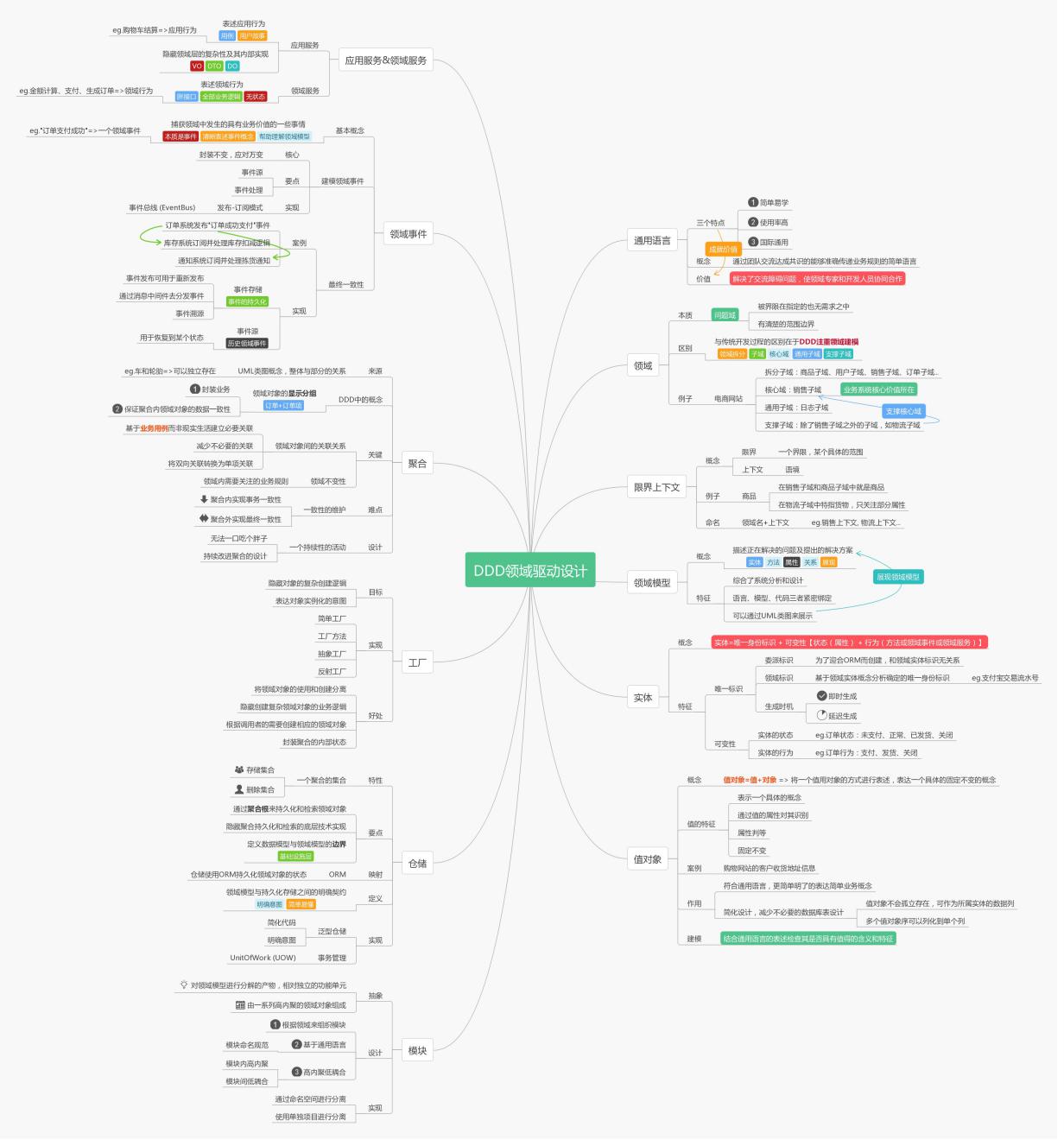
# **DDD 模式从天书到实践**



另存为此图后再看方便

### **背景**

正所谓有人的地方就有江湖，有设计的地方也一定会有架构。如果你是一位软件行业的老鸟，你一定会有这样的经历：一个业务的初期，普通的 CRUD 就能满足，业务线也很短，此时系统的一切都看起来很 nice，但随着迭代的不断演化，以及业务逻辑越来越复杂，我们的系统也越来越冗杂，模块彼此关联，甚至没有人能描述清楚每个细节。当新需求需要修改一个功能时，往往光回顾该功能涉及的流程就需要很长时间，更别提修改带来的不可预知的影响面。于是 RD 就加开关，小心翼翼地切流量上线，一有问题赶紧关闭开关。

面对此般场景，你要么跑路，要么重构。重构是克服演进式设计中大杂烩问题的主力，通过在单独的类及方法级别上做一系列小步重构来完成，我们可以很容易重构出一个独立的类来放某些通用的逻辑，但是，你会发现你很难给它一个业务上的含义，只能给予一个技术维度描绘的含义。你正在一边重构一边给后人挖坑。

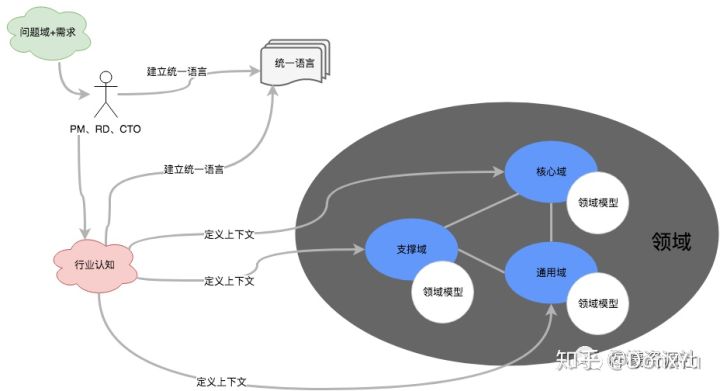
在互联网开发“小步快跑，迭代试错”的大环境下，DDD 似乎是一种比较“古老而缓慢”的思想。然而，由于互联网公司也逐渐深入实体经济，业务日益复杂，我们在开发中也越来越多地遇到传统行业软件开发中所面临的问题。

怎么解决这个问题呢？其实法宝就是今天的主题，领域驱动设计！！相信你读完本文一定会有所启发。

### **DDD 介绍**

DDD 全程是 Domain-Driven Design，中文叫领域驱动设计，是一套应对复杂软件系统分析和设计的面向对象建模方法论。

以前的系统分析和设计是分开的，导致需求和成品非常容易出现偏差，两者相对独立，还会导致沟通困难，DDD 则打破了这种隔阂，提出了领域模型概念，统一了分析和设计编程，使得软件能够更灵活快速跟随需求变化。



DDD 的发展史

相信之前或多或少一定听说过领域驱动（DDD），繁多的概念会不会让你眼花缭乱？抽象的逻辑是不是感觉缺少落地实践？可能这也是 DDD 一直没得到盛行的原因吧。

话说 1967 年有了 OOP，1982 年有了 OOAD（面向对象分析和设计），它是成熟版的 OOP，目标就是解决复杂业务场景，这个过程中逐渐形成了一个领域驱动的思潮，一转眼到 2003 年的时候，Eric Evans 发表了一篇著作 *Domain-driven Design: Tackling Complexity in the Heart of Software*，正式定义了领域的概念，开始了 DDD 的时代。算下来也有接近 20 年的时间了，但是，事实并不像 Eric Evans 设想的那样容易，DDD 似乎一直不温不火，没有能“风靡全球”。

2013 年，Vaughn Vernon 写了一本 *Implementing Domain-Driven Design* 进一步定义了 DDD 的领域方向，并且给出了很多落地指导，它让人们离 DDD 又进了一步。

同时期，随着互联网的兴起，Rod Johnson 这大哥以轻量级极简风格的 Spring Cloud 抢占了所有风头，虽然 Spring 推崇的失血模式并非 OOP 的皇家血统，但是谁用关心这些呢？毕竟简化开发的成本才是硬道理。

就在我们用这张口闭口 Spring 的时候，我们意识到了一个严重的问题，我们应对复杂业务场景的时候，Spring 似乎并不能给出更合理的解决方案，于是分而治之的思想下应生了微服务，一改以往单体应用为多个子应用，一下子让人眼前一亮，于是我们没日没夜地拆分服务，加之微服务提供的注册中心、熔断、限流等解决方案，我们用得不亦乐乎。

人们在踩过诸多拆分服务的坑（拆分过细导致服务爆炸、拆分不合理导致频分重构等）之后，开始死锁原因了，到底有没有一种方法论可以指导人们更加合理地拆分服务呢？众里寻他千百度，DDD 却在灯火阑珊处，有了 DDD 的指导，加之微服务的事件，才是完美的架构。

### **DDD 与微服务的关系**

背景中我们说到，有 DDD 的指导，加之微服务的事件，才是完美的架构，这里就详细说下它们的关系。

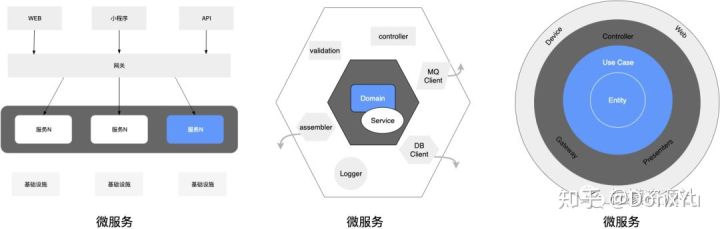
系统的复杂度越来越来高是必然趋势，原因可能来自自身业务的演进，也有可能是技术的创新，然而一个人和团队对复杂性的认知是有极限的，就像一个服务器的性能极限一样，解决的办法只有分而治之，将大问题拆解为小问题，最终突破这种极限。微服务在这方面都给出来了理论指导和最佳实践，诸如注册中心、熔断、限流等解决方案，但微服务并没有对“应对复杂业务场景”这个问题给出合理的解决方案，这是因为微服务的侧重点是治理，而不是分。

我们都知道，架构一个系统的时候，应该从以下几方面考虑：

1. 功能维度
2. 质量维度（包括性能和可用性）
3. 工程维度

微服务在第二个做得很好，但第一个维度和第三个维度做的不够。这就给 DDD 了一个“可乘之机”，DDD 给出了微服务在功能划分上没有给出的很好指导这个缺陷。所以说它们在面对复杂问题和构建系统时是一种互补的关系。

从架构角度看，微服务中的服务所关注的范围，正是 DDD 所推崇的六边形架构中的领域层，和整洁架构中的 entity 和 use cases 层。如下图所示：



DDD 与微服务如何协作

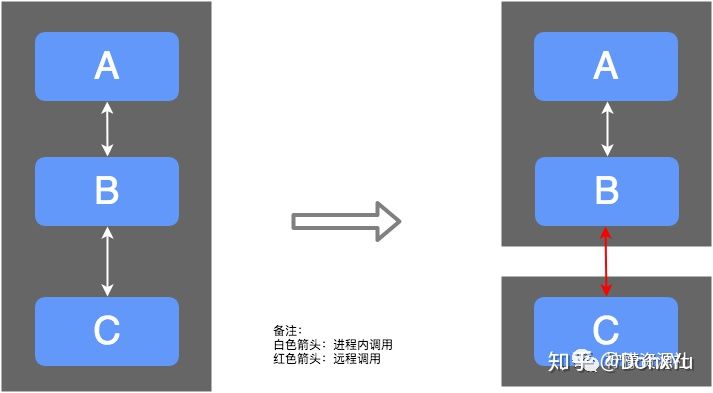
知道了 DDD 与微服务还不够，我们还需要知道他们是怎么协作的。

一个系统（或者一个公司）的业务范围和在这个范围里进行的活动，被称之为领域，领域是现实生活中面对的问题域，和软件系统无关，领域可以划分为子域，比如电商领域可以划分为商品子域、订单子域、发票子域、库存子域 等，在不同子域里，不同概念会有不同的含义，所以我们在建模的时候必须要有一个明确的边界，这个边界在 DDD 中被称之为限界上下文，它是系统架构内部的一个边界，《整洁之道》这本书里提到：

系统架构是由系统内部的架构边界，以及边界之间的依赖关系所定义的，与系统中组件之间的调用方式无关。  
所谓的服务本身只是一种比函数调用方式成本稍高的，分割应用程序行为的一种形式，与系统架构无关。

所以复杂系统划分的第一要素就是划分系统内部架构边界，也就是划分上下文，以及明确之间的关系，这对应之前说的第一维度（功能维度），这就是 DDD 的用武之处。其次，我们才考虑基于非功能的维度如何划分，这才是微服务发挥优势的地方。

假如我们把服务划分成 ABC 三个上下文：



我们可以在一个进程内部署单体应用，也可以通过远程调用来完成功能调用，这就是目前的微服务方式，更多的时候我们是两种方式的混合，比如 A 和 B 在一个部署单元内，C 单独部署，这是因为 C 非常重要，或并发量比较大，或需求变更比较频繁，这时候 C 独立部署有几个好处：

1. C 独立部署资源：资源更合理的倾斜，独立扩容缩容。
2. 弹力服务：重试、熔断、降级等，已达到故障隔离。
3. 技术栈独立：C 可以使用其他语言编写，更合适个性化团队技术栈。
4. 团队独立：可以由不同团队负责。

架构是可以演进的，所以拆分需要考虑架构的阶段，早期更注重业务逻辑边界，后期需要考虑更多方面，比如数据量、复杂性等，但即使有这个方针，也常会见仁见智，没有人能一下子将边界定义正确，其实这里根本就没有明确的对错。

即使边界定义的不太合适，通过聚合根可以保障我们能够演进出更合适的上下文，在上下文内部通过实体和值对象来对领域概念进行建模，一组实体和值对象归属于一个聚合根。

按照 DDD 的约束要求：

* 第一，聚合根来保证内部实体规则的正确性和数据一致性；
* 第二，外部对象只能通过 id 来引用聚合根，不能引用聚合根内部的实体；
* 第三，聚合根之间不能共享一个数据库事务，他们之间的数据一致性需要通过最终一致性来保证。

有了聚合根，再基于这些约束，未来可以根据需要，把聚合根升级为上下文，甚至拆分成微服务，都是比较容易的。

### **DDD 的相关术语与基本概念**

讨论完宏观概念以后，让我们来认识一下 DDD 的一些概念吧，每个概念我都为你找了一个 Spring 模式开发的映射概念，方便你理解，但要仅仅作为理解用，不要过于依赖。

另外，这里你可能需要结合后面的代码反复结合理解，才能融汇贯通到实际工作中。

### **领域**

映射概念：切分的服务。

领域就是范围。范围的重点是边界。领域的核心思想是将问题逐级细分来减低业务和系统的复杂度，这也是 DDD 讨论的核心。

### **子域**

映射概念：子服务。

领域可以进一步划分成子领域，即子域。这是处理高度复杂领域的设计思想，它试图分离技术实现的复杂性。这个拆分的里面在很多架构里都有，比如 C4。

### **核心域**

映射概念：核心服务。

在领域划分过程中，会不断划分子域，子域按重要程度会被划分成三类：核心域、通用域、支撑域。

决定产品核心竞争力的子域就是核心域，没有太多个性化诉求。

桃树的例子，有根、茎、叶、花、果、种子等六个子域，不同人理解的核心域不同，比如在果园里，核心域就是果是核心域，在公园里，核心域则是花。有时为了核心域的营养供应，还会剪掉通用域和支撑域（茎、叶等）。

### **通用域**

映射概念：中间件服务或第三方服务。

被多个子域使用的通用功能就是通用域，没有太多企业特征，比如权限认证。

### **支撑域**

映射概念：企业公共服务。

对于功能来讲是必须存在的，但它不对产品核心竞争力产生影响，也不包含通用功能，有企业特征，不具有通用性，比如数据代码类的数字字典系统。

### **统一语言**

映射概念：统一概念。

定义上下文的含义。它的价值是可以解决交流障碍，不管你是 RD、PM、QA 等什么角色，让每个团队使用统一的语言（概念）来交流，甚至可读性更好的代码。

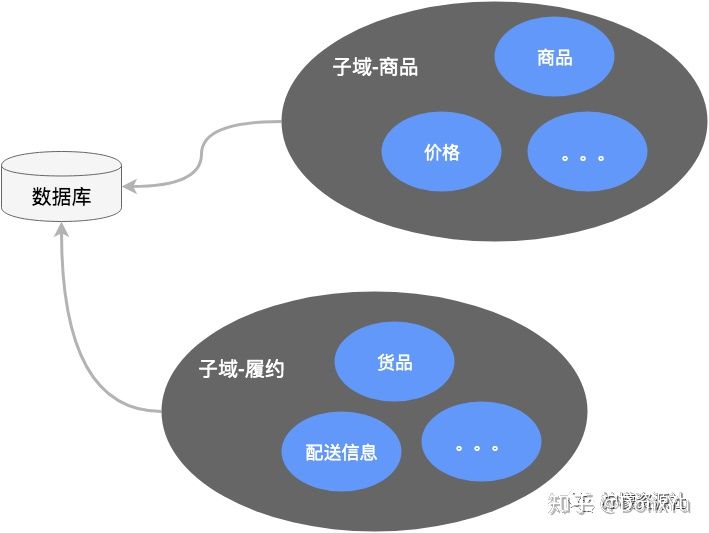
通用语言包含属于和用例场景，并且能直接反应在代码中。

可以在事件风暴（开会）中来统一语言，甚至是中英文的映射、业务与代码模型的映射等。可以使用一个表格来记录。

### **限界上下文**

映射概念：服务职责划分的边界。

定义上下文的边界。领域模型存在边界之内。对于同一个概念，不同上下文会有不同的理解，比如商品，在销售阶段叫商品，在运输阶段就叫货品。



理论上，限界上下文的边界就是微服务的边界，因此，理解限界上下文在设计中非常重要。

### **聚合**

映射概念：包。

聚合概念类似于你理解的包的概念，每个包里包含一类实体或者行为，它有助于分散系统复杂性，也是一种高层次的抽象，可以简化对领域模型的理解。

拆分的实体不能都放在一个服务里，这就涉及到了拆分，那么有拆分就有聚合。聚合是为了保证领域内对象之间的一致性问题。

在定义聚合的时候，应该遵守不变形约束法则：

1. 聚合边界内必须具有哪些信息，如果没有这些信息就不能称为一个有效的聚合；
2. 聚合内的某些对象的状态必须满足某个业务规则：

* 一个聚合只有一个聚合根，聚合根是可以独立存在的，聚合中其他实体或值对象依赖与聚合根。
* 只有聚合根才能被外部访问到，聚合根维护聚合的内部一致性。

### **聚合根**

映射概念：包。

一个上下文内可能包含多个聚合，每个聚合都有一个根实体，叫做聚合根，一个聚合只有一个聚合根。

### **实体**

映射概念：Domain 或 entity。

《领域驱动设计模式、原理与实践》一书中讲到，实体是具有身份和连贯性的领域概念，可以看出，实体其实也是一种特殊的领域，这里我们需要注意两点：唯一标示（身份）、连续性。两者缺一不可。

你可以想象，文章可以是实体，作者也可以是，因为它们有 id 作为唯一标示。

### **值对象**

映射概念：Domain 或 entity。

为了更好地展示领域模型之间的关系，制定的一个对象，本质上也是一种实体，但相对实体而言，它没有状态和身份标识，它存在的目的就是为了表示一个值，通常使用值对象来传达数量的形式来表示。

比如 money，让它具有 id 显然是不合理的，你也不可能通过 id 查询一个 money。

定义值对象要依照具体场景的区分来看，你甚至可以把 Article 中的 Author 当成一个值对象，但一定要清楚，Author 独立存在的时候是实体，或者要拿 Author 做复杂的业务逻辑，那么 Author 也会升级为聚合根。

最后，给出摘自网络的一张图，比较全，索性就直接 copy 过来了，便于你宏观回顾 DDD 的相关概念：

四种 Domain 模式

除了晦涩难懂的概念外，让我们最难接受的可能就是模型的运用了，Spring 思想中，Domain 只是数据的载体，所有行为都在 Service 中使用 Domain 封装后流转，而 OOP 讲究一对象维度来执行业务，所以，DDD 中的对象是用行为的（理解这点非常重要哦）。

这里我为你总结了全部的四种领域模式，供你区分和理解：

1. 失血模型
2. 贫血模型
3. 充血模型
4. 胀血模型

### **背景**

先说明一下示例背景，由于公司项目不能外泄的原因，我这里模拟一个文章管理系统（这个系统相对简单，理论上可以不使用 DDD，在这里仅做举例），业务需求有：发布文章、修改文章、文章分类搜索和展示等。

使用 Spring 开发的话，你脑海中一定浮现的是如下代码。

文章类：Article

public class Article implements Serializable {

private Integer id;

private String title;

private Integer classId;

private Integer authorId;

private String authorName;

private String content;

private Date pubDate;

//getter/setter/toString

}

DAO 类：ArticleDao/ArticleImpl

public interface ArticleDao extends BaseDao<Article>{

//...

}

Repository("articleDao")

public class ArticleDaoImpl implements ArticleDao{

//...

}

Service 类：ArticleService

public interface ArticleService extends BaseService<Article>{

//...

}

@Service(value="articleService")

public class ArticleServiceImpl implements ArticleService {

//...

}

Controller 类：略。

### **四种模式示例**

### **失血模型**

Domain Object 只有属性的 getter/setter 方法的纯数据类，所有的业务逻辑完全由 business object 来完成。

public class Article implements Serializable {

private Integer id;

private String title;

private Integer classId;

private Integer authorId;

private String authorName;

private String content;

private Date pubDate;

//getter/setter/toString

}

public interface ArticleDao {

public Article getArticleById(Integer id);

public Article findAll();

public void updateArticle(Article article);

}

### **贫血模型**

简单来说，就是 Domain Object 包含了不依赖于持久化的领域逻辑，而那些依赖持久化的领域逻辑被分离到 Service 层。

public class Article implements Serializable {

private Integer id;

private String title;

private Integer classId;

private Integer authorId;

private String authorName;

private String content;

private Date pubDate;

//getter/setter/toString

//判断是否是热门分类(假设等于57或102的类别的文章就是热门分类的文章)

public boolean isHotClass(Article article){

return Stream.of(57,102)

.anyMatch(classId -> classId.equals(article.getClassId()));

}

//更新分类，但未持久化，这里不能依赖Dao去操作实体化

public Article changeClass(Article article, ArticleClass ac){

return article.setClassId(ac.getId());

}

}

@Repository("articleDao")

public class ArticleDaoImpl implements ArticleDao{

@Resource

private ArticleDao articleDao;

public void changeClass(Article article, ArticleClass ac){

article.changeClass(article, ac);

articleDao.update(article)

}

}

注意这个模式不在 Domain 层里依赖 DAO。持久化的工作还需要在 DAO 或者 Service 中进行。

这样做的优缺点

优点：各层单向依赖，结构清晰。

缺点：

* Domain Object 的部分比较紧密依赖的持久化 Domain Logic 被分离到 Service 层，显得不够 OO
* Service 层过于厚重

### **充血模型**

充血模型和第二种模型差不多，区别在于业务逻辑划分，将绝大多数业务逻辑放到 Domain 中，Service 是很薄的一层，封装少量业务逻辑，并且不和 DAO 打交道：

Service (事务封装) —> Domain Object <—> DAO

public class Article implements Serializable {

@Resource

private static ArticleDao articleDao;

private Integer id;

private String title;

private Integer classId;

private Integer authorId;

private String authorName;

private String content;

private Date pubDate;

//getter/setter/toString

//使用articleDao进行持久化交互

public List<Article> findAll(){

return articleDao.findAll();

}

//判断是否是热门分类(假设等于57或102的类别的文章就是热门分类的文章)

public boolean isHotClass(Article article){

return Stream.of(57,102)

.anyMatch(classId -> classId.equals(article.getClassId()));

}

//更新分类，但未持久化，这里不能依赖Dao去操作实体化

public Article changeClass(Article article, ArticleClass ac){

return article.setClassId(ac.getId());

}

}

所有业务逻辑都在 Domain 中，事务管理也在 Item 中实现。这样做的优缺点如下。

优点：

* 更加符合 OO 的原则；
* Service 层很薄，只充当 Facade 的角色，不和 DAO 打交道。

缺点：

* DAO 和 Domain Object 形成了双向依赖，复杂的双向依赖会导致很多潜在的问题。
* 如何划分 Service 层逻辑和 Domain 层逻辑是非常含混的，在实际项目中，由于设计和开发人员的水平差异，可能 导致整个结构的混乱无序。

### **胀血模型**

基于充血模型的第三个缺点，有同学提出，干脆取消 Service 层，只剩下 Domain Object 和 DAO 两层，在 Domain Object 的 Domain Logic 上面封装事务。

Domain Object (事务封装，业务逻辑) <—> DAO

似乎 Ruby on rails 就是这种模型，它甚至把 Domain Object 和 DAO 都合并了。

这样做的优缺点：

* 简化了分层
* 也算符合 OO

该模型缺点：

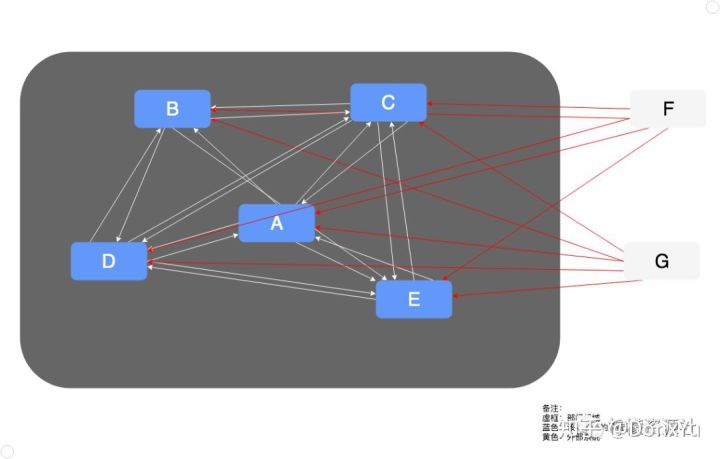
* 很多不是 Domain Logic 的 Service 逻辑也被强行放入 Domain Object ，引起了 Domain Object 模型的不稳定；
* Domain Object 暴露给 Web 层过多的信息，可能引起意想不到的副作用。

### **运用 DDD 改造现有旧系统实践**

假如你是一个团队 Leader 或者架构师，当你接手一个旧系统维护及重构的任务时，你该如何改造呢？是否觉得哪里都不对但由于业务认知的不熟悉而无从下手呢？其实这里我可以教你一套方法来应对这种窘境。

你要做的大概以下几点：

1. 通过公共平台大概梳理出系统之间的调用关系（一般中等以上公司都具备 RPC 和 HTTP 调用关系，无脑的挨个系统查询即可），画出来的可能会很乱，也可能会比较清晰，但这就是现状。



2. 分配组员每个人认领几个项目，来梳理项目维度关系，这些关系包括：对外接口、交互、用例、MQ 等的详细说明。个别核心系统可以画出内部实体或者聚合根。

3. 小组开会，挨个 review 每个系统的业务概念，达到组内统一语言。



4. 根据以上资料，即可看出哪些不合理的调用关系（比如循环调用、不规范的调用等），甚至不合理的分层。

5. 根据主线业务自顶向下细分领域，以及限界上下文。此过程可能会颠覆之前的系统划分。

6. 根据业务复杂性，指定领域模型，选择贫血或者充血模型。团队内部最好实行统一习惯，以免出现交接成本过大。

7. 分工进行开发，并设置 deadline，注意，不要单一的设置一个 deadline，要设置中间 check 时间，比如 dealline 是 1 月 20 日，还要设置两个 check 时间，分别沟通代码风格及边界职责，以免 deadline 时延期。

### **DDD 与 Spring 家族的完美结合**

还用前面提到的文章管理系统，我为你说明一下 DDD 开发的关注点。

### **模块（Module）**

模块（Module）是 DDD 中明确提到的一种控制限界上下文的手段，在我们的工程中，一般尽量用一个模块来表示一个领域的限界上下文。

如代码中所示，一般的工程中包的组织方式为 {com.公司名.组织架构.业务.上下文.\*}，这样的组织结构能够明确地将一个上下文限定在包的内部。

import com.company.team.bussiness.counter.\*;//计数上下文

import com.company.team.bussiness.category.\*;//分类上下文

import com.company.team.bussiness.comment.\*;//评论上下文

对于模块内的组织结构，一般情况下我们是按照领域对象、领域服务、领域资源库、防腐层等组织方式定义的。

import com.company.team.bussiness.cms.domain.valobj.\*;//领域对象-值对象

import com.company.team.bussiness.cms.domain.entity.\*;//领域对象-实体

import com.company.team.bussiness.cms.domain.aggregate.\*;//领域对象-聚合根

import com.company.team.bussiness.cms.service.\*;//领域服务

import com.company.team.bussiness.cms.repo.\*;//领域资源库

import com.company.team.bussiness.cms.facade.\*;//领域防腐层

### **领域对象**

领域驱动要解决的一个重要的问题，就是解决对象的贫血问题，而领域对象则最直接的反应了这个能力。

我们可以定义聚合根（文章）和值对象（计数器），来举例说明。聚合根持有文章的 id 和文章的计数数据，这里计数器之所以被列为值对象，而非实体的一个属性，是因为计数器是由多部分组成的，比如真实阅读量、推广阅读量等。

在文章领域对象中，我们需要定义个一个方法，来获取文章的计数量，用于页面上显示，这个逻辑可能会很复杂，涉及到爆文、专栏作者级别、发布时间等因素。

package com.company.team.bussiness.domain.aggregate;

import ...;

public class Article {

@Resource

private CategoryRepository categoryRepository;

private int articleId; //文章id

...

private ArticleCount articleCount; //文章计数器

//getter & setter

//查询计数显示数量，这里简化一些逻辑，甚至是不符合实际业务场景，这不重要，这里只为直观表达意思

public Integer getShowArticleCount() {

if(this.articleCount == null){

return 0;

}

return this.articleCount.realCount + categoryRepository.getCategoryWeight(this.category) + (this.articleCount.adCount \* DayUtils.calDaysByNow(this.articleCount.deadDays));

}

}

与以往的仅有 getter、setter 的业务对象不同，领域对象具有了行为，对象更加丰满。同时，比起将这些逻辑写在服务内（例如 Service），领域功能的内聚性更强，职责更加明确。

### **资源库**

领域对象需要资源存储，资源库可以理解成 DAO，但它比 DAO 更宽泛，存储的手段可以是多样化的，常见的无非是数据库、分布式缓存、本地缓存等。资源库（Repository）的作用，就是对领域的存储和访问进行统一管理的对象。

在系统中，我们是通过如下的方式组织资源库的。

import com.company.team.bussiness.repo.dao.ArticleDao;//数据库访问对象-文章

import com.company.team.bussiness.repo.dao.CommentDao;//数据库访问对象-评论

import com.company.team.bussiness.repo.dao.po.ArticlePO;//数据库持久化对象-文章

import com.company.team.bussiness.repo.dao.po.CommentPO;//数据库持久化对象-评论

import com.company.team.bussiness.repo.cache.ArticleObj;//分布式缓存访问对象-文章缓存访问

资源库对外的整体访问由 Repository 提供，它聚合了各个资源库的数据信息，同时也承担了资源存储的逻辑（例如缓存更新机制等）。

在资源库中，我们屏蔽了对底层奖池和奖品的直接访问，而是仅对文章的聚合根进行资源管理。代码示例中展示了资源获取的方法（最常见的 Cache Aside Pattern）。

package com.company.team.bussiness.repo;

import ...;

@Repository

public class ArticleRepository {

@Autowired

private ArticleDao articleDao;

@AutoWired

private articleDaoCacheAccessObj articleCacheAccessObj;

public Article getArticleById(int articleId) {

Article article = articleCacheAccessObj.get(articleId);

if(article!=null){

return article;

}

article = getArticleFromDB(articleId);

articleCacheAccessObj.add(articleId, article);

return article;

}

private Article getArticleFromDB(int articleId) {...}

}

比起以往将资源管理放在服务中的做法，由资源库对资源进行管理，职责更加明确，代码的可读性和可维护性也更强。

### **防腐层**

亦称适配层。在一个上下文中，有时需要对外部上下文进行访问，通常会引入防腐层的概念来对外部上下文的访问进行一次转义。

有以下几种情况会考虑引入防腐层：

* 需要将外部上下文中的模型翻译成本上下文理解的模型。
* 不同上下文之间的团队协作关系，如果是供奉者关系，建议引入防腐层，避免外部上下文变化对本上下文的侵蚀。
* 该访问本上下文使用广泛，为了避免改动影响范围过大。

package com.company.team.bussiness.facade;

import ...;

@Component

public class ArticleFacade {

@Resource

private ArticleService articleService;

public Article getArticle(ArticleContext context) {

ArticleResponse resp = articleService.getArticle(context.getArticleId());

return buildArticle(resp);

}

private Article buildArticle(ArticleResponse resp) {...}

}

如果内部多个上下文对外部上下文需要访问，那么可以考虑将其放到通用上下文中。

### **领域服务**

上文中，我们将领域行为封装到领域对象中，将资源管理行为封装到资源库中，将外部上下文的交互行为封装到防腐层中。此时，我们再回过头来看领域服务时，能够发现领域服务本身所承载的职责也就更加清晰了，即就是通过串联领域对象、资源库和防腐层等一系列领域内的对象的行为，对其他上下文提供交互的接口。

package com.company.team.bussiness.service.impl

import ...;

@Service

public class CommentServiceImpl implements CommentService {

@Resource

private CommentFacade commentFacade;

@Resource

private ArticleRepository articleRepo;

@Resource

private ArticleService articleService;

@Override

public CommentResponse commentArticle(CommentContext commentContext) {

Article article = articleRepo.getArticleById(commentContext.getArticleId());//获取文章聚合根

commentFacade.doComment(commentContext);//增加计数信息

return buildCommentResponse(commentContext,article);//组装评论后的文章信息

}

private CommentResponse buildCommentResponse(CommentContext commentContext, Article article) {...}

}

可以看到在省略了一些防御性逻辑（异常处理、空值判断等）后，领域服务的逻辑已经足够清晰明了。

### **示范包结构**



### **反思思考**

DDD 将领域层进行了细分，是 DDD 比较 MVC 框架的最大亮点。

DDD 能做到这一点，主要是因为 DDD 将领域层进行了细分，比如说领域对象有实体、聚合，动作和操作叫做领域服务，能力叫做领域能力等，而 MVC 架构并没有对业务元素进行细分，所有的业务都是 Service，从而导致 Controller 层和 Service 层很难定义出技术约束，因为都是 Service，你不会知道这个 Service 是用来描述对象的还是来描述一个业务操作的。

针对未来业务扩展方面，聚合根升级为上下文，甚至拆分成微服务，也是应对复杂问题的重要手段。

实体和值对象是对现有编程习惯最大的变化，但不要过度关注而忽略了领域对象之间的关系。

DDD 本身是方法论，是提供理论指导的，所以不要奢求像 Spring 那样给你一个 Demo 照着写，希望读者看完后多多反思。