领域驱动设计 DDD 之概览

作者：valarchie
原文链接：https://ld246.com/article/1572972342481
来源网站：链滴

在从事开发多年之后，你是否会感觉自己只是一个业务CRUD Boy，并认为业务没有多少技术含量。是否会陷入业务的泥潭中，各种复杂交错的业务规则使得代码开始腐烂开始失控，项目开始变得难以护，迭代举步维艰。如果你开始意识到这个问题的话，那么我十分推荐你开始学习领域驱动设计面向域建模的设计方式。

DDD是什么呢?

DDD即Domain Driven Design，翻译成中文的话就是领域驱动设计，首先我们应该先理解这里的领是什么意思？假设公司内部正在开发一套电商平台，而电商平台中包含了库存、订单、商品等核心业。这些核心业务逻辑其实呈现的就是电商平台领域。通俗的理解就是一整套体系的业务知识即代表了个领域。好比在线教育平台，它需要有一套体系的业务，包括招生、线上教学、课程等内容。我们将些业务抽象出领域模型，而这些领域模型表达了产品经理所阐述的业务需求，我们反复地用这些领域型与产品经理进行讨论沟通最终确定初步的领域模型。再使用初步的领域模型指导代码设计开发。

简而言之就是：

业务知识 --> 领域模型 --> 项目设计与代码开发

不同的电商平台的核心业务逻辑大都是相似的，这部分领域知识是可以进行复用，区别在于不同公司用的不同的编程语言，不同的前端控制框架，数据库框架。采用领域驱动设计的好处在于项目以领域型为核心，而Spring MVC、Struts等前端控制框架或者Hibernate、Mybatis对象数据库框架属于外技术基础，领域模型其实并不与这些基础技术产生耦合，所以在领域模型不变的情况下，我们是很容对我们的基础设施进行更换的。

那我们之前的开发也是有这些业务逻辑支撑？那我们之前传统的开发模式为什么不能称之为面向领域动设计呢？回想一下我们之前的代码开发，比如一个实现一个购物车下订单的功能，我们根据订单表字段，依次用商品字段、金额字段、用户字段等拼凑出订单表中的一条记录，然后写入到订单表当中。其实我们是面向数据库在进行开发，以数据库为重心，而业务逻辑零散地分布在各个Service当中，采用的是面向过程的编程方式而不是面向对象的方式，也就没有形成一套有机的业务逻辑。

而面向领域驱动设计则不一样，它以领域为重心，以刚才的购物车下订单功能为例子。在DDD当中，将购物车相关的业务逻辑封装到一个ShoppingCart对象中，并直接调用shoppingCart.takeOrder()订单的方法，代码的重心从生成订单表中的记录转移到购物车对象本身，而具体数据库中如何生成这记录并不属于我们的核心业务逻辑，它被下放到基础设施层，由Repository或者Dao等数据交互对象责去持久化我们对领域模型下达的指令所产生的数据库变化。

项目中的代码通过不同领域模型之间的配合体现了领域专家的业务意图，使得系统内部形成一套自运的有机整体。在以往面向过程式的开发方式，我们很难让领域专家或者产品经理直接看懂我们的业务辑，而DDD的优势在于shoppingCart.takeOrder()这种类似白话的方式直接体现出业务含义。而我代码中的领域模型和领域专家口中的领域模型是一致的。甚至我们可以在没有基础设施技术支持的情下，直接建模领域模型开始编写代码并测试验证业务逻辑。

领域模型总结

- 抽象了领域内的核心概念，并建立概念之间的关系；
- 领域模型维护了领域内的数据之间一致性的，也即我们的业务规则；

为什么我们需要DDD

1. 通过统一语言使得我们开发人员与领域专家（产品经理）能够更好的沟通。更准确的表达我们的业，而这些业务代码按照正如领域专家所想的那样工作。

2. 通过战术设计将业务知识进行集中，浓缩在领域模型当中。

3. 以往的业务代码中，我们总需要嵌入许多注释，因为过程式的代码自解释的能力很差，而最好的设就是代码本身。通过代码本身将领域中的知识呈现出来。

4. 通过战略设计解构复杂的业务系统，并使其简单化。

战术设计和战略设计是DDD针对局部和整体的设计指导。

贫血症和失忆症

传统的开发模式中，我们经常使用的是一个JavaBean，其中只有映射到数据库的字段，并没有业务为。通过填充这个JavaBean，并在对象外部进行业务逻辑的编写，如计算订单的最终金额填充到Javaean中再交由数据库映射框架进行持久化。

而其实这就是Evans所说的贫血症，因为数据和业务行为隔离开来，形成一种无机的代码组成。其实并非面向对象的编码方式，当我们看到Order对象时，我们根本不知道它含有计算最终金额的业务逻，而这其实就是一种所谓的贫血症引起的失忆症。数据和行为并没有紧密的联系到一起。

```
public static void buyProduct(Long orderId, Double price, Long productId, Float discount) {

        Order order = new Order();

        order.setOrderId(orderId);
        order.setProductId(productId);
        order.setDiscount(discount);
        order.setPrice(price);

        // 计算最终付款金额
        if(discount == 0) {
            throw new RuntimeException("折扣不可为0！");
        }

        Double paid = price * discount;
        order.setPaid(paid);

        OrderDao.save(order);

}
```

而更好的方式我们应该通过将数据和业务行为整合到一个对象中去，让二者形成一种有机的代码组成在GRASP对象职责中有一个原则就是当一个对象拥有某个方法所需的属性时，那么更应该将这个方法置到这个对象中去，而不是放在其他地方。

```
 public static void buyProduct_(Long orderId, Double price, Long productId, Float discount) {

        Order order = new Order(orderId, price, productId, discount);

        order.calculatePaid();

        OrderDao.save(order);

}
```

上面的例子更加符合我们对于业务的描述，但仅仅强调将业务逻辑封装到数据对象中去还不够，我们

需要通过这些对象之间的协作来进行业务的表达。一个很显而易见的例子就是关于转账的例子。

```
  public static void main(String[] args) {

        Account a = new Account(5);

        Account b = new Account(5);

        double transferMoney = 4;

        if (a.getMoney() < transferMoney) {
            throw new RuntimeException("余额不足！");
        }

        a.setMoney(a.getMoney() - transferMoney);

        b.setMoney(b.getMoney() + transferMoney);

    }
```

更好的做法我们应该借鉴面向对象的建模方式，将领域知识封装到账户Account模型中去。

```
public void transfer(Account another, double transferMoney) {

        if (money < transferMoney) {
            throw new RuntimeException("余额不足！");
        }

        money = money - transferMoney;
        another.setMoney(another.getMoney() + transferMoney);

    }
```

A账户向B账户进行转账。

```
  public static void main(String[] args) {

        Account a = new Account(5);
        Account b = new Account(5);

        double transferMoney = 4;

        a.transfer(b, transferMoney);

    }
```

我们通过领域模型之间的协作，呈现出来的代码就像白话一样。有主语谓语宾语。主语是A账户，谓是transfer()方法，宾语是B账户。这样一来，代码的自解释能力也就非常强了。就算产品经理不懂编语言的话，看到我们的代码的话也能理解其中的业务目的。

DDD的适用场景

那什么场景才是适合DDD的场景呢?

在可预见的未来中，项目的业务复杂度会越来越高，那就非常适合使用DDD的设计方式。而如果项目部都是一些非常简单的增删查改而很少包含业务知识的话，那真是想D也D不起来，因为DDD的思想是为了通过模型来表达领域知识，而领域知识本身就很匮乏的话，表达也就无从谈起。

如何DDD

我们可以通过和领域专家（产品经理）使用一致的通用语言，利用通用语言抽象出领域模型，在根据些领域模型进行代码的落地开发，这样一来便能更好的在代码中去体现业务领域知识。我们需要转变们以往的思维惯性，少从技术层面考虑，而更应该从业务层面去考虑。我的理解是，DDD是一套基于域为核心的面向对象编程的方法论。它主要通过两种设计来实现DDD。一种是战术设计，你可以理解在单个微服务中的设计。一种是战略设计，你可以理解为多个微服务之间如何进行协作。

战术设计

战术设计侧重点在于局部的设计，主要有以下几个概念：

- 实体：有唯一标识有生命周期，可以理解为通过实体可以对应到数据库的记录。
- 值对象：用来描述实体的属性。
- 聚合：包含实体和值对象，并维护了事务一致性。
- 资源库：用于获取或保存聚合。
- 领域服务：放置一些不适合在聚合中的业务逻辑。

我们简单的讲解一下这几个概念是如何在单个限界上下文（即单个微服务）中进行工作的。聚合由实、值对象进行组成，它维护了事务的一致性。而聚合又由资源库进行持久化以及查找或者获取。而当些业务规则并不能很好的放入实体或者值对象上时，我们可以使用领域服务。

战略建模

战略设计侧重点在于整体内的不同局部如何协作的设计，主要有以下几个概念：

- 限界上下文：一类领域模型运作的环境。比如电商中的商品模块即一个限界上下文。
- 上下文映射图：不同类的领域模型如何交互。比如在电商平台中商品模块是如何与库存模块进行交的。

限界上下文是一种概念上的边界，领域模型便工作于其中，也即一个限界上下文对应了我们设计的一微服务。而不同上下文如何进行沟通的话，则利用上下文映射图的概念来进行指导开发。

关于DDD的讨论非常之多，每个人的见解都不一样，这也是DDD为什么难以流行起来的原因之一。是DDD的思想还是非常值得借鉴的。关于DDD的学习个人非常推荐一定要阅读原著DDD以及IDDD。

以下是笔者上传的关于DDD和IDDD的高清PDF书籍，希望可以帮到想学习DDD的友们。

《领域驱动设计：软件核心复杂性应对之道》

提取码：6290

《实现领域驱动设计》

提取码：xl3t

《领域驱动设计精粹》

提取码：sgr2

关于DDD的理解各有不同，欢迎网友评论一起探讨。

原文链接：领域驱动设计 DDD 之概览