52. 2024收官|支付系统领域建模最佳实践

大家好，我是隐墨星辰，专注境内/跨境支付架构设计十余年。

在2024年的大部分周末，我们都是全家人呆在图书馆，看书、写作业、辅导作业、写文章，各忙各的，虽然单调，却很充实。我自己写了差不多30万字，画了300多幅手绘风格的图。有兴趣的可以翻翻公众号。

今天是2024年最后一天，聊聊领域建模，也算是对自己做为一个架构师在过去一年工作学习的一个交代。

领域建模，是技术人的世界观之一，也是一个成长的过程。毕业很长一段时间，领域建模于我而言都是一个很高大上而又不实用的概念，多年后才有一些粗浅的领悟。

所以我将更多地从实用性出发，主要聊聊什么是领域建模，为什么需要领域建模，以及一些实践经验。而不是直接套用那套经典的DDD教材的内容。

1. 什么是领域建模

领域建模是领域驱动设计（DDD）这一软件开发方法论的一部分，关于DDD有很多权威的经典书，我也翻过几本，但在几个团队中落地都不太理想，只有领域建模是相对好落地的。后来想通了，**理论一定要结合实际，一定不能教条主义**，毛爷爷说得非常在理。

我理解的领域建模就是**在深刻理解业务的基础上，为业务抽象出一套领域模型及相关的服务**。通常包括实体、属性（值对象）、行为（方法）、状态机、聚合根、实体关系以及领域服务。

还是有点小抽象，拿经典的支付场景举个例子。

实体就是我们可以区分的对象，比如一笔支付单就是一个实体，订单号是它的唯一标识。

属性就是实体拥有的特性。比如一笔支付单有订单号、支付状态、支付金额等，这些都是属性。

实体关系比较好理解，就是实体和实体之间的关系。比如支付单、退款单之间的关系，很明显支付单和退款单之间有一对多关系，而多笔退款单之间没有关系。

行为就是这个模型对外提供的服务，比如状态推进，自检服务等。比如一笔支付发起了退款，是否可以退款也是行为的一部分。

状态机用于标识模型的业务状态。比如支付就有初始化，支付中，支付成功，支付失败等。

聚合根是指在一个聚合中负责维护内部一致性的实体。比如退款单一定有一个支付单的存在，这个支付单就是一个聚合根。

领域服务通常是指那些不适合放在实体中的服务集合。比如支付完成后，需要发消息通知其它域，这个操作就不适合放在支付单里，就需要有支付领域服务来承载。

2. 为什么需要领域建模

做一些简单的项目，是完全不需要领域建模的。就像在农村建一栋三层小洋楼，并不需要很高深的土木工程的理论知识和严密的设计。但是要建设一栋摩天大楼就一定需要科学的土木工程的理论知识和严密的设计。

所在如果只是为一个小型电商做个支付模块，我建议简单为上。但是如果设计一个专业的持牌第三方支付平台或者需要处理几千上万亿规模的资金量，那领域建模基本上就是不二之选。

领域建模的好处首先是**准确反映业务需求**。开发团队只有在深入理解业务流程和规则的基础上，才能抽象出好的领域模型，并确保系统功能与实际需求一致。

其次是**提升系统可维护性**。一个清晰的领域模型使得系统结构更加明晰，真正做到高内聚低耦合，便于后续的维护和扩展。

还有就是**提高团队沟通效率**。领域模型作为团队内部的共享语言，减少了开发过程中因理解偏差带来的沟通成本。

最后是**快速支持业务变更**。业务持续在变，工作这么多年就没有见过不变的业务，而设计良好的领域模型能够快速适应和调整，确保系统的灵活性和适应性。以前在知乎上看到过一个有趣的问题：“为什么公司需要那么多软件工程师，他们为什么不把代码一次写到位？”这还不是因为业务一直在变化么？

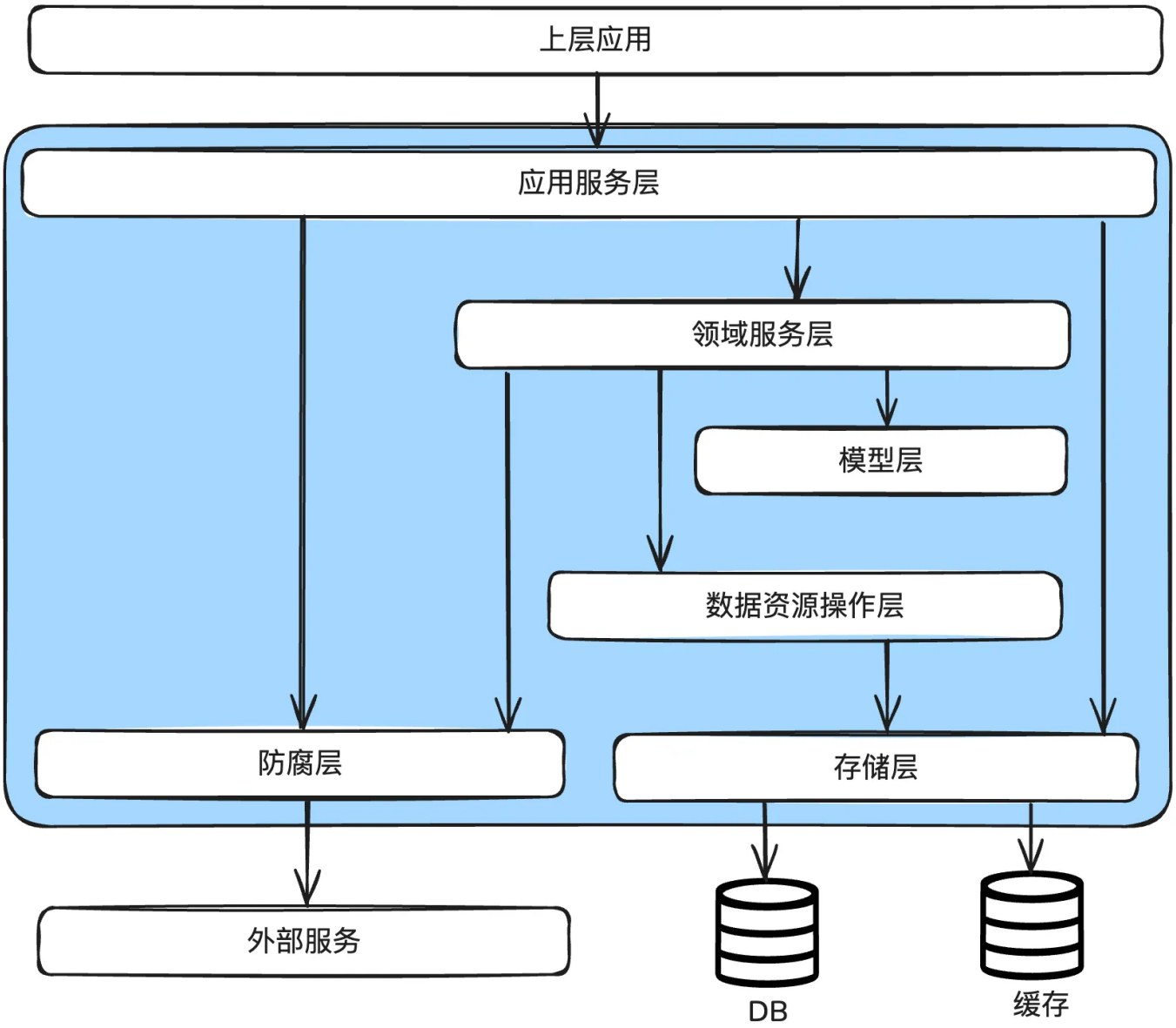
3. 领域建模的一些实践经验

领域建模并不只是把实体定义清楚就足够，还包括分层架构设计、模型自检等。

下面以支付和退款业务场景的领域建模为示例说明。

3.1. 分层架构设计

为了构建一个可维护、可扩展且高效的支付系统，采用分层架构是一种行之有效的设计方法。分层架构将系统划分为不同的层，每一层负责特定的职责，从而实现关注点分离。



上图是一个典型的领域建模里面的分层架构。当然也不是绝对的，这只是其中的一种参考形态。

1. **应用服务层**：对上游提供服务，做基本的检查，比如参数是否合法等，然后转发给领域服务。比如受理上游的支付、退款请求。
2. **领域服务层**：实现复杂的业务流程处理。需要使用模型，调用数据资源操作层保存到数据库，也需要调用外部服务。
3. **模型层**：包括业务实体、属性、业务规则、状态机、模型自检能力等。比如支付单，退款单模型等。
4. **数据资源操作层**：负责与存储层进行数据交互。包括增、删、改、查处理。
5. **存储层**：实现具体的持久化机制，如关系型数据库、NoSQL数据库、缓存等。
6. **防腐层**：处理与外部系统的交互，防止外部模型污染内部领域模型。比如下游服务升级接口，

那我们只需修改防腐层，而不需要修改内部其它层的代码。

示例如下：

应用服务层

对上游提供服务，做基本的检查，比如参数是否合法等，然后转发给领域服务。比如受理上游的支付、退款请求。

```
public class PaymentApplicationServiceImpl implements PaymentApplicationService {
    private PaymentRepository paymentRepository;
    private PaymentDomainService paymentDomainService;

    public PayResponse pay(PayRequest request) throws Exception {
        check(request);
        PayRequestDto payRequestDto = buildPayRequestDto(request);
        PayResponseDto payResponseDto = paymentDomainService.pay(payRequestDto);
        return covertFromDto(payResponseDto);
    }

    // 其他应用服务方法
}
```

领域服务层

实现复杂的业务流程处理。需要使用模型，调用数据资源操作层保存到数据库，也需要调用外部服务。

需要注意下面示例中的GatewayService是防腐层定义的服务，不是渠道网关直接提供的服务。

```java
public class PaymentDomainServiceImpl implements PaymentDomainService {
    private PaymentRepository paymentRepository;
    private RefundRepository refundRepository;
    private GatewayService gatewayService;

    public PayResponseDto pay(PayRequestDto request) throws Exception {
        // 业务逻辑，如验证支付信息、调用支付网关等
        PaymentOrder order = buildOrder(request);
        save(order);
        pay(order);
        // 其他业务操作
    }

    // 其他领域服务方法
}
```

模型层

包括业务实体、属性、业务规则、状态机、模型自检能力等。比如支付单，退款单模型等。

```java
public class PaymentOrder {
    private String paymentId;
    private Money paymentAmount;
    private PaymentStatus previousStatus;
    private PaymentStatus currentStatus;
    private List<RefundOrder> refundOrders;

    public void transferStatus(PaymentEvent event) throws Exception {
        // 通过状态机根据当前状态和事件来推进目标状态
        PaymentStatus targetStatus = PaymentStatus.getTargetStatus(currentStatus, event);
        if (null != targetStatus) {
            previousStatus = currentStatus;
            currentStatus = targetStatus;
        }
    }

    public boolean canRefund(RefundOrder) {
        // 是否能退款判断
    }

    // 其他方法省略
}
```

数据资源操作层

负责与存储层进行数据交互。包括增、删、改、查处理。

```
public interface PaymentRepository {
    void save(PaymentOrder paymentOrder);
    PaymentOrder findById(String paymentId);
    // 其他数据访问方法
}
```

存储层

实现具体的持久化机制，如关系型数据库、NoSQL数据库、缓存等。

```
public interface PaymentDao {
    void save(PaymentOrderDo paymentOrderDo);
    PaymentOrderDo findById(String paymentId);
    // 其他数据访问方法
}
```

防腐层

处理与外部系统的交互，防止外部模型污染内部领域模型。比如下游服务升级接口，那我们只需修改防腐层，而不需要修改内部其它层的代码。

很多时候大家不想写这一层，觉得太麻烦，直接在内部代码引用外部的服务，一旦外部要升级，就有可能导致内部很多地方修改代码。

```
public class GatewayServiceImpl implements GatewayService {
    private ExternalGatewaySerivce externalGatewaySerivce;

    public GatewayServicePaymentResponse pay(GatewayRequest request) throws Exception {
        // 将内部模型转换为外部系统需要的格式
        ExternalGatewayRequestDto dto = convertToDto(request);
        ExternalGatewayResponseDto responseDto = externalGatewaySerivce.pay(dto);
        // 将外部响应转换回内部模型
        return convertFromDto(responseDto);
    }
}
```

3.2. 模型自检

在复杂的业务系统中，模型自检机制是很重要的。它能够确保数据的一致性和业务规则的正确性。

以退款举例，需要支付成功后才能退款，总退款金额不能超过支付金额（可能有多次部分退），已经拒付就不能再次退款等。这些校验可以放到很多地方，比如应用服务层、领域服务层、模型层。我个人建议是放在模型层，因为这属于模型自检的范围。

代码比较简单，就不占字数了。

3.3. 使用状态机推进而不是直接setStatus

状态机在领域建模中用于管理实体的状态及其转换，确保业务流程的正确性。支付系统中的各个实体（支付单、退款单、撤销单）都有各自的状态及相应的转换规则。

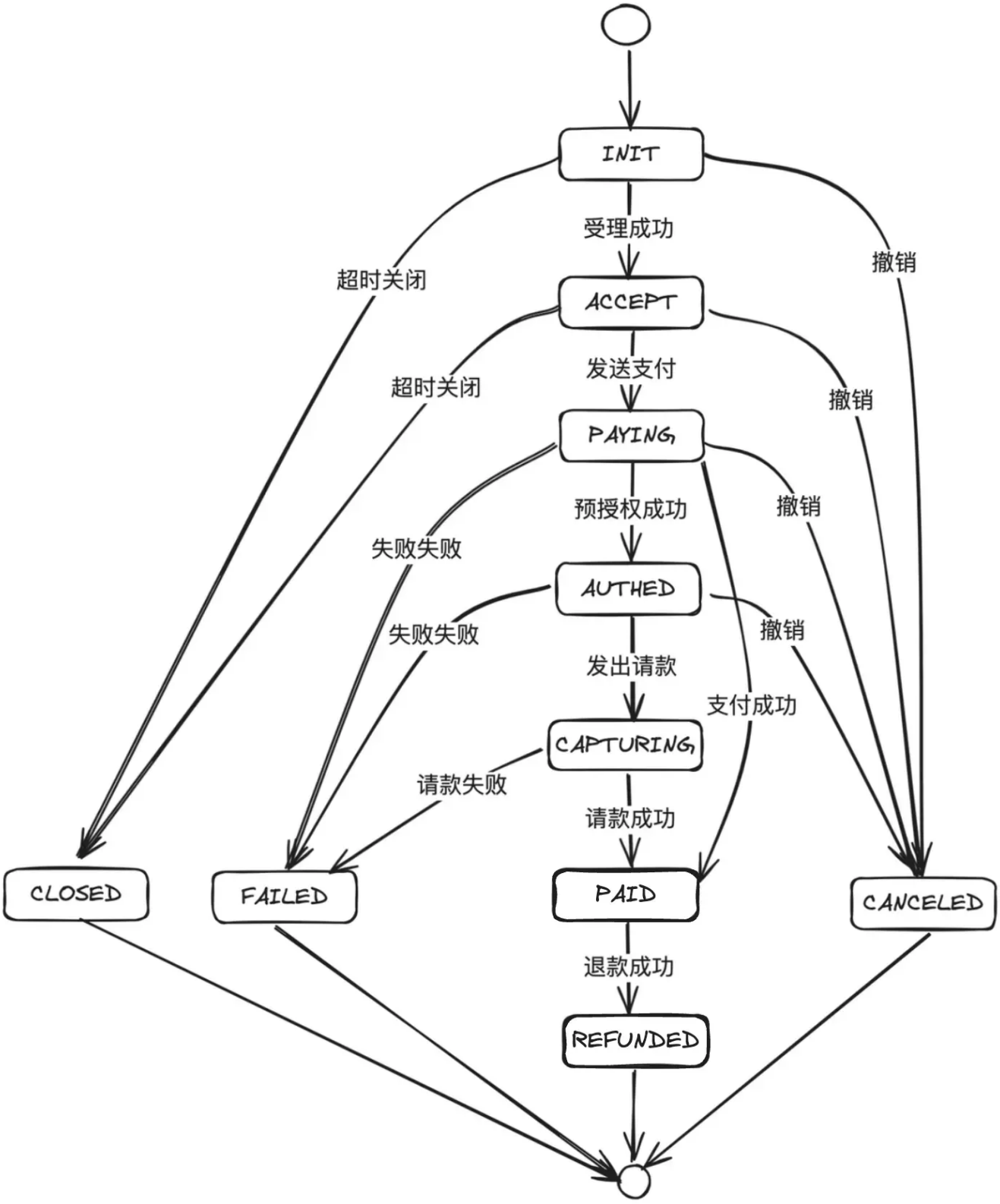
有个常见的新人误区，就是把状态的流转放在领域服务层，在DomainService里直接调用模型的setStatus方法。我个人不建议这么做，有两个原因：

1. 领域服务的判断很有可能是错的，比如当前是fail，不能再推进到success，但是代码有bug，没有判断当前状态，就直接调用了setStatus从fail推进到了success。
2. 状态的流转，也是模型的一部分，应该传入事件，由模型自己根据状态机的配置，当前状态，传入的事件来驱动。比如下面代码：

```java
public class PaymentOrder {
    public void transferStatus(PaymentEvent event) throws Exception {
        // 通过状态机根据当前状态和事件来推进目标状态
        PaymentStatus targetStatus = PaymentStatus.getTargetStatus(currentStatus, event);
        assertNotNull(targetStatus,
        if (null != targetStatus) {
            previousStatus = 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");
        }
    }

    // 其他方法省略
}
```

另外，在设计状态机时，不要把所有业务状态全部放进去，那样很不好维护，比如下面这个就是一个很差的状态机设计。具体怎么优化，可以参考公号发的那篇状态机设计最佳实践的文章。


INIT
受理成功
超时关闭
撤销
ACCEPT
发送支付
超时关闭
撤销
PAYING
预授权成功
撤销
失败失败
AUTHED
失败失败
撤销
发出请款
支付成功
CAPTURING
请款失败
请款成功
CLOSED
FAILED
PAID
CANCELED
退款成功
REFUNDED

3.4. 模型和视图分离

上层的业务需求可能会经常变，尤其是支持多个业务时，各个业务关注点可能不一样，也有一种可能就是上游只需要感知部分属性，直接把整个模型透出去是不合适的，这就要用到模型和视图分离的设计思路。

具体来说，可以分为**命令模型（Command Model）和查询视图（Query View）**，这类似于CQRS（Command Query Responsibility Segregation）模式的思想。

命令模型负责处理业务逻辑和状态变更。在支付系统中，命令模型包括支付单、退款单和撤销单等实体，每个实体有自己的状态机，负责管理其生命周期和状态转换。

查询视图负责提供高效的数据读取和展示。在支付系统中，查询视图可以汇总支付单、退款单和撤销单的信息，提供一个统一的视图，方便上层应用使用。

好处主要有两个，一是**简化复杂性**，命令模型专注于业务逻辑，查询视图专注于数据展示，避免了在单一模型中混杂复杂的读写逻辑。二是**扩展性**，独立的查询模型可以根据不同的展示需求灵活调整，不影响命令模型的业务逻辑。

3.5. 保持模型的简洁性

现实中经常看到过度设计的情况出现，如何平衡可扩展性和过度设计也是一门艺术。我个人建议大可不必为了未来的可能需求而增加不必要的复杂性。我们只实现当前明确需要的功能和结构就已经足够。

只要我们确保模型是简单的，那么就容易理解和维护，未来扩展也是比较方便的。且领域模型本来也应该是可持续迭代和优化的。

4. 最佳实践总结

稍微做个小总结。根据个人经验，遵循以下最佳实践能够有效提升模型的质量和系统的稳定性：

1. **明确核心实体和边界**：准确识别业务领域中的核心实体及其边界，确保模型能够恰到好处覆盖业务需求。
2. **定义清晰的实体关系**：通过一对一、一对多等关系，合理组织实体之间的关系，反映真实的业务流程或关系。
3. **集成状态机设计**：为每个核心实体设计状态机，明确各状态及其转换规则，确保业务流程的正确性和一致性。
4. **分层架构设计**：

- **应用服务层**：对上游提供服务，做基本的检查，比如参数是否合法等，然后转发给领域服务。
- **领域服务层**：实现复杂的业务流程处理。
- **模型层**：定义业务实体、属性、方法、状态机、模型自检能力等。
- **数据资源操作层**：负责与存储层进行数据交互。包括增、删、改、查处理。
- **存储层**：实现具体的持久化机制，存储业务数据。
- **防腐层**：处理与外部系统的交互，防止外部模型污染内部领域模型。

5. **区分模型和视图**：
   - **命令模型**：专注于处理业务逻辑和状态变更。
   - **查询视图**：提供符合业务诉求的数据读取和展示视图。
6. **实现业务规则的自检机制**：在模型层面嵌入业务规则的校验，确保数据的一致性和业务逻辑的正确性。
7. **保持模型的简洁性和可维护性**：避免过度设计，保持模型的简洁性，同时确保其具备良好的可扩展性，以应对未来的需求变化。
8. **持续迭代和优化**：领域模型不是一成不变的，应根据实际业务需求和系统反馈，持续进行迭代和优化。

5. 结束语

领域建模只是我们众多软件设计方法论中的一个，在复杂的支付系统中，准确的领域模型不仅能够提升开发效率，还能确保系统在应对复杂业务需求时的稳定性和灵活性，但也会带来一定的复杂度，对架构师抽象现实世界的能力要求也是比较高的。另一方面，领域建模也不是万能的，在一些简单的系统里，完全没有必要使用。

时间过得很快，世界变化则更快。过去的2024年，飞速发展的AI和不断突进的WEB3带来不少新的机会，同时也带来不少焦虑。

无论初心是改变世界，或是让自己的小家庭更幸福，又或是让自己过得开心就好，愿大家在2025年都能收获自己所愿。

这是《图解支付系统设计与实现》专栏系列文章中的第（52）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”，及时获得最新文章更新推送，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