1 装饰者模式核心
2 饮料加配料示例
    2.1 抽象组件（饮料）
    2.2 具体组件（Coffee）
    2.3 抽象装饰者（配料）
    2.4 具体装饰者类（Mocha）
    2.5 客户端代码
3 JavaIO 示例
4 UML 类图
5 装饰者模式的特点
6 装饰者模式的适用性
参考

1 装饰者模式核心

- 核心类：Component(组件类)、Decorator(装饰者类)
- 具体组件：继承于抽象组件，实现特定的核心功能
- 具体装饰者：继承于抽象装饰者，包装抽象组件，对抽象组件的核心功能（方法）做修饰
- **关键：**
    - 客户端不会感觉到装饰过的组件与未装饰过的组件之间的差异（**运用了多态，其静态类型是抽象组件类**）
    - 装饰者的目的在于装饰抽象组件的方法，**而不是添加新的方法（功能）**，即使添加新的 public 方法，由于面向接口，只能使用父类中的方法，导致新添加的方法不起作用
    - 即装饰者模式改变的只是外表（通过不同的外表来装饰核心，例如QQ空间相同的内容，不同的展示风格），核心并没有改变

2 饮料加配料示例

- 针对饮料有很多种，而配料也有很多种的情况
- 如果使用配料直接继承饮料的话，那就会导致“类型爆炸”
- 使用装饰者模式可以有效的避免创建太多的子类对象

2.1 抽象组件（饮料）

- 饮料有它的描述，饮料的大小，和cost()

```
public abstract class Beverage {
    protected String description = "Unknown Beverage";//需要被子类继承，所有不能定义为私有
    //因为所有的具体组件都返回的是 description，所以不需要设置为抽象类
    public String getDescription() {
        return description;
    }
    public abstract double cost();
    protected Size size;
}
```

2.2 具体组件（Coffee）

- Coffee有固定的属性，且默认大小为中杯
- 具体组件也可以通过工厂方法或其他创建者模式创建

```
public class Coffee extends Beverage {
    public Coffee() {
        this.description = "coffee";
        this.size = Size.GRADNDE;//默认为中杯
    }
    public Coffee(Size size) {
        this.description = "coffee";
        this.size = size;
    }
    //根据不同的大小返回不同的值
    @Override
    public double cost() {
        if (size == Size.VENTI) {
            return 2.8;
        } else if (size == TALL){
            return 0.8;
        } else {
            return 1.8;
        }
    }
}
```

2.3 抽象装饰者（配料）

- 抽象装饰者继承了抽象组件，且关联了一个抽象组件（相当于代理），真正调用的最终还是关联的抽象组件

```
public abstract class CondimentDecorator extends Beverage {
    protected Beverage beverage;
    //因为饮料的价格只和饮料的种类和饮料的大小、配料的种类有关，不和配料的大小有关，所以这个Size属性应该是抽象组件的属性
    //protected Size size;//装饰者也可以有自己的属性，如果是公共的就可以提到抽象装饰者中，私有属性一般是用来更加不同的值，做不同的装饰吧
    //这里定义一个抽象方法是因为被配料装饰后，返回的描述是根据配料的类型来变化的，
    //所以必须使用方法来返回，且每个子类配料的方法内容都不一样，所以使用抽象方法，让子类实现
    //这里也说明了，为什么抽象组件中也有getDescription()方法，而不是直接通过属性调用返回，因为饮料被配料装饰后，返回描述时必须使用方法来方法，所以抽象组件类（所有类的父类）中必须有一个这样的接口，这样多态时子类才能拥有这个方法
    public abstract String getDescription();
}
```

2.4 具体装饰者类（Mocha）

- 具体装饰者类需要重新 getDescription()、cost() 方法
- 这里可以看出 getDescription()、cost() 方法都进行了递归操作

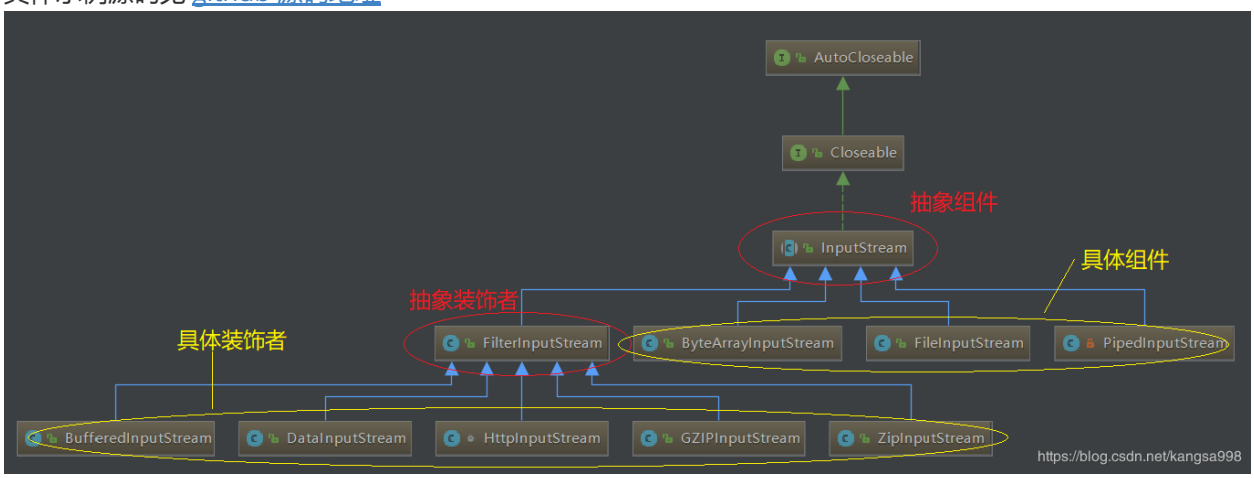
```
public class Mocha extends CondimentDecorator {
    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
    @Override
    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }
    @Override
    public double cost() {
        return beverage.cost() + 0.33;
    }
}
```

2.5 客户端代码

```
    @Test
    public void test() {
        Beverage beverage = new Coffee(Size.TALL);//创建一个具体组件
        beverage = new Milk(beverage);//用Milk装饰类包装它
        beverage = new Mocha(beverage);//用Mocha装饰类包装它
        System.out.println(beverage.getDescription());
        System.out.println(beverage.cost());
    }
```

3 JavaIO 示例

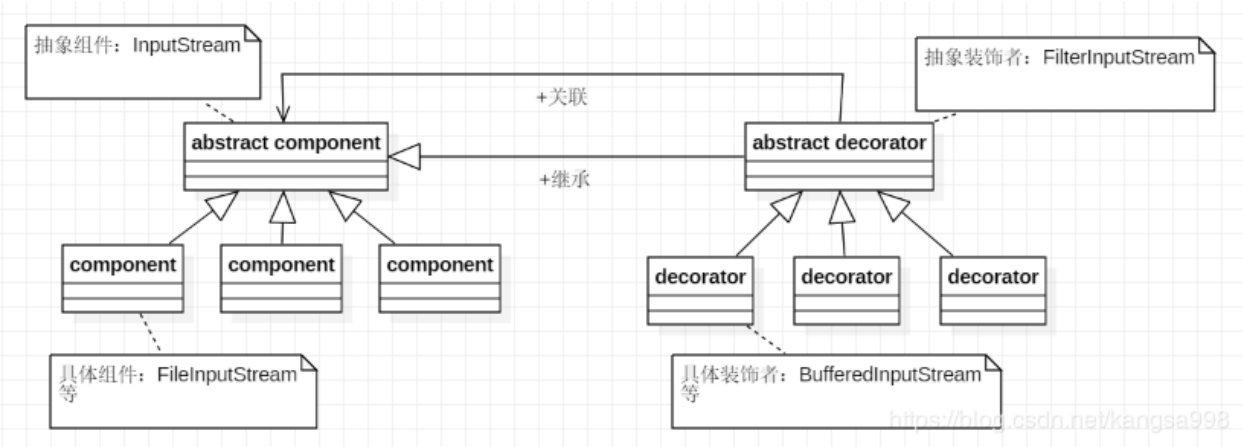
- java.io 包大量运用了装饰者模式
- **抽象组件：** InputStream，大量的具体组件通过继承 InputStream 来实现相关功能
- **抽象装饰者：** FilterInputStream，我们可以通过继承它来实现自定义的装饰功能
- 具体示例源码见 github 源码地址



4 UML 类图

- 可以没有抽象装饰者，直接通过具体装饰者来继承、依赖抽象组件

- 一般情况下都是有抽象组件的，不然达不到扩展的目的



5 装饰者模式的特点

- 可以很容易的重复添加一个特性
- 可以递归的嵌套多个装饰
- 对客户透明，客户不知道某个具体组件有没有被装饰
- 符合开发-闭合原则：通过新增具体组件和具体装饰者来达到扩展的目的

6 装饰者模式的适用性

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责
- 当通过直接继承生成子类的数量太多时，用装饰者模式

参考

大话设计模式 Head First 设计模式 设计模式 github 源码地址