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？

在前面几节课中，我们学习了面向对象的一些理论知识，比如，面向对象四大特性、接口和抽象类、面向对象和面向过程编程风格、基于接口而非实现编程和多用组合少用继承设计思想等等。接下来，我们再用四节课的时间，通过两个更加贴近实战的项目来进一步学习，如何将这些理论应用到实际的软件开发中。

据我了解，大部分工程师都是做业务开发的，所以，今天我们讲的这个实战项目也是一个典型的业务系统开发案例。我们都知道，很多业务系统都是基于MVC三层架构来开发的。实际上，更确切点讲，这是一种基于贫血模型的MVC三层架构开发模式。

虽然这种开发模式已经成为标准的Web项目的开发模式，但它却违反了面向对象编程风格，是一种彻彻底底的面向过程的编程风格，因此而被有些人称为反模式（anti-pattern）。特别是**领域驱动设计**（Domain Driven Design，简称DDD）盛行之后，这种基于贫血模型的传统的开发模式就更加被人诟病。而基于充血模型的DDD开发模式越来越被人提倡。所以，我打算用两节课的时间，结合一个虚拟钱包系统的开发案例，带你彻底弄清楚这两种开发模式。

考虑到你有可能不太了解我刚刚提到的这几个概念，所以，在正式进入实战项目的讲解之前，我先带你搞清楚下面几个问题：

- 什么是贫血模型？什么是充血模型？
- 为什么说基于贫血模型的传统开发模式违反OOP？
- 基于贫血模型的传统开发模式既然违反OOP，那又为什么如此流行？
- 什么情况下我们应该考虑使用基于充血模型的DDD开发模式？

好了，让我们带着这些问题，正式开始今天的学习吧！

什么是基于贫血模型的传统开发模式？

我相信，对于大部分的后端开发工程师来说，MVC三层架构都不会陌生。不过，为了统一我们之间对MVC的认识，我还是带你一块来回顾一下，什么是MVC三层架构。

MVC三层架构中的M表示Model，V表示View，C表示Controller。它将整个项目分为三层：展示层、逻辑层、数据层。MVC三层开发架构是一个比较笼统的分层方式，落实到具体的开发层面，很多项目也并不会100%遵从MVC固定的分层方式，而是会根据具体的项目需求，做适当的调整。

比如，现在很多Web或者App项目都是前后端分离的，后端负责暴露接口给前端调用。这种情况下，我们一般就将后端项目分为Repository层、Service层、Controller层。其中，Repository层负责数据访问，Service层负责业务逻辑，Controller层负责暴露接口。当然，这只是其中一种分层和命名方式。不同的项目、不同的团队，可能会对此有所调整。不过，万变不离其宗，只要是依赖数据库开发的Web项目，基本的分层思路都大差不差。

刚刚我们回顾了MVC三层开发架构。现在，我们再来看一下，什么是贫血模型？

实际上，你可能一直都在用贫血模型做开发，只是自己不知道而已。不夸张地讲，据我了解，目前几乎所有的业务后端系统，都是基于贫血模型的。我举一个简单的例子来给你解释一下。

```
////////// Controller+VO(View Object) //////////
public class UserController {
  private UserService userService; //通过构造函数或者IOC框架注入

  public UserVo getUserById(Long userId) {
    UserBo userBo = userService.getUser(userId);
    UserVo userVo = [...convert userBo to userVo...];
    return userVo;
  }
}

public class UserVo {//省略其他属性、get/set/construct方法
  private Long id;
  private String name;
  private String cellphone;
}

////////// Service+BO(Business Object) //////////
public class UserService {
  private UserRepository userRepository; //通过构造函数或者IOC框架注入

  public UserBo getUserById(Long userId) {
    UserEntity userEntity = userRepository.getUserById(userId);
    UserBo userBo = [...convert userEntity to userBo...];
    return userBo;
  }
}

public class UserBo {//省略其他属性、get/set/construct方法
  private Long id;
  private String name;
  private String cellphone;
}

////////// Repository+Entity //////////
public class UserRepository {
  public UserEntity getUserById(Long userId) { //... }
}

public class UserEntity {//省略其他属性、get/set/construct方法
  private Long id;
  private String name;
  private String cellphone;
}
```

我们平时开发Web后端项目的时候，基本上都是这么组织代码的。其中，UserEntity和UserRepository组成了数据访问层，UserBo和UserService组成了业务逻辑层，UserVo和UserController在这里属于接口层。

从代码中，我们可以发现，UserBo是一个纯粹的数据结构，只包含数据，不包含任何业务逻辑。业务逻辑集中在UserService中。我们通过UserService来操作UserBo。换句话说，Service层的数据和业务逻辑，被分割为BO和Service两个类中。像UserBo这样，只包含数据，不包含业务逻辑的类，就叫作**贫血模型**（Anemic Domain Model）。同理，UserEntity、UserVo都是基于贫血模型设计的。这种贫血模型将数据与操作分离，破坏了面向对象的封装特性，是一种典型的面向过程的编程风格。

什么是基于充血模型的DDD开发模式？

刚刚我们讲了基于贫血模型的传统的开发模式。现在我们再讲一下，另外一种最近更加被推崇的开发模式：

基于充血模型的DDD开发模式。

首先，我们先来看一下，什么是充血模型？

在贫血模型中，数据和业务逻辑被分割到不同的类中。**充血模型**（Rich Domain Model）正好相反，数据和对应的业务逻辑被封装到同一个类中。因此，这种充血模型满足面向对象的封装特性，是典型的面向对象编程风格。

接下来，我们再来看一下，什么是领域驱动设计？

领域驱动设计，即DDD，主要是用来指导如何解耦业务系统，划分业务模块，定义业务领域模型及其交互。领域驱动设计这个概念并不新颖，早在2004年就被提出了，到现在已经有十几年的历史了。不过，它被大众熟知，还是基于另一个概念的兴起，那就是微服务。

我们知道，除了监控、调用链追踪、API网关等服务治理系统的开发之外，微服务还有另外一个更加重要的工作，那就是针对公司的业务，合理地做微服务拆分。而领域驱动设计恰好就是用来指导划分服务的。所以，微服务加速了领域驱动设计的盛行。

不过，我个人觉得，领域驱动设计有点儿类似敏捷开发、SOA、PAAS等概念，听起来很高大上，但实际上只值“五分钱”。即便你没有听说过领域驱动设计，对这个概念一无所知，只要你是在开发业务系统，也或多或少都在使用它。做好领域驱动设计的关键是，看你对自己所做业务的熟悉程度，而并不是对领域驱动设计这个概念本身的掌握程度。即便你对领域驱动搞得再清楚，但是对业务不熟悉，也并不一定能做出合理的领域设计。所以，不要把领域驱动设计当银弹，不要花太多的时间去过度地研究它。

实际上，基于充血模型的DDD开发模式实现的代码，也是按照MVC三层架构分层的。Controller层还是负责暴露接口，Repository层还是负责数据存取，Service层负责核心业务逻辑。它跟基于贫血模型的传统开发模式的区别主要在Service层。

在基于贫血模型的传统开发模式中，Service层包含Service类和BO类两部分，BO是贫血模型，只包含数据，不包含具体的业务逻辑。业务逻辑集中在Service类中。在基于充血模型的DDD开发模式中，Service层包含Service类和Domain类两部分。Domain就相当于贫血模型中的BO。不过，Domain与BO的区别在于它是基于充血模型开发的，既包含数据，也包含业务逻辑。而Service类变得非常单薄。总结一下的话就是，基于贫血模型的传统的开发模式，重Service轻BO；基于充血模型的DDD开发模式，轻Service重Domain。

基于充血模型的DDD设计模式的概念，今天我们只是简单地介绍了一下。在下一节课中，我会结合具体的项目，通过代码来给你展示，如何基于这种开发模式来开发一个系统。

为什么基于贫血模型的传统开发模式如此受欢迎？

前面我们讲过，基于贫血模型的传统开发模式，将数据与业务逻辑分离，违反了OOP的封装特性，实际上是一种面向过程的编程风格。但是，现在几乎所有的Web项目，都是基于这种贫血模型的开发模式，甚至连Java Spring框架的官方demo，都是按照这种开发模式来编写的。

我们前面也讲过，面向过程编程风格有种种弊端，比如，数据和操作分离之后，数据本身的操作就不受限制了。任何代码都可以随意修改数据。既然基于贫血模型的这种传统开发模式是面向过程编程风格的，那它又为什么会被广大程序员所接受呢？关于这个问题，我总结了下面三点原因。

第一点原因是，大部分情况下，我们开发的系统业务可能都比较简单，简单到就是基于SQL的CRUD操作，所以，我们根本不需要动脑子精心设计充血模型，贫血模型就足以应付这种简单业务的开发工作。除此之外，因为业务比较简单，即便我们使用充血模型，那模型本身包含的业务逻辑也并不会很多，设计出来的领域模型也会比较单薄，跟贫血模型差不多，没有太大意义。

第二点原因是，充血模型的设计要比贫血模型更加有难度。因为充血模型是一种面向对象的编程风格。我们从一开始就要设计好针对数据要暴露哪些操作，定义哪些业务逻辑。而不是像贫血模型那样，我们只需要定义数据，之后有什么功能开发需求，我们就在Service层定义什么操作，不需要事先做太多设计。

第三点原因是，思维已固化，转型有成本。基于贫血模型的传统开发模式经历了这么多年，已经深得人心、习以为常。你随便问一个旁边的大龄同事，基本上他过往参与的所有Web项目应该都是基于这个开发模式的，而且也没有出过啥大问题。如果转向用充血模型、领域驱动设计，那势必有一定的学习成本、转型成本。很多人在没有遇到开发痛点的情况下，是不愿意做这件事情的。

什么项目应该考虑使用基于充血模型的DDD开发模式？

既然基于贫血模型的开发模式已经成为了一种约定俗成的开发习惯，那什么样的项目应该考虑使用基于充血模型的DDD开发模式呢？

刚刚我们讲到，基于贫血模型的传统的开发模式，比较适合业务比较简单的系统开发。相对应的，基于充血模型的DDD开发模式，更适合业务复杂的系统开发。比如，包含各种利息计算模型、还款模型等复杂业务的金融系统。

你可能会有一些疑问，这两种开发模式，落实到代码层面，区别不就是一个将业务逻辑放到Service类中，一个将业务逻辑放到Domain领域模型中吗？为什么基于贫血模型的传统开发模式，就不能应对复杂业务系统的开发？而基于充血模型的DDD开发模式就可以呢？

实际上，除了我们能看到的代码层面的区别之外（一个业务逻辑放到Service层，一个放到领域模型中），还有一个非常重要的区别，那就是两种不同的开发模式会导致不同的开发流程。基于充血模型的DDD开发模式的开发流程，在应对复杂业务系统的开发的时候更加有优势。为什么这么说呢？我们先来回忆一下，我们平时基于贫血模型的传统的开发模式，都是怎么实现一个功能需求的。

不夸张地讲，我们平时的开发，大部分都是SQL驱动（SQL-Driven）的开发模式。我们接到一个后端接口的开发需求的时候，就去看接口需要的数据对应到数据库中，需要哪张表或者哪几张表，然后思考如何编写SQL语句来获取数据。之后就是定义Entity、BO、VO，然后模板式地往对应的Repository、Service、Controller类中添加代码。

业务逻辑包裹在一个大的SQL语句中，而Service层可以做的事情很少。SQL都是针对特定的业务功能编写的，复用性差。当我要开发另一个业务功能的时候，只能重新写个满足新需求的SQL语句，这就可能导致各种长得差不多、区别很小的SQL语句满天飞。

所以，在这个过程中，很少有人会应用领域模型、OOP的概念，也很少有代码复用意识。对于简单业务系统来说，这种开发方式问题不大。但对于复杂业务系统的开发来说，这样的开发方式会让代码越来越混乱，最终导致无法维护。

如果我们在项目中，应用基于充血模型的DDD的开发模式，那对应的开发流程就完全不一样了。在这种开发模式下，我们需要事先理清楚所有的业务，定义领域模型所包含的属性和方法。领域模型相当于可复用的业

务中间层。新功能需求的开发，都基于之前定义好的这些领域模型来完成。

我们知道，越复杂的系统，对代码的复用性、易维护性要求就越高，我们就越应该花更多的时间和精力在前期设计上。而基于充血模型的DDD开发模式，正好需要我们前期做大量的业务调研、领域模型设计，所以它更加适合这种复杂系统的开发。

重点回顾

今天的内容到此就讲完了，我们来一起回顾一下，你应该掌握的重点内容。

我们平时做Web项目的业务开发，大部分都是基于贫血模型的MVC三层架构，在专栏中我把它称为传统的开发模式。之所以称之为“传统”，是相对于新兴的基于充血模型的DDD开发模式来说的。基于贫血模型的传统开发模式，是典型的面向过程的编程风格。相反，基于充血模型的DDD开发模式，是典型的面向对象的编程风格。

不过，DDD也并非银弹。对于业务不复杂的系统开发来说，基于贫血模型的传统开发模式简单够用，基于充血模型的DDD开发模式有点大材小用，无法发挥作用。相反，对于业务复杂的系统开发来说，基于充血模型的DDD开发模式，因为前期需要在设计上投入更多时间和精力，来提高代码的复用性和可维护性，所以相比基于贫血模型的开发模式，更加有优势。

课堂讨论

今天课堂讨论的话题有两个。

1. 你做经历的项目中，有哪些是基于贫血模型的传统的开发模式？有哪些是基于充血模型的DDD开发模式呢？请简单对比一下两者的优劣。
2. 对于我们举的例子中，UserEntity、UserBo、UserVo包含的字段都差不多，是否可以合并为一个类呢？

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- 业余爱好者 2019-11-27 07:59:53
  一直贫血而不自知 [46赞]

- 乐 2019-11-27 12:04:30
  ## 为什么贫血模型盛行

  下面几项自己都中过招（环境问题和个人问题）：

  ### 环境问题 ##

  * 近朱者赤，近墨者黑
  * 大多数人都是模仿别人的代码，而别人的代码基本上都是 demo，没有复杂的业务逻辑，基本是贫血模型
  * 找不到好的指导与学习对象
  * 接触不到复杂业务项目
  * 做 web 项目的，很大一部分就是简单的 CURD，贫血模型就能解决
  * 公司以任务数来衡量个人价值

个人问题

* 不考虑项目质量属性
* 只关心当前业务，没有意识去思考后期该如何维护和响应业务变更
* 求快不求质
* 个人以任务数来自我满足
* 没有 60 分和 100 分的概念
* 需求分析、设计、编码合为一体

如何理解充血模型

先推荐一本书：整洁架构设计

先说一下充血模型中各组件的角色：

* controller 主要服务于非业务功能，比如说数据验证
* service 服务于 use case，负责的是业务流程与对应规则
* Domain 服务于核心业务逻辑和核心业务数据
* rep 用于与外部交互数据

额外说一点，业务开发个人倾向于六边形架构，而非传统的三层架构。六边形架构更能体现当下 web 应用的场景

六边形项目结构（根据实际情况自行组织与定义）：

* InboundHandler 代替 controller
* *WebController：处理 web 接口
* *WeChatController：处理微信公众号接口
* *AppController：处理 app 接口
* *MqListener：处理 消息
* *RpcController：处理子系统间的调用
* service 服务于 use case，负责的是业务流程与对应规则
* CQPS + SRP：读写分离和单一原则将 use case 分散到不同的 service 中，避免一个巨大的 service 类（碰到过 8000 行的 service）
* Domain 服务于核心业务逻辑和核心业务数据
* 最高层组件，不会依赖底层组件
* 易测试
* outBoundhandle 代替 rep
* MqProducer：发布消息
* Cache：从缓存获取数据
* sql：从数据库获取数据
* Rpc：从子系统获取数据

各层之间的数据模型不要共用，主要是因为稳定性不同，各层数据模型的变更原因和变更速率是不同的，离 IO 设备越近的的稳定性越差，比如说 controller 层的 VO，rep 层的 entity。Domain 层是核心业务逻

辑和核心业务数据，稳定性是最高的

几个不太容易理解的点（我刚开始碰到的时候很费解）：

* use case 和 核心业务逻辑该如何定义与区分
* 哪些该放到 service 里面，哪些该放到 Domain 中
* rep 是依赖于 service 的，而不是 service 依赖 rep 层
* 业务逻辑是最高层组件（最稳定的），rep 层是底层组件
* 接口能反转依赖关系

一剂良药：所有的中间层都是为了解耦 [35赞]

- 墨雨 2019-11-27 08:37:42
  老师，我平常做web开发都是，entity,dao,service,controller.对vo,bo不是很理解，也没有用到。有没有demo呢？ [13赞]

  作者回复2019-11-27 15:22:47
  我抽空写个demo，放到我的github上吧
  https://github.com/wangzheng0822

- ㄟ(ε)ㄏ 2019-11-27 01:17:46
  个人感觉业务被贫血模型绑架的另一个原因是以前缓存nosql这些技术不不成熟 刚毕业那会哪有什么redis，机器的内存也不多。都是公司堆在角落的旧机器。一些业务如果在domain里实现可能会hold住数据库中的大部分数据。所以业务上都需要翻译成sql的where和join来减少网络和内存的开销。功能都被sql抢了去，想充血也充不起来。现在随便开个项目不带个redis老板都会质疑一下。mysql的访问也都是能少就少，不行再多加几台云服务器。老板也显得更有面儿。 [11赞]

- lizi 2019-11-27 00:55:10
  沙发，不睡觉，听课。哈哈，加班好累， [9赞]

- 有铭 2019-11-27 10:13:28
  我个人认为，充血模型在web开发领域不流行的一个根本原因，在于互联网兴起后各种层出不穷的需求变动，以及短命的项目生存周期，充血模型应对复杂业务确实很有优势，但是这是建立在复杂业务本身其实相对稳定上，比如银行的业务，虽然复杂，但是其实很稳定。但是要是换在互联网，今天改需求明天改需求，甚至很多时候根本就是推倒了重来的需求，充血模型面对这种状态，根本是力不从心的 [8赞]

- 李小四 2019-11-27 08:56:53
  设计模式_10
  # 问题:
  - 1. 做的Android项目更多，Android开发也是经历了MVC==>MVP(依然是一种MVC架构)==>MVVM的模式演进。类MVC模式比较多，在UI相关的开发中，只用过贫血模式(之前也尝试过使用充血模式，但考虑到不一致带来的成本就放弃了)；在UI无关的复杂服务类开发中，也用过充血模型(虽然我不知道它叫充血模型)。我认为贫血模型的优点是更容易看懂，充血模型的优点是更能应对复杂业务。
  - 2. 我认为还是不要放在同一个类中，原因是：成本大于收益。成本：一个复杂的类，在被不同的模块调用时充当着不同的角色，甚至，不同的模块调用不同的字段，需要大篇幅的文档来描述这些差异。稍有修改，复杂度的增加非线性。优点：代码重用。

感想:
软件开发处理的是工程学问题，解决方案依赖场景，一个新技术的火爆一定是解决了当前主流场景的痛点问题，随着规模和复杂度的变化，场景也随之变化；争论贫血模式更好还是充血模式更好，争论哪个开发语言更好，这样的问题都是伪命题，我们更应该投入精力的是为当前场景选择最合适的解决方案。 [8赞]

- grey927 2019-11-27 16:55:35
能否用代码表达一下充血模型，其实还是不太理解 [6赞]

作者回复2019-11-27 17:44:32
下一节课有的

- 小晏子 2019-11-27 08:48:12
基本上经历过的web项目都是基于贫血模型开发模式的，entity，bo，vo不能放在一个类里，每个对象的应用场景不同，entity是映射数据库字段的，bo，vo适合业务和展示相关的，而且entity相对来讲变化不多，bo，vo可能会频繁变化，所以不适合放在同一个类里 [6赞]

- 花儿少年 2019-11-27 11:21:09
有本书叫ddd 原则，模式与实践可以翻阅一下
首先要明白一点ddd适用范围，多数业务就是CRUD就可以搞定，理解起来也没有困难，为啥不继续用贫血模型。
充血模型就像老师说的需要精心设计，以应对变化，如果没有一个复杂的业务场景就根本用不到，或者说用起来很难受是一个似是而非的东西。
况且充血模型只建议用在核心域，还有通用域和支撑域呢，不要一上来就ddd [4赞]

- Lrwin 2019-11-27 10:56:08
我觉得代码架构和业务架构一样，只要将关注点分离就可以。
简单的系统，困难的不在于领域的拆分，而在于时间成本的控制。从软件工程角度考虑，时间，成本，范围三角理论可以进行分析、。
我们所说的复杂系统，更看重业务的复杂度，将复杂度降低的方法则是分而治之。这样可以降低复杂度。
复杂要解决三个问题：规模问题，结构问题和需求变化问题。无论是技术复杂度或业务复杂度，只要能解决这三种复杂度问题就是好的方法。
DDD模型其实无异，都是将Model层做重。因为业务核心是技术无关的。传统MVC用于C/S模型，也依然是重Model层的。我觉得软件设计的方法没有变化，只是大家看到的视角不同罢了。
软件架构有两个含义：1.参与的元素有哪些 2.元素间的关系是什么。 从抽象角度来看，非常简单。 [3赞]

- 梦倚栏杆 2019-11-27 06:21:26
第一个还没有太多的感受，还需要时间来练习感受
第二个是否合成一个各有优劣，可能还是和写代码人的功底有关：
拆分开的优势：各层的防腐隔离，当前层的变化不影响其他层。
拆分开的劣势：来一个迭代需求，比如需要加一下邮箱等， rd很有可能在三个类里各加一个字段，从上改到下，完全看不出隔离的优势，就看不到了一层层的类转换
反过来就是不拆分的优劣
[3赞]

- _呱太_ 2019-11-28 19:07:45
小争哥，对设计模式这些概念都有了解，一直想找一些源码来看看顺便练下手，请问下有没有推荐的 C++ 比较经典的开源代码呢 [2赞]

- 丶Zero灬 2019-11-28 12:54:43

我们公司的ERP系统使用的是贫血模型，支付系统由一个OOP的忠实践行者设计的，使用的是充血模型。两个系统我都有在维护，先说ERP。遇到新需求的时候，就像老师说的SQL驱动。从后往前返回数据。简单的需求还好，像一些复杂的模块。看着service一个方法动辄几百，甚至出现过上千行的。真心感觉改不动。在业务还比较简单的时候贫血模型还够用，但随着业务发展，service层越来越重。这时还不做封装抽象，系统真心不好维护。很多时候都要依靠老员工的讲解。而，支付系统，在开发新功能的时候，因为封装抽象已经做的很好，改动起来还是蛮愉快的。没有那么多的重复性代码。不过，因为封装的太多太深，在刚接手的时候确实不好读。业务逻辑逻辑分散在各处。当然，也可能是本人水平有限。 [2赞]

- Lonely绿豆蛙 2019-11-28 09:26:36
  最近边看边重构自己的项目，感觉真的是从码农视角转到了架构师层次~ [2赞]

- 深度 · 仁 2019-11-28 00:23:23
  一拍大腿，靠，说到心坎里去了，各种细碎的sql，就为了解决某个小功能！ 业务熟悉，领域驱动设计就是屠龙刀，业务不熟悉，DDD也就值个半毛钱！茶不思，饭不想，期待后面的文章更新 [2赞]

- Geek_Zjy 2019-11-27 10:41:18
  看到「领域驱动设计有点儿类似敏捷开发、SOA、PAAS 等概念，听起来很高大上，但实际上只值“五分钱”。」时，不知道引起了多少人的共鸣，O(∩_∩)O~。 做技术的本身就经常会遇到沟通问题，一些人还总喜欢“造概念”，唯恐别人听懂了，争哥这句话无疑说中了我们的心坎儿。
  当然我这里也不是说 DDD 不好（看后面的争哥也没这个意思），但是每个理论都有自己的局限性和适用性，看很多文章在讲一些理论时，总是恨不得把自己的理论（其实也算不得自己的）吹成银弹，态度上就让人很难接受。
  我还是喜欢争哥的风格，逻辑很清晰，也很严谨，很务实。

  关于老师的问题。
  说句实话，我们就没有写过充血模型的代码。
  我们会把 UserEntity、UserBo 混着用， UserBo 和 UserVo 之间转换时有时还会用 BeanUtils 之类的工具 copy 。
  对于复杂的逻辑，我们就用复杂 SQL 或者 Service 中的代码解决。

  不过我在翻一些框架时，比如 Java 的并发包时不可避免的需要梳理 Lock、Condition、Synchronizer 之间的关系。比如看 Spring IOC 时，也会需要梳理围绕着 Context 、 Factory 展开的很多类之间的关系。
  就好像你要“混某个圈子”时，就不可避免的“拜码头”，认识一堆“七大姑八大姨”，然后你才能理解整个“圈子”里的关系和运转逻辑。
  我也经常会有疑问， DDD 和面向对象究竟是什么关系，也会猜想：是不是面向对象主要关注“圈子”内的问题，而 DDD 主要关注“圈子”之间的问题？有没有高手可以回答一下。
  （其实我最近一直都想订隔壁DDD的课，但是考虑到精力的问题，以及担心学不会，主要不是争哥讲O(∩_∩)O~，所以没下手） [2赞]

  作者回复2019-11-27 15:05:39
  哈哈，多谢认可，我写这篇文字的时候，还害怕搞DDD的人会来骂我，看来是我多虑了。隔壁的DDD课程可以去学下，管它是不是我写的，看看他咋“吹”的也好。

- 默 2019-11-27 08:36:25
  我经历的所有的项目都是贫血模型，对于充血模型没尝试过，如果说业务数据与业务操作分离，那么在设计接口时如何将繁多的业务属性当成参数传入给接口方法呢？是说将所有属性的getter和setter直接生成在接口中吗？那么对于接口中使用这些零散的属性是怎样使用的呢？觉得二者确实还得从实际情况出发，看业务复杂度及对代码设计是否有要求，但往往都是为了习惯、速度而忽略了设计初心。

  对于三类对象是否合并成一个，我上次的留言就正好说了这个问题，确实很难判别，起初是分开三个对象

，但是为了所谓的分层、隔离干净避免耦合，都没有体现，在业务调整方面分开3个对象并没体现优势，可能是我参与的项目都比较简单，基本上不涉及到需要解耦来进行代码的扩展性调整，反而是增减改属性比较频繁，这样就涉及到一位同学说的，一下修改3个对象而且最好属性值的复制，如果是用委托就不说了，如果不是委托，那么属性值复制多数依赖BeanUtils类通过类反射完成，还要保证属性类型的一致性；如果用委托，那么意味着3个对象的组合难免又是对分层上的一个污点。所以我觉得两者适用在不同的设计要求、团队人员的一往习惯以及项目的目标，如果公司涉及的项目没有那么复杂、设计要求不高，我更倾向于合并为一个。

[2赞]

- (´田ω田`) 2019-11-27 00:51:18
  第二题，不合并有好处。UserEntity是具体到数据库的，假如有不同版本的UserEntity，相互之间有少部分不同，抽象出一个BO层在Service层做UserEntity到BO的转化，可以方便Bo在Service层传播，方便不同项目之间复用、统一处理；再往上抽象出的VO，一般通过接口返回给前端展示，也可以是多种有差异的BO转化为同一种VO方便前端统一处理 [2赞]

- 杨树敏 2019-11-28 21:15:53
  回想起来经历的互联网项目一直都是贫血模型, 究其原因:
  1.互联网项目多采用敏捷开发, 需求多变, 设计充血模型的性价比低;
  2.互联网小产品业务场景相对简单, 业务难点更多的出现在并发, 性能上, 在存储,内存成本急剧下降的当下, 粗暴的横向扩展成为见效更快的选择. [1赞]