← 组合模式 外观模式 →

装饰器模式

装饰器模式（Decorator Pattern）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。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

我们通过下面的实例来演示装饰器模式的用法。其中，我们将把一个形状装饰上不同的颜色，同时又不改变形状类。

介绍

意图：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

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。

何时使用：在不想增加很多子类的情况下扩展类。

如何解决：将具体功能职责划分，同时继承装饰者模式。

关键代码： 1、Component 类充当抽象角色，不应该具体实现。 2、修饰类引用和继承 Component 类，具体扩展类重写父类方法。

应用实例： 1、孙悟空有 72 变，当他变成"庙宇"后，他的根本还是一只猴子，但是他又有了庙宇的功能。 2、不论一幅画有没有画框都可以挂在墙上，但是通常都是有画框的，并且实际上是画框被挂在墙上。在挂在墙上之前，画可以被蒙上玻璃，装到框子里；这时画、玻璃和画框形成了一个物体。

优点：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。

缺点：多层装饰比较复杂。

使用场景： 1、扩展一个类的功能。 2、动态增加功能，动态撤销。

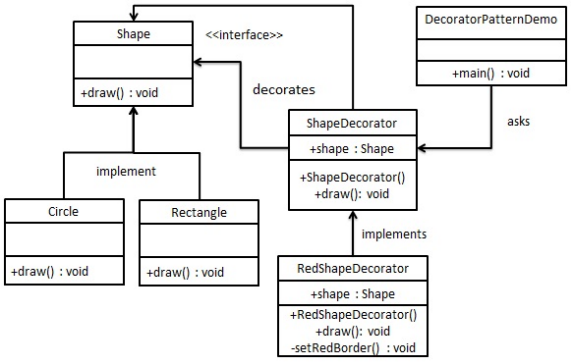
注意事项：可代替继承。

实现

我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类。然后我们创建一个实现了 *Shape* 接口的抽象装饰类 *ShapeDecorator*，并把 *Shape* 对象作为它的实例变量。

RedShapeDecorator 是实现了 *ShapeDecorator* 的实体类。

DecoratorPatternDemo，我们的演示类使用 *RedShapeDecorator* 来装饰 *Shape* 对象。



步骤 1

创建一个接口：

Shape.java

```
public interface Shape {
   void draw();
}
```

步骤 2

创建实现接口的实体类。

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

步骤 3

创建实现了 *Shape* 接口的抽象装饰类。

ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

HTML / CSS
JavaScript
服务端
数据库
移动端
XML 教程
ASP.NET
Web Service
开发工具
网站建设

步骤 4

创建扩展了 *ShapeDecorator* 类的实体装饰类。

RedShapeDecorator.java

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

步骤 5

使用 *RedShapeDecorator* 来装饰 *Shape* 对象。

DecoratorPatternDemo.java

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

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

步骤 6

执行程序，输出结果：

```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

← 组合模式 外观模式 →

2 篇笔记 **写笔记**

一个更易理解的实例：

装饰模式为已有类动态附加额外的功能就像LOL、王者荣耀等类Dota游戏中，英雄升级一样。每次英雄升级都会附加一个额外技能点学习技能。具体的英雄就是ConcreteComponent，技能栏就是装饰器Decorator，每个技能就是ConcreteDecorator；

```
//Component 英雄接口
public interface Hero {
    //学习技能
    void learnSkills();
}
//ConcreteComponent 具体英雄盲僧
public class BlindMonk implements Hero {

    private String name;

    public BlindMonk(String name) {
        this.name = name;
    }

    @Override
    public void learnSkills() {
        System.out.println(name + "学习了以上技能！");
    }
}
//Decorator 技能栏
public class Skills implements Hero{

    //持有一个英雄对象接口
    private Hero hero;

    public Skills(Hero hero) {
        this.hero = hero;
    }

    @Override
    public void learnSkills() {
        if(hero != null)
            hero.learnSkills();
    }
}
//ConreteDecorator 技能：Q
```

```
public class Skill_Q extends Skills{

    private String skillName;

    public Skill_Q(Hero hero,String skillName) {
        super(hero);
        this.skillName = skillName;
    }

    @Override
    public void learnSkills() {
        System.out.println("学习了技能Q:" +skillName);
        super.learnSkills();
    }
}
//ConreteDecorator 技能：W
public class Skill_W extends Skills{

    private String skillName;

    public Skill_W(Hero hero,String skillName) {
        super(hero);
        this.skillName = skillName;
    }

    @Override
    public void learnSkills() {
        System.out.println("学习了技能W:" + skillName);
        super.learnSkills();
    }
}
//ConreteDecorator 技能：E
public class Skill_E extends Skills{

    private String skillName;

    public Skill_E(Hero hero,String skillName) {
        super(hero);
        this.skillName = skillName;
    }

    @Override
    public void learnSkills() {
        System.out.println("学习了技能E:"+skillName);
        super.learnSkills();
    }
}
//ConreteDecorator 技能：R
public class Skill_R extends Skills{

    private String skillName;

    public Skill_R(Hero hero,String skillName) {
        super(hero);
        this.skillName = skillName;
    }

    @Override
    public void learnSkills() {
        System.out.println("学习了技能R:" +skillName );
        super.learnSkills();
    }
}
//客户端：召唤师
public class Player {
    public static void main(String[] args) {
        //选择英雄
        Hero hero = new BlindMonk("李青");

        Skills skills = new Skills(hero);
        Skills r = new Skill_R(skills,"猛龙摆尾");
        Skills e = new Skill_E(r,"天雷破/摧筋断骨");
        Skills w = new Skill_W(e,"金钟罩/铁布衫");
        Skills q = new Skill_Q(w,"天音波/回音击");
        //学习技能
        q.learnSkills();
    }
}
```

输出：

```
学习了技能Q:天音波/回音击
学习了技能W:金钟罩/铁布衫
学习了技能E:天雷破/摧筋断骨
学习了技能R:猛龙摆尾
李青学习了以上技能！
```

周霆 1年前 (2017-10-02)

游戏里英雄皮肤的实现 是不是也比较适合装饰器模式

```
public interface Hero {
    public void init();
}

public class victor implements Hero {
    @Override
    public void init() {
        System.out.println("维克托：输出型英雄 武器：步枪");
    }
}

public abstract class HeroDecorator implements Hero {
    private Hero heroDecorator;

    public HeroDecorator(Hero heroDecorator) {
        this.heroDecorator = heroDecorator;
```

```
    }

    @Override
    public void init() {
        heroDecorator.init();
    }
}

public class GalacticWarriors extends HeroDecorator {
    private Hero heroDecorator;

    public GalacticWarriors(Hero heroDecorator) {
        super(heroDecorator);
    }

    @Override
    public void init() {
        super.init();
        setSkin();
    }

    private void setSkin() {
        System.out.println("皮肤:银河战士套装");
    }
}

public class DecoratorPatternDemo {
    public static void main(String[] args) {
        Hero victor = new victor();
        GalacticWarriors galacticWarriors = new GalacticWarriors(victor);
        galacticWarriors.init();
    }
}
```

该用户昵称违规 7个月前 (04-16)