结构型模式

- 结构型模式(Structural Pattern)关注如何将现有类或对象组织在一起形成更加强大的结构
- 不同的结构型模式从不同的角度组合类或对象，它们在尽可能满足各种面向对象设计原则的同时为类或对象的组合提供一系列巧妙的解决方案

七、适配器模式

1. 介绍

1.1 模式说明

定义一个包装类，用于包装不兼容接口的对象

> 包装类 = 适配器Adapter；被包装对象 = 适配者Adaptee = 被适配的类

1.2 主要作用

把一个类的接口变换成客户端所期待的另一种接口，从而使原本接口不匹配而无法一起工作的两个类能够在一起工作。

> 适配器模式的形式分为：类的适配器模式 & 对象的适配器模式

1.3 解决的问题

原本由于接口不兼容而不能一起工作的那些类可以在一起工作

---

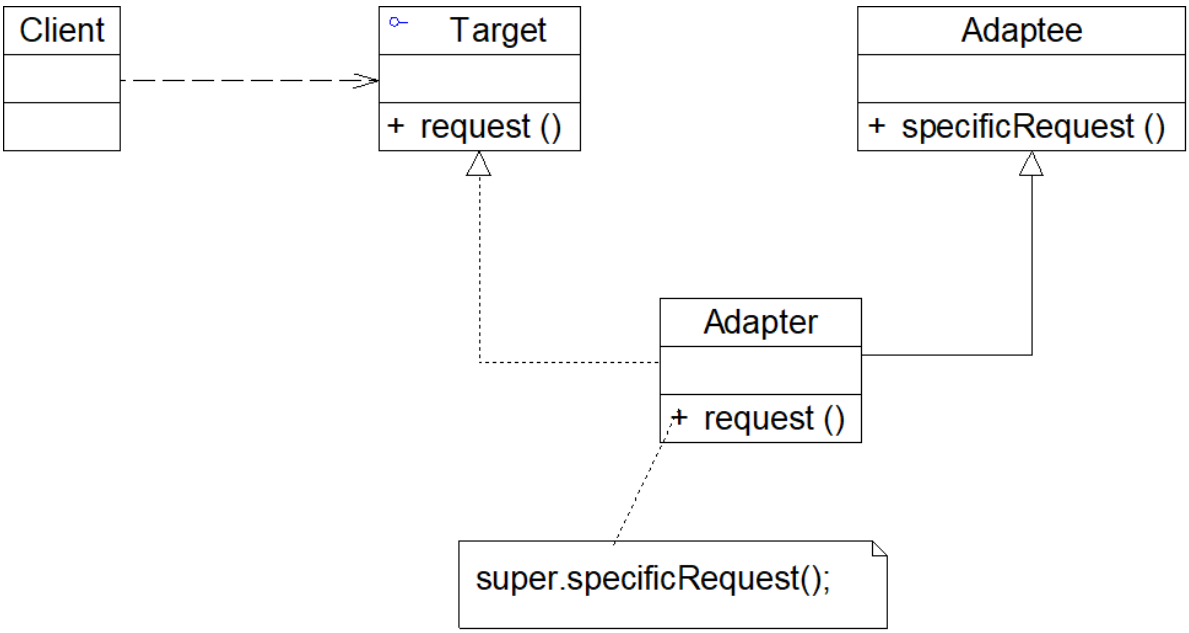
2. 模式原理

2.1 类的适配器模式

类的适配器模式是把适配的类的API转换成为目标类的API。

2.1.1 UML

类适配器：



- 冲突：Target期待调用Request方法，而Adaptee并没有（这就是所谓的不兼容了）。
- 解决方案：为使Target能够使用Adaptee类里的SpecificRequest方法，故提供一个中间环节Adapter类**（继承Adaptee & 实现Target接口）**，把Adaptee的API与Target的API衔接起来（适配）。

2.1.2 代码

步骤1： 创建**Target接口**

```
public interface Target {

    //这是源类Adapteee没有的方法
    public void Request();
}
```

步骤2： 创建**源类（Adaptee）**

```
public class Adaptee {

    public void SpecificRequest(){
    }
}
```

步骤3： 创建**适配器类（Adapter）**

```
//适配器Adapter继承自Adaptee，同时又实现了目标(Target)接口。
public class Adapter extends Adaptee implements Target {

    //目标接口要求调用Request()这个方法名，但源类Adaptee没有方法Request()
    //因此适配器补充上这个方法名
    //但实际上Request()只是调用源类Adaptee的SpecificRequest()方法的内容
    //所以适配器只是将SpecificRequest()方法作了一层封装，封装成Target可以调用的Request()而已
    @Override
    public void Request() {
        this.SpecificRequest();
    }
}
```

步骤4：定义具体使用目标类，并通过Adapter类调用所需要的方法从而实现目标

```
public class AdapterPattern {

    public static void main(String[] args){

        Target mAdapter = new Adapter();
        mAdapter.Request () ;

    }
}
```

2.1.3 实例讲解 接下来我用一个实例来对类的适配器模式进行更深一步的介绍。**a. 实例概况**

- 背景：小成买了一个进口的电视机
- 冲突：进口电视机要求电压（110V）与国内插头标准输出电压（220V）不兼容

- 解决方案：设置一个适配器将插头输出的220V转变成110V

> 即适配器模式中的类的适配器模式

b. 使用步骤

步骤1： 创建**Target接口（期待得到的插头）**：能输出110V（将220V转换成110V）

```
public interface Target {

    //将220V转换输出110V（原有插头（Adaptee）没有的）
    public void Convert_110v();
}
```

步骤2： 创建源类（原有的插头）

```
class PowerPort220V{
//原有插头只能输出220V
    public void Output_220v(){
    }
}
```

步骤3：创建适配器类（Adapter）

```
class Adapter220V extends PowerPort220V implements Target{
   //期待的插头要求调用Convert_110v()，但原有插头没有
    //因此适配器补充上这个方法名
    //但实际上Convert_110v()只是调用原有插头的Output_220v()方法的内容
    //所以适配器只是将Output_220v()作了一层封装，封装成Target可以调用的
Convert_110v()而已

    @Override
    public void Convert_110v(){
      this.Output_220v;
    }
}
```

步骤4：定义具体使用目标类，并通过Adapter类调用所需要的方法从而实现目标（不需要通过原有插头）

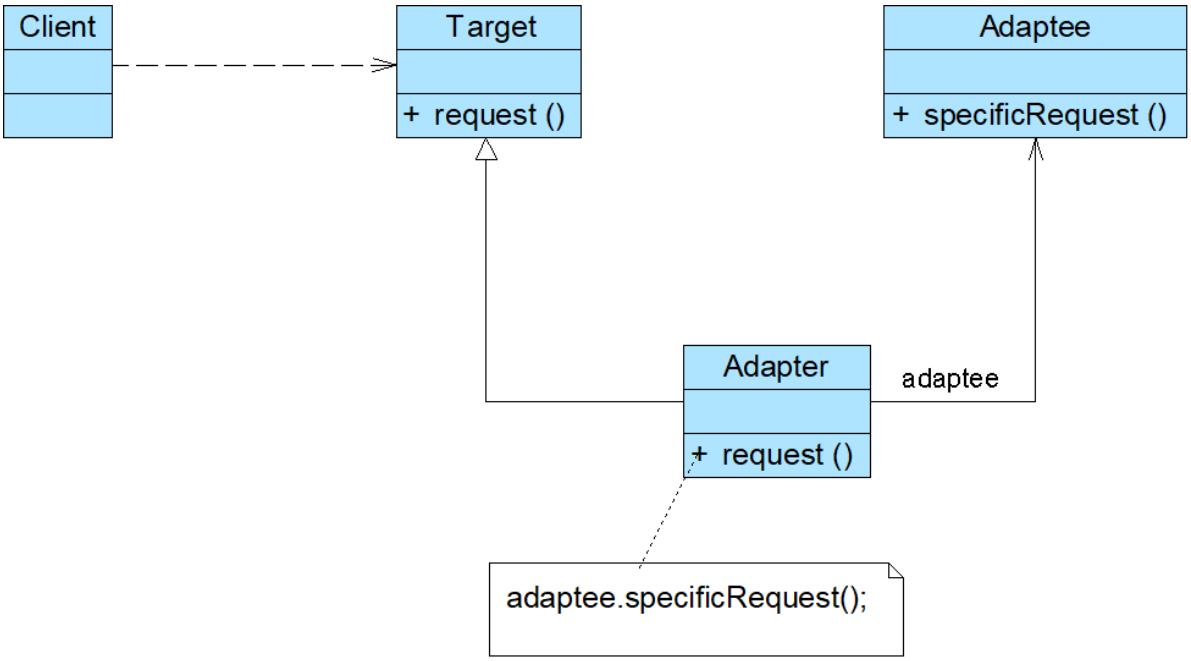
```
//进口机器类
class ImportedMachine {

    @Override
    public void Work() {
        System.out.println("进口机器正常运行");
    }
}

//通过Adapter类从而调用所需要的方法
public class AdapterPattern {
    public static void main(String[] args){

        Target mAdapter220V = new Adapter220V();
        ImportedMachine mImportedMachine = new ImportedMachine();

```

```
        //用户拿着进口机器插上适配器（调用Convert_110v()方法）
        //再将适配器插上原有插头（Convert_110v()方法内部调用Output_220v()方法输出220V）
        //适配器只是个外壳，对外提供110V，但本质还是220V进行供电
        mAdapter220V.Convert_110v();
        mImportedMachine.Work();
    }
}
```

2.2 对象的适配器模式

与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到Adaptee类，而是使用委派关系连接到Adaptee类。

对象适配器：



2.2.2 使用步骤

步骤1： 创建**Target接口**

```
public interface Target {

    //这是源类Adapteee没有的方法
    public void Request();
}
```

步骤2： 创建**源类（Adaptee）**

```
public class Adaptee {

    public void SpecificRequest(){
    }
}
```

步骤3： 创建**适配器类（Adapter）**

```
class Adapter implements Target{
    // 直接关联被适配类
```

```
    private Adaptee adaptee;

    // 可以通过构造函数传入具体需要适配的被适配类对象
    public Adapter (Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void Request() {
        // 这里是使用委托的方式完成特殊功能
        this.adaptee.SpecificRequest();
    }
}
```

步骤4：定义具体使用目标类，并通过Adapter类调用所需要的方法从而实现目标

```
public class AdapterPattern {
    public static void main(String[] args){
        //需要先创建一个被适配类的对象作为参数
        Target mAdapter = new Adapter(new Adaptee());
        mAdapter.Request();

    }
}
```

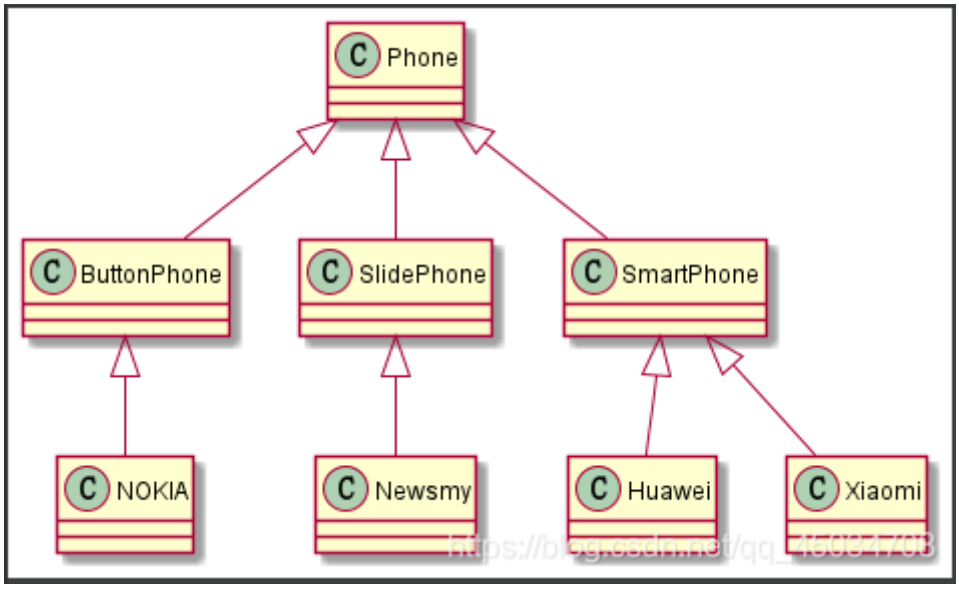
八、桥接模式

1、引例

虽然智能手机占据绝大市场，但诺基亚等老牌手机仍可用作于老年机、学生机等。

需求：对不同手机类型的不同品牌（比如按键手机：诺基亚、翻盖手机：纽曼、智能手机：华为、小米）实现操作编程(比如: 开机、关机、打电话)。

先来说说一般解法：将不同手机类型继承父类手机，最后各个品牌再继承对应手机类型：



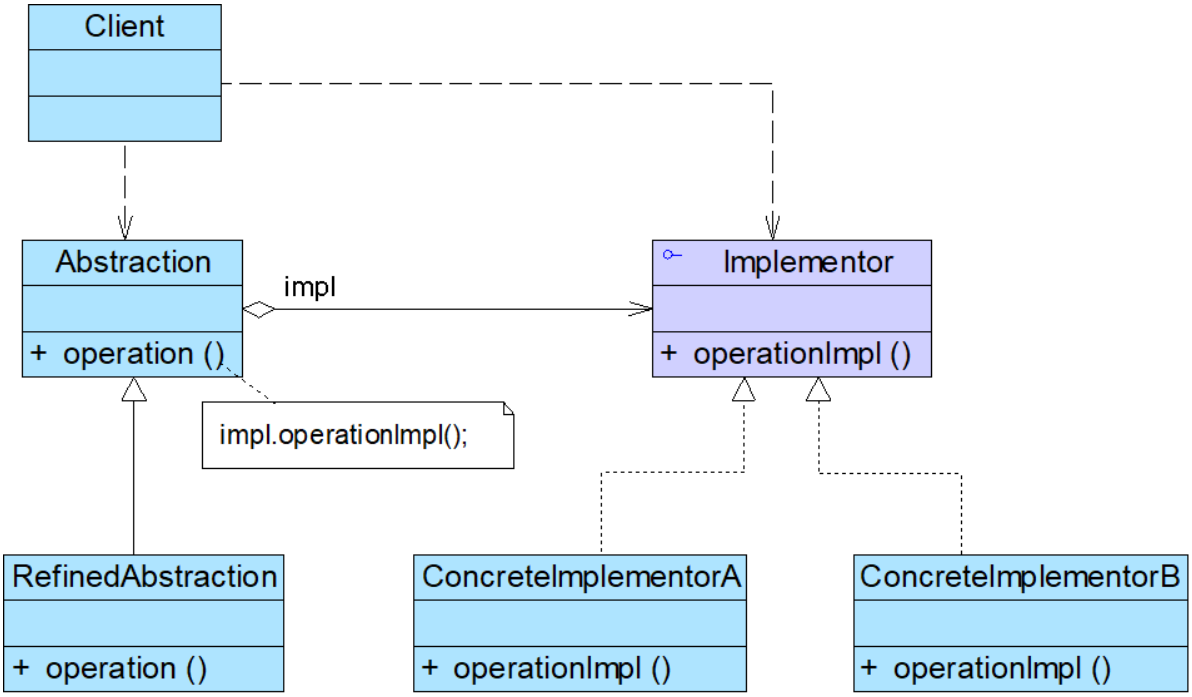
弊端：乍一看没问题，但其实不易扩展（类爆炸），如果增加新的手机类型（比如新兴的折叠式），就需要增加各个手机品牌的类去继承（比如已继承智能手机的华为小米）。同样如果我们增加一个手机品牌，也要在各个手机样式类下增加。违反了单一职责原则，维护成本高。

解决方案就是下面的主角：桥接模式。

2、桥接模式

桥接模式(Bridge)是一种结构型设计模式。顾名思义，就像搭个桥连接起来，通过使用封装、聚合及继承等行为让不同的类承担不同的职责，将实现与抽象放在两个不同的类层次中，使两个层次可以独立改变，保持各部分的独立性以及应对他们的功能扩展。

3、原理类图：



- Client类：客户端调用
- Abstraction抽象类：充当桥接类，维护了Implementor接口（即它的实现类ConcreteImplementorA...）
- RefindAbstraction类：是抽象类的子类
- Implementor接口：行为实现接口
- ConcreteImplementorA/B类：行为的具体实现类

从UML类图看出，抽象类和接口是聚合的关系，即调用和被调用的关系。如此一来搭好桥后，具体实现类调用方法=》父类抽象类的方法=》行为接口方法=》具体接口行为实现类，以完成连接，同时两者又相互独立易扩展：

4、代码

实现类接口

```
1 public interface Implementor {
2     public void operationImpl();
3 }
```

具体实现类

```
public class ConcreteImplementor implements Implementor {
    public void operationImpl() {
        //具体业务方法的实现
    }
}
```

抽象类

```
public abstract class Abstraction {
    protected Implementor impl; //定义实现类接口对象

    public void setImpl(Implementor impl) {
        this.impl=impl;
    }

    public abstract void operation(); //声明抽象业务方法
}
```

扩充抽象类（细化抽象类）

```
public class RefinedAbstraction extends Abstraction {
    public void operation() {
        //业务代码
        impl.operationImpl(); //调用实现类的方法
        //业务代码
    }
}
```

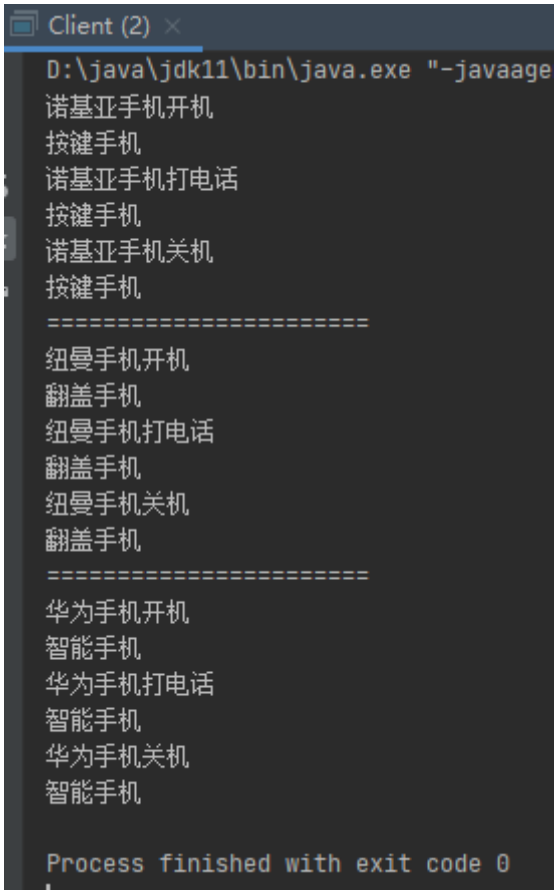
5、实例

```
//接口
public interface Brand {
    void open(); //开机
    void close(); //关机
    void call();//打电话
}

//接口实现类
public class NOKIA implements Brand{
    @Override
    public void open() {
        System.out.println("诺基亚手机开机");
    }
    @Override
    public void close() {
        System.out.println("诺基亚手机关机");
    }
    @Override
    public void call() {
        System.out.println("诺基亚手机打电话");
    }
}

public class Newsmy implements Brand{
    @Override
```

```
    public void open() {
        System.out.println("纽曼手机开机");
    }
    @Override
    public void close() {
        System.out.println("纽曼手机关机");
    }
    @Override
    public void call() {
        System.out.println("纽曼手机打电话");
    }
}

public class Huawei implements Brand{
    @Override
    public void open() {
        System.out.println("华为手机开机");
    }
    @Override
    public void close() {
        System.out.println("华为手机关机");
    }
    @Override
    public void call() {
        System.out.println("华为手机打电话");
    }
}

public class Xiaomi implements Brand{
    @Override
    public void open() {
        System.out.println("小米手机开机");
    }
    @Override
    public void close() {
        System.out.println("小米手机关机");
    }
    @Override
    public void call() {
        System.out.println("小米手机打电话");
    }
}

//抽象类
public abstract class Phone {
    private Brand brand;//手机品牌接口
    public Phone(Brand brand) {//构造器
        super();
        this.brand = brand;
    }
    public void open() {
        this.brand.open();
    }
    public void close() {
        this.brand.close();
    }
    public void call() { this.brand.call(); }
}
```

```java

//抽象子类
public class ButtonPhone extends Phone{
    public ButtonPhone(Brand brand) {
        super(brand);
    }
    public void open() {
        super.open();
        System.out.println("按键手机");
    }
    public void close() {
        super.close();
        System.out.println("按键手机");
    }
    public void call() {
        super.call();
        System.out.println("按键手机");
    }
}

public class SlidePhone extends Phone{
    public SlidePhone(Brand brand) {
        super(brand);
    }
    public void open() {
        super.open();
        System.out.println("翻盖手机");
    }
    public void close() {
        super.close();
        System.out.println("翻盖手机");
    }
    public void call() {
        super.call();
        System.out.println("翻盖手机");
    }
}

public class SmartPhone  extends Phone{
    public SmartPhone(Brand brand) {
        super(brand);
    }
    public void open() {
        super.open();
        System.out.println("智能手机");
    }
    public void close() {
        super.close();
        System.out.println("智能手机");
    }
    public void call() {
        super.call();
        System.out.println("智能手机");
    }
}

//客户端调用
public class Client {
```

```
142     public static void main(String[] args) {
143         Phone phone1 = new ButtonPhone(new NOKIA());
144         phone1.open();
145         phone1.call();
146         phone1.close();
147 
148         System.out.println("======================");
149         Phone phone2 = new SlidePhone(new Newsmy());
150         phone2.open();
151         phone2.call();
152         phone2.close();
153 
154         System.out.println("======================");
155         Phone phone3 = new SmartPhone(new Huawei());
156         phone3.open();
157         phone3.call();
158         phone3.close();
159     }
160 }
```



6、总结

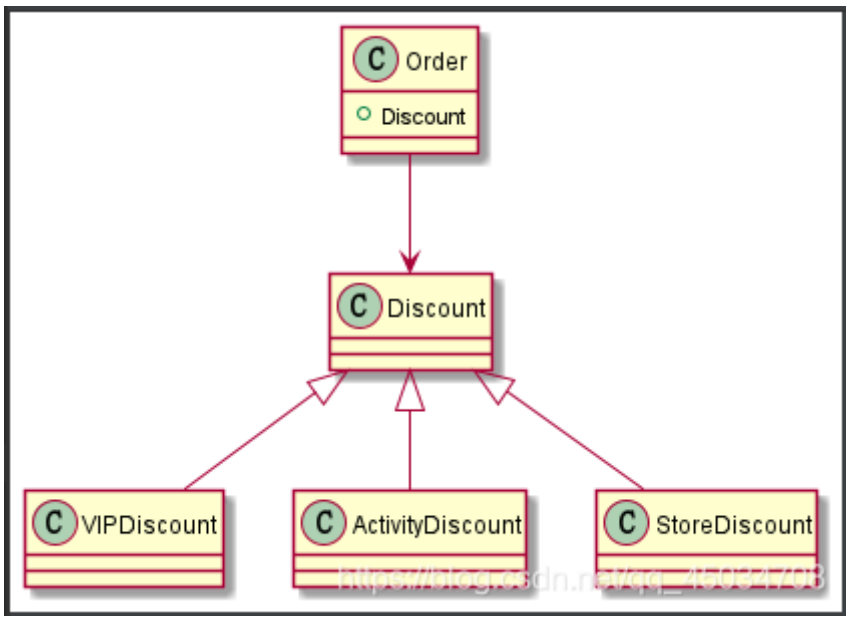
- 实现了抽象和实现部分的分离，从而极大的提供了系统的灵活性，让抽象部分和实现部分独立开来，这有助于系统进行分层设计，从而产生更好的结构化系统。
- 桥接模式替代多层继承方案，可以减少子类的个数，降低系统的管理和维护成本。
- 桥接模式的引入增加了系统的理解和设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计和编程
- 常见的应用场景:

-JDBC驱动程序 **-银行转账系统** 转账分类: 网上转账，柜台转账，AMT转账 转账用户类型：普通用户，银卡用户，金卡用户... **-消息管理** 消息类型：即时消息，延时消息 消息分类：手机短信，邮件消息，QQ消息...

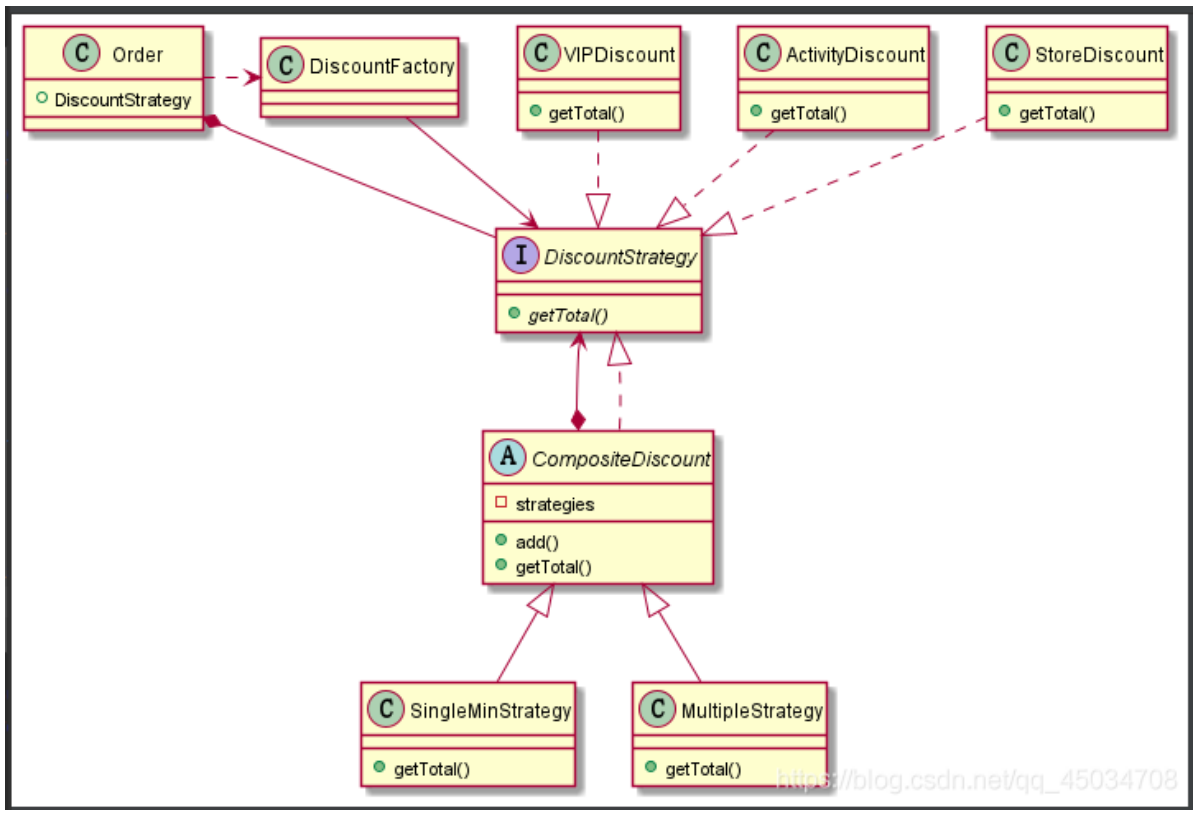
九、组合模式

以双十一的折扣为例，有VIP折扣、活动折扣、满减、店铺优惠券、红包……

先来看看一般的写法：



当用户只满足一种折扣方案时，这种方法还能应对。 但精打细算的我们往往是同时满足多种折扣方案，这时就可以用组合模式，把这些单个折扣方案组合容纳起来，然后定义解决折扣冲突策略。实现单个对象和组合对象的统一，让调用者使用时不必在区分。



把组合对象CompositeDiscount定义成抽象类，SingleMinStrategy和MultipleStrategy继承它，表示解决冲突的策略，分别是取最小折扣和取折上折。 一般解决折扣冲突都是折上折，但商家往往更精明，推出互斥券之类的，即取最小折扣。也可以自定义其他折扣冲突策略。

实例代码：

```
public interface DiscountStrategy {
    public double getTotal(double price);
}

public class VIPDiscount implements DiscountStrategy {
    //95折
    @Override
    public double getTotal(double price) {
        return 0.95*price;
    }
}
public class ActivityDiscount implements DiscountStrategy{
    //9折
    @Override
    public double getTotal(double price) {
        return 0.9*price;
    }
}

public class StoreDiscount implements DiscountStrategy{
    //满500超出部分打6折
    @Override
    public double getTotal(double price) {
        return 500+0.6*(price-500);
    }
}
```

```
public abstract class CompositeDiscount implements DiscountStrategy {
    protected List<DiscountStrategy> strategies =new ArrayList(); //容器
    public void add(DiscountStrategy discountStrategy){ //添加叶子结点
        strategies.add(discountStrategy);
    }

    @Override
    public double getTotal(double price) {
        return price;
    }
}

//多种折扣选最低折扣
public class SingleMinStrategy extends CompositeDiscount {
    @Override
    public double getTotal(double price) {
        double rtn=price;
        for (DiscountStrategy s: strategies) {
            rtn=Math.min(rtn,s.getTotal(price));
        }
        return rtn;
    }
}

//多种折扣用折上折
public class MultipleStrategy extends CompositeDiscount {
    @Override
```

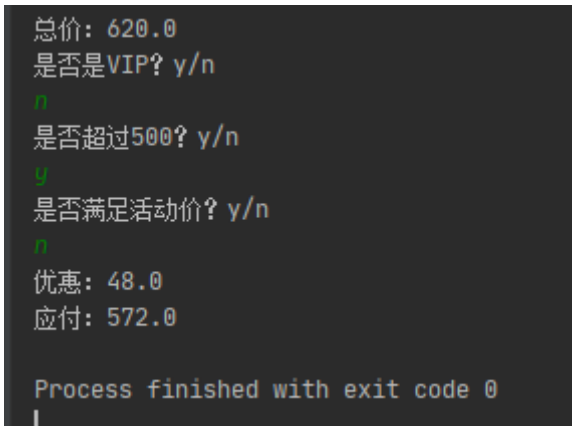
```
    public double getTotal(double price) {
        double rtn = price;
        for (DiscountStrategy s : strategies) {
            rtn = s.getTotal(rtn);
        }
        return rtn;
    }
}
```

```
public class DiscountFactory {
    public DiscountStrategy create(String type){ //工厂来创建相应策略
        //单一折扣策略
        if("ynn".equals(type))return new VIPDiscount();
        else if("nyn".equals(type))return new StoreDiscount();
        else if("nny".equals(type))return new ActivityDiscount();
        else{  //多种折扣策略
            CompositeDiscount compositeDiscount;
            System.out.println("请选择冲突解决方案：1.折上折 2.最低折");
            Scanner scanner=new Scanner(System.in);
            int type2=scanner.nextInt();
            if(type2==1){
                compositeDiscount=new MultipleStrategy();
            }
            else{
                compositeDiscount=new SingleMinStrategy();
            }
            if(type.charAt(1)=='y')compositeDiscount.add(new
StoreDiscount());
            if(type.charAt(0)=='y')compositeDiscount.add(new VIPDiscount());
            if(type.charAt(2)=='y')compositeDiscount.add(new
ActivityDiscount());
            return compositeDiscount;
        }
    }
}

```

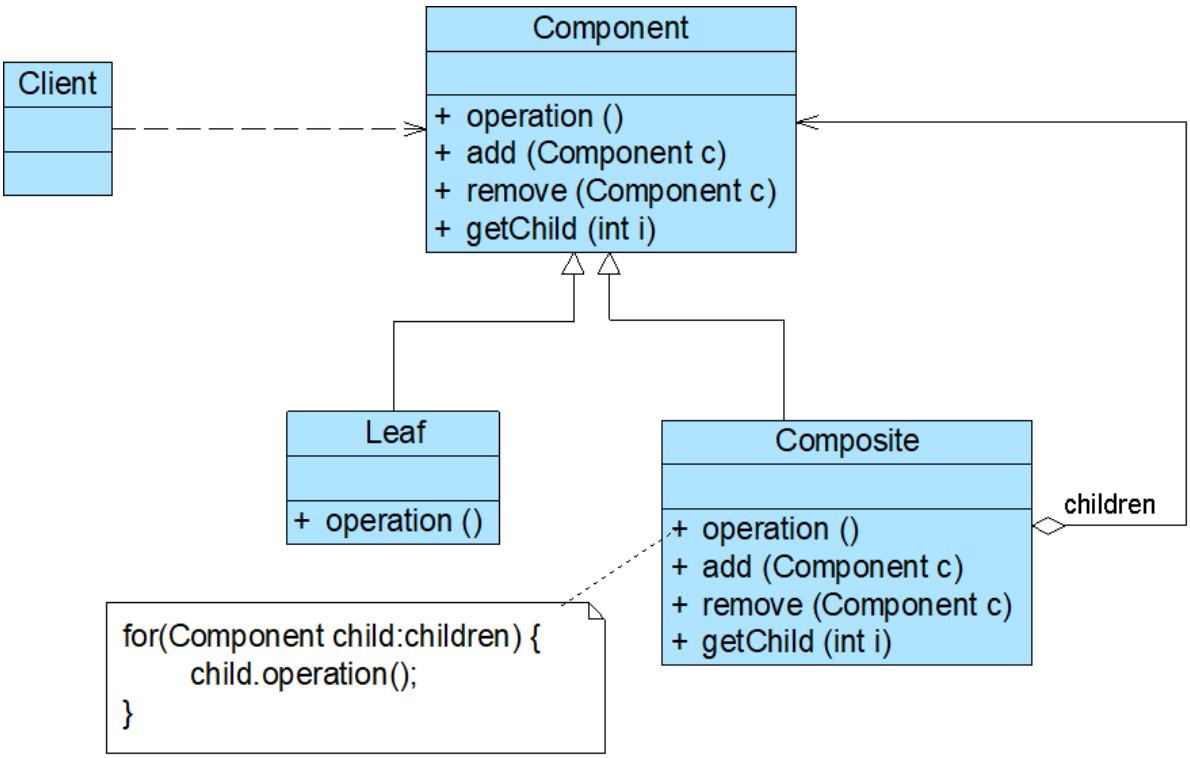
```
public class Order {
    public double price;
    private String type;
    public DiscountStrategy discountStrategy;
    public Order(double price) {
        this.price=price;
    }

    public void display(){
        System.out.println("总价："+price);
        System.out.println("是否是VIP？y/n");
        Scanner scanner=new Scanner(System.in);
        type=scanner.next();
        System.out.println("是否超过500？y/n");
        String tmp;
        tmp=scanner.next();
        type+=tmp;
        System.out.println("是否满足活动价？y/n");
        tmp=scanner.next();
```

```
        type+=tmp;
        DiscountFactory discountFactory=new DiscountFactory();
        double discountPrice=discountFactory.create(type).getTotal(price);
        System.out.println("优惠："+(price-discountPrice));
        System.out.println("应付："+discountPrice);

    }
}

public class Client {
    public static void main(String[] args) {
        Order order=new Order(620);
        order.display();
    }
}
```

运行结果：



UML



代码

抽象构件角色

```
public abstract class Component {
    public abstract void add(Component c); //增加成员
    public abstract void remove(Component c); //删除成员
    public abstract Component getChild(int i); //获取成员
    public abstract void operation();  //业务方法
}
```

叶子构件角色

```
public class Leaf extends Component {
    public void add(Component c) {
        //异常处理或错误提示
    }

    public void remove(Component c) {
        //异常处理或错误提示
    }

    public Component getChild(int i) {
        //异常处理或错误提示
        return null;
    }

    public void operation() {
        //叶子构件具体业务方法的实现
    }
}
```

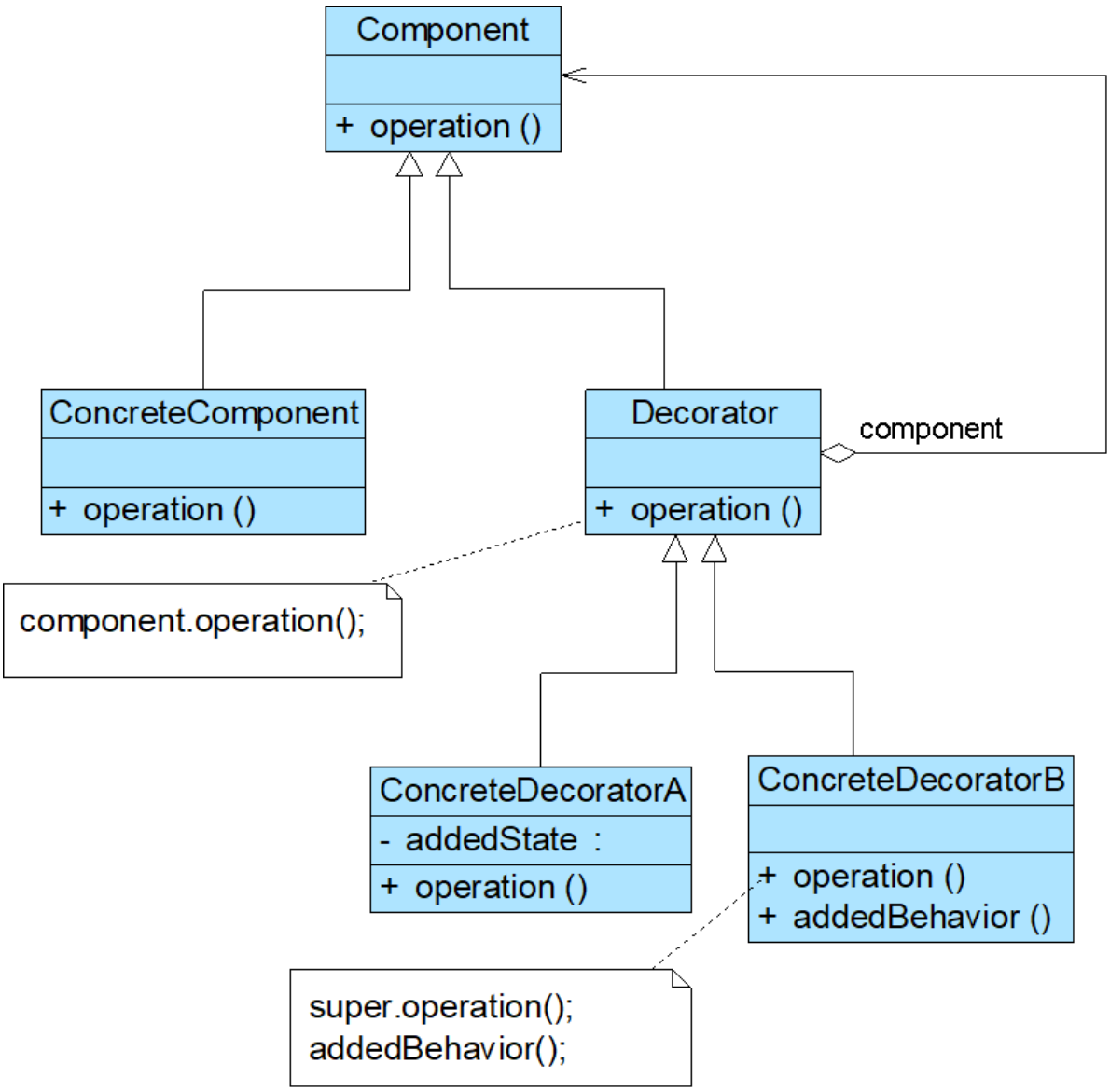
容器构件角色

```java
public class Composite extends Component {
    private ArrayList<Component> list = new ArrayList<Component>();

    public void add(Component c) {
        list.add(c);
    }

    public void remove(Component c) {
        list.remove(c);
    }

    public Component getChild(int i) {
        return (Component)list.get(i);
    }

    public void operation() {
        //容器构件具体业务方法的实现，将递归调用成员构件的业务方法
        for(Object obj:list) {
            ((Component)obj).operation();
        }
    }
}
```

十、装饰模式

需求：设现在有单品咖啡：Espresso(意大利浓咖啡)和LongBlack(美式咖啡)，调料有Milk(牛奶)和sugar(糖)，客户可以点单品咖啡或单品咖啡+调料的组合，计算相应费用。要求在扩展新的咖啡种类时，具有良好的扩展性、改动维护方便。

1、UML



- Component抽象类：主体，比如类似前面的`Drink`。
- ConcreteComponent类：具体的主体，比如前面的单品咖啡。
- Decorator类：装饰者，比如前面的调料
- ConcreteDecorator类：具体的装饰者，比如前面的牛奶。

2、代码

抽象构件类

```
public abstract class Component {
    public abstract void operation();
}
```

具体构件类

```
public class ConcreteComponent  extends Component {
    public void operation() {
        //实现基本功能
    }
}
```

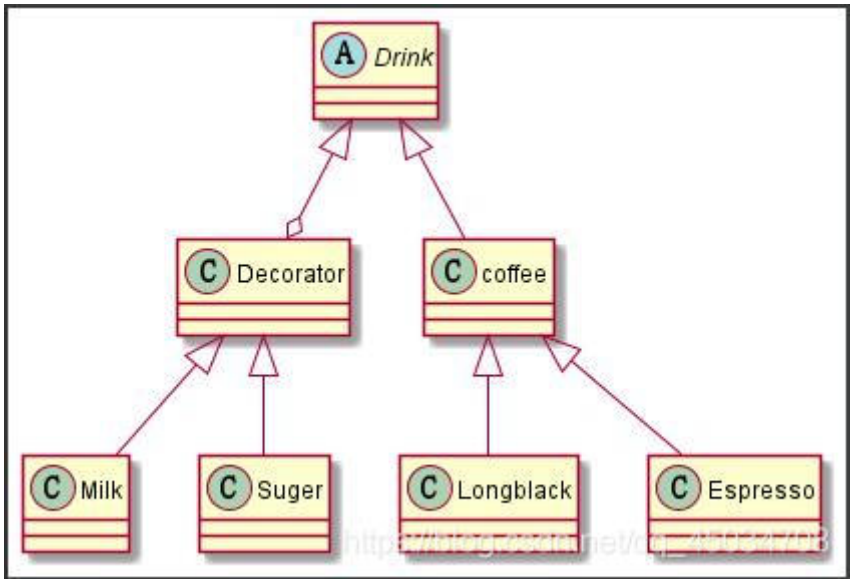
抽象装饰类

```java
public class Decorator extends Component {
    private Component component; //维持一个对抽象构件对象的引用

    //注入一个抽象构件类型的对象
    public Decorator(Component component) {
        this.component=component;
    }

    public void operation() {
        component.operation();  //调用原有业务方法
    }
}
```

具体装饰类

```java
public class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component component) {
        super(component);
    }

    public void operation() {
        super.operation(); //调用原有业务方法
        addedBehavior(); //调用新增业务方法
    }

    //新增业务方法
    public void addedBehavior() {
        ……
    }
}
```

装饰者解法类图:



- Drink 类就是前面说的抽象类
- Decorator 是一个装饰类，含有一个被装饰的对象(Drink obj)和的cost()方法进行一个费用的叠加计算，递归的计算价格
- Milk和Suger是具体的装饰者
- Coffee是被装饰者主体
- LongBlack和Espresso是具体实现的被装饰者实体

抽象类

```
public abstract class Drink {//抽象类
    public String des; // 描述
    private float price = 0.0f;
    public String getDes() {
        return des;
    }
    public void setDes(String des) {
        this.des = des;
    }
    public float getPrice() {
        return price;
    }
    public void setPrice(float price) {
        this.price = price;
    }
    //计算费用的抽象方法
    public abstract float cost();
}
```

装饰者

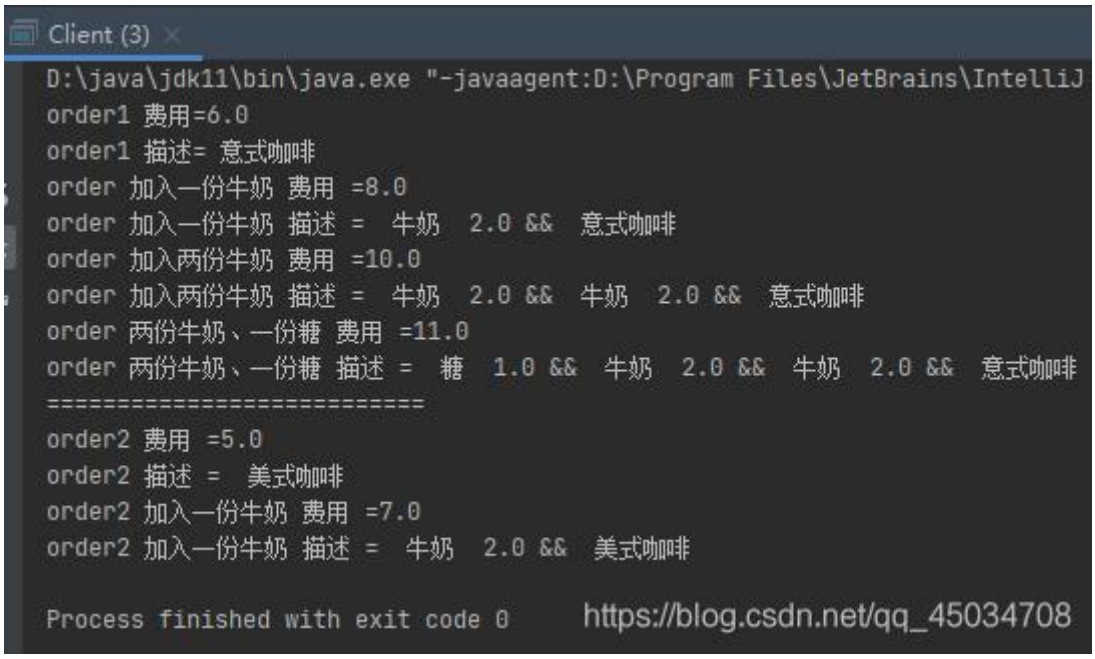
```
public class Decorator extends Drink {//装饰者
    private Drink obj;
    public Decorator(Drink obj) { //组合
        this.obj = obj;
    }
    @Override
    public float cost() {
        // getPrice 自己价格
        return super.getPrice() + obj.cost();
    }
    @Override
    public String getDes() {
        // obj.getDes() 输出被装饰者的信息
        return des + " " + getPrice() + " && " + obj.getDes();
    }
}

public class Milk extends Decorator {//装饰者子类
    public Milk(Drink obj) {
        super(obj);
        setDes(" 牛奶 ");
        setPrice(2.0f);
    }
}

public class Suger extends Decorator {//装饰者子类
    public Suger(Drink obj) {
        super(obj);
        setDes(" 糖 ");
        setPrice(1.0f);
    }
}
```

被装饰者

```
public class Coffee  extends Drink {//被装饰者
    @Override
    public float cost() {
        return super.getPrice();
    }
}

public class Espresso extends Coffee {//被装饰者子类
    public Espresso() {
        setDes(" 意式咖啡 ");
        setPrice(6.0f);
    }
}

public class LongBlack extends Coffee {//被装饰者子类
    public LongBlack() {
        setDes(" 美式咖啡 ");
        setPrice(5.0f);
    }
}
```

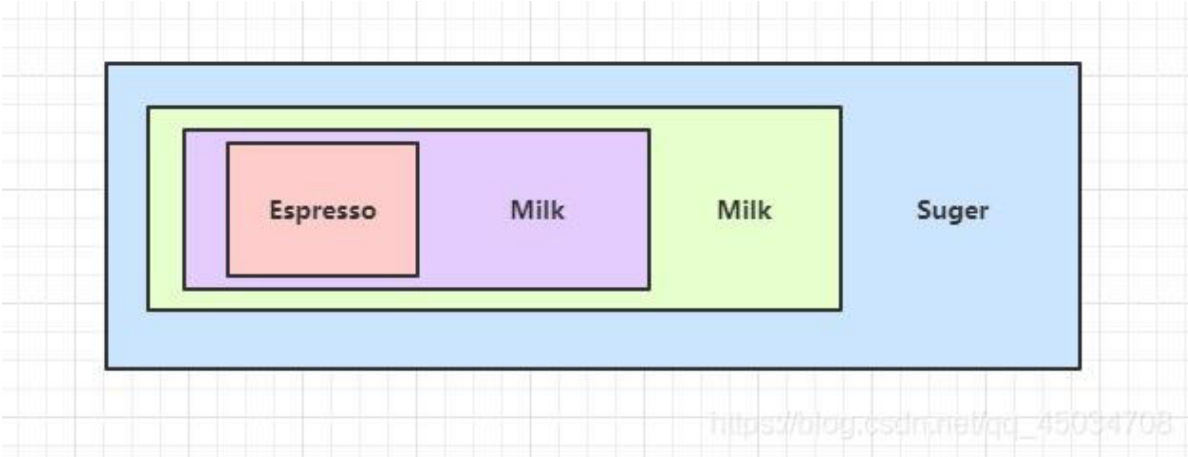
客户端测试

```
public class Client {
    public static void main(String[] args) {
        // 阿姨的卡布奇诺：意式加两份牛奶、一份糖
        // 1. 点一份Espresso
        Drink order = new Espresso();
        System.out.println("order1 费用=" + order.cost());
        System.out.println("order1 描述=" + order.getDes());
        // 2.1 order 加一份牛奶
        order = new Milk(order);
        System.out.println("order 加入一份牛奶 费用 =" + order.cost());
        System.out.println("order 加入一份牛奶 描述 = " + order.getDes());
        // 2.2 order 再加一份牛奶
        order = new Milk(order);
        System.out.println("order 加入两份牛奶 费用 =" + order.cost());
        System.out.println("order 加入两份牛奶 描述 = " + order.getDes());
        // 3. order 加一份糖
        order = new Suger(order);
        System.out.println("order 两份牛奶、一份糖 费用 =" + order.cost());
        System.out.println("order 两份牛奶、一份糖 描述 = " + order.getDes());

        System.out.println("==========================");
        //美式咖啡加一份牛奶
        //1. 点一份LongBlack
        Drink order2 = new LongBlack();
        System.out.println("order2 费用 =" + order2.cost());
        System.out.println("order2 描述 = " + order2.getDes());
        //2. order2 加一份牛奶
        order2 = new Milk(order2);
        System.out.println("order2 加入一份牛奶 费用 =" + order2.cost());
        System.out.println("order2 加入一份牛奶 描述 = " + order2.getDes());
    }
}
```



小结

装饰者模式就像打包一个快递，不断的动态添加新的功能，可以组合出所有情况：



十一、外观模式

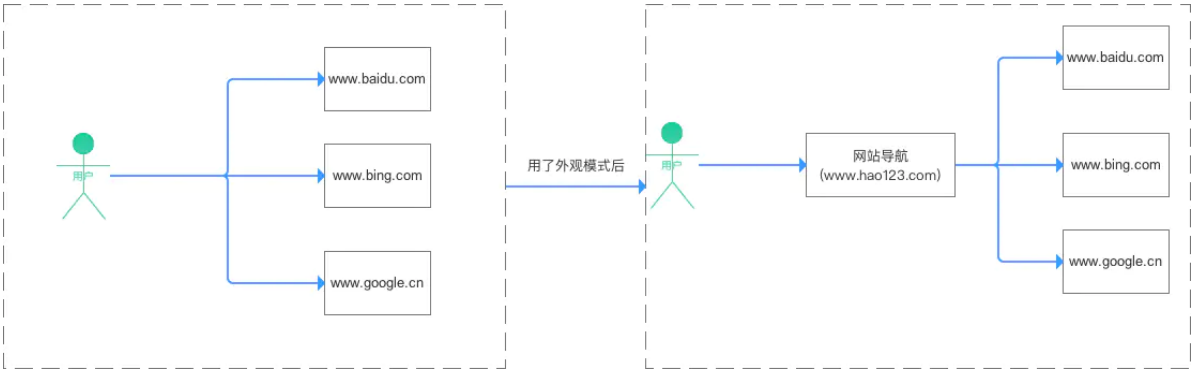
1. 介绍

1.1 定义

定义了一个高层、统一的接口，外部与通过这个统一的接口对子系统中的一群接口进行访问。

> 通过创建一个统一的外观类，用来包装子系统中一个 / 多个复杂的类，客户端可通过调用外观类的方法来调用内部子系统中所有方法

以前我需要在搜索栏逐个搜索网站地址；有了网站导航（用了外观模式）后，就方便很多了



1.2 主要作用

- 实现客户类与子系统类的松耦合
- 降低原有系统的复杂度
- 提高了客户端使用的便捷性，使得客户端无须关心子系统的工作细节，通过外观角色即可调用相关功能。

> 引入外观角色之后，用户只需要与外观角色交互；用户与子系统之间的复杂逻辑关系由外观角色来实现

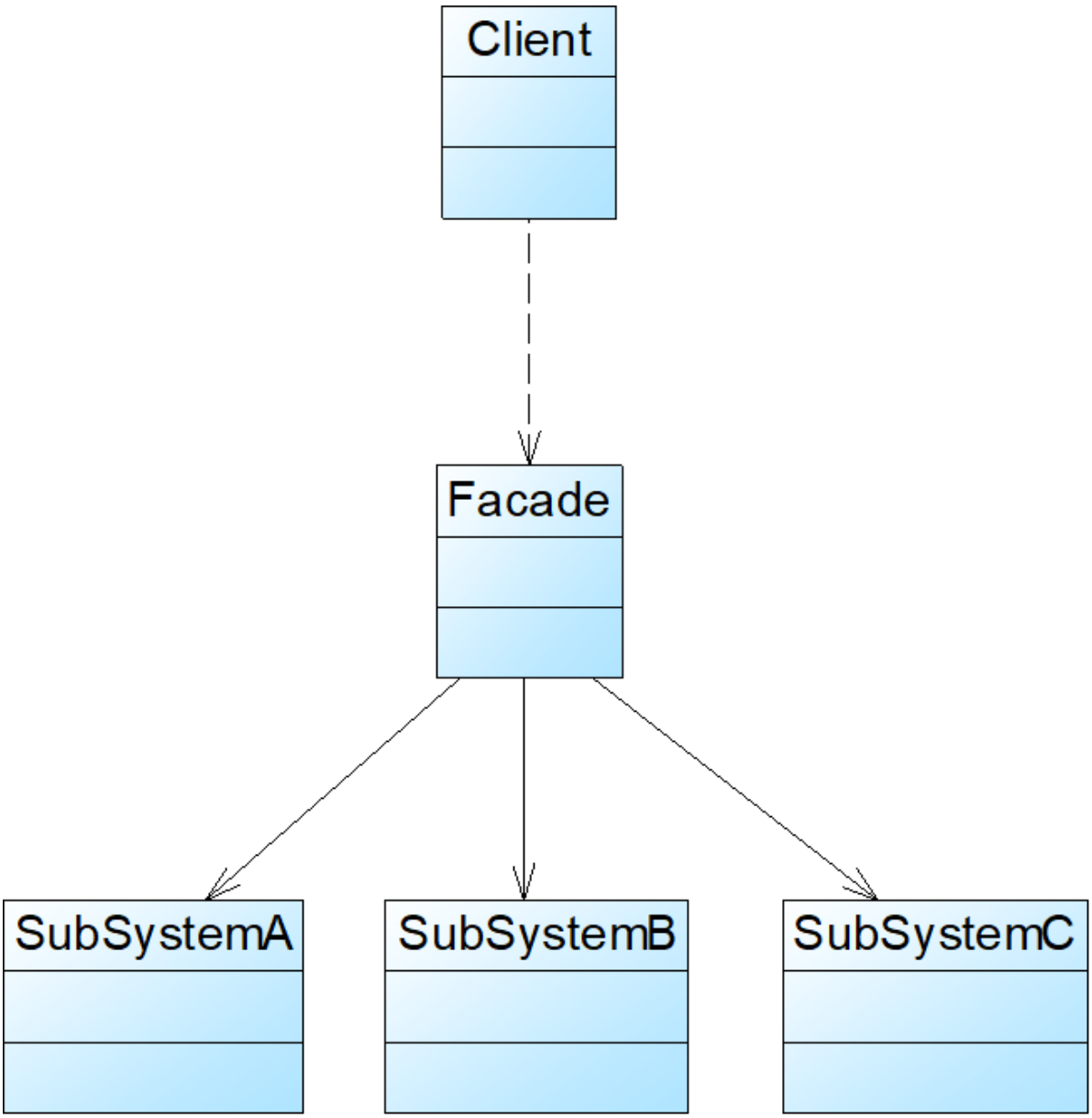
1.3 解决的问题

- 避免了系统与系统之间的高耦合度
- 使得复杂的子系统用法变得简单

---

2. 模式原理

2.1 UML



2.2 实例讲解

接下来我用一个实例来对建造者模式进行更深一步的介绍。**a. 实例概况**

- 背景：小成的爷爷已经80岁了，一个人在家生活：每次都需要打开灯、打开电视、打开空调；睡觉时关闭灯、关闭电视、关闭空调；
- 冲突：行动不方便，走过去关闭那么多电器很麻烦，代码如下：

电器类：

```
//灯类
public class SubSystemA_Light {
    public void on(){
        System.out.println("打开了灯....");
    }

    public void off(){
        System.out.println("关闭了灯....");
    }
}

//电视类
public class SubSystemB_Television {
```

```
    public void on(){
        System.out.println("打开了电视....");
    }

    public void off(){
        System.out.println("关闭了电视....");
    }
}

//空调类
public class SubSystemC_Aircondition {
    public void on(){
        System.out.println("打开了电视....");
    }

    public void off(){
        System.out.println("关闭了电视....");
    }
}
```

外观类：智能遥控器

```
public class Facade {

    SubSystemA_Light light;
    SubSystemB_Television television ;
    SubSystemC_Aircondition aircondition;

    //传参
    public Facade(SubSystemA_Light light, SubSystemB_Television television,
SubSystemC_Aircondition aircondition) {
        this.light = light;
        this.television = television;
        this.aircondition = aircondition;

    }

    //起床后一键开电器
    public void on

    {
        System.out.prinln("起床了");
        light.on();
        television.on();
        aircondition.on();

    }

    //睡觉时一键关电器
          System.out.prinln("睡觉了");
                  light.off();
                  television.off();
          aircondition.off();
        }
}
```

客户端调用：爷爷使用智能遥控器的时候

```
public class Facade Pattern{
      public static void main(String[] args){
        {
            //实例化电器类
            SubSystemA_Light light = new SubSystemA_Light();
            SubSystemB_Television television = new SubSystemB_Television();
            SubSystemC_Aircondition aircondition = new
SubSystemC_Aircondition();

            //传参
            Facade facade = new Facade(light,television,aircondition);

            //客户端直接与外观对象进行交互
            facade.on;
            System.out.prinln("可以看电视了");
            facade.off;
            System.out.prinln("可以睡觉了");
}
}
```

结果

```
起床了
打开了灯
打开了电视
打开了空调
可以看电视了

睡觉了
关闭了灯
关闭了电视
关闭了空调
可以睡觉了
```

与适配器模式的区别

- 外观模式的实现核心主要是——由外观类去保存各个子系统的引用，实现由一个统一的外观类去包装多个子系统类，然而客户端只需要引用这个外观类，然后由外观类来调用各个子系统中的方法。
- 这样的实现方式非常类似适配器模式，然而外观模式与适配器模式不同的是：**适配器模式是将一个对象包装起来以改变其接口，而外观是将一群对象“包装”起来以简化其接口。它们的意图是不一样的，适配器是将接口转换为不同接口，而外观模式是提供一个统一的接口来简化接口。**

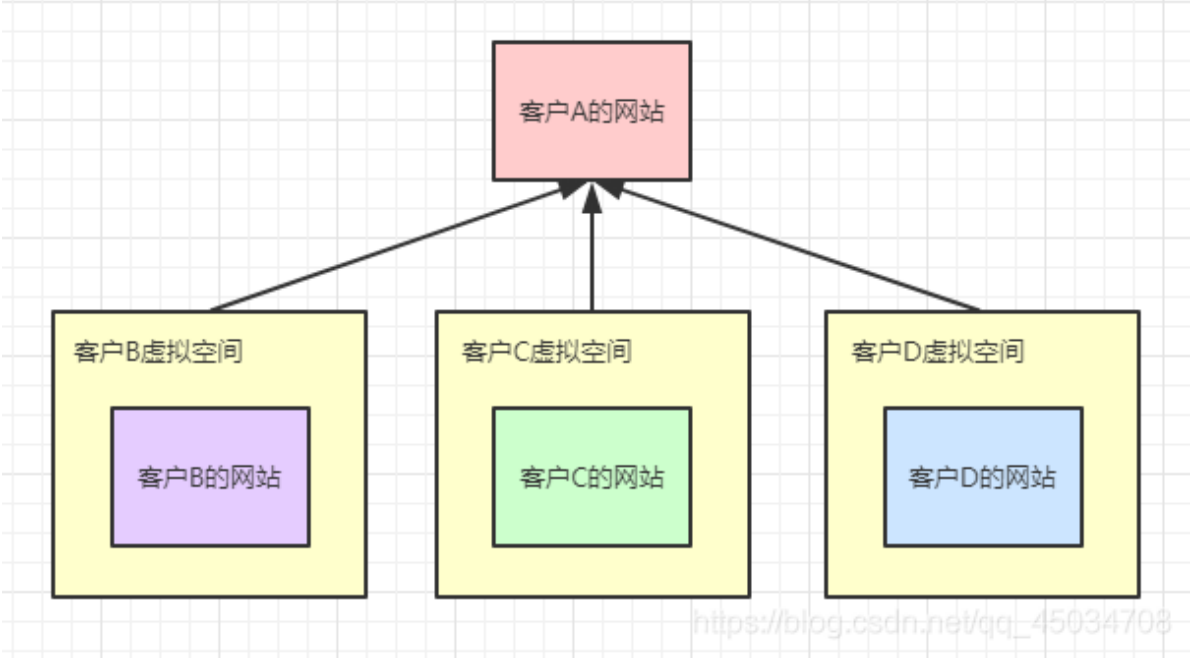
十二、享元模式

场景：现有一外包公司，帮客户A做了一个产品展示网站，网站做好后更多客户觉得效果不错，也希望做个类似网站，但不同的是有客户要求以新闻形式发布、有客户要求以微信公众号形式发布、有客户希望以博客形式发布。合理设计达到代码复用，灵活易维护扩展。

1、一般解法

直接复制粘贴一份，然后再根据客户不同要求，进行定制修改，给每一个网站租用了一个空间。

示意图如下：



问题分析： 首先需要的网站结构相似度很高（设普通网站，非高访问大并发），如果分成多个虚拟空间来处理，相当于一个相同网站的实例对象很多，造成服务器的资源浪费。

解决思路： 全部整合到一个网站中，共享其相关的代码和数据，对于硬盘、内存、CPU、数据库空间等服务器资源都可以达成共享，减少服务器资源。对于代码来说，由于是一份实例，维护和扩展都更加容易=》享元模式。

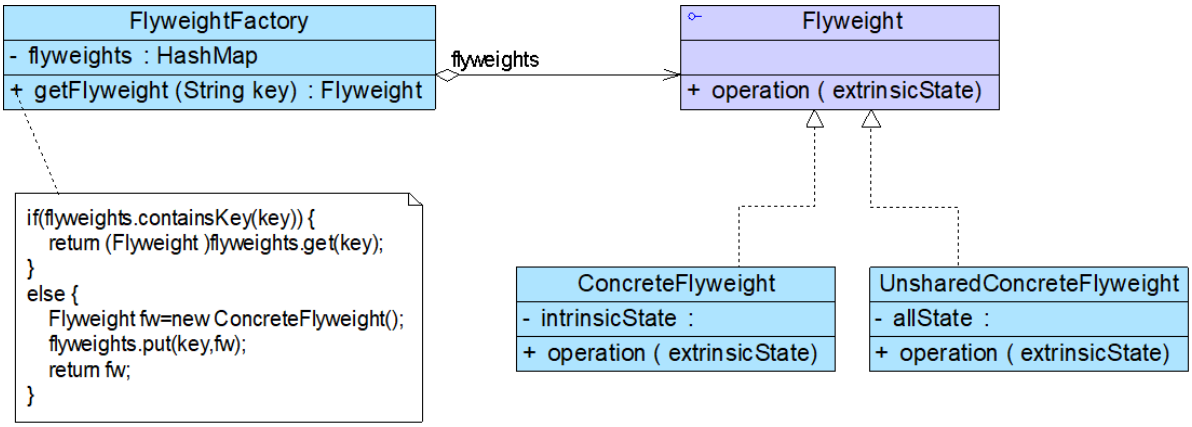
2、享元模式

享元模式（Flyweight Pattern） 也叫蝇量模式，是一种结构型模式，“享”就表示共享，“元”表示对象。运用共享技术有效地支持大量细粒度的对象，享元模式能够解决重复对象的内存浪费的问题，当系统中有大量相似对象，需要缓冲池时，不需总是创建新对象，可以从缓冲池里拿。这样可以降低系统内存，同时提高效率。

常用于系统底层开发，解决系统的性能问题。像数据库连接池，里面都是创建好的连接对象，在这些连接对象中有我们需要的则直接拿来用，避免重新创建，如果没有我们需要的，则创建一个。

享元模式经典的应用场景就是池技术了，String常量池、数据库连接池、缓冲池等等都是享元模式的应用，享元模式是池技术的重要实现方式。

3、UML



- FlyWeight抽象类：享元角色，定义出对象的外部状态和内部状态（下面说明）的接口或实现。
- ConcreteFlyWeight：是具体的享元角色，是具体的产品类，实现抽象角色定义相关业务。
- UnSharedConcreteFlyWeight：是不可共享的角色，一般不会出现在享元工厂。
- FlyWeightFactory享元工厂类：用于构建一一个池容器(集合)，同时提供从池中获取对象方法。

4、代码

抽象享元类

```
public abstract class Flyweight {
    public abstract void operation(String extrinsicState);
}
```

具体享元类

```
public class ConcreteFlyweight extends Flyweight {
    //内部状态intrinsicState作为成员变量，同一个享元对象其内部状态是一致的
    private String intrinsicState;
    public ConcreteFlyweight(String intrinsicState) {
        this.intrinsicState = intrinsicState;
    }

    //外部状态extrinsicState在使用时由外部设置，不保存在享元对象中，即使是同一个对象，在每一次调用时可以传入不同的外部状态
    public void operation(String extrinsicState) {
        //实现业务方法
    }
}
```

非共享具体享元类

```
public class UnsharedConcreteFlyweight extends Flyweight {
    public void operation(String extrinsicState) {
        //实现业务方法
    }
}
```

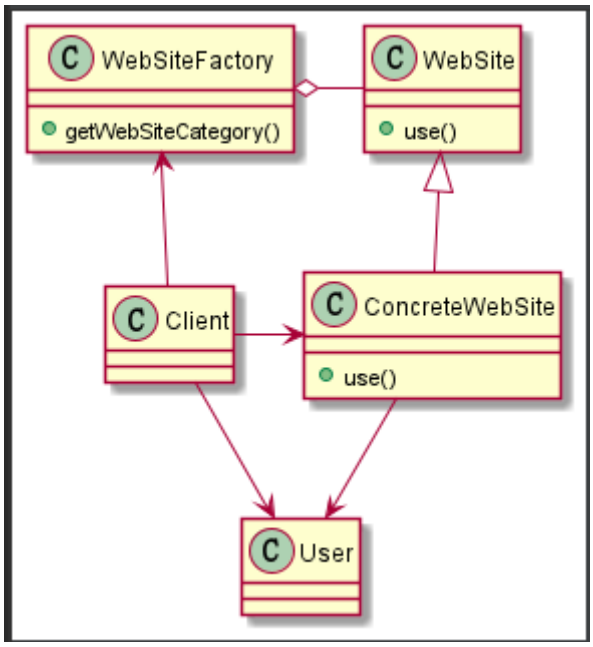
享元工厂类

```
public class FlyweightFactory {
    //定义一个HashMap用于存储享元对象，实现享元池
    private HashMap flyweights = new HashMap();

    public Flyweight getFlyweight(String key) {
        //如果对象存在，则直接从享元池获取
        if (flyweights.containsKey(key)) {
            return (Flyweight)flyweights.get(key);
        }
        //如果对象不存在，先创建一个新的对象添加到享元池中，然后返回
        else {
            Flyweight fw = new ConcreteFlyweight();
            flyweights.put(key,fw);
            return fw;
        }
    }
}
```

使用享元模式来解决引例需求。

类图

User就是享元模式中的外部状态。



抽象类

```
public abstract class WebSite {
    public abstract void use(User user);//抽象方法
}
```

抽象实现子类

```
public class ConcreteWebSite extends WebSite {//具体网站
    //共享的部分，内部状态
    private String type = ""; //网站发布的形式(类型)
    //构造器
    public ConcreteWebSite(String type) {

        this.type = type;
    }
    @Override
    public void use(User user) {
        // TODO Auto-generated method stub
        System.out.println("网站的发布形式为:" + type + " 在使用中 .. 使用者是" + user.getName());
    }
}
```

享元工厂类

```
public class WebSiteFactory {//根据需要返回压一个网站
    //集合， 充当池的作用
    private HashMap<String, ConcreteWebSite> pool = new HashMap<>();

    //根据网站的类型，返回一个网站， 如果没有就创建一个网站，并放入到池中,并返回
    public WebSite getWebSiteCategory(String type) {
        if(!pool.containsKey(type)) {
            //就创建一个网站，并放入到池中
            pool.put(type, new ConcreteWebSite(type));
        }
```

```
        return (WebSite)pool.get(type);
    }
    //获取网站分类的总数 (池中有多少个网站类型)
    public int getWebSiteCount() {
        return pool.size();
    }
}
```

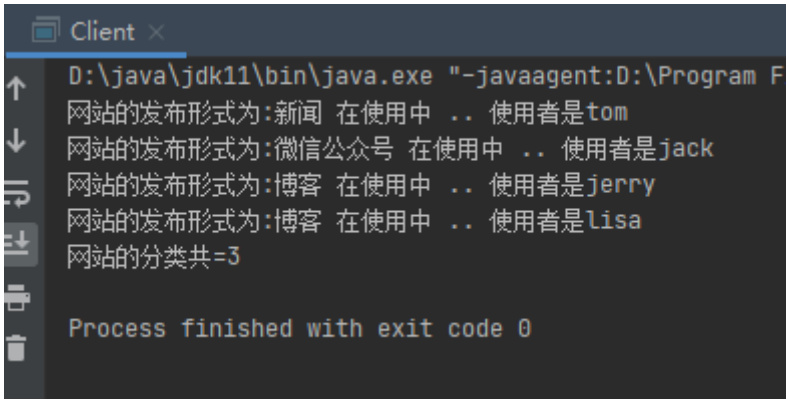
不可共享类

```
public class User { //外部状态
    private String name;
    public User(String name) {
        super();
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

客户端调用

```
public class Client {
    public static void main(String[] args) {
        // 创建一个工厂类
        WebSiteFactory factory = new WebSiteFactory();

        // 客户要一个以新闻形式发布的网站
        WebSite webSite1 = factory.getWebSiteCategory("新闻");
        webSite1.use(new User("tom"));

        // 客户要一个以微信公众号形式发布的网站
        WebSite webSite2 = factory.getWebSiteCategory("微信公众号");
        webSite2.use(new User("jack"));

        // 客户要一个以博客形式发布的网站
        WebSite webSite3 = factory.getWebSiteCategory("博客");
        webSite3.use(new User("jerry"));

        // 客户要一个以博客形式发布的网站
        WebSite webSite4 = factory.getWebSiteCategory("博客");
        webSite4.use(new User("lisa"));

        System.out.println("网站的分类共=" + factory.getWebSiteCount());
    }
}
```

运行结果:



十三、代理模式

静态代理模式（Proxy Pattern）

1. 介绍

1.1 定义

给目标对象提供一个代理对象，并由代理对象控制对目标对象的引用

> 代理对象：起到中介作用，连接客户端和目标对象例子：电脑桌面的快捷方式。电脑对某个程序提供一个快捷方式（代理对象），快捷方式连接客户端和程序，客户端通过操作快捷方式就可以操作那个程序

1.2 主要作用

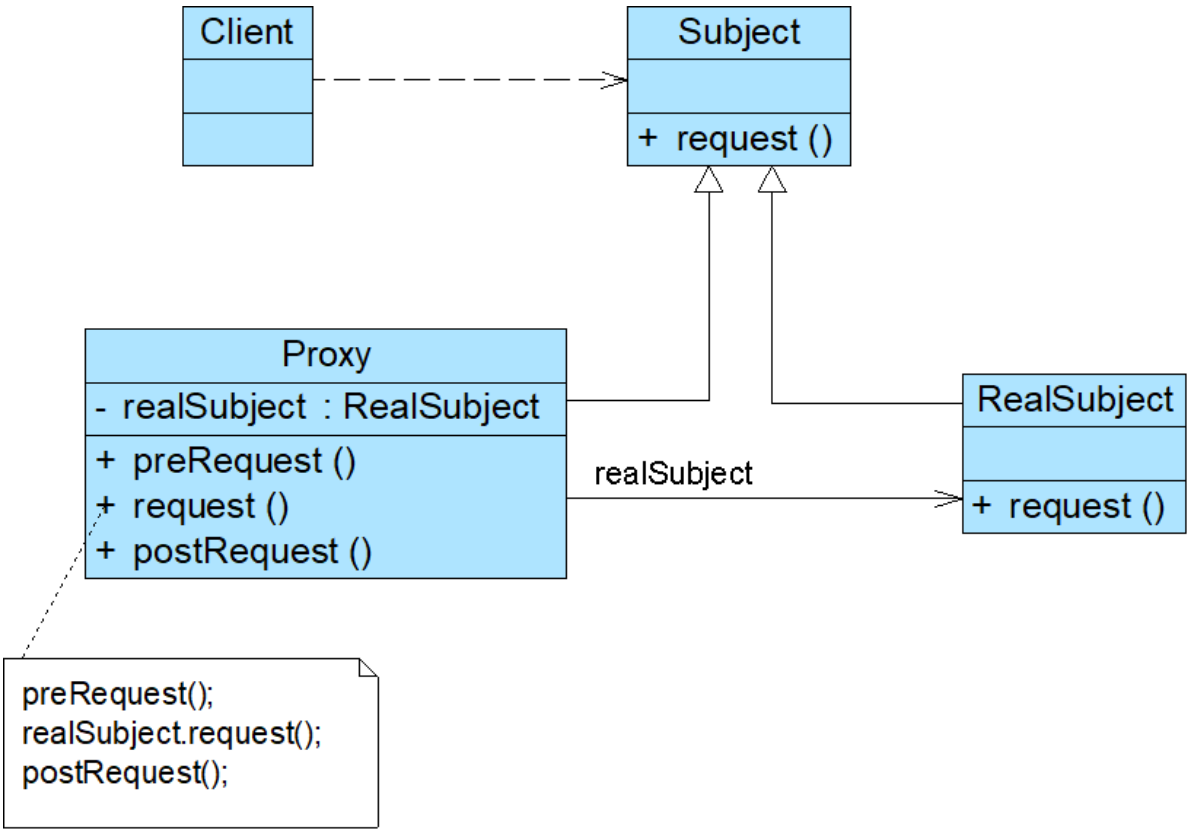
通过引入**代理对象**的方式来间接访问**目标对象**

1.3 解决的问题

防止**直接**访问目标对象给系统带来的不必要复杂性。

2. 模式原理

2.1 UML



2.2 代码

抽象主题类

```
public abstract class Subject {
    public abstract void request();
}
```

真实主题类

```
public class RealSubject extends Subject{
    public void request() {
        //业务方法具体实现代码
    }
}
```

代理类

```
public class Proxy extends Subject {
    private RealSubject realSubject = new RealSubject();  //维持一个对真实主题对象的引用
    public void preRequest() {
        ……...
    }

    public void request() {
        preRequest();
        realSubject.request();  //调用真实主题对象的方法
        postRequest();
    }

    public void postRequest() {
        ……
```

```
    }
}
```

2.2 实例讲解

接下来我用一个实例来对代理模式进行更深一步的介绍。

a. 实例概况

- 背景：小成希望买一台最新的顶配Mac电脑
- 冲突：国内还没上，只有美国才有
- 解决方案：寻找代购进行购买

> 代购（代理对象） 代替 我（真实对象） 去买Mac（间接访问的操作）

b. 使用步骤

步骤1： 创建抽象对象接口（Subject）：声明你（真实对象）需要让代购（代理对象）帮忙做的事（买Mac)

```
public interface Subject {
                public void buyMac();
}
```

步骤2： 创建真实对象类（RealSubject）,即”我“

```
public class RealSubject implement Subject{
    @Override
    public void buyMac() {
        System.out.println(”买一台Mac“);
    }
}
```

步骤3：创建代理对象类（Proxy），即”代购“，并通过代理类创建真实对象实例并访问其方法

```
public class Proxy  implements Subject{

    @Override
    public void buyMac{

      //引用并创建真实对象实例，即”我“
      RealSubject realSubject = new RealSubject();

      //调用真实对象的方法，进行代理购买Mac
      realSubject.buyMac（）；
      //代理对象额外做的操作
      this.WrapMac();
    }

     public void WrapMac(){
      System.out.println(”用盒子包装好Mac“);
    }
}
```

步骤4：客户端调用

```
public class ProxyPattern {

    public static void main(String[] args){

    Subject proxy = new Proxy () ;
    proxy.buyMac();
    }

}
```

结果输出

```
买一台Mac
用盒子包装好Mac
```