

12 | 实战一（下）：如何利用基于充血模型的DDD开发一个虚拟钱包系统？

王争 · 设计模式之美



上一节课，我们做了一些理论知识的铺垫性讲解，讲到了两种开发模式，基于贫血模型的传统开发模式，以及基于充血模型的 DDD 开发模式。今天，我们正式进入实战环节，看如何分别用这两种开发模式，设计实现一个钱包系统。

话不多说，让我们正式开始今天的学习吧！

钱包业务背景介绍

很多具有支付、购买功能的应用（比如淘宝、滴滴出行、极客时间等）都支持钱包的功能。应用为每个用户开设一个系统内的虚拟钱包账户，支持用户充值、提现、支付、冻结、透支、转账、查询账户余额、查询交易流水等操作。下图是一张典型的钱包功能界面，你可以直观地感受一下。

账 户	交易记录
可用余额 (元)	充值 +130
124.62 充值 提现	2019-08-31 05:18:36
交易记录	提现 -50
优惠券	2019-08-14 04:15:26
月账单	支付 -70
	2019-07-15 17:13:06

一般来讲，每个虚拟钱包账户都会对应用户的一个真实的支付账户，有可能是银行卡账户，也有可能是三方支付账户（比如支付宝、微信钱包）。为了方便后续的讲解，我们限定钱包暂时只支持充值、提现、支付、查询余额、查询交易流水这五个核心的功能，其他比如冻结、透支、转赠等不常用的功能，我们暂不考虑。为了让你理解这五个核心功能是如何工作的，接下来，我们来一块儿看下它们的业务实现流程。

1. 充值

用户通过三方支付渠道，把自己银行卡账户内的钱，充值到虚拟钱包账号中。这整个过程，我们可以分解为三个主要的操作流程：第一个操作是从用户的银行卡账户转账到应用的公共银行卡账户；第二个操作是将用户的充值金额加到虚拟钱包余额上；第三个操作是记录刚刚这笔交易流水。

① 用户银行卡： $\xrightarrow{150\text{元}}$ 应用的公共银行卡

② 用户虚拟钱包 +150元

③ 记录交易： 充值 +150元

2. 支付

用户用钱包内的余额，支付购买应用内的商品。实际上，支付的过程就是一个转账的过程，从用户的虚拟钱包账户划钱到商家的虚拟钱包账户上。除此之外，我们也需要记录这笔支付的交易流水信息。

① 用户虚拟钱包 $\xrightarrow{90\text{元}}$ 商家虚拟钱包

② 记录交易： - 90元

3. 提现

除了充值、支付之外，用户还可以将虚拟钱包中的余额，提现到自己的银行卡中。这个过程实际上就是扣减用户虚拟钱包中的余额，并且触发真正的银行转账操作，从应用的公共银行账户转钱到用户的银行账户。同样，我们也需要记录这笔提现的交易流水信息。

① 用户虚拟钱包 -100元

② 应用的公共银行卡 $\xrightarrow{100\text{元}}$ 用户银行卡

③ 记录交易： 提现 -100元

4. 查询余额

查询余额功能比较简单，我们看一下虚拟钱包中的余额数字即可。

5. 查询交易流水

查询交易流水也比较简单。我们只支持三种类型的交易流水：充值、支付、提现。在用户充值、支付、提现的时候，我们会记录相应的交易信息。在需要查询的时候，我们只需要将之前记录的交易流水，按照时间、类型等条件过滤之后，显示出来即可。

钱包系统的设计思路

根据刚刚讲的业务实现流程和数据流转图，我们可以把整个钱包系统的业务划分为两部分，其中一部分单纯跟应用内的虚拟钱包账户打交道，另一部分单纯跟银行账户打交道。我们基于这样一个业务划分，给系统解耦，将整个钱包系统拆分为两个子系统：虚拟钱包系统和三方支付系统。



为了能在有限的篇幅内，将今天的内容讲透彻，我们接下来只聚焦于虚拟钱包系统的设计与实现。对于三方支付系统以及整个钱包系统的设计与实现，我们不做讲解。你可以自己思考下。

现在我们来看下，如果要支持钱包的这五个核心功能，虚拟钱包系统需要对应实现哪些操作。我画了一张图，列出了这五个功能都会对应虚拟钱包的哪些操作。注意，交易流水的记录和查询，我暂时在图中打了个问号，那是因为这块比较特殊，我们待会再讲。

钱包	虚拟钱包
充值	+ 余额
提现	- 余额
支付	+ - 余额
查询余额	查询余额
查询交易流水	? ? ?

从图中我们可以看出，虚拟钱包系统要支持的操作非常简单，就是余额的加加减减。其中，充值、提现、查询余额三个功能，只涉及一个账户余额的加减操作，而支付功能涉及两个账户的余额加减操作：一个账户减余额，另一个账户加余额。

现在，我们再来看一下图中问号的那部分，也就是交易流水该如何记录和查询？我们先来看一下，交易流水都需要包含哪些信息。我觉得下面这几个信息是必须包含的。

交易流水ID	交易时间	交易金额	交易类型	入账钱包账号	出账钱包账号
--------	------	------	------	--------	--------

充值、提现、支付


从图中我们可以发现，交易流水的数据格式包含两个钱包账号，一个是入账钱包账号，一个是出账钱包账号。为什么要有两个账号信息呢？这主要是为了兼容支付这种涉及两个账户的交易类型。不过，对于充值、提现这两种交易类型来说，我们只需要记录一个钱包账户信息就够了。

整个虚拟钱包的设计思路到此讲完了。接下来，我们来看一下，如何分别用基于贫血模型的传统开发模式和基于充血模型的 DDD 开发模式，来实现这样一个虚拟钱包系统？

基于贫血模型的传统开发模式

实际上，如果你有一定 Web 项目的开发经验，并且听明白了我刚刚讲的设计思路，那对你来说，利用基于贫血模型的传统开发模式来实现这样一个系统，应该是一件挺简单的事情。不过，为了对比两种开发模式，我还是带你一块儿来实现一遍。

这是一个典型的 Web 后端项目的三层结构。其中，Controller 和 VO 负责暴露接口，具体的代码实现如下所示。注意，Controller 中，接口实现比较简单，主要就是调用 Service 的方法，所以，我省略了具体的代码实现。

 复制代码

```
1 public class VirtualWalletController {
2     // 通过构造函数或者IOC框架注入
3     private VirtualWalletService virtualWalletService;
4
5     public BigDecimal getBalance(Long walletId) { ... } //查询余额
6     public void debit(Long walletId, BigDecimal amount) { ... } //出账
7     public void credit(Long walletId, BigDecimal amount) { ... } //入账
8     public void transfer(Long fromWalletId, Long toWalletId, BigDecimal amount) { .
9     //省略查询transaction的接口
10 }
```

Service 和 BO 负责核心业务逻辑，Repository 和 Entity 负责数据存取。Repository 这一层的代码实现比较简单，不是我们讲解的重点，所以我也省略掉了。Service 层的代码如下所示。注意，这里我省略了一些不重要的校验代码，比如，对 amount 是否小于 0、钱包是否存在的校验等等。

```

1 public class VirtualWalletBo { //省略getter/setter/constructor方法
2     private Long id;
3     private Long createTime;
4     private BigDecimal balance;
5 }
6
7 public Enum TransactionType {
8     DEBIT,
9     CREDIT,
10    TRANSFER;
11 }
12
13 public class VirtualWalletService {
14     // 通过构造函数或者IOC框架注入
15     private VirtualWalletRepository walletRepo;
16     private VirtualWalletTransactionRepository transactionRepo;
17
18     public VirtualWalletBo getVirtualWallet(Long walletId) {
19         VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
20         VirtualWalletBo walletBo = convert(walletEntity);
21         return walletBo;
22     }
23
24     public BigDecimal getBalance(Long walletId) {
25         return walletRepo.getBalance(walletId);
26     }
27
28     @Transactional
29     public void debit(Long walletId, BigDecimal amount) {
30         VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
31         BigDecimal balance = walletEntity.getBalance();
32         if (balance.compareTo(amount) < 0) {
33             throw new NoSufficientBalanceException(...);
34         }
35         VirtualWalletTransactionEntity transactionEntity = new VirtualWalletTransacti
36         transactionEntity.setAmount(amount);
37         transactionEntity.setCreateTime(System.currentTimeMillis());
38         transactionEntity.setType(TransactionType.DEBIT);
39         transactionEntity.setFromWalletId(walletId);
40         transactionRepo.saveTransaction(transactionEntity);
41         walletRepo.updateBalance(walletId, balance.subtract(amount));
42     }
43
44     @Transactional
45     public void credit(Long walletId, BigDecimal amount) {
46         VirtualWalletTransactionEntity transactionEntity = new VirtualWalletTransacti
47         transactionEntity.setAmount(amount);

```



```

48     transactionEntity.setCreateTime(System.currentTimeMillis());
49     transactionEntity.setType(TransactionType.CREDIT);
50     transactionEntity.setFromWalletId(walletId);
51     transactionRepo.saveTransaction(transactionEntity);
52     VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
53     BigDecimal balance = walletEntity.getBalance();
54     walletRepo.updateBalance(walletId, balance.add(amount));
55 }
56
57 @Transactional
58 public void transfer(Long fromWalletId, Long toWalletId, BigDecimal amount) {
59     VirtualWalletTransactionEntity transactionEntity = new VirtualWalletTransacti
60     transactionEntity.setAmount(amount);
61     transactionEntity.setCreateTime(System.currentTimeMillis());
62     transactionEntity.setType(TransactionType.TRANSFER);
63     transactionEntity.setFromWalletId(fromWalletId);
64     transactionEntity.setToWalletId(toWalletId);
65     transactionRepo.saveTransaction(transactionEntity);
66     debit(fromWalletId, amount);
67     credit(toWalletId, amount);
68 }
69 }


```

基于充血模型的 DDD 开发模式

刚刚讲了如何利用基于贫血模型的传统开发模式来实现虚拟钱包系统，现在，我们再来看一下，如何利用基于充血模型的 DDD 开发模式来实现这个系统？

在上一节课中，我们讲到，基于充血模型的 DDD 开发模式，跟基于贫血模型的传统开发模式的主要区别就在 Service 层，Controller 层和 Repository 层的代码基本上相同。所以，我们重点看一下，Service 层按照基于充血模型的 DDD 开发模式该如何来实现。

在这种开发模式下，我们把虚拟钱包 VirtualWallet 类设计成一个充血的 Domain 领域模型，并且将原来在 Service 类中的部分业务逻辑移动到 VirtualWallet 类中，让 Service 类的实现依赖 VirtualWallet 类。具体的代码实现如下所示：

 复制代码

```

1 public class VirtualWallet { // Domain领域模型(充血模型)
2     private Long id;
3     private Long createTime = System.currentTimeMillis();

```


```
4     private BigDecimal balance = BigDecimal.ZERO;
5
6     public VirtualWallet(Long preAllocatedId) {
7         this.id = preAllocatedId;
8     }
9
10    public BigDecimal balance() {
11        return this.balance;
12    }
13
14    public void debit(BigDecimal amount) {
15        if (this.balance.compareTo(amount) < 0) {
16            throw new InsufficientBalanceException(...);
17        }
18        this.balance = this.balance.subtract(amount);
19    }
20
21    public void credit(BigDecimal amount) {
22        if (amount.compareTo(BigDecimal.ZERO) < 0) {
23            throw new InvalidAmountException(...);
24        }
25        this.balance = this.balance.add(amount);
26    }
27 }
28
29 public class VirtualWalletService {
30     // 通过构造函数或者IOC框架注入
31     private VirtualWalletRepository walletRepo;
32     private VirtualWalletTransactionRepository transactionRepo;
33
34     public VirtualWallet getVirtualWallet(Long walletId) {
35         VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
36         VirtualWallet wallet = convert(walletEntity);
37         return wallet;
38     }
39
40     public BigDecimal getBalance(Long walletId) {
41         return walletRepo.getBalance(walletId);
42     }
43
44     @Transactional
45     public void debit(Long walletId, BigDecimal amount) {
46         VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
47         VirtualWallet wallet = convert(walletEntity);
48         wallet.debit(amount);
49         VirtualWalletTransactionEntity transactionEntity = new VirtualWalletTransacti
50         transactionEntity.setAmount(amount);
51         transactionEntity.setCreateTime(System.currentTimeMillis());
52         transactionEntity.setType(TransactionType.DEBIT);
```

```

53     transactionEntity.setFromWalletId(walletId);
54     transactionRepo.saveTransaction(transactionEntity);
55     walletRepo.updateBalance(walletId, wallet.balance());
56 }
57
58 @Transactional
59 public void credit(Long walletId, BigDecimal amount) {
60     VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
61     VirtualWallet wallet = convert(walletEntity);
62     wallet.credit(amount);
63     VirtualWalletTransactionEntity transactionEntity = new VirtualWalletTransacti
64     transactionEntity.setAmount(amount);
65     transactionEntity.setCreateTime(System.currentTimeMillis());
66     transactionEntity.setType(TransactionType.CREDIT);
67     transactionEntity.setFromWalletId(walletId);
68     transactionRepo.saveTransaction(transactionEntity);
69     walletRepo.updateBalance(walletId, wallet.balance());
70 }
71
72 @Transactional
73 public void transfer(Long fromWalletId, Long toWalletId, BigDecimal amount) {
74     //...跟基于贫血模型的传统开发模式的代码一样...
75 }
76 }
77

```

看了上面的代码，你可能会说，领域模型 VirtualWallet 类很单薄，包含的业务逻辑很简单。相对于原来的贫血模型的设计思路，这种充血模型的设计思路，貌似并没有太大优势。你说得没错！这也是大部分业务系统都使用基于贫血模型开发的原因。不过，如果虚拟钱包系统需要支持更复杂的业务逻辑，那充血模型的优势就显现出来了。比如，我们要支持透支一定额度和冻结部分余额的功能。这个时候，我们重新来看一下 VirtualWallet 类的实现代码。

 复制代码

```

1 public class VirtualWallet {
2     private Long id;
3     private Long createTime = System.currentTimeMillis();
4     private BigDecimal balance = BigDecimal.ZERO;
5     private boolean isAllowedOverdraft = true;
6     private BigDecimal overdraftAmount = BigDecimal.ZERO;
7     private BigDecimal frozenAmount = BigDecimal.ZERO;
8
9     public VirtualWallet(Long preAllocatedId) {
10         this.id = preAllocatedId;
11     }

```

```

12
13     public void freeze(BigDecimal amount) { ... }
14     public void unfreeze(BigDecimal amount) { ...}
15     public void increaseOverdraftAmount(BigDecimal amount) { ... }
16     public void decreaseOverdraftAmount(BigDecimal amount) { ... }
17     public void closeOverdraft() { ... }
18     public void openOverdraft() { ... }
19
20     public BigDecimal balance() {
21         return this.balance;
22     }
23
24     public BigDecimal getAvaliableBalance() {
25         BigDecimal totalAvaliableBalance = this.balance.subtract(this.frozenAmount);
26         if (isAllowedOverdraft) {
27             totalAvaliableBalance += this.overdraftAmount;
28         }
29         return totalAvaliableBalance;
30     }
31
32     public void debit(BigDecimal amount) {
33         BigDecimal totalAvaliableBalance = getAvaliableBalance();
34         if (totoalAvaliableBalance.compareTo(amount) < 0) {
35             throw new InsufficientBalanceException(...);
36         }
37         this.balance = this.balance.subtract(amount);
38     }
39
40     public void credit(BigDecimal amount) {
41         if (amount.compareTo(BigDecimal.ZERO) < 0) {
42             throw new InvalidAmountException(...);
43         }
44         this.balance = this.balance.add(amount);
45     }
46 }
47

```

领域模型 VirtualWallet 类添加了简单的冻结和透支逻辑之后，功能看起来就丰富了很多，代码也没那么单薄了。如果功能继续演进，我们可以增加更加细化的冻结策略、透支策略、支持钱包账号（VirtualWallet id 字段）自动生成的逻辑（不是通过构造函数经外部传入 ID，而是通过分布式 ID 生成算法来自动生成 ID）等等。VirtualWallet 类的业务逻辑会变得越来越复杂，也就很值得设计成充血模型了。

辩证思考与灵活应用

对于虚拟钱包系统的设计与两种开发模式的代码实现，我想你应该有个比较清晰的了解了。不过，我觉得还有两个问题值得讨论一下。

第一个要讨论的问题是：在基于充血模型的 DDD 开发模式中，将业务逻辑移动到 Domain 中，Service 类变得很薄，但在我们的代码设计与实现中，并没有完全将 Service 类去掉，这是为什么？或者说，Service 类在这种情况下担当的职责是什么？哪些功能逻辑会放到 Service 类中？

区别于 Domain 的职责，Service 类主要有下面这样几个职责。

1.Service 类负责与 Repository 交流。在我的设计与代码实现中，VirtualWalletService 类负责与 Repository 层打交道，调用 Repository 类的方法，获取数据库中的数据，转化成领域模型 VirtualWallet，然后由领域模型 VirtualWallet 来完成业务逻辑，最后调用 Repository 类的方法，将数据存回数据库。

这里我再稍微解释一下，之所以让 VirtualWalletService 类与 Repository 打交道，而不是让领域模型 VirtualWallet 与 Repository 打交道，那是因为我们想保持领域模型的独立性，不与任何其他层的代码（Repository 层的代码）或开发框架（比如 Spring、MyBatis）耦合在一起，将流程性的代码逻辑（比如从 DB 中取数据、映射数据）与领域模型的业务逻辑解耦，让领域模型更加可复用。

2.Service 类负责跨领域模型的业务聚合功能。VirtualWalletService 类中的 transfer() 转账函数会涉及两个钱包的操作，因此这部分业务逻辑无法放到 VirtualWallet 类中，所以，我们暂且把转账业务放到 VirtualWalletService 类中了。当然，虽然功能演进，使得转账业务变得复杂起来之后，我们也可以将转账业务抽取出来，设计成一个独立的领域模型。

3.Service 类负责一些非功能性及与三方系统交互的工作。比如幂等、事务、发邮件、发消息、记录日志、调用其他系统的 RPC 接口等，都可以放到 Service 类中。

第二个要讨论问题是：在基于充血模型的 DDD 开发模式中，尽管 Service 层被改造成了充血模型，但是 Controller 层和 Repository 层还是贫血模型，是否有必要也进行充血领域建模

呢？

答案是没有必要。Controller 层主要负责接口的暴露，Repository 层主要负责与数据库打交道，这两层包含的业务逻辑并不多，前面我们也提到了，如果业务逻辑比较简单，就没必要做充血建模，即便设计成充血模型，类也非常单薄，看起来也很奇怪。

尽管这样的设计是一种面向过程的编程风格，但我们只要控制好面向过程编程风格的副作用，照样可以开发出优秀的软件。那这里的副作用怎么控制呢？

就拿 Repository 的 Entity 来说，即便它被设计成贫血模型，违反面向对象编程的封装特性，有被任意代码修改数据的风险，但 Entity 的生命周期是有限的。一般来讲，我们把它传递到 Service 层之后，就会转化成 BO 或者 Domain 来继续后面的业务逻辑。Entity 的生命周期到此就结束了，所以也并不会被到处任意修改。

我们再来说说 Controller 层的 VO。实际上 VO 是一种 DTO（Data Transfer Object，数据传输对象）。它主要是作为接口的数据传输载体，将数据发送给其他系统。从功能上来讲，它理应不包含业务逻辑、只包含数据。所以，我们将它设计成贫血模型也是比较合理的。

重点回顾

今天的内容到此就讲完了。我们一块来总结回顾一下，你应该重点掌握的知识点。

基于充血模型的 DDD 开发模式跟基于贫血模型的传统开发模式相比，主要区别在 Service 层。在基于充血模型的开发模式下，我们将部分原来在 Service 类中的业务逻辑移动到了一个充血的 Domain 领域模型中，让 Service 类的实现依赖这个 Domain 类。

在基于充血模型的 DDD 开发模式下，Service 类并不会完全移除，而是负责一些不适合放在 Domain 类中的功能。比如，负责与 Repository 层打交道、跨领域模型的业务聚合功能、幂等事务等非功能性的工作。

基于充血模型的 DDD 开发模式跟基于贫血模型的传统开发模式相比，Controller 层和 Repository 层的代码基本上相同。这是因为，Repository 层的 Entity 生命周期有限，Controller 层的 VO 只是单纯作为一种 DTO。两部分的业务逻辑都不会太复杂。业务逻辑主

要集中在 Service 层。所以，Repository 层和 Controller 层继续沿用贫血模型的设计思路是没有问题的。

课堂讨论

这两节课中对于 DDD 的讲解，都是我的个人主观看法，你可能会不同看法。

欢迎在留言区说一说你对 DDD 的看法。如果觉得有帮助，你也可以把这篇文章分享给你的朋友。

AI智能总结

本文深入介绍了基于充血模型的领域驱动设计（DDD）开发虚拟钱包系统的方法。通过对钱包业务的背景和核心功能的讨论，以及虚拟钱包系统的设计思路和实现细节的详细阐述，读者可以深入了解基于充血模型的DDD开发模式。与传统的基于贫血模型的开发模式相比，本文重点强调了在基于充血模型的开发模式下，Service层的变化和功能。在这种模式下，部分业务逻辑被移动到充血的Domain领域模型中，使得Service类的实现依赖于这个Domain类。同时，文章还讨论了Service层在处理与Repository层的交互、跨领域模型的业务聚合功能和幂等事务等方面的作用。此外，文章还强调了基于充血模型的DDD开发模式下，Controller层和Repository层的代码基本上相同的特点。总的来说，本文通过深入介绍虚拟钱包系统的业务背景、设计思路和两种开发模式的实现，为读者提供了对基于充血模型的DDD开发模式的深入理解和实践指导。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (304)

最新 精选



miracle

2019-11-29

建议将完整一些的代码放到 github 上 然后感兴趣的话可以自行去github 上研究或者提 pr

作者回复: 好的，我把完整代码抽空整理好放到github上

<https://github.com/wangzheng0822>

共 20 条评论 >

👍 75



邹佳敏

2019-12-02

看了一圈评论，好像没有人和我有同样的疑惑？

争哥说了很多交易流水表的设计，明明已经详细介绍了字段冗余的表1要明显优于表2，但为何在虚拟钱包的交易流水表的设计里，使用的又是字段紧凑的表2呢？

那么，在底层虚拟钱包的交易流水表里，同样会存在数据不一致的情况呀？A转出被记录下来了，B转入失败。

作者回复: 是有这个问题 我改下

共 18 条评论 >

👍 27



落叶飞逝的恋

2019-11-29

还有一点，期待老师实现一个完整的案例的代码以供我们参考琢磨。

作者回复: 完整案例代码可能就太多了

共 9 条评论 >

👍 11



斜杠青年

2019-12-18

有一个人问题不太懂 数据持久的话 没有set get方法 如何进行持久化？

作者回复: 如果你用orm框架持久化 必须有get set 那就要妥协

共 5 条评论 >

👍 8



Wiggins

2019-12-02

老师你好，看完自己实现的时候有个疑问，每次实例化VirtualWallet时候他的balance都会被初始化为0，我又不想把balance set的方法暴露出来，但是如果Domain不跟Repository层交互的话，就无法获取到当前其中的余额。请问下老师是否只能在构造函数中传入这一种办法？

作者回复: 可以放到构造函数中

共 2 条评论 >

👍 4



饭粒

2019-12-03

看完这篇对 DDD 也有了初步的认识了，区别了贫血模式的开发，DDD 应用 OOP 的设计实现提高了封装性，在业务对象类 VirtualWallet 中封装数据和基本的数据处理过程，service 使用业务对象类暴露的方法过程以完成完整的功能。实现上业务对象类具备的封装，单一职责等特性，这样在易用，易维护，易扩展，易读等方面较之贫血模型都会有提高。

另外有两个问题请教下老师：

- 1.贫血模型的 service 中有 VirtualWalletRepository，VirtualWalletTransactionRepository 两个 repository，看字面应该是区分是否带事务，不太明白这样写的好处或用意？因为我现在一般是直接在 service 上直接加事务。
- 2.钱包交易流水和虚拟钱包的交易流水的功能区分还不是特别清楚，示例代码也没有体现。事物一致性的日志记录不能直接用钱包交易流水线吗？

作者回复: 1、transaction能处理分布式事务

2、再看遍文章吧 都有讲到



3



晨间新闻

2019-12-03

看了下项目代码，service里的方法多数都是获取对象列表，对象入库，删除，很多方法都不是具体某个对象的某个动作，不像余额加减一样，是一个动作，对应某个属性的变化。感觉是不是用不上DDD啊。

作者回复: 简单的业务确实用不上ddd



3



好饿早知道送外卖了

2019-12-02

对于前端同学而言、DDD是不是类似于MVVM啊？只是没有数据绑定的业务映射

作者回复: 是的



2



JRich

2020-11-19

老师，上一节不是说业务模型是BO吗，怎么这里又叫domain呢，我们实际开发过程中数据库对象（entity）叫domain，这个该怎么区分呢？

作者回复: 至于叫什么不是关键，关键是理解它是用来做什么的



Dana

2020-11-13

老师里面的代码 不怎么看得懂 没学过 java

作者回复: 应该差不多吧，理解思路是重点

