# **领域驱动**设计（一）什么是**领域驱动**设计

2019-05-14 18:34:51 [我想说的csdn](https://me.csdn.net/wang2963973852" \t "http://cache.baiducontent.com/_blank) 阅读数 323  收藏

分类专栏： [项目开发](https://blog.csdn.net/wang2963973852/category_6333348.html" \t "http://cache.baiducontent.com/_blank)[网络框架](https://blog.csdn.net/wang2963973852/category_7517687.html" \t "http://cache.baiducontent.com/_blank)

版权声明：本文为博主原创文章，遵循[CC 4.0 BY-SA](http://creativecommons.org/licenses/by-sa/4.0/" \t "http://cache.baiducontent.com/_blank)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/wang2963973852/article/details/90212560>

##### **为何需要领域驱动设计？**

****微服务和DDD的关系：****  
从国内来看，2014 年微服务的兴起是 DDD 的一个重要里程碑。不可否认，很多人是因为微服务才了解 DDD 的。在听说了微服务架构之后，人们觉得采用微服务架构会让系统开发与运维管理变得简单高效，同时实现的系统会更加合理，更加高可用、高性能，但是当他们实际去做微服务架构的时候，有不少人会发现自己做得并不好，没法取得人们“吹捧”的那些效果，“就算用了微服务架构也不能解决他们的问题，反而带来很多开发与运维上的负担”。于是他们去咨询、去找方法，最后发现其实是自己划分微服务的方法出错了，这个时候才知道人们在谈论微服务的时候，其实都没有讲到一个点：****应该用 DDD 的思想去指导微服务的实践****

因此我们需要去了解DDD

##### **什么是领域驱动设计**

DDD(Domain-Driven Design **领域驱动**设计)  
**领域驱动**设计其实分为两部分，一方面是**领域驱动**领域模型设计，一方面是领域模型驱动代码实现

##### **领域模型是什么：**

领域模型其实就是一种公共语言，一种开发人员和相关领域专家之间交流沟通的公共语言，使用这种语言可以将团队沟通和软件实现紧密结合到一起

****统一语言****概念在 DDD 中极为重要，在一个系统的构建过程中，往往业务人员关注的是业务架构，而技术人员则关注系统架构的表述方式；在将业务架构映射到系统架构的时候，需要经过一层“翻译”工作；而业务只要发生变化，就会影响到系统，系统就要重新对业务进行翻译，这会使工作变得复杂、低效

在 DDD 中，使用一个统一语言，可以直接将业务架构与系统架构绑定，不需要进一步去翻译，从而增强系统对业务的响应速度

##### **DDD 的生命周期：**

总结起来，DDD 的一个生命周期是这样的：在设计和实现一个系统的时候，这个系统所要处理问题的领域专家和开发人员以一套统一语言进行协作，共同完成该领域模型的构建

在这个过程中，业务架构和系统架构等问题都得到了解决，之后将领域模型中关于系统架构的主体映射为实现代码，完成系统的实现落地。而用什么方式去做领域模型的构建，方法是多样的，Alberto 自己就为此发明了 Event Storming（事件风暴），并成为了一种经典的 DDD 落地模式。

DDD 并不是直接给你建议某一个系统架构，它的执行结果是呈现一个方案，可以从这个构建出的模型中决定你去用什么技术来实现什么样的架构，进而来完成一个系统的设计。

##### **什么是领域（Domain）？**

任何一个系统都会属于某个特定的领域  
比如论坛是一个领域，只要你想做一个论坛，那这个论坛的核心业务是确定的，都有用户发帖、回帖等核心基本功能

比如电商平台、普通电商系统，这种都属于网上电商领域，只要是这个领域的系统，那都有商品浏览、购物车、下单、减库存、付款交易等核心环节

所以，同一个领域的系统都具有相同的核心业务，因为他们要解决的问题的本质是类似的

一个领域本质上可以理解为就是一个问题域，只要是同一个领域，那问题域就相同。所以，只要我们确定了系统所属的领域，那这个系统的核心业务，即要解决的关键问题、问题的范围边界就基本确定了  
通常我们说，要成为一个领域的专家，必须要在这个领域深入研究很多年才行。因为只有你研究了很多年，你才会遇到非常多的该领域的问题，同时你解决这个领域中的问题的经验也非常丰富。很多时候，领域专家比技术专家更加吃香，比如金融领域的专家

**领域驱动**设计”中的“领域”不仅包括系统架构的相关问题，还涉及到系统所支持的业务等内容，但它是与具体的开发技术**无关**的，也就是说 DDD 关注的是要系统中所要解决的问题的业务、流程和数据等内容是如何工作的，在这些东西理清之后，DDD 去构建出一个模型，接着再去选择具体的实现技术。DDD 强调的是解耦具体实现技术，所以它可以迅速梳理核心业务逻辑。

##### **什么是设计（Design）？**

设计指的是领域模型的设计，为什么是领域模型的设计而不是架构设计或其他的什么设计呢？因为DDD是一种基于模型驱动开发的软件开发思想，强调领域模型是整个系统的核心  
领域模型绑定了领域和代码实现，确保了最终的代码实现就一定是解决了领域中的核心问题的

##### **什么是驱动（Driven）？**

我们传统的思路是数据库驱动开发  
但是**领域驱动**设计是：  
1）**领域驱动**领域模型设计；  
2）领域模型驱动代码实现。

这个就和传统的开发形成鲜明对比了，我们总是以领域为边界，分析领域中的核心问题（核心关注点），然后设计对应的领域模型，再通过领域模型驱动代码实现。而像数据库设计、持久化技术等这些都不是DDD的核心，而是外围的东西

**领域驱动**设计（DDD）告诉我们的最大价值是：当我们要开发一个系统时，应该尽量先把领域模型想清楚，然后再开始动手编码，这样的系统后期才会很好维护。但是，很多项目（尤其是互联网项目，为了赶工）都是一开始模型没想清楚，一上来就开始建表写代码，代码写的非常冗余，完全是过程式的思考方式，最后导致系统非常难以维护

根据上面的连个步骤，我们只要保证领域模型的设计是正确的，就能确定领域模型可以解决领域中的核心问题；同理，我们只要保证代码实现是严格按照领域模型的意图来落地的，那就能保证最后出来的代码能够解决领域的核心问题的

##### **概念总结：**

* 领域就是问题域，有边界，领域中有很多问题； 任何一个系统要解决的那个大问题都对应一个领域；
* 通过建立领域模型来解决领域中的核心问题，模型驱动的思想；
* 领域建模的目标针对我们在领域中所关心的问题，即只针对核心关注点，而不是整个领域中的所有问题；
* 领域模型在设计时应考虑一定的抽象性、通用性，以及复用价值； 通过领域模型驱动代码的实现，确保代码让领域模型落地，代码最终能解决问题；
* 领域模型是系统的核心，是领域内的业务的直接沉淀，具有非常大的业务价值； 技术架构设计或数据存储等是在领域模型的外围，帮助领域模型进行落地；

##### **理解领域知识是基础**

做领域设计之前必须要对领域内的各种业务场景和各种业务规则非常了解，知道到底哪些是核心业务关注点这样才能有能力表达出系统该做成什么样子。所以，要知道一个系统到底该做成什么样子是非常重要的

假设你现在打算做一个电商平台，但是你对这个领域没什么了解，那你一定得先去了解下该领域内主流的电商平台，比如淘宝、天猫、京东、亚马逊等。这个了解的过程就是你沉淀领域知识的过程。如果你不了解，就算你领域建模的能力再强，各种技术架构能力再强也是使不上力

##### **拆分领域**

上面我们明白了，领域建模的基础是要先理解领域，让自己成为领域专家。但是，有时一个领域往往太复杂，涉及到的领域概念、业务规则、交互流程太多，导致我们没办法直接针对这个大的领域进行领域建模。所以，我们需要将领域进行拆分，本质上就是把大问题拆分为小问题，然后各个击破  
既然要把一个大的领域划分为多个小的领域（子域），那最关键的就是要理清每个子域的边界；然后要搞清楚哪些子域是核心子域，哪些是非核心子域，哪些是公共支撑子域；然后，还要思考子域之间的联系是什么

****那么，我们该如何划分子域呢？****  
我的个人看法是从业务相关性的角度去思考，也就是按业务功能为出发点进行划分  
例如：电商系统，通常一个电商系统都会包含好几个大块，比如：

* 会员中心：负责用户账号登录、用户信息的管理；
* 商品中心：负责商品的展示、导航、维护；
* 订单中心：负责订单的生成和生命周期管理；
* 交易中心：负责交易相关的业务；
* 库存中心：负责维护商品的库存；
* 促销中心：负责各种促销活动的支持；

****关键词：根据业务拆分子域****

上面这些中心看起来很自然，因为大家对电子商务的这个领域都已经非常熟悉了，所以都没什么疑问。但是如果我们遇到一个冷门的领域，就没办法这么容易的去划分子域了。这就需要我们先去努力理解领域内的知识

领域的划分也不是一成不变的，比如，也许我们一开始将商品和商品的库存都放在商品中心里，但是后来由于库存的维护越来越复杂，导致揉在一起对我们的系统维护带来一定的困难时，我们就会考虑将两者进行拆分，这个就是所谓的业务垂直分割。

****细化子域****  
通过上面的两步，我们了解了领域里的知识，也对领域进行了子域划分。但这样还不够，凭这些我们还无法进行后续的领域模型设计  
我们还必须再进一步细化每个子域，进一步明确每个子域的核心关注点，即需求细化，我觉得我们需要细化的方面有以下几点：

1）领域概念：梳理出我们关注的概念、概念的关系，并统一交流词汇，形成统一语言；  
2）业务规则：梳理出我们关注的各种业务规则，DDD中叫不变性（invariants），比如唯一性规则，余额不能小于零等；  
3）业务场景：梳理出核心业务场景，比如电商平台中的加入购物车、提交订单、发起付款等核心业务场景；  
4）业务流程：梳理出关键业务流程，比如订单处理流程，退款流程等；

上面我们了解了DDD那么我们来总结一下DDD的好处：

##### **DDD的好处**

****1）简化数据存储****  
在基于数据驱动的开发方式，也就是传统的多层开发架构中，大家定义了一堆DAL来操作数据,一般都需要把关系型数据转换为对象  
这种做法会导致一系列的性能问题，以及数据冗余问题，并且其实大多数技术人员的SQL能力不强也导致SQL的维护等问题

****2）更多的了解上下文****  
便于技术人员和业务人员进行充分的沟通  
我们很多软件的问题，大家都知道是需求的问题，也就是客户的需求我们很难理解准确，引入**领域驱动**设计之后，我们可以让业务人员更多的参与系统，更早的参与系统。

****3）统一语言****  
例如：Jack爱Rose  
在传统的数据驱动里面，因为是面向数据的，所以我们一般是使用一个service来处理两个对象之间的关系，因此我们一般会这么写：

UserService.Love(Jack, Rose)

但是我们业务人员很奇怪谁Love谁？ 为什么要UserService?

如果我们写成下面这样

Jack.Love(Rose)

或者我们要表达公司雇佣员工，使用如下模式

Company.hire(employee)

来代替

companyservice.hire(company,employee)

这样我们就更容易让业务人员参与进来，而且代码可以更易于表示真实的业务场景

****总的来说：****  
DDD是将传统的****UI********+Service+DataBase**** 的架构模式，改变成了****UI+DomainService +DomainEntity****的形式，两者看起来没有什么区别，实际上编程的世界观完全不同

* UI**+**Service**+**DataBase模式从接触到需求开始，我们考虑的是该如何设计数据库表，然后根据业务流程设计需要哪些行为，然后数据用数据库实现，行为使用服务实现
* UI**+**DomainService**+**DomainEntity，接触到需求第一步就是考虑领域模型，DDD让我们首先考虑的是业务语言，而不是数据

DDD强调业务抽象和面向对象编程，而不是过程式业务逻辑实现。重点不同导致编程世界观不同