设计模式之策略模式实战

课程目标

1. 了解代码重构
2. 了解策略模式的定义、应用场景
3. 了解 JDK 中策略模式的应用
4. 了解设计原则（开闭原则、单一职责原则）

从一个真实需求案例开始

- 背景为某电商网站上线初期

v1 版本

需求：显示商品价格 iPhone XR 价格 7100 元

```
@ResponseBody

public String getPrice() {

    return "7100.00"

}
```

v2 版本

需求：需要根据客户类型，进行不同的折扣，新客户不打折扣，针对老客户打 9 折。 最简单的 实现代码 版本 1

```
if ("新客户".equals(customType)) {

    System.out.println("抱歉！新客户没有折扣！");

    return originalPrice;

}else if ("老客户".equals(customType)) {

    System.out.println("恭喜你！老客户打 9 折！");

    originalPrice = originalPrice.multiply(new
BigDecimal(0.9)).setScale(2,BigDecimal.ROUND_HALF_UP);
```

```
    return originalPrice;

}
```

V3 版本

现在需求变更了，增加了针对 VIP 客户打 8 折

```
if ("新客户".equals(customType)) {

    System.out.println("抱歉！新客户没有折扣！");

    return originalPrice;

}else if ("老客户".equals(customType)) {

    System.out.println("恭喜你！老客户打 9 折！");

    originalPrice = originalPrice.multiply(new
BigDecimal(0.9)).setScale(2,BigDecimal.ROUND_HALF_UP);

    return originalPrice;

}else if("VIP 客户".equals(customType)){

    System.out.println("恭喜你！VIP 客户打 8 折！");

    originalPrice = originalPrice.multiply(new
BigDecimal(0.8)).setScale(2,BigDecimal.ROUND_HALF_UP);

    return originalPrice;

}
```

V3.5 版本 重构

寻找代码问题，出现了多重条件判断语句，代码耦合严重，不便于维护

第一次重构

提取方法，将分支中的语句提取成单独的方法，通过方法名降低维护难度

```
    public BigDecimal getNewCustomerPrice(BigDecimal originalPrice) {

        return originalPrice;

    }
```

```
    public BigDecimal getOldCustomerPrice(BigDecimal originalPrice) {

        return originalPrice.multiply(new BigDecimal(0.9)).setScale(2,BigDecimal.ROUND_HALF_UP);



    }



    public BigDecimal getVIPCustomerPrice(BigDecimal originalPrice) {

        return originalPrice.multiply(new BigDecimal(0.8)).setScale(2,BigDecimal.ROUND_HALF_UP);



    }
```

第二次重构

提取接口，重构实现类。将报价策略抽象成接口，针对不同的报价策略实现不同的具体类。

```
public interface ICustomerQuotation {

    public BigDecimal getQuotation(BigDecimal originalPrice);

}
```

具体策略实现类

```
public class NewCustomerQuotation implements ICustomerQuotation {

    @Override

    public BigDecimal getQuotation(BigDecimal originalPrice) {

        return originalPrice;

    }

}

public class OldCustomerQuotation implements ICustomerQuotation {

    @Override

    public BigDecimal getQuotation(BigDecimal originalPrice) {

         return originalPrice.multiply(new
BigDecimal(0.9)).setScale(2,BigDecimal.ROUND_HALF_UP);

    }
```

```
}

public class VIPCustomerQuotation implements ICustomerQuotation {

    @Override

    public BigDecimal getQuotation(BigDecimal originalPrice) {

         return originalPrice.multiply(new
BigDecimal(0.8)).setScale(2,BigDecimal.ROUND_HALF_UP);

    }

}
```

客户端调用

```
        ICustomerQuotation quotation;

        if (Constant.NEW.equalsIgnoreCase(customerType)) {

            quotation = new NewCustomerQuotation();

        } else if (Constant.OLD.equalsIgnoreCase(customerType)) {

            quotation = new OldCustomerQuotation();

        } else if (Constant.VIP.equalsIgnoreCase(customerType)) {

            quotation = new VIPCustomerQuotation();

        } else {

            quotation = new NewCustomerQuotation();

        }

        BigDecimal price = quotation.getQuotation(originalPrice);
```

第三次重构

提取 context，通过 context 持有具体策略，屏蔽客户端对具体策略方法的依赖，通过 context 进行隔离。

```
public class CustomerQuotationContext {

    private ICustomerQuotation quotation;

    public CustomerQuotationContext(ICustomerQuotation quotation) {

        this.quotation = quotation;

    }
```

```
    public String getPrice(BigDecimal originalPrice) {
        System.out.println("进行一些前置处理");
        BigDecimal price = quotation.getQuotation(originalPrice);
        System.out.println("进行一些后置处理");
        return price.toPlainString();
    }
}
```

客户端调用

```
    @RequestMapping("/getMyQuotation")
    @ResponseBody
    public String getPrice(String customerType) {
        BigDecimal myPrice = originalPrice;
        ICustomerQuotation quotation;
        if (Constant.NEW.equalsIgnoreCase(customerType)) {
            quotation = new NewCustomerQuotation();
        } else if (Constant.OLD.equalsIgnoreCase(customerType)) {
            quotation = new OldCustomerQuotation();
        } else if (Constant.VIP.equalsIgnoreCase(customerType)) {
            quotation = new VIPCustomerQuotation();
        } else {
            quotation = new NewCustomerQuotation();
        }
        CustomerQuotationContext customerQuotationContext = new
CustomerQuotationContext(quotation);
        return customerQuotationContext.getPrice(originalPrice);
    }
```

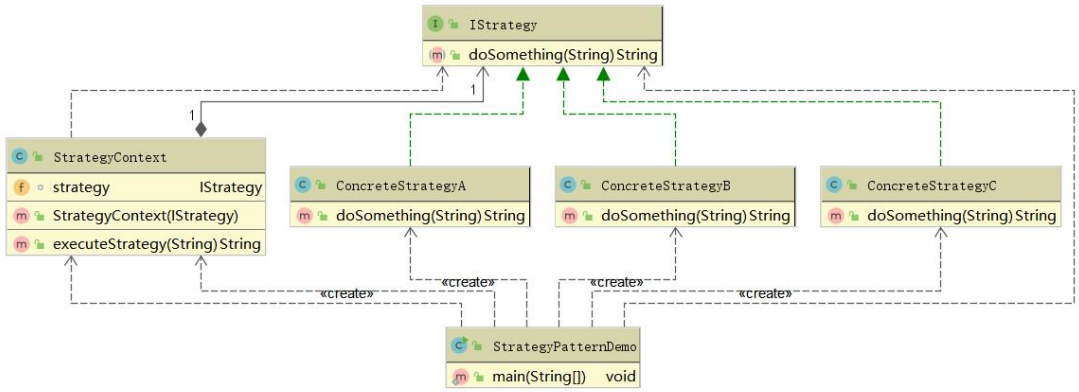
这就是策略模式

引入策略模式

定义

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。

类图



使用场景

1. 多个类只有在算法或行为上稍有不同的场景。
2. 算法需要自由切换的场景。
3. 需要屏蔽算法规则的场景。

电商案例需求变更

V4 版本

增加 MVP 用户，MVP 用户 7 折。 因为使用了策略模式，所以很容易扩展，简单的实现一个针对 MVP 客户的策略实现类即可。

```
public class MVPCustomerQuotation implements ICustomerQuotation {

    @Override

    public BigDecimal getQuotation(BigDecimal originalPrice) {
```

```
        return originalPrice.multiply(new BigDecimal(0.7)).setScale(2,BigDecimal.ROUND_HALF_UP);
    }
}
```

V5 版本

需求变更增加邮费，新客户、老客户邮费 10.11 元，VIP 客户 9.11 元，MVP 客户 0.11 元。 在策略接口中增加新方法 getShippingFee(),具体策略类进行实现。

```
public class MVPCustomerQuotation implements ICustomerQuotation {

    @Override
    public BigDecimal getQuotation(BigDecimal originalPrice) {
        return originalPrice.multiply(new BigDecimal(0.7)).setScale(2,BigDecimal.ROUND_HALF_UP);
    }


    @Override
    public String getCustomerType() {
        return Constant.MVP;
    }


    @Override
    public BigDecimal getShippingFee() {
        return new BigDecimal(0.11);
    }
}
```

在 context 上下文中进行处理，对客户端进行屏蔽

```
    public String getPrice(BigDecimal originalPrice) {
        System.out.println("进行一些前置处理");
        BigDecimal price = quotation.getQuotation(originalPrice);
```

```
        price = price.add(quotation.getShippingFee()).setScale(2, BigDecimal.ROUND_HALF_UP);
        System.out.println("进行一些后置处理");
        return price.toPlainString();
    }
```

电商案例继续重构 V6 版本

如何进一步优化代码呢，能否减少 if else 判断？

1. 对策略接口进行改进，使用@PostConstruct 对策略对象进行 map 管理

```
public interface ICustomerQuotation {
    public static Map<String, ICustomerQuotation> map = new ConcurrentHashMap();

    public BigDecimal getQuotation(BigDecimal originalPrice);

    public BigDecimal getShippingFee();

    public String getCustomerType();

    @PostConstruct
    default public void init() {
        map.put(getCustomerType(), this);
    }
}
```

2. 通过 spring 继续重构，具体实现策略增加@Service，通过 spring 进行管理

```
@Service
public class NewCustomerQuotation implements ICustomerQuotation {
    @Override
    public BigDecimal getQuotation(BigDecimal originalPrice) {
```

```
        return originalPrice;
    }

    @Override
    public String getCustomerType() {
        return Constant.NEW;
    }
    @Override
    public BigDecimal getShippingFee() {
        return new BigDecimal(10.11);
    }
}
```

3. 简化客户端调用

```
@Controller
public class QuotationController {
    public static final BigDecimal originalPrice = new BigDecimal(7100.00).setScale(2,
BigDecimal.ROUND_HALF_UP);

    @RequestMapping("/getMyQuotation")
    @ResponseBody
    public String getPrice(String customerType) {
        BigDecimal myPrice = originalPrice;
        CustomerQuotationContext customerQuotationContext = new
CustomerQuotationContext(customerType);
        return customerQuotationContext.getPrice(originalPrice);
    }
}
```

策略模式其他应用案例

1. 诸葛亮的锦囊妙计，每一个锦囊就是一个策略。三国刘备取西川时，谋士庞统给的上、中、下三个计策：

   上策：挑选精兵，昼夜兼行直接偷袭成都，可以一举而定，此为上计计也。

   中策：杨怀、高沛是蜀中名将，手下有精锐部队，而且据守关头，我们可以装作要回荆州，引他们轻骑来见，可就此将其擒杀，而后进兵成都，此为中计。

   下策：退还白帝，连引荆州，慢慢进图益州，此为下计。

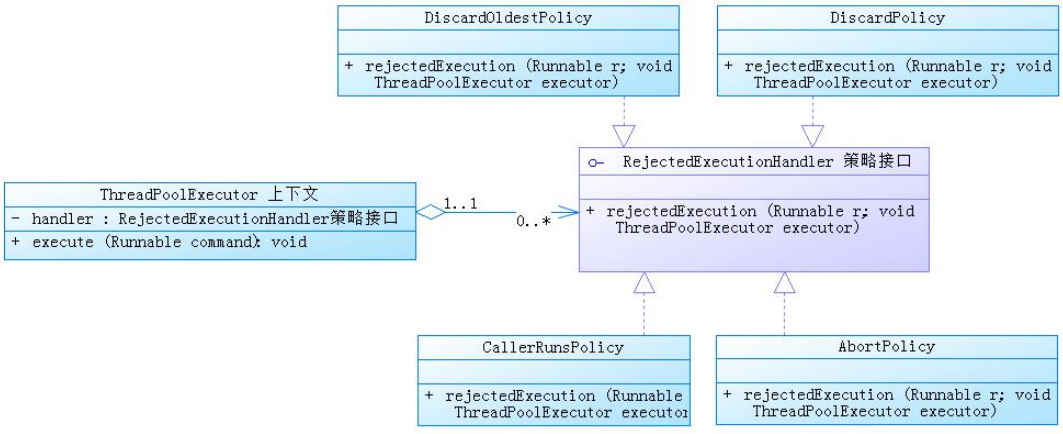
   这三个计策都是取西川的计策，也就是攻取西川这个问题的具体的策略算法，刘备可以采用上策，可以采用中策，当然也可以采用下策，由此可见策略模式的各种具体的策略算法都是平等的，可以相互替换。那谁来选择具体采用哪种计策（算法）？在这个故事中当然是刘备选择了，也就是外部的客户端选择使用某个具体的算法，然后把该算法（计策）设置到上下文当中。还有一种情况就是客户端不选择具体的算法，把这个事交给上下文，这相当于刘备说我不管有哪些攻取西川的计策，我只要结果（成功的拿下西川），具体怎么攻占（有哪些计策，怎么选择）由参谋部来决定（上下文）。

2. 旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略。

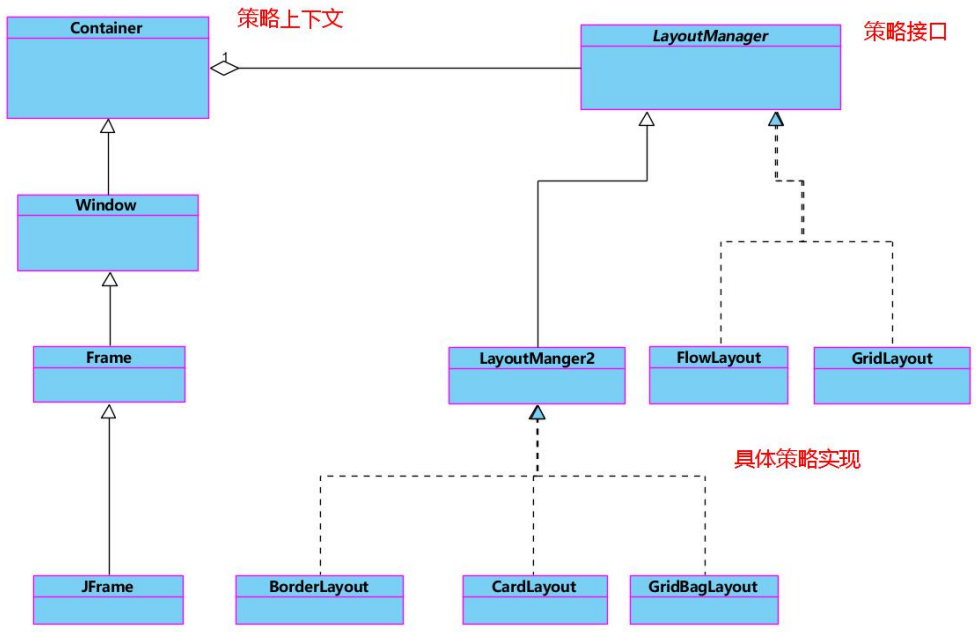
策略模式在 JDK 中的应用

ThreadPoolExecutor

RejectedExecutionHandler 是一个策略接口，用在当线程池中没有多余的线程来执行任务，并且保存任务的多列也满了（指的是有界队列），对仍在提交给线程池的任务的处理策略。



Swing 中的 LayoutManager



Collections 排序中的应用 Comparator

我们如果需要控制某个类的次序，而该类本身不支持排序（即没有实现 Comparable 接口）；那么可以建立一个该类的比较器来排序，这个比较器只需要实现 Comparator 接口即可。,通过实现 Comparator 类来新建一个比较器，然后通过该比较器来对类进行排序。Comparator 接口其实就是一种策略模式的实践 事例代码： 抽象策略类 Comparator

```
public interface Comparator<T> {

    int compare(T o1, T o2);

    boolean equals(Object obj);

 }
```

具体策略类 SortComparator

```
public class SortComparator implements Comparator {



    @Override

    public int compare(Object o1, Object o2) {

        Student student1 = (Student) o1;

        Student student2 = (Student) o2;
```

```
        return student1.getAge() - student2.getAge();
    }
}
```

策略模式上下文 Collections

```
public class Client {


    public static void main(String[] args) {


        Student stu[] = {
                new Student("张三" ,23),
                new Student("李四" ,26)
                , new Student("王五" ,22)};
        Arrays. sort(stu,new SortComparator());
        System.out.println(Arrays.toString(stu));


        List<Student> list = new ArrayList<>(3);
        list.add( new Student("zhangsan" ,31));
        list.add( new Student("lisi" ,30));
        list.add( new Student("wangwu" ,35));
        Collections.sort(list,new SortComparator());
        System.out.println(list);


    }


}
```

策略模式优缺点

优点：

1. 算法可以自由切换。
2. 避免使用多重条件判断。
3. 扩展性良好 (1) 策略模式提供了对“开闭原则”的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。 (2) 算法的使用就和算法本身分开，符合“单一职责原则”； (3) 策略模式提供了一种算法的复用机制，由于将算法单独提取出来封装在策略类中，因此不同的环境类可以方便地复用这些策略类。

缺点

1. 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法。换言之，策略模式只适用于客户端知道所有的算法或行为的情况。
2. 策略模式将造成系统产生很多具体策略类，任何细小的变化都将导致系统要增加一个新的具体策略类。
3. 无法同时在客户端使用多个策略类，也就是说，在使用策略模式时，客户端每次只能使用一个策略类，不支持使用一个策略类完成部分功能后再使用另一个策略类来完成剩余功能的情况。