# 策略模式简介

策略模式用于算法的自由切换与扩展，它是使用较为广泛的设计模式之一。策略模式对应于解决某一问题的一个算法族，允许用户从该算法族中任选一个算法解决某一问题，同时可以方便地更换算法或者增加新的算法。策略模式实现了算法定义和算法使用的分离，它通过继承和多态的机制实现对算法族的使用和管理，是一个简单、实用的设计模式。

## 概述

1. 在很多情况下，实现某一个目标的途径不止一条，例如在外出旅游时游客可以选择多种不同的出行方式，如骑自行车、坐汽车、坐火车或者坐飞机，可根据实际情况（目的地距离、旅游预算、旅游时间等）来选择一种最适合的出行方式。在制定计划时，如果目的比较远，时间不多、但不差钱，可以选择坐飞机去旅游；如果目的地虽远，但假期长，且需控制旅游成本时可以选择坐火车或汽车；如果从健康和环保角度考虑，而且有足够的毅力，自行车游或者徒步旅游也是个不错的选择。在软件开发中也常常会遇到类似的情况，实现某一功能（例如排序、查找等）有多种算法，一种常用的方法是通过硬编码（Hard Coding）将所有的算法集中在一个类中，在该类中提供多个方法，每一个方法对应一个具体的算法；当然也可以将这些算法封装在一个统一的方法中，通过if……else等条件判断语句进行选择。这两种实现方法都可以称为硬编码。如果需要增加一种新的算法，需要修改算法类的源代码；更换算法也需要修改客户端调用代码。在这个统一的算法类封装了大量算法，代码非常复杂，维护也很困难。此时可以使用一种设计模式来实现灵活地选择算法，还能方便地增加新的算法，该设计模式就是策略模式。
2. 在策略模式中可以定义一些独立的类来封装不同的算法，每一个类封装一种具体的算法，在这里每一个封装算法的类都可以称为一种策略（Strategy），为了保证这些策略在使用时具有一致性，一般会提供一个抽象的策略类来做算法的声明，而每种算法对应一个具体策略类。
3. 策略模式（Strategy Pattern）的定义如下：定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法可以独立于使用它的客户而变化。
4. 策略模式又称为政策（Policy）模式，它是一种对象行为型模式。

## 结构

策略模式的结构并不复杂，其结构图如下：



图中，策略模式包含以下3个角色：

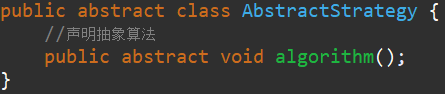
1. Context（环境类）：环境类是使用算法的角色，它在解决某个问题（即实现某个功能）时可以采用多种策略。在环境类中维持一个对抽象策略类的引用实例，用于定义所采用的策略。
2. Strategy（抽象策略类）：抽象策略类为所支持的算法声明了抽象方法，是所有策略类的父类，它可以是抽象类或具体类，也可以是接口。环境类通过抽象策略类中声明的方法在运行时调用具体策略类中实现的算法。
3. ConcreteStrategy（具体策略类）：具体策略类实现了在抽象策略类中声明的算法，在运行时具体策略类通过多态性覆盖在环境类中定义的抽象策略类对象，使用一种具体的算法实现某个业务功能。

# 实现

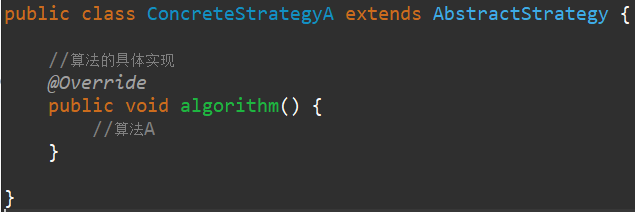
## 实现原理

策略模式是一个很容易理解和使用的设计模式，策略模式是对算法的封装，它把算法的责任和算法本身分隔开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列具体策略类里面，作为抽象策略类的子类。在策略模式中对环境类和抽象策略类的理解非常重要，环境类是需要使用算法的类。在一个系统中可以存在多个环境类，它们可能需要重用一些相同的算法。

1. 在使用策略模式时需要将算法从环境类Context中提取出来，首先应该创建一个抽象策略类，其典型代码如下：

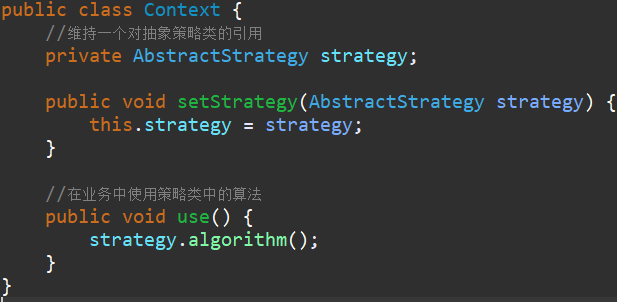


1. 然后将封装每一种具体算法的类作为抽象策略类的子类，代码如下：

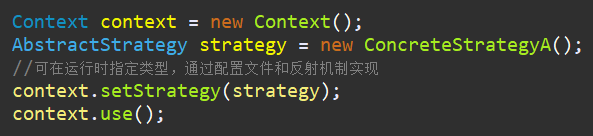


其他具体策略类与之类似。

1. 对于Context类而言，在它与抽象策略类之间建立一个关联关系，其典型代码如下：



1. 在Context类中定义一个AbstractStrategy类型的对象strategy，通过注入的方式在客户端传入一个具体策略对象，客户端代码片段如下：



在客户端只需要注入一个具体策略类对象，可以将具体策略类的类名存储在配置文件中，通过反射来动态创建具体策略对象，从而使得用户可以灵活地更换具体策略类，增加新的具体策略类也很方便。策略模式提供了一种可插入式（Pluggable）算法的实现方案。

## 实例

实例说明：某软件公司为某电影院开发了一套影院售票系统，在该系统中需要为不同类

型的用户提供不同的电影打折方式，具体打折方案如下：

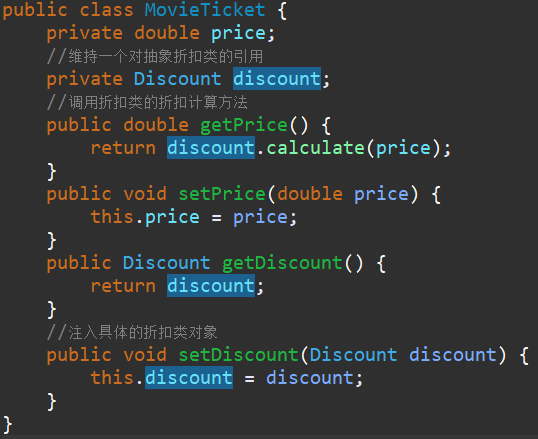
1. 学生凭学生证可享受票价8折优惠；
2. 年龄在大于10周岁及以下的儿童可享受每张票减免10元的优惠（原始票价需大于等于20元）
3. 影视VIP用户除享受票价半折优惠外还可进行积分，积分累计到一定额度可换取电影赠送的礼品

该系统在将来可能还要根据需要引入新的打折方式。现使用策略模式对其进行设计，通过分析，本实例的结构图如下所示：

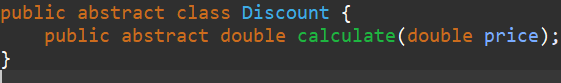


图中，MovieTicket充当环境类角色，Discount充当抽象策略角色，StudentDiscount、ChildrenDiscount和VIPDiscount充当具体策略角色。

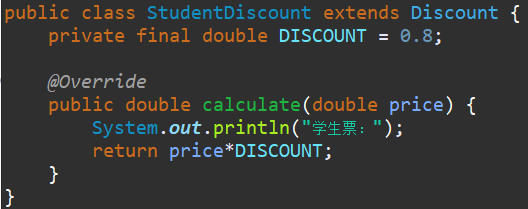
1. MovieTicket：电影票类，充当环境类



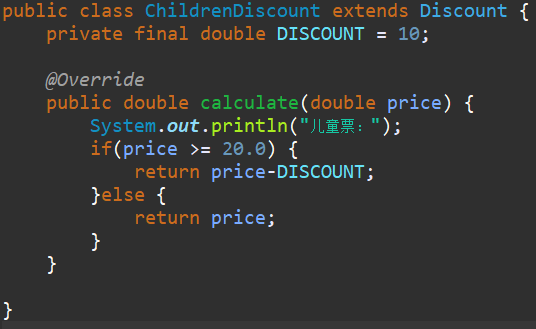
1. Discount：折扣类，充当抽象策略类



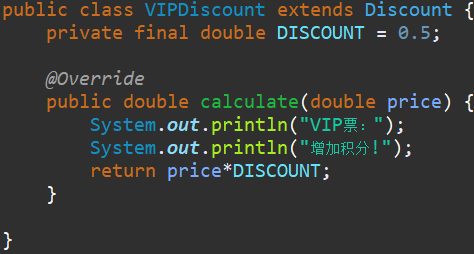
1. StudentDiscount：学生票折扣类，充当具体策略类



1. ChildrenDiscount：儿童票折扣类，充当具体策略类



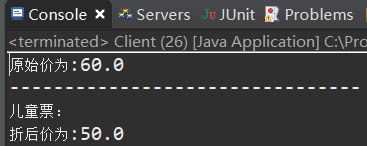
1. VIPDiscount：VIP票折扣类，充当具体策略类



1. Client：客户端测试类



运行结果如下：



如果需要增加新的打折方式，原有代码均无须修改，只要增加一个新的折扣类作为抽象折扣类的子类。

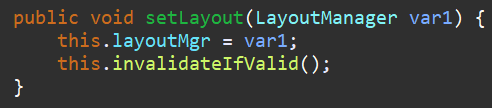
如果需要更换具体策略类，可以通过配置文件（如XML文件）的方式达到无须修改源代码就可以更换具体策略类的目的。

## JavaSE中的布局管理

JavaSE中的容器布局管理是策略模式的一个经典案例，其基本结构如下：



在JavaSE开发中，用户需要对容器对象Container中的成员对象（例如按钮、文本框等GUI控件）进行布局（Layout），在程序运行期间由客户端动态决定一个Container对象如何布局，Java语言在JDK中提供了几种不同的布局方式，封装在不同的类中，例如BorderLayout、FlowLayout、GridBagLayout和CardLayout等。在上图中，Container充当环境类角色Context、LayoutManager作为所有布局类的公共父类扮演了抽象策略角色，它给出所有具体布局类所需的接口，而具体策略类是LayoutManager的子类，也就是各种具体的布局类，它们封装了不同的布局方式。任何人都可以开发自己的布局类，只需要将自己的设计的布局类作为LayoutManager的子类即可。在JDK中，Container类的代码片段如下：



从上述代码中可以看出，Container作为环境类，它针对抽象策略类LayoutManager进行编程，用户在使用时只需要在setLayout( )方法中传入一个具体布局对象即可，而无须关心该布局对象的具体实现。

# 优缺点及适用环境

策略模式用于算法的自由切换与扩展，它是应用较为广泛的设计模式之一。策略模式对应于解决某一问题的一个算法族，允许用户从该算法族中任选一个算法来解决该问题，同时可以方便地更换算法或者增加新的算法，只要涉及算法的封装，复用、切换和扩展都可以考虑使用策略模式。

## 优点

策略模式的优点主要如下：

1. 策略模式提供了对外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. 策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族，恰当地使用继承可以把公共的代码移到抽象策略类中，从而避免重复的代码。
3. 策略模式提供了一种可以替换继承关系的办法。如果不使用策略模式，那么使用算法的环境类就可能会有一些子类，每一个子类提供一种不同的算法。但是这样依赖算法的使用就和算法本身混在一起，不符合单一职责原则，决定使用哪一种算法的逻辑和该算法本身混合在一起，从而不可能再独立演化；而且使用继承无法实现算法或行为在程序运行时的动态切换。
4. 使用策略模式可以避免多重条件选择语句。多重条件选择语句不易维护，它把取哪一种算法或行为的逻辑与算法或行为本身的实现逻辑混合在一起，将它们全部硬编码在一个庞大的多重条件选择语句中，比直接继承环境类的办法还要原始和落后。
5. 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

策略模式的缺点主要如下：

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那。换而言之，策略模式只适用于客户端知道所有算法或行为的情况。
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完成剩余功能的情况。

## 适用环境

在以下情况下可以考虑使用策略模式：

1. 一个系统需要动态地在几种算法中选择一种，那么可以将这些算法封装待一个一个的具体算法类中，而这些具体算法类都是一个抽象算法类的子类。换而言之，这些具体算法均有统一的接口，根据里氏代换原则和面向对象的多态性，客户端可以选择使用一个具体算法类，并只需要维持一个数据类型是抽象算法类的对象。
2. 一个对象有很多行为，如果不用恰当的模式，这些行为则只好使用多重条件选择语句来实现。此时使用策略模式把这些行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句。
3. 不希望客户端知道复杂的、与算法相关的数据结构，在具体策略类中封装算法与相关的数据结构，可以提高算法的保密性与安全性。