转载请注明链接：https://blog.csdn.net/feather_wch/article/details/78490144

> 本文以面试题的形式归纳总结，可以直接学习和背诵。
> 鸣谢: 本文基础部分归纳总结自《Head First 设计模式》

如果有帮助，请点个赞，万分感谢！

装饰者模式

版本：2018/8/24-1(14:02)

介绍

1、什么是装饰者模式?

> 1. 装饰者模式(Decorator Pattern)
> 2. 允许向一个现有的对象添加新功能，又不改变其结构。
> 3. 属于 结构型模式
> 4. 现有的类的一个包装类。
> 5. 装饰者包装一个组件，本质是装饰者内部持有组件的一个引用。

定义

2、装饰者模式的定义?

> 动态的将责任附加到对象身上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

设计原则

3、类应该对扩展开放，对修改关闭。

实例

4、实例：星巴克咖啡

> 场景：星巴克需要出售各种饮料，并且在饮料上加上调料，比如+奶泡+豆浆的浓缩咖啡，比如+摩卡和豆浆的烘焙咖啡。
>
> 1. Beverage饮料类（抽象组件）
> 2. Condiment调料类(抽象装饰者)
> 3. 具体饮料类(继承Beverage)
> 4. 具体调料类(Condiment)
>
> Beverage(抽象组件):

```
public abstract class Beverage {
        String description = "unKnow description";
        public String getDescription() {
                return description;
        }
        public void setDescription(String description) {
                this.description = description;
        }
        public abstract double cost();
}
```

> 具体饮料：浓缩咖啡、烘焙咖啡等等

```
// Espresso、DarkRoast、HouseBlend 都同理
public class Espresso extends Beverage{
        public Espresso(){
                description = "Espresso ";
        }
        public double cost() {
                return 8;
        }
}
```

> Condiment调料类(抽象装饰者):

```
public abstract class Condiment extends Beverage{
        public abstract String getDescription();
}
```

> 具体调料(装饰者)-奶泡、豆浆、摩卡

```
// Whip\Soy\Mocha 都同理
public class Mocha extends Condiment{
  // 内部持有组件的引用
        Beverage beverage;
        public Mocha(Beverage beverage) {
                this.beverage = beverage;
        }

        public String getDescription() {
                return beverage.getDescription() + ", Mocha ";
        }

        public double cost() {
                return beverage.cost() + 1;
        }

}
```

> 测试:

```
                Beverage beverage = new Espresso();
                System.out.println(beverage.getDescription()+beverage.cost());

                beverage = new HouseBlend();
                beverage = new Soy(beverage);
                beverage = new Whip(beverage);
                beverage = new Mocha(beverage);
                System.out.println(beverage.getDescription()+beverage.cost());

                beverage = new DarkRoast();
                beverage = new Whip(beverage);
                beverage = new Soy(beverage);
                System.out.println(beverage.getDescription()+beverage.cost());
```

5、实现方法总结

> 1. 定义一个抽象组件Component---为所有类的超类
> 2. 定义抽象装饰者Decorator，继承自抽象组件C
> 3. 实现各种具体组件，都继承自抽象组件C
> 4. 实现各种具体装饰者，都继承自抽象装饰者D
> 5. 使用代码：

```
Component c = new ChildComponent(); //某个具体组件
c = new DecoratorOne(c); //进行一层装饰
c = new DecoratorTwo(c); //进行第二层装饰
c = new DecoratorThree(c); //进行第三层装饰
c.doSomeThing(); //用最终装饰好的组件去做一些操作
```

问题反思

6、 实现过程中使用了继承，不是说利用装饰而避免继承的吗?

> 1. 现象：装饰者(Decorator)继承自组件(Component): `public abstract class Condiment extends Beverage`
> 2. 这里实际意义是“装饰”。
> 3. 通过继承只是为了“类型匹配”而不是为了继承获得某些行为。
> 4. 如：只是为了装饰者和组件类型匹配，从而层层装饰。

```
Beverage beverage = new HouseBlend();
// 装饰者对象可以继续作为组件对象进行包装
beverage = new Soy(beverage);
beverage = new Whip(beverage);
```

7、装饰者模式是如何如何获得行为?

> 在“装饰”的过程中,添加新的行为，而不是继承自超类，这就是组合对象。

8、通常装饰者模式采用抽象类实现，而java中可以用接口实现。

Java中的应用场景

9、装饰者模式在Java中有那些典型的应用场景呢?

> Java I/O:
>
> 1. `FileInputStream` > `BufferedInputStream` > `LineNumberInputStream` 层层包装。

10、Java IO中的装饰者模式

> 1. FileInputStream(被装饰组件，提供最基本的字节读取功能)
> 2. BufferedInputStream(装饰者：增加利用缓冲输入增强性能和用readline()方法来增强接口两种行为)
> 3. LineNumberInputStream(装饰者：加上计算行数的能力)

```
File file = new File("bufferedinputstream.txt");
InputStream in = new BufferedInputStream(
                        new FileInputStream(file), 512);
```

11、用装饰者模式中的组件和装饰者来拆分Java IO

> 1. 抽象组件：InputStream
> 2. 具体组件(被装饰者)：FileInputStream、StringBufferInputStream、ByteArrayInputStream
> 3. 抽象装饰者：FilterInputStream
> 4. 具体装饰者：BufferedInputStream(具有缓冲的输入流)、DataInputSream等等

12、Java IO采用装饰者的缺点

> 设计中包含大量的小类，导致使用该API程序员的困扰。

实现自己的Java I/O

13、实现一个InputStream用于将得到的字符串中的大写字母都转换为小写模式。

> 1. 主要实现方法就是，继承FilterInputStream然后重写 `read()` 和 `read(byte b[], int offset, int len)` 方法。

LowerCaseInputStream:

```
public class LowerCaseInputStream extends FilterInputStream{

        protected LowerCaseInputStream(InputStream arg0) {
                super(arg0);
        }
//for byte
        public int read() throws IOException{
                int c = super.read();
                return ((c == -1)? c : Character.toLowerCase((char)c));
        }
//for byte array
        public int read(byte b[], int offset, int len) throws IOException{
                int result = super.read(b, offset, len);
                for(int i = offset; i < offset + result; i++) {
                        b[i] = (byte)(Character.toLowerCase((char)b[i]));
                }
                return result;
        }

}
```

> 测试程序(需要写好一个txt文件，这里放在E盘):

```
        int c;
        try {
                //nested Decorators
                InputStream inputStream =
                                new LowerCaseInputStream(
                                        new BufferedInputStream(
                                                new FileInputStream("E:\\test.txt")));

                while((c = inputStream.read()) >= 0) {
                        System.out.print((char)c);
                }

        } catch (IOException e) {
                e.printStackTrace();
        }
```

参考资料

1. Head First 设计模式
2. 装饰器模式 | 菜鸟教程
3. BufferedInputStream(缓冲输入流)的认知、源码和示例