## 装饰器模式（修饰模式）

定义：使用分层对象来动态的向单个对象添加功能。

给对象添加功能，要么直接修改对象添加相应的功能，要么派生对应的子类来扩展，亦或是使用对象组合的方式，面向对象是我们应该尽量使用对象组合，而不是对象继承

某些对象是可装饰的，可以通过将其他类包装在这个可装饰对象的四周，来将功能分层。

装饰器必须具有和他所装饰的对象相同的接口。

JavaIO中的应用：

Java I/O类库需要多种不同的功能组合，所以使用了装饰器模式。

FilterXxx类是JavaIO提供的装饰器基类，即我们要想实现一个新的装饰器，就要继承这些类。

装饰器与继承：

问题：

修饰模式做的增强功能按照继承的特点也是可以实现的，为什么还要提出修饰设计模式呢？

继承实现的增强类和修饰模式实现的增强类有何区别？

继承实现的增强类：

优点：代码结构清晰，而且实现简单

缺点：对于每一个的需要增强的类都要创建具体的子类来帮助其增强，这样会导致继承体系过于庞大。

修饰模式实现的增强类：

优点：内部可以通过多态技术对多个需要增强的类进行增强

缺点：需要内部通过多态技术维护需要增强的类的实例。进而使得代码稍微复杂。

 在以下两种情况下可以考虑使用装饰器模式：

    (1)需要在不影响其他对象的情况下，以动态、透明的方式给对象添加职责。

    (2)如果不适合使用子类来进行扩展的时候，可以考虑使用装饰器模式。

  装饰器模式的示例代码如下([**Java**](http://lib.csdn.net/base/java)语言描述)：

   (1)组件对象的接口，可以给这些对象动态的添加职责

**[java]** [view plain](http://blog.csdn.net/hust_is_lcd/article/details/7884320) [copy](http://blog.csdn.net/hust_is_lcd/article/details/7884320)

1. **public** **abstract** **class** Component {
2. /\*\*
3. \* 示例方法
4. \*/
5. **public** **abstract** **void** operation();
6. }

   (2)具体实现组件对象接口的对象

**[java]** [view plain](http://blog.csdn.net/hust_is_lcd/article/details/7884320) [copy](http://blog.csdn.net/hust_is_lcd/article/details/7884320)

1. **public** **class** ConcreteComponent **extends** Component {
3. **public** **void** operation() {
4. //相应的功能处理
5. }
7. }

    (3)装饰器接口，维持一个指向组件对象的接口对象， 并定义一个与组件接口一致的接口

**[java]** [view plain](http://blog.csdn.net/hust_is_lcd/article/details/7884320) [copy](http://blog.csdn.net/hust_is_lcd/article/details/7884320)

1. **public** **abstract** **class** Decorator **extends** Component {
2. /\*\*
3. \* 持有组件对象
4. \*/
5. **protected** Component component;
7. /\*\*
8. \* 构造方法，传入组件对象
9. \* @param component 组件对象
10. \*/
11. **public** Decorator(Component component) {
12. **this**.component = component;
13. }
15. **public** **void** operation() {
16. //转发请求给组件对象，可以在转发前后执行一些附加动作
17. component.operation();
18. }

21. }

    (4)装饰器的具体实现对象，向组件对象添加职责，operationFirst()，operationLast()为前后需要添加的功能。具体的装饰器类ConcreteDecoratorB代码相似，不在给出。

**[java]** [view plain](http://blog.csdn.net/hust_is_lcd/article/details/7884320) [copy](http://blog.csdn.net/hust_is_lcd/article/details/7884320)

1. **public** **class** ConcreteDecoratorA **extends** Decorator {
2. **public** ConcreteDecoratorA(Component component) {
3. **super**(component);
4. }
5. **private** **void** operationFirst(){ } //在调用父类的operation方法之前需要执行的操作
6. **private** **void** operationLast(){ } //在调用父类的operation方法之后需要执行的操作
7. **public** **void** operation() {
8. //调用父类的方法，可以在调用前后执行一些附加动作
9. operationFirst(); //添加的功能
10. **super**.operation();  //这里可以选择性的调用父类的方法，如果不调用则相当于完全改写了方法，实现了新的功能
11. operationLast(); //添加的功能
12. }
13. }

    (5) 客户端使用装饰器的代码

**[java]** [view plain](http://blog.csdn.net/hust_is_lcd/article/details/7884320) [copy](http://blog.csdn.net/hust_is_lcd/article/details/7884320)

1. **public** **class** Client{
2. **public** **static** **void** main(String[] args){
3. Component c1 = **new** ConcreteComponent (); //首先创建需要被装饰的原始对象(即要被装饰的对象)
4. Decorator decoratorA = **new** ConcreteDecoratorA(c1); //给对象透明的增加功能A并调用
5. decoratorA .operation();
6. Decorator decoratorB = **new** ConcreteDecoratorB(c1); //给对象透明的增加功能B并调用
7. decoratorB .operation();
8. Decorator decoratorBandA = **new** ConcreteDecoratorB(decoratorA);//装饰器也可以装饰具体的装饰对象，此时相当于给对象在增加A的功能基础上在添加功能B
9. decoratorBandA.operation();
10. }
11. }