电商服务开放平台-DDD设计实战二

“DDD” VS DDD:服务开放平台解耦改造思路

--楼兰

一、MVC挺好的，为什么要用DDD?

1、DDD简介

DDD全称是Domain-Driven-Design，领域驱动设计。是在2004年由Eric Evans(NoSQL概念提出者)提出的一种架构思想。DDD在诞生之后的很长一段时间内，并没有引起业界太多的关注，而直到2014年之后，微服务技术大行其道后，DDD才开始在业界火了起来。去哪儿网已经开始全面支持DDD， 爱奇艺、阿里等很多互联网企业都大量分享过关于DDD在企业内落地的经验。下面这本就是DDD的诞生之作。

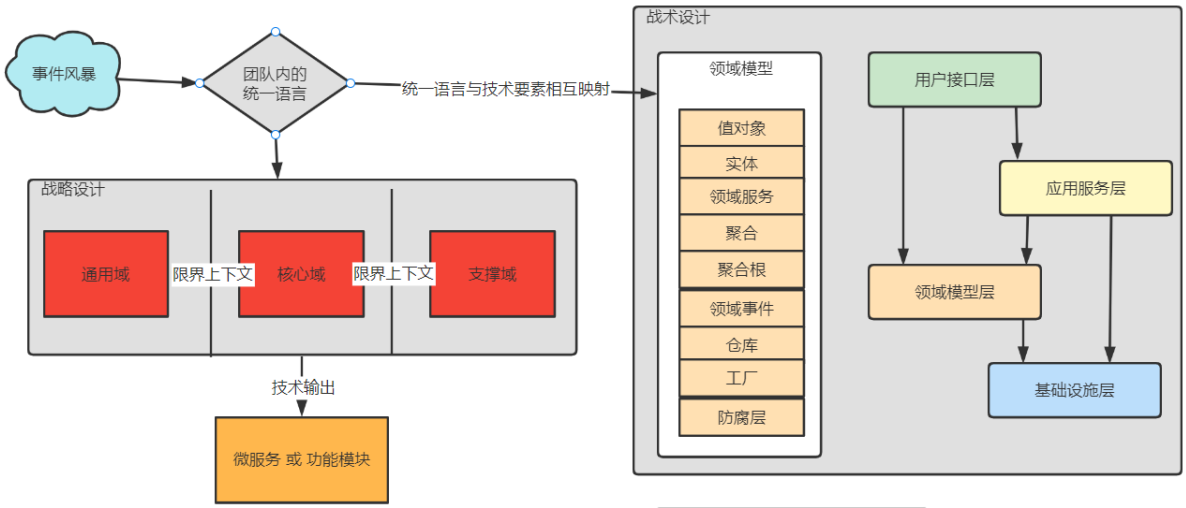
这个是人民邮电出版社的最新翻译版。VIP资料中会提供上一版本的精校PDF版本。

另外，对于DDD，这本原著行文有点硬核，比较难懂。业界有另一部非常经典的DDD著作，行文就比较自然，更方便去阅读理解。也简称为IDDD，其经典程度相当于人工智能中提到的西瓜书和花书。

然后，在我们的VIP拓展课程中，也有一个关于DDD的详细的分享课程，都可以作为DDD比较好的入门资料。

2、DDD有什么用？什么时候要考虑用DDD呢？

从书名就能看到，"软件核心复杂性应对之道"，这意味着DDD要处理的问题就是很复杂的，所以DDD本身，也是一种学习曲线比较陡的设计思想。关于DDD的学习和应用，对任何人都不是一个简单容易的过程。DDD中一大堆概念需要去理解，这就造成了不小的学习门槛。而在落地实践时的理解，更是仁者见仁智者见智。例如，下面大致列出了一些DDD中的基础概念：



这其中每一个概念都是行业内多年优秀开发经验的积累。要把这其中每一个概念，理解透彻并且能落地实践，都是非常费力的，尤其在使用DDD的初期，很多思维方式都需要转变。这会影响到整个开发团队，甚至连需求人员都包含其中。关于这些难点，通过实战课程加上之前推荐的资料，加上后面的实战课程，我都会通过理论+实战的方式带你去一一解决。但是，在学习之处，有一个问题需要你自己先理解，就是之前电商那么复杂的项目都开发得好好的，到了这个简单的服务开放平台，为什么要费劲巴拉的用DDD呢?

你或许应该听说过网上对于DDD的一句很经典的评价。“**项目越复杂，DDD的收益越大。而项目如果很简单，DDD反而会事倍功半**”。那是不是我们这个简单的服务开放平台就不应该用DDD呢?

其实，这里就首先需要你对DDD所要面对的"软件核心复杂性"有所理解。你可以先想想，要做一个软件，最复杂的地方到底是什么？是MQ、缓存、微服务这些技术吗？不是，这些技术你可能现在不会，以后学学也就会了。对于一个成熟的开发团队，这些基础技术更加不是问题。那，是庞大的业务体量，是复杂的业务流程吗？也不是。如果都是同样的CRUD，再复杂的业务流程，你一个人做不来，有一个团队来做，也就没什么了。

而如果你作为一个架构师，真正让你头疼的，往往是各种三高架构、线程安全等等这些稀奇古怪的问题。如何发现问题，如何对现有项目进行改造升级，这些才是软件所要面临的最为核心的复杂性。而这一类问题其实有一个共同点，就是变化，并且是超出你之前意料之外的变化。比如，之前的电商项目，从一开始就知道要去处理秒杀的问题，所以，从一开始，就可以引入Redis，做各种各样的并发设计。这些设计虽然复杂，你带领的是一个完整的开发团队，也就没有什么复杂的了。甚至如果你已经做过一个实现了，换成另外的场景再要实现一次，那就更容易了。而真正难以控制的是，在项目初期并没有考虑到秒杀的高并发问题，甚至都没有用上Redis，等到业务中出现了秒杀问题的时候，再要去临时变更你的项目设计，甚至可能需要重构，这时项目的质量就很难得到保证。并且，这种问题，随着你的项目规模越来越庞大，所面临的风险也会越来越大。而这种问题，就是DDD真正需要面对的软件核心复杂性问题。

因此，如果你要做的项目就是一个需求非常确定的项目，那么，不管项目多复杂，其实都没有必要非要转型成为DDD。像之前电商项目那样的短平快的设计方式会更为快捷。画清楚设计图，就可以按MVC去实现了。而如果你的项目中，有很多不确定性，以往的设计模式会遇到非常多的变数，这时DDD就是一个很好的选项了。

所以，你不妨把我们后面的实战项目当成是一个起点，而把互联网上的这些开放平台当作目标，这样才能更好的理解DDD。

二、MVC三层架构 VS DDD四层架构

学习DDD，首先就需要了解其中一大堆的基础概念(概念图中战术设计那一部分)。这些概念会像MVC的Controller、Service等一样贯穿整个开发过程。

关于这些基础的概念，在之前推荐的资料中都有大量的介绍。下面从一个小案例开始，来真实理解下这些概念是如何影响我们的开发方式的。后续的实战项目中，也会按照这个模式来对DDD进行落地。

业务案例说明

我们先来看一个简单的需求案例：用户购买商品后，向商家进行支付。

当软件团队接受这样的需求时，团队当中的产品设计人员就会尝试对这个业务进行动作拆解，最终确定出如下的步骤：

1、 从数据库中查出用户和商户的账户信息。

2、调用风控系统的微服务，进行风险评估。

3、实现转入转出操作，计算双方的金额变化，保存到数据库。

4、发送交易情况给kafka，进行后续审计和风控。

于是，开发人员使用MVC架构很快的完成了如下的伪代码：

```
public class PaymentController{
    private PayService payService;
    public Result pay(String merchantAccount,BigDecimal amount){
        Long userId = (Long) session.getAttribute("userId");
        return payService.pay(userId, merchantAccount, amount);
    }
}

public class PayServiceImpl extends PayService{
    private AccountDao accountDao;//操作数据库
    private KafkaTemplate<String, String> kafkaTemplate;//操作kafka
    private RiskCheckService riskCheckService;//风控微服务接口

    public Result pay(Long userId,String merchantAccount,BigDecimal amount){
        // 1. 从数据库读取数据
        AccountDO clientDO = accountDAO.selectByUserId(userId);
        AccountDO merchantDO =
        accountDAO.selectByAccountNumber(merchantAccount);
        // 2. 业务参数校验
        if (amount>(clientDO.getAvailable()) {
            throw new NoMoneyException();
        }
        // 3. 调用风控微服务
        RiskCode riskCode = riskCheckService.checkPayment(...);
        // 4. 检查交易合法性
        if("0000"!= riskCode){
            throw new InvalideOperException();
        }
        // 5. 计算新值，并且更新字段
        BigDecimal newSource = clientDO.getAvailable().subtract(amount);
        BigDecimal newTarget = merchantDO.getAvailable().add(amount);
        clientDO.setAvailable(newSource);
        merchantDO.setAvailable(newTarget);
        // 6. 更新到数据库
        accountDAO.update(clientDO);
        accountDAO.update(merchantDO);
        // 7. 发送审计消息
```

```
        String message = sourceUserId + "," + targetAccountNumber + "," +
targetAmount;
        kafkaTemplate.send(TOPIC_AUDIT_LOG, message);
        return Result.SUCCESS;
    }
}
```

我们可以看到，在一段业务代码中包含了参数校验、数据读取、业务计算、调用外部服务、发送消息等多种逻辑。在这个案例中都写到了同一个方法中，我们在实际工作中，绝大部分代码都或多或少的类似于这样的结构。即使经常会被拆分成多个子方法，但是实际效果是一样的。这样的代码样式，就被叫做**事务脚本(Transaction Script)**。这种事务脚本的代码在功能上没有什么问题，但是长久来看，他就给系统带来了非常多老化的风险。

问题1：代码层面的软件老化风险

首先从代码层面来看，这段代码在以后的迭代中会不断的膨胀。

- 数据来源被固定：AccountDO类是一个纯数据结构，映射了数据库中的一个表。如果表结构改了，比如表名改了，字段改了，或者表要做Sharding分库分表，都会导致数据格式不兼容。
- 业务逻辑无法复用：数据格式不兼容的问题会导致整个业务逻辑都无法复用。每个用例中都有大量的if-else分支，以后新加入业务逻辑，可能又要添加新的if-else分支。而这众多的逻辑分支会让代码变得非常难以理解。
- 业务逻辑与数据存储相互依赖：当业务逻辑越来越复杂时，新加入的逻辑可能要调整表结构或者消息的格式。而这些调整都会导致原有的逻辑需要跟着一起动。甚至可能出现一个新功能的增加导致所有原有功能都需要重构的情况。

在这种事务脚本式的架构下，一般做第一个需求都会非常快。但是往后迭代的过程中，代码会不断膨胀，做一个需求需要的时间也会呈指数级上升。绝大部分时间都需要花费在老功能的重构和兼容上，最终拖慢整个项目的创新速度，直到最后不得不推翻重构。

问题2：架构层面的软件老化风险

对于这样的事务脚本，代码的外部依赖非常严重。当任何一个依赖发生变化时，这一段代码可能都要进行大改。

- 数据结构变更：这里面的AccountDO类是一个纯数据结构，映射了数据库中的一个表。从长远来看，表的结构很有可能是会改变的，甚至存储数据的介质也可能会变，例如从Oracle转向MySQL，从关系型数据库转向NewSQL、NoSQL甚至是大数据体系。
- JDBC依赖调整：AccountDao依赖于具体的JDBC实现。如果未来更换ORM体系，迁移成本会非常大。例如从Hibernate转为MyBatis，或者转向SpringData。
- 第三方服务变更：这里面的RiskCheckService是由风控系统提供的一个微服务接口，那未来如果微服务系统做了结构调整，这一段业务流程也需要做大量调整，成本会非常大。
- MQ中间件调整：今天我们用kafka发消息，那未来如果改用RocketMQ呢？如果消息序列化的方式要从String改成Binary呢？

这样的案例中，每个组件的变化都足以导致整个业务逻辑的变更。而当整个项目不断变得复杂，基本上每一次功能的更迭都会需要花费大量的时间用来做各种库升级，解决jar包冲突。最终这个应用就会演变成一个不敢升级，不敢写新功能，并且随时会爆发的炸弹。

问题3 随之而来的实施及测试问题

这样强依赖的代码，还会带来随之而来的一系列测试问题：

设施搭建困难：代码中依赖了好几个外部组件，一个测试用例需要确保所有依赖都能跑起来，这在项目早期是非常困难的。

测试用例覆盖困难：这样一段由多个子步骤耦合而成的事务脚本，要完整覆盖到所有的情况，所需要的测试用例会呈指数级增长。功能组件不够独立，造成单元测试困难，而一个小小的需求变更，测试人员就需要进行完整的回归测试，这样才能覆盖到所有的情况。这种情况通常就是给线上bug埋下的伏笔。

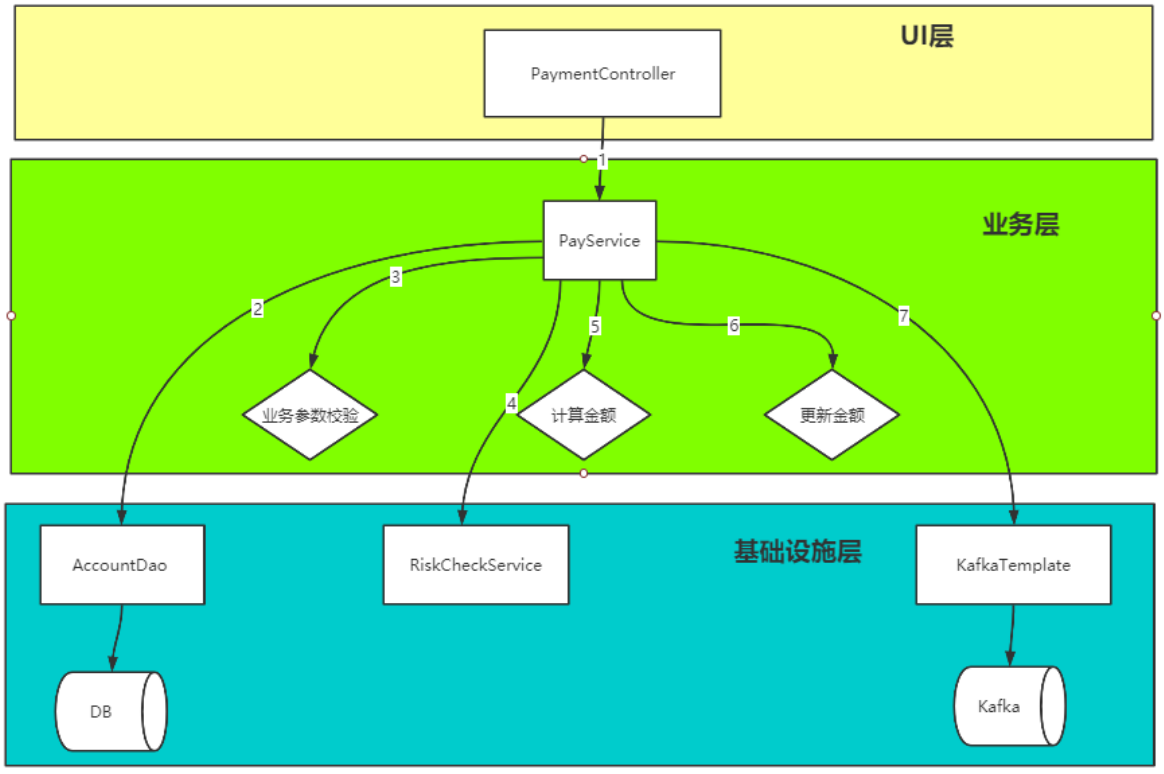
问题总结：

这样一段我们经常写的代码其实违背了好几个软件设计的原则：

- 单一职责原则：要求每一个对象/类应该只有一个变更的原因。但是这个案例中，导致变更的原因非常多。
- 依赖翻转原则：要求在代码中依赖的只是抽象，而不是具体的实现。这个案例中，RiskCheckService虽然只是一个接口，但是与他对应的是整个具体的风控系统，也算是依赖了实现。
- 开放封闭原则：要求整个代码对扩展开放，而对修改封闭。在这个案例中，金额的计算应该属于可被修改的代码，以后随时可能加上一些收取手续费之类的逻辑。所以该逻辑应该被封装成不可修改的计算类。

支付功能重构实战

那要如何按照DDD的思想来对这个功能进行重构呢？我们先画一张流程图，来描述这一段代码：



在我们现在的三层结构中，上层对于下层有直接的依赖关系，我们需要对这张图上的每个节点做抽象和整理，来降低转账业务与外部依赖的耦合度。将引起业务代码变化的因素一点一点的隔离开来。

在我们进行代码改造的过程中，会逐渐的接触到一些DDD中的概念，这会给你带来一些困惑。但是没关系，这些概念在后面的章节当中都会一点点的给你讲明白。在这里，你只需要跟上这种隔离变化的思路。

1、抽象数据存储层

这一步常见的操作是加一个数据接口层，降低系统对数据库的直接依赖。

1-1、改造Account实体类：

```
public class Account{
    private Long id;
    private Long accountNumber;
    private BigDecimal available;

    public void withdraw(BigDecimal money){
        //转入操作
        available = available + money;
    }
    public void deposit(BigDecimal money){
        //转出操作
        if(available < money){
            throws new InsufficientMoneyException();
        }
        available = available - money;
    }
}
```

在这一步改造过程当中：原有的AccountDO只是单纯的和数据库表的映射，只有数据没有行为，被称为**贫血模型**。而新增的Account就是基于领域逻辑的**实体类(Entity)**。他的字段和数据库存储不需要有必然的联系。Entity中包含数据，同时也包含完全内敛的业务行为，称为**充血模式**。

在你习惯的贫血模型下，要理解清楚Account实体涉及到了哪些业务动作，就只能从上层的Service中一点点梳理。而一旦项目变得复杂，Service也会变得错综复杂，梳理业务也就变得更困难。这时候，从Account实体上就很难梳理清楚业务关系。以后业务要进行演进，自然也就更困难。这种问题，被形象的描述为**贫血失忆症**。而反观充血模型，系统中对于Account实体的所有业务行为，可以从实体中一目了然。所以通常意义下，充血模型能够更好的体现业务能力。

1-2、新建对象存储接口类

完成实体对象的抽取后，还需要避免业务逻辑代码和数据库的直接耦合。

```
public interface AccountRepository {
    .......
}
public class AccountRepositoryImpl implements AccountRepository {
    @Autowired
    private AccountDao accountDAO;
    @Autowired
    private AccountBuilder accountBuilder;
    @Override
    public Account find(Long id) {
        AccountDO accountDO = accountDAO.selectById(id);
        return accountBuilder.toAccount(accountDO);
    }
    @Override
    public Account find(Long accountNumber) {
        AccountDO accountDO = accountDAO.selectByAccountNumber(accountNumber);
        return accountBuilder.toAccount(accountDO);
    }
    @Override
    public Account save(Account account) {
```

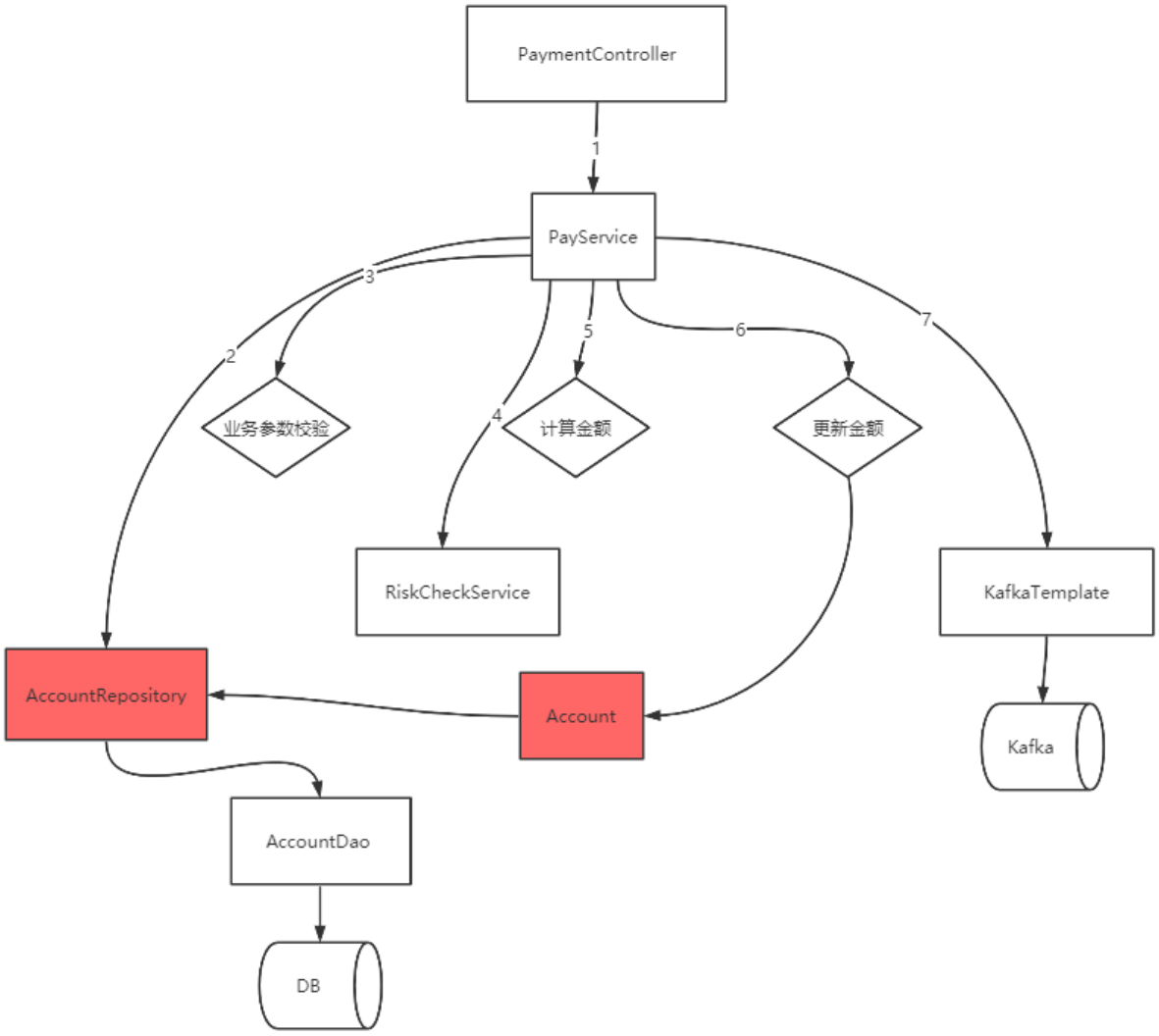
```
        AccountDO accountDO = accountBuilder.fromAccount(account);
        if (accountDO.getId() == null) {
            accountDAO.insert(accountDO);
        } else {
            accountDAO.update(accountDO);
        }
    return accountBuilder.toAccount(accountDO);
}
```

在这一步的改造过程当中：

- DAO对应的是一个特定的数据库类型的操作，相当于SQL的封装。所有操作的对象都应该是DO类，所有的接口都可以根据数据库实现的不同而改变。
- Repository对应的是Entity对象读取存储的抽象，在领域模型中称为**仓库(Repository)**。在接口层面做统一，不关注底层实现。他的具体实现类都通过调用DAO实现各种操作，并通过工厂对象实现DO与领域实体的转化。

通过抽象出**仓库**，改变了业务代码的思维方式，让业务逻辑不再需要面向数据库编程，而是面向实体编程。他作为一个接口类，也比较容易通过Mock方式造数据进行测试。

于是，我们的代码图变成了这样：



2、抽象第三方服务

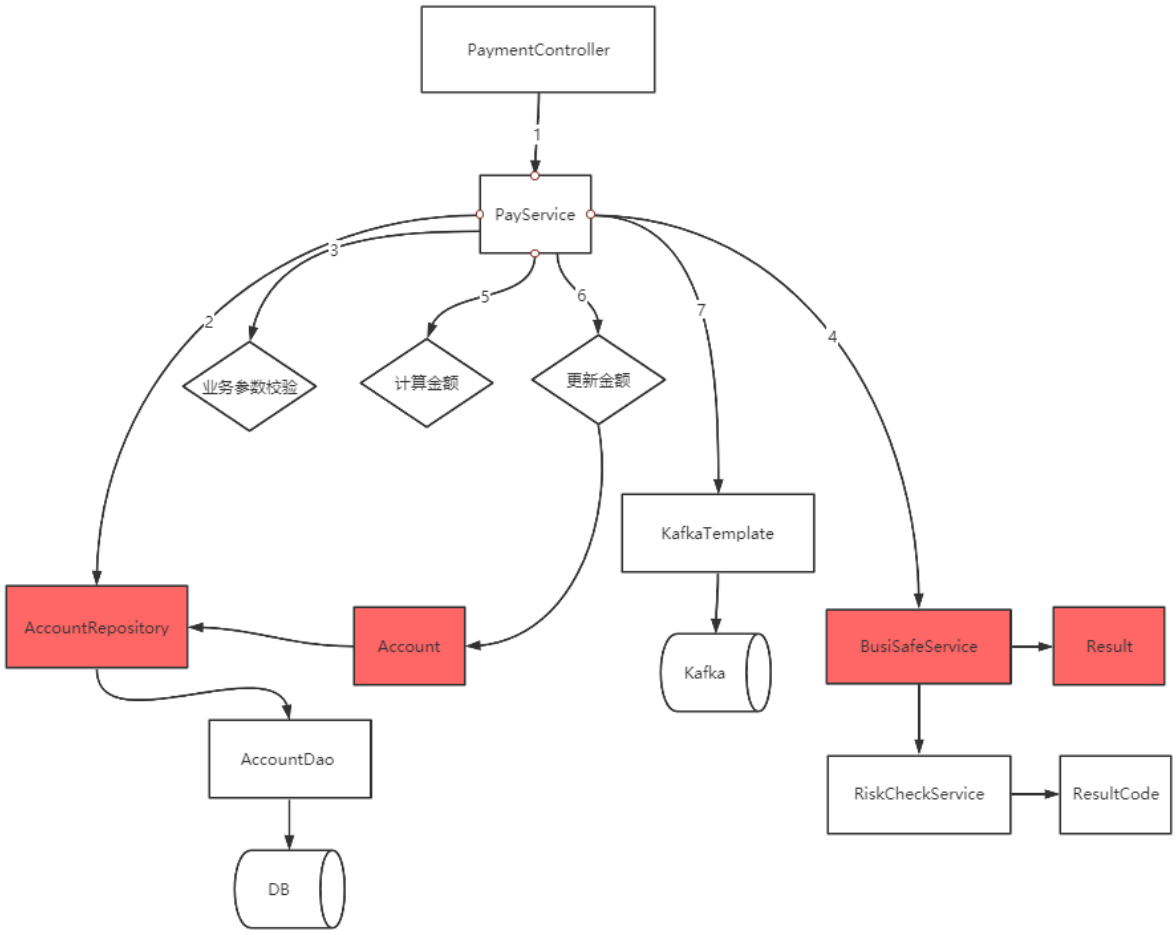
类似于数据库的抽象，所有第三方的服务也需要通过接口来进行隔离，以防止第三方服务状态不可控，入参出参强耦合的问题。例如，在这个例子中，针对交易安全，抽象出一个BusiSafeService的服务。

```
public interface BusiSafeService{
    .......
}
public class BusiSafeServiceImpl implements BusiSafeService{
    @Autowired
    private RiskChkService riskChkService;

    public Result checkBusi(Long userId,Long mechantAccount,BigDecimal money){
        //参数封装
        RiskCode riskCode = riskCheckService.checkPayment(...);
        if("0000".equals(reskCode.getCode()){
            return Result.SUCCESS;
        }
        return Result.REJECT;
    }
}
```

很多时候我们会去依赖其他的外部系统，而被依赖的系统可能包含不合理的数据结果、API、协议或者技术实现。如果对外部系统强依赖，会导致我们的系统被"腐蚀"。这时候，适当的加入一个接口层，能够有效隔离外部依赖和内部逻辑，无论外部如何变更，内部代码可以尽可能的保持稳定。这种常见的设计模式叫做**Anti-Conrruption Layer(ACL 防腐层)**。

防腐层的设计有非常多有效的实现方式。例如，使用适配器模式，可以扩展在不同数据类型之间进行数据转化的逻辑，降低对业务代码的侵入。另外，也可以在防腐层中添加一些对外部调用通用的逻辑，例如缓存、兜底、功能开关等。这些都可以帮助业务代码进行合理的业务抽象。

加入防腐层后，我们的代码就变成了这样：



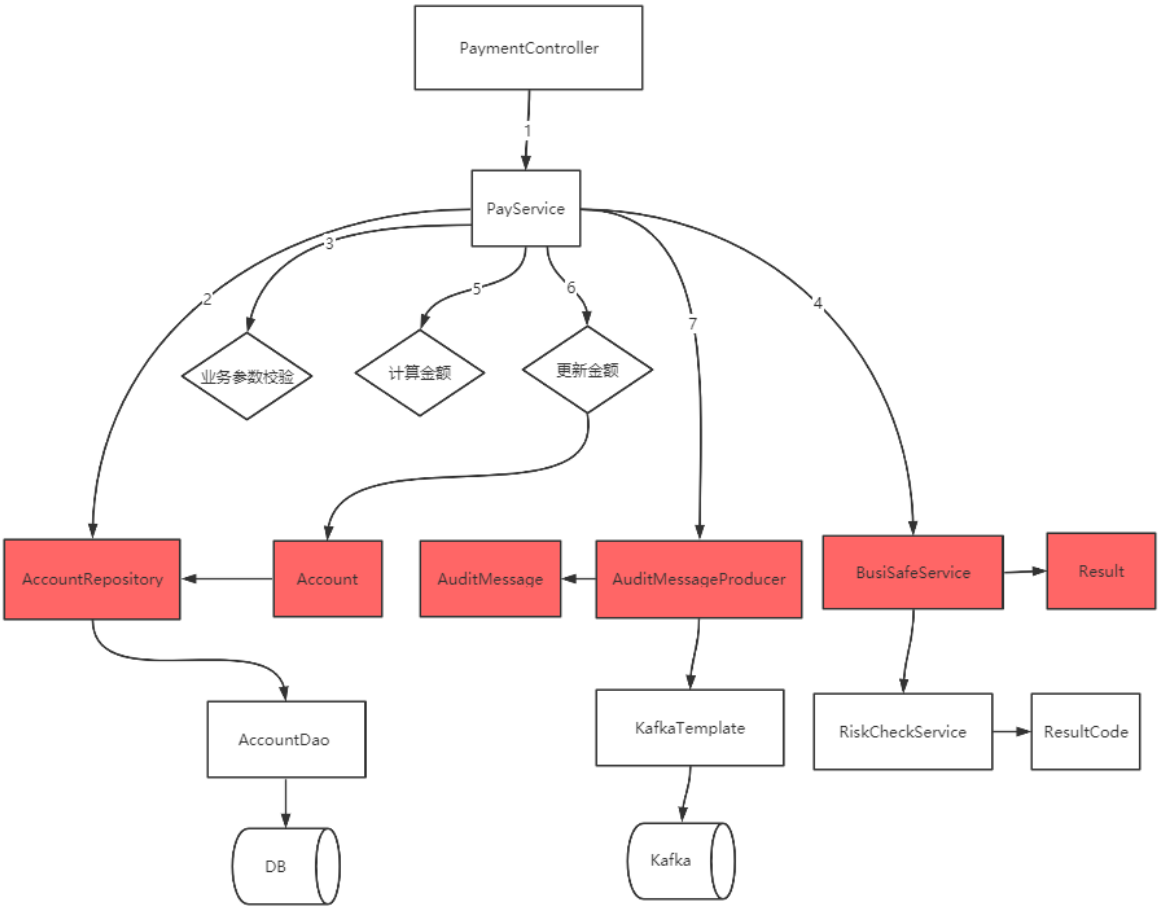
3、抽象中间件

接下来我们对kakfa发送消息这个步骤进行抽象。类似于上一步对第三方服务的抽象，可以在外部依赖的各个中间件之间设计一层ACL，让业务代码不再依赖具体的实现逻辑。这样，不管以后更换中间件或者修改消息传输协议，整个业务代码是不需要动的。

```
public class AuditMessage{
    private Long UserId;
    private Long clientAccount;
    private Long merchantAccount;
    private BigDecimal money;
    private Date data;
    .....
}
public interface AuditMessageProducer{
    ....
}
public class AuditMessageProducerImpl implements AuditMessageProducer{
    private KafkaTemplate<String,String> kafkaTemplate;
    public SendResult send(AuditMessage message){
        String messageBody = message.getBody();
        kafkaTemplate.send("some topic",messageBody);
        return SendResult.SUCCESS;
    }
}
```

这里的结果分析跟上一步抽象第三方服务类似。

经过抽象后，我们的代码就变成了这个样子：



4、重构业务逻辑

我们对外部依赖都一一做了梳理，已经可以减少外部依赖对核心业务的影响了。但是，还有很多业务逻辑跟外部依赖混合的代码。包括业务参数检查、账户余额校验、金额增减等。这些业务逻辑依然会影响核心代码的稳定性。这里，我们提供一种方案按照以下几个步骤来重新梳理业务逻辑。

4-1，抽象实体(Entity)和值对象(Value Object)：

核心的业务方法payService.pay(Long userId,String merchantAccount,BigDecimal amount)暴露出来的参数都是不具有类型意义的基础类型，这很容易造成理解偏差。传错参数这样的问题很难在开发时被发现。

所以，我们需要将userId, merchantAccount, amount这些参数值封装成一些具有业务意义的对象，即UserId,AccountNumber, Money这样的几个类，并在这些类中封装一些空判断之类的参数校验。这些类不需要具有唯一ID这样的属性，但是却能很好的表示实体之间的关系。这些类就被称为**值对象(Value Object)**。这样我们原始代码中的第2步业务参数校验就可以交由值对象去统一完成。这样既让业务代码得到检查，也能避免这些类似的参数校验逻辑被分散到各个业务代码中，对整个系统形成污染。

关于值对象，这里的理解是不够深刻全面的。但是这些类确实可以理解成是值对象。

4-2：用实体(Entity)封装单对象的状态行为

在我们之前抽象数据存储层时，曾经抽象出了一个Account这样的实体类。那现在，我们可以把原代码中关于金额增减这样的单对象的状态行为封装到实体类中。然后，我们原始代码中的第5步就可以简化为

```
clientAccount.deposit(money);
targetAccount.withdraw(money);
```

4-3：用领域服务(Domain Service)封装多实体逻辑

在这个案例里，我们发现两个账户的转入和转出操作实际上应该是一体的，也就是说这种行为应该要被封装到一起。特别是如果考虑到未来这个逻辑可能会发生变化，例如增加扣除手续费的逻辑。那这时这个转账操作放到PayServiec中就不合适了，而在任何一个Account中去做又无法保证操作的一体性。这时，就需要一个新的类去包含跨领域对象的行为。这种对象称为领域服务(Domain Service)。

我们创建一个AccountTransferService的领域服务类

```
public interface AccountTransferService{
    void transfer(Account sourceAccount,Account targetAccount,Money money);
}
public class AccountTransferServiceImpl implements AccountTransferService{
    public  void transfer(Account sourceAccount,Account targetAccount,Money
money){
        sourceAccount.deposit(money);
        targetAccount.withdraw(money);
    }
}
```

这样，原始业务代码中的6和7步骤简化成了一行：

```
accontTransferService.transfer(clientAccount,merchantAccount,money)
```

整体重构后的业务代码：

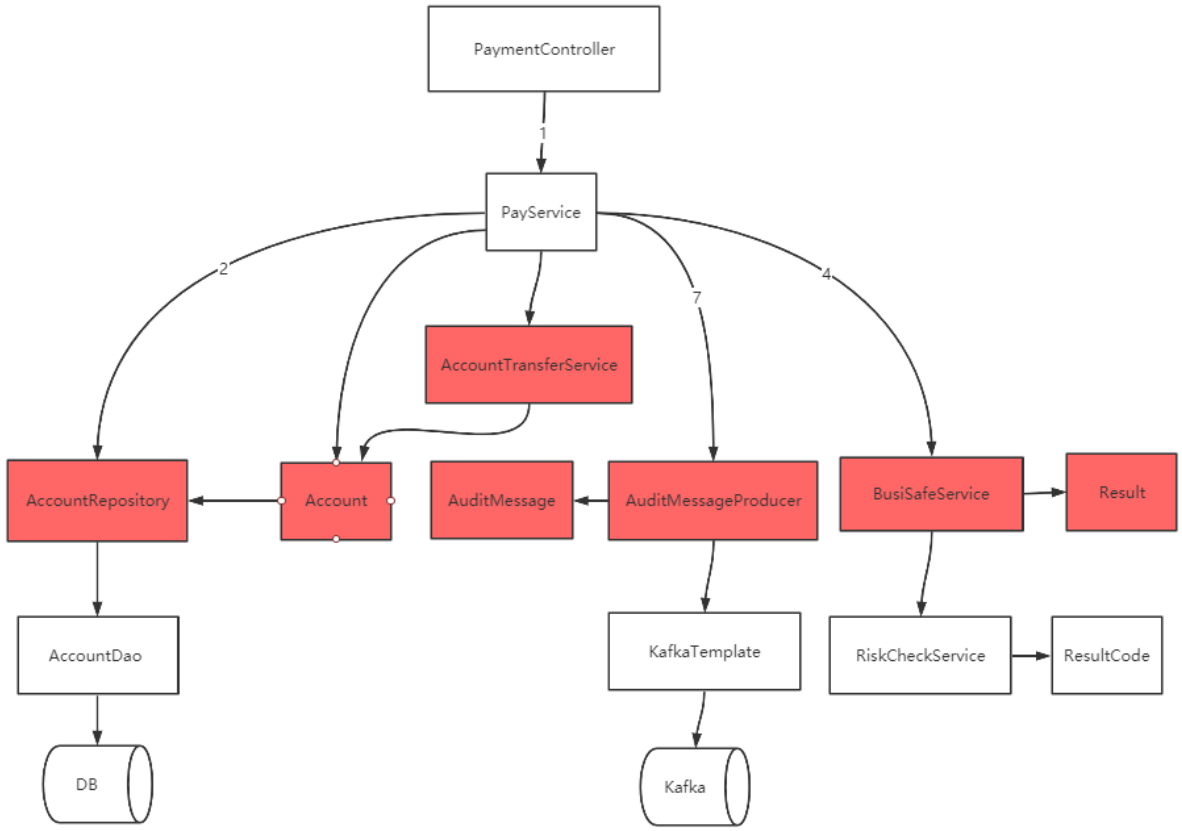
```
public class PaymentController{
    private PayService payService;
    public Result pay(String merchantAccount,BigDecimal amount){
        Long userId = (Long) session.getAttribute("userId");
        return payService.pay(userId, merchantAccount, amount);
    }
}

public class PayServiceImpl extends PayService{
    private AccountRepository accountRepository;
    private AuditMessageProducer auditMessageProducer;
    private BusiSafeService busiSafeService;
    private AccountTransferService accountTransferService;

    public Result pay(Long userId,String merchantAccount,BigDecimal amount){
        // 参数校验
        Money money = new Money(amount);
        UserId clientId = new UserId(userId);
        AccountNumber merchantNumber = new AccountNumber(merchantAccount);
        // 读数据
        Account clientAccount = accountRepository.find(clientId);
        Account merAccount = accountRepository.find(merchantNumber);
        // 交易检查
        Result preCheck =  busiSafeService。
checkBusi(clientAccount,merAccount,money);
        if(preCheck ! = Result.SUCCESS) {
            return Result.REJECT;
        }
```

```
        // 业务逻辑
        accountTransferService.transfer(clientAccount,merAccount,money);
        // 保存数据
        accountRepository.save(clientAccount);
        accountRepository.save(merAccount);
        // 发送审计消息
        AuditMessage message = new AuditMessage(clientAccount,
merAccount,money);
        auditMessageProducer.send(message);
        return Result.SUCCESS;
    }
}
```

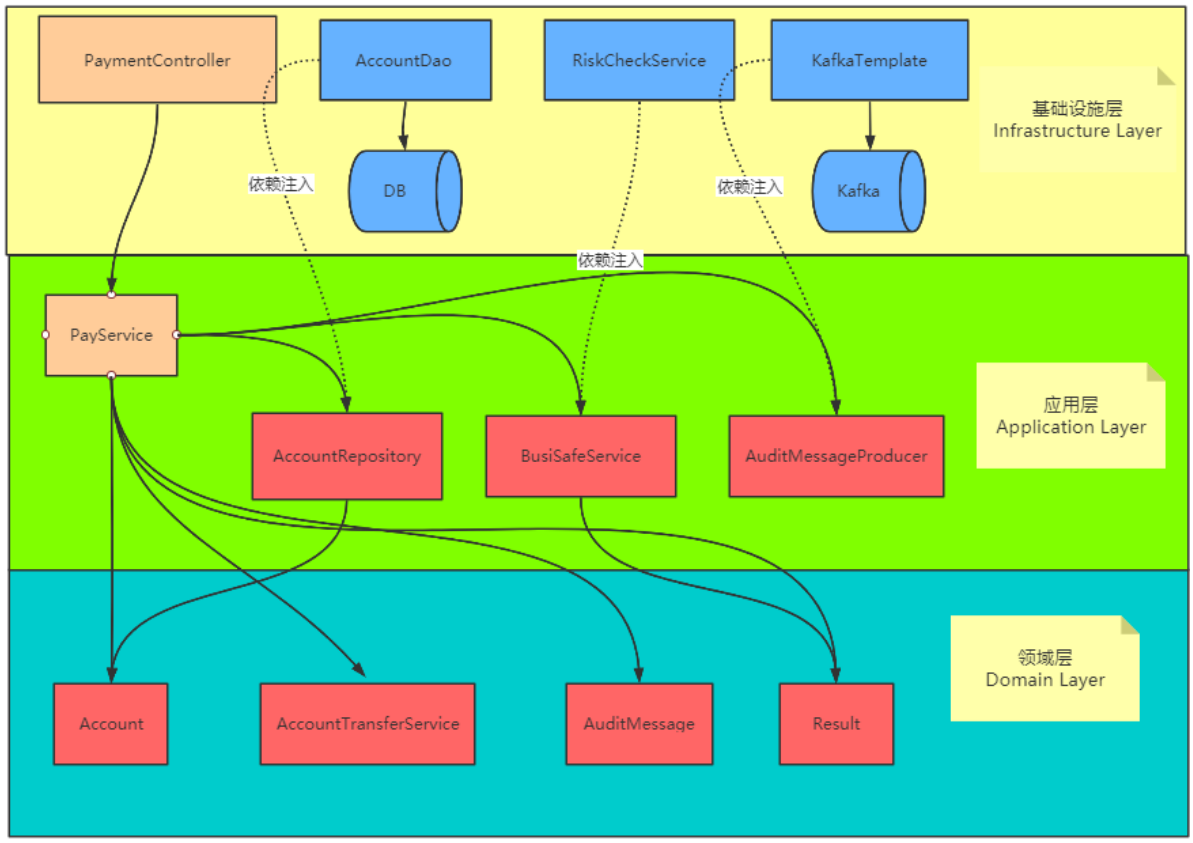
经过重构后的代码有以下几个特征：

- 业务逻辑清晰，数据流转与业务逻辑完全分离。
- Entity,ValueObject,DomainService 这些之前提到的对象都是完全独立的，没有外部依赖，却包含了所有核心业务逻辑，可以单独进行测试。他们共同构成了一个**领域(Domain)**
- 原有的PayService不再包含任何计算逻辑，仅仅作为组件编排

然后我们重新梳理下业务代码，整体就成了这样：



然后我们把这个图按照组件之间的依赖关系，重新编排一下，就变成了这样：

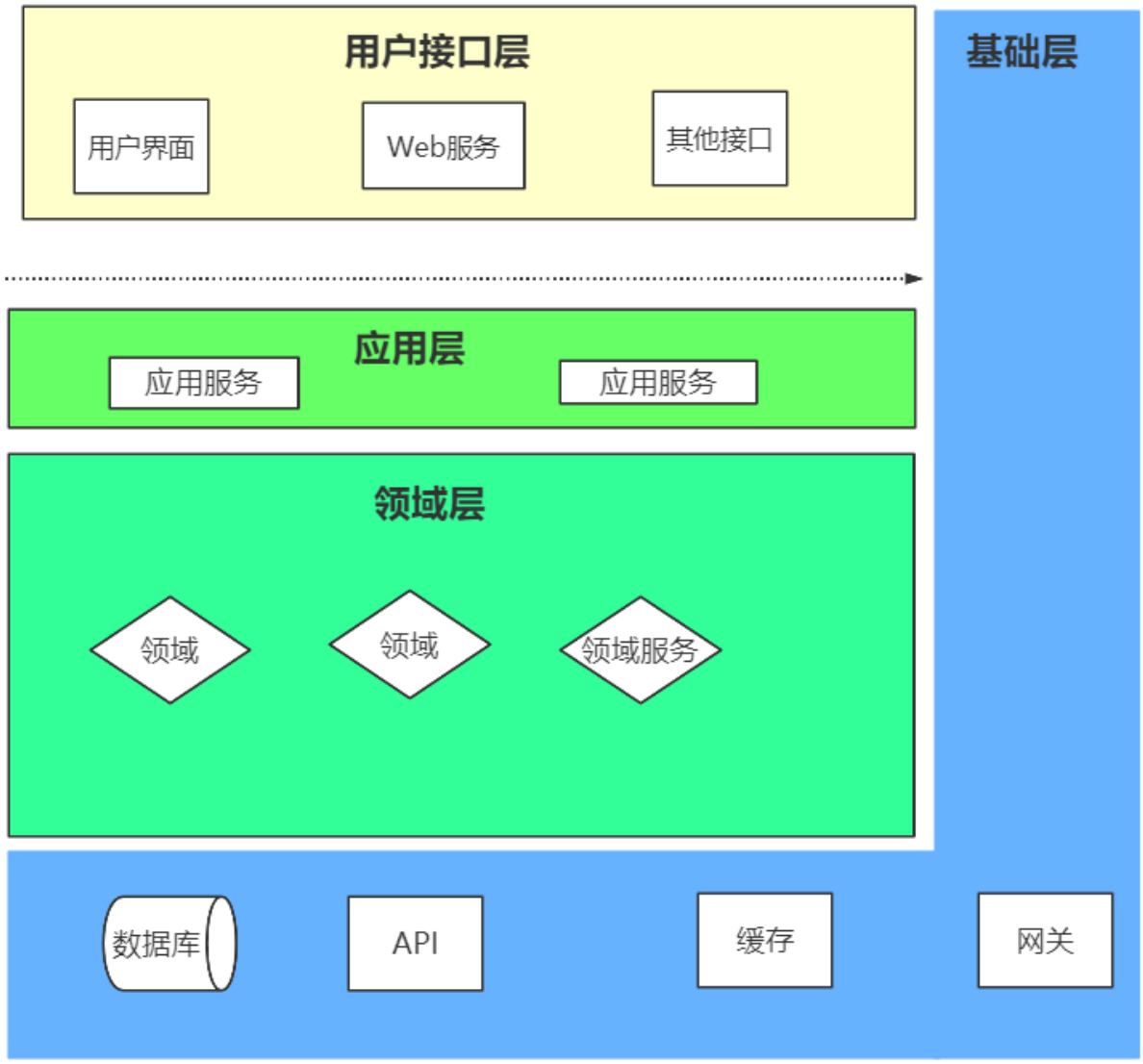


这时候我们再来看整个应用的依赖关系：

- 最底层不再是数据库，而是实体(Entity)，值对象(ValueObject)，领域服务(Domain Service)。这些对象都不依赖任何外部服务和框架。这些对象可以打包成一个**领域层(Domain Layer)**。领域层没有任何外部依赖关系。
- 原有的Service层，经过重新编排后，只依赖于一些抽象出来的防腐层(ACL)和仓库工厂(Repository)。他们的具体实现都是通过依赖注入进来的。他们一起负责整个组件的编排，这样就可以把他们打包成一个**应用层(Application Layer)**。应用层依赖领域层，但是不依赖具体实现。
- 最后是一些与外部依赖打交道的组件。这些组件的实现通常依赖外部具体的技术实现和框架，可以统称为**基础设施层(Infrastructure Layer)**。

四层架构

这样，我们对一个基于MVC的事务脚本经过重新编排后，开发业务时，可以优先开发领域层的业务逻辑，然后再写应用层的服务编排，而具体的外部依赖与具体实现可以最后再完成。整体就形成了一种以领域优先的架构形式。最后，我们整个应用会呈现这样一种结构：



而这个结构，就是Eric Evans在《领域驱动设计-软件核心复杂性应对之道》书中提到的DDD四层架构模型。而这种组织架构和代码的方式，就是**DDD 领域驱动设计**。

三、使用DDD优先实现单机版领域划分

接下来，回到服务开放平台项目，我们开始尝试使用DDD的四层架构对各个领域进行优化。

使用DDD之前，人们通常的做法是提前对需求做出预估，然后在实现层面做大量的提前设计，使用各种各样的设计模式来优化实现机制，给项目预留更多的扩展空间。这也是为什么越来越强调程序员要贴近业务的根源。但是每一种设计只能提前应对一部分需求问题，一旦需求变更没有按照预期推进，这些设计就会沦为鸡肋。即不能解决项目的实际需求，同时又增加了项目的复杂度。

而DDD强调技术主动理解业务，业务如何实现，程序就如何构建。DDD强调在对职责进行清晰地划分后，让技术只要“刚刚好”的解决业务问题即可。至于以后的需求变更，可以从领域划分的角度，拆分落实到不同的领域，然后交由各个领域分工合作。此时各种各样的设计模式、工具、框架等，就不再是解决业务的核心，而是随手拈来的工具。

这里由于问题域是未知的，所以肯定也没有办法说一下就能提供一个完美的解决方案。但是，有六个字你一定很熟悉。**高内聚，低耦合**。让整个系统内的各个业务模块功能尽量内聚，业务模块与业务模块之间减少相互依赖。这样即使面对更复杂的问题，也可以通过将问题拆解成为多个业务模块之间的协作。

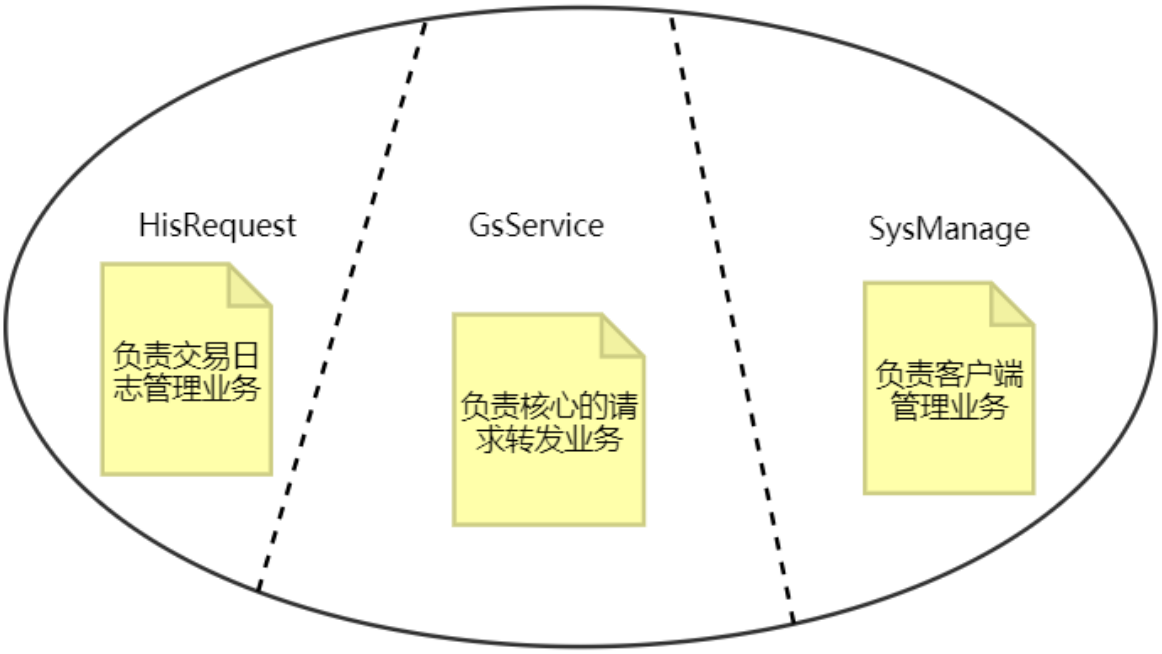
那要怎么设计一个高内聚、低耦合的高效的应用呢？这也同样没有完美的解决方案，同时，这也是IT人不断想要追求摸索的一个问题。每个架构师都有自己的一套理解，那有没有一套能够让架构师和程序员都能够理解的方法论呢？上一课在推荐DDD四层架构时，提到的单一职责原则、依赖反转原则、开放封闭原则，就是比较好的实践路线。提到这，有没有想起上一节课演练的DDD四层架构？他就是这三个原则的具体推演结果。所以，带着之前分析的这些问题，来体验一下用DDD实现的tlmall-openV2版本。

1、构建领域地图

DDD强调的是领域驱动设计，那么第一步当然是要划分领域。其实领域一词，我们并不陌生。但是领域的本质是什么？其实是边界。我们通常会说Java是一个领域，其实是为了跟C、golang等其他语言区分开。我们会说软件是一个领域，其实是为了跟硬件区分开。在生活当中，通过合理的领域划分，可以很好的复杂的大问题域拆分成一个个简单、明确的小问题域，从而减少整体的复杂度。简而言之，就是“专门的人干专门的事”。

这种领域划分的思想其实我们并不陌生。在很多需求比较明确的场景，我们经常会在Controller层对业务做比较明确的分工。比如，在上一章节的tlmall-open单体架构的示例中，我们很自然的按照项目的业务领域划分成了TransferController,SyaMangeController,HisRequestController三个领域。这种按照直觉划分的方式，在很多需求已经足够明确的场景，确实是一种最为容易推进的方式。

这样，自然而然我们就可以形成这个比较简单的领域地图：



当然，我们这里之所以轻松愉快的就形成了这样的领域地图，其实是源于在之前版本中，我们已经对服务开放平台的整体需求有了很明确的预演。但是在真实项目中，往往这个过程并不会如此愉快。因为领域并不是天然存在的，这需要依赖于软件团队对软件需求的合理分析。并且分析的过程，即不能太过偏向于真实的业务领域，也不能太过偏向于极度抽象的技术领域，需要在整个项目团队内形成共识，然后通过初步抽象的通用语言将设计结果固化下来。

关于领域如何划分，这时DDD实践过程中最为基础也是最为重要的一个步骤，会影响到项目后续的各个环节。而很多软件团队，由于赶时间，往往忽视了前期的设计过程。要么寥寥几笔把主线业务描述清楚就结束，技术团队落地时，需要做大量的健壮性补充，要么过于精细，一开始就在考虑数据应该怎么传递，怎么存储等等细节问题，造成团队内非核心技术人员参与度不高。因此，为了提高前期领域划分的合理性。在DDD中推荐了事件风暴会议这样的具体形式，也强调了统一语言的理论模型。如果你的软件团队对于DDD的落地实践把握还不是很充分时，建议尽量严格的按照DDD的方式进行前期项目设计，这对于后期软件交付是有恨到的好处的。

另外，在我们之前的推演过程中，也提到过，基于DDD的四层架构，可以很好的做一些单元测试。在很多项目团队中，也可以针对各个核心环节，优先构建单元测试案例，从而形成一些TDD测试驱动设计的实践场景。

2、使用四层架构巩固领域基础

当tlmall-open项目形成领域地图后，将包结构给竖过来。将原本按照技术层次划分出来的Controller、Service等等这些包结构换成按业务来划分包结构。这样，自然而然的就形成了tlmall-openV2中的clientmanage(负责App管理)，genservice(负责请求转发)，hisrequest(负责历史日志)三个包结构。每个包结构中包含了实现该业务所需要的全部代码。这样包与包之间就自然而然的形成了一些逻辑边界。实际上，这就抽象出了DDD中的三个领域。

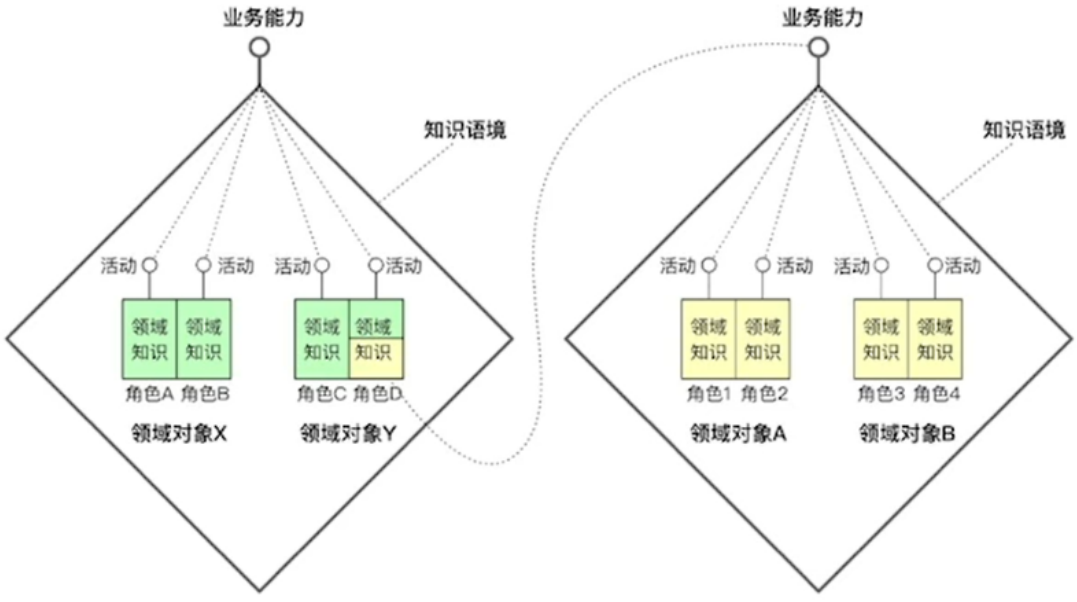
接下来，在每个领域内，按照之前的四层架构划分，自然而然也就能够实现业务与技术相分离的高内聚，低耦合的领域实现。例如对于clientmanage领域，就实现了业务实体与数据结构之间的分离。未来clientinfo领域内要做业务变革，可以 优先设计对应的业务实体，快速对业务进行试错。而至于表结构的变动，可以在业务变革完成之后再通过仓库延后实现，直接并且高效。

这种领域优先的设计方式，实际上就是典型的DDD风格。通过这种设计方式，实际上是将技术人员的思维方式调整成了业务人员的思维方式。因为在业务人员视野中，项目就是由一个一个功能模块组合而成的，至于Controller、Service这些，在业务人员视野中是不存在的。而通过这种领域优先的设计方式，技术人员也可以将业务人员日后的业务需求直接映射成项目中的模块结构，而不需要再将业务需求转换成产品需求。减少了模型转换的过程，自然也就提高了信息传递的效率。

3、划分限界上下文，巩固领域划分

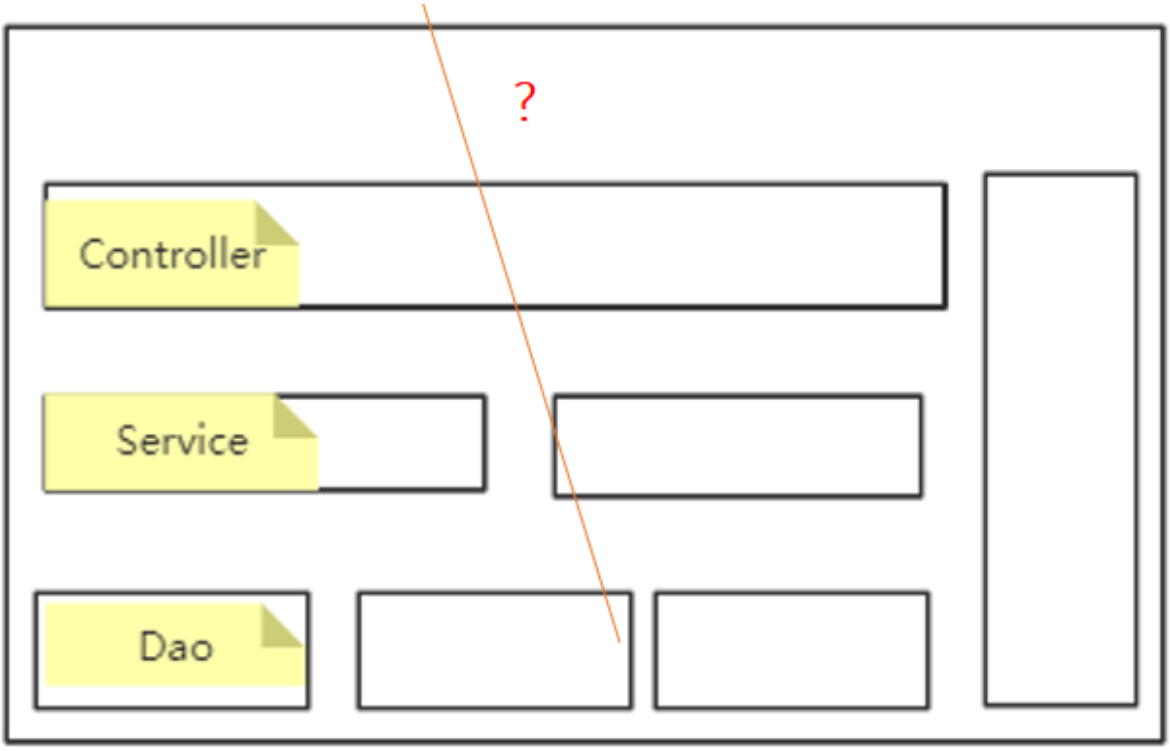
形成领域划分之后，项目中还有很多业务是需要领域之间进行合作的。比如genservice(请求转发)领域，在实现的过程中，还需要从clientmanager领域获取App的相关信息，也需要与hisrequest领域合作，及时记录日志。

在DDD的设计思想中，领域应该是足够内敛的，每个领域内的业务能力是与当前领域的知识语境紧密相连的。一个领域想要调用另外一个领域的业务能力，只能通过对方暴露出来的业务能力进行协作，而不能去干预对方领域的实现细节。这样，领域与领域之间就形成了一个逻辑边界，成为限界上下文。



限界上下文是领域模型的知识语境　　限界上下文是业务能力的纵向切分

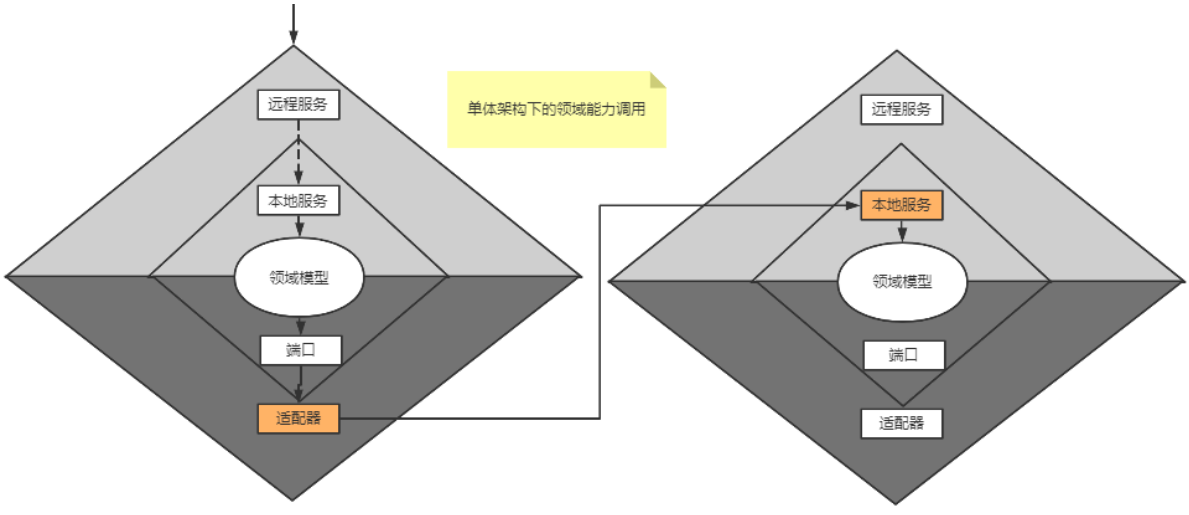
传统MVC设计方式中，强调的是技术隔离，而并没有从业务上的限界上下文边界，所以，逻辑边界是混乱的。这样的结构，在DDD中称为"大泥球"结构。这种"大泥球"结构越大，就越难以进行服务化拆分。

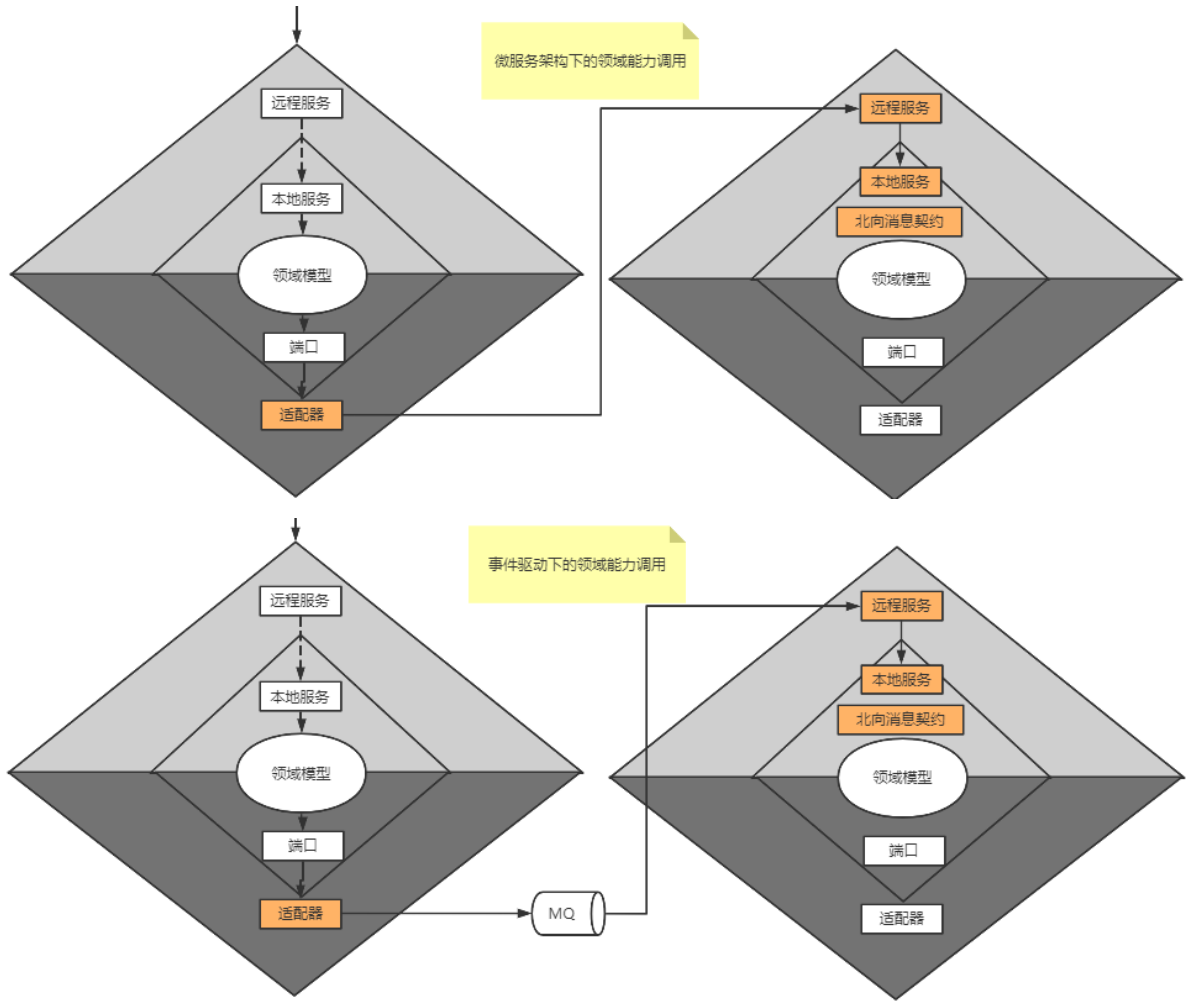


而在DDD中，会通过设计将逻辑上的限界上下文直接映射到代码结构中，在后续的设计中，也不断强调限界上下文的重要性。严守限界上下文的约束，这是DDD落地实践的关键。

DDD中关于这些限界上下文的设计是完善的，但是如何进行落地，是没有具体指导的。在tlmall-openV2中，给hisrequest和clientinfo两个领域都设计了一个adaptor层，在这里面，通过对应的接口，来将这两个领域的业务能力暴露给genservice。而genserivce领域，也通过adaptor层，暴露出对外的结构，这些接口，将会用来指导后端服务如何进行服务接入。这样，通过这些adaptor层的抽象，保证了不管领域外部的业务实现是怎么样的，领域内部的逻辑结构是比较稳定的。以后不管clientinfo领域对于App的管理方式发生什么样的变动，只要Adaptor层的接口是稳定的，对genservice领域来说，就不会有任何影响。

这种通过接口进行合作的方式有没有觉得很熟悉？是的，Dubbo、Feign这些微服务框架都是通过接口进行交互的。所以在未来，所谓的单体架构、微服务架构、甚至包括MQ事件驱动架构，在DDD的视野中，都是领域之间的不同协作方式而已，对领域内部都是没有影响的。日后只要调整接口层的协作方式，就可以让整个项目在不影响业务稳定性的基础上，进行灵活的架构调整。甚至，如果你的领域设计得足够全面，可以完全兼容这些不同的架构。





4、从单体架构开始快速验证

接下来考虑如何让后端服务快速接入的问题。这个问题需要由tlmall-open来制定一个后端服务接入的规范。在tlmall-open版本，这个规范非常的粗狂，只要后端服务提供一个feign接口，剩下的接入逻辑由tlmall-open自行组织。这样的实现方式会使得每一个后端服务在tlmall-open中都需要进行适配，不便于服务快速接入，服务接入的质量也无法得到保障。

1》tlmall-openV2中的解决方案

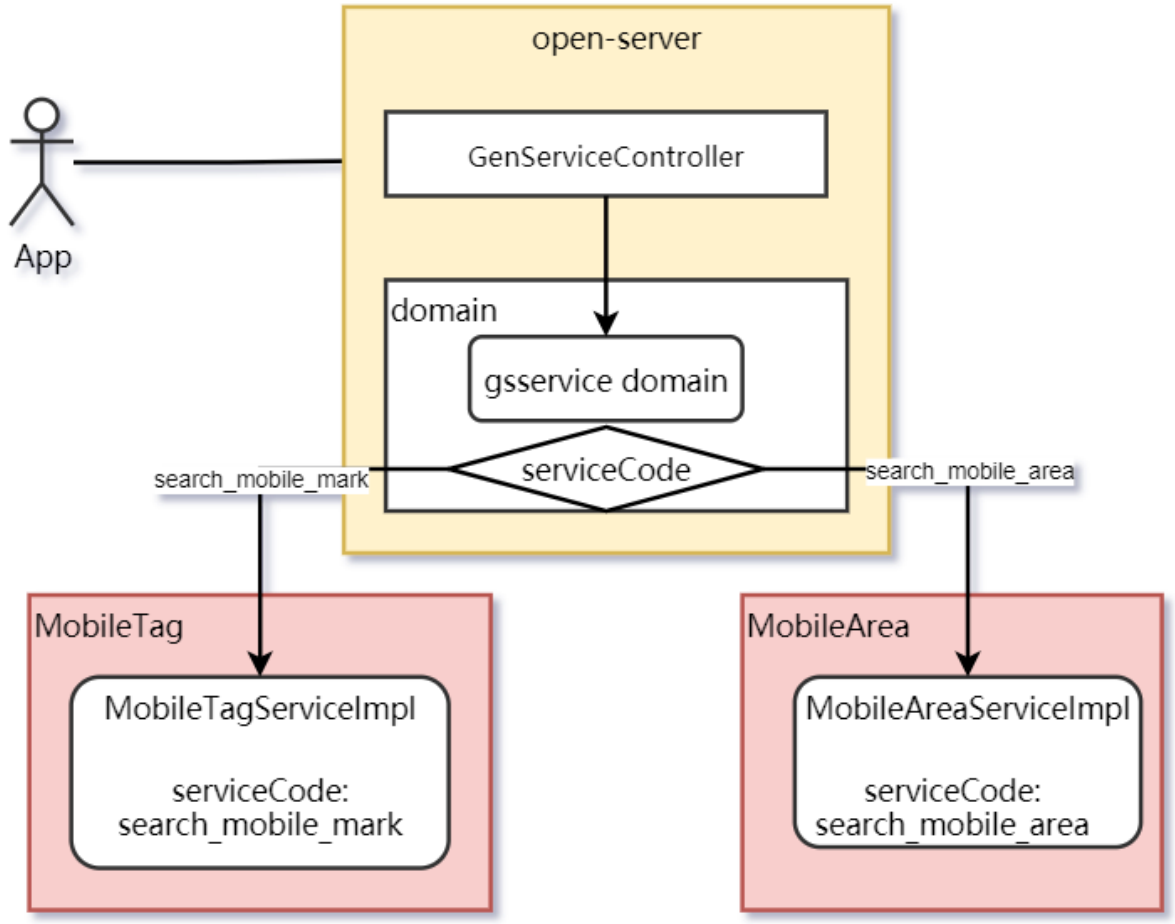
在这里，你首先需要思考的是，原来的接入规范不合理，那应该要指定一个怎样的接入规范呢？在下一节课中，我自然会带你实现一个基于Nacos的优化版的服务接入规范，让后端服务可以自由接入，而不需要影响tlmall-open-server服务本身，这是所有项目实战课都会带你做的事情。

但是在真实企业项目中，解决方案并不是现成的，是需要一点点摸索的。你要如何确定确实需要Nacos服务来支持的方案呢？当你项目原本没有使用Nacos，你向项目经理申请需要部署Nacos的时候，你要如何去申请？直接说老师已经帮你设计好了方案？你是不是需要先有个可行性的测试，来验证你的方案是否可行，然后再去向领导申请需要部署Nacos？

所以，在tlmall-openV2项目中，我会先带你实现一个基于本地SPI机制实现的服务接入策略。每个后端服务只需要提供一个GsService接口的实现类以及一个GsService的SPI配置文件，就可以接入到open-server当中。

在GsService接口中，有两个方法。

- serviceCode()方法返回当前服务支持App的哪种服务请求(由App申请的serviceCode参数决定)。
- doBusi方法则负责处理具体的服务。open-server会将App传入的所有业务请求都推送给当前服务的doBusi方法，由后端服务进行具体的业务处理。



以后在真实项目部署时，只要将open-server项目将所有依赖包打到同一个目录中，而后端服务，比如mobiletag和mobilearea，只要将jar包也部署到open-server项目的依赖目录中，就可以完成与open-server的接入了。

2》为什么要这么做？微服务时代，单体架构真的淘汰了吗？

为什么要这么做？其实自然是因为我们已经对基于Nacos的微服务体系有一定的了解。通过我们以往对于微服务架构的了解，其实很容易得到这样一个结论："**几乎所有基于接口的调用方式都可以很顺滑的从单体架构过渡到微服务架构**"。因为Dubbo、Feign这样的微服务RPC框架，本身就都是配合接口来使用的。我们可以很容易的将这种基于SPI的本地化接口扩展方式升级成为电商项目中的基于Nacos的分布式接口扩展方式。而基于DDD良好的领域隔离，这种改造方式对于各个业务领域原有的核心业务基本不会有太大的影响，我们甚至可以和某一次业务流程升级的常规迭代一起，顺滑的完成单体架构到微服务架构的升级。

其实，在这个示例中可以看到，tlmall-openV2实际上是先将tlmall-open退化成了一个单体架构，但是项目的逻辑结构是得到了优化的。这也涉及到互联网一个旷日持久的争论，微服务时代了，单体架构还有用吗？

现在微服务技术大行其道，你写个Demo应用可能都会直接用上微服务，单体架构成了你眼中鄙视链的底层。但是，微服务概念的提出者Martin Fowler，在提出微服务时，却提出了另外一个与微服务有点冲突的观点：MonolithFirst单体架构优先。

> 参见 https://martinfowler.com/bliki/MonolithFirst.html

MonolithFirst

3 June 2015

Martin Fowler

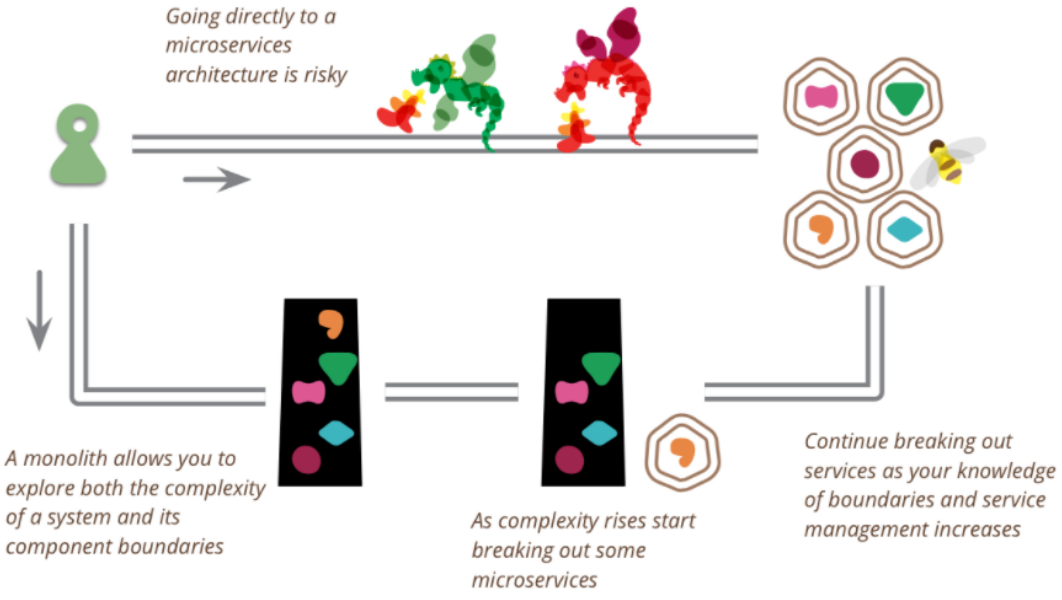
EVOLUTIONARY DESIGN
MICROSERVICES

As I hear stories about teams using a microservices architecture, I've noticed a common pattern.

1. Almost all the successful microservice stories have started with a monolith that got too big and was broken up
2. Almost all the cases where I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble.

- 几乎所有成功应用微服务的系统都由一个过大单体项目拆分而来。
- 几乎所有我听说过的一开始就选择使用微服务架构，并从0开始构建的系统，最终都陷入到严重的麻烦当中。

This pattern has led many of my colleagues to argue that **you shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile.** .



在你所忽视的单体架构中，其实蕴涵着非常大的重要性。只不过，由于在Martin Fowler提出这篇文章时，微服务架构的技术环境还远没有现在这么丰富，所以很多人看不到如何从一个单体架构顺滑的过渡到微服务架构。而在微服务技术已经非常完善的今天，这种技术视野就成了一种理所当然的本能。

但是，要注意，通过接口的方式来落地DDD中的限界上下文，是一种很好的方式。但是并不是DDD的标准解决方案。在不同的技术场景下，限界上下文也会有不同的落地方式。甚至在很多技术场景下，并没有现成的框架支持。很多互联网公司甚至都不是基于开源的框架来构建自己的应用。但是，如果你目前没有这么丰富的开发经验，不妨在后续学习微服务技术的同时，都回头想想这个问题。

至此，我们已经可以规划后续的改造过程了。我们可以将每个后端服务的GsService接口的ServiceCode实现，改为Nacos服务发现体系中的服务名。而Naocs基于HTTP的实现方式，也很方便可以通过服务名调用到远端的微服务，实现GsService接口中doBusi方法的业务逻辑，这样就能形成一套能够兼容之前电商项目的服务接入规范了。

当然，在具体实现过程中，还是有一些问题需要解决的，但是这都已经不影响项目整体的架构设计了。下一节课就会带你具体实现这种规范，并接入电商项目的内部服务。

总结

在这一章节中，我们先从一个简单的业务场景，按照我们最为熟悉的MVC进行初步构建，并一步步推演出了DDD的很多基础概念以及四层架构。这些就是我们使用DDD最基础的武器。

接下来，我们就用这些基础武器对tulingmall-open服务开放平台进行改造，实现了tulingmall-openV2的单体架构版本。在这个过程中，系统架构虽然是被降级了，但是整个结构变得更清晰，扩展性也更明显。很多之前看来比较棘手的问题，只要按照DDD的概念进行落地，就自然而然的看到了方向。

然后，我们以DDD的视角重新审视了一下单机架构与微服务架构之间的关系。项目设计得是否高效，并不取决于用了多么复杂的技术或者用了多么高大上的设计，简单的单体架构经过良好的优化设计，也可以有很强大的生命力。越是复杂的项目，就越应该重视单体架构的快速验证和快速试错，这对于提高复杂项目的运行效率是非常重要的。当然，你要不要接受这个结论，还是需要你自己去思考和总结。

最后，还记得最开始你对tlmall-open的吐槽吗？不妨拿出来，自己尝试下改造把。

> 有道云分享地址： https://note.youdao.com/s/2mVJsLVI