**[DDD不是架构设计方法](https://www.cnblogs.com/Leo_wl/p/10995361.html)**

DDD不是架构设计方法

## **[一文读懂DDD](https://www.cnblogs.com/xiguain/p/10939679.html)**

2019-05-28 19:18 by 春哥大魔王, 940

## **何为DDD**

DDD不是架构设计方法，不能把每个设计细节具象化，DDD是一套体系，决定了其开放性，体系中可以用任何一种方法来解决这些问题，但是如果一些关键问题没有具体方案落地，可能让团队无所适从。

有的小伙伴觉得DDD太虚了，具体在我们进行业务代码编写落地中DDD主要解决什么问题呢？

总结起来说主要目的有两点：

* 建立业务术语，统一PM/RD/QA需求沟通术语。
* 梳理业务边界，将业务领域逻辑内聚。

## **搞定DDD要解决的问题**

* 如何进行领域建模
* 如何识别Bounded Context
* 如何在战术层面寻找对象

## **DDD术语**

**战略建模**

* 界限上下文（Bounded Context）
* 上下文映射图（Context Mapping）

**战术建模**

* 聚合-Aggregate
* 实体-Entity
* 值对象-Value Objects
* 资源库-Repository
* 领域服务-Domain Services
* 领域事件-Domain Events
* 模块-Modules

## **Bound Context（BC）**

领域中的BC被封装为高内聚的模块，这种特性让DDD对架构并没有太大侵入性。架构可以应用于领域内部的结构，也可以包围着领域模型，系统中可以采用多种风格的架构。

DDD的战略设计上提出了BC（Bounded Context，界限上下文）。UL（Ubiquitous Language，通用语言）是团队的共享语言，只要是团队的一员，就需要使用UL，可以保证各个概念在各自上下文中无歧义。BC和UL是DDD的两大支柱，相辅相成。

一个业务领域划分成多个BC，BC之间通过Context Map进行集成，BC是一个显示边界，领域模型在这个边界之内，领域模型是关于某个特定业务领域的软件模型，领域模型通过对象模型来实现，这些对象同时包含了数据和行为，并表达了准确的业务含义。  
广义上讲，领域是一个组织所做的事情及其中所包含的一切，表示整个业务系统，领域表示应该为整个业务系统创建统一的，内聚的全功能模型，领域模型存在于BC内。

通过BC隔离系统复杂性，将复杂度内聚于边界之内。

一个大型系统的领域模型完全统一是不可行的，也不是一种经济有效的方式。任何一个大型项目都会存在多个模型，不同模型代码组织在一起软件可能会出现bug，同时更加不可靠并且难以理解。团队之间沟通也会变的混乱。

当划分为多个模型之后，在模型之内，团队可以自由工作，直到自己的界限并且恪守界限。所以需要确保模型纯洁，一致和统一。  
所以需要明确定义模型应用上下文，根据团队组织或者软件系统或者物理表现来设置模型边界。

## **Context Map 上下文图**

多个系统之间会发生关系，存在交互，需要在项目中创建一个所有模型上下文的全局视图，减少混乱。一般通过Context Map表示系统关系总体视图。  


U表示上游（Upstream）的被依赖方，D表示下游（Downstream）的依赖方。防腐层（ACL）放在下游，将上游的消息转化为下游的领域模型。

Context Map通过下面几种方式表征界限上下文之间的关系：

* 共享内核-Shared Kernel
* 客户/供应商-Customer/Supplier
* 追随者-Conformist
* 防腐层-Anticorruption Layer
* 公开主机服务-Open Host Service
* 各行其道-Separate Way

**共享内核-Shared Kernel**

当不同团队开发一些紧密相关的应用程序时，团队之间需要进行协调，通常可以将两个团队共享的子集剥离出来形成共享内核（Shared Kernel），双方进行持续集成（Continuous Integration）。共享内核（Shared Kernel）是业务领域中公共的部分，同时也是团队间容易达成且必须达成共识的领域部分。

**客户/供应商-Customer/Supplier**

不同系统之间存在依赖关系时，下游系统依赖上游系统，下游系统是客户，上游系统是供应商，双方协定好需求，由上游系统完成模型的构建和开发，并交付给下游系统使用，之后进行联调、测试。这种模式建立在团队之间友好合作和支持的情况下。  
当两个具有上游/下游关系的团队不归同一个管理者指挥时，Customer/Supplier这样的合作模式就不会奏效。勉强应用这种模式会给下游团队带来麻烦。

**追随者-Conformist**

当两个开发团队具有上/下游关系时，如果上游团队没有动机来满足下游团队的需求，那么下游团队将无能为力。出于利他主义的考虑，上游开发人员可能会做出承诺，但他们可能不会履行承诺。下游团队出于良好的意愿会相信这些承诺，从而根据一些永远不会实现的特性来制定计划。下游项目只能被搁置．直到团队最终学会利用现有条件自力更生为止。下游团队不会得到根据他们的需求而量身定做的接口。  
这时候“客户/供应商”模式就不凑效了，那么下游系统只能去追随上游系统，下游系统严格遵从上游系统的模型，简化集成。  
通过严格遵从上游团队的模型，可以消除在 BC之间进行转换的复杂性。尽管这会限制下游设计人员的风格，而且可能不会得到理想的应用程序模型，但选择 Conformist模式可以极大地简化集成。此外，这样还可以与供应商团队共享一种 UL。供应商处于驾驶者的位置上，因此最好使他们能够容易沟通。

**防腐层-Anticorruption Layer**

前面介绍了在两个BC之间集成时可以进行的各种合作，从高度合作的 Shared Kernel模式或 Customer/Supplier Team到单方面的Conformist模式。如果是一种更悲观的关系，假设一个团队既不可能与另一个团队合作也无法利用他们的设计时，该如何应对。  
这时候我们需要使用防腐层（Anticorruption Layer）模式将上游系统的影响降低。

**公开主机服务-Open Host Service**

当一个子系统必须与大量其他系统进行集成时，为每个集成都定制一个转换层可能会减慢团队的工作速度。如果一个子系统有某种内聚性，那么或许可以把它描述为一组 Service,这组 Service满足了其他子系统的公共需求。  
公开主机服务（Open Host Service）能够允许系统将一组Service公开出去公其他系统访问。定义一个协议，把你的子系统作为一组 Service供其他系统访问。开放这个协议，以便所有需要与你的子系统集成的人都可以使用它。当有新的集成需求时，就增强并扩展这个协议，但个别团队的特殊需求除外。

**各行其道-Separate Way**

当两个系统之间的关系并非必不可少时，两者完全可以彼此独立，各自独立建模，独立发展，互不影响。

## **领域事件**

领域专家所关心的发生在领域中的一些事件。将领域中所发生的活动建模成一系列的离散事件。每个事件都用领域对象来表示...领域事件是领域模型的组成部分，表示领域中所发生的事情。

“重要的事件肯定会在系统其它地方引起反应，因此理解为什么会有这些反应同样也很重要。”

当然领域事件并不是DDD所必须的。

一个领域事件可以理解为是发生在一个特定领域中的事件，是你希望在同一个领域中其他部分知道并产生后续动作的事件。但是并不是所有发生过的事情都可以成为领域事件。一个领域事件必须对业务有价值，有助于形成完整的业务闭环，也即一个领域事件将导致进一步的业务操作。

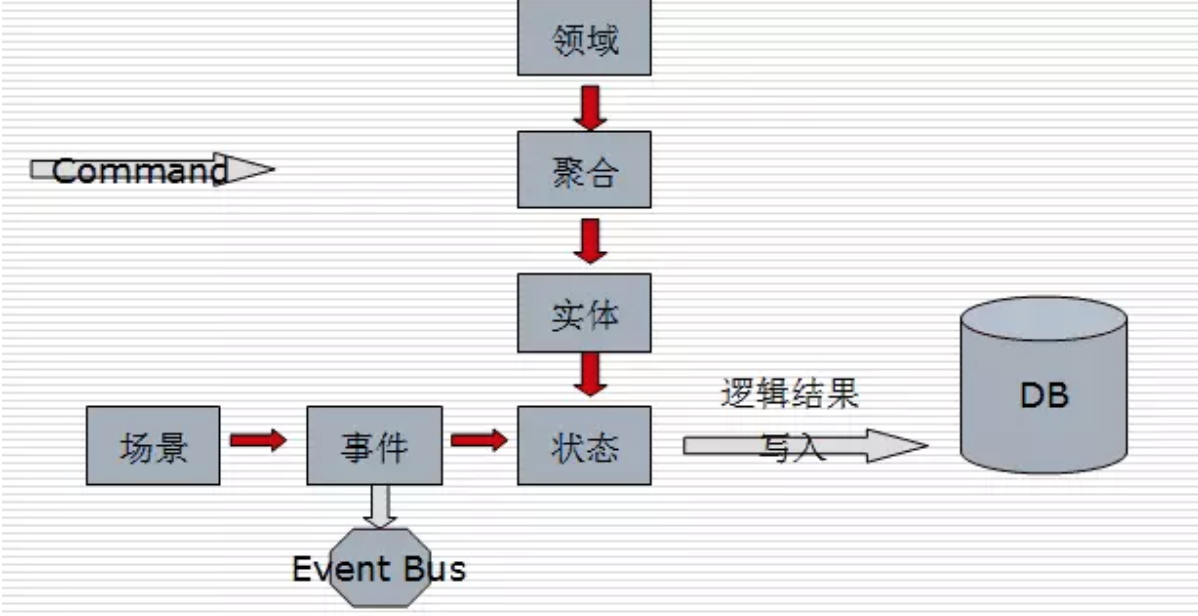
领域事件可以是业务流程的一个步骤，例如订单提交，客户付费100元，订单完工等。领域事件也可以是定时发生的事情，例如每晚对账完成。或者是一个事件发生后引发的后续动作，例如客户输错密码三次后发生锁定账户的事件。

领域事件也是一种基于事件的架构（EDA）。事件架构的好处可以把处理的流程解耦，实现系统可扩展性，提高主业务流程的内聚性。

如果改为事件驱动模式，把订单提交后触发一个事件，在订单保存后，触发订单提交事件。通知和后续的各种服务动作可以通过订阅这个事件，在自己的实现空间内实现对应的逻辑，这样就把订单提交和后续其他非主要活动从订单提交业务中剥离，实现了订单提交业务高内聚和低耦合性。

首先是解决领域的聚合性问题。DDD中的聚合有一个原则是，在单个事务中，只允许对一个聚合对象进行修改，由此产生的其他改变必须在单独的事务中完成。如果一个业务跨多个聚合对象，领域事件会是一个不错的工具来解决这个问题。通过领域事件的方式可以达到各个组件之间的数据一致性，通过最终一致性取代事务一致性。

其次领域事件也是一种领域分析的工具，有时从领域专家的话中，我们看不出领域事件的迹象，但是业务需求依然有可能需要领域事件。动态流的事件模型加上结合DDD的聚合实体状态和BC，可以有效进行领域建模。



领域事件可以通过观察者模式和订阅模式进行实现。比较常见的实现方式是事件总线（Event Bus）。

## **事件风暴**

事件风暴是一项团队活动，旨在通过领域事件识别出聚合根，进而划分微服务的限界上下文。在活动中，团队先通过头脑风暴的形式罗列出领域中所有的领域事件，整合之后形成最终的领域事件集合，然后对于每一个事件，标注出导致该事件的命令（Command），再然后为每个事件标注出命令发起方的角色，命令可以是用户发起，也可以是第三方系统调用或者是定时器触发等。最后对事件进行分类整理出聚合根以及限界上下文。

**举个例子**

在我们的一次产品的重构活动中也采用了事件风暴方法。系统代码维护了10几年，代码中存在大量的“坏味道”：重复代码，过长函数，过大的类，过长的参数列表，发散式变化，霰弹式修改，镀金问题，注释不清等问题。实际研发过程中也是经常出现一点改动都可能会引起不可预测的结果，重构势在必行。  
但是在重构过程中，也没有人可以说清楚现有系统的逻辑，如何重构成为了一个难题。重构过程我们引入了咨询公司给我们的方法，采用了事件风暴的办法，通过对领域中所发生的事情（也就是领域事件）来探索这个领域，并且使用便签来描述领域中的事件，这些便签会沿着时间轴贴到一个很大的建模面板上。  
举例来说，能够引发事件的事情包括用户行为、外部系统所发生的事情以及时间的流逝。事件也有助于找到领域的边界，对术语的不同阐述可能就意味着存在边界。

准备工作，四色贴纸：  
橙色：事件，某个动作的结果，以“XX已XX”的方式表示，比如“用户信息已查询”  
蓝色：属性，事件相关的输入、输出数据等  
黄色：命令，某个动作，比如“查找用户信息”  
绿色：实体，命令的触发者

* 开始梳理业务，将结果贴到白版上
* 继续深入梳理，将整个过程的模型、关键数据等梳理出来，贴在白板上

确定重构指导思路，执行重构动作，重构的同时引入单元测试保障重构的质量

## **实体和值对象**

实体不仅需要知道它是什么？而且还需要知道它是哪个？而值对象只需要知道它是什么？

实体：许多对象不是由它们的属性来定义，而是通过一系列的连续性（continuity）和标识（identity）来从根本上定义的。只要一个对象在生命周期中能够保持连续性，并且独立于它的属性（即使这些属性对系统用户非常重要），那它就是一个实体。

值对象：当你只关心某个对象的属性时，该对象便可作为一个值对象。为其添加有意义的属性，并赋予它相应的行为。我们需要将值对象看成不变对象，不要给它任何身份标识，还应该尽量避免像实体对象一样的复杂性。

实体对象相对容易理解，我们常见的类的都可以看成是实体对象。值对象在DDD中相对而言是难以理解并且容易误用的。

为什么需要使用值对象，书中给了一个解释：

使用不变的值对象使得我们做更少的职责假设

使用值对象在不同的BC中进行数据交换，可以避免不同BC对实体对象的状态变更而引发的数据依赖关系，实现最小化的集成。

值类型用于度量和描述事物，DDD中建议应尽量使用值对象来建模而不是实体对象，因为值对象非常容易地对值对象进行创建、测试、使用、优化和维护。

**领域服务**

领域中的服务表示一个无状态的操作,它用于实现特定于某个领域的任务。  
当某个揉作不适合放在聚合和值对象上时，最好的方式便是使用领域服务了。有时我们傾向于使用聚合根上的静态方法来实现这些这些操作，但是在 DDD中,这是一种坏味道。

《实现领域驱动设计》书中给出了一个例子，对User进行认证的例子。例子中给出的需求是：

* 系统必须对User进行认证，并且只有当Tenant处于激活状态时候才能对User进行认证。
* 必须对密码进行加密，并且不能使用明文密码

对以上的需求，我们可以把认证的方法写在User类或者Tenant类中，不过对于以上解决方案，似乎都给模型带来了太多的问题。

对于后一种方案， 我们必须从以下回种解决办法中选择一种：

在Tenant中处理对密码的加密，然后将加密后的密码传给User。这种方法违背了单一职责原则

1. 由于一个User必须保征对密码的加密，它可能已经知道了一些加密信息。如果是这样，我们可以在User上创建一个方法，该方法对明文密码进行认证。但是在这种方式下，认证过程变成了Tenant上的Facade。而实际的认证 功能全在User上。另外User上的认证方法必须声明为Protected,以防止外界 客户端对认证方法的直接调用。
2. Tenant依赖于User对密码进行加密，然后将加密后的密码与原有密码进行匹配。这种方法似乎在对象协作之间增加了额外的步骤。此时,Tenant依然需 要知道认证细节。

让客户端对密码进行加密。然后将其传给Tenant，这样导致的问题在于客户端承载了它本不应该有的职责。

UserDescriptor userDescriptor =

DomainRegistry

.authenticationService()

.authenticate(tenantID,userName,password);

## **模块**

在DDD中，模块表示了一个命名的容器，用于存放领域中内聚在一起的类。

模块应该包含一組具有高内聚性的概念集合.这样做的好处是可以在不同的模块之间实现松耦合。否则，我们应该修改模型以重新划分这些概念。……由于模块名是UL的一部分，模块名应该反映出它们在领域中的概念。[Evans]

模块的设计是基于领域模型的，要符合通用语言的表述。其次，模块的设计要符合高内聚低耦合的设计思想。



## **模块和BC的关系**

模块与子域和限界上下文并不是一致的概念，模块也是一种独立的建模方法。对于何时应该对领域模型进行分离，何时将领域模型建模成一个整体，应该仔细地思考与对待。有时通用语言可以很好地帮助我们做出正确的选择。但是另外的时候，其中的术语将变得非常含糊。在这种情况下,我们并不清楚如何划分上下文边界。此时,我们可以首先将它们放在一起，使用模块来对模型进行划分，面不是限界上下文。

但是，这并不意味着我们就应该限制对限界上下文的创建。我们应该通过通用语言的需求来划分模型边界。但限界上下文不是用来代替模块的。使用摸块的目的在于组织那些内聚在一起的领域对象，对于那些内聚性不强或者没有内聚性的领域对象来说,我们应该将它们划分在不同的模块中。

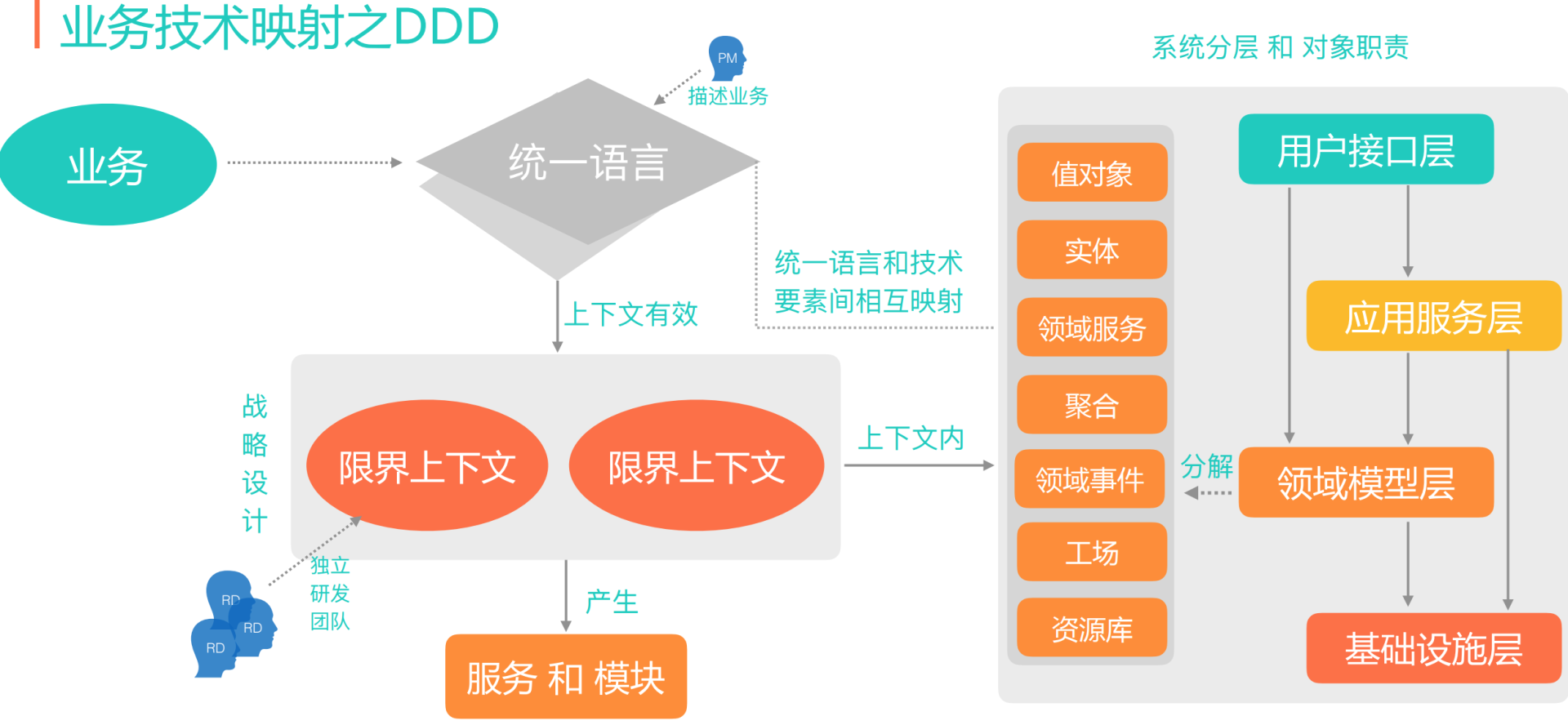
## **集成BC（界限上下文）**

一个项目中会存在多个BC，业务需要对它们进行集成。有多种直接的方法进行集成。最简单的方式就是一个BC中暴露API，然后在另外一个BC中通过RPC进行调用。

另外我们也可以通过消息机制进行集成，系统通过消息队列或者发布-订阅机制进行通讯。

第三种方式是通过使用RESTful的方式进行集成。当然，还存在有其他的集成方式。

## **结尾一张图**

如果你还是云里雾里，参考这张图：  


分类: **[[12]Architecture](https://www.cnblogs.com/Leo_wl/category/225687.html" \t "https://www.cnblogs.com/Leo_wl/p/_blank)**