第02课：装饰器模式

> 装饰器模式动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

某一天隔壁老王赤果果地来到百货商店，打算给自己买一套装备，武装到牙齿。他买了衣服裤子和帽子，于是老王这样做：

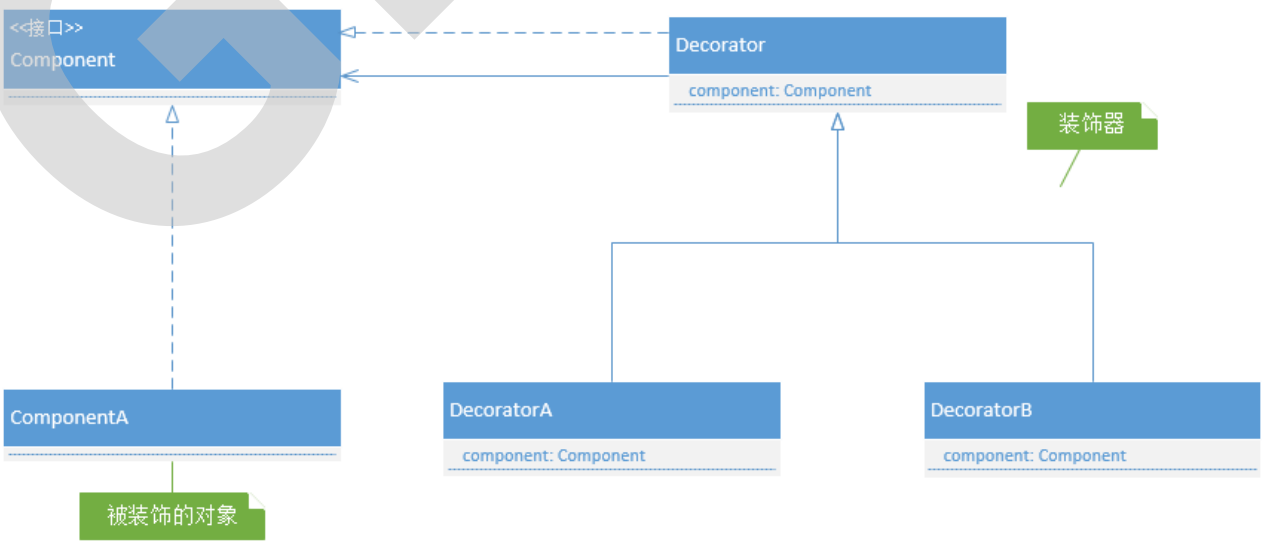
```
public class LaoWang {

    public void show(){
        System.out.println("我穿上衣服，累计花费100元");
        System.out.println("我穿上裤子，累计花费250元");
        System.out.println("我穿上帽子，花费300元");
        //........
        //........
    }
}
```

但老王很快发现了问题，每买一件装备都要修改一次 show() 方法，违背了开闭原则。在前面的内容中已经解释过开闭原则：对扩展开放、对修改关闭。每增加一件装备都相当于扩展了一个功能，我们不应该用修改原方法的方式来扩展功能。

机智的老王学过设计模式，很快就想到了使用装饰器模式，装饰器模式动态地把功能附加到对象上。

装饰器模式类图：



观察上图，装饰器模式中主要有两个角色：

- 装饰器
- 被装饰的对象

用老王买装备的例子来说，老王就是被装饰的对象，而衣服裤子帽子等就是装饰器。

装饰器和被装饰的对象有两个特点，也是装饰器模式的关键：

- 他们实现同一个接口；
- 装饰器中使用了被装饰的对象。

代码实现

下面我们来简单的实现上面的例子。

老王（被装饰的对象）：

```
public interface Person {
    /**
     * 计算累计消费
     * @return
     */
    public Double cost();
    public void show();
}

public class LaoWang implements Person{

    @Override
    public Double cost() {
        return 0.0; //赤果果的时候累计消费为0
    }

    @Override
    public void show() {
        System.out.println("我是赤果果的老王");
    }
}
```

装饰器超类，和被装饰的对象实现同一个接口 Person：

```
public abstract class ClothesDecorator implements Person {
    //装饰器中要使用被装饰器的对象，构造方法中传入
    protected Person person;

    public ClothesDecorator(Person person){
        this.person = person;
    }

}
```

具体的装饰，夹克和帽子：

```
public class Jacket extends ClothesDecorator {

    public Jacket(Person person) {
        super(person);
    }
    @Override
    public void show() {
        person.show();
        System.out.println("穿上夹克，累计消费" + this.cost());
    }

    @Override
    public Double cost() {
        return person.cost() + 100; //夹克100元
    }
}
public class Hat extends ClothesDecorator {

    public Hat(Person person) {
        super(person);
    }

    @Override
    public void show() {
        //执行已有功能
        person.show();
        //此处是附加的功能
        System.out.println("戴上帽子，累计消费" + this.cost());
    }

    @Override
    public Double cost() {
        return person.cost() + 50; //帽子50元
    }
}
```

测试：

```
Person laowang = new LaoWang();
//穿上夹克
laowang = new Jacket(laowang);
//戴上帽子
laowang = new Hat(laowang);

laowang.show();
System.out.println("买单，老王总共消费："+laowang.cost());
```

效果：

```
我是赤果果的老王
穿上夹克，累计消费100.0
戴上帽子，累计消费150.0
买单，老王总共消费：150.0
```

用了装饰器模式，老王还想穿裤子、鞋子，只要分别创建裤子、鞋子的装饰类就可以动态地穿上了，而不用修改已写好的类，深入贯彻落实了**开闭原则**。

使用装饰器模式的几个关键点：

- **装饰器**和**被装饰类**要实现同一个接口（实际开发中也可能用继承）。
- 装饰器中的方法可以调用被装饰对象提供的方法，以此实现功能累加的效果，例如，夹克装饰器和帽子装饰器中调用了 `person.cost() + xx` 实现累计消费金额的累加。

实际案例

Java 中的 IO 流

在学习 Java 基础的过程中，学习 IO 流是必不可少的，同时这也是最令人头疼。我们先来简单地写一下 IO 流的应用。

先准备一个文本文件：

```
hello world!
I am laowang
```

用输入流读取内容：

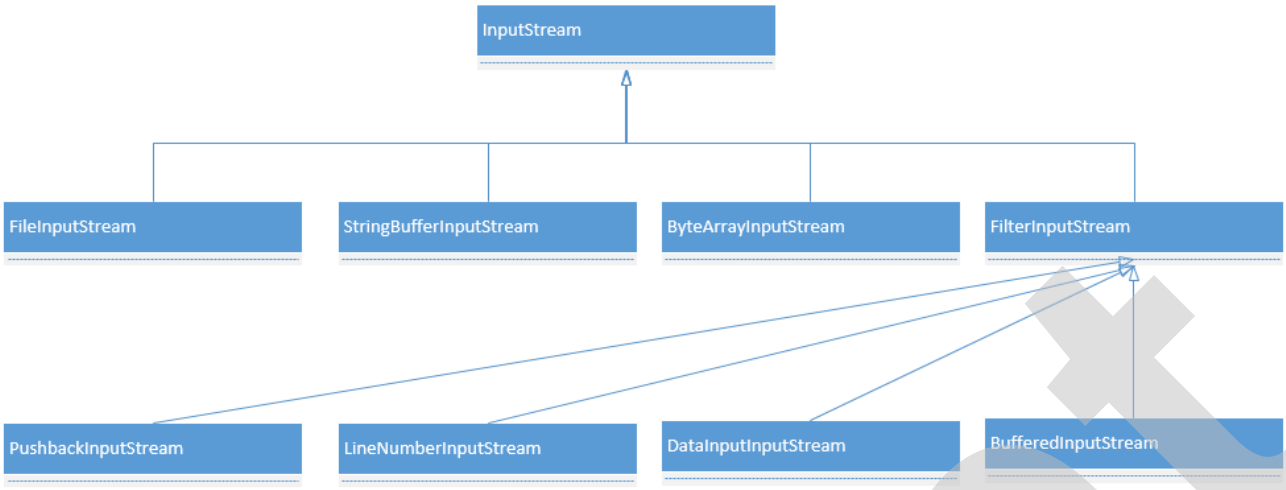
```
DataInputStream in = new DataInputStream(new FileInputStream("test.txt"));
String str;
while((str =  in.readLine()) != null){
    System.out.println(str);
}
in.close();
```

结果：

```
"D:\Program Files (
hello world!
I am laowang
```

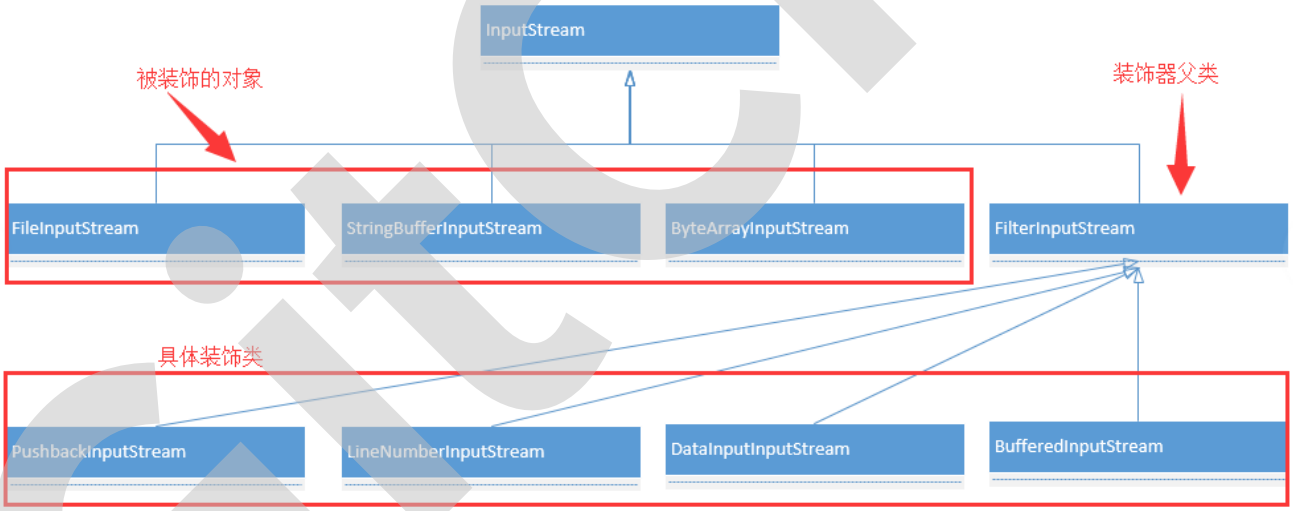
笔者刚入门 Java 的时候，学到上面的代码，其实还算简单，但后来又了解到还有 BufferedInputStream、StringBufferInputStream 等等。许多 InputStream 子类互相嵌套，除了输入流，还有输出流 OutputStream，以及各种字符流 Writter 和 Reader，庞大的 IO 流家族简直令人崩溃。

后来学习看了下 IO 流的族谱，总算对此有了个大概的了解：



上图是 InputStream 是家族，OutputStream 也是类似。看了上图，对整体结构清晰了不少，但是对 `new DataInputStream(new FileInputStream("test.txt"));` 这种调用方式还是一知半解，不明白**为什么这样嵌套？到底谁嵌套谁？**

直到学习了装饰器模式，赫然发现 IO 家族族谱怎么长得这么像装饰器模式的类图？不妨回到文章开头将装饰器类图和 InputStream 族类图对比一下，你会发现 InputStream 类其实就是被装饰对象的超类，FileInputStream、StringBufferInputStream 等就是要被装饰的对象，FilterInputStream 以及其子类就是装饰器。



回顾老王穿衣服的代码：

```
Person laowang = new LaoWang();
//穿上夹克
laowang = new Jacket(laowang);
```

换种写法：

```
Person laowang = new Jacket(new LaoWang());
```

是不是和 `new DataInputStream(new FileInputStream("test.txt"));` 一模一样？

学到这里，有种豁然开朗的感觉，原来 IO 流根本不难，只要了解其家族成员，谁嵌套谁的问题也就迎刃而解。

根据装饰器模式，我们来自定义一个装饰器，将所有英文字母转换为空格：

```
public class CharacterInputStream extends FilterInputStream {

    public CharacterInputStream(InputStream in) {
        super(in);
    }

    @Override
    public int read() throws IOException {
        //ASCLL码对照,[97,122] 和 [65,90]是英文字母
        int c = super.read();
        if(c >= 97 && c <= 122 || c >= 65 && c <= 90){
            return 32; //32是空格
        }else{
            return c;
        }
    }

}
```

测试

准备一个文本：

```
hello/world!
I am*&^ laowang*632
```

```
DataInputStream in = new DataInputStream(
                        new CharacterInputStream(
                            new FileInputStream("test.txt")));

String str;
while((str =  in.readLine()) != null){
    System.out.println(str);
}
```

结果：

文本中的英文字母成功地被转换成了空格。

IO 家族中的输出流，字符输入输出流等都是同样的道理，此处不再详述。

JSON 格式化日志

打印日志是开发中最常做的事，是调试 bug 最重要的手段之一。

有一天隔壁老王公司的系统出了个问题，老王查看日志时，其中有一段业务日志是一个 JSON 字符串，内容是用户银行卡数据：


```
16:26:39.609 [main] ERROR com.itzhoujun.design.Demo -  {"code": 0, "data": {"status": 1,"number": "215646454","account_name": "吴系挂","type":"中国银行" ,"address": "某某支行
 ","icon": "http://xxx.xxx.xx"}}
16:26:39.611 [main] ERROR com.itzhoujun.design.Demo -  {"code": 0, "data": {"status": 1,"number": "215646454","account_name": "吴系挂","type":"中国银行" ,"address": "某某支行
 ","icon": "http://xxx.xxx.xx"}}
16:26:39.611 [main] ERROR com.itzhoujun.design.Demo -  {"code": 0, "data": {"status": 1,"number": "215646454","account_name": "吴系挂","type":"中国银行" ,"address": "某某支行
 ","icon": "http://xxx.xxx.xx"}}
16:26:39.611 [main] ERROR com.itzhoujun.design.Demo -  {"code": 0, "data": {"status": 1,"number": "215646454","account_name": "吴系挂","type":"中国银行" ,"address": "某某支行
 ","icon": "http://xxx.xxx.xx"}}
16:26:39.611 [main] ERROR com.itzhoujun.design.Demo -  {"code": 0, "data": {"status": 1,"number": "215646454","account_name": "吴系挂","type":"中国银行" ,"address": "某某支行
 ","icon": "http://xxx.xxx.xx"}}
```


老王发现大量的 JSON 日志堆积在一起时，很难用肉眼快速筛选中自己想要的数据。怎么办？

难道每次都要把日志内容复制出来用工具转换一下吗？太麻烦了。

于是老王就想如果打印日志的时候能够自动将 JSON 格式的日志格式化显示就好了。

可是老王用的是 Apache 的 Log4j 打印日志，不可能去改源码，咋整？经验丰富的老王想到了装饰器模式，动态地给日志增加功能。

首先我们需要一个将 JSON 字符串格式化的工具类，下方格式化 JSON 的具体代码逻辑可以不看，与装饰器模式本身无关，笔者也是直接从网上抄来的。

```
package com.itzhoujun.design.decorator;

public class Json {

    private static String getLevelStr(int level) {
        StringBuffer levelStr = new StringBuffer();
        for (int levelI = 0; levelI < level; levelI++) {
            levelStr.append("\t");
        }
        return levelStr.toString();
    }

    public static String format(String s){
        int level = 0;
        //存放格式化的json字符串
        StringBuffer jsonForMatStr = new StringBuffer();
        for(int index=0;index<s.length();index++)//将字符串中的字符逐个按行输出
        {
            //获取s中的每个字符
            char c = s.charAt(index);

            //level大于0并且jsonForMatStr中的最后一个字符为\n,jsonForMatStr加入\t
            if (level > 0 && '\n' == jsonForMatStr.charAt(jsonForMatStr.length() - 1)) {
                jsonForMatStr.append(getLevelStr(level));
            }
            //遇到"{"和"["要增加空格和换行，遇到"}"和"]"要减少空格，以对应，遇到","要换行
            switch (c) {
                case '{':
                case '[':
                    jsonForMatStr.append(c + "\n");
                    level++;
                    break;
                case ',':

                    jsonForMatStr.append(c + "\n");
```

```
                    break;
                case '}':
                case ']':
                    jsonForMatStr.append("\n");
                    level--;
                    jsonForMatStr.append(getLevelStr(level));
                    jsonForMatStr.append(c);
                    break;
                default:
                    jsonForMatStr.append(c);
                    break;
            }
        }
        return jsonForMatStr.toString();
    }
}
```

重点是装饰类：

```
package com.itzhoujun.design.decorator;
import org.apache.log4j.Logger;

public class JsonFormatLoggerDecorator extends Logger {

    protected static Logger logger;

    public JsonFormatLoggerDecorator(Logger logger){
        super(JsonFormatLoggerDecorator.class.getName());
        JsonFormatLoggerDecorator.logger = logger;
    }

    @Override
    public void error(Object message) {
        if(message instanceof String){ //严格来说这里要判断是否是json格式
            logger.error("\n"+Json.format((String)message));
        }else{
            logger.error(message);
        }
    }
}
```

把装饰的过程写到工厂类中：

```
package com.itzhoujun.design.decorator;

import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;

public class MyLoggerFactory {

    public static Logger getLogger(String name){
        //此处可以进行多层装饰，给日志增加多个功能
        return new JsonFormatLoggerDecorator(LogManager.getLogger(name));
    }
}
```

测试：

```
String str= " {\"code\": 0, \"data\": {\"status\": 1,\"number\": \"215646454\",\"account_name\":
\"吴系挂\",\"type\":\"中国银行\" ,\"address\": \"某某支行\",\"icon\": \"http://xxx.xxx.xx\"}}";

//此处可以写成类静态变量
Logger logger = MyLoggerFactory.getLogger(Demo.class.getName());
//实际开发时，可以传入对象，在装饰类中将对象转换成json字符串。此处只是测试所以直接传入json字符串
logger.error(str);
```

效果：

除了格式化 JSON，将来老王还能随心所欲的给日志附加其他功能，例如老王觉得日志挤在一起很起来很不舒服，想要给每一条日志都加一行空行，只要写一个装饰类，然后在 MyLoggerFactory 进行装饰即可。

总结

装饰器模式的作用是动态给对象增加一些功能，而不需要修改对象本身。

优点：

- 扩展功能的方式比较灵活；
- 每一个装饰器相互独立，需要修改时不会互相影响。

缺点：

多层装饰比较复杂，就像 Java IO 流，对于初学者不友好。