**Java 设计模式 装饰模式(Decorator)**

@author ixenos

**装饰模式是什么**

1.装饰模式以对客户端透明的方式**对象**的功能，**是继承关系的一个替代方案**，但装饰模式可以在不创造更多子类的情况下，对对象的功能进行扩展。

2.装饰模式以对客户端透明的方式**动态**地给一个对象附上更多的责任。换言之，客户端并不会觉得对象在装饰前后有什么不同。

3.装饰模式把对客户端的调用委派到被装饰类，而功能的增强在装饰类进行，装饰模式的关键在于这种扩展是完全透明的。

　　例如IO中的过滤流，具体就是BufferedInputStream的构造器接受一个节点流，而相关操作底层调用节点流的操作。

4.装饰模式是在不必改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。

5.本质上，利用了的接口的多态性

**装饰模式的特点**

1.装饰对象和真实对象**具有相同的接口**，这样客户端就可以以和真实对象相同的方式**与装饰对象进行交互**；

2.装饰对象**包含一个真实对象的引用**；

3.装饰对象接受来自客户端的请求，它把这些**请求转发给真实对象**；

4.装饰对象可以在**转发这些请求之前或之后增加一些附加功能**；

　　这样就确保了在运行时不用修改给定对象的结构就可以增加对象的功能，在面向对象设计中通常使用继承对类的功能进行扩展

**装饰模式的角色**

**抽象构件角色**（Component）：(如InputStream)给出一个抽象接口，作为共同接口，以规范接收附加责任的对象。

**具体构件角色**（Concrete Component）：(如FileInputStream)定义一个将要被附加责任的类，真实角色。

**装饰角色**（Decorator）：(如FilterInputStream)持有一个构件对象的引用，并定义一个与抽象构件接口一致的接口。

**具体装饰角色**（Concrete Decorator）：(如BufferedInputStream)负责给构件对象附加责任。

**抽象构件角色**：

1 public interface Component{

2 public void doSomeThing();

3 }

**具体构件角色**：

1 public class ConcreteComponent implements Component{

2

3 @Override

4 public void doSomeThing(){

5 System.out.println("功能A")；

6 }

7 }

**装饰角色**：

1 /\*

2 一、实现构件接口

3 二、内部持有构件变量，用以接受构件从而进行装饰

4 三、以此建立了一个装饰器，由装饰器子类来拓展功能

5 \*/

6 public class Decorator implements Component{

7 private Component component;

8

9 public Decorator(Component component){

10 this.component = component;

11 }

12

13 /\*借由真实构件来调用\*/

14 @Override

15 public void doSomeThing(){

16 component.doSomeThing();

17 }

18 }

**具体装饰角色**：

1 /\*

2 通过继承包装着构件的装饰器，另外添加功能；

3

4 而这对客户端来说是透明的，因为客户端把请求提交给构件，

5 产出也是构件产出，装饰器顺道唤醒了其他操作，

6 而这些操作可以用以优化处理过程或者增加额外的功能，

7 比如BufferedInputStream、DataInputStream等

8

9 \*/

10

11 public class ConcreteDecorator1 extends Decorator{

12 public ConcreteDecorator1(Component component){

13 super(component);

14 }

15

16 @Override

17 public void doSomeThing(){

18 super.doSomeThing();

19 //功能增加

20 this.doAnotherThing();

21 }

22

23 private void doAnotherThing(){

24 System.out.println("功能B")；

25 }

27 }

**另一个具体装饰角色**：

1 public class ConcreteDecorator2 extends Decorator{

2 public ConcreteDecorator2(Component component){

3 super(component);

4 }

5

6 @Override

7 public void doSomeThing(){

8 super.doSomeThing();

9 //功能增加

10 this.doAnotherThing();

11 }

12

13 private void doAnotherThing(){

14 System.out.println("功能C")；

15 }

16 }

**客户端**：

1 public Client{

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

3 Component component = new ConcreteComponent();

4 Component component1 = new ConcreteDecorator1(component);

5

6 component1.doSomeThing();

7

8 //又包装一层

9 Component component3 = new ConcreteDecorator2(conponent1);

10

11 component2.doSoneThing();

12

13 /\*或者直接用匿名对象直接多层包装，这是IO常用的做法\*/

14 Component component3 = new ConcreteDecorator2(new ConcreteDecorator1(new ConcreteComponent()));

15

16 component3.doSomeThing();

17 }

18 }

**装饰模式对比继承**

* 装饰模式
  + 用来扩展**特定对象**的功能
  + **动态**，**运行时**分配职责
  + 不需要生成特定对象的子类，**装饰对象相当于具有复用性的工具**

* 继承
  + 用来扩展**一类对象**的功能
  + **静态**，**编译时**分配职责
  + 需要生成大量对应功能的子类，不可复用

**装饰模式之于流的嵌套**

　　流的嵌套从装饰模式的角度来说过滤流之间是无所谓顺序的，但考虑到具体应用，执行输出的流在外层，缓冲流在中间是坠吼滴！

　　装饰模式的要求是：具体装饰对象给具体构件对象附加功能（注意是对象），要求扩展对象功能的同时，对客户端透明，这就需要具体装饰角色转发调用请求给具体构件角色，而具体装饰角色在转发的过程中可以对其进行扩展。具体到流就是过滤流内部维护一个从构造器接受的节点流，而各基本方法可以添加缓冲，再在合适的时候释放流，而且更可增加一些额外的功能。