设计模式学习笔记（二）工厂模式、模板模式和策略模式的混合使用

一、工厂模式

工厂模式又叫做工厂方法模式，是一种 **创建型** 设计模式，一般是在父类中提供一个创建对象的方法，允许子类决定实例化对象的类型。

1.1 工厂模式介绍

工厂模式是Java 中比较常见的一种设计模式，实现方法是定义一个统一创建对象的接口，让其子类自己决定去实例化那个工厂类，解决不同条件下创建不同实例的问题。工厂方法模式在实际使用时会和其他的设计模式一起结合，而不是单独使用。比如在Lottery 项目中奖品的发放就是工厂+模板+策略模式。

1.2 工厂模式实现

举个例子，比如要实现不同奖品的发放业务，有优惠券、实体商品和会员电子卡这些奖品，那么我们可以定义这三种类型奖品的接口：

序号	类型	接口	描述
1	优惠券	`CouponResult sendCoupon(String uId, String couponNumber, String uuid)`	返回优惠券信息（对象类型）
2	实体商品	`Boolean deliverGoods(DeliverReq req)`	返回是否发送实体商品（布尔类型）
3	爱奇艺会员电子卡	`void grantToken(String bindMobileNumber, String cardId)`	执行发放会员卡（空类型）

从上表可以看出，不同的奖品有不同的返回类型需求，那么我们该如何处理这些数据，并对应返回呢？常规思路可以想到通过统一的入参 `AwardReq` ，出参 `AwardRes` ，外加上一个 `PrizeController` 来具体实现这些奖品的数据处理任务：

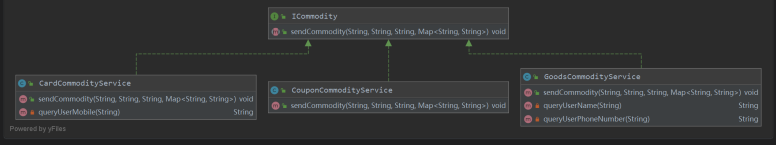
```
AwardReq
AwardRes
PrizeController
```

但是这样势必会造成 `PrizeController` 这个类中逻辑判断过多，后期如果要继续扩展奖品类型，是非常困难和麻烦的。比如可以看看 `PrizeController` 中的代码：

```
public class PrizeController {

    private Logger logger = LoggerFactory.getLogger(PrizeController.class);

    public AwardRes AwardToUser(AwardReq awardReq) {
        String reqJson = JSON.toJSONString(awardReq);
        AwardRes awardRes = null;

        try {
            logger.info("奖品发放开始{}。 awardReq:{}", awardReq.getuId(), reqJson);
            if (awardReq.getAwardType() == 1) {
                CouponService couponService = new CouponService();
                CouponResult couponResult = couponService.sendCoupon(awardReq.getuId(),
awardReq.getAwardNumber(), awardReq.getBizId());
                if ("0000".equals(couponResult.getCode())) {
                    awardRes = new AwardRes(0000, "发放成功");
                } else {
                    awardRes = new AwardRes(0001, "发送失败");
                }
            } else if (awardReq.getAwardType() == 2) {
                GoodsService goodsService = new GoodsService();
                DeliverReq deliverReq = new DeliverReq();
                deliverReq.setUserName(queryUserName(awardReq.getuId()));
                deliverReq.setUserPhone(queryUserPhoneNumber(awardReq.getuId()));
                deliverReq.setSku(awardReq.getAwardNumber());
                deliverReq.setOrderId(awardReq.getBizId());
                deliverReq.setConsigneeUserName(awardReq.getExtMap().get("consigneeUserName"));
                deliverReq.setConsigneeUserPhone(awardReq.getExtMap().get("consigneeUserPhone"));
                deliverReq.setConsigneeUserAddress(awardReq.getExtMap().get("consigneeUserAddress"));
                Boolean isSuccess = goodsService.deliverGoods(deliverReq);
                if (isSuccess) {
                    awardRes = new AwardRes(0000, "发放成功");
                } else {
                    awardRes = new AwardRes(0001, "发送失败");
                }
            } else {
                IQiYiCardService iQiYiCardService = new IQiYiCardService();
                iQiYiCardService.grantToken(queryUserPhoneNumber(awardReq.getuId()),
awardReq.getAwardNumber());
                awardRes = new AwardRes(0000, "发送成功");
            }
            logger.info("奖品发放完成{}。", awardReq.getuId());
        } catch (Exception e) {
            logger.error("奖品发放失败{}。req:{}", awardReq.getuId(), reqJson, e);
            awardRes = new AwardRes(0001, e.getMessage());
        }
        return awardRes;

    }
```

在 `PrizeController` 的类中，我们发现使用了很多简单的 `if-else` 判断。而且整个代码看起来很长，对于后续迭代和扩展会造成很大的麻烦，因此在考虑设计模式的单一职责原则后，我们可以利用工厂模式对奖品处理返回阶段进行抽取，让每个业务逻辑在自己所属的类中完成。

首先，我们从业务逻辑中发现无论是那种奖品，都需要发送，因此可以提炼出统一的入参接口和发送方法： `ICommodity` 、 `sendCommodity(String uId, String awardId, String bizId, Map<String, String> extMap)` 入参内容包括 `用户Id` ， `奖品Id` ， `yewuId` ， `扩展字段` 进行实现业务逻辑的统一，具体如下UML图



然后，我们可以在具体的奖品内部实现对应的逻辑。

最后创建奖品工厂 `StoreFactory` ，可以通过奖品类型判断来实现不同奖品的服务，如下所示：

```
public class StoreFactory {

    public ICommodity getCommodityService(Integer commodityType) {
        if (null == commodityType) {
```

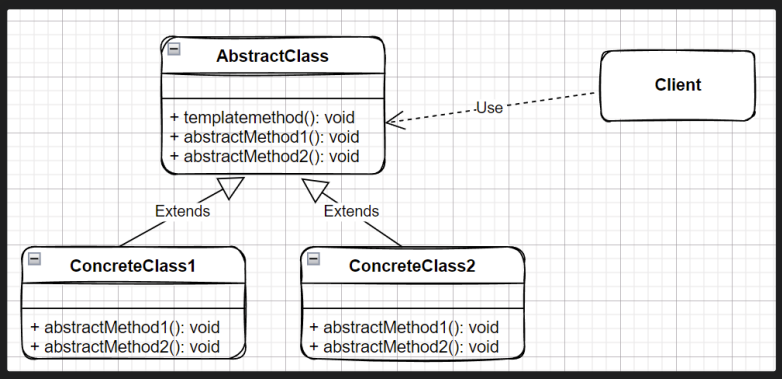
```
            return null;
        }
        if (1 == commodityType) {
            return new CouponCommodityService();
        }
        if (2 == commodityType) {
            return new GoodsCommodityService();
        }
        if (3 == commodityType) {
            return new CardCommodityService();
        }
        throw new RuntimeException("不存在的商品服务类型");
    }
}
```

二、模板模式（Template pattern）

模板模式的核心就是：通过一个公开定义抽象类中的方法模板，让继承该抽象类的子类重写方法实现该模板。它是一种 **类行为型模式**。

2.1 模板模式介绍

定义一个操作的大致框架，然后将具体细节放在子类中实现。也就是通过在抽象类中定义模板方法，让继承该子类具体实现模板方法的细节。我们来看看模板模式的UML图：



- `AbstractClass`：抽象类，在抽象类中定义了一系列基本操作，这些操作可以是具体的，也可以是抽象的，每一个基本操作对应算法的一个步骤，在其子类中可以重定义或实现这些步骤。同时在抽象类中实现了一个模板方法`TemplateMethod()`，用于定义一个算法的框架。
- `ConcreteClass`：具体子类，实现抽象类中声明的抽象方法，并完成子类特定算法的步骤
- `Client`：客户端，使用模板方法模式

2.2 模板模式实现

举个例子，在爬取不同网页资源并生成对应推广海报业务时，我们会有固定的步骤，如：模拟登录、爬取信息、生成海报。这个时候就可以将流程模板抽离出来，让对应子类去实现具体的步骤。比如爬取微信公众号、淘宝、京东、当当网的网页服务信息。

首先，定义一个抽象类`NetMall`，然后再在该类中定义对应的模拟登录`login`、爬取信息`reptile`、生成海报`createBase`的抽象方法让子类继承。具体代码如下所示：

```
public abstract class NetMall {

    String uId;   // 用户ID
    String uPwd;  // 用户密码

    public NetMall(String uId, String uPwd) {
        this.uId = uId;
        this.uPwd = uPwd;
    }
    // 1.模拟登录
    protected abstract Boolean login(String uId, String uPwd);

    // 2.爬虫提取商品信息(登录后的优惠价格)
    protected abstract Map<String, String> reptile(String skuUrl);

    // 3.生成商品海报信息
    protected abstract String createBase64(Map<String, String> goodsInfo);

    /**
     * 生成商品推广海报
     *
     * @param skuUrl 商品地址(京东、淘宝、当当)
     * @return 海报图片base64位信息
     */
    public String generateGoodsPoster(String skuUrl) {
        if (!login(uId, uPwd)) return null;             // 1. 验证登录
        Map<String, String> reptile = reptile(skuUrl);  // 2. 爬虫商品
        return createBase64(reptile);                   // 3. 组装海报
    }

}
```

接下来以抓取京东网页信息为例实现具体步骤：

```
public class JDNetMall extends NetMall {

    public JDNetMall(String uId, String uPwd) {
        super(uId, uPwd);
    }
    //1.模拟登录
    public Boolean login(String uId, String uPwd) {
        return true;
    }
    //2.网页爬取
    public Map<String, String> reptile(String skuUrl) {
        String str = HttpClient.doGet(skuUrl);
        Pattern p9 = Pattern.compile("(?<=title\\>).*(?=</title)");
        Matcher m9 = p9.matcher(str);
        Map<String, String> map = new ConcurrentHashMap<String, String>();
        if (m9.find()) {
            map.put("name", m9.group());
        }
        map.put("price", "5999.00");
        return map;
    }
```

```
    //3.生成海报
    public String createBase64(Map<String, String> goodsInfo) {
        BASE64Encoder encoder = new BASE64Encoder();
        return encoder.encode(JSON.toJSONString(goodsInfo).getBytes());
    }

}
```

最后进行测试：

```
@Test
public void test_NetMall() {
    NetMall netMall = new JDNetMall("ethan", "******");
    String base64 = netMall.generateGoodsPoster("https://item.jd.com/100008348542.html");
}
```

模板模式主要是提取子类中的核心公共代码，让每个子类对应完成所需的内容即可。

三、策略模式（Strategy Pattern）

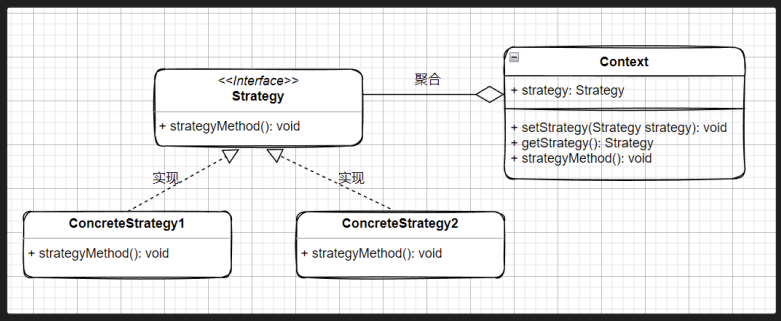
策略模式是一种 **行为类型** 模式，如果在一个系统中有许多类，而区分他们的只是它们的行为，这个时候就可以利用策略模式来进行切换。

3.1 策略模式介绍

在侧率模式中，我们创建表示各种策略的对象和一个行为随着侧率对象改变而改变的 context 对象。

比如诸葛亮的锦囊妙计，每一个锦囊都是一个策略。在业务逻辑中，我们一般是使用具有同类可替代的行为逻辑算法场景，比如，不同类型的交易方式（信用卡、支付宝、微信），生成唯一ID的策略（UUID、雪花算法、Leaf算法）等，我们都可以先用策略模式对其进行行为包装，然后提供给外界进行调用。

注意，如果一个系统中的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。下面来看看对应的UML结构图：



- `Stategy`：抽象策略结构，定义各种不同的算法实现接口，上下文 `Context` 通过这个接口调用不同算法
- `ConcreteStrategy1`、`ConcreteStrategy2`：实现抽象策略定义的接口，提供具体的算法实现
- `Context`：上下文类，也叫环境类，持有策略类的引用，是外界调用策略的接口

3.2 策略模式实现

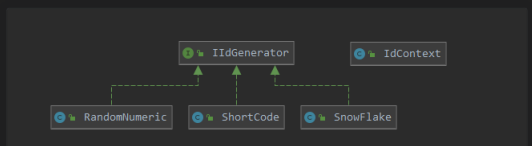
就拿生成唯一ID业务来举例子，比如在雪花算法提出之前，我们一般使用的是UUID 来确认唯一ID。但是如果需要有序的生成ID，这个时候就要考虑一下其他的生成方法，比如雪花、Leaf等算法了。

可能刚开始我们是直接写一个类，在类里面调用UUID算法来生成，但是需要调用其他方法时，我们就必须在这个类里面用 `if-else` 等逻辑判断，然后再转换成另外的算法中。这样的做法和前面提到的工厂模式一样，会提高类之间的耦合度。所以我们可以使用策略模式将这些策略抽离出来，单独实现，防止后期若需要扩展带来的混乱。

首先，定义一个ID生成的接口 `IIdGenerator`

```
public interface IIdGenerator {
    /**
     * 获取ID，目前有三种实现方式
     * 1.雪花算法，主要用于生成单号
     * 2.日期算法，用于生成活动标号类，特性是生成数字串较短，但是指定时间内不能生成太多
     * 3.随机算法，用于生成策略ID
     * @return ID 返回ID
     */
    long nextId();
}
```

让不同生成ID策略实现该接口：



下面是雪花算法的具体实现：

```
public class SnowFlake implements IIdGenerator {

    private Snowflake snowflake;

    @PostConstruct
    public void init() {
        //总共有5位，部署0~32台机器
        long workerId;
        try {
            workerId = NetUtil.ipv4ToLong(NetUtil.getLocalhostStr());
        } catch (Exception e) {
            workerId = NetUtil.getLocalhostStr().hashCode();
        }

        workerId = workerId >> 16 & 31;

        long dataCenterId = 1L;
        snowflake = IdUtil.createSnowflake(workerId, dataCenterId);
    }

    @Override
    public long nextId() {
        return snowflake.nextId();
```

```
    }
}
```

其次还要定义一个ID策略控制类 `IdContext` ,通过外部不同的策略，利用统一的方法执行ID策略计算，如下所示：

```
@Configuration
public class IdContext {

    @Bean
    public Map<Constants.Ids, IIdGenerator> idGenerator(SnowFlake snowFlake, ShortCode shortCode,
RandomNumeric randomNumeric) {
        Map<Constants.Ids, IIdGenerator> idGeneratorMap = new HashMap<>(8);
        idGeneratorMap.put(Constants.Ids.SnowFlake, snowFlake);
        idGeneratorMap.put(Constants.Ids.ShortCode, shortCode);
        idGeneratorMap.put(Constants.Ids.RandomNumeric, randomNumeric);
        return idGeneratorMap;
    }
}
```

所以在最后测试时，直接调用 `idGeneratorMap` 就可以实现不同策略服务的调用：

```
 @Test
 public void init() {
     logger.info("雪花算法策略，生成ID: {}", idGeneratorMap.get(Constants.Ids.SnowFlake).nextId());
     logger.info("日期算法策略，生成ID: {}", idGeneratorMap.get(Constants.Ids.ShortCode).nextId());
     logger.info("随机算法策略，生成ID: {}", idGeneratorMap.get(Constants.Ids.RandomNumeric).nextId());
 }
```

四、三种模式的混合使用

在实际业务开发中，一般是多种设计模式一起混合使用。而工厂模式和策略模式搭配使用就是为了消除 `if-else` 的嵌套，下面就结合工厂模式中的案例来介绍一下：

4.1 策略模式+工厂模式

在第一节中的工厂模式中，我们利用工厂实现不同类型的奖品发放，但是在 `StoreFactory` 中还是有 `if-else` 嵌套的问题：

```
public class StoreFactory {

    public ICommodity getCommodityService(Integer commodityType) {
        if (null == commodityType) {
            return null;
        }
        if (1 == commodityType) {
            return new CouponCommodityService();
        }
        if (2 == commodityType) {
            return new GoodsCommodityService();
        }
        if (3 == commodityType) {
            return new CardCommodityService();
        }
        throw new RuntimeException("不存在的商品
```

这个时候可以利用策略模式消除 `if-else` 语句：

```
public class StoreFactory {
    /**设置策略Map**/
    private static Map<Integer, ICommodity> strategyMap = Maps.newHashMap();

    public static ICommodity getCommodityService(Integer commodityType) {
        return strategyMap.get(commodityType);
    }
    /**提前将策略注入 strategyMap **/
    public static void register(Integer commodityType, ICommodity iCommodity) {
        if (0 == commodityType || null == iCommodity) {
            return;
        }
        strategyMap.put(commodityType, iCommodity);
    }
}
```

在奖品接口中继承 `InitializingBean` ,便于注入策略 `strategyMap`

```
public interface ICommodity extends InitializingBean {

    void sendCommodity(String uId, String commodityId, String bizId, Map<String, String> extMap);
}
```

然后再具体策略实现上注入对应策略：

```
@Component
public class GoodsCommodityService implements ICommodity {

    private Logger logger = LoggerFactory.getLogger(GoodsCommodityService.class);

    private GoodsService goodsService = new GoodsService();

    @Override
    public void sendCommodity(String uId, String commodityId, String bizId, Map<String, String> extMap) {
        DeliverReq deliverReq = new DeliverReq();
        deliverReq.setUserName(queryUserName(uId));
        deliverReq.setUserPhone(queryUserPhoneNumber(uId));
        deliverReq.setSku(commodityId);
        deliverReq.setOrderId(bizId);
        deliverReq.setConsigneeUserName(extMap.get("consigneeUserName"));
        deliverReq.setConsigneeUserPhone(extMap.get("consigneeUserPhone"));
        deliverReq.setConsigneeUserAddress(extMap.get("consigneeUserAddress"));
        Boolean isSuccess = goodsService.deliverGoods(deliverReq);
        if (!isSuccess) {
            throw new RuntimeException("实物商品发送失败");
        }
    }

    private String queryUserName(String uId) {
        return "ethan";
    }

    private String queryUserPhoneNumber(String uId) {
        return "12312341234";
    }

    @Override
    public void afterPropertiesSet() throws Exception {
```

```
        StoreFactory.register(2, this);
    }
}
```

最后进行测试：

```
@SpringBootTest
public class ApiTest {

    private Logger logger = LoggerFactory.getLogger(ApiTest.class);

    @Test
    public void commodity_test() {
        //1.优惠券
        ICommodity commodityService = StoreFactory.getCommodityService(1);
        commodityService.sendCommodity("10001", "sdfsfdsdfsdfs", "1212121212", null);

        //2.实物商品
        ICommodity commodityService1 = StoreFactory.getCommodityService(2);
        Map<String, String> extMap = new HashMap<String, String>();
        extMap.put("consigneeUserName", "ethan");
        extMap.put("consigneeUserPhone", "12312341234");
        extMap.put("consigneeUserAddress", "北京市 海淀区 xxx");
        commodityService1.sendCommodity("10001", "sdfsfdsdfsdfs", "1212121212", extMap);

        //3.第三方兑换卡
        ICommodity commodityService2 = StoreFactory.getCommodityService(3);
        commodityService2.sendCommodity("10001", "SSDIIUIUHJHJ","12312312312",null);

    }
}
```

```
模拟发放优惠券一张：10001,sdfsfdsdfsdfs,1212121212
2022-03-23 16:45:02.351  INFO 174964 --- [           main] c.b.i.d.s.impl.CouponCommodityService    : 请求参数[优惠券] => uId: 10001 commodityId: sdfsfdsdfsdfs bizId: 1212121212 extMap: null
2022-03-23 16:45:02.410  INFO 174964 --- [           main] c.b.i.d.s.impl.CouponCommodityService    : 测试结果[优惠券]: {"code":0,"info":"发放成功"}
模拟发放实物商品：{"consigneeUserAddress":"北京市 海淀区 xxx","consigneeUserName":"ethan","consigneeUserPhone":"12312341234","orderId":"1212121212","sku":"sdfsfdsdfsdfs","userName":"ethan","userPhone":"
模拟发放爱奇艺会员卡：15200101232,12312312312
2022-03-23 16:45:02.424  INFO 174964 --- [           main] c.b.i.d.store.impl.CardCommodityService  : 请求参数[爱奇艺兑换卡] => uId: 10001 commodityId: SSDIIUIUHJHJ bizId: 12312312312 extMap: null
2022-03-23 16:45:02.424  INFO 174964 --- [           main] c.b.i.d.store.impl.CardCommodityService  : 测试结果[爱奇艺兑换卡] : success
```

4.2 策略模式+工厂模式+模板模式

还是以之前的例子，上面我们已经用策略+工厂模式实现了业务，如何将模板模式也应用其中呢？我们先看看核心的 `ICommodity` 接口：

```
public interface ICommodity extends InitializingBean {

    void sendCommodity(String uId, String commodityId, String bizId, Map<String, String> extMap);
}
```

在这个接口中，只有一个 `sendCommodity` 方法，那么如果在具体实现策略的类中，需要不同的实现方法，这个时候我们就可以利用模板模式的思路，将接口换成抽象类：

```
public abstract class AbstractCommodity implements InitializingBean {

    public void sendCommodity(String uId, String commodityId, String bizId, Map<String, String> extMap) {
        //不支持操作异常,继承的子类可以任意选择方法进行实现
        throw new UnsupportedOperationException();
    }

    public String templateTest(String str) {
        throw new UnsupportedOperationException();
    }
}
```

如上，继承的子类方法可以任意实现具体的策略，以优惠券为例：

```
@Component
public class CouponCommodityService extends AbstractCommodity {

    private Logger logger = LoggerFactory.getLogger(CouponCommodityService.class);

    private CouponService couponService = new CouponService();

    @Override
    public void sendCommodity(String uId, String commodityId, String bizId, Map<String, String> extMap) {

        CouponResult couponResult = couponService.sendCoupon(uId, commodityId, bizId);
        logger.info("请求参数[优惠券] => uId: {} commodityId: {} bizId: {} extMap: {}", uId, commodityId,
bizId, JSON.toJSON(extMap));
        logger.info("测试结果[优惠券]: {}", JSON.toJSON(couponResult));
        if (couponResult.getCode() != 0000) {
            throw new RuntimeException(couponResult.getInfo());
        }
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        StoreFactory.register(1, this);
    }
}
```

这样的好处在于，子类可以根据需求在抽象类中选择继承一些方法，从而实现对应需要的功能。

综上，在日常业务逻辑中对于设计模式的使用，并不是非得一定要代码中有设计模式才行，简单的逻辑就用 `if-else` 即可。如果有复杂的业务逻辑，而且也符合对应的设计模式，这样使用模式才能真正够提高代码的逻辑性和可扩展性。

参考资料

《重学Java设计模式》

《大话设计模式》

http://c.biancheng.net/view/1376.html

分类：设计模式

标签：设计模式

2 0

« 上一篇： 信管知识梳理（二）常规信息系统集成技术（网络协议、网络存储技术、网络工程、数据仓库和中间件）

» 下一篇： 设计模式学习笔记（三）简单工厂、工厂方法和抽象工厂之间的区别

posted @ 2022-03-23 17:59 归斯君 阅读(1310) 评论(0) 编辑 收藏 举报