

## 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？

2019-11-29 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 20:15 大小 18.56M



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、转赠、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） 124.62                      充值   提现	充值                                      +130 2019-08-31 05:18:36
交易记录	提现                                      -50 2019-08-14 04:15:26
优惠券	支付                                      -70 2019-07-15 17:13:06
月账单	

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，然后触发真正的银行转账操作，从应用的公共银行账户转钱到商家的银行账户（注意，这里并不是从用户的银行账户转钱到商家的银行账户）。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 应用的公共银行卡  $\xrightarrow{90\text{元}}$  商家银行卡

③ 记录交易： 支付-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
--------	------	------	------	--------	--------

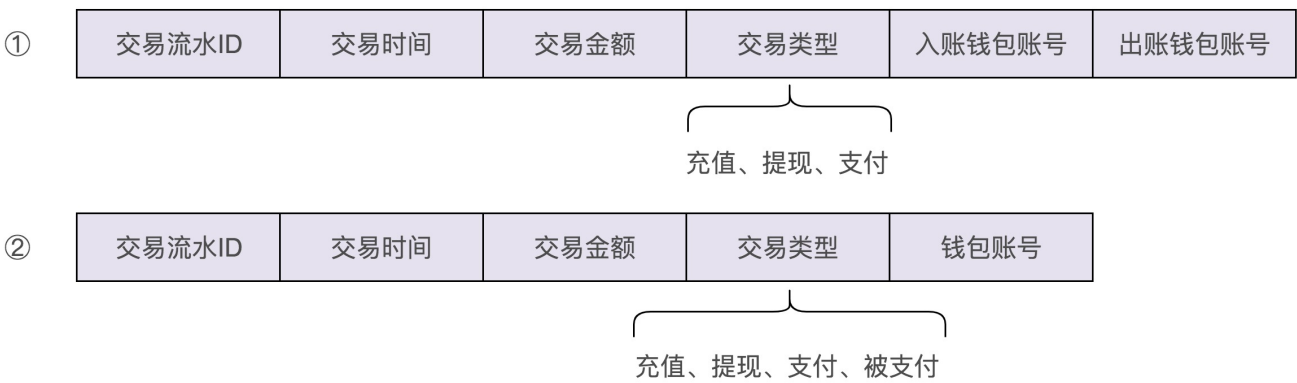
└──────────┘  
充值、提现、支付

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，所以，这样的交易流水数据格式的设计稍微有点浪费存储空间。

实际上，我们还有另外一种交易流水数据格式的设计思路，可以解决这个问题。我们把“支付”这个交易类型，拆为两个子类型：支付和被支付。支付单纯表示出账，余额扣减，被支付单纯表示入账，余额增加。这样我们在设计交易流水数据格式的时候，只需要记录一个账户信息即可。我画了一张两种交易流水数据格式的对比图，你可以对比着看一下。



**那以上两种交易流水数据格式的设计思路，你觉得哪一个更好呢？**

答案是第一种设计思路更好些。因为交易流水有两个功能：一个是业务功能，比如，提供用户查询交易流水信息；另一个是非业务功能，保证数据的一致性。这里主要是指支付操作数据的一致性。

支付实际上就是一个转账的操作，在一个账户上加上一定的金额，在另一个账户上减去相应的金额。我们需要保证加金额和减金额这两个操作，要么都成功，要么都失败。如果一个成功，一个失败，就会导致数据的不一致，一个账户明明减掉了钱，另一个账户却没有收到钱。

保证数据一致性的方法有很多，比如依赖数据库事务的原子性，将两个操作放在同一个事务中执行。但是，这样的做法不够灵活，因为我们的有可能做了分库分表，支付涉及的两个账户可能存储在不同的库中，无法直接利用数据库本身的事务特性，在一个事务中执行两个账户的操作。当然，我们还有一些支持分布式事务的开源框架，但是，为了保证数据的强一致性，它们的实现逻辑一般都比较复杂、本身的性能也不高，会影响业务的执行时间。所以，

更加权衡的一种做法就是，不保证数据的强一致性，只实现数据的最终一致性，也就是我们刚刚提到的交易流水要实现的非业务功能。

对于支付这样的类似转账的操作，我们在操作两个钱包账户余额之前，先记录交易流水，并且标记为“待执行”，当两个钱包的加减金额都完成之后，我们再回过头来，将交易流水标记为“成功”。在给两个钱包加减金额的过程中，如果有任意一个操作失败，我们就将交易记录的状态标记为“失败”。我们通过后台补漏 Job，拉取状态为“失败”或者长时间处于“待执行”状态的交易记录，重新执行或者人工介入处理。

如果选择第二种交易流水的设计思路，使用两条交易流水来记录支付操作，那记录两条交易流水本身又存在数据的一致性问题，有可能入账的交易流水记录成功，出账的交易流水信息记录失败。所以，权衡利弊，我们选择第一种稍微有些冗余的数据格式设计思路。

**现在，我们再思考这样一个问题：充值、提现、支付这些业务交易类型，是否应该让虚拟钱包系统感知？换句话说，我们是否应该在虚拟钱包系统的交易流水中记录这三种类型？**

答案是否定的。虚拟钱包系统不应该感知具体的业务交易类型。我们前面讲到，虚拟钱包支持的操作，仅仅是余额的加加减减操作，不涉及复杂业务概念，职责单一、功能通用。如果耦合太多业务概念到里面，势必影响系统的通用性，而且还会导致系统越做越复杂。因此，我们不希望将充值、支付、提现这样的业务概念添加到虚拟钱包系统中。

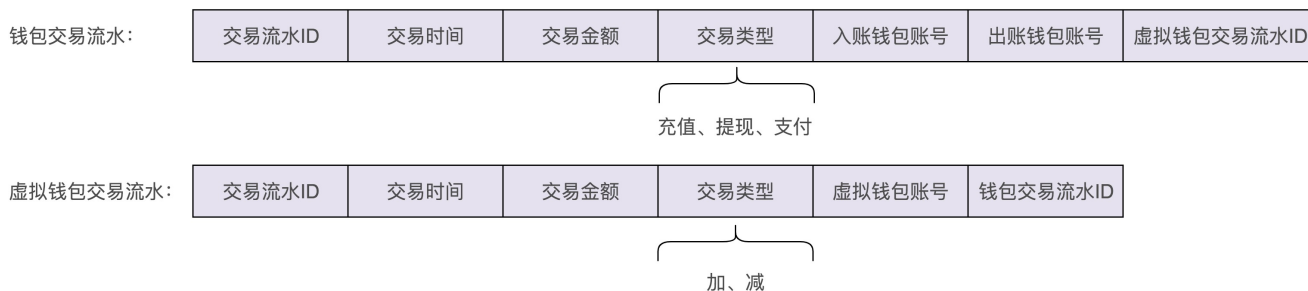
**但是，如果我们不在虚拟钱包系统的交易流水中记录交易类型，那在用户查询交易流水的时候，如何显示每条交易流水的交易类型呢？**

从系统设计的角度，我们不应该在虚拟钱包系统的交易流水中记录交易类型。从产品需求的角度来说，我们又必须记录交易流水的交易类型。听起来比较矛盾，这个问题该如何解决呢？

我们可以通过记录两条交易流水信息的方式来解决。我们前面讲到，整个钱包系统分为两个子系统，上层钱包系统的实现，依赖底层虚拟钱包系统和三方支付系统。对于钱包系统来说，它可以感知充值、支付、提现等业务概念，所以，我们在钱包系统这一层额外再记录一条包含交易类型的交易流水信息，而在底层的虚拟钱包系统中记录不包含交易类型的交易流水信息。



为了让你更好地理解刚刚的设计思路，我画了一张图，你可以对比着我的讲解一块儿来看。



我们通过查询上层钱包系统的交易流水信息，去满足用户查询交易流水的功能需求，而虚拟钱包中的交易流水就只是用来解决数据一致性问题。实际上，它的作用还有很多，比如用来对账等。限于篇幅，这里我们就不展开讲了。

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 }
```

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。

 复制代码

```
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 class VirtualWalletService {
8     // 通过构造函数或者 IOC 框架注入
9     private VirtualWalletRepository walletRepo;
10    private VirtualWalletTransactionRepository transactionRepo;
11
12    public VirtualWalletBo getVirtualWallet(Long walletId) {
13        VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
14        VirtualWalletBo walletBo = convert(walletEntity);
15        return walletBo;
16    }
17
18    public BigDecimal getBalance(Long walletId) {
19        return virtualWalletRepo.getBalance(walletId);
20    }
21
22    public void debit(Long walletId, BigDecimal amount) {
23        VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
24        BigDecimal balance = walletEntity.getBalance();
25        if (balance.compareTo(amount) < 0) {
26            throw new NoSufficientBalanceException(...);
27        }
28        walletRepo.updateBalance(walletId, balance.subtract(amount));
29    }
30
31    public void credit(Long walletId, BigDecimal amount) {
32        VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
33        BigDecimal balance = walletEntity.getBalance();
34        walletRepo.updateBalance(walletId, balance.add(amount));
35    }
36
37    public void transfer(Long fromWalletId, Long toWalletId, BigDecimal amount) {
38        VirtualWalletTransactionEntity transactionEntity = new VirtualWalletTransa
39        transactionEntity.setAmount(amount);
40        transactionEntity.setCreateTime(System.currentTimeMillis());
41        transactionEntity.setFromWalletId(fromWalletId);
42        transactionEntity.setToWalletId(toWalletId);
43        transactionEntity.setStatus(Status.TO_BE_EXECUTED);
```

```

44     Long transactionId = transactionRepo.saveTransaction(transactionEntity);
45     try {
46         debit(fromWalletId, amount);
47         credit(toWalletId, amount);
48     } catch (InsufficientBalanceException e) {
49         transactionRepo.updateStatus(transactionId, Status.CLOSED);
50         ...rethrow exception e...
51     } catch (Exception e) {
52         transactionRepo.updateStatus(transactionId, Status.FAILED);
53         ...rethrow exception e...
54     }
55     transactionRepo.updateStatus(transactionId, Status.EXECUTED);
56 }
57 }

```

以上便是利用基于贫血模型的传统开发模式来实现的虚拟钱包系统。尽管我们对代码稍微做了简化，但整体的业务逻辑就是上面这样子。其中大部分代码逻辑都非常简单，最复杂的是 Service 中的 transfer() 转账函数。我们为了保证转账操作的数据一致性，添加了一些跟 transaction 相关的记录和状态更新的代码，理解起来稍微有点难度，你可以对照着之前讲的设计思路，自己多思考一下。

## 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：

```

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) {

```

 复制代码

```
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.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.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 virtualWalletRepo.getBalance(walletId);
42     }
43
44     public void debit(Long walletId, BigDecimal amount) {
45         VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
46         VirtualWallet wallet = convert(walletEntity);
47         wallet.debit(amount);
48         walletRepo.updateBalance(walletId, wallet.balance());
49     }
50
51     public void credit(Long walletId, BigDecimal amount) {
52         VirtualWalletEntity walletEntity = walletRepo.getWalletEntity(walletId);
53         VirtualWallet wallet = convert(walletEntity);
54         wallet.credit(amount);
55         walletRepo.updateBalance(walletId, wallet.balance());
56     }
57
58     public void transfer(Long fromWalletId, Long toWalletId, BigDecimal amount)
```

```
59 //... 跟基于贫血模型的传统开发模式的代码一样...
60 }
61 }
```

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 (totalAvaliableBalance.compareTo(amount) < 0) {
```

```
35         throw new InsufficientBalanceException(...);
36     }
37     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.add(amount);
45 }
46 }
```

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，违反面相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。

点击参加小程序学习打卡 

# 8个月，攻克设计模式



扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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。

上一篇 11 | 实战一（上）：业务开发常用的基于贫血模型的MVC架构违背OOP吗？

下一篇 13 | 实战二（上）：如何对接鉴权这样一个功能开发做面向对象分析？

## 精选留言 (86)

 写留言



**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>



 4

 23



**potato00fa**

2019-11-29

我对DDD的看法就是，它可以把原来最重的service逻辑拆分并且转移一部分逻辑，可以使代码可读性略微提高，另一个比较重要的点是使得模型充血以后，基于模型的业务抽象在不断的迭代之后会越来越明确，业务的细节会越来越精准，通过阅读模型的充血行为代

码，能够极快的了解系统的业务，对于开发来说能说明显的提升开发效率。  
在维护性上来说，如果项目新进了开发人员，如果是贫血模型的service代码，无论代码...  
展开 ▾

💬 2

👍 13



淡忘

2019-11-29

这两天一直在思考ddd，就等课程更新，这样一说就理解了，domain模型使用充血模型设计，使之具备独立性，而业务无关的vo，po就可以使用贫血模型进行设计，因为不涉及具体复杂业务，如果control层需要调用多个领域模型，则把相关的领域服务组合在一起，这里有个小问题，就是do转为dto这个过程，应该是在应用层完成还是领域层完成，如果在应用层完成，好像属于把领域模型暴露出去了，希望老师可以在指点一下

展开 ▾

💬 5

👍 13



辣么大

2019-11-29

理解OOP，我们就不难理解DDD：

DDD第一原则：将数据和操作结合。（贫血模型将数据和操作分离，违反OOP的原则。）

DDD第二原则：界限上下文。这是将“单一指责”应用于我们的领域模型。

DDD is nothing more than OOP applied to business models. DDD其实就是把OOP...

展开 ▾

💬 9

👍 10



Cy23

2019-11-29

听完一遍，看来我需要在听一遍，php视乎要理解JAVA的有点差异啊

💬 10

👍 6



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

◀ ▶

💬 1

👍 4



墨雨

2019-11-29

看了老师的这篇文章让我对 entity, bo, vo 有了一个更清晰的认识。我是这样理解的, entity 是对数据库的映射, vo 是前端展示的映射, bo 在 DDD 充血模型中我看到了他的用处, 看起来他是将 entity 的一些逻辑业务分离了出来做了一个解耦 (在我看来貌似没有 bo 或者说 Domain 类似 加余额减余额的逻辑也可以写在 entity 中, 只是这样做对于专注于数据库的 entity 来说逻辑更复杂了, 维护起来会很困难), 同时也解决了 entity 暴露...  
展开 ∨

4

4



风之射手座

2019-11-30

第2步支付处理流程感觉有点问题:

从用户的虚拟钱包转90到商家虚拟钱包应该就完了, 不应该再从应用公共银行卡再划钱到商家银行卡。如果要即时划转到商家银行卡, 就要记得把商家的虚拟钱包减少90。  
好像是这样吧?

展开 ∨

1

3



join

2019-11-30

看到这里, 感觉才真正理解充血模型的作用:

真正的业务逻辑都放在充血的领域对象中, 与具体使用什么框架 (比如Spring, MyBatis), 具体使用什么数据库无关。这样有利于保护领域对象中的数据, 比如钱包中的余额, 当有入账和出账操作时, 余额在领域对象中自动执行加减操作, 而不是将余额暴露在Ser...

展开 ∨

...

2



睡觉

2019-11-29

在我看来, Repository与Domain都是service的底层。Repository复杂数据的存储, Domain负责业务逻辑, service将两者融合。

展开 ∨

...

2



落叶飞逝的恋

2019-11-29

DDD 中VirtualWalletService convert哪里定义了。

展开 ∨

...

2



老姜

2019-11-29

更新流水出现异常会导钱包操作成功了，但是就是状态是错误的？是不是应该把生成流水放到一个事务里面，更新钱包和更新流水状态放到另外一个事务里面会避免这个问题？



2



Angus

2019-11-29

我理解的ddd分为四层，用户接口层，应用层，领域层，基础设施层。领域服务还是跟基础设施层打交道，领域服务主要是提供这个领域的业务行为，通过应用层聚合领域服务，而应用层正是和领域专家建立统一语言的一层，

展开



2



Monday

2019-12-01

老师，最后一张设计图祭出了两条流水线，是分别对应数据库的两张表吗？



1



曾泽伟

2019-11-29

在平时开发的时候，我更喜欢将service分成多层（可以理解为2层），一层是基于数据层的简单封装操作dataService，另一层，是专门用来组合调用dataService方法的，这样既有封装又有能让开发人员更好的理解，不知道我这种方式是否可以？有什么需要改进的地方吗？

另外时间长了，项目可能就面目全非了，不知道这个情况，该怎么办？...

展开



1



sprinty

2019-11-29

感谢老师的分享，收获很多，也产生了两个问题：

问题1：Entity 转换成 Domain 的代码应该在哪一层实现？感觉在 Service 层不大合适，因为可能多个 Service 会使用到。

问题2：如果涉及到表单的保存，入参是一个保存全量数据的对象(比如，创建一个新用户...

展开



1



grey927





2019-12-02

如果是简单的增删改查附加一点点判断逻辑的操作，是不是不太适合用充血模型



**DullBird**

2019-12-02

1. 看完文章理解了之前的一个疑惑点。之前一直疑惑Domain模型，一定要自己去和数据库打交道，并且只能访问自己管理的数据。

学习到了service可以控制操作db。这样让Domain模型更加干净，简洁。

2. 之前做OOP设计的时候，会被业务必须要，而模型不应该有的场景所困扰，最后做出的决定都是模型加上好了，再外面调用方再弄一遍不就重复了么。学习了今天的知识，得...

展开 ∨



**真飞鸟**

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？

展开 ∨



**Eden Ma**

2019-12-02

iOS端是否可以这样理解：MVVM中VM层相当于Domain，就V层中控制器的业务逻辑抽到了VM中，M层负责数据和数据库操作，因为M层数据是对外暴露的依然是贫血。V层中的View相当于文中的Controller对外展示数据，只做展示功能，所以可以继续贫血，VM和V双向绑定。因为没有BO这种，所以更多的是对控制器数据和业务逻辑的抽离到VM。而MVP也MVVM也蛮相似。所以我可以理解MVVM和MVP算是运用到了充血模型吗？

展开 ∨

