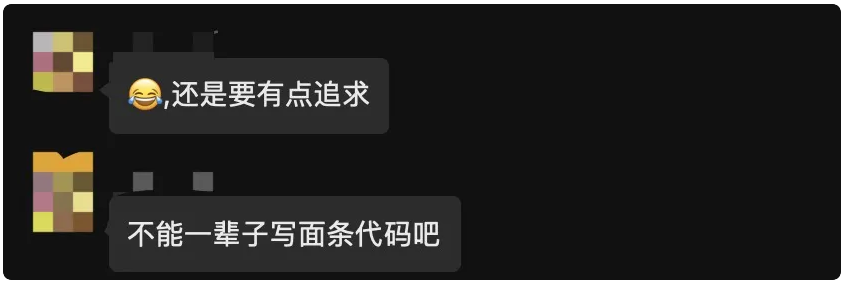
A1. 常见设计模式在支付系统中的应用场景

1. 前言
2. 基本原则
3. 责任链Chain of Responsibility
4. 工厂Factory
5. 建造者Builder
6. 代理Proxy
7. 中介者Mediator
8. 外观Facade
9. 观察者Observer
10. 策略Strategy
11. 状态State
12. 模板方法Template Method
13. 适配器Adapter
14. 装饰器Decorator
15. 迭代器 Iterator
16. 其它一些设计模式
17. 一些容易混淆的点
    17.1. 工厂与建造者
    17.2. 代理、装饰器、中介者
    17.3. 装饰器与适配器
    17.4. 观察者与发布-订阅
18. 混合使用多种设计模式
19. 过度设计
20. 结束语

大家好，我是隐墨星辰，深耕境内/跨境支付架构设计十余年。今天聊聊设计模式。

一个引子，如下：



在大学的时候，就开始翻GoF的设计模式，当时啃了好久，还是不甚了解，后来只好放弃。以至在毕业多年后仍然在写面条式代码，感觉没有什么是if else和for循环搞不定的事。

写多了，也看多了，慢慢开始对设计模式有了一些新的理解。

前段时间和一个朋友聊天，他说他请了一个教练在学羽毛球，我问他有什么心得，他说：“没请教练前，觉得自己也会打羽毛球，不就是挥挥拍子把球接住再打出去吗，请了教练后，才知道握拍、手腕发力等都是有一整套科学理论的，想打好一定要学会应用这些理论。”

软件的设计模式其实也是如此，掌握了编程语言的基本语法，会写if else和for，的确就可以写代码了，但是想写出高水平的代码，离不开设计模式这一理论的熟练应用。

1. 前言

对软件设计模式的理解和应用，基本上可以算做初级研发工程师和高级研发工程师的分水岭。

我在面试时很喜欢问候选人对设计模式的理解，以及实际应用情况，大部分候选人都能回答单例，工厂等，再多问几句，就惨不忍睹。其实也能理解，这些内容对于初学者而言，基本只能靠死记硬背，记不长久。

今天聊聊我理解的设计模式，在支付系统经常用到的场景，以及容易混淆的点，里面讲到的概念可能和一些权威的论述有所出入。

下面的内容全部源自我这么多年所写代码的抽象总结，和以前一样，也不可避免会夹杂一些我个人不成熟的见解，请各位以“取其精华，去其糟粕”的精神辩证地看待此文内容。

2. 基本原则

对于初学者，设计模式里面的很多东西，我们日常其实都在使用，只是没有意识到，也就是所谓的“日用而不自知”，只是有大佬们把这些给总结出来后，我们可以依此做一些刻意练习，以达到在需要的时候能够“信手拈来”。

软件行业有一些流行很广的理论或原则，比如：**低耦合高内聚，扩展开放修改关闭，依赖倒置，单一职责，最少知识原则**等，简单地理解，就是**隔离策略，**内部复杂实现，外部简单调用，更通俗地说：“把复杂留给自己，把简单留给别人”。

回到设计模式，基本上也都符合这些基本原则：比如代理模式、装饰器模式等都扩展了新的能力却不必修改原有代码（扩展开放修改关闭）；外观模式、工厂模式等调用方只需要自己要用什么，不需要知道对方怎么实现（最少知识原则）；责任链模式和策略模式里面每一个处理器或策略只需要处理一个小功能（单一职责），其它的不一一列举。

3. 责任链Chain of Responsibility

责任链模式很简单，简单地理解就是把处理器一个个串起来，每个处理器只关心自己的那一部分实现就可以。

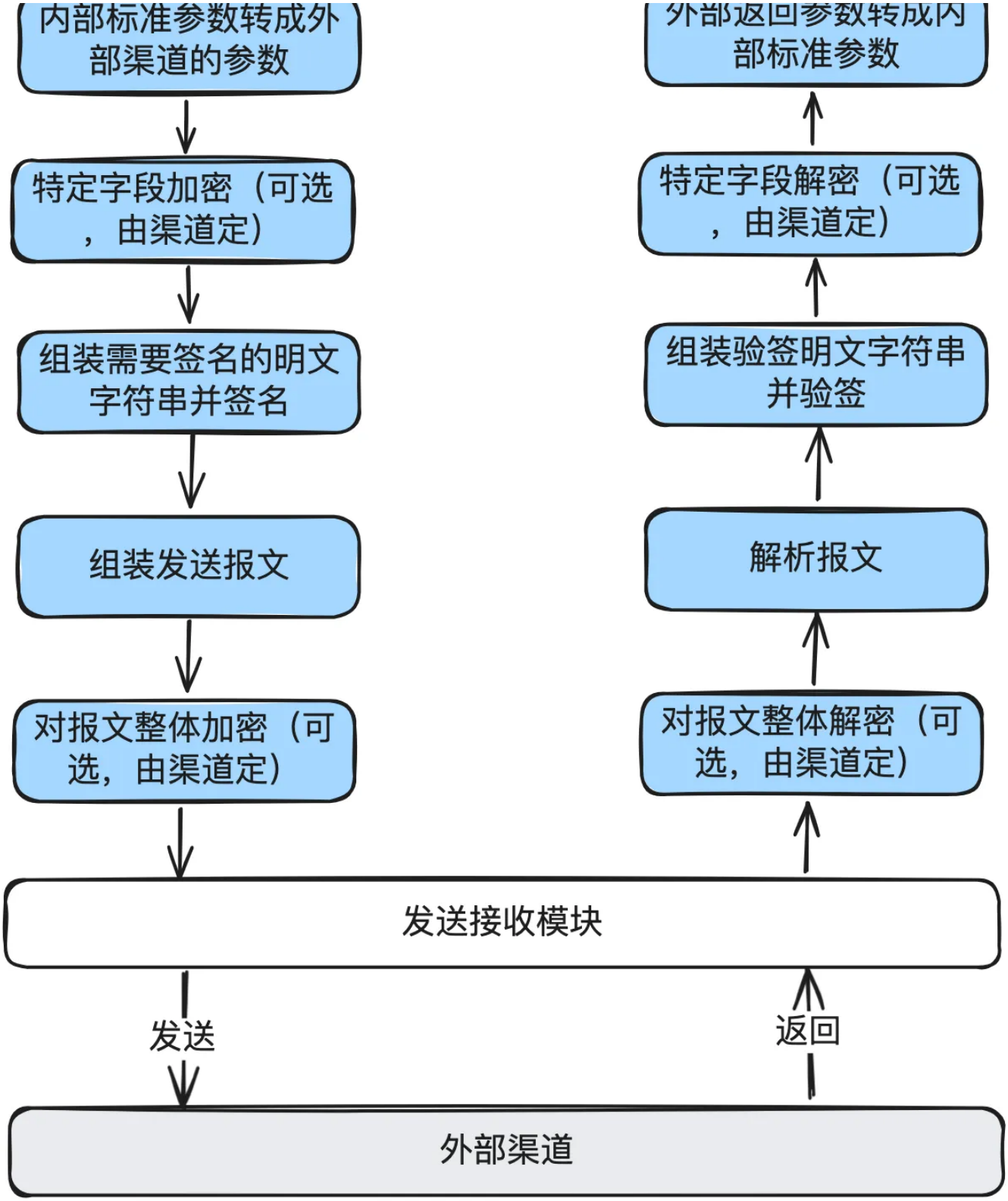
这个很像工厂里面的流水线作业，每个节点（或工人）只做一件事，从终点出来的就是一个成品。

在支付系统中应用也非常广泛，举两个典型场景：

1. **收银台可用支付方式过滤**：平台一共有30种支付方式，每个支付方式有自己的限额，有些特定支付方式只能支持特定业务等，用户请求进来后如何高效过滤出可用支付方式列表?

如果不用责任链，那就写很多for循环和if else，随着支付方式多起来，业务逻辑越来越复杂，后面维护成本必须很高。如果使用责任链，每种判断都实现不同的处理器，把处理器串起来，依次执行完成，结果就出来了。

2. **外部渠道网关实现**：支付系统对接的外部渠道网关，需要做报文转换、签名、报文体组装、外发、解析报文、验签、转换报文这几步。也是一个很好的责任链场景，每一步就相当于一个节点，所有的结点串起来，就完成一个外部接口的处理。



一个简单的示例：

`GatewayHandler` 接口定义了处理逻辑的执行方法。

```java
public interface GatewayHandler {
    void process(GatewayContext context);
}
```

具体的 Handler 实现：

内部参数转外部参数Handler

```java
public class ParameterTransformHandler implements GatewayHandler {
    @Override
    public void process(GatewayContext context) {
        // 实现内部参数到外部参数的转换逻辑
    }
}
```

签名Handler

```java
public class SignatureHandler implements GatewayHandler {
    @Override
    public void process(GatewayContext context) {
        // 不需要签名
        if (!context.isNeedSign()) {
            return;
        }

        context.setSignMessage(sign(context.getSignPlainContext(),
                                    context.getInterfaceInfo().getSignConfig()));
    }

    // 签名方法略
}
```

其它Handler的实现略。

`GatewayHandlerFactory` 是一个工厂类，用于构建处理责任链。

```java
public class GatewayHandlerFactory {
    private static final List<GatewayHandler> handlers = new ArrayList<>();

    static {
        handlers.add(new ParameterTransformHandler());
        handlers.add(new EncryptionHandler());
        handlers.add(new SignatureHandler());
        // 添加其他handler
        ... ...
    }

    public List<GatewayHandler> getHandlers() {
        // 添加其他handler
        return handlers;
    }
}
```

执行Service

```java
public class GatewayServiceImpl implements GatewayService {
    @Override
    public GatewayResponse process(GatewayRequest request) {
        // 转成网关的上下文
        GatewayContext context = buildGatewayContext(request);

        // 获取责任链，依次执行
        List<GatewayHandler> handlers = GatewayHandlerFactory.getHandlers();
        handlers.forEach(handler -> handler.process(context));

        // 转换返回
        return convertResponse(context.getResponseParam);
    }

    // 其它代码略
    ... ...
}
```

4. 工厂Factory

工厂模式顾名思义就是生产东西，**使用者只管向工厂说我要什么，至于怎么生产，那是工厂的事**。

严格意义上的工厂模式区分为工厂方法，抽象工厂。在工厂方法中，只定义创建对象的方法，再由具体的工厂类来实现，而抽象工厂就更为复杂一些。

但我们日常研发中使用更多的是非常简单的工厂，也就是所谓的简单工厂，比如在前面责任链模式中介绍的创建链的工厂类`GatewayHandlerFactory`，由工厂负责构建处理器责任链，使用者只调用getHandlers()来获得处理器列表，不关心生产的细节。

```java
public class GatewayHandlerFactory {
    private static final List<GatewayHandler> handlers = new ArrayList<>();

    static {
        handlers.add(new ParameterTransformHandler());
        handlers.add(new EncryptionHandler());
        handlers.add(new SignatureHandler());
        // 添加其他handler
        ... ...
    }

    public List<GatewayHandler> getHandlers() {
        // 添加其他handler
        return handlers;
    }
}
```

5. 建造者Builder

建造者模式就是通过提供一个建造者类，逐步构建复杂对象的各个部分，并在最后调用build()方法时返回最终构建的对象。

更简单的理解，就是把PO类的setXXX方法给封装起来，使代码更简洁，尤其是当类的属性很多的时候。

常用的实现方式是直接在PO类的上面使用@lombok.Builder注解。

示例：

```java
@Data
@Builder
@ToString
public class GatewayResponse {
    private boolean success;
    private String errorMessage;
    private String responseContext;
}
```

使用：

```java
public class GatewayServiceImpl implements GatewayService {
    private GatewayResponse buildResponse(String assembledResponseMessage) {
        return GatewayResponse.builder()
                .success(true)
                .responseContext(assembledResponseMessage)
                .build();
    }

    // 其它方法略
}
```

6. 代理Proxy

代理模式就是提供一个代理对象来替换被访问的目标对象，但**不改变被代理对象的业务能力**，只是可以增加访问控制、打印日志等通用功能。

这个没有太多好说的，因为太常见。比如Spring的AOP，业务代码只需要管业务逻辑，而打印摘要日志（方法名、出入参、耗时等）、事务管理等使用切面搞定。还有rpc框架提供的远程代理，业务能力没有变，但加入了访问控制等能力。

示例：

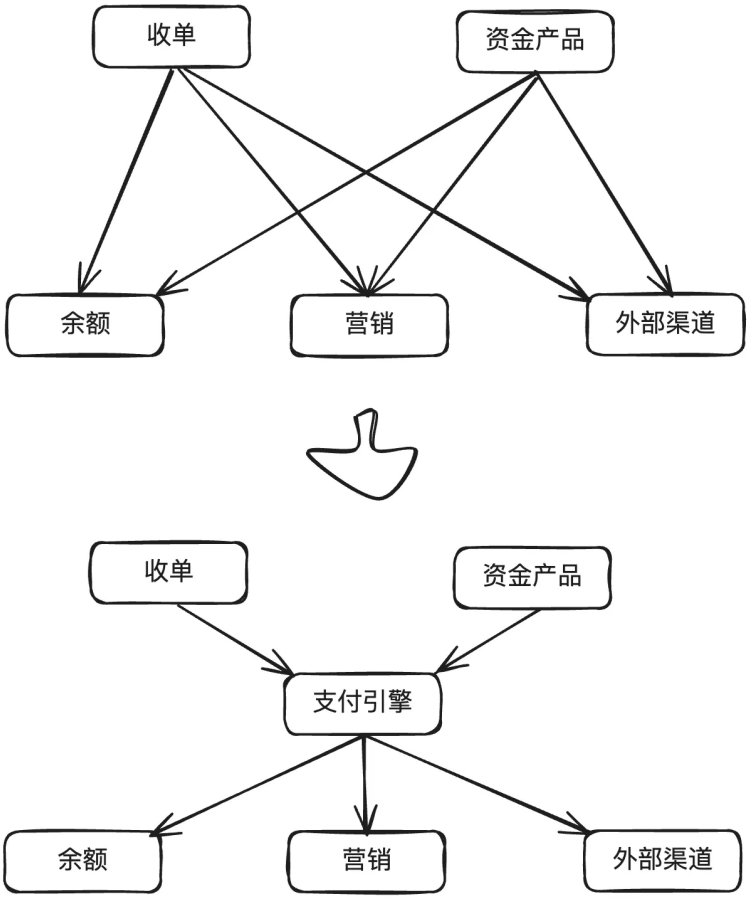
```java
public class GatewayServiceImpl implements GatewayService {
    @Override
    // 服务入口统一打印摘要日志
    @DigestLog
    public GatewayResponse process(GatewayRequest request) {
        // 业务代码略
    }
}
```

7. 中介者Mediator

中介者模式通过一个中介对象来封装一系列对象的访问。

一个典型的案例就是支付引擎。支付引擎做为一个中介者，封装了多个支付工具，比如余额、营销（比如红包，随机减等）、外部支付渠道等。

上游的收单、资金产品等都需要用到支付工具，就只需要调用支付引擎就行。如果没有支付引擎这个中介者，上游的多个应用都需要对接下游的多个支付工具。



8. 外观Facade

外观模式简单理解就是设计接口，也就是所谓的**接口与实现分离，客户端只需要知道接口定义，不需要知道具体实现**。

像rpc框架一般都会定义一组接口，客户端只需要引用这些接口定义，由服务端进行具体实现。内部很多服务也基本是先设计接口，再实现服务。

最简单的示例如下：

定义接口：

```java
public interface GatewayService {
    GatewayResponse process(GatewayRequest request);
}
```

实现：

```java
public class GatewayServiceImpl implements GatewayService {
    @Override
    @DigestLog
    public GatewayResponse process(GatewayRequest request) {
        // 业务代码略
    }
}
```

9. 观察者Observer

观察者模式从名字上已经非常清楚，也就是当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知。

这里面的重点仍然是**职责单一原则**，被观察者只需要处理自己的业务，然后把自己的变更通知到各观察者，再由各观察者根据这些变更，去处理自己的业务。

支付系统应用的场景非常多，比如为提高运算速度，支付方式被缓存在内存中，运营人员如果在后台关闭了某个支付方式，那就需要通知所有的机器刷新缓存，就可以用到观察者模式。

手机客户端程序里面按钮事件处理，也是典型的观察者模式。

示例：

Java

```java
// 观察者接口
public interface PaymentStateListen {
    void stateChanged(Order order);
}

// 被观察者接口
public interface OrderService {
    void registerPaymentStateListen(PaymentStateListen listen);
    void unregisterPaymentStateListen(PaymentStateListen listen);
    void notifyPaymentStateListens(Order order);
}

// 具体被观察者
public class OrderServiceImpl implements OrderService {
    private List<PaymentStateListen> paymentStateListens;

    public OrderServiceImpl() {
        paymentStateListens = new ArrayList<>();
    }

    public void processOrder(Order order) {
        // 业务处理
        process(order);
        // 通知监听者
        notifyPaymentStateListens(order);
    }

    @Override
    public void registerPaymentStateListen(PaymentStateListen listen) {
        paymentStateListens.add(listen);
    }

    @Override
    public void unregisterPaymentStateListen(PaymentStateListen listen) {
        PaymentStateListens.remove(listen);
    }

    @Override
    public void notifyPaymentStateListens(Order order) {
        // 通知全部注册的监听者
        handlers.forEach(handler -> handler.stateChanged(order));
    }
}

// 观察者具体实现，以及注册观察者等代码略
```

10. 策略Strategy

策略模式最简单的理解就是用于**替换if else**。简单的代码，通常使用if else就已经足够，不需要杀鸡用牛刀，但是当每个if里面的代码逻辑都非常复杂，且预见后面可能还会增加，那就推荐使用策略模式。

这也是**职责单一原则**和**扩展开放修改关闭**的体现，一个策略只搞定一个if分支的逻辑就够，需要新增业务时，只需要新增策略，不需要修改原有代码。

支付系统使用策略模式也比较多，比如在跨境场景下接了很多外部渠道，各国的渠道接口对于请求号的要求是不一样的，有些要求使用32字符串，有些要求16位，有些要求固守字符开头等。这个时候就可以使用策略模式，预先制定好策略，不同的渠道配置不同的策略，根据不同的策略生成不同格式的请求号。

下面是一个简单的示例：

```java
// 策略接口
public interface BuildRequestNoStrategy {
    // 根据通用的requestNo生成特定格式的requestNo
    String build(String requestNo, String ext);
    // 策略名
    String name();
}

// 16位长度的策略
public class Len16Strategy implements BuildRequestNoStrategy {
    @Override
    public String build(String requestNo, String ext) {
        // 拼接代码略
        return buildLen16(requestNo);
    }
}

// 32位且固定前缀策略
public class PrefixLen32 implements BuildRequestNoStrategy {
    @Override
    public String build(String requestNo, String ext) {
        // 拼接代码略
        return prefixLen32(requestNo, ext);
    }
}

// 其它策略代码略
... ...

// 策略工厂
public class BuildRequestNoStrategyFactory {
    // 初始化代码略
    ... ...

    // 获取策略
    public BuildRequestNoStrategy getStrategy(String name) {
        return strategyMap.get(name);
    }
}

// 使用
public class PayServiceImpl implements PayService {
    public Order pay(Order order) {
        // 其它代码略
        ... ...
```

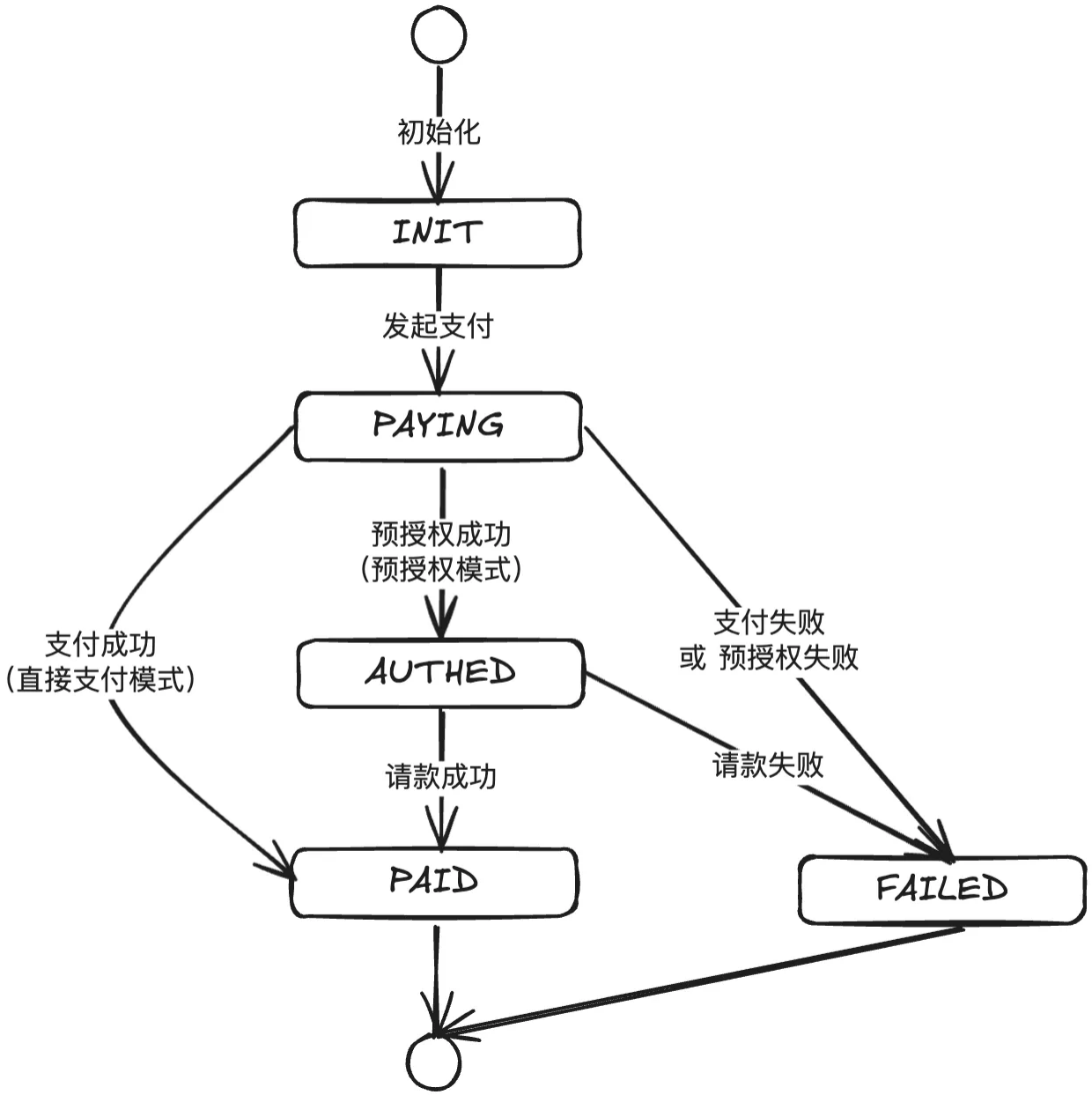
```
46
47          // 设置渠道请求号
48          order.setChannelRequestNo(buildChannelRequestNo(order.getChannel()
    ));
49
50           // 其它代码略
51          ... ...
52      }
53
54      // 根据渠道配置的策略生成不同的格式的单号
55      private String buildChannelRequestNo(Channel channel) {
56          // 生成默认请求号
57          String requestNo = buildDefaultRequestNo();
58
59          // 获取策略
60          RequestNoStrategyInfo strategyInfo = channel.getRequestNoStrategyI
    nfo();
61          if (null == strategyInfo) {
62              return requestNo;
63          }
64
65          BuildRequestNoStrategy strategy = BuildRequestNoStrategyFactory.ge
    tStrategy(
66              strategyInfo.getName());
67
68          if (null == strategy) {
69              return requestNo;
70          }
71
72          return strategy.build(requestNo, strategyInfo.getExt());
73      }
74  }
```

11. 状态State

状态模式可以简单理解为有限状态机的实现，就是把和状态相关的操作封装到一个独立的状态机类里面，而不是耦合在业务代码中。

支付系统里面有很多单据，每种单据都有自己的状态，比如支付单有“初始化”，“支付中”，“预授权成功”，“请款中”，“成功”，“失败”共6种状态，如果不使用状态模式，那就直接在订单类里面使用String来定义，状态的推进全部写if else或case when来实现，会导致这部分的代码很容易出错，要不就写得很复杂。

下面是一个典型的状态机设计：



从状态机设计图中可以看到，状态之间的变迁是通过事件来驱动的，比如INIT到PAYING，事件就是发起支付。

如果不使用状态机，示例：

```java
if (status.equals("PAYING") {
    status = "SUCCESS";
} else if (...) {
    ...
}
```

或者：

```java
class OrderDomainService {
    public void notify(PaymentNotifyMessage message) {
        PaymentModel paymentModel = loadPaymentModel(message.getPaymentId());
        // 直接设置状态
        paymentModel.setStatus(PaymentStatus.valueOf(message.status);
        // 其它业务处理
        ... ...
    }
}
```

或者：

```java
 public void transition(Event event) {
     switch (currentState) {
         case INIT:
             if (event == Event.PAYING) {
                 currentState = State.PAYING;
             } else if (event == Event.SUCESS) {
                 currentState = State.SUCESS;
             } else if (event == Event.FAIL) {
                 currentState = State.FAIL;
             }
             break;
             // Add other case statements for different states and events
     }
 }
```

如果换成状态机模式，如下：

定义状态基类

```java
/**
 * 状态基类
 */
public interface BaseStatus {
}
```

定义事件基类

```java
/**
 * 事件基类
 */
public interface BaseEvent {
}
```

定义“状态–事件对”，**指定的状态只能接受指定的事件**

```java
/**
 * 状态事件对，指定的状态只能接受指定的事件
 */
public class StatusEventPair<S extends BaseStatus, E extends BaseEvent> {
    /**
     * 指定的状态
     */
    private final S status;
    /**
     * 可接受的事件
     */
    private final E event;

    public StatusEventPair(S status, E event) {
        this.status = status;
        this.event = event;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof StatusEventPair) {
            StatusEventPair<S, E> other = (StatusEventPair<S, E>)obj;
            return this.status.equals(other.status) && this.event.equals(other.event);
        }
        return false;
    }

    @Override
    public int hashCode() {
        // 这里使用的是google的guava包。com.google.common.base.Objects
        return Objects.hash(status, event);
    }
}
```

定义状态机

```java
/**
 * 状态机
 */
public class StateMachine<S extends BaseStatus, E extends BaseEvent> {
    private final Map<StatusEventPair<S, E>, S> statusEventMap = new HashMap<>();

    /**
     * 只接受指定的当前状态下，指定的事件触发，可以到达的指定目标状态
     */
    public void accept(S sourceStatus, E event, S targetStatus) {
        statusEventMap.put(new StatusEventPair<>(sourceStatus, event), targetStatus);
    }

    /**
     * 通过源状态和事件，获取目标状态
     */
    public S getTargetStatus(S sourceStatus, E event) {
        return statusEventMap.get(new StatusEventPair<>(sourceStatus, event));
    }
}
```

定义支付的状态机。注：**支付、退款等不同的业务状态机是独立的。**

Java

```java
/**
 * 支付状态机
 */
public enum PaymentStatus implements BaseStatus {

    INIT("INIT", "初始化"),
    PAYING("PAYING", "支付中"),
    PAID("PAID", "支付成功"),
    FAILED("FAILED", "支付失败"),
    ;

    // 支付状态机内容
    private static final StateMachine<PaymentStatus, PaymentEvent> STATE_MACHINE = new StateMachine<>();
    static {
        // 初始状态
        STATE_MACHINE.accept(null, PaymentEvent.PAY_CREATE, INIT);
        // 支付中
        STATE_MACHINE.accept(INIT, PaymentEvent.PAY_PROCESS, PAYING);
        // 支付成功
        STATE_MACHINE.accept(PAYING, PaymentEvent.PAY_SUCCESS, PAID);
        // 支付失败
        STATE_MACHINE.accept(PAYING, PaymentEvent.PAY_FAIL, FAILED);
    }

    // 状态
    private final String status;
    // 描述
    private final String description;

    PaymentStatus(String status, String description) {
        this.status = status;
        this.description = description;
    }

    /**
     * 通过源状态和事件类型获取目标状态
     */
    public static PaymentStatus getTargetStatus(PaymentStatus sourceStatus, PaymentEvent event) {
        return STATE_MACHINE.getTargetStatus(sourceStatus, event);
    }
}
```

定义支付事件。注：**支付、退款等不同业务的事件是不一样的。**

```java
/**
 * 支付事件
 */
public enum PaymentEvent implements BaseEvent {
    // 支付创建
    PAY_CREATE("PAY_CREATE", "支付创建"),
    // 支付中
    PAY_PROCESS("PAY_PROCESS", "支付中"),
    // 支付成功
    PAY_SUCCESS("PAY_SUCCESS", "支付成功"),
    // 支付失败
    PAY_FAIL("PAY_FAIL", "支付失败");

    /**
     * 事件
     */
    private String event;
    /**
     * 事件描述
     */
    private String description;

    PaymentEvent(String event, String description) {
        this.event = event;
        this.description = description;
    }
}
```

在支付单模型中声明状态和根据事件推进状态的方法：

```java
/**
 * 支付单模型
 */
public class PaymentModel {
    /**
     * 其它所有字段省略
     */

    // 上次状态
    private PaymentStatus lastStatus;
    // 当前状态
    private PaymentStatus currentStatus;


    /**
     * 根据事件推进状态
     */
    public void transferStatusByEvent(PaymentEvent event) {
        // 根据当前状态和事件，去获取目标状态
        PaymentStatus targetStatus = PaymentStatus.getTargetStatus(currentStatus, event);
        // 如果目标状态不为空，说明是可以推进的
        if (targetStatus != null) {
            lastStatus = currentStatus;
            currentStatus = targetStatus;
        } else {
            // 目标状态为空，说明是非法推进，进入异常处理，这里只是抛出去，由调用者去具体处理
            throw new StateMachineException(currentStatus, event, "状态转换失败");
        }
    }
}
```

代码注释已经写得很清楚，其中StateMachineException是自定义，不想定义的话，直接使用RuntimeException也是可以的。

在支付业务代码中的使用：**只需要paymentModel.transferStatusByEvent(PaymentEvent.valueOf(message.getEvent()))**

```java
/**
 * 支付领域域服务
 */
public class PaymentDomainServiceImpl implements PaymentDomainService {

    /**
     * 支付结果通知
     */
    public void notify(PaymentNotifyMessage message) {
        PaymentModel paymentModel = loadPaymentModel(message.getPaymentId());
        try {

            // 状态推进
            paymentModel.transferStatusByEvent(PaymentEvent.valueOf(message.getEvent()));
            savePaymentModel(paymentModel);
            // 其它业务处理
            ... ...
        } catch (StateMachineException e) {
            // 异常处理
            ... ...
        } catch (Exception e) {
            // 异常处理
            ... ...
        }
    }
}
```

好处：

1. 定义了明确的状态、事件。

2. 状态机的推进，**只能通过“当前状态、事件、目标状态”来推进**，不能通过if else 或case switch来直接写。比如：STATE_MACHINE.accept(INIT, PaymentEvent.PAY_PROCESS, PAYING);

3. 避免终态变更。比如线上碰到if else写状态机，渠道异步通知比同步返回还快，异步通知回来把订单更新为“PAIED”，然后同步返回的代码把单据重新推进到PAYING。

12. 模板方法Template Method

模板方法简单地理解就是预先定义了一个执行的框架，把公共的部分抽出来，一些具体的业务处理让子类去完成。

支付系统也有很多地方会用到，比如在服务的入口使用模板方法，负责异常处理，返回值封装等，业务代码只需要管业务逻辑就行。

同时在支付系统中，一般比较少用声明式事务，而是使用事务模板，但是spring 原生的事务模板有一定的缺陷，需要我们自己扩展。

下面是一个扩展spring事务模板的例子：

Java

```java
public class FlowTransactionTemplate {

    public static <R> R execute(FlowContext context, Supplier<R> callback) {
        TransactionTemplate template = context.getTransactionTemplate();
        Assert.notNull(template, "transactionTemplate cannot be null");

        PlatformTransactionManager transactionManager = template.getTransactionManager();
        Assert.notNull(transactionManager, "transactionManager cannot be null");

        boolean commit = false;
        try {
            TransactionStatus status = transactionManager.getTransaction(new DefaultTransactionDefinition());
            R result = null;
            try {
                result = callback.get();
            } catch (Exception e) {
                transactionManager.rollback(status);
                throw e;
            }
            transactionManager.commit(status);
            commit = true;
            return result;
        } finally {
            if (commit) {
                invokeAfterCommit(context);
            }
        }
    }

    private static void invokeAfterCommit(FlowContext context) {
        try {
            context.invokeAfterCommit();
        } catch (Exception e) {
            // 打印日志
            ... ...
        }
    }
}

```

使用：

```java
public void process(FlowContext context) {
    context.setTransactionTemplate(dataSourceManager.getTransactionTemplate());

    List<FlowProcess> flows = fetchFlow(context);

    for (FlowProcess flow : flows) {
        // 把Spring模板方法修改自己的模板方法，其它不变
        FlowTransactionTemplate.execute(context, () -> {
            flow.execute(context);

            context.getPayOrder().putJournal(context.getJournal());
            context.getPayOrder().transToNextStatus(context.getJournal().getTargetStatus());
            save(context.getPayOrder());

            return true;
        });
    }
}

```

可以看到在模板里面，定义了主处理，异常处理，事务提交和回滚，事务提交后处理等逻辑，真正的业务代码只需要管业务代码怎么写就行。

13. 适配器Adapter

适配器用于两种不同类型的接口通过一个中间类做适配转换，以达到兼容的效果。

在支付系统中，后端一般都是提供RPC服务，但是客户端往往需要HTTP服务，这种情况下也需要适配器：把HTTP适配到RPC服务，或者说把HTTP服务转换成RPC服务。

在代码中最常见的就是Java IO类库里的字节流与字符流的处理。比如FileInputStream提供的**字节流**，但是业务上想操作**字符流**，于是提供InputStreamReader做为适配器，**适配字节流到字符流**。

示例：

```java
// 使用 InputStreamReader 适配字节流到字符流
InputStreamReader isr = new InputStreamReader(new FileInputStream("test.txt"));
```

14. 装饰器Decorator

装饰器模式简单地说就是包装了原来的类，并扩展一些额外的业务能力。注意这些额外的业务能力和代理模式里面打印日志、权限管控等是不相同的，后者和业务无关。

最典型的例子就是Java IO操作相关的类。比如FileInputStream是一个基础的类，BufferedInputStream则包装了一个FileInputStream，然后**额外提供缓冲读写的能力**。

示例：

```java
// 使用装饰器增加缓冲功能
BufferedInputStream bis = new BufferedInputStream(FileInputStream("input.txt"));
```

15. 迭代器 Iterator

迭代器模式就是提供一种方法访问集合的所有元素。

深入看一下List等集合类的实现类，都实现了Iterable。用得最多的forEach，底层代码就是迭代器模式。

示例代码就不贴了，随处可见。

16. 其它一些设计模式

有些设计模式过于简单，有些在支付系统中比较少用，就不一一做详细介绍。

比较简单的设计模式且大家经常在使用的有：

单例模式（Singleton）：写代码的小伙伴多少都写过单例，还经常有所谓懒加载的讨论。

原型模式（Prototype）：java里面clone就是一个典型的实现，也经常有浅拷贝出现问题后，转而自己实现深拷贝的情况。

在支付系统中比较少用的设计模式有：

享元模式（Flyweight）（留意与缓存的区别。在渠道开关可以考虑使用）、桥接模式（Bridge）、命令模式（Command）、备忘录模式（Memento）（加入审批流的配置变更需要用到备忘录模式）、访问者模式（Visitor）、解释器模式（Interpreter）（如果实现自己的规则引擎会用到）等；

17. 一些容易混淆的点

17.1. 工厂与建造者

建造者和工厂都是创建型设计模式，都用于构建复杂的对象，但两者各有不同的侧重点。**工厂模式对外提供一个接口构建对象，调用者不知道构建细节，而建造者模式需要调用者知道构建细节**。

比如前面生成责任链的工厂，只对外提供一个getHandlers()，而在建造者模式示例中，调用者需要针对每个参数做设置。

17.2. 代理、装饰器、中介者

代理、装饰器、中介者也经常容易混淆。三个都可以在不修改原有代码的基础上增加能力，区别在于：

- 代理不修改被代理对象的能力，只是增加访问控制等能力。
- 装饰器包装原对象，增加更多能力。
- 中介者用于封装多个对象的访问。

代理和装饰器都是封装一个被访问对象，而中介者封装多个对象。

代理不修改被访问对象的业务能力，装饰器增加了业务处理能力。

17.3. 装饰器与适配器

在前面的例子中，BufferedInputStream封装InputStream是**装饰器**，InputStreamReader封装FileInputStream却是**适配器**。因为前者**都是字节流**，BufferedInputStream只是**扩展了缓冲读写**的能力，而后者是提供**字节流到字符流的转换**。

装饰器示例：两个类处理的都是字节流。

```java
// 使用装饰器增加缓冲功能
BufferedInputStream bis = new BufferedInputStream(FileInputStream("input.txt"));
```

适配器示例：字节流到字符串转换。

```java
// 使用 InputStreamReader 适配字节流到字符流
InputStreamReader isr = new InputStreamReader(new FileInputStream("test.txt"));
```

17.4. 观察者与发布–订阅

观察者模式通常用于单机环境，适用简单的一对多场景。

发布–订阅需要用到消息中间件，适用于复杂的解耦通信场景。

18. 混合使用多种设计模式

大部分代码是多种设计模式的组合使用。

比如上面的责任链模式，通过工厂来构建责任链等。

又比如消息中间件，同时兼顾发布–订阅（维护订阅列表，消息发布时通知所有订阅者）和中介者模式（由消息中间件负责消息路由和分发，解耦消息的发送方和接收方直接调用）。

19. 过度设计

所有的设计建议都遵守适用原则，能简单地做就简单地做，也就是所谓的不要过度设计。

举几个在设计模式范围内过度设计的小例子。

1. 使用两个简单的if else就可以解决，却使用策略模式。
2. 通过简单的回调函数就可以解决，却使用观察者模式。
3. 业务流程较为简单，只需要在主函数做很少的显式调用，却使用责任链模式。

20. 结束语

设计模式是软件研发中绕不开的技术课题，最好的学习方法就是结合自己的项目代码，边看边总结边练习，多次实践之后，理解每种模式的特点和应用场景，碰到某个问题，什么模式最适合，基本上就是“信手拈来”。

文章只介绍了个人一点理解和经验总结，只能算是入门。建议有兴趣的读者多翻翻GoF的书。

这是《图解支付系统设计与实现》专栏番外系列文章中的第（1）篇。

深耕境内/跨境支付架构设计十余年，欢迎关注并星标公众号“隐墨星辰”，和我一起深入解码支付系统的方方面面。

专栏系列文章PDF合集不定时更新，欢迎关注我的公众号“隐墨星辰”，留言“PDF”获取。

有个支付系统设计与实现讨论群，添加个人微信（yinmon_sc）备注666进入。

隐墨星辰 个人微信

10年顶尖境内/跨境支付公司架构经验

著有《图解支付系统设计与实现》

备注666进支付讨论群