设计模式-装饰器模式

参考

-

装饰器允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式数据结构型模式，它是作为现有的类的一个包装。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能

使用场景

- 扩展一个类的功能
- 动态增加功能

Demo分析

> 分析：
>
> 当我们完成了，各种图形的绘制类之后我们需要考虑设置图形的各种颜色。此时，我们需要在不影响图形绘制方法的时候去增加图形颜色设置绘制的一些方法装饰器模式此时就可以起到作用。

Go实现

```
package main

import "fmt"

/**
* 抽象父类
 */
type Shape interface {
```

```
    Draw()
}

/**
* 共同的方法，属性
 */
type shapeCommFunc struct {
    name string
}

func (c *shapeCommFunc) Draw() {
    fmt.Printf("Draw:  %s\n", c.name)
}

type Circle struct {
    x, y, radius int
    *shapeCommFunc
}

type Rectangle struct {
    x, y, width, height int
    *shapeCommFunc
}

/**
 * 装饰器
 */
type Decoder struct {
    shape Shape
    *ColorDecoder
}

/**
 * 颜色装饰器
 */
type ColorDecoder struct {
    color string
}

func (c *ColorDecoder) setBorderColor() {
    fmt.Printf("设置Border:%s\n", c.color)
}

func (c *Decoder) Draw() {
    c.setBorderColor()
    c.shape.Draw()
}

func test() {
    redCircle := &Decoder{&Circle{x: 1, y: 2, radius: 3, shapeCommFunc:
&shapeCommFunc{"Circle"}}, &ColorDecoder{"red"}}
    redCircle.Draw()
    redRectangle := &Decoder{&Rectangle{x: 1, y: 2, width: 3, height: 4,
shapeCommFunc: &shapeCommFunc{"Rectangle"}}, &ColorDecoder{"blue"}}
    redRectangle.Draw()
```

```
}

func main() {
    test()
}
```

输出

```
Draw:  Circle
设置Border:blue
Draw:  Rectangle
```

Python实现

```
class Shape(object):
  """
    ## 形状抽象类
  """
  def __init__(self, shapeName):
    self.shapeName = shapeName
  def draw(self,):
    print(f"Draw {self.shapeName}")

class Circle(Shape):
  def __init__(self, shapeName, radius):
    super().__init__(shapeName)
    self.radius = radius

class Rectangle(Shape):
  def __init__(self, shapeName, width, height):
    super().__init__(shapeName)
    self.width = width
    self.height = height

class ShapeDecorator(Shape):
  """
    ## 装饰器抽象类
  """
  def __init__(self, shape):
    self.shape = shape
  def draw(self,):
    self.shape.draw()

class ColorDecorator(ShapeDecorator):
  """
    ## 颜色装饰器
  """
  def __init__(self, shape, borderColor="red"):
    super().__init__(shape)
    self.borderColor = borderColor
  def draw(self):
    self.setRedBorder()
    self.shape.draw()
```

```
  def setRedBorder(self):
    print(f"Border Color: {self.borderColor}")

def test():
  c = Circle("Circle", 10)
  redCircle = ColorDecorator(c, "red")
  redCircle.draw()
  r = Rectangle("Rectangle", 10, 10)
  blueRectangle = ColorDecorator(r, "blue")
  blueRectangle.draw()

if __name__ == "__main__":
  test()
```

输出

```
Border Color: red
Draw Circle
Border Color: blue
Draw Rectangle
```

总结

> 装饰器可以帮助我们修改已经封装好的部分代码，基于原有的功能下装饰器可以给原对象添加功能。实现了对原对象的功能扩展。python中自带的装饰器也可以很好的达到这个效果。