# 反射机制

## 什么是Java反射

就是正在运行，动态获取这个类的所有信息。

## 反射机制的作用

  1，反编译：.class-->.java

   2．通过反射机制访问java对象的属性，方法，构造方法等；

## 反射机制的应用场景

Jdbc 加载驱动-----

Spring IOC

框架

## 反射机制获取类有三种方法

|  |
| --- |
| **//第一种方式：**  **Classc1 = Class.*forName*("Employee");**  **//第二种方式：**  **//java中每个类型都有class 属性.**  **Classc2 = Employee.class;**    **//第三种方式：**  **//java语言中任何一个java对象都有getClass 方法**  **Employeee = new Employee();**  **Classc3 = e.getClass(); //c3是运行时类 (e的运行时类是Employee)** |

## 反射创建对象的方式

|  |
| --- |
| **Class<?> forName = Class.*forName*("com.itmayiedu.entity.User");**  **// 创建此Class 对象所表示的类的一个新实例 调用了User的无参数构造方法.**  **Object newInstance = forName.newInstance();** |

实例化有参构造函数

|  |
| --- |
| **Class<?> forName = Class.*forName*("com.itmayiedu.entity.User");**  **Constructor<?> constructor = forName.getConstructor(String.class, String.class);**  **User newInstance = (User) constructor.newInstance("123", "123");** |

## 反射创建api

|  |  |
| --- | --- |
| 方法名称 | 作用 |
| getDeclaredMethods [] | 获取该类的所有方法 |
| getReturnType() | 获取该类的返回值 |
| getParameterTypes() | 获取传入参数 |
| getDeclaredFields() | 获取该类的所有字段 |
| setAccessible | 允许访问私有成员 |

## 使用反射为类私有属性赋值

|  |
| --- |
| **// 为user对象私有属性赋值**  **Class<?> classUser = Class.*forName*("com.itmayiedu.entity.User");**  **// 获取到当前的所有属性**  **Field[] fields = classUser.getDeclaredFields();**  **for (Field field : fields) {**  **System.*out*.println(field.getName());**  **}**  **// 获取当前所有的方法**  **Method[] declaredMethods = classUser.getDeclaredMethods();**  **for (Method method : declaredMethods) {**  **System.*out*.println(method.getName());**  **}**  **// 初始化对象**  **User user = (User) classUser.newInstance();**  **Field declaredField = classUser.getDeclaredField("id");**  **// 标记为true 允许反射赋值**  **declaredField.setAccessible(true);**  **declaredField.set(user, "20");**  **System.*out*.println("使用反射机制给id赋值为:"+user.getId());** |

## JDBC反射加载驱动

|  |
| --- |
| **public class DemoJDBC {**  **public static void main(String[] args) throws Exception {**  **// 加载驱动类**  **Class.*forName*("com.mysql.jdbc.Driver");**    **// 通过DriverManager获取数据库连接**  **String url = "jdbc:mysql://192.168.1.150/test";**  **String user = "teamtalk";**  **String password = "123456";**  **Connection connection = (Connection) DriverManager.*getConnection*(**  **url, user, password);**    **PreparedStatement statement = (PreparedStatement) connection.prepareStatement(**  **"insert persion (name, age) value (?, ?)");**  **statement.setString(1, "hdu");**  **statement.setInt(2, 21);**  **statement.executeUpdate();**    **ResultSet resultSet = statement.executeQuery("select \* from persion");**  **// 操作ResultSet结果集**  **while (resultSet.next()) {**  **// 第一种获取字段方式**  **System.*out*.println(resultSet.getString(1) + " " +**  **resultSet.getString(2) + " " + resultSet.getString(3));**  **}**  **// 关闭数据库连接**  **resultSet.close();**  **statement.close();**  **connection.close();**  **}**  **}** |

## 禁止使用反射机制初始化

将构造函数为私有化

# 设计模式

## 设计模式分类

创建型模式，共五种：**工厂方法模式、抽象工厂模式**、**单例模式**、建造者模式、**原型模式。**

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

## 设计模式的六大原则

### 开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，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。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

### 里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

### 依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

### 迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的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。

### 合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

# 单例模式

## 什么是单例

保证一个类只有一个实例，并且提供一个访问该全局访问点

## 单例应用场景

1. Windows的Task Manager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~

2. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

3. 网站的计数器，一般也是采用单例模式实现，否则难以同步。

4. 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。

5. Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。

6. 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。

7. 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。

8. 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。

9. HttpApplication 也是单位例的典型应用。熟悉ASP.Net(IIS)的整个请求生命周期的人应该知道HttpApplication也是单例模式，所有的HttpModule都共享一个HttpApplication实例.

## 单例优缺点

优点：

1.在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就 防止其它对象对自己的实例化，确保所有的对象都访问一个实例

2.单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。

3.提供了对唯一实例的受控访问。

4.由于在系统内存中只存在一个对象，因此可以 节约系统资源，当 需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。

5.允许可变数目的实例。

6.避免对共享资源的多重占用。

缺点：

1.不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。

2.由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。

3.单例类的职责过重，在一定程度上违背了“单一职责原则”。

4.滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

## 单例创建方式

1. **饿汉式:类初始化时,会立即加载该对象，线程天生安全,调用效率高。**
2. **懒汉式: 类初始化时,不会初始化该对象,真正需要使用的时候才会创建该对象,具备懒加载功能。**
3. 静态内部方式:结合了懒汉式和饿汉式各自的优点，真正需要对象的时候才会加载，加载类是线程安全的。
4. 枚举单例: 使用枚举实现单例模式 优点:实现简单、调用效率高，枚举本身就是单例，由jvm从根本上提供保障!避免通过反射和反序列化的漏洞， 缺点没有延迟加载。
5. 双重检测锁方式 (因为JVM本质重排序的原因，可能会初始化多次，不推荐使用)

### 饿汉式

|  |
| --- |
| //饿汉式  **public** **class** SingletonDemo01 {  // 类初始化时,会立即加载该对象，线程天生安全,调用效率高  **private** **static** SingletonDemo01 *singletonDemo01* = **new** SingletonDemo01();  **private** SingletonDemo01() {  System.***out***.println("SingletonDemo01初始化");  }  **public** **static** SingletonDemo01 getInstance() {  System.***out***.println("getInstance");  **return** *singletonDemo01*;  }  **public** **static** **void** main(String[] args) {  SingletonDemo01 s1 = SingletonDemo01.*getInstance*();  SingletonDemo01 s2 = SingletonDemo01.*getInstance*();  System.***out***.println(s1 == s2);  }  } |

### 懒汉式

|  |
| --- |
| //懒汉式  **public** **class** SingletonDemo02 {  //类初始化时，不会初始化该对象，真正需要使用的时候才会创建该对象。  **private** **static** SingletonDemo02 *singletonDemo02*;  **private** SingletonDemo02() {    }  **public** **synchronized** **static** SingletonDemo02 getInstance() {  **if** (*singletonDemo02* == **null**) {  *singletonDemo02* = **new** SingletonDemo02();  }  **return** *singletonDemo02*;  }  **public** **static** **void** main(String[] args) {  SingletonDemo02 s1 = SingletonDemo02.*getInstance*();  SingletonDemo02 s2 = SingletonDemo02.*getInstance*();  System.***out***.println(s1 == s2);  }  } |

### 静态内部类

|  |
| --- |
| // 静态内部类方式  **public** **class** SingletonDemo03 {  **private** SingletonDemo03() {  System.***out***.println("初始化..");  }  **public** **static** **class** SingletonClassInstance {  **private** **static** **final** SingletonDemo03 ***singletonDemo03*** = **new** SingletonDemo03();  }  // 方法没有同步  **public** **static** SingletonDemo03 getInstance() {  System.***out***.println("getInstance");  **return** SingletonClassInstance.***singletonDemo03***;  }  **public** **static** **void** main(String[] args) {  SingletonDemo03 s1 = SingletonDemo03.*getInstance*();  SingletonDemo03 s2 = SingletonDemo03.*getInstance*();  System.***out***.println(s1 == s2);  }  } |

　优势：兼顾了懒汉模式的内存优化（使用时才初始化）以及饿汉模式的安全性（不会被反射入侵）。

　劣势：需要两个类去做到这一点，虽然不会创建静态内部类的对象，但是其 Class 对象还是会被创建，而且是属于永久带的对象。

### 枚举方式

#### 什么是枚举

枚举本身是单例的，一般用于项目中定义常量。

|  |
| --- |
| **enum** UserEnum {  ***HTTP\_200***(200, "请求成功"),***HTTP\_500***(500,"请求失败");  **private** Integer code;  **private** String name;  UserEnum(Integer code, String name) {  **this**.code = code;  **this**.name = name;  }  **public** Integer getCode() {  **return** code;  }  **public** **void** setCode(Integer code) {  **this**.code = code;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  }  **public** **class** TestEnum {  **public** **static** **void** main(String[] args) {  System.***out***.println(UserEnum.***HTTP\_500***.getCode());  }  } |

|  |
| --- |
| /使用枚举实现单例模式 优点:实现简单、枚举本身就是单例，由jvm从根本上提供保障!避免通过反射和反序列化的漏洞 缺点没有延迟加载  **public** **class** User {  **public** **static** User getInstance() {  **return** SingletonDemo04.***INSTANCE***.getInstance();  }  **private** **static** **enum** SingletonDemo04 {  ***INSTANCE***;  // 枚举元素为单例  **private** User user;  **private** SingletonDemo04() {  System.***out***.println("SingletonDemo04");  user = **new** User();  }  **public** User getInstance() {  **return** user;  }  }  **public** **static** **void** main(String[] args) {  User u1 = User.*getInstance*();  User u2 = User.*getInstance*();  System.***out***.println(u1 == u2);  }  } |

### 双重检测锁

|  |
| --- |
| **public** **class** SingletonDemo04 {  **private** SingletonDemo04 singletonDemo04;  **private** SingletonDemo04() {  }  **public** SingletonDemo04 getInstance() {  **if** (singletonDemo04 == **null**) {  **synchronized** (**this**) {  **if** (singletonDemo04 == **null**) {  singletonDemo04 = **new** SingletonDemo04();  }  }  }  **return** singletonDemo04;  }  } |

### 单例防止反射漏洞攻击

在构造函数中，只能允许初始化化一次即可。

|  |
| --- |
| **private** **static** **boolean** *flag* = **false**;  **private** SingletonDemo04() {  **if** (*flag* == **false**) {  *flag* = !*flag*;  } **else** {  **throw** **new** RuntimeException("单例模式被侵犯！");  }  }  **public** **static** **void** main(String[] args) {  } |

### 如何选择单例创建方式

如果不需要延迟加载单例，可以使用枚举或者饿汉式，相对来说枚举性好于饿汉式。

如果需要延迟加载，可以使用静态内部类或者懒汉式，相对来说静态内部类好于懒韩式。

最好使用饿汉式

# UML使用

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

Grady Booch的描述对象集合和它们之间的关系的方法。James Rumbaugh的对象建模技术（OMT）。Ivar Jacobson的包括用例方法的方式。还有其他一些想法也对UML起到了作用，UML是Booch, Rumbaugh, Jacobson。UML已经被对象管理组织（OMG）接受为标准，这个组织还制定了通用对象请求代理体系结构（CORBA），是分布式对象编程行业的领头羊。计算机辅助软件工程（CASE）产品的供应商也支持UML，并且它基本上已经被所有的软件开发产品制造商所认可，这其中包括IBM和微软(用于它的VB环境)。

UML规范用来描述建模的概念有，类（对象的）、对象、关联、职责、行为、接口、用例、包、顺序、协作，以及状态。

参考资料: https://www.cnblogs.com/xiluhua/p/6667935.html

# 工厂模式

## 什么是工厂模式

实现了创建者和调用者分离，工厂模式分为简单工厂、工厂方法、抽象工厂模式

## 工厂模式好处

工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。

利用工厂模式可以降低程序的耦合性，为后期的维护修改提供了很大的便利。

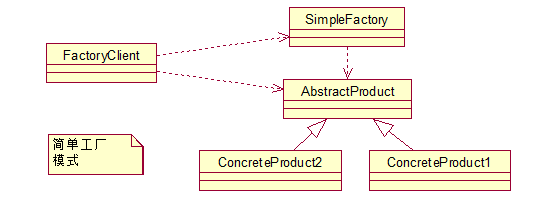
将选择实现类、创建对象统一管理和控制。从而将调用者跟我们的实现类解耦。

工厂与容器概念

## 工厂模式分类

### 简单工厂模式

简单工厂模式相当于是一个工厂中有各种产品，创建在一个类中，客户无需知道具体产品的名称，只需要知道产品类所对应的参数即可。**但是工厂的职责过重，而且当类型过多时不利于系统的扩展维护。**



|  |
| --- |
| **public** **interface** Car {  **public** **void** run();  }  **public** **class** AoDi **implements** Car {  **public** **void** run() {  System.***out***.println("我是奥迪汽车..");  }  }  **public** **class** JiLi **implements** Car {  **public** **void** run() {  System.***out***.println("我是吉利汽车...");  }  }  **public** **class** CarFactory {  **public** **static** Car createCar(String name) {  **if** (StringUtils.*isEmpty*(name)) {  **return** **null**;  }  **if**(name.equals("奥迪")){  **return** **new** AoDi();  }  **if**(name.equals("吉利")){  **return** **new** JiLi();  }  **return** **null**;  }  }  **public** **class** Client01 {  **public** **static** **void** main(String[] args) {  Car aodi =CarFactory.*createCar*("奥迪");  Car jili =CarFactory.*createCar*("吉利");  aodi.run();  jili.run();  }  } |

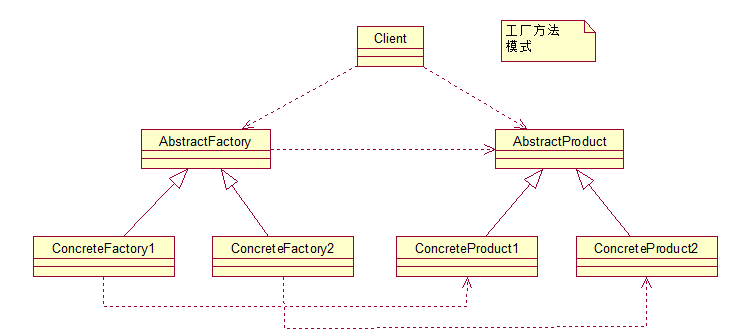
#### 单工厂的优点/缺点

优点：简单工厂模式能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。明确区分了各自的职责和权力，有利于整个软件体系结构的优化。

缺点：很明显工厂类集中了所有实例的创建逻辑，容易违反GRASPR的高内聚的责任分配原则

### 工厂方法模式

### 什么是工厂方法模式



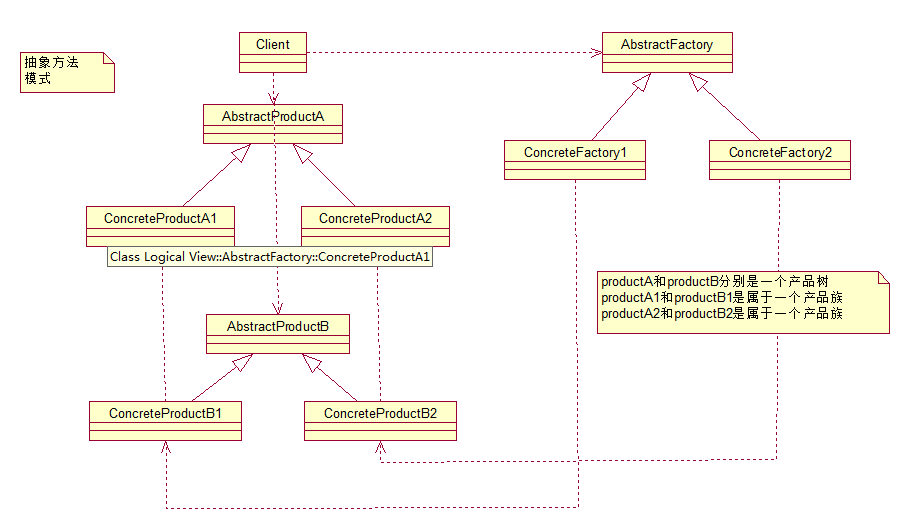
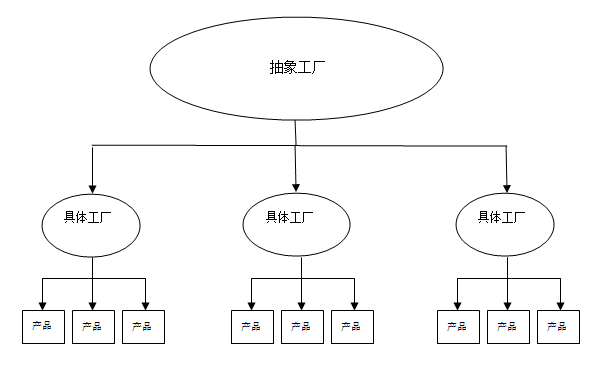
工厂方法模式Factory Method，又称多态性工厂模式。在工厂方法模式中，核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给子类去做。该核心类成为一个抽象工厂角色，仅负责给出具体工厂子类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

|  |
| --- |
| **public** **interface** Car {  **public** **void** run();  }  **public** **class** AoDi **implements** Car {  **public** **void** run() {  System.***out***.println("我是奥迪汽车..");  }  }  **public** **class** JiLi **implements** Car {  **public** **void** run() {  System.***out***.println("我是吉利汽车...");  }  }  **public** **class** JiLiFactory **implements** CarFactory {  **public** Car createCar() {  **return** **new** JiLi();  }  }  **public** **class** AoDiFactory **implements** CarFactory {  **public** Car createCar() {    **return** **new** AoDi();  }  }  **public** **class** Client {  **public** **static** **void** main(String[] args) {  Car aodi = **new** AoDiFactory().createCar();  Car jili = **new** JiLiFactory().createCar();  aodi.run();  jili.run();  }  } |

### 抽象工厂模式

#### 什么是抽象工厂模式

抽象工厂简单地说是工厂的工厂，抽象工厂可以创建具体工厂，由具体工厂来产生具体产品。



|  |
| --- |
| //发动机  **public** **interface** Engine {  **void** run();  **void** start();  }  **class** EngineA **implements** Engine {  **public** **void** run() {  System.***out***.println("转的快!");  }  **public** **void** start() {  System.***out***.println("启动快,自动档");  }  }  **class** EngineB **implements** Engine {  **public** **void** run() {  System.***out***.println("转的慢!");  }  **public** **void** start() {  System.***out***.println("启动快,手动档");  }  }  //座椅  **public** **interface** Chair {  **void** run();  }  **class** ChairA **implements** Chair{  **public** **void** run() {  System.***out***.println("可以自动加热!");  }    }  **class** ChairB **implements** Chair{  **public** **void** run() {  System.***out***.println("不能加热!");  }    }  **public** **interface** CarFactory {  // 创建发动机  Engine createEngine();  // 创建座椅  Chair createChair();  }  **public** **class** JiLiFactory **implements** CarFactory {  **public** Engine createEngine() {    **return** **new** EngineA();  }  **public** Chair createChair() {    **return** **new** ChairA();  }  }  **public** **class** Client002 {  **public** **static** **void** main(String[] args) {  CarFactory carFactory=**new** JiLiFactory();  Engine engine=carFactory.createEngine();  engine.run();  engine.start();  }    } |

### 简单工厂、工厂方法、抽象工厂之小结、区别

简单工厂 ： 用来生产同一等级结构中的任意产品。（不支持拓展增加产品）

　工厂方法 ：用来生产同一等级结构中的固定产品。（支持拓展增加产品）

抽象工厂 ：用来生产不同产品族的全部产品。（不支持拓展增加产品；支持增加产品族）

# 代理模式

## 什么是代理模式

通过代理控制对象的访问,可以详细访问某个对象的方法，在这个方法调用处理，或调用后处理。既(AOP微实现) ,AOP核心技术面向切面编程。



## 代理模式应用场景

SpringAOP、事物原理、日志打印、权限控制、远程调用、安全代理 可以隐蔽真实角色

## 代理的分类

静态代理(静态定义代理类)

动态代理(动态生成代理类)

Jdk自带动态代理

Cglib 、javaassist（字节码操作库）

## 静态代理

### 什么是静态代理

由程序员创建或工具生成代理类的源码，再编译代理类。所谓静态也就是在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了。

### 静态代理代码

|  |
| --- |
| **public** **interface** IUserDao {  **void** save();  }  **public** **class** UserDao **implements** IUserDao {  **public** **void** save() {  System.***out***.println("已经保存数据...");  }  }  代理类  **public** **class** UserDaoProxy **implements** IUserDao {  **private** IUserDao target;  **public** UserDaoProxy(IUserDao iuserDao) {  **this**.target = iuserDao;  }  **public** **void** save() {  System.***out***.println("开启事物...");  target.save();  System.***out***.println("关闭事物...");  }  } |

## 动态代理

### 什么是动态代理

1.代理对象,不需要实现接口

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(需要我们指定创建代理对象/目标对象实现的接口的类型)

3.动态代理也叫做:JDK代理,接口代理

### JDK动态代理

1)原理：是根据类加载器和接口创建代理类（此代理类是接口的实现类，所以必须使用接口 面向接口生成代理，位于java.lang.reflect包下）

2)实现方式：

1. 通过实现InvocationHandler接口创建自己的调用处理器 IvocationHandler handler = new InvocationHandlerImpl(…);

2. 通过为Proxy类指定ClassLoader对象和一组interface创建动态代理类Class clazz = Proxy.getProxyClass(classLoader,new Class[]{…});

3. 通过反射机制获取动态代理类的构造函数，其参数类型是调用处理器接口类型Constructor constructor = clazz.getConstructor(new Class[]{InvocationHandler.class});

4. 通过构造函数创建代理类实例，此时需将调用处理器对象作为参数被传入Interface Proxy = (Interface)constructor.newInstance(new Object[] (handler));

**缺点：jdk动态代理，必须是面向接口，目标业务类必须实现接口**

|  |
| --- |
| // 每次生成动态代理类对象时,实现了InvocationHandler接口的调用处理器对象  **public** **class** InvocationHandlerImpl **implements** InvocationHandler {  **private** Object target;// 这其实业务实现类对象，用来调用具体的业务方法  // 通过构造函数传入目标对象  **public** InvocationHandlerImpl(Object target) {  **this**.target = target;  }  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  Object result = **null**;  System.***out***.println("调用开始处理");  result = method.invoke(target, args);  System.***out***.println("调用结束处理");  **return** result;  }  **public** **static** **void** main(String[] args) **throws** NoSuchMethodException, SecurityException, InstantiationException,  IllegalAccessException, IllegalArgumentException, InvocationTargetException {  // 被代理对象  IUserDao userDao = **new** UserDao();  InvocationHandlerImpl invocationHandlerImpl = **new** InvocationHandlerImpl(userDao);  ClassLoader loader = userDao.getClass().getClassLoader();  Class<?>[] interfaces = userDao.getClass().getInterfaces();  // 主要装载器、一组接口及调用处理动态代理实例  IUserDao newProxyInstance = (IUserDao) Proxy.*newProxyInstance*(loader, interfaces, invocationHandlerImpl);  newProxyInstance.save();  }  } |

### CGLIB动态代理

原理：利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

#### 什么是CGLIB动态代理

使用cglib[Code Generation Library]实现动态代理，并不要求委托类必须实现接口，底层采用asm字节码生成框架生成代理类的字节码

#### CGLIB动态代理相关代码

|  |
| --- |
| **public** **class** CglibProxy **implements** MethodInterceptor {  **private** Object targetObject;  // 这里的目标类型为Object，则可以接受任意一种参数作为被代理类，实现了动态代理  **public** Object getInstance(Object target) {  // 设置需要创建子类的类  **this**.targetObject = target;  Enhancer enhancer = **new** Enhancer();  enhancer.setSuperclass(target.getClass());  enhancer.setCallback(**this**);  **return** enhancer.create();  }  **public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {  System.***out***.println("开启事物");  Object result = proxy.invoke(targetObject, args);  System.***out***.println("关闭事物");  // 返回代理对象  **return** result;  }  **public** **static** **void** main(String[] args) {  CglibProxy cglibProxy = **new** CglibProxy();  UserDao userDao = (UserDao) cglibProxy.getInstance(**new** UserDao());  userDao.save();  }  } |

#### CGLIB动态代理与JDK动态区别

java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

Spring中。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP

2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

JDK动态代理只能对实现了接口的类生成代理，而不能针对类 。  
CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法 。  
因为是继承，所以该类或方法最好不要声明成final ，final可以阻止继承和多态。

# 建造者模式

## 什么是建造者模式

![C:\Users\Administrator\AppData\Roaming\Tencent\Users\644064779\QQ\WinTemp\RichOle\](D`39LS[T7(6B[585K0XVX.png](data:image/png;base64,)

建造者模式：是将一个复杂的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。

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的Test结合起来得到的。

建造者模式通常包括下面几个角色：

1、Builder：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。

2、ConcreteBuilder：实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例。

3、Director：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

4、Product：要创建的复杂对象。

## 建造者应用场景

1、去肯德基，汉堡、可乐、薯条、炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。

19元每周三 汉堡+可乐+薯条=套餐可能会发生改变。

2、JAVA 中的 StringBuilder 数组（单个字符）字整合在一起 字符串

使用场景：

1、需要生成的对象具有复杂的内部结构。

2、需要生成的对象内部属性本身相互依赖。

与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。

## 实际案例

这里以游戏开发中人物的构造过程为例。在游戏中创建一个形象时，需要对每个部位进行创建。简化而言，需要创建头部，身体和四肢。

头部、体部、四肢

### 建立一个人物对象Person

|  |
| --- |
| **public** **class** Person {  **private** String head;  **private** String body;  **private** String foot;  **public** String getHead() {  **return** head;  }  **public** **void** setHead(String head) {  **this**.head = head;  }  **public** String getBody() {  **return** body;  }  **public** **void** setBody(String body) {  **this**.body = body;  }  **public** String getFoot() {  **return** foot;  }  **public** **void** setFoot(String foot) {  **this**.foot = foot;  }  } |

### Builder（给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建）

|  |
| --- |
| **public** **interface** PersonBuilder {  **void** builderHead();  **void** builderBody();  **void** builderFoot();  Person BuilderPersion(); //组装  } |

### ConcreteBuilder（实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例）

|  |
| --- |
| **public** **class** ManBuilder **implements** PersonBuilder {  **private** Person person;  **public** ManBuilder() {  person = **new** Person();//创建一个Person实例,用于调用set方法  }  **public** **void** builderHead() {  person.setHead("建造者头部分");  }  **public** **void** builderBody() {  person.setBody("建造者身体部分");  }  **public** **void** builderFoot() {  person.setFoot("建造者头四肢部分");  }  **public** Person BuilderPersion() {  **return** person;  }  } |

### Director（调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建）

|  |
| --- |
| **public** **class** PersonDirector {  **public** Person constructPerson(PersonBuilder pb) {  pb.builderHead();  pb.builderBody();  pb.builderFoot();  **return** pb.BuilderPersion();  }  **public** **static** **void** main(String[] args) {  PersonDirector pb = **new** PersonDirector();  Person person = pb.constructPerson(**new** ManBuilder());  System.***out***.println(person.getHead());  System.***out***.println(person.getBody());  System.***out***.println(person.getFoot());  }  } |

# 模板方法

## 什么是模板方法

模板方法模式：定义一个操作中的算法骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的

重复代码全部在父类里面，不同业务的，使用抽象方法，抽取给子类进行实现。抽取过程---抽象方法。

某些特定步骤。

核心：处理某个流程的代码已经都具备，但是其中某个节点的代码暂时不能确定。因此，我们采用工厂方法模式，将这个节点的代码实现转移给

子类完成。即：处理步骤在父类中定义好，具体的实现延迟到子类中定义。

说白了，就是将一些相同操作的代码，封装成一个算法的骨架。核心的部分留在子类中操作，在父类中只把那些骨架做好。

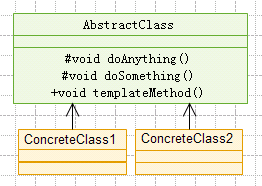
例如：

1.去银行办业务，银行给我们提供了一个模板就是：先取号，排对，办理业务（核心部分我们子类完成），给客服人员评分，完毕。

这里办理业务是属于子类来完成的，其他的取号，排队，评分则是一个模板。

2.去餐厅吃饭，餐厅给提供的一套模板就是：先点餐，等待，吃饭（核心部分我们子类完成），买单

这里吃饭是属于子类来完成的，其他的点餐，买单则是餐厅提供给我们客户的一个模板。



## 模板方法具体实现

这里使用银行办理业务为例

首先，定义一个模板。模板中把办理业务用作核心部分，让子类来实现。

|  |
| --- |
| //模板方法  **public** **abstract** **class** BankTemplateMethod {  // 1.取号排队  **public** **void** takeNumber() {  System.***out***.println("取号排队。。");  }  // 2.每个子类不同的业务实现，由各自子类实现.  **abstract** **void** transact();  // 3.评价  **public** **void** evaluate() {  System.***out***.println("反馈评价..");  }    **public** **void** process(){  takeNumber();  transact();  evaluate();  }  } |

## 具体的模板方法的子类

|  |
| --- |
| **public** **class** DrawMoney **extends** BankTemplateMethod {  @Override  **void** transact() {  System.***out***.println("我要取款");  }  } |

## 客户端测试

|  |
| --- |
| **public** **class** Client {  **public** **static** **void** main(String[] args) {  BankTemplateMethod bankTemplate=**new** DrawMoney();  bankTemplate.process();  }    } |

## 匿名内部类方式

|  |
| --- |
| BankTemplateMethod bankTemplateMethod=**new** BankTemplateMethod() {    @Override  **void** transact() {  System.***out***.println("我要存钱.");    }  };  bankTemplateMethod.process(); |

## 什么时候使用模板方法

实现一些操作时，整体步骤很固定，但是呢。就是其中一小部分容易变，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。

## 开发中应用场景

其实，各个框架中，都有模板方法模式的影子。

数据库访问的封装、Junit单元测试、**servlet中关于doGet/doPost**方法的调用

Hibernate中模板程序、spring中JDBCTemplate，HibernateTemplate等等

# 适配模式

## 什么是适配器

在设计模式中，适配器模式（英语：adapter pattern）有时候也称包装样式或者包装(wrapper)。将一个类的接口转接成用户所期待的。一个适配使得因接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

## 适配器分类

适配器分为，类适配器、对象适配、接口适配方式

类适配器方式采用继承方式，对象适配方式使用构造函数传递

## 适配器案例

我们就拿日本电饭煲的例子进行说明，日本电饭煲电源接口标准是110V电压，而中国标准电压接口是220V，所以要想在中国用日本电饭煲，需要一个电源转换器。

### 定义日本和中国两种接口及其实现

我们先定义日本220V电源接口和实现。

### 110V电源接口

|  |
| --- |
| //日本110V 电源接口  **public** **interface** JP110VInterface {  **public** **void** connect();  } |

### 110V电源接口实现

|  |
| --- |
| **public** **class** JP110VInterfaceImpl **implements** JP110VInterface {  @Override  **public** **void** connect() {  System.***out***.println("日本110V,接通电源,开始工作..");  }  } |

### 我们再定义中国220V电源接口和实现。

|  |
| --- |
| **public** **interface** CN220VInterface {  **public** **void** connect();  } |

|  |
| --- |
| **public** **class** CN220VInterfaceImpl **implements** CN220VInterface {  @Override  **public** **void** connect() {  System.***out***.println("中国220V,接通电源,开始工作");    }  } |

### 定义一个电压适配器

要想在中国使用日本电饭煲，需要把电饭煲110v的电源接口适配成我们220V的电源接口，这就需要一个电源适配器：

|  |
| --- |
| **public** **class** ElectricCooker {  **private** JP110VInterface jp110VInterface;//日本电饭煲  ElectricCooker(JP110VInterface jp110VInterface){  **this**.jp110VInterface=jp110VInterface;  }    **public** **void** cook(){  jp110VInterface.connect();  System.***out***.println("开始做饭了..");  }  } |

### 定义一个电压适配器

|  |
| --- |
| **public** **class** PowerAdaptor **implements** JP110VInterface {  **private** CN220VInterface cn220VInterface;  **public** PowerAdaptor(CN220VInterface cn220VInterface) {  **this**.cn220VInterface = cn220VInterface;  }  @Override  **public** **void** connect() {  cn220VInterface.connect();  }  } |

### 测试开始运行

|  |
| --- |
| **public** **class** AdaptorTest {  **public** **static** **void** main(String[] args) {  CN220VInterface cn220VInterface = **new** CN220VInterfaceImpl();  PowerAdaptor powerAdaptor = **new** PowerAdaptor(cn220VInterface);  // 电饭煲  ElectricCooker cooker = **new** ElectricCooker(powerAdaptor);  cooker.cook();//使用了适配器,在220V的环境可以工作了。  }  } |

### 适配器应用场景

我们根据上面的适配器的特点的介绍中，我们来分析下适配器模式的几类比较适用的使用场景：

1、我们在使用第三方的类库，或者说第三方的API的时候，我们通过适配器转换来满足现有系统的使用需求。

2、我们的旧系统与新系统进行集成的时候，我们发现旧系统的数据无法满足新系统的需求，那么这个时候，我们可能需要适配器，完成调用需求。

3、我们在使用不同数据库之间进行数据同步。(我这里只是分析的是通过程序来说实现的时候的情况。还有其他的很多种方式[数据库同步])。

OutputStreamWriter：是Writer的子类，将输出的字符流变为字节流，即：将一个字符流的输出对象变为字节流的输出对象。

InputStreamReader：是Reader的子类，将输入的字节流变为字符流，即：将一个字节流的输入对象变为字符流的输入对象。

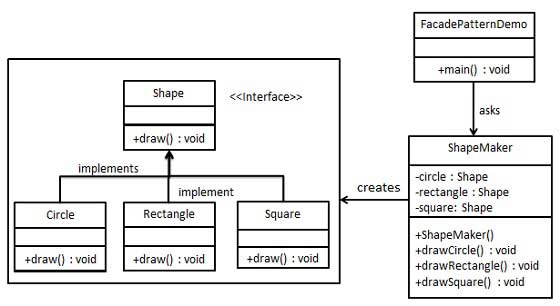
SpringMVC 适配器

# 外观模式

## 什么是外观模式

外观模式（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。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。

这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。



## 外观模式例子

用户注册完之后，需要调用阿里短信接口、邮件接口、微信推送接口。

|  |
| --- |
| **public** **interface** EamilSmsService {  **public** **void** sendSms();  }  **public** **class** EamilSmsServiceImpl **implements** EamilSmsService{  **public** **void** sendSms() {  System.***out***.println("发送邮件消息");    }  } |

|  |
| --- |
| //微信消息推送  **public** **interface** WeiXinSmsService {  **public** **void** sendSms();  }  **public** **class** EamilSmsServiceImpl **implements** EamilSmsService{  @Override  **public** **void** sendSms() {  System.***out***.println("发送邮件消息");    }  } |

|  |
| --- |
| //阿里短信消息  **public** **interface** AliSmsService {  **public** **void** sendSms();  }  **public** **class** AliSmsServiceImpl **implements** AliSmsService {  @Override  **public** **void** sendSms() {  System.***out***.println("支付宝发送消息...");  }  } |

门面类

|  |
| --- |
| **public** **class** Computer {  AliSmsService aliSmsService;  EamilSmsService eamilSmsService;  WeiXinSmsService weiXinSmsService;  **public** Computer() {  aliSmsService = **new** AliSmsServiceImpl();  eamilSmsService = **new** EamilSmsServiceImpl();  weiXinSmsService = **new** WeiXinSmsServiceImpl();  }  **public** **void** sendMsg() {  aliSmsService.sendSms();  eamilSmsService.sendSms();  weiXinSmsService.sendSms();  }  } |

|  |
| --- |
| **public** **class** Client {  **public** **static** **void** main(String[] args) {  // AliSmsService aliSmsService= new AliSmsServiceImpl();  // EamilSmsService eamilSmsService= new EamilSmsServiceImpl();  // WeiXinSmsService weiXinSmsService= new WeiXinSmsServiceImpl();  // aliSmsService.sendSms();  // eamilSmsService.sendSms();  // weiXinSmsService.sendSms();  **new** Computer().sendMsg();  }  } |

# 原型模式

## 什么是原型模式

克隆

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

## 原型模式应用场景

1. 类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等，通过原型拷贝避免这些消耗。   
   （2）通过new产生的一个对象需要非常繁琐的数据准备或者权限，这时可以使用原型模式。   
   （3）一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用，即保护性拷贝。

Spring框架中的多例就是使用原型。

## 原型模式UML类图（通用）

原型模式主要用于对象的复制，它的核心是就是类图中的原型类Prototype。Prototype类需要具备以下两个条件：   
　　（1）实现Cloneable接口。在java语言有一个Cloneable接口，它的作用只有一个，就是在运行时通知虚拟机可以安全地在实现了此接口的类上使用clone方法。在java虚拟机中，只有实现了这个接口的类才可以被拷贝，否则在运行时会抛出CloneNotSupportedException异常。   
　　（2）重写Object类中的clone方法。Java中，所有类的父类都是Object类，Object类中有一个clone方法，作用是返回对象的一个拷贝，但是其作用域protected类型的，一般的类无法调用，因此Prototype类需要将clone方法的作用域修改为public类型。

## 原型模式分类

## 演示实例

|  |
| --- |
| /\*  \* 书本类型，扮演的是ConcretePrototype角色，而Cloneable扮演Prototype角色  \*/  **public** **class** Book **implements** Cloneable {  **private** String title;// 标题  **private** ArrayList<String> image = **new** ArrayList<String>();// 图片名列表  **public** Book() {  **super**();  }  /\*\*  \* 重写拷贝方法  \*/  @Override  **protected** Book clone() {  **try** {  Book book = (Book) **super**.clone();//  book.image=(ArrayList<String>)**this**.image.clone();//深复制  **return** book;  } **catch** (CloneNotSupportedException e) {  e.printStackTrace();  }  **return** **null**;  }  **public** ArrayList<String> getImage() {  **return** image;  }  **public** **void** addImage(String img) {  **this**.image.add(img);  }  **public** String getTitle() {  **return** title;  }  **public** **void** setTitle(String title) {  **this**.title = title;  }  /\*\*  \* 打印内容  \*/  **public** **void** showBook() {  System.***out***.println("----------------------Start----------------------");  System.***out***.println("title：" + title);  **for** (String img : image) {  System.***out***.println("image name:" + img);  }  System.***out***.println("----------------------End----------------------");  }  }  客户端代码  **public** **class** Client02 {  **public** **static** **void** main(String[] args) {  Book book1 = **new** Book();  book1.setTitle("书1");  book1.addImage("图1");  book1.showBook();  //以原型方式拷貝一份  Book book2 = book1.clone();  book2.showBook();  book2.setTitle("书2");  book2.addImage("圖2");  book2.showBook();  //再次还原打印书本  book1.showBook();  }  } |

## 原型模式分为浅复制和深复制

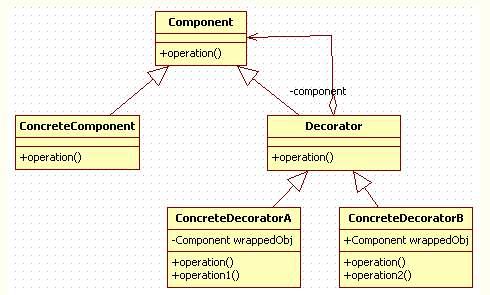
浅复制 —-只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“（浅复制）浅拷贝”，换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么浅复制出来的对象也会相应改变。

深复制 —-在计算机中开辟了一块新的内存地址用于存放复制的对象。

# 装饰模式

## 什么是装饰模式

　装饰器模式，也成为包装模式，顾名思义，就是对已经存在的某些类进行装饰，以此来扩展一些功能。其结构图如下：



Component为统一接口，也是装饰类和被装饰类的基本类型。

ConcreteComponent为具体实现类，也是被装饰类，他本身是个具有一些功能的完整的类。

Decorator是装饰类，实现了Component接口的同时还在内部维护了一个ConcreteComponent的实例，并可以通过构造函数初始化。而Decorator本身，通常采用默认实现，他的存在仅仅是一个声明：我要生产出一些用于装饰的子类了。而其子类才是赋有具体装饰效果的装饰产品类。

ConcreteDecorator是具体的装饰产品类，每一种装饰产品都具有特定的装饰效果。可以通过构造器声明装饰哪种类型的ConcreteComponent，从而对其进行装饰。

## 装饰模式案例

|  |
| --- |
| //房屋基础接口  **public** **interface** House {  **void** run();  }  //房屋装饰类  **public** **class** HouseDecorate **implements** House {  **private** House house;  **public** HouseDecorate(House house){  **this**.house=house;  }  @Override  **public** **void** run() {  house.run();  }  }  **public** **class** HouseDecorateImpl **extends** HouseDecorate {  **public** HouseDecorateImpl(House house) {  **super**(house);  }  @Override  **public** **void** run() {  **super**.run();  System.***out***.println("贴上墙纸..");  }  }  客户端调用  **public** **class** ClientTest {  **public** **static** **void** main(String[] args) {  HouseImpl houseImpl = **new** HouseImpl();  houseImpl.run();  System.***out***.println("###新增贴上墙纸..###");  HouseDecorate houseDecorate = **new** HouseDecorateImpl(houseImpl);  houseDecorate.run();  }  } |

## 装饰与代理区别

装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。换句话 说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。因此，当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模 式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

## 装饰模式应用场景

在IO中，具体构件角色是节点流，装饰角色是过滤流。

FilterInputStream和FilterOutputStream是装饰角色，而其他派生自它们的类则是具体装饰角色。

DataoutputStream out=new DataoutputStream(new FileoutputStream());

这就是 装饰者模式，DataoutputStream是装饰者子类，FileoutputStream是实现接口的子类。

这里不会调用到装饰者类--FilteroutputStream,只是作为继承的另一种方案，对客户端来说是透明的，是为了功能的扩张.

# 策略模式

## 什么是策略模式

定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

## 策略模式由三种角色组成

## 策略模式应用场景

策略模式的用意是针对一组算法或逻辑，将每一个算法或逻辑封装到具有共同接口的独立的类中，从而使得它们之间可以相互替换。策略模式使得算法或逻辑可以在不影响到客户端的情况下发生变化。说到策略模式就不得不提及OCP(Open Closed Principle) 开闭原则，即对扩展开放，对修改关闭。策略模式的出现很好地诠释了开闭原则，有效地减少了分支语句。

## 策略模式代码

此代码通过模拟不同会员购物车打折力度不同分为三种策略，初级会员，中级会员，高级会员。

|  |
| --- |
| //策略模式 定义抽象方法 所有支持公共接口  **abstract** **class** Strategy {  // 算法方法  **abstract** **void** algorithmInterface();  }  **class** StrategyA **extends** Strategy {  @Override  **void** algorithmInterface() {  System.***out***.println("算法A");  }  }  **class** StrategyB **extends** Strategy {  @Override  **void** algorithmInterface() {  System.***out***.println("算法B");  }  }  **class** StrategyC **extends** Strategy {  @Override  **void** algorithmInterface() {  System.***out***.println("算法C");  }  }  // 使用上下文维护算法策略  **class** Context {  Strategy strategy;  **public** Context(Strategy strategy) {  **this**.strategy = strategy;  }  **public** **void** algorithmInterface() {  strategy.algorithmInterface();  }  }  **class** ClientTestStrategy {  **public** **static** **void** main(String[] args) {  Context context;  context = **new** Context(**new** StrategyA());  context.algorithmInterface();  context = **new** Context(**new** StrategyB());  context.algorithmInterface();  context = **new** Context(**new** StrategyC());  context.algorithmInterface();  }  } |

## 策略模式应用场景

# 观察者模式

## 什么是观察者模式

观察者模式（Observer），是一种行为性模型，行为型模式关注的是系统中对象之间的相互交互，解决系统在运行时对象之间的相互通信和协作，进一步明确对象的职责。相比来说，创建型模式关注对象的创建过程，结构型模式关注对象和类的组合关系。

模式的职责

## 模式的职责

观察者模式主要用于1对N的通知。当一个对象的状态变化时，他需要及时告知一系列对象，令他们做出相应。

实现有两种方式：

推：每次都会把通知以广播的方式发送给所有观察者，所有的观察者只能被动接收。

拉：观察者只要知道有情况即可，至于什么时候获取内容，获取什么内容，都可以自主决定。

## 模式的实现

|  |
| --- |
| //观察者的接口，用来存放观察者共有方法  **public** **interface** Observer {  // 观察者方法  **void** update(Subjecct subjecct);  }  //观察对象的父类  **public** **class** Subjecct {  //观察者的存储集合  **private** List<Observer> list = **new** ArrayList<>();  // 注册观察者方法  **public** **void** registerObserver(Observer obs) {  list.add(obs);  }  // 删除观察者方法  **public** **void** removeObserver(Observer obs) {  list.remove(obs);  **this**.notifyAllObserver();  }  // 通知所有的观察者更新  **public** **void** notifyAllObserver() {  **for** (Observer observer : list) {  observer.update(**this**);  }  }  }  //具体观察者对象的实现  **public** **class** RealObserver **extends** Subjecct {  //被观察对象的属性  **private** **int** state;  **public** **int** getState(){  **return** state;  }  **public** **void** setState(**int** state){  **this**.state=state;  //主题对象(目标对象)值发生改变  **this**.notifyAllObserver();  }    }  **public** **class** Client {  **public** **static** **void** main(String[] args) {  // 目标对象  RealObserver subject = **new** RealObserver();  // 创建多个观察者  ObserverA obs1 = **new** ObserverA();  ObserverA obs2 = **new** ObserverA();  ObserverA obs3 = **new** ObserverA();  // 注册到观察队列中  subject.registerObserver(obs1);  subject.registerObserver(obs2);  subject.registerObserver(obs3);  // 改变State状态  subject.setState(300);  System.***out***.println(obs1.getMyState());  System.***out***.println(obs2.getMyState());  System.***out***.println(obs3.getMyState());  // 改变State状态  subject.setState(400);  System.***out***.println(obs1.getMyState());  System.***out***.println(obs2.getMyState());  System.***out***.println(obs3.getMyState());  }  } |

## 观察者模式应用场景

关联行为场景，需要注意的是，关联行为是可拆分的，而不是“组合”关系。

事件多级触发场景。

跨系统的消息交换场景，如消息队列、事件总线的处理机制。