链滴

领域驱动设计（DDD）- 多研究些架构，少谈些框架

作者：ibut

原文链接：https://ld246.com/article/1534049567832

来源网站：链滴

至少 30 年以前，一些软件设计人员就已经意识到领域建模和设计的重要性，并形成一种思潮，Eric Evans 将其定义为领域驱动设计（Domain-Driven Design，简称 DDD）。在互联网开发“小步快，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行业软件开发中所面临的问题。本文就先来讲一下这些问题，然后再尝试在实践中用 DDD 的思想来决这些问题。
<h3 id="过度耦合">过度耦合</h3>
在业务初期，功能大都非常简单，普通的 CRUD 就基本能满足要求，此时系统是清晰的。但随产品的不断迭代和演化，业务逻辑变得越来越复杂，我们的系统也越来越冗杂。各个模块之间彼此关，甚至到后期连相应的开发者都很难说清模块的具体功能和意图到底是啥。这就会导致在想要修改一功能时，要追溯到这个功能需要修改的点就要很长时间，更别提修改带来的不可预知的影响面。 比如图所示:

订单服务中提供了查询、创建订单相关的接口，也提供了订单评价、支付的接口。同时订单表是个大，包含了非常多字段。我们在维护代码时，将会导致牵一发而动全身，很可能原本我们只是想改下评相关的功能，却影响到了创建订单的核心流程。虽然我们可以通过测试来保证功能的完备性，但当我在订单领域有大量需求同时并行开发时将会出现改动重叠、恶性循环、疲于奔命修改各种问题的局面而且大量的全量回归会给测试带来不可接受的灾难。（ 模块内聚度低、高耦合）
但现实中绝大部分公司都是这样一个状态，然后一般他们的解决方案是不断的重构系统，让系统设计随着业务成长也进行不断的演进。通过重构出一些独立的类来存放某些通用的逻辑解决混乱问题但是我们很难给它一个业务上的含义，只能以技术纬度进行描述，那么带来的问题就是其他人接手这代码的时候不知道这个的含义或者只能通过修改通用逻辑来达到某些需求。
事实上，你可能意识到问题之所在。在解决现实问题时，我们会将问题映射到脑海中的概念模型在模型中解决问题，再将解决方案转换为实际的代码。上述问题在于我们解决了设计到代码之间的重，但提炼出来的设计模型，并不具有实际的业务含义，这就导致在开发新需求时，其他同学并不能很然地将业务问题映射到该设计模型。设计似乎变成了重构者的自娱自乐，代码继续腐败，重新重构…无休止的循环。
用 DDD 则可以很好地解决领域模型到设计模型的同步、演化，最后再将反映了领域的设计模型为实际的代码。
注：模型是我们解决实际问题所抽象出来的概念模型，领域模型则表达与业务相关的事实；设计型则描述了所要构建的系统。
<h3 id="贫血领域对象">贫血领域对象</h3>
贫血领域对象（Anemic Domain Object）是指仅用作数据载体，而没有行为和动作的领域对象
实际上，领域模型本身也不是一个陌生的单词，说直白点，在早期开发中，领域模型就是数据库计。因为你想：我们做传统项目的流程或者说包括现在我们做项目的流程，都是首先讨论需求，然后数据库建模，在需求逐步确定的过程不断的去变更数据库的设计，接着我们在项目开发阶段，发现有关系没有建、有些字段少了、有些表结构设计不合理，又在不断的去调整设计，最后上线。在传统项中，数据库是整个项目的根本，数据模型出来以后后续的开发都是围绕着数据展开，然后形成如下的个架构 :

很显然，这其中存在的问题如下：

Service 层很重，所有逻辑处理基本都放在 service 层。

POJO 作为 Service 层非常重要的一个实体，会因为不同场景的需求做不同的变化和组合，就会成 POJO 的几种不同模型(失血、贫血、充血)，以此用来形容领域模型太胖或者太瘦。

随着业务变得复杂以后，包括数据结构的变化，那么各个模块就需要进行修改，原本清晰的系统过不断的演 化变得复杂、冗余、耦合度高，后果就非常严重。

我们试想一下如果一个软件产品不依赖数据库存储设备，那我们怎么去设计这个软件呢？如果没了数据存储，那么我们的领域模型就得基于程序本身来设计。那这个就是 DDD 需要去考虑的问题。
<h3 id="DDD中的基本概念">DDD 中的基本概念</h3>
<h4 id="实体-Entity-">实体（Entity）</h4>
当一个对象由其标识(而不是属性)区分时，这种对象称为实体(Entity)。比如当两个对象的标识不时，即使两个对象的其他属性全都相同，我们也认为他们是两个完全不同的实体。
<h4 id="值对象-Value-Object-">值对象（Value Object）</h4>
当一个对象用于对事物进行描述而没有唯一标识时，那么它被称作值对象。因为在领域中并不是何时候一个事物都需要有一个唯一的标识，也就是说我们并不关心具体是哪个事物，只关心这个事物什么。比如下单流程中，对于配送地址来说，只要是地址信息相同，我们就认为是同一个配送地址。于不具有唯一标示，我们也不能说"这一个"值对象或者"那一个"值对象。
<h4 id="领域服务-Domain-Service-">领域服务（Domain Service）</h4>
一些重要的领域行为或操作，它们不太适合建模为实体对象或者值对象，它们本质上只是一些操，并不是具体的事物，另一方面这些操作往往又会涉及到多个领域对象的操作，它们只负责来协调这领域对象完成操作而已，那么我们可以归类它们为领域服务。它实现了全部业务逻辑并且通过各种校手段保证业务的正确性。同时呢，它也能避免在应用层出现领域逻辑。理解起来，领域服务有点 facae 的味道。
<h4 id="聚合及聚合根-Aggregate-Aggregate-Root-">聚合及聚合根（Aggregate，Aggregate Rot）</h4>
聚合是通过定义领域对象之间清晰的所属关系以及边界来实现领域模型的内聚，以此来避免形成综复杂的、难以维护的对象关系网。聚合定义了一组具有内聚关系的相关领域对象的集合，我们可以聚合看作是一个修改数据的单元。
聚合根属于实体对象，它是领域对象中一个高度内聚的核心对象。(聚合根具有全局的唯一标识而实体只有在聚合内部有唯一的本地标识，值对象没有唯一标识，不存在这个值对象或那个值对象的法)
若一个聚合仅有一个实体，那这个实体就是聚合根；但要有多个实体，我们就要思考聚合内哪个象有独立存在的意义且可以和外部领域直接进行交互。
<h4 id="工厂-Factory-">工厂（Factory）</h4>
DDD 中的工厂也是一种封装思想的体现。引入工厂的原因是：有时创建一个领域对象是一件相比较复杂的事情，而不是简单的 new 操作。工厂的作用是隐藏创建对象的细节。事实上大部分情况，领域对象的创建都不会相对太复杂，故我们仅需使用简单的构造函数创建对象就可以。隐藏创建对细节的好处是显而易见的，这样就可以不会让领域层的业务逻辑泄露到应用层，同时也减轻应用层负，它只要简单调用领域工厂来创建出期望的对象就可以了。
<h4 id="仓储-Repository-">仓储（Repository）</h4>
资源仓储封装了基础设施来提供查询和持久化聚合操作。这样能够让我们始终关注在模型层面，对象的存储和访问都委托给资源库来完成。它不是数据库的封装，而是领域层与基础设施之间的桥梁DDD 关心的是领域内的模型，而不是数据库的操作。
<h3 id="如何实践DDD">如何实践 DDD</h3>
DDD 概念理解起来有点抽象, 这个有点像设计模式，感觉很有用，但是自己开发的时候又不知道么应用到代码里面，或者生搬硬套后自己看起来都很别扭，那么接下来我们就以一个简单的转盘抽奖例来详细介绍我们如何通过 DDD 来解构一个中型的基于微服务架构的系统，从而做到系统的高内聚低耦合。
首先看下抽奖系统的大致需求：
运营——可以配置一个抽奖活动，该活动面向一个特定的用户群体，并针对一个用户群体发放一不同类型的奖品（优惠券，激活码，实物奖品等）。
设计领域模型的一般步骤如下：

根据需求划分出初步的领域和限界上下文，以及上下文之间的关系；

进一步分析每个上下文内部，识别出哪些是实体，哪些是值对象；

对实体、值对象进行关联和聚合，划分出聚合的范畴和聚合根；

为聚合根设计仓储，并思考实体或值对象的创建方式；

在工程中实践领域模型，并在实践中检验模型的合理性，倒推模型中不足的地方并重构。

<h3 id="战略建模">战略建模</h3>
战略和战术设计是站在 DDD 的角度进行划分。战略设计侧重于高层次、宏观上去划分和集成限上下文，而战术设计则关注更具体使用建模工具来细化上下文。
<h3 id="领域">领域</h3>
现实世界中，领域包含了问题域和解系统。一般认为软件是对现实世界的部分模拟。在 DDD 中解系统可以映射为一个个限界上下文，限界上下文就是软件对于问题域的一个特定的、有限的解决方。
<h3 id="限界上下文">限界上下文</h3>
<blockquote>
一个由显示边界限定的特定职责。领域模型便存在于这个边界之内。在边界内，每一个模型概念包括它的属性和操作，都具有特殊的含义。
</blockquote>
一个给定的业务领域会包含多个限界上下文，想与一个限界上下文沟通，则需要通过显示边界进通信。系统通过确定的限界上下文来进行解耦，而每一个上下文内部紧密组织，职责明确，具有较高内聚性。
一个很形象的隐喻：细胞质所以能够存在，是因为细胞膜限定了什么在细胞内，什么在细胞外，且确定了什么物质可以通过细胞膜。
<h3 id="划分限界上下文">划分限界上下文</h3>
划分限界上下文，不管是 Eric Evans 还是 Vaughn Vernon，在他们的大作里都没有怎么提及。/p>
显然我们不应该按技术架构或者开发任务来创建限界上下文，应该按照语义的边界来考虑。
我们的实践是，考虑产品所讲的通用语言，从中提取一些术语称之为概念对象，寻找对象之间的系；或者从需求里提取一些动词，观察动词和对象之间的关系；我们将紧耦合的各自圈在一起，观察们内在的联系，从而形成对应的界限上下文。形成之后，我们可以尝试用语言来描述下界限上下文的责，看它是否清晰、准确、简洁和完整。简言之，限界上下文应该从需求出发，按领域划分。
<h4 id="针对功能层面划分边界">针对功能层面划分边界</h4>
前文提到，我们的用户划分为运营和用户。其中，运营对抽奖活动的配置十分复杂但相对低频。户对这些抽奖活动配置的使用是高频次且无感知的。根据这样的业务特点，我们首先将抽奖平台划分为 C 端抽奖和 M 端抽奖管理平台两个子域，让两者完全解耦。
在确认了 M 端领域和 C 端的限界上下文后，我们再对各 自上下文内部进行限界上下文的划分，下来以 C 端用户为例来划分界限上下文。
用户-通过活动页面参与不同类型的抽奖活动，产品的需求概述如下：
<blockquote>

抽奖资格(什么情况下会有抽奖机会、抽奖次数、抽 奖的活动起始时间) 。

</blockquote>
<blockquote>
<ol start="2">

抽奖的奖品(实物、优惠券、理财金、购物卡...) 。

</blockquote>
<blockquote>
<ol start="3">
奖品自身的配置，概率、库存、某些奖品在有限的概率下还只能被限制抽到多少次等。

</blockquote>
<blockquote>
<ol start="4">
风控对接, 防止恶意薅羊毛，能够限制用户参与抽奖的频率。。

</blockquote>
首先，抽奖上下文作为整个领域的核心，承担着用户抽奖的核心业务，抽奖中包含了奖品和用户体的概念。

在设计初期，我们曾经考虑划分出抽奖和发奖两个领域，前者负责选奖，后者负责将选中的奖品放出去。但在实际开发过程中，我们发现这两部分的逻辑紧密连接，难以拆分。并且单纯的发奖逻辑够简单，仅仅是调用第三方服务进行发奖，不足以独立出来成为一个领域。
对于活动的限制，我们定义了活动准入的通用语言，将活动开始/结束时间，活动可参与次数等限制件都收拢到活动准入上下文中。

对于抽奖的奖品库存量，由于库存的行为与奖品本身相对解耦，库存关注点更多是库存内容的核，且库存本身具备通用性，可以被奖品之外的内容使用，因此我们定义了独立的库存上下文。
由于 C 端存在一些刷单行为，我们根据产品需求定义了风控上下文，用于对活动进行风控。
最后，活动准入、风控、抽奖等领域都涉及到一些次数的限制，因此我们定义了计数上下文。</
>
可以看到，通过 DDD 的限界上下文划分，我们界定出抽奖、活动准入、风控、计数、库存等五上下文，每个上下文在系统中都高度内聚。
<h3 id="上下文映射图">上下文映射图</h3>
在进行上下文划分之后，我们还需要进一步梳理上下文之间的关系。
<blockquote>
康威（梅尔·康威）定律
任何组织在设计一套系统（广义概念上的系统）时，所交付的设计方案在结构上都与该组织的沟通结保持一致。 （是不是和微服务理念也一样:joy: ）
</blockquote>
康威定律告诉我们，系统结构应尽量的与组织结构保持一致。这里，我们认为团队结构（无论是部组织还是团队间组织）就是组织结构，限界上下文就是系统的业务结构。因此，团队结构应该和限上下文保持一致。
梳理清楚上下文之间的关系，从团队内部的关系来看，有如下好处：

任务更好拆分，一个开发人员可以全身心的投入到相关的一个单独的上下文中；

沟通更加顺畅，一个上下文可以明确自己对其他上下文的依赖关系，从而使得团队内开发直接更的对接。

从团队间的关系来看，明确的上下文关系能够带来如下帮助：

每个团队在它的上下文中能够更加明确自己领域内的概念，因为上下文是领域的解系统；

对于限界上下文之间发生交互，团队与上下文的一致性，能够保证我们明确对接的团队和依赖的下游。

限界上下文之间的映射关系

合作关系（Partnership）：两个上下文紧密合作的关系，一荣俱荣，一损俱损。

共享内核（Shared Kernel）：两个上下文依赖部分共享的模型。

客户方-供应方开发（Customer-Supplier Development）：上下文之间有组织的上下游依赖。/p>

遵奉者（Conformist）：下游上下文只能盲目依赖上游上下文。

防腐层（Anticorruption Layer）：一个上下文通过一些适配和转换与另一个上下文交互。

开放主机服务（Open Host Service）：定义一种协议来让其他上下文来对本上下文进行访问。

发布语言（Published Language）：通常与 OHS 一起使用，用于定义开放主机的协议。

大泥球（Big Ball of Mud）：混杂在一起的上下文关系，边界 不清晰。

另谋他路（SeparateWay）：两个完全没有任何联系的上下文。
上文定义了上下文映射间的关系，经过我们的反复斟酌，抽奖平台上下文的映射关系图如下：

<img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https://b3logfile.com/pipe/pipe/ibut/ibut/9f99709d876f4b9fa3d2ff001507db3f.png?imageView2/2/nterlace/1/format/jpg">
由于抽奖，风控，活动准入，库存，计数五个上下文都处在抽奖领域的内部，所以它们之间符合“一俱荣，一损俱损”的合作关系（PartnerShip，简称 PS）。
同时，抽奖上下文在进行发券动作时，会依赖券码、平台券、外卖券三个上下文。抽奖上下文通防腐层（Anticorruption Layer，ACL）对三个上下文进行了隔离，而三个券上下文通过开放主机服（Open Host Service）作为发布语言（Published Language）对抽奖上下文提供访问机制。
通过上下文映射关系，我们明确的限制了限界上下文的耦合性，即在抽奖平台中，无论是上下文部交互（合作关系）还是与外部上下文交互（防腐层），耦合度都限定在数据耦合（Data Coupling的层级。
<h3 id="战术建模--细化上下文">战术建模——细化上下文</h3>
抽奖平台的核心上下文是抽奖上下文，接下来介绍下我们对抽奖上下文的建模

在抽奖上下文中，我们通过抽奖(DrawLottery)这个聚合根来控制抽奖行为，可以看到，一个抽奖包了抽奖 ID（LotteryId）以及多个奖池（AwardPool），而一个奖池针对一个特定的用户群体（UserGoup）设置了多个奖品（Award）。
另外，在抽奖领域中，我们还会使用抽奖结果（SendResult）作为输出信息，使用用户领奖记（UserLotteryLog）作为领奖凭据和存根。
谨慎使用值对象
在实践中，我们发现虽然一些领域对象符合值对象的概念，但是随着业务的变动，很多原有的定会发生变更，值对象可能需要在业务意义具有唯一标识，而对这类值对象的重构往往需要较高成本。此在特定的情况下，我们也要根据实际情况来权衡领域对象的选型。
<h4 id="DDD工程实现-代码设计">DDD 工程实现-代码设计</h4>
在对上下文进行细化后，我们开始在工程中真正落地 DDD。
<h3 id="模块">模块</h3>
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.*}，这样的组结构能够明确的将一个上下文限定在包的内部。

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、防腐层等组织方定义的。

<h3 id="领域对象">领域对象</h3>
前文提到，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。这里我们用之前定义抽奖（DrawLottery）聚合根和奖池（AwardPool）值对象来具体说明。
抽奖聚合根持有了抽奖活动的 id 和该活动下的所有可用奖池列表，它的一个最主要的领域功能是根据一个抽奖发生场景（DrawLotteryContext），选择出一个适配的奖池，即 chooseAwardPool 方法。
chooseAwardPool 的逻辑是这样的：DrawLotteryContext 会带有用户抽奖时的场景信息（抽得分或抽奖时所在的城市），DrawLottery 会根据这个场景信息，匹配一个可以给用户发奖的 AwardPool。

在匹配到一个具体的奖池之后，需要确定最后给用户的奖品是什么。
这部分的领域功能在 AwardPool 内。

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。
<h3 id="资源库">资源库</h3>
领域对象需要资源存储，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。在抽奖平台中我们是通过如下的方式组织资源库的。

/b3logfile.com/pipe/pipe/ibut/ibut/92ae2724a2b64c1581c89ee714570281.png?imageView2/
/interlace/1/format/jpg">
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的逻辑（例如缓存更新机制等）。
在抽奖资源库中，我们屏蔽了对底层奖池和奖品的直接访问，而是仅对抽奖的聚合根进行资源管。代码示例中展示了抽奖资源获取的方法（最常见的 Cache Aside Pattern）。
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，代码的可读性和维护性也更强。
<img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https:
/b3logfile.com/pipe/pipe/ibut/ibut/aca4db428c3f4d9db372b525c39ae3ba.png?imageView2/
/interlace/1/format/jpg">
<h3 id="防腐层">防腐层</h3>
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。

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。
在抽奖平台中，我们定义了用户城市信息防腐层
(UserCityInfoFacade)，用于外部的用户城市信息上下文（微服务架构下表现为用户城市信息服）。
以用户信息防腐层举例，它以抽奖请求参数(LotteryContext)为入参，以城市信息(MtCityInfo)输出。
<img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https:
/b3logfile.com/pipe/pipe/ibut/ibut/5a6b7ac295f94033b934394ed28e80ca.png?imageView2/2
interlace/1/format/jpg">
<h3 id="领域服务">领域服务</h3>
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，对其他上文提供交互的接口。
我们以抽奖服务为例（issueLottery），可以看到在省略了一些防御性逻辑（异常处理，空值判等）后，领域服务的逻辑已经足够清晰明了。
<img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https:
/b3logfile.com/pipe/pipe/ibut/ibut/1452b419cda9431689dc8dd524805d99.png?imageView2
2/interlace/1/format/jpg">
<h3 id="数据流转">数据流转</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https:
/b3logfile.com/pipe/pipe/ibut/ibut/d1ccca2a9bfa4ed1961b0aeb6a21f464.png?imageView2/2/
nterlace/1/format/jpg">
在抽奖平台的实践中，我们的数据流转如上图所示。
首先领域的开放服务通过信息传输对象（DTO）来完成与外界的数据交互；在领域内部，我们通领域对象（DO）作为领域内部的数据和行为载体；在资源库内部，我们沿袭了原有的数据库持久化象（PO）进行数据库资源的交互。同时，DTO 与 DO 的转换发生在领域服务内，DO 与 PO 的转换

生在资源库内。
与以往的业务服务相比，当前的编码规范可能多造成了一次数据转换，但每种数据对象职责明确数据流转更加清晰。
<h4 id="上下文集成">上下文集成</h4>
通常集成上下文的手段有多种，常见的手段包括开放领域服务接口、开放 HTTP 服务以及消息发-订阅机制。
在抽奖系统中，我们使用的是开放服务接口进行交互的。最明显的体现是计数上下文，它作为一通用上下文，对抽奖、风控、活动准入等上下文都提供了访问接口。
同时，如果在一个上下文对另一个上下文进行集成时，若需要一定的隔离和适配，可以引入防腐的概念。这一部分的示例可以参考前文的防腐层代码示例。
<h4 id="分离领域">分离领域</h4>
接下来讲解在实施领域模型的过程中，如何应用到系统架构中。
我们采用的微服务架构风格，与 Vernon 在《实现领域驱动设计》并不太一致，更具体差异可阅他的书体会。
如果我们维护一个从前到后的应用系统：
下图中领域服务是使用微服务技术剥离开来，独立部署，对外暴露的只能是服务接口，领域对外露的业务逻辑只能依托于领域服务。而在 Vernon 著作中，并未假定微服务架构风格，因此领域层暴的除了领域服务外，还有聚合、实体和值对象等。此时的应用服务层是比较简单的，获取来自接口层请求参数，调度多个领域服务以实现界面层功能。
<img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https://b3logfile.com/pipe/pipe/ibut/ibut/82346f645f58450d8eae9ffd01a237b8.png?imageView2/2/interlace/1/format/jpg">
随着业务发展，业务系统快速膨胀，我们的系统属于核心时：
应用服务虽然没有领域逻辑，但涉及到了对多个领域服务的编排。当业务规模庞大到一定程度，排本身就富含了业务逻辑（除此之外，应用服务在稳定性、性能上所做的措施也希望统一起来，而非落各处），那么此时应用服务对于外部来说是一个领域服务，整体看起来则是一个独立的限界上下文
此时应用服务对内还属于应用服务，对外已是领域服务的概念，需要将其暴露为微服务。
注：具体的架构实践可按照团队和业务的实际情况来，此处仅为作者自身的业务实践。除分层架外，如 CQRS 架构也是不错的选择
以下是一个示例。我们定义了抽奖、活动准入、风险控制等多个领域服务。在本系统中，我们需集成多个领域服务，为客户端提供一套功能完备的抽奖应用服务。这个应用服务的组织如下：
<img src="https://ld246.com/images/img-loading.svg" alt="imagepng" data-src="https://b3logfile.com/pipe/pipe/ibut/ibut/fb94a8dd7e14457c812bbb7cf1e0fd07.png?imageView2/2interlace/1/format/jpg">
<h3 id="领域驱动的好处">领域驱动的好处</h3>
用 DDD 可以很好的解决领域模型到设计模型的同步、演进最后映射到实际的代码逻辑，总的来，DDD 开发模式有以下几个好处：

DDD 能让我们知道如何抽象出限界上下文以及如何去分而治之。

分而治之：把复杂的大规模软件拆分成若干个子模块，每一个模块都能独立运行和解决相关问题并且分割后各个部分可以组装成为一个整体。

抽象：使用抽象能够精简问题空间，而且问题越小越容易理解，比如说我们要对接支付，抽象的度应该是支付，而不是具体的微信支付还是支付宝支付。

DDD 的限界上下文可以完美匹配微服务的要求。在系统复杂之后，我们都需要用分治来拆解问。一般有两种方式，技术维度和业务维度。技术维度是类似 MVC 这 样，业务维度则是指按业务领域划分系统。 微服务架构更强调从业务维度去做分治来应对系统复杂度， 而 DDD 也是同样的着重业

视角。

来源：领域驱动设计（DDD）在美团点评业务系统的实践