策略模式:

前言:

> 作者：韩数
>
> Github：https://github.com/hanshuaikang
>
> 时间:2019-01-26
>
> JDK版本：1.8

定义:

> 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。本模式使得算法可独立于使用它的客户而变化。

适用范围:

> 1.在一个系统中，有很多相似的类，而区分这些类的仅仅是不同的行为。那么策略模式可以像电脑主机一样模块化的让一个对象在不同的行为中选择一种一种行为。
>
> 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。

优缺点:

> **优点:**
>
> 代码耦合度比较底，相对来说比较灵活一些
>
> 可以避免使用if else多重判断语句
>
> 比较有弹性，可扩展性比较好
>
> **缺点**
>
> 策略类会比较多，之后的代码实战中会发现这个问题
>
> 所有策略类都需要对外暴露

前提引入:

韩数独创之对话流:

老板: 阿呆，你去给我编写一个鸭子类（严重吐槽，请大家不要想歪，本书依靠head frist系列书籍，为了避免读者读书的时候代码和书籍有不同的地方影响理解，故没有修正）

阿呆: 内心戏(不就写个实体类吗。写个Duck类，然后把鸭子外貌，飞，叫这样的特征定义了，方法实现了就OK了,Nice,完美)，老板没问题，保证完成任务！

a week has later.... 阿呆信心满满的把写好的Duck类交给了老板。

老板: 不错，写的不错，哎呀，可是，我突然又想要一只橡皮鸭，这只鸭子，不会飞，吱吱叫，我小时候最喜欢的玩具，这样吧，你去写写这个啥橡皮鸭吧。

阿呆：（内心戏: 这橡皮鸭，这，跟我上次写的那个鸭子不是一个品种啊，怎么还吱吱叫，鸭子不都嘎嘎嘎叫吗，算了，不就是再写一个类继承Duck类吗，把fly，quack，display这三个方法覆盖重写了就好了，Nice，完美，我简直是一个天才！）老板没问题，保证完成任务！

a week has later.... 阿呆信心满满的把写好的RubberDuck类交给了老板。

老板: 不错，真好，对了，阿呆呀，我那个侄女，她喜欢那个绿毛鸭，会飞，咕咕叫，头上长绿毛的那种，你看能写么？

阿呆：(内心戏：MMP，略) 老板没问题，保证完成任务！

a week has later...

老板: 那个黑天鸭...

阿呆：(内心戏：emmmmmp) 老板没问题，保证完成任务！

老板: 那个七小天鸭...

阿呆：(内心戏：emmmmmp) 老板没问题，保证完成任务！

老板: 那个派大鸭...

阿呆：(内心戏：emmmmmp) 老板没问题，保证完成任务！

a year has later...

阿呆：卒

这么玩儿下去肯定不行，只通过继承，必然可以完成老板的要求，万一有一万只不同品种的鸭子，不敢往下想了，而且Duck是所有类的父类，这要是Duck改一点点，想到后面还有几万个Duck的孩子要改，不禁倒吸一口凉气，这是，阿呆的弟弟二呆出场了，说：

这世间鸭子千千万，不过数种，记得我之前给你讲那个电脑主机的故事么，把所有零件设计成可拆卸更换的模块

只留下那个大家通用的模块不要动，在鸭子身上就是游泳，哪种鸭子不会游泳？你说，其他的，飞呀，叫什么的，我们单独分离出来，最后老板要啥鸭子，我们给他组装一下不就得了。

此时，设计模式中一句宝典浮出水面，那就是：**分离变和不变的部分。**

大家听了不禁啧啧称赞，纷纷叹道妙呀，妙呀，真是妙啊！

代码实战：

焕然一新后的鸭子类：

```
public abstract class Duck {

    /*
     * 面向超类编程，主类Duck只保留所有鸭子通用不变的特征比如游泳
     * 变化的部分单独封装，提高代码的弹性，避免因为单一的向下继承
     * 造成的代码的灵活性降低，避免过于耦合情况的发生。
     */

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }


    //定义set方法，可以动态的设定鸭子飞行的行为
    public void setFlyBehavior (FlyBehavior fb) {
        flyBehavior = fb;
    }

    //定义set方法，可以动态的设定鸭子飞行的行为
    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    //鸭子的外表，这里定义为抽象方法，父类只做声明，不负责实现
    abstract void display();

    //鸭子的行为，Duck不适合实现，交给相应的模块实现。
    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }


    //所有鸭子都会游泳
    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

可能会有人不太懂，FlyBehavior flyBehavior; QuackBehavior quackBehavior;是什么意思，你想啊，虽然显卡,CPU，音响，内存条都模块化了，但是也总的留个插头方便接入不是。

定义飞行行为的接口，为啥是接口呢？不是类，面向接口（超类）编程，可以更好的利用面向对象中的多态，第二个也可以提高程序相互调用中的安全性。提高程序的灵活性，可扩展性。

```
public interface FlyBehavior {
    public void fly();
}
```

同理叫声接口：

```
public interface QuackBehavior {
    public void quack();
}
```

比如嘎嘎叫的鸭子，我们就定义一个Quack类实现QuackBehavior的接口，并编写quack方法的实现为嘎嘎叫，吱吱叫的鸭子，我们就定义一个Squeak类实现QuackBehavior的接口，并编写quack方法的实现为吱吱叫，等等，咕咕叫，喔喔叫，哇我叫，等等等等等，你开心就好。飞的行为同理。

```
/**
 *
 * 定义鸭子叫声是嘎嘎嘎的行为
 *
 */


public class Quack implements QuackBehavior {
    public void quack() {
        System.out.println("嘎嘎嘎");
    }
}



/**
 *
 * @author hansu
 * 定义鸭子吱吱叫的行为
 *
 */


public class Squeak implements QuackBehavior {
    public void quack() {
        System.out.println("吱吱吱");
    }
}
```

```
/**
 *
 * 定义鸭子不会飞的行为
 *
 *
 */
```

```
public class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println("I can't fly");
    }
}



/***
 *
 * 定义鸭子是会飞的行为
 *
 */

public class FlyWithWings implements FlyBehavior {
    public void fly() {
        System.out.println("我会飞!哈哈哈");
    }
}
```

好了，现在模块是写好了，可是，我们怎么样编写鸭子的子类把这些模块装上去呢？二呆缓缓说，急啥，且听我娓娓道来。

哟，要是想把这些模块来组装

那你就要深入进去它的心房

把模块放入构造器中

变成一把直接就上膛的手枪

yo，freestyle

比如橡皮鸭的特征是吱吱叫，不会飞，身子是橡皮做的，于是就把FlyNoWay，Squeak模块组装一下，然后只需实现一下父类Duck的display方法就会得到一只崭新的完全满足甲方要求的橡皮鸭了！

如下：

```
/***
 *
 * demo1：橡皮鸭，特征，不会飞，吱吱叫
 *
 */

public class RubberDuck extends Duck {

    public RubberDuck() {
        /*
         * 注:因为RubberDuck继承Duck类，所有Duck类中定义的
         * flyBehavior和quackBehavior可以直接赋值
         */
        //定义橡皮鸭不会飞的行为
        flyBehavior = new FlyNoWay();
```

```
        //定义橡皮鸭吱吱叫的行为
        quackBehavior = new Squeak();
    }

    public void display() {
        System.out.println("我是一个橡皮鸭，我的身体是橡皮做哒");
    }
}
```

编写测试代码Text:

```
public class Text {

    public static void main(String[] args) {
        Text t = new Text();
        t.rubberDuckDemoText();

        System.out.println("\n现在有请活的鸭子闪亮登场！\n");

        t.liveDuckDemoText();
    }



    public void rubberDuckDemoText() {

        RubberDuck rubberDuck = new RubberDuck();
        rubberDuck.display();
        rubberDuck.performFly();
        rubberDuck.performQuack();




    }

    public void liveDuckDemoText() {

        LiveDuck liveDuck = new LiveDuck();
        liveDuck.display();
        liveDuck.performFly();
        liveDuck.performQuack();




    }


}
```

，

Out:

我是一个橡皮鸭，我的身体是橡皮做哒 I can't fly 吱吱吱

现在有请活的鸭子闪亮登场！

我是一只活鸭子 我会飞!哈哈哈 嘎嘎嘎

最后，二呆和老板幸福的生活在了一起。

实战总结：

在这里大家就会发现了，虽然这样的确比继承单一Duck类重写方法方便高效了很多，但是如果鸭子特征超级多的话，也需要编写超级多的行为类，同时，每个行为类都必须是可实例化的，这针对某些情况来说并不太适合，但是，设计模式有二十七种呢，更不要说其他设计模式了，更是多到数不胜数，所以在合适的情况下选择合适的设计模式可以显著提高我们代码的效率和质量，这点是毋庸置疑的。

写在最后：

欢迎大家给小星星，您的星星是我写下去的不竭动力！

源码部分请移步本人Github下载：

Github地址：

Github：https://github.com/hanshuaikang/design-pattern-java

参考资料：

菜鸟教程:http://www.runoob.com/design-pattern/strategy-pattern.html

Head frist of 设计模式