

设计模式

1.策略模式+工厂模式

(1). 策略模式

定义一系列的算法,把每一个算法封装起来, 并且使它们可相互替换

策略模式把对象本身和运算规则区分开来, 因此我们整个模式也分为三个部分。

环境类(Context):用来操作策略的上下文环境, 也就是我们游客。抽象策略类(Strategy):策略的抽象, 出行方式的抽象具体策略类(ConcreteStrategy):具体的策略实现, 每一种出行方式的具体实现。

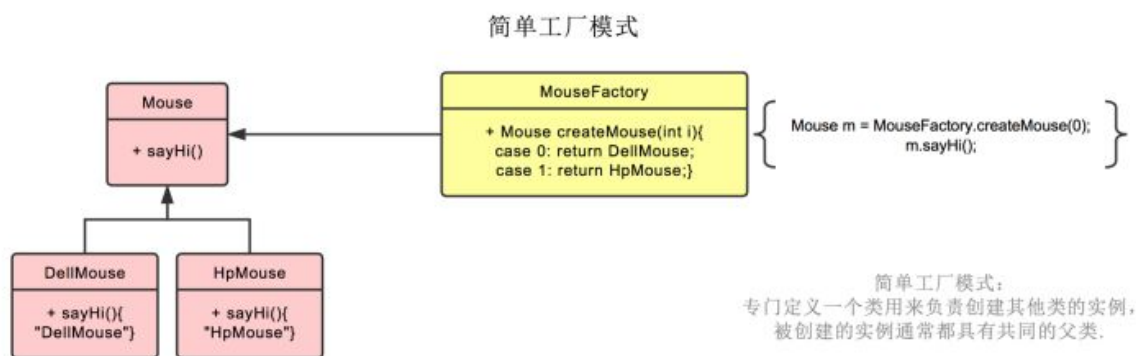
个人理解, 策略方式可以把client 层面和处理层面分开, 对于client的层面, 只需要告诉要进行的行为的名字就可以了, 比如在使用优惠券时进行八五折, 那么就输入八五折相关规定好的字眼就可以了, 不需要了解后面的进行操作是怎么样。策略者模式通常用于各种不同的选择, 在优惠券中就可以是1折, 2折, 各种折扣等。

(2).工厂模式

作用:

1. 当调用者想创建一个对象时, 只需要知道名称就可以在工厂获取具体的对象。
2. 扩展性较强, 当想增加一个类时, 在工厂中进行增加即可。
3. 调用者可以只关注到产品的接口, 不需要了解内部的实现。

一、简单工厂模式



只有一个工厂来创建其他类的实例, 在这种情况下是一个项目比较简单的时候可以使用, 像dao层中便可以使用一个工厂来创建实例。

在图上的例子便是将鼠标分为了两种类型, 一种是惠普的鼠标, 一种是戴尔的鼠标。

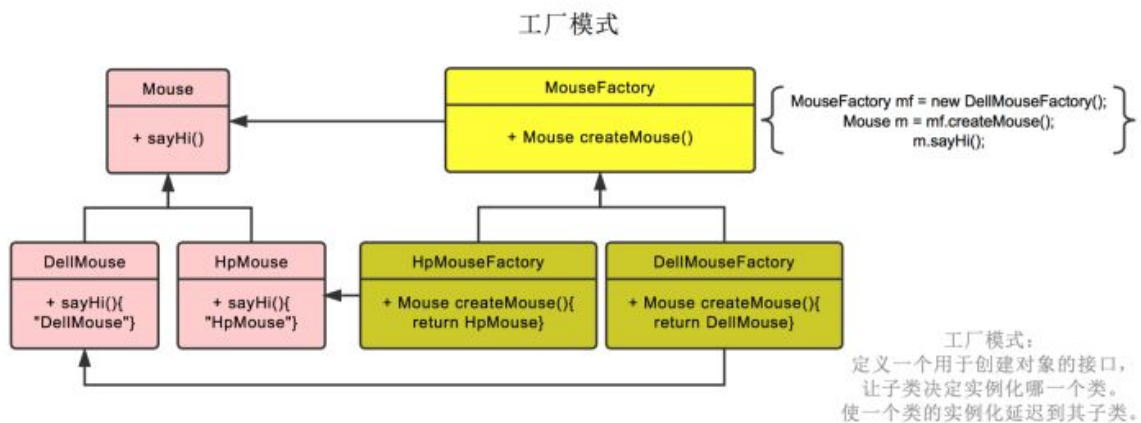
```
1 public class MouseFactory {
2     private static Map<String, Mouse> mouseMap = new HashMap<>();
3     static {
4
5         mouseMap.put("Dell", new DellMouse());
6
7         mouseMap.put("Hp", new HpMouse());
8
9     }
```

```

10     public static Mouse getMouse(String mouseType) {
11
12         return mouseMap.get(mouseType);
13     }
14 }
15 //工厂类与策略模式进行结合
16 public class Context {
17     private Mouse mouse;
18
19     public Context(Mouse mouse){
20         this.mouse = mouse;
21     }
22
23     public String executeStrategy(){
24         return mouse.createMouse();
25     }
26 }
27
28 public class StrategyPattern {
29     public static void main(String[] args) {
30         //这一部是工厂来的
31         Mouse mouse = Factory.getMouse("Hp");
32         Context context = new Context(mouse);
33         System.out.println(context.executeStrategy());
34     }
35 }

```

二、工厂模式



工厂模式也就是鼠标工厂是个父类，有生产鼠标这个接口。

得力鼠标工厂，惠普鼠标工厂继承它，可以分别生产戴尔鼠标，惠普鼠标。
生产哪种鼠标不再由参数决定，而是创建鼠标工厂时，由戴尔鼠标工厂创建。
后续直接调用鼠标工厂.生产鼠标()即可

```

1 //父类工厂类
2 public class MouseFactory{
3     private static Map<String, Mouse> mouseMap = new HashMap<>();
4     static {
5
6         mouseMap.put("Dell", new DellMouseFactory());
7
8         mouseMap.put("Hp", new HpMouseFactory());
9     }
10 }

```

```

9
10     }
11     public static MouseFactory getMouse(String mouseType) {
12
13         return mouseMap.get(mouseType);
14     }
15 }
16 //子类工厂类 惠普鼠标
17 public class HpMouseFactory extends MouseFactory{
18     public static Mouse createMouse(){
19         return new HpMouse();
20     }
21 }
22 //戴尔鼠标
23 public class DellMouseFactory extends MouseFactory{
24     public static Mouse createMouse(){
25         return new DeliMouse();
26     }
27 }
28 //接口
29 public interface Mouse{
30     void sayHi();
31 }
32 public class DellMouse extends Mouse{
33     public void sayHi(){
34         System.out.println("Deli hello");
35     }
36 }
37
38 public class HpMouse extends Mouse{
39     public void sayHi(){
40         System.out.println("Hp hello");
41     }
42 }

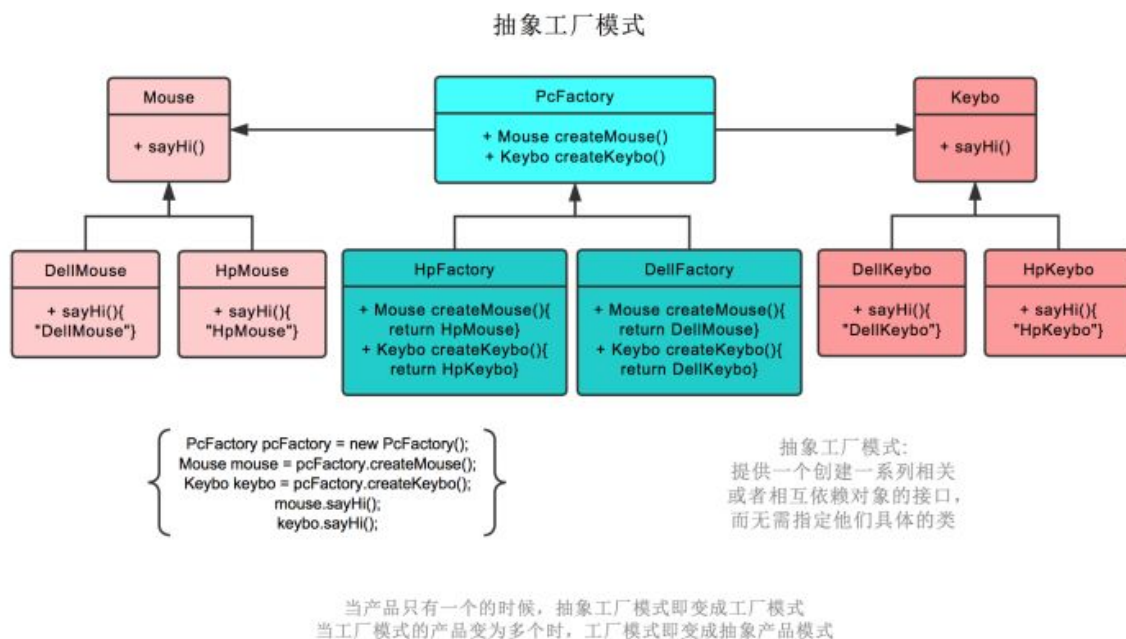
```

```

1 //策略者模式结合
2 public class Context {
3     private Mouse mouse;
4
5     public Context(Mouse mouse){
6         this.mouse = mouse;
7     }
8
9     public String executeStrategy(){
10         return mouse.sayHi();
11     }
12 }
13 public class StrategyPattern {
14     public static void main(String[] args) {
15         //这一部是工厂来的
16         MouseFactory mouseFactory = MouseFactory.getMouse("Hp");
17         Mouse mouse = mouseFactory.getMouse("Hp");
18         Context context = new Context(mouse);
19         System.out.println(context.executeStrategy());
20     }
21 }
22

```

三、抽象工厂模式



抽象工厂的实现更为复杂，父类的范围也越大。在抽象工厂模式中，假设我们需要增加一个工厂假设我们增加华硕工厂，则我们需要增加华硕工厂，和戴尔工厂一样，继承PC厂商。之后创建华硕鼠标，继承鼠标类。创建华硕键盘，继承键盘类。即可。

2. 观察者模式

观察者模式 (Observer)，又叫**发布-订阅模式 (Publish/Subscribe)**，定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

(1)、优点

- 观察者和被观察者是抽象耦合的
- 建立了一套触发机制

(4) . 缺点

- 如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间
- 如果观察者和观察目标间有循环依赖，可能导致系统崩溃
- 没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的

代码实现

在这里是运用了一个b站粉丝关注和告知粉丝视频发布的简单例子。

```

1 //粉丝就是观察者
2 public interface Observer {
3     //更新
4     public void update();
5     //关注up主
6     public void subscribeChannel(Channel ch);
7     //取消关注

```

```

8      public void unsubscribe(Channel ch,Subscriber subscriber);
9  }
10
11  public class Subscriber implements Observer{
12      private String name;
13      private List<Channel> channelList = new ArrayList<>();
14      private Subscriber subscriber;
15      //当up主更新时，会执行该方法
16      @Override
17      public void update(){
18          System.out.println(name + ",你关注的up主发布了视频");
19      }
20      //将关注的人放在list中
21      @Override
22      public void subscribeChannel(Channel ch){
23          channelList.add(ch);
24      }
25
26      public Subscriber(String name) {
27          this.name = name;
28      }
29
30      //取消关注up主
31      @Override
32      public void unsubscribe(Channel ch, Subscriber subscriber){
33          this.subscriber = subscriber;
34          channelList.remove(ch);
35          ch.unsubscribe(subscriber);
36          System.out.println(name + "你已成功取消关注");
37      }
38  }
39

```

```

1  public interface Subject {
2      void subscribe(Subscriber sub);
3      public void unsubscribe(Subscriber sub);
4      public void notifySubscriber();
5      public void upload(String title);
6
7  }
8
9  public class Channel implements Subject{
10      List<Subscriber> subs = new ArrayList<>();
11      private String title;
12
13      @Override
14      public void subscribe(Subscriber sub){
15          //增加关注者
16          subs.add(sub);
17      }
18
19      @Override
20      public void unsubscribe(Subscriber sub){
21          subs.remove(sub);
22      }
23      @Override
24      public void notifySubscriber(){

```

```

25         //通知到每个关注的人
26         for(Subscriber sub : subs){
27             sub.update();
28         }
29     }
30
31     @Override
32     public void upload(String title){
33         this.title = title;
34         notifySubscriber();
35     }
36
37
38 }
39

```

```

1  public class Bilibili {
2      public static void main(String[] args) {
3          Channel javaLearning = new Channel();
4          //两个关注者
5          Subscriber s1 = new Subscriber("Ivy");
6          Subscriber s2 = new Subscriber("Tom");
7
8          javaLearning.subscribe(s1);
9          javaLearning.subscribe(s2);
10         //关注了javaLearning up主
11         s1.subscribeChannel(javaLearning);
12         s2.subscribeChannel(javaLearning);
13         //up主上传了名为如何学java的视频
14         javaLearning.upload("How to learn java");
15
16         s1.unsubscribe(javaLearning,s1);
17
18         javaLearning.upload("快速入门Javaweb");
19     }
20 }

```

3. 适配器模式

适配器模式的内容比较简单，举个例子，像在每个国家使用的电压和插头都是不一样的，在中国是一种，在国外又是一种，所以当我们出国要给一些设备充电时，就需要用到适配器，他是一个中介，能把我们平常用到的接口转成该国家常使用的接口进行充电。所以基本原理就是这样。

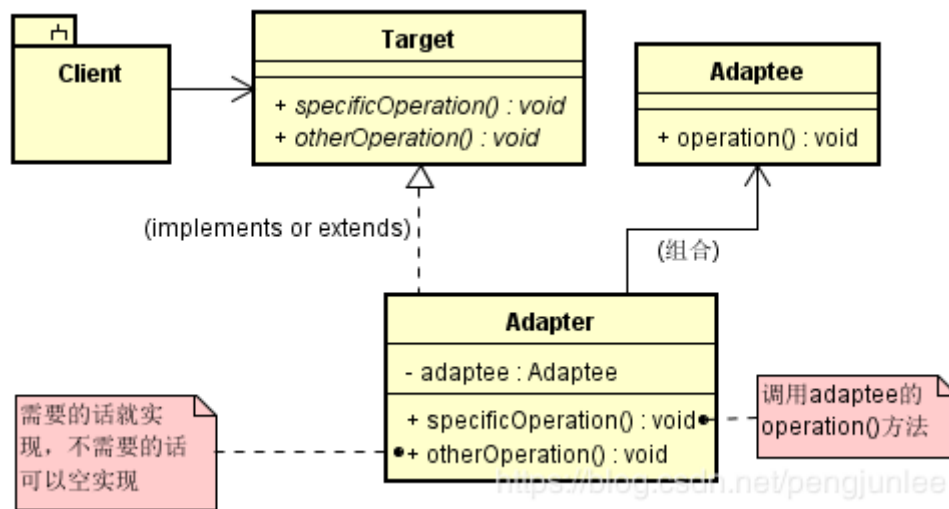
在如下举了一个苹果充电线和安卓充电线的例子，要让苹果手机通过转换器用安卓数据线充上电。

Adaptee: 被适配的类。在下面的例子便是指**苹果手机线**

Adapter: 该类对适配器类和目标接口类进行适配，在对象适配器模式下通过组合方式实现，即：Adapter类继承Target类或者实现Target接口，并在其内部包含一个Adaptee对象的引用，通过对其内部的Adaptee对象的调用实现客户端所需要的接口。 在下面的例子便是

LightningToMicroUsbAdapter

目标(Target)接口类: 客户端所需要的接口。在下面的例子便是安卓手机线



```

1  //苹果手机线的接口
2  interface LightningPhone {
3      //充电
4      void recharge();
5      //使用充电线
6      void useLightning();
7  }
8  //安卓手机线的接口
9  interface MicroUsbPhone {
10     //充电
11     void recharge();
12     //使用充电线
13     void useMicroUsb();
14 }
15 //苹果手机
16 class Iphone implements LightningPhone {
17     private boolean connector;
18
19     @Override
20     public void useLightning() {
21         connector = true;
22         System.out.println("Lightning connected");
23     }
24
25     @Override
26     public void recharge() {
27         if (connector) {
28             System.out.println("Recharge started");
29             System.out.println("Recharge finished");
30         } else {
31             System.out.println("Connect Lightning first");
32         }
33     }
34 }
35 //安卓手机
36 class Android implements MicroUsbPhone {
37     private boolean connector;
38
39     @Override
40     public void useMicroUsb() {
41         connector = true;
42         System.out.println("MicroUsb connected");

```

```

43     }
44
45     @Override
46     public void recharge() {
47         if (connector) {
48             System.out.println("Recharge started");
49             System.out.println("Recharge finished");
50         } else {
51             System.out.println("Connect Microusb first");
52         }
53     }
54 }
55 //适配器，作为一个中介将适配器类和目标接口类进行适配，实现目标接口类
56 /* exposing the target interface while wrapping source object */
57 class LightningToMicroUsbAdapter implements MicroUsbPhone {
58     //内部对Adaptee 对象的引用
59     private final LightningPhone lightningPhone;
60
61     public MicroUsbToLightningAdapter(LightningPhone lightningPhone) {
62         this.lightningPhone = lightningPhone;
63     }
64
65     @Override
66     public void useMicroUsb() {
67         System.out.println("MicroUsb connected");
68         lightningPhone.useLightning();
69     }
70
71     @Override
72     public void recharge() {
73         lightningPhone.recharge();
74     }
75 }
76 //客户端
77 public class AdapterDemo {
78     static void rechargeMicroUsbPhone(MicroUsbPhone phone) {
79         phone.useMicroUsb();
80         phone.recharge();
81     }
82
83     static void rechargeLightningPhone(LightningPhone phone) {
84         phone.useLightning();
85         phone.recharge();
86     }
87
88     public static void main(String[] args) {
89         Android android = new Android();
90         Iphone iPhone = new Iphone();
91
92         System.out.println("Recharging android with Microusb");
93         rechargeMicroUsbPhone(android);
94
95         System.out.println("Recharging iPhone with Lightning");
96         rechargeLightningPhone(iPhone);
97
98         System.out.println("Recharging iPhone with MicroUsb");
99         rechargeMicroUsbPhone(new LightningToMicroUsbAdapter (iPhone));
100    }

```



```
101 | }
102 |
```

作用：

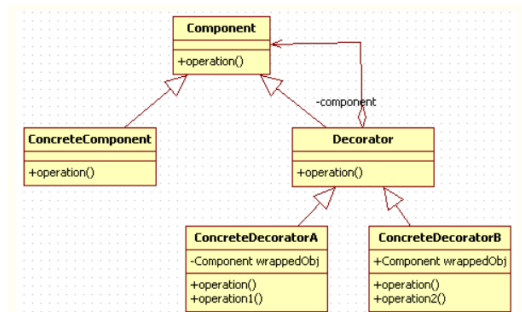
1. 将目标类和适配者类解耦，通过引入一个适配器类重用现有的适配者类，而无需修改原有代码。
2. 一个对象适配器可以把适配者类和它的子类都适配到目标接口。

4.装饰器模式

装饰器的核心便是进行包装，像是一个层层包装。从一个最基础物品，通过包装装饰可以获得一个升级版的物品。

举一个生活中贴近的例子，比如说去奶茶店买奶茶，你可以选择一个基础的商品珍珠奶茶，然后进行甜度的调整，配料的添加，加珍珠，芋圆等，都可以进行选择，然后价格也随之叠加，但本质上还是一个珍珠奶茶。

装饰器模式，顾名思义，就是对已经存在的某些类进行装饰，以此来扩展一些功能。其结构图如下：



- Component为统一接口，也是装饰类和被装饰类的基本类型。
- ConcreteComponent为具体实现类，也是被装饰类，他本身是个具有一些功能的完整的类。
- Decorator是装饰类，实现了Component接口的同时还在内部维护了一个ConcreteComponent的实例，并可以通过构造函数初始化。而Decorator本身，通常采用默认实现，他的存在仅仅是一个声明，我要生产出一些用于装饰的子类了。而其子类才是赋有具体装饰效果的装饰产品类。
- ConcreteDecorator是具体的装饰产品类，每一种装饰产品都具有特定的装饰效果。可以通过构造器声明装饰哪种类型的ConcreteComponent，从而对其进行装饰。[j.csdn.net/010295735](https://www.cnblogs.com/010295735)

```
1  //Component 接口
2  public interface MilkTea{
3      //价格
4      public double cost();
5      //配料
6      public String getIngredinetes();
7  }
8
9  public class SimpleMilkTea implements MilkTea{
10     @Override
11     public double cost(){
12         return 10;
13     }
14     //拿一点点的波霸举例
15     @Override
16     public String getIngredients(){
17         return "BoBa"
18     }
19 }
20
21 //装饰器抽象类 使用 了MilkTea的接口
22 public abstract class MilkTeaDecorator implements MilkTea{
23     private final MilkTea decoratedMilkTea;
```

```

24     public MilkTeaDecorator(MilkTea m){
25         this.decoratedMilkTea = m;
26     }
27     @Override
28     public double cost(){
29         return decoratedMilkTea.cost();
30     }
31
32     @Override
33     public String getIngredients(){
34         return decoratedMilkTea.getIngredients();
35     }
36
37 }
38
39 //继承了装饰器
40 class WithMilk extends MilkTeaDecorator{
41     public WithMilk(MilkTea m){
42         super(m);
43     }
44     //加了牛奶 + 1块钱
45     @Override
46     public double cost(){
47         return super.cost()+1;
48     }
49     //配料加上了牛奶
50     @Override
51     public String getIngredient(){
52         return super.getIngredients + ",Milk";
53     }
54 }
55
56
57 public class Main{
58     public static void main(String[] args){
59         MilkTea m = new SimpleMilkTea();
60
61         m = new WithMilk(c);
62
63     }
64 }

```

总而言之：

- 1 在不改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。
- 2 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。
- 3 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分增加附加的功能。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。