1、简介

策略模式(Strategy)是一种设计模式，它允许在运行时根据需要选择算法的行为。这个模式将每个算法封装到一个类中，并使它们可互换。让客户端代码可以独立于算法变化而改变其行为。

策略模式通常应用于需要多种算法进行操作的场景，如排序、搜索、数据压缩等。在这些情况下，不同的算法有不同的优缺点和适用性，因此需要进行选择。

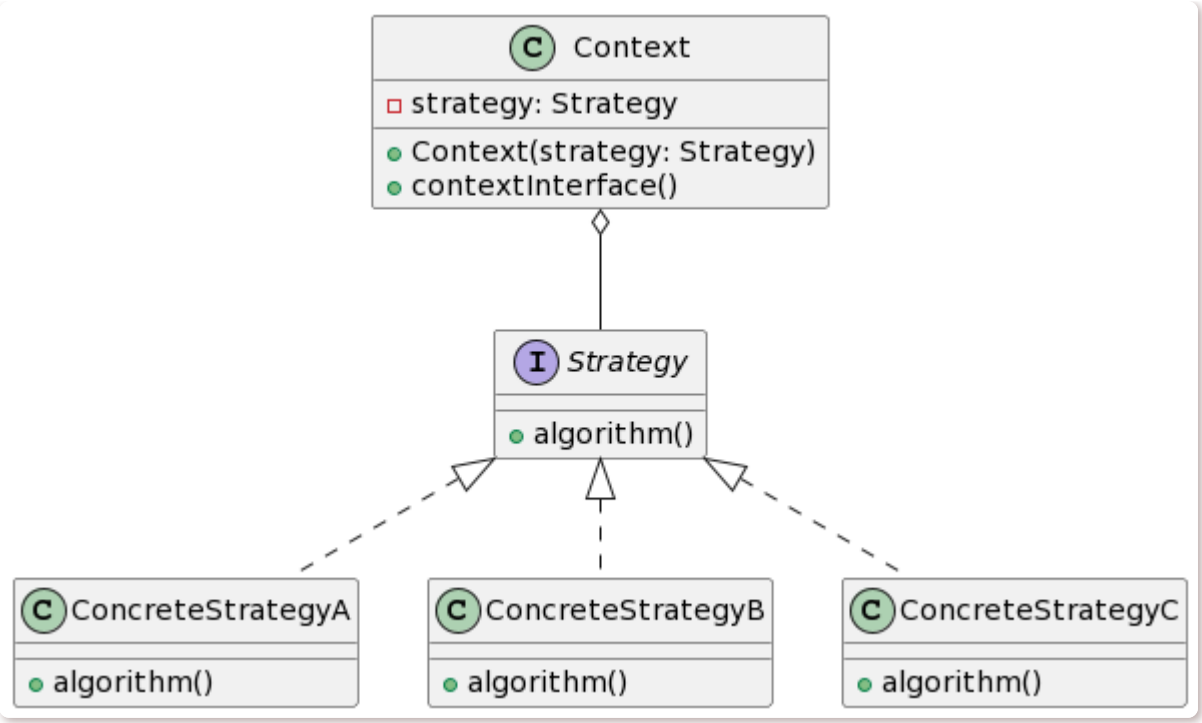
通过使用策略模式，我们可以轻松地切换算法，而无需修改客户端代码。这使得代码更加灵活、可扩展、易于维护，减少了重复的代码，并提高了代码的可读性。

2、结构

策略模式的结构包括以下几个部分：

1. 抽象策略（Strategy）类：定义所有支持的算法或行为的**公共接口或抽象类**。
2. 具体策略（Concrete Strategy）类：**实现抽象策略接口**，提供具体的算法或行为。
3. 环境（Context）类：持有一个对抽象策略的引用，并且通过该引用调用具体策略类中实现的算法或行为。

在策略模式中，客户端代码仅与环境类及其抽象策略接口交互，无需关心具体实现细节。当需要更改算法或行为时，只需要修改具体策略类即可，而无需修改客户端代码或其他策略类。



3、实现方式

3.1、案例引入

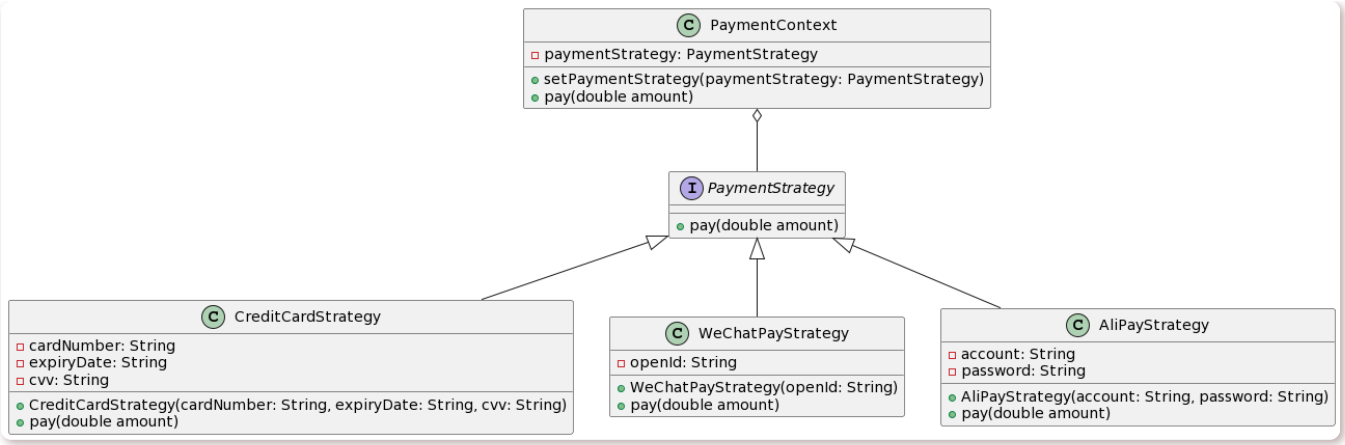
在支付系统中，我们就可以使用策略模式，针对不同的支付方式封装成不同的策略类。每个策略类负责实现一个特定的支付方式，并提供相应的算法来处理付款。这样，客户端代码就可以通过选择不同的策略类来实现不同的支付方式，而无需了解每种支付方式的具体实现细节。

例如，假设我们有一个电子商务网站，支持**信用卡、微信支付和支付宝支付**三种支付方式。每种支付方式都有自己的实现方式和规则，但**客户端并不需要知道支付方式的具体实现**，只需要**选择一个支付策略即可**。这时候，我们可以使用策略模式来实现支付系统，将**每种支付方式封装成一个支付策略类**，让客户端根据需要选择不同的支付策略类。

3.2、结构分析

在这个案例中，代码中的各个结构可以对应到策略模式中的不同角色：

1. **Context**（上下文）：对应代码中的 *PaymentContext* 类，负责调用具体支付策略对象的付款方法。
2. **Strategy**（策略）：对应代码中的 *PaymentStrategy* 接口，定义了所有支持的支付方式的公共接口。
3. **ConcreteStrategy**（具体策略）：对应代码中的 *CreditCardStrategy*、*WeChatPayStrategy* 和 *AliPayStrategy* 类，实现了 *PaymentStrategy* 接口，提供了不同的支付方式实现逻辑。



3.3、具体实现

针对上述场景，具体实现代码如下：

首先定义一个支付策略接口 *PaymentStrategy*，该接口声明了一个方法 *pay(double amount)* 用于处理付款：

```
public interface PaymentStrategy {
    void pay(double amount);
}
```

接着定义三个具体的支付策略类，分别是 *CreditCardStrategy*、*WeChatPayStrategy* 和 *AliPayStrategy*。每个类实现了 *PaymentStrategy* 接口，并提供了自己的付款实现方式：

```
/**
 * 信用卡
 */
public class CreditCardStrategy implements PaymentStrategy {
    /** 信用卡卡号 */
    private String cardNumber;
    /** 信用卡有效期 */
    private String expiryDate;
    /** 信用卡CVV码 */
    private String cvv;

    public CreditCardStrategy(String cardNumber, String expiryDate, String cvv) {
        this.cardNumber = cardNumber;
        this.expiryDate = expiryDate;
        this.cvv = cvv;
    }

    @Override
    public void pay(double amount) {
        // 基于信用卡的付款逻辑
        System.out.println("使用信用卡支付 " + amount + " 元");
    }
```

```java
}

/**
 * 微信
 */
public class WeChatPayStrategy implements PaymentStrategy {
    /** 微信openId */
    private String openId;

    public WeChatPayStrategy(String openId) {
        this.openId = openId;
    }

    @Override
    public void pay(double amount) {
        // 基于微信支付的付款逻辑
        System.out.println("使用微信支付 " + amount + " 元");
    }
}

/**
 * 支付宝
 */
public class AliPayStrategy implements PaymentStrategy {
    private String account;
    private String password;

    public AliPayStrategy(String account, String password) {
        this.account = account;
        this.password = password;
    }

    @Override
    public void pay(double amount) {
        // 基于支付宝的付款逻辑
        System.out.println("使用支付宝支付 " + amount + " 元");
    }
}
```

最后定义一个支付上下文类 `PaymentContext`，该类负责根据客户端选择的支付方式创建对应的支付策略对象，并调用其付款方法：

```java
public class PaymentContext {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy)
{
        this.paymentStrategy = paymentStrategy;
    }
```

```
    public void pay(double amount) {
        if(paymentStrategy == null) {
            throw new IllegalArgumentException("支付策略不能为空");
        }
        paymentStrategy.pay(amount);
    }
}
```

客户端代码可以根据需要选择不同的支付方式，例如：

```
// 创建一个信用卡支付策略对象
CreditCardStrategy creditCardStrategy = new CreditCardStrategy("1234567890123456", "2023-12", "123");
// 创建一个微信支付策略对象
WeChatPayStrategy weChatPayStrategy = new WeChatPayStrategy("wxopenid123456");
// 创建一个支付宝支付策略对象
AliPayStrategy aliPayStrategy = new AliPayStrategy("alipayaccount@aliyun.com", "alipaypassword");

// 创建一个支付上下文对象
PaymentContext paymentContext = new PaymentContext();

// 使用信用卡支付
paymentContext.setPaymentStrategy(creditCardStrategy);
paymentContext.pay(100.0);

// 使用微信支付
paymentContext.setPaymentStrategy(weChatPayStrategy);
paymentContext.pay(200.0);

// 使用支付宝支付
paymentContext.setPaymentStrategy(aliPayStrategy);
paymentContext.pay(300.0);
```

这个示例演示了如何使用策略模式来实现支付系统。客户端代码可以根据需要选择不同的支付策略，而无需关心具体的支付方式实现细节。同时，新的支付方式也可以很方便地添加到系统中，只需要新增一个实现了 *PaymentStrategy* 接口的类即可。

运行结果如下：

```
使用信用卡支付 100.0 元
使用微信支付 200.0 元
使用支付宝支付 300.0 元
```

4、对比模板方法模式

模板方法模式和策略模式都属于面向对象编程中的行为型设计模式，它们的目标都是**封装算法和行为**，以提高代码复用性和可维护性。但它们的实现方式和应用场景有所不同。

- **模板方法模式**：定义了一个**算法的骨架**，将具体实现延迟到子类中完成，在保持算法结构不变的同时允许子类灵活地实现算法的具体步骤。这种模式常用于处理一些重复性较高的操作，比如在编写一系列相似的程序时，可以使用该模式来避免重复代码的出现；
- **策略模式**：定义了一系列**算法族**（即一组相似的算法），并将每个算法封装起来，使得它们可以互相替换。这样就能够使得算法的变化独立于使用算法的客户端。这种模式适用于需要在运行时动态选择算法的情况，或者需要对多个相关但不完全相同的算法进行封装的情况。

> 🌈
>
> 使用白话文来讲就是：
>
> - 模板方法模式固定了某一个事件的具体流程，运行拓展的是中间的某一个流程；
> - 策略模式则是针对某一个算法(等同上述的流程)支持拓展。

因此，虽然这两种模式都是用于封装算法和行为，但模板方法模式是基于**继承**实现的，它只有一个具体的模板类和一些具体的子类，而策略模式则是基于**组合**实现的，它包含了一组策略类和一个具体的上下文类。两者的应用场景和实现方式不同，需要根据具体需求选择合适的模式来使用。

	模板方法模式	策略模式
定义	定义算法骨架，子类实现具体步骤	封装一系列算法，并使其互相替换
实现方式	基于继承实现	基于组合实现
子类数量	通常只有一个抽象模板类和多个具体子类	可以有多个具体策略类
调用	父类定义算法流程，子类重写具体方法	上下文类选择并调用具体策略
目的	避免重复代码，保持算法结构不变	允许动态选择算法

在开发过程中这两个模式经常配合使用，**策略模式通常被用来代替模板方法模式中的部分具体算法。**

具体而言，在使用模板方法模式时，如果某些具体步骤需要根据特定条件进行选择或者动态替换，就可以把这些步骤视为算法族，并使用策略模式来封装和管理它们。这样可以使得算法选择更加灵活，同时还能够保持模板方法模式的算法结构不变。另外，策略模式还可以在运行时动态地替换算法，不需要修改源代码即可实现。

例如，在一个游戏开发中，我们可以使用模板方法模式定义一个游戏的基本流程，包括游戏开始、游戏进行、游戏结束等步骤。然后，针对不同类型的游戏，我们可以使用策略模式来定义不同的游戏规则，如赛车游戏、飞行游戏等，每种游戏规则都是一种具体的策略。这样，我们就可以根据实际需求来动态地选择并组合不同的游戏规则，从而实现不同类型的游戏。

5、策略模式优缺点

策略模式主要优点和缺点如下：

优点：

1. 策略模式使得各种算法可以在不修改原有代码的情况下替换或者新增，提高了代码的可扩展性和可维护性；
2. 策略模式可以避免由于多重条件语句导致的代码复杂度增加和可读性降低的问题；
3. 策略模式将算法的实现从上下文中解耦出来，使得算法可以独立进行单元测试；
4. 策略模式符合开闭原则，即对扩展开放，对修改关闭，可以通过增加新的策略类来扩展应用，而无需修改原有代码。

缺点：

1. 如果策略数量过多，会导致类数量增加，增加系统的复杂度；
2. 客户端需要知道所有的策略类，并选择合适的策略类，这可能会导致客户端代码较为复杂；
3. 策略模式将算法的实现从上下文中解耦出来，同时也意味着上下文不能控制策略的执行顺序，需要客户端自行控制执行顺序。

优点	缺点
提高代码的可扩展性和可维护性	策略数量过多，增加类数量和系统复杂度
避免由于多重条件语句导致的代码复杂度增加和可读性降低的问题	客户端需要知道所有的策略类，并选择合适的策略类，导致客户端代码较为复杂
将算法的实现从上下文中解耦出来，使得算法可以独立进行单元测试	上下文不能控制策略的执行顺序，需要客户端自行控制执行顺序
符合开闭原则，即对扩展开放，对修改关闭	

6、应用场景

策略模式通常适用于以下场景中：

1. 系统需要动态地在几种算法中选择一种，或者根据不同的条件选择不同的算法；
2. 系统中有许多类似的行为，但是具体实现上有所不同，可以使用策略模式将这些行为抽象出来，并定义一个接口或抽象类，然后由具体的实现类来实现这个接口或抽象类；
3. 一些算法使用了相同的数据，但是实现上有所不同，可以使用策略模式来避免代码重复和代码膨胀，节省代码维护成本。

> 例如，在电商系统中，针对不同的促销活动，可能会有不同的优惠计算方式，比如满减、打折等。如果使用简单的if-else语句来实现这些计算方式，会导致代码复杂度增加，可读性降低，扩展也不方便。使用策略模式，可以将各种优惠计算方式进行抽象和封装，客户端只需要知道各个策略类的作用，就可以方便地调用它们进行计算。

在Java和Spring中，以下是一些应用了策略模式的例子：

1. Java中的Collections.sort()方法使用了策略模式。在调用该方法时，可以传递一个Comparator对象作为参数，该对象定义了排序的策略。
2. 在Spring框架中，JdbcTemplate类使用了策略模式来处理各种不同类型的数据访问操作。JdbcTemplate类接受一个回调对象，该对象提供了执行SQL查询或更新的具体实现。
3. 在Spring Security框架中，AuthenticationProvider接口使用了策略模式。AuthenticationProvider接口定义了验证用户身份的策略，并且可以同时支持多种不同的身份验证方式。
4. 在Java中，Thread类的构造函数可以接受一个Runnable对象作为参数，该对象定义了线程运行的策略。