定义:动态（组合）地给一个对象增加一些额外的职责。就增加功能而言，Decorator模式比生成子类（继承）更为灵活（消除重复代码 & 减少子类个数）。——《设计模式》GoF

在某些情况下我们可能会“过度地使用继承来扩展对象的功能”，由于继承为类型引入的静态特质，使得这种扩展方式缺乏灵活性；并且随着子类的增多（扩展功能的增多），各种子类的组合（扩展功能的组合）会导致更多子类的膨胀。

如何使“对象功能的扩展”能够根据需要来动态地实现？同时避免“扩展功能的增多”带来的子类膨胀问题？从而使得任何“功能扩展变化”所导致的影响将为最低？

装饰器模式（Decorator Pattern）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属于结构型模式，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。

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

主要解决：一般的，为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。

何时使用：在不想增加很多子类的情况下扩展类。

如何解决：将具体功能职责划分，同时继承装饰者模式。

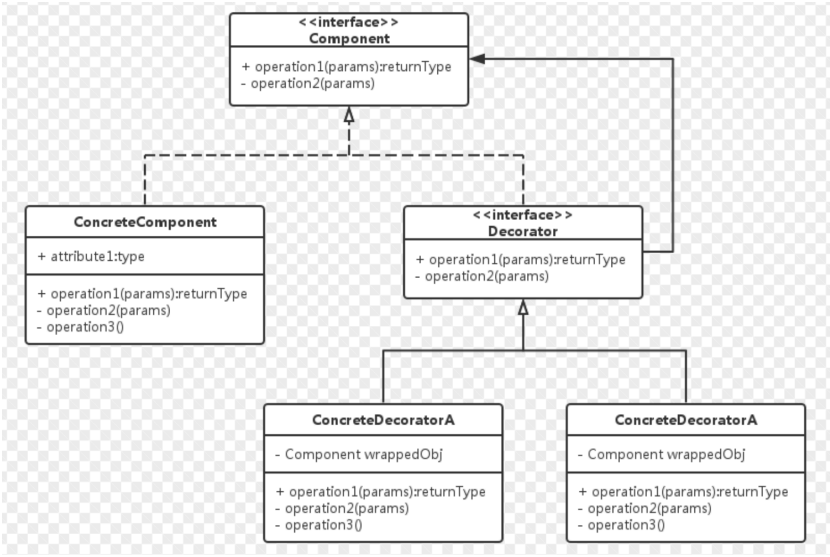
关键代码： 1、Component 类充当抽象角色，不应该具体实现。 2、修饰类引用和继承 Component 类，具体扩展类重写父类方法。

优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

缺点：多层装饰比较复杂。

使用场景： 1、扩展一个类的功能。 2、动态增加功能，动态撤销。

注意事项：可代替继承。



装饰器模式的重要组成部分

①装饰器模式特点:

- 装饰对象和真实对象有相同的接口。这样客户端对象就能以和真实对象相同的方式和装饰对象交互。
- 装饰对象包含一个真实对象的引用（reference）
- 装饰对象接受所有来自客户端的请求。它把这些请求转发给真实的对象。
- 装饰对象可以在转发这些请求以前或以后增加一些附加功能。这样就确保了在运行时，不用修改给定对象的结构就可以在外部增加附加的功能。在面向对象的设计中，通常是通过继承来实现对给定类的功能扩展。

②装饰器模式由组件和装饰者组成:

- 抽象组件（Component）：需要装饰的抽象对象。
- 具体组件（ConcreteComponent）：是我们需要装饰的对象。
- 抽象装饰类（Decorator）：内含指向抽象组件的引用及装饰者共有的方法。
- 具体装饰类（ConcreteDecorator）：被装饰的对象。

```
#include <iostream>
#include <string>
using namespace std;

class ICake {  //蛋糕基类
public:
    virtual void ShowCake() = 0;
    virtual ~ICake() {}
public:
    string m_strName;
    int m_nCost;
    int m_nTotalCost;
};
class OriginalCake : public ICake { //定义一个原味蛋糕,用装饰器给这个蛋糕加各种料
public:
    OriginalCake(){
        m_strName = "原味蛋糕";
        m_nCost = 10;
    }
    void ShowCake(){
        m_nTotalCost += m_nCost;
        cout << m_strName << ": " << m_nCost << "元" << endl;
    }
};
class DecoratorCake : public ICake { //装饰器基类
public:
    virtual void ShowCake() = 0; //装饰类和被装饰的类有相同接口
    virtual ~DecoratorCake(){};
protected:
    ICake * m_pCake;  //装饰类包含被装饰类的一个指针
};
class DecoratorCakeMilk : public DecoratorCake{ //给原味蛋糕加奶油
public:
    DecoratorCakeMilk(ICake *pCake){
        m_pCake = pCake;
    }
    void ShowCake(){
        this->m_strName = m_pCake->m_strName + ",加奶油";
        this->m_nCost = m_pCake->m_nCost + 2;
        m_nTotalCost += this->m_nCost;
        cout << this->m_strName << ",2元,共需要 " << m_nTotalCost <<  "元" << endl;
    }
};
class DecoratorCakePeanut : public DecoratorCake{ //给原味蛋糕加花生粒
public:
    DecoratorCakePeanut(ICake *pCake){
        m_pCake = pCake;
    }
    void ShowCake(){
```

```
        this->m_strName = m_pCake->m_strName + ",加花生粒";
        this->m_nCost = m_pCake->m_nCost + 3;
        m_nTotalCost += m_nCost;
        cout << this->m_strName << ",3元,共需要 " << m_nTotalCost <<  "元" << endl;
    }
};
int main()
{
    OriginalCake *pCake = new OriginalCake();
    pCake->ShowCake();
    DecoratorCakeMilk *pMilk = new DecoratorCakeMilk(pCake);
    pMilk->ShowCake();
    DecoratorCakePeanut *pPeanut = new DecoratorCakePeanut(pCake);
    pPeanut->ShowCake();
    delete pCake;
    delete pMilk;
    delete pPeanut;
    return 0;
}
```

```
192:DesignPattnsStudy weishichun$ ls Decorator
Decorator_1.cpp          Decorator装饰器模式.pdf
192:DesignPattnsStudy weishichun$ g++ -o Decorator.out Decorator_1.cpp
192:DesignPattnsStudy weishichun$ ./Decorator.out
原味蛋糕：10元
原味蛋糕,加奶油,2元,共需要 12元
原味蛋糕,加花生粒,3元,共需要 13元
192:DesignPattnsStudy weishichun$
```