46. 资深支付架构师视角：实战从问题定义到代码落地的完整套路

大家好，我是隐墨星辰，深耕境内/跨境支付架构设计十余年。

今天从一个实际案例入手，介绍站在架构师的角度，如何识别并定义问题，提炼需求，技术方案选型，再到详细设计，利用AI的能力协助写出核心的代码，最后是验证与调优。

解决问题存在一定的模式，也可以称之为框架，总结出自己的思考和解题框架，以后再碰到同类型的问题就可以如庖丁解牛一样容易。

很多年前，我写代码仍然是if else，最多加个for，也就是所谓的面条式编程。

这三板斧应付简单的业务也没有太大问题，大不了50行代码能搞定的事，我就写个500行。但是一旦需要写一些稍微偏底层的框架，就无从下手。

不过现在的研发工程师比我们以前幸运很多，借助于AI辅助编码，只要把设计写出来，由AI协助实现一个稍微复杂一点的框架也不是什么大问题。

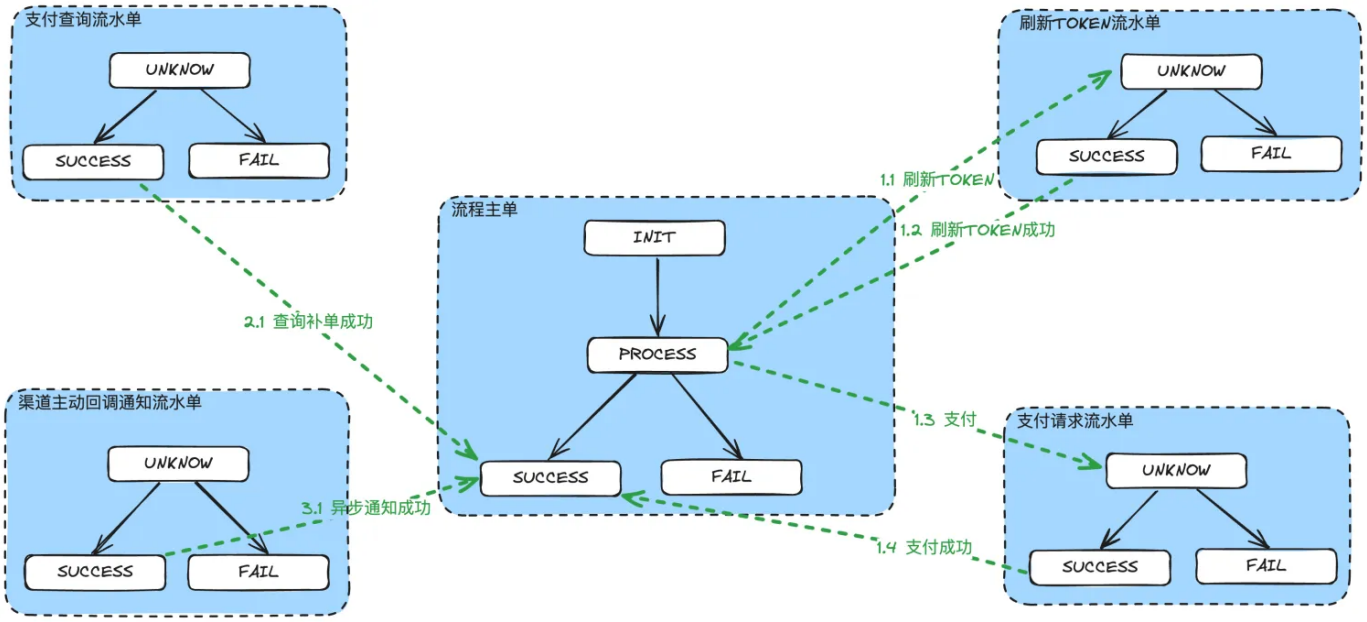
1. 识别问题

在支付系统中，有一个基本的需求就是流程编排，对应的模块就是流程引擎。

比如收银支付需要把整个支付流程串起来，包括用户校验，商户校验，合约校验，风控校验，余额校验等；商户入驻需要针对不同类型的商户串不同的流程；外发渠道请求支付时需要针对不同的渠道交互编排不同的流程。

以渠道流程编排为例，我刚入行第三方支付时，还没有断直连，快捷支付对接了中工农建交招等各种银行，当然还有银联。不同的渠道交互模式是不一样的，有些是先提供独立的签约接口，有些是签约+支付合并在一个接口，有些需要由渠道发送并验证短信验证码，有些需要由我方发送并验证短信验证码，有些需要先刷新token后，再去做支付。

下面是一个典型的渠道交互。



当时没有经验，于是每个渠道写一套独立的接入代码，各种逻辑都能通过if else编排出来。

但实际上，这些交互都可以分解成一个个组件操作，然后通过流程引擎做一个编排，就可以减少很多重复的代码，也能提高系统的健壮性，还能减少经验不足的工程师出现的问题可能性。

小结一下，识别问题：**为每个渠道写独立的代码，冗余代码过多，可维护性差，研发效率低，需要有一个流程编排的能力。**

2. 明确需求

针对不同的渠道接入模式，配置出一些通用的配置，然后通过**流程引擎**驱动**流程配置**，完成和渠道的交互，减少冗余代码，提高可维护性。

3. 方案选型

有个经典的“奥卡姆剃刀原理”，核心思想是：“如无必要，勿增实体”。所以我们先看看市场上有哪些解决方案。

流程引擎在企业软件中无处不在，尤其是内部审批业务里面。市场上的确有很多现成的流程引擎，老牌的比如jBPM，Activiti等都很知名，中国后起之秀且开源的项目liteflow也是其中佼佼者。

其中jBPM，Activiti的配置过于繁琐和复杂，一个简单的支付流程，可能有上百行的配置，放弃。

liteflow的配置非常简单，可以做为备选方案。liteflow的核心思想：所有的操作都是组件，把组件串起来，完成业务能力。

而liteflow的问题之一也是因为其配置过于简单，把业务推进以及判断逻辑全部放在代码中去，导致在配置时无法看到推进的条件。比如下面是一个支付流程：

```xml
<flow>
  <chain name="commonPayChain">
    THEN(transitionToProcess, requestPay,
      SWITH(subOrderStateSwitch).to(transitionToSuccess, transitionToFail))
  </chain>
</flow>
```

我们只知道先推进到支付中，再发起请求，然后判断子订单状态，最后推进到成功或失败。

如果我们一定要选择业界现有的技术解决方案，个人推荐使用liteflow。

还有一个方案，就是自己实现一个订制化的流程引擎。制定流程引擎有几个诉求：

1. 配置要简单清晰。
2. 各种触发条件、推进条件、操作等需要在配置文件中体现。

配置文件也有多种格式，比如XML，YAML，JSON等。除了这几种，还有另外一种选择：**自定义一种语法，使用链式处理。**

市面上也很多这样使用的。

在测试Mock框架Mockito中，我们经常这样使用：

```Java
when(mockedList.get(0)).thenReturn("first element");
```

在java的流式代码中，我们经常这样使用：

```Java
names.stream()
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList());
```

在jQuery中，我们经常这样使用（很多年前我也写JQuery）：

```JavaScript
$('#element').css('color', 'red').show()
  .click(function() { alert('Clicked!'); });
```

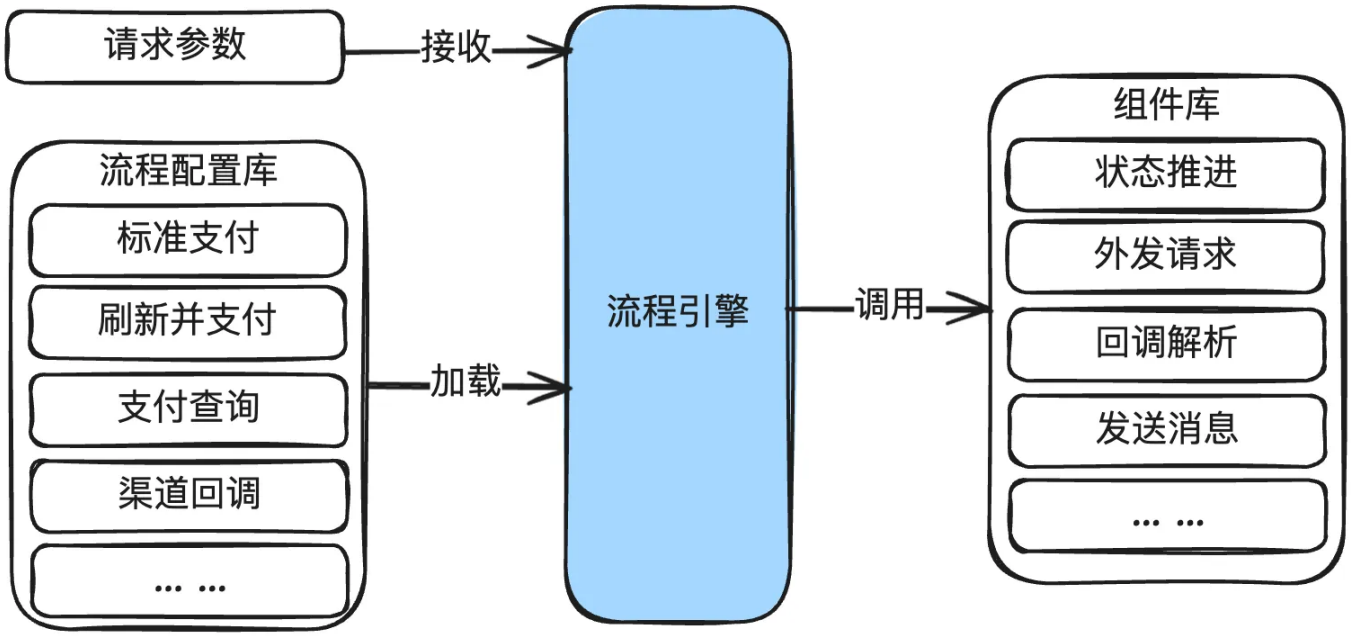
所有以上这些都有一个共同的特点：**链式调用**。

链式调用有个好处：非常接近自然语言，容易理解。最终选择此方案。

4. 详细设计

使用**链式调用**的形式编写配置文件，易于理解和配置。关键字和示例在后面有。

4.1. 整体架构图



说明：

1. 流程引擎在启动时加载所有配置，生成不同的处理链。
2. 根据请求的参数，找到对应的链。
3. 根据处理链的配置，从第一个节点开始，驱动调用不同的组件，直接全部节点运行完毕。

4.2. 状态、事件、操作

分解支付业务的处理到最小粒度，只有三个：订单状态，当前的事件，触发的操作。

也就是：在指定主订单状态的情况下，当前过来了什么事件，进而触发什么样的操作。

- **状态（State）**：表示流程中的不同阶段，例如：`INIT`、`PROCESS`、`SUCCESS`、`FAIL`。
- **事件（Event）**：触发状态转移的事件，例如：`CREATE`、`QUERY`、`CALLBACK`。
- **操作（Operation）**：需要执行的操作，例如：`PAY`、`PAY_QUERY`、`REFUND`。

4.3. 定义关键字

和前面Mockitor或JAVA流式处理一样，**链式调用**需要有一些专用语法。对于支付流程来说，有以下几个关键字：

1. **whenOrderState**：初始条件，主订单状态要满足给定条件。比如：

whenOrderState(CommonOrderState.INIT)。

2. **onEvent**：触发事件。比如：onEvent(CommonEvent.CREATE)。
3. **transitionOrderStateTo**：推进主订单状态。比如：transitionOrderStateTo(CommonOrderState.PROCESS)。
4. **request**：请求操作。比如：request(CommonOperation.PAY)。
5. **when**：判断request返回的数据，比如：when("subOrder.currentState == SubOrderState.S")。
6. **notify**：发出订单消息给其它域。比如：notify()。
7. **then**：进入子流程。比如：when("subOrder.currentState == SubOrderState.S").then(xxx)。
8. **subFlow**：子流程标识。比如subFlow().request(xxx)。

下面是一个通用支付配置的示例：

```Java
whenOrderState(CommonOrderState.INIT)
.onEvent(CommonEvent.CREATE)
.transitionOrderStateTo(CommonOrderState.PROCESS)
.request(CommonOperation.PAY)
    .when("subOrder.currentState == SubOrderState.S")
        .transitionOrderStateTo(CommonOrderState.SUCCESS)
    .when("subOrder.currentState == SubOrderState.F")
        .transitionOrderStateTo(CommonOrderState.FAIL)
    .when("subOrder.currentState == SubOrderState.U && subOrder.webForm != null")
        .notifyNode();
```

简单说明一下：

1. 当初始状态为INIT（初始），触发了CREATE（创建），主订单状态流转到PROCESS（处理中），并发起请求PAY（支付）。
2. 支付结果回来后，当子订单结果为S（成功）就推进主订单成功，当子订单结果为F(失败）就推进主订单失败，当子订单结果为U（未知）且webForm（表单）不为空就发出消息通知其它域。

4.4. 支持Groovy脚本

需要支持配置中的groovy脚本。比如.when("subOrder.currentState == SubOrderState.F")中"subOrder.currentState == SubOrderState.F"就是groovy脚本，当脚本结果为真，就执行后面的操作。

注意在groovy脚本引擎上下文中导入必要的类，包括但不限于 SubOrderState等类，否则脚本运行会报错。

4.5. 核心组件

主要有以下几大类组件：

1. **枚举**：状态、事件等枚举。
2. **模型**：主订单、子订单等模型。
3. **配置**：支付、退款、支付查询等配置。
4. **上下文**：链式处理的上下文。
5. **处理器**：推进状态，外发渠道，解析回调等处理器。

详细如下：

`com.yinmo.flowengine`

|– `enums` ：枚举，包括状态，事件等

| |– `FlowOperation` ：操作枚举基类

| |– `CommonOperation` ：通用操作类型：支付，支付查询，支付回调等

| |– `FlowState` ：状态基类

| |– `CommonOrderState` ：通用订单状态

| |– `SubOrderState` ：子订单状态

| |– `FlowEvent` ：事件基类

| |– `CommonEvent` ：通用事件类型

|– `handler` ：处理器

| |– `HandlerNodeType` ：节点处理器类型

| |– `HandlerNode` ：节点处理器

| |- `SubOrderHandler` ：子订单处理器

| |- `SubOrderSendHandler` ：子订单发送处理器

| |- `SubOrderCallbackHandler` ：子订单回调处理器

|- `config`： 配置

| |- `FlowConfig` ：流程配置接口

| |- `AbstractConfig` ：抽象流程配置

| |- `CommonPayConfig` ：通用支付配置类

| |- `RefreshTokenPayConfig` ：刷新TOKEN支付配置类

| |- `CommonPayQueryConfig` ：通用支付查询配置类

| |- `CommonPayCallbackConfig` ：通用支付回调配置类

|- `context`： 上下文

| |- `FlowContext` ：流程上下文

| |- `GatewayCallbackContext` ：网关回调上下文

|- `com.yinmo.flowengine.model`： 模型

| |- `Order` ：主订单

| |- `SubOrder` ：子订单

|- `com.yinmo.flowengine.service`： 服务

| |- `FlowEngineService` ：流程引擎服务接口

| |- `impl.FlowEngineServiceImpl` ：流程引擎服务实现

|- `com.yinmo.flowengine.test`： 测试类

   |- `CommonPayTest` ：通用支付测试

   |- `RefreshTokenPayTest` ：刷新TOKEN支付测试

   |- `CommonPayQueryTest` ：通用支付查询测试

   |- `CommonPayCallbackTest` ：通用支付回调测试

需要特别说明的几点：

1. 状态、事件等枚举都先定义接口做为基类，主要是考虑扩展性。

2. 配置根据渠道模式来配置，比如有50条渠道，抽象出10个模式，只需要配置10个配置文件就行。

4.6. 部分核心类定义说明

这些描述主要用于生成ChatGPT的提示词用的，ChatGPT能理解并可以在生成代码时生成对应的注释。

1. **模型：**
   - `Order` 模型：
     - 属性包括：String orderId；FlowState previousState；FlowState currentState；Money transactionAmount；List subOrders；String webForm等。
     - 增加方法：transitionToState(FlowState)。
   - `SubOrder` 模型：
     - 属性包含：String subOrderId, String parentOrderId，SubOrderState previousState, SubOrderState currentState, Money transactionAmount, String channelResponseCode, String channelResponseMessage, String standardResponseCode, String standardResponseMessage, String webForm, String sendToChannelContext, String receiveFromChannelContext等。
     - 增加方法：transitionToState(SubOrderState)。

2. **枚举：**
   - 定义interface：FlowState，FlowEvent，FlowOperation。所有枚举全部继承于interface，方便扩展。
   - interface FlowOperation增加方法： `String name()` 、 `String getMethod()` 和 `boolean isCallback()` 。isCallback()用于表示是否是一个外部渠道回调操作，回调是直接解析，不需要发给渠道。getMethod()获取方法名，会发给网关，网关根据这个参数组装发给外部渠道的参数。
   - 通用主订单状态：enum CommonOrderState implements FlowState。
     - 有：INIT, PROCESS, SUCCESS, FAIL。
   - 通用子订单状态：enum SubOrderState implements FlowState。
     - 有：I(Init), U(Unkown), S(Success), F(Fail)。

- 通用操作类型：enum CommonOperation implements FlowOperation。
  - 有：PAY("pay", false), PAY_QUERY("payQuery", false), REFUND("refund", false), REFRESH_TOKEN("refreshToken", false), PARSE("parse", true)等。
  - 示例：PAY("pay", false)为例：说明是一个支付操作，“pay"会发给网关，网关根据这个参数组装发给外部渠道的参数，“false"表示是否是一个回调。
- 通用事件：enum CommonEvent implements FlowEvent。
  - 有：CREATE(订单创建事件，驱动系统向渠道发起支付退款等操作），QUERY（订单查询事件，驱动系统向渠道发起查询操作），CALLBACK（外部渠道主动回调通知事件，驱动系统直接解析网关接收到的报文）。

3. 上下文：

- **FlowContext** 。
  - 属性：`channelName`、`order`、`event`、`configName`、`gatewayCallbackContext` 和 `subOrder` 等。
- **GatewayCallbackContext。**
  - 属性：`subOrderId`、`state`、`callbackMessage`、`channelResponseCode`、`channelResponseMessage`、`standardResponseCode`、`standardResponseMessage` 等。
- 上下文用于在处理器和服务之间传递需要处理的数据。

4. 配置相关：

- interface FlowConfig。
  - 方法：String name()，HandlerNode getHandlerNode(FlowState orderState, FlowEvent event)。
- abstract class `AbstractConfig` implements FlowConfig。
  - 核心能力：使用构建器模式来定义流程。
  - 属性：`Map<FlowState, Map<FlowEvent, HandlerNode>>` 存储配置。
  - 方法：void init()；abstract initConfig()让子类实现。在@PostConstruct init()调用initConfig进行初始化，子类在initConfig()配置脚本。
  - 提供方法定义状态、事件、转换、请求和通知。
- 所有具体的流程配置类继续自AbstractConfig。所有流程配置类提供String name()方法，返回当前的类简要名称用于标识流程名称。示例：getClass().getSimpleName()。

- 定义子类： `CommonPayConfig` ， `RefreshTokenPayConfig` ， `CommonPayQueryConfig` ， `CommonPayCallbackConfig` ，全部继承自 `AbstractConfig` 。

5. 处理节点：

- 节点处理器类型：enum HandlerNodeType。
  - 有：TRANSITION（推进主订单状态变更），REQUEST（向外部渠道发送请求，或解析外部渠道回调报文），NOTIFY（发送消息通知其他域）
- 处理器节点：class `HandlerNode` ，表示流程中的一个节点。
  - 属性：节点类型、下一个状态、操作、回调处理器和子节点列表等字段。
  - 确保 `HandlerNode` 能通过其子节点来完成链式操作。

6. 处理器和服务：

- class `SubOrderHandler` 。
  - 方法：void handle(FlowContext, CommonOperation)。
  - 根据操作委托给 `SubOrderSendHandler` 或 `SubOrderCallbackHandler` 。
- `class SubOrderSendHandler` 。
  - 方法：void handle(FlowContext, CommonOperation)。
  - 创建子订单，保存到数据库，组装网关报文，发送给网关，由网关发送给外部渠道，解析渠道返回的报文，更新子订单，保存子订单最新数据到数据库。把子订单放到FlowContext中。
- class SubOrderCallbackHandler。
  - 方法：void handle(FlowContext, CommonOperation)。
  - 创建子订单，解析渠道主动回调通知返回的报文，更新子订单，保存子订单最新数据到数据库。把子订单放到FlowContext中。
- interface `FlowEngineService` 。
  - void execute(FlowContext);
- class FlowEngineServiceImpl implements `FlowEngineServiceImpl` 。
  - 功能：执行从配置生成的处理节点列表来处理流程。迭代处理节点，并根据需要处理链式操作，通过递归处理子节点来实现。
  - 属性：@Autowired List configs， Map<String, Config> configMap;
  - 方法：@PostConstruct init()方法用于初始化。在init()方法中，把自动装配的List

configs转存到configMap，在execute方法中直接使用configMap.get(context.getConfigName())获取配置。

- 其它：初始化完成后，使用fastjson格式化打印configMap的内容。

7. **日志：**
   - 在需要日志的类中使用 `@Slf4j` 。
   - 以下情况需要打印日志：关键处理，状态变更，只写了空方法还没有具体实现的方法，异常。

8. **测试：**
   - 编写测试类： `CommonPayTest` ， `CommonPayQueyTest` ， `CommonPayCallbackConfig` ，RefreshTokenPayTest。
   - 所有测试类使用Mockitor进行mock子订单的返回值。
   - 数据校验使用断言。
   - 所有测试类需要有三个方法：testSubOrderSuccess(), testSubOrderSuccessFail(), testSubOrderUnkown()。分别是子订单返回成功，返回失败，返回未知。

5. 代码实现

代码全部由ChatGPT生成。为减少篇幅，这里只展示了核心的两个类。

也可以根据提示词自己生成，生成代码的完整提示词，在前面的文章有详细介绍：https://mp.weixin.qq.com/s/XtMUlHBuJs7trnsbdgHjMg。

抽象配置类AbstractConfig，**提供whenOrderState, onEvent, when，then，subFlow，request, transitionOrderStateTo等自定义关键字的实现逻辑**。

Java

```java
public abstract class AbstractConfig implements FlowConfig {

    protected Map<FlowState, Map<FlowEvent, HandlerNode>> configMap = new HashMap<>();

    @PostConstruct
    public void init() {
        initConfig();
    }

    /**
     * 子类实现该方法来初始化配置
     */
    protected abstract void initConfig();

    @Override
    public HandlerNode getHandlerNode(FlowState orderState, FlowEvent event) {
        Map<FlowEvent, HandlerNode> eventMap = configMap.get(orderState);
        if (eventMap != null) {
            return eventMap.get(event);
        }
        return null;
    }

    protected NodeBuilder whenOrderState(FlowState orderState) {
        Map<FlowEvent, HandlerNode> eventMap = configMap.computeIfAbsent(orderState, k -> new HashMap<>());
        return new NodeBuilder(eventMap, null);
    }

    protected NodeBuilder subFlow() {
        HandlerNode subFlowNode = new HandlerNode(null);
        subFlowNode.setSubFlow(true); // 标记为子流程节点
        return new NodeBuilder(null, subFlowNode);
    }

    protected class NodeBuilder {
        private Map<FlowEvent, HandlerNode> eventMap;
        private HandlerNode rootNode;
        private HandlerNode currentNode;

        public NodeBuilder(Map<FlowEvent, HandlerNode> eventMap, HandlerNode rootNode) {
            this.eventMap = eventMap;
```

```
            this.rootNode = rootNode;
            this.currentNode = rootNode;
        }

        public NodeBuilder onEvent(FlowEvent event) {
            if (eventMap != null) {
                HandlerNode eventNode = new HandlerNode(null);
                eventMap.put(event, eventNode);
                this.rootNode = eventNode;
                this.currentNode = eventNode;
            } else {
                throw new IllegalStateException("子流程中不能调用 onEvent()");
            }
            return this;
        }

        public NodeBuilder transitionOrderStateTo(FlowState nextState) {
            currentNode.setNodeType(HandlerNodeType.TRANSITION);
            currentNode.setNextState(nextState);
            return this;
        }

        public NodeBuilder request(FlowOperation operation) {
            HandlerNode requestNode = new HandlerNode(HandlerNodeType.REQUEST);
            requestNode.setOperation(operation);
            currentNode.addChild(requestNode);
            this.rootNode = requestNode;
            this.currentNode = requestNode;
            return this;
        }

        public NodeBuilder when(String condition) {
            HandlerNode whenNode = new HandlerNode(null);
            whenNode.setConditionScript(condition);
            rootNode.addChild(whenNode);
            this.currentNode = whenNode;
            return this;
        }

        public NodeBuilder then(NodeBuilder nodeBuilder) {
            currentNode.addChild(nodeBuilder.rootNode);
            return this;
        }

        public NodeBuilder notifyNode() {
```

```
                HandlerNode notifyNode = new HandlerNode(HandlerNodeType.NOTIF
Y);
                currentNode.addChild(notifyNode);
                return this;
            }

            public HandlerNode getCurrentNode() {
                return currentNode;
            }
        }
    }
```

流程引擎入口实现类：FlowEngineServiceImpl，提供链式处理，并调用各处理器完成业务处理。

```java
public class FlowEngineServiceImpl implements FlowEngineService {

    private final Map<String, FlowConfig> configMap = new HashMap<>();
    @Autowired
    @Setter
    private List<FlowConfig> configs;
    @Autowired
    @Setter
    private SubOrderHandler subOrderHandler;

    @PostConstruct
    public void init() {
        for (FlowConfig config : configs) {
            configMap.put(config.name(), config);
        }
        log.info("ConfigMap initialized with configs: {}", JSON.toJSONStr
ing(configMap));
    }

    @Override
    public void execute(FlowContext context) {
        FlowConfig config = configMap.get(context.getConfigName());
        if (config == null) {
            log.error("No configuration found for name: {}", context.getC
onfigName());
            return;
        }

        FlowState currentState = context.getOrder().getCurrentState();
        FlowEvent event = context.getEvent();
        log.info("当前主订单状态：{}， 触发事件：{}", currentState.name(), ev
ent.name());

        HandlerNode handlerNode = config.getHandlerNode(currentState, eve
nt);
        if (handlerNode == null) {
            log.error("No handler node found for state: {} and event: {}"
, currentState, event);
            return;
        }

        executeNode(context, handlerNode);
    }

    private void executeNode(FlowContext context, HandlerNode node) {
```

```
        // 处理节点，根据节点类型执行相应的操作
        if (node.getNodeType() != null) {
            switch (node.getNodeType()) {
                case TRANSITION:
                    // 推进主订单状态变更
                    context.getOrder().transitionToState(node.getNextState());
                    log.info("主订单状态变更为：{} -> {}", context.getOrder().getPreviousState(), context.getOrder().getCurrentState());
                    break;
                case REQUEST:
                    // 向外部渠道发送请求，或解析外部渠道回调报文
                    subOrderHandler.handle(context, (CommonOperation) node.getOperation());
                    log.info("执行操作：{}", node.getOperation());
                    break;
                case NOTIFY:
                    // 发送消息通知其他域
                    notifyOtherDomains(context);
                    log.info("通知其它域：{}", context.getOrder());
                    break;
                default:
                    log.warn("未知的节点类型：{}", node.getNodeType());
            }
        }

        // 处理子节点
        if (node.getChildren() != null) {
            for (HandlerNode child : node.getChildren()) {
                // 如果有条件，执行Groovy脚本判断
                boolean conditionMet = true;
                if (child.getConditionScript() != null) {
                    conditionMet = evaluateCondition(context, child.getConditionScript());
                }

                if (conditionMet) {
                    executeNode(context, child);
                    // 如果条件满足，执行后不再检查其他兄弟节点
                    break;
                }
            }
        }
    }


    private boolean evaluateCondition(FlowContext context, String conditionScript) {
```

```
84
85          // 使用Groovy脚本引擎执行条件
86          try {
87              GroovyScriptEngineImpl scriptEngine = new GroovyScriptEngineI
    mpl();
88
89              // 设置脚本上下文
90              scriptEngine.put("subOrder", context.getSubOrder());
91              scriptEngine.put("order", context.getOrder());
92
93              // 将必要的导入语句和条件脚本拼接在一起
94              String fullScript = "import com.yinmo.flowengine.enums.*;\n"
    + conditionScript;
95
96              Object result = scriptEngine.eval(fullScript);
97              log.info("执行判断：{}，结果：{}", conditionScript, result);
98              if (result instanceof Boolean) {
99                  return (Boolean) result;
100             } else {
101                 log.warn("条件脚本未返回布尔值: {}", conditionScript);
102                 return false;
103             }
104         } catch (Exception e) {
105             log.error("执行条件脚本时出错: {}", conditionScript, e);
106             return false;
107         }
108     }
109
110
111     private void notifyOtherDomains(FlowContext context) {
112         // TODO: 实现通知其他域的逻辑
113         log.info("Notify other domains with context: {}", context);
114     }
115 }
```

6. 验证与调优

多写几份配置文件，比如支付，退款，支付查询，刷新后支付，回调通知等，多跑几个测试用例，正常，异常等，测试类要有基本的断言。如果有报错或不符合预期，就扔给ChatGPT去修复。

支付成功测试：

```
FlowEngineServiceImpl - 当前主订单状态: INIT,  触发事件: CREATE
FlowEngineServiceImpl - 主订单状态变更为: INIT -> PROCESS
FlowEngineServiceImpl - 执行操作: REFRESH_TOKEN
FlowEngineServiceImpl - 执行判断: subOrder.currentState == SubOrderState.F, 结果: false
FlowEngineServiceImpl - 执行判断: subOrder.currentState == SubOrderState.S, 结果: true
FlowEngineServiceImpl - 执行操作: PAY
FlowEngineServiceImpl - 执行判断: subOrder.currentState == SubOrderState.S, 结果: true
FlowEngineServiceImpl - 主订单状态变更为: PROCESS -> SUCCESS
```

支付失败测试：

```
FlowEngineServiceImpl - 当前主订单状态: INIT,  触发事件: CREATE
FlowEngineServiceImpl - 主订单状态变更为: INIT -> PROCESS
FlowEngineServiceImpl - 执行操作: REFRESH_TOKEN
FlowEngineServiceImpl - 执行判断: subOrder.currentState == SubOrderState.F, 结果: true
FlowEngineServiceImpl - 主订单状态变更为: PROCESS -> FAIL
```

其中的一小段测试代码：

```java
    @Test
    public void testSubOrderSuccess() {
        // 设置Mock行为
        SubOrder subOrder = new SubOrder();
        subOrder.setCurrentState(SubOrderState.S);

        Mockito.doAnswer(invocation -> {
            FlowContext context = invocation.getArgument(0);
            context.setSubOrder(subOrder);
            return null;
        }).when(subOrderSendHandler).handle(Mockito.any(), Mockito.any());

        // 构建上下文
        Order order = new Order();
        order.setOrderId("ORDER123");
        order.setCurrentState(CommonOrderState.INIT);

        FlowContext context = new FlowContext();
        context.setOrder(order);
        context.setEvent(CommonEvent.CREATE);
        context.setConfigName("CommonPayConfig");

        // 执行流程
        flowEngineService.execute(context);

        // 验证结果
        Assert.assertEquals(CommonOrderState.SUCCESS, order.getCurrentStat
e());
    }
```

7. 结束语

解决问题是我们的核心竞争力。

今天以“为支付流程这个细分领域构建一套专用的极轻量级流程引擎”为例子，介绍了在工作中如何做问题抽象，参考行业经验，设计解决方案，并利用AI辅助实现几乎所有的代码。希望能为大家在提高解决工作问题的能力方面提供一些有益的参考。

这里面有一个重要前提是做好知识储备，否则仍然无法顺利解决碰到的工作问题。拿这个案例来说，我们需要了解流程编排的特性，知道流程编排适用于哪些场景，行业有哪些流程引擎，各自的特点是什么，了解链式处理的特点，也需要了解如何使用AI辅助写复杂逻辑代码等。

具体使用ChatGPT生成全套流程引擎的代码，请参考上一篇文章：https://mp.weixin.qq.com/s/XtMUlHBuJs7trnsbdgHjMg。如果需要完整代码，请私信。

这是《图解支付系统设计与实现》专栏番外系列文章中的第（3）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。

隐墨星辰 公众号

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》
和我一起解码支付系统方方面面

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