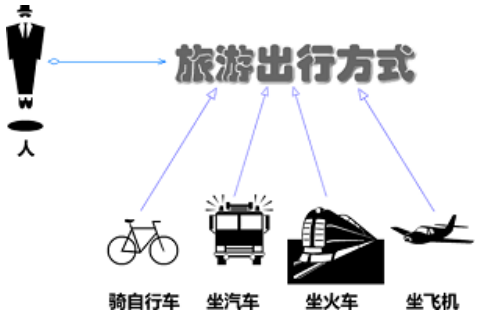
设计模式 (十八) 策略模式Strategy（对象行为型）

设计模式 (十八) 策略模式Strategy（对象行为型）[2]

1.概述[3]

> 在软件开发中也常常遇到类似的情况，实现某一个功能有多种算法[4]或者策略，我们可以根据环境或者条件的不同选择不同的算法或者策略来完成该功能。如查找、排序等，一种常用的方法是硬编码*(Hard Coding)*在一个类中，如需要提供多种查找算法，可以将这些算法写到一个类中，在该类中提供多个方法，每一个方法对应一个具体的查找算法；当然也可以将这些查找算法封装在一个统一的方法中，通过*if…else…*或者等条件判断语句来进行选择。这两种实现方法我们都可以称之为硬编码，如果需要增加一种新的查找算法，需要修改封装算法类的源代码；更换查找算法，也需要修改客户端调用代码。在这个算法类中封装了大量查找算法，该类代码将较复杂，维护较为困难。如果我们将这些策略包含在客户端，这种做法更不可取，将导致客户端程序庞大而且难以维护，如果存在大量可供选择的算法时问题将变得更加严重。
>
> 例子*1*：一个菜单功能能够根据用户的“皮肤”首选项来决定是否采用水平的还是垂直的排列形式。同事可以灵活增加菜单那的显示样式。
>
> 例子*2*：出行旅游：我们可以有几个策略可以考虑：可以骑自行车，汽车，做火车，飞机。每个策略都可以得到相同的结果，但是它们使用了不同的资源。选择策略的依据是费用，时间，使用工具还有每种方式的方便程度 。
>
> 
>

2.问题[5]

> 如何让算法和对象分开来，使得算法可以独立于使用它的客户而变化?

3.解决方案[6]

> 策略模式[7]：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。本模式使得算法可独立于使用它的客户而变化。*(Policy)*[8]*Definea family of algorithmsencapsulate each onemake them interchangeable. Strategy lets the algorithmvary independently from clients that use it.*
>
> 策略模式把对象本身和运算规则区分开来，其功能非常强大，因为这个设计模式本身的核心思想就是面向对象编程的多形性的思想。

4.适用性[9]

当存在以下情况时使用*Strategy*模式

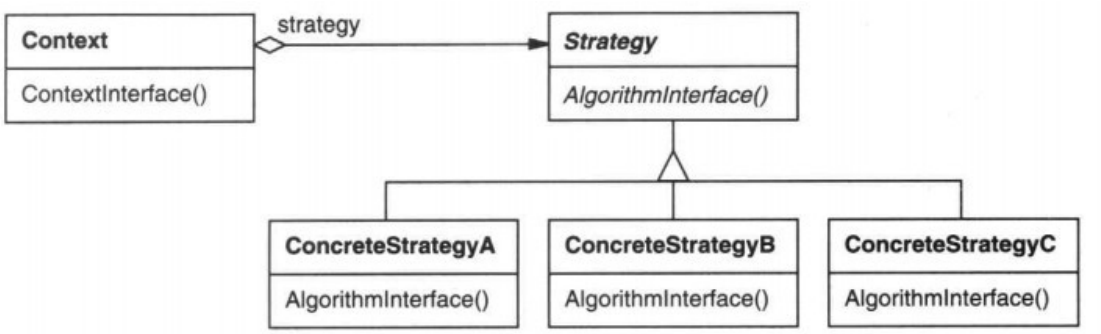
1）• 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。即一个系统需要动态地在几种算法中选择一种。

2）• 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间 /时间权衡的算法。当这些变体实现为一个算法的类层次时，可以使用策略模式。

3）• 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构[10]。

4）• 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的*Strategy*类中以代替这些条件语句。

5.结构[11]



6.模式的组成[12]

环境类*(Context)*[13]:用一个*ConcreteStrategy*对象来配置。维护一个对*Strategy*对象的引用。可定义一个接口来让*Strategy*访问它的数据。

抽象策略类*(Strategy)*:[14]定义所有支持的算法的公共接口。 *Context*使用这个接口来调用某*ConcreteStrategy*定义的算法。

具体策略类*(ConcreteStrategy)*[15]:以*Strategy*接口实现某具体算法。

7.效果[16]

*Strategy*模式有下面的一些优点：

1) 相关算法系列 *Strategy*类层次为*Context*定义了一系列的可供重用的算法或行为。 继承有助于析取出这些算法中的公共功能。

2) 提供了可以替换继承关系的办法： 继承提供了另一种支持多种算法或行为的方法。你可以直接生成一个*Context*类的子类，从而给它以不同的行为。但这会将行为硬行编制到 *Context*中，而将算法的实现与*Context*的实现混合起来,从而使*Context*难以理解、难以维护和难以扩展，而且还不能动态地改变算法。最后你得到一堆相关的类，它们之间的唯一差别是它们所使用的算法或行为。 将算法封装在独立的*Strategy*类中使得你可以独立于其*Context*改变它，使它易于切换、易于理解、易于扩展。

3) 消除了一些*if else*条件语句： *Strategy*模式提供了用条件语句选择所需的行为以外的另一种选择。当不同的行为堆砌在一个类中时，很难避免使用条件语句来选择合适的行为。将行为封装在一个个独立的*Strategy*类中消除了这些条件语句。含有许多条件语句的代码通常意味着需要使用*Strategy*模式。

4) 实现的选择 *Strategy*模式可以提供相同行为的不同实现。客户可以根据不同时间 /空间权衡取舍要求从不同策略中进行选择。

*Strategy*模式缺点：

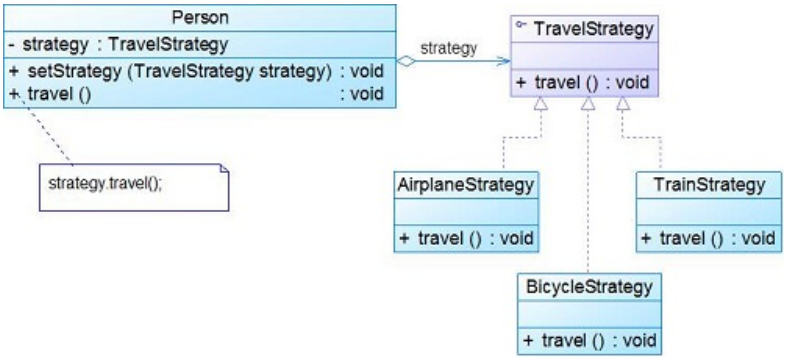
*1)*客户端必须知道所有的策略类，并自行决定使用哪一个策略类：本模式有一个潜在的缺点，就是一个客户要选择一个合适的*Strategy*就必须知道这些*Strategy*到底有何不同。此时可能不得不向客户暴露具体的实现问题。因此仅当这些不同行为变体与客户相关的行为时，才需要使用*Strategy*模式。

*2) Strategy*和*Context*之间的通信开销： 无论各个*ConcreteStrategy*实现的算法是简单还是复杂，它们都共享*Strategy*定义的接口。因此很可能某些 *ConcreteStrategy*不会都用到所有通过这个接口传递给它们的信息；简单的*ConcreteStrategy*可能不使用其中的任何信息！ 这就意味着有时*Context*会创建和初始化一些永远不会用到的参数。如果存在这样问题，那么将需要在*Strategy*和*Context*之间更进行紧密的耦合。

*3)*策略模式将造成产生很多策略类：可以通过使用享元模式在一定程度上减少对象的数量。 增加了对象的数目*Strategy*增加了一个应用中的对象的数目。有时你可以将 *Strategy*实现为可供各*Context*共享的无状态的对象来减少这一开销。任何其余的状态都由 *Context*维护。*Context*在每一次对*Strategy*对象的请求中都将这个状态传递过去。共享的 *Strategy*不应在各次调用之间维护状态。

8.实现[17]

*1)*出行旅游：



代码实现：

```
1. <?php
2. * 策略模式
3. * 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。本模式使得算法可独立于使用它的客户而变化
4. * 出行旅游
5. interface TravelStrategy{
6.     publicfunction travelAlgorithm();
7.  * 具体策略类(ConcreteStrategy)1：乘坐飞机
8. class AirPlanelStrategy implements TravelStrategy {
9.     publicfunction travelAlgorithm(){
10.         "travel by AirPlain""<BR>\r\n"
11.  * 具体策略类(ConcreteStrategy)2：乘坐火车
12. class TrainStrategy implements TravelStrategy {
13.     publicfunction travelAlgorithm(){
14.         "travel by Train""<BR>\r\n"
15.  * 具体策略类(ConcreteStrategy)3：骑自行车
16. class BicycleStrategy implements TravelStrategy {
17.     publicfunction travelAlgorithm(){
18.         "travel by Bicycle""<BR>\r\n"
19.  * 环境类(Context):用一个ConcreteStrategy对象来配置。维护一个对Strategy对象的引用。可定义一个接口来让Strategy访问它的数据。
20.  * 算法解决类，以提供客户选择使用何种解决方案：
21. class PersonContext{
22.     private$_strategy = null;
23.     publicfunction __construct(TravelStrategy $travel
24.         $this->_strategy = $travel
25.     publicfunction setTravelStrategy(TravelStrategy $travel
26.         $this->_strategy = $travel
27.     publicfunction travel(){
28.         return$this->_strategy ->travelAlgorithm();
29. // 乘坐火车旅行
30. $person PersonContext( TrainStrategy());
31. $person->travel();
32. // 改骑自行车
```

33. $person->setTravelStrategy(BicycleStrategy());
34. $person->travel();
35. ?>

```
<?php
/**
* 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。本模式使得算法可独立于使用它的客户而变化
*
*/


/**
* 出行旅游
*
*
*/
interface TravelStrategy{
        public function travelAlgorithm();
}


/**
 * 具体策略类(ConcreteStrategy)1：乘坐飞机
 */
class AirPlanelStrategy implements TravelStrategy {
        public function travelAlgorithm(){
                echo "travel by AirPlain", "<BR>\r\n";
        }
}


/**
 * 具体策略类(ConcreteStrategy)2：乘坐火车
 */
class TrainStrategy implements TravelStrategy {
        public function travelAlgorithm(){
                echo "travel by Train", "<BR>\r\n";
        }
}

/**
 * 具体策略类(ConcreteStrategy)3：骑自行车
 */
class BicycleStrategy implements TravelStrategy {
        public function travelAlgorithm(){
                echo "travel by Bicycle", "<BR>\r\n";
        }
}



/**
 *
 * 环境类(Context):用一个ConcreteStrategy对象来配置。维护一个对Strategy对象的引用。可定义一个接口来让Strategy访问它的数据。
 * 算法解决类，以提供客户选择使用何种解决方案：
 */
class PersonContext{
```

```
        private $_strategy = null;

        public function __construct(TravelStrategy $travel){
                $this->_strategy = $travel;
        }
        /**
        * 旅行
        */
        public function setTravelStrategy(TravelStrategy $travel){
                $this->_strategy = $travel;
        }
        /**
        * 旅行
        */
        public function travel(){
                return $this->_strategy ->travelAlgorithm();
        }
}

// 乘坐火车旅行
$person = new PersonContext(new TrainStrategy());
$person->travel();

// 改骑自行车
$person->setTravelStrategy(new BicycleStrategy());
$person->travel();

?>
```

*2)*排序策略:某系统提供了一个用于对数组数据进行操作的类，该类封装了对数组的常见操作，

如查找数组元素、对数组元素进行排序等。现以排序操作为例，使用策略模式设计该数组操作类，

使得客户端可以动态地更换排序算法，可以根据需要选择冒泡排序或选择排序或插入排序，

也能够灵活地增加新的排序算法。

9.与其他相关模式[18]

1）状态模式*[19]*

策略模式和其它许多设计模式比较起来是非常类似的。策略模式和状态模式最大的区别就是策略模式只是的条件选择只执行一次，而状态模式是随着实例参数（对象实例的状态）的改变不停地更改执行模式。换句话说，策略模式只是在

对象初始化的时候更改执行模式，而状态模式是根据对象实例的周期时间而动态地改变对象实例的执行模式。

通过环境类状态的个数来决定是使用策略模式还是状态模式。

策略模式的环境类自己选择一个具体策略类，具体策略类无须关心环境类状态模式的环境类由于外在因素需要放进一个具体状态中，

以便通过其方法实现状态的切换环境类和状态类之间存在一种双向的关联关系。

使用策略模式时，客户端需要知道所选的具体策略是哪一个，而使用状态模式时，客户端无须关心具体状态，环境

> 类的状态会根据用户的操作自动转换。
>
> 如果系统中某个类的对象存在多种状态，不同状态下行为有差异，而且这些状态之间可以发生转换时使用状态模式
>
> 如果系统中某个类的某一行为存在多种实现方式，而且这些实现方式可以互换时使用策略模式
>
> *2*）简单工厂[20]的区别：点击打开链接[21]
>
> 工厂模式是创建型模式 ，它关注对象创建，提供创建对象的接口. 让对象的创建与具体的使用客户无关。
> 策略模式是对象行为型模式 ，它关注行为和算法的封装 。它定义一系列的算法,把每一个算法封装起来, 并且使它们可相互替换。使得算法可独立于使用它的客户而变化
>
> 用我们上面提到旅行的例子：
> 我们去旅行。策略模式的做法：有几种方案供你选择旅行，选择火车好呢还是骑自行车，完全有客户自行决定去构建旅行方案（比如你自己需要去买火车票，或者机票）。而工厂模式是你决定哪种旅行方案后，不用关注这旅行方案怎么给你创建，也就是说你告诉我方案的名称就可以了，然后由工厂代替你去构建具体方案（工厂代替你去买火车票）。
>
> 上面的例子里面*client*代码：
> *$person = new PersonContext(new TrainStrategy());*
> *$person->travel();*
> 我们看到客户需要自己去创建具体旅行（*new TrainStrategy()*）实例。传递的是具体实例。
> 而工厂模式你只要告诉哪种旅行就可以了，不是传递一个具体实例，而是一个标识（旅行方案标识）。

10.总结与分析[22]

> *1*）策略模式[23]是一个比较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割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的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。
>
> *2*）在策略模式[24]中，应当由客户端自己决定在什么情况下使用什么具体策略角色。
> 策略模式[25]仅仅封装算法，提供新算法插入到已有系统中以及老算法从系统中“退休”的方便，策略模式并不决定在何时使用何种算法，算法的选择由客户端来决定。这在一定程度上提高了系统的灵活性，但是客户端需要理解所有具体策略类之间的区别，以便选择合适的算法，这也是策略模式的缺点之一，在一定程度上增加了客户端的使用难度。

转载指明原文出处：hguisu[26] 设计模式 (十八) 策略模式Strategy（对象行为型）[27] http://blog.csdn.net/hguisu/article/details/7558249[28]

Links

1. javascript:void(0);
2. http://blog.csdn.net/hguisu/article/details/7558249
3. http://blog.csdn.net/hguisu/article/details/7558249
4. http://lib.csdn.net/base/datastructure
5. http://blog.csdn.net/hguisu/article/details/7558249
6. http://blog.csdn.net/hguisu/article/details/7558249
7. http://blog.csdn.net/hguisu/article/details/7558249
8. http://blog.csdn.net/hguisu/article/details/7558249
9. http://blog.csdn.net/hguisu/article/details/7558249