

60 | 策略模式（上）：如何避免冗长的if-else/switch分支判断代码？

王争 · 设计模式之美



上两节课中，我们学习了模板模式。模板模式主要起到代码复用和扩展的作用。除此之外，我们还讲到了回调，它跟模板模式的作用类似，但使用起来更加灵活。它们之间的主要区别在于代码实现，模板模式基于继承来实现，回调基于组合来实现。

今天，我们开始学习另外一种行为型模式，策略模式。在实际的项目开发中，这个模式也比较常用。最常见的应用场景是，利用它来避免冗长的 if-else 或 switch 分支判断。不过，它的作用还不止如此。它也可以像模板模式那样，提供框架的扩展点等等。

对于策略模式，我们分两节课来讲解。今天，我们讲解策略模式的原理和实现，以及如何用它来避免分支判断逻辑。下一节课，我会通过一个具体的例子，来详细讲解策略模式的应用场景以及真正的设计意图。

话不多说，让我们正式开始今天的学习吧！

策略模式的原理与实现

策略模式，英文全称是 Strategy Design Pattern。在 GoF 的《设计模式》一书中，它是这样定义的：


Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

翻译成中文就是：定义一族算法类，将每个算法分别封装起来，让它们可以互相替换。策略模式可以使算法的变化独立于使用它们的客户端（这里的客户端代指使用算法的代码）。

我们知道，工厂模式是解耦对象的创建和使用，观察者模式是解耦观察者和被观察者。策略模式跟两者类似，也能起到解耦的作用，不过，它解耦的是策略的定义、创建、使用这三部分。接下来，我就详细讲讲一个完整的策略模式应该包含的这三个部分。

1. 策略的定义


策略类的定义比较简单，包含一个策略接口和一组实现这个接口的策略类。因为所有的策略类都实现相同的接口，所以，客户端代码基于接口而非实现编程，可以灵活地替换不同的策略。示例代码如下所示：

 复制代码

```
1 public interface Strategy {
2     void algorithmInterface();
3 }
4
5 public class ConcreteStrategyA implements Strategy {
6     @Override
7     public void algorithmInterface() {
8         //具体的算法...
9     }
10 }
11
12 public class ConcreteStrategyB implements Strategy {
13     @Override
14     public void algorithmInterface() {
15         //具体的算法...
16     }
17 }
```

2. 策略的创建


因为策略模式会包含一组策略，在使用它们的时候，一般会通过类型（type）来判断创建哪个策略来使用。为了封装创建逻辑，我们需要对客户端代码屏蔽创建细节。我们可以把根据 type 创建策略的逻辑抽离出来，放到工厂类中。示例代码如下所示：

 复制代码

```
1 public class StrategyFactory {
2     private static final Map<String, Strategy> strategies = new HashMap<>();
3
4     static {
5         strategies.put("A", new ConcreteStrategyA());
6         strategies.put("B", new ConcreteStrategyB());
7     }
8
9     public static Strategy getStrategy(String type) {
10         if (type == null || type.isEmpty()) {
11             throw new IllegalArgumentException("type should not be empty.");
12         }
13         return strategies.get(type);
14     }
15 }
```

一般来讲，如果策略类是无状态的，不包含成员变量，只是纯粹的算法实现，这样的策略对象是可以被共享使用的，不需要在每次调用 getStrategy() 的时候，都创建一个新的策略对象。针对这种情况，我们可以使用上面这种工厂类的实现方式，事先创建好每个策略对象，缓存到工厂类中，用的时候直接返回。

相反，如果策略类是有状态的，根据业务场景的需要，我们希望每次从工厂方法中，获得的都是新创建的策略对象，而不是缓存好可共享的策略对象，那我们就需要按照如下方式来实现策略工厂类。

 复制代码

```
1 public class StrategyFactory {
2     public static Strategy getStrategy(String type) {
3         if (type == null || type.isEmpty()) {
4             throw new IllegalArgumentException("type should not be empty.");
5         }
6     }
7 }
```


```
6     if (type.equals("A")) {
7         return new ConcreteStrategyA();
8     } else if (type.equals("B")) {
9         return new ConcreteStrategyB();
10    }
11
12    return null;
13 }
14 }
15
```

3. 策略的使用

刚刚讲了策略的定义和创建，现在，我们再来看一下，策略的使用。

我们知道，策略模式包含一组可选策略，客户端代码一般如何确定使用哪个策略呢？最常见的是运行时动态确定使用哪种策略，这也是策略模式最典型的应用场景。

这里的“运行时动态”指的是，我们事先并不知道会使用哪个策略，而是在程序运行期间，根据配置、用户输入、计算结果等这些不确定因素，动态决定使用哪种策略。接下来，我们通过一个例子来解释一下。

 复制代码

```
1 // 策略接口: EvictionStrategy
2 // 策略类: LruEvictionStrategy、FifoEvictionStrategy、LfuEvictionStrategy...
3 // 策略工厂: EvictionStrategyFactory
4
5 public class UserCache {
6     private Map<String, User> cacheData = new HashMap<>();
7     private EvictionStrategy eviction;
8
9     public UserCache(EvictionStrategy eviction) {
10         this.eviction = eviction;
11     }
12
13     //...
14 }
15
16 // 运行时动态确定，根据配置文件的配置决定使用哪种策略
17 public class Application {
18     public static void main(String[] args) throws Exception {
19         EvictionStrategy evictionStrategy = null;
```

```

20     Properties props = new Properties();
21     props.load(new FileInputStream("./config.properties"));
22     String type = props.getProperty("eviction_type");
23     evictionStrategy = EvictionStrategyFactory.getEvictionStrategy(type);
24     UserCache userCache = new UserCache(evictionStrategy);
25     //...
26 }
27 }
28
29 // 非运行时动态确定，在代码中指定使用哪种策略
30 public class Application {
31     public static void main(String[] args) {
32         //...
33         EvictionStrategy evictionStrategy = new LruEvictionStrategy();
34         UserCache userCache = new UserCache(evictionStrategy);
35         //...
36     }
37 }


```

从上面的代码中，我们也可以看出，“非运行时动态确定”，也就是第二个 Application 中的使用方式，并不能发挥策略模式的优势。在这种应用场景下，策略模式实际上退化成了“面向对象的多态特性”或“基于接口而非实现编程原则”。

如何利用策略模式避免分支判断？

实际上，能够移除分支判断逻辑的模式不仅仅有策略模式，后面我们要讲的状态模式也可以。对于使用哪种模式，具体还要看应用场景来定。策略模式适用于根据不同类型的动态，决定使用哪种策略这样一种应用场景。

我们先通过一个例子来看下，if-else 或 switch-case 分支判断逻辑是如何产生的。具体的代码如下所示。在这个例子中，我们没有使用策略模式，而是将策略的定义、创建、使用直接耦合在一起。

 复制代码

```

1 public class OrderService {
2     public double discount(Order order) {
3         double discount = 0.0;
4         OrderType type = order.getType();
5         if (type.equals(OrderType.NORMAL)) { // 普通订单
6             //...省略折扣计算算法代码

```

```
7     } else if (type.equals(OrderType.GROUPON)) { // 团购订单
8         //...省略折扣计算算法代码
9     } else if (type.equals(OrderType.PROMOTION)) { // 促销订单
10        //...省略折扣计算算法代码
11    }
12    return discount;
13 }
14 }
```

如何来移除掉分支判断逻辑呢？那策略模式就派上用场了。我们使用策略模式对上面的代码重构，将不同类型订单的打折策略设计成策略类，并由工厂类来负责创建策略对象。具体的代码如下所示：

```
1 // 策略的定义
2 public interface DiscountStrategy {
3     double calDiscount(Order order);
4 }
5 // 省略NormalDiscountStrategy、GrouponDiscountStrategy、PromotionDiscountStrategy类
6
7 // 策略的创建
8 public class DiscountStrategyFactory {
9     private static final Map<OrderType, DiscountStrategy> strategies = new HashMap<
10
11     static {
12         strategies.put(OrderType.NORMAL, new NormalDiscountStrategy());
13         strategies.put(OrderType.GROUPON, new GrouponDiscountStrategy());
14         strategies.put(OrderType.PROMOTION, new PromotionDiscountStrategy());
15     }
16
17     public static DiscountStrategy getDiscountStrategy(OrderType type) {
18         return strategies.get(type);
19     }
20 }
21
22 // 策略的使用
23 public class OrderService {
24     public double discount(Order order) {
25         OrderType type = order.getType();
26         DiscountStrategy discountStrategy = DiscountStrategyFactory.getDiscountStrate
27         return discountStrategy.calDiscount(order);
28     }
29 }
```

重构之后的代码就没有了 if-else 分支判断语句了。实际上，这得益于策略工厂类。在工厂类中，我们用 Map 来缓存策略，根据 type 直接从 Map 中获取对应的策略，从而避免 if-else 分支判断逻辑。等后面讲到使用状态模式来避免分支判断逻辑的时候，你会发现，它们使用的是同样的套路。本质上都是借助“查表法”，根据 type 查表（代码中的 strategies 就是表）替代根据 type 分支判断。

但是，如果业务场景需要每次都创建不同的策略对象，我们就要用另外一种工厂类的实现方式了。具体的代码如下所示：

```
1 public class DiscountStrategyFactory {
2     public static DiscountStrategy getDiscountStrategy(OrderType type) {
3         if (type == null) {
4             throw new IllegalArgumentException("Type should not be null.");
5         }
6         if (type.equals(OrderType.NORMAL)) {
7             return new NormalDiscountStrategy();
8         } else if (type.equals(OrderType.GROUPON)) {
9             return new GrouponDiscountStrategy();
10        } else if (type.equals(OrderType.PROMOTION)) {
11            return new PromotionDiscountStrategy();
12        }
13        return null;
14    }
15 }
```

这种实现方式相当于把原来的 if-else 分支逻辑，从 OrderService 类中转移到了工厂类中，实际上并没有真正将它移除。关于这个问题如何解决，我今天先暂时卖个关子。你可以在留言区说说你的想法，我在下一节课中再讲解。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

策略模式定义一族算法类，将每个算法分别封装起来，让它们可以互相替换。策略模式可以使算法的变化独立于使用它们的客户端（这里的客户端代指使用算法的代码）。

策略模式用来解耦策略的定义、创建、使用。实际上，一个完整的策略模式就是由这三个部分组成的。

策略类的定义比较简单，包含一个策略接口和一组实现这个接口的策略类。

策略的创建由工厂类来完成，封装策略创建的细节。

策略模式包含一组策略可选，客户端代码如何选择使用哪个策略，有两种确定方法：编译时静态确定和运行时动态确定。其中，“运行时动态确定”才是策略模式最典型的应用场景。

除此之外，我们还可以通过策略模式来移除 if-else 分支判断。实际上，这得益于策略工厂类，更本质上点讲，是借助“查表法”，根据 type 查表替代根据 type 分支判断。

课堂讨论

今天我们讲到，在策略工厂类中，如果每次都要返回新的策略对象，我们还是需要在工厂类中编写 if-else 分支判断逻辑，那这个问题该如何解决呢？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

本文深入介绍了如何利用策略模式来避免冗长的if-else/switch分支判断代码。通过定义、创建和使用策略类，以及利用策略工厂类来动态确定使用哪种策略，读者可以学习到策略模式的核心概念和实际应用。文章通过清晰的示例和详细的讲解，强调了策略模式的灵活性和可维护性，以及如何通过策略模式来移除if-else分支判断，从而提高代码的可读性和可扩展性。此外，文章还提到了策略模式适用于根据不同类型的动态决定使用哪种策略的应用场景。整体而言，本文为读者提供了一种有效的代码优化思路，对于希望了解策略模式及其实际应用的读者来说，是一篇值得阅读的文章。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (122)

最新 精选



卖火柴的托儿索

2020-04-26

这个策略和多态有什么区别？感觉用法差不多

作者回复: 完全两个东西 一个是模式 一个是技术 一个侧重应用场景 一个侧重代码实现

共 3 条评论 >

👍 8



改名不换人

2020-07-30

请问各位一个问题：如果在spring应用中，Map策略表中的对象应该是new创建的，还是用autowired注入？

作者回复: 要是自己new的, spring容器是不管理的, 也就是你在其他地方无法自动注入

共 3 条评论 >



宁锐

2020-03-20

仍然可以用查表法, 只不过存储的不再是实例, 而是class, 使用时获取对应的class, 再通过反射创建实例

共 22 条评论 >

271



下雨天

2020-03-20

策略模式和工厂模式区别:

工厂模式

1. 目的是创建不同且相关的对象
2. 侧重于"创建对象"
3. 实现方式上可以通过父类或者接口
4. 一般创建对象应该是现实世界中某种事物的映射, 有它自己的属性与方法!

策略模式

1. 目的的实现方便地替换不同的算法类
2. 侧重于算法(行为)实现
3. 实现主要通过接口
4. 创建对象对行为的抽象而非对对象的抽象, 很可能没有属于自己的属性。

共 3 条评论 >

85



Michael

2020-03-20

一般而言Java web开发中我们均使用spring框架, 可以使用运行时自定义注解给具体的策略类打上注解, 将具体的策略类放于spring 容器中, 工厂中注入直接根据类型获取即可. 不实用spring框架的话, 也可以用Java的反射做到获取到具体的策略类



47



zhengyu.nie

2020-05-06

查表策略还是挺常见的，搭配java.util.function各种接口挺好用的。

```
private final Map<String, Function<String, String>> PHONE_FUN_FACTORY =  
    ImmutableMap.of(  
        MessageDestination.LEGAL_PERSON.name(), legalPersonPhoneFunc(),  
        MessageDestination.ACTUAL_CONTROL.name(), actualPersonPhoneFunc()  
    );
```

我比较喜欢建一个StrategyContext类，简单工厂+策略模式。

Context类存在一个成员变量xxx，new StrategyContext(xxx).apply();

然后StrategyContext内部存放一个表，value是一堆function，通过成员变量查表找对应的func.apply即可

共 4 条评论 >

👍 26



攻城拔寨

2020-03-20

策略模式通常跟工厂一起配合使用。

策略侧重如何灵活选择替换，

工厂侧重怎么创建实例



👍 23



Jasongrass

2020-03-23

if else 不是必须要解决的问题，如果放在工厂类中，逻辑比较简单，未来的变动也不会很大，就是 OK 的。

共 5 条评论 >

👍 16



aoe

2020-07-28

原来策略模式的精髓就是：用Map代替冗长的if-else/switch分支判断！

哇！

共 2 条评论 >

👍 15



东征

2020-04-03

仍然使用查表法，表中存类型和class，根据类型获取对象时，使用class动态创建。或者表中存类型和一个创建函数，根据类型获取对象时，获取到创建函数，调用创建函数创建新对象

(java里可以用lambda或者对象代替函数)

共 2 条评论>

 12