# **一、引言**

 　　前面主题介绍的状态模式是对某个对象状态的抽象，而本文要介绍的策略模式也就是对策略进行抽象，策略的意思就是方法，所以也就是对方法的抽象，下面具体分享下我对策略模式的理解。

# **二、策略者模式介绍**

## **2.1 策略模式的定义**

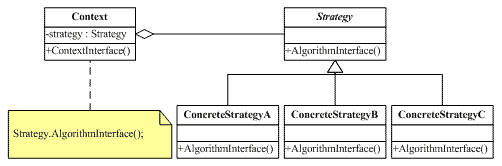
　　在现实生活中，策略模式的例子也非常常见，例如，中国的所得税，分为企业所得税、外商投资企业或外商企业所得税和个人所得税，针对于这3种所得税，针对每种，所计算的方式不同，个人所得税有个人所得税的计算方式，而企业所得税有其对应计算方式。如果不采用策略模式来实现这样一个需求的话，可能我们会定义一个所得税类，该类有一个属性来标识所得税的类型，并且有一个计算税收的CalculateTax()方法，在该方法体内需要对税收类型进行判断，通过if-else语句来针对不同的税收类型来计算其所得税。这样的实现确实可以解决这个场景吗，但是这样的设计不利于扩展，如果系统后期需要增加一种所得税时，此时不得不回去修改CalculateTax方法来多添加一个判断语句，这样明白违背了“开放——封闭”原则。此时，我们可以考虑使用策略模式来解决这个问题，既然税收方法是这个场景中的变化部分，此时自然可以想到对税收方法进行抽象。具体的实现代码见2.3部分。

　　前面介绍了策略模式用来解决的问题，下面具体给出策略的定义。策略模式是针对一组算法，将每个算法封装到具有公共接口的独立的类中，从而使它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。

## **2.2 策略模式的结构**

　　策略模式是对算法的包装，是把使用算法的责任和算法本身分割开，委派给不同的对象负责。策略模式通常把一系列的算法包装到一系列的策略类里面。用一句话慨括策略模式就是——“将每个算法封装到不同的策略类中，使得它们可以互换”。

　　下面是策略模式的结构图：



　　该模式涉及到三个角色：

* 环境角色（Context）：持有一个Strategy类的引用
* 抽象策略角色（Strategy）：这是一个抽象角色，通常由一个接口或抽象类来实现。此角色给出所有具体策略类所需实现的接口。
* 具体策略角色（ConcreteStrategy）：包装了相关算法或行为。

## **2.3 策略模式的实现**

　　下面就以所得税的例子来实现下策略模式，具体实现代码如下所示：

namespace StrategyPattern

{

// 所得税计算策略

public interface ITaxStragety

{

double CalculateTax(double income);

}

// 个人所得税

public class PersonalTaxStrategy : ITaxStragety

{

public double CalculateTax(double income)

{

return income \* 0.12;

}

}

// 企业所得税

public class EnterpriseTaxStrategy : ITaxStragety

{

public double CalculateTax(double income)

{

return (income - 3500) > 0 ? (income - 3500) \* 0.045 : 0.0;

}

}

public class InterestOperation

{

private ITaxStragety m\_strategy;

public InterestOperation(ITaxStragety strategy)

{

this.m\_strategy = strategy;

}

public double GetTax(double income)

{

return m\_strategy.CalculateTax(income);

}

}

class App

{

static void Main(string[] args)

{

// 个人所得税方式

InterestOperation operation = new InterestOperation(new PersonalTaxStrategy());

Console.WriteLine("个人支付的税为：{0}", operation.GetTax(5000.00));

// 企业所得税

operation = new InterestOperation(new EnterpriseTaxStrategy());

Console.WriteLine("企业支付的税为：{0}", operation.GetTax(50000.00));

Console.Read();

}

}

}

# **三、策略者模式在.NET中应用**

 　　在.NET Framework中也不乏策略模式的应用例子。例如，在.NET中，为集合类型ArrayList和List<T>提供的排序功能，其中实现就利用了策略模式，定义了IComparer接口来对比较算法进行封装，实现IComparer接口的类可以是顺序，或逆序地比较两个对象的大小，具体.NET中的实现可以使用反编译工具查看[List<T>.Sort(IComparer<T>)](http://msdn.microsoft.com/zh-cn/library/234b841s(v=vs.110).aspx" \t "http://www.cnblogs.com/zhili/p/_blank)的实现。其中List<T>就是承担着环境角色，而IComparer<T>接口承担着抽象策略角色，具体的策略角色就是实现了IComparer<T>接口的类，List<T>类本身实现了存在实现了该接口的类，我们可以自定义继承与该接口的具体策略类。

# **四、策略者模式的适用场景**

 　　在下面的情况下可以考虑使用策略模式：

* 一个系统需要动态地在几种算法中选择一种的情况下。那么这些算法可以包装到一个个具体的算法类里面，并为这些具体的算法类提供一个统一的接口。
* 如果一个对象有很多的行为，如果不使用合适的模式，这些行为就只好使用多重的if-else语句来实现，此时，可以使用策略模式，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，并体现面向对象涉及的概念。

# **五、策略者模式的优缺点**

 　　策略模式的主要优点有：

* 策略类之间可以自由切换。由于策略类都实现同一个接口，所以使它们之间可以自由切换。
* 易于扩展。增加一个新的策略只需要添加一个具体的策略类即可，基本不需要改变原有的代码。
* 避免使用多重条件选择语句，充分体现面向对象设计思想。

　　策略模式的主要缺点有：

* 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。这点可以考虑使用IOC容器和依赖注入的方式来解决，关于IOC容器和依赖注入（Dependency Inject）的文章可以参考：[IoC 容器和Dependency Injection 模式](http://www.cnblogs.com/lusd/articles/3175062.html" \t "http://www.cnblogs.com/zhili/p/_blank)。
* 策略模式会造成很多的策略类。

# **六、总结**

　　到这里，策略模式的介绍就结束了，策略模式主要是对方法的封装，把一系列方法封装到一系列的策略类中，从而使不同的策略类可以自由切换和避免在系统使用多重条件选择语句来选择针对不同情况来选择不同的方法。