

# 策略模式

在策略模式（**Strategy Pattern**）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。

在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 **context** 对象。策略对象改变 **context** 对象的执行算法。

## 介绍

- 意图：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。
- 主要解决：在有多种算法相似的情况下，使用 **if...else** 所带来的复杂和难以维护。
- 何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。
- 如何解决：将这些算法封装成一个一个的类，任意地替换。
- 关键代码：实现同一个接口。

应用实例： 1、诸葛亮的锦囊妙计，每一个锦囊就是一个策略。 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。 3、**JAVA AWT** 中的 **LayoutManager**。

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。

缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

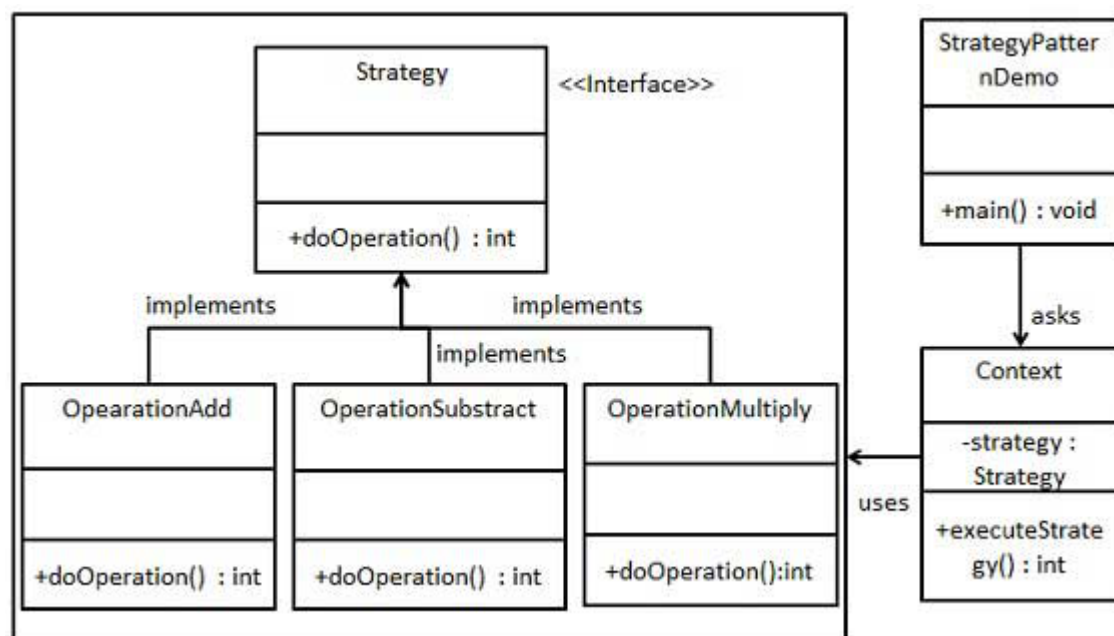
使用场景： 1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在許多行为中选择一种行为。 2、一个系统需要动态地在几种算法中选择一种。 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

注意事项：如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。

## 实现

我们将创建一个定义活动的 **Strategy** 接口和实现了 **Strategy** 接口的实体策略类。**Context** 是一个使用了某种策略的类。

**StrategyPatternDemo**，我们的演示类使用 **Context** 和策略对象来演示 **Context** 在它所配置或使用的策略改变时的行为变化。



## 步骤 1

创建一个接口。

### Strategy.java

```
public interface Strategy {
    public int doOperation(int num1, int num2);
}
```

## 步骤 2

创建实现接口的实体类。

### OperationAdd.java

```
public class OperationAdd implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}
```

### OperationSubtract.java

```
public class OperationSubtract implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 - num2;
    }
}
```

### OperationMultiply.java

```
public class OperationMultiply implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 * num2;
    }
}
```

### 步骤 3

创建 *Context* 类。

#### Context.java

```
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }

    public int executeStrategy(int num1, int num2){
        return strategy.doOperation(num1, num2);
    }
}
```

### 步骤 4

使用 *Context* 来查看当它改变策略 *Strategy* 时的行为变化。

#### StrategyPatternDemo.java

```
public class StrategyPatternDemo {
    public static void main(String[] args) {
        Context context = new Context(new OperationAdd());
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationSubtract());
        System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationMultiply());
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
    }
}
```

### 步骤 5

执行程序，输出结果：

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

☐ 空对象模式

模板模式 ☐



1 篇笔记

☐ 写笔记



与状态模式的比较

状态模式的类图和策略模式类似，并且都是能够动态改变对象的行为。但是状态模式是通过状态转移来改变 *Context* 所组合的 *State* 对象，而策略模式是通过 *Context* 本身的决策来改变组合的 *Strategy* 对

象。所谓的状态转移，是指 **Context** 在运行过程中由于一些条件发生改变而使得 **State** 对象发生改变，注意必须要是在运行过程中。

状态模式主要是用来解决状态转移的问题，当状态发生转移了，那么 **Context** 对象就会改变它的行为；而策略模式主要是用来封装一组可以互相替代的算法族，并且可以根据需要动态地去替换 **Context** 使用的算法。

---