# 装饰模式

## 问题引入

在现实生活中，常常需要为现有的产品添加新的功能或美化其外观。如装修房子、手机套等。在软件开发过程中，有时候想要使用一些现存的组件，这些组件可能只是完成了一些核心的功能，使用装饰模式可以在不改变组件现有结构的情况下动态的扩展其功能。

## 定义

装饰模式（Decorator Pattern）是指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即添加额外的功能）的模式。

## 优缺点

装饰模式的优点有：

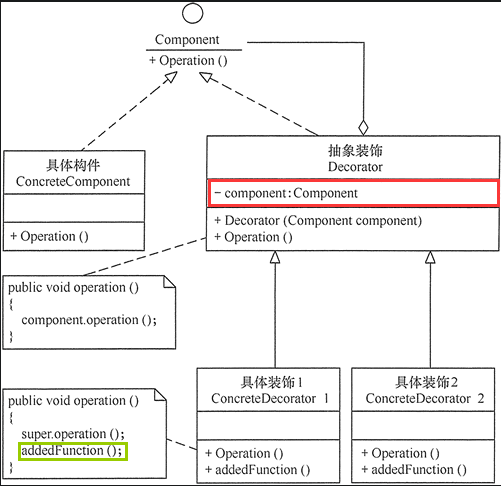
1. 采用装饰模式扩展对象的功能比采用继承方式更加灵活。
2. 可以设计出多个不同得具体装饰类，创造出多个不同行为的组合。
3. 装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

装饰模式的缺点有：

1. 装饰模式增加了许多子类，如果过度使用将造成程序的复杂程度变高。
2. 当使用了多层装饰，系统会变得非常复杂。

## 主要解决

一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。



## 结构

装饰模式主要包含以下角色：

1）抽象构件角色：定义一个抽象接口以规范准备接收附加职责的对象。

2）具体构建角色：实现抽象构件。可以通过具体装饰角色添加额外的职责。

3）抽象装饰角色：继承抽象构件，包含抽象构件实例。

4）具体装饰角色：实现抽象装饰，并未具体构件对象添加额外的职责。

不禁回想起小时候看过的日本动漫《四驱兄弟》：

土屋博士一直梦想着创造出能够更快加速的迷你四驱车，在经过各种尝试后，最终设计出了Saber。Saber的基本原理是利用空气摩擦力，但是Saber具有很大的改装空间，根据使用者的不同改装，甚至可以出现性能截然不同的四驱车。在Saber完成后，为了检验Saber的适用性和改装能力，土屋博士把Saber初始型号分发给一些喜爱迷你四驱车的孩子。土屋挑选的这些孩子性格不尽相同，以便观察不同的改装会给Saber带来怎样的后果。喜欢冲刺的星马豪把Saber改装成高速型赛车MagnumSaber（冲锋战神），，而擅长弯道的星马烈则把Saber改装成技术型赛车SonicSaber（音速战神）。同样使用Saber的还有黑泽的BlackSaber（黑色战神），小广成的Saber600等等。



## 应用场景

装饰模式通常在以下几种情况下被使用：

1. 当需要给一个现有类添加附加职责，又不能采用继承的方式进行扩充时。该类被隐藏或者该类是终极类或者采用继承方式会产生大量的子类。
2. 当需要通过对现有的一组基本功能进行排序组合而产生非常多的功能时。
3. 当对象的功能要求可以动态添加、动态撤销时。

装饰模式在 Java 语言中的最著名的应用莫过于 Java I/O 标准库的设计了。例如，InputStream 的子类 FilterInputStream，OutputStream 的子类 FilterOutputStream，Reader 的子类 BufferedReader 以及 FilterReader，还有 Writer 的子类 BufferedWriter、FilterWriter 以及 PrintWriter 等，它们都是抽象装饰类。

## 装饰模式与代理模式（静态代理）

在日常开发里面，我们经常需要给某个类的方法增加加某些特定的功能。例如：有婴儿，婴儿会吃饭和走动，如以下类：

public interface Human *{* void eat*()*;  
  
 void run*()*;  
*}*

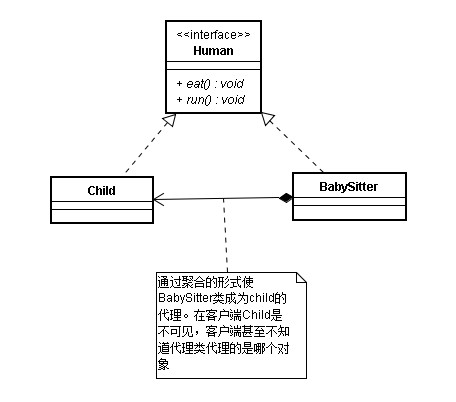
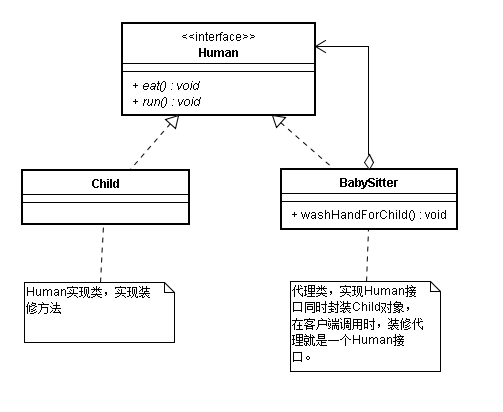
public class Baby implements Human*{* @Override  
 public void eat*() {* System.*out*.println*(*"Baby is eating..."*)*;  
 *}* @Override  
 public void run*() {* System.*out*.println*(*"Baby is running"*)*;  
 *}  
}*

突然有一天，家长发现不行，孩子不能随便吃东西，而且吃饭前一定要洗手。但是孩子太小(被委托方)，不会自己洗手。家长(Client 端)又没法照顾孩子。那简单，找个保姆照顾孩子！ 让保姆类和婴儿类共同实现同一个接口，让保姆类全程管理小孩，同时在家长眼里，只要看到保姆在帮孩子洗手就可以了。于是，有以下内容。

|  |  |
| --- | --- |
| */\*\*  \** ***@author*** *xzy  \** ***@date*** *2020-01-19 15:46  \* 说明：保姆（装饰模式）  \*/* public class Babysitter implements Human *{* private Human human;   public Babysitter*(*Human human*) {* this.human = human;  *}*  @Override  public void eat*() {* this.washHandForChild*()*;  human.eat*()*;  *}* @Override  public void run*() {* human.run*()*;  *}* public void washHandForChild*() {* System.*out*.println*(*"help the child to wash his hands"*)*;  *} }* | */\*\*  \** ***@author*** *xzy  \** ***@date*** *2020-01-19 15:50  \* 说明：保姆类（静态代理模式）  \*/* public class Babysitter2 implements Human*{* private Human human;   public Babysitter2*() {* this.human = new Baby*()*;  *}*  @Override  public void eat*() {* this.washHandForChild*()*;  human.eat*()*;  *}* @Override  public void run*() {* human.run*()*;  *}* public void washHandForChild*() {* System.*out*.println*(*"help the child to wash his hands"*)*;  *} }* |

好，那么家长就是给孩纸找了个保姆代理，让他附加了一些婴儿做不了事。同时家长也没有强迫孩纸自己学会洗手(不更改Child类)：

|  |  |
| --- | --- |
| public class Client *{* public static void main*(*String*[]* args*) {  /\*\*  \* 装饰模式  \*/* Human babysitter = new Babysitter*(*new Baby*())*;  babysitter.eat*()*;  *} }* | public class Client *{* public static void main*(*String*[]* args*) {*  */\*\*  \* 静态代理模式  \*/* Human babysitter2 = new Babysitter2*()*;  babysitter2.eat*()*;  *} }* |



左：装饰模式 右：静态代理模式

装饰模式和静态代理模式的区别：

1. 装饰模式主要关心的是对类的功能进行拓展，代理模式主要关心的是对被代理对象访问的控制。
2. 使用代理模式的时候被代理对象通常由代理类创建，使用装饰模式的时候被装饰对象通常由外界传入。换句话说，用代理模式，代理类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；当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器的构造器。
3. 装饰器模式和代理模式的使用场景不一样，比如IO流使用的是装饰者模式，可以层层增加功能。而代理模式则一般是用于增加特殊的功能，有些动态代理不支持多层嵌套。
4. 代理和装饰其实从另一个角度更容易去理解两个模式的区别：代理更多的是强调对对象的访问控制，比如说，访问A对象的查询功能时，访问B对象的更新功能时，访问C对象的删除功能时，都需要判断对象是否登陆，那么我需要将判断用户是否登陆的功能抽提出来，并对A对象、B对象和C对象进行代理，使访问它们时都需要去判断用户是否登陆，简单地说就是将某个控制访问权限应用到多个对象上；而装饰器更多的强调给对象加强功能，比如说要给只会唱歌的A对象添加跳舞功能，添加说唱功能等，简单地说就是将多个功能附加在一个对象上。所以，代理模式注重的是对对象的某一功能的流程把控和辅助，它可以控制对象做某些事，重心是为了借用对象的功能完成某一流程，而非对象功能如何。而装饰模式注重的是对对象功能的扩展，不关心外界如何调用，只注重对对象功能加强，装饰后还是对象本身。