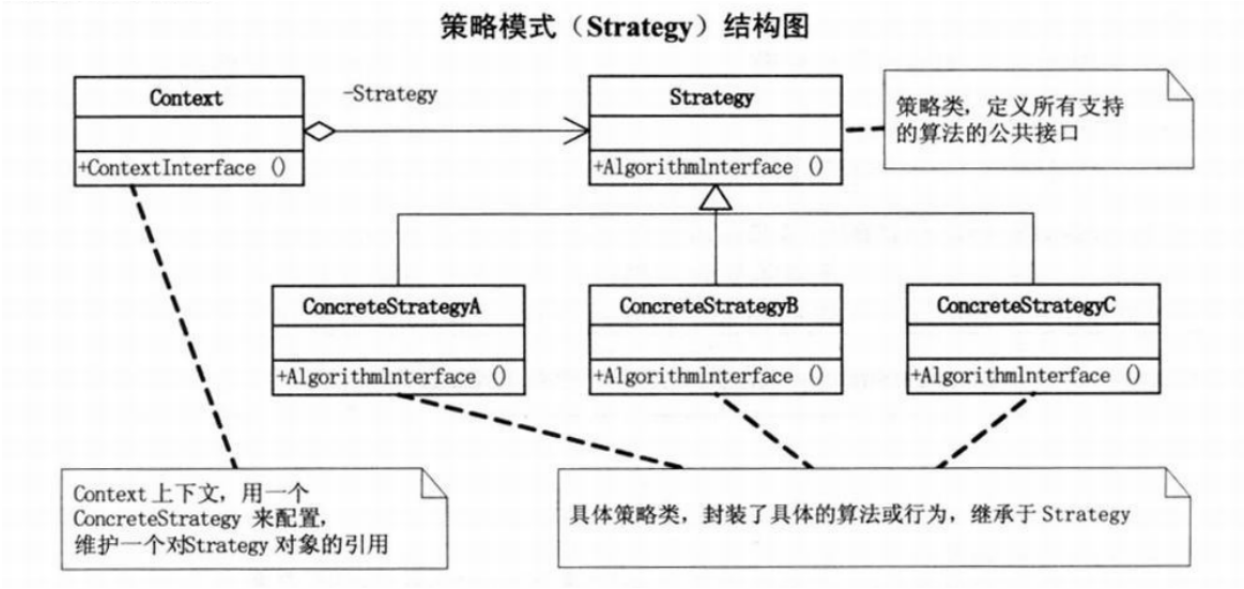
一、什么是策略模式

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数。

策略模式定义和封装了一系列的算法，它们是可以相互替换的，也就是说它们具有共性，而它们的共性就体现在策略接口的行为上，另外为了达到最后一句话的目的，也就是说让算法独立于使用它的客户而独立变化，我们需要让客户端依赖于策略接口。

一种很简单的解释，在我们的开发过程中，经常会遇到大量的if...else或者switch...case语句，当这些语句在开发中只是为了起到分流作用，这些分流和业务逻辑无关，那么这个时候就可以考虑用策略模式。

二、策略模式的结构



Context 与 Strategy 之间是聚合关系，即群体与个体的关系

这个模式涉及到三个角色：

1. 上下文环境(Context)角色：持有一个Strategy的引用。
2. 抽象策略(Strategy)角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
3. 具体策略(ConcreteStrategy)角色：包装了相关的算法或行为

三、策略模式的应用场景

举一个例子，商场搞促销--打8折，满200送50，满1000送礼物，这种促销就是策略。

再举一个例子，dota里面的战术，玩命四保一，三伪核体系，推进体系，大招流体系等，这些战术都是一种策略。

应用场景：

1、 **多个类只区别在表现行为不同，可以使用Strategy模式，在运行时动态选择具体要执行的行为。**

2、 需要在不同情况下使用不同的策略(算法)，或者策略还可能在未来用其它方式来实现。

3、 对客户隐藏具体策略(算法)的实现细节，彼此完全独立。

四、策略模式的优缺点

优点：

1、结构清晰，把策略分离成一个个单独的类「替换了传统的 if else」

2、代码耦合度降低，安全性提高「各个策略的细节被屏蔽」

缺点：

1、客户端必须要知道所有的策略类，否则你不知道该使用那个策略，所以策略模式适用于提前知道所有策略的情况下

2、策略类数量增多（每一个策略类复用性很小，如果需要增加算法，就只能新增类）

五、策略模式和简单工厂模式的异同

在上篇文章已经提过了，传送地址：深入理解设计模式（二）：02 简单工厂模式.note

六、策略模式的实现

Strategy类，定义所有支持的算法的公共接口

```
// Strategy类，定义所有支持的算法的公共接口
public interface Strategy {
   // 算法方法
   public void AlgorithmInterface();
}
```

ConcreteStrategy，封装了具体的算法或行为，继承于Strategy

```
public class ConcreteStrategy1 implements Strategy{
    @Override
    public void AlgorithmInterface() {
        System.out.println("使用了算法A解决问题。");
    }
}

public class ConcreteStrategy2 implements Strategy{
    @Override
    public void AlgorithmInterface() {
        System.out.println("使用了算法B解决问题。");
    }
}
```

Context，用一个ConcreteStrategy来配置，维护一个对Strategy对象的引用

```
public class Context {
    private Strategy strategy;
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
    public void invokeStrategy() {
        strategy.AlgorithmInterface();
    }
}
```

客户端代码

```
// 简单工厂模式中，使用者只需要传递参数，让工厂类来判断返回的具体对象；
// 在策略模式中，使用者需要自己创建并传递想要使用的对象，然后将该对象作为参数传递进去，通过该对象调用不同的算法。
public class Client {
    public static void main(String[] args) {
        Context contextA = new Context(new ConcreteStrategy1());
        contextA.invokeStrategy();  // 只传递具体对象，对象中方法的调用交给context对象。
        Context contextB = new Context(new ConcreteStrategy2());
        contextB.invokeStrategy();
    }
}
```

七、策略模式和简单工厂模式的结合

改造后的Context

```
public class FactoryAndContext {
    Strategy strategy = null;

    public FactoryAndContext(String option) {
```

```
        switch (option) {
            case "A" :
                strategy = new ConcreteStrategy1();
                break;
            case "B" :
                strategy = new ConcreteStrategy2();
                break;
            default:
                break;
        }
    }

    public void invokeStrategy() {
        strategy.AlgorithmInterface();
    }
}
```

改造后的客户端代码

```
public class Client1 {
    public static void main(String[] args) {
        FactoryAndContext context = new FactoryAndContext("A");
        context.invokeStrategy();
        FactoryAndContext context1 = new FactoryAndContext("B");
        context1.invokeStrategy();
    }
}
```

对比下改造前后的区别不难看出，改造前客户端需要认识两个类，Context和ConcreteStrategy。而策略模式和简单工厂模式结合后，客户端只需要认识一个类Context，降低了耦合性。

八、策略枚举的实现

我们可以使用枚举在一个类中实现以上所有的功能及三种不同的角色，下面看看怎么通过枚举来实现策略模式

```
public enum EnumAndStrategy implements Calculator{
    ADD ("+") {
        @Override
        public int exec(int a, int b) {
            return a + b;
        }
    },
    SUB ("-") {
        @Override
        public int exec(int a, int b) {
```

```
            return a - b;
        }
    };

    private String operational; // 接收运算符

    // 枚举类的构造方法默认是private可以不写
    EnumAndStrategy(String operational) {
        this.operational = operational;
    }

    public String getOperational() {
        return operational;
    }
}

interface Calculator{
    public int exec(int a, int b);
}
```

客户端实现

```
public class Client2 {
    public static void main(String[] args) {
        int exec = EnumAndStrategy.ADD.exec(10, 12);
        System.out.println(exec);
        int exec1 = EnumAndStrategy.SUB.exec(12, 10);
        System.out.println(exec1);
    }
}
```

九、总结

策略模式，实质就是封装了一些算法，让算法可以互相替换，用户可以自由选择这些算法进行操作。策略模式本身理解起来没什么难点，但是在实际应用中其本身主要结合工厂模式一起使用。