.out.println("价格为 5999 元");
        }
    }
}
```

其中 `name` 为内部状态，`version` 为外部状态。`showGoodsPrice` 方法根据 `version` 的不同会打印出不同的价格。

(3) 享元工厂

```
public class GoodsFactory {
    private static Map<String, Goods> pool=new HashMap<String, Goods>();
    public static Goods getGoods(String name){
        if(pool.containsKey(name)){
            System.out.println("使用缓存, key 为:"+name);
            return pool.get(name);
        }else{
```

```

        Goods goods=new Goods(name);
        pool.put(name,goods);
        System.out.println("创建商品,key为:"+name);
        return goods;
    }
}
}

```

享元工厂 GoodsFactory 用来创建 Goods 对象。通过 Map 容器来存储 Goods 对象，将内部状态 name 作为 Map 的 key，以便标识 Goods 对象。如果 Map 容器中包含此 key，则使用 Map 容器中存储的 Goods 对象；否则就新建 Goods 对象，并放入 Map 容器中。

(4) 客户端调用

在客户端中调用 GoodsFactory 的 getGoods 方法来创建 Goods 对象，并调用 Goods 的 showGoodsPrice 方法来显示产品的价格，如下所示：

```

public class Client {
    public static void main(String[]args) {
        Goods goods1=GoodsFactory.getGoods("iphone7");
        goods1.showGoodsPrice("32G");
        Goods goods2=GoodsFactory.getGoods("iphone7");
        goods2.showGoodsPrice("32G");
        Goods goods3=GoodsFactory.getGoods("iphone7");
        goods3.showGoodsPrice("128G");
    }
}

```

运行结果如下：

创建商品,key 为:iphone7 价格为 5199 元

使用缓存,key 为:iphone7 价格为 5199 元

使用缓存,key 为:iphone7 价格为 5999 元

从输出可以看出，只有第一次是创建 Goods 对象，后面因为 key 值相同，所以均使用了对象池中的 Goods 对象。在这个例子中，name 作为内部状态是不变的，并且作为 Map 的 key 值是可以共享的。而 showGoodsPrice 方法中需要传入的 version 值则是外部状态，它的值是变化的。

2. 享元模式的使用场景

- 系统中存在大量的相似对象。
- 需要缓冲池的场景。

6.5 行为型设计模式

行为型模式主要处理类或对象如何交互及如何分配职责。它共有 11 种模式：策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式和解释器模式。本节会介绍策略模式、模板方法模式和观察者模式。

6.5.1 策略模式

当我们写代码时总会遇到一种情况，就是我们会有很多的选择，由此衍生出很多的 `if...else`，或者 `case`。如果每个条件语句中包含了一个简单的逻辑，那还比较容易处理；但如果在一个条件语句中又包含了多个条件语句，就会使得代码变得臃肿，维护的成本也会加大，这显然违背了开放封闭原则。本节我们将讲解策略模式，来看看它是如何解决如上所说的问题的。

定义：定义一系列的算法，把每一个算法封装起来，并且使它们可相互替换。策略模式使得算法可独立于使用它的客户而独立变化。

策略模式的结构图如图 6-9 所示。

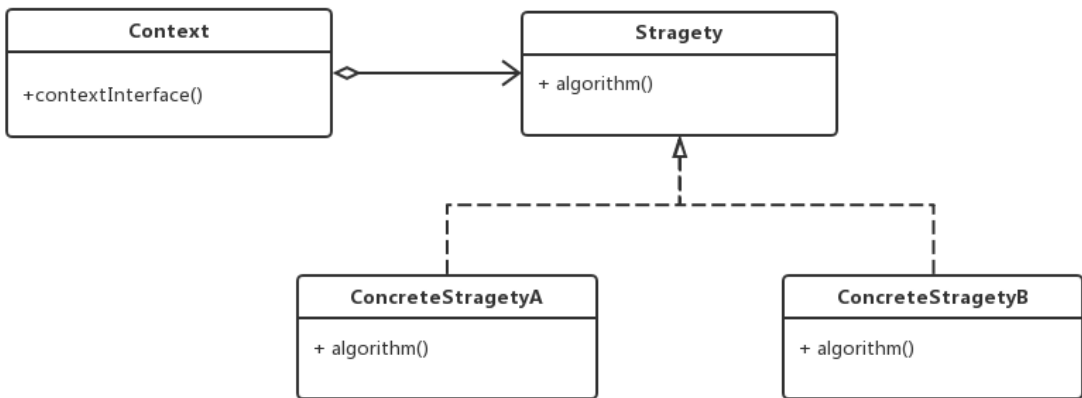


图 6-9 策略模式的结构图

在策略模式中有如下角色。

- **Context：**上下文角色，用来操作策略的上下文环境，起到承上启下的作用，屏蔽高层模块对策略、算法的直接访问。
- **Strategy：**抽象策略角色，策略、算法的抽象，通常为接口。
- **ConcreteStrategy：**具体的策略实现。

1. 策略模式的简单实现

请大家原谅笔者太爱武侠，本节仍旧举武侠的例子。张无忌作为一个大侠会遇到很多对手，如果每遇到一个对手他都用自己最厉害的武功去应战，这显然是不明智的。于是张无忌想出了 3 种应战的策略，分别对付 3 个实力层次的对对手。

(1) 定义策略接口

策略接口有一个 `fighting` 方法用于战斗：

```
public interface FightingStrategy {  
    public void fighting();  
}
```

(2) 具体策略实现

分别定义 3 个策略来实现策略接口，用来对付 3 个实力层次的对手，代码如下所示：

```
public class WeakRivalStrategy implements FightingStrategy {  
    @Override  
    public void fighting() {  
        System.out.println("遇到了较弱的对手，张无忌使用太极剑");  
    }  
}  
  
public class CommonRivalStrategy implements FightingStrategy {  
    @Override  
    public void fighting() {  
        System.out.println("遇到了普通的对手，张无忌使用圣火令神功");  
    }  
}  
  
public class StrongRivalStrategy implements FightingStrategy {  
    @Override  
    public void fighting() {  
        System.out.println("遇到了强大的对手，张无忌使用乾坤大挪移");  
    }  
}
```

(3) 上下文角色

上下文角色的构造方法包含了策略类，通过传进来不同的具体策略来调用不同策略的 `fighting` 方法，如下所示：

```
public class Context {  
    private FightingStrategy fightingStrategy;  
    public Context(FightingStrategy fightingStrategy) {  
        this.fightingStrategy = fightingStrategy;  
    }  
    public void fighting(){  
        fightingStrategy.fighting();  
    }  
}
```

(4) 客户端调用

张无忌对不同实力层次的手，采用了不同的策略来应战。为了举例，这里省略了对不同实力层次进行判断的条件语句，代码如下所示。

```
public class ZhangWuJi {
    public static void main(String[] args) {
        Context context;
        //张无忌遇到对手宋青书，采用对较弱对手的策略
        context = new Context(new WeakRivalStrategy());
        context.fighting();
        //张无忌遇到对手灭绝师太，采用对普通对手的策略
        context = new Context(new CommonRivalStrategy());
        context.fighting();
        //张无忌遇到对手成昆，采用对强大对手的策略
        context = new Context(new StrongRivalStrategy());
        context.fighting();
    }
}
```

上面只是举了一个简单的例子，其实情况会很多：比如遇到普通的手，也不能完全用圣火令神功；比如当遇到周芷若和赵敏时就需要手下留情，采用太极剑；又比如遇到强劲的对手张三丰，由于是自己师公，也不能使用乾坤大挪移。类似这样的情况会很多，这样在每个策略类中可能会出现很多条件语句。但是试想一下如果我们不用策略模式来封装这些条件语句，那么可能会导致一个条件语句中又包含了多个条件语句，这样会使代码变得臃肿，维护的成本也会加大。

2. 策略模式的使用场景和优缺点

• 使用场景：

- 对客户隐藏具体策略（算法）的实现细节，彼此完全独立。
- 针对同一类型问题的多种处理方式，仅仅是具体行为有差别时。
- 在一个类中定义了很多行为，而且这些行为在这个类里的操作以多个条件语句的形式出现。策略模式将相关的条件分支移入它们各自的 `Strategy` 类中，以代替这些条件语句。

• 优点：

- 使用策略模式可以避免使用多重条件语句。多重条件语句不易维护，而且易出错。
- 易于拓展。当需要添加一个策略时，只需要实现接口就可以了。

• 缺点：

- 每一个策略都是一个类，复用性小。如果策略过多，类的数量会增多。
- 上层模块必须知道有哪些策略，才能够使用这些策略，这与迪米特原则相违背。

6.5.2 模板方法模式

在软件开发中，有时会遇到类似的情况：某个方法的实现需要多个步骤，其中有些步骤是固定的；而有些步骤并不固定，存在可变性。为了提高代码的复用性和系统的灵活性，可以使用模板方法模式来应对这类情况。

定义：定义一个操作中的算法框架，而将一些步骤延迟到子类中，使得子类不改变一个算法的结构即可重定义算法的某些特定步骤。

模板方法模式的结构图如图 6-10 所示。

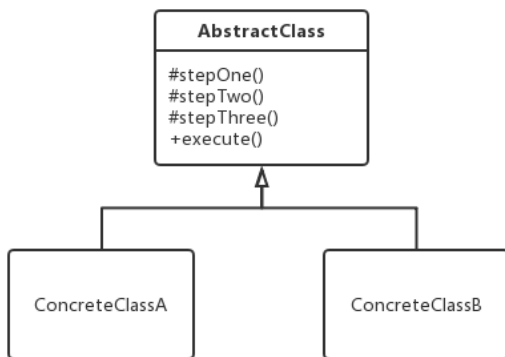


图 6-10 模板方法模式的结构图

在模板方法模式中有如下角色。

- **AbstractClass**：抽象类，定义了一套算法框架。
- **ConcreteClass**：具体实现类。

1. 模板方法的简单实现

模板方法实际就是封装固定的流程，像模板一样，第一步做什么，第二步又做什么，都在抽象类中定义好。子类可以有不同的算法实现，在算法框架不被修改的前提下实现某些步骤的算法替换。

(1) 创建抽象类，定义算法框架

接着举武侠的例子。一个武侠要战斗的时候，也有一套固定的通用模式，那就是运行内功、开启经脉、准备武器和使用招式，我们把这些用代码表示，如下所示：

```

public abstract class AbstractSwordsman {
    //该方法为 final，防止算法框架被覆写
    public final void fighting(){
        //运行内功，抽象方法
        neigong();
        //调整经脉，具体方法
        meridian();
    }
}
  
```

```

        //如果有武器，则准备武器
        if(hasWeapons()){//2
            weapons();
        }
        //使用招式
        moves();
        //钩子方法
        hook();//1
    }
    //空实现方法
    protected void hook(){
    protected abstract void neigong();
    protected abstract void weapons();
    protected abstract void moves();
    protected void meridian(){
        System.out.println("开启正经与奇经");
    }
    /**
     * 是否有武器，默认是有武器的，钩子方法
     * @return
     */
    protected boolean hasWeapons(){
        return true;
    }
}

```

这个抽象类包含了 3 种类型的方法，分别是抽象方法、具体方法和钩子方法。抽象方法是交由子类去实现的，具体方法则是父类实现了子类公共的方法。在上面的例子中就是武侠开启经脉的方式都一样，所以就在具体方法中实现。钩子方法则分为两类：第一类在上面代码注释 1 处，它有一个空实现的方法，子类可以视情况来决定是否要覆盖它；第二类在注释 2 处，这类钩子方法的返回类型通常是 `boolean` 类型的，其一般用于对某个条件进行判断，如果条件满足则执行某一步骤，否则将不执行。

（2）具体实现类

武侠就拿张无忌、张三丰来作为例子，代码如下所示：

```

public class ZhangWuJi extends AbstractSwordsman {
    @Override
    protected void neigong() {
        System.out.println("运行九阳神功");
    }
    @Override
    protected void weapons() {

```

```

    }
    @Override
    protected void moves() {
        System.out.println("使用招式乾坤大挪移");
    }
    @Override
    protected boolean hasWeapons() {
        return false;
    }
}

```

张无忌没有武器，所以 `hasWeapons` 方法返回 `false`，这样也不会进入 `weapons` 方法了。接下来看张三丰的代码：

```

public class ZhangSanFeng extends AbstractSwordsman {
    @Override
    protected void neigong() {
        System.out.println("运行纯阳无极功");
    }
    @Override
    protected void weapons() {
        System.out.println("使用真武剑");
    }
    @Override
    protected void moves() {
        System.out.println("使用招式神门十三剑");
    }
    @Override
    protected void hook() {
        System.out.println("突然肚子不舒服，老夫先去趟厕所");
    }
}

```

最后，张三丰突然感觉肚子不舒服，所以就实现了钩子方法 `hook`，用来处理一些自定义的逻辑。

(3) 客户端调用

```

public class Client {
    public static void main(String[] args) {
        ZhangWuJi zhangWuJi=new ZhangWuJi();
        zhangWuJi.fighting();
        ZhangSanFeng zhangSanFeng=new ZhangSanFeng();
        zhangSanFeng.fighting();
    }
}

```


2. 模板方法模式的使用场景和优缺点

- 使用场景：
 - 多个子类有共有的方法，并且逻辑基本相同时。
 - 面对重要、复杂的算法，可以把核心算法设计为模板方法，周边相关细节功能则由各个子类实现。
 - 需要通过子类来决定父类算法中的某个步骤是否执行，实现子类对父类的反向控制。
- 优点：
 - 模板方法模式通过把不变的行为搬移到超类，去除了子类中的重复代码。
 - 子类实现算法的某些细节，有助于算法的扩展。
- 缺点：
 - 每个不同的实现都需要定义一个子类，这会导致类的个数的增加，设计更加抽象。

6.5.3 观察者模式

观察者模式又被称为发布-订阅模式，属于行为型设计模式的一种，是一个在项目中经常使用的模式。它的定义如下。

定义：定义对象间一种一对多的依赖关系，每当一个对象改变状态时，则所有依赖于它的对象都会得到通知并被自动更新。

观察者模式的结构图如图 6-11 所示。

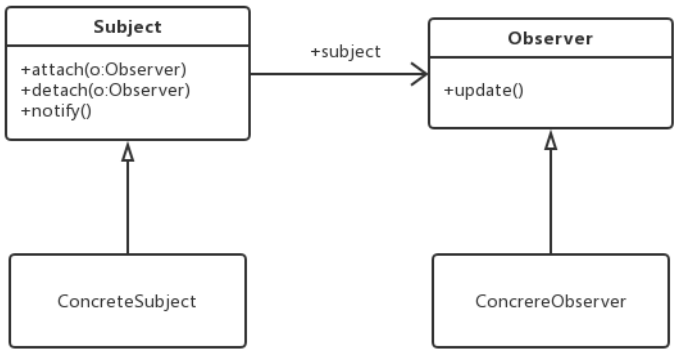


图 6-11 观察者模式的结构图

在观察者模式中有如下角色。

- **Subject**: 抽象主题（抽象被观察者）。抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。
- **ConcreteSubject**: 具体主题（具体被观察者）。该角色将有关状态存入具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知。
- **Observer**: 抽象观察者，是观察者的抽象类。它定义了一个更新接口，使得在得到主题

更改通知时更新自己。

- **ConcreteObserver**: 具体观察者，实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态。

1. 观察者模式的简单实现

关于观察者模式这种发布-订阅的形式，我们可以拿微信公众号来举例。假设微信用户就是观察者，微信公众号是被观察者，有多个微信用户关注了“程序猿”这个公众号，当这个公众号更新时就会通知这些订阅的微信用户。接下来用代码实现，如下所示：

(1) 抽象观察者

里面只定义了一个更新的方法：

```
public interface Observer {
    public void update(String message);
}
```

(2) 具体观察者

微信用户是观察者，里面实现了更新的方法，如下所示：

```
public class WeixinUser implements Observer {
    // 微信用户名
    private String name;
    public WeixinUser(String name) {
        this.name = name;
    }
    @Override
    public void update(String message) {
        System.out.println(name + "-" + message);
    }
}
```

(3) 抽象被观察者

抽象被观察者，提供了 **attach**、**detach**、**notify** 三个方法，如下所示：

```
public interface Subject {
    /**
     * 增加订阅者
     * @param observer
     */
    public void attach(Observer observer);
    /**
     * 删除订阅者
     * @param observer
     */
}
```

```

    */
    public void detach(Observer observer);
    /**
     * 通知订阅者更新消息
     */
    public void notify(String message);
}

```

(4) 具体被观察者

微信公众号是具体主题（具体被观察者），里面存储了订阅该公众号的微信用户，并实现了抽象主题中的方法：

```

public class SubscriptionSubject implements Subject {
    //存储订阅公众号的微信用户
    private List<Observer> weixinUserlist = new ArrayList<Observer>();
    @Override
    public void attach(Observer observer) {
        weixinUserlist.add(observer);
    }
    @Override
    public void detach(Observer observer) {
        weixinUserlist.remove(observer);
    }
    @Override
    public void notify(String message) {
        for (Observer observer : weixinUserlist) {
            observer.update(message);
        }
    }
}

```

(5) 客户端调用

```

public class Client {
    public static void main(String[] args) {
        SubscriptionSubject mSubscriptionSubject=new SubscriptionSubject();
        //创建微信用户
        WeixinUser user1=new WeixinUser("杨影枫");
        WeixinUser user2=new WeixinUser("月眉儿");
        WeixinUser user3=new WeixinUser("紫轩");
        //订阅公众号
        mSubscriptionSubject.attach(user1);
        mSubscriptionSubject.attach(user2);
        mSubscriptionSubject.attach(user3);
        //公众号更新发出消息给订阅的微信用户
    }
}

```

```
mSubscriptionSubject.notify("刘望舒的专栏更新了");  
}  
}
```

2. 观察者模式的使用场景和优缺点

- 使用场景：
 - 关联行为场景。需要注意的是，关联行为是可拆分的，而不是“组合”关系。
 - 事件多级触发场景。
 - 跨系统的消息交换场景，如消息队列、事件总线的处理机制。
- 优点：
 - 观察者和被观察者之间是抽象耦合，容易扩展。
 - 方便建立一套触发机制。
- 缺点：在应用观察者模式时需要考虑一下开发效率和运行效率的问题。程序中包括一个被观察者、多个观察者，开发、调试等内容会比较复杂，而且在 Java 中消息的通知一般是顺序执行的，那么一个观察者卡顿，会影响整体的执行效率，在这种情况下，一般会采用异步方式。

6.6 本章小结

本章通过设计模式的分类，对每个分类中常用的设计模式进行了讲解。由于篇幅限制，因此还有很多设计模式没有介绍，这就需要大家在业余时间自行学习。需要注意的是学习设计模式最忌讳生搬硬套，为了设计模式而设计。设计模式主要解决的问题就是设计模式的六大原则，只要我们设计的代码遵循这六大原则，那么就是优秀的代码。

第 7 章

事件总线

为了简化并且更加高质量地在 Activity、Fragment、Thread 和 Service 等之间的通信，同时解决组件之间高耦合的同时仍能继续高效地通信，事件总线设计出现了。提到事件总线我们会想到 EventBus 和 otto，所以本章就来讲解它们的使用方法以及原理。

7.1 解析 EventBus

EventBus 是一款针对 Android 优化的发布-订阅事件总线。它简化了应用程序内各组件间、组件与后台线程间的通信。其优点是开销小，代码更优雅，以及将发送者和接收者解耦。如果 Activity 和 Activity 进行交互还好说，但如果 Fragment 和 Fragment 进行交互则着实令人头疼。这时我们会使用广播来处理，但是使用广播略嫌麻烦并且效率也不高。如果传递的数据是实体类，需要序列化，那么传递的成本会有点高。

7.1.1 使用 EventBus

在讲到 EventBus 的基本用法之前，我们需要了解 EventBus 的三要素以及它的 4 种 ThreadMode。

EventBus 的三要素如下。

- Event：事件。可以是任意类型的对象。
- Subscriber：事件订阅者。在 EventBus 3.0 之前消息处理的方法只能限定于 onEvent、onEventMainThread、onEventBackgroundThread 和 onEventAsync，它们分别代表 4 种线程模型。而在 EventBus 3.0 之后，事件处理的方法可以随便取名，但是需要添加一个注解 @Subscribe，并且要指定线程模型（默认为 POSTING）。4 种线程模型下面会讲到。

- **Publisher**: 事件发布者。可以在任意线程任意位置发送事件，直接调用 EventBus 的 `post(Object)` 方法。可以自己实例化 EventBus 对象，但一般使用 `EventBus.getDefault()` 就可以。根据 `post` 函数参数的类型，会自动调用订阅相应类型事件的函数。

EventBus 的 4 种 ThreadMode（线程模型）如下。

- **POSTING**（默认）：如果使用事件处理函数指定了线程模型为 POSTING，那么该事件是在哪个线程发布出来的，事件处理函数就会在哪个线程中运行，也就是说发布事件和接收事件在同一个线程中。在线程模型为 POSTING 的事件处理函数中尽量避免执行耗时操作，因为它会阻塞事件的传递，甚至有可能引起 ANR。
- **MAIN**：事件的处理会在 UI 线程中执行。事件处理的时间不能太长，长了会导致 ANR。
- **BACKGROUND**：如果事件是在 UI 线程中发布出来的，那么该事件处理函数就会在新的线程中运行；如果事件本来就是在子线程中发布出来的，那么该事件处理函数直接在发布事件的线程中执行。在此事件处理函数中禁止进行 UI 更新操作。
- **ASYNC**：无论事件在哪个线程中发布，该事件处理函数都会在新建的子线程中执行；同样，此事件处理函数中禁止进行 UI 更新操作。

1. EventBus 基本用法

EventBus 使用起来分为以下 5 个步骤。

（1）自定义一个事件类

```
public class MessageEvent {
    ...
}
```

（2）在需要订阅事件的地方注册事件

```
EventBus.getDefault().register(this);
```

（3）发送事件

```
EventBus.getDefault().post(messageEvent);
```

（4）处理事件

```
@Subscribe (threadMode = ThreadMode.MAIN)
public void XXX(MessageEvent messageEvent) {
    ...
}
```

前面说过，消息处理的方法可以随便取名，但是需要添加一个注解 `@Subscribe`，并且要指定线程模型（默认为 POSTING）。

(5) 取消事件订阅

```
EventBus.getDefault().unregister(this);
```

2. EventBus 应用举例

前面讲到了 EventBus 的基本用法，但是这过于简单，这里举一个例子来应用 EventBus。

(1) 添加依赖库

首先配置 gradle，如下所示：

```
compile 'org.greenrobot:eventbus:3.0.0'
```

(2) 定义消息事件类

```
public class MessageEvent {
    private String message;
    public MessageEvent(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

(3) 注册和取消订阅事件

在 MainActivity 中注册和取消订阅事件，如下所示：

```
public class MainActivity extends AppCompatActivity {
    private TextView tv_message;
    private Button bt_message;
    private Button bt_subscription;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tv_message=(TextView) this.findViewById(R.id.tv_message);
        tv_message.setText("MainActivity");
        bt_subscription=(Button) this.findViewById(R.id.bt_subscription);
        bt_subscription.setText("注册事件");
        bt_message=(Button) this.findViewById(R.id.bt_message);
        bt_message.setText("跳转到 SecondActivity");
        bt_message.setOnClickListener(new View.OnClickListener() {
```

```

        @Override
        public void onClick(View v) {
            startActivity(new Intent(MainActivity.this, SecondActivity.
                class));
        }
    });
    bt_subscription.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            //注册事件
            EventBus.getDefault().register(MainActivity.this);
        }
    });
}
@Override
protected void onDestroy() {
    super.onDestroy();
    //取消注册事件
    EventBus.getDefault().unregister(this);
}
}

```

在 `MainActivity` 中定义了两个 `Button`：一个用来注册事件，另一个用来跳转到 `SecondActivity`。

（4）事件订阅者处理事件

在 `MainActivity` 中自定义方法来处理事件，在这里 `ThreadMode` 设置为 `MAIN`，事件的处理会在 UI 线程中执行，用 `TextView` 来展示收到的事件消息：

```

@Subscribe(threadMode = ThreadMode.MAIN)
public void onMoonEvent(MessageEvent messageEvent) {
    tv_message.setText(messageEvent.getMessage());
}

```

（5）事件发布者发布事件

创建了 `SecondActivity` 来发布消息，代码如下所示：

```

public class SecondActivity extends AppCompatActivity {
    private Button bt_message;
    private TextView tv_message;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}

```



```

        setContentView(R.layout.activity_main);
        tv_message=(TextView)this.findViewById(R.id.tv_message);
        tv_message.setText("SecondActivity");
        bt_message=(Button)this.findViewById(R.id.bt_message);
        bt_message.setText("发送事件");
        bt_message.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                EventBus.getDefault().post(new MessageEvent("欢迎关注刘望舒的博客"));
                finish();
            }
        });
    }
}

```

在 SecondActivity 中，我们定义“发送事件”按钮来发送事件并将 SecondActivity finish 掉。好了，运行程序，如图 7-1 所示。接下来我们点击 MainActivity 中的“注册事件”按钮来注册事件，然后点击“跳转到 SECONDACTIVITY”按钮，这时跳转到 SecondActivity，如图 7-2 所示。接下来点击“发送事件”按钮，这个时候 SecondActivity 被 finish 掉，因此界面展示的是 MainActivity，如图 7-3 所示。可以看到 MainActivity 的 TextView 显示“欢迎关注刘望舒的博客”，MainActivity 成功地收到了 SecondActivity 发送的事件。

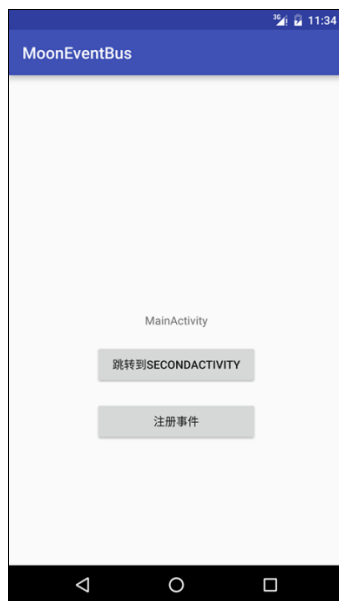


图 7-1 程序初始样式



图 7-2 跳转到 SecondActivity



图 7-3 MainActivity 接收到事件

(6) ProGuard 混淆规则

最后不要忘了在 ProGuard 中加入如下混淆规则：

```
-keepattributes *Annotation*
-keepclassmembers class ** {
    @org.greenrobot.eventbus.Subscribe <methods>;
}
-keep enum org.greenrobot.eventbus.ThreadMode { *; }
# Only required if you use AsyncExecutor
-keepclassmembers class * extends org.greenrobot.eventbus.util.
ThrowableFailureEvent {
    <init>(java.lang.Throwable);
}
```

3. EventBus 的黏性事件

除了上面讲的普通事件外，EventBus 还支持发送黏性事件，就是在发送事件之后再订阅该事件也能收到该事件，这跟黏性广播类似。为了验证黏性事件，我们修改以前的代码，如下所示。

(1) 订阅者处理黏性事件

在 MainActivity 中新写一个方法用来处理黏性事件：

```
Subscribe(threadMode = ThreadMode.POSTING, sticky = true)
public void ononMoonStickyEvent(MessageEvent messageEvent) {
    tv_message.setText(messageEvent.getMessage());
}
```

(2) 发送黏性事件

在 `SecondActivity` 中定义一个 `Button` 来发送黏性事件：

```
bt_subscription.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        EventBus.getDefault().postSticky(new MessageEvent("黏性事件"));
        finish();
    }
});
```

现在运行代码再来看看效果。首先，我们在 `MainActivity` 中并没有点击“注册事件”按钮，而是直接跳到 `SecondActivity` 中点击发送“黏性事件”按钮。这时界面回到 `MainActivity`，我们看到 `TextView` 仍旧显示着 `MainActivity` 的字段，这是因为我们现在还没有订阅事件。接下来我们点击“注册事件”按钮，`TextView` 内容发生改变，显示“黏性事件”，说明黏性事件被成功接收到了。

7.1.2 源码解析 EventBus

前面讲到了 `EventBus` 的用法，本节我们将讲解 `EventBus` 的源码。

1. EventBus 构造方法

当我们要使用 `EventBus` 时，首先会调用 `EventBus.getDefault()` 来获取 `EventBus` 实例。现在查看 `getDefault` 方法做了什么，如下所示：

```
public static EventBus getDefault() {
    if (defaultInstance == null) {
        synchronized (EventBus.class) {
            if (defaultInstance == null) {
                defaultInstance = new EventBus();
            }
        }
    }
    return defaultInstance;
}
```

很明显这是一个单例模式，采用了双重检查模式（DCL）。接下来查看 `EventBus` 的构造方法做了什么：

```
public EventBus() {
    this(DEFAULT_BUILDER);
}
```

这里 DEFAULT_BUILDER 是默认的 EventBusBuilder，用来构造 EventBus：

```
private static final EventBusBuilder DEFAULT_BUILDER = new EventBusBuilder();
```

this 调用了 EventBus 的另一个构造方法，如下所示：

```
EventBus(EventBusBuilder builder) {
    subscriptionsByEventType = new HashMap<>();
    typesBySubscriber = new HashMap<>();
    stickyEvents = new ConcurrentHashMap<>();
    mainThreadPoster = new HandlerPoster(this, Looper.getMainLooper(), 10);
    backgroundPoster = new BackgroundPoster(this);
    asyncPoster = new AsyncPoster(this);
    indexCount = builder.subscriberInfoIndexes != null ? builder.
        subscriberInfoIndexes.size() : 0;
    subscriberMethodFinder = new SubscriberMethodFinder(builder.
        subscriberInfoIndexes, builder.strictMethodVerification,
        builder.ignoreGeneratedIndex);
    logSubscriberExceptions = builder.logSubscriberExceptions;
    logNoSubscriberMessages = builder.logNoSubscriberMessages;
    sendSubscriberExceptionEvent = builder.sendSubscriberExceptionEvent;
    sendNoSubscriberEvent = builder.sendNoSubscriberEvent;
    throwSubscriberException = builder.throwSubscriberException;
    eventInheritance = builder.eventInheritance;
    executorService = builder.executorService;
}
```

我们可以通过构造一个 EventBusBuilder 来对 EventBus 进行配置，这里采用了建造者模式。

2. 订阅者注册

获取 EventBus 后，便可以将订阅者注册到 EventBus 中。下面来看一下 register 方法：

```
public void register(Object subscriber) {
    Class<?> subscriberClass = subscriber.getClass();
    List<SubscriberMethod>
subscriberMethods = subscriberMethodFinder.findSubscriberMethods
(subscriberClass); //1
    synchronized (this) {
        for (SubscriberMethod subscriberMethod : subscriberMethods) {
            subscribe(subscriber, subscriberMethod); //2
        }
    }
}
```

(1) 查找订阅者的订阅方法

上面代码注释 1 处的 `findSubscriberMethods` 方法找出一个 `SubscriberMethod` 的集合，也就是传进来的订阅者的所有订阅方法，接下来遍历订阅者的订阅方法来完成订阅者的注册操作。可以看出 `register` 方法做了两件事：一件事是查找订阅者的订阅方法，另一件事是订阅者的注册。在 `SubscriberMethod` 类中，主要用来保存订阅方法的 `Method` 对象、线程模式、事件类型、优先级、是否是黏性事件等属性。下面就来查看 `findSubscriberMethods` 方法，如下所示：

```
List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass) {
    List<SubscriberMethod> subscriberMethods = METHOD_CACHE.get
        (subscriberClass); //1
    if (subscriberMethods != null) {
        return subscriberMethods;
    }
    if (ignoreGeneratedIndex) {
        subscriberMethods = findUsingReflection(subscriberClass);
    } else {
        subscriberMethods = findUsingInfo(subscriberClass); //3
    }
    if (subscriberMethods.isEmpty()) {
        throw new EventBusException("Subscriber " + subscriberClass
            + " and its super classes have no public methods with the
            @Subscribe annotation");
    } else {
        METHOD_CACHE.put(subscriberClass, subscriberMethods); //2
        return subscriberMethods;
    }
}
```

上面代码注释 1 处从缓存中查找是否有订阅方法的集合，如果找到了就立马返回。如果缓存中没有，则根据 `ignoreGeneratedIndex` 属性的值来选择采用何种方法来查找订阅方法的集合。`ignoreGeneratedIndex` 属性表示是否忽略注解器生成的 `MyEventBusIndex`。如何生成 `MyEventBusIndex` 类以及它的使用，可以参考官方文档 <http://greenrobot.org/eventbus/documentation/subscriber-index/>，这里就不再讲解了。`ignoreGeneratedIndex` 的默认值是 `false`，可以通过 `EventBusBuilder` 来设置它的值。在注释 2 处找到订阅方法的集合后，放入缓存，以免下次继续查找。我们在项目中经常通过 `EventBus` 单例模式来获取默认的 `EventBus` 对象，也就是 `ignoreGeneratedIndex` 为 `false` 的情况，这种情况调用了注释 3 处的 `findUsingInfo` 方法：

```
private List<SubscriberMethod> findUsingInfo(Class<?> subscriberClass) {
    FindState findState = prepareFindState();
    findState.initForSubscriber(subscriberClass);
    while (findState.clazz != null) {
        findState.subscriberInfo = getSubscriberInfo(findState); //1
```

```

        if (findState.subscriberInfo != null) {
            SubscriberMethod[] array = findState.subscriberInfo.
                getSubscriberMethods(); //2
            for (SubscriberMethod subscriberMethod : array) {
                if (findState.checkAdd(subscriberMethod.method,
                    subscriberMethod.eventType)) {
                    findState.subscriberMethods.add(subscriberMethod);
                }
            }
        } else {
            findUsingReflectionInSingleClass(findState); //3
        }
        findState.moveToSuperclass();
    }
    return getMethodsAndRelease(findState);
}

```

上面代码注释 1 处通过 `getSubscriberInfo` 方法来获取订阅者信息。在我们开始查找订阅方法的时候并没有忽略注解器为我们生成的索引 `MyEventBusIndex`。如果我们通过 `EventBusBuilder` 配置了 `MyEventBusIndex`，便会获取 `subscriberInfo`。注释 2 处调用 `subscriberInfo` 的 `getSubscriberMethods` 方法便可以得到订阅方法相关的信息。如果没有配置 `MyEventBusIndex`，便会执行注释 3 处的 `findUsingReflectionInSingleClass` 方法，将订阅方法保存到 `findState` 中。最后再通过 `getMethodsAndRelease` 方法对 `findState` 做回收处理并返回订阅方法的 `List` 集合。默认情况下是没有配置 `MyEventBusIndex` 的，因此现在查看一下 `findUsing- ReflectionInSingleClass` 方法的执行过程，如下所示：

```

private void findUsingReflectionInSingleClass(FindState findState) {
    Method[] methods;
    try {
        methods = findState.clazz.getDeclaredMethods(); //1
    } catch (Throwable th) {
        methods = findState.clazz.getMethods();
        findState.skipSuperClasses = true;
    }
    for (Method method : methods) {
        int modifiers = method.getModifiers();
        if ((modifiers & Modifier.PUBLIC) != 0 && (modifiers & MODIFIERS_
            IGNORE) == 0) {
            Class<?>[] parameterTypes = method.getParameterTypes();
            if (parameterTypes.length == 1) {
                Subscribe subscribeAnnotation = method.getAnnotation(Subscribe.class);
                if (subscribeAnnotation != null) {
                    Class<?> eventType = parameterTypes[0];

```

```

        if (findState.checkAdd(method, eventType)) {
            ThreadMode threadMode = subscribeAnnotation.threadMode();
            findState.subscriberMethods.add(new SubscriberMethod
                (method, eventType, threadMode, subscribeAnnotation.priority(),
                subscribeAnnotation.sticky()));
        }
    }
}
...
}
}
}

```

上面代码注释 1 处通过反射来获取订阅者中所有的方法，并根据方法的类型、参数和注解来找到订阅方法。找到订阅方法后将订阅方法的相关信息保存到 `findState` 中。

(2) 订阅者的注册过程

在查找完订阅者的订阅方法以后便开始对所有的订阅方法进行注册。我们再回到 `register` 方法中（在第 315 页），在那里的注释 2 处调用了 `subscribe` 方法来对订阅方法进行注册，如下所示：

```

private void subscribe(Object subscriber, SubscriberMethod subscriberMethod) {
    Class<?> eventType = subscriberMethod.eventType;
    Subscription newSubscription = new Subscription(subscriber,
        subscriberMethod); //1
    CopyOnWriteArrayList<Subscription> subscriptions =
        subscriptionsByEventType.get(eventType); //2
    if (subscriptions == null) {
        subscriptions = new CopyOnWriteArrayList<>();
        subscriptionsByEventType.put(eventType, subscriptions);
    } else {
        //判断订阅者是否已经被注册
        if (subscriptions.contains(newSubscription)) {
            throw new EventBusException("Subscriber " + subscriber.
                getClass() + " already registered to event " + eventType);
        }
    }
    int size = subscriptions.size();
    for (int i = 0; i <= size; i++) {
        if (i == size || subscriberMethod.priority > subscriptions.get(i).
            subscriberMethod.priority) {
            subscriptions.add(i, newSubscription); //3
            break;
        }
    }
}

```

```

    }
}
List<Class<?>> subscribedEvents = typesBySubscriber.get(subscriber); //4
if (subscribedEvents == null) {
    subscribedEvents = new ArrayList<>();
    typesBySubscriber.put(subscriber, subscribedEvents);
}
subscribedEvents.add(eventType);
if (subscriberMethod.sticky) {
    if (eventInheritance) {
        //黏性事件的处理
        Set<Map.Entry<Class<?>, Object>> entries = stickyEvents.entrySet();
        for (Map.Entry<Class<?>, Object> entry : entries) {
            Class<?> candidateEventType = entry.getKey();
            if (eventType.isAssignableFrom(candidateEventType)) {
                Object stickyEvent = entry.getValue();
                checkPostStickyEventToSubscription(new Subscription,
                    stickyEvent);
            }
        }
    } else {
        Object stickyEvent = stickyEvents.get(eventType);
        checkPostStickyEventToSubscription(new Subscription,
            stickyEvent);
    }
}
}
}

```

首先，上面代码注释 1 处会根据 subscriber（订阅者）和 subscriberMethod（订阅方法）创建一个 Subscription（订阅对象）。注释 2 处根据 eventType（事件类型）获取 Subscriptions（订阅对象集合）。如果 Subscriptions 为 null 则重新创建，并将 Subscriptions 根据 eventType 保存在 subscriptionsByEventType（Map 集合）。注释 3 处按照订阅方法的优先级插入到订阅对象集合中，完成订阅方法的注册。注释 4 处通过 subscriber 获取 subscribedEvents（事件类型集合）。如果 subscribedEvents 为 null 则重新创建，并将 eventType 添加到 subscribedEvents 中，并根据 subscriber 将 subscribedEvents 存储在 typesBySubscriber（Map 集合）。如果是黏性事件，则从 stickyEvents 事件保存队列中取出该事件类型的事件发送给当前订阅者。总结一下，subscribe 方法主要就是做了两件事：一件事是将 Subscriptions 根据 eventType 封装到 subscriptionsByEventType 中，将 subscribedEvents 根据 subscriber 封装到 typesBySubscriber 中；第二件事就是对黏性事件的处理。

3. 事件的发送

在获取 EventBus 对象以后，可以通过 post 方法来进行对事件的提交。post 方法的源码如下所示：


```

public void post(Object event) {
    //PostingThreadState 保存着事件队列和线程状态信息
    PostingThreadState postingState = currentPostingThreadState.get();
    //获取事件队列，并将当前事件插入事件队列
    List<Object> eventQueue = postingState.eventQueue;
    eventQueue.add(event);
    if (!postingState.isPosting) {
        postingState.isMainThread = Looper.getMainLooper() == Looper.
            myLooper();
        postingState.isPosting = true;
        if (postingState.canceled) {
            throw new EventBusException("Internal error. Abort state was
                not reset");
        }
        try {
            //处理队列中的所有事件
            while (!eventQueue.isEmpty()) {
                postSingleEvent(eventQueue.remove(0), postingState);
            }
        } finally {
            postingState.isPosting = false;
            postingState.isMainThread = false;
        }
    }
}
}

```

首先从 `PostingThreadState` 对象中取出事件队列，然后再将当前的事件插入事件队列。最后将队列中的事件依次交由 `postSingleEvent` 方法进行处理，并移除该事件。之后查看 `postSingleEvent` 方法里做了什么：

```

private void postSingleEvent(Object event, PostingThreadState postingState)
    throws Error {
    Class<?> eventClass = event.getClass();
    boolean subscriptionFound = false;
    //eventInheritance 表示是否向上查找事件的父类，默认为 true
    if (eventInheritance) {
        List<Class<?>> eventTypes = lookupAllEventTypes(eventClass);
        int countTypes = eventTypes.size();
        for (int h = 0; h < countTypes; h++) {
            Class<?> clazz = eventTypes.get(h);
            subscriptionFound |= postSingleEventForEventType(event,
                postingState, clazz);
        }
    }
}

```

```

    } else {
        subscriptionFound = postSingleEventForEventType(event, postingState,
            eventClass);
    }
    //找不到该事件时的异常处理
    if (!subscriptionFound) {
        if (logNoSubscriberMessages) {
            Log.d(TAG, "No subscribers registered for event " + eventClass);
        }
        if (sendNoSubscriberEvent && eventClass != NoSubscriberEvent.class
            && eventClass != SubscriberExceptionEvent.class) {
            post(new NoSubscriberEvent(this, event));
        }
    }
}
}

```

`eventInheritance` 表示是否向上查找事件的父类，它的默认值为 `true`，可以通过在 `EventBusBuilder` 中进行配置。当 `eventInheritance` 为 `true` 时，则通过 `lookupAllEventTypes` 找到所有的父类事件并存在 `List` 中，然后通过 `postSingleEventForEventType` 方法对事件逐一处理。`postSingleEventForEventType` 方法的源码如下所示：

```

private boolean postSingleEventForEventType(Object event, PostingThreadState
postingState, Class<?> eventClass) {
    CopyOnWriteArrayList<Subscription> subscriptions;
    synchronized (this) {
        subscriptions = subscriptionsByEventType.get(eventClass); //1
    }
    if (subscriptions != null && !subscriptions.isEmpty()) {
        for (Subscription subscription : subscriptions) { //2
            postingState.event = event;
            postingState.subscription = subscription;
            boolean aborted = false;
            try {
                postToSubscription(subscription, event, postingState.
                    isMainThread);
                aborted = postingState.canceled;
            } finally {
                postingState.event = null;
                postingState.subscription = null;
                postingState.canceled = false;
            }
            if (aborted) {
                break;
            }
        }
    }
}

```

```

    }
    return true;
}
return false;
}

```

上面代码注释 1 处同步取出该事件对应的 Subscriptions（订阅对象集合）。注释 2 处遍历 Subscriptions，将事件 event 和对应的 Subscription（订阅对象）传递给 postingState 并调用 postToSubscription 方法对事件进行处理。接下来查看 postToSubscription 方法：

```

private void postToSubscription(Subscription subscription, Object event,
boolean isMainThread) {
    switch (subscription.subscriberMethod.threadMode) {
        case POSTING:
            invokeSubscriber(subscription, event);
            break;
        case MAIN:
            if (isMainThread) {
                invokeSubscriber(subscription, event);
            } else {
                mainThreadPoster.enqueue(subscription, event);
            }
            break;
        case BACKGROUND:
            if (isMainThread) {
                backgroundPoster.enqueue(subscription, event);
            } else {
                invokeSubscriber(subscription, event);
            }
            break;
        case ASYNC:
            asyncPoster.enqueue(subscription, event);
            break;
        default:
            throw new
IllegalStateException("Unknown thread mode: " + subscription.
subscriberMethod.threadMode);
    }
}

```

取出订阅方法的 threadMode（线程模式），之后根据 threadMode 来分别处理。如果 threadMode 是 MAIN，若提交事件的线程是主线程，则通过反射直接运行订阅的方法；若其不是主线程，则需要 mainThreadPoster 将我们的订阅事件添加到主线程队列中。mainThreadPoster 是 HandlerPoster 类型的，继承自 Handler，通过 Handler 将订阅方法切换到主线程执行。

4. 订阅者取消注册

取消注册则需要调用 `unregister` 方法，如下所示：

```
public synchronized void unregister(Object subscriber) {
    List<Class<?>> subscribedTypes = typesBySubscriber.get(subscriber); //1
    if (subscribedTypes != null) {
        for (Class<?> eventType : subscribedTypes) {
            unsubscribeByEventType(subscriber, eventType); //2
        }
        typesBySubscriber.remove(subscriber); //3
    } else {
        Log.w(TAG, "Subscriber to unregister was not registered before:"
            + subscriber.getClass());
    }
}
```

我们在订阅者注册的过程中讲到过 `typesBySubscriber`，它是一个 `map` 集合。上面代码注释 1 处通过 `subscriber` 找到 `subscribedTypes`（事件类型集合）。注释 3 处将 `subscriber` 对应的 `eventType` 从 `typesBySubscriber` 中移除。注释 2 处遍历 `subscribedTypes`，并调用 `unsubscribeByEventType` 方法：

```
private void unsubscribeByEventType(Object subscriber, Class<?> eventType) {
    List<Subscription> subscriptions = subscriptionsByEventType.get
        (eventType); //1
    if (subscriptions != null) {
        int size = subscriptions.size();
        for (int i = 0; i < size; i++) {
            Subscription subscription = subscriptions.get(i);
            if (subscription.subscriber == subscriber) {
                subscription.active = false;
                subscriptions.remove(i);
                i--;
                size--;
            }
        }
    }
}
```

上面代码注释 1 处通过 `eventType` 来得到对应的 `Subscriptions`（订阅对象集合），并在 `for` 循环中判断如果 `Subscription`（订阅对象）的 `subscriber`（订阅者）属性等于传进来的 `subscriber`，则从 `Subscriptions` 中移除该 `Subscription`。EventBus 的源码就讲到这里了，下面讲解 `otto` 的相关知识。

7.2 解析 otto

otto 是 Square 公司发布的一个发布-订阅模式框架，它基于 Google Guava 项目中的 event bus 模块开发，针对 Android 平台做了优化和加强。虽然 Square 已经停止了对 otto 的更新并推荐使用 RxJava 和 RxAndroid 来替代它，但是 otto 的设计理念和源码仍旧值得我们学习。

7.2.1 使用 otto

(1) 添加依赖库

首先配置 gradle，如下所示：

```
compile 'com.squareup:otto:1.3.8'
```

(2) 定义消息类

与 EventBus 一样，我们接着定义消息类，它是一个 bean 文件，如下所示：

```
public class BusData {
    public String message;
    public BusData(String message){
        this.message=message;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

(3) 单例封装 Bus

otto 的 Bus 类相当于 EventBus 中的 EventBus 类，它封装了 otto 的主要功能。但它不是一个单例，其每次都要用 new 创建出来，这样显然不是很方便。因此我们用单例模式将它封装起来，如下所示：

```
public class OttoBus extends Bus{
    private volatile static OttoBus bus;
    private OttoBus (){}
}
public static OttoBus getInstance() {
    if (bus == null) {
        synchronized (OttoBus.class){
            if(bus==null){
```

```

        bus = new OttoBus();
    }
}
return bus;
}
}

```

(4) 注册和取消注册订阅事件

otto 同样需要注册和取消注册订阅事件，通过 OttoBus 得到 Bus 对象，调用 Bus 的 register 和 unregister 方法来注册和取消注册，同时我们定义一个 Button，点击这个 Button 跳转到 SecondActivity，SecondActivity 用来发送事件。具体代码如下所示：

```

public class MainActivity extends AppCompatActivity {
    private Button bt_jump;
    private TextView tv_message;
    private Bus bus;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tv_message= (TextView) this.findViewById(R.id.tv_message);
        bt_jump= (Button) this.findViewById(R.id.bt_jump);
        bt_jump.setText("跳转到 SecondActivity");
        bt_jump.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startActivity(new Intent(MainActivity.this,SecondActivity.class));
            }
        });
        bus=OttoBus.getInstance();
        bus.register(this);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        bus.unregister(this);
    }
}

```

(5) 事件订阅者处理事件

它和 EventBus 一样用 @Subscribe 来订阅事件，在 MainActivity 中添加如下代码：

```
@Subscribe
public void setContent(BusData data) {
    tv_message.setText(data.getMessage());
}
```

同样地用 `textView` 来显示接收到的消息。

(6) 使用 `post` 发送事件

创建 `SecondActivity`，并设置一个 `Button`，点击发送事件，并 `finish` 掉自身，如下所示：

```
public class SecondActivity extends AppCompatActivity {
    private Button bt_jump;
    private OttoBus bus;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bt_jump= (Button) this.findViewById(R.id.bt_jump);
        bt_jump.setText("发送事件");
        bt_jump.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                OttoBus.getInstance().post(new BusData("刘望舒的博客更新了"));
                finish();
            }
        });
    }
}
```

这样我们运行程序，点击 `MainActivity` 的“跳转到 `SECONDACTIVITY`”按钮直接跳转到 `SecondActivity`，再点击“发送事件”按钮，`SecondActivity` 被 `finish` 掉，回到 `MainActivity`，`MainActivity` 中 `textView` 的文字变为“刘望舒的博客更新了”。

(7) 使用 `@Produce` 来发布事件

`Produce` 注解用于生产发布事件。需要注意的是，它生产事件前需要进行注册，并且在生产完事件后需要取消注册。如果使用这种方法，则在跳转到发布者所在的类中时会立即产生事件并触发订阅者，修改 `SecondActivity`，代码如下所示：

```
public class SecondActivity extends AppCompatActivity {
    private Button bt_jump;
    private OttoBus bus;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```

super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
bt_jump= (Button) this.findViewById(R.id.bt_jump);
bt_jump.setText("发送事件");
bt_jump.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        finish();
    }
});
bus=OttoBus.getInstance();
bus.register(this);
}
@Produce
public BusData setInitialContent() {
    return new BusData("刘望舒的博客更新了");
}
@Override
protected void onDestroy() {
    super.onDestroy();
    bus.unregister(this);
}
}

```

在 MainActivity 跳转到 SecondActivity 时，MainActivity 会马上收到事件。

7.2.2 源码解析 otto

前面讲到了 otto 的用法，本节我们讲解 otto 的源码。首先来看一下 otto 源码各个类的作用，如图 7-4 所示。

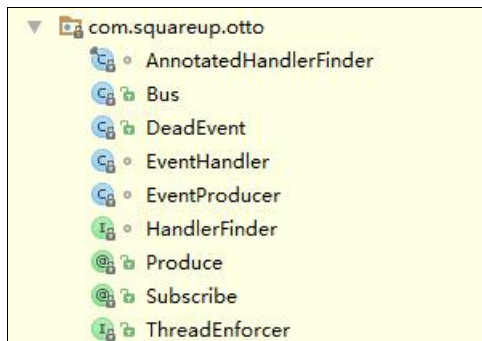


图 7-4 otto 源码中的类

如图 7-4 所示，otto 的源码并不多，其主要类的功能如下。

- **Produce、Subscribe:** 发布者和订阅者注解类。
- **Bus:** 事件总线类，用来注册和取消注册，维护发布-订阅模型，并处理事件调度分发。
- **HandlerFinder、AnnotatedHandlerFinder:** 用来查找发布者和订阅者。
- **EventProducer、EventHandler:** 分别封装发布者和订阅者的数据结构。

1. otto 构造方法

在使用 otto 时，首先要创建 Bus 类。Bus 类的构造方法如下所示：

```
public Bus() {  
    this(DEFAULT_IDENTIFIER);  
}
```

这个 DEFAULT_IDENTIFIER 是一个字符串"default",this 调用了 Bus 类的另一个构造方法：

```
public Bus(String identifier) {  
    this(ThreadEnforcer.MAIN, identifier);  
}
```

ThreadEnforcer.MAIN 意味着默认在主线程中调度事件。再往里看 this 又调用了什么，如下所示：

```
public Bus(ThreadEnforcer enforcer, String identifier) {  
    this(enforcer, identifier, HandlerFinder.ANNOTATED);  
}
```

第一个参数我们提到了，就是事件调度的简称，identifier 为 Bus 的名称，默认是"default"。而 identifier 则是 HandlerFinder，用于在 register、unregister 的时候寻找所有的 subscriber 和 producer。再往里查看 this 又调用了什么，如下所示：

```
Bus(ThreadEnforcer enforcer, String identifier, HandlerFinder handlerFinder) {  
    this.enforcer = enforcer;  
    this.identifier = identifier;  
    this.handlerFinder = handlerFinder;  
}
```

这是最终调用的 Bus 的构造方法，在这里要首先记住 handlerFinder 指的就是传进来的 HandlerFinder.ANNOTATED，后面在注册时会用到 handlerFinder 这个属性。

2. 注册

生成 Bus 类后，我们要调用它的 register 方法来进行注册。register 方法如下所示：

```
public void register(Object object) {  
    if (object == null) {  
        throw new NullPointerException("Object to register must not be null.");  
    }  
}
```

```

        enforcer.enforce(this);
        Map<Class<?>, EventProducer> foundProducers = handlerFinder.
            findAllProducers(object); //1
        ...
    }

```

上面代码注释 1 处调用了 `handlerFinder` 的 `findAllProducers` 方法。此前讲到构造方法时，我们知道该 `handlerFinder` 指的是 `HandlerFinder.ANNOTATED`。ANNOTATED 的代码如下所示：

```

HandlerFinder ANNOTATED = new HandlerFinder() {
    @Override
    public Map<Class<?>, EventProducer> findAllProducers(Object listener) {
        return AnnotatedHandlerFinder.findAllProducers(listener);
    }
    @Override
    public Map<Class<?>, Set<EventHandler>> findAllSubscribers(Object
        listener) {
        return AnnotatedHandlerFinder.findAllSubscribers(listener);
    }
};

```

从上面代码的 `findAllProducers` 方法和 `findAllSubscribers` 方法的返回值可以推断出一个注册类只能有一个发布者，但却可以有多个订阅者。`findAllProducers` 方法最终调用的是 `AnnotatedHandlerFinder` 的 `findAllProducers` 方法：

```

static Map<Class<?>, EventProducer> findAllProducers(Object listener) {
    final Class<?> listenerClass = listener.getClass();
    Map<Class<?>, EventProducer> handlersInMethod = new HashMap<Class<?>,
        EventProducer>();
    Map<Class<?>, Method> methods = PRODUCERS_CACHE.get(listenerClass); //1
    if (null == methods) {
        methods = new HashMap<Class<?>, Method>();
        loadAnnotatedProducerMethods(listenerClass, methods); //2
    }
    if (!methods.isEmpty()) {
        for (Map.Entry<Class<?>, Method> e : methods.entrySet()) { //3
            EventProducer producer = new EventProducer(listener, e.getValue());
            handlersInMethod.put(e.getKey(), producer);
        }
    }
    return handlersInMethod;
}

```

`PRODUCERS_CACHE` 是一个 `ConcurrentHashMap`，它的 key 为 `bus.register()` 时传入的 class，

而 value 是一个 map，这个 map 的 key 是事件的 class，value 是生产事件的方法。上面代码注释 1 处首先在 PRODUCERS_CACHE 根据传入的对象的类型查找是否有缓存的事件方法。如果没有，就调用注释 2 处的代码利用反射去寻找所有使用了 @Produce 注解的方法，并且将结果缓存到 PRODUCERS_CACHE 中。接着在注释 3 处遍历这些事件方法，并为每个事件方法创建 EventProducer 类，将这些 EventProducer 类作为 value 存入 handlersInMethod 并返回。接下来我们返回查看 register 方法，如下所示：

```
public void register(Object object) {
    if (object == null) {
        throw new NullPointerException("Object to register must not be null.");
    }
    enforcer.enforce(this);
    Map<Class<?>, EventProducer> foundProducers = handlerFinder.
        findAllProducers(object);
    for (Class<?> type : foundProducers.keySet()) {
        final EventProducer producer = foundProducers.get(type);
        EventProducer previousProducer = producersByType.putIfAbsent(type,
            producer); //1
        if (previousProducer != null) {
            throw new IllegalArgumentException("Producer method for type " + type
                + " found on type " + producer.target.getClass()
                + ", but already registered by type " + previousProducer.target.
                    getClass() + ".");
        }
        Set<EventHandler> handlers = handlersByType.get(type);
        if (handlers != null && !handlers.isEmpty()) {
            for (EventHandler handler : handlers) {
                dispatchProducerResultToHandler(handler, producer); //2
            }
        }
    }
    ...
}
```

调用完 findAllProducers 方法后，会在上面代码注释 1 处检查是否有该类型的发布者已经存在。如果存在，则抛出异常；如果不存在，则调用注释 2 处的 dispatchProducerResultToHandler 方法来触发和发布者对应的订阅者以处理事件。接下来 register 方法的后一部分代码就不贴上来。这跟此前的流程大致一样，就是调用 findAllSubscribers 方法来查找所有使用了 @Subscribe 注解的方法。跟此前不同的是，一个注册类可以有多个订阅者。随后判断是否有该类型的订阅者存在，也就是判断注册类是否已经注册，如果存在，则抛出异常；如果不存在，则查找是否有和这些订阅者对应的发布者，如果有的话，就会触发对应的订阅者处理事件。

3. 发送事件

我们会调用 Bus 的 `post` 方法来发送事件，它的代码如下所示：

```
public void post(Object event) {
    if (event == null) {
        throw new NullPointerException("Event to post must not be null.");
    }
    enforcer.enforce(this);
    Set<Class<?>> dispatchTypes = flattenHierarchy(event.getClass()); //1
    boolean dispatched = false;
    for (Class<?> eventType : dispatchTypes) {
        Set<EventHandler> wrappers = getHandlersForEventType(eventType);
        if (wrappers != null && !wrappers.isEmpty()) {
            dispatched = true;
            for (EventHandler wrapper : wrappers) {
                enqueueEvent(event, wrapper); //2
            }
        }
    }
    if (!dispatched && !(event instanceof DeadEvent)) {
        post(new DeadEvent(this, event));
    }
    dispatchQueuedEvents(); //3
}
```

上面代码注释 1 处的 `flattenHierarchy` 方法首先会从缓存中查找传进来的 `event` (消息事件类) 的所有父类，如果没有则找到 `event` 的所有父类并存储入缓存中。接下来遍历这些父类，找到它们的所有订阅者，并在注释 2 处将这些订阅者压入线程的事件队列中。在注释 3 处调用 `dispatchQueuedEvents` 方法，依次取出事件队列中的订阅者来处理相应 `event` 的事件。

4. 取消注册

取消注册时，我们会调用 Bus 的 `unregister` 方法。`unregister` 方法如下所示：

```
public void unregister(Object object) {
    if (object == null) {
        throw new NullPointerException("Object to unregister must not be null.");
    }
    enforcer.enforce(this);
    Map<Class<?>, EventProducer> producersInListener = handlerFinder.
        findAllProducers(object); //1
    for (Map.Entry<Class<?>, EventProducer> entry : producersInListener.
        entrySet()) {
```

```
final Class<?> key = entry.getKey();
EventProducer producer = getProducerForEventType(key);
EventProducer value = entry.getValue();
if (value == null || !value.equals(producer)) {
    throw new IllegalArgumentException(
        "Missing event producer for an annotated method. Is " + object.
        getClass()
        + " registered?");
}
producersByType.remove(key).invalidate();//2
}
...
}
```

取消注册分为两部分：一部分是订阅者取消注册，另一部分是发布者取消注册。这两部分的代码类似，因此，上面的代码只列出了发布者取消注册的代码。在上面代码注释 1 处得到所有使用 `@Produce` 注解的方法，并遍历这些方法；调用注释 2 处的代码从缓存中清除所有和传进来的注册类相关的发布者，以完成发布者的取消注册操作。

函数响应式编程

函数式编程是一种编程范式。我们常见的编程范式有命令式编程、函数式编程和逻辑式编程。我们常见的面向对象编程是一种命令式编程。命令式编程是面向计算机硬件的抽象，有变量、赋值语句、表达式和控制语句。而函数式编程是面向数学的抽象，将计算描述为一种表达式求值，函数可以在任何地方定义，并且可以对函数进行组合。

响应式编程是一种面向数据流和变化传播的编程范式，数据更新是相关联的。把函数式编程里的一套思路和响应式编程合起来就是函数响应式编程。

函数响应式编程可以极大地简化项目，特别是处理嵌套回调的异步事件、复杂的列表过滤和变换或者时间相关问题。在 Android 开发中使用函数响应式编程的主要有两大框架：一个是 RxJava，另一个是 Google 推出的 Agera。本章我们来学习 RxJava。

8.1 RxJava 基本用法

8.1.1 RxJava 概述

1. ReactiveX 与 RxJava

在讲到 RxJava 之前我们首先要了解什么是 ReactiveX，因为 RxJava 是 ReactiveX 的一种 Java 实现。ReactiveX 是 Reactive Extensions 的缩写，一般简称为 Rx。微软给的定义是，Rx 是一个函数库，让开发者可以利用可观察序列和 LINQ 风格查询操作符来编写异步和基于事件的程序。开发者可以用 Observables 表示异步数据流，用 LINQ 操作符查询异步数据流，用 Schedulers 参数化异步数据流的并发处理，Rx 可以这样定义：Rx = Observables + LINQ + Schedulers。

2. 为何要用 RxJava

说到异步操作，我们会想到 Android 的 AsyncTask 和 Handler。但是随着请求的数量越来越多，代码逻辑将会变得越来越复杂而 RxJava 却仍旧能保持清晰的逻辑。RxJava 的原理就是创建一个 Observable 对象来干活，然后使用各种操作符建立起来的链式操作，就如同流水线一样，把你想要处理的数据一步一步地加工成你想要的成品，然后发射给 Subscriber 处理。

3. RxJava 与观察者模式

RxJava 的异步操作是通过扩展的观察者模式来实现的，不了解观察者模式的读者可以先看一下 6.5.3 节。RxJava 有 4 个角色 Observable、Observer、Subscriber 和 Subject，这 4 个角色后面会具体讲解。Observable 和 Observer 通过 subscribe 方法实现订阅关系，Observable 就可以在需要的时候通知 Observer。

8.1.2 RxJava 基本实现

在使用 RxJava 前请先在 Android Studio 中配置 gradle:

```
dependencies {  
    ...  
    compile 'io.reactivex:rxjava:1.2.0'  
    compile 'io.reactivex:rxandroid:1.2.1'  
}
```

其中 RxAndroid 是 RxJava 在 Android 平台的扩展。它包含了一些能够简化 Android 开发的工具，比如特殊的调度器（后文会提到）。

RxJava 的基本用法分为如下 3 个步骤。

1. 创建 Observer（观察者）

它决定事件触发的时候将有怎样的行为，代码如下所示：

```
Subscriber subscriber=new Subscriber<String>() {  
    @Override  
    public void onCompleted() {  
        Log.d(TAG,"onCompleted");  
    }  
    @Override  
    public void onError(Throwable e) {  
        Log.d(TAG,"onError");  
    }  
    @Override  
    public void onNext(String s) {  
        Log.d(TAG,"onNext"+s);  
    }  
}
```

```

        @Override
        public void onStart() {
            Log.d(TAG, "onStart");
        }
    };

```

其中 `onCompleted`、`onError` 和 `onNext` 是必须要实现的方法，其含义如下。

- **onCompleted**: 事件队列完结。RxJava 不仅把每个事件单独处理，其还会把它们看作一个队列。当不会再有新的 `onNext` 发出时，需要触发 `onCompleted()` 方法作为完成标志。
- **onError**: 事件队列异常。在事件处理过程中出现异常时，`onError()` 会被触发，同时队列自动终止，不允许再有事件发出。
- **onNext**: 普通的事件。将要处理的事件添加到事件队列中。
- **onStart**: 它会在事件还未发送之前被调用，可以用于做一些准备工作。例如数据的清零或重置。这是一个可选方法，默认情况下它的实现为空。

当然，如果来实现简单的功能，也可以用到 `Observer` 来创建观察者。`Observer` 是一个接口，而上面用到的 `Subscriber` 是在 `Observer` 的基础上进行了扩展。在后文的 `Subscribe` 订阅过程中 `Observer` 也会先被转换为 `Subscriber` 来使用。用 `Observer` 创建观察者，如下所示：

```

Observer<String> observer = new Observer<String>() {
    @Override
    public void onCompleted() {
        Log.d(TAG, "onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        Log.d(TAG, "onError");
    }
    @Override
    public void onNext(String s) {
        Log.d(TAG, "onNext" + s);
    }
};

```

2. 创建 Observable（被观察者）

它决定什么时候触发事件以及触发怎样的事件。RxJava 使用 `create` 方法来创建一个 `Observable`，并为它定义事件触发规则，如下所示：

```

Observable observable = Observable.create(new Observable.OnSubscribe
<String>() {
    @Override
    public void call(Subscriber<? super String> subscriber) {

```



```

        subscriber.onNext("杨影枫");
        subscriber.onNext("月眉儿");
        subscriber.onCompleted();
    }
});

```

通过调用 `Subscriber` 的方法，不断地将事件添加到任务队列中。也可用 `just` 方法来实现：

```
Observable observable = Observable.just("杨影枫", "月眉儿");
```

上述代码会依次调用 `onNext("杨影枫")`、`onNext("月眉儿")`、`onCompleted()`。还可以用 `from` 方法来实现，如下所示：

```
String[] words = {"杨影枫", "月眉儿"};
Observable observable = Observable.from(words);
```

上述代码调用的方法顺序和 `just` 方法是一样的。

3. Subscribe（订阅）

订阅只需要一行代码就可以了，如下所示：

```
observable.subscribe(subscriber);
```

运行代码查看 log：

```

D/RxJava: onStart
D/RxJava: onNext 杨影枫
D/RxJava: onNext 月眉儿
D/RxJava: onCompleted

```

和预想的一样先调用 `onStart` 方法，接着调用两个 `onNext` 方法，最后调用 `onCompleted` 方法。

8.1.3 RxJava 的不完整定义回调

前面介绍了回调的接收主要依赖于 `subscribe(Observer)` 和 `subscribe(Subscriber)`。此外，`RxJava` 还提供了另一种回调方式，也就是不完整回调。在讲到不完整回调之前我们首先要了解 `Action`。查看 `RxJava` 源码，我们发现其提供了一堆 `Action`，如图 8-1 所示。

我们打开 `Action0` 来看看：

```

public interface Action0 extends Action {
    void call();
}

```

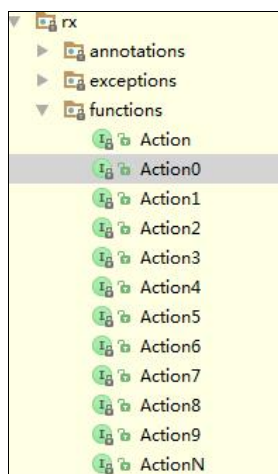


图 8-1 Action

再打开 Action1:

```
public interface Action1<T> extends Action {
    void call(T t);
}
```

最后看看 Action9:

```
public interface Action9<T1, T2, T3, T4, T5, T6, T7, T8, T9> extends Action {
    void call(T1 t1, T2 t2, T3 t3, T4 t4, T5 t5, T6 t6, T7 t7, T8 t8, T9 t9);
}
```

很明显, Action 后的数字代表回调的参数类型数量, 8.1.2 节中的创建 Observer 和订阅代码也就可以改写为下面的代码:

```
Action1<String> onNextAction = new Action1<String>() {
    @Override
    public void call(String s) {
        Log.d(TAG, "onNext" + s);
    }
};

Action1<Throwable> onErrorAction = new Action1<Throwable>() {
    @Override
    public void call(Throwable throwable) {

    }
};

Action0 onCompletedAction = new Action0() {
    @Override
    public void call() {
```

```

        Log.d(TAG, "onCompleted");
    }
};
observable.subscribe(onNextAction, onErrorAction, onCompletedAction);

```

我们定义了 `onNextAction` 来处理 `onNext` 的回调，同理，我们还定义了 `onErrorAction` 和 `onCompletedAction` 来分别处理 `onError` 和 `onCompleted` 的回调，最后我们把它们传给 `subscribe` 方法。很显然这样写的灵活度更大一些；同时我们也可以只传一个或者两个 `Action`，如下所示：

```

observable.subscribe(onNextAction);
observable.subscribe(onNextAction, onErrorAction);

```

第一行只定义了 `onNextAction` 来处理 `onNext` 的回调；而第二行则定义了 `onNextAction` 处理 `onNext` 的回调，`onErrorAction` 来处理 `onError` 的回调。

8.2 RxJava 的 Subject

`Subject` 既可以是一个 `Observer` 也可以是一个 `Observable`，它是连接 `Observer` 和 `Observable` 的桥梁。因此，`Subject` 可以被理解为 `Subject = Observable + Observer`。RxJava 提供了以下 4 种 `Subject`。

1. PublishSubject

`PublishSubject` 只会把在订阅发生的时间点之后来自原始 `Observable` 的数据发射给观察者。需要注意的是，`PublishSubject` 可能会一创建完成就立刻开始发射数据，因此这里会有一个风险：在 `Subject` 被创建后到有观察者订阅它之前这个时间段内，一个或多个数据可能会丢失。如果要确保来自原始 `Observable` 的所有数据都被分发，则可以当所有观察者都已经订阅时才开始发射数据，或者改用 `ReplaySubject`。

2. BehaviorSubject

当 `Observer` 订阅 `BehaviorSubject` 时，它开始发射原始 `Observable` 最近发射的数据。如果此时还没有收到任何数据，它会发射一个默认值，然后继续发射其他任何来自原始 `Observable` 的数据。如果原始的 `Observable` 因为发生了一个错误而终止，`BehaviorSubject` 将不会发射任何数据，但是会向 `Observer` 传递一个异常通知。

3. ReplaySubject

不管 `Observer` 何时订阅 `ReplaySubject`，`ReplaySubject` 均会发射所有来自原始 `Observable` 的数据给 `Observer`。有不同类型的 `ReplaySubject`，它们用于限定 `Replay` 的范围，例如设定 `Buffer` 的具体大小，或者设定具体的时间范围。如果使用 `ReplaySubject` 作为 `Observer`，注意不要在多个线程中调用 `onNext`、`onComplete` 和 `onError` 方法。这可能会导致顺序错乱，并且违反了 `Observer` 规则。

4. AsyncSubject

当 Observable 完成时, AsyncSubject 只会发射来自原始 Observable 的最后一个数据。如果原始的 Observable 因为发生了错误而终止, AsyncSubject 将不会发射任何数据,但是会向 Observer 传递一个异常通知。

8.3 RxJava 操作符入门

RxJava 操作符的类型分为创建操作符、变换操作符、过滤操作符、组合操作符、错误处理操作符、辅助操作符、条件和布尔操作符、算术和聚合操作符及连接操作符等,而这些操作符类型下又有很多操作符,每个操作符可能还有很多变体。因为篇幅有限,这里只介绍相对常用的操作符,以及这些操作符的常规用法,以便帮助大家入门。如果读者想要更详细地学习操作符,请查阅官方文档: <http://reactivex.io/RxJava/javadoc/>。

8.3.1 创建操作符

在 8.1.2 节中我们已经用到了创建操作符 create、just 和 from,这里就不赘述它们的用法了。除了它们,还有 defer、range、interval、start、repeat 和 timer 等创建操作符。这里会讲解 interval、range 和 repeat。

1. interval

创建一个按固定时间间隔发射整数序列的 Observable,相当于定时器,如下所示:

```
Observable.interval(3, TimeUnit.SECONDS)
    .subscribe(new Action1<Long>() {
        @Override
        public void call(Long mlong) {
            Log.d(TAG, "interval:" + mlong.intValue());
        }
    });
```

上面的代码每隔 3s 就会调用 call 方法并打印 Log。

2. range

创建发射指定范围的整数序列的 Observable,可以拿来替代 for 循环,发射一个范围内的有序整数序列。第一个参数是起始值,并且不小于 0;第二个参数为终值,左闭右开。

```
Observable.range(0, 5)
    .subscribe(new Action1<Integer>() {
        @Override
        public void call(Integer integer) {
            Log.d(TAG, "range:" + integer.intValue());
        }
    });
```

```
    }
  });
```

输出结果为:

D/RxJava: range:0

D/RxJava: range:1

D/RxJava: range:2

D/RxJava: range:3

D/RxJava: range:4

3. repeat

创建一个 N 次重复发射特定数据的 Observable, 如下所示:

```
Observable.range(0, 3)
    .repeat(2)
    .subscribe(new Action1<Integer>() {
        @Override
        public void call(Integer integer) {
            Log.d(TAG, "repeat:" +integer.intValue());
        }
    });
```

输出结果为:

D/RxJava: repeat:0

D/RxJava: repeat:1

D/RxJava: repeat:2

D/RxJava: repeat:0

D/RxJava: repeat:1

D/RxJava: repeat:2

8.3.2 变换操作符

变换操作符的作用是对 Observable 发射的数据按照一定规则做一些变换操作, 然后将变换后的数据发射出去。变换操作符有 map、flatMap、concatMap、switchMap、flatMapIterable、buffer、groupBy、cast、window、scan 等, 这里会讲解 map、flatMap、cast、concatMap、flatMapIterable、buffer 和 groupBy。

1. map

map 操作符通过指定一个 Func 对象, 将 Observable 转换为一个新的 Observable 对象并发射, 观察者将收到新的 Observable 处理。假设我们要访问网络, Host 地址时常是变化的, 它有时是

测试服务器地址，有时可能是正式服务器地址，但是具体界面的 URL 地址则是不变的。因此，我们可以用 `map` 操作符来进行转换字符操作，如下所示：

```
final String Host="http://blog.csdn.net/";
Observable.just("itachi85").map(new Func1<String, String>() {
    @Override
    public String call(String s) {
        return Host+s;
    }
}).subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        Log.d(TAG, "map:"+s);
    }
});
```

输出结果为：

D/RxJava: map:http://blog.csdn.net/itachi85

2. flatMap、cast

`flatMap` 操作符将 `Observable` 发射的数据集合变换为 `Observable` 集合，然后将这些 `Observable` 发射的数据平坦化地放进一个单独的 `Observable`。`cast` 操作符的作用是强制将 `Observable` 发射的所有数据转换为指定类型。假设我们仍旧访问网络，但是要访问同一个 Host 的多个界面，我们可以使用 `for` 循环在每个界面的 URL 前添加 Host。但是 `RxJava` 提供了一个更方便的操作，如下所示：

```
final String Host="http://blog.csdn.net/";
List<String> mlist = new ArrayList<>();
mlist.add("itachi85");
mlist.add("itachi86");
mlist.add("itachi87");
mlist.add("itachi88");
Observable.from(mlist).flatMap(new Func1<String, Observable<?>>() {
    @Override
    public Observable<?> call(String s) {
        return Observable.just(Host+s);
    }
}).cast(String.class).subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        Log.d(TAG, "flatMap:"+s);
    }
});
```

首先用 `ArrayList` 存储要访问的界面 URL，然后通过 `flatMap` 转换成 `Observable`。`cast` 操作符将 `Observable` 中的数据转换为 `String` 类型。输出结果为：

```
D/RxJava: flatMap:http://blog.csdn.net/itachi85
D/RxJava: flatMap:http://blog.csdn.net/itachi86
D/RxJava: flatMap:http://blog.csdn.net/itachi87
D/RxJava: flatMap:http://blog.csdn.net/itachi88
```

另外，`flatMap` 的合并允许交叉，也就是说可能会交错地发送事件，最终结果的顺序可能并不是原始 `Observable` 发送时的顺序。

3. concatMap

`concatMap` 操作符功能与 `flatMap` 操作符一致；不过，它解决了 `flatMap` 交叉问题，提供了一种能够把发射的值连续在一起的函数，而不是合并它们。`concatMap` 的使用方法和 `flatMap` 类似，这里就不重复贴代码了。

4. flatMapIterable

`flatMapIterable` 操作符可以将数据包装成 `Iterable`，在 `Iterable` 中我们就可以对数据进行处理了，如下所示：

```
Observable.just(1,2,3).flatMapIterable(new Func1<Integer, Iterable<Integer>>() {
    @Override
    public Iterable<Integer> call(Integer s) {
        List<Integer> mlist=new ArrayList<Integer>();
        mlist.add(s+1);//1
        return mlist;
    }
}).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "flatMapIterable:"+integer);
    }
});
```

在上面代码注释 1 处对每个数都加 1，因此输出结果为：

```
D/RxJava: flatMapIterable:2
D/RxJava: flatMapIterable:3
D/RxJava: flatMapIterable:4
```

5. buffer

`buffer` 操作符将源 `Observable` 变换为一个新的 `Observable`，这个新的 `Observable` 每次发射一

组列表值而不是一个一个发射。和 `buffer` 操作符类似的还有 `window` 操作符，只不过 `window` 操作符发射的是 `Observable` 而不是数据列表，如下所示：

```
Observable.just(1,2,3,4,5,6)
    .buffer(3)
    .subscribe(new Action1<List<Integer>>() {
        @Override
        public void call(List<Integer> integers) {
            for (Integer i : integers) {
                Log.d(TAG, "buffer:"+i);
            }
            Log.d(TAG, "-----");
        }
    });
```

`buffer(3)`的意思是缓存容量为 3，输出结果为：

```
D/RxJava: buffer:1
D/RxJava: buffer:2
D/RxJava: buffer:3
D/RxJava: -----
D/RxJava: buffer:4
D/RxJava: buffer:5
D/RxJava: buffer:6
D/RxJava: -----
```

6. groupBy

`groupBy` 操作符用于分组元素，将源 `Observable` 变换成一个发射 `Observables` 的新 `Observable`（分组后的）。它们中的每一个新 `Observable` 都发射一组指定的数据，如下所示：

```
Swordsman s1 = new Swordsman("韦一笑", "A");
Swordsman s2 = new Swordsman("张三丰", "SS");
Swordsman s3 = new Swordsman("周芷若", "S");
Swordsman s4 = new Swordsman("宋远桥", "S");
Swordsman s5 = new Swordsman("殷梨亭", "A");
Swordsman s6 = new Swordsman("张无忌", "SS");
Swordsman s7 = new Swordsman("鹤笔翁", "S");
Swordsman s8 = new Swordsman("宋青书", "A");
Observable<GroupedObservable<String, Swordsman>> GroupedObservable
= Observable.just(s1, s2, s3, s4, s5, s6, s7, s8)
    .groupBy(new Func1<Swordsman, String>() {
        @Override
        public String call(Swordsman swordsman) {
```



```

        return swordsman.getLevel();
    }
});
Observable.concat(GroupedObservable).subscribe(new Action1<Swordsman>() {
    @Override
    public void call(Swordsman swordsman) {
        Log.d(TAG, "groupBy:" + swordsman.getName() + "---" +
            swordsman.getLevel());
    }
});

```

这里创建了《倚天屠龙记》里的 8 个武侠，对其按照实力等级进行划分，从高到低依次是 SS、S、A。使用 `groupBy` 可以帮助我们对某一个 key 值进行分组，将相同的 key 值数据排在一起。在这里我们的 key 值就是武侠的 Level（实力等级）。其中 `concat` 是组合操作符，后面会介绍它。

输出结果为：

```

D/RxJava: groupBy:韦一笑---A
D/RxJava: groupBy:殷梨亭---A
D/RxJava: groupBy:宋青书---A
D/RxJava: groupBy:张三丰---SS
D/RxJava: groupBy:张无忌---SS
D/RxJava: groupBy:周芷若---S
D/RxJava: groupBy:宋远桥---S
D/RxJava: groupBy:鹤笔翁---S

```

8.3.3 过滤操作符

过滤操作符用于过滤和选择 `Observable` 发射的数据序列，让 `Observable` 只返回满足我们条件的数据。过滤操作符有 `filter`、`elementAt`、`distinct`、`skip`、`take`、`skipLast`、`takeLast`、`ignoreElements`、`throttleFirst`、`sample`、`debounce` 和 `throttleWithTimeout` 等，这里会介绍 `filter`、`elementAt`、`distinct`、`skip`、`take`、`ignoreElements`、`throttleFirst` 和 `throttleWithTimeOut`。

1. filter

`filter` 操作符是对源 `Observable` 产生的结果自定义规则进行过滤，只有满足条件的结果才会提交给订阅者，如下所示。

```

Observable.just(1,2,3,4).filter(new Func1<Integer, Boolean>() {
    @Override
    public Boolean call(Integer integer) {
        return integer>2;//1
    }
});

```

```

    }
    }).subscribe(new ActionListener<Integer>() {
        @Override
        public void call(Integer integer) {
            Log.d(TAG, "filter:" + integer);
        }
    });

```

上面代码注释 1 处设定大于 2 的数字会被返回并被提交给订阅者。

输出结果为：

D/RxJava: filter:3

D/RxJava: filter:4

2. elementAt

`elementAt` 操作符用来返回指定位置的数据。和它类似的有 `elementAtOrDefault(int,T)`，其可以允许默认值。具体代码如下所示：

```

Observable.just(1,2,3,4).elementAt(2).subscribe(new ActionListener<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "elementAt:" + integer);
    }
});

```

输出结果为：

D/RxJava: elementAt:3

3. distinct

`distinct` 操作符用来去重，其只允许还没有发射过的数据项通过。和它类似的还有 `distinctUntilChanged` 操作符，它用来去掉连续重复的数据。

```

Observable.just(1,2,2,3,4,1).distinct().subscribe(new ActionListener<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "distinct:" + integer);
    }
});

```

输出结果为：

D/RxJava: distinct:1

D/RxJava: distinct:2

D/RxJava: distinct:3

D/RxJava: distinct:4

4. skip、take

skip 操作符将源 Observable 发射的数据过滤掉前 n 项；而 take 操作符则只取前 n 项；另外还有 skipLast 和 takeLast 操作符，则是从后面进行过滤操作。先来看 skip 操作符，如下所示：

```
Observable.just(1,2,3,4,5,6).skip(2).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG "skip:" + integer);
    }
});
```

输出结果为：

D/RxJava: skip:3

D/RxJava: skip:4

D/RxJava: skip:5

D/RxJava: skip:6

接下来查看 take 操作符，如下所示：

```
Observable.just(1,2,3,4,5,6).take(2).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "take:" + integer);
    }
});
```

输出结果为：

D/RxJava: take:1

D/RxJava: take:2

5. ignoreElements

ignoreElements 操作符忽略所有源 Observable 产生的结果，只把 Observable 的 onCompleted 和 onError 事件通知给订阅者。具体如下所示：

```
Observable.just(1, 2, 3, 4).ignoreElements().subscribe(new Observer<Integer>() {
    @Override
    public void onCompleted() {
        Log.d(TAG, "onCompleted");
    }
});
```

```

        @Override
        public void onError(Throwable e) {
            Log.d(TAG, "onError");
        }
        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "onNext");
        }
    }
});

```

输出结果为:

D/RxJava: onCompleted

6. throttleFirst

`throttleFirst` 操作符则会定期发射这个时间段里源 `Observable` 发射的第一个数据, `throttleFirst` 操作符默认在 `computation` 调度器上执行(关于调度器后面会讲到)。和 `throttleFirst` 操作符类似的有 `sample` 操作符, 它会定时地发射源 `Observable` 最近发射的数据, 其他的都会被过滤掉。`throttleFirst` 操作符的使用示例如下所示:

```

Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for(int i=0;i<10;i++){
            subscriber.onNext(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        subscriber.onCompleted();
    }
}).throttleFirst(200, TimeUnit.MILLISECONDS).subscribe(new
    Action1<Integer>() {
        @Override
        public void call(Integer integer) {
            Log.d(TAG, "throttleFirst:"+integer);
        }
    }
});

```

每隔 100ms 发射一个数据。`throttleFirst` 操作符设定的时间为 200ms, 因此, 它会发射每个 200ms 内的第一个数据, 输出结果为:

```
D/RxJava: throttleFirst:0
D/RxJava: throttleFirst:2
D/RxJava: throttleFirst:4
D/RxJava: throttleFirst:6
D/RxJava: throttleFirst:8
```

7. throttleWithTimeOut

通过时间来限流。源 Observable 每次发射出来一个数据后就会进行计时。如果在设定好的时间结束前源 Observable 有新的数据发射出来，这个数据就会被丢弃，同时 throttleWithTimeOut 重新开始计时。如果每次都是在计时结束前发射数据，那么这个限流就会走向极端：只会发射最后一个数据。其默认在 computation 调度器上执行。和 throttleWithTimeOut 操作符类似的有 deounce 操作符，它不仅可以使用时间来进行过滤，还可以根据一个函数来进行限流。throttleWithTimeOut 操作符的使用示例如下所示：

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for (int i = 0; i < 10; i++) {
            subscriber.onNext(i);
            int sleep = 100;
            if (i % 3 == 0) {
                sleep = 300;
            }
            try {
                Thread.sleep(sleep);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        subscriber.onCompleted();
    }
}).throttleWithTimeout(200, TimeUnit.MILLISECONDS).subscribe(new
    Action1<Integer>() {
        @Override
        public void call(Integer integer) {
            Log.d("wangshu", "throttleWithTimeOut:" + integer);
        }
    });
```

每隔 100ms 发射一个数据。当发射的数据是 3 的倍数的时候，下一个数据就延迟到 300ms 再发射。这里设定的过滤时间是 200ms，也就是说发射间隔小于 200ms 的数据会被过滤掉。

打印结果如下所示：

```
D/RxJava: throttleWithTimeOut:0
D/RxJava: throttleWithTimeOut:3
D/RxJava: throttleWithTimeOut:6
D/RxJava: throttleWithTimeOut:9
```

8.3.4 组合操作符

组合操作符可以同时处理多个 Observable 来创建我们所需要的 Observable。组合操作符有 startWith、merge、concat、zip、combineLatest、join 和 switch 等，这里介绍 startWith、merge、concat、zip 和 combineLatest。

1. startWith

startWith 操作符会在源 Observable 发射的数据前面插上一些数据，如下所示：

```
Observable.just(3, 4, 5).startWith(1, 2)
    .subscribe(new Action1<Integer>() {
        @Override
        public void call(Integer integer) {
            Log.i(TAG, "startWith:"+integer);
        }
    });
```

输出结果为：

```
D/RxJava: startWith:1
D/RxJava: startWith:2
D/RxJava: startWith:3
D/RxJava: startWith:4
D/RxJava: startWith:5
```

2. merge

merge 操作符将多个 Observable 合并到一个 Observable 中进行发射，merge 可能会让合并的 Observable 发射的数据交错。

```
Observable<Integer> obs1 =Observable.just(1,2,3).subscribeOn(Schedulers.io());
Observable<Integer> obs2 =Observable.just(4,5,6);
Observable.merge(obs1,obs2).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "merge:"+integer);
    }
});
```

输出结果为:

```
D/RxJava: merge:4
D/RxJava: merge:5
D/RxJava: merge:6
D/RxJava: merge:1
D/RxJava: merge:2
D/RxJava: merge:3
```

3. concat

将多个 Observable 发射的数据进行合并发射。`concat` 严格按照顺序发射数据, 前一个 Observable 没发射完成是不会发射后一个 Observable 的数据的。

```
Observable<Integer> obs1 =Observable.just(1,2,3).subscribeOn(Schedulers.io());
Observable<Integer> obs2 =Observable.just(4,5,6);
Observable.concat(obs1,obs2).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "concat:"+integer);
    }
});
```

输出结果为:

```
D/RxJava: concat:1
D/RxJava: concat:2
D/RxJava: concat:3
D/RxJava: concat:4
D/RxJava: concat:5
D/RxJava: concat:6
```

4. zip

`zip` 操作符合并两个或者多个 Observable 发射出的数据项, 根据指定的函数变换它们, 并发射一个新值。

```
Observable<Integer> obs1 =Observable.just(1,2,3);
Observable<String> obs2 =Observable.just("a","b","c");
Observable.zip(obs1, obs2, new Func2<Integer, String, String>() {
    @Override
    public String call(Integer integer, String s) {
        return integer+s;
    }
});
```

```

    }
    }).subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
            Log.d(TAG, "zip:"+s);
        }
    });

```

输出结果为:

I/RxJava: zip:1a

I/RxJava: zip:2b

I/RxJava: zip:3c

5. combineLatest

当两个 Observable 中的任何一个发射了数据时, 使用一个函数结合每个 Observable 发射的最近数据项, 并且基于这个函数的结果发射数据。combineLatest 操作符和 zip 有些类似。只不过 zip 操作符作用于最近未打包的两个 Observable, 只有当原始的 Observable 中的每一个都发射了一条数据时 zip 才发射数据; 而 combineLatest 操作符作用于最近发射的数据项, 在原始 Observable 中的任意一个发射了数据时发射一条数据。这讲起来有些抽象, 下面来看具体例子:

```

Observable<Integer> obs1 =Observable.just(1,2,3);
Observable<String> obs2 =Observable.just("a","b","c");
Observable.combineLatest(obs1, obs2, new Func2<Integer, String,
String>() {
    @Override
    public String call(Integer integer, String s) {
        return integer+s;
    }
}).subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        Log.d(TAG, "combineLatest:"+s);
    }
});

```

如果其中的一个 Observable 还有数据没有发射, 那么 combineLatest 操作符会将两个 Observable 最新发射的数据组合在一起。比如上面的例子, 第一个 Observable 最新的数据是 6, 然后第二个 Observable 的数据依次在变, 然后再把它们组合在一起。

输出结果为:

D/RxJava: combineLatest:3a

D/RxJava: combineLatest:3b

D/RxJava: combineLastest:3c

8.3.5 辅助操作符

辅助操作符可以帮助我们更加方便地处理 Observable。辅助操作符包括 delay、DO、subscribeOn、observeOn、timeout、materialize、dematerialize、timeInterval、timestamp 和 to 等。在这里介绍 delay、Do、subscribeOn、observeOn 和 timeout。

1. delay

delay 操作符让原始 Observable 在发射每项数据之前都暂停一段指定的时间段。

```
Observable.create(new Observable.OnSubscribe<Long>() {
    @Override
    public void call(Subscriber<? super Long> subscriber) {
        Long currentTime=System.currentTimeMillis()/1000;
        subscriber.onNext(currentTime);
    }
}).delay(2, TimeUnit.SECONDS).subscribe(new Action1<Long>() {
    @Override
    public void call(Long aLong) {
        Log.d(TAG, "delay:"+(System.currentTimeMillis()/1000-aLong));
    }
});
```

输出结果为:

D/RxJava: delay:2

2. Do

Do 系列操作符就是为原始 Observable 的生命周期事件注册一个回调,当 Observable 的某个事件发生时就会调用这些回调。RxJava 中有很多 Do 系列操作符,如下所示。

- doOnEach: 为 Observable 注册这样一个回调,当 Observable 每发射一项数据时就会调用它一次,包括 onNext、onError 和 onCompleted。
- doOnNext: 只有执行 onNext 的时候会被调用。
- doOnSubscribe: 当观察者订阅 Observable 时就会被调用。
- doOnUnsubscribe: 当观察者取消订阅 Observable 时就会被调用;Observable 通过 onError 或者 onCompleted 结束时,会取消订阅所有的 Subscriber。
- doOnCompleted: 当 Observable 正常终止调用 onCompleted 时会被调用。
- doOnError: 当 Observable 异常终止调用 onError 时会被调用。
- doOnTerminate: 当 Observable 终止(无论是正常终止还是异常终止)之前会被调用。
- finallyDo: 当 Observable 终止(无论是正常终止还是异常终止)之后会被调用。

这里拿 `doOnNext` 来举例，代码如下所示：

```
Observable.just(1,2)
    .doOnNext(new Action1<Integer>() {
        @Override
        public void call(Integer item) {
            Log.d(TAG,"call:" + item);
        }
    }).subscribe(new Subscriber<Integer>() {
    @Override
    public void onNext(Integer item) {
        Log.d(TAG,"onNext:" + item);
    }
    @Override
    public void onError(Throwable error) {
        Log.d(TAG,"Error:" + error.getMessage());
    }
    @Override
    public void onCompleted() {
        Log.d(TAG,"onCompleted");
    }
    });
```

输出结果为：

D/RxJava: call:1

D/RxJava: onNext:1

D/RxJava: call:2

D/RxJava: onNext:2

D/RxJava: onCompleted

3. `subscribeOn`、`observeOn`

`subscribeOn` 操作符用于指定 `Observable` 自身在哪个线程上运行。如果 `Observable` 需要执行耗时操作，一般可以让其在新开的一个子线程上运行。`observeOn` 用来指定 `Observer` 所运行的线程，也就是发射出的数据在哪个线程上使用。一般情况下会指定在主线程中运行，这样就可以修改 UI。具体示例如下所示：

```
Observable<Integer> obs= Observable.create(new Observable.OnSubscribe
<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        Log.d(TAG,"Observable:" + Thread.currentThread().getName());
        subscriber.onNext(1);
    }
});
```

```

        subscriber.onCompleted();
    }
});
obs.subscribeOn(Schedulers.newThread()).observeOn
(AndroidSchedulers.mainThread()).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "Observer:" + Thread.currentThread().getName());
    }
});

```

`subscribeOn(Schedulers.newThread())` 表示 `Observable` 运行在新开的线程，`observeOn(AndroidSchedulers.mainThread())` 表示运行在主线程。其中，`AndroidSchedulers` 是 `RxAndroid` 库提供的 `Scheduler`，`Scheduler` 的知识后面会讲到。

输出结果为：

D/RxJava: Observable:RxNewThreadScheduler-2

D/RxJava: Observer:main

4. timeout

如果原始 `Observable` 过了指定的一段时长没有发射任何数据，`timeout` 操作符会以一个 `onError` 通知终止这个 `Observable`，或者继续执行一个备用的 `Observable`。`timeout` 有很多变体，这里介绍其中的一种：`timeout(long, TimeUnit, Observable)`，它在超时会切换到使用一个你指定的备用的 `Observable`，而不是发送错误通知。它默认在 `computation` 调度器上执行。具体示例如下所示：

```

Observable<Integer> obs = Observable.create(new Observable.OnSubscribe
<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for(int i=0; i<4; i++){
            try {
                Thread.sleep(i * 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            subscriber.onNext(i);
        }
        subscriber.onCompleted();
    }
}).timeout(200, TimeUnit.MILLISECONDS, Observable.just(10, 11));
obs.subscribe(new Action1<Integer>() {

```

```

@Override
public void call(Integer integer) {
    Log.d(TAG, "timeout:" + integer);
}
});

```

如果 Observable 在 200ms 这段时长没有发射数据，就会切换到 Observable.just(10,11)，输出结果为：

```

D/RxJava: timeout:0
D/RxJava: timeout:1
D/RxJava: timeout:2
D/RxJava: timeout:10
D/RxJava: timeout:11

```

8.3.6 错误处理操作符

RxJava 在错误出现的时候就会调用 Subscriber 的 onError 方法将错误分发出去，由 Subscriber 自己来处理错误。但是如果每个 Subscriber 都处理一遍的话，工作量就有点大了，这时候可以使用错误处理操作符。错误处理操作符有 catch 和 retry。

1. catch

catch 操作符拦截原始 Observable 的 onError 通知，将它替换为其他数据项或数据序列，让产生的 Observable 能够正常终止或者根本不终止。RxJava 将 catch 实现为以下 3 个不同的操作符。

- **onErrorReturn**: Observable 遇到错误时返回原有 Observable 行为的备用 Observable，备用 Observable 会忽略原有 Observable 的 onError 调用，不会将错误传递给观察者。作为替代，它会发射一个特殊的项并调用观察者的 onCompleted 方法。
- **onErrorResumeNext**: Observable 遇到错误时返回原有 Observable 行为的备用 Observable，备用 Observable 会忽略原有 Observable 的 onError 调用，不会将错误传递给观察者。作为替代，它会发射备用 Observable 的数据。
- **onExceptionResumeNext**: 它和 onErrorResumeNext 类似。不同的是，如果 onError 收到的 Throwable 不是一个 Exception，它会将错误传递给观察者的 onError 方法，不会使用备用的 Observable。

这里拿 onErrorReturn 操作符来举例，如下所示。

```

Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {

```

```

        for (int i = 0; i < 5; i++) {
            if (i > 2) {
                subscriber.onError(new Throwable("Throwable"));
            }
            subscriber.onNext(i);
        }
        subscriber.onCompleted();
    }
}).onErrorReturn(new Func1<Throwable, Integer>() {
    @Override
    public Integer call(Throwable throwable) {
        return 6;
    }
}).subscribe(new Observer<Integer>() {
    @Override
    public void onCompleted() {
        Log.d(TAG, "onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        Log.d(TAG, "onError:" + e.getMessage());
    }
    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "onNext:" + integer);
    }
});

```

输出结果为:

```

D/RxJava: onNext:0
D/RxJava: onNext:1
D/RxJava: onNext:2
D/RxJava: onNext:6
D/RxJava: onCompleted

```

2. retry

`retry` 操作符不会将原始 `Observable` 的 `onError` 通知传递给观察者, 它会订阅这个 `Observable`, 再给它一次机会无错误地完成其数据序列。`retry` 总是传递 `onNext` 通知给观察者, 由于重新订阅, 这可能会造成数据项重复。RxJava 中的实现为 `retry` 和 `retryWhen`。这里拿 `retry(long)` 来举例, 它指定最多重新订阅的次数。如果次数超了, 它不会尝试再次订阅, 而会把最新的一个 `onError` 通知传递给自己的观察者, 如下所示。

```

Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for (int i = 0; i < 5; i++) {
            if (i==1) {
                subscriber.onError(new Throwable("Throwable"));
            } else {
                subscriber.onNext(i); //1
            }
        }
        subscriber.onCompleted();
    }
}).retry(2).subscribe(new Observer<Integer>() {
    @Override
    public void onCompleted() {
        Log.d(TAG, "onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        Log.d(TAG, "onError:" + e.getMessage());
    }
    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "onNext:" + integer);
    }
});

```

上面的重新订阅次数为 2，在 $i=0$ 的时候会调用上面代码注释 1 处的 `onNext` 方法。此外重试的这两次同样会调用 `onNext` 方法，这样一共会调用 3 次 `onNext` 方法，最后才会调用 `onError` 方法，输出结果为：

```

D/RxJava: onNext:0
D/RxJava: onNext:0
D/RxJava: onNext:0
D/RxJava: onError:Throwable

```

8.3.7 条件操作符和布尔操作符

条件操作符和布尔操作符可用于根据条件发射或变换 `Observable`，或者对它们做布尔运算。现在先来了解条件操作符。

1. 布尔操作符

布尔操作符有 `all`、`contains`、`isEmpty`、`exists` 和 `sequenceEqual`，这里介绍前 3 个操作符。

(1) all

all 操作符根据一个函数对源 **Observable** 发射的所有数据进行判断，最终返回的结果就是这个判断结果。这个函数使用发射的数据作为参数，内部判断所有的数据是否满足我们定义好的判断条件。如果全部都满足则返回 **true**，否则就返回 **false**。

```
Observable.just(1,2,3,4)
    .all(new Func1<Integer, Boolean>() {
        @Override
        public Boolean call(Integer integer) {
            Log.d(TAG, "call:"+integer);
            return integer<3;
        }
    }).subscribe(new Subscriber<Boolean>() {
        @Override
        public void onCompleted() {
            Log.d(TAG, "onCompleted");
        }
        @Override
        public void onError(Throwable e) {
            Log.d(TAG, "onError:"+e.getMessage());
        }
        @Override
        public void onNext(Boolean aBoolean) {
            Log.d(TAG, "onNext:"+aBoolean);
        }
    });
```

输出结果为:

D/RxJava: call:1

D/RxJava: call:2

D/RxJava: onNext:false

D/RxJava: onCompleted

(2) contains、isEmpty

contains 操作符用来判断源 **Observable** 所发射的数据是否包含某一个数据。如果包含该数据，会返回 **true**；如果源 **Observable** 已经结束了却还没有发射这个数据，则返回 **false**。**isEmpty** 操作符用来判断源 **Observable** 是否发射过数据。如果发射过该数据，就会返回 **false**；如果源 **Observable** 已经结束了却还没有发射这个数据，则返回 **true**。

```
Observable.just(1,2,3).contains(1).subscribe(new Action1<Boolean>() {
    @Override
```

```

        public void call(Boolean aBoolean) {
            Log.d(TAG, "contains:"+aBoolean);
        }
    });
    Observable.just(1,2,3).isEmpty().subscribe(new Action1<Boolean>() {
        @Override
        public void call(Boolean aBoolean) {
            Log.d(TAG, "isEmpty:"+aBoolean);
        }
    });

```

输出结果为:

D/RxJava: contains:true

D/RxJava: isEmpty:false

2. 条件操作符

条件操作符有 `amb`、`defaultIfEmpty`、`skipUntil`、`skipWhile`、`takeUntil` 和 `takeWhile` 等, 这里介绍前两种。

(1) `amb`

`amb` 操作符对于给定两个或多个 `Observable`, 它只发射首先发射数据或通知的那个 `Observable` 的所有数据。

```

Observable.amb(Observable.just(1,2,3).delay(2, TimeUnit.SECONDS),Observable.
just(4,5,6)).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "amb:"+integer);
    }
});

```

第一个 `Observable` 延时 2s 发射, 所以很显然最终只会发射第二个 `Observable`, 输出结果为:

D/RxJava: amb:4

D/RxJava: amb:5

D/RxJava: amb:6

(2) `defaultIfEmpty`

发射来自原始 `Observable` 的数据。如果原始 `Observable` 没有发射数据, 就发射一个默认数据, 如下所示:


```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        subscriber.onCompleted();//1
    }
}).defaultIfEmpty(3).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d(TAG, "defaultIfEmpty:"+integer);
    }
});
```

在上面代码注释 1 处可以看出并没有发射数据而是直接调用 `onCompleted` 方法来完成事件，输出结果为：

D/RxJava: defaultIfEmpty:3

8.3.8 转换操作符

转换操作符用来将 `Observable` 转换为另一个对象或数据结构，转换操作符有 `toList`、`toSortedList`、`toMap`、`toMultiMap`、`getIterator` 和 `nest` 等，这里介绍前 3 种。

1. toList

`toList` 操作符将发射多项数据且为每一项数据调用 `onNext` 方法的 `Observable` 发射的多项数据组合成一个 `List`，然后调用一次 `onNext` 方法传递整个列表。

```
Observable.just(1,2,3).toList().subscribe(new Action1<List<Integer>>() {
    @Override
    public void call(List<Integer> integers) {
        for(int integer :integers){
            Log.d(TAG, "toList:" + integer);
        }
    }
});
```

输出结果为：

D/RxJava: toList:1

D/RxJava: toList:2

D/RxJava: toList:3

2. toSortedList

`toSortedList` 操作符类似于 `toList` 操作符；不同的是，它会对产生的列表排序，默认是自然

升序。如果发射的数据项没有实现 `Comparable` 接口，会抛出一个异常。当然，若发射的数据项没有实现 `Comparable` 接口，可以使用 `toSortedList(Func2)` 变体，其传递的函数参数 `Func2` 会作用于比较两个数据项。

```
Observable.just(3,1,2).toSortedList().subscribe(new Action1<List<Integer>>() {
    @Override
    public void call(List<Integer> integers) {
        for(int integer :integers){
            Log.d(TAG, "toSortedList:" + integer);
        }
    }
});
```

输出结果为：

D/RxJava: toSortedList:1

D/RxJava: toSortedList:2

D/RxJava: toSortedList:3

3.toMap

`toMap` 操作符收集原始 `Observable` 发射的所有数据项到一个 `Map`（默认是 `HashMap`），然后发射这个 `Map`。你可以提供一个用于生成 `Map` 的 `key` 的函数，也可以提供一个函数转换数据项到 `Map` 存储的值（默认数据项本身就是值）。下面的例子场景类似于 8.3.2 节的 `groupBy` 操作符，这里就不赘述了，例子如下所示。

```
Swordsman s1 = new Swordsman("韦一笑", "A");
Swordsman s2 = new Swordsman("张三丰", "SS");
Swordsman s3 = new Swordsman("周芷若", "S");
Observable.just(s1,s2,s3).toMap(new Func1<Swordsman, String>() {
    @Override
    public String call(Swordsman swordsman) {
        return swordsman.getLevel();//1
    }
}).subscribe(new Action1<Map<String, Swordsman>>() {
    @Override
    public void call(Map<String, Swordsman> stringSwordsmanMap) {
        Log.d(TAG, "toMap:" + stringSwordsmanMap.get("SS").getName()); //2
    }
});
```

在上面代码注释 1 处根据 `Swordsman` 的等级作为 `key` 值，在注释 2 处打印出 `key` 值为“SS”的名字，输出结果为：

D/TAG: toMap:张三丰

8.4 RxJava 的线程控制

1. 内置的 Scheduler

如果我们不指定线程，默认是在调用 `subscribe` 方法的线程上进行回调的。如果我们想切换线程，就需要使用 `Scheduler`。RxJava 已经内置了如下 5 个 `Scheduler`。

- `Schedulers.immediate()`: 直接在当前线程运行，它是 `timeout`、`timeInterval` 和 `timestamp` 操作符的默认调度器。
- `Schedulers.newThread()`: 总是启用新线程，并在新线程执行操作。
- `Schedulers.io()`: I/O 操作（读写文件、读写数据库、网络信息交互等）所使用的 `Scheduler`。行为模式和 `newThread()` 差不多，区别在于 `io()` 的内部实现是用一个无数量上限的线程池，可以重用空闲的线程，因此多数情况下 `io()` 比 `newThread()` 更有效率。
- `Schedulers.computation()`: 计算所使用的 `Scheduler`，例如图形的计算。这个 `Scheduler` 使用固定线程池，大小为 CPU 核数。不要把 I/O 操作放在 `computation()` 中，否则 I/O 操作的等待时间会浪费 CPU。它是 `buffer`、`debounce`、`delay`、`interval`、`sample` 和 `skip` 操作符的默认调度器。
- `Schedulers.trampoline()`: 当我们想在当前线程执行一个任务时，并不是立即时，可以用 `trampoline()` 将它入队。这个调度器将会处理它的队列并且按序运行队列中的每一个任务。它是 `repeat` 和 `retry` 操作符默认的调度器。

另外，RxAndroid 也提供了一个常用的 `Scheduler`：

- `AndroidSchedulers.mainThread()`——RxAndroid 库中提供的 `Scheduler`，它指定的操作在主线程中运行。

2. 控制线程

在 RxJava 中用 `subscribeOn` 和 `observeOn` 操作符来控制线程。它们的使用方法已经在 8.3.5 节中介绍过了。

8.5 RxJava 的使用场景

关于 RxJava，实际有很多可以使用它的场景，这里讲解 RxJava 结合 OkHttp、Retrofit 访问网络及用 RxJava 实现 RxBus。

8.5.1 RxJava 结合 OkHttp 访问网络

RxJava 结合 Retrofit 访问网络是比较好的搭配，当然 RxJava 结合 OkHttp 访问网络也是可以的。不了解 OkHttp 的读者可以查看 5.6 节的内容。这里仍旧访问 5.4.1 节用到的淘宝 IP 库。首先创建 `Observable`，代码如下所示：

```

private Observable<String> getObservable(final String ip) {
    Observable observable = Observable.create(new Observable.OnSubscribe<String>() {
        @Override
        public void call(final Subscriber<? super String> subscriber) {
            mOkHttpClient = new OkHttpClient();
            RequestBody formBody = new FormBody.Builder()
                .add("ip", ip)
                .build();
            Request request = new Request.Builder()
                .url("http://ip.taobao.com/service/getIpInfo.php")
                .post(formBody)
                .build();
            Call call = mOkHttpClient.newCall(request);
            call.enqueue(new Callback() {
                @Override
                public void onFailure(Call call, IOException e) {
                    subscriber.onError(new Exception("error"));
                }
                @Override
                public void onResponse(Call call, Response response) throws IOException {
                    String str = response.body().string();
                    subscriber.onNext(str); //1
                    subscriber.onCompleted();
                }
            });
        }
    });
    return observable;
}

```

我们将根据 OkHttp 的回调来定义事件的规则，在上面代码注释 1 处调用 `subscriber.onNext` 来将请求返回的数据添加到事件队列中。接下来实现观察者，如下所示：

```

private void postAsyncHttp(String size) {
    getObservable(size).subscribeOn(Schedulers.io()).observeOn(
        AndroidSchedulers.mainThread()).subscribe(new Subscriber<String>()
    {
        @Override
        public void onCompleted() {
            Log.d(TAG, "onCompleted");
        }
        @Override

```

```

        public void onError(Throwable e) {
            Log.d(TAG, e.getMessage());
        }
        @Override
        public void onNext(String s) {
            Log.d(TAG, s);
            Toast.makeText(getApplicationContext(), "请求成功", Toast.
                LENGTH_SHORT).show();
        }
    });
}

```

将访问网络回调设置为主线程，所以 Toast 是能正常显示的。因此，使用 RxJava 也解决了 OkHttp 的回调不在主线程的问题。最后调用 postAsyncHttp 方法传入 IP 地址就可以了。

8.5.2 RxJava 结合 Retrofit 访问网络

此节的例子是基于 5.7.1 节改编的，因此关于 Retrofit 的相关问题在本节不做赘述。这里仍旧访问淘宝 IP 库。

1. 使用前的准备工作

首先配置 build.gradle:

```

dependencies {
    ...
    compile 'io.reactivex:rxjava:1.2.0'
    compile 'io.reactivex:rxandroid:1.2.1'
    compile 'com.squareup.retrofit2:retrofit:2.1.0'
    compile 'com.squareup.retrofit2:converter-gson:2.1.0'
    compile 'com.squareup.retrofit2:adapter-rxjava:2.1.0'
}

```

其中最后一个配置是 Retrofit 中 RxJava 转换器的依赖。

2. 修改请求网络接口

Retrofit 的请求接口返回的是 Call。若结合 RxJava 使用，则需要把 Call 改为 Observable，代码如下所示：

```

public interface IpServiceForPost {
    @FormUrlEncoded
    @POST("getIpInfo.php")
    Observable<IpModel> getIpMsg(@Field("ip") String first);
}

```

3. 修改请求网络方法

```
private void postIpInformation(String ip) {
    String url = "http://ip.taobao.com/service/";
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(url)
        .addConverterFactory(GsonConverterFactory.create())
        .addCallAdapterFactory(RxJavaCallAdapterFactory.create())//1
        .build();
    IpServiceForPost ipService = retrofit.create(IpServiceForPost.
class);
    subscription=ipService.getIpMsg(ip).subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread()).subscribe(new
        Subscriber<IpModel>() {
            @Override
            public void onCompleted() {
                Log.d(TAG, "onCompleted");
            }
            @Override
            public void onError(Throwable e) {
                Log.d(TAG, "onError");
            }
            @Override
            public void onNext(IpModel ipModel) {
                String country=ipModel.getData().getCountry();
                Toast.makeText(getApplicationContext(), country, Toast.LENGTH_
SHORT).show();//2
            }
        });
}
```

在上面代码注释 1 处添加 `addCallAdapterFactory` 方法，这样此前定义的接口返回的就不是 `Call` 了，而是 `Observable`，这样就能用 `RxJava` 提供的方法来对请求网络的回调进行处理。在注释 2 处我们根据 `onNext` 方法回调得到的 `IpModel` 来获取传入的 IP 地址的国家信息。最后，只需要调用 `postIpInformation` 方法传入想要查询的 IP 地址就可以了。这里可以对 `Retrofit` 的创建过程以及 `RxJava` 的线程控制进行封装。因为这并不是本节的重点，所以这里就不进行封装了。

4. 请求返回数据格式封装

首先来看 `IpModel` 类：

```
public class IpModel {
    private int code;
    private IpData data;
    public void setCode(int code) {
```

```

        this.code = code;
    }
    public int getCode() {
        return this.code;
    }
    ...
}

```

淘宝 IP 库的返回数据格式就是 `int` 类型的 `code` 和 `IpData` 类型的 `data`。对于日常开发来说，返回的 `code` 值和 `message` 值一般是固定的，而 `data` 值则是不断变化的。如果每次 `data` 值变化我们都要带上 `code` 值和 `message` 值，这种写法显然不是很好。这里假设淘宝 IP 库的返回数据格式的 `data` 值是变化的，将 `IpModel` 类改写为如下代码：

```

public class HttpResult<T> {
    private int code;
    private T data;
    public int getCode() {
        return code;
    }
    public void setCode(int code) {
        this.code = code;
    }
    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
}

```

接着修改请求接口，如下所示：

```

public interface IpServiceForPost{
    @FormUrlEncoded
    @POST("getIpInfo.php")
    Observable<HttpResult<IpData>> getIpMsg(@Field("ip") String first);
}

```

因为知道请求返回的 `data` 值是 `IpData` 类型的，所以 `Observable` 泛型设置为 `HttpResult<IpData>`。最后修改请求网络方法，如下所示：

```

private void postIpInformation(String ip) {
    ...
    subscription=ipService.getIpMsg(ip).subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread()).subscribe(new

```

```

        Subscriber<HttpResult<IpData>>>() {
            @Override
            public void onCompleted() {
            }
            @Override
            public void onError(Throwable e) {
            }
            @Override
            public void onNext(HttpResult<IpData> ipDataHttpResult) {
                IpData data=ipDataHttpResult.getData();
                Toast.makeText(getApplicationContext(), data.getCountry(),
                    Toast.LENGTH_SHORT).show();
            }
        });
    }
}

```

5. 取消请求

如果只使用 Retrofit, 可以用 Call 的 cancel 方法来取消请求。在 RxJava 中可以使用如下代码:

```

@Override
public void onStop() {
    super.onStop();
    if (subscription != null && subscription.isUnsubscribed()) {
        subscription.unsubscribe();
    }
}
}

```

当然, 取消请求的方法也可以放在其他生命周期方法中, 其中 subscription 为 subscribe 方法的返回值。当调用 unsubscribe 方法时, 其内部会调用 Call 的 cancel 方法来取消请求。当然一个界面也可能会用多个请求。如果想取消多个请求, 可以使用 CompositeSubscription, 如下所示:

```

private CompositeSubscription compositeSubscription = new
CompositeSubscription();
...
// 发送请求的地方, 将返回的 subscription 添加到 compositeSubscription 中
compositeSubscription.add(subscription);
...
//取消所有请求
@Override
public void onStop() {
    super.onStop();
    compositeSubscription.unsubscribe();
}
}

```


8.5.3 用 RxJava 实现 RxBus

在第 7 章中我们学习了事件总线，本节我们用 RxJava 来实现事件总线 RxBus，以替代 EventBus 和 otto。

1. 创建 RxBus

首先创建 RxBus。这里的 RxBus 只是支持基本的功能，读者如果想要添加一些功能，则可以自定义添加，代码如下所示：

```
public class RxBus {
    private static volatile RxBus rxBus;//1
    private final Subject<Object, Object> subject = new SerializedSubject<>
(PublishSubject.create()); //3
    private RxBus() {
    }
    public static RxBus getInstance() {//2
        if (rxBus == null) {
            synchronized (RxBus.class) {
                if (rxBus == null) {
                    rxBus = new RxBus();
                }
            }
        }
        return rxBus;
    }
    public void post(Object ob) {
        subject.onNext(ob);
    }
    public <T> Observable<T> toObservable(Class<T> eventType) {
        return subject.ofType(eventType); //4
    }
}
```

上面代码注释 1 处用到了 volatile，不了解它的读者可以查看 4.2.4 节。在注释 2 处又用到了单例模式的双重检查模式，在 6.3.1 节介绍过它。因为 Subject 是非线程安全的，因此在注释 3 处将 Subject 转换为一个 SerializedSubject。注释 4 处的 subject 的 ofType 方法只会发送指定类型的数据。ofType 方法源码如下所示：

```
public final <R> Observable<R> ofType(final Class<R> klass) {
    return filter(InternalObservableUtils.isInstanceOf(klass))
        .cast(klass);
}
```

ofType 方法中包含了 filter 和 cast 的操作。通过 filter 操作符来判定是否是指定的类型，如

果不是就不提交给订阅者。cast 操作符则用来将 Observable 转换成指定类型的 Observable。

2. 发送事件

我们在 RxBusActivity 中定义一个 Button，点击这个 Button 时就会发送事件，代码如下所示：

```
public class RxBusActivity extends AppCompatActivity {
    private Button bt_post;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_rx_bus);
        bt_post= (Button) this.findViewById(R.id.bt_post);
        bt_post.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                RxBus.getInstance().post(new MessageEvent("用 RxJava 实现 RxBus")); //1
            }
        });
    }
}
```

上面代码注释 1 处仍旧发送 7.1.1 节中定义的 MessageEvent，其内部的 message 内容设置为“用 RxJava 实现 RxBus”。

3. 接收事件

在 RxBusFragment 中接收事件，如下所示：

```
public class RxBusFragment extends Fragment {
    ...
    @Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        subscription=
        RxBus.getInstance().toObservable(MessageEvent.class).subscribe(new
        Action1<MessageEvent>() {
            @Override
            public void call(MessageEvent messageEvent) {
                if(messageEvent!=null) {
                    tv_text.setText(messageEvent.getMessage()); //1
                }
            }
        });
    }
}
```

```

    }
}

```

在上面代码注释 1 处将接收到的事件内容显示到 TextView 上。

4. 取消订阅事件

当然要记得取消订阅事件，防止内存泄漏：

```

@Override
public void onDestroy() {
    super.onDestroy();
    if (subscription != null && !subscription.isUnsubscribed()) {
        subscription.unsubscribe();
    }
}

```

最后运行程序，点击“发送”按钮，效果如图 8-2 所示。当我们点击“发送”按钮时，RxBusFragment 顺利接收到了事件，并在 TextView 中显示接收到的 message 内容：“用 RxJava 实现 RxBus”。

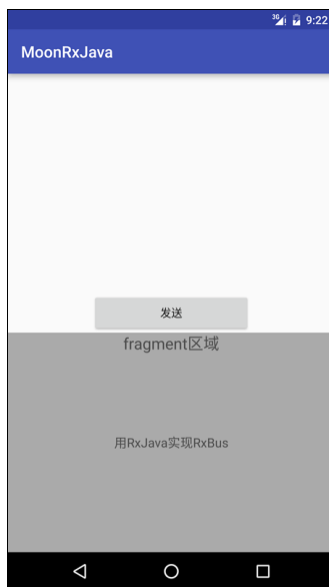


图 8-2 运行效果

8.6 RxJava 源码解析

RxJava 的源码解析会从 3 个方面去分析，分别是 RxJava 的订阅过程、变换过程和线程切换过程。

8.6.1 RxJava 的订阅过程

首先从最基本的 RxJava 的订阅过程开始分析。先看一段 RxJava 的基本使用方法，如下所示：

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        subscriber.onNext(1);
    }
}).subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
    }
    @Override
    public void onError(Throwable e) {
    }
    @Override
    public void onNext(Integer integer) {
    }
});
```

之后查看 Observable 的 create 方法做了什么：

```
public static <T> Observable<T> create(OnSubscribe<T> f) {
    return new Observable<T>(hook.onCreate(f));
}
```

在此可以看出，create 方法主要就是创建了 Observable 类，其中，hook 指的是 RxJava-ObservableExecutionHook。它的 onCreate 方法如下所示：

```
public <T> OnSubscribe<T> onCreate(OnSubscribe<T> f) {
    return f;
}
```

RxJavaObservableExecutionHook 的 onCreate 方法并没有做什么，只是返回了传进来的 OnSubscribe。Observable（被观察者）已经产生了，把它标记为 ob1，后面会用到它。接着回来查看 Observable 的构造方法，如下所示：

```
protected Observable(OnSubscribe<T> f) {
    this.onSubscribe = f;
}
```

在 Observable 也就是 ob1 的构造方法里，将传进来的 OnSubscribe 赋值给 onSubscribe 属性，这个 onSubscribe 我们标记为 on1，后面也会用到它。接下来会调用 Observable 的 subscribe 方

法并传入 Observer（观察者），完成订阅操作。现在来查看 Observable 的 subscribe 方法做了什么，如下所示：

```
static <T> Subscription subscribe(Subscriber<? super T> subscriber,
Observable<T> observable) {
    ...
    subscriber.onStart();
    if (!(subscriber instanceof SafeSubscriber)) {
        subscriber = new SafeSubscriber<T>(subscriber); //1
    }
    try {
        hook.onSubscribeStart(observable, observable.onSubscribe).call
            (subscriber); //2
        return hook.onSubscribeReturn(subscriber);
    } catch (Throwable e) {
        ...
        return Subscriptions.unsubscribed();
    }
}
```

在上面代码注释 1 处，如果传进来的 Subscriber 不是 SafeSubscriber 类型的，则将它封装成 SafeSubscriber。SafeSubscriber 继承自 Subscriber，它在 Subscriber 的基础上添加了更加完善的处理。比如在 onComplete 和 onError 方法调用时不会再调用 onNext 方法，并且保证 onComplete 和 onError 方法只有一个被执行。在注释 2 处，onSubscribeStart 方法如下所示：

```
public <T> OnSubscribe<T> onSubscribeStart(Observable<? extends T>
observableInstance, final OnSubscribe<T> onSubscribe) {
    return onSubscribe;
}
```

它会返回 OnSubscribe。因此，注释 2 处就是调用 OnSubscribe.call(subscriber)来完成订阅的。

8.6.2 RxJava 的变换过程

将前面 RxJava 基本用法的代码改为如下代码：

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        subscriber.onNext(1);
    }
}).map(new Func1<Integer, String>() {
    @Override
    public String call(Integer integer) {
```

```

        return "a" + integer;
    }
}).subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
    }
});

```

这里用到了变换操作符 `map`。现在查看 `map` 方法做了什么，如下所示：

```

public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {
    return lift(new OperatorMap<T, R>(func));
}

```

这里创建了 `OperatorMap`，并将 `func` 传进去。现在查看 `OperatorMap` 做了什么，如下所示：

```

public final class OperatorMap<T, R> implements Operator<R, T> {
    final Func1<? super T, ? extends R> transformer;
    public OperatorMap(Func1<? super T, ? extends R> transformer) {
        this.transformer = transformer;
    }
    @Override
    public Subscriber<? super T> call(final Subscriber<? super R> o) {
        MapSubscriber<T, R> parent = new MapSubscriber<T, R>(o, transformer);
        o.add(parent);
        return parent;
    }
    static final class MapSubscriber<T, R> extends Subscriber<T> {
        ...
    }
}

```

`OperatorMap` 实现了 `Operator` 接口的 `call` 方法，且在 `call` 方法中创建了 `MapSubscriber` 并返回。接着回到 `map` 方法，来查看 `lift` 方法，如下所示：

```

public final <R> Observable<R> lift(final Operator<? extends R, ? super T>
operator) {
    return new Observable<R>(new OnSubscribeLift<T, R>(onSubscribe, operator));
}

```

在此可以看到 `lift` 方法返回一个新建的 `Observable` 对象，这里标记为 `ob2`，并传入了一个 `OnSubscribeLift` 对象。此前讲过 `Observable` 的构造方法，因此可以得知这个 `OnSubscribeLift` 会作为 `ob2` 的 `OnSubscribe` 属性，这里把它标记为 `on2`。`OnSubscribeLift` 需要传入的参数分别为 `onSubscribe` 和 `operator`，`operator` 指的就是 `OperatorMap`，而 `onSubscribe` 是前文标记的 `on1`。

OnSubscribeLift 的代码如下所示:

```
public final class OnSubscribeLift<T, R> implements OnSubscribe<R> {
    static final RxJavaObservableExecutionHook hook = RxJavaPlugins.
        getInstance().getObservableExecutionHook();
    final OnSubscribe<T> parent;
    final Operator<? extends R, ? super T> operator;
    public OnSubscribeLift(OnSubscribe<T> parent, Operator<? extends R, ?
        super T> operator) {
        this.parent = parent; //1
        this.operator = operator;
    }
    @Override
    public void call(Subscriber<? super R> o) {
        try {
            Subscriber<? super T> st = hook.onLift(operator).call(o); //2
            try {
                st.onStart();
                parent.call(st); //3
            } catch (Throwable e) {
                Exceptions.throwIfFatal(e);
                st.onError(e);
            }
        } catch (Throwable e) {
            Exceptions.throwIfFatal(e);
            o.onError(e);
        }
    }
}
```

在上面代码注释 1 处传进来的 parent 是 on1, 在注释 2 处调用了 hook.onLift(operator)。查看 onLift 方法里做了什么:

```
public <T, R> Operator<? extends R, ? super T> onLift(final Operator<?
    extends R, ? super T> lift) {
    return lift;
}
```

它只是返回了 lift。这个 lift 指的就是 OperatorMap。接下来调用 OperatorMap 的 call 方法。前面讲到过 OperatorMap 的 call 方法会返回 MapSubscriber 类, 我们把 MapSubscriber 标记为 sub2。在注释 3 处完成 sub2 的订阅, 相当于 on1.call(sub2)。在我们调用完 map 方法后又调用了 subscribe 方法, 我们传入了一个 Action1, Action1 最终会封装成 Subscriber 类型。这里我们把自己传入的 Action1 标记为 sub1。subscribe 方法我们前面讲过, 它会调用如下方法:

```
hook.onSubscribeStart(observable, observable.onSubscribe).call(subscriber);
```

因为此前调用过 `map` 操作符，所以这里传入的 `observable.onSubscribe` 指的是 `on2`。在分析 RxJava 的订阅过程时，我们得知 `onSubscribeStart` 方法会返回 `onSubscribe`，也就是 `on2`，相当于 `on2.call(sub1)`。`on2` 指的是 `OnSubscribeLift`，因此再回到 `OnSubscribeLift` 的 `call` 方法，在那里的注释 3 处调用 `on1` 的 `call` 方法。`on1` 的 `call` 方法是这样自定义的：

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        subscriber.onNext(1);
    }
})
```

这里的 `subscriber` 是 `sub2`，也就是 `MapSubscriber`。现在查看 `MapSubscriber` 的 `onNext` 方法：

```
public final class OperatorMap<T, R> implements Operator<R, T> {
    ...
    static final class MapSubscriber<T, R> extends Subscriber<T> {
        final Subscriber<? super R> actual;
        final Func1<? super T, ? extends R> mapper;
        boolean done;
        public MapSubscriber(Subscriber<? super R> actual, Func1<? super T, ?
        extends R> mapper) {
            this.actual = actual;
            this.mapper = mapper;
        }
        @Override
        public void onNext(T t) {
            R result;
            try {
                result = mapper.call(t);
            } catch (Throwable ex) {
                ...
            }
            actual.onNext(result); //1
        }
        ...
    }
}
```

在上面代码注释 1 处又调用了 `actual` 的 `onNext` 方法，`actual` 指的就是 `sub1`，也就是我们传入的 `Action1`。RxJava 的变换过程就讲到这里了。总之，RxJava 的变换过程有些绕，因此这里结合上文的源码分析给出一个简单的流程图（如图 8-3 所示）。

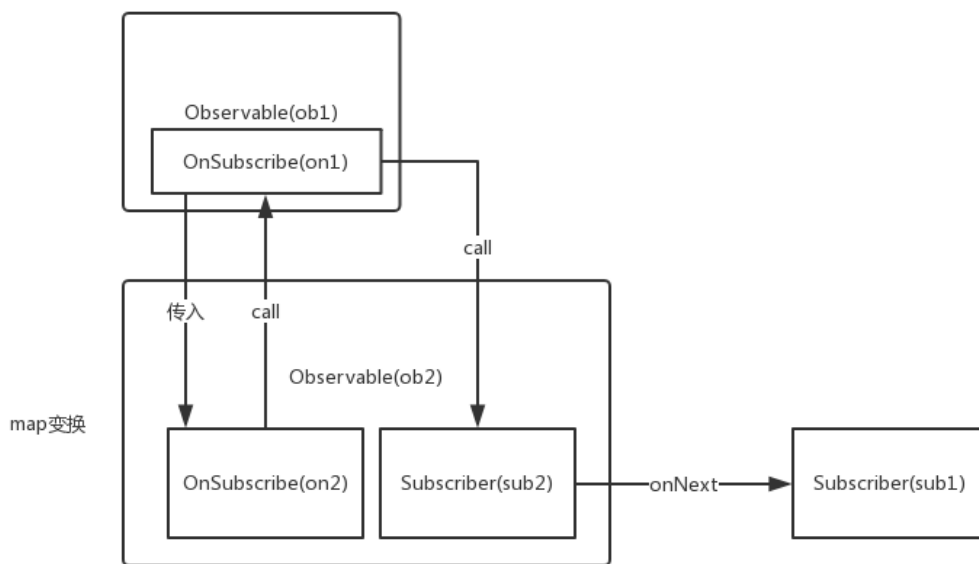


图 8-3 变换过程流程图

8.6.3 RxJava 的线程切换过程

为了分析 RxJava 的线程切换过程，这里仍旧拿一段代码举例，如下所示：

```

Observable<Integer> obs= Observable.create(new Observable.OnSubscribe
<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        subscriber.onNext(1);
        subscriber.onCompleted();
    }
});
obs.subscribeOn(Schedulers.newThread()).observeOn
(AndroidSchedulers.mainThread()).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
    }
});

```

1. subscribeOn 分析

创建 Observable 后，调用它的 subscribeOn 方法。现在先来查看 subscribeOn 方法做了什么，如下所示：

```

public final Observable<T> subscribeOn(Scheduler scheduler) {
    if (this instanceof ScalarSynchronousObservable) {
        return ((ScalarSynchronousObservable<T>)this).scalarScheduleOn

```

```

        (scheduler);
    }
    return create(new OperatorSubscribeOn<T>(this, scheduler)); //1
}

```

上面代码注释 1 处的 `create` 方法仍旧是生成一个新的 `Observable`，并传入一个 `OperatorSubscribeOn` 类。`OperatorSubscribeOn` 需要传入两个参数：第一个参数 `this`，指的是我们最先创建的 `Observable`，第二个参数是一个 `Scheduler`。`OperatorSubscribeOn` 的代码如下所示：

```

public final class OperatorSubscribeOn<T> implements OnSubscribe<T> {
    final Scheduler scheduler;
    final Observable<T> source;
    public OperatorSubscribeOn(Observable<T> source, Scheduler scheduler) {
        this.scheduler = scheduler;
        this.source = source;
    }
    @Override
    public void call(final Subscriber<? super T> subscriber) {
        final Worker inner = scheduler.createWorker(); //1
        subscriber.add(inner);
        inner.schedule(new Action0() { //2
            @Override
            public void call() {
                final Thread t = Thread.currentThread();

                Subscriber<T> s = new Subscriber<T>(subscriber) {
                    @Override
                    public void onNext(T t) {
                        subscriber.onNext(t);
                    }
                };
                ...
            }
        });
    }
}

```

上面代码注释 1 处创建了 `Worker`，并在注释 2 处调用了 `Worker` 的 `schedule` 方法。这个 `Worker` 可以说就是线程处理的代理执行者，它是调用 `Scheduler` 的 `createWorker` 方法创建的。我们在 8.6.3 节开头的代码示例中传入的 `Scheduler` 是 `Schedulers.newThread()`，因此先来查看 `Scheduler` 的 `newThread` 方法做了什么，如下所示：

```

public static Scheduler newThread() {
    return getInstance().newThreadScheduler();
}

```

`Scheduler` 是一个单例类，其返回自身的 `newThreadScheduler` 属性。这个属性最终指的是 `NewThreadScheduler`，它的代码如下所示：

```
public final class NewThreadScheduler extends Scheduler {
    private final ThreadFactory threadFactory;
    public NewThreadScheduler(ThreadFactory threadFactory) {
        this.threadFactory = threadFactory;
    }
    @Override
    public Worker createWorker() {
        return new NewThreadWorker(threadFactory); //1
    }
}
```

此前在 `OperatorSubscribeOn` 中调用了 `Scheduler` 的 `createWorker` 方法，其实就是调用 `NewThreadScheduler` 的 `createWorker` 方法。上面代码注释 1 处返回了新建的 `NewThreadWorker`：

```
public NewThreadWorker(ThreadFactory threadFactory) {
    ScheduledExecutorService exec = Executors.newScheduledThreadPool(1,
        threadFactory); //1
    boolean cancelSupported = tryEnableCancelPolicy(exec);
    if (!cancelSupported && exec instanceof ScheduledThreadPoolExecutor) {
        registerExecutor((ScheduledThreadPoolExecutor)exec);
    }
    schedulersHook = RxJavaPlugins.getInstance().getSchedulersHook();
    executor = exec;
}
```

在上面代码注释 1 处，看到了我们熟悉的线程池，不了解的读者请查看本书的 4.4 节。关于线程池的细节这里就不去查看了，接下来回头查看 `OperatorSubscribeOn` 类。可以看到在第 377 页 `OperatorSubscribeOn` 类的注释 2 处调用了 `Worker` 的 `schedule` 方法，而 `Worker` 指的是 `NewThreadWorker`。接下来查看 `NewThreadWorker` 的 `schedule` 方法：

```
@Override
public Subscription schedule(final Action0 action, long delayTime,
    TimeUnit unit) {
    if (isUnsubscribed) {
        return Subscriptions.unsubscribed();
    }
    return scheduleActual(action, delayTime, unit);
}
```

之后查看返回的 `scheduleActual` 方法：

```
public ScheduledAction scheduleActual(final Action0 action, long
```

```

delayTime, TimeUnit unit) {
    Action0 decoratedAction = schedulersHook.onSchedule(action);
    ScheduledAction run = new ScheduledAction(decoratedAction);
    Future<?> f;
    if (delayTime <= 0) {
        f = executor.submit(run);
    } else {
        f = executor.schedule(run, delayTime, unit);
    }
    run.add(f);
    return run;
}

```

可以看到最终线程切换的处理均由线程池处理。`subscribeOn` 就分析到这里，接下来分析 `observeOn`。

2. observeOn 分析

`observeOn` 方法如下所示：

```

public final Observable<T> observeOn(Scheduler scheduler, boolean delayError,
int bufferSize) {
    if (this instanceof ScalarSynchronousObservable) {
        return ((ScalarSynchronousObservable<T>)this).scalarScheduleOn
            (scheduler);
    }
    return lift(new OperatorObserveOn<T>(scheduler, delayError, bufferSize));
}

```

`lift` 方法此前讲过，这里就不赘述了。现在查看 `OperatorObserveOn` 类：

```

public final class OperatorObserveOn<T> implements Operator<T, T> {
    ...
    public OperatorObserveOn(Scheduler scheduler, boolean delayError, int
bufferSize) {
        this.scheduler = scheduler;
        this.delayError = delayError;
        this.bufferSize = (bufferSize > 0) ? bufferSize : RxRingBuffer.SIZE;
    }
    @Override
    public Subscriber<? super T> call(Subscriber<? super T> child) {
        if (scheduler instanceof ImmediateScheduler) {
            return child;
        } else if (scheduler instanceof TrampolineScheduler) {
            return child;
        } else {

```

```

        ObserveOnSubscriber<T> parent = new ObserveOnSubscriber<T>
            (scheduler, child, delayError, bufferSize); //1
        parent.init();
        return parent;
    }
}
...
}

```

`OperatorObserveOn` 和此前讲过的 `OperatorMap` 类似，在上面代码注释 1 处新建了 `ObserveOnSubscriber` 类：

```

private static final class ObserveOnSubscriber<T> extends Subscriber<T>
    implements Action0 {
    ...
    @Override
    public void onNext(final T t) {
        if (isUnsubscribed() || finished) {
            return;
        }
        if (!queue.offer(on.next(t))) {
            onError(new MissingBackpressureException());
            return;
        }
        schedule();
    }
    @Override
    public void onCompleted() {
        ...
        schedule();
    }
    @Override
    public void onError(final Throwable e) {
        ...
        schedule();
    }
    ...
}

```

我们可以发现在 `onNext`、`onCompleted` 和 `onError` 方法的最后都调用了 `schedule` 方法：

```

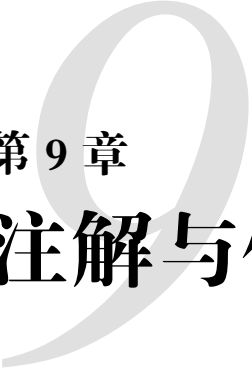
protected void schedule() {
    if (counter.getAndIncrement() == 0) {
        recursiveScheduler.schedule(this);
    }
}

```

`recursiveScheduler` 是一个 Worker, `this` 指的是 `observeOnSubscriber`, 这意味着 `observeOnSubscriber` 的 `onNext` 方法都被切换到 `recursiveScheduler` 的线程做处理, 从而达到线程切换的目的。

8.7 本章小结

本章主要介绍了 RxJava 的使用方法和原理。有关 RxJava 的知识完全可以写一本书, 在篇幅有限的基础上笔者并不能讲解 RxJava 的所有知识点, 比如本章也只介绍了 RxJava 的部分操作符。RxJava 的操作符就如同 API, 我们只需要记住分类, 并且根据分类来学习主要的操作符, 其他的操作符在应用时去查文档就可以了。另外, RxJava 的使用场景也有很多, 这里也只是介绍了 3 种, 其余的使用场景则需要读者查阅更多的资料了解, 并在实际开发中去逐步积累。



第 9 章

注解与依赖注入框架

在许多程序设计语言里，比如 Java、C#，依赖注入是一种较流行的设计模式，在 Android 开发中也有很多实用的依赖注入框架，可以帮助我们少写一些样板代码，达到各个类之间解耦的目的。在本章中我们就来学习依赖注入的原理，以及常用的依赖注入框架 **ButterKnife** 和 **Dagger2**。由于要分析 **ButterKnife** 和 **Dagger2** 的原理，而这两个框架都是使用编译时注解，因此，在本章开头我们首先来学习一下注解。

9.1 注解

注解有很多知识点，并且不是本章的重点，这里讲解注解只是为了后文要分析 **ButterKnife** 和 **Dagger2** 的原理。因此，只会讲解一小部分与 **ButterKnife** 和 **Dagger2** 相关的注解知识点。从 JDK 5 开始，Java 增加了注解，注解是代码里的特殊标记，这些标记可以在编译、类加载、运行时被读取，并执行相应的处理。通过使用注解，开发人员可以在不改变原有逻辑的情况下，在源文件中嵌入一些补充的信息。代码分析工具、开发工具和部署工具可以通过这些补充信息进行验证、处理或者进行部署。

9.1.1 注解分类

注解分为标准注解和元注解，下面会分别介绍它们。

1. 标准注解

标准注解有以下 4 种。

- **@Override**：对覆盖超类中的方法进行标记，如果被标记的方法并没有实际覆盖超类中的方法，则编译器会发出错误警告。

- **@Deprecated**: 对不鼓励使用或者已过时的方法添加注解, 当编程人员使用这些方法时, 将会在编译时显示提示信息。
- **@SuppressWarnings**: 选择性地取消特定代码段中的警告。
- **@SafeVarargs**: JDK 7 新增, 用来声明使用了可变长度参数的方法, 其在与泛型类一起使用时不会出现类型安全问题。

2. 元注解

除了标准注解, 还有元注解, 它用来注解其他注解, 从而创建新的注解。元注解有以下几种。

- **@Target**: 注解所修饰的对象范围。
- **@Inherited**: 表示注解可以被继承。
- **@Documented**: 表示这个注解应该被 JavaDoc 工具记录。
- **@Retention**: 用来声明注解的保留策略。
- **@Repeatable**: JDK 8 新增, 允许一个注解在同一声明类型 (类、属性或方法) 上多次使用。

其中 **@Target** 注解取值是一个 `ElementType` 类型的数组, 其中有以下几种取值, 对应不同的对象范围。

- `ElementType.TYPE`: 能修饰类、接口或枚举类型。
- `ElementType.FIELD`: 能修饰成员变量。
- `ElementType.METHOD`: 能修饰方法。
- `ElementType.PARAMETER`: 能修饰参数。
- `ElementType.CONSTRUCTOR`: 能修饰构造方法。
- `ElementType.LOCAL_VARIABLE`: 能修饰局部变量。
- `ElementType.ANNOTATION_TYPE`: 能修饰注解。
- `ElementType.PACKAGE`: 能修饰包。
- `ElementType.TYPE_PARAMETER`: 类型参数声明。
- `ElementType.TYPE_USE`: 使用类型。

其中 **@Retention** 注解有 3 种类型, 分别表示不同级别的保留策略。

- `RetentionPolicy.SOURCE`: 源码级注解。注解信息只会保留在 .java 源码中, 源码在编译后, 注解信息被丢弃, 不会保留在 .class 中。
- `RetentionPolicy.CLASS`: 编译时注解。注解信息会保留在 .java 源码以及 .class 中。当运行 Java 程序时, JVM 会丢弃该注解信息, 不会保留在 JVM 中。
- `RetentionPolicy.RUNTIME`: 运行时注解。当运行 Java 程序时, JVM 也会保留该注解信息, 可以通过反射获取该注解信息。

9.1.2 定义注解

1. 基本定义

定义新的注解类型使用 `@interface` 关键字，这与定义一个接口很像，如下所示：

```
public @interface Swordsman {
    ...
}
```

定义完注解后，就可以在程序中使用该注解：

```
@Swordsman
public class AnnotationTest {
    ...
}
```

2. 定义成员变量

注解只有成员变量，没有方法。注解的成员变量在注解定义中以“无形参的方法”形式来声明，其“方法名”定义了该成员变量的名字，其返回值定义了该成员变量的类型：

```
public @interface Swordsman {
    String name();
    int age();
}
```

上面的代码定义了两个成员变量，这两个成员变量以方法的形式来定义。定义了成员变量后，使用该注解时就应该为该注解的成员变量指定值：

```
public class AnnotationTest {
    @Swordsman(name = "张无忌", age = 23)
    public void fighting() {
        ...
    }
}
```

也可以在定义注解的成员变量时，使用 `default` 关键字为其指定默认值，如下所示：

```
public @interface Swordsman {
    String name() default "张无忌";
    int age() default 23;
}
```

因为注解定义了默认值，所以使用时可以不为这些成员变量指定值，而是直接使用默认值：

```
public class AnnotationTest {
    @Swordsman
```

```

    public void fighting() {
        ...
    }
}

```

3. 定义运行时注解

可以用 `@Retention` 来设定注解的保留策略，这 3 个策略的生命周期长度为 `SOURCE < CLASS < RUNTIME`。生命周期短的能起作用的地方，生命周期长的一定也能起作用。一般如果需要在运行时去动态获取注解信息，那只能用 `RetentionPolicy.RUNTIME`；如果要在编译时进行一些预处理操作，比如生成一些辅助代码，就用 `RetentionPolicy.CLASS`；如果只是做一些检查性的操作，比如 `@Override` 和 `@SuppressWarnings`，则可选用 `RetentionPolicy.SOURCE`。当设定为 `RetentionPolicy.RUNTIME` 时，这个注解就是运行时注解，如下所示：

```

@Retention(RetentionPolicy.RUNTIME)
public @interface Swordsman {
    String name() default "张无忌";
    int age() default 23;
}

```

4. 定义编译时注解

同样地，如果将 `@Retention` 的保留策略设定为 `RetentionPolicy.CLASS`，这个注解就是编译时注解，如下所示：

```

@Retention(RetentionPolicy.CLASS)
public @interface Swordsman {
    String name() default "张无忌";
    int age() default 23;
}

```

9.1.3 注解处理器

如果没有处理注解的工具，那么注解也不会有什么大的作用。对于不同的注解有不同的注解处理器。虽然注解处理器的编写会千变万化，但是其也有处理标准，比如：针对运行时注解会采用反射机制处理，针对编译时注解会采用 `AbstractProcessor` 来处理。本节就针对前面讲到的运行时注解和编译时注解来编写注解处理器。

1. 运行时注解处理器

处理运行时注解需要用到反射机制。首先我们要定义运行时注解，如下所示：

```

@Documented
@Target(METHOD)
@Retention(RUNTIME)
public @interface GET {
}

```

```
String value() default "";
}
```

上面的代码是 Retrofit 中定义的 @GET 注解。其定义了 @Target(METHOD)，这等效于 @Target(ElementType.METHOD)，意味着 GET 注解应用于方法。接下来应用该注解，如下所示：

```
public class AnnotationTest {
    @GET(value = "http://ip.taobao.com/59.108.54.37")
    public String getIpMsg() {
        return "";
    }
    @GET(value = "http://ip.taobao.com/")
    public String getIp() {
        return "";
    }
}
```

上面的代码为 @GET 的成员变量赋值。接下来写一个简单的注解处理器，如下所示：

```
public class AnnotationProcessor {
    public static void main(String[] args) {
        Method[] methods = AnnotationTest.class.getDeclaredMethods();
        for (Method m:methods){
            GET get= m.getAnnotation(GET.class);
            System.out.println(get.value());
        }
    }
}
```

上面的代码用到了两个反射方法：getDeclaredMethods 和 getAnnotation，它们都属于 AnnotatedElement 接口，Class、Method 和 Filed 等类都实现了该接口。调用 getAnnotation 方法返回指定类型的注解对象，也就是 GET。最后调用 GET 的 value 方法返回从 GET 对象中提取元素的值。输出结果为：

```
http://ip.taobao.com/59.108.54.37
http://ip.taobao.com/
```

2. 编译时注解处理器

处理编译时注解的步骤稍微有点多，首先我们仍旧要先定义注解。

(1) 定义注解

这里首先在项目中新建一个 Java Library 来专门存放注解，这个 Library 名为 annotations。接下来定义注解，如下所示：

```

@Retention(CLASS)
@Target(FIELD)
public @interface BindView {
    int value() default 1;
}

```

上面代码中定义的注解类似于 ButterKnife 的 @BindView 注解。

(2) 编写注解处理器

我们在项目中再新建一个 Java Library 来存放注解处理器，这个 Library 名为 processor。我们来配置 processor 库的 build.gradle:

```

apply plugin: 'java'
dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    compile project(':annotations')
}
sourceCompatibility = "1.7"
targetCompatibility = "1.7"

```

接下来编写注解处理器 ClassProcessor，它继承 AbstractProcessor，如下所示：

```

public class ClassProcessor extends AbstractProcessor {

    @Override
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
    }

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv) {
        ...
        return true;
    }

    @Override
    public Set<String> getSupportedAnnotationTypes() {
        Set<String> annotataions = new LinkedHashSet<String>();
        annotataions.add(BindView.class.getCanonicalName());
        return annotataions;
    }

    @Override
    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.latestSupported();
    }
}

```

```

    }
}
}

```

process 方法的实现会在后文讲到，这里首先分别介绍这 4 个方法的作用。

- **init**: 被注解处理工具调用，并输入 `ProcessingEnvironment` 参数。`ProcessingEnvironment` 提供很多有用的工具类，比如 `Elements`、`Types`、`Filer` 和 `Messenger` 等。
- **process**: 相当于每个处理器的主函数 `main()`，在这里写你的扫描、评估和处理注解的代码，以及生成 Java 文件。输入参数 `RoundEnvironment`，可以让你查询出包含特定注解的被注解元素。
- **getSupportedAnnotationTypes**: 这是必须指定的方法，指定这个注解处理器是注册给哪个注解的。注意，它的返回值是一个字符串的集合，包含本处理器想要处理的注解类型的合法全称。
- **getSupportedSourceVersion**: 用来指定你使用的 Java 版本，通常这里返回 `SourceVersion.latestSupported()`。

在 Java 7 以后，也可以使用注解来代替 `getSupportedAnnotationTypes` 方法和 `getSupportedSourceVersion` 方法，如下所示：

```

@SupportedSourceVersion(SourceVersion.RELEASE_8)
@SupportedAnnotationTypes("com.example.annotation.cls.BindView")
public class ClassProcessor extends AbstractProcessor {
    @Override
    public synchronized void init(ProcessingEnvironment env){
        super.init(env);
    }

    @Override
    public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment env) {
        ...
    }
}

```

但是考虑到 Android 兼容性的问题，这里不建议采用这种注解的方式。接下来编写还未实现的 `process` 方法，如下所示：

```

public boolean process(Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv) {
    Messenger messenger = processingEnv.getMessenger();
    for (Element element : roundEnv.getElementsAnnotatedWith(BindView.class)) {
        if (element.getKind() == ElementKind.FIELD) {;

```

```

        messenger.printMessage(Diagnostic.Kind.NOTE, "printMessage:"
            + element.toString());
    }
}
return true;
}

```

这里用到 `Messenger` 的 `printMessage` 方法来打印出注解修饰的成员变量的名称, 这个名称会在 `Android Studio` 的 `Gradle Console` 窗口中打印出来。

(3) 注册注解处理器

为了使用注解处理器, 需要用服务文件来注册它。现在我们就来创建这个服务文件。首先在 `processor` 库的 `main` 目录下新建 `resources` 资源文件夹, 接下来在 `resources` 中再建立 `META-INF/services` 目录文件夹。最后在 `META-INF/services` 中创建 `javax.annotation.processing.Processor` 文件, 这个文件中的内容是注解处理器的名称。这里我们的 `javax.annotation.processing.Processor` 文件的内容为 `com.example.processor.ClassProcessor`。整个项目的目录结构如图 9-1 所示。

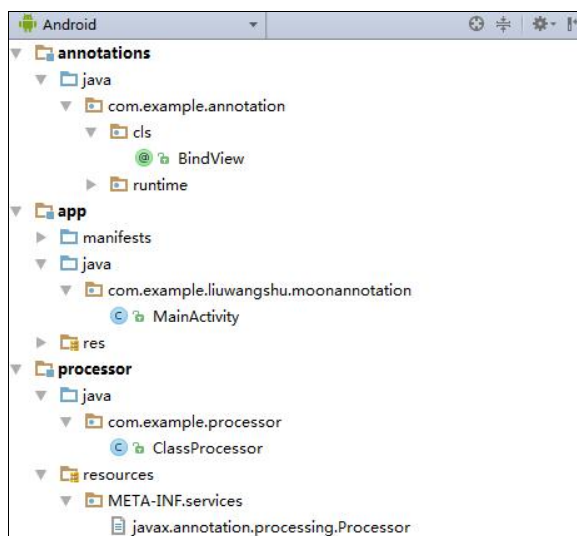


图 9-1 项目目录结构

如果你觉得前面创建服务文件的步骤比较麻烦, 也可以使用 `Google` 开源的 `AutoService`, 它用来帮助开发者生成 `META-INF/services/javax.annotation.processing.Processor` 文件。首先我们添加该开源库, 可以在 `File`→`Project Structure` 搜索“`auto-service`”查找该库并添加, 如图 9-2 所示。也可以在 `processor` 的 `build.gradle` 中直接添加如下代码:

```

dependencies {
    ...
    compile 'com.google.auto.service:auto-service:1.0-rc2'
}

```

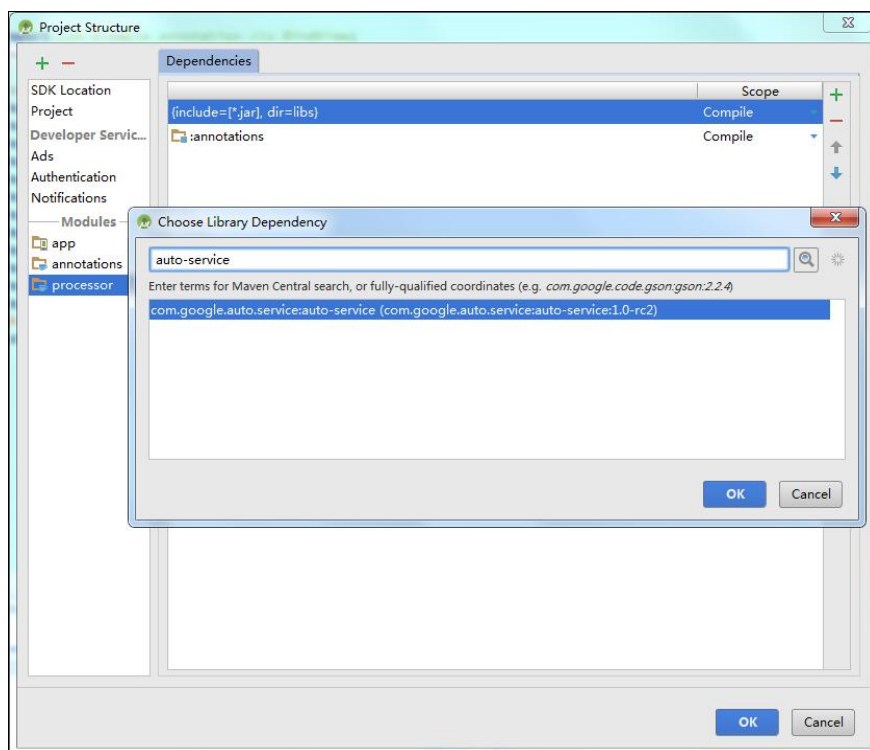


图 9-2 添加 AutoService 库

最后在注解处理器 `ClassProcessor` 中添加 `@AutoService(Processor.class)` 就可以了：

```
@AutoService(Processor.class)
public class ClassProcessor extends AbstractProcessor {
    ...
}
```

(4) 应用注解

接下来在我们的主工程项目（`app`）中引用注解。首先要在主工程项目的 `build.gradle` 中引用 `annotations` 和 `processor` 这两个库：

```
dependencies {
    ...
    compile project(':annotations')
    compile project(':processor')
}
```

接下来在 `MainActivity` 中应用注解，如下所示：

```
public class MainActivity extends AppCompatActivity {
    @BindView(value = R.id.tv_text)
    TextView tv_text;
```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
}

```

最后，我们先 Clean Project 再 Make Project，在 Gradle Console 窗口中打印的结果如图 9-3 所示。

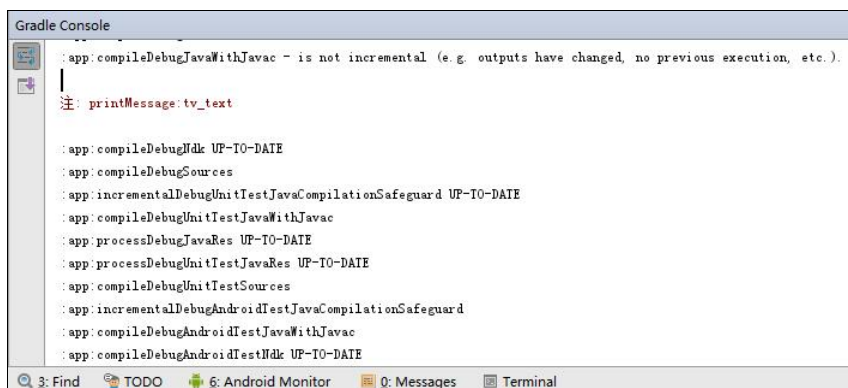


图 9-3 Make Project 打印结果

可以发现编译时会打印出 `@BindView` 注解修饰的成员变量名：`tv_text`。

(5) 使用 android-apt 插件

我们的主工程项目（app）中引用了 `processor` 库，但注解处理器只在编译处理期间需要用到，编译处理完后就没有实际作用了，而主工程项目添加了这个库会引入很多不必要的文件。为了处理这个问题我们需要引入插件 `android-apt`。它主要有两个作用：

- 仅仅在编译时期去依赖注解处理器所在的函数库并进行工作，但不会打包到 APK 中。
- 为注解处理器生成的代码设置好路径，以便 Android Studio 能够找到它。

接下来介绍如何使用它。首先需要在整个工程（Project）的 `build.gradle` 中添加如下语句：

```

buildscript {
    ...
    dependencies {
        ...
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}

```

接下来在主工程项目（app）的 `build.gradle` 中以 `apt` 的方式引入注解处理器 `processor`，如下所示：


```

...
apply plugin: 'com.neenbedankt.android-apt'
...
dependencies {
...
    //compile project(':processor')
    apt project(':processor')
}

```

9.2 依赖注入的原理

讲完了注解，我们再来学习依赖注入。如果只是单纯介绍依赖注入，本节内容可能会比较难以理解。因此我们就先从控制反转讲起，通过控制反转我们能更好地理解依赖注入。

9.2.1 控制反转与依赖注入

1. 控制反转

在讲到依赖注入前，这里有必要举一个比较形象的例子。笔者非常喜欢机械手表，通过机械表的背透或者打开后盖，会发现里面有很多齿轮，类似于图 9-4 这样。这些齿轮相互独立并且相互啮合在一起，协同工作，组成一个齿轮组去完成某一项任务。如果这些齿轮中有一个出现了问题，可能就会影响整个齿轮组的正常运作。齿轮组中齿轮之间的啮合关系与软件系统中对象之间的耦合关系非常相似。对象之间的耦合关系是无法避免的，而且随着工业级应用的规模越来越庞大，对象之间的依赖关系也越来越复杂，经常会出现对象之间的多重依赖性关系。为了解决对象之间耦合度过高的问题，软件专家 Michael Mattson 提出了 IoC 理论，用来实现对象之间的解耦。IoC 是 Inversion of Control 的缩写，即控制反转。IoC 理论提出的观点大体是这样的：借助于“第三方”实现具有依赖关系的对象之间的解耦。如图 9-5 所示，引入 IoC 容器后，使得 A、B、C、D 这 4 个对象没有了耦合关系，齿轮之间的传动全部依靠 IoC 容器。如果去掉中间的 IoC 容器，我们就会发现 A、B、C、D 这 4 个对象之间已经没有了耦合关系，彼此之间毫无联系，如图 9-6 所示。

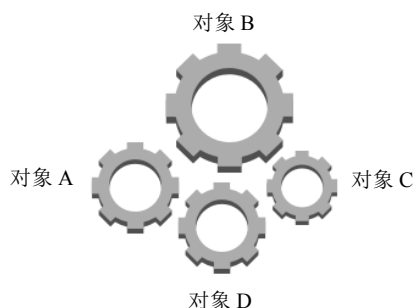


图 9-4 对象的耦合

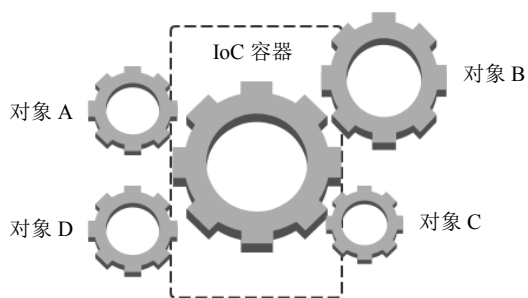


图 9-5 IoC 解耦

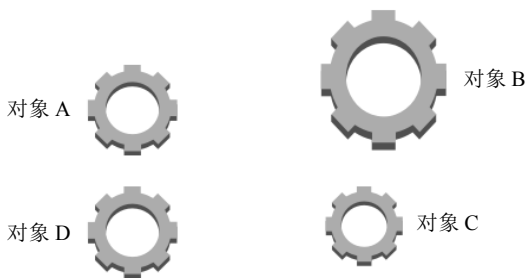


图 9-6 去掉 IoC 容器

软件系统在没有引入 IoC 容器之前，如图 9-4 所示，对象 A 依赖于对象 B，那么对象 A 在初始化或者运行到某一点的时候，自己必须主动去创建对象 B 或者使用已经创建的对象 B。无论是创建对象 B 还是使用对象 B，控制权都在自己手上。软件系统在引入 IoC 容器之后，情形就完全改变了，如图 9-5 所示。由于 IoC 容器的加入，对象 A 与对象 B 之间失去了直接联系，所以，当对象 A 运行到需要对象 B 的时候，IoC 容器会主动创建一个对象 B 注入到对象 A 需要的地方。通过引入 IoC 容器前后的对比，可以看出：对象 A 获得依赖对象 B 的过程，由主动行为变为被动行为，控制权颠倒过来了，这就是控制反转这个名称的由来。

2. 依赖注入

2004 年，Martin Fowler 探讨了一个问题，控制反转是“哪些方面的控制被反转了”呢？最终他得出了答案：“获得依赖对象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为由 IoC 容器主动注入。于是，他给控制反转取了一个更合适的名字，叫作依赖注入（Dependency Injection），简称 DI。所谓依赖注入，是指由 IoC 容器在运行期间，动态地将某种依赖关系注入到对象中。

9.2.2 依赖注入的实现方式

编写代码时我们常常会发现有一些类是依赖于其他类的，所以类 A 可能需要一个类 B 的引用或对象。为了理解得更清晰些，这里举一个汽车的例子。汽车类 Car 包含了引擎 Engine 等组件，如下所示：

```
public class Car {
    private Engine mEngine;
    public Car() {
        mEngine = new PetrolEngine();
    }
}
```

代码本身没有错，但是 Car 和 Engine 高度耦合，在 Car 中需要自己创建 Engine，并且 Car 还需要知道 Engine 的实现方法，也就是 Engine 的实现类 PetrolEngine 的存在。另外，一旦 Engine 的类型变为其他的实现比如 DieselEngine，则需要修改 Car 的构造方法。以上问题需要用依赖注

入来解决。接下来，就用依赖注入的 3 种常用方式来改造上面的代码。

1. 构造方法注入

通过 Car 的构造方法，向 Car 传递了 Engine 对象，如下所示：

```
public class Car {  
    private Engine mEngine;  
    public Car(Engine mEngine) {  
        this.mEngine=mEngine;  
    }  
}
```

2. Setter 方法注入

通过 Car 的 set 方法向 Car 传递 Engine 对象，如下所示：

```
public class Car {  
    private Engine mEngine;  
    public void set(Engine mEngine) {  
        this.mEngine=mEngine;  
    }  
}
```

3. 接口注入

在接口中定义需要注入的信息，并通过接口完成注入。接口代码如下所示：

```
public interface ICar {  
    public void setEngine(Engine engine);  
}
```

接着 Car 类实现接口 ICar：

```
public class Car implements ICar{  
    private Engine mEngine;  
    @Override  
    public void setEngine(Engine engine) {  
        this.mEngine=engine;  
    }  
}
```

通过以上 3 种注入方法，明显将 Car 和 Engine 解耦合了。Car 不关心 Engine 的实现，即使 Engine 的类型变换了，Car 也无须做任何修改。

9.3 依赖注入框架

Android 目前主流的依赖注入框架有 ButterKnife 和 Dagger2。本节我们会学习这两个框架，但在此之前我们需要明白为何要使用依赖注入框架。

9.3.1 为何使用依赖注入框架

从依赖注入的 3 种常用方式可以看出依赖注入似乎很简单，那么为何还有用依赖注入的框架呢？我们可以简单地通过一个构造方法或是使用 Setter 方法传递需要的依赖。这种做法对于简单的依赖来说是可行的，但对于复杂的依赖就未必可行。回到上面 Car 的例子，汽车的引擎由曲柄连杆机构和配气机构两大机构，以及冷却、润滑、点火、燃料供给、启动系统等五大系统组成，而曲柄连杆机构由机体组、活塞连杆组、曲轴飞轮组三部分组成。同样地，其他机构和系统下也由很多部分组成。另外，汽车不只有引擎，其还有底盘、车身、电气设备等部件，每个部件又由很多部分组成。如果用依赖注入，那么就需要为汽车的每个部分都创建类，我们可以预料到最终将会得到许多类，并且有着复杂的树状或者图状结构的依赖。为此，我们必须按照正确顺序创建对象才能创建好依赖，从叶子节点依赖开始，依次传递到每个父节点依赖，依此类推，直到传递到最高点或是根节点依赖。如果我们还使用构造方法注入或者 Setter 方法注入，为了传递依赖就要编写相当多的复杂代码，这些代码也是我们时常要避免编写的样板代码。为了避免此问题的产生，也就诞生了依赖注入框架。下面主要讲解两种 Android 常用的依赖注入框架 ButterKnife 和 Digger2。

9.3.2 解析 ButterKnife

ButterKnife 从严格意义讲不算是依赖注入框架，它只是专注于 Android 系统的 View 注入框架，并不支持其他方面的注入。它可以减少大量的 findViewById 以及 setOnClickListener 代码，简化代码并提升开发效率。首先我们来学习如何使用 ButterKnife。

1. ButterKnife 的注解使用方法

(1) 添加依赖库

首先在 Project 的 build.gradle 文件中添加如下代码：

```
buildscript {
    ...
    dependencies {
        ...
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
```

这里引用了 android-apt 插件，不了解它的读者可以查看 9.1.3 节。接下来在 Module:app 的

build.gradle 文件中添加如下代码:

```
apply plugin: 'com.neenbedankt.android-apt'
...
dependencies {
    ...
    compile 'com.jakewharton:butterknife:8.4.0'
    apt 'com.jakewharton:butterknife-compiler:8.4.0'
}
```

(2) 绑定控件

用注解@BindView 绑定控件 id, 代码如下所示:

```
public class MainActivity extends AppCompatActivity {
    @BindView(R.id.tv_text)
    TextView tv_text;//1
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);
        tv_text.setText("BindView");
    }
}
```

需要注意的是上面代码注释 1 处的 TextView 的类修饰符不能是 private 或者 static, 否则会报错。

用注解@BindViews 绑定多个控件 id, 代码如下所示:

```
public class MainActivity extends AppCompatActivity {
    @BindViews({R.id.bt_button1, R.id.bt_button2, R.id.bt_button3})
    List<Button> buttonList;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);
        buttonList.get(0).setText("button1");
        buttonList.get(1).setText("button2");
        buttonList.get(2).setText("button3");
    }
}
```

(3) 绑定资源

可以用注解 @BindString、@BindArray、@BindBool、@BindColor、@BindDimen、

`@BindDrawable` 和 `@BindBitmap`。这里我们拿 `@BindString`、`@BindArray` 和 `@BindDimen` 来进行举例。

```
@BindString(R.string.app_name)
String appName;
@BindArray(R.array.swordsman)
String[] swordsman;
@BindDimen(R.dimen.activity_horizontal_margin)
float margin;
```

(4) 绑定监听

用 `@OnClick` 来对“点击事件”进行监听，同理也可以用 `@OnLongClick` 来对“长按点击事件”进行监听，如下所示：

```
@OnClick(R.id.bt_button1)
public void showToast() {
    Toast.makeText(this, "onClick", Toast.LENGTH_SHORT).show();
}

@OnLongClick(R.id.bt_button2)
public boolean setText(Button button) {
    button.setText("长按点击事件");
    return true;
}
```

用 `@OnTextChanged` 来监听 `EidtText`，如下所示：

```
@OnTextChanged(value = R.id.et_edittext, callback = OnTextChanged.Callback.
    BEFORE_TEXT_CHANGED)
void beforeTextChanged(CharSequence s, int start, int count, int after) {

}

@OnTextChanged(value = R.id.et_edittext, callback = OnTextChanged.
    Callback.TEXT_CHANGED)
void onTextChanged(CharSequence s, int start, int before, int count) {

}

@OnTextChanged(value = R.id.et_edittext, callback = OnTextChanged.
    Callback.AFTER_TEXT_CHANGED)
void afterTextChanged(Editable s) {

}
```

使用 `@OnTouch` 处理触摸事件，如下所示：

```
@OnTouch(R.id.bt_button3)
public boolean onTouch(View view, MotionEvent event){
    return true;
}
```

使用@OnClick 对列表 item 的点击事件进行监听，如下所示：

```
@OnClick(R.id.lv_list)
void onItemClick(int position) {
    Toast.makeText(this, "onItemClick" + position, Toast.LENGTH_SHORT).
    show();
}
```

(5) 可选绑定

@BindView 或者其他的注解操作符，如果不能找到目标资源，则会引发异常。为了防止异常，可以添加@Nullable 注解：

```
@Nullable
@BindView(R.id.tv_topText)
TextView name;
```

2. 在 Fragment 和 Adapter 中使用 ButterKnife

除了前面在 Activity 中使用 ButterKnife 之外，我们还可以在 Fragment 中使用它，如下所示：

```
@BindView(R.id.tv_topText)
TextView tv_topText;
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_moon, container, false);
    ButterKnife.bind(this, view);
    return view;
}
```

同样，在 Adapter 中我们也需要引用控件资源，也可以使用 ButterKnife，如下所示：

```
public class SwordsmanAdapter extends ArrayAdapter<Swordsman>{
    ...
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        Swordsman swordsman=getItem(position);
        View view;
        ViewHolder viewHolder;
        if(convertView==null){
            view= LayoutInflater.from(getContext()).inflate(resourceId,
```

```

        parent, false);
        viewHolder = new ViewHolder(view);
        view.setTag(viewHolder);
    } else {
        view = convertView;
        viewHolder = (ViewHolder) view.getTag();
    }
    viewHolder.tv_name.setText(swordsman.getName());
    return view;
}

class ViewHolder {
    @BindView(R.id.tv_name)
    TextView tv_name;
    public ViewHolder(View view) {
        ButterKnife.bind(this, view);
    }
}
}

```

ButterKnife 在 ViewHolder 类中完成绑定操作，剩余的代码像平常一样调用即可。

3. ButterKnife 原理解析

ButterKnife 采用的是编译时注解，不了解注解的读者可先查看本章的 9.1 节。ButterKnife 自定义了很多我们常用的注解，比如 @BindView 和 @OnClick。现在先来看 @BindView 的源码，如下所示：

```

@Retention(CLASS) @Target(FIELD)
public @interface BindView {
    @IdRes int value();
}

```

Retention(CLASS) 表明 @BindView 注解是编译时注解，@Target(FIELD) 则表明 @BindView 注解应用于成员变量。接下来使用 @BindView 注解来绑定 TextView 控件，后文会用到这些代码，如下所示：

```

public class MainActivity extends AppCompatActivity {
    @BindView(R.id.tv_text)
    TextView tv_text;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);
    }
}

```


(1) ButterKnifeProcessor 源码分析

要处理注解需要注解处理器，ButterKnife 的注解处理器是 ButterKnifeProcessor，它的源码在 <https://github.com/JakeWharton/butterknife/tree/master/butterknife-compiler>。ButterKnifeProcessor 继承自 AbstractProcessor，它的主要处理逻辑都在 process 方法中，如下所示：

```
@AutoService(Processor.class)
public final class ButterKnifeProcessor extends AbstractProcessor {
    ...
    @Override public boolean process(Set<? extends TypeElement> elements,
        RoundEnvironment env) {
        Map<TypeElement, BindingSet> bindingMap = findAndParseTargets(env); //1
        for (Map.Entry<TypeElement, BindingSet> entry : bindingMap.entrySet())
        { //2
            TypeElement typeElement = entry.getKey();
            BindingSet binding = entry.getValue(); //3
            JavaFile javaFile = binding.brewJava(sdk);
            try {
                javaFile.writeTo(filer); //4
            } catch (IOException e) {
                error(typeElement, "Unable to write binding for type %s: %s",
                    typeElement, e.getMessage());
            }
        }
        return true;
    }
    ...
}
```

查看上面代码注释 1 处调用的 findAndParseTargets 方法：

```
private Map<TypeElement, BindingSet> findAndParseTargets(RoundEnvironment
env) {
    Map<TypeElement, BindingSet.Builder> builderMap = new LinkedHashMap<>();
    Set<TypeElement> erasedTargetNames = new LinkedHashSet<>();
    scanForRClasses(env);
    ...
    for (Element element : env.getElementsAnnotatedWith(BindView.class)) {
        try {
            parseBindView(element, builderMap, erasedTargetNames); //1
        } catch (Exception e) {
            logParsingError(element, BindView.class, e);
        }
    }
}
```

```
...
    return bindingMap;
}
```

`findAndParseTargets` 方法会查找所有 `ButterKnife` 的注解来进行解析，前文我们使用了 `@BindView` 注解，因此这里只截取了处理 `@BindView` 注解的部分。接着查看上面代码注释 1 处的 `parseBindView` 方法，如下所示。

```
private void parseBindView(Element element, Map<TypeElement, BindingSet.
Builder> builderMap,
    Set<TypeElement> erasedTargetNames) {
    TypeElement enclosingElement = (TypeElement) element.getEnclosingElement();
    boolean hasError = isInaccessibleViaGeneratedCode(BindView.class,
        "fields", element)
        || isBindingInWrongPackage(BindView.class, element); //1
    ...
    if (hasError) {
        return;
    }
    int id = element.getAnnotation(BindView.class).value(); //2
    BindingSet.Builder builder = builderMap.get(enclosingElement);
    if (builder != null) { //3
        String existingBindingName = builder.findExistingBindingName(getId(id));
        if (existingBindingName != null) {
            error(element, "Attempt to use @%s for an already bound ID %d on '%s'.
                (%s.%s)",
                BindView.class.getSimpleName(), id, existingBindingName,
                enclosingElement.getQualifiedName(), element.getSimpleName());
            return;
        }
    } else {
        builder = getOrCreateBindingBuilder(builderMap, enclosingElement);
    }
    String name = element.getSimpleName().toString();
    TypeName type = TypeName.get(elementType);
    boolean required = isFieldRequired(element);
    builder.addField(getId(id), new FieldViewBinding
        (name, type, required)); //4
    erasedTargetNames.add(enclosingElement);
}
```

上面代码注释 1 处的 `isInaccessibleViaGeneratedCode` 方法里面检查了 3 个点，它们分别是：

- 方法修饰符不能为 `private` 和 `static`；
- 包含类型不能为非 `Class`；

- 包含类的修饰符不能是 `private`。

`isBindingInWrongPackage` 方法判断了这个类的包名不能以 `android.` 和 `java.` 开头。上面代码注释 2 处获取注解的标注的值。接下来注释 3 处判断是否存在 `BindingSet.Builder` 的值, 若没有则创建, 有则复用。在注释 4 处可以看出, 将注解修饰的类型的信息存储在 `FieldViewBinding` 中, 并将 `FieldViewBinding` 传入 `BindingSet.Builder` 的 `addField` 方法中。这样注解修饰的类型的信息以及注解的成员变量的值都存储在 `BindingSet` 中。

接下来我们回到 `process` 方法, 我们从上面也知道了第 400 页注释 1 处的 `findAndParseTargets` 方法主要用于查找和解析注解。在注释 2 处遍历 `findAndParseTargets` 方法返回的 `Map` 集合, 在注释 3 处得到 `BindingSet` 的值, 并调用了它的 `brewJava` 方法, 如下所示:

```
JavaFile brewJava(int sdk) {
    return JavaFile.builder(bindingClassName.packageName(), createType(sdk))
        .addFileComment("Generated code from Butter Knife. Do not modify!")
        .build();
}
```

`brewJava` 方法将使用注解的类生成一个 `JavaFile`, 在 `process` 方法的注释 4 处(在第 400 页), 将该 `JavaFile` 输出成 `Java` 文件。在 `build-generated-source-apt` 目录下可以找到生成的 `Java` 文件, 这里生成的文件名为 `MainActivity_ViewBinding`。先不分析这个文件做了什么, 后面会讲到, 现在先来分析一下 `ButterKnife` 的 `bind` 方法。

(2) ButterKnife 的 bind 方法

为了使用 `ButterKnife`, 我们需要用 `ButterKnife.bind()` 方法来绑定上下文。现在先来查看 `bind` 方法做了什么:

```
@NonNull @UiThread
public static Unbinder bind(@NonNull Activity target) {
    View sourceView = target.getWindow().getDecorView();
    return createBinding(target, sourceView);
}
```

`bind` 方法有很多重载方法, 上面的代码只是其中的一种, 也就是传入 `Activity` 的情况。得到 `Activity` 的 `DecorView`, 并将 `DecorView` 和 `Activity` 传入 `createBinding` 方法中, 这个 `DecorView` 后文会提到。`createBinding` 方法如下所示:

```
private static Unbinder createBinding(@NonNull Object target, @NonNull
View source) {
    Class<?> targetClass = target.getClass();
    if (debug) Log.d(TAG, "Looking up binding for " + targetClass.getName());
    Constructor<? extends Unbinder> constructor =
        findBindingConstructorForClass(targetClass); //1
    if (constructor == null) {
```

```

        return Unbinder.EMPTY;
    }
    try {
        return constructor.newInstance(target, source);//2
    }
    ...
}

```

上面代码注释 1 处调用了 `findBindingConstructorForClass` 方法，如下所示：

```

@Nullable @CheckResult @UiThread
private static Constructor<? extends Unbinder> findBindingConstructorForClass
(Class<?> cls) {
    Constructor<? extends Unbinder> bindingCtor = BINDINGS.get(cls);//1
    if (bindingCtor != null) {
        if (debug) Log.d(TAG, "HIT: Cached in binding map.");
        return bindingCtor;
    }
    String clsName = cls.getName();
    if (clsName.startsWith("android.") || clsName.startsWith("java.")) {
        if (debug) Log.d(TAG, "MISS: Reached framework class. Abandoning
            search.");
        return null;
    }
    try {
        Class<?> bindingClass = Class.forName(clsName + "_ViewBinding");//2
        bindingCtor = (Constructor<? extends Unbinder>) bindingClass.getConstructor
            (cls, View.class);//3
        if (debug) Log.d(TAG, "HIT: Loaded binding class and constructor.");
    } catch (ClassNotFoundException e) {
        if (debug) Log.d(TAG, "Not found. Trying superclass " + cls.
            getSuperclass().getName());
        bindingCtor = findBindingConstructorForClass(cls.getSuperclass());
    } catch (NoSuchMethodException e) {
        throw new RuntimeException("Unable to find binding constructor for "
            + clsName, e);
    }
    BINDINGS.put(cls, bindingCtor);//4
    return bindingCtor;
}

```

在上面代码注释 1 处会先从 `BINDINGS` 中获取对应 `Class` 的 `Constructor` 实例，`BINDINGS` 是一个 `Class` 为 `key`、`Constructor` 为 `value` 的 `Map`。如果没有获取 `Constructor`，则在注释 2 处会通过反射来生成 `Class` 类，这个 `Class` 类就是我们此前生成的 `MainActivity_ViewBinding`。虽然

反射会影响一些性能，但是因为有 BINDERS 的存在（一个类只会在第一次反射生成，以后会从 BINDERS 中去取），也可以解决一些性能问题。在注释 3 处通过调用 `getConstructor` 方法将 `Class` 转换为 `Constructor`，`getConstructor` 方法中并没有做什么主要的操作。在注释 4 处将 `Constructor` 作为 value，`Class` 作为 key 存储在 BINDINGS 中，最后返回该 `Constructor`。接着在 `createBinding` 方法（在第 402 页）的注释 2 处，生成该 `Constructor` 的实例，也就是 `MainActivity_ViewBinding` 的实例。我们来查看在 `MainActivity_ViewBinding` 中做了什么。

（3）生成的辅助类分析

`MainActivity_ViewBinding.java`，如下所示：

```
public class MainActivity_ViewBinding<T extends MainActivity> implements
Unbinder {
    protected T target;
    @UiThread
    public MainActivity_ViewBinding(T target, View source) {
        this.target = target;
        target.tv_text = Utils.findRequiredViewAsType(source,
            R.id.tv_text, "field 'tv_text'", TextView.class); //1
    }
    @Override
    @CallSuper
    public void unbind() {
        T target = this.target;
        if (target == null) throw new IllegalStateException("Bindings already
            cleared.");
        target.tv_text = null;
        this.target = null;
    }
}
```

在前面我们已经知道在 `createBinding` 方法中调用 `newInstance` 方法生成 `MainActivity_ViewBinding` 实例时传入的 `source` 值是 `MainActivity` 的 `DecorView`。而 `target` 值为 `MainActivity`。接着我们来查看 `MainActivity_ViewBinding` 的构造方法。很明显，构造方法的 `source` 值就是 `MainActivity` 的 `DecorView`，而 `target` 值为 `MainActivity`。在上面代码注释 1 处调用了 `Utils` 的 `findRequiredViewAsType` 方法并将 `source` 值、`R.id.tv_text` 等参数传入，`findRequiredViewAsType` 方法如下所示：

```
public static <T> T findRequiredViewAsType(View source, @IdRes int id,
String who, Class<T> cls) {
    View view = findRequiredView(source, id, who); //1
    return castView(view, id, who, cls); //2
}
```

在上面代码注释 1 处调用 `findRequiredView` 方法：

```
public static View findRequiredView(View source, @IdRes int id, String who) {
    View view = source.findViewById(id);
    if (view != null) {
        return view;
    }
    ...
}
```

`findRequiredView` 方法调用了 `DecorView` 的 `findViewById` 方法并将 `R.id.tv_text` 对应的 `View` 返回。接着 `findRequiredViewAsType` 方法的注释 2 处会调用 `castView` 方法：

```
public static <T> T castView(View view, @IdRes int id, String who, Class<T>
cls) {
    try {
        return cls.cast(view);
    } catch (ClassCastException e) {
        ...
    }
}
```

`castView` 方法会将 `View` 强制转换成传入的 `Class` 值的类型，这里的 `Class` 值从 `MainActivity_ViewBinding` 类（在第 404 页）的注释 1 处可以得知是 `TextView.class`，因此 `castView` 方法会将 `View` 强制转换为 `TextView` 并返回。再次回到 `MainActivity_ViewBinding` 辅助类，这个返回的 `TextView` 会赋值给 `target`，也就是 `MainActivity`，这样我们在 `MainActivity` 中就可以使用这个 `TextView` 了。

9.3.3 解析 Dagger2

Dagger2 是一个基于 JSR-330（Java 依赖注入）标准的依赖注入框架，在编译期间自动生成代码，负责依赖对象的创建。Dagger2 是 Dagger1（由 Square 公司受到 Guice 启发创建的）的分支，由谷歌接手开发，其受到了 AutoValue 项目的启发。我们首先来学习 Dagger2 的使用方法。

1. 注解使用方法

（1）添加依赖库

首先在 Project 的 `build.gradle` 文件中添加引用 `android-apt` 插件，如下所示：

```
buildscript {
    ...
    dependencies {
        ...
    }
}
```

```

        classpath 'com.neenbedankt.gradle.plugins.android-apt:1.8'
    }
}

```

接下来在 Module:app 的 build.gradle 文件中添加如下代码:

```

apply plugin: 'com.neenbedankt.android-apt'
...
dependencies {
    ...
    apt 'com.google.dagger:dagger-compiler:2.7'
    compile 'com.google.dagger:dagger:2.7'
}

```

(2) @Inject 和 @Component

在使用这两个注解前, 如果我们需要在 MainActivity 中调用一个类的方法, 则可能会按照如下代码这样:

```

public class Watch{
    public void work(){
        Log.i("wangshu","手表工作");
    }
}

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Watch watch =new Watch();
        watch.work();
    }
}

```

上面的代码如果使用 @Inject 和 @Component 注解, 则会按照如下步骤改写。首先改写 Watch 类, 如下所示:

```

public class Watch {
    @Inject
    public Watch(){
    }
    public void work(){
        Log.d("Dagger2","手表工作");
    }
}

```

`@Inject` 注解是 JSR-330 标准中的一部分，用于标记需要注入的依赖。在这里标记 `Watch` 构造方法，则表明 `Dagger2` 可以使用 `Watch` 构造方法构建对象。接下来用 `@Component` 注解来完成依赖注入。我们需要定义一个接口，接口命名建议为：目标类名+`Component`，在编译后 `Dagger2` 就会为我们生成名为 `Dagger+目标类名+Component` 的辅助类。具体代码如下所示：

```
@Component
public interface MainActivityComponent {
    void inject(MainActivity activity);
}
```

`Component` 则可以理解为注入器，它会把目标类依赖的实例注入到目标类中。在这里需要定义 `inject` 方法，传入需要注入依赖的目标类。在这个例子中需要注入依赖的目标类是 `MainActivity`。最后在 `MainActivity` 中调用 `Watch` 的 `work` 方法，代码如下所示：

```
public class MainActivity extends AppCompatActivity {
    @Inject//1
    Watch watch;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        DaggerMainActivityComponent.create().inject(this);//2
        watch.work();
    }
}
```

上面代码注释 1 处用 `@Inject` 标记需要注入的属性，注释 2 处调用编译生成的 `DaggerMainActivityComponent` 的 `inject` 方法来完成注入，注入的目标为 `MainActivity`。在这里我们提到 `@Inject` 有了两种注入方式，分别是成员变量注入和构造方法注入。还有一种注入方式叫作方法注入，当我们需要传类实例来注入到依赖时，会使用到它，方法注入会在构造方法调用后立即调用。在这里，我们先知道有这么一个概念，在第 10 章我们将用到方法注入。

（3）`@Module` 和 `@Provides`

如果项目中使用了第三方的类库，比如 `Gson`，若我们像如下代码这样写则会报错，因为我们不能将 `@Inject` 应用到 `Gson` 的构造方法中。

```
@Inject
Gson gson;
```

这个时候可以采用 `@Module` 和 `@Provides` 来处理。首先我们创建 `GsonModule` 类，如下所示：

```
@Module
public class GsonModule {
```



```

    @Provides
    public Gson provideGson() {
        return new Gson();
    }
}

```

将 `@Module` 标注在类上，用来告诉 `Component`，可以从这个类中获取依赖对象，也就是 `Gson` 类；`@Provides` 标记在方法上，表示可以通过这个方法来获取依赖对象的实例。通俗来讲，`@Module` 标注的类其实就是一个工厂，用来生成各种类；`@Provides` 标记的方法，就是用来生成这些类的实例的。接着来编写 `Component` 类，如下所示：

```

@Component(modules = GsonModule.class)
public interface MainActivityComponent {
    void inject(MainActivity activity);
}

```

这和此前的区别就是加上了 `modules = GsonModule.class`，用来指定 `Module`。需要注意的是，`Component` 中可以指定多个 `Module`。接下来在 `MainActivity` 中使用 `Gson`，如下所示：

```

public class MainActivity extends AppCompatActivity {
    @Inject
    Gson gson;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        DaggerMainActivityComponent.create().inject(this);
        String jsonData = "{\"name':'zhangwuji', 'age':20}";
        Man man=gson.fromJson(jsonData, Man.class);
        Log.d(TAG, "name---"+man.getName());
    }
}

```

还有一种情况，当我们使用依赖注入的时候，如果需要注入的对象是抽象的，则 `@Inject` 也无法使用，因为抽象的类并不能实例化，如下所示：

```

public abstract class Engine {
    public abstract String work();
}

```

上面定义一个 `Engine` 抽象类，接着定义它的实现类 `GasolineEngine`：

```

public class GasolineEngine extends Engine{
    @Inject
    public GasolineEngine(){

```

```

    }
    public String work() {
        return "汽油发动机发动";
    }
}

```

随后在 Car 类中引用 Engine ， 如下所示：

```

public class Car {
    private Engine engine;
    @Inject
    public Car(Engine engine) {
        this.engine = engine;
    }
    public String run() {
        return engine.work();
    }
}

```

这时编译程序，Dagger2 会报错，因为 Car 需要 Engine 对象，而 Engine 对象是抽象的，@Inject 无法提供实例。这时也可以采用@Module 和@Provides。首先修改 GasolineEngine 类，去掉@Inject：

```

public class GasolineEngine extends Engine{
    public String work() {
        return "汽油发动机发动";
    }
}

```

创建 EngineModule 类，如下所示：

```

@Module
public class EngineModule {
    @Provides
    public Engine provideEngine() {
        return new GasolineEngine();
    }
}

```

接着在 Component 中指定 EngineModule：

```

@Component(modules =EngineModule.class)
public interface MainActivityComponent {
    void inject(MainActivity activity);
}

```

最后在 MainActivity 中使用，如下所示：

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG="Dagger2";
    @Inject
    Car car;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        DaggerMainActivityComponent.create().inject(this);
        String str=car.run();
        Log.d(TAG, "car---"+str);
    }
}
```

（4）@Named 和 @Qualifier

@Qualifier 是限定符，@Named 则是 @Qualifier 的一种实现。当有两个相同的依赖时，它们都继承同一个父类或者均实现同一个接口。当它们被提供给高层时，Component 就不知道我们到底要提供哪一个依赖对象了，因为它找到了两个。比如在上面的汽车例子中，如果我们再提供一个 DieselEngine 给它，EngineModule 可以改写为如下代码所示：

```
@Module
public class EngineModule {
    @Provides
    public Engine provideGasoline() {
        return new GasolineEngine();
    }
    @Provides
    public Engine provideDiesel() {
        return new DieselEngine();
    }
}
```

编译时 Dagger2 会报错，因为我们提供了多个 Provides，Component 不知道要选哪个。这个时候我们就可以使用 @Named，代码如下所示：

```
@Module
public class EngineModule {
    @Provides
    @Named("Gasoline")
    public Engine provideGasoline() {
        return new GasolineEngine();
    }
}
```

```

@Provides
@Named("Diesel")
public Engine provideDiesel() {
    return new DieselEngine();
}
}

```

给不同的 `Provides` 定义不同的 `@Named`，接下来在 `Car` 类中指定要采用哪种 `Provides`：

```

public class Car {
    private Engine engine;
    @Inject
    public Car(@Named("Diesel") Engine engine) {
        this.engine = engine;
    }
    public String run() {
        return engine.work();
    }
}

```

上面的代码通过 `@Named` 来指定采用 `provideDiesel` 方法来生成实例。上面的例子也可以用 `@Qualifier` 来实现，`@Named` 传递的值只能是字符串；而 `@Qualifier` 则更灵活一些，`@Qualifier` 不是直接标记在属性上的，而是用来自定义注解的，如下所示：

```

@Qualifier
@Retention(RUNTIME)
public @interface Gasoline {
}

@Qualifier
@Retention(RUNTIME)
public @interface Diesel {
}

```

分别自定义两个注解 `@Gasoline` 和 `@Diesel`，接下来修改 `EngineModule` 类，如下所示：

```

@Module
public class EngineModule {
    @Provides
    @Gasoline
    public Engine provideGasoline() {
        return new GasolineEngine();
    }
    @Provides
    @Diesel

```

```

        public Engine provideDiesel() {
            return new DieselEngine();
        }
    }
}

```

最后在 Car 类指定需要哪种依赖：

```

public class Car {
    private Engine engine;
    @Inject
    public Car(@Gasoline Engine engine) {
        this.engine = engine;
    }
    public String run() {
        return engine.work();
    }
}

```

(5) @Singleton 和 @Scope

@Scope 是用来自定义注解的，而 @Singleton 则是用来配合实现局部单例和全局单例的。需要注意的是，@Singleton 本身不具备创建单例的能力。如果我们要两次使用 Gson，会这么做：

```

@Inject
Gson gson;
@Inject
Gson gson1;

```

gson 和 gson1 的内存地址不同，也就是新创建了两个 Gson。如果我们想让 Gson 在 MainActivity 中是单例，可以使用 @Singleton。首先在 GsonModule 中添加 @Singleton，如下所示：

```

@Module
public class GsonModule {
    @Singleton
    @Provides
    public Gson provideGson() {
        return new Gson();
    }
}

```

接下来在 MainActivityComponent 中添加 @Singleton：

```

@Singleton
@Component(modules = GsonModule.class)
public interface MainActivityComponent {

```

```
void inject(MainActivity activity);
}
```

我们在 MainActivity 中打印 Gson 的 hashCode 值，就会发现值是相同的，如下所示：

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG="Dagger2";
    @Inject
    Gson gson;
    @Inject
    Gson gson1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        DaggerMainActivityComponent.create().inject(this);
        Log.d(TAG, gson.hashCode() + "-----" + gson1.hashCode());
    }
}
```

Gson 在 MainActivity 中是单例，如果再创建一个 SecondActivity，SecondActivity 创建的 Gson 的内存地址和 MainActivity 创建的 Gson 的内存地址是不同的。因为 Gson 只是保证在 MainActivityComponent 中是单例的，我们创建 SecondActivity，就会重新创建一个 Component，这样只能保证 Gson 是局部单例（MainActivity 中）。如果想要实现全局单例，就需要保证对应的 Component 只有一个实例。查看@Singleton 的源码：

```
@Scope
@Documented
@Retention(RUNTIME)
public @interface Singleton {}
```

其实就是用@Scope 标识的注解。为了使 Gson 变为全局单例，我们可以用@Scope 结合 Application 来实现。当然也可以用@Singleton 结合 Application 来实现，只是用@Scope 可以自定义注解名称，这更灵活一些。先来定义@ApplicationScope 注解，如下所示：

```
@Scope
@Retention(RUNTIME)
public @interface ApplicationScope {
}
```

接下来在 GsonModule 中使用@ApplicationScope：

```
@Module
public class GsonModule {
    @ApplicationScope
```

```
@Provides
public Gson provideGson() {
    return new Gson();
}
}
```

为了处理多个 Activity，我们创建 ActivityComponent，并使用@ApplicationScope，如下所示：

```
@ApplicationScope
@Component(modules = GsonModule.class)
public interface ActivityComponent {
    void inject(MainActivity activity);
    void inject(SecondActivity activity);
}
```

创建 App 类继承自 Application，用来提供 ActivityComponent 实例，如下所示：

```
public class App extends Application{
    ActivityComponent activityComponent;
    @Override
    public void onCreate() {
        super.onCreate();
        activityComponent = DaggerActivityComponent.builder().build();
    }
    public static App get(Context context) {
        return (App) context.getApplicationContext();
    }
    ActivityComponent getActivityComponent(){
        return activityComponent;
    }
}
```

最后在 MainActivity 中实现如下代码：

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG="Dagger2";
    private Button bt_jump;
    @Inject
    Gson gson;
    @Inject
    Gson gson1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

```

        setContentView(R.layout.activity_main);
        App.get(MainActivity.this).getActivityComponent().inject(this);
        onClick();
        Log.d(TAG, gson.hashCode() + "-----" + gson1.hashCode());
    }
    private void onClick(){
        bt_jump= (Button) findViewById(R.id.bt_jump);
        bt_jump.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Intent intent=new Intent(MainActivity.this,SecondActivity.class);
                startActivity(intent);
            }
        });
    }
}

```

SecondActivity 中的代码也是类似的,运行程序,发现 MainActivity 和 SecondActivity 的 Gson 的内存地址是一样的。这样也就实现了 Gson 的全局单例。当然这只是简单用到了 @Scope, @Scope 的作用远远不止如此。我们要做一个应用肯定要用到很多 Component, 一般划分的规则就是有一个全局的 Component, 比如 AppComponent。每个界面有一个 Component, 比如一个 Activity 定义一个 Component, 一个 Fragment 定义一个 Component。当然, 这不是必需的, 如果页面之间依赖的类是一样的, 可以共用一个 Component。这样算来一个应用会有很多 Component, 为了管理这些 Component, 就可以使用自定义 Scope 注解。它可以更好地管理 Component 与 Module 之间的匹配关系。比如编译器会检查 Component 管理的 Module, 若发现管理的 Module 中的标注创建类实例方法的 Scope 注解与 Component 类中的 Scope 注解不一样, 就会报错。Scope 注解还可以更好地管理 Component 之间的组织方式, 不同的组织方式定义为不同的 Scope 注解名称, 方便管理。

(6) @Component 的 dependencies

@Component 也可以用 dependencies 依赖于其他 Component。我们重新创建一个 Swordsman 类:

```

public class Swordsman {
    @Inject
    public Swordsman() {
    }
    public String fighting() {
        return "欲为大树, 莫与草争";
    }
}

```


接下来，按照惯例创建 `SwordsmanModule` 和 `SwordsmanComponent`，如下所示：

```
@Module
public class SwordsmanModule {
    @Provides
    public Swordsman provideSwordsman() {
        return new Swordsman();
    }
}

@Component(modules = SwordsmanModule.class)
public interface SwordsmanComponent {
    Swordsman getSwordsman();
}
```

在 `ActivityComponent` 中通过 `@Component` 的 `dependencies` 来引入 `SwordsmanComponent`。

```
@ApplicationScope
@Component(modules = GsonModule.class, dependencies = SwordsmanComponent.class)
public interface ActivityComponent {
    void inject(MainActivity activity);
    void inject(SecondActivity activity);
}
```

随后在此前定义的 `App` 中引入 `SwordsmanComponent`，如下所示：

```
public class App extends Application{
    ActivityComponent activityComponent;
    @Override
    public void onCreate() {
        super.onCreate();
        activityComponent = DaggerActivityComponent.builder().
            swordsmanComponent(DaggerSwordsmanComponent.builder()
                .build()).build();
    }
    public static App get(Context context) {
        return (App) context.getApplicationContext();
    }
    ActivityComponent getActivityComponent(){
        return activityComponent;
    }
}
```

最后我们在 `SecondActivity` 中使用 `Swordsman`，如下所示：

```
public class SecondActivity extends AppCompatActivity {
```

```

@Inject
Swordsman swordsman;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_second);
    App.get(SecondActivity.this).getActivityComponent().inject(this);
    String sd=swordsman.fighting();
    Log.d(TAG, "swordsman---" + sd);
}
}

```

2. 懒加载

Dagger2 提供了懒加载模式，在@Inject 的时候不初始化，而是使用的时候，调用 get 方法来获取实例。接着我们改写上面提到的 Swordsman，将它改写为懒加载模式。

```

public class SecondActivity extends AppCompatActivity {
    private static final String TAG="Dagger2";
    @Inject
    Lazy<Swordsman> swordsmanLazy;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        App.get(SecondActivity.this).getActivityComponent().inject(this);
        Swordsman swordsman=swordsmanLazy.get();
        swordsman.fighting();
        String sd1=swordsman.fighting();
        Log.d(TAG, "lazy---" + sd1);
    }
}

```

3. Dagger2 原理解析

因为 Dagger2 的使用情况很多，这里只对基本的使用方法进行分析。我们先写一个简单的例子，创建 Watch、WatchModule 和 ActivityComponent，代码如下所示：

```

public class Watch {
    public String work(){
        return "手表工作";
    }
}

@Module
public class WatchModule {

```

```

@Provides
public Watch provideWatch() {
    return new Watch();
}

}

@Component(modules = WatchModule.class)
public interface ActivityComponent {
    void inject(MainActivity activity);
}

```

在 MainActivity 中使用 Watch，如下所示：

```

public class MainActivity extends AppCompatActivity {
    @Inject
    Watch watch;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        DaggerActivityComponent.create().inject(this);
        String sr=watch.work();
        Log.i("wangshu",sr);
    }
}

```

这时编译程序会在 build 目录中生成辅助类，如图 9-7 所示。它们分别是 DaggerActivityComponent、WatchModule_ProvideWatchFactory 和 MainActivity_MembersInjector。

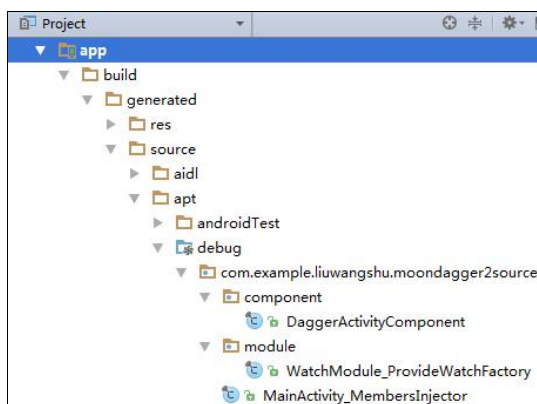


图 9-7 编译生成的辅助类

我们从注入的入口开始分析，也就是如下代码：

```
DaggerActivityComponent.create().inject(this);
```

首先从 DaggerActivityComponent 的代码开始分析，如下所示：

```
public final class DaggerActivityComponent implements ActivityComponent {
    private Provider<Watch> provideWatchProvider;
    private MembersInjector<MainActivity> mainActivityMembersInjector;
    private DaggerActivityComponent(Builder builder) {
        assert builder != null;
        initialize(builder); //3
    }
    public static Builder builder() {
        return new Builder();
    }
    public static ActivityComponent create() {
        return builder().build(); //1
    }
    @SuppressWarnings("unchecked")
    private void initialize(final Builder builder) {
        this.provideWatchProvider = WatchModule_ProvideWatchFactory.create
            (builder.watchModule); //6
        this.mainActivityMembersInjector = MainActivity_MembersInjector.create
            (provideWatchProvider); //4
    }

    @Override
    public void inject(MainActivity activity) {
        mainActivityMembersInjector.injectMembers(activity); //5
    }

    public static final class Builder {
        private WatchModule watchModule;
        private Builder() {}
        public ActivityComponent build() {
            if (watchModule == null) {
                this.watchModule = new WatchModule(); //2
            }
            return new DaggerActivityComponent(this);
        }
        public Builder watchModule(WatchModule watchModule) {
            this.watchModule = Preconditions.checkNotNull(watchModule);
            return this;
        }
    }
}
```

从上面代码注释 1 处可以看出 `create` 方法调用了 `builder().build()`，调用 `builder` 的 `build` 方法会在注释 2 处新建 `WatchModule`。注释 3 处调用 `initialize` 方法来初始化，分别初始化了 `provideWatchProvider` 和 `mainActivityMembersInjector` 这两个成员变量。可以看到注释 4 处将 `provideWatchProvider` 作为参数传入 `MainActivity_MembersInjector` 的 `create` 方法中，这个方法会返回一个 `MainActivity_MembersInjector` 类。当我们调用 `inject` 方法时，其实是调用注释 5 处的代码，也就是调用 `MainActivity_MembersInjector` 的 `injectMembers` 方法。`MainActivity_MembersInjector` 的代码如下所示：

```
public final class MainActivity_MembersInjector implements MembersInjector<MainActivity> {
    private final Provider<Watch> watchProvider;
    public MainActivity_MembersInjector(Provider<Watch> watchProvider) {
        assert watchProvider != null;
        this.watchProvider = watchProvider;
    }
    public static MembersInjector<MainActivity> create(Provider<Watch> watchProvider) {
        return new MainActivity_MembersInjector(watchProvider);
    }
    @Override
    public void injectMembers(MainActivity instance) {
        if (instance == null) {
            throw new NullPointerException("Cannot inject members into a null reference");
        }
        instance.watch = watchProvider.get(); //1
    }
    public static void injectWatch(MainActivity instance, Provider<Watch> watchProvider) {
        instance.watch = watchProvider.get();
    }
}
```

`MainActivity_MembersInjector` 的 `injectMembers` 方法会调用上面代码注释 1 处的代码，调用 `watchProvider` 的 `get` 方法并赋值给 `MainActivity` 的 `watch`。这里 `watchProvider` 是调用 `MainActivity_MembersInjector` 的 `create` 方法传进来的，也就是上文提到的 `provideWatchProvider`，它是 `DaggerActivityComponent` 类（在第 419 页）的注释 6 处调用 `WatchModule_ProvideWatchFactory` 的 `create` 方法生成的，如下所示：

```
public final class WatchModule_ProvideWatchFactory implements Factory<Watch> {
    private final WatchModule module;
    public WatchModule_ProvideWatchFactory(WatchModule module) {
        assert module != null;
```

```

        this.module = module;
    }
    @Override
    public Watch get() {
        return Preconditions.checkNotNull(
            module.provideWatch(), "Cannot return null from a non-@Nullable
            @Provides method");//1
    }
    public static Factory<Watch> create(WatchModule module) {
        return new WatchModule_ProvideWatchFactory(module);
    }
}

```

从 create 方法可以得知，watchProvider 实际上就是 WatchModule_ProvideWatchFactory，查看 WatchModule_ProvideWatchFactory 的 get 方法，它会返回注释 1 处的代码。最后查看 WatchModule 的 provideWatch 方法里做了什么：

```

@Module
public class WatchModule {
    @Provides
    public Watch provideWatch() {
        return new Watch();
    }
}

```

很显然这个 provideWatch 是我们此前定义的，它会返回一个 Watch 对象。讲了这么多，其实这 3 个辅助类的作用就是在我们调用 inject 方法时，将新创建的 Watch 类赋值给 MainActivity 的成员变量 Watch。其中 WatchModule_ProvideWatchFactory 用来生成 Watch 实例；Dagger2Activity_MemberInject 将 Watch 实例赋值给 MainActivity 的成员变量 Watch；DaggerActivityComponent 则作为程序入口和桥梁，负责初始化 WatchModule_ProvideWatchFactory 和 Dagger2Activity_MemberInject，并将它们串联起来。

9.4 本章小结

本章最初的设计就是要讲解依赖注入框架 ButterKnife 和 Dagger2 的使用方法和原理，但是笔者发现要讲解这些内容，还需要介绍注解和依赖注入的知识，因此本章也加上了这两点内容。这样通过本章的阅读，读者就可以很通顺地理解依赖注入框架的使用和原理了。关于 Dagger2，它的学习成本以及维护成本稍微有点高，读者使用前要慎重考虑。另外，ButterKnife 也可以用谷歌提供的 Data Binding 框架来替代。Data Binding 框架的相关内容会在第 10 章进行介绍。

第 10 章

应用架构设计

随着 Android 这一移动开发技术的成熟，Android 应用架构设计得到了越来越多企业以及开发者的重视，并因此衍生出了 Android 架构师这一职位。好的架构设计会带来很多好处，比如更易维护、拓展，等等；而差的架构设计或者没有架构设计，则会使应用在后期的维护和拓展时产生很多严重问题。目前 Android 的框架模式主要有 MVC、MVP 和 MVVM，虽说最近比较流行 MVP 和 MVVM，但是 MVC 也没有过时之说，我们主要还是要根据业务来选择合适的架构。本章我们就来学习这 3 种框架模式及其应用的架构设计。

10.1 MVC 模式

MVC（Model-View-Controller，模型-视图-控制器）模式是 20 世纪 80 年代 Smalltalk-80 出现的一种软件设计模式，其后来得到了广泛的应用。它用一种业务逻辑、数据、界面显示分离的方法组织代码，在改进和个性化定制界面及用户交互的同时，无须重新编写业务逻辑。Android 中 MVC 的角色定义如下。

- 模型层（Model）：我们针对业务模型，建立的数据结构和相关的类，就可以理解为 Model。Model 是与 View 无关，而与业务相关的。
- 视图层（View）：一般采用 XML 文件或者 Java 代码进行界面的描述，也可以使用 JavaScript+HTML 等方式作为 View 层。
- 控制层（Controller）：Android 的控制层通常在 Activity、Fragment 或者由它们控制的其他业务类中。

MVC 简单来说就是通过 Controller 来操作 Model 层的数据，并且返回给 View 层展示，如图 10-1 所示。

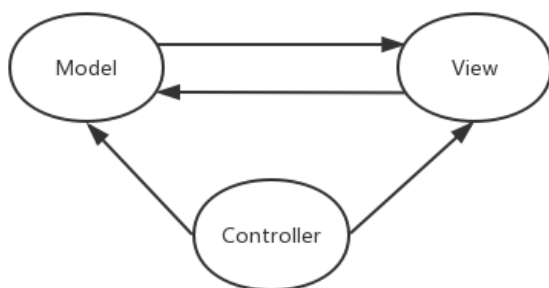


图 10-1 MVC 框架模式

Android 中的 MVC 也有它的缺点：

- Activity 并不是一个标准的 MVC 模式中的 Controller，它的首要职责是加载应用的布局 and 初始化用户界面，接受并处理来自用户的操作请求，进而做出响应。随着界面及其逻辑的复杂度不断提升，Activity 类的职责不断增加，以致变得庞大臃肿。
- View 层和 Model 层相互耦合，不易开发和维护。

为了解决以上问题，产生了 MVP 以及 MVVP 这两种框架。关于 MVC，这里就不用代码来举例了，想必读者应该已经非常熟悉了。接下来我们学习 MVP 模式。

10.2 MVP 模式

MVP（Model-View-Presenter）是 MVC 的演化版本，MVP 的角色定义如下。

- Model：主要提供数据的存取功能。Presenter 需要通过 Model 层来存储、获取数据。
- View：负责处理用户事件和视图部分的展示。在 Android 中，它可能是 Activity、Fragment 类或者是某个 View 控件。
- Presenter：作为 View 和 Model 之间沟通的桥梁，它从 Model 层检索数据后返回给 View 层，使得 View 和 Model 之间没有耦合。

在 MVP 里（见图 10-2），Presenter 完全将 Model 和 View 进行了分离，主要的程序逻辑在 Presenter 里实现。而且，Presenter 与具体的 View 是没有直接关联的，而是通过定义好的接口进行交互，从而使得在变更 View 时可以保持 Presenter 的不变，这点符合面向接口编程的特点。View 只应该有简单的 Set/Get 方法，以及用户输入和设置界面显示的内容，除此之外就不应该有更多的内容。绝不允许 View 直接访问 Model，这就是其与 MVC 的很大不同之处。



图 10-2 MVP 模式

10.2.1 应用 MVP 模式

这里举一个小的例子来实现简单的 MVP 模式。这个例子用来访问淘宝 IP 库。访问一个 IP 地址，并在界面上显示该 IP 所对应的国家、地区和城市，不了解淘宝 IP 库的读者请查看 5.4 节。在这个例子中要访问网络，为了实现方便，这里采用了 OkHttp 的封装库 OkHttpFinal。首先我们在 build.gradle 中配置 OkHttpFinal：

```
dependencies {
    ...
    compile 'cn.finalteam:okhttpfinal:2.0.7'
}
```

为了使用 OkHttpFinal，我们需要在自定义 Application 中实现如下代码：

```
public class MvpApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        OkHttpFinalConfiguration.Builder builder = new OkHttpFinalConfiguration.
            Builder();
        OkHttpFinal.getInstance().init(builder.build());
    }
}
```

1. 实现 Model

首先我们要创建 Model 实体 IpInfo：

```
public class IpInfo {
    private int code;
    private IpData data;
    public int getCode() {
        return code;
    }
    ...
}
```

IpData 的部分代码如下所示：

```
public class IpData {
    private String country;
    private String city;
    ...
    public String getCountry() {
        return country;
    }
}
```

```
...
}
```

随后定义获取网络数据的接口类:

```
public interface NetTask<T> {
    void execute(T data , LoadTasksCallBack callBack);
}
```

这里有一个回调监听接口 `LoadTasksCallBack` , 它定义了网络访问回调的各种状态:

```
public interface LoadTasksCallBack<T> {
    void onSuccess(T t);
    void onStart();
    void onFailed();
    void onFinish();
}
```

接下来我们编写 `NetTask` 的实现类以获取数据, 如下所示:

```
public class IpInfoTask implements NetTask<String> {
    private static IpInfoTask INSTANCE = null;
    private static final String HOST = "http://ip.taobao.com/service/
    getIpInfo.php";
    private LoadTasksCallBack loadTasksCallBack;
    private IpInfoTask() {

    }

    public static IpInfoTask getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new IpInfoTask();
        }
        return INSTANCE;
    }

    @Override
    public void execute(String ip, final LoadTasksCallBack
    loadTasksCallBack) {
        RequestParams requestParams = new RequestParams();
        requestParams.addFormDataPart("ip", ip);
        HttpRequest.post(HOST, requestParams, new BaseHttpRequestCallback
        <IpInfo>() {
            @Override
            public void onStart() {
                super.onStart();
```

```

        loadTasksCallBack.onStart();
    }

    @Override
    protected void onSuccess(IpInfo ipInfo) {
        super.onSuccess(ipInfo);
        loadTasksCallBack.onSuccess(ipInfo);
    }

    @Override
    public void onFinish() {
        super.onFinish();
        loadTasksCallBack.onFinish();
    }

    @Override
    public void onFailure(int errorCode, String msg) {
        super.onFailure(errorCode, msg);
        loadTasksCallBack.onFailed();
    }
}
});
}
}

```

`IpInfoTask` 是一个单例类，在 `execute` 方法中通过 `OkHttpFinal` 来获取数据，同时在 `OkHttpFinal` 的回调函数中调用自己定义的回调函数 `loadTasksCallBack`。

2. 实现 Presenter

首先定义一个契约接口 `IpInfoContract`，契约接口主要用来存放相同业务的 `Presenter` 和 `View` 的接口，便于查找和维护。其代码如下所示：

```

public interface IpInfoContract {
    interface Presenter {
        void getIpInfo(String ip);
    }

    interface View extends BaseView<Presenter> {
        void setIpInfo(IpInfo ipInfo);
        void showLoading();
        void hideLoading();
        void showError();
        boolean isActive();
    }
}

```

在此可以看到 **Presenter** 接口定义了获取数据的方法，而 **View** 定义了与界面交互的方法。其中，**isActive** 方法用于判断 **Fragment** 是否添加到了 **Activity** 中。另外，**View** 接口继承自 **BaseView** 接口，**BaseView** 接口如下所示：

```
public interface BaseView<T> {  
    void setPresenter(T presenter);  
}
```

BaseView 接口的目的就是给 **View** 绑定 **Presenter**，接着实现 **Presenter** 接口，如下所示：

```
public class IpInfoPresenter implements IpInfoContract.Presenter,  
LoadTasksCallBack<IpInfo> {  
    private NetTask netTask;  
    private IpInfoContract.View addTaskView;  
    public IpInfoPresenter(IpInfoContract.View addTaskView, NetTask netTask) {  
        this.netTask = netTask;  
        this.addTaskView=addTaskView;  
    }  
    @Override  
    public void getIpInfo(String ip) {  
        netTask.execute(ip,this); //1  
    }  
    @Override  
    public void onSuccess(IpInfo ipInfo) {  
        if(addTaskView.isActive()){  
            addTaskView.setIpInfo(ipInfo);  
        }  
    }  
    @Override  
    public void onStart() {  
        if(addTaskView.isActive()){  
            addTaskView.showLoading();  
        }  
    }  
    @Override  
    public void onFailed() {  
        if(addTaskView.isActive()){  
            addTaskView.showError();  
            addTaskView.hideLoading();  
        }  
    }  
    @Override  
    public void onFinish() {  
        if(addTaskView.isActive()){
```

```

        addTaskView.hideLoading();
    }
}
}

```

`IpInfoPresenter` 中含有 `NetTask` 和 `IpInfoContract.View` 的实例（后面会讲），并且实现了 `LoadTasksCallBack` 接口。在上面代码注释 1 处，将自身传入 `NetTask` 的 `execute` 方法中来获取数据，并回调给 `IpInfoPresenter`，最后通过 `addTaskView` 来和 `View` 进行交互，并更改界面。这回我们应该明白了：**Presenter** 就是一个中间人的角色，其通过 `NetTask`，也就是 **Model** 层来获得和保存数据，然后再通过 `View` 更新界面，这期间通过定义接口使得 `View` 和 **Model** 没有任何交互。最后看看 `View` 层的实现。

3. 实现 View

在上面的契约接口 `IpInfoContract` 中我们已经定义了 `View` 接口，实现它的是 `IpInfoFragment`。其代码如下所示：

```

public class IpInfoFragment extends Fragment implements IpInfoContract.View {
    private TextView tv_country;
    private TextView tv_area;
    private TextView tv_city;
    private Button bt_ipinfo;
    private Dialog mDialog;
    private IpInfoContract.Presenter mPresenter;
    public static IpInfoFragment newInstance() {
        return new IpInfoFragment();
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View root = inflater.inflate(R.layout.fragment_ipinfo, container,
            false);
        tv_country= (TextView) root.findViewById(R.id.tv_country);
        tv_area= (TextView) root.findViewById(R.id.tv_area);
        tv_city= (TextView) root.findViewById(R.id.tv_city);
        bt_ipinfo= (Button) root.findViewById(R.id.bt_ipinfo);
        return root;
    }
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        mDialog=new ProgressDialog(getActivity());
        mDialog.setTitle("获取数据中");
    }
}

```

```

        bt_ipinfo.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                mPresenter.getIpInfo("39.155.184.147");//2
            }
        });
    }
    @Override
    public void setPresenter(IpInfoContract.Presenter presenter) {//1
        mPresenter=presenter;
    }
    @Override
    public void setIpInfo(IpInfo ipInfo) {
        if(ipInfo!=null&&ipInfo.getData()!=null){
            IpData ipData=ipInfo.getData();
            tv_country.setText(ipData.getCountry());
            tv_area.setText(ipData.getArea());
            tv_city.setText(ipData.getCity());
        }
    }
    @Override
    public void showLoading() {
        mDialog.show();
    }
    @Override
    public void hideLoading() {
        if(mDialog.isShowing()) {
            mDialog.dismiss();
        }
    }
    @Override
    public void showError() {
        Toast.makeText(getActivity().getApplicationContext(),"网络出错",
            Toast.LENGTH_SHORT).show();
    }
    @Override
    public boolean isActive() {
        return isAdded();
    }
}

```

在上面代码注释 1 处通过实现 `setPresenter` 方法来注入 `IpInfoPresenter`。在注释 2 处则调用

IpInfoPresenter 的 `getIpInfo` 方法来获取 IP 地址的信息。另外, `IpInfoFragment` 实现了 `View` 接口, 用来接收 `IpInfoPresenter` 的回调并更新界面。那么 `IpInfoFragment` 是在哪里调用 `setPresenter` 来注入 `IpInfoPresenter` 的呢? 答案是在 `IpInfoActivity` 中, 代码如下所示:

```
public class IpInfoActivity extends AppCompatActivity {
    private IpInfoPresenter ipInfoPresenter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_ipinfo);
        IpInfoFragment ipInfoFragment = (IpInfoFragment)
            getSupportFragmentManager().findFragmentById(R.id.contentFrame);
        if (ipInfoFragment == null) {
            ipInfoFragment = IpInfoFragment.newInstance();//1
            ActivityUtils.addFragmentToActivity(getSupportFragmentManager(),
                ipInfoFragment, R.id.contentFrame);//2
        }
        IpInfoTask ipInfoTask = IpInfoTask.getInstance();
        ipInfoPresenter = new IpInfoPresenter(ipInfoFragment, ipInfoTask);
        ipInfoFragment.setPresenter(ipInfoPresenter);//3
    }
}
```

在这个例子中 `IpInfoActivity` 并不作为 `View` 层, 而是作为 `View`、`Model` 和 `Presenter` 三层的纽带。在上面代码注释 1 处的代码新建 `IpInfoFragment`, 接着通过注释 2 处的代码来将 `IpInfoFragment` 添加到 `IpInfoActivity` 中。紧接着创建 `IpInfoTask`, 并将它和 `IpInfoFragment` 作为参数传入 `IpInfoPresenter`, 并在注释 3 处将 `IpInfoPresenter` 注入到 `IpInfoFragment` 中。可以看到 `IpInfoPresenter` 和 `IpInfoFragment` 是互相注入的。注释 2 处的 `ActivityUtils` 的代码如下所示:

```
public class ActivityUtils {
    public static void addFragmentToActivity(FragmentManager fragmentManager,
                                             Fragment fragment, int frameId) {
        FragmentTransaction transaction = fragmentManager.beginTransaction();
        transaction.add(frameId, fragment);
        transaction.commit();
    }
}
```

上面的代码负责提交事务, 将 `Fragment` 添加到 `Activity` 中。整个项目的结构目录如图 10-3 所示。

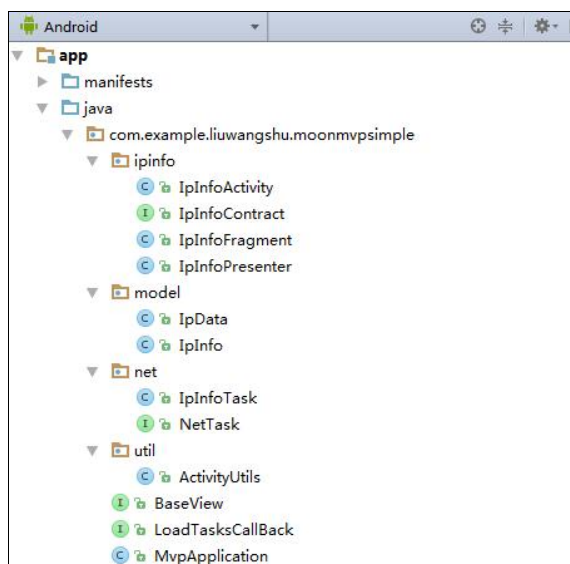


图 10-3 项目的结构目录

在图 10-3 中可以看到，ipinfo 包用于存放查询 IP 地址信息业务的类，net 包用于存放访问网络的类，model 包用于存放实体类。为了方便理解，这里给出这个 MVP 例子的结构图，如图 10-4 所示。

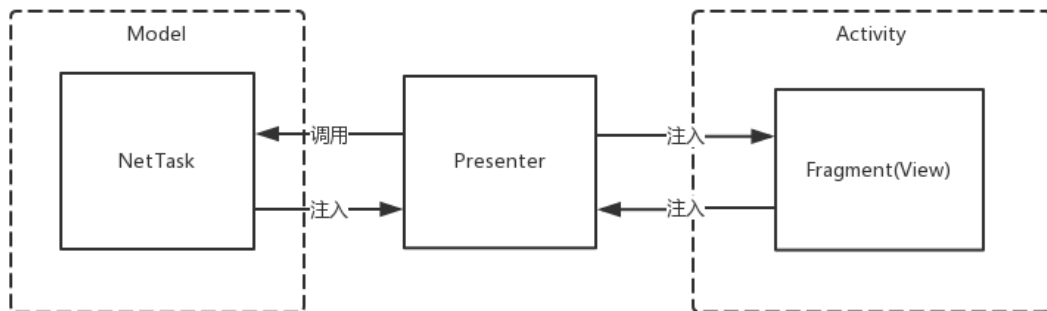


图 10-4 项目的结构图

从图 10-4 中可以看出，View 和 Model 之间没有联系，View 与 Presenter 通过接口进行交互，并在 Activity 中进行相互注入。Model 层的 NetTask 在 Activity 中注入 Presenter，并等待 Presenter 调用。好了，MVP 的简单例子就讲到这里。其实还有很多种方式实现 MVP，在这里我只是讲了一种最基础的方式。如果读者感兴趣，可以查看谷歌官方的 MVP 实例：<https://github.com/googleamples/android-architecture/tree/todo-mvp/>。

10.2.2 MVP 结合 RxJava 和 Dagger2

在第 8 章和第 9 章中我们分别介绍了 RxJava 和 Dagger2，本节将介绍如何将它们应用到 MVP 模式中。首先，我们先结合 10.2.1 节的例子介绍 MVP 结合 RxJava，之后，我们在此基础

上再结合 Dagger2。

1. MVP 结合 RxJava

本节的例子是在 10.2.1 节的例子基础上进行更改的，主要加入了 RxJava 和 5.7 节讲过的 Retrofit，用来访问网络。为此，首先要配置 build.gradle，如下所示：

```
dependencies {
    ...
    compile 'io.reactivex:rxjava:1.2.0'
    compile 'io.reactivex:rxandroid:1.2.1'
    compile 'com.squareup.retrofit2:retrofit:2.1.0'
    compile 'com.squareup.retrofit2:converter-gson:2.1.0'
    compile 'com.squareup.retrofit2:adapter-rxjava:2.1.0'
}
```

定义访问网络的接口，如下所示：

```
public interface IpService {
    @FormUrlEncoded
    @POST("getIpInfo.php")
    Observable<IpInfo> getIpInfo(@Field("ip") String ip);
}
```

getIpInfo 方法返回 Observable 类型是为了支持 RxJava。接着修改 NetTask 接口：

```
public interface NetTask<T> {
    Subscription execute(T data , LoadTasksCallBack callBack);
}
```

紧接着修改 NetTask 接口的实现类 IpInfoTask，代码如下所示：

```
public class IpInfoTask implements NetTask<String> {
    private static IpInfoTask INSTANCE = null;
    private static final String HOST = "http://ip.taobao.com/service/";
    private Retrofit retrofit;
    private IpInfoTask() {
        createRetrofit();//1
    }
    public static IpInfoTask getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new IpInfoTask();
        }
        return INSTANCE;
    }
    private void createRetrofit() {
```

```

        retrofit = new Retrofit.Builder().
            baseUrl(HOST).
            addConverterFactory(GsonConverterFactory.create()).
            addCallAdapterFactory(RxJavaCallAdapterFactory.create()).
            build();
    }
    @Override
    public Subscription execute(String ip, final LoadTasksCallBack
loadTasksCallBack) {
        IpService ipService = retrofit.create(IpService.class);
        Subscription Subscription = ipService.getIpInfo(ip).subscribeOn
(Schedulers.io()).observeOn(AndroidSchedulers.mainThread())
.subscribe(new Subscriber<IpInfo>() {
            @Override
            public void onStart() {
                loadTasksCallBack.onStart();
            }
            @Override
            public void onCompleted() {
                loadTasksCallBack.onFinish();
            }
            @Override
            public void onError(Throwable e) {
                loadTasksCallBack.onFailed();
            }
            @Override
            public void onNext(IpInfo ipInfo) {
                loadTasksCallBack.onSuccess(ipInfo);
            }
        });
        return Subscription;
    }
}

```

IpInfoTask 的变化主要是使用了 RxJava 和 Retrofit 替代 OkHttpFinal 来访问网络，在上面代码注释 1 处调用 createRetrofit 方法来创建和配置 Retrofit。在 execute 方法中使用 RxJava 结合 Retrofit 来访问网络，并返回 Subscription（在 8.5.2 节已经讲过它们结合起来访问网络的用法，这里就不赘述了）。为了能取消网络请求，我们先来定义 BasePresenter 接口：

```

public interface BasePresenter {
    void subscribe();
    void unsubscribe();
}

```

接下来改写 **Presenter** 接口，使它继承 **BasePresenter** 接口：

```
public interface IpInfoContract {
    interface Presenter extends BasePresenter{
        void getIpInfo(String ip);
    }
    ...
}
```

改写 **IpInfoPresenter** 类，代码如下所示：

```
public class IpInfoPresenter implements IpInfoContract.Presenter,
LoadTasksCallBack<IpInfo> {
    private NetTask netTask;
    private IpInfoContract.View addTaskView;
    private CompositeSubscription mSubscriptions;
    private Subscription subscription;
    public IpInfoPresenter(IpInfoContract.View addTaskView, NetTask netTask) {
        this.netTask = netTask;
        this.addTaskView = addTaskView;
        mSubscriptions = new CompositeSubscription();
    }
    @Override
    public void getIpInfo(String ip) {
        subscription = netTask.execute(ip, this);//1
        subscribe();
    }
    @Override
    public void onSuccess(IpInfo ipInfo) {
        if (addTaskView.isActive()) {
            addTaskView.setIpInfo(ipInfo);
        }
    }
    ...
    @Override
    public void subscribe() {
        if(subscription!=null) {
            mSubscriptions.add(subscription);
        }
    }
    @Override
    public void unsubscribe() {
        if (mSubscriptions != null && mSubscriptions.hasSubscriptions()) {
            mSubscriptions.unsubscribe();
        }
    }
}
```

```

    }
}
}

```

当调用 `getIpInfo` 方法时就会调用上面代码注释 1 处的代码来访问网络并返回 `subscription`，接着调用 `subscribe` 方法将 `subscription` 传入 `mSubscriptions`，此外在 `unsubscribe` 方法中调用 `mSubscriptions` 的 `unsubscribe` 方法。在 8.5.2 节中，我们讲过这个方法内部会调用 `Call` 的 `cancel` 方法来取消网络请求。最后改写 `IpInfoFragment`，加入如下代码来取消网络请求：

```

@Override
public void onPause() {
    super.onPause();
    mPresenter.unsubscribe();
}

```

2. MVP 结合 Dagger2

本小节要在上面代码示例的基础上加入 `Dagger2`。从图 10-4 中可以发现，在 MVP 模式中我们应用了多次注入，具体注入代码如下所示：

```

IpInfoTask ipInfoTask = new IpInfoTask();
ipInfoPresenter = new IpInfoPresenter(ipInfoFragment, ipInfoTask);
ipInfoFragment.setPresenter(ipInfoPresenter);

```

将 `IpInfoTask` 注入 `IpInfoPresenter`，再将 `IpInfoPresenter` 注入 `IpInfoFragment`，下面我们就用 `Dagger2` 来替代这些代码完成注入。我们可以将这次替代工作分为两部分：一部分是 `NetTask` 的相关注入，另一部分是 `IpInfoPresenter` 的相关注入。首先我们要在项目中引入 `Dagger2` 的依赖库，这在 9.3.3 节中已经介绍过了，这里就不赘述了。

(1) NetTask 相关注入

首先我们编写 `NetTaskModule`，代码如下所示：

```

@Module
public class NetTaskModule {
    @Singleton
    @Provides
    NetTask provideIpInfoTask() {
        return new IpInfoTask(); //1
    }
}

```

`NetTaskModule` 相当于一个工厂类，用来提供对象。在上面代码注释 1 处新建了 `IpInfoTask`。在此前的代码中 `IpInfoTask` 是一个单例，在这里去掉 `IpInfoTask` 的单例模式，因为用 `Dagger2` 可以实现 `IpInfoTask` 的全局单例。当然，起作用的并不是 `@Singleton`。接下来实现

NetTaskComponent，代码如下所示：

```
@Singleton
Component(modules =NetTaskModule.class)//1
public interface NetTaskComponent {
    NetTask getNetTask();
}
```

在上面代码注释 1 处引入 NetTaskModule。接着改写 MvpApplication:

```
public class MvpApplication extends Application {
    private NetTaskComponent netTaskComponent;
    @Override
    public void onCreate() {
        super.onCreate();
        netTaskComponent=DaggerNetTaskComponent.builder().netTaskModule
            (new NetTaskModule()).build();
    }
    public NetTaskComponent getTasksRepositoryComponent() {
        return netTaskComponent;
    }
}
```

可以看到 NetTaskComponent 是一个全局单例，因此 IpInfoTask 也是一个全局单例。

(2) IpInfoPresenter 相关注入

为了方便管理 Component，创建 FragmentScoped:

```
@Documented
@Scope
@Retention(RetentionPolicy.RUNTIME)
public @interface FragmentScoped {
}
```

IpInfoPresenter 的构造方法需要一个 NetTask 和一个 IpInfoContract.View，此前我们已经将 NetTask 相关注入的逻辑处理好了。接下来处理 IpInfoContract.View 的相关注入逻辑。首先创建 IpInfoModule，如下所示：

```
@Module
public class IpInfoModule {
    private IpInfoContract.View mView;
    public IpInfoModule(IpInfoContract.View mView){
        this.mView=mView;
    }
    @Provides
```

```

        IpInfoContract.View provideIpInfoContract() {
            return mView;
        }
    }
}

```

`IpInfoModule` 用来提供 `IpInfoContract.View`，但是这个 `IpInfoContract.View` 是 `IpInfoModule` 的构造方法传进来的。接着照例创建 `IpInfoComponent`：

```

@FragmentScoped
@Component(dependencies = NetTaskComponent.class, modules = IpInfoModule.class)
public interface IpInfoComponent {
    void inject(IpInfoActivity ipInfoActivity);
}

```

注入的目标类是 `IpInfoActivity`。紧接着改写 `IpInfoPresenter`，如下所示：

```

public class IpInfoPresenter implements IpInfoContract.Presenter,
LoadTasksCallBack<IpInfo> {
    private NetTask netTask;
    private IpInfoContract.View addTaskView;
    private CompositeSubscription mSubscriptions;
    private Subscription subscription;
    @Inject
    public IpInfoPresenter(IpInfoContract.View addTaskView, NetTask netTask) {
        this.netTask = netTask;
        this.addTaskView = addTaskView;
        mSubscriptions = new CompositeSubscription();
    }
    @Inject
    void setPresenter() {
        addTaskView.setPresenter(this); //1
    }
    ...
}

```

`IpInfoPresenter` 的构造方法用 `@Inject` 标记，意味着 `Dagger2` 可以使用 `IpInfoPresenter` 构造方法来构建 `IpInfoPresenter`。`IpInfoPresenter` 构造方法需要的两个参数，我们已经处理好准备注入了。接下来 `setPresenter` 方法用 `@Inject` 标记，使用的是方法注入。这个 `setPresenter` 方法会紧接着 `IpInfoPresenter` 的构造方法调用后调用。很明显它是为了将自身传入 `IpInfoContract.View` 的 `setPresenter` 方法中。这相当于替代了此前在 `IpInfoActivity` 中定义的 `ipInfoFragment.setPresenter(ipInfoPresenter)`。最后在 `IpInfoActivity` 中完成注入操作，如下所示：

```

public class IpInfoActivity extends AppCompatActivity {
    @Inject

```

```

        IpInfoPresenter ipInfoPresenter;//2
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_ipinfo);
            IpInfoFragment ipInfoFragment = (IpInfoFragment)
            getSupportFragmentManager().findFragmentById(R.id.contentFrame);
            if (ipInfoFragment == null) {
                ipInfoFragment = IpInfoFragment.newInstance();
                ActivityUtils.addFragmentToActivity(getSupportFragmentManager(),
                    ipInfoFragment, R.id.contentFrame);
            }
            //        IpInfoTask ipInfoTask = new IpInfoTask();
            //        ipInfoPresenter = new IpInfoPresenter(ipInfoFragment, ipInfoTask);//3
            //        ipInfoFragment.setPresenter(ipInfoPresenter);//4
            DaggerIpInfoComponent.builder().ipInfoModule(new IpInfoModule
            (ipInfoFragment)).netTaskComponent(((MvpApplication) getApplication()).
            getTasksRepositoryComponent()).build().inject(this);//1
        }
    }
}

```

在上面代码注释 1 处完成注入操作，它和注释 2 处的代码替代了被注释掉的 3 行代码。首先注释 1 处的代码分别将我们需要的 `IpInfoContract.View` 和 `NetTask` 注入 `IpInfoActivity` 中。其中，`IpInfoContract.View` 就是 `IpInfoModule` 传进来的 `ipInfoFragment`。紧接着注释 2 处的代码调用了 `IpInfoPresenter` 的构造方法完成了相当于注释 3 处代码的作用，调用完 `IpInfoPresenter` 的构造方法后会紧接着调用 `addTaskView.setPresenter(this)`来完成注释 4 处代码的作用。

10.3 MVVM 模式

MVVM(Model-View-ViewModel)最早于 2005 年被微软的 WPF 和 Silverlight 的架构师 John Gossman 提出。将 Presenter 改为 ViewModel，其和 MVP 类似，不同的是 ViewModel 跟 Model 和 View 进行双向绑定：当 View 发生改变时，ViewModel 通知 Model 进行更新数据；同理 Model 数据更新后，ViewModel 通知 View 更新。MVVM 的结构图如图 10-5 所示。在 2015 年 Google 的 I/O 大会中，发布了 MVVM 的支持库 Data Binding。因此，在讲到 MVVM 的实现前，我们首先要了解 Data Binding。

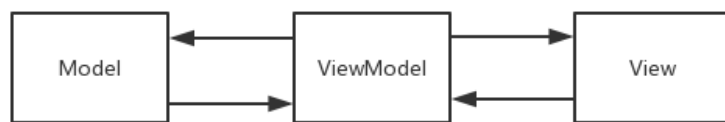


图 10-5 MVVM 模式

10.3.1 解析 Data Binding

在 Data Binding 之前，我们不可避免地要编写大量诸如 `findViewById()`、`setText()` 和 `setOnClickListener()` 等代码。通过 Data Binding，我们可以通过声明式布局以精简的代码来绑定应用程序逻辑和布局，这样就不用编写大量的模板代码了。若想使用 Data Binding，要确保 Android 的 Gradle 插件版本不低于 1.5.0-alpha1，Android Studio 的版本不低于 1.3。在 Module 的 `build.gradle` 中加上如下配置就可以使用了：

```
android {  
    ...  
    dataBinding {  
        enabled = true  
    }  
}
```

1. 基础使用

现在我们写一个简单的例子来展现 Data Binding 的基本使用方法。首先，我们创建一个简单的 Model 实体类，如下所示：

```
public class Swordsman {  
    private String name;  
    private String level;  
    public Swordsman(String name, String level) {  
        this.name = name;  
        this.level = level;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getLevel() {  
        return level;  
    }  
    public void setLevel(String level) {  
        this.level = level;  
    }  
}
```

接着编写布局文件，如下所示：

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



```

xmlns:android="http://schemas.android.com/apk/res/android">
<data>
    <variable
        name="swordsman"
        type="com.example.liuwangshu.moondatabinding.model.Swordsman" />
</data>
<LinearLayout
    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="@{swordsman.name}" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{swordsman.level}" />
</LinearLayout>
</layout>

```

根节点由某一个容器（比如 `LinearLayout`）变为 `layout`，`layout` 中包含了 `data` 节点和传统的视图，在 `data` 中定义了 `variable` 节点。其中，`name` 属性表示变量的名称，`type` 表示这个变量的类型，也就是 `Swordsman` 这个实体类的位置。`variable` 节点的每一个变量都会在 `Binding` 辅助类中生成对应的 `getters` 和 `setters`。接着将 `@{swordsman.name}` 和 `@{swordsman.level}` 赋值给 `TextView` 的 `text` 属性。最后在 `Activity` 中将实体类和布局文件进行了绑定，如下所示：

```

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityMainBinding binding = DataBindingUtil.setContentView(this,
            R.layout.activity_main); //1
        Swordsman swordsman = new Swordsman("张无忌", "S");
        binding.setSwordsman(swordsman); //2
    }
}

```

上面代码注释 1 处的代码替换 `setContentView()`，注释 2 处的代码用来给布局中的控件进行赋值。其中 `ActivityMainBinding` 是系统根据 `activity_main.xml` 生成的一个 `ViewModel` 类（`Binding` 辅助类），它包含了布局文件中所有的绑定关系，并会根据绑定表达式给布局文件赋值。运行程序，效果如图 10-6 所示。

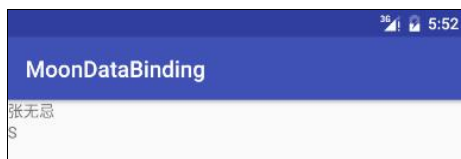


图 10-6 Data Binding 例子运行效果

2. 事件处理

Data Binding 也可以对点击事件进行处理，其有两种写法。其中第一种写法如下所示：

```
...
<Button
    android:id="@+id/bt_next"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
/>
...
```

首先在 XML 中定义一个 Button，并设定它的 id。接下来在 Java 代码中引用它：

```
...
binding.btNext.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});
...
```

btNext 指的就是 XML 中 id 为 bt_next 的 Button。当我们给控件设定 id 时，就会在 Binding 辅助类中生成一个相应的 public final 字段，以供调用。

还有一种写法，如下所示：

```
...

<variable
    name="OnClicklistener"
    type="android.view.View.OnClickListener" />

...
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="@{OnClicklistener}"
/>

...
```

在控件里通过@{}来设置自己的点击事件，接着在 Java 代码中引用它：

```
binding.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

    }
});
```

3. 布局属性

Data Binding 的 XML 布局中有很多用法，本节将一一介绍这些用法。

(1) import 用法与别名

data 节点支持 import 用法，如下所示：

```
<data>
    <import type="com.example.liuwangshu.moondatabinding.model.Swordsman"/>
    <variable
        name="swordsman"
        type="Swordsman" />
</data>
```

data 节点中的 import 需要写明具体导入的类名，这一点和 Java 的 import 是不同的。如果 import 引用了两个相同的类名，我们就可以用别名来区分它们，如下所示：

```
<data>
    <import type="com.example.liuwangshu.moondatabinding.model.Swordsman"
        alias="man"/>
    <import type="com.example.liuwangshu.moondatabinding.Swordsman"/>
    <variable
        name="man"
        type="Swordsman" />
    <variable
        name="swordsman"
        type="Swordsman"/>
</data>
```

(2) 变量定义

在前面，我们知道了如何在 XML 中定义实体类。此外，我们也可以定义一些基本数据类型和 String 类型：

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

```

<data>
    <variable
        name="name"
        type="String"/>
    <variable
        name="age"
        type="int" />
    <variable
        name="man"
        type="boolean" />
</data>
<LinearLayout
    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="@{name}" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{String.valueOf(age)}" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{String.valueOf(man)}" />

</LinearLayout>
</layout>

```

java.lang.* 包中的类会被自动导入，因此无须使用 import。在 XML 中分别定义了 String、int 和 boolean 类型，并在 TextView 中使用这些定义的变量。接下来在 Activity 中使用变量，如下所示：

```

...
    binding.setName("风清扬");
    binding.setAge(30);
    binding.setMan(true);
...
}

```

除了可以定义基本类型的变量外，我们还可以定义 List、Map 等这样的集合变量，如下所示：

```

<layout
    xmlns:android="http://schemas.android.com/apk/res/android">

```

```

<data>
    <import type="java.util.ArrayList" />
    <import type="java.util.Map" />
    <variable
        name="list"
        type="ArrayList<String>" />
    <variable
        name="map"
        type="Map<String,String>" />
    <variable
        name="arrays"
        type="String[]" />
</data>

<LinearLayout
    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="@{list.get(1)}" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{map.get(`age`)}" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{arrays[1]}" />

</LinearLayout>
</layout>

```

首先通过 `import` 引入需要的类。为了支持泛型，我们可以使用转义字符，虽然这样会在代码中标记红色，但却可以运行。`List` 集合可以通过 `list[index]` 来获取数值，也可以调用 `list.get(index)` 达到相同的目的。同样，`Map` 集合也是如此。在 `Activity` 中使用变量，如下所示：

```

ArrayList list=new ArrayList();
list.add("0");
list.add("1");
binding.setList(list);

Map map=new HashMap();

```

```
map.put("age", "30");
binding.setMap(map);

String[] arrays = {"张无忌", "慕容龙城"};
binding.setArrays(arrays);
```

(3) 自定义 Binding 类名

在默认情况下, Binding 辅助类的名称取决于布局文件的命名, 其以大写字母开头, 移除下画线, 后续的单词首字母大写并以 Binding 结尾。比如在我们此前的例子中, 布局文件的名称为 activity_main, 则生成的 Binding 辅助类名为 ActivityMainBinding。这个类经过编译会生成在如图 10-7 所示的目录中。

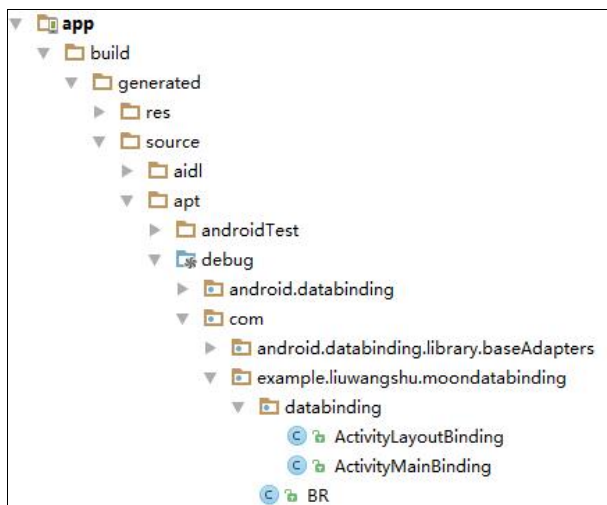


图 10-7 ActivityMainBinding 辅助类生成位置

从图 10-7 中可知, ActivityMainBinding 生成于 MainActivity 所在包的 databinding 包中。我们可以通过修改布局文件的 data 节点中的 class 属性, 来改变 Binding 辅助类的命名与生成的位置。指定全类名的方式, 我们改写 activity_main.xml 的 data 节点如下所示:

```
<data class="com.moon.ActivityFrist">
...
</data>
```

这时编译程序, 辅助类生成于 com.moon 包中, 名称为 ActivityFrist, 接着在 MainActivity 中使用它就可以了:

```
import com.moon.ActivityFrist;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```

        super.onCreate(savedInstanceState);
        ActivityFrist binding = DataBindingUtil.setContentView(this, R.
            layout.activity_main);
        ...
    }
}

```

Binding 类默认生成于 MainActivity 所在包的 databinding 包中，其效果和不设置 data 节点中的 class 属性是一样的：

```

<data class="ActivityFrist">
    ...
</data>

```

也可以通过如下代码将 Binding 类生成在项目的包的根目录下：

```

<data class=".ActivityFrist">
    ...
</data>

```

（4）静态方法调用

在布局文件中也可以直接调用静态方法，达到对数据进行转换的目的。我们首先写一个静态方法，如下所示：

```

public class Utils {
    public static String getName(Swordsman swordsman) {
        return swordsman.getName();
    }
}

```

通过 getName 方法得到 Swordsman 的 name 属性，接下来在布局文件中引入和定义我们需要的类，如下所示：

```

<layout
    xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <import type="com.example.liuwangshu.moondatabinding.util.Utils"/>
        <variable
            name="swordsman"
            type="Swordsman" />
    </data>

    <LinearLayout
        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="@{Utils.getName(swordsman)}" />

    </LinearLayout>
</layout>

```

在 `TextView` 中我们调用了 `Utils` 的 `getName` 方法。`getName` 方法需要传入的 `swordsman`，要在 `Activity` 中设定：

```

Swordsman swordsman=new Swordsman("独孤求败","SS");
binding.setSwordsman(swordsman);

```

(5) 支持表达式

XML 中的表达式与 Java 表达式有很多相似之处。下面是二者的相同之处。

- 数学表达式：+ - / * %
- 字符串拼接：+ -
- 逻辑表达式：&& ||
- 位操作符：& | ^
- 一元操作符：+ - ! ~
- 位移操作符：>> >>> <<
- 比较操作符：== > < >= <=
- instanceof
- 分组操作符：()
- 字面量
- character, String, numeric, null
- 强转、方法调用
- 字段访问
- 数组访问：[]
- 三元操作符：?:

下面举两个简单的例子：

```

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{'Age :` + String.valueOf(age)}"
/>

```



```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="三目运算"
    android:visibility="@{man?View.VISIBLE:View.GONE}"
/>
```

第一个 `TextView` 用了字符串的拼接。第二个 `TextView` 则采用了三目运算，用到了 `View.VISIBLE` 和 `View.GONE`，因此不要忘了 `import View` 类。

(6) Converter

`Converter` 指的是转换器，其把数据格式转为需要的格式。假设 `TextView` 需要显示一个 `String` 类型的数据，但是你只有一个 `Date` 类型的数据，解决方法就是可以定义一个静态方法来做格式转换，也可以使用 `Data Binding` 提供的 `Converter`。下面先来定义布局文件：

```
<layout
    xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="time"
            type="java.util.Date" />
    </data>

    <LinearLayout
        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="@{time}"
        />
    </LinearLayout>
</layout>
```

`TextView` 的 `text` 属性需要的是 `String` 类型的值，这里给了一个 `Date` 类型的值，因此，我们需要写一个 `Converter` 来进行类型转换：

```
public class Utils {
    @BindingConversion
    public static String convertDate(Date date) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        return sdf.format(date);
    }
}
```

`convertDate` 方法在哪个类中并不重要，重要的是 `@BindingConversion` 注解。`convertDate` 方法会将 `Date` 类型转换为 `String` 类型的日期并且返回。最后在 `Activity` 中使用 `TextView` 控件，如下所示：

```
ActivityLayoutBinding binding=DataBindingUtil.setContentView(this, R.
    layout.activity_layout);
binding.setTime(new Date());
```

运行程序会在界面上显示当前时间。

4. 动态更新和双向绑定

在此前的例子中，如果 `Model` 实体类的内容发生变化，那么界面 `UI` 是不会动态更新的。`Data Binding` 提供了 3 种动态更新机制，根据 `Model` 实体类的内容来动态更新 `UI`，分别对应于类（`Observable`）、字段（`ObservableField`）和集合类型（`Observable` 容器类）。从 `MVVM` 模式的角度来讲，通过动态更新机制，当 `Model` 发生变化时，就会通知 `ViewModel` 对 `View` 进行动态更新。结合动态更新机制，我们还可以实现双向绑定：当 `View` 发生变化时，`ViewModel` 也会通知 `Model` 进行数据更新。下面首先学习 3 种动态更新机制，随后学习双向绑定知识。

（1）使用 `Observable`

通过继承 `BaseObservable` 来实现动态更新，代码如下所示：

```
public class ObSwordsman extends BaseObservable {
    private String name;
    private String level;
    public ObSwordsman(String name, String level) {
        this.name = name;
        this.level = level;
    }
    @Bindable
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
        notifyPropertyChanged(BR.name);
    }
    @Bindable
    public String getLevel() {
        return level;
    }
    public void setLevel(String level) {
        this.level = level;
```

```

        notifyPropertyChanged(BR.level);
    }
}

```

我们只需要在 `getter` 上使用 `@Bindable` 注解，在 `setter` 中通知更新就可以了。其中 `BR` 是编译时生成的类，其功能与 `R.java` 类似，用 `@Bindable` 标记过的 `getter` 方法会在 `BR` 中生成一个相应的字段。在 `setter` 中调用 `notifyPropertyChanged(BR.name)` 通知系统 `BR.name` 这个字段的数据已经发生变化并更新 UI。我们在 `Activity` 中使用 `ObSwordsman`，如下所示：

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = DataBindingUtil.setContentview(this, R.layout.activity_update);
    obSwordsman = new ObSwordsman("任我行", "A");
    binding.setObSwordsman(obSwordsman);
    binding.btUpdateObSwordsman.setOnClickListener(new
    View.OnClickListener() {
        @Override
        public void onClick(View view) {
            obSwordsman.setName("石破天");
        }
    });
}
}

```

首先创建 `ObSwordsman` 类，并与布局文件进行绑定。当点击 `Button` 时，更改了 `ObSwordsman` 的 `name` 字段，这时候会发现，界面上显示的 `name` 也发生了改变。布局代码如下所示：

```

<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <import type="com.example.liuwangshu.moondatabinding.model.
        ObSwordsman" />
        <variable
            name="obswordsman"
            type="ObSwordsman" />
    </data>
    <LinearLayout
        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="@{obswordsman.name}" />
    </LinearLayout>

```

```

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{obswordsman.level}" />
<Button
    android:id="@+id/bt_updata_obswordsman"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="使用 Observable 更新数据" />
</LinearLayout>
</layout>

```

(2) 使用 ObservableField

除了继承 `BaseObservable` 来实现动态更新外，还可以使用系统为我们提供的所有基本数据类型对应的 `Observable` 类，比如 `ObservableInt`、`ObservableFloat`、`ObservableBoolean` 等，也可以使用引用数据类型和基本数据类型通用的 `ObservableField`。它们都继承自 `BaseObservable`。现在以 `ObservableField` 为例，创建一个 `ObFSwordsman` 类，如下所示：

```

public class ObFSwordsman {
    public ObservableField<String> name = new ObservableField<>();
    public ObservableField<String> level = new ObservableField<>();
    public ObFSwordsman(String name, String level) {
        this.name.set(name);
        this.level.set(level);
    }
    public ObservableField<String> getName() {
        return name;
    }
    public void setName(ObservableField<String> name) {
        this.name = name;
    }
    public ObservableField<String> getLevel() {
        return level;
    }
    public void setLevel(ObservableField<String> level) {
        this.level = level;
    }
}

```

其实现起来比继承 `BaseObservable` 要方便一些，接下来在 `Activity` 更新数据：

```

@Override
protected void onCreate(Bundle savedInstanceState) {

```

```

        super.onCreate(savedInstanceState);
        binding = DataBindingUtil.setContentView(this, R.layout.activity_
update);
        obfSwordsman = new ObFSwordsman("风清扬", "S");
        binding.setObfsswordsman(obfSwordsman);
        binding.btUpdataObfsswordsman.setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View view) {
                obfSwordsman.name.set("令狐冲");
            }
        });
    }
}

```

(3) 使用 Observable 容器类

如果有多个 Swordsman 类型的数据需要动态更新怎么办？系统同样也提供了 Observable 容器类帮助我们解决这一问题。Observable 容器类包括 ObservableArrayList 和 ObservableArrayMap。这个场景使用 ObservableArrayList 是最佳选择。这时无须创建符合动态更新机制的 Model 实体类，只需要在代码中应用 ObservableArrayList 就可以了，如下所示：

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = DataBindingUtil.setContentView(this, R.layout.activity_
update);
    list = new ObservableArrayList<>();
    swordsman1 = new Swordsman("张无忌", "S");
    swordsman2 = new Swordsman("周芷若", "B");
    list.add(swordsman1);
    list.add(swordsman2);
    binding.setList(list);
    binding.btUpdataObmap.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            swordsman1.setName("杨过");
            swordsman2.setName("小龙女");
            list.add(swordsman1);
        }
    });
}
}

```

(4) 双向绑定

从 MVVM 模式的角度来讲，双向绑定就是 Model 和 View 通过 ViewModel 进行双向动态更新。前面讲到了 Model 实体类发生变化，UI 会动态更新；反过来，如果 UI 发生变化，Model 实体类该如何动态更新呢？Data Binding 在 Android Studio 2.1 Preview 3 之后，开始支持双向绑定，前提是需要结合动态更新机制。拿此前讲过的 ObSwordsman 为例，布局文件如下所示：

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <import type="com.example.liuwangshu.moondatabinding.model.
        ObSwordsman" />
        <variable
            name="obswordsman"
            type="ObSwordsman" />
    </data>
    <LinearLayout
        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="@{obswordsman.name}" />
        <EditText
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@={obswordsman.name}" />
        <Button
            android:id="@+id/bt_updata_bind"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:text="双向绑定重置" />
    </LinearLayout>
</layout>
```

在布局文件中定义 EditText 来改变 ObSwordsman 的 name 字段，关键就是将 "@{obswordsman.name}" 改为 "@={obswordsman.name}"。定义 TextView 来动态显示 ObSwordsman 的 name 字段的变化。最后定义 Button 来重置 ObSwordsman 的 name 字段。Activiy 中的代码如下所示：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```

        binding = DataBindingUtil.setContentView(this, R.layout.activity_
update);
        obSwordsman = new ObSwordsman("任我行", "A");
        binding.setObswordsman(obSwordsman);
        binding.btUpdataBind.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                obSwordsman.setName("任我行");
            }
        });
    }
}

```

这样运行程序，当 EditText 编辑的内容发生变化时，TextView 中的内容也会相应地变化。当我们点击 Button 时，EditText 以及 TextView 的内容都会重置为“任我行”。这样就实现了一个简单的双向绑定。

5. 结合 RecyclerView

Data Binding 除了在 Activity 中使用之外，还可以在 ListView 和 RecyclerView 中使用。RecyclerView 是用来替代 ListView 的，不了解 RecyclerView 的同学请查看第 1 章的内容，本节将讲解 Data Binding 结合 RecyclerView 的内容。首先创建 Activity 的布局文件(activity_recycler.xml)：

```

<layout
    xmlns:android="http://schemas.android.com/apk/res/android">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <android.support.v7.widget.RecyclerView
            android:id="@+id/recycler"
            android:layout_width="match_parent"
            android:layout_height="match_parent">
        </android.support.v7.widget.RecyclerView>
    </LinearLayout>
</layout>

```

RecyclerView 控件定义了 id，这样就可以在 Activity 中用 Data Binding 来使用 RecyclerView 控件了。接着定义 item 的布局 (item_swordsman.xml)，如下所示：

```

<layout
    xmlns:android="http://schemas.android.com/apk/res/android">
    <data >
        <import type="com.example.liuwangshu.moondatabinding.model.Swordsman"/>
        <variable

```

```

        name="swordsman"
        type="Swordsman" />
</data>
<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="@{swordsman.name}" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:text="@{swordsman.level}" />
</LinearLayout>
</layout>

```

这里仍旧显示 Swordsman 的信息, 包括 name 和 level。接下来定义 RecyclerView 的 Adapter, 如下所示:

```

public class SwordsmanAdapter extends RecyclerView.Adapter
<SwordsmanAdapter.SwordsmanViewHolder> {
    private List<Swordsman> mList;
    public SwordsmanAdapter(List<Swordsman> mList) {
        this.mList = mList;
    }
    @Override
    public SwordsmanViewHolder onCreateViewHolder(ViewGroup parent, int
    viewType) {
        ItemSwordsmanBinding binding = DataBindingUtil.inflate
        (LayoutInflater.from(parent.getContext()), R.layout.item_swordsman,
        parent, false);
        return new SwordsmanViewHolder(binding); //1
    }
    @Override
    public void onBindViewHolder(SwordsmanViewHolder holder, int position) {
        Swordsman swordsman = mList.get(position);
        holder.getBinding().setSwordsman(swordsman); //2
    }
    @Override
    public int getItemCount() {
        return mList.size();
    }
}

```



```

    }
    public class SwordsmanViewHolder extends RecyclerView.ViewHolder {
        ItemSwordsmanBinding binding;
        public SwordsmanViewHolder(ViewDataBinding binding) {
            super(binding.getRoot());
            this.binding = (ItemSwordsmanBinding) binding;
        }
        public ItemSwordsmanBinding getBinding() {
            return binding;
        }
    }
}

```

与常规 RecyclerView 的 Adapter 使用方法不同的是，在上面代码注释 1 处我们没有将 View 传入 SwordsmanViewHolder 中，而是将 ItemSwordsmanBinding 传了进去。在 SwordsmanViewHolder 中并没有通过 findViewById 来找到相应的控件，而是提供了返回 ItemSwordsmanBinding 的 getBinding 方法。在注释 2 处将相应 position 的 Swordsman 赋值给绑定的布局文件。最后在 Activity 中使用 RecyclerView，代码如下所示：

```

public class RecyclerViewActivity extends AppCompatActivity {
    private ActivityRecyclerViewBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = DataBindingUtil.setContentView(this, R.layout.activity_recycler);
        initRecyclerView();
    }
    private void initRecyclerView() {
        LinearLayoutManager manager = new LinearLayoutManager(RecyclerViewActivity.this);
        binding.recycler.setLayoutManager(manager);
        SwordsmanAdapter adapter=new SwordsmanAdapter(getList());//1
        binding.recycler.setAdapter(adapter);.//2
    }
    private List<Swordsman> getList(){
        List<Swordsman> list =new ArrayList<>();
        Swordsman swordman1=new Swordsman("杨影枫","SS");
        Swordsman swordman2=new Swordsman("月眉儿","A");
        list.add(swordman1);
        list.add(swordman2);
        return list;
    }
}

```

在此设定了两个 Swordsman 并添加到 List 中,并在上面代码注释 1 处赋值 SwordsmanAdapter。在注释 2 处调用 RecyclerView 的 setAdapter 方法,将 SwordsmanAdapter 传进去来完成适配器的设置工作。运行程序,效果如图 10-8 所示。

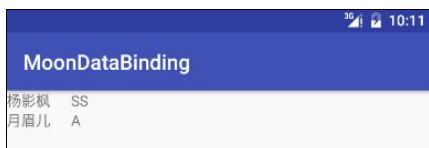


图 10-8 运行效果

10.3.2 应用 Data Binding

讲完了 MVVM 的支持库 Data Binding,本节就来应用 Data Binding。在 10.2.1 节中我们学习了 MVP 模式,并举了一个简单的例子,我们就在这个 MVP 例子的基础上使用 Data Binding。首先我们要在 build.gradle 中引入 Data Binding,此前讲过相关内容,这里就不赘述了。在这个 MVP 例子中 IpInfoActivity 只包含了一个 IpInfoFragment,因此没有必要使用 Data Binding。需要使用 Data Binding 的是 IpInfoFragment,先修改它的布局文件,如下所示:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="ipdata"
            type="com.example.liuwangshu.moonmvpsimple.model IpData" />
    </data>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:gravity="center_horizontal">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="30dp"
            android:text="@{ipdata.country}" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="30dp"
            android:text="@{ipdata.area}" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
```

```

        android:layout_marginTop="30dp"
        android:text="@{ipdata.city}" />
    <Button
        android:id="@+id/bt_ipinfo"ed
        android:layout_width="200dp"
        android:layout_height="50dp"
        android:layout_marginTop="30dp"
        android:text="获取 Ip 信息" />
</LinearLayout>
</layout>

```

在布局文件中引入了 Data Binding 相关的节点以及赋值。IpInfoFragment 的代码如下所示：

```

public class IpInfoFragment extends Fragment implements IpInfoContract.View {
    private Dialog mDialog;
    private IpInfoContract.Presenter mPresenter;
    private FragmentIpinfoBinding binding;
    public static IpInfoFragment newInstance() {
        return new IpInfoFragment();
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        if (binding == null) {
            binding = FragmentIpinfoBinding.inflate(inflater, container,
                false); //1
        }
        return binding.getRoot();
    }
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        mDialog = new ProgressDialog(getActivity());
        mDialog.setTitle("获取数据中");
        binding.btIpinfo.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                mPresenter.getIpInfo("39.155.184.147");
            }
        });
    }
    @Override
    public void setPresenter(IpInfoContract.Presenter presenter) {
        mPresenter = presenter;
    }
}

```

```
    }  
    @Override  
    public void setIpInfo(IpInfo ipInfo) {  
        if (ipInfo != null && ipInfo.getData() != null) {  
            IpData ipData = ipInfo.getData();  
            binding.setIpdata(ipData); //2  
        }  
    }  
    ...  
}
```

此前我们一直用 `DataBindingUtil` 的 `setContentView` 方法来绑定布局，上面代码注释 1 处的代码则是另一种绑定布局的方式。我们无须像此前那样将布局中的控件各个赋值，而是用注释 2 处的代码给布局中的控件进行赋值。这个例子并不算严格意义上的 MVVM 模式或 MVP 模式，因为它同时使用了 `Presenter` 和 `ViewModel`。在这个例子中主要就是使用了 `Data Binding` 以减少 `findViewById` 等样板代码的编写。

10.4 本章小结

本章讲解了应用的架构设计，主要介绍了 Android 的框架模式 MVC、MVP 和 MVVM，并讲解了 MVP 结合 `RxJava` 和 `Dagger2` 的知识，以及 MVVM 的支持库 `Data Binding`。其主要用意就是希望读者能更加重视 Android 应用的架构设计。应用架构设计是一个很庞大的知识体系，涉及很多知识点，并且随着业务和技术框架的更新，应用架构设计会变得愈加复杂并且变化繁多。本章也只是抛砖引玉地介绍了应用架构设计的基本知识点，其余内容需要读者在实际开发中去逐步拓展。

第 11 章

系统架构与 MediaPlayer 框架

本章将介绍 Android 系统架构和 MediaPlayer 框架。本书主要是讲解 Android 应用层知识的，在最后一章为何要讲解 Android 系统底层源码呢？其主要目的就是要告诉读者，作为一个 Android 开发者，了解 Android 系统底层源码是十分必要的。希望本章能起到过渡的作用：带领读者由应用层过渡到系统底层。由于篇幅有限，而系统源码的知识又太多，因此本章只会介绍系统架构和 MediaPlayer 框架的部分源码，并且不会拘泥于源码细节，旨在帮助读者学习一些入门知识。如果读者想要了解更多有关系统底层源码的知识，请阅读专门介绍系统源码的书籍。另外需要提醒大家注意的一点就是，阅读本章需要有一定的 C/C++ 基础。

11.1 Android 系统架构

Android 系统架构分为 5 层，从上到下依次是应用层、应用框架层、系统运行库层、硬件抽象层和 Linux 内核层，如图 11-1 所示。

1. 应用层

系统内置的应用程序以及非系统级的应用程序均属于应用层。其负责与用户进行直接交互，通常都是用 Java 进行开发的。

2. 应用框架层 (Java Framework)

应用框架层为开发人员提供了开发应用程序所需要的 API，我们平常开发应用程序都是调用这一层所提供的 API，当然也包括系统的应用。这一层是用 Java 代码编写的，可以称为 Java Framework。下面来看这一层所提供的主要组件，如表 11-1 所示。

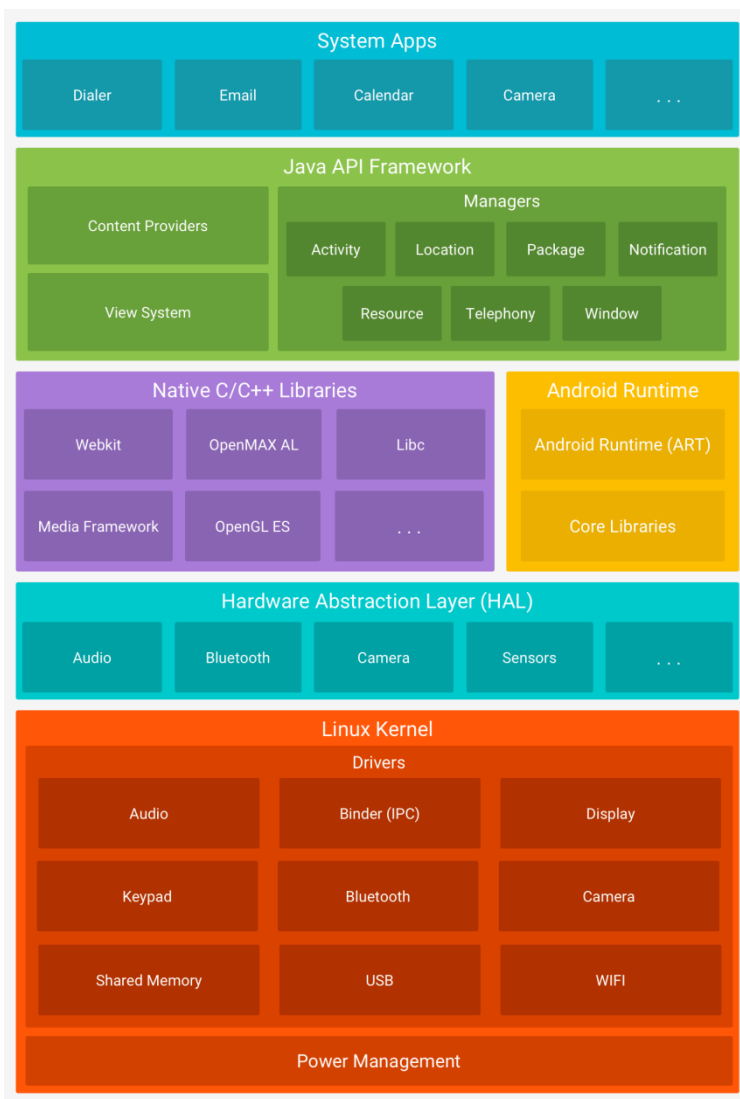


图 11-1 Android 系统架构

表 11-1 应用框架层提供的组件

名称	功能描述
Activity Manager（活动管理器）	管理各个应用程序生命周期以及通常的导航回退功能
Location Manager（位置管理器）	提供地理位置以及定位功能服务
Package Manager（包管理器）	管理所有安装在 Android 系统中的应用程序
Notification Manager（通知管理器）	使得应用程序可以在状态栏中显示自定义的提示信息
Resource Manager（资源管理器）	提供应用程序使用的各种非代码资源，如本地化字符串、图片、布局文件、颜色文件等
Telephony Manager（电话管理器）	管理所有的移动设备功能

续表

名称	功能描述
Window Manager（窗口管理器）	管理所有开启的窗口程序
Content Providers（内容提供器）	使得不同应用程序之间可以共享数据
View System（视图系统）	构建应用程序的基本组件

3. 系统运行库层（Native）

从图 11-1 中可以看出，系统运行库层分为两部分，分别是 C/C++程序库和 Android 运行时库。下面分别介绍它们。

（1）C/C++程序库

C/C++程序库能被 Android 系统中的不同组件所使用，并通过应用程序框架为开发者提供服务。表 11-2 列出了主要的 C/C++程序库。

表 11-2 主要的 C/C++程序库

名称	功能描述
OpenGL ES	3D 绘图函数库
Libc	从 BSD 继承来的标准 C 系统函数库，专门为基于嵌入式 Linux 的设备定制
Media Framework	多媒体库，支持多种常用的音频、视频格式录制和回放
SQLite	轻型的关系型数据库引擎
SGL	底层的 2D 图形渲染引擎
SSL	安全套接层，是为网络通信提供安全及数据完整性的一种安全协议
FreeType	可移植的字体引擎，它提供统一的接口来访问多种字体格式文件

（2）Android 运行时库

从图 11-1 中可以看出，运行时库又分为核心库和 ART（Android 5.0 系统之后，Dalvik 虚拟机被 ART 取代）。核心库提供了 Java 语言核心库的大多数功能，这样开发者可以使用 Java 语言来编写 Android 应用。相较于 JVM，Dalvik 虚拟机（以下简称 DVM）是专门为移动设备定制的，其允许在有限的内存中同时运行多个虚拟机的实例，并且每一个 DVM 应用作为一个独立的 Linux 进程执行。独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。而替代 DVM 的 ART 机制与 DVM 不同。在 DVM 下，应用每次运行的时候，字节码都需要通过即时编译器转换为机器码，这会使得应用的运行效率降低；而在 ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码并存储在本地，这样应用每次运行时就无须执行编译了，运行效率也会大大提升。

4. 硬件抽象层（HAL）

硬件抽象层是位于操作系统内核与硬件电路之间的接口层，其目的在于将硬件抽象化。为

了保护硬件厂商的知识产权，它隐藏了特定平台的硬件接口细节，为操作系统提供虚拟硬件平台，使其具有硬件无关性，可在多种平台上进行移植。从软硬件测试的角度来看，软硬件的测试工作都可分别基于硬件抽象层来完成，这使得软硬件测试工作的并行进行成为可能。通俗来讲，就是将控制硬件的动作放在硬件抽象层中。

5. Linux 内核层

Android 的核心系统服务基于 Linux 内核，在此基础上添加了部分 Android 专用的驱动。系统的安全性、内存管理、进程管理、网络协议栈和驱动模型等都依赖于该内核。Android 系统的 5 层架构就讲到这里，了解以上知识对我们以后分析系统源码有很大的帮助。

11.2 Android 系统源码目录

在学习 MediaPlayer 框架源码之前，我们要先了解 Android 系统源码目录，以便为后期源码学习打下基础。关于源码的阅读，你可以访问 <http://androidxref.com/> 来阅读系统源码。当然，最好是将源码下载下来，下载源码可以使用清华大学开源软件镜像站提供的 Android 镜像：<https://mirrors.tuna.tsinghua.edu.cn/help/AOSP/>。如果觉得麻烦也可以查找国内的网盘进行下载，笔者推荐使用该百度网盘地址下载：<http://pan.baidu.com/s/1ngsZs>，它提供了多个 Android 版本的源码下载。

11.2.1 整体结构

各个版本的源码目录基本是类似的，如果是编译后的源码目录，则会多增加一个 out 文件夹，用来存储编译产生的文件。Android 7.0 的根目录结构说明如表 11-3 所示。

表 11-3 系统根目录结构

Android 源码根目录	描述
abi	应用程序二进制接口
art	全新的 ART 运行环境
bionic	系统 C 库
bootable	启动引导相关代码
build	存放系统编译规则及 generic 等基础开发包配置
cts	Android 兼容性测试套件标准
dalvik	Dalvik 虚拟机
developers	开发者目录
development	应用程序开发相关
device	设备相关配置
docs	参考文档目录

续表

Android 源码根目录	描述
external	开源模组相关文件
frameworks	应用程序框架，Android 系统核心部分，由 Java 和 C++编写
hardware	主要是硬件抽象层的代码
libcore	核心库相关文件
libnativehelper	动态库，实现 JNI 库的基础
ndk	NDK 相关代码，帮助开发人员在应用程序中嵌入 C/C++代码
out	编译完成后的代码输出在此目录
pdk	Plug Development Kit 的缩写，本地开发套件
platform_testing	平台测试
prebuilts	x86 和 arm 架构下预编译的一些资源
sdk	SDK 和模拟器
packages	应用程序包
system	底层文件系统库、应用和组件
toolchain	工具链文件
tools	工具文件
Makefile	全局 Makefile 文件，用来定义编译规则

从表 11-3 中可以看出，系统源码分类清晰，内容庞大且复杂，11.4 节要讲的 MediaPlayer 框架也只是 frameworks 目录中很小的一部分。接下来分析 packages 中的内容，也就是应用层部分。

11.2.2 应用层部分

应用层位于整个 Android 系统的最上层，开发者开发的应用程序以及系统内置的应用程序都在应用层。源码根目录中的 packages 目录对应着系统应用层。它的目录结构如表 11-4 所示。

表 11-4 packages 目录

packages 目录	描述
apps	核心应用程序
experimental	第三方应用程序
inputmethods	输入法目录
providers	内容提供者目录
screensavers	屏幕保护
services	通信服务
wallpapers	墙纸

从表 11-4 的目录结构中可以发现，`packages` 目录存放着系统核心应用程序、第三方的应用程序和输入法等。这些应用都是运行在系统应用层的，因此 `packages` 目录对应着系统的应用层。

11.2.3 应用框架层部分

应用框架层是系统的核心部分，其一方面向上提供接口给应用层调用，另一方面向下与 C/C++ 程序库以及硬件抽象层等进行衔接。应用框架层的主要实现代码在 `/frameworks/base` 和 `/frameworks/av` 目录下，其中 `/frameworks/base` 目录结构如表 11-5 所示。

表 11-5 `/frameworks/base` 目录

<code>/frameworks/base</code> 目录	描述	<code>/frameworks/base</code> 目录	描述
<code>api</code>	定义 API	<code>cmds</code>	重要命令：am、app_proce 等
<code>core</code>	核心库	<code>data</code>	字体和声音等数据文件
<code>docs</code>	文档	<code>graphics</code>	图形图像相关
<code>include</code>	头文件	<code>keystore</code>	和数据签名证书相关
<code>libs</code>	库	<code>location</code>	地理位置相关库
<code>media</code>	多媒体相关库	<code>native</code>	本地库
<code>nfc-extras</code>	NFC 相关	<code>obex</code>	蓝牙传输
<code>opengl</code>	2D/3D 图形 API	<code>packages</code>	设置、TTS、VPN 程序
<code>sax</code>	XML 解析器	<code>services</code>	系统服务
<code>telephony</code>	电话通信管理	<code>test-runner</code>	测试工具相关
<code>tests</code>	测试相关	<code>tools</code>	工具
<code>wifi</code>	Wi-Fi 无线网络		

11.2.4 C/C++ 程序库部分

系统运行库层（Native）中的 C/C++ 程序库的类型繁多，功能强大，C/C++ 程序库并不完全在一个目录中。这里给出几个常用且比较重要的 C/C++ 程序库所在的目录位置，如表 11-6 所示。

表 11-6 C/C++ 程序库部分

目录位置	描述
<code>bionic/</code>	Google 开发的系统 C 库，以 BSD 许可形式开源
<code>/frameworks/av/media</code>	系统媒体库
<code>/frameworks/native/opengl</code>	第三方图形渲染库
<code>/frameworks/native/services/surfaceflinger</code>	图形显示库，主要负责图形的渲染、叠加和绘制等功能
<code>external/sqlite</code>	轻型关系数据库 SQLite 的 C++ 实现

讲完 C/C++ 程序库部分，剩下的部分我们在表 11-3 中已经给出：Android 运行时库的代码放在 `art/` 目录中。硬件抽象层的代码放在 `hardware/` 目录中，这是手机厂商改动最大的一部分，根据手机终端所采用的硬件平台会有不同的实现。

11.3 Source Insights 使用

Source Insights 是阅读系统源码的必备利器，它是 Windows 平台下的软件。首先我们来新建源码工程。通过菜单项 `Project→New Project` 即可指定源码的目录，可以添加整个 Android 系统源码，也可以只把 `/frameworks/base` 目录添加到工程中，以后再根据需要添加其他目录。这里我们只添加 `/frameworks/base` 目录的代码，如图 11-2 所示。

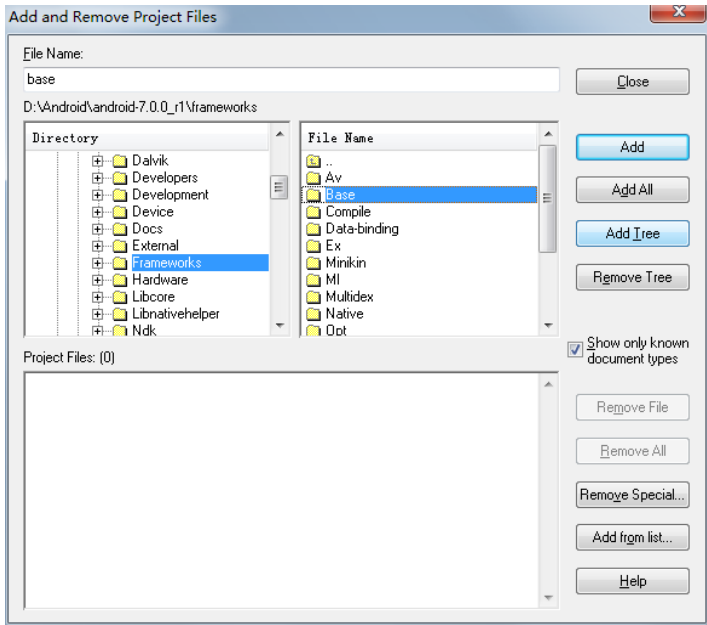


图 11-2 导入项目

选择 `/frameworks/base` 目录后点击 “Add Tree” 按钮。`base` 目录的代码就添加到源码工程中。

1. 定位文件

Source Insights 的定位文件功能十分强大，我们只需要知道源码文件名就可以轻松地找到它，比如我们要找 `MediaPlayer.java`，只要在文件搜索框中输入 “`MediaPlayer`” 即可，如图 11-3 所示。

2. 全局搜索

Source Insights 的另一个好用功能就是全局搜索，默认快捷键为 `CTRL+/,` 或者点击最上面工具栏中类似 R 的图标，弹出的搜索框如图 11-4 所示。在 `Search In` 的输入选项中我们可以自定义搜索的范围。

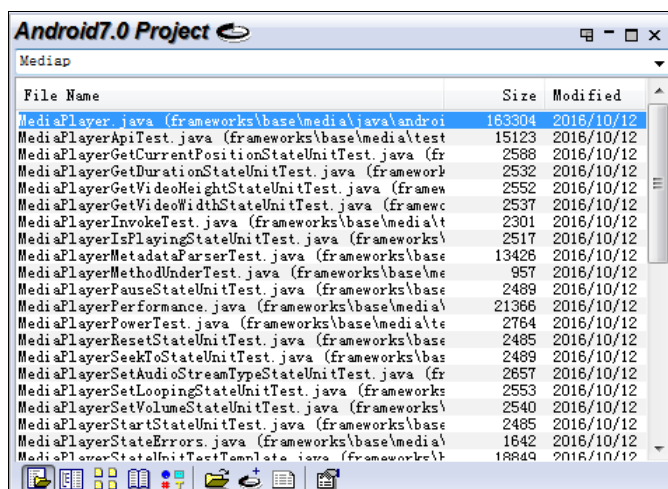


图 11-3 定位文件

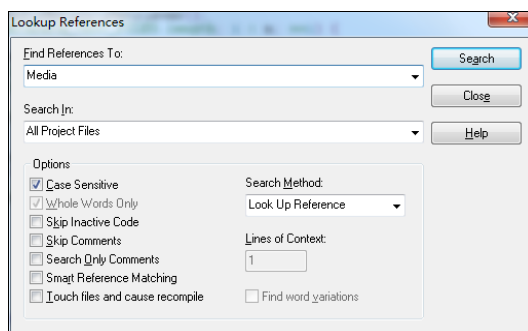


图 11-4 全局搜索

当然，Source Insights 的功能不止以上几种。相信随着使用次数的增多，你就会熟练掌握它的大部分功能，这里就不过多介绍了。

11.4 MediaPlayer 框架

在 11.1 节中我们学习了系统架构，为了更好地理解系统架构，最好的办法就是读系统源码，这需要从应用层开始分析，一直到 Native 层（又可称为 Native Framework 层）。本节介绍 MediaPlayer 框架的 Java Framework 层、JNI 层和 Native 层。

11.4.1 Java Framework 层的 MediaPlayer 分析

Java Framework 层也就是应用框架层，后文简称为 Java 层。Android 系统默认的音乐播放器 Music，在 packages/apps/Music 目录中。在 MediaPlayerService.java 的 onCreate 方法中创建了 MultiPlayer，如下所示：

packages/apps/Music/src/com/android/music/MediaPlaybackService.java

```

@Override
public void onCreate() {
    super.onCreate();

    ...

    mPlayer = new MultiPlayer();
    mPlayer.setHandler(mMediaPlayerHandler);
    ...
}

```

MultiPlayer 是 MediaPlayerService 的内部类, 它实现了对 MediaPlayer 的调用, 如下所示:
packages/apps/Music/src/com/android/music/MediaPlaybackService.java

```

private class MultiPlayer {

    ...

    private boolean setDataSourceImpl(MediaPlayer player, String path) {
        try {
            player.reset();
            player.setOnPreparedListener(null);
            if (path.startsWith("content://")) {
                player.setDataSource(MediaPlaybackService.this, Uri.parse(
                    path));
            } else {
                player.setDataSource(path);
            }
            player.setAudioStreamType(AudioManager.STREAM_MUSIC);
            player.prepare();
        } catch (IOException ex) {
            // TODO: notify the user why the file couldn't be opened
            return false;
        } catch (IllegalArgumentException ex) {
            // TODO: notify the user why the file couldn't be opened
            return false;
        }
        ...
        return true;
    }

    ...
}

```

在 MultiPlayer 的 setDataSourceImpl 方法中调用了 MediaPlayer 的 reset、setDataSource 和 prepare 等方法。这样系统应用就调用了应用框架层的 MediaPlayer。我们来查看 MediaPlayer 的

prepare 方法，如下所示：

frameworks/base/media/java/android/media/MediaPlayer.java

```
public void prepare() throws IOException, IllegalStateException {
    _prepare();
    scanInternalSubtitleTracks();
}
```

prepare 方法中调用了 _prepare() 方法：

```
private native void _prepare() throws IOException, IllegalStateException;
```

我们会发现 _prepare 方法前面有一个 native 修饰符，这表明 _prepare 是一个 native 方法，表示这个方法会由非 Java 层实现，也就是后文讲到的 JNI 层来实现。

11.4.2 JNI 层的 MediaPlayer 分析

JNI 也是用 Native 语言编写的，它属于 Native 层。但是为了便于学习和分析，本书将与 JNI 相关的代码划分到 JNI 层，JNI 层是 Native 层的一部分。在分析 JNI 层的 MediaPlayer 代码前，首先要简单介绍 JNI。JNI 是 Java Native Interface 的缩写，通过 JNI 可以让 Java 方法与 Native 方法相互调用，其中 Native 方法一般是用 C/C++ 编写的。因此，JNI 就是连接 Java 层和 Native 层的桥梁。要使用 JNI 需要先加载 JNI 库，在 MediaPlayer.java 中有如下代码：

frameworks/base/media/java/android/media/MediaPlayer.java

```
static {
    System.loadLibrary("media_jni");
    native_init();
}
```

在静态代码块中加载 JNI 库，并调用 native_init 方法。关于 native_init 方法，后文会进行分析。这里加载的名为 "media_jni" 的 JNI 库指的是 libmedia.so。MediaPlayer 的 JNI 层的代码在 frameworks/base/media/jni/android_media_MediaPlayer.cpp 中。我们需要通过 JNI 调用 Native 方法，那么是不是应该有一个结构来保存 Java 层 Native 方法与 JNI 层方法的对应关系？答案是肯定的，这个结构就是 android_media_MediaPlayer.cpp 中定义的 JNINativeMethod 数组，代码如下所示：

frameworks/base/media/jni/android_media_MediaPlayer.cpp

```
static const JNINativeMethod gMethods[] = {
    ...
    {"_setDataSource", "(Ljava/io/FileDescriptor;JJ)V", (void *)android_
        media_MediaPlayer_setDataSourceFD},
    {"_setDataSource", "(Landroid/media/MediaDataSource;)V", (void *)android_
        media_MediaPlayer_setDataSourceCallback },
```

```

        {"_setVideoSurface", "(Landroid/view/Surface;)V", (void *)android_
            media_MediaPlayer_setVideoSurface},
        {"_prepare", "()V", (void *)android_media_MediaPlayer_prepare},
        ...
    };

```

从上面来看，JNINativeMethod 数组有 3 个参数。

- 第一个参数，比如"_prepare"，它是 Java 层 Native 方法的名称。在 Java 层调用 Native 方法，交由 JNI 层来实现。
- 第二个参数是 Java 层 Native 方法的参数和返回值。其中()中的字符代表参数，后面的字母则代表返回值。
- 第三个参数是 Java 层 Native 方法对应的 JNI 层的方法，比如_prepare 方法对应 JNI 层的方法为 android_media_MediaPlayer_prepare。而通过 JNI 层的方法就可以调用 Native 层相应的方法。

前面在 MediaPlayer.java 的静态代码块中调用了 native_init 方法，这个方法是一个 Native 方法，查询 JNINativeMethod 数组，它对应的是 android_media_MediaPlayer_native_init 方法，如下所示：

frameworks/base/media/jni/android_media_MediaPlayer.cpp

```

static void
android_media_MediaPlayer_native_init(JNIEnv *env)
{
    jclass clazz;
    clazz = env->FindClass("android/media/MediaPlayer");//1
    if (clazz == NULL) {
        return;
    }
    fields.context = env->GetFieldID(clazz, "mNativeContext", "J");
    if (fields.context == NULL) {
        return;
    }
    fields.post_event = env->GetStaticMethodID(clazz, "postEventFromNative",
        "(Ljava/lang/Object;IIILjava/lang/Object;)V");//2
    ...
}

```

上面代码注释 1 处的代码是 JNI 层调用 Java 层，获取 MediaPlayer 对象。在注释 2 处的代码获取了 Java 层的 postEventFromNative 方法。可以看出 native_init 做了初始化操作，获取了上下文对象和一些方法。从上述代码可知，JNI 层可以调用 Java 层的代码；换句话说，就是 Native 层是可以通过 JNI 层来调用 Java 层代码的。下面介绍 Java 层通过 JNI 层来调用 Native 层的代码。

_prepare 对应的 JNI 层的方法为 android_media_MediaPlayer_prepare，代码如下所示：

frameworks/base/media/jni/android_media_MediaPlayer.cpp

```
static void
android_media_MediaPlayer_prepare(JNIEnv *env, jobject thiz)
{
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz); //1
    if (mp == NULL) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }
    sp<IGraphicBufferProducer> st = getVideoSurfaceTexture(env, thiz);
    mp->setVideoSurfaceTexture(st);
    process_media_player_call( env, thiz, mp->prepare(), "java/io/
    IOException", "Prepare failed." ); //2
}
```

在上面代码注释 1 处得到 MediaPlayer 的强指针，接着在注释 2 处的 process_media_player_call 方法中，调用 mp->prepare() 来调用 MediaPlayer 的 prepare 方法。也就是通过 JNI 层的 android_media_MediaPlayer_prepare 方法调用了 Native 层的 prepare 方法。因此，通过 JNI 的 NativeMethod 数组间接实现了 Java 层与 Native 层的关联，如图 11-5 所示。

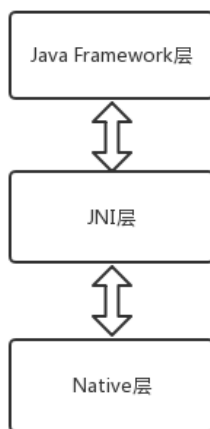


图 11-5 Java Framework 层与 Native 层的关联

11.4.3 Native 层的 MediaPlayer 分析

MediaPlayer 的 Native 层整体是一个 C/S（Client/Server）架构，Client 端和 Server 端是运行在两个进程中的，它们之间是通过 Binder 机制来进行通信的。Client 端 MediaPlayer 对应的动态库是 libmedia.so，Server 端 MediaPlayerService 对应的动态库为 libmediaservice.so。首先我们分析 Client 端的实现。

1. Client 端分析

MediaPlayer Client 端的功能实现定义的头文件为 `mediaplayer.h`，相应的源文件为 `mediaplayer.cpp`。我们接着来查看 `mediaplayer.cpp` 的 `prepare` 方法，如下所示：

frameworks/av/media/libmedia/mediaplayer.cpp

```
status_t MediaPlayer::prepare()
{
    ...
    status_t ret = prepareAsync_l();
    ...
}
```

`prepare` 方法会调用 `prepareAsync_l` 方法：

frameworks/av/media/libmedia/mediaplayer.cpp

```
status_t MediaPlayer::prepareAsync_l()
{
    if ( (mPlayer != 0) && ( mCurrentState & (MEDIA_PLAYER_INITIALIZED |
        MEDIA_PLAYER_STOPPED) ) ) {
        if (mAudioAttributesParcel != NULL) {
            mPlayer->setParameter(KEY_PARAMETER_AUDIO_ATTRIBUTES,
                *mAudioAttributesParcel);
        } else {
            mPlayer->setAudioStreamType(mStreamType);
        }
        mCurrentState = MEDIA_PLAYER_PREPARING;
        return mPlayer->prepareAsync(); //1
    }
    ALOGE("prepareAsync called in state %d, mPlayer(%p)", mCurrentState,
        mPlayer.get());
    return INVALID_OPERATION;
}
```

在上面代码注释 1 处调用了 `mPlayer` 的 `prepareAsync` 方法，那么 `mPlayer` 指的是什么呢？我们带着这个问题来查看 `mediaplayer.cpp` 的 `setDataSource` 方法，如下所示：

frameworks/av/media/libmedia/mediaplayer.cpp

```
status_t MediaPlayer::setDataSource(const sp<IDataSource> &source)
{
    ALOGV("setDataSource(IDataSource)");
    status_t err = UNKNOWN_ERROR;
    const sp<IMediaPlayerService> service(getMediaPlayerService()); //1
    if (service != 0) {
        sp<IMediaPlayer> player(service->create(this, mAudioSessionId)); //2
        if ((NO_ERROR != doSetRetransmitEndpoint(player)) ||
```

```

        (NO_ERROR != player->setDataSource(source))) {
            player.clear();
        }
        err = attachNewPlayer(player); //3
    }
    return err;
}

```

在上面代码注释 1 处通过 `getMediaPlayerService` 方法得到 `IMediaPlayerService` 指针。关于 `getMediaPlayerService` 方法，在后面的 Server 端分析中会讲到。`IMediaPlayerService` 指针指向的就是 MediaPlayer 的 Service 端：`MediaPlayerService`。又在注释 2 处通过 `IMediaPlayerService` 的 `create` 方法得到 `IMediaPlayer` 指针。通过 `IMediaPlayer` 指针就可以调用 `MediaPlayerService` 所提供的各种功能。接下来看注释 3 处的 `attachNewPlayer` 方法：

frameworks/av/media/libmedia/mediaplayer.cpp

```

status_t MediaPlayer::attachNewPlayer(const sp<IMediaPlayer>& player)
{
    ...
    mPlayer = player;
    ...
}

```

`attachNewPlayer` 方法将 `player` 赋值给 `mPlayer`。因此，前面遗留下的问题得到了解决：`mPlayer` 指的就是 `IMediaPlayer` 指针，调用 `mPlayer` 的 `prepareAsync` 方法其实就是调用 `MediaPlayerService` 的 `prepareAsync` 方法。

2. Server 端分析

在多媒体架构中除了 C/S 架构中的 Client 和 Server，还有一个全局的 `ServiceManager`，它用来管理系统中的各种 Service。Client、Server 和 `ServiceManager` 的关系如图 11-6 所示。

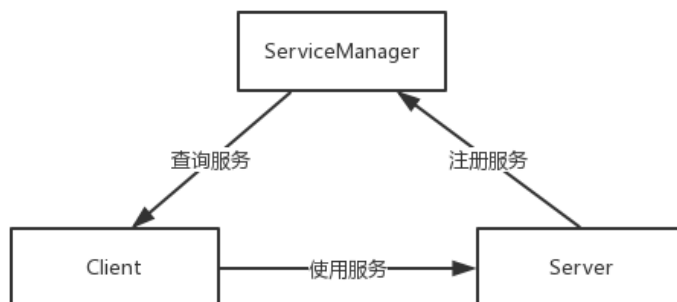


图 11-6 Client、Server 和 ServiceManager 的关系

从图 11-6 中可知，首先 Server 进程会注册一些 Service 到 `ServiceManager` 中，Client 要使用某个 Service，则需要先到 `ServiceManager` 查询 Service 的相关信息，然后根据 Service 的相关信息与 Service 所在的 Server 进程建立通信通路，这样 Client 就可以使用 Service 了。Android

的多媒体服务是由一个叫作 **MediaServer** 的服务进程提供的，它是一个可执行程序，在 Android 系统启动时，**MediaServer** 也会被启动。它的入口函数如下所示：

frameworks/av/media/mediaserver/main_mediaserver.cpp

```
int main(int argc __unused, char **argv __unused)
{
    signal(SIGPIPE, SIG_IGN);
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm(defaultServiceManager()); //1
    ALOGI("ServiceManager: %p", sm.get());
    InitializeIcuOrDie();
    MediaPlayerService::instantiate(); //2
    ResourceManagerService::instantiate();
    registerExtensions();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

MediaServer 需要向 **ServiceManager** 注册服务，在上面代码注释 1 处得到 **IServiceManager** 指针，这样我们就可以与另一个进程的 **ServiceManager** 进行通信。注释 2 处的代码用来初始化 **MediaPlayerService**。我们来查看 **MediaPlayerService** 的 **instantiate** 方法，如下所示：

frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp

```
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
        String16("media.player"), new MediaPlayerService());
}
```

上面的方法调用 **IServiceManager** 的 **addService** 方法，向 **ServiceManager** 添加一个名为 **media.player** 的 **MediaPlayerService** 服务。这样 **MediaPlayerService** 就被添加到 **ServiceManager** 中，**MediaPlayer** 就可以通过字符串 **"media.player"** 来查询 **ServiceManager**。那么，**MediaPlayer** 是在哪里进行查询的呢？让我们再回到 **mediaplayer** 的 **setDataSource** 方法，如下所示：

frameworks/av/media/libmedia/mediaplayer.cpp

```
status_t MediaPlayer::setDataSource(const sp<IDataSource> &source)
{
    ALOGV("setDataSource(IDataSource)");
    status_t err = UNKNOWN_ERROR;
    const sp<IMediaPlayerService> service(getMediaPlayerService()); //1
    if (service != 0) {
        sp<IMediaPlayer> player(service->create(this, mAudioSessionId)); //2
        if ((NO_ERROR != doSetRetransmitEndpoint(player)) ||
            (NO_ERROR != player->setDataSource(source))) {
            player.clear();
        }
    }
}
```

```

    }
    err = attachNewPlayer(player);
}
return err;
}

```

上面代码注释 1 处的 `getMediaPlayerService` 方法是在 `IMediaDeathNotifier` 中定义的，如下所示：

frameworks/av/media/libmedia/IMediaDeathNotifier.cpp

```

IMediaDeathNotifier::getMediaPlayerService()
{
    ...
    if (sMediaPlayerService == 0) {
        sp<IServiceManager> sm = defaultServiceManager(); //1
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.player")); //2
            if (binder != 0) {
                break;
            }
            ALOGW("Media player service not published, waiting...");
            usleep(500000); // 0.5 s
        } while (true);
    }
    ...
}

```

在上面代码注释 1 处得到 `IServiceManager` 指针，并在注释 2 处查询名称为 "media.player" 的服务。这样 `MediaPlayer` 就获取了 `MediaPlayerService` 的信息，通过信息就可以与在 `MediaServer` 进程中的 `MediaPlayerService` 进行通信了。当我们调用 `MediaPlayer` 的 `setDataSource` 方法时，会调用 `MediaPlayerService` 的 `setDataSource` 方法：

frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp

```

status_t MediaPlayerService::Client::setDataSource(
    const sp<IStreamSource> &source) {
    player_type playerType = MediaPlayerFactory::getPlayerType(this,
        source); //1
    sp<MediaPlayerBase> p = setDataSource_pre(playerType); //2
    if (p == NULL) {
        return NO_INIT;
    }
    setDataSource_post(p, p->setDataSource(source));
    return mStatus;
}

```

在上面代码注释 1 处调用了 `MediaPlayerFactory` 的 `getPlayerType` 方法，以获取播放器的类

型 `playerType`，接着调用 `setDataSource_pre`，如下所示：

frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp

```
sp<MediaPlayerBase> MediaPlayerService::Client::setDataSource_pre(
    player_type playerType)
{
    sp<MediaPlayerBase> p = createPlayer(playerType);
    ...
}
```

在 `setDataSource_pre` 方法中调用了 `createPlayer` 方法并传入 `playerType`，`createPlayer` 方法的代码如下所示：

frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp

```
sp<MediaPlayerBase> MediaPlayerService::Client::createPlayer(player_type
playerType)
{
    sp<MediaPlayerBase> p = mPlayer;
    if ((p != NULL) && (p->playerType() != playerType)) {
        ALOGV("delete player");
        p.clear();
    }
    if (p == NULL) {
        p = MediaPlayerFactory::createPlayer(playerType, this, notify,
        mPid); //1
    }
    if (p != NULL) {
        p->setUID(mUID);
    }
    return p;
}
```

在上面代码注释 1 处调用了 `MediaPlayerFactory` 的 `createPlayer` 方法，代码如下所示：

frameworks/av/media/libmediaplayerservice/MediaPlayerFactory.cpp

```
sp<MediaPlayerBase> MediaPlayerFactory::createPlayer(
    player_type playerType,
    void* cookie,
    notify_callback_f notifyFunc,
    pid_t pid) {
    sp<MediaPlayerBase> p;
    IFactory* factory;
    status_t init_result;
    ...
    p = factory->createPlayer(pid); //1
```

```

...
    return p;
}

```

在上面代码注释 1 处会调用 factory 的 createPlayer 方法来创建播放器。在 Android 6.0 中，MediaPlayerFactory 中有 3 个播放器 Factory，它们分别是 StagefrightPlayerFactory、NuPlayerFactory 和 TestPlayerFactory。其中，StagefrightPlayerFactory 用于创建 Stagefright 框架。Stagefright 框架在 Android 2.0 中被引入系统以替代庞大复杂的 OpenCore 框架。但是由于 Stagefright 存在严重漏洞，并被黑客所利用，因此在 Android 7.0 中，谷歌去掉了 StagefrightPlayerFactory，也就是 Stagefright 框架。默认的播放器使用谷歌自己研发的 NuPlayer 框架。NuPlayerFactory 的 createPlayer 方法如下所示：

frameworks/av/media/libmediaplayerservice/MediaPlayerFactory.cpp

```

virtual sp<MediaPlayerBase> createPlayer(pid_t pid) {
    return new NuPlayerDriver(pid);
}

```

createPlayer 方法中会创建一个 NuPlayerDriver 类，也就是 NuPlayer 框架。MediaPlayer 框架就讲到这里。本节所讲的 MediaPlayer 框架知识如图 11-7 所示。

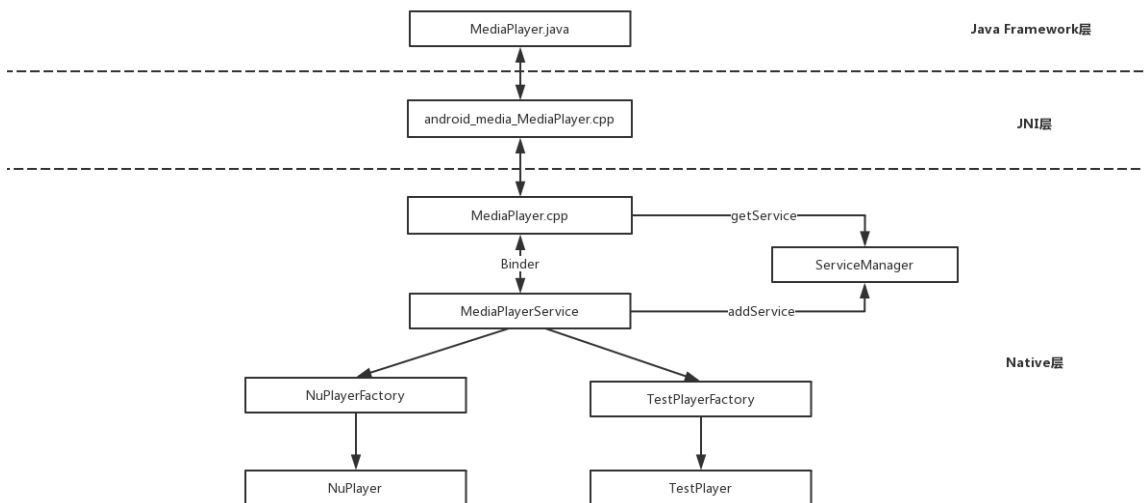


图 11-7 MediaPlayer 框架

从图 11-7 中可以得知，如果我们调用 MediaPlayer 的一个方法比如 setDataSource 方法，这个方法调用就会经过 JNI 层和 Native 层，一直到调用 NuPlayer 的 setDataSource 方法。当然，这只是 MediaPlayer 框架的调用过程，再往下调用还会涉及 NuPlayer 框架、Audio 系统、Surface 系统和 HAL 层等。如果想要深入了解这些框架和系统，或者是其他系统源码，那就从现在开始学习系统源码吧。

11.5 本章小结

本章介绍了 Android 系统架构、系统源码目录，并以 MediaPlayer 框架为例讲解了如何阅读系统底层源码。其主要用意就是拓展读者的思路，提醒读者 Android 并非只有 Android 应用层开发，一个优秀的 Android 开发者也不应仅仅满足做一个“API 小王子”，系统底层源码还等着我们去探索呢。鉴于篇幅有限，本书并没有过多介绍系统底层源码，对此感兴趣的读者可以阅读一些有关 Android 系统底层源码的专业书籍。

后记

到这里，本书就结束了。通过本书的学习，想必你对 Android 开发有了更加深入的理解，并且会成为一名优秀的 Android 开发者。由于篇幅有限，我还有很多想要讲的技术知识无法在本书中展现，这些技术知识我会继续在自己的博客和微信公众号上分享出来。如果感兴趣，你可以继续关注我的博客和微信公众号。我的独立博客：<http://liuwangshu.cn>，CSDN 博客：<http://blog.csdn.net/itachi85>。

也欢迎大家关注我的微信公众号，第一时间获得博客更新提醒及更多成体系的 Android 相关原创技术“干货”。

扫一扫下方二维码即可关注作者。



