**一14一 一14一 一12一 一11一** 一10.5一 一10.5一 一10一

**目 录**

[1. UML类图 5](#_Toc22139)

[1.1. 概念 5](#_Toc17733)

[1.2. 类 5](#_Toc28353)

[1.3. 类的UML图示 5](#_Toc21551)

[1.4. 类与类之间的关系（4类9种） 6](#_Toc29680)

[1.4.1. 关联关系 6](#_Toc894)

[1.4.1.1. 双向关联 6](#_Toc11159)

[1.4.1.2. 单向关联 6](#_Toc1394)

[1.4.1.3. 自关联 7](#_Toc15900)

[1.4.1.4. 多重性关联 7](#_Toc19296)

[1.4.1.5. 聚合关系 7](#_Toc8189)

[1.4.1.6. 组合关系 8](#_Toc1630)

[1.4.2. 依赖关系 8](#_Toc23104)

[1.4.3. 泛化关系（继承关系） 8](#_Toc28283)

[1.4.4. 接口与实现关系 9](#_Toc21331)

[1.5. 实例分析1——登录模块 9](#_Toc3233)

[1.6. 实例分析2——注册模块 10](#_Toc995)

[1.7. 实例分析3——售票机控制程序 11](#_Toc25741)

[2. 设计模式java/android 13](#_Toc8051)

[2.1. 介绍 13](#_Toc1922)

[2.2. 六大原则 13](#_Toc14020)

[2.3. 设计模式分类 13](#_Toc7588)

[2.3.1. 创建型模式 13](#_Toc19009)

[2.3.2. 结构型模式 14](#_Toc8991)

[2.3.3. 行为型模式 14](#_Toc13586)

[2.3.4. 其他特殊模式 14](#_Toc16259)

[2.4. 创建型模式规则 Rules of thumb 14](#_Toc23940)

[3. 设计模式六大原则 15](#_Toc4345)

[3.1. 单一职责原则，Single Responsibility Principle，SRP 15](#_Toc15839)

[3.1.1. 定义： 15](#_Toc21063)

[3.1.2. 问题由来： 15](#_Toc12532)

[3.1.3. 解决方案： 15](#_Toc14771)

[3.1.4. 举例说明，用一个类描述动物呼吸这个场景： 15](#_Toc2003)

[3.1.5. 优点 17](#_Toc8266)

[3.2. 里氏替换原则，Liskov Substitution Principle，LSP 17](#_Toc23187)

[3.2.1. 定义 17](#_Toc15383)

[3.2.2. 问题由来： 17](#_Toc21533)

[3.2.3. 解决方案： 17](#_Toc19784)

[3.2.4. 举例说明继承的风险，我们需要完成一个两数相减的功能，由类A来负责。 18](#_Toc31041)

[3.2.5. 其他 19](#_Toc23898)

[3.3. 依赖倒置原则，Dependence Inversion Principle，DIP 19](#_Toc11695)

[3.3.1. 定义： 19](#_Toc26816)

[3.3.2. 问题由来： 19](#_Toc3467)

[3.3.3. 解决方案： 19](#_Toc20129)

[3.3.4. 举例说明 19](#_Toc10411)

[3.3.5. 其他 21](#_Toc32669)

[3.4. 接口隔离原则,InterfaceSegregation Principles，ISP 21](#_Toc21701)

[3.4.1. 定义： 21](#_Toc27587)

[3.4.2. 问题由来： 21](#_Toc13848)

[3.4.3. 解决方案： 21](#_Toc28950)

[3.4.4. 举例来说明接口隔离原则： 21](#_Toc23128)

[3.4.5. 其他 24](#_Toc22353)

[3.5. 迪米特原则,Law of Demeter，LOD 25](#_Toc24538)

[3.5.1. 定义： 25](#_Toc2771)

[3.5.2. 问题由来： 25](#_Toc13148)

[3.5.3. 解决方案： 25](#_Toc6100)

[3.5.4. 迪米特法则又叫最少知道原则 25](#_Toc225)

[3.5.5. 举一个例子： 25](#_Toc12369)

[3.5.5.1. 违反迪米特法则的设计 25](#_Toc6250)

[3.5.5.2. 修改后的代码如下 26](#_Toc6741)

[3.6. 开闭原则，Open Close Principle，OCP 27](#_Toc19872)

[3.6.1. 定义： 27](#_Toc4487)

[3.6.2. 问题由来： 27](#_Toc20907)

[3.6.3. 解决方案： 27](#_Toc15907)

[3.6.4. 再回想一下其他的5项原则 28](#_Toc16577)

[4. ----外卡型----对象池模式（Object Pool） 28](#_Toc13514)

[4.1. 对象池模式（Object Pool Pattern） 28](#_Toc10549)

[4.2. 特点 29](#_Toc19920)

[4.3. UML类图 29](#_Toc16407)

[4.4. MessagePool源码分析 30](#_Toc22510)

[4.5. 示例 31](#_Toc22424)

[4.6. 总结 33](#_Toc2429)

[4.7. 优点 34](#_Toc3540)

[4.8. 陷阱 34](#_Toc27822)

[4.9. 讨论与批判 34](#_Toc7859)

[5. ----创建型----原型模式（Prototype） 34](#_Toc9815)

[5.1. 原型模式（Prototype Pattern） 34](#_Toc23036)

[5.2. 特点 34](#_Toc13389)

[5.3. UML类图 35](#_Toc1741)

[5.4. 原型模式的通用代码： 35](#_Toc26466)

[5.5. 深拷贝和浅拷贝 37](#_Toc2073)

[5.6. 示例与源码 37](#_Toc15998)

[5.7. 总结 38](#_Toc29153)

[5.8. 优点 38](#_Toc13174)

[5.9. 缺点 38](#_Toc32049)

[6. 单例模式（Singleton） 38](#_Toc20354)

[6.1. 特点：  38](#_Toc31032)

[6.2. 主要优点 39](#_Toc19591)

[6.3. 主要缺点 39](#_Toc2549)

[6.4. UML类图 39](#_Toc29072)

[6.5. 示例与源码 39](#_Toc18749)

[6.5.1. 懒汉法，线程不安全,lazy initialization, thread-unsafety 39](#_Toc3113)

[6.5.2. 懒汉法,线程安全,lazy initialization, thread-safety, double-checked 39](#_Toc17723)

[6.5.3. 饿汉法，线程安全,eager initialization thread-safety 40](#_Toc13689)

[6.5.4. 静态内部类，线程安全,static inner class thread-safety 40](#_Toc28558)

[6.5.5. 枚举法,线程安全,enum thread-safety 41](#_Toc18770)

[6.5.6. 登记式 41](#_Toc27310)

[6.6. 完美双重检验法 42](#_Toc26252)

[6.7. 总结 42](#_Toc20303)

[7. 工厂方法（Factory Method） 42](#_Toc19298)

[7.1. 特点 43](#_Toc4548)

[7.2. UML类图 44](#_Toc13751)

[7.3. 示例与源码 44](#_Toc12227)

[7.3.1. 产品 44](#_Toc28258)

[7.3.2. 工厂 45](#_Toc20721)

[7.3.2.1. 工厂方法 45](#_Toc8176)

[7.3.2.1.1. multi ConcreateFactory 45](#_Toc6429)

[7.3.2.1.2. single ConcreteFactory 46](#_Toc23689)

[7.3.2.1.3. 总结对比 47](#_Toc8475)

[7.3.2.2. 简单工厂 47](#_Toc5075)

[7.4. 总结 47](#_Toc29587)

[8. 抽象工厂（Abstract Factory） 48](#_Toc261)

[8.1. 特点 48](#_Toc5453)

[8.2. UML类图 48](#_Toc6797)

[8.3. 示例与源码 49](#_Toc7349)

[8.3.1. 产品相关类 49](#_Toc6136)

[8.3.1.1. IButton 接口和其子类： 49](#_Toc15023)

[8.3.1.2. IText 接口和其实现子类：  49](#_Toc2040)

[8.3.2. 工厂相关类 50](#_Toc26365)

[8.3.3. 测试 50](#_Toc32704)

[8.4. MediaPlayer使用抽象工厂 51](#_Toc13494)

[8.5. 优点 51](#_Toc30215)

[8.6. 缺点 51](#_Toc14247)

[9. 抽象工厂VS工厂方法 51](#_Toc12155)

[10. 建造者模式（Builder） 51](#_Toc27032)

[10.1. 特点 52](#_Toc3941)

[10.2. UML类图 52](#_Toc14018)

[10.3. Builder 模式的通用代码： 52](#_Toc30410)

[10.4. 示例与源码 54](#_Toc21251)

[10.5. 总结 56](#_Toc27993)

[10.6. 优点： 56](#_Toc17800)

[10.7. 缺点： 56](#_Toc27350)

[10.8. Builder 模式 VS 工厂方法模式 56](#_Toc18190)

[10.8.1. 几个不同点： 56](#_Toc3309)

[10.8.2. 其次是 uml 类图的差异： 57](#_Toc12011)

[11. ----结构型----享元模式（Flyweight） 57](#_Toc9361)

[11.1. 享元模式（Flyweight Pattern） 57](#_Toc19824)

[11.2. 特点 57](#_Toc8015)

[11.3. 享元模式与对象池模式对比 57](#_Toc18032)

[11.4. UML类图 58](#_Toc13375)

[11.5. 享元模式的基础代码 58](#_Toc21743)

[11.5.1. Flyweight.class 58](#_Toc14711)

[11.5.2. ConcreteFlyweight.class 58](#_Toc26944)

[11.5.3. FlyweightFactory.class 58](#_Toc28212)

[11.5.4. 测试代码 59](#_Toc23836)

[11.6. Java 中的享元模式 59](#_Toc9994)

[11.7. 示例与源码 59](#_Toc5501)

[11.7.1. Shape.class用来定义一个图形的基本行为： 59](#_Toc29919)

[11.7.2. Circle.class Shape 的实现子类，用来画圆形： 59](#_Toc26686)

[11.7.3. ShapeFactory.class 图形享元工厂类： 59](#_Toc25727)

[11.7.4. 测试代码 60](#_Toc3026)

[11.8. 总结 60](#_Toc9223)

[11.8.1. 享元模式优点： 60](#_Toc18828)

[11.8.2. 享元模式缺点： 60](#_Toc1152)

[11.8.3. 讨论 61](#_Toc12110)

[12. 适配器模式（Adapter） 61](#_Toc16725)

[12.1. 介绍 61](#_Toc9119)

[12.2. 特点 61](#_Toc1171)

[12.3. 适配器模式的使用场景可以有以下几种： 61](#_Toc6002)

[12.4. UML类图 61](#_Toc5651)

[12.4.1. 类适配器模式 62](#_Toc11046)

[12.4.2. 对象配器模式 62](#_Toc21753)

[12.4.3. 对比 62](#_Toc2184)

[12.5. 示例与源码 62](#_Toc22067)

[12.5.1. 类适配器模式 62](#_Toc476)

[12.5.2. 对象适配器模式 63](#_Toc6102)

[12.6. 总结 64](#_Toc17975)

[12.7. 优点 64](#_Toc30867)

[12.8. ListAdapter适配器模式的代码实现 65](#_Toc15799)

[13. 桥接模式（Bridge） 66](#_Toc23443)

[13.1. 桥接模式（Bridge Pattern） 66](#_Toc29781)

[13.2. 特点 66](#_Toc28770)

[13.3. UML类图 67](#_Toc21633)

[13.4. 桥接模式的通用代码 67](#_Toc31228)

[13.5. Android 源码桥接模式分析 68](#_Toc6264)

[13.6. 示例与源码 70](#_Toc17134)

[13.7. 总结 71](#_Toc22975)

[14. 装饰者模式（Decorator、Wrapper） 71](#_Toc6880)

[14.1. 概述 72](#_Toc11827)

[14.2. 特点 72](#_Toc31173)

[14.3. UML类图 72](#_Toc10589)

[14.4. 扩展原有系统的功能的步骤 72](#_Toc15071)

[14.5. Java 中的装饰者模式 73](#_Toc24549)

[14.6. Android 中的装饰者模式 73](#_Toc14273)

[14.7. 示例与源码 73](#_Toc32730)

[14.8. 总结 76](#_Toc29374)

[15. 组合模式（Composite、Part-Whole） 76](#_Toc56)

[15.1. 概述 76](#_Toc3658)

[15.2. 特点 76](#_Toc17860)

[15.3. 安全的组合模式 77](#_Toc10931)

[15.3.1. 安全组合模式的 uml 类图 77](#_Toc2177)

[15.3.2. 安全组合模式的通用代码 77](#_Toc32052)

[15.4. 透明的组合模式 78](#_Toc6392)

[15.4.1. 透明组合模式 uml 类图 78](#_Toc11137)

[15.4.2. 透明组合模式的通用代码 78](#_Toc2069)

[15.5. 安全组合和透明组合的对比 80](#_Toc9404)

[15.6. 示例与源码 80](#_Toc10960)

[15.7. 总结 81](#_Toc27375)

[15.7.1. 组合模式的优点： 81](#_Toc16432)

[15.7.2. 组合模式的缺点： 81](#_Toc6532)

[16. 外观模式（Facade） 81](#_Toc29811)

[16.1. 概述 81](#_Toc6227)

[16.2. 特点 81](#_Toc4699)

[16.3. UML类图 81](#_Toc10296)

[16.4. 外观模式的通用代码： 82](#_Toc31726)

[16.5. 示例与源码 83](#_Toc7511)

[16.6. 总结 84](#_Toc3627)

[16.6.1. 优点： 84](#_Toc31858)

[16.6.2. 缺点： 84](#_Toc28551)

[17. 代理模式（Proxy） 84](#_Toc7165)

[17.1. 概述 84](#_Toc1835)

[17.2. 特点 85](#_Toc11995)

[17.3. UML类图 85](#_Toc2893)

[17.4. 代理模式的通用代码： 85](#_Toc3553)

[17.5. 动态+保护代理模式示例与源码 86](#_Toc18199)

[17.6. 总结 88](#_Toc32404)

[17.6.1. 优点： 88](#_Toc31988)

[17.6.2. 缺点： 89](#_Toc26815)

[18. 适配器 VS 装饰者 VS 桥接 VS 代理 VS 外观 89](#_Toc12363)

[19. ----行为型----观察者模式（Observer） 90](#_Toc14804)

[19.1. 特点 90](#_Toc14816)

[19.2. UML类图 91](#_Toc27700)

[19.3. 有几点需要注意的是 91](#_Toc25722)

[19.4. Java Api源码示例 91](#_Toc6610)

[19.4.1. Observer是一个接口，定义了一个update方法。 91](#_Toc31616)

[19.4.2. Observable是一个类，可被继承 91](#_Toc31945)

[19.4.3. 基本应用 93](#_Toc17315)

[19.4.3.1. 被观察者 93](#_Toc5605)

[19.4.3.2. 观察者1、2、3... 93](#_Toc21981)

[19.4.3.3. 调用 93](#_Toc12333)

[19.5. 异步观察者模式 94](#_Toc22337)

[19.5.1. 异步观察模式的观察者接口、观察者的包装类 94](#_Toc29161)

[19.5.2. 异步观察者模式的被观察者对象 94](#_Toc7283)

[19.5.3. 异步观察者模式调用示例 96](#_Toc28354)

[19.5.3.1. 被观察者 96](#_Toc32195)

[19.5.3.2. 观察者1、2、3... 96](#_Toc3046)

[19.5.3.3. 调用 96](#_Toc2367)

[20. 责任链模式（Chain-of-responsibility） 97](#_Toc20059)

[20.1. 概述 97](#_Toc27195)

[20.2. 特点 97](#_Toc11500)

[20.3. UML类图 97](#_Toc23065)

[20.4. 责任联模式的通用代码 97](#_Toc2629)

[20.5. 示例与源码 99](#_Toc19546)

[20.6. 总结 101](#_Toc24897)

[21. 命令模式（Command） 101](#_Toc22596)

[21.1. 概述 101](#_Toc15492)

[21.2. 特点 101](#_Toc13468)

[21.3. UML类图 103](#_Toc7103)

[21.4. 命令模式的通用代码： 103](#_Toc32324)

[21.5. 示例与源码 104](#_Toc15454)

[21.6. 总结 106](#_Toc2240)

[22. 策略模式（Strategy） 106](#_Toc10359)

[22.1. 概述 107](#_Toc12271)

[22.2. 特点 107](#_Toc17950)

[22.3. UML图 107](#_Toc7291)

[22.4. 策略模式的概念 107](#_Toc12954)

[22.5. 策略模式的应用场景(Volley) 107](#_Toc14535)

[22.5.1. RetryPolicy.java 请求重试策略类，定义了三个接口 108](#_Toc20498)

[22.5.2. DefaultRetryPolicy.java 默认重试策略，继承了RetryPolicy ，具体实现了接口 108](#_Toc17005)

[22.5.3. Request.java 请求类，持有一个抽象策略类RetryPolicy 的引用，最终给客户端调用 108](#_Toc24986)

[22.5.4. BasicNetwork.java 网络处理类，处理Request，使用到了对应的重试策略 108](#_Toc1159)

[22.6. 自定义简单的策略模式实例 109](#_Toc12502)

[23. 状态模式（State） 110](#_Toc26964)

[23.1. Think in Java Demo 110](#_Toc20876)

[23.2. 概述 111](#_Toc17916)

[23.3. 特点 111](#_Toc31444)

[23.4. UML类图 111](#_Toc6528)

[23.5. 状态模式的通用代码： 111](#_Toc27345)

[23.6. wiki 上的 demo ，实现一堆字符串的一个大小写间隔打印： 112](#_Toc22)

[23.7. 总结 114](#_Toc12556)

[23.7.1. 优点 114](#_Toc26247)

[23.7.2. 缺点 114](#_Toc23797)

[24. 状态模式VS策略模式 114](#_Toc9535)

[25. 中介者模式（Mediator） 114](#_Toc14609)

[25.1. 概述 114](#_Toc10553)

[25.2. 特点 115](#_Toc16599)

[25.3. UML类图 115](#_Toc16619)

[25.4. 中介者模式的通用代码 116](#_Toc11860)

[25.5. 示例与源码 117](#_Toc5590)

[25.6. 总结 120](#_Toc9738)

[25.6.1. 优点： 120](#_Toc24433)

[25.6.2. 缺点： 120](#_Toc31397)

[26. 迭代器模式（Iterator、Cursor） 120](#_Toc639)

[26.1. 概述 120](#_Toc3964)

[26.2. 特点 120](#_Toc10265)

[26.3. UML类图 121](#_Toc22519)

[26.4. 迭代器模式的通用代码： 121](#_Toc29163)

[26.5. 示例与源码 122](#_Toc9045)

[26.6. 总结 123](#_Toc23926)

[27. 模板方法（Template Method） 124](#_Toc8036)

[27.1. 概述 124](#_Toc13915)

[27.2. 特点 124](#_Toc4642)

[27.3. UML类图 124](#_Toc14326)

[27.4. 模板方法模式的通用代码： 125](#_Toc2417)

[27.5. 示例与源码 126](#_Toc16592)

[27.6. 总结 127](#_Toc31027)

[27.6.1. 优点： 127](#_Toc20151)

[27.6.2. 缺点： 127](#_Toc17298)

[28. 解释器模式（Interpreter） 127](#_Toc28795)

[28.1. 概述 127](#_Toc17473)

[28.2. 特点 127](#_Toc668)

[28.3. UML类图 128](#_Toc32497)

[28.4. 解释器模式的通用代码 128](#_Toc30325)

[28.5. 示例与源码 129](#_Toc21077)

[28.6. 总结 131](#_Toc8097)

[29. 访问者模式（Visitor） 131](#_Toc23659)

[29.1. 概述 131](#_Toc8624)

[29.2. 特点 131](#_Toc25060)

[29.3. UML类图 132](#_Toc10347)

[29.4. 分派的概念 132](#_Toc23003)

[29.4.1. 静态分派 132](#_Toc10030)

[29.4.2. 动态分派 133](#_Toc2111)

[29.4.3. 编程呢言的分派类型 134](#_Toc36)

[29.4.4. 双重分派 134](#_Toc9372)

[29.5. 访问者模式的通用代码 135](#_Toc14960)

[29.6. 总结 137](#_Toc14379)

[29.6.1. 优点： 137](#_Toc26245)

[29.6.2. 缺点： 137](#_Toc30025)

# UML类图

## 概念

**UML**

UML(Unified Modeling Language,统一建模语言或[标准建模语言](http://baike.baidu.com/view/3458435.htm" \t "http://baike.baidu.com/_blank))，是始于1997年一个[OMG](http://baike.baidu.com/subview/138917/5088704.htm" \t "http://baike.baidu.com/_blank)标准，它是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持，包括由需求分析到规格，到构造和配置。 面向对象的分析与设计(OOA&D，OOAD)方法的发展在80年代末至90年代中出现了一个高潮，[UML](http://baike.baidu.com/view/23396.htm" \t "http://baike.baidu.com/_blank)是这个高潮的产物。它不仅统一了Booch、Rumbaugh和Jacobson的表示方法，而且对其作了进一步的发展，并最终统一为大众所接受的[标准建模语言](http://baike.baidu.com/view/3458435.htm" \t "http://baike.baidu.com/_blank)。

**UML类图**

在UML的静态机制中类图是一个重点，它不但是设计人员关心的核心，更是实现人员关注的核心。建模工具也主要根据类图来产生代码。类图在UML的9个图中占据了一个相当重要的地位。James Rumbaugh对类的定义是：类是具有相似结构、行为和关系的一组对象的描述符。类是面向对象系统中最重要的构造块。类图显示了一组类、接口、协作以及他们之间的关系。在UML中问题域最终要被逐步转化，通过类来建模，通过编程语言构建这些类从而实现系统。类加上他们之间的关系就构成了类图，类图中还可以包含接口、包等元素，也可以包括对象、链等实例

## 类

类(Class)封装了数据和行为，是面向对象的重要组成部分，它是具有相同属性、操作、关系的对象集合的总称。在系统中，每个类都具有一定的职责，职责指的是类要完成什么样的功能，要承担什么样的义务。一个类可以有多种职责，设计得好的类一般只有一种职责。在定义类的时候，将类的职责分解成为类的属性和操作（即方法）。类的属性即类的数据职责，类的操作即类的行为职责。设计类是面向对象设计中最重要的组成部分，也是最复杂和最耗时的部分。

在软件系统运行时，类将被实例化成对象(Object)，对象对应于某个具体的事物，是类的实例(Instance)。

类图(Class Diagram)使用出现在系统中的不同类来描述系统的静态结构，它用来描述不同的类以及它们之间的关系。

在系统分析与设计阶段，类通常可以分为三种，实体类(Entity Class)、控制类(Control Class)和边界类(Boundary Class)：

**实体类：**

实体类对应系统需求中的每个实体，它们通常需要保存在永久存储体中，一般使用数据库表或文件来记录，实体类既包括存储和传递数据的类，还包括操作数据的类。实体类来源于需求说明中的名词，如学生、商品等。

**控制类：**

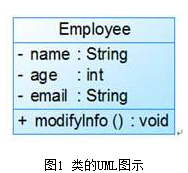
控制类用于体现应用程序的执行逻辑，提供相应的业务操作，将控制类抽象出来可以降低界面和数据库之间的耦合度。控制类一般是由动宾结构的短语（动词+名词）转化来的名词，如增加商品对应有一个商品增加类，注册对应有一个用户注册类等

**边界类：**

边界类用于对外部用户与系统之间的交互对象进行抽象，主要包括界面类，如对话框、窗口、菜单等。

在面向对象分析和设计的初级阶段，通常首先识别出实体类，绘制初始类图，此时的类图也可称为领域模型，包括实体类及其它们之间的相互关系。

## 类的UML图示

在UML类图中，类一般由三部分组成：

(1) 第一部分是类名：每个类都必须有一个名字，类名是一个字符串。

(2) 第二部分是类的属性(Attributes)：属性是指类的性质，即类的成员变量。一个类可以有任意多个属性，也可以没有属性

UML规定属性的表示方式为： 可见性 名称:类型 [ = 缺省值 ]

“可见性”表示该属性对于类外的元素而言是否可见，包括公有(public) +；私有(private) - ；受保护(protected)#三种

“名称”表示属性名，用一个字符串表示。

“类型”表示属性的数据类型，可以是基本数据类型，也可以是用户自定义类型。

“缺省值”是一个可选项，即属性的初始值。

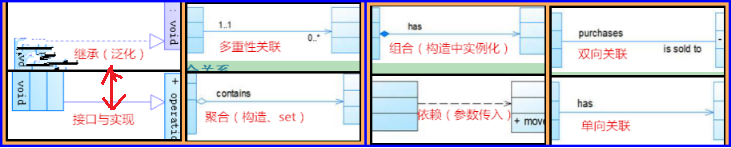
(3) 第三部分是类的操作(Operations)：操作是类的任意一个实例对象都可以使用的行为，是类的成员方法。

UML规定操作的表示方式为： 可见性 名称(参数列表) [ : 返回类型]

“返回类型”是一个可选项，表示方法的返回值类型，依赖于具体的编程语言，可以是基本数据类型，也可以是用户自定义类型，还可以是空类型(void)，如果是构造方法，则无返回类型。

## 类与类之间的关系（4类9种）

在软件系统中，类并不是孤立存在的，类与类之间存在各种关系，对于不同类型的关系，UML提供了不同的表示方式。



### 关联关系

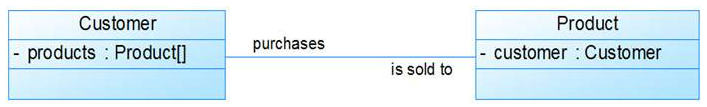
关联(Association)关系是类与类之间最常用的一种关系，它是一种结构化关系，用于表示一类对象与另一类对象之间有联系，如汽车和轮胎、师傅和徒弟、班级和学生等等。在UML类图中，用实线连接有关联关系的对象所对应的类，在使用Java、C#和C++等编程语言实现关联关系时，通常将一个类的对象作为另一个类的成员变量。在使用类图表示关联关系时可以在关联线上标注角色名，一般使用一个表示两者之间关系的动词或者名词表示角色名（有时该名词为实例对象名），关系的两端代表两种不同的角色，因此在一个关联关系中可以包含两个角色名，角色名不是必须的，可以根据需要增加，其目的是使类之间的关系更加明确。



**在UML中，关联关系通常又包含如下几种形式：**

#### 双向关联

默认情况下，关联是双向的。例如：顾客(Customer)购买商品(Product)并拥有商品，反之，卖出的商品总有某个顾客与之相关联。因此，Customer类和Product类之间具有双向关联关系



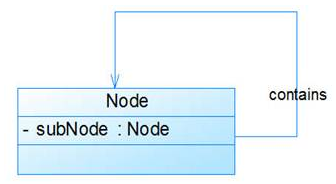
#### 单向关联

类的关联关系也可以是单向的，单向关联用带箭头的实线表示。例如：顾客(Customer)拥有地址(Address)，则Customer类与Address类具有单向关联关系



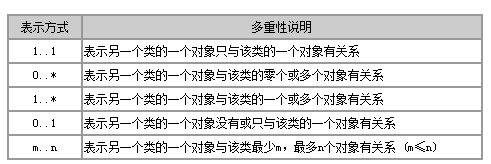
#### 自关联

在系统中可能会存在一些类的属性对象类型为该类本身，这种特殊的关联关系称为自关联。例如：一个节点类(Node)的成员又是节点Node类型的对象



#### 多重性关联

多重性关联关系又称为重数性(Multiplicity)关联关系，表示两个关联对象在数量上的对应关系。在UML中，对象之间的多重性可以直接在关联直线上用一个数字或一个数字范围表示。对象之间可以存在多种多重性关联关系，常见的多重性表示方式如表所示：

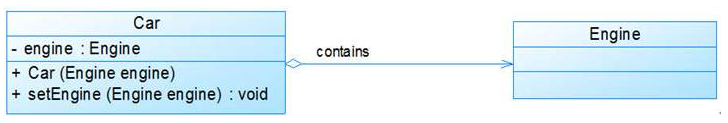


例如：一个界面(Form)可以拥有零个或多个按钮(Button)，但是一个按钮只能属于一个界面，因此，一个Form类的对象可以与零个或多个Button类的对象相关联，但一个Button类的对象只能与一个Form类的对象关联



#### 聚合关系

聚合(Aggregation)关系表示整体与部分的关系。在聚合关系中，成员对象是整体对象的一部分，但是成员对象可以脱离整体对象独立存在。在UML中，聚合关系用带空心菱形的直线表示。例如：汽车发动机(Engine)是汽车(Car)的组成部分，但是汽车发动机可以独立存在，因此，汽车和发动机是聚合关系



在代码实现聚合关系时，成员对象通常作为构造方法、Setter方法或业务方法的参数注入到整体对象中

public class Car {

private Engine engine;

//构造注入

public Car(Engine engine) {

this.engine = engine;

}

//设值注入

public void setEngine(Engine engine) {

this.engine = engine;

}

……

}

public class Engine {

……

}

#### 组合关系

组合(Composition)关系也表示类之间整体和部分的关系，但是在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也将不存在，成员对象与整体对象之间具有同生共死的关系。在UML中，组合关系用带实心菱形的直线表示。例如：人的头(Head)与嘴巴(Mouth)，嘴巴是头的组成部分之一，而且如果头没了，嘴巴也就没了，因此头和嘴巴是组合关系：



在代码实现组合关系时，通常在整体类的构造方法中直接实例化成员类

public class Head {

private Mouth mouth;

public Head() {

mouth = new Mouth(); //实例化成员类

}

……

}

public class Mouth {

……

}

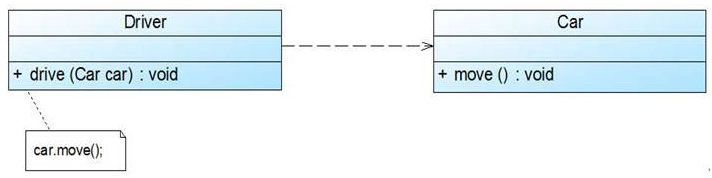
### 依赖关系

依赖(Dependency)关系是一种使用关系，特定事物的改变有可能会影响到使用该事物的其他事物，在需要表示一个事物使用另一个事物时使用依赖关系。大多数情况下，依赖关系体现在某个类的方法使用另一个类的对象作为参数。在UML中，依赖关系用带箭头的虚线表示，由依赖的一方指向被依赖的一方。例如：驾驶员开车，在Driver类的drive()方法中将Car类型的对象car作为一个参数传递，以便在drive()方法中能够调用car的move()方法，且驾驶员的drive()方法依赖车的move()方法，因此类Driver依赖类Car。在系统实施阶段，依赖关系通常通过三种方式来实现：

第一种也是最常用的一种方式是如图1所示的将一个类的对象作为另一个类中方法的参数；

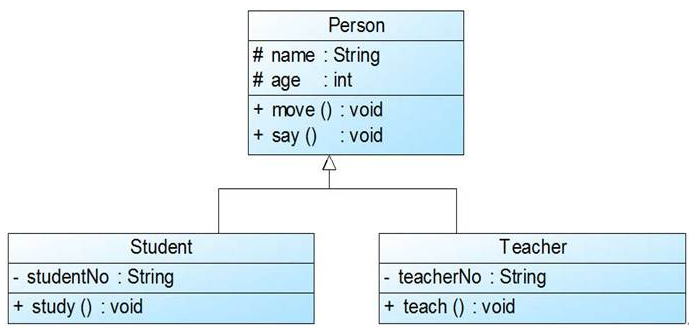
第二种方式是在一个类的方法中将另一个类的对象作为其局部变量；

第三种方式是在一个类的方法中调用另一个类的静态方法



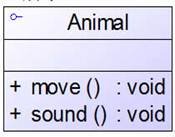
### 泛化关系（继承关系）

泛化(Generalization)关系也就是继承关系，用于描述父类与子类之间的关系，父类又称作基类或超类，子类又称作派生类。在UML中，泛化关系用带空心三角形的直线来表示。在代码实现时，我们使用面向对象的继承机制来实现泛化关系，如在Java语言中使用extends关键字、在C++/C#中使用冒号“：”来实现。例如：Student类和Teacher类都是Person类的子类，Student类和Teacher类继承了Person类的属性和方法，Person类的属性包含姓名(name)和年龄(age)，每一个Student和Teacher也都具有这两个属性，另外Student类增加了属性学号(studentNo)，Teacher类增加了属性教师编号(teacherNo)，Person类的方法包括行走move()和说话say()，Student类和Teacher类继承了这两个方法，而且Student类还新增方法study()，Teacher类还新增方法teach()

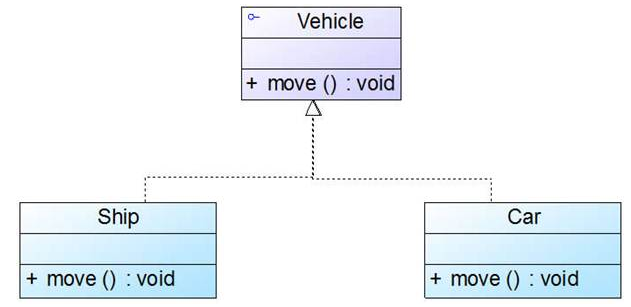


### 接口与实现关系

在很多面向对象语言中都引入了接口的概念，如Java、C#等，在接口中，通常没有属性，而且所有的操作都是抽象的，只有操作的声明，没有操作的实现。UML中用与类的表示法类似的方式表示接口



接口之间也可以有与类之间关系类似的继承关系和依赖关系，但是接口和类之间还存在一种实现(Realization)关系，在这种关系中，类实现了接口，类中的操作实现了接口中所声明的操作。在UML中，类与接口之间的实现关系用带空心三角形的虚线来表示。例如：定义了一个交通工具接口Vehicle，包含一个抽象操作move()，在类Ship和类Car中都实现了该move()操作，不过具体的实现细节将会不一样



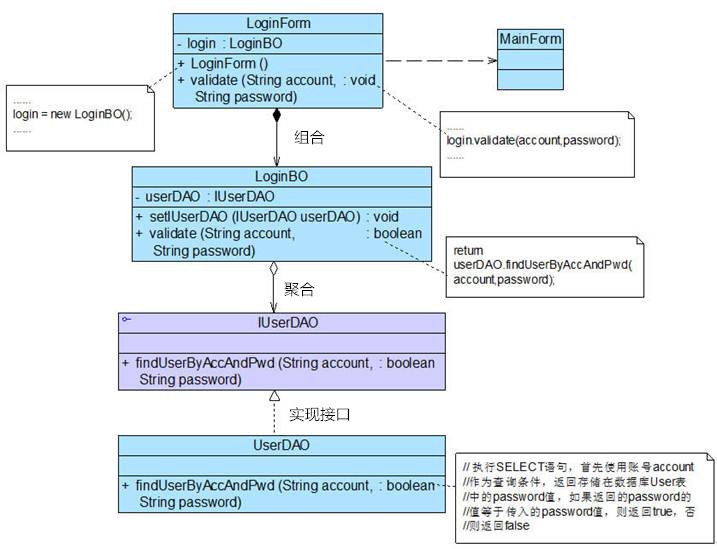
## 实例分析1——登录模块

某基于C/S的即时聊天系统登录模块功能描述如下：用户通过登录界面(LoginForm)输入账号和密码，系统将输入的账号和密码与存储在数据库(User)表中的用户信息进行比较，验证用户输入是否正确，如果输入正确则进入主界面(MainForm)，否则提示“输入错误”。

**根据以上描述绘制初始类图。**

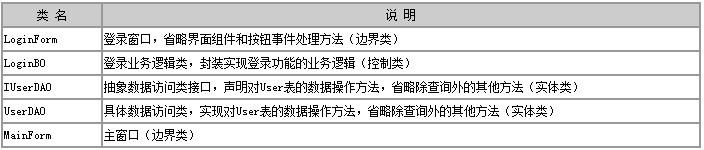
**参考解决方案：**

**参考类图如下：**

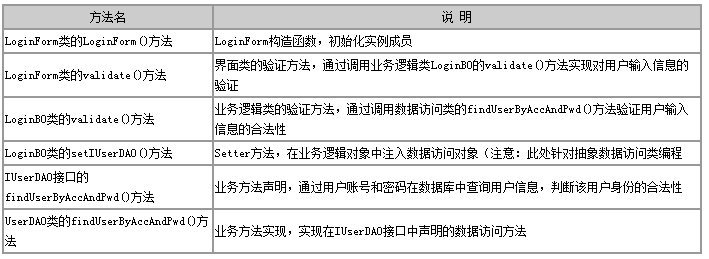


考虑到系统扩展性，在本实例中引入了抽象数据访问接口IUserDAO，再将具体数据访问对象注入到业务逻辑对象中，可通过配置文件（如XML文件）等方式来实现，将具体的数据访问类类名存储在配置文件中，如果需要更换新的具体数据访问对象，只需修改配置文件即可，原有程序代码无须做任何修改。

**类说明：**



**方法说明：**



## 实例分析2——注册模块

某基于Java语言的C/S软件需要提供注册功能，该功能简要描述如下：用户通过注册界面(RegisterForm)输入个人信息，用户点击“注册”按钮后将输入的信息通过一个封装用户输入数据的对象(UserDTO)传递给操作数据库的数据访问类，为了提高系统的扩展性，针对不同的数据库可能需要提供不同的数据访问类，因此提供了数据访问类接口，如IUserDAO，每一个具体数据访问类都是某一个数据访问类接口的实现类，如OracleUserDAO就是一个专门用于访问Oracle数据库的数据访问类。

**根据以上描述绘制类图**

为了简化类图，个人信息仅包括账号(userAccount)和密码(userPassword)，且界面类无需涉及界面细节元素。

**参考解决方案：**

在以上功能说明中，可以分析出该系统包括三个类和一个接口，这三个类分别是注册界面类RegisterForm、用户数据传输类UserDTO、Oracle用户数据访问类OracleUserDAO，接口是抽象用户数据访问接口IUserDAO。它们之间的关系如下：

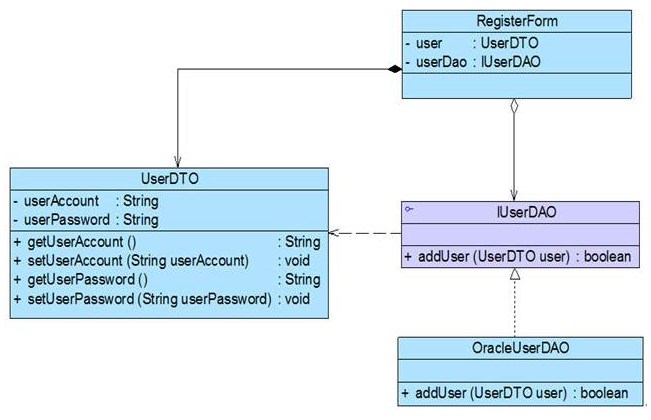
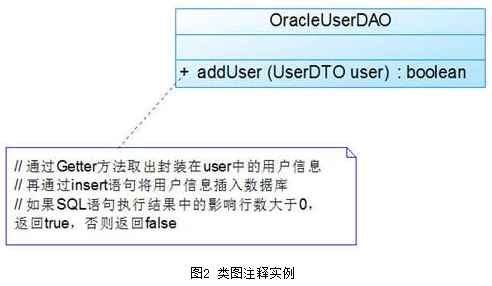
(1) 在RegisterForm中需要使用UserDTO类传输数据且需要使用数据访问类来操作数据库，因此RegisterForm与UserDTO和IUserDAO之间存在关联关系，在RegisterForm中可以直接实例化UserDTO，因此它们之间可以使用组合关联。

(2) 由于数据库类型需要灵活更换，因此在RegisterForm中不能直接实例化IUserDAO的子类，可以针对接口IUserDAO编程，再通过注入的方式传入一个IUserDAO接口的子类对象，因此RegisterForm和IUserDAO之间具有聚合关联关系。

(3) OracleUserDAO是实现了IUserDAO接口的子类，因此它们之间具有类与接口的实现关系。

(4) 在声明IUserDAO接口的增加用户信息方法addUser()时，需要将在界面类中实例化的UserDTO对象作为参数传递进来，然后取出封装在UserDTO对象中的数据插入数据库，因此addUser()方法的函数原型可以定义为：public boolean addUser(UserDTO user)，在IUserDAO的方法addUser()中将UserDTO类型的对象作为参数，故IUserDAO与UserDTO存在依赖关系。

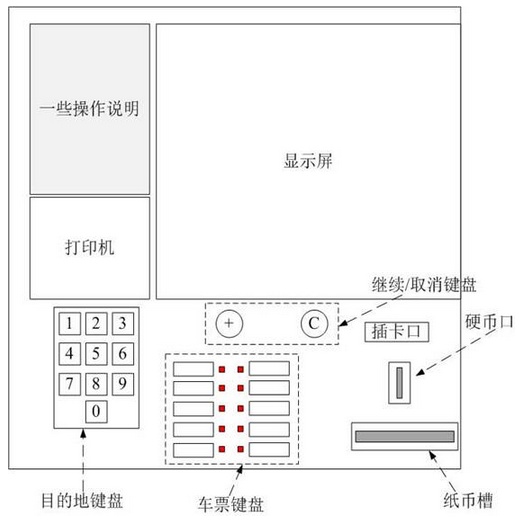
**通过以上分析，该实例参考类图如图所示：**

注意：在绘制类图或其他UML图形时，可以通过注释(Comment)来对图中的符号或元素进行一些附加说明，如果需要详细说明类图中的某一方法的功能或者实现过程，可以使用如图2所示表示方式：

## 实例分析3——售票机控制程序

某运输公司决定为新的售票机开发车票销售的控制软件。图给出了售票机的面板示意图以及相关的控制部件。



**售票机相关部件的作用如下所述：**

(1) 目的地键盘用来输入行程目的地的代码（例如，200表示总站）。

(2) 乘客可以通过车票键盘选择车票种类（单程票、多次往返票和座席种类）。

(3) 继续/取消键盘上的取消按钮用于取消购票过程，继续按钮允许乘客连续购买多张票。

(4) 显示屏显示所有的系统输出和用户提示信息。

(5) 插卡口接受MCard（现金卡），硬币口和纸币槽接受现金。

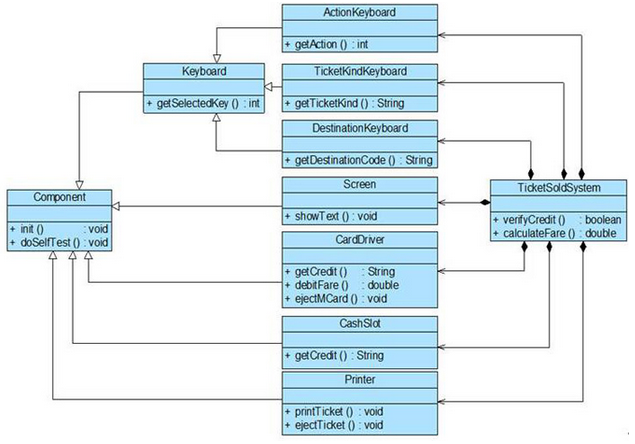
(6) 打印机用于输出车票。

(7) 所有部件均可实现自检并恢复到初始状态。

**现采用面向对象方法开发该系统，使用UML进行建模，绘制该系统的初始类图。**

**参考解决方案：**

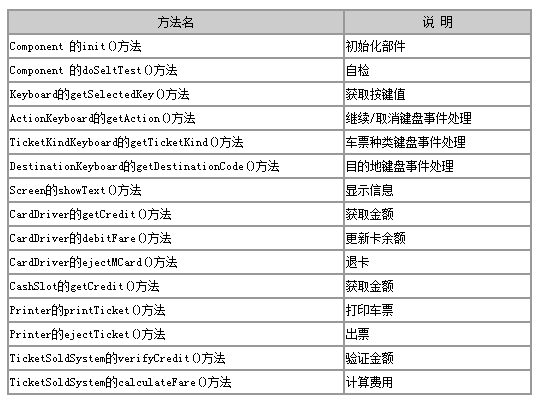
**参考类图如下：**



**类说明：**



**方法说明：**



# 设计模式java/android

## 介绍

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。一个设计模式并不是能够用于代码中的，而是提供一个在不同情况下解决问题的模板示例。设计模式允许开发者用一个大家都熟知的编码方式进行交流，有利于软件的扩展性和可维护性。

## 六大原则

**设计模式当然也需要遵循面向对象设计的6大原则：**

* 单一职责原则，Single Responsibility Principle，SRP
* 开闭原则，Open Close Principle，OCP
* 里氏替换原则，Liskov Substitution Principle，LSP
* 依赖倒置原则，Dependence Inversion Principle ，DIP
* 接口隔离原则,InterfaceSegregation Principles，ISP
* 迪米特原则,Law of Demeter，LOD

## 设计模式分类

**设计模式主要分三个类型:创建型、结构型和行为型。**

### 创建型模式

Abstract Factory（抽象工厂模式）：为创建一组相关或者是相互依赖的对象提供一个接口，而不需要制定它们的具体类。

　　Builder（建造者模式）：将一个复杂对象的构建和它的表示分离，使得同样的构建过程可以创建不同的表示。

　　Factory Method（工厂方法模式）：定义一个创建对象的接口，让子类决定实例化哪个类。

　　Prototype（原型模式）：用原型实例指向创建对象的种类，并通过拷贝这些原型创建新的对象。

　　Singleton（单例模式）：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。

### 结构型模式

　　Adapter（适配器模式）：适配器模式把一个类的接口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。

　　Bridge（桥接模式）：将抽象部分与实现部分分离，使他们都可以独立地进行变化。

　　Composite（组合模式）：组合模式允许你将对象组合成树形结构来表现“整体/部分”层次结构，并且能够让客户端以一致的方式处理个别对象以及组合对象。

　　Decorator（装饰者模式）：动态地将职责附加到对象上，对于扩展功能来说，装饰者提供了有别于继承的另一种选择。

　　Facade（外观模式）：外观模式提供一个统一的接口，用来访问子系统中的一群接口，外观定义了一个高层接口，让子系统更容易使用。

　　Flyweight（享元模式）：使用共享对象可有效地支持大量细粒度的对象。

　　Proxy（代理模式）：为另一个对象提供一个代理以控制对这个对象的访问。

### 行为型模式

　　Chain of responsibility（责任链模式）：使多个对象都有机会处理请求，从而避免请求的送发者和接收者之间的耦合关系

　　Command（命令模式）：将一个请求封装为一个对象，从而使你可以用不同的请求对客户进行参数化，对请求排队和记录请求日志，以及支持可撤销的操作

　　Interpreter（解释器模式）：给定一个语言，定义他的文法的一个表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。   
　　Iterator（迭代器模式）：提供一个方法顺序访问一个聚合对象的各个元素，而又不需要暴露该对象的内部表示。

　　Mediator（中介者模式）：用一个中介对象封装一些列的对象交互

　　Memento（备忘录模式）：在不破坏对象的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态

　　Observer（观察者模式）：定义对象间一对多的依赖关系，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。

　　State（状态模式）：允许对象在其内部状态改变时改变他的行为。对象看起来似乎改变了他的类

　　Strategy（策略模式）：定义一系列的算法，把他们一个个封装起来，并使他们可以互相替换，本模式使得算法可以独立于使用它们的客户

　　Template method（模板方法模式）：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，TemplateMethod使得子类可以不改变一个算法的结构即可以重定义该算法得某些特定步骤

Visitor（访问者模式）：表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变各元素类的前提下定义作用于这个元素的新操作

### 其他特殊模式

Object Pool（对象池模式）：维护一定数量的对象集合，使用时直接向对象池申请，以跳过对象的 expensive initialization 。

## 创建型模式规则 Rules of thumb

有些时候创建型模式是可以重叠使用的，有一些抽象工厂模式和原型模式都可以使用的场景，这个时候使用任一设计模式都是合理的；在其他情况下，他们各自作为彼此的补充：抽象工厂模式可能会使用一些原型类来克隆并且返回产品对象。

抽象工厂模式，建造者模式和原型模式都能使用单例模式来实现他们自己；抽象工厂模式经常也是通过工厂方法模式实现的，但是他们都能够使用原型模式来实现；

通常情况下，设计模式刚开始会使用工厂方法模式（结构清晰，更容易定制化，子类的数量爆炸），如果设计者发现需要更多的灵活性时，就会慢慢地发展为抽象工厂模式，原型模式或者建造者模式（结构更加复杂，使用灵活）；

原型模式并不一定需要继承，但是它确实需要一个初始化的操作，工厂方法模式一定需要继承，但是不一定需要初始化操作；

使用装饰者模式或者组合模式的情况通常也可以使用原型模式来获得益处；

单例模式中，只要将构造方法的访问权限设置为 private 型，就可以实现单例。但是原型模式的 clone 方法直接无视构造方法的权限来生成新的对象，所以，单例模式与原型模式是冲突的，在使用时要特别注意。

# 设计模式六大原则

## 单一职责原则，Single Responsibility Principle，SRP

### 定义：

就一个类而言，应该仅有一个引起它变化的原因。通俗的说，即一个类只负责一项职责。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，所以如果类执行的功能过多就要考虑将类进行拆分了。

### 问题由来：

类T负责两个不同的职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。

### 解决方案：

遵循单一职责原则。分别建立两个类T1、T2，使T1完成职责P1功能，T2完成职责P2功能。这样，当修改类T1时，不会使职责P2发生故障风险；同理，当修改T2时，也不会使职责P1发生故障风险。

说到单一职责原则，很多人都会不屑一顾。因为它太简单了。稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。而避免出现这一问题的方法便是遵循单一职责原则。虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。为什么会出现这种现象呢？因为有职责扩散。所谓职责扩散，就是因为某种原因，职责P被分化为粒度更细的职责P1和P2。

比如：类T只负责一个职责P，这样设计是符合单一职责原则的。后来由于某种原因，也许是需求变更了，也许是程序的设计者境界提高了，需要将职责P细分为粒度更细的职责P1，P2，这时如果要使程序遵循单一职责原则，需要将类T也分解为两个类T1和T2，分别负责P1、P2两个职责。但是在程序已经写好的情况下，这样做简直太费时间了。所以，简单的修改类T，用它来负责两个职责是一个比较不错的选择，虽然这样做有悖于单一职责原则。（这样做的风险在于职责扩散的不确定性，因为我们不会想到这个职责P，在未来可能会扩散为P1，P2，P3，P4……Pn。所以记住，在职责扩散到我们无法控制的程度之前，立刻对代码进行重构。）

### 举例说明，用一个类描述动物呼吸这个场景：

class Animal{

public void breathe(String animal){

System.out.println(animal+"呼吸空气");

}

}

public class Client{

public static void main(String[] args){

Animal animal = new Animal();

animal.breathe("牛");

animal.breathe("羊");

animal.breathe("猪");

}

}

运行结果：

牛呼吸空气

羊呼吸空气

猪呼吸空气

程序上线后，发现问题了，并不是所有的动物都呼吸空气的，比如鱼就是呼吸水的。修改时如果遵循单一职责原则，需要将Animal类细分为陆生动物类Terrestrial，水生动物Aquatic，代码如下：

class Terrestrial{

public void breathe(String animal){

System.out.println(animal+"呼吸空气");

}

}

class Aquatic{

public void breathe(String animal){

System.out.println(animal+"呼吸水");

}

}

public class Client{

public static void main(String[] args){

Terrestrial terrestrial = new Terrestrial();

terrestrial.breathe("牛");

terrestrial.breathe("羊");

terrestrial.breathe("猪");

Aquatic aquatic = new Aquatic();

aquatic.breathe("鱼");

}

}

运行结果：

牛呼吸空气

羊呼吸空气

猪呼吸空气

鱼呼吸水

我们会发现如果这样修改花销是很大的，除了将原来的类分解之外，还需要修改客户端。而直接修改类Animal来达成目的虽然违背了单一职责原则，但花销却小的多，代码如下：

class Animal{

public void breathe(String animal){

if("鱼".equals(animal)){

System.out.println(animal+"呼吸水");

}else{

System.out.println(animal+"呼吸空气");

}

}

}

public class Client{

public static void main(String[] args){

Animal animal = new Animal();

animal.breathe("牛");

animal.breathe("羊");

animal.breathe("猪");

animal.breathe("鱼");

}

}

可以看到，这种修改方式要简单的多。但是却存在着隐患：有一天需要将鱼分为呼吸淡水的鱼和呼吸海水的鱼，则又需要修改Animal类的breathe方法，而对原有代码的修改会对调用“猪”“牛”“羊”等相关功能带来风险，也许某一天你会发现程序运行的结果变为“牛呼吸水”了。这种修改方式直接在代码级别上违背了单一职责原则，虽然修改起来最简单，但隐患却是最大的。还有一种修改方式：

class Animal{

public void breathe(String animal){

System.out.println(animal+"呼吸空气");

}

public void breathe2(String animal){

System.out.println(animal+"呼吸水");

}

}

public class Client{

public static void main(String[] args){

Animal animal = new Animal();

animal.breathe("牛");

animal.breathe("羊");

animal.breathe("猪");

animal.breathe2("鱼");

}

}

可以看到，这种修改方式没有改动原来的方法，而是在类中新加了一个方法，这样虽然也违背了单一职责原则，但在方法级别上却是符合单一职责原则的，因为它并没有动原来方法的代码。这三种方式各有优缺点，那么在实际编程中，采用哪一中呢？其实这真的比较难说，需要根据实际情况来确定。我的原则是：只有逻辑足够简单，才可以在代码级别上违反单一职责原则；只有类中方法数量足够少，才可以在方法级别上违反单一职责原则；

例如本文所举的这个例子，它太简单了，它只有一个方法，所以，无论是在代码级别上违反单一职责原则，还是在方法级别上违反，都不会造成太大的影响。实际应用中的类都要复杂的多，一旦发生职责扩散而需要修改类时，除非这个类本身非常简单，否则还是遵循单一职责原则的好。

### 优点

* 可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；
* 提高类的可读性，提高系统的可维护性；
* 变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可显著降低对其他功能的影响。

需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都适用单一职责原则。

## 里氏替换原则，Liskov Substitution Principle，LSP

这项原则最早是在1988年，由麻省理工学院的一位姓里的女士（Barbara Liskov）提出来的。

### 定义

定义一：如果对每一个类型为 T1的对象 o1，都有类型为 T2 的对象o2，使得以 T1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 T2 是类型 T1 的子类型；定义2：所有引用基类的地方必须能透明地使用其子类的对象。

### 问题由来：

有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会导致原有功能P1发生故障。

### 解决方案：

当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。

继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能会产生故障。

### 举例说明继承的风险，我们需要完成一个两数相减的功能，由类A来负责。

class A{

public int func1(int a, int b){

return a-b;

}

}

public class Client{

public static void main(String[] args){

A a = new A();

System.out.println("100-50="+a.func1(100, 50));

System.out.println("100-80="+a.func1(100, 80));

}

}

运行结果：

100-50=50

100-80=20

后来，我们需要增加一个新的功能：完成两数相加，然后再与100求和，由类B来负责。即类B需要完成两个功能：

两数相减；两数相加，然后再加100。

由于类A已经实现了第一个功能，所以类B继承类A后，只需要再完成第二个功能就可以了，代码如下：

class B extends A{

public int func1(int a, int b){

return a+b;

}

public int func2(int a, int b){

return func1(a,b)+100;

}

}

public class Client{

public static void main(String[] args){

B b = new B();

System.out.println("100-50="+b.func1(100, 50));

System.out.println("100-80="+b.func1(100, 80));

System.out.println("100+20+100="+b.func2(100, 20));

}

}

类B完成后，运行结果：

100-50=150

100-80=180

100+20+100=220

我们发现原本运行正常的相减功能发生了错误。原因就是类B在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类B重写后的方法，造成原本运行正常的功能出现了错误。在本例中，引用基类A完成的功能，换成子类B之后，发生了异常。在实际编程中，我们常常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合，组合等关系代替。

### 其他

面向对象的语言的三大特点就是继承、封装和多态，里氏替换原则就是依赖于继承和多态这两大特性，通俗来讲，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者异常，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。

**里氏替换原则通俗来讲：子类可以扩展父类的功能，但不能改变父类原有的功能。包含以下4层含义：**

* 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法；
* 子类中可以增加自己特有的方法；
* 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松；
* 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

看上去很不可思议，因为我们会发现在自己编程中常常会违反里氏替换原则，程序照样跑的好好的。所以大家都会产生这样的疑问，假如我非要不遵循里氏替换原则会有什么后果？后果就是：你写的代码出问题的几率将会大大增加。

## 依赖倒置原则，Dependence Inversion Principle，DIP

### 定义：

高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

### 问题由来：

类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。

### 解决方案：

将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在java中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

### 举例说明

依赖倒置原则的核心思想是面向接口编程，我们依旧用一个例子来说明面向接口编程比相对于面向实现编程好在什么地方。场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

class Book{

public String getContent(){

return "很久很久以前有一个阿拉伯的故事……";

}

}

class Mother{

public void narrate(Book book){

System.out.println("妈妈开始讲故事");

System.out.println(book.getContent());

}

}

public class Client{

public static void main(String[] args){

Mother mother = new Mother();

mother.narrate(new Book());

}

}

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

运行良好，假如有一天，需求变成这样：不是给书而是给一份报纸，让这位母亲讲一下报纸上的故事，报纸的代码如下：

class Newspaper{

public String getContent(){

return "林书豪38+7领导尼克斯击败湖人……";

}

}

这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改Mother才能读。假如以后需求换成杂志呢？换成网页呢？还要不断地修改Mother，这显然不是好的设计。原因就是Mother与Book之间的耦合性太高了，必须降低他们之间的耦合度才行。

我们引入一个抽象的接口IReader。读物，只要是带字的都属于读物：

interface IReader{

public String getContent();

}

Mother类与接口IReader发生依赖关系，而Book和Newspaper都属于读物的范畴，他们各自都去实现IReader接口，这样就符合依赖倒置原则了，代码修改为：

class Newspaper implements IReader {

public String getContent(){

return "林书豪17+9助尼克斯击败老鹰……";

}

}

class Book implements IReader{

public String getContent(){

return "很久很久以前有一个阿拉伯的故事……";

}

}

class Mother{

public void narrate(IReader reader){

System.out.println("妈妈开始讲故事");

System.out.println(reader.getContent());

}

}

public class Client{

public static void main(String[] args){

Mother mother = new Mother();

mother.narrate(new Book());

mother.narrate(new Newspaper());

}

}

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

妈妈开始讲故事

林书豪17+9助尼克斯击败老鹰……

这样修改后，无论以后怎样扩展Client类，都不需要再修改Mother类了。这只是一个简单的例子，实际情况中，代表高层模块的Mother类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。

### 其他

采用依赖倒置原则给多人并行开发带来了极大的便利，比如上例中，原本Mother类与Book类直接耦合时，Mother类必须等Book类编码完成后才可以进行编码，因为Mother类依赖于Book类。修改后的程序则可以同时开工，互不影响，因为Mother与Book类一点关系也没有。参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在很流行的TDD开发模式就是依赖倒置原则最成功的应用。

传递依赖关系有三种方式，以上的例子中使用的方法是接口传递，另外还有两种传递方式：构造方法传递和setter方法传递，相信用过Spring框架的，对依赖的传递方式一定不会陌生。

**在实际编程中，我们一般需要做到如下3点：**

* 低层模块尽量都要有抽象类或接口，或者两者都有。
* 变量的声明类型尽量是抽象类或接口。
* 使用继承时遵循里氏替换原则。

依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

## 接口隔离原则,InterfaceSegregation Principles，ISP

### 定义：

客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

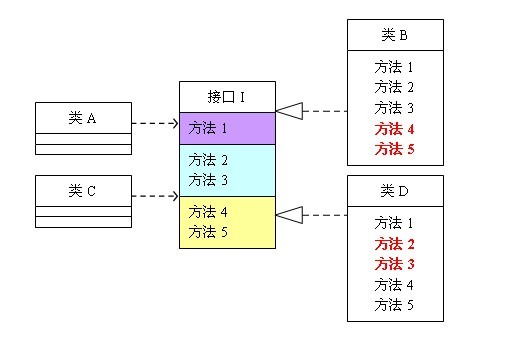
### 问题由来：

类A通过接口I依赖类B，类C通过接口I依赖类D，如果接口I对于类A和类B来说不是最小接口，则类B和类D必须去实现他们不需要的方法。

### 解决方案：

将臃肿的接口I拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。

### 举例来说明接口隔离原则：



这个图的意思是：类A依赖接口I中的方法1、方法2、方法3，类B是对类A依赖的实现。类C依赖接口I中的方法1、方法4、方法5，类D是对类C依赖的实现。对于类B和类D来说，虽然他们都存在着用不到的方法（也就是图中红色字体标记的方法），但由于实现了接口I，所以也必须要实现这些用不到的方法。对类图不熟悉的可以参照程序代码来理解，代码如下：

interface I {

public void method1();

public void method2();

public void method3();

public void method4();

public void method5();

}

class A{

public void depend1(I i){

i.method1();

}

public void depend2(I i){

i.method2();

}

public void depend3(I i){

i.method3();

}

}

class B implements I{

public void method1() {

System.out.println("类B实现接口I的方法1");

}

public void method2() {

System.out.println("类B实现接口I的方法2");

}

public void method3() {

System.out.println("类B实现接口I的方法3");

}

//对于类B来说，method4和method5不是必需的，但是由于接口A中有这两个方法，

//所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。

public void method4() {}

public void method5() {}

}

class C{

public void depend1(I i){

i.method1();

}

public void depend2(I i){

i.method4();

}

public void depend3(I i){

i.method5();

}

}

class D implements I{

public void method1() {

System.out.println("类D实现接口I的方法1");

}

//对于类D来说，method2和method3不是必需的，但是由于接口A中有这两个方法，

//所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。

public void method2() {}

public void method3() {}

public void method4() {

System.out.println("类D实现接口I的方法4");

}

public void method5() {

System.out.println("类D实现接口I的方法5");

}

}

public class Client{

public static void main(String[] args){

A a = new A();

a.depend1(new B());

a.depend2(new B());

a.depend3(new B());

C c = new C();

c.depend1(new D());

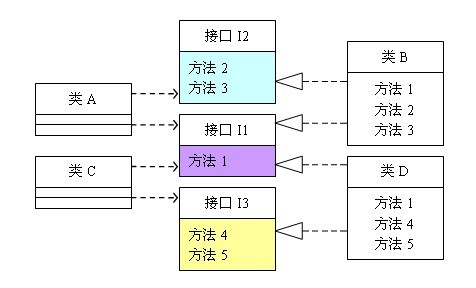
c.depend2(new D());

c.depend3(new D());

}

}

可以看到，如果接口过于臃肿，只要接口中出现的方法，不管对依赖于它的类有没有用处，实现类中都必须去实现这些方法，这显然不是好的设计。如果将这个设计修改为符合接口隔离原则，就必须对接口I进行拆分。在这里我们将原有的接口I拆分为三个接口，拆分后的设计如图2所示：



interface I1 {

public void method1();

}

interface I2 {

public void method2();

public void method3();

}

interface I3 {

public void method4();

public void method5();

}

class A{

public void depend1(I1 i){

i.method1();

}

public void depend2(I2 i){

i.method2();

}

public void depend3(I2 i){

i.method3();

}

}

class B implements I1, I2{

public void method1() {

System.out.println("类B实现接口I1的方法1");

}

public void method2() {

System.out.println("类B实现接口I2的方法2");

}

public void method3() {

System.out.println("类B实现接口I2的方法3");

}

}

class C{

public void depend1(I1 i){

i.method1();

}

public void depend2(I3 i){

i.method4();

}

public void depend3(I3 i){

i.method5();

}

}

class D implements I1, I3{

public void method1() {

System.out.println("类D实现接口I1的方法1");

}

public void method4() {

System.out.println("类D实现接口I3的方法4");

}

public void method5() {

System.out.println("类D实现接口I3的方法5");

}

}

### 其他

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。本文例子中，将一个庞大的接口变更为3个专用的接口所采用的就是接口隔离原则。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

说到这里，很多人会觉的接口隔离原则跟之前的单一职责原则很相似，其实不然。其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构建。

**采用接口隔离原则对接口进行约束时，要注意以下几点：**

* 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不挣的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。
* 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
* 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。

## 迪米特原则,Law of Demeter，LOD

### 定义：

一个对象应该对其他对象保持最少的了解。

### 问题由来：

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。

### 解决方案：

尽量降低类与类之间的耦合。自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

### 迪米特法则又叫最少知道原则

最早是在1987年由美国Northeastern University的Ian Holland提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

### 举一个例子：

有一个集团公司，下属单位有分公司和直属部门，现在要求打印出所有下属单位的员工ID。

#### 违反迪米特法则的设计

//总公司员工

class Employee{

private String id;

public void setId(String id){

this.id = id;

}

public String getId(){

return id;

}

}

//分公司员工

class SubEmployee{

private String id;

public void setId(String id){

this.id = id;

}

public String getId(){

return id;

}

}

//分公司管理员

class SubCompanyManager{

public List<SubEmployee> getAllEmployee(){

List<SubEmployee> list = new ArrayList<SubEmployee>();

for(int i=0; i<100; i++){

SubEmployee emp = new SubEmployee();

//为分公司人员按顺序分配一个ID

emp.setId("分公司"+i);

list.add(emp);

}

return list;

}

}

//总公司管理员

class CompanyManager{

public List<Employee> getAllEmployee(){

List<Employee> list = new ArrayList<Employee>();

for(int i=0; i<30; i++){

Employee emp = new Employee();

//为总公司人员按顺序分配一个ID

emp.setId("总公司"+i);

list.add(emp);

}

return list;

}

//打印所有员工----printAllEmployee(SubCompanyManager sub)

public void printAllEmployee(SubCompanyManager sub){

List<SubEmployee> list1 = sub.getAllEmployee();

for(SubEmployee e:list1){

System.out.println(e.getId());

}

List<Employee> list2 = this.getAllEmployee();

for(Employee e:list2){

System.out.println(e.getId());

}

}

}

//客户端

public class Client{

public static void main(String[] args){

CompanyManager e = new CompanyManager();

e.printAllEmployee(new SubCompanyManager());

}

}

现在这个设计的主要问题出在CompanyManager中，根据迪米特法则，只与直接的朋友发生通信，而SubEmployee类并不是CompanyManager类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。

#### 修改后的代码如下

class SubCompanyManager{

public List<SubEmployee> getAllEmployee(){

List<SubEmployee> list = new ArrayList<SubEmployee>();

for(int i=0; i<100; i++){

SubEmployee emp = new SubEmployee();

//为分公司人员按顺序分配一个ID

emp.setId("分公司"+i);

list.add(emp);

}

return list;

}

public void printEmployee(){

List<SubEmployee> list = this.getAllEmployee();

for(SubEmployee e:list){

System.out.println(e.getId());

}

}

}

class CompanyManager{

public List<Employee> getAllEmployee(){

List<Employee> list = new ArrayList<Employee>();

for(int i=0; i<30; i++){

Employee emp = new Employee();

//为总公司人员按顺序分配一个ID

emp.setId("总公司"+i);

list.add(emp);

}

return list;

}

public void printAllEmployee(SubCompanyManager sub){

sub.printEmployee();

List<Employee> list2 = this.getAllEmployee();

for(Employee e:list2){

System.out.println(e.getId());

}

}

}

修改后，为分公司增加了打印人员ID的方法，总公司直接调用来打印，从而避免了与分公司的员工发生耦合。

迪米特法则的初衷是降低类之间的耦合，由于每个类都减少了不必要的依赖，因此的确可以降低耦合关系。但是凡事都有度，虽然可以避免与非直接的类通信，但是要通信，必然会通过一个“中介”来发生联系，例如本例中，总公司就是通过分公司这个“中介”来与分公司的员工发生联系的。过分的使用迪米特原则，会产生大量这样的中介和传递类，导致系统复杂度变大。所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。

## 开闭原则，Open Close Principle，OCP

### 定义：

一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

### 问题由来：

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，可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。

### 解决方案：

当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。

开闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。开闭原则可能是设计模式六项原则中定义最模糊的一个了，它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。以前，如果有人告诉我“你进行设计的时候一定要遵守开闭原则”，我会觉的他什么都没说，但貌似又什么都说了。因为开闭原则真的太虚了。

在仔细思考以及仔细阅读很多设计模式的文章后，终于对开闭原则有了一点认识。其实，我们遵循设计模式前面其他5大原则，以及使用23种设计模式的目的就是遵循开闭原则。也就是说，只要我们对其他5项原则遵守的好了，设计出的软件自然是符合开闭原则的，这个开闭原则更像是其他五项原则遵守程度的“平均得分”，前面5项原则遵守的好，平均分自然就高，说明软件设计开闭原则遵守的好；如果其他5项原则遵守的不好，则说明开闭原则遵守的不好。

其实笔者认为，开闭原则无非就是想表达这样一层意思：用抽象构建框架，用实现扩展细节。因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

### 再回想一下其他的5项原则

其他5项原则恰恰是告诉我们用抽象构建框架，用实现扩展细节的注意事项而已：

* 单一职责原则告诉我们实现类要职责单一；
* 里氏替换原则告诉我们不要破坏继承体系；
* 依赖倒置原则告诉我们要面向接口编程；
* 接口隔离原则告诉我们在设计接口的时候要精简单一；
* 迪米特法则告诉我们要降低耦合。

而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。

最后说明一下如何去遵守这六个原则。对这六个原则的遵守并不是是和否的问题，而是多和少的问题，也就是说，我们一般不会说有没有遵守，而是说遵守程度的多少。任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。

# ----外卡型----对象池模式（Object Pool）

## 对象池模式（Object Pool Pattern）

这个模式为常见 23 种设计模式之外的设计模式，介绍的初衷主要是在平时的 Android 开发中经常会看到，比如 ThreadPool 和 MessagePool 等。

　　在 java 中，所有对象的内存由虚拟机管理，所以在某些情况下，需要频繁创建一些生命周期很短使用完之后就可以立即销毁，但是数量很大的对象集合，那么此时 GC 的次数必然会增加，这时候为了减小系统 GC 的压力，对象池模式就很适用了。对象池模式也是创建型模式之一，它不是根据使用动态的分配和销毁内存，而是维护一个已经初始化好若干对象的对象池以供使用。客户端使用的时候从对象池中去申请一个对象，当该对象使用完之后，客户端会返回给对象池，而不是立即销毁它，这步操作可以手动或者自动完成。

从 Java 语言的特性来分析一下，在 Java 中，对象的生命周期大致包括三个阶段：对象的创建，对象的使用，对象的清除。因此，对象的生命周期长度可用如下的表达式表示：T = T1 + T2 +T3。其中T1表示对象的创建时间，T2 表示对象的使用时间，而 T3 则表示其清除时间。由此，我们可以看出，只有 T2 是真正有效的时间，而 T1、T3 则是对象本身的开销。下面再看看 T1、T3 在对象的整个生命周期中所占的比例。Java对象是通过构造函数来创建的，在这一过程中，该构造函数链中的所有构造函数也都会被自动调用。另外，默认情况下，调用类的构造函数时，Java 会把变量初始化成确定的值：所有的对象被设置成 null，整数变量（byte、short、int、long）设置成 0， float 和 double 变量设置成 0.0，逻辑值设置成false。所以用new关键字来新建一个对象的时间开销是很大的，如下表所示：

|  |  |  |
| --- | --- | --- |
| **运算操作** | **示例** | **标准化时间** |
| 本地赋值 | i = n | 1.0 |
| 实例赋值 | this.i = n | 1.2 |
| 方法调用 | Funct() | 5.9 |
| 新建对象 | New Object() | 980 |
| 新建数组 | New int[10] | 3100 |

从表中可以看出，新建一个对象需要980个单位的时间，是本地赋值时间的980倍，是方法调用时间的166倍，而若新建一个数组所花费的时间就更多了。

　　再看清除对象的过程，我们知道，Java 语言的一个优势，就是 Java 程序员勿需再像 C/C++ 程序员那样，显式地释放对象，而由称为垃圾收集器（Garbage Collector）的自动内存管理系统，定时或在内存凸现出不足时，自动回收垃圾对象所占的内存。凡事有利总也有弊，这虽然为 Java 程序设计者提供了极大的方便，但同时它也带来了较大的性能开销。这种开销包括两方面，首先是对象管理开销，GC为了能够正确释放对象，它必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等。其次，在 GC 开始回收“垃圾”对象时，系统会暂停应用程序的执行，而独自占用 CPU。

因此，如果要改善应用程序的性能，一方面应尽量减少创建新对象的次数；同时，还应尽量减少 T1、T3 的时间，而这些均可以通过对象池技术来实现。所以对象池主要是用来提升性能，在某些情况下，对象池对性能有极大的帮助。但是还有一点需要注意，对象池会增加对象生命周期的复杂度，这是因为从对象池获取的对象和返还给对象池的对象都没有真正的创建或者销毁。

## 特点

　　维护一定数量的对象集合，使用时直接向对象池申请，以跳过对象的”expensive initialization”。

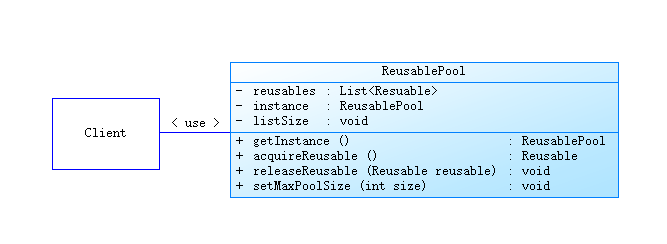
　　从上面的介绍可以总结：对象池模式适用于“需要使用到大量的同一类对象，这些对象的初始化会消耗大量的系统资源，而且它们只需要使用很短的时间，这种操作会对系统的性能有一定影响”的情况，总结一下就是在下面两种分配模式下可以选择使用对象池：

* 对象以固定的速度不断地分配，垃圾收集时间逐步增加，内存使用率随之增大；
* 对象分配存在爆发期，而每次爆发都会导致系统迟滞，并伴有明显的GC中断。

在绝大多数情况下，这些对象要么是数据容器，要么是数据封装器，其作用是在应用程序和内部消息总线、通信层或某些API之间充当一个信封。这很常见。例如，数据库驱动会针对每个请求和响应创建 Request 和 Response 对象，消息系统会使用 Message 和 Event 封装器，等等。对象池可以帮助保存和重用这些构造好的对象实例。

对象池模式会预先创建并初始化好一个对象集合用于重用，当某处需要一个该类型的新对象时，它直接向对象池申请，如果此时对象池中有已经预先初始化好的对象，它就直接返回，如果没有，对象池就会创建一个并且返回；当使用完该对象之后，它会将该对象返还给对象池，对象池会将其重用以避免该对象笨重的初始化操作。有一点需要注意的是，一旦一个对象被对象池重用返回给需要使用的地方之后，该对象已经存在引用都将会变成非法不可使用。需要注意的是，在一些系统资源很紧缺的系统上，对象池模式将会限制对象池最大的大小，当已经达到最大的数量之后，再次获取可能会抛出异常或者直接被阻塞直到有一个对象被对象池回收

## UML类图



Reusable：

对象类，该对象在实际使用中需要频繁的创建和销毁，并且初始化操作会很很消耗时间和性能；

Client：

使用 Reusable 类对象的角色；

ReusablePool：

管理 Reusable 对象类的所有对象，供 Client 角色使用。

通常情况下，我们希望所有的 Reusable 对象都会被 ResuablePool 管理，所以可以使用单例模式构建 ReusablePool，当然其他的工厂方法模式也是可以的 。Client 调用 getInstance 方法获取到 ReusablePool 的对象，接着调用 acquireReusable 方法去获取 Reusable 对象，使用完之后调用 releaseReusable 方法传入该对象重新交给 ReusablePool 管理。

## MessagePool源码分析

　在 android 中使用 new Message() ， Message.obtain() 和 Handler.obtainMessage() 都能够获得一个 Message 对象，但是为什么后两个的效率会高于前者呢？先来看看源码的 Message.obtain() 函数：

public static Message obtain() {

synchronized (sPoolSync) {

if (sPool != null) {

Message m = sPool;

sPool = m.next;

m.next = null;

m.flags = 0; // clear in-use flag

sPoolSize--;

return m;

}

}

return new Message();

}

返回的是 sPool 这个对象，继续看看 Handler.obtainMessage() 函数：

public final Message obtainMessage() {

return Message.obtain(this);

}

一样是调用到了 Message.obtain() 函数，那么我们就从这个函数开始分析， Message 类是如何构建 MessagePool 这个角色的呢？看看这几处的代码：

//变量的构建

private static final Object sPoolSync = new Object();

private static Message sPool;

private static int sPoolSize = 0;

private static final int MAX\_POOL\_SIZE = 50;

private static boolean gCheckRecycle = true;

Message 对象的回收

public void recycle() {

if (isInUse()) {

if (gCheckRecycle) {

throw new IllegalStateException("This message cannot be recycled because it "

+ "is still in use.");

}

return;

}

recycleUnchecked();

}

void recycleUnchecked() {

// Mark the message as in use while it remains in the recycled object pool.

// Clear out all other details.

flags = FLAG\_IN\_USE;

what = 0;

arg1 = 0;

arg2 = 0;

obj = null;

replyTo = null;

sendingUid = -1;

when = 0;

target = null;

callback = null;

data = null;

synchronized (sPoolSync) {

if (sPoolSize < MAX\_POOL\_SIZE) {

next = sPool;

sPool = this;

sPoolSize++;

}

}

}

从这几处代码可以很清楚的看到：

sPool 这个静态对象实际上是相当于构建了一个长度为 MAX\_POOL\_SIZE 的链表， recycleUnchecked 方法以内置锁的方式（线程安全），判断当前对象池的大小是否小于50，若小于50，则会被加到链表的头部，并把 next 对象指向上一次链表的头部；若大于等于50，则直接丢弃掉，那么这些被丢弃的Message将交由GC处理。

recycleUnchecked 方法将待回收的Message对象字段先置空，以便节省 MessagePool 的内存和方便下一次的直接使用；

源码这么一看， android 中 MessagePool 其实实现方式还是很简单的。

## 示例

　我们参照上面的 uml 类图写一个 demo，首先要定义 Reusable 这个角色，为了直观，我们将该类定义了很多成员变量，用来模拟 expensive initialization：

**Reusable.class**

**public** **class** **Reusable** {

**public** **String** a;

**public** **String** b;

**public** **String** c;

**public** **String** d;

**public** **String** e;

**public** **String** f;

**public** **String** g;

**public** **ArrayList**<**String**> h;

**public** **ArrayList**<**String**> i;

**public** **ArrayList**<**String**> j;

**public** **ArrayList**<**String**> k;

**public** **ArrayList**<**String**> l;

**public** **Reusable**(){

h = **new** ArrayList<>();

i = **new** ArrayList<>();

j = **new** ArrayList<>();

k = **new** ArrayList<>();

l = **new** ArrayList<>();

}

}

**然后用一个 IReusablePool.class接口来定义对象池的基本行为：**

**public** **interface** IReusablePool {

**Reusable** **requireReusable**();

**void** **releaseReusable**(**Reusable** reusable);

**void** **setMaxPoolSize**(**int** size);

}

**实现该接口：**

**public** **class** **ReusablePool** **implements** IReusablePool {

**private** **static** **final** **String** **TAG** = "ReusablePool";

**private** **static** **volatile** **ReusablePool** ***instance*** = **null**;

**private** **static** List<**Reusable**> ***available*** = **new** ArrayList<>();//可用

**private** **static** List<**Reusable**> ***inUse*** = **new** ArrayList<>();//在用

**private** **static** **final** **byte**[] **lock** = **new** **byte**[]{};

**private** **static** **int** ***maxSize*** = 5;

**private** **int** currentSize = 0;

**private** **ReusablePool**() {

***available*** = **new** ArrayList<>();

***inUse*** = **new** ArrayList<>();

}

**public** **static** **ReusablePool** **getInstance**() {

**if** (***instance*** == **null**) {

**synchronized** (**ReusablePool**.**class**) {

**if** (***instance*** == **null**) {

***instance*** = **new** ReusablePool();

}

}

}

**return** ***instance***;

}

@Override

**public** **Reusable** **requireReusable**() {

**synchronized** (**lock**) {//同步的

**if** (currentSize >= ***maxSize***) {

**throw** **new** RuntimeException("pool has gotten its maximum size");

}

**if** (***available***.size() > 0) {//可用集合大小大于0

**Reusable** reusable = ***available***.get(0);//可用得到集合头部对象

***available***.remove(0);//移除头部对象

currentSize++;//在使用大小自增1

***inUse***.add(reusable);//加入在用集合

**return** reusable;//返回对象

} **else** {//可用集合为空

**Reusable** reusable = **new** Reusable();//新建一个对象

***inUse***.add(reusable);

currentSize++;

**return** reusable;

}

}

}

@Override

**public** **void** **releaseReusable**(**Reusable** reusable) {

**if** (reusable != **null**) {

reusable.a = **null**;

reusable.b = **null**;

reusable.c = **null**;

reusable.d = **null**;

reusable.e = **null**;

reusable.f = **null**;

reusable.g = **null**;

reusable.h.clear();//只是清除集合中的对象，后人使用时集合已初始化，减少了初始化时间

reusable.i.clear();

reusable.j.clear();

reusable.k.clear();

reusable.l.clear();

}

**synchronized** (**lock**) {//同步的

***inUse***.remove(reusable);//在用集合中移除

***available***.add(reusable);//可用集合中添加

currentSize--;//在用大小自减1

}

}

@Override

**public** **void** **setMaxPoolSize**(**int** size) {

**synchronized** (**lock**) {//同步的

***maxSize*** = size;

}

}

}

**最后测试程序：**

**new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**try** {

**Reusable** reusable = **ReusablePool**.*getInstance*().requireReusable();

**Log**.*e*("CORRECT", "get a Reusable object " + reusable);

**Thread**.*sleep*(5000);

**ReusablePool**.*getInstance*().releaseReusable(reusable);

}**catch** (**Exception** e){

**Log**.*e*("ERROR", e.getMessage());

}

}

}).start();

**需要说明的是：**

* releaseReusable 方法不要忘记将 Reusable 对象中的成员变量重置，要不然下次使用会有问题；
* 模拟了 5s 的延时，在 demo 中如果达到了 maxPoolSize ，继续操作是抛出 RunTimeException ，这个在实际使用该过程中可以按照情况更改（可以抛出异常，新建一个对象返回或者直接在多线程模式下阻塞，直到有新对象，参考链接： https://en.wikipedia.org/wiki/Object\_pool\_pattern#Handling\_of\_empty\_pools）；
* 在程序中使用的是单例模式获取的 ReusablePool 对象，这个在实际使用过程中也可以换成工厂方法模式等的其他设计模式。

## 总结

对象池模式从上面来分析可以说是用处很大，但是这个模式一定要注意使用的场景，也就是最上面提到的两点：“对象以固定的速度不断地分配，垃圾收集时间逐步增加，内存使用率随之增大”和“对象分配存在爆发期，而每次爆发都会导致系统迟滞，并伴有明显的GC中断”，其他的一般情况下最好不要使用对象池模式，我们后面还会提到它的缺点和很多人对其的批判。

## 优点

优点上面就已经阐述的很清楚了，对于那些”the rate of instantiation and destruction of a class is high” 和 “the cost of initializing a class instance is high”的场景优化是及其明显的，例如 database connections，socket connections，threads 和类似于 fonts 和 Bitmap 之类的对象。

## 陷阱

上面已经提过了，使用对象池模式时，每次进行回收操作前都需要将该对象的相关成员变量重置，如果不重置将会导致下一次重复使用该对象时候出现预料不到的错误，同时的，如果回收对象的成员变量很大，不重置还可能会出现内存 OOM 和信息泄露等问题。另外的，对象池模式绝大多数情况下都是在多线程中访问的，所以做好同步工作也是极其重要的。原文：https://en.wikipedia.org/wiki/Object\_pool\_pattern#Pitfalls

除了以上两点之外，还需要注意以下问题：

* 引用泄露：对象在系统中某个地方注册了，但没有返回到池中。
* 过早回收：消费者已经决定将对象返还给对象池，但仍然持有它的引用，并试图执行写或读操作，这时会出现这种情况。
* 隐式回收：当使用引用计数时可能会出现这种情况。
* 大小错误：这种情况在使用字节缓冲区和数组时非常常见：对象应该有不同的大小，而且是以定制的方式构造，但返回对象池后却作为通用对象重用。
* 重复下单：这是引用泄露的一个变种，存在多路复用时特别容易发生：一个对象被分配到多个地方，但其中一个地方释放了该对象。
* 就地修改：对象不可变是最好的，但如果不具备那样做的条件，就可能在读取对象内容时遇到内容被修改的问题。
* 缩小对象池：当池中有大量的未使用对象时，要缩小对象池。

对于如何根据使用情况放大和缩小对象池的大小，一个广为人知但鲜有人用的技巧：对象池 这篇文章中讲述的很清楚，感兴趣的可以看看

## 讨论与批判

一些开发者不建议在某些语言例如 Java 中使用对象池模式，特别是对象只使用内存并且不会持有其他资源的语言。这些反对者持有的观点是 new 的操作只需要 10 条指令，而使用对象池模式则需要成百上千条指令，明显增加了复杂度，而且 GC 操作只会去扫描“活着”的对象引用所指向的内存，而不是它们的成员变量所使用的那块内存，这就意味着，任何没有引用的“死亡”对象都能够被 GC 以很小的代价跳过，相反如果使用对象池模式，持有大量“活着“的对象反而会增加 GC 的时间。原文：<https://en.wikipedia.org/wiki/Object_pool_pattern#Criticism>

# ----创建型----[原型模式](https://www.baidu.com/s?wd=%E5%8E%9F%E5%9E%8B%E6%A8%A1%E5%BC%8F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3rjIbPj6zPjTsmHczmvc10ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1TvPjfYnHDd" \t "http://zhidao.baidu.com/_blank)（Prototype）

## 原型模式（Prototype Pattern）

该模式有一个样板实例，用户从这个样板对象中复制出一个内部属性一致的对象，这个过程在 C++ 中就是一个克隆。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以使程序运行效率更高。   
　　这个模式的重点在于，客户端的代码在不知道要实例化何种特定类的情况下，就可以制造出新的实例。

## 特点

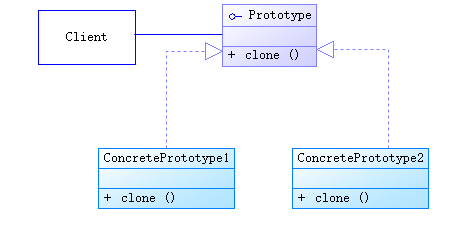
用原型实例指向创建对象的种类，并通过拷贝这些原型创建新的对象。

原型模式的适用场景：

* 类初始化需要消耗很多的资源，这个资源包括数据，硬件资源等，简而言之，这个实例化的过程很昂贵时，就可以使用原型模式；
* 频繁的创建相似的对象时，比如需要在一个循环体内创建对象，假如对象创建过程比较复杂或者循环次数很多的话，使用原型模式不但可以简化创建过程，而且可以使系统的整体性能提高很多；
* 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；
* 一个对象需要提供给其他对象访问，并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；
* 避免在客户端调用一个对象子类的生成器，即和抽象工厂模式一样，子类的生成器对客户端不透明。

需要注意的，通过实现 Cloneable 接口的原型模式在调用 clone 函数构造实例并不一定比通过 new 操作速度快，只有通过 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。

## UML类图



Client：

客户端用户；

Prototype：

抽象类或者接口，声明具备 clone 的能力；

ConcretePrototype：

具体的原型类。

为了实现原型模式，首先要声明一个虚基类或者接口，该类中有一个单纯的虚方法 clone，任何一个需要多态构造函数性质的类都需要继承自该虚类或者实现该接口，并且实现 clone 方法；然后客户端不直接调用类的构造函数，而是调用原型类的 clone 方法用来生成另一个对象，或者使用其他设计模式提供的功能来调用 clone 方法，比如工厂模式等。

## 原型模式的通用代码：

**ConcretePrototype.class**

**public** **class** **ConcretePrototype** **implements** Cloneable{

**private** **String** string;

**private** **ArrayList**<**String**> stringList;

**public** **ConcretePrototype**() {

stringList = **new** ArrayList<>();

}

**public** **String** **getString**() {

**return** string;

}

**public** **void** **setString**(**String** string) {

**this**.string = string;

}

**public** **ArrayList**<**String**> **getStringList**() {

**return** stringList;

}

**public** **void** **setStringList**(**ArrayList**<**String**> stringList) {

**this**.stringList = stringList;

}

**public** **ConcretePrototype** **clone**() {

**try** {

**ConcretePrototype** copy = (**ConcretePrototype**) **super**.clone();

copy.setString(**this**.getString());

copy.setStringList((**ArrayList**<**String**>) getStringList().clone());

**return** copy;

} **catch** (**CloneNotSupportedException** e) {

e.printStackTrace();

}

**return** **null**;

}

}

我们直接使用系统的 Cloneable 接口来作为 Prototype 角色，接着重写Object类中的clone方法。Java 中，所有类的父类都是 Object 类，Object 类中有一个 clone 方法，作用是返回对象的一个拷贝，但是其作用域 protected 类型的，一般的类无法调用，因此，Prototype 类需要将 clone 方法的作用域修改为 public 类型。

**测试代码：**

**ConcretePrototype** src = **new** ConcretePrototype();

src.setString("src");

src.getStringList().add("src 1");

src.getStringList().add("src 2");

**ConcretePrototype** des = src.clone();

des.setString("des");

des.getStringList().add("des 1");

des.getStringList().add("des 2");

**Log**.*e*("shawn", "src.string="+src.getString() +"；des.string = "+des.getString());//src.string = src；des.string = des

**StringBuilder** builder = **new** StringBuilder();

**for** (**String** temp : src.getStringList()) {

builder.append(temp).append(" ");

}

**Log**.*e*("shawn", "src.stringList = " + builder.toString());//src.stringList = src 1 src 2

builder = **new** StringBuilder();

**for** (**String** temp : des.getStringList()) {

builder.append(temp).append(" ");

}

**Log**.*e*("shawn", "des.stringList = " + builder.toString());//des.stringList = src 1 src 2 des 1 des 2

**上面的代码很简单，使用了 Cloneable 接口，但是其实 Cloneable 接口就是一个空类：**

/\*\*

\* This (empty) interface must be implemented by all classes that wish to

\* support cloning. The implementation of {@code clone()} in {@code Object}

\* checks if the object being cloned implements this interface and throws

\* {@code CloneNotSupportedException} if it does not.

\*

\* **@see** Object#clone

\* **@see** CloneNotSupportedException

\*/

**public** **interface** Cloneable {

// Marker interface

}

**注释很清楚了，是用来赋予一个类的 clone 功能的，继续看看 Object 类的 clone 函数：**

**protected** **Object** **clone**() **throws** CloneNotSupportedException {

**if** (!(**this** **instanceof** Cloneable)) {

**throw** **new** CloneNotSupportedException("Class " + getClass().getName() + " doesn't implement Cloneable");

}

**return** internalClone();

}

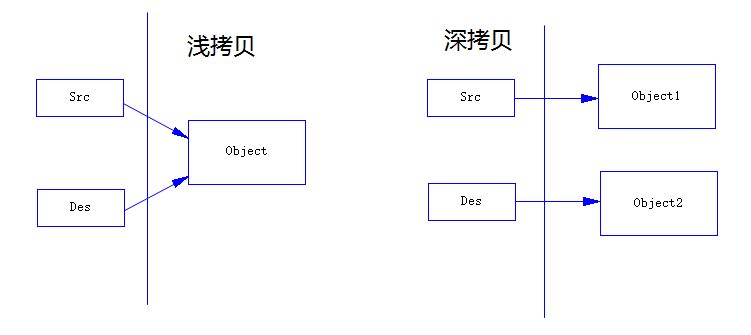
这也是为什么需要继承 Cloneable 接口的原因，不继承该接口这里就会直接抛出 CloneNotSupportedException 异常，internalClone 是一个 native 函数：

private native Object internalClone();

它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。

## 深拷贝和浅拷贝

　这里需要注意的是浅拷贝（影子拷贝）与深拷贝的问题，学过 C++ 对拷贝都应该印象很深，在上面的例子中用的是一个 String 和一个 ArrayList 的对象，对于这两个对象，clone 函数里的拷贝写法是不一样的，一个是直接 set ，另一个需要继续调用 ArrayList 的 clone 方法生成一个 ArrayList 的拷贝对象 set 进 copy 对象中，如果直接将源对象中的 ArrayList 对象 set 进 copy 对象中，就会造成客户端获取到 copy 对象之后，可以通过 copy 对象修改源对象的数据，这在保护原型模式中是绝对不允许的，所以这里不能使用浅拷贝，必须要使用深拷贝。Object 类的 clone 方法只会拷贝对象中的 8 种基本数据类型 ，byte 、char、short、int、long、float、double、boolean，对于数组、容器对象、引用对象等都不会拷贝，这就是浅拷贝，如果要实现深拷贝，必须将原型模式中的数组、容器对象、引用对象等另行拷贝，如上面的 ArrayList 一样。String 这个类型必须要单独拿出来说一下，这个类型实际上算是一个浅拷贝，因为 src 与 拷贝后的 copy 对象指向的是同一个内存区域，但是由于 Java 中 String 的特殊不可变性（除去反射之外，不能修改一个 String 对象所指向的字符串内容），具体可以参考这篇博客：[Java中的String为什么是不可变的？ – String源码分析](http://blog.csdn.net/zhangjg_blog/article/details/18319521)，所以从实际表现效果来看，String 和 8 种基础类型一样，可以认为是深拷贝。



## 示例与源码

原型模式的代码结构很简单，我们这就以 Android 中的 Intent 类为例，来简单了解一下 Intent 类的原型模式：

**Intent.class**

/\*\*

\* Copy constructor.

\*/

**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;

**this**.mContentUserHint = o.mContentUserHint;

**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);

}

}

@Override

**public** **Object** **clone**() {

**return** **new** Intent(**this**);//构造函数中原始对象作为参数将原始对象的数据逐个拷贝一遍

}

和我们平时使用的 clone 函数不一样，Intent 类并没有调用 super.clone ，而是专门写了一个拷贝构造函数用来克隆对象，我们在上文提到过，使用 clone 和 new 需要根据构造函数的成本来决定，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，然后在构造函数中将原始对象的数据逐个拷贝一遍，这样，整个克隆过程就完成了

## 总结

原型模式是非常简单的一个设计模式，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，应该尽量使用深拷贝，避免操作副本时影响原始对象的问题。

同时需要特别注意的是使用原型模式复制对象不会调用类的构造方法。因为对象的复制是通过调用 Object 类的 clone 方法来完成的，它直接在内存中复制数据，因此不会调用到类的构造方法。不但构造方法中的代码不会执行，甚至连访问权限都对原型模式无效。

原型模式的优点和缺点基本也就明了了

## 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，原型模式可以更好的体现其优点，而且可以向客户端隐藏制造新实例的复杂性；

## 缺点

直接在内存中拷贝，构造函数是不会执行的，在实际开发中应该注意这个潜在的问题，另外，对象的复制有时候相当复杂，特别需要注意的是不彻底深拷贝的问题。

# 单例模式（Singleton）

## 特点：

1. 单例设计模式保证一个类只有一个实例。

2. 要提供一个访问该类对象实例的全局访问点。

单例模式的使用很广泛，比如：线程池（threadpool）、缓存（cache）、对话框、处理偏好设置、和注册表（registry）的对象、日志对象，充当打印机、显卡等设备的驱动程序的对象等，这些类的对象只能有一个实例，如果制造出多个实例，就会导致很多问题的产生，程序的行为异常，资源使用过量，或者不一致的结果等，所以单例模式最主要的特点：

* 构造函数不对外开放，一般为private；
* 通过一个静态方法或者枚举返回单例类对象；
* 确保单例类的对象有且只有一个，尤其是在多线程的环境下；
* 确保单例类对象在反序列化时不会重新构建对象。

通过将单例类构造函数私有化，使得客户端不能通过 new 的形式手动构造单例类的对象。单例类会暴露一个共有静态方法，客户端需要条用这个静态方法获取到单例类的唯一对象，在获取到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，这是单例模式较关键的一个地方。

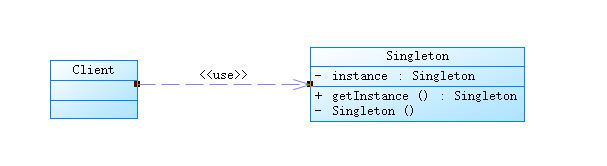
## 主要优点

* 单例模式提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它。
* 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象单例模式无疑可以提高系统的性能。
* 允许可变数目的实例。基于单例模式我们可以进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例，既节省系统资源，又解决了单例对象共享过多有损性能的问题。

## 主要缺点

* 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。
* 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品的本身的功能融合到一起。
* 现在很多面向对象语言(如Java、C#)的运行环境都提供了自动垃圾回收的技术，因此，如果实例化的共享对象长时间不被利用，系统会认为它是垃圾，会自动销毁并回收资源，下次利用时又将重新实例化，这将导致共享的单例对象状态的丢失。
* 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。

## UML类图



类图很简单，Singleton 类有一个 static 的 instance对象，类型为 Singleton ，构造函数为 private，提供一个 getInstance() 的静态函数，返回刚才的 instance 对象，在该函数中进行初始化操作

## 示例与源码

写法很多，总结一下：

### 懒汉法，线程不安全,lazy initialization, thread-unsafety

**public** **class** **Single** {

**private** **static** **Single** ***mInstance = null***;

**private** **Single**(){

}

**public** **static** **Single** **getInstance**(){

**if** (***mInstance*** == **null**) {

***mInstance*** = **new** Single();

}

**return** ***mInstance***;

}

}

### 懒汉法,线程安全,lazy initialization, thread-safety, double-checked

需要做到线程安全，就需要确保任意时刻只能有且仅有一个线程能够执行new Singleton对象的操作，所以可以在getInstance()函数上加上 synchronized 关键字，类似于：

**public** **static** **synchronized** **Single** **getInstance**(){

**if** (***mInstance*** == **null**) {

***mInstance*** = **new** Single();

}

**return** ***mInstance***;

}

但是套用《Head First》上的一句话，对于绝大部分不需要同步的情况来说，synchronized 会让函数执行效率糟糕一百倍以上（Since synchronizing a method could in some extreme cases decrease performance by a factor of 100 or higher），所以就有了double-checked（双重检测）的方法,不会受到 synchronized 的性能影响:

**public** **class** **Single** {

**private volatile static Single** ***mInstance = null***;

**private** **Single**(){

}

**public** **static** **Single** **getInstance**(){

**if** (***mInstance*** == **null**) {

**synchronized** (**Single**.**class**) {

**if** (***mInstance*** == **null**) {

***mInstance*** = **new** Single();

}

}

}

**return** ***mInstance***;

}

}

### 饿汉法，线程安全,eager initialization thread-safety

**public** **class** **Single** {

**private** **static** **Single** ***mInstance*** = **new** Single();

**private** **Single**(){

}

**public** **static** **Single** **getInstance**(){

**return** ***mInstance***;

}

}

**或者**

**public** **class** **Single** {

**private** **static** **Single** ***mInstance*** = **null**;

**static** {

***mInstance*** = **new** Single();

}

**private** **Single**(){

}

**public** **static** **Single** **getInstance**(){

**return** ***mInstance***;

}

}

代码都很简单，一个是直接进行初始化，另一个是使用静态块进行初始化，目的都是一个：在该类进行加载的时候就会初始化该对象，而不管是否需要该对象。这么写的好处是编写简单，而且是线程安全的，但是这时候初始化instance显然没有达到lazy loading的效果。

### 静态内部类，线程安全,static inner class thread-safety

由于在java中，静态内部类是在使用中初始化的，所以可以利用这个天生的延迟加载特性，去实现一个简单，延迟加载，线程安全的单例模式：

**public** **class** **Single** {

**public** **Single**() {

}

**private** **static** **class** **SingleHolder** {

**public** **static** **final** **Single** **mInstance** = **new** Single();

}

**public** **static** **Single** **getInstance**() {

**return** **SingleHolder**.**mInstance**;

}

}

定义一个 SingletonHolder 的静态内部类，在该类中定义一个外部类 Singleton 的静态对象，并且直接初始化，在外部类 Singleton 的 getInstance() 方法中直接返回该对象。由于静态内部类的使用是延迟加载机制，所以只有当线程调用到 getInstance() 方法时才会去加载 SingletonHolder 类，加载这个类的时候又会去初始化 instance 变量，所以这个就实现了延迟加载机制，同时也只会初始化这一次，所以也是线程安全的，写法也很简单。

**PS上面提到的所有实现方式都有两个共同的缺点：**

都需要额外的工作(Serializable、transient、readResolve())来实现序列化，否则每次反序列化一个序列化的对象实例时都会创建一个新的实例。

可能会有人使用反射强行调用我们的私有构造器（如果要避免这种情况，可以修改构造器，让它在创建第二个实例的时候抛异常）。

### 枚举法,线程安全,enum thread-safety

JDK1.5 之后加入 enum 特性，可以使用 enum 来实现单例模式：

**enum** SingleEnum{

**INSTANCE**("enum singleton thread-safety");

**private** **String** name;

**SingleEnum**(**String** name){

**this**.name = name;

}

**public** **String** **getName**(){

**return** name;

}

}

使用枚举除了线程安全和防止反射强行调用构造器之外，还提供了自动序列化机制，防止反序列化的时候创建新的对象。因此，Effective Java推荐尽可能地使用枚举来实现单例。但是很不幸的是 android 中并不推荐使用 enum ，主要是因为在 java 中枚举都是继承自 java.lang.Enum 类，首次调用时，这个类会调用初始化方法来准备每个枚举变量。每个枚举项都会被声明成一个静态变量，并被赋值。在实际使用时会有点问题

### 登记式

登记式单例实际上维护了一组单例类的实例，将这些实例存放在一个Map（登记薄）中，对于已经登记过的实例，则从Map直接返回，对于没有登记的，则先登记，然后返回。这种方式极少见到，另外其实内部实现还是用的饿汉式单例，因为其中的static方法块，它的单例在类被装载的时候就被实例化了。

//类似Spring里面的方法，将类名注册，下次从里面直接获取。

**public** **class** **Singleton** {

**private** **static** Map<**String**,**Singleton**> ***map*** = **new** HashMap<**String**,**Singleton**>();

**static**{

**Singleton** single = **new** Singleton();

***map***.put(single.getClass().getName(), single);

}

//保护的默认构造函数

**protected** **Singleton**(){}

//静态工厂方法,返还此类惟一的实例

**public** **static** **Singleton** **getInstance**(**String** name) {

**if**(name == **null**) {

name = **Singleton**.**class**.getName();

**System**.**out**.println("name == null"+"--->name="+name);

}

**if**(***map***.get(name) == **null**) {

**try** {

***map***.put(name, (**Singleton**) **Class**.*forName*(name).newInstance()); //使用反射创建新对象

} **catch** (**InstantiationException** e) {

e.printStackTrace();

} **catch** (**IllegalAccessException** e) {

e.printStackTrace();

} **catch** (**ClassNotFoundException** e) {

e.printStackTrace();

}

}

**return** ***map***.get(name);

}

//一个示意性的方法

**public** **String** **about**() {

**return** "Hello, I am RegSingleton.";

}

}

## 完美双重检验法

**public** **class** **SingleInstance** {

**private** **volatile** **static** **SingleInstance** ***mInstance***;

**private** **SingleInstance**() {

}

**public** **static** **SingleInstance** **getInstance**() {

**if** (***mInstance*** == **null**) {

**synchronized** (**SingleInstance**.**class**) {

**if** (***mInstance*** == **null**) {

***mInstance*** = **new** SingleInstance();

}

}

}

**return** ***mInstance***;

}

}

## 总结

　　综上所述，平时在 android 中使用 double-checked 或者 SingletonHolder 都是可以的，由于 android 中的多进程机制，在不同进程中无法创建同一个 instance 变量，就像 Application 类会初始化两次一样，这点需要注意。

　　但是不管采取何种方案，请时刻牢记单例的三大要点：

* 线程安全；
* 延迟加载；
* 序列化与反序列化安全

# 工厂方法（Factory Method）



在实际开发过程中我们都习惯于直接使用 new 关键字用来创建一个对象，可是有时候对象的创造需要一系列的步骤：你可能需要计算或取得对象的初始设置；选择生成哪个子对象实例；或在生成你需要的对象之前必须先生成一些辅助功能的对象，这个时候就需要了解该对象创建的细节，也就是说使用的地方与该对象的实现耦合在了一起，不利于扩展，为了解决这个问题就需要用到我们的工厂方法模式，它适合那些创建复杂的对象的场景，工厂方法模式也是一个使用频率很高的设计模式。

## 特点

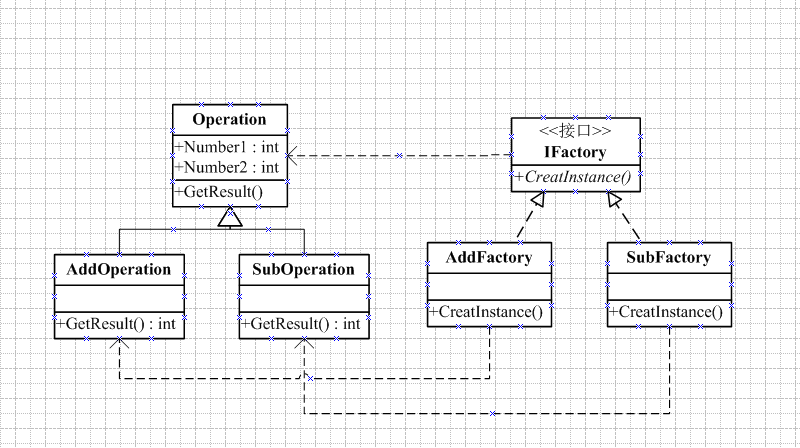
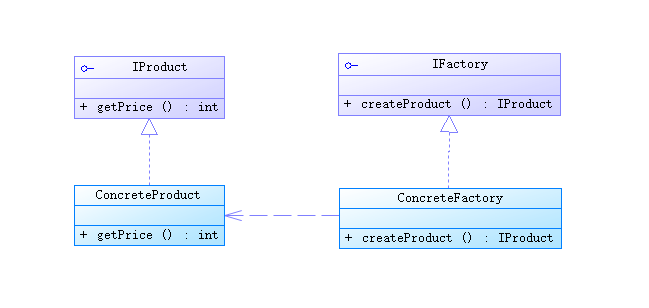
工厂方法模式（Factory Method Pattern）定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个，工厂方法让类把实例化推迟到子类，这样的设计将对象的创建封装其来，以便于得到更松耦合，更有弹性的设计。

工厂方法模式是创建型设计模式之一，是结构较为简单的一种模式，在我们平时的开发过程中应用也是非常的广泛，比如 ArrayList，HashSet，与 Iterator 之间就能算是一种工厂方法。

简单工厂模式（Simple Factory）是工厂方法模式的一种，工厂方法模式的特点总结一下：

* 简单工厂模式从某种意义上来说不算是真正的设计模式，但仍不失为一个简单的方法，可以将客户程序从具体类中解耦；
* 工厂方法模式使用继承，把对象的创建委托给子类，子类实现工厂方法来创建对象，也就是说允许将实例化延迟到子类进行；
* 工厂方法模式是一个非常典型的“针对抽象编程，而不是具体类编程”例子。

## UML类图



上图为工厂方法模式的uml类图，几个角色的分工也很明确，主要分为四大模块：

* 一是抽象工厂接口，其为工厂方法模式的核心，它定义了一个工厂类所具备的基本行为；
* 二是具体工厂，其实现了具体的业务逻辑；
* 三是抽象产品接口，它定义了所有产品的公共行为；
* 四是具体产品，为实现抽象产品的某个具体产品的对象。

简单工厂模式和工厂方法模式的区别就在于简单工厂模式将抽象工厂接口这个角色给精简掉了，是工厂方法模式的一个弱化版本。

　从这种设计的角度来思考，工厂方法模式是完全符合设计原则的，它将对象的创建封装起来，以便于得到更松耦合，更有弹性的设计，而且工厂方法模式依赖于抽象的接口，将实例化的任务交给子类去完成，有非常好的可扩充性。

## 示例与源码

　我们以一个简单的玩具工厂为例，工厂中生产小孩的玩具，女生的玩具和男生的玩具，先写一个 IToy 的抽象产品接口用来定义玩具的基本行为模式，然后实现该接口生成几个玩具的具体产品类 ChildrenToy和MenToy 类：

### 产品

**IToy.class**

**public** **interface** IToy {

**String** **getName**();

**float** **price**();

**void** **play**();

}

**MenToy.class**

**public** **class** **MenToy** **implements** IToy{

@Override

**public** **String** **getName**() {

**return** "PS4";

}

@Override

**public** **float** **price**() {

**return** 2300;

}

@Override

**public** **void** **play**() {

**Log**.*e*("play", "a man is playing GTA5 on ps4");

}

}

**ChildrenToy.class**

**public** **class** **ChildrenToy** **implements** IToy{

@Override

**public** **String** **getName**() {

**return** "toy car";

}

@Override

**public** **float** **price**() {

**return** 10.5f;

}

@Override

**public** **void** **play**() {

**Log**.*e*("play", "a child is playing a toy car");

}

}

**完成产品的两个角色之后，接下来要定义工厂类的两个角色，根据工厂方法模式和简单工厂模式的不同，可以有两种不同的写法：**

### 工厂

工厂方法模式需要先写出一个工厂类的抽象接口来定义行为，这个时候根据实际情况我们可以分为两种实现方式，第一种写法会有多个 ConcreteFactory 的角色；第二种写法只会有一个 ConcreteFactory 的角色，根据传入的参数不同而返回不同的产品对象：

#### 工厂方法

##### multi ConcreateFactory

**IToyCreator.class**

**public** **interface** IToyCreator {

IToy **createToy**();

}

**MenToyCreator.class**

**public** **class** **MenToyCreator** **implements** IToyCreator {

**private** **static** **final** **String** **TAG** = "MenToyCreator";

@Override

**public** IToy **createToy**() {

IToy toy = **new** MenToy();

**Log**.*e*(**TAG**, "buy a/an " + toy.getName()+" for " + toy.price() + " yuan, and then ---");

**return** toy;

}

}

**ChildrenToyCreator.class**

**public** **class** **ChildrenToyCreator** **implements** IToyCreator {

**private** **static** **final** **String** **TAG** = "ChildrenToyCreator";

@Override

**public** IToy **createToy**() {

IToy toy = **new** ChildrenToy();

**Log**.*e*(**TAG**, "buy a/an " + toy.getName()+" for " + toy.price() + " yuan, and then ---");

**return** toy;

}

}

**最后直接可以根据需要创建不同的 IToy 对象了，测试代码如下：**

**private** **void** **doIt**(IToyCreator toyCreator) {

IToy toy = toyCreator.createToy();

toy.play();

**Log**.*e*(**TAG**, "buy a/an " + toy.getName()+" for " + toy.price() + " yuan, and then ---");

}

@Override

**public** **void** **onClick**(**View** v) {

IToyCreator toyCreator;

**switch** (v.getId()) {

**case** **R**.**id**.**btn\_child**:

toyCreator = **new** ChildrenToyCreator();

doIt(toyCreator);

**break**;

**case** **R**.**id**.**btn\_men**:

toyCreator = **new** MenToyCreator();

doIt(toyCreator);

**break**;

}

}

##### single ConcreteFactory

**IToyCreator.class**

**public** **interface** IToyCreator {

<T **extends** IToy> IToy **createToy**(**Class**<T> clazz);

}

**ConcreteToyCreator.class**

**public** **class** **ConcreteToyCreator** **implements** IToyCreator{

**private** **static** **final** **String** **TAG** = "ConcreteToyCreator";

@Override

**public** <T **extends** IToy> IToy **createToy**(**Class**<T> clazz) {

**if** (clazz == **null**){

**throw** **new** IllegalArgumentException("argument must not be null");

}

**try** {

IToy toy = clazz.newInstance();

**Log**.*e*(**TAG**, "buy a/an " + toy.getName()+" for " + toy.price() + " yuan, and then ---");

toy.play();

**return** toy;

} **catch** (**Exception** e) {

**throw** **new** UnknownError(e.getMessage());

}

}

}

这种写法直接传入一个 Class 对象，接着利用反射的方式进行对象的创建，可以说从某种意义上精简了很多的工厂实现类，不用一个具体产品类就对应需要一个具体工厂类了

**下面为测试代码：**

@Override

**public** **void** **onClick**(**View** v) {

IToyCreator2 toyCreator = **new** ConcreteToyCreator();

IToy toy;

**switch** (v.getId()) {

**case** **R**.**id**.**btn\_child**:

toy = toyCreator.createToy(**ChildrenToy**.**class**);

toy.play();

**Log**.*e*("cgy", "buy a/an " + toy.getName()+" for " + toy.price() + " yuan, and then ---");

**break**;

**case** **R**.**id**.**btn\_men**:

toy = toyCreator.createToy(**MenToy**.**class**);

toy.play();

**Log**.*e*("cgy", "buy a/an " + toy.getName()+" for " + toy.price() + " yuan, and then ---");

**break**;

}

}

##### 总结对比

单个工厂实现类的方法对比前面的多个工厂实现类的方法来说更加的简洁和动态，而且对于以后新增的产品类来说也能够不用修改原来的代码，符合开闭原则，但是这种写法在某些情况下是不适用的，比如不同的 IToy 对象设置了不同的构造函数，参数都不一样，用反射来实现就不适用了，这个时候就只能为每一个具体产品类都定义一个对应的具体工厂类了。

#### 简单工厂

同样是上面的代码，具体工厂实现类只有一个的时候，我们还是为工厂提供了一个抽象类，那么，如果将 IToyCreator 这个角色精简掉，只留下 ConcreteToyCreator 的这个角色，将其中的产品生成方法设置为静态应该也是没问题的：

**public** **class** **ToyCreator**{

**private** **static** **final** **String** **TAG** = "ToyCreator";

**public** **static** <T **extends** IToy> IToy **createToy**(**Class**<T> clazz) {

**if** (clazz == **null**){

**throw** **new** IllegalArgumentException("argument must not be null");

}

**try** {

IToy toy = clazz.newInstance();

**Log**.*e*(**TAG**, "buy a/an " + toy.getName()+" for " + toy.price() + " yuan, and then ---");

toy.play();

**return** toy;

} **catch** (**Exception** e) {

**throw** **new** UnknownError(e.getMessage());

}

}

}

像这样的方式就称为简单工厂模式，上面也说过，是工厂方法模式的一个弱化版本，缺点就是失去了被子类继承的特性，所有的压力都集中在工厂类中，不利于维护。

## 总结

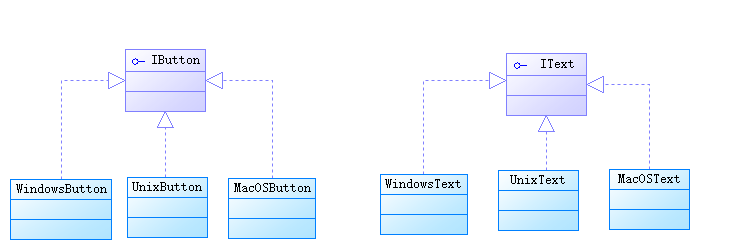
总的来说，工厂方法模式是一个很好的设计模式，它遵循了一个“尽可能让事情保持抽象”的原则，松耦合的设计原则也能够很好的符合开闭原则，将类的实例化推迟到子类，同时也摈弃了简单工厂模式的缺点。

但是同时工厂方法模式也有一些缺点，每次我们为工厂方法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，当产品种类非常多时，会出现大量的与之对应的工厂对象，这必然会导致类结构的复杂化，所以对于简单的情况下，使用工厂方法模式就需要考虑是不是有些“重”了。

# 抽象工厂（Abstract Factory）

抽象工厂模式和工厂方法模式稍有区别。工厂方法模式中工厂类生产出来的产品都是具体的，也就是说每个工厂都会生产某一种具体的产品，但是如果工厂类中所生产出来的产品是多种多样的，工厂方法模式也就不再适用了，就要使用抽象工厂模式了。

抽象工厂模式的起源或者最早的应用，是对不同操作系统的图形化解决方案，比如在不同操作系统中的按钮和文字框的不同处理，展示效果也不一样，对于每一个操作系统，其本身构成一个工厂类，而按钮与文字框控件构成一个产品类，两种产品类两种变化，各自有自己的特性，比如 Windows，Unix 和 Mac OS 下的 Button 和 Text 等。所以据此，我们可以初步构建框架：



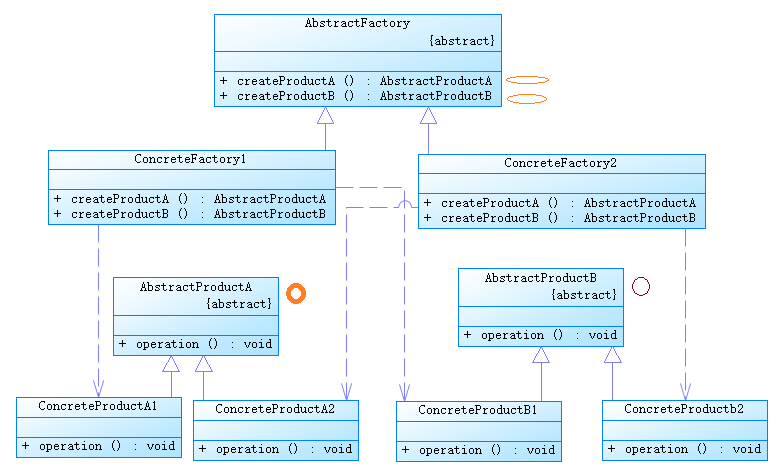
然后对于 Windows 系统来说需要生成的是 WindowsButton 和 WindowsText 产品类对象，其他两个系统一样也需要对应的对象。为了达到“为创建一组相关或者是相互依赖的对象提供一个接口，而不需要指定它们的具体类”的松散耦合原则，这时使用抽象工厂模式就非常契合。

## 特点

抽象工厂模式提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指明具体类。

和工厂方法模式一样，抽象工厂模式依然符合“针对抽象编程，不针对具体类编程”的原则，将客户端和具体类解耦，增加扩展性，契合设计模式中的依赖倒置原则和里氏替换原则。

## UML类图



**AbstractFactory：**

抽象工厂角色（对应 IFactory 接口），它声明了一组用于创建不同产品的方法，每一个方法对应一种产品，如图中的 IFactory 接口就有 createButton 和 createText 方法用于创建 IButton 对象和 IText 对象。

**ConcreteFactory：**

具体工厂角色（对应 WindowFactory，UnixFactory 和 MacOSFactory 类），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，这些产品构成一个产品种类，每一个产品都位于某个产品等级结构中。

**AbstractProduct：**

抽象产品角色（对应 IButton 和 IText 接口），他定义了几种产品的基本行为。

**ConcreteProduct：**

具体产品角色（对应WindowsButton 和 WindowsText 等 6 个实现类），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。

## 示例与源码

我们直接根据上面的 uml 类图来构建我们的图形系统，Button 和 Text 的角色是产品类，他们对应都有 3 个实现的产品子类；Windows ， Unix 和 MacOS 这几个系统应该为具体的工厂类：

### 产品相关类

产品类主要是 IButton 接口和其实现子类，IText 接口和其实现子类。

#### IButton 接口和其子类：

**IButton.class**

**public** **interface** IButton {

**void** **show**();

}

**WindowsButton.class**

**public** **class** **WindowsButton** **implements** IButton {

@Override

**public** **void** **show**() {

**Log**.*e*("show", "this is a Windows button");

}

}

**UnixButton.class**

**public** **class** **UnixButton** **implements** IButton{

@Override

**public** **void** **show**() {

**Log**.*e*("show", "this is a Unix button");

}

}

#### IText 接口和其实现子类：

**IText.class**

**public** **interface** IText {

**void** **show**();

}

**WindowsText.class**

**public** **class** **WindowsText** **implements** IText{

@Override

**public** **void** **show**() {

**Log**.*e*("show", "this is a Windows text");

}

}

**UnixText.class**

**public** **class** **UnixText** **implements** IText{

@Override

**public** **void** **show**() {

**Log**.*e*("show", "this is a Unix text");

}

}

### 工厂相关类

工厂类的作用主要是用来创建对应的两个产品种类的对象：  **IFactory.class**

**public** **interface** IFactory {

//生成对应按钮

IButton **createButton**();

// 生成对应文字

IText **createText**();

}

**WindowsFactory.class**

**public** **class** **WindowsFactory** **implements** IFactory {

@Override

**public** IButton **createButton**() {

**return** **new** WindowsButton();

}

@Override

**public** IText **createText**() {

**return** **new** WindowsText();

}

}

**UnixFactory.class**

**public** **class** **UnixFactory** **implements** IFactory{

@Override

**public** IButton **createButton**() {

**return** **new** UnixButton();

}

@Override

**public** IText **createText**() {

**return** **new** UnixText();

}

}

### 测试

@Override

**public** **void** **onClick**(**View** v) {

**switch** (v.getId()) {

**case** **R**.**id**.**tv\_win**:

**WindowsFactory** wfactory = **new** WindowsFactory();

wfactory.createButton().show();

wfactory.createText().show();

**break**;

**case** **R**.**id**.**tv\_unix**:

**UnixFactory** ufactory = **new** UnixFactory();

ufactory.createButton().show();

ufactory.createText().show();

**break**;

**default**:

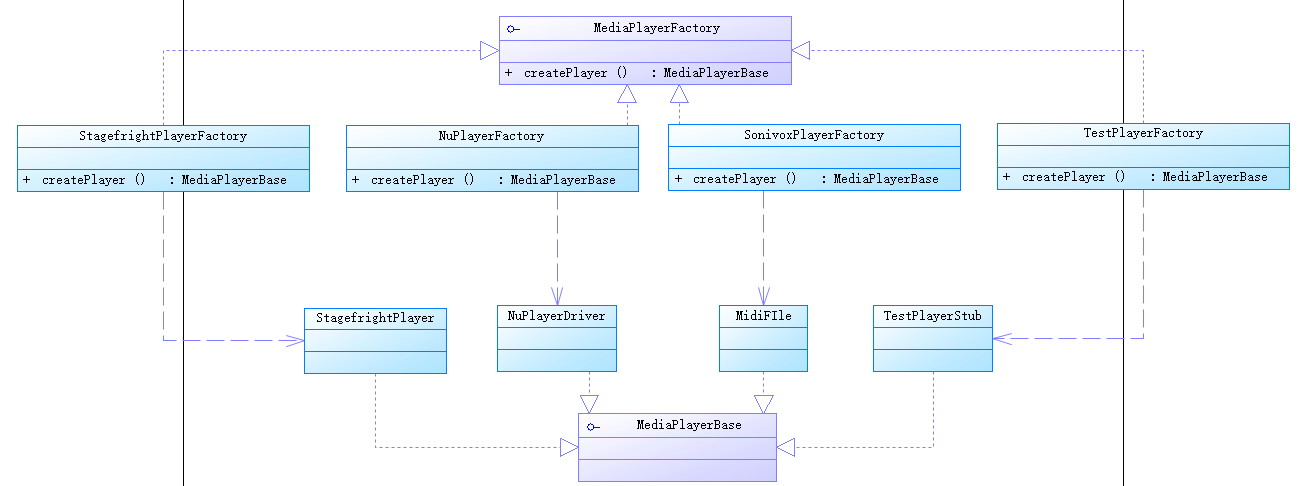
**break**;

}

}

## MediaPlayer使用抽象工厂

抽象工厂模式在 Android 源码中的实现相对来说是比较少的，其中一个比较确切的例子是 Android 底层对 MediaPlayer 的创建，具体类图如下所示：



四种 MediaPlayerFactory 分别会生成不同的 MediaPlayer 基类：StagefrightPlayer、NuPlayerDriver、MidFile 和 TestPlayerStub ，四者均继承于MediaPlayerBase 。

## 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，它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，使抽象工厂模式在切换产品类时更加灵活、容易。

## 缺点

第一个也是最明显的就是类文件的大大增多，第二个是如果要扩展新的产品类，就需要去修改抽象工厂类的最下层接口，这就会导致所有的具体工厂类均会被修改。

# 抽象工厂VS工厂方法

　　其实对比一下两种模式的 uml 图，就可以发现其实工厂方法模式是潜伏在抽象工厂模式中的，抽象工厂中的每个方法都可以单独抽出来作为一个工厂方法。抽象工厂的任务是定义一个负责创建一组产品的接口，这个接口内的每个方法都负责创建一个具体产品，同时我们利用实现抽象工厂的子类来提供这些具体的做法，所以在抽象工厂中利用工厂方法来实现生产方法是相当自然的做法。这么一对比，可以知道抽象工厂模式比工厂模式的一个优点就是在它可以将一群相关的产品集合起来。

对比一下，使用场景也就清楚了：当需要创建产品家族和想让制造的相关产品集合起来时，使用抽象工厂模式；当仅仅想把客户代码从需要实例化的具体类中解耦，或者如果目前还不明确将来需要实例化哪些具体类时，可以使用工厂方法模式。

# 建造者模式（Builder）

建造者模式又被称为生成器模式，是创造性模式之一，与工厂方法模式和抽象工厂模式不同，后两者的目的是为了实现多态性，而 Builder 模式的目的则是为了将对象的构建与展示分离。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。一个复杂的对象有大量的组成部分，比如汽车它有车轮、方向盘、发动机、以及各种各样的小零件，要将这些部件装配成一辆汽车，这个装配过程无疑会复杂，对于这种情况，为了实现在构建过程中对外部隐藏具体细节，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，同时也能够将两者之间的耦合降到最低。

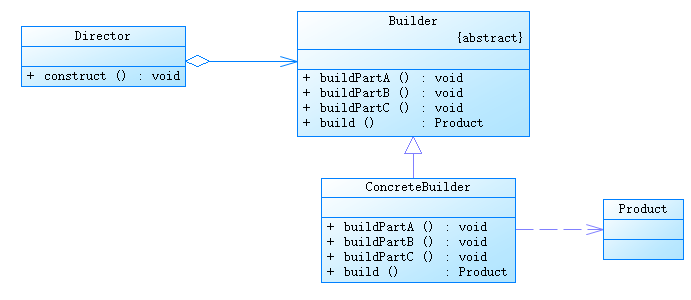
## 特点

　　将一个复杂对象的构建和它的表示分离，使得同样的构建过程可以创建不同的表示。Builder 模式适用的使用场景：

* 相同的方法，不同的执行顺序，产生不同的事件结果；
* 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；
* 产品类非常复杂，或者产品类中的调用顺序不同产生不同的作用，这个时候使用建造者模式非常适合；
* 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。

　　在实际开发过程中，Builder 模式可以使用工厂模式或者任一其他创建型模式去生成自己的指定组件，Builder 模式也是实现 fluent interface 一种非常好的方法。

## UML类图



Product 产品模块

产品的相关类；

Builder 接口或抽象类

规范产品的组件，一般是由子类实现具体的组件过程，需要注意的是这个角色在实际使用过程中可以省略，最典型的就是像 AlertDialog.Builder 一样，省略 Builder 虚拟类，将 ConcreteBuilder 写成一个静态内部类；

ConcreateBuilder 类

具体的 Builder 类；

Director 类

统一组装过程，同样值得注意的是，在现实开发过程中，Director 角色也经常会被省略，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，也就是上面提到的 fluent interface ，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，最典型的仍然是 AlertDialog.Builder 类，使用这种方式不仅去除了 Director 角色，使得整个结构更加简单，也能对 Product 对象的组件过程有着更精细的控制。

## Builder 模式的通用代码：

**Product.class**

**public** **class** **Product** {

**public** **int** partB;

**public** **int** partA;

**public** **int** **getPartA**() {

**return** partA;

}

**public** **void** **setPartA**(**int** partA) {

**this**.partA = partA;

}

**public** **int** **getPartB**() {

**return** partB;

}

**public** **void** **setPartB**(**int** partB) {

**this**.partB = partB;

}

@Override

**public** **String** **toString**() {

**return** "partA : " + partA + " partB : " + partB;

}

}

产品类在此声明了两个 setter 方法，然后是 Builder 相关类：

**Builder.class**

**public** **abstract** **class** **Builder** {

**public** **abstract** **void** **buildPartA**(**int** partA);

**public** **abstract** **void** **buildPartB**(**int** partB);

**public** **abstract** **Product** **build**();

}

**ConcreteBuilder.class**

**public** **class** **ConcreteBuilder** **extends** **Builder**{

**private** **Product** product = **new** Product();

@Override

**public** **void** **buildPartA**(**int** partA) {

product.setPartA(partA);

}

@Override

**public** **void** **buildPartB**(**int** partB) {

product.setPartB(partB);

}

@Override

**public** **Product** **build**() {

**return** product;

}

}

Builder 这两个类用来封装对 Product 属性的设置，最后在 build 方法中返回设置完属性的 Product 对象

**Director.class**

**public** **class** **Director** {

**private** **Builder** builder;

**public** **Director**(**Builder** builder) {

**this**.builder = builder;

}

**public** **void** **construct**(**int** partA, **int** partB) {

builder.buildPartA(partA);

builder.buildPartB(partB);

}

}

封装了 Builder 对象

**测试程序**

**Builder** builder = **new** ConcreteBuilder();

**Director** director = **new** Director(builder);

director.construct(1, 2);

**Product** product = builder.build();

**Log**.*e*("shawn", product.toString());//partA : 1 partB : 2

代码一目了然，这里需要提到的一点是针对不同的产品可以去构建不同的 ConcreteBuilder 类，使得一个 ConcreteBuilder 类对应一个 Product 类，这点和工厂方法模式很类似

## 示例与源码

Builder 模式在实际开发中出现和使用的频率也是很高的，比如上面提到的 AlertDialog.Builder ，还比如非常有名的第三方开源框架 Universal-Image-Loader 库中的 ImageLoaderConfig ，他们都是使用的静态内部 Builder 类。

这里的 demo 也使用最简单的内部静态 Builder 类去实现，精简完之后只有 ConcreteBuilder 和 Product 角色，并且使用链式调用去实现上面提到的 fluent interface：

**Computer.class**

**public** **class** **Computer** {

**private** **String** CPU;

**private** **String** GPU;

**private** **String** memoryType;

**private** **int** memorySize;

**private** **String** storageType;

**private** **int** storageSize;

**private** **String** screenType;

**private** **float** screenSize;

**private** **String** OSType;

**public** **static** **class** **Builder** {

//默认初始值

**private** **String** CPU = "inter-i3";

**private** **String** GPU = "GTX-960";

**private** **String** memoryType = "ddr3 1666MHz";

**private** **int** memorySize = 8;//8GB

**private** **String** storageType = "hdd";

**private** **int** storageSize = 1024;//1TB

**private** **String** screenType = "IPS";

**private** **float** screenSize = 23.8f;

**private** **String** OSType = "Windows 10";

**public** **Builder**() {

}

**public** **Builder** **setCPU**(**String** CPU) {

**this**.CPU = CPU;

**return** **this**;

}

**public** **Builder** **setGPU**(**String** GPU) {

**this**.GPU = GPU;

**return** **this**;

}

**public** **Builder** **setMemoryType**(**String** memoryType) {

**this**.memoryType = memoryType;

**return** **this**;

}

**public** **Builder** **setMemorySize**(**int** memorySize) {

**this**.memorySize = memorySize;

**return** **this**;

}

**public** **Builder** **setStorageType**(**String** storageType) {

**this**.storageType = storageType;

**return** **this**;

}

**public** **Builder** **setStorageSize**(**int** storageSize) {

**this**.storageSize = storageSize;

**return** **this**;

}

**public** **Builder** **setScreenType**(**String** screenType) {

**this**.screenType = screenType;

**return** **this**;

}

**public** **Builder** **setScreenSize**(**float** screenSize) {

**this**.screenSize = screenSize;

**return** **this**;

}

**public** **Builder** **setOSType**(**String** OSType) {

**this**.OSType = OSType;

**return** **this**;

}

**public** **Computer** **create**() {

**return** **new** Computer(**this**);

}

}

**private** **Computer**(**Builder** builder) {

CPU = builder.CPU;

GPU = builder.GPU;

memoryType = builder.memoryType;

memorySize = builder.memorySize;

storageType = builder.storageType;

storageSize = builder.storageSize;

screenType = builder.screenType;

screenSize = builder.screenSize;

OSType = builder.OSType;

}

}

Computer 为产品类，它有一个 Builder 的静态内部类用于设置相关属性，

**测试代码：**

**Computer** computer = **new** **Computer**.Builder()

.setCPU("inter-skylake-i7")

.setGPU("GTX-Titan")

.setMemoryType("ddr4-2133MHz")

.setMemorySize(16)

.setStorageType("ssd")

.setStorageSize(512)

.setScreenType("IPS")

.setScreenSize(28)

.setOSType("Ubuntu/Window10")

.create();

这里需要提到的关键点是关于相关属性的默认值问题：

* 对于必要的属性值，无法给出其默认值的最好是通过 Builder 类的构造函数传入，比如 AlertDialog.Builder 类的 Context，这样也能防止使用时的疏忽；
* 对于非必要属性来说，最好是为其生成一个默认的属性值，这样使用者只用设置需要更改的属性即可；
* 每个 setter 函数都加上 return this 用来实现优美的 fluent interface 设计。

## 总结

　　Builder 模式在 Android 开发中也很常用，通常作为配置类的构建器将配置的构建和表示分离开来，同时也将配置从目标类中隔离开来，避免了过多的 setter 方法。Builder 模式比较常见的实现形式是通过调用链实现，这样的方式也会使得代码更加简洁和易懂，而且同时也可以避免了目标类被过多的接口“污染”。

## 优点：

* 将一个复杂对象的创建过程封装起来，使得客户端不必知道产品内部组成的细节；
* 允许对象通过多个步骤来创建，并且可以改变过程和选择需要的过程；
* 产品的实现可以被替换，因为客户端只看到一个抽象的接口；
* 创建者独立，容易扩展。

## 缺点：

* 会产生多余的 Builder 对象以及 Director 对象，消耗内存；
* 与工厂模式相比，采用 Builder 模式创建对象的客户，需要具备更多的领域知识。

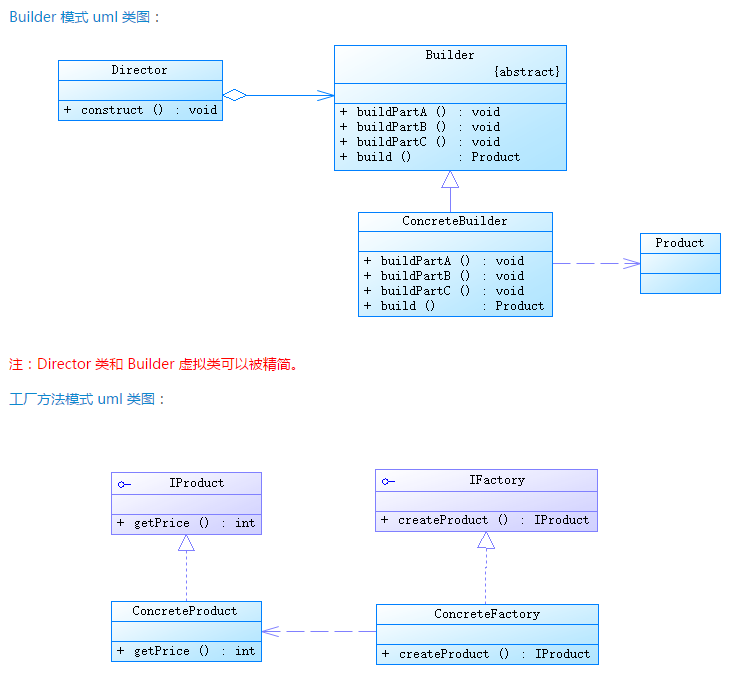
## Builder 模式 VS 工厂方法模式

Builder 模式和工厂方法模式都是属于创建型模式，他们有一些共同点：这两种设计模式的都将一个产品类对象的创建过程封装起来，让客户端从具体产品类的生成中解耦，不必了解产品类构造的细节。

### 几个不同点：

* Builder 模式允许对象的创建通过多个步骤来创建，而且可以改变这个过程，也可以选择需要改变的属性；工厂方法模式不一样，它只有一个步骤，也就无法改变这个过程，更加无法选择性改变属性了；
* Builder 模式的目的是将复杂对象的构建和它的表示分离；而工厂方法模式则是定义一个创建对象的接口，由子类决定要实例化的类是哪一个，将实例化推迟到子类；
* 最明显的当然还是代码的差异，Builder 模式中客户端可以调用 set 方法，而工厂方法模式只能调用工厂类提供的相关方法。

### 其次是 uml 类图的差异：



uml 类图的相似性还是很高的，所以通常我们会根据实际表现和用途来区别 Buidler 模式和工厂方法模式（这点和装饰者模式与保护代理模式的区别类似，要从实际表现与使用的目的区别）。

# ----结构型----享元模式（Flyweight）

## 享元模式（Flyweight Pattern）

Flyweight 代表轻量级的意思，享元模式是对象池的一种实现。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，缓存可共享的对象，来达到对象共享和避免创建过多对象的效果，这样一来就可以提升性能，避免内存移除和频繁 GC 等。

享元模式的一个经典使用案例是文本系统中图形显示所用的数据结构，一个文本系统能够显示的字符种类就是那么几十上百个，那么就定义这么些基础字符对象，存储每个字符的显示外形和其他的格式化数据等，而不用每次都去新建对象，这样就可以避免创建成千上万的重复对象，大大提高对象的重用率。

## 特点

　　使用共享对象可有效地支持大量细粒度的对象。 共享模式支持大量细粒度对象的复用，所以享元模式要求能够共享的对象必须是细粒度对象。在了解享元模式之前我们先要了解两个概念：内部状态、外部状态：

内部状态：在享元对象内部不随外界环境改变而改变的共享部分；

外部状态：随着环境的改变而改变，不能够共享的状态就是外部状态。

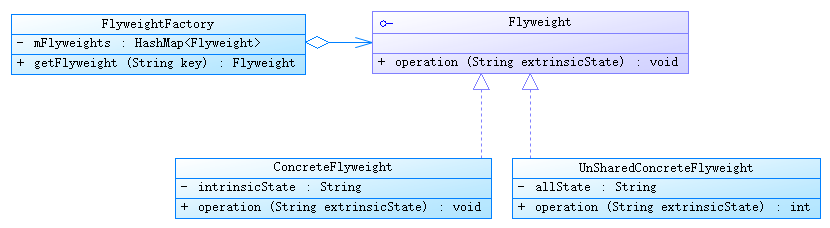
由于享元模式区分了内部状态和外部状态，所以我们可以通过设置不同的外部状态使得相同的对象可以具备一些不同的特性，而内部状态设置为相同部分。在我们的程序设计过程中，我们可能会需要大量的细粒度对象，如果这些对象除了几个参数不同外其他部分都相同，这个时候我们就可以利用享元模式来大大减少应用程序当中的对象。如何利用享元模式呢？这里我们只需要将他们少部分的不同的状态当做参数移动到类实例的外部去，然后在方法调用的时候将他们传递过来就可以了。这里也就说明了一点：内部状态存储于享元对象内部，而外部状态则应该由客户端来考虑。

## 享元模式与对象池模式对比

　　看了上面的特征会让我们想到对象池模式，确实，对象池模式和享元模式有很多的相同点，但是他们有一个很重要的不同点：享元模式通常情况下获取的是不可变的实例，而从对象池模式中获取的对象通常情况下是可变的。所以使用享元模式用来避免创建多个拥有同样状态的对象，只创建一个并且在应用的不同地方都使用这一个实例；而对象池中的资源从使用的角度上看具有不同的状态并且每个都需要单独控制，但是又不想花费一定的资源区频繁的创建和销毁这些资源对象，毕竟他们都有相同的初始化过程。

简而言之，享元模式更加倾向于状态的不可变性，而对象池模式则是状态的可变性。

## UML类图



Flyweight：

享元对象抽象基类或者接口；

ConcreteFlyweight：

具体的享元对象；

UnsharedConcreteFlyweight：

非共享具体享元类，指出那些不需要共享的Flyweight子类；

FlyweightFactory：

享元工厂，负责管理享元对象池和创建享元对象。

## 享元模式的基础代码

### Flyweight.class

**public** **interface** Flyweight {

**void** **operation**();

}

### ConcreteFlyweight.class

**public** **class** **ConcreteFlyweight** **implements** Flyweight{

**private** **String** intrinsicState;

**public** **ConcreteFlyweight**(**String** state) {

intrinsicState = state;

}

@Override

**public** **void** **operation**() {

**Log**.*e*("Shawn", "ConcreteFlyweight----" + intrinsicState);

}

}

### FlyweightFactory.class

**public** **class** **FlyweightFactory** {

**private** **HashMap**<**String**, Flyweight> mFlyweights = **new** HashMap<>();

**public** Flyweight **getFlyweight**(**String** key) {

Flyweight flyweight = mFlyweights.get(key);

**if** (flyweight == **null**) {

flyweight = **new** ConcreteFlyweight(key);

mFlyweights.put(key, flyweight);

}

**return** flyweight;

}

}

### 测试代码

**private** **void** **Yes**() {

**FlyweightFactory** factory = **new** FlyweightFactory();

Flyweight flyweight1 = factory.getFlyweight("a");

Flyweight flyweight2 = factory.getFlyweight("b");

Flyweight flyweight3 = factory.getFlyweight("a");

**Log**.*e*("Shawn", "flyweight1==flyweight2 : " + (flyweight1 == flyweight2));//false

**Log**.*e*("Shawn", "flyweight1==flyweight3 : " + (flyweight1 == flyweight3));//true

}

## Java 中的享元模式

在 Java 中，最经典使用享元模式的案例就应该是 String 了，String 存在常量池中，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，则直接使用该缓存，而不会重复创建

**String** str1 = **new** String("abc");

**String** str2 = **new** String("abc");

**String** str3 = "abc";

**String** str4 = "ab" + "c";

str1 == str2; //false

str3 == str4; //true

str1 和 str2 是两个不同的对象，这个应该显而易见，而 str3 和 str4 由于都是使用的 String 享元池，所以他们两个是同一个对象。

## 示例与源码

我们这以一个图形系统为例，用来画不同颜色的圆形：

### Shape.class用来定义一个图形的基本行为：

**public** **interface** Shape {

**void** **draw**();

}

### Circle.class Shape 的实现子类，用来画圆形：

**public** **class** **Circle** **implements** Shape{

**String** color;

**public** **Circle**(**String** color) {

**this**.color = color;

}

@Override

**public** **void** **draw**() {

**Log**.*e*("Shawn", "画了一个" + color +"的圆形");

}

}

### ShapeFactory.class 图形享元工厂类：

**public** **class** **ShapeFactory** {

**private** **HashMap**<**String**, Shape> shapes = **new** HashMap<>();

**public** Shape **getShape**(**String** color) {

Shape shape = shapes.get(color);

**if** (shape == **null**) {

shape = **new** Circle(color);

shapes.put(color, shape);

}

**return** shape;

}

**public** **int** **getSize**() {

**return** shapes.size();

}

}

### 测试代码

**ShapeFactory** factory = **new** ShapeFactory();

Shape shape1 = factory.getShape("红色");

shape1.draw();

Shape shape2 = factory.getShape("灰色");

shape2.draw();

Shape shape3 = factory.getShape("绿色");

shape3.draw();

Shape shape4 = factory.getShape("红色");

shape4.draw();

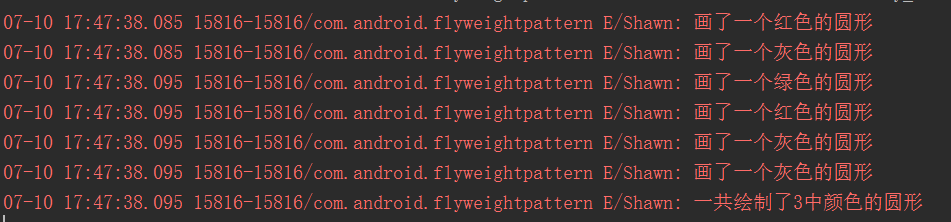
Shape shape5 = factory.getShape("灰色");

shape5.draw();

Shape shape6 = factory.getShape("灰色");

shape6.draw();

**Log**.*e*("Shawn", "一共绘制了"+factory.getSize()+"中颜色的圆形");



## 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。它可以大大减少应用程序创建对象的数量和频率，降低程序内存的占用，增强程序的性能，但它同时也增加了系统的复杂性，需要分离出外部状态和内部状态，内部状态为不变的共享部分，存储于享元对象内部；而外部状态具有固化特性，应当由客户端来负责，不应该随着内部状态改变而改变，否则会导致系统的逻辑混乱。

### 享元模式优点：

能够极大的减少系统中对象的个数；

享元模式由于使用了外部状态，外部状态相对独立，不会影响到内部状态，所以享元模式使得享元对象能够在不同的环境中被共享。

### 享元模式缺点：

由于享元模式需要区分外部状态和内部状态，使得应用程序在某种程度上来说更加复杂化了；

为了使对象可以共享，享元模式需要将享元对象的状态外部化，而读取外部状态使得运行时间变长。

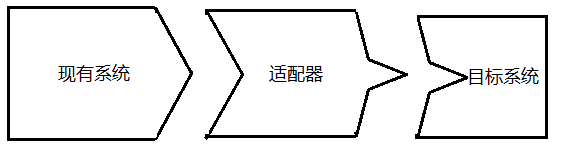
### 讨论

我在查阅相关书籍和网络资料的过程中，看到有些文章会把 Android 中的 MessagePool 定义为享元模式，但是对比了对象池模式和享元模式之后，我更倾向于认为它是对象池模式，因为从上面介绍的对比来看，MessagePool 中对象池有初始化的 size，每次从 MessagePool 中去 obtain Message 对象的时候，获取的都是一个初始对象，其中的状态都需要去根据需求变化，而享元模式则更倾向于重用具有相同状态的对象，这个对象着重于在应用的每个使用地方它的状态都具有相同性，从这个原则来看就已经排除是享元模式了，不过还是个人的看法，有没有大神指导一下，不胜感激~~~~

# 适配器模式（Adapter）

## 介绍

适配器模式在开发中使用的频率也是很高的，像 ListView 和 RecyclerView 的 Adapter 等都是使用的适配器模式。在我们的实际生活中也有很多类似于适配器的例子，比如香港的插座和大陆的插座就是两种格式的，为了能够成功适配，一般会在中间加上一个电源适配器，形如：



说到底，适配器模式是将原来不兼容的两个类融合在一起，它有点类似于粘合剂，将不同的东西通过一种转换使得它们能够协作起来。碰到要在两个完全没有关系的类之间进行交互，第一个解决方案是修改各自类的接口，但是如果无法修改源代码或者其他原因导致无法更改接口，此时怎么办？这种情况我们往往会使用一个 Adapter ，在这两个接口之间创建一个粘合剂接口，将原本无法协作的类进行兼容，而且不用修改原来两个模块的代码，符合开闭原则。

## 特点

适配器模式把一个类的接口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。所以，这个模式可以通过创建适配器进行接口转换，让不兼容的接口兼容，这可以让客户实现解耦。如果在一段时间之后，我们想要改变接口，适配器可以将改变的部分封装起来，客户就不必为了应对不同的接口而每次跟着修改。

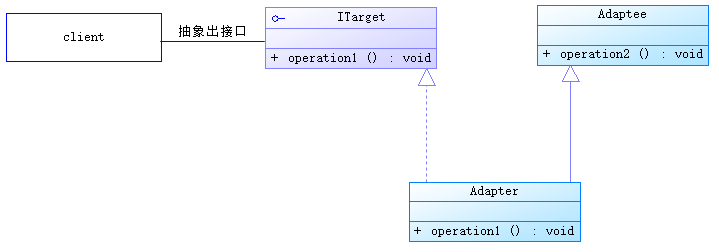
## 适配器模式的使用场景可以有以下几种：

* 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；
* 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联系的一些类，包括一些可能在将来引进的类一起工作；
* 需要一个统一的输出接口，而输入端的类型不可预知。

## UML类图

**适配器模式在实际使用过程中有“两种”方式：对象适配器和类适配器。**

### 类适配器模式



类适配器是通过实现 ITarget 接口以及继承 Adaptee 类来实现接口转换，目标接口需要的是 operation1() 的操作，而 Adaptee 类只能提供一个 operation2() 的操作，因此就出现了不兼容的情况，此时通过 Adapter 实现一个 operation1() 函数将 Adaptee 的 operation2() 转换为 ITarget 需要的操作，以此实现兼容。类适配器模式有三个角色：

Target：

目标角色，也就是所期待得到的接口，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，适配器把源接口转换成目标接口，所以这一个角色必须是具体类。

### 对象配器模式

uml 类图和类适配器模式基本一样，区别就在于对象适配器模式与 Adaptee 的关系是 Dependency，而类适配器是 Generalization ，一个是依赖，一个是继承。所以 Adapter 类会持有一个 Adaptee 对象的引用，并且通过 operation1() 方法将该 Adaptee 对象与 ITarget 接口的相关操作衔接起来。

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，这种模式比类适配器模式更加灵活，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 类出现了一些奇怪的接口，用于使用成本较高。因此，对象适配器模式更加灵活和实用。

### 对比

类适配器模式使用的是继承的方式，而对象适配器模式则使用的是组合的方法。从设计模式的角度来说，对象适配器模式遵循 OO 设计原则的“多用组合，少用继承”，这是一个优点，但是类适配器模式有一个好处是它不需要重新实现整个被适配者的行为，毕竟类适配器模式使用的是继承的方式，当然这么做的坏处就是失去了使用组合的弹性。

所以在实际过程中需要根据使用情况而定，如果 Adaptee 类的行为很复杂，但是 Adapter 适配器类并不需要这些大部分的无关行为，那么使用对象适配器模式是合适的，但是如果需要重新实现大部分 Adaptee 的行为，那么就要考虑是否使用类适配器模式了。

## 示例与源码

### 类适配器模式

我们以最上面说到的香港的英式三角插座和大陆的三角插座为例，来构造类适配器模式，首先是两个插座格式类：

**IChinaOutlet.class**

public interface IChinaOutlet {

public String getChinaType();

}

**ChinaOutlet.class**

public class ChinaOutlet implements IChinaOutlet{

@Override

public String getChinaType() {

return "Chinese three - pin socket";

}

}

上面是中式插座的输出格式，然后是香港的英式插座输出格式：

**HKOutlet.class**

public class HKOutlet {

public String getHKType() {

return "British three - pin socket";

}

}

为了将香港的英式插座转换为中式插座，我们需要构造一个 Adapter 类，目的是进行插座格式的转换：

**OutletAdapter.class**

public class OutletAdapter extends HKOutlet implements IChinaOutlet{

@Override

public String getChinaType() {

String type = getHKType();

type = type.replace("Chinese", "British");

return type;

}

}

这样就实现了插座接口的转换，例子很简单，明了。当然这个例子很简单，要的就是要学会这个思想：在不修改原来类的基础上，将原来类进行扩展后使用在新的目标系统上。

### 对象适配器模式

**笔记本类（clien客户端）**

//笔记本

**public** **class** **Jotter** {

**private** VoltageAdapter adapter;

**public** **void** **inputVoltage**(){

**System**.**out**.println("笔记本得到输入电压" +adapter.transformVoltage()+"v");//调用VoltageAdapter适配器目标接口的函数

}

**public** **void** **setAdapter**(VoltageAdapter adapter){//添加适配，笔记本需要的是VoltageAdapter适配器目标接口的函数

**this**.adapter = adapter;

}

}

**供电器（adaptee,需要适配的类）**

//供电器

**public** **class** **PowerSupplyDevice** {

**private** **int** standardVoltage =220;//标准电压

//输出220v电压

**public** **int** **powerSupply**(){

**System**.**out**.println("标准电压提供220v");

**return** standardVoltage;

}

}

**电源适配器接口,（Target,客户端想要的目标接口）**

//电源适配器

**public** **interface** VoltageAdapter {

//转换电压

**public** **int** **transformVoltage**();//目标接口的函数，供客户端调用

}

**电源适配器类，（adapter,包装adaptee）**

//电源适配器

**public** **class** **MyVoltageAdapter** **implements** VoltageAdapter{

**private** **int** voltage; //电压

//标准电压设备220v

**private** **PowerSupplyDevice** powerSupplyDevice ;

**public** **MyVoltageAdapter**(**PowerSupplyDevice** powerSupplyDevice) {

**this**.powerSupplyDevice = powerSupplyDevice;

}

//转换电压，重写接口的方法，用上需要被适配的类，包装被适配的类，客户端调用此方法时，就将电源和电脑关联上了

@Override

**public** **int** **transformVoltage**(){

**int** voltage = powerSupplyDevice.powerSupply();

**int** lowerVoltage = voltage/14;

**System**.**out**.println("转化中电压为="+lowerVoltage+"v");

**return** lowerVoltage;

}

}

**调用**

**public** **class** **MainActivity** **extends** **Activity** {

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

**Jotter** jt = **new** Jotter();//客户端

**PowerSupplyDevice** powerSupplyDevice = **new** PowerSupplyDevice();//adaptee被适配的类

VoltageAdapter adapter = **new** MyVoltageAdapter(powerSupplyDevice);//适配器，将adaptee和客户端关联

jt.setAdapter(adapter);//添加适配，方便用适配器目标接口里的函数

jt.inputVoltage();//这个方法里调用了 适配器目标接口里的函数，该函数被子类重写，重写方法里将adaptee和客户端关联

}

}

## 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，使之能够很好的进行合作。但是，在实际开发中， Adapter 模式也会可以根据实际情况进行适当的变更，最典型的就是 ListView 和 RecyclerView 了，这种设计方式使得整个 UI 架构变得非常灵活，能够拥抱变化。所以在实际使用的时候，遵循上面说过的三种场景：

* 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；
* 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联系的一些类，包括一些可能在将来引进的类一起工作；
* 需要一个统一的输出接口，而输入端的类型不可预知。

## 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；

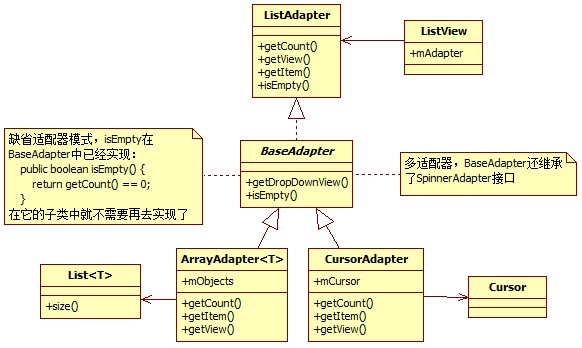
* 更好的扩展性

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。

总结一下就是对扩展开放和对修改关闭的开闭原则吧。

当然适配器模式也有一些缺点，如果在一个系统中过多的使用适配器模式，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 B 类的实现，这样就增加了维护性，过多的使用就显得很没有必要了，不如直接对系统进行重构。

## ListAdapter适配器模式的代码实现



为了简明直接，我省略了相关的其他适配器，只以此两个适配器为例。ListViews做为client，他所需要的目标接口(target interface)就是ListAdapter，包含getCount(),getItem(),getView()等几个基本的方法，为了兼容List<T>,Cursor等数据类型作为数据源，我们专门定义两个适配器来适配他们：ArrayAdapter和CursorAdapter。这两个适配器，说白了，就是针对目标接口对数据源进行兼容修饰。这就是适配器模式。其中BaseAdapter实现了如isEmpty()方法,使子类在继承BaseAdapter后不需要再实现此方法，这就是缺省适配器，这也是缺省适配器的一个最明显的好处。 我们以最简单的若干个方法举例如下

**ListAdapter接口如下（为了简单，我省略了继承自Adapter接口）：**

**public** **interface** ListAdapter {

**public** **int** **getCount**();

**Object** **getItem**(**int** position);

**long** **getItemId**(**int** position);

**View** **getView**(**int** position, **View** convertView, **ViewGroup** parent);

**boolean** **isEmpty**();

}

**抽象类BaseAdapter,我省略其他代码，只列出两个方法，以作示意:**

**public** **abstract** **class** **BaseAdapter** **implements** **ListAdapter**, **SpinnerAdapter** {

// ... ...

**public** **View** **getDropDownView**(**int** position, **View** convertView, **ViewGroup** parent) {

**return** getView(position, convertView, parent);

}

**public** **boolean** **isEmpty**() {

**return** getCount() == 0;

}

}

**ArrayAdapter对List<T>进行封装成ListAdapter的实现，满足ListView的调用：**

**public** **class** **ArrayAdapter**<T> **extends** **BaseAdapter** **implements** **Filterable** {

**private** **List**<T> mObjects;

//我只列出这一个构造函数，大家懂这个意思就行

**public** **ArrayAdapter**(**Context** context, **int** textViewResourceId, T[] objects) {

init(context, textViewResourceId, 0, Arrays.asList(objects));

}

**public** **ArrayAdapter**(**Context** context, **int** resource, List<T> objects) {

init(context, resource, 0, objects);

}

**private** **void** **init**(**Context** context, **int** resource, **int** textViewResourceId, **List**<T> objects) {

mContext = context;

mInflater = (**LayoutInflater**)context.getSystemService(Context.LAYOUT\_INFLATER\_SERVICE);

mResource = mDropDownResource = resource;

mObjects = objects; //引用对象，也是表达了组合优于继承的意思

mFieldId = textViewResourceId;

}

**public** **int** **getCount**() {

**return** mObjects.size();

}

**public** T **getItem**(**int** position) {

**return** mObjects.get(position);

}

**public** **View** **getView**(**int** position, **View** convertView, **ViewGroup** parent) {

**return** createViewFromResource(position, convertView, parent, mResource);

}

// ... ...

}

　 我们就如此成功的把List<T>作为数据源以ListView想要的目标接口的样子传给了ListView，同理CursorAdapter也是一模一样的道理，就不写具体代码了。适配器本身倒是不难，但是提供了解决不兼容问题的惯用模式。

# 桥接模式（Bridge）

## 桥接模式（Bridge Pattern）

结构型设计模式之一,桥接，顾名思义，就是用来连接两个部分，使得两个部分可以互相通讯或者使用，桥接模式的作用就是为被分离了的抽象部分和实现部分搭桥。在现实生活中也有很多这样的例子，一个物品在搭配不同的配件时会产生不同的动作和结果，例如一辆赛车搭配的是硬胎或者是软胎就能够在干燥的马路上行驶，而如果要在下雨的路面行驶，就需要搭配雨胎了，这种根据行驶的路面不同，需要搭配不同的轮胎的变化的情况，我们从软件设计的角度来分析，就是一个系统由于自身的逻辑，会有两个或多个维度的变化，有时还会形成一种树状的关系，而为了应对这种变化，我们就可以使用桥接模式来进行系统的解耦。

桥接模式，作用是将一个系统的抽象部分和实现部分分离，使它们都可以独立地进行变化，对应到上面就是赛车的种类可以相对变化，轮胎的种类可以相对变化，形成一种交叉的关系，最后的结果就是一种赛车对应一种轮胎就能够成功产生一种结果和行为。

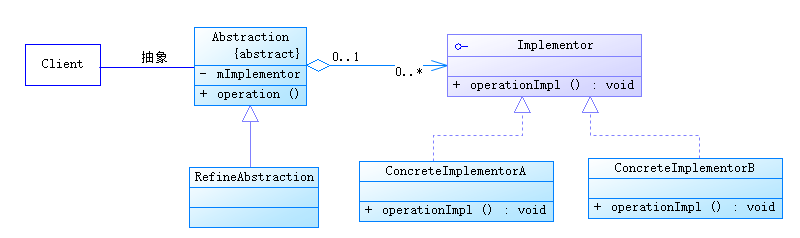
## 特点

将抽象部分与实现部分分离，使他们都可以独立地进行变化。为了达到让抽象部分和实现部分独立变化的目的，抽象部分会拥有实现部分的接口对象，有了实现部分的接口对象之后，就能够通过这个接口来调用具体实现部分的功能。桥接在程序上就体现成了抽象部分拥有实现部分的接口对象，维护桥接就是维护这个关系，也就是说，桥接模式中的桥接是一个单方向的关系，只能够抽象部分去使用实现部分的对象，而不能反过来。

桥接模式适用于以下的情形：

* 如果一个系统需要在构建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他们在抽象层建立一个关联关系；
* 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极具增加的系统；
* 一个类存在两个独立变化的维度，而这两个维度都需要进行扩展。

## UML类图



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

可以为接口或者是抽象类，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基本操作的业务方法；

ConcreteImplementorA 和 ConcreteImplementorB ：实现部分的具体实现

完善实现部分中定义的具体逻辑。

## 桥接模式的通用代码

**Implementor.class**

**public** **interface** Implementor {

**void** **operationImpl**();

}

**ConcreteImplementorA.class**

**public** **class** **ConcreteImplementorA** **implements** Implementor{

@Override

**public** **void** **operationImpl**() {

//具体实现

}

}

**Abstraction.class**

**public** **abstract** **class** **Abstraction** {

**private** Implementor implementor;

**public** **Abstraction**(Implementor implementor) {

**this**.implementor = implementor;

}

**public** **void** **operation**() {

implementor.operationImpl();

}

}

**RefinedAbstraction.class**

**public** **class** **RefinedAbstraction** **extends** **Abstraction**{

**public** **RefinedAbstraction**(Implementor implementor) {

**super**(implementor);

}

**public** **void** **refinedOperation**() {

//对 Abstraction 中的 operation 方法进行扩展

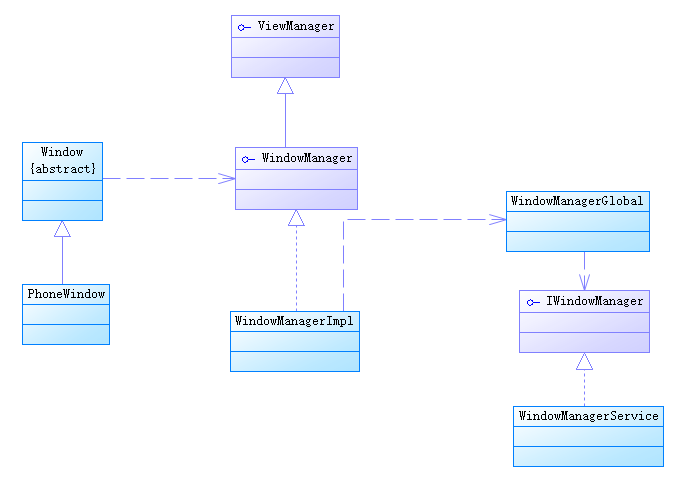
}

}

当然上面的 uml 类图和通用代码只是最常用的实现方式而已，在实际使用中可能会有其他的情况，比如 Implementor 只有一个类的情况，虽然这时候可以不去创建 Implementor 接口，精简类的层次，但是我建议还是需要抽象出实现部分的接口，在我们下面要讲到的 Android 源码桥接模式分析中，Android 系统就算只有一个实现部分具体类，也抽象出了一个实现接口，可以学习一下。总而言之，保持分离状态，这样的话，他们才不会互相影响，才可以分别扩展。

## Android 源码桥接模式分析

桥接模式在 Android 源码中的使用频率也是很高的，这里就分析一下最典型的 Window 与 WindowManager 之间的桥接模式，先来看看他们的 uml 图：



Window 类和 PhoneWindow 类为抽象部分，PhoneWindow 为 Window 类的唯一实现子类，在 Window 类中，持有了一个 WindowManager 类的引用，并且在 setWindowManager 方法中将其赋值为了 WindowManagerImpl 对象：

public void setWindowManager(WindowManager wm, IBinder appToken, String appName,

boolean hardwareAccelerated) {

...

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

}

对应着的 WindowManager 接口和 WindowManagerImpl 类就组成了实现部分，所以说这四个类使用的就是典型的桥接模式，Window 中的 addView，removeView 等操作都桥接给了 WindowManagerImpl 去处理。但是实际上 WindowManagerImpl 中并没有去实现 addView 方法，在 WindowManagerImpl 类中持有了一个 WindowManagerGlobal 对象的引用，这个引用是通过单例模式获取的，而 addView 等方法就直接交给了 WindowManagerGlobal 类去处理：

public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params) {

applyDefaultToken(params);

mGlobal.addView(view, params, mDisplay, mParentWindow);

}

继续分析 WindowManagerGlobal 类的 addView 函数，在该函数中，最后调用到了 ViewRootImpl 类中的 setView 成员方法，ViewRootImpl 类中的 setView 方法最后会调用到 scheduleTraversals 方法，scheduleTraversals 方法：

void scheduleTraversals() {

if (!mTraversalScheduled) {

...

notifyRendererOfFramePending();

pokeDrawLockIfNeeded();

}

}

void pokeDrawLockIfNeeded() {

final int displayState = mAttachInfo.mDisplayState;

if (mView != null && mAdded && mTraversalScheduled

&& (displayState == Display.STATE\_DOZE

|| displayState == Display.STATE\_DOZE\_SUSPEND)) {

try {

mWindowSession.pokeDrawLock(mWindow);

} catch (RemoteException ex) {

// System server died, oh well.

}

}

}

看到了 mWindowSession 对象，是不是很熟悉，mWindowSession 对象的创建：

mWindowSession = WindowManagerGlobal.getWindowSession();

嗯，又回到了 WindowManagerGlobal 类，去看看它的 getWindowSession 方法：

public static IWindowSession getWindowSession() {

synchronized (WindowManagerGlobal.class) {

if (sWindowSession == null) {

try {

InputMethodManager imm = InputMethodManager.getInstance();

IWindowManager windowManager = getWindowManagerService();

sWindowSession = windowManager.openSession(

new IWindowSessionCallback.Stub() {

@Override

public void onAnimatorScaleChanged(float scale) {

ValueAnimator.setDurationScale(scale);

}

},

imm.getClient(), imm.getInputContext());

} catch (RemoteException e) {

Log.e(TAG, "Failed to open window session", e);

}

}

return sWindowSession;

}

}

WindowSession 是通过 IWindowManager 创建的：

public static IWindowManager getWindowManagerService() {

synchronized (WindowManagerGlobal.class) {

if (sWindowManagerService == null) {

sWindowManagerService = IWindowManager.Stub.asInterface(

ServiceManager.getService("window"));

try {

sWindowManagerService = getWindowManagerService();

ValueAnimator.setDurationScale(sWindowManagerService.getCurrentAnimatorScale());

} catch (RemoteException e) {

Log.e(TAG, "Failed to get WindowManagerService, cannot set animator scale", e);

}

}

return sWindowManagerService;

}

}

跨进程之间的通信，最终调用到的是 WindowManagerService 中，这里需要简单提到的一句是 WMS 和 AMS 一样都是由 SystemServer 启动的，是在另一个进程中的，这也是为什么需要跨进程之间的通信

## 示例与源码

以上面说到的车与轮胎的关系来作为一个例子，车为抽象部分，轮胎为实现部分，抽象部分的相关类：

**ITire.class**

**public** **interface** ITire {

**String** **run**();

}

**RainyTire.class**

**public** **class** **RainyTire** **implements** **ITire**{

@Override

**public** **String** **run**() {

**return** "run on the rainy road.";

}

}

**SandyTire.class**

**public** **class** **SandyTire** **implements** **ITire**{

@Override

**public** **String** **run**() {

**return** "run on the sandy road.";

}

}

**Car.class**

**public** **abstract** **class** **Car** {

**private** ITire tire;

**public** **Car**(ITire tire) {

**this**.tire = tire;

}

**public** ITire **getTire**() {

**return** tire;

}

**public** **abstract** **void** **run**();

}

**RacingCar.class**

**public** **class** **RacingCar** **extends** **Car**{

**public** **RacingCar**(ITire tire) {

**super**(tire);

}

@Override

**public** **void** **run**() {

**Log**.*e*("shawn", "racing car " + getTire().run());

}

}

**SedanCar.class**

**public** **class** **SedanCar** **extends** **Car**{

**public** **SedanCar**(ITire tire) {

**super**(tire);

}

@Override

**public** **void** **run**() {

**Log**.*e*("shawn", "sedan car " + getTire().run());

}

}

**测试代码**

Car car = null;

switch (v.getId()) {

case R.id.btn\_sedanCar\_with\_rainyTire:

car = new SedanCar(new RainyTire());

break;

case R.id.btn\_sedanCar\_with\_sandyTire:

car = new SedanCar(new SandyTire());

break;

case R.id.btn\_racingCar\_with\_rainyTire:

car = new RacingCar(new RainyTire());

break;

case R.id.btn\_racingCar\_with\_sandyTire:

car = new RacingCar(new SandyTire());

break;

}

car.run();

**最后的结果**

com.android.bridgepattern E/shawn: sedan car run on the rainy road.

com.android.bridgepattern E/shawn: sedan car run on the sandy road.

com.android.bridgepattern E/shawn: racing car run on the rainy road.

com.android.bridgepattern E/shawn: racing car run on the sandy road

例子很简单，一目了然，使用桥接模式的优点大家也能够看出来， Car 和 Tire 的逻辑完全分离，各自搭配，大大降低了耦合度

## 总结

桥接模式的目的是为了将抽象部分与实现部分解耦，可以将一个 N \* N 的系统进行拆分，减少类的数量。桥接模式适合使用在需要跨越多个平台的图形和窗口系统上，优点很明显：

* 将实现予以解耦，让它和抽象之间不再永久绑定，使用组合的关系，降低耦合度，符合开闭原则；
* 抽象和实现之间可以独立扩展，不会影响到彼此；
* 对于“具体抽象类”所做的改变，不会影响到客户端。

但是桥接模式的缺点也很明显，一个是增加了系统的复杂度，二个是不容易设计，对于抽象和实现的分离把握，是不是需要分离，如何分离等这些问题对于设计者来说要有一个恰到好处的分寸，理解桥接模式很容易，要设计合理不容易。

桥接有时候类似于多继承方案，但是多继承方案往往违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差，桥接模式是比多继承方案更好的解决方法，这点与装饰者模式类似。

# 装饰者模式（Decorator、Wrapper）

## 概述

装饰者模式（Decorator Pattern），装饰者模式也称为包装模式（Wrapper Pattern），结构型模式之一，其使用一种对客户端透明的方式来动态的扩展对象的功能，同时它也是继承关系的一种替代方案之一，但比继承更加灵活。在现实生活中也可以看到很多装饰者模式的例子，或者可以大胆的说装饰者模式无处不在，就拿一件东西来说，可以给它披上无数层不一样的外壳，但是这件东西还是这件东西，外壳不过是用来扩展这个东西的功能而已，这就是装饰者模式，装饰者的这个角色也许各不相同但是被装饰的对象本质是不变的。

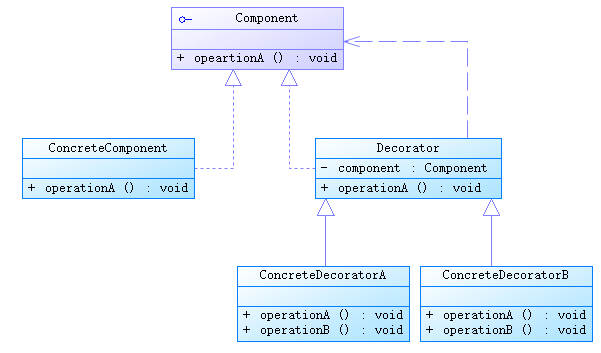
我们的目标是允许类统一扩展，在不修改现有代码的情况下，就可搭配新的行为。如能实现这样的目标，有什么好处呢？这样的设计具有弹性，可以应对改变，可以接受新的功能来应对改变的需求，也就是 OO 原则中的对扩展开放和对修改关闭的开闭原则。

## 特点

动态地给一个对象添加一些额外的职责，就增加功能来说，装饰者模式相比生成子类更加灵活，提供了有别于继承的另一种选择。

装饰者模式可以静态的，或者根据需要可以动态的在运行时为一个对象扩展功能。被装饰者和众多的装饰者都是继承自一个接口，他们有着一样的行为特性。装饰者模式是继承的另一种选择方式，继承是在编译的时候为类添加新的行为，并且这个改变会影响所有原来该类的实体，装饰者模式就不一样，它提供一种能够在运行时根据需要选择不同运行对象的功能。装饰者模式和继承这两种方式的不同之处在某些扩展功能的情况下显得尤为重要，在一些面向对象编程的语言中，类无法在运行时被创建，而且当需要扩张功能时，这些行为往往无法预测，这就意味着在每个可能的情况下，这个类都需要被创建，所以对比之下，装饰者模式优点在于每个装饰者都是对象，在运行时被创建，并且能够在每次使用时根据需要自己组合。

## UML类图



Component：抽象组件

可以是一个接口或者是抽象类，其充当的就是被装饰的原始对象，用来定义装饰者和被装饰者的基本行为。

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

装饰组件对象，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，如果是装饰逻辑单一，只有一个的情况下我们可以忽略该类直接作为具体的装饰者。

ConcreteDecoratorA 和 ConcreteDecoratorB：装饰者具体实现类

对抽象装饰者的具体实现。

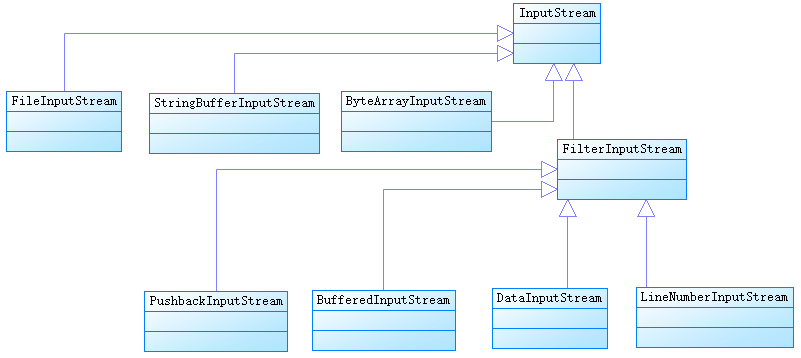
## 扩展原有系统的功能的步骤

在已有的 Component 和 ConcreteComponent 体系下，实现装饰者模式来扩展原有系统的功能，可以分为 5 个步骤

* 继承或者实现 Component 组件，生成一个 Decorator 装饰者抽象类；
* 在生成的这个 Decorator 装饰者类中，增加一个 Component 的私有成员对象；
* 将 ConcreteComponent 或者其他需要被装饰的对象传入 Decorator 类中并赋值给上一步的 Component 对象；
* 在装饰者 Decorator 类中，将所有的操作都替换成该 Component 对象的对应操作；
* 在 ConcreteDecorator 类中，根据需要对应覆盖需要重写的方法。

## Java 中的装饰者模式

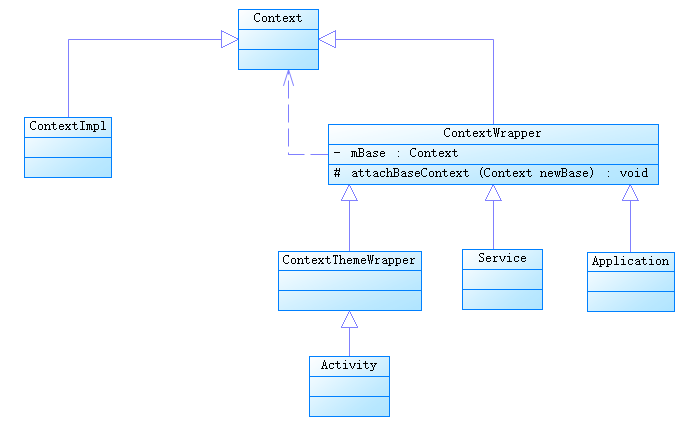
最典型的就是 Java 中的 java.io 包下面的 InputStream 和 OutputStream 相关类了，初学 Java 的时候，看到这些类，头都大了，其实学了装饰者模式之后，再理解这些类就很简单了，画一个简单的类图来表示：



InputStream 类是一个抽象组件， FileInputStream，StringBufferInputStream 和 ByteArrayInputStream 类都是可以被装饰者包起来的具体组件；FilterInputStream 是一个抽象装饰者，所以它的四个子类都是一个个装饰者了。

## Android 中的装饰者模式

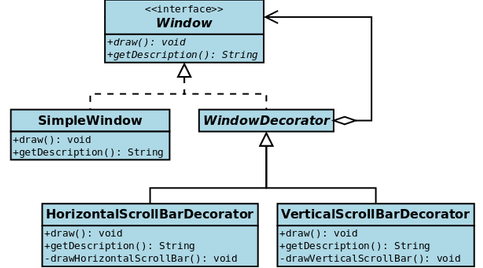
对于 android 开发工程师来说，最最重要的就应该是“上帝类” Context 和其子类了，这些类就不用解释了，上一张类图基本就明确了：



所以对于 Application，Activity 和 Service 等类来说，他们只是一个个装饰者，都是用来装饰 ContextImpl 这个被装饰者类，Application 是在 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；Activity 是通过 handleLaunchActivity 方法设置的 ContextImpl 实例，这个方法会获取到一个Activity对象，在performLaunchActivity函数中会调用该activity的attach方法，这个方法把一个ContextImpl对象attach到了Activity中

## 示例与源码

这里以一个图形系统中的 Window 为例，一般情况窗口都是能够垂直或者是左右滑动的，所以为了能够更好的支持 Window 的滑动，给滑动的 Window 加上一个 ScrollBar 是一个不错的方法，为了重用代码，水平滑动的 Window 和垂直滑动的 Window 我们就能够使用装饰者模式去处理，基本类图如下所示：



根据类图，我们首先实现 Window 这个接口：

**IWindow.class**

**public** **interface** IWindow {

**void** **draw**();

**String** **getDescription**();

}

然后是被装饰者 SimpleWindow 类，它实现了窗口的基本行为：

**SimpleWindow.class**

**public** **class** **SimpleWindow** **implements** IWindow {

@Override

**public** **void** **draw**() {

**Log**.*e*("shawn", "drawing a window");

}

@Override

**public** **String** **getDescription**() {

**return** "a window";

}

}

然后是装饰者类角色的抽象父类：

**WindowDecorator.class**

**public** **abstract** **class** **WindowDecorator** **implements** IWindow{

**private** IWindow window;

**public** **WindowDecorator**(IWindow window) {

**this**.window = window;

}

@Override

**public** **void** **draw**() {

window.draw();

}

@Override

**public** **String** **getDescription**() {

**return** window.getDescription();

}

}

最后是实现该装饰者父类的装饰者子类：

**HorizontalScrollBarDecorator.class**

**public** **class** **HorizontalScrollBarDecorator** **extends** **WindowDecorator** {

**public** **HorizontalScrollBarDecorator**(IWindow window) {

**super**(window);

}

@Override

**public** **void** **draw**() {

**super**.draw();

**Log**.*e*("shawn", "then drawing the horizontal scroll bar");

}

@Override

**public** **String** **getDescription**() {

**return** **super**.getDescription() + " with horizontal scroll bar";

}

}

**VerticalScrollBarDecorator.class**

**public** **class** **VerticalScrollBarDecorator** **extends** **WindowDecorator** {

**public** **VerticalScrollBarDecorator**(IWindow window) {

**super**(window);

}

@Override

**public** **void** **draw**() {

**super**.draw();

**Log**.*e*("shawn", "then drawing the vertical scroll bar");

}

@Override

**public** **String** **getDescription**() {

**return** **super**.getDescription() + " with vertical scroll bar";

}

}

**测试代码**

switch (v.getId()) {

case R.id.btn\_horizontal\_window:

IWindow horizontalWindow = new HorizontalScrollBarDecorator(new SimpleWindow());

horizontalWindow.draw();

Log.e("shawn", "window description : " + horizontalWindow.getDescription());

break;

case R.id.btn\_vertical\_window:

IWindow verticalWindow = new VerticalScrollBarDecorator(new SimpleWindow());

verticalWindow.draw();

Log.e("shawn", "window description : " + verticalWindow.getDescription());

break;

}

**最后的结果**

com.android.decoratorpattern E/shawn: drawing a window

com.android.decoratorpattern E/shawn: then drawing the horizontal scroll bar

com.android.decoratorpattern E/shawn: window description : a window with horizontal scroll bar

com.android.decoratorpattern E/shawn: drawing a window

com.android.decoratorpattern E/shawn: then drawing the vertical scroll bar

com.android.decoratorpattern E/shawn: window description : a window with vertical scroll bar

代码一目了然，结构清晰。其实说到底，每一个写过 Android 程序的人都应该用过装饰者模式，因为每写一个 Activity，就相当于是写了一个装饰者类，不经意间就用了装饰者模式

## 总结

装饰者模式和代理模式有点类似，很多时候需要仔细辨别，容易混淆，倒不是说会把代理模式看成装饰者模式，而是会把装饰者模式看作代理模式。区分一下，装饰者模式的目的是透明地为客户端对象扩展功能，是继承关系的一种替代方案，而代理模式则是给一个对象提供一个代理对象，并由代理对象来控制对原有对象的引用。装饰者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。

同时有几个要点需要提一下：

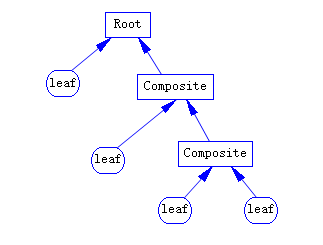
* 继承属于扩展形式之一，但不一定是达到弹性设计的最佳方案；
* 在我们的设计，应该尽量对修改关闭，对扩展开发，无需修改现有代码；
* 组合和委托可用于在运行时动态加上新的行为；
* 装饰者可以在被装饰者行为的前后根据实际情况加上自己的行为，必要时也可以将被装饰者行为给替换掉；
* 可以用无数个装饰者包装一个组件，也就是说，装饰者 A 包装了被装饰者 B ，装饰者 C 再包装装饰者 A，根据实际情况这种行为可以累加到多层，通俗讲就是套上多层外壳；
* 同时，被装饰者也可以存在多个，也就是说 ConcreteComponent 这个角色也可以是多个的。

装饰者模式的优点就是它的特点：可以在运行时动态，透明的为一个组件扩展功能，比继承更加灵活；缺点也很明显：它会导致设计中出现许多小对象，如果过度使用，会让程序变得很复杂

# 组合模式（Composite、Part-Whole）

## 概述

组合模式（Composite Pattern），它也称为部分整体模式（Part-Whole Pattern），结构型模式之一。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。这个最典型的例子就是数据结构中的树了，如果一个节点有子节点，那么它就是枝干节点，如果没有子节点，那么它就是叶子节点，那么怎么把枝干节点和叶子节点统一当作一种对象处理呢？这就需要用到组合模式了。



## 特点

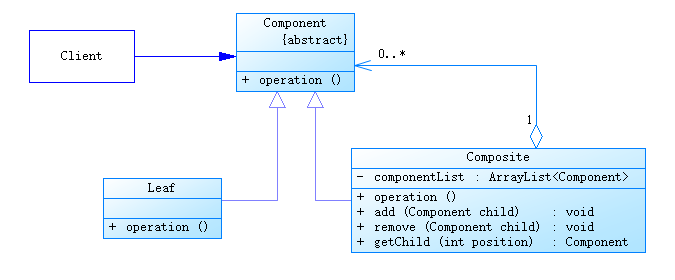
组合模式允许你将对象组合成树形结构来表现“整体/部分”层次结构，并且能够让客户端以一致的方式处理个别对象以及组合对象。组合模式让我们能用树形方式创建对象的结构，树里面包含了组合构件以及叶子构件的对象，而且能够把相同的操作应用在组合构件和叶子构件上，换句话说，在大多数情况下我们可以忽略组合对象和叶子对象之间的差别。组合模式使用的场景：

* 表示对象的部分-整体结构层次时；
* 从一个整体中能够独立出部分模块或功能的场景。

**组合模式在实际使用中会有两种情况：安全的组合模式与透明的组合模式。**

## 安全的组合模式

### 安全组合模式的 uml 类图



* Component：抽象根节点，为组合中的对象声明接口行为，是所有节点的抽象。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：增加定义枝干节点的行为，存储子节点，实现 Component 接口中的有关的操作；
* Leaf：在组合中表示叶子结点对象，叶子节点没有子节点，实现 Component 接口中的全部操作；
* Client：通过 Component，Composite 和 Leaf 类操纵组合节点对象。

### 安全组合模式的通用代码

**Component.class**

**public** **abstract** **class** **Component** {

**public** **abstract** **void** **operation**();

}

**Composite.class**

**public** **class** **Composite** **extends** **Component**{

**private** **ArrayList**<**Component**> componentList = **new** ArrayList<>();

@Override

**public** **void** **operation**() {

**Log**.*e*("shawn", "this is composite " + **this** + " -------start");

**for** (**Component** component : componentList) {

component.operation();

}

**Log**.*e*("shawn", "this is composite " + **this** + " -------end");

}

**public** **void** **add**(**Component** child) {

componentList.add(child);

}

**public** **void** **remove**(**Component** child) {

componentList.remove(child);

}

**public** **Component** **getChild**(**int** position) {

**return** componentList.get(position);

}

}

**Leaf.class**

**public** **class** **Leaf** **extends** **Component**{

@Override

**public** **void** **operation**() {

**Log**.*e*("shawn", "this if leaf " + **this**);

}

}

**Client 测试代码：**

**Composite** root = **new** Composite();

**Leaf** leaf1 = **new** Leaf();

**Composite** branch = **new** Composite();

root.add(leaf1);

root.add(branch);

**Leaf** leaf2 = **new** Leaf();

branch.add(leaf2);

root.operation();

**最后输出结果：**

this is composite com.android.compositepattern.composite.Composite@a37f4d8 -------start

this if leaf com.android.compositepattern.composite.Leaf@1d7d4031

this is composite com.android.compositepattern.composite.Composite@ec97316 -------start

this if leaf com.android.compositepattern.composite.Leaf@5dae497

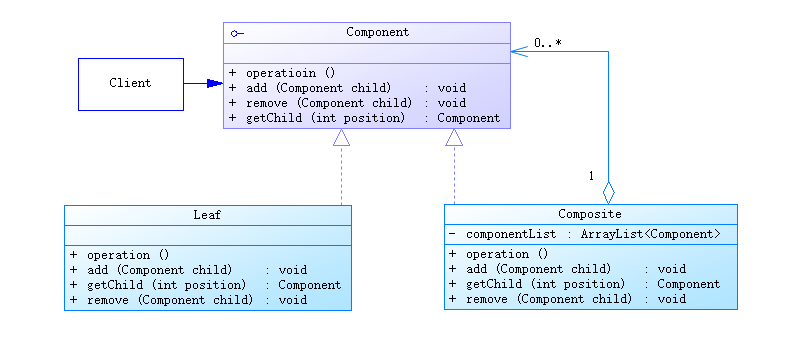
this is composite com.android.compositepattern.composite.Composite@ec97316 -------end

this is composite com.android.compositepattern.composite.Composite@a37f4d8 -------end

代码很简单，结果就是一个简单的树形结构，但是仔细看看客户端代码，就能发现它违反了 6 个设计模式原则中依赖倒置原则，客户端不应该直接依赖于具体实现，而应该依赖于抽象，既然是面向接口编程，就应该把更多的焦点放在接口的设计上，于是这样就产生了透明的组合模式。

## 透明的组合模式

### 透明组合模式 uml 类图



### 透明组合模式的通用代码

和安全的组合模式差异就是在将 Composite 的操作放到了 Component 中，这就造成 Leaf 角色也要实现 Component 中的所有方法。实现的代码做出相应改变：

**Component.class**

**public** **interface** Component {

**void** **operation**();

**void** **add**(Component child);

**void** **remove**(Component child);

Component **getChild**(**int** position);

}

**Composite.class**

**public** **class** **Composite** **implements** Component{

**private** **ArrayList**<Component> componentList = **new** ArrayList<>();

@Override

**public** **void** **operation**() {

**Log**.*e*("shawn", "this is composite " + **this** + " -------start");

**for** (Component component : componentList) {

component.operation();

}

**Log**.*e*("shawn", "this is composite " + **this** + " -------end");

}

@Override

**public** **void** **add**(Component child) {

componentList.add(child);

}

@Override

**public** **void** **remove**(Component child) {

componentList.remove(child);

}

@Override

**public** Component **getChild**(**int** position) {

**return** componentList.get(position);

}

}

**Leaf.class**

**public** **class** **Leaf** **implements** Component {

@Override

**public** **void** **operation**() {

**Log**.*e*("shawn", "this if leaf " + **this**);

}

@Override

**public** **void** **add**(Component child) {

**throw** **new** UnsupportedOperationException("leaf can't add child");

}

@Override

**public** **void** **remove**(Component child) {

**throw** **new** UnsupportedOperationException("leaf can't remove child");

}

@Override

**public** Component **getChild**(**int** position) {

**throw** **new** UnsupportedOperationException("leaf doesn't have any child");

}

}

**Client 测试代码**

Component root = **new** Composite();

Component leaf1 = **new** Leaf();

Component branch = **new** Composite();

root.add(leaf1);

root.add(branch);

Component leaf2 = **new** Leaf();

branch.add(leaf2);

root.operation();

最后产生的结果是一样的，由于是在 Component 类中定义了所有的行为，所以客户端就不用直接依赖于具体 Composite 和 Leaf 类的实现，遵循了依赖倒置原则——依赖抽象，而不依赖具体实现。但是也违反了单一职责原则与接口隔离原则，让 Leaf 类继承了它本不应该有的方法，并且不太优雅的抛出了 UnsupportedOperationException ，这样做的目的就是为了客户端可以透明的去调用对应组件的方法，将枝干节点和子节点一视同仁。

另外，将 Component 写成一个虚基类，并且实现所有的 Composite 方法，而且默认都抛出异常，只让 Composite 去覆盖重写父类的方法，而 Leaf 类就不需要去实现 Composite 的相关方法，这么去实现当然也是可以的。

## 安全组合和透明组合的对比

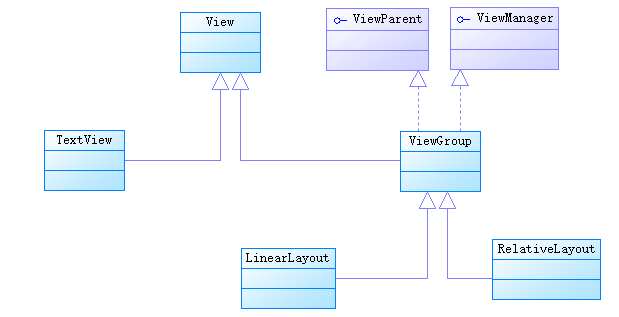
安全的组合模式将责任区分开来放在不同的接口中，这样一来，设计上就比较安全，也遵循了单一职责原则和接口隔离原则，但是也让客户端必须依赖于具体的实现；

透明的组合模式，以单一职责原则和接口隔离原则原则换取透明性，遵循依赖倒置原则，客户端就直接依赖于 Component 抽象即可，将 Composite 和 Leaf 一视同仁，也就是说，一个元素究竟是枝干节点还是叶子节点，对客户端是透明的。

所以这是一个很典型的折衷案例，尽管我们受到设计原则的指导，但是我们总是需要观察某原则对我们的设计所造成的影响。有时候这个需要去根据实际案例去分析，毕竟有些时候 6 种设计模式原则在实际使用过程中是会冲突的，是让客户端每次使用的时候都去先检查类型还是赋予子节点不应该有的行为，这都取决于设计者的观点，总体而言，这两种方案都是可行的。

## 示例与源码

组合模式在实际生活过程中的例子就数不胜数了，比如菜单、文件夹等等。我们这就以 Android 中非常经典的实现为例来分析一下。View 和 ViewGroup 想必应该都非常熟悉，其实他们用到的就是组合模式，我们先来看看他们之间的 uml 类图：



**ViewManager.class**

public interface ViewManager {

public void addView(View view, ViewGroup.LayoutParams params);

public void updateViewLayout(View view, ViewGroup.LayoutParams params);

public void removeView(View view);

}

只定义了关于 View 操作的三个方法。ViewParent 类是用来定义一个 父 View 角色所具有的职责，在 Android 中，一般能成为父 View 的也只有 ViewGroup

**ViewGroup.class**

public interface ViewParent {

public void requestLayout();

public boolean isLayoutRequested();

....

}

从 uml 类图中可以注意到一点，ViewGroup 和 View 使用的安全的组合模式，而不是透明的组合模式，怪不得有时候使用前需要将 View 强转成 ViewGroup

## 总结

使用组合模式，我们能把相同的操作应用在组合和个别对象上，换句话说，在大多数情况下，我们可以忽略对象组合和个别对象之间的差别。组合模式适用于一些界面 UI 的结构设计上，典型的例子就是Android，iOS 和 Java 等都提供了相应的 UI 框架。

### 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，无需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建类型进行限制，不能依赖类型系统来施加这些约束，因为在大多数情况下，他们都来自于相同的抽象层，此时，必须进行类型检查来实现，这个实现过程较为复杂。

# 外观模式（Facade）

## 概述

外观模式也称为门面模式，它在开发过程中运用频率非常高，尤其是第三方 SDK 基本很大概率都会使用外观模式。通过一个外观类使得整个子系统只有一个统一的高层的接口，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。其实我们在开发过程中可能已经使用过很多次外观模式，只是没有从理论层面去了解它。

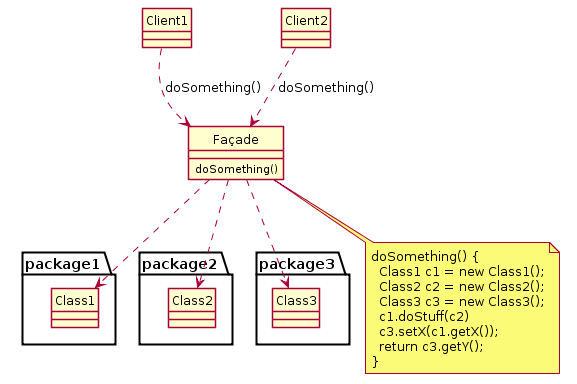
## 特点

外观模式提供一个统一的接口，用来访问子系统中的一群接口，外观定义了一个高层接口，让子系统更容易使用。外观模式简化了接口，并且同时允许我们让客户端和子系统之间避免紧耦合。外观模式的使用场景：

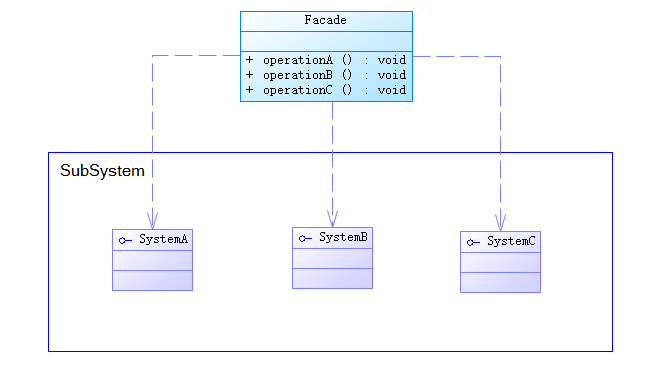
* 为一个复杂子系统提供一个简单接口。Facade 可以提供一个简单统一的接口，对外隐藏子系统的具体实现、隔离变化。
* 当需要构建一个层次结构的子系统时，使用 Facade 模式定义子系统中每层的入口点。如果子系统之间是相互依赖，你可以让他们仅通过 Facade 接口进行通信，从而简化了它们之间的依赖关系。

## UML类图

外观模式没有一个一般化的类图描述，我们先用一个结构图来说明：



根据结构图抽象出一个类图：



外观模式有两个角色：

* Facade 角色：系统对外的统一接口，客户端连接子系统功能的入口。
* 子系统角色（SubSystem）角色：可以同时有一个或者多个子系统，每个子系统都不是一个单独的类，而是一个类的集合。每个子系统都可以被客户端直接调用，或者被门面角色调用。子系统并不知道门面的存在，对于子系统而言，门面仅仅是另外一个客户端而已。

## 外观模式的通用代码：

**子系统：**

**public** **class** **SystemA** {

**public** **void** **operation1**(){

**System**.**out**.print("SystemA:operation1\n");

}

**public** **void** **operation2**(){

**System**.**out**.print("SystemA:operation2\n");

}

**public** **void** **operation3**(){

**System**.**out**.print("SystemA:operation3\n");

}

}

**public** **class** **SystemB** {

**public** **void** **operation1**(){

**System**.**out**.print("SystemB:operation1\n");

}

**public** **void** **operation2**(){

**System**.**out**.print("SystemB:operation2\n");

}

**public** **void** **operation3**(){

**System**.**out**.print("SystemB:operation3\n");

}

}

**然后是外观对象 IFacade.class**

**public** **interface** IFacade {

**void** **operationA**();

**void** **operationB**();

**void** **operationC**();

}

**Facade.class**

**public** **class** **Facade** **implements** IFacade{

**private** **SystemA** systemA = **new** SystemA();

**private** **SystemB** systemB = **new** SystemB();

@Override

**public** **void** **operationA**() {

systemA.operation1();

systemB.operation1();

}

@Override

**public** **void** **operationB**() {

systemA.operation2();

systemB.operation2();

}

@Override

**public** **void** **operationC**() {

systemA.operation3();

systemB.operation3();

}

}

**最后的测试代码：**

IFacade facade = **new** Facade();

facade.operationA();

facade.operationB();

facade.operationC();

**结果输出如下：**

SystemA:operation1

SystemB:operation1

SystemA:operation2

SystemB:operation2

SystemA:operation3

SystemB:operation3

## 示例与源码

这里直接展示一段简单电脑启动时的伪代码来表示即可：

/\* Complex parts \*/

**class** **CPU** {

**public** **void** freeze() { ... }

**public** **void** jump(**long** position) { ... }

**public** **void** execute() { ... }

}

**class** **Memory** {

**public** **void** load(**long** position, **byte**[] data) { ... }

}

**class** **HardDrive** {

**public** **byte**[] read(**long** lba, **int** size) { ... }

}

/\* Facade \*/

**class** **ComputerFacade** {

**private** **CPU** processor;

**private** **Memory** ram;

**private** **HardDrive** hd;

**public** **ComputerFacade**() {

**this**.processor = **new** CPU();

**this**.ram = **new** Memory();

**this**.hd = **new** HardDrive();

}

**public** **void** **start**() {

processor.freeze();

ram.load(BOOT\_ADDRESS, hd.read(BOOT\_SECTOR, SECTOR\_SIZE));

processor.jump(BOOT\_ADDRESS);

processor.execute();

}

}

/\* Client \*/

**class** **You** {

**public** **static** **void** **main**(**String**[] args) {

**ComputerFacade** computer = **new** ComputerFacade();

computer.start();

}

}

## 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。外观类遵循了一个很重要设计模式原则：迪米特原则（最少知识原则），它让客户端依赖于最少的类，直接依赖外观类而不是依赖于所有的子系统类。

### 优点：

* 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；
* 外观类对子系统的接口封装，使得系统更易于使用；
* 更好的划分访问层次，通过合理使用Facade，可以帮助我们更好地划分访问的层次。有些方法是对系统外的，有些方法是系统内部使用的。把需要暴露给外部的功能集中到外观类中，这样既方便客户端使用，也很好地隐藏了内部的细节。

### 缺点：

* 外观类接口膨胀，由于子系统的接口都由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；
* 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。

# 代理模式（Proxy）

## 概述

代理模式（Proxy Pattern），代理模式也称为委托模式，是一个非常重要的设计模式，不少设计模式也都会有代理模式的影子。代理在我们日常生活中也很常见，比如上网时连接的代理服务器地址，更比如我们平时租房子，将找房子的过程代理给中介等等，都是代理模式在日常生活中的使用例子。

代理模式中的代理对象能够连接任何事物：一个网络连接，一个占用很多内存的大对象，一个文件，或者是一些复制起来代价很高甚至根本不可能复制的一些资源。总之，代理是一个由客户端调用去访问幕后真正服务的包装对象，使用代理可以很容易地转发到真正的对象上，或者在此基础上去提供额外的逻辑。代理模式中也可以提供额外的功能，比如在资源集中访问操作时提供缓存服务，或者在操作真正执行到对象之前进行前提条件的检查。对于客户端来说，使用代理对象，和使用真正的对象其实是类似的，因为他们都实现了同样的接口。

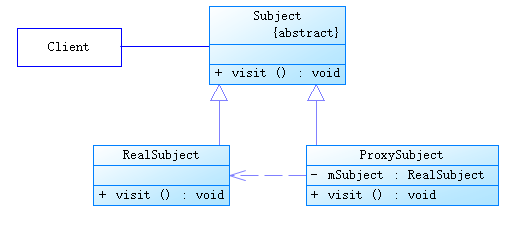
## 特点

代理模式为一个对象提供一个代理，以控制对这个对象的访问。使用代理模式创建代理，让代理对象控制对某个对象的访问，被代理的对象可以是远程的对象，创建开销很大的对象，或者是需要安全控制的对象，所以代理模式的使用场景为：当无法或者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，为了保证客户端的透明性，委托对象与代理对象需要实现同样的接口。

　　代理模式可以大致分为静态代理和动态代理。静态代理模式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，也就是说我们的代码在运行前代理类的 class 编译文件就已经存在；而动态代理则与静态代理相反，在 Java 或者 Android 中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 阶段完全不需要知道代理谁，代理谁我们将会在执行阶段决定，在 Java 中，也提供了相关的动态代理接口 InvocationHandler 类，这个我们在后面源码的时候会用到。 代理模式根据实际使用的场景也可以分为以下几种，静态和动态代理都可以应用于上述的几种情形，两者是各自独立变化的：

* 远程代理（Remote Proxy）：为某个在不同的内存地址空间的对象提供局部代理，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，类似于 C/S 模式（主要拦截并控制远程方法的调用，做代理防火墙之类的）；
* 虚拟代理（Virtual Proxy）：使用一个代理对象标识一个十分耗资源的对象，并在真正需要时才创建，实现一个延迟加载的机制；
* 保护代理（Protection Proxy）：是用代理控制对原始对象的访问，该类型的代理通常被用于原始对象有不同访问权限的情况；
* 智能引用（Smart Proxy）：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；
* 写时拷贝（克隆）代理（Copy-on-write Proxy）：其实是虚拟代理的一个分支，提供了拷贝大对象的时候只有在对象真正变化后才会进行拷贝（克隆）的操作，即延迟拷贝。

## UML类图



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；

RealSubjct：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，而客户端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对应的接口方法，以此起到代理的作用；

Client：客户类，

使用代理类的部分

## 代理模式的通用代码：

**Subject.class**

**public** **abstract** **class** **Subject** {

**public** **abstract** **void** **operation**();

}

**RealSubject.class**

**public** **class** **RealSubject** **extends** **Subject**{

@Override

**public** **void** **operation**() {

//the real operation

}

}

**ProxySubject.class**

**public** **class** **ProxySubject** **extends** **Subject**{

**private** **Subject** realSubject;

**public** **ProxySubject**(**Subject** realSubject) {

**this**.realSubject = realSubject;

}

@Override

**public** **void** **operation**() {

**if** (realSubject != **null**) {

realSubject.operation();

} **else** {

//do something else

}

}

}

**客户端 Client 代码**

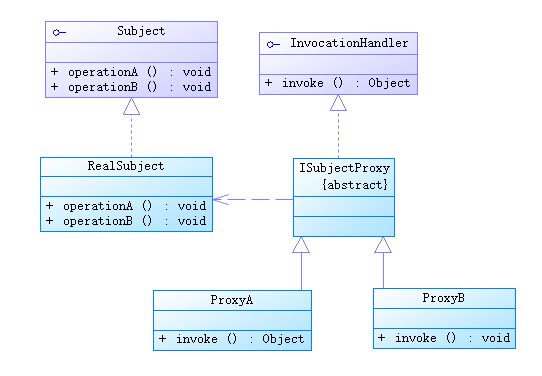
**ProxySubject** subject = **new** ProxySubject(**new** RealSubject());

subject.operation();

## 动态+保护代理模式示例与源码

上面用到的是静态代理，都是在编译阶段生成的 class 文件，所以这里以一个动态+保护代理模式的 demo 为例：

**UML图**



这里我们用到了 InvocationHandler 这个类，这是 Java 为我们提供的一个便捷动态代理接口，作用就是用来在运行时动态生成代理对象。其次是保护代理，这里就以 ProxyA 和 ProxyB 两个代理类为例，一个只能调用 operationA 方法，另一个只能调用 operationB 方法。首先来看看 Subject 和 RealSubject 类：

**Subject.class**

**public** **interface** Subject {

**String** **operationA**();

**String** **operationB**();

}

**RealSubjct.class**

**public** **class** **RealSubject** **implements** Subject{

@Override

**public** **String** **operationA**() {

**return** "this is operationA";

}

@Override

**public** **String** **operationB**() {

**return** "this is operationB";

}

}

**定义完了抽象主题类和真实主题类之后，开始构造代理类，首先是代理类的虚基类：**

**ISubjectProxy.class**

**public** **abstract** **class** **ISubjectProxy** **implements** InvocationHandler {

**protected** Subject subject;

**public** **ISubjectProxy**(Subject subject) {

**this**.subject = subject;

}

}

**然后是 ProxyA 和 ProxyB 这两个代理子类：**

**ProxyA.class**

**public** **class** **ProxyA** **extends** **ISubjectProxy**{

**public** **ProxyA**(Subject subject) {

**super**(subject);

}

@Override

**public** **Object** **invoke**(**Object** proxy, **Method** method, **Object**[] args) **throws** Throwable {

**if** (method.getName().equals("operationB")){

**throw** **new** UnsupportedOperationException("ProxyA can't invoke operationB");

}**else** **if** (method.getName().equals("operationA")) {

**return** method.invoke(subject, args);

}

**return** **null**;

}

}

**ProxyB.class**

**public** **class** **ProxyB** **extends** **ISubjectProxy**{

**public** **ProxyB**(Subject subject) {

**super**(subject);

}

@Override

**public** **Object** **invoke**(**Object** proxy, **Method** method, **Object**[] args) **throws** Throwable {

**if** (method.getName().equals("operationA")){

**throw** **new** UnsupportedOperationException("ProxyB can't invoke operationA");

} **else** **if** (method.getName().equals("operationB")) {

**return** method.invoke(subject, args);

}

**return** **null**;

}

}

**最后是 Client 的代码：**

@Override

**public** **void** **onClick**(**View** v) {

Subject subject = **new** RealSubject();

**ISubjectProxy** proxy = **null**;

**switch** (v.getId()) {

**case** **R**.**id**.**proxy\_a**:

proxy = **new** ProxyA(subject);

**break**;

**case** **R**.**id**.**proxy\_b**:

proxy = **new** ProxyB(subject);

**break**;

}

//.newProxyInstance([ClassLoader](eclipse-javadoc:%E2%98%82=%E6%A8%A1%E5%BC%8F--%E4%BB%A3%E7%90%86/D:\\/adt-bundle-windows-x86_64-20140702\\/sdk\\/platforms\\/android-22\\/android.jar<java.lang.reflect(Proxy.class%E2%98%83Proxy~newProxyInstance~Ljava.lang.ClassLoader;~\\%E2%98%83Ljava.lang.Class\\<*>;~Ljava.lang.reflect.InvocationHandler;%E2%98%82ClassLoader) loader, [Class](eclipse-javadoc:%E2%98%82=%E6%A8%A1%E5%BC%8F--%E4%BB%A3%E7%90%86/D:\\/adt-bundle-windows-x86_64-20140702\\/sdk\\/platforms\\/android-22\\/android.jar<java.lang.reflect(Proxy.class%E2%98%83Proxy~newProxyInstance~Ljava.lang.ClassLoader;~\\%E2%98%83Ljava.lang.Class\\<*>;~Ljava.lang.reflect.InvocationHandler;%E2%98%82Class)<?>[] interfaces, [InvocationHandler](eclipse-javadoc:%E2%98%82=%E6%A8%A1%E5%BC%8F--%E4%BB%A3%E7%90%86/D:\\/adt-bundle-windows-x86_64-20140702\\/sdk\\/platforms\\/android-22\\/android.jar<java.lang.reflect(Proxy.class%E2%98%83Proxy~newProxyInstance~Ljava.lang.ClassLoader;~\\%E2%98%83Ljava.lang.Class\\<*>;~Ljava.lang.reflect.InvocationHandler;%E2%98%82InvocationHandler) invocationHandler)

Subject sub = (Subject) **Proxy**.*newProxyInstance*(subject.getClass().getClassLoader(),

subject.getClass().getInterfaces(), proxy);

**try** {

**Log**.*e*("Shawn", sub.operationA());

}**catch** (**UnsupportedOperationException** e){

**Log**.*e*("Shawn", e.getMessage());

}

**try** {

**Log**.*e*("Shawn", sub.operationB());

}**catch** (**UnsupportedOperationException** e){

**Log**.*e*("Shawn", e.getMessage());

}

}

**结果**

this is operationA

ProxyA can't invoke operationB

ProxyB can't invoke operationA

this is operationB

代码很简单，第一点是使用 InvocationHandler 实现了动态代理，第二点是根据 method 的名字实现了保护代理。这里只总结了保护代理，虚拟代理这里简单举个例子： moduleA 为 modleB 的 lib，所以如果在 moduleA 中需要用到 moduleB 实现的对象，就可以使用代理模式，具体步骤是在 moduleA 中定义接口，moduleB 去实现该接口，moduleA 中定义一个代理对象并提供一些默认操作，当 moduleB 的相关模块初始化之后，将该对象设置到 moduleA 的代理对象中以替代原来的默认实现，之后代理对象就能成功调用 moduleB 中定义的行为了，而且也实现了延迟加载。远程代理和其他的代理模式大家去网上查阅一下相关资料，万变不离其宗，道理是一样的。

## 总结

代理模式应用广泛，超级广泛，很多设计模式都会有代理模式的影子，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，而且代理模式的缺点也很少，总结一下代理模式的优缺点：

### 优点：

* 代理作为调用着和真实对象的中间层，降低了模块间和系统的耦合性；
* 可以以一个小对象代理一个大对象，达到优化系统提高运行速度的目的；
* 提供 RealSubject 的权限管理；
* 容易扩展，RealSubject 和 ProxySubject 都接口化了，RealSubject更改业务后只要接口不变，ProxySubject 可以不做任何修改。

### 缺点：

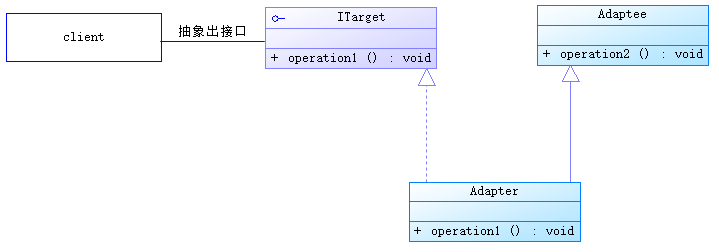
* 调用者和真实对象多了一个中间层，对象的创建，函数调用使用的反射等都增加调用响应的时间；
* 设计模式的通病：类的增加，不过这点也不严重。

# 适配器 VS 装饰者 VS 桥接 VS 代理 VS 外观

**这几个都是结构型设计模式，他们有些类似，在实际使用过程中也容易搞混，我们在这就给他们做一个对比：**

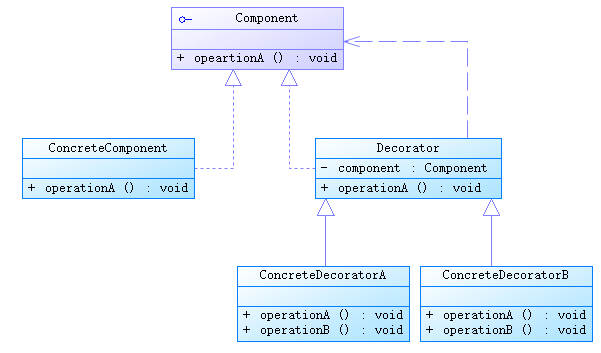
**适配器模式**

适配器模式和其他四个设计模式一般不容易搞混，它的作用是将原来不兼容的两个类融合在一起，uml 图也和其他的差别很大。



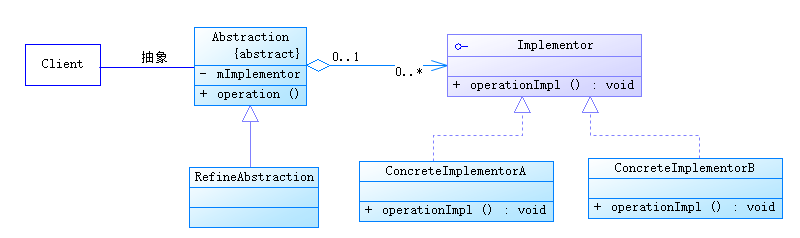
**装饰者模式**

装饰者模式结构上类似于代理模式，但是和代理模式的目的是不一样的，装饰者是用来动态地给一个对象添加一些额外的职责，装饰者模式为对象加上行为，而代理则是控制访问。



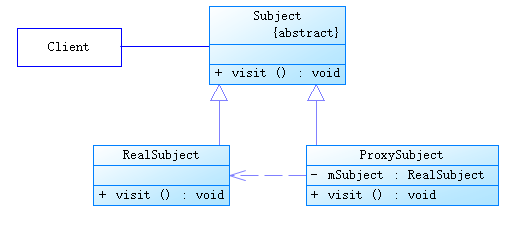
**桥接模式**

桥接模式的目的是为了将抽象部分与实现部分分离，使他们都可以独立地进行变化，所以说他们两个部分是独立的，没有实现自同一个接口，这是桥接模式与代理模式，装饰者模式的区别。



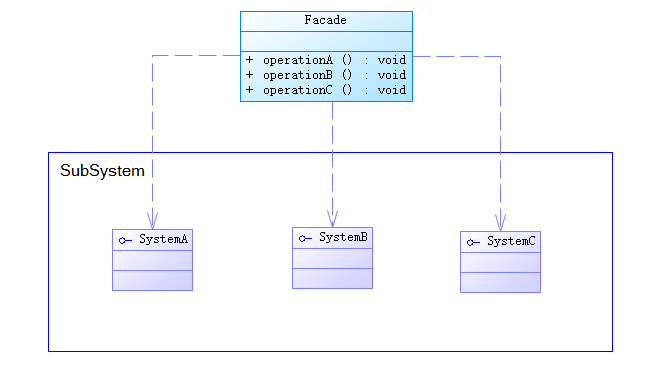
**代理模式**

代理模式为另一个对象提供代表，以便控制客户对对象的访问，管理的方式有很多种，比如远程代理和虚拟代理等，这个在上面有，这里就不说了，而装饰者模式则是为了扩展对象。



**外观模式**

外观模式提供一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。 适配器模式将一个或多个类接口变成客户端所期望的一个接口，虽然大多数资料所采用的例子中适配器只适配一个类，但是你可以适配许多类来提供一个接口让客户端访问；类似的，外观模式 也可以只针对一个拥有复杂接口的类提供简化的接口，两种模式的差异，不在于他们“包装”了几个类，而是在于它们的意图。适配器模式 的意图是，“改变”接口符合客户的期望；而外观模式的意图是，提供子系统的一个简化接口。



# ----行为型----观察者模式（Observer）

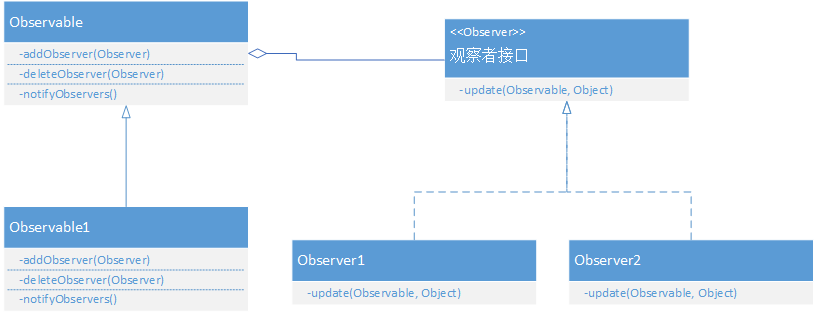
## 特点

　　观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象状态发生改变时，它的所有依赖者都会收到通知并自动更新。它实现了 Subject 和 Observer 之间的松耦合，Subject只知道观察者实现了 Observer 接口，主题不需要知道具体的类是谁、做了些什么或其他任何细节。任何时候我们都可以增加新的观察者。因为主题唯一依赖的东西是一个实现Observer 接口的对象列表，所以我们可以随时增加观察者，同样的，也可以在任何时候删除观察者，当然更不用去关心观察者的类型，只要实现了Observer接口即可，Subject 最后只会发通知给实现了 Observer 接口的观察者。Subject 和 Observer 之间实现松耦合之后，双方代码的修改都不会影响到另外一方，当然前提是双方得遵守接口的规范（接口隔离原则）。

**观察者模式使用的场景：**

* 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。
* 广播机制
* ListView数据更改
* 点击事件
* ContentObserver

## UML类图



**Observable:**

也叫Subject，抽象主题，也就是被观察的角色，抽象主题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。

**Observable1：**

也叫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，具体主题角色对象又叫做具体被观察者角色。

**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。

**Observer1、2：**

ConcreteObserver，具体的观察者，该对象实现抽象观察者角色所定义的更新接口，以便在主题的状态变化时更新自身的状态。

## 有几点需要注意的是

* Subject 和 Observer 是一个一对多的关系，也就是说观察者只要实现 Observer 接口并把自己注册到 Subject 中就能够接收到消息事件；
* Java API有内置的观察者模式类：java.util.Observable 类和 java.util.Observer 接口，这分别对应着 Subject 和 Observer 的角色；
* 使用 Java API 的观察者模式类，需要注意的是被观察者在调用 notifyObservers() 函数通知观察者之前一定要调用 setChanged() 函数，要不然观察者无法接到通知；
* 使用 Java API 的缺点也很明显，由于 Observable 是一个类，java 只允许单继承的缺点就导致你如果同时想要获取另一个父类的属性时，你只能选择适配器模式或者是内部类的方式，而且由于 setChanged() 函数为 protected 属性，所以你除非继承 Observable 类，否则你根本无法使用该类的属性，这也违背了设计模式的原则：多用组合，少用继承

## Java Api源码示例

### Observer是一个接口，定义了一个update方法。

**public** **interface** Observer {

**void** **update**(**Observable** observable, **Object** data);

}

### Observable是一个类，可被继承

**public** **class** **Observable**{

List<Observer> observers = **new** ArrayList<Observer>();//存观察者的list集合

**boolean** changed = **false**;//是否改变的标志

**public** Observable() {

}

**public** **void** **addObserver**(Observer observer) {

**if** (observer == **null**) {

**throw** **new** NullPointerException("observer == null");

}

**synchronized** (**this**) {

**if** (!observers.contains(observer))

observers.add(observer);

}

}

**protected** **void** **clearChanged**() {

changed = **false**;

}

**public** **int** **countObservers**() {//返回观察者的数量

**return** observers.size();

}

**public** **synchronized** **void** **deleteObserver**(Observer observer) {//删除单个观察者

observers.remove(observer);

}

**public** **synchronized** **void** **deleteObservers**() {//删除所有观察者

observers.clear();

}

**public** **boolean** **hasChanged**() {

**return** changed;

}

**public** **void** **notifyObservers**() {//不含参数

notifyObservers(**null**);

}

**public** **void** **notifyObservers**(**Object** data) {

**int** size = 0;

Observer[] arrays = **null**;

**synchronized** (**this**) {

**if** (hasChanged()) {//判断改变

clearChanged();//清除改变

size = observers.size();

arrays = **new** Observer[size];

observers.toArray(arrays);

}

}

**if** (arrays != **null**) {

**for** (Observer observer : arrays) {//正序通知

observer.update(**this**, data);

}

}

}

**protected** **void** **setChanged**() {

changed = **true**;

}

}

### 基本应用

#### 被观察者

**public** **class** **SimpleObservable** **extends** **Observable** {

**private** **int** data = 0;

**public** **int** **getData**() {

**return** data;

}

**public** **void** **setData**(**int** data) {

**if** (**this**.data != data) {

**this**.data = data;

setChanged();//先调这个函数，表明已改变

notifyObservers("gagafa");//通知所有观察者

}

}

}

#### 观察者1、2、3...

**public** **class** **SimpleObserver** **implements** Observer{

**private** **Context** mContext;

**public** **SimpleObserver**(**Context** context, **SimpleObservable** observable) {

mContext = context;

observable.addObserver(**this**);//被观察者添加观察者

}

@Override

**public** **void** **update**(**Observable** observable, **Object** data) {

**String** className = **this**.getClass().getSimpleName() + "----";

**String** oData = ((**SimpleObservable**)observable).getData() + "----";

oData += data.toString();

**Toast**.*makeText*(mContext, className + oData, **Toast**.**LENGTH\_SHORT**).show();

**Log**.*e*("tag", className + oData);

}

}

#### 调用

**public** **class** **MainActivity** **extends** **Activity** {

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

**SimpleObservable** observable = **new** SimpleObservable();//初始化被观察者

**new** SimpleObserver(**MainActivity**.**this**, observable);//初始化1号观察者

**new** OtherObserver(**MainActivity**.**this**, observable);//初始化2号观察者

observable.setData(1);//改变data，会通知

observable.setData(2);

observable.setData(4);

/\*输出

OtherObserver----1----gagafa

SimpleObserver----1----gagafa

OtherObserver----2----gagafa

SimpleObserver----2----gagafa

OtherObserver----4----gagafa

SimpleObserver----4----gagafa

\*/

}

}

## 异步观察者模式

上面介绍的都是同步观察者模式。同步观察者模式会有阻塞问题。各个观察者按照顺序执行update方法。一旦有一个observer比较耗时的话，后序的observer也得等着。异步观察者模式就不一样，不会有这样的阻塞问题。自定义异步的

### 异步观察模式的观察者接口、观察者的包装类

/\*\*

\* 异步观察模式的观察者接口

\*/

**public** **interface** AsyncObserver {

**void** **update**(**AsyncObservable** o, **Object** arg);

/\*\*

\* 观察者的包装类（内部类，只在被观察者基类里包装观察者基类）

\*/

**public** **class** **Wrapper** {

**private** AsyncObserver observer;

**public** **Wrapper**(AsyncObserver o) {

**this**.observer = o;

}

**public** AsyncObserver **getAsyncObserver**() {

**return** observer;

}

**public** **void** **update**(**AsyncObservable** observable, **Object** o) {

**new** Handler(observable, o).start();

}

**class** **Handler** **extends** **Thread** {

**AsyncObservable** observable;

**Object** object;

**public** **Handler**(**AsyncObservable** observable, **Object** o) {

**this**.observable = observable;

**this**.object = o;

}

@Override

**public** **void** **run**() {

observer.update(observable, object);

}

}

}

}

### 异步观察者模式的被观察者对象

**public** **class** **AsyncObservable** {

List<**Wrapper**> wappers = **new** ArrayList<**Wrapper**>();

**boolean** changed = **false**;

**public** **AsyncObservable**() {

}

//根据AsyncObserver添加wapper

**public** **void** **addObserver**(AsyncObserver o) {

**if** (o == **null**){

**throw** **new** NullPointerException();

}

**Wrapper** wrapper = **new** Wrapper(o);

**if** (!wappers.contains(wrapper)) {

wappers.add(wrapper);

}

}

//根据AsyncObserver删除单个wapper

**public** **synchronized** **void** **deleteObserver**(AsyncObserver o) {

Iterator<**Wrapper**> iterator = wappers.iterator();

**while** (iterator.hasNext()) {

**Wrapper** wrapper = iterator.next();

**if** (wrapper.getAsyncObserver() == o) {

wappers.remove(wrapper);

**break**;

}

}

}

**public** **void** **notifyObservers**() {

notifyObservers(**null**);

}

//通知改变，wrapper自己有异步updata方法

**public** **void** **notifyObservers**(**Object** arg) {

**Object**[] arrLocal;

**synchronized** (**this**) {

**if** (hasChanged()){

arrLocal = wappers.toArray();

clearChanged();

}

}

**for** (**Wrapper** wrapper : wappers)

wrapper.update(**this**, arg);

}

**public** **synchronized** **void** **deleteObservers**() {

wappers.clear();

}

**protected** **synchronized** **void** **setChanged**() {

changed = **true**;

}

**protected** **synchronized** **void** **clearChanged**() {

changed = **false**;

}

**public** **synchronized** **boolean** **hasChanged**() {

**return** changed;

}

**public** **synchronized** **int** **countObservers**() {

**return** wappers.size();

}

}

### 异步观察者模式调用示例

#### 被观察者

**public** **class** **AsyncSimpleObservable** **extends** **AsyncObservable** {

**private** **int** data = 0;

**public** **int** **getData**() {

**return** data;

}

**public** **void** **setData**(**int** data) {

**if** (**this**.data != data) {

**this**.data = data;

setChanged();//先调这个函数，表明已改变

notifyObservers("gagafa");//通知所有观察者

}

}

}

#### 观察者1、2、3...

**public** **class** **AsyncSimpleObserver** **implements** AsyncObserver{

**private** **Context** mContext;

**public** **AsyncSimpleObserver**(**Context** context, **AsyncObservable** observable) {

mContext = context;

observable.addObserver(**this**);//被观察者添加观察者

}

@Override

**public** **void** **update**(**AsyncObservable** observable, **Object** data) {

**String** className = **this**.getClass().getSimpleName() + "----";

**String** oData = ((**AsyncSimpleObservable**)observable).getData() + "----";

oData += data.toString();

//Toast.makeText(mContext, className + oData, Toast.LENGTH\_SHORT).show();异步中不可以操作view

**Log**.*e*("tag", className + oData);

}

}

#### 调用

**public** **class** **MainActivity** **extends** **Activity** {

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

**AsyncSimpleObservable** observable = **new** AsyncSimpleObservable();

**new** AsyncSimpleObserver(**MainActivity**.**this**, observable);

**new** AsyncOtherObserver(**MainActivity**.**this**, observable);

observable.setData(1);//改变data，会通知

observable.setData(2);

observable.setData(4);

}

}

# 责任链模式（Chain-of-responsibility）

## 概述

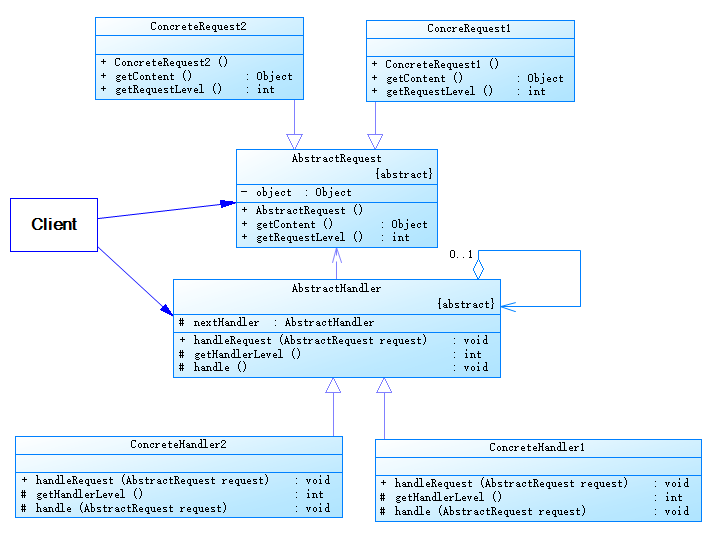
责任联模式又称为职责链模式，是行为型设计模式之一。顾名思义，责任链模式中存在一个链式结构，多个节点首尾相连，每个节点都可以被拆开再连接，因此，链式结构具有很好的灵活性。将这样一种结构应用于编程领域，将每一个节点看作是一个对象，每一个对象拥有不同的处理逻辑，将一个请求从链式的首段发出，沿着链的路径依次传递给每一个节点对象，直至有对象处理这个请求为止，这就是责任链或者职责链的通俗定义。

## 特点

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，直到有对象处理它为止。责任链模式的使用场景：

* 多个对象可以处理统一请求，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。

## UML类图



* Handler：抽象处理者角色，声明一个请求处理的方法，并在其中保持对下一个处理节点 Hanlder 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。

对于请求 Request 来说，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

## 责任联模式的通用代码

**首先是请求角色：**

**AbstractRequest.class**

**public** **abstract** **class** **AbstractRequest** {

**private** **Object** object;

**public** **AbstractRequest**(**Object** object) {

**this**.object = object;

}

**public** **Object** **getContent**() {

**return** object;

}

**public** **abstract** **int** **getLevel**();

}

**ConcreteRequest1.class**

**public** **class** **ConcreteRequest1** **extends** **AbstractRequest**{

**public** **ConcreteRequest1**(**Object** object) {

**super**(object);

}

@Override

**public** **int** **getLevel**() {

**return** 1;

}

}

**ConcreteRequest2.class**

**public** **class** **ConcreteRequest2** **extends** **AbstractRequest**{

**public** **ConcreteRequest2**(**Object** object) {

**super**(object);

}

@Override

**public** **int** **getLevel**() {

**return** 2;

}

}

**然后是Handler 角色**

**AbstractHandler.class**

**public** **abstract** **class** **AbstractHandler** {

**protected** **AbstractHandler** nextHandler;

**public** **final** **void** **handleRequest**(**AbstractRequest** request) {

**if** (getHandlerLevel() == request.getLevel()) {

handle(request);

} **else** {

**if** (nextHandler != **null**) {

nextHandler.handleRequest(request);

} **else** {

**System**.**out**.print("there is no handler that can handle this request");

}

}

}

**protected** **abstract** **int** **getHandlerLevel**();

**protected** **abstract** **void** **handle**(**AbstractRequest** request);

}

**ConcreteHandler1.class**

**public** **class** **ConcreteHandler1** **extends** **AbstractHandler**{

@Override

**protected** **int** **getHandlerLevel**() {

**return** 1;

}

@Override

**protected** **void** **handle**(**AbstractRequest** request) {

**System**.**out**.print("ConcreteHandler1 handle this request : " + request.getContent() + "\n");

}

}

**ConcreteHandler2.class**

**public** **class** **ConcreteHandler2** **extends** **AbstractHandler**{

@Override

**protected** **int** **getHandlerLevel**() {

**return** 2;

}

@Override

**protected** **void** **handle**(**AbstractRequest** request) {

**System**.**out**.print("ConcreteHandler2 handle this request : " + request.getContent() + "\n");

}

}

**最后客户端测试代码：**

**AbstractHandler** handler1 = **new** ConcreteHandler1();

**AbstractHandler** handler2 = **new** ConcreteHandler2();

handler1.nextHandler = handler2;

**AbstractRequest** request1 = **new** ConcreteRequest1("request1");

**AbstractRequest** request2 = **new** ConcreteRequest2("request2");

handler1.handleRequest(request1);

handler1.handleRequest(request2);

**结果如下：**

ConcreteHandler1 handle this request : request1

ConcreteHandler2 handle this request : request2

## 示例与源码

其实责任链模式在实际项目过程中遇到的非常多，Android 和 Java 源码也一样，举几个简单的例子：

ViewGroup 和 View 中 touch 事件的分发，子 View 的 onTouchEvent 返回 true 代码消费该事件并不再传递，false 代表不消费并且传递到父 ViewGroup 去处理，这些树形结构的子 View 就是责任链上一个个处理对象；

OrderedBroadcast，有序广播的每一个接收者按照优先级依次接受消息，如果处理完成之后可以调用 abortBroadcast 终止广播，不是自己处理的就可以传递给下一个处理者；

try-catch语句，每一个 catch 根据 Exception 类型进行匹配，形成一个责任链，如果有一个 catch 语句与该 Exception 符合，这个 Exception 就交由给它进行处理，之后所有 catch 语句都不会再次执行。

我们这就以 wiki 上的代码为例：每个人都有一个支出审批的额度，超过审批额度之后将会传递给下一个审批人，直到最后：

**请求审批类PurchaseRequest**

**public** **class** **PurchaseRequest** {

**private** **int** amount;

**private** **String** purpose;

**public** **PurchaseRequest**(**int** amount, **String** purpose) {

**this**.amount = amount;

**this**.purpose = purpose;

}

**public** **int getAmount**() {

**return** amount;

}

**public** **void** **setAmount**(**int** amt) {

amount = amt;

}

**public** **String** **getPurpose**() {

**return** purpose;

}

**public** **void** **setPurpose**(**String** reason) {

purpose = reason;

}

}

**Handler审批人抽象类**

**public** **abstract** **class** **PurchasePower** {

**protected** **static** **final** **int** **BASE** = 500;

**protected** **PurchasePower** successor;

**abstract** **protected** **int** **getAllowable**();

**abstract** **protected** **String** **getRole**();

**public** **void** **setSuccessor**(**PurchasePower** successor) {

**this**.successor = successor;

}

**public** **void** **processRequest**(**PurchaseRequest** request){

**if** (request.getAmount() < **this**.getAllowable()) {

**System**.**out**.println(**this**.getRole() + " will approve $" + request.getAmount());

} **else** **if** (successor != **null**) {

successor.processRequest(request);

}

}

}

**ManagerPPower.class**

public class ManagerPPower extends PurchasePower {

protected int getAllowable(){

return BASE\*10;

}

protected String getRole(){

return "Manager";

}

}

**DirectorPPower.class**

public class DirectorPPower extends PurchasePower {

protected int getAllowable(){

return BASE\*20;

}

protected String getRole(){

return "Director";

}

}

**VicePresidentPPower.class**

public class VicePresidentPPower extends PurchasePower {

protected int getAllowable(){

return BASE\*40;

}

protected String getRole(){

return "Vice President";

}

}

**PresidentPPower.class**

public class PresidentPPower extends PurchasePower {

protected int getAllowable(){

return BASE\*60;

}

protected String getRole(){

return "President";

}

}

**最后是客户端的测试代码：**

**PurchasePower** manager = **new** ManagerPPower();

**PurchasePower** director = **new** DirectorPPower();

**PurchasePower** vp = **new** VicePresidentPPower();

**PurchasePower** president = **new** PresidentPPower();

manager.setSuccessor(director);

director.setSuccessor(vp);

vp.setSuccessor(president);

**PurchaseRequest** request = **new** PurchaseRequest(15000, "General");

manager.processRequest(request);

**测试结果：**

Vice President will approve $15000

## 总结

责任链模式的优点显而易见，可以对请求者和处理者关系解耦，提高代码的灵活性，通过改变链内的成员或调动他们的次序，允许你动态地新增或者删除职责；但是责任链模式最大的缺点在于对链中请求处理者的遍历，如果处理者太多那么必定会影响性能，特别是在一些递归调用中，而且不容易观察运行时的特征，有碍于除错。

很多资料中会介绍纯和不纯的责任链模式，在标准的责任链模式中，责任链上的一个节点只允许有两个行为：处理或者推给下个节点处理，而不允许处理完之后又推给下个节点，前者被很多资料称为纯的责任链模式，而后者被称为不纯的责任链模式。其实在实际的系统里，纯的责任链很难找到。如果坚持责任链不纯便不是责任链模式，那么责任链模式便不会有太大意义了。

# 命令模式（Command）

## 概述

命令模式（Command Pattern），它是行为型设计模式之一。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，那么所有的这些被封装的方法就会被挨个执行调用。

## 特点

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。

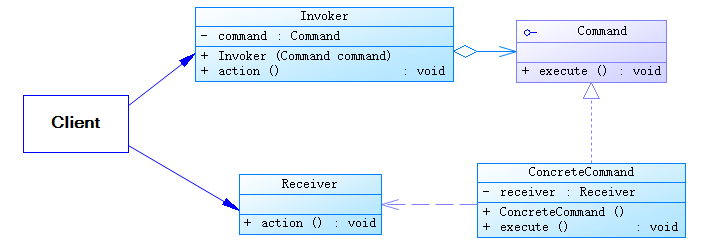
**命令模式的使用场景：**

* 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；
* 系统需要将一组操作组合在一起，即支持宏命令。

**wiki 上列出的具体使用场景：**

* GUI buttons and menu items
* 在 Java Swing 和 Delphi 语言中，一个 Action 是一个命令，除了执行预定的命令之外，一个 Action 可能会有一个相关联的图标，键盘快捷键，气泡提示文本等等。一个工具栏按钮或者菜单的元素可能完全用一个 Action 对象进行初始化。
* Macro recording
* 如果所有的用户动作都代表了一个个命令对象，那么一个程序就能够轻易的保存一系列的命令对象，之后能够将这一系列的命令重新执行一遍来实现一个回播的动作。如果程序中嵌入了脚本引擎，那么每个命令对象都能够实现 toScript() 方法，用户的动作也能够被轻松的保存为脚本对象。
* Mobile Code
* 类似于 Java 这种能够通过 URLClassloders 将代码变成流从一个地方传输到另一个地方的语言，并且代码库中的命令使新的行为能够被传递到远程位置（EJB Command，Master Worker模式）。
* Multi-level undo
* 如果程序中所有的用户动作都被实现成了命令对象，程序就能够保存最近被执行的命令对象，当用户想要撤销某些命令时，程序就可以简单的 pop 出最近的命令并且执行他的 undo 方法。
* Networking
* 能够将所有的命令对象通过网络传输到另外一台设备上去执行，比如端游中的用户操作等。
* Parallel Processing
* 所有的命令都被写成了共享资源中的任务并且并发的被很多线程同时执行（在远程机器上这种变体可能这个会被称为 Master/Worker 模式）。
* Progress bars
* 我们假定程序有一系列需要按顺序执行的命令，如果每个命令对象都有一个 getEstimatedDuration() 方法，那么程序就可以轻松估算出整体的执行时间，然后展示一个有意义的进度条来反应当前所有任务的执行程度。
* Thread pools
* 一个具有代表性的线程池可能会有一个 public 的 addTask() 方法用来添加一个工作任务到内部的等待队列中，这个线程池中会有一系列的线程用来执行队列中的一系列命令对象。普遍的，这些对象会实现一个通用的接口，比如 Runnable 等，用来允许线程池执行这些命令，虽然这个线程池类并不了解它会被用来处理具体的什么任务。
* Transactional behavior
* 类似于 undo 操作，一个数据库引擎或者软件安装器可能会维护一个已经执行或者将要执行的操作列表，如果其中的一个失败了，所有其他的都会被还原或者抛弃（通常被称为 rollback，回滚）。举个例子，如果相关联的两个数据库表必须要更新，并且第二个更新失败，这个事务将会被回滚，所以第一个表的更新也会被抛弃。
* Wizards
* 向导页是用户在点击最后一页”结束”按钮时候弹出来的几页配置页（配置用户的使用习惯等），在这种情况下，一个通常分离用户操作代码和程序代码的方法就是使用命令对象实现向导功能。这个命令对象会在向导页第一次展示的时候被创建，每个向导页将它们自己的 GUI 变化保存在一个命令对象中，所以这个对象被定位为用户的进一步操作。“结束”动作简单的触发了一个 excete() 动作，这样一来，这个命令类将会达到预期的效果。

## UML类图



Receiver：接收者角色

该类负责具体实施或执行一个请求，说的通俗一点就是，执行具体逻辑的角色，以上面说到的“关机”操作命令为例，其接收者角色就是真正执行各项关机逻辑的底层代码。任何一个类都能成为一个接收者，而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 ：命令接口

定义所有具体命令类基本行为的抽象接口；

ConreteCommand：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，如上面提到的“关机”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，如果将这一些列的具体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，“关机”命令为例，关机这个命令一般就对应着一个关机方法，执行关机命令就相当于由这个关机方法去执行具体的逻辑，这个关机方法就可以看作是请求者；

Client：客户端角色

## 命令模式的通用代码：

**Receiver.class 具体逻辑执行者**

**public** **class** **Receiver** {

**public** **void** **action**() {

**System**.**out**.print("执行具体的操作");

}

}

**Command.class 抽象命令类**

**public** **interface** Command {

**void** **execute**();

}

**ConcreteCommand.clas 具体命令类**

**public** **class** **ConcreteCommand** **implements** Command {

**private** **Receiver** receiver;

**public** **ConcreteCommand**(**Receiver** receiver) {

**this**.receiver = receiver;

}

@Override

**public** **void** **execute**() {

receiver.action();

}

}

**Invoker.class 请求者**

**public** **class** **Invoker** {

**private** Command command;

**public** **Invoker**(Command command) {

**this**.command = command;

}

**public** **void** **action**() {

command.execute();

}

}

**Client 客户端**

**Receiver** receiver = **new** Receiver();

Command command = **new** ConcreteCommand(receiver);

**Invoker** invoker = **new** Invoker(command);

invoker.action();

## 示例与源码

这就以一个简单的控制电灯亮灭和门开关的情景为例：

**Light.class 和 Door.class 具体逻辑执行类**

**public** **class** **Light** {

**public** **void** **lightOn**() {

**System**.**out**.print("light on\n");

}

**public** **void** **lightOff**() {

**System**.**out**.print("light off\n");

}

}

**public** **class** **Door** {

**public** **void** **doorOpen**() {

**System**.**out**.print("door open\n");

}

**public** **void** **doorClose**() {

**System**.**out**.print("door close\n");

}

}

**Command.class 抽象命令类**

**public** **interface** Command {

**void** **execute**();

}

**然后是电灯命令类：**

**LightOnCommand.class 和 LightOffCommand.class**

**public** **class** **LightOnCommand** **implements** Command{

**public** **Light** light;

**public** **LightOnCommand**(**Light** light) {

**this**.light = light;

}

@Override

**public** **void** **execute**() {

light.lightOn();

}

}

**public** **class** **LightOffCommand** **implements** Command{

**public** **Light** light;

**public** **LightOffCommand**(**Light** light) {

**this**.light = light;

}

@Override

**public** **void** **execute**() {

light.lightOff();

}

}

**门的相关命令类：**

**DoorOpenCommand.class 和 DoorCloseCommand.class**

**public** **class** **DoorOpenCommand** **implements** Command{

**public** **Door** door;

**public** **DoorOpenCommand**(**Door** door) {

**this**.door = door;

}

@Override

**public** **void** **execute**() {

door.doorOpen();

}

}

**public** **class** **DoorCloseCommand** **implements** Command{

**public** **Door** door;

**public** **DoorCloseCommand**(**Door** door) {

**this**.door = door;

}

@Override

**public** **void** **execute**() {

door.doorClose();

}

}

**然后是一个无操作默认命令类：**

**NoCommand.class**

**public** **class** **NoCommand** **implements** Command{

@Override

**public** **void** **execute**() {

}

}

**最后是请求类：**

**Controller.class**

**public** **class** **Controller** {

**private** Command[] onCommands;

**private** Command[] offCommands;

**public** **Controller**() {

onCommands = **new** Command[2];

offCommands = **new** Command[2];

Command noCommand = **new** NoCommand();//以免未赋值时报空指针异常

onCommands[0] = noCommand;

onCommands[1] = noCommand;

offCommands[0] = noCommand;

offCommands[1] = noCommand;

}

**public** **void** **setCommand**(**int** slot, Command onCommand, Command offCommand) {

onCommands[slot] = onCommand;

offCommands[slot] = offCommand;

}

**public** **void** **onCommand**(**int** slot) {

onCommands[slot].execute();

}

**public** **void** **offCommand**(**int** slot) {

offCommands[slot].execute();

}

}

**测试代码**

**Light** light = **new** Light();

**Door** door = **new** Door();

**LightOnCommand** lightOnCommand = **new** LightOnCommand(light);

**LightOffCommand** lightOffCommand = **new** LightOffCommand(light);

**DoorOpenCommand** doorOpenCommand = **new** DoorOpenCommand(door);

**DoorCloseCommand** doorCloseCommand = **new** DoorCloseCommand(door);

**Controller** controller = **new** Controller();

controller.setCommand(0, lightOnCommand, lightOffCommand);//0号位置的开关

controller.setCommand(1, doorOpenCommand, doorCloseCommand);//1号位置的开关

controller.onCommand(0);

controller.offCommand(0);

controller.onCommand(1);

controller.offCommand(1);

**结果：**

light on

light off

door open

door close

这样就实现了对 Light 和 Door 的控制了，其实这个例子只是实现了对命令模式的基本框架而已，命令模式的用处其实在于它的日志和回滚撤销功能，每次执行命令的时候都打印出相应的关键日志，或者每次执行后都将这个命令保存进列表中并且每个命令实现一个 undo 方法，以便可以进行回滚。另外，也可以构造一个 MacroCommand 宏命令类用来按次序先后执行几条相关联命令。当然可以发散的空间很多很多，感兴趣的可以自己去实现，原理都是一样的。

## 总结

命令模式将发出请求的对象和执行请求的对象解耦，被解耦的两者之间通过命令对象进行沟通，调用者通过命令对象的 execute 放出请求，这会使得接收者的动作被调用，调用者可以接受命令当作参数，甚至在运行时动态地进行，命令可以支持撤销，做法是实现一个 undo 方法来回到 execute 被执行前的状态。MacroCommand 宏命令类是命令的一种简单的延伸，允许调用多个命令，宏方法也可以支持撤销。日志系统和事务系统可以用命令模式来实现。

　　命令模式的优点很明显，调用者和执行者之间的解耦，更灵活的控制性，以及更好的扩展性等。缺点更明显，就是类的爆炸，大量衍生类的创建，这也是大部分设计模式的“通病”，是一个没有办法避免的问题。

# 策略模式（Strategy）

## 概述

策略模式（Strategy Pattern，或者叫 Policy Pattern），也是行为型模式之一。通常在软件开发中，我们为了一个功能可能会设计多种算法和策略，然后根据实际使用情况动态选择对应的算法和策略，比如排序算法中的快速排序，冒泡排序等等，根据时间和空间的综合考虑进行运行时选择。

针对这种情况，一个常规的方法是将多种算法写在一个类中，每一个方法对应一个具体的排序算法，或者写在一个方法中，通过 if…else 或者 switch…case 条件来选择具体的排序算法。这两种方法我们都成为硬编码，虽然很简单，但是随着算法数量的增加，这个类就会变得越来越臃肿，维护的成本就会变高，修改一处容易导致其他地方的错误，增加一种算法就需要修改封装算法类的源代码，即违背了开闭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，这就是下面讲到的策略模式。

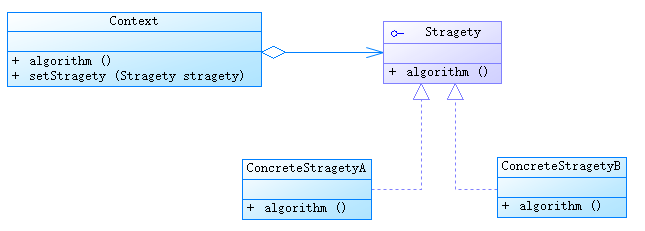
## 特点

策略模式定义了一系列的算法，并将每一个算法封装起来，而且使他们可以相互替换，让算法独立于使用它的客户而独立变化。

　　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。

## UML图



* Strategy（抽象策略类）： 通常由一个接口或者抽象类实现。
* ConcreteStrategy（具体策略角色）：包装了相关的算法和行为。一般有多个
* Context（环境角色）：持有一个策略类的引用，最终给客户端调用。

## 策略模式的概念

The Strategy Pattern defines a family of algorithms,encapsulates each one,and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

翻译：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。策略模式使得算法可独立于使用它的客户而变化。

* 需要使用ConcreteStrategy提供的算法。
* 内部维护一个Strategy的实例。
* 负责动态设置运行时Strategy具体的实现算法。
* 负责跟Strategy之间的交互和数据传递。

## 策略模式的应用场景(Volley)

定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。策略模式使得算法可独立于使用它的客户而变化。实际开发中用途，比如google开源的android网络框架volley，volley的超时重发重发机制就使用到了典型策略模式，看源码

### RetryPolicy.java 请求重试策略类，定义了三个接口

//请求重试策略类 Strategy

**public** **interface** RetryPolicy {

//超时时间

**public** **int** **getCurrentTimeout**();

//重试次数

**public** **int** **getCurrentRetryCount**();

//针对错误异常的处理

**public** **void** **retry**(**VolleyError** error) **throws** VolleyError;

}

### DefaultRetryPolicy.java 默认重试策略，继承了RetryPolicy ，具体实现了接口

**public** **class** **DefaultRetryPolicy** **implements** **RetryPolicy** {

@Override

**public** **int** **getCurrentTimeout**() {

**return** mCurrentTimeoutMs;

}

@Override

**public** **int** **getCurrentRetryCount**() {

**return** mCurrentRetryCount;

}

@Override

**public** **void** **retry**(**VolleyError** error) **throws** VolleyError {

mCurrentRetryCount++;

mCurrentTimeoutMs += (mCurrentTimeoutMs \* mBackoffMultiplier);

**if** (!hasAttemptRemaining()) {

**throw** error;

}

}

}

### Request.java 请求类，持有一个抽象策略类RetryPolicy 的引用，最终给客户端调用

**public** **abstract** **class** **Request**<T> **implements** Comparable<**Request**<T>> {

**private** **RetryPolicy** mRetryPolicy; //持有策略类的引用

**public** **Request**(**int** method, **String** url, **Response**.**ErrorListener** listener) {

setRetryPolicy(**new** **DefaultRetryPolicy**()); //设置策略类

}

**public** **Request**<?> **setRetryPolicy**(**RetryPolicy** retryPolicy) {

mRetryPolicy = retryPolicy;//设置策略类

**return** **this**;

}

**public** **final** **int** **getTimeoutMs**() {

**return** mRetryPolicy.getCurrentTimeout();

}

}

### BasicNetwork.java 网络处理类，处理Request，使用到了对应的重试策略

/\*\*

\*遇到异常请求使用到了策略类的getTimeoutMs()方法

\*/

**private** **static** **void** attemptRetryOnException(String logPrefix, Request<?> request, VolleyError exception) **throws** VolleyError {

RetryPolicy retryPolicy = request.getRetryPolicy();

**int** oldTimeout = request.getTimeoutMs();

**try** {

retryPolicy.retry(exception);

} **catch** (VolleyError e) {

}

}

/\*\*

\* 网速慢，获取重试策略的重试次数getCurrentRetryCount

\*/

**private** **void** logSlowRequests(**long** requestLifetime, Request<?> request,

**byte**[] responseContents, StatusLine statusLine) {

**if** (DEBUG || requestLifetime > SLOW\_REQUEST\_THRESHOLD\_MS) {

VolleyLog.d("HTTP response for request=<%s> [lifetime=%d], [size=%s], " +

"[rc=%d], [retryCount=%s]", request, requestLifetime,

responseContents != **null** ? responseContents.length : "null",

statusLine.getStatusCode(), request.getRetryPolicy().getCurrentRetryCount());

}

}

## 自定义简单的策略模式实例

商品实现针对不同等级会员显示对应的会员价，比如一本书原价100元，一级会员打97折，三级会员打8折

**PriceStrategy.java，策略类，可以使抽象类或接口，定义算法的公共接口**

/\*\*

\* 定义一个计算价格的接口

\* 它属于抽象策略类

\*/

**public** **interface** PriceStrategy {

**double** **priceStrategyInterface**(**double** price);

}

**SuperVipStrategy.java，超级会员算法，打八折，继承PriceStrategy**

/\*\*

\* 超级会员类

\* 它属于具体策略类，继承了计算价格接口

\*/

**public** **class** **SuperVipStrategy** **implements** PriceStrategy {

@Override

**public** **double** **priceStrategyInterface**(**double** price) {

**return** price \* 0.8; //打八折

}

}

**OneVipStrategy.java，一级会员算法，打九七折，继承PriceStrategy**

/\*\*

\* 一级会员类

\* 它属于具体策略类

\*/

**public** **class** **OneVipStrategy** **implements** PriceStrategy {

@Override

**public** **double** **priceStrategyInterface**(**double** price) {

**return** 0.97 \* price; //打九七折

}

}

**Price.java 上下文环境，持有PriceStrategy 的引用。**

/\*\*

\* 环境角色

\*/

**public** **class** **Price** {

**private** PriceStrategy priceStrategy;

**public** **Price**(PriceStrategy priceStrategy){

**this**.priceStrategy = priceStrategy;

}

**public** **double** **getVipPrice**(**double** price){

**return** priceStrategy.priceStrategyInterface(price);

}

}

**测试实例**

**SuperVipStrategy** supVipSrategy = **new** SuperVipStrategy();

**OneVipStrategy** onVipStrategy = **new** OneVipStrategy();

**System**.**out**.print("超级会员价="+ **new** Price(supVipSrategy).getVipPrice(100));

**System**.**out**.print("一级员价="+ **new** Price(onVipStrategy).getVipPrice(100));

**优点**

1、 提供了一种替代继承的方法，而且既保持了继承的优点(代码重用)，还比继承更灵活(算法独立，可以任意扩展)；

2、 避免程序中使用多重条件转移语句，使系统更灵活，并易于扩展；

3、 遵守大部分GRASP原则和常用设计原则，高内聚、低偶合；

4、 易于进行单元测试，各个算法区分开，可以针对每个算法进行单元测试；

**缺点**

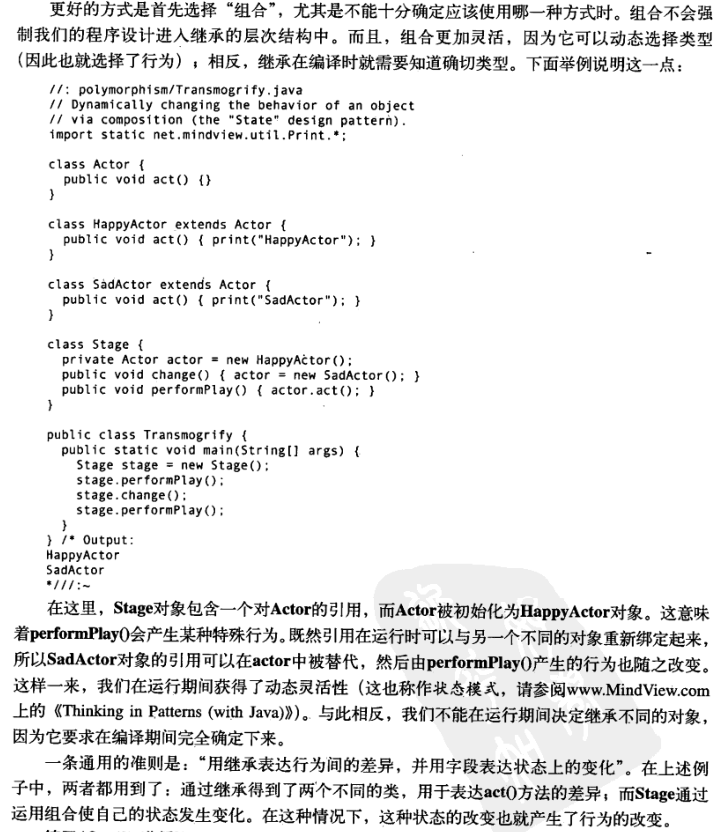
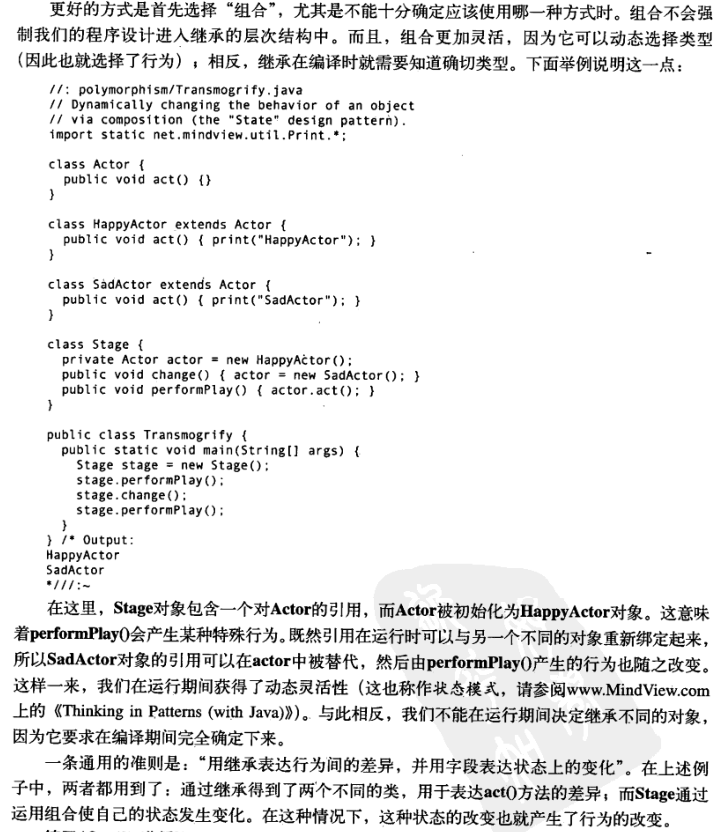
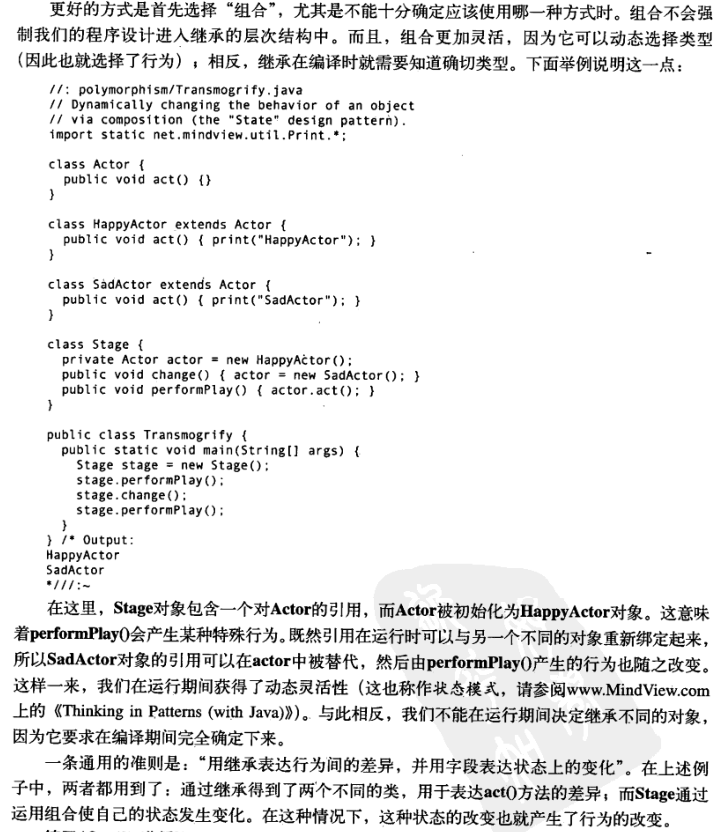
1、 因为每个具体策略类都会产生一个新类，所以会增加系统需要维护的类的数量；

2、 选择何种算法需要客户端来创建对象，增加了耦合，这里可以通过与工厂模式结合解决该问题；

3、 程序复杂化。

# 状态模式（State）

## Think in Java Demo



## 概述

状态模式（State Pattern），也是行为型设计模式之一。状态模式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类图几乎完全一样，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。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；而策略模式可以想象成是除了继承之外的一种弹性替代方案，如果你使用继承定义了一个类的行为，你将被这个行为困住，甚至要修改它都很难，有了策略模式，你可以通过组合不同的对象来改变行为。状态模式的意图是让一个对象在其内部状态发生改变的时候，其行为也随之改变。

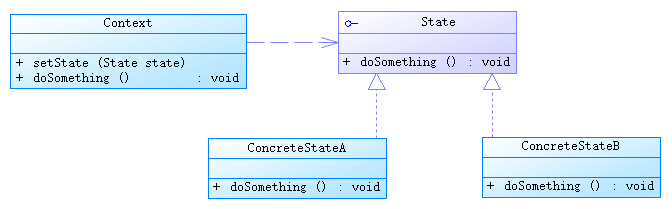
## 特点

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。

状态模式的使用场景：

* 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；
* 代码中包含大量与状态有关的条件语句，例如，一个操作中含有庞大的多分枝语句（if-else 或者 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。

## UML类图



* Context：环境类，定义客户感兴趣的接口，维护一个 State 子类，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的 State 中定义的接口，从而达到不同状态下的不同行为。

## 状态模式的通用代码：

**状态接口以及相关子类：**

**State.class**

**public** **interface** State {

**void** **doSomething**();

}

**ConcreteStateA.class**

**public** **class** **ConcreteStateA** **implements** State {

@Override

**public** **void** **doSomething**() {

**System**.**out**.print("this is ConcreteStateA's function\n");

}

}

**ConcreteStateB.class**

**public** **class** **ConcreteStateB** **implements** State{

@Override

**public** **void** **doSomething**() {

**System**.**out**.print("this is ConcreteStateB's function\n");

}

}

**NullState.class**

**public** **class** **NullState** **implements** State{

@Override

**public** **void** **doSomething**() {

//do nothing

}

}

**Context类以及测试代码：**

**Context.class**

**public** **class** **Context** {

**private** State state = **new** NullState();

**void** **setState**(State state) {

**this**.state = state;

}

**void** **doSomething**() {

state.doSomething();

}

**public** **static** **void** **main**(**String**[] args) {

**Context** context = **new** Context();

context.setState(**new** ConcreteStateA());

context.doSomething();

context.setState(**new** ConcreteStateB());

context.doSomething();

}

}

**最后结果：**

this is ConcreteStateA's function

this is ConcreteStateB's function

## wiki 上的 demo ，实现一堆字符串的一个大小写间隔打印：

**Statelike.class**

**interface** Statelike {

**void** **writeName**(**StateContext** context, **String** name);

}

**StateLowerCase.class**

**class** **StateLowerCase** **implements** Statelike {

@Override

**public** **void** **writeName**(**final** **StateContext** context, **final** **String** name) {

**System**.**out**.println(name.toLowerCase());

context.setState(**new** **StateMultipleUpperCase**());

}

}

**StateMultipleUpperCase.class**

**class** **StateMultipleUpperCase** **implements** Statelike {

/\*\* Counter local to this state \*/

**private** **int** count = 0;

@Override

**public** **void** **writeName**(**final** **StateContext** context, **final** **String** name) {

**System**.**out**.println(name.toUpperCase());

/\* Change state after StateMultipleUpperCase's writeName() gets invoked twice \*/

**if**(++count > 1) {

context.setState(**new** StateLowerCase());

}

}

}

**StateContext.class**

**class** **StateContext** {

**private** Statelike myState;

**StateContext**() {

setState(**new** StateLowerCase());

}

**void** **setState**(**final** Statelike newState) {

myState = newState;

}

**public** **void** **writeName**(**final** **String** name) {

myState.writeName(**this**, name);

}

**public** **static** **void** **main**(**String**[] args) {

**final** **StateContext** sc = **new** StateContext();

sc.writeName("Monday");

sc.writeName("Tuesday");

sc.writeName("Wednesday");

sc.writeName("Thursday");

sc.writeName("Friday");

sc.writeName("Saturday");

sc.writeName("Sunday");

}

}

**最后结果：**

monday

TUESDAY

WEDNESDAY

Thursday

FRIDAY

SATURDAY

sunday

## 总结

状态模式的关键点在于不同的状态下对于统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实例。在 if-else 或者 switch-case 形势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，易于出错不易维护，通过状态模式能够很好的消除这类“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。和程序状态机（PSM）不同，状态模式用类代表状态，状态的转换可以由 State 类或者 Context 类控制。

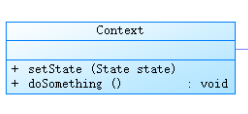
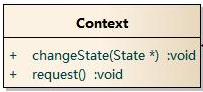
### 优点

* 通过将每个状态封装进一个类，将以后所做的修改局部化；
* 将所有与一个特定状态相关的行为封装到一个对象中，繁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增加可维护性和可扩展性。

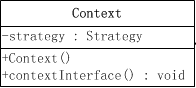
### 缺点

* 当然也很明显，也是绝大部分设计模式的通病，类数目的增多。

# 状态模式VS策略模式

* 状态模式最经典的应该还是在于行为改变状态，状态模式在context中可以通过一系列方法改变context中的state的类型
* 可以通过set方法啊，change或者state内部的方法改变自己的类型



策略模式里面，可以清晰的看见，采用那种算法，哪么就让 context实现那种算法，其决定权在于我们,在于客服端，这个 context本身不能决定，算法对象本身不能决定，因为他是死的

只能通过构造函数，或者new 新的state，而不能通过context中的set之类的方法改变他的状态，总之，每次都是new

# 中介者模式（Mediator）

## 概述

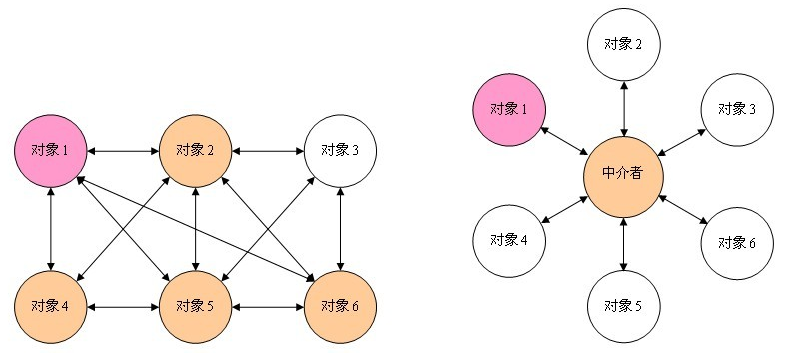
中介者模式（Mediator Pattern），也是行为型模式之一，中介者模式也称为调解者模式或者调停者模式，顾名思义，它的作用是在若干类或者若干模块之间承当中介。

通常情况下，一个程序必然要包含大量的类，随着项目的进行，类和模块的数量必然要进一步增加，特别是当需要维护或者重构时，类与类之间复杂的网状结构会让事情变得越来越复杂，降低程序的可读性和可维护性，并且修改一个类需要牵涉到其他类，不符合开闭原则。所以此时中介者模式能够将这些网状结构的类变成星型依赖，所以类都只依赖于中介者，不直接依赖于其他类。

## 特点

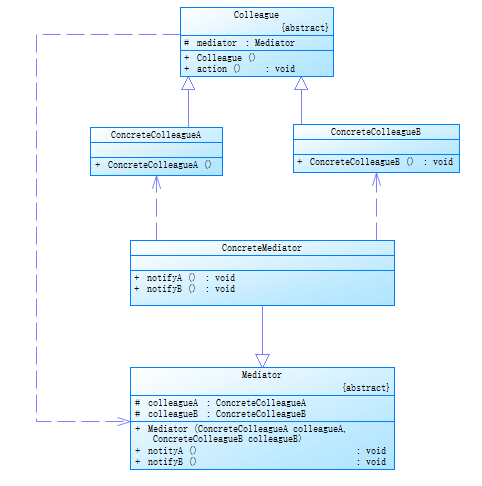
中介者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用，从而使耦合松散，而且可以独立地改变它们之间的交互。

中介者模式可以将多对多的网状结构转换成一对多的星型结构，达到降低系统的复杂性，提高可扩展性的作用。在没有中介者角色之前，所有对象都需要依赖其他对象，持有他们的引用，也就是说对象之间是紧耦合的，有了中介者之后，一切就简单了，每个对象都会在自己状态改变时，告诉中介者，每个对象都会对中介者发出的请求做出回应。



所以中介者模式适合的场景就很明确了：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，来解决紧耦合的问题。

## UML类图



* Mediator：抽象中介者角色，定义了同事对象到中介者对象的接口，可以通过抽象类或者接口的方式实现；
* ConcreteMediator：具体中介者角色，继承或者实现了抽象中介者，实现了父类定义的方法，它从具体的同事对象接受消息，向具体同事对象发出命令；
* Colleague：抽象同事类角色，定义了同事对象的接口，它只知道中介者而不知道其他的同事对象；
* ConcreteColleague：具体同事类角色，继承抽象同事类，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。

## 中介者模式的通用代码

**抽象中介者角色：**

**Mediator.class**

**public** **abstract** **class** **Mediator** {

**protected** **Colleague** colleagueA;

**protected** **Colleague** colleagueB;

**public** **Mediator**(**Colleague** colleagueA, **Colleague** colleagueB) {

**this**.colleagueA = colleagueA;

**this**.colleagueB = colleagueB;

}

**public** **abstract** **void** **notifyColleagueA**();

**public** **abstract** **void** **notifyColleagueB**();

}

**具体中介者角色：**

**ConcreteMediator.class**

**public** **class** **ConcreteMediator** **extends** **Mediator**{

**public** **ConcreteMediator**(**Colleague** colleagueA, **Colleague** colleagueB) {

**super**(colleagueA, colleagueB);

}

@Override

**public** **void** **notifyColleagueA**() {

**if** (colleagueA != **null**) {

colleagueA.operation();

}

}

@Override

**public** **void** **notifyColleagueB**() {

**if** (colleagueB != **null**) {

colleagueB.operation();

}

}

}

**抽象同事类：**

**Colleague.class**

**public** **abstract** **class** **Colleague** {

**protected** **Mediator** mediator;

**public** **void** **setMediator**(**Mediator** mediator) {

**this**.mediator = mediator;

}

**public** **abstract** **void** **operation**();

}

**具体同事类：**

**ConcreteColleagueA.class**

**public** **class** **ConcreteColleagueA** **extends** **Colleague**{

**public** **void** **notifyColleagueB**() {

mediator.notifyColleagueB();

}

@Override

**public** **void** **operation**() {

**System**.**out**.print("this is ConcreteColleagueA's operation\n");

}

}

**ConcreteColleagueB.class**

**public** **class** **ConcreteColleagueB** **extends** **Colleague**{

**public** **void** **notifyColleagueA**() {

mediator.notifyColleagueA();

}

@Override

**public** **void** **operation**() {

**System**.**out**.print("this is ConcreteColleagueB's operation\n");

}

}

**测是代码：**

**Colleague** colleagueA = **new** ConcreteColleagueA();

**Colleague** colleagueB = **new** ConcreteColleagueB();

**Mediator** mediator = **new** ConcreteMediator(colleagueA, colleagueB);

colleagueA.setMediator(mediator);

colleagueB.setMediator(mediator);

((**ConcreteColleagueA**)colleagueA).notifyColleagueB();

((**ConcreteColleagueB**)colleagueB).notifyColleagueA();

**最后结果：**

this is ConcreteColleagueB's operation

this is ConcreteColleagueA's operation

两个 Colleague 类成功通过 Mediator 进行了相互作用。上面这个是中介者模式的标准写法，就我自己在项目中实际使用中介者模式来说，有时候将同事子类抽象出一个 Colleague 父类是不太合理的，因为子类之间的业务逻辑的不同，导致他们很难抽象出一些公用方法，所以这时候使用中介者模式，可以省去 Colleague 这个角色，让 Mediator 直接依赖于几个同事子类；同时也可以不定义Mediator接口，把具体的中介者对象实现成为单例，这样同事对象不再持有中介者，而是在需要的时候直接获取中介者对象并调用；中介者也不再持有同事对象，而是在具体处理方法里面去创建，或获取，或从数据传入需要的同事对象。

## 示例与源码

在 Android 源码中也有很多使用中介者模式的例子，比如最突出的就是 Binder 中的 Binder Driver 这个角色，它连接了 Binder client ， Binder server 和 ServiceManager，相当于一个中介者，即AIDL。这里仍然以 wiki 的 demo 为例，使用 Mediator 来控制 3 个按钮实现 book、view 和 search 的功能：

**同事类角色：**

**Command.class**

interface Command {

void execute();

}

**BtnView.class**

class BtnView extends JButton implements Command {

Mediator med;

BtnView(ActionListener al, Mediator m) {

super("View");

addActionListener(al);

med = m;

med.registerView(this);

}

public void execute() {

med.view();

}

}

**BtnSearch.class**

class BtnSearch extends JButton implements Command {

Mediator med;

BtnSearch(ActionListener al, Mediator m) {

super("Search");

addActionListener(al);

med = m;

med.registerSearch(this);

}

public void execute() {

med.search();

}

}

**BtnBook.class**

class BtnBook extends JButton implements Command {

Mediator med;

BtnBook(ActionListener al, Mediator m) {

super("Book");

addActionListener(al);

med = m;

med.registerBook(this);

}

public void execute() {

med.book();

}

}

**LblDisplay.class**

class LblDisplay extends JLabel {

Mediator med;

LblDisplay(Mediator m) {

super("Just start...");

med = m;

med.registerDisplay(this);

setFont(new Font("Arial", Font.BOLD, 24));

}

}

**中介者角色：**

**Mediator.class**

interface Mediator {

void book();

void view();

void search();

void registerView(BtnView v);

void registerSearch(BtnSearch s);

void registerBook(BtnBook b);

void registerDisplay(LblDisplay d);

}

class ParticipantMediator implements Mediator {

BtnView btnView;

BtnSearch btnSearch;

BtnBook btnBook;

LblDisplay show;

//....

public void registerView(BtnView v) {

btnView = v;

}

public void registerSearch(BtnSearch s) {

btnSearch = s;

}

public void registerBook(BtnBook b) {

btnBook = b;

}

public void registerDisplay(LblDisplay d) {

show = d;

}

public void book() {

btnBook.setEnabled(false);

btnView.setEnabled(true);

btnSearch.setEnabled(true);

show.setText("booking...");

}

public void view() {

btnView.setEnabled(false);

btnSearch.setEnabled(true);

btnBook.setEnabled(true);

show.setText("viewing...");

}

public void search() {

btnSearch.setEnabled(false);

btnView.setEnabled(true);

btnBook.setEnabled(true);

show.setText("searching...");

}

}

**最后的测试程序：**

class MediatorDemo extends JFrame implements ActionListener {

Mediator med = new ParticipantMediator();

MediatorDemo() {

JPanel p = new JPanel();

p.add(new BtnView(this, med));

p.add(new BtnBook(this, med));

p.add(new BtnSearch(this, med));

getContentPane().add(new LblDisplay(med), "North");

getContentPane().add(p, "South");

setSize(400, 200);

setVisible(true);

}

public void actionPerformed(ActionEvent ae) {

Command comd = (Command) ae.getSource();

comd.execute();

}

public static void main(String[] args) {

new MediatorDemo();

}

}

## 总结

在面向对象的变成语言里，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，如果几个类之间的关系并不复杂，耦合也很少，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需要多多考虑，权衡利弊。

### 优点：

* 适当的使用中介者模式可以避免同事类之间的过度耦合，使得各同事类之间可以相对独立地使用；
* 使用中介者模式可以将对象间一对多的关联转变为一对一的关联，使对象间的关系易于理解和维护；
* 使用中介者模式可以将对象的行为和协作进行抽象，能够比较灵活的处理对象间的相互作用。

### 缺点：

* 使用中介者模式需要权衡一下，不能因为同事类少就不适合使用中介者模式，也不能因为同事类多就一定要使用中介者模式，重要的是解耦合，就算是三个类，他们直接的耦合很严密，导致一个类的修改会严重影响到另外两个类，这时候就可以考虑使用中介者，另一方面，类如果很多但是相互都是简单的连接，耦合性低，使用中介者模式就显得不是那么必要了；
* 随着同事子类的增多和类之间关系的复杂化，中介者会变得越来越庞大，减少可维护性。

# 迭代器模式（Iterator、Cursor）

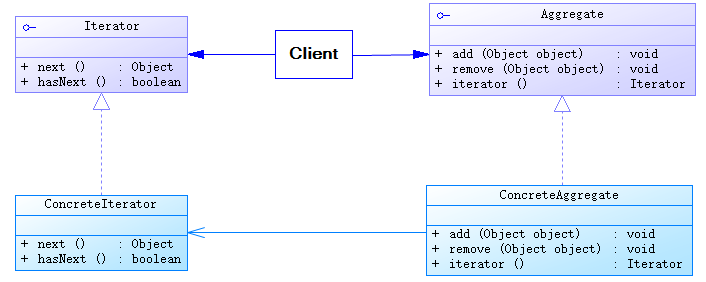
## 概述

迭代器模式（Iterator Pattern），又称为游标（Cursor Pattern）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一个第三者——迭代器，很好地解决了上面所述的弊端。

## 特点

* 提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。
* 迭代器模式让我们能游走于聚合内的每一个元素，而又不暴露其内部的表示，把游走的任务放在迭代器上，而不是聚合上。这样简化了聚合的接口和实现，也让责任各得其所。
* 迭代器模式使用的场景也很简单：遍历一个容器对象时。

## UML类图



迭代器模式角色：

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。

ConcreteIterator：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。

ConcreteAggregate：具体容器类

具体迭代器角色与该容器相关联。

Client：客户类

## 迭代器模式的通用代码：

**迭代器类：**

**Iterator.class**

**public** **interface** Iterator<T> {

**boolean** **hasNext**();

T **next**();

}

**ConcreteIterator.class**

**public** **class** **ConcreteIterator**<T> **implements** Iterator<T>{

**private** List<T> list;

**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;

}

}

**容器类：**

**Aggregation.class**

**public** **interface** Aggregation<T> {

**void** **add**(T obj);

**void** **remove**(T obj);

Iterator<T> **iterator**();

}

**ConcreteAggregation.class**

**public** **class** **ConcreteAggregation**<T> **implements** Aggregation<T>{

**private** List<T> list = **new** ArrayList<>();

@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<>(list);

}

}

**客户端代码**

Aggregation<**String**> a = **new** ConcreteAggregation<>();

a.add("a");

a.add("b");

a.add("c");

Iterator<**String**> iterator = a.iterator();

**while** (iterator.hasNext()) {

**System**.**out**.print(iterator.next());

}

**测试结果**

Abc

## 示例与源码

迭代器这个模式对于很多开发者来说几乎不会自己去实现一个迭代器，但是我们平时使用的频率不会低，在 Android 中，除了各种数据结构体，如 List、Map 等所包含的迭代器外，数据库查询的 Cursor 也是一个迭代器。我们这里就简单分析一下 ArrayList 的 Iterator 源码：

**Iterator.class**

public interface Iterator<E> {

boolean hasNext();

E next();

void remove();

}

**ArrayList.Itr.class**

private class Itr implements Iterator<E> {

int cursor; // index of next element to return

int lastRet = -1; // index of last element returned; -1 if no such

int expectedModCount = modCount;

public boolean hasNext() {

return cursor != size;

}

@SuppressWarnings("unchecked")

public E next() {

checkForComodification();

int i = cursor;

if (i >= size)

throw new NoSuchElementException();

Object[] elementData = ArrayList.this.elementData;

if (i >= elementData.length)

throw new ConcurrentModificationException();

cursor = i + 1;

return (E) elementData[lastRet = i];

}

public void remove() {

if (lastRet < 0)

throw new IllegalStateException();

checkForComodification();

try {

ArrayList.this.remove(lastRet);

cursor = lastRet;

lastRet = -1;

expectedModCount = modCount;

} catch (IndexOutOfBoundsException ex) {

throw new ConcurrentModificationException();

}

}

final void checkForComodification() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

}

}

这就是 ArrayList 迭代器的具体实现，从源码中我们可以看到有一个 checkForComodification() 函数，抛出的异常 ConcurrentModificationException 应该很多人认识，如果modCount 不等于expectedModCount，则抛出 ConcurrentModificationException 异常，一般情况下出现在遍历的同时调用了 ArrayList.remove 等操作对数据集合进行了更改，例如多线程中当一个线程删除了元素，由于 modCount 是 AbstarctList 的成员变量，因此可能会导致在其他线程中modCount 和 expectedModCount 值不等。

## 总结

　　对于迭代模式来说，其自身优点很明显：

* 迭代模式简化了聚集的接口，迭代子具备了一个遍历接口，这样聚集的接口就不必具备遍历接口；
* 每一个聚集对象都可以有一个或多个迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。因此，一个聚集对象可以同时有几个迭代在进行之中；
* 由于遍历算法被封装在迭代子角色里面，因此迭代的算法可以独立于聚集角色变化。

而其缺点就是对类文件的增加。其实迭代模式发展至今，几乎每一种高级语言都有相应的内置实现，对于开发者来说，已经很少会去由自己来实现迭代器了，因此，对于迭代器模式我们更多地是在于了解而非应用。

# 模板方法（Template Method）



## 概述

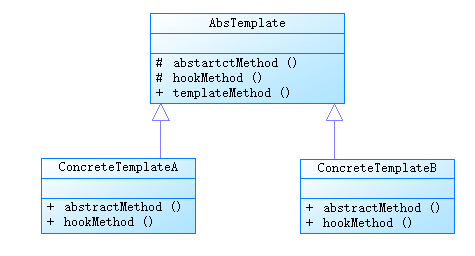
板方法模式（Template Method Pattern），也是行为型设计模式之一。在面向对象开发过程中，通常会遇到这样的一个问题，我们知道一个算法所需的关键步骤，并确定了这些步骤的执行顺序，但是，某些步骤的具体实现是未知的，或者说某些步骤的实现是会随着环境的变化而改变的，这时候我们就可以创建一个算法的模板，将算法定义成一组步骤，其中的任何步骤都可以是抽象的，由子类负责实现，这可以确保算法的结构保持不变，同时由子类提供部分实现。

## 特点

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新定义该算法的某些特定步骤。模板方法模式的使用场景：

* 多个子类有共有的方法，并且逻辑基本相同时；
* 重要、复杂的算法，可以把核心算法设计成模板方法，周边相关细节功能则由各个子类实现；
* 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。

## UML类图



* 抽象模板（Abstract Template）角色有如下责任：
* 定义了一个或多个抽象操作，以便让子类实现，这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。
* 定义并实现了一个模板方法，这个模板方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现，顶级逻辑也有可能调用一些具体方法。
* 具体模板（Concrete Template）角色有如下责任：
  + 实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。
  + 每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

## 模板方法模式的通用代码：

**ConcreteTemplateA.class**

**public** **class** **ConcreteTemplateA** **extends** **AbstractTemplate**{

@Override

**protected** **void** **abstractMethod**() {

**System**.**out**.print("ConcreteTemplateA abstractMethod\n");

}

}

**ConcreteTemplateB.class**

**public** **class** **ConcreteTemplateB** **extends** **AbstractTemplate**{

@Override

**protected** **void** **abstractMethod**() {

**System**.**out**.print("ConcreteTemplateB abstractMethod\n");

}

@Override

**protected** **void** **hookMethod**() {

**System**.**out**.print("ConcreteTemplateB hookMethod\n");

}

}

**AbstractTemplate.class**

**public** **abstract** **class** **AbstractTemplate** {

/\*\*

\* 模板方法

\*/

**public** **void** **templateMethod**(){

//调用基本方法

abstractMethod();

hookMethod();

concreteMethod();

}

/\*\*

\* 基本方法的声明（由子类实现）

\*/

**protected** **abstract** **void** **abstractMethod**();

/\*\*

\* 基本方法(空方法)

\*/

**protected** **void** **hookMethod**(){

**System**.**out**.print("AbstractTemplate hookMethod\n");

}

/\*\*

\* 基本方法（已经实现）

\*/

**private** **final** **void** **concreteMethod**(){

**System**.**out**.print("AbstractTemplate concreteMethod\n");

}

}

**测试代码**

**AbstractTemplate** templateA = **new** ConcreteTemplateA();

templateA.templateMethod();

**AbstractTemplate** templateB = **new** ConcreteTemplateB();

templateB.templateMethod()

**最后结果**

ConcreteTemplateA abstractMethod

AbstractTemplate hookMethod

AbstractTemplate concreteMethod

ConcreteTemplateB abstractMethod

ConcreteTemplateB hookMethod

AbstractTemplate concreteMethod

基类定义了一系列的步骤，然后子类按照这个步骤去实现，注意到两个方法，一个是 abstractMethod 和 hookMethod ，这两个方法的定位是不一样的， abstractMethod() 方法所代表的就是强制子类实现的剩余逻辑，而 hookMethod() 方法是可选择实现的逻辑，不是必须实现的。

## 示例与源码

Android 中使用模板方法模式最典型的例子就是 AsyncTask 和 Activity 的生命周期函数，这两个类基本都会有接触到，感兴趣的可以深入去了解。

　　我们这里以一个游戏的初始化为例子：

**Game .class**

abstract class Game {

protected int playersCount;

abstract void initializeGame();

abstract void makePlay(int player);

abstract boolean endOfGame();

abstract void printWinner();

/\* A template method : \*/

public final void playOneGame(int playersCount) {

this.playersCount = playersCount;

initializeGame();

int j = 0;

while (!endOfGame()) {

makePlay(j);

j = (j + 1) % playersCount;

}

printWinner();

}

}

**大富翁**

class Monopoly extends Game {

void initializeGame() {

System.out.print("Monopoly initializeGame\n");

}

void makePlay(int player) {

return player > 4 ? 4 : palyer;

}

boolean endOfGame() {

System.out.print("Monopoly endOfGame\n");

}

void printWinner() {

System.out.print("Monopoly printWinner\n");

}

//....

}

**象棋**

class Chess extends Game {

void initializeGame() {

System.out.print("Chess initializeGame\n");

}

void makePlay(int player) {

return 2;

}

boolean endOfGame() {

System.out.print("Chess endOfGame\n");

}

void printWinner() {

System.out.print("Chess printWinner\n");

}

//.....

}

## 总结

模板方法模式用四个字概括就是：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，这就要求父类提取公用的代码，提升代码的复用率，同时也带来了更好的可封装性。

### 优点：

封装不变部分，扩展可变部分；

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，加大理解的难度，而且由于父类将步骤固定，所以有时候会增加扩展的难度。

# 解释器模式（Interpreter）

## 概述

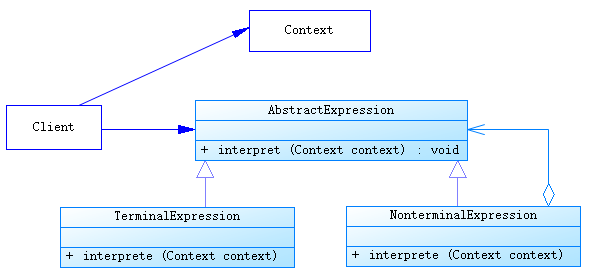
解释器模式（Interpreter Pattern），也是行为型设计模式之一，是一种用的比较少的设计模式，其提供了一种解释语言的语法或表达式的方式，该模式定义了一个表达式接口，通过该接口解释一个特定的上下文。在这么多的设计模式中，解释器模式在实际运用上相对来说要少很多，因为我们很少会去构造一个语言的文法。虽然你几乎用不到这个模式，但是看一看还是能受到一定的启发的。

## 特点

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。 该模式的使用场景相当广泛，总的概括下来大概有以下几种：

* 专用的数据库查询语言，比如 SQL；
* 通常用来解释通信协议的专用计算机语言；
* 如果某个简单的语言需要解释执行而且可以将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。

## UML类图



* 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 interpret() 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

实现了抽象表达式角色所要求的接口，主要是一个interpret()方法；文法中的每一个终结符都有一个具体终结表达式与之相对应。比如有一个简单的公式R=R1+R2，在里面R1和R2就是终结符，对应的解析R1和R2的解释器就是终结符表达式。

* NonterminalExpression：非终结符表达式

文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式R=R1+R2中，“+”就是非终结符，解析“+”的解释器就是一个非终结符表达式。

* Context：上下文环境类

这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如R=R1+R2，我们给R1赋值100，给R2赋值200。这些信息需要存放到环境角色中，很多情况下我们使用Map来充当环境角色就足够了。

* 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。

## 解释器模式的通用代码

public abstract class AbstractExpression {

/\*\*

\* 抽象的解析方法

\* @param context 上下文环境对象

\*/

public abstract void interpret(Context context);

}

public class TerminalExpression extends AbstractExpression{

@Override

public void interpret(Context context) {

//实现文法中与终结符有关的解释操作

}

}

public class NonterminalExpression extends AbstractExpression{

@Override

public void interpret(Context context) {

//实现文法中与非终结符有关的解释操作

}

}

public class Context {

}

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

}

}

## 示例与源码

为了说明解释器模式的实现办法，这里就以 wiki 上的算术表达式的解释为例，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。这个简单的文法如下：

Expression ::= plus | minus | variable | number

Plus ::= expression expression '+'

Minus ::= expression expression '-'

Variable ::= 'a' | 'b' | 'c' | ... | 'z'

Digit ::= '0' | '1' | ... | '9'

Number ::= digit | digit number

定义一个包含逆波兰表达式的语言：

a b +

a b c + -

a b + c a - -

于是根据上面的语言和文法，我们可以简单写出下面的示例：

**AbstractExpression，TerminalExpression，NonterminalExpression：**

interface Expression {

public int interpret(Map<String,Expression> variables);

}

class Number implements Expression {

private int number;

public Number(int number) { this.number = number; }

public int interpret(Map<String,Expression> variables) { return number; }

}

class Plus implements Expression {

Expression leftOperand;

Expression rightOperand;

public Plus(Expression left, Expression right) {

leftOperand = left;

rightOperand = right;

}

public int interpret(Map<String,Expression> variables) {

return leftOperand.interpret(variables) + rightOperand.interpret(variables);

}

}

class Minus implements Expression {

Expression leftOperand;

Expression rightOperand;

public Minus(Expression left, Expression right) {

leftOperand = left;

rightOperand = right;

}

public int interpret(Map<String,Expression> variables) {

return leftOperand.interpret(variables) - rightOperand.interpret(variables);

}

}

class Variable implements Expression {

private String name;

public Variable(String name) { this.name = name; }

public int interpret(Map<String,Expression> variables) {

if(null==variables.get(name)) return 0; //Either return new Number(0).

return variables.get(name).interpret(variables);

}

}

**Context**

class Evaluator {

private Expression syntaxTree;

public Evaluator(String expression) {

Stack<Expression> expressionStack = new Stack<Expression>();

for (String token : expression.split(" ")) {

if (token.equals("+")) {

Expression subExpression = new Plus(expressionStack.pop(), expressionStack.pop());

expressionStack.push( subExpression );

}

else if (token.equals("-")) {

// it's necessary remove first the right operand from the stack

Expression right = expressionStack.pop();

// ..and after the left one

Expression left = expressionStack.pop();

Expression subExpression = new Minus(left, right);

expressionStack.push( subExpression );

}

else

expressionStack.push( new Variable(token) );

}

syntaxTree = expressionStack.pop();

}

public int calculate(Map<String,Expression> context) {

return syntaxTree.interpret(context);

}

}

**Client**

public class InterpreterExample {

public static void main(String[] args) {

String expression = "w x z - +";

Evaluator sentence = new Evaluator(expression);

Map<String,Expression> variables = new HashMap<String,Expression>();

variables.put("w", new Number(5));

variables.put("x", new Number(10));

variables.put("z", new Number(42));

int result = sentence.calculate(variables);

System.out.println(result);

}

}

## 总结

解释器模式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。

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繁琐，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。

# 访问者模式（Visitor）

## 概述

访问者模式（Visitor Pattern），这也是行为型设计模式之一。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，它可以算是 23 中设计模式中最复杂的一个，但它的使用频率并不是很高，大多数情况下，你并不需要使用访问者模式，但是当你一旦需要使用它时，那你就是需要使用它了。

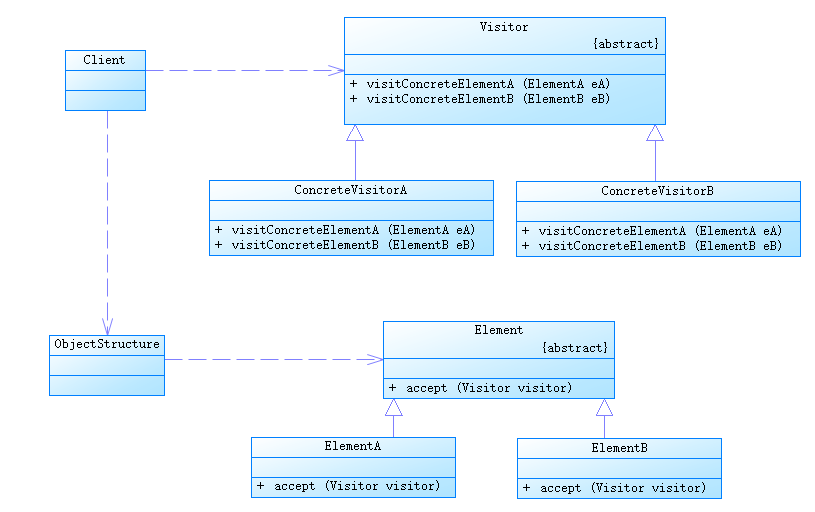
访问者模式的基本想法是，软件系统中拥有一个由许多对象构成的、比较稳定的对象结构，这些对象的类都拥有一个 accept 方法用来接受访问者对象的访问。访问者是一个接口，它拥有一个 visit 方法，这个方法对访问到的对象结构中不同类型的元素做出不同的处理。在对象结构的一次访问过程中，我们遍历整个对象结构，对每一个元素都实施 accept 方法，在每一个元素的 accept 方法中会调用访问者的 visit 方法，从而使访问者得以处理对象结构的每一个元素，我们可以针对对象结构设计不同的访问者类来完成不同的操作，达到区别对待的效果。

## 特点

封装一些作用于某种数据结构中的个元素的操作，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。访问者模式使用的场景

* 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；
* 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。

## UML类图



* 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：具体地访问者，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。

## 分派的概念

变量被声明时的类型叫做变量的静态类型(Static Type)，有些人又把静态类型叫做明显类型(Apparent Type)；而变量所引用的对象的真实类型又叫做变量的实际类型(Actual Type)。比如：

List list = null;

list = new ArrayList();

声明了一个变量list，它的静态类型（也叫明显类型）是List，而它的实际类型是ArrayList。

根据对象的类型而对方法进行的选择，就是分派(Dispatch)，分派(Dispatch)又分为两种，即静态分派和动态分派。静态分派(Static Dispatch)发生在编译时期，分派根据静态类型信息发生。静态分派对于我们来说并不陌生，方法重载就是静态分派；动态分派(Dynamic Dispatch)发生在运行时期，动态分派动态地置换掉某个方法。接着我们来具体看看静态分派和动态分派：

### 静态分派

Java通过方法重载支持静态分派，我们以一个例子来看一下：

public class StaticDispatch {

public void sayHello(Human guy) {

System.out.println("hello, guy!");

}

public void sayHello(Man guy) {

System.out.println("hello, man!");

}

public void sayHello(Women guy) {

System.out.println("hello, women!");

}

public static void main(String[] args) {

Human man = new Man();

Human women = new Women();

StaticDispatch sd = new StaticDispatch();

sd.sayHello(man);

sd.sayHello(women);

}

}

class Human {

}

class Man extends Human {

}

class Women extends Human {

}

最后的输出结果：

hello, guy!

hello, guy!

没错，程序就是大家熟悉的重载（Overload），而且大家也应该能知道输出结果，但是为什么输出结果会是这个呢，先来看一下代码的定义：

Human man = new Man();

我们把 Human 称为变量的静态类型， Man 称为变量的实际类型，其中，变量的静态类型和动态类型在程序中都可以发生变化，而区别是变量的静态类型是在编译阶段就可知的，但是动态类型要在运行期才可以确定，编译器在编译的时候并不知道变量的实际类型是什么。现在回到代码中，由于方法的接受者已经确定是 StaticDispatch 的实例sd了，所以最终调用的是哪个重载版本也就取决于传入参数的类型了。实际上，虚拟机（应该说是编译器）在重载时时通过参数的静态类型来当判定依据的，而且静态类型在编译期就可知，所以编译器在编译阶段就可根据静态类型来判定究竟使用哪个重载版本。于是对于例子中的两个方法的调用都是以Human为参数的版本，Java中，所有**以静态类型来定位方法执行版本的分派动作，都称为静态分派**。

### 动态分派

Java 通过方法的重写支持动态分派，它和多态的另外一个重要体现有很大的关联，这个体现是什么，可能大家也能猜出，没错，就是重写（override），我们来看看例子：

public class DynamicDispatch {

public static void main(String[] args) {

Human man = new Man();

Human women = new Women();

man.sayHello();

women.sayHello();

man = new Women();

man.sayHello();

}

}

abstract class Human {

protected abstract void sayHello();

}

class Man extends Human {

@Override

protected void sayHello() {

System.out.println("hello man!");

}

}

class Women extends Human {

@Override

protected void sayHello() {

System.out.println("hello women!");

}

}

输出结果很显然：

hello man!

hello women!

hello women!

其实由两次改变 man 变量的实际类型导致调用函数版本不同，我们就可以知道，虚拟机是根据变量的实际类型来调用重写方法的。我们也可以从例子中看出，变量的实际类型是在运行期确定的，重写方法的调用也是根据实际类型来调用的，而不是根据静态类型。我们把这种在运行期根据实际类型来确定方法执行版本的分派动作，称为动态分派。

### 编程呢言的分派类型

一个方法所属的对象叫做方法的接收者，方法的接收者与方法的参数统称做方法的宗量。比如下面例子中的Test类：

public class Test {

public void print(String str){

System.out.println(str);

}

}

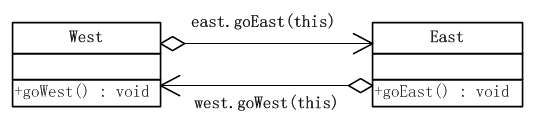
在上面的类中，print() 方法属于 Test 对象，所以它的接收者也就是 Test 对象了。print()方法有一个参数是 str，它的类型是 String。

　　根据分派可以基于多少种宗量，可以将面向对象的语言划分为单分派语言（Uni-Dispatch）和多分派语言（Multi-Dispatch）。单分派语言根据一个宗量的类型进行对方法的选择，多分派语言根据多于一个的宗量的类型对方法进行选择。C++ 和 Java均是单分派语言，多分派语言的例子包括 CLOS 和 Cecil 。按照这样的区分，Java 就是动态的单分派语言，因为这种语言的动态分派仅仅会考虑到方法的接收者的类型，同时又是静态的多分派语言，因为这种语言对重载方法的分派会考虑到方法的接收者的类型以及方法的所有参数的类型。

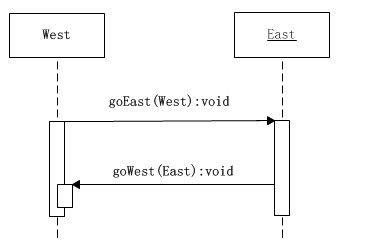
　　在一个支持动态单分派的语言里面，有两个条件决定了一个请求会调用哪一个操作：一是请求的名字，二是接收者的真实类型。单分派限制了方法的选择过程，使得只有一个宗量可以被考虑到，这个宗量通常就是方法的接收者。在 Java 语言里面，如果一个操作是作用于某个类型不明的对象上面，那么对这个对象的真实类型测试仅会发生一次，这就是动态的单分派的特征。

### 双重分派

一个方法根据两个宗量的类型来决定执行不同的代码，这就是“双重分派”。Java 语言不支持动态的多分派，也就意味着 Java 不支持动态的双分派。但是通过使用访问者模式，也可以在 Java 语言里实现动态的双重分派。在 Java 中可以通过两次方法调用来达到两次分派的目的。类图如下所示：



在图中有两个对象，左边的叫做 West ，右边的叫做 East 。现在 West 对象首先调用 East 对象的 goEast() 方法，并将它自己传入。在 East 对象被调用时，立即根据传入的参数知道了调用者是谁，于是反过来调用“调用者”对象的 goWest() 方法。通过两次调用将程序控制权轮番交给两个对象，其时序图如下所示：



这样就出现了两次方法调用，程序控制权被两个对象像传球一样，首先由 West 对象传给了 East 对象，然后又被返传给了 West 对象。但是仅仅返传了一下球，并不能解决双重分派的问题。关键是怎样利用这两次调用，以及 Java 语言的动态单分派功能，使得在这种传球的过程中，能够触发两次单分派。

## 访问者模式的通用代码

　　动态单分派在 Java 语言中是在子类重写父类的方法时发生的。换言之，West 和 East 都必须分别置身于自己的类型等级结构中，就正如上面的访问者模式 uml 类图，我们写出访问者模式的通用代码：

**Visitor角色：**

**West.class**

public abstract class West {

public abstract void goWest1(SubEast1 east);

public abstract void goWest2(SubEast2 east);

}

**ConcreteVisitor角色：**

**SubWest1.class**

public class SubWest1 extends West{

@Override

public void goWest1(SubEast1 east) {

System.out.println("SubWest1 + " + east.myName1());

}

@Override

public void goWest2(SubEast2 east) {

System.out.println("SubWest1 + " + east.myName2());

}

}

**SubWest2.class**

public class SubWest2 extends West{

@Override

public void goWest1(SubEast1 east) {

System.out.println("SubWest2 + " + east.myName1());

}

@Override

public void goWest2(SubEast2 east) {

System.out.println("SubWest2 + " + east.myName2());

}

}

**Element角色：**

**East.class**

public abstract class East {

public abstract void goEast(West west);

}

**ConcreteElement角色：**

**SubEast1.class**

public class SubEast1 extends East{

@Override

public void goEast(West west) {

west.goWest1(this);

}

public String myName1(){

return "SubEast1";

}

}

**SubEast2.class**

public class SubEast2 extends East{

@Override

public void goEast(West west) {

west.goWest2(this);

}

public String myName2(){

return "SubEast2";

}

}

**Client.class**

public class Client {

public static void main(String[] args) {

//组合1

East east = new SubEast1();

West west = new SubWest1();

east.goEast(west);

//组合2

east = new SubEast1();

west = new SubWest2();

east.goEast(west);

}

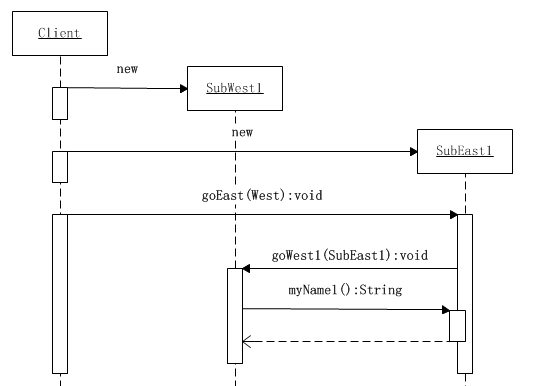
}

**最后结果如下：**

SubWest1 + SubEast1

SubWest2 + SubEast1

系统运行时，会首先创建 SubWest1 和 SubEast1 对象，然后客户端调用 SubEast1 的goEast() 方法，并将 SubWest1 对象传入。由于 SubEast1 对象重写了其超类 East 的 goEast() 方法，因此，这个时候就发生了一次动态的单分派。当 SubEast1 对象接到调用时，会从参数中得到 SubWest1 对象，所以它就立即调用这个对象的 goWest1() 方法，并将自己传入。由于 SubEast1 对象有权选择调用哪一个对象，因此，在此时又进行一次动态的方法分派。这个时候 SubWest1 对象就得到了 SubEast1 对象。通过调用这个对象 myName1() 方法，就可以打印出自己的名字和 SubEast 对象的名字，其时序图如下所示：



由于这两个名字一个来自East等级结构，另一个来自West等级结构中，因此，它们的组合式是动态决定的，这就是在 Java 中动态双重分派的实现机制。

## 总结

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以及优缺点。

### 优点：

* 各角色职责分离，符合单一职责原则；
* 能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能，具有良好的扩展性；
* 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；
* 灵活性。

### 缺点：

* 具体元素对访问者公布细节，违反了迪米特原则；
* 具体元素变更时导致修改成本大；
* 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。