链滴

# 《Head First 设计模式》：策略模式

作者：jingqueyimu
原文链接：https://ld246.com/article/1590584765399
来源网站：链滴

# 正文

## 一、定义

策略模式定义了算法族，分别封装起来，让它们之间可以相互替换，此模式让算法的变化独立于使用法的客户。

**要点：**

- 策略模式把系统中会变化的部分抽出来封装。

## 二、实现步骤

### 1、创建策略接口

```
/**
 * 策略接口
 */
public interface Strategy {

    /**
     * 执行策略行为
     */
    public void perform();
}
```

### 2、创建策略接口的实现类

### （1）策略实现类 A

```
/**
 * 策略实现类A
 */
public class StrategyImplA implements Strategy {

    /**
     * A策略行为
     */
    @Override
    public void perform() {
        System.out.println("perform A...");
    }
}
```

### （2）策略实现类 B

```
/**
 * 策略实现类B
 */
public class StrategyImplB implements Strategy {

    /**
     * B策略行为
     */
    @Override
    public void perform() {
        System.out.println("perform B...");
    }
}
```

## 3、在使用策略的类中，声明并使用接口类型的策略变量

```
/**
 * 策略使用者
 */
public class StrategyUser {

    /**
     * 声明接口类型的策略变量
     */
    private Strategy strategy;

    /**
     * 通过构造实例化策略
     */
    public StrategyUser(Strategy strategy) {
        this.strategy = strategy;
    }

    /**
     * 执行策略使用者的行为
```

```
    */
    public void doBehavior() {
        // do something...

        // 使用策略
        strategy.perform();

        // do something...
    }
}
```

## 4、通过实例化不同的策略实现类，来改变使用者的行为

```
public class Test {

    public static void main(String[] args) {
        // 使用策略A
        StrategyUser userA = new StrategyUser(new StrategyImplA());
        userA.doBehavior();
        // 使用策略B
        StrategyUser userB = new StrategyUser(new StrategyImplB());
        userB.doBehavior();
    }
}
```

# 三、举个栗子

## 1、背景

Joe 上班的公司做了一套相当成功的模拟鸭子游戏：SimUDuck。游戏中会出现各种鸭子，鸭子的种及属性如下：

- 种类：绿头鸭、红头鸭、橡皮鸭、诱饵鸭。
- 属性：外观、游泳行为、飞行行为、呱呱叫行为（叫声）。

不同种类的鸭子所对应的属性如下：

- 绿头鸭：绿头鸭的外观、会游泳、会飞行、呱呱叫。
- 红头鸭：红头鸭的外观、会游泳、会飞行、呱呱叫。
- 橡皮鸭：橡皮鸭的外观、会游泳（漂浮）、不会飞行、吱吱叫。
- 诱饵鸭：诱饵鸭的外观、会游泳（漂浮）、不会飞行、不会叫。

## 2、要点

- 由于不同种类的鸭子可能具有不同的飞行行为、呱呱叫行为，因此，可以使用策略模式把这两种行抽出来。

## 3、实现

## （1）创建行为接口

```
/**
 * 飞行行为接口
 */
public interface FlyBehavior {

    public void fly();
}

/**
 * 呱呱叫行为接口
 */
public interface QuackBehavior {

    public void quark();
}
```

## （2）实现行为接口

```
/**
 * 用翅膀飞行
 */
public class FlyWithWings implements FlyBehavior {

    @Override
    public void fly() {
        System.out.println("I'm flying!");
    }
}

/**
 * 不会飞行
 */
public class FlyNoWay implements FlyBehavior {

    @Override
    public void fly() {
        System.out.println("I can't flying!");
    }
}

/**
 * 呱呱叫
 */
public class Quack implements QuackBehavior {

    @Override
    public void quark() {
        System.out.println("Quack");
    }
}
```

```java
/**
 * 吱吱叫
 */
public class Squeak implements QuackBehavior {

    @Override
    public void quark() {
        System.out.println("Squeak");
    }
}

/**
 * 不会叫
 */
public class MuteQuack implements QuackBehavior {

    @Override
    public void quark() {
        System.out.println("<<Silence>>");
    }
}
```

### （3）创建鸭子抽象类，并使用行为接口

```java
/**
 * 鸭子抽象类
 */
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }

    /**
     * 外观
     */
    public abstract void display();

    /**
     * 游泳行为
     */
    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }

    /**
     * 飞行行为
     */
    public void performFly() {
        flyBehavior.fly();
    }
```

```
    /**
     * 呱呱叫行为
     */
    public void performQuark() {
        quackBehavior.quark();
    }
}
```

### （4）创建鸭子子类，并指定具体的行为实现

```
/**
 * 绿头鸭
 */
public class MallardDuck extends Duck {

    public MallardDuck() {
        // 指定具体的飞行行为
        flyBehavior = new FlyWithWings();
        // 指定具体的呱呱叫行为
        quackBehavior = new Quack();
    }

    @Override
    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}

/**
 * 红头鸭
 */
public class RedHeadDuck extends Duck {

    public RedHeadDuck() {
        // 指定具体的飞行行为
        flyBehavior = new FlyWithWings();
        // 指定具体的呱呱叫行为
        quackBehavior = new Quack();
    }

    @Override
    public void display() {
        System.out.println("I'm a red head duck");
    }
}

/**
 * 橡皮鸭
 */
public class RubberDuck extends Duck {

    public RubberDuck() {
        // 指定具体的飞行行为
        flyBehavior = new FlyNoWay();
```

```
        // 指定具体的呱呱叫行为
        quackBehavior = new Squeak();
    }

    @Override
    public void display() {
        System.out.println("I'm a rubber duck");
    }
}

/**
 * 诱饵鸭
 */
public class DecoyDuck extends Duck {

    public DecoyDuck() {
        // 指定具体的飞行行为
        flyBehavior = new FlyNoWay();
        // 指定具体的呱呱叫行为
        quackBehavior = new MuteQuack();
    }

    @Override
    public void display() {
        System.out.println("I'm a decoy duck");
    }
}
```

### （5）测试

```
public class Test {

    public static void main(String[] args) {
        // 绿头鸭
        MallardDuck mallardDuck = new MallardDuck();
        mallardDuck.performFly();
        mallardDuck.performQuark();
        // 红头鸭
        RedHeadDuck redHeadDuck = new RedHeadDuck();
        redHeadDuck.performFly();
        redHeadDuck.performQuark();
        // 橡皮鸭
        RubberDuck rubberDuck = new RubberDuck();
        rubberDuck.performFly();
        rubberDuck.performQuark();
        // 诱饵鸭
        DecoyDuck decoyDuck = new DecoyDuck();
        decoyDuck.performFly();
        decoyDuck.performQuark();
    }
}
```