私塾在线《研磨设计模式》
——跟着CC学设计系列精品教程

本节课程概览

- 学习装饰模式

一：初识装饰模式

包括：定义、结构、参考实现

二：体会装饰模式

包括：场景问题、不用模式的解决方案、使用模式的解决方案

三：理解装饰模式

包括：认识装饰模式、Java中的装饰模式应用、装饰模式和AOP 、装饰模式的优缺点

四：思考装饰模式

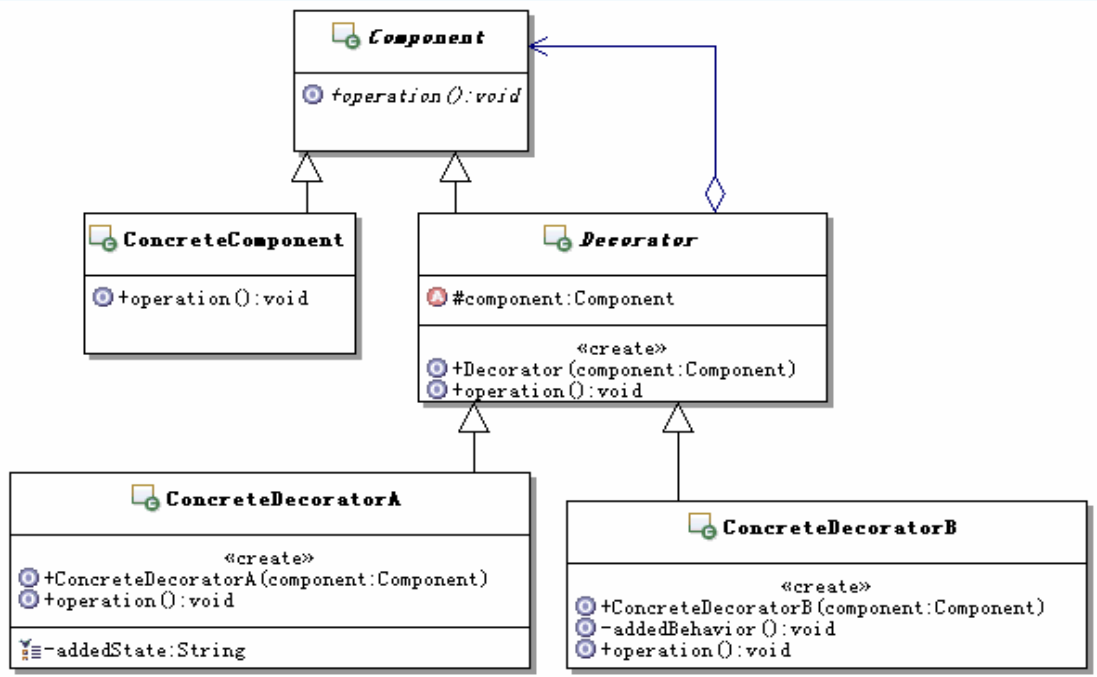
包括：装饰模式的本质、何时选用

初识装饰模式

- 定义

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

- 结构和说明



体会装饰模式

Component：

组件对象的接口，可以给这些对象动态的添加职责。

ConcreteComponent：

具体的组件对象，实现组件对象接口，通常就是被装饰器装饰的原始对象，也就是可以给这个对象添加职责。

Decorator：

所有装饰器的抽象父类，需要定义一个与组件接口一致的接口，并持有一个Component对象，其实就是持有一个被装饰的对象。

注意这个被装饰的对象不一定是最原始的那个对象了，也可能是被其它装饰器装饰过后的对象，反正都是实现的同一个接口，也就是同一类型。

ConcreteDecorator：

实际的装饰器对象，实现具体要向被装饰对象添加的功能。

体会装饰模式

- 复杂的奖金计算

奖金计算是相对复杂的功能，尤其是对于业务部门的奖金计算方式，是非常复杂的，除了业务功能复杂外，另外一个麻烦之处是计算方式还经常需要变动，因为业务部门经常通过调整奖金的计算方式来激励士气。

先从业务上看看现有的奖金计算方式的复杂性：

1：首先是奖金分类：对于个人，大致有个人当月业务奖金、个人累计奖金、个人业务增长奖金、及时回款奖金、限时成交加码奖金等等；

2：对于业务主管或者是业务经理，除了个人奖金外，还有：团队累计奖金、团队业务增长奖金、团队盈利奖金等等。

3：其次是计算奖金的金额，又有这么几个基数：销售额、销售毛利、实际回款、业务成本、奖金基数等等；

4：另外一个就是计算的公式，针对不同的人、不同的奖金类别、不同的计算奖金的金额，计算的公式是不同的，就算是同一个公式，里面计算的比例参数也有可能是不同的。

体会装饰模式

- 简化后的奖金计算体系

看了上面奖金计算的问题，所幸我们只是来学习设计模式，并不是真的要去实现整个奖金计算体系的业务，因此也没有必要把所有的计算业务都罗列在这里，为了后面演示的需要，简化一下，演示用的奖金计算体系如下：

1：每个人当月业务奖金 = 当月销售额 X 3%

2：每个人累计奖金 = 总的回款额 X 0.1%

3：团队奖金 = 团队总销售额 X 1%

体会装饰模式

- 不用模式的解决方案

- 有何问题

对于奖金计算，光是计算方式复杂，也就罢了，不过是实现起来会困难点，相对而言还是比较好解决的，不过是用程序把已有的算法表达出来。

最痛苦的是，这些奖金的计算方式，经常发生变动，几乎是每个季度都会有小调整，每年都有大调整，这就要求软件的实现要足够灵活，要能够很快进行相应调整和修改，否则就不能满足实际业务的需要。

举个简单的例子来说，现在根据业务需要，需要增加一个“环比增长奖金”，就是本月的销售额比上个月有增加，而且要达到一定的比例，当然增长比例越高，奖金比例越大。那么软件就必须要重新实现这么个功能，并正确的添加到系统中去。过了两个月，业务奖励的策略发生了变化，不再需要这个奖金了，或者是另外换了一个新的奖金方式了，那么软件就需要把这个功能从软件中去掉，然后再实现新的功能。

体会装饰模式

那么上面的要求该如何实现呢？

很明显，一种方案是通过继承来扩展功能；另外一种方案就是到计算奖金的对象里面，添加或者删除新的功能，并在计算奖金的时候，调用新的功能或是不调用某些去掉的功能，这种方案会严重违反开-闭原则。

还有一个问题，就是在运行期间，不同人员参与的奖金计算方式也是不同的，举例来说：如果是业务经理，除了参与个人计算部分外，还要参加团队奖金的计算，这就意味着需要在运行期间动态来组合需要计算的部分，也就是会有一堆的if-else。

体会装饰模式

总结一下，奖金计算面临如下问题：

（1）计算逻辑复杂

（2）要有足够灵活性，可以方便的增加或者减少功能

（3）要能动态的组合计算方式，不同的人参与的计算不同

上面描述的奖金计算的问题，绝对没有任何夸大成分，相反已经简化不少了，还有更多麻烦没有写上来，毕竟我们的重点在设计模式，而不是业务。

把上面的问题抽象一下，设若有一个计算奖金的对象，现在需要能够灵活的给它增加和减少功能，还需要能够动态的组合功能，每个功能就相当于在计算奖金的某个部分。

现在的问题就是：如何才能够透明的给一个对象增加功能，并实现功能的动态组合呢？

体会装饰模式

- 使用模式来解决的思路

虽然经过简化，业务简单了很多，但是需要解决的问题不会少，还是要解决：要透明的给一个对象增加功能，并实现功能的动态组合。

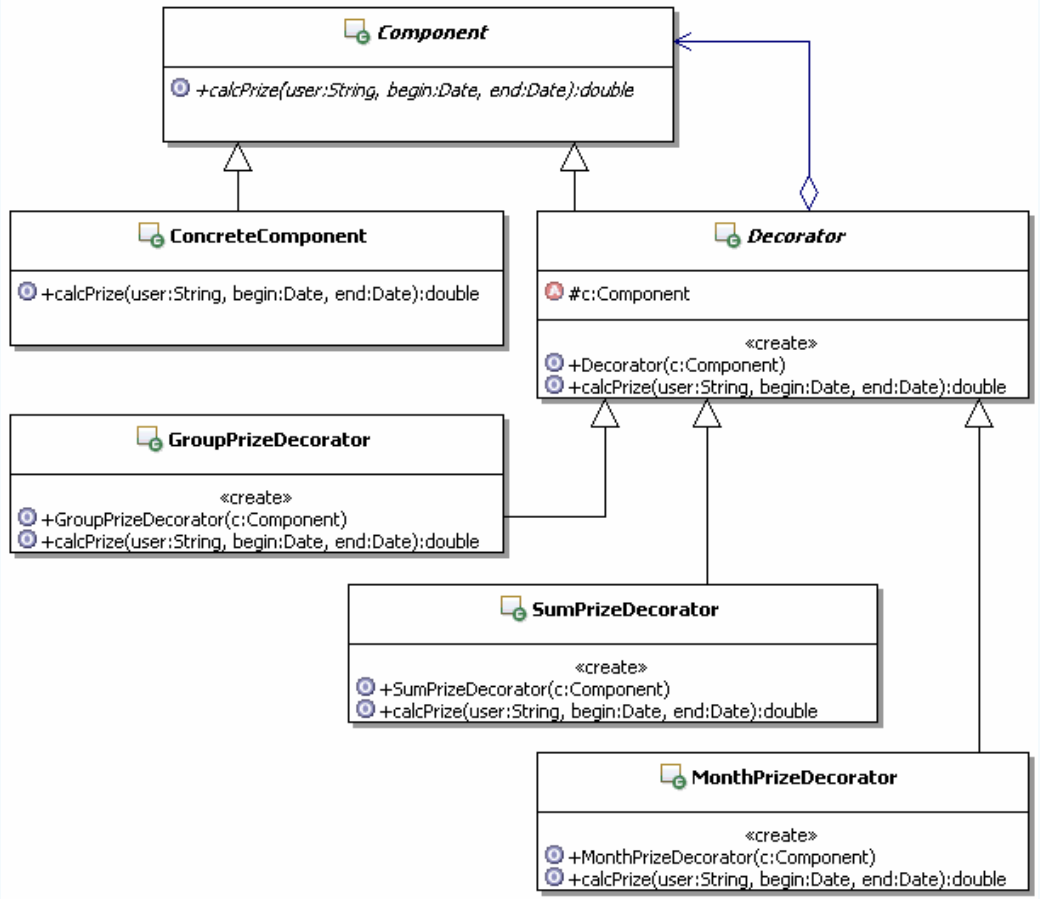
所谓透明的给一个对象增加功能，换句话说就是要给一个对象增加功能，但是不能让这个对象知道，也就是不能去改动这个对象。而实现了能够给一个对象透明的增加功能，自然就能够实现功能的动态组合。

在装饰模式的实现中，为了能够和原来使用被装饰对象的代码实现无缝结合，是通过定义一个抽象类，让这个类实现与被装饰对象相同的接口，然后在具体实现类里面，转调被装饰的对象，在转调的前后添加新的功能，这就实现了给被装饰对象增加功能，这个思路跟“对象组合”非常类似。

在转调的时候，如果觉得被装饰的对象的功能不再需要了，还可以直接替换掉，也就是不再转调，而是在装饰对象里面完全全新的实现。

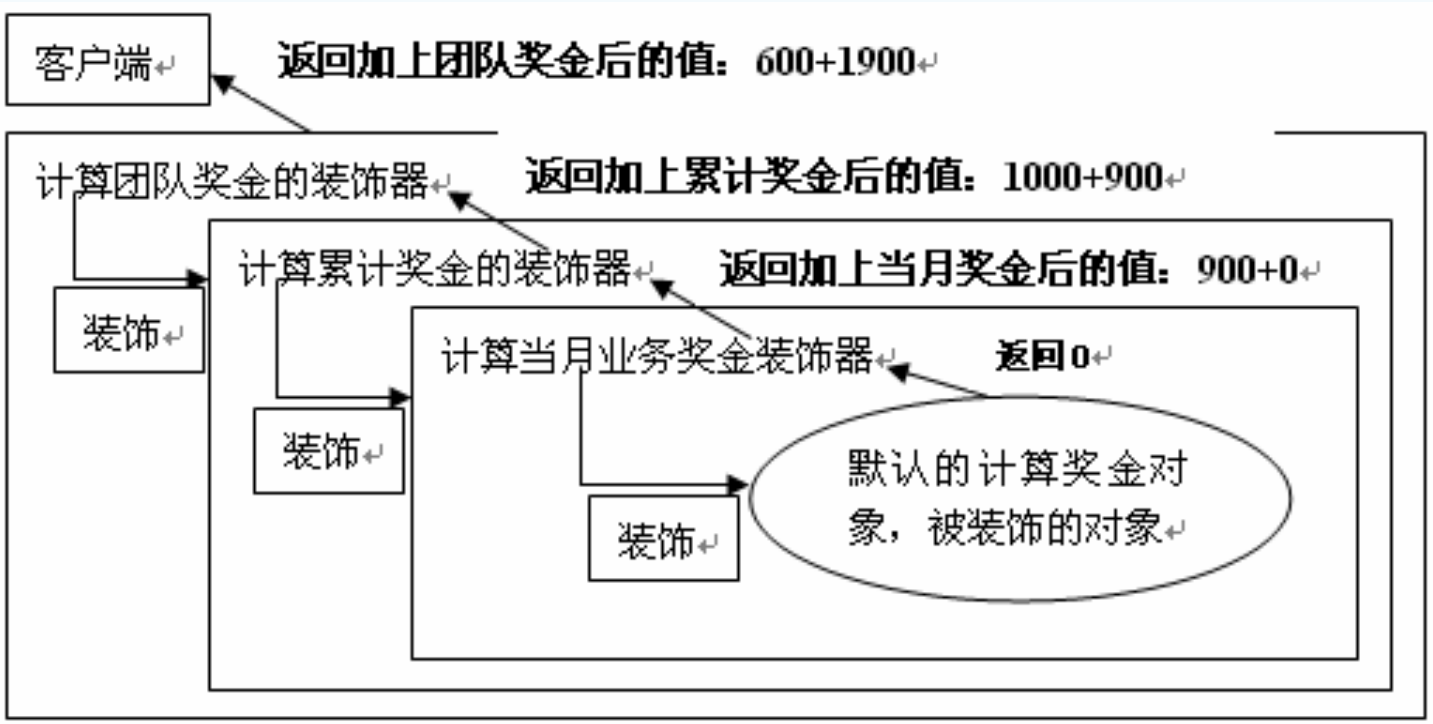
体会装饰模式

- 使用模式来解决的类图



体会装饰模式

- 画个图来说明奖金的计算过程



这个图很好的揭示了装饰模式的组合和调用过程，请仔细体会一下。

理解装饰模式

n 认识装饰模式

1：装饰模式的功能

装饰模式能够实现动态的为对象添加功能，是从一个对象外部来给对象增加功能，相当于是改变了对象的外观。当装饰过后，从外部使用系统的角度看，就不再是使用原始的那个对象了，而是使用被一系列的装饰器装饰过后的对象。

这样就能够灵活的改变一个对象的功能，只要动态组合的装饰器发生了改变，那么最终所得到的对象的功能也就发生了改变。

变相的还得到了另外一个好处，那就是装饰器功能的复用，可以给一个对象多次增加同一个装饰器，也可以用同一个装饰器装饰不同的对象。

2：对象组合

在面向对象设计中，有一条很基本的规则就是“尽量使用对象组合，而不是对象继承”来扩展和复用功能 ，装饰模式的思考起点就是这个规则 。

理解装饰模式

3：装饰器

装饰器实现了对被装饰对象的某些装饰功能，可以在装饰器里面调用被装饰对象的功能，获取相应的值，这其实是一种递归调用。

在装饰器里不仅仅是可以给被装饰对象增加功能，还可以根据需要选择是否调用被装饰对象的功能，如果不调用被装饰对象的功能，那就变成完全重新实现了，相当于动态修改了被装饰对象的功能。

另外一点，各个装饰器之间最好是完全独立的功能，不要有依赖，这样在进行装饰组合的时候，才没有先后顺序的限制，也就是先装饰谁和后装饰谁都应该是一样的，否则会大大降低装饰器组合的灵活性。

理解装饰模式

4：装饰器和组件类的关系

装饰器是用来装饰组件的，装饰器一定要实现和组件类一致的接口，保证它们是同一个类型，并具有同一个外观，这样组合完成的装饰才能够递归的调用下去。

组件类是不知道装饰器的存在的，装饰器给组件添加功能是一种透明的包装，组件类毫不知情。需要改变的是外部使用组件类的地方，现在需要使用包装后的类，接口是一样的，但是具体的实现类发生了改变。

5：退化形式

如果仅仅只是想要添加一个功能，就没有必要再设计装饰器的抽象类了，直接在装饰器里面实现跟组件一样的接口，然后实现相应的装饰功能就可以了。但是建议最好还是设计上装饰器的抽象类，这样有利于程序的扩展。

理解装饰模式

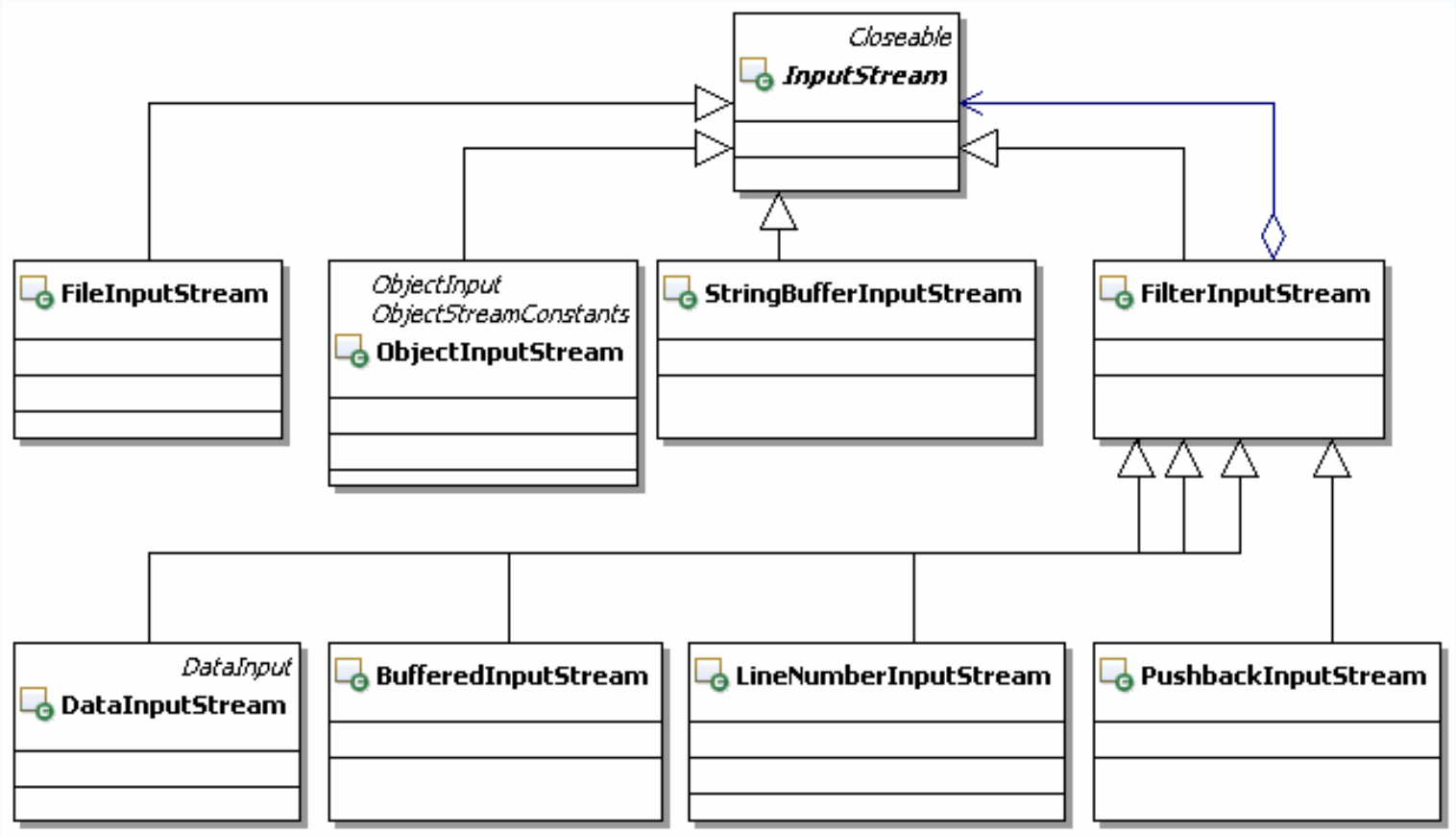
- Java中的装饰模式应用

  Java中典型的装饰模式应用——I/O流

```
//流式读取文件
DataInputStream din = null;
try{
    din = new DataInputStream(
        new BufferedInputStream(
            new FileInputStream("IOTest.txt")
        )
    );
    //然后就可以获取文件内容了
    byte bs []= new byte[din.available()];
    din.read(bs);
    String content = new String(bs);
    System.out.println("文件内容===="+content);
}finally{
    din.close();
}
```

理解装饰模式

Java的I/O对象层次图



理解装饰模式

查看上图会发现，它的结构和装饰模式的结构几乎是一样的：

1：InputStream就相当于装饰模式中的Component。

2：其实FileInputStream、ObjectInputStream、StringBufferInputStream这几个对象是直接继承了InputSream，还有几个直接继承InputStream的对象，比如：ByteArrayInputStream、PipedInputStream等。这些对象相当于装饰模式中的ConcreteComponent，是可以被装饰器装饰的对象。

3：那么FilterInputStream就相当于装饰模式中的Decorator，而它的子类DataInputStream、BufferedInputStream、LineNumberInputStream和PushbackInputStream就相当于装饰模式中的ConcreteDecorator了。另外FilterInputStream和它的子类对象的构造器，都是传入组件InputStream类型，这样就完全符合前面讲述的装饰器的结构了。

理解装饰模式

- 自己实现的I/O流的装饰器——第一版

来个功能简单点的，实现把英文加密存放吧，也谈不上什么加密算法，就是把英文字母向后移动两个位置，比如：a变成c，b变成d，以此类推，最后的y变成a，z就变成b，而且为了简单，只处理小写的，够简单的吧。

测试中可能会出现输出一片空白，要把这个问题搞清楚，就需要把上面I/O流的内部运行和基本实现搞明白，分开来看看具体的运行过程吧。

先看看成功输出流中的内容的写法的运行过程：

1：当执行到“dout.write(”abcdxyz“.getBytes());”这句话的时候，会调用DataOutputStream的write方法，把数据输出到BufferedOutputStream中；

2：由于BufferedOutputStream流是一个带缓存的流，它默认缓存8192byte，也就是默认流中的缓存数据到了8192byte，它才会自动输出缓存中的数据；

3：而目前要输出的字节肯定不到8192byte，因此数据就被缓存在BufferedOutputStream流中了，而不会被自动输出

理解装饰模式

4：当执行到“dout.close();”这句话的时候：会调用关闭DataOutputStream流，这会转调到传入DataOutputStream中的流的close方法，也就是BufferedOutputStream的close方法，而BufferedOutputStream的close方法继承自FilterOutputStream，在FilterOutputStream的close方法实现里面，会先调用输出流的方法flush，然后关闭流。也就是此时BufferedOutputStream流中缓存的数据会被强制输出；

5：BufferedOutputStream流中缓存的数据被强制输出到EncryptOutputStream流，也就我们自己实现的流，没有缓存，经过处理后继续输出；

6：EncryptOutputStream流会把数据输出到FileOutputStream中，FileOutputStream会直接把数据输出到文件中，因此，这种实现方式会输出文件的内容。

理解装饰模式

再来看看不能输出流中的内容的写法的运行过程：

1：当执行到“dout.write(”abcdxyz“.getBytes());”这句话的时候，会调用DataOutputStream的write方法，把数据输出到EncryptOutputStream中；

2：EncryptOutputStream流，也就是我们自己实现的流，没有缓存，经过处理后继续输出，把数据输出到BufferedOutputStream中；

3：由于BufferedOutputStream流是一个带缓存的流，它默认缓存8192byte，也就是默认流中的缓存数据到了8192byte，它才会自动输出缓存中的数据；

4：而目前要输出的字节肯定不到8192byte，因此数据就被缓存在BufferedOutputStream流中了，而不会被自动输出

5：当执行到“dout.close();”这句话的时候：会调用关闭DataOutputStream流，这会转调到传入DataOutputStream流中的流的close方法，也就是EncryptOutputStream的close方法，而EncryptOutputStream的close方法继承自OutputStream，在OutputStream的close方法实现里面，是个空方法，什么都没有做。因此，这种实现方式没有flush流的数据，也就不会输出文件的内容，自然是一片空白了。

理解装饰模式

- 自己实现的I/O流的装饰器——第二版

要让我们写的装饰器跟其它Java中的装饰器一样用，最合理的方案就应该是：让我们的装饰器继承装饰器的父类，也就是FilterOutputStream类，然后使用父类提供的功能来协助完成想要装饰的功能。

理解装饰模式

- 装饰模式和AOP

  关于AOP的知识，可以参看私塾在线上关于AOP的课程，或者是Spring的课程。

  可以使用装饰模式做出类似AOP的效果

理解装饰模式

- 装饰模式的优缺点
  1：比继承更灵活
  2：更容易复用功能
  3：简化高层定义
  4：会产生很多细粒度对象

思考装饰模式

- 装饰模式的本质

  装饰模式的本质是：**动态组合**

- 何时选用装饰模式

1：如果需要在不影响其它对象的情况下，以动态、透明的方式给对象添加职责，可以使用装饰模式，这几乎就是装饰模式的主要功能

2：如果不合适使用子类来进行扩展的时候，可以考虑使用装饰模式，因为装饰模式是使用的“对象组合”的方式。所谓不适合用子类扩展的方式，比如：扩展功能需要的子类太多，造成子类数目呈爆炸性增长。