《领域驱动设计 软件核心复杂性应对之道》及《DDD实战课》学习总结

张赛 00515860

**DDD是什么？**

DDD(Domain-Driven Design)，领域驱动设计，2004 年埃里克·埃文斯（Eric Evans）发表了《领域驱动设计》（Domain-Driven Design –Tackling Complexity in the Heart of Software）这本书，从此领域驱动设计诞生。DDD 核心思想是通过领域驱动设计方法定义领域模型，从而确定业务和应用边界，保证业务模型与代码模型的一致性。

我的理解，领域设计是架构设计的最高层抽象，是业务向技术架构转换的纽带。是软件所有强相关人员一起讨论业务，统一语言，输出所有人都看的懂的软件顶层设计的过程。

这样的讨论，群策群力，逻辑完整可信，场景和依赖关系考虑全面，后续返工少。一次把事情做对。

**DDD能做什么？**

         大家都知道软件的最高境界是高内聚、低耦合，符合SOLID原则。那么，到底如何符合这些原则呢，就像说你要实事求是，遵守客观规律这样永远正确的废话一样，关键是这么高度的抽象，在具体落地时如何保证有效落地，这就是DDD的事情了。

         DDD生来就是为了实现业务与代码模型的一致，根据业务划分领域，实现代码领域内高内聚低耦合的。

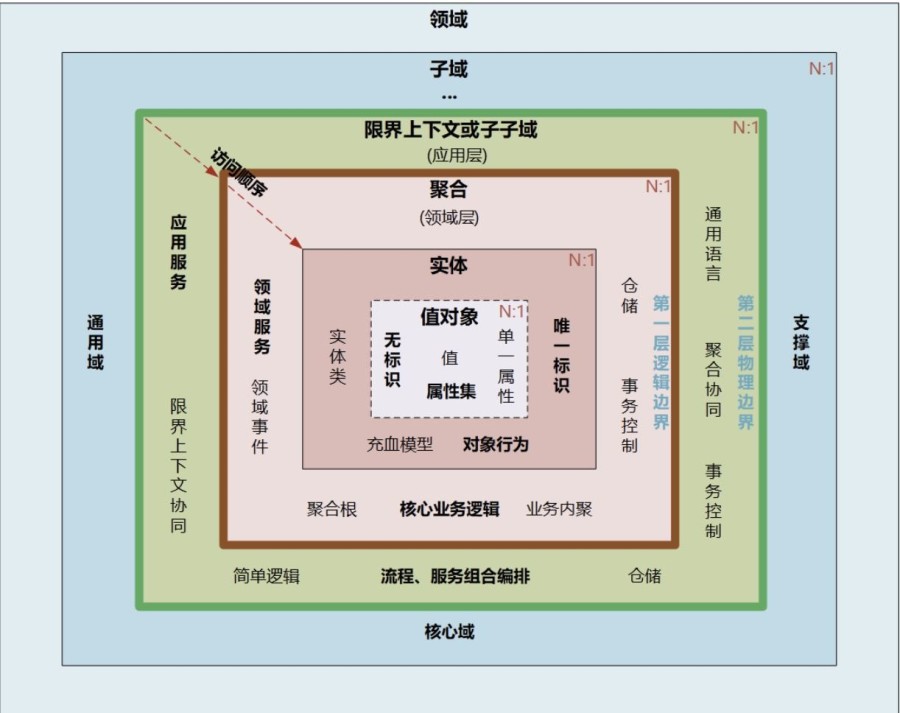


图1 DDD示意图

         DDD试图分离技术实现的复杂性，并围绕业务概念构建领域模型来控制业务的复杂性，以解决软件难以理解，难以演进的问题。DDD不是架构，而是一种架构设计方法论，它通过边界划分将复杂业务领域简单化，帮我们设计出清晰的领域和应用边界，可以很容易地实现架构演进。

         DDD的核心目标是让核心实体内聚到最核心位置，支撑域和通用域区分开，聚焦核心价值，逐层提供能力，外层向内层依赖，严禁反向依赖。

         核心域，就是软件的核心价值所在，卖点所在。

**DDD于微服务架构的价值**

         目前见到华为，乃至业界，很多上微服务架构的产品或系统，都存在很多问题。甚至问题严重到收益远小于问题的程度。是微服务架构本身的问题么？显然并不是。

         微服务架构不过是放大了的类图，以前在一个进程里低内聚、高耦合的问题，在微服务架构下，被放大了，资源消耗、循环依赖带来的难扩展，难定位，难维护的问题凸显了而已。

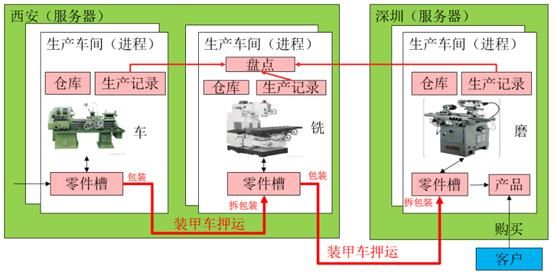


图2  微服务原理图

如上图，原来一个车间里的生产线，按微服务架构拆成多个后，数据分散后统计分析、不断的跨车间的装运零件，而且在华为严格的安全红线要求下必须用重装甲车运输，效率实在低的难以入目。

         所以，这些车间关系如何，顺序如何，哪些应该在一起，哪些可以远一点？这个事情至关重要。这也就是DDD的价值所在了。

**DDD如何做**

         DDD理解上一直比较晦涩，它没有像UML那样直观表达效果，也没有4+1那样的直观逻辑。

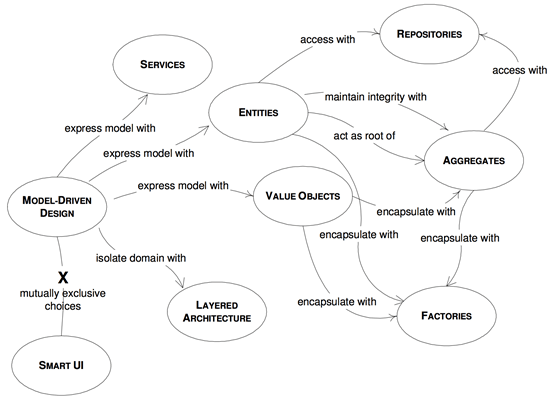


图3 DDD概念及关系图

以上是DDD所涉及的概念及其关系。还是很难懂，不过没关系，下面列详细过程，并举例来说明。

领域建模时，我们会根据场景分析过程中产生的领域对象，比如命令、事件等之间关系，找出产生命令的实体，分析实体之间的依赖关系组成聚合，为聚合划定限界上下文，建立领域模型以及模型之间的依赖。领域模型利用限界上下文向上可以指导微服务设计，通过聚合向下可以指导聚合根、实体和值对象的设计。

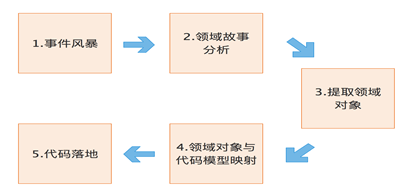


图4 DDD工作流程图

**1 事件风暴，梳理领域事件**

         领域事件可以是业务流程的一个步骤，比如投保业务缴费完成后，触发投保单转保单的动作；也可能是定时批处理过程中发生的事件，比如批处理生成季缴保费通知单，触发发送缴费邮件通知操作；或者一个事件发生后触发的后续动作，比如密码连续输错三次，触发锁定账户的动作。

         比如：“如果发生……，则……”“当做完……的时候，请通知……” “发生……时，则……”等。

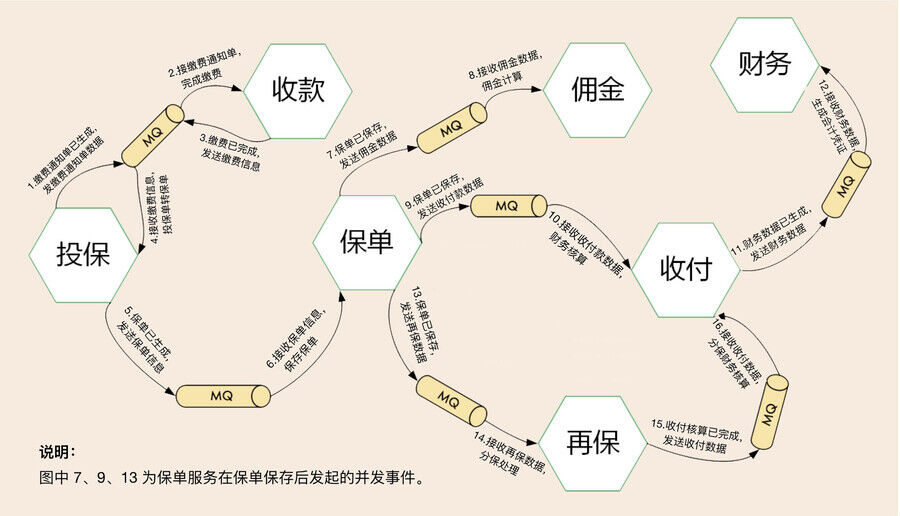


图5 事件风暴示意图

举例如上图。

领域事件处理包括：事件构建和发布、事件数据持久化、事件总线、消息中间件、事件接收和处理等。

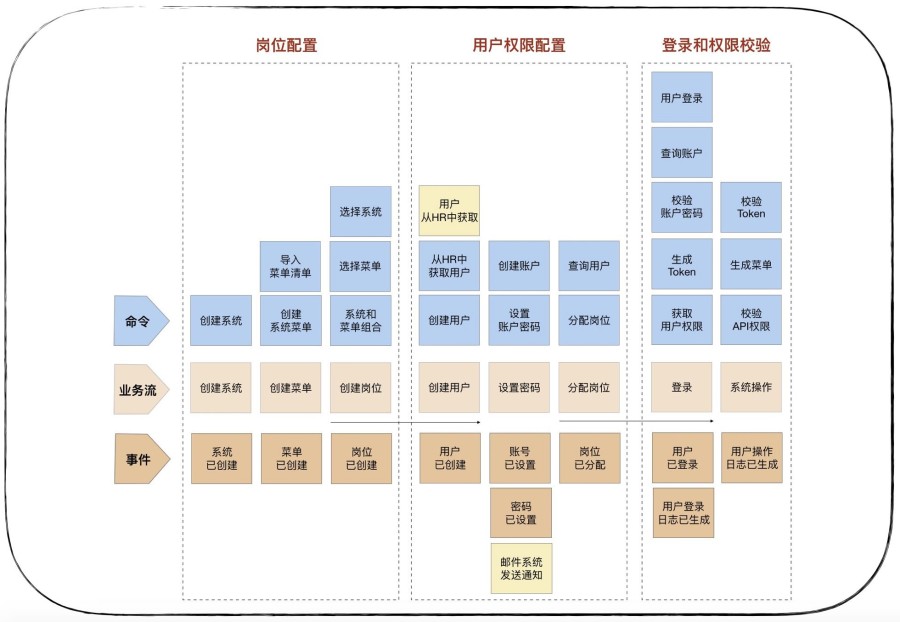


图6 事件风暴工作板示意

**2 划分领域，识别核心域，通用域，支撑域**

         通过用户故事分析会形成一个个的领域对象，这些领域对象对应领域模型的业务对象，每一个业务对象和领域对象都有通用的名词术语，并且一一映射。

         决定产品和公司核心竞争力的子域是核心域，它是业务成功的主要因素和公司的核心竞争力。没有太多个性化的诉求，同时被多个子域使用的通用功能子域是通用域。还有一种功能子域是必需的，但既不包含决定产品和公司核心竞争力的功能，也不包含通用功能的子域，它就是支撑域。

         如图6分析后得出以下领域划分：

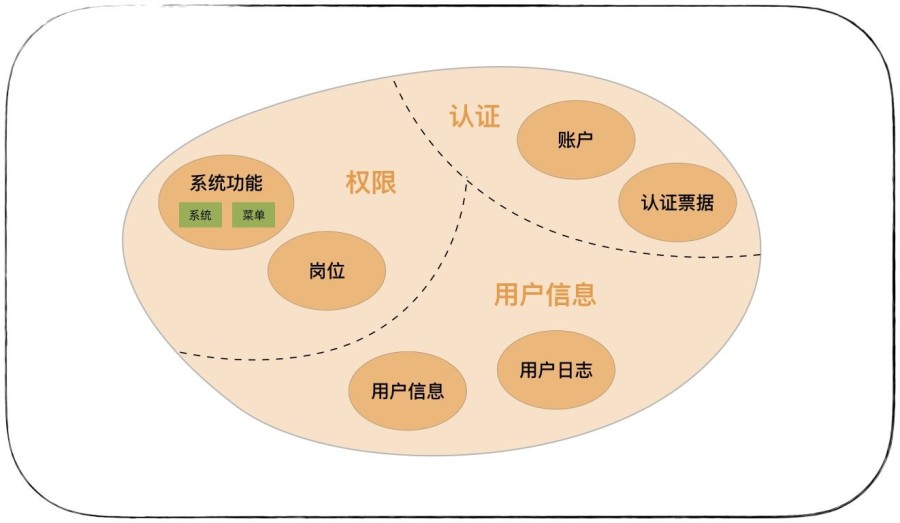


图7 领域划分

**3限界上下文**

限界上下文的定义就是：用来封装通用语言和领域对象，提供上下文环境，保证在领域之内的一些术语、业务相关对象等（通用语言）有一个确切的含义，没有二义性。这个边界定义了模型的适用范围，使团队所有成员能够明确地知道什么应该在模型中实现，什么不应该在模型中实现。

图7，在划分领域的同时，实际上已经做了限界上下文了。 以上为战略设计。

**4 根据领域识别实体和值对象**

         实体，就是整个业务生命周期过程中都有唯一ID标识的对象。比如人，有身份证号。对应代码里的DO。

值对象，没有业务标识符的数据。比如人的居住地址，是附属信息。

这个跟数据库设计中的E-R图有点类似。

**5 寻找聚合和聚合根，梳理依赖关系，确认最终模型**

         领域模型内的实体和值对象就好比个体，而能让实体和值对象协同工作的组织就是聚合，它用来确保这些领域对象在实现共同的业务逻辑时，能保证数据的一致性。

         所以，系统的复杂性就体现在了聚合以及聚合间的耦合。

         聚合根的主要目的是为了避免由于复杂数据模型缺少统一的业务规则控制，而导致聚合、实体之间数据不一致性的问题。

如果把聚合比作组织，那聚合根就是这个组织的负责人。聚合根也称为根实体，它不仅是实体，还是聚合的管理者。

举保险的例子说明聚合设计：

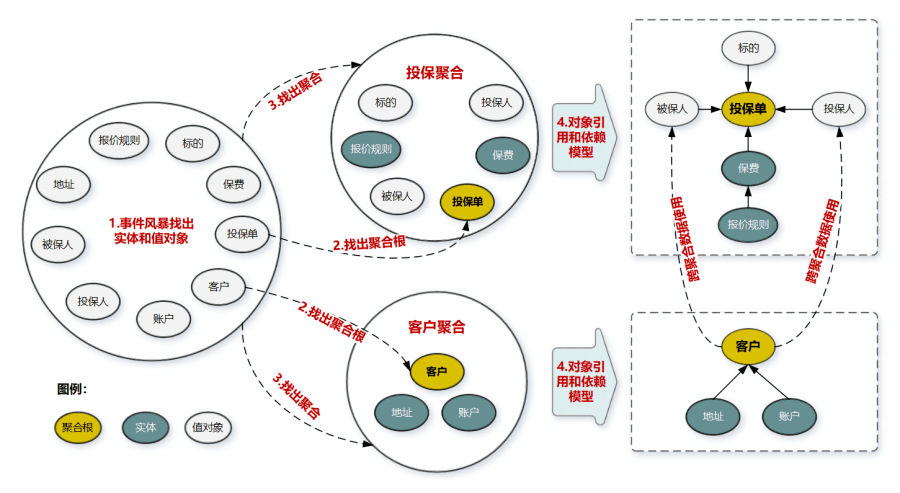


图8 聚合设计

（1）       通过事件风暴和实体及值对象的识别。

（2）       选择合适对象为根实体，比如保险单、客户。判断根实体办法：是否有独立的生命周期？是否有全局唯一 ID？是否可以创建或修改其它对象？是否有专门的模块来管这个实体。

（3）       根据业务单一职责和高内聚原则，找出与聚合根关联的所有紧密依赖的实体和值对象。构建出 1 个包含聚合根（唯一）、多个实体和值对象的对象集合，这个集合就是聚合。在图中我们构建了客户和投保这两个聚合。

（4）       在聚合内根据聚合根、实体和值对象的依赖关系，画出对象的引用和依赖模型。这里我需要说明一下：投保人和被保人的数据，是通过关联客户 ID 从客户聚合中获取的，在投保聚合里它们是投保单的值对象，这些值对象的数据是客户的冗余数据，即使未来客户聚合的数据发生了变更，也不会影响投保单的值对象数据。从图中我们还可以看出实体之间的引用关系，比如在投保聚合里投保单聚合根引用了报价单实体，报价单实体则引用了报价规则子实体。

（5）       多个聚合根据业务语义和上下文一起划分到同一个限界上下文内。

综上可以看出，聚合设计是对第二步领域划分的一个补充和证明。

**6 根据领域模型输出代码模型, 代码落地**

         先看下DDD的四层架构模型：

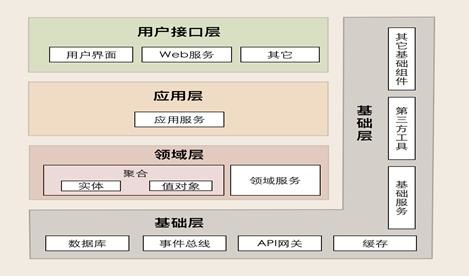


         图9  DDD四层架构

         理论上依赖关系是自上向下。这里基础层可能会有点问题，需要依赖倒置，注入到其它几层，将依赖反转。

         适配上述四层架构，代码模型应该为：

**微服务一级目录结构**

微服务一级目录是按照 DDD 分层架构的分层职责来定义的。从下面这张图中，我们可以看到，在代码模型里分别为用户接口层、应用层、领域层和基础层，建立了 interfaces、application、domain 和 infrastructure 四个一级代码目录。



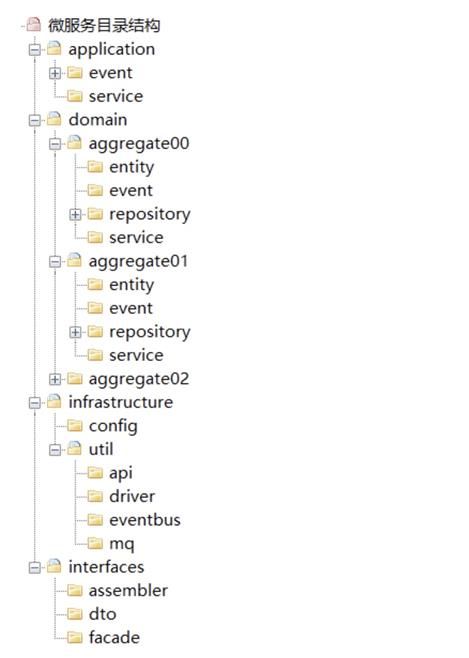


图10 代码目录

这些目录的职能和代码形态是这样的。

**Interfaces（用户接口层）：**

它主要存放用户接口层与前端交互、展现数据相关的代码。前端应用通过这一层的接口，向应用服务获取展现所需的数据。这一层主要用来处理用户发送的 Restful 请求，解析用户输入的配置文件，并将数据传递给

Interfaces 的代码目录结构有：assembler、dto 和 façade 三类。Assembler：实现 DTO 与领域对象之间的相互转换和数据交换。一般来说 Assembler 与 DTO 总是一同出现。Dto：它是数据传输的载体，内部不存在任何业务逻辑，我们可以通过 DTO 把内部的领域对象与外界隔离。Facade：提供较粗粒度的调用接口，将用户请求委派给一个或多个应用服务进行处理。

**Application（应用层）：**

它主要存放应用层服务组合和编排相关的代码。应用服务向下基于微服务内的领域服务或外部微服务的应用服务完成服务的编排和组合，向上为用户接口层提供各种应用数据展现支持服务。应用服务和事件等代码会放在这一层目录里。

Application 层。数据的组装、数据传输格式以及 Facade 接口等代码都会放在这一层目录里。

Application 的代码目录结构有：event 和 service。Event（事件）：这层目录主要存放事件相关的代码。它包括两个子目录：publish 和 subscribe。前者主要存放事件发布相关代码，后者主要存放事件订阅相关代码（事件处理相关的核心业务逻辑在领域层实现）。这里提示一下：虽然应用层和领域层都可以进行事件的发布和处理，但为了实现事件的统一管理，我建议你将微服务内所有事件的发布和订阅的处理都统一放到应用层，事件相关的核心业务逻辑实现放在领域层。通过应用层调用领域层服务，来实现完整的事件发布和订阅处理流程。Service（应用服务）：这层的服务是应用服务。应用服务会对多个领域服务或外部应用服务进行封装、编排和组合，对外提供粗粒度的服务。应用服务主要实现服务组合和编排，是一段独立的业务逻辑。你可以将所有应用服务放在一个应用服务类里，也可以把一个应用服务设计为一个应用服务类，以防应用服务类代码量过大。

**Domain（领域层）：**

它主要存放领域层核心业务逻辑相关的代码。领域层可以包含多个聚合代码包，它们共同实现领域模型的核心业务逻辑。聚合以及聚合内的实体、方法、领域服务和事件等代码会放在这一层目录里。

Domain 是由一个或多个聚合包构成，共同实现领域模型的核心业务逻辑。聚合内的代码模型是标准和统一的，包括：entity、event、repository 和 service 四个子目录。而领域层聚合内部的代码目录结构是这样的。

Aggregate（聚合）：它是聚合软件包的根目录，可以根据实际项目的聚合名称命名，比如权限聚合。在聚合内定义聚合根、实体和值对象以及领域服务之间的关系和边界。聚合内实现高内聚的业务逻辑，它的代码可以独立拆分为微服务。以聚合为单位的代码放在一个包里的主要目的是为了业务内聚，而更大的目的是为了以后微服务之间聚合的重组。聚合之间清晰的代码边界，可以让你轻松地实现以聚合为单位的微服务重组，在微服务架构演进中有着很重要的作用。

Entity（实体）：它存放聚合根、实体、值对象以及工厂模式（Factory）相关代码。实体类采用充血模型，同一实体相关的业务逻辑都在实体类代码中实现。跨实体的业务逻辑代码在领域服务中实现。

Event（事件）：它存放事件实体以及与事件活动相关的业务逻辑代码。

Service（领域服务）：它存放领域服务代码。一个领域服务是多个实体组合出来的一段业务逻辑。你可以将聚合内所有领域服务都放在一个领域服务类中，你也可以把每一个领域服务设计为一个类。如果领域服务内的业务逻辑相对复杂，我建议你将一个领域服务设计为一个领域服务类，避免由于所有领域服务代码都放在一个领域服务类中，而出现代码臃肿的问题。领域服务封装多个实体或方法后向上层提供应用服务调用。

Repository（仓储）：它存放所在聚合的查询或持久化领域对象的代码，通常包括仓储接口和仓储实现方法。为了方便聚合的拆分和组合，我们设定了一个原则：一个聚合对应一个仓储。特别说明：按照 DDD 分层架构，仓储实现本应该属于基础层代码，但为了在微服务架构演进时，保证代码拆分和重组的便利性，我是把聚合仓储实现的代码放到了聚合包内。这样，如果需求或者设计发生变化导致聚合需要拆分或重组时，我们就可以将包括核心业务逻辑和仓储代码的聚合包整体迁移，轻松实现微服务架构演进。

**Infrastructure（基础层）：**

它主要存放基础资源服务相关的代码，为其它各层提供的通用技术能力、三方软件包、数据库服务、配置和基础资源服务的代码都会放在这一层目录里。

Infrastructure 的代码目录结构有：config 和 util 两个子目录。Config：主要存放配置相关代码。Util：主要存放平台、开发框架、消息、数据库、缓存、文件、总线、网关、第三方类库、通用算法等基础代码，你可以为不同的资源类别建立不同的子目录。

**如何应用DDD持续演进ManageOne微服务架构**

         未完待续… …