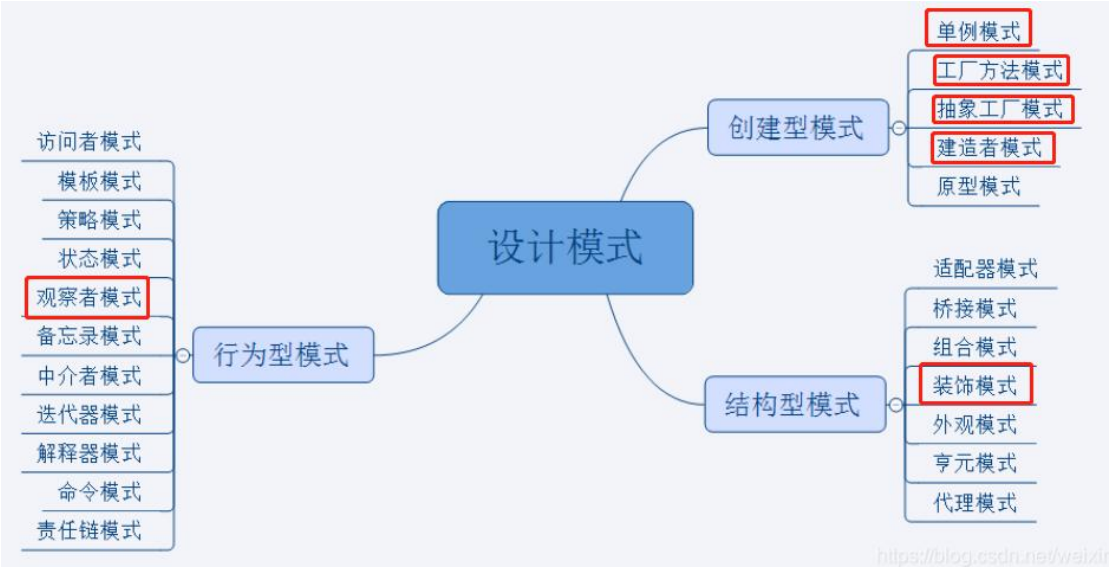
设计模式

MRL Liu

2022 年 02 月 02 日

设计模式是一套被**反复使用**、**多数人知晓**的、经过分类编目的**代码设计经验**的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性、程序的重用性。**设计模式**一共有 23 种，创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。如下



一、设计模式的六大原则

1、开放封闭原则

思想：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化。

描述：一个软件产品在生命周期内，都会发生变化，既然变化是一个既定的事实，我们就应该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。

优点：单一原则告诉我们，每个类都有自己负责的职责，里氏替换原则不能破坏继承关系的体系。

2、里氏代换原则

思想：使用的基类可以在任何地方使用继承的子类，完美的替换基类。

描述：子类可以扩展父类的功能，但不能改变父类原有的功能。子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法，子类中可以增加自己特有的方法。

优点：增加程序的健壮性，即使增加了子类，原有的子类还可以继续运行，互不影响。

3、依赖倒转原则

思想：面向接口编程，依赖于抽象而不依赖于具体

描述：它要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，这个是开放封闭原则的基础。

4、接口隔离原则

描述：使用多个隔离的接口，比使用单个接口要好。

例如：支付类的接口和订单类的接口，需要把这俩个类别的接口变成俩个隔离的接口

优点：降低依赖，降低耦合。

5、迪米特法则（最少知道原则）

思想：一个对象应当对其他对象有尽可能少地了解，简称类间解耦

描述：一个类尽量减少自己对其他对象的依赖，原则是低耦合，高内聚，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。

优点：低耦合，高内聚。

6、单一职责原则

思想：一个方法只负责一件事情。

描述：单一职责原则很简单，一个方法 一个类只负责一个职责，各个职责的程序改动，不影响其它程序。 这是常识，几乎所有程序员都会遵循这个原则。

优点：降低类和类的耦合，提高可读性，增加可维护性和可拓展性，降低可变性的风险。

二、面试常见的设计模式

1、创建型模式

（1）单例模式

单例模式（Singleton）可能是最简单、最常见也最有效的一种设计模式。**单例模式用于创建整个程序全局唯一的一个对象**，例如整个 Windows 系统只能打开一个任务管理器。

单例模式的实现借助了类的 static 成员变量，其实现可以简单分为如下三点：

（1）将构造函数私有化，外部无法调用

（2）在类中添加一个 static 的类指针，用来指向构造的唯一一个对象。

（3）提供一个公有的创建对象方法，外部只能通过该方法创建其对象。

代码如下：

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

//单例类
class Singleton{
public:
    //设置一个公有的静态成员函数，用来作为外界唯一的创建接口
    static Singleton* Instance(){
        if(_instance==0){
            _instance=new Singleton();
        }
        return _instance;
    };
protected:
    //构造函数被保护，不能被外界访问
    Singleton(){
        cout<<"构造函数执行成功"<<endl;
    };
private:
    //设置一个私有的静态类指针，用来保存全局唯一的实例对象
    static Singleton* _instance;//静态成员变量
};
// 全局变量类外声明
Singleton* Singleton::_instance=NULL;

// 主程序函数
int main(){
    // 创建了两次对象，但是构造函数只执行了一次，说明整个程序只有一个对象，两个指针都指向该对象
    Singleton* sgn1 =Singleton::Instance();
    Singleton* sgn2 =Singleton::Instance();
    system("pause");
    return 0;
}
```

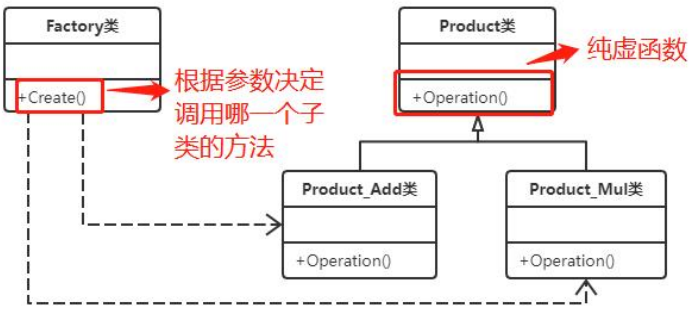
（2）工厂模式

工厂模式（Factory）简单来讲就是让创建对象有了统一的接口，具体化类的工作延迟到了子类中，其实现用到了 C++的继承和纯虚函数的知识。

工厂模式严格来讲有三种类型：简单工厂模式、工厂模式和抽象工厂模式。

（1）简单工厂模式

我们将被创建对象类命名为 Product，创建对象的类命名为 Factory。Factory 调用 Product 的统一接口 operation()，其具体实现由各自的子类决定。

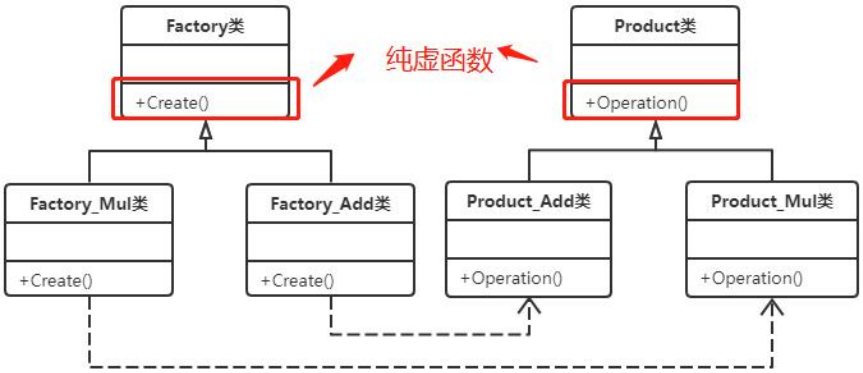


```cpp
//产品基类
class Product
{
public:
    virtual int operation(int a, int b) = 0;//
protected:
    Product(){};
};

//产品的子类Add
class Product_Add : public Product{
public:
    int operation(int a, int b){
        return a + b;
    }
};
//产品的子类Mul
class Product_Mul : public Product{
public:
    int operation(int a, int b){
        return a * b;
    }
};
```

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

//工厂类
class Factory{
public:
    Product* Create(int i){
        switch (i){
        case 1:
            return new Product_Add;
            break;
        case 2:
            return new Product_Mul;
            break;
        default:
            break;
        }
    }
};

int main(){
    Factory* factory = new Factory();
    int add_result = factory->Create(1)->operation(1, 2);
    int mul_result = factory->Create(2)->operation(1, 2);
    cout <<"op_add: " <<add_result << endl;
    cout <<"op_multiply: " << mul_result << endl;
    system("pause");
    return 0;
}
```

（2）工厂模式

上述的简单工厂模式中还存在不足就是每次增加一个新的 Product 子类，都需要修改 Factory 的内部代码，这违背了开放-封闭原则（对扩展开放，对修改封闭）。工厂模式抽象出了 Factory 基类，不同的 Factory 子类负责创建一个对应的产品，所以可以修改如下。

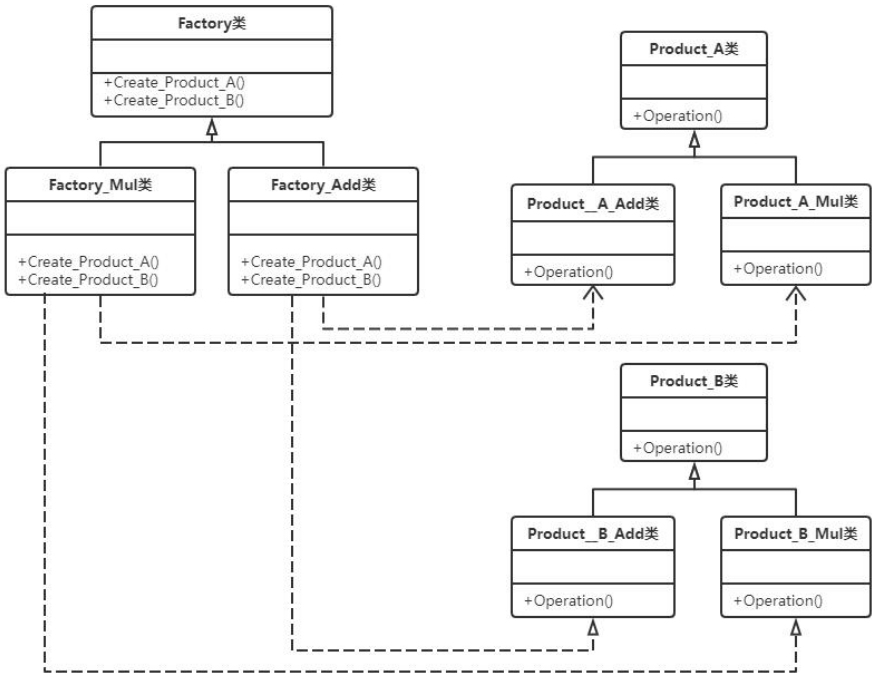



```cpp
//产品基类
class Product
{
public:
    virtual int operation(int a, int b) = 0;//
protected:
    Product(){};
};

//产品的子类Add
class Product_Add : public Product{
public:
    int operation(int a, int b){
        return a + b;
    }
};
//产品的子类Mul
class Product_Mul : public Product{
public:
    int operation(int a, int b){
        return a * b;
    }
};
```

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

//工厂类
class Factory{
public:
    virtual Product* Create()=0;
};

//工厂子类
class Factory_Add:public Factory{
public:
    Product* Create(){
        return new Product_Add;
    }
};

//工厂子类
class Factory_Mul:public Factory{
public:
    Product* Create(){
        return new Product_Mul;
    }

};

int main(){
    Factory_Add* factory_add = new Factory_Add();
    Factory_Mul* factory_mul = new Factory_Mul();
    int add_result = factory_add->Create()->operation(1, 2);
    int mul_result = factory_mul->Create()->operation(1, 2);
    cout <<"op_add: " <<add_result << endl;
    cout <<"op_multiply: " << mul_result << endl;
    system("pause");
    return 0;
}
```

（3）抽象工厂模式

工厂模式中，一个工厂只生产一种产品，如果想要一个工厂生产多个产品，就需要抽象工厂模式，抽象工厂模式提供了创建一系列相互依赖对象的接口。抽象工厂模式本质上是简单工厂模式和工厂模式的组合。



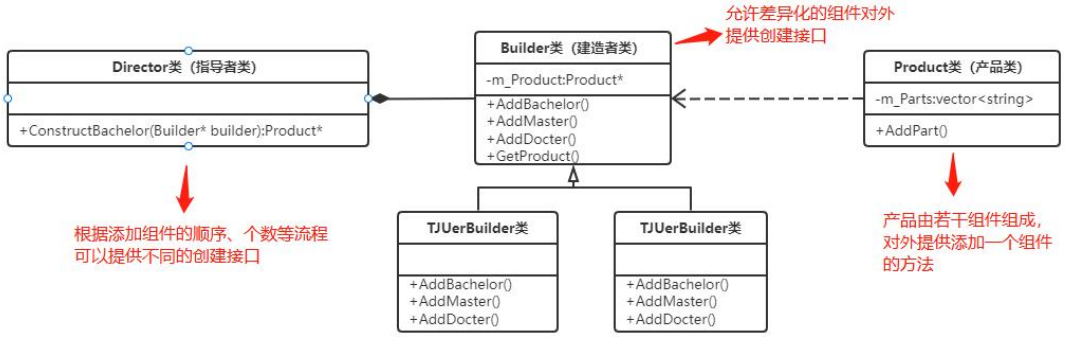

```cpp
//产品基类
class Product_A
{
public:
    virtual int operation(int a, int b) = 0;//
protected:
    Product_A(){};
};

//产品的子类Add
class Product_A_Add : public Product_A{
public:
    int operation(int a, int b){
        return a + b;
    }
};
//产品的子类Mul
class Product_A_Mul : public Product_A{
public:
    int operation(int a, int b){
        return a * b;
    }
};

//产品基类
class Product_B
{
public:
    virtual int operation(int a, int b) = 0;//
protected:
    Product_B(){};
};

//产品的子类Add
class Product_B_Add : public Product_B{
public:
    int operation(int a, int b){
        return (a + b)*2;
    }
};
//产品的子类Mul
class Product_B_Mul : public Product_B{
public:
    int operation(int a, int b){
        return a * b*2;
    }
};
```

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

//工厂类
class Factory{
public:
    virtual Product_A* Create_Product_A()=0;
    virtual Product_B* Create_Product_B()=0;
};

//工厂类
class Factory_Add:public Factory{
public:
    Product_A* Create_Product_A(){
        return new Product_A_Add;
    }
    Product_B* Create_Product_B(){
        return new Product_B_Add;
    }
};

//工厂类
class Factory_Mul:public Factory{
public:
    Product_A* Create_Product_A(){
        return new Product_A_Mul;
    }
    Product_B* Create_Product_B(){
        return new Product_B_Mul;
    }

};

int main(){
    Factory_Add* factory_add = new Factory_Add();
    Factory_Mul* factory_mul = new Factory_Mul();
    int A_add_result = factory_add->Create_Product_A()->operation(1, 2);
    int A_mul_result = factory_mul->Create_Product_A()->operation(1, 2);
    int B_add_result = factory_add->Create_Product_B()->operation(1, 2);
    int B_mul_result = factory_mul->Create_Product_B()->operation(1, 2);
    cout <<"A_op_add: " <<A_add_result << endl;
    cout <<"A_op_multiply: " << A_mul_result << endl;
    cout <<"B_op_add: " <<B_add_result << endl;
    cout <<"B_op_multiply: " << B_mul_result << endl;
    system("pause");
    return 0;
}
```

（3）建造者模式

建造者模式（Builder）适用于创建包含不同组件的对象，它分为 Product、Builder 和 Director 三部分，分别定义了产品组件的添加方式、差异组件的扩展方式、组件装配流程的

对外接口，如下：




```cpp
class Product
{
public:
    void AddPart(string part){
        m_Parts.push_back(part);
    };
    void Show(){
        cout << "组成部分:" << endl;
        for(auto part : m_Parts)
        {
            cout << part << endl;
        }
    };

    int PartsCount(){
        return m_Parts.size();
    };
private:
    vector<string>   m_Parts;
};


class Builder
{
public:
    Builder(){m_Product = new Product();};
    virtual void AddBachelor()=0;
    virtual void AddMaster()=0;
    virtual void AddDoctor()=0;
    Product* GetProduct(){
        if(m_Product->PartsCount() > 0)
            return m_Product;
        else{
            cout << "组件不足" << std::endl;
            return NULL;
        }
    };
protected:
    Product*    m_Product;
};

class Director
{
public:
    Product* ConstructBachelor(Builder* builder){
        builder->AddBachelor();
        return builder->GetProduct();
    };
    Product* ConstructMaster(Builder* builder){
        builder->AddBachelor();
        builder->AddMaster();
        return builder->GetProduct();
    };
    Product* ConstructDocter(Builder* builder){
        builder->AddBachelor();
        builder->AddMaster();
        builder->AddDoctor();
        return builder->GetProduct();
    };
};
```

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class TJUerBuilder: public Builder
{
public:
    void AddBachelor() override{
        m_Product->AddPart("天大学士学位");
    };
    void AddMaster() override{
        m_Product->AddPart("天大硕士学位");
    };
    void AddDoctor() override{
        m_Product->AddPart("天大博士学位");
    };
};
class NKUerBuilder: public Builder
{
public:
    void AddBachelor() override{
        m_Product->AddPart("南开学士学位");
    };
    void AddMaster() override{
        m_Product->AddPart("南开硕士学位");
    };
    void AddDoctor() override{
        m_Product->AddPart("南开博士学位");
    };
};

int main(int argc, char *argv[])
{
    Director director;

    Product* tjuer = director.ConstructBachelor(new
TJUerBuilder);
    tjuer->Show();

    Product* nkuer = director.ConstructDocter(new
NKUerBuilder);
    nkuer->Show();

    system("pause");
    return 0;
}
```

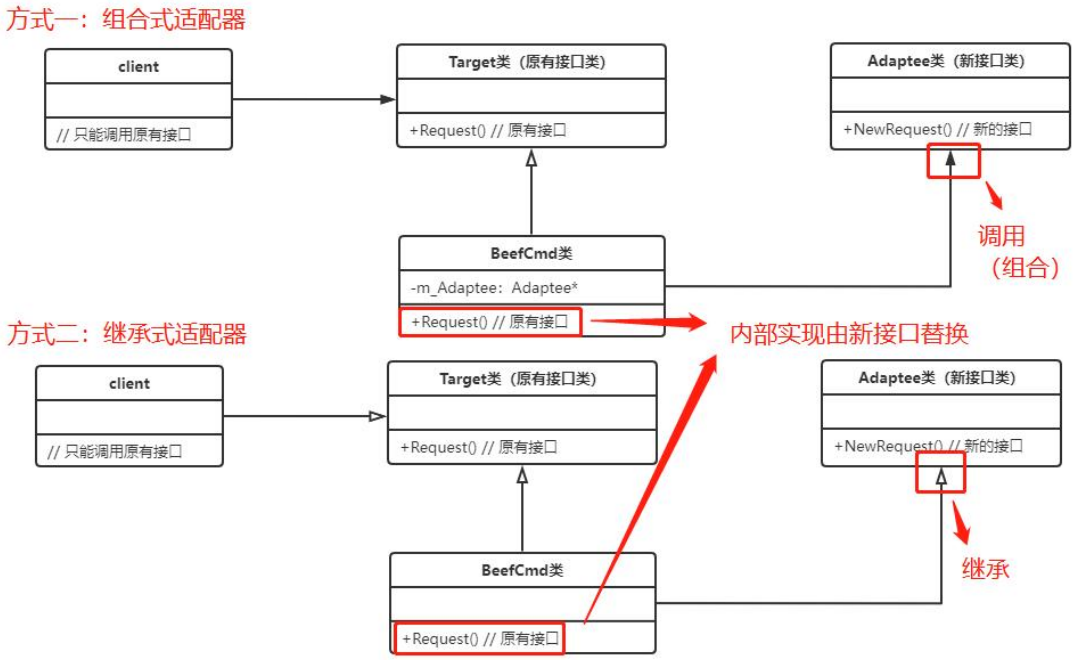
2、结构型模式

（1）适配器模式

适配器模式（Adapter）是用来在旧的系统接口中替换成新的系统接口的设计模式，简单说对于同一个功能存在一个老版本的类对象 A，因为需求升级开发了新的类对象 B，A 和 B 实现相同的功能但是类的接口名称不一致，原有程序无法直接调用类 B。而且由于系统生态的原因，大部分程序只能调用类 A 的接口，所以也不能舍弃 A。

这种情况下就得使用适配器模式。假设存在一个原有接口 Target 和一个新的接口

Adaptee，现在要使 Target 中的接口的实现被替换成 Adaptee 的实现。适配器模式的做法是首先从 Target 派生出一个子类 Adapter，Adapter 可以调用 Adaptee 的函数（组合方式）或继承 Adaptee 的函数来替换自身父类的实现。




```
//原有系统接口A: Target
class SysATarget
{
public:
    virtual void getName()
    {
        cout << "获取系统A中员工的名称" << endl;
    }
};

//新的系统接口B: Adaptee
class SysBAdaptee
{
public:
    virtual void getSysBName()
    {
        cout << "获取系统B中员工的名称" << endl;
    }
};
```

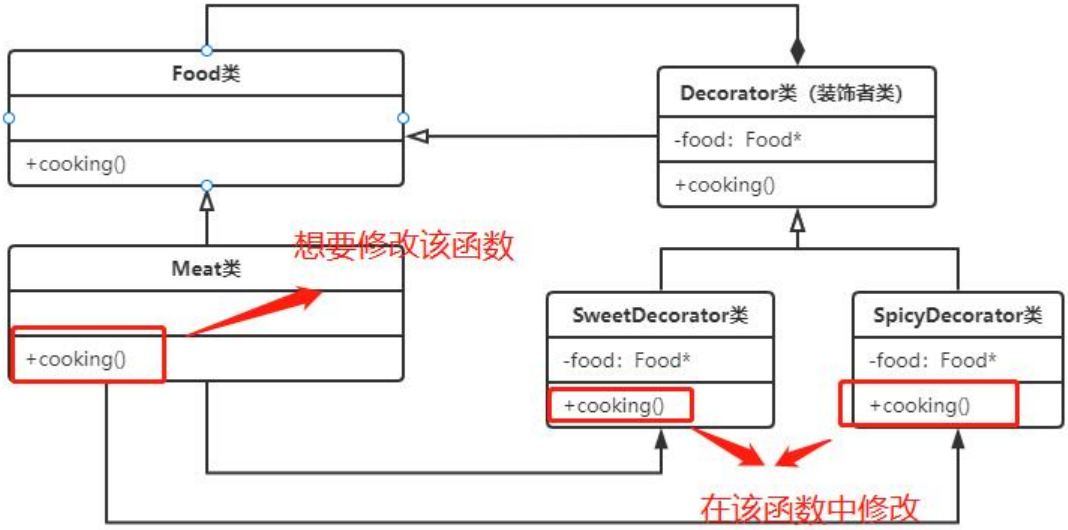
```
#include <iostream>
using namespace std;
//组合适配器: Adapter
class Adapter1: public SysATarget
{
public:
    Adapter1(SysBAdaptee *pSysB) :m_pSysB(pSysB){}
    void getName() override
    {
        m_pSysB->getSysBName();
    }
private:
    SysBAdaptee *m_pSysB;
};

//继承适配器: Adapter
class Adapter2: public SysATarget,public SysBAdaptee
{
public:
    void getName() override
    {
        this->getSysBName();
    }
};

int main()
{
    //方式一：以组合的方式创建适配器
    SysATarget *sysA_1 = new Adapter1(new
SysBAdaptee);
    sysA_1->getName();
    //方式二：以继承的方式创建适配器
    SysATarget *sysA_2 = new Adapter2;
    sysA_2->getName();
    system("pause");
    return 0;
}
```

（2）装饰者模式

装饰者模式（Decorator）也叫包装模式。如果给一个已有的类添加新的实现方法，一种思路是继承该类，在子类中添加；另一种思路是组合。装饰者模式就是利用组合的方式在已由类添加新的功能。

假如现在有个 Food 类，其有一个公有接口 Cooking()，Food 类派生出了 1 个子类 Meat，

其 Cooking()会输出肉。



这时候由于新的需求，需要修改 Meat 类的 Cooking()，使其输出加了辣味或甜味的肉，如果不直接修改 Meat 类，一种方式是继承自 Meat 类，重写其 Cooking()。装饰者模式的思路是不继承 Meat()类，而是在 Food 派生出的新的子类 Decorator，这个 Decorator 具有一个 Food 指针，从 Decorator 派生出更多子类，例如 SweetDecorator。

由于 Decorator 和 Meat 的父类都是 Food，所以 SweetDecorator 也可修饰 Decorator 类及其子类。如果想要一块添加了辣味和甜味的肉，不需要再新加装饰类，只需要用将 SweetDecorator 作为参数构造出一个 SpicyDecorator 类即可，可见代码。

```
class Food
{
public:
    virtual void Cooking() = 0;//烹饪
};

class Meat :public Food
{
public:
    void Cooking()
    {
        cout << "Cooking Meat" << endl;
    }
};

class Decorator : public Food
{
protected:
    Decorator(Food* food) :m_food(food){};
    Food* m_food;
};
```

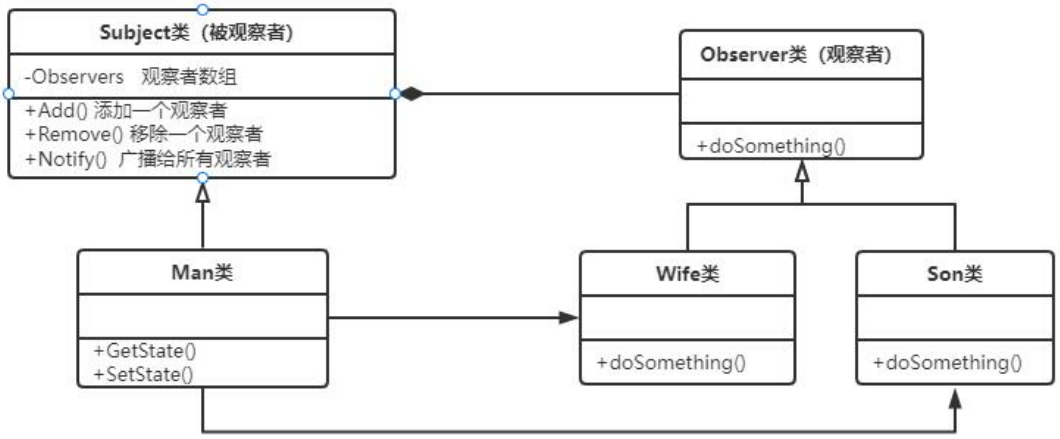
```
class SweetDecorator :public Decorator
{
public:
    SweetDecorator(Food* food) :Decorator(food){};//调用基类的构造器
    void Cooking()
    {
        cout << "Add Sweet" << endl;
        m_food->Cooking();
    }
};
class SpicyDecorator :public Decorator
{
public:
    SpicyDecorator(Food* food) :Decorator(food){};//调用基类的构造器
    void Cooking()
    {
        cout << "Add Spicy" << endl;
        m_food->Cooking();
    }
};

int main(int argc, char *argv[])
{
    //创建肉
    Food* meat = new Meat();
    //创建装饰器 甜味、辣味
    Decorator* sweetDecorator= new SweetDecorator(meat);
    Decorator* spicyDecorator = new SpicyDecorator(meat);
    //甜的肉
    sweetDecorator->Cooking();
    cout << "**********" << endl;
    //辣的肉
    spicyDecorator->Cooking();
    cout << "**********" << endl;
    //又甜又辣的肉
    Decorator* sweetSpicyDecorator= new SpicyDecorator(sweetDecorator);
    sweetSpicyDecorator->Cooking();

    system("pause");
    return 0;
}
```

3、行为型模式

（1）观察者模式

观察者模式（Observer）很可能是应用最多、影响最广的设计模式之一，也叫发布-订阅模式。观察者模式的核心是一对多的通知，当被观察者状态发生变化时（发生某个事件）通知所有观察者做出响应。

经典应用场景：微信朋友圈动态通知、消息通知、邮件通知等。经典的 MVC（Model/View/Control）结构就采用了其思路，其将业务逻辑和表示层进行接口。



代码实现思路：被观察者发生某个事件时所有观察者都要做出相应，这就要求被观察者要维护一个观察者的数组，然后当某个函数（暂定为 notify）被调用时就遍历该数组调用观察者的函数，这就要求所有的观察者提供一个统一的接口（暂定为 doSomething）。

观察者 Observer 和被观察者 Subject 的接口实现也非常简单，主要是被观察者要维护一个 vector 数组。

```
// 观察者基类
class Observer{
public:
    virtual void doSomething()=0;
};
// 被观察者基类
class Subject{
public:
    virtual void Add(Observer* obr){
        this->observers.push_back(obr);
    };
    virtual void Remove(Observer* obr){
        auto pos=find(observers.begin(),observers.end(),obr);
        if(pos!=observers.end()){
            observers.erase(pos);
        }
    };
    virtual void Notify(){
        for(const auto& obs:observers){
            obs->doSomething();
        }
    };
private:
    vector<Observer*> observers;
};
```

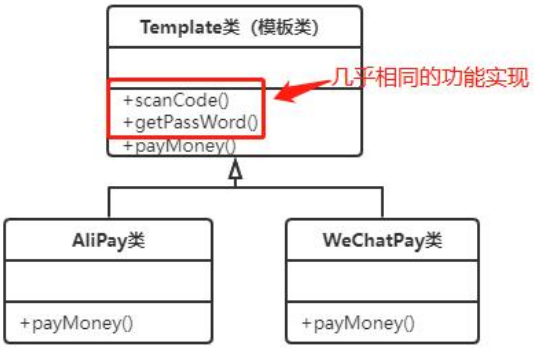
测试场景，丈夫回家发通知给妻子和孩子，丈夫是被观察者，妻子和孩子是观察者。如下：

```cpp
// 被观察者子类
class Man:public Subject{
public:
    Man(){
        this->state=0;
    }
    void setState(int newState) {
        cout<<"丈夫：累了一天，准备回家，给老婆孩子发通知..."<<endl;
        this->state=newState;
        this->Notify();
    }
    int getState() const{
        return state;
    }
private:
    int state;
};

// 观察者子类1
class Wife:public Observer{
public:
    void doSomething() override {
        cout<<"妻子：接到通知，老公要回来了，准备做饭。"<<endl;
    }
};
// 观察者子类2
class Son:public  Observer{
public:
    void doSomething() override{
        cout<<"孩子：接到通知，爸爸快回来了，准备学习。"<<endl;
    }
};

int main(int argc, char *argv[])
{
    Man man;
    Observer* wife=new Wife;
    Observer* son=new Son;
    man.Add(wife);
    man.Add(son);

    //下班了 发消息
    man.setState(1);

    delete wife;
    delete son;

    system("pause");
    return 0;
}
```

（2）模板模式

模板模式（template）是一个非常自然也非常简单的设计模式，我们甚至不知不觉都在使用它。它解决的是如何组织业务（算法）逻辑相似的对象，例如微信支付、支付宝支付。模板模式采用继承的方式实现这一点，为算法逻辑相似的对象定义一个共同的抽象基类，子类负责实现具体的细节。

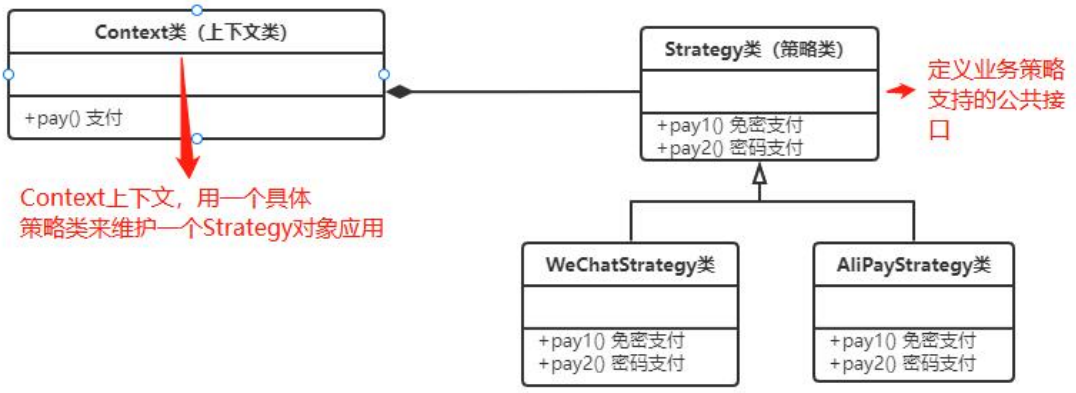
生活中常见的模板模式的例子：付款购物的流程：扫描获取二维码、获取支付密码、支付金额，其中支付金额可以由微信支付和支付宝支付两种接口来完成，其他的则可以使用同一套方法。



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

class PayTemplate{
public:
    void pay(int count){
        this->scanCode();
        this->getPassWord();
        this->payMoney(count);
    };
protected:
    void scanCode(){cout<<"已成功扫描二维码"<<endl;};
    void getPassWord(){cout<<"已成功获取支付密码"<<endl;};
    virtual void payMoney(int count)=0;
};


class WeChatPayStrategy:public PayTemplate{
    void payMoney(int count) override{
        cout<<"微信密码支付"<<count<<endl;
    }
};

class AliPayStrategy:public PayTemplate{
    void payMoney(int count) override{
        cout<<"支付宝密码支付"<<count<<endl;
    }
};

int main(int argc, char *argv[])
{
    PayTemplate* payTemplate1 = new WeChatPayStrategy;
    payTemplate1->pay(100);
    PayTemplate* payTemplate2 = new AliPayStrategy;
    payTemplate2->pay(100);
    delete payTemplate1;
    delete payTemplate2;

    system("pause");
    return 0;
}
```

（3）策略模式

策略模式（Strategy）和模板模式解决的都是同一种问题，即如何组织业务（算法）逻辑相似的对象。例如开发一个支付模块，有微信支付和支付宝支付 2 种软件包的各种接口供调用，由于这两种支付是不同厂商开发的，接口名称可能都不统一。前面所述的模板模式直接为这 2 种支付定义一个共同基类；策略模式的方式是定义一个 Strategy 类，其中定义支付业务都会用到的公共接口，由此派生出两个具体的支付策略类。Context 类负责根据 Strategy 类提供的公共接口来开发具体业务：例如支付时选用免密支付方式还是密码支付方式。



策略模式的优点是使用继承的方式已于修改和扩展；缺点是破坏了类的封装性，继承中父类的实现细节暴露给了子类，策略类会增多，所有的策略类都会对外暴露。

```
class Strategy{
public:
    virtual void pay1(int count)=0; // 业务逻辑接口：免密支付
    virtual void pay2(int count)=0; // 业务逻辑接口：密码支付
};

class Context{
public:
    Context(Strategy* strategy,int payMethord=1){
        this->strategy=strategy;
        this->payMethord = payMethord;
    };
    void pay(int count){
        if(this->payMethord==1)
            this->strategy->pay1(count);
        else if(this->payMethord==2)
            this->strategy->pay2(count);
        else{
            cout<<"无此支付方式，支付失败"<<endl;
        }
    }
private:
    Strategy* strategy;
    int payMethord;
};
```

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

class WeChatPayStrategy:public Strategy{
    void pay1(int count) override{
        cout<<"微信免密支付"<<count<<endl;
    }
    void pay2(int count) override{
        cout<<"微信密码支付"<<count<<endl;
    }
};

class AliPayStrategy:public Strategy{
    void pay1(int count) override{
        cout<<"支付宝免密支付"<<count<<endl;
    }
    void pay2(int count) override{
        cout<<"支付宝密码支付"<<count<<endl;
    }
};

int main(int argc, char *argv[])
{
    Context* context1 = new Context(new WeChatPayStrategy,1);
    context1->pay(100);
    Context* context2 = new Context(new AliPayStrategy,2);
    context2->pay(100);
    delete context1;
    delete context2;

    system("pause");
    return 0;
}
```

（4）命令模式

命令模式（Command）可以实现消息通信的发送者和接收者的完全解耦，发送者和接收者之间没有直接饮用官谢，发送者只需要知道如何发送请求。

例如服务员和厨师的沟通场景，厨师是命令接收者 Receiver，服务员是命令发送者 Invoker。普通写法下，Invoker 根据不同的命令直接调用 Receiver 的不同操作函数，这样子开发者需要频繁修改 Invoker 类。命令模式下，Receiver 和 Invoker 之间添加一个 Command 类，Invoker 只调用 Command 的一个统一接口，不同的命令由 Command 不同的子类内部进行处理，新加一个命令时，开发者不需要改动 Invoker，只需要继承一个 Command 子类，然后在其内部调用 Receiver 的合适操作接口即可。