**为什么需要DDD？**

<https://mp.weixin.qq.com/s/W-VW2cGh3nHPe-mShuJIOQ>

## 为什么需要DDD？

没有实施DDD的情况下，我们经常会遇到什么问题？

1. 开发人员热衷于技术而不是深入了解业务。这是技术人员的职责使然，一个不高级的开发，通常他的业务经验不重要；一个高级的开发，通常因为竞业，也无法继续干类似的业务。所以开发人员对业务天然的没有足够的兴趣。但是开发过程中，对业务不够熟悉，很容易发现开发做了半天得到的，并不是用户和产品想要的；或者下次再有需求的时候，技术上的改动特别大，成本很高。
2. 业务协作不畅，一个需求提到好几个团队都能做，但是开发过程你推我我推你，需求一拖再拖，有时候项目中期还得找其他团队求资源。
3. 对项目工时的估计占用了不少精力，还不准确。估时这件事能成为管理层和开发人员之间的拉锯战。
4. 服务之间紧耦合，牵一发而动全身，一个非核心的业务抖一抖，客户都说没法用。

## 那么DDD怎么解决这些问题？

* 找到边界：让设计系统的人知道一个业务的边界在哪里。只有知道边界在哪里，才能在需求到来的时候，轻易地找到相关团队，各个业务之间也才能真正解耦，降低非核心功能对核心功能的影响。
* 知识获取：保证设计系统的人能够低成本地了解业务，让大家在怎么做，怎么验收方面达成一致。在此基础上，还可以免费得到一款评估时间的工具，项目的交付就更有把握。

但是在此我要多说一下，我们现实实践中已经有一部分领域概念的影子了——谁能说他不知道自己的组织是干啥的？或者说哪个组织没有业务重心呢？可是为什么大家没有获得上边说的这诸多好处呢？那是因为DDD实践过程中巧妙地将设计模型落地成为开发的模型，让需求方和实施方说一种语言，才能真正跨越了需求和实现之间的鸿沟。所以实践DDD，绝不是只有开发写写代码就行，而是要跟产品，设计以及领域专家一起完成设计，才能得到DDD的好处。

## DDD是什么呢？

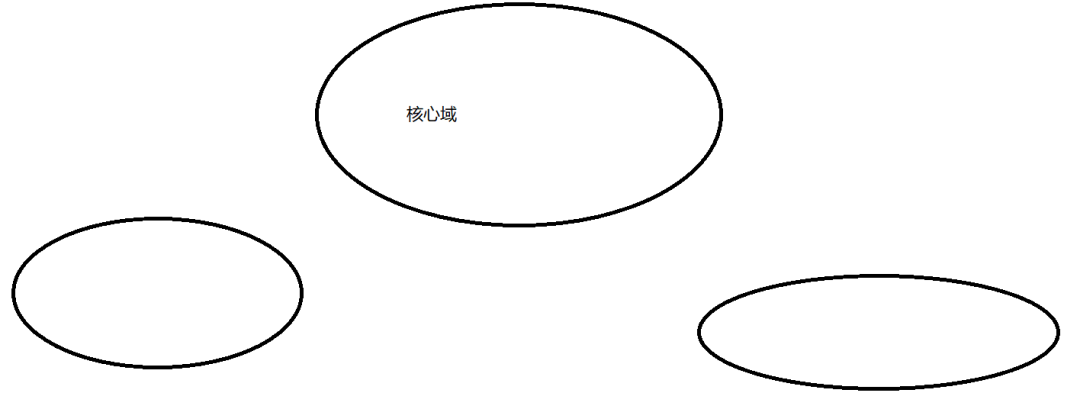
我们先看几个\*DD：

* TDD：驱动测试开发
* BDD：行为驱动开发
* ADD：不是单词Add，而是API驱动开发
* DDD：领域驱动设计

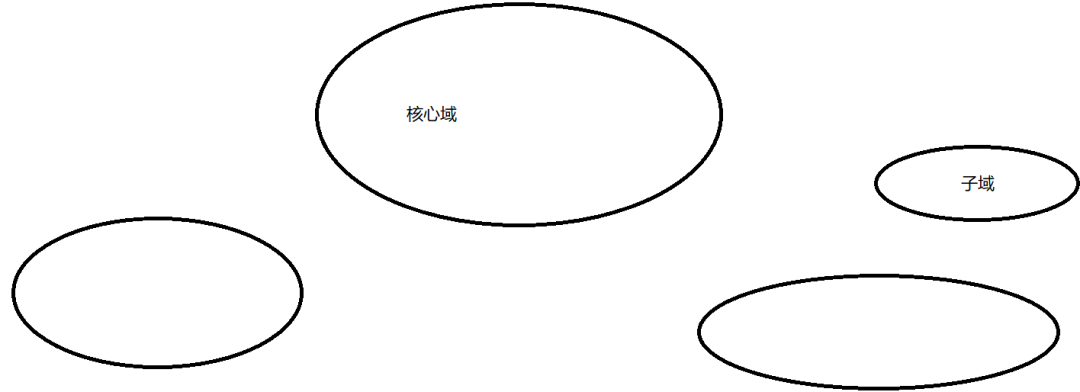
前面几个概念落脚点都是开发，而DDD，是设计。  
它有三个关键词：领域，驱动，设计。领域，是要探索业务的边界；驱动,表示前者是后者的决定性因素；设计，包括产品设计，UIUE设计，软件设计。它不仅仅是开发架构的方案，而是完整的解决方案实施思路。正是因为它是完整的方案，才能让领域专家，产品和研发真正在同一个角度去思考和沟通，避免推诿扯皮，含糊不清。

## 那么怎么做DDD呢？

实施DDD一般有两步，并且需要开发，产品和领域专家的通力合作。为了实施速度有所保障，还有一些项目加速和项目管理工具：  
**战略设计**  
战略设计可以说是搭建了业务思想上的框架。这个阶段要做这么几件事：  
**限界上下文分离领域模型**  
限界上下文这个名字乍一看，每个字我都认识，但是这个词是啥意思？原文说它是语义和语境上的边界，我的理解是，它是在描述组织交付出来，面向客户的交付边界。如果是在SaaS场景，一个限界上下文应该是一个独立交付的软件；在PaaS场景，它应该说的是一个独立售卖的模块。它的含义是找到一个边界，要把这个边界以外的当成是无法改变的客观环境，不要幻想这个边界以外的人会配合你一起完成交付。那这一步设计就很好理解了，就是找到你业务对外承诺的边界，你要发展的业务在这个边界内，而不在此之外。如果你是对内交付的系统，那么你对其他同事交付的业务边界，就是你的业务限界上下文。  
一个组织里，最核心的限界上下文被称为核心域。通常除了它，还有通用子域和支撑子域。通用子域是很成熟的业务，通常可以外包或者购买现成的解决方案，比如搜索子域可以通过ES来支持；支撑子域通常没有现成产品，但是它没有核心域重要，因此也可以一定程度的外包，避免在核心域之外浪费资源，比如大多数公司的数据库中间件是在开源产品上做了一些定制开发和维护。  
限界上下文这个概念的目的是为了在业务扩展的时候，防止向领域内注入概念，导致业务变得没有边界，纠缠在一起。  
在做这一步的时候，DDD要求以领域专家意见为准，正所谓领域驱动嘛。当实施了DDD方法以后，不论是领域专家还是开发，都应该拒绝向领域注入与业务无关的概念，比如存储方式等。这与我们日常工作从如何存储开始构建业务系统是完全不同的。只有把这些技术概念放到业务之外，我们的业务核心往往才能足够集中，易于迁移，而且不论采用什么东西存储，用什么东西展示，它的逻辑都可以不变。  
这个过程中我们通常可以得到这样一个模型：



**限界上下文发展通用语言**  
当我们有了业务的限界上下文以后，就需要在这个限界上下文中发展一种语言用于表达软件模型，这个语言就叫做这个限界上下文里的通用语言。它可以是任何计算机语言、人类语言或者图形，只要能让团队内的每个人都能看懂。  
通用语言不止是名词，它应该使用一系列具体的模型场景来描述领域模型。它描述了各种业务组件（不是技术组件）做什么，而不是用例或者用户故事。  
比如微信朋友圈点赞这个场景，通用语言可能是：用户可以通过点赞，使得某个朋友圈的Feed发出人收到被点赞的通知，达到互动的目的。  
但是到这一步，我们怎么能验证领域模型能与领域专家的心智保持一致呢？那就是为这个模型写验收测试，并交由领域专家评估。一种做法是验收测试采用given-when-then语法（中文可以用假如-当-那么），便于阅读理解。验收测试也可以用脑图，文字来描述，甚至DDD不反对采用单元测试框架写验收测试，只要领域专家能够阅读并理解写出的验收测试。  
这一步做完后我们的模型图形其实没什么变化，但是现在，开发能够更充分的了解业务了。  
**使用子域处理遗留系统**  
我们代码不是在真空里运行，它们免不了会跟一些遗留系统打交道，这些遗留系统的边界并不清晰。因此我们会将遗留系统放到一个子域里，把它们的问题放到我们的设计之外。这一步做完后我们的图案与之前没有本质上的区别，无非是多了一点子域。

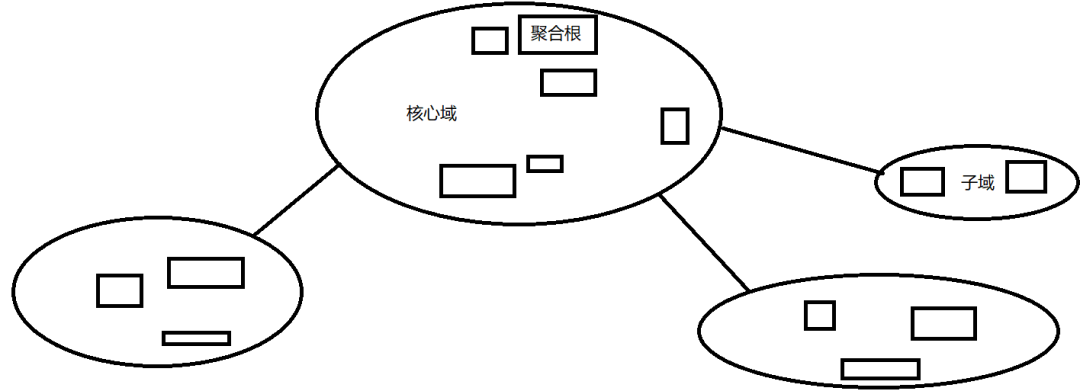


**使用上下文映射来集成多个限界上下文**  
上下文映射是两个限界上下文之间的连线，表示了这两个概念之间的关系，也表示这这两个概念的通用语言的翻译。通常来说，不同的限界上下文是不同的团队在维护，那么此时它也代表着两个团队之间合作的关系。  
我们常见的映射关系是RPC接口。然而在领域设计里，限界上下文之间使用RPC是有风险的方案，因为会承受网络风险，还意味着两个限界上下文之间存在紧耦合。如果系统A阻塞请求系统B，B又请求C，就很容易导致集成火车事故：火车里某一节车厢有问题就会变成整列火车的问题。  
最好的限界上下文映射关系采用事件的订阅，但是这要求领域专家在设计的时候就考虑不同领域之间通知的延迟对于业务的影响，以及如何消除影响。如果不采用DDD的方式，领域专家通常无法意识到领域之间的同步成本，技术人员也很容易一头撞进集成火车里。这就是我说的：DDD的目标是找到边界和促进学习知识，不仅仅是开发学习业务，领域专家也是在学习系统的边界与设计。

**战术设计**  
在战术设计阶段包括如下设计：

1. 把一些实体和值对象放一起，称为聚合。
2. 利用领域事件通知相关系统。

**聚合怎么设计**  
一个限界上下文里通常有多个聚合，聚合逻辑上是相对独立的。怎么理解聚合的概念呢？在DDD实践中，聚合是事务的边界；聚合之间并不保证事务，只能用最终一致性。任何需要事务保护的逻辑都应该在一个聚合内。在限界上下文里，将其他聚合能力整合在一起对外提供能力的聚合，被称为聚合根；其他聚合也被称为实体。  
此外，一个限界上下文里还有值对象，它也代表了某种相对独立的概念。怎么区分实体和值对象呢？这取决于业务。如果一个名词，具有多种动词去操作它，那么它应该是一个实体；如果一个名词，在系统里只是被传递而没有业务逻辑，那么它就是值对象。  
由于聚合是事务的边界，那么每个聚合在设计阶段，最重要的是找到业务的不变性，也就是说，在事务提交前后，数据的约束条件。比如说，你在知乎对一条回答点赞，那么这条回答的点赞数量必须立刻多1，那么点赞的动作和点赞的计数，就应当在一个聚合内。  
领域专家必然希望任何事情都能在触发后立刻完成，所以在沟通的过程中要不断质疑，如果不实时地做一件事，会不会有问题。甚至可以用一个夸张到显然无法接受的时间长度来质疑，以促成领域专家对此认真思考。  
在聚合被设计出来以后，我们的模型图看起来会是这样的：



**领域事件怎么设计**  
我们说聚合之间要采用最终一致性，而通常的做法是采用领域事件实现最终一致性。领域事件的名称应该采用通用语言命名，才能符合领域专家的心智。完整的时间名词应该是名词和动词构成的，动词应该是过去时。领域事件的名字和属性应该能够完整描述这个事件的含义。  
事件里通常至少包含业务动作和其业务参数，也可以增加更多的下游关注的事件信息，避免下游为了完成处理还需查询。  
领域事件会持久保存在专门的数据表中，用来表示领域事件的因果关系。  
有一种专门的存储方式是事件溯源，它不需要存储数据当前是什么，而是从历史事件中按顺序应用重建，得到当前的数据。这样写入时的成本只有校验后持久化，也没有增加和删除的能力。如果事件很多，性能问题很大，也可以加上缓存和快照，优化性能。这种方案通常会与CQRS方案一起做。