架构师如何应对复杂业务场景？领域建模的实战案例解析

阿里妹导读：你还在用面向对象的语言写面向过程的代码吗？你是否正在被复杂的业务逻辑折磨？是否有时觉得应用开发没意思、没挑战、技术含量低？其实，应用开发一点都不简单，也不无聊，业务的变化比底层基础实施的变化要多得多，封装这些变化需要很好的业务理解力，抽象能力和建模能力。

今天我们邀请阿里高级技术专家张建飞，一起来聊聊为什么需要领域建模，什么是好的模型，又该如何搭建。

为什么要领域建模？

软件的世界里没有银弹，是用事务脚本还是领域模型没有对错之分，关键看是否合适。实际上，CQRS就是对事务脚本和领域模型两种模式的综合，因为对于Query和报表的场景，使用领域模型往往会把简单的事情弄复杂，此

时完全可以用奥卡姆剃刀把领域层剃掉，直接访问Infrastructure。我个人也是坚决反对过度设计的，因此对于简单业务场景，我强力建议还是使用事务脚本，其优点是简单、直观、易上手。但对于复杂的业务场景，你再这么玩就不行了，因为一旦业务变得复杂，事务脚本就很难应对，容易造成代码的“一锅粥”，系统的腐化速度和复杂性呈指数级上升。

目前比较有效的治理办法就是领域建模，因为领域模型是面向对象的，在封装业务逻辑的同时，提升了对象的内聚性和重用性，因为使用了通用语言（Ubiquitous Language），使得隐藏的业务逻辑得到显性化表达，使得复杂性治理成为可能。talk is cheap，直接上一个银行转账的例子，对事务脚本和领域模型进行比较，孰优孰劣一目了然。

银行转账事务脚本实现

在事务脚本的实现中，关于在两个账号之间转账的领域业务逻辑都被写在了MoneyTransferService的实现里面了，而Account仅仅是getters和setters的数据结构，也就是我们说的贫血模式。

```
public class MoneyTransferServiceTransactionScriptImpl
      implements MoneyTransferService {
  private AccountDao accountDao;
  private BankingTransactionRepository bankingTransactionRepository;
  . . .
  @Override
  public BankingTransaction transfer(
      String fromAccountId, String toAccountId, double amount) {
    Account fromAccount = accountDao.findById(fromAccountId);
    Account toAccount = accountDao.findById(toAccountId);
    . . .
    double newBalance = fromAccount.getBalance() - amount;
    switch (fromAccount.getOverdraftPolicy()) {
    case NEVER:
      if (newBalance < 0) {
        throw new DebitException("Insufficient funds");
      }
      break;
    case ALLOWED:
      if (newBalance < -limit) {
        throw new DebitException(
            "Overdraft limit (of " + limit + ") exceeded: " + newBalance);
      }
      break;
    }
    fromAccount.setBalance(newBalance);
    toAccount.setBalance(toAccount.getBalance() + amount);
    BankingTransaction moneyTransferTransaction =
        new MoneyTranferTransaction(fromAccountId, toAccountId, amount);
    bankingTransactionRepository.addTransaction(moneyTransferTransaction);
    return moneyTransferTransaction;
  }
}
```

上面的代码大家看起来应该比较眼熟，因为目前大部分系统都是这么写的。**需求评审完，工程师画几张UML图完成设计，就开始向上面这样怼业务代码了，这样写基本不用太费脑，完全是面向过程的代码风格**。有些同学可能会说，我这样写也可以实现系统功能啊，还是那句话“just because you can, doesn't mean you should”。说句不好听的，正是有这么多“没有追求”、“不求上进”的码农才造成了应用系统的混乱、败坏了应用开发的名声。这也是为什么很多应用开发工程师觉得工作没意思，技术含量低，觉得整天就是写if-else的业务逻辑代码，系统又烂，工作繁琐、无聊、没有成长、没有成就感，所以转向去做中间件啊，去写JDK啊，觉得那个NB。

实际上，应用开发一点都不简单也不无聊，业务的变化比底层Infrastructure的变化要多得多，解决的难度也丝毫不比写底层代码容易，只是很多人选择了用无聊的方式去做。其实我们是有办法做的更优雅的，这种优雅的方式就是领域建模，唯有掌握了这种优雅你才能实现从工程师向应用架构的转型。

同样的业务逻辑，接下来就让我们看一下用DDD是怎么做的。

银行转账领域模型实现

如果用DDD的方式实现，Account实体除了账号属性之外，还包含了行为和业务逻辑，比如debit()和credit()方法。

```
// @Entity
public class Account {
  // @Id
  private String id;
  private double balance;
  private OverdraftPolicy overdraftPolicy;
  . . .
  public double balance() { return balance; }
  public void debit(double amount) {
    this.overdraftPolicy.preDebit(this, amount);
    this.balance = this.balance - amount;
    this.overdraftPolicy.postDebit(this, amount);
  }
  public void credit(double amount) {
    this.balance = this.balance + amount;
  }
}
```

而且透支策略OverdraftPolicy也不仅仅是一个Enum了，而是被抽象成包含了业务规则并采用了策略模式的对象。

```
public interface OverdraftPolicy {
  void preDebit(Account account, double amount);
  void postDebit(Account account, double amount);
}
public class NoOverdraftAllowed implements OverdraftPolicy {
  public void preDebit(Account account, double amount) {
    double newBalance = account.balance() - amount;
    if (newBalance < 0) {
      throw new DebitException("Insufficient funds");
    }
  }
  public void postDebit(Account account, double amount) {
  }
}
public class LimitedOverdraft implements OverdraftPolicy {
  private double limit;
  . . .
  public void preDebit(Account account, double amount) {
    double newBalance = account.balance() - amount;
    if (newBalance < -limit) {
      throw new DebitException(
          "Overdraft limit (of " + limit + ") exceeded: " + newBalance);
    }
  }
  public void postDebit(Account account, double amount) {
  }
}
```

而Domain Service只需要调用Domain Entity对象完成业务逻辑即可。

```
public class MoneyTransferServiceDomainModelImpl
      implements MoneyTransferService {
  private AccountRepository accountRepository;
  private BankingTransactionRepository bankingTransactionRepository;
  . . .
  @Override
  public BankingTransaction transfer(
      String fromAccountId, String toAccountId, double amount) {
    Account fromAccount = accountRepository.findById(fromAccountId);
    Account toAccount = accountRepository.findById(toAccountId);
    . . .
    fromAccount.debit(amount);
    toAccount.credit(amount);
    BankingTransaction moneyTransferTransaction =
        new MoneyTranferTransaction(fromAccountId, toAccountId, amount);
    bankingTransactionRepository.addTransaction(moneyTransferTransaction);
    return moneyTransferTransaction;
  }
}
```

通过上面的DDD重构后，原来在事务脚本中的逻辑，被分散到Domain Service，Domain Entity和OverdraftPolicy三个满足SOLID的对象中，在继续

阅读之前，我建议可以自己先体会一下DDD的好处。

领域建模的好处

面向对象

- 封装：Account的相关操作都封装在Account Entity上，提高了内聚性和可重用性。
- 多态：采用策略模式的OverdraftPolicy（多态的典型应用）提高了代码的可扩展性。

业务语义显性化

- 通用语言：“一个团队，一种语言”，将模型作为语言的支柱。确保团队在内部的所有交流中，代码中，画图，写东西，特别是讲话的时候都要使用这种语言。例如账号，转账，透支策略，这些都是非常重要的领域概念，如果这些命名都和我们日常讨论以及PRD中的描述保持一致，将会极大提升代码的可读性，减少认知成本。
- 显性化：就是将隐式的业务逻辑从一推if-else里面抽取出来，用通用语言去命名、去写代码、去扩展，让其变成显示概念，比如“**透支策略”这个重要的业务概念，按照事务脚本的写法，其含义完全淹没在代码逻辑中没有突显出来**，看代码的人自然也是一脸懵逼，而领域模型里面将其用策略模式抽象出来，不仅提高了代码的可读性，可扩展性也好了很多。

如何进行领域建模?

建模方法

领域建模这个话题太大，关于此的长篇大论和书籍也很多，比如什么通过语法和句法深入分析法，在我看来这些方法论有些繁琐了。好的模型应该是建立在对业务深入理解的基础上，如果业务理解不到位，你再怎么分析句子也不可能产出好的模型。就我自己的经验而言，建模也是一个不断迭代的过程，所以一开始可以简单点来，就采用**两步建模法**抓住一些核心概念，然后写一些代码验证一下run一下，看看顺不顺，如果很顺滑，说明没毛病，否则就要看看是不是需要调整一下模型，随着项目的进行和对业务理解的不断深入，这种迭代将持续进行。

那什么是两步建模法呢？也就是只需要两个步骤就能建模了，**首先从User Story找名词和动词，然后用UML类图画出领域模型**。是不是很简约？简约并不意味着简单，对于业务架构师和系统分析师来说，见功力的地方往往就在于此。

举个栗子，比如让你设计一个中介系统，一个典型的User Story可能是“小明去找工作，中介说你留个电话，有工作机会我会通知你”，这里面的关键名词很可能就是我们需要的领域对象，小明是求职者，电话是求职者的属性，中介包含了中介公司，中介员工两个关键对象；工作机会肯定也是关键领域对象；通知这个动词暗示我们这里用观察者模式会比较合适。**然后再梳理一下领域对象之间的关系，一个求职者可以应聘多个工作机会，一个工作机会也可以被多个求职者应聘，M2M的关系，中介公司可以包含多个员工，O2M的关系**。对于这样简单的场景，这个建模就差不多了。

当然我们的业务场景往往比这个要复杂，而且不是所有的名词都是领域对象也可能是属性，也不是所有的动词都是方法也可能是领域对象，所以要具体问题具体对待，这个对待的过程需要我们有很好的业务理解力，抽象能力以及建模的经验（知道为什么公司的job model里那么强调技术人员的业务理解

力和抽象能力了吧），比如通常情况下，价格和库存只是订单和商品的一个属性，但是在阿里系电商业务场景下，价格计算和库存扣减的复杂程度可以让你怀疑人生，因此作为电商中台，把价格和库存单独当成一个域（Domain）去对待是很必要的。

另外，建模不是一个一次性的工作，往往随着业务的变化以及我们对业务的理解越来越深入才能看清系统的全貌，所以迭代重构是免不了的，也就是要Agile Modelling。

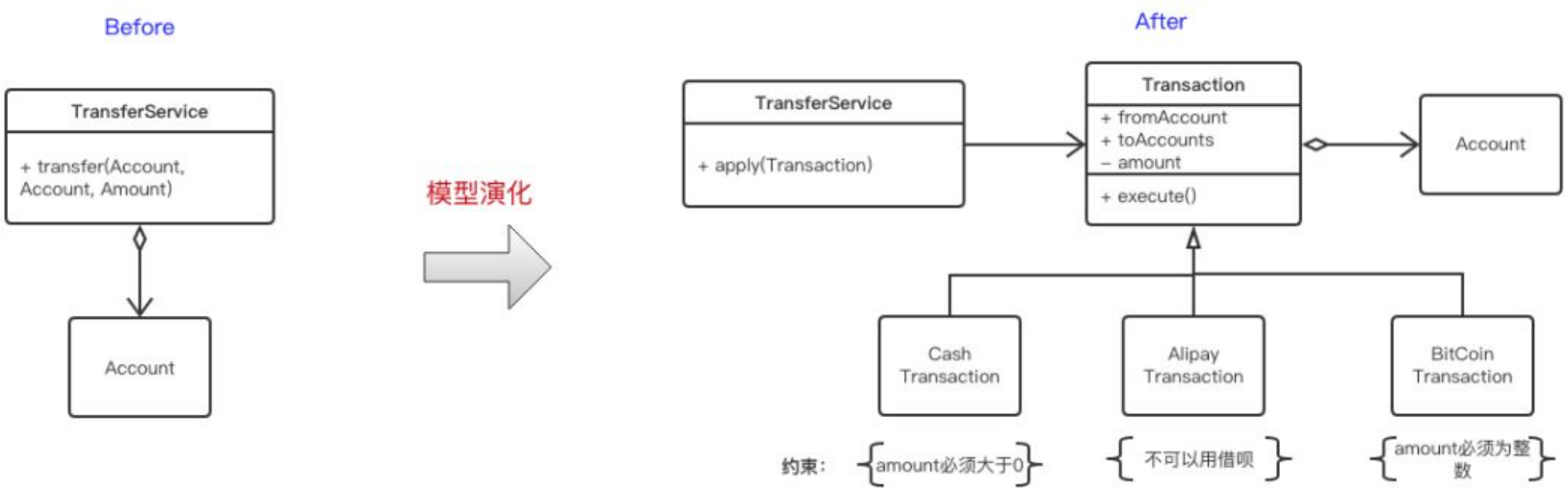
模型统一和模型演化

建模的过程很像盲人摸象，不同背景人用不同的视角看同一个东西，其理解也是不一样的。比如两个盲人都摸到大象鼻子，一个人认为是像蛇（活的能动），而另一个人认为像消防水管（可以喷水），那么他们将很难集成。双方都无法接受对方的模型，因为那不符合自己的体验。**事实上，他们需要一个新的抽象，这个抽象需要把蛇的“活着的特性”与消防水管的“喷水功能”合并到一起**，而这个抽象还应该排除先前两个模型中一些不确切的含义和属性，比如毒牙，或者卷起来放到消防车上去的行为。这就是模型的统一。

世界上唯一不变的就是变化，模型和代码一样也需要不断的重构和精化，每一次的精化之后，开发人员应该对领域知识有了更加清晰的认识。这使得理解上的突破成为可能，之后，一系列快速的改变得到了更符合用户需要并更加切合实际的模型。其功能性及说明性急速增强，而复杂性却随之消失。

这种突破需要我们对业务有更加深刻的领悟和思考，然后再加上重构的勇气和能力，勇气是项目工期很紧你敢不敢重构，能力是你有没有完备的CI保证你的重构不破坏现有的业务逻辑。还是以开篇的转账来举个例子，假如转账业务开始变的复杂，要支持现金，信用卡，支付宝，比特币等多种通道，且

没种通道的约束不一样，还要支持一对多的转账。那么你还是用一个transfer(fromAccount, toAccount)就不合适了，可能需要抽象出一个专门的领域对象Transaction，这样才能更好的表达业务，其演化过程如下：



领域服务

什么是领域服务？

有些领域中的动作，它们是一些动词，看上去却不属于任何对象。它们代表了领域中的一个重要的行为，所以不能忽略它们或者简单地把它们合并到某个实体或者值对象中。**当这样的行为从领域中被识别出来时，最佳实践是将它声明成一个服务**。这样的对象不再拥有内置的状态。它的作用仅仅是为领域提供相应的功能。Service往往是以一个活动来命名，而不是Entity来命名。例如开篇转账的例子，转账（transfer）这个行为是一个非常重要的领域概念，但是它是发生在两个账号之间的，归属于账号Entity并不合适，因为一个账号Entity没有必要去关联他需要转账的账号Entity，这种情况下，使用MoneyTransferDomainService就比较合适了。

识别领域服务，主要看它是否满足以下三个特征：

1. 服务执行的操作代表了一个领域概念，这个领域概念无法自然地隶属于一个实体或者值对象。
2. 被执行的操作涉及到领域中的其他的对象。
3. 操作是无状态的。

应用服务和领域服务如何划分?

在领域建模中，我们一般将系统划分三个大的层次，即应用层（Application Layer），领域层（Domain Layer）和基础实施层（Infrastructure Layer），关于这三个层次的详细内容可以参考我的另一篇SOFA框架的分层设计。可以看到在App层和Domain层都有服务（Service），这两个Service如何划分呢，什么样的功能应该放在应用层，什么样的功能应该放在领域层呢?

决定一个服务（Service）应该归属于哪一层是很困难的。如果所执行的操作概念上属于应用层，那么服务就应该放到这个层。**如果操作是关于领域对象的，而且确实是与领域有关的、为领域的需要服务，那么它就应该属于领域层。**总的来说，涉及到重要领域概念的行为应该放在Domain层，而其它非领域逻辑的技术代码放在App层，例如参数的解析，上下文的组装，调用领域服务，消息发送等。还是银行转账的case为例，下图给出了划分的建议：

Partitioning Services into Layers

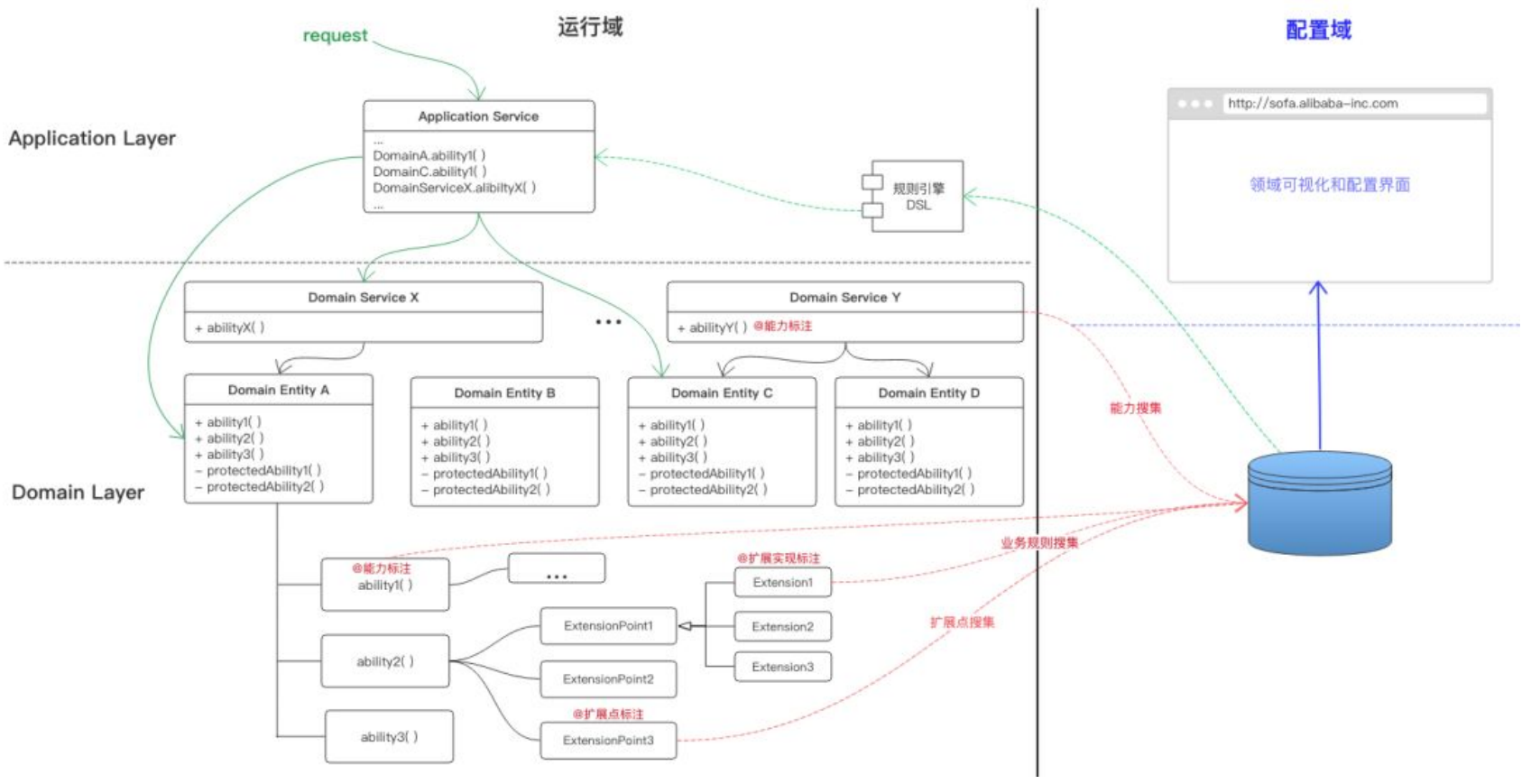
Application	Funds Transfer App Service: 1.Digests input (e.g. XML request), 2.sends message to domain service for fulfillment, 3.listens for confirmation, 4.decides to send notification using infrastructure service.
Domain	Funds Transfer Domain Service: Interacts with necessary Account and Ledger objects, making appropriate debits and credits, supplies confirmation of result (transfer allowed or not, etc.)
Infrastructure	Send Notification Service: Sends emails, letters, etc. as directed by application.

业务可视化和可配置化

好的领域建模可以降低应用的复杂性，而可视化和可配置化主要是帮助大家（主要是非技术人员，比如产品，业务和客户）直观地了解系统和配置系统，提供了一种“code free”的解决方案，也是SaaS软件的主要卖点。要注意的是可视化和可配置化难免会给系统增加额外的复杂度，必须慎之又慎，最好是能使可视化和配置化的逻辑与业务逻辑尽量少的耦合，否则破坏了原有的架构，把事情搞的更复杂就得不偿失了。

在可扩展设计中，我已经介绍了我们SOFA架构是如何通过扩展点的设计来支撑不同业务差异化的需求的，**那么可否更进一步，我们将领域的行为（也叫能力）和扩展点用可视化的方式呈现出来，并对于一些不需要编码实现的扩展点用配置的方式去完成呢**。当然是可以的，比如还是开篇转账的例子，对于透支策略OverdraftPolicy这个业务扩展点，新来一个业务说透支额度不能超过1000，我们可以完全结合规则引擎进行配置化完成，而不需要编码。

所以我能想到的一种还比较优雅的方式，是通过Annotation注解的方式对领域能力和扩展点进行标注，然后在系统bootstrap阶段，通过代码扫描的方式，将这些能力点和扩展点收集起来上传到中心服务器，然后再通过GUI的方式呈现出来，从而做到业务的可视化和可配置化。大概的示意图如下：



有同学可能会问流程要不要可视化，这里要分清楚两个概念，业务逻辑流和工作流，很多同学混淆了这两个概念。业务逻辑流是响应一次用户请求的业务处理过程，其本身就是业务逻辑，对其编排和可视化的意义并不是很大，无外乎只是把代码逻辑可视化了，在我们的SOFA框架中，是通过扩展点和策略模式来处理业务的分支情况，而我看到我们阿里很多的内部系统将这种响应一次用户请求的业务逻辑用很重的工作流引擎来做，美其名曰流程可编排，实质上往往是把简单的事情复杂化了。而工作流是指完成一项任务所需要不同节点的连接，节点主要分为自动节点和人工节点，其中每个人工节点都需要用户的参与，也就是响应一次用户的请求，比如审批流程中的经理审批节点，CRM销售过程的业务员的处理节点等等。

此时可以考虑使用工作流引擎，特别是当你的系统需要让用户自定义流程的时候，那就不得不使用可视化和可配置的工作流引擎了，除此之外，最好不要自找麻烦。当然也不排除有用的特别合适的案例，只是我还没看见，如果有看见的同学也请告诉我一声，一起交流学习。

End

你可能还喜欢

点击下方图片即可阅读

[分布式存储系统的一致性是什么？]

[知识图谱数据构建的“硬骨头”，](#)

[阿里工程师如何拿下？](#)

[如何用架构师思维解读区块链技术？](#)

关注「阿里技术」

把握前沿技术脉搏