结构型模式描述如何将类或者对象结合在一起形成更大的结构。

结构型模式分为；

- 类结构型模式
  关心类的组合，由多个类可以组合成一个更大的系统。在类结构型模式中一般只存在继承关系和实现关系。
- 对象结构型模式
  关心类与对象的组合，通过关联关系使得在一个类中定义另一个类的实例对象，然后通过该对象调用其方法。根据“合成复用原则”，在系统中尽量使用关联关系来代替继承关系，因此大部分结构型模式都是对象结构型模式。

包含模式：

- 适配器模式
- 桥接模式
- 组合模式
- 装饰模式
- 外观模式
- 享元模式
- 代理模式

1 适配器模式

2 桥接模式

3 装饰模式

装饰模式：动态地给一个对象增加一些额外地职责，就增加对象地功能来说，装饰模式比生成子类实现更为灵活。

包含的角色：

- 抽象构件：定义了对象的接口
- 具体构件：定义了具体的构建对象，实现了在抽象构件中声明的方法，装饰器可以额外添加职责(方法)。
- 抽象装饰类：是抽象构建的子类，用于给具体构建增加职责，但具体职责在其子类中实现。
- 具体装饰类：抽象修饰类的子类，负责向构建添加新的职责。

代码：

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

class Shape
{
public:
    virtual void draw() = 0;
    virtual ~Shape(){};
};

class Circle : public Shape
{
public:
    virtual void draw() {
        cout << "Circle" << endl;
    };
    virtual ~Circle(){};
};

class Rectangle : public Shape
{
public:
    virtual void draw() {
        cout << "Rectangle" << endl;
    };
    virtual ~Rectangle(){};
};

class Decotator : public Shape
{
protected:
    Shape * D;
public:
    Decotator(){};
    Decotator(Shape *C_Shape) { this->D = C_Shape; }
    virtual void draw() { D->draw(); };
    virtual ~Decotator(){};
};

class RedDecotator: public Decotator
{
public:
    RedDecotator(){};
    RedDecotator(Shape *pcmp) : Decotator(pcmp){}
    virtual ~RedDecotator(){};
    virtual void draw() {
        Decotator::draw();
        setRedBorder();

    }
```

```
private:
    void setRedBorder()
    {
        cout << "Red" << endl;
    }
};

int main()
{
    Shape * circle = new Circle();
    Decotator redCircle = new RedDecotator(new Circle());
    Decotator redRectangle = new RedDecotator(new Rectangle());
    redCircle.draw();
    redRectangle.draw();
    circle->draw();
}
```

3.1 模式分析：
与继承关系相比，关联关系的主要优势在于不会破坏类的封装性，而且继承是耦合程度比较大的静态关系，无法在程序运行时动态扩展。在软件开发阶段，关联关系虽然不会比继承关系减少编码量，但是更容易维护。缺点就是比继承关系要创建更多的对象。

3.2 优点

- 比继承有更多的灵活性，可以通过动态的方式来扩展一个对象的功能，并通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同的行为组合
- 具体构建类和具体修饰类可以独立变化，用户可以根据需要添加新的构建类和具体装饰类。

3.3 缺点

- 会生成很多小对象
- 装饰类比继承更容易出错，排错也很困难

外观模式

代理模式