

小滴课堂，愿景：让编程不在难学，让技术与生活更加有趣

相信我们，这个是可以让你学习更加轻松的平台，里面的课程绝对会让你技术不断提升

欢迎加小D讲师的微信： xdclass-lw

我们官方网站：<https://xdclass.net>

千人IT技术交流QQ群： 718617859

重点来啦：加讲师微信 免费赠送你干货文档大集合，包含前端，后端，测试，大数据，运维主流技术文档（持续更新）

<https://mp.weixin.qq.com/s/qYnjcDYGFDQorWmSfE7lpQ>



添加讲师获取课程技术答疑！

Wechat: xdclass-anna

新
客
户

下单咨询
添加微信



: xdclass-anna



旭瑶&小滴课堂 愿景："让编程不再难学，让

技术与生活更加有趣"

更多架构课程请访问 xdclass.net

第一章 旭瑶-小滴课堂架构师成长系列之 软件设计模式课程介绍

第1集 架构师成长系列-软件设计模式课程介绍

简介：讲解软件设计模式课程介绍，课程适合人员和学后水平

- 设计模式课程介绍

从0到1讲解软件开发的设计模式，20多种设计模式详解

不止简单掌握还有多个需求案例实战，更好的实践设计模式的思想，中间穿插JDK、框架源码里面设计模式的应用。

真正提升写代码+架构的内功

- 为什么要学为设计模式课程

- 公司项目需要重构，但是不知道如何下手，做到高内聚低耦合
- 为了装下武林高手，面试应对面试官的各种刁难（看招聘要求）

- 为了写些高大上的代码，让新手看不懂。或者为了看懂高手写的代码
- 为了提升自己内力，更好的理解框架源码设计思想，封装中间件
- 让的代码更好重用、可读、可靠、可维护、可拓展



别嫌我啰嗦！

- 课程学前基础
 - 掌握 java/python/js/c/c++/go 其中一门语言
 - 课程基于java语言实现，重点是掌握其思想，语言只是实现工具
 - PS:不会上面的基础也没关系,这些基础都有课程，联系我们客服即可，且是刚录制的新版课程

- 适合人群 Chief Technology Officer 首席技术官

- 中高级 客户端、前端、后端工程师、项目经理、CTO 必备知识

- 从传统软件公司过渡到互联网公司的人员
- 只要是软件开发人员，都建议学这个课程，提升内力

- 学后水平

- 掌握软件设计的六大原则 **SOLID**
- 掌握 **创建型模式** 工厂模式、抽象工厂模式、单例模式、建造者模式、原型模式
- 掌握 **结构型模式** 适配器模式、桥接模式、装饰器模式、代理模式、组合模式、外观模式、享元模式
- 掌握 **行为型模式** 责任链模式、迭代器模式、观察者模式、状态模式、策略模式、模板模式、备忘录模式、命令模式等
- 掌握20多种设计模式的应用场景、优点、缺点和需求案例实战
- 掌握多个源码里面设计模式的应用和面试题

- 课程技术技术栈和环境说明

- JDK11 + IDEA旗舰版

- 学习形式

- 视频讲解 + 文字笔记 + 原理分析 + 交互流程图
 - 配套源码 + 笔记 + 技术群答疑(我答疑，联系客服进群)

- 课程有配套的源码，在每章-每集的资料里面，如果没有用到代码就不保存
- 设计模式课程重点在于理解，会比较枯燥，我这边会尽量用生动趣味方式给大家讲



第2集 设计模式全家桶课程大纲速览

简介：讲解设计模式全家桶课程大纲和效果演示

- 课程案例+效果图演示
- 目录大纲浏览
- 学习要求
 - 保持谦虚好学、术业有专攻、在校的学生，有些知识掌握也比工作好几年掌握的人厉害
 - 如果自己操作的情况和视频不一样，导入课程代码对比验证基本就可以发现问题了
 - 设计模式比较抽象-枯燥，是提升内力的关键内容，多看几遍，结合网上的博文。

大哥，火





技术与生活更加有趣" 更多架构课程请访问 xdclass.net

第二章 想成为架构师的你，不可不知道的设计模式精髓

第1集 设计模式的六大原则你知道多少

简介：讲解设计模式的六大设计原则

- 设计模式是站在设计原则的基础之上的，所以在学习设计模式之前，有必要对这些设计原则先做一下了解
- 软件设计开发原则
 - 为了让的代码更好重用性，可读性，可靠性，可维护性
 - 诞生出了很多软件设计的原则，这6大设计原则是我们要掌握的
 - 将六大原则的英文首字母拼在一起就是SOLID（稳定的），所以也称之为SOLID原则



一开始是你们说要这样做的，现在错了要我背锅

- 单一职责原则 RSP Single Responsibility Principle

- 一个类只负责一个功能领域中的相应职责，就一个类而言，应该只有一个引起它变化的原因
- 是实现高内聚、低耦合的指导方针
- 解释：
 - 高内聚
 - 尽可能让类的每个成员方法只完成一件事（最大限度的聚合）
 - 模块内部的代码，相互之间的联系越强，内聚就越高，模块的独立性就越好
 - 低耦合：减少类内部，一个成员方法调用另一个成员方法，不要有牵一发动全身

- 开闭原则 OCP Open-Closed Principle

- 对扩展开放，对修改关闭，在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果

- 里氏替换原则LSP Liskov Substitution Principle
 - 任何基类可以出现的地方，子类一定可以出现
 - 在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象
 - controller->service->dao
- 依赖倒转原则 DIP Dependency Inversion Principle
 - 是开闭原则的基础，针对接口编程，依赖于抽象而不依赖于具体
 - 高层模块不应该依赖低层模块，二者都应该依赖其抽象
- 接口隔离原则 ISP Interface Segregation Principle
 - 客户端不应该依赖那些它不需要的接口
 - 使用多个隔离的接口，比使用单个接口要好，降低类之间的耦合度
- 迪米特法则 LoD Lay of Demeter
 - 最少知道原则，一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立
 - 类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及
 - 通过引入一个合理的第三者来降低现有对象之间的耦合

度

第2集 大佬们常说的设计模式到底是什么

简介：讲解设计模式到底是什么

- 设计模式简介
 - 由来：是软件开发人员在软件开发过程中面临的一般问题的解决方案。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
 - 好处：为了重用代码、让代码更容易被他人理解、保证代码可靠性
 - 坏处：对不熟悉设计模式的同学，看起来更绕更难理解

总是套路得人心



- 什么是GOF (Gang of Four)

在 1994 年，由 四位作者合称 GOF (全拼 Gang of Four) 四人合著出版了一本名为 Design Patterns - Elements of Reusable Object-Oriented Software. 他们所提出的设计模式主要是基于以下的面向对象设计原则。

- 1) 对接口编程而不是对实现编程。
- 2) 优先使用对象组合而不是继承

- 常见的三大设计模式分类

- 创建型模式
- 提供了一种在创建对象的同时隐藏创建逻辑的方式，使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活

常用：工厂模式、抽象工厂模式、单例模式、建造者模式

不常用：原型模式

- 结构型模式
- 关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式

常用：适配器模式、桥接模式、装饰器模式、代理模式

不常用：组合模式、外观模式、享元模式、

- 行为型模式
- 特别关注对象之间的通信

常用：责任链模式、迭代器模式、观察者模式、状态模式、策略模式、模板模式

不常用：备忘录模式、命令模式

几乎不用：访问者模式、中介者模式、解释器模式

第3集 多个业务场景浏览-设计模式使用前后的区别

简介：多个案例-使用设计模式前后的区别

- 多个案例浏览-使用设计模式前后的区别
- 部分设计模式你已经熟练使用了



旭瑶&小滴课堂 愿景："让编程不再难学，让

技术与生活更加有趣" 更多架构课程请访问 xdclass.net

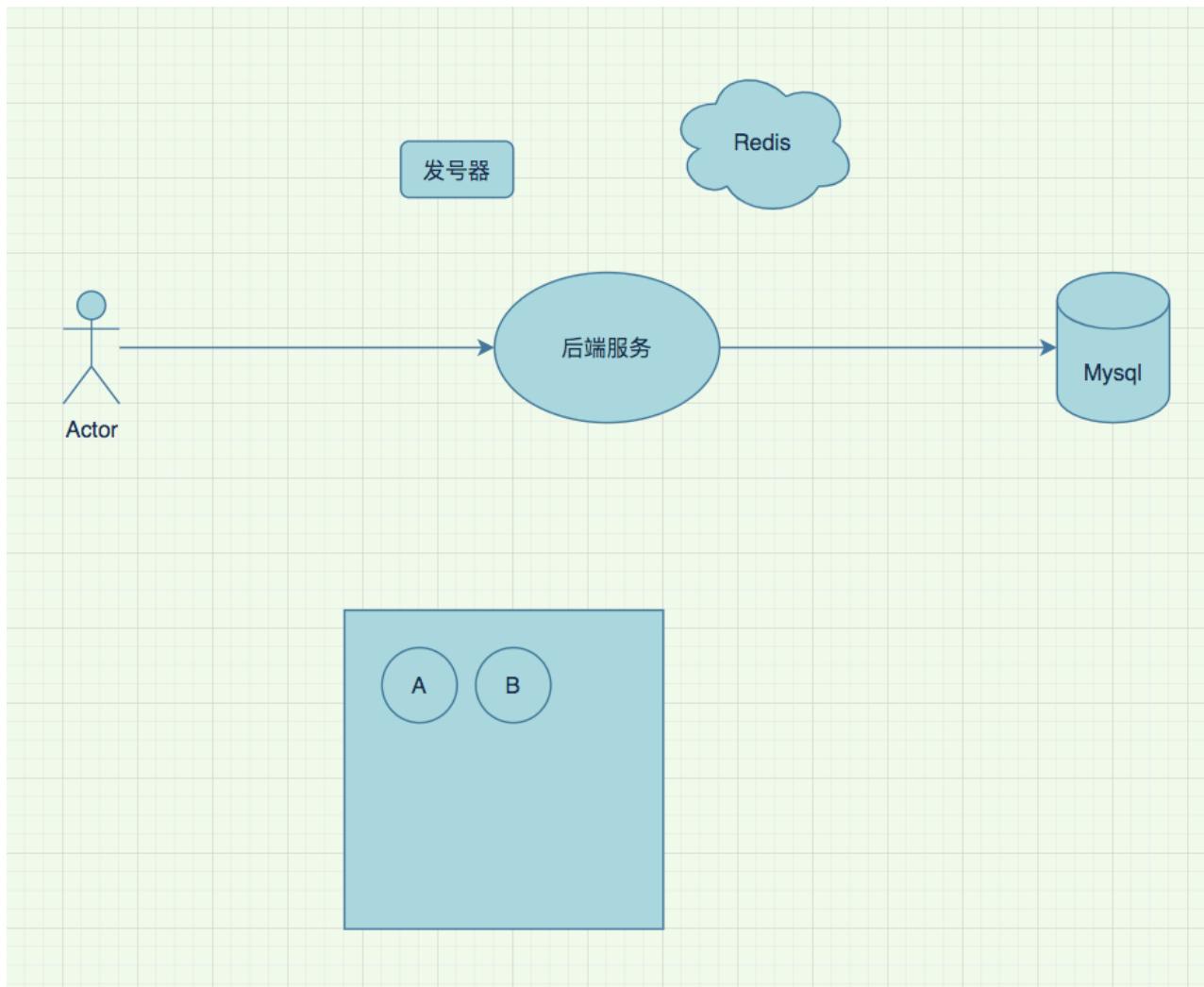
第三章 创建型设计模式-单例设计模式和应用

第1集 江湖传言里的设计模式-单例设计模式

简介：什么是单例设计模式和应用

- 备注：面试重点考查
- 单例设计模式：
 - 这个是最简单的设计模式，所以拎出来第一个讲，但事实却不是。
 - 单例意思只包含一个对象被称为单例的特殊类
 - 通过单例模式可以保证系统中，应用该模式的类只有一个对象实例
- 使用场景
 - 业务系统全局只需要一个对象实例，比如发号器、redis连接对象等
 - Spring IOC容器中的bean默认就是单例
 - spring boot 中的controller、service、dao层中通过@autowire的依赖注入对象默认都是单例的
- 分类：

- 懒汉：就是所谓的懒加载，延迟创建对象
 - 饿汉：与懒汉相反，提前创建对象
- 实现步骤
 - 私有化构造函数
 - 提供获取单例的方法



第2集 代码实战-单例设计模式中的懒汉实现方式

简介：单例设计模式中的懒汉方式实战

- 你知道单例设计模式-懒汉方式有几种实现方式

第3集 单例模式中的懒汉实现+双重检查锁定+内存模型

简介：单例设计模式中的懒汉方式实战

- 改进实现方式

```
public class SingletonLazy {  
  
    //private static SingletonLazy instance;  
    /**  
     * 构造函数私有化  
     */  
    private SingletonLazy(){}  
  
    /**  
     * 单例对象的方法  
     */  
    public void process(){  
        System.out.println("方法调用成功");  
    }  
}
```

```
/**  
 * 第一种方式  
 * 对外暴露一个方法获取类的对象  
 *  
 * 线程不安全，多线程下存在安全问题  
 *  
 */  
  
//    public static SingletonLazy getInstance()  
{  
//        if(instance == null){  
//            instance = new SingletonLazy();  
//        }  
//        return instance;  
//    }  
  
  
/**  
 * 第二种实现方式  
 * 通过加锁 synchronized 保证单例  
 *  
 * 采用synchronized 对方法加锁有很大的性能开销  
 *  
 * 解决办法：锁粒度不要这么大  
 *  
 * @return  
 */  
  
//    public static synchronized SingletonLazy  
getInstance(){  
//        if(instance == null){
```

```
//           instance = new SingletonLazy();
//       }
//       return instance;
//   }

/*
 * 第三种实现方式
 *
 * DCL 双重检查锁定 (Double-Checked-Locking) ,在多线程情况下保持高性能
 *
 * 这是否安全, instance = new
SingletonLazy(); 并不是原子性操作
 * 1、分配空间给对象
 * 2、在空间内创建对象
 * 3、将对象赋值给引用instance
 *
 * 假如线程 1-》 3-》 2顺序, 会把值写会主内存, 其他
线程就会读取到instance最新的值, 但是这个是不完全的对象
 * (指令重排)
 * @return
 */
public static SingletonLazy
getInstance(){
//     if(instance == null){
//         // A、B
//         synchronized
(SingletonLazy.class){

```

```
//                     if(instance == null) {
//                         instance = new
SingletonLazy();
//                     }
//                 }
//             }
//             return instance;
//         }

/**
 * volatile是Java提供的关键字，它具有可见性和有序性，
 *
 * 指令重排序是JVM对语句执行的优化，只要语句间没有依赖，那JVM就有权对语句进行优化
 *
 * 禁止了指令重排
 */
private static volatile SingletonLazy
instance;
public static SingletonLazy getInstance(){
    //第一重检查
    if(instance == null){
        // A、B，锁定
        synchronized (SingletonLazy.class){
            //第二重检查
            if(instance == null) {
```

```
        instance = new
SingletonLazy();
    }
}

return instance;
}

}
```

第4集 单例模式中的饿汉实现和选择问题

简介：单例设计模式中的饿汉方式实战

- 饿汉方式：提前创建好对象
- 优点：实现简单，没有多线程同步问题
- 缺点：不管有没有使用，`instance`对象一直占着这段内存
- 如何选择：
 - 如果对象不大，且创建不复杂，直接用饿汉的方式即可
 - 其他情况则采用懒汉实现方式

第5集 JDK源码里面的单例设计模式

简介：单例设计模式在JDK源码里面的应用

- JDK中Runtime类 饿汉方式

```
1  /*
2  * Every Java application has a single instance of class
3  * {@code Runtime} that allows the application to interface with
4  * the environment in which the application is running. The current
5  * runtime can be obtained from the {@code getRuntime} method.
6  * <p>
7  * An application cannot create its own instance of this class.
8  *
9  * @author unscrubed
10 * @see java.lang.Runtime#getRuntime()
11 * @since 1.0
12 */
13
14 package java.lang; ← Red arrow here
15
16 import ...
17
18 /**
19  * Returns the runtime object associated with the current Java application.
20  * Most of the methods of class {@code Runtime} are instance
21  * methods and must be invoked with respect to the current runtime object.
22  *
23  * @return the {@code Runtime} object associated with the current
24  * Java application.
25  */
26
27 public class Runtime { ← Red arrow here
28     private static final Runtime currentRuntime = new Runtime();
29
30     private static Version version;
31
32     /**
33      * Returns the runtime object associated with the current Java application.
34      * Most of the methods of class {@code Runtime} are instance
35      * methods and must be invoked with respect to the current runtime object.
36      *
37      * @return the {@code Runtime} object associated with the current
38      * Java application.
39      */
40
41     public static Runtime getRuntime() {
42         return currentRuntime;
43     }
44
45     /**
46      * Don't let anyone else instantiate this class
47      */
48     private Runtime() {}
49
50 }
```

- JDK中Desktop类 懒汉方式

```
25
26 package java.awt; ← Red arrow here
27 import ...
28 ...
29 /**
30 * The Desktop class allows interact with various desktop capabilities.
31 *
32 * <p> Supported operations include:
33 * <ul>
34 * <li>launching the user-default browser to show a specified
35 * URL</li>
36 * <li>launch the user-default mail client with an optional
37 * @code mailto URI</li>
38 * <li>launching a registered application to open, edit or print a
39 * specified file.</li>
40 * </ul>
41 *
42 * <p>This class provides methods corresponding to these
43 * operations. The methods look for the associated application
44 * registered on the current platform, and launch it to handle a URI
45 * or file. If there is no associated application or the associated
46 * application fails to be launched, an exception is thrown.
47 *
48 * Please see @link Desktop.Action for the full list of supported operations
49 * and capabilities.
50 *
51 * <p> An application is registered to a URI or file type.
52 * The mechanism of registering, accessing, and
53 * launching the associated application is platform-dependent.
54 *
55 * <p> Each operation is an action type represented by the @link
56 \* Desktop.Action class.
57 *
58 * <p> Note: when some action is invoked and the associated
59 * application is executed, it will be executed on the same system as
60 * the one on which the Java application was launched.
61 *
62 * @see Action
63 * @since 1.6
64 * @author Armin Chen
65 * @author George Zhang
66 */
67 public class Desktop {
68 }
69 
```



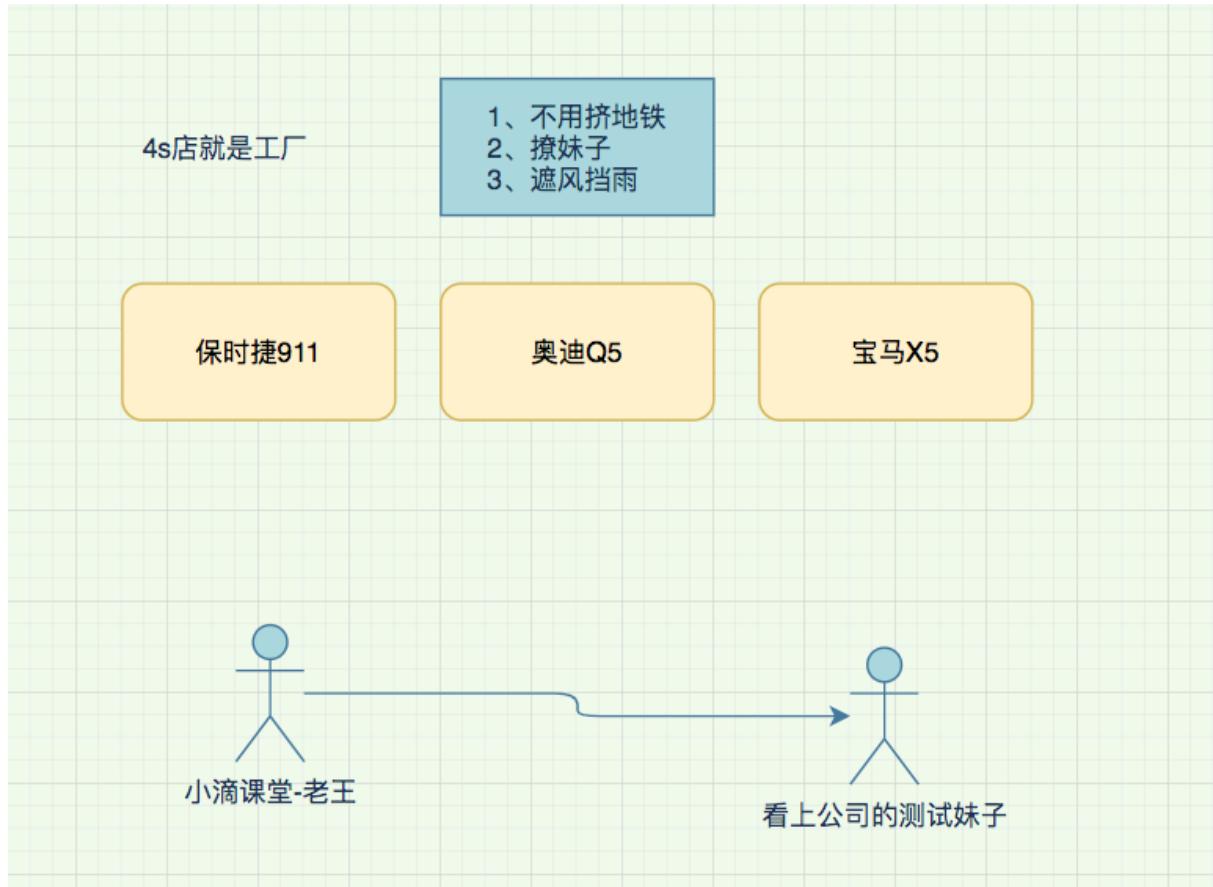
技术与生活更加有趣"更多架构课程请访问 xdclass.net

第四章 创建型设计模式-工厂模式和应用

第1集 原生社会过渡-常见的工厂设计模式

简介：介绍常见的工厂设计模式和应用

- 工厂模式介绍：
 - 它提供了一种创建对象的最佳方式，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象
- 例子：
 - 需要购买一辆车，不用管车辆如何组装，且可以购买不同类型的比如轿车、SUV、跑车，直接去4s店购买就行（4s店就是工厂）
 - 工厂生产电脑，除了A品牌、还可以生产B、C、D品牌电脑
 - 业务开发中，支付很常见，里面有统一下单和支付接口，具体的支付实现可以微信、支付宝、银行卡等



- 工厂模式有 3 种不同的实现方式
 - 简单工厂模式：通过传入相关的类型来返回相应的类，这种方式比较单一，可扩展性相对较差；
 - 工厂方法模式：通过实现类实现相应的方法来决定相应的返回结果，这种方式的可扩展性比较强；
 - 抽象工厂模式：基于上述两种模式的拓展，且支持细化产品
- 应用场景：
 - 解耦：分离职责，把复杂对象的创建和使用的过程分开
 - 复用代码 降低维护成本：
 - 如果对象创建复杂且多处需用到，如果每处都进行

编写，则很多重复代码，如果业务逻辑发生了改变，需用四处修改；

- 使用工厂模式统一创建，则只要修改工厂类即可，降低成本

第2集 电商支付应用案例-简单工厂模式实践指南

简介：电商支付里面的案例-简单工厂模式的实践指南和应用

- 简单工厂模式
 - 又称静态工厂方法，可以根据参数的不同返回不同类的实例，专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类
 - 由于工厂方法是静态方法，可通过类名直接调用，而且只需要传入简单的参数即可
- 核心组成
 - Factory：工厂类，简单工厂模式的核心，它负责实现创建所有实例的内部逻辑
 - IProduct：抽象产品类，简单工厂模式所创建的所有对象的父类，描述所有实例所共有的公共接口
 - Product：具体产品类，是简单工厂模式的创建目标
- 实现步骤
 - 创建抽象产品类，里面有产品的抽象方法，由具体的品类去实现
 - 创建具体产品类，继承了他们的父类，并实现具体方法
 - 创建工厂类，提供了一个静态方法createXXX用来生产

产品，只需要传入你想产品名称

- 优点：
 - 将对象的创建和对象本身业务处理分离可以降低系统的耦合度，使得两者修改起来都相对容易。
- 缺点
 - 工厂类的职责相对过重，增加新的产品需要修改工厂类的判断逻辑，这一点与开闭原则是相违背
 - 即开闭原则（Open Close Principle）对扩展开放，对修改关闭，程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果
 - 将会增加系统中类的个数，在一定程度上增加了系统的复杂度和理解难度，不利于系统的扩展和维护，创建简单对象就不用模式

第3集 工厂设计模式实践指南-工厂方法模式 《上》

简介：工厂方法模式的实践指南和应用《上》

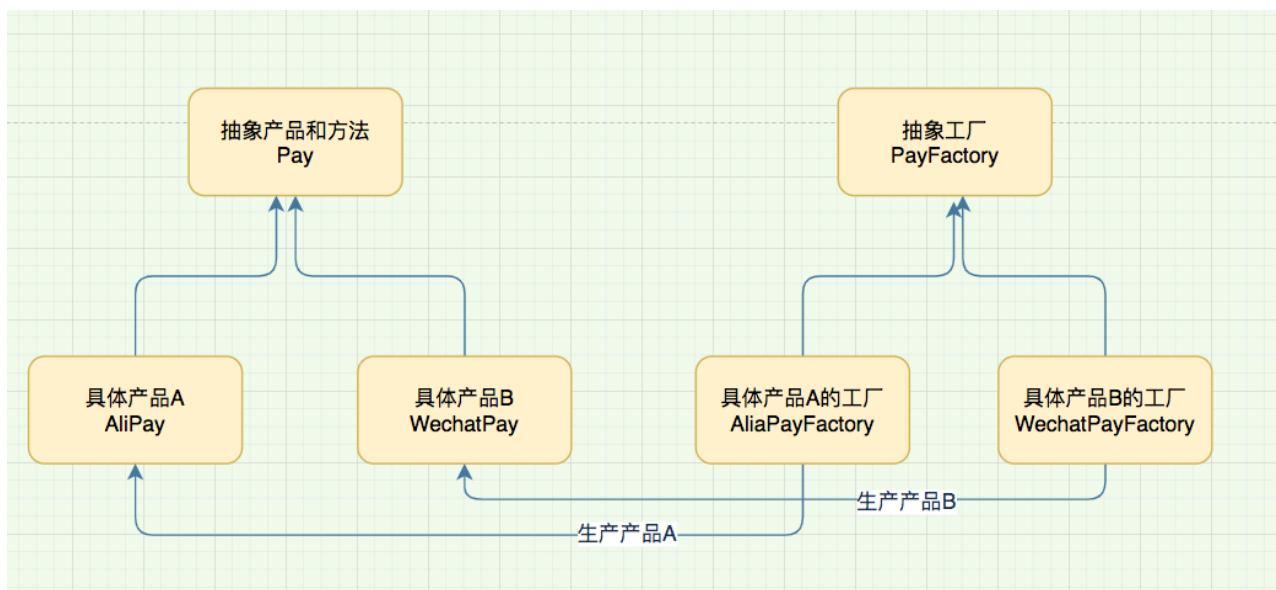
- 工厂方法模式
 - 又称工厂模式，是对简单工厂模式的进一步抽象化，其好处是可以使系统在不修改原来代码的情况下引进新的产品，即满足开闭原则
 - 通过工厂父类定义负责创建产品的公共接口，通过子类来确定所需要创建的类型
 - 相比简单工厂而言，此种方法具有更多的可扩展性和复

用性，同时也增强了代码的可读性

- 将类的实例化（具体产品的创建）延迟到工厂类的子类（具体工厂）中完成，即由子类来决定应该实例化哪一个类。

- 核心组成

- IProduct：抽象产品类，描述所有实例所共有的公共接口
- Product：具体产品类，实现抽象产品类的接口，工厂类创建对象，如果有多个需要定义多个
- IFactory：抽象工厂类，描述具体工厂的公共接口
- Factory：具体工场类，实现创建工作类对象，实现抽象工厂类的接口，如果有多个需要定义多个



第4集 工厂设计模式实践指南-工厂方法模式 《下》

简介：工厂方法模式的实践指南和应用《下》

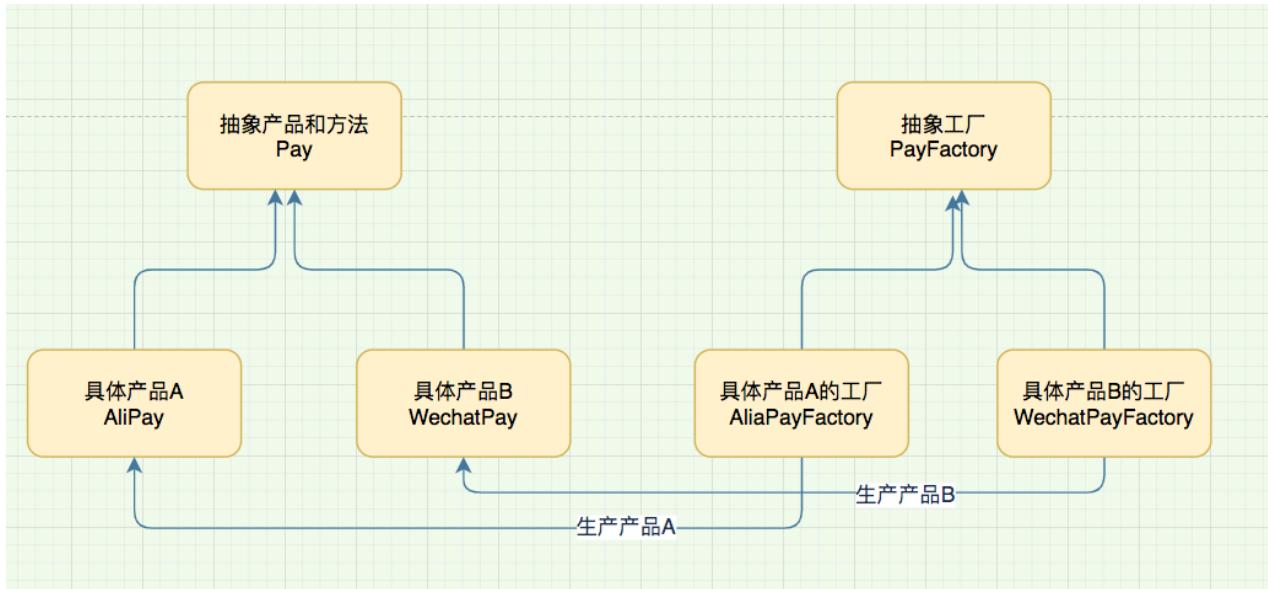
- 编码实践

```
/**
 * 抽象工厂方法
 */
public interface PayFactory {
    Pay getPay();
}

/**
 * 具体产品工厂
 */

```

```
public class AliPayFactory implements  
PayFactory {  
    @Override  
    public Pay getPay() {  
        return new AliPay();  
    }  
}  
  
/**  
 * 抽象产品  
 */  
public interface Pay {  
    /**  
     * 统一下单  
     */  
    void unifiedorder();  
}  
  
/**  
 * 具体产品  
 */  
public class AliPay implements Pay {  
  
    @Override  
    public void unifiedorder() {  
        System.out.println("支付宝支付 统一下单接  
口");  
    }  
}
```



- 优点：

- 符合开闭原则，增加一个产品类，只需要实现其他具体的产品类和具体的工厂类；
- 符合单一职责原则，每个工厂只负责生产对应的产品
- 使用者只需要知道产品的抽象类，无须关心其他实现类，满足迪米特法则、依赖倒置原则和里氏替换原则
 - 迪米特法则：最少知道原则，实体应当尽量少地与其他实体之间发生相互作用
 - 依赖倒置原则：针对接口编程，依赖于抽象而不依赖于具体
 - 里氏替换原则：俗称LSP, 任何基类可以出现的地方，子类一定可以出现，对实现抽象化的具体步骤的规范

- 缺点：

- 增加一个产品，需要实现对应的具体工厂类和具体产品类；

- 每个产品需要有对应的具体工厂和具体产品类

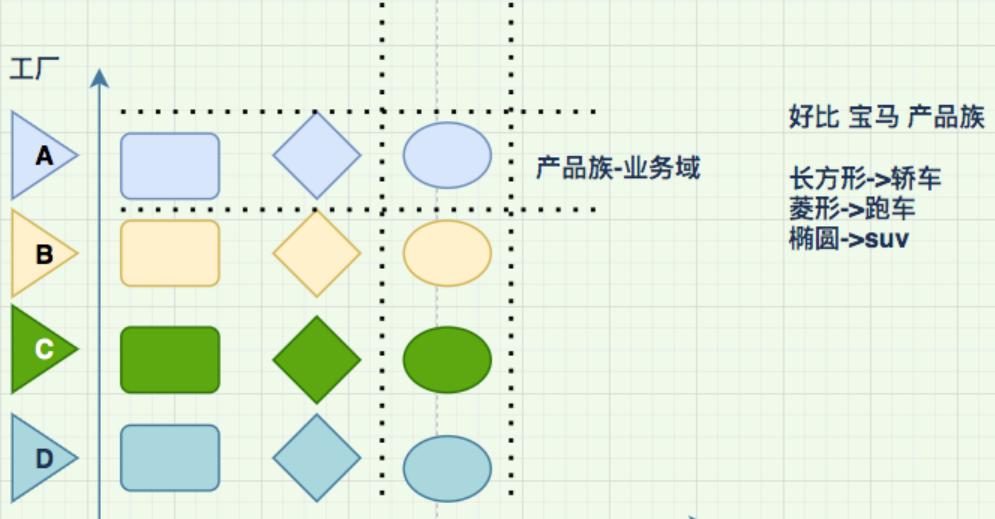
第5集 工厂设计模式实践指南- 抽象工厂方法模式《上》

简介：抽象工厂方法模式的实践指南和应用《上》

- 工厂模式有 3 种不同的实现方式
 - 简单工厂模式：通过传入相关的类型来返回相应的类,这种方式比较单一,可扩展性相对较差;

- 工厂方法模式：通过实现类实现相应的方法来决定相应的返回结果,这种方式的可扩展性比较强；
 - **抽象工厂模式：**基于上述两种模式的拓展，是工厂方法模式的升级版，当需要创建的产品有多个产品线时使用**抽象工厂模式是比较好的选择**
 - 抽象工厂模式在 Spring 中应用得最为广泛的一种设计模式
-
- 背景
 - 工厂方法模式引入工厂等级结构，解决了简单工厂模式中工厂类职责过重的问题
 - 但工厂方法模式中每个工厂只创建一类具体类的对象，后续发展可能会导致工厂类过多，因此将一些相关的具体类组成一个“具体类族”，由同一个工厂来统一生产，强调的是一系列相关的产品对象！！！

产品等级结构 好比 奔驰跑车、奥迪跑车、宝马跑车等其他品牌



订单功能：不仅包括统一下单Pay，还有退款、提现等产品族

- 实现步骤

1、定义两个接口 Pay、Refund

2、创建具体的Pay产品、创建具体的Refund产品

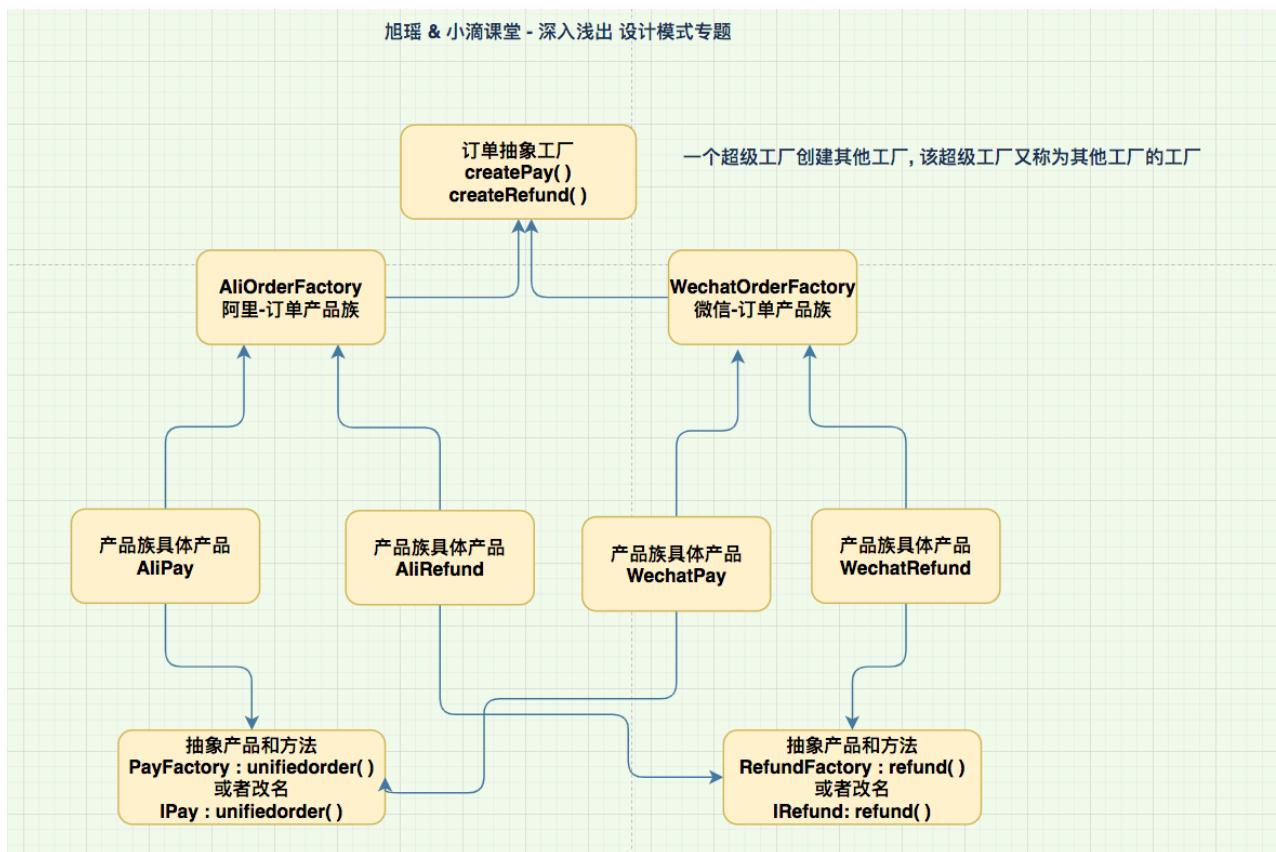
3、创建抽象工厂 OrderFactory 接口

里面两个方法 createPay/createRefund

4、创建支付宝产品族AliOrderFactory，实现OrderFactory 抽象工厂

5、创建微信支付产品族WechatOrderFactory，实现 OrderFactory 抽象工厂

6、定义一个超级工厂创造器，通过传递参数获取对应的工厂



- 防止有干扰，注释掉原先的类

第6集 工厂设计模式实践指南- 抽象工厂方法模式《下》

简介：抽象工厂方法模式的实践指南和应用《下》

- 编码实践

```
/**
 * 超级工厂，定义同个产品族的其他相关子工厂
 */
public interface OrderFactory {
    PayFactory createPay();
    RefundFactory createRefund();
}

//产品族工厂的产品，可以不叫Factory，看公司团队规范，比如类名叫 IPay 也可以的
public interface PayFactory {
    /**
     * 统一下单
     */
    void unifiedorder();
}
```

```
}
```

```
//产品族工厂
```

```
public class AliOrderFactory implements OrderFactory {
```

```
    @Override
```

```
        public PayFactory createPay() {
```

```
            return new AliPay();
```

```
        }
```

```
    @Override
```

```
        public RefundFactory createRefund() {
```

```
            return new AliRefund();
```

```
        }
```

```
}
```

```
//具体产品
```

```
public class AliPay implements PayFactory {
```

```
    @Override
```

```
        public void unifiedorder() {
```

```
            System.out.println("支付宝支付 统一下单接口");
```

```
        }
```

```
}
```

```
//超级工厂生产器，传参生产对应的子工厂
```

```
public class FactoryProducer {
```

```
public static OrderFactory  
getFactory(String type) {  
    if (type.equalsIgnoreCase("WECHAT")) {  
        return new WechatOrderFactory();  
    } else if  
(type.equalsIgnoreCase("ALI")) {  
        return new AliOrderFactory();  
    }  
    return null;  
}  
  
}  
  
//Main函数使用  
OrderFactory orderFactory =  
FactoryProducer.getFactory("ALI");  
orderFactory.createPay().unifiedorder();  
orderFactory.createRefund().refund();
```

- 工厂方法模式和抽象工厂方法模式
 - 当抽象工厂模式中每一个具体工厂类只创建一个产品对象，抽象工厂模式退化成工厂方法模式
- 优点
 - 当一个产品族中的多个对象被设计成一起工作时，它能保证使用方始终只使用同一个产品族中的对象

- 产品等级结构扩展容易，如果需要增加多一个产品等级，只需要增加新的工厂类和产品类即可，比如增加银行支付、退款
- 缺点
 - 产品族扩展困难，要增加一个系列的某一产品，既要在抽象的工厂和抽象产品里修改代码，不是很符合开闭原则
 - 增加了系统的抽象性和理解难度



旭瑤&小滴课堂 愿景："让编程不再难学，让

技术与生活更加有趣" 更多架构课程请访问 xdclass.net

第五章 创建型设计模式 建造者+原型模式的应用场景

第1集 创建型设计模式-Prototype原型设计模式 实战《上》

简介：讲解原型设计模式介绍和应用场景《上》

- 原型设计模式Prototype

- 是一种对象创建型模式，使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，主要用于创建重复的对象，同时又能保证性能
- 工作原理是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自己来实现创建过程
- 应该是最简单的设计模式了，实现一个接口，重写一个方法即完成了原型模式

- 核心组成

- **Prototype**: 声明克隆方法的接口,是所有具体原型类的公共父类，Cloneable接口
- **ConcretePrototype** : 具体原型类
- **Client**: 让一个原型对象克隆自身从而创建一个新的对象

- 应用场景

- 创建新对象成本较大，新的对象可以通过原型模式对已

有对象进行复制来获得

- 如果系统要保存对象的状态，做备份使用

第2集 创建型设计模式-Prototype原型设计模式 实战《下》

简介：讲解原型设计模式介绍和应用场景《下》

- 遗留问题：

- 通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的
- 浅拷贝实现 Cloneable，深拷贝是通过实现 Serializable 读取二进制流

- 拓展

- 浅拷贝

如果原型对象的成员变量是基本数据类型（int、double、byte、boolean、char等），将复制一份给克隆对象；

如果原型对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，也就是说原型对象和克隆对象的成员变量指向相同的内存地址

通过覆盖Object类的clone()方法可以实现浅克隆

- 深拷贝

无论原型对象的成员变量是基本数据类型还是引用类型，都将复制一份给克隆对象，如果需要实现深克隆，可以通过序列化(Serializable)等方式来实现

- 原型模式是内存二进制流的拷贝，比new对象性能高很多，使用的时候记得注意是选择浅拷贝还是深拷贝

- 优点

- 当创建新的对象实例较为复杂时，使用原型模式可以简化对象的创建过程，可以提高新实例的创建效率
 - 可辅助实现撤销操作，使用深克隆的方式保存对象的状态，使用原型模式将对象复制一份并将其状态保存起来，以便在需要的时候使用恢复到历史状态

- 缺点

- 需要为每一个类配备一个克隆方法，对已有的类进行改造时，需要修改源代码，违背了“开闭原则”
- 在实现深克隆时需要编写较为复杂的代码，且当对象之间存在多重的嵌套引用时，需要对每一层对象对应的类都必须支持深克隆

```
public class Person implements Cloneable,  
Serializable {  
  
    private String name;  
  
    private int age;  
  
    private List<String> list = new ArrayList<>()  
();  
  
    public List<String> getList() {  
        return list;  
    }  
  
    public void setList(List<String> list) {  
        this.list = list;  
    }  
  
    public Person() {  
        System.out.println("构造函数调用");  
    }  
}
```

```
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

@Override
protected Person clone() throws
CloneNotSupportedException {
    return (Person) super.clone();
}

/**
 * 深拷贝
 * @return
 */
public Object deepClone() {
```

```
try {
    //输出 序列化
    ByteArrayOutputStream baos = new
ByteArrayOutputStream();
    ObjectOutputStream oos = new
ObjectOutputStream(baos);
    oos.writeObject(this);

    //输入 反序列化
    ByteArrayInputStream bais = new
ByteArrayInputStream(baos.toByteArray());
    ObjectInputStream ois = new
ObjectInputStream(bais);
    Person copyObj = (Person)
ois.readObject();

    return copyObj;
} catch (Exception e) {
    e.printStackTrace();
    return null;
}
}

}
```

第3集 创建型设计模式-建造者模式应用介绍

《上》

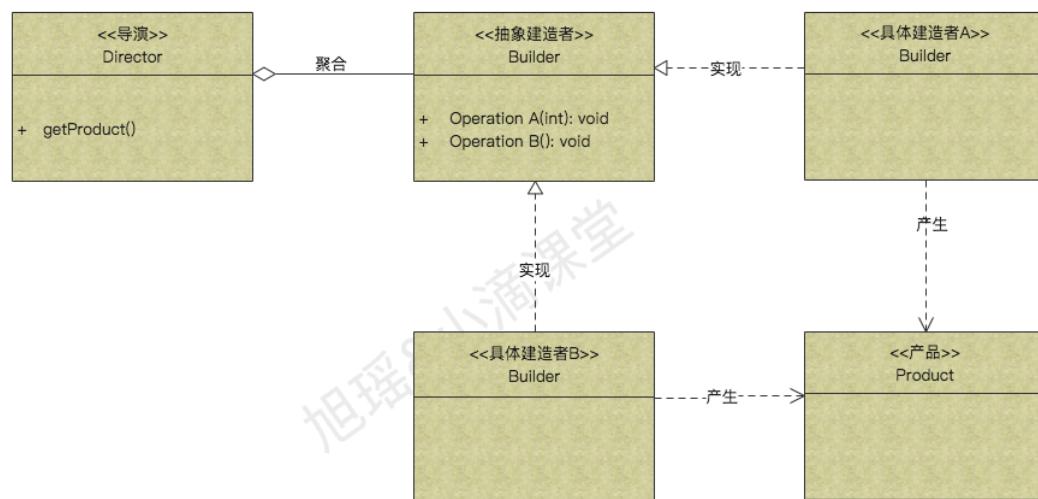
简介：讲解建造者模式介绍和应用场景《上》

- 建造者模式（Builder Pattern）
 - 使用多个简单的对象一步一步构建成一个复杂的对象，将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示
 - 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，不需要知道内部的具体构建细节
- 场景举例
 - KFC创建套餐：套餐是一个复杂对象，它一般包含主食如汉堡、烤翅等和饮料如果汁、可乐等组成部分，不同的套餐有不同的组合，而KFC的服务员可以根据顾客的要求，一步一步装配这些组成部分，构造一份完整的套餐



- 电脑有低配、高配，组装需要CPU、内存、电源、硬盘、主板等

小滴课堂&旭瑶



- 核心组成
 - Builder: 抽象建造者, 定义多个通用方法和构建方法
 - ConcreteBuilder: 具体建造者, 可以有多个
 - Director: 指挥者, 控制整个组合过程, 将需求交给建

造者，由建造者去创建对象

- Product：产品角色

第4集 创建型设计模式-建造者模式应用介绍 《下》

简介：讲解建造者模式介绍和应用场景《下》

- 编码实践（同个设计模式 包括博文、书籍等不会完全相同，大体思想一致就行）

```
/**
```

```
* 声明了建造者的公共方法
```

```
*/\n\npublic interface Builder {\n\n    /**\n     * 细节方法\n     */\n    void buildCpu();\n\n    void buildMainboard();\n\n    void buildDisk();\n\n    void buildPower();\n\n    void buildMemory();\n\n    Computer createComputer();\n}\n\n/**\n * 小滴课堂,愿景: 让技术不再难学\n *\n * @Description 将产品和创建过程进行解耦, 使用相同的\n * 创建过程创建不同的产品, 控制产品生产过程\n *\n * Director是全程指导组装过程, 具体的\n * 细节还是builder去操作\n *\n * @Author 二当家小D
```

```
* @Remark 有问题直接联系我，源码-笔记-技术交流群 微  
信： xdclass6  
* @Version 1.0  
**/
```

```
public class Director {  
  
    public Computer craete(Builder builder){  
  
        builder.buildMemory();  
        builder.buildCpu();  
        builder.buildMainboard();  
        builder.buildDisk();  
        builder.buildPower();  
  
        return builder.createComputer();  
  
    }  
  
}
```

```
public class HighComputerBuilder implements  
Builder{  
  
    private Computer computer = new Computer();
```

```
@Override
public void buildCpu() {
    computer.setCpu("高配 CPU");
}

@Override
public void buildMainboard() {
    computer.setMainboard("高配 主板");
}

@Override
public void buildDisk() {
    computer.setDisk("高配 磁盘");
}

@Override
public void buildPower() {
    computer.setPower("高配 电源");
}

@Override
public void buildMemory() {
    computer.setMemory("高配 内存");
}

@Override
public Computer createComputer() {
    return computer;
}
```

```
}
```

- 优点
 - 客户端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
 - 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，更加精细地控制产品的创建过程
 - 增加新的具体建造者无须修改原有类库的代码，符合开闭原则
 - 建造者模式结合**链式编程**来使用，代码上更加美观
- 缺点
 - 建造者模式所创建的产品一般具有较多的共同点，如果产品差异大则不建议使用
- JDK里面的应用
 - tcp传输协议 protobuf 生成的api、java中的 StringBuilder（不完全一样，思想一样）
- 建造者模式与抽象工厂模式的比较:
 - 建造者模式返回一个组装好的完整产品，**抽象工厂模式**返回一系列相关的产品，这些产品位于不同的产品等级结构，构成了一个产品族



技术与生活更加有趣" 更多架构课程请访问 xdclass.net

第六章 不知道结构型设计模式？面试官会说 出门右转

第1集 接口之间的桥梁-适配器设计模式你知道多少

简介：讲解Adapeter设计模式和应用场景

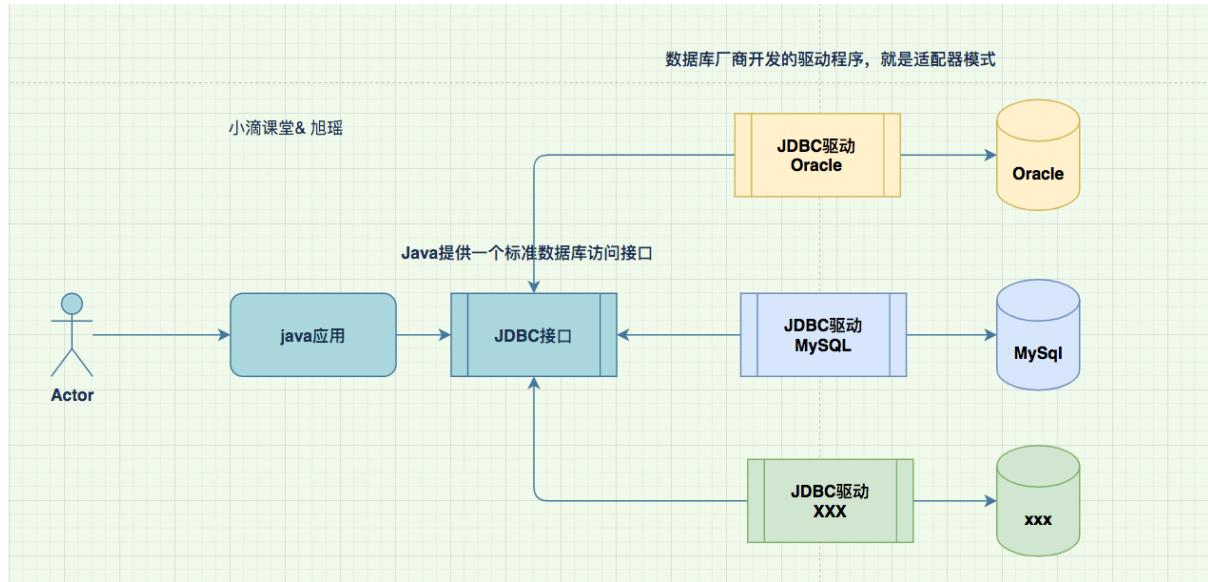
- 适配器模式 (Adapter Pattern)
 - 见名知意，是作为两个不兼容的接口之间的桥梁，属于结构型模式
 - 适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作
 - 常见的几类适配器
 - 类的适配器模式
 - 想将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可

- 对象的适配器模式
 - 想将一个对象转换成满足另一个新接口的对象时，可以创建一个适配器类，持有原类的一个实例，在适配器类的方法中，调用实例的方法就行
 - 接口的适配器模式
 - 不想实现一个接口中所有的方法时，可以创建一个Adapter，实现所有方法，在写别的类的时候，继承Adapter类即
-
- 应用场景
 - 电脑需要读取内存卡的数据，读卡器就是适配器



- 日常使用的转换头，如电源转换头，电压转换头
- 系统需要使用现有的类，而这些类的接口不符合系统的需要
- JDK中InputStreamReader就是适配器
- JDBC就是我们用的最多的适配器模式

JDBC给出一个客户端通用的抽象接口，每一个具体数据库厂商 如 SQL Server、Oracle、MySQL等，就会开发JDBC驱动，就是一个介于JDBC接口和数据库引擎接口之间的适配器软件



第2集 提高开发效率 接口适配器在日常开发里面的应用

简介：接口的适配器案例实战

- 设计模式的疑惑
 - 会感觉到好像是理解了模式的功能，但是一到真实的系统开发中，就不知道如何使用这个模式了
 - 前面每个案例都有讲实际的编码操作，大家一定要充分领悟
- 接口适配器

有些接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要实现部分接口就可以了

- 编码实战

第3集 生产环境接口-需要兼容新的业务怎么办？

简介：适配器案例实战，生产环境得的接口需要兼容新的业务

- 需求背景

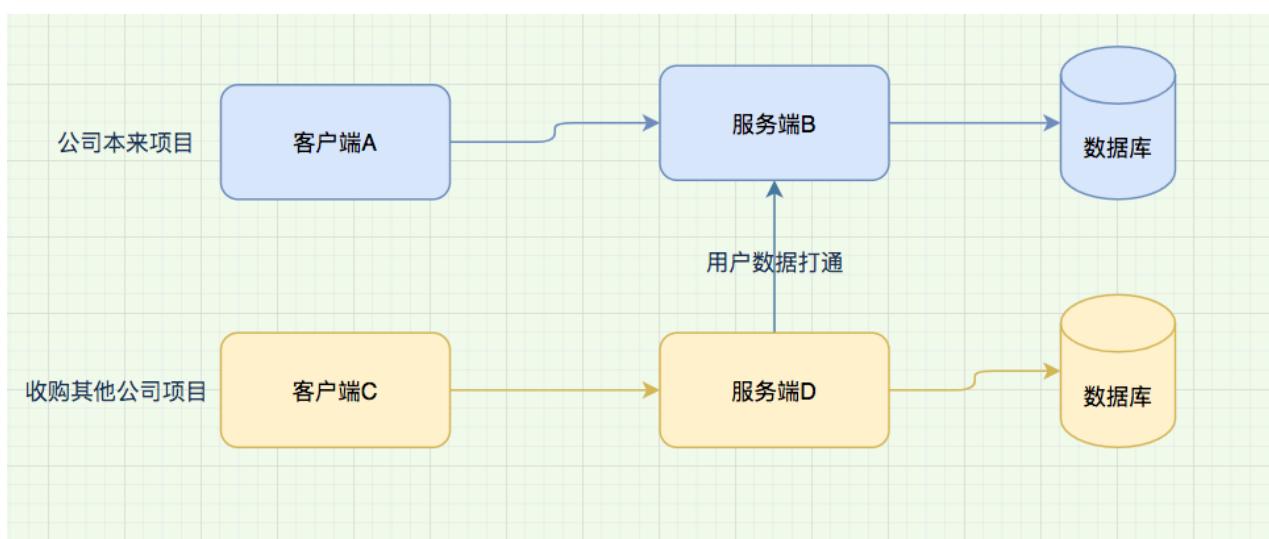
小滴课堂里面有个电商支付项目，里面有个登录功能，已经线上运行了

客户端A 调用生产环境的登录接口B，且线上稳定运行了好几年。

某天，公司接到收购了别的公司的项目，需要把这套系统部署在一起，且收购的项目也有对应的客户端C，但是两个客户端和服务端的协议不一样

需求：收购的项目客户端C，需要做公司原来项目的用户数据打通，连接公司的服务端登录接口B，你能想到几个解决方案？

- 1、修改就项目B的登录接口，兼容C客户端协议（可能影响线上接口，不稳定）
- 2、新增全新的登录接口F，采用C客户端协议（和旧的业务逻辑会重复）
- 3、新增一个转换协议接口，客户端C调用旧的B接口之前，使用转换接口转换下协议（适配器模式，推荐这种方式）



- 总结

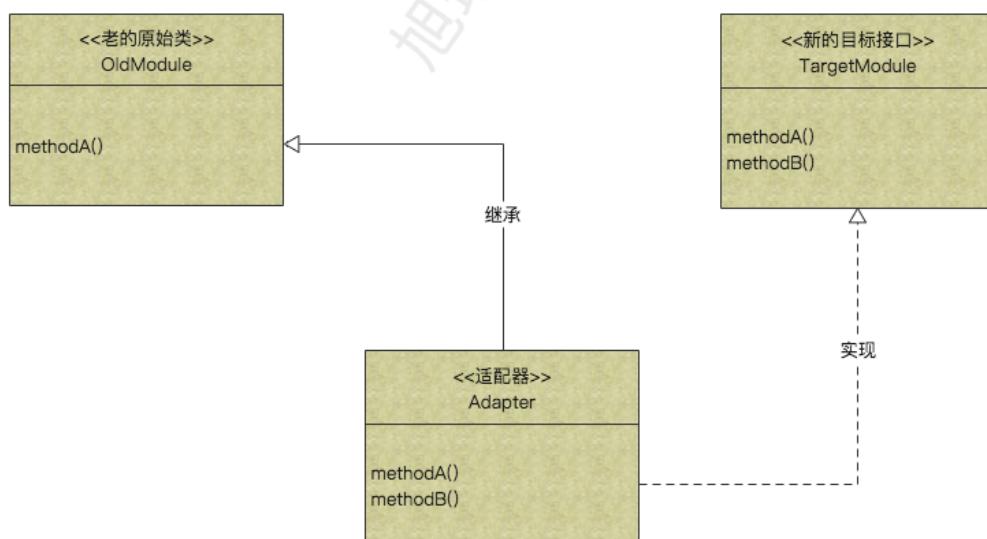
- 在使用一些旧系统或者是类库时，经常会出现接口不兼容的问题，适配器模式在解决这类问题具有优势
 - 学习设计模式一定不要局限代码层面，要从软件系统整体去考虑,而不是为了使用设计模式，而去使用设计模式
-
- 优点
 - 可以让任何两个没有关联的类一起运行，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作
 - 增加灵活度, 提高复用性，适配器类可以在多个系统使用,符合开闭原则
 - 缺点
 - 整体类的调用链路增加，本来A可以直接调用C，使用适配器后 是A调用B， B再调用C

第4集 适配器设计模式-类的适配器

简介：适配器案例实战，类的适配器

- 类的适配器模式

- 想将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可



```
/**  
 * 小滴课堂,愿景: 让技术不再难学 xdclass.net  
 *  
 * @Description 老的类, 里面有1个方法  
 * @Author 二当家小D
```

```
* @Remark 有问题直接联系我，源码-笔记-技术交流群 微  
信： xdclass6  
* @Version 1.0  
**/  
  
public class OldModule {  
  
    public void methodA(){  
        System.out.println("OldModule  
methodA");  
    }  
  
}  
  
  
  
  
public interface TargetModule {  
  
    /**  
     * 和需要适配的类方法名一样  
     */  
    void methodA();  
  
    /**  
     * 新的方法，如果有多个新的方法直接编写就行  
     */  
    void methodB();  
  
    void methodC();  
  
}
```

```
public class Adapter extends OldModule
implements TargetModule {

    /**
     * 新的方法，和老的类方法不一样
     */
    @Override
    public void methodB() {
        System.out.println("Adapter methodB");
    }

    /**
     * 新的方法，和老的类方法不一样
     */
    @Override
    public void methodC() {
        System.out.println("Adapter methodC");
    }

}

public static void main(String[] args) {
    TargetModule targetModule = new
Adapter();
    targetModule.methodA();
    targetModule.methodB();
```

```
targetModule.methodC();  
}
```

第5集 设计模式疑惑指南-桥接设计模式

简介：学设计模式的疑惑，桥接设计模式的介绍

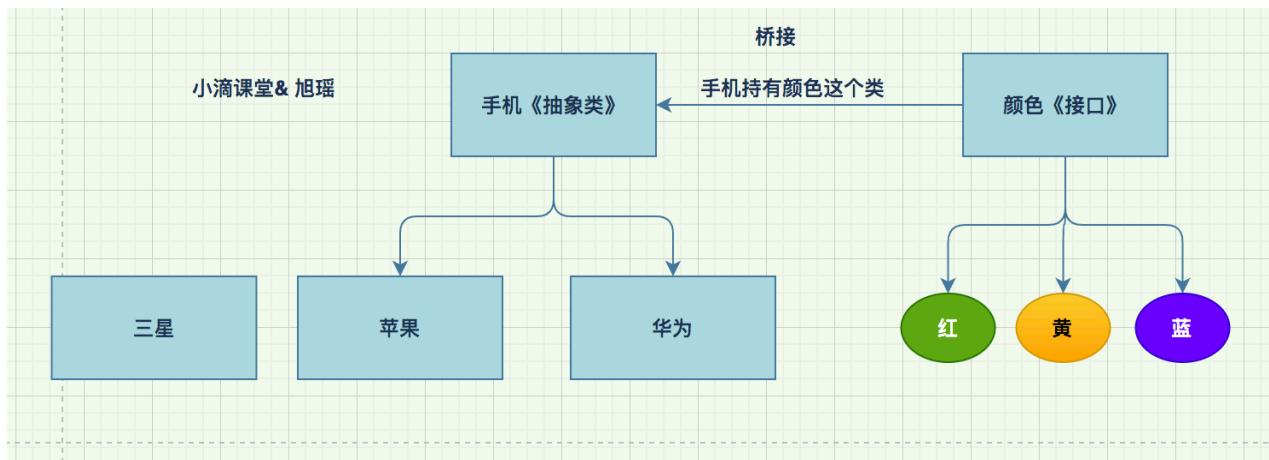
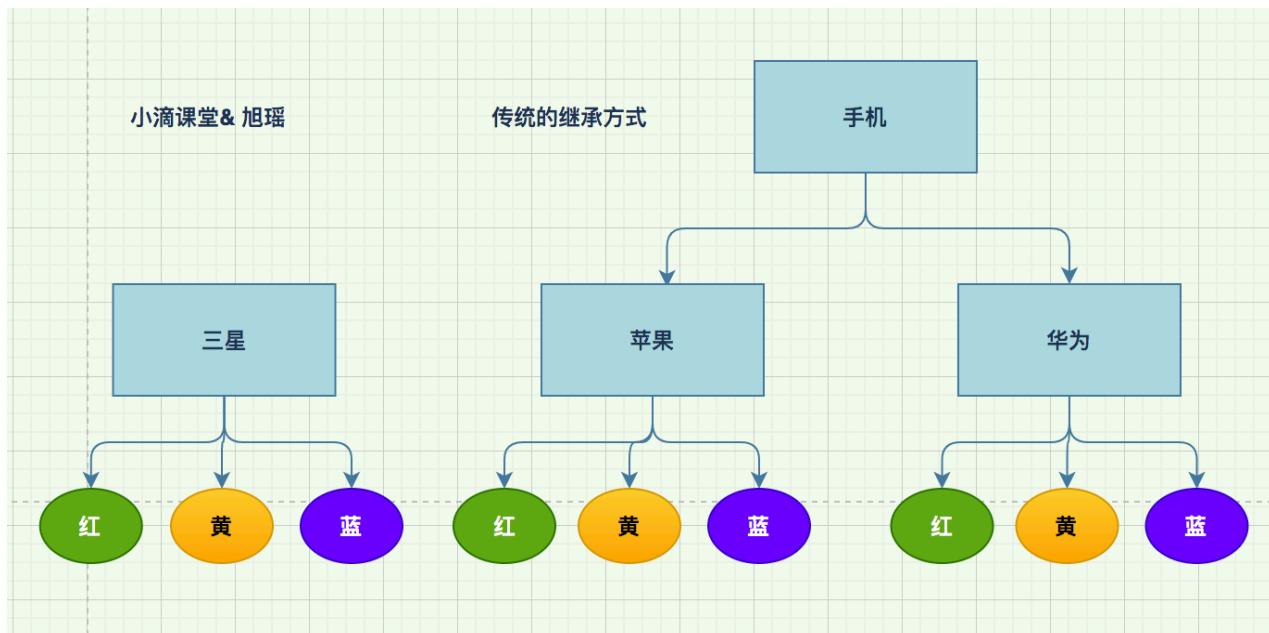
- 打个鸡血
- 牛逼的工程师，在日常开发，明明用了某种设计模式了，但他还没意识到是哪种模式，只知道这样做是最合适的，这种就是已经把设计模式融会贯通了。
 - 而那种刚学了一点设计模式就到处想用，以及把精力浪费在区分各个设计模式上的人，未来就陷入死循环里面。

- 桥接设计模式
 - 适配器模式类似，包括以后经常会遇到意思接近一样的设计模式，因为大神往往就是多个模式混用，且根据不同的场景进行搭配，桥接设计模式也是结构型模式
 - 将抽象部分与实现部分分离，使它们都可以独立的变化
 - 通俗来说，是通过组合来桥接其它的行为/维度
- 应用场景
 - 系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性
 - 不想使用继承导致系统类的个数急剧增加的系统
 - 有时候一个类，可能会拥有多个变化维度，比如啤酒，有不同的容量和品牌，可以使用继承组合的方式进行开发，假如维度很多，就容易出现类的膨胀，使用桥接模式就可以解决这个问题，且解耦
- 业务背景

我们需要构建一个手机类，我们知道手机有很多品牌，苹果、华为等，从另外一个颜色维度，又有多种颜色，红、黄、蓝等，

那如果描述这些类的话，传统方式就直接通过继承，就需要特别多的类，品牌2，颜色3，就是6个类了，如果后续再增加品牌就更多了，类数目将会激增，即所谓的类爆炸

使用桥接模式就可以解决这个问题，且灵活度大大提高



第6集 桥接设计模式案例实战

简介：桥接设计模式的案例实战

- 编码实战

```
/**  
 * 小滴课堂,愿景：让技术不再难学  
 *  
 * @Description 抽象角色 手机  
 * @Author 二当家小D  
 * @Remark 有问题直接联系我，源码-笔记-技术交流群
```

```
* @Version 1.0
*/
public abstract class Phone {
    /**
     * 通过组合的方式来桥接其他行为
     */
    protected Color color;
    public void setColor(Color color) {
        this.color = color;
    }
    /**
     * 手机的方法
     */
    abstract public void run();
}
```

```
//颜色维度
public interface Color {
    void useColor();
}

//具体颜色
public class BlueColor implements Color {
    @Override
    public void useColor() {
        System.out.println("蓝色");
    }
}
```

```
/***
 * 小滴课堂,愿景: 让技术不再难学
 *
 * @Description 手机的实例化
 * @Author 二当家小D
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群
 * @Version 1.0
 **/


public class HWPhone extends Phone {

    public HWPhone(Color color){
        super.setColor(color);
    }

    @Override
    public void run() {
        color.useColor();
        System.out.println("华为手机");
    }
}

//使用, 把组合的形式
public static void main(String[] args) {
    HWPhone blueHwPhone = new HWPhone(new
BlueColor());
    blueHwPhone.run();

    HWPhone redHwPhone = new HWPhone(new
RedColor());
}
```

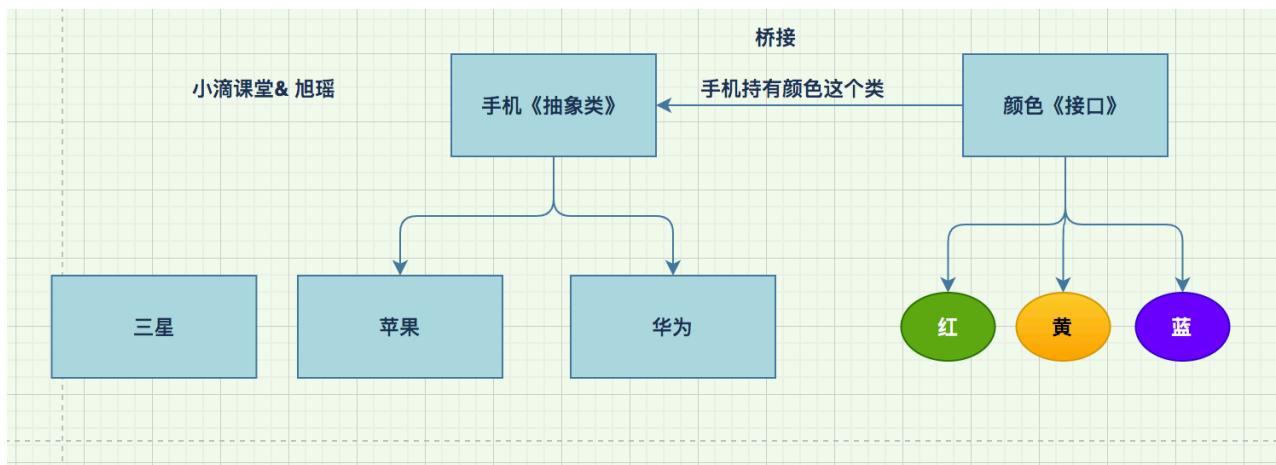
```

        redHwPhone.run();

        ApplePhone applePhone = new
ApplePhone(new RedColor());
        applePhone.run();

    }

```



- 优点
 - 抽象和实现的分离。
 - 优秀的扩展能力，符合开闭原则
- 缺点
 - 增加系统的理解与设计难度
 - 使用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与编程，比如抽象类汽车，里面聚合了颜色类，有点像对象适配器
- 总结和对比

- 按GOF的说法，桥接模式和适配器模式用于设计的不同阶段，
 - 桥接模式用于设计的前期，精细化的设计，让系统更加灵活
 - 适配器模式用于设计完成之后，发现类、接口之间无法一起工作，需要进行填坑
- 适配器模式经常用在第三方API协同工作的场合，在功能集成需求越来越多的今天，这种模式的使用频度越来越高，包括有些同学听过 外观设计模式，这个也是某些场景和适配器模式一样

第7集 将对象组合成树形结构的模式-组合设计模式讲解

简介：介绍组合设计模式

- 组合设计模式
 - 又叫部分整体模式，将对象组合成树形结构以表示“部

“分-整体”的层次结构，可以更好的实现管理操作

- 组合模式使得用户可以使用一致的方法操作单个对象和组合对象
- 部分-整体对象的基本操作多数是一样的，但是应该还会有不一样的地方
- 核心：组合模式可以使用一棵树来表示

- 应用场景

- 银行总行，总行有前台、后勤、网络部门等，辖区下还有地方分行，也有前台、后勤、网络部门，最小的分行就没有子分行了
- 公司也是，总公司下有子公司，每个公司大部分的部门都类似
- 文件夹和文件，都有增加、删除等api，也有层级管理关系
- 当想表达对象的部分-整体的层次结构
- 当我们的要处理的对象可以生成一颗树形结构，我们要对树上的节点和叶子进行操作时，它能够提供一致的方式，而不用考虑它是节点还是叶子

- 角色

- 组合部件 (Component)：它是一个抽象接口，表示树根，例子：总行

- 合成部件 (Composite) : 和组合部件类似，也有自己的子节点，例子：总行下的分行
- 叶子 (Leaf) : 在组合中表示子节点对象，注意是没有子节点，例子：最小地方的分行

第8集 组合设计模式实战-文件展示-树叶子节点

简介：组合设计模式文件展示-树叶子节点

- 编码实战

```
/**  
 * 小滴课堂,愿景：让技术不再难学 https://xdclass.net  
 * @Description 根节点, 抽象类, 通用的属性和方法  
 * @Author 二当家小D  
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群 微信  
xdclass6  
 * @Version 1.0  
 */
```

```
public abstract class Root {  
    private String name;  
    public Root(String name){  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public abstract void addFile(Root root);  
  
    public abstract void removeFile(Root root);  
  
    public abstract void display(int depth);  
}  
  
/**  
 * 小滴课堂,愿景: 让技术不再难学 xdclass.net  
 *  
 * @Description 具体的文件夹, 里面可以添加子文件夹或者文件  
 * @Author 二当家小D  
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群 微信  
xdclass6  
 * @Version 1.0  
**/
```

```
public class Folder extends Root {  
  
    List<Root> folders = new ArrayList<>();  
  
    public Folder(String name){  
        super(name);  
    }  
  
    public List<Root> getFolders() {  
        return folders;  
    }  
  
    public void setFolders(List<Root> folders)  
    {  
        this.folders = folders;  
    }  
  
    @Override  
    public void addFile(Root root) {  
        folders.add(root);  
    }  
  
    @Override  
    public void removeFile(Root root) {  
        folders.remove(root);  
    }  
}
```

```
    @Override
    public void display(int depth) {

        StringBuilder sb = new StringBuilder();
        for(int i=0; i<depth;i++){
            sb.append( "-");
        }
        //打印横线和当前文件名

        System.out.println(sb.toString()+this.getName()
);

        for(Root r : folders){
            //每个下级， 横线多2个
            r.display(depth+2);
        }

    }

}

/**
 * 小滴课堂,愿景：让技术不再难学 xdclass.net
 *
 * @Description 这个类是没有节点，不用存储其他子类数组，所以是叶子节点
 * @Author 二当家小D
 * @Remark 有问题直接联系我，源码-笔记-技术交流群，微信 xdclass6
 * @Version 1.0
```

```
**/




public class File extends Root{



    public File(String name){
        super(name);
    }

    @Override
    public void addFile(Root root) {

    }

    @Override
    public void removeFile(Root root) {

    }

    @Override
    public void display(int depth) {
        StringBuilder sb = new StringBuilder();
        for(int i=0; i<depth;i++){
            sb.append( "-" );
        }
        //打印横线和当前文件名

        System.out.println(sb.toString()+this.getName() );
    }
}
```

```
    }  
}
```

//使用

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

//创造根文件夹

```
Root root1 = new Folder("C://");
```

//建立子文件

```
Root desktop = new Folder("桌面");
```

Root myComputer = new Folder("我的电
脑");

//建立子文件

```
Root javaFile = new  
File("HelloWorld.java");
```

//建立文件夹关系

```
root1.addFile(desktop);
```

```
root1.addFile(myComputer);
```

//建立文件关系

```
myComputer.addFile(javaFile);
```

//从0级开始展示，每下一级，多2条横线

```
root1.display(0);
```

//另外一个根

```
Root root2 = new Folder("D://");
root2.display(0);

}
```

- 缺点
 - 客户端需要花更多时间理清类之间的层次关系
- 优点
 - 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
 - 方便创建出复杂的层次结构



旭瑶&小滴课堂 愿景："让编程不再难学，让技术与生活更加有趣" 更多架构课程请访问 xdclass.net

第七章 你知道结构型设计模式装饰器+代理模式？留下来聊聊

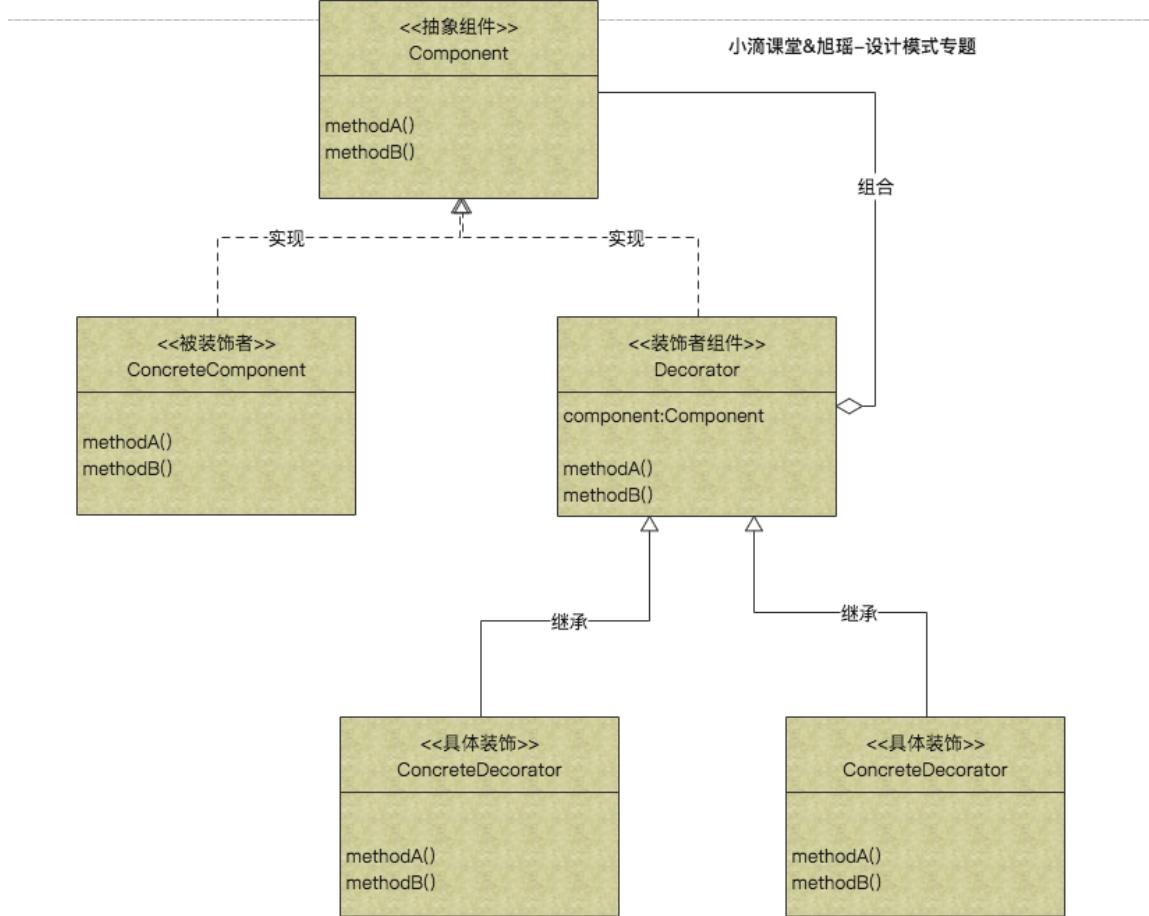
第1集 如虎添翼的设计模式-装饰器设计模式介绍

简介：讲解-装饰器设计模式介绍和应用场景

- 装饰器设计模式（Decorator Pattern）
 - 也叫包装设计模式，属于结构型模式，它是作为现有的类的一个包装，允许向一个现有的对象添加新的功能，同时又不改变其结构
 - 给对象增加功能，一般两种方式 继承或关联组合，将一个类的对象嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为来增强功能，这个就是装饰器模式，比继承模式更加灵活
- 应用场景
 - 小滴课堂-老王，本来计划买跑车撩妹的，结果口袋没钱，改买自行车，为了显得突出，店家提供多种改装方案，加个大的喇叭、加个防爆胎等，经过装饰之后成为目的更明确的自行车，更能解决问题。像这种不断为对象添加装饰的模式就叫 Decorator 模式，Decorator 指的是装饰物。



- 以动态、透明的方式给单个对象添加职责，但又能不改变其结构
- JDK源码里面应用的最多的就是IO流，大量使用装饰设计模式



- 角色 (装饰者和被装饰者有相同的超类(Component))
 - 抽象组件 (Component)
 - 定义装饰方法的规范，最初的自行车，仅仅定义了自行车的API；
 - 被装饰者 (ConcreteComponent)
 - Component的具体实现，也就是我们要装饰的具体对象
 - 实现了核心角色的具体自行车
 - 装饰者组件 (Decorator)

- 定义具体装饰者的行为规范, 和Component角色有相同的接口, 持有组件(Component)对象的实例引用
 - 自行车组件 都有 名称和价格
-
- 具体装饰物 (ConcreteDecorator)
 - 负责给构件对象装饰附加的功能
 - 比如 喇叭, 防爆胎

第2集 满足老王的改装搭配-玩转装饰器设计模式 案例实战

简介：装饰器设计模式案例实战

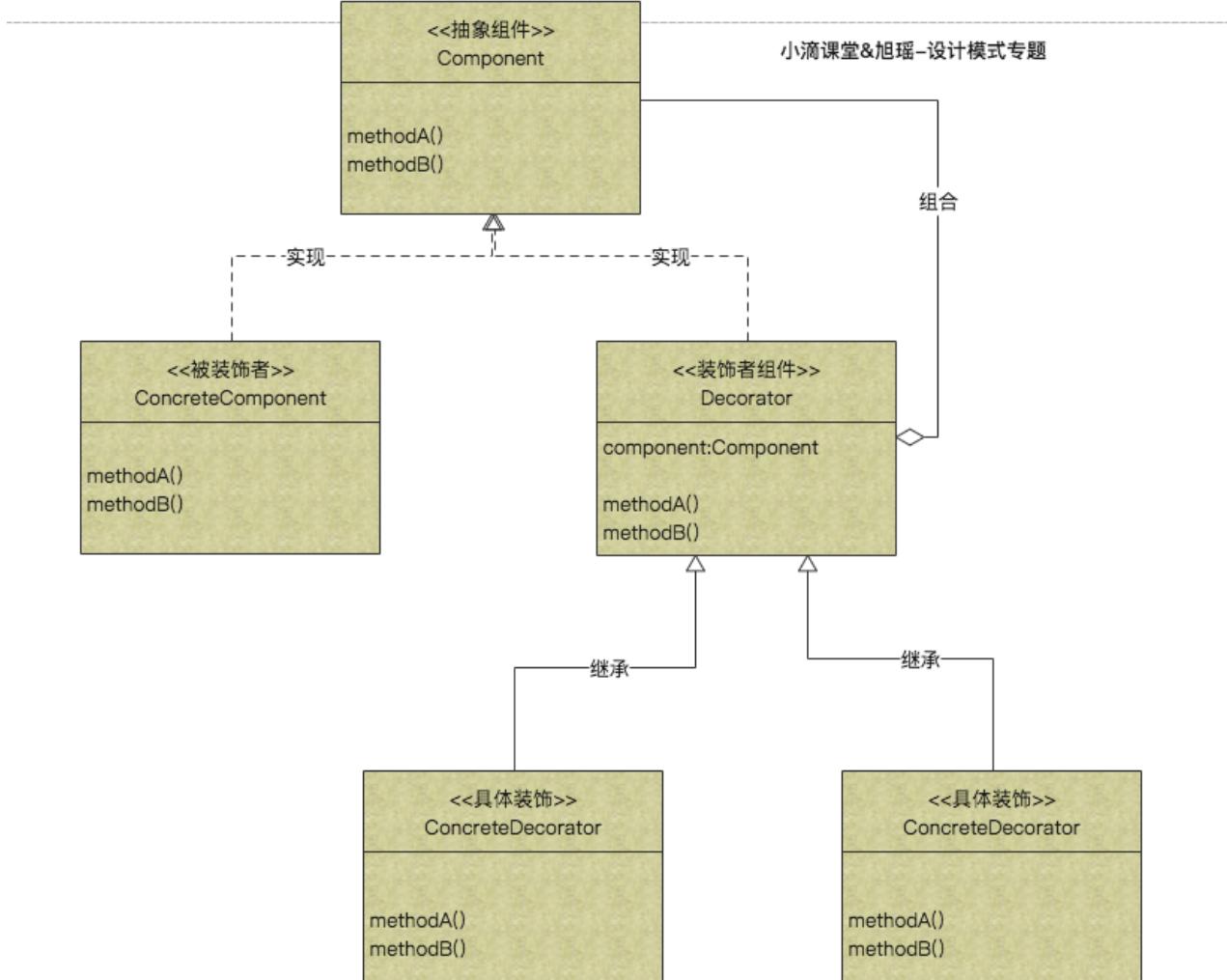
- 背景需求

小滴课堂-老王，由于公司发了项目奖金，不够买跑车，就先买自行车，店家里面 有小号、中号、大号等规格的自行车。

然后改造加一个喇叭，后来不够又要加多一个，一个防爆胎不够，又有两个，存在很多个随机组合的改装。

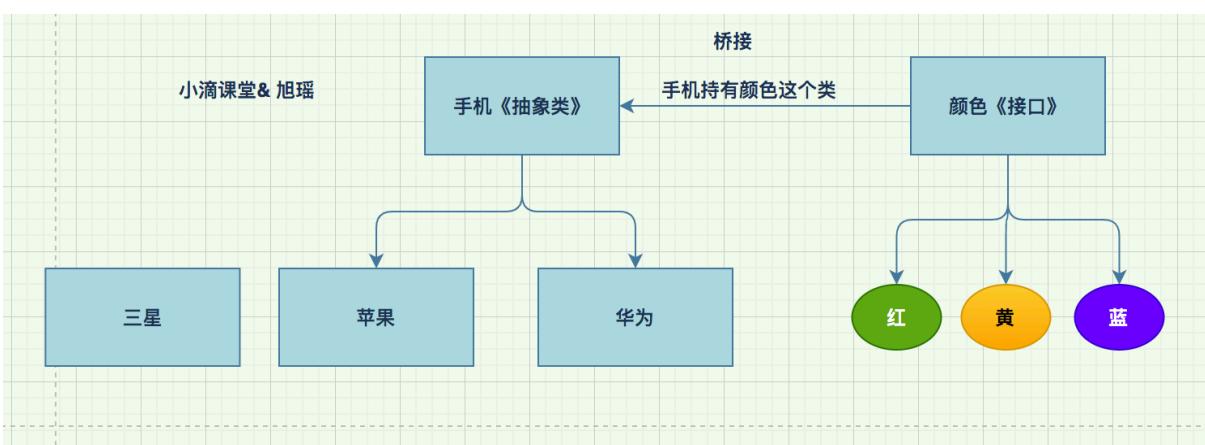
店家就苦恼了，这样的结构难搞，价格也难算，而且需求再变动，就更麻烦了。

使用装饰者就可以解决这个问题



- 不采用设计模式，你会怎么做？自己试试，然后看是否够灵活
- 采用装饰器设计模式编码实战
- 优点
 - 装饰模式与继承关系的目的都是要扩展对象的功能，但装饰模式可以提供比继承更多的灵活性。

- 使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合，原有代码无须改变，符合“开闭原则”
- 缺点
 - 装饰模式增加了许多子类，如果过度使用会使程序变得很复杂（多层包装）
 - 增加系统的复杂度，加大学习与理解的难度
- 装饰器模式和桥接模式对比
 - 相同点都是通过封装其他对象达到设计的目的，和对象适配器也类似，有时也叫半装饰设计模式
 - 没有装饰者和被装饰者的主次区别，桥接和被桥接者是平等的，桥接可以互换，不用继承自同一个父类
比如例子里面的，可以是Phone持有Color，也可以是Color持有Phone



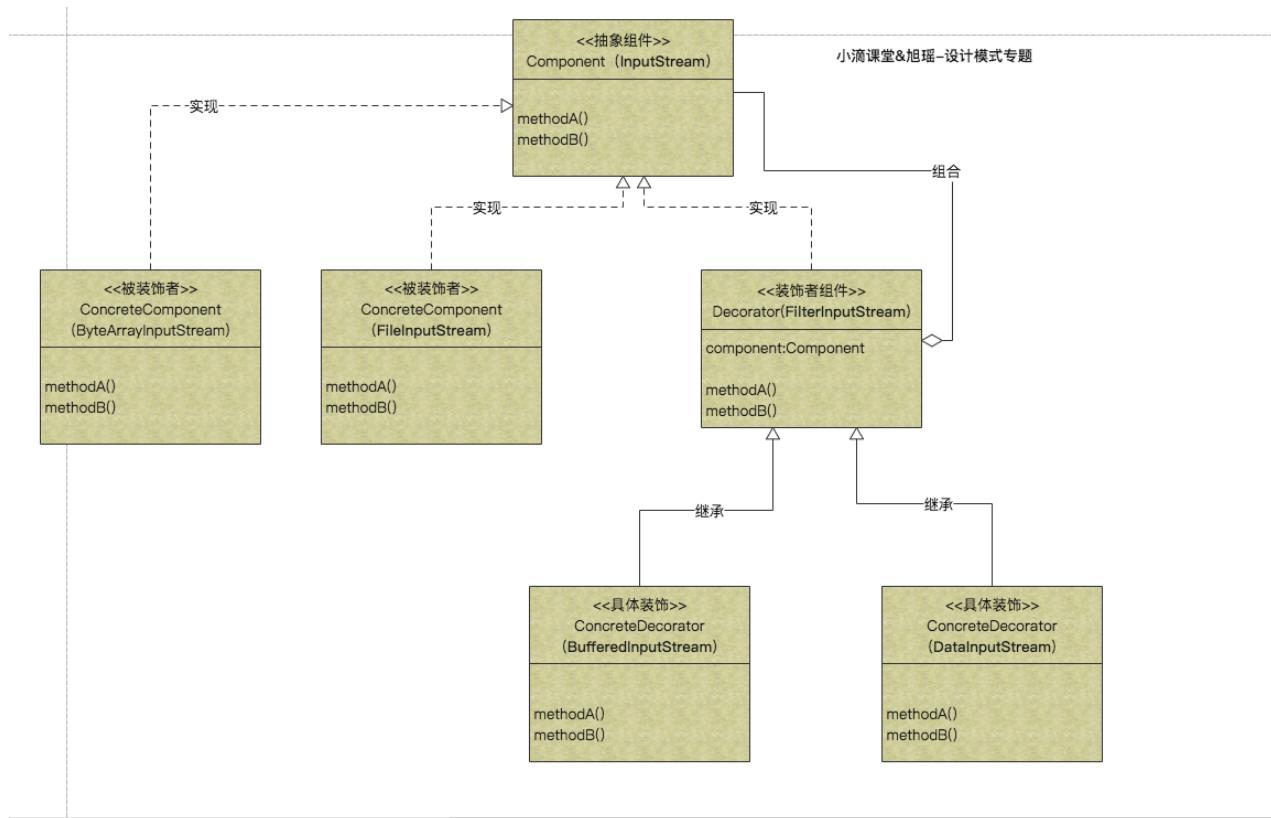
- 桥接模式不用使用同一个接口；装饰模式用同一个接口装饰，接口在父类中定义

第3集 JDK源码里面的Stream IO流-装饰器设计模式应用

简介：讲解-装饰器设计模式在JDK源码里面应用场景

- 抽象组件 (Component) : InputStream
 - 定义装饰方法的规范
- 被装饰者 (ConcreteComponent) : FileInputStream 、
ByteArrayInputStream
 - Component的具体实现，也就是我们要装饰的具体对象

- 装饰者组件 (Decorator) : FilterInputStream
 - 定义具体装饰者的行为规范, 和Component角色有相同的接口, 持有组件(Component)对象的实例引用
 - 自行车组件 都有 名称和价格
- 具体装饰物 (ConcreteDecorator) : BufferedInputStream、DataInputStream
 - 负责给构件对象装饰附加的功能
 - 比如 喇叭, 防爆胎



- 应用场景

```
//添加了Buffer缓冲功能  
InputStream inputStream = new  
BufferedInputStream(new FileInputStream(""));
```

第4集 加盟商来啦-你需要掌握的代理设计模式 《上》

简介：讲解代理设计模式，让代理帮你完成工作《上》

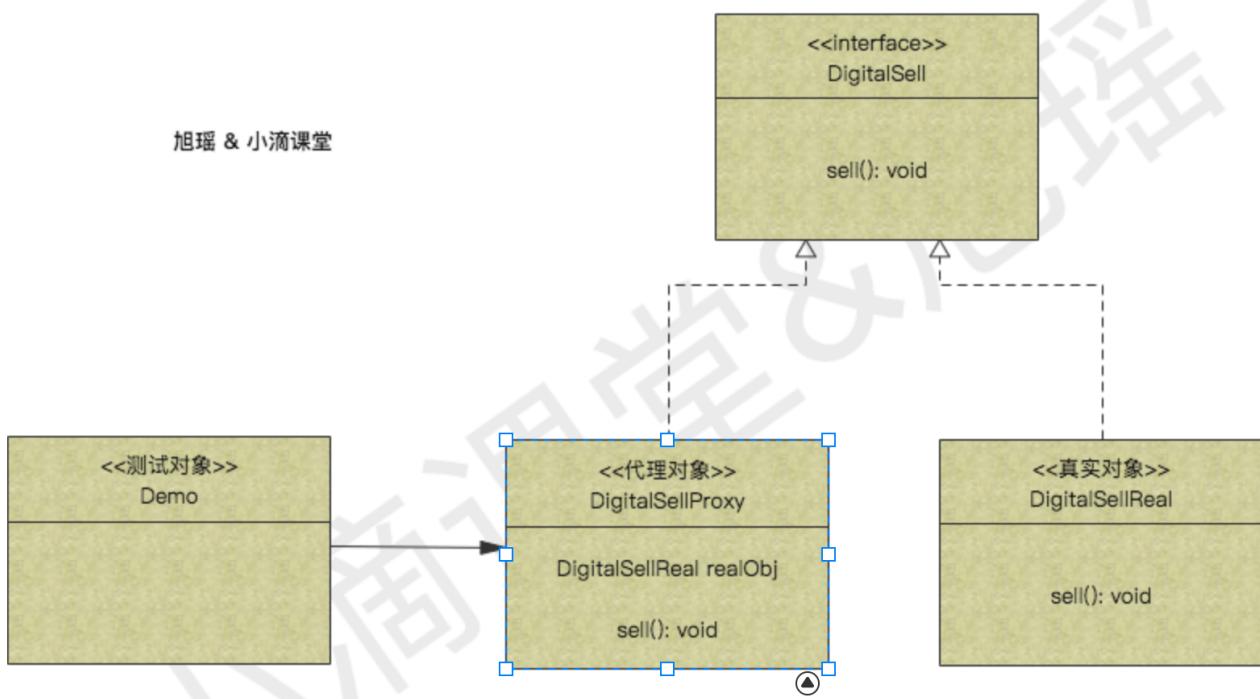
- 代理设计模式（Proxy Pattern）
 - 为其他对象提供一种代理以控制对这个对象的访问，属于结构型模式。
 - 客户端并不直接调用实际的对象，而是通过调用代理，来间接的调用实际的对象
- 应用场景

- 各大数码专营店，代理厂商进行销售对应的产品，代理商持有真正的授权代理书
- 客户端不想直接访问实际的对象，或者访问实际的对象存在困难，通过一个代理对象来完成间接的访问
- 想在访问一个类时做一些控制，或者增强功能



- 角色

- Subject: 抽象接口,真实对象和代理对象都要实现的一个抽象接口，好比销售数码产品
- Proxy: 包含了对真实对象的引用,从而可以随意的操作真实对象的方法，好比 代理加盟店
- RealProject : 真实对象，好比厂商销售数码产品



第5集 加盟商来啦-你需要掌握的代理设计模式《下》

简介：讲解代理设计模式，让代理帮你完成工作《下》

- 业务需求

小滴课堂-老王，想开个数码小卖部，为以后退休生活做准备，代理各大厂商的手机和电脑，用代理设计模式帮他实现下

Subject 卖手机

RealProject 苹果、华为厂商，核心是卖手机，但是选址不熟悉

Proxy 老王数码专卖店：代理卖手机，附加选地址，增加广告等

- 编码实战

```
/**  
 * 抽取公共的方法  
 */  
public interface DigitalSell {  
  
    void sell();  
}
```

```
/**  
 * 小滴课堂，愿景：让技术不再难学  
 *  
 * @Description 真实的对象  
 * @Author 二当家小D
```

```
* @Remark 有问题直接联系我，源码-笔记-技术交流群
* @Version 1.0
*/
public class DigitalSellReal implements
DigitalSell{
    @Override
    public void sell() {
        System.out.println("销售华为手机");
    }
}

/**
 * 小滴课堂,愿景：让技术不再难学 https://xdclass.net
 *
 * @Description 代理对象，增加了功能
 * @Author 二当家小D
 * @Remark 有问题直接联系我，源码-笔记-技术交流群 微信
xdclass6
 * @Version 1.0
*/
public class DigitalSellProxy implements
DigitalSell {
```

```
private DigitalSell realObj = new
DigitalSellReal();

@Override
public void sell() {

    makeAddress();
    realObj.sell();
    makeAD();
}

private void makeAddress(){
    System.out.println("一个人流量很高的地
址");
}

private void makeAD(){
    System.out.println("投放广告");
}

}

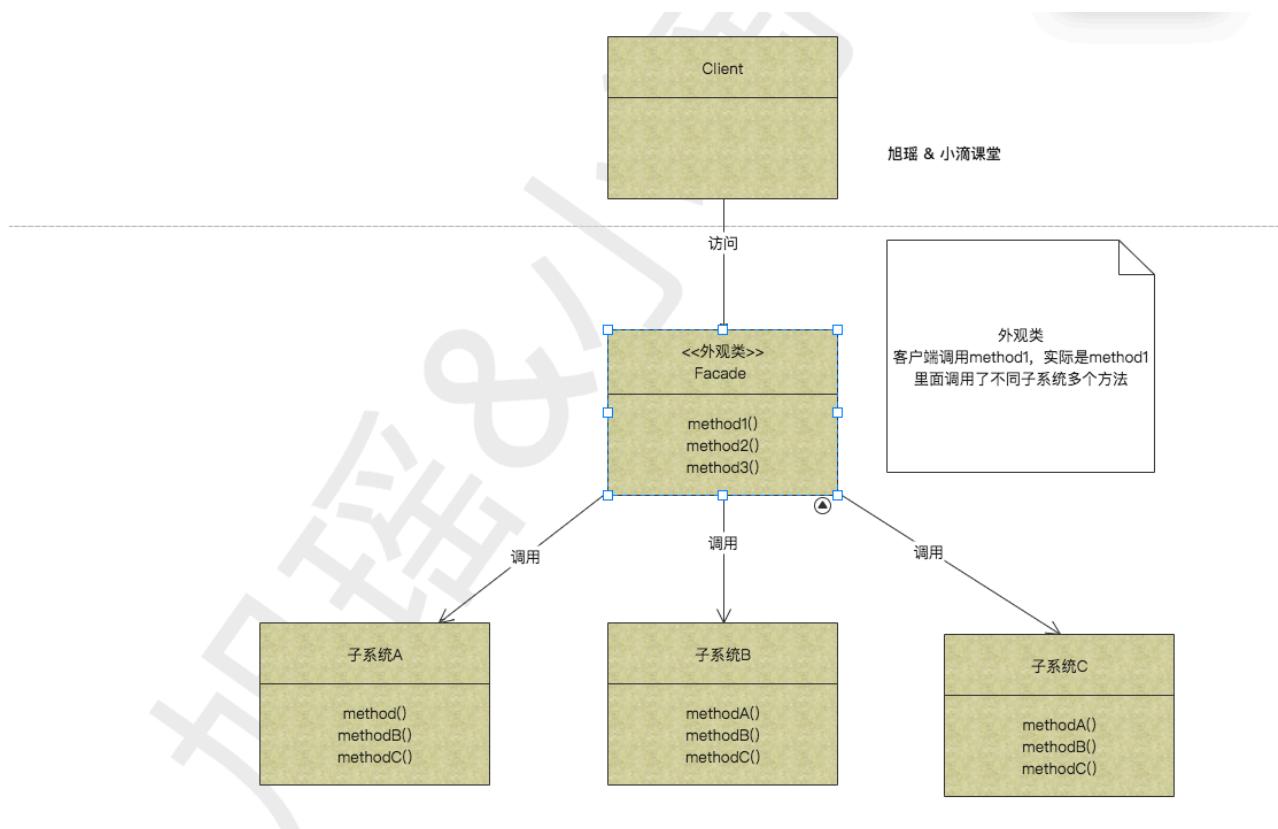
//使用
public static void main(String[] args) {

    //真实对象的行为
    DigitalSell realObj = new
DigitalSellReal();
    realObj.sell();

    //代理对象的行为
}
```

```
DigitalSell proxy = new  
DigitalSellProxy();  
proxy.sell();  
}
```

- 优点
 - 可以在访问一个类时做一些控制，或增加功能
 - 操作代理类无须修改原本的源代码，符合开闭原则，系统具有较好的灵活性和可扩展性
- 缺点
 - 增加系统复杂性和调用链路



- 有静态代理和动态代理两种
 - 动态代理也有多种方式，cglib、jdk，可以看看新版

Springboot专题的spring5模块

- 和装饰器模式的区别：
 - 代理模式主要是两个功能
 - 保护目标对象
 - 增强目标对象，和装饰模式类似了

第6集 大量使用第三方SDK-它们常用的外观设计模式你知道多少？

简介：讲解外观设计模式的介绍和应用场景

- 外观设计模式 Facade Pattern
 - 门面模式，隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口
 - 定义了一个高层接口，这个接口使得这系统更加容易使用

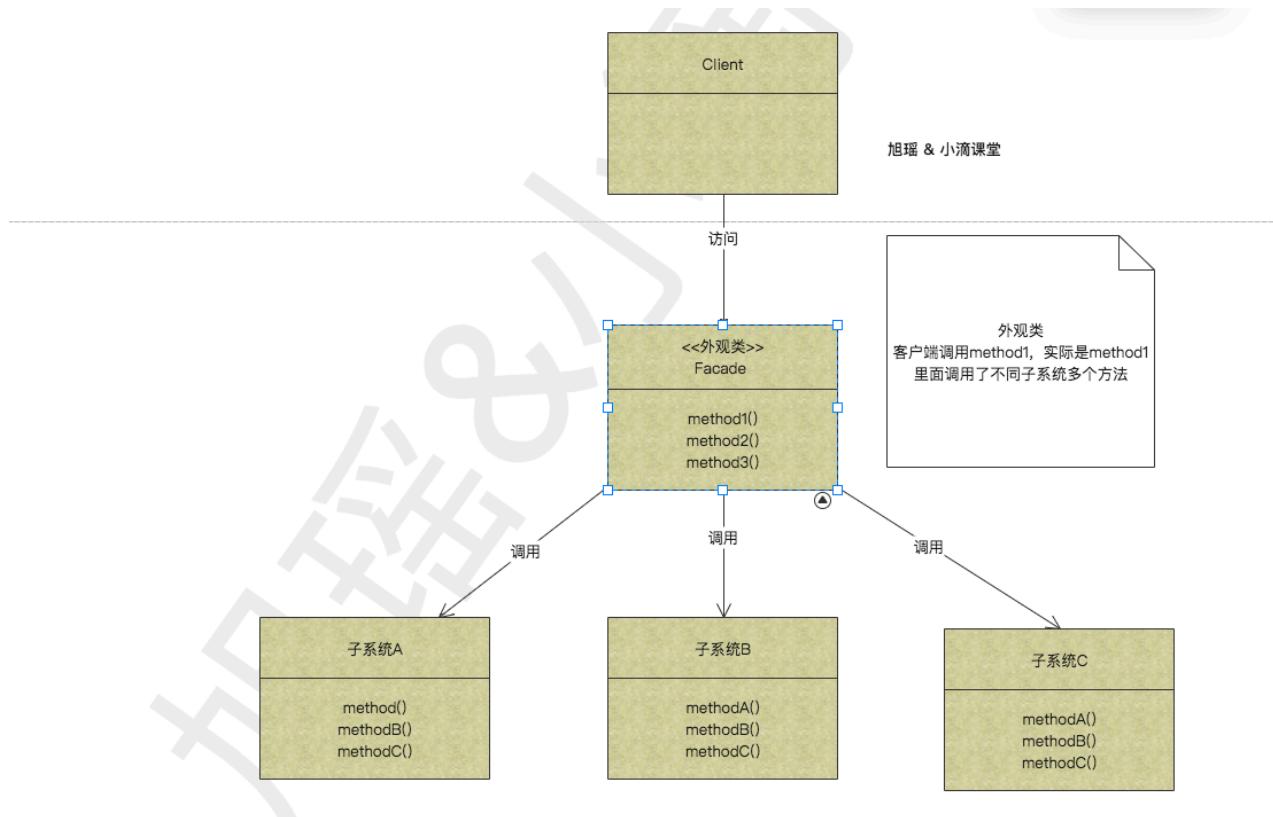


- 应用场景

- 在外人看来，小滴课堂-老王是负责消息推送这个工作，看起来很轻松，但他们不知道里面有多复杂，老王加班多久才输出一个统一的接口，只要轻松操作就可以完成复杂的事情
- 开发里面MVC三层架构，在数据访问层和业务逻辑层、业务逻辑层和表示层的层与层之间使用interface接口进行交互，不用担心内部逻辑，降低耦合性
- 各种第三方SDK大多会使用外观模式，通过一个外观类，也就是整个系统的接口只有一个统一的高层接口，这对用户屏蔽很多实现细节，外观模式经常用在封装API的常用手段
- 对于复杂难以维护的老系统进行拓展，可以使用外观设计模式
- 需要对一个复杂的模块或子系统提供一个外界访问的接口，外界对子系统的访问只要黑盒操作

- 角色

- 外观角色(Facade): 客户端可以调用这个角色的方法, 这个外观方法知道多个子系统的功能和实际调用
- 子系统角色(SubSystem): 每个子系统都可以被客户端直接调用, 子系统并不知道门面的存在,



第7集 案例实战-外观设计模式在多渠道消息推送里面的应用

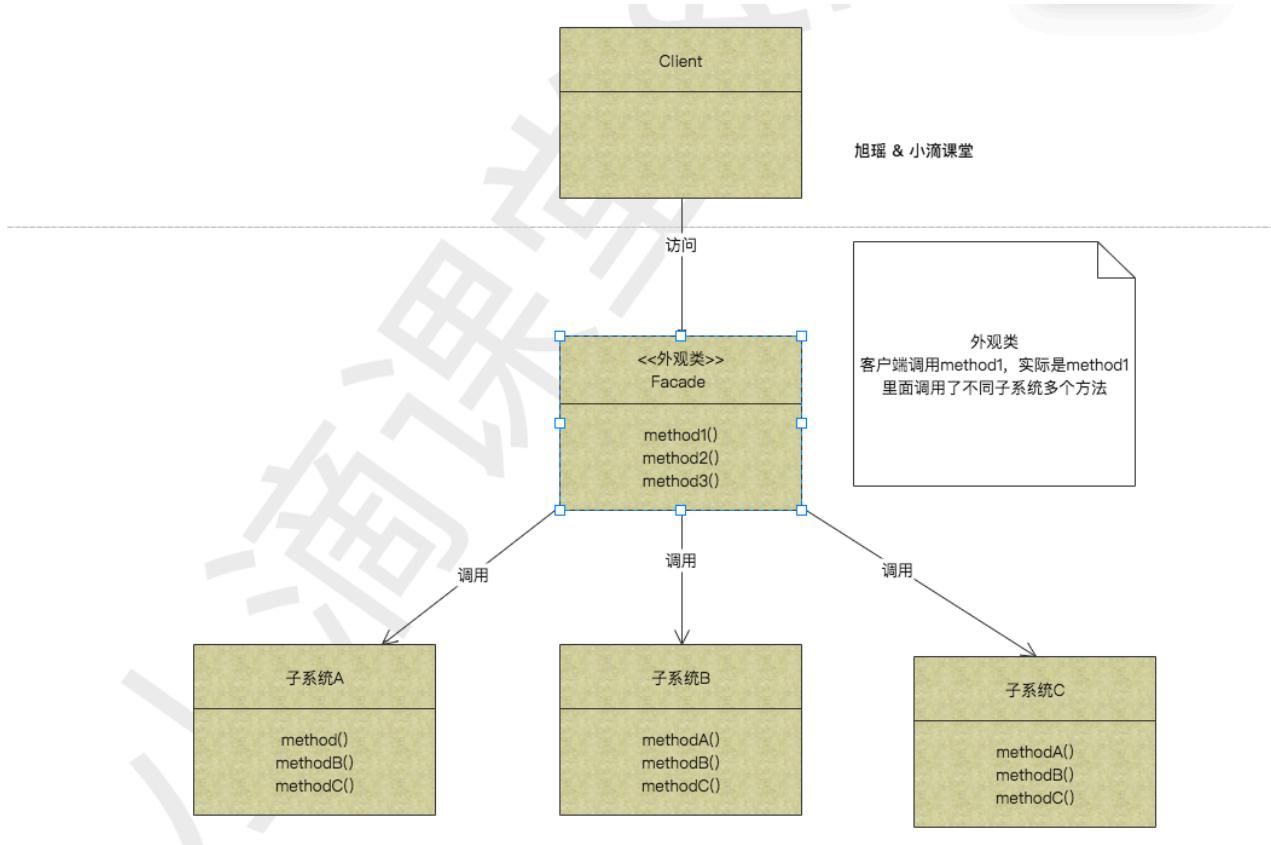
简介：讲解外观设计模式的案例实战

- 业务需求

在外人看来，小滴课堂-老王是负责消息推送这个工作，看起来很轻松，但他们不知道里面有多复杂

需要对接微信消息、邮件消息、钉钉消息等，老王加班长期加班没有女友，才输出一个统一的接口，只要轻松操作就可以完成复杂的事情

用外观设计模式帮老王完成这个需求



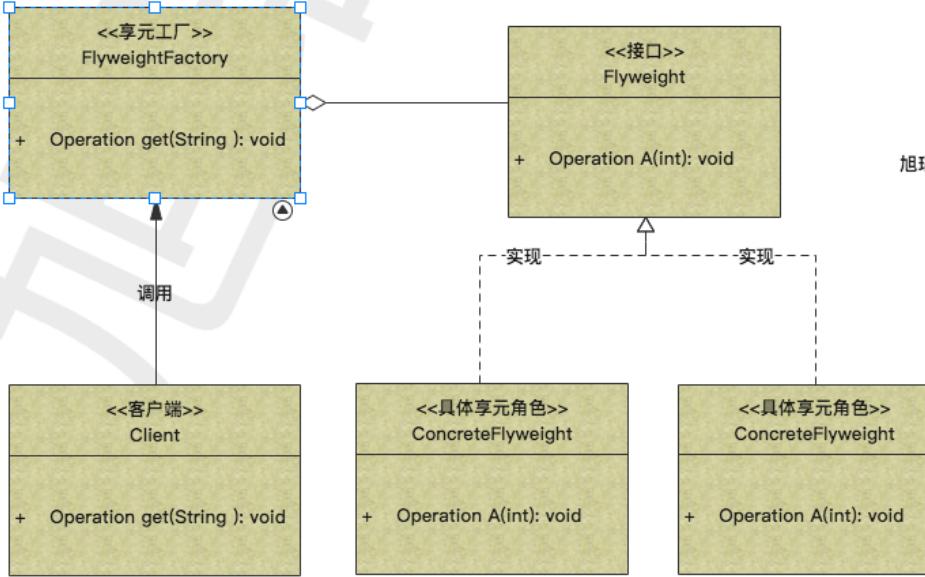
- 优点
 - 减少了系统的相互依赖，提高了灵活性
 - 符合依赖倒转原则
 - 针对接口编程，依赖于抽象而不依赖于具体
 - 符合迪米特法则
 - 最少知道原则，一个实体应当尽量少地与其他实体之间发生相互作用
- 缺点
 - 增加了系统的类和链路
 - 不是很符合开闭原则，如果增加了新的逻辑，需要修改 facade 外观类

第8集 Flyweight Pattern享元设计模式你知道多少

简介：讲解享元设计模式的介绍和应用场景

- 享元设计模式(Flyweight Pattern)
 - 属于结构型模式，主要用于减少创建对象的数量，以减少内存占用和提高性能，它提供了减少对象数量从而改善应用所需的对象结构的方式。
 - 享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象
- 应用场景
 - JAVA 中的 String，如果字符串常量池里有则返回，如果没有则创建一个字符串保存在字符串常量池里面
 - 数据库连接池、线程池等
 - 如果系统有大量相似对象，或者需要用需要缓冲池的时候可以使用享元设计模式，也就是大家说的池化技术

- 如果发现某个对象的生成了大量细粒度的实例，并且这些实例除了几个参数外基本是相同的，如果把那些共享参数移到类外面，在方法调用时将他们传递进来，就可以通过共享对象，减少实例的个数
- 内部状态
 - 不会随环境的改变而有所不同，是可以共享的
- 外部状态
 - 不可以共享的，它随环境的改变而改变的，因此外部状态是由客户端来保持（因为环境的变化一般是由客户端引起的）
- 角色
 - 抽象享元角色：为具体享元角色规定了必须实现的方法，而外部状态就是以参数的形式通过此方法传入
 - 具体享元角色：实现抽象角色规定的方法。如果存在内部状态，就负责为内部状态提供存储空间。
 - 享元工厂角色：负责创建和管理享元角色。要想达到共享的目的，这个角色的实现是关键
 - 客户端角色：维护对所有享元对象的引用，而且还需要存储对应的外部状态



第9集 老王接网站外包-享元设计模式案例实战和优缺点

简介：讲解享元设计模式的案例实战和优缺点

- 案例实战

小滴课堂-老王为了增加收入，开始接了外包项目，开发了一个AI网站模板，可以根据不同的客户需求自动生成不同类型的网站电商类、企业产品展示、信息流等。

在部署的时候就麻烦了，是不是每个机器都用租用云服务器，购买独立域名呢

这些网站结构相似度很高，而且都不是高访问量网站，可以先公用服务器资源，减少服务器资源成本，类似虚拟机或者Docker

```
public class Company {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public Company(){}  
}
```

```
public Company(String name){  
    this.name = name;  
}  
}  
//抽象接口，  
public abstract class CloudWebSite {  
  
    public abstract void run(Company company);  
  
}  
  
/**  
 * 小滴课堂,愿景：让技术不再难学 https://xdclass.net  
 *  
 * @Description  
 * @Author 二当家小D  
 * @Remark 有问题直接联系我，源码-笔记-技术交流群 微  
信 xdclass6  
 * @Version 1.0  
 */  
public class ConcreteWebSite extends  
CloudWebSite {  
    private String category;  
  
    public ConcreteWebSite(String category){  
        this.category = category;  
    }  
}
```

```
    @Override
    public void run(Company company) {
        System.out.println("网站分
类: "+category+", 公司: "+company.getName());
    }
}
```

```
//工厂
public class WebSiteFactory {
    /**
     * map里面的key是分类
     */
    private Map<String,ConcreteWebSite> map =
new HashMap<>();
    /**
     * 根据key获取分类站点
     * @param category
     * @return
     */
    public CloudWebSite
getWebSiteByCategory(String category){

    if(map.containsKey(category)){
        return map.get(category);
    }else {
        ConcreteWebSite site = new
ConcreteWebSite(category);
```

```
        map.put(category,site);

        return site;
    }

}

/***
 * 获取分类个数
 * @return
 */
public int getWebsiteCategorySize(){
    return map.size();
}

}

//使用
public static void main(String[] args) {

    WebSiteFactory factory = new
WebSiteFactory();

    CloudWebSite companySite1 =
factory.getWebSiteByCategory("企业官网");
    companySite1.run(new Company("小滴课
堂"));

    CloudWebSite companySite2 =
factory.getWebSiteByCategory("企业官网");
}
```

```
companySite2.run(new Company("旭瑶课堂"));

CloudWebSite byteDanceSite =
factory.getWebSiteByCategory("信息流");
byteDanceSite.run(new Company("字节跳动"));

CloudWebSite ucNews =
factory.getWebSiteByCategory("信息流");
ucNews.run(new Company("优视科技"));

System.out.println("网站分类总数:"
+factory.getWebsiteCategorySize());

}
```

- 优点

- 大大减少了对象的创建，降低了程序内存的占用，提高效率

- 缺点

- 提高了系统的复杂度，需要分离出内部状态和外部状态

- 注意划分内部状态和外部状态，否则可能会引起线程安全问题，必须有一个工厂类加以控制
- 享元设计模式和原型、单例模式的区别
 - 原型设计模式是指定创建对象的种类，然后通过拷贝这些原型来创建新的对象。
 - 单例设计模式保证一个类仅有一个实例



旭瑶&小滴课堂

愿景："让编程不再难学，让

技术与生活更加有趣" 更多架构课程请访问 xdclass.net

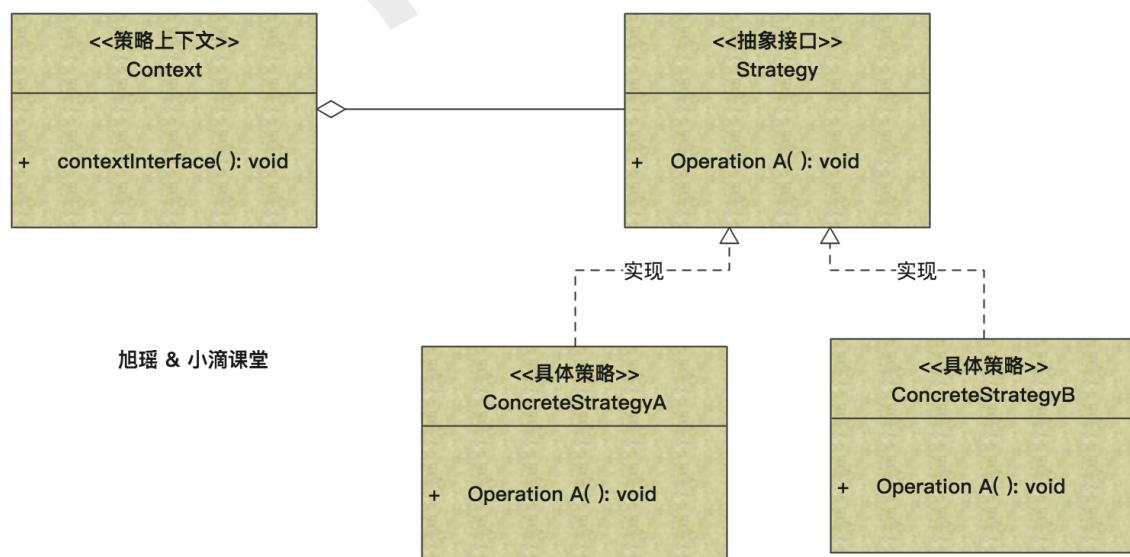
第八章 关注对象之间的通信-行为型模式应用《上》

第1集 三十六计-玩转策略模式和应用场景

简介：策略设计模式介绍和应用场景

- 策略模式(Strategy Pattern)
 - 定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换
 - 淘宝天猫双十一, 正在搞活动有打折的、有满减的、有返利的等等, 这些算法只是一种策略, 并且是随时都可能互相替换的, 我们就可以定义一组算法, 将每个算法都封装起来, 并且使它们之间可以互换
- 应用场景
 - 老王计划外出旅游, 选择骑自行车、坐汽车、飞机等, 每一种旅行方式都是一个策略
 - Java AWT中的LayoutManager, 即布局管理器
 - 如果在一个系统里面有许多类, 它们之间的区别仅在于它们的行为, 那么可以使用策略模式
 - 不希望暴露复杂的、与算法有关的数据结构, 那么可以使用策略模式来封装算法

- 角色
 - Context上下文：屏蔽高层模块对策略、算法的直接访问，封装可能存在的变化
 - Strategy策略角色：抽象策略角色，是对策略、算法家族的抽象，定义每个策略或算法必须具有的方法和属性
 - ConcreteStrategy具体策略角色：用于实现抽象策略中的操作，即实现具体的算法



第2集 策略设计模式实战之电商多场景促销活动方案

简介：策略设计模式应用实战-电商多场景促销活动方案

- 业务需求

老王面试进了大厂，是电商项目的营销活动组，负责多个营销活动，有折扣、优惠券抵扣、满减等，项目上线后，产品经理找茬，经常新增营销活动，导致代码改动多，加班严重搞的老王很恼火。

他发现这些都是活动策略，商品的价格是根据不同的活动策略进行计算的，因此用策略设计模式进行了优化，后续新增策略后只要简单配置就行了，不用大动干戈

- 编码实战

```
public class ProductOrder {
```

```
private double oldPrice;

private int userId;

private int productId;

public ProductOrder(double oldPrice, int
userId, int productId){
    this.oldPrice = oldPrice;
    this.userId = userId;
    this.productId = productId;
}

//set get 方法省略

}

/***
 * 小滴课堂,愿景: 让技术不再难学 https://xdclass.net
 *
 * @Description
 * @Author 二当家小D
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群 微信
xdclass6
 * @Version 1.0
 **/


public abstract class Strategy {
    /**
     * 根据简单订单对象, 计算商品折扣后的价格
     * @param productOrder
}
```

```
* @return
*/
public abstract double
computePrice(ProductOrder productOrder);

}

public class PromotionContext {

    private Strategy strategy;

    public PromotionContext(Strategy strategy)
    {
        this.strategy = strategy;
    }

    /**
     * 根据策略计算最终的价格
     * @param productOrder
     * @return
     */
    public double executeStrategy(ProductOrder
productOrder){
        return
strategy.computePrice(productOrder);
    }

}

public class NormalActivity extends Strategy{
```

```
    @Override
    public double computePrice(ProductOrder
productOrder) {

        return productOrder.getOldPrice();
    }

}

public class DiscountActivity extends
Strategy{

    /**
     * 具体的折扣
     */
    private double rate;

    public DiscountActivity(double rate){
        this.rate = rate;
    }

    @Override
    public double computePrice(ProductOrder
productOrder) {
        //一系列复杂的计算

        return  productOrder.getOldPrice() *
rate;
}
```

```
    }

}

public class VoucherActivity extends Strategy {

    /**
     * 传入优惠券
     */
    private double voucher;

    public VoucherActivity(double voucher){
        this.voucher = voucher;
    }

    @Override
    public double computePrice(ProductOrder productOrder) {

        if(productOrder.getOldPrice() > voucher){
            return productOrder.getOldPrice() - voucher;
        }else {
            return 0;
        }
    }
}
```

```
//使用

public static void main(String[] args) {
    ProductOrder productOrder = new
ProductOrder(800,1,32);
    PromotionContext context;
    double finalPrice;

    //不同策略算出不同的活动价格

    //没活动
    context = new PromotionContext(new
NormalActivity());
    finalPrice =
context.executeStrategy(productOrder);
    System.out.println("NormalActivity =
"+finalPrice);

    //折扣策略
    context = new PromotionContext(new
DiscountActivity(0.8));
    finalPrice =
context.executeStrategy(productOrder);
    System.out.println("DiscountActivity =
"+finalPrice);

    //优惠券抵扣
    context = new PromotionContext(new
VoucherActivity(100));
```

```
    finalPrice =  
context.executeStrategy(productOrder);  
    System.out.println("VoucherActivity =  
"+finalPrice);  
  
}
```

- 优点
 - 满足开闭原则，当增加新的具体策略时，不需要修改上下文类的代码，上下文就可以引用新的具体策略的实例
 - 避免使用多重条件判断，如果不使用策略模式可能会使用多重条件语句不利于维护，和工厂模式的搭配使用可以很好地消除代码if-else的多层嵌套（工厂模式主要是根据参数，获取不同的策略）
- 缺点
 - 策略类数量会增多，每个策略都是一个类，复用的可能性很小
 - 对外暴露了类所有的行为和算法，行为过多导致策略类膨胀
- JDK源码的应用
 - Comparator 接口常用的 compare()方法，就是一个策略设计模式的应用，把 Comparator 作为参数使用生成

不同的排序策略

```
List<Student> list = new ArrayList<>();
list.add(new Student("Anna", 15));
list.add(new Student("小D", 18));
list.add(new Student("老王", 20));

// 对伙伴的集合按年龄进行排序
Collections.sort(list, new
Comparator<Student>() {
    @Override
    public int compare(Student s1, Student
s2) {
        // 升序
        //return s1.getAge()-s2.getAge();

        // 降序
        // return s2.getAge()-s1.getAge();
    }
});
```

第3集 大象装进冰箱分几步？模板方法设计模式介绍

简介： 模板方法设计模式讲解和应用场景介绍

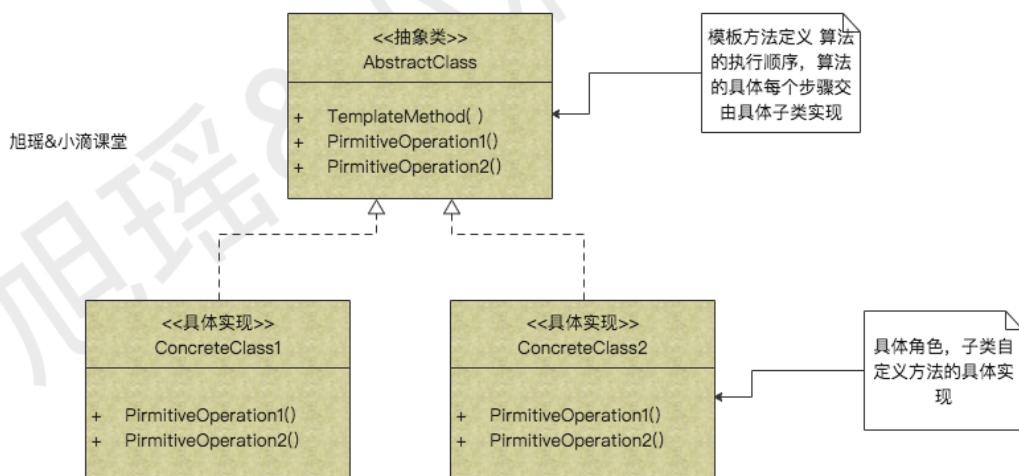
- 模板方法模式Template Method
 - 定义一个操作中的算法骨架，将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤，属于行为型模式
- 应用场景
 - javaweb里面的Servlet, HttpServlet类提供了一个service()方法，
 - 有多个子类共有逻辑相同的方法，可以考虑作为模板方法
 - 设计一个系统时知道了算法所需的关键步骤，且确定了这些步骤的执行顺序，但某些步骤的具体实现还未知，可以延迟到子类进行完成

```
/**  
 * 抽象类(Abstract Class)  
 */
```

```
public abstract class AbstractClass {  
  
    /**  
     * 模版方法  
     */  
    public void templateMethod() {  
        specificMethod();  
        abstractMethod1();  
        abstractMethod2();  
    }  
  
    /**  
     * 具体方法  
     */  
    public void specificMethod() {  
        System.out.println("抽象类中的具体方法被调用");  
    }  
  
    // 抽象方法1  
    public abstract void abstractMethod1();  
  
    // 抽象方法2  
    public abstract void abstractMethod2();  
}
```

- 角色

- 抽象模板(Abstract Template): 定义一个模板方法，这个模板方法一般是一个具体方法，给出一个顶级算法骨架，而逻辑骨架的组成步骤在相应的抽象操作中，推迟到子类实现
 - 模板方法：定义了算法的骨架，按某种顺序调用其包含的基本方法
 - 基本方法：是整个算法中的一个步骤，包括抽象方法和具体方法
 - 抽象方法：在抽象类中申明，由具体子类实现。
 - 具体方法：在抽象类中已经实现，在具体子类中可以继承或重写它
- 具体模板(Concrete Template): 实现父类所定义的一个或多个抽象方法，它们是一个顶级算法逻辑的组成步骤



第4集 项目里程碑把控案例-模板方法设计模式案例实战

简介： 模板方法设计模式讲解和应用场景介绍

- 需求背景

小滴课堂-老王成功晋升为管理者，但是团队来了很多新兵，由于团队水平参差不齐，经常有新项目进来，但整体流程很不规范。

一个项目的生命周期：需求评审-设计-开发-测试-上线-运维。整个周期里面，需求评审-设计是固定的操作，而其他步骤则流程耗时等是根据项目来定的。

因此老王梳理了一个模板，来规范化项目，他只管核心步骤和项目里程碑产出的结果，具体的工时安排和开发就让团队成员去操作

- 编码实战
- 优点
 - 扩展性好，对不变的代码进行封装，对可变的进行扩展，符合开闭原则
- 提高代码复用性 将相同部分的代码放在抽象的父类中，将不同的代码放入不同的子类中
 - 通过一个父类调用其子类的操作，通过对子类的具体实现扩展不同的行为，实现了反向控制
- 缺点
 - 每一个不同的实现都需要一个子类来实现，导致类的个数增加，会使系统变得复杂

- 模板方法模式和建造者模式区别

两者很大的交集，建造者模式比模板方法模式多了一个指挥类，该类体现的是模板方法模式中抽象类的固定算法的功能，是一个创建对象的固定算法

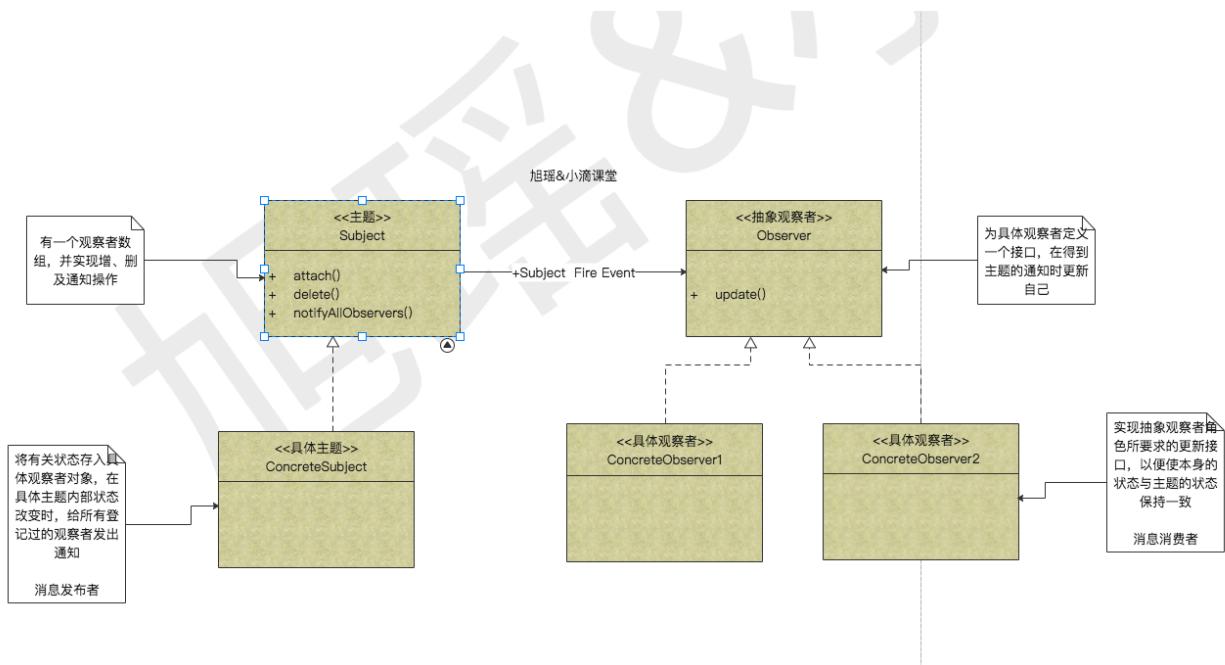
第5集 经典中的经典-Observer观察者设计模式介绍和应用

简介： Observer观察者设计模式介绍和应用

- 观察者模式
 - 定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新，也叫做发布订阅模式**Publish/Subscribe**，属于行为型模式

- 应用场景
 - 消息通知里面：邮件通知、广播通知、微信朋友圈、微博私信等，就是监听观察事件
 - 当一个对象的改变需要同时改变其它对象，且它不知道具体有多少对象有待改变的时候，考虑使用观察者模式
- 角色
 - Subject主题：持有多个观察者对象的引用，抽象主题提供了一个接口可以增加和删除观察者对象；有一个观察者数组，并实现增、删及通知操作
 - Observer抽象观察者：为具体观察者定义一个接口，在得到主题的通知时更新自己
 - ConcreteSubject具体主题：将有关状态存入具体观察者对象，在具体主题内部状态改变时，给所有登记过的观察者发出通知
 - ConcreteObserver具体观察者：实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态保

持一致



第6集 老王上班摸鱼-Observer观察者设计模式案例实战

简介： Observer观察者设计模式案例实战

- 业务需求

小滴课堂-老王，技术比较厉害，因此上班不想那么辛苦，领导又在周围，所以选了个好位置，方便监听老板的到来，当领导即将出现时老王可以立马观察到，赶紧工作。

用观察者模式帮助老王实现这个需求

- 编码

```
/**  
 * 小滴课堂,愿景：让技术不再难学  
 *  
 * @Description 消息发布者  
 * @Author 二当家小D  
 * @Remark 有问题直接联系我，源码-笔记-技术交流群  
 * @Version 1.0  
 */  
  
public class Subject {  
    private List<Observer> observerList = new  
    ArrayList<>();  
  
    /**
```

```
* 新增观察者
* @param observer
*/
public void addObserver(Observer observer){
    this.observerList.add(observer);
}

/**
*删除观察者
* @param observer
*/
public void deleteObserver(Observer
observer){
    this.observerList.remove(observer);
}

public void notifyAllObserver(){
    for(Observer
observer:this.observerList){
        observer.update();
    }
}

public interface Observer {
    /**
     * 观察到消息后进行的操作，就是响应
     */
}
```

```
    void update();
}

/**
 * 小滴课堂,愿景: 让技术不再难学 https://xdclass.net
 * @Description 消息发布者
 * @Author 二当家小D
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群 微信
xdclass6
 * @Version 1.0
**/


public class BossConcreteSubject extends
Subject {

    public void doSomething(){
        System.out.println("老板完成自己的工作");

        //还有其他操作
        System.out.println("视察公司工作情况");
        super.notifyAllObserver();

    }
}

/**
 * 小滴课堂,愿景: 让技术不再难学 https://xdclass.net
 *
 * @Description 消息的消费者
*/
```

```
* @Author 二当家小D
* @Remark 有问题直接联系我，源码-笔记-技术交流群 微信
xdclass6
* @Version 1.0
*/
public class LWConcreteObserver implements
Observer {

    @Override
    public void update() {
        System.out.println("老王发现领导到来，暂停
在线摸鱼，回归工作");
    }
}

public class AnnaConcreteObserver implements
Observer {

    @Override
    public void update() {
        System.out.println("Anna小姐姐发现领导到
来，暂停在线摸鱼，回归工作");
    }
}

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

```
//创建一个主题，老板
BossConcreteSubject subject = new
BossConcreteSubject();

//创建观察者，就是摸鱼的同事
Observer lwObserver = new
LWConcreteObserver();

//创建观察者，就是摸鱼的同事
Observer annaObserver = new
AnnaConcreteObserver();

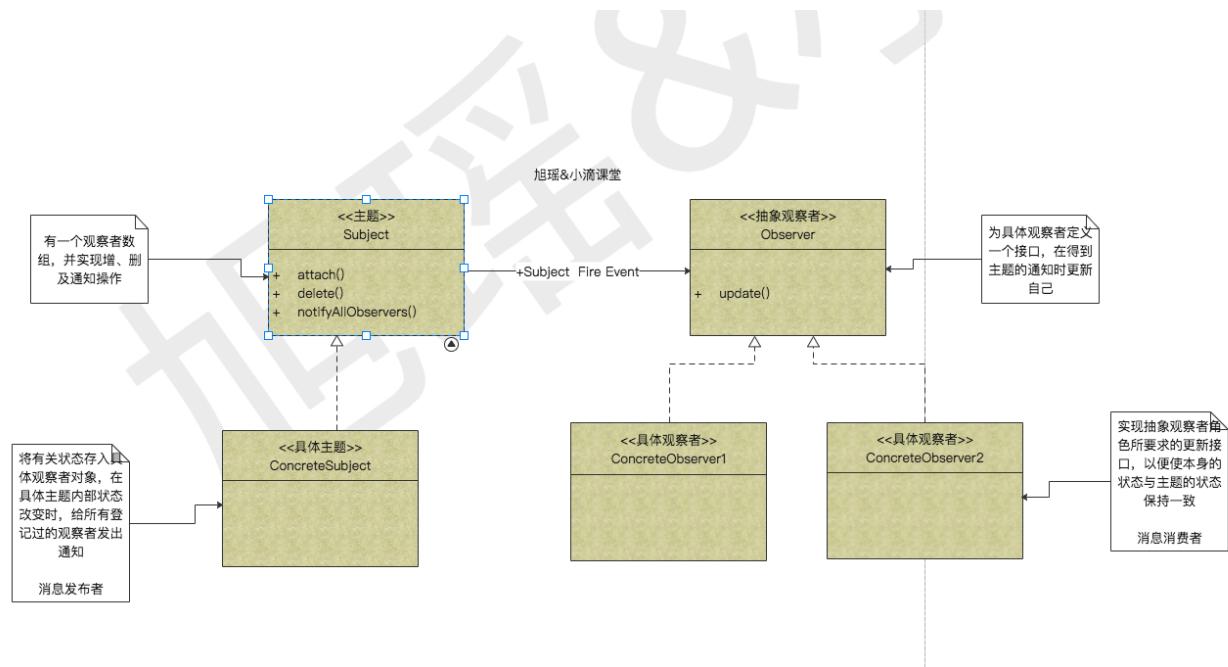
//建立对应的关系，老板这个主题被同事进行观察
subject.addObserver(lwObserver);
subject.addObserver(annaObserver);

//主题开始活动，里面会通知观察者（相当于发布消息）
subject.doSomething();

}
```

- 优点
 - 降低了目标与观察者之间的耦合关系，目标与观察者之间建立了一套触发机制
 - 观察者和被观察者是抽象耦合的

- 缺点
 - 观察者和观察目标之间有循环依赖的话，会触发它们之间进行循环调用，可能导致系统崩溃
 - 一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间



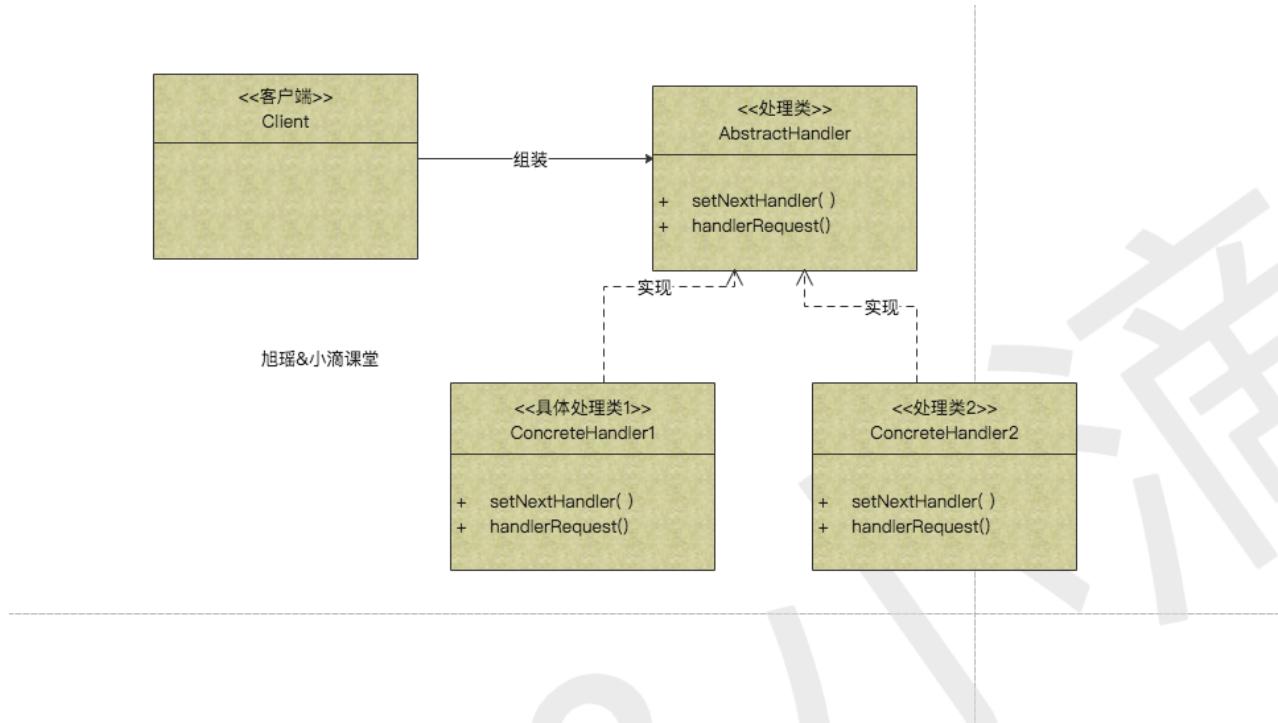
第九章 关注对象之间的通信-行为型模式应用《下》

第1集 责任链设计模式介绍和应用场景

简介：责任链设计模式介绍和应用场景

- 责任链设计模式(Chain of Responsibility Pattern)
 - 客户端发出一个请求，链上的对象都有机会来处理这一请求，而客户端不需要知道谁是具体的处理对象
 - 让多个对象都有机会处理请求，避免请求的发送者和接收者之间的耦合关系，将这个对象连成一条调用链，并沿着这条链传递该请求，直到有一个对象处理它才终止
 - 有两个核心行为：一是处理请求，二是将请求传递到下一节点
- 应用场景
 - Apache Tomcat 对 Encoding 编码处理的处理，SpringBoot里面的拦截器、过滤器链
 - 在请求处理者不明确的情况下向多个对象中的一个提交请求
 - 如果有多个对象可以处理同一个请求，但是具体由哪个对象处理是由运行时刻动态决定的，这种对象就可以使用职责链模式
- 角色
 - Handler抽象处理器：定义了一个处理请求的接口

- ConcreteHandler具体处理者： 处理所负责的请求， 可访问它的后续节点， 如果可处理该请求就处理， 否则就将该请求转发给它的后续节点



第2集 责任链设计模式案例实战-老王设计的反作弊系统的风控级别

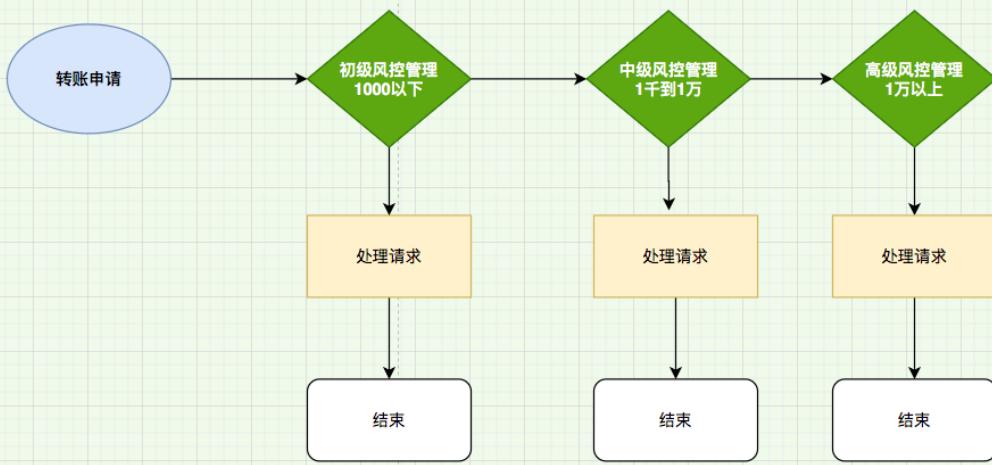
简介：责任链设计模式案例实战

- 业务需求

风控规则，就是对于每个场景，定义一些规则，来进行相应的控制，比如银行借款、支付宝提现、大额转账等 会触发不同的策略。

像互联网金融行业的话，除了公司内部政策，所处的外部环境经常发生变化，比如国家经常会出政策，这些都经常需要调整相应的风控参数和风控级别。

例子：支付宝转账，根据转账额度不同，会触发的风控级别不一样，1000元以下直接转，1千到1万需要手机号验证码，1万到以上需要刷脸验证。



● 编码实战

```
public class Request {  
    /**  
     * 类别  
     */  
    private String requestType;  
  
    /**  
     * 金额  
     */  
    private int money;  
  
    //set get方法省略  
}  
  
public enum RequestType {
```

```
    /**
     * 转账
     */
    TRANSFER,
}

/**
 * 提现
 */
CASH_OUT;

}

/**
 * 小滴课堂,愿景：让技术不再难学
https://xdclass.net
*
* @Description 风控级别抽象类
* @Author 二当家小D
* @Remark 有问题直接联系我，源码-笔记-技术交流群 微信
xdclass6
*
* @Version 1.0
*/
public abstract class RiskControlManager {
    protected String name;

    /**
     * 更严格的风控策略
     */
    protected RiskControlManager superior;
```

```
public RiskControlManager(String name){  
    this.name = name;  
}  
/**  
 * 设置更严格的风控策略  
 * @param superior  
 */  
public void setSuperior(RiskControlManager  
superior){  
    this.superior = superior;  
}  
/**  
 * 处理请求  
 * @param request  
 */  
public abstract void handlerRequest(Request  
request);  
}  
  
/**  
 * 小滴课堂,愿景: 让技术不再难学  
https://xdclass.net  
*  
* @Description 风控级别抽象类  
* @Author 二当家小D  
* @Remark 有问题直接联系我, 源码-笔记-技术交流群 微信  
xdclass6  
* @Version 1.0  
**/
```

```
public class FirstRiskControlManager extends RiskControlManager {

    public FirstRiskControlManager(String name) {
        super(name);
    }

    /**
     * 1000元以内可以直接处理
     * @param request
     */
    @Override
    public void handlerRequest(Request request) {
        if(RequestType.valueOf(request.getRequestType())
        )!=null && request.getMoney()<=1000){
            System.out.println("普通操作，输入支付密码即可");
            System.out.println(name+":"+request.getRequestType() + "， 金额:"+request.getMoney() +" 处理完成");
        }else {
    }
```

```
//下个节点进行处理
    if(superior!=null){
        superior.handlerRequest(request);
    }
}

public class SecondRiskControlManager extends RiskControlManager {
    public SecondRiskControlManager(String name) {
        super(name);
    }

    /**
     * 处理 1千到1万之间
     * @param request
     */
    @Override
    public void handlerRequest(Request request)
{
```

```
if(RequestType.valueOf(request.getRequestType()
)!=null && request.getMoney()>1000 &&
request.getMoney()<10000){

    System.out.println("稍大额操作，输入支
付密码+短信验证码即可");

System.out.println(name+"：" +request.getRequestT
ype() + "， 金额：" +request.getMoney() + " 处理完
成");

}else {

    //下个节点进行处理
    if(superior!=null){

superior.handlerRequest(request);

    }

}

}

public class ThirdRiskControlManager extends
RiskControlManager {

    public ThirdRiskControlManager(String name)
{
```

```
super(name);  
}  
  
@Override  
public void handlerRequest(Request request)  
{  
  
if(RequestType.valueOf(request.getRequestType()  
)!=null && request.getMoney()>10000){  
  
    System.out.println("大额操作，输入支付  
密码+验证码+人脸识别 " );  
  
    System.out.println(name+"："+request.getRequestT  
ype() + "， 金额："+request.getMoney() +" 处理完  
成");  
  
}else {  
    //下个节点进行处理  
    if(superior!=null){  
  
superior.handlerRequest(request);  
    }  
}  
}  
  
}  
  
//使用
```

```
public static void main(String[] args) {  
  
    RiskControlManager firstControlManager  
= new FirstRiskControlManager("初级风控");  
  
    RiskControlManager secondControlManager  
= new SecondRiskControlManager("中级风控");  
  
    RiskControlManager thirdControlManager  
= new ThirdRiskControlManager("高级风控");  
  
    //形成调用链  
  
    firstControlManager.setSuperior(secondControlMa  
nager);  
  
    secondControlManager.setSuperior(thirdControlMa  
nager);  
  
    //使用  
    Request request1 = new Request();  
  
    request1.setRequestType(RequestType.CASH_OUT.na  
me());  
    request1.setMoney(20000);  
  
    firstControlManager.handlerRequest(request1);
```

}

- 优点
 - 客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者降低了耦合度
 - 通过改变链内的调动它们的次序，允许动态地新增或者删除处理类，比较很方便维护
 - 增强了系统的可扩展性，可以根据需要增加新的请求处理类，满足开闭原则
 - 每个类只需要处理自己该处理的工作，明确各类的责任范围，满足单一职责原则
- 缺点
 - 处理都分散到了单独的职责对象中，每个对象功能单一，要把整个流程处理完，需要很多的职责对象，会产生大量的细粒度职责对象
 - 不能保证请求一定被接收；
 - 如果链路比较长，系统性能将受到一定影响，而且在进行代码调试时不太方便
- 日志处理级别

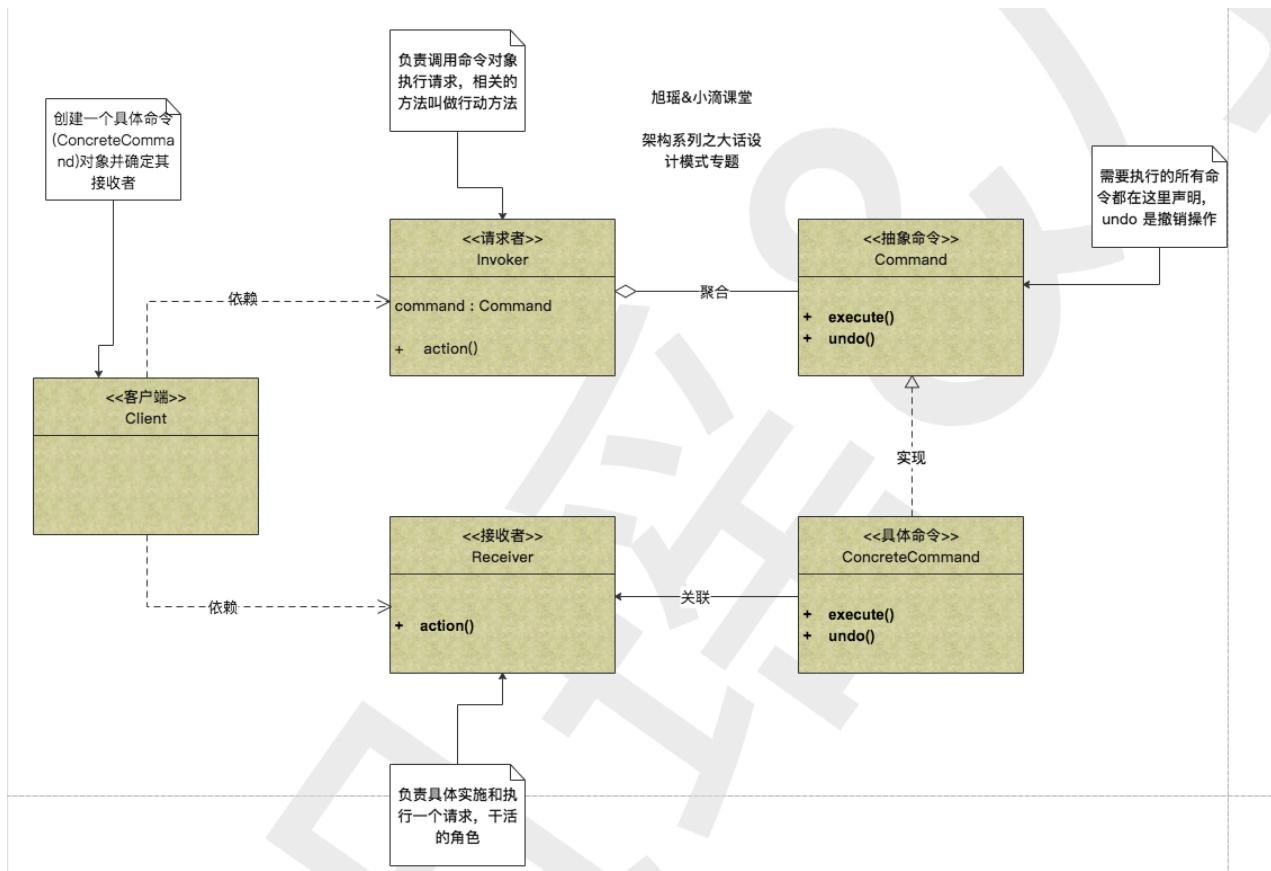
- debug->info->warning->error

第3集 Command Pattern命令设计模式介绍和应用场景

简介：命令设计模式介绍和应用场景

- 命令设计模式（Command Pattern）
 - 请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的对象，并把该命令传给相应的对象执行命令，属于行为型模式
 - 命令模式是一种特殊的策略模式，体现的是多个策略执行的问题，而不是选择的问题
- 应用场景

- 只要是你认为是命令的地方，就可以采用命令模式
 - 日常每个界面、按钮、键盘 事件操作都是 命令设计模式
-
- 角色
 - 抽象命令(Command): 需要执行的所有命令都在这里声明
 - 具体命令(ConcreteCommand): 定义一个接收者和行为之间的弱耦合，实现execute()方法，负责调用接收者的相应操作， execute()方法通常叫做执行方法。
 - 接受者(Receiver): 负责具体实施和执行一个请求，干活的角色，命令传递到这里是应该被执行的，实施和执行请求的方法叫做行动方法
 - 请求者(Invoker): 负责调用命令对象执行请求，相关的方法叫做行动方法
 - 客户端(Client): 创建一个具体命令(ConcreteCommand)对象并确定其接收者。



//接受者，命令执行者

```

public class Receiver {
    public void doSomething() {
        System.out.println("Receiver---doSomething");
    }
}

```

//抽象命令

```

public interface Command {
    /**
     * 执行动作
     */
}

```

```
    void execute();  
}  
  
//具体命令  
public class ConcreteCommand implements Command  
{  
    /**  
     * 对哪个receiver类进行命令处理  
     */  
    private Receiver receiver;  
  
    public ConcreteCommand(Receiver receiver) {  
        this.receiver = receiver;  
    }  
  
    /**  
     * 必须实现一个命令  
     */  
    @Override  
    public void execute() {  
        System.out.println("ConcreteCommand---  
execute");  
        receiver.doSomething();  
    }  
}  
  
//请求者  
public class Invoker {
```

```
private Command command;

public Invoker(Command command){

    this.command = command;
}

/**
 * 执行命令
 */
public void action(){

    this.command.execute();
}

}

//使用

public static void main(String[] args) {
    //创建接收者
    Receiver receiver = new Receiver();
    //创建命令对象，设定它的接收者
    Command command = new
ConcreteCommand(receiver);

    //创建请求者，把命令对象设置进去
    Invoker invoker = new Invoker(command);
    //执行方法
    invoker.action();
```

}

第4集 命令设计模式之智能家居控制案例实战

简介：命令设计模式案例实战

- 业务需求

小滴课堂老王-搬新家了，他想实现智能家居，开发一个app，可以控制家里的家电，比如控制空调的开关、加热、制冷 等功能

利用命令设计模式，帮老王完成这个需求，注意：动作请求者就是手机app，动作的执行者是家电的不同功能

- 编码

```
/**
```

```
* 小滴课堂,愿景: 让技术不再难学
https://xdclass.net

*
* @Description 命令执行者
* @Author 二当家小D
* @Remark 有问题直接联系我, 源码-笔记-技术交流群 微信
xdclass6
* @Version 1.0
**/


public class ConditionReceiver {

    public void on(){
        System.out.println("空调开启了");
    }

    public void off(){
        System.out.println("空调关闭了");
    }

    public void cool(){
        System.out.println("空调开始制冷");
    }

    public void warm(){
        System.out.println("空调开始制暖");
    }
}
```

```
public interface Command {  
  
    /**  
     * 执行动作  
     */  
    void execute();  
  
}  
  
/**  
 * 小滴课堂,愿景: 让技术不再难学  
https://xdclass.net  
*  
* @Description 具体命令  
* @Author 二当家小D  
* @Remark 有问题直接联系我, 源码-笔记-技术交流群 微信  
xdclass6  
* @Version 1.0  
**/  
public class OnCommand implements Command{  
  
    /**  
     * 对哪个receiver 进行命令处理  
     */  
    private ConditionReceiver receiver;  
  
    public OnCommand(ConditionReceiver  
receiver){  
        this.receiver = receiver;  
    }  
}
```

```
}

/**
 * 必须实现一个命令的调用
 */
@Override
public void execute() {

    System.out.println("OnCommand ->
execute");
    receiver.on();

}

}

public class OffCommand implements Command{

    /**
     * 对哪个receiver 进行命令处理
     */
    private ConditionReceiver receiver;

    public OffCommand(ConditionReceiver
receiver){
        this.receiver = receiver;
    }

    /**
     * 必须实现一个命令的调用
    
```

```
 */
@Override
public void execute() {

    System.out.println("OffCommand ->
execute");
    receiver.off();

}

public class CoolCommand implements Command{

    /**
     * 对哪个receiver 进行命令处理
     */
    private ConditionReceiver receiver;

    public CoolCommand(ConditionReceiver
receiver) {
        this.receiver = receiver;
    }

    /**
     * 必须实现一个命令的调用
     */
    @Override
    public void execute() {
```

```
        System.out.println("CoolCommand ->
execute");
        receiver.cool();

    }

}

/**
 * 小滴课堂,愿景：让技术不再难学
https://xdclass.net
 *
 * @Description 请求者
 * @Author 二当家小D
 * @Remark 有问题直接联系我，源码-笔记-技术交流群 微信
xdclass6
 * @Version 1.0
 */
public class AppInvoker {

    private Command onCommand;
    private Command offCommand;

    private Command coolCommand;
    private Command warmCommand;

    public void setOnCommand(Command onCommand)
{
```

```
        this.onCommand = onCommand;
    }

    public void setOffCommand(Command offCommand) {
        this.offCommand = offCommand;
    }

    public void setCoolCommand(Command coolCommand) {
        this.coolCommand = coolCommand;
    }

    public void setWarmCommand(Command warmCommand) {
        this.warmCommand = warmCommand;
    }

    /**
     * 开机
     */
    public void on(){
        onCommand.execute();
    }

    /**
     * 关机
     */
}
```

```
public void off(){
    offCommand.execute();
}

public void warm(){
    warmCommand.execute();
}

public void cool(){
    coolCommand.execute();
}

}

//使用
public static void main(String[] args) {
    //创建接受者， 空调就是接受者
    ConditionReceiver receiver = new
ConditionReceiver();

    //创建命令对象， 设置命令的接受者
    Command onCommand = new
OnCommand(receiver);

    Command offCommand = new
OffCommand(receiver);

    Command coolCommand = new
CoolCommand(receiver);
```

```
    Command warmCommand = new
WarmCommand(receiver);

        //创建请求者，把命令对象设置进去，app就是请求
发起者

    AppInvoker appInvoker = new
AppInvoker();
    appInvoker.setOnCommand(onCommand);
    appInvoker.setOffCommand(offCommand);
    appInvoker.setCoolCommand(coolCommand);
    appInvoker.setWarmCommand(warmCommand);

    appInvoker.on();
    System.out.println();

    appInvoker.cool();
    System.out.println();

    appInvoker.warm();
    System.out.println();

    appInvoker.off();

}
```

- 优点

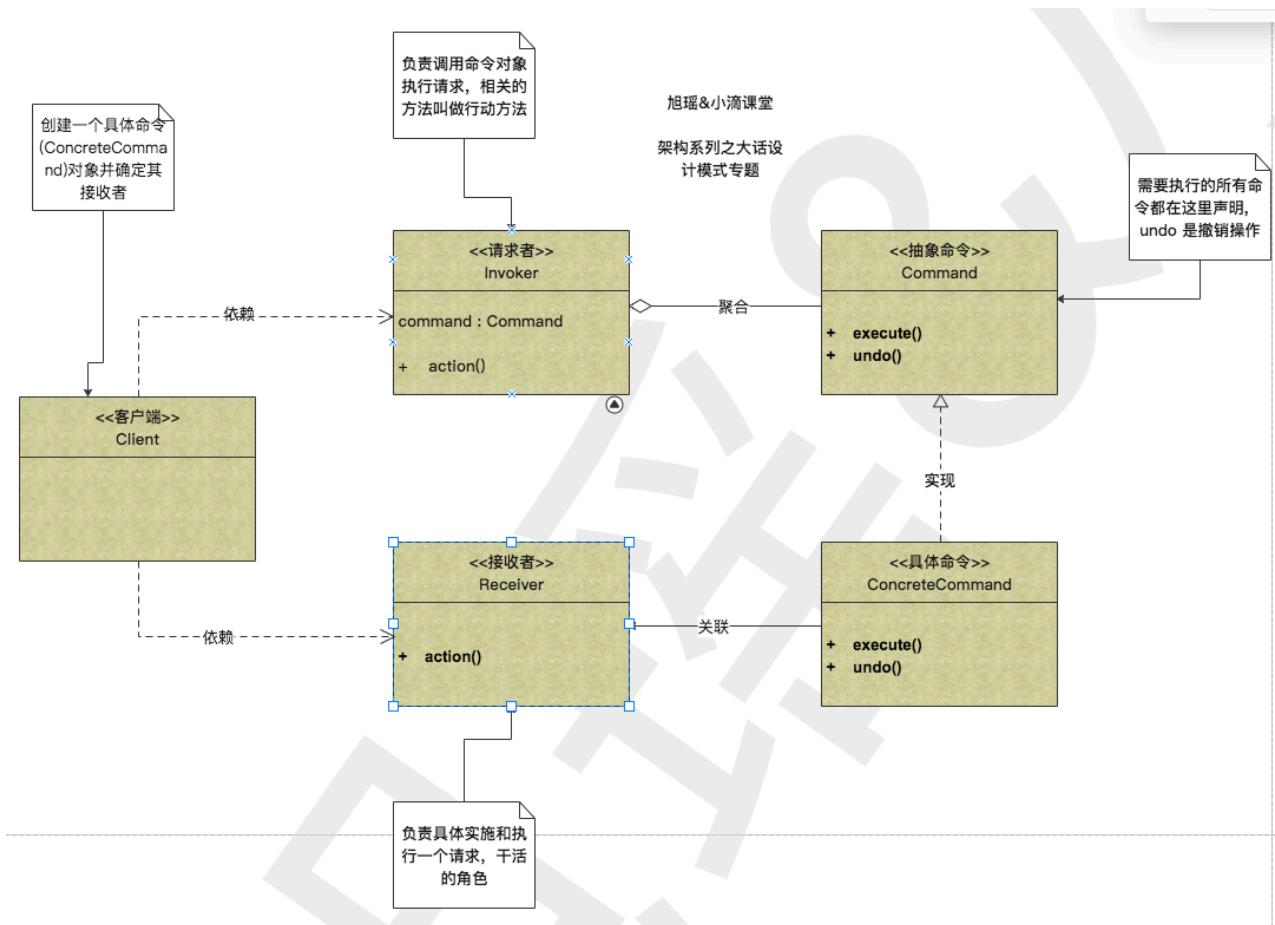
- 调用者角色与接收者角色之间没有任何依赖关系，不需

要了解到底是哪个接收者执行，降低了系统耦合度

- 扩展性强，新的命令可以很容易添加到系统中去。

- 缺点

- 过多的命令模式会导致某些系统有过多的具体命令类



第5集 IteratorPattern迭代器设计模式介绍和应用场景

简介：迭代器设计模式介绍和应用场景

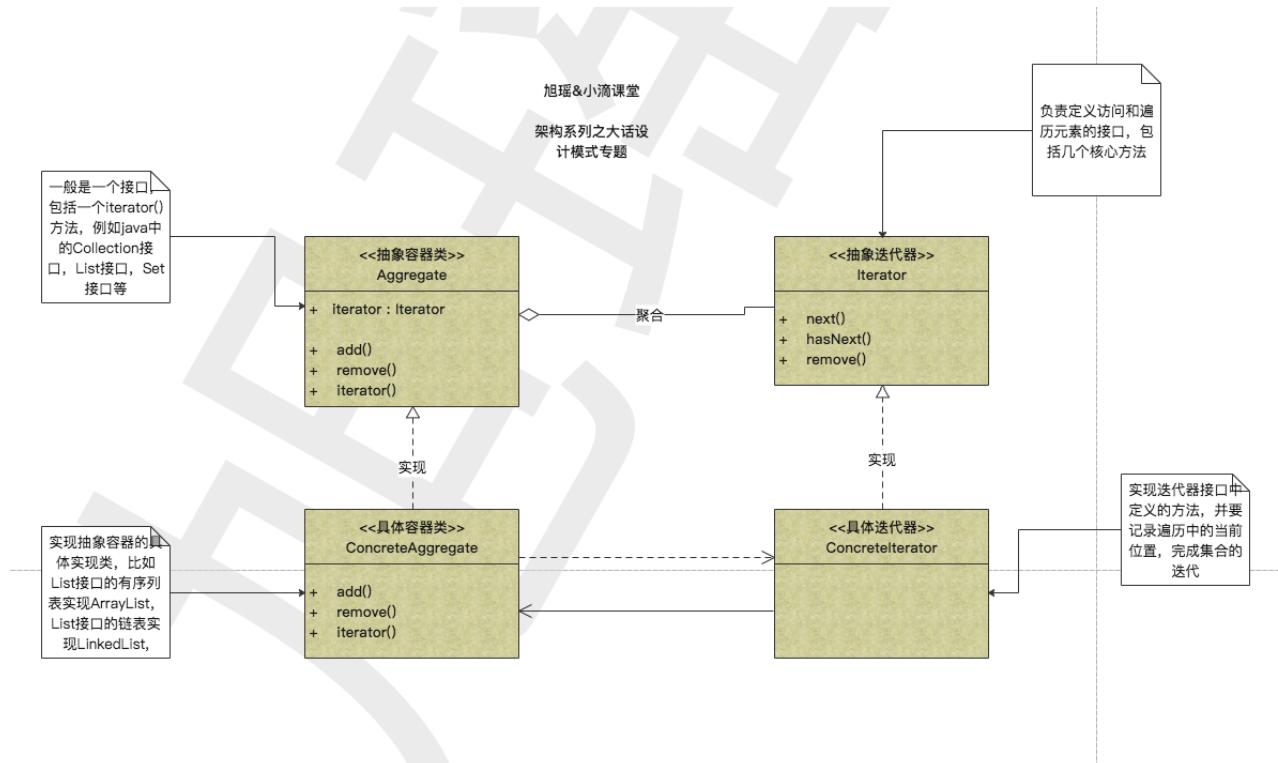
- 迭代器设计模式 (Iterator Pattern)
 - 提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部实现, 属于行为型模式
 - 应该是java中应用最多的设计模式之一

提到迭代器, 想到它是与集合相关的, 集合也叫容器, 可以将集合看成是一个可以包容对象的容器, 例如List, Set, Map, 甚至数组都可以叫做集合, 迭代器的作用就是把容器中的对象一个一个地遍历出来

- 应用场景
 - 一般来说, 迭代器模式是与集合是共存的, 只要实现一个集合, 就需要同时提供这个集合的迭代器, 就像java 中的Collection, List、Set、Map等都有自己的迭代器
 - JAVA 中的 iterator迭代器
- 角色
 - 抽象容器 (Aggregate) : 提供创建具体迭代器角色的

接口，一般是接口，包括一个iterator()方法，例如java中的Collection接口，List接口，Set接口等。

- 具体容器角色（ConcreteAggregate）：实现抽象容器的具体实现类，比如List接口的有序列表实现ArrayList，List接口的链表实现LinkedList，Set接口的哈希列表的实现HashSet等。
- 抽象迭代器角色（Iterator）：负责定义访问和遍历元素的接口，包括几个核心方法，取得下一个元素的方法next()，判断是否遍历结束的方法isDone()（或者叫hasNext()），移除当前对象的方法remove()
- 具体迭代器角色（ConcretelIterator）：实现迭代器接口中定义的方法，并要记录遍历中的当前位置，完成集合的迭代

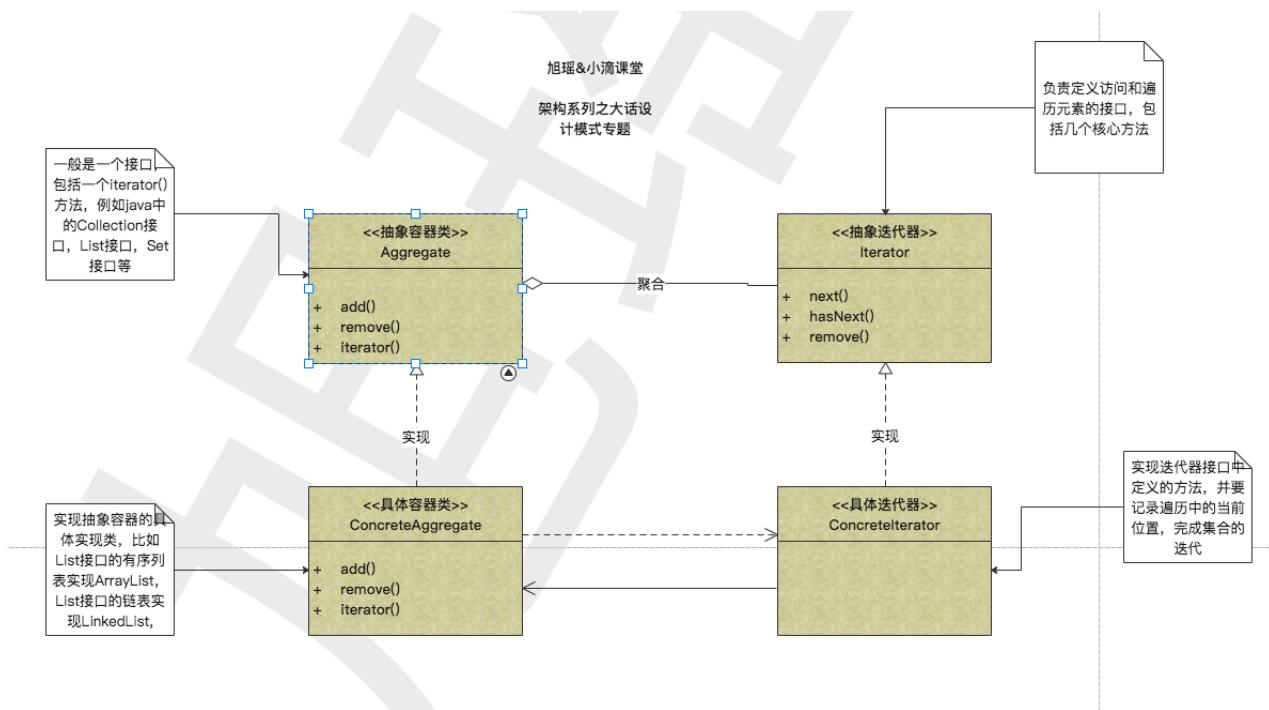


第6集 迭代器模式案例实战之自定义List集合容器

简介：迭代器设计模式案例实战

- 案例需求

自定义一个集合容器，并实现里面的迭代器功能，List集合容器的简化版本



- 编码实战

```
/**
```

```
* 小滴课堂，愿景：让技术不再难学
```

```
*  
 * @Description 抽象迭代器  
 * @Author 二当家小D  
 * @Remark 有问题直接联系我，源码-笔记-技术交流群  
 * @Version 1.0  
 **/  
  
public interface Iterator {  
  
    /**  
     * 获取下个元素  
     * @return  
     */  
    Object next();  
  
    /**  
     * 是否有下一个  
     * @return  
     */  
    boolean hasNext();  
    /**  
     * 删除元素  
     * @param obj  
     * @return  
     */  
    Object remove(Object obj);  
  
}
```

```
/**  
 * 小滴课堂,愿景: 让技术不再难学  
 *  
 * @Description 具体的迭代器  
 * @Author 二当家小D  
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群  
 * @Version 1.0  
 */  
  
public class ConcreteIterator implements  
Iterator {  
  
    private List list;  
  
    private int index = 0;  
  
    public ConcreteIterator(List list){  
        this.list = list;  
    }  
  
    @Override  
    public Object next() {  
  
        Object obj = null;  
        if(this.hasNext()){  
            obj = this.list.get(index);  
            index++;  
        }  
        return obj;  
    }  
}
```

```
        }

        return obj;
    }

@Override
public boolean hasNext() {
    if(index == list.size()){
        return false;
    }
    return true;
}

@Override
public Object remove(Object obj) {
    return list.remove(obj);
}

}

/**
 * 小滴课堂,愿景：让技术不再难学
 *
 * @Description 抽象容器建立了
 * @Author 二当家小D
 * @Remark 有问题直接联系我，源码-笔记-技术交流群
 * @Version 1.0
 */
public interface ICollection {
```

```
    void add(Object obj);

    void remove(Object obj);

    Iterator iterator();

}

/**
 * 小滴课堂,愿景: 让技术不再难学
 *
 * @Description 容器简化版
 * @Author 二当家小D
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群
 * @Version 1.0
 */

```

```
public class MyCollection implements
ICollection {

    private List list = new ArrayList();

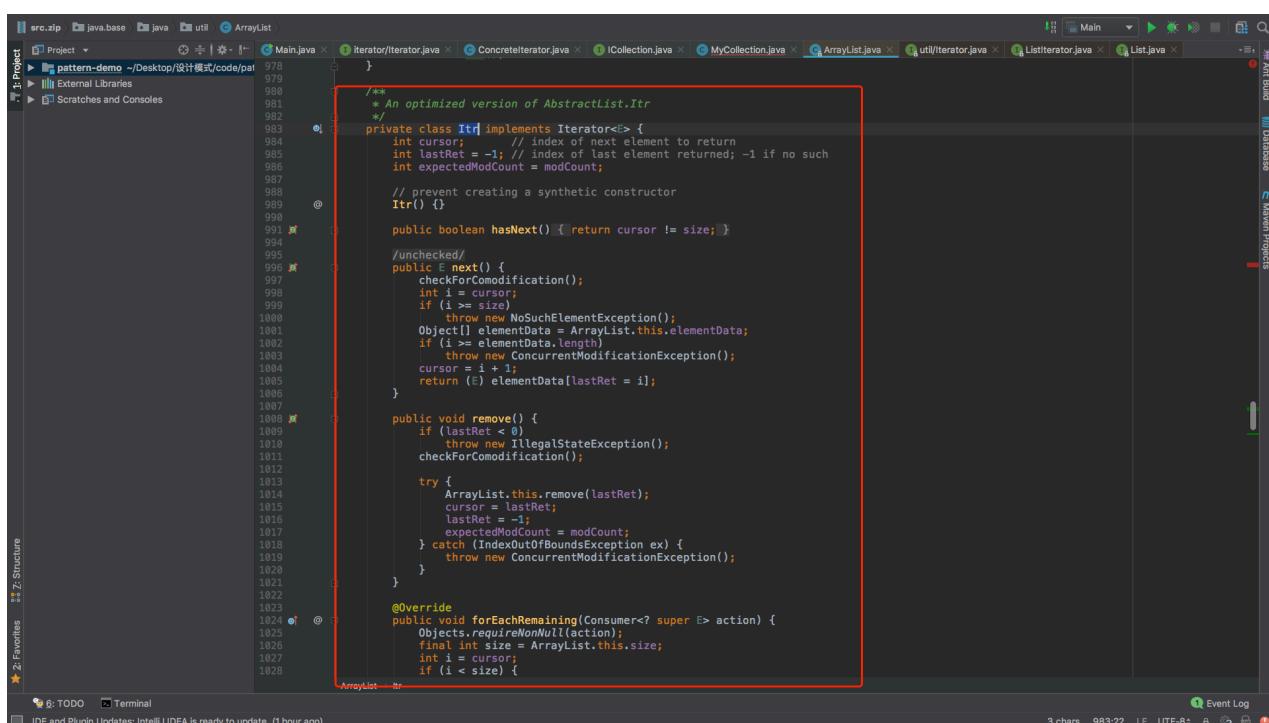
    @Override
    public void add(Object obj) {
        list.add(obj);
    }

    @Override
    public void remove(Object obj) {
```

```
        list.remove(obj);  
    }  
  
    @Override  
    public Iterator iterator() {  
        return new ConcreteIterator(list);  
    }  
}  
  
//使用  
public static void main(String[] args) {  
  
    ICollection collection = new  
MyCollection();  
    collection.add("小滴课堂老王");  
    collection.add("小滴课堂Anna小姐姐");  
    collection.add("小滴课堂二当家小D");  
    collection.add("小滴课堂刘一手");  
    collection.add("小滴课堂老帆");  
  
    Iterator iterator =  
collection.iterator();  
  
    while (iterator.hasNext()) {  
        Object obj = iterator.next();  
        System.out.println(obj);  
    }  
}
```

- 优点
 - 可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据
 - 支持以不同的方式遍历一个聚合对象
- 缺点
 - 对于比较简单的遍历（像数组或者有序列表），使用迭代器方式遍历较为繁琐
 - 迭代器模式在遍历的同时更改迭代器所在的集合结构会导致出现异常

- JDK源码 ArrayList的迭代器例子



The screenshot shows the IntelliJ IDEA interface with the file `Main.java` open. A red box highlights the `Itr` class definition within the `AbstractList` class. The code implements the `Iterator` interface and provides optimized methods for iteration.

```

    /**
     * An optimized version of AbstractList.Itr
     */
    private class Itr implements Iterator<E> {
        int cursor; // index of next element to return
        int lastRet = -1; // index of last element returned; -1 if no such
        int expectedModCount = modCount;

        // prevent creating a synthetic constructor
        Itr() {}

        public boolean hasNext() { return cursor != size; }

        @unchecked
        public E next() {
            checkForComodification();
            int i = cursor;
            if (i >= size)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
        }

        public void remove() {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();

            try {
                ArrayList.this.remove(lastRet);
                cursor = lastRet;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }

        @Override
        public void forEachRemaining(Consumer<? super E> action) {
            Objects.requireNonNull(action);
            final int size = ArrayList.this.size;
            int i = cursor;
            if (i < size) {
        }
    }
}

```



旭瑤&小滴课堂 愿景："让编程不再难学，让

技术与生活更加有趣" 更多架构课程请访问 xdclass.net

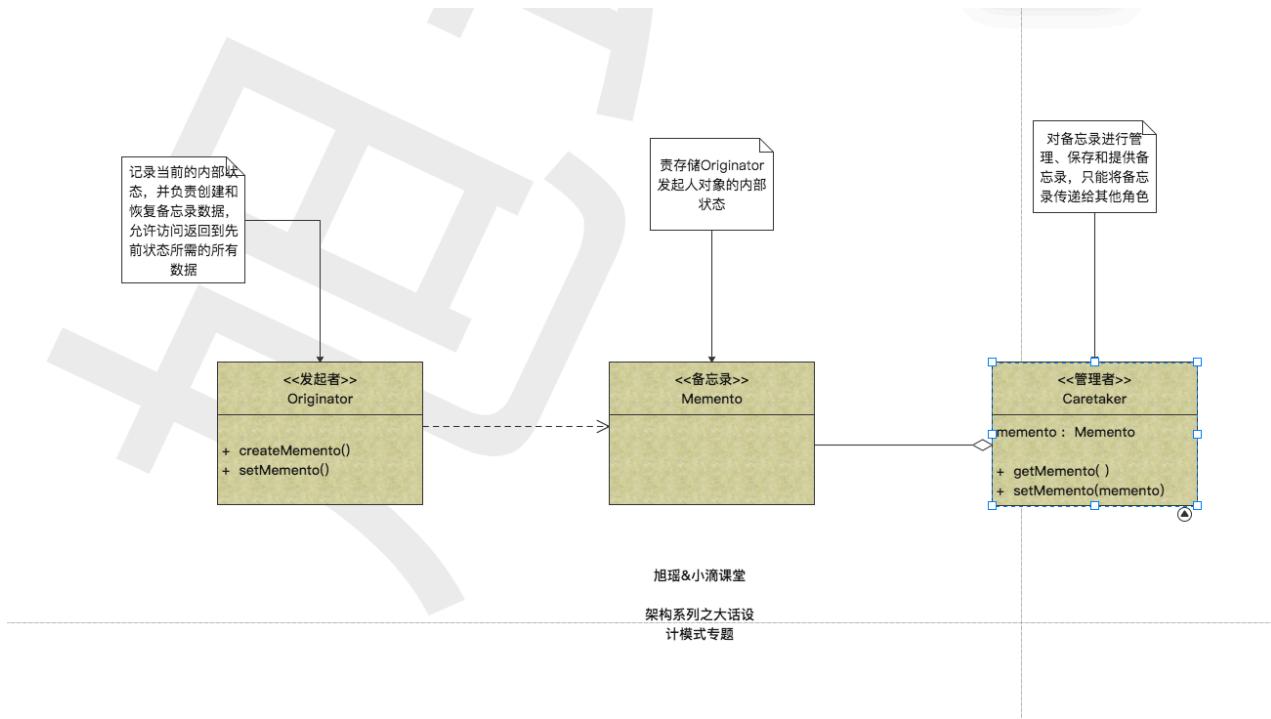
第十章 关注对象之间的通信-行为型模式 终极篇

第1集 备忘录设计模式介绍和应用场景

简介：备忘录设计模式介绍和应用场景

- 备忘录设计模式(Memento Pattern)
 - 在不破坏封闭的前提下，捕获一个对象的内部状态，保存对象的某个状态，以便在适当的时候恢复对象，又叫做快照模式，属于行为模式
 - 备忘录模式实现的方式需要保证被保存的对象状态不能被对象从外部访问，

- 应用场景
 - 玩游戏的时候肯定有存档功能，下一次登录游戏时可以从上次退出的地方继续游戏
 - 棋盘类游戏的悔棋、数据库事务回滚
 - 需要记录一个对象的内部状态时，为了允许用户取消不确定或者错误的操作，能够恢复到原先的状态
 - 提供一个可回滚的操作，如ctrl+z、浏览器回退按钮
- 角色
 - Originator: 发起者，记录当前的内部状态，并负责创建和恢复备忘录数据，允许访问返回到先前状态所需的所有数据，可以根据需要决定Memento存储自己的哪些内部状态
 - Memento: 备忘录，负责存储Originator发起人对象的内部状态，在需要的时候提供发起人需要的内部状态
 - Caretaker: 管理者，对备忘录进行管理、保存和提供备忘录，只能将备忘录传递给其他角色
 - Originator 和 Memento属性类似



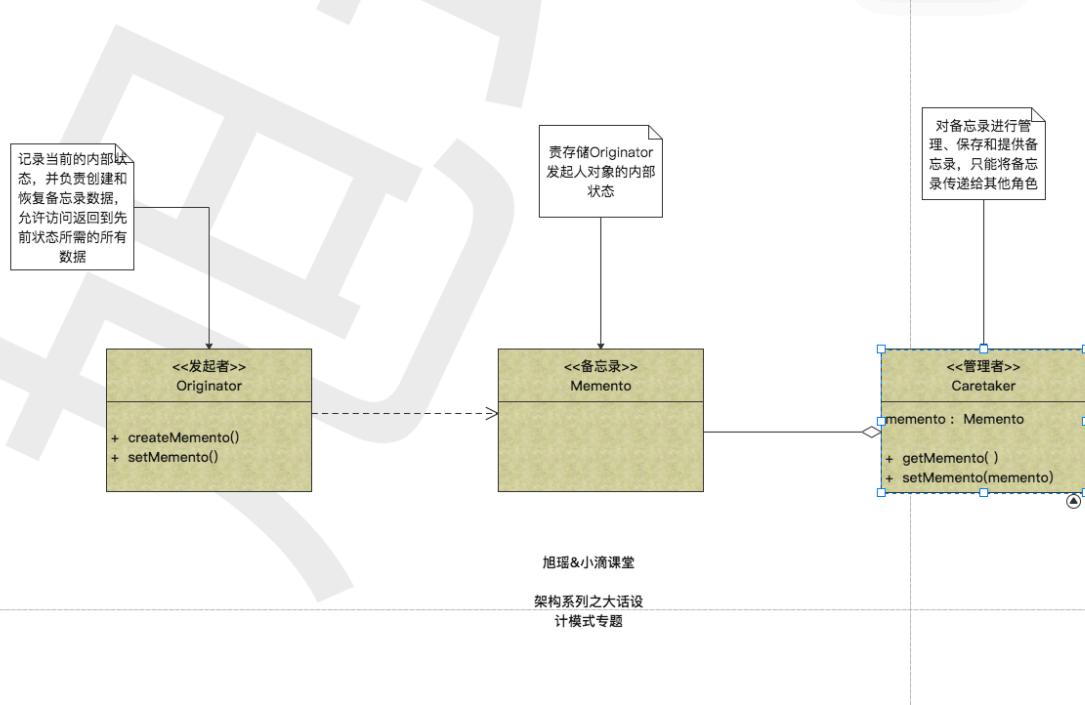
第2集 备忘录设计模式实战之游戏存档恢复

简介：备忘录设计模式案例实战

- 需求背景

小滴课堂 - 二当家小D 开发了一个游戏存档功能 拳皇97，无限生命，每次快要死的时候就恢复成刚开始的状态

使用备忘录设计模式帮他完成



```
/*
 * 小滴课堂,愿景: 让技术不再难学 https://xdclass.net
 *
 * @Description 发起者, Originator 和 Memento属性类似
 * @Author 二当家小D
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群 微信xdclass6
 * @Version 1.0
 */
```

```
public class RoleOriginator {
```

```
/**  
 * 生命力，会下降  
 */  
private int live = 100;  
  
/**  
 * 攻击力，会上涨  
 */  
private int attack = 50;  
  
public int getLive() {  
    return live;  
}  
  
public void setLive(int live) {  
    this.live = live;  
}  
  
public int getAttack() {  
    return attack;  
}  
  
public void setAttack(int attack) {  
    this.attack = attack;  
}  
  
public void display(){  
    System.out.println("开始====");  
    System.out.println("生命力: "+live);
```

```
        System.out.println("攻击力: "+attack);
        System.out.println("结束=====");
    }

    public void fight(){
        //攻击力会上涨
        this.attack = attack+2;

        //打架生命力会下降
        this.live = live - 10;
    }

}

/***
 * 保存快照，存储状态
 * @return
 */
public RoleStateMemento saveState(){
    return new
RoleStateMemento(live,attack);
}

/***
 * 恢复
 */

```

```
    public void recoveryState(RoleStateMemento
memento) {
        this.attack = memento.getAttack();
        this.live = memento.getLive();
    }

}

/***
 * 小滴课堂,愿景: 让技术不再难学 https://xdclass.net
 *
 * @Description 状态管理者
 * @Author 二当家小D
 * @Remark 有问题直接联系我, 源码-笔记-技术交流群
 * @Version 1.0
 **/


public class RoleStateCaretaker {

    private RoleStateMemento memento;

    public RoleStateMemento getMemento() {
        return memento;
    }

    public void setMemento(RoleStateMemento
memento) {
        this.memento = memento;
    }
}
```

```
    }

}

/**
 * 小滴课堂,愿景：让技术不再难学
 *
 * @Description 快照，备忘录
 * @Author 二当家小D
 * @Remark 有问题直接联系我，源码-笔记-技术交流群
 * @Version 1.0
 **/

public class RoleStateMemento {

    /**
     * 生命力，会下降
     */
    private int live;

    /**
     * 攻击力，会上涨
     */
    private int attack ;

    public RoleStateMemento(int live, int
attack) {
        this.live = live;
        this.attack = attack;
    }
}
```

```
}

public int getLive() {
    return live;
}

public void setLive(int live) {
    this.live = live;
}

public int getAttack() {
    return attack;
}

public void setAttack(int attack) {
    this.attack = attack;
}

}

//使用
public static void main(String[] args) {

    RoleOriginator role = new
RoleOriginator();
    role.display();
    role.fight();
    role.display();

    System.out.println("保存上面的快照");
}
```

```
RoleStateCaretaker caretaker = new  
RoleStateCaretaker();  
caretaker.setMemento(role.saveState());  
  
role.fight();  
role.fight();  
role.fight();  
role.fight();  
role.display();  
  
System.out.println("准备恢复快照");  
  
role.recoveryState(caretaker.getMemento());  
role.display();  
}
```

- 优点

- 给用户提供了一种可以恢复状态的机制
- 实现了信息的封装，使得用户不需要关心状态的保存细节

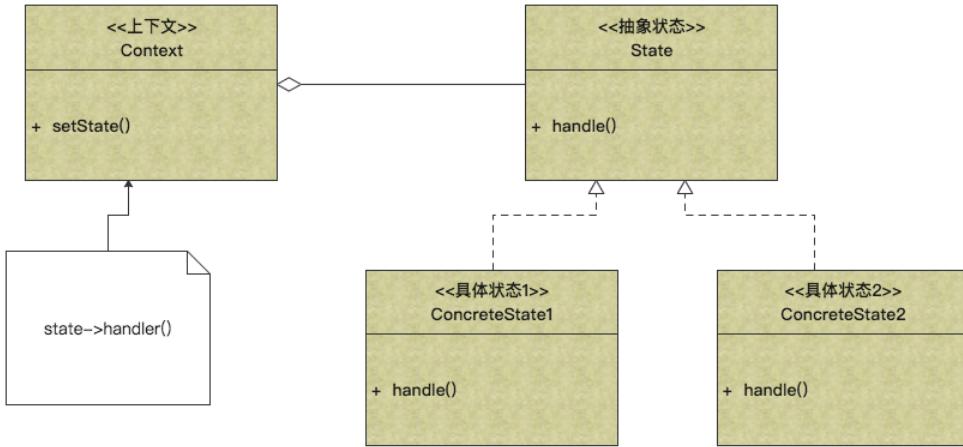
- 缺点

- 消耗更多的资源，而且每一次保存都会消耗一定的内存

第3集 状态设计模式介绍和应用场景

简介：状态设计模式介绍和应用场景

- 状态设计模式(State Pattern)
 - 对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为，属于行为型模式
 - 允许一个对象在其内部状态改变时改变它的行为
 - 状态模式是策略模式的孪生兄弟，它们的UML图是一样的，但实际上解决的是不同情况的两种场景问题
 - 工作中用的不多，基本策略模式比较多



• 应用场景

- 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为
- 代码中包含大量与对象状态有关的条件语句，比如一个操作中含有庞大的多分支的条件if else语句，且这些分支依赖于该对象的状态
- 电商订单状态：未支付、已支付、派送中，收货完成等状态，各个状态下处理不同的事情

• 角色

- Context 上下文: 定义了客户程序需要的接口并维护一个具体状态角色的实例，将与状态相关的操作委托给当前的Concrete State对象来处理

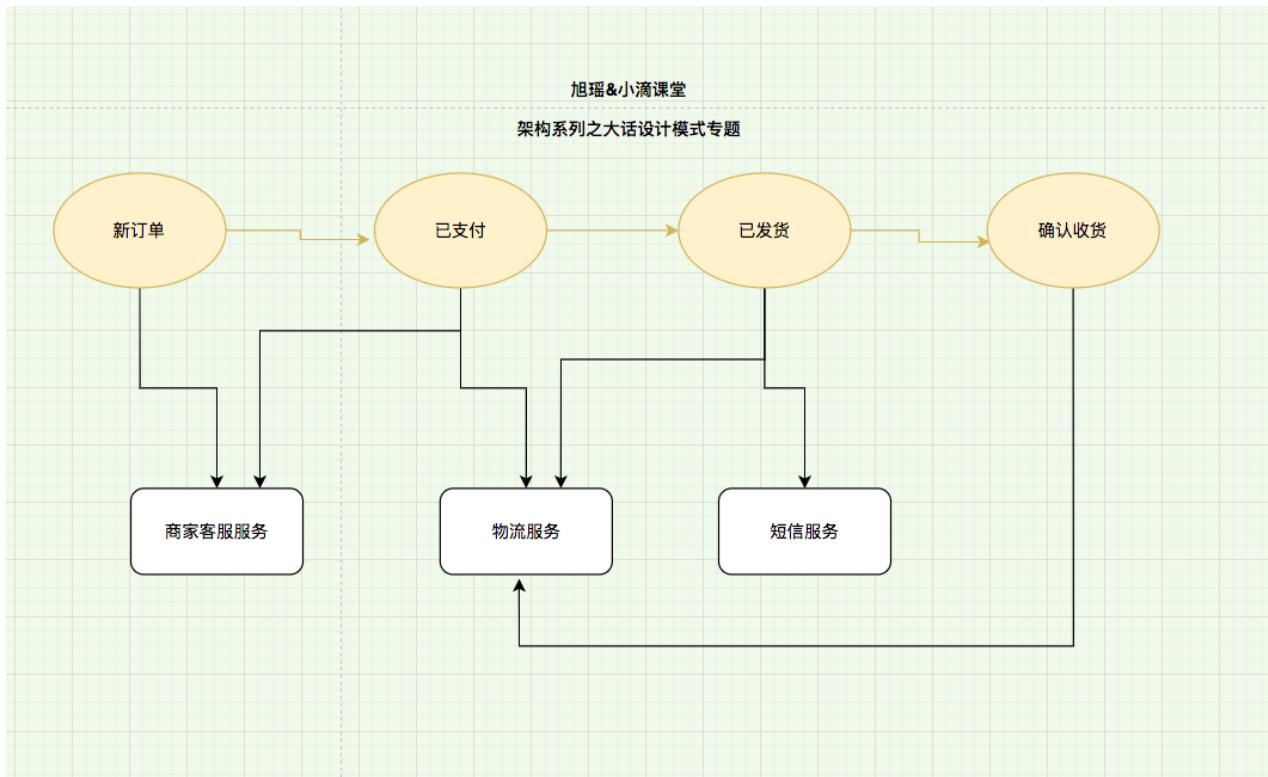
- State 抽象状态类: 定义一个接口以封装与Context的一个特定状态相关的行为。
- ConcreteState具体状态类: 实现抽象状态定义的接口。

第4集 状态设计模式案例实战之电商系统订单状态处理

简介：状态设计模式案例实战

- 业务需求

电商订单状态流转，每步都有不同的操作内容：新建订单/已支付/已发货/确认收货



- 编码

```

public interface State {
    void handle();
}

public class NewOrderState implements State{
    @Override
    public void handle() {
        System.out.println("新订单, 未支付");
        System.out.println("调用商户客服服务, 有新
订单\n");
    }
}

```

```
public class PayOrderState implements State{

    @Override
    public void handle() {
        System.out.println("新订单已经支付");
        System.out.println("调用商户客服服务，订单
已经支付");
        System.out.println("调用物流服务，未发货
\n");
    }
}

public class SendOrderState implements State{

    @Override
    public void handle() {
        System.out.println("订单已经发货");
        System.out.println("调用短信服务，告诉用户
已经发货");
        System.out.println("更新物流信息\n");
    }
}
/**
 * 小滴课堂,愿景：让技术不再难学 https://xdclass.net
 *
 * @Description
 * @Author 二当家小D
}
```

```
* @Remark 有问题直接联系我，源码-笔记-技术交流群 微信  
xdclass6  
* @Version 1.0  
**/  
  
public class OrderContext {  
  
    private State state;  
  
    public OrderContext(){}  
  
    public void setState(State state) {  
        this.state = state;  
        System.out.println("订单状态变更！！！");  
        this.state.handle();  
    }  
}  
  
//使用  
public static void main(String[] args) {  
  
    OrderContext orderContext = new  
OrderContext();  
  
    orderContext.setState(new  
NewOrderState());  
  
    orderContext.setState(new  
PayOrderState());
```

```
    orderContext.setState(new  
SendOrderState());  
  
}
```

- 优点
 - 只需要改变对象状态即可改变对象的行为
 - 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数
- 缺点
 - 状态模式的使用会增加系统类和对象的个数。
 - 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱
 - 状态模式对“开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码
- 状态设计和策略模式的区别
 - UML图一样，结构基本类似

- 状态模式重点在各状态之间的切换，从而做不同的事情
 - 策略模式更侧重于根据具体情况选择策略，并不涉及切换
-
- 状态模式不同状态下做的事情不同，而策略模式做的都是同一件事。例如，聚合支付平台，有支付宝、微信支付、银联支付，虽然策略不同，但最终做的事情都是支付
 - 状态模式，各个状态的同一方法做的是不同的事，不能互相替换

第5集 不常用的设计模式和学习建议

简介：介绍不常用的设计模式和学习建议

- 设计模式没有说一定是多少个，技术演进，模式也在不断更新
 - head first 设计模式、gof
- 像有些很少用的设计模式
 - 访问者模式、解释器模式、中介模式等，基本都用不上，简单知道就行的了
- 网上还有很多其他模式
 - MVC模式， Model-View-Controller（模型-视图-控制器）、传输模式等
 - 也有很多都是我们前面讲的模式组合或者变种
- 多看源码、网上的博文，多看几遍设计模式，这样反复才可以真正的体会到设计模式的好处
 - 一定不能固定死板
- 公司里面如果有重构需求（业务发展快的公司很多这样的需求，多发现原先存在的问题和结合设计模式的原则）



**这题我见过就是
想不起来**



技术与生活更加有趣" 更多架构课程请访问 xdclass.net

第十一章 大话 设计模式在框架和源码里面的应用

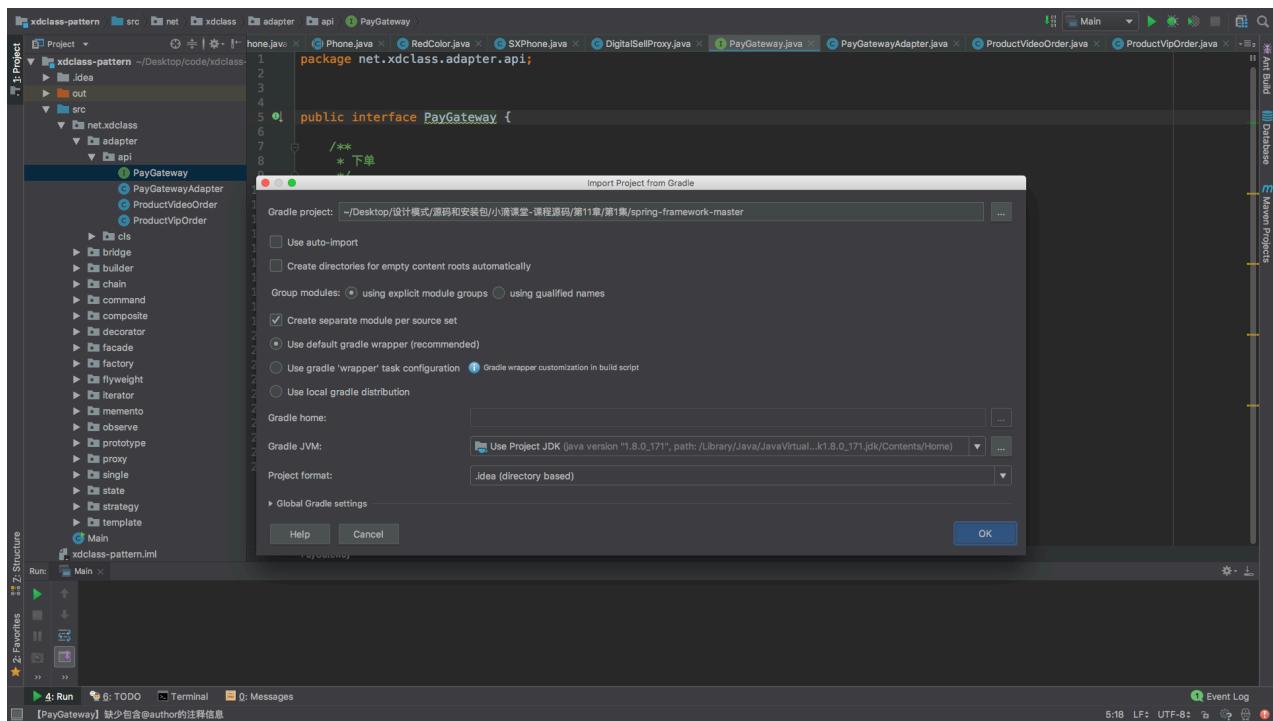
第1集 设计模式在框架和源码里面的应用和准备

简介：介绍设计模式的在jdk源码的应用和Spring源码准备

- 前面在jdk源码里面有讲多个设计模式，大家回想下
 - 迭代器设计模式:Iterator类
 - 装饰器模式： BufferedInputStream类
 - 单例设计模式JDK中Runtime类
 - 建造者模式 StringBuilder类
 - 适配器模式 JDBC数据库驱动
 - 享元模式 JAVA 中的 String
 - 策略设计模式 Comparator 接口常用的 compare()方法
- Spring准备源码地址
 - <https://github.com/spring-projects/spring-framework>

k

- 下载导入idea
- spring框架应用种设计模式：简单工厂模式、工厂方法模式、单例模式、代理模式、观察者模式 等等



第2集 单例设计模式在Spring框架里面的应用

简介：单例模式在Spring框架里面的应用

- scope属性值 singleton: 单例, 默认值, 调用getBean方法返回是同一个对象, 实例会被缓存起来, 效率比较高 当一个bean被标识为singleton时候, spring的IOC容器中只会存在一个该bean

```
<!--<bean id="video"
class="net.xdclass.sp.domain.Video"
scope="singleton"> -->
<bean id="video"
class="net.xdclass.sp.domain.Video"
scope="prototype">

    <property name="id" value="9" />
    <property name="title" value="Spring
5.x课程" />

</bean>
```

- scope属性值 prototype: 多例, 调用getBean方法创建不同的对象, 会频繁的创建和销毁对象造成很大的开销

```
private static void
testScope(ApplicationContext context){
    Video video1 =
(Video)context.getBean("video");

    Video video2 =
(Video)context.getBean("video");

    //靠匹配内存地址, == 是匹配内存地址
    System.out.println( video1 == video2 );

}
```

第3集 模板方法模式在Spring框架里面的应用之 JDBCTemplate

简介：讲解模板方法模式JDBCTempalte的应用

- 模板方法模式在Spring源码JDBCTemplate的应用
- 入口

```
public List<Map<String, Object>>  
queryForList(String sql)
```

```
public List<Map<String, Object>>  
queryForList(String sql)  
  
    // Methods dealing with static SQL (java.sql.Statement)  
    //  
    @Nullable  
    private <T> T execute(StatementCallback<T> action, boolean closeResources) throws DataAccessException {  
        Assert.notNull(action, "Callback object must not be null");  
        Connection con = DataSourceUtils.getConnection(obtainDataSource());  
        Statement stmt = null;  
        try {  
            stmt = con.createStatement();  
            applyStatementSettings(stmt);  
            T result = action.doInStatement(stmt);  
            handleWarnings(stmt);  
            return result;  
        } catch (SQLException ex) {  
            // Release Connection early, to avoid potential connection pool deadlock  
            // in the case when the exception translator hasn't been initialized yet.  
            String sql = getSql(action);  
            JdbcUtils.closeStatement(stmt);  
            stmt = null;  
            DataSourceUtils.releaseConnection(con, getDataSource());  
            con = null;  
            throw translateException("StatementCallback", sql, ex);  
        } finally {  
            if (closeResources) {  
                JdbcUtils.closeStatement(stmt);  
                DataSourceUtils.releaseConnection(con, getDataSource());  
            }  
        }  
    }  
    @Override  
    @Nullable  
    public <T> T execute(StatementCallback<T> action) throws DataAccessException {  
        return execute(action, closeResources: true);  
    }  
}
```

第4集 代理和策略模式在Spring框架里面的应用

简介：讲解代理和策略模式在Spring里面的应用

- 程序代码里面的代理

什么是静态代理：由程序创建或特定工具自动生成源代码，在程序运行前，代理类的.class文件就已经存在

什么是动态代理：在程序运行时，运用反射机制动态创建而成，无需手动编写代码

JDK动态代理

CGLIB动态代理

- 两种动态代理的区别：

- JDK动态代理：要求目标对象实现一个接口，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，这个时候就可以用CGLib动态代理
- CGLib动态代理，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的扩展
- JDK动态代理是自带的，CGLib需要引入第三方包
- CGLib动态代理基于继承来实现代理，所以无法对final类、private方法和static方法实现代理

- Spring AOP中的代理使用的默认策略：

- 如果目标对象实现了接口，则默认采用JDK动态代理

- 如果目标对象没有实现接口，则采用CgLib进行动态代理
 - 如果目标对象实现了接口，程序里面依旧可以指定使用 Cglib动态代理
- 核心类

AopProxy

JdkDynamicAopProxy

CglibAopProxy

```
ProxyFactoryBean # protected Object  
getProxy(AopProxy aopProxy)
```



第十二章 设计模式课程总结+学习路线

第1集 设计模式课程总结和学习路线

简介：课程知识点回顾和课程总结

- 回顾课程知识点和疑惑
 - 几十种设计模式记不住
 - 感觉学了很多-但又感觉啥都没学
 - 设计模式不是固定死板
- 学习建议
 - 遇到问题怎么解决?
 - 源码会一直存在github，地址: <https://github.com/jacksonxy/design-pattern>
 - 不做“有人生，没人养”的教程和项目，如果是我这边的课程的项目，有问题直接问我就可以，也会一直维护下去
- 学习路线
- 推荐社区网站
 - stackoverflow
 - dzone



今天谁也不能阻止我学习

小滴课堂，愿景：让编程不在难学，让技术与生活更加有趣

相信我们，这个是可以让你学习更加轻松的平台，里面的课程
绝对会让你技术不断提升

欢迎加小D讲师的微信：**xdclass-lw**

我们官方网站：<https://xdclass.net>

千人IT技术交流QQ群：**718617859**

重点来啦：加讲师微信 免费赠送你干货文档大集合，包含前
端，后端，测试，大数据，运维主流技术文档（持续更新）

<https://mp.weixin.qq.com/s/qYnjcDYGFDQorWmSfE7lpQ>



添加讲师获取课程技术答疑！

Wechat: xdclass-anna

search

新
客
户

下单咨询
添加微信



: xdclass-anna



零基础到Java高级开发工程师 架构全套学习路线

适合人群

- ✓ 零基础自学, 到高级Java后端工程师就业路线;
- ✓ 传统软件工程师 转型 互联网项目技术栈;
- ✓ 学习时间约1~2个月, 涵盖几十套核心课程, 基础+多个项目实战
- ✓ 一线城市平均薪酬15~25k

阶段一

○ Java工程师零基础就业班学习路线

1

新版Java从零基础到高级进阶

2 全新HTML+CSS零基础入门到进阶网页制作教程

3 全新Javascript零基础多实战例子教程

4 Linux/Centos7零基础入门到高级实战视频教程

5 全新Mysql数据库零基础入门到实战精讲

6 新版Javaweb+jdbc+maven零基础到开发
论坛项目实战

7 【面试必备】多线程并发编程+JUC+JDK源码教程

8 新版SSM框架-SpringBoot2.3+Spring5+Mybatis3.x
零基础到开发小滴课堂移动端系统综合实战

9 正版Redis4.x零基础整合springboot视频教程

阶段二

○中级后端工程师路线

1 IDEA零基础入门到进阶(整合阿里巴巴编码规范)

2 新版Maven3.5+Nexus私服搭建全套核心技术

3 Redis+Memcached+MongoDB+Redisson+Redis集群+Redis持久化

3 SpringBoot 2.x微信支付在线教育网站项目实战

4 前端-后端必备-Nginx零基础到分布式高并发专题

5 Jenkins持续集成 Git Gitlab Sonar视频教程

6 新一代微服务AlibabaCloud+SpringCloud全家桶实战

7 JMeter接口压力测试打造高性能服务

8 Redis高并发高可用集群百万级秒杀实战

9 权限框架Shiro+SpringBoot2.x零基础到高级实战

10 全新JDK8~JDK13全套新特性教程

阶段三

○ 高级后端工程师/技术经理路线

1 Docker实战视频教程入门到高级

2 互联网架构之JAVA虚拟机JVM零基础到高级实战

3 Linux/shell脚本编程视频教程

4 玩转搜索框架ElasticSearch7.x实战

5 Zookeeper+Dubbo微服务教程分布式视频教程

6 RocketMQ4.X教程消息队列

**7 SpringBoot2微服务Dubbo优惠券项目实战
Java架构全套视频教程**

**8 互联网架构之Netty4.X入门到进阶打造百万连接
服务器**

9 小滴课堂全栈/后端高级工程师面试专题第一季

扫码咨询客服小姐姐 >>

开启属于你的Java高级开发之路

