Android Source

Design Patterns

Analysis & Practice

Android

Based on Android 5.0 Lollipop

源码设计模式

解析与实战

◆ 何红辉 关爱民 著

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

Android Source

Design Patterns

Analysis & Practice

本书分为三部分，第一部分详细介绍了面向对象六大原则，以具体的示例讲述了六大原则在开发中的实际运用，通过将代码一步一步重构为符合 OOP 原则的实现，使代码更具扩展性、灵活性；第二部分是分析 Android 源码中运用的 23 种经典设计模式，以及这些模式在 Android 开发中的实战运用，从而将所学知识运用于实际开发中；第三部分通过对 MVC、MVP 两种应用架构的阐述，以具体的示例演示了这些架构模式对于应用软件开发的影响。全书理论结合实际，以具体的示例让读者从代码的演化过程中体会设计原则、设计模式、架构对于软件系统的影响，帮助读者达到学以致用的目的。

本书不仅仅分析了 Android 源代码的设计模式，更结合实例演示了如何使用这些设计模式。看这本书，既能学到如何分析、学习 Android 源代码，又能提高自己的架构设计水平，实在是一本难得的好书！

——徐宜生《Android 群英传》作者

这本书内容独树一帜，将设计模式和 Android 源码中的设计思想融合到一起，使开发者受益良多。我一定会认真阅读这本书。

——任玉刚（singwhatiwanna）百度 Android 资深工程师，《Android 开发艺术探索》作者

当你做了许多年开发以后就会意识到，设计模式非常重要，阅读源码也非常重要，而这本书很好地将这两者结合到了一起，对于那些想要进阶的开发者们，这将是一本非常合适的书籍。

——郭霖《第一行代码：Android》作者和 LitePal 框架作者

封面设计：董志桢

分类建议：计算机／移动开发／ Android

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-40671-2

定价:79.00 元

Android Source

Design Patterns

Analysis & Practice

Android

Based on Android 5.0 Lollipop

源码设计模式

解析与实战

◆ 何红辉 关爱民 著

人民邮电出版社

北京

图书在版编目（CIP）数据

Android源码设计模式解析与实战 / 何红辉，关爱民著. -- 北京 : 人民邮电出版社，2015.11（2015.12重印）
ISBN 978-7-115-40671-2

Ⅰ. ①A… Ⅱ. ①何… ②关… Ⅲ. ①移动终端—应用程序—程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2015)第244188号

内 容 提 要

本书专门介绍 Android 源代码的设计模式，共 26 章，主要讲解面向对象的六大原则、主流的设计模式以及 MVC 和 MVP 模式。主要内容为：优化代码的第一步、开闭原则、里氏替换原则、依赖倒置原则、接口隔离原则、迪米特原则、单例模式、Builder 模式、原型模式、工厂方法模式、抽象工厂模式、策略模式、状态模式、责任链模式、解释器模式、命令模式、观察者模式、备忘录模式、迭代器模式、模板方法模式、访问者模式、中介者模式、代理模式、组合模式、适配器模式、装饰模式、享元模式、外观模式、桥接模式，以及 MVC 的介绍与实战和 MVP 应用架构模式。每个章节都对某个模式做了深入的分析，并且会对模式相关的技术点进行深入拓展，让读者在掌握模式的同时学习到 Android 中的一些重要知识，通过实战帮助读者达到学以致用的目的，且能够将模式运用于项目中，开发出高质量的程序。

本书适合的读者为初、中、高级 Android 工程师，也可以作为大专院校相关师生的学习用书和培训学校的教材。

◆ 著　何红辉　关爱民
责任编辑　张　涛
责任印制　张佳莹　焦志炜

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

◆ 开本：800×1000　1/16
印张：31.75
字数：441 千字　2015 年 11 月第 1 版
印数：4001-5500 册　2015 年 12 月北京第 3 次印刷

定价：79.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

推荐序

设计模式本身并不复杂，但是设计模式的出现，却是 GOF 大师们耗费无数心血，研究成百上千的例子，历经千锤百炼取其精华而得之，所以，它的重要性毋庸置疑。几年前，我曾见过高焕堂老师一本类似书籍的原稿，可惜此书未能出版，心中一直对此遗憾。但今天有幸一窥 CSDN 社区专家何红辉、关爱民老师精心撰写的这本以 Android 源码为案例的设计模式解析与实战一书时，激动之情勃然而发。是的，本书的确是国内第一本以 Android 为平台介绍设计模式的书，并且书中实例还不是简单的 Sample，而是作者在自己开发实践中的经历和一些实际使用的精彩代码段，实用性很强。

另外，我觉得《Android 源码设计模式解析与实战》中的主人公小民就是那些不断追求技术进步，从而得以不断成长的 IT 技术人的代表，小民的成长过程基本上反映了我们现在程序员的成长经历，他的成功很值得我们学习和借鉴。

学习设计模式，是程序员自我修炼、提升实力过程中必不可少的一关。读完此书的您或许已是设计模式的熟手，但我个人觉得，程序员的自我修炼远未结束，因为在设计模式之后，更有像 Pattern Oriented Software Architecture 5 卷本这样的、着眼于更高层次的书籍需要我们认真、刻苦地学习。希望何红辉和关爱民继续努力，将来为广大工程师提供层次更深、味道更丰富的精神食粮。

邓凡平

畅销书《深入理解 Android》系列的总策划和主笔，著有《深入理解 Android：卷 I》《深入理解 Android：卷 II》和《深入理解 Android：Wi-Fi，NFC 和 GPS 卷》。

自序一

想写一本 Android 设计模式的书的念头由来已久，也许是从我开始接触 Android 开发后就有了，于是很早就在自己的记事本上记录了一些相关学习心得。2014 年 4 月我就在博客上连载了《Android 源码分析之设计模式》系列，简单分析 Android 源码中的一些设计模式。到了 2014 年年底开始写一些开发框架相关的博客，并且在此期间发布了 AndroidEventBus 开源库，此后就一直活跃于 Github、博客圈。2015 年 3 月，我开始在 Github 上创建 Android 源码设计模式分析的开源项目，借助开源力量在一个月之内发布了十多篇 Android 源码中设计模式分析的文章，一经发布便得到了业界的普遍好评。

这些文章得到了业界的认可，让我又想起了最初出书的念头。原因很简单，Android 是一个开源的系统，很多优秀的思想、架构、设计模式必然在它的源码中得以体现，而在开源社区发布的文章还不够深入。从学习“Hello World”开始，我们都是先从学习他人如何做，然后再到学着做，最后经过自己的理解与思考再到自己做，因此，学习这些优秀的实现正是我们每个开发人员成长过程中的重要一步。在学习 Android 源码的优秀设计之后，我们如何将设计模式运用在 Android 开发上成了至关重要的问题，正所谓学以致用。因此，设计模式在 Android 开发中的实战又成了第二个关键。恰好，这两个领域目前都没有相关的书籍，我和关爱民老师就考虑出版这样的一本书籍。一来是通过写书实现自我提升以及对知识的梳理，二来也希望本书能够让更多的 Android 开发人员了解设计模式，从而提高自己的代码质量。如此一来，也算是尽了我们的绵薄之力。

何红辉

于北京

自序二

很多 Android 源码的实现都有设计模式的影子，对于很多从事 Android 开发的读者来说，学习 Android 源码的最大障碍往往是对其设计思想的理解而非源码本身，很多时候我们能看懂一段源码但却不明白其思想，看懂的是这一段源码的实现逻辑而不懂的则是为什么逻辑会是这样，对于开发者来说，知其然却又不知其所以然往往是进阶中最大的阻力。在今年早些时候，何红辉老师找到我说想构思一本关于 Android 源码设计模式的书，而此时的我刚好在深究 Android 源码的一些架构设计。市面上有很多关于 Android 基础方面的书籍，这些书籍帮助很多开发者入门并走向 Android 的开发殿堂，而对 Android 内核解析方面的书籍和资料也有不少，我曾经也阅读过很多关于 Android 内核的书籍，这些书籍对我从了解到深入内核层面有很大的帮助，但是，像一些基于设计和架构方面的 Android 书籍却很少看到，基于这样一个契机，我俩一拍即合决定写一本这样的书来帮助一些开发者进阶提升。我是一个喜欢分享、喜欢传授且不拘一束的人，借此机会，我想将我的一些经验或方法通过此书分享给大家，希望大家在今后的开发道路上少走一些弯路。

写书是需要付出极大的努力，且远比想象中困难，特别是在开发技术飞速发展的今天想要跟上技术迭代的脚步极不容易，在这里我想先感谢阅读本书的读者们，因为你们的认可才给予我们最大的写作动力；其次是帮我审稿、阅稿，以及在我学习过程中不断指点我的李志豪、常正宇和刘峰前辈，还有人民邮电出版社的张涛编辑，在出版的过程中给予我们很多建议和教导；再次要感谢的是一起写书的搭档何红辉老师，因为他的想法才有本书的诞生，并且在百忙之中还帮助我阅稿并指正错误；最后要感谢的是我的家人和朋友，因为你们的理解才能让我在几个月的时间中集中精力写好本书。

关爱民

于重庆

前　言

本书的特色

本书的第一部分是面向对象六大原则，通过具体的示例讲解六大原则的定义与作用，以及遵循这些原则会存在什么问题，遵循这些原则又会得到什么好处。通过第一部分使得读者对于面向接口编程以及 OOP 的基本原则有一个深入的认识。

第二部分就是本书的核心部分，每个章节分析一个设计模式，每个模式分为如下几个部分：

（1）模式基本介绍，介绍模式的定义、使用场景、UML 类图，使读者对相关模式有一个大致的认识；

（2）模式的简单实现，该模式的经典实现，使读者从代码的角度进一步理解相关模式；

（3）Android 源码中的模式实现，模式在 Android 源码中的运用与分析；

（4）深入拓展，进一步了解运用该模式模块的核心机制，使得读者彻底了解 Android 的一些基本原理；

（5）模式实战，通过一个示例让读者学习到如何将该模式运用于 Android 开发中；

（6）总结，总结该模式的优、缺点。

每个章节都对某个模式做了深入的分析，并且会对与模式相关的技术点进行深入拓展，让读者在掌握模式的同时学习到 Android 中的一些重要知识，通过实战则使读者达到学以致用的效果，能够将模式运用于开发当中。

本书的第三部分则是简单介绍了 MVC 及 MVP 模式，使读者从更高的应用架构角度看待应用开发，从整体上把握应用的基本结构。

本书涵盖了基本的 OOP 原则、设计模式的详细介绍、分析、实战，使得读者能够通过设计模式解决一些局部的设计问题，从而达到可扩展、灵活的局部架构。最后的 MVC 与 MVP 则从架构的高度帮助读者避免出现臃肿的类型、紧耦合等问题，使得应用真正实现高内聚、低耦合的应用架构。

面向的读者

在行业内很多初、中级工程师甚至高级工程师由于某些原因都还停留在功能实现层面，甚至对设计模式、面向对象知之甚少，因此，很少考虑代码的设计问题。本书从源码的角度由浅入深地剖析设计模式的运用，让工程师们把设计与模式重视起来，提升自己的设计能力与代码质量。因此，本书适合的读者为初、中、高级 Android 工程师。另外，本书强调的是编程思想，而思想都是相通的，因此，其他开发领域的工程师也能从中受益。

本书并不是一本 Android 开发或者设计模式的入门书籍，因此，阅读本书时你最好掌握了 Android 以及设计模式的相关知识。如果还没有，那么《第一行代码》和《Android 开发艺术探索》都是 Android 领域的优秀书籍；对于设计模式而言，《设计模式之禅》和《设计模式：可复用面向对象软件的基础》都是很好的选择。

如何阅读本书

本书分为 3 部分，分别为面向对象六大原则、23 种设计模式解析、MVC 与 MVP。其中第二部分的各个章节之间没有相关的联系，因此读者可以从中选择自己感兴趣的章节进行阅读。初、中级工程师建议至少阅读第一部分、第二部分的常用模式以及第三部分，高级工程师可以选择自己感兴趣的部分进行阅读。判定你是否需要阅读某个章节的标准是，当你看到标题时是否对这个知识点了然于心，如果答案是否定的，那么阅读该章节还是很有必要的。当然，通读全书自然是最好的选择。

最后需要说明的一点是，编写任何一本书籍都难免会有一些错误或不准确甚至不正确的地方，我们很乐意听到读者对我们的意见或建议，您可以通过发邮件（邮箱地址：simplecoder.h@gmail.com）的方式向我们反馈，在这里致以我们诚挚的谢意。编辑联系邮箱为：zhangtao@ptpress.com.cn。

编者

致　谢

任何一本书的诞生都不是某个个人努力的产物，本书也一样。经过数月的笔耕不辍，本书最终得以出版，在这期间有很多人为此付出了自己的辛勤劳动。

首先要感谢我的搭档关爱民老师，他不仅分担了本书一半的写作任务，而且在写作的过程中任劳任怨，挤出一切闲暇时间以保证按时甚至提前完成写作，只为能让本书尽快出版。其次是要感谢张涛编辑，在本书的出版过程中给了我们很多鼓励，并且为本书的出版付出了很多的努力。还要感谢秦汉、潘超群（Joker）两位前辈在百忙之中抽出时间帮助审稿，保证了本书的质量。最后要感谢我的家人，在我写作的时候给我建议并帮我校稿，在整个过程中给予我很大的支持。

何红辉

2015 年 8 月 4 日于北京

写书是需要付出极大的努力与勇气的，且远比想象中困难，特别是在开发技术飞速发展的今天，想要跟上技术迭代的脚步极不容易。在这里我想先感谢阅读本书的读者们，因为你们的认可才能给予我们最大的写作动力；其次是帮我审稿以及在我学习过程中不断指点我的李志豪、常正宇和刘峰前辈；人民邮电出版社的张涛编辑，在出版的过程中给予我们很多建议和帮助；再次要感谢的是一起写书的搭档何红辉老师，因为他的想法才有了本书的诞生，并且在百忙之中还帮助我阅稿、指正错误；最后要感谢我的家人和朋友，因为你们的理解才能让我在几个月的时间中集中精力写好本书。

关爱民

2015 年 8 月 4 日于重庆

目　录

第 1 章　走向灵活软件之路——面向对象的六大原则……1

1.1　优化代码的第一步——单一职责原则……1
1.2　让程序更稳定、更灵活——开闭原则……5
1.3　构建扩展性更好的系统——里氏替换原则……12
1.4　让项目拥有变化的能力——依赖倒置原则……13
1.5　系统有更高的灵活性——接口隔离原则……16
1.6　更好的可扩展性——迪米特原则……18
1.7　总结……22

第 2 章　应用最广的模式——单例模式……23

2.1　单例模式介绍……23
2.2　单例模式的定义……23
2.3　单例模式的使用场景……23
2.4　单例模式 UML 类图……23
2.5　单例模式的简单示例……24
2.6　单例模式的其他实现方式……26
2.6.1　懒汉模式……26
2.6.2　Double Check Lock (DCL) 实现单例……26
2.6.3　静态内部类单例模式……27
2.6.4　枚举单例……28
2.6.5　使用容器实现单例模式……28
2.7　Android 源码中的单例模式……29
2.8　无名英雄——深入理解 LayoutInflater……33
2.9　运用单例模式……40
2.10　总结……42

第 3 章　自由扩展你的项目——Builder 模式……43

3.1　Builder 模式介绍……43
3.2　Builder 模式的定义……43
3.3　Builder 模式的使用场景……43
3.4　Builder 模式的 UML 类图……43
3.5　Builder 模式的简单实现……44
3.6　Android 源码中的 Builder 模式实现……46
3.7　深入了解 WindowManager……52
3.8　Builder 模式实战……59
3.9　总结……64

第 4 章　使程序运行更高效——原型模式……66

4.1　原型模式介绍……66
4.2　原型模式的定义……66
4.3　原型模式的使用场景……66
4.4　原型模式的 UML 类图……66
4.5　原型模式的简单实现……67
4.6　浅拷贝和深拷贝……69
4.7　Android 源码中的原型模式实现……72
4.8　Intent 的查找与匹配……74
4.8.1　App 信息表的构建……74
4.8.2　精确匹配……80
4.9　原型模式实战……83
4.10　总结……85

第 5 章　应用最广泛的模式——工厂方法模式……87

5.1　工厂方法模式介绍……87
5.2　工厂方法模式的定义……87
5.3　工厂方法模式的使用场景……87
5.4　工厂方法模式的 UML 类图……87
5.5　模式的简单实现……90

5.6 Android 源码中的工厂方法模式实现 ····93
5.7 关于 onCreate 方法 ····95
5.8 工厂方法模式实战 ····102
5.9 总结 ····105

第 6 章 创建型设计模式——抽象工厂模式 ····106

6.1 抽象工厂模式介绍 ····106
6.2 抽象工厂模式的定义 ····106
6.3 抽象工厂模式的使用场景 ····106
6.4 抽象工厂模式的 UML 类图 ····106
6.5 抽象工厂方法模式的简单实现 ····109
6.6 Android 源码中的抽象工厂方法模式实现 ····112
6.7 总结 ····116

第 7 章 时势造英雄——策略模式 ····117

7.1 策略模式介绍 ····117
7.2 策略模式的定义 ····117
7.3 策略模式的使用场景 ····117
7.4 策略模式的 UML 类图 ····118
7.5 策略模式的简单实现 ····118
7.6 Android 源码中的策略模式实现 ····123
7.6.1 时间插值器 ····123
7.6.2 动画中的时间插值器 ····124
7.7 深入属性动画 ····128
7.7.1 属性动画体系的总体设计 ····128
7.7.2 属性动画的核心类介绍 ····128
7.7.3 基本使用 ····129
7.7.4 流程图 ····130
7.7.5 详细设计 ····131
7.7.6 核心原理分析 ····131
7.8 策略模式实战应用 ····142
7.9 总结 ····144

第 8 章 随遇而安——状态模式 ····145

8.1 状态模式介绍 ····145
8.2 状态模式的定义 ····145
8.3 状态模式的使用场景 ····145
8.4 状态模式的 UML 类图 ····145
8.5 状态模式的简单示例 ····146
8.6 Wi-Fi 管理中的状态模式 ····150
8.7 状态模式实战 ····159
8.8 总结 ····164

第 9 章 使编程更有灵活性——责任链模式 ····165

9.1 责任链模式介绍 ····165
9.2 责任链模式的定义 ····165
9.3 责任链模式的使用场景 ····165
9.4 责任链模式的 UML 类图 ····165
9.5 责任链模式的简单实现 ····170
9.6 Android 源码中的责任链模式实现 ····173
9.7 责任链模式实战 ····178
9.8 总结 ····181

第 10 章 化繁为简的翻译机——解释器模式 ····182

10.1 解释器模式介绍 ····182
10.2 解释器模式的定义 ····182
10.3 解释器模式的使用场景 ····183
10.4 解释器模式的 UML 类图 ····184
10.5 解释器模式的简单实现 ····185
10.6 Android 源码中的解释器模式实现 ····189
10.7 关于 PackageManagerService ····195
10.8 总结 ····203

第 11 章 让程序畅通执行——命令模式 ····204

11.1 命令模式介绍 ····204
11.2 命令模式的定义 ····204
11.3 命令模式的使用场景 ····204
11.4 命令模式的 UML 类图 ····204
11.5 命令模式的简单实现 ····207
11.6 Android 源码中的命令模式实现 ····211
11.7 Android 事件输入系统介绍 ····214
11.8 命令模式实战 ····216
11.9 总结 ····223

第 12 章 解决、解耦的钥匙——观察者模式 ····224

12.1 观察者模式介绍 ····224
12.2 观察者模式的定义 ····224

12.3 观察者模式的使用场景……224
12.4 观察者模式的 UML 类图……224
12.5 观察者模式的简单实现……225
12.6 Android 源码分析……227
12.7 观察者模式的深入拓展……230
12.8 实战……238
12.9 总结……245

第 13 章 编程中的“后悔药”——备忘录模式……247

13.1 备忘录模式介绍……247
13.2 备忘录模式的定义……247
13.3 备忘录模式的使用场景……247
13.4 备忘录模式的 UML 类图……247
13.5 备忘录模式的简单示例……248
13.6 Android 源码中的备忘录模式……250
13.7 深度拓展……257
13.7.1 onSaveInstanceState 调用的时机……257
13.7.2 使用 V4 包存储状态的 bug……257
13.8 实战……260
13.9 总结……267

第 14 章 解决问题的“第三者”——迭代器模式……268

14.1 迭代器模式介绍……268
14.2 迭代器模式的定义……268
14.3 迭代器模式的使用场景……268
14.4 迭代器模式的 UML 类图……268
14.5 模式的简单实现……271
14.6 Android 源码中的模式实现……275
14.7 总结……277

第 15 章 抓住问题核心——模板方法模式……278

15.1 模板方法模式介绍……278
15.2 模板方法模式的定义……278
15.3 模板方法模式的使用场景……278
15.4 模板方法模式的 UML 类图……278
15.5 模板方法模式的简单示例……279
15.6 Android 源码中的模板方法模式……281
15.7 深度拓展……283
15.8 模板方法实战……296
15.9 总结……299

第 16 章 访问者模式……301

16.1 访问者模式介绍……301
16.2 访问者模式的定义……301
16.3 访问者模式的使用场景……301
16.4 访问者模式的 UML 类图……301
16.5 访问者模式的简单示例……302
16.6 Android 源码中的模式……306
16.7 访问者模式实战……309
16.8 总结……316

第 17 章 “和事佬”——中介者模式……317

17.1 中介者模式介绍……317
17.2 中介者模式的定义……318
17.3 中介者模式的使用场景……318
17.4 中介者模式的 UML 类图……318
17.5 中介者模式的简单实现……320
17.6 Android 源码中的中介者模式实现……324
17.7 中介者模式实战……326
17.8 总结……329

第 18 章 编程好帮手——代理模式……330

18.1 代理模式介绍……330
18.2 代理模式的定义……330
18.3 代理模式的使用场景……330
18.4 代理模式的 UML 类图……330
18.5 代理模式的简单实现……332
18.6 Android 源码中的代理模式实现……336
18.7 Android 中的 Binder 跨进程通信机制与 AIDL……340
18.8 代理模式实战……351
18.9 总结……355

第 19 章 物以类聚——组合模式……356

19.1 组合模式介绍……356
19.2 组合模式的定义……357
19.3 组合模式的使用场景……357
19.4 组合模式的 UML 类图……357
19.5 组合模式的简单实现……363

19.6 Android 源码中的模式实现 ……367
19.7 为什么 ViewGroup 有容器的功能 ……368
19.8 总结 ……370
第 20 章 得心应手的“粘合剂”——适配器模式 ……371
20.1 适配器模式介绍 ……371
20.2 适配器模式的定义 ……371
20.3 适配器模式的使用场景 ……371
20.4 适配器模式的 UML 类图 ……371
20.5 适配器模式应用的简单示例 ……372
20.5.1 类适配器模式 ……372
20.5.2 对象适配器模式 ……373
20.6 Android 源码中的适配器模式 ……375
20.7 深度拓展 ……380
20.8 实战演示 ……393
20.9 总结 ……402
第 21 章 装饰模式 ……403
21.1 装饰模式介绍 ……403
21.2 装饰模式的定义 ……403
21.3 装饰模式的使用场景 ……403
21.4 装饰模式的 UML 类图 ……403
21.5 模式的简单实现 ……406
21.6 Android 源码中的模式实现 ……408
21.7 Context 与 ContextImpl ……410
21.8 模式实战 ……419
21.9 总结 ……419
第 22 章 对象共享，避免创建多对象——享元模式 ……420
22.1 享元模式介绍 ……420
22.2 享元模式定义 ……420
22.3 享元模式的使用场景 ……420
22.4 享元模式的 UML 类图 ……420
22.5 享元模式的简单示例 ……421
22.6 Android 源码中的享元模式 ……424
22.7 深度拓展 ……429
22.7.1 深入了解 Android 的消息机制 ……429
22.7.2 子线程中创建 Handler 为何会抛出异常 ……438
22.8 总结 ……439
第 23 章 统一编程接口——外观模式 ……440
23.1 外观模式介绍 ……440
23.2 外观模式定义 ……440
23.3 外观模式的使用场景 ……440
23.4 外观模式的 UML 类图 ……440
23.5 外观模式的简单示例 ……441
23.6 Android 源码中的外观模式 ……443
23.7 深度拓展 ……448
23.7.1 Android 资源的加载与匹配 ……448
23.7.2 动态加载框架的实现 ……455
23.8 外观模式实战 ……461
23.9 总结 ……464
第 24 章 连接两地的交通枢纽——桥接模式 ……465
24.1 桥接模式介绍 ……465
24.2 桥接模式的定义 ……465
24.3 桥接模式的使用场景 ……465
24.4 桥接模式的 UML 类图 ……465
24.5 桥接模式的简单实现 ……467
24.6 Android 源码中的桥接模式实现 ……470
24.7 关于 WindowManagerService ……471
24.8 桥接模式实战 ……479
24.9 总结 ……482
第 25 章 MVC 的介绍与实战 ……483
25.1 MVC 的起源与历史 ……483
25.2 MVC 在 Android 中的实现 ……484
25.3 总结 ……486
第 26 章 MVP 应用架构模式 ……487
26.1 MVP 模式介绍 ……487
26.2 MVP 模式的三个角色 ……488
26.3 与 MVC、MVVM 的区别 ……488
26.4 MVP 的实现 ……489
26.5 MVP 与 Activity、Fragment 的生命周期 ……493

第 1 章　走向灵活软件之路——面向对象的六大原则

1.1 优化代码的第一步——单一职责原则

单一职责原则的英文名称是 Single Responsibility Principle，缩写是 SRP。SRP 的定义是：就一个类而言，应该仅有一个引起它变化的原因。简单来说，一个类中应该是一组相关性很高的函数、数据的封装。就像秦小波老师在《设计模式之禅》中说的："这是一个备受争议却又及其重要的原则。只要你想和别人争执、怄气或者是吵架，这个原则是屡试不爽的"。因为单一职责的划分界限并不是总是那么清晰，很多时候都是需要靠个人经验来界定。当然，最大的问题就是对职责的定义，什么是类的职责，以及怎么划分类的职责。

对于计算机技术，通常只单纯地学习理论知识并不能很好地领会其深意，只有自己动手实践，并在实际运用中发现问题、解决问题、思考问题，才能够将知识吸收到自己的脑海中。下面以我的朋友小民的事情说起。

自从 Android 系统发布以来，小民就是 Android 的铁杆粉丝，于是在大学期间一直保持着对 Android 的关注，并且利用课余时间做些小项目，锻炼自己的实战能力。毕业后，小民如愿地加入了心仪的公司，并且投入到了他热爱的 Android 应用开发行业中。将爱好、生活、事业融为一体，小民的第一份工作也算是顺风顺水，一切尽在掌握中。

在经历过一周的适应期以及熟悉公司的产品、开发规范之后，小民的开发工作就正式开始了。小民的主管是个工作经验丰富的技术专家，对于小民的工作并不是很满意，尤其小民最薄弱的面向对象设计，而 Android 开发又是使用 Java 语言，程序中的抽象、接口、六大原则、23 种设计模式等名词把小民弄得晕头转向。小民自己也察觉到了自己的问题所在，于是，小民的主管决定先让小民做一个小项目来锻炼这方面的能力。正所谓养兵千日用兵一时，磨刀不误砍柴工，小民的开发之路才刚刚开始。

在经过一番思考之后，主管挑选了使用范围广、难度也适中的图片加载器（ImageLoader）作为小民的训练项目。既然要训练小民的面向对象设计，那么就必须考虑到可扩展性、灵活性，而检测这一切是否符合需求的最好途径就是开源。用户不断地提出需求、反馈问题，小民的项目需要不断升级以满足用户需求，并且要保证系统的稳定性、灵活性。在主管跟小民说了这一特殊任务之后，小民第一次感到了压力，"生活不容易！"年仅 22 岁的小民发出了如此深刻的感叹！

挑战总是要面对的，何况是从来不服输的小民。主管的要求很简单，要小民实现图片加载，并

且要将图片缓存起来。在分析了需求之后，小民一下就放心下来了，“这么简单，原来我还以为很难呢……”小民胸有成足地喃喃自语。在经历了 10 分钟的编码之后，小民写下了如下代码：

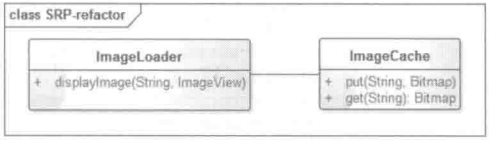
```
/**
 * 图片加载类
 */
public class ImageLoader {
    // 图片缓存
    LruCache<String, Bitmap> mImageCache;
    // 线程池,线程数量为 CPU 的数量
    ExecutorService mExecutorService = Executors.newFixedThreadPool (Runtime.
    getRuntime().availableProcessors());

    public ImageLoader() {
        initImageCache();
    }

    private void initImageCache() {
            // 计算可使用的最大内存
        final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);
            // 取四分之一的可用内存作为缓存
        final int cacheSize = maxMemory / 4;
        mImageCache = new LruCache<String, Bitmap>(cacheSize) {

            @Override
            protected int sizeOf(String key, Bitmap bitmap) {
                return bitmap.getRowBytes() * bitmap.getHeight() / 1024;
            }
        };
    }

    public void displayImage(final String url, final ImageView imageView) {
        imageView.setTag(url);
        mExecutorService.submit(new Runnable() {

            @Override
            public void run() {
                Bitmap bitmap = downloadImage(url);
                if (bitmap == null) {
                    return;
                }
                if (imageView.getTag().equals(url)) {
                    imageView.setImageBitmap(bitmap);
                }
                mImageCache.put(url, bitmap);
            }
        });
    }

    public Bitmap downloadImage(String imageUrl) {
        Bitmap bitmap = null;
        try {
            URL url = new URL(imageUrl);
            final HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            bitmap = BitmapFactory.decodeStream(conn.getInputStream());
            conn.disconnect();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        return bitmap;
    }
}
```

并且使用 Git 软件进行版本控制，将工程托管到 Github 上，伴随着 git push 命令的完成，小民的 ImageLoader 0.1 版本就正式发布了！如此短的时间内就完成了这个任务，而且还是一个开源项目，小民暗暗自喜，并幻想着待会儿被主管称赞。

在小民给主管报告了 ImageLoader 的发布消息的几分钟之后，主管就把小民叫到了会议室。这下小民纳闷了，怎么夸人还需要到会议室。“小民，你的 ImageLoader 耦合太严重啦！简直就没有设计可言，更不要说扩展性、灵活性了。所有的功能都写在一个类里怎么行呢，这样随着功能的增多，ImageLoader 类会越来越大，代码也越来越复杂，图片加载系统就越来越脆弱……”这简直就是当头棒喝，小民的脑海里已经听不清主管下面说的内容了，只是觉得自己之前没有考虑清楚就匆匆忙忙完成任务，而且把任务想得太简单了。

“你还是把 ImageLoader 拆分一下，把各个功能独立出来，让它们满足单一职责原则。”主管最后说道。小民是个聪明人，敏锐地捕捉到了单一职责原则这个关键词，他用 Google 搜索了一些资料之后，总算是对单一职责原则有了一些认识，于是打算对 ImageLoader 进行一次重构。这次小民不敢过于草率，也是先画了一幅 UML 图，如图 1-1 所示。



▲图 1-1

ImageLoader 代码修改如下所示：

```
/**
 * 图片加载类
 */
public class ImageLoader {
    // 图片缓存
    ImageCache mImageCache = new ImageCache() ;
    // 线程池,线程数量为 CPU 的数量
    ExecutorService mExecutorService = Executors.newFixedThreadPool (Runtime.
    getRuntime().availableProcessors());

    //加载图片
    public void displayImage(final String url, final ImageView imageView) {
        Bitmap bitmap = mImageCache.get(url);
        if (bitmap != null) {
            imageView.setImageBitmap(bitmap);
            return;
        }
        imageView.setTag(url);
        mExecutorService.submit(new Runnable() {

            @Override
            public void run() {
            Bitmap bitmap = downloadImage(url);
                if (bitmap == null) {
                    return;
                }
                if (imageView.getTag().equals(url)) {
                    imageView.setImageBitmap(bitmap);
                }
                mImageCache.put(url, bitmap);
            }
        });
    }
```

```
    public  Bitmap downloadImage(String imageUrl) {
        Bitmap bitmap = null;
        try {
            URL url = new URL(imageUrl);
            final HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            bitmap = BitmapFactory.decodeStream(conn.getInputStream());
            conn.disconnect();
        } catch (Exception e) {
            e.printStackTrace();
        }

        return bitmap;
    }
}
```

并且添加了一个 ImageCache 类用于处理图片缓存，具体代码如下：

```
public  class ImageCache {
    // 图片 LRU 缓存
    LruCache<String, Bitmap>mImageCache;

    public  ImageCache() {
        initImageCache();
    }

    private void initImageCache() {
        // 计算可使用的最大内存
        final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);
        // 取四分之一的可用内存作为缓存
        final int cacheSize = maxMemory / 4;
        mImageCache = new LruCache<String, Bitmap>(cacheSize) {

            @Override
            protected int sizeOf(String key, Bitmap bitmap) {
                return bitmap.getRowBytes() * bitmap.getHeight() / 1024;
            }
        };
    }

    public  void put(String url, Bitmap bitmap) {
        mImageCache.put(url, bitmap) ;
    }

    public  Bitmap get(String url) {
        return mImageCache.get(url) ;
    }
}
```

如图 1-1 和上述代码所示，小民将 ImageLoader 一拆为二，ImageLoader 只负责图片加载的逻辑，而 ImageCache 只负责处理图片缓存的逻辑，这样 ImageLoader 的代码量变少了，职责也清晰了；当与缓存相关的逻辑需要改变时，不需要修改 ImageLoader 类，而图片加载的逻辑需要修改时也不会影响到缓存处理逻辑。主管在审核了小民的第一次重构之后，对小民的工作给予了表扬，大致意思是结构变得清晰了许多，但是可扩展性还是比较欠缺。虽然没有得到主管的完全肯定，但也是颇有进步，再考虑到自己确实有所收获，小民原本沮丧的心里也略微地好转起来。

从上述的例子中我们能够体会到，单一职责所表达出的用意就是“单一”二字。正如上文所说，如何划分一个类、一个函数的职责，每个人都有自己的看法，这需要根据个人经验、具体的业务逻

辑而定。但是，它也有一些基本的指导原则，例如，两个完全不一样的功能就不应该放在一个类中。一个类中应该是一组相关性很高的函数、数据的封装。工程师可以不断地审视自己的代码，根据具体的业务、功能对类进行相应的拆分，这是程序员优化代码迈出的第一步。

1.2 让程序更稳定、更灵活——开闭原则

开闭原则的英文全称是 Open Close Principle，缩写是 OCP，它是 Java 世界里最基础的设计原则，它指导我们如何建立一个稳定的、灵活的系统。开闭原则的定义是：软件中的对象（类、模块、函数等）应该对于扩展是开放的，但是，对于修改是封闭的。在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会将错误引入原本已经经过测试的旧代码中，破坏原有系统。因此，当软件需要变化时，我们应该尽量通过扩展的方式来实现变化，而不是通过修改已有的代码来实现。当然，在现实开发中，只通过继承的方式来升级、维护原有系统只是一个理想化的愿景，因此，在实际的开发过程中，修改原有代码、扩展代码往往是同时存在的。

软件开发过程中，最不会变化的就是变化本身。产品需要不断地升级、维护，没有一个产品从第一版本开发完就再没有变化了，除非在下个版本诞生之前它已经被终止。而产品需要升级，修改原来的代码就可能会引发其他的问题。那么，如何确保原有软件模块的正确性，以及尽量少地影响原有模块，答案就是，尽量遵守本章讲述的开闭原则。

勃兰特·梅耶在 1988 年出版的《面向对象软件构造》一书中提出这一原则——开闭原则。这一想法认为，程序一旦开发完成，程序中一个类的实现只应该因错误而被修改，新的或者改变的特性应该通过新建不同的类实现，新建的类可以通过继承的方式来重用原类的代码。显然，梅耶的定义提倡实现继承，已存在的实现类对于修改是封闭的，但是新的实现类可以通过覆写父类的接口应对变化。

说了这么多，想必大家还是半懂不懂，还是让我们以一个简单示例说明一下。

在对 ImageLoader 进行了一次重构之后，小民的这个开源库获得了一些用户，小民第一次感受成功的快乐，对开源的热情也越发高涨起来！通过动手实现一些开源库来深入学习相关技术，不仅能够提升自我，也能更好地将这些技术运用到工作中，从而开发出更稳定、更优秀的应用，这就是小民的真实想法。

小民第一轮重构之后的 ImageLoader 职责单一、结构清晰，不仅获得了主管的一点肯定，还得到了用户的夸奖，算是个不错的开始。随着用户的增多，有些问题也暴露出来了，小民的缓存系统就是大家“吐槽”最多的地方。通过内存缓存解决了每次从网络加载图片的问题，但是，Android 应用的内存很有限，且具有易失性，即当应用重新启动之后，原来已经加载过的图片将会丢失，这样重启之后就需要重新下载！这又会导致加载缓慢、耗费用户流量的问题。小民考虑引入 SD 卡缓存，这样下载过的图片就会缓存到本地，即使重启应用也不需要重新下载了。小民在和主管讨论了该问题之后就投入了编程中，下面就是小民的代码。

DiskCache.java 类，将图片缓存到 SD 卡中：

```
public class DiskCache {
    static String cacheDir = "sdcard/cache/";
```

```
    // 从缓存中获取图片
    public  Bitmap get(String url) {
        return  BitmapFactory.decodeFile(cacheDir + url);
    }

    // 将图片缓存到内存中
    public  void  put(String url, Bitmap bmp) {
        FileOutputStream fileOutputStream = null;
          try {
              fileOutputStream = new FileOutputStream(cacheDir + url);
              bmp.compress(CompressFormat.PNG, 100, fileOutputStream);
        } catch (FileNotFoundException e) {
              e.printStackTrace();
        } final ly {
              if (fileOutputStream != null) {
                    try {
                        fileOutputStream.close();
                  } catch (IOException e) {
                        e.printStackTrace();
                }
            }
        }
    }
}
```

因为需要将图片缓存到 SD 卡中，所以，ImageLoader 代码有所更新，具体代码如下：

```
public  class ImageLoader {
    // 内存缓存
    ImageCache mImageCache = new ImageCache();
    // SD 卡缓存
    DiskCache mDiskCache = new DiskCache();
    // 是否使用 SD 卡缓存
    boolean isUseDiskCache = false;
    // 线程池,线程数量为 CPU 的数量
    ExecutorService mExecutorService = Executors.newFixedThreadPool (Runtime.
    getRuntime().availableProcessors());


    public  void displayImage(final String url, final ImageView imageView) {
         // 判断使用哪种缓存
        Bitmap bitmap = isUseDiskCache ? mDiskCache.get(url) : mImageCache.get (url);
         if (bitmap != null) {
             imageView.setImageBitmap(bitmap);
             return;
         }
          // 没有缓存，则提交给线程池进行下载
    }

    public  void useDiskCache(boolean useDiskCache) {
        isUseDiskCache = useDiskCache ;
    }
}
```

从上述的代码中可以看到，仅仅新增了一个 DiskCache 类和往 ImageLoader 类中加入了少量代码就添加了 SD 卡缓存的功能，用户可以通过 useDiskCache 方法来对使用哪种缓存进行设置，例如：

```
ImageLoader imageLoader = new ImageLoader() ;
 // 使用 SD 卡缓存
imageLoader.useDiskCache(true);
// 使用内存缓存
imageLoader.useDiskCache(false);
```

通过 useDiskCache 方法可以让用户设置不同的缓存，非常方便啊！小民对此很满意，于是提交给主管做代码审核。“小民，你思路是对的，但是有些明显的问题，就是使用内存缓存时用户就不能使用 SD 卡缓存。类似地，使用 SD 卡缓存时用户就不能使用内存缓存。用户需要这两种策略的综合，首先缓存优先使用内存缓存，如果内存缓存没有图片再使用 SD 卡缓存，如果 SD 卡中也没有图片最后才从网络上获取，这才是最好的缓存策略。”主管的解释真是一针见血，小民这时才如梦初醒，刚才还得意洋洋的脸上突然有些泛红……

于是小民按照主管的指点新建了一个双缓存类 DoubleCache，具体代码如下：

```
/**
 * 双缓存。获取图片时先从内存缓存中获取，如果内存中没有缓存该图片，再从 SD 卡中获取。
 *  缓存图片也是在内存和 SD 卡中都缓存一份
 */
public class DoubleCache {
    ImageCache mMemoryCache = new ImageCache();
    DiskCache mDiskCache = new DiskCache();

    // 先从内存缓存中获取图片，如果没有，再从 SD 卡中获取
    public  Bitmap get(String url) {
        Bitmap bitmap = mMemoryCache.get(url);
         if (bitmap == null) {
            bitmap = mDiskCache.get(url);
        }
         return bitmap;
    }

    // 将图片缓存到内存和 SD 卡中
    public void put(String url, Bitmap bmp) {
        mMemoryCache.put(url, bmp);
        mDiskCache.put(url, bmp);
    }
}
```

我们再看看最新的 ImageLoader 类，代码更新也不多：

```
public class ImageLoader {
    // 内存缓存
    ImageCache mImageCache = new ImageCache();
    // SD 卡缓存
    DiskCache mDiskCache = new DiskCache();
    // 双缓存
    DoubleCache mDoubleCache = new DoubleCache() ;
    // 使用 SD 卡缓存
    boolean isUseDiskCache = false;
    // 使用双缓存
    boolean isUseDoubleCache = false;
    // 线程池,线程数量为 CPU 的数量
    ExecutorService mExecutorService = Executors.newFixedThreadPool
    (Runtime.getRuntime().availableProcessors());


    public void displayImage(final String url, final ImageView imageView) {
        Bitmap bmp = null;
         if (isUseDoubleCache) {
            bmp = mDoubleCache.get(url);
        } else if (isUseDiskCache) {
            bmp = mDiskCache.get(url);
        } else {
```

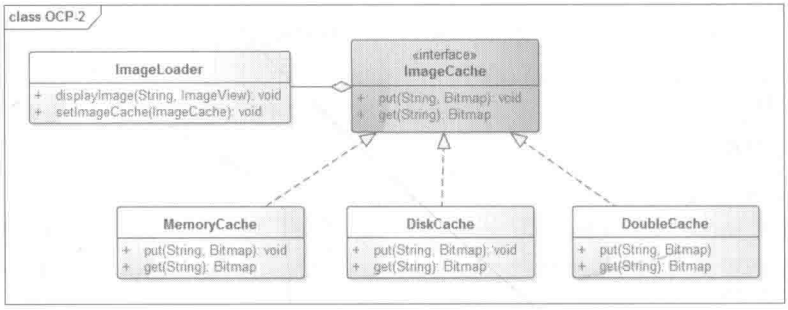
```
            bmp = mImageCache.get(url);
        }

        if ( bmp != null ) {
            imageView.setImageBitmap(bmp);
        }
        // 没有缓存，则提交给线程池进行异步下载图片
    }

    public  void useDiskCache(boolean useDiskCache) {
        isUseDiskCache = useDiskCache ;
    }

    public  void useDoubleCache(boolean useDoubleCache) {
        isUseDoubleCache = useDoubleCache ;
    }
}
```

通过增加短短几行代码和几处修改就完成了如此重要的功能。小民已越发觉得自己 Android 开发已经到了得心应手的境地，顿时感觉一阵春风袭来，小民感觉今天天空比往常敞亮许多。

“小民，你每次加新的缓存方法时都要修改原来的代码，这样很可能会引入 Bug，而且会使原来的代码逻辑变得越来越复杂。按照你这样的方法实现，用户也不能自定义缓存实现呀！”到底是主管水平高，一语道出了小民这缓存设计上的问题。

我们还是来分析一下小民的程序。小民每次在程序中加入新的缓存实现时都需要修改 ImageLoader 类，然后通过一个布尔变量来让用户选择使用哪种缓存，因此，就使得在 ImageLoader 中存在各种 if-else 判断语句，通过这些判断来确定使用哪种缓存。随着这些逻辑的引入，代码变得越来越复杂、脆弱，如果小民一不小心写错了某个 if 条件（条件太多，这是很容易出现的），那就需要更多的时间来排除，整个 ImageLoader 类也会变得越来越臃肿。最重要的是，用户不能自己实现缓存注入到 ImageLoader 中，可扩展性差，可扩展性可是框架的最重要特性之一。

“软件中的对象（类、模块、函数等）应该对于扩展是开放的，但是对于修改是封闭的，这就是开放——关闭原则。也就是说，当软件需要变化时，我们应该尽量通过扩展的方式来实现变化，而不是通过修改已有的代码来实现。”小民的主管补充到，小民听得云里雾里的。主管看小民这等反应，于是亲自“操刀”，为他画下了图 1-2 所示的 UML 图。



▲图 1-2

小民看到图 1-2 似乎明白些什么，但又不是很明确如何修改程序。主管看到小民这般模样，只

好亲自上阵，带着小民把 ImageLoader 程序按照图 1-2 进行了一次重构，具体代码如下：

```
public class ImageLoader {
    // 图片缓存
    ImageCache mImageCache = new MemoryCache();
    // 线程池,线程数量为 CPU 的数量
    ExecutorService mExecutorService = Executors.newFixedThreadPool
    (Runtime.getRuntime().availableProcessors());

    // 注入缓存实现
    public void setImageCache(ImageCache cache) {
        mImageCache = cache;
    }

    public void displayImage(String imageUrl, ImageView imageView) {
        Bitmap bitmap = mImageCache.get(imageUrl);
        if (bitmap != null) {
            imageView.setImageBitmap(bitmap);
            return;
        }
        // 图片没缓存，提交到线程池中下载图片
        submitLoadRequest(imageUrl, imageView);
    }

    private void submitLoadRequest(final String imageUrl,
            final ImageView imageView) {
        imageView.setTag(imageUrl);
        mExecutorService.submit(new Runnable() {

            @Override
            public void run() {
                Bitmap bitmap = downloadImage(imageUrl);
                    if (bitmap == null) {
                        return;
                    }
                    if (imageView.getTag().equals(imageUrl)) {
                        imageView.setImageBitmap(bitmap);
                    }
                    mImageCache.put(imageUrl, bitmap);
            }
        });
    }

    public Bitmap downloadImage(String imageUrl) {
        Bitmap bitmap = null;
         try {
            URL url = new URL(imageUrl);
             final HttpURLConnection conn = (HttpURLConnection) url.openConnection();
             bitmap = BitmapFactory.decodeStream(conn.getInputStream());
             conn.disconnect();
        } catch (Exception e) {
                e.printStackTrace();
        }

        return bitmap;
    }
}
```

经过这次重构，没有了那么多的 if-else 语句，没有了各种各样的缓存实现对象、布尔变量，代码确实清晰、简单了很多，小民对主管的崇敬之情又“泛滥”了起来。需要注意的是，这里的

ImageCache 类并不是小民原来的那个 ImageCache，这次重构程序，主管把它提取成一个图片缓存的接口，用来抽象图片缓存的功能，我们看看该接口的声明：

```
public interface ImageCache {
    public  Bitmap get(String url);
    public void put(String url, Bitmap bmp);
}
```

ImageCache 接口简单定义了获取、缓存图片两个函数，缓存的 key 是图片的 url，值是图片本身。内存缓存、SD 卡缓存、双缓存都实现了该接口，我们看看这几个缓存实现：

```
// 内存缓存 MemoryCache 类
public class MemoryCache implements ImageCache {
    private LruCache<String, Bitmap> mMemeryCache;

    public MemoryCache() {
        // 初始化 LRU 缓存
    }

     @Override
    public  Bitmap get(String url) {
        return mMemeryCache.get(url);
    }

    @Override
    public void put(String url, Bitmap bmp) {
        mMemeryCache.put(url, bmp);
    }
}

// SD 卡缓存 DiskCache 类
public class DiskCache implements ImageCache {
    @Override
    public  Bitmap get(String url) {
        return null/* 从本地文件中获取该图片 */;
    }

    @Override
    public void put(String url, Bitmap bmp) {
        // 将 Bitmap 写入文件中
    }
}

// 双缓存 DoubleCache 类
public class DoubleCache implements ImageCache{
    ImageCache mMemoryCache = new MemoryCache();
    ImageCache mDiskCache = new DiskCache();

    // 先从内存缓存中获取图片，如果没有，再从 SD 卡中获取
    public Bitmap get(String url) {
        Bitmap bitmap = mMemoryCache.get(url);
        if (bitmap == null) {
            bitmap = mDiskCache.get(url);
        }
        return bitmap;
    }

    // 将图片缓存到内存和 SD 卡中
    public void put(String url, Bitmap bmp) {
```

```
        mMemoryCache.put(url, bmp);
        mDiskCache.put(url, bmp);
    }
}
```

细心的读者可能注意到了，ImageLoader 类中增加了一个 **setImageCache**（ImageCache cache）函数，用户可以通过该函数设置缓存实现，也就是通常说的依赖注入。下面就看看用户是如何设置缓存实现的：

```
ImageLoader imageLoader = new ImageLoader() ;
        // 使用内存缓存
imageLoader.setImageCache(new MemoryCache());
        // 使用 SD 卡缓存
imageLoader.setImageCache(new DiskCache());
        // 使用双缓存
imageLoader.setImageCache(new DoubleCache());
        // 使用自定义的图片缓存实现
imageLoader.setImageCache(new ImageCache() {

        @Override
        public  void  put(String url, Bitmap bmp) {
            // 缓存图片
        }

        @Override
        public  Bitmap get(String url) {
                return  null/*从缓存中获取图片*/;
        }
    });
```

在上述代码中，通过 setImageCache（ImageCache cache）方法注入不同的缓存实现，这样不仅能够使 ImageLoader 更简单、健壮，也使得 ImageLoader 的可扩展性、灵活性更高。MemoryCache、DiskCache、DoubleCache 缓存图片的具体实现完全不一样，但是，它们的一个特点是，都实现了 ImageCache 接口。当用户需要自定义实现缓存策略时，只需要新建一个实现 ImageCache 接口的类，然后构造该类的对象，并且通过 setImageCache（ImageCache cache）注入到 ImageLoader 中，这样 ImageLoader 就实现了千变万化的缓存策略，且扩展这些缓存策略并不会导致 ImageLoader 类的修改。经过这次重构，小民的 ImageLoader 已经基本算合格了。咦！这不就是主管说的开闭原则么！“软件中的对象（类、模块、函数等）应该对于扩展是开放的，但是，对于修改是封闭的。而遵循开闭原则的重要手段应该是通过抽象……”小民细声细语地念叨着，陷入了思索中……

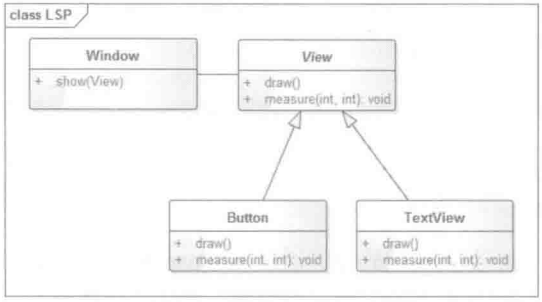
开闭原则指导我们，当软件需要变化时，应该尽量通过扩展的方式来实现变化，而不是通过修改已有的代码来实现。这里的“应该尽量”4 个字说明 OCP 原则并不是说绝对不可以修改原始类的。当我们嗅到原来的代码“腐化气味”时，应该尽早地重构，以便使代码恢复到正常的“进化”过程，而不是通过继承等方式添加新的实现，这会导致类型的膨胀以及历史遗留代码的冗余。我们的开发过程中也没有那么理想化的状况，完全地不用修改原来的代码，因此，在开发过程中需要自己结合具体情况进行考量，是通过修改旧代码还是通过继承使得软件系统更稳定、更灵活，在保证去除“代码腐化”的同时，也保证原有模块的正确性。

1.3 构建扩展性更好的系统——里氏替换原则

里氏替换原则英文全称是 Liskov Substitution Principle，缩写是 LSP。LSP 的第一种定义是：如果对每一个类型为 S 的对象 O1，都有类型为 T 的对象 O2，使得以 T 定义的所有程序 P 在所有的对象 O1 都代换成 O2 时，程序 P 的行为没有发生变化，那么类型 S 是类型 T 的子类型。上面这种描述确实不太好理解，我们再看看另一个直截了当的定义。里氏替换原则第二种定义：所有引用基类的地方必须能透明地使用其子类的对象。

我们知道，面向对象的语言的三大特点是继承、封装、多态，里氏替换原则就是依赖于继承、多态这两大特性。里氏替换原则简单来说就是，所有引用基类的地方必须能透明地使用其子类的对象。通俗点讲，只要父类能出现的地方子类就可以出现，而且替换为子类也不会产生任何错误或异常，使用者可能根本就不需要知道是父类还是子类。但是，反过来就不行了，有子类出现的地方，父类未必就能适应。说了那么多，其实最终总结就两个字：抽象。

小民为了深入地了解 Android 中的 Window 与 View 的关系，特意写了一个简单示例，为了便于理解，我们先看如图 1-3 所示。



▲图 1-3

我们看看具体的代码实现：

```
// 窗口类
public class Window {
    public void show(View child){
        child.draw();
    }
}

// 建立视图抽象，测量视图的宽高为公用代码，绘制实现交给具体的子类
public abstract class  View {
    public abstract void  draw() ;
    public void measure(int width, int height){
        // 测量视图大小
    }
}

// 按钮类的具体实现
public class Button extends View {
    public void draw(){
        // 绘制按钮
    }
}
// TextView 的具体实现
public class TextView extends View {
    public void draw(){
        // 绘制文本
    }
}
```

上述示例中，Window 依赖于 View，而 View 定义了一个视图抽象，measure 是各个子类共享的方法，子类通过覆写 View 的 draw 方法实现具有各自特色的功能，在这里，这个功能就是绘制

自身的内容。任何继承自 View 类的子类都可以设置给 show 方法，就是所说的里氏替换。通过里氏替换，就可以自定义各式各样、千变万化的 View，然后传递给 Window，Window 负责组织 View，并且将 View 显示到屏幕上。

里氏替换原则的核心原理是抽象，抽象又依赖于继承这个特性，在 OOP 当中，继承的优缺点都相当明显。优点有以下几点：

（1）代码重用，减少创建类的成本，每个子类都拥有父类的方法和属性；

（2）子类与父类基本相似，但又与父类有所区别；

（3）提高代码的可扩展性。

继承的缺点：

（1）继承是侵入性的，只要继承就必须拥有父类的所有属性和方法；

（2）可能造成子类代码冗余、灵活性降低，因为子类必须拥有父类的属性和方法。

事物总是具有两面性，如何权衡利与弊都是需要根据具体情况来做出选择并加以处理。里氏替换原则指导我们构建扩展性更好的软件系统，我们还是接着上面的 ImageLoader 来做说明。

图 1-2 也很好地反应了里氏替换原则，即 MemoryCache、DiskCache、DoubleCache 都可以替换 ImageCache 的工作，并且能够保证行为的正确性。ImageCache 建立了获取缓存图片、保存缓存图片的接口规范，MemoryCache 等根据接口规范实现了相应的功能，用户只需要在使用时指定具体的缓存对象就可以动态地替换 ImageLoader 中的缓存策略。这就使得 ImageLoader 的缓存系统具有了无限的可能性，也就是保证了可扩展性。

想象一种情况，当 ImageLoader 中的 setImageCache（ImageCache cache）中的 cache 对象不能够被子类所替换，那么用户如何设置不同的缓存对象，以及用户如何自定义自己的缓存实现，通过 1.3 节中的 useDiskCache 方法吗？显然不是的，里氏替换原则就为这类问题提供了指导原则，也就是建立抽象，通过抽象建立规范，具体的实现在运行时替换掉抽象，保证系统的扩展性、灵活性。开闭原则和里氏替换原则往往是生死相依、不弃不离的，通过里氏替换来达到对扩展开放，对修改关闭的效果。然而，这两个原则都同时强调了一个 OOP 的重要特性——抽象，因此，在开发过程中运用抽象是走向代码优化的重要一步。

1.4 让项目拥有变化的能力——依赖倒置原则

依赖倒置原则英文全称是 Dependence Inversion Principle，缩写是 DIP。**依赖倒置原则**指代了一种特定的解耦形式，使得高层次的模块不依赖于低层次的模块的实现细节的目的，依赖模块被颠倒了。这个概念有点不好理解，这到底是什么意思呢？

依赖倒置原则有以下几个关键点：

（1）高层模块不应该依赖低层模块，两者都应该依赖其抽象；

（2）抽象不应该依赖细节；

（3）细节应该依赖抽象。

在 Java 语言中，抽象就是指接口或抽象类，两者都是不能直接被实例化的；细节就是实现类，实现接口或继承抽象类而产生的类就是细节，其特点就是，可以直接被实例化，也就是可以加上一

个关键字 new 产生一个对象。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 语言中的表现就是：**模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的。**这又是一个将理论抽象化的实例，其实一句话就可以概括：面向接口编程，或者说是面向抽象编程，这里的抽象指的是接口或者抽象类。面向接口编程是面向对象精髓之一，也就是上面两节强调的抽象。

如果类与类直接依赖于细节，那么它们之间就有直接的耦合，当具体实现需要变化时，意味着要同时修改依赖者的代码，这限制了系统的可扩展性。在 1.3 节的图 1-3 中，ImageLoader 直接依赖于 MemoryCache，这个 MemoryCache 是一个具体实现，而不是一个抽象类或者接口。这导致了 ImageLoader 直接依赖了具体细节，当 MemoryCache 不能满足 ImageLoader 而需要被其他缓存实现替换时，此时就必须修改 ImageLoader 的代码，例如：

```
public class ImageLoader {
    // 内存缓存（直接依赖于细节）
    MemoryCache mMemoryCache = new MemoryCache();
     // 加载图片到 ImageView 中
    public void displayImage(String url, ImageView imageView) {
        Bitmap bmp = mMemoryCache.get(url);
         if (bmp == null) {
            downloadImage(url, imageView);
        } else {
            imageView.setImageBitmap(bmp);
        }
    }

    public void  setImageCache(MemoryCache cache) {
         mCache = cache ;
    }
    // 代码省略
}
```

随着产品的升级，用户发现 MemoryCache 已经不能满足需求，用户需要小民的 ImageLoader 可以将图片同时缓存到内存和 SD 卡中，或者可以让用户自定义实现缓存。此时，我们的 MemoryCache 这个类名不仅不能够表达内存缓存和 SD 卡缓存的意义，也不能够满足功能。另外，用户需要自定义缓存实现时还必须继承自 MemoryCache，而用户的缓存实现可不一定与内存缓存有关，这在命名上的限制也让用户体验不好。重构的时候到了！小民的第一种方案是将 MemoryCache 修改为 DoubleCache，然后在 DoubleCache 中实现具体的缓存功能。我们需要将 ImageLoader 修改如下：

```
public class ImageLoader {
    // 双缓存（直接依赖于细节）
    DoubleCache mCache = newDoubleCache();
    // 加载图片到 ImageView 中
    public void displayImage(String url, ImageView imageView) {
        Bitmap bmp = mCache.get(url);
         if (bmp == null) {
            // 异步下载图片
            downloadImageAsync(url, imageView);
        } else {
            imageView.setImageBitmap(bmp);
        }
    }
```

```
    public void  setImageCache(DoubleCache cache) {
        mCache = cache ;
    }
    // 代码省略
}
```

在程序中我们将 MemoryCache 修改成 DoubleCache，然后修改了 ImageLoader 中缓存类的具体实现，轻轻松松就满足了用户需求。等等！这不还是依赖于具体的实现类（DoubleCache）吗？当用户的需求再次变化时，我们又要通过修改缓存实现类和 ImageLoader 代码来实现？修改原有代码不是违反了 1.3 节中的开闭原则吗？小民突然醒悟了过来，低下头思索着如何才能让缓存系统更灵活，拥抱变化……

当然，这些都是在主管给出图 1-2 以及相应的代码之前，小民体验的煎熬过程。既然是这样，那显然主管给出的解决方案就能够让缓存系统更加灵活。一句话概括起来就是：依赖抽象，而不依赖具体实现。针对于图片缓存，主管建立的 ImageCache 抽象，该抽象中增加了 get 和 put 方法用以实现图片的存取。每种缓存实现都必须实现这个接口，并且实现自己的存取方法。当用户需要使用不同的缓存实现时，直接通过依赖注入即可，保证了系统的灵活性。我们再来简单回顾一下相关代码。

ImageCache 缓存抽象：

```
public interface ImageCache {
    public  Bitmap get(String url);
    public void put(String url, Bitmap bmp);
}
```

ImageLoader 类：

```
public class ImageLoader {
    // 图片缓存类，依赖于抽象，并且有一个默认的实现
    ImageCache mCache = new MemoryCache();

    // 加载图片
    public void displayImage(String url, ImageView imageView) {
        Bitmap bmp = mCache.get(url);
         if (bmp == null) {
            // 异步加载图片
            downloadImageAsync(url, imageView);
        } else {
            imageView.setImageBitmap(bmp);
        }
    }

    /**
     * 设置缓存策略,依赖于抽象
     */
    public void setImageCache(ImageCache cache) {
        mCache = cache;
    }
    // 代码省略
}
```

在这里，我们建立了 ImageCache 抽象，并且让 ImageLoader 依赖于抽象而不是具体细节。当需求发生变化时，小民只需要实现 ImageCahce 类或者继承其他已有的 ImageCache 子类完成相应的

缓存功能，然后将具体的实现注入到 ImageLoader 即可实现缓存功能的替换，这就保证了缓存系统的高可扩展性，有了拥抱变化的能力，这就是依赖倒置原则。从上述几节中我们发现，要想让系统更为灵活，抽象似乎成了我们唯一的手段。

1.5 系统有更高的灵活性——接口隔离原则

接口隔离原则英文全称是 InterfaceSegregation Principles，缩写是 ISP。ISP 的定义是：客户端不应该依赖它不需要的接口。另一种定义是：类间的依赖关系应该建立在最小的接口上。接口隔离原则将非常庞大、臃肿的接口拆分成更小的和更具体的接口，这样客户将会只需要知道他们感兴趣的方法。接口隔离原则的目的是系统解开耦合，从而容易重构、更改和重新部署。

接口隔离原则说白了就是，让客户端依赖的接口尽可能地小，这样说可能还是有点抽象，我们还是以一个示例来说明一下。在此之前我们来说一个场景，在 Java 6 以及之前的 JDK 版本，有一个非常讨厌的问题，那就是在使用了 OutputStream 或者其他可关闭的对象之后，我们必须保证它们最终被关闭了，我们的 SD 卡缓存类中就有这样的代码：

```
// 将图片缓存到内存中
public  void  put(String url, Bitmap bmp) {
    FileOutputStream fileOutputStream = null;
    try {
        fileOutputStream = new FileOutputStream(cacheDir + url);
        bmp.compress(CompressFormat.PNG, 100, fileOutputStream);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        if (fileOutputStream != null) {
            try {
                fileOutputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        } // end if
    } // end if finally
}
```

我们看到的这段代码可读性非常差，各种 try…catch 嵌套都是些简单的代码，但是会严重影响代码的可读性，并且多层级的大括号很容易将代码写到错误的层级中。大家应该对这类代码也非常反感，那我们看看如何解决这类问题。

我们可能知道 Java 中有一个 Closeable 接口，该接口标识了一个可关闭的对象，它只有一个 close 方法，如图 1-4 所示。

我们要讲的 FileOutputStream 类就实现了这个接口。我们从图 1-4 中可以看到，还有 100 多个类实现了 Closeable 这个接口，这意味着，在关闭这 100 多个类型的对象时，都需要写出像 put 方法中 finally 代码段那样的代码。这还了得！你能忍，反正小民是忍不了的！于是小民打算要发挥他的聪明才智解决这个问题，既然都是实现了 Closeable 接口，那只要我建一个方法统一来关闭这些对象不就可以了么？说干就干，于是小民写下来如下的工具类：

Closeable

Added in API

implements AutoCloseable

java.io.Closeable

▶ Known Indirect Subclasses

AbstractCursor, AbstractInterruptibleChannel, AbstractSelectableChannel, AbstractSelector, AbstractWindowedCursor, AssetFileDescriptor, AssetFileDescriptor.AutoCloseInputStream, AssetFileDescriptor.AutoCloseOutputStream, AssetManager.AssetInputStream, BackupDataInputStream, Base64InputStream, Base64OutputStream, BluetoothServerSocket, BluetoothSocket, and 122 others.

Class Overview

An `AutoCloseable` whose close method may throw an `IOException`.

Summary

Public Methods	
abstract void	close() Closes the object and release any system resources it holds.

▲图 1-4

```
public final class CloseUtils {

    Private CloseUtils() { }
    /**
     * 关闭 Closeable 对象
     * @param closeable
     */
    public  static void closeQuietly(Closeable closeable) {
        if (null != closeable) {
            try {
                closeable.close();
            }catch (IOException e) {
                    e.printStackTrace();
            }
        }
    }
}
```

我们再看看把这段代码运用到上述的 put 方法中的效果如何：

```
public  void put(String url, Bitmap bmp) {
    FileOutputStream fileOutputStream = null;
    try {
        fileOutputStream = new FileOutputStream(cacheDir + url);
        bmp.compress(CompressFormat.PNG, 100, fileOutputStream);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        CloseUtils.closeQuietly(fileOutputStream);
    }
}
```

代码简洁了很多！而且这个 closeQuietly 方法可以运用到各类可关闭的对象中，保证了代码的重用性。CloseUtils 的 closeQuietly 方法的基本原理就是依赖于 Closeable 抽象而不是具体实现（这不是 1.4 节中的依赖倒置原则么），并且建立在最小化依赖原则的基础上，它只需要知道这个对象

是可关闭，其他的一概不关心，也就是这里的接口隔离原则。

试想一下，如果在只是需要关闭一个对象时，它却暴露出了其他的接口函数，如 OutputStream 的 write 方法，这就使得更多的细节暴露在客户端代码面前，不仅没有很好地隐藏实现，还增加了接口的使用难度。而通过 Closeable 接口将可关闭的对象抽象起来，这样只需要客户端依赖于 Closeable 就可以对客户端隐藏其他的接口信息，客户端代码只需要知道这个对象可关闭（只可调用 close 方法）即可。小民设计的 ImageLoader 中的 ImageCache 就是接口隔离原则的运用，ImageLoader 只需要知道该缓存对象有存、取缓存图片的接口即可，其他的一概不管，这就使得缓存功能的具体实现对 ImageLoader 隐藏。这就是用最小化接口隔离了实现类的细节，也促使我们将庞大的接口拆分到更细粒度的接口当中，这使得我们的系统具有更低的耦合性，更高的灵活性。

Bob 大叔（Robert C Martin）在 21 世纪早期将**单一职责、开闭原则、里氏替换、接口隔离**以及**依赖倒置（也称为依赖反转）**5 个原则定义为 **SOLID** 原则，作为面向对象编程的 5 个基本原则。当这些原则被一起应用时，它们使得一个软件系统更清晰、简单，最大程度地拥抱变化。SOLID 被典型地应用在测试驱动开发上，并且是敏捷开发以及自适应软件开发基本原则的重要组成部分。在经过第 1.1～1.5 节的学习之后，我们发现这几大原则最终就可以化为这几个关键词：抽象、单一职责、最小化。那么，在实际开发过程中如何权衡、实践这些原则，大家需要在实践中多思考与领悟，正所谓”学而不思则罔，思而不学则殆”，只有不断地学习、实践、思考，才能够在积累的过程中有一个质的飞越。

1.6 更好的可扩展性——迪米特原则

迪米特原则英文全称为 Law of Demeter，缩写是 LOD，也称为最少知识原则（Least Knowledge Principle）。虽然名字不同，但描述的是同一个原则：一个对象应该对其他对象有最少的了解。通俗地讲，一个类应该对自己需要耦合或调用的类知道得最少，类的内部如何实现与调用者或者依赖者没关系，调用者或者依赖者只需要知道它需要的方法即可，其他的可一概不用管。类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

迪米特法则还有一个英文解释是 Only talk to your immedate friends，翻译过来就是：只与直接的朋友通信。什么叫做直接的朋友呢？每个对象都必然会与其他对象有耦合关系，两个对象之间的耦合就成为朋友关系，这种关系的类型有很多，如组合、聚合、依赖等。

下面我们就以租房为例来讲讲迪米特原则的应用。

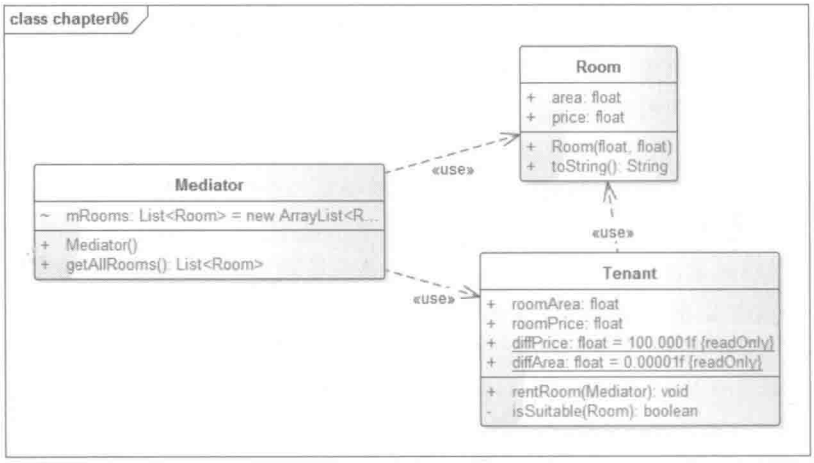
“北漂”的朋友比较了解，在北京租房绝大多数都是通过中介找房。我们设定的情况为：我只要求房间的面积和租金，其他的一概不管，中介将符合我要求的房子提供给我就可以。下面我们看看这个示例：

```
/**
 * 房间
 */
public class Room {
    public float area;
    public float price;
```

```
    public Room(float area, float price) {
        this.area = area;
        this.price = price;
    }

    @Override
    public String toString() {
        return "Room [area=" + area + ", price=" + price + "]";
    }

}

/**
 * 中介
 */
public class Mediator {
    List<Room> mRooms = new ArrayList<Room>();

    public Mediator() {
        for (inti = 0; i < 5; i++) {
            mRooms.add(new Room(14 + i, (14 + i) * 150));
        }
    }

    public  List<Room>getAllRooms() {
        return mRooms;
    }
}


/**
 * 租户
 */
public class Tenant {

    public float roomArea;
    public float roomPrice;
    public static final float diffPrice = 100.0001f;
    public static final float diffArea = 0.00001f;

    public void rentRoom(Mediator mediator) {
        List<Room>rooms = mediator.getAllRooms();
        for (Room room : rooms) {
            if (isSuitable(room)) {
              System.out.println("租到房间啦! " + room);
                break;
            }
        }
    }

    private boolean isSuitable(Room room) {
        return Math.abs(room.price - roomPrice) < diffPrice
                &&Math.abs(room.area - roomArea) < diffArea;
    }
}
```

从上面的代码中可以看到，Tenant 不仅依赖了 Mediator 类，还需要频繁地与 Room 类打交道。租户类的要求只是通过中介找到一间适合自己的房间罢了，如果把这些检测条件都放在 Tenant 类中，那么中介类的功能就被弱化，而且导致 Tenant 与 Room 的耦合较高，因为 Tenant 必须知道许

多关于 Room 的细节。当 Room 变化时 Tenant 也必须跟着变化。Tenant 又与 Mediator 耦合，这就出现了纠缠不清的关系。这个时候就需要我们分清谁才是我们真正的“朋友”，在我们所设定的情况下，显然是 Mediator（虽然现实生活中不是这样的）。上述代码的结构如图 1-5 所示。



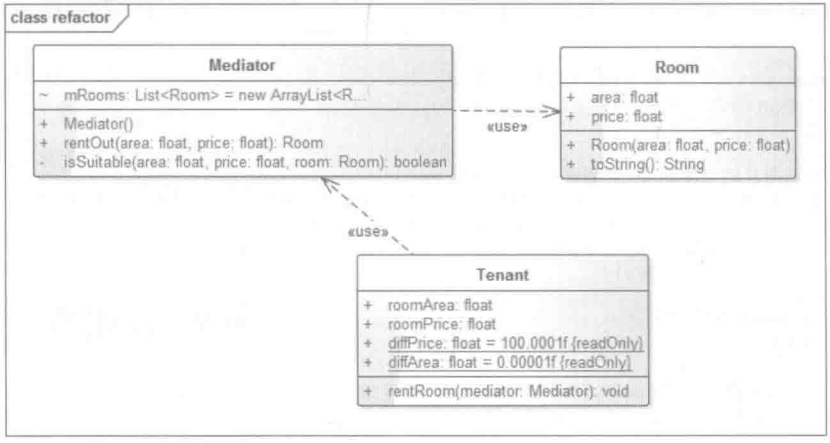
▲图 1-5

既然是耦合太严重，那我们就只能解耦了。首先要明确的是，我们只和我们的朋友通信，这里就是指 Mediator 对象。必须将 Room 相关的操作从 Tenant 中移除，而这些操作案例应该属于 Mediator。我们进行如下重构：

```
/**
 * 中介
 */
public class Mediator {
    List<Room> mRooms = new ArrayList<Room>();

    public Mediator() {
        for (inti = 0; i < 5; i++) {
            mRooms.add(new Room(14 + i, (14 + i) * 150));
        }
    }

    public  Room rentOut(float area, float price) {
        for (Room room : mRooms) {
            if (isSuitable(area, price, room)) {
                return room;
            }
        }

        return null;
    }

    private boolean isSuitable(float area, float price, Room room) {
        return Math.abs(room.price - price) < Tenant.diffPrice
            && Math.abs(room.area - area) < Tenant.diffPrice;
    }
}
```

```
/**
 * 租户
 */
public class Tenant {

    public float roomArea;
    public float roomPrice;
    public static final float diffPrice = 100.0001f;
    public static final float diffArea = 0.00001f;

    public void rentRoom(Mediator mediator) {
        System.out.println("租到房啦 " + mediator.rentOut(roomArea, roomPrice));
    }
}
```

重构后的结构图如图 1-6 所示。



▲图 1-6

只是将对于 Room 的判定操作移到了 Mediator 类中，这本应该是 Mediator 的职责，根据租户设定的条件查找符合要求的房子，并且将结果交给租户就可以了。租户并不需要知道太多关于 Room 的细节，比如与房东签合同，房东的房产证是不是真的，房内的设施坏了之后要找谁维修等。当我们通过我们的“朋友”——中介租了房之后，所有的事情我们都通过与中介沟通就好了，房东、维修师傅等这些角色并不是我们直接的“朋友”。“只与直接的朋友通信”这简单的几个字就能够将我们从复杂的关系网中抽离出来，使程序耦合度更低、稳定性更好。

通过上述示例以及小民的后续思考，迪米特原则这把利剑在小民的手中已经舞得风生水起。就拿 SD 卡缓存来说吧，ImageCache 就是用户的直接朋友，而 SD 卡缓存内部却是使用了 jake wharton 的 DiskLruCache 实现，这个 DiskLruCache 就不属于用户的直接朋友了，因此，用户完全不需要知道它的存在，用户只需要与 ImageCache 对象打交道即可，如将图片存到 SD 卡中的代码如下：

```
public void put(String url, Bitmap value) {
    DiskLruCache.Editor editor = null;
    try {
        // 如果没有找到对应的缓存，则准备从网络上请求数据，并写入缓存
        editor = mDiskLruCache.edit(url);
        if (editor != null) {
```

```
            OutputStream outputStream = editor.newOutputStream(0);
            if (writeBitmapToDisk(value, outputStream)) {
                // 写入 Disk 缓存
                editor.commit();
            } else {
                editor.abort();
            }
            CloseUtils.closeQuietly(outputStream);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

用户在使用 SD 卡缓存时，根本不知道 DiskLruCache 的实现，这就很好地对用户隐藏了具体实现。当小民已经“牛”到可以自己完成 SD 卡的 LRU 实现时，他就可以随心所欲地替换掉 jake wharton 的 DiskLruCache。小民的代码大体如下：

```
@Override
public  void put(String url, Bitmap bmp) {
    // 将 Bitmap 写入文件中
    FileOutputStream fos = null;
    try {
       // 构建图片的存储路径 ( 省略了对 url 取 md5)
        fos = new FileOutputStream("sdcard/cache/" + imageUrl2MD5(url));
        bmp.compress(CompressFormat.JPEG, 100, fos);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        if ( fos != null ) {
            try {
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    } // end if finally
}
```

SD 卡缓存的具体实现虽然被替换了，但用户根本不会感知到。因为用户根本不知道 DiskLruCache 的存在，他们没有与 DiskLruCache 进行通信，他们只认识直接“朋友”——ImageCache，ImageCache 将一切细节隐藏在了直接“朋友”的外衣之下，使得系统具有更低的耦合性和更好的可扩展性。

1.7 总结

在应用开发过程中，最难的不是完成应用的开发工作，而是在后续的升级、维护过程中让应用系统能够拥抱变化。拥抱变化也就意味着在满足需求且不破坏系统稳定性的前提下保持高可扩展性、高内聚、低耦合，在经历了各版本的变更之后依然保持清晰、灵活、稳定的系统架构。当然，这是一个比较理想的情况，但我们必须要朝着这个方向去努力，那么遵循面向对象六大原则就是我们走向灵活软件之路所迈出的第一步。

第 2 章　应用最广的模式——单例模式

2.1 单例模式介绍

单例模式是应用最广的模式之一，也可能是很多初级工程师唯一会使用的设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个全局对象，这样有利于我们协调系统整体的行为。如在一个应用中，应该只有一个 ImageLoader 实例，这个 ImageLoader 中又含有线程池、缓存系统、网络请求等，很消耗资源，因此，没有理由让它构造多个实例。这种不能自由构造对象的情况，就是单例模式的使用场景。

2.2 单例模式的定义

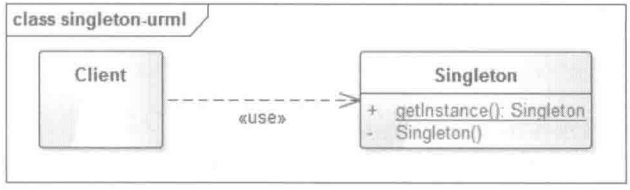
确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

2.3 单例模式的使用场景

确保某个类有且只有一个对象的场景，避免产生多个对象消耗过多的资源，或者某种类型的对象只应该有且只有一个。例如，创建一个对象需要消耗的资源过多，如要访问 IO 和数据库等资源，这时就要考虑使用单例模式。

2.4 单例模式 UML 类图

UML 类图如图 2-1 所示。



▲图 2-1

角色介绍：

（1）Client——高层客户端；

（2）Singleton——单例类。

实现单例模式主要有如下几个关键点：

（1）构造函数不对外开放，一般为 Private；

（2）通过一个静态方法或者枚举返回单例类对象；

（3）确保单例类的对象有且只有一个，尤其是在多线程环境下；

（4）确保单例类对象在反序列化时不会重新构建对象。

通过将单例类的构造函数私有化，使得客户端代码不能通过 new 的形式手动构造单例类的对象。单例类会暴露一个公有静态方法，客户端需要调用这个静态方法获取到单例类的唯一对象，在获取这个单例对象的过程中需要确保线程安全，即在多线程环境下构造单例类的对象也是有且只有一个，这也是单例模式实现中比较困难的地方。

2.5 单例模式的简单示例

单例模式是设计模式中比较简单的，只有一个单例类，没有其他的层次结构与抽象。该模式需要确保该类只能生成一个对象，通常是该类需要消耗较多的资源或者没有多个实例的情况。例如，一个公司只有一个 CEO、一个应用只有一个 Application 对象等。下面以公司里的 CEO 为例来简单演示一下，一个公司可以有几个 VP、无数个员工，但是 CEO 只有一个，请看下面示例。

示例实现代码

```
package com.dp.example.singleton;
// 普通员工
public class Staff {
    public void work()  {
        // 干活
    }
}

// 副总裁
public class VP extends Staff {
    @Override
    public void work() {
        // 管理下面的经理
    }
}

// CEO，饿汉单例模式
public class CEO extends Staff {
    private static final CEO mCeo = new CEO();
    // 构造函数私有
    private CEO() {
    }

    // 公有的静态函数，对外暴露获取单例对象的接口
    public static CEO getCeo() {
        return mCeo;
    }
```

```
    @Override
    public void work() {
        // 管理 VP
    }
}

// 公司类
public class Company {
    private List<Staff> allStaffs = new ArrayList<Staff>();
    public void addStaff(Staff per) {
        allStaffs.add(per);
    }

    public void showAllStaffs() {
        for (Staff per : allStaffs) {
            System.out.println("Obj : " + per.toString());
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Company cp = new Company() ;
        // CEO 对象只能通过 getCeo 函数获取
        Staff ceo1 = CEO.getCeo() ;
        Staff ceo2 = CEO.getCeo() ;
        cp.addStaff(ceo1);
        cp.addStaff(ceo2);
        // 通过 new 创建 VP 对象
        Staff vp1 = new VP() ;
        Staff vp2 = new VP() ;
        // 通过 new 创建 Staff 对象
        Staff staff1 = new Staff() ;
        Staff staff2 = new Staff() ;
        Staff staff3 = new Staff() ;

        cp.addStaff(vp1);
        cp.addStaff(vp2);
        cp.addStaff(staff1);
        cp.addStaff(staff2);
        cp.addStaff(staff3);

        cp.showAllStaffs();
    }
}
```

输出结果如下：

```
Obj : com.android.dp.book.chapter2.company.CEO@5e8fce95
Obj : com.android.dp.book.chapter2.company.CEO@5e8fce95
Obj : com.android.dp.book.chapter2.company.VP@3343c8b3
Obj : com.android.dp.book.chapter2.company.VP@222d2a10
Obj : com.android.dp.book.chapter2.company.Staff@1aa8c488
Obj : com.android.dp.book.chapter2.company.Staff@3dfeca64
Obj : com.android.dp.book.chapter2.company.Staff@22998b08
```

从上述的代码中可以看到，CEO 类不能通过 new 的形式构造对象，只能通过 CEO.getCEO() 函数来获取，而这个 CEO 对象是静态对象，并且在声明的时候就已经初始化，这就保证了 CEO 对象的唯一性。从输出结果中发现，CEO 两次输出的 CEO 对象都是一样的，而 VP、Staff 等类型的

对象都是不同的。这个实现的核心在于将 CEO 类的构造方法私有化，使得外部程序不能通过构造函数来构造 CEO 对象，而 CEO 类通过一个静态方法返回一个静态对象。

2.6 单例模式的其他实现方式

2.6.1 懒汉模式

懒汉模式是声明一个静态对象，并且在用户第一次调用 getInstance 时进行初始化，而上述的饿汉模式（CEO 类）是在声明静态对象时就已经初始化。懒汉单例模式实现如下。

```
public class Singleton {
    private static Singleton instance;
    private Singleton () {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton ();
        }
        return instance;
    }
}
```

读者可能已经发现了，getInstance（）方法中添加了 synchronized 关键字，也就是 getInstance 是一个同步方法，这就是上面所说的在多线程情况下保证单例对象唯一性的手段。细想一下，大家可能会发现一个问题，即使 instance 已经被初始化（第一次调用时就会被初始化 instance），每次调用 getInstance 方法都会进行同步，这样会消耗不必要的资源，这也是懒汉单例模式存在的最大问题。

最后总结一下，懒汉单例模式的优点是单例只有在使用时才会被实例化，在一定程度上节约了资源；缺点是第一次加载时需要及时进行实例化，反应稍慢，最大的问题是每次调用 getInstance 都进行同步，造成不必要的同步开销。这种模式一般不建议使用。

2.6.2 Double Check Lock (DCL)实现单例

DCL 方式实现单例模式的优点是既能够在需要时才初始化单例，又能够保证线程安全，且单例对象初始化后调用 getInstance 不进行同步锁。代码如下所示：

```
public class Singleton {
    private static Singleton sInstance = null;
    private Singleton() {
    }
    public void doSomething() {
        System.out.println("do sth.");
    }

    public static Singleton getInstance() {
        if (mInstance == null) {
            synchronized (Singleton.class) {
                if (mInstance == null) {
                    sInstance = new Singleton();
                }
            }
        }
```

```
        return sInstance;
    }
```

本程序的亮点自然都在 getInstance 方法上，可以看到 getInstance 方法中对 instance 进行了两次判空：第一层判断主要是为了避免不必要的同步，第二层的判断则是为了在 null 的情况下创建实例。这是什么意思呢？是不是有点摸不着头脑，下面就一起来分析一下。

假设线程 A 执行到 sInstance = new Singleton()语句，这里看起来是一句代码，但实际上它并不是一个原子操作，这句代码最终会被编译成多条汇编指令，它大致做了 3 件事情：

（1）给 Singleton 的实例分配内存；

（2）调用 Singleton()的构造函数，初始化成员字段；

（3）将 sInstance 对象指向分配的内存空间（此时 sInstance 就不是 null 了）。

但是，由于 Java 编译器允许处理器乱序执行，以及 JDK1.5 之前 JMM（Java Memory Model，即 Java 内存模型）中 Cache、寄存器到主内存回写顺序的规定，上面的第二和第三的顺序是无法保证的。也就是说，执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，并且在 3 执行完毕、2 未执行之前，被切换到线程 B 上，这时候 sInstance 因为已经在线程 A 内执行过了第三点，sInstance 已经是非空了，所以，线程 B 直接取走 sInstance，再使用时就会出错，这就是 DCL 失效问题，而且这种难以跟踪难以重现的错误很可能会隐藏很久。

在 JDK1.5 之后，SUN 官方已经注意到这种问题，调整了 JVM，具体化了 volatile 关键字，因此，如果 JDK 是 1.5 或之后的版本，只需要将 sInstance 的定义改成 private volatile static Singleton sInstance = null 就可以保证 sInstance 对象每次都是从主内存中读取，就可以使用 DCL 的写法来完成单例模式。当然，volatile 或多或少也会影响到性能，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：资源利用率高，第一次执行 getInstance 时单例对象才会被实例化，效率高。缺点：第一次加载时反应稍慢，也由于 Java 内存模型的原因偶尔会失败。在高并发环境下也有一定的缺陷，虽然发生概率很小。DCL 模式是使用最多的单例实现方式，它能够在需要时才实例化单例对象，并且能够在绝大多数场景下保证单例对象的唯一性，除非你的代码在并发场景比较复杂或者低于 JDK 6 版本下使用，否则，这种方式一般能够满足需求。

2.6.3 静态内部类单例模式

DCL 虽然在一定程度上解决了资源消耗、多余的同步、线程安全等问题，但是，它还是在某些情况下出现失效的问题。这个问题被称为双重检查锁定（DCL）失效，在《Java 并发编程实践》一书的最后谈到了这个问题，并指出这种“优化”是丑陋的，不赞成使用。而建议使用如下的代码替代：

```
public class Singleton {
    private Singleton() { }
    public static Singleton getInstance () {
        return SingletonHolder.sInstance;
    }

    /**
     * 静态内部类
     */
    private static class SingletonHolder {
```

```
        private static final Singleton sInstance = new Singleton();
    }
}
```

当第一次加载 Singleton 类时并不会初始化 sInstance，只有在第一次调用 Singleton 的 getInstance 方法时才会导致 sInstance 被初始化。因此，第一次调用 getInstance 方法会导致虚拟机加载 SingletonHolder 类，这种方式不仅能够确保线程安全，也能够保证单例对象的唯一性，同时也延迟了单例的实例化，所以这是推荐使用的单例模式实现方式。

2.6.4　枚举单例

前面讲解了一些单例模式实现方式，但是，这些实现方式不是稍显麻烦就是会在某些情况下出现问题。还有没有更简单的实现方式呢？我们看看下面的实现：

```
public enum SingletonEnum {
    INSTANCE;
    public void doSomething() {
        System.out.println("do sth.");
    }
}
```

什么？枚举！没错，就是枚举！

写法简单是枚举单例最大的优点，枚举在 Java 中与普通的类是一样的，不仅能够有字段，还能够有自己的方法。最重要的是默认枚举实例的创建是线程安全的，并且在任何情况下它都是一个单例。

为什么这么说呢？在上述的几种单例模式实现中，在一个情况下它们会出现重新创建对象的情况，那就是反序列化。

通过序列化可以将一个单例的实例对象写到磁盘，然后再读回来，从而有效地获得一个实例。即使构造函数是私有的，反序列化时依然可以通过特殊的途径去创建类的一个新的实例，相当于调用该类的构造函数。反序列化操作提供了一个很特别的钩子函数，类中具有一个私有的、被实例化的方法 readResolve()，这个方法可以让开发人员控制对象的反序列化。例如，上述几个示例中如果要杜绝单例对象在被反序列化时重新生成对象，那么必须加入如下方法：

```
private Object readResolve() throws ObjectStreamException {
    return sInstance;
}
```

也就是在 readResolve 方法中将 sInstance 对象返回，而不是默认的重新生成一个新的对象。而对于枚举，并不存在这个问题，因为即使反序列化它也不会重新生成新的实例。

2.6.5　使用容器实现单例模式

在学习了上述各类单例模式的实现之后，再来看看一种另类的实现，具体代码如下：

```
    public class SingletonManager {
    private static Map<String, Object> objMap = new HashMap<String,Object>();

    private SingletonManager() { }
    public static void registerService(String key, Object instance) {
        if (!objMap.containsKey(key) ) {
```

```
            objMap.put(key, instance) ;
        }
    }

    public static Object getService(String key) {
        return objMap.get(key) ;
    }

}
```

在程序的初始，将多种单例类型注入到一个统一的管理类中，在使用时根据 key 获取对象对应类型的对象。这种方式使得我们可以管理多种类型的单例，并且在使用时可以通过统一的接口进行获取操作，降低了用户的使用成本，也对用户隐藏了具体实现，降低了耦合度。

不管以哪种形式实现单例模式，它们的核心原理都是将构造函数私有化，并且通过静态方法获取一个唯一的实例，在这个获取的过程中必须保证线程安全、防止反序列化导致重新生成实例对象等问题。选择哪种实现方式取决于项目本身，如是否是复杂的并发环境、JDK 版本是否过低、单例对象的资源消耗等。

2.7 Android 源码中的单例模式

在 Android 系统中，我们经常会通过 Context 获取系统级别的服务，如 WindowsManagerService、ActivityManagerService 等，更常用的是一个 LayoutInflater 的类，这些服务会在合适的时候以单例的形式注册在系统中，在我们需要的时候就通过 Context 的 getSystemService(String name)获取。我们以 LayoutInflater 为例来说明，平时我们使用 LayoutInflater 较为常见的地方是在 ListView 的 getView 方法中：

```
@Override
public View getView(int position, View convertView, ViewGroup parent)
    View itemView = null;
    if (convertView == null) {
        itemView = LayoutInflater.from(mContext).inflate(mLayoutId, null);
        // 代码省略
    } else {
        // 代码省略
    }
    // 代码省略
    return itemView;
}
```

通常我们使用 LayoutInflater.from(Context)来获取 LayoutInflater 服务，下面看看 LayoutInflater.from (Context)的实现：

```
public static LayoutInflater from(Context context) {
    LayoutInflater LayoutInflater =(LayoutInflater) context.getSystemService(Context.
LAYOUT_INFLATER_SERVICE);

    if (LayoutInflater == null) {
        throw new AssertionError("LayoutInflater not found.");
    }
    return LayoutInflater;
}
```

可以看到 from(Context)函数内部调用的是 Context 类的 getSystemService(String key)方法，我们跟踪到 Context 类看到，该类是抽象类：

```
public abstract class Context {
    // 省略
}
```

getView 中使用的 Context 对象的具体实现类是什么呢？其实在 Application、Activity、Service 中都会存在一个 Context 对象，即 Context 的总个数为 Activity 个数 + Service 个数 + 1。而 ListView 通常都是显示在 Activity 中，那么我们就以 Activity 中的 Context 来分析。

我们知道，一个 Activity 的入口是 ActivityThread 的 main 函数，在 main 函数中创建一个新的 ActivityThread 对象，并且启动消息循环（UI 线程），创建新的 Activity、新的 Context 对象，然后将该 Context 对象传递给 Activity。下面看看 ActivityThread 源代码：

```
public static void main(String[] args) {
// 代码省略
        Process.setArgV0("<pre-initialized>");
        // 主线程消息循环
        Looper.prepareMainLooper();
        // 创建 ActivityThread 对象
        ActivityThread thread = new ActivityThread();
        thread.attach(false);

        if (sMainThreadHandler == null) {
            sMainThreadHandler = thread.getHandler();
        }

        AsyncTask.init();
        // 代码省略
        Looper.loop();
    }

    private void attach(boolean system) {
        sThreadLocal.set(this);
        mSystemThread = system;
        // 不是系统应用的情况
        if (!system) {
            ViewRootImpl.addFirstDrawHandler(new Runnable() {
                public void run() {
                    ensureJitEnabled();
                }
            });

android.ddm.DdmHandleAppName.setAppName("<pre-initialized>",UserHandle.myUserId());
            RuntimeInit.setApplicationObject(mAppThread.asBinder());
            IActivityManager mgr = ActivityManagerNative.getDefault();
            try {
                // 关联 mAppThread
                mgr.attachApplication(mAppThread);
            } catch (RemoteException ex) {
                // 省略
            }
        } else {
            // 省略
        }
}
```

在 main 方法中，我们创建一个 ActivityThread 对象后，调用了其 attach 函数，并且参数为 false。在 attach 函数中，参数为 false 的情况下（即非系统应用），会通过 Binder 机制与 ActivityManager Service 通信，并且最终调用 handleLaunchActivity 函数，我们看看该函数的实现：

```
private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
        // 代码省略
        Activity a = performLaunchActivity(r, customIntent);
        // 代码省略
    }

    private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
        // 代码省略
        Activity activity = null;
        try {
            java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
            activity = mInstrumentation.newActivity(        // 1. 创建 Activity
            cl, component.getClassName(), r.intent);
         // 代码省略
        } catch (Exception e) {
         // 省略
        }

        try {
        // 创建 Application 对象
            Application app = r.packageInfo.makeApplication(false, mInstrumentation);
            if (activity != null) {
                Context appContext = createBaseContextForActivity(r, activity);
                // 2. 获取 Context 对象
                CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());
                Configuration config = new Configuration(mCompatConfiguration);
                // 3. 将 appContext 等对象 attach 到 activity 中
                activity.attach(appContext, this, getInstrumentation(), r.token,
                r.ident, app, r.intent, r.activityInfo, title, r.parent,
                r.embeddedID, r.lastNonConfigurationInstances, config);

                // 代码省略
                // 4. 调用 Activity 的 onCreate 方法
                mInstrumentation.callActivityOnCreate(activity, r.state);
                // 代码省略
        } catch (SuperNotCalledException e) {
            throw e;
        } catch (Exception e) {
            // 代码省略
        }

        return activity;
    }


    private Context createBaseContextForActivity(ActivityClientRecord r,
            final Activity activity) {
        // 5. 创建 Context 对象，可以看到实现类是 ContextImpl
 ContextImpl appContext = ContextImpl.createActivityContext(this, r.packageInfo, r.token);
        appContext.setOuterContext(activity);
        Context baseContext = appContext;
        // 代码省略
        return baseContext;
    }
```

通过上面 1~5 注释处的代码分析可以知道，Context 的实现类为 ComtextImpl。我们继续跟踪 ContextImpl 类：

```
class ContextImpl extends Context {

// 代码省略
// ServiceFetcher 通过 getService 获取服务对象
    static class ServiceFetcher {
      int mContextCacheIndex = -1;

      // 获取系统服务
      public Object getService(ContextImpl ctx) {
         ArrayList<Object> cache = ctx.mServiceCache;
         Object service;
         synchronized (cache) {
            if (cache.size() == 0) {
               for (int i = 0; i < sNextPerContextServiceCacheIndex; i++) {
                  cache.add(null);
               }
            } else {
               service = cache.get(mContextCacheIndex); // 从缓存中获取 Service 对象
               if (service != null) {
                  return service;
               }
            }
            service = createService(ctx);
            cache.set(mContextCacheIndex, service);
            return service;
         }
      }

      /**
       * 子类覆写该方法用以创建服务对象
       */
      public Object createService(ContextImpl ctx) {
         throw new RuntimeException("Not implemented");
      }
   }

   // 1. Service 容器
   private static final HashMap<String, ServiceFetcher> SYSTEM_SERVICE_MAP =
         new HashMap<String, ServiceFetcher>();

   private static int sNextPerContextServiceCacheIndex = 0;
   // 2. 注册服务器
   private static void registerService(String serviceName, ServiceFetcher fetcher) {
      if (!(fetcher instanceof StaticServiceFetcher)) {
         fetcher.mContextCacheIndex = sNextPerContextServiceCacheIndex++;
      }
      SYSTEM_SERVICE_MAP.put(serviceName, fetcher);
   }

   // 3. 静态语句块，第一次加载该类时执行（只执行一次，保证实例的唯一性）
   static {
      //  代码省略
      // 注册 LayoutInflater service
      registerService(LAYOUT_INFLATER_SERVICE, new ServiceFetcher() {
            public Object createService(ContextImpl ctx) {
               return PolicyManager.makeNewLayoutInflater(ctx.getOuterContext());
            }});
```

```
        // 代码省略
    }

    // 4. 根据 key 获取对应的服务
    @Override
    public Object getSystemService(String name) {
        // 根据 name 来获取服务
        ServiceFetcher fetcher = SYSTEM_SERVICE_MAP.get(name);
        return fetcher == null ? null : fetcher.getService(this);
    }

    // 代码省略
}
```

从 ContextImpl 类的部分代码中可以看到，在虚拟机第一次加载该类时会注册各种 ServiceFatcher，其中就包含了 LayoutInflater Service。将这些服务以键值对的形式存储在一个 HashMap 中，用户使用时只需要根据 key 来获取到对应的 ServiceFetcher，然后通过 ServiceFetcher 对象的 getService 函数来获取具体的服务对象。当第一次获取时，会调用 ServiceFetcher 的 createService 函数创建服务对象，然后将该对象缓存到一个列表中，下次再取时直接从缓存中获取，避免重复创建对象，从而达到单例的效果。这种模式就是 2.6.5 小节中通过容器的单例模式实现方式，系统核心服务以单例形式存在，减少了资源消耗。

2.8 无名英雄——深入理解 LayoutInflater

LayoutInflater 在我们的开发中扮演着重要的角色，但很多时候我们都不知道它的重要性，因为它的重要性被隐藏在了 Activity、Fragment 等组件的光环之下。

LayoutInflater 是一个抽象类，具体代码如下：

```
public abstract class LayoutInflater {
    // 代码省略
}
```

既然是抽象不是具体的，那我们必须把这个深藏功与名的“家伙”找出来！需要先从 layoutInflater 的起源开始。在上文中知道，在加载 ContenxtImpl 时会通过如下代码将 LayoutInflater 的 ServiceFetcher 注入到容器中，具体代码如下：

```
registerService(LAYOUT_INFLATER_SERVICE, new ServiceFetcher() {
    public Object createService(ContextImpl ctx) {
        return PolicyManager.makeNewLayoutInflater(ctx.getOuterContext());
        }
});
```

这里调用了 PolicyManager. makeNewLayoutInflater 方法，继续往下看看：

```
public final class PolicyManager {
    // Policy 实现类
    private static final String POLICY_IMPL_CLASS_NAME =
        "com.android.internal.policy.impl.Policy";

    private static final IPolicy sPolicy;
```

```
    static {
        // 通过反射构造 Policy 对象
        try {
            Class policyClass = Class.forName(POLICY_IMPL_CLASS_NAME);
            sPolicy = (IPolicy)policyClass.newInstance();
        }
        // catch 代码段
}
    private PolicyManager() {}
    // 这里就是创建 PhoneWindow 对象的地方
    public static Window makeNewWindow(Context context) {
        return sPolicy.makeNewWindow(context);
    }

    // 通过 sPolicy 创建 LayoutInflater
    public static LayoutInflater makeNewLayoutInflater(Context context) {
        return sPolicy.makeNewLayoutInflater(context);
    }
}
```

PolicyManager 中通过反射构造了 Policy 实现类，这个类实现了 IPolicy 接口，通过这种形式将具体的 Policy 类对外进行隐藏实现。PolicyManager 实际上是一个代理类，具体的功能通过 sPolicy 对象进行实现，我们看看 sPolicy 对应的 Policy 类，也就是 com.android.internal.policy.impl.Policy：

```
public class Policy implements IPolicy {
    // 代码省略
    // 创建 PhoneWindow，这就是 Activity 中 Window 的具体实现类
    public Window makeNewWindow(Context context) {
        return new PhoneWindow(context);
    }

    // 创建 LayoutInflater,具体类为 PhoneLayoutInflater,这才是我们要关注的地方
    public LayoutInflater makeNewLayoutInflater(Context context) {
        return new PhoneLayoutInflater(context);
    }
}
```

此时，已经很清楚了，真正 LayoutInflater 的实现类就是 PhoneLayoutInflater。我们继续深入看看 PhoneLayoutInflater 的源代码：

```
public class PhoneLayoutInflater extends LayoutInflater {
    // 内置 View 类型的前缀，如 TextView 的完整路径是 android.widget.TextView
    private static final String[] sClassPrefixList = {
        "android.widget.",
        "android.webkit."
    };

    // 代码省略
    @Override protected View onCreateView(String name, AttributeSet attrs) throws ClassNot
FoundException {
        // 在 View 名字的前面添加前缀来构造 View 的完整路径，例如，类名为 TextView，那么 TextView 完整
        //路径是 android.widget.TextView
        for (String prefix : sClassPrefixList) {
            try {
                View view = createView(name, prefix, attrs);
                if (view != null) {
                    return view;
                }
            } catch (ClassNotFoundException e) {
```

```
            // 省略
        }
    }
    return super.onCreateView(name, attrs);
  }
}
```

代码不多，核心的程序语句就是覆写了 LayoutInflater 的 onCreateView 方法，该方法就是在传递进来的 View 名字前面加上“android.widget.”或者“android.webkit.”前缀用以得到该内置 View 类（如 TextView、Button 等都在 android.widget 包下）的完整路径。最后，根据类的完整路径来构造对应的 View 对象。

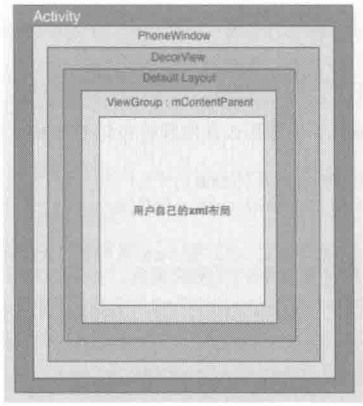
具体是一个怎样的流程呢？以 Activity 的 setContentView 为例，先来看看这个函数的实现：

```
public void setContentView(View view) {
    getWindow().setContentView(view);
    initActionBar();
}
```

Activity 的 setContentView 方法实际上调用的是 Window 的 setContentView，而 Window 是一个抽象类，上文提到 Window 的具体实现类是 PhoneWindow，我们看看 PhoneWindow 中对应的方法：

```
 @Override
public void setContentView(int layoutResID) {
        // 1. 当 mContentParent 为空时先构建 DecorView
        // 并且将 DecorView 包裹到 mContentParent 中
        if (mContentParent == null) {
            installDecor();
        } else {
            mContentParent.removeAllViews();
        }
        // 2. 解析 layoutResID
        mLayoutInflater.inflate(layoutResID, mContentParent);
    // 代码省略
    }
```

在分析之前，我们来看看一个 Window 的 View 层级图，如图 2-2 所示。



▲图 2-2

从图 2-2 中，我们看到 mDecor 中会加载一个系统定义好的布局，这个布局中又包裹了 mContentParent，而这个 mContentParent 就是我们设置的布局，并将添加到 parent 区域。在 PhoneWindow 的 setContentView 方法中也验证了这点，首先会构建 mContentParent 这个对象，然后通过 LayoutInflater 的 inflate 函数将指定布局的视图添加到 mContentParent 中。那么就先来看看 inflate 方法：

```
public View inflate(int resource, ViewGroup root) {
        // root 不为空，则会从 resource 布局解析到 View，并添加到 root 中
        return inflate(resource, root, root != null);
}

public View inflate(int resource, ViewGroup root, boolean attachToRoot) {
        // 获取 xml 资源解析器
    XmlResourceParser parser = getContext().getResources().getLayout(resource);
    try {
            return inflate(parser, root, attachToRoot);
     } finally {
            parser.close();
        }
    }

// 参数 1 为 xml 解析器，参数 2 为要解析布局的父视图，参数 3 为是否将要解析的视图添加到父视图中
public View inflate(XmlPullParser parser, ViewGroup root, boolean attachToRoot) {
        synchronized (mConstructorArgs) {
            final AttributeSet attrs = Xml.asAttributeSet(parser);
            Context lastContext = (Context)mConstructorArgs[0];
            // Context 对象
            mConstructorArgs[0] = mContext;
            // 存储父视图
            View result = root;
            try {
                // Look for the root node.
                int type;
                // 找到 root 元素
                while ((type = parser.next()) != XmlPullParser.START_TAG &&
                        type != XmlPullParser.END_DOCUMENT) {
                    // Empty
                }
                // 代码省略
                final String name = parser.getName();
                // 1. 解析 merge 标签
                if (TAG_MERGE.equals(name)) {
                    rInflate(parser, root, attrs, false);
                } else {
                    // 2. 不是 merge 标签那么直接解析布局中的视图
                    View temp;
                    if (TAG_1995.equals(name)) {
                        temp = new BlinkLayout(mContext, attrs);
                    } else {
                        // 3. 这里就是通过 xml 的 tag 来解析 layout 根视图
                        // name 就是要解析的视图的类名，如 RelativeLayout
                        temp = createViewFromTag(root, name, attrs);
                    }

                    ViewGroup.LayoutParams params = null;

                    if (root != null) {
                        // 生成布局参数
```

```
                params = root.generateLayoutParams(attrs);
                // 如果 attachToRoot 为 false,那么将给 temp 设置布局参数
                if (!attachToRoot) {
                    temp.setLayoutParams(params);
                }
            }
            // 解析 temp 视图下的所有子 View
            rInflate(parser, temp, attrs, true);
            // 如果 Root 不为空，且 attachToRoot 为 true，那么将 temp 添加到父视图中
            if (root != null && attachToRoot) {
                root.addView(temp, params);
            }
            // 如果 root 为空或者 attachToRoot 为 false,那么返回的结果就是 temp
            if (root == null || !attachToRoot) {
                result = temp;
            }
        }
    }
    // 省略 catch, finaly 代码
    return result;
}
```

上述的 inflate 方法中，主要有下面几步:

（1）解析 xml 中的根标签（第一个元素）;

（2）如果根标签是 merge，那么调用 rInflate 进行解析，rInflate 会将 merge 标签下的所有子 View 直接添加到根标签中;

（3）如果标签是普通元素，那么运行到代码 3，调用 createViewFromTag 对该元素进行解析;

（4）调用 rInflate 解析 temp 根元素下的所有子 View，并且将这些子 View 都添加到 temp 下;

（5）返回解析到的根视图。

我们先从简单的地方理解，即解析单个元素的 createViewFromTag，看看如下代码:

```
View createViewFromTag(View parent, String name, AttributeSet attrs) {
    if (name.equals("view")) {
        name = attrs.getAttributeValue(null, "class");
    }
    try {
        View view;
        // 1. 用户可以通过设置 LayoutInflater 的 factory 来自行解析 View，默认这些 Factory 都为
        //空，可以忽略这段
        if (mFactory2 != null) view = mFactory2.onCreateView(parent, name, mContext, attrs);
        else if (mFactory != null) view = mFactory.onCreateView(name, mContext, attrs);
        else view = null;
        // 代码省略
        // 2. 没有 Factory 的情况下通过 onCreateView 或者 createView 创建 View
        if (view == null) {
            // 3. 内置 View 控件的解析
            if (-1 == name.indexOf('.')) {
                view = onCreateView(parent, name, attrs);
            } else {
                // 4. 自定义控件的解析
                view = createView(name, null, attrs);
            }
        }
        return view;

    }
```

```
        // 省略 catch 块
    }
```

本程序重点就在代码 2，以及以后的代码，createViewFromTag 会将该元素的 parent 及名字传递过来。当这个 tag 的名字中没有包含“.”（在名字中查找“.”返回-1）时，LayoutInflater 会认为这是一个内置的 View，例如，我们在 xml 中声明一个内置 View 时大概是这样的：

```
<TextView
      android:id="@+id/my_textview "
      android:layout_width="60dp"
   android:layout_height="60dp" />
```

这里的 TextView 就是 xml 元素的名字，因此，在执行 infate 时就会调用代码 3 处的 onCreateView 来解析这个 TextView 标签。当我们自定义 View 时，在 xml 中必须写 View 的完整路径，例如：

```
<com.dp.custom.MyView
          android:id="@+id/my_custom_view "
          android:layout_width="fill_parent"
      android:layout_height="fill_parent" />
```

此时，就会调用代码注释 4 的 createView 来解析该 View。为什么要这么处理，它们之间又有什么不同呢？

在上文的 PhoneLayoutInflater 中我们知道，PhoneLayoutInflater 覆写了 onCreateView 方法，也就是代码 3 处的 onCreateView，该方法就是在 View 标签名的前面设置一个“android.widget.”前缀，然后再传递给 createView 进行解析。也就是说内置 View 和自定义 View 最终都调用了 createView 进行解析，只是 Google 为了让开发者在 xml 中更方便定义 View，只写 View 名称而不需要写完整的路径。在 LayoutInflater 解析时如果遇到只写类名的 View，那么认为是内置的 View 控件，在 onCreateView 方法中会将“android.widget.”前缀传递给 createView 方法，最后，在 createView 中构造 View 的完整路径来进行解析。如果是自定义控件，那么必须写完整的路径，此时调用 createView 且前缀为 null 进行解析。

关于 createView 的解释已经有很多，我们还是看下面的代码吧：

```
// 根据完整路径的类名通过反射机制构造 View 对象
   public final View createView(String name, String prefix, AttributeSet attrs)
         throws ClassNotFoundException, InflateException {
      // 1. 从缓存中获取构造函数
      Constructor<? extends View> constructor = sConstructorMap.get(name);
      Class<? extends View> clazz = null;
      try {
         // 2. 没有缓存构造函数
         if (constructor == null) {
            // 如果 prefix 不为空，那么构造完整的 View 路径，并且加载该类
            clazz = mContext.getClassLoader().loadClass(
                  prefix != null ? (prefix + name) : name).asSubclass(View.class);
            // 代码省略
            // 3. 从 Class 对象中获取构造函数
            constructor = clazz.getConstructor(mConstructorSignature);
            // 4. 将构造函数存入缓存中
            sConstructorMap.put(name, constructor);
         } else {
            // 代码省略
         }
```

```
            Object[] args = mConstructorArgs;
            args[1] = attrs;
            // 5. 通过反射构造 View
            final View view = constructor.newInstance(args);
            if (view instanceof ViewStub) {
                // always use ourselves when inflating ViewStub later
                final ViewStub viewStub = (ViewStub) view;
                viewStub.setLayoutInflater(this);
            }
            return view;

        }
        // 省略各种 catch、finaly 代码
    }
```

createView 相对比较简单，如果有前缀，那么构造 View 的完整路径，并且将该类加载到虚拟机中，然后获取该类的构造函数并且缓存起来，再通过构造函数来创建该 View 的对象，最后将 View 对象返回，这就是解析单个 View 的过程。而我们的窗口中是一棵视图树，LayoutInflater 需要解析完这棵树，这个功能就交给了 rInflate 方法，具体代码如下：

```
void rInflate(XmlPullParser parser, View parent, final AttributeSet attrs,
            boolean finishInflate) throws XmlPullParserException, IOException {
        // 1. 获取树的深度，深度优先遍历
        final int depth = parser.getDepth();
        int type;
        // 2. 挨个元素解析
        while (((type = parser.next()) != XmlPullParser.END_TAG || parser.getDepth() > depth)
         && type != XmlPullParser.END_DOCUMENT) {
            if (type != XmlPullParser.START_TAG) {
                continue;
            }
            final String name = parser.getName();
            if (TAG_REQUEST_FOCUS.equals(name)) {
                parseRequestFocus(parser, parent);
            } else if (TAG_INCLUDE.equals(name)) {  // 解析 include 标签
                parseInclude(parser, parent, attrs);
            } else if (TAG_MERGE.equals(name)) { // 解析 merge 标签，抛出异常，因为 merge 标签
                                                 //必须为根视图
                throw new InflateException("<merge /> must be the root element");
            } else if (TAG_1995.equals(name)) {// 闪烁视图,这里可以不用管
// 代码省略
            } else {
                // 3. 根据元素名进行解析
                final View view = createViewFromTag(parent, name, attrs);
                final ViewGroup viewGroup = (ViewGroup) parent;
                final ViewGroup.LayoutParams params = viewGroup.generateLayoutParams (attrs);
                // 递归调用进行解析，也就是深度优先遍历
                rInflate(parser, view, attrs, true);
                // 将解析到的 View 添加到 viewGroup 中，也就是它的 parent
                viewGroup.addView(view, params);
            }
        }
        if (finishInflate) parent.onFinishInflate();
    }
```

rInflate 通过深度优先遍历来构造视图树，每解析到一个 View 元素就会递归调用 rInflate，直到这条路径下的最后一个元素，然后再回溯过来将每个 View 元素添加到它们的 parent 中。通过 rInflate 的解析之后，整棵视图树就构建完毕。当调用了 Activity 的 onResume 之后，我们通过 setContentView

设置的内容就会出现在我们的视野中。

2.9 运用单例模式

在 Android 应用开发过程中，ImageLoader 是我们最为常用的开发工具库之一。Android 中最著名的 ImageLoader 就是 Universal-Image-Loader（https://github.com/nostra13/Android-Universal-Image-Loader），它的使用过程大概是这样的：

```
public  void initImageLoader(Context context) {
    // 1. 使用 Builder 构建 ImageLoader 的配置对象
    ImageLoaderConfiguration config = new ImageLoaderConfiguration.Builder(context)
        //加载图片的线程数
        .threadPriority(Thread.NORM_PRIORITY - 2)
        //解码图像的大尺寸，将在内存中缓存先前解码图像的小尺寸
        .denyCacheImageMultipleSizesInMemory()
        //设置磁盘缓存文件名称
        .discCacheFileNameGenerator(new Md5FileNameGenerator())
        //设置加载显示图片队列进程
        .tasksProcessingOrder(QueueProcessingType.LIFO)
        .writeDebugLogs()
        .build();
    // 2. 使用配置对象初始化 ImageLoader
    ImageLoader.getInstance().init(config);
    // 3. 加载图片
    ImageLoader.getInstance().displayImage("图片 url", myImageView);
}
```

代码中出现了熟悉的 getInstance()方法，因此，我们猜测这个 ImageLoader 使用的是单例模式。正好，小民的 ImageLoader 也是类似的实现，通过一个 getInstance 函数返回单例对象，具体代码如下：

```
public final class ImageLoader{
// ImageLoader 实例
    private static ImageLoadersInstance;
//网络请求队列
    private RequestQueue mImageQueue;
    // 缓存
    private volatile BitmapCache mCache = new MemoryCache();
    // 图片加载配置对象
    private ImageLoaderConfig mConfig;
    // 私有构造函数
    private ImageLoader() {
    }

    /**
     * 获取 ImageLoader 单例，DCL 形式
     * @return 单例对象
     */
    public static ImageLoadergetInstance() {
        if (sInstance == null) {
            synchronized (ImageLoader.class) {
                if (sInstance == null) {
                    sInstance = new ImageLoader();
                }
            }
        }
        return sInstance;
```

```
    }

    /**通过配置类初始化 ImageLoader，设置线程数量、缓存策略、加载策略等
     * @param config 配置对象
     */
    public void init(ImageLoaderConfig config) {
        mConfig = config;
        mCache = mConfig.bitmapCache;
        checkConfig();
        mImageQueue = new RequestQueue(mConfig.threadCount);
        mImageQueue.start();
    }
    // 代码省略
    // 加载图片的接口
    public void displayImage(ImageView imageView, String uri) {
        displayImage(imageView, uri, null, null);
    }

    public void displayImage(ImageView imageView, String uri, ImageListener listener) {
        displayImage(imageView, uri, null, listener);
    }

    public void displayImage(final ImageView imageView, final String uri,
    final DisplayConfig config, final ImageListener listener) {
        BitmapRequest request = new BitmapRequest(imageView, uri, config, listener);
        // 加载的配置对象,如果没有设置，则使用 ImageLoader 的配置
        request.displayConfig = request.displayConfig != null ? request.displayConfig
                : mConfig.displayConfig;
        // 添加到队列中
        mImageQueue.addRequest(request);
    }
    public void stop() {
        mImageQueue.stop();
    }

    // 图片加载 Listener，加载完成后回调给客户端代码
    public static interface ImageListener {
        public void onComplete(ImageView imageView, Bitmap bitmap, String uri);
    }
}
```

我们的 ImageLoader 类中将构造函数私有化，并且使用 Double Check Lock 的形式实现单例，用户通过 getInstance 方法获取 ImageLoader 单例对象。在用户使用之前需要使用 ImageLoaderConfig 来配置 ImageLoader，配置合理的情况下才会启动用户指定数量的线程来执行图片加载请求。当用户调用 displayImage 方法时，ImageLoader 会将请求构造成一个 BitmapRequest，然后将该请求添加到请求队列中，图片加载线程（RequestDispatcher）会从请求队列（RequestQueue）中获取图片加载请求，然后加载该图片，并且将图片显示到对应的 ImageView 上，最后将图片缓存到缓存系统中。

在后续章节中将会阐述关于该 ImageLoader 的更多细节，大家可以到 github（https://github.com/bboyfeiyu/simple_imageloader）下载该库的源代码，并且结合《教你写 Android ImageLoader 框架》系列博文（地址为 http://blog.csdn.net/column/details/android-imageloader.html）进行学习。

2.10 总结

单例模式是运用频率很高的模式，但是，由于在客户端通常没有高并发的情况，因此，选择哪种实现方式并不会有太大的影响。即便如此，出于效率考虑，我们推荐用 2.6.2 小节、2.6.3 小节使用的形式。

优点

（1）由于单例模式在内存中只有一个实例，减少了内存开支，特别是一个对象需要频繁地创建、销毁时，而且创建或销毁时性能又无法优化，单例模式的优势就非常明显。

（2）由于单例模式只生成一个实例，所以，减少了系统的性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后用永久驻留内存的方式来解决。

（3）单例模式可以避免对资源的多重占用，例如一个写文件操作，由于只有一个实例存在内存中，避免对同一个资源文件的同时写操作。

（4）单例模式可以在系统设置全局的访问点，优化和共享资源访问，例如，可以设计一个单例类，负责所有数据表的映射处理。

缺点

（1）单例模式一般没有接口，扩展很困难，若要扩展，除了修改代码基本上没有第二种途径可以实现。

（2）单例对象如果持有 Context，那么很容易引发内存泄漏，此时需要注意传递给单例对象的 Context 最好是 Application Context。

第3章 自由扩展你的项目——Builder模式

3.1 Builder模式介绍

Builder 模式是一步一步创建一个复杂对象的创建型模式，它允许用户在不知道内部构建细节的情况下，可以更精细地控制对象的构造流程。该模式是为了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隔离开来。

因为一个复杂的对象有很多大量组成部分，如汽车，有车轮、方向盘、发动机，还有各种小零件等，如何将这些部件装配成一辆汽车，这个装配过程很漫长，也很复杂，对于这种情况，为了在构建过程中对外部隐藏实现细节，就可以使用 Builder 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扩展，两者之间的耦合也降到最低。

3.2 Builder模式的定义

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。

3.3 Builder模式的使用场景

（1）相同的方法，不同的执行顺序，产生不同的事件结果时。

（2）多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时。

（3）产品类非常复杂，或者产品类中的调用顺序不同产生了不同的作用，这个时候使用建造者模式非常合适。

（4）当初始化一个对象特别复杂，如参数多，且很多参数都具有默认值时。

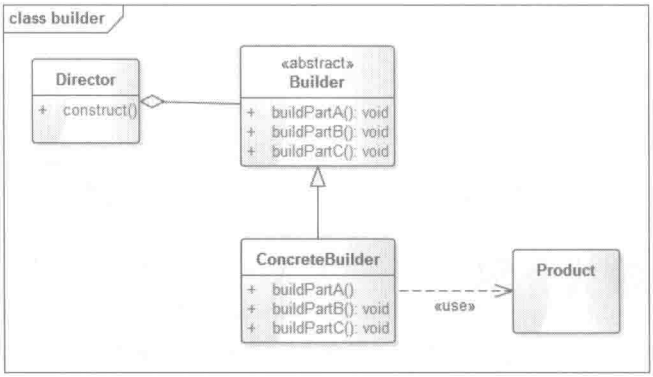
3.4 Builder模式的UML类图

UML 类图如图 3-1 所示。

角色介绍：

- Product 产品类——产品的抽象类；

- Builder——抽象 Builder 类，规范产品的组建，一般是由子类实现具体的组建过程；
- ConcreteBuilder——具体的 Builder 类；
- Director——统一组装过程。



▲图 3-1

3.5 Builder 模式的简单实现

计算机的组装过程较为复杂，并且组装顺序是不固定的，为了易于理解，我们把计算机组装的过程简化为构建主机、设置操作系统、设置显示器 3 个部分，然后通过 Director 和具体的 Builder 来构建计算机对象。请看下面的示例：

```
package com.dp.example.builder;

// 计算机抽象类，即 Product 角色
public abstract class Computer {
    protected String mBoard;
    protected String mDisplay;
    protected String mOS;

    protected Computer() {
    }
    // 设置 CPU 核心数
    public void setBoard(String board) {
        mBoard = core;
    }
    // 设置内存
    public void setDisplay(String display) {
        mDisplay = gb;
    }
    // 设置操作系统
    public abstract void setOS();

    @Override
    public String toString() {
        return "Computer [mBoard=" + mBoard + ", mDisplay=" + mDisplay
                + ", mOS=" + mOS + "]";
    }
```

```
}
// 具体的 Computer 类，Macbook
public class Macbook extends Computer {

    protected Macbook() {
    }

    @Override
    public void setOS() {
        mOS = "Mac OS X 10.10";
    }

}
// 抽象 Builder 类
public abstract class Builder {
    // 设置主机
    public abstract void buildBoard(String board);
    // 设置显示器
    public abstract void buildDisplay(String  display);
    // 设置操作系统
    public abstract void buildOS();
    // 创建 Computer
    public abstract Computer create();

}
// 具体的 Builder 类，MacbookBuilder
public class MacbookBuilder extends Builder {
    private Computer mComputer = new Macbook();
    @Override
    public void buildBoard(String board) {
        mComputer.setBoard(core);
    }
    @Override
    public void buildDisplay(String display) {
        mComputer.setDisplay(display);
    }
    @Override
    public void buildOS() {
        mComputer.setOs();
    }
    @Override
    public Computer create() {
        return mComputer;
    }

}
// Director 类，负责构造 Computer
public class Director {
    Builder mBuilder = null;
    /**
     * @param builder
     */
    public Director(Builder builder) {
        mBuilder = builder;
    }
    /**
     * 构建对象
     */
    public void construct(String board, String display) {
       mBuilder.buildBoard(board);
       mBuilder.buildDisplay(display);
       mBuilder.buildOS();
```

```
    }
}
// 测试代码
public class Test {
      public static void main(String[] args) {
         // 构建器
         Builder builder = new MacbookBuilder();
         // Director
         Director pcDirector = new Director(builder);
         // 封装构建过程，4 核、内存 2GB、Mac 系统
         pcDirector.construct("英特尔主板", "Retina 显示器");
         // 构建计算机，输出相关信息
         System.out.println("Computer Info : " + builder.create().toString());
    }
}
```

输出结果：

```
Computer Info : Computer [mBoard=英特尔主板, mDisplay=Retina 显示器, mOS=Mac OS X 10.10]
```

上述示例中，通过具体的 MacbookBuilder 来构建 Macbook 对象，而 Director 封装了构建复杂产品对象的过程，对外隐藏构建细节。Builder 与 Director 一起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对象。

值得注意的是，在现实开发过程中，Director 角色经常会被省略。而直接使用一个 Builder 来进行对象的组装，这个 Builder 通常为链式调用，它的关键点是每个 setter 方法都返回自身，也就是 return this，这样就使得 setter 方法可以链式调用，代码大致如下：

```
new TestBuilder().setA("A").setB("B").create();
```

通过这种形式不仅去除了 Director 角色，整个结构也更加简单，也能对 Product 对象的组装过程有更精细的控制。

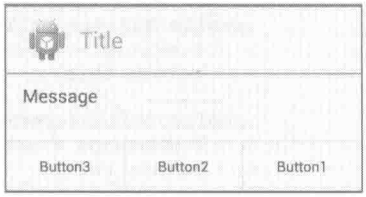
3.6 Android 源码中的 Builder 模式实现

在 Android 源码中，最常用到的 Builder 模式就是 AlertDialog.Builder，使用该 Builder 来构建复杂的 AlertDialog 对象。在开发过程中，我们经常用到 AlertDialog，具体示例如下：

```
//显示基本的 AlertDialog
private void showDialog(Context context) {
      AlertDialog.Builder builder = new AlertDialog.Builder(context);
      builder.setIcon(R.drawable.icon);
      builder.setTitle("Title");
      builder.setMessage("Message");
      builder.setPositiveButton("Button1",
      new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int whichButton) {
                     setTitle("点击了对话框上的 Button1");
                  }
               });
      builder.setNeutralButton("Button2",
      new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int whichButton) {
                     setTitle("点击了对话框上的 Button2");
```

```
                }
            });
        builder.setNegativeButton("Button3",
        new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
                setTitle("点击了对话框上的Button3");
                }
            });
        builder.create().show();  // 构建AlertDialog，并且显示
}
```

显示结果如图 3-2 所示。



▲图 3-2

从类名就可以看出这就是一个 Builder 模式，通过 Builder 对象来组装 Dialog 的各个部分，如 title、buttons、Message 等，将 Dialog 的构造和表示进行分离。下面看看 AlertDialog 的相关源码：

```
// AlertDialog
public class AlertDialog extends Dialog implements DialogInterface {
    // AlertController 接收Builder成员变量P中的各个参数
    private AlertController mAlert;

    // 构造函数
    protected AlertDialog(Context context, int theme) {
        this(context, theme, true);
    }

    // 构造AlertDialog
    AlertDialog(Context context, int theme, boolean createContextWrapper) {
        super(context, resolveDialogTheme(context, theme), createContextWrapper);
        mWindow.alwaysReadCloseOnTouchAttr();
        // 构造AlertController
        mAlert = new AlertController(getContext(), this, getWindow());
    }

    // 实际上调用的是mAlert的setTitle方法
    @Override
    public void setTitle(CharSequence title) {
        super.setTitle(title);
        mAlert.setTitle(title);
    }

    // 实际上调用的是mAlert的setCustomTitle方法
    public void setCustomTitle(View customTitleView) {
        mAlert.setCustomTitle(customTitleView);
    }

    public void setMessage(CharSequence message) {
        mAlert.setMessage(message);
    }

    // AlertDialog其他的代码省略

    // ************  Builder为AlertDialog的内部类   *******************
    public static class Builder {
        // 1. 存储AlertDialog的各个参数，如title、message、icon等
        private final AlertController.AlertParams P;
        // 属性省略

        public Builder(Context context) {
```

```
            this(context, resolveDialogTheme(context, 0));
        }

        public Builder(Context context, int theme) {
            P = new AlertController.AlertParams(new ContextThemeWrapper(
                context, resolveDialogTheme(context, theme)));
            mTheme = theme;
        }
        // Builder 的其他代码省略
        // 2. 设置各种参数
        public Builder setTitle(CharSequence title) {
            P.mTitle = title;
            return this;
        }
        public Builder setMessage(CharSequence message) {
            P.mMessage = message;
            return this;
        }
        public Builder setView(View view) {
            P.mView = view;
            P.mViewSpacingSpecified = false;
            return this;
        }
        // 3. 构建 AlertDialog, 传递参数
        public AlertDialog create() {
            // 4. 调用 new AlertDialog 构造对象，并且将参数传递给个体 AlertDialog
            final AlertDialog dialog = new AlertDialog(P.mContext, mTheme, false);
            // 5. 将 P 中的参数应用到 dialog 中的 mAlert 对象中
            P.apply(dialog.mAlert);
            dialog.setCancelable(P.mCancelable);
            if (P.mCancelable) {
                dialog.setCanceledOnTouchOutside(true);
            }
            dialog.setOnCancelListener(P.mOnCancelListener);
            if (P.mOnKeyListener != null) {
                dialog.setOnKeyListener(P.mOnKeyListener);
            }
            return dialog;
        }
    }
}
```

上述代码中，Builder 类可以设置 AlertDialog 中的 title、message、button 等参数，这些参数都存储在类型为 AlertController.AlertParams 的成员变量 P 中，AlertController.AlertParams 中包含了与 AlertDialog 视图中对应的成员变量。在调用 Builder 类的 create 函数时会创建 AlertDialog，并且将 Builder 成员变量 P 中保存的参数应用到 AlertDialog 的 mAlert 对象中，即 P.apply(dialog.mAlert)代码段。我们再看看 apply 函数的实现：

```
public void apply(AlertController dialog) {
    if (mCustomTitleView != null) {
        dialog.setCustomTitle(mCustomTitleView);
    } else {
        if (mTitle != null) {
            dialog.setTitle(mTitle);
        }
        if (mIcon != null) {
            dialog.setIcon(mIcon);
        }
        if (mIconId >= 0) {
```

```
            dialog.setIcon(mIconId);
        }
        if (mIconAttrId > 0) {
            dialog.setIcon(dialog.getIconAttributeResId(mIconAttrId));
        }
    }
    if (mMessage != null) {
        dialog.setMessage(mMessage);
    }
    if (mPositiveButtonText != null) {
        dialog.setButton(DialogInterface.BUTTON_POSITIVE, mPositiveButtonText,
        mPositiveButtonListener, null);
    }
    if (mNegativeButtonText != null) {
        dialog.setButton(DialogInterface.BUTTON_NEGATIVE, mNegativeButtonText,
        mNegativeButtonListener, null);
    }
    if (mNeutralButtonText != null) {
        dialog.setButton(DialogInterface.BUTTON_NEUTRAL, mNeutralButtonText,
        mNeutralButtonListener, null);
    }
    if (mForceInverseBackground) {
        dialog.setInverseBackgroundForced(true);
    }
    // 如果设置了 mItems，则表示是单选或者多选列表，此时创建一个 ListView
    if ((mItems != null) || (mCursor != null) || (mAdapter != null)) {
        createListView(dialog);
    }
    // 将 mView 设置给 Dialog
    if (mView != null) {
        if (mViewSpacingSpecified) {
            dialog.setView(mView, mViewSpacingLeft, mViewSpacingTop, mViewSpacingRight,
                mViewSpacingBottom);
        } else {
            dialog.setView(mView);
        }
    }
}
```

在 apply 函数中，只是将 AlertParams 参数设置到 AlertController 中，例如，将标题设置到 Dialog 对应的标题视图中，将 Message 设置到内容视图中等。当我们获取到 AlertDialog 对象后，通过 show 函数就可以显示这个对话框。我们看看 Dialog 的 show 函数（该函数在 Dialog 类中）：

```
// 显示 Dialog
public void show() {
    // 已经是显示状态，则 return
    if (mShowing) {
        if (mDecor != null) {
            if (mWindow.hasFeature(Window.FEATURE_ACTION_BAR)) {
                mWindow.invalidatePanelMenu(Window.FEATURE_ACTION_BAR);
            }
            mDecor.setVisibility(View.VISIBLE);
        }
        return;
    }

    mCanceled = false;

    // 1. onCreate 调用
    if (!mCreated) {
```

```
        dispatchOnCreate(null);
    }

    // 2. onStart
    onStart();
    // 3. 获取 DecorView
    mDecor = mWindow.getDecorView();
        // 代码省略
    // 4. 获取布局参数
    WindowManager.LayoutParams l = mWindow.getAttributes();
    if ((l.softInputMode
        & WindowManager.LayoutParams.SOFT_INPUT_IS_FORWARD_NAVIGATION) == 0) {
            WindowManager.LayoutParams nl = new WindowManager.LayoutParams();
        nl.copyFrom(l);
        nl.softInputMode |= WindowManager.LayoutParams.SOFT_INPUT_IS_FORWARD_NAVIGATION;
        l = nl;
    }

    try {
        // 5. 将 mDecor 添加到 WindowManager 中
          mWindowManager.addView(mDecor, l);
          mShowing = true;
          // 发送一个显示 Dialog 的消息
          sendShowMessage();
        } finally {
          }
}
```

在 show 函数中主要做了如下几个事情：

（1）通过 dispatchOnCreate 函数来调用 AlertDialog 的 onCreate 函数；

（2）然后调用 AlertDialog 的 onStart 函数；

（3）最后将 Dialog 的 DecorView 添加到 WindowManager 中。

很明显，这就是一系列典型的生命周期函数。那么按照惯例，AlertDialog 的内容视图构建按理应该在 onCreate 函数中，我们来看看是不是：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
                // 调用了 AlertController 的 installContent 方法
        mAlert.installContent();
    }
```

在 onCreate 函数中主要调用了 AlertController 的 installContent 方法，Dialog 中的 onCreate 函数只是一个空实现而已，可以忽略它。那么 AlertDialog 的内容视图必然就在 installContent 函数中，继续深入了解吧：

```
public void installContent() {
        /* 设置窗口，没有 title */
    mWindow.requestFeature(Window.FEATURE_NO_TITLE);
        // 省略其他设置
        // 设置窗口的内容视图布局
    mWindow.setContentView(mAlertDialogLayout);
        // 初始化 AlertDialog 其他子视图的内容
    setupView();
    }
```

installContent 函数的代码很少，但极为重要，它调用了 Window 对象的 setContentView，这个 setContentView 就与 Activity 中的一模一样，实际上 Activity 最终也是调用 Window 对象的 setContentView 函数。因此，这里就是设置 AlertDialog 的内容布局，这个布局就是 mAlertDialogLayout 字段的值，这个值在 AlertController 的构造函数中进行了初始化，具体代码如下：

```
public AlertController(Context context, DialogInterface di, Window window) {
// 代码省略
        TypedArray a = context.obtainStyledAttributes(null,
      com.android.internal.R.styleable.AlertDialog,
      com.android.internal.R.attr.alertDialogStyle, 0);
        // AlertDialog 的布局 id，也就是 alert_dialog.xml 布局
      mAlertDialogLayout = a.getResourceId(com.android.internal.R.styleable.AlertDialog_
      layout,com.android.internal.R.layout.alert_dialog);
        // 代码省略
      a.recycle();
    }
```

从 AlertController 的构造函数中可以看到，AlertDialog 的布局资源就是 alert_dialog.xml 这个文件，由于这个布局文件有点长，我们就不附上源代码，用图 3-3 来大致描述一下它的结构。

▲图 3-3

当通过 Builder 对象的 setTitle、setMessage 等方法设置具体内容时，就是将这些内容填充到对应的视图中。而 AlertDialog 也允许你通过 setView 传入内容视图，这个内容视图就是替换掉图 3-3 的蓝色区域，AlertDialog 预留了一个 costomPanel 区域用来显示用户自定义的内容视图。我们来看看 setupView 函数：

```
private void setupView() {
    // 1. 获取并初始化内容区域
    LinearLayout contentPanel = (LinearLayout) mWindow.findViewById(R.id.contentPanel);
    setupContent(contentPanel);
    // 2. 初始化按钮
    boolean hasButtons = setupButtons();
    // 3. 获取并初始化 Title 区域
    LinearLayout topPanel = (LinearLayout) mWindow.findViewById(R.id.topPanel);
    TypedArray a = mContext.obtainStyledAttributes(
    null, com.android.internal.R.styleable.AlertDialog, com.android.internal.R.attr.
    alertDialogStyle, 0);
    boolean hasTitle = setupTitle(topPanel);
    // 按钮区域的可见性
    View buttonPanel = mWindow.findViewById(R.id.buttonPanel);
    if (!hasButtons) {
```

```
        buttonPanel.setVisibility(View.GONE);
        mWindow.setCloseOnTouchOutsideIfNotSet(true);
    }
    // 4. 自定义内容视图区域
    FrameLayout customPanel = null;
    // 如果用户设置了内容视图，那么将它显示在 customPanel 的 custom 布局里面
    if (mView != null) {
        customPanel = (FrameLayout) mWindow.findViewById(R.id.customPanel);
        FrameLayout custom = (FrameLayout) mWindow.findViewById(R.id.custom);
        // 显示用户设置的视图
        custom.addView(mView, new LayoutParams(MATCH_PARENT, MATCH_PARENT));
        if (mViewSpacingSpecified) {
            custom.setPadding(mViewSpacingLeft, mViewSpacingTop, mViewSpacingRight,
                mViewSpacingBottom);
        }
        if (mListView != null) {
            ((LinearLayout.LayoutParams) customPanel.getLayoutParams()).weight = 0;
        }
    } else {
        mWindow.findViewById(R.id.customPanel).setVisibility(View.GONE);
    }
    // 代码省略
    // 设置背景
    setBackground(topPanel, contentPanel, customPanel, hasButtons, a, hasTitle, buttonPanel);
    a.recycle();
    }
```

这个 setupView 顾名思义就是初始化 AlertDialog 布局中的各个部分，如标题区域、按钮区域、内容区域等，在该函数调用之后整个 Dialog 的视图内容全部设置完毕。而这些各区域的视图都属于 mAlertDialogLayout 布局中的子元素，Window 对象又关联了 mAlertDialogLayout 的整个布局树，当调用完 setupView 之后整个视图树的数据都填充完毕，当用户调用 show 函数时，WindowManager 会将 Window 对象的 DecorView（也就是 mAlertDialogLayout 对应的视图，当然 DecorView 还有一个层次，我们不做过多讨论）添加到用户的窗口上，并且显示出来。至此，整个 Dialog 就出现在用户的视野中了！

在 AlertDialog 的 Builder 模式中并没有看到 Director 角色的出现，其实在很多场景中，Android 并没有完全按照 GOF 在《设计模式：可复用面向对象软件的基础》一书中描述的经典模式实现来做，而是做了一些修改，使得这个模式更易于使用。这里的 AlertDialog.Builder 同时扮演了上文中提到的 builder、ConcreteBuilder、Director 的角色，简化了 Builder 模式的设计。当模块比较稳定，不存在一些变化时，可以在经典模式实现的基础上做出一些精简，而不是照搬 GOF 上的经典实现，更不要生搬硬套，使程序失去架构之美。正是由于灵活地运用设计模式，Android 的源码很值得我们去学习。

3.7 深入了解 WindowManager

在上面分析的 Dialog 的 show 函数中，Dialog 的内容视图最终是通过 WindowManager 显示到手机屏幕上的。那么 WindowManager 的背后基本原理是怎样的，这就是本章要讲述的内容。

不单是 Dialog，所有需要显示到屏幕上的内容（包括 Activity 等）都是通过 WindowManager 来操作的，看来 WindowManager 是一个非常重要的子系统，这就是我们常说的 WMS（Window

Manager Service)。在本章中，作者会带领大家来理清 WindowManager 的基本知识脉络，抛开具体的细枝末节，我们只关心 WindowManager 和 WindowManagerService（后续简称 WMS）、Surface、SurfaceFlinger 等建立关联以及交互的一个基本过程，使得大家对于 WMS 这个复杂的系统有一个最基本的认识。

与 WindowManager 联系上的第一步就是通过 Context 的 getSystemService 方法。在上一章的单例模式中我们说到，各种系统服务会注册到 ContextImpl 的一个 map 容器中，然后通过该服务的字符串键来获取，WindowManager 也是在 ContextImpl 中注册的众多服务之一，我们看看下面的程序：

```
registerService(WINDOW_SERVICE, new ServiceFetcher() {
    Display mDefaultDisplay;
    public Object getService(ContextImpl ctx) {
        Display display = ctx.mDisplay;
        if (display == null) {
        if (mDefaultDisplay == null) {
            DisplayManager dm = (DisplayManager)ctx.getOuterContext().
            getSystemService(Context.DISPLAY_SERVICE);
            mDefaultDisplay = dm.getDisplay(Display.DEFAULT_DISPLAY);
        }
        display = mDefaultDisplay;
    }
                // 构造 WindowManagerImpl 对象
return new WindowManagerImpl(display);
}});
```

在最后一行代码中，我们看到了 WindowManager 在 Java 层的具体实现，也就是 WindowManagerImpl。那么 Dialog 对象又是怎样获取到 WindowManager 对象的呢？我们从上述代码知道，WindowManager 是注册到 ContextImpl 中的，而 getSystemService 也是 Context 定义的接口，因此，需要先从 Dialog 的构造函数进行分析，因为 Context 对象就是从 Dialog 构造函数传递进来的：

```
Dialog(Context context, int theme, boolean createContextThemeWrapper) {
    // 设置、包装 Context 相关的代码
    // 1. 获取 WindowManager
    mWindowManager = (WindowManager)context.getSystemService(Context.WINDOW_ SERVICE);
    Window w = PolicyManager.makeNewWindow(mContext);
    mWindow = w;
    // 设置 Window 的回调
    w.setCallback(this);
    // 设置 Window 的 WindowManager 对象
    w.setWindowManager(mWindowManager, null, null);
    // 代码省略
}
```

可以看到最终是通过 Window 对象的 setWindowManager 函数将 Window 对象与 Window Manager 建立了联系，该函数是在 Window 类中，我们看看这个函数实现：

```
public void setWindowManager(WindowManager wm, IBinder appToken, String appName,
boolean hardwareAccelerated) {
    // 参数赋值等代码
    if (wm == null) {
        wm = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE);
    }
    // 注意这里，调用了 createLocalWindowManager 函数
```

```
    mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);
}
```

最后一句代码很重要，调用的是 WindowManagerImpl 中的 createLocalWindowManage 方法，我们继续跟踪下去：

```
public WindowManagerImpl createLocalWindowManager(Window parentWindow) {
    return new WindowManagerImpl(mDisplay, parentWindow);
}
```

这个方法很简单，只是单纯地构建了一个 WindowManagerImpl 对象。与 ContextImpl 注册的 WindowManagerImpl 不同的是，这里多了一个 parentWindow 参数，也就是说，此时构建的 WindowManagerImpl 对象是与具体的 Window 关联的，而 ContextImpl 注册的并没有此参数（只有一个 mDisplay 参数）。此时，在 Java 层上 Window 对象就已经和 WindowManager 建立了第一步的联系。

当然这只是慢慢长路的第一步，我们还是看看 WindowManagerImpl 的核心代码：

```
public final class WindowManagerImpl implements WindowManager {
    private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();
    // Window 对象
    private final Window mParentWindow;
    // 代码省略
    private WindowManagerImpl(Display display, Window parentWindow) {
        mDisplay = display;
        mParentWindow = parentWindow;
    }
    @Override
    public void addView(View view, ViewGroup.LayoutParams params) {
        mGlobal.addView(view, params, mDisplay, mParentWindow);
    }
    @Override
    public void updateViewLayout(View view, ViewGroup.LayoutParams params) {
        mGlobal.updateViewLayout(view, params);
    }
    @Override
    public void removeViewImmediate(View view) {
        mGlobal.removeView(view, true);
    }
    // 代码省略
}
```

很显然，WindowManagerImpl 也不是具体“干活”的对象，它也只是一个穿着 Impl 羊皮的“狼”罢了，添加 View、移除 View、更新 View 的布局等具体的工作都交给了 WindowManagerGlobal 这个类，在这一切就绪之后，会调用 WindowManager 的 addView 方法请求系统将该 View 显示到屏幕上，经过上述代码分析可知，实际上调用的是 WindowManagerGlobal 中的 addView 方法，继续跟踪：

```
// 将 View 添加到 WindowManager 中，也就是在手机窗口中显示该 View
public void addView(View view, ViewGroup.LayoutParams params,
            Display display, Window parentWindow) {
        // 检查参数有效性等代码省略
        ViewRootImpl root;
        View panelParentView = null;
        synchronized (mLock) {
```

```
            // 代码省略
            // 1. 构建 ViewRootImpl
            root = new ViewRootImpl(view.getContext(), display);
            // 2. 给 View 设置布局参数
            view.setLayoutParams(wparams);
            // 3. 将 View 添加到 View 列表中
            mViews.add(view);
            // 4. 将 ViewRootImpl 对象 root 添加到 mRoots 对象中
            mRoots.add(root);
            mParams.add(wparams);
        }
            // do this last because it fires off messages to start doing things
        try {
            // 5. 调用 ViewRootImpl 的 setView 方法将 View 显示到手机窗口中
            root.setView(view, wparams, panelParentView);
        } catch (RuntimeException e) {

            }
    }
```

上面程序主要分了以下 4 个步骤，具体概括如下：

（1）构建 ViewRootImpl；

（2）将布局参数设置给 View；

（3）存储这些 ViewRootImpl、View、LayoutParam 到列表中；

（4）通过 ViewRootImpl 的 setView 将 View 显示到窗口上。

很多读者可能对 ViewRootImpl 并不陌生，乍一看它的名字可能会误以为它是一个 View，但实际上不是这样的，它继承自 Handler 类，是作为 native 层与 Java 层 View 系统通信的桥梁，比如我们熟知的 performTraversals 函数就是收到系统绘制 View 的消息之后，通过调用视图树的各个节点的 meature、layout、draw 方法来绘制整棵视图树。

对 WindowManager 有些了解的读者可能注意到，经过上述的剖析，我们才到 Framework 层，而 WMS 可是运行在 Native 层的，Framework 层如何与 Native 层建立关联、以及背后的原理是什么，我们还是继续挖掘吧。

既然 ViewRootImpl 是 Frameork 层与 Native 层的通信桥梁，以及从上述的分析来看，ViewRootImpl 很有可能是与 Native 层建立联系的关键点，不过我们这里只讨论与 WMS 的关联。从 WindowManagerGlobal 的 addView 函数中可以看到，第一个重要步骤就是构建了 ViewRootImpl 对象 root，我们看看它的构造函数：

```
public ViewRootImpl(Context context, Display display) {
    mContext = context;
      // 1. 获取 Window Session，也就是与 WindowManagerService 建立连接
    mWindowSession = WindowManagerGlobal.getWindowSession();
      // 代码省略
      // 保存当前线程，更新 UI 的线程只能是创建 ViewRootImpl 时的线程
      // 我们在应用开发中，如果在子线程中更新 UI 会抛出异常，但并不是因为只有 UI 线程才能更新 UI
      // 而是因为 ViewRootImpl 是在 UI 线程中创建的
    mThread = Thread.currentThread();
  }
```

我们看到 ViewRootImpl 的代码 1 处，有一个 WindowManagerGlobal.getWindowSession()方法，通过函数命名以及 WindowManagerGlobal 的作用来看，这很可能就是与 Native 层建立通信的地方。

有代码才有真相，我们继续跟踪。

```
public static IWindowSession getWindowSession() {
    synchronized (WindowManagerGlobal.class) {
        if (sWindowSession == null) {
            try {
                InputMethodManager imm = InputMethodManager.getInstance();
                // 1. 获取 WindowManagerService
                IWindowManager windowManager = getWindowManagerService();
                // 2. 与 WindowManagerSerice 建立一个 Session
                sWindowSession = windowManager.openSession(
                    imm.getClient(), imm.getInputContext());
                } catch (RemoteException e) {
                    Log.e(TAG, "Failed to open window session", e);
                }
            }
            return sWindowSession;
        }
    }

    // 获取 WindowManagerService
    public static IWindowManager getWindowManagerService() {
        synchronized (WindowManagerGlobal.class) {
            if (sWindowManagerService == null) {
                sWindowManagerService = IWindowManager.Stub.asInterface(
                ServiceManager.getService("window"));
                }
            return sWindowManagerService;
        }
    }
```

在 getWindowSession 函数中，Framework 层首先通过 getWindowManagerService 函数获取到 IWindowManager 对象，该函数中通过 ServiceManager.getService 函数获取 WMS，并且将 WMS 转换为 IWindowManager 类型。我们先看看 ServiceManager.getService 函数代码：

```
public static IBinder getService(String name) {
    try {
        IBinder service = sCache.get(name);
        if (service != null) {
            return service;
        } else {
            return getIServiceManager().getService(name);
        }
        } catch (RemoteException e) {
            Log.e(TAG, "error in getService", e);
        }
    return null;
}
```

从程序中可以看到 ServiceManager.getService 返回的是 IBinder 对象，也就是说 Android Framework 与 WMS 之间也是通过 Binder 机制进行通信，到了这一步我们已经与 WMS 建立了初步联系。获取 WMS 之后，又调用了 IWindowManager.Stub 类的 asInterface 函数，将获取到的 WMS 的 IBinder 对象转换成 WindowManager 对象。最后，通过 openSessiton 函数来与 WMS 建立一个通信会话，相当于 Framework 层与 Native 层建立了一个长期合作的“办事处”，双方有什么需求都通过这个 Session 来交换信息。

但是，此时 Dialog 或者 Activity 的 View 并不能显示在手机屏幕上，WMS 只是负责管理手机屏幕上 View 的 z-order，也就是说 WMS 管理当前状态下哪个 View 应该在最上层显示。仔细想一下，你发现其实 WMS 管理的并不是 Window，而是 View，只不过它管理的是属于某个 Window 下的 View。

与 WMS 建立 Session 之后就到了调用 ViewRootImpl 的 setView 方法了，该方法会向 WMS 发起显示 Dialog 或者 Activity 中的 DecorView 请求，具体代码如下：

```
public void setView(View view, WindowManager.LayoutParams attrs,
        View panelParentView) {
    synchronized (this) {
        // 1. 请求布局
        requestLayout();
        try {
            // 2. 向 WMS 发起请求
            res = sWindowSession.add(mWindow, mWindowAttributes,
            getHostVisibility(), mAttachInfo.mContentInsets);
        }
}
```

setView 很复杂，但是我们主要关注两步：

（1）requestLayout；

（2）向 WMS 发起显示当前 Window 的请求。

我们先看看 requestLayout 函数：

```
public void requestLayout() {
    checkThread();
    mLayoutRequested = true;
    scheduleTraversals(); //发送 DO_TRAVERSAL 消息
}
public void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        sendEmptyMessage(DO_TRAVERSAL);
    }
}
```

就是往 handler 中发送了一个 DO_TRAVERSAL 消息，这个消息会触发整个视图树的绘制操作，也就是最终会执行 performTraversals 函数，这是一个极为复杂又非常重要的函数。简单来说它主要做了如下操作：

```
private void performTraversals() {
    // 1. 获取 Surface 对象，用于图形绘制
    // 2. 丈量整个视图树的各个 View 的大小，performMeasure 函数
    // 3. 布局整个视图树，performLayout 函数
    // 4. 绘制整棵视图树，performDraw 函数
}
```

在第四步中，Framework 会获取到图形绘制表面 Surface 对象，然后获取它的可绘制区域，也就是我们的 Canvas 对象，然后 Framework 在这个 Canvas 对象上绘制，具体代码如下：

```
private void performDraw() {
    if (!mAttachInfo.mScreenOn && !mReportNextDraw) {
        return;
```

```
        }
        final boolean fullRedrawNeeded = mFullRedrawNeeded;
        mFullRedrawNeeded = false;
        // 代码省略
        mIsDrawing = true;
        try {
            // 调用绘制函数
            draw(fullRedrawNeeded);
        } finally {
            mIsDrawing = false;
        }
            // 代码省略
    }
    private void draw(boolean fullRedrawNeeded) {
        // 1. 获取绘制表面
        Surface surface = mSurface;
        if (!surface.isValid()) {
            return;
        }
        // 代码省略
        // 2. 绘图表面需要更新
        if (!dirty.isEmpty() || mIsAnimating) {
            // 3. 使用 GPU 绘制，也就是硬件加速
            if (attachInfo.mHardwareRenderer != null && attachInfo.mHardwareRenderer.isEnabled()) {
                // 代码省略
                // 使用硬件渲染器绘制
                attachInfo.mHardwareRenderer.draw(mView, attachInfo, this,
                animating ? null : mCurrentDirty);
            } else {
                // 代码省略
                // 4. 使用 CPU 绘制图形
                if (!drawSoftware(surface, attachInfo, yoff, scalingRequired, dirty)) {
                    return;
                }
            }
        }
        if (animating) {
            mFullRedrawNeeded = true;
            scheduleTraversals();
        }
    }
```

在 draw 函数中会获取到需要绘制的区域，以及判断是否使用 GPU 进行绘制。通常情况下使用的是 CPU 绘制，也就是调用的是 drawSoftware 函数来绘制。我们看看该函数的实现：

```
// 使用 CPU 绘制图形
private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int yoff,
    boolean scalingRequired, Rect dirty) {
    // Draw with software renderer.
    Canvas canvas;
    try {
        int left = dirty.left;
        int top = dirty.top;
        int right = dirty.right;
        int bottom = dirty.bottom;
        // 1. 获取指定区域的 Canvas 对象，用于 Framework 层绘图
        canvas = mSurface.lockCanvas(dirty);
            // 代码省略
    } // 省略 catch
    try {
        // 代码省略
```

```
        try {
            // 2. 从 DecorView 开始绘制，也就是整个 Window 的根视图，这会引起整棵树的重绘
            mView.draw(canvas);
            // 代码省略
        } finally {
        }
    } finally {
        try {
            // 3. 释放 Canvas 锁，然后通知 SurfaceFlinger 更新这块区域
            surface.unlockCanvasAndPost(canvas);
        } catch (IllegalArgumentException e) {
            return false;
        }
    }
    return true;
}
```

综上所述，上述的视图树绘制代码中主要分为下面几个步骤：

（1）判断是使用 CPU 绘制还是 GPU 绘制；

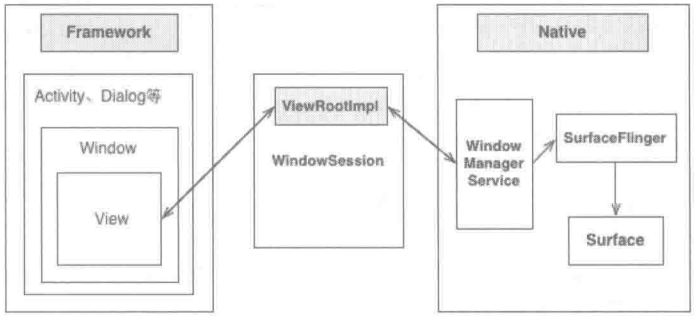
（2）获取绘制表面 Surface 对象；

（3）通过 Surface 对象获取并且锁住 Canvas 绘图对象；

（4）从 DecorView 开始发起整棵视图树的绘制流程；

（5）Surface 对象解锁 Canvas，并且通知 SurfaceFlinger 更新视图。

内容绘制完毕之后请求 WMS 显示该窗口上的内容，至此，Activity、Dialog 等组件的 View 就显示到用户的屏幕上了。整个 WMS 系统是极为复杂的，涉及的概念、技术非常广泛，各个层次的交互错综复杂，我们在此只是在一个比较高的层次上概括性地梳理了它的整个脉络，它的简化结构如图 3-4 所示。



▲图 3-4

关于 WindowManagerService 以及 Surface 系统的细节大家可以参考市面上的一些源码分析书籍，我们在此就不做过多分析了。

3.8 Builder 模式实战

经过了上面几个小节的各种源码分析，估计你已经有些疲惫了，让我们稍作休息，聊点轻松的

内容，那就是继续说小民的开发故事。随着小民的 ImageLoader 的不断演进，这个库的可扩展性、灵活性越来越高，在带来用户关注的同时小民也需要开发一些新特性来满足用户的需求，比较典型的问题就是配置 ImageLoader，如在前几章中讲到的缓存配置等，这还远远不够！用户需要更大的灵活性，更多的定制化，如设置图片在加载时 ImageView 显示的图片、加载失败后显示的图片、图片加载引擎线程数等。

这对于已经经过训练的小民来说难度不是很大了，随手就写下了下面的代码：

```
public class ImageLoader {
    // 图片缓存
    ImageCache mImageCache = new MemoryCache();
    // 图片加载中显示的图片 id
    int mLoadingImageId ;
    // 加载失败时显示的图片 id
    int mLoadingFailedImageId ;
    // 图片加载策略
    LoaderPolicy mLoaderPolicy ;
    // 线程池，线程数量为 CPU 的数量
    ExecutorService mExecutorService = Executors.newFixedThreadPool(Runtime.getRuntime()
    .availableProcessors());
    // 省略单例模式部分代码
    public void displayImage(String imageUrl, ImageView imageView) {
        Bitmap bitmap = mImageCache.get(imageUrl);
        if (bitmap != null) {
            imageView.setImageBitmap(bitmap);
            return;
        }
        // 提交图片加载请求
        submitLoadRequest(imageUrl, imageView);
    }
    public void setImageCache(ImageCache cache) {
        mImageCache = cache;
    }
    public void setLoadingImage(int resId) {
        mLoadingImageId = resId ;
    }

    public void setLoadingFailedImage(int resId) {
        mLoadingFailedImageId = resId ;
    }

    public void setLoadingPolicy(LoaderPolicy policy) {
        mLoaderPolicy = policy ;
    }

    public void setThreadCount(int count) {
        mExecutorService.shutdown();
        mExecutorService = null;
        // 设置新的线程数量
        mExecutorService = Executors.newFixedThreadPool(count);
    }

    private void submitLoadRequest(final String imageUrl, final ImageView imageView) {
        // 设置加载中的图片
        imageView.setImageResource(mLoadingImageId);

        imageView.setTag(imageUrl);
        mExecutorService.submit(new Runnable() {
```

```
            @Override
            public void run() {
                // 加载图片
                Bitmap bitmap = downloadImage(imageUrl);
                if (bitmap == null) {
                    // 设置加载图片失败后显示的图片
                    imageView.setImageResource(mLoadingFailedImageId);
                    return;
                }
                // 显示加载到的图片
            }
        });
    }

    public Bitmap downloadImage(String imageUrl) {
        Bitmap bitmap = null;
        // 省略下载图片过程
        return bitmap;
    }
}
```

小民的这段代码就是创建对应的成员变量，然后通过 setter 方法来设置这些变量值，使得这些特性都能够被用户定制，如此简单的代码，小民觉得就不需要让主管审核了，于是小民直接将代码推送到 Github 上，并且在发布日志上列出了这些更新。没过多久，小民收到了一个用户的反馈邮件，邮件主题是“ImageLoader 函数过多”，并且给小民附上了如图 3-5 所示界面。

▲图 3-5

邮件的内容如下：

Hi，小民，我在 Github 上看到你发布的 ImageLoader 觉得很好用，代码简单、灵活性好，但是，就是在 ImageLoader 里的函数较多，且用户可以在任何时候修改 ImageLoader 的配置，这就出现了一个问题，如在已经初始化了一个指定线程数量的线程池的情况下，用户再调用 setThreadCount 时应该如何处理呢？而且这样的设计也使得用户的使用成本变高。里面过多的函数暴露，也让用户在每次调用函数时都要仔细选择，你能否对这个程序做一些限制，比如用户只能在初始化时配置这些参数。谢谢！

小民看到这封信的第一感觉就是：言之有理。但是，怎么修改小民心里也没谱。于是他就在 Google 上搜索一些关于 ImageLoader 方面的资料，经过多方查找后，小民突然看到了一篇文章中的一段示例代码：

```
ImageLoaderConfiguration config = newImageLoaderConfiguration.Builder(context)
    .threadPriority(Thread.NORM_PRIORITY - 2)
```

```
    .denyCacheImageMultipleSizesInMemory()
    .discCacheFileNameGenerator(new Md5FileNameGenerator())
    .tasksProcessingOrder(QueueProcessingType.LIFO)
    .writeDebugLogs() // Remove for release app
    .build();
//初始化 ImageLoader
ImageLoader.getInstance().init(config);
```

这就是知名图片加载库 Universal-Image-Loader 的初始化配置代码。

这段程序让小民眼前一亮的是使用了 Builder 模式，小民想起了前几天刚学习的这个模式。真是“山重水复疑无路，柳暗花明又一村”。小民显然已经找到了答案，“将一个复杂对象的构建与它的表示分离”，然后用 Builder 模式来构建一个不可变的配置对象，并且将这个配置对象注入到 ImageLoader 中，也就是说它只能在构建时设置各个参数，一旦你调用 build()或者类似方法构建对象之后，它的属性就不可再修改，因为它没有 setter 方法，字段也都是隐藏的，用户只能在初始化时一次性构造这个配置对象，然后注入给 ImageLoader，ImageLoader 根据配置对象进行初始化。这样，上一个版本中的 setThreadCount、setImageCache 等方法就不需要出现在 ImageLoader 中了，用户可见的函数就会少很多，ImageLoader 的使用成本也随之降低了。

我们看看修改后的 ImageLoader，具体代码如下：

```
public final class ImageLoader {
    // 图片加载配置对象
    private ImageLoaderConfig mConfig;

    // 省略单例模式的代码

    /**初始化 ImageLoader
     * @param config
     */
    public void init(ImageLoaderConfig config) {
        mConfig = config;
        // 检测配置的合法性,内部会根据配置做一些初始化操作
        checkConfig();
        // 代码省略
    }
    // 加载图片的函数
    public void displayImage(String imageUrl, ImageView imageView) {
        Bitmap bitmap = mImageCache.get(imageUrl);
        if (bitmap != null) {
            imageView.setImageBitmap(bitmap);
            return;
        }
        // 添加加载请求
        submitLoadRequest(imageUrl, imageView);
    }
    private void submitLoadRequest(final String imageUrl, final ImageView imageView) {
        // 代码省略
    }
    public Bitmap downloadImage(String imageUrl) {
        Bitmap bitmap = null;
        // 代码省略
        return bitmap;
    }

}
```

上述代码中，把配置的代码基本上都封装到了 ImageLoaderConfig 和 Builder 对象中。一起来看看相关代码：

```
public class ImageLoaderConfig {

    // 图片缓存配置对象
    BitmapCache bitmapCache = new MemoryCache();
    // 加载图片时的 loading 和加载失败的图片配置对象
    DisplayConfig displayConfig = new DisplayConfig();
    //加载策略
    LoadPolicy loadPolicy = new SerialPolicy();
    // 线程数量,默认为 CPU 数量+ 1
    int threadCount = Runtime.getRuntime().availableProcessors() + 1;
    private ImageLoaderConfig() {
    }

    /**
     * 配置类的 Builder
     */
    public static class Buider {
        /**
         * 图片缓存配置对象
         */
        BitmapCache bitmapCache = new MemoryCache();

        /**
         * 加载图片时的 loading 和加载失败的图片配置对象
         */
        DisplayConfig displayConfig = new DisplayConfig();
        /**
         * 加载策略
         */
        LoadPolicy loadPolicy = new SerialPolicy();
        // 线程数量
        int threadCount = Runtime.getRuntime().availableProcessors() + 1;
        // 设置线程数量
        public Buider setThreadCount(int count) {
            threadCount = Math.max(1, count);
            return this;
        }
        // 设置缓存
        public Buider setCache(BitmapCache cache) {
            bitmapCache = cache;
            return this;
        }
        // 设置图片加载中显示的图片
        public Buider setLoadingPlaceholder(int resId) {
            displayConfig.loadingResId = resId;
            return this;
        }
        // 设置要加载的图片加载失败时显示的图片
        public Buider setNotFoundPlaceholder(int resId) {
            displayConfig.failedResId = resId;
            return this;
        }
        // 设置加载策略
        public Buider setLoadPolicy(LoadPolicy policy) {
            if (policy != null) {
                loadPolicy = policy;
            }
```

```
            return this;
        }
        void applyConfig(ImageLoaderConfig config) {
            config.bitmapCache = this.bitmapCache;
            config.displayConfig = this.displayConfig;
            config.loadPolicy = this.loadPolicy;
            config.threadCount = this.threadCount;
        }
        /**
         * 根据已经设置好的属性创建配置对象
         *
         * @return ImageLoaderConfig 对象
         */
        public ImageLoaderConfig create() {
            ImageLoaderConfig config = new ImageLoaderConfig();
            // 应用配置
            applyConfig(config);
            return config;
        }
    }
}
```

通过将 ImageLoaderConfig 的构造函数、字段私有化，使得外部不能访问内部属性，用户唯一能够设置属性的地方就是通过 Builder 对象了，也就是说用户只能通过 Builder 对象构造 ImageLoaderConfig 对象，这就是构建和表示相分离。

但是“使得相同的构建过程可以创建不同的表示”又是如何理解呢？在经典的 Builder 模式中还有一个 Director 和 ConcreteBuilder 角色，不同的 ConcreteBuilder 是可以创建不同的 Product 子类的，因此，也就是可以创建不同的表示。我们这里并没有使用经典实现，因此，不做过多的描述。

既然小民已经完成了使用 Builder 模式的程序重构，那我们看看用户的使用代码是怎样的，具体代码如下：

```
private void initImageLoader () {
    ImageLoaderConfig config = new ImageLoaderConfig.Buider()
        .setLoadingPlaceholder(R.drawable. loading)
        .setNotFoundPlaceholder(R.drawable. not_found)
        .setCache(new DoubleCache(this))
        .setThreadCount(4)
        .setLoadPolicy(new ReversePolicy()).create();
    // 将配置初始化到 ImageLoader 中
    ImageLoader.getInstance().init(config);
}
```

调用 init 函数之后，ImageLoader 就可以正常使用了，各种 setter 函数不会在用户调用 ImageLoader 方法时出现在视野中，它们已经被隔离到了 Builder 模式中。清晰、简单的 API 也是一个开源库必须要保证的地方，看来小民的 ImageLoader 又会“俘获”不少用户的心！

用 git 命令提交后，小民悠闲地靠在办公椅上看着外面的路灯……

3.9 总结

Builder 模式在 Android 开发中也较为常用，通常作为配置类的构建器将配置的构建和表示分离开来，同时也是将配置从目标类中隔离出来，避免过多的 setter 方法。Builder 模式比较常见的实现

形式是通过调用链实现，这样使得代码更简介、易懂，例如上文说到的 ImageLoader 就是通过 ImageLoaderConfig 进行配置，这样避免了目标类中被过多的接口“污染”。

优点

（1）良好的封装性，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。
（2）建造者独立，容易扩展。

缺点

会产生多余的 Builder 对象以及 Director 对象，消耗内存。

第 4 章　使程序运行更高效——原型模式

4.1 原型模式介绍

原型模式是一个创建型的模式。原型二字表明了该模式应该有一个样板实例，用户从这个样板对象中复制出一个内部属性一致的对象，这个过程也就是我们俗称的“克隆”。被复制的实例就是我们所称的“原型”，这个原型是可定制的。原型模式多用于创建复杂的或者构造耗时的实例，因为这种情况下，复制一个已经存在的实例可使程序运行更高效。

4.2 原型模式的定义

用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

4.3 原型模式的使用场景

（1）类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等，通过原型拷贝避免这些消耗。

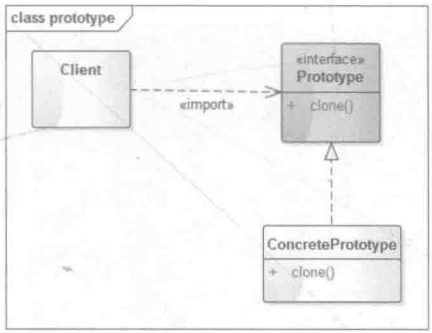
（2）通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，这时可以使用原型模式。

（3）一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用，即保护性拷贝。

需要注意的是，通过实现 Cloneable 接口的原型模式在调用 clone 函数构造实例时并不一定比通过 new 操作速度快，只有当通过 new 构造对象较为耗时或者说成本较高时，通过 clone 方法才能够获得效率上的提升。因此，在使用 Cloneable 时需要考虑构建对象的成本以及做一些效率上的测试。当然，实现原型模式也不一定非要实现 Cloneable 接口，也有其他的实现方式，本章将会对这些一一说明。

4.4 原型模式的 UML 类图

UML 类图如图 4-1 所示。



▲图 4-1

图 4-1 中的角色介绍。

- Client：客户端用户。
- Prototype：抽象类或者接口，声明具备 clone 能力。
- ConcretePrototype：具体的原型类。

4.5 原型模式的简单实现

下面以简单的文档拷贝为例来演示一下简单的原型模式，我们在这个例子中首先创建了一个文档对象，即 WordDocument，这个文档中含有文字和图片。用户经过了长时间的内容编辑后，打算对该文档做进一步的编辑，但是，这个编辑后的文档是否会被采用还不确定，因此，为了安全起见，用户需要将当前文档拷贝一份，然后再在文档副本上进行修改，这与《Effective Java》一书中提到的保护性拷贝有些类似，如此，这个原始文档就是我们上述所说的样板实例，也就是将要被“克隆”的对象，我们称为原型：

```
/**
 * 文档类型，扮演的是 ConcretePrototype 角色，而 cloneable 是代表 prototype 角色
*/
public class WordDocument implements Cloneable {
    // 文本
    private String mText;
    // 图片名列表
    private ArrayList<String> mImages = new ArrayList<String>();

    public WordDocument() {
        System.out.println("----------- WordDocument 构造函数 -----------");
    }
    @Override
    protected WordDocument clone() {
        try {
            WordDocument doc = (WordDocument) super.clone();
            doc.mText = this.mText;
            doc.mImages = this.mImages;
            return doc;
        } catch (Exception e) {
        }
        return null;
    }
```

```
    public String getText() {
        return mText;
    }
    public void setText(String mText) {
        this.mText = mText;
    }
    public List<string> getImages() {
        return mImages;
    }
    public void addImage(String img) {
        this.mImages.add(img);
    }
    /**
     * 打印文档内容
     */
    public void showDocument() {
        System.out.println("----------- Word Content Start -----------");
        System.out.println("Text : " + mText);
        System.out.println("Images List: ");
        for (String imgName : mImages) {
            System.out.println("image name : " + imgName);
        }
        System.out.println("----------- Word Content End -----------");
    }
}
```

通过 WordDocument 类模拟了 Word 文档中的基本元素，即文字和图片。WordDocument 在该原型模式示例中扮演的角色为 ConcretePrototype，而 Cloneable 的角色则为 Prototype。WordDocument 中的 clone 方法用以实现对象克隆。注意，这个方法并不是 Cloneable 接口中的，而是 Object 中的方法。Cloneable 也是一个标识接口，它表明这个类的对象是可拷贝的。如果没有实现 Cloneable 接口却调用了 clone()函数将抛出异常。在这个示例中，我们通过实现 Cloneable 接口和覆写 clone 方法实现原型模式。

下面看看 Client 端的使用：

```
public class Client {
    public static void main(String[] args) {
        // 1. 构建文档对象
        WordDocument originDoc = new WordDocument();
        // 2. 编辑文档，添加图片等
        originDoc.setText("这是一篇文档");
        originDoc.addImage("图片 1");
        originDoc.addImage("图片 2");
        originDoc.addImage("图片 3");
        originDoc.showDocument();

        // 以原始文档为原型，拷贝一份副本
        WordDocument doc2 = originDoc.clone();
        doc2.showDocument();
        // 修改文档副本，不会影响原始文档
        doc2.setText("这是修改过的 Doc2 文本");
        doc2.showDocument();

        originDoc.showDocument();
    }
}
```

输出结果如图 4-2 所示。

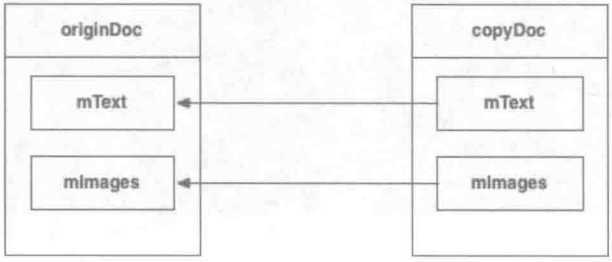
```
<terminated> Client [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Content
----------- WordDocument构造函数 -----------
----------- Word Content Start -----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
----------- Word Content End -----------
----------- Word Content Start -----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
----------- Word Content End -----------
----------- Word Content Start -----------
Text : 这是修改过的Doc2文本
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
----------- Word Content End -----------
----------- Word Content Start -----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
----------- Word Content End -----------
```

▲图 4-2

从图 4-2 中可以看到，doc2 是通过 originDoc.clone()创建的，并且 doc2 第一次输出的时候和 originDoc 输出是一样的，即 doc2 是 originDoc 的一份拷贝，它们的内容是一样的，而 doc2 修改了文本内容以后并不会影响 originDoc 的文本内容，这就保证了 originDoc 的安全性。还需要注意的是，通过 clone 拷贝对象时并不会执行构造函数！因此，如果在构造函数中需要一些特殊的初始化操作的类型，在使用 Cloneable 实现拷贝时，需要注意构造函数不会执行的问题。

4.6 浅拷贝和深拷贝

上述原型模式的实现实际上只是一个浅拷贝，也称为影子拷贝，这份拷贝实际上并不是将原始文档的所有字段都重新构造了一份，而是副本文档的字段引用原始文档的字段，如图 4-3 所示。



▲图 4-3

我们知道 A 引用 B 就是说两个对象指向同一个地址，当修改 A 时 B 也会改变，B 修改时 A 同样会改变。我们直接看下面的例子，将 main 函数的内容修改为如下：

```
public static void main(String[] args) {
    // 构建对象
    WordDocument originDoc = new WordDocument();
    // 修改文档
    originDoc.setText("这是一篇文档");
    originDoc.addImage("图片 1");
    originDoc.addImage("图片 2");
    originDoc.addImage("图片 3");
    originDoc.showDocument();
    // 拷贝文档
    WordDocument doc2 = originDoc.clone();

    doc2.showDocument();
    // 修改文字和图片
    doc2.setText("这是修改后的 Doc2 文本");
    doc2.addImage("哈哈.jpg");
    doc2.showDocument();

    originDoc.showDocument();
}
```

输出结果如图 4-4 所示。

```
<terminated> Client [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0
----------- WordDocument构造函数 -----------
----------- Word Content Start -----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
----------- Word Content End -----------
----------- Word Content Start -----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
----------- Word Content End -----------
----------- Word Content Start -----------
Text : 这是修改过的Doc2文本
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
image name : 哈哈.jpg
----------- Word Content End -----------
----------- Word Content Start -----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
image name : 哈哈.jpg
----------- Word Content End -----------
```

▲图 4-4

细心的读者可能发现了，最后两个文档信息输出是一致的。我们在 doc2 添加了一张名为“哈哈.jpg”的照片，但是，同时也显示在 originDoc 中了，这是怎么回事呢？学习过 C++的读者都会有比较深刻的体会，这是因为上文中 WordDocument 的 clone 方法中只是简单地进行浅拷贝，引用类型的新对象 doc2 的 mImages 只是单纯地指向了 this.mImages 引用，并没有重新构造一个 mImages

对象，然后将原始文档中的图片添加到新的 mImages 对象中，这样就导致 doc2 中的 mImages 与原始文档中的是同一个对象，因此，修改了其中一个文档中的图片，另一个文档也会受影响。doc2 的 mImages 添加了新的图片，实际上也就是往 originDoc 里添加了新的图片，所以，originDoc 里面也有“哈哈.jpg”图片文件。那如何解决这个问题呢？答案就是采用深拷贝，即在拷贝对象时，对于引用型的字段也要采用拷贝的形式，而不是单纯引用的形式。clone 方法修改如下：

```
/**
    * 克隆对象
    */
    @Override
    protected WordDocument clone() {
        try {
            WordDocument doc = (WordDocument) super.clone();
            doc.mText = this.mText;
            // 对 mImages 对象也调用 clone()函数，进行深拷贝
            doc.mImages = (ArrayList<String>) this.mImages.clone();
            return doc;
        } catch (Exception e) {
        }

        return null;
    }
```

如上述代码所示，将 doc.mImages 指向 this.mImages 的一份拷贝，而不是 this.mImages 本身，这样在 doc2 添加图片时并不会影响 originDoc，运行效果如图 4-5 所示。

```
<terminated> Client [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/
---------- WordDocument构造函数 ----------
---------- Word Content Start ----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
---------- Word Content End ----------
---------- Word Content Start ----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
---------- Word Content End ----------
---------- Word Content Start ----------
Text : 这是修改过的Doc2文本
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
image name : 哈哈.jpg
---------- Word Content End ----------
---------- Word Content Start ----------
Text : 这是一篇文档
Images List:
image name : 图片1
image name : 图片2
image name : 图片3
---------- Word Content End ----------
```

▲图 4-5

原型模式是非常简单的一个模式，它的核心问题就是对原始对象进行拷贝，在这个模式的使用过程中需要注意的一点就是：深、浅拷贝的问题。在开发过程中，为了减少错误，作者建议大家在使用该模式时尽量使用深拷贝，避免操作副本时影响原始对象的问题。

4.7 Android 源码中的原型模式实现

在上述的文档深拷贝示例中，mImages 对象调用了 clone()方法，这就是原型模式。我们知道，mImages 的实际类型为 ArrayList，我们看看相关源码：

```
public class ArrayList<E> extends AbstractList<E> implements Cloneable, Serializable,
RandomAccess {
    // 元素数量，也就是这个列表的大小
    int size;
    // 实际存储数据的数组
    transient Object[] array;
    /**
     * 返回一个副本对象
     */
    @Override public Object clone() {
        try {
            ArrayList<?> result = (ArrayList<?>) super.clone();
            result.array = array.clone();
            return result;
        } catch (CloneNotSupportedException e) {
                throw new AssertionError();
        }
    }
}
```

上述代码很简单，clone 方法首先克隆自身对象，然后再克隆存储元素的数组。大家可能发现了，size 并没有像 array 一样被克隆？这是为什么？原因就是 size 是整型，并不是一个对象，它是值类型，而不是引用类型，因此，不需要进行克隆。在将副本对象克隆好之后，直接返回给客户端对象即可。

有的读者可能会说 ArrayList 是 JDK 的代码，在我们比较熟悉的 Android 源码类型中有没有原型模式呢？答案是肯定的。在 Android 中，Intent 可能是我们最早接触的几个类型之一，它用于跳转 Activity、启动服务、发布广播等功能，它是 Android 系统各组件之间的纽带，也是组件之间传递数据的载体，正是 Intent 的存在才使得 Android 各个组件之间的耦合性很低，Android 的组件才如此灵活。

下面以 Intent 来分析源码中的原型模式，首先看如下示例：

```
Uri uri = Uri.parse("smsto:0800000123");
    Intent shareIntent = new Intent(Intent.ACTION_SENDTO, uri);
    shareIntent.putExtra("sms_body","The SMS text");
    // 克隆副本
    Intent intent = (Intent)shareIntent.clone() ;
    startActivity(intent);
```

上述代码中首先构造了一个发送短信的 Intent 对象，并且设置了短信内容，然后通过原始 Intent 的 clone()方法构建了一个副本 Intent，再使用这个副本 Intent 进行跳转，运行结果如图 4-6 所示。

从图 4-6 中可以看到，通过 shareIntent.clone 方法拷贝了一个对象 Intent，然后执行 startActivity(intent)，随即就进入了短信页面，号码为 0800000123，文本内容为 The SMS text，即这些内容都与 shareIntent 一致。

▲图 4-6

我们看看 Intent 的 clone()方法是如何实现的：

```
@Override
   public Object clone() {
      return new Intent(this);
   }
   /**
    * 拷贝构造函数
    */
   public Intent(Intent o) {
      this.mAction = o.mAction;
      this.mData = o.mData;
      this.mType = o.mType;
      this.mPackage = o.mPackage;
      this.mComponent = o.mComponent;
      this.mFlags = o.mFlags;
      if (o.mCategories != null) {
         this.mCategories = new ArraySet<String>(o.mCategories);
      }
      if (o.mExtras != null) {
         this.mExtras = new Bundle(o.mExtras);
      }
      if (o.mSourceBounds != null) {
         this.mSourceBounds = new Rect(o.mSourceBounds);
      }
      if (o.mSelector != null) {
         this.mSelector = new Intent(o.mSelector);
      }
      if (o.mClipData != null) {
         this.mClipData = new ClipData(o.mClipData);
      }
   }
```

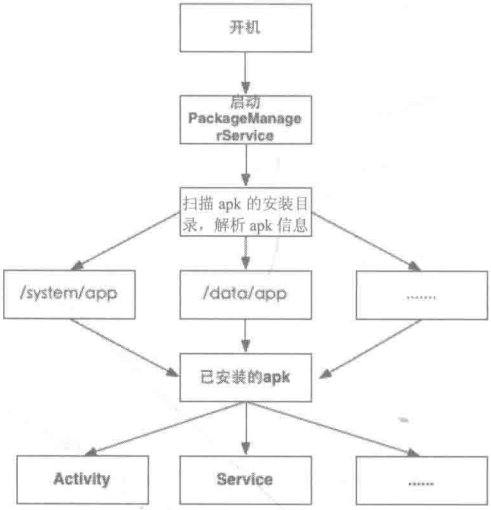
可以看到 clone 方法实际上在内部并没有调用 super.clone()方法来实现对象拷贝，而是调用了 new Intent(this)。在上文中我们提到过，使用 clone 和 new 需要根据构造对象的成本来决定，如果对象的构造成本比较高或者构造较为麻烦，那么使用 clone()函数效率较高，否则可以使用 new 的形式。这就和 C++中的拷贝构造函数完全一致，将原始对象作为构造函数的参数，然后在构造函数内将原始对象的数据逐个拷贝一遍。到此，整个克隆过程就完成了。

4.8 Intent 的查找与匹配

4.8.1 App 信息表的构建

在 Android 开发中，Intent 是极其重要的一个类，它是各个组件、进程之间通信的纽带。那么系统是如何通过 Intent 来查找到对应的组件的呢？就拿 Activity 跳转来说，在用户设置好了一个 Intent 之后，系统如何查找与匹配符合要求的 Activity？这就是我们本节需要学习的内容。

第 1 章我们说过，系统启动之后就会注册各种系统服务，如 WindowManagerService、ActivityManagerService 等，其中有一个就是 PackageManagerService（后续我们简称 PMS）。PMS 启动之后，会扫描系统中已安装的 apk 目录，例如系统 App 的安装目录为/system/app，第三方应用的目录为/data/app，PMS 会解析 apk 包下的 AndroidManifest.xml 文件得到 App 的相关信息，而每个 AndroidManifest.xml 又包含了 Activity、Service 等组件的注册信息，当 PMS 扫描并且解析完这些信息之后就构建好了整个 apk 的信息树，大致流程如图 4-7 所示。



▲图 4-7

下面我们来看看 PMS 解析已安装 apk 信息的过程。

PMS 是一个“尽职尽责”的类，对于 apk 的解析工作它在构造函数时就已经开始了！我们看看 PMS 的部分构造函数代码：

```
public PackageManagerService(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
        // 代码省略
            synchronized (mInstallLock) {
            // writer
                synchronized (mPackages) {
        // 代码省略
        // 获取/data 目录
                File dataDir = Environment.getDataDirectory();
                // /data/data 目录
                mAppDataDir = new File(dataDir, "data");
                // /data/app 目录，即第三方应用的安装目录
                mAppInstallDir = new File(dataDir, "app");
                // 代码省略

        File frameworkDir = new File(Environment.getRootDirectory(), "framework");
        // 加载 FrameWork 资源
        alreadyDexOpted.add(frameworkDir.getPath() + "/framework-res.apk");
        // 加载核心库
        alreadyDexOpted.add(frameworkDir.getPath() + "/core-libart.jar");
        // Collect ordinary system packages. 获取系统 App 的安装路径
        File systemAppDir = new File(Environment.getRootDirectory(), "app");
        mSystemInstallObserver = new AppDirObserver(
                systemAppDir.getPath(), OBSERVER_EVENTS, true, false);
                mSystemInstallObserver.startWatching();
                // 扫描系统 App 安装路径
                scanDirLI(systemAppDir, PackageParser.PARSE_IS_SYSTEM
                | PackageParser.PARSE_IS_SYSTEM_DIR, scanMode, 0);

        // 代码省略
        // 非核心应用
        if (!mOnlyCore) {
            mAppInstallObserver = new AppDirObserver(
                mAppInstallDir.getPath(), OBSERVER_EVENTS, false, false);
            mAppInstallObserver.startWatching();
            // 扫描第三方 App 安装路径
            scanDirLI(mAppInstallDir, 0, scanMode, 0);
            // 代码省略
        } else {
        // 代码省略
        }
        // 代码省略
    } // synchronized (mPackages)
    } // synchronized (mInstallLock)
}
```

从上述代码中可以看到，PMS 不仅需要加载系统已安装的各类 apk，在此之前还需要加载 Framework 资源与核心库，加载了资源与核心库之后才开始对扫描的指定目录下的 apk 文件进行解析，上述代码中只给出了系统 apk 的安装目录和第三方应用安装目录，扫描函数为 scanDirLI，我们来看看这个函数内部的实现：

```
private void scanDirLI(File dir, int flags, int scanMode, long currentTime) {
    // 1. 获取该目录下的所有文件
    String[] files = dir.list();
    // 代码省略
    int i;
    // 2. 解析目录下的所有 apk 文件
    for (i=0; i<files.length; i++) {
        File file = new File(dir, files[i]);
```

```
        // 如果不是 apk 文件，则忽略
        if (!isPackageFilename(files[i])) {
                continue;
                }
    // 3、解析 apk 文件
    scanPackageLI(file, flags|PackageParser.PARSE_MUST_BE_APK,
    scanMode, currentTime, null);
        // 代码省略
        }
    }

    private static final boolean isPackageFilename(String name) {
            return name != null && name.endsWith(".apk");
    }
```

scanDirLI 就是扫描指定目录下的所有 apk 文件，然后通过调用 scanPackageLI 函数进行解析。重点应该在 scanPackageLI 函数中，我们继续往下看：

```
private PackageParser.Package scanPackageLI(File scanFile, int parseFlags, int scanFlags,
long currentTime, UserHandle user) throws PackageManagerException {
        // 1. 创建一个包解析器
        PackageParser pp = new PackageParser();
        pp.setSeparateProcesses(mSeparateProcesses);
        pp.setOnlyCoreApps(mOnlyCore);
        pp.setDisplayMetrics(mMetrics);
        final PackageParser.Package pkg;
        try {
            // 2. 解析 apk 包
            pkg = pp.parsePackage(scanFile, parseFlags);
        } catch (PackageParserException e) {
            throw PackageManagerException.from(e);
        }
        // 代码省略

        // 3. 解析 apk 包中 Activity、Service 等组件
        PackageParser.Package scannedPkg = scanPackageLI(pkg, parseFlags, scanFlags
        | SCAN_UPDATE_SIGNATURE, currentTime, user);
        return scannedPkg;
    }
```

在 scanPackageLI 中首先构造了一个 PackageParser，也就是一个 apk 包解析器，在注释 2 处调用了 PackageParser 对象的 parsePackage 函数，具体代码如下：

```
public Package parsePackage(File packageFile, int flags) throws PackageParserException
{
    // 是否是文件夹类型
    if (packageFile.isDirectory()) {
        return parseClusterPackage(packageFile, flags);
    } else {
        // 解析单个 apk
        return parseMonolithicPackage(packageFile, flags);
    }
}
```

在 parsePackage 函数中会根据 packageFile 的类型来选择不同的解析方法，例如 packageFile 是一个文件夹，则会解析文件夹下的所有 apk；如果是一个文件，那么则调用 parseMonolithicPackage 解析单个 apk。这里我们直接看 parseMonolithicPackage 函数，具体代码如下：

```
public Package parsePackage(File sourceFile, String destCodePath,
        DisplayMetrics metrics, int flags) {
    // 代码省略
    try {
        // 1. 构建 AssetManager
        assmgr = new AssetManager();
        int cookie = assmgr.addAssetPath(mArchiveSourcePath);
        if (cookie != 0) {
            // 2. 构建资源
            res = new Resources(assmgr, metrics, null);
            // 3. 获取 AndroidManifext.xml 的解析器
            parser = assmgr.openXmlResourceParser(cookie, ANDROID_MANIFEST_FILENAME);
            assetError = false;
        } else {
                Slog.w(TAG, "Failed adding asset path:"+mArchiveSourcePath);
        }
    } catch (Exception e) {
    }
    try {
        //  4. 解析获取到的 AndroidManifext.xml
        pkg = parsePackage(res, parser, flags, errorText);
    } catch (Exception e) {
    }
    return pkg;
}
```

在 PackageParser 的 parsePackage 函数中我们看到，在注释 3 处得到了一个 AndroidManifest.xml 文件的解析器，在初学 Android 时我们就知道 Activity、Service 等组件都需要注册到 AndroidManifest.xml 中，解析这个文件就是构建与 Intent 有关的信息表。我们看到，在获取到 AndroidManifest.xml 的解析器之后调用了注释 4 处的 parsePackage(res, parser, flags, errorText)函数，这个函数的实现如下：

```
private Package parsePackage(
        Resources res, XmlResourceParser parser, int flags, String[] outError)
        throws XmlPullParserException, IOException {
        // 1. 解析到 AndroidManifext 的包名
        String pkgName = parsePackageName(parser, attrs, flags, outError);
        // 代码省略

        // 2. 构建 Package 对象
        final Package pkg = new Package(pkgName);
        boolean foundApp = false;
        // 3. 解析 AndroidManifest.xml 中的元素
        int outerDepth = parser.getDepth();
        while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
        && (type != XmlPullParser.END_TAG || parser.getDepth() > outerDepth)) {
            if (type == XmlPullParser.END_TAG || type == XmlPullParser.TEXT) {
                continue;
            }

            String tagName = parser.getName();
            if (tagName.equals("application")) {
                    // 代码省略
                // 4. 解析 Application 标签，我们的 Activity、Service 等都在这个标签内
                if (!parseApplication(pkg, res, parser, attrs, flags, outError)) {
                    return null;
                }
            } else if (tagName.equals("uses-permission")) {
                // 5. 解析用户权限标签
```

```
                if (!parseUsesPermission(pkg, res, parser, attrs, outError)) {
                    return null;
                }
            }
            // 解析其他标签，省略代码
        return pkg;
    }
```

这个 parsePackage 函数才是真正解析 AndroidManifest.xml 的函数，这里我们给出了两个标签，即 Application 和 use-permission。Application 中包含了 Activity、Service 等标签，也就是 Intent 所需的标签。我们看看解析 Application 标签的函数 parseApplication：

```
private boolean parseBaseApplication (Package owner, Resources res,
XmlPullParser parser, AttributeSet attrs, int flags, String[] outError)
throws XmlPullParserException, IOException {
        // 应用信息
        final ApplicationInfo ai = owner.applicationInfo;
        // 包名
        final String pkgName = owner.applicationInfo.packageName;
        // 1. 获取 Application 标签的 TypedArray
        TypedArray sa = res.obtainAttributes(attrs,
        com.android.internal.R.styleable.AndroidManifestApplication);
        // 获取应用名
        String name = sa.getNonConfigurationString(
        com.android.internal.R.styleable.AndroidManifestApplication_name, 0);
        // 应用图标
        ai.icon = sa.getResourceId(
        com.android.internal.R.styleable.AndroidManifestApplication_icon, 0);
        // 其他属性，代码省略
        // 获取 taskAffinity 属性
        ai.taskAffinity = buildTaskAffinityName(ai.packageName, ai.packageName,
        str, outError);

        final int innerDepth = parser.getDepth();
        int type;
        // 2. 迭代 Application 元素下的所有子元素
        while ((type = parser.next())!= XmlPullParser.END_DOCUMENT
        &&(type!= XmlPullParser.END_TAG||parser.getDepth()>innerDepth)){
            if (type == XmlPullParser.END_TAG || type == XmlPullParser.TEXT) {
                continue;
            }
            // 获取标签名
            String tagName = parser.getName();
            // 3. 解析 Activity
            if (tagName.equals("activity")) {
                Activity a = parseActivity(owner, res, parser, attrs, flags, outError, false,
                hardwareAccelerated);
                if (a == null) {
                    mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                    return false;
                }

                owner.activities.add(a);

            } else if (tagName.equals("receiver")) {
                Activity a = parseActivity(owner, res, parser, attrs, flags, outError, true, false);
                if (a == null) {
                    mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                    return false;
                }
```

```
                owner.receivers.add(a);
            }
            // 解析 Service、Provider 等其他标签的代码省略
        }

        return true;
    }
```

看到 Activity、Service、Provider、Receiver 等标签时我们心里的石头总算落地了，这个解析过程总算是有了一个里程碑式的突破，总算找到了它们，从 parseApplication 中我们看到这个过程就是普通的 xml 解析，根据不同的标签调用不同的解析方法，例如，解析 Activity 则会调用 parseActivity 函数，然后返回一个 Activity 实例，并且将这个实例添加到 Package 对象的 activities 列表中。

此时，我们需要回到上述的 PackageManagerService 类中的 scanPackageLI 的最后一步，也就是调用了 scanPackageLI(PackageParser.Package pkg,int parseFlags, int scanMode, long currentTime, UserHandle user)函数，该函数的实现如下：

```
private PackageParser.Package scanPackageLI(PackageParser.Package pkg, int parseFlags,
        int scanFlags, long currentTime, UserHandle user) throws PackageManagerException {
        boolean success = false;
        try {
            // 调用 scanPackageDirtyLI 进行扫描 apk 包
            final PackageParser.Package res = scanPackageDirtyLI(pkg, parseFlags, scanFlags,
            currentTime, user);

            success = true;
            return res;
        } finally {
            // 代码省略
        }
    }
```

这个函数调用了 scanPackageDirtyLI 函数，scanPackageDirtyLI 函数会将解析到的 Activity、Service 等添加到对应的 mActivities、mServices 等列表中。这个函数很复杂，大概有一千行，我们只看关键代码：

```
private PackageParser.Package scanPackageDirtyLI(PackageParser.Package pkg, int
parseFlags, int scanFlags, long currentTime, UserHandle user) throws
PackageManagerException {
        final File scanFile = new File(pkg.codePath);
        // 代码省略
        synchronized (mPackages) {
            // 代码省略
            N = pkg.receivers.size();
            r = null;
            for (i=0; i<N; i++) {
                PackageParser.Activity a = pkg.receivers.get(i);
                a.info.processName = fixProcessName(pkg.applicationInfo.processName,
                a.info.processName, pkg.applicationInfo.uid);
                // 添加 receiver 到 mReceivers 中
                mReceivers.addActivity(a, "receiver");
            }

            N = pkg.activities.size();
            r = null;
            for (i=0; i<N; i++) {
                PackageParser.Activity a = pkg.activities.get(i);
```

```
                a.info.processName = fixProcessName(pkg.applicationInfo.processName,
                a.info.processName, pkg.applicationInfo.uid);
                // 添加 Activity 到 mActivities 中
                mActivities.addActivity(a, "activity");
            }
        }
        return pkg;
    }
```

我们看到，这里将上一步解析到的 Activity、Service 添加到了 mActivities、mServices 中，这些类型定义是 PackageManagerService 的字段，我们看看下面的程序：

```
class PackageManagerService extends IPackageManager.Stub {
    //...
    //保存了所有 Activity 节点信息
    final ActivityIntentResolver mActivities = new ActivityIntentResolver();
    //保存了所有 BroadcastReceiver 节点信息
    final ActivityIntentResolver mReceivers = new ActivityIntentResolver();
    //保存了所有 Service 节点信息
    final ServiceIntentResolver mServices = new ServiceIntentResolver();
    //保存了所有 ContentProvider 节点信息
    final HashMap<ComponentName, PackageParser.Provider> mProvidersByComponent =
    new HashMap<ComponentName, PackageParser.Provider>();
    //...
}
```

到这一步，整个已安装 apk 的信息树就建立了，每个 apk 的应用名、包名、图标、Activity、Service 等信息都存储在系统中，当用户使用 Intent 跳转到某个 Activity 或者启动某个 Service 时，系统则会到这个信息表中进行查找，符合要求的组件则会被启动起来。这样就通过 Intent 将系统的各个组件连接在一起，使得 Android 系统成为一个组件可复用、灵活的系统。

4.8.2　精确匹配

上面分析了 apk 的信息表的构建过程，下面我们来分析一下 Intent 的查找与匹配过程。

在开发中，我们需要启动某个具体的 Activity 时，代码大致是这样的：

```
Intent intent=new Intent(mContext, MyActivity.class);
this.startActivity(intent);
```

这种情况下指定了具体的组件，也就是 MyActivity，此时在系统查找组件时会使用精确匹配，我们称为显示 Intent。

还有一种情况是不指定具体的组件，而是给出一些模糊的查询属性，例如：

```
Intent intent=new Intent(Intent. ACTION_SENDTO);
this.startActivity(intent);
```

这类我们称为隐式 Intent。

下面我们一起来看看这些 Intent 的查找与匹配过程。

startActivity 这个函数经过几个函数转发，最终会调用 startActivityForResult，我们看看这个函数：

```
public void startActivityForResult(Intent intent, int requestCode, Bundle options) {
        if (mParent == null) {
```

```
            // 1. 启动 Activity
            Instrumentation.ActivityResult ar =
                mInstrumentation.execStartActivity(
                    this, mMainThread.getApplicationThread(), mToken, this,
                    intent, requestCode, options);
            // 2. 发送启动请求
            if (ar != null) {
                mMainThread.sendActivityResult(
                    mToken, mEmbeddedID, requestCode, ar.getResultCode(),
                    ar.getResultData());
            }
        } else {
            // 代码省略
        }
    }
```

Activity 中的 startActivityForResult 函数直接调用了 Instrumentation 中的 execStartActivity 函数，具体代码如下：

```
public ActivityResult execStartActivity(
    Context who, IBinder contextThread, IBinder token, Activity target,
    Intent intent, int requestCode, Bundle options) {
        IApplicationThread whoThread = (IApplicationThread) contextThread;

        try {
            // 将 Intent 中的数据迁移到粘贴板中
            intent.migrateExtraStreamToClipData();
            // 准备离开当前进程
            intent.prepareToLeaveProcess();
            // 1. 调用 ActivityManagerService 的 startActivity 方法
            int result = ActivityManagerNative.getDefault()
                .startActivity(whoThread, who.getBasePackageName(), intent,
                        intent.resolveTypeIfNeeded(who.getContentResolver()),
                        token, target != null ? target.mEmbeddedID : null,
                        requestCode, 0, null, null, options);
            // 2. 检测结果,并且回调给调用端
            checkStartActivityResult(result, intent);
        } catch (RemoteException e) {
        }
        return null;
    }
```

execStartActivity 里面其实就是调用了 ActivityManagerService 的 startActivity 方法，这个方法里面其实就是调用了 ActivityStackSupervisor 的 startActivityMayWait 方法，该方法最后会调用 PMS 的 resolveIntent 方法，PMS 又出现在我们的视野中了！在 resolveIntent 方法中就调用了自身的 queryIntentActivities 方法，queryIntentActivities 会返回一个 ActivityInfo 对象列表，也就是符合 Intent 的 ActivityInfo 列表，而每个 ActivityInfo 对象就是一个 Activity 的档案对象，记录了一个 Acitivity 的相关信息。queryIntentActivities 的源码如下：

```
    @Override
    public List<ResolveInfo> queryIntentActivities(Intent intent,
            String resolvedType, int flags, int userId) {
        if (!sUserManager.exists(userId)) return Collections.emptyList();
            enforceCrossUserPermission(Binder.getCallingUid(), userId, false, "query
        intent activities");
        // 1. 获取 Intent 的 Component 对象
        ComponentName comp = intent.getComponent();
```

```
        // 2. 精确跳转时这个对象不为空
        if (comp != null) {
            final List<ResolveInfo>list= new ArrayList<ResolveInfo>(1);
// 3. 通过 Component 直接获取到 ActivityInfo 对象
            final ActivityInfo ai = getActivityInfo(comp,flags,userId);
            if (ai != null) {
                final ResolveInfo ri = new ResolveInfo();
                ri.activityInfo = ai;
                list.add(ri);
            }
            return list;
        }

        // 4. Component 为空则为隐式 Intent
        synchronized (mPackages) {
            final String pkgName = intent.getPackage();
            if (pkgName == null) {
                return mActivities.queryIntent(intent, resolvedType, flags, userId);
}
                // 5. 通过包名获取到 Package 对象
                final PackageParser.Package pkg = mPackages.get(pkgName);
                // 6. 通过 package 对象获取到 ActivityInfo
                if (pkg != null) {
                    return mActivities.queryIntentForPackage(intent, resolvedType, flags,
                    pkg.activities, userId);
            }
            return new ArrayList<ResolveInfo>();
        }
    }
```

上述函数运行大致是这么一个过程：如果 Intent 指明了 Componet，那么直接通过 Componet 就可以找到 ActivityInfo 列表，这个列表的数量只有一个，这个 ActivityInfo 就是指定的那个组件；如果没有指定具体的组件，那么 Component 为空，此时先查看 Intent 是否指定了要跳转到的目标组件所在的包名，如果有包名，则会通过包名获取到对应的 ActivityInfo；否则就需要通过 ActivityIntentResolver 等类的 queryIntentForPackage 进行模糊匹配，例如，根据 Action、Category 等。这里需要大家注意，mActivities 就是上个章节说到的存储了从 AndroidManifest.xml 中解析到的 Activity。既然已经找到了对应的 Activity 信息，那么最好就是启动对应的 Activity。对于显式 Intent 来说，就是直接跳转到具体的 Activity 中；对于隐式 Intent 来说，可能会弹出如图 4-8 所示的窗口让用户选择具体的应用。

▲图 4-8

到最后我们来理一理思路，在系统启动时 PackageManagerService 会启动，此时 PMS 将解析所有已安装的应用的信息，构建一个信息表，当用户通过 Intent 来跳转到某个组件时，会根据 Intent 中包含的信息到 PMS 中查找对应的组件列表，最后跳转到目标组件中。

4.9 原型模式实战

在开发中，我们有时候会满足一些需求，就是有的对象中的内容只允许客户端程序读取，而不允许修改。就拿小民最近在开发的一个简化版的新浪微博客户端来说，在用户登录之后，小民会通过一个 LoginSession 保存用户的登录信息，这些用户信息可能在 App 的其他模块被用来做登录校验、用户个人信息显示等。但是，这些信息在客户端程序是不允许修改的，而需要在其他模块被调用，因此，需要开放已登录用户信息的访问接口。我们先看看小民的实现：。

```
/**
 * 用户实体类
 */
public class User{
    public int age;
    public String name;
    public String phoneNum;
    public Address address;
    @Override
    public String toString() {
        return "User [ age=" + age + ", name=" + name + ", address="+ address + "] ";
    }

}

// 用户地址类，存储地址的详细信息
public class Address {
    // 城市
    public String city;
    // 区
    public String district;
    // 街道
    public String street;
    public Address(String aCity, String aDist, String aStreet) {
        city = aCity;
        district = aDist;
        street = aStreet;
    }
    @Override
    public String toString() {
        return "Address [ city=" + city + ", district=" + district + ", street=" + street + "] ";
    }

}

// 登录接口
public interface Login {
    void login();
}
// 登录实现
public class LoginImpl implements Login {
    @Override
    public void login() {
        // 登录到服务器,获取到用户信息
        User loginedUser = new User();
        // 将服务器返回的完整信息设置给 loginedUser 对象
        loginedUser.age = 22;
```

```
        loginedUser.name = "Mr.Simple";
        loginedUser.address = new Address("北京市", "海淀区", "花园东路");

//登录完之后将用户信息设置到 Sessin 中 LoginSession.getLoginSession()里
        setLoginedUser(loginedUser);
    }

}

// 登录 Session
public class LoginSession {
    static LoginSession sLoginSession = null;
    // 已登录用户
    private User loginedUser;
    private LoginSession() {

    }
    public static LoginSession getLoginSession() {
        if (sLoginSession == null) {
            sLoginSession = new LoginSession();
        }
        return sLoginSession;
    }
    // 设置已登录的用户信息，不对外开放
    void setLoginedUser(User user) {
        loginedUser = user;
    }

    public User getLoginedUser() {
        return loginedUser;
}
}
```

上述代码比较简单，就是在用户登录之后通过 LoginSession 的 setLoginedUser 函数将登录用户的信息设置到 Session 中，这个 setLoginedUser 是包级私有的，因此，外部模块无法调用，这在一定程度上满足了小民的需求，也就是外部客户端程序不能修改已登录的用户信息。

不巧的是，小民的开发搭档任大力也是一位经验不太丰富的工程师，因此，被主管安排与小民一块研发这个微博客户端，他在用户个人地址修改页面写出了类似这样的代码：

```
// 获取已登录的 User 对象
User newUser = LoginSession.getLoginSession().getLoginedUser();
// 更新用户
newUser.address = new Address("北京市", "朝阳区", "大望路");
```

在用户点击更新按钮时，直接调用了类似上述的代码来更新用户地址，而不是网络请求成功后才调用相关的个人信息更新函数，小民在测试中发现了这个问题，在网络请求未成功的情况下用户数据被修改，而且这个修改并不是在 LoginSession 包中，因为客户端代码只能通过 setLoginedUser 来更新用户信息，这就很奇怪了，小民找来任大力一起找程序的 Bug，他们在更新用户信息的代码下添加了这两行 Log 输出代码：

```
Log.d ("tag","temp User : " + tempUser);
Log.d("tag","已登录用户 : " + LoginSession.getLoginSession().getLoginedUser());
```

从 Log 中发现了问题：

```
temp User : User [ age=22, name=Mr.Simple, address=Address [  city=北京市, district=朝阳区, street=大望路]  ]
已登录用户 : User [ age=22, name=Mr.Simple, address=Address [  city=北京市, district=朝阳区, street=大望路]  ]
```

也就是上述说的，网络请求未成功的情况下修改了用户的 address 字段！小民感觉自己设置的用户信息更新只限于与 LoginSession 类在同一个包下的限制瞬间被突破了！这样一来，不管客户端代码是无意间写错了代码导致用户信息被修改，还是对于代码理解有误导致的问题，最终结果都是用户信息被修改了，小民一时也没有了解决方案，于是找来主管咨询。

“这类问题你们可以使用原型模式来进行保护性拷贝，也就是说在 LoginSession 的 getLoginUser()函数中返回的是已登录用户的一个拷贝，当更新用户地址的网络请求完成时，再通过包级私有的 LoginSession 中的 setLoginedUser 更新用户信息，当然，这个网络请求所在的包此时应该与 LoginSession 一致。”小民与任大力这才明白过来，于是在 User 类中覆写了 clone 方法：

```
/**
 * 用户实体类
 */
public class User implements Cloneable {
    @Override
    public User clone() {
        User user = null;
        try {
            user = (User) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }

        return user;
    }

    // 代码省略

}
```

并且在 LoginSession 中将 getLoginedUser 函数修改如下：

```
public User getLoginedUser() {
        return loginedUser;
        // return loginedUser.clone();
    }
```

这就使得在任何地方调用 getLoginedUser 函数获取到的用户对象都是一个拷贝对象，即使客户端代码一不小心修改了这个拷贝对象，也不会影响最初的已登录用户对象，对已登录用户信息的修改只能通过 setLoginedUser 这个方法，而只有与 LoginSession 在同一个包下的类才能访问这个包级私有方法，因此，确保了它的安全性。

4.10 总结

原型模式本质上就是对象拷贝，与 C++中的拷贝构造函数有些类似，它们之间容易出现的问题

也都是深拷贝、浅拷贝。使用原型模式可以解决构建复杂对象的资源消耗问题，能够在某些场景下提升创建对象的效率。还有一个重要的用途就是保护性拷贝，也就是某个对象对外可能是只读的，为了防止外部对这个只读对象修改，通常可以通过返回一个对象拷贝的形式实现只读的限制。

优点与缺点

优点

原型模式是在内存中二进制流的拷贝，要比直接 new 一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。

缺点

这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的，在实际开发当中应该注意这个潜在的问题。优点就是减少了约束，缺点也是减少了约束，需要大家在实际应用时考虑。

第 5 章 应用最广泛的模式——工厂方法模式

5.1 工厂方法模式介绍

工厂方法模式（Factory Pattern），是创建型设计模式之一。工厂方法模式是一种结构简单的模式，其在我们平时开发中应用很广泛，也许你并不知道，但是你已经使用了无数次该模式了，如 Android 中的 Activity 里的各个生命周期方法，以 onCreate 方法为例，它就可以看作是一个工厂方法，我们在其中可以构造我们的 View 并通过 setContentView 返回给 framework 处理等，相关内容我们下面再讲，先来看看工厂方法的模式定义。

5.2 工厂方法模式的定义

定义一个用于创建对象的接口，让子类决定实例化哪个类。

5.3 工厂方法模式的使用场景

在任何需要生成复杂对象的地方，都可以使用工厂方法模式。复杂对象适合使用工厂模式，用 new 就可以完成创建的对象无需使用工厂模式。

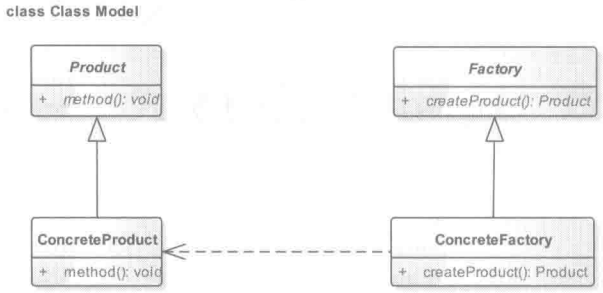
5.4 工厂方法模式的 UML 类图

UML 类图如图 5-1 所示。

根据图 5-1 我们可以得出如下一个工厂方法模式的通用模式代码。

程序清单 5-1　抽象产品类

```
public abstract class Product {
    /**
     * 产品类的抽象方法
     *由具体的产品类去实现
     */
    public abstract void method();
}
```



▲图 5-1

程序清单 5-2　具体产品类 A

```
public class ConcreteProductA extends Product {
    @Override
    public void method() {
        System.out.println("我是具体的产品 A");
    }
}
```

程序清单 5-3　具体产品类 B

```
public class ConcreteProductB extends Product {
    @Override
    public void method() {
        System.out.println("我是具体的产品 B");
    }
}
```

程序清单 5-4　抽象工厂类

```
public abstract class Factory {
    /**
     * 抽象工厂方法
     * 具体生产什么由子类去实现
     *
     * @return 具体的产品对象
     */
    public abstract Product createProduct();
}
```

程序清单 5-5　具体工厂类

```
public class ConcreteFactory extends Factory {
    @Override
    public Product createProduct() {
        return new ConcreteProductA();
    }
}
```

程序清单 5-6　客户类

```
public class Client {
    public static void main(String[] args) {
        Factory factory = new ConcreteFactory();
        Product p = factory.createProduct();
        p.method();
    }
}
```

这里的几个角色都很简单，主要分为四大模块，一是抽象工厂，其为工厂方法模式的核心；二是具体工厂，其实现了具体的业务逻辑；三是抽象产品，是工厂方法模式所创建的产品的父类；四是具体产品，为实现抽象产品的某个具体产品的对象。

上述的代码中我们在 Client 类中构造了一个工厂对象，并通过其生产了一个产品对象，这里我们得到的产品对象是 ConcreteProductA 的实例，如果想得到 ConcreteProductB 的实例，更改 ConcreteFactory 中的逻辑即可：

```
public class ConcreteFactory extends Factory {
    @Override
    public Product createProduct() {
        return new ConcreteProductB();
        // return new ConcreteProductA();
    }
}
```

这种方式比较常见，需要哪一个产品就生产哪一个，有时候也可以利用反射的方式更简洁地来生产具体产品对象，此时，需要在工厂方法的参数列表中传入一个 Class 类来决定是哪一个产品类：

```
public abstract class Factory {
    /**
     * 抽象工厂方法
     * 具体生产什么由子类去实现
     *
     * @param clz 产品对象类类型
     *
     * @return 具体的产品对象
     */
    public abstract <T extends Product> T createProduct(Class<T> clz);
}
```

对于具体的工厂类，则通过反射获取类的示例即可：

```
public class ConcreteFactory extends Factory {
    @Override
    public <T extends Product> T createProduct(Class<T> clz) {
        Product p = null;
        try {
            p = (Product) Class.forName(clz.getName()).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return (T) p;
    }
}
```

最后再来看看 Client 中的实现：

```
public class Client {
    public static void main(String[] args) {
        Factory factory = new ConcreteFactory();
        Product p = factory.createProduct(ConcreteProductB.class);
        p.method();
    }
}
```

需要哪一个类的对象就传入哪一个类的类型即可，这种方法比较简洁、动态，如果你不喜欢这

种方式，也可以尝试为每一个产品都定义一个具体的工厂，各司其职：

```
public class ConcreteFactoryA extends Factory {
    @Override
    public Product createProduct() {
        return new ConcreteProductA();
    }
}

public class ConcreteFactoryB extends Factory {
    @Override
    public Product createProduct() {
        return new ConcreteProductB();
    }
}

public class Client {
    public static void main(String[] args) {
        Factory factoryA = new ConcreteFactoryA();
        Product productA = factoryA.createProduct();
        productA.method();

        Factory factoryB = new ConcreteFactoryB();
        Product productB = factoryB.createProduct();
        productB.method();
    }
}
```

像这样拥有多个工厂的方式我们称为多工厂方法模式，同样地，回到我们最初的那个工厂方法模式，当我们的工厂只有一个的时候，我们还是为工厂提供了一个抽象类，那么，我们是否可以将其简化掉呢？如果确定你的工厂类只有一个，那么简化掉抽象类是肯定没问题的，我们只需要将对应的工厂方法改为静态方法即可：

```
public class Factory {
    public static Product createProduct() {
        return new ConcreteProductB();
        // return new ConcreteProductA();
    }
}
```

像这样的方式又称为简单工厂模式或静态工厂模式，它是工厂方法模式的一个弱化版本。

其实到这里大家应该可以发现，工厂方法模式是完全符合设计原则的，其降低了对象之间的耦合度，而且，工厂方法模式依赖于抽象的架构，其将实例化的任务交由子类去完成，有非常好的扩展性。

5.5 模式的简单实现

工厂方法模式对于大家来说是非常好理解的一个模式，即便是第一次听说，只要读者懂点 Java 知识，理解这个模式绝对不难，因此，对于模式的简单实现不再过多赘述，这里以一个生活中的小例子来说明。小民是一家汽车厂的厂长，对他来说，组装汽车没什么好神秘的，无非就是将一些进口的核心部件，比如发动机和一些国内的零部件组装起来，小民的汽车厂主要就是组装某款 SUV

车型，比如Q3、Q5、Q7，对于这类车型来说，内部结构差异并不是很大，因此，对小民来说，一条生产线足以应付这3种车型，对于该类生产线小民提供了一个抽象类定义：

```
public abstract class AudiFactory {
    /**
     * 某车型的工厂方法
     *
     * @param clz 具体的SUV型号类型
     *
     * @return 具体型号的SUV车对象
     */
    public abstract <T extends AudiCar> T createAudiCar(Class<T> clz);
}
```

那么有没有必要为每种车型都提供一条生产线呢？在这里，小民厂里所生产的3种SUV车型可能在主结构上并没有什么差异，因此，对于小民来说没有必要为每一种车型都提供一条不同的生产线，一条生产线即可：

```
public class AudiCarFactory extends AudiFactory {
    @Override
    public <T extends AudiCar> T createAudiCar(Class<T> clz) {
        AudiCar car = null;
        try {
            car = (AudiCar) Class.forName(clz.getName()).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return (T) car;
    }
}
```

对于这3种车型，除了一些车都有的基本共性外，还提供了自动巡航功能，类似于无人驾驶，这些功能小民都使用一个抽象的基类来声明：

```
public abstract class AudiCar {
    /**
     * 汽车的抽象产品类
     *
     * 定义汽车的一个行为方法 车可以启动开走
     */
    public abstract void drive();

    /**
     * 汽车的抽象产品类
     *
     * 定义汽车的一个行为方法 车可以自动巡航
     */
    public abstract void selfNavigation();
}
```

接下来就是生产每一种具体的车型了：

```
public class AudiQ3 extends AudiCar {
    @Override
    public void drive() {
        System.out.println(" Q3 启动啦! ");
    }
```

```
    @Override
    public void selfNavigation() {
        System.out.println(" Q3 开始自动巡航! ");
    }
}
public class AudiQ5 extends AudiCar {
    @Override
    public void drive() {
        System.out.println(" Q5 启动啦! ");
    }

    @Override
    public void selfNavigation() {
        System.out.println(" Q5 开始自动巡航! ");
    }
}
public class AudiQ7 extends AudiCar {
    @Override
    public void drive() {
        System.out.println(" Q7 启动啦! ");
    }

    @Override
    public void selfNavigation() {
        System.out.println(" Q7 开始自动巡航! ");
    }
}
```

最后我们将各个类组装起来形成一条完整的流水线：

```
public class Client {
    public static void main(String[] args) {
        // 构造一个制造汽车的工厂对象
        AudiFactory factory = new AudiCarFactory();

        // 生产 Q3 并启动
        AudiQ3 audiQ3 = factory.createAudiCar(AudiQ3.class);
        audiQ3.drive();
        audiQ3.selfNavigation();

        // 生产 Q5 并启动
        AudiQ5 audiQ5 = factory.createAudiCar(AudiQ5.class);
        audiQ5.drive();
        audiQ5.selfNavigation();

        // 生产 Q7 并启动
        AudiQ7 audiQ7 = factory.createAudiCar(AudiQ7.class);
        audiQ7.drive();
        audiQ7.selfNavigation();
    }
}
```

输出结果如下：

```
Q3 启动啦!
Q3 开始自动巡航!
Q5 启动啦!
Q5 开始自动巡航!
Q7 启动啦!
Q7 开始自动巡航!
```

从上述大家可以看到，工厂方法模式的结构很简单，逻辑也基本类似，所以，上面例子与我们类图中所述的结构是很类似的。

5.6 Android 源码中的工厂方法模式实现

如本文开头所述，工厂方法模式应用很广泛，大家平时开发中经常会使用到的数据结构中其实也隐藏着对工厂方法模式的应用，以 List 和 Set 为例，List 和 Set 都继承于 Collection 接口，而 Collection 接口继承于 Iterable 接口，Iterable 接口很简单，就一个 iterator 方法：

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

这意味着 List 和 Set 接口也会继承该方法，平时比较常用的两个间接实现类 ArrayList 和 HashSet 中 iterator 方法的实现就是构造并返回一个迭代器对象：

```
public class ArrayList<E> extends AbstractList<E> implements Cloneable, Serializable,
RandomAccess {
    @Override public Iterator<E> iterator() {
        return new ArrayListIterator();
    }

    private class ArrayListIterator implements Iterator<E> {
        private int remaining = size;

        private int removalIndex = -1;

        private int expectedModCount = modCount;

        public boolean hasNext() {
            return remaining != 0;
        }

        @SuppressWarnings("unchecked") public E next() {
            ArrayList<E> ourList = ArrayList.this;
            int rem = remaining;
            if (ourList.modCount != expectedModCount) {
                throw new ConcurrentModificationException();
            }
            if (rem == 0) {
                throw new NoSuchElementException();
            }
            remaining = rem - 1;
            return (E) ourList.array[removalIndex = ourList.size - rem];
        }

        public void remove() {
            Object[] a = array;
            int removalIdx = removalIndex;
            if (modCount != expectedModCount) {
                throw new ConcurrentModificationException();
            }
            if (removalIdx < 0) {
                throw new IllegalStateException();
```

```
            }
            System.arraycopy(a, removalIdx + 1, a, removalIdx, remaining);
            a[--size] = null;  // Prevent memory leak
            removalIndex = -1;
            expectedModCount = ++modCount;
        }
    }
}

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable
{
    @Override
    public Iterator<E> iterator() {
        return backingMap.keySet().iterator();
    }
}
```

HashSet 的 iterator 方法中会返回成员变量 backingMap 中对应 HashSet 对象元素的迭代器对象，最终返回的是 KeySet 中的一个迭代器对象：

```
public class HashMap<K, V> extends AbstractMap<K, V> implements Cloneable, Serializable
{
    Iterator<K> newKeyIterator() { return new KeyIterator(); }

    private final class KeySet extends AbstractSet<K> {
        public Iterator<K> iterator() {
            return newKeyIterator();
        }
    }
}
```

ArrayList 和 HashSet 中的 iterator 方法其实就相当于一个工厂方法，专为 new 对象而生，这里 iterator 方法是构造并返回一个具体的迭代器。当然，Java 源码中对工厂方法的应用范例也不少，大家可以自己体会。Android 中对工厂方法模式的应用更多，读者其实第一次接触 Android 时就已经遇到了但未察觉到，相信以下代码对一个 Android 新手来说都不陌生：

```
public class AActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(new LinearLayout(this));
    }
}
```

这里构造一个线性布局 LinearLayout 对象并设置为当前 Activity 的根布局，这段代码看似再平常不过了，实质上，onCreate 方法就相当于一个工厂方法，为什么呢？因为我们知道 LinearLayout 是一个 ViewGroup，而 ViewGroup 又继承于 View，简单地说就是，所有控件都是 View 的子类，上面的代码中，我们在 AActivity 的 onCreate 方法中构造一个 View 对象，并设置为当前界面的 ContentView 返回给 framework 处理，如果现在又有一个 BActivity，这时我们又在其 onCreate 方法中通过 setContentView 方法设置另外不同的 View，这是不是就是一个工厂模式的结构呢？其实设计模式离我们非常近！

5.7 关于 onCreate 方法

上面我们提到 onCreate 方法就相当于一个工厂方法，那么，这个方法又是从何而来的呢？大家在学习 Android 的第一天就知道，onCreate 方法是一个 Activity 的入口点，而我们在 Java 基础的学习中也曾学到，一个类在构造时构造方法总是被首先调用，但是，极少在 Activity 中涉及构造方法的使用，这是因为 Android framework 将 Activity 作为一个业务控制者已经为我们简化了很大一部分的逻辑构思，大部分情况下只需要关注其生命周期即可而不用考虑更多。通常情况下，我们的 App 应用存在 Activity 的话都会有一个主 Activity，对于开发者来说常常会认为这个 Activity 就是应用的程序入口，实质并非如此，对于一个应用程序来说，其真正的入口是在 ActivityThread 类中，ActivityThread 中含有我们熟悉的 main 方法：

```
public final class ActivityThread {
    public static void main(String[] args) {
        SamplingProfilerIntegration.start();

        CloseGuard.setEnabled(false);

        Environment.initForCurrentUser();

        EventLogger.setReporter(new EventLoggingReporter());

        Security.addProvider(new AndroidKeyStoreProvider());

        final File configDir = Environment.getUserConfigDirectory(UserHandle.myUserId());
        TrustedCertificateStore.setDefaultUserDirectory(configDir);
        Process.setArgV0("<pre-initialized>");
        Looper.prepareMainLooper();
        ActivityThread thread = new ActivityThread();
        thread.attach(false);
        if (sMainThreadHandler == null) {
            sMainThreadHandler = thread.getHandler();
        }
        if (false) {
            Looper.myLooper().setMessageLogging(new
                    LogPrinter(Log.DEBUG, "ActivityThread"));
        }
        Looper.loop();
        throw new RuntimeException("Main thread loop unexpectedly exited");
    }
}
```

注意，ActivityThread 是一个 final 类，也就是说它是不能被继承的，一个应用程序对应一个 ActivityThread，当 Zygote 进程孵化出一个新的应用进程后，会执行 ActivityThread 的 main 方法，main 方法中做了一些比较常规的逻辑，比如准备 Looper 和消息队列，然后调用 ActivityThread 的 attach 方法将其绑定到 ActivityManagerService 中，开始不断地读取消息队列中的消息并分发消息，这是一个典型的基于事件驱动的模型。这里我们主要来看看 attach 方法中都做了什么：

```
private void attach(boolean system) {
    sCurrentActivityThread = this;
    mSystemThread = system;
```

```
    if (!system) {
        // ......省略部分无关代码......

        final IActivityManager mgr = ActivityManagerNative.getDefault();
        try {
            mgr.attachApplication(mAppThread);
        } catch (RemoteException ex) {
        }
        // ......省略部分无关代码......
    } else {
        // ......省略 System 部分的代码......
    }
    // ......省略部分无关代码......
}
```

attach 方法里根据 boolean 类型的参数 system 分为两部分，我们只看非 System 的部分，这部分代码也比较简单，主要是调用 ActivityManagerService 的 attach 方法将 ActivityThread 绑定至 ActivityManagerService 中，注意，这里 ActivityManagerNative.getDefault() 所返回的是一个 ActivityManagerService 对象，关于 ActivityManagerService 的相关部分在其他章节也会有所阐述，这里不再多说，而在 attachApplication 方法中，其主要还是调用了 attachApplicationLocked 方法来进行绑定：

```
public final class ActivityManagerService extends ActivityManagerNative implements
Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
    // ......省略部分代码......
    @Override
    public final void attachApplication(IApplicationThread thread) {
        synchronized (this) {
            int callingPid = Binder.getCallingPid();
            final long origId = Binder.clearCallingIdentity();
            attachApplicationLocked(thread, callingPid);
            Binder.restoreCallingIdentity(origId);
        }
    }
    // ......省略部分代码......
}
```

attachApplicationLocked 方法逻辑相对来说要长很多，但是，其中比较重要的部分就是 bindApplication 和 attachApplicationLocked 这两个方法的调用：

```
private final boolean attachApplicationLocked(IApplicationThread thread, int pid) {
    // ......省略部分无关代码......
    try {
        // ......省略为 bindApplication 方法调用准备参数的代码......
        thread.bindApplication(processName, appInfo, providers, app.instrumentationClass,
profilerInfo, app.instrumentationArguments, app.instrumentationWatcher, app.
instrumentationUiAutomationConnection, testMode, enableOpenGlTrace,
isRestrictedBackupMode || !normalMode, app.persistent, new Configuration(mConfiguration),
app.compat, getCommonServicesLocked(app.isolated), mCoreSettingsObserver.
getCoreSettingsLocked());
        // ......省略部分无关代码......
    } catch (Exception e) {
        // ......省略部分无关代码......
        return false;
    }
    // ......省略部分无关代码......
    if (normalMode) {
```

```
        try {
            if (mStackSupervisor.attachApplicationLocked(app)) {
                didSomething = true;
            }
        } catch (Exception e) {
            badApp = true;
        }
    }
    // ......省略部分无关代码......
    return true;
}
```

bindApplication 的参数很多，其主要作用就如方法的名字一样，将 ApplicationThread 对象绑定到 ActivityManagerService，这里不再对其做过多的介绍，我们主要看看 mStackSupervisor.attachApplicationLocked(app)，这个 mStackSupervisor 指向的是一个 ActivityStackSupervisor 类的对象，该类中的 attachApplicationLocked 方法描述如下：

```
public final class ActivityStackSupervisor implements DisplayListener {
    boolean attachApplicationLocked(ProcessRecord app) throws RemoteException {
        for (int displayNdx = mActivityDisplays.size() - 1; displayNdx >= 0; --displayNdx)
{
            ArrayList<ActivityStack> stacks = mActivityDisplays.valueAt(displayNdx).mStacks;
            for (int stackNdx = stacks.size() - 1; stackNdx >= 0; --stackNdx) {
                final ActivityStack stack = stacks.get(stackNdx);
                if (!isFrontStack(stack)) {
                    continue;
                }
                ActivityRecord hr = stack.topRunningActivityLocked(null);
                if (hr != null) {
                    if (hr.app == null && app.uid == hr.info.applicationInfo.uid &&
                        processName.equals(hr.processName)) {
                        try {
                            if (realStartActivityLocked(hr, app, true, true)) {
                                didSomething = true;
                            }
                        } catch (RemoteException e) {
                            throw e;
                        }
                    }
                }
            }
        }
        return didSomething;
    }
}
```

方法代码比较长，但是真正对我们有意义的就一句：

```
if (realStartActivityLocked(hr, app, true, true)) {
    didSomething = true;
}
```

看这段条件判断式中的 realStartActivityLocked 方法，就如其名字所示，在其内部处理了“真正”启动 Activity 的逻辑：

```
final boolean realStartActivityLocked(ActivityRecord r, ProcessRecord app, boolean
andResume, boolean checkConfig) throws RemoteException {
    // 冻结尚未启动的其他 Activity
```

```
r.startFreezingScreenLocked(app, 0);

// 向 WindowManager 设置 Token，标识当前 App 要位于前台显示
mWindowManager.setAppVisibility(r.appToken, true);

// 搜集启动较慢的 App 的信息
r.startLaunchTickingLocked();

// 检查配置信息
if (checkConfig) {
    Configuration config = mWindowManager.updateOrientationFromAppTokens (mService.
    mConfiguration, r.mayFreezeScreenLocked(app) ? r.appToken : null);
    mService.updateConfigurationLocked(config, r, false, false);
}

// 设置相关参数信息
r.app = app;
app.waitingToKill = null;
r.launchCount++;
r.lastLaunchTime = SystemClock.uptimeMillis();

int idx = app.activities.indexOf(r);
if (idx < 0) {
    app.activities.add(r);
}
mService.updateLruProcessLocked(app, true, null);
mService.updateOomAdjLocked();

final ActivityStack stack = r.task.stack;
try {
    if (app.thread == null) {
        throw new RemoteException();
    }
    List<ResultInfo> results = null;
    List<ReferrerIntent> newIntents = null;
    if (andResume) {
        results = r.results;
        newIntents = r.newIntents;
    }
    // 是否桌面的 Activity ，是的话将其添加到当前 Activity 栈的底部
    if (r.isHomeActivity() && r.isNotResolverActivity()) {
        mService.mHomeProcess = r.task.mActivities.get(0).app;
    }
    mService.ensurePackageDexOpt(r.intent.getComponent().getPackageName());
    r.sleeping = false;
    r.forceNewConfig = false;
    mService.showAskCompatModeDialogLocked(r);
    r.compat = mService.compatibilityInfoForPackageLocked(r.info.applicationInfo);
    String profileFile = null;
    ParcelFileDescriptor profileFd = null;
    if (mService.mProfileApp != null && mService.mProfileApp.equals(app.processName)) {
        if (mService.mProfileProc == null || mService.mProfileProc == app) {
            mService.mProfileProc = app;
            profileFile = mService.mProfileFile;
            profileFd = mService.mProfileFd;
        }
    }
    app.hasShownUi = true;
    app.pendingUiClean = true;
    if (profileFd != null) {
        try {
            profileFd = profileFd.dup();
```

```
            } catch (IOException e) {
                if (profileFd != null) {
                    try {
                        profileFd.close();
                    } catch (IOException o) {
                    }
                    profileFd = null;
                }
            }
        }
        ProfilerInfo profilerInfo = profileFile != null ? new ProfilerInfo(profileFile,
        profileFd, mService.mSamplingInterval, mService.mAutoStopProfiler) : null;
        app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_TOP);

        // 所有参数信息到位后准备启动 Activity
        app.thread.scheduleLaunchActivity(new Intent(r.intent),r.appToken,System.identity
HashCode(r),r.info,new Configuration(mService.mConfiguration),r.compat,r.launchedFrom
Package, r.task.voiceInteractor, app.repProcState, r.icicle, r.persistentState, results,
newIntents, !andResume, mService.isNextTransitionForward(), profilerInfo);

        if ((app.info.flags&ApplicationInfo.FLAG_CANT_SAVE_STATE) != 0) {
            if (app.processName.equals(app.info.packageName)) {
                mService.mHeavyWeightProcess = app;
                Message msg = mService.mHandler.obtainMessage( ActivityManagerService.
                POST_HEAVY_NOTIFICATION_MSG);
                msg.obj = r;
                mService.mHandler.sendMessage(msg);
            }
        }
    } catch (RemoteException e) {
        if (r.launchFailed) {
            mService.appDiedLocked(app);
            stack.requestFinishActivityLocked(r.appToken, Activity.RESULT_CANCELED, null,
            "2nd-crash", false);
            return false;
        }
        app.activities.remove(r);
        throw e;
    }
    r.launchFailed = false;

    if (andResume) {
        stack.minimalResumeActivityLocked(r);
    } else {
        r.state = ActivityState.STOPPED;
        r.stopped = true;
    }
    if (isFrontStack(stack)) {
        mService.startSetupActivityLocked();
    }
    mService.mServices.updateServiceConnectionActivitiesLocked(r.app);
    return true;
}
```

realStartActivityLocked 方法中首先会准备启动 Activity 的参数信息，准备完毕后调用 ApplicationThread 的 scheduleLaunchActivity 方法准备启动 Activty：

```
public final class ActivityThread {
    private class ApplicationThread extends ApplicationThreadNative {
        public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
```

```
ActivityInfo info, Configuration curConfig, CompatibilityInfo compatInfo, String referrer,
IVoiceInteractor voiceInteractor, int procState, Bundle state, PersistableBundle persis
tentState, List<ResultInfo> pendingResults, List<ReferrerIntent> pendingNewIntents, boolean
notResumed, boolean isForward, ProfilerInfo profilerInfo) {
            updateProcessState(procState, false);
            ActivityClientRecord r = new ActivityClientRecord();

            r.token = token;
            r.ident = ident;
            r.intent = intent;
            r.referrer = referrer;
            r.voiceInteractor = voiceInteractor;
            r.activityInfo = info;
            r.compatInfo = compatInfo;
            r.state = state;
            r.persistentState = persistentState;
            r.pendingResults = pendingResults;
            r.pendingIntents = pendingNewIntents;
            r.startsNotResumed = notResumed;
            r.isForward = isForward;
            r.profilerInfo = profilerInfo;

            updatePendingConfiguration(curConfig);
            sendMessage(H.LAUNCH_ACTIVITY, r);
        }
    }
}
```

scheduleLaunchActivity 逻辑很简单，对启动的信息进行准备，这里构造了一个 ActivityClientRecord 对象，并将相关参数设置，最后通过 sendMessage 方法发送一个启动消息到消息队列，由 ActivityThread 的 Handler 处理启动，ActivityThread 内部有个继承于 Handler 的子类 H，在其内部负责处理相关的消息。上述我们调用 sendMessage 将消息发送到 H 中，H 根据对应的消息值处理消息，比如这里我们发送的标识是 LAUNCH_ACTIVITY，对应到 H 中则会调用 handleLaunchActivity 方法：

```
private class H extends Handler {
    public static final int LAUNCH_ACTIVITY = 100;
    // ......省略大量常量代码......
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case LAUNCH_ACTIVITY: {
                final ActivityClientRecord r = (ActivityClientRecord) msg.obj;

                r.packageInfo=getPackageInfoNoCheck(r.activityInfo.applicationInfo,r.compatInfo);
                handleLaunchActivity(r, null);
            } break;
            // ......省略分支语句代码......
        }
    }
    // ......省略部分无关代码......
}
```

handleLaunchActivity 方法的逻辑比较简单，主要就是调用了 performLaunchActivity 方法：

```
private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    // ......省略部分无关代码......
    Activity a = performLaunchActivity(r, customIntent);
    // ......省略部分无关代码......
```

```
}
```

performLaunchActivity 方法才是我们的重点，其中处理了具体的 Activity 启动逻辑。

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    // 获取 ActivityInfo
    ActivityInfo aInfo = r.activityInfo;

    // 获取 PackageInfo
    if (r.packageInfo == null) {
        r.packageInfo = getPackageInfo(aInfo.applicationInfo,r.compatInfo,Context.
        CONTEXT_INCLUDE_CODE);
    }
    // 获取 ComponentName
    ComponentName component = r.intent.getComponent();
    if (component == null) {
        component = r.intent.resolveActivity(mInitialApplication.getPackageManager());
        r.intent.setComponent(component);
    }
    if (r.activityInfo.targetActivity != null) {
        component = new ComponentName(r.activityInfo.packageName,r.activityInfo.targetActivity);
    }
    // 构造 Activity 对象并设置参数
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
        activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);
    } catch (Exception e) {
    }
    try {
        // 获取 Application 对象
        Application app = r.packageInfo.makeApplication(false, mInstrumentation);
        if (activity != null) {
            // 将 Application 对象、Context 对象绑定到 Activity 对象
            activity.attach(appContext, this, getInstrumentation(), r.token, r.ident, app,
            r.intent, r.activityInfo, title, r.parent, r.embeddedID,
            r.lastNonConfigurationInstances, config, r.referrer, r.voiceInteractor);
            // 调用 Activity 的 onCreate 方法
            if (r.isPersistable()) {
                mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);
            } else {
                mInstrumentation.callActivityOnCreate(activity, r.state);
            }
        }
    } catch (SuperNotCalledException e) {
        throw e;
    } catch (Exception e) {
    }
    return activity;
}
```

Instrumentationcall 中 callActivityOnCreate 方法的调用就会执行对 Activity onCreate 方法的调用：

```
public class Instrumentation {
    public void callActivityOnCreate(Activity activity, Bundle icicle) {
        prePerformCreate(activity);
        activity.performCreate(icicle);
        postPerformCreate(activity);
    }
}
```

最终就会调用到 Activity 中的 onCreate 方法：

```
public class Activity extends ContextThemeWrapper implements
        LayoutInflater.Factory2, Window.Callback, KeyEvent.Callback,
        OnCreateContextMenuListener, ComponentCallbacks2,
        Window.OnWindowDismissedCallback {
            final void performCreate(Bundle icicle) {
                onCreate(icicle);
                mActivityTransitionState.readState(icicle);
                performCreateCommon();
    }
}
```

其实不只是 onCreate 方法，像其他的生命周期方法都可以通过类似的方式去找到其调用逻辑，这里不再赘述。

5.8 工厂方法模式实战

工厂方法模式比较简单，可以在很多编程实战的地方应用到它，以作者以前项目中的一个数据存储为例，大家知道 Android 数据持久化有很多种方式，可以使用 Android 为我们提供的 SharedPreferences 和 SQLite，也可以使用常规的文件存储，但是，对数据操作的方法无非就是增、删、改、查，我们可以将每一种数据操作的方式作为一个产品类，在抽象产品类中定义操作的方法：

```
public abstract class IOHandler {
    /**
     * 添加一条个人信息
     *
     * @param id   身份证号码
     * @param name 名字
     */
    public abstract void add(String id, String name);

    /**
     * 删除一条个人信息
     *
     * @param id 身份证号码
     */
    public abstract void remove(String id);

    /**
     * 更新一条个人信息
     *
     * @param id   身份证号码
     * @param name 名字
     */
    public abstract void update(String id, String name);

    /**
     * 查询身份证对应的人名
     *
     * @param id 身份证
     * @return 人名
     */
    public abstract String query(String id);
}
```

对于每一种持久化方式我们都定义一个具体的 IO 处理类，这里就定义 3 种，分别是普通文件存储、xml 文件存储和 SQLite 数据库存储。

```
public class FileHandler extends IOHandler {
    @Override
    public void add(String id, String name) {
        // 业务处理
    }

    @Override
    public void remove(String id) {
        // 业务处理
    }

    @Override
    public void update(String id, String name) {
        // 业务处理
    }

    @Override
    public String query(String id) {
        // 业务处理
        return "AigeStudio";
    }
}

public class XMLHandler extends IOHandler {
    @Override
    public void add(String id, String name) {
        // 业务处理
    }

    @Override
    public void remove(String id) {
        // 业务处理
    }

    @Override
    public void update(String id, String name) {
        // 业务处理
    }

    @Override
    public String query(String id) {
        return "SMBrother";
    }
}

public class DBHandler extends IOHandler {
    @Override
    public void add(String id, String name) {
        // 业务处理
    }

    @Override
    public void remove(String id) {
        // 业务处理
    }

    @Override
```

```
    public void update(String id, String name) {
        // 业务处理
    }

    @Override
    public String query(String id) {
        return "Android";
    }
}
```

这里基于篇幅限制，具体的存储操作就不再多说了，只对 query 设置了不同的一个返回结果以作区分，接下来就是对工厂类的定义，这里的逻辑都比较简单，故以简单工厂的方式实现：

```
public class IOFactory {
    /**
     * 获取 IO 处理者
     *
     * @param clz IOHandler 类型的类类型
     * @return IOHandler 对象
     */
    public static <T extends IOHandler> T getIOHandler(Class<T> clz) {
        IOHandler handler = null;
        try {
            handler = (IOHandler) Class.forName(clz.getName()).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return (T) handler;
    }
}
```

最后，我们在 Activity 中试着将从不同数据文件中查询得到的数据展示在一个 TextView 上：

```
public class FactoryActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_factory);

        // 获取显示查询内容的 TextView 对象
        final TextView tvContent = (TextView) findViewById(R.id.factory_content_tv);

        // 获取查询普通文件数据的按钮对象并设置监听
        Button btnFile = (Button) findViewById(R.id.factory_file_btn);
        btnFile.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                IOHandler handler = IOFactory.getIOHandler(FileHandler.class);
                Log.d("AigeStudio", handler.query("4455645646"));
                tvContent.setText(handler.query("4455645646"));
            }
        });

        // 获取查询 xml 文件数据的按钮对象并设置监听
        Button btnXml = (Button) findViewById(R.id.factory_xml_btn);
        btnXml.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                IOHandler handler = IOFactory.getIOHandler(XMLHandler.class);
                tvContent.setText(handler.query("4455645646"));
```

```
            }
        });

        // 获取查询数据库数据的按钮对象并设置监听
        Button btnDB = (Button) findViewById(R.id.factory_db_btn);
        btnDB.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                IOHandler handler = IOFactory.getIOHandler(DBHandler.class);
                tvContent.setText(handler.query("4455645646"));
            }
        });
    }
}
```

此程序运行界面很简单，就 3 个按钮和一个文本控件，不再给出 xml 布局文件了，大家可以自行尝试。当然，工厂方法模式的应用绝不止这么简单，如我们还可以将其应用在控件或者布局上等。

5.9 总结

总的来说，工厂方法模式是一个很好的设计模式，在本文多处地方都有对其优点进行总结，但是缺点也是难以避免的，每次我们为工厂方法模式添加新的产品时就要编写一个新的产品类，同时还要引入抽象层，这必然会导致类结构的复杂化，所以，在某些情况比较简单时，是否要使用工厂模式，需要设计者权衡利弊了。

第 6 章　创建型设计模式——抽象工厂模式

6.1 抽象工厂模式介绍

抽象工厂模式（Abstract Factory Pattern），也是创建型设计模式之一。第 5 章中我们已经了解了工厂方法模式，那么这个抽象工厂又是怎么一回事呢？大家联想一下现实生活中的工厂肯定都是具体的，也就是说每个工厂都会生产某一种具体的产品，那么抽象工厂意味着生产出来的产品是不确定的，那这岂不是很奇怪？抽象工厂模式起源于以前对不同操作系统的图形化解决方案，如不同操作系统中的按钮和文本框控件其实现不同，展示效果也不一样，对于每一个操作系统，其本身就构成一个产品类，而按钮与文本框控件也构成一个产品类，两种产品类两种变化，各自有自己的特性，如 Android 中的 Button 和 TextView，iOS 中的 Button 和 TextView，Window Phone 中的 Button 和 TextView 等。

6.2 抽象工厂模式的定义

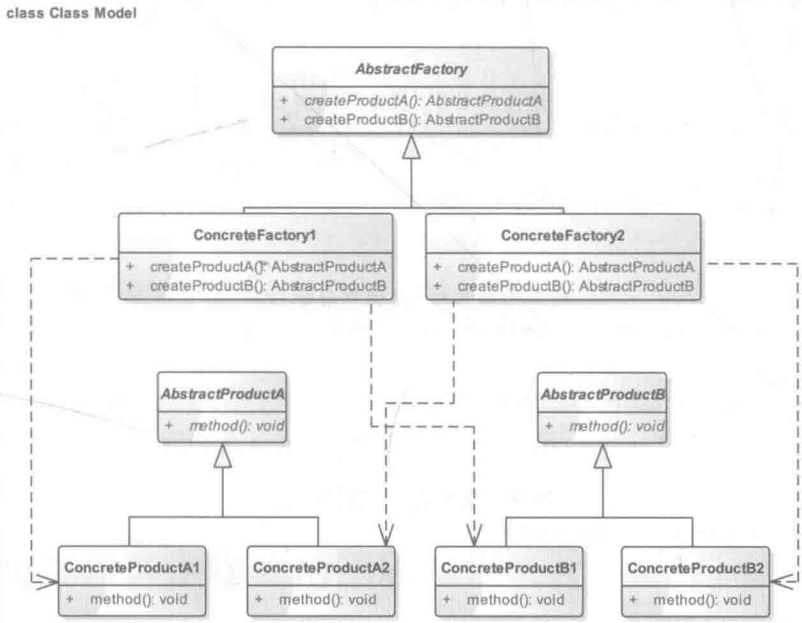
为创建一组相关或者是相互依赖的对象提供一个接口，而不需要指定它们的具体类。

6.3 抽象工厂模式的使用场景

一个对象族有相同的约束时可以使用抽象工厂模式。是不是听起来很抽象？举个例子，Android、iOS、Window Phone 下都有短信软件和拨号软件，两者都属于 Software 软件的范畴，但是，它们所在的操作系统平台不一样，即便是同一家公司出品的软件，其代码的实现逻辑也是不同的，这时候就可以考虑使用抽象工厂方法模式来产生 Android、iOS、Window Phone 下的短信软件和拨号软件。

6.4 抽象工厂模式的 UML 类图

UML 类图如图 6-1 所示。



▲图 6-1

根据类图可以得出如下一个抽象工厂模式的通用模式代码。

程序清单 6-1　抽象产品类 A

```
public abstract class AbstractProductA {
    /**
     * 每个具体的产品子类需要实现的方法
     */
    public abstract void method();
}
```

程序清单 6-2　抽象产品类 B

```
public abstract class AbstractProductB {
    /**
     * 每个具体的产品子类需要实现的方法
     */
    public abstract void method();
}
```

程序清单 6-3　具体产品类 A1

```
public class ConcreteProductA1 extends AbstractProductA {
    @Override
    public void method() {
        System.out.println("具体产品 A1 的方法");
    }
}
```

程序清单 6-4　具体产品类 A2

```
public class ConcreteProductA2 extends AbstractProductA {
    @Override
    public void method() {
```

```
        System.out.println("具体产品 A2 的方法");
    }
}
```

程序清单 6-5　具体产品类 B1

```
public class ConcreteProductB1 extends AbstractProductB {
    @Override
    public void method() {
        System.out.println("具体产品 B1 的方法");
    }
}
```

程序清单 6-6　具体产品类 B2

```
public class ConcreteProductB2 extends AbstractProductB {
    @Override
    public void method() {
        System.out.println("具体产品 B2 的方法");
    }
}
```

程序清单 6-7　抽象工厂类

```
public abstract class AbstractFactory {
    /**
     * 创建产品 A 的方法
     *
     * @return 产品 A 对象
     */
    public abstract AbstractProductA createProductA();

    /**
     * 创建产品 B 的方法
     *
     * @return 产品 B 对象
     */
    public abstract AbstractProductB createProductB();
}
```

程序清单 6-8　具体工厂类 1

```
public class ConcreteFactory1 extends AbstractFactory {
    @Override
    public AbstractProductA createProductA() {
        return new ConcreteProductA1();
    }

    @Override
    public AbstractProductB createProductB() {
        return new ConcreteProductB1();
    }
}
```

程序清单 6-9　具体工厂类 2

```
public class ConcreteFactory2 extends AbstractFactory {
    @Override
    public AbstractProductA createProductA() {
        return new ConcreteProductA2();
    }

    @Override
    public AbstractProductB createProductB() {
        return new ConcreteProductB2();
```

```
    }
}
```

虽然抽象工厂方法模式的类繁多，但是，主要还是分为 4 类。

AbstractFactory：抽象工厂角色，它声明了一组用于创建一种产品的方法，每一个方法对应一种产品，如上述类图中的 AbstractFactory 中就定义了两个方法，分别创建产品 A 和产品 B。

ConcreteFactory：具体工厂角色，它实现了在抽象工厂中定义的创建产品的方法，生成一组具体产品，这些产品构成了一个产品种类，每一个产品都位于某个产品等级结构中，如上述类图中的 ConcreteFactory1 和 ConcreteFactory2。

AbstractProduct：抽象产品角色，它为每种产品声明接口，比如上述类图中的 AbstractProductA 和 AbstractProductB。

ConcreteProduct：具体产品角色，它定义具体工厂生产的具体产品对象，实现抽象产品接口中声明的业务方法，如上述类图中的 ConcreteProductA1、ConcreteProductA2、ConcreteProductB1 和 ConcreteProductB2。

6.5 抽象工厂方法模式的简单实现

在第 5 章讲过的工厂方法模式中，以小民的车厂为例阐述了工厂方法模式，但是，后来小民发现一个问题，虽然 Q3、Q5、Q7 都是一个车系，但是三者之间的零部件差别却是很大，就拿 Q3 和 Q7 来说，Q3 使用的发动机是国产的，而 Q7 则是原装进口的；Q3 轮胎是普通的轮胎，而 Q7 则使用的是全尺寸越野轮胎；还有 Q3 使用的是比较普通的制动系统，而 Q7 则使用的是制动性能极好的制动系统。Q3、Q7 对应的是一系列车，而发动机、轮胎、制动系统则对应的是一系列零部件，两者是两种不同的产品类型，这时候就可以将抽象工厂模式应用到其中，首先，汽车工厂需要生产轮胎、发动机、制动系统这 3 种部件。

程序清单 6-10　抽象车厂类

```
public abstract class CarFactory {
    /**
     * 生产轮胎
     *
     * @return 轮胎
     */
    public abstract ITire createTire();

    /**
     * 生产发动机
     *
     * @return 发动机
     */
    public abstract IEngine createEngine();

    /**
     * 生产制动系统
     *
     * @return 制动系统
     */
    public abstract IBrake createBrake();
}
```

这里我们为每一种零部件产品定义一个接口，并分别创建两个不同的实现类表示不同的零部件产品。

程序清单 6-11　轮胎相关类

```
public interface ITire {
    /**
     * 轮胎
     */
    void tire();
}

public class NormalTire implements ITire {
    @Override
    public void tire() {
        System.out.println("普通轮胎");
    }
}

public class SUVTire implements ITire {
    @Override
    public void tire() {
        System.out.println("越野轮胎");
    }
}
```

程序清单 6-12　发动机相关类

```
public interface IEngine {
    /**
     * 发动机
     */
    void engine();
}

public class DomesticEngine implements IEngine {
    @Override
    public void engine() {
        System.out.println("国产发动机");
    }
}

public class ImportEngine implements IEngine {
    @Override
    public void engine() {
        System.out.println("进口发动机");
    }
}
```

程序清单 6-13　制动系统相关类

```
public interface IBrake {
    /**
     * 制动系统
     */
    void brake();
}

public class NormalBrake implements IBrake {
    @Override
    public void brake() {
        System.out.println("普通制动");
```

```
    }
}

public class SeniorBrake implements IBrake {
    @Override
    public void brake() {
        System.out.println("高级制动");
    }
}
```

对于生产 Q3 的工厂，其使用的零部件不同，而生产 Q7 的工厂呢，其零部件也不同。

程序清单 6-14　Q3 工厂类

```
public class Q3Factory extends CarFactory {
    @Override
    public ITire createTire() {
        return new NormalTire();
    }

    @Override
    public IEngine createEngine() {
        return new DomesticEngine();
    }

    @Override
    public IBrake createBrake() {
        return new NormalBrake();
    }
}
```

程序清单 6-15　Q7 工厂类

```
public class Q7Factory extends CarFactory {
    @Override
    public ITire createTire() {
        return new SUVTire();
    }

    @Override
    public IEngine createEngine() {
        return new ImportEngine();
    }

    @Override
    public IBrake createBrake() {
        return new SeniorBrake();
    }
}
```

最后，我们在一个 Client 客户类中模拟。

程序清单 6-16　客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造一个生产 Q3 的工厂
        CarFactory factoryQ3 = new Q3Factory();
        factoryQ3.createTire().tire();
        factoryQ3.createEngine().engine();
        factoryQ3.createBrake().brake();

        System.out.println("------------------------------");
```

```
        // 构造一个生产 Q7 的工厂
        CarFactory factoryQ7 = new Q7Factory();
        factoryQ7.createTire().tire();
        factoryQ7.createEngine().engine();
        factoryQ7.createBrake().brake();
    }
}
```

输出结果如下：

```
普通轮胎
国产发动机
普通制动
------------------------------
越野轮胎
进口发动机
高级制动
```

上面我们只是模拟了两个车系 Q3 和 Q7 的工厂，如果此时我们需要增加 Q5 的工厂呢？那么对应的轮胎、制动系统和发动机类又要增加，这里就可以看出抽象工厂方法模式的一个弊端，就是类的陡增，如果工厂类过多，势必导致类文件非常多，因此，在实际开发中一定要权衡慎用。

6.6 Android 源码中的抽象工厂方法模式实现

抽象工厂方法模式在 Android 源码中的实现相对来说是比较少的，上一节中我们曾说过 Activity 的 onCreate 方法其实就相当于一个工厂方法：

```
public class BaseActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(new LinearLayout(this));
    }
}
```

同样地对于另外一个组件 Service 来说，其 onBind 方法也可以看作是一个工厂方法：

```
public class BaseService extends Service {
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return new Binder();
    }
}
```

如果从 framework 的角度来看 Activity 和 Service 都可以看作是一个具体的工厂，这样来看相当于一个抽象工厂方法模式的雏形也没错。另一个更确切的例子是 Android 底层对 MediaPlayer 的创建：

```
sp<MediaPlayerBase>  MediaPlayerFactory::createPlayer(player_type  playerType,  void*
cookie, notify_callback_f notifyFunc) {
    sp<MediaPlayerBase> p;
    IFactory* factory;
    status_t init_result;
    Mutex::Autolock lock_(&sLock);
```

```
    if (sFactoryMap.indexOfKey(playerType) < 0) {
        // ......省去一行代码......
        return p;
    }
    factory = sFactoryMap.valueFor(playerType);
    CHECK(NULL != factory);
    p = factory->createPlayer();

    if (p == NULL) {
        // ......省去一行代码......
        return p;
    }
    init_result = p->initCheck();
    if (init_result == NO_ERROR) {
        p->setNotifyCallback(cookie, notifyFunc);
    } else {
        // ......省去一行代码......
        p.clear();
    }
    return p;
}
```

MediaPlayerFactory 是 Android 底层为创建不同 MediaPlayer 所定义的一个类，其中就调用上述的 createPlayer 函数来创建不同的 MediaPlayer，而每一种具体的 MediaPlayer 创建完毕后最终都会调用 MediaPlayerFactory 的 registerFactory_l 函数将其注册到 MediaPlayerFactory 中：

```
status_t MediaPlayerFactory::registerFactory_l(IFactory* factory, player_type type) {
    if (NULL == factory) {
        // ......省去一行代码......
        return BAD_VALUE;
    }
    if (sFactoryMap.indexOfKey(type) >= 0) {
        // ......省去一行代码......
        return ALREADY_EXISTS;
    }
    if (sFactoryMap.add(type, factory) < 0) {
        // ......省去一行代码......
        return UNKNOWN_ERROR;
    }
    return OK;
}
```

这里要注意的是，MediaPlayerFactory 其本质只是用来管理 Android 内置的 4 种不同的 MediaPlayer，而对于每一种具体的 MediaPlayer 则由一个具体的 Factory 类来创建，比如下面是 StagefrightPlayerFactory 这个抽象工厂类创建 StagefrightPlayer 的代码逻辑：

```
class StagefrightPlayerFactory : public MediaPlayerFactory::IFactory {
    public:virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/, int fd, int64_t
    offset, int64_t length, float /*curScore*/) {
        if (legacyDrm()) {
            sp<DataSource> source = new FileSource(dup(fd), offset, length);
            String8 mimeType;
            float confidence;
            if (SniffWVM(source, &mimeType, &confidence, NULL /* format */)) {
                return 1.0;
            }
        }
```

```
        if (getDefaultPlayerType() == STAGEFRIGHT_PLAYER) {
            char buf[20];
            lseek(fd, offset, SEEK_SET);
            read(fd, buf, sizeof(buf));
            lseek(fd, offset, SEEK_SET);
            uint32_t ident = *((uint32_t*)buf);
            if (ident == 0x5367674f) // 'OggS'
                return 1.0;
        }
        return 0.0;
    }
    virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/, const char* url, float
/*curScore*/) {
        if (legacyDrm() && !strncasecmp("widevine://", url, 11)) {
            return 1.0;
        }
        return 0.0;
    }
    virtual sp<MediaPlayerBase> createPlayer() {
        return new StagefrightPlayer();
    }
    private:bool legacyDrm() {
        char value[PROPERTY_VALUE_MAX];
        if (property_get("persist.sys.media.legacy-drm", value, NULL) && (!strcmp("1",
        value) || !strcasecmp("true", value))) {
            return true;
        }
        return false;
    }
};
```

NuPlayerFactory 这个抽象工厂类创建 NuPlayerDriver 的代码逻辑如下。

```
class NuPlayerFactory : public MediaPlayerFactory::IFactory {
    public:virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/, const char* url,
    float curScore) {
        static const float kOurScore = 0.8;
        if (kOurScore <= curScore)
            return 0.0;
        if (!strncasecmp("http://", url, 7) || !strncasecmp("https://", url, 8)|| !strncasecmp
        ("file://", url, 7)) {
            size_t len = strlen(url);
            if (len >= 5 && !strcasecmp(".m3u8", &url[len - 5])) {
                return kOurScore;
            }
            if (strstr(url,"m3u8")) {
                return kOurScore;
            }
            if ((len >= 4 && !strcasecmp(".sdp", &url[len - 4])) || strstr(url, ".sdp?"))
{
                return kOurScore;
            }
        }
        if (!strncasecmp("rtsp://", url, 7)) {
            return kOurScore;
        }
        return 0.0;
    }
    virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/, const
    sp<IStreamSource>& /*source*/, float /*curScore*/) {
        return 1.0;
```

```
    }
    virtual sp<MediaPlayerBase> createPlayer() {
        return new NuPlayerDriver;
    }
};
```

SonivoxPlayerFactory 这个抽象工厂类创建 MidiFile 的代码逻辑如下。

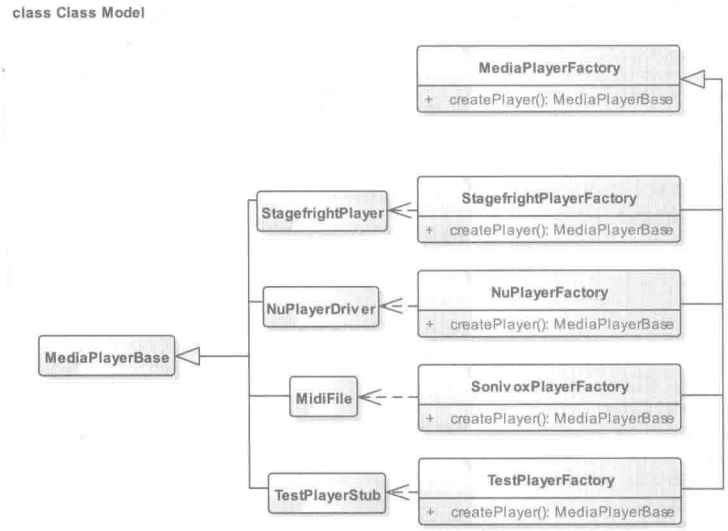
```
class SonivoxPlayerFactory : public MediaPlayerFactory::IFactory {
    public:virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/, const char* url,
    float curScore) {
        static const float kOurScore = 0.4;
        static const char* const FILE_EXTS[] = { ".mid", ".midi", ".smf", ".xmf", ".mxmf",
        ".imy", ".rtttl", ".rtx", ".ota" };
        if (kOurScore <= curScore)
            return 0.0;
        int lenURL = strlen(url);
        for (int i = 0; i < NELEM(FILE_EXTS); ++i) {
            int len = strlen(FILE_EXTS[i]);
            int start = lenURL - len;
            if (start > 0) {
                if (!strncasecmp(url + start, FILE_EXTS[i], len)) {
                    return kOurScore;
                }
            }
        }
        return 0.0;
    }
    virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/, int fd, int64_t offset,
    int64_t length, float curScore) {
        static const float kOurScore = 0.8;
        if (kOurScore <= curScore)
            return 0.0;
        EAS_DATA_HANDLE easdata;
        if (EAS_Init(&easdata) == EAS_SUCCESS) {
            EAS_FILE locator;
            locator.path = NULL;
            locator.fd = fd;
            locator.offset = offset;
            locator.length = length;
            EAS_HANDLE  eashandle;
            if (EAS_OpenFile(easdata, &locator, &eashandle) == EAS_SUCCESS) {
                EAS_CloseFile(easdata, eashandle);
                EAS_Shutdown(easdata);
                return kOurScore;
            }
            EAS_Shutdown(easdata);
        }
        return 0.0;
    }
    virtual sp<MediaPlayerBase> createPlayer() {
        return new MidiFile();
    }
};
```

TestPlayerFactory 这个抽象工厂类创建 TestPlayerStub 的代码逻辑如下。

```
class TestPlayerFactory : public MediaPlayerFactory::IFactory {
    public:virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/, const char* url,
    float /*curScore*/) {
        if (TestPlayerStub::canBeUsed(url)) {
```

```
            return 1.0;
        }
        return 0.0;
    }
    virtual sp<MediaPlayerBase> createPlayer() {
        return new TestPlayerStub();
    }
};
```

这 4 种 MediaPlayer Factory 分别会生成不同的 MediaPlayer 基类：StagefrightPlayer、NuPlayerDriver、MidiFile 和 TestPlayerStub。四者均继承于 MediaPlayerBase。相关类图如图 6-2 所示。



▲图 6-2

抽象工厂方法模式在 Android 开发中使用得并不多，大家看到源码中的应用其实也是极少的，主要是在开发过程中很少会出现多个产品种类的情况，大部分情况我们使用工厂模式即可解决，因此，这里对该模式不再做过多的介绍。

6.7 总结

抽象工厂方法模式的优点：

一个显著的优点是分离接口与实现，客户端使用抽象工厂来创建需要的对象，而客户端根本就不知道具体的实现是谁，客户端只是面向产品的接口编程而已，使其从具体的产品实现中解耦，同时基于接口与实现的分离，使抽象该工厂方法模式在切换产品类时更加灵活、容易。

抽象工厂方法模式的缺点：

上面我们也有所提及，一是类文件的爆炸性增加，二是不太容易扩展新的产品类，因为每当我们增加一个产品类就需要修改抽象工厂，那么所有的具体工厂类均会被修改。

第7章 时势造英雄——策略模式

7.1 策略模式介绍

在软件开发中也常常遇到这样的情况：实现某一个功能可以有多种算法或者策略，我们根据实际情况选择不同的算法或者策略来完成该功能。例如，排序算法，可以使用插入排序、归并排序、冒泡排序等。

针对这种情况，一种常规的方法是将多种算法写在一个类中。例如，需要提供多种排序算法，可以将这些算法写到一个类中，每一个方法对应一个具体的排序算法；当然，也可以将这些排序算法封装在一个统一的方法中，通过 if...else...或者 case 等条件判断语句来选择具体的算法。这两种实现方法我们都可以称为硬编码。然而，当很多个算法集中在一个类中时，这个类就会变得臃肿，这个类的维护成本会变高，在维护时也更容易引发错误。如果我们需要增加一种新的排序算法，需要修改封装算法类的源代码。这就明显违反了我们上面所说的 OCP 原则和单一职责原则。

如果将这些算法或者策略抽象出来，提供一个统一的接口，不同的算法或者策略有不同的实现类，这样在程序客户端就可以通过注入不同的实现对象来实现算法或者策略的动态替换，这种模式的可扩展性、可维护性也就更高，也就是我们本章要说的策略模式。

7.2 策略模式的定义

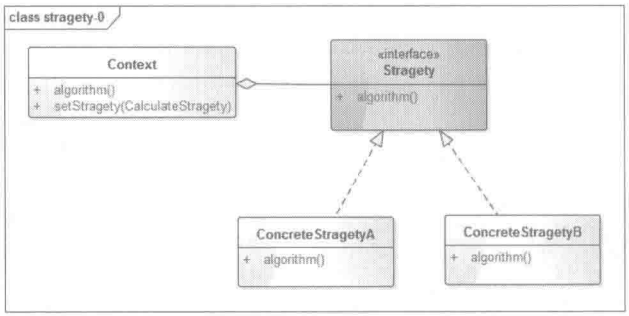
策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

7.3 策略模式的使用场景

- 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。
- 需要安全地封装多种同一类型的操作时。
- 出现同一抽象类有多个子类，而又需要使用 if-else 或者 switch-case 来选择具体子类时。

7.4 策略模式的 UML 类图

UML 类图如图 7-1 所示。



▲图 7-1

图 7-1 中的角色介绍：

- Context——用来操作策略的上下文环境；
- Stragety——策略的抽象；
- ConcreteStragetyA、ConcreteStragetyB——具体的策略实现。

7.5 策略模式的简单实现

通常如果一个问题有多个解决方案时，最简单的方式就是利用 if-else 或者 switch-case 方式根据不同的情景选择不同的解决方案，但这种简单的方案问题太多，例如耦合性太高、代码臃肿、难以维护等。但是，如果解决方案中包括大量的处理逻辑需要封装，或者处理方式变动较大的时候则就显得混乱、复杂，当需要增加一种方案时就需要修改类中的代码。怎么对于修改不关闭？不是说好的要遵循开闭原则吗？

是的，if-else 这种方法确实不会遵循开闭原则，而应对这种情况策略模式就能很好地解决这类问题，它将各种方案分离开来，让程序客户端根据具体的需求来动态地选择不同的策略方案。

下面我们以在北京坐公共交通工具的费用计算来演示一个简单示例。在 2014 年 12 月 28 号之后，北京提高公交价格，不再是单一票价制，而是分段计价，也就是说乘坐的距离越远，价格越高。

显然，公交车和地铁的价格计算方式是不一样的，但是，我们的示例中是需要计算乘不同出行工具的成本，下面是我们的第一个版本的代码：

```
public class PriceCalculator {
    // 公交车类型
    private static final int BUS = 1;
    // 地铁类型
    private static final int SUBWAY = 2;
    public static void main(String[] args) {
        PriceCalculator calculator = new PriceCalculator();
        System.out.println("坐 16 公里的公交车票价为 : " + calculator.calculatePrice(16, BUS));
```

```
        System.out.println("坐 16 公里的地铁票价为 : " + calculator.calculatePrice(16, SUBWAY));
    }

    /**
     * 北京公交车,十公里之内一元钱，超过十公里之后每加一元钱可以乘 5 公里
     * @paramkm 公里
     * @return
     */
    private int busPrice(int km) {
        // 超过十公里的总距离
        int extraTotal = km - 10;
        // 超过的距离是 5 公里的倍数
        int extraFactor = extraTotal / 5;
        // 超过的距离对 5 公里取余
        int fraction = extraTotal % 5;
        // 价格计算
        int price = 1 + extraFactor * 1;
        return fraction > 0 ? ++price : price ;
    }
    /**
     * 6 公里(含)内 3 元; 6~12 公里(含)4 元; 12~22 公里(含)5 元; 22~32 公里(含)6 元;
     * @paramkm 公里
     * @return
     */
    private int subwayPrice(int km) {
        if (km<= 6) {
            return 3;
        } elseif (km> 6 &&km< 12) {
            return 4;
        } elseif (km> 12 &&km< 22) {
            return 5;
        } elseif (km> 22 &&km< 32) {
            return 6;
        }
        // 其他距离我们简化为 7 元
        return 7;
    }
    int calculatePrice(int km, int type) {
        if (type == BUS) {
            return busPrice(km);
        } elseif (type == SUBWAY) {
            return subwayPrice(km);
        }
        return 0;
    }
}
```

PriceCalculator 类很明显的问题就是并不是单一职责，首先它承担了计算公交车和地铁乘坐价格的职责；另一个问题就是通过 if-else 的形式来判断使用哪种计算形式。当我们增加一种出行方式时，如出租车，那么我们就需要在 PriceCalculator 中增加一个方法来计算出租车出行的价格，并且在 calculatePrice(int km, int type)函数中增加一个判断，代码大致添加后如下：

```
public class PriceCalculator {
    // 公交车类型
    private static final int BUS = 1;
    // 地铁类型
    private static final int SUBWAY = 2;
    // 出租车类型
```

```
    private static final int TAXI = 3;
    /**
     * 简单计算为每公里 2 元钱
     *
     * @param km
     * @return
     */
    private int taxiPrice(int km) {
        return km * 2;
    }
    int calculatePrice(int km, int type) {
        if (type == BUS) {
            return busPrice(km);
        } else if (type == SUBWAY) {
            return subwayPrice(km);
        } else if (type == TAXI) {
            return taxiPrice(km) ;
        }
        return 0 ;
    }
}
```

此时的代码已经比较混乱，各种 if-else 语句缠绕其中。当价格的计算方法变化时，需要直接修改这个类中的代码，那么很可能有一段代码是其他几个计算方法所共同使用的，这就容易引入错误。另外，在增加出行方式时，我们又需要在 calculatePrice 中添加 if-else，此时很有可能就是复制上一个 if-else，然后手动进行修改，手动复制代码也是最容易引入错误的做法之一。这类代码必然是难以应对变化的，它会使得代码变得越来越臃肿，难以维护，我们解决这类问题的手法也就是本章谈到的策略模式。当然，我们也可以把每种计算方法独立成一个函数，然后外部调用对应的方法即可，但这也是另一种耦合的形式，对于可变性较大的算法族来说还是不适合使用这种方式。

下面我们对上述示例用策略模式进行重构。

首先我们需要定义一个抽象的价格计算接口，这里命名为 CalculateStrategy，具体代码如下：

```
/**
 * 计算接口
 */
public interface CalculateStrategy {
    /**
     * 按距离来计算价格
     * @param km 公里
     * @return 返回价格
     */
    int calculatePrice(int km);
}
```

对于每一种出行方式我们都有一个独立的计算策略类，这些策略类都实现了 CalculateStrategy 接口，例如下面是公交车和地铁的计算策略类：

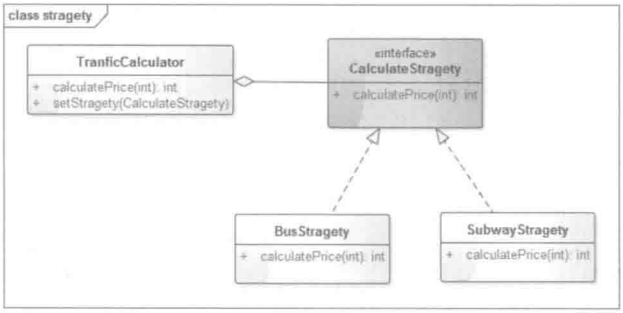
```
    // 公交车价格计算策略
public class BusStrategy implements CalculateStrategy {
    /**
     * 北京公交车，十公里之内一元钱，超过十公里之后每加一元钱可以乘 5 公里
     * @param km 公里
     * @return
     */
    @Override
```

```
    public int calculatePrice(int km) {
        // 超过 10 公里的总距离
        int extraTotal = km - 10;
        // 超过的距离是 5 公里的倍数
        int extraFactor = extraTotal / 5;
        // 超过的距离对 5 公里取余
        int fraction = extraTotal % 5;
        // 价格计算
        int price = 1 + extraFactor * 1;
        return fraction > 0 ? ++price : price;
    }

}

    // 地铁价格计算策略
public class SubwayStrategy implements CalculateStrategy {

    /**
     * 6 公里(含)内 3 元; 6~12 公里(含)4 元; 12~22 公里(含)5 元; 22~32 公里(含)6 元。
     * @param km 公里
     * @return
     */
    @Override
    public int calculatePrice(int km) {
        if (km <= 6) {
            return 3;
        } else if (km > 6 && km < 12) {
            return 4;
        } else if (km > 12 && km < 22) {
            return 5;
        }
        else if (km > 22 && km < 32) {
            return 6;
        }
        // 其他距离我们简化为 7 元钱
        return 7;
    }

}
```

我们再创建一个扮演 Context 角色的类，这里将它命名为 TranficCalculator，具体代码如下：

```
    // 公交出行价格计算器
public class TranficCalculator {
    public static void main(String[] args) {
        TranficCalculator calculator = new TranficCalculator();
        // 设置计算策略
        calculator.setStrategy(new BusStrategy());
        // 计算价格
        System.out.println("公交车乘 16 公里的价格 : " + calculator.calculatePrice(16));
    }
    CalculateStrategy mStrategy;
    public void setStrategy(CalculateStrategy mStrategy) {
        this.mStrategy = mStrategy;
    }
    public int calculatePrice(int km) {
        return mStrategy.calculatePrice(km);
    }
}
```

经过上述的重构之后，去掉了各种各样的 if-else 语句，结构变得也很清晰，其结构如图 7-2 所示。



▲图 7-2

这种方案在隐藏实现的同时，可扩展性变得很强，例如，当我们需要增加出租车的计算策略时，只需要添加一个出租车计算策略类，然后将该策略设置给 TranficCalculator，最好直接通过 TranficCalculator 对象的计算方法即可。示例代码如下：

```
    // 出租车计算策略
public class TaxiStrategy implements CalculateStrategy {
    // 价格我们简单计算为公里数 * 2
    @Override
    public int calculatePrice(int km) {
        return km * 2;
    }

}
```

将策略注入到 TranficCalculator 中。

```
    // 公交车出行价格计算器
public class TranficCalculator {
    public static void main(String[] args) {
        TranficCalculator calculator = new TranficCalculator();
        // 设置计算策略
        calculator.setStrategy(new TaxiStrategy());
        System.out.println("出租车乘 16 公里的价格:" + calculator.calculatePrice(16));
    }
    CalculateStrategy mStrategy;
    public void setStrategy(CalculateStrategy stragety) {
        this. mStrategy = stragety;
    }
    public int calculatePrice(int km) {
        return mStrategy.calculatePrice(km);
    }
}
```

通过上述示例我们可以清晰地看出二者的区别所在。前者通过 if-else 来解决问题，虽然实现较为简单，类型层级单一，但暴露的问题非常明显，即代码臃肿，逻辑复杂，难以升级和维护，没有结构可言；后者则是通过建立抽象，将不同的策略构建成一个具体的策略实现，通过不同的策略实现算法替换。在简化逻辑、结构的同时，增强了系统的可读性、稳定性、可扩展性，这对于较为复杂的业务逻辑显得更为直观，扩展也更为方便。

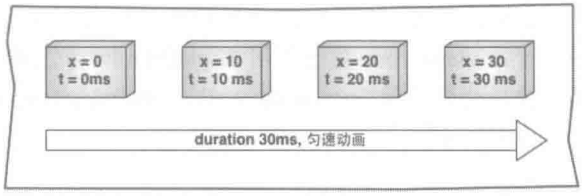
7.6 Android 源码中的策略模式实现

随着技术的发展，工程师们已经越来越重视用户体验、用户交互。因此，动画成了很多应用中必不可少的部分，一个简单的引导页面也要做成动画的效果，一个按钮的隐藏也需要加入淡入淡出的动画效果。动画的实现原理就是在短时间内快速地进行画面切换，这个切换频率需要达到人眼感觉不出卡顿，例如标准的电影是 24 帧/秒。在比较流畅时，Android 上的动画能够达到 60 帧/秒，人眼基本看不出间隔，所以，在我们看到这个动画就非常流畅。

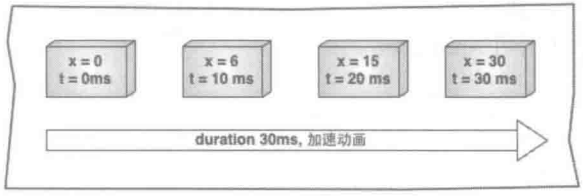
单纯是动画还不足以满足我们的需求，在动画执行的过程中，我们还需要一些动态效果，这有点类似于电影的慢镜头，有的时候我们需要它慢一点，有的时候需要快一点，这样动画也变得灵动起来。这些动态效果就是通过插值器（TimeInterpolator）实现的，我们只需要对 Animation 对象设置不同的插值器就可以实现不同的动态效果。

7.6.1 时间插值器

在开始之前我们需要先了解动画中的 TimeInterpolator，也就是时间插值器。它的作用是根据时间流逝的百分比来计算出当前属性值改变的百分比，系统预置的有线性插值器（LinearInterpolator）用于匀速动画；加速减速插值器（AccelerateDecelerateInterpolator）用于起始时动画加速，结尾时动画减速；减速插值器（DecelerateInterpolator）用于随着时间的推移动画越来越慢，即减速动画。这些插值器就是策略模型的典型运用。可能这么说还有点抽象，我们看如图 7-3 和图 7-4 所示。



▲图 7-3 （匀速动画）



▲图 7-4 （加速动画）

图 7-3 中是一个匀速动画，其采用了线性插值器，在 30ms 内，View 的 x 属性实现从 0 到 30 的变换，由于是匀速动画，因此，在相同的时间间隔内 x 的变化范围是一致的，例如，在时间 $t = 0$ 时，x 的坐标为 0；当 $t = 10$ ms 时，$x = 10$。也就是说当在时间间隔为 10ms 时，x 的变化范围都是

10，所以，形成了速度不变的动画。再看图 7-4 中的加速动画，在相同的时间间隔内，时间越靠后速度变得越快，也就是说在同等的时间间隔内 x 的变化范围越来越大，也就形成了加速动画的效果。

当然插值器并不是一个“人在战斗”，它还有一位“好基友”—TypeEvaluator，也就是类型估值器。它的作用是根据当前属性改变的百分比来计算改变后的属性值，也就是说 TypeEvaluator 计算得到的才是属性的值。时间插值器计算得到当前时间点的时间流逝百分比，TypeEvaluator 根据这个百分比、属性起始值、目标值来计算出当前时刻该属性的值，最后这个值被设置给 View，不断地重复这个过程就形成了动画。系统预置的有整型属性（IntEvaluator）、浮点型属性（FloatEvaluator）和 Color 属性（ArgbEvaluato）。

7.6.2　动画中的时间插值器

时间插值器运用于动画中，而动画作用于 View 上。因此，需要从 View 上切入到动画，再从动画中寻找插值器的身影。当我们要对某个 View 执行某个动画时，我们首先会构建一个 Animation 对象，然后调用 View 的 startAnimation(Animation animation)方法，此时动画就启动了：

```
public void startAnimation(Animation animation) {
    // 1. 初始化动画开始时间
    animation.setStartTime(Animation.START_ON_FIRST_FRAME);
    // 2. 对 View 设置动画
    setAnimation(animation);
    //3. 刷新父类缓存
    invalidateParentCaches();
    //4. 刷新 View 本身及子 View
    invalidate(true);
}
```

startAnimation 中首先设置了动画的起始时间，然后将该动画设置到该 View 中，最后再向 ViewGroup 请求刷新视图，随后 ViewGroup 就会调用 dispatchDraw 方法对这个 View 所在的区域进行重绘。对于某个 View 的重绘最终会调用 ViewGroup 中的 drawChild(Canvas canvas, View child, long drawingTime)方法，我们看看这个函数：

```
protected boolean drawChild(Canvas canvas, View child, long drawingTime) {
        return child.draw(canvas, this, drawingTime);
    }
```

只是进行了一个转发，我们再看一下 View 的 draw(Canvas canvas, ViewGroup parent, long drawingTime)方法中是如何调用 Animation 的：

```
boolean draw(Canvas canvas, ViewGroup parent, long drawingTime) {
    //......
    //查看是否需要清除动画信息
    final int flags = parent.mGroupFlags;
    // 代码省略
    //获取设置的动画信息
    final Animation a = getAnimation();

    if (a != null) {
        //绘制动画
        more = drawAnimation(parent, drawingTime, a, scalingRequired);
        // 代码省略
    } else {
```

```
        //...
    }
}
```

可以看出在父类调用 View 的 draw 方法中，会先判断是否设置了清除动画的标记，然后再获取该 View 动画的信息，如果设置了动画，就会调用 View 中的 drawAnimation 方法，具体代码如下：

```
private boolean drawAnimation(ViewGroup parent, long drawingTime,
        Animation a, boolean scalingRequired) {

    Transformation invalidationTransform;
    final int flags = parent.mGroupFlags;
    //1. 判断动画是否已经初始化过
    final boolean initialized = a.isInitialized();
    if (!initialized) {
    // 初始化动画，代码省略
    // 如果设置了动画的监听,则触发对应的回调
        onAnimationStart();
    }

    // 获取 Transformation 对象，存储动画的信息
    final Transformation t = parent.getChildTransformation();
    // 2. 调用了 Animation 的 getTransformation，这里就是通过计算获取动画的相关值
    boolean more = a.getTransformation(drawingTime, t, 1f);
    // 代码省略
    if (more) {
        //3. 根据具体实现，判断当前动画类型是否需要进行调整位置大小，然后刷新不同的区域
        if (!a.willChangeBounds()) {
            //...
        }else{
            // 获取重绘的区域
            a.getInvalidateRegion(0, 0, mRight - mLeft, mBottom - mTop, region,
                    invalidationTransform);
            parent.mPrivateFlags |= PFLAG_DRAW_ANIMATION;
            // 重新计算有效区域
            final int left = mLeft + (int) region.left;
            final int top = mTop + (int) region.top;
            // 更新这块区域
            parent.invalidate(left, top, left + (int) (region.width() + .5f),
            top + (int) (region.height() + .5f));
        }
    }
    return more;
}
```

在 drawAnimation 中主要的操作是动画初始化、动画操作、界面刷新。在 drawAnimation 中首先会判断动画是否进行了初始化，如果未初始化则先初始化，然后调用动画监听器的 onStart 函数。动画的具体实现是通过 Animation 中的 getTransformation(long currentTime, Transformation outTransformation,float scale)方法，我们看看相关代码：

```
public boolean getTransformation(long currentTime, Transformation outTransformation,
float scale) {
    mScaleFactor = scale;
    return getTransformation(currentTime, outTransformation);

}
```

在上面的方法中，主要是获取缩放系数和调用 Animation.getTransformation(long currentTime,

Transformation outTransformation)来计算和应用动画效果：

```
Interpolator mInterpolator;  // 插值器成员变量
public boolean getTransformation(long currentTime, Transformation outTransformation) {
    // 代码省略
    float normalizedTime;
    // 1. 计算当前时间的流逝百分比
    if (duration != 0) {
        normalizedTime = ((float) (currentTime - (mStartTime + startOffset))) /
                (float) duration;
    } else {
        // time is a step-change with a zero duration
        normalizedTime = currentTime < mStartTime ? 0.0f : 1.0f;
    }
    // 动画是否已经完成
    final boolean expired = normalizedTime >= 1.0f;
    mMore = !expired;

    // 代码省略
    if((normalizedTime>=0.0f|| mFillBefore) && (normalizedTime<=1.0f || mFillAfter)) {
       // 代码省略
        // 2. 通过插值器获取动画执行百分比
        final float interpolatedTime = mInterpolator.getInterpolation(normalizedTime);
        // 3. 应用动画效果
        applyTransformation(interpolatedTime, outTransformation);
    }

    // 4. 如果动画执行完毕,那么触发动画完成的回调或者执行重复动画等操作
    // 代码省略

    if (!mMore && mOneMoreTime) {
        mOneMoreTime = false;
        return true;
    }
    return mMore;
}
```

在上述函数中，首先会获取已经流逝的动画执行时间百分比，然后再通过插值器来重新计算这个百分比，也就是调用了插值器的 getInterpolation(float input)方法来获取当前的时间百分比，并且以此来计算当前动画的属性值，例如，线性插值器的输出百分比就是输入的百分比，不做任何处理，使得动画的速率不会发生变化：

```
public class LinearInterpolator implements Interpolator {
    public float getInterpolation(float input) {
        return input;
    }
}
```

我们再看看加速插值器的代码：

```
public class AccelerateInterpolator implements Interpolator {
    private final float mFactor;
    private final double mDoubleFactor;
    public AccelerateInterpolator() {
        mFactor = 1.0f;
        mDoubleFactor = 2.0;
    }
```

```
    public float getInterpolation(float input) {
        // 默认为1.0f,随着时间的推移,变化范围越大
        if (mFactor == 1.0f) {
            return input * input;
        } else {
            return (float)Math.pow(input, mDoubleFactor);
        }
    }
}
```

我们看到，在默认情况下，AccelerateInterpolator 的 getInterpolation 方法中会对 input 进行乘方操作，这个 input 就是流逝的时间百分比，input 的取值为 0.0f～1.0f，当 input 逐渐增大时，input * input 的变化范围越来越大，使得动画的属性值在同一时间段内的变化范围更大，从而实现了加速动画的效果。例如，动画执行总时间为 1 秒，使用的是 AccelerateInterpolator 插值器，在动画执行了 100 毫秒时百分比为 0.1，那么此时通过插值器的计算，百分比成了 0.01；又过了 100 毫秒，此时百分比为 0.2，经过插值器的计算变为 0.04；在执行到 300 毫秒时，百分比经过插值器计算会变为 0.09。我们看到，在相同的 100 毫秒内百分比的变化频率逐渐增大，100 到 200 毫秒之间的变化值为 0.03，200 到 300 毫秒之前的变化范围则是 0.05，这样在同一时间段内百分比差距越来越大，也就形成了加速的效果。

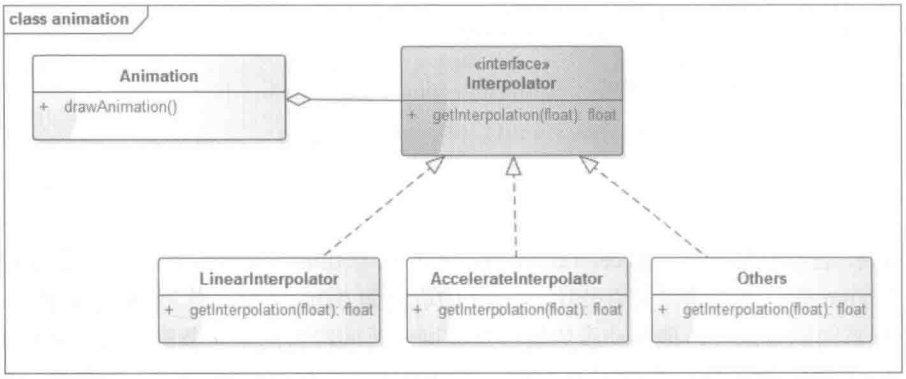
在调用了插值器的 getInterpolation 方法之后，会继续调用动画类的 applyTransformation 方法将属性应用到对应的对象中。applyTransformation 在 Animation 基类中是空实现，那我们选择缩放动画（ScaleAnimation）来看看它的具体实现：

```
@Override
  protected void applyTransformation(float interpolatedTime, Transformation t) {
      float sx = 1.0f;
      float sy = 1.0f;
      float scale = getScaleFactor();
      if (mFromX != 1.0f || mToX != 1.0f) {
          sx = mFromX + ((mToX - mFromX) * interpolatedTime);
      }
      if (mFromY != 1.0f || mToY != 1.0f) {
          sy = mFromY + ((mToY - mFromY) * interpolatedTime);
      }
      // 通过Matrix实现View的缩放
      if (mPivotX == 0 && mPivotY == 0) {
          t.getMatrix().setScale(sx, sy);
      } else {
          t.getMatrix().setScale(sx, sy, scale * mPivotX, scale * mPivotY);
      }
  }
```

当执行完 applyTransformation 之后，View 的属性就发生了变化，不断地重复这个过程，动画就随之产生了。

这个过程中，插值器扮演了很重要的角色。它将动画的速率计算封装到一个抽象中，也就是 Interpolator 中，该接口中只有一个 getInterpolation(float input)方法，通过这个方法来修改动画的流逝时间百分比，以此达到动画的加速、减速等效果，结构如图 7-5 所示。

Interpolator 就是这个计算策略的抽象，LinearInterpolator、CycleInterpolator 等插值器就是具体的实现策略，通过注入不同的插值器实现不同的动态效果。



▲图 7-5

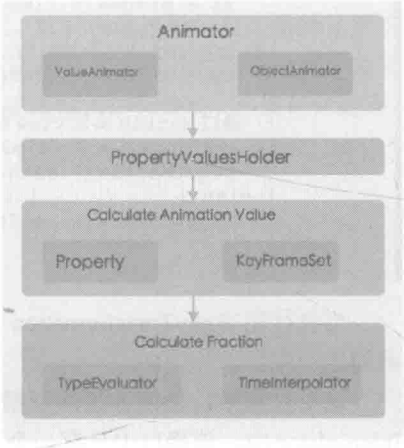
7.7 深入属性动画

在 Android 3.0 之前，Android 提供了几种动画类型：View Animation、Drawable Animation、Property Animation。View Animation 相当简单，不过只能支持简单的缩放、平移、旋转、透明度这几个基本的动画，且有一定的局限性。例如：希望 View 有一个颜色的切换动画；希望可以使用 3D 旋转动画；希望当动画停止时，View 的位置就是当前的位置。这些 View Animation 都无法做到。因此，Google 在 Android 3.0 提供了属性动画，这使得动画系统变得极其强大。考虑到很多应用会兼容 Android 3.0 以下的设备，在这种情况下就需要兼容 Android 3.0 以下的动画库，Nineoldanimations 就是这么一个兼容库，它使得 Android 3.0 一下的设备也能够使用属性动画。NineOldAnimations 通过判断系统版本来选择实现属性动画的方式，它的 API 与 Android 自带的接口一致，因此，在这里我们选择适用性更广的 NineOldAnimations 作为分析对象。

7.7.1　属性动画体系的总体设计

总体设计如图 7-6 所示。

图 7-6 是属性动画体系的整体设计，Animator 通过 PropertyValuesHolder 来更新对象的目标属性，如果用户没有设置目标属性的 Property 对象，那么会通过反射的形式调用目标属性的 setter 方法来更新属性值；否则，通过 Property 的 set 方法来设置属性值。这个属性值则通过 KeyFrameSet 的计算得到，而 KeyFrameSet 又是通过时间插值器和类型估值器来计算，在动画执行过程中不断地计算当前时刻目标属性的值，然后更新属性值来达到动画效果。



▲图 7-6

7.7.2　属性动画的核心类介绍

在进行下一步的分析之前，我们先来了解一下属性动画中一些核心的类以及它们的作用。

- ValueAnimator：该类是 Animator 的子类，实现了动画的整个处理逻辑，也是属性动画最为

核心的类。

- ObjectAnimator：对象属性动画的操作类，继承自 ValueAnimator，通过该类使用动画的形式操作对象的属性。
- TimeInterpolator：时间插值器，它的作用是根据时间流逝的百分比来计算出当前属性值改变的百分比，系统预置的有线性插值器（LinearInterpolator）、加速减速插值器（AccelerateDecelerate nterpolator）和减速插值器（DecelerateInterpolator）等。
- TypeEvaluator：TypeEvaluator 的中文翻译为类型估值算法，它的作用是根据当前属性改变的百分比来计算改变后的属性值，系统预置的有针对整型属性（IntEvaluator）、针对浮点型属性（FloatEvaluator）和针对 Color 属性（ArgbEvaluator）。
- Property：属性对象，主要是定义了属性的 set 和 get 方法。
- PropertyValuesHolder：PropertyValuesHolder 是持有目标属性 Property、setter 和 getter 方法、以及关键帧集合的类。
- KeyframeSet：存储一个动画的关键帧集合。
- AnimationProxy：在 Android 3.0 以下使用 View 的属性动画的辅助类。

7.7.3 基本使用

示例 1：改变一个对象（myObject）的 translationY 属性，让其沿着 y 轴向上平移一段距离，该动画在默认时间内完成，动画的完成时间是可以定义的，想要更灵活的效果还可以定义插值器和估值算法，但是，一般来说我们不需要自定义，系统已经预置了一些，能够满足常用的动画：

```
ObjectAnimator.ofFloat(myObject, "translationY", -myObject.getHeight()).start();
```

示例 2：改变一个对象的背景色属性，典型的情形是改变 View 的背景色，下面的动画可以让背景色在 3s 内实现从 0xFFFF8080 到 0xFF8080FF 的渐变，并且动画会无限循环而且会有反转的效果：

```
ValueAnimator colorAnim = ObjectAnimator.ofInt(this, "backgroundColor", /*Red*/
0xFFFF8080,/*Blue*/0xFF8080FF);
colorAnim.setDuration(3000);
colorAnim.setEvaluator(new ArgbEvaluator());
colorAnim.setRepeatCount(ValueAnimator.INFINITE);
colorAnim.setRepeatMode(ValueAnimator.REVERSE);
colorAnim.start();
```

示例 3：动画集合，5s 内对 View 的旋转、平移、缩放和透明度都进行了改变：

```
AnimatorSet set = new AnimatorSet();
set.playTogether(
    ObjectAnimator.ofFloat(myView, "rotationX", 0, 360),
    ObjectAnimator.ofFloat(myView, "rotationY", 0, 180),
    ObjectAnimator.ofFloat(myView, "rotation", 0, -90),
    ObjectAnimator.ofFloat(myView, "translationX", 0, 90),
    ObjectAnimator.ofFloat(myView, "translationY", 0, 90),
    ObjectAnimator.ofFloat(myView, "scaleX", 1, 1.5f),
    ObjectAnimator.ofFloat(myView, "scaleY", 1, 0.5f),
    ObjectAnimator.ofFloat(myView, "alpha", 1, 0.25f, 1)
);
```

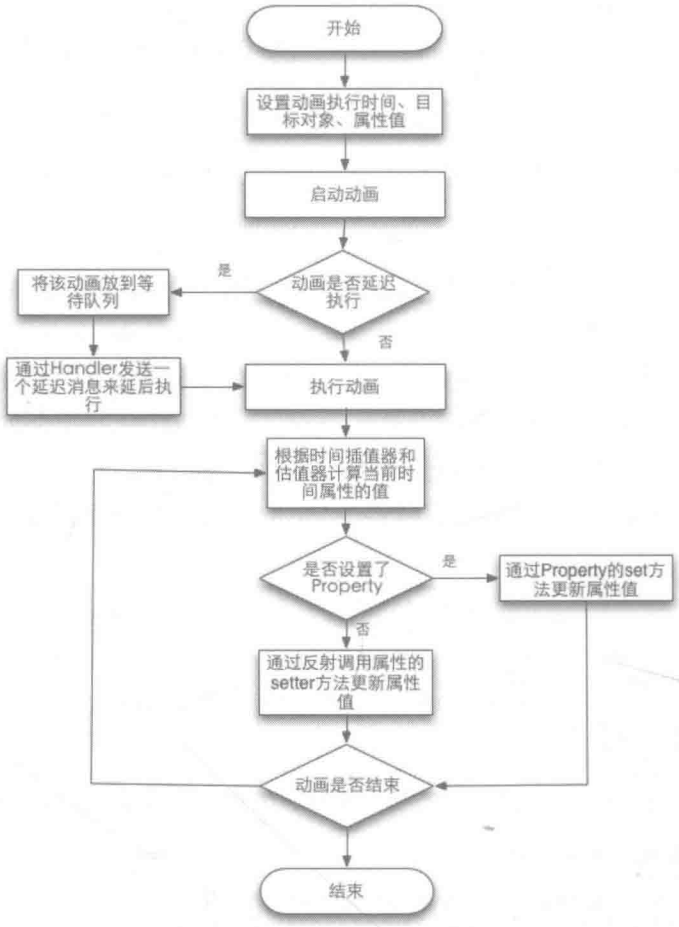
```
set.setDuration(5 * 1000).start();
```

示例 4：下面是个简单的调用方式，其 animate 方法是属性动画特有的，动画持续时间为 2s，在 y 轴上旋转 720°，并且平移到（100, 100）的位置：

```
Button myButton = (Button)findViewById(R.id.myButton);
animate(myButton).setDuration(2000).rotationYBy(720).x(100).y(100);
```

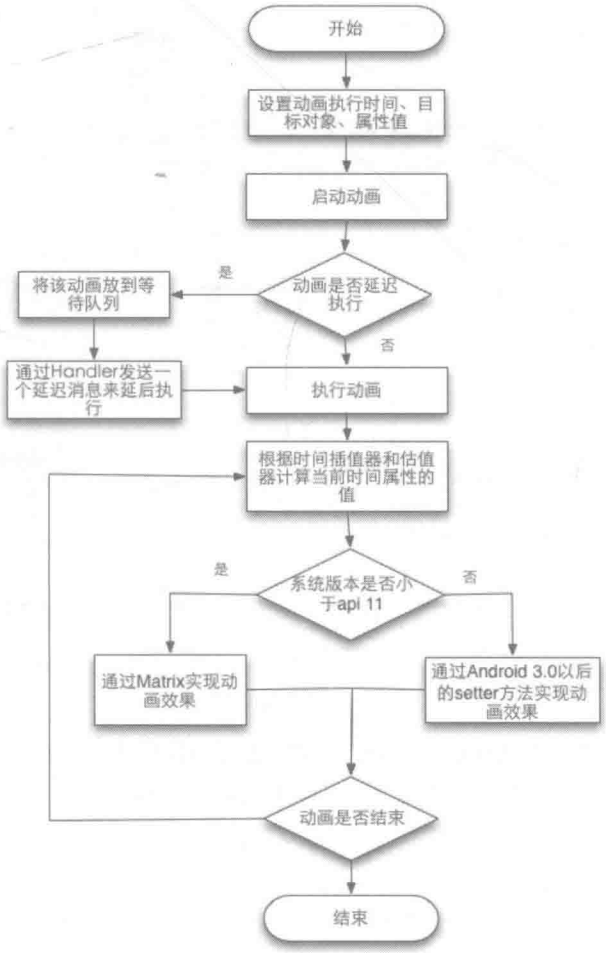
7.7.4　流程图

1. ValueAnimator 流程图（如图 7-7 所示）



▲图 7-7

2. ObjectAnimator 流程图（如图 7–8 所示）



▲图 7–8

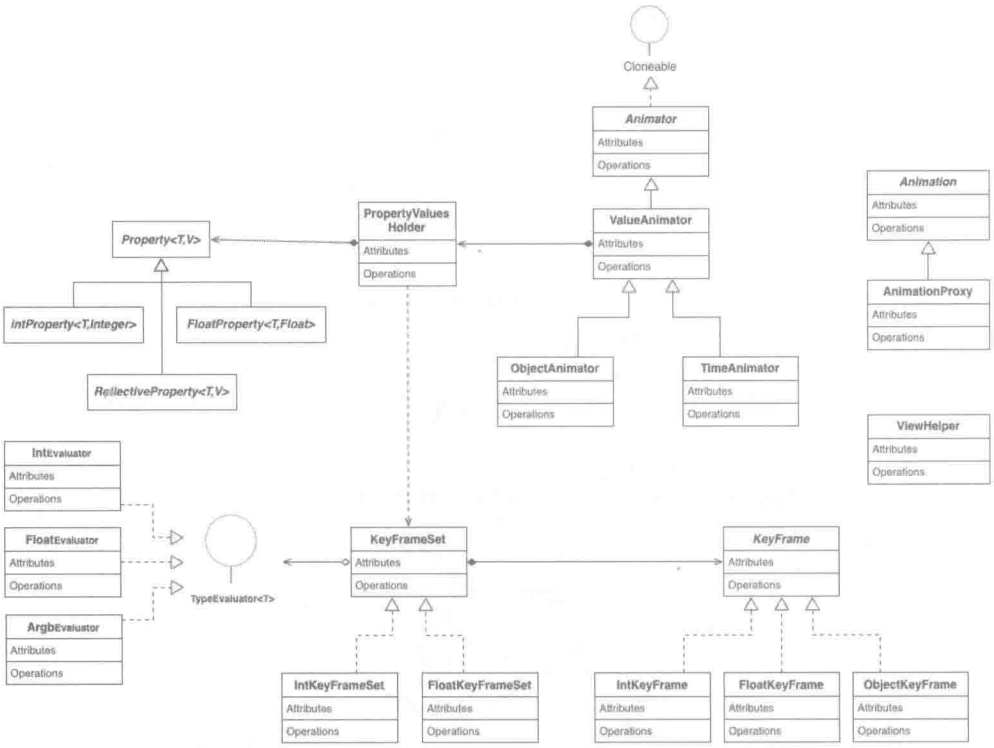
7.7.5 详细设计

详细设计如图 7-9 所示。

7.7.6 核心原理分析

我们以一个具体的示例作为分析的入口，例如，要对某个 View 的 x 轴缩放到原来的 0.3 倍，动画执行时间为 1s，具体代码如下：

```
ValueAnimator colorAnim=ObjectAnimator.ofFloat(myView,"scaleX",0.3f);
colorAnim.setDuration(1000);
colorAnim.start();
```



▲图 7-9

它的背后是什么原理，在 Android 3.0 以上和以下的系统版本中它的执行代码有什么不一样，这就是我们本节要研究的。首先我们从它的入口，即 ObjectAnimator 入手：

```
public static ObjectAnimator ofFloat(Object target, String propertyName, float... val
ues) {
        // 1. 构建动画对象
    ObjectAnimator anim = new ObjectAnimator(target, propertyName);
    // 2. 设置属性值
    anim.setFloatValues(values);
    return anim;
}
```

在 ObjectAnimator 的 ofFloat 函数中会先构建属性动画对象，然后根据设置的属性值来初始化各个时间段对应的属性值，这个属性值就是 values 参数，它是一个可变参数，如果是一个参数，那么该参数为目标值；如果是两个参数，那么一个是起始值，另一个是目标值。我们先看看 setFloatValues 函数的实现：

```
@Override
public void setFloatValues(float... values) {
        // mValues 是动画的各个数值的集合
        PropertyValuesHolder[] mValues;
    if (mValues == null || mValues.length == 0) {
        if (mProperty != null) {
            setValues(PropertyValuesHolder.ofFloat(mProperty, values));
```

```
        } else {
            setValues(PropertyValuesHolder.ofFloat(mPropertyName, values));
        }
    } else {
        super.setFloatValues(values);
    }
}
```

我们看到 setFloatValues 出现了一个类 PropertyValuesHolder，这个类是该动画库的一个核心类之一，它的作用就是保存属性的名称和它的 setter、getter 方法，以及它的目标值。我们看看它的一些相关函数：

```
public class PropertyValuesHolder implements Cloneable {
    // 属性名称
    String mPropertyName;
    // 属性对象
    protected Property mProperty;
    //属性的 setter 方法
    Method mSetter = null;
    //  属性的 getter 方法
    private Method mGetter = null;

    // 属性值的类类型，如 float、int 等。
    Class mValueType;
    // 这里是动画关键帧的即可，即在 duration 时间内的动画帧集合，它保存的是在每个时刻该属性对应的值
    KeyframeSet mKeyframeSet = null;

    public static PropertyValuesHolder ofFloat(Property<?, Float> property, float...
    values) {
        // 1. 构建的是 FloatPropertyValuesHolder
        return new FloatPropertyValuesHolder(property, values);
    }

    // 内部类，Float 类型的 PropertyValuesHolder
    static class FloatPropertyValuesHolder extends PropertyValuesHolder {
        //int mJniSetter;
        private FloatProperty mFloatProperty;// float 型属性
        FloatKeyframeSet mFloatKeyframeSet;// 动画的关键帧，这个是重点
        float mFloatAnimatedValue;

                // 代码省略
        // 2. 构造函数
        public FloatPropertyValuesHolder(Property property, float... values) {
            super(property);
            // 设置目标属性值
            setFloatValues(values);
            if (property instanceof  FloatProperty) {
                mFloatProperty = (FloatProperty) mProperty;
            }
        }

        // 3. 设置动画的目标值
        @Override
        public void setFloatValues(float... values) {
            // 4. 调用了父类的方法
            super.setFloatValues(values);
            mFloatKeyframeSet = (FloatKeyframeSet) mKeyframeSet;// 5. 获取动画关键帧
        }

        // 计算当前的动画值
```

```
        @Override
        void calculateValue(float fraction) {
            mFloatAnimatedValue = mFloatKeyframeSet.getFloatValue(fraction);
        }
        // 代码省略
        }
}
```

该类是属性和属性值的辅助类，它保存了属性的名称、setter、getter，以及该属性在 duration 时间段内各个时刻对应属性数值（mKeyframeSet）。这样，当执行动画时，动画库只需要根据动画的执行时间，到 mKeyframeSet 中查询这个时刻对应的属性值，然后修改执行动画的对象的目标属性值，连续这个过程即可达到动画的效果。在这个例子中，我们的属性是 scaleX，目标属性值是 0.3f。因此，对应的属性类为 FloatPropertyValuesHolder，还有一种是 IntPropertyValuesHolder，这都很好理解，就不多介绍了。

这个例子中，我们会调用注释 2 中的构造函数，该函数会调用 setFloatValues 来设置动画的目标值，然后会到达注释 3 处的函数，在这个函数中调用了父类中对应的方法，然后就获取到了动画的关键帧，这有可能计算各个时刻的属性值的操作放在了父类（即 PropertyValuesHolder）的 setFloatValues 函数中。我们一起来看看 setFloatValues 实现：

```
public void setFloatValues(float... values) {
    mValueType = float.class;
    mKeyframeSet = KeyframeSet.ofFloat(values);
}
```

可以看到该函数又调用了 KeyFrameSet 的 ofFloat 方法，继续看下面的程序：

```
class KeyframeSet {
    int mNumKeyframes;
    Keyframe mFirstKeyframe;
    Keyframe mLastKeyframe;
    Interpolator mInterpolator; // only used in the 2-keyframe case
    ArrayList<Keyframe> mKeyframes; // only used when there are not 2 keyframes
    TypeEvaluator mEvaluator;

    public KeyframeSet(Keyframe... keyframes) {
        mNumKeyframes = keyframes.length;
        mKeyframes = new ArrayList<Keyframe>();
        mKeyframes.addAll(Arrays.asList(keyframes));
        mFirstKeyframe = mKeyframes.get(0);
        mLastKeyframe = mKeyframes.get(mNumKeyframes - 1);
        mInterpolator = mLastKeyframe.getInterpolator();
    }

    public static KeyframeSet ofFloat(float... values) {
        int numKeyframes = values.length;
        FloatKeyframe keyframes[] = new FloatKeyframe[Math.max(numKeyframes,2)];
        if (numKeyframes == 1) {
            keyframes[0] = (FloatKeyframe) Keyframe.ofFloat(0f);
            keyframes[1] = (FloatKeyframe) Keyframe.ofFloat(1f, values[0]);
        } else {
            keyframes[0] = (FloatKeyframe) Keyframe.ofFloat(0f, values[0]);
            for (int i = 1; i < numKeyframes; ++i) {
                keyframes[i] = (FloatKeyframe) Keyframe.ofFloat((float) i /
                (numKeyframes - 1), values[i]);
            }
```

```
        }
        return new FloatKeyframeSet(keyframes);
    }

}
```

我们看到关键帧的计算就在 ofFloat 函数中。如果用户设置了一个目标值，那么这个值就是最终的值，它的起始值会被默认设置为 0；如果用户设置了大于等于 1 个目标值，这些关键帧都会被存储到 KetFrameSet 对象中。设置完关键帧之后，我们就会调用 start()方法启动动画了。我们看看 ObjectAnimator 的 start()方法：

```
@Override
public void start() {
    // 输出 Log 的代码，省略
    super.start();
}
```

这支持调用了父类的 start()方法，它的父类是 ValueAnimator，我们看看该类的 start()方法：

```
@Override
public void start() {
    start(false);
}

private void start(boolean playBackwards) {
    // 判断 Looper 是否为空,这里的 Looper 是 UI 线程的 Looper
    if (Looper.myLooper() == null) {
        throw new AndroidRuntimeException("Animators may only be run on Looper threads");
    }
    // 设置一些基本状态
    mPlayingBackwards = playBackwards;
    mCurrentIteration = 0;
    mPlayingState = STOPPED;
    // 启动动画
    mStarted = true;
    mStartedDelay = false;
    // 将该动画加入到等待执行的动画队列中
    sPendingAnimations.get().add(this);
    // 是否延时
    if (mStartDelay == 0) {
        setCurrentPlayTime(getCurrentPlayTime());
//设置动画的开始执行时间，因为动画会有一个 duration
        // 这个开始执行时间+duration 就是结束时间
        mPlayingState = STOPPED;
        mRunning = true;
        // 触发动画监听器，代码省略
    }
    AnimationHandler animationHandler = sAnimationHandler.get();
    // 代码省略
    // 发送一个动画启动的消息给 AnimationHandler
    animationHandler.sendEmptyMessage(ANIMATION_START);
}
```

我们看到，启动动画的操作设置了一些基本参数，然后把自己添加到待执行的动画列表（sPendingAnimations）中，通过 AnimationHandler 发送了一个 ANIMATION_START 消息，真是哪里都有 Handler 的事啊！我们继续看下面的程序：

```
@Override
public void handleMessage(Message msg) {
    boolean callAgain = true;
    // 正要执行的动画列表
    ArrayList<ValueAnimator> animations = sAnimations.get();
    // 需要 delay 执行的动画列表
    ArrayList<ValueAnimator> delayedAnims = sDelayedAnims.get();
    switch (msg.what) {
        case ANIMATION_START:
            // 还未执行的动画列表
            ArrayList<ValueAnimator> pendingAnimations = sPendingAnimations.get();
            if (animations.size() > 0 || delayedAnims.size() > 0) {
                callAgain = false;
            }
            // 执行完正在等待执行的动画列表，我们当前的动画在 start()方法中就是添加到这个列表中的
            // 即这一行，sPendingAnimations.get().add(this);
            while (pendingAnimations.size() > 0) {
                // 复制一份等待执行的动画列表
                ArrayList<ValueAnimator> pendingCopy =
                        (ArrayList<ValueAnimator>) pendingAnimations.clone();
                // 清空 pendingAnimations
                pendingAnimations.clear();
                int count = pendingCopy.size();
                for (int i = 0; i < count; ++i) {
                    ValueAnimator anim = pendingCopy.get(i);
                    // 如果这个动画设置了延时,那么把它放到延迟执行的动画列表中，否则直接启动该动画
                    if (anim.mStartDelay == 0) {
                        // 启动动画
                        anim.startAnimation();
                    } else {
                        // 加入到延迟执行的动画列表中
                        delayedAnims.add(anim);
                    }
                }
            }

        case ANIMATION_FRAME:
            // 正在执行的动画数量
            int numAnims = animations.size();
            int i = 0;
            while (i < numAnims) {
                ValueAnimator anim = animations.get(i);
                if (anim.animationFrame(currentTime)) {
                    endingAnims.add(anim);
                }
                if (animations.size() == numAnims) {
                    ++i;
                } else {
                    --numAnims;
                    endingAnims.remove(anim);
                }
            }
            // 结束动画
            if (endingAnims.size() > 0) {
                for (i = 0; i < endingAnims.size(); ++i) {
                    endingAnims.get(i).endAnimation();
                }
                endingAnims.clear();
            }

            // 继续发送动画帧消息
            if (callAgain && (!animations.isEmpty() || !delayedAnims.isEmpty())) {
```

```
                sendEmptyMessageDelayed(ANIMATION_FRAME, Math.max(0, sFrameDelay -
                        (AnimationUtils.currentAnimationTimeMillis() - currentTime)));
            }
            break;
        }
    }
```

该 AnimationHandler 的作用简单来说就是处理动画的启动和各个关键帧动画的执行，在 ANIMATION_START 阶段，将 sPendingAnimations 中的动画取出，并且根据当前时间和动画的执行时间对比，将一些需要延迟的动画放到 delay 列表中，将不需要延迟的动画立即执行。动画开始执行后，并没有 break 出来，而是直接到 ANIMATION_FRAME 阶段，在这个阶段，会遍历已经 ready 的动画列表和 delay 的动画列表，并且根据当前时间与动画的执行时间进行比对，如果到了动画的执行时间，那么执行动画，然后才是遍历 sAnimations 中的动画，其中的动画就是立即可以执行的动画，最后，如果动画列表中的动画没有被执行完成，则会发送一个 ACTION_FRAME 消息，使得它又进入到 ANIMATION_FRAME 这个处理流程，然后又是重复上述的动画执行过程，直到动画全部执行完毕。sAnimations 等列表的声明如下：

```
// 所有可以立即执行的动画
private static final ThreadLocal<ArrayList<ValueAnimator>> sAnimations =
        new ThreadLocal<ArrayList<ValueAnimator>>() {
            @Override
            protected ArrayList<ValueAnimator> initialValue() {
                return new ArrayList<ValueAnimator>();
            }
        };

// The per-thread set of animations to be started on the next animation
// frame    所有未被处理的动画
private static final ThreadLocal<ArrayList<ValueAnimator>> sPendingAnimations =
        new ThreadLocal<ArrayList<ValueAnimator>>() {
            @Override
            protected ArrayList<ValueAnimator> initialValue() {
                return new ArrayList<ValueAnimator>();
            }
        };
```

那么，我们的动画是什么时候添加到这个 sAnimations 列表中的呢？

从程序中看到，在 ANIMATION_START 中，对于没有延时的动画，调用了一个 anim.startAnimation()方法，我们看看它做了些什么操作：

```
private void startAnimation() {
    initAnimation();// 1. 初始化动画
    sAnimations.get().add(this);// 2. 将动画添加到 sAnimation 中
    if (mStartDelay > 0 && mListeners != null) {
        ArrayList<AnimatorListener> tmpListeners =
                (ArrayList<AnimatorListener>) mListeners.clone();
        int numListeners = tmpListeners.size();
        for (int i = 0; i < numListeners; ++i) {
            tmpListeners.get(i).onAnimationStart(this);//3. 回调动画开始的 hook
        }
    }
}
```

从程序中可以看到，进行动画初始化之后，在 startAnimation 中确实将该动画添加到 sAnimations

中了，并且执行了相应的回调方法。

我们看看 ObjectAnimator 中的 initAnimation 函数到底做了些什么：

```
@Override
void initAnimation() {
    if (!mInitialized) {
    // 注意这里，很重要
        if ((mProperty == null) && AnimatorProxy.NEEDS_PROXY && (mTarget
        instanceof View) && PROXY_PROPERTIES.containsKey(mPropertyName)) {
            setProperty(PROXY_PROPERTIES.get(mPropertyName));
        }
        int numValues = mValues.length;
        for (int i = 0; i < numValues; ++i) {
            mValues[i].setupSetterAndGetter(mTarget);
        }
        super.initAnimation();
    }
}
```

我们看到 initAnimation 中有个很长的 if 判断语句，这个判断语句就是对系统版本进行判断的地方，如果系统版本小于 API 11，那么，AnimatorProxy.NEEDS_PROXY 为真。我们看看它的声明：

```
public static final boolean NEEDS_PROXY = Integer.valueOf(Build.VERSION.SDK).intValue
() < Build.VERSION_CODES.HONEYCOMB;
```

可以看到当 Android SDK 的版本号小于 HONEYCOMB，NEEDS_PROXY 就为真，HONEYCOMB 就是系统 3.0 的代号，即 API 11 的代号。如果 NEEDS_PROXY 为真，且属性是合法的属性，那么就会进入到 if 语句中，即会调用 setProperty(PROXY_PROPERTIES.get(mPropertyName))。在这个例子中，我们的属性是 scaleX，可以看到，PROXY_PROPERTIES 会 get 一个 key 为 mPropertyName 的值，这个 mPropertyName 就是我们传递进来的 scaleX 属性。我们看看 PROXY_PROPERTIES 是什么：

```
private static final Map<String, Property> PROXY_PROPERTIES = new HashMap<String,
Property>();

    static {
        PROXY_PROPERTIES.put("alpha", PreHoneycombCompat.ALPHA);
        PROXY_PROPERTIES.put("pivotX", PreHoneycombCompat.PIVOT_X);
        PROXY_PROPERTIES.put("pivotY", PreHoneycombCompat.PIVOT_Y);
        PROXY_PROPERTIES.put("translationX", PreHoneycombCompat.TRANSLATION_X);
        PROXY_PROPERTIES.put("translationY", PreHoneycombCompat.TRANSLATION_Y);
        PROXY_PROPERTIES.put("rotation", PreHoneycombCompat.ROTATION);
        PROXY_PROPERTIES.put("rotationX", PreHoneycombCompat.ROTATION_X);
        PROXY_PROPERTIES.put("rotationY", PreHoneycombCompat.ROTATION_Y);
        PROXY_PROPERTIES.put("scaleX", PreHoneycombCompat.SCALE_X);
        PROXY_PROPERTIES.put("scaleY", PreHoneycombCompat.SCALE_Y);
        PROXY_PROPERTIES.put("scrollX", PreHoneycombCompat.SCROLL_X);
        PROXY_PROPERTIES.put("scrollY", PreHoneycombCompat.SCROLL_Y);
        PROXY_PROPERTIES.put("x", PreHoneycombCompat.X);
        PROXY_PROPERTIES.put("y", PreHoneycombCompat.Y);
    }
```

PROXY_PROPERTIES 是一个 HashMap，我们传进来的是 scaleX，那么我们 get 得到的就应该是 PreHoneycombCompat.SCALE_X，继续看它具体是什么：

```
static Property<View, Float> SCALE_X = new FloatProperty<View>("scaleX") {
```

```
        @Override
        public void setValue(View object, float value) {
            AnimatorProxy.wrap(object).setScaleX(value);
        }

        @Override
        public Float get(View object) {
            return AnimatorProxy.wrap(object).getScaleX();
        }
    };
```

这原来就是包装了一下 View，并且在 setValue 时通过它的包装对它调用 setScaleX 函数。那么 AnimatorProxy 又是什么呢？其实它就是低于 API 11 时对 View 进行包装，从而通过矩阵来达到 View 的缩放、平移、翻转等效果的动画代理类。看看下面的部分代码：

```
public final class AnimatorProxy extends Animation {
    private AnimatorProxy(View view) {
        setDuration(0); // perform transformation immediately
        setFillAfter(true); // persist transformation beyond duration
        view.setAnimation(this);
        mView = new WeakReference<View>(view);
    }

    public void setScaleX(float scaleX) {
        if (mScaleX != scaleX) {
            prepareForUpdate();
            mScaleX = scaleX;
            invalidateAfterUpdate();
        }
    }

    private void prepareForUpdate() {
        View view = mView.get();
        if (view != null) {
            computeRect(mBefore, view);
        }
    }

    private void invalidateAfterUpdate() {
        View view = mView.get();
        final RectF after = mAfter;
        computeRect(after, view);
        after.union(mBefore);
                // 刷新 View 所在的区域
         ((View) view.getParent()).invalidate(
                (int) Math.floor(after.left),
                (int) Math.floor(after.top),
                (int) Math.ceil(after.right),
                (int) Math.ceil(after.bottom));
    }

    private void computeRect(final RectF r, View view) {
        final float w = view.getWidth();
        final float h = view.getHeight();
        r.set(0, 0, w, h);
        final Matrix m = mTempMatrix;
        m.reset();
        // 进行矩阵变化
        transformMatrix(m, view);
        mTempMatrix.mapRect(r);
```

```
        r.offset(view.getLeft(), view.getTop());
                // 代码省略
    }
        // 代码省略

    @Override
    protected void applyTransformation(float interpolatedTime, Transformation t) {
        View view = mView.get();
        if (view != null) {
                        t.setAlpha(mAlpha);
                        // 应用矩阵到 View 中
                        transformMatrix(t.getMatrix(), view);
        }
    }
}
```

很明显，就是通过矩阵变换达到动画的效果，关于矩阵的使用和原理请参考其他资料，这里不再赘述。

在低于 API 11 的版本中，当目标属性是 scaleX 时会调用 AnimationProxy 的 setScaleX 函数实现缩放动画效果，这个 Proxy 的动画的 duration 为 0，也就是动画会立即完成。每次需要更新属性时，都会调用 FloatProperty 的 setValue 方法，而 setValue 又调用 AnimationProxy 的 setScaleX()方法，setScaleX 在低于 API 11 时通过矩阵变换来完成缩放效果。在动画执行周期内不断地执行这个 setScaleX 函数就达到了动画效果。

当大于等于 API 11 时它又是怎样的呢？

我们看到，在 initAnimation 中，还有一个地方需要注意：

```
void initAnimation() {
    if (!mInitialized) {
        if ((mProperty == null) && AnimatorProxy.NEEDS_PROXY && (mTarget instanceof
        View) && PROXY_PROPERTIES.containsKey(mPropertyName)) {
            setProperty(PROXY_PROPERTIES.get(mPropertyName));
        }
        int numValues = mValues.length;
        for (int i = 0; i < numValues; ++i) {
                        // 1. 设置 setter 和 getter 方法
            mValues[i].setupSetterAndGetter(mTarget);
        }
        super.initAnimation();
    }
}
```

看一下注释 1 处的程序，这里会初始化属性的 setter 和 getter 方法。通过反射调用这些函数就可以修改其属性值了。

那么，在什么时候会调用相应的方法来更新属性值呢？更新属性值，我们想到的应该是在 ANIMATION_FRAME 中进行，那里就是循环调用动画，直到动画结束，因此，那应该是更新属性的地方。我们注意到一个地方，具体程序如下：

```
int numAnims = animations.size();
int i = 0;
while (i < numAnims) {
    ValueAnimatoranim = animations.get(i);
    if(anim.animationFrame(currentTime)){
    //这里有一个使用时间的判断，就是获取当前时间的属性值的地方
```

```
endingAnims.add(anim);
        }
    }
```

这里有一个关键的函数，即 animationFrame 是个关键函数。它是根据当前时间计算动画在当前时刻的属性值，具体代码如下：

```
booleananimationFrame(long currentTime) {
booleandone = false;
        // 代码省略
switch (mPlayingState) {
caseRUNNING:
caseSEEKED:
floatfraction = mDuration> 0
? (float) (currentTime - mStartTime) / mDuration : 1f;
// 代码省略

animateValue(fraction);// 计算当前动画的属性值
break;
        }

return done;
    }
```

通过时间百分比计算得到了 fraction 之后调用了 animateValue 函数，我们看看 ObjectAnimator 中的 animateValue 函数实现：

```
@Override
void animateValue(float fraction) {
    super.animateValue(fraction);// 计算属性值
    int numValues = mValues.length;
    for (int i = 0; i < numValues; ++i) {
        mValues[i].setAnimatedValue(mTarget);// 更新属性值
    }
}
```

在本实例中，PropertyValuesHolder 为 FloatPropertyValuesHolder 类型，因此，mValues 也就是 FloatPropertyValuesHolder 类型，继续深入 FloatPropertyValuesHolder 中的 setAnimatedValue 方法：

```
@Override
void setAnimatedValue(Object target) {
// 1. 如果有 float property，则通过 setValue 来更新属性值
    if (mFloatProperty != null) {
        mFloatProperty.setValue(target, mFloatAnimatedValue);
        return;
    }
   if (mProperty != null) {// 2. 如果有属性,则通过 setValue 来更新
        mProperty.set(target, mFloatAnimatedValue);
        return;
    }
    if (mSetter != null) {// 3. 否则通过反射调用属性的 setter 函数来更新属性值
        try {
            mTmpValueArray[0] = mFloatAnimatedValue;
            // 通过 setter 函数更新属性值
            mSetter.invoke(target, mTmpValueArray);
        }
        // catch 省略
    }
}
```

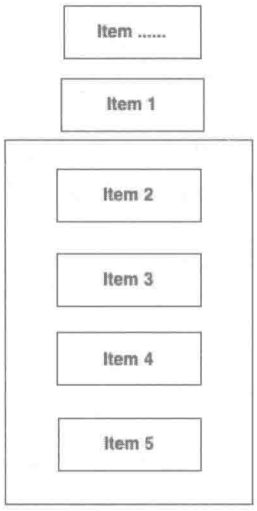
可以看到，这种形式的更新方式主要是通过 setValue 和反射。

可以看到，小于 API 11 时会设置 Property，对于本例中的 scaleX，这个 Property 为 FloatProperty，因此，会在 setAnimationValue 中执行注释 1 的代码，最后使用矩阵变换的形式变换 View；如果大于等于 API 11 则会调用目标对象的 setter 方法来更新属性值，例如 scaleX 对应的 setter 方法为 setScaleX。NineOldAnimations 通过这两种形式来实现属性动画在低版本系统的兼容，保证了 UI 效果的一致性。

7.8 策略模式实战应用

经过小民的几次重构，他的 ImageLoader 逐渐稳定起来，在灵活性、可定制化上都有了一定的成绩。小民的编程能力也有了很大的提升，在工作上越来越得心应手。这不，在完成了主管分配的工作之后，小民悠哉地坐在办公椅上，喝了口茶，仰望着天空，顿时觉得今天的阳光特别明媚，幽幽地说了一句："这才是生活嘛"。

休息了半晌，小民回到电脑傍，发现有 ImageLoader 的用户发来了邮件，大致的意思是让小民的 ImageLoader 不仅能够实现顺序加载图片，也能够逆序加载。什么意思呢？在我们加载图片时，加载请求会被封装成一个 Request 对象添加到请求队列中，ImageLoader 会为每个请求分配一个序列号，越晚加入的请求序列号值越大，默认情况下 ImageLoader 会按照先后顺序加载图片。但是现实中，我们可能需要最后添加到队列的请求先被执行，例如，我们在滚动 ListView 时，在最后一项肯定是最晚被加载的，此时它里面的图片加载请求也就是最晚被添加到请求队列。如果上面有 100 个加载请求，就需要等到这前 100 个请求加载完之后才会加载当前显示在我们面前的最好的一项，这必然是不符合情理的。如图 7-10 所示，Item 5 是当前显示在手机屏幕上的，但是是最后一个添加到请求列表的，Item 1 以上的已经变得不可见，它是早于 Item 5 加入到请求列表。



▲图 7-10

此时，我们的需求是在屏幕上显示的 Item View 的图片优先被加载，因此，我们就需要 ImageLoader 支持从请求队列的尾部开始加载。

"策略模式！"小民看完用户需求之后脱口而出。在经历了上几次的失败之后，小民痛下决心学习设计模式，并且加以实践，对于一些常用的模式已是熟记在心。既然已经知道了解决方案，那我们直接看看小民的实现吧：

```
/**
 * 加载策略接口
*/
public interface LoadPolicy {
    public int compare(BitmapRequest request1, BitmapRequest request2);
}
/**
 * 顺序加载策略
*/
```

```
public class SerialPolicy implements LoadPolicy {

    @Override
    public int compare(BitmapRequest request1, BitmapRequest request2) {
        // 按照添加到队列的序列号顺序来执行
        return request1.serialNum - request2.serialNum;
    }

}

/**
 * 逆序加载策略,即从最后加入队列的请求进行加载
*/
public class ReversePolicy implements LoadPolicy {

    @Override
    public int compare(BitmapRequest request1, BitmapRequest request2) {
        // 注意, Bitmap 请求要先执行最晚加入队列的请求,ImageLoader 的策略
        return request2.serialNum - request1.serialNum;
    }
}
```

首先定义了一个 LoadPolicy 接口，在这个接口中有一个 compare 方法，用来对比两个请求。小民默认实现了顺序加载、逆序加载两个策略，因为每个请求都有一个序列号，这个序列号以递增的形式增长，越晚加入队列的请求的序列号越大，而我们的请求列队是优先级队列，因此，我们需要在图片加载请求类中实现 Comparable 接口，以此实现对这些请求的排序处理。我们看看这部分代码：

```
public class BitmapRequest implements Comparable<BitmapRequest> {

    /**
     * 加载策略
     */
    LoadPolicy mLoadPolicy = new SerialPolicy();


    @Override
    public int compareTo(BitmapRequest another) {
        // 委托给 LoadPolicy 进行处理，实现按照策略模式
        return mLoadPolicy.compare(this, another);
    }
}
```

就这么简单，用户在配置 ImageLoader 时可以设置加载策略，这个策略会被设置给每个图片加载请求对象，具体代码如下：

```
public void displayImage(final ImageView imageView, final String uri,
        final DisplayConfig config, final ImageListener listener) {
    BitmapRequest request = new BitmapRequest(imageView, uri, config, listener);
    // 加载的配置对象,如果没有设置则使用 ImageLoader 的配置
    request.displayConfig = request.displayConfig != null ? request.displayConfig
            : mConfig.displayConfig;
    // 设置加载策略
    request.setLoadPolicy(mConfig.loadPolicy);
    // 添加到队列中
    mImageQueue.addRequest(request);
}
```

图片加载请求对象的 compare 方法通过转发给 mLoadPolicy 处理来实现这些请求在请求队列中

的排序，通过修改排序策略来实现修改图片加载顺序的效果。不同的策略有不同的排序方式，策略的可替换性又一次保证了 ImageLoader 的可扩展性。

7.9 总结

策略模式主要用来分离算法，在相同的行为抽象下有不同的具体实现策略。这个模式很好地演示了开闭原则，也就是定义抽象，注入不同的实现，从而达到很好的可扩展性。

优点

- 结构清晰明了、使用简单直观；
- 耦合度相对而言较低，扩展方便；
- 操作封装也更为彻底，数据更为安全。

缺点

- 随着策略的增加，子类也会变得繁多。

第 8 章　随遇而安——状态模式

8.1 状态模式介绍

状态模式中的行为是由状态来决定的，不同的状态下有不同的行为。状态模式和策略模式的结构几乎完全一样，但它们的目的、本质却完全不一样。状态模式的行为是平行的、不可替换的，策略模式的行为是彼此独立、可相互替换的。用一句话来表述，状态模式把对象的行为包装在不同的状态对象里，每一个状态对象都有一个共同的抽象状态基类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。

8.2 状态模式的定义

当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

8.3 状态模式的使用场景

（1）一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为。

（2）代码中包含大量与对象状态有关的条件语句，例如，一个操作中含有庞大的多分支语句（if-else 或 switch-case），且这些分支依赖于该对象的状态。

状态模式将每一个条件分支放入一个独立的类中，这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化，这样通过多态来去除过多的、重复的 if-else 等分支语句。

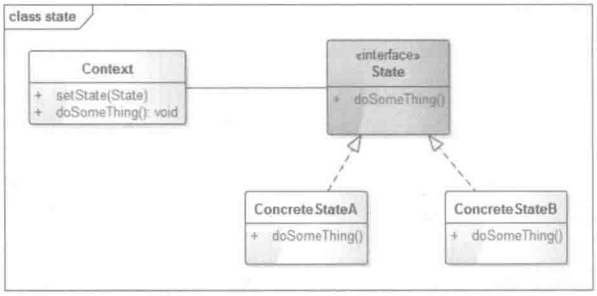
8.4 状态模式的 UML 类图

UML 类图如图 8-1 所示。

角色介绍，如图 8-1 所示。

- Context：环境类，定义客户感兴趣的接口，维护一个 State 子类的实例，这个实例定义了对象的当前状态。

- State：抽象状态类或者状态接口，定义一个或者一组接口，表示该状态下的行为。
- ConcreteStateA、ConcreteStateB：具体状态类，每一个具体的状态类实现抽象 State 中定义的接口，从而达到不同状态下的不同行为。



▲图 8-1

8.5 状态模式的简单示例

下面我们就以电视遥控器为例来演示一下状态模式的实现。我们首先将电视的状态简单分为开机状态和关机状态，在开机状态下可以通过遥控器进行频道切换、调整音量等操作，但是，此时重复按开机键是无效的；而在关机状态下，频道切换、调整音量、关机都是无效的操作，只有按开机按钮时会生效。也就是说电视的内部状态决定了遥控器的行为，我们看看第一版实现：

```
/**
 * 电视遥控器，含有开机、关机、下一频道、上一频道、调高音量、调低音量这几个功能
 */
public class TvController {
    // 开机状态
    private final static int POWER_ON = 1;
    // 关机状态
    private final static int POWER_OFF = 2;
    private int mState = POWER_OFF;

    public void powerOn() {
        mState = POWER_ON;
        if (mState == POWER_OFF) {
            System.out.println("开机啦");
        }
    }

    public void powerOff() {
        mState = POWER_OFF;
        if (mState == POWER_ON) {
            System.out.println("关机啦");
        }
    }

    public void nextChannel() {
        if (mState == POWER_ON) {
            System.out.println("下一频道");
        } else {
            System.out.println("两个红灯提示没有开机");
        }
```

```
    }

    public void prevChannel() {
        if (mState == POWER_ON) {
            System.out.println("上一频道");
        } else {
            System.out.println("两个红灯提示没有开机");
        }
    }

    public void turnUp() {
        if (mState == POWER_ON) {
            System.out.println("调高音量");
        } else {
            System.out.println("两个红灯提示没有开机");
        }
    }

    public void turnDown() {
        if (mState == POWER_ON) {
            System.out.println("调低音量");
        } else {
            System.out.println("两个红灯提示没有开机");
        }
    }
}
```

可以看到，在 TvController 类中，通过 mState 字段存储了电视的状态，并且在各个操作中根据状态来判断是否应该执行。这就导致了在每个功能中都需要使用 if-else，代码重复、相对较为混乱，这是在只有两个状态和简单几个功能函数的情况下，那么当状态变成 5 个、功能函数变为 10 个呢？每个函数中都要用 if-else 进行判断，而这些代码都充斥在一个类中，这些重复的代码无法被提取出来，这使得这个类变得越来越难以维护。

状态模式就是为解决这类的问题而出现的，我们将这些状态用对象来代替，将这些行为封装到对象中，使得在不同的状态下有不同的实现，这样就将这些 if-else 从 TvController 类中去掉，整个结构也变得清晰起来。我们看看实现代码：

```
    // 电视状态接口,定义了电视操作的函数
public interface TvState {
    public void nextChannel();
    public void prevChannel();
    public void turnUp();
    public void turnDown();
}


/**
 * 关机状态，此时只有开机功能是有效的
*/
public class PowerOffState implements TvState {

    @Override
    public void nextChannel() {
    }

    @Override
    public void prevChannel() {
    }
```

```
    @Override
    public void turnUp() {
    }

    @Override
    public void turnDown() {
    }
}


/**
 * 开机状态,此时再触发开机功能不做任何操作
 */
public class PowerOnState implements TvState {

    @Override
    public void nextChannel() {
        System.out.println("下一频道");
    }

    @Override
    public void prevChannel() {
        System.out.println("上一频道");
    }

    @Override
    public void turnUp() {
        System.out.println("调高音量");
    }

    @Override
    public void turnDown() {
        System.out.println("调低音量");
    }

}

/**
 * 电源操作接口
 */
public interface PowerController {
    public void powerOn();

    public void powerOff();
}

/**
 * 电视遥控器，类似于经典状态模式中的 Context
 */
public class TvController implementsPowerController {
    TvState mTvState;

    public void setTvState(TvState mTvState) {
        this.mTvState = mTvState;
}

@Override ide
    public void powerOn() {
        setTvState(new PowerOnState());
        System.out.println("开机啦");
    }
```

```
    @Override
    public void powerOff() {
        setTvState(new PowerOffState());
        System.out.println("关机啦");
    }

    public void nextChannel() {
        mTvState.nextChannel();
    }

    public void prevChannel() {
        mTvState.prevChannel();
    }

    public void turnUp() {
        mTvState.turnUp();
    }

    public void turnDown() {
        mTvState.turnDown();
    }
}
```

下面是客户端的调用代码。

```
public class Client {
    public static void main(String[] args) {
        TvController tvController = new TvController();
        // 设置开机状态
        tvController.powerOn();
        // 下一频道
        tvController.nextChannel();
        // 调高音量
        tvController.turnUp();
        // 设置关机状态
        tvController.powerOff();
        // 调高音量，此时不会生效
        tvController.turnUp();
    }
}
```

输出结果如下：

```
开机啦;
下一频道;
调高音量;
关机啦。
```

上述实现中，我们抽象了一个 TvState 接口，该接口中有操作电视的所有函数，该接口有两个实现类，即开机状态（PowerOnState）和关机状态 （PowerOffState）。开机状态下只有开机功能是无效的，也就是说在已经开机的时候用户再按开机键不会产生任何反应；而在关机状态下，只有开机功能是可用的，其他功能都不会生效。同一个操作，如调高音量的 turnUp 函数，在关机状态下无效，在开机状态下就会将电视的音量调高，也就是说电视的内部状态影响了电视遥控器的行为。状态模式将这些行为封装到状态类中，在进行操作时将这些功能转发给状态对象，不同的状态有不同的实现，这样就通过多态的形式去除了重复、杂乱的 if-else 语句，这也正是状态模式的精髓所在。

8.6 Wi-Fi 管理中的状态模式

“现在很多人，到什么地方去就先问‘有没有 Wi-Fi’，在这里我们并不是要吐槽流量费的问题，而是从一个侧面体现出 Wi-Fi 在我们手机使用过程中的重要性。对于工程师来说，我们更关心的是在 Wi-Fi 开、关按钮的背后做了什么。

如图 8-2 所示就是我们设置 Wi-Fi 的界面。

▲图 8-2

一个开关按钮，当 Wi-Fi 关闭时列表上为空；当 Wi-Fi 开启之后首先扫描周边的 Wi-Fi，然后展示到 ListView 上。这个设置界面是一个叫做 WifiSettings 的 Fragment 实现的，它的部分代码如下：

```
// Wi-Fi 设置界面
public class WifiSettings extends RestrictedSettingsFragment
        implements DialogInterface.OnClickListener {
    // Wi-Fi 扫描器
    private final Scanner mScanner;
    // Wi-Fi 管理器
    private WifiManager mWifiManager;
    // Wi-Fi 开关控制器
    private WifiEnabler mWifiEnabler;

   public WifiSettings() {
        super(DISALLOW_CONFIG_WIFI);
        // 初始化 intentfilter
        // 监听 Wi-Fi 状态相关的广播
        mReceiver = new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                handleEvent(context, intent);
            }
        };
        mScanner = new Scanner();
}

    @Override
    public void onResume() {
        super.onResume();
        if (mWifiEnabler != null) {
            mWifiEnabler.resume();
        }

// 注册广播接收器
getActivity().registerReceiver(mReceiver, mFilter);
        updateAccessPoints();
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        // 1. 获取 WifiManager
        mWifiManager = (WifiManager) getSystemService(Context.WIFI_SERVICE);
        // 代码省略
```

```
        // 2. 初始化显示 Wi-Fi 热点的 ListView
mEmptyView = initEmptyView();
    }

    protected TextView initEmptyView() {
        TextView emptyView = (TextView) getActivity().findViewById(android.R.id.empty);
        getListView().setEmptyView(emptyView);
        return emptyView;
    }

    @Override
    public void onStart() {
        super.onStart();
        // 将 actionBarSwitch 与 WifiEnabler 关联起来
        mWifiEnabler = createWifiEnabler();
    }

    WifiEnabler createWifiEnabler() {
            final SettingsActivity activity = (SettingsActivity) getActivity();
            return new WifiEnabler(activity, activity.getSwitchBar());
        }
    // 代码省略
    }
```

WifiSettings 中这个 Fragment 作用主要是构建了一个设置界面、Wi-Fi 开关切换按钮、一个 Wi-Fi 热点显示列表，当 Wi-Fi 启动后开始扫描周边的 Wi-Fi，并且将扫描到的 Wi-Fi 显示到 ListView 中。在这里还有一个重要的类 WifiEnabler，就是一个 Wi-Fi 开关的控制类，在 onStart 函数中创建，它的构造函数是这样的：

```
mWifiEnabler  = new WifiEnabler(activity,activity.getSwitchBar()) ;
```

这个 SwitchBar 就是 Wi-Fi 开关控制按钮，与按钮建立关联到底是怎么回事？我们直接看看 WifiEnabler 程序：

```
    public class WifiEnabler implements CompoundButton.OnCheckedChangeListener  {
    private final Context mContext;
private SwitchBar mSwitchBar;

    private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            if (WifiManager.WIFI_STATE_CHANGED_ACTION.equals(action)) {
                handleWifiStateChanged(intent.getIntExtra(
                        WifiManager.EXTRA_WIFI_STATE, WifiManager.WIFI_STATE_UNKNOWN));
            }
            // 其他 if-else
        }
    };

    public WifiEnabler(Context context, SwitchBar switchBar) {
        mContext = context;
        mSwitchBar = switchBar;

        mWifiManager = (WifiManager) context.getSystemService(Context.WIFI_SERVICE);

        mIntentFilter = new IntentFilter(WifiManager.WIFI_STATE_CHANGED_ACTION);
        // The order matters! We really should not depend on this. :(
```

```
        mIntentFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
        mIntentFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);

        setupSwitchBar();
    }

    public void setupSwitchBar() {
        final int state = mWifiManager.getWifiState();
        handleWifiStateChanged(state);
        if (!mListeningToOnSwitchChange) {
            mSwitchBar.addOnSwitchChangeListener(this);
            mListeningToOnSwitchChange = true;
        }
        mSwitchBar.show();
    }

public void resume() {
        // 注册广播接收器
        mContext.registerReceiver(mReceiver, mIntentFilter);
        // 设置 mSwitchBar 的监听器
        mSwitchBar.setOnCheckedChangeListener(this);
    }
}
```

WifiEnabler 类创建了一个广播来监听 WiFi 状态的改变，并且自身实现了 CompoundButton.OnCheckedChangeListener 接口，这个接口会监听 WiFi 开关按钮的状态修改。当 Wi-Fi 状态需要改变时首先会被广播接收器 mReceiver 接收到，此时会通过 handleWifiStateChanged 函数修改 WiFi 开关按钮的状态，该函数的代码如下：

```
private void handleWifiStateChanged(int state) {
    switch (state) {
        case WifiManager.WIFI_STATE_ENABLING:  // WiFi 开启中
            mSwitchBar.setEnabled(false);
            break;
        case WifiManager.WIFI_STATE_ENABLED:  // WiFi 已开启
            setSwitchChecked(true);
            mSwitchBar.setEnabled(true);
            break;
        case WifiManager.WIFI_STATE_DISABLING:  // WiFi 关闭中
            mSwitchBar.setEnabled(false);
            break;
        case WifiManager.WIFI_STATE_DISABLED:   // WiFi 已关闭
            setSwitchChecked(false);
            mSwitchBar.setEnabled(true);
            break;
        default:
            setSwitchChecked(false);
            mSwitchBar.setEnabled(true);
            break;
    }
}
```

Wi-Fi 状态修改之后会导致 mSwitchBar 开关按钮的状态的修改，而 mSwitchBar 的状态修改又会触发 onCheckedChanged 回调函数，在这个函数中又会调用 mWifiManager.setWifiEnabled(isChecked)的状态，具体代码如下：

```
public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
mSwitchBar.setEnabled(false);
```

```
        // 调用 mWifiManager.setWifiEnabled(isChecked) 修改 WiFi 的状态
        if (!mWifiManager.setWifiEnabled(isChecked)) {
            // Error
            mSwitchBar.setEnabled(true);
        }
}
```

在 onCheckedChanged 函数中又会调用 mWifiManager 来修改 WiFi 的状态，mWifiManager 的类型为 WifiManager，我们一起看看这个类的 setWifiEnabled 函数：

```
public boolean setWifiEnabled(boolean enabled) {
        try {
            // 实际上调用的是 WifiService 的 setWifiEnabled
            return mService.setWifiEnabled(enabled);
        } catch (RemoteException e) {
            return false;
        }
    }
```

WifiManager 的 setWifiEnabled 函数也是一个空壳，它实际上转发给了 WifiService 类 setWifiEnabled 方法，这个 WifiService 和 ActivityManagerService、WindowManagerService 一起在系统启动时被注入到 ServiceManager 中，相关代码如下：

```
class ServerThread {
    private static final String TAG = "SystemServer";

    public void initAndLoop() {
        // 代码省略
        WifiService wifi = null;
        try {
            // 启动 WiFi service
            wifi = new WifiService(context);
            ServiceManager.addService(Context.WIFI_SERVICE, wifi);
        } catch (Throwable e) {
            reportWtf("starting Wi-Fi Service", e);
        }
        // 代码省略
    }
}
```

还是继续看 WifiService 的 setWifiEnabled 函数程序：

```
public synchronized boolean setWifiEnabled(boolean enable) {
    // 代码省略
    mWifiController.sendMessage(CMD_WIFI_TOGGLED);
    return true;
}
```

这还不是真正实现功能的地方！实际工作是通过 mWifiController 发送了一个 CMD_WIFI_TOGGLED 消息，看起来和 Handler 的功能很像啊！mWifiController 的类型为 WifiController，sendMessage 这个方法在 WifiController 的父类 StateMachine 中，因此，我们继续进入 StateMachine 程序中：

```
// WiFi controller 的父类，StateMachine 类
public class StateMachine {
    // 发送消息
```

```
        public final void sendMessage(int what) {
        SmHandler smh = mSmHandler;
        if (smh == null) return;
        // 通过 SmHandler 对象发送消息
        smh.sendMessage(obtainMessage(what));
    }
}
```

StateMachine 中的 sendMessage 方法实际上是通过 StateMachine 的内部类 SmHandler 对象发送的消息，那么我们看看 SmHandler 的相关代码：

```
private static class SmHandler extends Handler {
        /** Reference to the StateMachine */
        private StateMachine mSm;

        // 状态信息类，用于管理状态层级
        private class StateInfo {
            /** 当前状态 */
            State state;
            /** 当前状态的上一个状态，即 parent state */
            StateInfo parentStateInfo;
            /** 当前状态是否被激活，如果已被激活则为 true */
            boolean active;
        }

        // 处理发送到 StateMachine 的消息
        @Override
        public final void handleMessage(Message msg) {
            if (!mHasQuit) {
                mMsg = msg;
                State msgProcessedState = null;
                if (mIsConstructionCompleted) {
                    // 1. 处理消息，并且返回处理了该消息的状态
                    msgProcessedState = processMsg(msg);
                } else if (!mIsConstructionCompleted && (mMsg.what == SM_INIT_CMD)
                && (mMsg.obj == mSmHandlerObj)) {
                        /** Initial one time path. */
                        mIsConstructionCompleted = true;
                        invokeEnterMethods(0);
                } else {
                }
                // 2. 执行状态转换
                performTransitions(msgProcessedState, msg);
            }
        }
    }
```

SmHandler 通过消息的机制类控制 WiFi 的状态，当 WiFi 状态发生改变时，外部通过 sendMessage 发送一个对应的消息，在 SmHandler 中的 handleMessage 首先调用 processMsg 将这个消息交给当前状态进行处理，如果当前状态不能处理，那么就会交给它的上一个状态处理，最后返回处理了这个消息的状态，具体代码如下：

```
    private final State processMsg(Message msg) {
            StateInfo curStateInfo = mStateStack[mStateStackTopIndex];
            // 是否退出消息
            if (isQuit(msg)) {
                transitionTo(mQuittingState);
} else {
```

```
    // 调用了状态对象的 processMessage 函数来处理消息
            while (!curStateInfo.state.processMessage(msg)) {
                /**
* 消息没有被处理，那么交给 parent state 处理，
* 这里类似于触摸事件的处理机制
                 */
                curStateInfo = curStateInfo.parentStateInfo;
                if (curStateInfo == null) {
                 // 到最后都没有被处理
                 // 则调用 mSm 的 unhandledMessage
                    mSm.unhandledMessage(msg);
                    break;
                }
            }
        }
        return (curStateInfo != null) ? curStateInfo.state : null;
    }
```

处理完该状态之后，会调用 performTransitions 函数退出旧的状态，并且进入到新状态中，具体代码如下：

```
// 进行状态转换
private void performTransitions(State msgProcessedState, Message msg) {
    // 原始状态
    State orgState = mStateStack[mStateStackTopIndex].state;
    // 目标状态
    State destState = mDestState;
    if (destState != null) {
        while (true) {
            // 退出某些状态，并且进入新的状态
            StateInfo commonStateInfo = setupTempStateStackWithStatesToEnter(destState);
            invokeExitMethods(commonStateInfo);
            int stateStackEnteringIndex = moveTempStateStackToStateStack();
            // 进入某个 WiFi 状态
            invokeEnterMethods(stateStackEnteringIndex);
           // 后续处理以及判断跳出循环，代码省略
        }
        mDestState = null;
    }
    // 代码省略
}

// 调用状态的 enter 函数
private final void invokeEnterMethods(int stateStackEnteringIndex) {
    for (int i = stateStackEnteringIndex; i <= mStateStackTopIndex; i++) {
        // 执行状态的 enter()函数
        mStateStack[i].state.enter();
        mStateStack[i].active = true;
    }
}
```

此时我们已经看到了状态模式的雏形。

现在让我们从 setWifiEnabled 函数的复杂调用链中回过神来。本章主题是状态模式，不是 WiFi 控制相关细节，我们关注的状态类就是 State，它代表了 WiFi 的某个状态，这个状态的定义如下：

```
public class State implements IState {
    protected State() {
    }
```

```
    /* 进入当前状态之后调用该函数
     */
    @Override
    public void enter() {
    }

    /* 退出该状态后调用该函数
     */
    @Override
    public void exit() {
    }

    //处理消息
    @Override
    public boolean processMessage(Message msg) {
        return false;
}
// 代码省略
}
```

状态之间并不是可以随意切换的，它们有一种层级关系，这些层级关系在 StateMachine 的构造函数中被定义，我们看看相关代码：

```
// WifiStateMachine
    public WifiStateMachine(Context context, String wlanInterface) {
        super("WifiStateMachine");

        addState(mDefaultState);
            addState(mInitialState, mDefaultState);
            addState(mSupplicantStartingState, mDefaultState);
            addState(mSupplicantStartedState, mDefaultState);
                addState(mDriverStartingState, mSupplicantStartedState);
                addState(mDriverStartedState, mSupplicantStartedState);
                    addState(mScanModeState, mDriverStartedState);
                    // 代码省略
                addState(mTetheredState, mSoftApStartedState);
                addState(mUntetheringState, mSoftApStartedState);
        // 初始化模式为 mInitialState
        setInitialState(mInitialState);

    }
}
```

在构造函数中调用了 addState 函数，这些函数最终都会调用 SmHandler 中的 addState 函数，我们看看这个函数的实现：

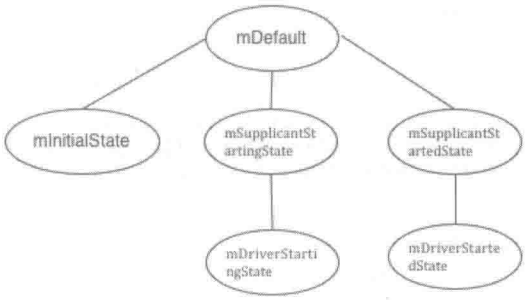
```
// 添加状态,state 为当前状态，parent 为上一个状态，也就是 parent 状态
private final StateInfo addState(State state, State parent) {

    StateInfo parentStateInfo = null;
    // 获取 parent 对应的 StateInfo
    if (parent != null) {
        parentStateInfo = mStateInfo.get(parent);
        if (parentStateInfo == null) {
            // Recursively add our parent as it's not been added yet.
            parentStateInfo = addState(parent, null);
        }
    }
    // 获取当前状态对应的 StateInfo
    StateInfo stateInfo = mStateInfo.get(state);
```

```
    if (stateInfo == null) {
        stateInfo = new StateInfo();
        mStateInfo.put(state, stateInfo);
    }

    // 建立层级关系
    stateInfo.state = state;
    // 上一个 StateInfo
    stateInfo.parentStateInfo = parentStateInfo;
    stateInfo.active = false;
    if (mDbg) mSm.log("addStateInternal: X stateInfo: " + stateInfo);
        return stateInfo;
}
```

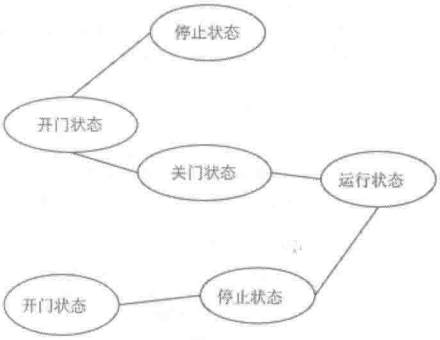
该函数就是在状态之间建立一个层级关系，这是一个树形的层级关系。例如，从 StateMachine 的构造函数中可以得到状态之间的部分层级图，如图 8-3 所示。

状态之间并不是跨越式转换的，当前状态只能转换到上一状态或者下一状态，例如，当 WiFi 的状态为 mDefault 时，它不能直接跳转到 mDriverStartedState 状态，需要先转换到 mSupplicantStartedState 状态，然后才能转换到 mDriverStartedState。StateMachine 内部就是维护了一个状态切换序列，并且不同的状态下对于相同的事件有不同的处理。

上述介绍比较抽象，我们用现实生活中的例子来说明。如我们给电梯指定了如下几个状态：停止、运行中、开门状态、关门状态，这几个状态之间肯定是不能够随意切换的，例如有人上了电梯，关门之后电梯开始往 10 楼运行，这个时候是运行状态，如果此时允许直接开门（也就是开门状态），那是不可以的。所以，在状态之间也是需要一个转换规则来控制的，这个转换规则就是我们这里的状态层级关系，例如，电梯里的状态关系可以转换为如图 8-4 所示。



▲图 8-3



▲图 8-4

不同状态对于不同的指令的反应也是完全不一样的，例如，在电梯运行中，用户按开门按钮电梯肯定不会开门，在开门状态下电梯也绝对不会直接运行。WiFi 工作状态机制也是同理，除了对状态之间的转换进行控制之外，还通过状态模式来对不同的命令进行不同的处理。上文我们说过，State 类就是状态类的基类，它与 WiFi 相关的子类都定义在 WifiStateMachine 中。

State 的类有 enter、exit、processMessage 三个函数，进入状态之后会调用 enter 函数，退出时会调用 exit 函数，处理具体的消息时则调用 processMessage。上文中讲到状态模式的核心是“当一个对象的内在状态改变时允许改变其行为”，processMessage 就是我们的关注点，不同的状态下就

是依赖这个函数实现不同的行为。这里我们以 WiFi 扫描这个功能为例来说明状态模式。

在请求扫描 WiFi 时，会发送一个 CMD_START_SCAN 消息，请求 WiFi 驱动进行 WiFi 扫描，但是，在初始（InitialState）状态下，WiFi 驱动还没有进行加载与启动，因此，在初始状态下该命令不会被处理，具体代码如下：

```
// 初始状态
class InitialState extends State {
        @Override
        public void enter() {
            // WiFi 驱动被卸载
            mWifiNative.unloadDriver();
        }

        @Override
        public boolean processMessage(Message message) {
            switch (message.what) {
                // 其他命令处理

                // 扫描 WiFi 的命令直接返回 NOT_HANDLED,也就是未处理
                default:
                    return NOT_HANDLED;
            }
            return HANDLED;
        }
        // 代码省略
    }
```

当处在 WiFi 驱动加载中状态（DriverStartingState）时，扫描 WiFi 的请求会被延后处理，当 WiFi 驱动加载完之后该请求被重新提交，DriverStartingState 类如下：

```
    // WiFi 驱动正在启动中
    class DriverStartingState extends State {
        private int mTries;
        @Override
        public void enter() {
            mTries = 1;
            /* Send ourselves a delayed message to start driver a second time */
            sendMessageDelayed(obtainMessage(CMD_DRIVER_START_TIMED_OUT,
            ++mDriverStartToken, 0), DRIVER_START_TIME_OUT_MSECS);
        }
        @Override
        public boolean processMessage(Message message) {
            switch(message.what) {
               // 代码省略

// 开始扫描 WiFi 的命令，以下这些命令在 DriverStartingState 下都是不处理
                case CMD_START_SCAN:
                // 其他命令
                case CMD_RECONNECT:
                    deferMessage(message); // 加入延迟处理消息列表
                    break;
                default:
                    return NOT_HANDLED;
            }
            return HANDLED;
        }
// 代码省略
    }
```

在 WiFi 驱动加载完之后进入 DriverStartedState 状态，扫描 WiFi 的命令会被处理，此时会请求 WiFi 驱动进行扫描工作，相关代码如下。

```
// WiFi 驱动已启动的状态
class DriverStartedState extends State {
    @Override
    public void enter() {
        // 其他工作
    }

    @Override
    public boolean processMessage(Message message) {
        switch(message.what) {
            // WiFi 驱动已启动的状态，扫描 WiFi 的命令会被执行
            // 调用了 WiFi 驱动开始扫描
            case CMD_START_SCAN:
                noteScanStart(message.arg1, (WorkSource) message.obj);
                startScanNative(WifiNative.SCAN_WITH_CONNECTION_SETUP);
                break;
            // 代码省略
            default:
                return NOT_HANDLED;
        }
        return HANDLED;
    }
    // 代码省略
}
```

从上述示例中我们看到，在不同的状态下对于扫描 WiFi 这个请求的处理是完全不一样的。在初始状态下扫描请求被直接忽略，在驱动加载中状态下 WiFi 扫描请求被添加到延迟处理的消息列表，在驱动加载完成状态下扫描 WiFi 的请求直接被处理。不同的状态下改变了扫描 WiFi 请求的行为，这就是状态模式的精髓所在。它的实现原理就是将请求的处理封装到状态类中，在不同的状态类中对同一个请求进行不同的处理。它能够消除一些重复的 if-else 逻辑，使得程序的结构更为清晰，可扩展性、稳定性也有一定的提高。

8.7 状态模式实战

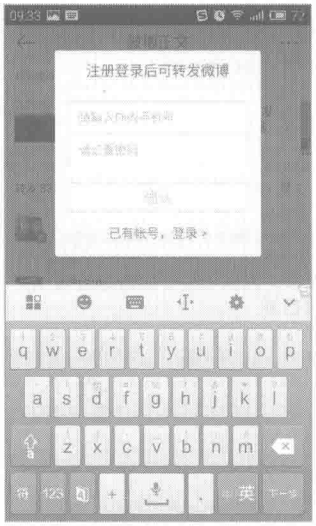
在开发过程中，我们用到状态模式最常见的地方应该是用户登录系统。在用户已登录和未登录的情况下，对于同一事件的处理行为是不一样的，例如，在新浪微博中，用户在未登录的情况下点击转发按钮，此时会先让用户登录，然后再执行转发操作，如图 8-5 和图 8-6 所示。如果是已登录的情况下，那么用户输入转发的内容后就可以直接进行操作。可见，在这两种状态下，对于转发这个操作的处理动画完全不一样，当状态改变时对于转发操作的行为发生了改变。

下面我们用状态模式来简单实现这个过程，首先创建一个 Android 项目，里面含有两个 Activity，分别为 MainActivity、LoginActivity，MainActivity 是应用第一个 Activity，有转发和注销用户功能，LoginActivity 则为用户登录界面。

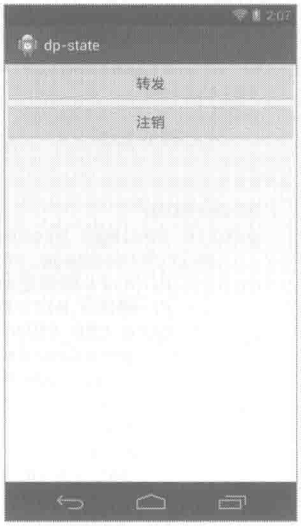
用户的默认状态为未登录状态，此时用户在 MainActivity 界面点击转发时会先跳转到登录界面，然后在登录界面登录成功后再回到 MainActivity 页面，此时，用户再进行转发操作就可以实现

真正的转发功能，如图 8-7、图 8-8、图 8-9 和图 8-10 所示。

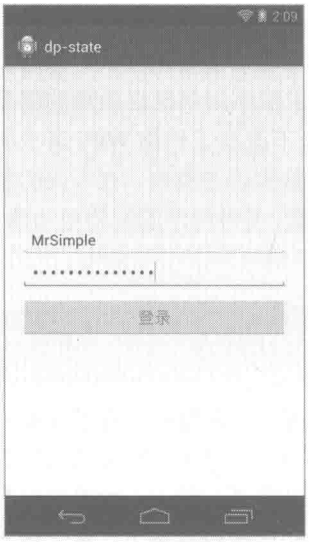
▲图 8-5



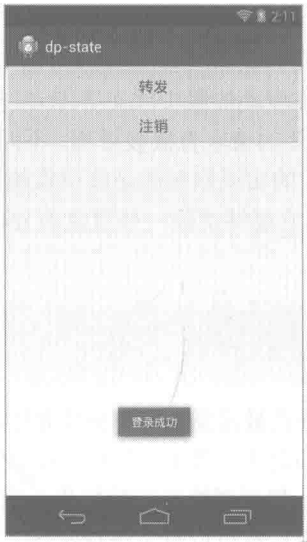
▲图 8-6



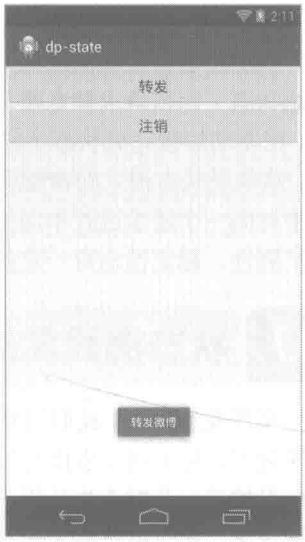
▲图 8-7　未登录状态下点转发



▲图 8-8　登录界面



▲图 8-9　登录成功，已登录状态



▲图 8-10　已登录状态下执行转发功能

MainActivity 的代码如下：

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        // 转发按钮
        findViewById(R.id.forward_btn).setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // 调用 LoginContext 的转发函数
            LoginContext.getLoginContext().forward(MainActivity.this);
            }
        });

        // 注销按钮
        findViewById(R.id.logout_btn).setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // 设置为注销状态
                LoginContext.getLoginContext().setState(new LogoutState());
            }
        });

    }
}
```

LoginActivity 则是在用户输入用户名和密码之后执行登录，成功之后将 LoginContext 的状态设置为已登录状态，并且返回 MainActivity 页面，具体代码如下：

```
public class LoginActivity extends Activity {

    EditText usrNameEditText;
    EditText pwdEditText;

    protected void onCreate(android.os.Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_login);

        usrNameEditText = (EditText) findViewById(R.id.username_edittext);
        pwdEditText = (EditText) findViewById(R.id.pwd_edittext);

        // 登录按钮
        findViewById(R.id.login_btn).setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                login();
                finish();
            }
        });
    }

    private void login() {
        String usrName = usrNameEditText.getText().toString().trim();
        String pwd = pwdEditText.getText().toString().trim();
        // 执行网络请求,进行登录 .....

        // 登录成功后修改为已登录状态
        LoginContext.getLoginContext().setState(new LoginedState());
        Toast.makeText(getApplicationContext(), "登录成功", Toast.LENGTH_LONG).show();
    }
}
```

这里的 LoginContext 就是状态模式中的 Context 角色，是用户的操作对象和状态管理对象，LoginContext 将相关操作委托给状态对象，这样在状态发生改变时，LoginContext 的行为就发生了改变。我们看看 LoginContext 的相关代码：

```
/**
 * LoginContext，用户接口和状态管理类
 */
public class LoginContext {
    // 用户状态，默认为未登录状态
    UserState mState = new LogoutState();
    // 单例
    static LoginContext sLoginContext = new LoginContext();

    privateLoginContext() {
    }

    public static LoginContext getLoginContext() {
        return sLoginContext;
    }

    public void setState(UserState aState) {
        this.mState = aState;
    }

    // 转发
    public void forward(Context context) {
        mState.forward(context);
    }

    public void comment(Context context) {
        mState.comment(context);
    }

}
```

LoginContext 通过 setState 来对状态进行修改，并且将操作委托给状态对象。不同的状态对象对于同一个操作进行不同的处理，状态对象相关的代码如下：

```
/**
 * 用户状态
 */
public interface UserState {
    /**
     * 转发
     */
    public void forward(Context context);

    /**
     * 评论
     */
    public void comment(Context context);
}

/**
 * 已登录状态
 */
public class LoginedState implements UserState {

    @Override
```

```
    public void forward(Context context) {
        Toast.makeText(context, "转发微博", Toast.LENGTH_LONG).show();
    }

    @Override
    public void comment(Context context) {
        Toast.makeText(context, "评论微博", Toast.LENGTH_LONG).show();
    }
}

/**
 * 注销状态，即未登录状态
 */
public class LogoutState implements UserState {

    @Override
    public void forward(Context context) {
        gotoLoginActivity(context);
    }

    @Override
    public void comment(Context context) {
        gotoLoginActivity(context);
    }

    private void gotoLoginActivity(Context context) {
        Intent intent = new Intent(context, LoginActivity.class);
        context.startActivity(intent);
    }

}
```

在 UserState 中有两个方法，即转发和评论。在已登录的状态下，用户调用这两个函数都会执行；但是在未登录的情况下则会跳转到登录页面，当用户登录成功之后才能进行这些操作。也就是说，在用户状态改变之后用户的行为也发生了改变。试想一下，如果没有使用状态模式，那么在任何调用转发或者评论功能的地方都需要进行是否登录的判断，或者再添加一个层次对这个判断逻辑进行统一的封装，如果已登录则执行相应的功能，否则跳转到登录界面，代码逻辑大致如下：

```
public void forward(Context context ) {
    if ( 用户已登录 ) {
        // 进行转发
    } else {
        // 跳转到登录界面
    }
}
```

但不管哪种形式，都会存在 if-else 的重复逻辑。如果再增加一个用户状态，那么每个功能函数中都需要添加对这个状态的判断。如果增加一个功能，也要在这个函数中对所有的状态进行判断，然后根据状态进行相应的功能。这使得这个模块很脆弱，一旦忘了某个用户状态进行判断，那么很可能引发用户权限问题。状态模式将这些行为封装到状态类中，不仅消除了重复的 if-else 逻辑、结构更为清晰，也使得这个模块的可扩展性、灵活性更高。

8.8 总结

状态模式的关键点在于不同的状态下对于同一行为有不同的响应，这其实就是一个将 if-else 用多态来实现的一个具体示例。在 if-else 或者 switch-case 形式下根据不同的状态进行判断，如果是状态 A 那么执行方法 A、状态 B 执行方法 B，但这种实现使得逻辑耦合在一起，易于出错，通过状态模式能够很好地消除这类“丑陋”的逻辑处理，当然并不是任何出现 if-else 的地方都应该通过状态模式重构，模式的运用一定要考虑所处的情景以及你要解决的问题，只有符合特定的场景才建议使用对应的模式。

状态模式的优点

State 模式将所有与一个特定的状态相关的行为都放入一个状态对象中，它提供了一个更好的方法来组织与特定状态相关的代码，将烦琐的状态判断转换成结构清晰的状态类族，在避免代码膨胀的同时也保证了可扩展性与可维护性。

状态模式的缺点

状态模式的使用必然会增加系统类和对象的个数。

第9章 使编程更有灵活性——责任链模式

9.1 责任链模式介绍

责任链模式（Iterator Pattern），是行为型设计模式之一。什么是“链”？我们将多个节点首尾相连所构成的模型称为链，比如生活中常见的锁链，就是由一个个圆角长方形的铁环串起来的结构。对于链式结构，每个节点都可以被拆开再连接，因此，链式结构也具有很好的灵活性。将这样一种结构应用于编程领域，将每一个节点看作是一个对象，每一个对象拥有不同的处理逻辑，将一个请求从链式的首端发出，沿着链的路径依次传递给每一个节点对象，直至有对象处理这个请求为止，我们将这样的一种模式称为责任链模式，这样的解释是不是更通俗易懂呢？我们还是看看责任链模式的标准定义。

9.2 责任链模式的定义

使多个对象都有机会处理请求，从而避免了请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，只到有对象处理它为止。

9.3 责任链模式的使用场景

多个对象可以处理同一请求，但具体由哪个对象处理则在运行时动态决定。

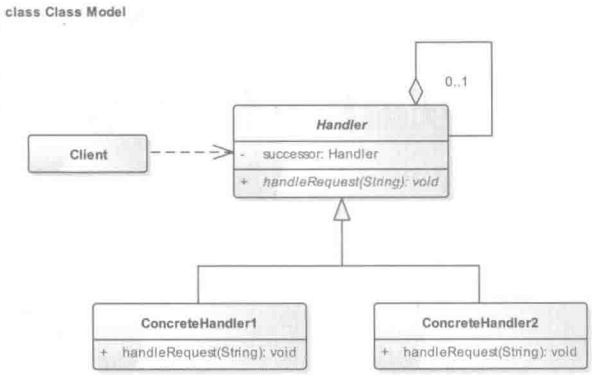
在请求处理者不明确的情况下向多个对象中的一个提交一个请求。

需要动态指定一组对象处理请求。

9.4 责任链模式的 UML 类图

责任链模式的 UML 类图如图 9-1 所示。

根据类图我们可以得出如下一个责任链模式简化版的通用模式代码。



▲图 9-1

程序清单 9-1　抽象处理者

```
public abstract class Handler {
    protected Handler successor;// 下一节点的处理者

    /**
     * 请求处理
     *
     * @param condition 请求条件
     */
    public abstract void handleRequest(String condition);
}
```

程序清单 9-2　具体的处理者 1

```
public class ConcreteHandler1 extends Handler {
    @Override
    public void handleRequest(String condition) {
        if (condition.equals("ConcreteHandler1")) {
            System.out.println("ConcreteHandler1 handled");
            return;
        } else {
            successor.handleRequest(condition);
        }
    }
}
```

程序清单 9-3　具体的处理者 2

```
public class ConcreteHandler2 extends Handler {
    @Override
    public void handleRequest(String condition) {
        if (condition.equals("ConcreteHandler2")) {
            System.out.println("ConcreteHandler2 handled");
            return;
        } else {
            successor.handleRequest(condition);
        }
    }
}
```

程序清单 9-4　客户类

```
public class Client {
    public static void main(String[] args) {
```

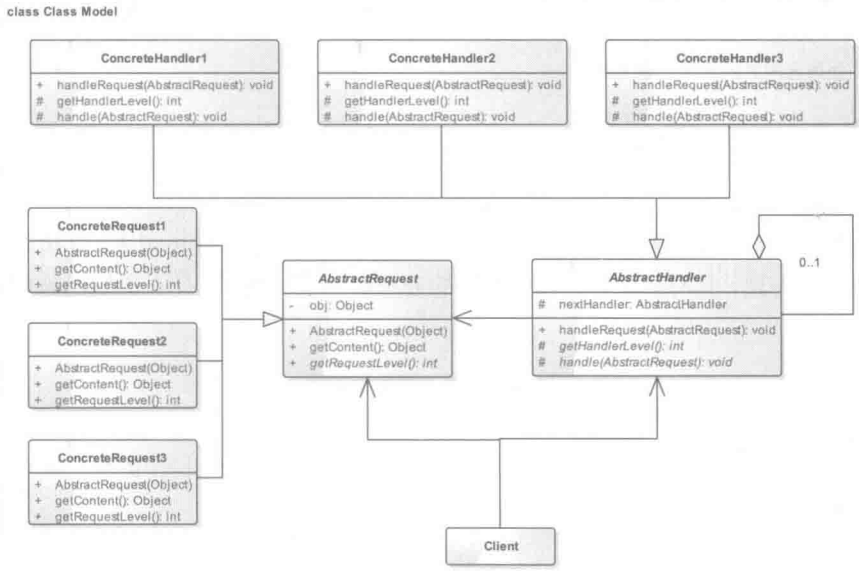
```
        // 构造一个 ConcreteHandler1 对象
        ConcreteHandler1 handler1 = new ConcreteHandler1();

        // 构造一个 ConcreteHandler2 对象
        ConcreteHandler2 handler2 = new ConcreteHandler2();

        // 设置 handler1 的下一个节点
        handler1.successor = handler2;

        // 设置 handler2 的下一个节点
        handler2.successor = handler1;

        // 处理请求
        handler1.handleRequest("ConcreteHandler2");
    }
}
```

角色介绍。

Handler：抽象处理者角色，声明一个请求处理的方法，并在其中保持一个对下一个处理节点 Handler 对象的引用。

ConcreteHandler：具体处理者角色，对请求进行处理，如果不能处理则将该请求转发给下一个节点上的处理对象。

上面我们说到这是一个简化版的通用模式代码，为什么这么说呢？因为对于请求来说，其形式是固定的，就是一个字符串，而判断一个节点上的对象是否能够处理该请求的标志，则是该字符串是否与之匹配。然而在大多数情况下，责任链中的请求和对应的处理规则是不尽相同的，在这种情况下可以将请求进行封装，同时对请求的处理规则也进行封装作为一个独立的对象，类图如图 9-2 所示。



▲图 9-2

首先我们来看 AbstractHandler 抽象处理者，其声明了处理者对象处理请求的方法和获取处理级别的方法，并对具体的处理转发逻辑进行了实现。

程序清单 9-5　抽象处理者

```
public abstract class AbstractHandler {
    protected AbstractHandler nextHandler;// 下一节点上的处理者对象

    /**
     * 处理请求
     *
     * @param request 请求对象
     */
    public final void handleRequest(AbstractRequest request) {
        // 判断当前处理者对象的处理级别是否与请求者的处理级别一致
        if (getHandleLevel() == request.getRequestLevel()) {
            // 一致则由该处理对象处理
            handle(request);
        } else {
            // 否则将该请求对象转发给下一个节点上的请求对象
            if (nextHandler != null) {
                nextHandler.handleRequest(request);
            } else {
                // 当所有处理者对象均不能处理该请求时输出
                System.out.println("All of handler can not handle the request");
            }
        }
    }

    /**
     * 获取处理者对象的处理级别
     *
     * @return 处理级别
     */
    protected abstract int getHandleLevel();

    /**
     * 每个处理者对象的具体处理方式
     *
     * @param request 请求者对象
     */
    protected abstract void handle(AbstractRequest request);
}
```

在这种情况下我们的责任转发逻辑由抽象处理类控制，而对于抽象请求者，其内部也声明了一个获取请求级别的方法，其与抽象处理者中返回的处理级别保持对应，什么级别的处理逻辑就对应什么样的请求级别。

程序清单 9-6　抽象请求者

```
public abstract class AbstractRequest {
    private Object obj;// 处理对象

    public AbstractRequest(Object obj) {
        this.obj = obj;
    }

    /**
     * 获取处理的内容对象
     *
```

```
     * @return 具体的内容对象
     */
    public Object getContent() {
        return obj;
    }

    /**
     * 获取请求级别
     *
     * @return 请求级别
     */
    public abstract int getRequestLevel();
}
```

其他的就不多介绍了，我们分别实现了 3 个请求者和 3 个处理者对象，逻辑很简单。

程序清单 9-7　请求者

```
public class Request1 extends AbstractRequest {
    public Request1(Object obj) {
        super(obj);
    }

    @Override
    public int getRequestLevel() {
        return 1;
    }
}

public class Request2 extends AbstractRequest {
    public Request2(Object obj) {
        super(obj);
    }

    @Override
    public int getRequestLevel() {
        return 2;
    }
}

public class Request3 extends AbstractRequest {
    public Request3(Object obj) {
        super(obj);
    }

    @Override
    public int getRequestLevel() {
        return 3;
    }
}
```

程序清单 9-8　处理者

```
public class Handler1 extends AbstractHandler {
    @Override
    protected int getHandleLevel() {
        return 1;
    }

    @Override
    protected void handle(AbstractRequest request) {
        System.out.println("Handler1 handle request:"
        + request.getRequestLevel());
```

```
    }
}

public class Handler2 extends AbstractHandler {
    @Override
    protected int getHandleLevel() {
        return 2;
    }

    @Override
    protected void handle(AbstractRequest request) {
        System.out.println("Handler2 handle request:"
        + request.getRequestLevel());
    }
}

public class Handler3 extends AbstractHandler {
    @Override
    protected int getHandleLevel() {
        return 3;
    }

    @Override
    protected void handle(AbstractRequest request) {
        System.out.println("Handler3 handle request:"
        + request.getRequestLevel());
    }
}
```

下面是客户类，具体的输出结果大家可以自行尝试，这里不再介绍。

程序清单 9-9　客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造 3 个处理者对象
        AbstractHandler handler1 = new Handler1();
        AbstractHandler handler2 = new Handler2();
        AbstractHandler handler3 = new Handler3();

        // 设置当前处理者对象下一个节点的处理者对象
        handler1.nextHandler = handler2;
        handler2.nextHandler = handler3;

        // 构造 3 个请求者对象
        AbstractRequest request1 = new Request1("Request1");
        AbstractRequest request2 = new Request2("Request2");
        AbstractRequest request3 = new Request3("Request3");

        // 总是从链式的首端发起请求
        handler1.handleRequest(request1);
        handler1.handleRequest(request2);
        handler1.handleRequest(request3);
    }
}
```

9.5 责任链模式的简单实现

小民某天接到通知说需要出差去×国进修学习新技术，小民听到这消息心中一喜，于是收拾完

背包后踏上了去×国进修的旅途。小去×国学习一趟花费了近5万元，于是小民上班后的第一天就向组长申请报销费用，组长一看是笔不小的数目，他没有权限审批，于是组长就拿着票据去找部门主管，主管一看要报这么多钱，自己权限内只能批五千以下的费用，这完全超出了自己的权限范围，于是主管又跑去找经理，经理一看二话不说直接拿着票据奔向了老板的办公室，因为他也只能批一万以下的费用。类似的情况对上班族来说肯定并不少见，上面的这个情景其实就是一个责任链的小例子，每一个人，准确地说是每一类人代表这条链上的一个节点，小民是请求的发起者，而老板则是处于链条顶端的类，小民从链的底端开始发出一个申请报账的请求，首先由组长处理该请求，组长比对后发现自己权限不够于是将该请求转发给位于链中下一个节点的主管，主管比对后也发现自己权限不够又将该请求转发给经理，而经理也基于同样的原因将请求转发给老板，这样层层转达直至请求被处理，从中大家可以看到一个显而易见的事，就是至始至终小民只与组长产生了关联，后面具体由谁处理的票据，小民并不关心，唯一在乎的是报账的结果，责任链模式在这里很好地将请求的发起者与处理者解耦。如果我们在代码中模拟这个过程也是很直观的，首先还是先声明一个抽象的领导类。

程序清单 9-10　抽象领导者

```
public abstract class Leader {
    protected Leader nextHandler;// 上一级领导处理者

    /**
     * 处理报账请求
     *
     * @param money 能批复的报账额度
     */
    public final void handleRequest(int money) {
        if (money <= limit()) {
            handle(money);
        } else {
            if (null != nextHandler)
                nextHandler.handleRequest(money);
        }
    }

    /**
     * 自身能批复的额度权限
     *
     * @return 额度
     */
    public abstract int limit();

    /**
     * 处理报账行为
     *
     * @param money 具体金额
     */
    public abstract void handle(int money);
}
```

在这个抽象的领导类中只做了两件事，一是定义了两个抽象接口方法来确定一个领导者应有的行为和属性，二是声明了一个处理报账请求的方法来确定当前领导是否有能力处理报账请求，如果没有这权限，则将该请求转发给上级领导处理。接下来则是各个领导类的实现。

程序清单 9-11　各个具体的领导者

```
public class GroupLeader extends Leader {
    @Override
    public int limit() {
        return 1000;
    }

    @Override
    public void handle(int money) {
        System.out.println("组长批复报销" + money + "元");
    }
}

public class Director extends Leader {
    @Override
    public int limit() {
        return 5000;
    }

    @Override
    public void handle(int money) {
        System.out.println("主管批复报销" + money + "元");
    }
}

public class Manager extends Leader {
    @Override
    public int limit() {
        return 10000;
    }

    @Override
    public void handle(int money) {
        System.out.println("经理批复报销" + money + "元");
    }
}

public class Boss extends Leader {
    @Override
    public int limit() {
        return Integer.MAX_VALUE;
    }

    @Override
    public void handle(int money) {
        System.out.println("老板批复报销" + money + "元");
    }
}
```

最后，小民从组长开始发起请求申请报账。

程序清单 9-12　客户类小民

```
public class XiaoMin {
    public static void main(String[] args) {
        // 构造各个领导对象
        GroupLeader groupLeader = new GroupLeader();
        Director director = new Director();
        Manager manager = new Manager();
        Boss boss = new Boss();
```

```
        // 设置上一级领导处理者对象
        groupLeader.nextHandler = director;
        director.nextHandler = manager;
        manager.nextHandler = boss;

        // 发起报账申请
        groupLeader.handleRequest(50000);
    }
}
```

这里大家可能会想，可不可以直接越过组长找主管报账呢？答案是肯定的，这也是责任链模式的灵活之处，请求的发起可以从责任链的任何一个节点处开始，同时也可以改变责任链内部传递的规则，如主管不在，我们完全可以跨过主管从组长直接将请求转送给经理。

对于责任链中的一个处理者对象，其只有两个行为，一是处理请求，二是将请求转送给下一个节点，不允许某个处理者对象在处理了请求后又将请求转送给上一个节点的情况。对于一条责任链来说，一个请求最终只有两种情况，一是被某个处理对象所处理，另一个是所有对象均未对其处理，对于前一种情况我们称该责任链为纯的责任链，对于后一种情况我们称为不纯的责任链，在实际应用中，我们所见到的责任链模式大多为不纯的责任链。

9.6 Android 源码中的责任链模式实现

责任链模式在Android源码中比较类似的实现莫过于对事件的分发处理，每当用户接触屏幕时，Android 都会将对应的事件包装成一个事件对象从 ViewTree 的顶部至上而下地分发传递，关于事件的包装我们会在命令模式中浅述，关于 IMS 对事件的处理我们也会在访问者模式中讲解，这里不再赘述，这里我们主要来看看 ViewGroup 中是如何将事件派发到子 View 的，我们知道 ViewGroup 中执行事件派发的方法是 dispatchTouchEvent，在该方法中其对事件进行了同一的分发：

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onTouchEvent(ev, 1);
    }

    // 对于辅助功能的事件处理
    if (ev.isTargetAccessibilityFocus() && isAccessibilityFocusedViewOrHost()) {
        ev.setTargetAccessibilityFocus(false);
    }

    boolean handled = false;
    if (onFilterTouchEventForSecurity(ev)) {
        final int action = ev.getAction();
        final int actionMasked = action & MotionEvent.ACTION_MASK;

        // 处理原始的 DOWN 事件
        if (actionMasked == MotionEvent.ACTION_DOWN) {
            // 这里主要是在新事件开始时处理完上一个事件
            cancelAndClearTouchTargets(ev);
            resetTouchState();
        }

        // 检查事件拦截
        final boolean intercepted;
```

```
if (actionMasked == MotionEvent.ACTION_DOWN
        || mFirstTouchTarget != null) {
    final boolean disallowIntercept=(mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
    if (!disallowIntercept) {
        intercepted = onInterceptTouchEvent(ev);
        // 恢复事件防止其改变
        ev.setAction(action);
    } else {
        intercepted = false;
    }
} else {
    intercepted = true;
}

// 如果事件被拦截了，则进行正常的事件分发
if (intercepted || mFirstTouchTarget != null) {
    ev.setTargetAccessibilityFocus(false);
}

// 检查事件是否取消
final boolean canceled = resetCancelNextUpFlag(this)
        || actionMasked == MotionEvent.ACTION_CANCEL;

// 如果有必要的话，为 DOWN 事件检查所有的目标对象
final boolean split = (mGroupFlags & FLAG_SPLIT_MOTION_EVENTS) != 0;
TouchTarget newTouchTarget = null;
boolean alreadyDispatchedToNewTouchTarget = false;

// 如果事件未被取消并未被拦截
if (!canceled && !intercepted) {

// 如果有辅助功能的参与，则直接将事件投递到对应的 View，否则将事件分发给所有的子 View
    View childWithAccessibilityFocus = ev.isTargetAccessibilityFocus()
    ? findChildWithAccessibilityFocus() : null;

    // 如果事件为起始事件
    if (actionMasked == MotionEvent.ACTION_DOWN
            || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
            || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
        final int actionIndex = ev.getActionIndex();
        final int idBitsToAssign = split ? 1 << ev.getPointerId(actionIndex)
                : TouchTarget.ALL_POINTER_IDS;

        removePointersFromTouchTargets(idBitsToAssign);

        final int childrenCount = mChildrenCount;

        // 如果 TouchTarget 为空并且子元素不为 0
        if (newTouchTarget == null && childrenCount != 0) {
            final float x = ev.getX(actionIndex);
            final float y = ev.getY(actionIndex);

            // 由上至下去寻找一个可以接收该事件的子 View
            final ArrayList<View> preorderedList = buildOrderedChildList();
            final boolean customOrder = preorderedList == null
                    && isChildrenDrawingOrderEnabled();
            final View[] children = mChildren;

            // 遍历子元素
            for (int i = childrenCount - 1; i >= 0; i--) {
                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;
            }

            // 如果这个子元素无法接收 Pointer Event 或这个事件点压根就没有落在子元素的边
            // 界范围内
            if (!canViewReceivePointerEvents(child)
                    || !isTransformedTouchPointInView(x, y, child, null)) {
                ev.setTargetAccessibilityFocus(false);

                // 那么就跳出该次循环继续遍历
                continue;
            }

            // 找到 Event 该由哪个子元素持有
            newTouchTarget = getTouchTarget(child);
            if (newTouchTarget != null) {
                newTouchTarget.pointerIdBits |= idBitsToAssign;
                break;
            }

            resetCancelNextUpFlag(child);

            // 投递事件执行触摸操作
            // 如果子元素还是一个 ViewGroup，则递归调用重复此过程
            // 如果子元素是一个 View，那么则会调用 View 的 dispatchTouchEvent，并最终由
            // onTouchEvent 处理
            if(dispatchTransformedTouchEvent(ev,false,child,idBitsToAssign)){
                // 子 View 在其边界范围内接收事件
                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);
        }
        if (preorderedList != null) preorderedList.clear();
    }

    // 如果发现没有子元素可以持有该次事件
    if (newTouchTarget == null && mFirstTouchTarget != null) {
        newTouchTarget = mFirstTouchTarget;
```

```
                while (newTouchTarget.next != null) {
                    newTouchTarget = newTouchTarget.next;
                }
                newTouchTarget.pointerIdBits |= idBitsToAssign;
            }
        }
    }

    if (mFirstTouchTarget == null) {
        handled = dispatchTransformedTouchEvent(ev, canceled, null,
        TouchTarget.ALL_POINTER_IDS);
    } else {
        TouchTarget predecessor = null;
        TouchTarget target = mFirstTouchTarget;
        while (target != null) {
            final TouchTarget next = target.next;
            if (alreadyDispatchedToNewTouchTarget && target == newTouchTarget) {
                handled = true;
            } else {
                final boolean cancelChild = resetCancelNextUpFlag(target.child)
                        || intercepted;
                if (dispatchTransformedTouchEvent(ev, cancelChild,
                        target.child, target.pointerIdBits)) {
                            handled = true;
                }
                if (cancelChild) {
                    if (predecessor == null) {
                        mFirstTouchTarget = next;
                    } else {
                        predecessor.next = next;
                    }
                    target.recycle();
                    target = next;
                    continue;
                }
            }
            predecessor = target;
            target = next;
        }
    }

    if (canceled
            || actionMasked == MotionEvent.ACTION_UP
            || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
                resetTouchState();
    } else if (split && actionMasked == MotionEvent.ACTION_POINTER_UP) {
        final int actionIndex = ev.getActionIndex();
        final int idBitsToRemove = 1 << ev.getPointerId(actionIndex);
        removePointersFromTouchTargets(idBitsToRemove);
    }
}

if (!handled && mInputEventConsistencyVerifier != null) {
    mInputEventConsistencyVerifier.onUnhandledEvent(ev, 1);
}
return handled;
}
```

这里我们主要看看 dispatchTransformedTouchEvent 方法是如何调度子元素 dispatchTouchEvent 方法的：

```
private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel,
  View child, int desiredPointerIdBits) {
    final boolean handled;
    final int oldAction = event.getAction();

    // 如果事件被取消
    if (cancel || oldAction == MotionEvent.ACTION_CANCEL) {
        event.setAction(MotionEvent.ACTION_CANCEL);

        // 如果没有子元素
        if (child == null) {
            // 那么就直接调用父类的 dispatchTouchEvent，注意，这里的父类终会为 View 类
            handled = super.dispatchTouchEvent(event);
        } else {
            // 如果有子元素则传递 cancle 事件
            handled = child.dispatchTouchEvent(event);
        }
        event.setAction(oldAction);
        return handled;
    }

    // 计算即将被传递的点的数量
    final int oldPointerIdBits = event.getPointerIdBits();
    final int newPointerIdBits = oldPointerIdBits & desiredPointerIdBits;

    // 如果事件没有相应的点，那么就丢弃该事件
    if (newPointerIdBits == 0) {
        return false;
    }

    // 声明临时变量保存坐标转换后的 MotionEvent
    final MotionEvent transformedEvent;

    // 如果事件点的数量一致
    if (newPointerIdBits == oldPointerIdBits) {

        // 子元素为空或子元素有一个单位矩阵
        if (child == null || child.hasIdentityMatrix()) {

            // 再次区分子元素为空的情况
            if (child == null) {
                // 为空则调用父类 dispatchTouchEvent
                handled = super.dispatchTouchEvent(event);
            } else {
                // 否则尝试获取 xy 方向上的偏移量（如果通过 scrollTo 或 scrollBy 对子视图进行滚动的话）
                final float offsetX = mScrollX - child.mLeft;
                final float offsetY = mScrollY - child.mTop;

                // 将 MotionEvent 进行坐标变换
                event.offsetLocation(offsetX, offsetY);

                // 再将变换后的 MotionEvent 传递给子元素
                handled = child.dispatchTouchEvent(event);

                // 复位 MotionEvent 以便之后再次使用
                event.offsetLocation(-offsetX, -offsetY);
            }
            // 如果通过以上的逻辑判断，当前事件被持有则可以直接返回
            return handled;
        }
        transformedEvent = MotionEvent.obtain(event);
    } else {
```

```
            transformedEvent = event.split(newPointerIdBits);
        }

        // 下述是一样的不再赘述
        if (child == null) {
            handled = super.dispatchTouchEvent(transformedEvent);
        } else {
            final float offsetX = mScrollX - child.mLeft;
            final float offsetY = mScrollY - child.mTop;
            transformedEvent.offsetLocation(offsetX, offsetY);
            if (! child.hasIdentityMatrix()) {
                transformedEvent.transform(child.getInverseMatrix());
            }

            handled = child.dispatchTouchEvent(transformedEvent);
        }

        transformedEvent.recycle();
        return handled;
    }
```

ViewGroup 事件投递的递归调用就类似于一条责任链，一旦其寻找到责任者，那么将由责任者持有并消费掉该次事件，具体地体现在 View 的 onTouchEvent 方法中返回值的设置，如果 onTouchEvent 返回 false，那么意味着当前 View 不会是该次事件的责任人，将不会对其持有；如果为 true 则相反，此时 View 会持有该事件并不再向外传递。

9.7 责任链模式实战

责任链模式与其说在 Android 中的应用倒不如说是在 Java 中的应用，毕竟上层应用开发大多都是基于 Java 的，而我们的设计模式也是针对 Java 而言，如果你有兴趣，完全可以使用责任链模式来替代各种分支或条件判断语句，不过大多数情况下这都是多此一举的。Android 中我们可以借鉴责任链模式的思想来优化 BroadcastReceiver 使之成为一个全局的责任链处理者，具体方法很简单，我们知道 Broadcast 可以被分为两种，一种是 Normal Broadcast 普通广播，另一种是 Ordered Broadcast 有序广播，普通广播是异步的，发出时可以被所有的接收者收到；而有序广播则是根据优先级依次传播的，直到有接收者将其终止或所有接收者都不终止它，有序广播的这一特性与我们的责任链模式很相近，通过它可以轻松地实现一种全局的责任链事件处理，这里我们创建 3 个 BroadcastReceiver：

```
public class FirstReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // 获取 Intent 中附加的限制值
        int limit = intent.getIntExtra("limit", -1001);

        // 如果限制值等于 1000，则处理，否则继续转发给下一个 Receiver
        if (limit == 1000) {
            // 获取 Intent 中附加的字符串消息并 Toast
            String msg = intent.getStringExtra("msg");
            Toast.makeText(context, msg, Toast.LENGTH_SHORT).show();

            // 终止广播
```

```
            abortBroadcast();
        } else {
            // 添加信息发送给下一个 Receiver
            Bundle b = new Bundle();
            b.putString("new", "Message from FirstReceiver");
            setResultExtras(b);
        }
    }
}
```

在发送广播的时候我们在Intent中附加两个值，一个int类型的值用于存储权限值，另一个String类型的则存储消息，一个接收者能否处理本次广播的唯一条件则是看广播中所附加的权限值是否与自身的相等，如上面FirstReceiver的处理权限值为1000，则只有当limit == 1000时才会处理本次广播，否则在广播的Intent中再附加一条信息传播出去给下一个接收者：

```
public class SecondReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // 获取 Intent 中附加的限制值
        int limit = intent.getIntExtra("limit", -1001);

        // 如果限制值等于 100，则处理，否则继续转发给下一个 Receiver
        if (limit == 100) {
            // 获取 Intent 中附加的字符串消息
            String msg = intent.getStringExtra("msg");

            // 获取上一个 Receiver 增加的消息
            Bundle b = getResultExtras(true);
            String str = b.getString("new");

            // Toast
            Toast.makeText(context, msg + "-----" + str, Toast.LENGTH_SHORT).show();

            // 终止广播
            abortBroadcast();
        } else {
            // 添加信息发送给下一个 Receiver
            Bundle b = new Bundle();
            b.putString("new", "Message from SecondReceiver");
            setResultExtras(b);
        }
    }
}

public class ThirdReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // 获取 Intent 中附加的限制值
        int limit = intent.getIntExtra("limit", -1001);

        // 如果限制值等于 10，则处理，否则继续转发给下一个 Receiver
        if (limit == 10) {
            // 获取 Intent 中附加的字符串消息
            String msg = intent.getStringExtra("msg");

            // 获取上一个 Receiver 增加的消息
            Bundle b = getResultExtras(true);
            String str = b.getString("new");
```

```
            // Toast
            Toast.makeText(context, msg + "-----" + str, Toast.LENGTH_SHORT).show();

            // 终止广播
            abortBroadcast();
        } else {
                // 添加信息发送给下一个 Receiver
                Bundle b = new Bundle();
                b.putString("new", "Message from ThirdReceiver");
                setResultExtras(b);
        }
    }
}
```

最后我们还要在 AndroidManifest.xml 文件中声明这 3 个 Receiver，并设定对应的权限值：

```
<receiver android:name=".broadcast.order.FirstReceiver">
    <intent-filter android:priority="1000">
        <action android:name="com.aigestudio.action.ORDER_BROADCAST"/>
    </intent-filter>
</receiver>
<receiver android:name=".broadcast.order.SecondReceiver">
    <intent-filter android:priority="100">
        <action android:name="com.aigestudio.action.ORDER_BROADCAST"/>
    </intent-filter>
</receiver>
<receiver android:name=".broadcast.order.ThirdReceiver">
    <intent-filter android:priority="10">
        <action android:name="com.aigestudio.action.ORDER_BROADCAST"/>
    </intent-filter>
</receiver>
```

完成后，我们在一个 Activity 中发送一个广播：

```
public class OrderActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_order);

        Button btnSend = (Button) findViewById(R.id.order_send_btn);
        btnSend.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent i = new Intent();
                i.setAction("com.aigestudio.action.ORDER_BROADCAST");
                i.putExtra("limit", 100);
                i.putExtra("msg", "Message from OrderActivity");
                sendOrderedBroadcast(i, null);
            }
        });
    }
}
```

这里我们将 limit 设置为 100，也就是说只有 SecondReceiver 才会处理它，具体效果这里就不演示了，大家可以自行尝试。

9.8 总结

世界不是完美的，所以不会有完美的事物存在。就像所有的设计模式一样，有优点也有缺点，但总体来说优点必定大于缺点或者说缺点相对于优点来说更可控。责任链模式也一样，优点显而易见，可以对请求者和处理者关系解耦，提高代码的灵活性。责任链模式的最大缺点是对链中请求处理者的遍历，如果处理者太多那么遍历必定会影响性能，特别是在一些递归调用中，要慎重。

第 10 章　化繁为简的翻译机——解释器模式

10.1 解释器模式介绍

解释器模式（Interpreter Pattern）是一种用得比较少的行为型模式，其提供了一种解释语言的语法或表达式的方式，该模式定义了一个表达式接口，通过该接口解释一个特定的上下文。在这么多的设计模式中，解释器模式在实际运用上相对来说要少很多，因为我们很少会自己去构造一个语言的文法。虽然如此，既然它能够在设计模式中有一席之位，那么必定有它的可用之处，本章会以最直白的言语来阐述清楚解释器模式是如何工作的。

10.2 解释器模式的定义

给定一个语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

与其他设计模式不同的是，解释器模式涉及编程语言理论知识较多，就拿上面对该模式的定义来说，可能会有很多读者根本看不懂这句话的意思，什么叫文法？为什么叫解释器？其又是如何表达的？要彻底搞懂其含义，我们首先要对文法有一个大体的认识。何谓文法？举个简单的例子大家一看就会懂，假设有如下短语。

> 我是程序员

这个只有 5 个字的短语我相信没人会说看不懂，有朋友会说这与我们要讲的文法有什么关系呢？确实，这一个短语体现不了什么，那么我们再看几条。

> 我是设计师
> 我是搬运工
> ……

那么上面的这些短语有什么规律呢？回顾一下我们初中语文或英语的知识，上面这些短语中的“我”可以看成是主语，而“是”则表示谓语，“程序员”、“设计师”和“搬运工”这些名词则可看成宾语，也就是说上面的这些短语都可看成是一个“主谓宾”的结构，而这样的结构我们则称为一条文法，我们可以通过该文法来造出更多符合该文法的语句。当然文法的概念范围非常广，并不局限于主谓宾、定状补这样的语法结构，还是用上面的短语来举例，我们也可以把上面的这些短语看

成是一条“我是[名词]”这样的结构，这也可以看作一条文法。

如上所述文法的概念范围很广，对于我们程序员来说更愿意接受 abcd 这类字符形式的表示方式，假设有如下以 ab 开头 ef 结尾中间排列 N(N>=0)个 cd 的字符串。

```
abcd......cdef
```

随着 N 值的不同具体的字符串也会不一样，我们可以从中得到类似“abef”、“abcdef”、“abcdcdef”之类的字符串。试想一下，是否可以将其表示为一个具体的表达式规则呢？答案是肯定的。在计算机科学中，我们将上述字符串中的“a”、“b”、“c”、“d”、“e”和“f”这 6 个字符称为一种形式语言的字符表，而这些字符组成的集合，如上面的“abcd.....cdef”这样由字符表构成的字符串则称为形式语言，注意，这里是“语言”不是“文法”。假如定义一个符号 S（也可以是 A、F、G 等），从符号 S 出发推导上述字符串，那么就可以得到如下的推导式：

```
S ::= abA*ef
A ::= cd
```

其中符号“::=”表示推导；符号“*”表示闭包，意思就是符号 A 可以有 0 或 N 个重复；S 和 A 则称非终结符号，因为它们能推导出式子右边的表达式，同时又因为整个推导式是从 S 出发的，因此，这个 S 也称为初始符号；而 abef 和 cd 这些字符不能再被推导我们将之为终结符号。而上面的推导式意思也很简单，与我们小学时学的因式分解极其相似。像这样的从一个具体的符号出发，通过不断地应用一些产生式规则从而生成一个字符串的集合，我们将描述这个集合的文法称为形式文法，顾名思义，形式文法与形式语言相对应，用来描述形式语言。一般情况下，解释器模式中描述的也是形式语言，定义的也是形式文法，当然你也可以用来描述语言和文法，但它们的范畴都太大，而且对我们编程来说很少会涉猎。

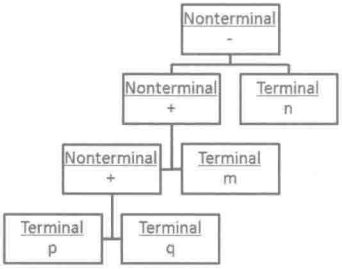
我们对形式语言和形式文法有了一个初步的了解后再来审读一下解释器模式的定义：给定一个语言（如由 abcdef 六个字符组成的字符串集合），定义它的文法的一种表示（如上面给出的 S ::= abA*ef 和 A ::= cd），并定义一个解释器，该解释器使用该表示来解释语言中的句子。这样看起来相对来说是不是更好理解些了？现在只有解释器没说清楚了，究竟什么是解释器，它又是如何工作的呢？与其说成解释倒不如说成翻译更好理解，你也可以将解释器简单地理解为一个翻译机。还是用上面的例子中的文法来举例，其就是用来翻译类似“abef”、“abcdef”和“abcdcdef”之类的字符串句子，这样一来是否对解释器模式的定义有了更进一步的理解了？在编程的时候我们很少会涉及如上那种标准的文法推导式，很多时候会直接使用类似“A*B”这样的字符串规则表达式表示文法，该文法表示“AigeAigeAigeStudio”、“Studio”、“AigeStudio”之类的字符串。

10.3 解释器模式的使用场景

解释器模式的使用场景其实相当广泛，总的概括下来大致有如下两种。

(1)如果某个简单的语言需要解释执行而且可以将该语言中的语句表示为一个抽象语法树时可以考虑使用解释器模式。

这种场景很好理解，如一个简单的含有加减运算的数学表达式：p+q+m-n，像这样的表达式其构成无非就两种，一种是以 pqmn 这类的具体参数表示的符号，其无法再被推导，如上面我们所述，其也被称为终结符号；另一种则是以“+”和“-”构成的算术运算符，在该运算符的两边总能找到有意义的具体计算参数，我们也称其为非终结符号，如上的数学表达式我们也可以将其表示为一棵抽象语法树，如图 10-1 所示。



▲图 10-1

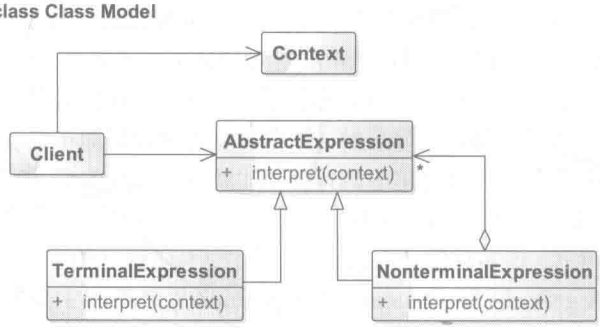
那么为什么要说是“简单的语言”呢？试想一下，如果我们的表达式是 p+q-m/n*x%y^z 这样的呢？这个表达式看起来也不算“复杂”，先别急，等我们看完之后的内容你就会知道这样的一个表达式对于使用解释器模式来解释是多么复杂。

（2）在某些特定的领域出现不断重复的问题时，可以将该领域的问题转化为一种语法规则下的语句，然后构建解释器来解释该语句。

如需要将一段阿拉伯数字转换为中文的数字，又或者将某个小写英文短语转换为大写，如 aigestudio 这样的字符串，我们需要将其转换为大写 AIGESTUDIO，这时对于这样的转换来说，其实就是一个不断重复的问题，因为所有的阿拉伯或中文数字和大小写字母都是固定的，也就是说它们都是一个个终结符，不同的只是其具体内容而已，这时候就可以尝试使用解释器模式来解决类似的问题。

10.4 解释器模式的 UML 类图

解释器模式的通用类图如图 10-2 所示。



▲图 10-2

根据类图可以得出如下一个解释器模式的通用模式代码。

程序清单 10-1　抽象表达式

```
public abstract class AbstractExpression {
    /**
     * 抽象的解析方法
     *
```

```
     * @param ctx 上下文环境对象
     */
    public abstract void interpret(Context ctx);
}
```

程序清单 10-2 终结符表达式

```
public class TerminalExpression extends AbstractExpression {
    @Override
    public void interpret(Context ctx) {
        // 实现文法中与终结符有关的解释操作
    }
}
```

程序清单 10-3 非终结符表达式

```
public class NonterminalExpression extends AbstractExpression {
    @Override
    public void interpret(Context ctx) {
        // 实现文法中与非终结符有关的解释操作
    }
}
```

程序清单 10-4 上下文环境类 包含解释器之外的全局信息

```
public class Context {
}
```

程序清单 10-5 客户类

```
public class Client {
    public static void main(String[] args) {
        // 根据文法对特定句子构建抽象语法树后解释
    }
}
```

角色介绍。

AbstractExpression：抽象表达式。

声明一个抽象的解释操作父类，并定义一个抽象的解释方法，其具体的实现在各个具体的子类解释器中完成。

TerminalExpression：终结符表达式。

实现文法中与终结符有关的解释操作。文法中每一个终结符都有一个具体的终结表达式与之对应。

NonterminalExpression：非终结符表达式。

实现文法中与非终结符有关的解释操作。

Context：上下文环境类。

包含解释器之外的全局信息。

Client：客户类。

解析表达式，构建抽象语法树，执行具体的解释操作等。

10.5 解释器模式的简单实现

如本章开头所说，解释器模式的应用范围相当广泛，一个比较常见的场景是对算术表达式的解

释，如表达式“m + n + p”，如果我们使用解释器模式对该表达式进行解释，那么代表数字的 m、n 和 p 三个字母我们就可以看成是终结符号，而“+”这个算术运算符号则可当作非终结符号。同样地，我们可以先创建一个抽象解释器表示数学运算。

程序清单 10-6　抽象的算术运算解释器 为所有解释器共性的提取

```
public abstract class ArithmeticExpression {
    /**
     * 抽象的解析方法
     * 具体的解析逻辑由具体的子类实现
     *
     * @return 解析得到具体的值
     */
    public abstract int interpret();
}
```

在该抽象解释器的解释方法 interpret 中，我们没有像前面的例子那样使用一个 Context 对象作为 interpret 方法的签名，在本例中运算的结果都是作为参数返回，因此，没有必要使用额外的对象存储信息。ArithmeticExpression 有两个直接子类 NumExpression 和 OperatorExpression，其中 NumExpression 用于对数字进行解释：

程序清单 10-7　数字解释器 仅仅为了解释数字

```
public class NumExpression extends ArithmeticExpression {
    private int num;

    public NumExpression(int num) {
        this.num = num;
    }

    @Override
    public int interpret() {
        return num;
    }
}
```

代码很简单，逻辑也很明确，就不多说了。OperatorExpression 依然是一个抽象类，其声明两个 ArithmeticExpression 类型的成员变量存储运算符号两边的数字解释器，这两个成员变量会在构造方法中被赋值。

程序清单 10-8　运算符号抽象解释器 为所有运算符号解释器共性的提取

```
public abstract class OperatorExpression extends ArithmeticExpression {
    // 声明两个成员变量存储运算符号两边的数字解释器
    protected ArithmeticExpression exp1, exp2;

    public OperatorExpression(ArithmeticExpression exp1,
            ArithmeticExpression exp2) {
                this.exp1 = exp1;
                this.exp2 = exp2;
    }
}
```

OperatorExpression 也有一个直接子类 AdditionExpression，顾名思义其表示对加法运算进行解释，其逻辑都很简单，就不做过多说明了。

程序清单 10-9 加法运算抽象解释器

```
public class AdditionExpression extends OperatorExpression {
    public AdditionExpression(ArithmeticExpression exp1,
            ArithmeticExpression exp2) {
                super(exp1, exp2);
    }

    @Override
    public int interpret() {
        return exp1.interpret() + exp2.interpret();
    }
}
```

上面就是本例中所要使用到的所有解释器。除此之外，我们创建一个 Calculator 类来处理一些相关的业务。

程序清单 10-10 处理与解释相关的一些业务

```
public class Calculator {
    // 声明一个 Stack 栈存储并操作所有相关的解释器
    private Stack<ArithmeticExpression> mExpStack = new Stack<ArithmeticExpression>();

    public Calculator(String expression) {
        // 声明两个 ArithmeticExpression 类型的临时变量，存储运算符左右两边的数字解释器
        ArithmeticExpression exp1, exp2;

        // 根据空格分割表达式字符串
        String[] elements = expression.split(" ");

        /*
         * 循环遍历表达式元素数组
         */
        for (int i = 0; i < elements.length; i++) {
            /*
             * 判断运算符号
             */
            switch (elements[i].charAt(0)) {
            case '+':// 如果是加号
                // 则将栈中的解释器弹出作为运算符号左边的解释器
                exp1 = mExpStack.pop();

                // 同时将运算符号数组下标下一个元素构造为一个数字解释器
                exp2 = new NumExpression(Integer.valueOf(elements[++i]));

                // 通过上面两个数字解释器构造加法运算解释器
                mExpStack.push(new AdditionExpression(exp1, exp2));
                break;
            default:// 如果为数字
                /*
                 * 如果不是运算符则为数字
                 * 若是数字，直接构造数字解释器并压入栈
                 */
                mExpStack.push(new NumExpression(Integer.valueOf(elements[i])));
                break;
            }
        }
    }

    /**
     * 计算结果
     *
```

```
     * @return 最终的计算结果
     */
    public int calculate() {
        return mExpStack.pop().interpret();
    }
}
```

这里要注意的是，为了简化问题我们约定表达式字符串的每个元素之间必须使用空格间隔开，如“1 + 22 + 333 + 4444”这样的表达式字符串则是合法的，而“1+22+333+4444”则不合法，因此，我们才能在 Calculator 的构造方法中通过空格来拆分字符串。Calculator 类的逻辑很好理解，这里还是以“1 + 22 + 333 + 4444”这个字符串为例，首先将其拆分为有 7 个元素组成的字符串数组，然后循环遍历，首先遍历到的元素为 1，那么将其作为参数构造一个 NumExpression 对象压入栈，其次是加号运算符，此时我们则将刚才压入栈的由元素 1 作为参数构造的 NumExpression 对象抛出作为加号运算符左边的数字解释器，而右边的数字解释器呢？我们只需要将当前数组下标+1 获取到数组元素中加号右边的数字 22，将其作为参数构造一个 NumExpression 对象即可，最后通过左右两个数字解释器作为参数构造一个 AdditionExpression 加法解释器对象压入栈即可，这个过程其实就是在构建语法树，只不过我们将其单独封装在了一个类里而不是在 Client 客户类里进行，最后，我们公布一个 calculate 方法执行解释并返回结果。后面的 Client 客户类就很简单了，构造一个 Calculator 对象，调用 calculate 方法输出结果即可。

程序清单 10-11　客户类

```
public class Client {
    public static void main(String[] args) {
        Calculator c = new Calculator("153 + 3589 + 118 + 555");
        System.out.println(c.calculate());
    }
}
```

上面我们只是简单地对加法运算定义了解释器，如果现在又想引入减法运算怎么办呢？很简单，定义一个减法解释器即可。

程序清单 10-12　减法运算抽象解释器

```
public class SubtractionExpression extends OperatorExpression {
    public SubtractionExpression(ArithmeticExpression exp1,
    ArithmeticExpression exp2) {
        super(exp1, exp2);
    }

    @Override
    public int interpret() {
        return exp1.interpret() - exp2.interpret();
    }
}
```

同样地，SubtractionExpression 也继承于 OperatorExpression，表示对运算符号的解释器，其实现逻辑也很简单不再多说。然后我们还需修改一下 Calculator 类中构建语法树的逻辑，添加一条对“-”号的分支判断处理即可。

```
case '-':// 如果是减号
    exp1 = mExpStack.pop();
```

```
    exp2 = new NumExpression(Integer.valueOf(elements[++i]));
    mExpStack.push(new SubtractionExpression(exp1, exp2));
    break;
```

这样，我们则可以处理我们的减法运算了。

程序清单 10-13　客户类

```
public class Client {
    public static void main(String[] args) {
        Calculator c = new Calculator("153 + 3589 + 118 - 555 - 597 -66");
        System.out.println(c.calculate());
    }
}
```

具体的输出结果就不多说了，大家可自行尝试，这里我们可以看到解释器模式的一个优点，就是灵活性强，上面的例子中我们只实现了对加减法的解释计算，如果想实现更多的运算法则，如乘除取余等，只需创建对应的运算解释器即可，但是混合运算要比简单的加减法运算复杂得多，还要考虑不同符号的运算优先级，这也是文章开头我们说，在“简单的语言”中适用解释器模式。

从上面的两个例子中可以看到，具体的文法规则与解释器之间其实是由对应关系的，大多数情况下两者之间是一一对应的关系，即一条文法对应一个解释器，当然，我们也可以为一条文法创建多个不同的解释器，但是反过来就不行。这个很好理解，如上例中对于加法解释器我们实现的是对加法运算的解释，其对应一个解释器 AdditionExpression，当然也可以再为其创建一个解释器 XXXXXXExpression，但是一个解释器却不能即解释加法运算又解释减法运算，否则就违背了解释器模式的定义。说到解释器模式的定义，我们提到过抽象语法树，而我们在上面两个例子中都有构建抽象语法树的相关逻辑，第一个例子中我们在客户类里构建由 17 个数字和一个数字或字母构成的语法树；而第二个例子则根据具体表达式动态构建相应的语法树。从这点可以看出解释器模式并不包含对抽象语法树的构建，其构建逻辑应由客户根据具体的情况去生成。

将一条具体的文法通过一个解释器解释，把复杂的文法规则分离为简单的功能进行解释，最后将其组合成一棵抽象语法树解释执行。至此，我们可以看到解释器模式的原理与本质：将复杂的问题简单化、模块化，分离实现、解释执行。

10.6 Android 源码中的解释器模式实现

对于 Android 来说，解释器模式的应用并不多见，我们也很难在系统的源码中找到其经典实现，但是，我们依然可以在一些地方看到对解释器模式原理的应用。相信大家自从第一天学习 Android 起就知道 AndroidManifest.xml 这个应用配置文件，如果说我们的应用是一本书的话，这个配置文件就相当于书的目录，其中包含大量应用配置的声明定义，那么在 Android 中是如何读取这个配置文件的呢？这里就不得不说到 PackageParser 这个类，该类对 AndroidManifest.xml 中每一个组件标签创建了相应的类用以存储相应的信息：

```
package android.content.pm;

public class PackageParser {
    public final static class Package {}
    public final static class Permission extends Component<IntentInfo> {}
```

```
    public final static class PermissionGroup extends Component<IntentInfo> {}
    public final static class Activity extends Component<ActivityIntentInfo> {}
    public final static class Service extends Component<ServiceIntentInfo> {}
    public final static class Provider extends Component<ProviderIntentInfo> {}
    public final static class Instrumentation extends Component {}
    public final static class ActivityIntentInfo extends IntentInfo {}
    public final static class ServiceIntentInfo extends IntentInfo {}
    public static final class ProviderIntentInfo extends IntentInfo {}
}
```

如上代码所示，PackageParser 为 Activity、Service、Provider、Permission 等构件在其内部以内部类的方式创建了对应的类，按照解释器模式的定义，这些类其实都对应 AndroidManifest.xml 文件中的一个标签，也就是一条文法，其在对该配置文件解析时充分运用了解释器模式分离实现、解释执行的特性。

在 Andorid 中，解析某个 apk 文件会调用到 PackageManagerService（以下简称 PMS）中的 scanPackageLI 方法，该方法有两种实现：

```
private PackageParser.Package scanPackageLI(File scanFile, int parseFlags,
      int scanMode, long currentTime, UserHandle user);

private PackageParser.Package scanPackageLI(PackageParser.Package pkg,
      int parseFlags, int scanMode, long currentTime, UserHandle user);
```

两者唯一的区别是签名列表中第一个参数，第一种实现为 File 类型的对象，而第二种实现为 PackageParser.Package 类型的对象。在具体解析某个文件时会先调用第一种实现解析 apk 文件，再调用第二种实现将解析后的信息保存至 PMS：

```
private PackageParser.Package scanPackageLI(File scanFile, int parseFlags,
      int scanMode, long currentTime, UserHandle user) {
         mLastScanError = PackageManager.INSTALL_SUCCEEDED;

         // 获取文件路径
         String scanPath = scanFile.getPath();

         // 设置解析标志 mDefParseFlags 默认为 0
         parseFlags |= mDefParseFlags;

         // 使用 scanPath 作为参数构造 PackageParser 对象
         PackageParser pp = new PackageParser(scanPath);
         // ......省去两行代码......

         // 调用 PackageParser 对象的 parsePackage 方法解析 APK 文件
         final PackageParser.Package pkg = pp.parsePackage(scanFile, scanPath,
         mMetrics, parseFlags);
         // ......省去无数行代码......

         // 调用该方法的第二种实现，将解析后的信息保存至 PMS
         PackageParser.Package scannedPkg = scanPackageLI(pkg, parseFlags,
               scanMode | SCAN_UPDATE_SIGNATURE, currentTime, user);

         // ......省去多行代码......
         return scannedPkg;
}
```

这里重点分析 PackageParser 类，该类如上所说，承担着具体的解析重任，上述代码中首先调

用了其 parsePackage 方法，该方法也有两种实现：

```
public Package parsePackage(File sourceFile, String destCodePath,
DisplayMetrics metrics, int flags);

private Package parsePackage(Resources res, XmlResourceParser parser,
int flags, String[] outError);
```

第一种实现就是上面代码中调用到的方法，其主要逻辑就是为第二种实现准备参数后调用第二个方法：

```
public Package parsePackage(File sourceFile, String destCodePath,
        DisplayMetrics metrics, int flags) {
    // ......省去多行代码......
    try {
        // 调用第二种实现进行解析
        pkg = parsePackage(res, parser, flags, errorText);
    } catch (Exception e) {
        // ......省去两行代码......
    }
    // ......省去多行代码......
    return pkg;
}
```

而 parsePackage 方法的第二种实现逻辑则要复杂得多，在其内部主要对整个 AndroidManifest.xml 配置文件进行具体的解析：

```
private Package parsePackage(Resources res, XmlResourceParser parser,
        int flags, String[] outError) throws XmlPullParserException,
        IOException {
            // ......省去无数行代码......
            final Package pkg = new Package(pkgName);
            // ......省去多行代码......
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() > outerDepth)) {
        // ......省去 3 行代码......

        String tagName = parser.getName();
        if (tagName.equals("application")) {
            // ......省去多行代码......
            if (!parseApplication(pkg, res, parser, attrs, flags, outError)) {
                return null;
            }
        } else if (tagName.equals("keys")) {
            if (!parseKeys(pkg, res, parser, attrs, outError)) {
                return null;
            }
        } else if (tagName.equals("permission-group")) {
            if (parsePermissionGroup(pkg, flags, res, parser, attrs,
                    outError) == null) {
                        return null;
            }
        } else if (tagName.equals("permission")) {
            if (parsePermission(pkg, res, parser, attrs, outError) == null) {
                return null;
            }
        } else if (tagName.equals("permission-tree")) {
            if (parsePermissionTree(pkg, res, parser, attrs, outError) == null) {
                return null;
```

```
            }
        } else if (tagName.equals("uses-permission")) {
            if (!parseUsesPermission(pkg, res, parser, attrs, outError)) {
                return null;
            }
        } else if (tagName.equals("uses-configuration")) {
            // ......省去多行代码......
        } else if (tagName.equals("uses-feature")) {
            // ......省去多行代码......
        } else if (tagName.equals("uses-sdk")) {
            // ......省去多行代码......
        } else if (tagName.equals("supports-screens")) {
            // ......省去多行代码......
        } else if (tagName.equals("protected-broadcast")) {
            // ......省去多行代码......
        } else if (tagName.equals("instrumentation")) {
            // ......省去两行代码......
        } else if (tagName.equals("original-package")) {
            // ......省去多行代码......
        } else if (tagName.equals("adopt-permissions")) {
            // ......省去多行代码......
        } else if (tagName.equals("uses-gl-texture")) {
            // ......省去两行代码......
        } else if (tagName.equals("compatible-screens")) {
            // ......省去两行代码......
        } else if (tagName.equals("supports-input")) {
            // ......省去两行代码......
        } else if (tagName.equals("eat-comment")) {
            // ......省去两行代码......
        } else if (RIGID_PARSER) {
            // ......省去 4 行代码......
        } else {
            // ......省去两行代码......
        }
    }
    // ......省去很多行代码......
    return pkg;
}
```

可以看到，parsePackage 方法的主要作用其实就是对 AndroidManifest.xml 配置文件中 manifest 下的每个子节点进行解析，这里主要来看看 parseApplication 方法是如何对 application 节点进行解析的：

```
private boolean parseApplication(Package owner, Resources res,
        XmlPullParser parser, AttributeSet attrs, int flags,
        String[] outError) throws XmlPullParserException, IOException {
            // ......省去对 application 节点内属性解析的代码......
            final int innerDepth = parser.getDepth();

    int type;
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() > innerDepth)) {
        if (type == XmlPullParser.END_TAG || type == XmlPullParser.TEXT) {
            continue;
        }
        String tagName = parser.getName();
        if (tagName.equals("activity")) {
            Activity a = parseActivity(owner, res, parser, attrs, flags,
            outError, false, hardwareAccelerated);
            if (a == null) {
```

```
            // ......省去一行代码......
            return false;
        }
        owner.activities.add(a);
    } else if (tagName.equals("receiver")) {
        Activity a = parseActivity(owner, res, parser, attrs, flags,
                outError, true, false);
        if (a == null) {
            // ......省去一行代码......
            return false;
        }
        owner.receivers.add(a);
    } else if (tagName.equals("service")) {
        Service s = parseService(owner, res, parser, attrs, flags, outError);
        if (s == null) {
            // ......省去一行代码......
            return false;
        }
        owner.services.add(s);
    } else if (tagName.equals("provider")) {
        Provider p = parseProvider(owner, res, parser, attrs, flags, outError);
        if (p == null) {
            // ......省去一行代码......
            return false;
        }
        owner.providers.add(p);
    } else if (tagName.equals("activity-alias")) {
            Activity a = parseActivityAlias(owner, res, parser, attrs,
            flags, outError);
        if (a == null) {
            // ......省去一行代码......
            return false;
        }
        owner.activities.add(a);
    } else if (parser.getName().equals("meta-data")) {
        if ((owner.mAppMetaData = parseMetaData(res, parser, attrs,
                owner.mAppMetaData, outError)) == null) {
            // ......省去一行代码......
            return false;
        }
    } else if (tagName.equals("library")) {
        // ......省去多行代码......
    } else if (tagName.equals("uses-library")) {
        // ......省去多行代码......
    } else if (tagName.equals("uses-package")) {
        // ......省去一行代码......
    } else {
        // ......省去多行代码......
    }
  }
  return true;
}
```

从程序中可以看到，parseApplication 除了对 application 节点内的属性进行解析外，其主要逻辑还是通过遍历解析其子节点，然后继续分别调用不同的解析方法对其进行解析，这里以 parseActivity 为例，parseActivity 的具体逻辑执行如下：

```
private Activity parseActivity(Package owner, Resources res,
        XmlPullParser parser, AttributeSet attrs, int flags,
```

```
        String[] outError, boolean receiver, boolean hardwareAccelerated)
        throws XmlPullParserException, IOException {
            // ......省去多行代码......
            mParseActivityArgs.tag = receiver ? "<receiver>" : "<activity>";
            // ......省去两行代码......

            // 注意，这里的 Activity 是继承于 Component 的，而非我们平时应用开发所用的 Activity
            Activity a = new Activity(mParseActivityArgs, new ActivityInfo());
            // ......省去多行代码......

            int outerDepth = parser.getDepth();
            int type;
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() > outerDepth)) {
        if (type == XmlPullParser.END_TAG || type == XmlPullParser.TEXT) {
            continue;
        }

        if (parser.getName().equals("intent-filter")) {
            ActivityIntentInfo intent = new ActivityIntentInfo(a);
            if (!parseIntent(res, parser, attrs, true, intent, outError)) {
                return null;
            }
            if (intent.countActions() == 0) {
            } else {
                a.intents.add(intent);
            }
        } else if (!receiver && parser.getName().equals("preferred")) {
            ActivityIntentInfo intent = new ActivityIntentInfo(a);
            if (!parseIntent(res, parser, attrs, false, intent, outError)) {
                return null;
            }
            if (intent.countActions() == 0) {
            } else {
                if (owner.preferredActivityFilters == null) {
                    owner.preferredActivityFilters = new ArrayList<ActivityIntentInfo>();
                }
                owner.preferredActivityFilters.add(intent);
            }
        } else if (parser.getName().equals("meta-data")) {
            if ((a.metaData = parseMetaData(res, parser, attrs, a.metaData,
                    outError)) == null) {
                return null;
            }
        } else {
            // ......省去多行代码......
        }
    }
    // ......省去 3 行代码......
    return a;
}
```

这里要注意的是，parseActivity 方法不仅承担着对 Activity 的解析，其同样承担着对 Broadcast 的解析。与 parseApplication 方法类似，parseActivity 内部逻辑也是遍历其子标签，并调用相应的方法对其进行解析，如上述代码中的 parseIntent 和 parseMetaData，其逻辑也很好理解，这里就不再多说了。

10.7 关于 PackageManagerService

上面我们在分析 PackageParser 的时候提到 PackageManagerService（以下简称 PMS），顾名思义其作用就是对我们的 apk 安装包进行管理的服务，其类定义在 frameworks\base\services\java\com\android\server\pm\PackageManagerService.java 中，该类在 Android API 19 中有一万多行代码，虽然代码很多，但是，大部分逻辑都是在处理类似我们上面所说的文件解析和数据结构操作，作为 Android 在 Java 层的服务之一，其与所有其他的服务一样都是由 SystemServer 所启动：

```
class ServerThread {
    public void initAndLoop() {
        // 准备消息循环
        Looper.prepareMainLooper();
        // ......省去多行代码......

        // 声明 IPackageManager 临时变量
        IPackageManager pm = null;
        // ......省去无数行代码......
        try {
            // 通过 PackageManagerService 的静态 main 方法得到 IPackageManager 对象
            pm = PackageManagerService.main(context, installer,
                    factoryTest != SystemServer.FACTORY_TEST_OFF, onlyCore);
        } catch (RuntimeException e) {
            // ......省去两行代码......
        }
        // ......省去无数行代码......

        // 进入消息循环
        Looper.loop();
    }
    // ......省去几行代码......
}

public class SystemServer {
    public static final int FACTORY_TEST_OFF = 0;
    // ......省去几行代码......
    public static void main(String[] args) {
        // ......省去多行代码......
        ServerThread thr = new ServerThread();
        thr.initAndLoop();
    }
}
```

从程序中可以看到，在 SystemServer 的 main 方法中构造了一个 ServerThread 对象，然后调用其 initAndLoop 方法，initAndLoop 方法逻辑相当多，其主要目的很简单，就是初始化各种系统服务，并启动消息循环开始接收处理消息。在 initAndLoop 方法中，通过 PackageManagerService 的静态 main 方法得到一个 IPackageManager 对象：

```
public class PackageManagerService extends IPackageManager.Stub {
    // ......省去多行代码......
    public static final IPackageManager main(Context context,
      Installer installer, boolean factoryTest, boolean onlyCore) {
            // 构造一个包管理服务对象
            PackageManagerService m = new PackageManagerService(context,
```

```
                installer, factoryTest, onlyCore);
        // 将该包管理服务添加至 ServiceManager
        ServiceManager.addService("package", m);
        return m;
    }
    // ......省去多行代码......
}
```

ServiceManager 是 Binder 进程间通信机制的守护进程，其目的很简单，就是管理 Android 系统里的 Binder 对象。这里我们重点来看 PackageManagerService 的构造过程：

```
public PackageManagerService(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
        // 向事件日志写入事件，标识 PackageManagerService 启动
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_START,
        SystemClock.uptimeMillis());
    /*
     * SDK 版本检查
     */
    if (mSdkVersion <= 0) {
        Slog.w(TAG, "**** ro.build.version.sdk not set!");
    }
    // 上下文环境对象
    mContext = context;
    // 开机模式是否为工厂模式
    mFactoryTest = factoryTest;
    // 是否仅启动内核
    mOnlyCore = onlyCore;
    // 如果编译版本为 eng 则不需要 dex 优化
    mNoDexOpt = "eng".equals(SystemProperties.get("ro.build.type"));
    // 构造 DisplayMetrics 对象以便获取尺寸数据
    mMetrics = new DisplayMetrics();
    // 构造 Settings 对象存储系统运行时的设置信息
    mSettings = new Settings(context);
    /*
     * 添加一些用户 uid
     */
    mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM_UID,
            ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_PRIVILEGED);
    mSettings.addSharedUserLPw("android.uid.phone", RADIO_UID,
            ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_PRIVILEGED);
    mSettings.addSharedUserLPw("android.uid.log", LOG_UID,
            ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_PRIVILEGED);
    mSettings.addSharedUserLPw("android.uid.nfc", NFC_UID,
            ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_PRIVILEGED);
    mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH_UID,
            ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_PRIVILEGED);
    mSettings.addSharedUserLPw("android.uid.shell", SHELL_UID,
            ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_PRIVILEGED);
    /*
     * 判断是否在不同的进程
     */
    String separateProcesses = SystemProperties
            .get("debug.separate_processes");
    if (separateProcesses != null && separateProcesses.length() > 0) {
        if ("*".equals(separateProcesses)) {
            mDefParseFlags = PackageParser.PARSE_IGNORE_PROCESSES;
            mSeparateProcesses = null;
            Slog.w(TAG, "Running with debug.separate_processes: * (ALL)");
        } else {
            mDefParseFlags = 0;
```

```
        mSeparateProcesses = separateProcesses.split(",");
        Slog.w(TAG, "Running with debug.separate_processes: "
                + separateProcesses);
    }
} else {
    mDefParseFlags = 0;
    mSeparateProcesses = null;
}
// Installer 由 SystemServer 构造，这里通过该对象负责与底层通信，进行具体的安装卸载等操作
mInstaller = installer;
/*
 * 测量显示对象
 * 这段代码相信不少读者都用过
 */
WindowManager wm = (WindowManager) context
        .getSystemService(Context.WINDOW_SERVICE);
Display d = wm.getDefaultDisplay();
d.getMetrics(mMetrics);

synchronized (mInstallLock) {
    synchronized (mPackages) {
        // 启动消息处理线程
        mHandlerThread.start();
        // 通过消息处理线程的 Looper 对象构造一个处理消息的 Handler 对象
        mHandler = new PackageHandler(mHandlerThread.getLooper());
        // 使用看门狗监测当前消息处理线程
        Watchdog.getInstance().addThread(mHandler, mHandlerThread.getName());
        // 获取当前的 Data 目录
        File dataDir = Environment.getDataDirectory();
        // 具体对应的是/data/data 目录
        mAppDataDir = new File(dataDir, "data");
        // 具体对应的是/data/app 目录
        mAppInstallDir = new File(dataDir, "app");
        // 具体对应的是/data/app-lib 目录
        mAppLibInstallDir = new File(dataDir, "app-lib");
        // 具体对应的是/data/app-asec 目录
        mAsecInternalPath = new File(dataDir, "app-asec").getPath();
        // 具体对应的是/data/user 目录
        mUserAppDataDir = new File(dataDir, "user");
        // 具体对应的是/data/app-private 目录
        mDrmAppPrivateInstallDir = new File(dataDir, "app-private");
        // 构造 UserManagerService 对象
        sUserManager = new UserManagerService(context, this,
                mInstallLock, mPackages);
        // 读取并解析/etc/permissions 下的权限配置文件中的信息并保存到全局变量
        readPermissions();
        // 尝试读取并解析 mac_permissions.xml 文件
        mFoundPolicyFile = SELinuxMMAC.readInstallPolicy();
        // 读取并解析 package.xml 等配置文件
        mRestoredSettings = mSettings.readLPw(this,
                sUserManager.getUsers(false), mSdkVersion, mOnlyCore);
        /*
         * 判断是否有自定义的解析界面
         */
        String customResolverActivity = Resources.getSystem()
                .getString(R.string.config_customResolverActivity);
        if (TextUtils.isEmpty(customResolverActivity)) {
            customResolverActivity = null;
        } else {
            mCustomResolverComponentName = ComponentName
                    .unflattenFromString(customResolverActivity);
        }
```

```
// 标识扫描开始时间的时间戳
long startTime = SystemClock.uptimeMillis();
// 将事件写入事件日志
EventLog.writeEvent(
    EventLogTags.BOOT_PROGRESS_PMS_SYSTEM_SCAN_START, startTime);
// 设置扫描模式
int scanMode = SCAN_MONITOR | SCAN_NO_PATHS | SCAN_DEFER_DEX | SCAN_BOOTING;
if (mNoDexOpt) {
    Slog.w(TAG, "Running ENG build: no pre-dexopt!");
    // 是否对 APK 文件进行 dex 优化
    scanMode |= SCAN_NO_DEX;
}
// 构造一个 String 类型的 HashSet，保存已经 dex 优化的文件路径
final HashSet<String> alreadyDexOpted = new HashSet<String>();
// 获取 Java 启动类库的路径，并将其添加至 alreadyDexOpted
String bootClassPath = System
        .getProperty("java.boot.class.path");
if (bootClassPath != null) {
    String[] paths = splitString(bootClassPath, ':');
    for (int i = 0; i < paths.length; i++) {
        alreadyDexOpted.add(paths[i]);
    }
} else {
    Slog.w(TAG, "No BOOTCLASSPATH found!");
}
// 标识外部 lib 是否有优化
boolean didDexOpt = false;
/*
 * 确保所有的外部 lib 都被优化
 */
if (mSharedLibraries.size() > 0) {
    Iterator<SharedLibraryEntry> libs = mSharedLibraries
    .values().iterator();
    while (libs.hasNext()) {
        String lib = libs.next().path;
        if (lib == null) {
            continue;
        }
        try {
            if (dalvik.system.DexFile.isDexOptNeeded(lib)) {
                alreadyDexOpted.add(lib);
                mInstaller.dexopt(lib, Process.SYSTEM_UID, true);
                didDexOpt = true;
            }
        } catch (FileNotFoundException e) {
            Slog.w(TAG, "Library not found: " + lib);
        } catch (IOException e) {
            Slog.w(TAG, "Cannot dexopt " + lib
                        + "; is it an APK or JAR? "
                        + e.getMessage());
        }
    }
}
/*
 * 将/system/framework 目录下两个文件添加至 alreadyDexOpted，而不需要优化
 */
File frameworkDir = new File(Environment.getRootDirectory(),"framework");
alreadyDexOpted.add(frameworkDir.getPath()+ "/framework-res.apk");
alreadyDexOpted.add(frameworkDir.getPath() + "/core-libart.jar");
// 列出/system/frameworks 目录下的文件
String[] frameworkFiles = frameworkDir.list();
if (frameworkFiles != null) {
```

```
        for (int i = 0; i < frameworkFiles.length; i++) {
            File libPath = new File(frameworkDir, frameworkFiles[i]);
            String path = libPath.getPath();
            /*
             * 跳过已经优化的文件
             */
            if (alreadyDexOpted.contains(path)) {
                continue;
            }
            /*
             * 跳过不是 APK 或 JAR 的文件
             */
            if (!path.endsWith(".apk") && !path.endsWith(".jar")) {
                continue;
            }
            try {
                /*
                 * 判断 APK 或 JAR 文件是否需要优化
                 */
                if (dalvik.system.DexFile.isDexOptNeeded(path)) {
                    // 通过 Installer 进行优化
                    mInstaller.dexopt(path, Process.SYSTEM_UID, true);
                    didDexOpt = true;
                }
            } catch (FileNotFoundException e) {
                    Slog.w(TAG, "Jar not found: " + path);
            } catch (IOException e) {
                    Slog.w(TAG, "Exception reading jar: " + path, e);
            }
        }
    }
    /*
     * 如果上面的代码中我们对某个文件做过 dex 优化
     * 那么 didDexOpt 就会设置为 true，而本段代码逻辑就会被执行
     */
    if (didDexOpt) {
        // 获取/data/dalvik-cache 目录的 File 对象
        File dalvikCacheDir = new File(dataDir, "dalvik-cache");
        // 遍历该目录下的文件
        String[] files = dalvikCacheDir.list();
        if (files != null) {
            for (int i = 0; i < files.length; i++) {
                String fn = files[i];
                /*
                 * 删除以"data@app@"和"data@app-private@"开头的文件
                 */
                if (fn.startsWith("data@app@")
                        || fn.startsWith("data@app-private@")) {
                    Slog.i(TAG, "Pruning dalvik file: " + fn);
                    (new File(dalvikCacheDir, fn)).delete();
                }
            }
        }
    }
    // 构造 AppDirObserver 对象，监控/system/frameworks 目录的变化
    mFrameworkInstallObserver = new AppDirObserver(
            frameworkDir.getPath(), OBSERVER_EVENTS, true, false);
            mFrameworkInstallObserver.startWatching();
// 调用 scanDirLI 方法扫描/system/frameworks 目录下的 apk 文件进行安装，扫描模式为非优化模式
    scanDirLI(frameworkDir, PackageParser.PARSE_IS_SYSTEM
            | PackageParser.PARSE_IS_SYSTEM_DIR, scanMode
            | SCAN_NO_DEX, 0);
```

```
/*
 * 这一步与上面类似只不过其针对的是/system/priv-app 目录
 */
File privilegedAppDir = new File(
        Environment.getRootDirectory(), "priv-app");
mPrivilegedInstallObserver = new AppDirObserver(
        privilegedAppDir.getPath(), OBSERVER_EVENTS, true, true);
        mPrivilegedInstallObserver.startWatching();
        scanDirLI(privilegedAppDir, PackageParser.PARSE_IS_SYSTEM
        | PackageParser.PARSE_IS_SYSTEM_DIR
        | PackageParser.PARSE_IS_PRIVILEGED, scanMode, 0);
/*
 * 同上，针对的是/system/app 目录
 */
File systemAppDir = new File(Environment.getRootDirectory(),"app");
mSystemInstallObserver = new AppDirObserver(
        systemAppDir.getPath(), OBSERVER_EVENTS, true, false);
        mSystemInstallObserver.startWatching();
        scanDirLI(systemAppDir, PackageParser.PARSE_IS_SYSTEM
        | PackageParser.PARSE_IS_SYSTEM_DIR, scanMode, 0);
/*
 * 同上，针对的是/vendor/app 目录
 */
File vendorAppDir = new File("/vendor/app");
mVendorInstallObserver = new AppDirObserver(
        vendorAppDir.getPath(), OBSERVER_EVENTS, true, false);
        mVendorInstallObserver.startWatching();
        scanDirLI(vendorAppDir, PackageParser.PARSE_IS_SYSTEM
        | PackageParser.PARSE_IS_SYSTEM_DIR, scanMode, 0);
if (DEBUG_UPGRADE)
    Log.v(TAG, "Running installd update commands");
mInstaller.moveFiles();
// 构造一个 String 类型的 ArrayList 列表，存储不存在的系统 packages 路径
final List<String> possiblyDeletedUpdatedSystemApps = new ArrayList<String>();
if (!mOnlyCore) {
    // 遍历 Settings 的成员变量 mPackages
    Iterator<PackageSetting> psit = mSettings.mPackages
            .values().iterator();
    while (psit.hasNext()) {
        PackageSetting ps = psit.next();
        /*
         * 如果不是系统 App，则跳过
         */
        if ((ps.pkgFlags & ApplicationInfo.FLAG_SYSTEM) == 0) {
            continue;
        }
        /*
         * 如果已经被扫描到了
         */
        final PackageParser.Package scannedPkg = mPackages
                .get(ps.name);
        if (scannedPkg != null) {
            /*
             * 如果在 disabled 列表中
             * 那么，说明它是通过 OTA 方式升级更新添加的
             * 因此，清除相应的数据
             */
            if (mSettings.isDisabledSystemPackageLPr(ps.name)) {
                Slog.i(TAG,
                        "Expecting better updatd system app for "
                                + ps.name
                                + "; removing system app");
```

```
                // 移除其信息
                removePackageLI(ps, true);
            }
            continue;
        }
        /*
         * 如果不在 disabled 列表中
         * 则直接清除相应的数据
         */
        if (!mSettings.isDisabledSystemPackageLPr(ps.name)) {
            psit.remove();
            String msg = "System package " + ps.name
                    + " no longer exists; wiping its data";
            reportSettingsProblem(Log.WARN, msg);
            removeDataDirsLI(ps.name);
        }
        /*
         * 否则，通过 codePath 判断其是否有可能被更新或删除
         */
        else {
            final PackageSetting disabledPs = mSettings
                    .getDisabledSystemPkgLPr(ps.name);
            if (disabledPs.codePath == null
                    || !disabledPs.codePath.exists()) {
                possiblyDeletedUpdatedSystemApps.add(ps.name);
            }
        }
    }
}
// 获取未完成安装的 apk 包 PackageSetting 列表
ArrayList<PackageSetting> deletePkgsList = mSettings
        .getListOfIncompleteInstallPackagesLPr();
// 清除未完成安装的安装包
for (int i = 0; i < deletePkgsList.size(); i++) {
    cleanupInstallFailedPackage(deletePkgsList.get(i));
}
// 并删除临时文件
deleteTempPackageFiles();
// 清除在 mSettings 中没有被使用的 ShareUserSettins
mSettings.pruneSharedUsersLPw();
/*
 * 如果是普通模式，则还需要处理一些额外的情况
 */
if (!mOnlyCore) {
    EventLog.writeEvent(
            EventLogTags.BOOT_PROGRESS_PMS_DATA_SCAN_START,
            SystemClock.uptimeMillis());
    // 构造 AppDirObserver 对象，监控/data/app 目录的变化
    mAppInstallObserver = new AppDirObserver(
            mAppInstallDir.getPath(), OBSERVER_EVENTS, false,
            false);
    mAppInstallObserver.startWatching();
    // 扫描该目录
    scanDirLI(mAppInstallDir, 0, scanMode, 0);
    // 构造 AppDirObserver 对象，监控/data/app-private 目录的变化
    mDrmAppInstallObserver = new AppDirObserver(
            mDrmAppPrivateInstallDir.getPath(),
            OBSERVER_EVENTS, false, false);
    mDrmAppInstallObserver.startWatching();
    // 扫描该目录
    scanDirLI(mDrmAppPrivateInstallDir,
            PackageParser.PARSE_FORWARD_LOCK, scanMode, 0);
```

```
        /**
         * 后面这部分代码逻辑较简单
         * 遍历 possiblyDeletedUpdatedSystemApps
         * 处理通过 OTA 更新和删除的 APK 文件
         */
        for (String deletedAppName : possiblyDeletedUpdatedSystemApps) {
            PackageParser.Package deletedPkg = mPackages
                    .get(deletedAppName);
            mSettings
                    .removeDisabledSystemPackageLPw(deletedAppName);

            String msg;
            if (deletedPkg == null) {
                msg = "Updated system package " + deletedAppName
                        + " no longer exists; wiping its data";
                removeDataDirsLI(deletedAppName);
            } else {
                msg = "Updated system app + "
                        + deletedAppName
                        + " no longer present; removing system privileges for "
                        + deletedAppName;

                deletedPkg.applicationInfo.flags &= ~ApplicationInfo.FLAG_SYSTEM;

                PackageSetting deletedPs = mSettings.mPackages
                        .get(deletedAppName);
                deletedPs.pkgFlags &= ~ApplicationInfo.FLAG_SYSTEM;
            }
            reportSettingsProblem(Log.WARN, msg);
        }
    } else {
        mAppInstallObserver = null;
        mDrmAppInstallObserver = null;
    }
    // 给需要使用 ShareLibrary 的 package 找到相应的 ShareLibrary 路径,并将其保存至 package
    // 的 usesLibraryFiles 中
    updateAllSharedLibrariesLPw();

    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SCAN_END,
            SystemClock.uptimeMillis());
    Slog.i(TAG, "Time to scan packages: "
                + ((SystemClock.uptimeMillis() - startTime) / 1000f)
                + " seconds");
    /*
     * 校对 SDK 版本
     */
    final boolean regrantPermissions = mSettings.mInternalSdkPlatform != mSdkVersion;
    if (regrantPermissions)
        Slog.i(TAG, "Platform changed from "
                + mSettings.mInternalSdkPlatform + " to "
                + mSdkVersion
                + "; regranting permissions for internal storage");
    mSettings.mInternalSdkPlatform = mSdkVersion;
    // 更新相关信息，并给需要使用权限的 APK 分配相应的权限
    updatePermissionsLPw(
            null,
            null,
            UPDATE_PERMISSIONS_ALL| (regrantPermissions ? (UPDATE_PERMISSIONS_REPLACE_
            PKG | UPDATE_PERMISSIONS_REPLACE_ALL) : 0));
    /*
     * 如果本次启动是第一次正常的启动
     * 那么同时还需要初始化一些默认的首选 App
```

```
            */
            if (!mRestoredSettings && !onlyCore) {
                mSettings.readDefaultPreferredAppsLPw(this, 0);
            }
            // 将信息写到相关文件中
            mSettings.writeLPr();

            EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_READY,
                    SystemClock.uptimeMillis());
            // 通知 GC 回收资源
            Runtime.getRuntime().gc();

            mRequiredVerifierPackage = getRequiredVerifierLPr();
        }
    }
}
```

整个 PackageManagerService 的构造过程很长，但是，逻辑并不复杂，条理也很清晰，关于 PackageManagerService 中的一些具体方法，大家可以自行查阅相关资料，这里不再赘述。总体来说，PackageManagerService 的作用就如它名字所描述的那样，主要用来管理应用程序包，其与其他系统服务的通信也比较单一，并不复杂，因此，相对于 Android 中的一些核心服务来说还算是比较好理解的。

10.8 总结

优点

解释器模式的优点我们在本文的两个例子中已有所提及，最大的优点是其灵活的扩展性，当我们想对文法规则进行扩展延伸时，只需要增加相应的非终结符解释器，并在构建抽象语法树时，使用到新增的解释器对象进行具体的解释即可，非常方便。

缺点

解释器模式的缺点也显而易见，因为对于每一条文法都可以对应至少一个解释器，其会生成大量的类，导致后期维护困难；同时，对于过于复杂的文法，构建其抽象语法树会显得异常烦琐，甚至有可能会出现需要构建多棵抽象语法树的情况，因此，对于复杂的文法并不推荐使用解释器模式。

第 11 章　让程序畅通执行——命令模式

11.1 命令模式介绍

命令模式（Command Pattern），是行为型设计模式之一。命令模式相对于其他的设计模式来说并没有那么多的条条框框，其实它不是一个很“规矩”的模式，不过，就是基于这一点，命令模式相对于其他的设计模式更为灵活多变。我们接触比较多的命令模式个例无非就是程序菜单命令，如在操作系统中，我们点击“关机”命令，系统就会执行一系列的操作，如先是暂停处理事件，保存系统的一些配置，然后结束程序进程，最后调用内核命令关闭计算机等，对于这一系列的命令，用户不用去管，用户只需点击系统的关机按钮即可完成如上一系列的命令。而我们的命令模式其实也与之相同，将一系列的方法调用封装，用户只需调用一个方法执行，那么所有这些被封装的方法就会被挨个执行调用。

11.2 命令模式的定义

将一个请求封装成一个对象，从而让用户使用不同的请求把客户端参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。

11.3 命令模式的使用场景

需要抽象出待执行的动作，然后以参数的形式提供出来——类似于过程设计中的回调机制，而命令模式正是回调机制的一个面向对象的替代品。

在不同的时刻指定、排列和执行请求。一个命令对象可以有与初始请求无关的生存期。

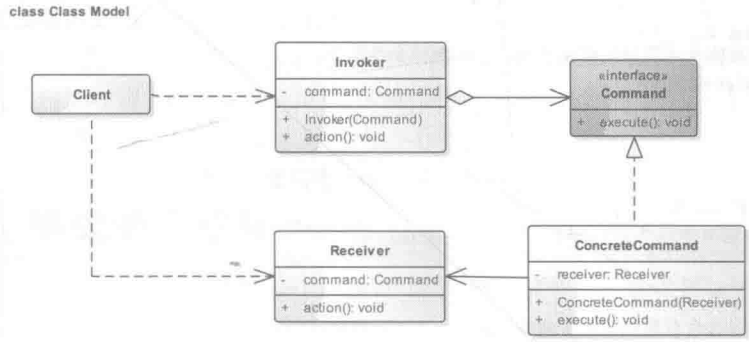
需要支持取消操作。

支持修改日志功能，这样当系统崩溃时，这些修改可以被重做一遍。

需要支持事务操作。

11.4 命令模式的 UML 类图

UML 类图如图 11-1 所示。



▲图 11-1

根据类图可以得出如下一个命令模式的通用模式代码。

程序清单 11-1　接收者类

```
public class Receiver {
    /**
     * 真正执行具体命令逻辑的方法
     */
    public void action() {
        System.out.println("执行具体操作");
    }
}
```

程序清单 11-2　抽象命令接口

```
public interface Command {
    /**
     * 执行具体操作的命令
     */
    void execute();
}
```

程序清单 11-3　具体命令类

```
public class ConcreteCommand implements Command {
    private Receiver receiver;// 持有一个对接收者对象的引用

    public ConcreteCommand(Receiver receiver) {
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        // 调用接收者的相关方法来执行具体逻辑
        receiver.action();
    }
}
```

程序清单 11-4　请求者类

```
public class Invoker {
    private Command command;// 持有一个对相应命令对象的引用

    public Invoker(Command command) {
        this.command = command;
    }
```

```
    public void action() {
        // 调用具体命令对象的相关方法，执行具体命令
        command.execute();
    }
}
```

程序清单 11-5　客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造一个接收者对象
        Receiver receiver = new Receiver();

        // 根据接收者对象构造一个命令对象
        Command command = new ConcreteCommand(receiver);

        // 根据具体的对象构造请求者对象
        Invoker invoker = new Invoker(command);

        // 执行请求方法
        invoker.action();
    }
}
```

角色介绍。

Receiver：接收者角色。

该类负责具体实施或执行一个请求，说得通俗点就是，执行具体逻辑的角色，以本章开头的“关机”命令操作为例，其接收者角色就是真正执行各项关机逻辑的底层代码。任何一个类都可以成为一个接收者，而在接收者类中封装具体操作逻辑的方法我们则称为行动方法。

Command：命令角色。

定义所有具体命令类的抽象接口。

ConcreteCommand：具体命令角色。

该类实现了 Command 接口，在 execute 方法中调用接收者角色的相关方法，在接收者和命令执行的具体行为之间加以弱耦合。而 execute 则通常称为执行方法，如本文开头所述“关机的”操作实现，具体可能还包含很多相关的操作，比如保存数据、关闭文件、结束进程等，如果将这一系列具体的逻辑处理看作接收者，那么调用这些具体逻辑的方法就可以看作是执行方法。

Invoker：请求者角色。

该类的职责就是调用命令对象执行具体的请求，相关的方法我们称为行动方法，还是用“关机”为例，“关机”这个菜单命令一般就对应一个关机方法，我们点击了“关机”命令后，由这个关机方法去调用具体的命令执行具体的逻辑，这里的“关机”对应的这个方法就可以看做是请求者。

Client：客户端角色。

以“关机”的例子来说就相当于人，很好理解不再多说。

这里其实大家可以看到，命令模式的应用其实可以用一句话来概述，就是将行为调用者与实现者解耦，一个简单的例子，我们常会这样来调用代码：

```
Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
mPaint.setShader(new SweepGradient(canvas.getWidth() / 2, canvas.getHeight() / 2, new int[]
{ 0xff44adde, 0xffbc4579, 0xffe9c43f, 0xffdccb48, 0xff44adde }, null));
```

这样的逻辑调用再熟悉不过了，先是 new 一个 Paint 对象，紧接着调用 Paint 中的 setShader 方法为 Paint 设置一个着色器。但是，如果我的调用逻辑比较复杂或者说调用行为有多种实现时，这种将行为调用者与行为实现者耦合在一起的写法就会很有问题。这时候使用设计模式来解耦就很有必要，当然，这里只是作为一个引子给大家思考，不多做分析。

11.5 命令模式的简单实现

命令模式总体来说并不难，只是相对比较烦琐，你想想一个简单的调用关系被解耦成多个部分，必定会增加类的复杂度，但是即便如此，命令模式的结构依然清晰。大家小时候应该都玩过俄罗斯方块游戏，这里以古老的俄罗斯方块游戏为例，看看我们在命令模式下是如何操控俄罗斯方块变换的。一般来说，俄罗斯方块游戏中都有 4 个按钮，两个左右移动的按钮，一个快速落下的按钮，还有一个变化方块形状的按钮，这是比较经典的游戏原型。一个玩游戏的人就相当于我们的客户端，而游戏上的 4 个按钮就相当于请求者，或者也可以称为调用者，执行具体按钮命令的逻辑方法可以看作是命令角色，当然，游戏内部具体是怎么实现的我们不知道，也不在这里探讨，仅作例子分析，最后真正执行处理具体逻辑的则是游戏本身，你可以看作是各种机器码计算处理来执行的具体逻辑，这里我们将它看作是接收者角色。逻辑分析比较清楚了，我们来将其“翻译”成代码，首先是我们的接收者，这里以俄罗斯方块游戏本身作为接收者角色。

程序清单 11-6　接收者角色 俄罗斯方块游戏

```
public class TetrisMachine {
    /**
     * 真正处理"向左"操作的逻辑代码
     */
    public void toLeft() {
        System.out.println("向左");
    }

    /**
     * 真正处理"向右"操作的逻辑代码
     */
    public void toRight() {
        System.out.println("向右");
    }

    /**
     * 真正处理"快速落下"操作的逻辑代码
     */
    public void fastToBottom() {
        System.out.println("快速落下");
    }

    /**
     * 真正处理"改变形状"操作的逻辑代码
     */
    public void transform() {
        System.out.println("改变形状");
    }
}
```

TetrisMachine 类是整个命令模式中唯一处理具体代码逻辑的地方，其他的类都是直接或间接地

调用到该类的方法，这就是接收者角色，处理具体的逻辑。如上文我们所说，接收者类只是一个普通的类，任何类都可以作为接收者。接下来我们定义一个接口作为命令角色的抽象。

程序清单 11-7　命令者抽象 定义执行方法

```
public interface Command {
    /**
     * 命令执行方法
     */
    void execute();
}
```

然后就是 4 个具体命令：向左移、向右移、掉落和变换。

程序清单 11-8　具体命令者 向左移的命令类

```
public class LeftCommand implements Command {
    // 持有一个接收者俄罗斯方块游戏对象的引用
    private TetrisMachine machine;

    public LeftCommand(TetrisMachine machine) {
        this.machine = machine;
    }

    @Override
    public void execute() {
        // 调用游戏机里的具体方法执行操作
        machine.toLeft();
    }
}
```

程序清单 11-9　具体命令者 向右移的命令类

```
public class RightCommand implements Command {
    // 持有一个接收者俄罗斯方块游戏对象的引用
    private TetrisMachine machine;

    public RightCommand(TetrisMachine machine) {
        this.machine = machine;
    }

    @Override
    public void execute() {
        // 调用游戏机里的具体方法执行操作
        machine.toRight();
    }
}
```

程序清单 11-10　具体命令者 快速落下的命令类

```
public class FallCommand implements Command {
    // 持有一个接收者俄罗斯方块游戏对象的引用
    private TetrisMachine machine;

    public FallCommand(TetrisMachine machine) {
        this.machine = machine;
    }

    @Override
    public void execute() {
        // 调用游戏里的具体方法执行操作
        machine.fastToBottom();
    }
}
```

程序清单 11-11 具体命令者 快速落下的命令类

```
public class TransformCommand implements Command {
    // 持有一个接收者俄罗斯方块游戏对象的引用
    private TetrisMachine machine;

    public TransformCommand(TetrisMachine machine) {
        this.machine = machine;
    }

    @Override
    public void execute() {
        // 调用游戏机里的具体方法执行操作
        machine.transform();
    }
}
```

从程序中可以看到，命令者角色类中的方法名称与 TetrisMachine 接收者角色类中的方法名可以不一样，两者之间仅是一种弱耦合。对于请求者，这里我们以一个 Buttons 类来表示，命令由按钮来执行。

程序清单 11-12 请求者类 命令由按钮发起

```
public class Buttons {
    private LeftCommand leftCommand;// 向左移动的命令对象引用
    private RightCommand rightCommand;// 向右移动的命令对象引用
    private FallCommand fallCommand;// 快速落下的命令对象引用
    private TransformCommand transformCommand;// 变换形状的命令对象引用

    /**
     * 设置向左移动的命令对象
     *
     * @param leftCommand 向左移动的命令对象
     */
    public void setLeftCommand(LeftCommand leftCommand) {
        this.leftCommand = leftCommand;
    }

    /**
     * 设置向右移动的命令对象
     *
     * @param rightCommand 向右移动的命令对象
     */
    public void setRightCommand(RightCommand rightCommand) {
        this.rightCommand = rightCommand;
    }

    /**
     * 设置快速落下的命令对象
     *
     * @param fallCommand 快速落下的命令对象
     */
    public void setFallCommand(FallCommand fallCommand) {
        this.fallCommand = fallCommand;
    }

    /**
     * 设置变换形状的命令对象
     *
     * @param transformCommand 变换形状的命令对象
     */
```

```
    public void setTransformCommand(TransformCommand transformCommand) {
        this.transformCommand = transformCommand;
    }

    /**
     * 按下按钮向左移动
     */
    public void toLeft() {
        leftCommand.execute();
    }

    /**
     * 按下按钮向右移动
     */
    public void toRight() {
        rightCommand.execute();
    }

    /**
     * 按下按钮快速落下
     */
    public void fall() {
        fallCommand.execute();
    }

    /**
     * 按下按钮改变形状
     */
    public void transform() {
        transformCommand.execute();
    }
}
```

最后，由客户端来决定如何调用。

程序清单 11-13　客户类

```
public class Player {
    public static void main(String[] args) {
        // 首先要有俄罗斯方块游戏
        TetrisMachine machine = new TetrisMachine();

        // 根据游戏我们构造 4 种命令
        LeftCommand leftCommand = new LeftCommand(machine);
        RightCommand rightCommand = new RightCommand(machine);
        FallCommand fallCommand = new FallCommand(machine);
        TransformCommand transformCommand = new TransformCommand(machine);

        // 按钮可以执行不同的命令
        Buttons buttons = new Buttons();
        buttons.setLeftCommand(leftCommand);
        buttons.setRightCommand(rightCommand);
        buttons.setFallCommand(fallCommand);
        buttons.setTransformCommand(transformCommand);

        // 具体按下哪个按钮玩家说了算
        buttons.toLeft();
        buttons.toRight();
        buttons.fall();
        buttons.transform();
    }
}
```

或许大家在看了这一长篇代码后心存疑惑，明明就是一个很简单的调用逻辑，为什么要做得如此复杂呢？对于大部分开发者来说更愿意接受如下的代码。

程序清单 11-14　示例客户类

```
public class Player {
    public static void main(String[] args) {
        //俄罗斯方块游戏
        TetrisMachine machine = new TetrisMachine();

        // 要实现怎样的控制方式，我们就直接调用相关的函数就行
        machine.toLeft();
        machine.toRight();
        machine.fastToBottom();
        machine.transform();
    }
}
```

调用逻辑做得如此复杂，这是因为开发起来方便，每次我们增加或修改游戏功能只需修改 TetrisMachine 类就行了，然后对应地改一改 Player 类，一切都很方便。但是，对开发者自己来说是方便了，那么，如果有一天开发者不再负责这个项目了呢？这样的逻辑留给后来者，没人会觉得方便。设计模式有一条重要的原则：对修改关闭对扩展开放，大家可以细细体会。

除此之外，使用命令模式的另一个好处是可以实现命令记录的功能，如在上例中，我们在请求者 Buttons 里使用一个数据结构来存储执行过的命令对象，以此可以方便地知道刚刚执行过哪些命令动作，并可以在需要时恢复，具体代码大家可自行尝试，这里不再给出。

11.6 Android 源码中的命令模式实现

Android 中关于命令模式的使用虽然不少，但是并不算是典型，很多方面的应用与其他大多数设计模式一样都有一定的变种，一个比较典型的例子是在 Android 的事件机制中底层逻辑对事件的转发处理，Android 的每一种事件在屏幕上产生后都会经由底层逻辑将其转换为一个 NotifyArgs 对象，NotifyArgs 本身并无任何实现，只是定义了抽象的方法体：

```
struct NotifyArgs {
    virtual ~NotifyArgs() { }

    virtual void notify(const sp<InputListenerInterface>& listener) const = 0;
};
```

注：本书不会过多涉及 Android 底层的逻辑实现 这里的代码仅作介绍不会深入分析，这里以按键事件为例，在经过转化后，一次按键事件会被封装为一个 NotifyKeyArgs 对象：

```
struct NotifyKeyArgs : public NotifyArgs {
    nsecs_t eventTime;
    int32_t deviceId;
    uint32_t source;
    uint32_t policyFlags;
    int32_t action;
    int32_t flags;
    int32_t keyCode;
    int32_t scanCode;
```

```
    int32_t metaState;
    nsecs_t downTime;

    inline NotifyKeyArgs() { }

    NotifyKeyArgs(nsecs_t eventTime, int32_t deviceId, uint32_t source, uint32_t
    policyFlags, int32_t action, int32_t flags, int32_t keyCode, int32_t scanCode, int32_t
    metaState, nsecs_t downTime);

    NotifyKeyArgs(const NotifyKeyArgs& other);

    virtual ~NotifyKeyArgs() { }

    virtual void notify(const sp<InputListenerInterface>& listener) const;
};
```

这里的 NotifyArgs 实质上就相当于一个命令者抽象，而 NotifyKeyArgs 等子类则相当于一个具体命令者，那么谁来承担请求者的角色呢？由 Android 事件机制中一个重要的角色 InputDispatcher 来承担，具体的事件命令者将事件转发给 InputDispatcher，并由其来封装具体的事件操作，这里依然以按键事件为例，在 InputDispatcher 中的执行函数为 notifyKey：

```
void InputDispatcher::notifyKey(const NotifyKeyArgs* args) {
#if DEBUG_INBOUND_EVENT_DETAILS
    // ......省略几行调试代码......
#endif
    if (!validateKeyEvent(args->action)) {
        return;
    }

    uint32_t policyFlags = args->policyFlags;
    int32_t flags = args->flags;
    int32_t metaState = args->metaState;
    if((policyFlags & POLICY_FLAG_VIRTUAL)||(flags & AKEY_EVENT_FLAG_VIRTUAL_HARD_KEY)){
        policyFlags |= POLICY_FLAG_VIRTUAL;
        flags |= AKEY_EVENT_FLAG_VIRTUAL_HARD_KEY;
    }
    if (policyFlags & POLICY_FLAG_FUNCTION) {
        metaState |= AMETA_FUNCTION_ON;
    }
    policyFlags |= POLICY_FLAG_TRUSTED;

    int32_t keyCode = args->keyCode;
    if (metaState & AMETA_META_ON && args->action == AKEY_EVENT_ACTION_DOWN) {
        int32_t newKeyCode = AKEYCODE_UNKNOWN;
        if (keyCode == AKEYCODE_DEL) {
            newKeyCode = AKEYCODE_BACK;
        } else if (keyCode == AKEYCODE_ENTER) {
            newKeyCode = AKEYCODE_HOME;
        }
        if (newKeyCode != AKEYCODE_UNKNOWN) {
            AutoMutex _l(mLock);
            struct KeyReplacement replacement = {keyCode, args->deviceId};
            mReplacedKeys.add(replacement, newKeyCode);
            keyCode = newKeyCode;
            metaState &= ~AMETA_META_ON;
        }
    } else if (args->action == AKEY_EVENT_ACTION_UP) {
        AutoMutex _l(mLock);
        struct KeyReplacement replacement = {keyCode, args->deviceId};
```

```
        ssize_t index = mReplacedKeys.indexOfKey(replacement);
        if (index >= 0) {
            keyCode = mReplacedKeys.valueAt(index);
            mReplacedKeys.removeItemsAt(index);
            metaState &= ~AMETA_META_ON;
        }
    }
    KeyEvent event;
    event.initialize(args->deviceId, args->source, args->action, flags, keyCode,
    args->scanCode, metaState, 0, args->downTime, args->eventTime);

    mPolicy->interceptKeyBeforeQueueing(&event, /*byref*/ policyFlags);

    bool needWake;
    {
        mLock.lock();
        if (shouldSendKeyToInputFilterLocked(args)) {
            mLock.unlock();

            policyFlags |= POLICY_FLAG_FILTERED;
            if (!mPolicy->filterInputEvent(&event, policyFlags)) {
                return;
            }
            mLock.lock();
        }
        int32_t repeatCount = 0;
        KeyEntry* newEntry = new KeyEntry(args->eventTime, args->deviceId, args->source,
        policyFlags, args->action, flags, keyCode, args->scanCode, metaState, repeatCount,
        args->downTime);

        needWake = enqueueInboundEventLocked(newEntry);
        mLock.unlock();
    }
    if (needWake) {
        mLooper->wake();
    }
}
```

事件对命令模式应用的这部分逻辑还算明朗，除了上述的命令者与请求者角色外，接收者角色则由具体的硬件驱动承担，过于底层的机制讲解不再本书的范畴，这里不再多说。除了事件这块程序外，Android 在对应用程序包管理的部分也有对命令模式应用的体现。PackageManagerService 也是 Android 系统的 Service 之一，其主要功能在于实现对应用包的解析、管理、卸载等操作，在 PackageManagerService 中，其对包的相关消息处理由其内部类 PackageHandler 承担，其将需要处理的请求作为对象通过消息传递给相关的方法，而对于包的安装、移动以及包大小的测量则分别封装为 HandlerParams 的 3 个具体子类 InstallParams、MoveParams 和 MeasureParams，HandlerParams 的逻辑不多，主要是对 3 个抽象方法的声明：

```
private abstract class HandlerParams {
    private static final int MAX_RETRIES = 4;

    private int mRetries = 0;

    private final UserHandle mUser;

    HandlerParams(UserHandle user) {
        mUser = user;
    }
```

```
    UserHandle getUser() {
        return mUser;
    }

    final boolean startCopy() {
        boolean res;
        try {
            if (++mRetries > MAX_RETRIES) {
                mHandler.sendEmptyMessage(MCS_GIVE_UP);
                handleServiceError();
                return false;
            } else {
                handleStartCopy();
                res = true;
            }
        } catch (RemoteException e) {
            mHandler.sendEmptyMessage(MCS_RECONNECT);
            res = false;
        }
        handleReturnCode();
        return res;
    }

    final void serviceError() {
        handleServiceError();
        handleReturnCode();
    }

    abstract void handleStartCopy() throws RemoteException;
    abstract void handleServiceError();
    abstract void handleReturnCode();
}
```

这里结合命令模式来看，HandlerParams 也是一个抽象命令者，而对于其 3 个子类则对应于各个功能不同的具体命令角色，请求者依然可以看作是由 PackageManagerService 承担的。

11.7 Android 事件输入系统介绍

上面我们在做模式源码分析时曾提到 InputDispatcher 这个类，它是 Android 事件输入系统的一个重要部分，整个 Android 事件输入系统可以分为几个部分。首当其冲的是 InputReader，其主要作用就是将输入事件从硬件节点中读取后转化为一个 Event 事件，并将该事件分发给 InputDispatcher，InputDispatcher 的职责就是接收来自 InputReader 的 Event 事件，并像其名字所述那样，将事件分发给合适的窗口并监控 ANR 的发生；然后则是创建 InputReader 和 InputDispatcher 的父亲 InputManager，由其创建两者并提供 Policy 对事件进行预处理；最后则是涉及系统 Service 以及面向用户的几个相关模块，如 ActivityManager、WindowManager 和 View 等。

在 Android 4.4 中，事件输入的 Service 部分实现在/frameworks/base/services/input 下，而在 Android 5.0 中该部分实现移至/frameworks/native/services/inputflinger/下，虽然我们将整个事件输入系统分为如上的 4 个部分，但是，在 Android 中负责管理事件输入的是 InputManagerService，其也是系统级的 Service 之一，主要职责就是协调上面我们所述的 4 个部分，InputManagerService 和其他系统级的 Service 一样，在 SystemServer 中被初始化。

```
public final class SystemServer {
    // ......省去部分代码......

    private void startOtherServices() {
        // ......省去部分代码......

        inputManager = new InputManagerService(context);

        wm = WindowManagerService.main(context, inputManager, mFactoryTestMode !=
        FactoryTest.FACTORY_TEST_LOW_LEVEL, !mFirstBoot, mOnlyCore);

        ServiceManager.addService(Context.WINDOW_SERVICE, wm);
        ServiceManager.addService(Context.INPUT_SERVICE, inputManager);

        mActivityManagerService.setWindowManager(wm);

        inputManager.setWindowManagerCallbacks(wm.getInputMonitor());
        inputManager.start();

        // ......省去部分代码......
    }
    // ......省去一个方法代码......
}
```

从上述源码中可以看到，在 SystemServer 的 startOtherServices 方法中，不但构造并启动了 InputManagerService，而且在启动 InputManagerService 之前将它自身注入到 WindowManagerService 中进行关联。InputManagerService 的构造主要是对 native 层初始化，但是它的实现逻辑非常复杂：

```
public InputManagerService(Context context) {
    this.mContext = context;
    this.mHandler = new InputManagerHandler(DisplayThread.get().getLooper());

    mUseDevInputEventForAudioJack = context.getResources().getBoolean(R.bool.
    config_useDevInputEventForAudioJack);
    mPtr = nativeInit(this, mContext, mHandler.getLooper().getQueue());

    LocalServices.addService(InputManagerInternal.class, new LocalService());
}
```

这里主要是调用了 nativeInit 方法进行一系列的初始化操作，nativeInit 方法的实现在 native 层，在其中构造 NativeInputManager 对象：

```
static jlong nativeInit(JNIEnv* env, jclass clazz, jobject serviceObj, jobject contextObj,
jobject messageQueueObj) {
    // ......省去几行代码......

    NativeInputManager* im = new NativeInputManager(contextObj, serviceObj,
    messageQueue->getLooper());

    // ......省去两行代码......
}
```

在 NativeInputManager 的逻辑中，首先会创建一个 EventHub 来监听所有的事件输入：

```
NativeInputManager::NativeInputManager(jobject contextObj, jobject serviceObj, const
sp<Looper>& looper) : mLooper(looper), mInteractive(true) {
    // ......省去几行代码......

    sp<EventHub> eventHub = new EventHub();
```

```
    mInputManager = new InputManager(eventHub, this, this);
}
```

之后就会相继初始化 InputManager、InputDispatcher、InputReader、InputReaderThread 和 InputDispatcherThread。其中 InputReaderThread 和 InputDispatcherThread 是由 InputManager 管理的两个重要线程，其作用和具体的逻辑均与 InputReader 和 InputDispatcher 相关。这一系列的初始化操作完毕后，InputManagerService 就会在 SystemServer 里调用其 start 方法开始运作，其中最为主要的是，使上面提到的两个线程 InputReaderThread 和 InputDispatcherThread 运作。InputReaderThread 启动后会进入死循环，每次循环都会调用 loopOnce 函数，第一次循环时会扫描/dev/input/设备节点目录，并通过 ioctl 函数获取设备信息及判断其类型，当设备发生改变后则通过 NativeInputManager 的回调函数通知处于 Java 层的 InputManagerService 做出相应的处理。如上所述，InputReaderThread 只负责从设备端读取事件信息，每一个事件信息在 InputReaderThread 里最终都会被封装为一个 NotifyArgs 对象，通过 NotifyArgs 中的 Notify 方法调用到 InputDispatcher 相应的 Notify 接口，并由 InputDispatcher 来完成事件的分派。这部分内容比较烦琐，限于篇幅这里不再赘述，有兴趣的读者可以参考“Android 事件分发完全解析之事件从何而来”这篇文章（网址为 http://blog.csdn.net/aigestudio/article/details/44746625）。

11.8 命令模式实战

命令模式在 GUI 上应用广泛，特别是菜单之类的设计，比如执行某项命令，并为该命令提供撤销或重做等功能，这里就通过命令模式在 Android 上实现一个简单的绘图板，该绘图板功能比较简单，只能以触点轨迹绘制线条。首先我们声明一个抽象接口 Ibrush，用它定义不同笔触需要实现的方法。

程序清单 11-15　抽象笔触

```
public interface IBrush {
    /**
     * 触点接触时
     *
     * @param path 路径对象
     * @param x    当前位置的 x 坐标
     * @param y    当前位置的 y 坐标
     */
    void down(Path path, float x, float y);

    /**
     * 触点移动时
     *
     * @param path 路径对象
     * @param x    当前位置的 x 坐标
     * @param y    当前位置的 y 坐标
     */
    void move(Path path, float x, float y);

    /**
     * 触点离开时
     *
     * @param path 路径对象
```

```
     * @param x    当前位置的 x 坐标
     * @param y    当前位置的 y 坐标
     */
    void up(Path path, float x, float y);
}
```

为了简便起见，这里只定义两种类型的笔触，一种为普通的线条，另一种为由圆点组成的线条轨迹。

程序清单 11-16　普通笔触

```
public class NormalBrush implements IBrush {
    @Override
    public void down(Path path, float x, float y) {
        path.moveTo(x, y);
    }

    @Override
    public void move(Path path, float x, float y) {
        path.lineTo(x, y);
    }

    @Override
    public void up(Path path, float x, float y) {

    }
}
```

程序清单 11-17　圆形笔触

```
public class CircleBrush implements IBrush {
    @Override
    public void down(Path path, float x, float y) {

    }

    @Override
    public void move(Path path, float x, float y) {
        path.addCircle(x, y, 10, Path.Direction.CW);
    }

    @Override
    public void up(Path path, float x, float y) {

    }
}
```

对于每一次路径的绘制，都可以有两个命令，一个是绘制命令，另一个是撤销命令，我们将其封装为一个命令接口，注意，这里结合命令模式去构思。

程序清单 11-18　绘制命令接口

```
public interface IDraw {
    /**
     * 绘制命令
     *
     * @param canvas 画布对象
     */
    void draw(Canvas canvas);

    /**
```

```
     * 撤销命令
     */
    void undo();
}
```

这里我们只有绘制路径一种方法。

程序清单 11-19　具体绘制路径命令

```
public class DrawPath implements IDraw {
    public Path path;// 需要绘制的路径
    public Paint paint;// 绘制画笔

    @Override
    public void draw(Canvas canvas) {
        canvas.drawPath(path, paint);
    }

    @Override
    public void undo() {

    }
}
```

虽然只有一个绘制路径的命令，但是，我们为其提供了撤销和重做的功能，因此，依然需要一个请求者角色来对命令做进一步封装。

程序清单 11-20　绘制请求封装类

```
public class DrawInvoker {
    // 绘制列表
    private List<DrawPath> drawList = Collections.synchronizedList(new ArrayList<DrawPath>());

    // 重做列表
    private List<DrawPath> redoList = Collections.synchronizedList(new ArrayList<DrawPath>());

    /**
     * 增加一个命令
     *
     * @param command ...
     */
    public void add(DrawPath command) {
        redoList.clear();
        drawList.add(command);
    }

    /**
     * 撤销上一步的命令
     */
    public void undo() {
        if (drawList.size() > 0) {
            DrawPath undo = drawList.get(drawList.size() - 1);
            drawList.remove(drawList.size() - 1);
            undo.undo();
            redoList.add(undo);
        }
    }

    /**
     * 重做上一步撤销的命令
     */
```

```
    public void redo() {
        if (redoList.size() > 0) {
            DrawPath redoCommand = redoList.get(redoList.size() - 1);
            redoList.remove(redoList.size() - 1);
            drawList.add(redoCommand);
        }
    }

    /**
     * 执行命令
     *
     * @param canvas ...
     */
    public void execute(Canvas canvas) {
        if (drawList != null) {
            for (DrawPath tmp : drawList) {
                tmp.draw(canvas);
            }
        }
    }

    /**
     * 是否可以重做
     *
     * @return ...
     */
    public boolean canRedo() {
        return redoList.size() > 0;
    }

    /**
     * 是否可以撤销
     *
     * @return ...
     */
    public boolean canUndo() {
        return drawList.size() > 0;
    }
}
```

然后往哪儿绘制我们需要一个具体的接收者，这里承担这个重任的是一个 SurfaceView 对象，所有的绘制操作都在这个 SurfaceView 对象中进行。

程序清单 11-21　绘制的真正执行者画布

```
public class DrawCanvas extends SurfaceView implements SurfaceHolder.Callback {
    public boolean isDrawing, isRunning;// 标识是否可以绘制、绘制线程是否可以运行

    private Bitmap mBitmap;// 绘制到的位图对象
    private DrawInvoker mInvoker;// 绘制命令请求对象
    private DrawThread mThread;// 绘制线程

    public DrawCanvas(Context context, AttributeSet attrs) {
        super(context, attrs);

        mInvoker = new DrawInvoker();
        mThread = new DrawThread();

        getHolder().addCallback(this);
    }
```

```
    /**
     * 增减一条绘制路径
     *
     * @param path ...
     */
    public void add(DrawPath path) {
        mInvoker.add(path);
    }

    /**
     * 重做上一步撤销的绘制
     */
    public void redo() {
        isDrawing = true;
        mInvoker.redo();
    }

    /**
     * 撤销上一步的绘制
     */
    public void undo() {
        isDrawing = true;
        mInvoker.undo();
    }

    /**
     * 是否可以撤销
     *
     * @return ...
     */
    public boolean canUndo() {
        return mInvoker.canUndo();
    }

    /**
     * 是否可以重做
     *
     * @return ...
     */
    public boolean canRedo() {
        return mInvoker.canRedo();
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        isRunning = true;
        mThread.start();
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder,int format,int width,int height) {
        mBitmap = Bitmap.createBitmap(width, height, Bitmap.Config.RGB_565);
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        boolean retry = true;
        isRunning = false;
        while (retry) {
            try {
                mThread.join();
                retry = false;
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    private class DrawThread extends Thread {
        @Override
        public void run() {
            Canvas canvas = null;
            while (isRunning) {
                if (isDrawing) {
                    try {
                        canvas = getHolder().lockCanvas(null);
                        if (mBitmap == null) {
                            mBitmap = Bitmap.createBitmap(1, 1, Bitmap.Config.ARGB_8888);
                        }
                        Canvas c = new Canvas(mBitmap);
                        c.drawColor(0, PorterDuff.Mode.CLEAR);

                        canvas.drawColor(0, PorterDuff.Mode.CLEAR);
                        mInvoker.execute(c);
                        canvas.drawBitmap(mBitmap, 0, 0, null);
                    } finally {
                        getHolder().unlockCanvasAndPost(canvas);
                    }
                    isDrawing = false;
                }

            }
        }
    }
}
```

最后，在 Activity 中整合各个功能模块。

程序清单 11-22　Activity 类

```
public class DrawActivity extends Activity {
    private DrawCanvas mCanvas;// 绘制画布
    private DrawPath mPath;// 路径绘制命令
    private Paint mPaint;// 画笔对象
    private IBrush mBrush;// 笔触对象

    private Button btnRedo, btnUndo;// 重做、撤销按钮

    /**
     * 点击事件
     * @param view ...
     */
    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.ac_draw_color_red_btn:// 切换为红色
                mPaint = new Paint();
                mPaint.setStrokeWidth(3);
                mPaint.setColor(0xFFFF0000);
                break;
            case R.id.ac_draw_color_green_btn:// 切换为绿色
                mPaint = new Paint();
                mPaint.setStrokeWidth(3);
                mPaint.setColor(0xFF00FF00);
                break;
```

```
            case R.id.ac_draw_color_blue_btn:// 切换为蓝色
                mPaint = new Paint();
                mPaint.setStrokeWidth(3);
                mPaint.setColor(0xFF0000FF);
                break;
            case R.id.ac_draw_operate_undo_btn:// 撤销操作
                mCanvas.undo();
                if (!mCanvas.canUndo()) {
                    btnUndo.setEnabled(false);
                }
                btnRedo.setEnabled(true);
                break;
            case R.id.ac_draw_operate_redo_btn:// 重做操作
                mCanvas.redo();
                if (!mCanvas.canRedo()) {
                    btnRedo.setEnabled(false);
                }
                btnUndo.setEnabled(true);
                break;
            case R.id.ac_draw_brush_circle_btn:// 切换为圆形笔触
                mBrush = new CircleBrush();
                break;
            case R.id.ac_draw_brush_normal_btn:// 切换为正常笔触
                mBrush = new NormalBrush();
                break;
        }
    }

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

        mPaint = new Paint();
        mPaint.setColor(0xFFFFFFFF);
        mPaint.setStrokeWidth(3);

        mBrush = new NormalBrush();

        mCanvas = (DrawCanvas) findViewById(R.id.ac_draw_canvas);
        mCanvas.setOnTouchListener(new DrawTouchListener());

        btnRedo = (Button) findViewById(R.id.ac_draw_operate_redo_btn);
        btnRedo.setEnabled(false);
        btnUndo = (Button) findViewById(R.id.ac_draw_operate_undo_btn);
        btnUndo.setEnabled(false);
    }

    private class DrawTouchListener implements View.OnTouchListener {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            if (event.getAction() == MotionEvent.ACTION_DOWN) {
                mPath = new DrawPath();
                mPath.paint = mPaint;
                mPath.path = new Path();
                mBrush.down(mPath.path, event.getX(), event.getY());
            } else if (event.getAction() == MotionEvent.ACTION_MOVE) {
                mBrush.move(mPath.path, event.getX(), event.getY());
            } else if (event.getAction() == MotionEvent.ACTION_UP) {
                mBrush.up(mPath.path, event.getX(), event.getY());
                mCanvas.add(mPath);
                mCanvas.isDrawing = true;
```

```
                btnUndo.setEnabled(true);
                btnRedo.setEnabled(false);
            }
            return true;
        }
    }
}
```

为了增加趣味性，这里我们增加对画笔颜色切换的功能，最终的运行效果如图 11-2 所示。

▲图 11-2

大家可以自行尝试，这里不再多说。命令模式的应用并不局限于 GUI，可以毫不夸张地说，你可以在任何地方应用命令模式，如上面 Activity 的 onClick 方法中我们使用到了分支语句来处理不同按钮的点击事件，当然，这里每个事件的逻辑都很简单，如果事件逻辑都很复杂，如一个撤销功能有几百行或上千行代码，这时候就可以考虑使用命令模式来将整个分支语句进行拆分，一个命令对应一条分支简化逻辑，增强代码的可读性。

11.9 总结

在命令模式中其充分体现了几乎所有设计模式的通病，就是类的膨胀，大量衍生类的创建，这是一个不可避免的问题，但是，其给我们带来的好处也非常多，更弱的耦合性、更灵活的控制性以及更好的扩展性，不过，在实际开发过程中是不是需要采用命令模式还是需要斟酌。

第 12 章　解决、解耦的钥匙——观察者模式

12.1 观察者模式介绍

观察者模式是一个使用率非常高的模式，它最常用的地方是 GUI 系统、订阅——发布系统。因为这个模式的一个重要作用就是解耦，将被观察者和观察者解耦，使得它们之间的依赖性更小，甚至做到毫无依赖。以 GUI 系统来说，应用的 UI 具有易变性，尤其是前期随着业务的改变或者产品的需求修改，应用界面也会经常性变化，但是业务逻辑基本变化不大，此时，GUI 系统需要一套机制来应对这种情况，使得 UI 层与具体的业务逻辑解耦，观察者模式此时就派上用场了。

12.2 观察者模式的定义

定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

12.3 观察者模式的使用场景

- 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。

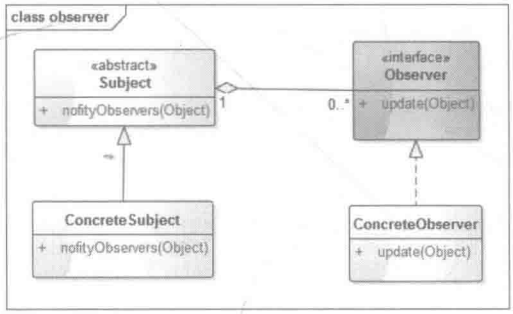
12.4 观察者模式的 UML 类图

UML 类图如图 12-1 所示。

角色介绍。

- Subject：抽象主题，也就是被观察（Observable）的角色，抽象主题角色把所有观察者对象的引用保存在一个集合里，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，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出通知，具体主题角色又叫做具体被观察者（Concrete Observable）角色。



▲图 12-1

- Observer：抽象观察者，该角色是观察者的抽象类，它定义了一个更新接口，使得在得到主题的更改通知时更新自己。
- ConcreteObserver：具体的观察者，该角色实现抽象观察者角色所定义的更新接口，以便在主题的状态发生变化时更新自身的状态。

12.5 观察者模式的简单实现

开发技术前线网站（www.devtf.cn）是一个聚合了关于 Android、iOS 新技术文章的开源库等内容的网站，在这里可以看到新的技术、强大的开源库、新的信息，经常浏览这类网站不仅能够开阔眼界，也能让我们接触到前言的科技信息。

开发技术前线不仅是个内容发布页面，也支持用户邮箱订阅，每周发布周报后会将优质的内容推送给订阅用户，这种模式叫做发布——订阅模式，它的另一个名称是观察者模式，如图 12-2 就是订阅开发技术前线之后收到的周报。

▲图 12-2

下面让我们来简单模拟一下开发技术前线的发布——订阅过程：

```
/**
 * 程序员是观察者
*/
public class Coder implements Observer {
    public String name ;

    public Coder(String aName) {
        name = aName ;
    }

    @Override
    public void update(Observable o, Object arg) {
        System.out.println( "Hi, " +  name + ", DevTechFrontier 更新啦, 内容 : " + arg);
    }

    @Override
    public String toString() {
        return "码农 : " + name;
    }

}

/**
 * DevTechFrontier 即开发技术前线， 这个网站是被观察者角色,当它有更新时所有的观察者 (这里是程序员)
都会接到相应的通知
 */
public class DevTechFrontier extends Observable {

    public void postNewPublication(String content) {
        // 标识状态或者内容发生改变
        setChanged();
        // 通知所有观察者
        notifyObservers(content);
    }
}

// 测试代码
public class Test {
    public static void main(String[] args) {
        // 被观察的角色
        DevTechFrontier DevTechFrontier = new DevTechFrontier();
        // 观察者
        Coder mrsimple = new Coder("mr.simple");
        Coder coder1 = new Coder("coder-1");
        Coder coder2 = new Coder("coder-2");
        Coder coder3 = new Coder("coder-3");

        // 将观察者注册到可观察对象的观察者列表中
        DevTechFrontier.addObserver(mrsimple);
        DevTechFrontier.addObserver(coder1);
        DevTechFrontier.addObserver(coder2);
        DevTechFrontier.addObserver(coder3);

        // 发布消息
        DevTechFrontier.postNewPublication("新的一期开发技术前线周报发布啦!");
    }
}
```

输出结果：

```
Hi, coder-3, DevTechFrontier 更新啦, 内容 : 新的一期开发技术前线周报发布啦!
Hi, coder-2, DevTechFrontier 更新啦, 内容 : 新的一期开发技术前线周报发布啦!
Hi, coder-1, DevTechFrontier 更新啦, 内容 : 新的一期开发技术前线周报发布啦!
Hi, mr.simple, DevTechFrontier 更新啦, 内容 : 新的一期开发技术前线周报发布啦!
```

可以看到所有订阅了开发技术前线的用户都收到了更新消息，一对多的订阅——发布系统就完成了。

Observer 和 Observable 是 JDK 中的内置类型，可见观察者模式是非常重要的，这里 Observer 是抽象的观察者角色，Coder 扮演的是具体观察者的角色；Observable 对应的是抽象主题角色，DevTechFrontier 则是具体的主题角色。Coder 是具体的观察者，它们订阅了 DevTechFrontier 这个具体的可观察对象，当 DevTechFrontier 有更新时，会遍历所有观察者（这里是 Coder），然后给这些观察者发布一个更新的消息，即调用 Coder 中的 update 方法，这样就达到了一对多的通知功能。在这个过程中，通知系统都是依赖 Observer 和 Observable 这些抽象类，因此，对于 Coder 和 DevTechFrontier 完全没有耦合，保证了订阅系统的灵活性、可扩展性。

12.6 Android 源码分析

ListView 是 Android 中最重要的控件之一，而 ListView 最重要的一个功能就是 Adapter，后面我们会分析 Adapter 模式。通常，在我们往 ListView 添加数据后，都会调用 Adapter 的 notifyDataSetChanged()方法，这是为什么呢？今天就来揭开它的神秘面纱。

第一步我们就跟进这个方法 notifyDataSetChanged，这个方法定义在 BaseAdapter 中，具体代码如下：

```
public abstract class BaseAdapter implements
ListAdapter, SpinnerAdapter {
    // 数据集观察者
    private final DataSetObservable mDataSetObservable = new DataSetObservable();

    // 代码省略

    public void registerDataSetObserver(DataSetObserver observer) {
        mDataSetObservable.registerObserver(observer);
    }

    public void unregisterDataSetObserver(DataSetObserver observer) {
        mDataSetObservable.unregisterObserver(observer);
    }

    /**
     * 当数据集变化时，通知所有观察者
     */
    public void notifyDataSetChanged() {
        mDataSetObservable.notifyChanged();
    }
}
```

一看 BaseAdapter 代码就大体有了这么一个认识：BaseAdapter 是一个观察者模式！那么 BaseAdapter 是如何运作的？这些观察者又是什么呢？我们一步一步来分析。

我们先到 mDataSetObservable.notifyChanged()函数中看看：

```
/**
 *  数据集观察者
 */
public class DataSetObservable extends Observable<DataSetObserver> {
    /**
     * 调用每个观察者的 onChaned 函数来通知它们被观察者发生了变化
     */
    public void notifyChanged() {
        synchronized(mObservers) {
            // 调用所有观察者的 onChanged 方式
            for (int i = mObservers.size() - 1; i >= 0; i--) {
                mObservers.get(i).onChanged();
            }
        }
    }
    // 代码省略
}
```

这个代码很简单，就是在 mDataSetObservable.notifyChanged()中遍历所有观察者，并且调用它们的 onChanged 方法，从而告知观察者发生了变化。

那么这些观察者是从哪里来的呢？其实这些观察者就是 ListView 通过 setAdapter 方法设置 Adapter 产生的，我们看看相关代码：

```
    @Override
    public void setAdapter(ListAdapter adapter) {
        // 如果已经有了一个 Adapter,那么先注销该 Adapter 对应的观察者
        if (mAdapter != null && mDataSetObserver != null) {
            mAdapter.unregisterDataSetObserver(mDataSetObserver);
        }

        // 代码省略
        super.setAdapter(adapter);

        if (mAdapter != null) {
            mAareAllItemsSelectable = mAdapter.areAllItemsEnabled();
            mOldItemCount = mItemCount;
            // 获取数据的数量
            mItemCount = mAdapter.getCount();
            checkFocus();
            // 注意这里 : 创建一个一个数据集观察者
            mDataSetObserver = new AdapterDataSetObserver();
            // 将这个观察者注册到 Adapter 中，实际上是注册到 DataSetObservable 中
            mAdapter.registerDataSetObserver(mDataSetObserver);
            // 代码省略
        } else {
            // 代码省略
        }

        requestLayout();
}
```

从程序中可以看到，在设置 Adapter 时会构建一个 AdapterDataSetObserver，这就是上面所说的观察者，最后，将这个观察者注册到 Adapter 中，这样我们的被观察者、观察者都有了。这个时候可能你就有点不明白，AdapterDataSetObserver 是什么？它是如何运作的？那么就先来看看 AdapterDataSetObserver，AdapterDataSetObserver 定义在 ListView 的父类 AbsListView 中，具体代码如下：

```
class AdapterDataSetObserver extends AdapterView<ListAdapter>.AdapterDataSetObserver
{
    @Override
    public void onChanged() {
        super.onChanged();
        if (mFastScroller != null) {
            mFastScroller.onSectionsChanged();
        }
    }

    @Override
    public void onInvalidated() {
        super.onInvalidated();
        if (mFastScroller != null) {
            mFastScroller.onSectionsChanged();
        }
    }
}
```

它又继承自 AbsListView 的父类 AdapterView 的 AdapterDataSetObserver，具体代码如下。

```
class AdapterDataSetObserver extends DataSetObserver {

    private Parcelable mInstanceState = null;
    // 上文讲过，调用 Adapter 的 notifyDataSetChanged 时会调用所有观察者的 onChanged 方法,核心
    // 实现就在这里
    @Override
    public void onChanged() {
        mDataChanged = true;
        mOldItemCount = mItemCount;
        // 获取 Adapter 中数据的数量
        mItemCount = getAdapter().getCount();
        if (AdapterView.this.getAdapter().hasStableIds() && mInstanceState != null
        && mOldItemCount == 0 && mItemCount > 0) {
            AdapterView.this.onRestoreInstanceState(mInstanceState);
            mInstanceState = null;
        } else {
            rememberSyncState();
        }
        checkFocus();
        // 重新布局 ListView、GridView 等 AdapterView 组件
        requestLayout();
    }

    // 代码省略

    public void clearSavedState() {
        mInstanceState = null;
    }
}
```

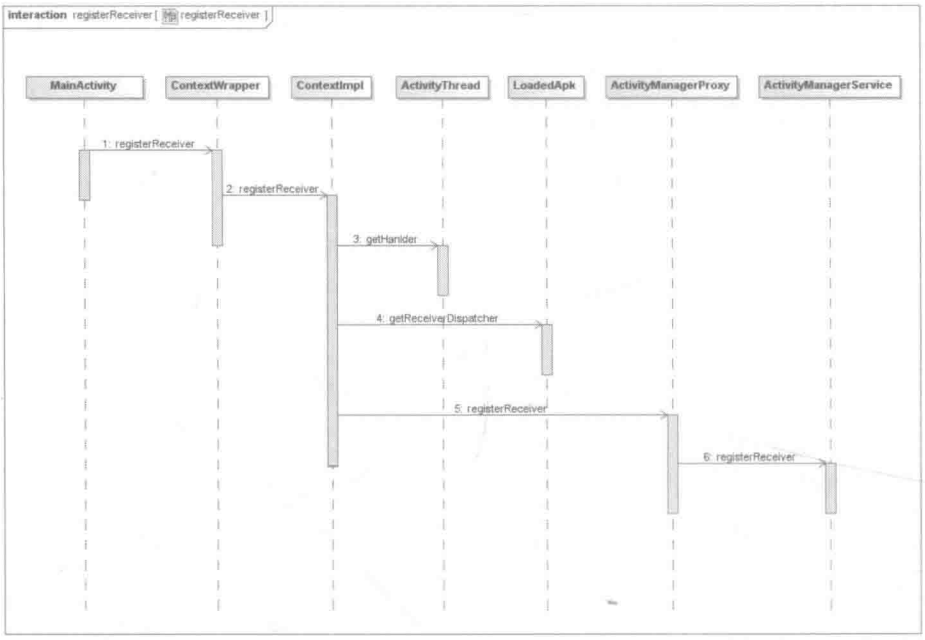
到这里我们就知道了，当 ListView 的数据发生变化时，调用 Adapter 的 notifyDataSetChanged 函数，这个函数又会调用 DataSetObservable 的 notifyChanged 函数，这个函数会调用所有观察者（AdapterDataSetObserver）的 onChanged 方法，在 onChanged 函数中又会调用 ListView 重新布局的函数使得 ListView 刷新界面，这就是一个观察者模式！

最后，我们再整理一下这个过程，AdapterView 中有一个内部类 AdapterDataSetObserver，在 ListView 中设置 Adapter 时会构建一个 AdapterDataSetObserver，并且注册到 Adapter 中，这就是一

个观察者。而 Adapter 中包含一个数据集可观察者 DataSetObservable，在数据数量发生变更时，开发者手动调用 Adapter.notifyDataSetChanged，而 notifyDataSetChanged 实际上会调用 DataSetObservable 的 notifyChanged 函数，该函数会遍历所有观察者的 onChanged 函数。在 AdapterDataSetObserver 的 onChanged 函数中会获取 Adapter 中数据集的新数量，然后调用 ListView 的 requestLayout()方法重新进行布局，更新用户界面。

12.7 观察者模式的深入拓展

BroadcastReceiver 是 Android 的四大组件之一，它作为应用内、进程间的一种重要通信手段，能够将某个消息通过广播的形式传递给它注册的对应广播接收器的对象，接收对象需要通过 Context 的 registerReceiver 函数注册到 AMS 中，当通过 sendBroadcast 发送广播时，所有注册了对应的 IntentFilter 的 BroadcastReceiver 对象就会接收到这个消息，BroadcastReceiver 的 onReceive 方法就会被调用，这就是一个典型的发布——订阅模式，也就是我们的观察者模式。广播接收器的注册调用时序图如图 12-3 所示。



▲图 12-3

例如，我们在 MainActivity 中注册一个广播接收器，示例代码如下：

```
public class MainActivity extends Activity {
    @Override
    public void onResume() {
        super.onResume();
        IntentFilter updateFilter = new IntentFilter("info.update");
```

```
        registerReceiver(updateReceiver, updateFilter);
    }
    BroadcastReceiver updateReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
        Toast.makeText(getApplicationContext(), "update", Toast.LENGTH_SHORT).show();
      }
    };
    // 代码省略
}
```

我们在 MainActivity 的 onResume 里注册了一个只接收 Action 为“info.update”的广播接收器，应用内的其他地方发布一个 Action 为“info.update”的广播时，就会触发 updateReceiver 的 onReceive 函数。下面我们就来一步一步分析广播接收器的基本原理。

我们发现 registerReceiver 函数并不是在 Activity 中实现，因此，我们把目标移向 Activity 的父类 ContextWrapper，registerReceiver 函数如下：

```
public class ContextWrapper extends Context {
    Context mBase;
    // 代码省略

    @Override
    public Intent registerReceiver(
        BroadcastReceiver receiver, IntentFilter filter) {
        return mBase.registerReceiver(receiver, filter);
    }

    // 代码省略

}
```

这里的成员变量 mBase，在本书的第 1 章中我们讲过这个 mBase 对象是一个 ContextImpl 类的实例。继续转移到 ContextImpl 的 registerReceiver 函数，具体代码如下：

```
class ContextImpl extends Context {
    // 代码省略

    @Override
    public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter) {
        return registerReceiver(receiver, filter, null, null);
    }

    @Override
    public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter,
            String broadcastPermission, Handler scheduler) {
               return registerReceiverInternal(receiver, filter, broadcastPermission,
               scheduler, getOuterContext());
    }

    private Intent registerReceiverInternal(BroadcastReceiver receiver,
            IntentFilter filter, String broadcastPermission,
            Handler scheduler, Context context) {
               IIntentReceiver rd = null;
        if (receiver != null) {
            if (mPackageInfo != null && context != null) {
                if (scheduler == null) {
                    scheduler = mMainThread.getHandler();    // 获取 Handler 来投递消息
                }
        // 获取 IIntentReceiver 对象，通过它与 AMS 交互，并且通过 Handler 传递消息
```

```
                rd = mPackageInfo.getReceiverDispatcher(
                    receiver, context, scheduler,
                    mMainThread.getInstrumentation(), true);
            } else {
                // 代码省略
            }
        }
        try {
    // 调用 ActivityManagerNative 的 registerReceiver
            return ActivityManagerNative.getDefault().registerReceiver(
                    mMainThread.getApplicationThread(),
                    rd, filter, broadcastPermission);
        } catch (RemoteException e) {
                return null;
        }
    }

    // 代码省略

}
```

注册广播接收器的函数调用最终进入到了 ContextImpl 的 registerReceiverInternal 这个函数，这里的成员变量 mPackageInfo 是一个 LoadedApk 实例，它是用来负责处理广播的接收。参数 broadcastPermission 和 scheduler 都为 null，而参数 context 是上面的函数通过调用函数 getOuterContext 得到的，这里它指向的是 MainActivity。

由于条件 mPackageInfo != null 和 context != null 都成立，而且条件 scheduler == null 也成立，于是就调用 mMainThread.getHandler 来获得一个 Handler，这个 Hanlder 是用来分发 ActivityManagerService 发送过的广播，这里的成员变量 mMainThread 是一个 ActivityThread 实例，我们先来看看 ActivityThread.getHandler 函数的实现，然后再回过头来继续分析 ContextImpl.registerReceiverInternal 函数：

```
public final class ActivityThread {
    final H mH = new H();
    private final class H extends Handler {
        public void handleMessage(Message msg) {
            switch (msg.what) {
            // 代码省略
            }
        }
    // 代码省略
    }

    final Handler getHandler() {
        return mH;
    }
  // 代码省略
}
```

有了这个 Handler 之后，就可以分发消息给应用程序处理了。

再回到上一步的 ContextImpl.registerReceiverInternal 函数中，它通过 mPackageInfo.getReceiverDispatcher 函数获得一个 IIntentReceiver 接口对象 rd，这是一个 Binder 对象，接下来会把它传给 ActivityManagerService，ActivityManagerService 在收到相应的广播时，就是通过这个 Binder 对象来通知 MainActivity 来接收的。

我们也是先来看一下 mPackageInfo.getReceiverDispatcher 函数的实现：

```
final class LoadedApk {
 // 代码省略

    public IIntentReceiver getReceiverDispatcher(BroadcastReceiver r,
    Context context, Handler handler, Instrumentation instrumentation, boolean registered) {
        synchronized (mReceivers) {
            LoadedApk.ReceiverDispatcher rd = null;
            HashMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher> map = null;
            if (registered) {
                map = mReceivers.get(context);
                if (map != null) {
                    rd = map.get(r);
                }
            }
            if (rd == null) {
                rd = new ReceiverDispatcher(r, context, handler,
                    instrumentation, registered);
                if (registered) {
                    if (map == null) {
                     // BroadcastReceiver 为 key, ReceiverDispatcher 为 value
                       map = new HashMap<BroadcastReceiver, LoadedApk.
                       ReceiverDispatcher>();
                           // 以 context 为 key,map 为 value 存储到 mReceivers 中
                        mReceivers.put(context, map);
                    }
                    map.put(r, rd);
                }
            } else {
                rd.validate(context, handler);
            }
            return rd.getIIntentReceiver();
        }
    }

    // 代码省略

    static final class ReceiverDispatcher {

        final static class InnerReceiver extends IIntentReceiver.Stub {
            final WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
            // 代码省略

            InnerReceiver(LoadedApk.ReceiverDispatcher rd, boolean strong) {
                mDispatcher = new WeakReference<LoadedApk.ReceiverDispatcher>(rd);
                // 代码省略
            }
            // 代码省略
        }

        // 代码省略

        final IIntentReceiver.Stub mIIntentReceiver;
        final Handler mActivityThread;

        // 代码省略

        ReceiverDispatcher(BroadcastReceiver receiver, Context context,
        Handler activityThread, Instrumentation instrumentation,
                boolean registered) {
```

```
            // 代码省略

            mIIntentReceiver = new InnerReceiver(this, !registered);
            mActivityThread = activityThread;

            // 代码省略
        }

        // 代码省略

        IIntentReceiver getIIntentReceiver() {
            return mIIntentReceiver;
        }

    }

    // 代码省略

}
```

在 LoadedApk 类的 getReceiverDispatcher 函数中，首先看一下参数 r 是不是已经有相应的 ReceiverDispatcher 存在了，如果有，就直接返回；否则就新创建一个 ReceiverDispatcher，并且以 r 为 Key 值保存在一个 HashMap 中，而这个 HashMap 又被存储在 mReceivers 这个 map 中，mReceivers 以 Context 为 Key、map 为 value。即这里的 MainActivity 为 Key 保存在 LoadedApk 的成员变量 mReceivers 中，这样，只要给定一个 Activity 和 BroadcastReceiver，就可以查看 LoadedApk 里面是否已经存在相应的广播接收发布器 ReceiverDispatcher 了。

在新创建广播接收发布器 ReceiverDispatcher 时，会在构造函数里创建一个 InnerReceiver 实例，这是一个 Binder 对象，实现了 IIntentReceiver 接口，可以通过 ReceiverDispatcher.getIIntentReceiver 函数来获得，获得后就会把它传给 ActivityManagerService，以便接收广播。在 ReceiverDispatcher 类的构造函数中，还会把传进来的 Handle 类型的参数 activityThread 保存下来，以便后面在分发广播的时候使用。

现在，再回到 ContextImpl.registerReceiverInternal 函数，在获得了 IIntentReceiver 类型的 Binder 对象后，就开始把它注册到 ActivityManagerService 中，具体代码如下：

```
class ActivityManagerProxy implements IActivityManager
{
    // 代码省略

    public Intent registerReceiver(IApplicationThread caller,
            IIntentReceiver receiver,
            IntentFilter filter, String perm) throws RemoteException
    {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
        data.writeInterfaceToken(IActivityManager.descriptor);
        data.writeStrongBinder(caller != null ? caller.asBinder() : null);
        data.writeStrongBinder(receiver != null ? receiver.asBinder() : null);
        filter.writeToParcel(data, 0);
        data.writeString(perm);
        // 向 AMS 提交注册广播接收器的请求
        mRemote.transact(REGISTER_RECEIVER_TRANSACTION, data, reply, 0);
        reply.readException();
        Intent intent = null;
```

```
        int haveIntent = reply.readInt();
        if (haveIntent != 0) {
            intent = Intent.CREATOR.createFromParcel(reply);
        }
        reply.recycle();
        data.recycle();
        return intent;
    }

    // 代码省略

}
```

这个函数通过Binder驱动程序就进入到ActivityManagerService中的registerReceiver函数中去了。

```
public final class ActivityManagerService extends ActivityManagerNative
        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
    // 代码省略

    public Intent registerReceiver(IApplicationThread caller,
    IIntentReceiver receiver, IntentFilter filter, String permission) {
        synchronized(this) {
            // 1. 获取  ProcessRecord
            ProcessRecord callerApp = null;
            if (caller != null) {
                callerApp = getRecordForAppLocked(caller);
                if (callerApp == null) {
                    // 代码省略
                }
            }

            List allSticky = null;

            // 2. 根据Action查找匹配的sticky 接收器
            Iterator actions = filter.actionsIterator();
            if (actions != null) {
                while (actions.hasNext()) {
                    String action = (String)actions.next();
                    allSticky = getStickiesLocked(action, filter, allSticky);
                }
            } else {
                // 代码省略
            }
            Intent sticky = allSticky != null ? (Intent)allSticky.get(0) : null;
            // 代码省略
            if (receiver == null) {
                return sticky;
            }
               // 3. 获取ReceiverList
            ReceiverList rl
                = (ReceiverList)mRegisteredReceivers.get(receiver.asBinder());
            if (rl == null) {
                rl = new ReceiverList(this, callerApp, Binder.getCallingPid(),
                Binder.getCallingUid(), receiver);

                if (rl.app != null) {
                    rl.app.receivers.add(rl);
                } else {
                    // 代码省略
                }
                mRegisteredReceivers.put(receiver.asBinder(), rl);
```

```
            }
        // 4. 构建 BroadcastFilter 对象，并且添加到 ReceiverList 中
            BroadcastFilter bf = new BroadcastFilter(filter, rl, permission);
            rl.add(bf);
            // 代码省略
            mReceiverResolver.addFilter(bf);
                // 代码省略
            return sticky;
        }
    }
    // 代码省略
}
```

函数首先是获得调用 registerReceiver 函数的应用程序进程记录块，即注释 1 处，然后获取这个进程对应的 pid、uid。注释 2 处获取到 IntentFilter 的所有 Action，在 MainActivity 中构建的 filter 只有一个 Action，就是前面描述的“info.update”，这里先通过 getStickiesLocked 函数查找一下有没有对应的 sticky intent 列表存在。什么是 Sticky Intent 呢？我们在最后一次调用 sendStickyBroadcast 函数来发送某个 Action 类型的广播时，系统会把代表这个广播的 Intent 保存下来，这样，后来调用 registerReceiver 来注册相同 Action 类型的广播接收器，就会得到这个最后发出的广播，这就是为什么叫做 Sticky Intent 了。这个最后发出的广播虽然被处理完了，但是仍然被粘在 ActivityManagerService 中，以便下一个注册相应 Action 类型的广播接收器还能继承处理。

这里，假设不使用 sendStickyBroadcast 来发送“info.update”类型的广播，于是，得到的 allSticky 和 sticky 都为 null。

继续往下看，这里传进来的 receiver 不为 null，于是，继续往下执行到注释 3 处。这里其实就是把广播接收器 receiver 保存到一个 ReceiverList 列表中，这个列表的宿主进程是 rl.app，这里就是 MainActivity 所在的进程，在 ActivityManagerService 中，用一个进程记录块来表示这个应用程序进程，它里面有一个列表 receivers，专门用来保存这个进程注册的广播接收器。接着，又把这个 ReceiverList 列表以 receiver 为 Key 值保存在 ActivityManagerService 的成员变量 mRegisteredReceivers 中，这些都是为了方便在收到广播时，快速找到对应的广播接收器。

再往下看注释 4，只是把广播接收器 receiver 保存起来，但是还没有把它和 filter 关联起来，这里就创建一个 BroadcastFilter 来把广播接收器列表 rl 和 filter 关联起来，然后保存在 ActivityManagerService 的成员变量 mReceiverResolver 中，这样，就将广播接收器 receiver 及其要接收的广播类型 filter 保存在 ActivityManagerService 中，以便以后能够接收到相应的广播并进行处理。

广播的发送过程比广播接收器的注册过程要复杂得多，不过这个过程仍然是以 ActivityManagerService 为中心

这里，我们简单总结一下 Android 应用程序发送广播的过程。

（1）通过 sendBroadcast 把一个广播通过 Binder 发送给 ActivityManagerService，ActivityManagerService 根据这个广播的 Action 类型找到相应的广播接收器，然后把这个广播放进自己的消息队列中，就完成第一阶段对这个广播的异步分发：

```
public final class ActivityManagerService extends ActivityManagerNative
        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {

    private final int broadcastIntentLocked(ProcessRecord callerApp,
            String callerPackage, Intent intent, String resolvedType,
```

```
            IIntentReceiver resultTo, int resultCode, String resultData,
            Bundle map, String requiredPermission,
            boolean ordered, boolean sticky, int callingPid, int callingUid) {
            intent = new Intent(intent);
        // 代码省略
        // Figure out who all will receive this broadcast.
        List receivers = null;
        List<BroadcastFilter> registeredReceivers = null;
        try {
            if (intent.getComponent() != null) {
                // 代码省略
            } else {
                // 查询到该 Intent 对应的 BroadcastFilter,也就是接收器列表
                registeredReceivers = mReceiverResolver.queryIntent(intent, resolvedType, false);
            }
        } catch (RemoteException ex) {
            // 代码省略
        }
            // 代码省略
            if (!replaced) {
                mParallelBroadcasts.add(r);
                scheduleBroadcastsLocked();   // 处理广播的异步分发
            }
            registeredReceivers = null;
            NR = 0;
        }
    }
    // 代码省略
}
```

（2）ActivityManagerService 在消息循环中处理这个广播，并通过 Binder 机制把这个广播分发给注册的 ReceiverDispatcher，ReceiverDispatcher 把这个广播放进 MainActivity 所在线程的消息队列中，就完成第二阶段对这个广播的异步分发：

```
public final class ActivityManagerService extends ActivityManagerNative
        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
    // 代码省略
    private final void scheduleBroadcastsLocked() {
        // 代码省略
        if (mBroadcastsScheduled) {
            return;
        }
        // mHandler 发送一个空消息
        mHandler.sendEmptyMessage(BROADCAST_INTENT_MSG);
        mBroadcastsScheduled = true;
    }
    // 代码省略
// mHandler 的类型为 Handler,是 AMS 的内部类
 final Handler mHandler = new Handler() {
        public void handleMessage(Message msg) {
            switch (msg.what) {
            ......
            case BROADCAST_INTENT_MSG: {
                ......
                processNextBroadcast(true);   // 处理下一个广播
            } break;
            ......
            }
        }
    }
```

```
}
```

（3）ReceiverDispatcher 的内部类 Args 在 MainActivity 所在的线程消息循环中处理这个广播，最终是将这个广播分发给所注册的 BroadcastReceiver 实例的 onReceive 函数进行处理：

```
final class LoadedApk {
    // 代码省略
    static final class ReceiverDispatcher {
        // 代码省略
        final class Args implements Runnable {
            // 代码省略
            public void run() {
                BroadcastReceiver receiver = mReceiver;
                Intent intent = mCurIntent;
                try {
                    ClassLoader cl =  mReceiver.getClass().getClassLoader();
                    intent.setExtrasClassLoader(cl);
                    if (mCurMap != null) {
                        mCurMap.setClassLoader(cl);
                    }
                    receiver.setOrderedHint(true);
                    receiver.setResult(mCurCode, mCurData, mCurMap);
                    receiver.clearAbortBroadcast();
                    receiver.setOrderedHint(mCurOrdered);
                    receiver.setInitialStickyHint(mCurSticky);
                    // 调用接收器的 onReceive 方法
                    receiver.onReceive(mContext, intent);
                } catch (Exception e) {
                    // 代码省略
                }
            }
        }
    }
    // 代码省略
}
```

简单来说，广播这就是一个订阅——发布的过程，通过一些 map 存储 BroadcastReceiver，key 就是封装了这些广播的信息类，如 Action 之类的。当发布一个广播时通过 AMS 到这个 map 中查询注册了这个广播的 IntentFilter 的 BroadcastReceiver，然后通过 ReceiverDispatcher 将广播分发给各个订阅对象，从而完成这个发布——订阅过程。

12.8 实战

小民最近在工作的过程中用到了广播接收器，用广播接收器的原因是小民需要在 Fragment 之间、Activity 与 Fragment 之间进行通信，需要通信就必须有对方的引用，这就导致耦合性较高，而且广播接收器的使用成本相对比较高，小民在给主管反馈了这个问题之后，主管给小民提供了一个新的解决方案：使用事件总线。

如果你不知道事件总线是什么，那么没有关系，下面我们先来看这样一个场景：

你是否在开发的过程中遇到过想在 Activity-B 中回调 Activity-A 中的某个函数，但 Activity 又不能手动创建对象来设置一个 Listener；你是否想在某个 Service 中想更新 Activity 或者 Fragment 中的界面？或者类的组件之间的交互问题……

以 Activity 与 Fragment 之间的通信为例，按照 Android 官方给的思路是，Activity 实现某个接口，然后在 Fragment-A 关联上 Activity 之后，将 Activity 强转为接口类型，在某个时刻 Fragment 中回调这个接口，这个过程是不是有点复杂呢？如果你也这么觉得，那也就是你继续看下去的理由了。

经过思考你会发现 Android 中的 Activity、Fragment、Service 之间的交互是比较麻烦的，可能我们首先想到的是使用广播接收器来在它们之间进行交互，例如，上述所说在 Activity-B 中发送一个广播，在 Activity-A 中注册一个广播接收器来接收该广播，但使用广播接收器稍显麻烦，如果你要将一个实体类当作数据在组件之间传递，那么该实体类还得实现序列化接口，这个成本实在有点高！程序如下所示：

```
// ActivityA 中注册广播接收器
class ActivityA extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        registerReceiver(new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                User person = intent.getParcelableExtra("user") ;
            }
        }, new IntentFilter("my_action")) ;
    }
    // ......
}

// ActivityB 中发布广播
class ActivityB extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Intent intent  = new Intent("my_action") ;
        intent.putExtra("user", new User("mr.simple")) ;
        sendBroadcast(intent);
    }
    // ......
}
// 实体类实现序列化
class User implements Parcelable {
    String name ;
    public User(String aName) {
        name = aName ;
    }
   // 代码省略
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(name);
    }
}
```

从程序中可以看出是很麻烦！

我们再来看一个示例，在开发过程中，我们经常要在子线程中做一些耗时操作，然后将结果更新到 UI 线程，除了 AsyncTask 之外，Thread 加 Handler 是我们经常用的手段，我们看看如下示例：

```
class MyActivity extends Activity {
        Handler mHandler = new Handler () {
            public void handleMessage(android.os.Message msg) {
```

```
                if ( msg.what == 1 ) {
                    User user = (User)msg.obj ;
                    // do sth
                }
            };
        } ;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            // ......

            new Thread(
                new Runnable() {
                    public void run() {
                        // do sth
                        User newUser = new User("simple") ;
                        Message msg = mHandler.obtainMessage() ;
                        msg.what = 1 ;
                        msg.obj = newUser ;
                        mHandler.sendMessage(msg) ;
                    }
            }).start();
        }

    }
```

是不是又是相当麻烦！

此时，你应该冷静下来思考一下，还有哪些情况适合使用事件总线，例如，穿透多个类只为一个回调的 Listener……

事件总线框架就是为了简化这些操作而出现的，并且降低组件之间的耦合，到底如何解决呢？继续往下看。

事件总线就是简化了 Activity、Fragment、Service 等组件之间的交互，很大程度上降低了它们之间的耦合，使得我们的代码更加简洁，耦合性更低，提升代码的质量。

“这是个好东西啊！”小民在使用完之后感叹道，但是，小民没有停留在使用的层面，他决定自己实现一个事件总线库，并且命名为 AndroidEventBus。

小民在分析了已存在的总线库，发现它们或多或少地存在一些问题，并且给 AndroidEventBus 设定了一些目标，如表 12-1 所示。

表 12-1

名　称	订阅函数是否可异步执行	特　点
Greenrobot 的 EventBus	是	使用 name pattern 模式，效率高，但使用不方便
Otto	否	使用注解，使用方便，但效率比不上 EventBus
AndroidEventBus	是	订阅函数支持 tag（类似 BroadcastReceiver 的 Action）使得事件的投递更加准确，能适应更多使用场景。使用方便，但效率比不上 EventBus

小民打算使用注解的形式实现事件总线，这样用户在使用时比较方便，并且让接收函数可以执行在异步线程。当然，这个库的定制化、灵活性也是小民的要求之一。大体思路如下：

在需要接收事件的对象初始化时将自身注册到事件总线中，事件总线会遍历这个对象的所属类中的所有函数，然后根据订阅函数的事件参数和@Subscriber 注解的 tag 值构建一个 EventType，这个 EventType 标识了一种事件类型，以这个 EventType 为 key，订阅对象列表为 value 构建一个 map，存储这整个事件——订阅者表，当通过 EventBus 发布事件时，会根据事件类型和 tag 来构建一个 EventType，再根据这个 EventType 到表中查找所有的订阅者，然后执行它们的订阅函数。如图 12-4 所示。

▲图 12-4

下面我们先看看 Subscriber 注解的定义：

```
/**
 * 事件接收函数的注解类,运用在函数上
*/
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Subscriber {
    /**
     * 事件的 tag,类似于 BroadcastReceiver 中的 Action,事件的标识符
     * @return
     */
    String tag() default EventType.DEFAULT_TAG;

    /**
     * 事件执行的线程,默认为主线程
     * @return
     */
    ThreadMode mode() default ThreadMode.MAIN;
}
```

该注解作用于函数上，并且有一个 tag 和 mode，分别为事件的 tag（类似广播接收器的 Action）和该订阅函数的执行线程。

在将订阅对象注册到事件总线时会找到所有使用了 Subscriber 注解的类，并且以 tag 和事件类型作为 key，订阅函数列表作为 value 存储在 map 中，对应代码如下：

```
public final class EventBus {
 /**
     * 订阅 map
```

```
     */
    private final Map<EventType, CopyOnWriteArrayList<Subscription>> mSubcriberMap = new
    ConcurrentHashMap<EventType, CopyOnWriteArrayList<Subscription>>();
    /**
     * the subscriber method hunter, find all of the subscriber's methods
     * annotated with @Subcriber
     */
    SubsciberMethodHunter mMethodHunter = new SubsciberMethodHunter(mSubcriberMap);


    /**
     * 注册订阅对象
     */
    public void register(Object subscriber) {
        if (subscriber == null) {
            return;
        }

        synchronized (this) {
            mMethodHunter.findSubcribeMethods(subscriber);
        }
    }
}
```

查找订阅方法的工作交给了 SubsciberMethodHunter 类，我们看看相关的函数：

```
public class SubsciberMethodHunter {
    /**
     * the event bus's subscriber's map
     */
    Map<EventType, CopyOnWriteArrayList<Subscription>> mSubcriberMap;

    /**
     * @param subscriberMap
     */
    public SubsciberMethodHunter(Map<EventType, CopyOnWriteArrayList<Subscription>>
    subscriberMap) {
        mSubcriberMap = subscriberMap;
    }

    /**
     * @param subscriber
     * @return
     */
    public void findSubcribeMethods(Object subscriber) {
        if (mSubcriberMap == null) {
            throw new NullPointerException("the mSubcriberMap is null. ");
        }
        Class<?> clazz = subscriber.getClass();
        // 查找类中符合要求的注册方法,直到 Object 类
        while (clazz != null && !isSystemCalss(clazz.getName())) {
            final Method[] allMethods = clazz.getDeclaredMethods();
            for (int i = 0; i < allMethods.length; i++) {
                Method method = allMethods[i];
                // 根据注解来解析函数
                Subscriber annotation = method.getAnnotation(Subscriber.class);
                if (annotation != null) {
                    // 获取方法参数
                    Class<?>[] paramsTypeClass = method.getParameterTypes();
                    // 只支持一个参数的订阅函数
                    if (paramsTypeClass != null && paramsTypeClass.length == 1) {
```

```
                    Class<?> paramType = convertType(paramsTypeClass[0]);
                    EventType eventType = new EventType(paramType, annotation.tag());
                    TargetMethod subscribeMethod = new TargetMethod(method, paramType,
                        annotation.mode());
                    subscibe(eventType, subscribeMethod, subscriber);
                }
            }
        } // end for

        // 获取父类 Class 类型，以便后续查找父类中符合要求的订阅方法
        clazz = clazz.getSuperclass();
    }
}

    /** 将找到的订阅函数添加到订阅表中
     * @param event 事件类型，EventType
     * @param method 目标方法
     * @param subscriber 订阅者
     */
    private void subscibe(EventType event, TargetMethod method, Object subscriber) {
        CopyOnWriteArrayList<Subscription> subscriptionLists = mSubcriberMap.get(event);
        if (subscriptionLists == null) {
            subscriptionLists = new CopyOnWriteArrayList<Subscription>();
        }

        Subscription newSubscription = new Subscription(subscriber, method);
        if (subscriptionLists.contains(newSubscription)) {
            return;
        }

        subscriptionLists.add(newSubscription);
        // 将事件类型 key 和订阅者信息存储到 map 中
        mSubcriberMap.put(event, subscriptionLists);
    }
    }
```

这样就将每个订阅对象的所有订阅函数存储在一个表中了，这个 key 就是 EventType，而 EventType 又由订阅函数的参数类型和 tag 决定，当发布一个事件时，就从表中获得订阅了这个事件的所有订阅函数列表，然后将它们执行在指定的线程。

发布事件的代码如下：

```
public final class EventBus {

    /**
     * the thread local event queue, every single thread has it's own queue.
     */
    ThreadLocal<Queue<EventType>> mLocalEvents = new ThreadLocal<Queue<EventType>>() {
        protected java.util.Queue<EventType> initialValue() {
            return new ConcurrentLinkedQueue<EventType>();
        };
    };
    /**
     * the event dispatcher
     */
    EventDispatcher mDispatcher = new EventDispatcher();

    // 发布事件
    public void post(Object event, String tag) {
        mLocalEvents.get().offer(new EventType(event.getClass(), tag));
```

```
        mDispatcher.dispatchEvents(event);
    }
}
```

可以看到，在发布事件时，首先将事件添加到 mLocalEvents 这个事件队列中，然后通过 EventDispatcher 来对事件进行处理，例如，根据 EventType 找到订阅者列表，根据订阅者列表的线程模型找到对应的事件处理器，最后通过事件处理器执行订阅函数，相关代码如下：

```
public final class EventBus {
    // 事件队列
    ThreadLocal<Queue<EventType>> mLocalEvents = new ThreadLocal<Queue<EventType>>() {
        protected java.util.Queue<EventType> initialValue() {
            return new ConcurrentLinkedQueue<EventType>();
        };
    };
// 代码省略


    // 事件分发器
    private class EventDispatcher {
        / / 将接收方法执行在 UI 线程
        EventHandler mUIThreadEventHandler = new UIThreadEventHandler();
        //哪个线程执行的 post,接收方法就执行在哪个线程
        EventHandler mPostThreadHandler = new DefaultEventHandler();
        // 异步线程中执行订阅方法
        EventHandler mAsyncEventHandler = new AsyncEventHandler();
        // 处理事件
        void dispatchEvents(Object aEvent) {
            Queue<EventType> eventsQueue = mLocalEvents.get();
            while (eventsQueue.size() > 0) {
                deliveryEvent(eventsQueue.poll(), aEvent);
            }
        }
        /**
         * 根据 aEvent 查找到所有匹配的集合,然后处理事件
         * @param type
         * @param aEvent
         */
        private void deliveryEvent(EventType type, Object aEvent) {
            Class<?> eventClass = aEvent.getClass();
            List<EventType> eventTypes = null;
            // 如果有缓存则直接从缓存中取
            if (mCacheEventTypes.containsKey(eventClass)) {
                eventTypes = mCacheEventTypes.get(type);
            } else {
            // 查找订阅者的订阅函数，找到匹配的订阅类型
                eventTypes = mMatchPolicy.findMatchEventTypes(type, aEvent);
                mCacheEventTypes.put(type, eventTypes);
            }
            // 处理事件
            for (EventType eventType : eventTypes) {
                handleEvent(eventType, aEvent);
            }
        }

        /**
         * 处理单个事件
         * @param eventType
         * @param aEvent
         */
```

```
        private void handleEvent(EventType eventType, Object aEvent) {
        // 根据 EventType 找到订阅者
            List<Subscription> subscriptions = mSubcriberMap.get(eventType);
            if (subscriptions == null) {
                return;
            }
            // 将事件分发到所有的订阅者中，也就是执行订阅函数
            for (Subscription subscription : subscriptions) {
                final ThreadMode mode = subscription.threadMode;
                // 根据线程模型来获取对应的事件处理器
                EventHandler eventHandler = getEventHandler(mode);
                // 处理事件
                eventHandler.handleEvent(subscription, aEvent);
            }
        }

        private EventHandler getEventHandler(ThreadMode mode) {
            if (mode == ThreadMode.ASYNC) {
                return mAsyncEventHandler;
            }
            if (mode == ThreadMode.POST) {
                return mPostThreadHandler;
            }
            return mUIThreadEventHandler;
        }
    } // end of EventDispatcher
}
```

发布事件之后会将事件添加到事件列表中，会根据事件类型和 tag 构造 EventType，发布的事件是一个，但是 EventType 可能有多个，例如，用户在注册订阅函数时参数类型为 List<String>，那在用户发布事件时可能参数就会传递 ArrayList<String>类型。因此，在查找订阅函数时会将参数类型是发布事件类型的父类也返回，这样使得在订阅事件参数类型与发布类型之间有父子关系的情况下也能正常匹配到。

找到了对应的 EventType 之后，就会根据这个 EventType 从表中获得所有的订阅者，然后根据这些订阅者的线程模型找到一个对应的事件处理器，最后将这个订阅函数执行到对应的线程中。至此，整个订阅——发布流程完毕。当然，与广播接收器类似，不要忘记在 Activity 等对象的生命周期结束时，调用 EventBus 的 unregister 函数来注销，否则会造成内存泄露！

关于 AndroidEventBus 更详细的信息请参考源码，项目网址为 https://github.com/bboyfeiyu/AndroidEventBus。

12.9 总结

观察者模式主要的作用就是对象解耦，将观察者与被观察者完全隔离，只依赖于 Observer 和 Observable 抽象，例如，ListView 就是运用了 Adapter 和观察者模式使得它的可扩展性、灵活性非常强，而耦合度却很低，这是设计模式在 Android 源码中优秀运用的典范。那么为什么 Android 架构师们会这样设计 ListView，它们如何达到低耦合、高灵活性呢？广播接收器和事件总线在观察者模式上有什么相似之处，又有什么不同？大家可以仔细思考，后续章节我们再继续相关话题。

优点

- 观察者和被观察者之间是抽象耦合，应对业务变化；
- 增强系统灵活性、可扩展性。

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问题，程序中包括一个被观察者、多个观察者、开发和调试等内容会比较复杂，而且在 Java 中消息的通知默认是顺序执行，一个观察者卡顿，会影响整体的执行效率，在这种情况下，一般考虑采用异步的方式。

第 13 章　编程中的“后悔药”——备忘录模式

13.1 备忘录模式介绍

备忘录模式是一种**行为模式**，该模式用于保存对象当前状态，并且在之后可以再次恢复到此状态，这有点像我们平时说的“后悔药”。备忘录模式实现的方式需要保证被保存的对象状态不能被对象从外部访问，目的是为了保护好被保存的这些对象状态的完整性以及内部实现不向外暴露。

13.2 备忘录模式的定义

在不破坏封闭的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样，以后就可将该对象恢复到原先保存的状态。

13.3 备忘录模式的使用场景

（1）需要保存一个对象在某一个时刻的状态或部分状态。

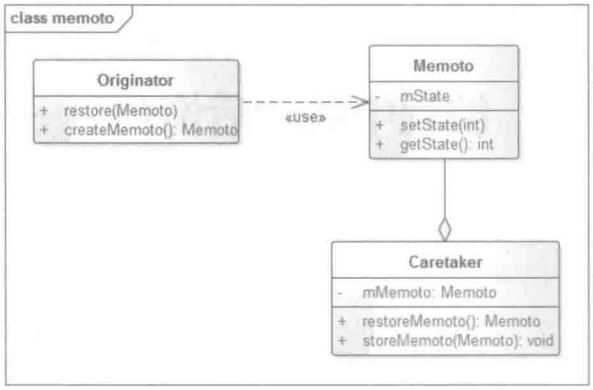
（2）如果用一个接口来让其他对象得到这些状态，将会暴露对象的实现细节并破坏对象的封装性，一个对象不希望外界直接访问其内部状态，通过中间对象可以间接访问其内部状态。

13.4 备忘录模式的 UML 类图

UML 类图如图 13-1 所示。

角色介绍。

- Originator：负责创建一个备忘录，可以记录、恢复自身的内部状态。同时 Originator 还可以根据需要决定 Memento 存储自身的哪些内部状态。
- Memento：备忘录角色，用于存储 Originator 的内部状态，并且可以防止 Originator 以外的对象访问 Memento。
- Caretaker：负责存储备忘录，不能对备忘录的内容进行操作和访问，只能够将备忘录传递给其他对象。



▲图 13-1

13.5 备忘录模式的简单示例

对备忘录模式来说，比较贴切的场景应该是游戏中的存档功能，该功能就是将游戏进度存储到本地文件系统或者数据库中，下次再次进入时从本地加载进度，使得玩家能够继续上一次的游戏之旅，这里我们就以“使命召唤”这款游戏为例来简单演示一下备忘录模式的实现。

首先我们建立游戏类、备忘录类、Caretaker 类，玩游戏到某个节点对游戏进行存档，然后退出游戏，再重新进入时从存档中读取进度，并且进入存档时的进度。

首先看游戏类。

```
/**
 * 使命召唤游戏（数据模型可能不太合理，这里我们只是简单演示）
 */
public class CallOfDuty {
    private int mCheckpoint = 1;
    private int mLifeValue = 100;
    private String mWeapon = "沙漠之鹰";

        // 玩游戏
    public void play() {
        System.out.println("玩游戏 : " + String.format("第%d关", mCheckpoint) + " 奋战杀敌中");
        mLifeValue -= 10;
        System.out.println("进度升级啦");
        mCheckpoint++;
        System.out.println("到达 " + String.format("第%d关", mCheckpoint));
    }
    // 退出游戏
    public void quit() {
        System.out.println("-------------");
        System.out.println("退出前的游戏属性 : " + this.toString());
        System.out.println("退出游戏");
        System.out.println("-------------");
    }

        /**
         * 创建备忘录
         */
    public Memoto createMemoto() {
```

```
        Memoto memoto = newMemoto();
        memoto.mCheckpoint = mCheckpoint;
        memoto.mLifeValue = mLifeValue;
        memoto.mWeapon = mWeapon;
        return memoto;
    }
        // 恢复游戏
    public void restore(Memoto memoto) {
        this.mCheckpoint = memoto.mCheckpoint;
        this.mLifeValue = memoto.mLifeValue;
        this.mWeapon = memoto.mWeapon;
        System.out.println("恢复后的游戏属性 : " + this.toString());
    }

    // setter 和 getter 省略

        @Override
    public String toString() {
        return "CallOfDuty [mCheckpoint=" + mCheckpoint + ", mLifeValue=" + mLifeValue
                + ", mWeapon=" + mWeapon + "]";
    }
}
```

在 CallOfDuty 游戏类中，我们存储了几个关键字段，关卡、人物的生命值、武器，当调用 play 函数玩游戏时，我们对关卡和人物的生命值进行修改。在该类中可以通过 createMemoto 函数来创建该用户的备忘录对象，也就是将自身的状态保存到一个 Memoto 对象中。外部可以通过 restore 函数将 CallOfDuty 对象的状态从备忘录对象中恢复。

我们先来看看备忘录对象，它只是存储 CallOfDuty 对象的字段，具体代码如下：

```
/**
 * 备忘录类
 */
public class Memoto {
    public int mCheckpoint;
    public int mLifeValue;
    public String mWeapon;

    @Override
    public String toString() {
        return "Memoto [mCheckpoint=" + mCheckpoint + ", mLifeValue=" + mLifeValue + ",
            mWeapon="+ mWeapon + "]";
    }

}
```

这是一个无状态、无操作的实体类，只负责用来存储 Originator 角色的一些数据，防止外部直接访问 Originator。

而备忘录的操作者则是 Caretaker 角色，我们看看相关的代码：

```
/**
 * Caretaker,负责管理 Memoto
 */
public class Caretaker {
    Memoto mMemoto;  // 备忘录

    /**
     * 存档
```

```
     */
    public void archive(Memoto memoto) {
        this.mMemoto = memoto;
    }

    /**
     * 获取存档
     */
    public Memoto getMemoto() {
        return mMemoto;
    }
}
```

Caretaker 类的职责很简单，就是负责管理 Memoto 对象，也就是备忘录对象。

我们再来看看客户端使用的代码。

```
public class Client {
public static void main(String[] args) {
        // 构建游戏对象
        CallOfDuty game = new CallOfDuty();
        // 1. 打游戏
        game.play();

        Caretaker caretaker = new Caretaker();
        // 2. 游戏存档
        caretaker.archive(game.createMemoto());
        // 3. 退出游戏
        game.quit();
        // 4. 恢复游戏
        CallOfDuty newGame = new CallOfDuty();
        newGame.restore(caretaker.getMemoto());
    }
}
```

上述过程大致有如下 4 步：

（1）开始游戏，闯关升级；

（2）游戏退出之前进行存档；

（3）退出游戏；

（4）重新启动游戏，从存档中恢复游戏进度。

CallOfDuty 在这里为 Originator 角色，也就是需要存储数据的对象，在这里并没有直接存储 CallOfDuty 的对象，而是通过 Memoto 对 CallOfDuty 对象的数据进行存储，然后再存储 Memoto 对象，最终对 Memoto 的存取操作则交给 Caretaker 对象。在这个过程中，各个角色职责清晰、单一，代码也比较简单，即对外屏蔽了对 CallOfDuty 角色的直接访问，在满足了对象状态存取功能的同时也使得该模块的结构保持清晰、整洁。

13.6 Android 源码中的备忘录模式

在 Android 开发中，状态模式应用是 Activity 中的状态保存，也就是里面的 onSaveInstanceState 和 onRestoreInstanceState。当 Activity 不是正常方式退出，且 Activity 在随后的时间内被系统杀死之前会调用这两个方法让开发人员可以有机会存储 Activity 相关的信息，并且在下次返回 Activity

时恢复这些数据。通过这两个函数，开发人员能够在某些特殊场景下存储与界面相关的信息，提升用户体验，例如，用户在写了一大段表白短信之后，此时一个电话打进来，用户的短信输入界面被退到后台，如果在打电话的过程中短信应用被系统杀死了，那么在用户再次进入短信应用时，上一次输入的内容将不会存在，Activity 的状态存储机制就是为了应对这种情况出现的。那么它们的原理是怎样的呢？我们还是一起来看看源码：

onSaveInstanceState 和 onRestoreInstanceState 都是 Activity 中的函数，我们首先来分析 onSaveInstanceState。

```
protected void onSaveInstanceState(Bundle outState) {
        // 1. 存储当前窗口的视图树的状态
        outState.putBundle(WINDOW_HIERARCHY_TAG, mWindow.saveHierarchyState());
        // 2. 存储 Fragment 的状态
        Parcelable p = mFragments.saveAllState();
        if (p != null) {
            outState.putParcelable(FRAGMENTS_TAG, p);
        }
        // 3. 如果用户还设置了 Activity 的 ActivityLifecycleCallbacks
        // 那么调用这些 ActivityLifecycleCallbacks 的 onSaveInstanceState 进行存储状态
        getApplication().dispatchActivitySaveInstanceState(this, outState);
}
```

上述 onSaveInstanceState 函数中，主要分为 3 步：

（1）存储窗口的视图树的状态；

（2）存储 Fragment 的状态；

（3）调用 Activity 的 ActivityLifecycleCallbacks 的 onSaveInstanceState 函数进行状态存储。

我们首先来看第一步，在这一步将 Window 对象中的视图树中各个 View 状态存储到 Bundle 中，这样一来，当用户重新进入到该 Activity 时，用户 UI 的结构、状态才会被重新恢复，以此来保证用户界面的一致性。Window 类的具体实现类是 PhoneWindow，我们看看 PhoneWindow 中 saveHierarchyState 函数的实现：

```
    @Override
public Bundle saveHierarchyState() {
    Bundle outState = new Bundle();
    if (mContentParent == null) {
        return outState;
    }

        // 通过 SparseArray 类来存储，这相当于一个 key 为整型的 map
        SparseArray<Parcelable> states = new SparseArray<Parcelable>();
        // 调用 mContentParent 的 saveHierarchyState 方法，这个 mContentParent 就是调用 Activity 的
        // setContentView 函数设置的内容视图，它是内容试图的根节点，在这里存储整棵视图树的结构
        mContentParent.saveHierarchyState(states);
        // 将视图树结构放到 outState 中
        outState.putSparseParcelableArray(VIEWS_TAG, states);

        // 保存当前界面中获取了焦点的 View
        View focusedView = mContentParent.findFocus();
        if (focusedView != null) {
            // 持有焦点的 View 必须要设置 id,否则重新进入该界面时不会恢复它的焦点状态
            if (focusedView.getId() != View.NO_ID) {
                outState.putInt(FOCUSED_ID_TAG, focusedView.getId());
            } else {
                // 代码省略
```

```
            }
        }

        // 存储整个面板的状态
        SparseArray<Parcelable> panelStates = new SparseArray<Parcelable>();
        savePanelState(panelStates);
        if (panelStates.size() > 0) {
            outState.putSparseParcelableArray(PANELS_TAG, panelStates);
        }
        // 存储 ActionBar 的状态
        if (mActionBar != null) {
            SparseArray<Parcelable> actionBarStates = new SparseArray<Parcelable>();
            mActionBar.saveHierarchyState(actionBarStates);
            outState.putSparseParcelableArray(ACTION_BAR_TAG, actionBarStates);
        }
        return outState;
    }
```

在 saveHierarchyState 函数中，主要是存储了与当前 UI、ActionBar 相关的 View 状态，这里用 mContentParent 来分析。这个 mContentParent 就是我们通过 Activity 的 setContentView 函数设置的内容视图，它是整个内容视图的根节点，存储它层级结构中的 View 状态也就存储了用户界面的状态。mContentParent 是一个 ViewGroup 对象，但是，saveHierarchyState 并不是在 ViewGroup 中，而是在 ViewGroup 的父类 View 中，我们看看下面的程序：

```
    public void saveHierarchyState(SparseArray<Parcelable> container) {
        // 又调用了 dispatchSaveInstanceState 来存储状态
        dispatchSaveInstanceState(container);
    }

    /**
     * 真正存储 View 状态的函数
     */
    protected void dispatchSaveInstanceState(SparseArray<Parcelable> container) {
        // 1. 注意：如果 View 没有设置 id，那么这个 View 的状态将不会被存储
        if (mID != NO_ID && (mViewFlags & SAVE_DISABLED_MASK) == 0) {
        mPrivateFlags&= ~PFLAG_SAVE_STATE_CALLED;
        // 2. 调用 onSaveInstanceState 函数获取自身的状态
        Parcelable state = onSaveInstanceState();
        if ((mPrivateFlags & PFLAG_SAVE_STATE_CALLED) == 0) {
            throw new IllegalStateException(
                "Derived class did not call super.onSaveInstanceState()");
        }
        if (state != null) {
            //  3. 将自身状态放到 container 中，key 为 id、value 为自身状态
            container.put(mID, state);
        }
    }
    // View 类默认存储的状态为空
    protected Parcelable onSaveInstanceState() {
        mPrivateFlags |= PFLAG_SAVE_STATE_CALLED;
        return BaseSavedState.EMPTY_STATE;
    }
}
```

在 View 类中的 saveHierarchyState 函数调用了 dispatchSaveInstanceState 函数来存储自身的状态，而 ViewGroup 则覆写了 dispatchSaveInstanceState 函数来存储自身以及子视图的状态。ViewGroup 的 dispatchSaveInstanceState 函数如下：

```
    @Override
    protected void dispatchSaveInstanceState(SparseArray<Parcelable> container) {
        super.dispatchSaveInstanceState(container);
        final int count = mChildrenCount;
        final View[] children = mChildren;
        for (int i = 0; i < count; i++) {
            View c = children[i];
            if ((c.mViewFlags & PARENT_SAVE_DISABLED_MASK) != PARENT_SAVE_DISABLED) {
                c.dispatchSaveInstanceState(container);
            }
        }
}
```

在 ViewGroup 的 dispatchSaveInstanceState 函数中会调用 super. dispatchSaveInstanceState 函数存储自身的状态，然后遍历所有子视图，并且调用子视图的 dispatchSaveInstanceState 函数来存储它们的状态。如果这个子视图是 ViewGroup 类型，那么则再次执行这个过程；如果这个子视图是 View 类型，那么调用的是 View 的 dispatchSaveInstanceState 函数，在 View 类的 dispatchSaveInstanceState 函数的注释 1 中有一个很重要的地方，就是在 View 没有设置 id 时，这个 View 的状态不会被存储到 Bundle 中，这个 id 就是我们平时在 xml 中通过 android:id 属性来设置的标识 View 的唯一性的 id。具体程序如下：

```
<?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="horizontal" >

<ImageView
android:id="@+id/my_imageview"
android:layout_width="64dp"
android:layout_height="64dp"
android:contentDescription="@string/app_name" />

<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginLeft="20dp"
android:textSize="18sp" />

</LinearLayout>
```

在上述的 xml 中，ImageView 的状态会被存储，而 TextView 因为没有设置 id，它的状态将不会被存储。这里还有一个重要的点，是这个 id 在一个 Activity 的布局中必须是唯一的，因为这些 View 的状态是通过 SparseArray 来存储的，这相当于一个 map，它的 key 是 View 的 id、value 为 View 的状态。当在一个视图树中含有两个相同的 id 时，那么必然会出现状态覆盖的情况。

这些被存储的状态通过 onSaveInstanceState 函数得到，但在 View 类中我们看到返回的是一个空状态，这就意味着，当我们需要存储 View 的状态时，需要覆写 onSaveInstanceState 方法，将要存储的数据放到 Parcelable 对象中，并且将它返回。我们看看 TextView 的实现：

```
@Override
public Parcelable onSaveInstanceState() {
    Parcelable superState = super.onSaveInstanceState();
```

```
    // Save state if we are forced to
    boolean save = mFreezesText;
    int start = 0;
    int end = 0;

    if (mText != null) {
        start = getSelectionStart();
        end = getSelectionEnd();
        if (start >= 0 || end >= 0) {
            save = true;
        }
    }

    // 存储 TextView 的 start、end 及文本内容
    if (save) {
        SavedState ss = new SavedState(superState);
        // XXX Should also save the current scroll position!
        ss.selStart = start;
        ss.selEnd = end;

        if (mText instanceof Spanned) {
            Spannable sp = new SpannableString(mText);
            for (ChangeWatcher cw : sp.getSpans(0, sp.length(), ChangeWatcher.class)) {
                sp.removeSpan(cw);
            }

            if (mEditor != null) {
                removeMisspelledSpans(sp);
                sp.removeSpan(mEditor.mSuggestionRangeSpan);
            }

            ss.text = sp;
        } else {
            ss.text = mText.toString();
        }

        if (isFocused() && start >= 0 && end >= 0) {
            ss.frozenWithFocus = true;
        }
        ss.error = getError();
        // 返回状态对象
        return ss;
    }

    return superState;
}
```

调用了 View 的 onSaveInstanceState 函数之后就得到了 View 要存储的数据，此时执行到注释 3 处，这里以 View 的 id 为 key，以状态为值存储到 container（SparseArray 类型）中：

```
if (state != null) {
    //  3. 将自身状态放到 container 中，key 为 id、value 为自身状态
    container.put(mID, state);
}
```

经过一层一层地遍历之后，整个视图树的状态就被存储起来了。

存储完 Window 的视图树状态信息之后，便会执行存储 Fragment 中的状态信息、回退栈等。这个存储 Fragment 状态信息也是调用它的 onSaveInstanceState 方法，存储 Fragment 的 View 视图树

状态，最后就是调用用户设置的 ActivityLifecycleCallbacks 的 onSaveInstanceState 方法，让用户能够再做一些额外的处理。至此，整个存储过程就完成了！

那么问题就来了，这些存储了状态信息的 Bundle 数据存储在哪呢？

我们上文说到，调用 onSaveInstanceState 是在 Activity 被销毁之前，更确切的说法是在调用 Activity 的 onStop 之前。Activity 的 onStop 方法在 ActivityThread 的 performStopActivity 函数中，相关的代码如下：

```
    final void performStopActivity(IBinder token, boolean saveState) {
        // 获取 ActivityClientRecord
        ActivityClientRecord r = mActivities.get(token);
        // 执行 performStopActivityInner，saveState 表示是否要存储状态
        performStopActivityInner(r, null, false, saveState);
    }

    private void performStopActivityInner(ActivityClientRecord r,
            StopInfo info, boolean keepShown, boolean saveState) {
        if (r != null) {
            if (!r.activity.mFinished && saveState) {
                if (r.state == null) {
                    // 执行 ACtivity 的 OnSaveInstanceState 函数
                    callCallActivityOnSaveInstanceState(r);
                }
            }
            if (!keepShown) {
                try {
                    // 执行 Activity 的 onStop 函数
                    r.activity.performStop();
                } catch (Exception e) {

                }
                r.stopped = true;
            }
            r.paused = true;
        }
    }
    private void callCallActivityOnSaveInstanceState(ActivityClientRecord r) {
    r.state = new Bundle();
    r.state.setAllowFds(false);
    if (r.isPersistable()) {
        r.persistentState = new PersistableBundle();
        mInstrumentation.callActivityOnSaveInstanceState(r.activity, r.state,
        r.persistentState);
    } else {
        // 调用 Instumentation 的 callActivityOnSaveInstanceState 函数
        // 实际上会调用 Activity 的 onSaveInstanceState 函数
        mInstrumentation.callActivityOnSaveInstanceState(r.activity, r.state);
    }
}
```

在 performStopActivity 函数中通过 token 从 mActivities 中获取了一个 ActivityClientRecord 对象，这个对象就存储了 Activity 的信息，我们的状态信息就包含在其中。获取到这个 ActivityClientRecord 之后，调用了 performStopActivityInner 函数，这个函数执行大致有如下 4 步：

（1）判断是否需要存储 Activity 状态；

（2）如果需要存储 Activity 状态，调用 onSaveInstanceState 函数；

（3）将状态信息存储到 ActivityClientRecord 对象的 state 字段中；

（4）调用 Activity 的 onStop 方法。

也就是说在执行 onStop 函数之前，系统会根据情况来选择是否存储 Activity 的状态，并且将这些状态存储到 mActivities 中。这个 mActivities 维护了一个 Activity 的信息表，当 Activity 重新启动时，会从 mActivities 查询对应的 ActivityClientRecord，如果这个记录对象中含有状态信息，那么则调用 Activity 的 onResoreInstanceState 函数，我们看看相关的代码：

```
// 启动 Activity
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    // 代码省略
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
        // 1. 构建 Activity
        activity = mInstrumentation.newActivity(
                cl, component.getClassName(), r.intent);
        // 代码省略
    } catch (Exception e) {
    }

    try {
        // 2. 创建一个 Application
        Application app = r.packageInfo.makeApplication(false, mInstrumentation);
        if (activity != null) {
            // 创建 Context,类型为 ContextImpl
            Context appContext = createBaseContextForActivity(r, activity);
            // 3. 关联 appContext、Application 对象到 Activity 中
            activity.attach(appContext, this, getInstrumentation(), r.token,
                r.ident, app, r.intent, r.activityInfo, title, r.parent,
                r.embeddedID, r.lastNonConfigurationInstances, config);

            // 4. 调用 Activity 的 onCreate 方法
            mInstrumentation.callActivityOnCreate(activity, r.state);
            // 5. 调用 Activity 的 OnRestoreInstanceState 恢复状态
            if (!r.activity.mFinished) {
                if (r.isPersistable()) {
                    if (r.state != null || r.persistentState != null) {
                        mInstrumentation.callActivityOnRestoreInstanceState(activity,
                            r.state,r.persistentState);
                    }
                } else if (r.state != null) {
                    mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
                }
            }
        }
        r.paused = true;
        // 6. 将 Activity 的信息记录对象存到 mActivities 中
        mActivities.put(r.token, r);

        } catch (SuperNotCalledException e) {
            throw e;

        } catch (Exception e) {

        }
    return activity;
}
```

在注释 5 处，系统会判断 ActivityClientRecord 对象中的 state 是否为空，如果不为空则说明存储了该 Activity 的状态，此时就会调用 Activity 的 onSaveInstanceState 函数获取 Activity 的 UI 状态信息，然后将这些状态信息传递给 Activity 的 onCreate 函数，使得用户可以在 onCreate 函数中恢复 UI 上的状态。

在这个过程中，Activity 扮演了 Caretaker 角色，负责存储、恢复 UI 的状态信息；Activity、Fragment、View、ViewGroup 等对象为 Originator 角色，也就是需要存储状态的对象；Memoto 则是由 Bundle 类扮演。Activity 在停止之前会根据 Activity 的退出情景来选择是否需要存储状态，在重新启动该 Activity 时会判断 ActivityClientRecord 对象中是否存储了 Activity 的状态，如果含有状态则调用 Activity 的 onRestoreInstanceState 函数，从而使得 Activity 的 UI 效果与上次保持一致，这样一来，就保证了在非正常情况退出 Activity 时不会丢失数据的情况，很好地提升了用户体验。

13.7 深度拓展

13.7.1 onSaveInstanceState 调用的时机

在上文中我们说到，onSaveInstanceState 并不是在每次 Activity 退出之前都会调用，只是在某些特定的情况之下它才会被调用，那么这些特定的场景是怎样的呢？

这些特定场景总结成一句话就是：当某个 Activity 变得容易被系统销毁时，该 Activity 的 onSaveInstanceState 函数就会被执行，除非该 Activity 是被用户主动销毁的，如当用户按 Back 键时。

意思就是说，该 Activity 还没有被销毁，而仅仅是一种可能性，这种可能性有这么几种情况。

（1）当用户按下 Home 键时。

这是显而易见的，系统不知道你按下 Home 键后要运行多少其他的程序，自然也不知道 Activity A 是否会被销毁，因此，系统会调用 onSaveInstanceState()，让用户有机会保存某些非永久性的数据。

（2）长按 Home 键，选择运行其他的程序时。

（3）按下电源键（关闭屏幕显示）时。

（4）从 Activity A 中启动一个新的 Activity 时。

（5）屏幕方向切换时，如从竖屏切换到横屏时。

（6）电话打入等情况发生时。

一句话概括就是，不是用户主动退出某个 Activity 或者跳转到其他 Activity 的情况下就会触发 onSaveInstanceState。换句话说，onSaveInstanceState()的调用遵循一个重要原则，即当系统存在“未经你许可”时销毁了我们的 Activity，则 onSaveInstanceState()会被系统调用，这是系统的责任，因为它必须要提供一个机会让用户保存数据。

13.7.2 使用 V4 包存储状态的 bug

在我们使用较老的 support V4 包中的 FragmentActiivty 来用 Fragment 时，可能会遇到这样一个空指针异常。

```
Caused by: java.lang.NullPointerException
    at
```

```
android.support.v4.app.FragmentManagerImpl.saveFragmentBasicState(FragmentManager.jav
a:1576)
    at
android.support.v4.app.FragmentManagerImpl.saveAllState(FragmentManager.java:1617)
    at
android.support.v4.app.FragmentActivity.onSaveInstanceState(FragmentActivity.java:481
)
    at android.app.Activity.performSaveInstanceState(Activity.java:1022)
    at
android.app.Instrumentation.callActivityOnSaveInstanceState(Instrumentation.java:1180
)
    at android.app.ActivityThread.performPauseActivity(ActivityThread.java:3144)
```

引起这个问题的原因是，如果一个 Fragment 保存的视图状态为空，并且用户可见提示为真，那么 result 这个 Bundle 对象就不会被初始化，但是，若使用了这个对象，就会导致空指针异常，相关代码如下：

```
Bundle saveFragmentBasicState(Fragment f) {
    Bundle result = null;

    if (mStateBundle == null) {
        mStateBundle = new Bundle();
    }
    f.performSaveInstanceState(mStateBundle);
    if (!mStateBundle.isEmpty()) {
        result = mStateBundle;
        mStateBundle = null;
    }

    if (f.mView != null) {
        saveFragmentViewState(f);
    }
    if (f.mSavedViewState != null) {
        if (result == null) {
            result = new Bundle();
        }
        result.putSparseParcelableArray(
                FragmentManagerImpl.VIEW_STATE_TAG, f.mSavedViewState);
    }
    if (!f.mUserVisibleHint) {

        // 此时 result 为空，这里会报出异常
        result.putBoolean(FragmentManagerImpl.USER_VISIBLE_HINT_TAG,
            f.mUserVisibleHint);
    }

    return result;
}
```

当 saveFragmentBasicState 函数中的最后 if 判断为真，且 result 还未被初始化时就被抛出这个异常，解决的方法就是把最后一个 if 语句修改如下：

```
if (!f.mUserVisibleHint) {
    if (result == null) {
        result = new Bundle();
    }
    // 此时 result 为空，这里会报出异常
    result.putBoolean(FragmentManagerImpl.USER_VISIBLE_HINT_TAG, f.mUserVisibleHint);
}
```

但这是从源码的角度修复 bug，在开发过程中你通常不会采用这种形式。此时，解决这个问题的方法有两个，第一覆写 FragmentActivity 中的 onSaveInstanceState 方法，不调用父类中的 onSaveInstanceState，也就是说不保存该 Activity 中所有 View 的状态，这样也就不会调用到上述崩溃的代码段了，具体代码如下：

```
@Override
    protected void onSaveInstanceState(Bundle outState) {
        //super.onSaveInstanceState(outState);

    }
```

还有一种方法就是覆写 Fragment 中的 onSaveInstanceState 方法，在这个方法中随便存入一个数据，代码如下所示：

```
public class MyFragment extends Fragment {
    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        // 随便存入一个数据
        outState.putString("DO NOT CRASH", "Baby");
    }
    // 代码省略
}
```

这样保证了原来在 saveFragmentBasicState 中引发崩溃的 result 对象被指向 Fragment 存储状态信息的 Bundle 对象中，源码逻辑如下：

```
Bundle saveFragmentBasicState(Fragment f) {
    Bundle result = null;
    if (mStateBundle == null) {
        mStateBundle = new Bundle();
    }
    f.performSaveInstanceState(mStateBundle);
    // Fragment 的 mStateBundle 不为空，此时 result 指向了 mStateBundle
    if (!mStateBundle.isEmpty()) {
        // result 此时指向了 mStateBundle
        result = mStateBundle;
        mStateBundle = null;
    }
    if (!f.mUserVisibleHint) {
        // 此时 result 为空，这里会报出异常
        result.putBoolean(FragmentManagerImpl.USER_VISIBLE_HINT_TAG,
            f.mUserVisibleHint);
    }

    return result;
}
```

“在 App 开发过程中尽可能使用 Fragment 替代 Activity”，Google 官方的这个建议无疑让许多 Android 开发者开始关注、使用 Fragment。但随着使用 Fragment 的人数增多，Fragment 存在的各种问题也开始暴露，在各种 Android 社区中，已经开始有人质疑用 Fragment 替代 Activity 在应用开发中是否真的像 Google 说的那样有益，质疑的理由大概有如下几个：

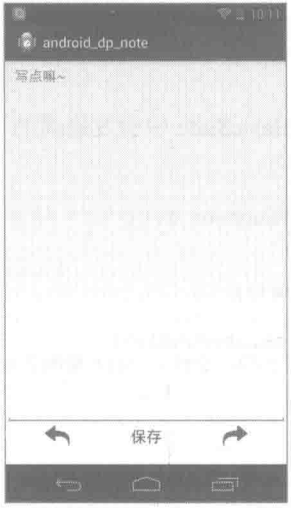
- 在使用 Fragment 时，只能选择使用默认的构造方法，而不能自由地构造我们想要的构造方法；

- 嵌套使用 Fragment 很容易出现各种奇怪的 bug，或者是受到种种限制；
- 复杂的生命周期。

有些热衷于开源的人为了改善上述不足，进行了技术开发，如 Square 开发了 Flow 和 Mortar 库来试图替代 Fragment。随便提一下，修复上述 bug 的开发者就是 Square“当家码农”Jake Wharton。但随着时间的推移，Fragment 还是大家最常用的组件，知道 Flow 和 Mortar 的寥寥无几。

13.8 实战

小民最近闲来无事，打算开发一款“小清新”的记事本应用，虽说市面上已经有各种各样的便签、记事本软件，功能强大的有印象笔记，这些他都不喜欢，觉得还是用自己开发的好用，于是小民决定自己实现这款软件，主要用于记录一些自己平时的小创意、小想法。既然是记录想法，那就需要方便修改，因此，记事本中的撤销、重做功能就是小民这个记事本软件的必须功能，效果图如图 13-2 所示。



▲图 13-2

布局的 xml 程序如下：

```
<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="${relativePackage}.${activityClass}" >

    <EditText
    android:id="@+id/note_edittext"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:gravity="left"
```

```
    android:hint="@string/note_tips" />

    <RelativeLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dp"
    android:paddingLeft="50dp"
    android:paddingRight="50dp" >

        <ImageView
        android:id="@+id/undo_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:src="@drawable/undo" />

        <TextView
        android:id="@+id/save_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="@string/save"
        android:textSize="20sp" />

        <ImageView
        android:id="@+id/redo_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:src="@drawable/redo" />
    </RelativeLayout>

</LinearLayout>
```

这是 MainActivity 中的布局，因此，相关的代码写在 MainActivity 中。小民首先初始化 View、设置点击事件等，代码如下所示：

```
public class MainActivity extends Activity {

    EditText mNodeEditText;  // 编辑器
    TextView mSaveTv;        // 保存按钮
    ImageView mUndoBtn;      // 撤销按钮
    ImageView mRedoBtn;      // 重做按钮

    // 最大存储数量
    private static final int MAX = 30;
    // 存储 30 条记录
    List<Memoto> mMemotos = new ArrayList<Memoto>(MAX);

    int mIndex = 0;

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

        // 初始化视图
        initViews();
    }

    private void initViews() {
```

```
        mNodeEditText = (EditText) findViewById(R.id.note_edittext);
        mUndoBtn = (ImageView) findViewById(R.id.undo_btn);
        mUndoBtn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // 返回上一个记录点
                restoreEditText(getPrevMemoto());
                makeToast("撤销 : ");
            }
        });

        mRedoBtn = (ImageView) findViewById(R.id.redo_btn);
        mRedoBtn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // 恢复状态,恢复到下一个记录点
                restoreEditText(getNextMemoto());
                makeToast("重做 : ");
            }
        });

        mSaveTv = (TextView) findViewById(R.id.save_btn);
        mSaveTv.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                saveMemoto(createMemotoForEditText());
                makeToast("保存笔记 : ");
            }
        });
    }
    private void makeToast(String msgPrex) {
        Toast.makeText(this, msgPrex + mNodeEditText.getText() + ", 光标位置 : "
            + mNodeEditText.getSelectionStart(), Toast.LENGTH_LONG).show();
    }
    // 代码省略
}
```

上述代码是初始化布局中的控件，并且为这些控件设置点击事件的处理。点击保存按钮时，会为编辑器创建一个 Memoto 对象用以保存编辑器的文本和光标位置，并且将这个 Memoto 存到一个列表当中。当用户点击撤销按钮时，从 Memoto 列表取出上一个记录，即恢复到上一个状态；用户点击重做按钮时，也就重复上一个撤销了的动作。这部分的代码如下所示：

```
    public class MainActivity extends Activity {

        // 代码省略
        /**
         * 保存备忘录
         * @param memoto
         */
        public void saveMemoto(Memoto memoto) {
            if (mMemotos.size() > MAX) {
                mMemotos.remove(0);
            }
            mMemotos.add(memoto);
            mIndex = mMemotos.size() - 1;
        }
```

```
        // 获取上一个存档，相当于撤销功能
        public Memoto getPrevMemoto() {
            mIndex = mIndex > 0 ? --mIndex : mIndex;
            return mMemotos.get(mIndex);
        }

        // 获取下一个存档，相当于重做功能
        public Memoto getNextMemoto() {
            mIndex = mIndex < mMemotos.size() - 1 ? ++mIndex : mIndex;
            return mMemotos.get(mIndex);
        }

        // 为编辑器创建 Memoto 对象
        private Memoto createMemotoForEditText() {
            Memoto memoto = new Memoto();
            memoto.text = mNodeEditText.getText().toString();
            memoto.cursor = mNodeEditText.getSelectionStart();
            return memoto;
        }
        // 恢复编辑器状态
        private void restoreEditText(Memoto memoto) {
            mNodeEditText.setText(memoto.text);
            mNodeEditText.setSelection(memoto.cursor);
        }

    }
```

小民是个细心的人，不光要存储编辑器的文本内容，还要把光标的位置也存储起来，这样避免手动移动光标，因此，小民建立一个类来存储这些数据，上述的代码也是操作这个类：

```
/**
 * 存储 EditText 的文本与光标位置
 */
public class Memoto {
    public String text;
    public int cursor;
}
```

记事本的第一版就算完成了，作为一名程序员，小民测试程序的第一步自然是要输入“Hello World”。为了测试功能，在输入了 Hello 之后特地保存了一次，然后再输入 World，如图 13-3 和图 13-4 所示。

在调试程序时，小民不小心把 world 写成了 word！幸好上一步做了保存，而且有撤销功能，小民点击了左下角的撤销按钮，返回上一状态，如图 13-5 所示。

记事本新版本上线了，项目组相对悠闲了些，小民的主管近来也是没啥事，一转头就看到小民在认真地玩手机，于是便到小民的工位来一探究竟。

“小民，干啥呢？”主管问。

“我自己写了一个便签软件练练手，也请您给指导一下？”小民答道。

“好吧，反正最近也没啥事！”主管爽快地答应了，说着就拿起小民的鼠标在屏幕上操作起来。

“小民，你这个 MainActivity 的‘职责’太混乱了！要负责 View 部分的逻辑，还要负责管理你的便签的记录、修改编辑器的状态，把这些代码耦合在一起会造成类型膨胀，后续难以维护，你的 MainActivity 也会越来越脆弱的，怎么刚学完六大原则就忘了……像你这种需要保存对象状态的情

况可以使用备忘录模式，把保存数据的逻辑、职责分离出去，这样耦合更低，每个类的职责也清晰。”把代码看过一遍之后，主管面色有点凝重地说到。

▲图 13-3

▲图 13-4

▲图 13-5

“主管，我这也是刚开始开发，在设计上还没有进行更多考虑，我待会儿再重构程序……”小民有点底气不足地回答。

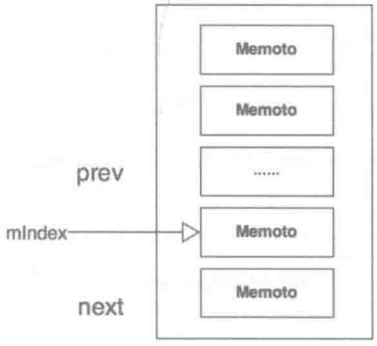
“那行，我也只是给你提供点建议，你再完善一下，我先忙我自己的事了。”主管说完便走开了。

备忘录模式，小民平时倒是看过相关的资料，但是还没有运用过，于是小民在补充了一些基本知识之后便开始对记事本项目进行第一遍重构，主要是运用备忘录模式将管理编辑器数据的职责独立出来。首先建立了一个 NoteCaretaker 类来负责管理 Memoto 对象，具体代码如下：

```
// 负责管理 Memoto 对象
public class NoteCaretaker {
    // 最大存储数量
    private static final int MAX = 30;
    // 存储 30 条记录
    List<Memoto> mMemotos = new ArrayList<Memoto>(MAX);

    int mIndex = 0;

    /**
     * 保存备忘录到记录列表中
     * @param memoto
     */
    public void saveMemoto(Memoto memoto) {
        if (mMemotos.size() > MAX) {
            mMemotos.remove(0);
        }
        mMemotos.add(memoto);
        mIndex = mMemotos.size() - 1;
    }

    // 获取上一个存档信息，相当于撤销功能
    public Memoto getPrevMemoto() {
```

```
        mIndex = mIndex > 0 ? --mIndex : mIndex;
        return mMemotos.get(mIndex);
    }

    // 获取下一个存档信息，相当于重做功能
    public Memoto getNextMemoto() {
        mIndex = mIndex < mMemotos.size() - 1 ? ++mIndex : mIndex;
        return mMemotos.get(mIndex);
    }
}
```

在 NodeCaretaker 中会维护一个备忘录列表，然后使用 mIndex 标识编辑器当前所在的记录点，通过 getPrevMemoto、getNextMemoto 分别获取上一个、下一个记录点的备忘录，以此来达到撤销、重做的功能，如图 13-6 所示。



▲图 13-6

然后自定义了一个 NodeEditText 类，该类继承自 EditText，具体代码如下：

```
// Note 编辑器类
public class NoteEditText extends EditText {
    public NoteEditText(Context context) {
        this(context, null);
    }

    public NoteEditText(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public NoteEditText(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }
    // 创建备忘录对象，即存储编辑器的指定数据
    public Memoto createMemoto() {
        Memoto noteMemoto = newMemoto();
        // 存储文本与光标位置
        noteMemoto.text = getText().toString();
        noteMemoto.cursor = getSelectionStart();
        return noteMemoto;
    }
    // 从备忘录中恢复数据
    public void restore(Memoto memoto) {
        setText(memoto.text);
        // 设置光标位置
```

```
        setSelection(memoto.cursor);
    }
}
```

该类就是添加了两个函数，分别是 createMemoto 和 restore 函数。createMemoto 函数是创建一个存储了编辑器文本、光标位置数据的 Memoto 对象，并且返回给客户端；restore 函数是从 Memoto 对象中恢复编辑器的文本和光标位置。通过添加这两个函数，使得 EditText 能够管理自身的状态信息。如果没有 NodeEditText，那么 createMemoto 和 restore 函数的功能就需要放到其他的类型中，而这两个功能放到其他类型又不太合适，如 MainActivity。最重要的原因是这两个功能操作的都是 EditText 自身的字段，因此它属于这个类型。

最后，MainActivity 的代码就是如下这样：

```
public class MainActivity extends Activity {

    NoteEditText mNodeEditText;
    TextView mSaveTv;
    ImageView mUndoBtn;
    ImageView mRedoBtn;
    // note 备忘录管理器
    NoteCaretaker mCaretaker = newNoteCaretaker();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 初始化视图
        initViews();
    }

    private void initViews() {
        mNodeEditText = (NoteEditText) findViewById(R.id.note_edittext);
        mUndoBtn = (ImageView) findViewById(R.id.undo_btn);
        mUndoBtn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // 返回上一个记录点
                mNodeEditText.restore(mCaretaker.getPrevMemoto());
                makeToast("撤销 : ");
            }
        });

        mRedoBtn = (ImageView) findViewById(R.id.redo_btn);
        mRedoBtn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // 恢复状态,恢复到下一个记录点
                mNodeEditText.restore(mCaretaker.getNextMemoto());
                makeToast("重做 : ");
            }
        });

        mSaveTv = (TextView) findViewById(R.id.save_btn);
        mSaveTv.setOnClickListener(new OnClickListener() {

            @Override
```

```
            public void onClick(View v) {
                mCaretaker.saveMemoto(mNodeEditText.createMemoto());
                makeToast("保存笔记 : ");
            }
        });
    }

    private void makeToast(String msgPrex) {
        Toast.makeText(this, msgPrex + mNodeEditText.getText() + ", 光标位置 : "
            + mNodeEditText.getSelectionStart(), Toast.LENGTH_LONG).show();
    }
}
```

从上述程序中看到代码量少了很多，也简单了很多，职责更为单一、明确。主要的功能就是初始化 View，并且为每个按钮设置点击事件的处理函数，然后调用特定的方法即可。界面相关的类型和业务逻辑的类型隔离开来，即避免了类型膨胀、逻辑混乱等问题，也使得整个应用朝着健康的方向发展。

这里也告诉了我们另一个道理:重构不是到了某一个阶段才应该实施，而是伴随着开发存在的，当你觉得某个类甚至某个函数过于混乱、背负的职责太多时，这就是要进行重构的信号，此时应该在真正了解了该类或者该函数的作用之后，对它进行相应的处理，例如，将不属于它的职责分离到别的类中，将复杂的函数提取为几个粒度更小的函数，通过持续地重构使得软件系统保持整洁，这样一来，你的软件系统才会健康、稳步地发展。

13.9 总结

备忘录模式是在不破坏封装的条件下，通过备忘录对象（Memoto）存储另外一个对象内部状态的快照，在将来合适的时候把这个对象还原到存储起来的状态。

优点

- 给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。
- 实现了信息的封装，使得用户不需要关心状态的保存细节。

缺点

消耗资源，如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

第 14 章　解决问题的“第三者”——迭代器模式

14.1 迭代器模式介绍

迭代器模式（Iterator Pattern）又称为游标（Cursor）模式，是行为型设计模式之一。迭代器模式算是一个比较古老的设计模式，其源于对容器的访问，比如 Java 中的 List、Map、数组等，我们知道对容器对象的访问必然会涉及遍历算法，我们可以将遍历的方法封装在容器中，或者不提供遍历方法。如果我们将遍历的方法封装到容器中，那么对于容器类来说就承担了过多的功能，容器类不仅要维护自身内部的数据元素而且还要对外提供遍历的接口方法，因为遍历状态的存储问题还不能对同一个容器同时进行多个遍历操作，如果我们不提供遍历方法而让使用者自己去实现，又会让容器的内部细节暴露无遗，正因于此，迭代模式应运而生，在客户访问类与容器体之间插入了一个第三者——迭代器，很好地解决了上面所述的弊端。

14.2 迭代器模式的定义

提供一种方法顺序访问一个容器对象中的各个元素，而又不需要暴露该对象的内部表示。

14.3 迭代器模式的使用场景

遍历一个容器对象时。

14.4 迭代器模式的 UML 类图

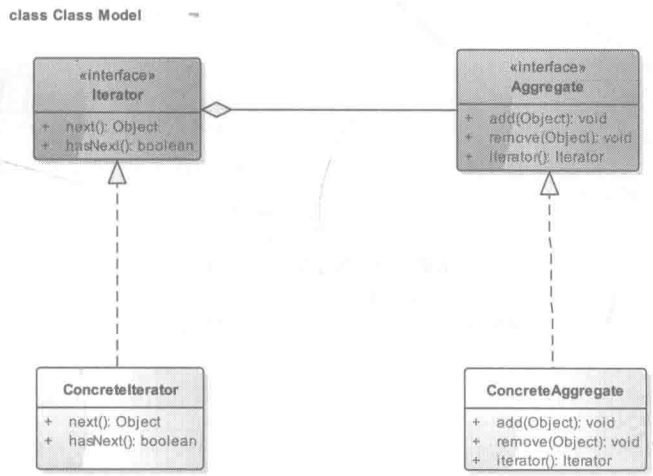
UML 类图如图 14-1 所示。

根据类图可以得出如下一个迭代器模式的通用模式代码。

程序清单 14-1　迭代器接口

```
public interface Iterator<T> {
    /**
     * 是否还有下一个元素
     *
```

```
     * @return true 表示有，false 表示没有
     */
    boolean hasNext();

    /**
     * 返回当前位置的元素并将位置移至下一位
     */
    T next();
}
```



▲图 14-1

程序清单 14-2　具体迭代器类

```
public class ConcreteIterator<T> implements Iterator<T> {
    private List<T> list = new ArrayList<T>();
    private int cursor = 0;

    public ConcreteIterator(List<T> list) {
        this.list = list;
    }

    @Override
    public boolean hasNext() {
        return cursor != list.size();
    }

    @Override
    public T next() {
        T obj = null;
        if (this.hasNext()) {
            obj = this.list.get(cursor++);
        }
        return obj;
    }
}
```

程序清单 14-3　容器接口

```
public interface Aggregate<T> {
    /**
```

```
     * 添加一个元素
     *
     * @param obj 元素对象
     */
    void add(T obj);

    /**
     * 移除一个元素
     *
     * @param obj
     */
    void remove(T obj);

    /**
     * 获取容器的迭代器
     *
     * @return 迭代器对象
     */
    Iterator<T> iterator();
}
```

程序清单 14-4　具体容器类

```
public class ConcreteAggregate<T> implements Aggregate<T> {
    private List<T> list = new ArrayList<T>();

    @Override
    public void add(T obj) {
        list.add(obj);
    }

    @Override
    public void remove(T obj) {
        list.remove(obj);
    }

    @Override
    public Iterator<T> iterator() {
        return new ConcreteIterator<T>(list);
    }
}
```

程序清单 14-5　客户类

```
public class Client {
    public static void main(String[] args) {
        Aggregate<String> a = new ConcreteAggregate<>();
        a.add("Aige");
        a.add("Studio\n");
        a.add("SM");
        a.add(" Brother");
        Iterator<String> i = a.iterator();
        while (i.hasNext()) {
            System.out.print(i.next());
        }
    }
}
```

角色介绍。

Iterator：迭代器接口。

迭代器接口负责定义、访问和遍历元素的接口。

Concrete Iterator：具体迭代器类。

具体迭代器类的目的主要是实现迭代器接口，并记录遍历的当前位置。

Aggregate：容器接口。

容器接口负责提供创建具体迭代器角色的接口。

Concrete Aggregate：具体容器类。

具体迭代器角色与该容器相关联。

Client：客户类。

14.5 模式的简单实现

小民和小辉分别在公司的两个事业部，某天老板安排任务让他们俩统计一下各自部门的员工数据，这很好办嘛，建一个类用数据结构把所有员工数据存进去即可，老板要看的时候给他用 for 循环实现，还是比较容易的，下面就先为员工创建一个实体类：

```
public class Employee {
    private String name;// 姓名
    private int age;// 年龄
    private String sex;// 性别
    private String position;// 职位

    public Employee(String name, int age, String sex, String position) {
        this.name = name;
        this.age = age;
        this.sex = sex;
        this.position = position;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                ", position='" + position + '\'' +
                '}';
    }
}
```

为了简化代码，这里就不提供 setter 和 getter 方法了，同时，为了便于查看信息，这里我们重写了 toString 方法，接下来就是小民和小辉两个部门间人员数据的构建了，对于小民的部门，小民创建一个 CompanyMin 类来存储人员信息，这里为了简化代码，我们不提供对外添加人员信息的 add 方法，只在构造方法中固定数据：

```
public class CompanyMin {
    private List<Employee> list = new ArrayList<>();

    public CompanyMin() {
        list.add(new Employee("小民", 96, "男", "程序猿"));
        list.add(new Employee("小芸", 22, "女", "测试"));
        list.add(new Employee("小方", 18, "女", "测试"));
        list.add(new Employee("可儿", 21, "女", "设计"));
```

```
        list.add(new Employee("朗情", 19, "女", "设计"));
    }

    public List<Employee> getEmployees() {
        return list;
    }
}
```

而对于小辉呢，同样也创建了一个 CompanyHui 类来作为容器：

```
public class CompanyHui {
    private Employee[] array = new Employee[3];

    public CompanyHui() {
        array[0] = new Employee("辉哥", 108, "男", "程序猿");
        array[1] = new Employee("小红", 98, "男", "程序猿");
        array[2] = new Employee("小辉", 88, "男", "程序猿");
    }

    public Employee[] getEmployees() {
        return array;
    }
}
```

可见小民和小辉的内部实现是两种方式，小民的人员信息容器的内部实质是使用的一个 List 来存储人员信息，而小辉的实质上使用的是一个数组，如果老板要查看人员信息就必须遍历两个容器：

```
public class Boss {
    public static void main(String[] args) {
        CompanyMin min = new CompanyMin();
        List minList = min.getEmployees();
        for (int i = 0; i < minList.size(); i++) {
            System.out.println(minList.get(i).toString());
        }

        CompanyHui hui = new CompanyHui();
        Employee[] huiArray = hui.getEmployees();
        for (int i = 0; i < huiArray.length; i++) {
            System.out.println(huiArray[i]);
        }
    }
}
```

这样的代码没有什么问题，老板也能查看到人员的信息，但是，这里要注意的是，如果其他部门的人员信息也有各自不同的实现，那么对于每一个部门的信息容器我们都要在 Boss 类中增加一段遍历逻辑，这是很不科学的？这时候可以用迭代器实现，将遍历逻辑封装，怎么做呢？首先我们需要定义一个迭代器接口：

```
public interface Iterator {
    /**
     * 是否还有下一个元素
     *
     * @return true 表示有，false 表示没有
     */
    boolean hasNext();

    /**
     * 返回当前位置的元素，并将位置移至下一位
```

```
    */
    Object next();
}
```

对于小民和小辉部门的人员信息容器，我们分别创建一个对应的迭代器：

```
public class MinIterator implements Iterator {
    private List<Employee> list;
    private int position;

    public MinIterator(List<Employee> list) {
        this.list = list;
    }

    @Override
    public boolean hasNext() {
        return !(position > list.size() - 1 || list.get(position) == null);
    }
    @Override
    public Object next() {
        Employee e = list.get(position);
        position++;
        return e;
    }
}
public class HuiIterator implements Iterator {
    private Employee[] array;
    private int position;

    public HuiIterator(Employee[] array) {
        this.array = array;
    }
    @Override
    public boolean hasNext() {
        return !(position > array.length - 1 || array[position] == null);
    }
    @Override
    public Object next() {
        Employee e = array[position];
        position++;
        return e;
    }
}
```

同时，我们定义一个容器类的接口，在该接口中定义一个能够返回容器迭代器的方法：

```
public interface Company {
    /**
     * 返回一个迭代器对象
     *
     * @return 迭代器对象
     */
    Iterator iterator();
}
```

这时我们修改一下两个容器类使之实现容器接口，并返回对应的迭代器对象：

```
public class CompanyMin implements Company {
    private List<Employee> list = new ArrayList<>();

    public CompanyMin() {
```

```
        list.add(new Employee("小民", 96, "男", "程序猿"));
        list.add(new Employee("小芸", 22, "女", "测试"));
        list.add(new Employee("小方", 18, "女", "测试"));
        list.add(new Employee("可儿", 21, "女", "设计"));
        list.add(new Employee("朗情", 19, "女", "设计"));
    }

    public List<Employee> getEmployees() {
        return list;
    }

    @Override
    public Iterator iterator() {
        return new MinIterator(list);
    }
}

public class CompanyHui implements Company {
    private Employee[] array = new Employee[3];

    public CompanyHui() {
        array[0] = new Employee("辉哥", 108, "男", "程序猿");
        array[1] = new Employee("小红", 98, "男", "程序猿");
        array[2] = new Employee("小辉", 88, "男", "程序猿");
    }

    public Employee[] getEmployees() {
        return array;
    }

    @Override
    public Iterator iterator() {
        return new HuiIterator(array);
    }
}
```

最后在 Boss 类中查阅人员信息将变得更简单：

```
public class Boss {
    public static void main(String[] args) {
        CompanyMin min = new CompanyMin();
        check(min.iterator());

        CompanyHui hui = new CompanyHui();
        check(hui.iterator());
    }

    private static void check(Iterator iterator) {
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toString());
        }
    }
}
```

具体的输出结果不再给出，大家可以自行尝试。其实，在上述例子中只是做了一个假设，在所谓的自定义“容器”类中又使用了 Java 本身所提供的数据结构来存储数据，这也算是一种不恰当的实现方式，因为迭代器模式的规定不像其他模式那么严格，实现也因人而异，好在大部分高级语言的容器类都为我们提供了相应的迭代器而不需要开发者去实现。

14.6 Android 源码中的模式实现

迭代器这个模式对开发者来说几乎不会自己去实现一个迭代器，就拿 Android 来说，其除了各种数据结构体，如 List、Map 等所包含的迭代器外，Android 自身源码中也为我们提供了迭代器遍历数据，最为典型的例子就是数据库查询使用 Cursor，当我们使用 SQLiteDatabase 的 query 方法查询数据库时，会返回一个 Cursor 游标对象，该游标对象实质就是一个具体的迭代器，我们可以使用它来遍历数据库查询所得的结果集，这里给出一个简单的查询 SQLLite 数据库的例子。首先定义一个 SQLiteOpenHelper：

```
public class DBHelper extends SQLiteOpenHelper {
    public DBHelper(Context context) {
        super(context, "DB_AIGE", null, 1);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE table_aige (_id INTEGER PRIMARY KEY AUTOINCREMENT, name
        TEXT, sex TEXT)");
        db.execSQL("INSERT INTO table_aige (name, sex) values ('Aige', 'man')");
        db.execSQL("INSERT INTO table_aige (name, sex) values ('SMBrother', 'man')");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

    }
}
```

为了方便起见，直接在 onCreate 方法中创建表并直接插入了两条数据，同时，也是为了简便起见，就不再创建数据库操作相关的业务逻辑方法了，这里直接用一个 ContentProvider 来对数据库进行操作：

```
public class DataProvider extends ContentProvider {
    private DBHelper mDBHelper;

    @Override
    public boolean onCreate() {
        mDBHelper = new DBHelper(getContext());
        return true;
    }
    @Override
    public Cursor query(Uri uri, String[] projection, String selection, String[]
    selectionArgs, String sortOrder) {
        SQLiteDatabase db = mDBHelper.getReadableDatabase();
        return db.query("table_aige", projection, null, null, null, null, null);
    }
    @Override
    public String getType(Uri uri) {
        return null;
    }
    @Override
    public Uri insert(Uri uri, ContentValues values) {
```

```
        return null;
    }
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }
    @Override
    public int update(Uri uri, ContentValues values,String selection,String[] selectionArgs){
        return 0;
    }
}
```

代码非常简单，我们这里只实现了查询方法，其余均保持默认，当然，作为四大组件之一，我们还需要在 AndroidManifest.xml 文件中对该 ContentProvider 进行声明：

```
<provider
    android:name=".design.iterator.DataProvider"
    android:authorities="com.aigestudio.test.DataProvider"/>
```

最后我们在 Activity 中通过这个 ContentProvider 查询数据库：

```
public class IteratorActivity extends ListActivity {
    private static final String[] PROJECTION = new String[]{"name", "sex"};
    private static final Uri URI = Uri.parse("content://com.aigestudio.test.DataProvider/
    table_aige");

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Cursor cursor = getContentResolver().query(URI, PROJECTION, null, null, null);
        List<Map<String, String>> list = new ArrayList<>();
        cursor.moveToFirst();
        do {
            Map<String, String> item = new HashMap<>();
            item.put("name", cursor.getString(0));
            item.put("sex", cursor.getString(1));
            list.add(item);
        } while (cursor.moveToNext());
        cursor.close();
        setListAdapter(new SimpleAdapter(this, list, android.R.layout.simple_list_item_2, new
        String[]{"name", "sex"}, new int[]{android.R.id.text1, android.R.id.text2}));
    }
}
```

具体的显示结果如图 14-2 所示。

▲图 14-2

14.7 总结

对于迭代模式来说，其自身优点很明显也很单一，支持以不同的方式去遍历一个容器对象，也可以有多个遍历，弱化了容器类与遍历算法之间的关系，而其缺点就是对类文件的增加。

大家也可能会想到其他的语言，如 C++、Python、PHP 等，它们内部也有众多容器体的定义，当然，也有相应的迭代器。迭代器模式发展至今，几乎每一种高级语言都有相应的内置实现，对于开发者而言，已经极少会去由自己来实现迭代器了，因此，对于本章的内容更多地是在于了解而非应用。

第 15 章　抓住问题核心——模板方法模式

15.1 模板方法模式介绍

在面向对象开发过程中，通常会遇到这样的一个问题，我们知道一个算法所需的关键步骤，并确定了这些步骤的执行顺序，但是，某些步骤的具体实现是未知的，或者说某些步骤的实现是会随着环境的变化而改变的，例如，执行程序的流程大致如下：

（1）检查代码的正确性；

（2）链接相关的类库；

（3）编译相关代码；

（4）执行程序。

对于不同的程序设计语言，上述 4 个步骤都是不一样的，但是，它们的执行流程是固定的，这类问题的解决方案就是我们本章要讲的模板方法模式。

15.2 模板方法模式的定义

定义一个操作中的算法的框架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

15.3 模板方法模式的使用场景

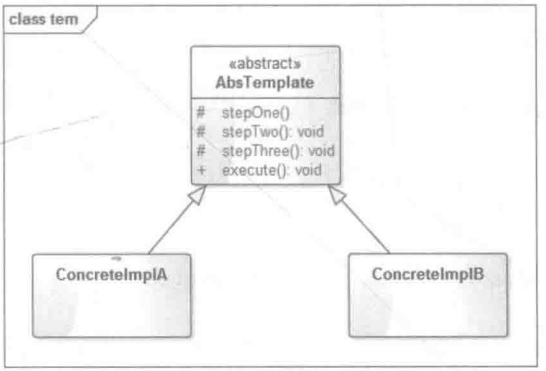
（1）多个子类有公有的方法，并且逻辑基本相同时。

（2）重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。

（3）重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数约束其行为。

15.4 模板方法模式的 UML 类图

UML 类图如图 15-1 所示。



▲图 15-1

角色介绍。

- AbsTemplate：抽象类，定义了一套算法框架。
- ConcreteImplA：具体实现类 A。
- ConcreteImplB：具体实现类 B。

15.5 模板方法模式的简单示例

模板方法实际上是封装一个固定流程，就像是一套执行模板一样，第一步该做什么，第二步该做什么都已经在抽象类中定义好。而子类可以有不同的算法实现，在框架不被修改的情况下实现某些步骤的算法替换，下面以打开计算机这个动作来简单演示一下模板方法。打开计算机的整个过程都是相对固定的，首先启动计算机电源，计算机检测自身状态没有问题时将进入操作系统，对用户进行验证之后即可登录计算机，下面我们使用模板方法来模拟一下这个过程：

```
package com.dp.example.templatemethod;
// 抽象的 Computer
public abstract class AbstractComputer {

    protected void powerOn() {
        System.out.println("开启电源");
    }
    protected void checkHardware() {
        System.out.println("硬件检查");
    }
    protected void loadOS() {
        System.out.println("载入操作系统");
    }

    protected void login() {
        System.out.println("小白的计算机无验证，直接进入系统");
    }
    /**
     * 启动计算机方法，步骤固定为开启电源、系统检查、加载操作系统、用户登录。该方法为 final，防止算法框
架被覆写
     */
    public final void startUp() {
        System.out.println("------ 关机 START ------");
```

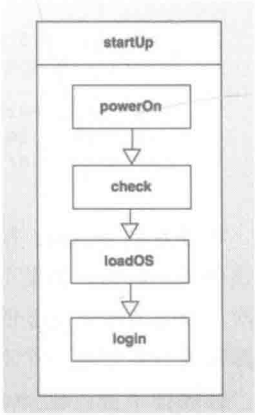
```
        powerOn();
        checkHardware();
        loadOS();
        login();
        System.out.println("------ 关机 END ------");
    }
}
package com.dp.example.templatemethod;
/**
 * 程序员的计算机
*/
public class CoderComputer extends AbstractComputer {
    @Override
    protected void login() {
        System.out.println("程序员只需要进行用户和密码验证就可以了");
    }
}
package com.dp.example.templatemethod;
/**
 * 军用计算机
 *
 * @author mrsimple
 */
public class MilitaryComputer extends AbstractComputer {
    @Override
    protected void checkHardware() {
        super.checkHardware();
        System.out.println("检查硬件防火墙");
    }
    @Override
    protected void login() {
        System.out.println("进行指纹识别等复杂的用户验证");
    }
}
package com.dp.example.templatemethod;
// 测试代码
public class Test {
    public static void main(String[] args) {
        AbstractComputer comp = new CoderComputer();
        comp.startUp();

        comp = new MilitaryComputer();
        comp.startUp();
    }
}
```

输出结果如下。

```
------ 开机 START ------
开启电源
硬件检查
载入操作系统
程序员只需要进行用户和密码验证就可以了
------ 关机 END ------
------ 开机 START ------
开启电源
硬件检查
检查硬件防火墙
载入操作系统
进行指纹之别等复杂的用户验证
------ 关机 END ------
```

通过上面的例子可以看到，在 startUp 方法中有一些固定的步骤，依次为开启电源、检查硬件、加载系统、用户登录 4 个步骤，这 4 个步骤是计算机开机过程中不会变动，如图 15-2 所示。

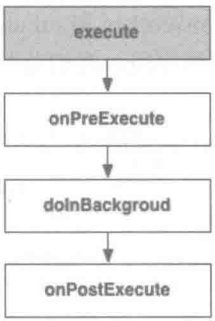
但是，不同用户的这几个步骤的实现可能各不相同，因此，子类需要覆写相应的方法来进行自定义处理，这里需要注意的是 startUp 为 final 方法，这样就保证了逻辑流程不能被子类修改，子类只能够改变某一步骤中的具体实现，这样就保证了这个逻辑流程的稳定性。startUp 中的这几个算法步骤我们可以称为是一个套路，也称为模板方法，这也是该模式的由来。



▲图 15-2

15.6 Android 源码中的模板方法模式

在 Android 中，AsyncTask 是比较常用的一个类型，这个类就使用了模板方法模式。关于 AsyncTask 更详细的分析，请参考《Android 中 AsyncTask 的使用与源码分析》这篇文章（网址为 http://blog.csdn.net/bboyfeiyu/article/details/8973058），我们这里只分析在该类中使用的模板方法模式。



▲图 15-3

在使用 AsyncTask 时，我们都知道把耗时的方法放在 doInBackground(Params... params)中，在 doInBackground 之前，如果还想做一些类似初始化的操作，可以把实现写在 onPreExecute 方法中，当 doInBackground 方法执行完成后，会执行 onPostExecute 方法，而我们只需要构建 AsyncTask 对象，然后执行 execute 方法即可。我们可以看到，它整个执行过程其实是一个框架，具体的实现都需要子类来完成，而且它执行的算法框架是固定的，调用 execute 后会依次执行 onPreExecute、doInBackground、onPostExecute，当然也可以通过 onProgressUpdate 来更新进度，我们可以简单地理解为如图 15-3 所示。

下面我们看源码，首先我们看执行异步任务的入口，即 execute 方法：

```
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
        return executeOnExecutor(sDefaultExecutor, params);
    }

    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;
        exec.execute(mFuture);
        return this;
    }
```

可以看到 execute 方法是一个 final 方法，它调用了 executeOnExecutor 方法，在该方法中会判断该任务的状态，如果不是 PENDING 状态则抛出异常，这也解释了为什么 AsyncTask 只能被执行一次，因为 AsyncTask 的 RUNNING 和 FINISHED 状态都会抛出异常，因此，每次使用 AsyncTask 时都需要重新创建一个对象。

继续往下看，在 executeOnExecutor 方法中首先执行了 onPreExecute 方法，因为 AsyncTask 的要求是需要在 UI 线程中调用 execute 方法，因此，onPreExecute 方法也执行在 UI 线程，然后将 params 参数传递给了 mWorker 对象的 mParams 字段，并且执行了 exec.execute(mFuture)方法。

mWorker 和 mFuture 又是什么呢？其实 mWorker 只是实现了 Callable 接口，并添加了一个参数数组字段，我们分别来分析，通过跟踪代码可以看到，这两个字段都是在构造函数中初始化：

```
public AsyncTask() {
    // 构建一个 Worker 对象
    mWorker = new WorkerRunnable<Params, Result>() {
        public Result call() throws Exception {
            mTaskInvoked.set(true);

            Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
                // 调用 doInBackground,并且将结果传递出去
            return postResult(doInBackground(mParams));
        }
    };
    mFuture = new FutureTask<Result>(mWorker) {
        @Override
        protected void done() {
            try {
                postResultIfNotInvoked(get());
            } catch (InterruptedException e) {
                android.util.Log.w(LOG_TAG, e);
            } catch (ExecutionException e) {
            // 代码省略
        }
    };
}
```

简单地说就是，mFuture 包装了这个 mWorker 对象，在这个 mFuture 对象的 run 函数中又会调用 mWorker 对象的 call 方法，在 call 方法中又调用了 doInBackgroud 函数。因为 mFuture 提交给了线程池来执行，所以，使得 doInBackgroud 执行在非 UI 线程。得到 doInBackgroud 的结果后，通过 postResult 传递结果给 UI 线程，我们再看看 postResult 的实现：

```
private Result postResult(Result result) {
    Message message = sHandler.obtainMessage(MESSAGE_POST_RESULT,
    new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}
```

```
private static class InternalHandler extends Handler {
    @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
    @Override
    public void handleMessage(Message msg) {
        AsyncTaskResult result = (AsyncTaskResult) msg.obj;
        switch (msg.what) {
            case MESSAGE_POST_RESULT:
                // 调用了 AsyncTask 的 finish 方法
                result.mTask.finish(result.mData[0]);
                break;
            case MESSAGE_POST_PROGRESS:
                result.mTask.onProgressUpdate(result.mData);
                break;
        }
    }
}
private void finish(Result result) {
    if (isCancelled()) {
        onCancelled(result);
    } else {
        // 结果通过 onPostExecute 回调给用户
        onPostExecute(result);
    }
    mStatus = Status.FINISHED; // 修改状态
}
```

从程序中可以看到，postResult 就是把一个消息（msg.what == MESSAGE_POST_RESULT）发送给 sHandler，sHandler 为 InternalHandler 类型，当 InternalHandler 接到 MESSAGE_POST_RESULT 类型的消息时就会调用 result.mTask.finish(result.mData[0]) 方法。我们可以看到 result 为 AsyncTaskResult 类型，具体源码如下：

```
@SuppressWarnings({"RawUseOfParameterizedType"})
private static class AsyncTaskResult<Data> {
    final AsyncTask mTask;
    final Data[] mData;

    AsyncTaskResult(AsyncTask task, Data... data) {
        mTask = task;
        mData = data;
    }
}
```

从上述程序可以看到 mTask 就是 AsyncTask 对象，调用 AsyncTask 对象的 finish 方法时又调用了 onPostExecute，这个时候整个执行过程就完成了。

总之，execute 方法内部封装了 onPreExecute、doInBackground、onPostExecute 这个逻辑流程，用户可以根据自己的需求再覆写这几个方法，使得用户可以很方便地使用异步任务来完成耗时的操作及更新 UI，这其实就是通过线程池来执行耗时的任务，得到结果之后，通过 Handler 将结果传递到 UI 线程来执行。

15.7 深度拓展

另一个比较好的模板方法示例就是 Activity 的生命周期函数，例如，Activity 从启动到显示到

窗口中会经历如下过程：onCreate、onStart、onResume，这就是一个典型的 Activity 启动流程，也是一个模板方法的运用。

我们知道，在 Android 系统启动时，第一个启动起来的进程就是 zygote 进程，然后由 zygote 启动 SystemServer，再后就是启动 ActivityManagerService、WindowManagerService 等系统核心服务，这些服务承载着整个 Android 系统与客户端程序交互的重担。zygote 除了启动系统服务与进程之外，普通的用户进程也由 zygote 进程 fork 而来，当一个应用进程启动起来后，就会加载用户在 AndroidManifest.xml 中配置的默认加载的 Activity，此时加载的入口是 ActivityThread.main(String[] args)方法，这个方法就是类似于 C 语言中的 main 方法，是整个应用程序的入口。

在 ActivityThread.main(String[] args)这个方法中，主要的功能就是创建 Application 和 Activity，并且调用 Activity 的一些生命周期函数，如 onCreate、onResume 等。下面就从 ActivityThread.main (String[] args)这个入口开始学习：

```
public static final void main(String[] args) {
    SamplingProfilerIntegration.start();
    Process.setArgV0("<pre-initialized>");
    // 1. 创建 UI 线程的 Looper
    Looper.prepareMainLooper();
    if (sMainThreadHandler == null) {
        sMainThreadHandler = new Handler();
    }
    // 2. 创建 ActivityThread 对象,但它并不是线程，感觉是封装了 UI 线程消息循环与操作 Activity 生命
    // 周期的工具类
    ActivityThread thread = new ActivityThread();
    // 3. 调用 ActivityThread 的 attach 方法，这是核心方法，Application、Activity 的创建入口就在这里
    thread.attach(false);
    // 4. 启动 UI 线程消息循环
    Looper.loop();

    thread.detach();
}


// UI 线程与 Application 关联
private final void attach(boolean system) {
    sThreadLocal.set(this);
    mSystemThread = system;
    // 不是系统进程
    if (!system) {
        android.ddm.DdmHandleAppName.setAppName("<pre-initialized>");
        RuntimeInit.setApplicationObject(mAppActivityManagerNativeThread.asBinder());
        // 5. 获取 ActivityManagerService 实例
        IActivityManager mgr = ActivityManagerNative.getDefault();
        try {
            // 6. 注意这里，将 mAppThread 与 Application 进行管理
            mgr.attachApplication(mAppThread);
        } catch (RemoteException ex) {
        }
    } else {
        // 代码省略
    }
    // 代码省略
}
```

从上述程序可以看到，在 ActivityThread.main 中主要的功能就是创建了 UI 线程消息循环，并

且启动了消息循环。最重要的是创建了 ActivityThread，并调用了 attach 方法。在 attach 方法中又调用了 ActivityManagerService 中的 attachApplication(mAppThread)方法，这个才是我们的重点。mAppThread 是 ApplicationThread 类型，它也不是一个 Thread，而是一个 Binder，负责与远程的 ActivityManagerService 进行交互。在了解 attachApplication 之前，我们先来了解一下 ApplicationThread 与 ActivityThread 的关联，具体代码如下。

```
public final class ActivityThread {
// 定义 mAppThread 与 mH
final ApplicationThread mAppThread = new ApplicationThread();
 final Looper mLooper = Looper.myLooper();
 final H mH = new H();

// 通过 mH 发送消息
  private void sendMessage(int what, Object obj, int arg1, int arg2, boolean async) {
      Message msg = Message.obtain();
      msg.what = what;
      msg.obj = obj;
      msg.arg1 = arg1;
      msg.arg2 = arg2;
      if (async) {
         msg.setAsynchronous(true);
      }
      mH.sendMessage(msg);
   }
   // AppcationThread 内部类，负责与 ActivityManagerService 交互
   private final class ApplicationThread extends ApplicationThreadNative {
      // 代码省略
      // 向 UI 线程发一个 resume activity 的消息
      public final void scheduleResumeActivity(IBinder token, boolean isForward) {
         SendMessage(H.RESUME_ACTIVITY, token, isForward ? 1 : 0);
      }
      // 发送启动 Activity 的消息
      public final void scheduleLaunchActivity(Intent intent, IBinder token,
      int ident, ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
            List<Intent> pendingNewIntents, boolean notResumed, Boolean
            isForward) {
               ActivityClientRecord r=new ActivityClientRecord();

               r.token=token;
               r.ident=ident;
               r.intent=intent;
               r.activityInfo=info;
               r.state=state;
               // 代码省略

            // 向 UI 线程发一个 H.LAUNCH_ACTIVITY 消息
         sendMessage(H.LAUNCH_ACTIVITY, r);
      }

      // 省略代码
   } // end ApplicationThread


   // H 内部类，ApplicationThread 通过 H 类型的对象向 UI 线程投递消息
   private final class H extends Handler {
      public static final int LAUNCH_ACTIVITY = 100;
      public static final int PAUSE_ACTIVITY = 101;
      public static final int PAUSE_ACTIVITY_FINISHING= 102;
```

```
        // 其他程序省略
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case LAUNCH_ACTIVITY: {
                    ActivityClientRecord r = (ActivityClientRecord)msg.obj;
                    r.packageInfo = getPackageInfoNoCheck(
                        r.activityInfo.applicationInfo);
                    // 加载 Activity
                    handleLaunchActivity(r, null);
                } break;
                // activity pause
                case PAUSE_ACTIVITY:
                    handlePauseActivity((IBinder)msg.obj,false, msg.arg1 != 0,msg.arg2);
                    maybeSnapshot();
                    break;
                // 代码省略
            }

        }

    } // end H

} // end ActivityThread
```

ApplicationThread 与 ActivityManagerService 进行交互，通过 H 类的对象 mH 来发送消息，并且在 H 类的 handleMessage 函数中进行处理，然后 handleMessage 根据消息的类型调用 ActivityThread 中对应的方法，比如 handleResumeActivity 来使 Activity 变得可见等。因为 H 的消息队列就是主线程的消息队列，因此，这些过程都在 UI 线程中处理。

下面继续关注 ActivityManagerService 中的 attachApplication(mAppThread)：

```
public final void attachApplication(IApplicationThread thread) {
    synchronized (this) {
        int callingPid = Binder.getCallingPid();
        final long origId = Binder.clearCallingIdentity();
        // 1. 转到了 attachApplicationLocked
        attachApplicationLocked(thread, callingPid);
        Binder.restoreCallingIdentity(origId);
    }
}

    private final boolean attachApplicationLocked(IApplicationThread thread,
            int pid) {
        // 进程 Record
        ProcessRecord app;
    // 代码省略
        try {
    // 2. ApplicationThread 绑定 Application
            thread.bindApplication(processName, appInfo, providers,
            app.instrumentationClass, profilerInfo, app.instrumentationArguments,
                    app.instrumentationWatcher,
                    app.instrumentationUiAutomationConnection, testMode, enableOpenGlTrace,
                    isRestrictedBackupMode || !normalMode, app.persistent,
                    new Configuration(mConfiguration), app.compat, getCommonServicesLocked(),
                    mCoreSettingsObserver.getCoreSettingsLocked());
        // 代码省略
        } catch (Exception e) {
            return false;
        }
```

```
    // 代码省略
        // 正常模式启动 Activity
        if (normalMode) {
            try {
    // 3. 调用 ActivityStackSupervisor 的 attachApplicationLocked 函数
                if (mStackSupervisor.attachApplicationLocked(app)) {
                    didSomething = true;
                }
            } catch (Exception e) {
                badApp = true;
            }
        }
        // 代码省略
        return true;
    }
```

在省略了一些代码之后，我们抓住了需要关心的点，即注释 2 处，这里才是真正启动 Activity 的地方。mStackSupervisor 的类型为 ActivityStackSupervisor，我们看看注释 2 处的 ActivityStackSupervisor 类中的 attachApplicationLocked：

```
boolean attachApplicationLocked(ProcessRecord app) throws RemoteException {
        final String processName = app.processName;
        boolean didSomething = false;
        for(int displayNdx=mActivityDisplays.size()-1;displayNdx >= 0; --displayNdx) {
            ArrayList<ActivityStack>stacks=mActivityDisplays.valueAt(displayNdx). mStacks;
            for (int stackNdx = stacks.size() - 1; stackNdx >= 0; --stackNdx) {
            // 从 Activity 栈中取出 Activity
                final ActivityStack stack = stacks.get(stackNdx);
                if (!isFrontStack(stack)) {
                    continue;
                }
                // 获取栈顶的 ActivityRecord
                ActivityRecord hr = stack.topRunningActivityLocked(null);
                if (hr != null) {
                    if (hr.app == null && app.uid == hr.info.applicationInfo.uid
                    && processName.equals(hr.processName)) {
                        try {
                            // 启动 Activity
                            if (realStartActivityLocked(hr, app, true, true)) {
                                didSomething = true;
                            }
                            } catch (RemoteException e) {
                                    throw e;
                        }
                    }
                }
            }
        }
        return didSomething;
    }
```

在 attachApplicationLocked 函数中会遍历整个 Activity 栈，并且找出栈顶 ActivityStack，然后调用 realStartActivityLocked 函数来真正启动 Activity。我们看看 realStartActivityLocked 函数：

```
final boolean realStartActivityLocked(ActivityRecord r,
        ProcessRecord app, boolean andResume, boolean checkConfig)
        throws RemoteException {

    final ActivityStack stack = r.task.stack;
```

```
    try {
// 设置 app 的 dex 文件的所在位置,指定包名
        mService.ensurePackageDexOpt(r.intent.getComponent().getPackageName());
        // 1. 调用 ActivityThread 的 scheduleLaunchActivity 启动 Activity
        app.thread.scheduleLaunchActivity(new Intent(r.intent), r.appToken,
                System.identityHashCode(r),r.info,new Configuration(mService. mConfiguration),
                r.compat,r.task.voiceInteractor,app.repProcState,r.icicle,r.persistentState,
                results,newIntents,!andResume,mService.isNextTransitionForward(), profilerInfo);
    } catch (RemoteException e) {
        throw e;
    }
    return true;
}
```

程序实现起来较复杂，最终在注释 1 处调用了 ActiivityThread 中的内部类的 ApplicationThread 中的 scheduleLaunchActivity 函数。该函数中会发送一个 H.LAUNCH_ACTIVITY 消息给 UI 线程，具体代码如下：

```
public class ActivityThread {

    private class ApplicationThread extends ApplicationThreadNative {

        public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
                ActivityInfo info, Configuration curConfig, CompatibilityInfo compatInfo,
                IVoiceInteractor voiceInteractor, int procState, Bundle state,
                PersistableBundle persistentState, List<ResultInfo> pendingResults,
                List<Intent> pendingNewIntents, boolean notResumed, boolean isForward,
                ProfilerInfo profilerInfo) {
                    updateProcessState(procState, false);
                    ActivityClientRecord r = new ActivityClientRecord();
                    // 初始化代码
                    // 更新配置
                    updatePendingConfiguration(curConfig);
                    // 发送启动 Activity 的消息
                    sendMessage(H.LAUNCH_ACTIVITY, r);
        }
    }
}
```

ActivityThread 的内部类 H 的 handleMessage 函数中会处理该消息，然后会调 ActivityThread 的 handleLaunchActivity(ActivityClientRecord r, Intent customIntent)函数，在这个函数中会创建将要启动的 Activity，并且调用其生命周期函数 onCreate、onResume。ApplicationThread 中的 scheduleLaunchActivity 函数的代码在上文已经给出，下面我们看看 ActivityThread 的 handleLaunchActivity (ActivityClientRecord r, Intent customIntent)函数：

```
// ActivityThread 中的 handleLaunchActivity
private final void handleLaunchActivity(ActivityClientRecord r, Intent customIntent)
{
        // 代码省略
        // 1. 创建并且加载 Activity，调用其 onCreate 函数
        Activity a = performLaunchActivity(r, customIntent);

        if (a != null) {
            r.createdConfig = new Configuration(mConfiguration);
            Bundle oldState = r.state;
            // 2. 调用 Activity 的 onResume 方法，使 Activity 变得可见
            handleResumeActivity(r.token, false, r.isForward);
```

```
    }
}
private final Activity performLaunchActivity(ActivityClientRecord r,
Intent customIntent) {
    // 代码省略
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
        // 3. 创建 Activity
        activity = mInstrumentation.newActivity(
                cl, component.getClassName(), r.intent);
        r.intent.setExtrasClassLoader(cl);
        if (r.state != null) {
            r.state.setClassLoader(cl);
        }
    } catch (Exception e) {
    }
    try {
        // 4. 创建 Application
        Application app = r.packageInfo.makeApplication(false, mInstrumentation);

        if (activity != null) {
            // 构建 Context
            ContextImpl appContext = new ContextImpl();
            appContext.init(r.packageInfo, r.token, this);
            appContext.setOuterContext(activity);
            // 获取 Activity 的 title
            CharSequence title = r.activityInfo.loadLabel(appContext.
            getPackageManager());
            Configuration config = new Configuration(mConfiguration);
                // 5. Activity 与 context、Application 关联起来
            activity.attach(appContext, this, getInstrumentation(), r.token,
                    r.ident, app, r.intent, r.activityInfo, title, r.parent,
                    r.embeddedID, r.lastNonConfigurationInstance,
                    r.lastNonConfigurationChildInstances, config);
    // 配置 Activity，代码省略
            // 6. 回调 Activity 的 onCreate 方法
    if (r.isPersistable()) {
    mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);
} else {
    mInstrumentation.callActivityOnCreate(activity, r.state);
}
            // 代码省略
            // 7. 执行 Activity 的 onStart 方法
            if (!r.activity.mFinished) {
                activity.performStart();
                r.stopped = false;
            }
            // 代码省略
        }
        r.paused = true;

        mActivities.put(r.token, r);

    } catch (SuperNotCalledException e) {
        throw e;
    }
    return activity;
}
```

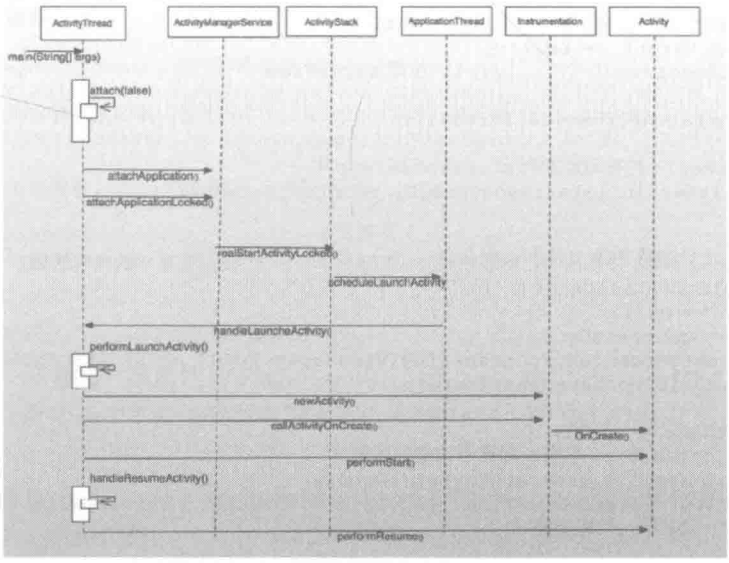
如上述代码所示，在调用 ActivityThread 的 handleLaunchActivity(ActivityClientRecord r, Intent

customIntent)后会调用 performLaunchActivity 函数来创建 Activity，并且将 Activity 与 Application 关联上，然后调用 Activity 的 onCreate、onStart 函数。再之后是调用 handleResumeActivity 函数，该函数的实现如下：

```
    final void handleResumeActivity(IBinder token, boolean clearHide, boolean isForward,
            boolean reallyResume) {
        // 1. 最终调用 Activity 的 onResume 方法
        ActivityClientRecord r = performResumeActivity(token, clearHide);
        if (r != null) {
            final Activity a = r.activity;
            if (r.window == null && !a.mFinished && willBeVisible) {
                // 获取了 Activity 的 Window
                r.window = r.activity.getWindow();
                // 获取了 Window 的 DecorView，也就是最顶级的视图
                View decor = r.window.getDecorView();
                // 将 DecorView 先隐藏
                decor.setVisibility(View.INVISIBLE);
                // 获取 WindowManager
                ViewManager wm = a.getWindowManager();
                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;
                    // 将 DecorView 添加到窗口中
                    wm.addView(decor, l);
                }
            }

            if (!r.activity.mFinished && willBeVisible
            && r.activity.mDecor != null && !r.hideForNow) {
                // 代码省略
                if (r.activity.mVisibleFromClient) {
                    // 使得 Activity 变得可见，其实是设置 DecorView 变为可见
                    r.activity.makeVisible();
                }
               }
            // 最后通知 ActivityManagerNative，该 Activity 已经变为 resume 状态
            if (reallyResume) {
                try {
                    ActivityManagerNative.getDefault().activityResumed(token);
                } catch (RemoteException ex) {
                }
            }
        }
    }

    // 真正执行 Activity 的 onResume 方法
  public final ActivityClientRecord performResumeActivity(IBinder token,
            boolean clearHide) {
        ActivityClientRecord r = mActivities.get(token);

        if (r != null && !r.activity.mFinished) {
                try {
                // 代码省略
                // 执行 onResume
                r.activity.performResume();
            } catch (Exception e) {
            }
```

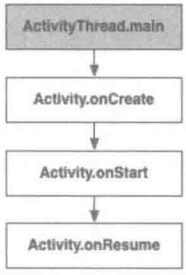
```
        }
        return r;
    }
```

handleResumeActivity 函数又调用 performResumeActivity 函数来回调 Activity 的 onResume 函数，并且 DecorView 添加到 WindowManager 中，最后将 Activity 的 DecorView 设置为可见，并且通知 ActivityManagerService 渲染视图，因此，在 onResume 函数之后，Activity 就显示在屏幕上了。时序图如图 15-4 所示。



▲图 15-4

也就是说，在调用了 ActivityThread 的 main 方法之后，会依次执行如图 15-5 所示的流程。



▲图 15-5

这就是一个典型的模板方法，ActivityThread 的 main 函数被调用之后，依次执行 Activity 的 onCreate、onStart、onResume 函数，用户通常在 Activity 的子类中覆写 onCreate 方法，并且在该方法中调用 setContentView 来设置布局。从上文我们知道，在 onResume 之后，Activity 的布局内容

就显示到窗口上了，那么用户的布局视图是如何添加到 Activity 的呢？我们还是从源码中寻找答案。

首先看看 Activity 中的 setContentView 到底做了什么：

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

我们可以看到，实际上调用了 mWindow 的 setContentView 方法，本书中我们已经多次提到 Window 的实现类为 PhoneWindow，我们就看看 PhoneWindow 的 setConentView，其核心源码如下：

```
@Override
public void setContentView(int layoutResID) {
    if (mContentParent == null) {
        installDecor();          // 1. 生成 DecorView
    } else {
        mContentParent.removeAllViews();
}
// 2. 将 layoutResId 的布局添加到 mContentParent 中
    mLayoutInflater.inflate(layoutResID, mContentParent);
}

    // 构建 mDecor 对象,并且初始化标题栏和 Content Parent(我们要显示的内容区域)
    private void installDecor() {
    if (mDecor == null) {
        mDecor = generateDecor();            // 3. 构建 DecorView
        mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_DESCENDANTS);
        mDecor.setIsRootNamespace(true);
    }
if (mContentParent == null) {
    // 4. 获取 ContentView 容器，即显示内容的区域
        mContentParent = generateLayout(mDecor);
        mTitleView = (TextView)findViewById(com.android.internal.R.id.title);
    // 5. 设置 Title 等，代码省略
    }
}
    protected DecorView generateDecor() {
    return new DecorView(getContext(), -1);     // 构建 mDecor 对象
}
```

我们可以看到，setContentView 的基本流程简单概括就是如下几步。

（1）构建 mDecor 对象。mDecor 就是整个窗口的顶层视图，它主要包含了 Title 和 Content View 两个区域，Title 区域就是我们的标题栏，Content View 区域就是显示 xml 布局内容中的区域。

（2）设置一些关于窗口的属性，初始化标题栏区域和内容显示区域。

这里比较复杂的就是 generateLayout(mDecor)这个函数，我们一起继续深入分析：

```
// 返回用于显示我们设置的页面内容的 ViewGroup 容器
protected ViewGroup generateLayout(DecorView decor) {
    // 1. 获取窗口的 Style 属性
    TypedArray a = getWindowStyle();
    // 设置一些窗口属性等，代码省略

    // 2. 根据一些属性来选择不同的顶层视图布局,例如，设置了 FEATURE_NO_TITLE 的属性，那么就选择没有
    // Title 区域的那个布局
    // layoutResource 布局就是整个 Activity 的布局，其中含有 title 区域和 content 区域，content 区
    // 域就是用来显示通过 setContentView 设置进来的内容区域，也就是要显示的视图
```

```
    int layoutResource;   // 布局资源 id
    int features = getLocalFeatures();
    if ((features & ((1 << FEATURE_LEFT_ICON) | (1 << FEATURE_RIGHT_ICON))) != 0) {
        // 省略
    } else if ((features & (1 << FEATURE_NO_TITLE)) == 0) {
        if (mIsFloating) {
            layoutResource = com.android.internal.R.layout.dialog_title;
        } else {
            layoutResource = com.android.internal.R.layout.screen_title;
        }

    }
    mDecor.startChanging();
    // 3. 加载视图
    View in = mLayoutInflater.inflate(layoutResource, null);
    // 4. 将 layoutResource 的内容添加到 mDecor 中
    decor.addView(in, new ViewGroup.LayoutParams(MATCH_PARENT, MATCH_PARENT));
    // 5. 获取到内容显示区域，这是一个 ViewGroup 类型，其实是 FrameLayout
    ViewGroup contentParent = (ViewGroup)findViewById(ID_ANDROID_CONTENT);

    // 6. 设置一些背景、title 等属性 ，代码省略

    mDecor.finishChanging();        // 完成解析
    return contentParent;           // 返回 contentParent 布局，而不是整个 DecorView
  }
```

其步骤如下所示。

（1）获取用户设置的一些属性与 Flag。

（2）根据一些属性选择不同的顶层视图布局，如 FEATURE_NO_TITLE 则选择没有 title 的布局文件等。

这里我们选择 layoutResource =com.android.internal.R.layout.screen_title 的情形来分析，首先看看 screen_title.xml 的布局代码：

```
<?xml version="1.0" encoding="utf-8"?>
<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" />
    <!--  title 区域-->
    <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>
    <!--内容显示区域，例如 main_activity.xml 布局就会被放到这个 ViewGroup 下面 -->
    <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>
```

我们可以看到有两个区域，即 title 区域和 content 区域，generateLayout 函数获取 content 区域的代码如下：

```
// 1. 获取到我们的内容显示区域，这是一个 ViewGroup 类型的，其实是 FrameLayout
      ViewGroup contentParent = (ViewGroup)findViewById(ID_ANDROID_CONTENT);
```

获取的就是 xml 中 id 为 content 的 FrameLayout，这个 content 就是内容显示区域，整个布局对应的效果如图 15-6 所示。

▲图 15-6

这两个区域就组成了 mDecor 视图，我们的 main_activity.xml 就是放在内容视图这个区域的。

（3）加载顶层布局文件，转换为 View，将其添加到 mDecor 中。

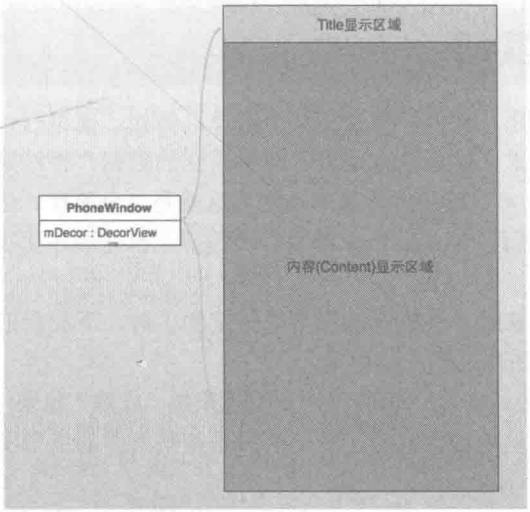
（4）获取内容容器 Content Parent，即用于显示内容的区域。

（5）设置一些背景图和 title 等。

经过这几步，我们就得到了 mContentParent，这就是用来装载视图的 ViewGroup，再回过头来看 setContentView 函数：

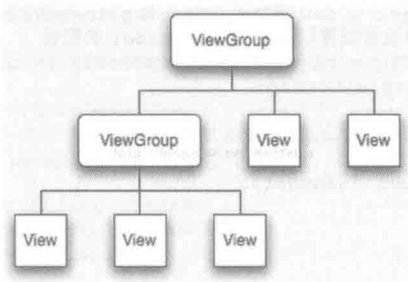
```
public void setContentView(int layoutResID) {
    if (mContentParent == null) {
    // 1. 生成 DecorView，并且根据窗口属性加载顶级视图布局
        // 获取 mContentParent、设置一些基本属性等
        installDecor();
    } else {
        mContentParent.removeAllViews();
    }
    // 2. 将 layoutResId 加载到 mContentParent 中
    // 这里的 layoutResId 就是 Activity 的 onCreate 函数执行的布局 id
    mLayoutInflater.inflate(layoutResID, mContentParent);
}
```

实际上就是将 layoutResId 这个布局的视图附加到 mContentParent 中，这个 layoutResId 就是开发人员在 onCreate 函数中设置给 Activity 的那个布局资源 id，也这是说 Google 预先设置了一些布局资源，这些布局资源里面有一个留给开发者们设置窗口内容的区域，也就是 content 区域，我们通过 setContentView 设置的布局会被添加到 content 布局中，这个 content 布局是一个 FrameLayout，因此，它是一个 ViewGroup，如图 15-7 所示。



▲图 15-7

ViewGroup 从语义上来说就是视图组，它也继承自 View 类，它其实就是视图的容器。Android 中通过 ViewGroup 来组织、管理子视图，例如常见的 FrameLayout、LinearLayout、ListView 等都是 ViewGroup 类型，总之，只要能包含其他 View 或者 ViewGroup 的都是 ViewGroup 类型。而 View 就是 UI 界面上的可见的组件类型，任何在 UI 上可见的都为 View 的子类，TextView、Button、ImageView、FrameLayout、LinearLayout、ListView 等都是 View 的子类。这样一来，Android 就可以使用 ViewGroup 和 View 来构建视图树，如图 15-8 所示。



▲图 15-8

这样，ViewGroup 类型的视图管理和嵌套在里面的 ViewGroup 以及 View 控件组成了丰富多彩的用户界面。

当 Activity 启动时，通过 onCreate 函数让用户设置自己的界面，系统将这个布局界面添加到一个内置的布局界面的 content 区域中，此时，DecorView 就建立起来了。然后调用 onStart 函数，并且在调用 onResume 函数之前将 DecorView 添加到 WindowManager 中，并且设置 Activity 为可见，然后通知 ActivityManagerService，该 Activity 已变为 resume 状态，使得系统能够渲染 Activity 的视图，至此，Activity 的视图就会显示在手机上了。

15.8 模板方法实战

做什么事都得有个顺序，软件开发也是一个道理，例如，要进行一个网络请求，那必须先创建请求，然后执行请求、解析返回的数据、将结果回调给 UI，不管如何变化，这 4 个整体步骤基本不变，这就是一个流程。还有上述所说的 Activity 的显示流程、AsyncTask 的执行流程等，都是顺序化执行的表现。

从上文的讲解中，大家对于模板方法也有了一定的了解，下面我们就以实战的角度来看看模板方法在开发中的运用。

小民的 ImageLoader 经过一步一步的重构变得越来越“成熟”起来，但总体来说，图片加载流程还是相对固定的，变化的只是某些具体的细节实现，加载流程大致如图 15-9 所示。

▲图 15-9

图片加载请求创建之后，首先检测是否有缓存，如果没有则从网络上下载该图片，并且将图片添加到缓存中，最好将图片分发给 UI 线程进行更新，这些流程被封装在 Loader 的内部，外部只需要调用对应的接口即可。

首先用户通过 ImageLoader 的 displayImage 接口来请求加载图片到某个 ImageView 中，相关代码如下：

```
    public final class SimpleImageLoader {
    // 代码省略

    public void displayImage(final ImageView imageView, final String uri,
    final DisplayConfig config, final ImageListener listener) {
        BitmapRequest request = new BitmapRequest(imageView, uri, config, listener);
        // 加载的配置对象,如果没有设置则使用 ImageLoader 的配置
        request.displayConfig = request.displayConfig != null ? request.displayConfig
                : mConfig.displayConfig;
        // 设置加载策略
        request.setLoadPolicy(mConfig.loadPolicy);
        // 添加到队列中
        mImageQueue.addRequest(request);
    }

}
```

在 displayImage 函数中会将 ImageView、图片 uri、listener 以及请求配置封装到一个 BitmapRequest 中，并且将这个请求添加到请求队列，等待加载线程的执行。在初始化 ImageLoader 时会默认启动 4 个图片加载线程，它们会在后台不断地从请求队列中获取请求，并且交给对应的 Loader 加载，这些图片加载线程我们命名为 RequestDispatcher，相关代码如下：

```
// 网络请求 Dispatcher,继承自 Thread,从网络请求队列中循环读取请求并且执行
final class RequestDispatcher extends Thread {
    @Override
    public void run() {
        try {
            while (!this.isInterrupted()) {
```

```
                // 从队列中获取请求
                final BitmapRequest request = mRequestQueue.take();
                if (request.isCancel) {
                    continue;
                }
                // 从图片的 uri 中获取 schema，如网络图片，那么 schema 为 http 或者 https
                final String schema = parseSchema(request.imageUri);
                // 根据图片 schema 来选择对应的 Loader
                Loader imageLoader = LoaderManager.getInstance().getLoader(schema);
                // 加载图片
                imageLoader.loadImage(request);
            }
        } catch (InterruptedException e) {
            Log.i("", "### 请求分发器退出");
        }
    }
}
```

在 RequestDispatcher 的 run 函数中会不断地从请求队列中获取请求，然后通过请求图片的 uri 获取到对应的 schema。这个 schema 就是一个字符串，例如，网络图片是以 http 或者 https 开头的，那么 schema 就为 http 或者 https。对于本地图片来说，它的 uri 格式是“file:// + 绝对路径”。因此，本地图片的 schema 就是 file://。这些 schema 与 Loader 的对应关系我们通过 LoaderManager 进行管理，相关代码如下：

```
// Loader 管理器，以图片的 uri 为 key、loader 为 value 构建一个表
// 使得调用端可以通过 schema 获取对应的 Loader
public class LoaderManager {
    public static final String HTTP = "http";
    public static final String HTTPS = "https";
    public static final String FILE = "file";
    // loader 表，图片 uri 的 schema 为 key，图片加载器为 value
    private Map<String, Loader> mLoaderMap = new HashMap<String, Loader>();
    // 默认加载器，NullObject Pattern
    private Loader mNullLoader = new NullLoader();
    // 代码省略
    // 构造函数中注册 3 种格式的 uri、loader
    private LoaderManager() {
        register(HTTP, new UrlLoader());
        register(HTTPS, new UrlLoader());
        register(FILE, new LocalLoader());
    }
    public Loader getLoader(String schema) {
        if (mLoaderMap.containsKey(schema)) {
            return mLoaderMap.get(schema);
        }
        return mNullLoader;
    }
}
```

LoaderManager 是一个单例，在构造 LoaderManager 实例时会默认注册 3 种形式的图片 uri 与 Loader 到 mLoaderMap 中，这样在外部需要 Loader 时只需通过 schema 就可以获取到，这个 Loader 就是负责加载图片的角色，例如从网络上加载、从本地加载等，AbsLoader 实现了 Loader 接口，并且将 loadImage 设置为 final，使得子类不可覆写该方法。更为重要的是，在 loadImage 中指定了图片加载的流程，也就是如图 15-8 所示的先判断是否有缓存，没有缓存再加载，加载成功添加到缓存，最后分发到调用端程序中，相关代码如下：

```
public interface Loader {
    public void loadImage(BitmapRequest request);
}
/**
 * 抽象 Loader，定义了加载图片的通用接口以及实现了通用功能，并且定义了图片加载流程
 */
public abstract class AbsLoader implements Loader {
    // 图片缓存
    private static BitmapCache mCache = SimpleImageLoader.getInstance().getImageCache();
    @Override
    public final void loadImage(BitmapRequest request) {
        // 1. 从缓存中获取图片
        Bitmap resultBitmap = mCache.get(request);
        // 没有缓存
        if (resultBitmap == null) {
            // 2. 显示图片加载中
            showLoading(request);
            // 3. 加载图片
            resultBitmap = onLoadImage(request);
            // 4. 缓存图片
            cacheBitmap(request, resultBitmap);
        } else {
            request.justCacheInMem = true;
        }
        // 5. 将图片分发给调用端（执行在 UI 线程）
        deliveryToUIThread(request, resultBitmap);
    }

    protected abstract Bitmap onLoadImage(BitmapRequest result);
}
```

AbsLoader 中的 loadImage 就是一个模板方法，它通过 loadImage 规定了加载图片的执行流程，并且将该方法设置为 final，这样子类就不能够破坏这个固定的执行流程，子类可以修改的就是具体的加载过程，这个工作由 onLoadImage 函数实现，例如，从网络上加载图片。由于 loadImage 函数是在 RequestDispatcher 的 run 函数中调用的，因此，onLoadImage 也是执行在子线程 RequestDispatcher 中，所以，onLoadImage 在这里是一个同步函数。

我们看看加载网络图片和加载本地图片两个类的简单实现，具体代码如下：

```
// 网络图片的 Loader
public class UrlLoader extends AbsLoader {

    @Override
    public Bitmap onLoadImage(BitmapRequest request) {
        InputStream is = null;
        Bitmap bitmap = null;
        try {
            URL url = new URL(request.imageUri);
            final HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            is = conn.getInputStream();
            bitmap = BitmapFactory.decodeStream(is);
            // 关闭连接
            conn.disconnect();
        } catch (Exception e) {
        } finally {
            IOUtil.closeQuietly(is);
        }
        return bitmap;
    }
```

```
}
// 加载本地图片的 Loader
public class LocalLoader extends AbsLoader {
    @Override
    public Bitmap onLoadImage(BitmapRequest request) {
        final String imagePath = Uri.parse(request.imageUri).getPath();
        final File imgFile = new File(imagePath);
        if (!imgFile.exists()) {
            return null;
        }
        // 从 SD 卡中加载的图片仅缓存到内存中,不做本地缓存
        request.justCacheInMem = true;

        // 加载图片
        return BitmapFactory.decodeFile(imagePath, null);

    }
}
```

这两个 Loader 继承自 AbsLoader，并且只实现了 onLoadImage 函数，UrlLoader 从网络上加载图片，因此，它需要建立 http 请求，读取网络上的输入流，通过这个输入流来构建图片。而 LocalLoader 则是加载本地图片，获取到图片的本地路径之后，通过 BitmapFactory.decodeFile 函数即可创建图片。当然，在上述代码中为了更简洁，我们省略了对图片进行缩放操作，图片缩放是很重要的一步，因此，如果没有这一步很可能会造成 OOM，而且如果图片很多，要显示的 ImageView 很小，此时将图片完整地加载进内存也是很消耗资源的。

在 onLoadImage 从具体的来源加载了图片之后，最终会调用 AbsLoader 中的 deliveryToUIThread (request, resultBitmap)函数，该函数是将加载得到的图片分发给调用端，因此，我们在构建 BitmapRequest 时就有一个 Listener 参数，这个参数就是调用端传递进来的，此时获取到 Bitmap 之后，就可以将该 listener 的回调函数执行到 UI 线程中，这样就使得调用端可以在回调函数中直接更新 UI 了，至此，整个加载过程结束。

本章只关注 AbsLoader 的加载流程，也就是 loadImage 中的执行过程，大致分为如下几步：

（1）判断是否有缓存；

（2）如果没有缓存则加载图片，并且再缓存起来；

（3）将图片分发给调用端，这个过程执行在 UI 线程。

LoadImage 函数是 final 的，也就是子类不允许修改执行流程，而只能修改具体的某一步。在该示例中，子类能够定制化的就是 onLoadImage 函数，也就是具体的图片加载函数，例如，UrlLoader 从网络上加载、LocalLoader 从本地文件加载，这样通过封装流程，使得固定的执行顺序不能被子类随意修改，而子类只能替换掉某些具体的算法策略，不仅能够复用父类的代码，也能避免功能遭到破坏。对外部客户端程序来说，模板方法模式隐藏了具体的执行流程，对外暴露一个统一的接口，使得客户端程序的使用成本更小，接口更简洁。

15.9 总结

模板方法模式用 4 个字概括就是：流程封装。也就是把某个固定的流程封装到一个 final 函数中，并且让子类能够定制这个流程中的某些或者所有步骤，这就要求父类提取共用的代码，提升代

码的复用率，同时也带来了更好的可扩展性。

优点

（1）封装不变部分，扩展可变部分。
（2）提取公共部分代码，便于维护。

缺点

模板方法会带来代码阅读的难度，会让用户觉得难以理解。

第 16 章　访问者模式

16.1 访问者模式介绍

访问者模式是一种将数据操作与数据结构分离的设计模式，它是《设计模式》中 23 种设计模式中最复杂的一个，但它的使用频率并不高，正如《设计模式》的作者 GOF 对访问者模式的描述：大多数情况下，你并需要使用访问者模式，但是当你一旦需要使用它时，那你就是真地需要它了。

访问者模式的基本想法是，软件系统中拥有一个由许多对象构成的、比较稳定的对象结构，这些对象的类都拥有一个 accept 方法用来接受访问者对象的访问。访问者是一个接口，它拥有一个 visit 方法，这个方法对访问到的对象结构中不同类型的元素作出不同的处理。在对象结构的一次访问过程中，我们遍历整个对象结构，对每一个元素都实施 accept 方法，在每一个元素的 accept 方法中会调用访问者的 visit 方法，从而使访问者得以处理对象结构的每一个元素，我们可以针对对象结构设计不同的访问者类来完成不同的操作，达到区别对待的效果。

16.2 访问者模式的定义

封装一些作用于某种数据结构中的各元素的操作，它可以在不改变这个数据结构的前提下定义作用于这些元素的新的操作。

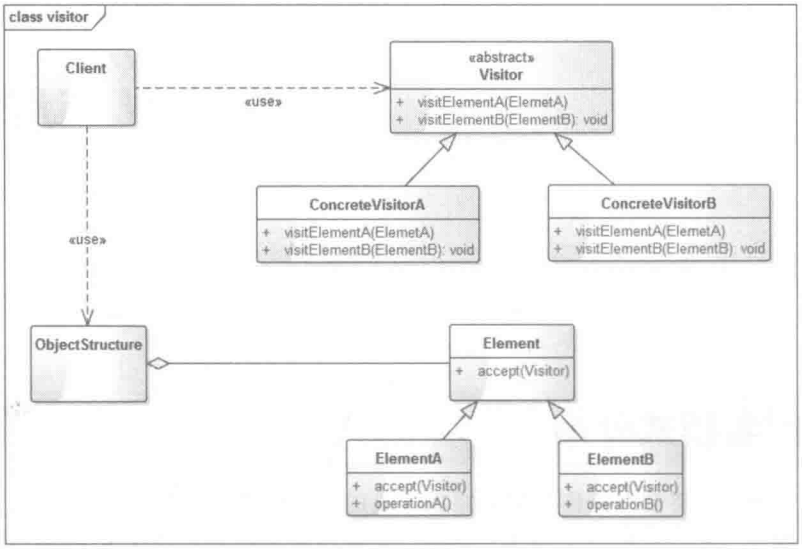
16.3 访问者模式的使用场景

（1）对象结构比较稳定，但经常需要在此对象结构上定义新的操作。

（2）需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免这些操作“污染”这些对象的类，也不希望在增加新操作时修改这些类。

16.4 访问者模式的 UML 类图

UML 类图如图 16-1 所示。



▲图 16-1

角色介绍。

- Visitor：接口或者抽象类，它定义了对每一个元素（Element）访问的行为，它的参数就是可以访问的元素，它的方法个数理论上来讲与元素个数是一样的，因此，访问者模式要求元素的类族要稳定，如果经常添加、移除元素类，必然会导致频繁地修改 Visitor 接口，如果出现这种情况，则说明不适合使用访问者模式。
- ConcreteVisitor：具体的访问者，它需要给出对每一个元素类访问时所产生的具体行为。
- Element：元素接口或者抽象类，它定义了一个接受访问者（accept）的方法，其意义是指每一个元素都要可以被访问者访问。
- ElementA、ElementB：具体的元素类，它提供接受访问方法的具体实现，而这个具体的实现，通常情况下是使用访问者提供的访问该元素类的方法。
- ObjectStructure：定义当中所提到的对象结构，对象结构是一个抽象表述，它内部管理了元素集合，并且可以迭代这些元素供访问者访问。

16.5 访问者模式的简单示例

访问者模式是一个结构、概念都较为复杂的模式，使用频率也不高，但是并不代表可以将它的作用忽略，当有合适的场景时，访问者模式会带来意想不到的灵活性，所以，我们首先通过一个简单的示例来学习它的典型应用。

在年终时，公司都会给员工进行业绩考核，以此来评定该员工的绩效及年终奖、晋升等，这些评定都是由公司高层来负责，但是，不同领域的管理人员对于员工的评定标准是不一样的，为了简单明了地说明问题，我们把员工简单分为工程师和经理，评定员工的分别为 CEO 和 CTO，我们假定 CTO 只关注工程师的代码量、经理的新产品数量，而 CEO 关注的是工程师的 KPI 和经理的 KPI

以及新产品数量，从中可以看出，CEO 和 CTO 对于不同员工的关注点是不一样的，这就需要对不同的员工类型进行不同的处理。访问者模式此时可以派上用场了，我们看看下面相关代码的实现：

```
// 员工基类
public abstract class Staff {
    public String name;
    // 员工 kpi
    public int kpi;

    public Staff(String aName) {
        this.name = aName;
        kpi = new Random().nextInt(10);
    }
    // 接受 Visitor 的访问
    public abstract void accept(Visitor visitor);
}
```

Staff 类定义了员工的基本信息及一个 accept 方法，accept 方法表示接受访问者的访问，由子类具体实现。下面看看工程师和经理的相关代码：

```
// 工程师
public class Engineer extends Staff {
    public Engineer(String aName) {
        super(aName) ;
    }
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
    // 工程师这一年写的代码数量
    public int getCodeLines() {
        return new Random().nextInt(10 * 10000);
    }
}
// 经理类型
public class Manager extends Staff {
    private int products;  // 产品数量
    public Manager(String aName) {
        super(aName);
        products = new Random().nextInt(10);
    }
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
    // 一年内做的产品数量
    public int getProducts() {
        return products;
    }
}
```

在工程师类中添加了获取代码行数的函数，而在经理类型中则添加了获取新产品数量的函数，它们的职责是不一样的，也正是由于它们的差异性才使得访问者模式能够发挥它的作用。Staff、Engineer、Manager 3 个类型就是对象结构，这些类型相对稳定，不会发生变化。

然后将这些员工添加到一个业务报表类中，公司高层可以通过该报表类的 showReport 函数查看所有员工的业绩，具体代码如下：

```
// 员工业务报表 1 类
public class BusinessReport {
    List<Staff> mStaffs = new LinkedList<Staff>();

    public BusinessReport() {
        mStaffs.add(new Manager("王经理"));
        mStaffs.add(new Engineer("工程师-Shawn.Xiong"));
        mStaffs.add(new Engineer("工程师-Kael"));
        mStaffs.add(new Engineer("工程师-Chaossss"));
        mStaffs.add(new Engineer("工程师-Tiiime"));
    }


    /**
     * 为访问者展示报表
     * @param visitor 公司高层，如 CEO、CTO
     */
    public void showReport(Visitor visitor) {
        for (Staff staff : mStaffs) {
            staff.accept(visitor);
        }
    }
}
```

下面看看 Visitor 类型的定义，Visitor 声明了两个 visit 函数，分别是对工程师和经理的访问函数，具体代码如下：

```
public interface Visitor {
    // 访问工程师类型
    public void visit(Engineer engineer);
    // 访问经理类型
    public void visit(Manager leader);
}
```

首先定义了一个 Visitor 接口，该接口有两个 visit 函数，参数分别为 Engineer、Manager，也就是说对于 Engineer、Manager 的访问会调用两个不同的方法，以此达成区别对待、差异化处理。具体的实现类为 CEOVisitor、CTOVisitor 类，具体代码如下：

```
// CEO 访问者,只关注业绩
public class CEOVisitor implements Visitor {
    @Override
    public void visit(Engineer engineer) {
        System.out.println("工程师 : " + engineer.name + ", KPI : " + engineer.kpi);
    }

    @Override
    public void visit(Manager mgr) {
        System.out.println("经理 : " + mgr.name + ", KPI : " + mgr.kpi + ", 新产品数量 : "
            + mgr.getProducts());
    }
}
```

在 CEO 的访问者中，CEO 只关注 Engineer 员工的 KPI，而对于 Manager 类型的员工除了 KPI 之外还有该 Manager 本年度新开发产品的数量。两类员工的关注点略有不同，通过两个 visitor 方法分别进行处理。而如果我们不使用 Visitor 模式，只通过一个 visit 函数进行处理，那么就需要在这个 visit 函数中对不同的员工类型进行判断，然后分别进行处理，代码大致如下：

```
class ReportUitl {
```

```
    public void visit(Staff staff) {
        if (staff instanceof Manager) {
            Manager mgr = (Manager) staff;
                System.out.println("经理 : " + mgr.name + ", KPI : " + mgr.kpi + ", 新产品
                    数量 : "+ mgr.getProducts());
        } else {
                Engineer engineer = (Engineer) staff;
                System.out.println("工程师 : " + engineer.name + ", KPI : " + engineer.kpi);
        }
    }
}
```

这就导致了 if-else 逻辑的嵌套以及类型的强制转换，难以扩展和维护，当类型较多时，这个 ReportUtil 就会很复杂。而使用 Visitor 模式，通过同一个函数对不同的元素类型进行相应的处理，使得结构更加清晰、灵活性更高。

例如，CEO 的关注点主要是在业绩上，而 CTO 更关注的是员工在技术层面的贡献，因此，我们添加一个 CTO 的 Visitor 类：

```
public class CTOVisitor implements Visitor {
    @Override
    public void visit(Engineer engineer) {
        System.out.println("工程师 :" + engineer.name + ",代码函数:" + engineer.getCodeLines());
    }
    @Override
    public void visit(Manager leader) {
        System.out.println("经理 : " + leader.name + "产品数量 : " + leader.getProducts());
    }
}
```

CTO 的 Visitor 中只关注工程师写的代码量，而经理只关注他的新产品数量。不同的 visit 函数会对元素进行不同的操作，而通过注入不同的 Visitor 又可以替换掉访问者的具体实现，使得对元素的操作变得更灵活，可扩展性更高，同时也消除了类型转换、if-else 等“丑陋”的代码。

下面是客户端代码：

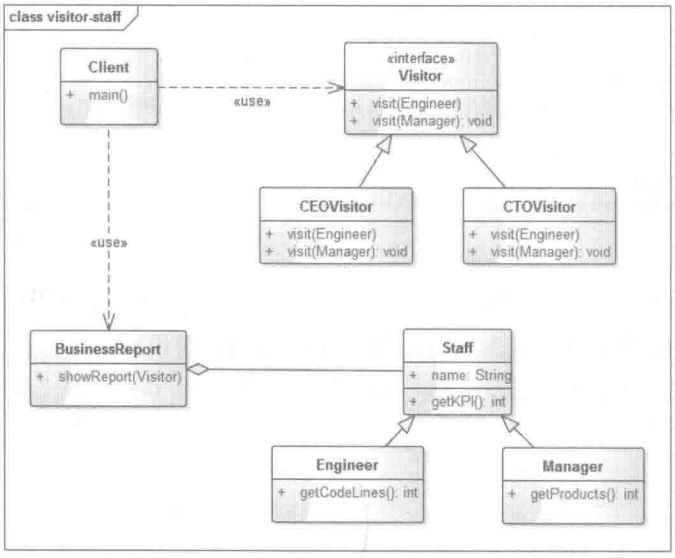
```
    public class Client {
    public static void main(String[] args) {
        // 构建报表
        BusinessReport report = new BusinessReport();
        System.out.println("======== 给 CEO 看的报表 ====== ");
        // 设置访问者,这里是 CEO
        report.showReport(new CEOVisitor());
        System.out.println("\n======== 给 CTO 看的报表 ====== ");
        // 注入另一个访问者,CTO
        report.showReport(new CTOVisitor());
    }
}
```

客户端代码中，首先构建了一个报表对象，该对象中维护了所有员工的集合，然后通过报表类的 showReport 函数为 Visitor 对象提供一个访问接口，在这个函数中遍历所有的员工，然后调用员工的 accept 函数接受访问者的访问，每个访问者对不同类型的员工调用对应的 visit 函数实现不同的操作。

具体输出如下。

```
======== 给 CEO 看的报表 ======
经理 : 王经理, KPI : 8, 新产品数量 : 1
工程师 : 工程师-Shawn.Xiong, KPI : 3
工程师 : 工程师-Kael, KPI : 2
工程师 : 工程师-Chaossss, KPI : 0
工程师 : 工程师-Tiiime, KPI : 6

======== 给 CTO 看的报表 ======
经理 : 王经理产品数量 : 1
工程师 : 工程师-Shawn.Xiong, 代码函数 : 88887
工程师 : 工程师-Kael, 代码函数 : 47415
工程师 : 工程师-Chaossss, 代码函数 : 41141
工程师 : 工程师-Tiiime, 代码函数 : 17252
```

在上述示例中，Staff 扮演了 Element 角色，而 Engineer 和 Manager 都是 ConcreteElement；CEOVisitor 和 CTO 都是具体的 Visitor 对象；而 BusinessReport 就是 ObjectStructure；Client 就是客户端代码。它们的结构如图 16-2 所示。



▲图 16-2

访问者模式最大的优点就是增加访问者非常容易，我们从代码中可以看到，如果要增加一个访问者，你新创建一个实现了 Visitor 接口的类，然后实现两个 visit 函数来对不同的元素进行不同的操作，从而达到数据对象与数据操作相分离的效果。如果不使用访问者模式，而又想对不同的元素进行不同的操作，那么必定需要使用 if-else 和类型转换，这使得代码难以升级维护。此时，访问者模式的作用便体现出来了。

16.6 Android 源码中的模式

在现阶段的 Android 开发中，注解越来越流行起来，按照处理时期，注解又可以分为两种类型，一种是运行时注解，另一种是编译期注解。运行时注解由于性能问题被一些人所诟病，编译注解的

核心原理依赖 APT（Annotation Processing Tools）实现，例如，著名的 ButterKnife、Dagger、Retrofit 等开源库都是基于 APT。那么编译期注解是如何工作的？它与访问者模式又有什么样的关系？这也是我们本章要讨论的。

编译时 Annotation 解析的基本原理是，在某些代码元素上（如类型、函数、字段等）添加注解，在编译时编译器会检查 AbstractProcessor 的子类，并且调用该类型的 process 函数，然后将添加了注解的所有元素都传递到 process 函数中，使得开发人员可以在编译期进行相应的处理，例如，根据注解生成新的 Java 类，这也就是 ButterKnife 等开源库的基本原理。

在编译处理的时候，是分开进行的。如果在某个处理中产生了新的 Java 源文件，那么就需要另外一个处理来处理新生成的源文件，如此往复，直到没有新文件被生成为止。在完成处理之后，再对 Java 代码进行编译。JDK 5 中提供了 APT 工具用来对注解进行处理。APT 是一个命令行工具，与之配套的还有一套用来描述程序语义结构的 Mirror API，Mirror API（com.sun.mirror.*）描述的是程序在编译时的静态结构，通过 Mirror API 可以获取到被注解的 Java 类型元素的信息，从而提供相应的处理逻辑，具体的处理工作交给 APT 工具来完成。编写注解处理器的核心是 AnnotationProcessorFactory 和 AnnotationProcessor 两个接口，后者表示的是注解处理器，而前者则是为某些注解类型创建注解处理器的工厂。

对于编译器来说，代码中的元素结构是基本不变的，例如，组成代码的基本元素有包、类、函数、字段、类型参数、变量。JDK 中为这些元素定义了一个基类，也就是 Element 类，它有如下几个子类：

（1）PackageElement——包元素，包含了某个包下的信息，可以获取到包名等；

（2）TypeElement——类型元素，如某个字段属于某种类型；

（3）ExecutableElement——可执行元素，代表了函数类型的元素；

（4）VariableElement——变量元素；

（5）TypeParameterElement——类型参数元素。

因为注解可以指定作用在哪些元素上，因此，通过上述的抽象来对应这些元素，例如下面这个注解，指定的是只能作用于函数上面，并且这个注解只能保留在 class 文件中：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.CLASS)
@interface Test {
    String value();
}
```

该注解因为只能作用于函数类型，因此，它对应的元素类型就是 ExecutableElement，当我们想通过 APT 处理这个注解时就可以获取目标对象上的 Test 注解，并且将所有这些元素转换为 ExecutableElement 元素，以便获取到它们对应的信息。

我们看看元素基类的实现，完整的类路径为 javax.lang.model.element.Element：

```
public interface Element {
    // 代码省略
    // 获取元素类型
    ElementKind getKind();

    // 获取修饰符，如public、static、final 等
```

```
    Set<Modifier> getModifiers();

    // 接受访问者的访问
<R, P> R accept(ElementVisitor<R, P> v, P p);
}
```

我们看到 Element 定义了一个代码元素的一些通用接口，其中很显眼的就是 accept 函数，这个函数接收一个 ElementVisitor 和类型为 P 的参数，ElementVisitor 就是访问者类型，而 P 则用于传递一些额外的参数给 Visitor，到这里读者应该明白访问者模式了，我们继续看看 ElementVisitor 的定义：

```
// 元素访问者
public interface ElementVisitor<R, P> {
    //访问元素
    R visit(Element e, P p);
    //访问包元素
    R visitPackage(PackageElement e, P p);
    //访问一个类型元素
    R visitType(TypeElement e, P p);
    //访问一个变量类型
    R visitVariable(VariableElement e, P p);
    //访问一个可执行的元素
    R visitExecutable(ExecutableElement e, P p);
    //访问一个参数元素
    R visitTypeParameter(TypeParameterElement e, P p);
    // 处理未知的元素类型，这是为了应对后续 Java 语言的扩展而预留的接口
    // 例如后续元素类型增加了，那么通过这个接口就可以处理上述没有声明的类型
    R visitUnknown(Element e, P p);
}
```

在 ElementVisitor 中定义了多个 visit 接口，每个接口处理一种元素类型，这就是典型的访问者模式！我们知道，一个类元素和函数元素是完全不一样的，它们的结构不一样，因此，编译器对它们的操作肯定也是不一样，通过访问者模式正好可以解决数据结构与数据操作分离的问题，避免某些操作“污染”数据对象类。但是，与经典访问者不同的是，ElementVisitor 预留了一个 visitUnknown 函数来应对元素结构的变化，例如，后续 Java 语言中添加了某个新的元素类型叫做 NewElement，那么这个元素就会被 visitUnknown 函数访问，此时 JDK 只需继承现有的 Visitor 类，然后覆写 visitUnknown 函数并且对 NewElement 进行相应的处理即可应对变化。经典模式中要求的是元素对象结构稳定，但是软件的本质就是变化，JDK 的这种设计恰好是灵活运用设计模式的优秀示例，通过添加一个预留函数来应对未来可能的元素结构变化，增强了可扩展性。

当 Visitor 对元素结构进行访问时就可以针对不同的类型进行不同的处理，例如 SimpleElementVisitor6 就是其中的一个访问者，它基本上没做什么操作，直接返回了元素的默认值，具体代码如下：

```
@SupportedSourceVersion(RELEASE_6)
public class SimpleElementVisitor6<R, P> extends AbstractElementVisitor6<R, P> {
    protected final R DEFAULT_VALUE;
    protected SimpleElementVisitor6(R defaultValue){
        DEFAULT_VALUE = defaultValue;
    }
    protected R defaultAction(Element e, P p) {
        return DEFAULT_VALUE;
    }
    public R visitType(TypeElement e, P p) {
```

```
        return defaultAction(e, p);
    }
    public R visitExecutable(ExecutableElement e, P p) {
        return defaultAction(e, p);
    }
    // 代码省略
}
```

另一个是提取元素类型的访问者是 ElementKindVisitor6，它的相关代码如下所示：

```
@SupportedSourceVersion(RELEASE_6)
public class ElementKindVisitor6<R, P>
                extends SimpleElementVisitor6<R, P> {
    // 代码省略
    // 访问类型元素，如类、注解、枚举、接口
    @Override
    public R visitType(TypeElement e, P p) {
        ElementKind k = e.getKind();
        switch(k) {
        case ANNOTATION_TYPE:
            return visitTypeAsAnnotationType(e, p);
        case CLASS:
            return visitTypeAsClass(e, p);
        case ENUM:
            return visitTypeAsEnum(e, p);
        case INTERFACE:
            return visitTypeAsInterface(e, p);
        default:
            throw new AssertionError("Bad kind " + k + " for TypeElement" + e);
        }
    }
}
```

ElementKindVisitor6 对于不同的类型进行不同的处理，提取各元素的类型信息，例如，上述代码中对于 Type 类型的元素将分别进行处理，如类、枚举、接口等。但是，由于这些元素以及访问者的具体实现都是 Oracle 未开放的部分，因此，具体实现细节无法看到，但访问者模式的结构已经出来了，我们也不必过多地追求细节。

首先，编译器将代码抽象成一个代码元素的树，然后在编译时对整棵树进行遍历访问，每个元素都有一个 accept 函数接受访问者的访问，每个访问者中有对应的 visit 函数，例如，visitType 函数就是对类型元素的访问，在每个 visit 函数中对不同的类型进行不同的处理，这样就达到了差异处理的效果，同时将数据结构与数据操作分离，使得每个类型的职责单一，易于升级维护。JDK 还特意预留了 visitUnknown 接口来应对 Java 语言后续发展可能添加元素类型的问题，灵活地将访问者模式的缺点化解。

16.7 访问者模式实战

由于访问者模式在开发中运用得并不多，因此，我们实战部分不打算以访问者模式为主题，而是以更具实战意义的编译期注解，也就是 APT 为例。访问者模式的示例大家可以参考简单实现部分的业务报表实现，再结合 APT 中的元素、访问者进行理解即可。下面就一起来看看编译期注解的运用。

在 Android 开发中，我们接触的比较多的编译期注解的库应该是 Jake Wharton 的 ButterKnife 库，这个库是针对 View、资源 ID 等进行注解的开源库，它能够去除掉一些“丑陋不堪”的样板式代码，使得我们的代码更加简洁、易于维护，同时基于 APT 也使得它的效率得到保证。ButterKnife 是针对 View 等进行注解的开源库，如果要对对象进行注解则可以使用 Square 公司的 Dagger 开源库，目前，该库已经交由 Google 进行维护，它的新名字为 Dagger2。

首先我们是在没有使用 ButterKnife 时，初始化一个 Activity 中的各个控件的代码：

```
public class ExampleActivity extends Activity {
    TextView title;
    ImageViewicon;
    TextView footer;

    @Override public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simple_activity);
        // 通过 findViewById 进行视图查找，然后进行类型转换
        title = (TextView) findViewById(R.id.title);
        icon = (ImageView) findViewById(R.id.icon);
        footer = (TextView) findViewById(R.id.footer);
    }
}
```

在 ExampleActivity 函数里的 onCreate 函数中，通常会对各个子视图进行初始化，这些代码看起来重复性很高，而且“丑陋不堪”，当一个布局中含有 10 个以上的 View 时，再加上为某些 View 添加上事件处理等，这部分的代码将占用大的篇幅。ButterKnife 就是为了简化这些工作而出现的，让开发人员专注在真正有用的代码上，使用 ButterKnife 之后我们的代码变成了这样：

```
public class ExampleActivity extends Activity {
    @InjectView(R.id.title) TextView title;
    @InjectView(R.id.icon) ImageView icon;
    @InjectView(R.id.footer) TextView footer;
    @Override public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simple_activity);
        // 将 Activity 注入 ButterKnife
        ButterKnife.inject(this);
    }
}
```

当运行完 onCreate 函数之后，Activity 中的几个 View 就已经被初始化了。findViewById、强制转换等样板代码被去除了，代码变得更加简单，使得我们可以更专注在代码逻辑的编写上，整个类型也更易于维护。我们不禁要问 ButterKnife 的原理是什么呢？@InjectView 又是什么？ButterKnife 的 inject 函数又有什么作用？

上文我们说过，在代码编译时会扫描 AbstractProcessor 的所有子类，并且调用这些子类的 process 函数，这个函数就会将所有的代码元素传递进来，此时只需要在这个 process 函数中获取所有添加了某个注解的元素，然后对这些元素进行操作，使之能够满足我们的需求，这样我们就可以在编译期对源代码进行处理，例如生成新的类等。在运行时，我们通过一些接口对这些新生成的类进行调用以此完成我们的功能。

说了这么多还是太抽象了，还是以小民编写的程序为例来为大家一一解答。

小民自从知道 ButterKnife 之后也被它的魅力所吸引，于是决定研究一下，经过一番学习得知，ButterKnife 是基于编译时注解，然后通过 APT 生成辅助类，在运行时通过 inject 函数调用那些生成的辅助类来完成功能。小民决定自己写一个只支持 View 的 ID 注入的简化版 ButterKnife 来深入学习，这个库被命名为 SimpleDagger。

首先小民建立了一个注解类，具体代码如下：

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.CLASS)
public @interface ViewInjector {
    int value();
}
```

因为我们的这个注解只支持 View 的 ID 注入，因此，它的目标元素是字段，它只存在于 class 文件中，因为，一旦过了编译期就不再需要它了。关于注解方面的基础知识我们不做过多讲解，对这方面不了解的读者可以先阅读相关书籍，例如《Java 编程思想》、《Java 核心技术》。

添加这个注解之后，还需要在编译期对这个注解进行处理。上文说到，编译器会在编译时检测所有的 AbstractProcessor，并且调用它的 process 函数来让开发人员对代码元素进行处理，因此，新创建一个 AbstractProcessor 的子类，具体代码如下：

```
    @SupportedAnnotationTypes("org.simple.injector.anno.*")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class ViewInjectorProcessor extends AbstractProcessor {
    //所有注解处理器的列表
    List<AnnotationHandler> mHandlers = new LinkedList<AnnotationHandler>();
    //类型与字段的关联表,用于在写入 Java 文件时按类型来写不同的文件和字段
    final Map<String,List<VariableElement>>map=newHashMap<String, List<VariableElement>>();
    // 生成辅助类的 Writer 类
    AdapterWriter mWriter;

    @Override
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        // 注册注解处理器
        registerHandlers();
        // 初始化代码生成器
        mWriter = new DefaultJavaFileWriter(processingEnv);
    }
    // 注册处理器
    private void registerHandlers() {
        mHandlers.add(new ViewInjectHandler());
    }
    @Override
    public boolean process(Set<? extends TypeElement>annotations,RoundEnvironment roundEnv){
        // 迭代所有的注解处理器，使得每个注解都有一个处理器
        for (AnnotationHandler handler : mHandlers) {
            // 关联 ProcessingEnvironment
            handler.attachProcessingEnv(processingEnv);
            // 解析注解相关的信息
            map.putAll(handler.handleAnnotation(roundEnv));
        }
        // 将解析到的数据写入到具体的类型中
        mWriter.generate(map);
        return true;
    }
    // 代码省略
```

```
}
```

在 ViewInjectorProcessor 类的上面我们看到如下注解@SupportedAnnotationTypes("org.simple.injector.anno.*")，这个注解表明这个类只支持 org.simple.injector.anno 路径下的注解，在该类的 init 函数中我们注册了一个注解处理器，也就是 ViewInjectHandler 类，该类实现了 AnnotationHandler 接口，该接口的声明如下。

```
// 注解处理接口
public interface AnnotationHandler {
    // 关联 ProcessingEnvironment
    void attachProcessingEnv(ProcessingEnvironment processingEnv);
    // 处理注解，将结果存储到 Map 中
    Map<String, List<VariableElement>>handleAnnotation(RoundEnvironment env);
}
```

该接口声明了两个函数，一个是关联 ProcessingEnvironment，另一个是 handleAnnotation 函数，负责处理标识了 ViewInjector 注解的元素。小民的设计思路是定义一个 AnnotationHandler 接口，每个实现类处理一种类型的注解，例如 ViewInjectHandler 只处理 ViewInject 注解。下面我们看看 ViewInjectHandler 的核心代码：

```
public class ViewInjectHandler implements AnnotationHandler {
    ProcessingEnvironment mProcessingEnv;
    @Override
    public void attachProcessingEnv(ProcessingEnvironment processingEnv) {
        mProcessingEnv = processingEnv;
    }
    @Override
    public Map<String,List<VariableElement>>handleAnnotation(RoundEnvironment roundEnv){
        Map<String, List<VariableElement>> annotationMap=new HashMap<String,
        List<VariableElement>>();
        // 1. 获取使用 ViewInjector 注解的所有元素
        Set<?extends Element>elementSet= roundEnv.getElementsAnnotatedWith(ViewInjector.class);
        for (Element element : elementSet) {
            // 2. 获取被注解的字段
            VariableElement varElement = (VariableElement) element;
            // 3. 获取字段所在类型的完整路径名，如一个 TextView 所在的 Activity 的完整路径，也就是变量的宿主类
            String className = getParentClassName(varElement);
            // 4. 获取这个宿主类型的所有元素，如某个 Activity 中的所有注解对象
            List<VariableElement> cacheElements = annotationMap.get(className);
            if (cacheElements == null) {
                cacheElements = new LinkedList<VariableElement>();
            }
            // 将元素添加到该类型对应的字段列表中
            cacheElements.add(varElement);
            // 以宿主类的路径为 key，所有字段列表为 value，存入 map
            // 这里是将所在字段按所属的类型进行分类
            annotationMap.put(className, cacheElements);
        }

        return annotationMap;
    }
    // 代码省略
}
```

在 handleAnnotation 函数中小民获取了所有被 ViewInject 注解标识了的 VariableElement 元素，然后将这些元素按照宿主类进行分类存到一个 map 中，key 就是宿主类的完整类路径，value 就是

这个宿主类中的所有被标识了 ViewInject 的 VariableElement 元素列表，例如，将上述 ExampleActivity 的示例替换成小民的 SimpleDagger，使用 ViewInject 注解标识 3 个 View，具体代码如下：

```
package com.simple.apt;
public class ExampleActivity extends Activity {
  @ViewInject (R.id.title) TextView title;
  @ViewInject (R.id.icon) ImageView icon;
  @ViewInject (R.id.footer) TextView footer;
  @Override public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.simple_activity);
// 其他代码暂时省略
SimpleDagger.inject(this);
  }
}
```

那么此时 ExampleActivity 的完整路径为 com.simple.apt.ExampleActivity，这个完整路径可以通过 VariableElement 元素获取到，这些 VariableElement 就是代表了 ExampleActiivty 中的 title、icon、footer 三个对象。因此，通过 ViewInjectHandler 的 handleAnnotation 处理之后，map 中就含有了以 com.simple.apt.ExampleActivity 为 key，以 title、icon、footer 三个成员变量对应的 VariableElement 列表为 value 的数据。

此时执行到 process 函数的最后一步，这里调用了 AdapterWriter 来生成辅助类，这个辅助类要生成的代码素材就是我们上述的 VariableElement 元素列表，调用的是 AdapterWriter 的 generate 函数，在 AdapterWriter 之下还建立了一个 AbsWriter 来封装一些通用逻辑，AbsWriter 核心代码如下：

```
public abstract class AbsWriter implements AdapterWriter {
    ProcessingEnvironment mProcessingEnv;
    Filer mFiler;
    // 代码省略
    @Override
    public void generate(Map<String, List<VariableElement>> typeMap) {
        Iterator<Entry<String,List<VariableElement>>>iterator=typeMap.entrySet().iterator();
        while (iterator.hasNext()) {
            Entry<String, List<VariableElement>> entry = iterator.next();
            List<VariableElement> cacheElements = entry.getValue();
            if (cacheElements == null || cacheElements.size() == 0) {
                continue;
            }

            // 取第一个元素来构造注入信息
            InjectorInfo info = createInjectorInfo(cacheElements.get(0));
            Writer writer = null;
            JavaFileObject javaFileObject;
            try {
                // 1. 创建源文件，也就是生成辅助类
                javaFileObject = mFiler.createSourceFile(info.getClassFullPath());
                writer = javaFileObject.openWriter();
                // 2. 写入 package、import、class 以及 findViews 函数等代码段
                generateImport(writer, info);
                // 3. 写入该类中的所有字段到 findViews 方法中
                for (VariableElement variableElement : entry.getValue()) {
                    writeField(writer, variableElement, info);
                }
                // 4. 写入 findViews 函数的大括号以及类的大括号
                writeEnd(writer);
```

```
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                IOUtil.closeQuitly(writer);
            }

        }
    }
    // 代码省略
}
```

在 AbsWriter 的 generate 函数中，定义了一个生成辅助类的逻辑骨架，分别为获取宿主类型的所有元素，并且通过第一个元素获取宿主类所在的包以及构建辅助类的类名等，然后创建一个新的 Java 类，最后，按照类的格式向辅助类中写入 import、所有被注解标识的元素的相应代码，所有生成的辅助类都是 InjectAdapter 的子类。实现功能代码在 DefaultJavaFileWriter 类中，核心代码如下：

```
public class DefaultJavaFileWriter extends AbsWriter {
    // 代码省略
    // 写入 import 以及类前面的类型声明
    @Override
    protected void generateImport(Writer writer, InjectorInfo info)
            throws IOException {
        writer.write("package " + info.packageName + " ;");
        writer.write("\n\n");
        writer.write("import org.simple.injector.adapter.InjectAdapter ;");
        writer.write("\n");
        writer.write("import org.simple.injector.util.ViewFinder;");

        writer.write("\n\n\n");
        writer.write("/* This class is generated by Simple ViewInjector,please don't modify! */ ");
        writer.write("\n");
        writer.write("public class " + info.newClassName
                + " implements InjectAdapter<" + info.classlName + "> { ");
        writer.write("\n");
        writer.write("\n");
        // 查找方法
        writer.write("  public void injects(" + info.classlName
                + " target)  { ");
        writer.write("\n");
    }

    // 写入一个结尾的大括号
    @Override
    protected void writeEnd(Writer writer) throws IOException {
        writer.write("  }");
        writer.write("\n\n");
        writer.write(" } ");
    }

    // 写入字段
    @Override
    protected void writeField(Writer writer, VariableElement element, InjectorInfo info)
            throws IOException {
        ViewInjector injector = element.getAnnotation(ViewInjector.class);
        String fieldName = element.getSimpleName().toString();
        writer.write(" target." + fieldName + " = ViewFinder.findViewById(target, "
                + injector.value() + "  ) ; ");
        writer.write("\n");
    }
```

```
}
```

在 DefaultJavaFileWriter 中分别写入了辅助类的各个部分，最终是写入字段的部分，也就是 writeField 函数，在该函数中，小民获取了这个字段的名字，并且写了如下一行代码：

```
target.fieldName = ViewFinder.findViewBydId(target, ViewInject 注解的值);
```

其实这就是一个初始化某个 View 的语句，target 在这个例子中就是 ExampleActivity，这个 ViewInject 注解的值就是 View 的 ID，我们知道每个含有 ID 的 View 最终都会在 R 类中生成一个整型的数值，这里的 View ID 就是这个整型数值。需要注意的是，这些被添加注解的字段都必须是非私有的，否则不能通过 target.fieldName 的形式直接访问。这些初始化代码都被写到了 InjectAdapter 子类的 inject 函数中，inject 函数传递一个 target 参数，这个 target 就是元素所在的类，如 ExampleActivity，而生成的辅助类的名称格式为宿主类+“$InjectAdapter”，如 ExampleActivity $InjectAdapter，它与 ExampleActivity 在同一个包中，因此，可以访问到 ExampleActivity 的 protected、package 权限的字段。InjectAdapter 的声明如下：

```
publicinterface InjectAdapter<T> {
voidinjects(T target);
}
```

这相当于我们为每个元素都生成一行初始化代码来替换手动在 ExampleActiivty 中进行 findViewById，当我们在 ExampleAcivity 的 onCreate 函数中调用 SimpleDagger 的 inject 函数时，会将 ExampleActivity 传递到 InjectAdapter 中，因此，最后为 ExampleActivity 生成的辅助类就成为了如下这样：

```
public class ExampleActivity$InjectAdapter implements InjectAdapter<ExampleActivity> {
  public void injects(ExampleActivity target)  {
      target.title =ViewFinder.findViewById(target, 2131099648  ) ;
      target.icon = ViewFinder.findViewById(target, 2131099649  ) ;
      target.footer=ViewFinder.findViewById(target, 2131099332  ) ;
  }
}
```

当调用 SimpleDagger 的 inject 时就会先通过传递进来的类名构建一个 InjectAdapter 子类的类名，例如，传递进来的是 ExampleActivity，那么此时辅助类的类名为 ExampleActivity$InjectAdapter，它是 InjectAdapter 的子类。获取到完整类名之后再反射构建一个对象，然后转换为 InjectAdapter，最好调用 inject 函数。而这个生成的 ExampleActivity$InjectAdapter 的 inject 函数中又对每个 View 进行了 findViewBydId，也就是对它们进行了初始化。至此，这些 View 字段就被自动初始化了！

我们最后再来理顺一下这个过程，大致分为如下几步：

（1）通过 ViewInject 注解标识一些 View 成员变量；

（2）通过 ViewInjecyProcessor 捕获添加了 ViewInject 注解的元素，并且按照宿主类进行分类；

（3）为每个含有 ViewInject 注解的宿主类生成一个 InjectAdapter 辅助类，并且在它的 inject 函数中生成初始化 View 的代码；

（4）在 SimpleDagger 的 inject 函数中构建生成辅助类，此时内部会调用这个 InjectAdapter 辅助类对象的 inject 函数，这个函数中又会初始化宿主类中的 View 成员变量，至此，View 就已经被初

始化了。

SimpleDagger 的完整代码在 https://github.com/bboyfeiyu/InjectDagger，有兴趣的读者可以下载学习。另外，SimpleDagger 是否适合使用 Visitor 模式？如果适合？应该如何重构使之运用 Visitor 模式？这留给大家思考。需要注意的是，在 Eclipse 中使用 APT 需要添加 JRE 库的引用，在 Android Studio 中则需要引用 APT 的插件。

16.8 总结

正如本章开头引用 GOF 的话所说：大多数情况下，你并需要使用访问者模式，但是，当你一旦需要使用它时，那你就是真地需要它了。在现实情况下，我们要根据具体的情况来评估是否适合使用访问者模式，例如，我们的对象结构是否足够稳定，使用访问者模式是否能够优化我们的代码，而不是使我们的代码变得更复杂。在使用一个模式之前，我们应该明确它的使用场景、它能解决什么问题等，以此来避免滥用设计模式的现象。所以，在学习设计模式时，一定要理解模式的适用性及优缺点。

访问者模式的优点。

（1）各角色职责分离，符合单一职责原则。

（2）具有优秀的扩展性。

（3）使得数据结构和作用于结构上的操作解耦，使得操作集合可以独立变化。

（4）灵活性。

访问者模式的缺点。

（1）具体元素对访问者公布细节，违反了迪米特原则。

（2）具体元素变更时导致修改成本大。

（3）违反了依赖倒置原则，为了达到“区别对待”而依赖了具体类，没有依赖抽象。

第 17 章 “和事佬”——中介者模式

17.1 中介者模式介绍

中介者模式（Mediator Pattern）也称为调解者模式或调停者模式，Mediator 本身就有调停者和调解者的意思。在日常生活中调停者或调解者这个角色我们见得比较多的是“和事佬”，也就是说调解两个有争端的人的角色，举个不恰当的例子，比如爸妈吵架，孩子或者双方父母则会出面劝架或阻止争吵，这里孩子或双方父母则是充当的是调解者的模式。而对于中介呢？大家平时听得最多的莫过于房产中介了，在房地产飞速发展的今天，各种房产中介公司如雨后春笋般冒出来，房产中介承担的责任很简单，就是在买房者与卖房者之间建立一座桥梁沟通两者，比如小民现在手里有套房子需要出售，于是他找房产中介并告诉房产中介房子的期望售价和房子的相关信息，然后房产中介则将这些信息收集汇总有时还会美化一番，然后把广告挂在橱窗里等待买房者，如果有人有意向购买这套房子，比如这里以小辉为例，房产中介就会告诉小辉关于这套房产的相关信息，如房龄、售价等，有时还会带小辉去房子里实际参观一番，最终小辉将自己的意愿告诉房产中介再由房产中介去与小民进行沟通。也就是说在整个房产协商的过程中，小民和小辉买卖双方是很少直接接触的，大多数情况下都是通过房产中介来传达信息，可能只有在最终进行房产交接时小民和小辉才会正式见面。这就是我们现实生活中常见的调解者与中介者，但是，这里我们要说的中介者模式比起上述的两个例子来要复杂得多，上面我们所述的例子中，中介者或调解者所要协调的也就两类对象，而我们的中介者模式中需要协调的是多类对象。这里还是举一个小例子来说明，本书的阅读者大多都是 IT 从业者，说得更准确些都是做 App 的，一般来说，一个做 App 的公司可以分为几大模块：运维、产品、开发、设计还有测试，每当一个新 App 开始制作或者新版本需要制作之前，公司会开一个研讨会，运维负责市场了解用户使用倾向，研发的会说用户反映界面不好看要求修改界面……这样大家你一句我一句的讨论一半天也没有啥实质性的结果，主要原因在于大家都站在自己的立场考虑问题，这样的团队研讨会即使再开下去也没什么意义，这时候如果有一个人不属于运维、产品、开发、设计任一方的人站出来说，大家把各自的意见都跟我说，由我来权衡并做出最终的决定，不管决定如何你们都照做。因为这个人不代表任何一方，于是大家都全票表示肯定，这么一来一个方案很快就会定下来，而这个作出最终决定的人我们就称之为中介者，这里的中介者与我们的中介者模式一样面对的是多类对象，这里其实大家可以看到中介者模式的一些特性，每一个中介者都会知道所有的同事，比如上面我们所说的运维、产品、开发、设计还有测试，但是这些同事呢可以互不相识，而中介者不属于同事的任何一方，也不偏袒任何一方，这么一说相信大家对中介者模式会有

一个非常清晰的了解。

17.2 中介者模式的定义

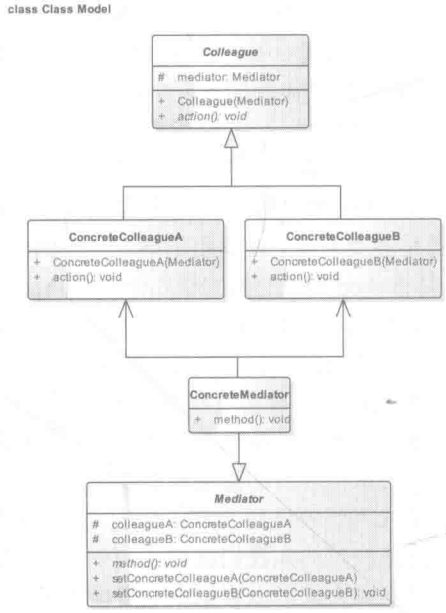
中介者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使它们可以松散偶合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立的变化。中介者模式将多对多的相互作用转化为一对多的相互作用。中介者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

17.3 中介者模式的使用场景

当对象之间的交互操作很多且每个对象的行为操作都依赖彼此时，为防止在修改一个对象的行为时，同时涉及修改很多其他对象的行为，可采用中介者模式，来解决紧耦合问题。该模式将对象之间的多对多关系变成一对多关系，中介者对象将系统从网状结构变成以调停者为中心的星形结构，达到降低系统的复杂性，提高可扩展性的作用。

17.4 中介者模式的 UML 类图

UML 类图如图 17-1 所示。



▲图 17-1

根据类图我们可以得出如下一个中介者模式的通用模式代码。

程序清单 17-1 抽象中介者

```
public abstract class Mediator {
    protected ConcreteColleagueA colleagueA;// 具体同事类 A
    protected ConcreteColleagueB colleagueB;// 具体同事类 B

    /**
     * 抽象中介方法、子类实现
     */
    public abstract void method();

    public void setColleagueA(ConcreteColleagueA colleagueA) {
        this.colleagueA = colleagueA;
    }

    public void setColleagueB(ConcreteColleagueB colleagueB) {
        this.colleagueB = colleagueB;
    }
}
```

程序清单 17-2 具体中介者

```
public class ConcreteMediator extends Mediator {
    @Override
    public void method() {
        colleagueA.action();

        colleagueB.action();
    }
}
```

程序清单 17-3 抽象同事

```
public abstract class Colleague {
    protected Mediator mediator;// 中介者对象

    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }

    /**
     * 同事角色的具体行为，由子类去实现
     */
    public abstract void action();
}
```

程序清单 17-4 具体同事 A

```
public class ConcreteColleagueA extends Colleague {
    public ConcreteColleagueA(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void action() {
        System.out.println("ColleagueA 将信息递交给中介者处理");
    }
}
```

程序清单 17-5 具体同事 B

```
public class ConcreteColleagueB extends Colleague {
    public ConcreteColleagueB(Mediator mediator) {
```

```
        super(mediator);
    }

    @Override
    public void action() {
        System.out.println("ColleagueB 将信息递交给中介者处理");
    }
}
```

角色介绍。

Mediator：抽象中介者角色，定义了同事对象到中介者对象的接口，一般以抽象类的方式实现。

ConcreteMediator：具体中介者窘色，继承于抽象中介者，实现了父类定义的方法，它从具体的同事对象接收消息，向具体同事对象发出命令。

Colleague：抽象同事类角色，定义了中介者对象的接口，它只知道中介者而不知道其他的同事对象。

ConcreteColleagueA/B：具体同事类角色，继承于抽象同事类，每个具体同事类都知道本身在小范围内的行为，而不知道它在大范围内的目的。

17.5 中介者模式的简单实现

中介者模式的例子很多，其实我们每天都在使用的电脑就是一个中介者模式应用的例子，以电脑主机为例，我们都知道电脑主机部分构成主要分为几块：CPU、内存、显卡、IO 设备，一般来说一台电脑有了前两块既可以运行启动了，当然，如果你要连接显示器显示画面，那么还需要加上显卡，如果你需要存储数据什么的，那还需要加上 IO 设备，但是，在本节中这些部分都不是重要的，因为它们分隔开来也就是一个普通的零部件而已，我们需要一样东西将这些零部件都整合起来变成一个完整的整体，这个东西就是主板，在这里主板就起到了中介者的作用，连接 CPU、内存、显卡和 IO 设备等，任何两个模块之间的通信都会经过主板去协调，这里以读取光盘为例，来看看主板是如何充当这个中介者角色的。首先还是定义一个抽象的中介者。

程序清单 17-6　抽象中介者

```
public abstract class Mediator {
    /**
     * 同事对象改变时通知中介者的方法
     * 在同事对象改变时由中介者去通知其他的同事对象
     *
     * @param c 同事对象
     */
    public abstract void changed(Colleague c);
}
```

抽象中介者中只是定义了一个抽象接口方法，具体的同事类通过该方法来通知中介者自身的状态改变。而具体的中介者这里就是指主板，由它负责联系各个具体同事类，也就是 CPU、内存、显卡、IO 设备等。

程序清单 17-7　主板中介者

```
public class MainBoard extends Mediator {
    private CDDevice cdDevice;// 光驱设备
```

```
private CPU cpu;// CPU
private SoundCard soundCard;// 声卡设备
private GraphicsCard graphicsCard;// 显卡设备

@Override
public void changed(Colleague c) {
    // 如果是光驱读取了数据
    if (c == cdDevice) {
        handleCD((CDDevice) c);
    }
    // 如果 CPU 处理完数据
    else if (c == cpu) {
        handleCPU((CPU) c);
    }
}

/**
 * 处理光驱读取数据后与其他设备的交互
 *
 * @param cdDevice 光驱设备
 */
private void handleCD(CDDevice cdDevice) {
    cpu.decodeData(cdDevice.read());
}

/**
 * 处理 CPU 读取数据后与其他设备的交互
 *
 * @param cpu CPU
 */
private void handleCPU(CPU cpu) {
    soundCard.soundPlay(cpu.getDataSound());
    graphicsCard.videoPlay(cpu.getDataVideo());
}

/**
 * 设置 CD 设备
 *
 * @param cdDevice CD 设备
 */
public void setCDDevice(CDDevice cdDevice) {
    this.cdDevice = cdDevice;
}

/**
 * 设置 CPU
 *
 * @param cpu CPU
 */
public void setCPU(CPU cpu) {
    this.cpu = cpu;
}

/**
 * 设置声卡设备
 *
 * @param soundCard 声卡设备
 */
public void setSoundCard(SoundCard soundCard) {
    this.soundCard = soundCard;
}
```

```
    /**
     * 设置显卡设备
     *
     * @param graphicsCard 显卡设备
     */
    public void setGraphicsCard(GraphicsCard graphicsCard) {
        this.graphicsCard = graphicsCard;
    }
}
```

抽象同事类里只有一个抽象中介者的引用，我们在构造方法中为其赋值。

程序清单 17-8　抽象同事

```
public abstract class Colleague {
    protected Mediator mediator;// 每一个同事都该知道其中介者

    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }
}
```

接下来就是各个具体的零部件了，首先是 CPU，其负责从主板传递来音、视频数据的解码。

程序清单 17-9　CPU 同事

```
public class CPU extends Colleague {
    private String dataVideo, dataSound;// 视频和音频数据

    public CPU(Mediator mediator) {
        super(mediator);
    }

    /**
     * 获取视频数据
     *
     * @return 视频数据
     */
    public String getDataVideo() {
        return dataVideo;
    }

    /**
     * 获取音频数据
     *
     * @return 音频数据
     */
    public String getDataSound() {
        return dataSound;
    }

    /**
     * 解码数据
     *
     * @param data 音、视频数据
     */
    public void decodeData(String data) {
        // 分割音、视频数据
        String[] tmp = data.split(",");

        // 解析音、视频数据
        dataVideo = tmp[0];
```

```
        dataSound = tmp[1];

        // 告诉中介者自身状态改变
        mediator.changed(this);
    }
}
```

而 CD 设备则负责读取光盘的数据并将数据提供给主板。

程序清单 17-10 光驱同事

```
public class CDDevice extends Colleague {
    private String data;// 视频数据

    public CDDevice(Mediator mediator) {
        super(mediator);
    }

    /**
     * 读取视频数据
     *
     * @return 视频数据
     */
    public String read() {
        return data;
    }

    /**
     * 加载视频数据
     */
    public void load() {
        // 实际情况中视频数据与音频数据都在一个数据流中
        data = "视频数据,音频数据";

        // 通知中介者 也就是主板数据改变
        mediator.changed(this);
    }
}
```

显卡和声卡分别用来播放视频和音频，其逻辑相对来说简单。

程序清单 17-11 显卡同事

```
public class GraphicsCard extends Colleague {
    public GraphicsCard(Mediator mediator) {
        super(mediator);
    }

    /**
     * 播放视频
     *
     * @param data 视频数据
     */
    public void videoPlay(String data) {
        System.out.println("视频: " + data);
    }
}
```

程序清单 17-12 声卡同事

```
public class SoundCard extends Colleague {
    public SoundCard(Mediator mediator) {
        super(mediator);
```

```
    }

    /**
     * 播放音频
     *
     * @param data 音频数据
     */
    public void soundPlay(String data) {
        System.out.println("音频：" + data);
    }
}
```

最后，我们通过一个客户类来模拟电脑播放电影的效果。

程序清单 17-13　客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造主板对象
        MainBoard mediator = new MainBoard();

        // 分别构造各个零部件
        CDDevice cd = new CDDevice(mediator);
        CPU cpu = new CPU(mediator);
        GraphicsCard vc = new GraphicsCard(mediator);
        SoundCard sc = new SoundCard(mediator);

        // 将各个零部件安装到主板
        mediator.setCDDevice(cd);
        mediator.setCPU(cpu);
        mediator.setGraphicsCard(vc);
        mediator.setSoundCard(sc);

        // 完成后就可以开始放片了
        cd.load();
    }
}
```

输出结果如下：

```
音频：音频数据
视频：视频数据
```

从上述程序演示读者可以明白，中介者模式就是用来协调多个对象之间的交互的，就像上面示例中的主板，如果没有主板这个中介者，那么电脑里的每一个零部件都要与其他零部件建立关联，比如 CPU 要与内存交互、CPU 要与显卡交互、CPU 还要与 IO 设备交互等，这么一来就会构成一个错综复杂的网状图，而中介者模式的出现则是将这一个错综复杂的网状图变成一个结构清晰的星形图，其中心就是中介者。

17.6 Android 源码中的中介者模式实现

中介者模式在 Android 源码中的一个比较好的例子是 Keyguard 锁屏的功能实现，这里给大家介绍一个找这类设计模式实现的小技巧，很多读者觉得在浩瀚的 Android 代码中寻找一个设计模式实在太难，其实也并非如此，为什么这么说呢？因为 Android 源码的命名相当规范，如你要找一个

设计模式，这里以中介者模式为例，只需要在源码中搜索“Mediator”关键词，你就会找到KeyguardViewMediator 这个类，顺藤摸瓜只需阅读 KeyguardViewMediator 的源码就可以找到其他的同事类，由此构建一个完整的设计模式类图，这相对于在整个 Android 源码“大海中捞针”要容易得多，这里再次体现命名的重要性。我们来看看这个 KeyguardViewMediator 中都有些什么。

```
public class KeyguardViewMediator {

    // ......省略一些代码......

    private AlarmManager mAlarmManager;

    private AudioManager mAudioManager;

    private StatusBarManager mStatusBarManager;

    private PowerManager mPM;

    private UserManager mUserManager;

    private SearchManager mSearchManager;

    private KeyguardViewManager mKeyguardViewManager;

    // ......省略一些代码......
}
```

首先，我们在 KeyguardViewMediator 中看到很多 XXXManager 管理器的成员变量，这些各种各样的管理器其实就是各个具体的同事类，Android 使用 KeyguardViewMediator 来充当中介者协调这些管理器的状态改变，同样地 KeyguardViewMediator 中也定义了很多方法来处理这些管理器的状态，以解锁或锁屏时声音的播放为例，在 KeyguardViewMediator 中对应 playSounds 方法来协调音频的这一状态。

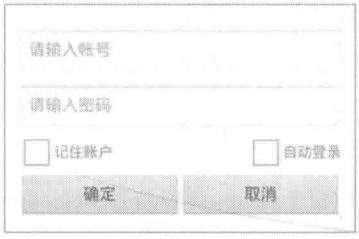
```
private void playSounds(boolean locked) {
    if (mSuppressNextLockSound) {
        mSuppressNextLockSound = false;
        return;
    }
    final ContentResolver cr = mContext.getContentResolver();
    if (Settings.System.getInt(cr, Settings.System.LOCKSCREEN_SOUNDS_ENABLED, 1) == 1) {
        final int whichSound = locked ? mLockSoundId : mUnlockSoundId;
        mLockSounds.stop(mLockSoundStreamId);
        if (mAudioManager == null) {
            mAudioManager = (AudioManager) mContext.getSystemService(Context.AUDIO_SERVICE);
            if (mAudioManager == null) return;
                mMasterStreamType = mAudioManager.getMasterStreamType();
        }
        if (mAudioManager.isStreamMute(mMasterStreamType)) return;
        mLockSoundStreamId = mLockSounds.play(whichSound, mLockSoundVolume,
        mLockSoundVolume, 1/*priortiy*/, 0/*loop*/, 1.0f/*rate*/);
    }
}
```

而其他管理器的协调大家也可以在 KeyguardViewMediator 中找到类似的协调方法，这里不再多说。另一个 Anadroid 中关于中介者模式的例子是 Binder 机制，Binder 机制是 Android 中非常重

要的一个机制，其用来绑定不同的系统级服务并进行跨进程通信，关于 Binder 机制的一些详细情况我们会在下一小节中阐述，这里我们简单地来了解一下其是如何应用到中介者模式的。在 Binder 机制中，有 3 个非常重要的组件 ServiceManager、Binder Driver 和 Bp Binder，其中 Bp Binder 是 Binder 的一个代理角色，其提供 IBinder 接口给各个客户端服务使用，这三者就扮演了一个中介者的角色。当手机启动后，ServiceManager 会先向 Binder Driver 进行注册，注意，这里 ServiceManager 虽然是一个特殊的服务，但毕竟还是一个服务，其特殊性在于，它在 Binder Driver 中是最先被注册的，其注册 ID 为 0，当其他的服务想要注册到 Binder Driver 时，会先通过这个 0 号 ID 获取到 ServiceManager 所对应的 IBinder 接口，该接口实质上的实现逻辑是由 BpBinder 实现的，获取到对应的接口后就回调用其中的 transact 方法，此后就会在 Binder Driver 中新注册一个 ID 1 来对应这个服务，如果有客户端想要使用这个服务，那么，它先会像 Binder Driver 一样获取到 ID 为 0 的接口，也就是 ServiceManager 所对应的接口，并调用其 transact 方法要求连接到刚才的服务，这时候 Binder Driver 就会将 ID 为 1 的服务回传给客户端并将相关消息反馈给 ServiceManager 完成连接。这里 ServiceManager 和 Binder Driver 就相当于一个中介者，协调各个服务端与客户端，关于 Binder 机制更多的信息大家可以查看第 18 章代理模式，这里不再多说。

17.7 中介者模式实战

中介者模式对大家来说在应用方面感觉很陌生，特别是在 Android 开发中。如果大家是做 Java 开发的没怎么用过中介者模式其实还可以理解，如果说在 Android 开发中没有使用过中介者模式，我就表示诧异了，为什么这么说？我们回顾一下中介者模式是用来做什么的？协调多个交互的对象，Android 中这么多形形色色的控件算不算是交互对象呢？当然是，既然如此你还说没有接触过中介者模式么？其实你已经在不经意间就已经使用到了中介者模式，打个比方说，很多应用都会有登录界面，如图 17-2 所示。



▲图 17-2

一个登录界面包含的元素不算多，一般情况下就像图 17-2 那样，包含两个文本框、两个按钮、两个复选框，有时候这两个复选框不一定会有，6 个元素看似简单，但是逻辑实现起来也许相当复杂，考虑这样一种需求，当用户名没有输入的时候除了取消按钮外均不可使用，当密码没有输入时除了账户文本输入框、记住账户复选框和取消按钮外其余均不可用，当且仅当账户密码均正确时所有控件才可用，如果勾选了自动登录，那么记住账户也会被同时勾选，并且下次登录时账户和文本框均被数据填充且不可编辑，如果自动登录被取消，那么记住账户依然保持勾选状态直至用户手动取消并且置空密码文本框但不置空账户文本框等，这么一个多 UI 控件交互的情景其实就是中介者模式的用武之地，平时很多情况下我们并不会按照中介者模式的框架来实现，大多数情况下我们直接让 Activity 来充当一个中介者并在其中处理相关逻辑。

```
public class LoginActivity extends Activity {
    private EditText etAccount, etPassWord;// 账户文本框、密码文本框
    private Button btnEnsure, btnCancel;// 确定按钮、取消按钮
    private CheckBox cbAuto, cbRemember;// 自动登录复选框、记住账户复选框
```

```
private String strAccount, strPassword;// 账户文本、密码文本

private boolean isAuto, isRemember;// 是否自动登录、是否记住账户

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

    // 获取账户文本框并设置文本改变监听
    etAccount = (EditText) findViewById(R.id.login_account_et);
    etAccount.addTextChangedListener(new TextWatcher() {
        @Override
        public void beforeTextChanged(CharSequence s, int start, int count, int after) {

        }

        @Override
        public void onTextChanged(CharSequence s, int start, int before, int count) {
            // 回传数据
            strAccount = s.toString();

            // 通知 Activity 状态改变
            change();
        }

        @Override
        public void afterTextChanged(Editable s) {

        }
    });
    // 获取密码文本框并设置文本改变监听和初始状态
    etPassWord = (EditText) findViewById(R.id.login_password_et);
    etPassWord.setEnabled(false);
    etPassWord.addTextChangedListener(new TextWatcher() {
        @Override
        public void beforeTextChanged(CharSequence s, int start, int count, int after) {

        }

        @Override
        public void onTextChanged(CharSequence s, int start, int before, int count) {
            // 回传数据
            strPassword = s.toString();

            // 通知 Activity 状态改变
            change();
        }

        @Override
        public void afterTextChanged(Editable s) {

        }
    });

    // 获取确定按钮并设置点击事件监听和初始化状态
    btnEnsure = (Button) findViewById(R.id.login_ensure_btn);
    btnEnsure.setEnabled(false);
    btnEnsure.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
```

```
                Toast.makeText(LoginActivity.this, "登录成功", Toast.LENGTH_SHORT).show();
            }
        });
        // 获取取消按钮并设置点击事件监听
        btnCancel = (Button) findViewById(R.id.login_cancel_btn);
        btnCancel.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(LoginActivity.this, "取消登录", Toast.LENGTH_SHORT).show();
            }
        });

        // 获取自动登录复选框并设置勾选事件监听和初始化状态
        cbAuto = (CheckBox) findViewById(R.id.login_auto_cb);
        cbAuto.setEnabled(false);
        cbAuto.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
                // 回传数据
                isAuto = isChecked;

                // 通知 Activity 状态改变
                change();
            }
        });
        // 获取记住账户复选框并设置勾选事件监听和初始化状态
        cbRemember = (CheckBox) findViewById(R.id.login_remember_cb);
        cbRemember.setEnabled(false);
        cbRemember.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener(){
            @Override
            public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
                // 回传数据
                isRemember = isChecked;

                // 通知 Activity 状态改变
                change();
            }
        });
    }
}
```

如上述代码所示，我们在 Activity 的 onCreate 方法中获取各个控件并初始化设置监听，每当空间状态改变时我们都将其改变的信息回传给 Activity，由 Activity 充当中介者来协调这些控件的状态。

```
/**
 * 各控件状态改变后协调各控件状态显示
 */
private void change() {
    if (TextUtils.isEmpty(strAccount)) {
        etPassWord.setEnabled(false);

        cbAuto.setEnabled(false);
        cbRemember.setEnabled(false);

        btnEnsure.setEnabled(false);
        btnCancel.setEnabled(true);
    } else if (!TextUtils.isEmpty(strAccount) &&
            TextUtils.isEmpty(strPassword)) {
                etPassWord.setEnabled(true);

        cbAuto.setEnabled(false);
```

```
        cbRemember.setEnabled(true);

        btnEnsure.setEnabled(false);
        btnCancel.setEnabled(true);
    } else if (!TextUtils.isEmpty(strAccount) &&
            !TextUtils.isEmpty(strPassword)) {
                etPassWord.setEnabled(true);

                cbAuto.setEnabled(true);
                cbRemember.setEnabled(true);

                btnEnsure.setEnabled(true);
                btnCancel.setEnabled(true);
    }
}
```

具体运行效果就不再演示了，逻辑相对简单大家可以自行尝试。

17.8 总结

在面向对象的编程语言里，一个类必然会与其他类产生依赖关系，如果这种依赖关系如网状般错综复杂，那么必然会影响我们的代码逻辑以及执行效率，适当地使用中介者模式可以对这种依赖关系进行解耦使逻辑结构清晰，但是，如果几个类间的依赖关系并不复杂，使用中介者模式反而会使得原本不复杂的逻辑结构变得复杂，所以，我们在决定使用中介者模式之前要多方考虑、权衡利弊。

第 18 章　编程好帮手——代理模式

18.1 代理模式介绍

代理模式（Proxy Pattern）也称为委托模式，是本书结构型设计模式所讲的第一个模式，其重要性不言而喻，相信在学过本章的内容后会发现不少设计模式中都有代理模式的影子。那么何谓代理呢？其实代理在我们日常生活中也并不少见，对于程序员来说最常接触的莫过于代理上网了，连上代理服务器地址，就可以轻松畅游全世界的网络；还有每天吃饭时赶进度是常事，叫公司的同事帮忙买饭也是一种代理；如果你碰到辞职老板不给你发工资，那么你还得请个律师帮你打官司，这也是一种代理。总而言之，也许你并不留意，但是代理的确是无处不在，现实生活中如此，我们的 Code 世界里也是如此！既然如此，我们来深究一下代理模式到底有多普遍。首先还是来看一个它的定义。

18.2 代理模式的定义

为其他对象提供一种代理以控制对这个对象的访问。

18.3 代理模式的使用场景

当无法或不想直接访问某个对象或访问某个对象存在困难时可以通过一个代理对象来间接访问，为了保证客户端使用的透明性，委托对象与代理对象需要实现相同的接口。

18.4 代理模式的 UML 类图

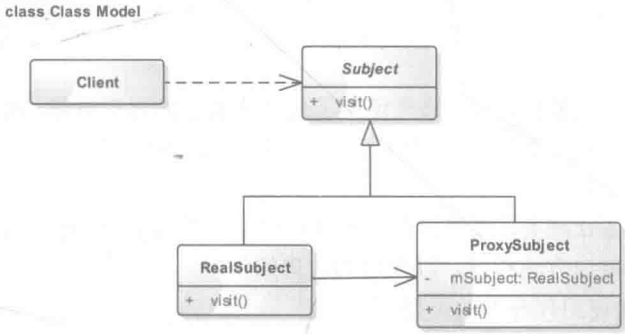
UML 类图如图 18-1 所示。

根据类图可以得出如下一个桥接模式的通用模式代码。

程序清单 18-1　抽象主题类

```
public abstract class Subject {
    /**
     * 一个普通的业务方法
```

```
    */
    public abstract void visit();
}
```



▲图 18-1

程序清单 18-2　实现抽象主题的真实主题类

```
public class RealSubject extends Subject {
    @Override
    public void visit() {
        // RealSubject 中 visit 的具体逻辑
        System.out.println("Real subject!");
    }
}
```

程序清单 18-3　代理类

```
public class ProxySubject extends Subject {
    private RealSubject mSubject;// 持有真实主题的引用

    public ProxySubject(RealSubject mSubject) {
        this.mSubject = mSubject;
    }

    @Override
    public void visit() {
        // 通过真实主题引用的对象调用真实主题中的逻辑方法
        mSubject.visit();
    }
}
```

程序清单 18-4　客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造一个真实主题对象
        RealSubject real = new RealSubject();

        // 通过真实主题对象构造一个代理对象
        ProxySubject proxy = new ProxySubject(real);

        // 调用代理的相关方法
        proxy.visit();
    }
}
```

角色介绍。

Subject：抽象主题类。

该类的主要职责是声明真实主题与代理的共同接口方法，该类既可以是一个抽象类也可以是一个接口。

RealSubject：真实主题类。

该类也称为被委托类或被代理类，该类定义了代理所表示的真实对象，由其执行具体的业务逻辑方法，而客户类则通过代理类间接地调用真实主题类中定义的方法。

ProxySubject：代理类。

该类也称为委托类或代理类，该类持有一个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，以此起到代理的作用。

Client：客户类，即使用代理类的类型。

18.5 代理模式的简单实现

就像本书的每一章一样，对模式的定义阐述是很枯燥的，但是又是必须的，我们只有在对每一种模式的定义了解后才能更好地通过例子来理解它。代理模式的例子如本文开头所述，你可以在日常生活中找到很多很多。小民以前在公司上班时，就遇到过被老板拖欠工资甚至克扣工资的情况，这种情况下小民还是通过法律途径来解决问题，一旦小民选择了走法律途径解决该纠纷，那么不可避免地就需要请一个律师来作为自己的诉讼代理人，我们将诉讼的流程抽象在一个接口类中。

程序清单 18-5　诉讼接口类

```
public interface ILawsuit {
    // 提交申请
    void submit();

    // 进行举证
    void burden();

    // 开始辩护
    void defend();

    // 诉讼完成
    void finish();
}
```

4 个方法非常简单，都是诉讼的一般流程。

程序清单 18-6　具体诉讼人

```
public class XiaoMin implements ILawsuit {
    @Override
    public void submit() {
        // 老板欠小民工资 小民只好申请仲裁
        System.out.println("老板拖欠工资！特此申请仲裁！");
    }

    @Override
    public void burden() {
        // 小民证据充足，不怕告不赢
```

```
        System.out.println("这是合同书和过去一年的银行工资流水！");
    }

    @Override
    public void defend() {
        // 铁证如山，辩护也没什么好说的
        System.out.println("证据确凿！不需要再说什么了！");
    }

    @Override
    public void finish() {
        // 结果也是肯定的，必赢
        System.out.println("诉讼成功！判决老板即日起七天内结算工资");
    }
}
```

如上所述，该类实现 ILawsuit 并对其中 4 个方法作出具体的实现逻辑，逻辑很简单，都只是输出一段话而已，当然，小民自己是不会去打官司的，于是小民请个律师代替自己进行诉讼。

程序清单 18-7　代理律师

```
public class Lawyer implements ILawsuit {
    private ILawsuit mLawsuit;// 持有一个具体被代理者的引用

    public Lawyer(ILawsuit lawsuit) {
        mLawsuit = lawsuit;
    }

    @Override
    public void submit() {
        mLawsuit.submit();
    }

    @Override
    public void burden() {
        mLawsuit.burden();
    }

    @Override
    public void defend() {
        mLawsuit.defend();
    }

    @Override
    public void finish() {
        mLawsuit.finish();
    }
}
```

律师类表示代理者律师，在该类里面会持有一个被代理者（这里也就是上面的 Aige 类）的引用，律师所执行的方法实质就是简单地调用被代理者中的方法，下面来看看客户类中具体的调用执行关系。

程序清单 18-8　客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造一个小民……
        ILawsuit xiaomin = new XiaoMin();
```

```
        // 构造一个代理律师并将小民作为构造参数传递进去
        ILawsuit lawyer = new Lawyer(xiaomin);

        // 律师提交诉讼申请
        lawyer.submit();

        // 律师进行举证
        lawyer.burden();

        // 律师代替小民进行辩护
        lawyer.defend();

        // 完成诉讼
        lawyer.finish();
    }
}
```

运行结果很简单这里不再给出，大家可以看到，其实代理模式也很简单，其主要还是一种委托机制，真实对象将方法的执行委托给代理对象，而且委托得干净利落毫不做作，这也是为什么代理模式也称为委托模式的原因，相信大家不难理解。除此之外，大家其实可以继续发散思维，其实我们的代理类完全可以代理多个被代理类，就像上面的例子一样，一个律师可以代理多个人打官司，这是没有任何问题的，而具体到底是代理的哪个人，这就要看代理类中所持有的实际对象类型，上述的例子中实际对象类型是 XiaoMin，也就是代理的 XiaoMin。

```
public class Lawyer implements ILawsuit {
    private ILawsuit mLawsuit;// 持有一个具体被代理者的引用

    public Lawyer(ILawsuit lawsuit) {
        mLawsuit = lawsuit;
    }

    // ......省去 4 个方法......
}
public class Client {
    public static void main(String[] args) {
        // 构造一个小民……
        ILawsuit xiaomin = new XiaoMin();

        // 构造一个代理律师并将小民作为构造参数传递进去
        ILawsuit lawyer = new Lawyer(xiaomin);

        // ......省去一些代码......
    }
}
```

代理其他人可以吗？当然可以，比如小辉，我们只需再定义个 XiaoHui 类实现 ILawsuit 即可，再在客户类中修改高层模块调用逻辑，完全没问题。

如本文开头所述，代理模式是一个非常重要的模式，因此，本章内容会相对多一点，还望读者耐心阅读。代理模式可以大致分为两大部分，一是静态代理，二是动态代理。静态代理如上述示例那样，代理者的代码由程序员自己或通过一些自动化工具生成固定的代码再对其进行编译，也就是说在我们的代码运行前代理类的 class 编译文件就已存在；而动态代理则与静态代理相反，通过反射机制动态地生成代理者的对象，也就是说我们在 code 阶段压根就不需要知道代理谁，代理谁我们将会在执行阶段决定。而 Java 也给我们提供了一个便捷的动态代理接口 InvocationHandler，实现

该接口需要重写其调用方法 invoke。

```
public class DynamicProxy implements InvocationHandler {
    @Override
    public Object invoke(Object arg0, Method arg1, Object[] arg2)
    throws Throwable {
            return null;
    }
}
```

在这里，我们主要通过 invoke 方法来调用具体的被代理方法，也就是真实的方法。动态代理可以使我们的代码逻辑更简洁，不过在这之前我们得首先完善动态代理类。

程序清单 18-9　动态代理类

```
public class DynamicProxy implements InvocationHandler {
    private Object obj;// 被代理的类引用

    public DynamicProxy(Object obj) {
        this.obj = obj;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        // 调用被代理类对象的方法
            Object result = method.invoke(obj, args);
            return result;
    }
}
```

如上代码所述，我们声明一个 Object 的引用，该引用将指向被代理类，而我们调用被代理类的具体方法则在 invoke 方法中执行，是不是很简洁？也就是说我们原来由代理类所做的工作现在由 InvocationHandler 来处理，不再需要关心到底代理谁。下面我们修改客户类的逻辑。

程序清单 18-10　修改后的客户类

```
    public static void main(String[] args) {
        // 构造一个小民……
        ILawsuit xiaomin = new XiaoMin();

        // 构造一个动态代理
        DynamicProxy proxy = new DynamicProxy(xiaomin);

        // 获取被代理类小民的 ClassLoader
        ClassLoader loader = xiaomin.getClass().getClassLoader();

        // 动态构造一个代理者律师
        ILawsuit lawyer = (ILawsuit) Proxy.newProxyInstance(loader,
              new Class[] { ILawsuit.class }, proxy);

        // 律师提交诉讼申请
        lawyer.submit();

        // 律师进行举证
        lawyer.burden();

        // 律师代替小民进行辩护
        lawyer.defend();
```

```
        // 完成诉讼
        lawyer.finish();
    }
}
```

运行结果和以前一致，不再给出。由此可见动态代理通过一个代理类来代理 N 多个被代理类，其实质是对代理者与被代理者进行解耦，使两者直接没有直接的耦合关系。相对而言静态代理则只能为给定接口下的实现类做代理，如果接口不同那么就需要重新定义不同代理类，较为复杂，但是静态代理更符合面向对象原则。在开发时具体使用哪种方式来实现代理，就看自己的偏好了。

静态代理和动态代理是从 code 方面来区分代理模式的两种方式，我们也可以从其适用范围来区分不同类型的代理实现。

- 远程代理（Remote Proxy）：为某个对象在不同的内存地址空间提供局部代理。使系统可以将 Server 部分的实现隐藏，以便 Client 可以不必考虑 Server 的存在。
- 虚拟代理（Virtual Proxy）：使用一个代理对象表示一个十分耗资源的对象并在真正需要时才创建。
- 保护代理（Protection Proxy）：使用代理控制对原始对象的访问。该类型的代理常被用于原始对象有不同访问权限的情况。
- 智能引用（Smart Reference）：在访问原始对象时执行一些自己的附加操作并对指向原始对象的引用计数。

这里要注意的是，静态和动态代理都可以应用于上述 4 种情形，两者是各自独立的变化。

18.6 Android 源码中的代理模式实现

Android 源码里有不少关于代理模式的实现，比如源码里的 ActivityManagerProxy 代理类，其具体代理的是 ActivityManagerNative 的子类 ActivityManagerService，ActivityManagerService 在本书其他的一些章节都略有介绍，这里不再多说，我们主要来看看 ActivityManagerProxy 这个类，该类与 ActivityManagerNative 这个类处于同一文件中。

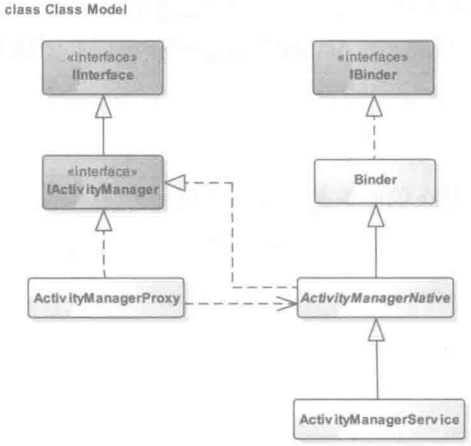
```
class ActivityManagerProxy implements IActivityManager{
    // ......省略类中的代码......
}
```

ActivityManagerProxy 实现了 IActivityManager 接口，该接口定义了一些 Activity 相关的接口方法，其中有一些我们在应用开发中也时常接触到。

```
public interface IActivityManager extends IInterface {
    public int startActivity(IApplicationThread caller, String callingPackage, Intent
intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode, int flags,
ProfilerInfo profilerInfo, Bundle options) throws RemoteException;

    // ......省略一些方法......

    public Intent registerReceiver(IApplicationThread caller, String callerPackage,
IIntentReceiver receiver, IntentFilter filter, String requiredPermission, int userId)
throws RemoteException;
```

```
    public void unregisterReceiver(IIntentReceiver receiver) throws RemoteException;

    // ......省略一些方法......

    public ComponentName startService(IApplicationThread caller, Intent service, String
resolvedType, int userId) throws RemoteException;

    public int stopService(IApplicationThread caller, Intent service, String resolvedType,
int userId) throws RemoteException;

    public int bindService(IApplicationThread caller, IBinder token, Intent service,
String resolvedType, IServiceConnection connection, int flags, int userId) throws
RemoteException;

    public boolean unbindService(IServiceConnection connection) throws RemoteException;

    // ......省略一些方法......

    public int checkPermission(String permission, int pid, int uid)

    // ......省略一些方法......

    public void overridePendingTransition(IBinder token, String packageName, int enterAnim,
int exitAnim) throws RemoteException;

    // ......省略一些代码......
}
```

上面的 IActivityManager 这个接口类就相当于代理模式中的抽象主题，那么真正的实现主题是谁呢？就是上面我们所提到过的继承于 ActivityManagerNative 的 ActivityManagerService 类，这几个类之间的大致关系如图 18-2 所示。



▲图 18-2

通过 UML 类图可以清晰地看到 ActivityManagerProxy 和 ActivityManagerNative 都实现了 IActivityManager，严格来说，ActivityManagerProxy 就是代理部分，而 ActivityManagerNative 就是真实部分，但 ActivityManagerNative 是个抽象类，其并不处理过多的具体逻辑，大部分具体逻辑的

实现都由其子类 ActivityManagerService 承担，这就是为什么我们说真实部分应该为 ActivityManager Service 而非 ActivityManagerNative。ActivityManagerService 是系统级的 Service 并且运行于独立的进程空间中，可以通过 ServiceManger 来获取它。而 ActivityManagerProxy 也运行于自己所处的进程空间中，两者并不相同，因此 ActivityManagerProxy 与 ActivityManagerService 的通信必定是通过跨进程来进行的，而由上述的类图我们也可以了解到，这里跨进程的实现是基于 Android 的 Binder 机制，同时通过上面我们对代理模式的 4 种适用情景可以判断，此处源码所实现的代理实质为远程代理。ActivityManagerProxy 在实际的逻辑处理中并未过多地被外部类使用，因为在 Android 中管理与维护 Activity 相关信息的类是另外一个叫做 ActivityManager 的类，ActivityManager 虽说管理着 Activity 的信息，但是实质上其大多数逻辑都由 ActivityManagerProxy 承担，这里以其中的 getAppTasks 方法为例，在 ActivityManager 中 getAppTasks 方法的逻辑如下。

```
public List<ActivityManager.AppTask> getAppTasks() {
    ArrayList<AppTask> tasks = new ArrayList<AppTask>();
    List<IAppTask> appTasks;
    try {
        appTasks = ActivityManagerNative.getDefault().getAppTasks(mContext.getPackageName());
    } catch (RemoteException e) {
        return null;
    }
    int numAppTasks = appTasks.size();
    for (int i = 0; i < numAppTasks; i++) {
        tasks.add(new AppTask(appTasks.get(i)));
    }
    return tasks;
}
```

逻辑很简单，或者说压根就没什么逻辑可言，主要只是简单地调用了 ActivityManagerNative 的 getDefault 方法获取一个 IActivityManager 类型的对象，然后通过该对象再调用其 getAppTasks 方法，注意，这里结合代理模式理解。而 ActivityManagerNative 的 getDefault 方法只是单纯地返回了。

```
static public IActivityManager getDefault() {
    return gDefault.get();
}
```

那么这个 gDefault 又是做什么的呢？具体如下代码所示。

```
private static final Singleton<IActivityManager>gDefault=new Singleton<IActivityManager>(){
    protected IActivityManager create() {
        IBinder b = ServiceManager.getService("activity");

        // ......省略 3 行代码......

        IActivityManager am = asInterface(b);

        // ......省略 3 行代码......

        return am;
    }
};
```

上述代码只是构造了一个 Singleton<IActivityManager>类型的 gDefault 对象，其中通过 ServiceManager.getService（“activity”）获取了一个系统级的 Service，而这个 Service 实质上就是

ActivityManagerService，这里也就完成创建了一个对 ActivityManagerService 的 Client 代理对象 ActivityManagerProxy 的实例。ActivityManagerProxy 中的 getAppTasks 方法逻辑就很明确，将数据打包跨进程传递给 Server 端 ActivityManagerService 处理并返回结果。

```
public List<IAppTask> getAppTasks(String callingPackage) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeString(callingPackage);
    mRemote.transact(GET_APP_TASKS_TRANSACTION, data, reply, 0);
    reply.readException();
    ArrayList<IAppTask> list = null;
    int N = reply.readInt();
    if (N >= 0) {
        list = new ArrayList<IAppTask>();
        while (N > 0) {
            IAppTask task = IAppTask.Stub.asInterface(reply.readStrongBinder());
            list.add(task);
            N--;
        }
    }
    data.recycle();
    reply.recycle();
    return list;
}
```

我们再来看看 ActivityManagerService 中对 getAppTasks 方法的逻辑处理，当然 getAppTasks 的具体逻辑不在本章的讨论范畴内，主要还是让大家对源码中的代理模式有个了解即可，这里不再赘述。

```
public List<IAppTask> getAppTasks(String callingPackage) {
    int callingUid = Binder.getCallingUid();
    long ident = Binder.clearCallingIdentity();

    synchronized(this) {
        ArrayList<IAppTask> list = new ArrayList<IAppTask>();
        try {
            if (localLOGV) Slog.v(TAG, "getAppTasks");

            final int N = mRecentTasks.size();
            for (int i = 0; i < N; i++) {
                TaskRecord tr = mRecentTasks.get(i);
                if (tr.effectiveUid != callingUid) {
                    continue;
                }
                Intent intent = tr.getBaseIntent();
                if (intent == null ||
                !callingPackage.equals(intent.getComponent().getPackageName())) {
                    continue;
                }
                ActivityManager.RecentTaskInfo taskInfo =
                        createRecentTaskInfoFromTaskRecord(tr);
                        AppTaskImpl taskImpl=new AppTaskImpl(taskInfo.persistentId,callingUid);
                        list.add(taskImpl);
            }
        } finally {
            Binder.restoreCallingIdentity(ident);
        }
        return list;
    }
```

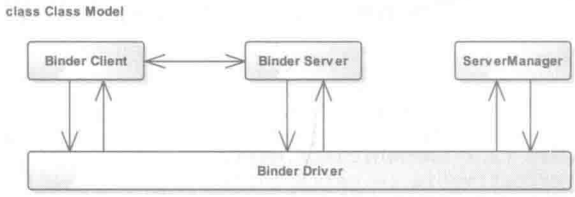
```
}
```

如果大家不了解 Android 中的跨进程通信，也许对上述代码理解起来会比较困难，如果看不懂这部分的内容，可以先跳过，接下来的部分我们将对 Android 的跨进程通信机制做一个大致的讲解。

18.7 Android 中的 Binder 跨进程通信机制与 AIDL

在 Code 的世界里，两个对象能直接相互访问的前提是这两个对象都存在于相同的内存地址空间中，如果两个对象分别存在于两个不同的进程中，比如上面我们说到的 ActivityManager 和 ActivityManagerService，那么这两个对象是不能直接相互调用的，这时我们就需要使用到一种跨进程通信技术，使存在于两个不同进程的对象能够互访，传统的跨进程通信方式有很多，比如 Socket、信号量、管道、内存共享、消息队列等，这些传统的跨进程通信方式在 Linux 中都存在，大家知道 Android 是基于 Linux 的，按理来说其使用现成的跨进程通信方式是理所应当，那么为什么还要大费周张地自己设计一个新的通信机制呢？这只能说明 Binder 具有其他通信方式无可比拟的优势，传统的跨进程通信机制，如 Socket，开销大且效率不高，而管道和队列拷贝次数多，更重要的是，对于移动设备来说，安全性相当重要，而传统的通信机制安全性低，大部分情况下接收方无法得到发送方进程的可信 PID/UID，难以对其身份甄别。而 Binder 在设计的时候就考虑到了以上各种缺点，在确保传输性能的同时又提高了安全性。

现在进入正题，不过在讲解 Binder 之前我们首先对 Binder 所涉及的 4 个主要模块作大致的了解，它们分别是 Binder Client、Binder Server、ServerManager 和 Binder Driver。这 4 者之间的关系类似于网络访问，Binder Client 相当于我们的客户端 PC，Binder Server 相当于服务器，ServerManager 相当于 DNS 服务器，而 Binder Driver 则相当于一个路由器，它们的关系如图 18-3 所示。



▲图 18-3

其中 Binder Driver 实现在内核空间中，而其余的 3 者 Binder Client、Binder Server、ServerManager 实现在用户空间中。

Binder Driver 位于内核空间中，其以字符设备中的 misc 类型注册，用户可以从/dev/binder 设备文件节点上，通过 open 和 ioctl 文件操作函数与 Binder Driver 进行通信，其主要负责 Binder 通信的建立，以及其在进程间的传递和 Binder 引用计数管理/数据包的传输等。而 Binder Client 与 Binder Server 之间的跨进程通信则统一通过 Binder Driver 处理转发，对于 Binder Client 来说，其只需要知道自己要使用的 Binder 的名字以及该 Binder 实体在 ServerManager 中的 0 号引用即可，访问的原理也比较简单，Binder Client 先是通过 0 号引用去访问 ServerManager 获取该 Binder 的引用，得到

引用后就可以像普通方法调用那样调用 Binder 实体的方法。最后我们的 ServerManager 则用来管理 Binder Server，Binder Client 可以通过它来查询 Binder Server 接口，刚才我们说到 Binder Client 可以通过 ServerManager 来获取 Binder 的引用，这个 Binder 的引用就是由 ServerManager 来转换的，这里与其说转换不如说映射更直接，Binder Server 在生成一个 Binder 实体的同时会为其绑定一个名字并将这个名字封装成一个数据包传递给 Binder Driver，Binder Driver 接收到这个数据包后，如果发现这个 Binder 是新传递来的，那么就会为其在内核空间中创建相应的 Binder 实体节点和一个对该实体节点的引用，这个实体节点在相应的源码中叫做 Binder_node 而其引用则叫做 Binder_ref，创建完毕后，Binder Driver 就会将该引用传递给 ServerManager，ServerManager 收到后就会从中取出该 Binder 的名字和引用插入一张数据表中，这跟 DNS 中存储的域名到 IP 地址的映射原理类似，而对于网络访问来说，而 DNS 服务器也并不一定对每一个 IP 地址都有域名映射的记录，我们常常也会碰到直接通过 IP 地址访问服务器的情况，而 Binder 也一样并非一定要在 ServerManager 中有记录，很多时候 Binder Server 会将一个 Binder 实体封装进数据包传递给 Binder Client，而此时 Binder Server 会在该数据包中标注 Binder 实体的位置，Binder Driver 则会为该匿名的 Binder 生成实体节点和实体引用，并将该引用传递给 Binder Client，那么有人可能会有疑问，既然 ServerManager 的作用类似于 DNS 服务器，那么其本质是不是应该是一个 Binder Server 才对呢？答案是肯定的，ServerManager 就是一个标准的 Binder Server，并且在 Android 中约定其在 Binder 通信的过程中唯一标识（类似于 IP 地址）永远是 0。

ServerManager 既然也一个标准的 Binder Server，那么它也应该对外公布其可用的接口方法，这里你可以将它看作一个联想服务器，你既然能让客户端访问，总得给客户端可访问的接口和数据吧，那么在 ServerManager 中这部分接口方法的定义是在 IServiceManager 中的。

```
public interface IServiceManager extends IInterface {

    public IBinder getService(String name) throws RemoteException;

    public IBinder checkService(String name) throws RemoteException;

    public void addService(String name, IBinder service, boolean allowIsolated) throws
    RemoteException;

    public String[] listServices() throws RemoteException;

    public void setPermissionController(IPermissionController controller) throws
    RemoteException;

    static final String descriptor = "android.os.IServiceManager";

    int GET_SERVICE_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION;
    int CHECK_SERVICE_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+1;
    int ADD_SERVICE_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+2;
    int LIST_SERVICES_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+3;
    int CHECK_SERVICES_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+4;
    int SET_PERMISSION_CONTROLLER_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+5;
}
```

可以看到 IServiceManager 中所定义的方法并不多，因为我们的 ServiceManager 所承担的只是“DNS 服务器”的角色，其主要作用如上面所说，只是简单地将 Binder 以字符表示的名字转换为一

个引用。而对于在 Binder Client 端的 ServiceManagerNative 和 ServiceManagerProxy 则都实现了该接口。

```
public abstract class ServiceManagerNative extends Binder implements IServiceManager {
    static public IServiceManager asInterface(IBinder obj) {
        if (obj == null) {
            return null;
        }
        IServiceManager in = (IServiceManager)obj.queryLocalInterface(descriptor);
        if (in != null) {
            return in;
        }
        return new ServiceManagerProxy(obj);
    }

    public ServiceManagerNative() {
        attachInterface(this, descriptor);
    }

    public boolean onTransact(int code, Parcel data, Parcel reply, int flags) {
        try {
            switch (code) {
            case IServiceManager.GET_SERVICE_TRANSACTION: {
                // ......省略具体的方法逻辑......
            }
            case IServiceManager.CHECK_SERVICE_TRANSACTION: {
                // ......省略具体的方法逻辑......
            }
            case IServiceManager.ADD_SERVICE_TRANSACTION: {
                // ......省略具体的方法逻辑......
            }
            case IServiceManager.LIST_SERVICES_TRANSACTION: {
                // ......省略具体的方法逻辑......
            }
            case IServiceManager.SET_PERMISSION_CONTROLLER_TRANSACTION: {
                // ......省略具体的方法逻辑......
            }
            }
        } catch (RemoteException e) {
        }
        return false;
    }
    public IBinder asBinder() {
        return this;
    }
}

class ServiceManagerProxy implements IServiceManager {
    public ServiceManagerProxy(IBinder remote) {
        mRemote = remote;
    }

    public IBinder asBinder() {
        return mRemote;
    }

    public IBinder getService(String name) throws RemoteException {
        // ......省略具体的方法逻辑......
    }
```

```
    public IBinder checkService(String name) throws RemoteException {
        // ......省略具体的方法逻辑......
    }

    public void addService(String name, IBinder service, boolean allowIsolated) throws
    RemoteException {
        // ......省略具体的方法逻辑......
    }

    public String[] listServices() throws RemoteException {
        // ......省略具体的方法逻辑......
    }

    public void setPermissionController(IPermissionController controller) throws
    RemoteException {
        // ......省略具体的方法逻辑......
    }
    private IBinder mRemote;
}
```

此处限于篇幅我们仅对整个过程有个大致了解，而不去过分深究每个方法逻辑是如何完成。而对于其 Binder Server 端则是由 C 来实现，即便两者是使用的语言不同，但在 Binder Server 中其所定义的接口函数依然还是与 Binder Client 一致，其代码位于 frameworks\native\cmds\servicemanager 目录下的 service_manager.c 中，大家有兴趣可以自行查看这里不再赘述。同样，这里大家可以联想到我们上面所讲的代理模式，是不是很相似呢，可以说几乎就是一个套路的。

Android 的 Binder 机制是一个庞大的体系模块，可以毫不夸张地说，整个 Android 系统之所以能够有条不紊地运行，很大程度上是依赖于 Binder 机制。上面将整个 Binder 机制概括了一遍，其中并没有涉及具体的逻辑代码，看似简单实质要复杂得多，对于应用开发者来说，平时能接触到的最多最难懂的莫过于 View 的运行机制，但 Binder 比它还要复杂百倍，而且 View 的底层通信实现也会依赖于 Binder，所以，在这里笔者难以以一章甚至一本书的篇幅来剖析整个 Binder 机制，而且对于开发者来说，这么底层的东西没必要太过于接触，只要学会如何使用它即可。但是，我们又应该如何使用 Binder 来进行跨进程通信呢？上面我们曾提到，整个 Binder 机制由四大部分构成，而对于底层的 Binder Driver 和 ServerManager 来说我们没必要自己去实现，因为这部分逻辑 Android 已经帮我们封装得很完美了，而恰恰也是这一部分逻辑最为复杂，我们只需实现自己的 Binder Client 和 Binder Server 即可，在上面我们浅述 ServerManager 的时候说到其 Binder Server 部分的代码在 C 语言中实现，而且逻辑非常复杂，如果让我们普通的应用开发者来编写这么一个 Binder Server 难度会非常大，而 Android 也充分考虑到这一点，提供了一个简便的方式来生成 Binder Server 端，这种方式就是 AIDL，其全称为 Android Interface Description Language，即 Android 接口描述语言。授之以鱼不如授之以渔，这里我们直接来引入一个小例子来说明其作用，用代码来模拟去银行开户、销户、存取钱的过程，首先我们创建一个接口类来定义开户、销户和存取钱的方法。

```
public interface IBank {
    /**
     * 开户
     *
     * @param name     户主名字
     * @param password 密码
     * @return 开户信息
     */
```

```
    String openAccount(String name, String password);

    /**
     * 存钱
     *
     * @param money   金额
     * @param account 账户
     * @return 存钱信息
     */
    String saveMoney(int money, String account);

    /**
     * 取钱
     *
     * @param money    金额
     * @param account  账户
     * @param password 密码
     * @return 取钱信息
     */
    String takeMoney(int money, String account, String password);

    /**
     * 销户
     *
     * @param account  账户
     * @param password 密码
     * @return 销户信息
     */
    String closeAccount(String account, String password);
}
```

然后我们继承 Binder 创建一个 BankBinder 子类并实现上述接口。

```
public class BankBinder extends Binder implements IBank {
    @Override
    public String openAccount(String name, String password) {
        return name + " 开户成功！帐号为：" + UUID.randomUUID().toString();
    }

    @Override
    public String saveMoney(int money, String account) {
        return "账户：" + account + " 存入 " + money + " 单位：人民币";
    }

    @Override
    public String takeMoney(int money, String account, String password) {
        return "账户：" + account + " 支取 " + money + " 单位：人民币";
    }

    @Override
    public String closeAccount(String account, String password) {
        return account + " 销户成功！";
    }
}
```

逻辑都很简单，这里主要为了说明 AIDL 的作用，因此，具体的逻辑就不去斟酌了，每个方法都只是简单地返回一个普通的字符串，最后我们继承 Service 组件创建一个 BankService 子类来表示服务端。

```
public class BankService extends Service {
```

```
    @Override
    public IBinder onBind(Intent intent) {
        return new BankBinder();
    }
}
```

Service 的标准套路，重写 onBind 方法并返回一个 IBinder 对象，这里直接返回上面创建的 BankBinder 的实例对象，你可以把这个 BankService 看作一个 Binder Server，只不过它目前是与应用程序处于同一进程空间中。接口有了，服务端也有了，最后就剩客户端了，客户端这里用一个普通的 Activity 的子类来模拟。

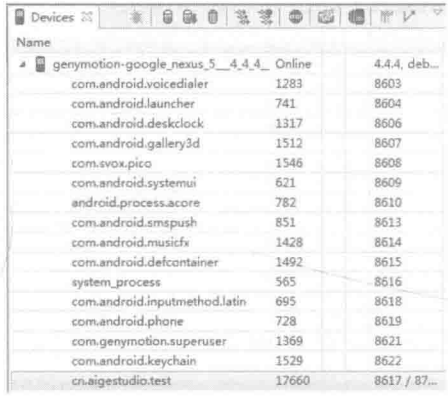
```
public class BankActivity extends Activity implements View.OnClickListener {
    private BankBinder mBankBinder;// 服务端的 Binder 对象

    private TextView tvMsg;// 显示处理结果的文本控件

    // 用于绑定 Service 的 ServiceConnection 对象
    private ServiceConnection conn = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            mBankBinder = (BankBinder) service;
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {

        }
    };

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

        // 绑定 Service
        bindService(new Intent("cn.aigestudio.test.aidl.bank.BankService"),conn, BIND_AUTO_CREATE);

        // 初始化文本控件
        tvMsg = (TextView) findViewById(R.id.aidl_bank_msg_tv);

        // 初始化按钮以及事件
        init(R.id.aidl_bank_open_btn);
        init(R.id.aidl_bank_save_btn);
        init(R.id.aidl_bank_take_btn);
        init(R.id.aidl_bank_close_btn);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        unbindService(conn);
    }

    private void init(int resID) {
        Button b = (Button) findViewById(resID);
        b.setOnClickListener(this);
    }

    @Override
```

```
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.aidl_bank_open_btn:
                tvMsg.setText(mBankBinder.openAccount("Aige", "123456789"));
                break;
            case R.id.aidl_bank_save_btn:
                tvMsg.setText(mBankBinder.saveMoney(6666666, "adasdasdd442f33"));
                break;
            case R.id.aidl_bank_take_btn:
                tvMsg.setText(mBankBinder.takeMoney(250, "adasdasdd442f33", "123456789"));
                break;
            case R.id.aidl_bank_close_btn:
                tvMsg.setText(mBankBinder.closeAccount("adasdasdd442f33", "123456789"));
                break;
        }
    }
}
```

代码逻辑都很简单，如果大家熟悉 Service 的使用，这段代码没有什么难度，在 Activity 的 onCreate 方法中我们就绑定了 Service，而在 ServiceConnection 对象的 onServiceConnected 方法中我们获取到 Service 所返回的 Binder 对象，并在点击事件中调用该 Binder 对象所描述的方法来执行代码逻辑，程序运行效果如图 18-4 所示。

以上为点击开户后服务端返回的信息，其他 3 个业务就不再展示了，布局很简单不再给出。大家可能会问这个示例与普通的 Service 绑定没有什么区别啊，的确，它就是一个简单的 Service 与 Activity 的绑定，不过我们的重点还不在这里。如果我们打开 Device Monitor 查看当前设备中运行的进程，会发现我们的示例其实只有一个进程存在，如图 18-5 所示。

▲图 18-4



▲图 18-5

当然这也是必然的，一个应用在正常情况下是只对应一个进程，现在我们将 BankService 置于另一个进程中试一下。

```
<service
    android:name=".aidl.bank.BankService"
    android:process="cn.aigestudio.BankService">
    <intent-filter>
        <action android:name="cn.aigestudio.test.aidl.bank.BankService"/>
    </intent-filter>
```

```
</service>
```

这时候再运行示例会发现程序报了一个类型转换的错误。

```
java.lang.ClassCastException: android.os.BinderProxy cannot be cast to cn.aigestudio.
test.aidl.bank.BankBinder
```

再看看 Device Monitor 中的进程，这时候就多了一个叫 cn.aigestudio.BankService 的进程，也就是上面我们设置的 BankService 所在的进程，这个进程与我们应用程序的进程明显已经不一样了，这时候我们无法直接通过 BankBinder 来调用另一个进程中的方法，这时候就需要使用 AIDL 来定义我们的接口并生成相应的编译文件，操作很简单，如果你用的是 Android Studio，直接选中包的状态下按 Alt+Insert 组合键，在弹出的菜单中选择 AIDL 即可，如图 18-6 所示。

▲图 18-6

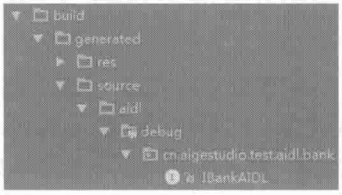
然后在出现的对话框中键入接口名字 Finish 即可，这里将该 AIDL 接口命名为 IBankAIDL。

```
package cn.aigestudio.test.aidl.bank;

interface IBankAIDL {
    /**
     * 开户
     *
     * @param name     户主名字
     * @param password 密码
     * @return 开户信息
     */
    String openAccount(String name, String password);

    /**
     * 存钱
     *
     * @param money   金额
     * @param account 账户
     * @return 存钱信息
     */
    String saveMoney(int money, String account);

    /**
     * 取钱
     *
     * @param money    金额
     * @param account  账户
     * @param password 密码
     * @return 取钱信息
     */
    String takeMoney(int money, String account, String password);

    /**
     * 销户
     *
     * @param account  账户
     * @param password 密码
     * @return 销户信息
     */
    String closeAccount(String account, String password);
}
```

这里只是将 IBank 中的接口方法简单地 Copy 了过来，这里要注意的是，在 Android Studio 中生成的 IBankAIDL 接口文件回存放在 main 目录下单独的 aidl 目录中，并与 Java 目录对应的包名一致，如图 18-7 所示。

定义好 AIDL 接口文件后，我们就可以 Build-Make Project 来生成对应的 Java 文件，该文件会被生成在 build/source/aidl/debug 目录中，也就是我们项目的编译目录下，如图 18-8 所示。

▲图 18-7



▲图 18-8

这个文件中的代码不像我们项目中其他的 Java 文件那样格式分明，而且更重要的是，其中有大量数据转储等逻辑，这里以 openAccount 方法为例，其在生成的 IBankAIDL 接口文件中是这样的。

```
@Override
public java.lang.String openAccount(java.lang.String name, java.lang.String password)
throws android.os.RemoteException {
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    java.lang.String _result;
    try {
        _data.writeInterfaceToken(DESCRIPTOR);
        _data.writeString(name);
        _data.writeString(password);
        mRemote.transact(Stub.TRANSACTION_openAccount, _data, _reply, 0);
        _reply.readException();
        _result = _reply.readString();
    } finally {
        _reply.recycle();
        _data.recycle();
    }
    return _result;
}
```

其主要作用只是对数据的一个打包封装，在 AIDL 文件中除了基本类型、String、List、Map 和 CharSequence 外，都需要使用 import 导入相应的类型或者包，即使这些类型与当前类型在同一包路径下，除此之外，如果你用 eclipse 开发，那么别忘了创建的 AIDL 文件的后缀名必须以.aidl 结尾。

接口有了，我们来看看如何使用。很简单，先来看服务端，我们修改 BankBinder 使其继承于生成的 AIDL 文件中的 Stub。

```
public class BankBinder extends IBankAIDL.Stub {
    @Override
    public String openAccount(String name, String password) {
        return name + " 开户成功！账号为：" + UUID.randomUUID().toString();
    }

    @Override
    public String saveMoney(int money, String account) {
        return "账户：" + account + " 存入 " + money + " 单位：人民币";
```

```
    }

    @Override
    public String takeMoney(int money, String account, String password) {
        return "账户: " + account + " 支取 " + money + " 单位: 人民币";
    }

    @Override
    public String closeAccount(String account, String password) {
        return account + " 销户成功! ";
    }
}
```

BankService 中的方法不变，只是修改了一下继承关系就搞定了对服务端的修改，而对于客户端呢？仅是 ServiceConnection 对象的 onServiceConnected 方法中获取 Binder 的方式不一样而已。

```
@Override
public void onServiceConnected(ComponentName name, IBinder service) {
    mBankBinder = IBankAIDL.Stub.asInterface(service);
}
```

这里我们通过 Stub 的 asInterface 来获取一个 IBankAIDL 对象，该对象会调用服务端 BankBinder 中对应的方法，客户端其余的代码仅是跟着变就行。

```
public class BankActivity extends Activity implements View.OnClickListener {
    private IBankAIDL mBankBinder;// 服务端的 Binder 对象

    private TextView tvMsg;// 显示处理结果的文本控件

    // 用于绑定 Service 的 ServiceConnection 对象
    private ServiceConnection conn = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            mBankBinder = IBankAIDL.Stub.asInterface(service);
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {

        }
    };

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

        // 绑定 Service
        bindService(new Intent("cn.aigestudio.test.aidl.bank.BankService"),conn,BIND_AUTO_CREATE);

        // 初始化文本控件
        tvMsg = (TextView) findViewById(R.id.aidl_bank_msg_tv);

        // 初始化按钮及事件
        init(R.id.aidl_bank_open_btn);
        init(R.id.aidl_bank_save_btn);
        init(R.id.aidl_bank_take_btn);
        init(R.id.aidl_bank_close_btn);
    }
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();
        unbindService(conn);
    }

    private void init(int resID) {
        Button b = (Button) findViewById(resID);
        b.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.aidl_bank_open_btn:
                try {
                    tvMsg.setText(mBankBinder.openAccount("Aige", "123456789"));
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
                break;
            case R.id.aidl_bank_save_btn:
                try {
                    tvMsg.setText(mBankBinder.saveMoney(6666666, "adasdasdd442f33"));
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
                break;
            case R.id.aidl_bank_take_btn:
                try {
                    tvMsg.setText(mBankBinder.takeMoney(250,"adasdasdd442f33","123456789"));
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
                break;
            case R.id.aidl_bank_close_btn:
                try {
                    tvMsg.setText(mBankBinder.closeAccount("adasdasdd442f33","123456789"));
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
                break;
        }
    }
}
```

注意，这里 RemoteException 是个必检异常，我们在这里捕获一下异常就是了，这时候再来运行一下示例，不会再有问题。具体的运行效果与上面我们给出的一致，这里就不再演示了。

不知道大家有没有注意，上述的示例服务端与客户端以及接口的定义已经有代理模式的影子存在了，可以细细体会。其实，从示例中我们也可以看到应用开发时所用到的跨进程通信也并不是很难，这最大程度上得益于 Android 对 Binder 的完美封装使得我们在开发的过程中不用去理会高深的 Binder Driver、ServerManager 等，再加上其为我们提供的自动化 AIDL 接口文件生成，更是极大地简化了开发者的工作。

18.8 代理模式实战

如本章开头所说，我们可以在日常生活中的很多地方看到代理模式的影子，代理的概念应用广泛。而在我们的程序语言中，代理模式的应用也是非常多，很多读者虽然没接触过设计模式，但是，可能在你现在或曾经编写的代码中就已经运用过代理模式了，对于我们平时应用开发来说并没有什么特别的代理模式例子，都是很普通的。这里以不同版本的 API 发送通知为例来阐述我们平时在应用开发中对代理模式的应用。

Android 中发送通知比较简单，我们只需要使用 NotificationManager 的 notify 方法发送一个通知即可。

```
public class ProxyActivity extends Activity implements View.OnClickListener {
    private NotificationManager nm;
    private NotificationCompat.Builder builder;

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

        nm = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);

        builder = new NotificationCompat.Builder(this);
        builder.setSmallIcon(R.drawable.ic_launcher)
                .setContentIntent(PendingIntent.getActivity(this, 0,
                        new Intent(this, NotifyActivity.class),
                        PendingIntent.FLAG_UPDATE_CURRENT));

        init(R.id.notify_proxy_send_btn);
    }

    private void init(int resID) {
        Button b = (Button) findViewById(resID);
        b.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.notify_proxy_send_btn:
                Notification n = builder.build();
                n.contentView = new RemoteViews(getPackageName(),
                R.layout.remote_notify_proxy_normal);
                nm.notify(0, n);
                break;
        }
    }
}
```

如上代码所示，当点击布局中以 notify_proxy_send_btn 为 ID 标识的按钮时就会发送一条通知，该通知我们使用了一个简单的自定义布局视图，如图 18-9 所示。

▲图 18-9

整个代码逻辑也很简单没有什么问题。但是大家知道 Android 的 API 版本迭代比较快，特别是 Notification 这块，很多 API 版本所提供的方法、样式等都不一样，再加上一些厂家的定制就更碎片化了。因此，虽说 Notification 比较简单，但是适配起来依然很困难，在笔者过往的经验中，会将 Notification 分为 4 类，一类是正常视图，也就是大家常看到状态栏中出现的高度为 64dp 的长条状通知视图；一类是在 API16 中引入的以 Style 方式展示的 MediaStyle、InboxStyle、BigTextStyle 和 BigPictureStyle 四种 Notification 风格样式；一类也是在 API16 中引入的可以将通知视图显示为 256dp 高度大视图的 bigContentView；最后一类则是在 L 中加入的 headsUpContentView。在实际测试的过程中 bigContentView 并不能在 API16、API17、API18 中很好地显示，但大部分 API19 的设备没有问题，锤子手机例外；而小米 2 虽然基于了 Android 5.0 也就是 API21 定制，但是它也没有使用到 Android 5.0 的 Notification。那么对于这样一个碎片化比较多的情况该如何通过代理模式去解决呢？很简单，在这种情况下我们为每种不同的 Notification 样式定义一个类，这里以正常的 64dp Height、256dp Height 和 headsUpContentView 为例，先定义一个抽象类表示通知。

```
public abstract class Notify {
    protected Context context;

    protected NotificationManager nm;
    protected NotificationCompat.Builder builder;

    public Notify(Context context) {
        this.context = context;

        nm=(NotificationManager)context.getSystemService(Context.NOTIFICATION_SERVICE);

        builder = new NotificationCompat.Builder(context);
        builder.setSmallIcon(R.drawable.ic_launcher)
                .setContentIntent(PendingIntent.getActivity(context, 0,
                        new Intent(context, NotifyActivity.class),
                        PendingIntent.FLAG_UPDATE_CURRENT));
    }

    /**
     * 发送一条通知
     */
    public abstract void send();

    /**
     * 取消一条通知
     */
    public abstract void cancel();
}
```

Notify 类很简单，首先是声明了 NotificationManager 和 NotificationCompat.Builder 两个成员变量用来处理与通知相关的一些逻辑，这两个变量所有子类都会用到，继而在构造方法中初始化它们，并为 builder 设置一些初始化参数。两个抽象方法 send 和 cancel 均由具体的子类去实现，如 64dp

Heigth 的 Notification 实现类。

```
public class NotifyNormal extends Notify {
    public NotifyNormal(Context context) {
        super(context);
    }

    @Override
    public void send() {
        Notification n = builder.build();
        n.contentView = new RemoteViews(context.getPackageName(),
        R.layout.remote_notify_proxy_normal);
        nm.notify(0, n);
    }

    @Override
    public void cancel() {
        nm.cancel(0);
    }
}
```

如上代码所示，NotifyNormal 类中只是简单地重写了 send 和 cancel 方法，send 方法中发送一个自定义视图的普通 Notification，而对于 256dp Height 的 bigContentView 呢，实现也很简单。

```
public class NotifyBig extends Notify {
    public NotifyBig(Context context) {
        super(context);
    }

    @Override
    public void send() {
        Notification n = builder.build();
        n.contentView = new RemoteViews(context.getPackageName(),
        R.layout.remote_notify_proxy_normal);
        n.bigContentView = new RemoteViews(context.getPackageName(),
        R.layout.remote_notify_proxy_big);
        nm.notify(0, n);
    }

    @Override
    public void cancel() {
        nm.cancel(0);
    }
}
```

仅仅是在 NotifyNormal 的 send 方法基础上增加一个 bigContentView，这里要说明一点，为什么我们要增加而不直接去掉普通 contentView 的设置呢，因为在 Android 4.0 也就是 API16 以上的系统中，通知列表里只允许位于列表顶端的一个 bigContentView 展示，而其他的 Notification 均会以普通 contentView 的形式展示，所以，这里只是增加而不是修改。最后对于 L 以上也就是 API20 开始的系统我们还可以为 Notification 增加一个 headsUpContentView，当我们的 App 以全屏的方式展示时如果收到通知，这个 headsUpContentView 就会浮动展示于屏幕顶部，类似于 Toast。

```
public class NotifyHeadsUp extends Notify {
    public NotifyHeadsUp(Context context) {
        super(context);
    }
```

```
    @Override
    public void send() {
        Notification n = builder.build();
        n.contentView = new RemoteViews(context.getPackageName(),
        R.layout.remote_notify_proxy_normal);
        n.bigContentView = new RemoteViews(context.getPackageName(),
        R.layout.remote_notify_proxy_big);
        n.headsUpContentView = new RemoteViews(context.getPackageName(),
        R.layout.remote_notify_proxy_normal);
        nm.notify(0, builder.build());
    }

    @Override
    public void cancel() {
        nm.cancel(0);
    }
}
```

同样地，我们也只是在 NotifyBig 的基础上增加了对 headsUpContentView 的设置，没有复杂的逻辑。最后定义一个代理类来整合上面的几个类。

```
public class NotifyProxy extends Notify {
    private Notify notify;

    public NotifyProxy(Context context) {
        super(context);

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            notify = new NotifyHeadsUp(context);
        } else if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
            notify = new NotifyBig(context);
        } else {
            notify = new NotifyNormal(context);
        }
    }

    @Override
    public void send() {
        notify.send();
    }

    @Override
    public void cancel() {
        notify.cancel();
    }
}
```

在 NotifyProxy 类中定义一个 Notify 类型的成员变量，在构造方法里根据 SDK 版本的不同去实例化不同的 Notify 子类，最终由该类的 send 和 cancel 方法去调用不同的逻辑实现，这样一来，我们的客户端也就是我们的 Activity 类中就很简单了，直接在事件中调用 NotifyProxy 中的方法即可。

```
public class ProxyActivity extends Activity implements View.OnClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_notify_proxy);
```

```
        init(R.id.notify_proxy_send_btn);
    }

    private void init(int resID) {
        Button b = (Button) findViewById(resID);
        b.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
                new NotifyProxy(this).send();
                break;
        }
    }
}
```

代码以及布局都很简单，具体运行效果就不再给出了。在实际开发过程中遇到类似的情况都可以使用代理模式来处理，当然代理模式的应用范畴也并不局限于此，这里限于篇幅不再一一列举。细心的读者可能会发现，上面示例代码有一点问题，3 个 Notify 的子类 NotifyNormal、NotifyBig 和 NotifyHeadsUp 从类的功能实现上来讲，无非后者都是前者功能的增加，比如 NotifyHeadsUp 相对与 NotifyBig 来说无非就是新增了对 headsUp 视图的设置，而 NotifyBig 相对于 NotifyNormal 来说也只是增加了对 bigContentView 的设置，但是在示例代码中，我们在每个类中都重写了上一个类中重复的代码，有没有简单的方式呢？答案是肯定的，在我们讲了结构型模式的另一种设计模式装饰模式后，你就知道该如何简化我们的逻辑了。

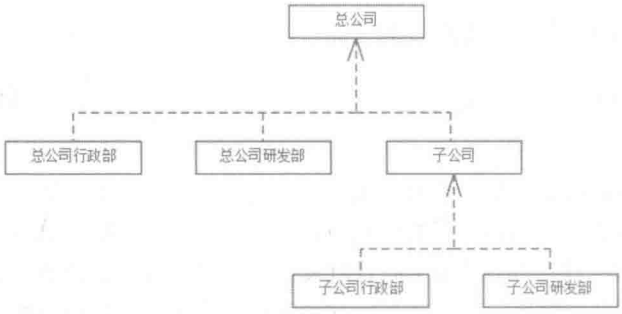
18.9 总结

代理模式应用广泛，这估计是本章说的最多的一句话了，但是事实就是那样，再往后我们讲到的其他形式的结构型模式中，你都可以看到代理模式的影子，有些模式单独作为一种设计模式，倒不如说是对代理模式的一种针对性优化。而且代理模式几乎没有什么缺点可言，它是细分化至很小的一种模式，要真的说一个缺点，那么就是所有设计模式的通病：对类的增加，不过在这种孰优孰劣的局势下，就算对类的稍微增加又何妨呢？

第 19 章　物以类聚——组合模式

19.1 组合模式介绍

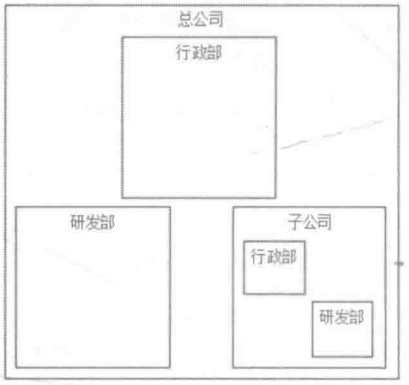
组合模式（Composite Pattern）也称为部分整体模式（Part-Whole Pattern），结构型设计模式之一，组合模式比较简单，它将一组相似的对象看作一个对象处理，并根据一个树状结构来组合对象，然后提供一个统一的方法去访问相应的对象，以此忽略掉对象与对象集合之间的差别。生活中一个比较典型的例子就是组织结构的树状图，如图 19-1 所示。



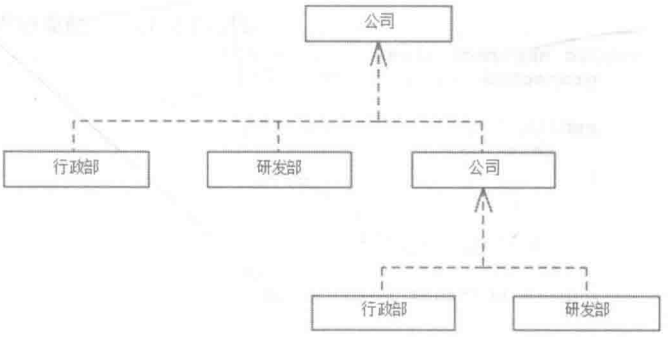
▲图 19-1

上面是一个公司的组织结构树状图，其中总公司下有行政部与研发部，而且总公司下属还有一个子公司，虽然子公司也包含行政部与研发部，但是从总公司的角度来看子公司就是一个独立的个体，与总公司所属的行政部和研发部平级，如果我们以一个嵌套盒子的形式来展示将会更加直观，比如我们将上述的树形结构转为一个嵌套盒子，总公司则是最外层的盒子，里面包含 3 个小盒子，分别表示总公司的行政部、研发部和子公司，而子公司内部又包含有研发部与行政部，如图 19-2 所示。

在这么一个结构中大家可以看到虽然总公司和子公司其本质不一样，但是它在我们的组织结构中是一样的，我们可以把它们看作一个抽象的公司，在组合模式中我们将这样的一个拥有分支的节点称之为枝干构件，位于树状结构顶部的枝干结构比较特殊，我们称为根结构件，因为其为整个树状图的始端，同样对于像行政部和研发部这样没有分支的结构，我们则称为叶子构件，这样的一个结构就是组合模式的雏形，如图 19-3 所示。



▲图 19-2



▲图 19-3

19.2 组合模式的定义

将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

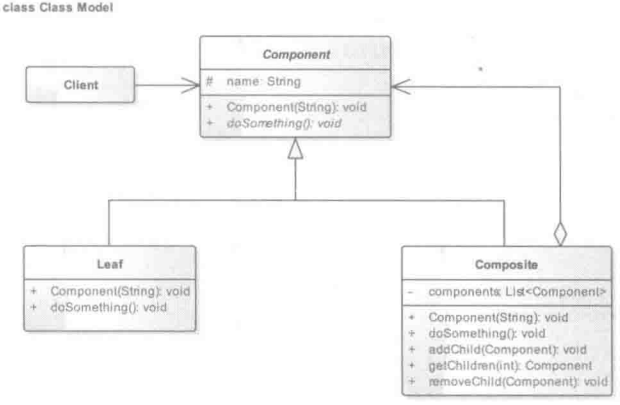
19.3 组合模式的使用场景

表示对象的部分-整体层次结构时。

从一个整体中能够独立出部分模块或功能的场景。

19.4 组合模式的 UML 类图

UML 类图如图 19-4 所示。



▲图 19-4

根据类图我们可以得出如下一个组合模式的通用模式代码。

程序清单 19-1　抽象根节点

```
public abstract class Component {
    protected String name;// 节点名

    public Component(String name) {
        this.name = name;
    }

    /**
     * 具体的逻辑方法由子类实现
     */
    public abstract void doSomething();
}
```

程序清单 19-2　具体枝干节点

```
public class Composite extends Component {
    /**
     * 存储节点的容器
     */
    private List<Component> components = new ArrayList<>();

    public Composite(String name) {
        super(name);
    }

    @Override
    public void doSomething() {
        System.out.println(name);
        if (null != components) {
            for (Component c : components) {
                c.doSomething();
            }
        }
    }

    /**
     * 添加子节点
     *
     * @param child
     *          子节点
     */
    public void addChild(Component child) {
        components.add(child);
    }

    /**
     * 移除子节点
     *
     * @param child
     *          子节点
     */
    public void removeChild(Component child) {
        components.remove(child);
    }

    /**
     * 获取子节点
     *
```

```
     * @param index
     *          子节点对应下标
     * @return 子节点
     */
    public Component getChildren(int index) {
        return components.get(index);
    }
}
```

程序清单 19-3 具体叶子节点

```
public class Leaf extends Component {
    public Leaf(String name) {
        super(name);
    }

    @Override
    public void doSomething() {
        System.out.println(name);
    }
}
```

程序清单 19-4 客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造一个根节点
        Composite root = new Composite("Root");

        // 构造两个枝干节点
        Composite branch1 = new Composite("Branch1");
        Composite branch2 = new Composite("Branch2");

        // 构造两个叶子节点
        Leaf leaf1 = new Leaf("Leaf1");
        Leaf leaf2 = new Leaf("Leaf2");

        // 将叶子节点添加至枝干节点中
        branch1.addChild(leaf1);
        branch2.addChild(leaf2);

        // 将枝干节点添加至根节点中
        root.addChild(branch1);
        root.addChild(branch2);

        // 执行方法
        root.doSomething();
    }
}
```

输出结果如下：

```
Root
Branch1
Leaf1
Branch2
Leaf2
```

角色介绍。

- Component：抽象根节点，为组合中的对象声明接口。在适当的情况下，实现所有类共有接口的缺省行为。声明一个接口用于访问和管理 Component 的子节点。可在递归结构中定义一个接

口，用于访问一个父节点，并在合适的情况下实现它。

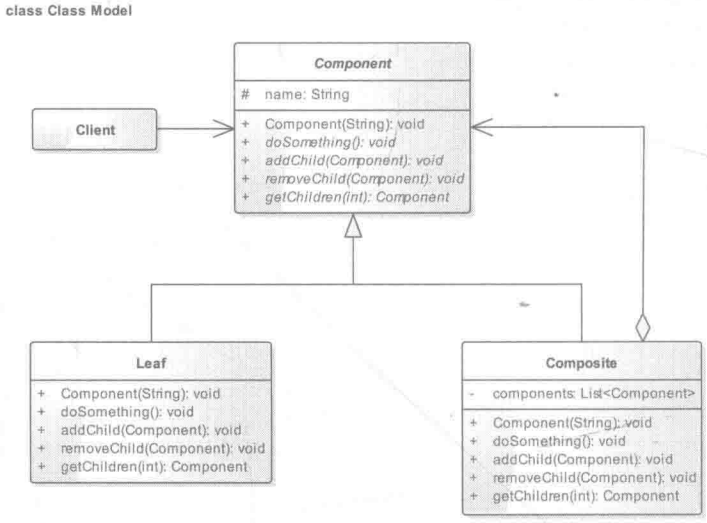
- Composite：定义有子节点的那些枝干节点的行为，存储子节点，在 Component 接口中实现与子节点有关的操作。
- Leaf：在组合中表示叶子节点对象，叶子节点没有子节点， 在组合中定义节点对象的行为。
- Client：通过 Component 接口操纵组合节点的对象。

上述的 UML 类图及其代码实现中有一个很大的弊端不知道大家注意到了没有，我们在 Client 客户类中直接使用了 Component 的实现类。

程序清单 19-5　客户类代码片段

```
public class Client {
    public static void main(String[] args) {
        // 构造一个根节点
        Composite root = new Composite("Root");

        // 构造两个枝干节点
        Composite branch1 = new Composite("Branch1");
        Composite branch2 = new Composite("Branch2");

        // 构造两个叶子节点
        Leaf leaf1 = new Leaf("Leaf1");
        Leaf leaf2 = new Leaf("Leaf2");
    }
}
```

这与本书第一章中所讲到的依赖倒置原则相违背，我们所定义的抽象 Component 在这里起的作用就不大，既然是面向接口编程，那么我们就该把更多的把焦点放在接口的设计上，如果我们稍微修改上面的类图将位于 Composite 中的一些实现方法定义到 Component 中，那么我们会得到一个不一样的组合模式，如图 19-5 所示。



▲图 19-5

像这样将组合所使用的方法定义在抽象类的方式称为透明的组合模式，而上面我们所说的组合模式则称为安全的组合模式。透明组合模式中不管是叶子节点还是枝干节点都有着相同的结构，那意味着我们无法通过 getChildren 方法得到子节点的类型，而必须在方法实现的内部进行判断，我们来看看这种方式的组合模式 UML 类图代码。

程序清单 19-6　透明的组合模式抽象根节点

```
public abstract class Component {
    protected String name;// 节点名

    public Component(String name) {
        this.name = name;
    }

    /**
     * 具体的逻辑方法由子类实现
     */
    public abstract void doSomething();

    /**
     * 添加子节点
     *
     * @param child
     *          子节点
     */
    public abstract void addChild(Component child);

    /**
     * 移除子节点
     *
     * @param child
     *          子节点
     */
    public abstract void removeChild(Component child);

    /**
     * 获取子节点
     *
     * @param index
     *          子节点对应下标
     * @return 子节点
     */
    public abstract Component getChildren(int index);
}
```

程序清单 19-7　透明的组合模式具体枝干节点

```
public class Composite extends Component {
    /**
     * 存储节点的容器
     */
    private List<Component> components = new ArrayList<>();

    public Composite(String name) {
        super(name);
    }

    @Override
    public void doSomething() {
        System.out.println(name);
```

```
        if (null != components) {
            for (Component c : components) {
                c.doSomething();
            }
        }
    }

    @Override
    public void addChild(Component child) {
        components.add(child);
    }

    @Override
    public void removeChild(Component child) {
        components.remove(child);
    }

    @Override
    public Component getChildren(int index) {
        return components.get(index);
    }
}
```

程序清单 19-8　透明的组合模式叶子节点

```
public class Leaf extends Component {
    public Leaf(String name) {
        super(name);
    }

    @Override
    public void doSomething() {
        System.out.println(name);
    }

    @Override
    public void addChild(Component child) {
        throw new UnsupportedOperationException("叶子节点没有子节点");
    }

    @Override
    public void removeChild(Component child) {
        throw new UnsupportedOperationException("叶子节点没有子节点");
    }

    @Override
    public Component getChildren(int index) {
        throw new UnsupportedOperationException("叶子节点没有子节点");
    }
}
```

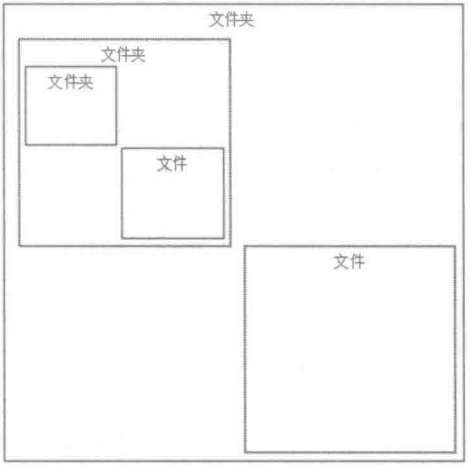
程序清单 19-9　客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造一个根节点
        Component root = new Composite("Root");

        // 构造两个枝干节点
        Component branch1 = new Composite("Branch1");
        Component branch2 = new Composite("Branch2");

        // 构造两个叶子节点
```

```
        Component leaf1 = new Leaf("Leaf1");
        Component leaf2 = new Leaf("Leaf2");

        // 将叶子节点添加至枝干节点中
        branch1.addChild(leaf1);
        branch2.addChild(leaf2);

        // 将枝干节点添加至根节点中
        root.addChild(branch1);
        root.addChild(branch2);

        // 执行方法
        root.doSomething();
    }
}
```

19.5 组合模式的简单实现

对于程序员来说，介绍一些组织结构想必一定会很陌生，但是即便如此，我们也能在其他地方看到组合模式的影子，如果你使用过电脑，知道什么是操作系统，那么对文件和文件夹这两个概念一定不会陌生，什么是文件呢？文件就是可以被具体程序执行的对象，那什么又是文件夹呢？文件夹就是可以存放文件和文件夹的对象如图 19-6 所示。



▲图 19-6

如此一看，操作系统的文件系统其实就是一种典型的组合模式例子，在这里我们就以此为例，看看一个简单的文件系统是如何构成的，首先声明一个 Dir 抽象类表示文件和文件夹。

程序清单 19-10　表示文件和文件夹的抽象类

```
public abstract class Dir {
    /**
     * 声明一个 List 成员变量存储文件夹下的所有元素
     */
    protected List<Dir> dirs = new ArrayList<>();
```

```
    private String name;// 当前文件或文件夹名

    public Dir(String name) {
        this.name = name;
    }

    /**
     * 添加一个文件或文件夹
     *
     * @param dir
     *          文件或文件夹
     */
    public abstract void addDir(Dir dir);

    /**
     * 移除一个文件或文件夹
     *
     * @param dir
     *          文件或文件夹
     */
    public abstract void rmDir(Dir dir);

    /**
     * 清空文件夹下所有元素
     */
    public abstract void clear();

    /**
     * 输出文件夹目录结构
     */
    public abstract void print();

    /**
     * 获取文件夹下所有的文件或子文件夹
     *
     * @return 文件夹下所有的文件或子文件夹
     */
    public abstract List<Dir> getFiles();

    /**
     * 获取文件或文件夹名
     *
     * @return 文件或文件夹名
     */
    public String getName() {
        return name;
    }
}
```

在该抽象类中我们定义了相关的抽象方法，大家可以看到这里用到的就是所谓的透明的组合模式，这里要实现的功能很简单，只是简单地打印一下目录结构，因此，声明一个成员变量来存储当前文件或文件夹的名称并提供对应的 getter 方法。接下来就是具体的文件夹类，该类中我们实现具体的文件夹方法逻辑：

程序清单 19-11　表示文件夹的类

```
public class Folder extends Dir {

    public Folder(String name) {
        super(name);
```

```
    }

    @Override
    public void addDir(Dir dir) {
        dirs.add(dir);
    }

    @Override
    public void rmDir(Dir dir) {
        dirs.remove(dir);
    }

    @Override
    public void clear() {
        dirs.clear();
    }

    @Override
    public void print() {
        System.out.print(getName() + "(");
        Iterator<Dir> iter = dirs.iterator();
        while (iter.hasNext()) {
            Dir dir = iter.next();
            dir.print();
            if (iter.hasNext()) {
                System.out.print(", ");
            }
        }
        System.out.print(")");
    }

    @Override
    public List<Dir> getFiles() {
        return dirs;
    }
}
```

像 addDir、rmDir 在这里的方法中实现逻辑都比较简单，这里主要就是 print 方法用来输出文件夹的目录结构，该方法逻辑也比较简单，首先输出自己的名字，也就是当前文件夹名，然后迭代遍历子元素，调用子元素的 print 方法输出其目录结构，如果遇到子元素还是个文件夹，那么递归遍历直至所有的输出元素均为文件为止。接下来看看文件类的实现。

程序清单 19-12　表示文件的类

```
public class File extends Dir {

    public File(String name) {
        super(name);
    }

    @Override
    public void addDir(Dir dir) {
        throw new UnsupportedOperationException("文件对象不支持该操作");
    }

    @Override
    public void rmDir(Dir dir) {
        throw new UnsupportedOperationException("文件对象不支持该操作");
    }
```

```
    @Override
    public void clear() {
        throw new UnsupportedOperationException("文件对象不支持该操作");
    }

    @Override
    public void print() {
        System.out.print(getName());
    }

    @Override
    public List<Dir> getFiles() {
        throw new UnsupportedOperationException("文件对象不支持该操作");
    }
}
```

文件类相对于文件夹类来说既不支持添加也不支持删除，因为文件不能作为文件夹来使用，它本是文件系统中的最小分割单位，因此，这里我们虽然实现了 Dir 中的一些添加、删除等操作方法，但是，其实现逻辑只是抛出一个不支持异常，因为就文件本身来说就不支持对应的操作。最后，在一个客户类中构建目录结构并输出。

程序清单 19-13　客户类

```
public class Client {
    public static void main(String[] args) {
        // 构造一个目录对象表示 C 盘根目录
        Dir diskC = new Folder("C");

        // C 盘根目录下有一个文件 ImbaMallLog.txt
        diskC.addDir(new File("ImbaMallLog.txt"));

        // C 盘根目录下还有 3 个子目录 Windows、PerfLogs、Program File
        Dir dirWin = new Folder("Windows");

        // Windows 目录下有文件 explorer.exe
        dirWin.addDir(new File("explorer.exe"));
        diskC.addDir(dirWin);

        // PerfLogs 目录
        Dir dirPer = new Folder("PerfLogs");

        // PerfLogs 目录下有文件 null.txt
        dirPer.addDir(new File("null.txt"));
        diskC.addDir(dirPer);

        // Program File 目录
        Dir dirPro = new Folder("Program File");

        // Program File 目录下有文件 ftp.txt
        dirPro.addDir(new File("ftp.txt"));
        diskC.addDir(dirPro);

        // 打印出文件结构
        diskC.print();
    }
}
```

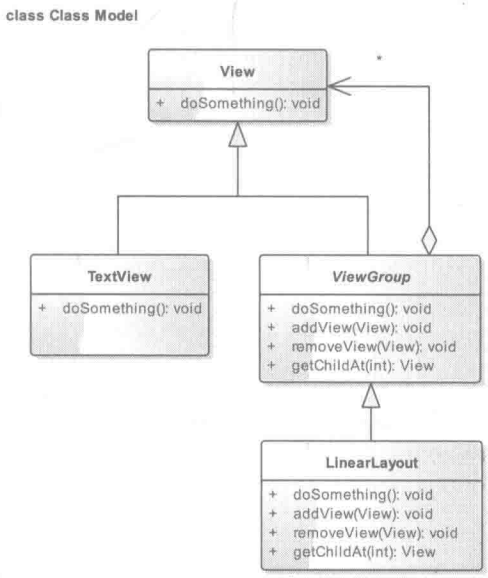
输出结果如下：

```
C(ImbaMallLog.txt, Windows(explorer.exe), PerfLogs(null.txt), Program File(ftp.txt))
```

这里我们以括号作为一个文件夹的内容范围，如上输出所示，C 盘文件夹下有 3 个子文件夹 Windows、PerfLogs 和 Program File 以及一个文件 ImbaMallLog.txt，而且 3 个子文件夹中还各自包含有子文件，一个典型的树状嵌套结构，这就是组合模式。

19.6 Android 源码中的模式实现

Android 源码中关于组合模式有一个非常经典实现，前面许多章节中我们所列举的一些源码中的设计模式实现都并非那么经典，因为在很多情况下会根据实际应用来调整一些设计模式的结构，所以，会造成很多给人感觉不是那么贴切的实现，但是在 Android 源码中关于组合模式这一块却很经典，我们几乎每天都会使用到，那就是 View 和 ViewGroup 的嵌套组合，如图 19-7 所示。



▲图 19-7

如图 19-7 所示，这里省略了 View 和 ViewGroup 类中的一些方法，在 Android 的这个视图层级中，容器一定是 ViewGroup，而且只有 ViewGroup 才能包含其他的 View，比如 LinearLayout 能包含 TextView、Button、CheckBox 等，但是反过来 TextView 是不能包含 LinearLayout 的，因为 TextView 直接继承于 View，其并非一个容器。这里 View 的视图层级中使用到的其实是一种安全的设计模式，如果你还不懂什么叫安全的设计模式可以回头再看几遍上面的内容，ViewGroup 相对于 View 只是多了几个对视图操作的方法。

程序清单 19-14　视图类

```
public class View {
    /**
     * 一个公共方法 不管是 View 还是子类都可调用
```

```
     */
    public void doSomething() {
        // 具体逻辑
    }
}
```

程序清单 19-15　容器类

```
public abstract class ViewGroup extends View {
    /**
     * 添加一个视图
     *
     * @param view
     *          视图
     */
    public void addView(View view) {
        // ......具体逻辑......
    }

    /**
     * 移除一个视图
     *
     * @param view
     *          视图
     */
    public void removeView(View view) {
        // ......具体逻辑......
    }

    /**
     * 获取对应下标的视图
     *
     * @param index
     *          下标
     * @return 视图
     */
    public View getChildAt(int index) {
        // ......具体逻辑......
        return null;
    }
}
```

19.7　为什么 ViewGroup 有容器的功能

要回答这个问题，就要先了解 View 类与 ViewGroup 类的差别在哪，首先我们知道 ViewGroup 是继承于 View 类的。

```
public abstract class ViewGroup extends View implements ViewParent, ViewManager {
    // ......省略具体的逻辑代码......
}
```

从继承的角度来说 ViewGroup 拥有 View 类所有的非私有方法，既然如此，两者的差别就在于 ViewGroup 所实现的 ViewParent 和 ViewManager 接口上，而事实也是如此，ViewManager 接口定义了 addView、removeView 等对子视图操作的方法。

```
public interface ViewManager {
    public void addView(View view, ViewGroup.LayoutParams params);
```

```
    public void updateViewLayout(View view, ViewGroup.LayoutParams params);

    public void removeView(View view);
}
```

而 ViewParent 则定义了刷新容器的接口 requestLayout 和其他一些焦点事件的处理的接口。

```
public interface ViewParent {
    /**
     * 请求重新布局
     */
    public void requestLayout();

    /**
     * 是否已经请求布局 这里有一点要注意 当我们调用 requestLayout 请求布局后，这一过程并非是立即执行的
     * Android 会将请求布局的操作以消息的形式发送至主线程的 Handler 并由其分发处理
     * 因此在调用 requestLayout 方法请求布局到布局真正接收到重新布局的命令时需要一段时间间隔
     */
    public boolean isLayoutRequested();

    /**
     * 当子视图需要收集视图层次中透明区域并报告给窗口排版组件时调用，需要在视图层次中"打洞"的视图，如
    SurfaceView 可以利用该 API
     * 来提高系统性能 当视图层次中没有这样的视图时，不需要该优化，使用它会稍微降低一些视图层次的性能
     *
     */
    public void requestTransparentRegion(View child);

    /**
     * 无效化子视图
     */
    public void invalidateChild(View child, Rect r);

    /**
     * 无效化子视图的部分或全部区域
     */
    public ViewParent invalidateChildInParent(int[] location, Rect r);

    /**
     * 获取当前 View 的 ViewParent 对象
     */
    public ViewParent getParent();

    /**
     * 请求子视图焦点
     */
    public void requestChildFocus(View child, View focused);

    // ......省略一些不常用的方法......
}
```

其中有一些方法我们是比较常见的也经常会用到，如 requestLayout、bringChildToFront 等，ViewGroup 除了所实现的这两个接口与 View 不一样外，还有重要的一点就是 ViewGroup 是抽象类，其将 View 中的 onLayout 方法重置为抽象方法，也就是说容器子类必须实现该方法来实现布局定位，我们知道 View 中的该方法是个空实现，因为对于一个普通的 View 来说该方法并没有什么实现价值，但是 ViewGroup 就不一样，要必须实现。除此之外，在 View 中比较重要的两个测绘流程的方法 onMeasure 和 onDraw 在 ViewGroup 中都没有被重写，相对于 onMeasure 方法，在 ViewGroup 中

增加了一些计算子 View 的方法，如 measureChildren、measureChildrenWithMargins 等；而对于 onDraw 方法，ViewGroup 定义了一个 dispatchDraw 方法来调用其每一个子 View 的 onDraw 方法，由此可见，ViewGroup 真的就象一个容器一样，其职责只是负责对子元素的操作而非具体的个体行为。

19.8 总结

我们平时在 Android 开发的过程中组合模式的应用并不算多，组合模式更适用于对一些界面 UI 的架构设计上，当然，绝大多数情况下，这部分代码都会由相应的程序语言提供，比如 Java 的 AWT、Android 和 iOS 的 UI 框架等，真正需要开发者去实现的不多，笔者在开发的过程中曾经使用到组合模式的案例也很少，唯一的一个组件实现是 Android 的目录选择器 DirSelector，其实现逻辑与本文开头组合模式简单实现的例子类似，这里不再累赘。

组合模式与我们前面所讲到的解释器模式有一定的类同，两者在迭代对象时都涉及递归的调用，但是组合模式所提供的属性层次结构使得我们能够一视同仁地对待单个对象和对象集合，不过这是以牺牲类的单一原则换来的，而且组合模式是通过继承来实现的，这样的做法缺少弹性。所以，和其他许多设计模式一样，在实用设计模式之前一定要想清楚利弊关系，不要造成设计模式的滥用。下面我们还是来罗列一下组合模式的优、缺点。

组合模式的优点：

组合模式可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，它让高层模块忽略了层次的差异，方便对整个层次结构进行控制。

高层模块可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了高层模块的代码。

在组合模式中增加新的枝干构件和叶子构件都很方便，无须对现有类库进行任何修改，符合“开闭原则”。

组合模式为树形结构的面向对象实现提供了一种灵活的解决方案，通过叶子对象和枝干对象的递归组合，可以形成复杂的树形结构，但对树形结构的控制却非常简单。

组合模式的缺点：

在新增构件时不好对枝干中的构件类型进行限制，不能依赖类型系统来施加这些约束，因为在大多数情况下，它们都来自于相同的抽象层，此时，必须行时进行类型检查来实现，这个实现过程较为复杂。

第 20 章　得心应手的“粘合剂”——适配器模式

20.1 适配器模式介绍

适配器模式在我们的开发中使用率极高，从代码中随处可见的 Adapter 就可以判断出来。从最早的 ListView、GridView 到现在最新的 RecyclerView 都需要使用 Adapter，并且在开发中我们遇到的优化问题、出错概率较大的地方也基本都出自 Adapter，这是一个让人又爱又恨的角色。

说到底，适配器是将两个不兼容的类融合在一起，它有点像粘合剂，将不同的东西通过一种转换使得它们能够协作起来。例如，经常碰到要在两个没有关系的类型之间进行交互，第一个解决方案是修改各自类的接口，但是如果没有源代码或者我们不愿意为了一个应用而修改各自的接口，此时怎么办？这种情况我们往往会使用一个 Adapter，在这两种接口之间创建一个“混血儿”接口，这个 Adapter 会将这两个接口进行兼容，在不修改原有代码的情况下满足需求。

20.2 适配器模式的定义

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

20.3 适配器模式的使用场景

（1）系统需要使用现有的类，而此类的接口不符合系统的需要，即接口不兼容。

（2）想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

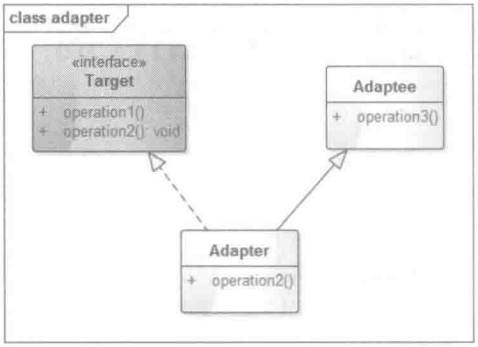
（3）需要一个统一的输出接口，而输入端的类型不可预知。

20.4 适配器模式的 UML 类图

UML 类图如图 20-1 所示。

适配器模式也分两种，即类适配器模式和对象适配器模式，首先学习类适配器模式，结构图如

图 20-1 所示。



▲图 20-1

如图 20-1 所示，类适配器是通过实现 Target 接口以及继承 Adaptee 类来实现接口转换，例如，目标接口需要的是 operation2，但是 Adaptee 对象只有一个 operation3，因此就出现了不兼容的情况。此时通过 Adapter 实现一个 operation2 函数将 Adaptee 的 operation3 转换为 Target 需要的 operation2，以此实现兼容。

角色介绍。

- Target：目标角色，也就是所期待得到的接口。注意：由于这里讨论的是类适配器模式，因此目标不可以是类。
- Adaptee：现在需要适配的接口。
- Adapter：适配器角色，也是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

20.5 适配器模式应用的简单示例

用电源接口做例子，笔记本电脑的电源一般都是用 5V 电压，但是我们生活中的电线电压一般都是 220V。这个时候就出现了不匹配的状况，在软件开发中我们称之为接口不兼容，此时就需要适配器来进行一个接口转换。在软件开发中有一句话正好体现了这点：任何问题都可以加一个中间层来解决。这个层我们可以理解为这里的 Adapter 层，通过这层来进行一个接口转换就达到了兼容的目的。

在上述电源接口这个示例中，5V 电压就是 Target 接口，220V 电压就是 Adaptee 类，而将电压从 220V 转换到 5V 就是 Adapter。

20.5.1 类适配器模式

具体程序如下所示。

```
    // Target 角色
public interface FiveVolt {
    public int getVolt5();
```

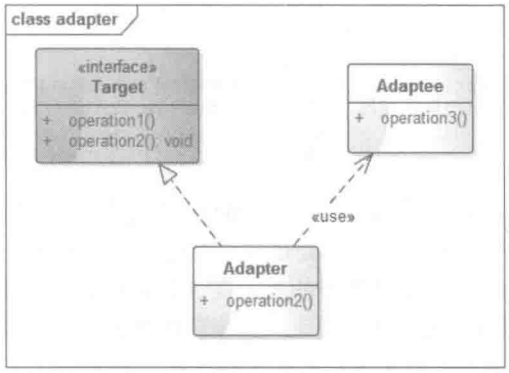
```
}
 // Adaptee 角色,需要被转换的对象
public class Volt220 {
    public int getVolt220() {
        return 220;
    }
}

// Adapter 角色，将 220V 的电压转换成 5V 的电压
public class VoltAdapter extends Volt220 implements FiveVolt {
    @Override
    public int getVolt5() {
        return 5;
    }
}
```

Target 角色给出了需要的目标接口，而 Adaptee 类则是需要被转换的对象。Adapter 则是将 Volt220 转换成 Target 的接口。对应的 Target 的目标是要获取 5V 的输出电压，而 Adaptee 正常输出电压是 220V，此时就需要电源适配器类将 220V 的电压转换为 5V 电压，解决接口不兼容的问题。

```
public class Test {
    public static void main(String[] args) {
        VoltAdapter adapter = new VoltAdapter();
        System.out.println("输出电压 : " + adapter.getVolt5());
    }
}
```

20.5.2 对象适配器模式

与类的适配器模式一样，对象的适配器模式把被适配的类的 API 转换成为目标类的 API，与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到 Adaptee 类，而是使用代理关系连接到 Adaptee 类，UML 图如图 20-2 所示。



▲图 20-2

从图 20-2 中可以看出，Adaptee 类（Volt220）并没有 getVolt5()方法，而客户端则期待这个方法。为使客户端能够使用 Adaptee 类，需要提供一个包装类 Adapter。这个包装类包装了一个 Adaptee 的实例，从而此包装类能够把 Adaptee 的 API 与 Target 类的 API 衔接起来。Adapter 与 Adaptee 是委派关系，这决定了适配器模式是对象的。示例代码如下。

```
/**
 * Target 角色
 */
public interface FiveVolt {
    public int getVolt5();
}

/**
 * Adaptee 角色,需要被转换的对象
 */
public class Volt220 {
    public int getVolt220() {
        return 220;
    }
}

// 对象适配器模式
public class VoltAdapter implements FiveVolt {

    Volt220 mVolt220;

    public VoltAdapter(Volt220 adaptee) {
        mVolt220 = adaptee;
    }

    public int getVolt220() {
        return mVolt220.getVolt220();
    }

    @Override
    public int getVolt5() {
        return 5;
    }

}
```

注意，这里为了节省代码，我们并没有遵循一些面向对象的基本原则。使用示例如下：

```
public class Test {
    public static void main(String[] args) {
        VoltAdapter adapter = new VoltAdapter(new Volt220());
        System.out.println("输出电压 : " + adapter.getVolt5());
    }
}
```

这种实现方式直接将要被适配的对象传递到 Adapter 中，使用组合的形式实现接口兼容的效果。这比类适配器方式更为灵活，它的另一个好处是被适配对象中的方法不会暴露出来，而类适配器由于继承了被适配对象，因此，被适配对象类的函数在 Adapter 类中也都含有，这使得 Adapter 类出现一些奇怪的接口，用户使用成本较高。因此，对象适配器模式更加灵活、实用。

在实际开发中 Adapter 通常应用于进行不兼容的类型转换的场景，还有一种就是输入有无数种情况，但是输出类型是统一的，我们可以通过 Adapter 返回一个统一的输出，而具体的输入留给用户处理，内部只需知道输出的是符合要求的类型即可。例如 ListView 的 Adapter，用户的 Item View 各式各样，但最终都是属于 View 类型，ListView 只需要知道 getView 返回的是一个 View 即可，具体是什么 View 类型并不需要 ListView 关心。而在使用 Adapter 模式的过程中建议尽量使用对象适配器的实现方式，多用合成或者聚合，少用继承。当然，具体问题具体分析，根据需要来选用实现

方式，最适合的才是最好的。

20.6 Android 源码中的适配器模式

在开发过程中，ListView 的 Adapter 是我们最为常见的类型之一。我们需要使用 Adapter 加载每个 Item View 的布局，并且进行数据绑定等操作。代码大致如下。

```
// 代码省略
 ListView myListView = (ListView)findViewById(listview_id);
 // 设置适配器
 myListView.setAdapter(new MyAdapter(context, myDatas));

// 适配器
public class MyAdapter extends BaseAdapter{

        private LayoutInflater mInflater;
        List<String> mDatas ;

        public MyAdapter(Context context, List<String> datas){
            this.mInflater = LayoutInflater.from(context);
            mDatas = datas ;
        }
        // 代码省略,getCount、getItem 等

        // 解析、设置、缓存 convertView 以及相关内容
        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
            ViewHolder holder = null;
            // Item View 的复用
            if (convertView == null) {
                holder = new ViewHolder();
                convertView = mInflater.inflate(R.layout.my_listview_item, null);
                // 获取 title
                holder.title = (TextView)convertView.findViewById(R.id.title);
                convertView.setTag(holder);
            } else {
                holder = (ViewHolder)convertView.getTag();
            }
            holder.title.setText(mDatas.get(position));
            return convertView;
        }

    }
```

初学 Android 的时候我们只知道 Google 给出来的结果就是这么写的，久而久之我们发现每次都写这些样板代码非常麻烦，就会产生这样的疑问：ListView 为什么要使用 Adapter 模式呢？

我们知道 ListView 作为最重要的控件，它需要能够显示各式各样的视图（Item View），每个人需要的显示效果各不相同，显示的数据类型、数量等也千变万化，那么如何应对这种变化成为架构师需要考虑的最重要特性之一。

Android 的做法是增加一个 Adapter 层来隔离变化，将 ListView 需要的关于 Item View 接口抽象到 Adapter 对象中，并且在 ListView 内部调用 Adapter 这些接口完成布局等操作。这样只要用户实现了 Adapter 的接口，并且将该 Adapter 设置给 ListView，ListView 就可以按照用户设定的 UI 效

果、数量、数据来显示每一项数据。ListView 最重要的问题是要解决每一项 Item 视图的输出，Item View 千变万化，但终究它都是 View 类型，Adapter 统一将 Item View 输出为 View ，这样就很好地应对了 Item View 的可变性。这虽然有些脱离 Adapter 模式将不兼容的接口转换为可用接口的使用场景，但也是 Adapter 模式的一种变种实现。

那么 ListView 是如何通过 Adapter 模式来实现应对千变万化的 UI 效果呢 ？下面我们一起来一探究竟吧。

我们发现在 ListView 中并没有 Adapter 相关的成员变量，其实 Adapter 在 ListView 的父类 AbsListView 中，AbsListView 是一个列表控件的抽象，我们看一看这个类。

```
public abstract class AbsListView extends AdapterView<ListAdapter> implements
TextWatcher,
        ViewTreeObserver.OnGlobalLayoutListener, Filter.FilterListener,
        ViewTreeObserver.OnTouchModeChangeListener,
        RemoteViewsAdapter.RemoteAdapterConnectionCallback {

        ListAdapter mAdapter ;

    // 关联到 Window 时调用，获取调用 Adapter 中的 getCount 方法等
    @Override
    protected void onAttachedToWindow() {
        super.onAttachedToWindow();
        // 代码省略
        // 给适配器注册一个观察者,该模式第 12 章已介绍
        if (mAdapter != null && mDataSetObserver == null) {
            mDataSetObserver = new AdapterDataSetObserver();
            mAdapter.registerDataSetObserver(mDataSetObserver);
            mDataChanged = true;
            mOldItemCount = mItemCount
            // 获取 Item 的数量,调用的是 mAdapter 的 getCount 方法
            mItemCount = mAdapter.getCount();
        }
        mIsAttached = true;
    }

   /**
     * 子类需要覆写 layoutChildren()函数来布局 child view,也就是 Item View
     */
    @Override
    protected void onLayout(boolean changed, int l, int t, int r, int b) {
        super.onLayout(changed, l, t, r, b);
        mInLayout = true;
        if (changed) {
            int childCount = getChildCount();
            for (int i = 0; i < childCount; i++) {
                getChildAt(i).forceLayout();
            }
            mRecycler.markChildrenDirty();
        }

        if (mFastScroller != null && mItemCount != mOldItemCount) {
            mFastScroller.onItemCountChanged(mOldItemCount, mItemCount);
        }
        // 布局 Child View
        layoutChildren();
        mInLayout = false;
```

```
        mOverscrollMax = (b - t) / OVERSCROLL_LIMIT_DIVISOR;
    }
    }
```

AbsListView 定义了集合视图的逻辑框架，比如 Adapter 模式的应用、复用 Item View 的逻辑、布局子视图的逻辑等，子类只需要覆写特定的方法即可实现集合视图的功能。首先在 AbsListView 类型的 View 中添加窗口（onAttachedToWindow 函数）时会调用 Adapter 中的 getCount 函数获取到元素的个数，然后在 onLayout 函数中调用 layoutChilden 函数对所有子元素进行布局。AbsListView 中并没有实现 layoutChilden 这个函数，具体的实现在子类中，这里我们分析 ListView 中的实现，具体代码如下。

```
@Override
    protected void layoutChildren() {
        // 代码省略
        try {
            super.layoutChildren();
            invalidate();
            // 代码省略
            // 根据布局模式来布局 Item View
            switch (mLayoutMode) {
            // 代码省略
            case LAYOUT_FORCE_BOTTOM:
                sel = fillUp(mItemCount - 1, childrenBottom);
                adjustViewsUpOrDown();
                break;
            case LAYOUT_FORCE_TOP:
                mFirstPosition = 0;
                sel = fillFromTop(childrenTop);
                adjustViewsUpOrDown();
                break;
    // 代码省略
            default:
                // 代码省略
                break;
            }

    }
```

ListView 覆写了 AbsListView 中的 layoutChilden 函数，在该函数中根据布局模式来布局 Item View，例如，默认情况是从上到下开始布局，但是，也有从下到上开始布局的，例如 QQ 聊天窗口的气泡布局，最新的消息就会布局到窗口的最底部。我们看看这两种实现。

```
// 从上到下填充 Item View  [ 只是其中一种填充方式 ]
    private View fillDown(int pos, int nextTop) {
        View selectedView = null;
        int end = (mBottom - mTop);
        if ((mGroupFlags & CLIP_TO_PADDING_MASK) == CLIP_TO_PADDING_MASK) {
            end -= mListPadding.bottom;
        }

        while (nextTop < end && pos < mItemCount) {
            boolean selected = pos == mSelectedPosition;
            // 通过 makeAndAddView 获取 Item View
            View child = makeAndAddView(pos, nextTop, true, mListPadding.left, selected);

            nextTop = child.getBottom() + mDividerHeight;
```

```
            if (selected) {
                selectedView = child;
            }
            pos++;
        }
        return selectedView;
    }
    // 从下到上布局
    private View fillUp(int pos, int nextBottom) {
        View selectedView = null;
        int end = 0;
    // 代码省略

        while (nextBottom > end && pos >= 0) {
            boolean selected = pos == mSelectedPosition;
            // 通过 makeAndAddView 获取 Item View
            View child=makeAndAddView(pos,nextBottom,false, mListPadding.left, selected);
            nextBottom = child.getTop() - mDividerHeight;
            if (selected) {
                selectedView = child;
            }
            pos--;
        }

        mFirstPosition = pos + 1;
        setVisibleRangeHint(mFirstPosition, mFirstPosition + getChildCount() - 1);
        return selectedView;
    }
```

在每一种布局方式的函数中都会从 makeAndAddView 函数获取一个 View，这个 View 就是 ListView 的每一项的视图，这里有一个 pos 参数，也就是对应这个 View 是 ListView 中的第几项。我们看看 makeAndAddView 中的实现。

```
// 添加 Item View
private View makeAndAddView(int position, int y, boolean flow, int childrenLeft,
        boolean selected) {
        View child;
    // 代码省略
    //  获取一个 item View
    child = obtainView(position, mIsScrap);
    // 将 item view 设置到对应的地方
    setupChild(child, position, y, flow, childrenLeft, selected, mIsScrap[0]);
    return child;
}
```

在 makeAndAddView 函数中主要分为两个步骤，第一是根据 position 获取一个 item View，然后将这个 View 布局到特定的位置。获取一个 item View 调用的是 obatinView 函数，这个函数在 AbsListView 中。核心代码如下。

```
// 获取一个 Item View，该函数在 AbsListView 中
    View obtainView(int position, boolean[] isScrap) {
        isScrap[0] = false;
        View scrapView;
        // 1. 从缓存的 Item View 中获取，ListView 的复用机制就在这里
        scrapView = mRecycler.getScrapView(position);
        View child;
        if (scrapView != null) {
            // 代码省略
```

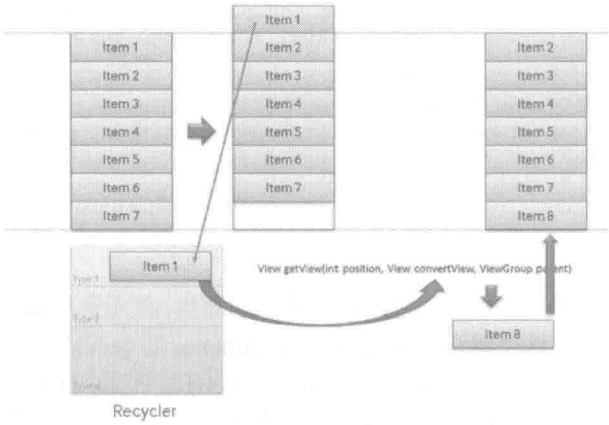
```
            // 2. 注意，这里将 scrapView 设置给了 Adapter 的 getView 函数
            child = mAdapter.getView(position, scrapView, this);
            // 代码省略
        } else {
            // 3. 没有缓存 View 的情况下 getView 的第二个参数为 null
            child = mAdapter.getView(position, null, this);
            // 代码省略
        }

        return child;
    }
```

obatinView 函数定义了列表控件的 Item View 的复用逻辑，首先会从 RecyclerBin 中获取一个缓存的 View，如果有缓存则将这个缓存的 View 传递到 Adapter 的 getView 第二个参数中，这也就是我们对 Adapter 的最常见的优化方式，即判断 getView 的 convertView 是否为空，如果为空则从 xml 中创建视图，否则使用缓存的 View。这样避免了每次都从 xml 加载布局的消耗，能够显著提升 ListView 等列表控件的效率。通常的实现如下。

```
// 解析、设置、缓存 convertView 以及相关内容
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        ViewHolder holder = null;
        // convertView 为空，表示没有缓存，此时需要从 xml 加载
        if (convertView == null) {
            holder = new ViewHolder();
            convertView = mInflater.inflate(R.layout.my_listview_item, null);
            // 获取 title
            holder.title = (TextView)convertView.findViewById(R.id.title);
            convertView.setTag(holder);
                    } else {
            // convertView 不为空，这里的 convertView 就是从 RecyclerBin
            // 取出的缓存视图
            holder = (ViewHolder)convertView.getTag();
        }
        // 数据绑定
        return convertView;
    }
```

通过这种缓存机制，即使有成千上万的数据项，ListView 也能够流畅运行，因此，只有填满一屏所需的 Item View 存在内存中。ListView 根据 Adapter 设置的数据项数量循环调用 getView 方法获取视图，第一次加载填满屏幕的数据项时 getView 的第二个参数 convertView 都为空，此时每次都需要从 xml 中加载布局文件，填充数据之后返回给 ListView。当整屏的数据项加载完毕之后用户向下滚动屏幕，此时 item1 滚出屏幕，并且一个新的项目从屏幕低端上来时，ListView 再请求一个视图，此时 item1 被缓存起来，在下一项数据加载时传递给 getView 的第二个参数 convertView，因此，convertView 此时不是空值，它的值是 item1。此时只需设定新的数据然后返回 convertView，这样就避免了每次都从 xml 加载、初始化视图，减少了时间、性能上的消耗。原理如图 20-3 所示。

了解了它的工作原理后，就可以重复利用 ListView 的 Item View，只要 convertView 不为空就直接使用，改变它绑定的数据就行了。当然，由于视图被缓存了，视图中的数据也会被缓存，因此，你需要在每次获取到了 Item view 时对每个数据项重新赋值，否则会出现数据错误的现象。



▲图 20-3

我们最后总结一下这个过程。

ListView 等集合控件通过 Adapter 来获取 Item View 的数量、布局、数据等，在这里最为重要的就是 getView 函数，这个函数返回一个 View 类型的对象，也就是 Item View。由于它返回的是一个 View 抽象，而千变万化的 UI 视图都是 View 的子类，通过依赖抽象这个简单的原则和 Adaper 模式就将 Item View 的变化隔离了，保证了 AbsListView 类族的高度可定制化。在获取了 View 之后，将这些 View 通过特定的布局方式设置到对应的位置上，再加上 Item View 的复用机制，整个 ListView 就运转起来了。

当然，这里的 Adapter 并不是经典的适配器模式，却是对象适配器模式的优秀示例，也很好地体现了面向对象的一些基本原则。这里的 Target 角色就是 View，Adapter 角色就是将 Item View 输出为 View 抽象的角色，Adaptee 就是需要被处理的 Item View。通过增加 Adapter 一层来将 Item View 的操作抽象起来，ListView 等集合视图通过 Adapter 对象获得 Item 的个数、数据、Item View 等，从而达到适配各种数据、各种 Item 视图的效果。因为 Item View 和数据类型千变万化，Android 的架构师们将这些变化的部分交给用户来处理，通过 getCount、getItem、getView 等几个方法抽象出来，也就是将 Item View 的构造过程交给用户来处理，灵活地运用了适配器模式，达到了无限适配、拥抱变化的目的。

20.7 深度拓展

虽然 ListView 已经足够强大了，但是还存在一些问题，例如，每次都需要自己创建一个 ViewHolder、手动判断是否有缓存 View 等。对于用户而言，这些都还可以改进，因此 Google 推出了一个更强大的控件-RecyclerView。顾名思义，这个 View 代表的就是一个可循环使用的视图集合控件，它定义了 ViewHolder 类型标准，封装了 View 缓存判断逻辑，更强大的是它可以通过一个 LayoutManager 将一个 RecyclerView 显示为不同的样式，例如 ListView、GridView 形式、瀑布流形式。光是这些就足以促使我们好好学习这个控件，它将是未来几年在 Android 开发中取代 ListView 作为最重要的控件。

开发技术前线网站的 Android 客户端就是使用了 RecyclerView 作为显示文章列表的控件，如图 20-4 所示。

▲图 20-4

这个界面很简单，每个文章列表就是几个普通的 TextView 组合而已。我们看看在主界面的布局 xml 中定义 RecyclerView 的代码如下。

```
<android.support.v7.widget.RecyclerView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/recyclerview"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

然后在 Activity 中进行初始化，除了类似 ListView 的 Adapter 之外，RecyclerView 还有一个额外的设置就是设置布局方式，这也是 RecyclerView 能够在布局上比 ListView 更为灵活的原因。因为它将布局通过桥接、组合的形式交由 LayoutManager 实现，而不是像 ListView、GridView 这些类自己实现，因为这会导致一个类只能有一种布局方式，当用户需要实现其他布局效果时就需要替换整个实现类。而通过桥接模式和组合模式，将布局这个职责分离开来，使得一个 RecyclerView 与不同的 LayoutManager 搭配就能做成各种布局效果，这也是桥接模式的经典应用。

我们看看图 20-4 的文章列表的实现代码，看看 Activity 中设置 RecyclerView 的代码。

```
// 1. 获取控件
RecyclerView mRecyclerView = (RecyclerView) rootView.findViewById(R.id.recyclerview);
// 2. 设置布局方式为线性布局
mRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));
mRecyclerView.setHasFixedSize(true);
List<Article> articles = new ArrayList<Article>();
// 3. 设置 Adapter
ArticleAdapter mAdapter = new ArticleAdapter(articles);
mRecyclerView.setAdapter(mAdapter);
```

上述代码较为简单，就是一个普通的初始化过程，唯一与 ListView 不太相同的是第二步中设置了 LayoutManager，这个 LayoutManager 的职责就是负责 RecylerView 元素的布局。默认的实现有 LinearLayoutManager、GridLayoutManager、StaggeredGridLayoutManager，分别是线性布局、网格布局和瀑布流布局。这些布局不仅可以竖向布局，还支持横向布局，这几个布局已经能够基本满足我们的需求了，可见 RecylerView 的高度定制化能力。

下面看看文章 Adapter 的实现。

```
public class ArticleAdapter extends Adapter<ViewHolder> {
    List<Article> mArticles; // 文章数据
    OnItemClickListener<Article> mClickListener; // 每一项的点击事件

    public ArticleAdapter(List<Article> dataSet) {
        mArticles = dataSet;
    }

    // 绑定每一项的数据
    @Override
    public void onBindViewHolder(ViewHolder viewHolder, int position) {
```

```
        if (viewHolder instanceof ArticleViewHolder) {
            bindViewForArticle(viewHolder, position);
        }
    }

    protected void bindViewForArticle(ViewHolder viewHolder, int position) {
        ArticleViewHolder articleViewHolder = (ArticleViewHolder) viewHolder;
        final Article article = getItem(position);
        articleViewHolder.titleTv.setText(article.title);
        articleViewHolder.publishTimeTv.setText(article.publishTime);
        articleViewHolder.authorTv.setText(article.author);
        // 设置点击事件
        articleViewHolder.itemView.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                if (mClickListener != null) {
                    mClickListener.onClick(article);
                }
            }
        });
    }

    @Override
    public int getItemCount() {
        return mArticles.size();
    }
    // 创建 ViewHolder
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
        return createArticleViewHolder(viewGroup);
    }

    protected ViewHolder createArticleViewHolder(ViewGroup viewGroup) {
        View itemView = LayoutInflater.from(viewGroup.getContext()).inflate(
                R.layout.recyclerview_article_item, viewGroup, false);
        returnnew ArticleViewHolder(itemView);
    }

    protected Article getItem(int position) {
        return mArticles.get(position);
    }

    public void setOnItemClickListener(OnItemClickListener<Article> mClickListener) {
        this.mClickListener = mClickListener;
    }
    // ViewHolder, 负责保持 Item View
    static class ArticleViewHolder extends RecyclerView.ViewHolder {
        public TextView titleTv;
        public TextView publishTimeTv;
        public TextView authorTv;

        public ArticleViewHolder(View itemView) {
            super(itemView);

            titleTv = (TextView) itemView.findViewById(R.id.article_title_tv);
            publishTimeTv = (TextView) itemView.findViewById(R.id.article_time_tv);
            authorTv = (TextView) itemView.findViewById(R.id.article_author_tv);
        }
    }
}
```

在RecyclerView中没有了getView函数，取而代之的是onBindViewHolder和onCreateViewHolder，这两个函数就相当于 getView 功能。首先会从 onCreateViewHolder 中获取 ViewHolder 对象，ViewHolder 中又含有 Item View 的引用，获取到 ViewHolder 之后又会通过 onBindViewHolder 函数来绑定数据，这两个过程实际上是将 getView 函数分解开来，使得加载视图和绑定数据分离开来。在 RecyclerView 中的操作单位也不再是 View，而是 ViewHolder。Android 中定义了一个 Adapter 基类，该类的一个泛型参数就是 ViewHolder，用户需要继承 ViewHolder 实现自己的 ViewHolder，对于 item view 的操作都从这个 ViewHolder 对象进行。通过 onBindViewHolder 和 onCreateViewHolder 两个函数，Android 屏蔽了对 Item View 缓存的手动判断，将这部分逻辑封装在 RecyclerView 中，这样就不需要像 ListView 中的 Adapter 的 getView 函数那样对 convertView 进行判空等操作，简化的逻辑，隐藏了具体实现。

最后将数据设置给 Adapter，再将 Adapter 设置给 RecyclerView，RecyclerView 再根据用户设置的 LayoutManager 将每个数据项布局到对应的位置，这样一来整个列表控件就显示在我们眼前了。Android 通过 onBindViewHolder 和 onCreateViewHolder 两个函数屏蔽了一些固定的逻辑，使得 Adapter 的使用更为简单，又通过 LayoutManager 实现具体的布局，使得 RecyclerView 具有更强大的定制能力，这就是 RecyclerView 比 ListView 更强大的原因所在。

如果我们看腻了如图 20-4 所示的单向列表效果，想要换成每行 3 篇文章的网格形式的布局，此时只需要将 LieanrLayoutManager 替换为一个 GridLayouManager 即可。具体代码如下。

```
    // 1. 获取控件，代码省略
    // 2. 设置布局方式为网格布局
mRecyclerView.setLayoutManager(new GridLayoutManager(getActivity(), 3));
    mRecyclerView.setHasFixedSize(true);
    List<Article> articles = new ArrayList<Article>();
    // 3. 设置 Adapter
```

效果如图 20-5 所示。

▲图 20-5

又如果你觉得高度一样的网格布局还是不好看，你想要高度不规则的瀑布流布局，那么此时需要做的同样也很简单。修改 LayoutManager 为 StaggeredGridLayoutManager，然后在 Adapter 中为每个 Item 设置一个高度即可。具体代码如下。

```
// 1. 获取控件，代码省略
// 2. 设置布局方式为瀑布流布局
mRecyclerView.setLayoutManager(new StaggeredGridLayoutManager(3,
        StaggeredGridLayoutManager.VERTICAL));
mRecyclerView.setHasFixedSize(true);
List<Article> articles = new ArrayList<Article>();
// 3. 设置 Adapter
```

还需要做的是在 Adapter 中设置每个 Item 的高度，例如。

```
@Override
public void onBindViewHolder(ViewHolder viewHolder, int position) {
    // 代码省略

    ArticleViewHolder articleViewHolder = (ArticleViewHolder) viewHolder;
    LayoutParams params =  articleViewHolder.itemView.getLayoutParams() ;
    // 随机生成一个高度
    params.height = new Random().nextInt(10) * 50 + 100 ;
    // 设置属性
    articleViewHolder.itemView.setLayoutParams(params);
}
```

此时的效果图如图 20-6 所示。

▲图 20-6

如图 20-6 所示，通过替换 LayoutManager 和添加几行代码瀑布流效果就呈现在我们面前了！RecyclerView 的功能真是非常强大，它的设计也是我们值得好好学习的示例。

不过，RecyclerView 也有几个小问题需要注意，第一个是它没有设置处理 Item 点击事件的 Listener 接口，例如 ListView 可以通过 setOnItemClickListener，但是，在 RecyclerView 中并没有这

个接口。如果要给 Item 设置点击事件，那么需要通过 Adapter 实现，通常的做法是定义一个点击接口，然后设置给 Adapter，最后在这个 View 的点击事件中触发这个类似于 OnItemClickListener 的回调函数。示例程序如下。

```
public class ArticleAdapter extends Adapter<ViewHolder> {
    private OnItemClickListener<Article> ClickListener ;

     @Override
    public void onBindViewHolder(ViewHolder viewHolder, int position) {
        // 代码省略
         ArticleViewHolder articleViewHolder = (ArticleViewHolder) viewHolder;
        // 设置 itemView 的点击事件，并且在里面调用 OnItemClickListener 的 onClick 方法
        articleViewHolder.itemView.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                if (mClickListener != null) {
                    mClickListener.onClick(article);
                }
            }
        });
    }

    public void setOnItemClickListener(OnItemClickListener<Article> listener) {
        mClickListener = listener ;
    }

    // 定义点击接口
    public static interface OnItemClickListener<T> {
        public void onClick(T item);
    }
}
```

第二个问题是，RecyclerView 没有可以设置 Header 和 footer 的接口，当需要实现 Header View 时，需要根据 viewType 进行分类处理。例如当 position 为 0 时，你把 viewType 设置为 0，其他位置的 Item 的 viewType 则为 1，也就是说第一个 item 是 Header，其他的是普通的 Item。因此，需要在 position 为 0 时返回 HeaderView 的 ViewHolder，而在其他位置时返回普通 item 的 ViewHolder。并且需要注意的是，item count 需要比数据集的数量多 1，这个 1 就是这个 Header View，另外，从数据集取数据时则需要把 position 减 1。因为第一个 View 为 Header，Header 通常情况下并不使用数据集中的数据，也就是说第一个 item 不需要数据，需要数据的是第二个 Item View，而此时它显示的数据应该是数据集中的第一个，因此取数据时需要把索引减 1。示例代码如下。

```
public class ArticleWithHeaderAdapter extends Adapter<ViewHolder> {
    private static final int HEADER_TYPE = 0;
    // 字段省略

    // 分类进行设置
    @Override
    public void onBindViewHolder(ViewHolder viewHolder, int position) {
        if (viewHolder instanceof ArticleViewHolder) {
            // 设置普通的 Item View
        } else if (viewHolder instanceof HeaderViewHolder) {
            // 设置 Header View
        }
    }
```

```
    // 根据分类创建对应的 ViewHolder
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
        if (viewType == HEADER_TYPE) {
            return createHeaderViewHolder(viewGroup);
        }
        return createArticleViewHolder(viewGroup);
    }

    private HeaderViewHolder createHeaderViewHolder(ViewGroup viewGroup) {
        View headerView = LayoutInflater.from(
                viewGroup.getContext()).inflate(R.layout.auto_slider, viewGroup, false);
        return new HeaderViewHolder(headerView);
    }


    /*
     * 多了一个 Header，因此数量要加 1
     */
    @Override
    public int getItemCount() {
        return mArticles == null ? 1 : mArticles.size() + 1;
    }

    @Override
    protected Article getItem(int position) {
        // 因为第一项视图是 Header，添加了一个 Header，因此数据索引要减去 1
        return mArticles.get(position - 1);
    }

    // 设置 viewType
    @Override
    public int getItemViewType(int position) {
        if (HEADER_TYPE == position) {
            return 0;
        }
        return 1;
    }

    static class ArticleViewHolder extends RecyclerView.ViewHolder {
        // 字段省略

        public HeaderViewHolder(View view) {
            super(view);
            // 代码省略
        }
    }

    static class HeaderViewHolder extends RecyclerView.ViewHolder {
        View headerView ;

        public HeaderViewHolder(View view) {
            super(view);
            // 代码省略
        }
    }
}
```

上述代码中有一个 getViewType 函数，在该函数中把第一个 item 的 viewType 设置为 0，其他位置的设置为 1。这样在 onCreateViewHolder 函数中就可以根据不同的 viewType（onCreateViewHolder

的第二个参数，不是 position，而是 viewType）来创建不同的 ViewHolder，不同的 ViewHolder 含有不同的 View，因此达到了不同显示效果的目的。上文中的开发技术前线网站的第一项就是一个 Header View，这个 Header View 是一个自动滚动的 ViewPager，其他的都是普通的文本控件的组合视图。

RecyclerView 很强大，也有些不足，例如上述提到的两点。但是，对于 Header View 的设置来说，似乎又没有很好的解决方案。如果通过像 ListView 那样的形式添加 header View 也经常会出现问题，例如添加 header view 在 setAdapter 之后，导致 header 不显示的问题。而 RecyclerView 的这种做法显然问题更为明显，当然它的定制性也更高，至于 Android 工程师这样设计可能有另外的考虑。总的来说，还是开头的那句话，RecyclerView 是个强大的、值得学习的控件！

RecyclerView 设计与实现

在上一节中我们学习了 RecyclerView 的一些基本使用，本章我们来学习一下 RecyclerView 的设计与实现，以及它与 ListView 等组件有什么比较明显的区别。

对于 RecyclerView 和 ListView 来说，比较相同的一点是使用了 Adapter 和观察者模式，相关的代码如下。

```
public class RecyclerView extends ViewGroup {

    public void setAdapter(Adapter adapter) {
        setAdapterInternal(adapter, false, true);
        requestLayout();
    }

    private void setAdapterInternal(Adapter adapter, boolean compatibleWithPrevious,
    boolean removeAndRecycleViews) {
        // 代码省略
        mAdapterHelper.reset();
        final Adapter oldAdapter = mAdapter;
        mAdapter = adapter;
        if (adapter != null) {
        // 注册观察者
            adapter.registerAdapterDataObserver(mObserver);
        }
        // 代码省略

        // 设置结构发生改变的标志位
        mState.mStructureChanged = true;
        // 刷新视图
        markKnownViewsInvalid();
}
}
```

在用 setAdapter 时最终也会注册一个观察者，这个观察者具体实现类是 RecyclerView 的内部类 RecyclerViewDataObserver，具体代码如下。

```
// 观察者实现类
private class RecyclerViewDataObserver extends AdapterDataObserver {
    @Override
    public void onChanged() {
        assertNotInLayoutOrScroll(null);
        // 代码省略
```

```
        // 需要重新布局
        if (!mAdapterHelper.hasPendingUpdates()) {
            requestLayout();
        }
    }
    // 代码省略
}
```

在数据集发生变化且调用了 Adapter 的 notifyDataSetChanged 之后就会调用 RecyclerViewDataObserver 的 onChanged 函数，在该函数中又会调用 RecyclerView 的 requestLayout 函数进行重新布局。这些过程与 ListView 的实现基本一致，最大的不同在于它们之间的布局实现上。在 ListView 中的布局是通过自身的 layoutChilden 函数来实现，而对于 RecyclerView 来说它的布局职责则是交给了 LayoutManager 对象。

```
public class RecyclerView extends ViewGroup {
    private Adapter mAdapter;
    private LayoutManager mLayout;
    // 代码省略

    public void setLayoutManager(LayoutManager layout) {
        // 代码省略
        mLayout = layout;
        if (layout != null) {
            mLayout.setRecyclerView(this);
            if (mIsAttached) {
                mLayout.onAttachedToWindow(this);
            }
        }
        // 设置布局管理器之后重新布局
        requestLayout();
    }
}
```

在设置了布局管理器之后就会调用 requestLayout 函数进行布局，然后会调用 onLayout 函数，我们看看相关的实现。

```
@Override
protected void onLayout(boolean changed, int l, int t, int r, int b) {
    eatRequestLayout();
    // 分发 layout
    dispatchLayout();
    resumeRequestLayout(false);
    mFirstLayoutComplete = true;
}

void dispatchLayout() {
    // 代码省略
    // 获取 Item 数量
    mState.mItemCount = mAdapter.getItemCount();
    mState.mDeletedInvisibleItemCountSincePreviousLayout = 0;

    mState.mInPreLayout = false;
    // 执行布局，调用 LayoutManager 的 onLayoutChilden 函数
    mLayout.onLayoutChildren(mRecycler, mState);

    mState.mStructureChanged = false;
    // 含有动画则执行 item 动画
    if (mState.mRunSimpleAnimations) {
```

```
        // 代码省略
    }
    // 代码省略
}
```

在 onLayout 函数中最终会调用 dispatchLayout 函数，而在 dispatchLayout 函数中又会调用 LayoutManager 的 onLayoutChilden 函数进行布局。在此，我们以 LinearLayoutManager 为例进行学习。下面看看 LinearLayoutManager 中的 onLayoutChilden 函数。

```
@Override
    public void onLayoutChildren(RecyclerView.Recycler recycler, RecyclerView.State state) {
    // 代码省略

        int startOffset;
        int endOffset;
        onAnchorReady(state, mAnchorInfo);
        detachAndScrapAttachedViews(recycler);
        mLayoutState.mIsPreLayout = state.isPreLayout();
        if (mAnchorInfo.mLayoutFromEnd) {
        // 代码省略
        } else {
            // 从上到下布局
            updateLayoutStateToFillEnd(mAnchorInfo);
            mLayoutState.mExtra = extraForEnd;
            fill(recycler, mLayoutState, state, false);
            endOffset = mLayoutState.mOffset;
            if (mLayoutState.mAvailable > 0) {
                extraForStart += mLayoutState.mAvailable;
            }
            // fill towards start
            updateLayoutStateToFillStart(mAnchorInfo);
            mLayoutState.mExtra = extraForStart;
            mLayoutState.mCurrentPosition += mLayoutState.mItemDirection;
            // 填充 Item View
            fill(recycler, mLayoutState, state, false);
            startOffset = mLayoutState.mOffset;
        }
        // 代码省略
    }

    int fill(RecyclerView.Recycler recycler, LayoutState layoutState,
            RecyclerView.State state, boolean stopOnFocusable) {
        // 存储当前可用空间
        final int start = layoutState.mAvailable;
        // 1. 计算 RecyclerView 的可用布局宽或高
        int remainingSpace = layoutState.mAvailable + layoutState.mExtra;
        LayoutChunkResult layoutChunkResult = new LayoutChunkResult();
        // 2. 迭代布局 item view
        while (remainingSpace > 0 && layoutState.hasMore(state)) {
            layoutChunkResult.resetInternal();
            // 3. 布局 item View
            layoutChunk(recycler, state, layoutState, layoutChunkResult);
            if (layoutChunkResult.mFinished) {
                break;
            }
            // 4. 计算布局偏移量
            layoutState.mOffset+=layoutChunkResult.mConsumed*layoutState. mLayoutDirection;

            if (!layoutChunkResult.mIgnoreConsumed || mLayoutState.mScrapList != null
                || !state.isPreLayout()) {
```

```
                layoutState.mAvailable -= layoutChunkResult.mConsumed;
                // 5. 计算剩余的可用空间
                remainingSpace -= layoutChunkResult.mConsumed;
            }
        // 代码省略
    }

    return start - layoutState.mAvailable;
}
```

在 onLayoutChilden 函数中会调用 fill 函数，在 fill 函数中又会循环地调用 layoutChunk 函数进行布局，每次布局完之后就会计算当前屏幕剩余的可利用空间，并且做出判断是否还需要布局 Item View。因此，先看看 layoutChunk 的实现。

```
void layoutChunk(RecyclerView.Recycler recycler, RecyclerView.State state,
        LayoutState layoutState, LayoutChunkResult result) {
    // 1. 获取 Item View
    View view = layoutState.next(recycler);
    // 2. 获取 Item View 的布局参数
    RecyclerView.LayoutParams params=(RecyclerView.LayoutParams)view.getLayoutParams();
    // 3. 丈量 Item View
    measureChildWithMargins(view, 0, 0);
    // 4. 计算该 Item View 消耗的宽度或高度
    result.mConsumed = mOrientationHelper.getDecoratedMeasurement(view);
    // item view 的上下左右坐标位置
    int left, top, right, bottom;
    // 5. 按照水平或竖直方向布局，计算 Item View 的上下左右坐标
    if (mOrientation == VERTICAL) {
        if (isLayoutRTL()) {
            right = getWidth() - getPaddingRight();
            left = right - mOrientationHelper.getDecoratedMeasurementInOther(view);
        } else {
            left = getPaddingLeft();
            right = left + mOrientationHelper.getDecoratedMeasurementInOther(view);
        }
        if (layoutState.mLayoutDirection == LayoutState.LAYOUT_START) {
            bottom = layoutState.mOffset;
            top = layoutState.mOffset - result.mConsumed;
        } else {
            top = layoutState.mOffset;
            bottom = layoutState.mOffset + result.mConsumed;
        }
    } else {
        // 竖直方向布局的计算方式
    }
    // 6. 布局 item view
    layoutDecorated(view, left + params.leftMargin, top + params.topMargin,
            right - params.rightMargin, bottom - params.bottomMargin);
}
```

在 layoutChunk 中首先从 layoutState 中获取到 Item View，然后获取 Item View 的布局参数、尺寸信息，并且根据布局方式（横向或者纵向）计算出 Item View

的上下左右坐标，最后调用 layoutDecorated 函数实现布局。layoutDecorated 函数定义在 LayoutManager 中，具体代码如下。

```
// LayoutManager
public void layoutDecorated(View child, int left, int top, int right, int bottom) {
        final Rect insets = ((LayoutParams) child.getLayoutParams()).mDecorInsets;
```

```
    child.layout(left + insets.left, top + insets.top, right - insets.right, bottom -
insets.bottom);
}
```

从上述程序可以看到，只是调用了 Item View 的 layout 函数将 Item View 布局到具体的位置。这样一来，就将布局的职责从 RecyclerView 分离到 LayoutManager 中，也使得 RecyclerView 更为灵活。

这里的 layoutChunk 函数很重要，我们看注释 1 处，首先通过 LayoutState 对象的 next 函数获取到 Item View，这里也是一个重要的地方。我们看看 LayoutState 函数的 next 实现。

```
View next(RecyclerView.Recycler recycler) {
    // 调用 Recycler 中的 getViewForPosition 获取 item View
    final View view = recycler.getViewForPosition(mCurrentPosition);
    mCurrentPosition += mItemDirection;
    return view;
}
```

实际上就是调用 RecyclerView.Recycler 对象 getViewForPosition 函数获取到 Item View，我们继续深入 RecyclerView.Recycler 类的相关代码。

```
public final class Recycler {
        final ArrayList<ViewHolder> mAttachedScrap = new ArrayList<ViewHolder>();
        private ArrayList<ViewHolder> mChangedScrap = null;
        final ArrayList<ViewHolder> mCachedViews = new ArrayList<ViewHolder>();

        // 根据 position 获取该位置对应的 View
        View getViewForPosition(int position, boolean dryRun) {
            boolean fromScrap = false;
            ViewHolder holder = null;
            // 1. 从 mChangedScrap 中获取 ViewHolder 缓存
            if (mState.isPreLayout()) {
                holder = getChangedScrapViewForPosition(position);
                fromScrap = holder != null;
            }
            // 2. 从 mAttachedScrap 中获取 ViewHolder 缓存
            if (holder == null) {
                holder = getScrapViewForPosition(position, INVALID_TYPE, dryRun);
                // 代码省略
            }
            if (holder == null) {
                final int offsetPosition = mAdapterHelper.findPositionOffset(position);
                // 从其他 ViewHolder 缓存中检测是否有缓存,代码省略

                // 3. 没有 ViewHolder，则需要创建 ViewHolder，这里会调用 onCreateViewHolder 函数
                if (holder == null) {
                    holder = mAdapter.createViewHolder(RecyclerView.this,
                    mAdapter.getItemViewType(offsetPosition));
                }
            }
            boolean bound = false;
            if (mState.isPreLayout() && holder.isBound()) {
                holder.mPreLayoutPosition = position;
            } else if (!holder.isBound() || holder.needsUpdate() || holder.isInvalid()) {
                final int offsetPosition = mAdapterHelper.findPositionOffset(position);
                // 4. 绑定数据，这里会调用 Adapter 的 onBindViewHolder
                mAdapter.bindViewHolder(holder, offsetPosition);
        // 代码省略
            }
```

```
            // 设置 Item View 的 LayoutParams
            // 5. 返回 itemView
            return holder.itemView;
        }
    }
```

我们知道在 RecyclerView 的 Adapter 中被缓存的单位已经不是 Item View 了，而是一个 ViewHolder，而原来的 ListView 则是缓存的 View。在 RecyclerView.Recycler 类中有 mAttachedScrap、mChangedScrap、mCachedViews 几个 ViewHolder 列表对象，它们就是用于缓存 ViewHolder 的。在通过 LayoutState 的 next 函数获取 Item View 时，实际上调用的是 RecyclerView.Recycler 的 getViewForPosition 函数，该函数首先会从这几个 ViewHolder 缓存中获取对应位置的 ViewHolder，如果没有缓存则调用 RecyclerView.Adapter 中的 createViewHolder 函数来创建一个新的 ViewHolder，我们看看 RecyclerView.Adapter 的 createViewHolder 函数。

```
public static abstract class Adapter<VH extends ViewHolder> {
    private final AdapterDataObservable mObservable = new AdapterDataObservable();
    private boolean mHasStableIds = false;

    public final VH createViewHolder(ViewGroup parent, int viewType) {
// 调用 onCreateViewHolder 创建 ViewHolder，用户需要覆写 onCreateViewHolder 函数
        final VH holder = onCreateViewHolder(parent, viewType);
        holder.mItemViewType = viewType;
        return holder;
    }
    // 创建 ViewHolder，子类需覆写
    public abstract VH onCreateViewHolder(ViewGroup parent, int viewType);
}
```

在 createViewHolder 函数中实际上调用了 onCreateViewHolder 函数创建 ViewHolder 对象。这也就是为什么在继承 RecyclerView.Adapter 时需要覆写 onCreateViewHolder 函数，并且在该函数中返回 ViewHolder 的原因。通过这个 onCreateViewHolder 函数会加载 Item View 视图，并且把 Item View 当作 ViewHolder 的构造参数传递给 ViewHolder，此时 ViewHolder 就构建完毕了。

调用了 Adapter 的 createViewHolder 后，此时执行到 Recycler 的 getViewForPosition 函数的注释 4 处，也就是调用了 Adapter 中的 bindViewHolder 函数，相关代码如下。

```
public final void bindViewHolder(VH holder, int position) {
            holder.mPosition = position;
            if (hasStableIds()) {
                holder.mItemId = getItemId(position);
            }
            // 调用 onBindViewHolder 绑定数据
            onBindViewHolder(holder, position);
            // 设置 holder 的 flags
            holder.setFlags(ViewHolder.FLAG_BOUND,
                    ViewHolder.FLAG_BOUND | ViewHolder.FLAG_UPDATE | ViewHolder.FLAG_INVALID);
        }

        // 绑定数据，子类需覆写
        public abstract void onBindViewHolder(VH holder, int position);
```

bindViewHolder 函数与 createViewHolder 如出一辙，只是它调用的是 onBindViewHolder 函数而已。与 onCreateViewHolder 一样，onBindViewHolder 也需要子类覆写，并且在这个函数中进行数据

绑定。在 getViewForPosition 中实际上相当于一个模板方法，它封装了获取、绑定 ViewHolder 的过程，子类只需要覆写特定的函数即可完成这个过程。

执行完 onBindViewHolder 函数之后数据就被绑定到 Item View 上了。

我们最后再来分析一下这个过程。

与 ListView 一样，RecyclerView 还是通过 Adapter 和观察者模式进行数据绑定，使得 RecyclerView 的灵活性得到保证。RecyclerView 的 Adapter 并不是 ListView 中的 Adapter，它是一个封装了 ViewHolder 的创建与绑定等逻辑，降低了用户的使用成本。RecyclerView 也将缓存单元从 Item View 换成了 ViewHoler，在一定程度上建立起了规范。RecyclerView 与 ListView 最大的不同是 RecyclerView 将布局的工作交给了 LayoutManager，在 LayoutManager 的 onLayoutChilden 中对 Item View 进行布局、执行动画等操作，这样一来，使得 RecyclerView 可以动态地替换掉布局方式，例如，在运行时给 RecyclerView 重新设置一个 LayoutManager 就可以将原来是线性布局的视图改成网格布局，这大大地增加了灵活性。将布局职责独立出来也符合单一职责原则，而使用组合代替继承也会减少耦合、增强健壮性，也使得 RecyclerView 的布局具有更好的扩展性。

20.8 实战演示

对于很多开发人员来说，炫酷的 UI 效果是最吸引他们注意力的，很多人也因为这些炫酷的效果而去学习一些比较知名的 UI 库。而做出炫酷效果的前提是必须对自定义 View 有所理解，小民自然也不例外。特别对于刚处在开发初期的小民，对于自定义 View 这件事觉得又神秘又酷，于是小民决定深入研究自定义 View 以及相关的知识点。

小民的同事洋叔是一位资深的研发人员，擅长写 UI 特效，在开发领域知名度颇高。最近洋叔刚发布了一个效果不错的圆形菜单，这个菜单的每个 Item 环形排布，并且可以转动。小民决定仿照洋叔的效果实现一遍，但是，对于小民这个阶段来说只要实现环形布局就不错了，转动部分作为下个版本功能，就当作自定义 View 的练习了。

在了解了自定义 View 相关的知识点之后，小民就写好了这个圆形菜单布局视图，我们一步一步来讲解，具体代码如下。

```
// 圆形菜单
public class CircleMenuLayout extends ViewGroup {
    // 圆形直径
    private int mRadius;
    // 该容器内 child item 的默认尺寸
    private static final float RADIO_DEFAULT_CHILD_DIMENSION = 1 / 4f;
    // 该容器的内边距,无视 padding 属性，如需边距请用该变量
    private static final float RADIO_PADDING_LAYOUT = 1 / 12f;
    // 该容器的内边距,无视 padding 属性，如需边距请用该变量
    private float mPadding;
    // 布局时的开始角度
    private double mStartAngle = 0;
    // 菜单项的文本
    private String[] mItemTexts;
    // 菜单项的图标
    private int[] mItemImgs;
    // 菜单的个数
```

```
    private int mMenuItemCount;
    // 菜单布局资源 id
    private int mMenuItemLayoutId = R.layout.circle_menu_item;
    // MenuItem 的点击事件接口
    private OnItemClickListener mOnMenuItemClickListener;

    public CircleMenuLayout(Context context, AttributeSet attrs) {
        super(context, attrs);
        // 无视 padding
        setPadding(0, 0, 0, 0);
    }

    // 设置菜单条目的图标和文本
    public void setMenuItemIconsAndTexts(int[] images, String[] texts) {
        if (images == null && texts == null) {
            throw new IllegalArgumentException("菜单项文本和图片至少设置其一");
        }

        mItemImgs = images;
        mItemTexts = texts;
        // 初始化 mMenuCount
        mMenuItemCount = images == null ? texts.length : images.length;
        if (images != null && texts != null) {
            mMenuItemCount = Math.min(images.length, texts.length);
        }
        // 构建菜单项
        buildMenuItems();
    }

    // 构建菜单项
    private void buildMenuItems() {
        // 根据用户设置的参数，初始化 menu item
        for (int i = 0; i < mMenuItemCount; i++) {
            View itemView = inflateMenuView(i);
            // 初始化菜单项
            initMenuItem(itemView, i);
            // 添加 view 到容器中
            addView(itemView);
        }
    }

    private View inflateMenuView(final int childIndex) {
        LayoutInflater mInflater = LayoutInflater.from(getContext());
        View itemView = mInflater.inflate(mMenuItemLayoutId, this, false);
        itemView.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                if (mOnMenuItemClickListener != null) {
                    mOnMenuItemClickListener.onClick(v, childIndex);
                }
            }
        });
        return itemView;
    }

    private void initMenuItem(View itemView, int childIndex) {
        ImageView iv = (ImageView) itemView
                .findViewById(R.id.id_circle_menu_item_image);
        TextView tv = (TextView) itemView
                .findViewById(R.id.id_circle_menu_item_text);
        iv.setVisibility(View.VISIBLE);
        iv.setImageResource(mItemImgs[childIndex]);
```

```
        tv.setVisibility(View.VISIBLE);
        tv.setText(mItemTexts[childIndex]);
    }
        // 设置 MenuItem 的布局文件，必须在 setMenuItemIconsAndTexts 之前调用
        public void setMenuItemLayoutId(int mMenuItemLayoutId) {
            this.mMenuItemLayoutId = mMenuItemLayoutId;
        }

        // 设置 MenuItem 的点击事件接口
        public void setOnItemClickListener(OnItemClickListener listener) {
            this.mOnMenuItemClickListener = listener;
        }
    // 代码省略
}
```

小民的思路大致是这样的，首先让用户通过 setMenuItemIconsAndTexts 函数将菜单项的图标和文本传递进来，根据这些图标和文本构建菜单项，菜单项的布局视图由 mMenuItemLayoutId 存储起来，这个 mMenuItemLayoutId 默认为 circle_menu_item.xml，这个 xml 布局为一个 ImageView 显示在一个文本控件的上面。为了菜单项的可定制性，小民还添加了一个 setMenuItemLayoutId 函数让用户可以设置菜单项的布局，希望用户可以定制各种各样的菜单样式。在用户设置了菜单项的相关数据之后，小民会根据用户设置进来的图标和文本数量来构建、初始化相等数量的菜单项，并且将这些菜单项添加到圆形菜单 CircleMenuLayout 中。然后添加了一个可以设置用户点击菜单项的处理接口的 setOnItemClickListener 函数，使得菜单的点击事件可以被用户自定义处理。

在将菜单项添加到 CircleMenuLayout 之后就是要对这些菜单项进行尺寸丈量和布局了，我们先来看丈量尺寸的代码具体如下。

```
//设置布局的宽高，并测量 menu item 宽高
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    // 丈量自身尺寸
    measureMyself(widthMeasureSpec, heightMeasureSpec);
    // 丈量菜单项尺寸
    measureChildViews();
}

private void measureMyself(int widthMeasureSpec, int heightMeasureSpec) {
    int resWidth = 0;
    int resHeight = 0;
    // 根据传入的参数，分别获取测量模式和测量值
    int width = MeasureSpec.getSize(widthMeasureSpec);
    int widthMode = MeasureSpec.getMode(widthMeasureSpec);
    int height = MeasureSpec.getSize(heightMeasureSpec);
    int heightMode = MeasureSpec.getMode(heightMeasureSpec);

    // 如果宽或高的测量模式非精确值
    if (widthMode != MeasureSpec.EXACTLY
        || heightMode != MeasureSpec.EXACTLY) {
        // 主要设置为背景图的高度
        resWidth = getSuggestedMinimumWidth();
        // 如果未设置背景图片，则设置为屏幕宽高的默认值
        resWidth = resWidth == 0 ? getDefaultWidth() : resWidth;
        resHeight = getSuggestedMinimumHeight();
        // 如果未设置背景图片，则设置为屏幕宽高的默认值
        resHeight = resHeight == 0 ? getDefaultWidth() : resHeight;
    } else {
```

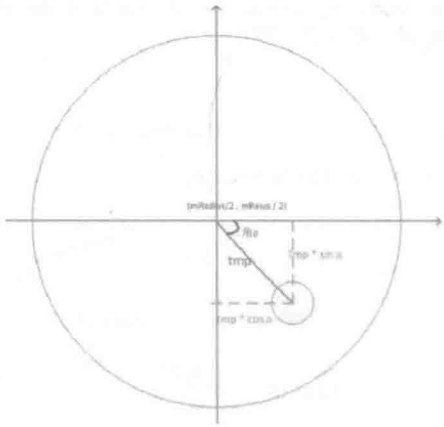
```
        // 如果都设置为精确值，则直接取小值
        resWidth = resHeight = Math.min(width, height);
    }
    setMeasuredDimension(resWidth, resHeight);
}

private void measureChildViews() {
    // 获得半径
    mRadius = Math.max(getMeasuredWidth(), getMeasuredHeight());
    // menu item 数量
    final int count = getChildCount();
    // menu item 尺寸
    int childSize = (int) (mRadius * RADIO_DEFAULT_CHILD_DIMENSION);
    // menu item 测量模式
    int childMode = MeasureSpec.EXACTLY;
    // 迭代测量
    for (int i = 0; i < count; i++) {
        final View child = getChildAt(i);
        if (child.getVisibility() == GONE) {
            continue;
        }
        // 计算 menu item 的尺寸，以及设置好的模式，去对 item 进行测量
        int makeMeasureSpec = -1;
        makeMeasureSpec = MeasureSpec.makeMeasureSpec(childSize, childMode);
        child.measure(makeMeasureSpec, makeMeasureSpec);
    }
    mPadding = RADIO_PADDING_LAYOUT * mRadius;
}
```

代码比较简单，就是先测量 CircleMenuLayout 的尺寸，然后测量每个菜单项的尺寸。尺寸获取了之后就到了布局这一步，这也是整个圆形菜单的核心所在。具体代码如下。

```
// 布局 menu item 的位置
@Override
protected void onLayout(boolean changed, int l, int t, int r, int b) {
    final int childCount = getChildCount();
    int left, top;
    // menu item 的尺寸
    int itemWidth = (int) (mRadius * RADIO_DEFAULT_CHILD_DIMENSION);
    // 根据 menu item 的个数，计算 item 的布局占用的角度
    float angleDelay = 360 / childCount;
    // 遍历所有菜单项，设置它们的位置
    for (int i = 0; i < childCount; i++) {
        final View child = getChildAt(i);
        if (child.getVisibility() == GONE) {
            continue;
        }
        // 菜单项的起始角度
        mStartAngle %= 360;
        // 计算，中心点到 menu item 中心的距离
        float distanceFromCenter = mRadius / 2f - itemWidth / 2 - mPadding;
        // distanceFromCenter cosa 即 menu item 中心点的 left 坐标
        left = mRadius/ 2 + (int) Math.round(distanceFromCenter
                    * Math.cos(Math.toRadians(mStartAngle)) - 1 / 2f
                    * itemWidth);
        // distanceFromCenter sina 即 menu item 的纵坐标
        top = mRadius/ 2 + (int) Math.round(distanceFromCenter
                    * Math.sin(Math.toRadians(mStartAngle)) - 1 / 2f
                    * itemWidth);
        // 布局 child view
        child.layout(left, top, left + itemWidth, top + itemWidth);
```

```
        // 叠加尺寸
        mStartAngle += angleDelay;
    }

}
```

onLayout 函数看起来稍显复杂，但它的含义就是将所有菜单项按照圆弧的形式布局。整个圆为 360°，如果每个菜单项占用的角度为 60°，第一个菜单项的角度为 0°~60°，那么第二个菜单项的角度就是 60°~120°，以此类推将所有菜单项按照圆形布局。首先要去计算每个菜单项的 left 和 top 位置，计算公式的图形化表示如图 20-7 所示。



▲图 20-7

图 20-7 右下角那个小圆就是我们的菜单项，那么它的 left 坐标就是 mRadius / 2 + tmp * coas , top 坐标则是 mRadius / 2 + tmp * sina 。这里的 tmp 就是我们代码中的 distanceFromCenter 变量。到了这一步之后小民的第一版圆形菜单算是完成了。

下面就来集成一下这个圆形菜单。

创建一个工程之后，首先在布局 xml 中添加圆形菜单控件，具体代码如下。

```
<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:background="@drawable/bg"
    android:gravity="center"
    android:orientation="horizontal" >

<com.dp.widgets.CircleMenuLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/id_menulayout"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/circle_bg" />

</LinearLayout>
```

为了更好地显示效果，在布局 xml 文件中为圆形菜单的上一层以及圆形菜单，本书都添加了一

个背景图。然后在 MainActivity 中设置菜单项数据以及点击事件等。代码如下所示：

```
public class MainActivity extends Activity {
    private CircleMenuLayout mCircleMenuLayout;
    // 菜单标题
    private String[] mItemTexts = new String [] {
            "安全中心 ", "特色服务", "投资理财",
            "转账汇款", "我的账户", "信用卡"
    };
    // 菜单图标
    private int[] mItemImgs = newint[] {
            R.drawable.home_mbank_1_normal,
            R.drawable.home_mbank_2_normal, R.drawable.home_mbank_3_normal,
            R.drawable.home_mbank_4_normal, R.drawable.home_mbank_5_normal,
            R.drawable.home_mbank_6_normal
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 初始化圆形菜单
        mCircleMenuLayout = (CircleMenuLayout) findViewById(R.id.id_menulayout);
        // 设置菜单数据项
        mCircleMenuLayout.setMenuItemIconsAndTexts(mItemImgs, mItemTexts);
        // 设置菜单项点击事件
        mCircleMenuLayout.setOnItemClickListener(new OnItemClickListener() {
            @Override
            public void onClick(View view, int pos) { Toast.makeText(MainActivity.this,
            mItemTexts[pos],Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

效果如图 20-8 和图 20-9 所示。

▲图 20-8　圆形菜单

▲图 20-9　点击菜单项

小民得意洋洋地蹦出了一个字：真酷！同时也为自己的学习能力感到骄傲，脸上写满了满足与自豪，感觉自己又朝高级工程师迈近了一步。

“这不是洋叔写的圆形菜单嘛，小民也下载了？”正准备下班的主管看到这个 UI 效果问清情况。小民只好把其中的缘由、实现方式一一说给主管听，小民还特地强调了 CircleMenuLayout 的可定制性，通过 setMenuItemLayoutId 函数设置菜单项的布局 id，这样菜单项的 UI 效果就可以被用户定制化了。主管扫视了小民的代码，似乎察觉出了什么。于是转身找来还在埋头研究代码的洋叔，并且把小民的实现简单介绍了一遍，洋叔老师在扫视了一遍代码之后就发现了其中的问题所在。

“小民，你刚才说用户通过 setMenuItemLayoutId 函数可以设定菜单项的 UI 效果。那么问题来了，在你的 CircleMenuLayout 中默认实现的是 circle_menu_item.xml 的逻辑，比如加载菜单项布局之后会通过 findViewById 找到布局中的各个子视图，并且进行数据绑定。例如设置图标和文字，但这是针对 circle_menu_item.xml 这个布局的具体实现。如果用户设置菜单项布局为 other_menu_item.xml，并且每个菜单项修改为一个 Button，那么，此时必须修改 CircleMenuLayout 中初始化菜单项的代码。因为布局变了，菜单项里面的子 View 类型也变化了，菜单需要的数据也发生了变化。例如菜单项不再需要图标，只需要文字。这样一来，用户每换一种菜单样式就需要修改一次 CircleMenuLayout 类一，并且设置菜单数据的接口也需要改变。这样就没有定制性可言了，而且明显违反了开闭原则。反复对 CircleMenuLayout 进行修改不免会引入各种各样的问题……”洋叔老师果然一针见血，小民这才发现了问题所在，于是请教洋叔应该如何处理比较合适。

“这种情况你应该使用 Adapter，就像 ListView 中的 Adapter 一样，让用户来自定义菜单项的布局、解析、数据绑定等工作，你需要知道的仅仅是每个菜单项都是一个 View。这样一来就将变化通过 Adapter 层隔离出去，你依赖的只是 Adapter 这个抽象。每个用户可以有不同的实现，你只需要实现圆形菜单的丈量、布局工作即可。这样就可以拥抱变化，可定制性就得到了保证。当然，你可以提供一个默认的 Adapter，也就是使用你的 circle_menu_item.xml 布局实现的菜单，这样没有定制需求的用户就可以使用这个默认的实现了。”小民频频点头，屡屡称是。“这确实是我之前没有考虑好，也是经验确实不足，我再好好重构一下。”小民发现问题之后也承认了自己的不足，两位前辈看小民这么好学就陪着小民一块重构代码。

在两位前辈的指点下，经过不到五分钟重构，小民的 CircleMenuLayout 成了下面这样。

```
// 圆形菜单
public class CircleMenuLayout extends ViewGroup {
    // 字段省略
    // 设置 Adapter
    public void setAdapter(ListAdapter mAdapter) {
        this.mAdapter = mAdapter;
    }

    // 构建菜单项
    private void buildMenuItems() {
        // 根据用户设置的参数，初始化 menu item
        for (int i = 0; i < mAdapter.getCount(); i++) {
            final View itemView = mAdapter.getView(i, null, this);
            final int position = i;
            itemView.setOnClickListener(new OnClickListener() {

                @Override
```

```
                public void onClick(View v) {
                    if (mOnMenuItemClickListener != null) {
                        mOnMenuItemClickListener.onClick(itemView, position);
                    }
                }
            });
            // 添加 view 到容器中
            addView(itemView);
        }
    }

    @Override
    protected void onAttachedToWindow() {
        if (mAdapter != null) {
            buildMenuItems();
        }
        super.onAttachedToWindow();
    }
    // 丈量、布局代码省略
}
```

现在的 CircleMenuLayout 把解析 xml、初始化菜单项的具体工作移除，添加了一个 Adapter，在用户设置了 Adapter 之后，在 onAttachedToWindow 函数中调用 Adapter 的 getCount 函数获取菜单项的数量，然后通过 getView 函数获取每个 View，最后将这些菜单项的 View 添加到圆形菜单中，圆形菜单布局再将它们布局到特定的位置即可。

我们看现在使用 CircleMenuLayout 是怎样的形式。首先定义了一个实体类 MenuItem 来存储菜单项图标和文本的信息，具体代码如下。

```
static class MenuItem {
    public int imageId;
    public String title;
    public MenuItem(String title, int resId) {
        this.title = title;
        imageId = resId;
    }
}
```

然后再实现一个 Adapter，这个 Adapter 的类型就是 ListAdapter。我们需要在 getView 中加载菜单项 xml、绑定数据等，相关代码如下。

```
static class CircleMenuAdapter extends BaseAdapter {
        List<MenuItem> mMenuItems;
        public CircleMenuAdapter(List<MenuItem> menuItems) {
            mMenuItems = menuItems;
        }

        // 加载菜单项布局，并初始化每个菜单
        @Override
        public View getView(final int position, View convertView, ViewGroup parent) {
            LayoutInflater mInflater = LayoutInflater.from(parent.getContext());
            View itemView = mInflater.inflate(R.layout.circle_menu_item, parent, false);
            initMenuItem(itemView, position);
            return itemView;
        }

        // 初始化菜单项
        private void initMenuItem(View itemView, int position) {
```

```
            // 获取数据项
            final MenuItem item = getItem(position);
            ImageView iv=(ImageView)itemView.findViewById(R.id.id_circle_menu_item_image);
            TextView tv=(TextView) itemView.findViewById(R.id.id_circle_menu_item_text);
            // 数据绑定
            iv.setImageResource(item.imageId);
            tv.setText(item.title);
        }
        // 省略获取 item count 等代码
    }
```

这与我们在 ListView 中使用 Adapter 是一致的，实现 getView、getCount 等函数，在 getView 中加载每一项的布局文件，并且绑定数据等。最终将菜单 View 返回，然后这个 View 就会被添加到 CircleMenuLayout 中。这一步的操作原来是放在 CircleMenuLayout 中的，现在被独立出来，并且通过 Adapter 进行了隔离。这样就将易变的部分通过 Adapter 抽象隔离开来，即使用户有成千上万种菜单项 UI 效果，那么通过 Adapter 就可以很容易地进行扩展、实现，而不需要每次都修改 CircleMenuLayout 中的代码。CircleMenuLayout 布局类相当于提供了一个圆形布局抽象，至于每一个子 View 是啥样的它并不关心。通过 Adapter 隔离变化，拥抱变化，就是这么简单。

“原来 ListView、RecyclerView 通过一个 Adapter 就是这个原因，通过 Adapter 将易变的部分独立出去交给用户处理。又通过观察者模式将数据和 UI 解耦合，使得 View 与数据没有依赖，一份数据可以作用于多个 UI，应对 UI 的易变性。原来如此！”小民最后总结道。

例如，当我们的产品发生变化，需要将圆形菜单修改为普通的 ListView 样式，那么要做的事很简单，就是将 xml 布局中的 CircleMenuLayout 修改为 ListView，然后将 Adapter 设置给 ListView 即可。具体代码如下。

```
public class MainActivity extends Activity {
    private ListView mListView;
    List<MenuItem> mMenuItems = new ArrayList<MenuItem>();

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

        // 模拟数据
        mockMenuItems();

        mListView = (ListView) findViewById(R.id.id_menulayout);
        // 设置适配器
        mListView.setAdapter(new CircleMenuAdapter(mMenuItems));
        // 设置点击事件
        mListView.setOnItemClickListener(new OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
                Toast.makeText(MainActivity.this, mMenuItems.get(position).title,
                Toast.LENGTH_SHORT).show();
            }
        });
}
```

这样就完成了 UI 替换，成本很低，也基本不会引发其他错误。这也就是为什么我们在 CircleMenuLayout 中要使用 ListAdapter 的原因，就是为了与现有的 ListView、GridView 等组件进

行兼容，当然我们也没有必要重新再定义一个 Adapter 类型。替换为 ListView 的效果如图 12-10 所示。

小民在重构完 CircleMenuLayout 之后深感收获颇多，为了报答主管和洋叔的指点嚷嚷着要请吃饭……

▲图 20-10

20.9 总结

Adapter 模式的经典实现在于将原本不兼容的接口融合在一起，使之能够很好地进行合作。但是，在实际开发中，Adapter 模式也有一些灵活的实现。例如 ListView 中的隔离变化，使得整个 UI 架构变得更灵活，能够拥抱变化。Adapter 模式在开发中运用非常广泛，因此，掌握 Adapter 模式是非常必要的。

优点

* 更好的复用性

系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到更好的复用。

* 更好的扩展性

在实现适配器功能的时候，可以调用自己开发的功能，从而自然地扩展系统的功能。

缺点

* 过多地使用适配器，会让系统非常零乱，不易整体把握。例如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此，如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

第 21 章　装饰模式

21.1 装饰模式介绍

装饰模式（Decorator Pattern）也称为包装模式（Wrapper Pattern），结构型设计模式之一，其使用一种对客户端透明的方式来动态地扩展对象的功能，同时它也是继承关系的一种替代方案之一。在现实生活中你也可以看见很多装饰模式的例子，或者可以大胆地说装饰模式无处不在，就拿人来说，人需要各式各样的衣着，不管你穿着怎样，但是，对于个人的本质来说是不变的，充其量只是在外面披上一层遮羞物而已，这就是装饰模式，装饰物也许各不相同但是装饰的对象本质是不变的。

21.2 装饰模式的定义

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。

21.3 装饰模式的使用场景

需要透明且动态地扩展类的功能时。

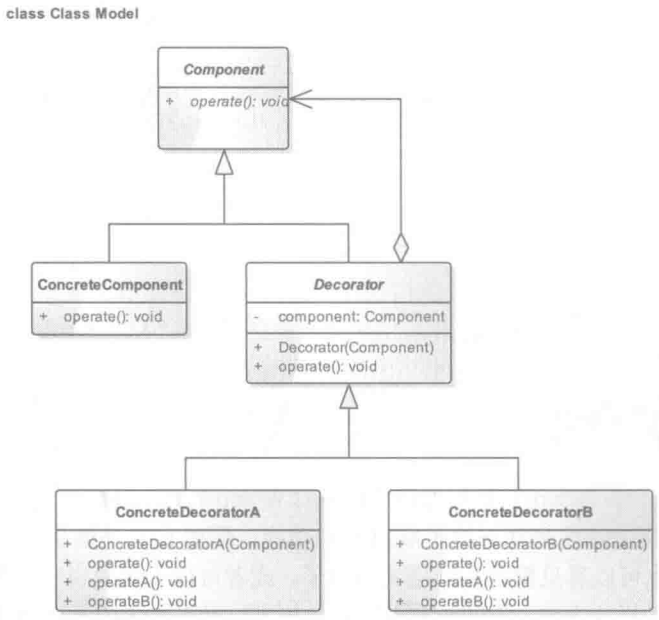
21.4 装饰模式的 UML 类图

UML 类图如图 21-1 所示。

根据类图可以得出如下一个装饰模式的通用模式代码。

程序清单 21-1　抽象组件类

```
public abstract class Component {
    /**
     * 抽象的方法，这个随你做
     * 同样地你也可以增加更多的抽象方法
     */
    public abstract void operate();
}
```



▲图 21-1

程序清单 21-2　组件具体实现类

```
public class ConcreteComponent extends Component {
    @Override
    public void operate() {
        // 具体逻辑，这个随你做
    }
}
```

程序清单 21-3　抽象装饰者

```
public abstract class Decorator extends Component {
    private Component component;// 持有一个 Component 对象的引用

    /**
     * 必要的构造方法 需要一个 Component 类型的对象
     *
     * @param component Component 对象
     */
    public Decorator(Component component) {
        this.component = component;
    }

    @Override
    public void operate() {
        component.operate();
    }
}
```

程序清单 21-4　装饰者具体实现类（注：ConcreteDecoratorB 和 ConcreteDecoratorA 本质并无区别只是实现不同，因此 ConcreteDecoratorB 的代码这里不再给出）

```
public class ConcreteDecoratorA extends Decorator {
    protected ConcreteDecoratorA(Component component) {
```

```
        super(component);
    }

    @Override
    public void operate() {
        // 装饰方法 A 和 B 既可在父类方法前调用也可在之后调用
        operateA();

        super.operate();

        operateB();
    }

    /**
     * 自定义的装饰方法 A
     */
    public void operateA() {
        // 装饰方法逻辑……
    }

    /**
     * 自定义的装饰方法 B
     */
    public void operateB() {
        // 装饰方法逻辑……
    }
}
```

程序清单 21-5　客户调用类

```
public class Client {
    public static void main(String[] args) {
        // 构造被装饰的组件对象
        Component component = new ConcreteComponent();

        // 根据组件对象构造装饰者对象 A 并调用，此时相当于给组件对象增加装饰者 A 的功能方法
        Decorator decorator = new ConcreteDecoratorA(component);
        decorator.operate();

        // 根据组件对象构造装饰者对象 B 并调用，此时相当于给组件对象增加装饰者 B 的功能方法
        Decorator decoratorB = new ConcreteDecoratorB(component);
        decoratorB.operate();
    }
}
```

角色介绍。

- Component：抽象组件。

可以是一个接口或抽象类，其充当的就是被装饰的原始对象。

- ConcreteComponent：组件具体实现类。

该类是 Component 类的基本实现，也是我们装饰的具体对象。

- Decorator：抽象装饰者。

顾名思义，其承担的职责就是为了装饰我们的组件对象，其内部一定要有一个指向组件对象的引用。在大多数情况下，该类为抽象类，需要根据不同的装饰逻辑实现不同的具体子类。当然，如果装饰逻辑单一，只有一个的情况下我们可以省略该类直接作为具体的装饰者。

- ConcreteDecoratorA：装饰者具体实现类。

只是对抽象装饰者作出具体的实现。

- ConcreteDecoratorB：同上。

同上。

- Client：客户类。

21.5 模式的简单实现

还是看文章开头我们所讲到的那个例子，人总是要穿衣服的，我们将人定义为一个抽象类，将其穿衣的行为定义为一个抽象方法。

```
public abstract class Person {
    /**
     * Person 下有一个穿着的抽象方法
     */
    public abstract void dressed();
}
```

该类其实就是上面我们所提及的抽象组件类，也就是我们需要装饰的原始对象，那么具体装饰谁呢？我们需要一个具体的实现类。

```
public class Boy extends Person {
    @Override
    public void dressed() {
        // Boy 类下 dressed 方法的基本逻辑
        System.out.println("穿了内衣内裤");
    }
}
```

Boy 类继承于 Person 类，该类仅对 Person 中的 dressed 方法作了具体的实现，而 Boy 类则是我们所要装饰的具体对象，现在需要一个装饰者来装饰我们的这个 Boy 对象，这里定义一个 PersonCloth 类来表示人所穿着的衣服。

```
public abstract class PersonCloth extends Person {
    protected Person mPerson;// 保持一个 Person 类的引用

    public PersonCloth(Person mPerson) {
        this.mPerson = mPerson;
    }

    @Override
    public void dressed() {
        // 调用 Person 类中的 dressed 方法
        mPerson.dressed();
    }
}
```

在 PersonCloth 类中我们保持了一个对 Person 类的引用，可以方便地调用具体被装饰对象中的方法，这也是为什么我们可以在不破坏原类层次结构的情况下为类增加一些功能，我们只需要在被装饰对象的相应方法前或后增加相应的功能逻辑即可。在装饰物只有一个的情况下，可不必声明一个抽象类作为装饰者抽象的提取，仅需定义一个普通的类表示装饰者即可，这里为了表明示例我们定义两种衣服类型，一个类 ExpensiveCloth 表示高档衣服。

```
public class ExpensiveCloth extends PersonCloth {
    public ExpensiveCloth(Person person) {
        super(person);
    }

    /**
     * 穿短袖
     */
    private void dressShirt() {
        System.out.println("穿件短袖");
    }

    /**
     * 穿皮衣
     */
    private void dressLeather() {
        System.out.println("穿件皮衣");
    }

    /**
     * 穿牛仔裤
     */
    private void dressJean() {
        System.out.println("穿条牛仔裤");
    }

    @Override
    public void dressed() {
        super.dressed();
        dressShirt();
        dressLeather();
        dressJean();
    }
}
```

逻辑依旧很简单不再多说，而另一个类 CheapCloth 则表示便宜的衣服。

```
public class CheapCloth extends PersonCloth {
    public CheapCloth(Person mPerson) {
        super(mPerson);
    }

    /**
     * 穿短裤
     */
    public void dressShorts() {
        System.out.println("穿条短裤");
    }

    @Override
    public void dressed() {
        super.dressed();
        dressShorts();
    }
}
```

这两个类本质上并没有区别，两者都是为原本 Boy 类中的 dressed 方法提供功能扩展，不过这种扩展并非是直接修改原有的方法逻辑或结构，更恰当地说，仅仅是在另一个类中将原有方法和新逻辑进行封装整合而已。最后我们来看看客户类中的调用。

```
public class Main {
    public static void main(String[] args) {
        // 首先我们要有一个 Person 男孩
        Person person = new Boy();

        // 然后为他穿上便宜的衣服，比如爱哥这样的 Boy
        PersonCloth clothCheap = new CheapCloth(person);
        clothCheap.dressed();

        // 或者给他穿上比较上档次的衣服，如 SM
        PersonCloth clothExpensive = new ExpensiveCloth(person);
        clothExpensive.dressed();
    }
}
```

21.6 Android 源码中的模式实现

源码中的装饰模式其实我们常接触但是并不一定有过真正的了解，Context 类在 Android 中被称为“上帝对象”，它本质是一个抽象类，其在我们装饰模式里就相当于抽象组件，而在其内部定义了大量的抽象方法，比如我们经常会用到的 startActivity 方法。

```
public abstract class Context {

    // ......省略一些代码......

    public abstract void startActivity(Intent intent);

    public abstract void startActivity(Intent intent, @Nullable Bundle options);

    // ......省略一些代码......
}
```

而其真正的实现是在 ContextImpl 中完成的，ContextImpl 继承自 Context 抽象类，并实现了 Context 中的抽象方法，具体代码如下。

```
class ContextImpl extends Context {

    // ......省略一些代码......

    @Override
    public void startActivity(Intent intent) {
        warnIfCallingFromSystemProcess();
        startActivity(intent, null);
    }

    @Override
    public void startActivity(Intent intent, Bundle options) {
        warnIfCallingFromSystemProcess();
        if ((intent.getFlags()&Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
            throw new AndroidRuntimeException(
                    "Calling startActivity() from outside of an Activity "
                    + " context requires the FLAG_ACTIVITY_NEW_TASK flag."
                    + " Is this really what you want?");
        }
        mMainThread.getInstrumentation().execStartActivity(
            getOuterContext(), mMainThread.getApplicationThread(), null,
            (Activity)null, intent, -1, options);
```

```
    }

    // ......省略一些代码......
}
```

这里 ContextImpl 就相当于组件具体实现类，那么谁来承担装饰者的身份呢？我们知道 Activity 从类层次上来说本质是一个 Context，如果大家留意过其代码会发现 Activity 并非直接继承于 Context，而是继承于 ContextThemeWrapper。

```
public class Activity extends ContextThemeWrapper
        implements LayoutInflater.Factory2,
        Window.Callback, KeyEvent.Callback,
        OnCreateContextMenuListener, ComponentCallbacks2,
        Window.OnWindowDismissedCallback {

        // ......省略具体的类代码......

}
```

而这个 ContextThemeWrapper 又是继承于 ContextWrapper。

```
public class ContextThemeWrapper extends ContextWrapper {

    // ......省略具体的类代码......

}
```

最终的这个 ContextWrapper 才继承于 Context，为什么类层次会这么复杂呢？其实这里就是一个典型的装饰模式，ContextWrapper 就是我们要找的装饰者，在 ContextWrapper 中有一个 Context 的引用。

```
public class ContextWrapper extends Context {
    Context mBase;

    public ContextWrapper(Context base) {
        mBase = base;
    }

    // ......省略一些代码......
}
```

其实从这里我们已经可以看出一点装饰者的模样了，但是好像还缺点什么，没错，一个来自于抽象组件的方法，这里我们还是先只看 startActivity，在 ContextWrapper 中同样有一个 startActivity 方法。

```
public class ContextWrapper extends Context {

    // ......省略一些代码......

    @Override
    public void startActivity(Intent intent) {
        mBase.startActivity(intent);
    }

    // ......省略一个方法......
```

```
    @Override
    public void startActivity(Intent intent, Bundle options) {
        mBase.startActivity(intent, options);
    }

    // ......省略一些代码......
}
```

这里可以看到 ContextWrapper 中的 startActivity 方法也仅仅是简单地调用了具体组件实现类 ContextImpl 中的对应方法而已，实质上 ContextWrapper 中的所有方法都仅仅是调用了 ContextImpl 中的对应方法，是不是与我们上面所讲的装饰模式例子或类图越来越像了？装饰模式应用的套路都是很相似的，对于具体方法的包装扩展则由 ContextWrapper 的具体子类完成，比如我们的 Actvity、Service 和 Application。这 3 个类的具体方法这里就不再给出，大家可自行查看。可以看到即便是在 Android 源码中应用，装饰模式也并不难理解，比较简单。

21.7 Context 与 ContextImpl

Context 对大家来说并不陌生，相信很多读者在第一天学习 Android 的时候就会接触到它，那么为什么这里我们会称为“上帝”对象呢？因为其在 Android 应用层开发中占据了绝对重要的地位，不管是 Framework 提供给我们的四大组件还是应用级别的 Application，还是负责表现层的 View 相关类，甚至连我们很多时候创建的实体类都会需要持有一个 Context 的引用，在 Framework 中，Context 贯穿了整个 Android 的 MVC 框架，它的应用很普遍。尽管如此，Context 的本质还是一个普通的抽象类，我们在上面的装饰模式源码分析中提到 Context 类中的 startActivity 方法，实质上除了 startActivity，Context 中还定义了大量相当牛的方法，这些方法不但与我们上面提到的四大组件密切相关，还与资源文件、文件管理、包管理、类加载、权限管理、系统级服务获取等各式各样的功能相切合。

```
public abstract class Context {
    // ......省略一些方法定义......

    // 启动一个 Activity
    public abstract void startActivity(Intent intent);

    // 启动一个 Activity
    public abstract void startActivity(Intent intent, @Nullable Bundle options);

    // 启动多个 Activity
    public abstract void startActivities(Intent[] intents);

    // 启动多个 Activity
    public abstract void startActivities(Intent[] intents, Bundle options);

    // 获取应用程序信息
    public abstract ApplicationInfo getApplicationInfo();

    // 启动一个 Service
    public abstract ComponentName startService(Intent service);

    // 停止一个 Service
    public abstract boolean stopService(Intent service);
```

```
    // 绑定一个 Service 到 Activity
    public abstract boolean bindService(Intent service, @NonNull ServiceConnection conn,
    @BindServiceFlags int flags);

    // 解绑一个 Service 到 Activity
    public abstract void unbindService(@NonNull ServiceConnection conn);

    // 发送一个广播
    public abstract void sendBroadcast(Intent intent,@Nullable String receiverPermission);

    // 发送一个广播
    public abstract void sendBroadcast(Intent intent, String receiverPermission, int appOp);

    // 发送一个有序广播
    public abstract void sendOrderedBroadcast(Intent intent,@Nullable String receiverPermission);

    // 发送一个有序广播
    public abstract void sendOrderedBroadcast(@NonNull Intent intent, @Nullable String
receiverPermission, BroadcastReceiver resultReceiver, @Nullable Handler scheduler, int
initialCode, @Nullable String initialData, @Nullable Bundle initialExtras);

    // 发送一个有序广播
    public abstract void sendOrderedBroadcast(Intent intent, String receiverPermission,
int appOp, BroadcastReceiver resultReceiver, Handler scheduler, int initialCode, String
initialData, Bundle initialExtras);

    // 发送一个会滞留的广播
    public abstract void sendStickyBroadcast(Intent intent);

    // 发送一个会滞留的有序广播
    public abstract void sendStickyOrderedBroadcast(Intent intent, BroadcastReceiver
resultReceiver, @Nullable Handler scheduler, int initialCode, @Nullable String initialData,
@Nullable Bundle initialExtras);

    // 注册一个广播接收者
    public abstract Intent registerReceiver(@Nullable BroadcastReceiver receiver,
    IntentFilter filter);

    // 注册一个广播接收者
    public abstract Intent registerReceiver(BroadcastReceiver receiver, IntentFilter
    filter, @Nullable String broadcastPermission, @Nullable Handler scheduler);

    // 取消注册的一个广播接收者
    public abstract void unregisterReceiver(BroadcastReceiver receiver);

    // 获取一个 ContentResolver 对象
    public abstract ContentResolver getContentResolver();

    // 获取主线程中的 Looper
    public abstract Looper getMainLooper();

    // 获取应用级的 Context 对象
    public abstract Context getApplicationContext();

    // 获取相应的系统服务对象
    public abstract Object getSystemService(String name);

    // 获取 Resources 对象处理资源文件
    public abstract Resources getResources();

    // 获取对象文件名的 SharedPreferences 对象
```

```
    public abstract SharedPreferences getSharedPreferences(String name, int mode);

    // 获取外部文件存储目录
    public abstract File getExternalFilesDir(@Nullable String type);

    // 获取多个（如果有的话）外部文件存储目录
    public abstract File[] getExternalFilesDirs(String type);

    // ......省略一些方法定义......
}
```

以上的这些方法我们在开发中都经常用到，当然这里仅仅是一个定义声明，如上面装饰模式源码分析中我们所说，Context 中方法的所有实现均由 ContextImpl 类承担，具体相关方法的实现这里就不再多说了，那么 ContextImpl 又是在何时、在何地被创建的呢？为什么上面我们会说装饰者类 ContextWrapper 中的 Context 引用指向的是一个 ContextImpl 的实例呢？要搞清楚这个问题，首先就要知道一个应用是从哪开始启动的，我们知道 Java 的应用入口是 main 方法，但是，在 Android 应用开发中我们从来都是从一个 Activity 的 onCreate 方法开始，从未接触过类似 main 一样的程序入口方法，但是，main 方法在 Android 中确实存在的，在应用级别的代码里，其匿藏在 ActivityThread 类中。

```
public final class ActivityThread {
    // ......省去一些代码......

    public static void main(String[] args) {
        // 启动 Android 的性能分析工具
        SamplingProfilerIntegration.start();

        // 暂时关闭 CloseGuard 检测 稍后会打开
        CloseGuard.setEnabled(false);

        // 为当前用户初始化环境参数
        Environment.initForCurrentUser();

        // 配置 EventLogger
        EventLogger.setReporter(new EventLoggingReporter());

        // 为 Security 安全系统添加 AndroidKeyStoreProvider 算法
        Security.addProvider(new AndroidKeyStoreProvider());

        // 获取并确保信用凭证在目录中
        final File configDir = Environment.getUserConfigDirectory(UserHandle.myUserId());
        TrustedCertificateStore.setDefaultUserDirectory(configDir);

        // 设置启动标识值，标识主循环准备初始化
        Process.setArgV0("<pre-initialized>");

        // 准备循环主线程（也就是当前线程）中的 Looper
        Looper.prepareMainLooper();

        // 构造 ActivityThread 对象并 Attach
        ActivityThread thread = new ActivityThread();
        thread.attach(false);

        // 获取主线程中的 Handler
        if (sMainThreadHandler == null) {
            sMainThreadHandler = thread.getHandler();
        }
```

```
        // 这段代码用于调试用
        if (false) {
            Looper.myLooper().setMessageLogging(new
                    LogPrinter(Log.DEBUG, "ActivityThread"));
        }

        // 调用 Looper 的 loop 方法使主线程中消息循环
        Looper.loop();

        // 如果代码执行到这里说明主线程消息循环出现异常，这时则抛出异常终止系统运行
        throw new RuntimeException("Main thread loop unexpectedly exited");
    }
}
```

ActivityThread 中的 main 方法逻辑并不多，大多复杂的逻辑都被封装在 ActivityThread 类中，而这里我们也主要只关注 ActivityThread 类，在上面的 main 方法代码中构造了一个 ActivityThread 对象，随后又为 sMainThreadHandler 这个 Handler 引用赋值，sMainThreadHandler 为 ActivityThread 的类变量，这里 sMainThreadHandler 指向的对象是由 ActivityThread 的 getHandler 方法获取，该方法只是单纯地返回了一个 H 类型的对象。

```
public final class ActivityThread {
    // ......省去一些代码......

    final H mH = new H();

    // ......省去一些代码......

    final Handler getHandler() {
        return mH;
    }

    // ......省去一些代码......
}
```

H 是 ActivityThread 的一个私有内部类，继承于 Handler，其作用很简单，与我们平时开发中使用 Handler 的作用一样，处理消息以及将消息压入队列，在 H 的 handleMessage 方法中可以得到一段处理 Activity 启动的代码逻辑。

```
public final class ActivityThread {
    // ......省去一些代码......

    private class H extends Handler {
        // ......省去大量代码......

        public void handleMessage(Message msg) {
            // ......省去一行代码......

            switch (msg.what) {
                case LAUNCH_ACTIVITY: {
                    // ......省去一行代码......

                    // 获取记录 Activity 状态等相关信息的 ActivityClientRecord 对象
                    final ActivityClientRecord r =  (ActivityClientRecord) msg.obj;

                    // 获取 PackageInfo
                    r.packageInfo = getPackageInfoNoCheck(r.activityInfo.applicationInfo,
```

```
r.compatInfo);

                    // 处理 Activity 启动
                    handleLaunchActivity(r, null);

                    // ......省去一行代码......
                } break;
                // ......省去多个分支语句代码......
            }
            // ......省去一行代码......
        }
        // ......省去一段代码......
    }
    // ......省去一些代码......
}
```

在处理启动 Activity 的这段分支代码里，主要是调用了 handleLaunchActivity 方法来处理启动的具体逻辑，在 handleLaunchActivity 方法中又会调用 performLaunchActivity 方法来获取一个 Activity 的实例。

```
public final class ActivityThread {
    // ......省去一些代码......

    private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
        // ......省去大量代码......

        Activity a = performLaunchActivity(r, customIntent);

        // ......省去很多代码......
    }
    // ......省去一些代码......
}
```

而这个 performLaunchActivity 方法才是真正处理 Activity 启动的关键所在，其中对大量 Activity 相关参数进行初始化。

```
public final class ActivityThread {
    // ......省去一些代码......

    private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
        // 获取 ActivityInfo 信息
        ActivityInfo aInfo = r.activityInfo;
        if (r.packageInfo == null) {
            r.packageInfo = getPackageInfo(aInfo.applicationInfo, r.compatInfo, Context.
            CONTEXT_INCLUDE_CODE);
        }

        // 获取要启动的 Activity 的组件信息
        ComponentName component = r.intent.getComponent();
        if (component == null) {
            component = r.intent.resolveActivity(mInitialApplication.getPackageManager());
            r.intent.setComponent(component);
        }

        // 根据相关信息构建组件对象
        if (r.activityInfo.targetActivity != null) {
            component=new ComponentName(r.activityInfo.packageName,r.activityInfo.targetActivity);
        }
```

```
// 这里开始构造具体的 Activity 对象
Activity activity = null;
try {
    // 获取 PackageInfo 所描述的类加载器
    java.lang.ClassLoader cl = r.packageInfo.getClassLoader();

    // 通过 Instrumentation 对象获取一个 Activity 对象
    activity=mInstrumentation.newActivity(cl,component.getClassName(),r.intent);
    StrictMode.incrementExpectedActivityCount(activity.getClass());

    // 设置相关参数准备初始化 Activity
    r.intent.setExtrasClassLoader(cl);
    r.intent.prepareToEnterProcess();
    if (r.state != null) {
        r.state.setClassLoader(cl);
    }
} catch (Exception e) {
    // ......省去异常处理代码......
}

try {
    // 获取 Application 对象
    Application app = r.packageInfo.makeApplication(false, mInstrumentation);

    if (activity != null) {
        // 获取 Context 对象
        Context appContext = createBaseContextForActivity(r, activity);
        CharSequence title=r.activityInfo.loadLabel(appContext.getPackageManager());

        // 构造 Configuration 对象
        Configuration config = new Configuration(mCompatConfiguration);

        // 通过 attach 方法将上述信息保存在 Activity 内部
        activity.attach(appContext, this, getInstrumentation(), r.token, r.ident,
        app, r.intent, r.activityInfo, title, r.parent, r.embeddedID,
        r.lastNonConfiguration Instances, config, r.referrer, r.voiceInteractor);

        if (customIntent != null) {
            activity.mIntent = customIntent;
        }
        r.lastNonConfigurationInstances = null;
        activity.mStartedActivity = false;

        // 设置主题
        int theme = r.activityInfo.getThemeResource();
        if (theme != 0) {
            activity.setTheme(theme);
        }
        activity.mCalled = false;

        // 根据是否需要持久化调用 callActivityOnCreate 方法通知 Activity 已被创建和启动
        if (r.isPersistable()) {
            mInstrumentation.callActivityOnCreate(activity,r.state,r.persistentState);
        } else {
            mInstrumentation.callActivityOnCreate(activity, r.state);
        }
        if (!activity.mCalled) {
            // ......省去异常处理代码......
        }
        r.activity = activity;
        r.stopped = true;
        if (!r.activity.mFinished) {
```

```
                activity.performStart();
                r.stopped = false;
            }
            if (!r.activity.mFinished) {
                // 同上
                if (r.isPersistable()) {
                    if (r.state != null || r.persistentState != null) {
                        mInstrumentation.callActivityOnRestoreInstanceState(activity,
                        r.state, r.persistentState);
                    }
                } else if (r.state != null) {
                    mInstrumentation.callActivityOnRestoreInstanceState(activity,
                    r.state);
                }
            }
            if (!r.activity.mFinished) {
                activity.mCalled = false;
                // 同上
                if (r.isPersistable()) {
                    mInstrumentation.callActivityOnPostCreate(activity,r.state,
                    r.persistentState);
                } else {
                    mInstrumentation.callActivityOnPostCreate(activity, r.state);
                }
                if (!activity.mCalled) {
                    throw new SuperNotCalledException("Activity " +
                    r.intent.getComponent().toShortString() + " did not call through to
                    super.onPostCreate()");
                }
            }
        }
        r.paused = true;

        mActivities.put(r.token, r);
    } catch (SuperNotCalledException e) {
        // ......省去异常处理代码......
    } catch (Exception e) {
        // ......省去异常处理代码......
    }
    return activity;
}
// ......省去一些代码......
}
```

由上述代码我们可以看到，启动一个 Activity 需要 3 个非常重要的对象：Application、Context 和 Configuration 对象。这里我们不对 Activity 的启动过程做过多分析，主要看看这里启动 Activity 所需要的 Context 对象是如何被构造出来的。

```
public final class ActivityThread {
    // ......省去一些代码......

    private Context createBaseContextForActivity(ActivityClientRecord r,
            final Activity activity) {
        ContextImpl appContext=ContextImpl.createActivityContext(this,r.packageInfo,r.token);
        appContext.setOuterContext(activity);
        Context baseContext = appContext;

        // ......省去两段代码......
        return baseContext;
    }
```

```
    // ......省去一些代码......
}
```

在 createBaseContextForActivity 方法中，通过 ContextImpl 的静态方法 createActivityContext 获得一个 ContextImpl 的实例对象，并通过 setOuterContext 方法将两者建立关联。同样地，如果我们回到 H 类中，我们会发现处理 Application 的相关逻辑。

```
public final class ActivityThread {
    // ......省去一些代码......

    private class H extends Handler {
        // ......省去一些代码......

        public void handleMessage(Message msg) {
            // ......省去一行代码......

            switch (msg.what) {
                // ......省去 N 多分支语句代码......
                case BIND_APPLICATION:
                    // ......省去一行代码......
                    AppBindData data = (AppBindData)msg.obj;
                    handleBindApplication(data);
                    // ......省去一行代码......
                    break;
                // ......省去 N 多分支语句代码......
            }
            // ......省去一行代码......
        }
        // ......省去一段代码......
    }
    // ......省去一些代码......
}
```

handleBindApplication 方法逻辑很复杂，但是我们只需要关心 Application 是如何被创建的，因此只需要关注一句代码。

```
public final class ActivityThread {
    // ......省去一些代码......

    private void handleBindApplication(AppBindData data) {
        Application app = data.info.makeApplication(data.restrictedBackupMode, null);
    }
    // ......省去一些代码......
}
```

handleBindApplication 中又调用 makeApplication 方法生成一个 Application 对象，根据前面对 Activity 的分析可以猜想在这个方法里，是否可以得到 Context 被创建以及关联的相关信息呢？答案是肯定的。

```
public final class ActivityThread {
    // ......省去一些代码......

    public Application makeApplication(boolean forceDefaultAppClass, Instrumentation
    instrumentation) {
        // 相对成员变量 mApplication 做非空判断免得重复构造
        if (mApplication != null) {
            return mApplication;
        }
```

```
        Application app = null;
        String appClass = mApplicationInfo.className;
        if (forceDefaultAppClass || (appClass == null)) {
            appClass = "android.app.Application";
        }
        try {
            java.lang.ClassLoader cl = getClassLoader();
            if (!mPackageName.equals("android")) {
                initializeJavaContextClassLoader();
            }

            // 获取一个 ContextImpl 对象
            ContextImpl appContext = ContextImpl.createAppContext(mActivityThread, this);

            // 获取一个 Application 对象
            app=mActivityThread.mInstrumentation.newApplication(cl,appClass,appContext);

            // 将 Application 与得到的 ContextImpl 对象关联
            appContext.setOuterContext(app);
        } catch (Exception e) {
            // ......省去异常处理代码......
        }
        mActivityThread.mAllApplications.add(app);
        mApplication = app;
        if (instrumentation != null) {
            try {
                instrumentation.callApplicationOnCreate(app);
            } catch (Exception e) {
                // ......省去异常处理代码......
            }
        }
        // ......省去一段代码......
        return app;
    }
    // ......省去一些代码......
}
```

这里可以看到，虽然具体的代码不尽相同，但是创建 Application 和 Context 以及将两者关联的套路基本是与 Activity 一致的，当然还有 Service，具体的就不再多说了，如上文我们所说，ContextImpl 中有大量碉堡的方法供我们调用，但是，在我们的一个应用中，到底存在几个这样的 Context 对象呢？其实从上文中都已经可以看到，一个 Application 对象会有一个 Context，而 Application 在应用中是唯一的，同时一个 Activity 或 Service 又分别表示一个 Context，因此，可以大胆地得出假设，一个应用中 Context 对象的总数应该等于 Activity 对象与 Service 对象之和再加上一个 Application，那么四大组件中其他的两个 BroadcastReciver 和 ContentProvider 没有保持 Context 对象了么？BroadcastReciver 并非直接或间接继承于 Context，但是每次接收广播的时候，onReceive 方法都会收到一个 Context 对象，该 Context 对象是 ReceiverRestrictedContext 的一个实例；而在 ContentProvider 中你可以调用其 getContext 方法获取一个 Context 对象。这些 Context 都直接或间接来自于 Application、Activity 和 Service。

在平时我们的应用开发中，经常会使用到 Context 对象，那么如上分析中，Context 在应用开发里的来源无非就 3 个，我们该如何选择使用哪一个 Context 对象呢？一个比较简单的辨别是，当你无法确定使用某个 Context 对象是否会造成长引用导致内存泄漏时就使用 Application 的 Context 对象，因为 Application 存在于整个应用的生命周期内。

21.8 模式实战

其实从上面一大堆的枯燥定义中已经可以看到，装饰模式其实并不复杂，就是一种类间的封装，我们在平时的开发过程中也经常会不经意地使用到类似的概念，比如我们常会在 Activity 的 onCreate 方法中做一些相关的初始化操作。

```
public class DecoratorActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // 初始化数据
        initData();

        // 初始化控件
        initViews();

        // 初始化事件
        initEvent();
    }

    /**
     * 初始化数据
     */
    private void initData() {

    }

    /**
     * 初始化控件
     */
    private void initViews() {
        setContentView(R.layout.activity_main);
    }

    /**
     * 初始化事件
     */
    private void initEvent() {

    }
}
```

其实如上的一些新增方法的调用就类似装饰模式中装饰者的职责，只不过这里我们没有保持对组件类的引用。因此，对大家来说，这个模式还是算很好理解的。

21.9 总结

装饰模式和我们前面所讲到的代理模式有点类似，有时甚至容易混淆，倒不是说会把代理当成装饰，而常常会是将装饰看作代理，所以大家一定要多留心，装饰模式是以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案；而代理模式则是给一个对象提供一个代理对象，并有代理对象来控制对原有对象的引用。装饰模式应该为所装饰的对象增强功能；代理模式对代理的对象施加控制，但不对对象本身的功能进行增强。

第 22 章　对象共享，避免创建多对象——享元模式

22.1 享元模式介绍

享元模式是对象池的一种实现，它的英文名称叫做 Flyweight，代表轻量级的意思。享元模式用来尽可能减少内存使用量，它适合用于可能存在大量重复对象的场景，来缓存可共享的对象，达到对象共享、避免创建过多对象的效果，这样一来就可以提升性能、避免内存移除等。

享元对象中的部分状态是可以共享，可以共享的状态成为内部状态，内部状态不会随着环境变化；不可共享的状态则称之为外部状态，它们会随着环境的改变而改变。在享元模式中会建立一个对象容器，在经典的享元模式中该容器为一个 Map，它的键是享元对象的内部状态，它的值就是享元对象本身。客户端程序通过这个内部状态从享元工厂中获取享元对象，如果有缓存则使用缓存对象，否则创建一个享元对象并且存入容器中，这样一来就避免了创建过多对象的问题。

22.2 享元模式定义

使用共享对象可有效地支持大量的细粒度的对象。

22.3 享元模式的使用场景

（1）系统中存在大量的相似对象。

（2）细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份。

（3）需要缓冲池的场景。

22.4 享元模式的 UML 类图

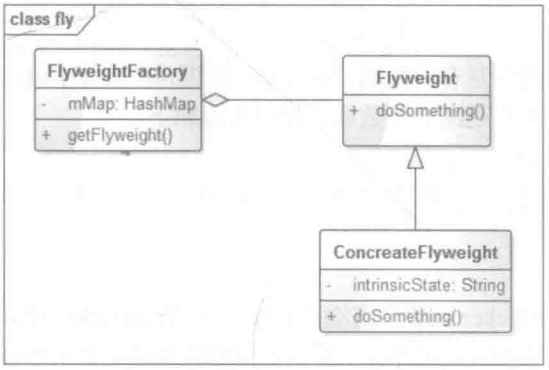
UML 类图如图 22-1 所示。

角色介绍。

Flyweight：享元对象抽象基类或者接口。

ConcreteFlyweight：具体的享元对象。

FlyweightFactory：享元工厂，负责管理享元对象池和创建享元对象。



▲图 22-1

22.5 享元模式的简单示例

过年回家买火车票是一件很困难的事，无数人用刷票插件软件在向服务端发出请求，对于每一个请求服务器都必须做出应答。在用户设置好出发地和目的地之后，每次请求都返回一个查询的车票结果。为了便于理解，我们假设每次返回的只有一躺列车的车票。那么当数以万计的人不间断在请求数据时，如果每次都重新创建一个查询的车票结果，那么必然会造成大量重复对象的创建、销毁，使得 GC 任务繁重、内存占用率高居不下。而这类问题通过享元模式就能够得到很好地改善，从城市 A 到城市 B 的车辆是有限的，车上的铺位也就是硬卧、硬卧、坐票 3 种。我们将这些可以公用的对象缓存起来，在用户查询时优先使用缓存，如果没有缓存则重新创建。这样就将成千上万的对象变为了可选择的有限数量。

首先我们创建一个 Ticket 接口，该接口定义展示车票信息的函数，具体代码如下。

```
public interface Ticket {
    public void showTicketInfo(String bunk);
}
```

它的一个具体的实现类是 TrainTicket 类，具体代码如下。

```
// 火车票
class TrainTicket implements Ticket {
    public String from; // 始发地
    public String to; // 目的地
    public String bunk; // 铺位
    publicint price;

    TrainTicket(String from, String to) {
        this.from = from;
        this.to = to;
    }

    @Override
```

```
    public void showTicketInfo(String bunk) {
        price = new Random().nextInt(300);
        System.out.println("购买 从 " + from + " 到 " + to + "的 "
                + bunk + " 火车票" + ", 价格 : " + price);
    }
}
```

数据库中表示火车票的信息有出发地、目的地、铺位、价格等字段，在购票用户每次查询时如果没有用某种缓存模式，那么返回车票数据的接口实现如下。

```
    public class TicketFactory {
        public static Ticket getTicket(String from, String to) {
            return new TrainTicket(from, to);
    }
}
```

在 TicketFactory 的 getTicket 函数中每次会 new 一个 TrainTicket 对象，也就是说如果在短时间内有 10000 万用户求购北京到青岛的车票，那么北京到青岛的车票对象就会被创建 10000 次，当数据返回之后这些对象变得无用了又会被虚拟机回收。此时就会造成大量的重复对象存在内存中，GC 对这些对象的回收也会非常消耗资源。如果用户的请求量很大可能导致系统变得极其缓慢，甚至可能导致 OOM。

正如上文所说，享元模式通过消息池的形式有效地减少了重复对象的存在。它通过内部状态标识某个种类的对象，外部程序根据这个不会变化的内部状态从消息池中取出对象。使得同一类对象可以被复用，避免大量重复对象。

使用享元模式很简单，只需要简单地改造一下 TicketFactory，具体代码如下。

```
// 车票工厂,以出发地和目的地为 key 缓存车票
public class TicketFactory {
    static Map<String, Ticket> sTicketMap = new ConcurrentHashMap<String, Ticket>();

    public static Ticket getTicket(String from, String to) {
        String key = from + "-" + to;
        if (sTicketMap.containsKey(key)) {
            System.out.println("使用缓存 ==> " + key);
            return sTicketMap.get(key);
        } else {
            System.out.println("创建对象 ==> " + key);
            Ticket ticket = new TrainTicket(from, to);
            sTicketMap.put(key, ticket);
            return ticket;
        }
    }
}
```

我们在 TicketFactory 添加了一个 map 容器，并且以出发地 + “-” + 目的地为键、以车票对象作为值存储车票对象。这个 map 的键就是我们说的内部状态，在这里就是出发地、横杠、目的地拼接起来的字符串，如果没有缓存则创建一个对象，并且将这个对象缓存到 map 中，下次再有这类请求时则直接从缓存中获取。这样即使有 10000 个请求北京到青岛的车票信息，那么出发地是北京、目的地是青岛的车票对象只有一个。这样就从这个对象从 10000 减到了 1 个，避免了大量的内存占用及频繁的 GC 操作。简单实现代码如下。

```
public class Test {
    public static void main(String[] args) {
        Ticket ticket01 = TicketFactory.getTicket("北京", "青岛");
        ticket01.showTicketInfo("上铺");
        Ticket ticket02 = TicketFactory.getTicket("北京", "青岛");
        ticket02.showTicketInfo("下铺");
        Ticket ticket03 = TicketFactory.getTicket("北京", "青岛");
        ticket03.showTicketInfo("坐票");
    }
}
```

运行结果：

```
创建对象 ==>北京-青岛
购买从北京到青岛的上铺火车票，价格 : 28
使用缓存 ==>北京-青岛
购买从北京到青岛的下铺火车票，价格 : 104
使用缓存 ==>北京-青岛
购买从北京到青岛的坐票火车票，价格 : 148
```

从输出结果可以看到，只有第一次查询车票时创建了一次对象，后续的查询都使用的是消息池中的对象。这其实就是相当于一个对象缓存，避免了对象的重复创建与回收。在这个例子中，内部状态就是出发地和目的地，内部状态不会发生变化；外部状态就是铺位和价格，价格会随着铺位的变化而变化。

在 JDK 中 String 也是类似消息池，我们知道在 Java 中 String 是存在于常量池中。也就是说一个 String 被定义之后它就被缓存到了常量池中，当其他地方要使用同样的字符串时，则直接使用的是缓存，而不会重复创建。例如下面这段代码。

```
private static void testString() {
        String str1 = new String("abc");
        String str2 = "abc";
        String str3 = new String("abc");
        String str4 = "ab" + "c";
        // 使用 equals 只判定字符值
        System.out.println(str1.equals(str2));
        System.out.println(str1.equals(str3));
        System.out.println(str3.equals(str2));

        // 等号判等，判定两个对象是不是同一个地址
        System.out.println(str1 == str2);
        System.out.println(str1 == str3);
        System.out.println(str3 == str2);
        System.out.println(str4 == str2);
    }
```

输出如下：

```
true
true
true
false
false
false
true
```

在前 3 个通过 equals 函数判定中，由于它们的字符值都相等，因此 3 个判等都为 true，因此，

String 的 equals 只根据字符值进行判断。而在后 4 个判断中则使用的是两个等号判断，两个等号判断代表的意思是判定这两个对象是否相等，也就是两个对象指向的内存地址是否相等。由于 str1 和 str3 都是通过 new 构建的，而 str2 则是通过字面值赋值的，因此这 3 个判定都为 false，因为它们并不是同一个对象。而 str2 和 str4 都是通过字面值赋值的，也就是直接通过双引号设置的字符串值，因此，最后一个通过“==”判定的值为 true，也就是说 str2 和 str4 是同一个字符串对象。因为 str4 使用了缓存在常量池中的 str2 对象。这就是享元模式在我们开发中的一个重要案例。

22.6 Android 源码中的享元模式

在用 Android 开发了一段时间之后，很多读者就应该知道了一个知识点：UI 不能够在子线程中更新。这原本就是一个伪命题，因为并不是 UI 不可以在子线程更新，而是 UI 不可以在不是它的创建线程里进行更新。只是绝大多数情况下 UI 都是从 UI 线程中创建的，因此，在其他线程更新时会抛出异常。在这种情况下，当我们在子线程完成了耗时操作之后，通常会通过一个 Handler 将结果传递给 UI 线程，然后在 UI 线程中更新相关的视图。代码大致如下。

```
public class MainActvity extends Activity {
    Handler mHandler = new Handler(Looper.getMainLooper()) ;

    private void doSomeThing() {
        new Thread(){
            @Override
            public void run() {
                // 耗时操作,得到结果,但不能在这个线程更新 UI

                // 可以通过 Handler 将结果传递到主线程中,并且更新 UI
                mHandler.post(new Runnable() {
                    @Override
                    public void run() {
                        // 在这里可以更新 UI
                    }
                });
            } // end of run
        };
    }
}
```

在 MainActivity 中首先创建了一个 Handler 对象，它的 Looper 就是 UI 线程的 Looper。在子线程执行完耗时操作之后，则通过 Handler 向 UI 线程传递一个 Runnable，即这个 Runnable 执行在 UI 线程中，然后在这个 Runnable 中更新 UI 即可。

那么 Handler、Looper 的工作原理又是什么呢？它们之间是如何协作的？

在讲此之前我们还需要了解两个概念，即 Message 和 MessageQueue。其实 Android 应用是事件驱动的，每个事件都会转化为一个系统消息，即 Message。消息中包含了事件相关的信息以及这个消息的处理人——Handler。每个进程中都有一个默认的消息队列，也就是我们的 MessageQueue，这个消息队列维护了一个待处理的消息列表，有一个消息循环不断地从这个队列中取出消息、处理消息，这样就使得应用动态地运作起来。它们的运作原理就像工厂的生产线一样，待加工的产品就是 Message，“传送带”就是 MessageQueue，工人们就对应处理事件的 Handler。这么一来 Message

就必然会产生很多对象，因为整个应用都是由事件，也就是 Message 来驱动的，系统需要不断地产生 Message、处理 Message、销毁 Message，难道 Android 没有 iOS 流畅就是这个原因吗？答案显然没有那么简单，重复构建大量的 Message 也不是 Android 的实现方式。那么我们先从 Handler 发送消息开始一步一步学习它的原理。

就用上面的例子来说，我们通过 Handler 传递了一个 Runnable 给 UI 线程。实际上 Runnable 会被包装到一个 Message 对象中，然后再投递到 UI 线程的消息队列。我们看看 Handler 的 post(Runnable run)函数。

```
public final boolean post(Runnable r)
{
   return  sendMessageDelayed(getPostMessage(r), 0);
}

private static Message getPostMessage(Runnable r) {
    Message m = Message.obtain();
    m.callback = r;
    return m;
}
```

在 post 函数中会调用 sendMessageDelayed 函数，但在此之前调用了 getPostMessage 将 Runnable 包装到一个 Message 对象中。然后再将这个 Message 对象传递给 sendMessageDelayed 函数，具体代码如下。

```
    public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
    // 获取到当前 Handler 所在的消息队列
    MessageQueue queue = mQueue;
    if (queue == null) {
        // 消息队列为空,直接返回
        return false;
    }
    // 将消息添加到消息队列中
    return enqueueMessage(queue, msg, uptimeMillis);
}
```

sendMessageDelayed 函数最终又调用了 sendMessageAtTime 函数，我们知道，post 消息时是可以延时发布的，因此，有一个 delay 的时间参数。在 sendMessageAtTime 函数中会判断当前 Handler 的消息队列是否为空，如果不为空那么就会将该消息追加到消息队列中。又因为我们的 Handler 在创建时就关联了 UI 线程的 Looper（如果不手动传递 Looper 那么 Handler 持有的 Looper 就是当前线程的 Looper，也就是说在哪个线程创建的 Handler，就是哪个线程的 Looper），Handler 从这个 Looper 中获取消息队列，这样一来 Runnable 就会被放到 UI 线程的消息队列了，因此，我们的 Runnable 在后续的某个时刻就会被执行在 UI 线程中。

这里我们不需要再深究 Handler、Looper 等角色的运作细节，我们这里关注的是享元模式的运

用。在上面的 getPostMessage 中会将 Runnable 包装为一个 Message，在前文没有说过，系统并不会构建大量的 Message 对象，那么它是如何处理的呢？

我们看到在 getPostMessage 中的 Message 对象是从一个 Message.obtain()函数返回的，并不是使用 new 来实现，如果使用 new 那么就是我们起初猜测的会构建大量的 Message 对象，当然到目前还不能下结论，我们看看 Message.obtain()的实现。

```
    public static Message obtain() {
        synchronized (sPoolSync) {
            if (sPool != null) {
                Message m = sPool;
                sPool = m.next;
                m.next = null;
                m.flags = 0; // 清空 in-use flag
                sPoolSize--;
                return m;
            }
        }
    return new Message();
}
```

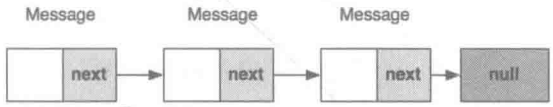
实现很简单，但是有一个很引人注意的关键词——Pool，它的中文意思称为池，难道是我们前文所说的共享对象池？目前我们依然不能确定，但是，此时已经看到了一些重要线索。现在就来看看 obtain 中的 sPoolSync、sPool 里是些什么程序。

```
/**
* Defines a message containing a description and arbitrary data object that can be
 * sent to a {@link Handler}. This object contains two extra int fields and an
 * extra object field that allow you to not do allocations in many cases.
 *
 * <p class="note">While the constructor of Message is public, the best way to get
 * one of these is to call {@link #obtain Message.obtain()} or one of the
 * {@link Handler#obtainMessage Handler.obtainMessage()} methods, which will pull
 * them from a pool of recycled objects.</p>
 */
public final class Message implements Parcelable {
    // 字段省略
    private static final Object sPoolSync = new Object();
    private static Message sPool;
    private static int sPoolSize = 0;

    // 代码省略
}
```

首先 Message 文档第一段的意思就是介绍了一下这个 Message 类的字段，以及说明 Message 对象是被发送到 Handler 的，对于我们来说作用不大。第二段的意思是建议我们使用 Message 的 obtain 方法获取 Message 对象，而不是通过 Message 的构造函数，因为 obtain 方法会从被回收的对象池中获取 Message 对象。然后再看看关键的字段，sPoolSync 是一个普通的 Object 对象，它的作用就是用于在获取 Message 对象时进行同步锁。再看 sPool 居然是一个 Message 对象，居然不是我们上面说的消息池之类的东西，既然它命名为 sPool 不可能是有名无实吧，我们再仔细看，发现了这个字段。

```
    // sometimes we store linked lists of these things
Message next;
```

这个字段就在 sPoolSync 上面，“山重水复疑无路，柳暗花明又一村”。一看上面的注释我们就明白了，原来 Message 消息池没有使用 map 这样的容器，使用的是链表！这个 next 就是指向下一个 Message 的。Message 的链表如图 22-2 所示。



▲图 22-2

每个 Message 对象都有一个同类型的 next 字段，这个 next 指向的就是下一个可用的 Message，最后一个可用的 Message 的 next 则为空。这样一来，所有可用的 Message 对象就通过 next 串连成一个可用的 Message 池。

那么这些Message对象什么时候会被放到链表中呢？我们在obtain函数中只看到了从链表中获取，并且看到存储。如果消息池链表中没有可用对象的时候，obtain 中则是直接返回一个通过 new 创建的 Message 对象，而且并没有存储到链表中。此时，我们再次遇到了难点，暂时找不到相关线索了。

此时我们只好回过头再看看 Message 类的说明，发现一个重要的句子。

“which will pull them from a pool of recycled objects.”，噢，原来在创建的时候不会把 Message 对象放到池中，在回收（这里的回收并不是指虚拟机回收 Message 对象）该对象时才会将该对象添加到链表中。

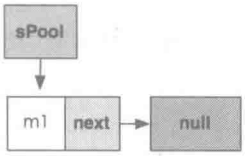
我们搜索一番之后果然发现了 Message 类中有类似 Bitmap 那样的 recycle 函数。具体代码如下。

```
/**
 * 将 Message 对象回收到消息池中
 */
public void recycle() {
    // 判断是否该消息还在使用
    if (isInUse()) {
        if (gCheckRecycle) {
            throw new IllegalStateException("This message cannot be recycled because it "
                    + "is still in use.");
        }
        return;
    }
    // 清空状态，并且将消息添加到消息池中
    recycleUnchecked();
}

void recycleUnchecked() {
    // 清空消息状态，设置该消息 in-use flag
    flags = FLAG_IN_USE;
    what = 0;
    arg1 = 0;
    arg2 = 0;
    obj = null;
    replyTo = null;
    sendingUid = -1;
    when = 0;
    target = null;
    callback = null;
```

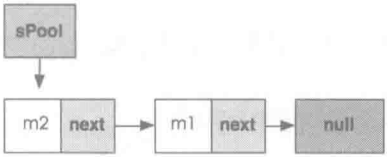
```
        data = null;
        // 回收消息到消息池中
        synchronized (sPoolSync) {
            if (sPoolSize < MAX_POOL_SIZE) {
                next = sPool;
                sPool = this;
                sPoolSize++;
            }
        }
    }
```

recycle 函数会将一个 Message 对象回收到一个全局的池中，这个池也就是我们上文说的链表。recycle 函数首先判断该消息是否还在使用，如果还在使用则抛出异常，否则调用 recycleUnchecked 函数处理该消息。recycleUnchecked 函数中先清空该消息的各字段，并且将 flags 设置为 FLAG_IN_USE，表明该消息已被使用，这个 flag 在 obtain 函数中会被置为 0，这样根据这个 flag 就能够追踪该消息的状态。然后判断是否要将该消息回收到消息池中，如果池的大小小于 MAX_POOL_SIZE 时，将自身添加到链表的表头。例如，当链表中还没有元素时，将第一个 Message 对象添加到链表中，此时 sPool 为 null,next 指向了 sPool，因此，next 也为 null，然后 sPool 又指向了 this，因此，sPool 就指向了当前这个被回收的对象，并且 sPoolSize 加 1。我们把这个被回收的 Message 对象命名为 m1，此时结构图如图 22-3 所示。

此时如果再插入一个名称为 m2 的 Message 对象，那么 m2 将会被插到表头中，此时 sPool 指向的就是 m2，结构如图 22-4 所示。



▲图 22-3



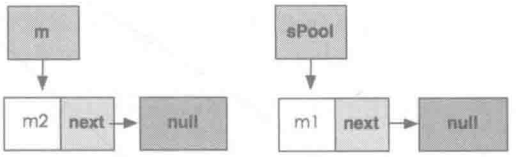
▲图 22-4

这个对象池的大小默认为 50，因此，如果池大小在小于 50 的情况下，被回收的 Message 就会被插到链表头部。

此时如果池中有元素，当我们调用 obtain 函数时，如果池中有元素就会从池中获取，实际上获取的也是表头元素，也就是这里的 sPool。然后再将 sPool 这个指针后移到下一个元素。具体代码如下。

```
public static Message obtain() {
        synchronized (sPoolSync) {
            if (sPool != null) {
                Message m = sPool;
                sPool = m.next;
                m.next = null;
                m.flags = 0; // 清空 in-use flag
                sPoolSize--;
                return m;
            }
        }
        return new Message();
    }
```

在 obtain 函数中，首先会声明一个 Message 对象 m，并且让 m 指向 sPool。sPool 实际上指向了 m2，因此，m 实际上指向的也是 m2，这里相当于保存了 m2 这个元素。下一步是 sPool 指向 m2 的下一个元素，也就是 m1。sPool 也完成后移之后此时把 m.next 置空，也就相当于 m2.next 变成了 null。最后就是 m 指向了 m2 元素，m2 的 next 为空，sPool 从原来的表头 m2 指向了下一个元素 m1，最后将对象池的元素减 1，这样 m2 就顺利地脱离了消息池队伍，返回给了调用 obtain 函数的客户端程序。此时结构如图 22-5 所示。



▲图 22-5

现在已经很明朗了，Message 通过在内部构建一个链表来维护一个被回收的 Message 对象的对象池，当用户调用 obtain 函数时会优先从池中取，如果池中没有可以复用的对象则创建这个新的 Message 对象。这些新创建的 Message 对象在被使用完之后会被回收到这个对象池中，当下次再调用 obtain 函数时，它们就会被复用。这里的 Message 相当于承担了享元模式中 3 个元素的职责，即是 Flyweight 抽象，又是 ConcreteFlyweight 角色，同时又承担了 FlyweightFactory 管理对象池的职责。因为 Android 应用是事件驱动的，因此，如果通过 new 创建 Message 就会创建大量重复的 Message 对象，导致内存占用率高、频繁 GC 等问题，通过享元模式创建一个大小为 50 的消息池，避免了上述问题的产生，使得这些问题迎刃而解。当然，这里的享元模式并不是经典的实现方式，它没有内部、外部状态，集各个职责于一身，甚至它更像是一个对象池，但这些都是很细节问题，我们关注的是灵活运用模式本身来解决问题。至于 Message 对象是否职责过多，既是实体类又是工厂类，这些问题每个人见仁见智，也许你觉得增加一个 MessagePool 来管理 Message 对象的回收、获取工作不会更好，这样也满足了单一职责原则；或者你觉得就这样用就挺好，没有必要增加管理类。这些我们不过多评论，原则上只是提供了一个可借鉴的规则，这个规则很多时候并不是一成不变的，可以根据实际场景进行取舍。规则是使读者避免走向软件大泥潭，灵活运用才是最终的目的所在。

22.7 深度拓展

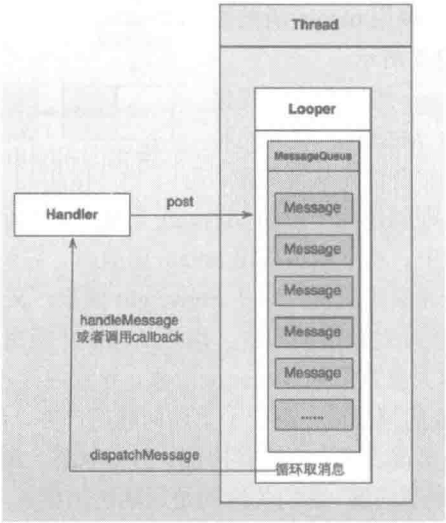
22.7.1 深入了解 Android 的消息机制

上文我们说到，Message、MessageQueue、Looper、Handler 的工作原理像是工厂的生产线，Looper 就是发动机，MessageQueue 就是传送带，Handler 就是工人，Message 则是待处理的产品。它们的结构图如图 22-6 所示。

前面的章节中我们多次提到 Android 应用程序的入口实际上是 ActivityThread.main 方法，在该方法中首先会创建 Application 和默认启动的 Activity，并且将它们关联在一起。而该应用的 UI 线程的消息循环也是在这个方法中创建的，具体源码如下。

```
public static void main(String[] args) {
        // 代码省略
        Process.setArgV0("<pre-initialized>");
        // 1. 创建消息循环 Looper,就是 UI 线程的消息队列
        Looper.prepareMainLooper();
```

```
        // 启动 ActivityThread，这里最终会启动应用程序
    ActivityThread thread = new ActivityThread();
    thread.attach(false);
    AsyncTask.init();
    Looper.loop();    // 2. 执行消息循环
}
```



▲图 22-6

执行 ActivityThread.main 方法后，应用程序就启动了，UI 线程的消息循环也在 Looper.loop() 函数中启动。此后 Looper 会一直从消息队列中取消息，然后处理消息。用户或者系统通过 Handler 不断地往消息队列中添加消息，这些消息不断地被取出、处理、回收，使得应用迅速地运转起来。

例如，我们在子线程中执行完耗时操作后通常需要更新 UI，但我们都“知道”不能在子线程中更新 UI。此时最常用的手段就是通过 Handler 将一个消息 post 到 UI 线程中，然后再在 Handler 的 handleMessage 方法中进行处理。但是有一点要注意，如果用在不传递 UI 线程所属的 Looper 的情况下，那么该 Handler 必须在主线程中创建！正确地使用示例如下。

```
class MyHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        // 更新 UI
    }
}

// 在 UI 线程中创建
MyHandler mHandler = new MyHandler() ;
// 开启新的线程
new Thread(){
    public void run() {
        // 耗时操作
        mHandler.sendEmptyMessage(123) ;
    };
}.start();
```

为什么必须要这么做呢？

其实每个 Handler 都会关联一个消息队列，消息队列被封装在 Lopper 中，而每个 Looper 又是 ThreadLocal 的，也就是说每个消息队列只会属于一个线程。因此，如果一个 Looper 在线程 A 中创建，那么该 Looper 只能够被线程 A 访问。而 Handler 则是一个消息投递、处理器，它将消息投递给消息队列，然后这些消息在消息队列中被取出，并且执行在关联了该消息队列的线程中。默认情况下，消息队列只有一个，即主线程的消息队列，这个消息队列是在 ActivityThread.main 方法中创建的，也就调用了 Lopper.prepareMainLooper()方法，创建 Looper 之后，最后会执行 Looper.loop()来启动消息循环。那么 Handler 是如何关联消息队列以及线程呢？我们还是深入源码来分析，首先看看 Handler 的构造函数。

```
public Handler() {
    // 代码省略
    mLooper = Looper.myLooper();    // 获取 Looper
    mQueue = mLooper.mQueue;        // 获取消息队列
    mCallback = null;
}
```

从 Handler 默认的构造函数中我们可以看到，Handler 会在内部通过 Looper.getLooper()来获取 Looper 对象，并且与之关联，最重要的就是获取到 Looper 持有的消息队列 mQueue。那么 Looper.getLooper()又是如何工作的呢？我们继续往下看。

```
public static Looper myLooper() {
    return sThreadLocal.get();
}

public static void prepareMainLooper() {
    prepare();
    setMainLooper(myLooper());
    myLooper().mQueue.mQuitAllowed = false;
}

private synchronized static void setMainLooper(Looper looper) {
    mMainLooper = looper;
}

public static void prepare() {
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    sThreadLocal.set(new Looper());
}
```

我们看到 myLooper()方法是通过 sThreadLocal.get()来获取的，关于 ThreadLocal 的资料请参考 Java 相关的书籍。那么 Looper 对象又是什么时候存储在 sThreadLocal 中的呢？有些读者可能看到了，上面给出的代码中给出了一个熟悉的方法 prepareMainLooper()，在这个方法中调用了 prepare()方法，在 prepare()方法中创建了一个 Looper 对象，并且将该对象设置给了 sThreadLocal。这样，队列就与线程关联上了。

我们再回到 Handler 中来，Looper 属于某个线程，消息队列存储在 Looper 中，因此，消息队列就通过 Looper 与特定的线程关联上。而 Handler 又与 Looper、消息队列关联，因此，Handler 最

终就和线程、线程的消息队列关联上了，通过该 Handler 发送的消息最终就会被执行在这个线程上。这就能解释上面提到的问题了，“在不传递 Looper 参数给 Handler 构造函数的情况下，用更新 UI 的 Handler 为什么必须在 UI 线程中创建？”。就是因为 Handler 要与主线程的消息队列关联上，这样 handleMessage 才会执行在 UI 线程中，更新 UI 才是被允许的。

创建了 Looper 后，会调用 Looper 的 loop 函数，在这个函数中会不断地从消息队列中取出、处理消息，具体源码如下。

```
public static void loop() {
    Looper me = myLooper();
    // 代码省略
    MessageQueue queue = me.mQueue;    // 1. 获取消息队列
    // 代码省略
    while (true) {        // 2. 死循环，即消息循环
        // 3. 获取消息 (might block )
        Message msg = queue.next();
        if (msg != null) {
            if (msg.target == null) {
                return;
            }
            // 代码省略
            msg.target.dispatchMessage(msg);    // 4. 处理消息
// 5. 回收消息，也就是我们分析享元模式时提到的将 Message 添加到消息池的操作
            msg.recycle();
        }
    } // end while
}
```

从上述程序可以看到，loop 方法中实质上就是建立一个死循环，然后通过从消息队列中逐个取出消息，最后就是处理消息、回收消息的过程。

在注释 3 处，调用了 MessageQueue 的 next 函数来获取下一条要处理的消息。这个 MessageQueue 在 Looper 的构造函数中构建，我们看看 next 函数的核心代码。

```
Message next() {
    for (;;) {
        // 1. 处理 Native 层的事件
        nativePollOnce(mPtr, nextPollTimeoutMillis);
        synchronized (this) {
            // Try to retrieve the next message.  Return if found.
            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            // 2. Java 层的消息队列
            Message msg = mMessages;
            // 代码省略
            if (msg != null) {
                // msg 这个消息有延迟，因此做一个延迟处理
                if (now < msg.when) {
                    // Next message is not ready.  Set a timeout to wake up when it is ready.
                    nextPollTimeoutMillis=(int)Math.min(msg.when- now, Integer.MAX_VALUE);
                } else {
                    // Got a message.
                    mBlocked = false;
                    if (prevMsg != null) {
                        prevMsg.next = msg.next;
                    } else {
                        mMessages = msg.next;
```

```
                }
                msg.next = null;
                msg.markInUse();
                // 3. 返回消息
                return msg;
            }
        } else {
            // No more messages.
            nextPollTimeoutMillis = -1;
        }
    }
    // 代码省略
} // end for
}
```

完整的 next 函数稍微有些复杂，但这里只分析核心的程序。next 函数的基本思路就是从消息队列中依次取出消息，如果这个消息到了执行时间，那么就将这条消息返回给 Looper，并且将消息队列链表的指针后移。这个消息队列链表结构与 Message 中的消息池结构一致，也是通过 Message 的 next 字段将多个 Message 对象串连在一起。但是在从消息队列获取消息之前，还有一个 nativePollOnce 函数的调用，第一个参数为 mPtr，第二个参数为超时时间。

这与 Native 层有什么关系？mPtr 又是什么？

其实这个 mPtr 可是大有来头，它存储了 Native 层的消息队列对象，也就是说 Native 层还有一个 MessageQueue 类型。mPtr 的初始化是在 MessageQueue 的构造函数中，具体代码如下。

```
public final class MessageQueue {
    private int mPtr; // used by native code
    // 消息队列构造
    MessageQueue(boolean quitAllowed) {
        mQuitAllowed = quitAllowed;
        mPtr = nativeInit();
    }
}
```

可以看到，mPtr 的值是 nativeInit 函数返回的，该函数在 android_os_MessageQueue.cpp 类中，我们继续跟踪代码。

```
static jint android_os_MessageQueue_nativeInit(JNIEnv* env, jclass clazz) {
    // 1. 构造 NativeMessageQueue
    NativeMessageQueue* nativeMessageQueue = new NativeMessageQueue();
    if (!nativeMessageQueue) {
        jniThrowRuntimeException(env, "Unable to allocate native queue");
        return 0;
    }

    nativeMessageQueue->incStrong(env);
    // 2. 将 NativeMessageQueue 对象转为一个整型变量
    return reinterpret_cast<jint>(nativeMessageQueue);
}
```

我们看到，在 nativeInit 函数中会构造一个 NativeMessageQueue 对象，然后将该对象转为一个整型值，并且返回给 Java 层中，而当 Java 层需要与 Native 层的 MessageQueue 通信时只要把这个 int 值传递给 Native 层，然后 Native 通过 reinterpret_cast 将传递进来的 int 转换为 NativeMessageQueue 指针即可得到这个 NativeMessageQueue 对象指针。首先看看 NativeMessageQueue 类的构造函数。

```
NativeMessageQueue::NativeMessageQueue() : mInCallback(false), mExceptionObj(NULL) {
    mLooper = Looper::getForThread();
    if (mLooper == NULL) {
        mLooper = new Looper(false);
        Looper::setForThread(mLooper);
    }
}
```

代码很简单，就是创建了一个 Native 层的 Looper，然后这个 Looper 设置给了当前线程。也就是说 Java 层的 MessageQueue 和 Looper 在 Native 层也都有，但是，它们功能并不是一一对应的。那么看看 Looper 究竟做了什么，首先看看它的构造函数，代码在 system/core/libutils/Looper.cpp 文件中。

```
Looper::Looper(bool allowNonCallbacks) :
        mAllowNonCallbacks(allowNonCallbacks), mSendingMessage(false),
        mResponseIndex(0), mNextMessageUptime(LLONG_MAX) {
    int wakeFds[2];
    // 1. 创建管道
    int result = pipe(wakeFds);
    // 管道的读、写端
    mWakeReadPipeFd = wakeFds[0];
    mWakeWritePipeFd = wakeFds[1];

    // 2. 创建 epoll 文件描述符
    mEpollFd = epoll_create(EPOLL_SIZE_HINT);

    struct epoll_event eventItem;
    memset(& eventItem,0,sizeof(epoll_event));//zero out unused members of data field union
    // 设置事件类型和文件描述符
    eventItem.events = EPOLLIN;
    eventItem.data.fd = mWakeReadPipeFd;
    // 3. 监听事件
    result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeReadPipeFd, & eventItem);
}
```

首先创建了一个管道（pipe），管道本质上就是一个文件，一个管道中含有两个文件描述符，分别对应读和写。一般的使用方式是一个线程通过读文件描述符来读管道的内容，当管道没有内容时，这个线程就会进入等待状态；而另外一个线程通过写文件描述符来向管道中写入内容，写入内容的时候，如果另一端正有线程正在等待管道中的内容，那么这个线程就会被唤醒。这个等待和唤醒的操作是通过 Linux 系统的 epoll 机制。要使用 Linux 系统的 epoll 机制，首先要通过 epoll_create 来创建一个 epoll 专用的文件描述符，即注释 2 的代码。最后通过 epoll_ctl 函数设置监听的事件类型为 EPOLLIN。此时 Native 层的 MessageQueue 和 Looper 就构建完毕了，在底层也通过管道和 epoll 建立了一套消息机制。Native 层构建完毕之后则会返回到 Java 层 Looper 的构造函数，因此，Java 层的 Looper 和 MessageQueue 也构建完毕。

这个过程有点绕，我们总结一下。

（1）首先构造 Java 层的 Looper 对象，Looper 对象又会在构造函数中创建 Java 层的 MessageQueue 对象。

（2）Java 层的 MessageQueue 的构造函数中调用 nativeInit 函数初始化 Native 层的 NativeMessageQueue，NativeMessageQueue 的构造函数又会创建 Native 层的 Looper，并且通过管道和 epoll 建立一套消息机制。

（3）Native 层构建完毕，将 NativeMessageQueue 对象转换为一个整型存储到 Java 层的 MessageQueue 的 mPtr 中。

（4）启动 Java 层的消息循环，不断地读取、处理消息。

这个初始化过程都是在 ActivityThread 的 main 函数中完成的，因此，main 函数运行之后，UI 线程消息循环就启动了，消息循环不断地从消息队列中读取、处理消息，使得系统运转起来。

我们继续回到 nativePollOnce 函数本身，每次循环去读消息时都会调用这个函数，我们看看它到底做了些什么？代码在 android_os_MessageQueue.cpp 中。

```
    static void android_os_MessageQueue_nativePollOnce(JNIEnv* env, jclass clazz,
        jint ptr, jint timeoutMillis) {
    NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
    nativeMessageQueue->pollOnce(env, timeoutMillis);
}
```

首先将传递进来的整型转换为 NativeMessageQueue 指针，这个整型就是在初始化时保存到 mPtr 的数值。然后调用了 NativeMessageQueue 的 pollOnce 函数。具体代码如下。

```
// NativeMessageQueue
void NativeMessageQueue::pollOnce(JNIEnv* env, int timeoutMillis) {
// 调用了 Native 层 Looper 的 pollOnce 函数
    mLooper->pollOnce(timeoutMillis);
    // 代码省略
}
```

这里的代码很简单，调用了 Native 层 Looper 的 pollOnce 函数。Native 层 Looper 类的完整路径是 system/core/libutils/Looper.cpp，pollOnce 函数如下。

```
int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** outData) {
    int result = 0;
    for (;;) {
        // 代码省略
        result = pollInner(timeoutMillis);
    }
}
```

该函数的核心在于调用了 pollInner，我们看看 pollInner 的相关实现。

```
// pollInner 函数
int Looper::pollInner(int timeoutMillis) {
    // 代码省略
    struct epoll_event eventItems[EPOLL_MAX_EVENTS];
    // 1. 从管道中读取事件
    int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS, timeoutMillis);
    // 获取锁
    mLock.lock();
    // 代码省略

Done: ;

    // Invoke pending message callbacks.
    mNextMessageUptime = LLONG_MAX;
    while (mMessageEnvelopes.size() != 0) {
        nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
        const MessageEnvelope& messageEnvelope = mMessageEnvelopes.itemAt(0);
        // 判断执行时间
```

```
        if (messageEnvelope.uptime <= now) {
            { // obtain handler
                sp<MessageHandler> handler = messageEnvelope.handler;
                Message message = messageEnvelope.message;
                mMessageEnvelopes.removeAt(0);
                mSendingMessage = true;
                mLock.unlock();
                // 处理消息
                handler->handleMessage(message);
            } // release handler

            mLock.lock();
            mSendingMessage = false;
            result = ALOOPER_POLL_CALLBACK;
        } else {
            // The last message left at the head of the queue determines the next wakeup time.
            mNextMessageUptime = messageEnvelope.uptime;
            break;
        }
    }

    // Release lock.
    mLock.unlock();
    // 代码省略
    return result;
}
```

从 pollInner 的核心代码中看，pollInner 实际上就是从管道中读取事件，并且处理这些事件。这样一来就相当于在 Native 层存在一个独立的消息机制，这些事件存储在管道中，而 Java 层的事件则存储在消息链表中。但这两个层次的事件都在 Java 层的 Looper 消息循环中进行不断地获取、处理等操作，从而实现程序的运转。但需要注意的是，Native 层的 NativeMessageQueue 实际上只是一个代理 Native Looper 的角色，它没有做什么实际工作，只是把操作转发给 Looper。而 Native Looper 则扮演了一个 Java 层的 Handler 角色，它能够发送消息、取消息、处理消息。

那么 Android 为什么要有两套消息机制呢？我们知道 Android 是支持纯 Native 开发的，因此，在 Native 层实现一套消息机制是必须的。另外，Android 系统的核心组件也都是运行在 Native 世界，各组件之间也需要通信，这样一来 Native 层的消息机制就变得很重要。

在分析了消息循环与消息队列的基本原理之后，最后看看消息处理逻辑。我们看到在 Java 层的 MessageQueue 的 next 函数的第 4 步调用了 msg.target.dispatchMessage(msg)来处理消息。其中 msg 是 Message 类型，我们看看源码。

```
public final class Message implements Parcelable {
    // 其他字段
    Handler target;          // target 处理
    Runnable callback;       // Runnable 类型的 callback
    // 代码省略
}
```

从源码中可以看到，target 是 Handler 类型。实际上就是转了一圈，通过 Handler 将消息传递给消息队列，消息队列又将消息分发给 Handler 来处理，其实这也是一个典型的命令模式，Message 就是一条命令，Handler 就是处理人，通过命令模式将操作和执行者解耦。我们继续回到 Handler 代码中，消息处理是调用了 Handler 的 dispatchMessage 方法，相关代码如下。

```
/**
 * Handler 子类必须实现这个方法来处理消息
 */
public void handleMessage(Message msg) {
}

private final void handleCallback(Message message) {
    message.callback.run();
}

public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}
```

从上述程序中可以看到，dispatchMessage 只是一个分发的方法，如果 Runnable 类型的 callback 为空则执行 handlerMessage 来处理消息，该方法为空，我们会将更新 UI 的代码写在该函数中；如果 callback 不为空，则执行 handleCallback 来处理，该方法会调用 callback 的 run 方法。其实这是 Handler 分发的两种类型，比如我们 post(Runnable callback)则 callback 就不为空，此时就会执行 Runnable 的 run 函数；当我们使用 Handler 来 sendMessage 时通常不会设置 callback，因此，也就执行 handlerMessage 这个分支。下面我们看看通过 Handler 来 post 一个 Runnable 对象的实现代码。

```
public final boolean post(Runnable r){
    return  sendMessageDelayed(getPostMessage(r), 0);
}
private final Message getPostMessage(Runnable r) {
    Message m = Message.obtain();
    m.callback = r;  // 设置了 callback
    return m;
}

public final boolean sendMessageDelayed(Message msg, long delayMillis) {
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}

public boolean sendMessageAtTime(Message msg, long uptimeMillis)
{
    boolean sent = false;
    MessageQueue queue = mQueue;
    if (queue != null) {
        msg.target = this; // 设置消息的 target 为当前 Handler 对象
            // 将消息插入到消息队列
        sent = queue.enqueueMessage(msg, uptimeMillis);
    }
    else {
        // 代码省略
    }
```

```
    return sent;
}
```

从上述程序可以看到，在 post(Runnable r)时，会将 Runnable 包装成 Message 对象，并且将 Runnable 对象设置给 Message 对象的 callback 字段，最后会将该 Message 对象插入消息队列。sendMessage 也是类似实现。

```
public final boolean sendMessage(Message msg) {
    return sendMessageDelayed(msg, 0);
}
```

因此不管是 post 一个 Runnbale 还是 Message，都会调用 sendMessageDelayed(msg, time)方法，然后该 Message 就会追加到消息队列中，当在消息队列中取出该消息时就会调用 callback 的 run 方法或者 Handler 的 handleMessage 来执行相应的操作。

最后总结一下就是消息通过 Handler 投递到消息队列，这个消息队列在 Handler 关联的 Looper 中，消息循环启动之后会不断地从队列中获取消息，其中消息的处理分为 Native 层和 Java 层，两个层次都有自己的消息机制，Native 层基于管道和 epoll，而 Java 层则是一个普通的链表。获取消息之后会调用消息的 callback 或者分发给对应 Handler 的 handleMessage 函数进行处理，这样就将消息、消息的分发、处理隔离开来，降低各个角色之间的耦合。消息被处理之后会被收回到消息池中便于下次利用，这样整个应用通过不断地执行这个流程就运转起来了。

22.7.2　子线程中创建 Handler 为何会抛出异常

先给一段程序。

```
new Thread(){
    Handler handler = null;
    public void run() {
        handler = new Handler();
    };
}.start();
```

上面的代码有问题吗？

如果你能够发现并且解释上述代码的问题，那么应该说您对 Handler、Looper、Thread 这几个概念已经很了解了。如果您还不太清楚，那么我们一起往下学习。

前面说过，Looper 对象是 ThreadLocal 的，即每个线程都有自己的 Looper，这个 Looper 可以为空。但是当你要在子线程中创建 Handler 对象时，如果 Looper 为空，那么就会抛出“Can't create handler inside thread that has not called Looper.prepare()”异常，为什么会这样呢？我们一起看源码。

```
public Handler() {
    // 代码省略
    mLooper = Looper.myLooper();     // 获取 myLooper
    if (mLooper == null) {
        throw new RuntimeException(

"Can't create handler inside thread that has not called Looper.prepare()");// 抛出异常
    }
    mQueue = mLooper.mQueue;
    mCallback = null;
}
```

从上述程序中我们可以看到，当 mLooper 对象为空时，抛出了该异常。这是因为该线程中的 Looper 对象还没有创建，因此，sThreadLocal.get()会返回 null。我们知道 Looper 是使用 ThreadLocal 存储的，也就是说它是和线程关联的，在子线程中没有手动调用 Looper.prepare 之前该线程的 Looper 就为空。因此，解决方法就是在构造 Handler 之前为当前线程设置 Looper 对象，解决方法如下。

```
new Thread(){
    Handler handler = null;
    public void run() {
        Looper.prepare(); // 1. 创建 Looper,并且会绑定到 ThreadLocal 中
        handler = new Handler();
        Looper.loop();  // 2. 启动消息循环
    };
}.start();
```

在代码中我们增加了 2 处，第一是通过 Looper.prepare()来创建 Looper，第二是通过 Looper.loop()来启动消息循环。这样该线程就有了自己的 Looper，也就是有了自己的消息队列。如果只创建 Looper，而不启动消息循环，虽然不会抛出异常，但是你通过 handler 来 post 或者 sendMessage 也不会有效，因为虽然消息被追加到消息队列了，但是并没有启动消息循环，也就不会从消息队列中获取消息并且执行！

在应用启动时，会开启一个主线程（UI 线程），并且启动消息循环，应用不停地从该消息队列中取出、处理消息达到程序运行的效果。Looper 对象封装了消息队列，Looper 对象被封装在 ThreadLocal 中，这使得不同线程之间的 Looper 不能被共享。而 Handler 通过与 Looper 对象绑定来实现与执行线程的绑定，handler 会把 Runnable（包装成 Message）或者 Message 对象追加到与线程关联的消息队列中，然后在消息循环中逐个取出消息，并且处理消息。当 Handler 绑定的 Looper 是主线程的 Looper，则该 Handler 可以在 handleMessage 中更新 UI，否则更新 UI 则会抛出异常。

22.8 总结

享元模式实现比较简单，但是它的作用在某些场景确实极其重要的。它可以大大减少应用程序创建的对象，降低程序内存的占用，增强程序的性能，但它同时也提高了系统的复杂性，需要分离出外部状态和内部状态，而且外部状态具有固化特性，不应该随内部状态改变而改变，否则导致系统的逻辑混乱。

享元模式的优点在于它大幅度地降低内存中对象的数量。但是，它做到这一点所付出的代价也是很高的。

- 享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。
- 享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

第 23 章　统一编程接口——外观模式

23.1 外观模式介绍

外观模式（Facade）在开发过程中的运用频率非常高，尤其是在现阶段各种第三方 SDK 充斥在我们的周边，而这些 SDK 很大概率会使用外观模式。通过一个外观类使得整个系统的接口只有一个统一的高层接口，这样能够降低用户的使用成本，也对用户屏蔽了很多实现细节。当然，在我们的开发过程中，外观模式也是我们封装 API 的常用手段，例如网络模块、ImageLoader 模块等。可能你已经在开发中运用过无数次外观模式，只是没有在理论层面认识它，本章我们就从理论结合实践的角度学习这个模式。

23.2 外观模式定义

要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式（Facade 模式）提供一个高层次的接口，使得子系统更易于使用。

23.3 外观模式的使用场景

（1）为一个复杂子系统提供一个简单接口。子系统往往因为不断演化而变得越来越复杂，甚至可能被替换。大多数模式使用时都会产生更多、更小的类，在这使子系统更具可重用性的同时也更容易对子系统进行定制、修改，这种易变性使得隐藏子系统的具体实现变得尤为重要。Facade 可以提供一个简单统一的接口，对外隐藏子系统的具体实现、隔离变化；

（2）当你需要构建一个层次结构的子系统时，使用 Facade 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，你可以让它们仅通过 Façade 接口进行通信，从而简化了它们之间的依赖关系。

23.4 外观模式的 UML 类图

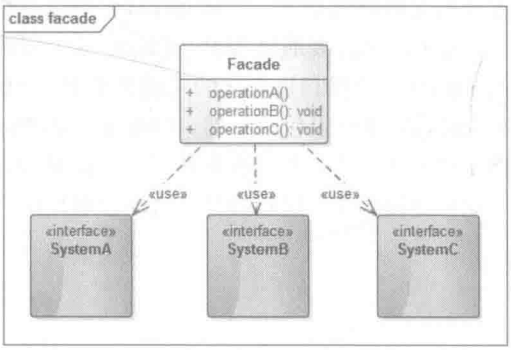
UML 类图如图 23-1 所示。

角色介绍。

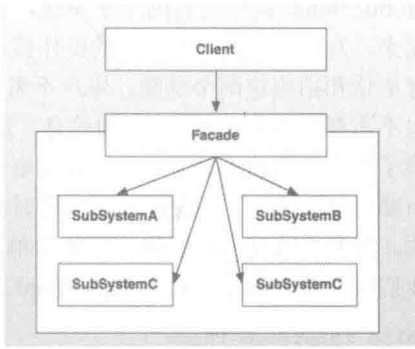
Façade：系统对外的统一接口，系统内部系统地工作。

SystemA、SystemB、SystemC：子系统接口（实现部分未在图 23-1 中给出）。

外观模式接口比较简单，就是通过一个统一的接口对外提供服务，使得外部程序只通过一个类就可以实现系统内部的多种功能，而这些实现功能的内部子系统之间可能也有交互，或者说完成一个功能需要几个子系统之间进行协作，如果没有封装，那么用户就需要操作几个子系统的交互逻辑，容易出现错误。而通过外观类来对外屏蔽这些复杂的交互，降低用户的使用成本。它的结构如图 23-2 所示。



▲图 23-1



▲图 23-2

23.5 外观模式的简单示例

生活中使用外观模式的例子非常多，任何一个类似中央调度结构的组织都类似外观模式。举个简单的例子，手机就是一个外观模式的例子，它集合了电话功能、短信功能、GPS、拍照等于一身，通过手机你就可以完成各种功能。而不是当你打电话时使用一个诺基亚 1100，要拍照时非得用一个相机，如果是这样每使用一个功能你就必须操作特定的设备，会使得整个过程很繁琐。而手机给了你一个统一的入口，集电话、上网、拍照等功能于一身，使用方便，操作简单。结构图如图 23-3 所示。

下面我们来简单模拟一下手机的外观模式实现，首先我们建立一个 MobilePhone 类。代码大致如下。

```
public class MobilePhone {
    private Phone mPhone = new PhoneImpl();
    private Camera mCamera = new SamsungCamera();

    public void dail() {
        mPhone.dail();
    }

    public void videoChat() {
        System.out.println("--> 视频聊天接通中");
        mCamera.open();
        mPhone.dail();
    }
```

▲图 23-3

```
    public void hangup() {
        mPhone.hangup();
    }

    public void takePicture() {
        mCamera.open();
        mCamera.takePicture();
    }

    public void closeCamera() {
        mCamera.close();
    }
}
```

MobilePhone 类中含有两个子系统，也就是拨号系统和拍照系统，MobilePhone 将这两个系统封装起来，为用户提供一个统一的操作接口，也就是说用户只需要通过 MobilePhone 这个类就可以操作打电话和拍照这两个功能。用户不需要知道有 Phone 这个接口以及它的实现类是 PhoneImpl，同样也不需要知道 Camera 相关的信息，通过 MobilePhone 就可以包揽一切。而在 MobilePhone 中也封装了两个子系统的交互，例如视频电话时需要先打开摄像头，然后再开始拨号，如果没有这一步的封装，每次用户实现视频通话功能时都需要手动打开摄像头、进行拨号，这样会增加用户的使用成本，外观模式使得这些操作更加简单、易用。

我们来看看 Phone 接口和 PhoneImpl。

```
public interface Phone {
    // 打电话
    public void dail();
    // 挂断
    public void hangup();
}

class PhoneImpl implements Phone {
    @Override
    public void dail() {
        System.out.println("打电话");
    }

    @Override
    public void hangup() {
        System.out.println("挂断");
    }
}
```

代码很简单，就是单纯的抽象与实现。Camera 也是类似的实现，具体代码如下。

```
public interface Camera {
    public void open();
    public void takePicture();
    public void close();
}

class SamsungCamera implements Camera {

    @Override
    public void open() {
        System.out.println("打开相机");
    }
```

```
    @Override
    public void takePicture() {
        System.out.println("拍照");
    }

    @Override
    public void close() {
        System.out.println("关闭相机");
    }
}
```

测试代码。

```
public class Test {
    public static void main(String[] args) {
        MobilePhone nexus6 = new MobilePhone();
        // 拍照
        nexus6.takePicture();
        // 视频聊天
        nexus6.videoChat();
    }
}
```

运行结果：

```
打开相机
拍照
打开相机
打电话
```

从上述代码中可以看到，外观模式就是统一接口封装。将子系统的逻辑、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，同时也对外隐藏了具体的实现，这样即使具体的子系统发生了变化，用户也不会感知到，因为用户使用的是 Facade 高层接口，内部的变化对于用户来说并不可见。这样一来就将变化隔离开来，使得系统也更为灵活。

23.6 Android 源码中的外观模式

在用 Android 开发过程中，Context 是最重要的一个类型，Context 意为上下文，也就是程序的运行环境。它封装了很多重要的操作，如 startActivity()、sendBroadcast()、bindService 等，因此，Context 对开发者来说是最重要的高层接口。Context 只是一个定义了很多接口的抽象类，这些接口的功能实现并不是在 Context 及其子类中，而是通过其他子系统来完成，例如 startActivity 的真正实现是通过 ActivityManagerService，获取应用包相关信息则是通过 PackageManagerService。Context 只是做了一个高层次的统一封装，正如上文所示，Context 只是一个抽象类，它的真正实现在 ContextImpl 类中，ContextImpl 就是今天我们要分析的外观类。

在本书的前面章节中已经提到多次，在应用启动时，首先会 fork 一个子进程，并且调用 ActivityThread.main 方法启动该进程。ActivityThread 又会构建 Application 对象，然后和 Activity、ContextImpl 关联起来，最后会调用 Activity 的 onCreate、onStart、onResume 函数使 Activity 运行起来，此时应用的用户界面就呈现在我们面前了。main 函数会间接地调用 ActivityThread 中的

handleLaunchActivity 函数启动默认的 Activity，handleLaunchActivity 代码如下。

```
private final void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
        // 代码省略
        // 1. 创建并且加载 Activity，调用其 onCreate 函数
        Activity a = performLaunchActivity(r, customIntent);
        if (a != null) {
            r.createdConfig = new Configuration(mConfiguration);
            Bundle oldState = r.state;
            // 2. 调用 Activity 的 onResume 方法，使 Activity 变得可见
            handleResumeActivity(r.token, false, r.isForward);
        }
    }

    private final Activity performLaunchActivity(ActivityClientRecord r,Intent customIntent){
        // 代码省略
        Activity activity = null;
        try {
            java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
            // 1. 创建 Activity
            activity = mInstrumentation.newActivity(
                    cl, component.getClassName(), r.intent);
        // 代码省略
        } catch (Exception e) {
        }

        try {
            // 2. 创建 Application
            Application app = r.packageInfo.makeApplication(false, mInstrumentation);
            if (activity != null) {
                // 3. ***** 构建 ContextImpl  ******
                ContextImpl appContext = new ContextImpl();
                // 初始化
                appContext.init(r.packageInfo, r.token, this);
                appContext.setOuterContext(activity);
                // 获取 Activity 的 title
                CharSequence title=r.activityInfo.loadLabel(appContext.getPackage Manager());
                Configuration config = new Configuration(mConfiguration);
                 // 4. Activity 与 Context、Application 关联起来
                activity.attach(appContext, this, getInstrumentation(), r.token,
                        r.ident, app, r.intent, r.activityInfo, title, r.parent,
                        r.embeddedID, r.lastNonConfigurationInstance,
                        r.lastNonConfigurationChildInstances, config);
                // 代码省略
                // 5. 回调 Activity 的 onCreate 方法
                mInstrumentation.callActivityOnCreate(activity, r.state);
                // 代码省略
            }
            r.paused = true;
            mActivities.put(r.token, r);
        } catch (SuperNotCalledException e) {
            throw e;
        } catch (Exception e) {
        }
        return activity;
    }
```

在 handleLaunchActivity 函数中会调用 perfromLaunchActivity 函数执行 Applicaton、ContextImpl、Activity 的创建工作，并且通过 Activity 类的 attach 函数将这 3 者关联起来，相关代码在注释 4 处。

而 Activity 本身又是 Context 的子类，因此，Activity 就具有了 Context 定义的所有方法。但 Activity 并不实现具体的功能，它只是继承了 Context 的接口，并且将相关的操作转发给 ContextImpl 对象。这个 ContextImpl 存储在 Activity 的上两层父类 ContextWrapper 中，变量名为 mBase，具体代码如下。

```
public class ContextThemeWrapper extends ContextWrapper {
    private Resources mResources;
    // 很鸡肋的函数覆写~
    @Override protected void attachBaseContext(Context newBase) {
        super.attachBaseContext(newBase);
    }
    // 代码省略
}
// ContextWrapper 类
public class ContextWrapper extends Context {
    Context mBase;

    protected void attachBaseContext(Context base) {
        if (mBase != null) {
            throw new IllegalStateException("Base context already set");
        }
        mBase = base;
    }
    // 代码省略
}
```

在 ActivityThread 类的 perfromLaunchActivity 函数中会调用 Activity 的 attach 方法将 ContextImpl 等对象关联到 Activity 中，这个 ContextImpl 最终会被 ContentWrapper 类的 mBase 字段引用。我们先看看 attach 方法的内部实现。

```
public class Activity extends ContextThemeWrapper
        implements LayoutInflater.Factory2,
        Window.Callback, KeyEvent.Callback,
        OnCreateContextMenuListener, ComponentCallbacks2 {

    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) {

        // 调用了 ContextThemeWrapper 的 attachBaseContext
        attachBaseContext(context);
        mMainThread = aThread;
        mApplication = application;
        // 其他字段的初始化
    }
}
```

attach 函数主要就是一些赋值操作，这里我们只关心 mBase 的初始化。在 attach 函数中，第一句就调用了 attachBaseContext 函数，该函数定义在 ContextWrapper 类，它就是简单地将 Context 参数传递给 mBase 字段。此时，我们的 Activity 内部就持有了 ContextImpl 的引用。

Activity 在开发过程中部分充当了代理的角色，例如，当我们通过 Activity 对象调用 sendBroadcast、getResource 等函数时，实际上 Activity 只是代理了 ContextImpl 的操作，也就是内部都调用了 mBase

对象的相应方法来处理，这些操作被封装在 Activity 的父类的 ContentWrapper 中。代码如下所示。

```
public class ContextWrapper extends Context {
    private Context mBase;

    @Override
    public void sendBroadcast(Intent intent) {
        mBase.sendBroadcast(intent);
    }

     @Override
    public Resources getResources()
    {
        return mBase.getResources();
    }

    @Override
    public PackageManager getPackageManager() {
        return mBase.getPackageManager();
    }
    // 代码省略
}
```

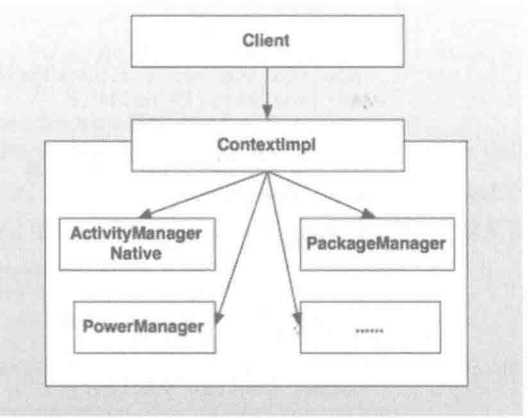
既然 ContextImpl 那么重要，包含了各个系统服务的调用与操作，那么我们来看看它的相关实现。

```
class ContextImpl extends Context {
    // 代码省略

        @Override
    public void sendBroadcast(Intent intent) {
        String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());
        try {
            ActivityManagerNative.getDefault().broadcastIntent(
                mMainThread.getApplicationThread(), intent, resolvedType, null,
                Activity.RESULT_OK, null, null, null, false, false);
        } catch (RemoteException e) {
        }
    }

        @Override
    public void startActivity(Intent intent) {
        mMainThread.getInstrumentation().execStartActivity(
            getOuterContext(),mMainThread.getApplicationThread(),null,null,intent,-1);
    }

    private Intent registerReceiverInternal(BroadcastReceiver receiver, int userId,
            IntentFilter filter, String broadcastPermission,
            Handler scheduler, Context context) {
            IIntentReceiver rd = null;
        // 代码省略
        try {
            return ActivityManagerNative.getDefault().registerReceiver(
                    mMainThread.getApplicationThread(), mBasePackageName,
                    rd, filter, broadcastPermission, userId);
        } catch (RemoteException e) {
            return null;
        }
    }
        @Override
    public ComponentName startService(Intent service) {
        try {
```

```
            ComponentName cn = ActivityManagerNative.getDefault().startService(
                mMainThread.getApplicationThread(), service,
                service.resolveTypeIfNeeded(getContentResolver()));
            // 代码省略
            return cn;
        } catch (RemoteException e) {
            return null;
        }
    }

    // 设置壁纸
@Override
    public void setWallpaper(Bitmap bitmap) throws IOException {
        getWallpaperManager().setBitmap(bitmap);
    }
        @Override
    public String getPackageName() {
        if (mPackageInfo != null) {
            return mPackageInfo.getPackageName();
        }
        throw new RuntimeException("Not supported in system context");
    }
}
```

从上述程序中可以看到，ContextImpl 内部封装了很多不同子系统的操作，例如，Activity 的跳转、发送广播、启动服务、设置壁纸等，这些工作并不是在 ContextImpl 中实现，而是转交给了具体的子系统进行处理。通过 Context 这个抽象了、定义了一组接口，ContextImpl 实现 Context 定义的接口，这使得用户可以通过 Context 这个接口统一进行与 Android 系统的交互，这样用户通常情况下就不需要对每个子系统进行了解，例如启动 Activity 时用户不需要手动调用 mMainThread.getInstrumentation().execStartActivity 函数进行执行，发送广播时也不需要直接操作 ActivityManagerNative 类。用户与系统服务的交互都通过 Context 的高层接口，这样对用户屏蔽了具体实现的细节，降低了使用成本。

通过 ContextImpl 的封装之后，用户与系统服务之间的交互如图 23-4 所示。



▲图 23-4

从上述示例来看，外观模式的实现非常简单，没有复杂的类型结构，只是通过一组高层接口封装了各个子系统的操作，并且统一提供给用户。试想一下如果没有外观模式的封装，那么用户就必须知道各个子系统的相关细节，甚至它们之间的协作流程，当子系统较多时这些相互之间的关系就会很乱，导致用户在使用相关功能时难度也会变大，易于出错。而使用外观模式的封装就避免了用户需要与多个子系统进行交互，降低了用户的使用成本，对外也屏蔽了具体细节，保证了系统的易用性、稳定性。

23.7 深度拓展

23.7.1 Android 资源的加载与匹配

在 Android 开发中，我们为了屏幕适配通常会为一个应用做多套资源，使得在不同分辨率的设备上能够尽量保持一致的 UI 效果。那么我们不禁要问 Android 的资源在哪？读取资源的形式又是怎样？下面我们就来简单分析一下 Android 的资源机制。

其实在 Android 应用程序资源的编译和打包之后就生成一个资源索引表文件 resources.arsc，这个应用程序资源会被打包到APK文件中。Android应用程序在运行的过程中，通过一个称为Resource来获取资源，但实际上 Resource 内部又是通过一个叫 AssetManager 的资源管理器来读取打包在 APK 文件里面的资源文件。那么 AssetManager 如何与应用关联在一起，它又是如何找到应用的资源，这就是我们本节要关注的重点。

我们在上文中说到，获取资源的操作实际上是由 ContextImpl 来完成的，Activity、Service 等组件的 getResource 函数最终都转发给了 ContextImpl 类型的 mBase 字段。也就是调用了 ContextImpl 的 getResource 函数，而这个 Resource 在 ContextImpl 关联到 Activity 之前就会初始化 Resource 对象，相关代码如下。

```
 private final Activity performLaunchActivity(ActivityClientRecord r, Intent
customIntent) {
        // 代码省略
        Activity activity = null;
        try {
            java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
            // 1. 创建 Activity
            activity = mInstrumentation.newActivity(
                    cl, component.getClassName(), r.intent);
         // 代码省略
        } catch (Exception e) {
        }
        try {
            // 2. 创建 Application
            Application app = r.packageInfo.makeApplication(false, mInstrumentation);
            if (activity != null) {
                // 3. ***** 构建 ContextImpl  ******
                ContextImpl appContext =createBaseContextForActivity(r, activity);
                // 启动 Activity 等代码省略
        } catch (Exception e) {
        }
        // 代码省略
}
    private Context createBaseContextForActivity(ActivityClientRecord r,
            final Activity activity) {
        // 创建 ContextImpl 对象
        ContextImpl appContext=ContextImpl.createActivityContext(this,r.packageInfo,r.token);
        // Outter Context 为 Activty
        appContext.setOuterContext(activity);
        Context baseContext = appContext;
        // 代码省略
        return baseContext;
    }
```

在上述 performLaunchActivity 函数中，首先创建了 Activity 与 Application，然后通过 createBaseContextForActivity 创建了一个 ContextImpl 对象。而在 createBaseContextForActivity 函数中又调用了 ContextImpl 类的 createActivityContext 静态函数，我们看看源码。

```
static ContextImpl createActivityContext(ActivityThread mainThread,
            LoadedApk packageInfo, IBinder activityToken) {
        // 代码省略

        return new ContextImpl(null, mainThread,
                packageInfo, activityToken, null, false, null, null);
    }

    private ContextImpl(ContextImpl container, ActivityThread mainThread,
            LoadedApk packageInfo,IBinder activityToken,UserHandle user,boolean restricted,.
            Display display, Configuration overrideConfiguration) {
            mOuterContext = this;
            mMainThread = mainThread;

            // 获取包信息
            mPackageInfo = packageInfo;
            // 获取资源管理器
            mResourcesManager = ResourcesManager.getInstance();
            // 代码省略

        Resources resources = packageInfo.getResources(mainThread);
        if (resources != null) {
            if (activityToken != null
                    || displayId != Display.DEFAULT_DISPLAY
                    || overrideConfiguration != null
                    || (compatInfo != null && compatInfo.applicationScale
                            != resources.getCompatibilityInfo().applicationScale)) {
                // 根据设备配置获取对应的资源
                resources = mResourcesManager.getTopLevelResources(packageInfo.getResDir(),
                packageInfo.getSplitResDirs(),packageInfo.getOverlayDirs(),
                packageInfo.getApplicationInfo().sharedLibraryFiles, displayId,
                overrideConfiguration, compatInfo, activityToken);
            }
        }
        mResources = resources;

       // 代码省略
    }
```

createActivityContext 函数中最终调用了 ContextImpl 的构造函数，在该函数中会初始化该进程的各个字段，例如资源、包信息、屏幕配置等。这里只关心与资源相关的代码。在通过 mPackageInfo 得到对应的资源之后，会调用 ResourcesManager 的 getTopLevelResources 来根据设备配置等相关信息获取到对应的资源，也就是资源的适配。getTopLevelResources 代码如下。

```
public Resources getTopLevelResources(String resDir, String[] splitResDirs,
            String[] overlayDirs, String[] libDirs, int displayId,
            Configuration overrideConfiguration, CompatibilityInfo compatInfo, IBinder
token) {
            final float scale = compatInfo.applicationScale;

        ResourcesKey key = new ResourcesKey(resDir, displayId, overrideConfiguration, scale,
token);
        Resources r;
```

```
        synchronized (this) {
            // 1. 判断是否已经加载过该资源
            WeakReference<Resources> wr = mActiveResources.get(key);
            r = wr != null ? wr.get() : null;
            if (r != null && r.getAssets().isUpToDate()) {
                return r;
            }
        }
        // 代码省略

        // 2. 还未加载过，则构建 AssetManager 对象
        AssetManager assets = new AssetManager();
        // 3. 将 APK 路径添加到 AssetManager 的资源路径中
        if (resDir != null) {
            if (assets.addAssetPath(resDir) == 0) {
                return null;
            }
        }
        // 代码省略

        // 屏幕分辨率
        DisplayMetrics dm = getDisplayMetricsLocked(displayId);
        // 设备配置
        Configuration config;

        // 4. 创建资源，参数 1 为 AssetManager
        r = new Resources(assets, dm, config, compatInfo, token);

        synchronized (this) {
            // 5. 缓存资源
            mActiveResources.put(key, new WeakReference<Resources>(r));
            return r;
        }
    }
```

首先会以 APK 路径、屏幕设备 id、配置等构建一个资源 key，根据这个 key 到 ActivityThread 类的 mActiveResources（HashMap 类型）中查询是否已经加载过该 APK 的资源，如果含有缓存那么直接使用缓存。这个 mActiveResources 维护了当前应用程序进程中加载的每一个 APK 文件及其对应的 Resources 对象的对应关系。如果没有缓存，那么就会新创建一个，并且保存在 mActiveResources 中。

在没有资源缓存的情况下，ActivityThread 会新创建一个 AssetManager 对象，并且调用 AssetManager 对象的 addAssetPath 函数来将参数 resDir 作为它的资源目录，这个 resDir 就是 APK 文件的绝对路径。创建了一个新的 AssetManager 对象之后，会将这个 AssetManager 对象作为 Resource 构造函数的第一个参数来构建一个新的 Resources 对象。这个新创建的 Resources 对象会以前面所创建的 ResourcesKey 对象为键值缓存在 mActiveResources 所描述的一个 HashMap 中，以便重复使用该资源时无需重复创建。

接下来，我们首先分析 AssetManager 类的构造函数和成员函数 addAssetPath 的实现，接着再分析 Resources 类的构造函数的实现，以便可以了解用来访问应用程序资源的 AssetManager 对象和 Resources 对象的创建以及初始化过程。

```
public AssetManager() {
```

```
        synchronized (this) {
            // 初始化
            init(false);
            // 确保加载了系统资源
            ensureSystemAssets();
        }
    }
    private AssetManager(boolean isSystem) {
        init(true);
    }

    private static void ensureSystemAssets() {
        synchronized (sSync) {
            if (sSystem == null) {
                // 系统资源的 AssetManager 对象
                AssetManager system = new AssetManager(true);
                system.makeStringBlocks(false);
                sSystem = system;
            }
        }
    }
```

在 AssetManager 的构造函数中，首先会调用 init 函数进行初始化，然后再调用 ensureSystemAssets 函数来加载系统资源，这些系统资源存储在 mSystem 对象中，mSystem 也是 AssetManager 类型。需要注意的是 init 函数并不是一个 Java 函数，也是一个 Native 层的方法，它的实现在 android_util_AssetManager.cpp 文件中，具体代码如下。

```
static void android_content_AssetManager_init(JNIEnv* env, jobject clazz)
{
    if (isSystem) {
        verifySystemIdmaps();
    }
    AssetManager* am = new AssetManager();  // 创建 Native 的 AssetManager
    am->addDefaultAssets();  // 添加默认的资源
    // 将 Native 层的 AssetManager 对象存储到 Java 层的 AssetManager 的 mObject 字段中
    env->SetIntField(clazz, gAssetManagerOffsets.mObject, (jint)am);
}
```

在 init 函数中首先创建了一个 Native 层的 AssetManager 对象，然后添加了默认的系统资源，最后将这个 AssetManager 对象转换为整型并且传递给 Java 层的 AssetManager 的 mObject 字段，这样 Java 层就相当于存有了一个 Native 层 AssetManager 的句柄。这里我们关注的是 addDefaultAssets 函数，具体代码如下。

```
static const char* kSystemAssets = "framework/framework-res.apk";

bool AssetManager::addDefaultAssets()
{
    const char* root = getenv("ANDROID_ROOT");
    // 构建系统路径
    String8 path(root);
    // 系统路径 + framework 资源 APK 路径，拼接成完整路径
    path.appendPath(kSystemAssets);
    return addAssetPath(path, NULL);
}
```

代码也比较简单，就是拼接系统资源路径，最好将该路径传递给 addAssetPath 函数中，实际上

就是将系统资源路径添加到 AssetManager 的资源路径列表中。

```
bool AssetManager::addAssetPath(const String8& path, void** cookie)
{
    AutoMutex _l(mLock);
    asset_path ap;
    String8 realPath(path);
    if (kAppZipName) {
        realPath.appendPath(kAppZipName);
    }
    // 判断 path 路径是否存在以及是否是文件类型,如果路径不合法则直接返回 false
    // ap.type = ::getFileType(realPath.string());
    if (ap.type == kFileTypeRegular) {
        ap.path = realPath;
    } else {
        ap.path = path;
        ap.type = ::getFileType(path.string());
        if (ap.type != kFileTypeDirectory && ap.type != kFileTypeRegular) {
            // 代码省略
            return false;
        }
    }

    // 添加到资源路径列表中
    mAssetPaths.add(ap);
   // 代码省略
    return true;
}
```

此时，在 ActivityThread 的 getTopLevelResources 函数中的 new AssetManager 的过程就执行完毕了，然后继续执行 AssetManager 对象的 addAssetPath 函数，具体代码如下。

```
public final int addAssetPath(String path) {
    int res = addAssetPathNative(path);
    return res;
}
```

在 Java 层的 AssetManager 的 addAssetPath 函数中实际上调用的是 Native 层的 addAssetPathNative 函数。需要注意的是，这个 path 参数必须是一个目录或者是一个 zip 压缩文件（APK 本质上就是一个 zip 文件）路径。这个 addAssetPathNative 函数在 Native 层对应的函数为 android_content_AssetManager_addAssetPath，该函数定义在 android_util_AssetManager.cpp 文件中。

```
static jint android_content_AssetManager_addAssetPath(JNIEnv* env, jobject clazz,
                                                      jstring path)
{
    ScopedUtfChars path8(env, path);
    if (path8.c_str() == NULL) {
        return 0;
    }
    // 将 Java AssetManager 对象中的 mObject 字段转换为 AssetManager 指针
    AssetManager* am = assetManagerForJavaObject(env, clazz);
    if (am == NULL) {
        return 0;
    }

void* cookie;
// 将新的路径添加到资源列表中
    bool res = am->addAssetPath(String8(path8.c_str()), &cookie);
```

```
    return (res) ? (jint)cookie : 0;
}
```

在android_content_AssetManager_addAssetPath函数中会调用assetManagerForJavaObject函数先从Java层的AssetManager对象中获取到mObject字段，该字段存储了由Native层AssetManager指针转换而来的整型值。此时需要通过这个整型值逆向地转换为Native层的AssetManager对象。然后将APK路径添加到资源路径中，这样也就含有了应用本身的资源。

我们再来分析这个过程，首先在Java层的AssetManager构造函数中调用了init函数初始化系统资源，创建与初始化完毕之后又调用了Java层AssetManager对象的addAssetPath函数将应用APK的路径传递给Native层的AssetManager，使得应用的资源也被添加到AssetManager中，这样资源路径就构建完毕了。

最后在ResourcesManager的getTopLevelResources函数中将初始化好的AssetManager、设备配置等作为参数来构造Java层的Resource对象，也就是上文中getTopLevelResources函数中的注释4处。Resource的构造函数如下。

```
public Resources(AssetManager assets, DisplayMetrics metrics, Configuration config,
        CompatibilityInfo compatInfo, IBinder token) {
        mAssets = assets;
        mMetrics.setToDefaults();
// 更新配置
        updateConfiguration(config, metrics);
        assets.ensureStringBlocks();
}
```

Resources类的构造函数首先将参数assets所指向的一个AssetManager对象保存在成员变量mAssets中，我们获取资源就是通过这个AssetManager对象进行操作。接下来调用updateConfiguration函数来设置设备配置信息，最后调用AssetManager的成员函数ensureStringBlocks来创建字符串资源池。

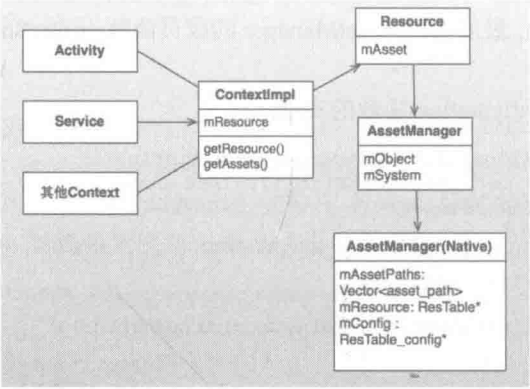
我们来看看updateConfiguration函数的实现。

```
public void updateConfiguration(Configuration config,
        DisplayMetrics metrics, CompatibilityInfo compat) {
        synchronized (mAccessLock) {
        // 与屏幕相关的信息
        if (metrics != null) {
            mMetrics.setTo(metrics);
        }
        mCompatibilityInfo.applyToDisplayMetrics(mMetrics);
        // 代码省略
        // 设备DPI
        if (mConfiguration.densityDpi != Configuration.DENSITY_DPI_UNDEFINED) {
            mMetrics.densityDpi = mConfiguration.densityDpi;
            mMetrics.density=mConfiguration.densityDpi*DisplayMetrics.DENSITY_DEFAULT_SCALE;
        }
        mMetrics.scaledDensity = mMetrics.density * mConfiguration.fontScale;

        // 代码省略
        // 屏幕宽高
        int width, height;
        if (mMetrics.widthPixels >= mMetrics.heightPixels) {
            width = mMetrics.widthPixels;
            height = mMetrics.heightPixels;
```

```
        } else {
            //noinspection SuspiciousNameCombination
            width = mMetrics.heightPixels;
            //noinspection SuspiciousNameCombination
            height = mMetrics.widthPixels;
        }
        // 代码省略
        // 根据配置初始化 AssetManager
        mAssets.setConfiguration(mConfiguration.mcc, mConfiguration.mnc,
                locale, mConfiguration.orientation,
                mConfiguration.touchscreen,
                mConfiguration.densityDpi, mConfiguration.keyboard,
                keyboardHidden, mConfiguration.navigation, width, height,
                mConfiguration.smallestScreenWidthDp,
                mConfiguration.screenWidthDp, mConfiguration.screenHeightDp,
                mConfiguration.screenLayout, mConfiguration.uiMode,
                Build.VERSION.RESOURCES_SDK_INT);
    }

}
```

Resources 类的成员函数 updateConfiguration 首先是根据参数 config 和 metrics 来更新设备的当前配置信息，例如，屏幕大小和密码、国家（地区）和语言、键盘配置情况等，接着再调用成员变量 mAssets 所指向的一个 Java 层的 AssetManager 对象的成员函数 setConfiguration 来将这些配置信息设置到与之关联的 C++层的 AssetManager 对象中。这样一来，在我们通过 Resource 获取资源时，Native 层就会根据这个配置信息寻找最适合的资源返回，从而达到多屏幕适配的效果。

从上述的分析中我们可以知道，Activity、ContextImpl、Resource、AssetManager 的结构图为图 23-5 所示。



▲图 23-5

经过上述的分析，我们也可以得出一个结论：应用的资源存储在 APK 中，一个 AssetManager 可以管理多个路径的资源文件，Resource 通过 AssetManager 获取资源。那就是说我们可以通过 AssetManager 的 addAssetPath 方法添加 APK 路径以达到一些资源替换或者换肤的效果，还有类似于动态加载一个未安装的 APK 时也可以通过这种形式加载插件 APK 的资源，使得插件 Activity 等组件可以通过资源类 R 来访问应用资源。

23.7.2 动态加载框架的实现

在上一节中我们学习了 Android 中的资源机制，本节我们将资源机制作为基本知识点来实现一个简单的动态加载框架。在开始之前我们需要知道，未安装的 APK 文件是可以通过 DexClassLoader 进行加载、运行的，具体代码如下。

```
private DexClassLoader createDexClassLoader(String apkPath) {
        File dexOutputDir = mContext.getDir("dex", Context.MODE_PRIVATE);
        DexClassLoader loader = new DexClassLoader(apkPath, dexOutputDir.
        getAbsolutePath(), null,
        mContext.getClassLoader());
        return loader;
    }
```

其中 mContext 是一个 Context 类型的对象，apkPath 就是这个 APK 在手机中的绝对路径，dexOutputDir 就是 APK 解压出的 dex 文件的存放目录。通过这种形式就有了一个可以加载这个 APK 的 ClassLoader。然后通过这个 ClassLoader 加载这个 APK 默认启动的 Activity，并且通过反射调用这个插件 Activity 的 onCreate、onStart、onResume 函数就会加载这个应用。

我们的实现原理当然没有上面所说的那么简单，还需要处理的问题包括加载插件 APK 的资源、处理它的生命周期函数等。不过在此之前，首先做的还是构建能够加载插件 APK 的 ClassLoader 和资源，我们新创建了一个 PluginManager 类来完成这些工作，具体代码如下。

```
// 插件管理器
public class PluginManager {
    static class PluginMgrHolder {
        static PluginManager sManager = new PluginManager();
    }

    // Application Context
    private static Context mContext;

    // 插件 APK 包名为 key、PluginApk 为值的 HashMap
    Map<String, PluginApk> sMap = new HashMap<String, PluginApk>();

    // 在使用 PluginManager 之前需要初始化一次
    public static void init(Context context) {
        mContext = context.getApplicationContext();
    }

    /**
     * 加载插件 APK 相关信息，包含插件 APK 的资源、ClassLoader 等
     * @param apkPath 插件的绝对路径
     */
    public final void loadApk(String apkPath) {
        PackageInfo packageInfo = queryPackageInfo(apkPath);
        if (packageInfo == null || TextUtils.isEmpty(packageInfo.packageName)) {
            return;
        }
        // 通过包名查询是否有缓存
        PluginApk pluginApk = sMap.get(packageInfo.packageName);
        if (pluginApk == null) {
            pluginApk = createApk(apkPath);
            if (pluginApk != null) {
                pluginApk.packageInfo = packageInfo;
                sMap.put(packageInfo.packageName, pluginApk);
```

```
            } else {
                throw new NullPointerException("PluginApk is null!");
            }
        }
    }
}
```

在 PluginManager 类中以插件 APK 的包名为 key，插件 APK 信息为值缓存了插件相关的信息。加载插件 APK 时会从 map 中检测是否含有缓存，如果有则使用缓存的 APK，否则通过 DexClassLoader 和资源创建一个 PluginApk 对象，最后将这个对象缓存到 map 中。创建 PluginApk 的代码是 createApk 函数，具体代码如下。

```
private PluginApk createApk(String apkPath) {
    PluginApk pluginApk = null;
    try {
        // 1. 创建 AssetManager
        AssetManager assetManager = AssetManager.class.newInstance();
        Method addAssetPath=assetManager.getClass().getMethod("addAssetPath",String.class);
        // 2. 将 APK 的目录添加到 AssetManager 的资源路径下
        addAssetPath.invoke(assetManager, apkPath);
        // 3. 以 AssetManager 和设备配置来构建 Resource 对象
        Resources pluginRes = new Resources(assetManager,
                mContext.getResources().getDisplayMetrics(), mContext.getResources()
                    .getConfiguration());
        // 4. 创建 PluginApk 对象，存储插件 APK 的资源对象
        pluginApk = new PluginApk(pluginRes);
        // 5. 存储 DexClassLoader 对象
        pluginApk.classLoader = createDexClassLoader(apkPath);
    } catch (Exception e) {
    }
    return pluginApk;
}

private DexClassLoader createDexClassLoader(String apkPath) {
    File dexOutputDir = mContext.getDir("dex", Context.MODE_PRIVATE);
    DexClassLoader loader = new DexClassLoader(apkPath, dexOutputDir.getAbsolutePath(), null,
    mContext.getClassLoader());
    return loader;
}
```

在 createApk 函数中首先通过 AssetManager 对象添加了插件 APK 所在的绝对路径，使得 AssetManager 可以获取到该插件 APK 的资源，然后又通过该 AssetManager 和设备配置等创建了 Resource 对象，并且将这个对象存储到 PluginApk 中，最后构建插件 APK 的 DexClassLoader 并存储在 PluginApk 对象中。最后这个 PluginApk 对象会缓存到 PluginManager 的信息表中。

在启动插件之前，用户需要调用 PluginManager 的 loadApk 函数加载插件 APK 的相关信息，也就是上述所说的资源、DexClassLoader 等，然后再调用 PluginManager 的 startActivity 函数启动插件，具体代码如下。

```
public void startActivity(Intent intent) {
    Intent myIntent = new Intent(mContext, ActivityProxy.class);
    Bundle extra = intent.getExtras();
    if (extra == null
            || !extra.containsKey(Constants.PLUGIN_CLASS_NAME)
        && !extra.containsKey(Constants.PACKAGE_NAME)) {
        thrownew IllegalArgumentException("没有设置插件的类路径和包名");
```

```
    }        myIntent.putExtras(intent);
    myIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    mContext.startActivity(myIntent);
}
```

需要注意的是，启动插件时需要在 Intent 中传递插件 APK 的包名和默认加载的 Activity 类名，否则会抛出异常。因为 PluginManager 需要知道插件 APK 默认的启动 Activity 才能将插件 APK 运行起来。还有一个重要的地方是，我们在 startActivity 中会重新构建一个 Intent，这个 Intent 启动的却是 ActivityProxy 这个类。我们是要启动插件 APK 的默认 Activity，怎么换成了 ActivityProxy 呢？

实际上我们动态加载的并不是插件 APK 的 Activity，而是一个 Activity 空壳，也就是这里的 ActivityProxy。在这个空壳 Activity 中包装了插件 Activity 的声明周期，也就说插件 Activity 和 ActivityProxy 是共存的。相关代码如下。

```
public class ActivityProxy extends Activity {
    // 声明周期控制器
    LifeCircleController mPluginController = new LifeCircleController(this);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // 调用 mPluginController 的 onCreate，实际上是启动插件 APK 的 Activity
        mPluginController.onCreate(getIntent().getExtras());
    }

    @Override
    public Resources getResources() {
        // 资源在加载 APK 时就已经构造
        Resources resources = mPluginController.getResources();
        return resources == null ? super.getResources() : resources;
    }

    @Override
    public Theme getTheme() {
        Theme theme = mPluginController.getTheme();
        return theme == null ? super.getTheme() : mPluginController.getTheme();
    }

    @Override
    public AssetManager getAssets() {
        return mPluginController.getAssets();
    }

    @Override
    protected void onStart() {
        super.onStart();
        mPluginController.onStart();
    }

    @Override
    protected void onResume() {
        super.onResume();
        mPluginController.onResume();
    }
    // 其他声明周期函数
}
```

在 ActivityProxy 中有一个 LifeCircleController，这个类就负责通过 DexClassLoader 加载插件

APK 的 Activity，我们看到 ActivityProxy 的声明周期函数中都调用了 LifeCircleController 对象对应的方法，而 LifeCircleController 中实际上又调用了插件 Activity 的生命周期方法。我们看看 LifeCircleController 的实现。

```
public class LifeCircleController implements Pluginable {

    Activity mProxy;
    PluginActivity mPlugin;
    Resources mResources;
    Theme mTheme;
    PluginApk mPluginApk;
    String mPluginClazz;

    // 构造函数，参数为 ActivityProxy
    public LifeCircleController(Activity activity) {
        mProxy = activity;
    }

    public void onCreate(Bundle bundle) {
        // 1. 获取通过 Intent 传递进来的插件 Activity 的类路径和包名
        mPluginClazz = bundle.getString(Constants.PLUGIN_CLASS_NAME);
        String packageName = bundle.getString(Constants.PACKAGE_NAME);
        // 2. 从 PluginManager 取出缓存的 PluginApk 对象
        mPluginApk = PluginManager.getInstance().getPluginApk(packageName);
        try {
            // 3. 加载插件 Activity
            mPlugin = (PluginActivity) loadPluginable(mPluginApk.classLoader, mPluginClazz);
            // 4. 将 ActivityProxy 对象注入到插件 Activity 中
            mPlugin.attach(mProxy, mPluginApk);
            // 设置资源为插件 APK 的资源
            mResources = mPluginApk.resources;
            // 5. 调用插件的 onCreate 函数
            mPlugin.onCreate(bundle);
        } catch (Exception e) {
            Log.e("", "### 加载插件类失败 ");
            e.printStackTrace();
        }
    }
}
```

在 ActivityProxy 的 onCreate 函数中调用了 LifeCircleController 的 onCreate 函数，在 LifeCircleController 的 onCreate 函数中主要做了如下几步操作。

（1）获取插件 Activity 的类路径、包名。

（2）通过包名到 PluginManager 中获取 PluginApk 信息。

（3）通过 PluginApk 中的 DexClassLoader 加载插件 Activity。

（4）将 ActivityProxy 对象注入到插件 Activity 中。

（5）调用插件 Activity 的 onCreate 函数。

在 PluginManager 的 startActivity 函数中我们强调过，跳转到插件 Activity 的 Intent 必须包含插件 Activity 的类路径、包名就是为了在这一步使用。获取到类名之后再得到 DexClassLoader，最后动态加载插件 Activity，并且将 ActivityProxy 注入到插件 Activity 中。最后调用插件 Activity 的 onCreate 函数。

对于插件 Activity 我们也定义了一个基类，叫做 PluginActivity，具体代码如下。

```
public abstract class PluginActivity extends Activity implements Pluginable,
Attachable<Activity> {
    protected Activity mProxyActivity;   // ActivityProxy 对象
    private Resources mResource;         // 插件 Activity 的资源
    PluginApk mPluginApk;

    @Override
    public void attach(Activity proxy, PluginApk apk) {
        mProxyActivity = proxy;
        mPluginApk = apk;
        mResource = apk.resources;
    }

    @Override
    public void setContentView(int layoutResID) {
        mProxyActivity.setContentView(layoutResID);
    }
    @Override
    public View findViewById(int id) {
        return mProxyActivity.findViewById(id);
    }

    @Override
    public Resources getResources() {
        return mResource;
    }

    @Override
    public Window getWindow() {
        return mProxyActivity.getWindow();
    }
    // 代码省略
}
```

从上述程序中可以看到，在 PluginActivity 中所有的函数几乎都转交给了 ActivityProxy 处理，比如 setContentView，实际上也是设置给了 ActivityProxy 对象。此时我们可以思考一下，我们在开发插件时会继承自 PluginActivity，然后在 onCreate 函数中设置 setContentView 为插件 Activity 设置布局内容，而 PluginActivity 调用的却是 ActivityProxy 的 setContentView 函数，就相当于是为 ActivityProxy 对象设置内容布局。当我们要启动插件 Actiivty 时实际上又是启动 ActivityProxy，又在 ActivityProxy 的 onCreate 函数中通过 LifeCircleController 动态地用 DexClassLoader 加载插件 Activity，并且调用它的 onCreate 函数、setContentView 方法等，而 setContentView 实际上调用了 ActivityProxy 的 setContentView，在调用 ActivityProxy 的 onStart、onResume 之后 ActivityProxy 就显示到我们面前了，它的视图就是通过插件 Activity 设置的内容。

一句话概括就是：ActivityProxy 加载的是插件 Activity 的内容与资源。这样一来，插件 Activity 就会以 ActivityProxy 的形式展现出来。

例如，我们有一个插件 APK 在 SD 卡的 plugins 目录下，它的包名为 com.example.plugin，默认启动的 Activity 类路径为 com.example.plugin.MainActivity。首先需要一个宿主应用来加载插件应用，宿主应用中要加载插件 APK 时代码调用如下。

```
public class MainActivity extends Activity {
    PluginManager pluginMgr;
```

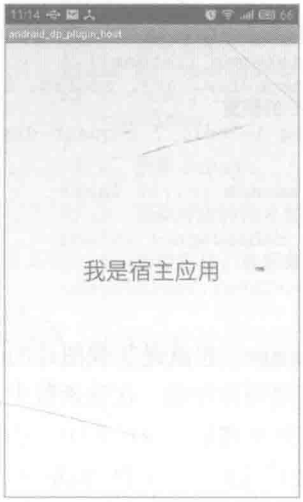
```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 1. 初始化 PluginManager
        PluginManager.init(getApplicationContext());
        pluginMgr = PluginManager.getInstance();
        // 2. 插件 APK 路径
        String pluginApkPath = Environment.getExternalStorageDirectory()
                    + File.separator + "plugins" + File.separator
                    + "plugin.apk";
        String pluginClazz = "com.example.plugin.MainActivity";
        String pluginPackage = "com.example.plugin";
        // 3. 加载 APK
        PluginManager.getInstance().loadApk(pluginApkPath);
        // 4. 构建 Intent，注意，传递插件默认启动的 Activity 类路径和包名
        Intent intent = new Intent();
        intent.putExtra(Constants.PLUGIN_CLASS_NAME, pluginClazz);
        intent.putExtra(Constants.PACKAGE_NAME, pluginPackage);
        // 5. 启动
        pluginMgr.startActivity(intent);
    }
}
```

我们再创建一个插件工程，这个工程需要引用动态加载框架的代码，然后新创建一个继承自 PluginActivity 的插件 Activity，具体代码如下。

```
public class MainActivity extends PluginActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.plugin_activity_main);

        // 通过 R 获取 View 控件
        final TextView textView = (TextView) findViewById(R.id.my_textview);
        // 通过 R 获取字符串资源
        textView.setText(R.string.plugin_text);

    }
}
```

生成 APK 之后将该插件 APK 命名为 plugin.apk，放到 SD 卡的 plugins 目录下，然后运行宿主 APK，而在宿主 APK 中又会加载 plugin.apk，此时得到的效果如图 23-6 和图 23-7 所示。

在宿主中，我们通过 PluginManager 启动了插件 APK。但是，实际上启动是 ActivityProxy，只是我们将插件 Activity 的 content view 设置给了 ActivityProxy，这样看起来像是启动了插件 Activity。在这个过程中最终的两点就是 content view 的设置和资源的替换。当然这只是一个最基本的实现，通过这个示例只是让大家对于动态加载有一个基本的认识，更完善的动态加载框架请查看 https://github.com/singwhatiwanna/dynamic-load-apk，也希望更多的人参与到这个开源项目来。



▲图 23-6 宿主

▲图 23-7 插件

23.8 外观模式实战

对于 SDK 和开源库来说，外观模式通常是使用率最高的模式，这些库通过外观类为用户提供统一的高层接口，使得用户不必了解一些更细节的实现。例如在使用 ImageLoader 时我们通常只操作一个 ImageLoader 类就可以完成，而不要了解网络请求类、缓存类以及它们的交互细节。又比如在使用友盟统计 SDK 时，我们基本上通过 MobclickAgent 这个类就可以完成我们所需的功能，至于 MobclickAgent 的内部有其他的什么类型、它们的具体交互是什么我们都不需要关心，这个 MobclickAgent 类也就是一个外观类。可见，外观模式实际上在我们的开发中无处不在，今天我们还是以小民的 ImageLoader（类名为 SimpleImageLoader，简称 ImageLoader）为例来讲述外观模式的运用。

在本书的前面章节中我们讲到，用户在使用 ImageLoader 时只需要操作 ImageLoader 类，其他的类型基本不用关心。ImageLoader 封装了内部逻辑，通过 ImageLoader 的接口用户就可以完成图片加载的操作。核心代码如下。

```
public final class SimpleImageLoader {
    // SimpleImageLoader 实例
    private static SimpleImageLoader sInstance;
    // 网络请求队列
    private RequestQueue mImageQueue;
    // 图片加载配置对象
    private ImageLoaderConfig mConfig;

    publicvoidinit(ImageLoaderConfig config) {
        mConfig = config;
        checkConfig();
        // 启动请求队列
        mImageQueue = new RequestQueue(mConfig.threadCount);
        mImageQueue.start();
    }
```

```
    // 加载图片
    public void displayImage(final ImageView imageView, final String uri,
            final DisplayConfig config, final ImageListener listener) {
        BitmapRequest request = new BitmapRequest(imageView, uri, config, listener);
        // 加载的配置对象，如果没有设置，则使用 ImageLoader 的配置
        request.displayConfig = request.displayConfig != null ? request.displayConfig
                : mConfig.displayConfig;
        // 设置加载策略
        request.setLoadPolicy(mConfig.loadPolicy);
        // 添加到队列中
        mImageQueue.addRequest(request);
    }
}
```

在 ImageLoader 类中对外暴露的函数基本只有 displayImage，也就是加载图片的函数。最简单的情况下，用户只需要传递 ImageView 和图片的 uri 即可实现图片加载。在该函数中，首先会将传递进来的参数转换为一个 BitmapRequest 对象，然后将该对象传递到请求队列中。请求队列的初始化就在 ImageLoader 的初始化函数中。该请求队列初始化之后就会启动 CPU 数量 + 1 个的请求处理线程，在这些线程的 run 中不断地从请求队列中获取请求、加载请求、最后将图片投递到 UI 线程更新 ImageView。首先我们来看消息队列的初始化。

```
public final class RequestQueue {
    // 请求队列
    private BlockingQueue<BitmapRequest> mRequestQueue = new PriorityBlockingQueue
<BitmapRequest>();

    // 默认的核心数
    public static int DEFAULT_CORE_NUMS = Runtime.getRuntime().availableProcessors() + 1;
    // NetworkExecutor,执行网络请求的线程
    private RequestDispatcher[] mDispatchers = null;

    // 启动 RequestDispatcher
    private final void startDispatchers() {
        mDispatchers = new RequestDispatcher[mDispatcherNums];
        for (int i = 0; i < mDispatcherNums; i++) {
            Log.e("", "### 启动线程 " + i);
            mDispatchers[i] = new RequestDispatcher(mRequestQueue);
            mDispatchers[i].start();
        }
    }

    public void start() {
        stop();
        startDispatchers();
    }
    // 代码省略
}
```

在 RequestQueue 中会启动指定数量的 RequestDispatcher 线程，每个 RequestDispatcher 本质上是一个线程，在它们的 run 函数中有一个死循环不断地查询请求队列中是否含有请求，并且处理这些请求。

```
public RequestDispatcher extends Thread {
    @Override
    public void run() {
        try {
```

```
            while (!this.isInterrupted()) {
                // 1. 读取请求
                final BitmapRequest request = mRequestQueue.take();
                if (request.isCancel) {
                    continue;
                }
                // 2. 获取 Schema
                final String schema = parseSchema(request.imageUri);
                // 3. 获取对应的图片加载器
                Loader imageLoader = LoaderManager.getInstance().getLoader(schema);
                // 4. 加载图片，最后会将结果传递给 UI 线程并更新 UI
                imageLoader.loadImage(request);
            }
        } catch (InterruptedException e) {
            Log.i("", "### 请求分发器退出");
        }
    }
}
```

在 RequestDispatcher 的 run 方法中主要分为 4 步。

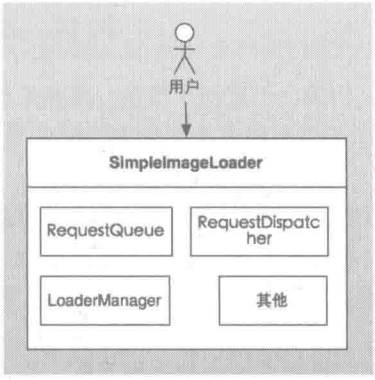
（1）从消息队列中获取消息。

（2）根据请求的 uri 地址获取图片的 uri schema。

（3）根据 schema 获取对应的图片 Loader。

（4）加载图片，并且将结果传递给 UI 线程，更新 ImageView。

从上述代码看，RequestDispatcher 中又封装了 LoaderManager 系统，而在 LoaderManager 中又管理了各个 Loader 类，而 Loader 类中又管理了图片的加载流程、图片缓存等逻辑。而这些相关的类型、逻辑都被封装在 ImageLoader 类的外衣之下，对于用户来说它们根本不知道这些类型的存在，几乎所有的操作都是通过 SimpleImageLoader 这个类完成。它们的结构如图 23-8 所示。



▲图 23-8

从图 23-8 中我们可以看到，ReuqestQueue 等类型都被封装在了 SimpleImageLoader 类之下，用户根本不知道这些类型的存在，而只需要操作 SimpleImageLoader 的接口就可以完成图片加载的操作。这样就避免了暴露过多的实现细节，也使得 ImageLoader 的使用更为简单，试想一下，每次都需要手动构建一个 BitmapRequest 类，然后添加到请求队列中，那么整个过程就增加了用户的使用成本，用户也直接依赖了具体的实现，这对于系统的升级、维护造成了一定的困难。通过

SimpleImageLoder 类将这些具体细节都隐藏起来，在降低用户使用成本的同时也增加了灵活性，例如，当你需要替换 RequestDispatcher 为线程池实现时，用户并不能感知到，因为原来的用户并不知道内部实现有 RequestDispatcher 这个类型，这就能很好地在不对用户产生影响的情况下对产品进行升级、维护。

23.9 总结

外观模式是一个高频率使用的设计模式，它的精髓就在于封装二字。通过一个高层次结构为用户提供统一的 API 入口，使得用户通过一个类型就基本能够操作整个系统，这样减少了用户的使用成本，也能够提升系统的灵活性。

优点

（1）对客户程序隐藏子系统细节，因而减少了客户对于子系统的耦合，能够拥抱变化。

（2）外观类对子系统的接口封装，使得系统更易于使用。

缺点

（1）外观类接口膨胀。由于子系统的接口都有外观类统一对外暴露，使得外观类的 API 接口较多，在一定程度上增加了用户使用成本。

（2）外观类没有遵循开闭原则，当业务出现变更时，可能需要直接修改外观类。

第 24 章 连接两地的交通枢纽——桥接模式

24.1 桥接模式介绍

桥接模式（Bridge Pattern）也称为桥梁模式，是结构型设计模式之一。在现实生活中大家都知道“桥梁”是连接河道两岸的主要交通枢纽，简而言之其作用就是连接河的两边，而我们的桥接模式与现实中的情况很相似，也是承担着连接“两边”的作用，那么具体是哪两边呢？这里先不着急，我们先来看看该模式的定义。

24.2 桥接模式的定义

将抽象部分与实现部分分离，使它们都可以独立地进行变化。

24.3 桥接模式的使用场景

从模式的定义中我们大致可以了解到，这里“桥梁”的作用其实就是连接“抽象部分”与“实现部分”，但是事实上，任何多维度变化类或者说多个树状类之间的耦合都可以使用桥接模式来实现解耦。

如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，可以通过桥接模式使它们在抽象层建立一个关联关系。

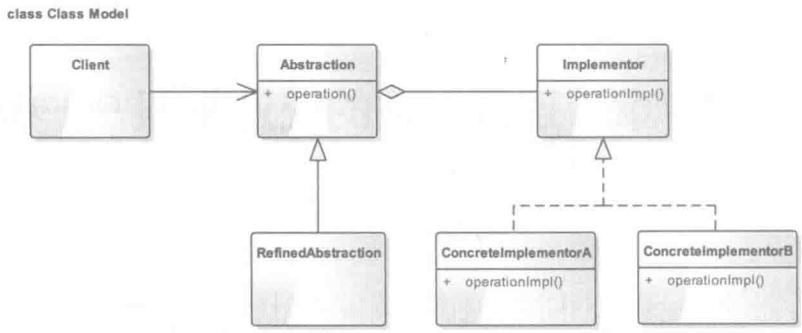
对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，也可以考虑使用桥接模式。

一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

24.4 桥接模式的 UML 类图

UML 类图如图 24-1 所示。

根据类图我们可以得出如下一个桥接模式的通用模式代码。



▲图 24-1

程序清单 24-1　实现部分的抽象接口

```
public interface Implementor {
    /**
     * 实现抽象部分的具体方法
     */
    public void operationImpl();
}
```

程序清单 24-2　实现部分具体的实现一

```
public class ConcreteImplementorA implements Implementor {
    @Override
    public void operationImpl() {
        // 具体的实现
    }
}
```

程序清单 24-3　实现部分具体的实现二

```
public class ConcreteImplementorB implements Implementor {
    @Override
    public void operationImpl() {
        // 具体的实现
    }
}
```

程序清单 24-4　抽象部分的实现

```
public abstract class Abstraction {
    private Implementor mImplementor;// 声明一个私有成员变量引用实现部分的对象

    /**
     * 通过实现部分对象的引用构造抽象部分的对象
     * @param implementor 实现部分对象的引用
     */
    public Abstraction(Implementor implementor) {
        mImplementor = implementor;
    }

    /**
     * 通过调用实现部分具体的方法实现具体的功能
     */
    public void operation() {
        mImplementor.operationImpl();
    }
}
```

程序清单 24-5 抽象部分的子类

```
public class RefinedAbstraction extends Abstraction {
    public RefinedAbstraction(Implementor implementor) {
        super(implementor);
    }

    /**
     * 对父类抽象部分中的方法进行扩展
     */
    public void refinedOperation() {
        // 对 Abstraction 中的方法进行扩展
    }
}
```

程序清单 24-6 客户类

```
public class Client {
    public static void main(String[] args) {
        // 客户调用逻辑
    }
}
```

角色介绍。

- Abstraction：抽象部分。

该类保持一个对实现部分对象的引用，抽象部分中的方法需要调用实现部分的对象来实现，该类一般为抽象类。

- RefinedAbstraction：优化的抽象部分。

抽象部分的具体实现，该类一般是对抽象部分的方法进行完善和扩展。

- Implementor：实现部分。

可以为接口或抽象类，其方法不一定要与抽象部分中的一致，一般情况下是由实现部分提供基本的操作，而抽象部分定义的则是基于实现部分这些基本操作的业务方法。

- ConcreteImplementorA/ConcreteImplementorB：实现部分的具体实现。

完善实现部分中方法定义的具体逻辑。

- Client：客户类，客户端程序。

24.5 桥接模式的简单实现

现实生活中有很多桥接模式应用的影子，比如开关与具体的电器，开关的类型有多种，而电器也是各式各样，这两者是独立变化的且又有耦合。还有程序员天天面对的显示屏，对于显示屏来说它的尺寸与生厂商之间也是一种二维关系，具体的尺寸与具体的厂商独立变化。而更贴近生活的例子就是我们在喝咖啡时，大家知道去咖啡馆喝咖啡一般分为 4 种。大杯加糖、大杯不加糖、小杯加糖、小杯不加糖，对一杯咖啡来说这 4 种实质上就两种变化，一是大杯小杯，二是加糖不加糖，不管怎样，我们都先来定义一个咖啡类。

```
public abstract class Coffee {
    protected CoffeeAdditives impl;

    public Coffee(CoffeeAdditives impl) {
```

```
        this.impl = impl;
    }

    /**
     * 咖啡具体是什么样的由子类决定
     */
    public abstract void makeCoffee();
}
```

Coffee 类中保持了对 CoffeeAdditives 的引用，以便调用具体的实现。同样地，咖啡还分大杯小杯，定义两个子类继承于 Coffee。

```
public class LargeCoffee extends Coffee {
    public LargeCoffee(CoffeeAdditives impl) {
        super(impl);
    }

    @Override
    public void makeCoffee() {
        System.out.println("大杯的" + impl + "咖啡");
    }
}
public class SmallCoffee extends Coffee {
    public SmallCoffee(CoffeeAdditives impl) {
        super(impl);
    }

    @Override
    public void makeCoffee() {
        System.out.println("小杯的" + impl + "咖啡");
    }
}
```

而对于加进咖啡中的糖，当然也可以选择不加，我们也用一个抽象类定义。

```
public abstract class CoffeeAdditives {
    /**
     * 具体要往咖啡里添加什么也是由子类实现
     *
     * @return 具体添加的东西
     */
    public abstract String addSomething();
}
```

注意，这里的 CoffeeAdditives 其实就对应于上面我们 UML 类图中的实现部分，而 Coffee 则对应于抽象部分，这里其实笔者还想多说一句，模式定义中所谓的“抽象”与“实现”实质上对应的是两个独立变化的维度，因此，上文中我们也曾说过，任何多维度变化类或者说多个树状类之间的耦合都可以使用桥接模式来实现解耦。在本例中，我们的 Coffee 类虽是一个抽象类，但它并非是所谓的“抽象部分”，而 CoffeeAdditives 类也并非一定就是“实现部分”，两者各自为一维度，独立变化，仅此而已，所谓的“抽象与实现分离”更偏向于我们实际的程序开发，两者并不一定挂钩，这里其实就可以看到桥接模式的应用性其实很广泛，并不局限于程序设计。我们再来看 CoffeeAdditives 对应的两个子类：加糖与不加糖。

```
public class Sugar extends CoffeeAdditives {
    @Override
    public String addSomething() {
```

```
        return "加糖";
    }
}
public class Ordinary extends CoffeeAdditives {
    @Override
    public String addSomething() {
        return "原味";
    }
}
```

不加糖我们以原味表示，最后来看客户类，将两者进行整合。

```
public class Main {
    public static void main(String[] args) {
        // 原汁原味
        Ordinary implOrdinary = new Ordinary();

        // 准备糖类
        Sugar implSugar = new Sugar();

        // 大杯咖啡 原味
        LargeCoffee largeCoffeeOrdinary = new LargeCoffee(implOrdinary);
        largeCoffeeOrdinary.makeCoffee();

        // 小杯咖啡 原味
        SmallCoffee smallCoffeeOrdinary = new SmallCoffee(implOrdinary);
        smallCoffeeOrdinary.makeCoffee();

        // 大杯咖啡 加糖
        LargeCoffee largeCoffeeSugar = new LargeCoffee(implSugar);
        largeCoffeeSugar.makeCoffee();

        // 小杯咖啡 加糖
        SmallCoffee smallCoffeeSugar = new SmallCoffee(implSugar);
        smallCoffeeSugar.makeCoffee();
    }
}
```

代码逻辑很简单，不难理解，这里输出结果就不给出了。如果此时咖啡厅为了满足更多人的习惯，推出中杯的咖啡怎么办呢？对应于本例来说，这种需求的变化其实就是 Coffee 类的变化，定义中杯类扩展 Coffee 类即可。

```
public class MiddleCoffee extends Coffee {
    public MiddleCoffee(CoffeeAdditives impl) {
        super(impl);
    }

    @Override
    public void makeCoffee() {
        System.out.println("中杯的" + impl + "咖啡");
    }
}
```

对应的客户类做出相应的增加即可。

```
public class Main {
    public static void main(String[] args) {
        // 原汁原味
        Ordinary implOrdinary = new Ordinary();

        // 准备糖类
```

```
        Sugar implSugar = new Sugar();

        // ..............................................

        // 中杯咖啡 原味
        MiddleCoffee middleCoffeeOrdinary = new MiddleCoffee(implOrdinary);
        middleCoffeeOrdinary.makeCoffee();

        // 中杯咖啡 加糖
        MiddleCoffee middleCoffeeSugar = new MiddleCoffee(implSugar);
        middleCoffeeSugar.makeCoffee();
    }
}
```

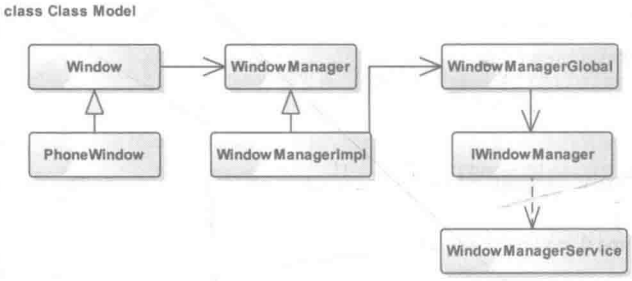
同样地，为了增加咖啡口味的种类，我们也可以让 CoffeeAdditives 类变化起来，增加更多的子类表示，诸如加奶、加蜂蜜、加盐等，具体的代码就不再给出了，相信大家已经很清楚。从本例我们可以看到，不管是 Coffee 变化了还是 CoffeeAdditives 变化了，其相对于对方而言都是独立的没有什么过多的交集，两者之间唯一的联系就是 Coffee 中保持的对 CoffeeAdditives 的引用，此乃两者之纽带，这就是桥接模式。

24.6 Android 源码中的桥接模式实现

桥接模式在 Android 中应用得相当广泛，但一般而言都是作用于大范围的，我们可以在源码中很多地方看到桥接模式的应用，比如在 View 的视图层级中，CheckBox、CompoundButton、Button、TextView 和 View 之间构成一个继承关系的视图层级，每一层视图都仅仅是对一种类型控件的描述，其定义了该类控件所拥有的基本属性和行为，但是将它们真正绘制到屏幕的部分是由与 View 相关的功能实现类 DisplayList、HardwareLayer 和 Canvas 负责。这两部分之间的关系可以看作是对桥接模式的应用。

除此之外，Adapter 与 AdapterView 之间也可以看作是对桥接模式的应用，有兴趣的同学可以自行研究。

以上两个应用方面相信大家在平时开发时也有使用到，只是并未将对其的认识上升到设计模式的层面，那么在 framework 内部的源码实现中，有没有对桥接模式比较经典的应用呢？答案是肯定的，而且还非常多，比较典型的是 Window 与 WindowManager 之间的关系，如图 24-2 所示。



▲图 24-2

如图 24-2 所示，在 framework 中 Window 和 PhoneWindow 构成窗口的抽象部分，其中 Window 类为该抽象部分的抽象接口，PhoneWindow 为抽象部分具体的实现及扩展。而 WindowManager 则为实现部分的基类，WindowManagerImpl 为实现部分具体的逻辑实现，其使用 WindowManagerGlobal 通过 IWindowManager 接口与 WindowManagerService（也就是我们常说的 WMS）进行交互，并由 WMS 完成具体的窗口管理工作，如下为 Window 与 WindowManager 桥梁搭建的主要代码，这在第 3 章的时候我们也有所提及故不再说明。

```
public abstract class Window {
    // ......省去一些代码......

    public void setWindowManager(WindowManager wm, IBinder appToken, String appName) {
        // 默认关闭硬件，加速调用重载方法
        setWindowManager(wm, appToken, appName, false);
    }

    public void setWindowManager(WindowManager wm, IBinder appToken, String appName,
            boolean hardwareAccelerated) {
        // ......省去 3 行代码......

        /*
         * 如果传入的 WindowManager 为空，则通过 getSystemService 获取 WindowManager
         */
        if (wm == null) {
            wm = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE);
        }

        // 将 WindowManager 和 Window 绑定
        mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);
    }

    // ......省去一些代码......
}
```

源码中更多关于桥接模式的应用就不再多说了，有兴趣的话大家可以自行查询，会发现很多地方其实都应用到了桥接模式。

24.7 关于 WindowManagerService

上述对源码中桥接模式的应用分析时，我们提到 WindowManagerService（下称 WMS）这个系统服务，可以毫不夸张地说，Android 的 framework 层主要就是由它与另外一个系统服务 ActivityManagerService（简称 AMS）还有 View 所构成，这 3 个模块穿插交互在整个 framework 中，掌握了它们之间的关系以及每一个步骤的逻辑，你对 framework 就至少了解百分之五十了。

和很多其他的系统服务一样，WMS 也是由 SystemServer 启动。

```
public void initAndLoop() {
    // 通过 WMS 的静态方法 main 获取一个 WindowManagerService 对象
    wm = WindowManagerService.main(context, power, display, inputManager, wmHandler,
factoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL, !firstBoot, onlyCore);

    // 将 WMS 添加至 ServiceManager 中
    ServiceManager.addService(Context.WINDOW_SERVICE, wm);
}
```

此后其他进程就可以通过用 ServiceManager 查询“window”来获取 WMS。WMS 的 mian 方法很简单，通过 Handler 的 runWithScissors 方法执行一个特殊的同步 Task 并在其中构造 WMS 的实例。

```
public static WindowManagerService main(final Context context,
        final PowerManagerService pm, final DisplayManagerService dm,
        final InputManagerService im, final Handler wmHandler,
        final boolean haveInputMethods, final boolean showBootMsgs,
        final boolean onlyCore) {
    final WindowManagerService[] holder = new WindowManagerService[1];

    // 执行同步 Task
    wmHandler.runWithScissors(new Runnable() {
        @Override
        public void run() {
            holder[0] = new WindowManagerService(context, pm, dm, im,
            haveInputMethods, showBootMsgs, onlyCore);
        }
    }, 0);
    return holder[0];
}
```

WMS 的构造方法逻辑不算复杂，大部分是对一些窗口管理将要使用到的成员变量进行初始化。

```
private WindowManagerService(Context context, PowerManagerService pm,
        DisplayManagerService displayManager, InputManagerService inputManager,
        boolean haveInputMethods, boolean showBootMsgs, boolean onlyCore) {
    // 一些成员变量赋值
    mContext = context;
    mHaveInputMethods = haveInputMethods;
    mAllowBootMessages = showBootMsgs;
    mOnlyCore = onlyCore;
    mLimitedAlphaCompositing = context.getResources().getBoolean(com.android.internal.
    R.bool.config_sf_limitedAlpha);
    mInputManager = inputManager;
    createDisplayContentLocked.mDisplayManagerService = displayManager;

    // ......省略一些代码......

    // 获取显示服务
    mDisplayManager = (DisplayManager)context.getSystemService(Context.DISPLAY_SERVICE);

    // 将 WMS 注册到显示服务中
    mDisplayManager.registerDisplayListener(this, null);

    // 为每一个 Display 分配一个 Content
    Display[] displays = mDisplayManager.getDisplays();
    for (Display display : displays) {
        createDisplayContentLocked(display);
    }

    // ......省略几行代码......

    // 构造 APP 事务对象
    mAppTransition = new AppTransition(context, mH);

    // 获取 IActivityManager 对象
    mActivityManager = ActivityManagerNative.getDefault();

    // ......省略一些代码......
```

```
    // 构造 Window 动画对象
    mAnimator = new WindowAnimator(this);

    // 初始化窗口管理策略
    initPolicy(UiThread.getHandler());

    // 开启绘制 Surface 事务
    SurfaceControl.openTransaction();

    // ......省略几行代码......
}
```

WMS 主要功能可以分为两方面，一是对窗口的管理；二是对事件的管理和分发。其接口方法以 AIDL 的方式定义在 IWindowManager.aidl 文件中，编译后会生成一个 IWindowManager.java 接口文件，这个接口文件定义了 WMS 绝大部分的功能方法，大家有兴趣可自行查看。作为窗口的管理承担者，WMS 中定义了许多各种不同的窗口，它们都被定义在 WMS 的成员变量中。

```
public class WindowManagerService extends IWindowManager.Stub implements Watchdog.Monitor,
WindowManagerPolicy.WindowManagerFuncs,      DisplayManagerService.WindowManagerFuncs,
DisplayManager.DisplayListener {
    // 已经完成启动的应用
    final ArrayList<AppWindowToken> mFinishedStarting = new ArrayList<AppWindowToken>();

    // 渐变窗口
    final ArrayList<FakeWindowImpl> mFakeWindows = new ArrayList<FakeWindowImpl>();

    // 尺寸正在改变的窗口
    final ArrayList<WindowState> mResizingWindows = new ArrayList<WindowState>();

    // 动画结束的窗口
    final ArrayList<WindowState> mPendingRemove = new ArrayList<WindowState>();

    // 即将释放 Surface 的窗口
    final ArrayList<WindowState> mDestroySurface = new ArrayList<WindowState>();

    // 失去焦点的窗口
    ArrayList<WindowState> mLosingFocus = new ArrayList<WindowState>();

    // 为了释放内存需要强制关闭的窗口
    ArrayList<WindowState> mForceRemoves;

    // 等待绘制的窗口
    ArrayList<Pair<WindowState, IRemoteCallback>> mWaitingForDrawn = new ArrayList<Pair<
    WindowState, IRemoteCallback>>();

    // 正处于动画中但需要 Relayout 的窗口
    final ArrayList<WindowState> mRelayoutWhileAnimating = new ArrayList<WindowState>();

    // 正在打开的应用
    final ArrayList<AppWindowToken> mOpeningApps = new ArrayList<AppWindowToken>();

    // 正在关闭的应用
    final ArrayList<AppWindowToken> mClosingApps = new ArrayList<AppWindowToken>();

    // 当前获得焦点的窗口
    WindowState mCurrentFocus = null;

    // 上一个获得焦点的窗口
    WindowState mLastFocus = null;
```

```
    // 输入法窗口下方的窗口
    WindowState mInputMethodTarget = null;

    // 输入法窗口
    WindowState mInputMethodWindow = null;

    // 墙纸窗口
    WindowState mWallpaperTarget = null;

    // 在切换过程中位于墙纸 Z-Order 下方的窗口
    WindowState mLowerWallpaperTarget = null;

    // 在切换过程中位于墙纸 Z-Order 上方的窗口
    WindowState mUpperWallpaperTarget = null;

    // 提示内存泄漏的警告窗口
    StrictModeFlash mStrictModeFlash;

    // 得到焦点的应用
    AppWindowToken mFocusedApp = null;
}
```

WMS 维护上述的各个成员变量值，可以看到大量线性表的应用，不同的窗口或同一个窗口在不同的状态阶段有可能位于不同的表中。虽然窗口的状态种类繁多，但是，对于 Android 来讲窗口的类型主要只有两种，一种是应用窗口，我们常见的 Activity 所处的窗口、应用对话框窗口、应用弹出窗口等都属于该类，与应用窗口相关的 Window 类主要是 PhoneWindow，其主要应用于手机，PhoneWindow 继承于 Window，其核心是 DecorView，应用窗口的添加主要就是通过 WindowManager 的 addView 方法将一个 DecorView 添加至 WindowManager 中，这部分内容我们在第 3 章也有所提及，这里就不再多说。另一种则是系统窗口，常见的屏幕顶部的状态栏、底部的导航栏、桌面窗口等都是系统窗口，系统窗口没有针对性的封装类，只需直接通过 WindowManager 的 addView 方法将一个 View 添加至 WindowManager 中即可，以屏幕顶部的状态栏为例，其添加逻辑在 PhoneStatusBar.java 的 addStatusBarWindow 方法中。

```
public class PhoneStatusBar extends BaseStatusBar implements DemoMode {
    @Override
    public void createAndAddWindows() {
        addStatusBarWindow();
    }

    private void addStatusBarWindow() {
        // 获取状态栏高度
        final int height = getStatusBarHeight();

        // 根据各种标识值构造布局参数
        final WindowManager.LayoutParams lp = new WindowManager.LayoutParams(
                ViewGroup.LayoutParams.MATCH_PARENT,
                height,
                WindowManager.LayoutParams.TYPE_STATUS_BAR,
                WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE
                    | WindowManager.LayoutParams.FLAG_TOUCHABLE_WHEN_WAKING
                    | WindowManager.LayoutParams.FLAG_SPLIT_TOUCH
                    | WindowManager.LayoutParams.FLAG_WATCH_OUTSIDE_TOUCH,
                PixelFormat.TRANSLUCENT);
        lp.flags |= WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED;
        lp.gravity = getStatusBarGravity();
```

```
        lp.setTitle("StatusBar");
        lp.packageName = mContext.getPackageName();

        // 构造具体的状态栏 View，也就是下面会用到的 mStatusBarWindow，其本质是一个 FrameLayout
        makeStatusBarView();

        // 将状态栏 View 添加至 WindowManager
        mWindowManager.addView(mStatusBarWindow, lp);
    }
}
```

前面我们说过，WMS 和其他系统服务一样也是由 SystemServer 启动，其也运行在系统进程里，当一个应用需要创建窗口时如何通过 IPC 请求 WMS 生成一个窗口，尔后再由 WMS 向应用返回和窗口交互的消息。第 3 章中我们在深入了解 WindowManager 时曾说到上述代码中的 addView 实质上是由 WindowManagerGlobal 的 addView 方法实现具体逻辑，辗转多次后最终会调用到 ViewRootImpl 的 setView 方法，在该方法中通过 addToDisplay 方法向 WMS 发起一个 Session 请求，这里要注意的是，IWindowSession 也是一个 AIDL 接口文件，需要将其编译后才生成 IWindowSession.java 接口，这里 addToDisplay 方法最终会调用到 Session 中的对应方法：

```
final class Session extends IWindowSession.Stub
        implements IBinder.DeathRecipient {
        @Override
    public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs,
            int viewVisibility, int displayId, Rect outContentInsets,
            InputChannel outInputChannel) {
        return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId,
                outContentInsets, outInputChannel);
    }
}
```

最终，我们还是回到了 WMS 中与其建立连接，如上代码所述，addToDisplay 方法最终返回 WMS 中 addWindow 所返回的结果，在 addWindow 方法中，WMS 处理了具体的窗口添加逻辑，首先是对权限的检查，区分系统权限与应用权限。

```
public int addWindow(Session session, IWindow client, int seq, WindowManager.LayoutParams attrs,
int viewVisibility, int displayId, Rect outContentInsets, InputChannel outInputChannel) {
        int[] appOp = new int[1];
        int res = mPolicy.checkAddPermission(attrs, appOp);
        if (res != WindowManagerGlobal.ADD_OKAY) {
            return res;
    }
    // ......省略一些代码......
}
```

注意，这里的 mPolicy 实质是一个 PhoneWindowManager，checkAddPermission 方法对权限做具体的检查，其逻辑比较简单，先判断窗口类型是否系统级的，如果不是系统级则返回一个 ADD_OKAY 值表示允许，否则就需要 SYSTEM_ALERT_WINDOW 或 INTERNAL_SYSTEM_WINDOW 权限。权限检查后则对相关的 Display 显示信息以及窗口信息进行校对，然后在 WMS 中获取对应的 WindowToken，再根据不同的窗口类型检查窗口的有效性。

```
public int addWindow(Session session, IWindow client, int seq, WindowManager.LayoutParams
attrs, int viewVisibility, int displayId, Rect outContentInsets, InputChannel
outInputChannel) {
        int[] appOp = new int[1];
```

```
    // ......省略一些代码......
    synchronized(mWindowMap) {
        boolean addToken = false;
        WindowToken token = mTokenMap.get(attrs.token);
        if (token == null) {
            // 如果窗口是子窗口
            if (type >= FIRST_APPLICATION_WINDOW && type <= LAST_APPLICATION_WINDOW) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
            // 如果窗口是输入法窗口
            if (type == TYPE_INPUT_METHOD) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
            // 如果窗口是墙纸窗口
            if (type == TYPE_WALLPAPER) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
            // 如果窗口是锁屏后显示的 DayDream 窗口
            if (type == TYPE_DREAM) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
            // 构造 WindowToken 对象
            token = new WindowToken(this, attrs.token, -1, false);
            addToken = true;
        } else if(type>=FIRST_APPLICATION_WINDOW && type <= LAST_APPLICATION_WINDOW) {
            // 获取应用的 AppWindowToken
            AppWindowToken atoken = token.appWindowToken;
            if (atoken == null) {
                return WindowManagerGlobal.ADD_NOT_APP_TOKEN;
            } else if (atoken.removed) {
                // 使用存在的 token 添加窗口
                return WindowManagerGlobal.ADD_APP_EXITING;
            }
            if (type == TYPE_APPLICATION_STARTING && atoken.firstWindowDrawn) {
                // 不能是启动窗口类型
                return WindowManagerGlobal.ADD_STARTING_NOT_NEEDED;
            }
        } else if (type == TYPE_INPUT_METHOD) {
            // 如果是输入法窗口，其 token 的 windowType 必须等于 TYPE_INPUT_METHOD
            if (token.windowType != TYPE_INPUT_METHOD) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
        } else if (type == TYPE_WALLPAPER) {
            // 如果是输入法窗口
            if (token.windowType != TYPE_WALLPAPER) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
        } else if (type == TYPE_DREAM) {
            // 如果是 DayDream 窗口
            if (token.windowType != TYPE_DREAM) {
                return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
            }
        }
        // ......省略一些代码......
    }
    // ......省略几行代码......
}
```

窗口有效性检查完后，就会为该窗口创建一个 WindowState 对象类，维护窗口的状态以及根据适当的机制来调整窗口的状态。

```
public int addWindow(Session session, IWindow client, int seq, WindowManager.LayoutParams
attrs, int viewVisibility, int displayId, Rect outContentInsets, InputChannel
outInputChannel) {
        int[] appOp = new int[1];
    // ......省略一些代码......
    synchronized(mWindowMap) {
        // ......省略一些代码......

        // 构造 WindowState 对象
        win = new WindowState(this, session, client, token,
                attachedWindow, appOp[0], seq, attrs, viewVisibility, displayContent);

        // 如果客户端已经被销毁
        if (win.mDeathRecipient == null) {
            return WindowManagerGlobal.ADD_APP_EXITING;
        }

        // 根据窗口类型调整属性
        mPolicy.adjustWindowParamsLw(win.mAttrs);

        // 设置该窗口是否只对自己的 UID 可见
        win.setShowToOwnerOnlyLocked(mPolicy.checkShowToOwnerOnly(attrs));

        // 根据窗口类型判定权限
        res = mPolicy.prepareAddWindowLw(win, attrs);
        if (res != WindowManagerGlobal.ADD_OKAY) {
            return res;
        }

        // 如果输出 Channel 的读通道为空
        if (outInputChannel != null && (attrs.inputFeatures
                & WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANNEL) == 0) {
            // 则创建通道
            String name = win.makeInputChannelName();
            InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
            win.setInputChannel(inputChannels[0]);
            inputChannels[1].transferTo(outInputChannel);

            // 向 InputManager 中注册该通道，以便当前窗口可以接收到事件
            mInputManager.registerInputChannel(win.mInputChannel,
            win.mInputWindowHandle);
        }
        // ......省略一些代码......
    }
    // ......省略几行代码......
}
```

上面我们说过，这里的 mPolicy 实质上就是一个 PhoneWindowManager 对象，这里调用了其 adjustWindowParamsLw 方法，该方法的逻辑很简单，只要是 TYPE_SYSTEM_OVERLAY 或 TYPE_SECURE_SYSTEM_OVERLAY 类型的窗口不要其获取焦点即可。最后将窗口与其他相关组件关联。

```
public int addWindow(Session session, IWindow client, int seq, WindowManager.LayoutParams
attrs, int viewVisibility, int displayId, Rect outContentInsets, InputChannel
outInputChannel) {
        int[] appOp = new int[1];
    // ......省略一些代码......
    synchronized(mWindowMap) {
        // ......省略一些代码......
        res = WindowManagerGlobal.ADD_OKAY;
```

```
// 重置当前线程 IPC 的 ID
origId = Binder.clearCallingIdentity();

// 是否需要添加 Token
if (addToken) {
    mTokenMap.put(attrs.token, token);
}

// 将窗口添加至 Session
win.attach();

// 将窗口添加至 WMS
mWindowMap.put(client.asBinder(), win);

// ......省略几行代码......

boolean imMayMove = true;

// 顺序排列窗口
if (type == TYPE_INPUT_METHOD) {
    // 如果是输入法窗口
    win.mGivenInsetsPending = true;
    mInputMethodWindow = win;
    addInputMethodWindowToListLocked(win);
    imMayMove = false;
} else if (type == TYPE_INPUT_METHOD_DIALOG) {
    // 如果是输入法对话框窗口
    mInputMethodDialogs.add(win);
    addWindowToListInOrderLocked(win, true);
    moveInputMethodDialogsLocked(findDesiredInputMethodWindowIndexLocked(true));
    imMayMove = false;
} else {
    // 如果是其他类型的窗口
    addWindowToListInOrderLocked(win, true);
    if (type == TYPE_WALLPAPER) {
        mLastWallpaperTimeoutTime = 0;
        displayContent.pendingLayoutChanges |= FINISH_LAYOUT_REDO_WALLPAPER;
    } else if ((attrs.flags&FLAG_SHOW_WALLPAPER) != 0) {
        displayContent.pendingLayoutChanges |= FINISH_LAYOUT_REDO_WALLPAPER;
    } else if (mWallpaperTarget != null
            && mWallpaperTarget.mLayer >= win.mBaseLayer) {
        displayContent.pendingLayoutChanges |= FINISH_LAYOUT_REDO_WALLPAPER;
    }
}
win.mWinAnimator.mEnterAnimationPending = true;

// 如果 displayContent 是默认的显示
if (displayContent.isDefaultDisplay) {
    // 则获取系统窗口区域的 insets
    mPolicy.getContentInsetHintLw(attrs, outContentInsets);
} else {
    // 否则将 outContentInsets 设置为空
    outContentInsets.setEmpty();
}

if (mInTouchMode) {
    // 标识用户直接触摸的窗口
    res |= WindowManagerGlobal.ADD_FLAG_IN_TOUCH_MODE;
}
if (win.mAppToken == null || !win.mAppToken.clientHidden) {
    // 标识应用窗口
```

```
            res |= WindowManagerGlobal.ADD_FLAG_APP_VISIBLE;
        }
        mInputMonitor.setUpdateInputWindowsNeededLw();

        boolean focusChanged = false;

        // 如果当前窗口可以接收按键事件
        if (win.canReceiveKeys()) {
            // 那么更新焦点，将窗口信息存入 InputDispatcher
            focusChanged=updateFocusedWindowLocked(UPDATE_FOCUS_WILL_ASSIGN_LAYERS,false);
            if (focusChanged) {
                imMayMove = false;
            }
        }
        if (imMayMove) {
            moveInputMethodWindowsIfNeededLocked(false);
        }

        // 分配最终层级值
        assignLayersLocked(displayContent.getWindowList());

        if (focusChanged) {finishUpdateFocusedWindowAfterAssignLayersLocked(false);
        }
        mInputMonitor.updateInputWindowsLw(false);

        if (win.isVisibleOrAdding() && updateOrientationFromAppTokensLocked(false)) {
            reportNewConfig = true;
        }
    }
    if (reportNewConfig) {
        sendNewConfiguration();
    }

    Binder.restoreCallingIdentity(origId);
    return res;
}
```

在 addWindow 方法最后的这一部分逻辑中，最重要的一步是通过 WindowState 的 attach 方法创建了一个关联 SurfaceSession 的对象用以与 SurfaceFlinger 通信，而 SurfaceSession 实际对应的是 native 层中的 SurfaceComposerClient 对象，SurfaceComposerClient 主要作用就是在应用进程与 SurfaceFlinger 服务之间建立连接，具体这部分的代码大家可以查阅相关资料，这里就不再多介绍了。以上就是一个系统窗口添加的主要逻辑。

24.8 桥接模式实战

上面我们曾经说过 View 的视图层级与执行真正的硬件绘制相关类之间的关系可以看作是一种桥接模式，其实这里也可以模仿这种行为来让我们的自定义控件以桥接的方式提供多种不同的实现机制，逻辑很简单，这里以进度条为例，Android 内部提供给的进度条样式就那么几种而且还都不好看，鉴于进度条的逻辑并不复杂，我们可以自己继承 View 类来实现进度条控件，这里就以自定义水平、垂直和圆形 3 种不同的进度条为例，为了后期扩展，我们以一个实现类来声明不同进度条需要实现的方法。

```
public abstract class BaseProgressBar {
```

```
    public static final int HORIZONTAL = 0;// 水平进度条标识
    public static final int VERTICAL = 1;// 垂直进度条标识
    public static final int CIRCLE = 2;// 圆形进度条标识

    protected Paint mPaint;

    /**
     * 构造方法内完成一些具体的初始化信息
     */
    protected BaseProgressBar() {
        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
    }

    /**
     * 获取测量高度，由具体子类实现
     *
     * @return 测量高度
     */
    public abstract int getMeasureHeight();

    /**
     * 获取测量宽度，由具体子类实现
     *
     * @return 测量宽度
     */
    public abstract int getMeasureWidth();

    /**
     * 具体的绘制操作，由具体子类实现
     *
     * @param view   View
     * @param canvas Canvas
     */
    public abstract void draw(View view, Canvas canvas);
}
```

然后对于每一种进度条我们给予其具体的实现，这里因篇幅有限，具体的实现逻辑就不再给出。

```
public class HorizontalProgressBar extends BaseProgressBar {
    @Override
    public int getMeasureHeight() {
        return 50;
    }

    @Override
    public int getMeasureWidth() {
        return 1000;
    }

    @Override
    public void draw(View view, Canvas canvas) {
        // 这里就可以执行具体的绘制水平进度条的操作
    }
}

public class VerticalProgressBar extends BaseProgressBar {
    @Override
    public int getMeasureHeight() {
        return 50;
    }

    @Override
```

```
    public int getMeasureWidth() {
        return 1000;
    }

    @Override
    public void draw(View view, Canvas canvas) {
        // 这里就可以执行具体的绘制垂直进度条的操作
    }
}

public class CircleProgressBar extends BaseProgressBar {
    @Override
    public int getMeasureHeight() {
        return 500;
    }

    @Override
    public int getMeasureWidth() {
        return 500;
    }

    @Override
    public void draw(View view, Canvas canvas) {
        // 这里就可以执行具体的绘制圆形进度条的操作
    }
}
```

然后我们需要一个继承于 View 的进度条类来分派绘制逻辑。

```
public class ProgressBar extends View {
    private static final int HEIGHT = 0x6846, WIDTH = 0x7889;

    private BaseProgressBar mBaseProgressBar;

    private int style = BaseProgressBar.CIRCLE;

    public ProgressBar(Context context) {
        this(context, null);
    }

    public ProgressBar(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public ProgressBar(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
        TypedArray a = context.obtainStyledAttributes(attrs,
        R.styleable.ProgressBar, defStyleAttr, 0);
        style = a.getInt(R.styleable.ProgressBar_style, BaseProgressBar.CIRCLE);

        if (style == BaseProgressBar.HORIZONTAL) {
            mBaseProgressBar = new HorizontalProgressBar();
        } else if (style == BaseProgressBar.VERTICAL) {
            mBaseProgressBar = new VerticalProgressBar();
        } else {
            mBaseProgressBar = new CircleProgressBar();
        }
        a.recycle();
    }

    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
```

```
        setMeasuredDimension(measureDimension(WIDTH, widthMeasureSpec),
        measureDimension(HEIGHT, heightMeasureSpec));
    }

    private int measureDimension(int type, int measureSpec) {
        if (style == BaseProgressBar.CIRCLE) {
            return mBaseProgressBar.getMeasureWidth();
        }

        int result;

        int mode = MeasureSpec.getMode(measureSpec);
        int size = MeasureSpec.getSize(measureSpec);

        if (mode == MeasureSpec.EXACTLY) {
            result = size;
        } else {
            if (type == HEIGHT) {
                result = mBaseProgressBar.getMeasureHeight();
            } else {
                result = mBaseProgressBar.getMeasureWidth();
            }
            if (mode == MeasureSpec.AT_MOST) {
                result = Math.min(result, size);
            }
        }
        return result;
    }

    @Override
    protected void onDraw(Canvas canvas) {
        mBaseProgressBar.draw(this, canvas);
    }
}
```

逻辑很简单，这里的 ProgressBar 类持有 BaseProgressBar 类的引用，通过在不同的绘制阶段调用不同进度条实现类的相应方法完成具体的逻辑。如本文所说，桥接模式的应用其实非常的广泛，除了上面我们提到的控件例子外，在进行数据库 dao 类设计时我们有时也会需要使用到桥接模式。而且对于 Android 来说，应用层与 Native 层之间的交互就是一个最好的例子，在需要操纵 Android 设备硬件时就需要使用一个连接应用层与 Native 的桥梁，这个桥梁通常是一个具体的类，比如提供操作相机的 Camera、播放音视频的 MediaPlayer、提供图形绘制接口的 OpenCV 等，这些 API 类为我们操作底层硬件提供了可能。

24.9 总结

桥接模式可以应用到许开发中，但是它应用得却不多，一个很重要的原因是对于抽象与实现的分离的把握，是不是需要分离、如何分离？对设计者来说要有一个恰到好处的分寸。不管怎么说，桥接模式的优点我们毋庸置疑，分离抽象与实现、灵活的扩展以及对客户来说透明的实现等。但是使用桥接模式也有一个不明显的缺点，上面我们也提到了，就是不容易设计，对开发者来说要有一定的经验要求。因此，对桥接模式应用来说，理解很简单，设计却不容易。

第 25 章 MVC 的介绍与实战

25.1 MVC 的起源与历史

MVC 全称是 Model-View-Controller 也就是模型——视图——控制器，在 1970 年的时候由 TrygveReenskaug 在 Smalltalk-80 系统上首次提出，最开始的时候并不是叫 MVC，而是 MVCE，也就是在 Model——View——Controller 中多了一个额外的 Editor，最初的 MVC 理念与现在的其实相差不大，其目的都是为了将数据模型和视图分离开来，并以控制器作为连接两者的桥梁以实现解耦，但是，那时候的 MVC 实现与现在有很大的差别，因为在当时那个年代计算机还只是个只会显示一行行代码的命令机器，那时还没有 GUI 的概念，因此，那个年代 MVC 刚提出的时候其实现是以 Model 为中心的，当 Model 发生改变后 View 和 Controller 都会接到 Model 改变的通知。后来随着计算机的发展，GUI 图形化界面开始出现，MVC 的实现方式也随之改变，但不管怎么改变都不会影响到 Model——View——Controller 这样的框架结构。

MVC 是一种框架模式而非设计模式，GOF 把 MVC 看作是 3 种设计模式：观察者模式、策略模式与组合模式的合体，而且其核心在观察者模式，也就是一个基于发布/订阅者模型的框架，很多时候在实际开发过程中我们常常还会在 MVC 框架中使用到其他的设计模式。那么框架模式与设计模式的区别又在何处呢？做开发的读者经常会听到这样或那样的框架，比如注解框架、通信框架、数据存储框架等通用性较强的框架，还有一些面向后端开发的，如 Struts、Spring 等框架，在这些框架中你常常会见到很多设计模式的应用。对框架来说，通常是对代码的重用，而对设计来说通常是对设计的重用，可能会比较拗口，你可以简单地这样理解框架面向于一系列相同行为代码的重用，而设计则面向的是一系列相同结构代码的重用，我们平常所说的架构则介于框架与设计之间。在软件开发领域有 3 种级别的重用：内部重用，即在同一应用中能公共使用的抽象块；代码重用，即将通用模块组合成库或工具集，以便在多个应用和领域都能使用；应用框架的重用，即为专用领域提供通用的或现成的基础结构，以获得最高级别的重用性。我们平时开发过程中比较常见的框架模式除了 MVC 以外还有 MVVC、MTV、CBD、ORM 以及第 26 章会讲到的 MVP。简而言之：框架是大智慧，用来对软件设计进行分工；设计模式是小技巧，对具体问题提出解决方案，以提高代码复用率，降低耦合度。

MVC 的优点很多，首先第一个就是理解起来比较容易，技术含量并不高，这对开发和维护来说成本较低也易于维护和修改，其次则是耦合性不高，表现层与业务层分离实现各司其职，对开发

来说也是很有利，前端工程师专注界面的研发，后端工程师则致力于业务逻辑的完善等。而对于 MVC 的缺点也有很多，由于它没有明确的定义，所以完全理解 MVC 模式并不是很容易。使用 MVC 模式需要精心计划，由于它的内部原理比较复杂，所以需要花费一些时间去思考。开发一个 MVC 模式架构的工程，将不得不花费相当可观的时间去考虑如何将 MVC 模式运用到应用程序中，同时由于 Model 和 View 要严格分离，这样也给调试应用程序带来了一定的困难。每个构件在使用之前都需要经过彻底的测试。另外，由于 MVC 模式将一个应用程序分成了 3 个部件，所以，这意味着同一个工程将包含比以前更多的文件。因此，对于一些小规模的项目，MVC 反而会带来更大的工作量以及复杂性。

25.2 MVC 在 Android 中的实现

Android 中对 MVC 的应用很经典，因为对 Android 本身来说，其界面部分的开发就涉及了模型——视图——控制器 3 者的交互，在 Android 中视图 View 层一般采用 XML 文件进行界面的描述，比如以下一段比较常见的 XML 布局文件代码。

```
<RelativeLayoutxmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:background="#FFF"
android:gravity="center"
android:padding="16dp">
</RelativeLayout>
```

而对于模型 Model 部分则大多对应于本地的数据文件或网络获取的数据体，很多情况下我们对这些数据的处理也会在这一层中进行。最后的控制器 Controller 部分则当之无愧地由 Activity 承担。

```
public class MainActivity extends Activity {

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

一般情况下会在 Activity 中获取数据以及界面元素，并将两者进行绑定，但其逻辑不能过于复杂，因为 Android 中规定一个 Activity 的响应时间是 5 秒，如果超过这个时间就有可能会被回收掉。

下面是一个简单的 Activty 加载数据并显示的例子，大家在应用开发中应该也见过不少这种情况，这里不再累赘述。

```
public class MVCActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // 设置视图层
        ListView lvContent = newListView(this);
        lvContent.setBackgroundColor(Color.WHITE);
        setContentView(lvContent);
```

```
        // 获取数据
        String[] data = getResources().getStringArray(R.array.data);

        // 数据绑定
        ArrayAdapter adapter = new ArrayAdapter(this, R.layout.activity_mvc_item, data);
        lvContent.setAdapter(adapter);
    }
}
```

数据部分：

```
<?xmlversion="1.0"encoding="utf-8"?>
<resources>
<string-arrayname="data">
<item>SM Brother</item>
<item>Aige</item>
<item>Mermaid Tank</item>
<item>Aunt Guo</item>
<item>Shit Brother</item>
<item>Doctor Xu</item>
<item>Uncle Seven</item>
<item>Very Sharp</item>
</string-array>
</resources>
```

这里的视图部分应该是 ListView 与其 Item 布局共同负责，下面是 Item 的布局。

```
<TextViewxmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:padding="15dp"
android:textColor="#000"/>
```

运行效果如图 25-1 所示。

▲图 25-1

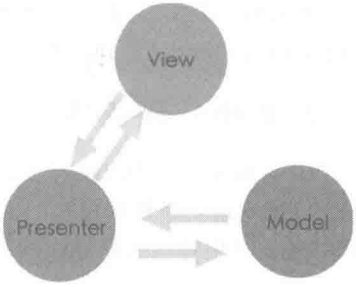
25.3 总结

从上面的小例子里我们可以看到，在 Android 的 UI 系统中，控制器 Activity 主要起到的作用就是解耦，将视图 View 和模型 Model 进行分离，两者在 Activity 中进行绑定或完成其他逻辑。总体来说，还是像本文开头所说的那样，MVC 更适合于规模比较大的项目，比如 Android 的 UI 系统框架，虽说上面的小例子中我们切实地感受到了 Android 在 MVC 方面的结构，但是，这个框架并非我们自己完成的，而是由 framework 给我们搭建好并提供给我们的，在平时的开发中，特别是用 Android 开发，我们并不常用到 MVC 模式去脱离 Android UI 系统构建自己的框架结构。

第 26 章　MVP 应用架构模式

26.1 MVP 模式介绍

在上一章中我们学习了著名的 MVC 架构，本章我们来学习 MVC 的一个演化版本——MVP 模式，MVP 全称 Model View Presenter。目前 MVP 在 Android 应用开发中越来越重要了，大家也都在讨论关于 MVP 的理论，在 Android Weekly 中关于 MVP 的文章也逐渐增多，只是体系化的资料非常少。MVP 能够有效地降低 View 复杂性，避免业务逻辑被塞进 View 中，使得 View 变成一个混乱的“大泥坑”。MVP 模式会解除 View 与 Model 的耦合，同时又带来了良好的可扩展性、可测试性，保证了系统的整洁性、灵活性。可能对于简单的应用来说 MVP 稍显麻烦，各种各样的接口与概念，使得整个应用充斥着零散的接口，但是对于比较复杂的应用来说，MVP 模式是一种良好的架构模式，它能够非常好地组织应用结构，使得应用变得灵活，拥抱变化。MVP 的结构图如图 26-1 所示。



▲图 26-1

MVP 模式可以分离显示层和逻辑层，它们之间通过接口进行通信，降低耦合。理想化的 MVP 模式可以实现同一份逻辑代码搭配不同的显示界面，因为它们之间并不依赖于具体，而是依赖于抽象。这使得 Presenter 可以运用于任何实现了 View 逻辑接口的 UI，使之具有更广泛的适用性，保证了灵活度。

我们知道在 Android 上，业务逻辑和数据存取是紧耦合的，很多缺乏经验的工程师很可能会将各种各样的业务逻辑塞进某个 Activity、Fragment 或者自定义的 View 类中，使得这些组件的单个类型臃肿不堪，其中又含有一些异步任务，导致某个类超过千行代码。当然，对于功能复杂的应用来说，一个类超过千行代码并不是什么大惊小怪的事，我们所要指出的重点是业务逻辑与 View 元素严重地耦合在一起导致了类型膨胀的问题。

对于一个可扩展、稳定的应用来说，我们需要定义分离各个层，主要是 UI 层、业务逻辑层和数据层。毕竟，我们不知道以后还要加入什么逻辑，是从本地数据库检索数据？还是从远程的服务器中？我们的 UI、数据库是否会被替换？例如，随着产品的升级，我们的 UI 可能会被重新设计，若 UI 发生了变化，此时由于业务逻辑耦合在 View 中，UI 变化导致我们修改新的 View 控件，此时

你就需要到原来的 View 中抽离具体的业务逻辑，这将是一件非常折磨人也易于出错的事，到最终你还是需要将业务逻辑抽离开来。

MVP 模式可以让 UI 界面和数据分离，我们的应用至少可以分为 3 层，这样使得我们也可以对这 3 个层次进行独立的单元测试。MVP 模式可以让我们从 Activity、Fragment 等 View 角色中分离大部分代码，使得每个类型的代码量大幅度减少，职责单一，易于维护。

MVP 并不是一个标准化的模式，它有很多种实现方式，我们也可以根据自己的需求和自己认为对的方式去修正 MVP 的实现方式，它可以随着 Presenter 的复杂程度变化。只要保证我们是通过 Presenter 将 View 和 Model 解耦合、降低类型复杂度、各个模块可以独立测试、独立变化，这就是正确的方向。在 Android 开发中，大多数人可能会把 Activity、Fragment 作为 View 角色来看待，因为它的职责是加载并且处理一些简单的与 View 相关的逻辑，它组织与管理 View 集合，我们可以把它看成是粗粒度的 View，当然你也可以将它们看成 Presenter，后续我们会以一个示例进行说明。

26.2 MVP 模式的三个角色

1. Presenter——交互中间人

Presenter 主要作为沟通 View 和 Model 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，也将业务逻辑从 View 角色上抽离出来。

2. View——用户界面

View 通常是指 Activity、Fragment 或者某个 View 控件，它含有一个 Presenter 成员变量。通常 View 需要实现一个逻辑接口，将 View 上的操作通过会转交给 Presenter 进行实现，最后，Presenter 调用 View 逻辑接口将结果返回给 View 元素。

3. Model——数据的存取

对于一个结构化的 App 来说，Model 角色主要是提供数据的存取功能。Presenter 需要通过 Model 层存储、获取数据，Model 就像一个数据仓库。更直白地说，Model 是封装了数据库 DAO 或者网络获取数据的角色，或者两种数据获取方式的集合。

26.3 与 MVC、MVVM 的区别

MVC 和 MVVM 的结构如图 26-2 和图 26-3 所示。

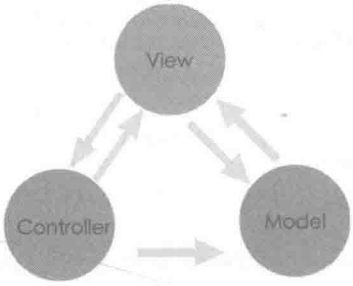
1. MVC 特点

（1）用户可以向 View 发送指令，再由 View 直接要求 Model 改变状态。

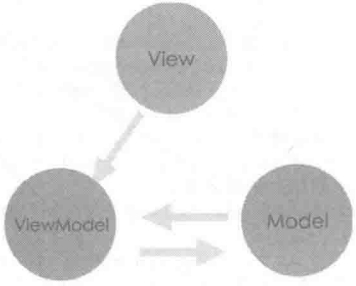
（2）用户也可以直接向 Controller 发送指令，再由 Controller 发送给 View。

（3）Controller 起到事件路由的作用，同时业务逻辑都部署在 Controller 中。

可以看到，MVC 的耦合性还是相对较高，View 可以直接访问 Model，导致 3 者之间构成回路。因此，MVP 与 MVC 的主要区别是，MVP 中的 View 不能直接访问 Model，需要通过 Presenter 发出请求，View 与 Model 不直接通信。



▲图 26-2 （MVC）



▲图 26-3 （MVVM）

2. MVVM 特点

MVVM 与 MVP 非常相似，唯一的区别是 View 和 Model 进行双向绑定（data-binding），两者之间有一方发生变化则会反应到另一方上。而 MVP 与 MVVM 的主要区别则是，MVP 中的 View 更新需要通过 Presenter，而 MVVM 则不需要，因为 View 与 Model 进行了双向绑定，数据的修改会直接反应到 View 角色上，而 View 的修改也会导致数据的变更。此时，ViewModel 角色需要做的只是业务逻辑的处理，以及修改 View 或者 Model 的状态。MVVM 模式有点像 ListView 与 Adapter、数据集的关系，这个 Adapter 就是 ViewModel 角色，它与 View 进行了绑定，又与数据集进行了绑定，当数据集合发生变化时，调用 Adapter 的 notifyDataSetChanged 之后 View 就直接更新，它们之间没有直接的耦合，使得 ListView 变得更为灵活。

26.4 MVP 的实现

下面我们通过一个简单的客户端示例来直观地体会 MVP 在开发中的运用。开发技术前线（网址：www.devtf.cn）是一个聚集了优质 Android、iOS 技术文章的网站，这个网站每天都会更新移动开发领域的优质技术文章。我们就以一个简单的客户端为例来讲述 MVP 的实战运用。我们的 UI 原型大致如图 26-4 所示。

▲图 26-4

进入应用之后，首先会从服务器端下拉最新的 20 篇文章，然后将每一篇文章的简介显示到列表上。当用户点击某项数据时进入到另一个页面，该页面加载这篇文章的详细内容。因此，我们的业务逻辑大概有下列 2 个。

（1）向服务器请求数据，并且将数据存储到数据库中。

（2）从数据库中加载文章列表。

我们的主界面（ArticlesActivity）就是一个 RecyclerView 和进度条，

在加载数据时显示进度条，加载完成之后隐藏，网络请求使用的是 Volley。我们先从 Presenter 相关的类型入手，用户需要从网络端获取文章，因此，需要一个数据获取接口；也可以从本地数据库获取缓存的数据，因此需要一个从数据库加载缓存的接口。这个 Presenter 我们命名为 ArticlePresenter，相关代码如下。

```
// Presenter，作为 View 和 Model 的中间人（中间人不就是中介么）
public class ArticlePresenter {
    // ArticleView 接口，代表了 View 接口角色
    ArticleViewInterface mArticleView;
    // 文章数据的 Model，也就是 Model 角色
    ArticleModel mArticleModel = new ArticleModelImpl();
    // 从网络上获取文章的 API
    ArticleAPI mArticleApi = new ArticleAPIImpl();

    public ArticlePresenter(ArticleViewInterface viewInterface) {
        mArticleView = viewInterface;
    }

    // 获取文章，也就是我们的业务逻辑
    public void fetchArticles() {
        mArticleView.showLoading();
        mArticleApi.fetchArticles(new DataListener<List<Article>>() {

            @Override
            public void onComplete(List<Article> result) {
                // 数据加载完，调用 View 的 showArticles 函数将数据传递给 View 显示
                mArticleView.showArticles(result);
                mArticleView.hideLoading();
                // 存储到数据库
                mArticleModel.saveArticles(result);
            }
        });
    }

    public void loadArticlesFromDB() {
        mArticleModel.loadArticlesFromCache(new DataListener<List<Article>>() {

            @Override
            public void onComplete(List<Article> result) {
                // 从数据库加载
                mArticleView.showArticles(result);
            }
        });
    }
}
```

在 ArticlePresenter 中持有了 View 和 Model 的引用，分别为 ArticleViewInterface 和 ArticleModelImpl，另外，还有一个操作网络请求的 ArticleAPIImpl 对象。ArticleViewInterface 就是主界面的逻辑接口，代表了 View 接口角色，用于 Presenter 回调 View 的操作。具体代码如下。

```
public interface ArticleViewInterface {
    public void showArticles(List<Article> articles);  // 展示数据
    public void showLoading();                         // 显示进度条
    public void hideLoading();                         // 隐藏进度条
}
```

ArticleModelImpl 则是对数据库的存取操作，用于保存网络上加载的数据，以及从数据库中加载

文章缓存。为了代码简单，我们这里模拟数据库的存取操作，将数据缓存到内存中。具体代码如下。

```
public class ArticleModelImpl implements ArticleModel {
    List<Article> mCachedArticles = new LinkedList<Article>();

    @Override
    public void saveArticles(List<Article> articles) {
        mCachedArticles.addAll(articles);
    }

    @Override
    public void loadArticlesFromCache(DataListener<List<Article>> listener) {
        listener.onComplete(mCachedArticles);
    }
}
```

ArticlesActivity 需要实现 ArticleViewInterface 接口，并且需要建立与 Presenter 的联系，ArticlesActivity 的业务逻辑都将交给 Presenter 进行处理，处理结果通过 ArticleViewInterface 接口回调给 ArticlesActivity 类。我们看看 ArticlesActivity 类的代码。

```
public class ArticlesActivity extends Activity implements ArticleViewInterface {
    RecyclerView mRecyclerView;
    ProgressBar mProgressBar;
    List<Article> mArticles = new LinkedList<Article>();
    ArticleAdapter mAdapter;
    ArticlePresenter mPresenter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 初始化各种控件
        initViews();
        // 初始化请求队列
        RequestQueueMgr.init(getApplicationContext());
        // 构建 ArticlePresenter，与 ArticlesActivity 建立关联
        mPresenter = new ArticlePresenter(this);
        // 请求文章数据
        mPresenter.fetchArticles();
    }

    private void initViews() {
        mRecyclerView = (RecyclerView) findViewById(R.id.articles_recycler_view);
        mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
        // 创建适配器
        mAdapter = new ArticleAdapter(mArticles);
        mAdapter.setOnItemClickListener(new OnItemClickListener() {
            @Override
            public void onClick(Article article) {
                // 处理文章的点击事件
                clickArticle(article);
            }
        });
        mRecyclerView.setAdapter(mAdapter);
        // 进度条
        mProgressBar = (ProgressBar) findViewById(R.id.loading_progressbar);
    }

    @Override
```

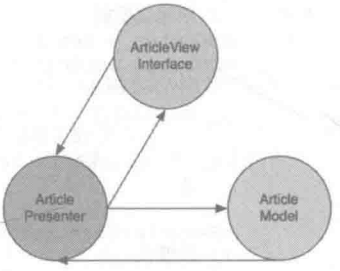
```
    public void showArticles(List<Article> result) {
        mArticles.addAll(result);                    // 更新数据源
        mAdapter.notifyDataSetChanged();             // 更新 UI
    }
    // 显示、隐藏进度条等其他代码省略
}
```

ArticlesActivity 实现了 ArticleViewInterface 接口，并且在 onCreate 函数中将自身传递给了 ArticlePresenter，此时作为 View 角色的 ArticlesActivity 就与 Presenter 建立了关联，而由于 ArticlePresenter 又有 ArticleModel 类型的成员变量，因此，Model-View-Presenter 的关系此时已经建立。

此时，我们就可以通过 Presenter 处理业务逻辑。例如，在 onCreate 函数的最后一句是调用 Presenter 的 fetchArticles 函数。该函数的作用就是从服务器上下拉最新的 20 篇文章，当请求成功之后，调用 ArticleViewInterface 的 showArticles 函数将数据传递给 View，也就是 ArticlesActivity 对象。因为 ArticlesActivity 实现了 ArticleViewInterface 接口，因此调用的就是 ArticlesActivity 类中的 showArticles 函数，在该函数中，我们将数据添加到 RecyclerView 的数据源中，并且通过 Adapter 通知 RecyclerView 更新 UI 界面。

通过这个用例我们看到，Presenter 对于 View 是完全解耦合的，Presenter 依赖的是 ArticleViewInterface 抽象，而不是 ArticlesActivity 这个类，这样当 UI 界面发生变化时，只需要新的 UI 实现了 ArticleViewInterface 以及相关的逻辑即可与 Presenter 迅速地协作起来，成本非常低。而由于 Presenter 将业务逻辑从 ArticlesActivity 抽离出来，使得 ArticlesActivity 变得非常轻量级，ArticlesActivity 此时的作用只是做一些 View 的初始化工作，职责单一、功能简单、易于维护。Presenter 与 View 的低耦合使得系统能够应对 UI 的易变性问题，也使得系统的 View 模块变得更易于维护。对于应用来说另一个问题就是数据模型与 View 的关系。MVP 中的 View 和 Model 不能直接通信，它们的交互都是通过 Presenter。从上述的代码中我们看到，ArticlePresenter 中不光只有 ArticleViewInterface，还持有了一个 ArticleModel 对象，这个 ArticleModel 自然就是 Model 角色，它负责处理数据，例如将数据存储到数据库中、从数据库加载缓存数据等。ArticleModel 同样也是一个接口，因此，Presenter 与 Model 也依赖于抽象而不是具体，使得 ArticleModel 的具体实现可以被轻易地替换。需要注意的是，在我们的示例中对于 ArticlePresenter 并没有进行接口抽象，而是使用了具体类，因为业务逻辑相对稳定，在此我们直接使用具体类即可。当然，如果你觉得你的业务逻辑相对来说易于变化，使用 Presenter 接口来应对变更是最好不过了。

由此可见 Model——View——Presenter 三者之间的关系都是松耦合，Presenter 持有的 View、Model 引用都是抽象，这样当 UI 发生变化时，我们只需要替换 View 即可；而数据库引擎需要替换时，只需要重新构建一个实现 ArticleModel 接口的类实现相关的存取逻辑即可。这样使得 View、Model、Presenter 三者之间可以独立的变化，测试也非常方便，可扩展性、灵活性都很高，如图 26-5 所示。



▲图 26-5

完整示例代码下载网址：https://www.github.com/bboyfeiyu/tech-frontier-mvp。

26.5 MVP 与 Activity、Fragment 的生命周期

综上所述，MVP 有很多优点，例如易于维护、易于测试、松耦合、复用性高、

健壮稳定、易于扩展等，但是，由于 Presenter 经常性地需要执行一些耗时操作，例如，我们上述示例的请求网络数据。而 Presenter 持有了 ArticlesActivity 的强引用，如果在请求结束之前 Activity 被销毁了，那么由于网络请求还没有返回，导致 Presenter 一直持有 ArticlesActivity 对象，使得 ArticlesActivity 对象无法被回收，此时就发生了内存泄露。

那么我们如何来解决这样的问题呢？

我们的答案是，通过弱引用和 Activity、Fragment 的声明周期来解决这个问题。首先建立一个 Presenter 抽象，我们命名为 BasePresenter，它是一个泛型类，泛型类型为 View 角色要实现的接口类型。具体代码如下。

```
public abstract class BasePresenter<T> {
    protected Reference<T> mViewRef;                // View 接口类型的弱引用

    public void attachView(T view) {
        mViewRef = new WeakReference<T>(view);   // 建立关联
    }

    protected T getView() {
        return mViewRef.get();
    }

    public boolean isViewAttached() {
        return mViewRef != null&& mViewRef.get() != null;
    }

    public void detachView() {
        if (mViewRef != null) {
            mViewRef.clear();
            mViewRef = null;
        }
    }
}
```

BasePresenter 有 4 个方法，分别与 View 建立关联、解除关联、判断是否与 View 建立了关联、获取 View。View 类型通过 BasePresenter 的泛型类型传递进来，Presenter 对这个 View 持有弱引用。通常情况下这个 View 类型应该是实现了某个特定接口的 Activity 或者 Fragment 等类型。

创建一个 MVPBaseActivity 基类，通过这个基类的声明周期函数来控制它与 Presenter 的关系，相关代码如下。

```
public abstract class MVPBaseActivity<V, Textends BasePresenter<V>>extends
        Activity {
    protected T mPresenter;   // Presenter 对象

    @SuppressWarnings("unchecked")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mPresenter = createPresenter();          // 创建 Presenter
```

```
        mPresenter.attachView((V) this);          // View 与 Presenter 建立关联
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mPresenter.detachView();
    }

    protected abstract T createPresenter();
}
```

MVPBaseActivity 含有两个泛型参数，第一个是 View 接口类型，第二个是 Presenter 的具体类型。通过泛型参数，使得一些通用的逻辑可以被抽象到 MVPBaseActivity 类中。例如，在 MVPBaseActivity 的 onCreate 函数中，会通过 createPresenter 函数创建一个具体的 Presenter，这个 Presenter 的类型就是 BasePresenter<T>类型。构建 Presenter 之后调用 attachView 函数与 Activity 建立关联。而在 onDestory 函数中，则会与 Activity 解除关联，从而避免内存泄漏。有的读者可能会问，如果在 onDestory 中解除了对 Activity 的引用，那么就没有必要再用弱引用了。这个问题的答案是，并不是在任何情况下 Activity 的 onDestory 都会被调用，一旦这种情况发生，弱引用也能够保证不会造成内存泄漏。而通过 MVPBaseActivity 的封装维护 Presenter 与 View 关联关系的代码，使得子类可以避免重复的代码，例如 ArticlesActivity 继承自 MVPBaseActivity 的实现如下。

```
public class ArticlesActivity extends
        MVPBaseActivity<ArticleViewInterface, ArticlePresenterV2>
            implements ArticleViewInterface {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initViews();
        RequestQueueMgr.init(getApplicationContext());
        mPresenter.fetchArticles();
    }
    // 代码省略
}
```

此时，Presenter 的创建以及与 View 建立关联等操作都被封装到 MVPBaseActivity 中，消除了子类重复代码的同时又避免了 Activity 的内存泄漏问题。你可以为 Fragment、FragmentActivity 等类型都建立一个类似这样的基类，这样一来你就可以将这种思想运用到更广阔的范围。

从整体效果来说，MVP 是开发过程中非常值得推荐的架构模式，它能够将各组件进行解耦，并且带来良好的可扩展性、可测试性、稳定性、可维护性，同时使得每个类型的职责相对单一、简单，避免了大量的“胖”的程序存在，例如数千行的 Activity 类。它有效的将业务逻辑、数据处理等工作从 Activity 等 View 元素中抽离出来，使得每个类尽可能简单，同时每个模块能够独立进行演化。它的思想也非常好地体现了面向对象的设计原则，也就是我们在第一章总结的几点，即抽象、单一职责、最小化、低耦合。