



尚硅谷 Java 设计模式

尚硅谷-韩顺平



第 1 章 内容介绍和授课方式.....	1
1.1 JAVA 设计模式内容介绍	1
1.1.1 先看几个经典的面试题	1
1.1.2 设计模式的重要性	3
1.2 课程亮点和授课方式.....	4
第 2 章 设计模式七大原则.....	6
2.1 设计模式的目的	6
2.2 设计模式七大原则	6
2.3 单一职责原则.....	7
2.3.1 基本介绍	7
2.3.2 应用实例	7
2.3.3 单一职责原则注意事项和细节	12
2.4 接口隔离原则(INTERFACE SEGREGATION PRINCIPLE).....	12
2.4.1 基本介绍	12
2.4.2 应用实例	13
2.4.3 应传统方法的问题和使用接口隔离原则改进	16
2.5 依赖倒转原则.....	21
2.5.1 基本介绍	21
2.5.2 应用实例	21
2.5.3 依赖关系传递的三种方式和应用案例	24
2.5.4 依赖倒转原则的注意事项和细节	28
2.6 里氏替换原则.....	29
2.6.1 OO 中的继承性的思考和说明	29
2.6.2 基本介绍	29
2.6.3 一个程序引出的问题和思考	30
2.6.4 解决方法	31
2.7 开闭原则.....	34
2.7.1 基本介绍	34
2.7.2 看下面一段代码	35
2.7.3 方式 1 的优缺点	38
2.7.4 改进的思路分析	38
2.8 迪米特法则.....	42
2.8.1 基本介绍	42
2.8.2 应用实例	42
2.8.3 应用实例改进	46
2.8.4 迪米特法则注意事项和细节	51
2.9 合成复用原则 (COMPOSITE REUSE PRINCIPLE)	51
2.9.1 基本介绍	51



2.10 设计原则核心思想	51
第 3 章 UML 类图	53
3.1 UML 基本介绍	53
3.2 UML 图	54
3.3 UML 类图	55
3.4 类图—依赖关系 (DEPENDENCE)	56
3.5 类图—泛化关系(GENERALIZATION)	57
3.6 类图—实现关系 (IMPLEMENTATION)	58
3.7 类图—关联关系 (ASSOCIATION)	59
3.8 类图—聚合关系 (AGGREGATION)	59
3.8.1 基本介绍	59
3.8.2 应用实例	59
3.9 类图—组合关系 (COMPOSITION)	60
3.9.1 基本介绍	60
3.9.2 应用案例	60
第 4 章 设计模式概述	63
4.1 掌握设计模式的层次	63
4.2 设计模式介绍	63
4.3 设计模式类型	63
第 5 章 单例设计模式	65
5.1 单例设计模式介绍	65
5.2 单例设计模式八种方式	65
5.3 饿汉式 (静态常量)	65
5.4 饿汉式 (静态代码块)	67
5.5 懒汉式(线程不安全)	69
5.6 懒汉式(线程安全, 同步方法)	71
5.7 懒汉式(线程安全, 同步代码块)	72
5.8 双重检查	73
5.9 静态内部类	75
5.10 枚举	77
5.11 单例模式在 JDK 应用的源码分析	78
5.11.1 单例模式在 JDK 应用的源码分析	78
5.12 单例模式注意事项和细节说明	79
第 6 章 工厂模式	80
6.1 简单工厂模式	80
6.1.1 看一个具体的需求	80



6.1.2 使用传统的方式来完成	80
6.1.3 传统的方式的优缺点	82
6.1.4 基本介绍	82
6.1.5 使用简单工厂模式	83
6.2 工厂方法模式	89
6.2.1 看一个新的需求	89
6.2.2 思路 1	89
6.2.3 思路 2	89
6.2.4 工厂方法模式介绍	89
6.2.5 工厂方法模式应用案例	89
6.3 抽象工厂模式	94
6.3.1 基本介绍	94
6.3.2 抽象工厂模式应用实例	95
6.4 工厂模式在 JDK-CALENDAR 应用的源码分析	99
6.5 工厂模式小结	103
第 7 章 原型模式	104
7.1 克隆羊问题	104
7.2 传统方式解决克隆羊问题	104
7.3 传统的方式的优缺点	105
7.4 原型模式-基本介绍	105
7.5 原型模式原理结构图-UML 类图	106
7.6 原型模式解决克隆羊问题的应用实例	106
7.7 原型模式在 SPRING 框架中源码分析	110
7.8 深入讨论-浅拷贝和深拷贝	110
7.8.1 浅拷贝的介绍	110
7.8.2 深拷贝基本介绍	111
7.9 深拷贝应用实例	111
7.10 原型模式的注意事项和细节	116
第 8 章 建造者模式	118
8.1 盖房项目需求	118
8.2 传统方式解决盖房需求	118
8.3 传统方式的问题分析	120
8.4 建造者模式基本介绍	121
8.5 建造者模式的四个角色	121
8.6 建造者模式原理类图	121
8.7 建造者模式解决盖房需求应用实例	122
8.8 建造者模式在 JDK 的应用和源码分析	129



8.9 建造者模式的注意事项和细节	129
第 9 章 适配器设计模式.....	131
9.1 现实生活中的适配器例子	131
9.2 基本介绍.....	131
9.3 工作原理.....	131
9.4 类适配器模式.....	132
9.4.1 类适配器模式介绍	132
9.4.2 类适配器模式应用实例	132
9.4.3 类适配器模式注意事项和细节	135
9.5 对象适配器模式.....	135
9.5.1 对象适配器模式介绍	135
9.5.2 对象适配器模式应用实例	136
9.5.3 对象适配器模式注意事项和细节	140
9.6 接口适配器模式.....	140
9.6.1 接口适配器模式介绍	140
9.6.2 接口适配器模式应用实例	140
9.7 适配器模式在 SPRINGMVC 框架应用的源码剖析	144
9.8 适配器模式的注意事项和细节	146
第 10 章 桥接模式.....	147
10.1 手机操作问题	147
10.2 传统方案解决手机操作问题	147
10.3 传统方案解决手机操作问题分析	148
10.4 桥接模式(BRIDGE)-基本介绍	148
10.5 桥接模式(BRIDGE)-原理类图	148
10.6 桥接模式解决手机操作问题	149
10.7 桥接模式在 JDBC 的源码剖析	156
10.8 桥接模式的注意事项和细节	157
10.9 常见的应用场景	158
第 11 章 装饰者设计模式.....	159
11.1 星巴克咖啡订单项目 (咖啡馆) :	159
11.2 方案 1-解决星巴克咖啡订单项目	159
11.3 方案 1-解决星巴克咖啡订单问题分析	159
11.4 方案 2-解决星巴克咖啡订单(好点)	160
11.5 方案 2-解决星巴克咖啡订单问题分析	160
11.6 装饰者模式定义	161
11.7 装饰者模式原理	161
11.8 装饰者模式解决星巴克咖啡订单	162



11.9 装饰者模式下的订单：2份巧克力+一份牛奶的 LONGBLACK.....	162
11.10 装饰者模式咖啡订单项目应用实例	162
11.11 装饰者模式在 JDK 应用的源码分析	170
第 12 章 组合模式.....	172
12.1 看一个学校院系展示需求	172
12.2 传统方案解决学校院系展示(类图)	172
12.3 传统方案解决学校院系展示存在的问题分析	172
12.4 组合模式基本介绍	173
12.5 组合模式原理类图	173
12.6 组合模式解决学校院系展示的 应用实例	174
12.7 组合模式在 JDK 集合的源码分析	183
12.8 组合模式的注意事项和细节	184
第 13 章 外观模式.....	185
13.1 影院管理项目	185
13.2 传统方式解决影院管理	185
13.3 传统方式解决影院管理问题分析	186
13.4 外观模式基本介绍	186
13.5 外观模式原理类图	186
13.6 外观模式解决影院管理	187
13.6.1 传统方式解决影院管理说明	187
13.6.2 外观模式应用实例	187
13.7 外观模式在 MYBATIS 框架应用的源码分析	197
13.8 外观模式的注意事项和细节	198
第 14 章 享元模式.....	199
14.1 展示网站项目需求	199
14.2 传统方案解决网站展现项目	199
14.3 传统方案解决网站展现项目-问题分析	199
14.4 享元模式基本介绍	200
14.5 享元模式的原理类图	200
14.6 内部状态和外部状态	201
14.7 享元模式解决网站展现项目	202
14.8 享元模式在 JDK-INTERGER 的应用源码分析	207
14.9 享元模式的注意事项和细节	209
第 15 章 代理模式.....	210
15.1 代理模式(PROXY).....	210
15.1.1 代理模式的基本介绍	210



15.2 静态代理	210
15.2.1 静态代码模式的基本介绍	210
15.2.2 应用实例	211
15.2.3 静态代理优缺点	214
15.3 动态代理	214
15.3.1 动态代理模式的基本介绍	214
15.3.2 JDK 中生成代理对象的 API	214
15.3.3 动态代理应用实例	214
15.4 CGLIB 代理	219
15.4.1 Cglib 代理模式的基本介绍	219
15.4.2 Cglib 代理模式实现步骤	220
15.4.3 Cglib 代理模式应用实例	220
15.5 几种常见的代理模式介绍— 几种变体	223
第 16 章 模板方法模式.....	225
16.1 豆浆制作问题	225
16.2 模板方法模式基本介绍	225
16.3 模板方法模式原理类图	225
16.3.1 模板方法模式的原理类图	225
16.4 模板方法模式解决豆浆制作问题	226
16.5 模板方法模式的钩子方法	230
16.6 模板方法模式在 SPRING 框架应用的源码分析	232
16.7 模板方法模式的注意事项和细节	233
第 17 章 命令模式.....	234
17.1 智能生活项目需求	234
17.2 命令模式基本介绍	234
17.3 命令模式的原理类图	235
17.4 命令模式解决智能生活项目	235
17.5 命令模式在 SPRING 框架 JDBCTEMPLATE 应用的源码分析	247
17.6 命令模式的注意事项和细节	248
第 18 章 访问者模式.....	249
18.1 测评系统的需求	249
18.2 传统方式的问题分析	249
18.3 访问者模式基本介绍	249
18.4 访问者模式的原理类图	250
18.5 访问者模式应用实例	251
18.6 访问者模式的注意事项和细节	258



尚硅谷 Java 设计模式

19.1 看一个具体的需求	259
19.2 传统的设计方案(类图)	259
19.3 传统的方式的问题分析	259
19.4 迭代器模式基本介绍	259
19.5 迭代器模式的原理类图	260
19.6 迭代器模式应用实例	260
19.7 迭代器模式在 JDK-ARRAYLIST 集合应用的源码分析	272
19.8 迭代器模式的注意事项和细节	273
第 20 章 观察者模式	275
20.1 天气预报项目需求,具体要求如下:	275
20.2 天气预报设计方案 1-普通方案	275
20.2.1 WeatherData 类	275
20.3 观察者模式原理	280
20.4 观察者模式解决天气预报需求	280
20.4.1 类图说明	280
20.4.2 代码实现	281
20.4.3 观察者模式的好处	288
20.5 观察者模式在 JDK 应用的源码分析	288
第 21 章 中介者模式	290
21.1 智能家庭项目	290
21.2 传统方案解决智能家庭管理问题	290
21.3 传统的方式的问题分析	290
21.4 中介者模式基本介绍	291
21.5 中介者模式的原理类图	291
21.6 中介者模式应用实例-智能家庭管理	291
21.7 中介者模式的注意事项和细节	301
第 22 章 备忘录模式	302
22.1 游戏角色状态恢复问题	302
22.2 传统方案解决游戏角色恢复	302
22.3 传统的方式的问题分析	302
22.4 备忘录模式基本介绍	302
22.5 备忘录模式的原理类图	303
22.6 游戏角色恢复状态实例	308
22.7 备忘录模式的注意事项和细节	313
第 23 章 解释器模式	315
23.1 四则运算问题	315



尚硅谷 Java 设计模式

23.2 传统方案解决四则运算问题分析	315
23.3 解释器模式基本介绍	315
23.4 解释器模式的原理类图	316
23.5 解释器模式来实现四则	316
23.6 解释器模式在 SPRING 框架应用的源码剖析	325
23.7 解释器模式的注意事项和细节	326
第 24 章 状态模式.....	327
24.1 APP 抽奖活动问题	327
24.2 状态模式基本介绍	327
24.3 状态模式的原理类图	327
24.4 状态模式解决 APP 抽奖问	328
24.5 状态模式在实际项目-借贷平台 源码剖析	341
24.6 状态模式的注意事项和细节	354
第 25 章 策略模式.....	355
25.1 编写鸭子项目，具体要求如下	355
25.2 传统方案解决鸭子问题的分析和代码实现	355
25.3 传统的方式实现的问题分析和解决方案	359
25.4 策略模式基本介绍	359
25.5 策略模式的原理类图	359
25.6 策略模式解决鸭子问题	360
25.7 策略模式在 JDK-ARRAYS 应用的源码分析	368
25.8 策略模式的注意事项和细节	371
第 26 章 职责链模式.....	372
26.1 学校 OA 系统的采购审批项目：需求是	372
26.2 传统方案解决 OA 系统审批，传统的设计方案(类图).....	372
26.3 传统方案解决 OA 系统审批问题分析	372
26.4 职责链模式基本介绍	373
26.5 职责链模式的原理类图	373
26.6 职责链模式解决 OA 系统采购审批	373
26.7 职责链模式在 SPRINGMVC 框架应用的源码分析	381
26.8 职责链模式的注意事项和细节	383

第 1 章 内容介绍和授课方式

1.1 Java 设计模式内容介绍

1.1.1 先看几个经典的面试题

➤ 原型设计模式问题：

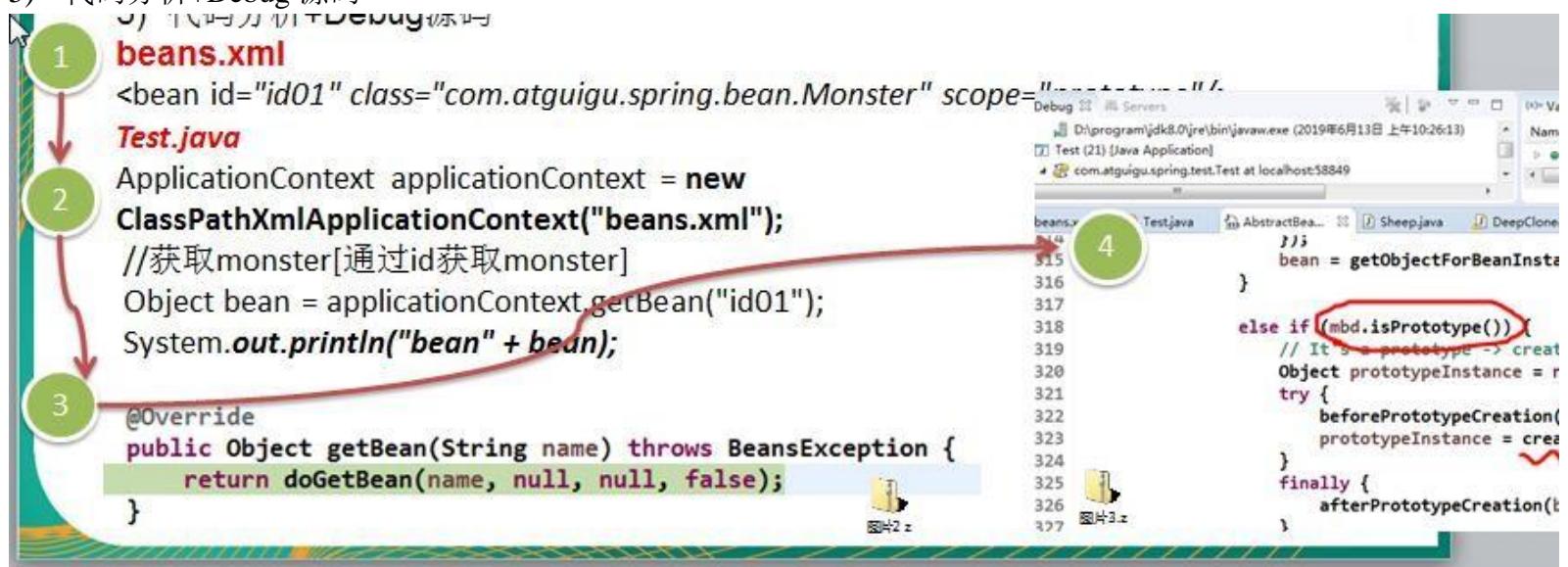
- 1) 有请使用 UML 类图画出原型模式核心角色
- 2) 原型设计模式的深拷贝和浅拷贝是什么，并写出深拷贝的两种方式的源码(重写 clone 方法实现深拷贝、使用序列化来实现深拷贝)
- 3) 在 Spring 框架中哪里使用到原型模式，并对源码进行分析

beans.xml

```
<bean id="id01" class="com.atguigu.spring.bean.Monster" scope="prototype"/>
```

- 4) Spring 中原型 bean 的创建，就是原型模式的应用

- 5) 代码分析+Debug 源码

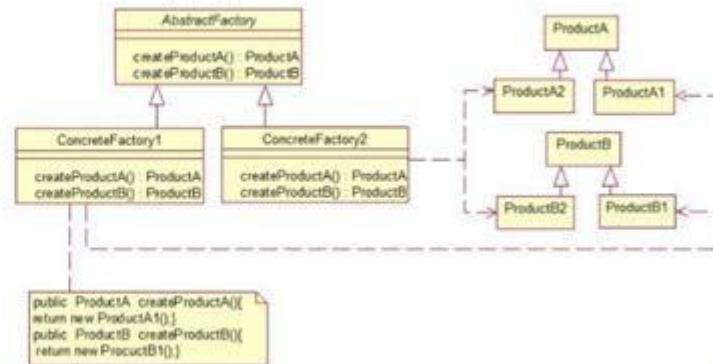


➤ 设计模式的七大原则：要求：1) 七大设计原则核心思想 2) 能够以类图的说明设计原则 3) 在项目实际开发中，

你在哪里使用到了 ocp 原则

设计模式常用的七大原则有：

- 1) 单一职责原则
- 2) 接口隔离原则
- 3) 依赖倒转原则
- 4) 里氏替换原则
- 5) 开闭原则 ocp
- 6) 迪米特法则
- 7) 合成复用原则



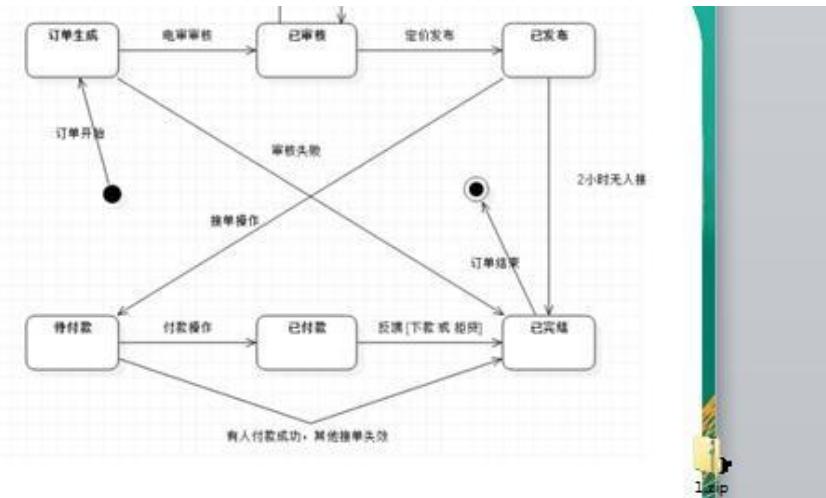
图片

先看几个经典的设计模式面试题

金融借贷平台项目: 借贷平台的订单, 有审核-发布-抢单 等等步骤, 随着操作的不同, 会改变订单的状态, 项目中的这个模块实现就会使用到状态模式, 请你使用**状态模式**进行设计, 并完成实际代码

问题分析 :

这类代码难以应对变化, 在添加一种状态时, 我们需要手动添加if/else, 在添加一种功能时, 要对所有的状态进行判断。因此代码会变得越来越臃肿, 并且一旦没有处理某个状态, 便会发生极其严重的BUG, 难以维护



事件	待审核 (1)	电审失败 (2)	定价发布 (3)	抢单 (4)	抢单失败 (5)	付款 (6)	支付失败 (7)	反馈 (8)
订单生成	已审核 (2)	已审核 (2)	已发布 (3)					
已审核 (2)				已审核 (2)	已审核 (2)			
已发布 (3)				待付款 (4)	待付款 (4)	已付款 (5)	已付款 (5)	已完结 (6)
待付款 (4)					已付款 (5)	已完结 (6)		已完结 (6)
已付款 (5)								
已完结 (6)								



➤ 解释器设计模式

- 1) 介绍解释器设计模式是什么?
- 2) 画出解释器设计模式的 UML 类图, 分析设计模式中的各个角色是什么?

- 3) 请说明 Spring 的框架中，哪里使用到了解释器设计模式，并做源码级别的分析
 - 4) Spring 框架中 SpelExpressionParser 就使用到解释器模式
 - 5) 代码分析+Debug 源码+模式角色分析说明
 - 1) Spring 框架中 SpelExpressionParser 就使用到解释器模式

1) Spring框架中 SpelExpressionParser就使用到解释器模式

2) 代码分析+Debug源码+模式角色分析说明

```
public class Interpreter {  
    public static void main(String[] args) {  
        SpelExpressionParser parser = new SpelExpressionParser();  
        Expression expression = parser.parseExpression("100 * (2 + 400) *  
        1 + 66");  
        int result = (Integer) expression.getValue();  
        System.out.println(result);  
    }  
}
```

```
public interface ExpressionParser {  
    Expression parseExpression(String expressionString) throws ParseException;  
    Expression parseExpression(String expressionString,  
        ParserContext context) throws ParseException;  
}
```

```
public abstract class TemplateAwareExpressionParser  
implements ExpressionParser {  
    public Expression parseExpression(String  
        expressionString, ParserContext context) { // 不同情况  
        返回不同的Expression. //看源码  
    }  
}
```

```
public class SpeExpressionParser  
extends  
TemplateAwareExpressionParser{
```

```
class InternalSpelExpressionParser  
extends  
TemplateAwareExpressionParser {
```



单例设计模式一共有**8**种写法，后面我们会依次讲到
饿汉式 两种
懒汉式 三种
双重检查
静态内部类
枚举

1.1.2 设计模式的重要性

-
- 1) 软件工程中，设计模式（design pattern）是对软件设计中普遍存在（反复出现）的各种问题，所提出的解决方案。这个术语是由埃里希·伽玛（Erich Gamma）等人在 1990 年代从建筑设计领域引入到计算机科学的
 - 2) 大厦 VS 简易房



- 3) 拿实际工作经历来说，当一个项目开发完后，如果客户提出增新功能，怎么办？。（可扩展性，使用设计模式，软件具有很好的扩展性）



- 4) 如果项目开发完后，原来程序员离职，你接手维护该项目怎么办？（维护性[可读性、规范性]）
- 5) 目前程序员门槛越来越高，一线 IT 公司(大厂)，都会问你在实际项目中使用过什么设计模式，怎样使用的，解决了什么问题。
- 6) 设计模式在软件中哪里？面向对象(oo)=>功能模块[设计模式+算法(数据结构)]=>框架[使用到多种设计模式]=>架构 [服务器集群]
- 7) 如果想成为合格软件工程师，那就花时间来研究下设计模式是非常必要的。

1.2 课程亮点和授课方式

- 1) 课程深入，非蜻蜓点水



-
- 2) 课程成体系，非星星点灯
 - 3) 高效而愉快的学习，设计模式很有用，其实也很好玩，很像小时候搭积木，怎样搭建更加稳定，坚固
 - 4) 设计模式很重要，因为包含很多编程思想，还是有一定难度的，我们努力做到通俗易懂
 - 5) 采用 应用场景->设计模式->剖析原理->分析实现步骤(图解)->代码实现-> 框架或项目源码分析(找到使用的地方) 的步骤讲解 [比如：建造者模式]
 - 6) 课程目标：让大家掌握本质，达能在工作中灵活运用解决实际问题和优化程序结构的目的。

第 2 章 设计模式七大原则

2.1 设计模式的目的

编写软件过程中，程序员面临着来自耦合性，内聚性以及可维护性，可扩展性，重用性，灵活性等多方面的挑战，设计模式是为了让程序(软件)，具有更好

- 1) 代码重用性 (即：相同功能的代码，不用多次编写)
- 2) 可读性 (即：编程规范性，便于其他程序员的阅读和理解)
- 3) 可扩展性 (即：当需要增加新的功能时，非常的方便，称为可维护)
- 4) 可靠性 (即：当我们增加新的功能后，对原来的功能没有影响)
- 5) 使程序呈现高内聚，低耦合的特性
分享金句：
6) 设计模式包含了面向对象的精髓，“懂了设计模式，你就懂了面向对象分析和设计（OOA/D）的精要”
7) Scott Meyers 在其巨著《Effective C++》就曾经说过：C++老手和 C++新手的区别就是前者手背上有很多伤疤

2.2 设计模式七大原则

设计模式原则，其实就是程序员在编程时，应当遵守的原则，也是各种设计模式的基础(即：设计模式为什么这样设计的依据)

➤ 设计模式常用的七大原则有：

- 1) 单一职责原则



- 2) 接口隔离原则
- 3) 依赖倒转(倒置)原则
- 4) 里氏替换原则
- 5) 开闭原则
- 6) 迪米特法则
- 7) 合成复用原则

2.3 单一职责原则

2.3.1 基本介绍

对类来说的，即一个类应该只负责一项职责。如类 A 负责两个不同职责：职责 1，职责 2。当职责 1 需求变更而改变 A 时，可能造成职责 2 执行错误，所以需要将类 A 的粒度分解为 A1, A2

2.3.2 应用实例

以交通工具案例讲解
看老师代码演示

1) 方案 1 [分析说明]

```
package com.atguigu.principle.singleresponsibility;

public class SingleResponsibility1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```



```
Vehicle vehicle = new Vehicle();

vehicle.run("摩托车");
vehicle.run("汽车");
vehicle.run("飞机");

}

}

// 交通工具类
// 方式 1
// 1. 在方式 1 的 run 方法中，违反了单一职责原则
// 2. 解决的方案非常的简单，根据交通工具运行方法不同，分解成不同类即可
class Vehicle {

    public void run(String vehicle)
    { System.out.println(vehicle + " 在公路上运
行....");
    }
}
```

2) 方案 2 [分析说明]

```
package com.atguigu.principle.singleresponsibility;

public class SingleResponsibility2 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```



```
RoadVehicle roadVehicle = new RoadVehicle();
```

```
    roadVehicle.run("摩托车");
```

```
    roadVehicle.run("汽车");
```

```
AirVehicle airVehicle = new AirVehicle();
```

```
    airVehicle.run("飞机");
```

```
}
```

```
}
```

//方案 2 的分析

//1. 遵守单一职责原则

//2. 但是这样做的改动很大，即将类分解，同时修改客户端

//3. 改进：直接修改 Vehicle 类，改动的代码会比较少=>方案 3

```
class RoadVehicle {
```

```
    public void run(String vehicle)
        { System.out.println(vehicle + "公路运行");
    }
```

```
}
```

```
class AirVehicle {
```

```
    public void run(String vehicle)
        { System.out.println(vehicle + "天空运行");
    }
```



```
}

class WaterVehicle {
    public void run(String vehicle)
        { System.out.println(vehicle + "水中运行");
    }
}
```

3) 方案 3 [分析说明]

```
package com.atguigu.principle.singleresponsibility;

public class SingleResponsibility3 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Vehicle2 vehicle2 = new Vehicle2();

        vehicle2.run("汽车");
        vehicle2.runWater("轮船");
        vehicle2.runAir("飞机");
    }
}
```

//方式 3 的分析



//1. 这种修改方法没有对原来的类做大的修改，只是增加方法

//2. 这里虽然没有在类这个级别上遵守单一职责原则，但是在方法级别上，仍然是遵守单一职责

```
class Vehicle2 {  
    public void run(String vehicle) {  
        //处理  
        System.out.println(vehicle + " 在公路上运行....");  
    }  
  
    public void runAir(String vehicle)  
    { System.out.println(vehicle + " 在天空上运  
        行....");  
    }  
  
    public void runWater(String vehicle)  
    { System.out.println(vehicle + " 在水中行....");  
    }  
  
    //方法 2.  
    //..  
    //..  
    //...  
}
```

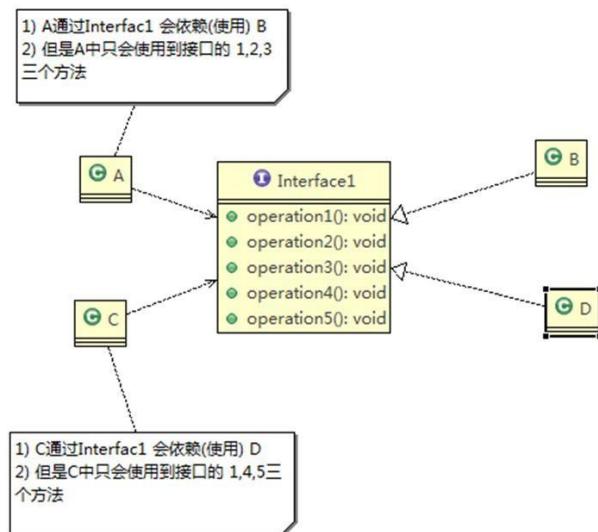
2.3.3 单一职责原则注意事项和细节

- 1) 降低类的复杂度，一个类只负责一项职责。
- 2) 提高类的可读性，可维护性
- 3) 降低变更引起的风险
- 4) 通常情况下，我们应当遵守单一职责原则，只有逻辑足够简单，才可以在代码级违反单一职责原则；**只有类中方法数量足够少，可以在方法级别保持单一职责原则**

2.4 接口隔离原则(Interface Segregation Principle)

2.4.1 基本介绍

- 1) 客户端不应该依赖它不需要的接口，即一个类对另一个类的依赖应该建立在最小的接口上
- 2) 先看一张图：





- 3) 类 A 通过接口 Interface1 依赖类 B, 类 C 通过接口 Interface1 依赖类 D, 如果接口 Interface1 对于类 A 和类 C 来说不是最小接口, 那么类 B 和类 D 必须去实现他们不需要的方法。
- 4) 按隔离原则应当这样处理:
将接口 **Interface1** 拆分为独立的几个接口(这里我们拆分成 3 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

2.4.2 应用实例

- 1) 类 A 通过接口 Interface1 依赖类 B, 类 C 通过接口 Interface1 依赖类 D, 请编写代码完成此应用实例。
2) 看老师代码-没有使用接口隔离原则代码

```
package com.atguigu.principle.segregation;

public class Segregation1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}

//接口
interface Interface1 {
    void operation1();
    void operation2();
}
```



```
void operation3();
void operation4();
void operation5();
}

class B implements Interface1
{ public void operation1()
{
    System.out.println("B 实现了 operation1");
}

public void operation2() {
    System.out.println("B 实现了 operation2");
}

public void operation3() {
    System.out.println("B 实现了 operation3");
}

public void operation4() {
    System.out.println("B 实现了 operation4");
}

public void operation5() {
    System.out.println("B 实现了 operation5");
}

}

class D implements Interface1
{ public void operation1() {
```



```
System.out.println("D 实现了 operation1");
}

public void operation2() {
    System.out.println("D 实现了 operation2");
}

public void operation3() {
    System.out.println("D 实现了 operation3");
}

public void operation4() {
    System.out.println("D 实现了 operation4");
}

public void operation5()
{
    System.out.println("D 实现了
operation5");
}

}

class A { //A 类通过接口 Interface1 依赖(使用) B 类, 但是只会用到 1,2,3 方法
    public void depend1(Interface1 i)
    {
        i.operation1();
    }

    public void depend2(Interface1 i)
    {
        i.operation2();
    }

    public void depend3(Interface1 i)
    {
        i.operation3();
    }
}
```



```
}
```

```
class C { //C 类通过接口 Interface1 依赖(使用) D 类, 但是只会用到 1,4,5 方法
```

```
    public void depend1(Interface1 i)
```

```
        { i.operation1();
```

```
    }
```

```
    public void depend4(Interface1 i)
```

```
        { i.operation4();
```

```
    }
```

```
    public void depend5(Interface1 i)
```

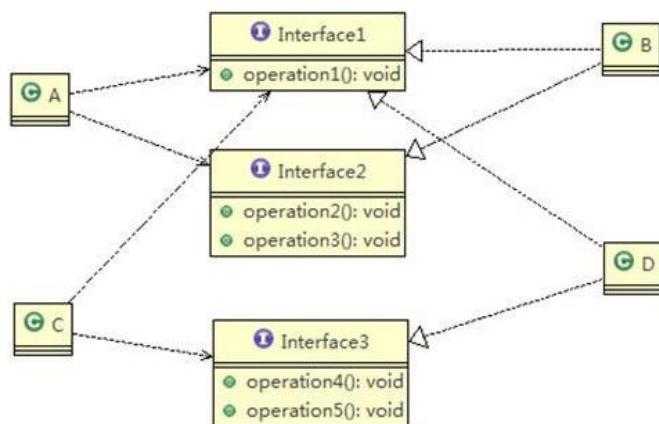
```
        { i.operation5();
```

```
    }
```

```
}
```

2.4.3 应传统方法的问题和使用接口隔离原则改进

- 1) 类 A 通过接口 Interface1 依赖类 B, 类 C 通过接口 Interface1 依赖类 D, 如果接口 Interface1 对于类 A 和类 C 来说不是最小接口, 那么类 B 和类 D 必须去实现他们不需要的方法
- 2) 将接口 **Interface1** 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
- 3) 接口 Interface1 中出现的方法, 根据实际情况拆分为三个接口



4) 代码实现

```
package com.atguigu.principle.segregation.improve;

public class Segregation1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // 使用一把
        A a = new A();
        a.depend1(new B()); // A 类通过接口去依赖 B 类 (重点)
        a.depend2(new B());
        a.depend3(new B());

        C c = new C();

        c.depend1(new D()); // C 类通过接口去依赖(使用)D 类
        c.depend4(new D());
        c.depend5(new D());
    }
}
```



```
}

}

// 接口 1
interface Interface1
{
    void
    operation1();
}

// 接口 2
interface Interface2
{
    void
    operation2();
    void operation3();
}

// 接口 3
interface Interface3
{
    void
    operation4();
    void operation5();
}
```



```
class B implements Interface1, Interface2 {
```



```
public void operation1() {
    System.out.println("B 实现了 operation1");
}

public void operation2() {
    System.out.println("B 实现了 operation2");
}

public void operation3() {
    System.out.println("B 实现了 operation3");
}

}

class D implements Interface1, Interface3
{
    public void operation1() {
        System.out.println("D 实现了 operation1");
    }

    public void operation4()
    {
        System.out.println("D 实现了
operation4");
    }

    public void operation5() {
        System.out.println("D 实现了 operation5");
    }
}
```



```
}
```

```
class A { // A 类通过接口 Interface1,Interface2 依赖(使用) B 类, 但是只会用到 1,2,3 方法
```

```
    public void depend1(Interface1 i)
```

```
        { i.operation1();
```

```
}
```

```
    public void depend2(Interface2 i)
```

```
        { i.operation2();
```

```
}
```

```
    public void depend3(Interface2 i)
```

```
        { i.operation3();
```

```
}
```

```
}
```

```
class C { // C 类通过接口 Interface1,Interface3 依赖(使用) D 类, 但是只会用到 1,4,5 方法
```

```
    public void depend1(Interface1 i)
```

```
        { i.operation1();
```

```
}
```

```
    public void depend4(Interface3 i)
```

```
        { i.operation4();
```

```
}
```

```
    public void depend5(Interface3 i) {
```



```
i.operation5();  
}  
}
```

2.5 依赖倒转原则

2.5.1 基本介绍

依赖倒转原则(Dependence Inversion Principle)是指：

- 1) 高层模块不应该依赖低层模块，二者都应该依赖其抽象
- 2) 抽象不应该依赖细节，细节应该依赖抽象
- 3) 依赖倒转(倒置)的中心思想是面向接口编程
- 4) 依赖倒转原则是基于这样的设计理念：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建的架构比以细节为基础的架构要稳定的多。在 java 中，抽象指的是接口或抽象类，细节就是具体的实现类
- 5) 使用接口或抽象类的目的是制定好规范，而不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

2.5.2 应用实例

请编程完成 Person 接收消息的功能。

- 1) 实现方案 1 + 分析说明

```
package com.atguigu.principle.inversion;  
  
public class DependecyInversion {
```



```
public static void main(String[] args)
{
    Person person = new Person();
    person.receive(new Email());
}

}

class Email {
    public String getInfo() {
        return "电子邮件信息: hello,world";
    }
}

//完成 Person 接收消息的功能
//方式 1 分析
//1. 简单，比较容易想到
//2. 如果我们获取的对象是 微信，短信等等，则新增类，同时 Perons 也要增加相应的接收方法
//3. 解决思路：引入一个抽象的接口 IReceiver，表示接收者，这样 Person 类与接口 IReceiver 发生依赖
// 因为 Email, WeiXin 等等属于接收的范围，他们各自实现 IReceiver 接口就 ok，这样我们就符合依赖倒转原则
class Person {
    public void receive(Email email )
    {
        System.out.println(email.getInfo());
    }
}
```



2) 实现方案 2(依赖倒转) + 分析说明

```
package com.atguigu.principle.inversion.improve;

public class DependecyInversion {

    public static void main(String[] args) {
        //客户端无需改变
        Person person = new Person();
        person.receive(new Email());

        person.receive(new WeiXin());

    }

}

//定义接口
interface IReceiver
{
    public String
    getInfo();
}

class Email implements IReceiver
{
    public String getInfo()
    {
        return "电子邮件信息: hello,world";
    }
}
```



```
//增加微信
class WeiXin implements IReceiver
{
    public String getInfo()
    {
        return "微信信息: hello,ok";
    }
}

//方式 2
class Person {
    //这里我们是对接口的依赖
    public void receive(IReceiver receiver)
    {
        System.out.println(receiver.getInfo());
    }
}
```

2.5.3 依赖关系传递的三种方式和应用案例

1) 接口传递

应用案例代码

2) 构造方法传递

应用案例代码

3) setter 方式传递



应用案例代码

4) 代码演示

```
package com.atguigu.principle.inversion.improve;

public class DependencyPass {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ChangHong changHong = new ChangHong();
        // OpenAndClose openAndClose = new OpenAndClose();
        // openAndClose.open(changHong);

        //通过构造器进行依赖传递
        // OpenAndClose openAndClose = new OpenAndClose(changHong);
        // openAndClose.open();
        //通过 setter 方法进行依赖传递
        OpenAndClose openAndClose = new OpenAndClose();
        openAndClose.setTv(changHong);
        openAndClose.open();

    }

}

// 方式 1： 通过接口传递实现依赖
// 开关的接口
```



```
// interface IOpenAndClose {  
//     public void open(ITV tv); //抽象方法,接收接口  
// }  
  
//  
// interface ITV { //ITV 接口  
//     public void play();  
// }  
  
//  
  
// class ChangHong implements ITV {  
//  
//     @Override  
//     public void play() {  
//         // TODO Auto-generated method stub  
//         System.out.println("长虹电视机， 打开");  
//     }  
  
//  
  
// }  
/// 实现接口  
// class OpenAndClose implements IOpenAndClose{  
//     public void open(ITV tv){  
//         tv.play();  
//     }  
// }  
  
// 方式 2: 通过构造方法依赖传递  
// interface IOpenAndClose {
```



```
// public void open(); //抽象方法
// }
// interface ITV { //ITV 接口
// public void play();
// }

// class OpenAndClose implements IOpenAndClose{
// public ITV tv; //成员
// public OpenAndClose(ITV tv){ //构造器
// this.tv = tv;
// }
// public void open(){
// this.tv.play();
// }
// }
```

```
// 方式 3 , 通过 setter 方法传递
interface IOpenAndClose {
    public void open(); // 抽象方法
```

```
    public void setTv(ITV tv);
}
```

```
interface ITV { // ITV 接口
    public void play();
}
```



```
class OpenAndClose implements IOpenAndClose
{
    private ITV tv;

    public void setTv(ITV tv)
    {
        this.tv = tv;
    }

    public void open()
    {
        this.tv.play();
    }
}

class ChangHong implements ITV {
    @Override
    public void play() {
        // TODO Auto-generated method stub
        System.out.println("长虹电视机， 打开");
    }
}
```

2.5.4 依赖倒转原则的注意事项和细节



- 1) 低层模块尽量都要有抽象类或接口，或者两者都有，程序稳定性更好。
- 2) 变量的声明类型尽量是抽象类或接口，这样我们的变量引用和实际对象间，就存在一个缓冲层，利于程序扩展和优化
- 3) 继承时遵循里氏替换原则

2.6 里氏替换原则

2.6.1 OO 中的继承性的思考和说明

- 1) 继承包含这样一层含义：父类中凡是已经实现好的方法，实际上是在设定规范和契约，虽然它不强制要求所有的子类必须遵循这些契约，但是如果子类对这些已经实现的方法任意修改，就会对整个继承体系造成破坏。
- 2) 继承在给程序设计带来便利的同时，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产生故障
- 3) 问题提出：在编程中，如何正确的使用继承? => 里氏替换原则

2.6.2 基本介绍

- 1) 里氏替换原则(Liskov Substitution Principle)在 1988 年，由麻省理工学院的以为姓里的女士提出的。
- 2) 如果对每个类型为 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。换句话说，所有引用基类的地方必须能透明地使用其子类的对象。
- 3) 在使用继承时，遵循里氏替换原则，在子类中尽量不要重写父类的方法
- 4) 里氏替换原则告诉我们，继承实际上让两个类耦合性增强了，在适当的情况下，可以通过聚合，组合，依赖来解决问题。



2.6.3一个程序引出的问题和思考

该看个程序，思考下问题和解决思路

```
package com.atguigu.principle.liskov;

public class Liskov {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        A a = new A();
        System.out.println("11-3=" + a.func1(11, 3));
        System.out.println("1-8=" + a.func1(1, 8));

        System.out.println("-----");
        B b = new B();
        System.out.println("11-3=" + b.func1(11, 3));//这里本意是求出 11-3 (本意)
        System.out.println("1-8=" + b.func1(1, 8));// 1-8
        System.out.println("11+3+9=" + b.func2(11, 3));

    }
}
```



```
// A 类
class A {
    // 返回两个数的差
    public int func1(int num1, int num2)
    {
        return num1 - num2;
    }
}

// B 类继承了 A
// 增加了一个新功能：完成两个数相加,然后和 9 求和
class B extends A {
    //这里，重写了 A 类的方法，可能是无意识
    public int func1(int a, int b)
    {
        return a + b;
    }

    public int func2(int a, int b)
    {
        return func1(a, b) + 9;
    }
}
```

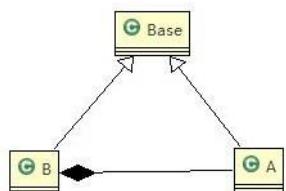
2.6.4 解决方法

- 1) 我们发现原来运行正常的相减功能发生了错误。原因就是类 B 无意中重写了父类的方法，造成原有功能出现错误。在实际编程中，我们常常会通过重写父类的方法完成新的功能，这样写起来虽然简单，但整个继承体系的

复用性会比较差。特别是运行多态比较频繁的时候

2) 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。

3) 改进方案



代码实现

```
package com.atguigu.principle.liskov.improve;

public class Liskov {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        A a = new A();
        System.out.println("11-3=" + a.func1(11, 3));
        System.out.println("1-8=" + a.func1(1, 8));

        System.out.println("-----");
        B b = new B();
        //因为 B 类不再继承 A 类，因此调用者，不会再 func1 是求减法
        //调用完成的功能就会很明确
        System.out.println("11+3=" + b.func1(11, 3)); //这里本意是求出 11+3
    }
}
```



```
System.out.println("1+8=" + b.func1(1, 8));// 1+8  
System.out.println("11+3+9=" + b.func2(11, 3));  
  
//使用组合仍然可以使用到 A 类相关方法  
System.out.println("11-3=" + b.func3(11, 3));// 这里本意是求出 11-3
```

```
}
```

```
}
```

```
//创建一个更加基础的基类
```

```
class Base {  
    //把更加基础的方法和成员写到 Base 类  
}
```

```
// A 类
```

```
class A extends Base {  
    // 返回两个数的差  
    public int func1(int num1, int num2)  
    { return num1 - num2;  
    }  
}
```

```
// B 类继承了 A
```



```
// 增加了一个新功能：完成两个数相加,然后和 9 求和
```

```
class B extends Base {
```

```
    //如果 B 需要使用 A 类的方法,使用组合关系
```

```
    private A a = new A();
```

```
//这里，重写了 A 类的方法，可能是无意识
```

```
public int func1(int a, int b)
```

```
    { return a + b;
```

```
}
```

```
public int func2(int a, int b)
```

```
    { return func1(a, b) + 9;
```

```
}
```

```
//我们仍然想使用 A 的方法
```

```
public int func3(int a, int b)
```

```
    { return this.a.func1(a,
```

```
        b);
```

```
}
```

```
}
```

2.7 开闭原则

2.7.1 基本介绍

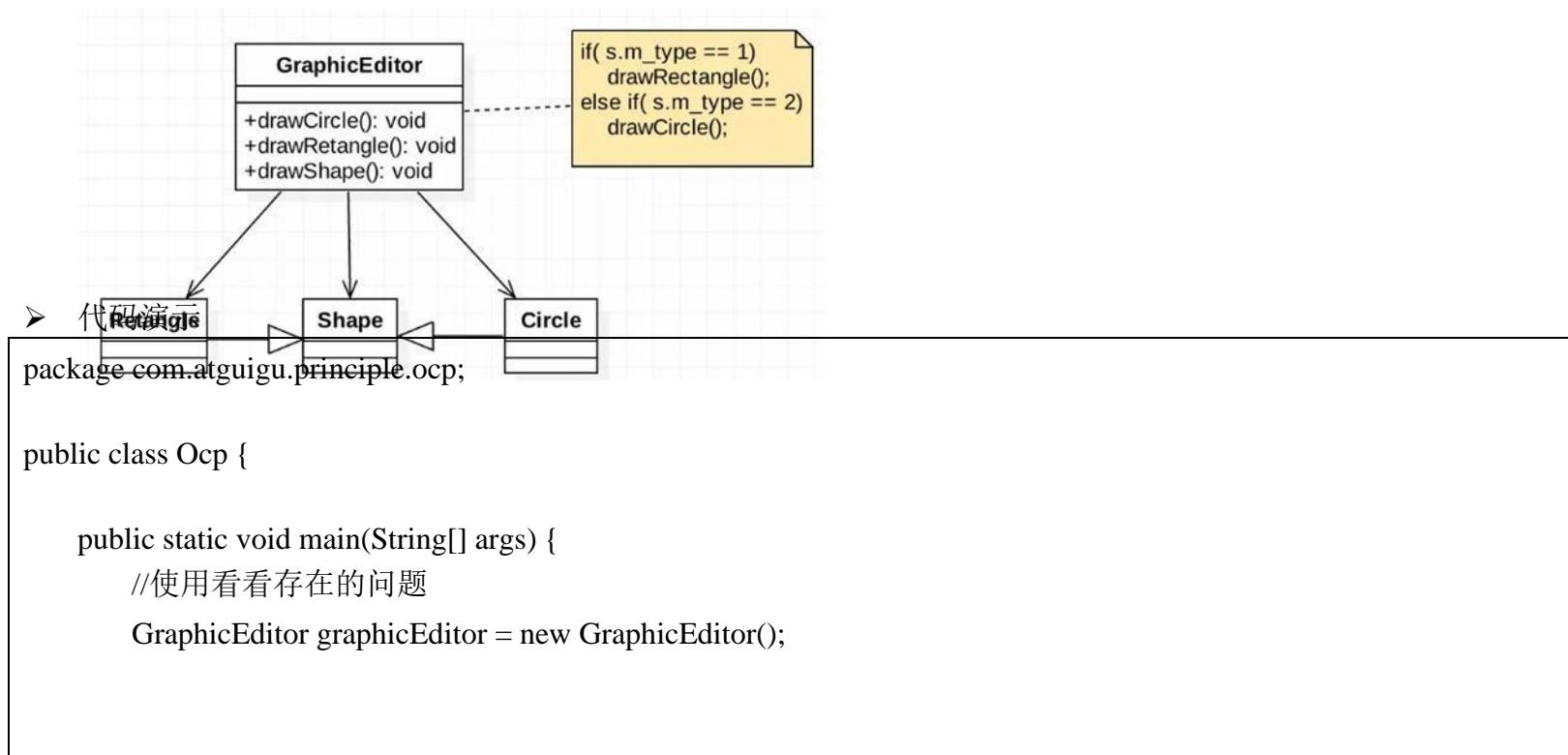
- 1) 开闭原则（Open Closed Principle）是编程中最基础、最重要的设计原则

- 2) 一个软件实体如类，模块和函数应该对扩展开放(对提供方)，对修改关闭(对使用方)。用抽象构建框架，用实现扩展细节。
- 3) 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。
- 4) 编程中遵循其它原则，以及使用设计模式的目的就是遵循开闭原则。

2.7.2 看下面一段代码

➤ 看一个画图形的功能。

类图设计，如下：





```
graphicEditor.drawShape(new Rectangle());
graphicEditor.drawShape(new Circle());
graphicEditor.drawShape(new Triangle());
}

}

//这是一个用于绘图的类 [使用方]
class GraphicEditor {
    //接收 Shape 对象，然后根据 type，来绘制不同的图形
    public void drawShape(Shape s) {
        if (s.m_type == 1)
            drawRectangle(s);
        else if (s.m_type == 2)
            drawCircle(s);
        else if (s.m_type == 3)
            drawTriangle(s);
    }

    //绘制矩形
    public void drawRectangle(Shape r)
    { System.out.println(" 绘制矩形 ");
    }

    //绘制圆形
    public void drawCircle(Shape r) {
```



```
System.out.println(" 绘制圆形 ");
}

//绘制三角形
public void drawTriangle(Shape r)
{ System.out.println(" 绘制三角形 ");
}

//Shape 类, 基类
class Shape {

    int m_type;
}

class Rectangle extends Shape
{
    Rectangle() {
        super.m_type = 1;
    }
}

class Circle extends Shape
{
    Circle() {
        super.m_type = 2;
    }
}
```



```
//新增画三角形  
class Triangle extends Shape  
{ Triangle()  
    super.m_type = 3;  
}  
}
```

2.7.3 方式1 的优缺点

- 1) 优点是比较好理解，简单易操作。
- 2) 缺点是违反了设计模式的 ocp 原则，即对扩展开放(提供方)，对修改关闭(使用方)。即当我们给类增加新功能的时候，尽量不修改代码，或者尽可能少修改代码。
- 3) 比如我们这时要新增加一个图形种类 三角形，我们需要做如下修改，修改的地方较多
- 4) 代码演示

方式 1 的改进的思路分析

2.7.4 改进的思路分析

思路：把创建 **Shape** 类做成抽象类，并提供一个抽象的 **draw** 方法，让子类去实现即可，这样我们有新的图形种类时，只需要让新的图形类继承 **Shape**，并实现 **draw** 方法即可，使用方的代码就不需要修改 -> 满足了开闭原则

改进后的代码：

```
package com.atguigu.principle.ocp.improve;  
  
public class Ocp {
```



```
public static void main(String[] args) {
    //使用看看存在的问题
    GraphicEditor graphicEditor = new GraphicEditor();
    graphicEditor.drawShape(new Rectangle());
    graphicEditor.drawShape(new Circle());
    graphicEditor.drawShape(new Triangle());
    graphicEditor.drawShape(new OtherGraphic());
}

}

//这是一个用于绘图的类 [使用方]
class GraphicEditor {
    //接收 Shape 对象，调用 draw 方法
    public void drawShape(Shape s)
    {
        s.draw();
    }
}

}

//Shape 类，基类
abstract class Shape
{
    int m_type;
}
```



```
public abstract void draw();//抽象方法  
}
```

```
class Rectangle extends Shape  
{ Rectangle()  
    super.m_type = 1;  
  
}  
  
@Override  
public void draw()  
{  
    // TODO Auto-generated method stub  
    System.out.println(" 绘制矩形 ");  
}  
}
```

```
class Circle extends Shape  
{ Circle()  
    super.m_type = 2;  
  
}  
  
@Override  
public void draw()  
{  
    // TODO Auto-generated method stub  
    System.out.println(" 绘制圆形 ");  
}  
}
```



```
//新增画三角形
class Triangle extends Shape
{
    Triangle()
    {
        super.m_type = 3;
    }
    @Override
    public void draw()
    {
        // TODO Auto-generated method stub
        System.out.println(" 绘制三角形 ");
    }
}

//新增一个图形
class OtherGraphic extends Shape
{
    OtherGraphic()
    {
        super.m_type = 4;
    }
    @Override
    public void draw()
    {
        // TODO Auto-generated method stub
        System.out.println(" 绘制其它图形 ");
    }
}
```

2.8 迪米特法则

2.8.1 基本介绍

- 1) 一个对象应该对其他对象保持最少的了解
- 2) 类与类关系越密切，耦合度越大
- 3) 迪米特法则(Demeter Principle)又叫最少知道原则，即一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类不管多么复杂，都尽量将逻辑封装在类的内部。对外除了提供的 public 方法，不对外泄露任何信息
- 4) 迪米特法则还有一个更简单的定义：只与直接的朋友通信
- 5) 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不是直接的朋友。也就是说，陌生的类最好不要以局部变量的形式出现在类的内部。

2.8.2 应用实例

- 1) 有一个学校，下属有各个学院和总部，现要求打印出学校总部员工 ID 和学院员工的 id
- 2) 编程实现上面的功能，看代码演示
- 3) 代码演示

```
package com.atguigu.principle.demeter;

import java.util.ArrayList;
import java.util.List;

//客户端
```



```
public class Demeter1 {  
  
    public static void main(String[] args) {  
        //创建了一个 SchoolManager 对象  
        SchoolManager schoolManager = new SchoolManager();  
        //输出学院的员工 id 和 学校总部的员工信息  
        schoolManager.printAllEmployee(new CollegeManager());  
  
    }  
  
}  
  
  
//学校总部员工类  
class Employee {  
    private String id;  
  
    public void setId(String id)  
    { this.id = id;  
    }  
  
    public String getId()  
    { return id;  
    }  
}
```



```
//学院的员工类
class CollegeEmployee
{
    private String id;

    public void setId(String id)
    {
        this.id = id;
    }

    public String getId()
    {
        return id;
    }
}

//管理学院员工的管理类
class CollegeManager {
    //返回学院的所有员工
    public List<CollegeEmployee> getAllEmployee()
    {
        List<CollegeEmployee> list = new
        ArrayList<CollegeEmployee>();

        for (int i = 0; i < 10; i++) { //这里我们增加了 10 个员工到 list
            CollegeEmployee emp = new CollegeEmployee();
            emp.setId("学院员工 id= " + i);
            list.add(emp);
        }

        return list;
    }
}
```



```
}

}

//学校管理类

//分析 SchoolManager 类的直接朋友类有哪些 Employee、CollegeManager
//CollegeEmployee 不是 直接朋友 而是一个陌生类，这样违背了 迪米特法则
class SchoolManager {
    //返回学校总部的员工
    public List<Employee> getAllEmployee()
    { List<Employee> list = new ArrayList<Employee>();

        for (int i = 0; i < 5; i++) { //这里我们增加了 5 个员工到 list
            Employee emp = new Employee();
            emp.setId("学校总部员工 id= " + i);
            list.add(emp);
        }
        return list;
    }

    //该方法完成输出学校总部和学院员工信息(id)
    void printAllEmployee(CollegeManager sub) {

        //分析问题
        //1. 这里的 CollegeEmployee 不是 SchoolManager 的直接朋友
        //2. CollegeEmployee 是以局部变量方式出现在 SchoolManager
    }
}
```



```
//3. 违反了 迪米特法则
```

```
//获取到学院员工
List<CollegeEmployee> list1 = sub.getAllEmployee();
System.out.println("-----学院员工-----");
for (CollegeEmployee e : list1) {
    System.out.println(e.getId());
}

//获取到学校总部员工
List<Employee> list2 = this.getAllEmployee();
System.out.println("-----学校总部员工-----");
for (Employee e : list2) {
    System.out.println(e.getId());
}
```

2.8.3 应用实例改进

- 1) 前面设计的问题在于 SchoolManager 中, CollegeEmployee 类并不是 SchoolManager 类的直接朋友 (分析)
- 2) 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
- 3) 对代码按照迪米特法则 进行改进. (看老师演示)
- 4) 代码演示

```
package com.atguigu.principle.demeter.improve;
```



```
import java.util.ArrayList;
import java.util.List;

//客户端
public class Demeter1 {

    public static void main(String[] args) {
        System.out.println("~~~使用迪米特法则的改进~~~");
        //创建了一个 SchoolManager 对象
        SchoolManager schoolManager = new SchoolManager();
        //输出学院的员工 id 和 学校总部的员工信息
        schoolManager.printAllEmployee(new CollegeManager());

    }

}

//学校总部员工类
class Employee {
    private String id;

    public void setId(String id)
    {
        this.id = id;
    }
}
```



```
public String getId()
{
    return id;
}

}

//学院的员工类
class CollegeEmployee
{
    private String id;

    public void setId(String id)
    {
        this.id = id;
    }

    public String getId()
    {
        return id;
    }
}

//管理学院员工的管理类
class CollegeManager {
    //返回学院的所有员工
    public List<CollegeEmployee> getAllEmployee() {
        List<CollegeEmployee> list = new ArrayList<CollegeEmployee>();
    }
}
```



```
for (int i = 0; i < 10; i++) { //这里我们增加了 10 个员工到 list
    CollegeEmployee emp = new CollegeEmployee();
    emp.setId("学院员工 id= " + i);
    list.add(emp);
}

return list;
}

//输出学院员工的信息
public void printEmployee() {
    //获取到学院员工
    List<CollegeEmployee> list1 = getAllEmployee();
    System.out.println("-----学院员工-----");
    for (CollegeEmployee e : list1) {
        System.out.println(e.getId());
    }
}

}

//学校管理类

//分析 SchoolManager 类的直接朋友类有哪些 Employee、CollegeManager
//CollegeEmployee 不是 直接朋友 而是一个陌生类，这样违背了 迪米特法则
class SchoolManager {
    //返回学校总部的员工
    public List<Employee> getAllEmployee() {
```



```
List<Employee> list = new ArrayList<Employee>();  
  
for (int i = 0; i < 5; i++) { //这里我们增加了 5 个员工到 list  
    Employee emp = new Employee();  
    emp.setId("学校总部员工 id= " + i);  
    list.add(emp);  
}  
  
return list;  
}
```

```
//该方法完成输出学校总部和学院员工信息(id)  
void printAllEmployee(CollegeManager sub) {
```

```
//分析问题  
//1. 将输出学院的员工方法，封装到 CollegeManager  
sub.printEmployee();
```

```
//获取到学校总部员工  
List<Employee> list2 = this.getAllEmployee();  
System.out.println("-----学校总部员工-----");  
for (Employee e : list2) {  
    System.out.println(e.getId());  
}  
}
```

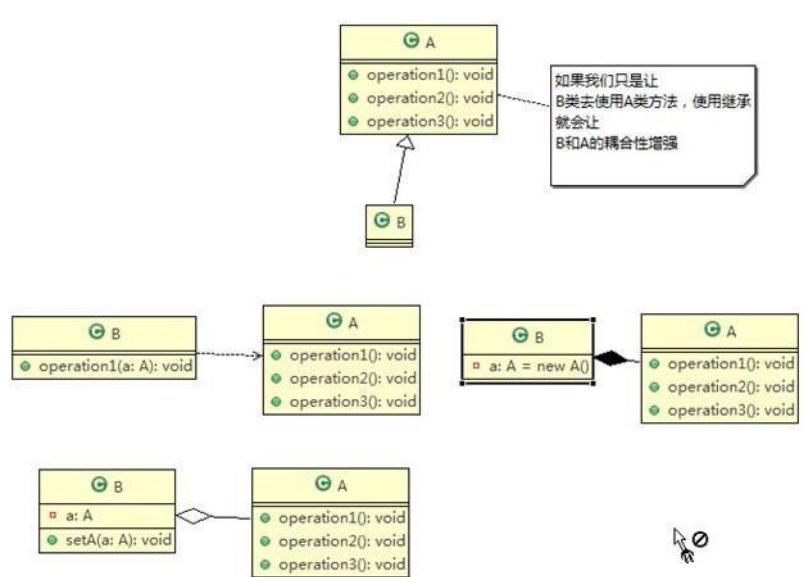
2.8.4 迪米特法则注意事项和细节

- 1) 迪米特法则的核心是降低类之间的耦合
- 2) 但是注意：由于每个类都减少了不必要的依赖，因此迪米特法则只是要求降低类间(对象间)耦合关系，并不是要求完全没有依赖关系

2.9 合成复用原则 (Composite Reuse Principle)

2.9.1 基本介绍

原则是尽量使用合成/聚合的方式，而不是使用继承



2.10 设计原则核心思想

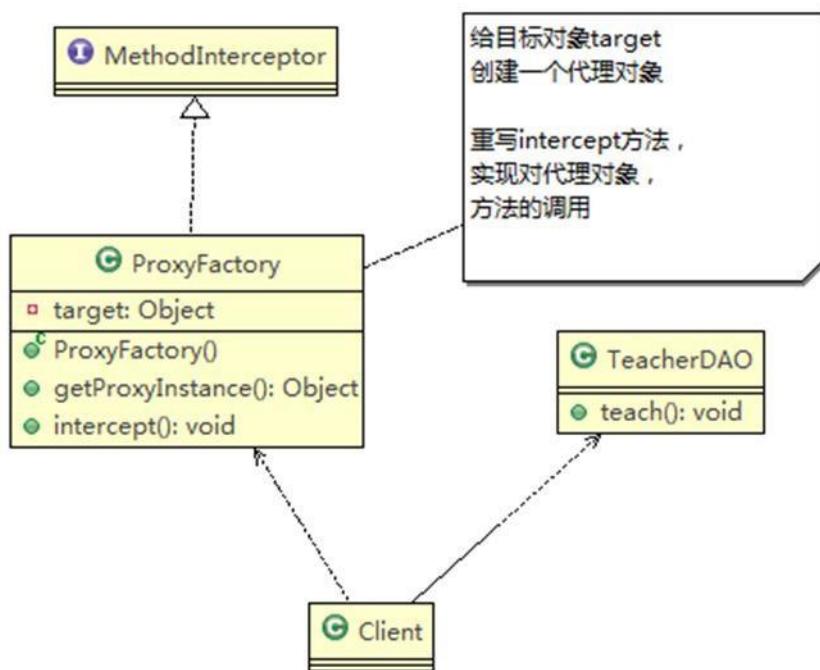


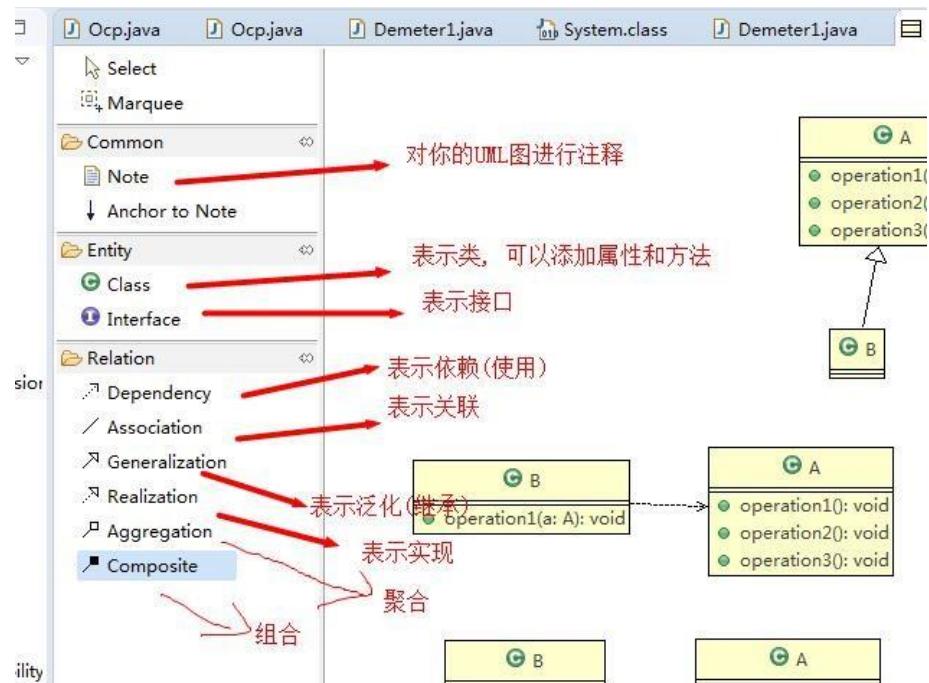
- 1) 找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。
- 2) 针对接口编程，而不是针对实现编程。
- 3) 为了交互对象之间的松耦合设计而努力

第3 章 UML 类图

3.1 UML 基本介绍

- 1) UML——Unified modeling language UML (统一建模语言), 是一种用于软件系统分析和设计的语言工具, 它用于帮助软件开发人员进行思考和记录思路的结果
- 2) UML 本身是一套符号的规定, 就像数学符号和化学符号一样, 这些符号用于描述软件模型中的各个元素和他们之间的关系, 比如类、接口、实现、泛化、依赖、组合、聚合等, 如右图:





- 3) 使用 UML 来建模，常用的工具有 Rational Rose，也可以使用一些插件来建模

Eclipse 安装 UML 插件 (AmaterasUML).zip

AmaterasUML_1.3.4.rar



3.2 UML 图

画 UML 图与写文章差不多，都是把自己的思想描述给别人看，关键在于思路和条理，UML 图分类：

- 1) 用例图(use case)
- 2) 静态结构图：类图、对象图、包图、组件图、部署图
- 3) 动态行为图：交互图（时序图与协作图）、状态图、活动图



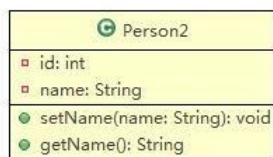
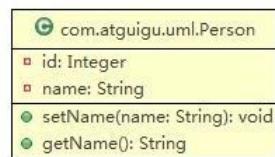
➤ 说明：

- 1) 类图是描述类与类之间的关系的，是 UML 图中最核心的
- 2) 在讲解设计模式时，我们必然会使用类图，为了让学员们能够把设计模式学到位，需要先给大家讲解类图
- 3) 温馨提示：如果已经掌握 UML 类图的学员，可以直接听设计模式的章节

3.3 UML 类图

- 1) 用于描述系统中的类(对象)本身的组成和类(对象)之间的各种静态关系。
- 2) 类之间的关系：依赖、泛化（继承）、实现、关联、聚合与组合。
- 3) 类图简单举例

```
public class Person{ //代码形式->类图
    private Integer id;
    private String name;
    public void setName(String
        name){ this.name=name;
    }
    public String
        getName(){ return
            name;
    }
}
```



3.4 类图—依赖关系 (Dependence)

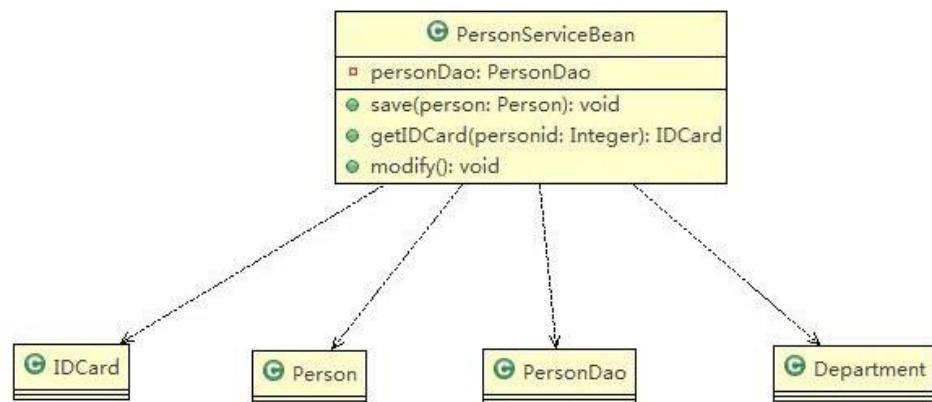
➤ 只要是在类中用到了对方，那么他们之间就存在依赖关系。如果没有对方，连编绎都通过不了。

```
public class PersonServiceBean
{
    private PersonDao personDao;//
    public void save(Person
person){}
    public IDCard getIDCard(Integer personid){}
}
```

```
public void modify(){
    Department department = new Department();
}
}
```

```
public class PersonDao{}
public class IDCard{}
public class Person{}
public class Department{}
```

➤ 对应的类图：



➤ 小结

- 1) 类中用到了对方
- 2) 如果是类的成员属性
- 3) 如果是方法的返回类型
- 4) 是方法接收的参数类型
- 5) 方法中使用到

3.5 类图—泛化关系(generalization)

➤ 泛化关系实际上就是继承关系，他是依赖关系的特例

public abstract class

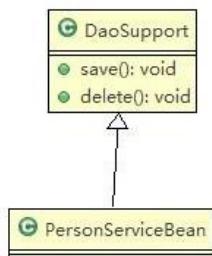
```

DaoSupport{ public void
    save(Object entity){
    }
    public void delete(Object id){
    }
}
  
```

```

public class PersonServiceBean extends DaoSupport{
}
  
```

➤ 对应的类图



➤ 小结:

- 1) 泛化关系实际上就是继承关系
- 2) 如果 A 类继承了 B 类, 我们就说 A 和 B 存在泛化关系

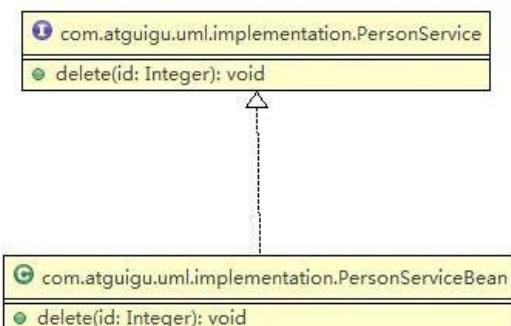
3.6 类图—实现关系 (Implementation)

实现关系实际上就是 **A** 类实现 **B** 接口, 他是依赖关系的特例

```
public interface PersonService
{
    public void delete(Integer id);
}

public class PersonServiceBean implements PersonService
{
    public void delete(Integer id){}
}
```

=>类图



小结:

3.7 类图—关联关系 (Association)

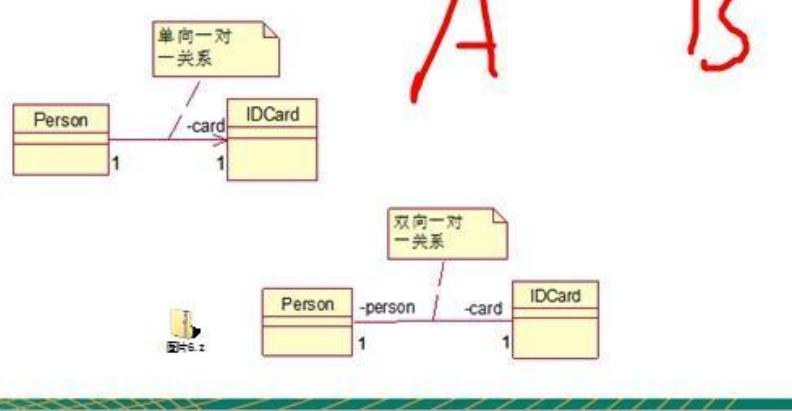
关联关系实际上就是类与类之间的联系，他是依赖关系的特例

关联具有导航性：即双向关系或单向关系

关系具有多重性：如“1”（表示有且仅有一个），“0...”（表示0个或者多个），“0, 1”（表示0个或者一个），“n...m”（表示n到m个都可以），“m...*”（表示至少m个）。

单向一对一关系

```
public class Person {
    private IDCard card;
}
public class IDCard{}
```



3.8 类图—聚合关系 (Aggregation)

3.8.1 基本介绍

聚合关系 (Aggregation) 表示的是整体和部分的关系，整体与部分可以分开。聚合关系是关联关系的特例，所以他具有关联的导航性与多重性。

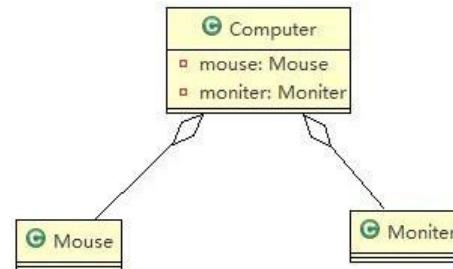
如：一台电脑由键盘(keyboard)、显示器(monitor)，鼠标等组成；组成电脑的各个配件是可以从电脑上分离出来的，使用带空心菱形的实线来表示：

3.8.2 应用实例

```
public class Computer{
    private Mouse mouse;
    private Monitor monitor;

    public void setMouse(Mouse mouse){
        this.mouse = mouse;
    }

    public void setMonitor(Monitor monitor){
        this.monitor = monitor;
    }
}
```



3.9 类图—组合关系（Composition）

3.9.1 基本介绍

组合关系：也是整体与部分的关系，但是整体与部分不可以分开。

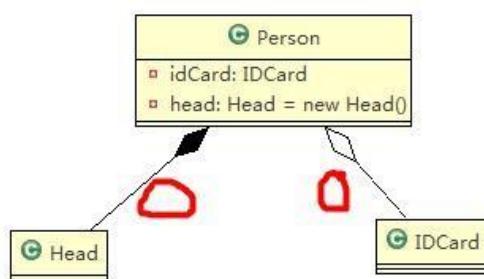
再看一个案例：在程序中我们定义实体：Person 与 IDCard、Head，那么 Head 和 Person 就是组合，IDCard 和 Person 就是聚合。

但是如果在程序中 Person 实体中定义了对 IDCard 进行级联删除，即删除 Person 时连同 IDCard 一起删除，那么 IDCard 和 Person 就是组合了。

3.9.2 应用案例

```
public class Person{  
    private IDCard card;  
    private Head head = new Head();  
}  
  
public class IDCard{}  
  
public class Head{}
```

对应的类图：

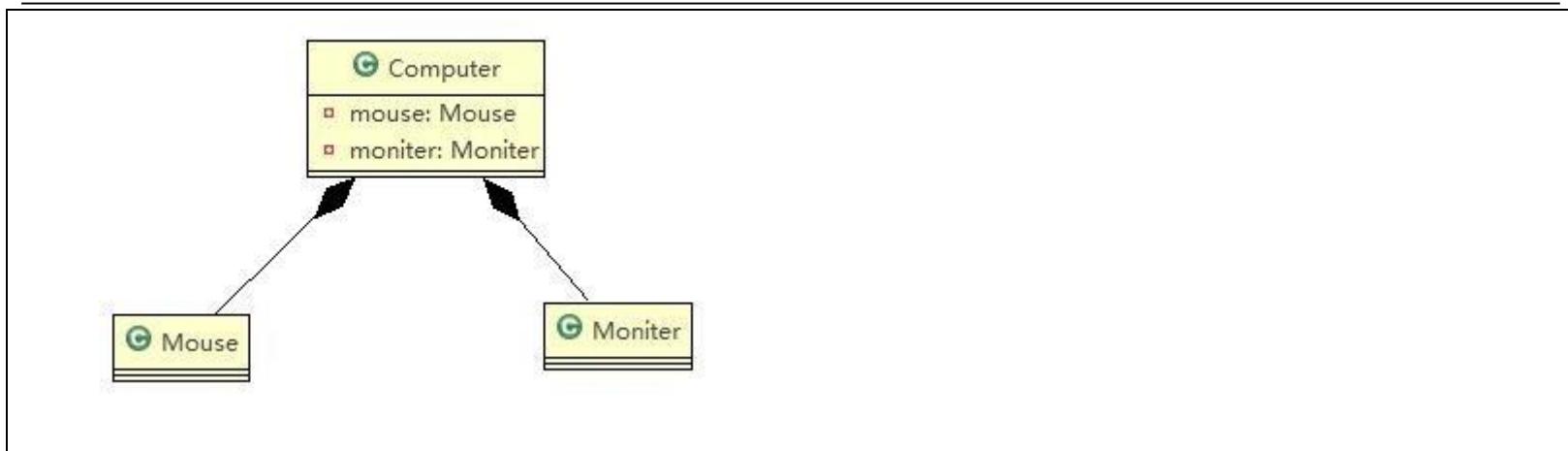


案例 2



```
public class Computer {  
    private Mouse mouse = new Mouse(); //鼠标可以和 computer 不能分离private  
    Moniter moniter = new Moniter(); //显示器可以和 Computer 不能分离public  
    void setMouse(Mouse mouse) {  
        this.mouse = mouse;  
    }  
    public void setMoniter(Moniter moniter)  
    { this.moniter = moniter;  
    }  
}  
public class Mouse {  
}  
public class Moniter {  
}
```

对应的类图





第 4 章 设计模式概述

4.1 掌握设计模式的层次

- 1) 第 1 层：刚开始学编程不久，听说过什么是设计模式
- 2) 第 2 层：有很长时间的编程经验，自己写了很多代码，其中用到了设计模式，但是自己却不知道
- 3) 第 3 层：学习过了设计模式，发现自己已经在使用了，并且发现了一些新的模式挺好用的
- 4) 第 4 层：阅读了很多别人写的源码和框架，在其中看到别人设计模式，并且能够领会设计模式的精妙和带来的好处。
- 5) 第 5 层：代码写着写着，自己都没有意识到使用了设计模式，并且熟练的写了出来。

4.2 设计模式介绍

- 1) 设计模式是程序员在面对同类软件工程设计问题所总结出来的有用的经验，模式不是代码，而是某类问题的通用解决方案，设计模式（Design pattern）代表了最佳的实践。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。
- 2) 设计模式的本质提高软件的维护性，通用性和扩展性，并降低软件的复杂度。
- 3) <<设计模式>>是经典的书，作者是 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides Design（俗称“四人组 GOF”）
- 4) 设计模式并不局限于某种语言，java, php, c++ 都有设计模式.

4.3 设计模式类型

设计模式分为三种类型，共 **23** 种

- 1) 创建型模式：单例模式、抽象工厂模式、原型模式、建造者模式、工厂模式。



- 2) 结构型模式：适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式。
- 3) 行为型模式：模版方法模式、命令模式、访问者模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式（Interpreter 模式）、状态模式、策略模式、职责链模式(责任链模式)。

注意：不同的书籍上对分类和名称略有差别



第 5 章 单例设计模式

5.1 单例设计模式介绍

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法(静态方法)。

比如 Hibernate 的 SessionFactory，它充当数据存储源的代理，并负责创建 Session 对象。SessionFactory 并不是轻量级的，一般情况下，一个项目通常只需要一个 SessionFactory 就够，这是就会使用到单例模式。

5.2 单例设计模式八种方式

单例模式有八种方式：

- 1) 饿汉式(静态常量)
- 2) 饿汉式 (静态代码块)
- 3) 懒汉式(线程不安全)
- 4) 懒汉式(线程安全，同步方法)
- 5) 懒汉式(线程安全，同步代码块)
- 6) 双重检查
- 7) 静态内部类
- 8) 枚举

5.3 饿汉式（静态常量）

饿汉式（静态常量）应用实例
步骤如下：



- 1) 构造器私有化(防止 new)
- 2) 类的内部创建对象
- 3) 向外暴露一个静态的公共方法。getInstance
- 4) 代码实现

```
package com.atguigu.singleton.type1;

public class SingletonTest01 {

    public static void main(String[] args) {
        //测试
        Singleton instance = Singleton.getInstance();
        Singleton instance2 = Singleton.getInstance();
        System.out.println(instance == instance2); // true
        System.out.println("instance.hashCode=" + instance.hashCode());
        System.out.println("instance2.hashCode=" + instance2.hashCode());
    }
}

//饿汉式(静态变量)

class Singleton {

    //1. 构造器私有化, 外部不new
    private Singleton() {
```

```
}

//2.本类内部创建对象实例
private final static Singleton instance = new Singleton();

//3. 提供一个公有的静态方法，返回实例对象
public static Singleton getInstance()

    { return instance;

    }

}
```

➤ 优缺点说明：

- 1) 优点：这种写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题。
- 2) 缺点：在类装载的时候就完成实例化，没有达到 Lazy Loading 的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费
- 3) 这种方式基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，在单例模式中大多数都是调用 getInstance 方法，但是导致类装载的原因有很多种，因此不能确定有其他的方式（或者其他静态方法）导致类装载，这时候初始化 instance 就没有达到 lazy loading 的效果
- 4) 结论：这种单例模式可用，可能造成内存浪费

5.4 饿汉式（静态代码块）

➤ 代码演示：



```
package com.atguigu.singleton.type2;

public class SingletonTest02 {

    public static void main(String[] args) {
        //测试
        Singleton instance = Singleton.getInstance();
        Singleton instance2 = Singleton.getInstance();
        System.out.println(instance == instance2); // true
        System.out.println("instance.hashCode=" + instance.hashCode());
        System.out.println("instance2.hashCode=" + instance2.hashCode());
    }
}

//饿汉式(静态变量)

class Singleton {

    //1. 构造器私有化, 外部不new
    private Singleton() {

    }

    //2.本类内部创建对象实例
}
```



```
private static Singleton instance;

static { // 在静态代码块中，创建单例对象
    instance = new Singleton();
}

//3. 提供一个公有的静态方法，返回实例对象
public static Singleton getInstance()

{
    return instance;
}

}
```

➤ 优缺点说明：

- 1) 这种方式和上面的方式其实类似，只不过将类实例化的过程放在了静态代码块中，也是在类装载的时候，就执行静态代码块中的代码，初始化类的实例。优缺点和上面是一样的。
- 2) 结论：这种单例模式可用，但是可能造成内存浪费

5.5 懒汉式(线程不安全)

➤ 代码演示：

```
package com.atguigu.singleton.type3;

public class SingletonTest03 {
```



```
public static void main(String[] args) {  
    System.out.println("懒汉式 1 , 线程不安全~");  
    Singleton instance = Singleton.getInstance();  
    Singleton instance2 = Singleton.getInstance();  
    System.out.println(instance == instance2); // true  
    System.out.println("instance.hashCode=" + instance.hashCode());  
    System.out.println("instance2.hashCode=" + instance2.hashCode());  
}  
  
}  
  
class Singleton {  
    private static Singleton instance;  
  
    private Singleton() {}  
  
    //提供一个静态的公有方法，当使用到该方法时，才去创建 instance  
    //即懒汉式  
    public static Singleton getInstance()  
    { if(instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}  
}
```



➤ 优缺点说明：

- 1) 起到了 **Lazy Loading** 的效果，但是只能在单线程下使用。
- 2) 如果在多线程下，一个线程进入了 if (singleton == null) 判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。所以在多线程环境下不可使用这种方式
- 3) 结论：在实际开发中，不要使用这种方式。

5.6 懒汉式(线程安全，同步方法)

➤ 代码演示：

```
package com.atguigu.singleton.type4;

public class SingletonTest04 {

    public static void main(String[] args) {
        System.out.println("懒汉式 2 , 线程安全~");
        Singleton instance = Singleton.getInstance();
        Singleton instance2 = Singleton.getInstance();
        System.out.println(instance == instance2); // true
        System.out.println("instance.hashCode=" + instance.hashCode());
        System.out.println("instance2.hashCode=" + instance2.hashCode());
    }
}
```



```
// 懒汉式(线程安全，同步方法)
class Singleton {

    private static Singleton instance;

    private Singleton() {}

    //提供一个静态的公有方法，加入同步处理的代码，解决线程安全问题
    //即懒汉式
    public static synchronized Singleton getInstance()

    { if(instance == null) {

        instance = new Singleton();

    }

    return instance;

}

}
```

➤ 优缺点说明：

- 1) 解决了线程安全问题
- 2) 效率太低了，每个线程在想获得类的实例时候，执行 getInstance()方法都要进行同步。而其实这个方法只执行一次实例化代码就够了，后面的想获得该类实例，直接 return 就行了。方法进行同步效率太低
- 3) 结论：在实际开发中，不推荐使用这种方式

5.7 懒汉式(线程不安全，同步代码块)



```
class Singleton {  
    private static Singleton singleton;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (singleton == null) {  
            synchronized (Singleton.class) {  
                singleton = new Singleton();  
            }  
        }  
        return singleton;  
    }  
}
```

不推荐使用

5.8 双重检查

➤ 代码演示

```
package com.atguigu.singleton.type6;  
  
public class SingletonTest06 {  
  
    public static void main(String[] args) {  
        System.out.println("双重检查");  
        Singleton instance = Singleton.getInstance();  
  
        Singleton instance2 = Singleton.getInstance();  
  
        System.out.println(instance == instance2); // true  
        System.out.println("instance.hashCode=" + instance.hashCode());  
        System.out.println("instance2.hashCode=" + instance2.hashCode());  
    }  
}
```



```
}

}

// 懒汉式(线程安全, 同步方法)
class Singleton {

    private static volatile Singleton instance;

    private Singleton() {}

    //提供一个静态的公有方法, 加入双重检查代码, 解决线程安全问题, 同时解决懒加载问题
    //同时保证了效率, 推荐使用

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



➤ 优缺点说明：

- 1) Double-Check 概念是多线程开发中常使用到的，如代码中所示，我们进行了两次 if (singleton == null) 检查，这样就可以保证线程安全了。
- 2) 这样，实例化代码只用执行一次，后面再次访问时，判断 if (singleton == null)，直接 return 实例化对象，也避免的反复进行方法同步。
- 3) 线程安全；延迟加载；效率较高
- 4) 结论：在实际开发中，推荐使用这种单例设计模式

5.9 静态内部类

➤ 代码演示：

```
package com.atguigu.singleton.type7;

public class SingletonTest07 {

    public static void main(String[] args)
    { System.out.println("使用静态内部类完成单例模式
    "); Singleton instance = Singleton.getInstance();
    Singleton instance2 = Singleton.getInstance();
    System.out.println(instance == instance2); // true
    System.out.println("instance.hashCode=" + instance.hashCode());
    System.out.println("instance2.hashCode=" + instance2.hashCode());

    }
}
```



```
}

// 静态内部类完成， 推荐使用
class Singleton {

    private static volatile Singleton instance;

    //构造器私有化
    private Singleton() {}

    //写一个静态内部类,该类中有一个静态属性 Singleton
    private static class SingletonInstance {

        private static final Singleton INSTANCE = new Singleton();

    }

    //提供一个静态的公有方法， 直接返回 SingletonInstance.INSTANCE

    public static synchronized Singleton getInstance() {

        return SingletonInstance.INSTANCE;

    }

}
```

➤ 优缺点说明：

- 1) 这种方式采用了类装载的机制来保证初始化实例时只有一个线程。



- 2) 静态内部类方式在 Singleton 类被装载时并不会立即实例化，而是在需要实例化时，调用 getInstance 方法，才会装载 SingletonInstance 类，从而完成 Singleton 的实例化。
- 3) 类的静态属性只会在第一次加载类的时候初始化，所以在这里，**JVM 帮助我们保证了线程的安全性，在类进行初始化时，别的线程是无法进入的。**
- 4) 优点：避免了线程不安全，利用静态内部类特点实现延迟加载，效率高
- 5) 结论：推荐使用.

5.10 枚举

➤ 代码演示

```
package com.atguigu.singleton.type8;

public class SingletonTest08 {
    public static void main(String[] args) {
        Singleton instance =
            Singleton.INSTANCE;
        Singleton instance2 = Singleton.INSTANCE;
        System.out.println(instance == instance2);

        System.out.println(instance.hashCode());
        System.out.println(instance2.hashCode());

        instance.sayOK();
    }
}

//使用枚举，可以实现单例，推荐
enum Singleton {
```



```
INSTANCE; //属性  
public void sayOK()  
{ System.out.println("ok~  
");  
}  
}
```

➤ 优缺点说明：

- 1) 这借助 JDK1.5 中添加的枚举来实现单例模式。不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象。
- 2) 这种方式是 **Effective Java** 作者 **Josh Bloch** 提倡的方式
- 3) 结论：推荐使用

5.11 单例模式在 JDK 应用的源码分析

5.11.1 单例模式在 JDK 应用的源码分析

- 1) 我们 JDK 中，`java.lang.Runtime` 就是经典的单例模式(饿汉式)
- 2) 代码分析+Debug 源码+代码说明

```
public class Runtime {  
    private static Runtime currentRuntime = new Runtime();  
  
    /**  
     * Returns the runtime object associated with the current Java application.  
     * Most of the methods of class <code>Runtime</code> are instance  
     * methods and must be invoked with respect to the current runtime object.  
     *  
     * @return the <code>Runtime</code> object associated with the current  
     *         Java application.  
     */  
    public static Runtime getRuntime() {  
        return currentRuntime;  
    }  
  
    /** Don't let anyone else instantiate this class */  
    private Runtime() {}  
  
    /**
```



5.12 单例模式注意事项和细节说明

- 1) 单例模式保证了系统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，使用单例模式可以提高系统性能
- 2) 当想实例化一个单例类的时候，必须要记住使用相应的获取对象的方法，而不是使用 new
- 3) 单例模式使用的场景：需要频繁的进行创建和销毁的对象、创建对象时耗时过多或耗费资源过多(即：**重量级对象**)，但又经常用到的对象、**工具类对象**、频繁访问数据库或文件的对象(比如**数据源**、**session 工厂**等)

第 6 章 工厂模式

6.1 简单工厂模式

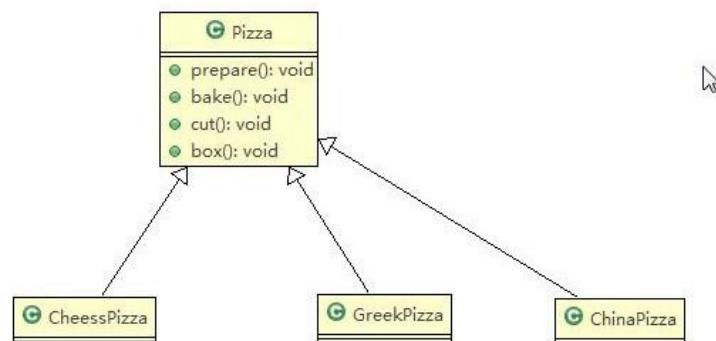
6.1.1 看一个具体的需求

看一个披萨的项目：要便于披萨种类的扩展，要便于维护

- 1) 披萨的种类很多(比如 GreekPizz、CheesePizz 等)
- 2) 披萨的制作有 prepare, bake, cut, box
- 3) 完成披萨店订购功能。

6.1.2 使用传统的方式来完成

- 1) 思路分析(类图)



编写 OrderPizza.java 去订购需要的各种 Pizza

- 2) 看老师代码的演示

```
public class OrderPizza {
```



```
// 构造器
// public OrderPizza() {
//     Pizza pizza = null;
//     String orderType; // 订购披萨的类型
//     do {
//         orderType = getType();
//         if (orderType.equals("greek")) {
//             pizza = new GreekPizza();
//             pizza.setName(" 希腊披萨 ");
//         } else if (orderType.equals("cheese")) {
//             pizza = new CheesePizza();
//             pizza.setName(" 奶酪披萨 ");
//         } else if (orderType.equals("pepper")) {
//             pizza = new PepperPizza();
//             pizza.setName(" 胡椒披萨 ");
//         } else {
//             break;
//         }
//         //输出 pizza 制作过程
//         pizza.prepare();
//         pizza.bake();
//         pizza.cut();
//         pizza.box();
//
//     } while (true);
// }
```

6.1.3 传统的方式的优缺点

- 1) 优点是比较容易理解，简单易操作。
- 2) 缺点是违反了设计模式的 **ocp** 原则，即对扩展开放，对修改关闭。即当我们给类增加新功能的时候，尽量不修改代码，或者尽可能少修改代码。
- 3) 比如我们这时要新增加一个 **Pizza** 的种类(**Pepper 披萨**)，我们需要做如下修改。
如果我们增加一个 **Pizza** 类，只要是订购 **Pizza** 的代码都需要修改。

```
//增加一段代码 OrderPizza.java //写
if (ordertype.equals("greek")) {
    pizza = new GreekPizza();
} else if (ordertype.equals("pepper")) {
    pizza = new PepperPizza();
} else if (ordertype.equals("cheese")) {
    pizza = new CheesePizza();
} else {
    break;
}}
```



4) 改进的思路分析

分析：修改代码可以接受，但是如果我们在其它的地方也有创建 **Pizza** 的代码，就意味着，也需要修改，而创建 **Pizza** 的代码，往往有多处。

思路：把创建 **Pizza** 对象封装到一个类中，这样我们有新的 **Pizza** 种类时，只需要修改该类就可，其它有创建到 **Pizza** 对象的代码就不需要修改了.-> 简单工厂模式

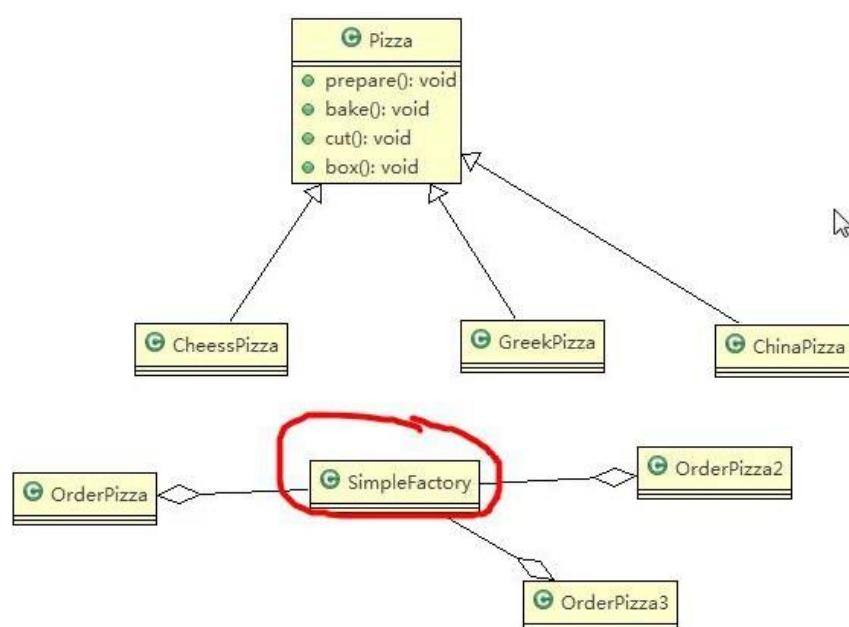
6.1.4 基本介绍

- 1) 简单工厂模式是属于创建型模式，是工厂模式的一种。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式

- 2) 简单工厂模式：定义了一个创建对象的类，由这个类来封装实例化对象的行为(代码)
- 3) 在软件开发中，当我们会用到大量的创建某种、某类或者某批对象时，就会使用到工厂模式。

6.1.5 使用简单工厂模式

- 1) 简单工厂模式的设计方案：定义一个可以实例化 Pizza 对象的类，封装创建对象的代码。



- 2) 看代码示例

```
package com.atguigu.factory.simplefactory.pizzastore.order;

import com.atguigu.factory.simplefactory.pizzastore.pizza.CheesePizza;
import com.atguigu.factory.simplefactory.pizzastore.pizza.GreekPizza;
import com.atguigu.factory.simplefactory.pizzastore.pizza.PepperPizza;
import com.atguigu.factory.simplefactory.pizzastore.pizza.Pizza;
```



```
//简单工厂类
public class SimpleFactory {

    //更加 orderType 返回对应的 Pizza 对象
    public Pizza createPizza(String orderType) {

        Pizza pizza = null;

        System.out.println("使用简单工厂模式");
        if (orderType.equals("greek")) {
            pizza = new GreekPizza();
            pizza.setName(" 希腊披萨 ");
        } else if (orderType.equals("cheese"))
            { pizza = new CheesePizza();
            pizza.setName(" 奶酪披萨 ");
        } else if (orderType.equals("pepper"))
            { pizza = new PepperPizza();
            pizza.setName("胡椒披萨");
        }

        return pizza;
    }

    //简单工厂模式也叫 静态工厂模式
}
```



```
public static Pizza createPizza2(String orderType) {
```

```
    Pizza pizza = null;
```

```
    System.out.println("使用简单工厂模式 2");
```

```
    if (orderType.equals("greek")) {
```

```
        pizza = new GreekPizza();
```

```
        pizza.setName(" 希腊披萨 ");
```

```
    } else if (orderType.equals("cheese"))
```

```
    { pizza = new CheesePizza();
```

```
        pizza.setName(" 奶酪披萨 ");
```

```
    } else if (orderType.equals("pepper"))
```

```
    { pizza = new PepperPizza();
```

```
        pizza.setName(" 胡椒披萨 ");
```

```
}
```

```
    return pizza;
```

```
}
```

```
}
```

```
//OrderPizza.java
```

```
package com.atguigu.factory.simplefactory.pizzastore.order;
```

```
import java.io.BufferedReader;
```



```
import java.io.IOException;
import java.io.InputStreamReader;

import com.atguigu.factory.simplefactory.pizzastore.pizza.Pizza;

public class OrderPizza {

    // 构造器
    // public OrderPizza() {
    //
    //     Pizza pizza = null;
    //     String orderType; // 订购披萨的类型
    //     do {
    //
    //         orderType = getType();
    //
    //         if (orderType.equals("greek")) {
    //
    //             pizza = new GreekPizza();
    //             pizza.setName(" 希腊披萨 ");
    //         } else if (orderType.equals("cheese")) {
    //
    //             pizza = new CheesePizza();
    //             pizza.setName(" 奶酪披萨 ");
    //         } else if (orderType.equals("pepper")) {
    //
    //             pizza = new PepperPizza();
    //             pizza.setName(" 胡椒披萨 ");
    //         } else {
    //
    //             break;
    //         }
    //     }
    // }
```



```
//          //输出 pizza 制作过程
//          pizza.prepare();
//          pizza.bake();
//          pizza.cut();
//          pizza.box();
//
//      } while (true);
//  }

//定义一个简单工厂对象
SimpleFactory simpleFactory;
Pizza pizza = null;

//构造器
public OrderPizza(SimpleFactory simpleFactory)
{
    setFactory(simpleFactory);
}

public void setFactory(SimpleFactory simpleFactory) {
    String orderType = ""; //用户输入的

    this.simpleFactory = simpleFactory; //设置简单工厂对象

    do {
        orderType = getType();
        pizza = this.simpleFactory.createPizza(orderType);
    }
}
```



```
//输出 pizza
if(pizza != null) { //订购成功
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
} else {
    System.out.println(" 订购披萨失败 ");
    break;
}
}while(true);
}

// 写一个方法，可以获取客户希望订购的披萨种类
private String getType()
{
    try {
        BufferedReader strin = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("input pizza 种类:");
        String str = strin.readLine();
        return str;
    } catch (IOException e)
    {
        e.printStackTrace();
        ; return "";
    }
}
```



```
}
```

6.2 工厂方法模式

6.2.1 看一个新的需求

披萨项目新的需求：客户在点披萨时，可以点不同口味的披萨，比如北京的奶酪 pizza、北京的胡椒 pizza 或者是伦敦的奶酪 pizza、伦敦的胡椒 pizza。

6.2.2 思路1

使用简单工厂模式，创建不同的简单工厂类，比如 BJPizzaSimpleFactory、LDPizzaSimpleFactory 等等。从当前这个案例来说，也是可以的，但是考虑到项目的规模，以及软件的可维护性、可扩展性并不是特别好。

6.2.3 思路2

使用工厂方法模式

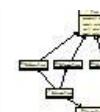
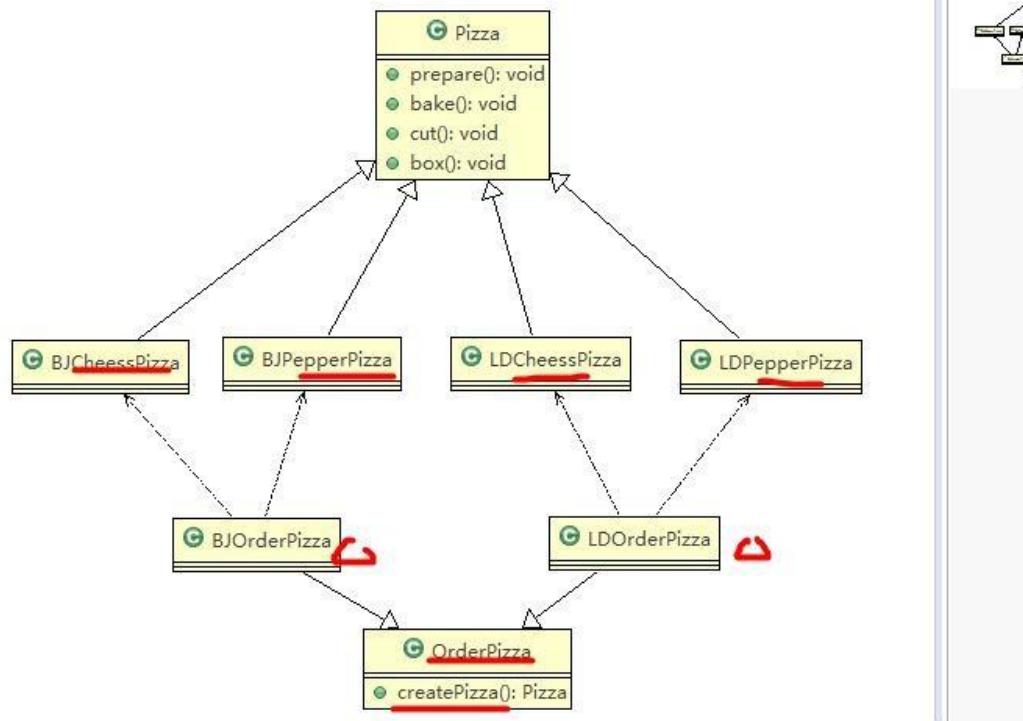
6.2.4 工厂方法模式介绍

- 1) 工厂方法模式设计方案：将披萨项目的实例化功能抽象成抽象方法，在不同的口味点餐子类中具体实现。
- 2) 工厂方法模式：定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方法模式将对象的实例化推迟到子类。

6.2.5 工厂方法模式应用案例

1) 披萨项目新的需求：客户在点披萨时，可以点不同口味的披萨，比如北京的奶酪 pizza、北京的胡椒 pizza 或者是伦敦的奶酪 pizza、伦敦的胡椒 pizza

2) 思路分析图解



3) 看老师代码实现

```
//OrderPizza.java 类
package com.atguigu.factory.factorymethod.pizzastore.order;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import com.atguigu.factory.factorymethod.pizzastore.pizza.Pizza;
```



```
public abstract class OrderPizza {  
  
    // 定义一个抽象方法，createPizza，让各个工厂子类自己实现  
    abstract Pizza createPizza(String orderType);  
  
    // 构造器  
    public OrderPizza()  
    {  
        Pizza pizza =  
            null;  
  
        String orderType; // 订购披萨的类型  
        do {  
            orderType = getType();  
  
            pizza = createPizza(orderType); // 抽象方法，由工厂子类完成  
  
            // 输出 pizza 制作过程  
            pizza.prepare();  
            pizza.bake();  
            pizza.cut();  
            pizza.box();  
  
        } while (true);  
    }  
}
```



```
// 写一个方法，可以获取客户希望订购的披萨种类
private String getType()

{ try {

    BufferedReader strin = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("input pizza 种类:");
    String str = strin.readLine();
    return str;

} catch (IOException e)

{ e.printStackTrace()
; return "";
}

}

}
```

```
package com.atguigu.factory.factorymethod.pizzastore.order;

import com.atguigu.factory.factorymethod.pizzastore.pizza.BJCheesePizza;
import com.atguigu.factory.factorymethod.pizzastore.pizza.BJPepperPizza;
import com.atguigu.factory.factorymethod.pizzastore.pizza.Pizza;

public class BJOrderPizza extends OrderPizza {
```



```
@Override
```

```
Pizza createPizza(String orderType) {  
  
    Pizza pizza = null;  
    if(orderType.equals("cheese")) {  
        pizza = new BJCheesePizza();  
    } else if (orderType.equals("pepper"))  
    { pizza = new BJPepperPizza();  
    }  
    // TODO Auto-generated method stub  
    return pizza;  
}
```

}

```
package com.atguigu.factory.factorymethod.pizzastore.order;
```

```
import com.atguigu.factory.factorymethod.pizzastore.pizza.BJCheesePizza;  
import com.atguigu.factory.factorymethod.pizzastore.pizza.BJPepperPizza;  
import com.atguigu.factory.factorymethod.pizzastore.pizza.LDCheesePizza;  
import com.atguigu.factory.factorymethod.pizzastore.pizza.LDPepperPizza;  
import com.atguigu.factory.factorymethod.pizzastore.pizza.Pizza;
```

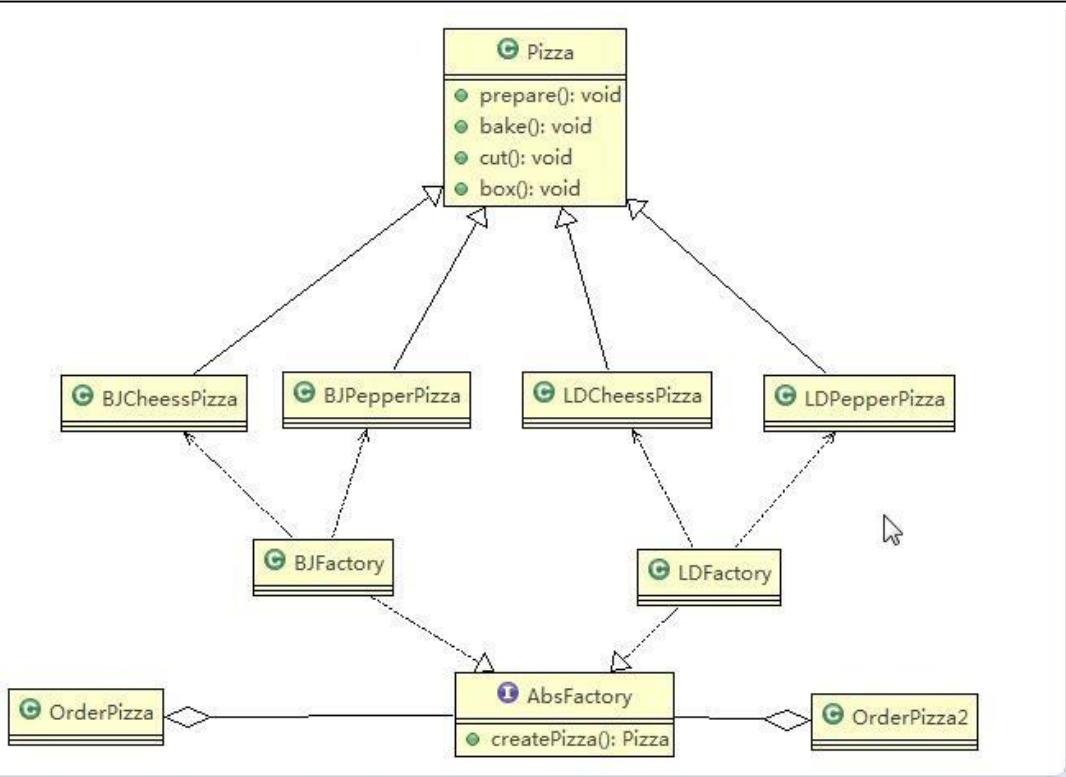
```
public class LDOOrderPizza extends OrderPizza {
```

```
@Override  
Pizza createPizza(String orderType) {  
  
    Pizza pizza = null;  
    if(orderType.equals("cheese")) {  
        pizza = new LDCheesePizza();  
    } else if (orderType.equals("pepper"))  
    { pizza = new LDPepperPizza();  
    }  
    // TODO Auto-generated method stub  
    return pizza;  
}  
  
}
```

6.3 抽象工厂模式

6.3.1 基本介绍

- 1) 抽象工厂模式：定义了一个 **interface** 用于创建相关或有依赖关系的对象簇，而无需指明具体的类。
- 2) 抽象工厂模式可以将简单工厂模式和工厂方法模式进行整合。
- 3) 从设计层面看，抽象工厂模式就是对简单工厂模式的改进(或者称为进一步的抽象)。
- 4) 将工厂抽象成两层，**AbsFactory**(抽象工厂) 和 具体实现的工厂子类。程序员可以根据创建对象类型使用对应的工厂子类。这样将单个的简单工厂类变成了工厂簇，更利于代码的维护和扩展。
- 5) 类图



6.3.2 抽象工厂模式应用实例

使用抽象工厂模式来完成披萨项目。

```
package com.atguigu.factory.absfactory.pizzastore.order;

import com.atguigu.factory.absfactory.pizzastore.pizza.Pizza;

//一个抽象工厂模式的抽象层(接口)
public interface AbsFactory {
    //让下面的工厂子类来具体实现
    public Pizza createPizza(String orderType);
}
```



```
package com.atguigu.factory.absfactory.pizzastore.order;

import com.atguigu.factory.absfactory.pizzastore.pizza.BJCheesePizza;
import com.atguigu.factory.absfactory.pizzastore.pizza.BJPepperPizza;
import com.atguigu.factory.absfactory.pizzastore.pizza.Pizza;

//这是工厂子类
public class BJFactory implements AbsFactory {

    @Override
    public Pizza createPizza(String orderType) {
        System.out.println(~使用的是抽象工厂模式~);
        // TODO Auto-generated method stub
        Pizza pizza = null;
        if(orderType.equals("cheese")) {
            pizza = new BJCheesePizza();
        } else if
            (orderType.equals("pepper")){ pizza =
            new BJPepperPizza();
        }
        return pizza;
    }

}
```

```
package com.atguigu.factory.absfactory.pizzastore.order;
```



```
import com.atguigu.factory.absfactory.pizzastore.pizza.LDCheesePizza;
import com.atguigu.factory.absfactory.pizzastore.pizza.LDPepperPizza;
import com.atguigu.factory.absfactory.pizzastore.pizza.Pizza;
```

```
public class LDFactory implements AbsFactory {
```

```
    @Override
```

```
    public Pizza createPizza(String orderType) {
        System.out.println(~使用的是抽象工厂模式~);
        Pizza pizza = null;
        if (orderType.equals("cheese"))
            { pizza = new
                LDCheesePizza();
        } else if (orderType.equals("pepper"))
            { pizza = new LDPepperPizza();
        }
        return pizza;
    }
```

```
}
```

```
//OrderPizza.java
package com.atguigu.factory.absfactory.pizzastore.order;

import java.io.BufferedReader;
```



```
import java.io.IOException;
import java.io.InputStreamReader;

import com.atguigu.factory.absfactory.pizzastore.pizza.Pizza;

public class OrderPizza {

    AbsFactory factory;

    // 构造器
    public OrderPizza(AbsFactory factory)
    {
        setFactory(factory);
    }

    private void setFactory(AbsFactory factory)
    {
        Pizza pizza = null;

        String orderType = ""; // 用户输入
        this.factory = factory;
        do {
            orderType = getType();
            // factory 可能是北京的工厂子类，也可能是伦敦的工厂子类
            pizza = factory.createPizza(orderType);
            if (pizza != null) { // 订购 ok
                pizza.prepare();
                pizza.bake();
                pizza.cut();
            }
        }
    }
}
```



```
    pizza.box();

} else {
    System.out.println("订购失败");
    break;
}

} while (true);

}

// 写一个方法，可以获取客户希望订购的披萨种类
private String getType()

{ try {

    BufferedReader strin = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("input pizza 种类:");
    String str = strin.readLine();
    return str;

} catch (IOException e)
{
    e.printStackTrace();
    ; return "";
}

}
}
```

6.4 工厂模式在 JDK-Calendar 应用的源码分析

- 1) JDK 中的 Calendar 类中，就使用了简单工厂模式



2) 源码分析+Debug 源码+说明

➤ 源码部分

```
package com.atguigu.jdk;
```

```
import java.util.Calendar;
```

```
public class Factory {
```

```
    public static void main(String[] args) {
```

```
        // TODO Auto-generated method stub
```

```
        // getInstance 是 Calendar 静态方法
```

```
        Calendar cal = Calendar.getInstance();
```

```
        // 注意月份下标从 0 开始，所以取月份要+1
```

```
        System.out.println("年:" + cal.get(Calendar.YEAR));
```

```
        System.out.println("月 :" + (cal.get(Calendar.MONTH) + 1));
```

```
        System.out.println("日:" + cal.get(Calendar.DAY_OF_MONTH));
```

```
        System.out.println("时:" + cal.get(Calendar.HOUR_OF_DAY));
```

```
        System.out.println("分:" + cal.get(Calendar.MINUTE));
```

```
        System.out.println("秒:" + cal.get(Calendar.SECOND));
```

```
}
```

```
}
```

```
//Calendar.java
```



```
public static Calendar getInstance()
{
    return createCalendar(TimeZone.getDefault(), Locale.getDefault(Locale.Category.FORMAT));
}

private static Calendar createCalendar(TimeZone zone,
                                      Locale aLocale) //根据 TimeZone zone, locale 创建对应的实例
{
    CalendarProvider provider =
        LocaleProviderAdapter.getAdapter(CalendarProvider.class, aLocale)
            .getCalendarProvider();
    if (provider != null)
    {
        try {
            return provider.getInstance(zone, aLocale);
        } catch (IllegalArgumentException iae) {
            // fall back to the default instantiation
        }
    }

    Calendar cal = null;

    if (aLocale.hasExtensions()) {
        String caltype = aLocale.getUnicodeLocaleType("ca");
        if (caltype != null) {
```



```
switch (caltype)
{
    case "buddhist":
        cal = new BuddhistCalendar(zone, aLocale);
        break;
    case "japanese":
        cal = new JapaneseImperialCalendar(zone, aLocale);
        break;
    case "gregory":
        cal = new GregorianCalendar(zone, aLocale);
        break;
}
}

if (cal == null) {
    // If no known calendar type is explicitly specified,
    // perform the traditional way to create a Calendar:
    // create a BuddhistCalendar for th_TH locale,
    // a JapaneseImperialCalendar for ja_JP_JP locale, or
    // a GregorianCalendar for any other locales.

    // NOTE: The language, country and variant strings are interned.

    if (aLocale.getLanguage() == "th" && aLocale.getCountry() == "TH")
        { cal = new BuddhistCalendar(zone, aLocale); }
    } else if (aLocale.getVariant() == "JP" && aLocale.getLanguage() == "ja"
        && aLocale.getCountry() == "JP") {
        cal = new JapaneseImperialCalendar(zone, aLocale);
    }
}
```



```
} else {
```



```
    cal = new GregorianCalendar(zone, aLocale);

}

}

return cal;

}
```

6.5 工厂模式小结

1) 工厂模式的意义

将实例化对象的代码提取出来，放到一个类中统一管理和维护，达到和主项目的依赖关系的解耦。从而提高项目的扩展和维护性。

2) 三种工厂模式(简单工厂模式、工厂方法模式、抽象工厂模式)

3) 设计模式的依赖抽象原则

- 创建对象实例时，不要直接 new 类，而是把这个 new 类的动作放在一个工厂的方法中，并返回。有的书上说，变量不要直接持有具体类的引用。
- 不要让类继承具体类，而是继承抽象类或者是实现 interface(接口)
- 不要覆盖基类中已经实现的方法。

第 7 章 原型模式

7.1 克隆羊问题

现在有一只羊 tom，姓名为: tom，年龄为: 1，颜色为: 白色，请编写程序创建和 tom 羊属性完全相同的 10 只羊。

7.2 传统方式解决克隆羊问题

1) 思路分析(图解)



2) 看老师代码的演示

```
package com.atguigu.prototype;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //传统的方法
        Sheep sheep = new Sheep("tom", 1, "白色");

        Sheep sheep2 = new Sheep(sheep.getName(), sheep.getAge(), sheep.getColor());
        Sheep sheep3 = new Sheep(sheep.getName(), sheep.getAge(), sheep.getColor());
        Sheep sheep4 = new Sheep(sheep.getName(), sheep.getAge(), sheep.getColor());
    }
}
```



```
Sheep sheep5 = new Sheep(sheep.getName(), sheep.getAge(), sheep.getColor());  
//....  
  
System.out.println(sheep);  
System.out.println(sheep2);  
System.out.println(sheep3);  
System.out.println(sheep4);  
System.out.println(sheep5);  
//...  
}  
}
```

7.3 传统的方式的优缺点

- 1) 优点是比较好理解，简单易操作。
- 2) 在创建新的对象时，总是需要重新获取原始对象的属性，如果创建的对象比较复杂时，效率较低
- 3) 总是需要重新初始化对象，而不是动态地获得对象运行时的状态，不够灵活
- 4) 改进的思路分析

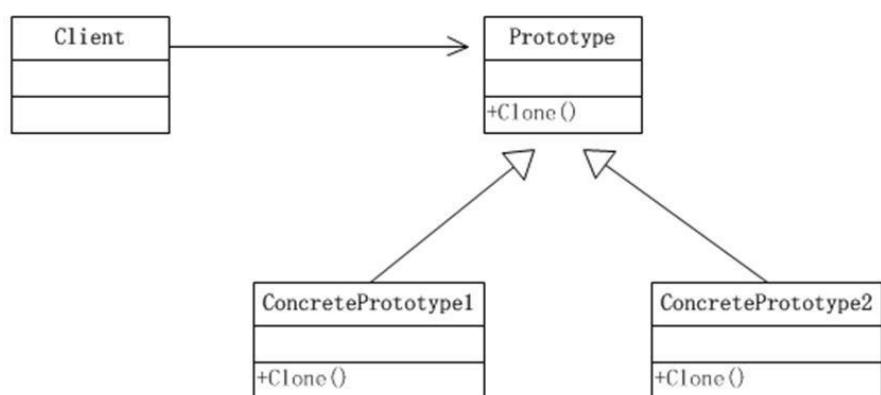
思路：Java 中 Object 类是所有类的根类，Object 类提供了一个 clone()方法，该方法可以将一个 Java 对象复制一份，但是需要实现 clone 的 Java 类必须要实现一个接口 Cloneable，该接口表示该类能够复制且具有复制的能力 => 原型模式

7.4 原型模式-基本介绍

基本介绍

- 1) 原型模式(Prototype 模式)是指：用原型实例指定创建对象的种类，并且通过拷贝这些原型，创建新的对象
- 2) 原型模式是一种创建型设计模式，允许一个对象再创建另外一个可定制的对象，无需知道如何创建的细节
- 3) 工作原理是：通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建，即 对象.clone()
- 4) 形象的理解：孙大圣拔出猴毛，变出其它孙大圣

7.5 原型模式原理结构图-uml 类图



➤ 原理结构图说明

- 1) Prototype：原型类，声明一个克隆自己的接口
- 2) ConcretePrototype：具体的原型类，实现一个克隆自己的操作
- 3) Client：让一个原型对象克隆自己，从而创建一个新的对象(属性一样)

7.6 原型模式解决克隆羊问题的应用实例

使用原型模式改进传统方式，让程序具有更高的效率和扩展性。

➤ 代码实现

```
package com.atguigu.prototype.improve;
```



```
public class Sheep implements Cloneable
{
    private String name;
    private int age;
    private String color;
    private String address = "蒙古羊";
    public Sheep friend; //是对象，克隆是会如何处理，默认是浅拷贝
    public Sheep(String name, int age, String color)
    {
        super();
        this.name = name;
        this.age = age;
        this.color = color;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public int getAge()
    {
        return age;
    }
    public void setAge(int age) {
```





```
// TODO Auto-generated method stub  
return sheep;  
}
```

```
}
```

```
package com.atguigu.prototype.improve;
```

```
public class Client {
```

```
    public static void main(String[] args) {  
        System.out.println("原型模式完成对象的创建");  
        // TODO Auto-generated method stub  
        Sheep sheep = new Sheep("tom", 1, "白色");  
  
        sheep.friend = new Sheep("jack", 2, "黑色");
```

```
        Sheep sheep2 = (Sheep)sheep.clone(); //克隆  
        Sheep sheep3 = (Sheep)sheep.clone(); //克隆  
        Sheep sheep4 = (Sheep)sheep.clone(); //克隆  
        Sheep sheep5 = (Sheep)sheep.clone(); //克隆
```

```
        System.out.println("sheep2 =" + sheep2 + "sheep2.friend=" + sheep2.friend.hashCode());
```

```

        System.out.println("sheep3 = " + sheep3 + "sheep3.friend=" + sheep3.friend.hashCode());
        System.out.println("sheep4 = " + sheep4 + "sheep4.friend=" + sheep4.friend.hashCode());
        System.out.println("sheep5 = " + sheep5 + "sheep5.friend=" + sheep5.friend.hashCode());
    }

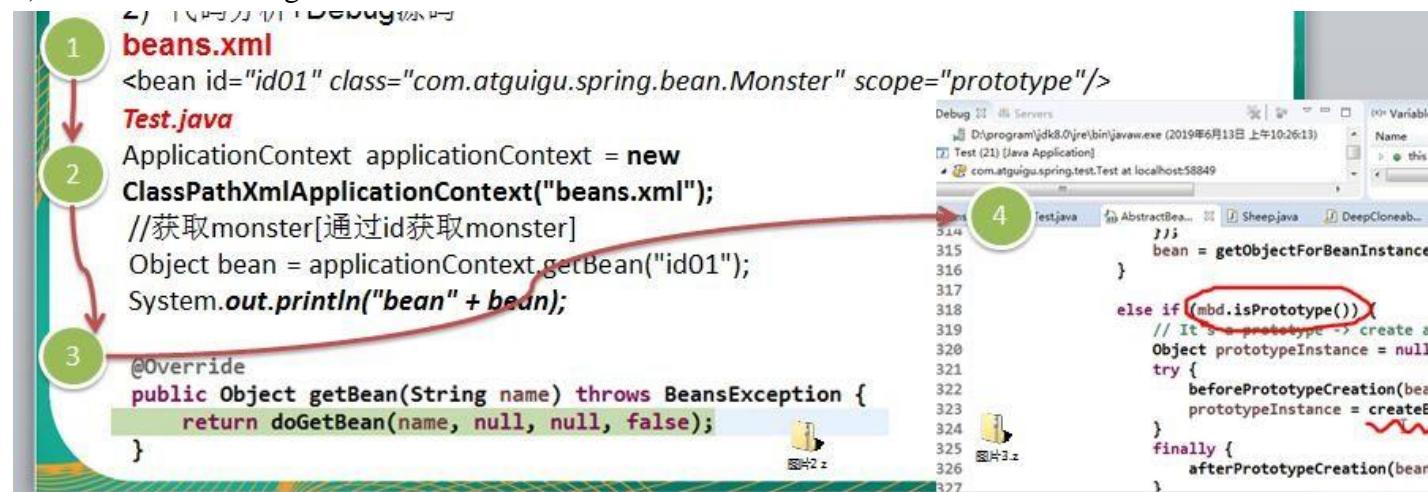
}

```

7.7 原型模式在 Spring 框架中源码分析

1) Spring 中原型 bean 的创建，就是原型模式的应用

2) 代码分析+Debug 源码



7.8 深入讨论-浅拷贝和深拷贝

7.8.1 浅拷贝的介绍

- 对于数据类型是基本数据类型的成员变量，浅拷贝会直接进行值传递，也就是将该属性值复制一份给新的对象。
- 对于数据类型是引用数据类型的成员变量，比如说成员变量是某个数组、某个类的对象等，那么浅拷贝会进行引用传递，也就是只是将该成员变量的引用值（内存地址）复制一份给新的对象。因为实际上两个对象的该成员变量都指向同一个实例。在这种情况下，在一个对象中修改该成员变量会影响到另一个对象的该成员变量值



- 3 前面我们克隆羊就是浅拷贝
- 4 浅拷贝是使用默认的 clone() 方法来实现
sheep = (Sheep) super.clone();

7.8.2 深拷贝基本介绍

- 1) 复制对象的所有基本数据类型的成员变量值
- 2) 为所有引用数据类型的成员变量申请存储空间，并复制每个引用数据类型成员变量所引用的对象，直到该对象可达的所有对象。也就是说，对象进行深拷贝要对整个对象(包括对象的引用类型)进行拷贝
- 3) 深拷贝实现方式 1：重写 **clone** 方法来实现深拷贝
- 4) 深拷贝实现方式 2：通过对对象序列化实现深拷贝(推荐)

7.9 深拷贝应用实例

- 1) 使用 重写 **clone** 方法实现深拷贝
- 2) 使用序列化来实现深拷贝
- 3) 代码演示

```
package com.atguigu.prototype.deepclone;

import java.io.Serializable;

public class DeepCloneableTarget implements Serializable, Cloneable {

    /**
     *
     */
}
```



```
*  
*/  
private static final long serialVersionUID = 1L;  
  
private String cloneName;  
  
private String cloneClass;  
  
//构造器  
public DeepCloneableTarget(String cloneName, String cloneClass)  
{ this.cloneName = cloneName;  
this.cloneClass = cloneClass;  
}  
  
//因为该类的属性，都是 String，因此我们这里使用默认的 clone 完成即可  
@Override  
protected Object clone() throws CloneNotSupportedException  
{ return super.clone();  
}  
}
```

```
//...  
package com.atguigu.prototype.deepclone;  
  
import java.io.ByteArrayInputStream;  
import java.io.ByteArrayOutputStream;
```



```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class DeepProtoType implements Serializable, Cloneable{

    public String name; //String 属性
    public DeepCloneableTarget deepCloneableTarget;// 引用类型
    public DeepProtoType()
    {
        super();
    }

    //深拷贝 - 方式 1 使用 clone 方法
    @Override
    protected Object clone() throws CloneNotSupportedException {

        Object deep = null;
        //这里完成对基本数据类型(属性)和 String 的克隆
        deep = super.clone();
        //对引用类型的属性，进行单独处理
        DeepProtoType deepProtoType = (DeepProtoType)deep;
        deepProtoType.deepCloneableTarget = (DeepCloneableTarget)deepCloneableTarget.clone();

        // TODO Auto-generated method stub
        return deepProtoType;
    }
}
```



```
}
```

```
//深拷贝 - 方式 2 通过对对象的序列化实现(推荐)

public Object deepClone() {

    //创建流对象
    ByteArrayOutputStream bos = null;
    ObjectOutputStream oos = null;
    ByteArrayInputStream bis = null;
    ObjectInputStream ois = null;

    try {

        //序列化
        bos = new ByteArrayOutputStream();
        oos = new ObjectOutputStream(bos);
        oos.writeObject(this); //当前这个对象以对象流的方式输出

        //反序列化
        bis = new ByteArrayInputStream(bos.toByteArray());
        ois = new ObjectInputStream(bis);
        DeepProtoType copyObj = (DeepProtoType)ois.readObject();

        return copyObj;
    }
}
```



```
        } catch (Exception e) {  
            // TODO: handle exception  
            e.printStackTrace();  
            return null;  
  
        } finally {  
            //关闭流  
            try {  
                bos.close();  
                oos.close();  
                bis.close();  
                ois.close();  
            } catch (Exception e2) {  
                // TODO: handle exception  
                System.out.println(e2.getMessage());  
            }  
        }  
  
    }  
  
}  
  
//Client.java  
package com.atguigu.prototype.deepclone;  
  
public class Client {
```



```
public static void main(String[] args) throws Exception {  
    // TODO Auto-generated method stub  
    DeepProtoType p = new DeepProtoType();  
    p.name = "宋江";  
    p.deepCloneableTarget = new DeepCloneableTarget("大牛", "小牛");  
  
    //方式 1 完成深拷贝  
  
    //    DeepProtoType p2 = (DeepProtoType) p.clone();  
    //  
    //    System.out.println("p.name=" + p.name + "p.deepCloneableTarget=" + p.deepCloneableTarget.hashCode());  
    //    System.out.println("p2.name=" + p.name + "p2.deepCloneableTarget=" + p2.deepCloneableTarget.hashCode());  
  
    //方式 2 完成深拷贝  
    DeepProtoType p2 = (DeepProtoType) p.deepClone();  
  
    System.out.println("p.name=" + p.name + "p.deepCloneableTarget=" + p.deepCloneableTarget.hashCode());  
    System.out.println("p2.name=" + p.name + "p2.deepCloneableTarget=" + p2.deepCloneableTarget.hashCode());  
  
}  
}
```

7.10 原型模式的注意事项和细节



- 1) 创建新的对象比较复杂时，可以利用原型模式简化对象的创建过程，同时也能够提高效率
- 2) 不用重新初始化对象，而是动态地获得对象运行时的状态
- 3) 如果原始对象发生变化(增加或者减少属性)，其它克隆对象的也会发生相应的变化，无需修改代码
- 4) 在实现深克隆的时候可能需要比较复杂的代码
- 5) 缺点：需要为每一个类配备一个克隆方法，这对全新的类来说不是很难，但对已有的类进行改造时，需要修改其源代码，违背了 ocp 原则，这点请同学们注意。

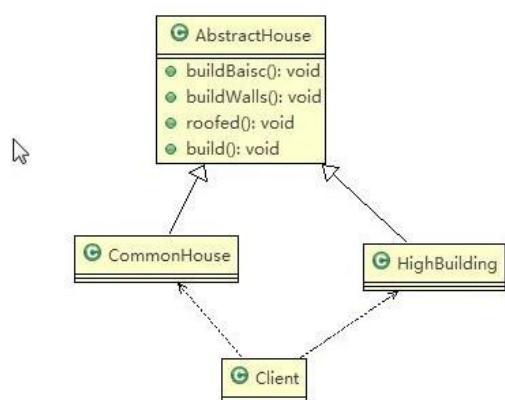
第 8 章 建造者模式

8.1 盖房项目需求

- 1) 需要建房子：这一过程为打桩、砌墙、封顶
- 2) 房子有各种各样的，比如普通房，高楼，别墅，各种房子的过程虽然一样，但是要求不要相同的。
- 3) 请编写程序，完成需求。

8.2 传统方式解决盖房需求

- 1) 思路分析(图解)



- 2) 看老师代码的演示

```
package com.atguigu.builder;

public abstract class AbstractHouse {

    //打地基
    public abstract void buildBasic();
```



```
//砌墙  
public abstract void buildWalls();  
//封顶  
public abstract void roofed();
```

```
public void build()  
{ buildBasic();  
    buildWalls();  
    roofed();  
}
```

```
}
```

```
package com.atguigu.builder;  
  
public class CommonHouse extends AbstractHouse {  
  
    @Override  
    public void buildBasic() {  
        // TODO Auto-generated method stub  
        System.out.println("普通房子打地基");  
    }  
  
    @Override  
    public void buildWalls() {  
        // TODO Auto-generated method stub  
    }  
}
```



```
System.out.println(" 普通房子砌墙 ");
}

@Override
public void roofed() {
    // TODO Auto-generated method stub
    System.out.println(" 普通房子封顶 ");
}

}
```

```
package com.atguigu.builder;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        CommonHouse commonHouse = new CommonHouse();
        commonHouse.build();
    }
}
```

8.3 传统方式的问题分析

- 1) 优点是比较好理解，简单易操作。



-
- 2) 设计的程序结构，过于简单，没有设计缓存层对象，程序的扩展和维护不好。也就是说，这种设计方案，把产品(即：房子) 和 创建产品的过程(即：建房子流程) 封装在一起，耦合性增强了。
 - 3) 解决方案：将产品和产品建造过程解耦 => 建造者模式。

8.4 建造者模式基本介绍

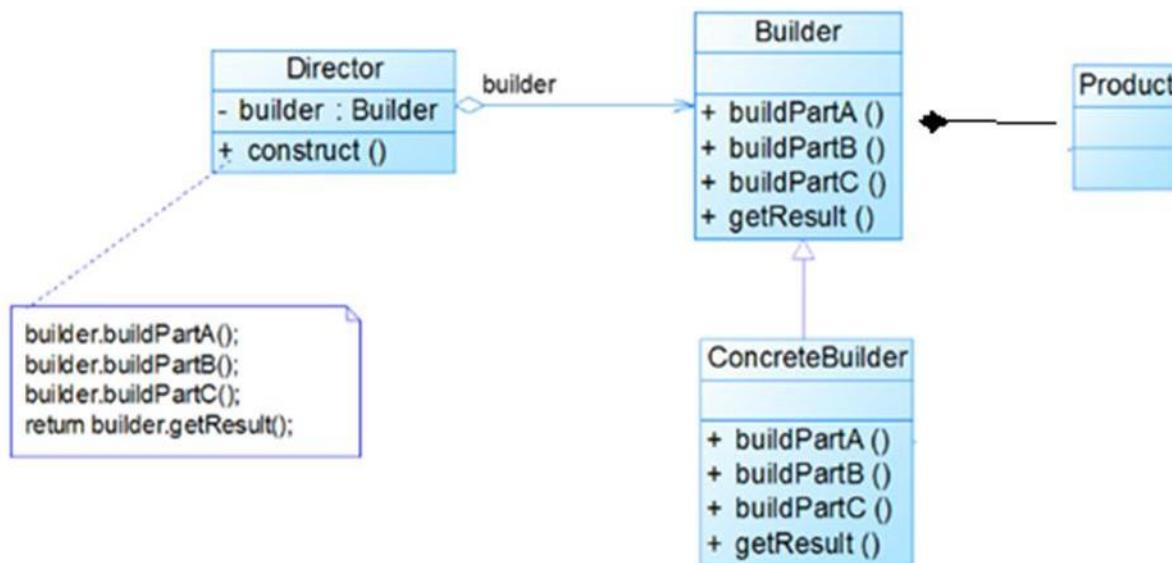
基本介绍

- 1) 建造者模式 (**Builder Pattern**) 又叫生成器模式，是一种对象构建模式。它可以将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方法可以构造出不同表现（属性）的对象。
- 2) 建造者模式 是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。

8.5 建造者模式的四个角色

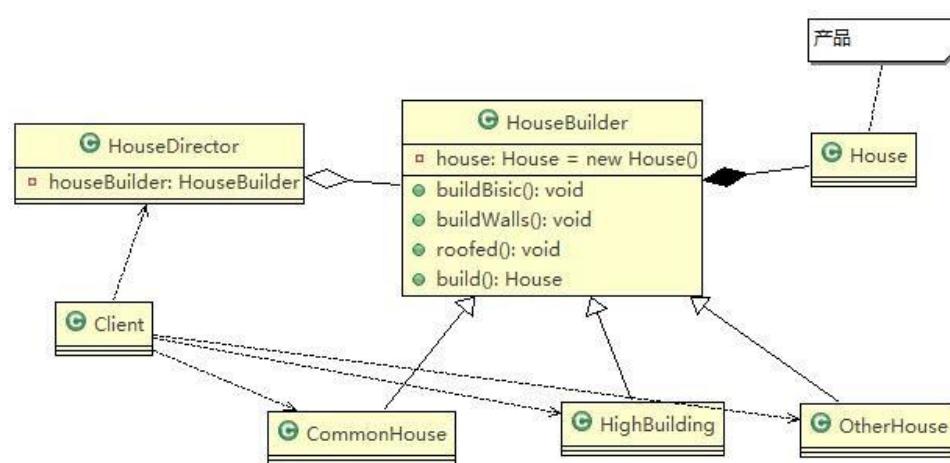
- 1) Product (产品角色)：一个具体的产品对象。
- 2) Builder (抽象建造者)：创建一个 Product 对象的各个部件指定的 接口/抽象类。
- 3) ConcreteBuilder (具体建造者)：实现接口，构建和装配各个部件。
- 4) Director (指挥者)：构建一个使用 Builder 接口的对象。它主要是用于创建一个复杂的对象。它主要有两个作用，一是：隔离了客户与对象的生产过程，二是：负责控制产品对象的生产过程。

8.6 建造者模式原理类图



8.7 建造者模式解决盖房需求应用实例

- 1) 需要建房子：这一过程为打桩、砌墙、封顶。不管是普通房子也好，别墅也好都需要经历这些过程，下面我们使用建造者模式(Builder Pattern)来完成
- 2) 思路分析图解(类图)



3) 代码实现

improve.zip

```
package com.atguigu.builder.improve;

public class Client {

    public static void main(String[] args) {

        //盖普通房子
        CommonHouse commonHouse = new CommonHouse();
        //准备创建房子的指挥者
        HouseDirector houseDirector = new HouseDirector(commonHouse);

        //完成盖房子，返回产品(普通房子)
        House house = houseDirector.constructHouse();

        //System.out.println("输出流程");

        System.out.println("-----");
        //盖高楼
        HighBuilding highBuilding = new HighBuilding();
        //重置建造者
        houseDirector.setHouseBuilder(highBuilding);
        //完成盖房子，返回产品(高楼)
        houseDirector.constructHouse();
    }
}
```



```
}
```

```
package com.atguigu.builder.improve;
```

```
public class CommonHouse extends HouseBuilder {
```

```
    @Override
```

```
    public void buildBasic() {
```

```
        // TODO Auto-generated method stub
```

```
        System.out.println(" 普通房子打地基 5 米 ");
```

```
}
```

```
    @Override
```

```
    public void buildWalls() {
```

```
        // TODO Auto-generated method stub
```

```
        System.out.println(" 普通房子砌墙 10cm ");
```

```
}
```

```
    @Override
```

```
    public void roofed() {
```

```
        // TODO Auto-generated method stub
```

```
        System.out.println(" 普通房子屋顶 ");
```

```
}
```



```
}
```

```
package com.atguigu.builder.improve;

public class HighBuilding extends HouseBuilder {

    @Override
    public void buildBasic() {
        // TODO Auto-generated method stub
        System.out.println(" 高楼的打地基 100 米 ");
    }

    @Override
    public void buildWalls() {
        // TODO Auto-generated method stub
        System.out.println(" 高楼的砌墙 20cm ");
    }

    @Override
    public void roofed() {
        // TODO Auto-generated method stub
        System.out.println(" 高楼的透明屋顶 ");
    }

}
```



```
package com.atguigu.builder.improve;

// 产 品 ->Product
public class House {

    private String baise;
    private String wall;
    private String roofed;

    public String getBaise() {
        return baise;
    }

    public void setBaise(String baise) {
        this.baise = baise;
    }

    public String getWall() {
        return wall;
    }

    public void setWall(String wall) {
        this.wall = wall;
    }

    public String getRoofed() {
        return roofed;
    }

    public void setRoofed(String roofed) {
        this.roofed = roofed;
    }
}
```



```
}
```

```
package com.atguigu.builder.improve;

// 抽象的建造者
public abstract class HouseBuilder {

    protected House house = new House();

    //将建造的流程写好, 抽象的方法
    public abstract void buildBasic();
    public abstract void buildWalls();
    public abstract void roofed();

    //建造房子好, 将产品(房子) 返回
    public House buildHouse()
    {
        return house;
    }
}
```

```
package com.atguigu.builder.improve;

//指挥者, 这里去指定制作流程, 返回产品
```

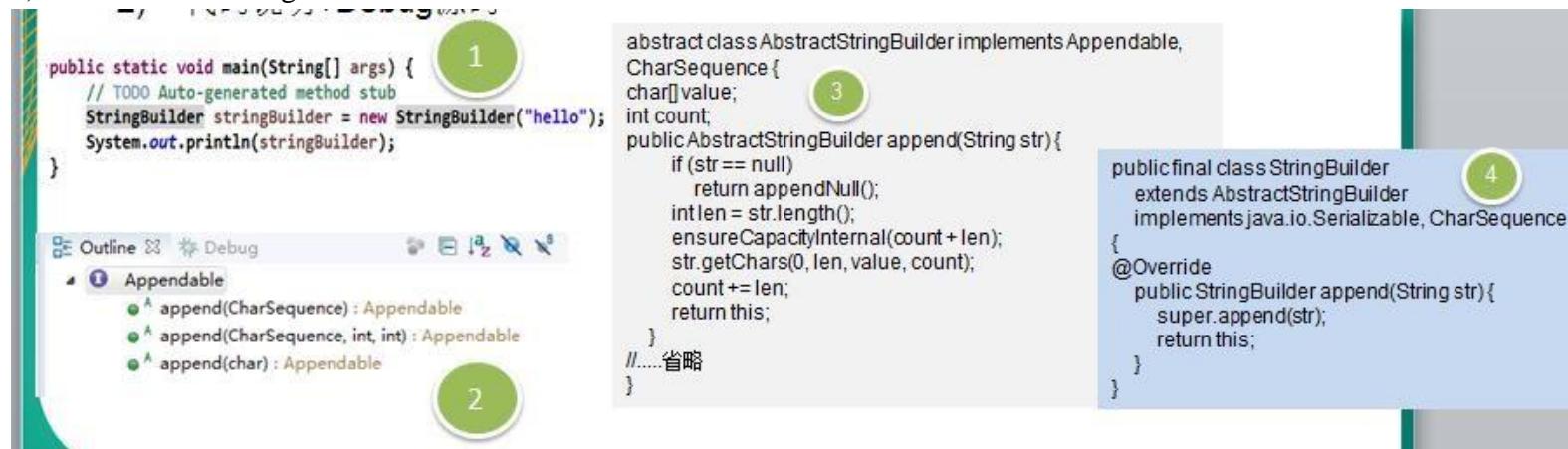


```
public class HouseDirector {  
  
    HouseBuilder houseBuilder = null;  
  
    //构造器传入 houseBuilder  
    public HouseDirector(HouseBuilder houseBuilder)  
    { this.houseBuilder = houseBuilder;  
    }  
  
    //通过 setter 传入 houseBuilder  
    public void setHouseBuilder(HouseBuilder houseBuilder)  
    { this.houseBuilder = houseBuilder;  
    }  
  
    //如何处理建造房子的流程，交给指挥者  
    public House constructHouse() {  
        houseBuilder.buildBasic();  
        houseBuilder.buildWalls();  
        houseBuilder.roofed();  
        return houseBuilder.buildHouse();  
    }  
}
```

8.8 建造者模式在 JDK 的应用和源码分析

1) java.lang.StringBuilder 中的建造者模式

2) 代码说明+Debug 源码



3) 源码中建造者模式角色分析

- ✓ `Appendable` 接口定义了多个 `append` 方法(抽象方法), 即 `Appendable` 为抽象建造者, 定义了抽象方法
- ✓ `AbstractStringBuilder` 实现了 `Appendable` 接口方法, 这里的 `AbstractStringBuilder` 已经是建造者, 只是不能实例化
- ✓ `StringBuilder` 即充当了指挥者角色, 同时充当了具体的建造者, 建造方法的实现是由 `AbstractStringBuilder` 完成, 而 `StringBuilder` 继承了 `AbstractStringBuilder`

8.9 建造者模式的注意事项和细节

- 1) 客户端(使用程序)不必知道产品内部组成的细节, 将产品本身与产品的创建过程解耦, 使得相同的创建过程可以创建不同的产品对象
- 2) 每一个具体建造者都相对独立, 而与其他的具体建造者无关, 因此可以很方便地替换具体建造者或增加新的具体建造者, 用户使用不同的具体建造者即可得到不同的产品对象



-
- 3) 可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程
 - 4) 增加新的具体建造者无须修改原有类库的代码，指挥者类针对抽象建造者类编程，系统扩展方便，符合“开闭原则”
 - 5) 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。
 - 6) 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大，因此在这种情况下，要考虑是否选择建造者模式.
- 7) 抽象工厂模式 VS 建造者模式

抽象工厂模式实现对产品家族的创建，一个产品家族是这样的一系列产品：具有不同分类维度的产品组合，采用抽象工厂模式不需要关心构建过程，只关心什么产品由什么工厂生产即可。而建造者模式则是要求按照指定的蓝图建造产品，它的主要目的是通过组装零配件而产生一个新产品

第 9 章 适配器设计模式

9.1 现实生活中的适配器例子

泰国插座用的是两孔的（欧标），可以买个多功能转换插头(适配器)，这样就可以使用了。

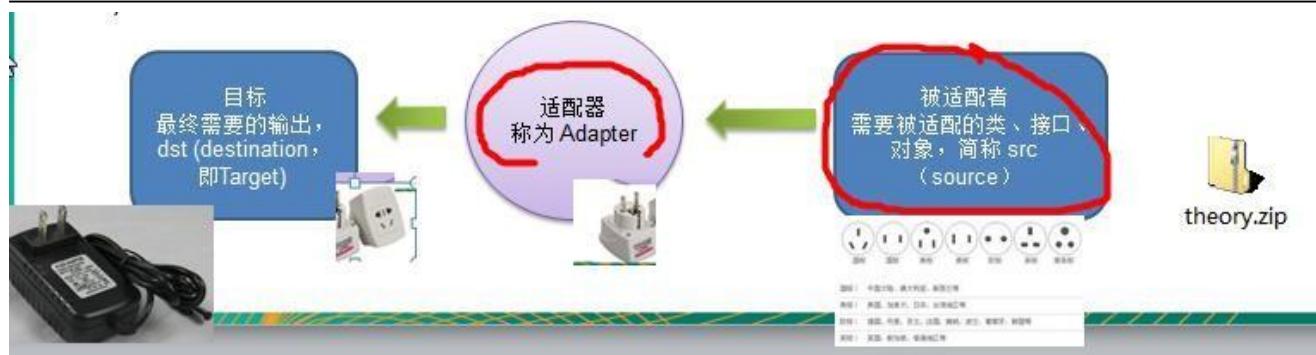


9.2 基本介绍

- 1) 适配器模式(Adapter Pattern)将某个类的接口转换成客户端期望的另一个接口表示，**主的目的是兼容性**，让原本因接口不匹配不能一起工作的两个类可以协同工作。其别名为包装器(Wrapper)
- 2) 适配器模式属于结构型模式
- 3) 主要分为三类：**类适配器模式、对象适配器模式、接口适配器模式**

9.3 工作原理

- 1) 适配器模式：将一个类的接口转换成另一种接口.让原本接口不兼容的类可以兼容
- 2) 从用户的角度看不到被适配者，是解耦的
- 3) 用户调用适配器转化出来的目标接口方法，适配器再调用被适配者的相关接口方法
- 4) 用户收到反馈结果，感觉只是和目标接口交互，如图



9.4 类适配器模式

9.4.1 类适配器模式介绍

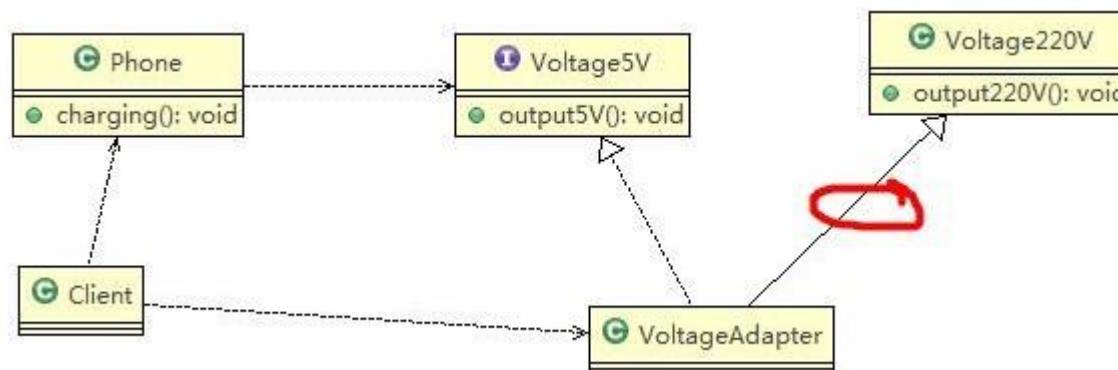
基本介绍：Adapter 类，通过继承 src 类，实现 dst 类接口，完成 src->dst 的适配。

9.4.2 类适配器模式应用实例

1) 应用实例说明

以生活中充电器的例子来讲解适配器，充电器本身相当于 Adapter，220V 交流电相当于 src (即被适配者)，我们的目 dst(即 目标)是 5V 直流电

2) 思路分析(类图)



3) 代码实现



classadapter.zip

```
package com.atguigu.adapter.classadapter;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println(" === 类适配器模式 =====");
        Phone phone = new Phone();
        phone.charging(new VoltageAdapter());
    }
}
```

```
package com.atguigu.adapter.classadapter;
```

```
//适配接口
public interface IVoltage5V
{
    public int output5V();
}
```

```
package com.atguigu.adapter.classadapter;

public class Phone {
```



```
//充电
public void charging(IVoltage5V iVoltage5V)
{
    if(iVoltage5V.output5V() == 5) {
        System.out.println("电压为 5V, 可以充电~~");
    } else if (iVoltage5V.output5V() > 5) {
        System.out.println("电压大于 5V, 不能充电~~");
    }
}
```

```
package com.atguigu.adapter.classadapter;

//被适配的类
public class Voltage220V {
    //输出 220V 的电压
    public int output220V()
    {
        int src = 220;
        System.out.println("电压=" + src + "伏");
        return src;
    }
}
```

```
package com.atguigu.adapter.classadapter;

//适配器类
```



```
public class VoltageAdapter extends Voltage220V implements IVoltage5V {  
  
    @Override  
  
    public int output5V() {  
  
        // TODO Auto-generated method stub  
        //获取到 220V 电压  
        int srcV = output220V();  
        int dstV = srcV / 44 ;//转成 5v  
        return dstV;  
  
    }  
  
}
```

9.4.3 类适配器模式注意事项和细节

- 1) Java 是单继承机制，所以类适配器需要继承 src 类这一点算是一个缺点，因为这要求 dst 必须是接口，有一定局限性；
- 2) src 类的方法在 Adapter 中都会暴露出来，也增加了使用的成本。
- 3) 由于其继承了 src 类，所以它可以根据需求重写 src 类的方法，使得 Adapter 的灵活性增强了。

9.5 对象适配器模式

9.5.1 对象适配器模式介绍

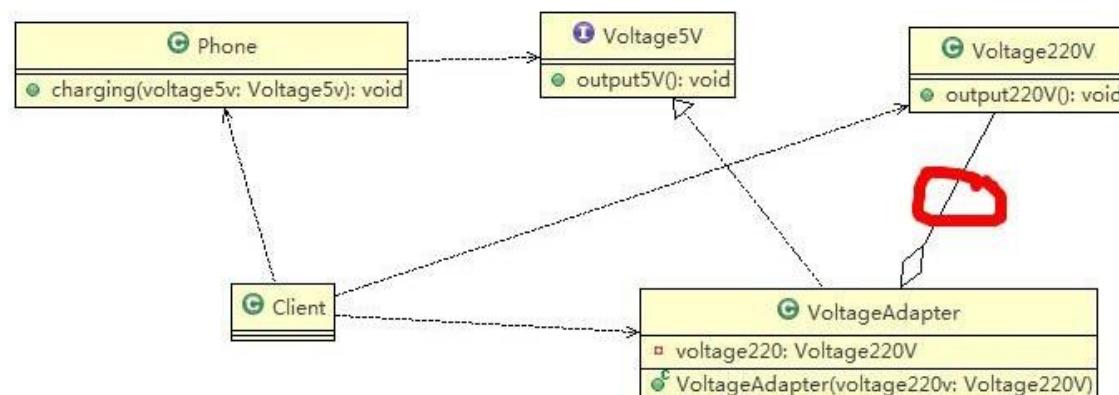
- 1) 基本思路和类的适配器模式相同，只是将 Adapter 类作修改，不是继承 src 类，而是持有 src 类的实例，以解决兼容性的问题。即：持有 src 类，实现 dst 类接口，完成 src->dst 的适配
- 2) 根据“合成复用原则”，在系统中尽量使用关联关系（聚合）来替代继承关系。
- 3) 对象适配器模式是适配器模式常用的一种

9.5.2 对象适配器模式应用实例

1) 应用实例说明

以生活中充电器的例子来讲解适配器，充电器本身相当于 Adapter，220V 交流电相当于 src (即被适配者)，我们的目 dst(即目标)是 5V 直流电，使用对象适配器模式完成。

2) 思路分析(类图)：只需修改适配器即可，如下：



3) 代码实现

```

objectadapter.zip
package com.atguigu.adapter.objectadapter;

```



```
public class Client {  
  
    public static void main(String[] args) {  
  
        // TODO Auto-generated method stub  
        System.out.println(" === 对象适配器模式 =====");  
        Phone phone = new Phone();  
  
        phone.charging(new VoltageAdapter(new Voltage220V()));  
  
    }  
  
}
```

```
package com.atguigu.adapter.objectadapter;  
  
//适配接口  
public interface IVoltage5V  
{  
    public int output5V();  
}
```

```
package com.atguigu.adapter.objectadapter;  
  
public class Phone {  
  
    //充电  
    public void charging(IVoltage5V iVoltage5V) {  
        if(iVoltage5V.output5V() == 5) {  
    }
```



```
System.out.println("电压为 5V, 可以充电~~");
} else if (iVoltage5V.output5V() > 5) {
    System.out.println("电压大于 5V, 不能充电~~");
}
}
```

```
package com.atguigu.adapter.objectadapter;
```

```
//被适配的类
public class Voltage220V {
    //输出 220V 的电压, 不变
    public int output220V()
        { int src = 220;
        System.out.println("电压=" + src + "伏");
        return src;
    }
}
```

```
package com.atguigu.adapter.objectadapter;
```

```
//适配器类
public class VoltageAdapter implements IVoltage5V {
    private Voltage220V voltage220V; // 关联关系-聚合
```



```
//通过构造器，传入一个 Voltage220V 实例
public VoltageAdapter(Voltage220V voltage220v) {

    this.voltage220V = voltage220v;
}

@Override
public int output5V() {

    int dst = 0;
    if(null != voltage220V) {

        int src = voltage220V.output220V();//获取 220V 电压
        System.out.println("使用对象适配器，进行适配~~");
        dst = src / 44;
        System.out.println("适配完成，输出的电压为=" + dst);
    }

    return dst;
}

}
```



9.5.3 对象适配器模式注意事项和细节

- 1) 对象适配器和类适配器其实算是同一种思想，只不过实现方式不同。

根据合成复用原则，使用组合替代继承，所以它解决了类适配器必须继承 src 的局限性问题，也不再要求 dst 必须是接口。

- 2) 使用成本更低，更灵活。

9.6 接口适配器模式

9.6.1 接口适配器模式介绍

- 1) 一些书籍称为：适配器模式(Default Adapter Pattern)或缺省适配器模式。
- 2) 核心思路：当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可有选择地覆盖父类的某些方法来实现需求
- 3) 适用于一个接口不想使用其所有的方法的情况。

9.6.2 接口适配器模式应用实例

- 1) Android 中的属性动画 ValueAnimator 类可以通过 addListener(AnimatorListener listener)方法添加监听器，那么常规写法如右：
- 2) 有时候我们不想实现 Animator.AnimatorListener 接口的全部方法，我们只想监听 onAnimationStart，我们会如下写

下写

```
ValueAnimator valueAnimator = ValueAnimator.ofInt(0, 100);
    valueAnimator.addListener(new AnimatorListenerAdapter() {
        @Override
        public void onAnimationStart(Animator animation) {
            //xxxx具体实现
        }
    });
    valueAnimator.start();
```

```
ValueAnimator valueAnimator = ValueAnimator.ofInt(0, 100);
    valueAnimator.addListener(new Animator.AnimatorListener() {
        @Override
        public void onAnimationStart(Animator animation) {
        }

        @Override
        public void onAnimationEnd(Animator animation) {
        }

        @Override
        public void onAnimationCancel(Animator animation) {
        }

        @Override
        public void onAnimationRepeat(Animator animation) {
        }
    });
    valueAnimator.start();
```

- 3) AnimatorListenerAdapter 类，就是一个接口适配器，代码如右图:它空实现了 Animator.AnimatorListener 类(src)的所有方法。
- 4) AnimatorListener 是一个接口。

```
public abstract class AnimatorListenerAdapter implements Animator.AnimatorListener {
    @Override //默认实现
    public void onAnimationCancel(Animator animation) {
    }

    @Override
    public void onAnimationEnd(Animator animation) {
    }

    @Override
    public void onAnimationRepeat(Animator animation) {
    }

    @Override
    public void onAnimationStart(Animator animation) {
    }

    @Override
    public void onAnimationPause(Animator animation) {
    }

    @Override
    public void onAnimationResume(Animator animation) {
    }
}
```

```
public static interface AnimatorListener {
    void onAnimationStart(Animator animation);

    void onAnimationEnd(Animator animation);

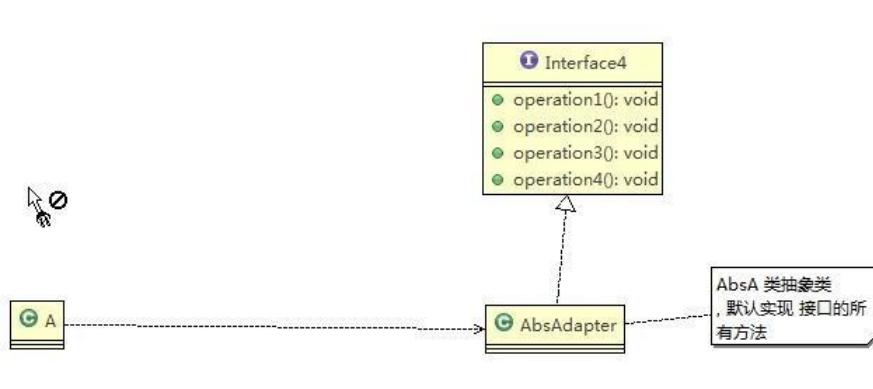
    void onAnimationCancel(Animator animation);

    void onAnimationRepeat(Animator animation);
}
```

5) 程序里的匿名内部类就是 Listener 具体实现类

```
new AnimatorListenerAdapter() {  
    @Override  
    public void onAnimationStart(Animator animation) {  
        //xxxx具体实现  
    }  
}
```

6) 案例说明



```
package com.atguigu.adapter.interfaceadapter;  
  
public interface Interface4 {  
    public void m1();  
    public void m2();  
    public void m3();  
    public void m4();  
}
```

```
package com.atguigu.adapter.interfaceadapter;
```



```
//在 AbsAdapter 我们将 Interface4 的方法进行默认实现
```

```
public abstract class AbsAdapter implements Interface4 {
```

```
    //默认实现
```

```
    public void m1() {
```

```
}
```

```
    public void m2() {
```

```
}
```

```
    public void m3() {
```

```
}
```

```
    public void m4() {
```

```
}
```

```
}
```

```
package com.atguigu.adapter.interfaceadapter;
```

```
public class Client {
```

```
    public static void main(String[] args) {
```

```
AbsAdapter absAdapter = new AbsAdapter() {  
    //只需要去覆盖我们需要使用接口方法  
    @Override  
  
    public void m1() {  
  
        // TODO Auto-generated method stub  
        System.out.println("使用了 m1 的方法");  
    }  
  
    absAdapter.m1();  
}  
}
```

9.7 适配器模式在 SpringMVC 框架应用的源码剖析

1) SpringMvc 中的 **HandlerAdapter**, 就使用了适配器模式

2) SpringMVC 处理请求的流程回顾

3) 使用 HandlerAdapter 的原因分析:

可以看到处理器的类型不同, 有多重实现方式, 那么调用方式就不是确定的, 如果需要直接调用 Controller 方法, 需要调用的时候就得不断是使用 if else 来进行判断是哪一种子类然后执行。那么如果后面要扩展 Controller, 就得修改原来的代码, 这样违背了 OCP 原则。

4) 代码分析+Debug 源码

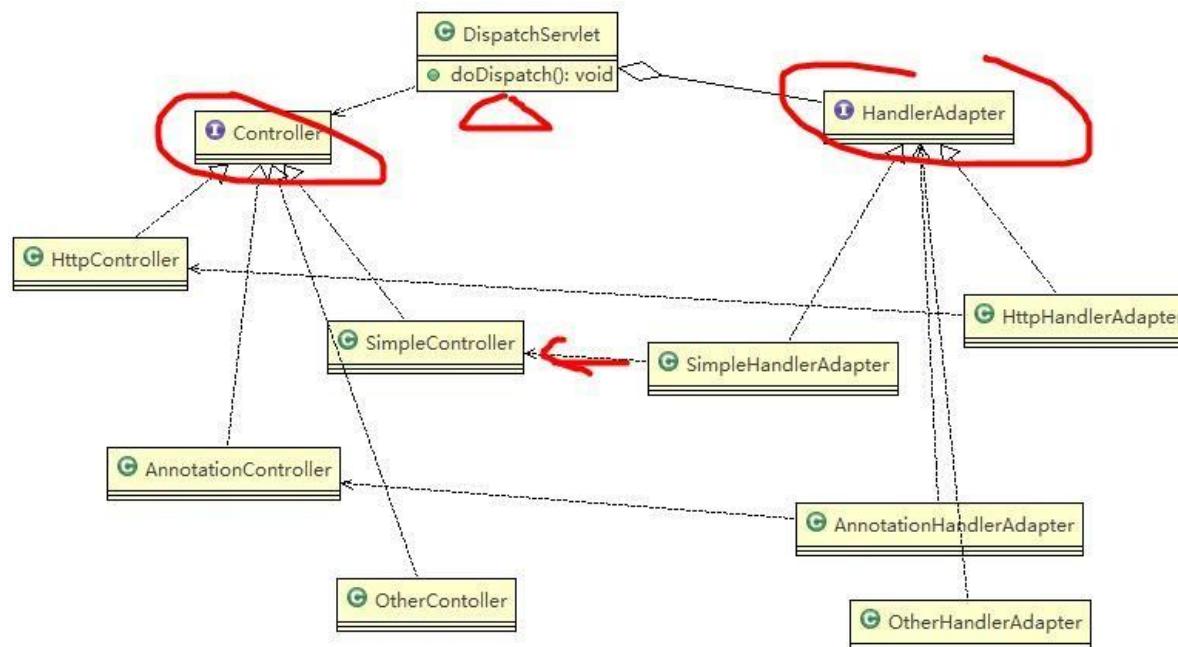
The screenshot shows a Java code editor and an outline view. The code editor displays `DispatcherServlet` with annotations 1 through 4. Annotation 1 points to the `getHandlerAdapter` method. Annotation 2 points to the `HandlerAdapter` interface in the outline view. Annotation 3 points to the `HandlerAdapter` class and its subclasses: `AbstractHandlerMethodAdapter`, `RequestMappingHandlerAdapter`, `AnnotationMethodHandlerAdapter`, `HttpRequestHandlerAdapter`, `SimpleControllerHandlerAdapter`, and `SimpleServletHandlerAdapter`. Annotation 4 points to a note about returning the appropriate `HandlerAdapter`.

```
public class DispatcherServlet extends FrameworkServlet {  
    // 通过HandlerMapping来映射Controller  
    mappedHandler = getHandler(processedRequest);  
    // 获取适配器  
    HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());  
    ...  
    // 通过适配器调用controller的方法并返回ModelAndView  
    mv = ha.handle(processedRequest, response, mappedHandler.getHandler());  
}  
  
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {  
    for (HandlerAdapter ha : this.handlerAdapters) {  
        if (logger.isTraceEnabled()) {  
            logger.trace("Testing handler adapter [" + ha + "]");  
        }  
        if (ha.supports(handler)) {  
            return ha;  
        }  
    }  
    throw new ServletException("No adapter for handler [" + handler +  
        "]: The DispatcherServlet configuration needs to include a HandlerAdapter th  
    }  
}
```

- 5) 动手写 SpringMVC 通过适配器设计模式获取到对应的 Controller 的源码

说明:

- Spring 定义了一个适配接口，使得每一种 Controller 有一种对应的适配器实现类
- 适配器代替 controller 执行相应的方法
- 扩展 Controller 时，只需要增加一个适配器类就完成了 SpringMVC 的扩展了，
这就是设计模式的力量



 springmvc.zip

源码:

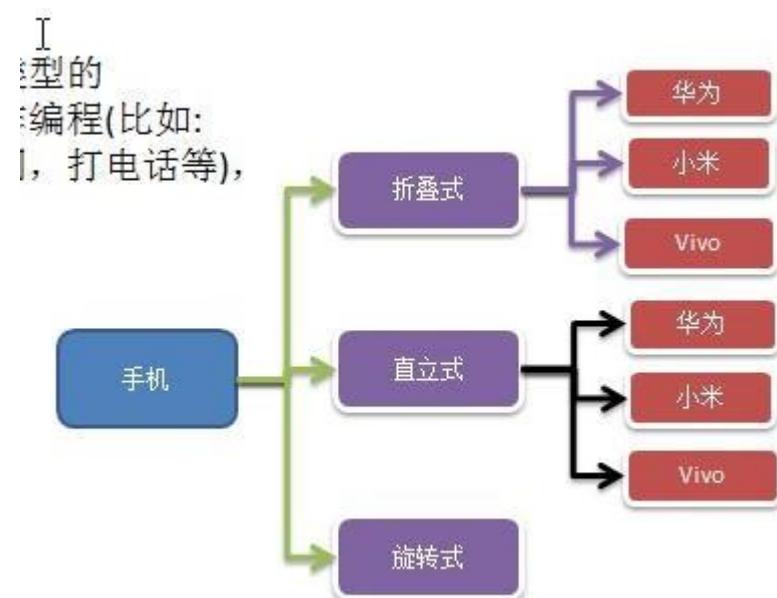
9.8 适配器模式的注意事项和细节

- 1) 三种命名方式，是根据 src 是以怎样的形式给到 Adapter (在 Adapter 里的形式) 来命名的。
- 2) 类适配器：以类给到，在 Adapter 里，就是将 src 当做类，继承
 对象适配器：以对象给到，在 Adapter 里，将 src 作为一个对象，持有
 接口适配器：以接口给到，在 Adapter 里，将 src 作为一个接口，实现
- 3) Adapter 模式最大的作用还是将原本不兼容的接口融合在一起工作。
- 4) 实际开发中，实现起来不拘泥于我们讲解的三种经典形式

第 10 章 桥接模式

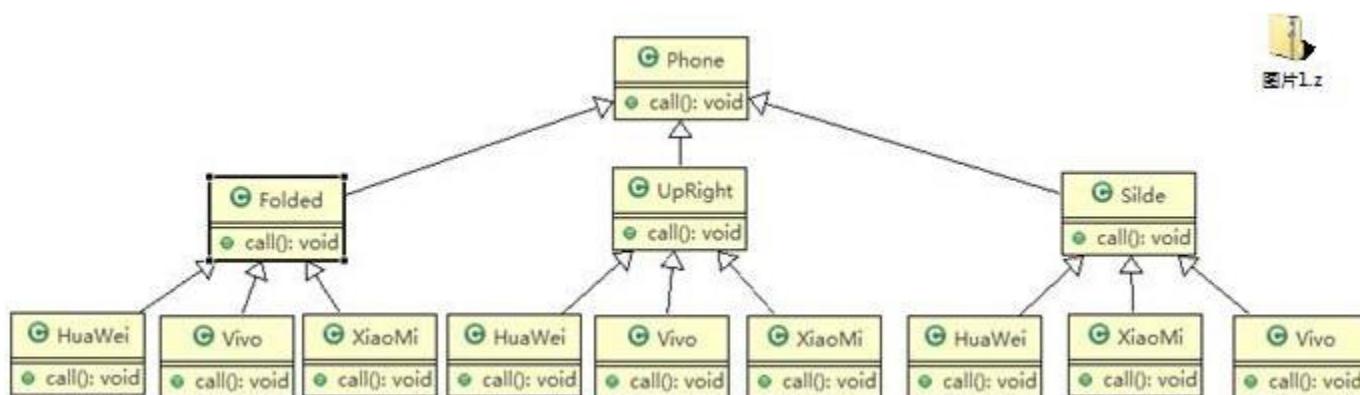
10.1 手机操作问题

现在对不同手机类型的不同品牌实现操作编程(比如:开机、关机、上网, 打电话等), 如图:



10.2 传统方案解决手机操作问题

传统方法对应的类图



10.3 传统方案解决手机操作问题分析

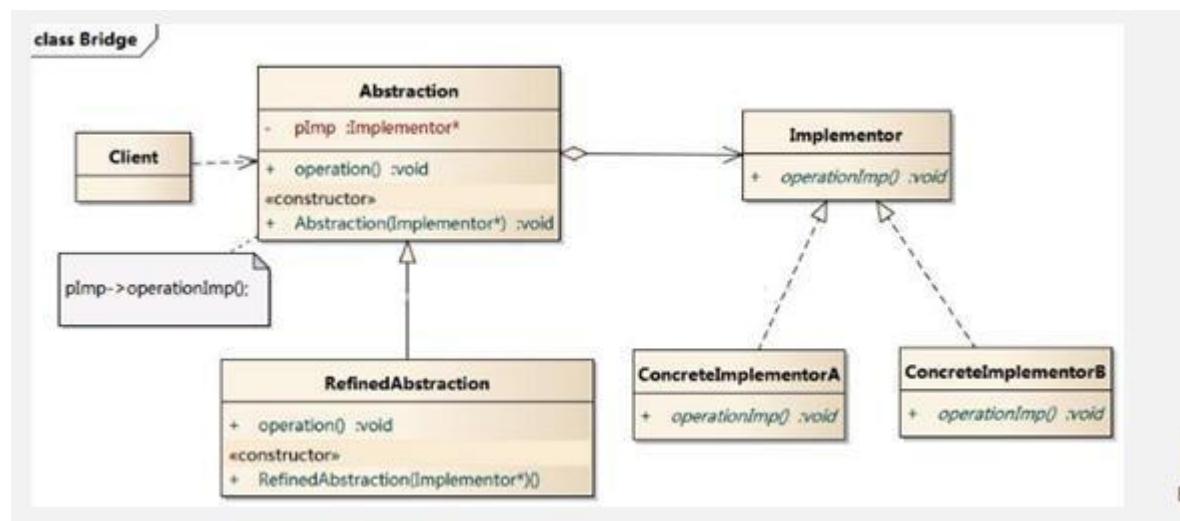
- 1) 扩展性问题(类爆炸), 如果我们再增加手机的样式(旋转式), 就需要增加各个品牌手机的类, 同样如果我们增加一个手机品牌, 也要在各个手机样式类下增加。
- 2) 违反了单一职责原则, 当我们增加手机样式时, 要同时增加所有品牌的手机, 这样增加了代码维护成本。
- 3) 解决方案-使用桥接模式

10.4 桥接模式(Bridge)-基本介绍

基本介绍

- 1) 桥接模式(Bridge 模式)是指: 将实现与抽象放在两个不同的类层次中, 使两个层次可以独立改变。
- 2) 是一种结构型设计模式
- 3) Bridge 模式基于类的最小设计原则, 通过使用封装、聚合及继承等行为让不同的类承担不同的职责。它的主要特点是把抽象(Abstraction)与行为实现(Implementation)分离开来, 从而可以保持各部分的独立性以及应对他们的功能扩展

10.5 桥接模式(Bridge)-原理类图



➤ 上图做了说明

- 1) Client 类: 桥接模式的调用者
- 2) 抽象类(Abstraction) : 维护了 Implementor / 即它的实现类 ConcreteImplementorA.., 二者是聚合关系, Abstraction

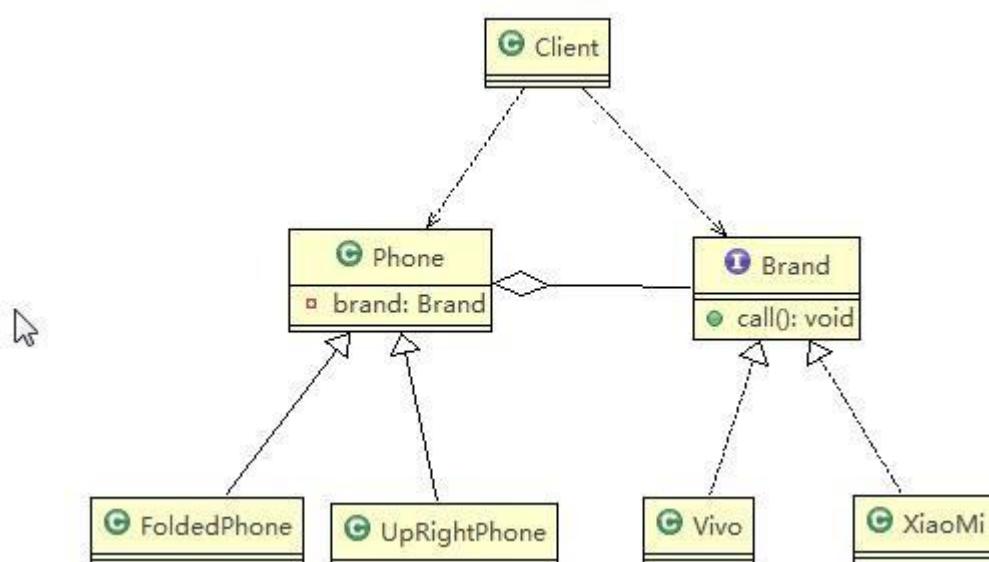
充当桥接类

- 3) RefinedAbstraction：是 Abstraction 抽象类的子类
- 4) Implementor：行为实现类的接口
- 5) ConcreteImplementorA /B : 行为的具体实现类
- 6) 从 UML 图：这里的抽象类和接口是聚合的关系，其实调用和被调用关系

10.6 桥接模式解决手机操作问题

使用桥接模式改进传统方式，让程序具有搞好的扩展性，利用程序维护

- 1) 应用实例说明(和前面要求一样)
- 2) 使用桥接模式对应的类图



- 3) 代码



```
package com.atguigu.bridge;
```



```
//接口
public interface Brand {
    void open();
    void close();
    void call();
}

package com.atguigu.bridge;

public class Client {

    public static void main(String[] args) {
        //获取折叠式手机 (样式 + 品牌 )
        Phone phone1 = new FoldedPhone(new XiaoMi());

        phone1.open();
        phone1.call();
        phone1.close();

        System.out.println("=====");
        Phone phone2 = new FoldedPhone(new Vivo());
        phone2.open();
    }
}
```



```
phone2.call();
phone2.close();

System.out.println("=====");
UpRightPhone phone3 = new UpRightPhone(new XiaoMi());

phone3.open();
phone3.call();
phone3.close();

System.out.println("=====");
UpRightPhone phone4 = new UpRightPhone(new Vivo());

phone4.open();
phone4.call();
phone4.close();
}

}

package com.atguigu.bridge;

//折叠式手机类，继承抽象类 Phone
```



```
public class FoldedPhone extends Phone {
```

```
    //构造器
```

```
    public FoldedPhone(Brand brand)
```

```
        { super(brand);
```

```
    }
```

```
    public void open()
```

```
        { super.open();
```

```
            System.out.println(" 折叠样式手机 ");
```

```
    }
```

```
    public void close()
```

```
        { super.close();
```

```
            System.out.println(" 折叠样式手机 ");
```

```
    }
```

```
    public void call()
```

```
        { super.call();
```

```
            System.out.println(" 折叠样式手机 ");
```

```
    }
```

```
}
```

```
package com.atguigu.bridge;
```

```
public abstract class Phone {
```



```
//组合品牌
private Brand brand;

//构造器
public Phone(Brand brand)
{
    super();
    this.brand = brand;
}

protected void open()
{
    this.brand.open();
    ;
}

protected void close()
{
    brand.close();
}

protected void call()
{
    brand.call();
}

}

package com.atguigu.bridge;

public class UpRightPhone extends Phone {
```



```
//构造器
public UpRightPhone(Brand brand)
{
    super(brand);
}

public void open()
{
    super.open();
    System.out.println(" 直立样式手机 ");
}

public void close()
{
    super.close();
    System.out.println(" 直立样式手机 ");
}

public void call()
{
    super.call();
    System.out.println(" 直立样式手机 ");
}

}

package com.atguigu.bridge;

public class Vivo implements Brand {

    @Override
}
```



```
public void open() {  
    // TODO Auto-generated method stub  
    System.out.println(" Vivo 手机开机 ");  
}
```

```
@Override  
public void close() {  
    // TODO Auto-generated method stub  
    System.out.println(" Vivo 手机关机 ");  
}
```

```
@Override  
public void call() {  
    // TODO Auto-generated method stub  
    System.out.println(" Vivo 手机打电话 ");  
}
```

```
}
```

```
package com.atguigu.bridge;  
  
public class XiaoMi implements Brand {  
  
    @Override  
    public void open() {  
        // TODO Auto-generated method stub  
    }
```



```
System.out.println(" 小米手机开机 ");

}

@Override
public void close() {
    // TODO Auto-generated method stub
    System.out.println(" 小米手机关机 ");
}

@Override
public void call() {
    // TODO Auto-generated method stub
    System.out.println(" 小米手机打电话 ");
}

}
```

10.7 桥接模式在 JDBC 的源码剖析

桥接模式在 JDBC 的源码剖析

- 1) Jdbc 的 **Driver** 接口，如果从桥接模式来看，Driver 就是一个接口，下面可以有 MySQL 的 Driver，Oracle 的 Driver，这些就可以当做实现接口类
- 2) 代码分析+Debug 源码

```

public class Driver extends NonRegisteringDriver implements
    java.sql.Driver {
    public Driver() throws SQLException {
    }
    static {
        try {
            //1.注册驱动
            //2.调用DriverManager中的getConnection
            DriverManager.registerDriver(new Driver());
        } catch (SQLException var1) {
            throw new RuntimeException("Can't register driver!");
        }
    }
}

```

DriverManager

- registeredDrivers : CopyOnWriteArrayList<DriverInfo>
- loginTimeout : int
- logWriter : PrintWriter
- logStream : PrintStream
- logSync : Object
- DriverManager()
- SET_LOG_PERMISSION : SQLPermission
- Deregister_Driver_Permission : SQLPermission
- getLogWriter() : PrintWriter
- setLogWriter(PrintWriter)
- getConnnection(String, Properties) : Connection
- getConnnection(String, String, String) : Connection
- getConnnection(String) : Connection
- delDriverString(Driver)

```

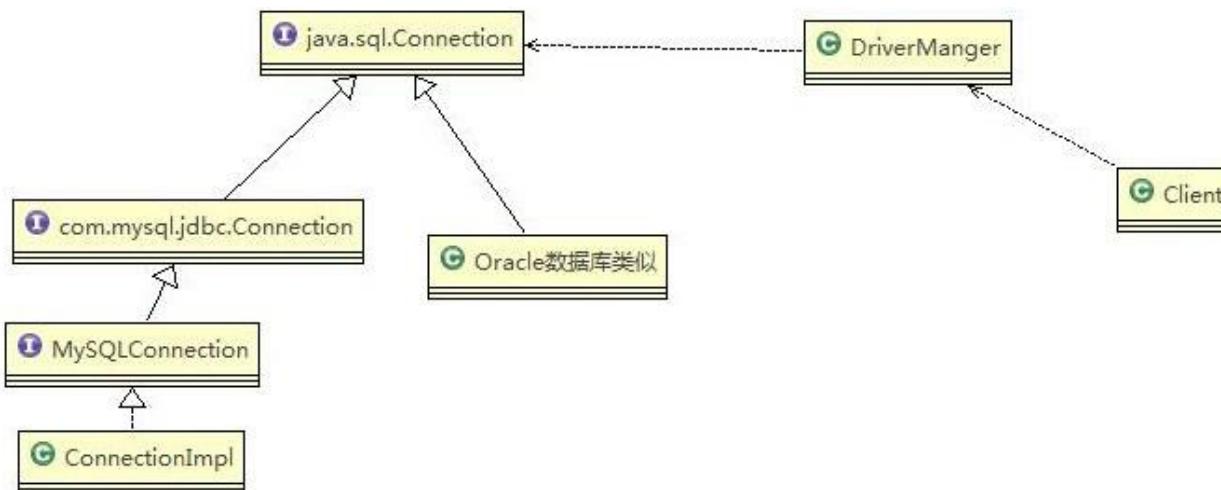
public class
ConnectionImpl
extends
ConnectionPrope
siml implement
MySQLConnectio
}

```

说明：

从上到下看，每一个类都有自己的抽象方法实现，同样地，每一个类也有对应的实现方法。

- 对 jdbc 源码分析的类图



10.8 桥接模式的注意事项和细节

- 1) 实现了抽象和实现部分的分离，从而极大的提供了系统的灵活性，让抽象部分和实现部分独立开来，这有助于系统进行分层设计，从而产生更好的结构化系统。
- 2) 对于系统的高层部分，只需要知道抽象部分和实现部分的接口就可以了，其它的部分由具体业务来完成。
- 3) 桥接模式替代多层继承方案，可以减少子类的个数，降低系统的管理和维护成本。



-
- 4) 桥接模式的引入增加了系统的理解和设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计和编程
 - 5) 桥接模式要求正确识别出系统中两个独立变化的维度(抽象、和实现)，因此其使用范围有一定的局限性，即需要有这样的应用场景。

桥接模式其它应用场景

对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。

10.9 常见的应用场景：

1) -JDBC 驱动程序

2) -银行转账系统

转账分类: 网上转账，柜台转账，AMT 转账

转账用户类型: 普通用户，银卡用户，金卡用户..

3) -消息管理

消息类型: 即时消息，延时消息

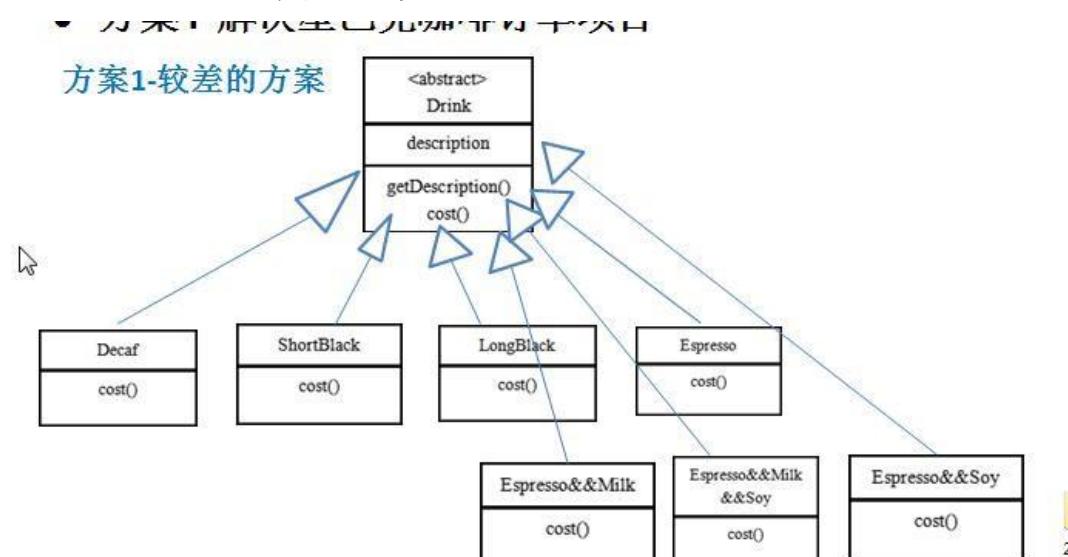
消息分类: 手机短信，邮件消息，QQ 消息...

第 11 章 装饰者设计模式

11.1 星巴克咖啡订单项目（咖啡馆）：

- 1) 咖啡种类/单品咖啡：Espresso(意大利浓咖啡)、ShortBlack、LongBlack(美式咖啡)、Decaf(无因咖啡)
- 2) 调料：Milk、Soy(豆浆)、Chocolate
- 3) 要求在扩展新的咖啡种类时，具有良好的扩展性、改动方便、维护方便
- 4) 使用 OO 的来计算不同种类咖啡的费用：客户可以点单品咖啡，也可以单品咖啡+调料组合。

11.2 方案1-解决星巴克咖啡订单项目



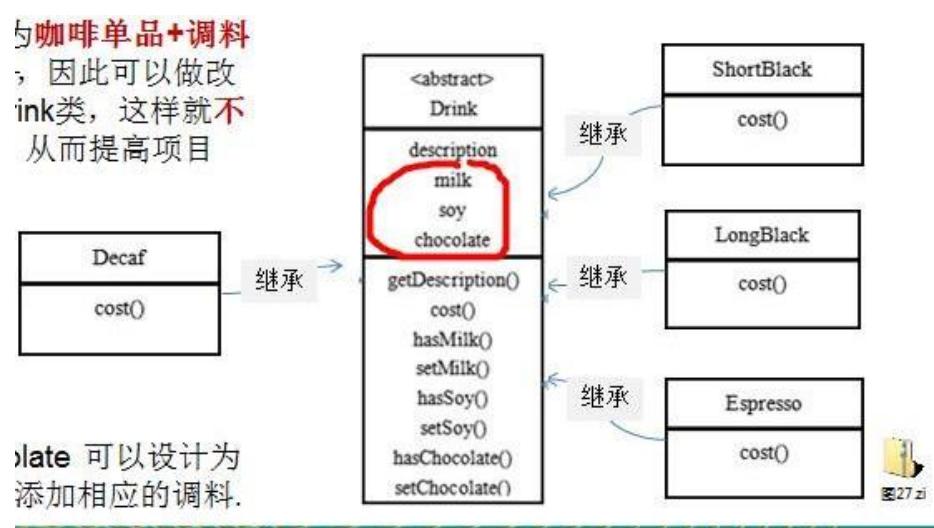
11.3 方案1-解决星巴克咖啡订单问题分析

- 1) Drink 是一个抽象类，表示饮料
- 2) des 就是对咖啡的描述，比如咖啡的名字
- 3) cost() 方法就是计算费用，Drink 类中做成一个抽象方法.
- 4) Decaf 就是单品咖啡，继承 Drink，并实现 cost
- 5) Espresso && Milk 就是单品咖啡+调料，这个组合很多

- 6) 问题：这样设计，会有很多类，当我们增加一个单品咖啡，或者一个新的调料，类的数量就会倍增，就会出现类爆炸

11.4 方案2-解决星巴克咖啡订单(好点)

- 1) 前面分析到方案1因为咖啡单品+调料组合会造成类的倍增，因此可以做改进，将调料内置到Drink类，这样就不会造成类数量过多。从而提高项目的维护性(如图)



- 2) 说明: milk, soy, chocolate 可以设计为 Boolean, 表示是否要添加相应的调料.

11.5 方案2-解决星巴克咖啡订单问题分析

- 1) 方案2可以控制类的数量，不至于造成很多的类
- 2) 在增加或者删除调料种类时，代码的维护量很大
- 3) 考虑到用户可以添加多份调料时，可以将 hasMilk 返回一个对应int
- 4) 考虑使用装饰者模式

11.6 装饰者模式定义

- 1) 装饰者模式：动态的将新功能附加到对象上。在对象功能扩展方面，它比继承更有弹性，装饰者模式也体现了开闭原则(**ocp**)
- 2) 这里提到的动态的将新功能附加到对象和 ocp 原则，在后面的应用实例上会以代码的形式体现，请同学们注意体会。

11.7 装饰者模式原理

- 1) 装饰者模式就像打包一个快递

主体：比如：陶瓷、衣服 (Component) // 被装饰者

包装：比如：报纸填充、塑料泡沫、纸板、木板(Decorator)

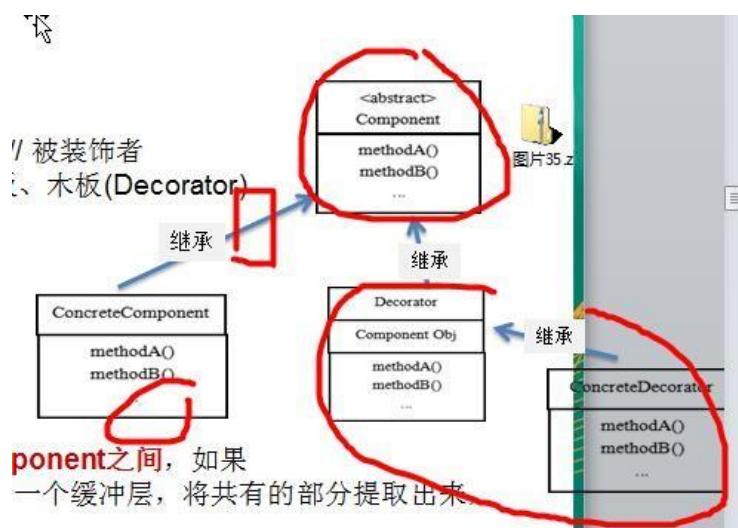
- 2) Component 主体：比如类似前面的 Drink

- 3) ConcreteComponent 和 Decorator

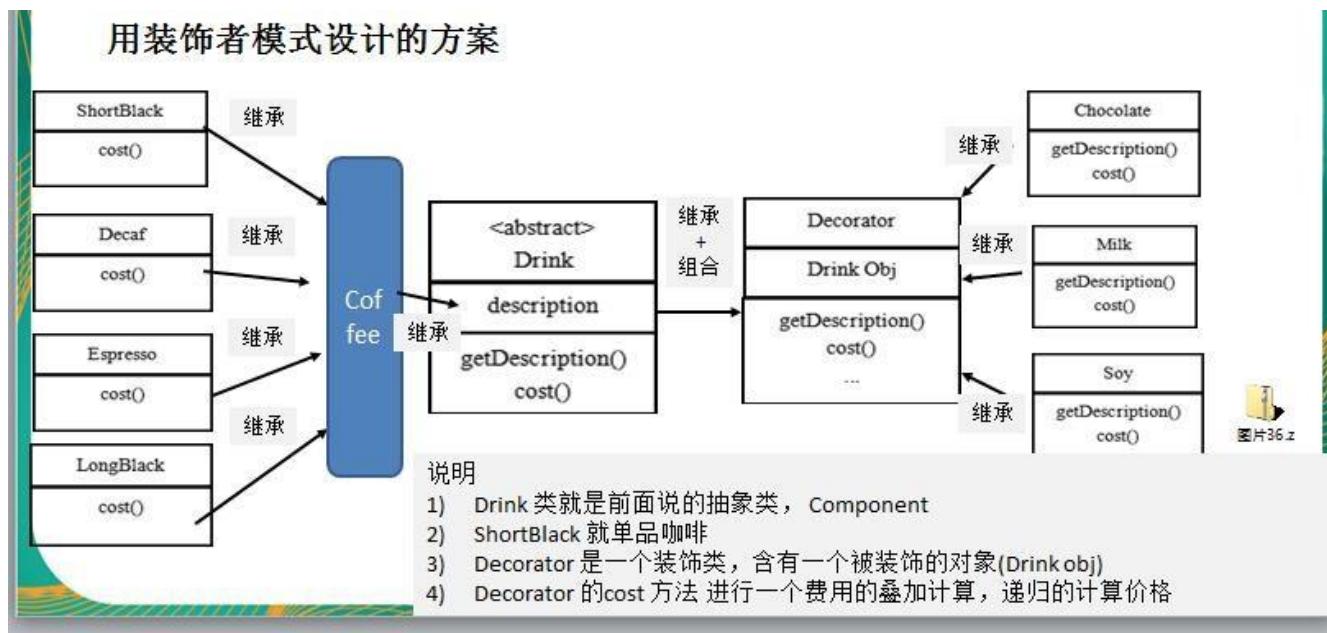
ConcreteComponent：具体的主体，
比如前面的各个单品咖啡

- 4) Decorator: 装饰者，比如各调料.

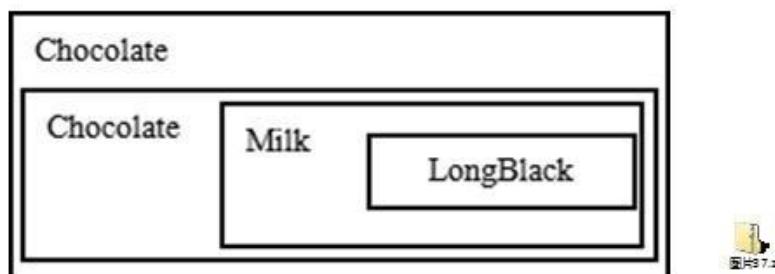
在如图的 Component 与 ConcreteComponent 之间，如果 ConcreteComponent 类很多，还可以设计一个缓冲层，将共有的部分提取出来，抽象层一个类。



11.8 装饰者模式解决星巴克咖啡订单



11.9 装饰者模式下的订单：2 份巧克力+一份牛奶的 LongBlack



说明

- 1) Milk包含了LongBlack
- 2) 一份Chocolate包含了(Milk+LongBlack)
- 3) 一份Chocolate包含了(Chocolate+Milk+LongBlack)
- 4) 这样不管是任何形式的单品咖啡+调料组合，通过递归方式可以方便的组合和维护。

11.10 装饰者模式咖啡订单项目应用实例

咖啡订单项目包结构



decorator.zip

```
package com.atguigu.decorator;

//具体的 Decorator， 这里就是调味品
public class Chocolate extends Decorator {

    public Chocolate(Drink obj)

        { super(obj);

            setDes(" 巧克力 ");

            setPrice(3.0f); // 调味品 的价格

        }

}
```

```
package com.atguigu.decorator;

public class Coffee   extends Drink {

    @Override

    public float cost()

        // TODO Auto-generated method stub

        return super.getPrice();

}
```



```
}
```

```
package com.atguigu.decorator;

public class CoffeeBar {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // 装饰者模式下的订单：2份巧克力+一份牛奶的 LongBlack

        // 1. 点一份 LongBlack
        Drink order = new LongBlack();

        System.out.println("费用 1=" + order.cost());
        System.out.println("描述=" + order.getDes());

        // 2. order 加入一份牛奶
        order = new Milk(order);

        System.out.println("order 加入一份牛奶 费用 =" + order.cost());
        System.out.println("order 加入一份牛奶 描述 = " + order.getDes());

        // 3. order 加入一份巧克力

        order = new Chocolate(order);

        System.out.println("order 加入一份牛奶 加入一份巧克力 费用 =" + order.cost());
    }
}
```



```
System.out.println("order 加入一份牛奶 加入一份巧克力 描述 = " + order.getDes());  
  
// 3. order 加入一份巧克力  
  
order = new Chocolate(order);  
  
System.out.println("order 加入一份牛奶 加入 2 份巧克力 费用 =" + order.cost());  
System.out.println("order 加入一份牛奶 加入 2 份巧克力 描述 = " + order.getDes());  
  
System.out.println("=====");  
  
Drink order2 = new DeCaf();  
  
System.out.println("order2 无因咖啡 费用 =" + order2.cost());  
System.out.println("order2 无因咖啡 描述 = " + order2.getDes());  
  
order2      =      new      Milk(order2);  
  
System.out.println("order2 无因咖啡 加入一份牛奶 费用 =" + order2.cost());  
System.out.println("order2 无因咖啡 加入一份牛奶 描述 = " + order2.getDes());  
  
}  
  
}
```



```
package com.atguigu.decorator;

public class DeCaf extends Coffee {

    public DeCaf() {
        setDes("无因咖啡");
        setPrice(1.0f);
    }
}
```

```
package com.atguigu.decorator;

public class Decorator extends Drink
{
    private Drink obj;

    public Decorator(Drink obj) { //组合
        // TODO Auto-generated constructor stub
        this.obj = obj;
    }

    @Override
    public float cost() {
        // TODO Auto-generated method stub
        // getPrice 自己价格
        return super.getPrice() + obj.cost();
    }
}
```



```
@Override  
public String getDes() {  
    // TODO Auto-generated method stub  
    // obj.getDes() 输出被装饰者的信息  
    return des + " " + getPrice() + " && " + obj.getDes();  
}  
  
}
```

```
package com.atguigu.decorator;  
  
public abstract class Drink {  
    { public String des; // 描述  
    private float price = 0.0f;  
    public String getDes() {  
        return des;  
    }  
    public void setDes(String des) {  
        this.des = des;  
    }  
}
```



```
public float getPrice()  
{ return price;
```



```
}
```

```
public void setPrice(float price)
```

```
{ this.price = price;
```

```
}
```

```
//计算费用的抽象方法
```

```
//子类来实现
```

```
public abstract float cost();
```

```
}
```

```
package com.atguigu.decorator;
```

```
public class Espresso extends Coffee {
```

```
    public Espresso() {
```

```
        setDes(" 意大利咖啡 ");
```

```
        setPrice(6.0f);
```

```
}
```

```
}
```

```
package com.atguigu.decorator;
```

```
public class LongBlack extends Coffee {
```

```
    public LongBlack() {
```



```
    setDes(" longblack ");
    setPrice(5.0f);
}
}
```

```
package com.atguigu.decorator;
```

```
public class Milk extends Decorator {
```

```
    public Milk(Drink obj)
    {
        super(obj);
        // TODO Auto-generated constructor stub
        setDes(" 牛 奶 ");
        setPrice(2.0f);
    }
}
```

```
package com.atguigu.decorator;
```

```
public class ShortBlack extends Coffee{
```

```
    public ShortBlack()
    {
        setDes(" shortblack
"); setPrice(4.0f);
    }
}
```

```
}
```

```
package com.atguigu.decorator;
```

```
public class Soy extends Decorator{
```

```
    public Soy(Drink obj)
```

```
    { super(obj);
```

```
        // TODO Auto-generated constructor stub
```

```
        setDes(" 豆浆 ");
```

```
        setPrice(1.5f);
```

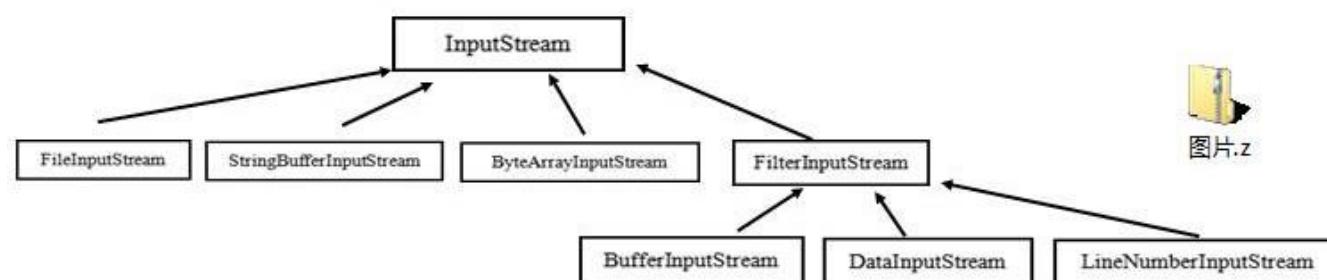
```
}
```

```
}
```

11.11 装饰者模式在 JDK 应用的源码分析

Java 的 IO 结构，FilterInputStream 就是一个装饰者

Java 的 IO 结构，FilterInputStream 就是一个装饰者



源码说明



```
package com.atguigu.jdk;

import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStream;

public class Decorator {

    public static void main(String[] args) throws Exception{
        // TODO Auto-generated method stub

        //说明
        //1. InputStream 是抽象类，类似我们前面讲的 Drink
        //2. FileInputStream 是 InputStream 子类，类似我们前面的 DeCaf, LongBlack
        //3. FilterInputStream 是 InputStream 子类：类似我们前面的 Decorator 修饰者
        //4. DataInputStream 是 FilterInputStream 子类，具体的修饰者，类似前面的 Milk, Soy 等
        //5. FilterInputStream 类 有 protected volatile InputStream in; 即含被装饰者
        //6. 分析得出在 jdk 的 io 体系中，就是使用装饰者模式

        DataInputStream dis = new DataInputStream(new FileInputStream("d:\\abc.txt"));
        System.out.println(dis.read());
        dis.close();
    }
}
```

第 12 章 组合模式

12.1 看一个学校院系展示需求

编写程序展示一个学校院系结构：需求是这样，要在一个页面中展示出学校的院系组成，一个学校有多个学院，一个学院有多个系。如图：

```
-----清华大学-----  
-----计算机学院-----  
计算机科学与技术  
软件工程  
网络工程  
-----信息工程学院-----  
通信工程  
信息工程
```

12.2 传统方案解决学校院系展示(类图)



12.3 传统方案解决学校院系展示存在的问题分析

- 1) 将学院看做是学校的子类，系是学院的子类，这样实际上是站在组织大小来进行分层次的
- 2) 实际上我们的要求是：在一个页面中展示出学校的院系组成，一个学校有多个学院，一个学院有多个系，因

此这种方案，不能很好实现的管理的操作，比如对学院、系的添加，删除，遍历等

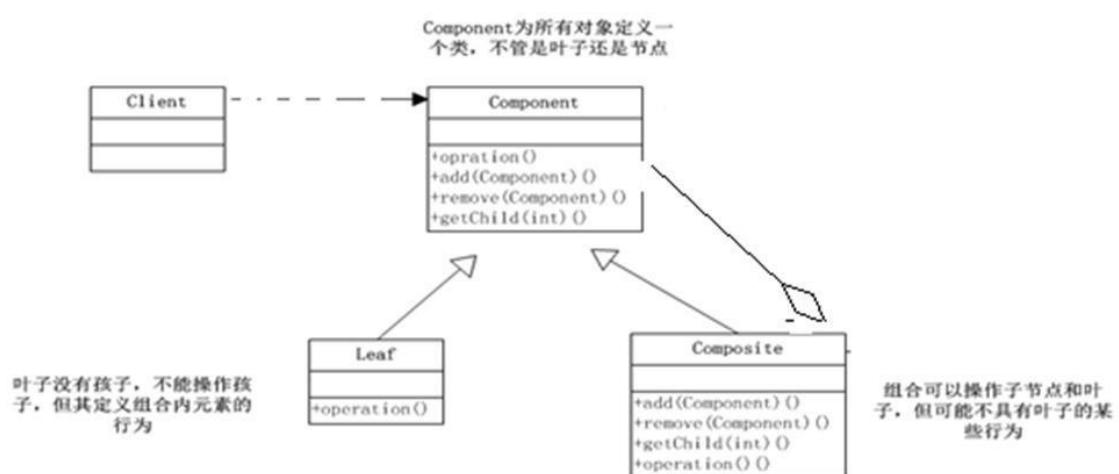
- 3) 解决方案：把学校、院、系都看做是组织结构，他们之间没有继承的关系，而是一个树形结构，可以更好的实现管理操作。=> 组合模式

12.4 组合模式基本介绍

基本介绍

- 1) 组合模式（Composite Pattern），又叫部分整体模式，它创建了对象组的树形结构，将对象组合成树状结构以表示“整体-部分”的层次关系。
- 2) 组合模式依据树形结构来组合对象，用来表示部分以及整体层次。
- 3) 这种类型的设计模式属于结构型模式。
- 4) 组合模式使得用户对单个对象和组合对象的访问具有一致性，即：组合能让客户以一致的方式处理个别对象以及组合对象

12.5 组合模式原理类图



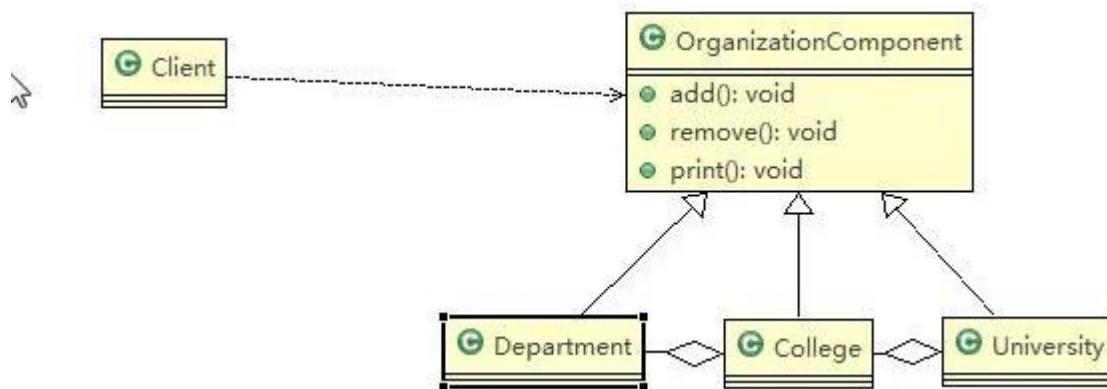
对原理结构图的说明-即(组合模式的角色及职责)

- 1) **Component** : 这是组合中对象声明接口，在适当情况下，实现所有类共有的接口默认行为，用于访问和管理 Component 子部件，Component 可以是抽象类或者接口
- 2) **Leaf** : 在组合中表示叶子节点，叶子节点没有子节点
- 3) **Composite** : 非叶子节点，用于存储子部件，在 Component 接口中实现子部件的相关操作，比如增加(add)，删除。

12.6 组合模式解决学校院系展示的应用实例

应用实例要求

- 1) 编写程序展示一个学校院系结构：需求是这样，要在一个页面中展示出学校的院系组成，一个学校有多个学院，一个学院有多个系。
- 2) 思路分析和图解(类图)



- 3) 代码实现



composite.zip

```
package com.atguigu.composite;
```



```
public class Client {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        //从大到小创建对象 学校  
        OrganizationComponent university = new University("清华大学", "中国顶级大学");  
  
        //创建 学院  
        OrganizationComponent computerCollege = new College("计算机学院", "计算机学院");  
        OrganizationComponent infoEngineercollege = new College("信息工程学院", "信息工程学院");  
  
        //创建各个学院下面的系(专业)  
        computerCollege.add(new Department("软件工程", "软件工程不错"));  
        computerCollege.add(new Department("网络工程", "网络工程不错"));  
        computerCollege.add(new Department("计算机科学与技术", "计算机科学与技术是老牌的专业"));  
  
        //  
        infoEngineercollege.add(new Department("通信工程", "通信工程不好学"));  
        infoEngineercollege.add(new Department("信息工程", "信息工程好学"));  
  
        //将学院加入到 学校  
        university.add(computerCollege);  
        university.add(infoEngineercollege);  
    }  
}
```



```
//university.print();
infoEngineercollege.print();
}
```

```
}
```

```
package com.atguigu.composite;

import java.util.ArrayList;
import java.util.List;

public class College extends OrganizationComponent {

    //List 中存放的 Department
    List<OrganizationComponent> organizationComponents = new ArrayList<OrganizationComponent>();

    // 构造器
    public College(String name, String des)
    {
        super(name, des);
        // TODO Auto-generated constructor stub
    }

    // 重写 add
    @Override
    protected void add(OrganizationComponent organizationComponent) {
        // TODO Auto-generated method stub
    }
}
```



```
// 将来实际业务中，Colleague 的 add 和 University add 不一定完全一样
organizationComponents.add(organizationComponent);

}

// 重写 remove
@Override
protected void remove(OrganizationComponent organizationComponent) {
    // TODO Auto-generated method stub
    organizationComponents.remove(organizationComponent);
}

@Override
public String getName() {
    // TODO Auto-generated method stub
    return super.getName();
}

@Override
public String getDes() {
    // TODO Auto-generated method stub
    return super.getDes();
}

// print 方法，就是输出 University 包含的学院
@Override
protected void print() {
```



```
// TODO Auto-generated method stub  
  
System.out.println("-----" + getName() + "-----");  
//遍历 organizationComponents  
for (OrganizationComponent organizationComponent : organizationComponents)  
    { organizationComponent.print();  
    }  
}  
  
}
```

```
package com.atguigu.composite;  
  
public class Department extends OrganizationComponent {  
  
    //没有集合  
  
    public Department(String name, String des)  
    { super(name, des);  
        // TODO Auto-generated constructor stub  
    }  
  
    //add , remove 就不用写了， 因为他是叶子节点  
  
    @Override
```



```
public String getName() {  
    // TODO Auto-generated method stub  
    return super.getName();  
}
```

```
@Override  
public String getDes() {  
    // TODO Auto-generated method stub  
    return super.getDes();  
}
```

```
@Override  
protected void print() {  
    // TODO Auto-generated method stub  
    System.out.println(getName());  
}
```

```
}  
  
package com.atguigu.composite;  
  
public abstract class OrganizationComponent {  
    { private String name; // 名字  
    private String des; // 说明  
}
```



```
protected void add(OrganizationComponent organizationComponent) {  
    //默认实现  
    throw new UnsupportedOperationException();  
}  
  
protected void remove(OrganizationComponent organizationComponent) {  
    //默认实现  
    throw new UnsupportedOperationException();  
}  
  
//构造器  
public OrganizationComponent(String name, String des)  
{ super();  
    this.name = name;  
    this.des = des;  
}  
  
public String getName()  
{ return name;  
}  
  
public void setName(String name)  
{ this.name = name;  
}  
  
public String getDes() {
```



```
    return des;  
}  
  
public void setDes(String des)  
{ this.des = des;  
}  
  
//方法 print, 做成抽象的, 子类都需要实现  
protected abstract void print();  
  
}
```

```
package com.atguigu.composite;  
  
import java.util.ArrayList;  
import java.util.List;  
  
//University 就是 Composite , 可以管理 College  
public class University extends OrganizationComponent {  
  
    List<OrganizationComponent> organizationComponents = new ArrayList<OrganizationComponent>();  
  
    // 构造器  
    public University(String name, String des) {  
        super(name, des);  
    }
```



```
// TODO Auto-generated constructor stub

}

// 重写 add
@Override
protected void add(OrganizationComponent organizationComponent) {
    // TODO Auto-generated method stub
    organizationComponents.add(organizationComponent);
}

// 重写 remove
@Override
protected void remove(OrganizationComponent organizationComponent) {
    // TODO Auto-generated method stub
    organizationComponents.remove(organizationComponent);
}

@Override
public String getName() {
    // TODO Auto-generated method stub
    return super.getName();
}

@Override
public String getDes() {
    // TODO Auto-generated method stub
}
```



```
return super.getDes();

}

// print 方法，就是输出 University 包含的学院
@Override

protected void print() {
    // TODO Auto-generated method stub
    System.out.println("-----" + getName() + "-----");
    //遍历 organizationComponents
    for (OrganizationComponent organizationComponent : organizationComponents)
        { organizationComponent.print();
    }
}

}
```

12.7 组合模式在 JDK 集合的源码分析

- 1) Java 的集合类-**HashMap** 就使用了组合模式
- 2) 代码分析+Debug 源码

1) 代码分析

```

public class HashMapComp {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Map<Integer, String> hashMap = new HashMap<Integer, String>();
        hashMap.put(0, "东游记");//直接存放叶子节点
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(1, "西游记");
        map.put(2, "红楼梦");
        hashMap.putAll(map);
        System.out.println(hashMap);
    }
}

2) 公共接口
public interface Map<K,V> {
    //add, remove...
}

3) 抽象类
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
}

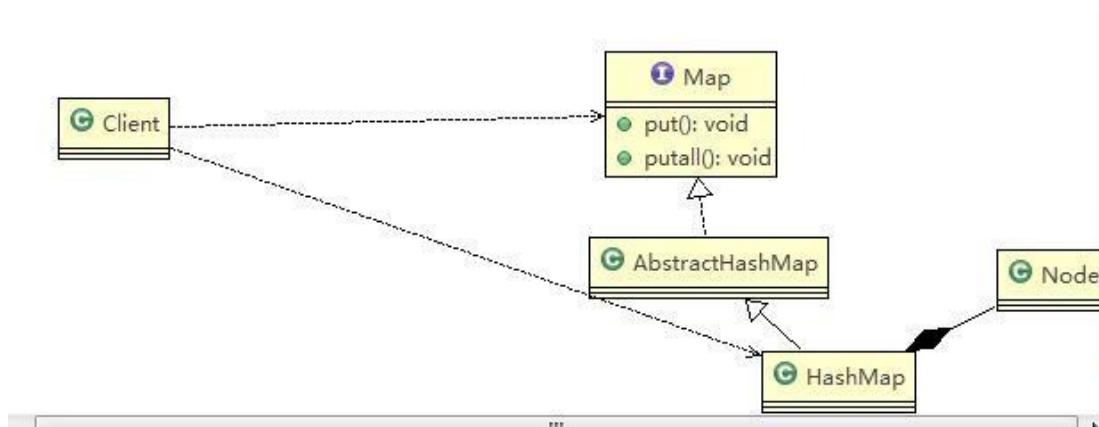
4) 实现类
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
}

```

Component

Comp

3) 类图



12.8 组合模式的注意事项和细节

- 1) 简化客户端操作。客户端只需要面对一致的对象而不用考虑整体部分或者节点叶子的问题。
- 2) 具有较强的扩展性。当我们要更改组合对象时，我们只需要调整内部的层次关系，客户端不用做出任何改动。
- 3) 方便创建出复杂的层次结构。客户端不用理会组合里面的组成细节，容易添加节点或者叶子从而创建出复杂的树形结构。
- 4) 需要遍历组织机构，或者处理的对象具有树形结构时，非常适合使用组合模式。
- 5) 要求较高的抽象性，如果节点和叶子有很多差异性的话，比如很多方法和属性都不一样，不适合使用组合模式。

第 13 章 外观模式

13.1 影院管理项目

组建一个家庭影院：

DVD 播放器、投影仪、自动屏幕、环绕立体声、爆米花机,要求完成使用家庭影院的功能，其过程为：

直接用遥控器：统筹各设备开关

开爆米花机

放下屏幕

开投影仪

开音响

开 DVD，选 dvd

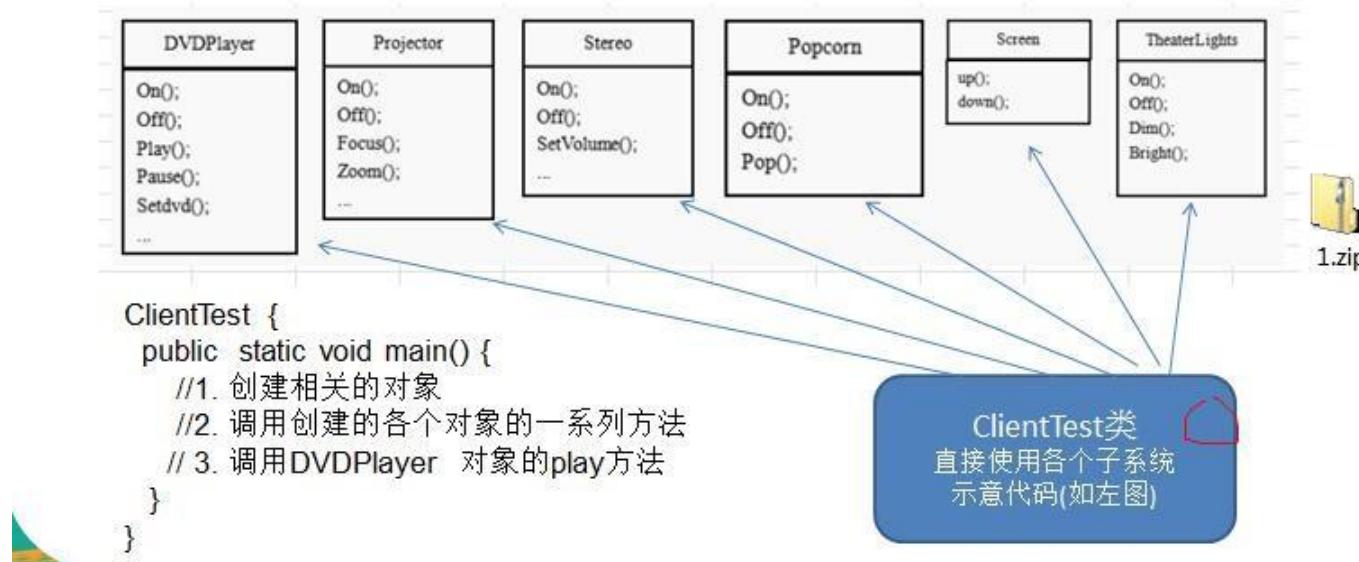
去拿爆米花

调暗灯光

播放

观影结束后，关闭各种设备

13.2 传统方式解决影院管理



13.3 传统方式解决影院管理问题分析

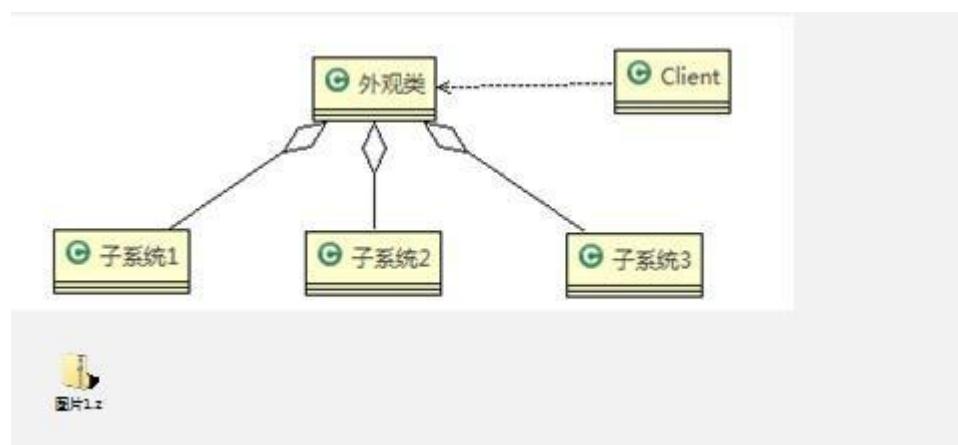
- 1) 在 ClientTest 的 main 方法中，创建各个子系统的对象，并直接去调用子系统(对象)相关方法，会造成调用过程混乱，没有清晰的过程
- 2) 不利于在 ClientTest 中，去维护对子系统的操作
- 3) 解决思路：定义一个高层接口，给子系统中的一组接口提供一个一致的界面(比如在高层接口提供四个方法 ready, play, pause, end)，用来访问子系统中的一群接口
- 4) 也就是说就是通过定义一个一致的接口(界面类)，用以屏蔽内部子系统的细节，使得调用端只需跟这个接口发生调用，而无需关心这个子系统的内部细节=> 外观模式

13.4 外观模式基本介绍

基本介绍

- 1) 外观模式（Facade），也叫“过程模式：外观模式为子系统中的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口使得这一子系统更加容易使用
- 2) 外观模式通过定义一个一致的接口，用以屏蔽内部子系统的细节，使得调用端只需跟这个接口发生调用，而无需关心这个子系统的内部细节

13.5 外观模式原理类图

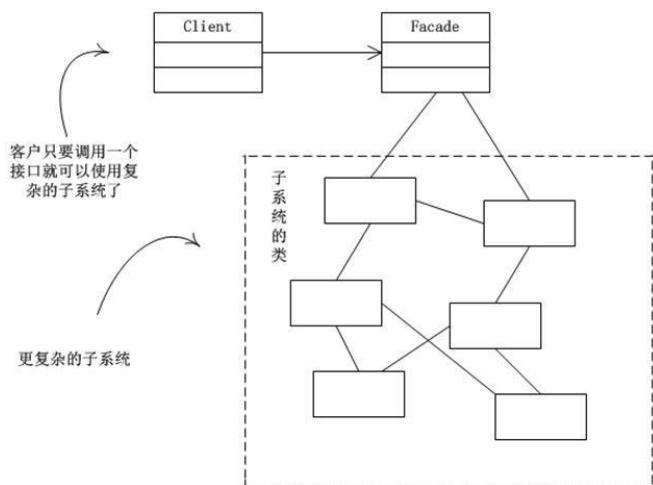


- 对类图说明(分类外观模式的角色)
- 1) 外观类(Facade): 为调用端提供统一的调用接口, 外观类知道哪些子系统负责处理请求, 从而将调用端的请求代理给适当子系统对象
 - 2) 调用者(Client): 外观接口的调用者
 - 3) 子系统的集合: 指模块或者子系统, 处理 Facade 对象指派的任务, 他是功能的实际提供者

13.6 外观模式解决影院管理

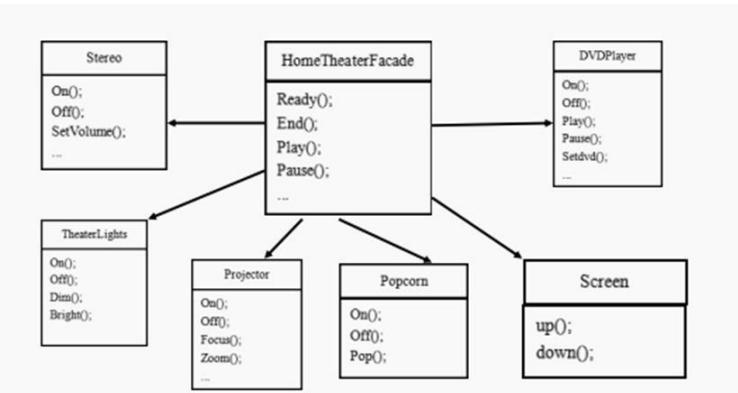
13.6.1 传统方式解决影院管理说明

- 1) 外观模式可以理解为转换一群接口, 客户只要调用一个接口, 而不用调用多个接口才能达到目的。比如: 在 pc 上安装软件的时候经常有一键安装选项 (省去选择安装目录、安装的组件等等), 还有就是手机的重启功能 (把关机和启动合为一个操作)。
- 2) 外观模式就是解决多个复杂接口带来的使用困难, 起到简化用户操作的作用
- 3) 示意图说明



13.6.2 外观模式应用实例

- 1) 应用实例要求
- 2) 使用外观模式来完成家庭影院项目
- 3) 思路分析和图解(类图)



- 4) 代码实现



facade.zip

```
package com.atguigu.facade;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //这里直接调用。。很麻烦
        HomeTheaterFacade homeTheaterFacade = new HomeTheaterFacade();

        homeTheaterFacade.ready();
        homeTheaterFacade.play();

        homeTheaterFacade.end();
    }
}
```



```
}
```

```
}
```

```
package com.atguigu.facade;
```

```
public class DVDPlayer {
```

```
    //使用单例模式, 使用饿汉式
```

```
    private static DVDPlayer instance = new DVDPlayer();
```

```
    public static DVDPlayer getInstance()
```

```
    { return instance;
```

```
}
```

```
    public void on()
```

```
    { System.out.println(" dvd on
```

```
        ");
```

```
}
```

```
    public void off()
```

```
    { System.out.println(" dvd off
```

```
        ");
```

```
}
```

```
    public void play() {
```

```
        System.out.println(" dvd is playing ");
```

```
}
```



```
//....  
public void pause()  
{ System.out.println(" dvd  
    pause ..");  
}  
}
```

```
package com.atguigu.facade;
```

```
public class HomeTheaterFacade {
```

```
//定义各个子系统对象  
private TheaterLight theaterLight;  
private Popcorn popcorn;  
private Stereo stereo;  
private Projector projector;  
private Screen screen;  
private DVDPlayer dVDPlayer;
```

```
//构造器
```

```
public HomeTheaterFacade()  
{ super();  
    this.theaterLight = TheaterLight.getInstance();  
    this.popcorn = Popcorn.getInstance();  
    this.stereo = Stereo.getInstance();  
    this.projector = Projector.getInstance();
```



```
this.screen = Screen.getInstance();
this.dVDPlayer = DVDPlayer.getInstanc();
}

//操作分成 4 步

public void ready()
{
    popcorn.on();
    popcorn.pop();
    screen.down();
    projector.on();
    stereo.on();
    dVDPlayer.on();
    theaterLight.dim();
}

public void play()
{
    dVDPlayer.play();
}

public void pause()
{
    dVDPlayer.pause();
}

public void end()
{
    popcorn.off();
}
```



```
theaterLight.bright();
screen.up();
projector.off();
stereo.off();
dVDPlayer.off();

}
```

```
}
```

```
package com.atguigu.facade;

public class Popcorn {

    private static Popcorn instance = new Popcorn();

    public static Popcorn getInstance()
    {
        return instance;
    }

    public void on()
    {
        System.out.println(" popcorn on ");
    }
}
```



```
public void off()
{
    System.out.println(" popcorn ff
");
}

public void pop() {
    System.out.println(" popcorn is poping  ");
}
}

package com.atguigu.facade;

public class Projector {

    private static Projector instance = new Projector();

    public static Projector getInstance()
    {
        return instance;
    }

    public void on() {
        System.out.println(" Projector on ");
    }

    public void off() {
        System.out.println(" Projector ff ");
    }
}
```



}

```
public void focus() {  
    System.out.println(" Projector is Projector ");  
}  
  
//...
```

```
package com.atguigu.facade;
```

```
public class Screen {
```

```
private static Screen instance = new Screen();
```

```
public static Screen getInstance()  
{ return instance;  
}
```

```
public void up()
{ System.out.println(" Screen up
");
}
```



```
public void down()
{
    System.out.println(" Screen down
");
}
```

```
package com.atguigu.facade;
```

```
public class Stereo {

    private static Stereo instance = new Stereo();

    public static Stereo getInstance()
    {
        return instance;
    }

    public void on()
    {
        System.out.println(" Stereo on
");
    }

    public void off()
    {
        System.out.println(" Screen off
");
    }
}
```



```
public void up() {
```



```
System.out.println(" Screen up.. ");

}

//...

}

package com.atguigu.facade;

public class TheaterLight {

    private static TheaterLight instance = new TheaterLight();

    public static TheaterLight getInstance()
    {
        return instance;
    }

    public void on()
    {
        System.out.println(" TheaterLight on ");
    }

    public void off()
    {
        System.out.println(" TheaterLight off ");
    }

    public void dim()
    {
        System.out.println(" TheaterLight dim.. ");
    }
}
```

```
}

public void bright() {
    System.out.println(" TheaterLight bright.. ");
}

}
```

13.7 外观模式在 MyBatis 框架应用的源码分析

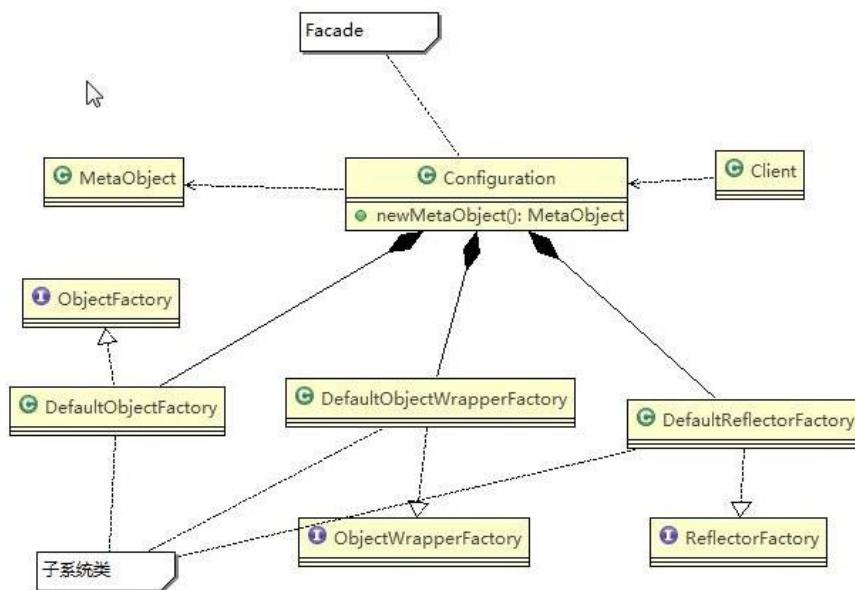
1) MyBatis 中的 Configuration 去创建 MetaObject 对象使用到外观模式

2) 代码分析+Debug 源码+示意图

```
public class Configuration {
    protected ReflectorFactory reflectorFactory = new DefaultReflectorFactory();
    protected ObjectFactory objectFactory = new DefaultObjectFactory();
    protected ObjectWrapperFactory objectWrapperFactory = new DefaultObjectWrapperFactory();
    public MetaObject newMetaObject(Object object) {
        return MetaObject.forObject
            (object, objectFactory, objectWrapperFactory, reflectorFactory);
    }
}
```

```
public class MetaObject{
    private MetaObject(Object object, ObjectFactory objectFactory, ObjectWra
    this.originalObject = object;
    this.objectFactory = objectFactory;
    this.objectWrapperFactory = objectWrapperFactory;
    this.reflectorFactory = reflectorFactory;
    if (object instanceof ObjectWrapper) {
        this.objectWrapper = (ObjectWrapper) object;
    } else if (objectWrapperFactory.hasWrapperFor(object)) {
        this.objectWrapper = objectWrapperFactory.getWrapperFor(this, object);
    } else if (object instanceof Map) {
        this.objectWrapper = new MapWrapper(this, (Map) object);
    } else if (object instanceof Collection) {
        this.objectWrapper = new CollectionWrapper(this, (Collection) object);
    } else {
        this.objectWrapper = new BeanWrapper(this, object);
    }
}
public static MetaObject forObject(Object object, ObjectFactory objectFac
if (object == null) {
    return SystemMetaObject.NULL_META_OBJECT;
}
```

3) 对源码中使用到的外观模式的角色类图



13.8 外观模式的注意事项和细节

- 1) 外观模式对外屏蔽了子系统的细节，因此外观模式降低了客户端对子系统使用的复杂性
- 2) 外观模式对客户端与子系统的耦合关系 - 解耦，让子系统内部的模块更易维护和扩展
- 3) 通过合理的使用外观模式，可以帮我们更好的划分访问的层次
- 4) 当系统需要进行分层设计时，可以考虑使用 Facade 模式
- 5) 在维护一个遗留的大型系统时，可能这个系统已经变得非常难以维护和扩展，此时可以考虑为新系统开发一个 Facade 类，来提供遗留系统的比较清晰简单的接口，让新系统与 Facade 类交互，提高复用性
- 6) 不能过多的或者不合理的使用外观模式，使用外观模式好，还是直接调用模块好。要以让系统有层次，利于维护为目的。

第 14 章 享元模式

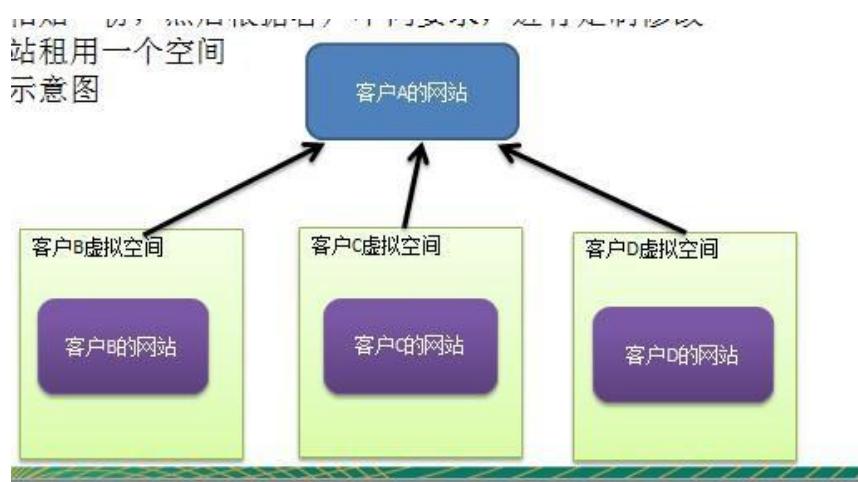
14.1 展示网站项目需求

小型的外包项目，给客户 A 做一个产品展示网站，客户 A 的朋友感觉效果不错，也希望做这样的产品展示网站，但是要求都有些不同：

- 1) 有客户要求以新闻的形式发布
- 2) 有客户要求以博客的形式发布
- 3) 有客户希望以微信公众号的形式发布

14.2 传统方案解决网站展现项目

- 1) 直接复制粘贴一份，然后根据客户不同要求，进行定制修改
- 2) 给每个网站租用一个空间
- 3) 方案设计示意图



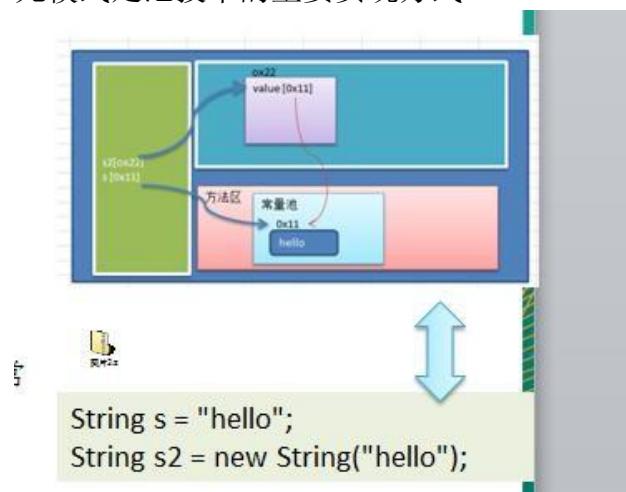
14.3 传统方案解决网站展现项目-问题分析

- 1) 需要的网站结构相似度很高，而且都不是高访问量网站，如果分成多个虚拟空间来处理，相当于一个相同网站的实例对象很多，造成服务器的资源浪费
- 2) 解决思路：整合到一个网站中，共享其相关的代码和数据，对于硬盘、内存、CPU、数据库空间等服务器资源都可以达成共享，减少服务器资源
- 3) 对于代码来说，由于是一份实例，维护和扩展都更加容易
- 4) 上面的解决思路就可以使用 享元模式 来解决

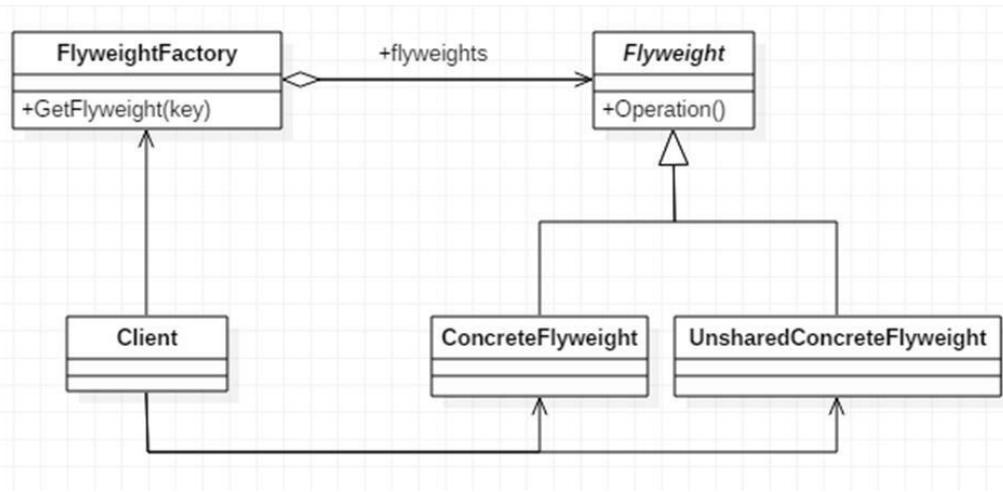
14.4 享元模式基本介绍

基本介绍

- 1) 享元模式（Flyweight Pattern）也叫 蝇量模式：运用共享技术有效地支持大量细粒度的对象
- 2) 常用于系统底层开发，解决系统的性能问题。像数据库连接池，里面都是创建好的连接对象，在这些连接对象中有我们需要的则直接拿来用，避免重新创建，如果没有我们需要的，则创建一个
- 3) 享元模式能够解决重复对象的内存浪费的问题，**当系统中有大量相似对象，需要缓冲池时。不需总是创建新对象，可以从缓冲池里拿。**这样可以降低系统内存，同时提高效率
- 4) 享元模式经典的应用场景就是**池技术**了，String 常量池、数据库连接池、缓冲池等等都是享元模式的应用，享元模式是池技术的重要实现方式



14.5 享元模式的原理类图



➤ 对类图的说明

对原理图的说明-即(模式的角色及职责)

- 1) FlyWeight 是抽象的享元角色, 他是产品的抽象类, 同时定义出对象的外部状态和内部状态(后面介绍) 的接口或实现
- 2) ConcreteFlyWeight 是具体的享元角色, 是具体的产品类, 实现抽象角色定义相关业务
- 3) UnSharedConcreteFlyWeight 是不可共享的角色, 一般不会出现在享元工厂。
- 4) FlyWeightFactory 享元工厂类, 用于构建一个池容器(集合), 同时提供从池中获取对象方法

14.6 内部状态和外部状态

比如围棋、五子棋、跳棋, 它们都有大量的棋子对象, 围棋和五子棋只有黑白两色, 跳棋颜色多一点, 所以棋子颜色就是棋子的内部状态; 而各个棋子之间的差别就是位置的不同, 当我们落子后, 落子颜色是定的, 但位置是变化的, 所以棋子坐标就是棋子的外部状态

- 1) 享元模式提出了两个要求: 细粒度和共享对象。这里就涉及到内部状态和外部状态了, 即将对象的信息分为两个部分: 内部状态和外部状态
- 2) 内部状态指对象共享出来的信息, 存储在享元对象内部且不会随环境的改变而改变
- 3) 外部状态指对象得以依赖的一个标记, 是随环境改变而改变的、不可共享的状态。

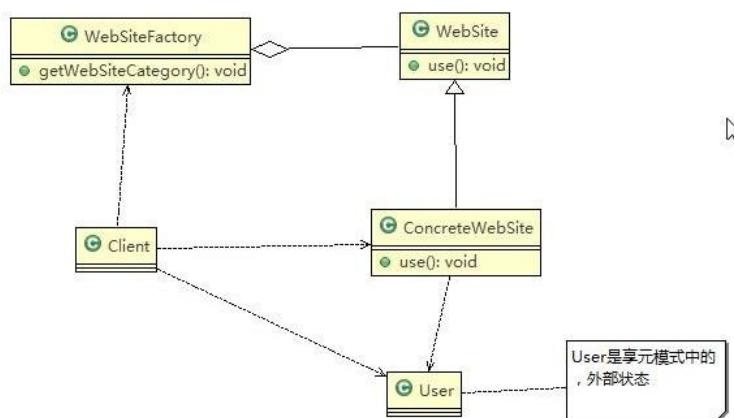
- 4) 举个例子：围棋理论上有 361 个空位可以放棋子，每盘棋都有可能有两三百个棋子对象产生，因为内存空间有限，一台服务器很难支持更多的玩家玩围棋游戏，如果用享元模式来处理棋子，那么棋子对象就可以减少到只有两个实例，这样就很好的解决了对象的开销问题

14.7 享元模式解决网站展现项目

1) 应用实例要求

使用享元模式完成，前面提出的网站外包问题

2) 思路分析和图解(类图)



3) 代码实现



`flyweight.zip`

```

package com.atguigu.flyweight;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
  
```



```
// 创建一个工厂类
WebSiteFactory factory = new WebSiteFactory();

// 客户要一个以新闻形式发布的网站
WebSite webSite1 = factory.getWebSiteCategory("新闻");

webSite1.use(new User("tom"));

// 客户要一个以博客形式发布的网站
WebSite webSite2 = factory.getWebSiteCategory("博客");

webSite2.use(new User("jack"));

// 客户要一个以博客形式发布的网站
WebSite webSite3 = factory.getWebSiteCategory("博客");

webSite3.use(new User("smith"));

// 客户要一个以博客形式发布的网站
WebSite webSite4 = factory.getWebSiteCategory("博客");

webSite4.use(new User("king"));

System.out.println("网站的分类共=" + factory.getWebSiteCount());
```



```
}
```

```
}
```

```
package com.atguigu.flyweight;
```

```
//具体网站
```

```
public class ConcreteWebSite extends WebSite {
```

```
    //共享的部分， 内部状态
```

```
    private String type = "";//网站发布的形式(类型)
```

```
    //构造器
```

```
    public ConcreteWebSite(String type) {
```

```
        this.type = type;
```

```
}
```

```
@Override
```

```
public void use(User user) {
```

```
    // TODO Auto-generated method stub
```

```
    System.out.println("网站的发布形式为:" + type + " 在使用中 .. 使用者是" + user.getName());
```

```
}
```



```
}
```

```
package com.atguigu.flyweight;
```

```
public class User {
```

```
    private String name;
```

```
    public User(String name)
```

```
        { super();
```

```
        this.name = name;
```

```
    }
```

```
    public String getName()
```

```
        { return name;
```

```
    }
```

```
    public void setName(String name)
```

```
        { this.name = name;
```

```
    }
```

```
}
```



```
package com.atguigu.flyweight;

public abstract class WebSite {

    public abstract void use(User user); //抽象方法
}
```

```
package com.atguigu.flyweight;

import java.util.HashMap;

// 网站工厂类，根据需要返回压一个网站
public class WebSiteFactory {

    //集合， 充当池的作用
    private HashMap<String, ConcreteWebSite> pool = new HashMap<>();

    //根据网站的类型， 返回一个网站，如果没有就创建一个网站，并放入到池中，并返回
    public WebSite getWebSiteCategory(String type)

        { if(!pool.containsKey(type)) {

            //就创建一个网站，并放入到池中
            pool.put(type, new ConcreteWebSite(type));

        }

        return (WebSite)pool.get(type);
}
```

```
}
```

```
//获取网站分类的总数(池中有多少个网站类型)
```

```
public int getWebSiteCount() {  
    return pool.size();  
}  
}
```

14.8 享元模式在 JDK-Integer 的应用源码分析

- 1) Integer 中的享元模式
- 2) 代码分析+Debug 源码+说明

```
public class FlyWeight {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Integer x = Integer.valueOf(127);  
        Integer y = new Integer(127);  
        Integer z = Integer.valueOf(127);  
        Integer w = new Integer(127);  
        System.out.println(x.equals(y));//?  
        System.out.println(x == y);//?  
        System.out.println(x == z);//?  
        System.out.println(w == x);//?  
        System.out.println(w == y);//?  
    }  
}
```

```
public final class Integer extends Number implements Comparable<Integer> {  
    public static Integer valueOf(int i) {  
        if (i >= IntegerCache.low && i <= IntegerCache.high)  
            return IntegerCache.cache[i + (-IntegerCache.low)];  
        return new Integer(i);  
    }  
    public Integer(int value) {  
        this.value = value;  
    }  
}
```

➤ 代码说明：

```
package com.atguigu.jdk;  
  
public class FlyWeight {
```



```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
  
    //如果 Integer.valueOf(x) x 在 -128 --- 127 直接，就是使用享元模式返回,如果不在  
    //范围类，则仍然 new  
  
    //小结：  
    //1. 在 valueOf 方法中，先判断值是否在 IntegerCache 中，如果不在，就创建新的 Integer(new)，否则，就直  
接从 缓存池返回  
    //2. valueOf 方法，就使用到享元模式  
    //3. 如果使用 valueOf 方法得到一个 Integer 实例，范围在 -128 - 127 ，执行速度比 new 快  
  
  
  
Integer x = Integer.valueOf(127); // 得到 x 实例，类型 Integer  
Integer y = new Integer(127); // 得到 y 实例，类型 Integer  
Integer z = Integer.valueOf(127);..  
Integer w = new Integer(127);  
  
System.out.println(x.equals(y)); // 大小， true  
System.out.println(x == y ); // false  
System.out.println(x == z ); // true  
System.out.println(w == x ); // false  
System.out.println(w == y ); // false  
  
  
  
Integer x1 = Integer.valueOf(200);  
Integer x2 = Integer.valueOf(200);
```



```
System.out.println("x1==x2" + (x1 == x2)); // false  
}  
}
```

14.9 享元模式的注意事项和细节

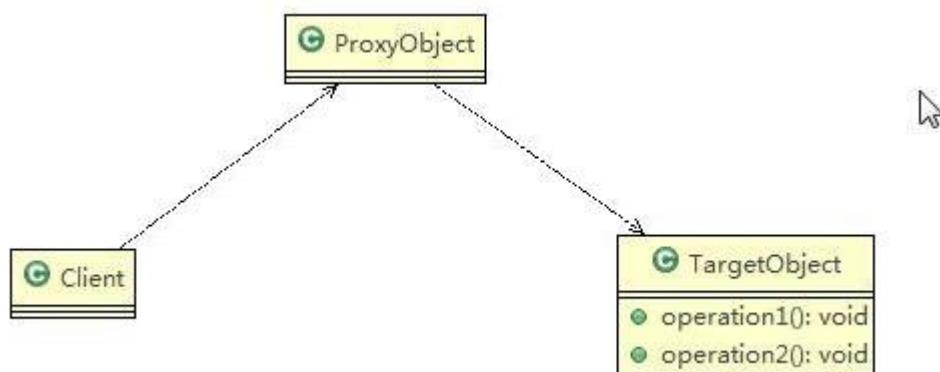
- 1) 在享元模式这样理解，“享”就表示共享，“元”表示对象
- 2) 系统中有大量对象，这些对象消耗大量内存，并且对象的状态大部分可以外部化时，我们就可以考虑选用享元模式
- 3) 用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象，用 `HashMap`/`HashTable` 存储
- 4) 享元模式大大减少了对象的创建，降低了程序内存的占用，提高效率
- 5) 享元模式提高了系统的复杂度。需要分离出内部状态和外部状态，而外部状态具有固化特性，不应该随着内部状态的改变而改变，这是我们使用享元模式需要注意的地方。
- 6) 使用享元模式时，注意划分内部状态和外部状态，并且需要有一个工厂类加以控制。
- 7) 享元模式经典的应用场景是需要缓冲池的场景，比如 `String` 常量池、数据库连接池

第 15 章 代理模式

15.1 代理模式(Proxy)

15.1.1 代理模式的基本介绍

- 1) 代理模式：为一个对象提供一个替身，以控制对这个对象的访问。即通过代理对象访问目标对象。这样做的好处是：可以在目标对象实现的基础上，增强额外的功能操作，即扩展目标对象的功能。
- 2) 被代理的对象可以是远程对象、创建开销大的对象或需要安全控制的对象
- 3) 代理模式有不同的形式，主要有三种 静态代理、动态代理 (JDK 代理、接口代理) 和 Cglib 代理 (可以在内存动态的创建对象，而不需要实现接口，他是属于动态代理的范畴)。
- 4) 代理模式示意图



15.2 静态代理

15.2.1 静态代码模式的基本介绍

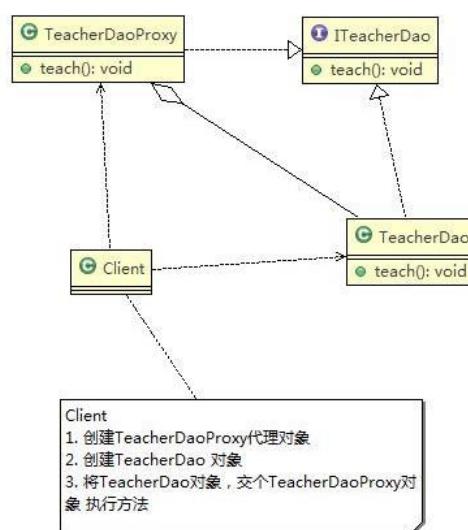
静态代理在使用时，需要定义接口或者父类，被代理对象（即目标对象）与代理对象一起实现相同的接口或者是继承相同父类。

15.2.2 应用实例

➤ 具体要求

- 1) 定义一个接口:ITeacherDao
- 2) 目标对象 TeacherDAO 实现接口 ITeacherDAO
- 3) 使用静态代理方式,就需要在代理对象 TeacherDAOProxy 中也实现 ITeacherDAO
- 4) 调用的时候通过调用代理对象的方法来调用目标对象.
- 5) **特别提醒:** 代理对象与目标对象要实现相同的接口,然后通过调用相同的方法来调用目标对象的方法

➤ 思路分析图解(类图)



➤ 代码实现



```
package com.atguigu.proxy.staticproxy;

public class Client {
```



```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    //创建目标对象(被代理对象)  
    TeacherDao teacherDao = new TeacherDao();  
  
    //创建代理对象, 同时将被代理对象传递给代理对象  
    TeacherDaoProxy teacherDaoProxy = new TeacherDaoProxy(teacherDao);  
  
    //通过代理对象, 调用到被代理对象的方法  
    //即: 执行的是代理对象的方法, 代理对象再去调用目标对象的方法  
    teacherDaoProxy.teach();  
}  
}
```

```
package com.atguigu.proxy.staticproxy;
```

```
//接口  
public interface ITeacherDao {  
    void teach(); // 授课的方法  
}
```

```
package com.atguigu.proxy.staticproxy;
```

```
public class TeacherDao implements ITeacherDao {
```



```
@Override  
public void teach() {  
    // TODO Auto-generated method stub  
    System.out.println(" 老师授课中    ..... ");  
}  
  
}
```

```
package com.atguigu.proxy.staticproxy;  
  
//代理对象,静态代理  
public class TeacherDaoProxy implements ITeacherDao{  
    private ITeacherDao target; // 目标对象, 通过接口来聚合  
  
    //构造器  
    public TeacherDaoProxy(ITeacherDao target)  
    { this.target = target;  
    }  
  
    @Override  
    public void teach() {  
        // TODO Auto-generated method stub  
        System.out.println("开始代理 完成某些操作。.....");//方法  
        target.teach();  
  
        System.out.println("提交。.....");//方法
```



```
}
```

15.2.3 静态代理优缺点

- 1) 优点: 在不修改目标对象的功能前提下, 能通过代理对象对目标功能扩展
- 2) 缺点: 因为代理对象需要与目标对象实现一样的接口, 所以会有很多代理类
- 3) 一旦接口增加方法, 目标对象与代理对象都要维护

15.3 动态代理

15.3.1 动态代理模式的基本介绍

- 1) 代理对象, 不需要实现接口, 但是目标对象要实现接口, 否则不能用动态代理
- 2) 代理对象的生成, 是利用 JDK 的 API, 动态的在内存中构建代理对象
- 3) 动态代理也叫做: JDK 代理、接口代理

15.3.2 JDK 中生成代理对象的 API

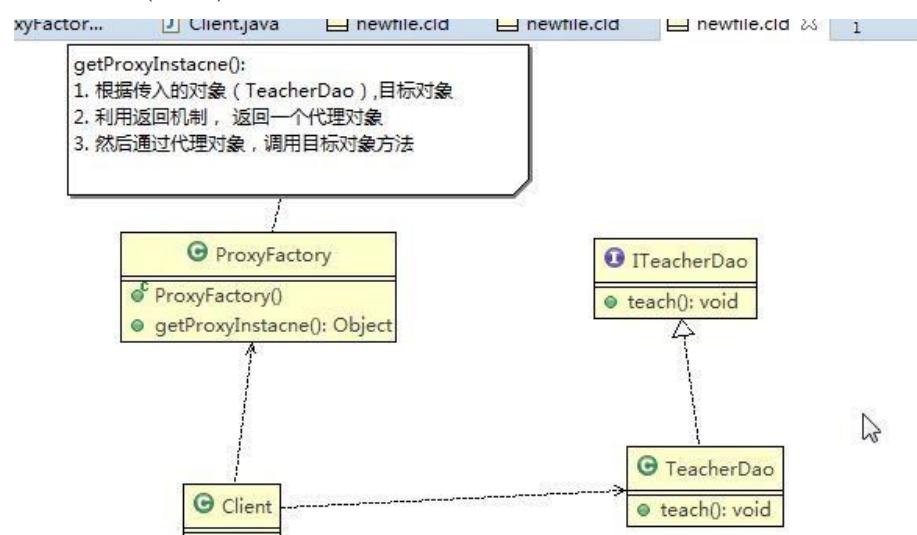
- 1) 代理类所在包: `java.lang.reflect.Proxy`
- 2) JDK 实现代理只需要使用 **newProxyInstance** 方法, 但是该方法需要接收三个参数, 完整的写法是:
`static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)`

15.3.3 动态代理应用实例

➤ 应用实例要求

将前面的静态代理改进成动态代理模式(即：JDK 代理模式)

➤ 思路图解(类图)



➤ 代码实现



```
dynamic.zip
package com.atguigu.proxy.dynamic;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //创建目标对象
        ITeacherDao target = new TeacherDao();

        //给目标对象， 创建代理对象，可以转成 ITeacherDao
    }
}
```



```
ITeacherDao proxyInstance = (ITeacherDao) new ProxyFactory(target).getProxyInstance();  
  
// proxyInstance=class com.sun.proxy.$Proxy0 内存中动态生成了代理对象  
System.out.println("proxyInstance=" + proxyInstance.getClass());  
  
//通过代理对象，调用目标对象的方法  
//proxyInstance.teach();  
  
proxyInstance.sayHello(" tom ");  
}  
}
```

```
package com.atguigu.proxy.dynamic;  
  
//接口  
public interface ITeacherDao {  
  
    void teach(); // 授课方法  
    void sayHello(String name);  
}
```

```
package com.atguigu.proxy.dynamic;  
  
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;
```



```
import java.lang.reflect.Proxy;
```

```
public class ProxyFactory {
```

```
    //维护一个目标对象，Object
```

```
    private Object target;
```

```
    //构造器，对 target 进行初始化
```

```
    public ProxyFactory(Object target) {
```

```
        this.target = target;
```

```
}
```

```
    //给目标对象 生成一个代理对象
```

```
    public Object getProxyInstance() {
```

```
        //说明
```

```
        /*
```

```
         * public static Object newProxyInstance(ClassLoader loader,
```

```
                                         Class<?>[] interfaces,
```

```
                                         InvocationHandler h)
```

//1. ClassLoader loader : 指定当前目标对象使用的类加载器, 获取加载器的方法固定

//2. Class<?>[] interfaces: 目标对象实现的接口类型, 使用泛型方法确认类型

//3. InvocationHandler h : 事情处理, 执行目标对象的方法时, 会触发事情处理器方法, 会把当前执行的目标对象方法作为参数传入



```
*/  
  
return Proxy.newProxyInstance(target.getClass().getClassLoader(),  
    target.getClass().getInterfaces(),  
    new InvocationHandler() {  
        @Override  
  
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
            // TODO Auto-generated method stub  
  
            System.out.println("JDK 代理开始~~");  
            //反射机制调用目标对象的方法  
            Object returnVal = method.invoke(target, args);  
            System.out.println("JDK 代理提交");  
            return returnVal;  
        }  
    }  
);  
  
}  
}
```

```
package com.atguigu.proxy.dynamic;  
  
public class TeacherDao implements ITeacherDao {  
  
    @Override  
    public void teach() {
```



```
// TODO Auto-generated method stub
System.out.println(" 老师授课中.... ");
}

@Override
public void sayHello(String name) {
    // TODO Auto-generated method stub
    System.out.println("hello " + name);
}

}
```

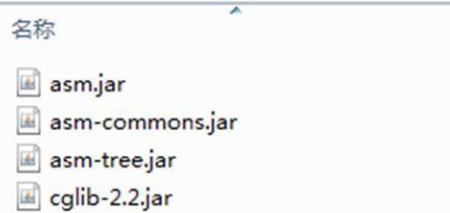
15.4 Cglib 代理

15.4.1 Cglib 代理模式的基本介绍

- 1) 静态代理和 JDK 代理模式都要求目标对象是实现一个接口,但是有时候目标对象只是一个单独的对象,并没有实现任何的接口,这个时候可使用目标对象子类来实现代理.这就是 **Cglib** 代理
- 2) Cglib 代理也叫作子类代理,它是在内存中构建一个子类对象从而实现对目标对象功能扩展,有些书也将Cglib 代理归属到动态代理。
- 3) Cglib 是一个强大的高性能的代码生成包,它可以在运行期扩展 java 类与实现 java 接口.它广泛的被许多 **AOP** 的框架使用,例如 Spring AOP, 实现方法拦截
- 4) 在 AOP 编程中如何选择代理模式:
 1. 目标对象需要实现接口, 用 JDK 代理
 2. 目标对象不需要实现接口, 用 Cglib 代理
- 5) Cglib 包的底层是通过使用字节码处理框架 ASM 来转换字节码并生成新的类

15.4.2 Cglib 代理模式实现步骤

- 1) 需要引入 cglib 的 jar 文件



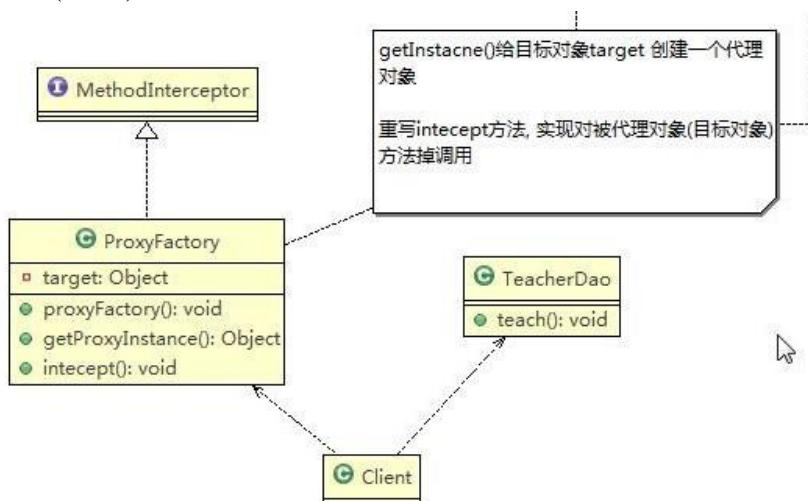
- 2) 在内存中动态构建子类, 注意代理的类不能为 final, 否则报错
java.lang.IllegalArgumentException:
- 3) 目标对象的方法如果为 final/static, 那么就不会被拦截, 即不会执行目标对象额外的业务方法.

15.4.3 Cglib 代理模式应用实例

➤ 应用实例要求

将前面的案例用 Cglib 代理模式实现

➤ 思路图解(类图)



➤ 代码实现+Debug 源码[待 debug]



cglb.zip

```
package com.atguigu.proxy.cglib;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //创建目标对象
        TeacherDao target = new TeacherDao();
        //获取到代理对象，并且将目标对象传递给代理对象
        TeacherDao proxyInstance = (TeacherDao)new ProxyFactory(target).getProxyInstance();

        //执行代理对象的方法，触发 intercept 方法，从而实现 对目标对象的调用
        String res = proxyInstance.teach();
        System.out.println("res=" + res);
    }
}
```

```
package com.atguigu.proxy.cglib;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
```



```
public class ProxyFactory implements MethodInterceptor {  
  
    //维护一个目标对象  
    private Object target;  
  
    //构造器，传入一个被代理的对象  
    public ProxyFactory(Object target) {  
        this.target = target;  
    }  
  
    //返回一个代理对象：是 target 对象的代理对象  
    public Object getProxyInstance() { //四部曲  
        //1. 创建一个工具类  
        Enhancer enhancer = new Enhancer();  
        //2. 设置父类  
        enhancer.setSuperclass(target.getClass());  
        //3. 设置回调函数  
        enhancer.setCallback(this);  
        //4. 创建子类对象，即代理对象  
        return enhancer.create();  
    }  
  
    //重写 intercept 方法，会调用目标对象的方法
```



```
@Override  
public Object intercept(Object arg0, Method method, Object[] args, MethodProxy arg3) throws Throwable {  
    // TODO Auto-generated method stub  
    System.out.println("Cglib 代理模式 ~~ 开始");  
    Object returnVal = method.invoke(target, args);  
    System.out.println("Cglib 代理模式 ~~ 提交");  
    return returnVal;  
}  
}
```

```
package com.atguigu.proxy.cglib;  
  
public class TeacherDao {  
  
    public String teach() {  
        System.out.println(" 老师授课中 , 我是 cglib 代理, 不需要实现接口 ");  
        return "hello";  
    }  
}
```

15.5 几种常见的代理模式介绍— 几种变体

1) 防火墙代理

内网通过代理穿透防火墙，实现对公网的访问。

2) 缓存代理

比如：当请求图片文件等资源时，先到缓存代理取，如果取到资源则 ok, 如果取不到资源，再到公网或者数据库取，然后缓存。



3) 远程代理

远程对象的本地代表，通过它可以把远程对象当本地对象来调用。远程代理通过网络和真正的远程对象沟通信息。

4) 同步代理：主要使用在多线程编程中，完成多线程间同步工作

同步代理：主要使用在多线程编程中，完成多线程间同步工作

第 16 章 模板方法模式

16.1 豆浆制作问题

编写制作豆浆的程序，说明如下：

- 1) 制作豆浆的流程 选材--->添加配料--->浸泡--->放到豆浆机打碎
- 2) 通过添加不同的配料，可以制作出不同口味的豆浆
- 3) 选材、浸泡和放到豆浆机打碎这几个步骤对于制作每种口味的豆浆都是一样的
- 4) 请使用 模板方法模式 完成 (说明：因为模板方法模式，比较简单，很容易就想到这个方案，因此就直接使用，不再使用传统的方案来引出模板方法模式)

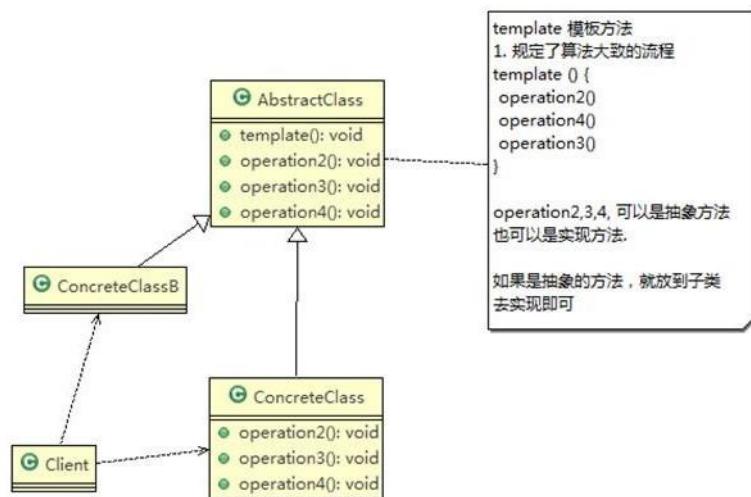
16.2 模板方法模式基本介绍

基本介绍

- 1) 模板方法模式 (Template Method Pattern)，又叫模板模式(Template Pattern)，z 在一个抽象类公开定义了执行它的方法的模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。
- 2) 简单说，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，使得子类可以不改变一个算法的结构，就可以重定义该算法的某些特定步骤
- 3) 这种类型的设计模式属于行为型模式。

16.3 模板方法模式原理类图

16.3.1 模板方法模式的原理类图



template 模板方法
1. 规定了算法大致的流程
template () {
 operation2()
 operation4()
 operation3()
}

operation2,3,4, 可以是抽象方法
也可以是实现方法。

如果是抽象的方法，就放到子类
去实现即可

- 对原理类图的说明-即(模板方法模式的角色及职责)
- 1) `AbstractClass` 抽象类, 类中实现了模板方法(template), 定义了算法的骨架, 具体子类需要去实现其它的抽象方法 `operationr2,3,4`
 - 2) `ConcreteClass` 实现抽象方法 `operationr2,3,4`, 以完成算法中特点子类的步骤

16.4 模板方法模式解决豆浆制作问题

1) 应用实例要求

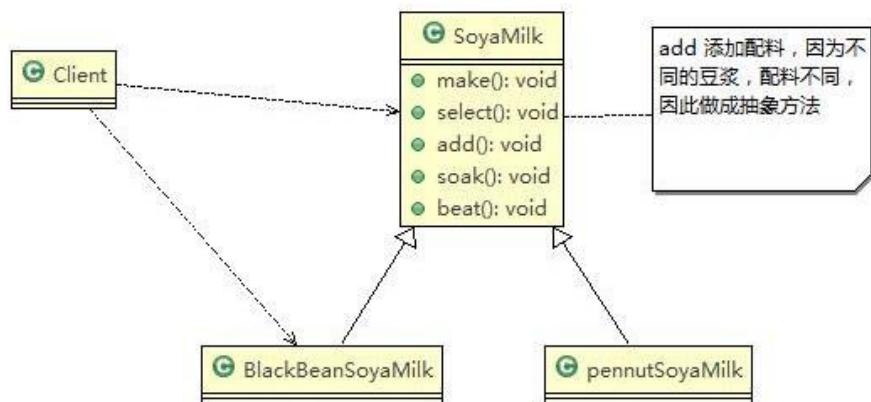
编写制作豆浆的程序, 说明如下:

制作豆浆的流程 选材--->添加配料--->浸泡--->放到豆浆机打碎

通过添加不同的配料, 可以制作出不同口味的豆浆

选材、浸泡和放到豆浆机打碎这几个步骤对于制作每种口味的豆浆都是一样的(红豆、花生豆浆。。.)

2) 思路分析和图解(类图)



3) 代码实现



```
package com.atguigu.template;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //制作红豆豆浆

        System.out.println("----制作红豆豆浆----");
        SoyaMilk redBeanSoyaMilk = new RedBeanSoyaMilk();
        redBeanSoyaMilk.make();

        System.out.println("---- 制作花生豆浆 ----");
        SoyaMilk peanutSoyaMilk = new PeanutSoyaMilk();
        peanutSoyaMilk.make();
    }
}
```



```
}
```

```
}
```

```
package com.atguigu.template;

public class PeanutSoyaMilk extends SoyaMilk {

    @Override
    void addCondiments() {
        // TODO Auto-generated method stub
        System.out.println("加入上好的花生");
    }
}
```

```
package com.atguigu.template;

public class RedBeanSoyaMilk extends SoyaMilk {

    @Override
    void addCondiments() {
        // TODO Auto-generated method stub
        System.out.println("加入上好的红豆");
    }
}
```



```
}
```

```
package com.atguigu.template;

//抽象类，表示豆浆
public abstract class SoyaMilk {

    //模板方法, make , 模板方法可以做成 final , 不让子类去覆盖.
    final void make() {

        select();
        addCondiments();
        soak();
        beat();

    }

    //选材料
    void select() {
        System.out.println("第一步：选择好的新鲜黄豆  ");
    }

    //添加不同的配料， 抽象方法，子类具体实现
    abstract void addCondiments();

    //浸泡
}
```

```
void soak() {  
    System.out.println("第三步， 黄豆和配料开始浸泡， 需要 3 小时");  
}  
  
void beat() {  
    System.out.println("第四步： 黄豆和配料放到豆浆机去打碎 ");  
}  
}
```

16.5 模板方法模式的钩子方法

- 1) 在模板方法模式的父类中，我们可以定义一个方法，它默认不做任何事，子类可以视情况要不要覆盖它，该方法称为“钩子”。
- 2) 还是用上面做豆浆的例子来讲解，比如，我们还希望制作纯豆浆，不添加任何的配料，请使用钩子方法对前面的模板方法进行改造
- 3) 看老师代码演示：



```
package com.atguigu.template.improve;  
  
//抽象类，表示豆浆  
public abstract class SoyaMilk {  
  
    //模板方法, make , 模板方法可以做成 final , 不让子类去覆盖.  
    final void make() {
```



```
select();
if(customerWantCondiments()) {
    addCondiments();
}
soak();
beat();
}
//选材料
void select() {
    System.out.println("第一步：选择好的新鲜黄豆  ");
}
//添加不同的配料， 抽象方法，子类具体实现
abstract void addCondiments();

//浸泡
void soak() {
    System.out.println("第三步， 黄豆和配料开始浸泡， 需要 3 小时 ");
}

void beat() {
    System.out.println("第四步： 黄豆和配料放到豆浆机去打碎  ");
}

//钩子方法， 决定是否需要添加配料
boolean customerWantCondiments() {
    return true;
}
```

```

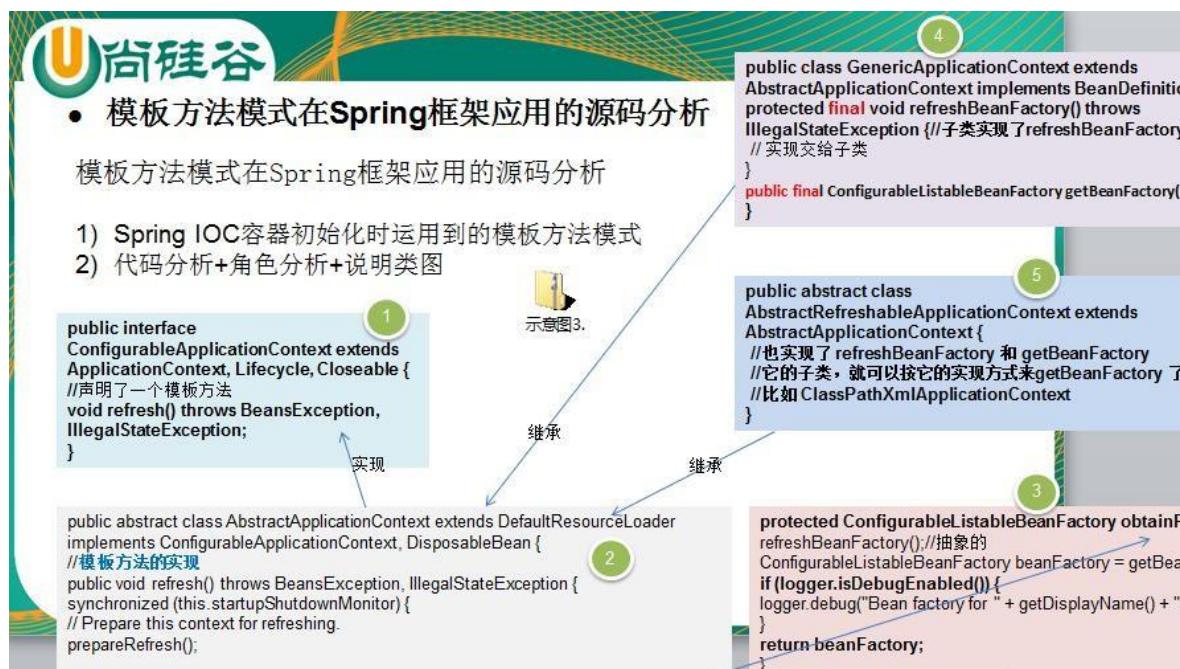
    }
}

}

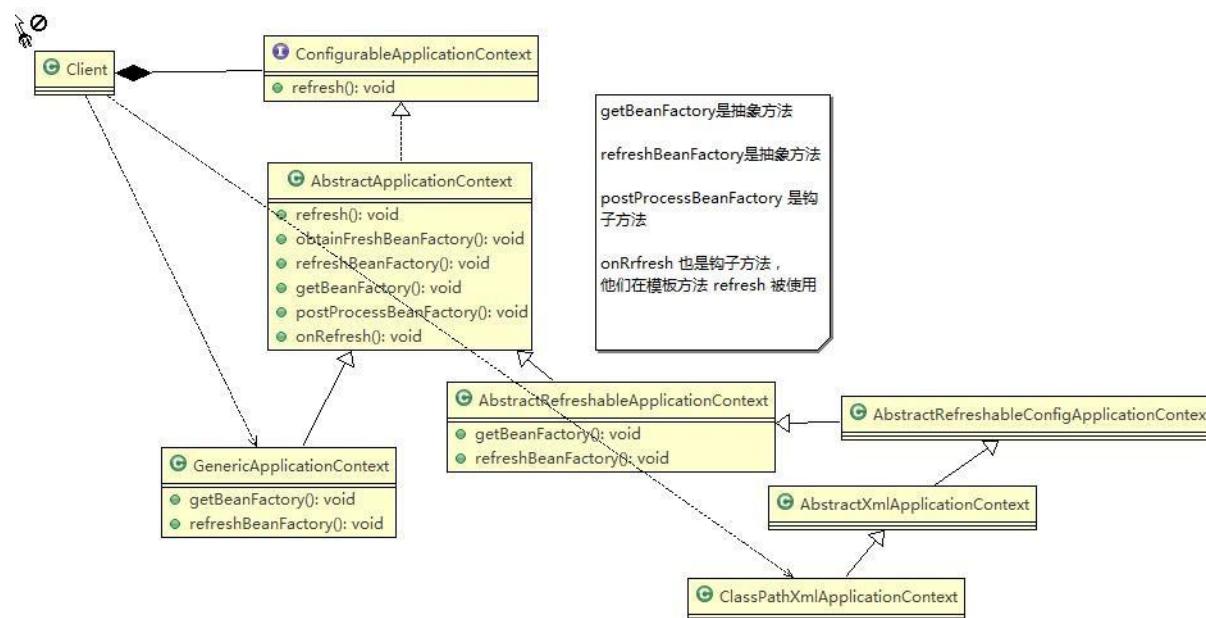
```

16.6 模板方法模式在 Spring 框架应用的源码分析

- 1) Spring IOC 容器初始化时运用到的模板方法模式
- 2) 代码分析+角色分析+说明类图



- 3) 针对源码的类图(说明层次关系)



16.7 模板方法模式的注意事项和细节

- 1) 基本思想是：算法只存在于一个地方，也就是在父类中，容易修改。需要修改算法时，只要修改父类的模板方法或者已经实现的某些步骤，子类就会继承这些修改
- 2) 实现了最大化代码复用。父类的模板方法和已实现的某些步骤会被子类继承而直接使用。
- 3) 既统一了算法，也提供了很大的灵活性。父类的模板方法确保了算法的结构保持不变，同时由子类提供部分步骤的实现。
- 4) 该模式的不足之处：每一个不同的实现都需要一个子类实现，导致类的个数增加，使得系统更加庞大
- 5) 一般模板方法都加上 final 关键字，防止子类重写模板方法。
- 6) 模板方法模式使用场景：当要完成在某个过程，该过程要执行一系列步骤，这一系列的步骤基本相同，但其个别步骤在实现时可能不同，通常考虑用模板方法模式来处理

第 17 章 命令模式

17.1 智能生活项目需求

看一个具体的需求



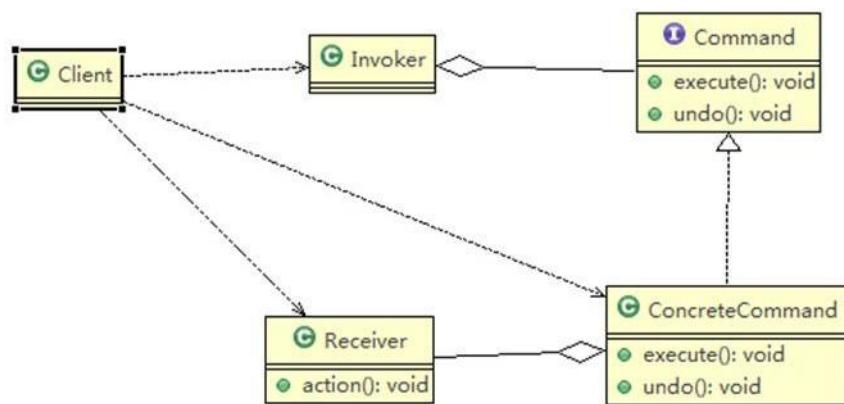
- 1) 我们买了一套智能家电，有照明灯、风扇、冰箱、洗衣机，我们只要在手机上安装 app 就可以控制对这些家电工作。
- 2) 这些智能家电来自不同的厂家，我们不想针对每一种家电都安装一个 App，分别控制，我们希望只要一个 app 就可以控制全部智能家电。
- 3) 要实现一个 app 控制所有智能家电的需要，则每个智能家电厂家都要提供一个统一的接口给 app 调用，这时就可以考虑使用命令模式。
- 4) 命令模式可将“动作的请求者”从“动作的执行者”对象中解耦出来。
- 5) 在我们的例子中，动作的请求者是手机 app，动作的执行者是每个厂商的一个家电产品

17.2 命令模式基本介绍

- 1) 命令模式（Command Pattern）：在软件设计中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，
我们只需在程序运行时指定具体的请求接收者即可，此时，可以使用命令模式来进行设计
- 2) 命令模式使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活，实现解耦。
- 3) 在命名模式中，会将一个请求封装为一个对象，以便使用不同参数来表示不同的请求(即命名)，同时命令模式也支持可撤销的操作。

- 4) 通俗易懂的理解：将军发布命令，士兵去执行。其中有几个角色：将军（命令发布者）、士兵（命令的具体执行者）、命令(连接将军和士兵)。
Invoker 是调用者（将军），Receiver 是被调用者（士兵），MyCommand 是命令，实现了 Command 接口，持有接收对象

17.3 命令模式的原理类图

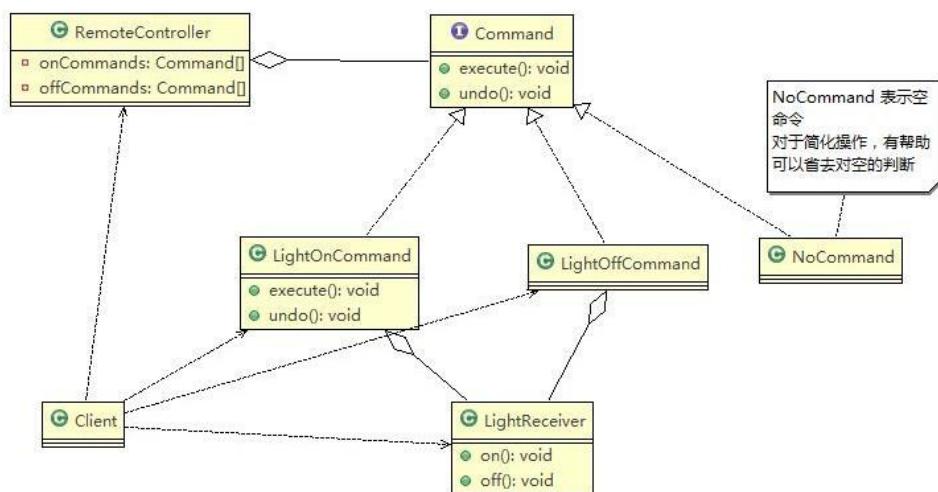


- 对原理类图的说明-即(命名模式的角色及职责)
- 1) Invoker 是调用者角色
 - 2) Command: 是命令角色，需要执行的所有命令都在这里，可以是接口或抽象类
 - 3) Receiver: 接受者角色，知道如何实施和执行一个请求相关的操作
 - 4) ConcreteCommand: 将一个接受者对象与一个动作绑定，调用接受者相应的操作，实现 execute

17.4 命令模式解决智能生活项目

应用实例要求

- 1) 编写程序，使用命令模式 完成前面的智能家电项目
- 2) 思路分析和图解



3) 代码实现



```

package com.atguigu.command;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        // 使用命令设计模式，完成通过遥控器，对电灯的操作

        // 创建电灯的对象(接受者)
        LightReceiver lightReceiver = new LightReceiver();

        // 创建电灯相关的开关命令
        LightOnCommand lightOnCommand = new LightOnCommand(lightReceiver);
    }
}
  
```



```
LightOffCommand lightOffCommand = new LightOffCommand(lightReceiver);
```

```
//需要一个遥控器
```

```
RemoteController remoteController = new RemoteController();
```

```
//给我们的遥控器设置命令, 比如 no = 0 是电灯的开和关的操作
```

```
remoteController.setCommand(0, lightOnCommand, lightOffCommand);
```

```
System.out.println("-----按下灯的开按钮-----");
```

```
remoteController.onButtonWasPushed(0);
```

```
System.out.println("-----按下灯的关按钮-----");
```

```
remoteController.offButtonWasPushed(0);
```

```
System.out.println("-----按下撤销按钮-----");
```

```
remoteController.undoButtonWasPushed();
```

```
System.out.println("=====使用遥控器操作电视机=====");
```

```
TVReceiver tvReceiver = new TVReceiver();
```

```
TVOffCommand tvOffCommand = new TVOffCommand(tvReceiver);
```

```
TVOnCommand tvOnCommand = new TVOnCommand(tvReceiver);
```

```
//给我们的遥控器设置命令, 比如 no = 1 是电视机的开和关的操作
```

```
remoteController.setCommand(1, tvOnCommand, tvOffCommand);
```



```
System.out.println("-----按下电视机的开按钮-----");
remoteController.onButtonWasPushed(1);
System.out.println("-----按下电视机的关按钮-----");
remoteController.offButtonWasPushed(1);
System.out.println("----- 按下撤销按钮 -----");
remoteController.undoButtonWasPushed();

}
```

```
package com.atguigu.command;
```

```
//创建命令接口
public interface Command {

    //执行动作(操作)
    public void execute();
    //撤销动作(操作)
    public void undo();
}
```

```
package com.atguigu.command;
```

```
public class LightOffCommand implements Command {
```



```
// 聚合 LightReceiver  
  
LightReceiver light;  
  
// 构造器  
public LightOffCommand(LightReceiver light)  
{  
    super();  
    this.light = light;  
}  
  
@Override  
public void execute()  
{  
    // TODO Auto-generated method stub  
    // 调用接收者的方法  
    light.off();  
}  
  
@Override  
public void undo()  
{  
    // TODO Auto-generated method stub  
    // 调用接收者的方法  
    light.on();  
}  
}
```



```
package com.atguigu.command;

public class LightOnCommand implements Command {

    //聚合 LightReceiver
    LightReceiver light;

    //构造器
    public LightOnCommand(LightReceiver light)
    {
        super();
        this.light = light;
    }

    @Override
    public void execute() {
        // TODO Auto-generated method stub
        //调用接收者的方法
        light.on();
    }

    @Override
    public void undo() {
        // TODO Auto-generated method stub
    }
}
```



```
//调用接收者的方法  
light.off();  
}
```

```
}
```

```
package com.atguigu.command;
```

```
public class LightReceiver {
```

```
    public void on() {  
        System.out.println(" 电灯打开了.. ");  
    }
```

```
    public void off() {  
        System.out.println(" 电灯关闭了.. ");  
    }
```

```
}
```

```
package com.atguigu.command;
```

```
/**  
 * 没有任何命令，即空执行：用于初始化每个按钮，当调用空命令时，对象什么都不做  
 * 其实，这样是一种设计模式，可以省掉对空判断  
 * @author Administrator  
 */
```



```
*/  
  
public class NoCommand implements Command {  
  
    @Override  
    public void execute() {  
        // TODO Auto-generated method stub  
  
    }  
  
    @Override  
    public void undo() {  
        // TODO Auto-generated method stub  
  
    }  
}
```

```
package com.atguigu.command;  
  
public class RemoteController {  
  
    // 开按钮的命令数组  
    Command[] onCommands;  
    Command[] offCommands;  
  
    // 执行撤销的命令
```



```
Command undoCommand;

// 构造器，完成对按钮初始化

public RemoteController() {

    onCommands = new Command[5];
    offCommands = new Command[5];

    for (int i = 0; i < 5; i++) {
        onCommands[i] = new NoCommand();
        offCommands[i] = new NoCommand();
    }
}

// 给我们的按钮设置你需要的命令
public void setCommand(int no, Command onCommand, Command offCommand)
{
    onCommands[no] = onCommand;
    offCommands[no] = offCommand;
}

// 按下开按钮
public void onButtonWasPushed(int no) { // no 0
    // 找到你按下的开的按钮，并调用对应方法
    onCommands[no].execute();
    // 记录这次的操作，用于撤销
}
```



```
undoCommand = onCommands[no];

}

// 按下开按钮
public void offButtonWasPushed(int no) { // no 0
    // 找到你按下的关的按钮，并调用对应方法
    offCommands[no].execute();
    // 记录这次的操作，用于撤销
    undoCommand = offCommands[no];

}

// 按下撤销按钮
public void undoButtonWasPushed()
{
    undoCommand.undo();
}
```

```
package com.atguigu.command;

public class TVOffCommand implements Command {

    // 聚合 TVReceiver
```



```
TVReceiver tv;

// 构造器
public TVOffCommand(TVReceiver tv)
{
    super();
    this.tv = tv;
}

@Override
public void execute() {
    // TODO Auto-generated method stub
    // 调用接收者的方法
    tv.off();
}

@Override
public void undo() {
    // TODO Auto-generated method stub
    // 调用接收者的方法
    tv.on();
}

package com.atguigu.command;

public class TVOnCommand implements Command {
```



```
// 聚合 TVReceiver  
  
TVReceiver tv;  
  
// 构造器  
public TVOnCommand(TVReceiver tv)  
{ super();  
    this.tv = tv;  
}  
  
@Override  
public void execute()  
{  
    // TODO Auto-generated method stub  
    // 调用接收者的方法  
    tv.on();  
}  
  
@Override  
public void undo()  
{  
    // TODO Auto-generated method stub  
    // 调用接收者的方法  
    tv.off();  
}  
}
```

```

package com.atguigu.command;

public class TVReceiver {

    public void on() {
        System.out.println(" 电视机打开了.. ");
    }

    public void off() {
        System.out.println(" 电视机关闭了.. ");
    }
}

```

17.5 命令模式在 Spring 框架JdbcTemplate 应用的源码分析

1) Spring 框架的 JdbcTemplate 就使用到了命令模式

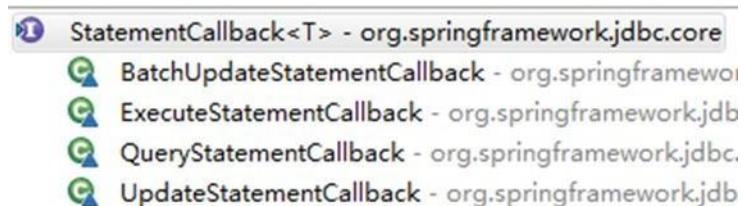
2) 代码分析

- 1) Spring 框架的 JdbcTemplate 就使用到了命令模式
- 2) 代码分析



3) 模式角色分析说明

- StatementCallback 接口，类似命令接口(Command)
- class QueryStatementCallback implements StatementCallback<T>, SqlProvider, 匿名内部类，实现了命令接口，同时也充当命令接收者
- 命令调用者是 JdbcTemplate，其中 execute(StatementCallback<T> action) 方法中，调用 action.doInStatement 方法。不同的实现 StatementCallback 接口的对象，对应不同的 doInStatement 实现逻辑
- 另外实现 StatementCallback 命令接口的子类还有 QueryStatementCallback、



17.6 命令模式的注意事项和细节

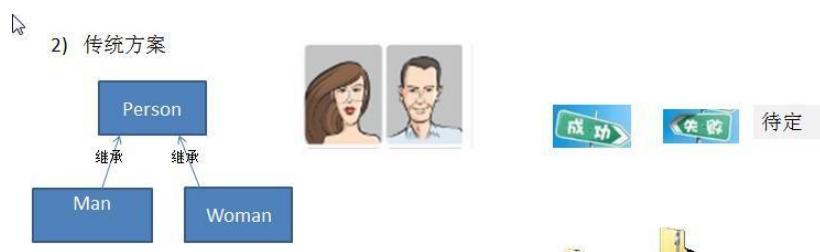
- 1) 将发起请求的对象与执行请求的对象解耦。发起请求的对象是调用者，调用者只要调用命令对象的 execute() 方法就可以让接收者工作，而不必知道具体的接收者对象是谁、是如何实现的，命令对象会负责让接收者执行请求的动作，也就是说：“请求发起者”和“请求执行者”之间的解耦是通过命令对象实现的，命令对象起到了纽带桥梁的作用。
- 2) 容易设计一个命令队列。只要把命令对象放到列队，就可以多线程的执行命令
- 3) 容易实现对请求的撤销和重做
- 4) 命令模式不足：可能导致某些系统有过多的具体命令类，增加了系统的复杂度，这点在在使用的时候要注意
- 5) 空命令也是一种设计模式，它为我们省去了判空的操作。在上面的实例中，如果没有用空命令，我们每按下一个按键都要判空，这给我们编码带来一定的麻烦。
- 6) 命令模式经典的应用场景：界面的一个按钮都是一条命令、模拟 CMD (DOS 命令) 订单的撤销/恢复、触发-反馈机制

第 18 章 访问者模式

18.1 测评系统的需求

完成测评系统需求

- 1) 将观众分为男人和女人，对歌手进行测评，当看完某个歌手表演后，得到他们对该歌手不同的评价(评价有不同的种类，比如成功、失败等)



- 2) 传统方案

18.2 传统方式的问题分析

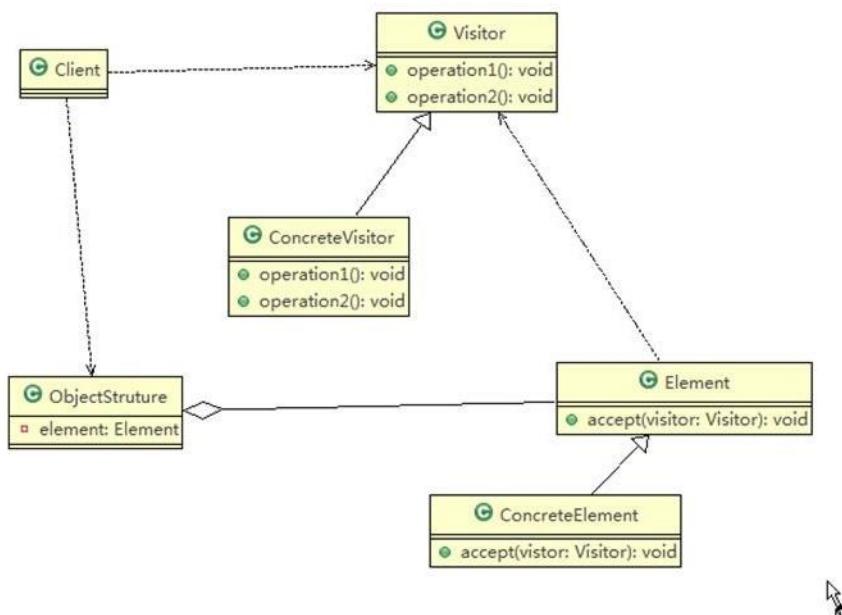
- 1) 如果系统比较小，还是 ok 的，但是考虑系统增加越来越多新的功能时，对代码改动较大，违反了 ocp 原则，不利于维护
- 2) 扩展性不好，比如增加了新的人员类型，或者管理方法，都不好做
- 3) 引出我们会使用新的设计模式 — 访问者模式

18.3 访问者模式基本介绍

- 1) 访问者模式（Visitor Pattern），封装一些作用于某种数据结构的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新操作。
- 2) 主要将数据结构与数据操作分离，解决 数据结构和操作耦合性问题
- 3) 访问者模式的基本工作原理是：在被访问的类里面加一个对外提供接待访问者的接口
- 4) 访问者模式主要应用场景是：需要对一个对象结构中的对象进行很多不同操作(这些操作彼此没有关联)，同时

需要避免让这些操作“污染”这些对象的类，可以选用访问者模式解决

18.4 访问者模式的原理类图



➤ 对原理类图的说明-

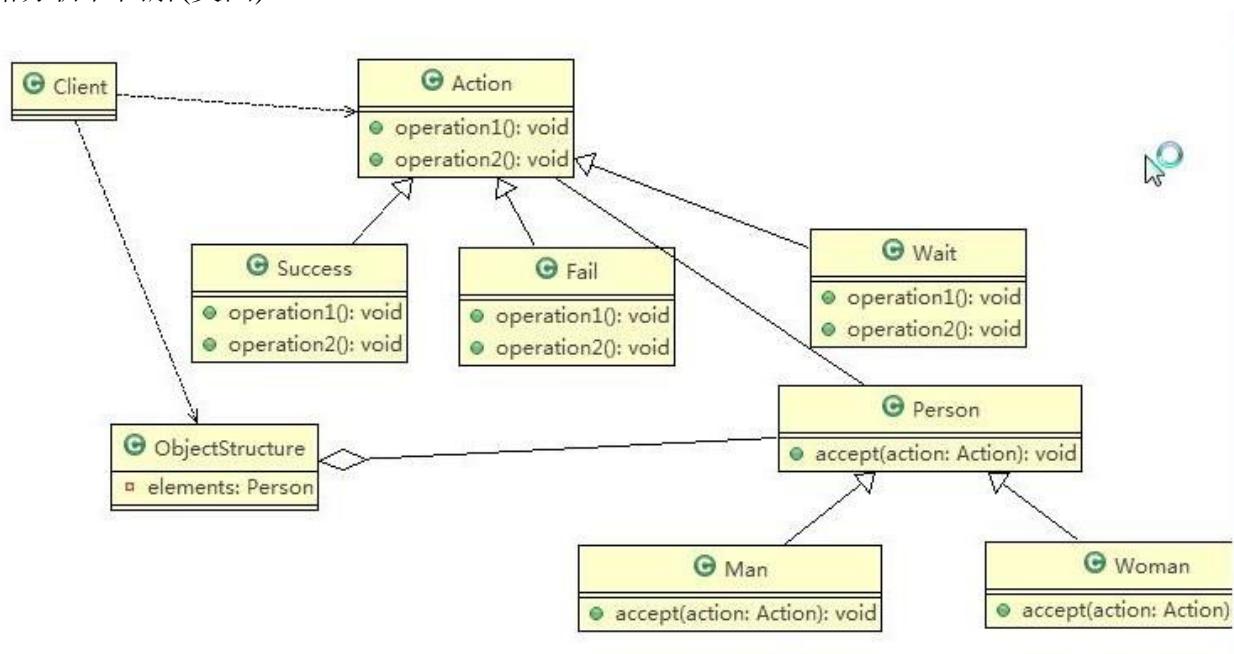
即(访问者模式的角色及职责)

- 1) Visitor 是抽象访问者，为该对象结构中的 ConcreteElement 的每一个类声明一个 visit 操作
- 2) ConcreteVisitor : 是一个具体的访问值 实现每个有 Visitor 声明的操作，是每个操作实现的部分.
- 3) ObjectStructure 能枚举它的元素， 可以提供一个高层的接口，用来允许访问者访问元素
- 4) Element 定义一个 accept 方法，接收一个访问者对象
- 5) ConcreteElement 为具体元素，实现了 accept 方法

18.5 访问者模式应用实例

应用实例要求

- 1) 将人分为男人和女人，对歌手进行测评，当看完某个歌手表演后，得到他们对该歌手不同的评价(评价有不同的种类，比如成功、失败等)，请使用访问者模式来说实现
- 2) 思路分析和图解(类图)



3) 代码实现

visitor.zip

```
package com.atguigu.visitor;

public abstract class Action {

    //得到男性 的测评
    public abstract void getManResult(Man man);

}
```



```
//得到女的 测评  
public abstract void getWomanResult(Woman woman);  
}
```

```
package com.atguigu.visitor;  
  
public class Client {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        //创建 ObjectStructure  
        ObjectStructure objectStructure = new ObjectStructure();  
  
        objectStructure.attach(new Man());  
        objectStructure.attach(new Woman());  
  
        //成功  
        Success success = new Success();  
        objectStructure.display(success);  
  
        System.out.println("=====");  
        Fail fail = new Fail();  
        objectStructure.display(fail);  
    }  
}
```



```
System.out.println("=====给的是待定的测评=====");
```

```
Wait wait = new Wait();
objectStructure.display(wait);
}

}
```

```
package com.atguigu.visitor;

public class Fail extends Action {

    @Override
    public void getManResult(Man man) {
        // TODO Auto-generated method stub
        System.out.println(" 男人给的评价该歌手失败 !");
    }

    @Override
    public void getWomanResult(Woman woman) {
        // TODO Auto-generated method stub
        System.out.println(" 女人给的评价该歌手失败 !");
    }
}
```



```
package com.atguigu.visitor;

public class Man extends Person {

    @Override
    public void accept(Action action) {
        // TODO Auto-generated method stub
        action.getManResult(this);
    }

}
```

```
package com.atguigu.visitor;

import java.util.LinkedList;
import java.util.List;

//数据结构，管理很多人（Man , Woman）
public class ObjectStructure {

    //维护了一个集合
    private List<Person> persons = new LinkedList<>();

    //增加到 list
    public void attach(Person p) {
        persons.add(p);
    }
}
```



```
}

//移除

public void detach(Person p)

{ persons.remove(p);

}

//显示测评情况

public void display(Action action)

{ for(Person p: persons) {

    p.accept(action);

}

}

}
```

```
package com.atguigu.visitor;
```

```
public abstract class Person {

    //提供一个方法，让访问者可以访问
    public abstract void accept(Action action);

}
```

```
package com.atguigu.visitor;
```

```
public class Success extends Action {
```



```
@Override  
public void getManResult(Man man) {  
    // TODO Auto-generated method stub  
    System.out.println(" 男人给的评价该歌手很成功 !");  
}
```

```
@Override  
public void getWomanResult(Woman woman) {  
    // TODO Auto-generated method stub  
    System.out.println(" 女人给的评价该歌手很成功 !");  
}
```

```
}
```

```
package com.atguigu.visitor;
```

```
public class Wait extends Action {
```

```
@Override  
public void getManResult(Man man) {  
    // TODO Auto-generated method stub  
    System.out.println(" 男人给的评价是该歌手待定 ..");  
}
```

```
@Override  
public void getWomanResult(Woman woman) {
```



```
// TODO Auto-generated method stub  
System.out.println(" 女人给的评价是该歌手待定 ..");  
}  
  
}
```

```
package com.atguigu.visitor;
```

```
//说明
```

```
//1. 这里我们使用到了双分派, 即首先在客户端程序中, 将具体状态作为参数传递 Woman 中(第一次分派)
```

```
//2. 然后 Woman 类调用作为参数的 "具体方法" 中方法 getWomanResult, 同时将自己(this)作为参数
```

```
// 传入, 完成第二次的分派
```

```
public class Woman extends Person{
```

```
    @Override
```

```
    public void accept(Action action) {
```

```
        // TODO Auto-generated method stub
```

```
        action.getWomanResult(this);
```

```
    }
```

```
}
```

4) 应用案例的小结-双分派

-上面提到了双分派, 所谓双分派是指不管类怎么变化, 我们都能找到期望的方法运行。双分派意味着得到执行的操作取决于请求的种类和两个接收者的类型



- 以上述实例为例，假设我们要添加一个 **Wait** 的状态类，考察 **Man** 类和 **Woman** 类的反应，由于使用了双分派，只需增加一个 Action 子类即可在客户端调用即可，不需要改动任何其他类的代码。

18.6 访问者模式的注意事项和细节

➤ 优点

- 1) 访问者模式符合单一职责原则、让程序具有优秀的扩展性、灵活性非常高
- 2) 访问者模式可以对功能进行统一，可以做报表、UI、拦截器与过滤器，适用于数据结构相对稳定的系统

➤ 缺点

- 1) 具体元素对访问者公布细节，也就是说访问者关注了其他类的内部细节，这是迪米特法则所不建议的，这样造成了具体元素变更比较困难
- 2) 违背了依赖倒转原则。访问者依赖的是具体元素，而不是抽象元素
- 3) 因此，如果一个系统有比较稳定的数据结构，又有经常变化的功能需求，那么访问者模式就是比较合适的。

第 19 章 迭代器模式

19.1 看一个具体的需求

编写程序展示一个学校院系结构：需求是这样，要在一个页面中展示出学校的院系组成，一个学校有多个学院，一个学院有多个系。如图：

```
-----计算机学院有以下专业-----
Java工程师
大数据工程师
前端工程师
信息安全
-----信息工程学院有以下专业-----
网络信息安全
电子技术
```

19.2 传统的设计方案(类图)



19.3 传统的方式的问题分析

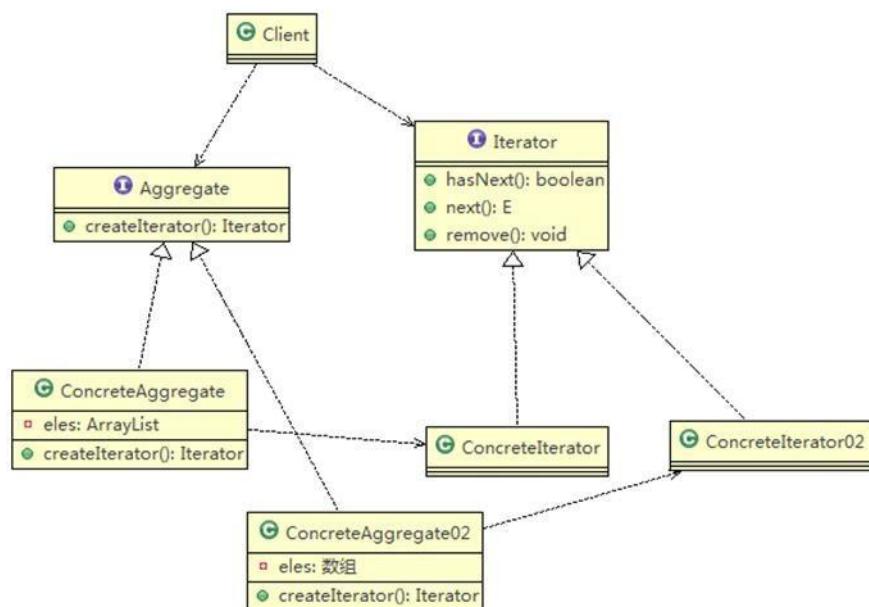
- 1) 将学院看做是学校的子类，系是学院的子类，这样实际上是站在组织大小来进行分层次的
- 2) 实际上我们的要求是：在一个页面中展示出学校的院系组成，一个学校有多个学院，一个学院有多个系，因此这种方案，不能很好实现的遍历的操作
- 3) 解决方案：=> 迭代器模式

19.4 迭代器模式基本介绍

基本介绍

- 1) 迭代器模式 (Iterator Pattern) 是常用的设计模式，属于行为型模式
- 2) 如果我们的集合元素是用不同的方式实现的，有数组，还有 java 的集合类，或者还有其他方式，当客户端要遍历这些集合元素的时候就要使用多种遍历方式，而且还会暴露元素的内部结构，可以考虑使用迭代器模式解决。
- 3) 迭代器模式，提供一种遍历集合元素的统一接口，用一致的方法遍历集合元素，不需要知道集合对象的底层表示，即：不暴露其内部的结构。

19.5 迭代器模式的原理类图



➤ 对原理类图的说明-即(迭代器模式的角色及职责)

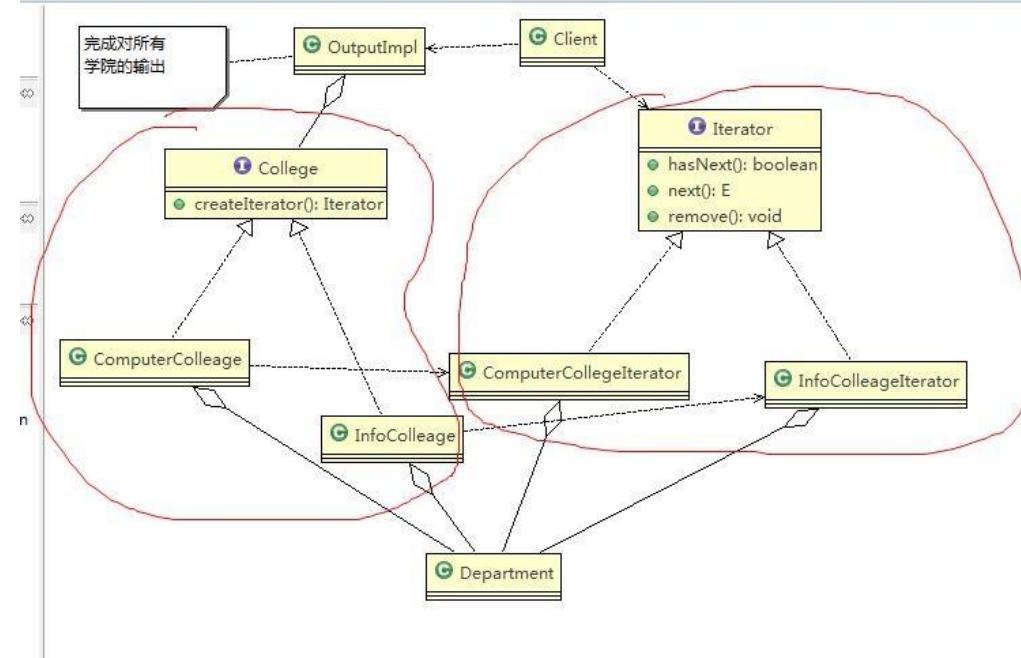
- 1) Iterator : 迭代器接口，是系统提供，含义 hasNext, next, remove
- 2) ConcreteIterator : 具体的迭代器类，管理迭代
- 3) Aggregate : 一个统一的聚合接口，将客户端和具体聚合解耦
- 4) ConcreteAggreate : 具体的聚合持有对象集合，并提供一个方法，返回一个迭代器，该迭代器可以正确遍历集合
- 5) Client : 客户端，通过 Iterator 和 Aggregate 依赖子类

19.6 迭代器模式应用实例

1) 应用实例要求

编写程序展示一个学校院系结构：需求是这样，要在一个页面中展示出学校的院系组成，一个学校有多个学院，一个学院有多个系。

2) 设计思路分析



3) 代码实现



```
package com.atguigu.iterator;

import java.util.ArrayList;
import java.util.List;
```



```
public class Client {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        //创建学院  
        List<College> collegeList = new ArrayList<College>();  
  
        ComputerCollege computerCollege = new ComputerCollege();  
        InfoCollege infoCollege = new InfoCollege();  
  
        collegeList.add(computerCollege);  
        //collegeList.add(infoCollege);  
  
        OutPutImpl outPutImpl = new OutPutImpl(collegeList);  
        outPutImpl.printCollege();  
    }  
  
}
```

```
package com.atguigu.iterator;  
  
import java.util.Iterator;  
  
public interface College {
```



```
public String getName();

//增加系的方法
public void addDepartment(String name, String desc);

//返回一个迭代器,遍历
public Iterator createIterator();
}
```

```
package com.atguigu.iterator;

import java.util.Iterator;

public class ComputerCollege implements College {

    Department[] departments;
    int numOfDepartment = 0;// 保存当前数组的对象个数

    public ComputerCollege()

    { departments = new
        Department[5];
        addDepartment("Java 专业", " Java 专业 ");
        addDepartment("PHP 专业", " PHP 专业 ");
        addDepartment("大数据专业", " 大数据专业 ");
    }

}
```



```
@Override  
public String getName() {  
    // TODO Auto-generated method stub  
    return "计算机学院";  
}  
  
@Override  
public void addDepartment(String name, String desc) {  
    // TODO Auto-generated method stub  
    Department department = new Department(name, desc);  
    departments[numOfDepartment] = department;  
    numOfDepartment += 1;  
}  
  
@Override  
public Iterator createIterator() {  
    // TODO Auto-generated method stub  
    return new ComputerCollegeIterator(departments);  
}  
}  
  
package com.atguigu.iterator;
```



```
import java.util.Iterator;

public class ComputerCollegeIterator implements Iterator {

    //这里我们需要 Department 是以怎样的方式存放=>数组
    Department[] departments;
    int position = 0; //遍历的位置

    public ComputerCollegeIterator(Department[] departments)
    {
        this.departments = departments;
    }

    //判断是否还有下一个元素
    @Override
    public boolean hasNext() {
        // TODO Auto-generated method stub
        if(position >= departments.length || departments[position] == null)
            { return false;
        }else {

            return true;
        }
    }
}
```



```
}
```

```
@Override
```

```
public Object next() {  
    // TODO Auto-generated method stub  
    Department department = departments[position];  
    position += 1;  
    return department;  
}
```

```
//删除的方法， 默认空实现
```

```
public void remove() {
```

```
}
```

```
}
```

```
package com.atguigu.iterator;
```

```
//系
```

```
public class Department {  
  
    private String name;  
    private String desc;  
    public Department(String name, String desc) {  
        super();  
    }
```



```
this.name = name;  
this.desc = desc;  
}  
public String getName()  
{ return name;  
}  
public void setName(String name)  
{ this.name = name;  
}  
public String getDesc()  
{ return desc;  
}  
public void setDesc(String desc)  
{ this.desc = desc;  
}  
}
```

```
package com.atguigu.iterator;  
  
import java.util.Iterator;  
import java.util.List;  
  
public class InfoColleageIterator implements Iterator {
```



```
List<Department> departmentList; // 信息工程学院是以 List 方式存放系  
int index = -1;//索引
```

```
public InfoColleageIterator(List<Department> departmentList)  
{ this.departmentList = departmentList;  
}
```

```
//判断 list 中还有没有下一个元素  
@Override  
public boolean hasNext() {  
    // TODO Auto-generated method stub  
    if(index >= departmentList.size() - 1) {  
        return false;  
    } else {  
        index += 1;  
        return true;  
    }  
}
```

```
@Override  
public Object next() {  
    // TODO Auto-generated method stub  
    return departmentList.get(index);
```



```
}
```

```
// 空实现 remove  
public void remove() {
```

```
}
```

```
}
```

```
package com.atguigu.iterator;  
  
import java.util.ArrayList;  
import java.util.Iterator;  
import java.util.List;  
  
public class InfoCollege implements College {  
  
    List<Department> departmentList;  
  
    public InfoCollege() {  
        departmentList = new ArrayList<Department>();  
        addDepartment("信息安全专业", " 信息安全专业 ");  
        addDepartment("网络安全专业", " 网络安全专业 ");  
        addDepartment("服务器安全专业", " 服务器安全专业 ");  
    }  
}
```



```
@Override  
public String getName() {  
    // TODO Auto-generated method stub  
    return "信息工程学院";  
}  
  
@Override  
public void addDepartment(String name, String desc) {  
    // TODO Auto-generated method stub  
    Department department = new Department(name, desc);  
    departmentList.add(department);  
}  
  
@Override  
public Iterator createIterator() {  
    // TODO Auto-generated method stub  
    return new InfoCollegeIterator(departmentList);  
}  
  
}  
  
package com.atguigu.iterator;  
  
import java.util.Iterator;  
import java.util.List;
```



```
public class OutPutImpl {  
  
    //学院集合  
    List<College> collegeList;  
  
    public OutPutImpl(List<College> collegeList) {  
  
        this.collegeList = collegeList;  
    }  
    //遍历所有学院,然后调用 printDepartment 输出各个学院的系  
    public void printCollege() {  
  
        //从 collegeList 取出所有学院, Java 中的 List 已经实现 Iterator  
        Iterator<College> iterator = collegeList.iterator();  
  
        while(iterator.hasNext()) {  
            //取出一个学院  
            College college = iterator.next();  
            System.out.println("==== "+college.getName() +"=====" );  
            printDepartment(college.createIterator()); //得到对应迭代器  
        }  
    }  
  
    //输出 学院输出 系
```

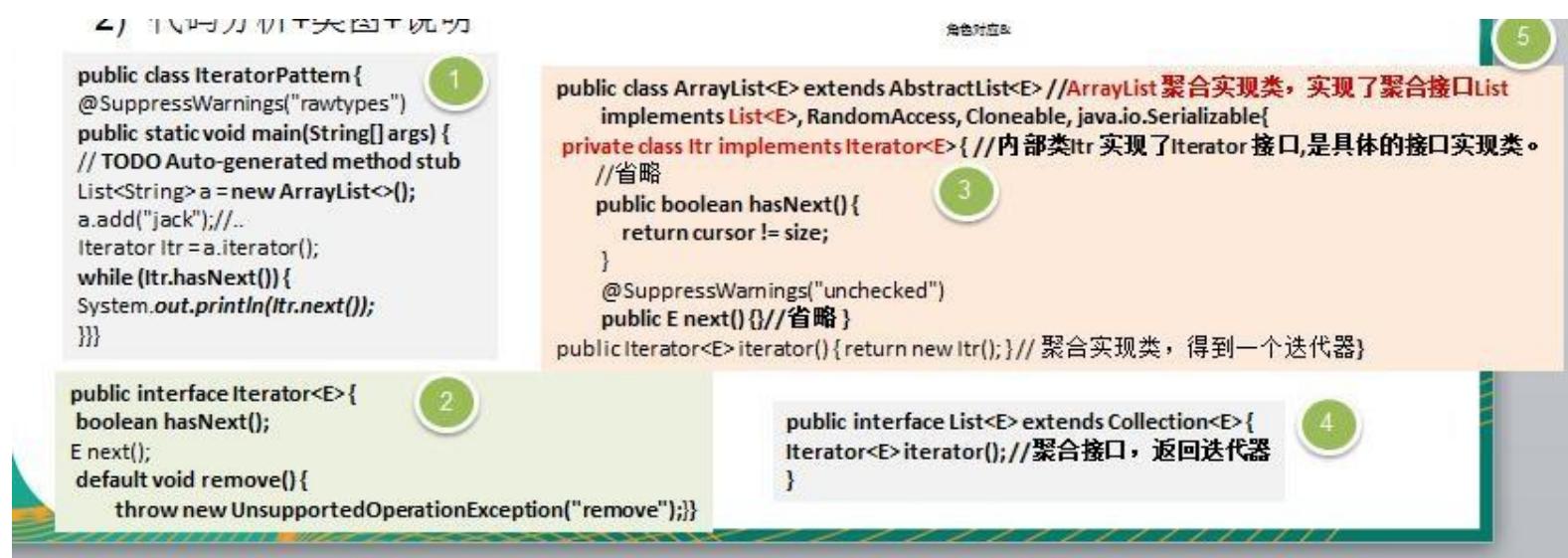
```

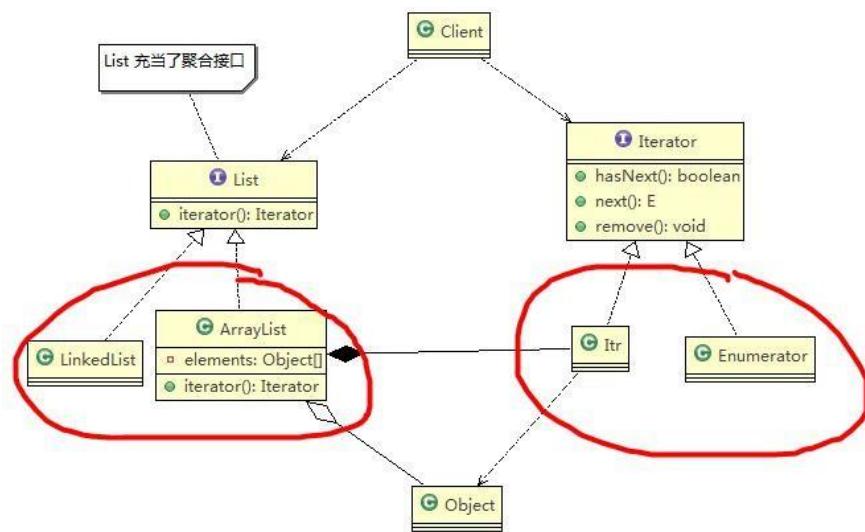
public void printDepartment(Iterator iterator)
{
    while(iterator.hasNext()) {
        Department d = (Department)iterator.next();
        System.out.println(d.getName());
    }
}

```

19.7 迭代器模式在 JDK-ArrayList 集合应用的源码分析

- 1) JDK 的 ArrayList 集合中就使用了迭代器模式
- 2) 代码分析+类图+说明





3) 对类图的角色分析和说明

- ✓ 内部类 Itr 充当具体实现迭代器 Iterator 的类，作为 ArrayList 内部类
- ✓ List 就是充当了聚合接口，含有一个 iterator() 方法，返回一个迭代器对象
- ✓ ArrayList 是实现聚合接口 List 的子类，实现了 iterator()
- ✓ Iterator 接口系统提供
- ✓ 迭代器模式解决了不同集合(ArrayList ,LinkedList) 统一遍历问题

19.8 迭代器模式的注意事项和细节

➤ 优点

- 1) 提供一个统一的方法遍历对象，客户不用再考虑聚合的类型，使用一种方法就可以遍历对象了。
- 2) 隐藏了聚合的内部结构，客户端要遍历聚合的时候只能取到迭代器，而不会知道聚合的具体组成。
- 3) 提供了一种设计思想，就是一个类应该只有一个引起变化的原因（叫做单一责任原则）。在聚合类中，我们把迭代器分开，就是要把管理对象集合和遍历对象集合的责任分开，这样一来集合改变的话，只影响到聚合对象。而如果遍历方式改变的话，只影响到了迭代器。
- 4) 当要展示一组相似对象，或者遍历一组相同对象时使用，适合使用迭代器模式



➤ 缺点

每个聚合对象都要一个迭代器，会生成多个迭代器不好管理类

第 20 章 观察者模式

20.1 天气预报项目需求, 具体要求如下:

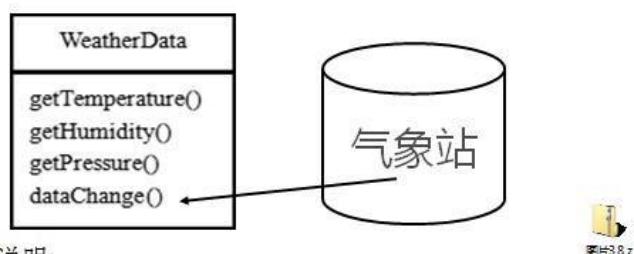
- 1) 气象站可以将每天测量到的温度, 湿度, 气压等等以公告的形式发布出去(比如发布到自己的网站或第三方)。
- 2) 需要设计开放型 **API**, 便于其他第三方也能接入气象站获取数据。
- 3) 提供温度、气压和湿度的接口
- 4) 测量数据更新时, 要能实时的通知给第三方

20.2 天气预报设计方案 1-普通方案

20.2.1 WeatherData 类

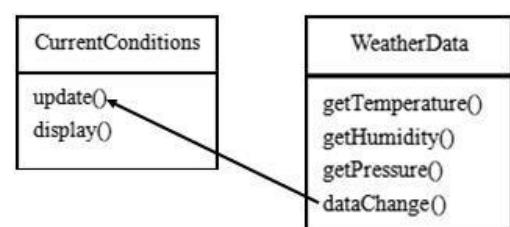
➤ 传统的设计方案

通过对气象站项目的分析, 我们可以初步设计出一个WeatherData类



说明:

- 1) 通过 `getXxx` 方法, 可以让第三方接入, 并得到相关信息。
- 2) 当数据有更新时, 气象站通过调用 `dataChange()` 去更新数据, 当第三方再次获取时, 就能得到最新数据, 当然也可以**推送**。



CurrentConditions(当前的天气情况)
可以理解成是我们气象局的网站 //**推送**



➤ 代码实现

```
package com.atguigu.observer;

public class Client {

    public static void main(String[] args) {
        //创建接入方 currentConditions
        CurrentConditions currentConditions = new CurrentConditions();
        //创建 WeatherData 并将 接入方 currentConditions 传递到 WeatherData 中
        WeatherData weatherData = new WeatherData(currentConditions);

        //更新天气情况
        weatherData.setData(30, 150, 40);

        //天气情况变化
        System.out.println("=====天气情况变化=====");
        weatherData.setData(40, 160, 20);

    }
}
```

```
package com.atguigu.observer;

/**
```



```
* 显示当前天气情况（可以理解成是气象站自己的网站）
* @author Administrator
*
*/
public class CurrentConditions {
    // 温度，气压，湿度
    private float temperature;
    private float pressure;
    private float humidity;

    //更新 天气情况，是由 WeatherData 来调用，我使用推送模式
    public void update(float temperature, float pressure, float humidity)
    {
        this.temperature = temperature;
        this.pressure = pressure;
        this.humidity = humidity;
        display();
    }

    //显示
    public void display() {
        System.out.println("***Today mTemperature: " + temperature + "***");
        System.out.println("***Today mPressure: " + pressure + "***");
        System.out.println("***Today mHumidity: " + humidity + "***");
    }
}
```



```
package com.atguigu.observer;

/**
 * 类是核心
 * 1. 包含最新的天气情况信息
 * 2. 含有 CurrentConditions 对象
 * 3. 当数据有更新时，就主动的调用 CurrentConditions 对象 update 方法(含 display)，这样他们（接入方）就看到最新的信息
 * @author Administrator
 *
 */
public class WeatherData
{
    private float temperatrue;
    private float pressure;
    private float humidity;
    private CurrentConditions currentConditions;
    //加入新的第三方

    public WeatherData(CurrentConditions currentConditions)
    {
        this.currentConditions = currentConditions;
    }

    public float getTemperature()
    {
        return temperatrue;
    }
}
```



```
public float getPressure()
{
    return pressure;
}

public float getHumidity()
{
    return humidity;
}

public void dataChange() {
    //调用 接入方的 update
    currentConditions.update(getTemperature(), getPressure(), getHumidity());
}

//当数据有更新时，就调用 setData
public void setData(float temperature, float pressure, float humidity)
{
    this.temperature = temperature;
    this.pressure = pressure;
    this.humidity = humidity;
    //调用 dataChange， 将最新的信息 推送给 接入方 currentConditions
    dataChange();
}

}
```

➤ 问题分析

1) 其他第三方接入气象站获取数据的问题

2) 无法在运行时动态的添加第三方 (新浪网站)

3) 违反 ocp 原则=>观察者模式

//在 WeatherData 中, 当增加一个第三方, 都需要创建一个对应的第三方的公告板对象, 并加入到dataChange, 不利于维护, 也不是动态加入

```
public void dataChange() {  
    currentConditions.update(getTemperature(), getPressure(), getHumidity());  
}
```

20.3 观察者模式原理

1) 观察者模式类似订牛奶业务

2) 奶站/气象局: Subject

3) 用户/第三方网站: Observer

➤ Subject: 登记注册、移除和通知

1) registerObserver 注册

2) removeObserver 移除

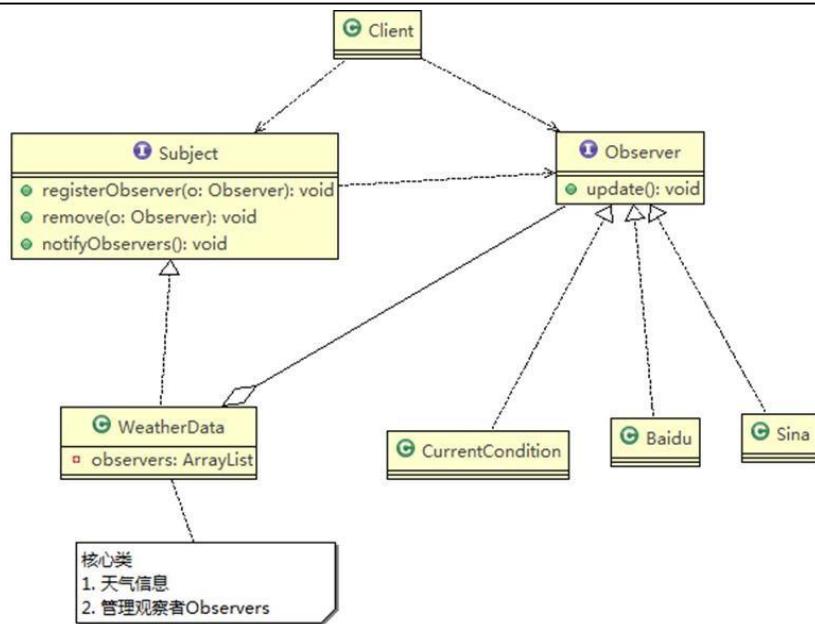
3) notifyObservers() 通知所有的注册的用户, 根据不同需求, 可以是更新数据, 让用户来取, 也可能是实施推送, 看具体需求定

➤ Observer: 接收输入

➤ 观察者模式: 对象之间多对一依赖的一种设计方案, 被依赖的对象为 Subject, 依赖的对象为 Observer, Subject 通知 Observer 变化, 比如这里的奶站是 Subject, 是 1 的一方。用户时 Observer, 是多的一方。

20.4 观察者模式解决天气预报需求

20.4.1 类图说明



20.4.2 代码实现



```
package com.atguigu.observer.improve;

public class BaiduSite implements Observer {

    // 温度, 气压, 湿度
    private float temperature;
    private float pressure;
    private float humidity;

    // 更新 天气情况, 是由 WeatherData 来调用, 我使用推送模式
    public void update(float temperature, float pressure, float humidity) {
        this.temperature = temperature;
    }
}
```



```
this.pressure = pressure;
this.humidity = humidity;
display();
}

// 显示
public void display() {
    System.out.println("==百度网站===");
    System.out.println("***百度网站 气温：" + temperature + "***");
    System.out.println("***百度网站 气压：" + pressure + "***");
    System.out.println("***百度网站 湿度：" + humidity + "***");
}

}
```

```
package com.atguigu.observer.improve;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //创建一个 WeatherData
        WeatherData weatherData = new WeatherData();

        //创建观察者
        CurrentConditions currentConditions = new CurrentConditions();
```



```
BaiduSite baiduSite = new BaiduSite();

// 注册到 weatherData
weatherData.registerObserver(currentConditions);
weatherData.registerObserver(baiduSite);

// 测试
System.out.println("通知各个注册的观察者，看看信息");
weatherData.setData(10f, 100f, 30.3f);

weatherData.removeObserver(currentConditions);
// 测试
System.out.println();
System.out.println("通知各个注册的观察者，看看信息");
weatherData.setData(10f, 100f, 30.3f);

}
```

```
package com.atguigu.observer.improve;

public class CurrentConditions implements Observer {

    // 温度，气压，湿度
    private float temperature;
```



```
private float pressure;  
private float humidity;  
  
// 更新天气情况，是由 WeatherData 来调用，我使用推送模式  
public void update(float temperature, float pressure, float humidity)  
{  
    this.temperature = temperature;  
    this.pressure = pressure;  
    this.humidity = humidity;  
    display();  
}  
  
// 显示  
public void display() {  
    System.out.println("***Today mTemperature: " + temperature + "***");  
    System.out.println("***Today mPressure: " + pressure + "***");  
    System.out.println("***Today mHumidity: " + humidity + "***");  
}
```

```
package com.atguigu.observer.improve;  
  
// 观察者接口，有观察者来实现  
public interface Observer {  
  
    public void update(float temperature, float pressure, float humidity);  
}
```



```
package com.atguigu.observer.improve;

//接口, 让 WeatherData 来实现
public interface Subject {

    public void registerObserver(Observer o);

    public void removeObserver(Observer o);

    public void notifyObservers();

}
```

```
package com.atguigu.observer.improve;

import java.util.ArrayList;

/**
 * 类是核心
 * 1. 包含最新的天气情况信息
 * 2. 含有 观察者集合, 使用 ArrayList 管理
 * 3. 当数据有更新时, 就主动的调用 ArrayList, 通知所有的 (接入方) 就看到最新的信息
 * @author Administrator
 *
 */
public class WeatherData implements Subject
{
    private float temperattrue;
    private float pressure;
```



```
private float humidity;
//观察者集合
private ArrayList<Observer> observers;

//加入新的第三方

public WeatherData() {
    observers = new ArrayList<Observer>();
}

public float getTemperature()
{
    return temperatrue;
}

public float getPressure()
{
    return pressure;
}

public float getHumidity()
{
    return humidity;
}

public void dataChange()
{
    //调用 接入方的 update

    notifyObservers();
}
```



```
}
```

```
//当数据有更新时，就调用 setData
public void setData(float temperature, float pressure, float humidity)
{
    this.temperatrue = temperature;
    this.pressure = pressure;
    this.humidity = humidity;

    //调用 dataChange， 将最新的信息推送给 接入方 currentConditions
    dataChange();
}

//注册一个观察者
@Override
public void registerObserver(Observer o) {
    // TODO Auto-generated method stub
    observers.add(o);
}

//移除一个观察者
@Override
public void removeObserver(Observer o) {
    // TODO Auto-generated method stub
    if(observers.contains(o)) {
        observers.remove(o);
    }
}
```



```
//遍历所有的观察者，并通知
@Override
public void notifyObservers() {
    // TODO Auto-generated method stub
    for(int i = 0; i < observers.size(); i++) {
        observers.get(i).update(this.temperatrue, this.pressure, this.humidity);
    }
}
```

20.4.3 观察者模式的好处

- 1) 观察者模式设计后，会以集合的方式来管理用户(Observer)，包括注册，移除和通知。
- 2) 这样，我们增加观察者(这里可以理解成一个新的公告板)，就不需要去修改核心类 WeatherData 不会修改代码，遵守了 ocp 原则。

20.5 观察者模式在 Jdk 应用的源码分析

- 1) Jdk 的 Observable 类就使用了观察者模式
- 2) 代码分析+模式角色分析

```
public class Observable {  
    private boolean changed = false;  
    private Vector<Observer> obs;  
  
    /** Construct an Observable with zero Observers. */  
    public Observable() { obs = new Vector<>(); }  
  
    // 1  
    public void registerObserver(Observer o) {  
        obs.add(o);  
    }  
  
    public void removeObserver(Observer o) {  
        obs.remove(o);  
    }  
  
    // 2  
    public void notifyObservers(Object arg) {  
        changed = true;  
        Iterator<Observer> iter = obs.iterator();  
        while (iter.hasNext()) {  
            Observer o = iter.next();  
            o.update(this, arg);  
        }  
    }  
}
```

3) 模式角色分析

- ✓ Observable 的作用和地位等价于我们前面讲过 Subject
- ✓ Observable 是类，不是接口，类中已经实现了核心的方法，即管理 Observer 的方法 add.. delete .. notify...
- ✓ Observer 的作用和地位等价于我们前面讲过的 Observer，有 update
- ✓ Observable 和 Observer 的使用方法和前面讲过的一样，只是 Observable 是类，通过继承来实现观察者模式

第 21 章 中介者模式

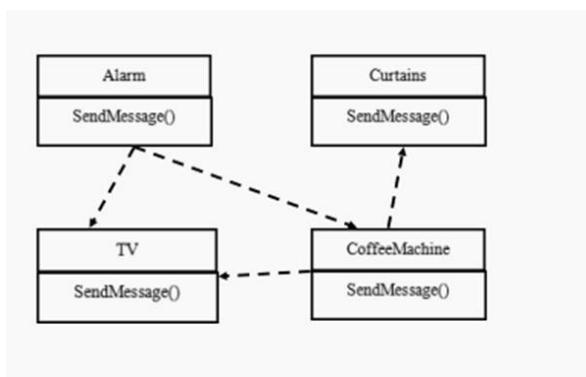
21.1 智能家庭项目



智能家居项目：

- 1) 智能家庭包括各种设备，闹钟、咖啡机、电视机、窗帘等
- 2) 主人要看电视时，各个设备可以协同工作，自动完成看电视的准备工作，比如流程为：闹铃响起->咖啡机开始做咖啡->窗帘自动落下->电视机开始播放

21.2 传统方案解决智能家庭管理问题



21.3 传统的方式的问题分析

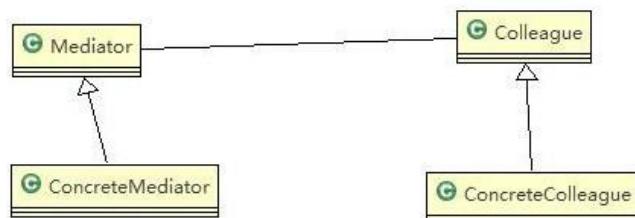
- 1) 当各电器对象有多种状态改变时，相互之间的调用关系会比较复杂
- 2) 各个电器对象彼此联系，你中有我，我中有你，不利于松耦合。
- 3) 各个电器对象之间所传递的消息(参数)，容易混乱
- 4) 当系统增加一个新的电器对象时，或者执行流程改变时，代码的可维护性、扩展性都不理想 考虑中介者模式

21.4 中介者模式基本介绍

基本介绍

- 1) 中介者模式（Mediator Pattern），用一个中介对象来封装一系列的对象交互。中介者使各个对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互
- 2) 中介者模式属于行为型模式，使代码易于维护
- 3) 比如 MVC 模式，C（Controller 控制器）是 M（Model 模型）和 V（View 视图）的中介者，在前后端交互时起到了中间人的作用

21.5 中介者模式的原理类图



➤ 对原理类图的说明-即(中介者模式的角色及职责)

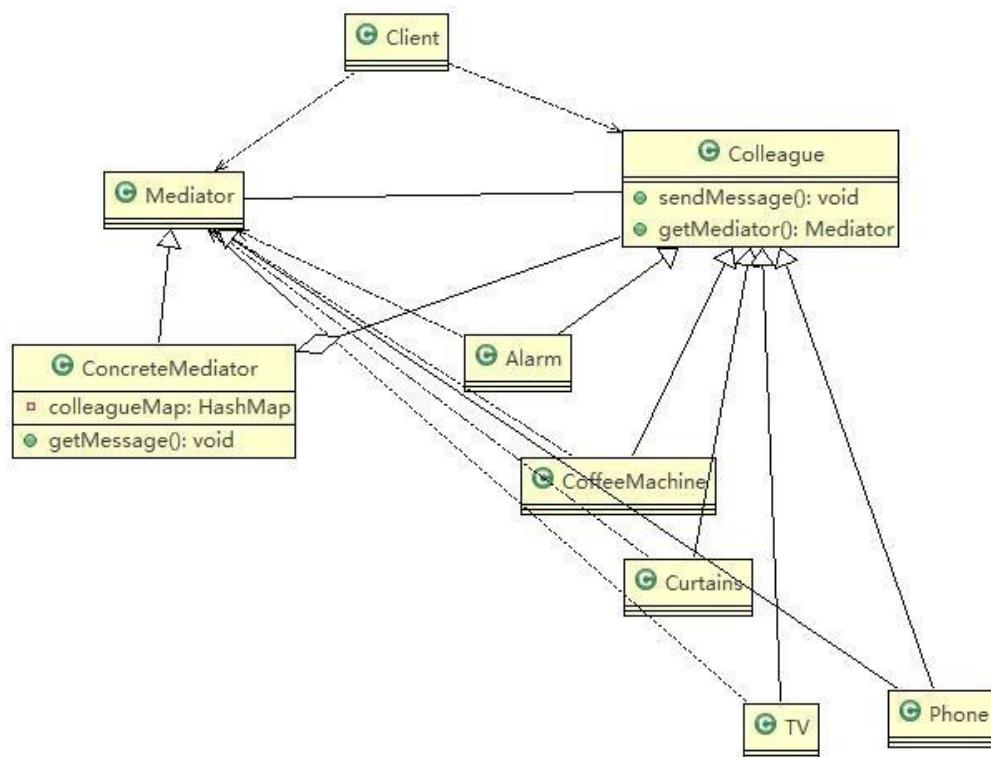
- 1) Mediator 就是抽象中介者,定义了同事对象到中介者对象的接口
- 2) Colleague 是抽象同事类
- 3) ConcreteMediator 具体的中介者对象, 实现抽象方法, 他需要知道所有的具体的同事类,即以一个集合来管理 HashMap,并接受某个同事对象消息, 完成相应的任务
- 4) ConcreteColleague 具体的同事类, 会有很多, 每个同事只知道自己的行为, 而不了解其他同事类的行为(方法), 但是他们都依赖中介者对象

21.6 中介者模式应用实例-智能家庭管理

1) 应用实例要求

完成前面的智能家庭的项目，使用中介者模式

2) 思路分析和图解(类图)



3) 代码实现



```

package com.atguigu.mediator.smarthouse;

//具体的同事类
public class Alarm extends Colleague {

    //构造器
    public Alarm(Mediator mediator, String name) {

```



```
super(mediator, name);

// TODO Auto-generated constructor stub
//在创建 Alarm 同事对象时，将自己放入到 ConcreteMediator 对象中[集合]
mediator.Register(name, this);

}

public void SendAlarm(int stateChange)

{ SendMessage(stateChange);

}

@Override

public void SendMessage(int stateChange) {

    // TODO Auto-generated method stub
    // 调用 的 中介者 对象 的 getMessage
    this.GetMediator().GetMessage(stateChange, this.name);

}

}
```

```
package com.atguigu.mediator.smarthouse;

public class ClientTest {

    public static void main(String[] args) {
        //创建一个中介者对象
        Mediator mediator = new ConcreteMediator();
```



```
//创建 Alarm 并且加入到 ConcreteMediator 对象的 HashMap  
Alarm alarm = new Alarm(mediator, "alarm");  
  
//创建了 CoffeeMachine 对象，并且加入到 ConcreteMediator 对象的 HashMap  
CoffeeMachine coffeeMachine = new CoffeeMachine(mediator,  
    "coffeeMachine");  
  
//创建 Curtains，并且加入到 ConcreteMediator 对象的 HashMap  
Curtains curtains = new Curtains(mediator, "curtains");  
TV tV = new TV(mediator, "TV");  
  
//让闹钟发出消息  
alarm.SendAlarm(0);  
coffeeMachine.FinishCoffee();  
alarm.SendAlarm(1);  
}  
}
```

```
package com.atguigu.mediator.smarthouse;  
  
public class CoffeeMachine extends Colleague {  
  
    public CoffeeMachine(Mediator mediator, String name)  
    { super(mediator, name);
```



```
// TODO Auto-generated constructor stub  
  
mediator.Register(name, this);  
  
}  
  
  
@Override  
  
public void SendMessage(int stateChange) {  
  
    // TODO Auto-generated method stub  
  
    this.GetMediator().GetMessage(stateChange, this.name);  
  
}  
  
  
public void StartCoffee()  
  
{ System.out.println("It's time to  
startcoffee!");  
  
}  
  
  
public void FinishCoffee() {  
  
    System.out.println("After 5 minutes!");  
  
    System.out.println("Coffee is ok!");  
  
    SendMessage(0);  
  
}  
  
}  
  
package com.atguigu.mediator.smarthouse;  
  
//同事抽象类  
public abstract class Colleague {
```



```
private Mediator mediator;  
  
public String name;  
  
public Colleague(Mediator mediator, String name) {  
  
    this.mediator = mediator;  
    this.name = name;  
  
}  
  
public Mediator GetMediator()  
{ return this.mediator;  
}  
  
public abstract void SendMessage(int stateChange);  
}
```

```
package com.atguigu.mediator.smarthouse;  
  
import java.util.HashMap;  
  
//具体的中介者类  
public class ConcreteMediator extends Mediator {  
    //集合，放入所有的同事对象  
    private HashMap<String, Colleague> colleagueMap;  
    private HashMap<String, String> interMap;
```



```
public ConcreteMediator() {  
    colleagueMap = new HashMap<String, Colleague>();  
    interMap = new HashMap<String, String>();  
}  
  
@Override  
public void Register(String colleagueName, Colleague colleague) {  
    // TODO Auto-generated method stub  
    colleagueMap.put(colleagueName, colleague);  
  
    // TODO Auto-generated method stub  
  
    if (colleague instanceof Alarm)  
        { interMap.put("Alarm",  
                     colleagueName);  
    } else if (colleague instanceof CoffeeMachine)  
        { interMap.put("CoffeeMachine", colleagueName);  
    } else if (colleague instanceof TV)  
        { interMap.put("TV",  
                     colleagueName);  
    } else if (colleague instanceof Curtains)  
        { interMap.put("Curtains", colleagueName);  
    }  
}
```



```
//具体中介者的核心方法
```



```
//1. 根据得到消息，完成对应任务
//2. 中介者在这个方法，协调各个具体的同事对象，完成任务
@Override
public void GetMessage(int stateChange, String colleagueName) {
    // TODO Auto-generated method stub

    //处理闹钟发出的消息
    if (colleagueMap.get(colleagueName) instanceof Alarm)
        { if (stateChange == 0) {
            ((CoffeeMachine) (colleagueMap.get(interMap
                .get("CoffeeMachine")))).StartCoffee();

            ((TV) (colleagueMap.get(interMap.get("TV")))).StartTv();
        } else if (stateChange == 1) {
            ((TV) (colleagueMap.get(interMap.get("TV")))).StopTv();
        }

    } else if (colleagueMap.get(colleagueName) instanceof CoffeeMachine)
        { ((Curtains) (colleagueMap.get(interMap.get("Curtains"))))
            .UpCurtains();

    } else if (colleagueMap.get(colleagueName) instanceof TV) {//如果 TV 发现消息

    } else if (colleagueMap.get(colleagueName) instanceof Curtains) {
        //如果是以窗帘发出的消息，这里处理...
    }
}
```



```
}
```

```
@Override  
public void SendMessage() {  
    // TODO Auto-generated method stub  
  
}  
  
}
```

```
package com.atguigu.mediator.smarthouse;  
  
public class Curtains extends Colleague {  
  
    public Curtains(Mediator mediator, String name)  
    { super(mediator, name);  
        // TODO Auto-generated constructor stub  
        mediator.Register(name, this);  
    }  
  
    @Override  
    public void SendMessage(int stateChange) {  
        // TODO Auto-generated method stub  
        this.GetMediator().GetMessage(stateChange, this.name);  
    }  
}
```



```
public void UpCurtains() {  
    System.out.println("I am holding Up Curtains!");  
}  
  
}
```

```
package com.atguigu.mediator.smarthouse;  
  
public abstract class Mediator {  
    //将给中介者对象，加入到集合中  
    public abstract void Register(String colleagueName, Colleague colleague);  
  
    //接收消息，具体的同事对象发出  
    public abstract void GetMessage(int stateChange, String colleagueName);  
  
    public abstract void SendMessage();  
}
```

```
package com.atguigu.mediator.smarthouse;  
  
public class TV extends Colleague {  
  
    public TV(Mediator mediator, String name)  
    { super(mediator, name);  
        // TODO Auto-generated constructor stub  
        mediator.Register(name, this);  
    }
```



```
}
```

```
@Override  
public void SendMessage(int stateChange) {  
    // TODO Auto-generated method stub  
    this.GetMediator().GetMessage(stateChange, this.name);  
}  
  
public void StartTv() {  
    // TODO Auto-generated method stub  
    System.out.println("It's time to StartTv!");  
}  
  
public void StopTv() {  
    // TODO Auto-generated method stub  
    System.out.println("StopTv!");  
}  
}
```

21.7 中介者模式的注意事项和细节

- 1) 多个类相互耦合，会形成网状结构，使用中介者模式将网状结构分离为星型结构，进行解耦
- 2) 减少类间依赖，降低了耦合，符合迪米特原则
- 3) 中介者承担了较多的责任，一旦中介者出现了问题，整个系统就会受到影响
- 4) 如果设计不当，中介者对象本身变得过于复杂，这点在实际使用时，要特别注意

第 22 章 备忘录模式

22.1 游戏角色状态恢复问题

游戏角色有攻击力和防御力，在大战 Boss 前保存自身的状态(攻击力和防御力)，当大战 Boss 后攻击力和防御力下降，从备忘录对象恢复到大战前的状态

22.2 传统方案解决游戏角色恢复



22.3 传统的方式的问题分析

- 1) 一个对象，就对应一个保存对象状态的对象，这样当我们游戏的对象很多时，不利于管理，开销也很大。
- 2) 传统的方式是简单地做备份，new 出另外一个对象出来，再把需要备份的数据放到这个新对象，但这就暴露了对象内部的细节
- 3) 解决方案：=> 备忘录模式

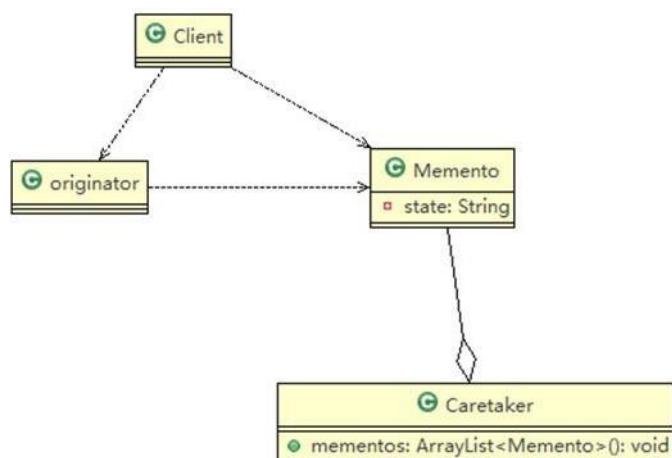
22.4 备忘录模式基本介绍

基本介绍

- 1) 备忘录模式（Memento Pattern）在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态

- 2) 可以这里理解备忘录模式：现实生活中的备忘录是用来记录某些要做的事情，或者是记录已经达成的共同意见的事情，以防忘记了。而在软件层面，备忘录模式有着相同的含义，备忘录对象主要用来记录一个对象的某种状态，或者某些数据，当要做回退时，可以从备忘录对象里获取原来的数据进行恢复操作
- 3) 备忘录模式属于行为型模式

22.5 备忘录模式的原理类图



- 对原理类图的说明-即(备忘录模式的角色及职责)
 - 1) originator : 对象(需要保存状态的对象)
 - 2) Memento : 备忘录对象,负责保存好记录, 即 Originator 内部状态
 - 3) Caretaker: 守护者对象,负责保存多个备忘录对象, 使用集合管理, 提高效率
 - 4) 说明: 如果希望保存多个 originator 对象的不同时间的状态, 也可以, 只需要要 HashMap <String, 集合>
- 代码实现



```
package com.atguigu.memento.theory;
```



```
import java.util.ArrayList;
import java.util.List;

public class Caretaker {

    //在 List 集合中会有很多的备忘录对象
    private List<Memento> mementoList = new ArrayList<Memento>();

    public void add(Memento memento)
    {
        mementoList.add(memento);
    }

    //获取到第 index 个 Originator 的 备忘录对象(即保存状态)
    public Memento get(int index) {
        return mementoList.get(index);
    }
}
```

```
package com.atguigu.memento.theory;

import java.util.ArrayList;
import java.util.HashMap;

public class Client {
```



```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
  
    Originator originator = new Originator();  
    Caretaker caretaker = new Caretaker();  
  
    originator.setState(" 状态#1 攻击力 100 ");  
  
    //保存了当前的状态  
    caretaker.add(originator.saveStateMemento());  
  
    originator.setState(" 状态 #2 攻击力 80 ");  
  
    caretaker.add(originator.saveStateMemento());  
  
    originator.setState(" 状态 #3 攻击力 50 ");  
    caretaker.add(originator.saveStateMemento());  
  
    System.out.println("当前的状态是 = " + originator.getState());  
  
    //希望得到状态 1, 将 originator 恢复到状态 1  
  
    originator.getStateFromMemento(caretaker.get(0));  
    System.out.println("恢复到状态 1 , 当前的状态是");
```



```
System.out.println("当前的状态是 = " + originator.getState());
```

```
}
```

```
}
```

```
package com.atguigu.memento.theory;
```

```
public class Memento
```

```
    { private String
```

```
        state;
```

```
//构造器
```

```
public Memento(String state)
```

```
    { super();
```

```
        this.state = state;
```

```
}
```

```
public String getState()
```

```
    { return state;
```

```
}
```

```
}
```



```
package com.atguigu.memento.theory;

public class Originator {

    private String state;//状态信息

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    //编写一个方法，可以保存一个状态对象 Memento
    //因此编写一个方法，返回 Memento
    public Memento saveStateMemento()
    {
        return new Memento(state);
    }

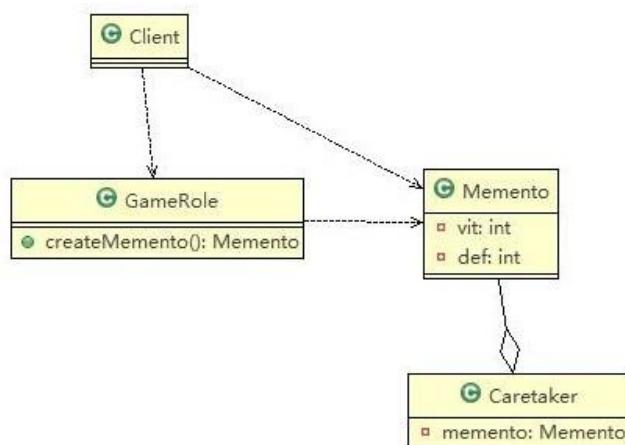
    //通过备忘录对象，恢复状态
    public void getStateFromMemento(Memento memento)
    {
        state = memento.getState();
    }
}
```

22.6 游戏角色恢复状态实例

1) 应用实例要求

游戏角色有攻击力和防御力，在大战 Boss 前保存自身的状态(攻击力和防御力)，当大战 Boss 后攻击力和防御力下降，从备忘录对象恢复到大战前的状态

2) 思路分析和图解(类图)



3) 代码实现

game.zip

```
package com.atguigu.memento.game;

import java.util.ArrayList;
import java.util.HashMap;
```



```
//守护者对象, 保存游戏角色的状态
public class Caretaker {

    //如果只保存一次状态
    private Memento memento;
    //对 GameRole 保存多次状态
    //private ArrayList<Memento> mementos;
    //对多个游戏角色保存多个状态
    //private HashMap<String, ArrayList<Memento>> rolesMementos;

    public Memento getMemento()
    {
        return memento;
    }

    public void setMemento(Memento memento)
    {
        this.memento = memento;
    }

}
```

```
package com.atguigu.memento.game;

public class Client {

    public static void main(String[] args) {
```



```
// TODO Auto-generated method stub
//创建游戏角色
GameRole gameRole = new GameRole();
gameRole.setVit(100);
gameRole.setDef(100);

System.out.println("和 boss 大战前的状态");
gameRole.display();

//把当前状态保存 caretaker
Caretaker caretaker = new Caretaker();
caretaker.setMemento(gameRole.createMemento());

System.out.println("和 boss 大战~~~");
gameRole.setDef(30);
gameRole.setVit(30);

gameRole.display();

System.out.println("大战后， 使用备忘录对象恢复到站前");

gameRole.recoverGameRoleFromMemento(caretaker.getMemento());
System.out.println("恢复后的状态");
gameRole.display();

}
```



```
}
```

```
package com.atguigu.memento.game;

public class GameRole {

    private int vit;
    private int def;

    //创建 Memento ,即根据当前的状态得到 Memento
    public Memento createMemento() {
        return new Memento(vit, def);
    }

    //从备忘录对象，恢复 GameRole 的状态
    public void recoverGameRoleFromMemento(Memento memento)
    {
        this.vit = memento.getVit();
        this.def = memento.getDef();
    }

    //显示当前游戏角色的状态
    public void display() {
        System.out.println("游戏角色当前的攻击力: " + this.vit + " 防御力: " + this.def);
    }

    public int getVit() {
```

```
    return vit;  
}  
  
public void setVit(int vit)  
{ this.vit = vit;  
}  
  
public int getDef()  
{ return def;  
}  
  
public void setDef(int def)  
{ this.def = def;  
}  
}
```

```
package com.atguigu.memento.game;  
  
public class Memento {  
  
    //攻击力  
    private int vit;  
    //防御力  
    private int def;
```



```
public Memento(int vit, int def)
{
    super();
    this.vit = vit;
    this.def = def;
}

public int getVit()
{
    return vit;
}

public void setVit(int vit)
{
    this.vit = vit;
}

public int getDef()
{
    return def;
}

public void setDef(int def)
{
    this.def = def;
}
```

22.7 备忘录模式的注意事项和细节

- 1) 给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态
- 2) 实现了信息的封装，使得用户不需要关心状态的保存细节
- 3) 如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存，这个需要注意
- 4) 适用的应用场景：1、后悔药。2、打游戏时的存档。3、Windows 里的 `ctrl + z`。4、IE 中的后退。4、数据库的事务管理



5) 为了节约内存，备忘录模式可以和原型模式配合使用

第 23 章 解释器模式

23.1 四则运算问题

通过解释器模式来实现四则运算，如计算 $a+b-c$ 的值，具体要求

- 1) 先输入表达式的形式，比如 $a+b+c-d+e$ ，要求表达式的字母不能重复
- 2) 在分别输入 a, b, c, d, e 的值
- 3) 最后求出结果：如图

```
请输入表达式: a+b+c-d+e
请输入a的值: 10
请输入b的值: 11
请输入c的值: 1
请输入d的值: 2
请输入e的值: 3
运算结果: a+b+c-d+e=23
```

23.2 传统方案解决四则运算问题分析

- 1) 编写一个方法，接收表达式的形式，然后根据用户输入的数值进行解析，得到结果
- 2) 问题分析：如果加入新的运算符，比如 $*/$ （等等，不利于扩展，另外让一个方法来解析会造成程序结构混乱，不够清晰。
- 3) 解决方案：可以考虑使用解释器模式，即： 表达式 \rightarrow 解释器(可以有多种) \rightarrow 结果

23.3 解释器模式基本介绍

基本介绍

- 1) 在编译原理中，一个算术表达式通过词法分析器形成词法单元，而后这些词法单元再通过语法分析器构建语法分析树，最终形成一颗抽象的语法分析树。这里的词法分析器和语法分析器都可以看做是解释器
- 2) 解释器模式（Interpreter Pattern）：是指给定一个语言(表达式)，定义它的文法的一种表示，并定义一个解释器，使用该解释器来解释语言中的句子(表达式)
- 3) 应用场景

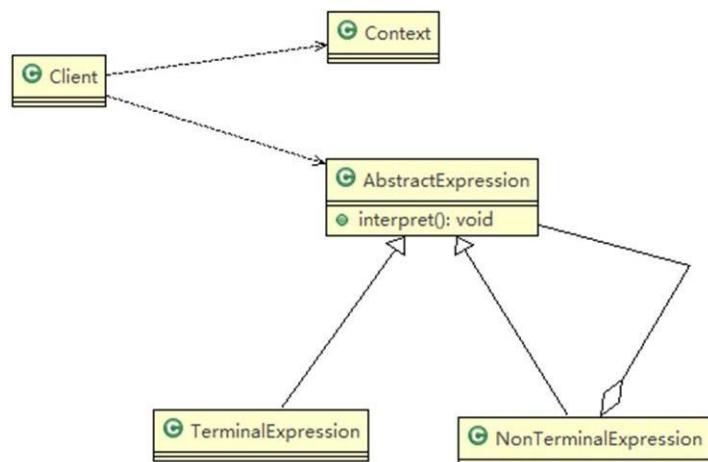
-应用可以将一个需要解释执行的语言中的句子表示为一个抽象语法树

-一些重复出现的问题可以用一种简单的语言来表达

-一个简单语法需要解释的场景

- 4) 这样的例子还有，比如编译器、运算表达式计算、正则表达式、机器人等

23.4 解释器模式的原理类图



➤ 对原理类图的说明-即(解释器模式的角色及职责)

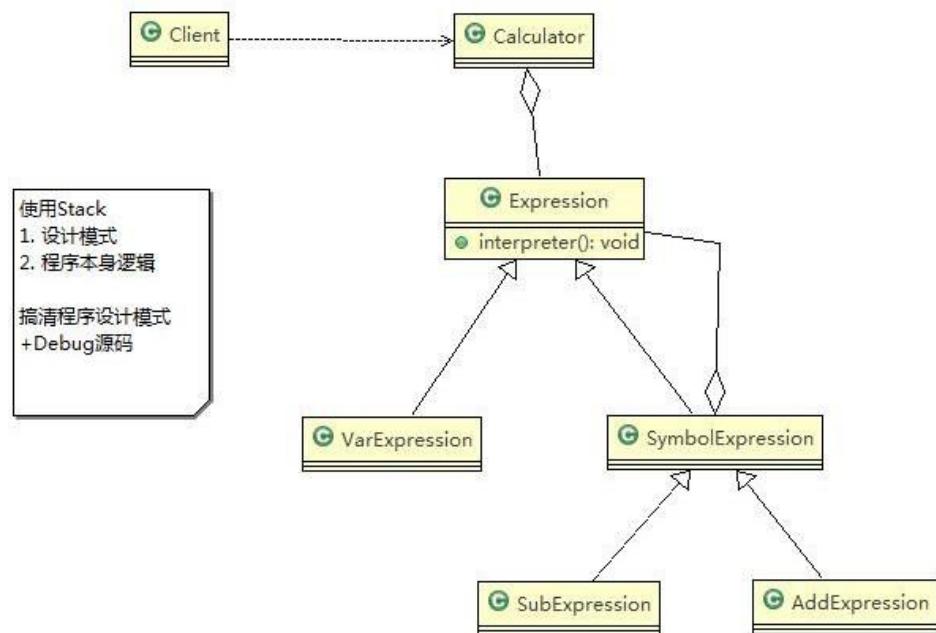
- 1) Context: 是环境角色,含有解释器之外的全局信息.
- 2) AbstractExpression: 抽象表达式, 声明一个抽象的解释操作,这个方法为抽象语法树中所有的节点所共享
- 3) TerminalExpression: 为终结符表达式, 实现与文法中的终结符相关的解释操作
- 4) NonTermialExpression: 为非终结符表达式, 为文法中的非终结符实现解释操作.
- 5) 说明: 输入 Context he TerminalExpression 信息通过 Client 输入即可

23.5 解释器模式来实现四则

- 1) 应用实例要求

通过解释器模式来实现四则运算，
如计算 $a+b-c$ 的值

2) 思路分析和图解(类图)



3) 代码实现


interpreter.zip

```

package com.atguigu.interpreter;

import java.util.HashMap;

/**
 * 加法解释器
 * @author Administrator
 *
 */
public class AddExpression extends SymbolExpression {
  
```



```
public AddExpression(Expression left, Expression right)
    { super(left, right);
    }

//处理相加
//var 仍然是 {a=10,b=20}..
//一会我们 debug 源码,就 ok
public int interpreter(HashMap<String, Integer> var) {
    //super.left.interpreter(var) : 返回 left 表达式对应的值 a = 10
    //super.right.interpreter(var): 返回 right 表达式对应值 b = 20
    return super.left.interpreter(var) + super.right.interpreter(var);
}

}
```

```
package com.atguigu.interpreter;
```

```
import java.util.HashMap;
import java.util.Stack;

public class Calculator {

    // 定义表达式
    private Expression expression;

    // 构造函数传参，并解析
}
```



```
public Calculator(String expStr) { // expStr = a+b
    // 安排运算先后顺序
    Stack<Expression> stack = new Stack<>();
    // 表达式拆分成字符数组
    char[] charArray = expStr.toCharArray(); // [a, +, b]

    Expression left = null;
    Expression right = null;
    // 遍历我们的字符数组， 即遍历 [a, +, b]
    // 针对不同的情况， 做处理
    for (int i = 0; i < charArray.length; i++) {
        switch (charArray[i]) {
            case '+': //
                left = stack.pop(); // 从 stack 取出 left => "a"
                right = new VarExpression(String.valueOf(charArray[++i])); // 取出右表达式 "b"
                stack.push(new AddExpression(left, right)); // 然后根据得到 left 和 right 构建 AddExpression 加入
                stack
                break;
            case '-': //
                left = stack.pop();
                right = new VarExpression(String.valueOf(charArray[++i]));
                stack.push(new SubExpression(left, right));
                break;
            default:
                // 如果是一个 Var 就创建要给 VarExpression 对象，并 push 到 stack
                stack.push(new VarExpression(String.valueOf(charArray[i])));
        }
    }
}
```



```
        break;
    }
}

//当遍历完整个 charArray 数组后， stack 就得到最后 Expression
this.expression = stack.pop();

}

public int run(HashMap<String, Integer> var) {
    //最后将表达式 a+b 和 var = {a=10,b=20}
    //然后传递给 expression 的 interpreter 进行解释执行
    return this.expression.interpreter(var);
}
```

```
package com.atguigu.interpreter;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;

public class ClientTest {

    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub
        String expStr = getExpStr(); // a+b
        HashMap<String, Integer> var = getValue(expStr); // var {a=10, b=20}
    }
}
```



```
Calculator calculator = new Calculator(expStr);
System.out.println("运算结果: " + expStr + "=" + calculator.run(var));
}

// 获得表达式
public static String getExpStr() throws IOException {
    System.out.print("请输入表达式: ");
    return (new BufferedReader(new InputStreamReader(System.in))).readLine();
}

// 获得值映射
public static HashMap<String, Integer> getValue(String expStr) throws IOException
{
    HashMap<String, Integer> map = new HashMap<>();

    for (char ch : expStr.toCharArray())
    {
        if (ch != '+' && ch != '-')
        {
            if (!map.containsKey(String.valueOf(ch)))
            {
                System.out.print("请输入" + String.valueOf(ch) + "的值: ");
                String in = (new BufferedReader(new InputStreamReader(System.in))).readLine();
                map.put(String.valueOf(ch), Integer.valueOf(in));
            }
        }
    }

    return map;
}
```



```
}
```

```
package com.atguigu.interpreter;

import java.util.HashMap;

/**
 * 抽象类表达式，通过 HashMap 键值对，可以获取到变量的值
 *
 * @author Administrator
 *
 */
public abstract class Expression {

    // a + b - c
    // 解释公式和数值, key 就是公式(表达式) 参数[a,b,c], value 就是具体值
    // HashMap {a=10, b=20}
    public abstract int interpreter(HashMap<String, Integer> var);

}
```

```
package com.atguigu.interpreter;

import java.util.HashMap;

public class SubExpression extends SymbolExpression {

    public SubExpression(Expression left, Expression right) {
```



```
super(left, right);

}

//求出 left 和 right 表达式相减后的结果
public int interpreter(HashMap<String, Integer> var) {
    return super.left.interpreter(var) - super.right.interpreter(var);
}
```

```
package com.atguigu.interpreter;
```

```
import java.util.HashMap;
```

```
/***
 * 抽象运算符号解析器 这里，每个运算符号，都只和自己左右两个数字有关系，
 * 但左右两个数字有可能也是一个解析的结果，无论何种类型，都是 Expression 类的实现类
 *
 * @author Administrator
 *
 */

```

```
public class SymbolExpression extends Expression {
```

```
    protected Expression left;
    protected Expression right;
```

```
    public SymbolExpression(Expression left, Expression right) {
```



```
this.left = left;
this.right = right;
}

//因为 SymbolExpression 是让其子类来实现，因此 interpreter 是一个默认实现
@Override
public int interpreter(HashMap<String, Integer> var) {
    // TODO Auto-generated method stub
    return 0;
}
```

```
package com.atguigu.interpreter;

import java.util.HashMap;

/**
 * 变量的解释器
 * @author Administrator
 *
 */
public class VarExpression extends Expression {

    private String key; // key=a,key=b,key=c
```

```

public VarExpression(String key)

    { this.key = key;

    }

// var 就是{a=10, b=20}

// interpreter 根据 变量名称， 返回对应值

@Override

public int interpreter(HashMap<String, Integer> var)

    { return var.get(this.key);

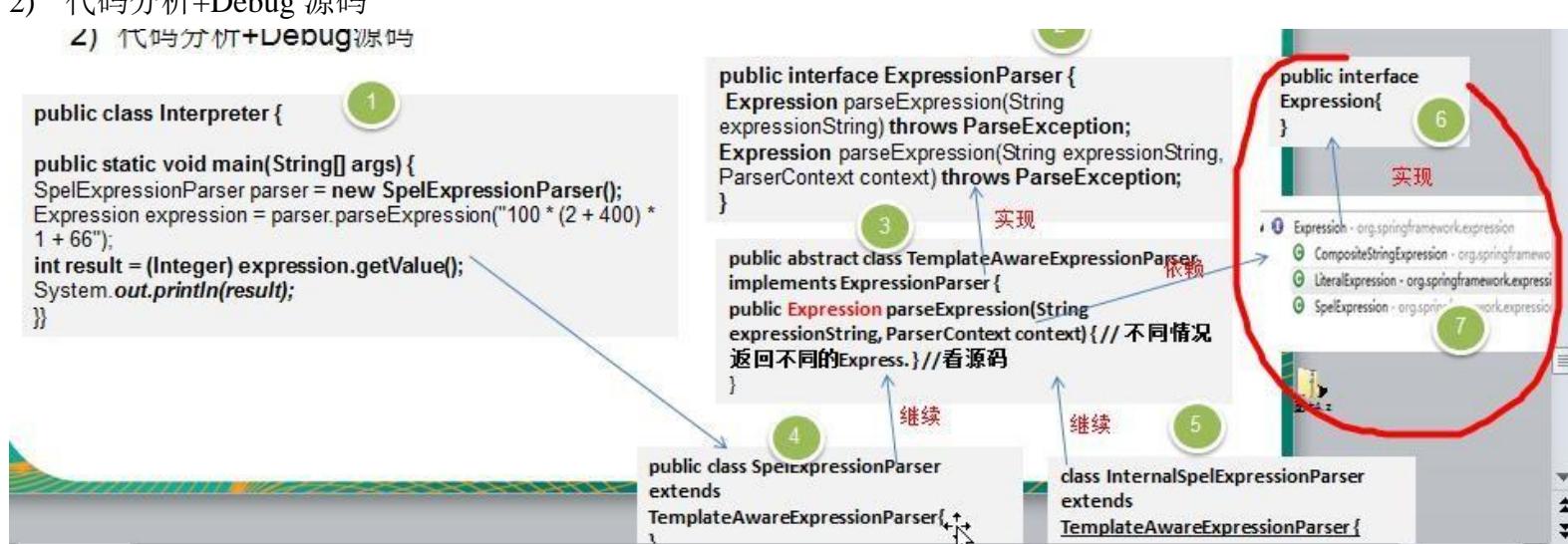
    }

}

```

23.6 解释器模式在 Spring 框架应用的源码剖析

- 1) Spring 框架中 SpelExpressionParser 就使用到解释器模式
- 2) 代码分析+Debug 源码
 ↳ 代码分析+Debug 源码



3) 说明

- Expression 接口 表达式接口
- 下面有不同的实现类，比如 SpelExpression，或者 CompositeStringExpression。
- 使用时候，根据你创建的不同的Parser 对象，返回不同的 Expression 对象

```
public Expression parseExpression(String expressionString, ParserContext context)
throws ParseException {
if (context == null) {
context = NON_TEMPLATE_PARSER_CONTEXT;
}

if (context.isTemplate()) {
return parseTemplate(expressionString, context); //返回的就是 CompositeStringExpression
}
else {
return doParseExpression(expressionString, context); //返回的就是 SpelExpression
}
}
```

- 使用得当 Expression 对象，调用 getValue 解释执行表达式，最后得到结果

23.7 解释器模式的注意事项和细节

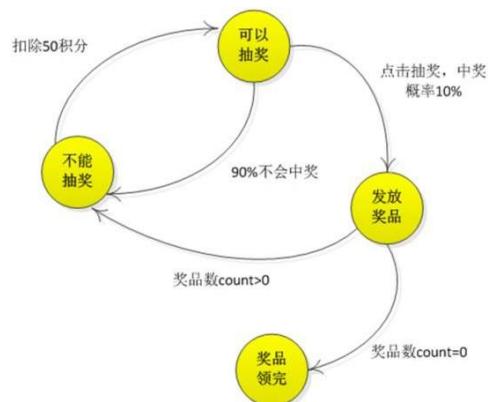
- 1) 当有一个语言需要解释执行，可将该语言中的句子表示为一个抽象语法树，就可以考虑使用解释器模式，让程序具有良好的扩展性
- 2) 应用场景：编译器、运算表达式计算、正则表达式、机器人等
- 3) 使用解释器可能带来的问题：解释器模式会引起类膨胀、解释器模式采用递归调用方法，将会导致调试非常复杂、效率可能降低。

第 24 章 状态模式

24.1 APP 抽奖活动问题

请编写程序完成 APP 抽奖活动 具体要求如下：

- 1) 假如每参加一次这个活动要扣除用户 50 积分，中奖概率是 10%
- 2) 奖品数量固定，抽完就不能抽奖
- 3) 活动有四个状态：可以抽奖、不能抽奖、发放奖品和奖品领完
- 4) 活动的四个状态转换关系图(右图)

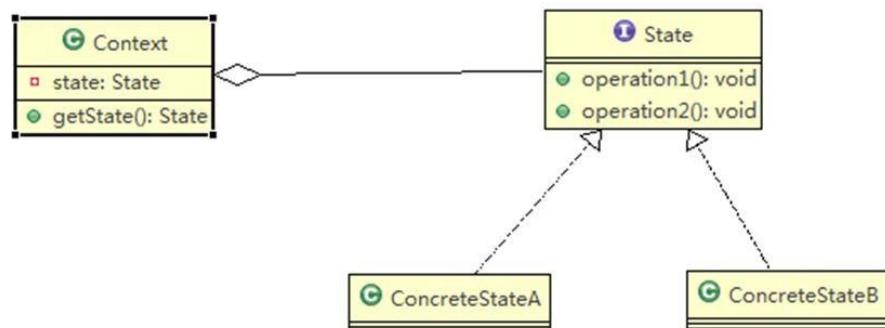


24.2 状态模式基本介绍

基本介绍

- 1) 状态模式（**State Pattern**）：它主要用来解决对象在多种状态转换时，需要对外输出不同的行为的问题。状态和行为是一一对应的，状态之间可以相互转换
- 2) 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

24.3 状态模式的原理类图



➤ 对原理类图的说明-即(状态模式的角色及职责)

- 1) Context 类为环境角色, 用于维护 State 实例, 这个实例定义当前状态
- 2) State 是抽象状态角色, 定义一个接口封装与 Context 的一个特点接口相关行为
- 3) ConcreteState 具体的状态角色, 每个子类实现一个与 Context 的一个状态相关行为

24.4 状态模式解决 APP 抽奖问题

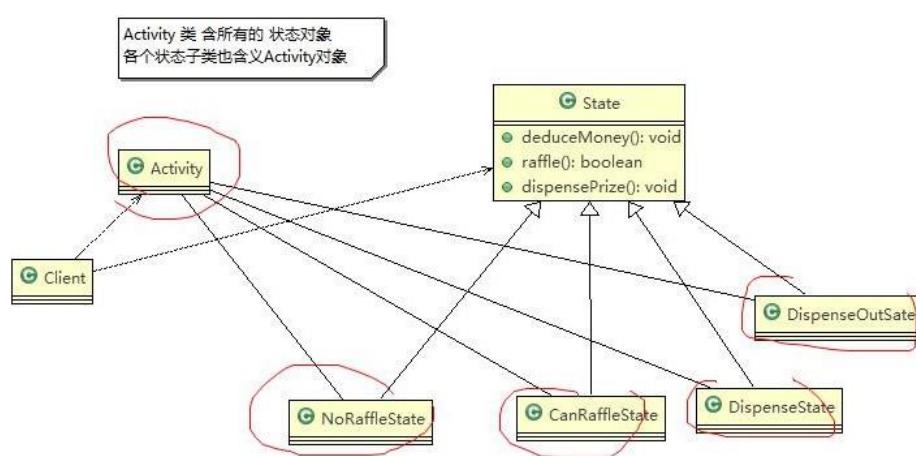
- 1) 应用实例要求

完成 APP 抽奖活动项目, 使用状态模式.

- 2) 思路分析和图解(类图)

-定义出一个接口叫状态接口, 每个状态都实现它。

-接口有扣除积分方法、抽奖方法、发放奖品方法





3) 代码实现

state.zip

```
package com.atguigu.state;

import java.util.Random;

/**
 * 可以抽奖的状态
 * @author Administrator
 *
 */
public class CanRaffleState extends State {

    RaffleActivity activity;

    public CanRaffleState(RaffleActivity activity) {
        this.activity = activity;
    }

    //已经扣除了积分，不能再扣
    @Override
    public void deductMoney() {
        System.out.println("已经扣取过了积分");
    }
}
```



```
//可以抽奖，抽完奖后，根据实际情况，改成新的状态
@Override
public boolean raffle()
{ System.out.println("正在抽奖，请稍等！");
  Random r = new Random();

  int num = r.nextInt(10);
  // 10% 中奖机会
  if(num == 0){
    // 改变活动状态为发放奖品 context
    activity.setState(activity.getDispenseState());

    return true;
  }else{
    System.out.println("很遗憾没有抽中奖品！");
    // 改变状态为不能抽奖
    activity.setState(activity.getNoRaffleState());
    return false;
  }
}

// 不能发放奖品
@Override
public void dispensePrize() {
  System.out.println("没中奖，不能发放奖品");
}
}
```



```
package com.atguigu.state;

/**
 * 状态模式测试类
 * @author Administrator
 *
 */
public class ClientTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // 创建活动对象，奖品有 1 个奖品
        RaffleActivity activity = new RaffleActivity(1);

        // 我们连续抽 300 次奖
        for (int i = 0; i < 30; i++) {
            System.out.println("-----第" + (i + 1) + "次抽奖-----");
            // 参加抽奖，第一步点击扣除积分
            activity.debuctMoney();

            // 第二步抽奖
            activity.raffle();
        }
    }
}
```



```
package com.atguigu.state;

/**
 * 奖品发放完毕状态
 * 说明，当我们 activity 改变成 DispenseOutState， 抽奖活动结束
 * @author Administrator
 *
 */
public class DispenseOutState extends State {

    // 初始化时传入活动引用
    RaffleActivity activity;

    public DispenseOutState(RaffleActivity activity) {
        this.activity = activity;
    }

    @Override
    public void deductMoney() {
        System.out.println("奖品发送完了，请下次再参加");
    }

    @Override
    public boolean raffle() {
        System.out.println("奖品发送完了，请下次再参加");
        return false;
    }
}
```



```
}
```

```
@Override  
public void dispensePrize() {  
    System.out.println("奖品发送完了, 请下次再参加");  
}  
}
```

```
package com.atguigu.state;  
  
/**  
 * 发放奖品的状态  
 * @author Administrator  
 *  
 */  
public class DispenseState extends State {  
  
    // 初始化时传入活动引用, 发放奖品后改变其状态  
    RaffleActivity activity;  
  
    public DispenseState(RaffleActivity activity)  
    { this.activity = activity;  
    }  
  
    //
```



```
@Override  
public void deductMoney() {  
    System.out.println("不能扣除积分");  
}  
  
@Override  
public boolean raffle() {  
    System.out.println("不能抽奖");  
    return false;  
}  
  
//发放奖品  
@Override  
public void dispensePrize()  
{ if(activity.getCount() > 0){  
    System.out.println("恭喜中奖了");  
    // 改变状态为不能抽奖  
    activity.setState(activity.getNoRaffleState());  
}  
else{  
    System.out.println("很遗憾， 奖品发送完了");  
    // 改变状态为奖品发送完毕, 后面我们就不可以抽奖  
    activity.setState(activity.getDispensOutState());  
    //System.out.println("抽奖活动结束");  
    //System.exit(0);  
}
```



```
}
```

```
}
```

```
package com.atguigu.state;

/**
 * 不能抽奖状态
 * @author Administrator
 *
 */
public class NoRaffleState extends State {

    // 初始化时传入活动引用，扣除积分后改变其状态
    RaffleActivity activity;

    public NoRaffleState(RaffleActivity activity)
    {
        this.activity = activity;
    }

    // 当前状态可以扣积分，扣除后，将状态设置成可以抽奖状态
    @Override
    public void deductMoney() {
        System.out.println("扣除 50 积分成功，您可以抽奖了");
        activity.setState(activity.getCanRaffleState());
    }
}
```



```
// 当前状态不能抽奖
@Override
public boolean raffle() {
    System.out.println("扣了积分才能抽奖喔！");
    return false;
}
```

```
// 当前状态不能发奖品
@Override
public void dispensePrize() {
    System.out.println("不能发放奖品");
}
```

```
package com.atguigu.state;
```

```
/**
 * 抽奖活动 //
 *
 * @author Administrator
 *
 */
public class RaffleActivity {

    // state 表示活动当前的状态，是变化
    State state = null;
```



```
// 奖品数量
int count = 0;

// 四个属性， 表示四种状态
State noRaffleState = new NoRaffleState(this);
State canRaffleState = new CanRaffleState(this);

State dispenseState = new DispenseState(this);
State dispenseOutState = new DispenseOutState(this);

//构造器
//1. 初始化当前的状态为 noRaffleState (即不能抽奖的状态)
//2. 初始化奖品的数量
public RaffleActivity( int count)

    { this.state = getNoRaffleState();
      this.count = count;
    }

//扣分, 调用当前状态的 deductMoney
public void deductMoney(){

    state.deductMoney();
}

//抽奖
public void raffle(){

    // 如果当前的状态是抽奖成功
```



```
if(state.raffle()){
    //领取奖品
    state.dispensePrize();
}

}

public State getState()
{
    return state;
}

public void setState(State state)
{
    this.state = state;
}

//这里请大家注意，每领取一次奖品，count--
public int getCount() {
    int curCount = count;
    count--;
    return curCount;
}

public void setCount(int count)
{
    this.count = count;
}
```



```
public State getNoRaffleState()
{
    return noRaffleState;
}

public void setNoRaffleState(State noRaffleState)
{
    this.noRaffleState = noRaffleState;
}

public State getCanRaffleState()
{
    return canRaffleState;
}

public void setCanRaffleState(State canRaffleState)
{
    this.canRaffleState = canRaffleState;
}

public State getDispenseState()
{
    return dispenseState;
}

public void setDispenseState(State dispenseState)
{
    this.dispenseState = dispenseState;
}

public State getDispensOutState()
{
    return dispensOutState;
}
```



```
}
```

```
public void setDispensOutState(State dispenseOutState)
    { this.dispenseOutState = dispenseOutState;
    }
}
```

```
package com.atguigu.state;
```

```
/**
 * 状态抽象类
 * @author Administrator
 *
 */

```

```
public abstract class State {
```

```
    // 扣除积分 - 50
```

```
    public abstract void deductMoney();
```

```
    // 是否抽中奖品
```

```
    public abstract boolean raffle();
```

```
    // 发放奖品
```

```
    public abstract void dispensePrize();
```

{}

24.5 状态模式在实际项目-借贷平台 源码剖析

- 1) 借贷平台的订单，有审核-发布-抢单 等等 步骤，随着操作的不同，会改变订单的状态，项目中的这个模块实现就会使用到状态模式
- 2) 通常通过 if/else 判断订单的状态，从而实现不同的逻辑，伪代码如下

```
if(审核){  
    //审核逻辑  
}elseif(发布){  
    //发布逻辑  
}elseif(接单){  
    //接单逻辑  
}
```

问题分析：

这类代码难以应对变化，在添加一种状态时，我们需要手动添加if/else，在添加一种功能时，要对所有的状态进行判断。因此代码会变得越来越臃肿，并且一旦没有处理某个状态，便会发生极其严重的BUG，难以维护

- 3) 使用状态模式完成 借贷平台项目的审核模块 [设计+代码]



```
package com.atguigu.state.money;
```

```
public abstract class AbstractState implements State {
```

```
    protected static final RuntimeException EXCEPTION = new RuntimeException("操作流程不允许");
```

```
    //抽象类， 默认实现了 State 接口的所有方法
```

```
    //该类的所有方法， 其子类(具体的状态类)， 可以有选择的进行重写
```



```
@Override
```

```
public void checkEvent(Context context)  
{ throw EXCEPTION;  
}
```

```
@Override
```

```
public void checkFailEvent(Context context)  
{ throw EXCEPTION;  
}
```

```
@Override
```

```
public void makePriceEvent(Context context)  
{ throw EXCEPTION;  
}
```

```
@Override
```

```
public void acceptOrderEvent(Context context)  
{ throw EXCEPTION;  
}
```

```
@Override
```

```
public void notPeopleAcceptEvent(Context context)  
{ throw EXCEPTION;  
}
```

```
@Override
```



```
public void payOrderEvent(Context context)
    { throw EXCEPTION;
}

@Override
public void orderFailureEvent(Context context)
    { throw EXCEPTION;
}

@Override
public void feedBackEvent(Context context)
    { throw EXCEPTION;
}
}
```

```
package com.atguigu.state.money;

//各种具体状态类
class FeedBackState extends AbstractState {

    @Override
    public String getCurrentState() {
        return StateEnum.FEED_BACKED.getValue();
    }
}
```



```
class GenerateState extends AbstractState {  
  
    @Override  
    public void checkEvent(Context context)  
    { context.setState(new  
        ReviewState());  
    }  
  
    @Override  
    public void checkFailEvent(Context context)  
    { context.setState(new FeedBackState());  
    }  
  
    @Override  
    public String getCurrentState() {  
        return StateEnum.GENERATE.getValue();  
    }  
}  
  
class NotPayState extends AbstractState {  
  
    @Override  
    public void payOrderEvent(Context context)  
    { context.setState(new PaidState());  
    }  
}
```



@Override



```
public void feedBackEvent(Context context)
    { context.setState(new FeedBackState());
    }

@Override
public String getCurrentState() {
    return StateEnum.NOT_PAY.getValue();
}

}

class PaidState extends AbstractState {

    @Override
    public void feedBackEvent(Context context)
    { context.setState(new FeedBackState());
    }

    @Override
    public String getCurrentState() {
        return StateEnum.PAID.getValue();
    }
}

class PublishState extends AbstractState {

    @Override
```



```
public void acceptOrderEvent(Context context) {  
    //把当前状态设置为 NotPayState。。。  
    //至于应该变成哪个状态，有流程图来决定  
    context.setState(new NotPayState());  
}  
  
@Override  
public void notPeopleAcceptEvent(Context context)  
{ context.setState(new FeedBackState());  
}  
  
@Override  
public String getCurrentState()  
{  
    return StateEnum.PUBLISHED.getValue();  
}  
}  
  
class ReviewState extends AbstractState {  
  
    @Override  
    public void makePriceEvent(Context context)  
    { context.setState(new PublishState());  
    }  
  
    @Override  
    public String getCurrentState() {
```



```
        return StateEnum.REVIEWED.getValue();
    }

}

package com.atguigu.state.money;

/**测试类*/
public class ClientTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //创建 context 对象
        Context context = new Context();
        //将当前状态设置为 PublishState
        context.setState(new PublishState());
        System.out.println(context.getCurrentState());

        //
        //publish --> not pay
        context.acceptOrderEvent(context);
        //
        //not pay --> paid
        context.payOrderEvent(context);
        //
        // 失败, 检测失败时, 会抛出异常
        // try {
        //     context.checkFailEvent(context);
        //     System.out.println("流程正常..");
        // } catch (Exception e) {
```



```
//         // TODO: handle exception
//         System.out.println(e.getMessage());
//     }

}

package com.atguigu.state.money;

//环境上下文
public class Context extends AbstractState{
    //当前的状态 state, 根据我们的业务流程处理, 不停的变化
    private State state;

    @Override
    public void checkEvent(Context context)
    {
        state.checkEvent(this);
        getCurrentState();
    }

    @Override
    public void checkFailEvent(Context context)
    {
        state.checkFailEvent(this);
        getCurrentState();
    }
}
```



```
@Override  
public void makePriceEvent(Context context)  
{ state.makePriceEvent(this);  
    getCurrentState();  
}  
  
@Override  
public void acceptOrderEvent(Context context)  
{ state.acceptOrderEvent(this);  
    getCurrentState();  
}  
  
@Override  
public void notPeopleAcceptEvent(Context context)  
{ state.notPeopleAcceptEvent(this);  
    getCurrentState();  
}  
  
@Override  
public void payOrderEvent(Context context)  
{ state.payOrderEvent(this);  
    getCurrentState();  
}  
  
@Override
```



```
public void orderFailureEvent(Context context)
    { state.orderFailureEvent(this);
        getCurrentState();
    }

@Override
public void feedBackEvent(Context context)
    { state.feedBackEvent(this);
        getCurrentState();
    }

public State getState()
    { return state;
    }

public void setState(State state)
    { this.state = state;
    }

@Override
public String getCurrentState() {
    System.out.println("当前状态：" + state.getCurrentState());
    return state.getCurrentState();
}
```



```
package com.atguigu.state.money;

/**
 * 状态接口
 * @author Administrator
 *
 */
public interface State {

    /**
     * 电审
     */
    void checkEvent(Context context);

    /**
     * 电审失败
     */
    void checkFailEvent(Context context);

    /**
     * 定价发布
     */
    void makePriceEvent(Context context);

    /**
     * 接单
     */
}
```



```
*/  
  
void acceptOrderEvent(Context context);  
  
/**  
 * 无人接单失效  
 */  
  
void notPeopleAcceptEvent(Context context);  
  
/**  
 * 付款  
 */  
  
void payOrderEvent(Context context);  
  
/**  
 * 接单有人支付失效  
 */  
  
void orderFailureEvent(Context context);  
  
/**  
 * 反馈  
 */  
  
void feedBackEvent(Context context);  
  
String getCurrentState();  
}
```



```
package com.atguigu.state.money;

/**
 * 状态枚举类
 * @author Administrator
 *
 */
public enum StateEnum {

    //订单生成
    GENERATE(1, "GENERATE"),

    //已审核
    REVIEWED(2, "REVIEWED"),

    //已发布
    PUBLISHED(3, "PUBLISHED"),

    //待付款
    NOT_PAY(4, "NOT_PAY"),

    //已付款
    PAID(5, "PAID"),

    //已完结
}
```



```
FEED_BACKED(6, "FEED_BACKED");
```

```
private int key;
```

```
private String value;
```

```
StateEnum(int key, String value)
```

```
    { this.key = key;
```

```
    this.value = value;
```

```
}
```

```
public int getKey() {return key;}
```

```
public String getValue() {return value;}
```

```
}
```

24.6 状态模式的注意事项和细节

- 1) 代码有很强的可读性。状态模式将每个状态的行为封装到对应的一个类中
- 2) 方便维护。将容易产生问题的 if-else 语句删除了，如果把每个状态的行为都放到一个类中，每次调用方法时都要判断当前是什么状态，不但会产出很多 if-else 语句，而且容易出错
- 3) 符合“开闭原则”。容易增删状态
- 4) 会产生很多类。每个状态都要一个对应的类，当状态过多时会产生很多类，加大维护难度
- 5) 应用场景：当一个事件或者对象有很多种状态，状态之间会相互转换，对不同的状态要求有不同的行为的时候，可以考虑使用状态模式

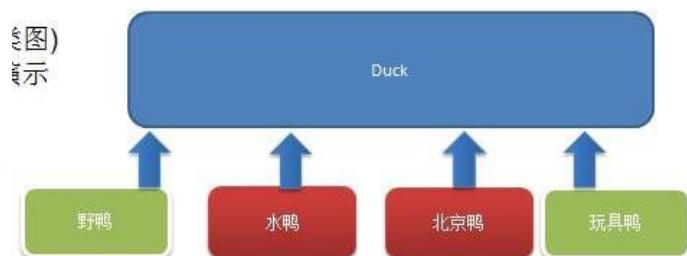
第 25 章 策略模式

25.1 编写鸭子项目，具体要求如下：

- 1) 有各种鸭子(比如野鸭、北京鸭、水鸭等， 鸭子有各种行为，比如叫、飞行等)
- 2) 显示鸭子的信息

25.2 传统方案解决鸭子问题的分析和代码实现

- 1) 传统的设计方案(类图)



- 2) 代码实现-看老师演示



```
package com.atguigu.strategy;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // 测试
    }
}
```



```
package com.atguigu.strategy;

public abstract class Duck {

    public Duck() {
    }

    public abstract void display();//显示鸭子信息

    public void quack() {
        System.out.println("鸭子嘎嘎叫~~");
    }

    public void swim() {
        System.out.println("鸭子会游泳~~");
    }

    public void fly() {
        System.out.println("鸭子会飞翔~~~");
    }
}
```

```
package com.atguigu.strategy;
```



```
public class PekingDuck extends Duck {  
  
    @Override  
    public void display() {  
        // TODO Auto-generated method stub  
        System.out.println("~~北京鸭~~~");  
    }  
  
    //因为北京鸭不能飞翔，因此需要重写 fly  
    @Override  
    public void fly() {  
        // TODO Auto-generated method stub  
        System.out.println("北京鸭不能飞翔");  
    }  
}
```

```
package com.atguigu.strategy;  
  
public class ToyDuck extends Duck{  
  
    @Override  
    public void display() {  
        // TODO Auto-generated method stub  
        System.out.println("玩具鸭");  
    }  
}
```



```
}
```

```
//需要重写父类的所有方法
```

```
public void quack() {
    System.out.println("玩具鸭不能叫~~~");
}
```

```
public void swim() {
    System.out.println("玩具鸭不会游泳~~~");
}
```

```
public void fly() {
    System.out.println("玩具鸭不会飞翔~~~");
}
}
```

```
package com.atguigu.strategy;
```

```
public class WildDuck extends Duck {

    @Override
    public void display() {
        // TODO Auto-generated method stub
        System.out.println(" 这是野鸭 ");
    }
}
```

{}

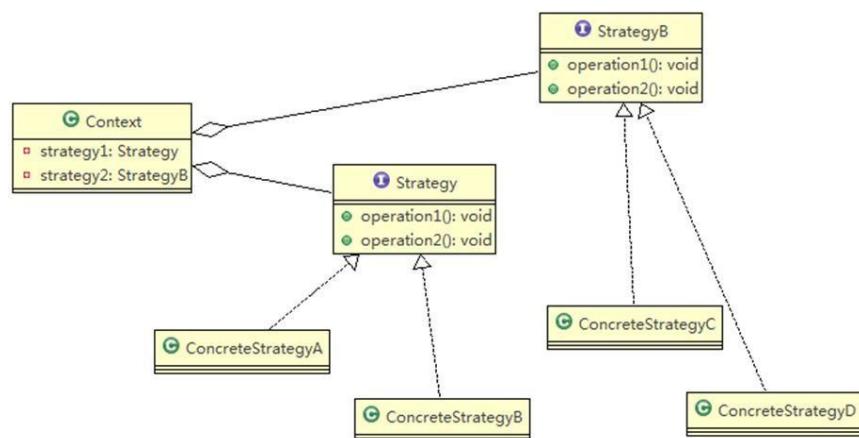
25.3 传统的方式实现的问题分析和解决方案

- 1) 其它鸭子，都继承了 Duck 类，所以 fly 让所有子类都会飞了，这是不正确的
- 2) 上面说的 1 的问题，其实是继承带来的问题：对类的局部改动，尤其超类的局部改动，会影响其他部分。会有溢出效应
- 3) 为了改进 1 问题，我们可以通过覆盖 fly 方法来解决 => 覆盖解决
- 4) 问题又来了，如果我们有一个玩具鸭子 ToyDuck，这样就需要 **ToyDuck** 去覆盖 **Duck** 的所有实现的方法 => 解决思路 -》 策略模式 (strategy pattern)

25.4 策略模式基本介绍

- 1) 策略模式 (Strategy Pattern) 中，定义算法族（策略组），分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户
- 2) 这算法体现了几个设计原则，第一、把变化的代码从不变的代码中分离出来；第二、针对接口编程而不是具体类（定义了策略接口）；第三、多用组合/聚合，少用继承（客户通过组合方式使用策略）。

25.5 策略模式的原理类图



说明：从上图可以看到，客户 context 有成员变量 strategy 或者其他的策略接口，至于需要使用到哪个策略，我们可以在构造器中指定

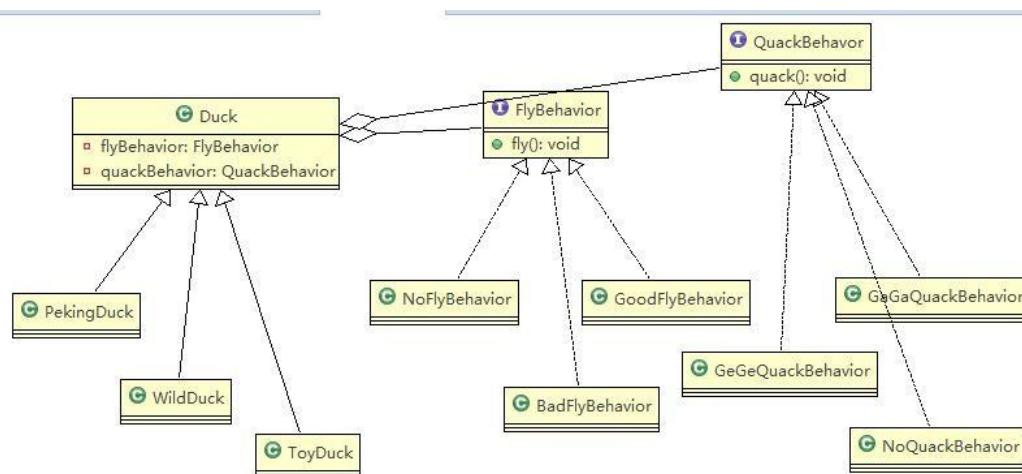
25.6 策略模式解决鸭子问题

1) 应用实例要求

编写程序完成前面的鸭子项目，要求使用策略模式

2) 思路分析(类图)

策略模式：分别封装行为接口，实现算法族，超类里放行为接口对象，在子类里具体设定行为对象。原则就是：分离变化部分，封装接口，基于接口编程各种功能。此模式让行为的变化独立于算法的使用者



3) 代码实现



```

package com.atguigu.strategy.improve;

public class BadFlyBehavior implements FlyBehavior {
}
  
```



```
@Override  
public void fly() {  
    // TODO Auto-generated method stub  
    System.out.println(" 飞翔技术一般 ");  
}  
}
```

```
package com.atguigu.strategy.improve;  
  
public class Client {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        WildDuck wildDuck = new WildDuck();  
        wildDuck.fly();  
  
        ToyDuck toyDuck = new ToyDuck();  
        toyDuck.fly();  
  
        PekingDuck pekingDuck = new PekingDuck();  
        pekingDuck.fly();  
  
        //动态改变某个对象的行为,北京鸭 不能飞  
        pekingDuck.setFlyBehavior(new NoFlyBehavior());  
        System.out.println("北京鸭的实际飞翔能力");  
    }  
}
```



```
    pekingDuck.fly();  
}  
  
}
```

```
package com.atguigu.strategy.improve;
```

```
public abstract class Duck {  
  
    //属性, 策略接口  
    FlyBehavior flyBehavior;  
    //其它属性<->策略接口  
    QuackBehavior quackBehavior;
```

```
    public Duck() {
```

```
    }
```

```
    public abstract void display();//显示鸭子信息
```

```
    public void quack() {  
        System.out.println("鸭子嘎嘎叫~~");  
    }
```

```
    public void swim() {  
        System.out.println("鸭子会游泳~~");
```



```
}
```

```
public void fly() {  
  
    //改进  
    if(flyBehavior != null)  
        { flyBehavior.fly();  
        }  
}
```

```
public void setFlyBehavior(FlyBehavior flyBehavior)  
{ this.flyBehavior = flyBehavior;  
}
```

```
public void setQuackBehavior(QuackBehavior quackBehavior)  
{ this.quackBehavior = quackBehavior;  
}
```

```
}
```

```
package com.atguigu.strategy.improve;
```

```
public interface FlyBehavior {
```



```
void fly() // 子类具体实现  
{
```

```
package com.atguigu.strategy.improve;  
  
public class GoodFlyBehavior implements FlyBehavior {  
  
    @Override  
    public void fly() {  
        // TODO Auto-generated method stub  
        System.out.println(" 飞翔技术高超 ~~~");  
    }  
}
```

```
package com.atguigu.strategy.improve;  
  
public class NoFlyBehavior implements FlyBehavior {  
  
    @Override  
    public void fly() {  
        // TODO Auto-generated method stub  
        System.out.println(" 不会飞翔 ");  
    }  
}
```



```
}
```

```
package com.atguigu.strategy.improve;

public class PekingDuck extends Duck {

    //假如北京鸭可以飞翔，但是飞翔技术一般
    public PekingDuck() {
        // TODO Auto-generated constructor stub
        flyBehavior = new BadFlyBehavior();

    }

    @Override
    public void display() {
        // TODO Auto-generated method stub
        System.out.println("~~北京鸭~~~");
    }
}
```

```
package com.atguigu.strategy.improve;
```



```
public interface QuackBehavior
{
    void quack(); //子类实现
}
```

```
package com.atguigu.strategy.improve;
```

```
public class ToyDuck extends Duck{
```

```
    public ToyDuck() {
        // TODO Auto-generated constructor stub
        flyBehavior = new NoFlyBehavior();
    }
```

```
    @Override
    public void display() {
        // TODO Auto-generated method stub
        System.out.println("玩具鸭");
    }
```

```
//需要重写父类的所有方法
```

```
    public void quack() {
        System.out.println("玩具鸭不能叫~~");
    }
```



```
public void swim() {  
    System.out.println("玩具鸭不会游泳~~");  
}
```

```
}
```

```
package com.atguigu.strategy.improve;
```

```
public class WildDuck extends Duck {
```

```
//构造器，传入 FlyBehavior 的对象
```

```
public WildDuck() {  
    // TODO Auto-generated constructor stub  
    flyBehavior = new GoodFlyBehavior();  
}
```

```
@Override
```

```
public void display() {  
    // TODO Auto-generated method stub  
    System.out.println("这是野鸭 ");  
}
```

```
}
```

25.7 策略模式在 JDK-Arrays 应用的源码分析

1) JDK 的 Arrays 的 Comparator 就使用了策略模式

2) 代码分析+Debug 源码+模式角色分析



```
package com.atguigu.jdk;

import java.util.Arrays;
import java.util.Comparator;

public class Strategy {

    public static void main(String[] args) {
```



```
// TODO Auto-generated method stub
//数组
Integer[] data = { 9, 1, 2, 8, 4, 3 };

// 实现降序排序, 返回-1 放左边, 1 放右边, 0 保持不变

// 说明
// 1. 实现了 Comparator 接口 (策略接口), 匿名类 对象 new Comparator<Integer>(){..}
// 2. 对象 new Comparator<Integer>(){..} 就是实现了 策略接口 的对象
// 3. public int compare(Integer o1, Integer o2){} 指定具体的处理方式
Comparator<Integer> comparator = new Comparator<Integer>()

{ public int compare(Integer o1, Integer o2) {

    if (o1 > o2) {

        return -1;

    } else {

        return 1;

    }

};

};

// 说明
/*
 * public static <T> void sort(T[] a, Comparator<? super T> c)
 * {
 *     if (c == null) {

 *         sort(a); //默认方法
 *     } else {

 *         if (LegacyMergeSort.userRequested)
 *             legacyMergeSort(a, c); //使用策略对象 c
 *     }
 * }
 */
```



```
        else
            // 使用策略对象 c
            TimSort.sort(a, 0, a.length, c, null, 0, 0);
    }

}

*/
//方式 1
Arrays.sort(data, comparator);

System.out.println(Arrays.toString(data)); // 降序排序

//方式 2- 同时 lambda 表达式实现 策略模式
Integer[] data2 = { 19, 11, 12, 18, 14, 13 };

Arrays.sort(data2, (var1, var2) ->
    { if(var1.compareTo(var2) > 0) {
        return -1;
    } else {
        return 1;
    }
});

System.out.println("data2=" + Arrays.toString(data2));

}
```



}

25.8 策略模式的注意事项和细节

- 1) 策略模式的关键是：分析项目中变化部分与不变部分
- 2) 策略模式的核心思想是：多用组合/聚合 少用继承；用行为类组合，而不是行为的继承。更有弹性
- 3) 体现了“对修改关闭，对扩展开放”原则，客户端增加行为不用修改原有代码，只要添加一种策略（或者行为）即可，避免了使用多重转移语句（if..else if..else）
- 4) 提供了可以替换继承关系的办法：策略模式将算法封装在独立的 Strategy 类中使得你可以独立于其 Context 改变它，使它易于切换、易于理解、易于扩展
- 5) 需要注意的是：每添加一个策略就要增加一个类，当策略过多是会导致类数目庞大

第 26 章 职责链模式

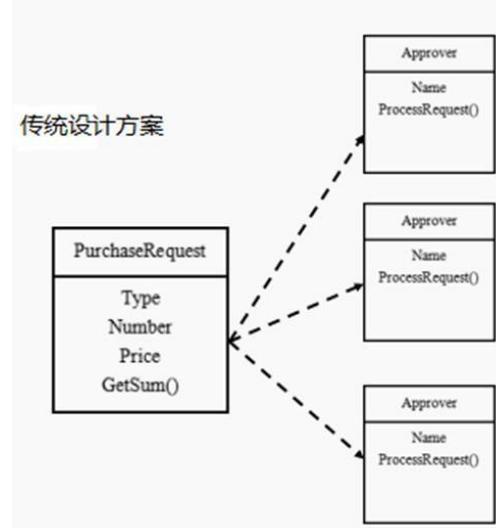
26.1 学校 OA 系统的采购审批项目：需求是

采购员采购教学器材

- 1) 如果金额 小于等于 5000, 由教学主任审批 ($0 \leq x \leq 5000$)
- 2) 如果金额 小于等于 10000, 由院长审批 ($5000 < x \leq 10000$)
- 3) 如果金额 小于等于 30000, 由副校长审批 ($10000 < x \leq 30000$)
- 4) 如果金额 超过 30000 以上, 有校长审批 ($x > 30000$)

请设计程序完成采购审批项目

26.2 传统方案解决 OA 系统审批，传统的设计方案(类图)



26.3 传统方案解决 OA 系统审批问题分析

- 1) 传统方式是：接收到一个采购请求后，根据采购金额来调用对应的 Approver (审批人)完成审批。
- 2) 传统方式的问题分析：客户端这里会使用到 分支判断(比如 switch) 来对不同的采购请求处理，这样就存在如下问题 (1) 如果各个级别的人员审批金额发生变化，在客户端的也需要变化 (2) 客户端必须明确的知道 有多少个审批级别和访问
- 3) 这样 对一个采购请求进行处理 和 Approver (审批人) 就存在强耦合关系，不利于代码的扩展和维护

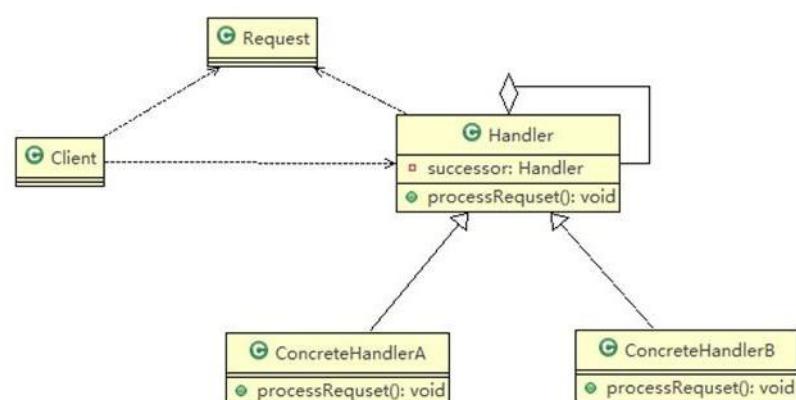
4) 解决方案 =》 职责链模式

26.4 职责链模式基本介绍

基本介绍

- 1) 职责链模式（Chain of Responsibility Pattern），又叫 责任链模式，为请求创建了一个接收者对象的链(简单示意图)。这种模式对请求的发送者和接收者进行解耦。
- 2) 职责链模式通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。
- 3) 这种类型的设计模式属于行为型模式

26.5 职责链模式的原理类图



➤ 对原理类图的说明-即(职责链模式的角色及职责)

- 1) Handler：抽象的处理器，定义了一个处理请求的接口，同时含义另外 Handler
- 2) ConcreteHandlerA , B 是具体的处理器，处理它自己负责的请求，可以访问它的后继者(即下一个处理器)，如果可以处理当前请求，则处理，否则就将该请求交给后继者去处理，从而形成一个职责链
- 3) Request ， 含义很多属性，表示一个请求

26.6 职责链模式解决 OA 系统采购审批

1) 应用实例要求

编写程序完成学校 OA 系统的采购审批项目：需求

采购员采购教学器材

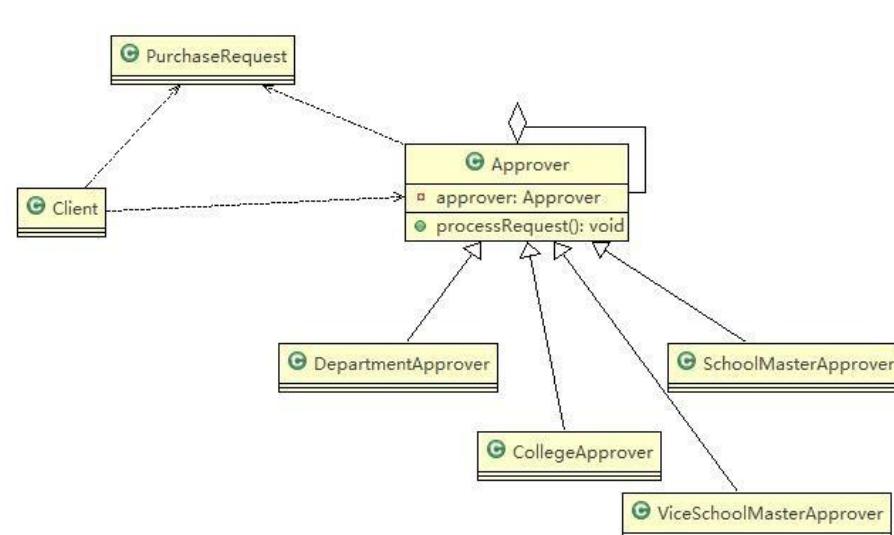
如果金额 小于等于 5000, 由教学主任审批

如果金额 小于等于 10000, 由院长审批

如果金额 小于等于 30000, 由副校长审批

如果金额 超过 30000 以上，有校长审批

2) 思路分析和图解(类图)



3) 代码实现

 responsibilitychain.zip

```
package com.atguigu.responsibilitychain;

public abstract class Approver {
```



```
Approver approver; //下一个处理器
String name; //名字

public Approver(String name) {
    // TODO Auto-generated constructor stub
    this.name = name;
}

//下一个处理器
public void setApprover(Approver approver)
{
    this.approver = approver;
}

//处理审批请求的方法，得到一个请求，处理是子类完成，因此该方法做成抽象
public abstract void processRequest(PurchaseRequest purchaseRequest);

}
```

```
package com.atguigu.responsibilitychain;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //创建一个请求
        PurchaseRequest purchaseRequest = new PurchaseRequest(1, 31000, 1);
    }
}
```



```
//创建相关的审批人
DepartmentApprover departmentApprover = new DepartmentApprover("张主任");
CollegeApprover collegeApprover = new CollegeApprover("李院长");
ViceSchoolMasterApprover viceSchoolMasterApprover = new ViceSchoolMasterApprover("王副校长");
SchoolMasterApprover schoolMasterApprover = new SchoolMasterApprover("佟校长");

//需要将各个审批级别的下一个设置好(处理人构成环形: )
departmentApprover.setApprover(collegeApprover);
collegeApprover.setApprover(viceSchoolMasterApprover);
viceSchoolMasterApprover.setApprover(schoolMasterApprover);
schoolMasterApprover.setApprover(departmentApprover);

departmentApprover.processRequest(purchaseRequest);
viceSchoolMasterApprover.processRequest(purchaseRequest);
}

}

package com.atguigu.responsibilitychain;

public class CollegeApprover extends Approver {
```



```
public CollegeApprover(String name) {  
    // TODO Auto-generated constructor stub  
    super(name);  
}  
  
@Override  
public void processRequest(PurchaseRequest purchaseRequest) {  
    // TODO Auto-generated method stub  
    if(purchaseRequest.getPrice() < 5000 && purchaseRequest.getPrice() <= 10000) {  
        System.out.println(" 请求编号 id= " + purchaseRequest.getId() + " 被 " + this.name + " 处理");  
    }else {  
        approver.processRequest(purchaseRequest);  
    }  
}
```

```
package com.atguigu.responsibilitychain;  
  
public class DepartmentApprover extends Approver {  
  
    public DepartmentApprover(String name) {  
        // TODO Auto-generated constructor stub  
        super(name);  
    }  
}
```



```
@Override  
public void processRequest(PurchaseRequest purchaseRequest) {  
    // TODO Auto-generated method stub  
    if(purchaseRequest.getPrice() <= 5000) {  
        System.out.println(" 请求编号 id= " + purchaseRequest.getId() + " 被 " + this.name + " 处理");  
    } else {  
        approver.processRequest(purchaseRequest);  
    }  
}
```

```
package com.atguigu.responsibilitychain;
```

```
//请求类  
public class PurchaseRequest {  
  
    private int type = 0; //请求类型  
    private float price = 0.0f; //请求金额  
    private int id = 0;  
    //构造器  
    public PurchaseRequest(int type, float price, int id)  
    {  
        this.type = type;  
        this.price = price;  
        this.id = id;  
    }
```



```
}
```

```
public int getType()
```

```
{ return type;
```

```
}
```

```
public float getPrice()
```

```
{ return price;
```

```
}
```

```
public int getId()
```

```
{ return id;
```

```
}
```

```
}
```

```
package com.atguigu.responsibilitychain;
```

```
public class SchoolMasterApprover extends Approver {
```

```
    public SchoolMasterApprover(String name) {
```

```
        // TODO Auto-generated constructor stub
```

```
        super(name);
```

```
    }
```



```
@Override  
public void processRequest(PurchaseRequest purchaseRequest) {  
    // TODO Auto-generated method stub  
    if(purchaseRequest.getPrice() > 30000) {  
        System.out.println(" 请求编号 id= " + purchaseRequest.getId() + " 被 " + this.name + " 处理");  
    } else {  
        approver.processRequest(purchaseRequest);  
    }  
}
```

```
package com.atguigu.responsibilitychain;  
  
public class ViceSchoolMasterApprover extends Approver {  
  
    public ViceSchoolMasterApprover(String name) {  
        // TODO Auto-generated constructor stub  
        super(name);  
    }  
  
    @Override  
    public void processRequest(PurchaseRequest purchaseRequest) {  
        // TODO Auto-generated method stub  
        if(purchaseRequest.getPrice() < 10000 && purchaseRequest.getPrice() <= 30000) {  
            System.out.println(" 请求编号 id= " + purchaseRequest.getId() + " 被 " + this.name + " 处理");  
        } else {  
        }  
    }  
}
```

```

        approver.processRequest(purchaseRequest);

    }

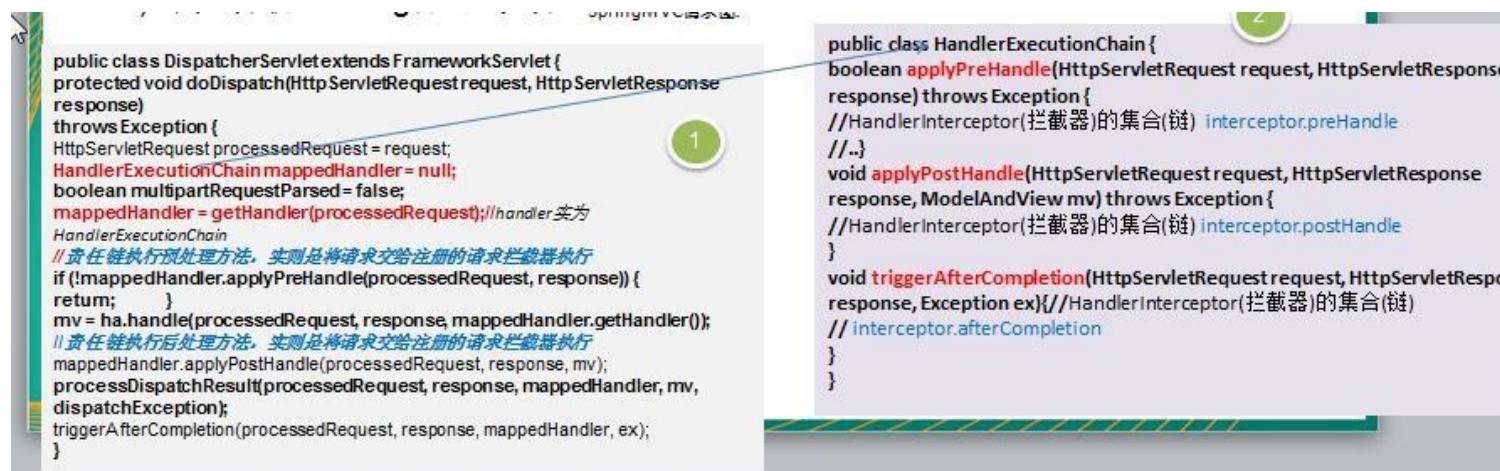
}

}

```

26.7 职责链模式在 SpringMVC 框架应用的源码分析

- 1) SpringMVC-HandlerExecutionChain 类就使用到职责链模式
- 2) SpringMVC 请求流程简图
- 3) 代码分析+Debug 源码+说明



The screenshot shows two code editors side-by-side. The left editor contains the `DispatcherServlet` class, and the right editor contains the `HandlerExecutionChain` class. A callout bubble with the number '1' points from the `mappedHandler.applyPreHandle(processedRequest, response);` line in the `DispatcherServlet` code to the `applyPreHandle` method in the `HandlerExecutionChain` class.

```

public class DispatcherServlet extends FrameworkServlet {
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response)
throws Exception {
HttpServletRequest processedRequest = request;
HandlerExecutionChain mappedHandler = null;
boolean multipartRequestParsed = false;
mappedHandler = getHandler(processedRequest); // handler 实为
HandlerExecutionChain
// 责任链执行预处理方法，实则是将请求交给注册的请求拦截器执行
if (!mappedHandler.applyPreHandle(processedRequest, response)) {
return;
}
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
// 责任链执行后处理方法，实则是将请求交给注册的请求拦截器执行
mappedHandler.applyPostHandle(processedRequest, response, mv);
processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
}
}

```

```

public class HandlerExecutionChain {
boolean applyPreHandle(HttpServletRequest request, HttpServletResponse
response) throws Exception {
// HandlerInterceptor(拦截器)的集合(链) interceptor.preHandle
//..
void applyPostHandle(HttpServletRequest request, HttpServletResponse
response, ModelAndView mv) throws Exception {
// HandlerInterceptor(拦截器)的集合(链) interceptor.postHandle
}
void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse
response, Exception ex){ // HandlerInterceptor(拦截器)的集合(链)
// interceptor.afterCompletion
}
}

```

- 4) 源码和说明

```

package com.atguigu.spring.test;

import org.springframework.web.servlet.HandlerExecutionChain;
import org.springframework.web.servlet.HandlerInterceptor;

public class ResponsibilityChain {

```



```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
  
    // DispatcherServlet  
  
    //说明  
    /*  
     *  
     * protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {  
     *     HandlerExecutionChain mappedHandler = null;  
     *     mappedHandler = getHandler(processedRequest); //获取到 HandlerExecutionChain 对象  
     *     //在 mappedHandler.applyPreHandle 内部 得到啦 HandlerInterceptor interceptor  
     *     //调用了拦截器的 interceptor.preHandle  
     *     if (!mappedHandler.applyPreHandle(processedRequest, response))  
     *         { return;  
     *     }  
  
     //说明： mappedHandler.applyPostHandle 方法内部获取到拦截器，并调用  
     //拦截器的 interceptor.postHandle(request, response, this.handler, mv);  
     mappedHandler.applyPostHandle(processedRequest, response, mv);  
    * }  
    *  
    *  
    * //说明： 在 mappedHandler.applyPreHandle 内部中，  
    * 还调用了 triggerAfterCompletion 方法，该方法中调用了
```



```
* HandlerInterceptor interceptor = getInterceptors()[i];  
try {  
    interceptor.afterCompletion(request, response, this.handler, ex);  
}  
catch (Throwable ex2) {  
    logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);  
}  
/*  
}  
  
}
```

5) 对源码总结

- springmvc 请求的流程图中，执行了 拦截器相关方法 interceptor.preHandler 等等
- 在处理 SpringMVC 请求时，使用到职责链模式还使用到适配器模式
- HandlerExecutionChain 主要负责的是请求拦截器的执行和请求处理,但是他本身不处理请求，只是将请求分配给链上注册处理器执行，这是职责链实现方式,减少职责链本身与处理逻辑之间的耦合,规范了处理流程
- HandlerExecutionChain 维护了 HandlerInterceptor 的集合，可以向其中注册相应的拦截器.

26.8 职责链模式的注意事项和细节

- 1) 将请求和处理分开，实现解耦，提高系统的灵活性
- 2) 简化了对象，使对象不需要知道链的结构
- 3) 性能会受到影响，特别是在链比较长的时候，因此需控制链中最大节点数量，一般通过在 Handler 中设置一个最大节点数量，在 setNext()方法中判断是否已经超过阈值，超过则不允许该链建立，避免出现超长链无意识地破坏系统性能



- 4) 调试不方便。采用了类似递归的方式，调试时逻辑可能比较复杂
- 5) 最佳应用场景：有多个对象可以处理同一个请求时，比如：多级请求、请假/加薪等审批流程、Java Web 中 Tomcat 对 Encoding 的处理、拦截器