**前言**

最近在做代码重构，发现了很多代码的烂味道。其他的不多说，今天主要说说那些又臭又长的if...else要如何重构。

在介绍更更优雅的编程之前，让我们一起回顾一下，不好的if...else代码

**一、又臭又长的if...else**

废话不多说，先看看下面的代码。

public interface IPay {   
 void pay();   
}   
  
@Service   
public class AliaPay implements IPay {   
 @Override   
 public void pay() {   
 System.out.println("===发起支付宝支付===");   
 }   
}   
  
@Service   
public class WeixinPay implements IPay {   
 @Override   
 public void pay() {   
 System.out.println("===发起微信支付===");   
 }   
}   
   
@Service   
public class JingDongPay implements IPay {   
 @Override   
 public void pay() {   
 System.out.println("===发起京东支付===");   
 }   
}   
  
@Service   
public class PayService {   
 @Autowired   
 private AliaPay aliaPay;   
 @Autowired   
 private WeixinPay weixinPay;   
 @Autowired   
 private JingDongPay jingDongPay;   
   
   
 public void toPay(String code) {   
 if ("alia".equals(code)) {   
 aliaPay.pay();   
 } else if ("weixin".equals(code)) {   
 weixinPay.pay();   
 } else if ("jingdong".equals(code)) {   
 jingDongPay.pay();   
 } else {   
 System.out.println("找不到支付方式");   
 }   
 }   
}

PayService类的toPay方法主要是为了发起支付，根据不同的code，决定调用用不同的支付类（比如：aliaPay）的pay方法进行支付。

这段代码有什么问题呢？也许有些人就是这么干的。

试想一下，如果支付方式越来越多，比如：又加了百度支付、美团支付、银联支付等等，就需要改toPay方法的代码，增加新的else...if判断，判断多了就会导致逻辑越来越多？

很明显，这里违法了设计模式六大原则的：开闭原则 和 单一职责原则。

**❝**

开闭原则：对扩展开放，对修改关闭。就是说增加新功能要尽量少改动已有代码。

❞

**❝**

单一职责原则：顾名思义，要求逻辑尽量单一，不要太复杂，便于复用。

❞

那有什么办法可以解决这个问题呢？

**二、消除if...else的锦囊妙计**

**1、使用注解**

代码中之所以要用code判断使用哪个支付类，是因为code和支付类没有一个绑定关系，如果绑定关系存在了，就可以不用判断了。

我们先定义一个注解。

@Retention(RetentionPolicy.RUNTIME)   
@Target(ElementType.TYPE)   
public @interface PayCode {   
  
 String value();   
 String name();   
}

在所有的支付类上都加上该注解

@PayCode(value = "alia", name = "支付宝支付")   
@Service   
public class AliaPay implements IPay {   
  
 @Override   
 public void pay() {   
 System.out.println("===发起支付宝支付===");   
 }   
}   
  
   
@PayCode(value = "weixin", name = "微信支付")   
@Service   
public class WeixinPay implements IPay {   
   
 @Override   
 public void pay() {   
 System.out.println("===发起微信支付===");   
 }   
}   
  
   
@PayCode(value = "jingdong", name = "京东支付")   
@Service   
public class JingDongPay implements IPay {   
   
 @Override   
 public void pay() {   
 System.out.println("===发起京东支付===");   
 }   
}

然后增加最关键的类：

@Service   
public class PayService2 implements ApplicationListener<ContextRefreshedEvent> {   
   
 private static Map<String, IPay> payMap = null;   
   
 @Override   
 public void onApplicationEvent(ContextRefreshedEvent contextRefreshedEvent) {   
 ApplicationContext applicationContext = contextRefreshedEvent.getApplicationContext();   
 Map<String, Object> beansWithAnnotation = applicationContext.getBeansWithAnnotation(PayCode.class);   
   
 if (beansWithAnnotation != null) {   
 payMap = new HashMap<>();   
 beansWithAnnotation.forEach((key, value) ->{   
 String bizType = value.getClass().getAnnotation(PayCode.class).value();   
 payMap.put(bizType, (IPay) value);   
 });   
 }   
 }   
   
 public void pay(String code) {   
 payMap.get(code).pay();   
 }   
}

PayService2类实现了ApplicationListener接口，这样在onApplicationEvent方法中，就可以拿到ApplicationContext的实例。我们再获取打了PayCode注解的类，放到一个map中，map中的key就是PayCode注解中定义的value，跟code参数一致，value是支付类的实例。

这样，每次就可以每次直接通过code获取支付类实例，而不用if...else判断了。如果要加新的支付方法，只需在支付类上面打上PayCode注解定义一个新的code即可。

注意：这种方式的code可以没有业务含义，可以是纯数字，只有不重复就行。

**2、动态拼接名称**

该方法主要针对code是有业务含义的场景。

@Service   
public class PayService3 implements ApplicationContextAware {   
 private ApplicationContext applicationContext;   
 private static final String SUFFIX = "Pay";   
  
 @Override   
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {   
 this.applicationContext = applicationContext;   
 }   
  
 public void toPay(String payCode) {   
 ((IPay) applicationContext.getBean(getBeanName(payCode))).pay();   
 }   
  
 public String getBeanName(String payCode) {   
 return payCode + SUFFIX;   
 }   
}

我们可以看到，支付类bean的名称是由code和后缀拼接而成，比如：aliaPay、weixinPay和jingDongPay。这就要求支付类取名的时候要特别注意，前面的一段要和code保持一致。调用的支付类的实例是直接从ApplicationContext实例中获取的，默认情况下bean是单例的，放在内存的一个map中，所以不会有性能问题。

特别说明一下，这种方法实现了ApplicationContextAware接口跟上面的ApplicationListener接口不一样，是想告诉大家获取ApplicationContext实例的方法不只一种。

**3、模板方法判断**

当然除了上面介绍的两种方法之外，spring的源码实现中也告诉我们另外一种思路，解决if...else问题。

我们先一起看看spring AOP的部分源码，看一下DefaultAdvisorAdapterRegistry的wrap方法

public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {   
 if (adviceObject instanceof Advisor) {   
 return (Advisor) adviceObject;   
 }   
 if (!(adviceObject instanceof Advice)) {   
 throw new UnknownAdviceTypeException(adviceObject);   
 }   
 Advice advice = (Advice) adviceObject;   
 if (advice instanceof MethodInterceptor) {   
 return new DefaultPointcutAdvisor(advice);   
 }   
 for (AdvisorAdapter adapter : this.adapters) {   
 if (adapter.supportsAdvice(advice)) {   
 return new DefaultPointcutAdvisor(advice);   
 }   
 }   
 throw new UnknownAdviceTypeException(advice);   
 }

重点看看supportAdvice方法，有三个类实现了这个方法。我们随便抽一个类看看

class AfterReturningAdviceAdapter implements AdvisorAdapter, Serializable {   
   
 @Override   
 public boolean supportsAdvice(Advice advice) {   
 return (advice instanceof AfterReturningAdvice);   
 }   
   
 @Override   
 public MethodInterceptor getInterceptor(Advisor advisor) {   
 AfterReturningAdvice advice = (AfterReturningAdvice) advisor.getAdvice();   
 return new AfterReturningAdviceInterceptor(advice);   
 }   
}

该类的supportsAdvice方法非常简单，只是判断了一下advice的类型是不是AfterReturningAdvice。

我们看到这里应该有所启发。

其实，我们可以这样做，定义一个接口或者抽象类，里面有个support方法判断参数传的code是否自己可以处理，如果可以处理则走支付逻辑。

public interface IPay {   
 boolean support(String code);   
 void pay();   
}   
  
@Service   
public class AliaPay implements IPay {   
 @Override   
 public boolean support(String code) {   
 return "alia".equals(code);   
 }   
   
 @Override   
 public void pay() {   
 System.out.println("===发起支付宝支付===");   
 }   
}   
   
@Service   
public class WeixinPay implements IPay {   
   
 @Override   
 public boolean support(String code) {   
 return "weixin".equals(code);   
 }   
   
 @Override   
 public void pay() {   
 System.out.println("===发起微信支付===");   
 }   
}   
  
@Service   
public class JingDongPay implements IPay {   
 @Override   
 public boolean support(String code) {   
 return "jingdong".equals(code);   
 }   
   
 @Override   
 public void pay() {   
 System.out.println("===发起京东支付===");   
 }   
}

每个支付类都有一个support方法，判断传过来的code是否和自己定义的相等。

@Service   
public class PayService4 implements ApplicationContextAware, InitializingBean {   
  
 private ApplicationContext applicationContext;   
 private List<IPay> payList = null;   
  
 @Override   
 public void afterPropertiesSet() throws Exception {   
 if (payList == null) {   
 payList = new ArrayList<>();   
 Map<String, IPay> beansOfType = applicationContext.getBeansOfType(IPay.class);   
   
 beansOfType.forEach((key, value) -> payList.add(value));   
 }   
 }   
   
 @Override   
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {   
 this.applicationContext = applicationContext;   
 }   
   
 public void toPay(String code) {   
 for (IPay iPay : payList) {   
 if (iPay.support(code)) {   
 iPay.pay();   
 }   
 }   
 }   
}

这段代码中先把实现了IPay接口的支付类实例初始化到一个list集合中，返回在调用支付接口时循环遍历这个list集合，如果code跟自己定义的一样，则调用当前的支付类实例的pay方法。

**4.策略+工厂模式**

这种方式也是用于code是有业务含义的场景。

* 策略模式定义了一组算法，把它们一个个封装起来, 并且使它们可相互替换。
* 工厂模式用于封装和管理对象的创建，是一种创建型模式。

public interface IPay {  
 void pay();  
}  
  
@Service  
public class AliaPay implements IPay {  
  
 @PostConstruct  
 public void init() {  
 PayStrategyFactory.register("aliaPay", this);  
 }  
  
  
 @Override  
 public void pay() {  
 System.out.println("===发起支付宝支付===");  
 }  
  
}  
  
@Service  
public class WeixinPay implements IPay {  
  
 @PostConstruct  
 public void init() {  
 PayStrategyFactory.register("weixinPay", this);  
 }  
  
 @Override  
 public void pay() {  
 System.out.println("===发起微信支付===");  
 }  
}  
  
@Service  
public class JingDongPay implements IPay {  
  
 @PostConstruct  
 public void init() {  
 PayStrategyFactory.register("jingDongPay", this);  
 }  
  
 @Override  
 public void pay() {  
 System.out.println("===发起京东支付===");  
 }  
}  
  
public class PayStrategyFactory {  
  
 private static Map<String, IPay> PAY\_REGISTERS = new HashMap<>();  
  
  
 public static void register(String code, IPay iPay) {  
 if (null != code && !"".equals(code)) {  
 PAY\_REGISTERS.put(code, iPay);  
 }  
 }  
  
  
 public static IPay get(String code) {  
 return PAY\_REGISTERS.get(code);  
 }  
}  
  
@Service  
public class PayService3 {  
  
 public void toPay(String code) {  
 PayStrategyFactory.get(code).pay();  
 }  
}

这段代码的关键是PayStrategyFactory类，它是一个策略工厂，里面定义了一个全局的map，在所有IPay的实现类中注册当前实例到map中，然后在调用的地方通过PayStrategyFactory类根据code从map获取支付类实例即可。

**5.责任链模式**

这种方式在代码重构时用来消除if...else非常有效。

* 责任链模式：将请求的处理对象像一条长链一般组合起来，形成一条对象链。请求并不知道具体执行请求的对象是哪一个，这样就实现了请求与处理对象之间的解耦。

常用的filter、spring aop就是使用了责任链模式，这里我稍微改良了一下，具体代码如下：

public abstract class PayHandler {  
  
 @Getter  
 @Setter  
 protected PayHandler next;  
  
 public abstract void pay(String pay);  
  
}  
  
@Service  
public class AliaPayHandler extends PayHandler {  
  
  
 @Override  
 public void pay(String code) {  
 if ("alia".equals(code)) {  
 System.out.println("===发起支付宝支付===");  
 } else {  
 getNext().pay(code);  
 }  
 }  
  
}  
  
@Service  
public class WeixinPayHandler extends PayHandler {  
  
 @Override  
 public void pay(String code) {  
 if ("weixin".equals(code)) {  
 System.out.println("===发起微信支付===");  
 } else {  
 getNext().pay(code);  
 }  
 }  
}  
  
@Service  
public class JingDongPayHandler extends PayHandler {  
  
  
 @Override  
 public void pay(String code) {  
 if ("jingdong".equals(code)) {  
 System.out.println("===发起京东支付===");  
 } else {  
 getNext().pay(code);  
 }  
 }  
}  
  
@Service  
public class PayHandlerChain implements ApplicationContextAware, InitializingBean {  
  
 private ApplicationContext applicationContext;  
 private PayHandler header;  
  
  
 public void handlePay(String code) {  
 header.pay(code);  
 }  
  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
 this.applicationContext = applicationContext;  
 }  
  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 Map<String, PayHandler> beansOfTypeMap = applicationContext.getBeansOfType(PayHandler.class);  
 if (beansOfTypeMap == null || beansOfTypeMap.size() == 0) {  
 return;  
 }  
 List<PayHandler> handlers = beansOfTypeMap.values().stream().collect(Collectors.toList());  
 for (int i = 0; i < handlers.size(); i++) {  
 PayHandler payHandler = handlers.get(i);  
 if (i != handlers.size() - 1) {  
 payHandler.setNext(handlers.get(i + 1));  
 }  
 }  
 header = handlers.get(0);  
 }  
}

这段代码的关键是每个PayHandler的子类，都定义了下一个需要执行的PayHandler子类，构成一个链式调用，通过PayHandlerChain把这种链式结构组装起来。

**6、其他的消除if...else的方法**

当然实际项目开发中使用if...else判断的场景非常多，上面只是其中几种场景。下面再列举一下，其他常见的场景。

**1.根据不同的数字返回不同的字符串**

public String getMessage(int code) {   
 if (code == 1) {   
 return "成功";   
 } else if (code == -1) {   
 return "失败";   
 } else if (code == -2) {   
 return "网络超时";   
 } else if (code == -3) {   
 return "参数错误";   
 }   
 throw new RuntimeException("code错误");   
}

其实，这种判断没有必要，用一个枚举就可以搞定。

public enum MessageEnum {   
 SUCCESS(1, "成功"),   
 FAIL(-1, "失败"),   
 TIME\_OUT(-2, "网络超时"),   
 PARAM\_ERROR(-3, "参数错误");   
  
 private int code;   
 private String message;   
  
 MessageEnum(int code, String message) {   
 this.code = code;   
 this.message = message;   
 }   
   
 public int getCode() {   
 return this.code;   
 }   
  
 public String getMessage() {   
 return this.message;   
 }   
   
 public static MessageEnum getMessageEnum(int code) {   
 return Arrays.stream(MessageEnum.values()).filter(x -> x.code == code).findFirst().orElse(null);   
 }   
}

再把调用方法稍微调整一下

public String getMessage(int code) {   
 MessageEnum messageEnum = MessageEnum.getMessageEnum(code);   
 return messageEnum.getMessage();   
}

完美。

**2.集合中的判断**

上面的枚举MessageEnum中的getMessageEnum方法，如果不用java8的语法的话，可能要这样写

public static MessageEnum getMessageEnum(int code) {   
 for (MessageEnum messageEnum : MessageEnum.values()) {   
 if (code == messageEnum.code) {   
 return messageEnum;   
 }   
 }   
 return null;   
}

对于集合中过滤数据，或者查找方法，java8有更简单的方法消除if...else判断。

public static MessageEnum getMessageEnum(int code) {   
 return Arrays.stream(MessageEnum.values()).filter(x -> x.code == code).findFirst().orElse(null);   
}

**3.简单的判断**

其实有些简单的if...else完全没有必要写，可以用三目运算符代替，比如这种情况：

public String getMessage2(int code) {   
 if(code == 1) {   
 return "成功";   
 }   
 return "失败";   
}

改成三目运算符：

public String getMessage2(int code) {   
 return code == 1 ? "成功" : "失败";   
}

修改之后代码更简洁一些。

**4.spring中的判断**

对于参数的异常，越早被发现越好，在spring中提供了Assert用来帮助我们检测参数是否有效。

public void save(Integer code，String name) {   
 if(code == null) {  
 throw Exception("code不能为空");   
 } else {  
 if(name == null) {  
 throw Exception("name不能为空");   
 } else {  
 System.out.println("doSave");  
 }  
 }  
 }

如果参数非常多的话，if...else语句会很长，这时如果改成使用Assert类判断，代码会简化很多：

public String save2(Integer code，String name) {   
 Assert.notNull(code,"code不能为空");   
 Assert.notNull(name,"name不能为空");   
 System.out.println("doSave");  
 }

当然，还有很多其他的场景可以优化if...else，我再这里就不一一介绍了，感兴趣的朋友可以给我留言，一起探讨和研究一下。