|12.1Decorator模式

【装饰边框与被装饰物的一致性】

假如现在有一块蛋糕，如果只涂上奶油，其他什么都不加，就是奶油蛋糕。如果加上草莓，就是草莓奶油蛋糕。如果再加上一块黑色巧克力板，上面用白色巧克力写上姓名，然后插上代表年龄的蜡烛，就变成了一块生日蛋糕。

不论是蛋糕、奶油蛋糕、草莓蛋糕还是生日蛋糕，它们的核心都是蛋糕。不过，经过涂上奶油，加上草莓等装饰后，蛋糕的味道变得更加甜美了，目的也变得更加明确了。

程序中的对象与蛋糕十分相似。首先有一个相当于蛋糕的对象，然后像不断地装饰蛋糕一样地不断地对其增加功能，它就变成了使用目的更加明确的对象。

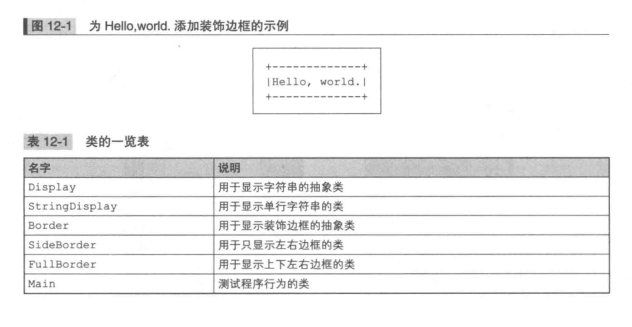
像这样不断地为对象添加装饰的设计模式被称为Decorator模式。Decorator指的是“装饰物”。

本章中，我们将学习Decorator模式的相关知识。

|12.2示例程序

本章中的示例程序的功能是给文字添加装饰边框。这里所谓的装饰边框是指用“-”“+”“1”

等字符组成的边框。图12-1是一个输出结果示例。



|12.3Decorator模式中的登场角色

在Decorator模式中有以下登场角色。

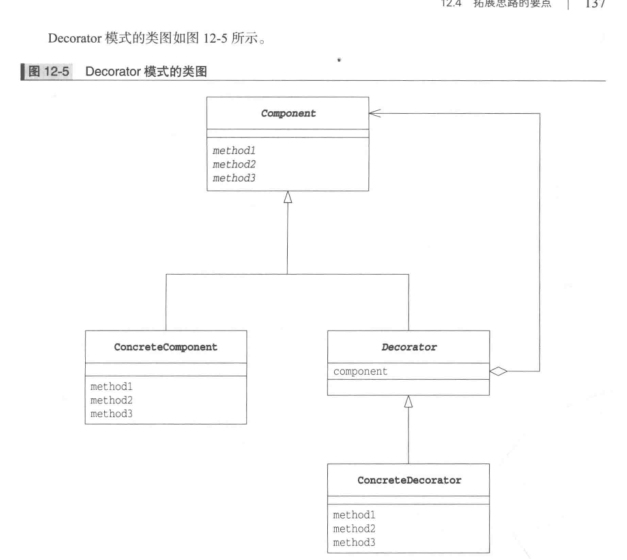
·Component增加功能时的核心角色。以本章开头的例子来说，装饰前的蛋糕就是Component角色。

Component角色只是定义了蛋糕的接口（API）。在示例程序中，由Display类扮演此角色。

◆ConcreteComponent该角色是实现了Component角色所定义的接口（APl）的具体蛋糕。在示例程序中，由stringDisplay类扮演此角色。

◆Decorator（装饰物）该角色具有与Component角色相同的接口（APl）。在它内部保存了被装饰对象—Component角色。Decorator角色知道自己要装饰的对象。在示例程序中，由Border类扮演此角色。

·ConcreteDecorator（具体的装饰物）该角色是具体的Decorator角色。在示例程序中，由sideBorder类和Fu11Border类扮演此角色。



|12.4拓展思路的要点

接口（API）的透明性

在Decorator模式中，装饰边框与被装饰物具有一致性。具体而言，在示例程序中，表示装饰边框的Border类是表示被装饰物的Display类的子类，这就体现了它们之间的一致性。也就是说，Border类（以及它的子类）与表示被装饰物的Display类具有相同的接口（API）。

这样，即使被装饰物被边框装饰起来了，接口（APl）也不会被隐藏起来。其他类依然可以调用getcolumns、getRows、getRowText以及show方法。这就是接口（APl）的“透明性”。

在示例程序中，实例b4被装饰了多次，但是接口（API）却没有发生任何变化。

得益于接口（API）的透明性，Decorator模式中也形成了类似于Composite模式中的递归结构。

也就是说，装饰边框里面的“被装饰物”实际上又是别的物体的“装饰边框”。就像是剥洋葱时以为洋葱心要出来了，结果却发现还是皮。不过，Decorator模式虽然与Composite模式一样，都具有递归结构，但是它们的使用目的不同。Decorator模式的主要目的是通过添加装饰物来增加对象的功能。

在不改变被装饰物的前提下增加功能

在Decorator模式中，装饰边框与被装饰物具有相同的接口（API）。虽然接口（API）是相同的，但是越装饰，功能则越多。例如，用SideBorder装饰Display后，就可以在字符串的左右两侧加上装饰字符。如果再用Fu11Border装饰，那么就可以在字符串的四周加上边框。此时，我们完全不需要对被装饰的类做任何修改。这样，我们就实现了不修改被装饰的类即可增加功能。

Decorator模式使用了委托。对“装饰边框”提出的要求（调用装饰边框的方法）会被转交（委托）给“被装饰物”去处理。以示例程序来说，就是sideBorder类的getcolumns方法调用了display.getcolumns（）。除此以外，getRows方法也调用了display.getRows（）。

可以动态地增加功能

Decorator模式中用到了委托，它使类之间形成了弱关联关系。因此，不用改变框架代码，就可以生成一个与其他对象具有不同关系的新对象。

只需要一些装饰物即可添加许多功能

使用Decorator模式可以为程序添加许多功能。只要准备一些装饰边框（ConcreteDecorator角色），即使这些装饰边框都只具有非常简单的功能，也可以将它们自由组合成为新的对象。这就像我们可以自由选择香草味冰激凌、巧克力冰激凌、草莓冰激凌、猕猴桃冰激凌等各种口味的冰激凌一样。如果冰激凌店要为顾客准备所有的冰激凌成品那真是太麻烦了。因此，冰激凌店只会准备各种香料，当顾客下单后只需要在冰激凌上加上各种香料就可以了。不管是香草味，还是咖啡朗姆和开心果的混合口味，亦或是香草味、草莓味和猕猴桃三重口味，顾客想吃什么口味都可以。Decorator模式就是可以应对这种多功能对象的需求的一种模式。

Java.io 包与Decorator模式

下面我们来谈谈java.io包中的类。java.io包是用于输入输出（Input/Output，简称/o）的包。这里，我们使用了Decorator模式。首先，我们可以像下面这样生成一个读取文件的实例。

Reader reader=new FileReader（"datafile.txt"）；然后，我们也可以像下面这样在读取文件时将文件内容放入缓冲区。

Reader reader=new BufferedReader（

new FileReader（"datafile.txt"）；

）；这样，在生成BufferedReader类的实例时，会指定将文件读取到FileReader类的实例中。

再然后，我们也可以像下面这样管理行号。

Reader reader=new LinelumberReadex（

New BufferedReader（

New FileReader（"datafile.txt"）；

)

)

无论是LineNumberReader类的构造函数还是BufferedReader类的构造函数，都可以接收Reader类（的子类）的实例作为参数，因此我们可以像上面那样自由地进行各种组合。

我们还可以只管理行号，但不进行缓存处理。

Reader reader=new LineNumberReadex（

new FileReader（"datafile.txt\*）；

）；接下来，我们还会管理行号，进行缓存，但是我们不从文件中读取数据，而是从网络中读取数据（下面的代码中省略了细节部分和异常处理）。

java.net.Socket socket=new Socket（hostname，portnumber）；Reader reader=new LineNumberReader（

new BufferedReadex（

new InputStreamReader（

socket.getInputStream（）

）；这里使用的InputStreamReader类既接收 getInputStream方法返回的InputStream类的实例作为构造函数的参数，也提供了Reader类的接口（APl）（这属于第2章学习过的Adapter模式）。

除了java.io包以外，我们还在avax.swing.border包中使用了Decorator模式。

javax.swing.border包为我们提供了可以为界面中的控件添加装饰边框的类。

导致增加许多很小的类

Decorator模式的一个缺点是会导致程序中增加许多功能类似的很小的类。

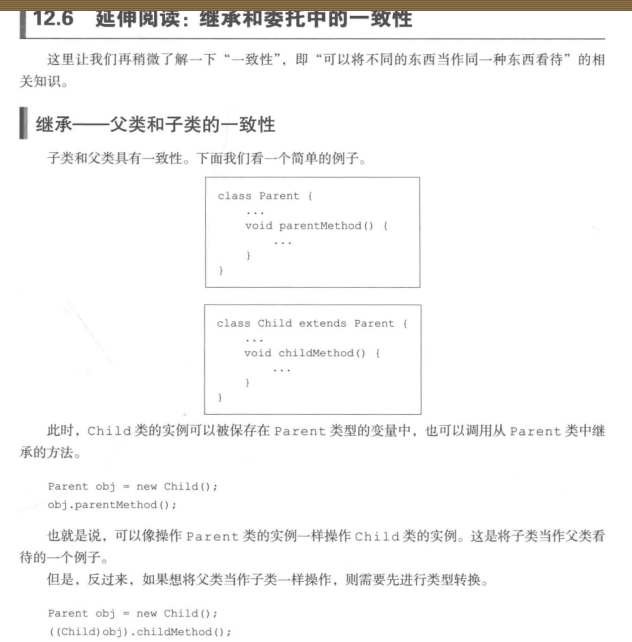
|12.5相关的设计模式

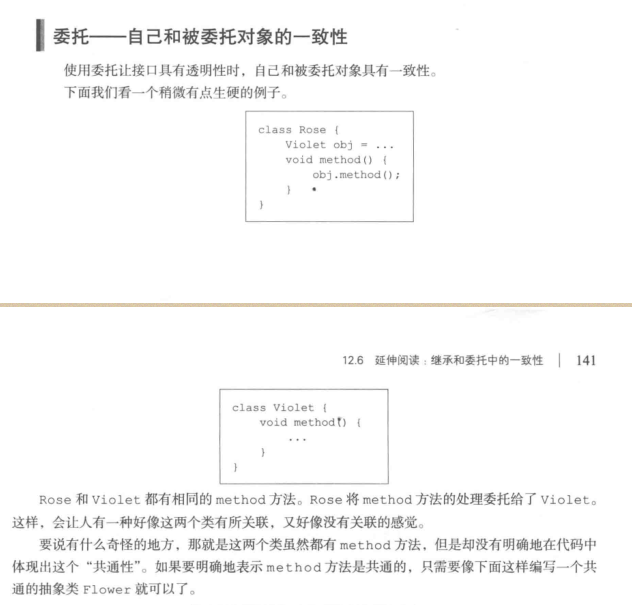
◆Adapter模式（第2章）Decorator模式可以在不改变被装饰物的接口（API）的前提下，为被装饰物添加边框（透明性）。

Adapter模式用于适配两个不同的接口（APl）。

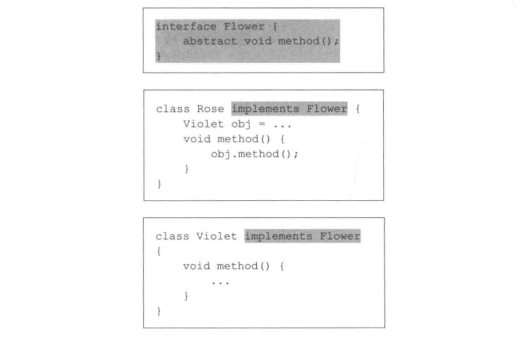
◆Stragety模式（第10章）Decorator模式可以像改变被装饰物的边框或是为被装饰物添加多重边框那样，来增加类的功能。

Stragety模式通过整体地替换算法来改变类的功能。









至此，大家可能会产生这样的疑问，即Rose类中的ob字段被指定为具体类型Violet真的好吗？如果指定为抽象类型Flower会不会更好呢？……究竟应该怎么做才好呢？其实没有固定答案，需求不同，做法也不同。