# DDD

## Tips

上帝类 not 一个比较常见的坏习惯是没有尽最大努力为领域行为找到一个归属的实体，轻易定义成领域服务，这容易让领域服务类变成"上帝类"。

Domain Primitive 领域概念显化 把telphone这个字符串变成个对象 对电话的操作集成到这个类里

让我们重新来定义一下Domain Primitive：Domain Primitive是一个在特定领域里，拥有精准定义的、可自我验证的、拥有行为的Value Object。

### BFF

 Backend For Frontend就是类似字节的examapi mineapi, 为了前端而搞的后端

多端应用  
我们在设计 API 时会考虑到不同设备的需求，也就是为不同的设备提供不同的 API，虽然它们可能是实现相同的功能，但因为不同设备的特殊性，它们对服务端的 API 访问也各有其特点，需要区别处理。

服务聚合  
随着微服务的兴起，原本在同一个进程内运行的业务流程被拆分到了不同的服务中。这在增加业务灵活性的同时，也让前端的调用变得更复杂。BFF 的出现为前端应用提供了一个对业务服务调用的聚合点，它屏蔽了复杂的服务调用链，让前端可以聚焦在所需要的数据上，而不用关注底层提供这些数据的服务。

非必要，莫新增  
我们在看到 BFF 带来的各种好处的同时，也要注意到它所带来的代码重复和工作量增加方面的问题。如果与已有 BFF 功能类似，且展现数据的要求也相近的话，一定要谨慎对待新增 BFF 的行为。因此，建议非必要，莫新增。

## What

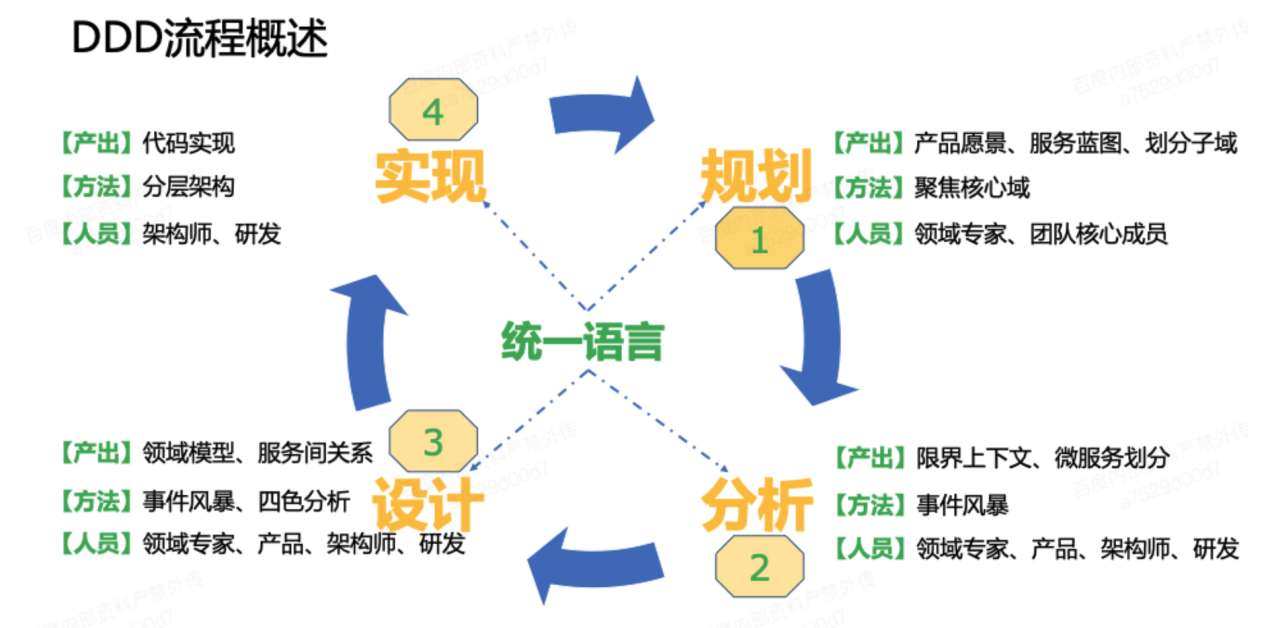
Domain Driven Design是一种应对软件复杂的设计方法论, 提倡聚焦业务领域而非技术,合理规划业务和技术的依赖, 从而达到一种稳定的领域模型应对多变的业务需求. 对技术开发来说往往关心技术选型, 工程实现细节, 而学习和理解DDD要求从软件开发的市场,技术,工程三个方面全面考虑. 先有需求的合理性,领域模型的合理性,才是技术方面的考虑.

x轴方向: 业务功能一定的前提下, 请求量变大,可以通过复制服务, 多部署几份即可.

z轴方向: 业务增长除了带来巨大的请求量,也会产生大量数据,z轴也要扩展,把数据分散在不同的地方存储, 这样每次请求只需要关心一部分数据.

y轴方向:业务扩展不仅仅流量变大,业务流程,场景,创新,业务复杂. y轴没有像xz轴那样通用的方案的.

y轴扩展达到瓶颈往往在于领域模型抽象的不合理,引入一个新的需求需要改动很多的老的逻辑, 特别依赖原先做这块逻辑的人的经验,领域知识没有很好的提炼和传承.



| 概念 | DDD定义 | 举例 |
| --- | --- | --- |
| 实体 | 实体一般对应业务对象，具有业务属性和业务行为 | 线索是个实体，线索的状态随着跟进活动推进需随时变化，需要根据唯一标识来追踪变化 |
| 值对象 | 值对象主要是属性集合，对实体的状态、特征进行描述 | 线索上的联系方式信息是值对象，不需要唯一标识去追踪联系方式的变化过程 |
| 聚合 | 聚合是由业务和逻辑紧密关联的实体和值对象组成的，是数据修改和持久化的基本单元 | 线索是个聚合，线索实体是该聚合的根实体，状态信息、联系方式信息等是聚合的附属 |
| 资源库 | 资源库是对资源访问的抽象。不局限于数据库、文件、网络存储。接口需要不依赖于具体的数据存储、ORM实现框架 | 线索这个聚合的访问通过线索资源库提供，资源库的实现因技术选型不同而不同，可以是数据库、文件等 |
| 领域事件 | 表示领域中所发生的重要事件，这种事件发生后通常会导致进一步的业务操作，或者系统其他地方引起反应 | 线索创建后会产生线索已创建领域事件，后续的线索分配服务、打标签服务可以监听该事件启动相应操作 |
| 领域服务 | 领域服务没有任何属性或数据，只是一个领域行为或动作，不适合放在任何聚合的领域行为 | 线索的查重行为属于领域服务，单个线索自身没法完成查重行为 |
| 应用服务 | 应用服务对应到一个具体业务场景，通过编排聚合、资源库、领域事件、外部适配接口、领域服务来完成 | 线索创建这个场景对应线索创建应用服务，该服务会编排线索查重、线索聚合创建、线索资源库创建、线索已创建领域事件发送等行为 |

业务需求的分析过程是从上而下的，由业务流程，到用户用例，到领域模型。而设计过程是相反的，自下而上的。从领域元素设计开始，最后才是应用服务的编排。

建议的设计优先级是先值对象 → 再实体 → 再聚合 → 再领域服务→ 最后是应用服务，优先考虑领域是否应该为值对象，其次是否为实体，划分出聚合。不属于实体或值对象中的领域行为放到领域服务，需要协调聚合的领域行为设计为领域服务或者应用服务。

任何业务代码逻辑优先映射到原子性的领域模型，比如值对象、实体、领域事件、资源库接口、外部适配接口，其次再映射到组合性领域模型，比如领域服务、应用服务

## Why

### 分而治之

核心域

-BC

-领域层

-聚合

支持域

DDD通过规划四重边界,把领域知识做了合理的固话和分层.

业务有核心域和支持域,业务域拆分多个界限上线文, 一个BC里又根据领域知识核心与否做分层,领域层里按照业务的强相关性划分聚合.

第一层边界

确定项目的愿景与目标,确定问题空间, 确定**核心子领域, 通用子领域,和支撑子领域.**

第二层边界

解决方案空间里的**界限上下文**就是一道进程隔离层面的物理边界.

第三层边界

每个界限上下文内, **分层架构**划分出来的接口层, 领域层, 应用层, 基础设施层之间的隔离.

第四层边界

领域层里为了保持领域的完整性和一致性, 引入**聚合**的设计做为隔离领域模型的最小单元.

### 领域建模

对领域知识抽象, 建模,让拥有数据的领域对象也拥有相应的行为能力.通过底层有序的建模可以灵活满足上层不同业务流程和场景的编排.

模型就是有凹凸面的乐高积木, 业务流程场景就是乐高积木搭建出来的各种形状,以有限数量的领域模型满足海量的业务.

### 通用语言

整个团队通用的业务语言协同, 问题和目标容易对齐, 跨团队沟通高效.

通用语言的适应范围一般是某个BC内.不同BC针对同一个概念可以有不同的语言.一个人是一个国家的公民, 一个公司的职员,一个家庭的成员,合作伙伴在销售域是客户, 在采购域是供应商, 在通用平台是第三方等. 一个无歧义的名字定义,可以贯通用户沟通, 需求文档描述, 设计方案代码编写的整个流程, 是团队资产的一部分.

### 架构演进

软件硬件分离:早期的软件都是硬件供应商开发, 预装在硬盘里.后来硬件和软件的分离,有很多专门做软件的公司出现.

开发和实施分离:早期的企业软件公司负责开发到实施的整个过程.后来开发和实施分离,有很多专门做软件的公司出现.

应用和架构分离:大型软件出现分布式架构,底层框架和中间件的出现降低了应用开发的门槛.

应用软件分层:DDD分层架构把应用代码分为业务和技术层次.

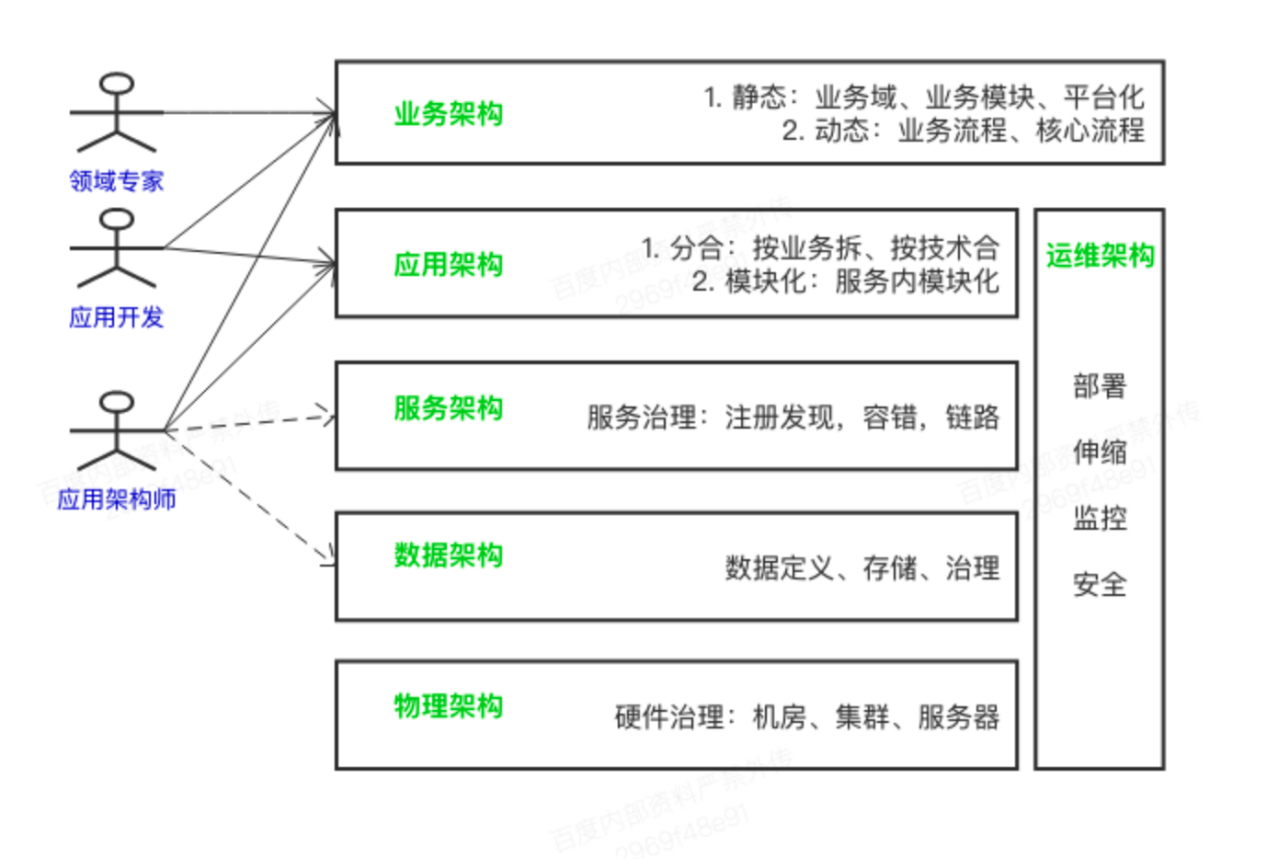
软件设计和运行环境分离:云计算把软件运行环境的硬件和操作系统分离, IAAS PAAS SAAS各层协同合作,云原生的普及, 应用软件的设计态少关心技术, 运行态不需要关心技术框架的细节.

应用开发和架构彻底分离: servicemesh serverless的背后就是想做到设计态的应用软件和技术框架无关, 让应用开发的专注力最大可能的集中在领域逻辑里.

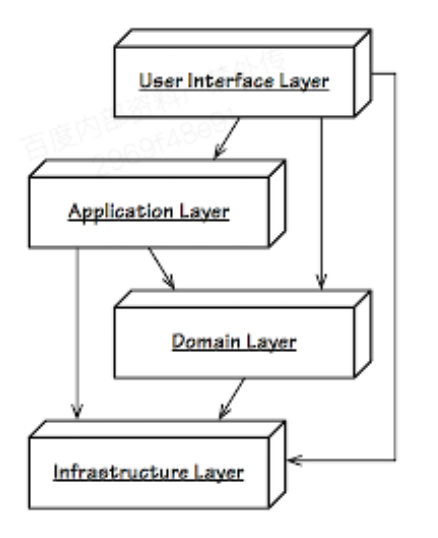
## 架构演进

### 应用架构

DDD的技术架构主要针对应用架构,但做为实施DDD的开发和应用架构师也需要关注业务架构,分布式服务架构和数据架构等.



### 分层架构

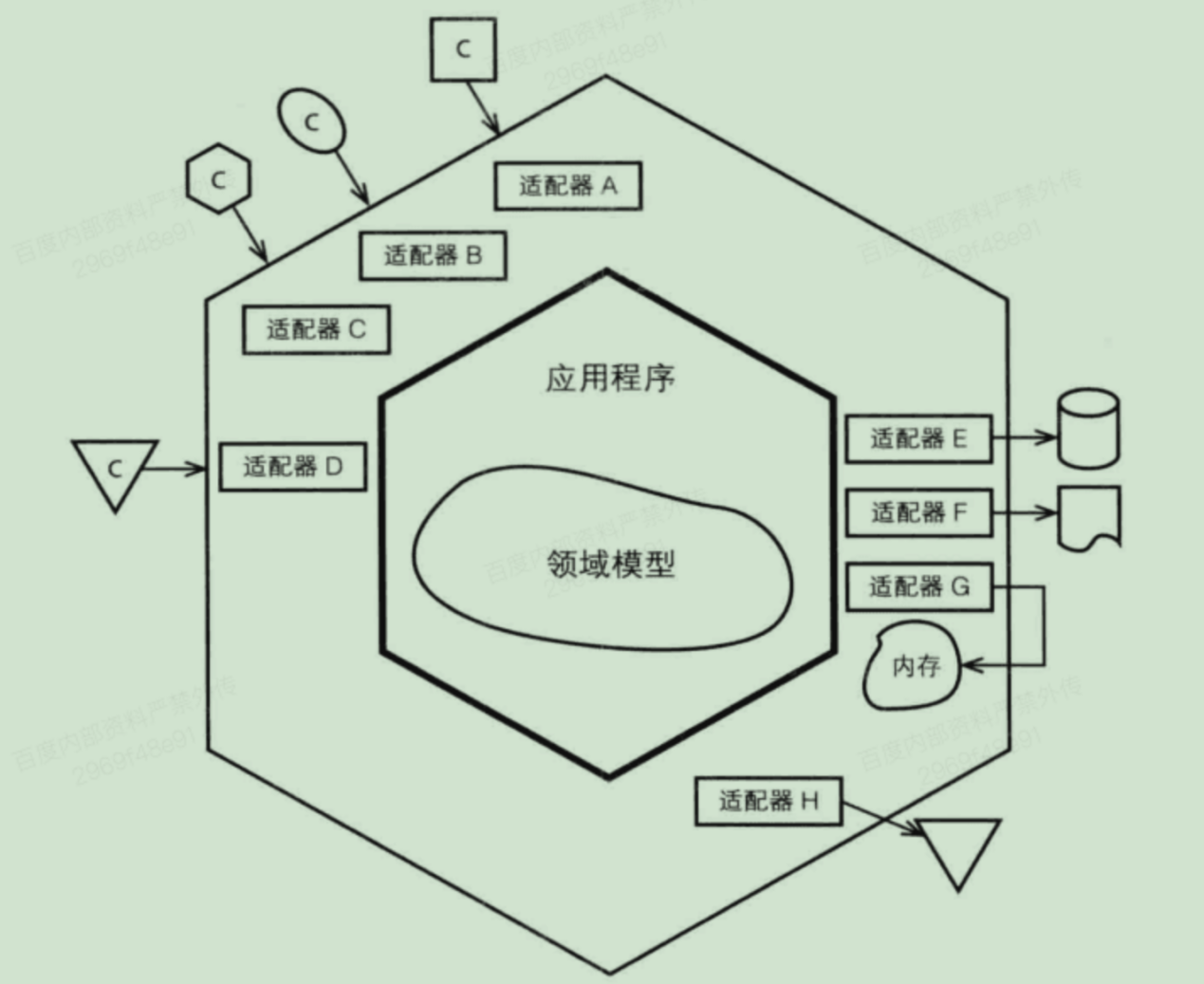


最早出现的DDD相关架构, 有EricEvans2003年提出, 也是大家俗称的分层架构.

特点:第一次提出把代码按照DDD的概念做了分层归属,特别是应用服务层和领域层是DDD特有的概念

缺点:各层次的依赖不够合理,都是根据请求的调用的方向自上而下产生依赖.比如领域层居然依赖了infrastructure层, 这会导致领域层没有完全给基础设施隔离, 领域的变化因素多了技术选型这个方面.

### 六边形架构



端口适配器架构,并没有融入太多DDD特色的应用服务层和领域层. 它只是笼统的分为内层的业务逻辑和外层的输入输出相关的适配层.适配层可以分为主动适配层和被动适配层.

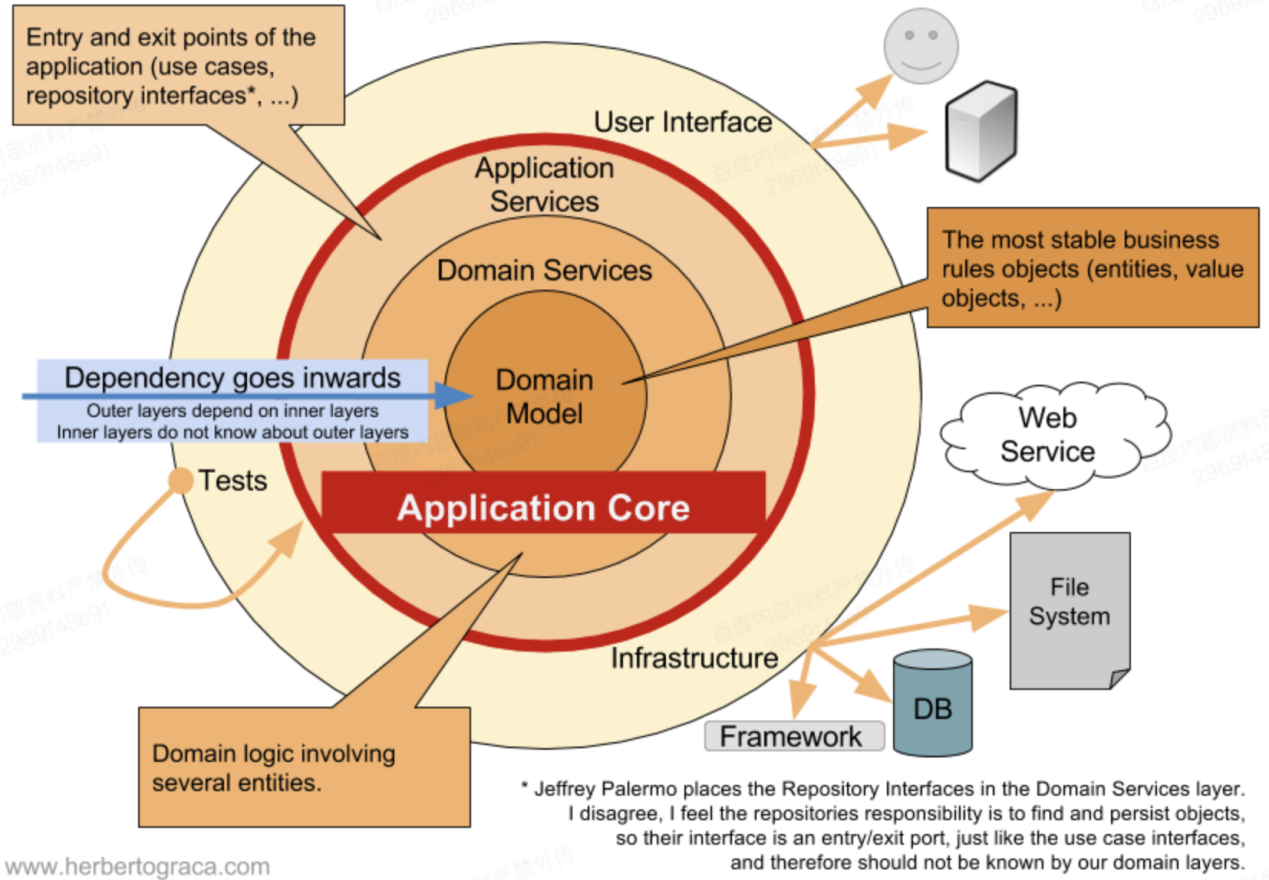
主动适配:指的是来自UI,命令行等输入型命令, controller就是一种端口,端口的具体实现就是应用逻辑自身.因此端口和具体实现都在应用系统内部.

被动适配:只访问存储设备,外部服务.每种访问就是一个端口,具体实现是各个具体的中间件.因此端口在整个应用系统内部,具体实现在外部.

每一种输入和输出都是一个端口，每个端口都有具体的实现逻辑，因此整个应用系统的架构就是一些列的端口+适配逻辑组成，架构图就是一个多边形形状。有几个端口需要根据应用系统的具体情况而定，只是六个端口比较形象而得名为六边形架构。

1. 外层依赖内层使得依赖更合理。端口就是接口，依赖接口编程。借此保证了应用和实现细节之间的隔离。2. 可测试更好

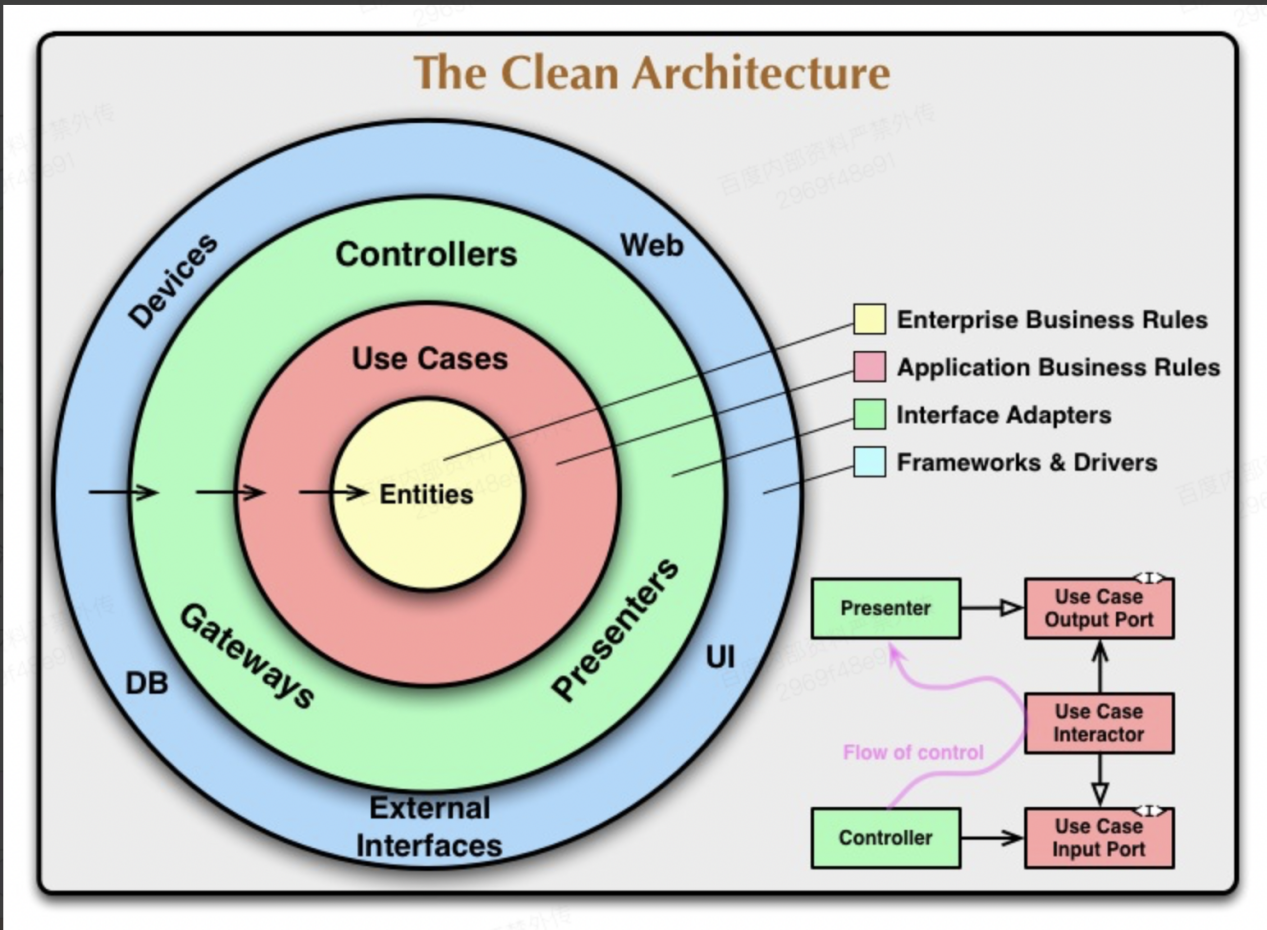
### 洋葱架构



洋葱架构针对六边形架构更进一步把内存的业务逻辑分为了DDD概念的应用服务层, 领域服务层和领域模型层.

1, 围绕独立的领域模型构建应用.2,内存定义结构,外层实现接口.3依赖的方向指向圆心.4, 所有的应用代码都可以独立于基础设施编译和运行.

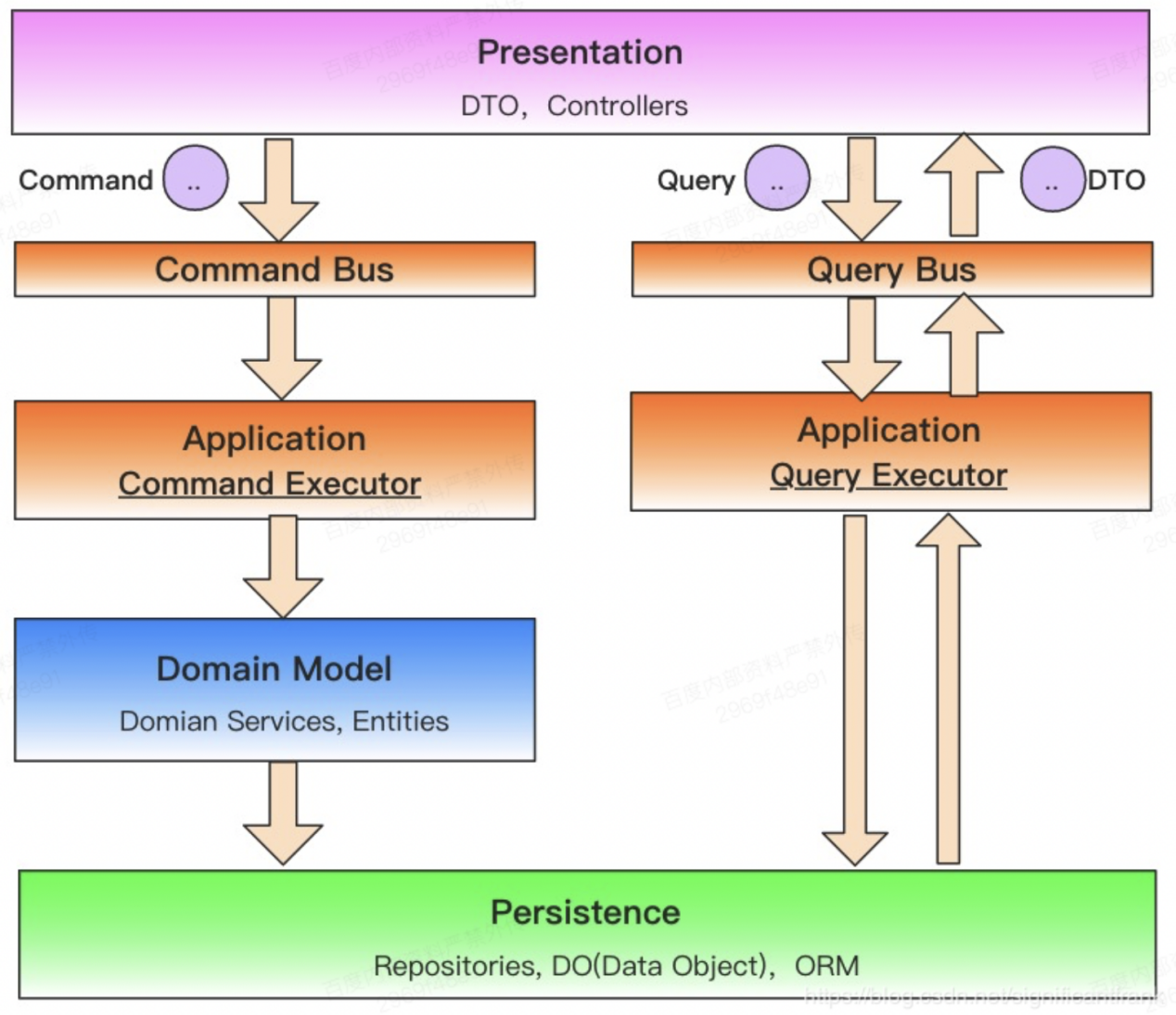
### 整洁架构



整洁架构跟洋葱架构基本一样, 只是层次上的角阀没有完成遵循DDD的概念.Entities对应领域层，User Cases：应用层，这层需要串联上游和下游的资源完成具体场景的问题。Controllers、Gateways、Presenters：本质上是各类适配器，即用户接口层+基础设施层。

特点：依赖方向是从外层圆向内层圆。任何内层圆里的代码不应该应用外层圆的代码。越里层越解决领域逻辑。

### CQRS



CQRS一般结合event sourcing一起使用，是一种读写分离架构，命令类型负责产生和修改数据，查询类型负责读取数据。适合读多写少场景下提高读性能。读写服务物理分离，数据改变通过消息中间件传递消息，达到同步意图。  
【评价】：读写分离场景不建议用CQRS。主从延迟会带来的数据的实时性和一致性问题。违背微服务数据私有原则，真正流量大时一般会用别的方式解决，比如缓存技术、分库分表、单独的数据访问服务做统一优化，单元化架构。目前技术栈里复杂度读场景的性能不是瓶颈，利用索引外置方案的ES能满足目前需求。

灵活多变条件下的读场景。读写在同一个服务，让读场景不用遵循领域分层，不需要从DB开始逐层传递数据，从而提高开发效率。  
【评价】：一个服务里没有完全使用领域分层，容易使得领域代码腐烂。目前建议查询场景也经过领域模型层，通过其他手段来提高查询场景开发效率。

## DDD架构

### 界限上下文

限定上下文与业务:当我们讲一个领域概念的时候一定有其特定的上下文. 比如商品这个概念,订单上下文里商品的代表就是商品的单价,库存. 库存的上下文里也有商品的概念. 但是代表的是商品的库存量,库存成本,库存存放位置等.不同的上下文有同一个概念, 但是表达的领域知识,所属的角色是不一样的,这就是BC的作用.

限定上下文与技术: BC之间的边界一旦确定了, 相互之间的协作方式也能确定下来,这就是对技术架构的约束.举例：下订单这个业务流程要求订单BC和库存BC、支付BC协作完成才算完整流程，而发送订单通知这个动作则可以稍后完成。这就对订单BC如何和其他BC协作产生了同步请求还是异步请求两种不同的方式。

### 划分BC

1, 根据相关性做分类

2, 再根据团队的粒度做裁剪

【语义相关性】创建一个订单、支付一个订单、发货一个订单、评价一个订单、搜索一个订单，这些有"订单"语义上的相关性，但是在业务上却有很大差别，涉及的业务逻辑和角色也不太一样，因此往往把它们划分为不同的BC。

【功能相关性-业务】创建订单、修改订单、合并订单、拆分订单这些属于针对订单的相关性操作，往往划为一个BC。

【功能相关性-非业务】订单BC、库存BC、支付BC等都有分析、监控、可扩展需求，往往会把这些非功能需求单独划分为一个个BC。

领域职责的归属有时很难划清楚，需要从多个角度思考，提问。比如这个需求谁会关心？这个需求如果我不做，谁一定会做？出了问题会是哪个领域的责任？

根据类似两个披萨的团队粒度，结合康威定律，一个BC不要跨团队维护，一个成员也不要同时维护太多个BC，这就要求针对相关性归类处的BC做一次适合团队维护的粒度裁剪。特别注意的是：如果团队小就没必要划分太多BC，人为制造太多边界，对微服务和BC的道理是类似的。中短期内的好处往往会被大家高估，但带来的副作用却经常被低估，比如最终一致性对业务连续性的牺牲，对部署资源和运维资源的浪费，业务链路太长对请求响应的延迟等等。特别是团队对于业务理解不透彻、对非业务需求的技术支持能力没跟上的情况下更是这样。

### BC与微服务

微服务的概念这里不详说，简单说微服务是包含高度相关功能的一个开发部署单元，有自己的技术自治性包括技术选型、弹性扩缩容、发布上线频率等，有自己的业务演变自治性。

【概念澄清】BC是根据领域逻辑的内聚情况形成一个整体。微服务是部署单元，一个微服务就是一个独立服务，一个独立进程。

【落地情况】一个微服务可以包含一个或多个BC，到底包含几个？需要根据团队大小、BC复杂度和技术特性来定。1）如果业务复杂度高，需要几个人一起维护，建议拆分。2）如果BC的技术复杂度高，比如需要严苛的高可用、高并发，考虑拆分成单独微服务独立部署，方便单独技术演进。推荐做法是一个微服务就是一个BC。当一个BC里有某个领域概念业务发展快，功能变多，但又没大到可以单独一个BC时，继续留在这个BC里。只要在原来的BC里做到了以聚合为单位做领域设计的话，只要时机成熟，可以在代价很小的情况下拆分独立出来。

【DDD怎么帮到微服务设计】微服务规划和落地时需要确定多大的业务范围是一个独立的微服务，划分好的微服务随着业务发展的拆分和合并，这些问题最终都要建立在DDD的四重边界基础上。这就是DDD和微服务的联系点。

### BC间的协作关系

DDD里划分BC是为了局部业务领域的独立性，但往往一个完整的业务流程需要多个BC协同完成，BC间的协作关系有很多种、协作关系有方向。

谁说上游、谁是下游？

BC间协作关系是有方向的。在上下文映射图中用Upstream代表上游，Downstream代表下游。不要从调用关系、数据流动方向判断谁是上下游。在AB协作关系中，如果A不用关心B的细节则A是上游，B是下游。

### 协作关系类型

共享内核：多个BC共享了一个组件。这个关系通常用来解除BC间的双向依赖、循环依赖

客户方-供应方：上下游间基于协商的接口进行协作，跟开发主机服务有点相似，但开放主机服务关系下的BC间耦合度更低

防腐层：下游为了防止上游接口、协议变化而主动引入的一个抽象层次。比如对接外部系统时，BC内的调用逻辑不直接依赖外部接口，而是依赖防腐接口，让防腐接口去依赖外部接口

开放主机服务：以开放的、稳定的协议提供服务，比如rest、rpc

发布/订阅事件：引入消息队列解耦关系

不要出现BC间的双向依赖、循环依赖，通过调整、合并领域职责，最后考虑引入共享模式解决

一个好的协作设计，各自职责一定是“分治”的，尽量少用集权的机制，少用共享模式。

遵守最小知识法则，不要BC内的职责和知识泄露到别的BC里，尽量用松耦合协作关系，比如防腐层、开发主机、发布订阅

### 数据模型

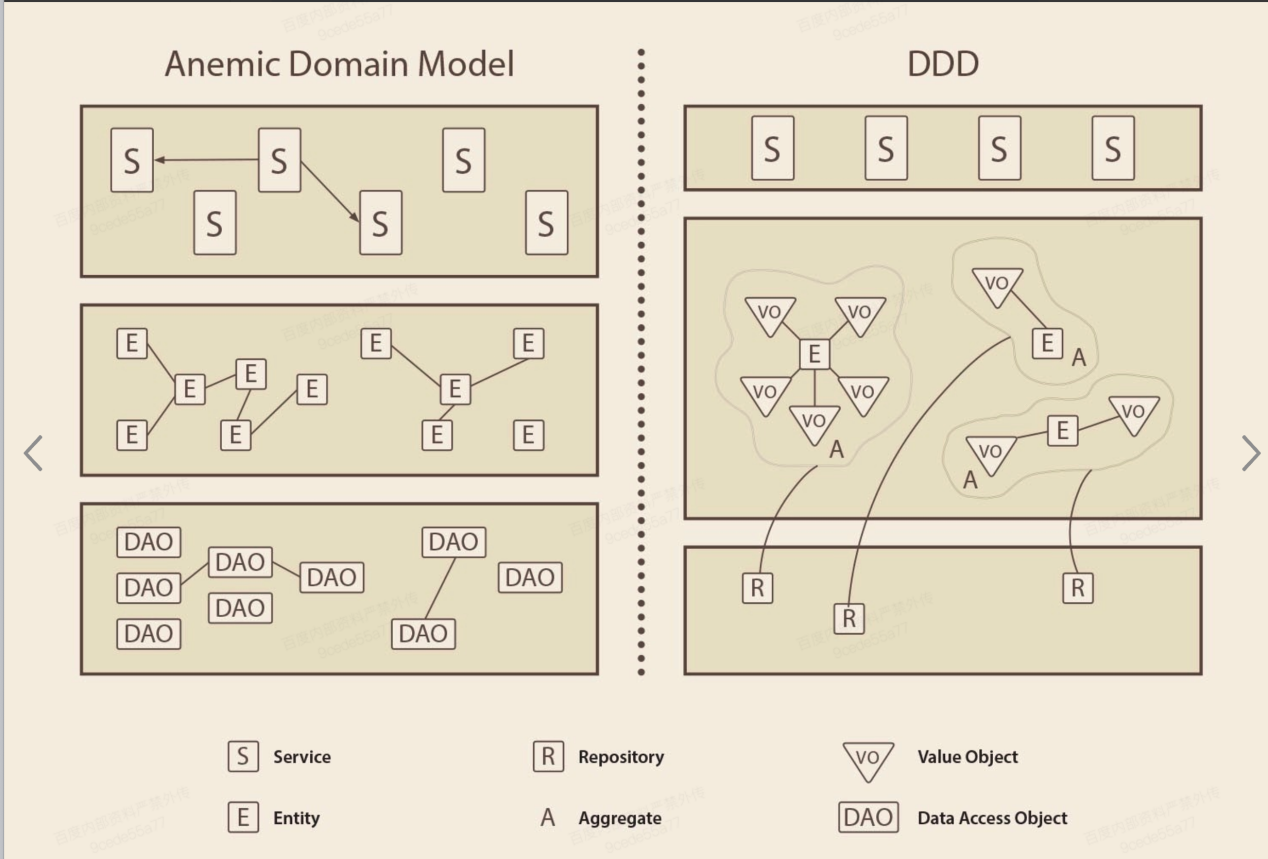
java bean：只有私有字段，及对应字段的getter和setter方法的java对象 = 贫血对象  
pojo：普通的java对象，不依赖jdk之外的框架，纯粹的java对象  
DTO：是个java bean，只有属性没有任何行为

entity：具有数据字段和内聚的行为，不对外部产生依赖，也是一种pojo对象  
value object：具有数据字段和内聚的行为，不对外部产生依赖，也是一种pojo对象。有些地方把跟UI展示数据有关的也有叫view object，建议改为UI Object简称为UO  
persistent object(PO)：维护与数据库表字段的映射关系，直接跟ORM框架无缝集成

### 贫血模型还是富血模型

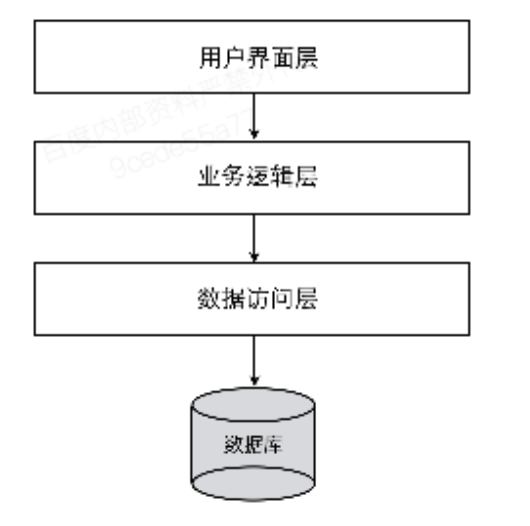
贫血指实体对象里只有属性和属性的getter和setter。富血指除了属性外，按照职责单一原则把属于实体对象的行为定义在实体里。

DDD设计推荐富血方式，根据实际情况甚至需要把行为定义到实体的值对象里。比如实体属性的格式校验、数学运算转换

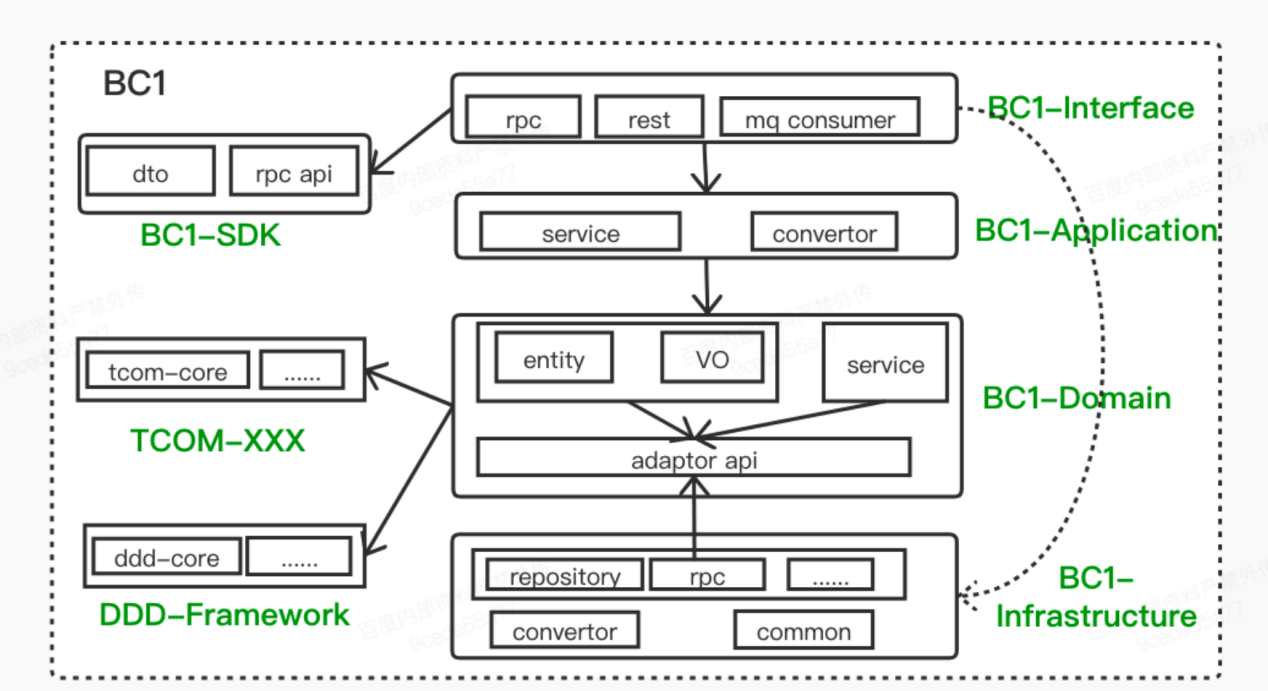


|  |  |  |
| --- | --- | --- |
| 对比点 | 贫血 | 富血 |
| 封装性 | 代码分散，需要多处修改，易出bug | 一处代码，single of truth |
| 测试性 | 有类似校验类的代码测试路径多 | 容易mock掉实体，不关心实体细节 |
| 代码整洁性 | 参数多，判断条件多 | 封装好的实体对上游而言只是单个参数 |
| service实现代码膨胀 | 包含很多实体该执行的逻辑的service类，  代码膨胀快，不易被应用层编排 | 实体和实体下面的值对象承担了必要的行为逻辑 |
| 简单实体、简单业务 | 有优势、学习门槛低 | 学习门槛高、对简单实体、业务没优势 |

### 整洁架构



业务逻辑层的AbcServiceImpl类是个上帝类，事务脚本，过程式业务逻辑实现。前几行代码做validation，接下来做convert，然后是业务处理逻辑的代码，中间穿插着通过RPC或者DAO获取更多的数据，拿到数据后，又是convert代码，然后接着一段业务逻辑代码，最后可能还要落库，发消息…等。



整洁分层架构是在整洁架构的各层次附上DDD含义的基础上，结合洋葱架构而得来。旨在保证领域模型的纯粹性，避免领域模型出现贫血模型。该架构具有以下特点：

共分5层interface、api层、application层、domain层、infrastructure层。每层是个jar包，bootstrap类放在interface层，单独出来也可以。

interface层以开放主机服务方式对接BC外部请求。

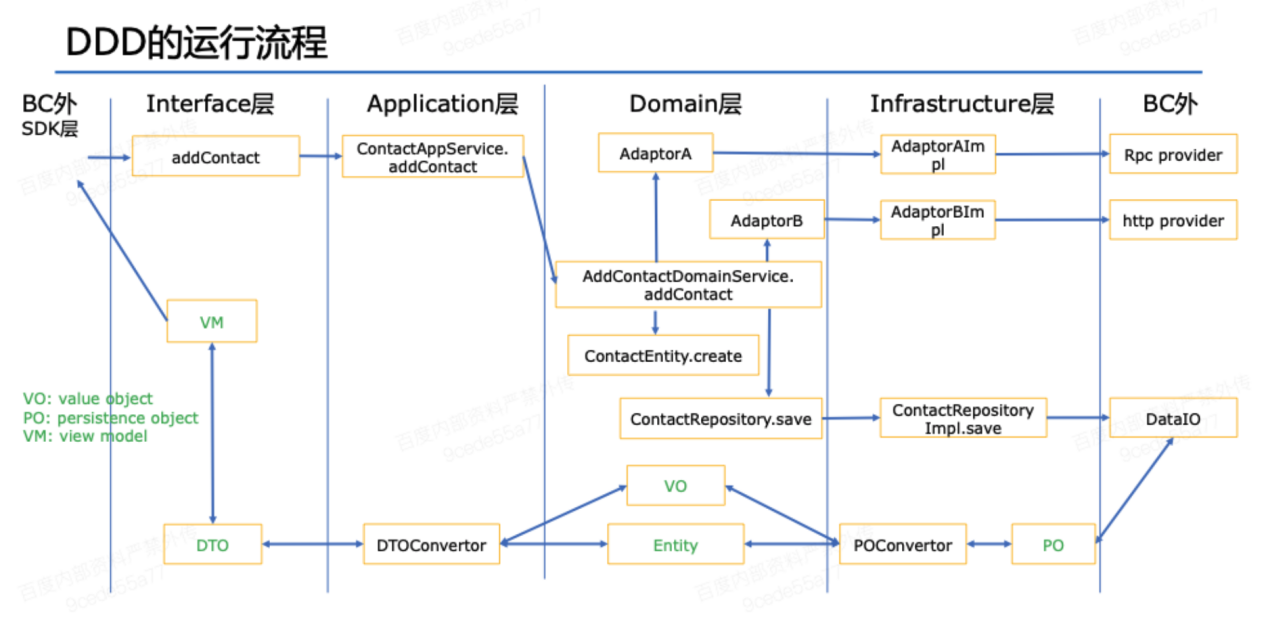
domain层不依赖其他层，通过adaptor包下的接口定义做到依赖倒置，adaptor接口参数也不能体现具体技术实现细节，domain里的实现逻辑只依赖接口。adaptor是很重要的防腐逻辑。domain层里以聚合为单位放置代码，便于以后系统拆分，以聚合为单位。

infra层负责向BC外部发出请求。如果infra里的adaptor接口实现逻辑非常多的时候，甚至可以考虑单独出来这一层。

注意：interface层虚线依赖infra层不是实际代码依赖，仅仅是为了springboot工程启动时能正常打包infra层代码。







原则：不同的分层里要用合适的数据对象。命名带上相应后缀。

interface、application层：入参是ui或者open api传入的DTO，消息对象。出参是DTO或者UO

domain层：只能见到领域对象 - 聚合、实体Entity、值对象VO

infra层：跟资源库相关的是持久化对象PO；跟adaptor相关的是目标rpc接口的sdk里定义的xxxDTO，建议转换为xxxVO在domain层使用，作为对外部服务xxxDTO的防腐。即domain层不依赖rpc provider.

## 架构原则详解

### interface层设计

用户接口层，负责对外暴露开放主机服务（对应DDD里的open host service），对接各种协议访问比如http、tcp。经过了接口层后最终都要转成应用服务所能认识的协议，并通过应用服务入参传入应用层。

暴露并实现接口：rpc、rest、mq subscriber

网关注册@Register，Swagger支持比如@ApiOperation

协议转换：请求上游如果不是直接对接前端，在gateway网关层已经做了协议转换

接口限流等功能

入参定义：http、mq consumer接口的入参对象可以定义在app层，interface层共用。

入参校验（只包含判断是否null等没有领域含义的校验逻辑）

依赖application层

【tips】不要做本属于application层的业务逻辑

【tips】不要依赖infrastructure层，通过adaptor接口

### application应用层设计

应用层类似设计模式的facade模式，把里层的领域层服务做编排，对外层提供粗粒度的接口。所以应用层主要做两件事：1. 编排用例，一个接口就对应了业务场景的用例。2. 处理一些横切面的事情。

编排用例：绝大部分情况下应用服务通过调用领域服务来编排业务流程

启动事务，目前统一用dataio提供的注解@DBTransactional

对入参做必要验证。注意：一定要思考哪些逻辑该在application层做，涉及到实体/值对象的属性校验肯定要下放到领域层的实体/值对象自己承担。

DTO到entity的互转：在访问domain层之前之后

调用一个或者多个领域服务

横切点功能：事务开启在应用层、打日志、异常处理

依赖情况：domain层，dataio sdk

【tips】不要做本属于domain层的业务逻辑，应用服务和领域服务职责怎么区分？详见[DDD的设计原则](http://wiki.baidu.com/pages/viewpage.action?pageId=1233535631)相关章节。自查问题：这段代码蕴含的知识是否与它所处的限界上下文的主要职责域直接有关？比如消息发布

### domain领域层设计

所有业务逻辑相关的代码都应该放在这个层，不应出现任何跟具体技术相关的词汇，比如资源相关的操作只有repository，没有dataio、mybatis、redis等具体实现repository的技术词汇

domain层除了依赖一些公共库之外不能依赖任何层，通过相应的adaptor接口做到和infra层的依赖倒置（对应DDD里的防腐层设计）

### infrastructure基础设施层设计

本BC访问外部系统、组件需要的逻辑都放在本层。主要实现一系列adaptor接口，典型的adaptor有：

对资源存储的访问比如数据库、缓存

对内部其他系统的适配逻辑，通过rpc或者http访问。熔断逻辑可在此维护。对rpc调用得到的出参进行防腐处理详见<各个分层对应的各种数据对象解释>

对外部第三方厂商系统的适配逻辑

对基础框架的适配逻辑，比如消息队列、配置中心、文件等。其中rcc配置中心目前对应用逻辑无侵入，当作正常的spring配置项使用。

## 领域模型

### 元素

值对象（Value Object）

实体（Entity）

聚合（Aggregate）

资源库（Repository）

工厂（Factory）

领域事件（Domain Event）

领域服务（Domain Service）

应用服务（Application Service）

### 职责

表示模型的是：实体、值对象、领域服务。这意味着所有领域逻辑都应该在这三种对象里。后续引入的领域事件也是一种领域逻辑。

聚合是为了进一步让领域逻辑更内聚，起到边界保护的作用。怎么保护呢？详见下面聚合设计原则部分。

工厂和资源库是为了管理领域对象的生命周期。工厂负责对象的创建（在复杂创建逻辑下才需要）。资源库负责聚合的加载、添加、修改、删除。

应用服务处于应用层，对领域逻辑编排、封装之后对上层接口层暴露。一次编排就是一个用户用例。

### 模型与服务

模型只是个逻辑概念，模型的数据方面由实体、值对象、领域消息来承载。领域行为包括实体里的行为、值对象里的行为、领域服务里的行为。

服务是个统称，在分层架构里不同的分层里对应不同的服务。领域层里有领域服务（domain service）、应用层里有应用服务（application service）、基础设施层里有基础设施服务、与外部系统交互的服务（open host services）。

### 实体

**不应该给实体定义太多的属性或行为，而应该寻找关联，发现其他一些实体或值对象，将属性或行为转移到其他关联的实体或值对象上**。比如Customer实体，他有一些地址信息，由于地址信息是一个完整的有业务含义的概念，所以，我们可以定义一个Address对象，然后把Customer的地址相关的信息转移到Address对象上。如果没有Address对象，而把这些地址信息直接放在Customer对象上，并且如果对于一些其他的类似Address的信息也都直接放在Customer上，会导致Customer对象很混乱，结构不清晰，最终导致它难以维护和理解

简单说实体是具有属性和行为的对象。但这个定义不足以说明实体本质。因为值对象也有属性和行为。

实体的本质特征是具有唯一标识和具有业务连续性变化。业务连续性变化指随着时间推移实体的某些属性会发生变化，比如订单的状态随着订单履约流程随时变化，而订单号这个唯一标识是用来追踪这一变化的。

标识是唯一的，用来对外沟通的手段。就像中国公民具有身份证，订单具有订单号一样，实体有自己的唯一标识，可以是系统内部生成的也可以是系统外部传入的，可以由业务规则生成，也可以无业务含义。

标识是不会变的：实体的其他属性可能会变，但唯一标识在实体的整个生命周期都不会变。订单号不会随着订单的创建、支付、发货、关闭而改变。

属性是实体的特征表现，具体分为基本属性和组合属性。

基本属性：是那些由开发语言内置的基本类型就能表示的属性如name、description字段往往是个字符串。

组合属性：不是一个基本类型就能表示出来的特征，比如重量Weight，既有数字也有单位，2kg和2g。组合属性会被设计为值对象

行为 相对于属性代表的是实体的静态特征，行为是实体的动态特征。实体的行为有：

变更属性的行为比如更改描述

对自身属性加工计算的行为

跟别的属性协作的行为。需要依赖别的实体信息来做出决策。这一类行为的设计需要特别注意，和协作的对象要分清职责范围。

注意：实体的创建、增删改查不属于实体自身的行为，是实体的生命周期管理，需要交由工厂和资源库来负责。

实体设计原则

把内聚性更强的属性尽量设计为单独的值对象，甚至包括没有唯一标识的"实体"。

需要遵循"信息专家"原则，避免实体变为贫血对象。实体不能只是拥有数据，也要具备数据对应的领域行为。应该由实体做的事情必须在实体内定义领域行为。

确保实体的创建方法返回的是一个有效的实体 - 接受必要的、经过校验的入参、如有集合成员属性需要初始化好

实体里须抛出特定业务类异常

如果实体逻辑里需要别的实体

如果是依赖本聚合内的实体，可以直接引用

如果是依赖本聚合外的实体，要么通过参数传入，要么把逻辑放置领域服务，建议通过领域服务来实现跨聚合的逻辑。

### 值对象

值对象是对实体的描述。一个值对象只描述一个实体，少数情况一个值对象是个通用的描述，可能会用来描述多个实体。

注意：值对象也有自己的行为，尽可能把属于值对象自己的行为放到值对象里

【无唯一标识】没有唯一标识的对象都是值对象。这也值对象和实体的本质区别。

甚至订单项OrderLine也是一个值对象。对用户而言只关心订单里的哪个产品、数量多少、价格多少等信息，不关心第几个订单项。只是订单项可以作为单独表存储，从数据存储的角度要求这个表要有个唯一id，但这只是个技术id，不是唯一标识。

【验证逻辑】有自我验证逻辑的属性，这些属性的验证逻辑如果放到实体里，会导致职责不够清晰，实体容易变的膨胀。

email属性，需要对是否一个有效的email地址做验证

收发货地址，需要验证地址格式有效性等

【计算逻辑】有自我计算、换算逻辑的属性：

Weight重量属性，两个重量间的四则运算逻辑应归属于Weight值对象

Money金额属性，金额包含数量和货币单位，Money字段间有换算逻辑

【同类属性】多个属性在描述一个领域特征

电话号码、座机、传真都在描述联系方式

存储到哪？如果够简单就随实体表一起保存，如果复杂可以有单独的表存储（此时单独表里每行记录的唯一id只是为了技术手段使用，在聚合外这个id没有任何领域含义）

谁来触发存储？所属的实体

是否具备唯一标识是实体和对象的本质区别。实体一旦创建会有后续状态变化，每个变化都需要通过唯一标识来追踪。

### **领域概念显性化（Domain Primitive）DP**

Domain Primitive的定义

让我们重新来定义一下Domain Primitive：Domain Primitive是一个在特定领域里，拥有精准定义的、可自我验证的、拥有行为的Value Object。

DP是一个传统意义上的Value Object，拥有Immutable的特性

DP是一个完整的概念整体，拥有精准定义

DP使用业务域中的原生语言

DP可以是业务域的最小组成部分、也可以构建复杂组合

注：Domain Primitive的概念和命名来自于Dan Bergh Johnsson & Daniel Deogun的书 Secure by Design。

使用Domain Primitive的三原则

让隐性的概念显性化

让隐性的上下文显性化

封装多对象行为

在完成了实体和值对象的设计后，有的时候会发现有些概念其实在领域上是存在的，但设计和代码里没有Class来体现，可能仅仅是一个基本类型参数加上散落的对该参数的判断检验逻辑，这个时候还需要思考应该把这个概念显性化，定义专门的Class并包含相应逻辑，入出参以相应Class为类型。

比如联系人实体的电话号码，它应该成为一个领域概念出现，电话号码的校验不属于其他实体或者值对象，应该属于TelPhone自己。有些书籍把这类概念叫做domain primitive，就好比编程语言有基本类型primitive type和自定义高级类型一样。但本质上这也是一种值对象。

领域概念显性化的好处：1. 有意义的领域概念。2. 入出参具备强类型检查。3. 相关逻辑典型的是校验、内部格式规整等具有合适归属地。

**case study: 联系人实体，包含属性电话号码**

|  |
| --- |
| **class** ContactEntity {      String telPhone; // Bad ->: TelPhone telPhone;  }  // Bad，代码散落没有抽取，职责不合理  **if** (telPhone == **null** && isValidTelPhone(telPhone)) {  **throw** **new** ValidationException();  }  // Bad，职责不合理  ValidationUtils.validateTelPhone(telPhone);  // Bad，注解方式只适合简单校验比如判断null  @NotNull @Pattern(regexp = "^0?[1-9]{2,3}-?\\d{8}$") String telPhone    //Good，在相应实体的校验中，委托给了telPhone值对象自己校验  contactEntity.validate() -> telPhone.validate()    ContactEntity findByPhone(String telPhone); // Good ->: ContactEntity findByPhone(TelPhone telPhone); |

### 聚合

聚合内部的对象之间可以相互引用，但是聚合外部如果要访问聚合内部的对象时，必须通过聚合根开始导航，绝对不能绕过聚合根直接访问聚合内的对象，也就是说聚合根是外部可以保持 对它的引用的唯一元素；

聚合内部的对象可以保持对其他聚合根的引用；

通常在大部分领域模型中，有70%的聚合通常只有一个实体，即聚合根，该实体内部没有包含其他实体，只包含一些值对象；另外30%的聚合中，***基本上也只包含两到三个实体。这意味着大部分的聚合都只是一个实体***，该实体同时也是聚合根。

引入聚合是为了划分对象之间的边界，聚合即DDD引入的第四重边界。一个聚合至少有一个实体，承担最主要概念的实体就是聚合根，也叫根实体。聚合边界之内需要保持领域概念的完整性，边界之外聚合需要有独立性，需要跟别的聚合协作。对象之间是否应该构成一个聚合是DDD设计里的难点和重点。独立性和完整性是聚合的两个最重要特性。

独立性

根实体是访问聚合边界的唯一入口。反过来讲若需要独立访问一个实体，则它只能作为聚合根。如果一个实体不是根实体，但同时真的需要被外界直接访问到，那么这个实体不应该在这个聚合中，应该独立成新的聚合。

【**case study**】比如账户和账户交易记录作为一个聚合，账户设计为根实体。但业务需要不通过账户而单独查询账户交易记录，比如所有账户的交易记录。如果账户交易记录不独立为一个聚合，则业务被迫需要每次通过账户来访问账户交易记录，这显然是不合理的

完整性

一个聚合代表一个完整的领域概念。完整性可以从几个方面理解：

【共享生命周期】就像面向对象设计的合成关系composite，对象间的生命周期一致，需要同时被创建，同时被销毁。组成一个聚合的其他实体的操作由根实体来协调

【case study】订单Order与订单项OrderLine，订单项不能离开订单而存在，订单如果没有订单项包含的信息就不是一个完整订单。同时用户只关心订单这个领域概念，用户可以不了解订单项这个领域概念，对用户而言只关心产品、产品数量、产品价格等信息。因此满足合成关系的对象往往会被设计为一个聚合。

【保证业务规则约束】施加在聚合内部各对象之上的各种业务约束需要满足，聚合的自我完备性需要保证包括数据、行为、约束

【case study】采购订单中添加和删除订单项，必须满足多个订单项金额之和不能超过订单的审批额度。如果订单和订单项不在一个聚合，这个业务规则很难得到保证

【满足事务一致性】聚合通常被认为是一个事务的边界。也就是说聚合内的根实体、其他实体、值对象数据应该保持在一个事务边界内。如果不一致说明聚合边界设计存在疑问。加载和修改聚合就是一个事务单元。事务虽然是技术实现的范畴，但可以反过来验证聚合设计是否合理

【case study】如果订单和订单项是一个聚合，则订单和订单项的增删改必须满足一致性。如果分属不同的表存储，则订单表或者订单项表的插入失败会导致整个插入操作失败。当然事务的实际执行层面是聚合对应的资源库的职责。

设计原则

【独立性】只有聚合根才是访问聚合边界的唯一入口

【完整性】尽量保持聚合领域概念的完整性

【访问原则】聚合之间应通过聚合根的身份标识进行引用，聚合内部的实体间可以通过对象引用

其他聚合需要加载一个聚合时（往往通过领域服务/应用服务协调），必须通过目标聚合的资源库，并返回聚合的完整内容，但资源库可以有支持多种加载条件的接口。注意：直接查询类：是否可以通过资源库返回聚合的部分信息？有些实施DDD的做法是查询类不走领域层，用类似DAO的方式而不是资源库方式。我建议我们保持领域完整性，都走资源库查询，而且每个查询返回聚合整体。在领域层或者应用层做信息裁剪

新增一个聚合必须调用根实体必要的校验和领域逻辑，并传递给资源库一个完整聚合

更新一个聚合必须调用根实体必要的校验和领域逻辑，并传递给资源库一个完整聚合。注意：资源库的实现层面可以做到局部更新

【聚合协作】同一个BC的聚合间协作可以采用本地事务保持一致性，跨BC（即跨微服务）的聚合协作通过基于消息机制的最终一致性

【聚合与资源库】不要在聚合中使用资源库

【聚合粒度】在上述规则满足的前提下，聚合设计尽量小，小聚合带来小的事务粒度、有更好的性能

设计过程

理顺领域对象图（对象图是领域建模的分析阶段的产出物）

把领域对象设计为值对象，实体。值对象依附于某个实体。

基于领域完整性和独立性识别出聚合、定义聚合根。注意：当完整性和独立性产生矛盾时，优先满足独立性。

本着聚合粒度尽量小原则，一开始可以把每个实体当作一个聚合。

合并实体。判断聚合是否足够完整？判断实体是否有独立性需求？如果实体A和实体B生命周期一致，同时A没有被外界独立访问的需求，则合并A和B为一个聚合。如果实体A有被独立访问的需求，则A作为单独聚合。

走查聚合设计粒度，综合考虑技术设计因素

|  |
| --- |
| 计算机与cpu、计算机与键盘  1. 计算机与cpu属于对象的合成关系，设计为一个聚合是合理的。计算机离开了cpu就不能称作为一台计算机，一台计算机里的cpu也不会单独被外界访问。适合设计为一个聚合，计算机为实体，cpu为值对象  2. 计算机与键盘属于对象的组合关系，这个依赖很弱，各自都可以单独存在，适合设计为两个聚合 |

|  |
| --- |
| 以银行的取款为例：当储户账户发起取款操作时，需要扣除账户Account的余额Balance，同时创建一条新的交易记录Transaction，以便于银行对账，并支持储户的交易查询功能。  1. 根据实体和值对象设计原则，Account和Transaction应该是实体，Balance是个典型的值对象（不存在唯一标识，对实体的一种描述）。  2. Account和Balance应该为一个聚合，Account是聚合根，没有争议。  3. 从聚合完整性考虑，账户余额扣除成功和取款的交易记录创建成功需要保持一致，否则会导致数据不一致。所以聚合的边界应该包括Account、Balance与Transaction这三个实体。  4. 从聚合独立性考虑，对于Transaction来说，由于储户可以执行交易查询功能，这意味着调用者可以绕开Account这个聚合根而单独查询Transaction。因此这里的Transaction需要具有独立性，应该单独为它建立一个聚合。  5. 这样的设计就打破了Account和Transaction的数据一致性问题，遵照聚合的独立性原则高于完整性原则这条，这个数据一致性问题需要用别的方案来解决。比如可以通过领域服务来协调两个聚合。 |

考试系统的问题Question和答案Answer，知识问答论坛的问题和答案：

1. 考试系统的Question和Answer，概念上强相关，构成了一个考题的整体，外部不会绕开Question单独查看Answer，因此设计为一个聚合比较合理。

2. 知识类论坛的问题和答案跟考试系统的不一样，虽然问题和答案也是概念强相关，但每个答案可以被读者单独回复、评论、点赞等，从独立性考虑，答案应该被设计为单独的聚合。问题和答案两个聚合通过引用各自标识。

|  |
| --- |
| 产品分类ProductCatagory和产品Product  1. 从正常需求讲，产品属于一个分类，产品分类和产品各自维护信息，应该设计为两个聚合。产品聚合引用产品分类的唯一标识，即产品里有一个字段叫产品分类CatagoryID，产品分类里有一个字段ProductID  2. 如果问当删除一个产品分类的时候，是不是产品的数据一致性就破坏了？并没有，删除产品分类只是删除了产品分类和ProductID的关系，并不需要删除分类下的所有产品。  3. 如果需要在删除一个产品分类的时候删除分类下的产品，则产品分类更像是产品的一个属性，应该设计为一个聚合，产品是聚合根，产品分类是值对象 |

拼团业务中，“团”与“团员”，业务规则：1.一个用户只能参团一次 2.团最多只能有N人，到达N人后，团的状态为”满团“

方案1：从概念完整性考虑，设计团为聚合根，下面有个List<团员>的实体集合。缺点：N很大比如=100的时候，有性能问题，加载一个团的时候需要把100个团员信息加载进来

方案2：除了看完整性，团员本身有独立的访问需求，设计团为聚合根、团员也是聚合根。团维护着它的业务规则：人数和状态，包含属性List<团员Id>。一个团员只能参加一个团的业务规则由领域服务承担

结论：方案2更优于方案1

### 资源库

资源库是对聚合访问的一种抽象。资源库不局限于数据库，还可能是文件、网络存储。接口需要不依赖于具体的数据存储、ORM实现框架。

资源库像集合一样，提供添加、更新、获取聚合等接口，达到聚合的生命周期管理。

【与工厂的区别】工厂负责聚合实例的生，垃圾回收负责聚合实例的死，资源库就负责聚合记录的查询与增删改。

【与DAO的区别】DAO模式也能做到数据访问，但DAO没有边界控制作用，没有聚合作为一个整体的概念，service类想查什么表就查什么表。

设计原则

一个聚合一个repository，对聚合的生命周期管理只有资源库这个入口。

仓储里面存放的对象一定是聚合，原因是之前提到的领域模型中是以聚合的概念去划分边界的；聚合是我们更新对象的一个边界，事实上我们把整个聚合看成是一个整体概念，要么一起被取出来，要么一起被删除。我们永远不会单独对某个聚合内的子对象进行单独查询或做更新操作。因此，我们只对聚合设计仓储

资源库的接口在领域层、实现在基础设施层

要访问聚合内的其他实体和值对象，也只能通过聚合对应的资源库进行，这就保护了聚合的封装性。

不要包含领域逻辑

|  |
| --- |
| // 领域层代码  **public** **interface** OrderRepository {      // 查询方法的命名更加倾向于自然语言，而不是find等技术语言      Optional<OrderEntity> orderOfId(OrderId orderId);      // 以下两个方法在内部实现时，需要组装为通用接口的 criteria      Collection<OrderEntity> allOrdersOfCustomer(CustomerId customerId);      Collection<OrderEntity> allOrdersOfCustomerWithSpecificStatus(CustomerId customerId, OrderStatus status);        // 在底层实现中，新增和更新都可以视为是保存  **void** save(OrderEntity order);  **void** saveAll(List<OrderEntity> orders);  }    // 基础设施层代码，实现这个接口的时候委托给了一个能完成通用DAO操作的内部对象。  **public** **class** OrderRepositoryImpl **implements** OrderRepository {      @Resource  **private** GenericRepository<OrderEntity> genericRepository;      **public** Optional<OrderEntity> orderOfId(OrderId orderId) {  **return** genericRepository.findById(orderId);      }      ......  } |

### 工厂

工厂负责聚合的对象实例化工作，需不需要定义专门的工厂类依据创建逻辑复杂与否而定。

不包含第二个实体的简单聚合可以直接new

聚合根实体自己定义静态创建方法

创建逻辑复杂时，设计单独的工厂类，隐藏创建细节

### 领域事件

领域设计时不应过多考虑性能等实现层面的事情，聚合间的引用只能是根实体ID

领域内事件

如果一个应用服务对应的用例包含了多个聚合的状态改变，一个方案是通过领域内的事件来协同，但这带来异步响应编程模式。另一个方案是通过领域服务来编排多个聚合行为，这意味着所有事情都是在一个事务内同步完成。

两个方案各有优劣，事件方式优点是松耦合，异步响应，缺点是流程不直观，从应用服务角度很难知道一个完整流程是什么，不太方便查找问题，事件响应逻辑出错会带来事务一致性问题。领域服务编排方案优点是流程直观，事务一致性。缺点是如果编排逻辑发生变化需要改动的逻辑多，流程太多的话可能会有性能问题。

推荐：优先考虑使用领域服务编排完成。微服务架构下每个微服务的职责相对聚焦，一个用例通常不会同时修改多个聚合。如果需要修改多个聚合也应该是一个BC内的编排，可以通过应用服务启动的数据库事务来保证强一致性。另外领域设计提倡先从领域知识出发做聚合设计，技术实现问题比如性能问题往往放到后面做为走查领域设计的参考因素。如果没有强一致性需求的话，甚至可以通过向分布式消息队列发送领域间事件来达到自己发送自己消费的效果。

领域间事件

DDD提倡BC间尽量解耦，尽可能使用发布订阅协作模式做上下游解耦，发布订阅消息即指领域间事件。

消息内容需要包含全部必要信息，让消息订阅者不必再次回查事件内容。但必要信息一定不意味着整个领域实体，需要对领域实体做裁剪后重新定义出领域事件对象。

领域消息需要包含租户信息，可以通过继承包含了租户等字段的公共消息类

如何传递消息往往借助分布式消息中间件的二方库，事务最终一致性也是消息中间件需要同步考虑的事情。

领域事件的发布应该在领域层还是应用层？

只要不会破坏各层的依赖顺序，在哪发布都行。取决于领域事件定义在哪层？一般推荐定义在领域层的聚合内。因此即使应用层发布事件也不会破坏依赖方向。则聚合、领域服务、应用服务都可以发布事件。

### 领域服务

但是基于类的面向对象语言规定任何属性或行为都必须放在对象里面。所以我们需要寻找一种新的模式来表示这种跨多个对象的操作，DDD认为服务是一个很自然的范式用来对应这种跨多个对象的操作，所以就有了领域服务这个模式。和领域对象不同，领域服务是以动词开头来命名的，比如资金转帐服务可以命名为MoneyTransferService。当然，你也可以把服务理解为一个对象，但这和一般意义上的对象有些区别。因为一般的领域对象都是有状态和行为的，而领域服务没有状态只有行为。需要强调的是领域服务是无状态的，它存在的意义就是协调领域对象共完成某个操作，所有的状态还是都保存在相应的领域对象中。我觉得模型（实体）与服务（场景）是对领域的一种划分，模型关注领域的个体行为，场景关注领域的群体行为，模型关注领域的静态结构，场景关注领域的动态功能。这也符合了现实中出现的各种现象，有动有静，有独立有协作。

领域服务还有一个很重要的功能就是可以避免领域逻辑泄露到应用层。因为如果没有领域服务，那么应用层会直接调用领域对象完成本该是属于领域服务该做的操作，这样一来，领域层可能会把一部分领域知识泄露到应用层。因为应用层需要了解每个领域对象的业务功能，具有哪些信息，以及它可能会与哪些其他领域对象交互，怎么交互等一系列领域知识。因此，引入领域服务可以有效的防治领域层的逻辑泄露到应用层。对于应用层来说，从可理解的角度来讲，通过调用领域服务提供的简单易懂但意义明确的接口肯定也要比直接操纵领域对象容易的多。这里似乎也看到了领域服务具有Façade的功能

领域服务也是一种领域逻辑，在分层架构里属于领域层。但领域服务没有任何属性/数据，只是一个领域行为/动作。

领域服务的三个典型场景

协调多个聚合完成业务操作。因为聚合不能持有别的聚合对象，只能引用外部聚合根的唯一标识，因此需要跨聚合的业务操作需要领域服务来完成。（如果没有业务逻辑的话，部分场景也可以由应用服务来完成）。

不适合任何聚合的领域行为。比如一个单据的导出行为是否应该属于该订单所在的聚合？由于导出需求涉及到格式的多样性等，建议设计成领域服务，由一个接口和多个实现类（每种格式）组成。

领域行为需要与访问包括数据库在内的外部资源协作时。

|  |
| --- |
| // 如何协调聚合根Account和聚合根Transaction之间的关系，需要用到领域服务来协调和封装  **public** **class** WithdrawingDomainService {      @Repository  **private** AccountRepository accountRepo;      @Repository  **private** TransactionRepository transRepo;    **public** **void** withDraw(AccountId id, Amount amount) {          AccountEntity account = accountRepo.findBy(id);          account.decrease(amount);          accountRepo.save(account);            Transaction transaction = Transaction.createFrom(id, amount);          transRepo.save(transaction);      }  } |

|  |
| --- |
| //“验证订单有效性”这一验证行为需要验证Order聚合边界内的信息比如是否提供了配送地址、联系人信息、是否含有有效订单项。但同时也要验证下订单的顾客是否为有效顾客，这是另一个聚合Customer的范畴。这一典型的跨聚合的行为需要领域服务。由于聚合之间的协作只能通过身份标识进行，Order聚合只是持有Customer的身份标识CustomerId，要获得Customer聚合就需要通过该Customer的资源库。    **public** **class** ValidatingOrderDomainService {  **private** CustomerRepository customerRepo;  **public** **boolean** isValid(Order order) {          //step1 验证订单内信息          order.validate();            //step2 验证客户信息          Optional<Customer> optCustomer = customerRepo.customerOf(order.getCustomerId());  **return** optCustomer.isValidCustomer();      }  } |

领域服务设计原则

粒度要小，一个领域服务只做一件事，便于应用服务编排。

作为参考可以要求领域服务的名称必须包含一个动词。对于同一个聚合关联的添加、更新、查询、删除等相关领域服务建议设计为四个不同的类，避免领域类太大。

一个比较常见的坏习惯是没有尽最大努力为领域行为找到一个归属的实体，轻易定义成领域服务，这容易让领域服务类变成"上帝类"。

### 应用服务

应用服务属于分层架构里的应用层，每个用户用例对应一个方法。

外观模式：对外提供粗粒度接口、对内分配职责的协作作用。类似设计模式的外观模式。

坚持不做业务决策的原则：应用服务自己不做任何业务逻辑的决策。每当你觉得应用服务里的代码做出的业务决策是跟本BC业务逻辑有关联时，往往是把不必要的业务代码蔓延到这层了。

应用服务方法的入参和出参都是DTO，调用领域层需要传入entity，返回interface层需要把entity转成DTO

应用服务的两个典型场景：

不包含领域逻辑的业务服务应被定义为应用服务。

与横切关注点协作的服务应被定义为应用服务：事务、认证授权等

DTO的设计原则

入参DTO尽量少暴露字段

多个用例下尽量不复用入参DTO

尽量复用出参DTO（可以比要求的多暴露字段）

创建和更新用例下，返回整个实体

DTO的校验不包含领域校验逻辑，比如唯一性校验, 就类似与checkparam只检查基本的, 业务的在领域里面

### 应用服务与领域服务

看起来领域服务和应用服务都能做编排功能，原则上是应用服务不应该包含领域层的逻辑，那怎么判断什么是领域层的业务逻辑？这个问题没有标准答案，设计时不用太条条框框。一个简单的判断标准可以自问一下：这段代码蕴含的知识是否与本BC的主要职责域直接有关？如果相关的话不要在放应用服务。

洋葱架构作者和Eric Evans都提到过不用规定的太死，外层一定只能调用它的直接内层。只要调用关系不会破坏层次的耦合就行（外层依赖内层，内层不依赖外层）。也就是说应用服务不一定非要通过领域服务来访问资源库和根实体，也可以直接引用资源库和根实体，领域服务也可以引用资源库和根实体。

不要用领域服务对根实体的方法做简单封装后供应用服务调用

验证订单是否有效

提交订单

移除购物车中已购买的商品

发送短信通知买家

取决于设计者怎么理解这四件事情，语义上看前面三件事跟本BC有密切关系，第四件事发送短信与本BC没有密切关系，出现在应用层也合理。

方案1：把四个步骤封装成一个领域服务placeOrderDomainService，应用服务OrderAppService调用它

方案2：把前三个步骤封装成一个领域服务placeOrderDomainService，应用服务OrderAppService调用它和执行第四步

方案3：把四个步骤都放在应用服务OrderAppService来执行

### 领域模型元素间的关系

业务需求的分析过程是从上而下的，由业务流程，到用户用例，到领域模型。而设计过程是相反的，自下而上的。从领域元素设计开始，最后才是应用服务的编排。

建议的设计优先级是先值对象 → 再实体 → 再聚合 → 再领域服务→ 最后是应用服务，优先考虑领域是否应该为值对象，其次是否为实体，划分出聚合。不属于实体或值对象中的领域行为放到领域服务，需要协调聚合的领域行为设计为领域服务或者应用服务。



## 实施过程

### 战略设计

战略阶段重要性不言而喻，但战略一旦定了不会经常变动。围绕整个产品的愿景、核心竞争力，回答要解决什么核心问题，用什么关键技术手段、配备多少人力、物力资源。

目标：战略设计阶段会完成：

【划出第一重边界】针对产品愿景、业务要解决的问题域，规划核心域、通用域、支撑域，做合适的资源投入

【划出第二重边界】基于分析阶段的领域提炼的成果，划分限界上下文（bounded context），确定限界上下文之间如何协作。

【划出第三重边界】架构层面制定分层架构，隔离关注点，减少核心的领域层沉淀并稳定下来，尽量不会因为外界变化而发生大的变动

### 模型分析

分析阶段需要业务、产品、技术团队紧密合作，围绕特定的问题域，提炼出完整的领域概念，形成领域的通用语言，为后续跨团队协同战术动作奠定坚实基础。

目标：识别提炼出领域概念，它们大致的关系是什么？比如概念A必须依附B存在，先做C再做D，做完E步骤必须见到F等等。

参与角色：PM、领域专家、敏捷小组

何时完成：比如需求讨论&交接会

产出：1. 维护本限界上下文的通用语言表 2. 维护本限界上下文的领域模型分析图

方法：业界有很多领域模型建模方法：名词动词、四色建模、事件风暴等。简单领域采用名词动词法就够了，复杂领域利用其他方法。推荐采用四色建模方法。

### 需求质量

正如[DDD的入门](http://wiki.baidu.com/pages/viewpage.action?pageId=1234179212)里说的，软件交付需要需求、技术、工程通力合作。需求和技术之间存在着一个明显的知识翻译流程，这个流程进行的质量往往决定了后续软件交付效果。这个翻译流程的质量影响面有方方面面，从市场/用户处如何提炼出有效需求，需求如何被抽象、裁剪，如何准确无误的解释给技术人员，技术人员如何在时效性、技术限制前提下再次抽象、裁剪后实现出来，每一次的信息传递熵增都会带来信息失真。

编写好的用户故事（user story）

用户故事应该只受到业务规则与业务流程变化的影响。只描述做什么，不关注怎么做。

不被UI操作流程捆绑，UI 操作不是业务行为 (用户想要的并不一定是真实需求)

不受技术实现干扰（技术现状只是眼前的苟且，真实需求才是诗和远方）

可验收，有交付标准（能衡量出效果的才是需求交付）

澄清横向影响点（cross impact analysis）

尽可能把横向影响点描述清楚，是否影响其他限界上下文，其他领域对象

### 四色建模法

分析阶段要对业务进行梳理抽象并可视化出来，让大家都理解要解决的问题和涉及的业务场景、业务流程、业务概念。比较有效的方法有事件风暴、四色建模。

事件风暴使用门槛低，该方法认为事件流很⼤程度上反映了现实业务逻辑，参与人员基于领域事件发生的时间线，把事件的前因后果逐步罗列出来。整个过程包含识别领域事件、决策命令、领域名词三个步骤。通过尝试回答这几个问题：这个业务涉及的系统产生了什么变化？变化由哪个角色通过什么方式触发的？系统变化产生了哪些结果？

四色建模来源于Peter Coad的《彩色UML建模》，旨在把所有的模型对象抽象为四种，并在模型图里用四种不同的颜色标记出来。

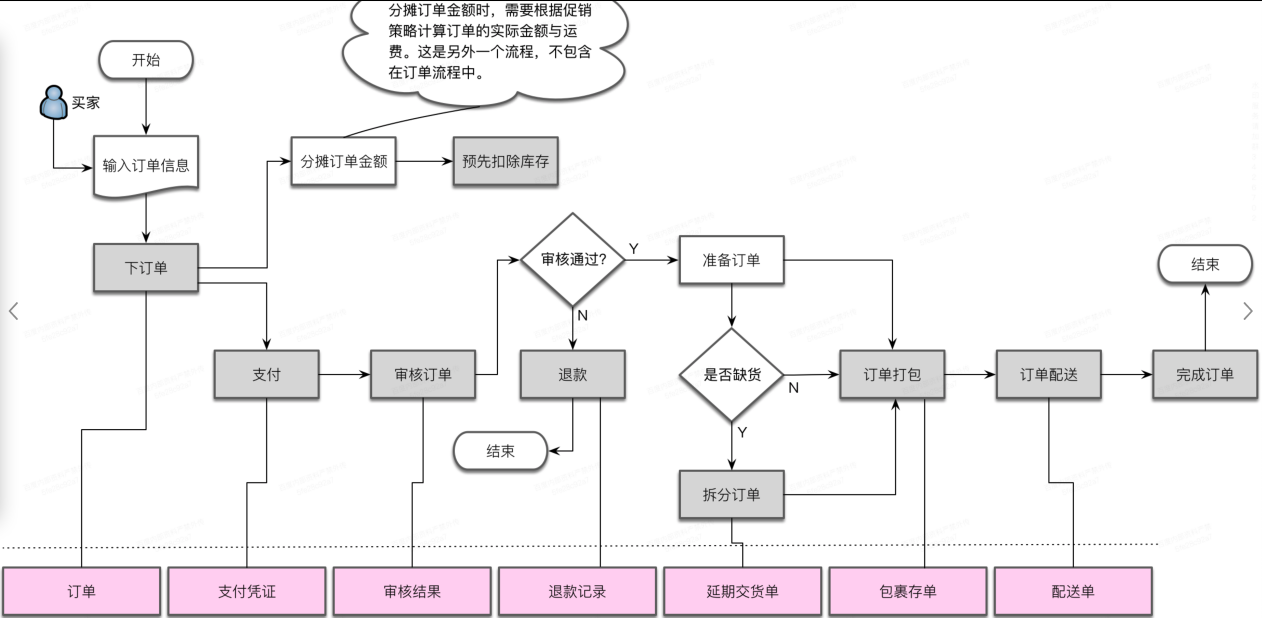
1. 时刻-时间段原型（Moment-Interval Archetype，MI）  
   这类对象用来记录某个时刻或某段时间内管理和运营数据，用粉色表示。比如订单系统里，订单就是典型的MI对象。
2. 参与方-地点-物品原型（Part-Place-Thing Archetype，PPT）  
   业务流程中的参与者，可以是人、物、地点。用绿色表示。
3. 角色原型（Role Archetype）  
   PPT对象在参与MI过程中，扮演的角色，用黄色表示。
4. 描述原型（Description Archetype）  
   对PPT对象的更详细描述，用蓝色表示。

MI PPT Role Desc

简单概括上面四种原型就是：具体一个什么人或物以某种角色在某个时刻或某段时间内在某个地点参与某个活动。 其中“具体、什么”就是描述原型，“人或物”、“某个地点”就是参与方-地点-物品原型，“角色”就是角色原型，而”**某个时刻或某段时间内的某个活动记录"就是时刻-时间段原型。**

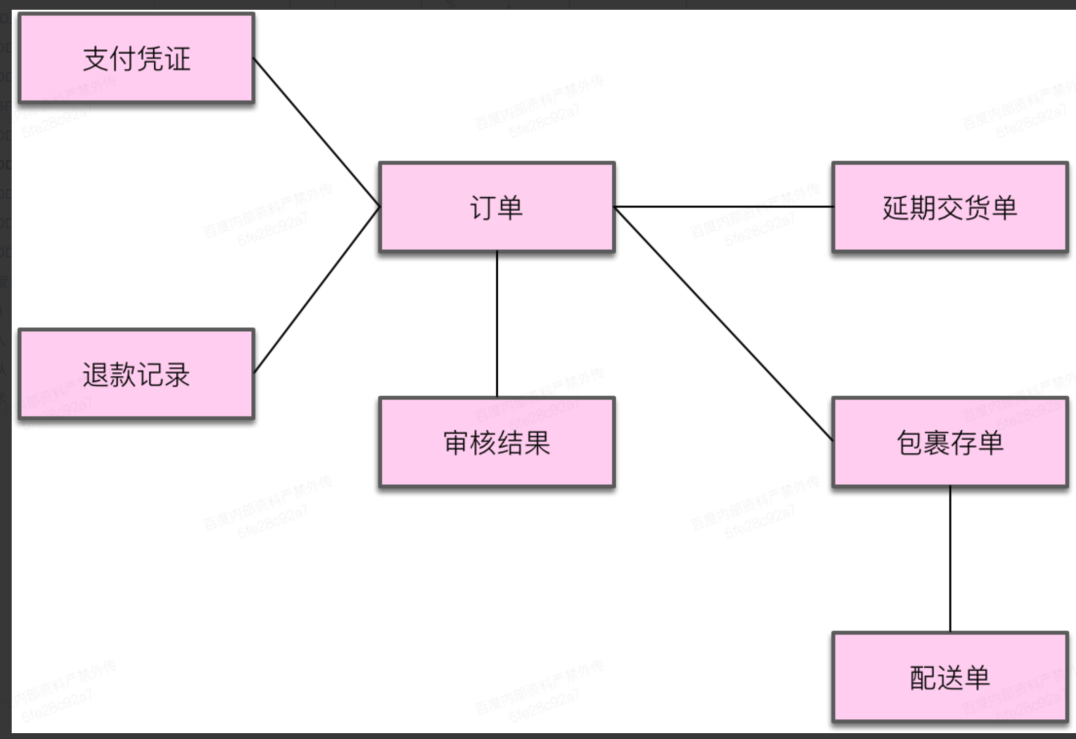
下面以电商系统下定单到订单配送完成举例，如何使用四色建模法来分析和提炼领域知识。

确定系统的关键业务流程，分解出关键步骤。比如订单系统里的主要步骤是下订单→ 预扣库存 → 支付 → 打包 → 配送 → 完成

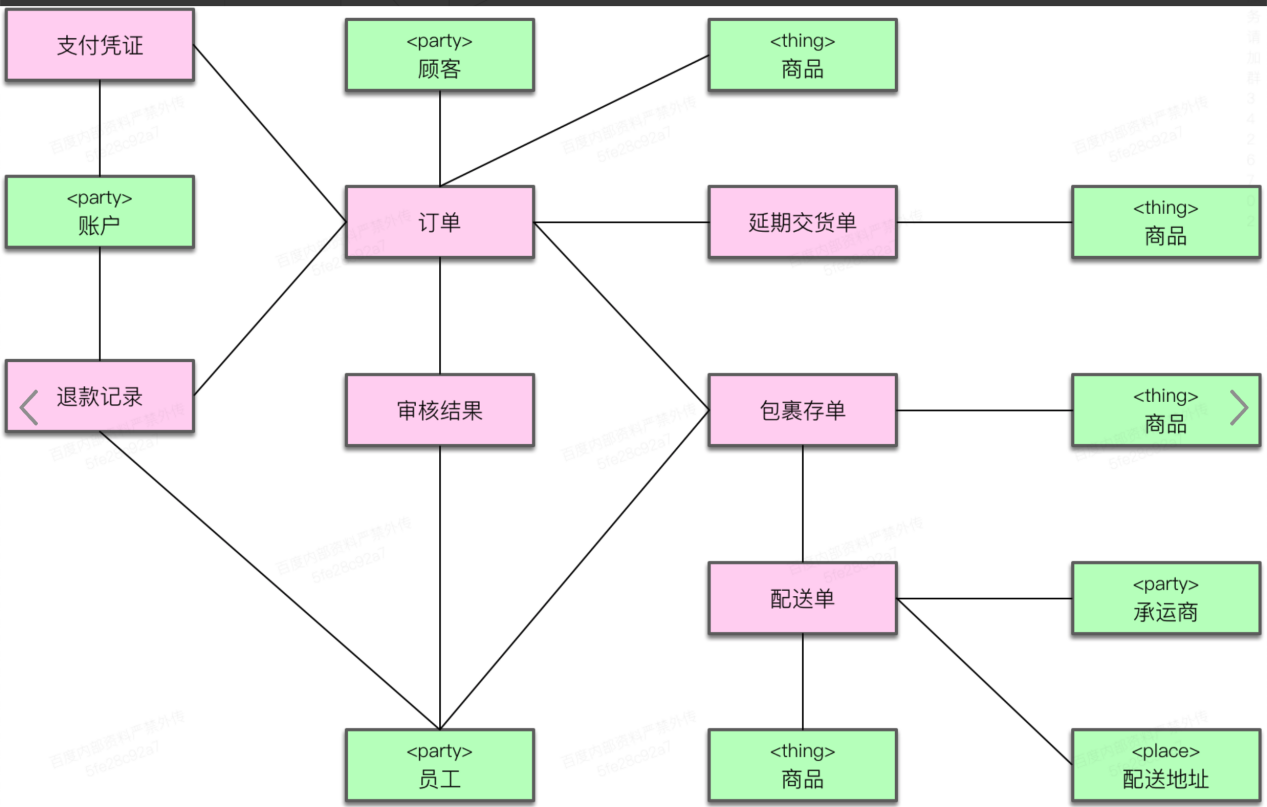


这些关键步骤**是否需要可追溯数据**，缺失了是不是影响运营管理？需要可追溯的数据正是四色建模里的MI对象。如何识别MI对象？一看某个时间点或者时间段内业务动作是否会产生数据。二问如果没有这份数据会影响企业运营管理吗？比如下订单动作会产生订单数据，如果没有这些订单数据可追溯，后续的发货、支付基于什么来操作？客户要退订单怎么办？这就意味着订单数据必须作为MI对象存在系统里。

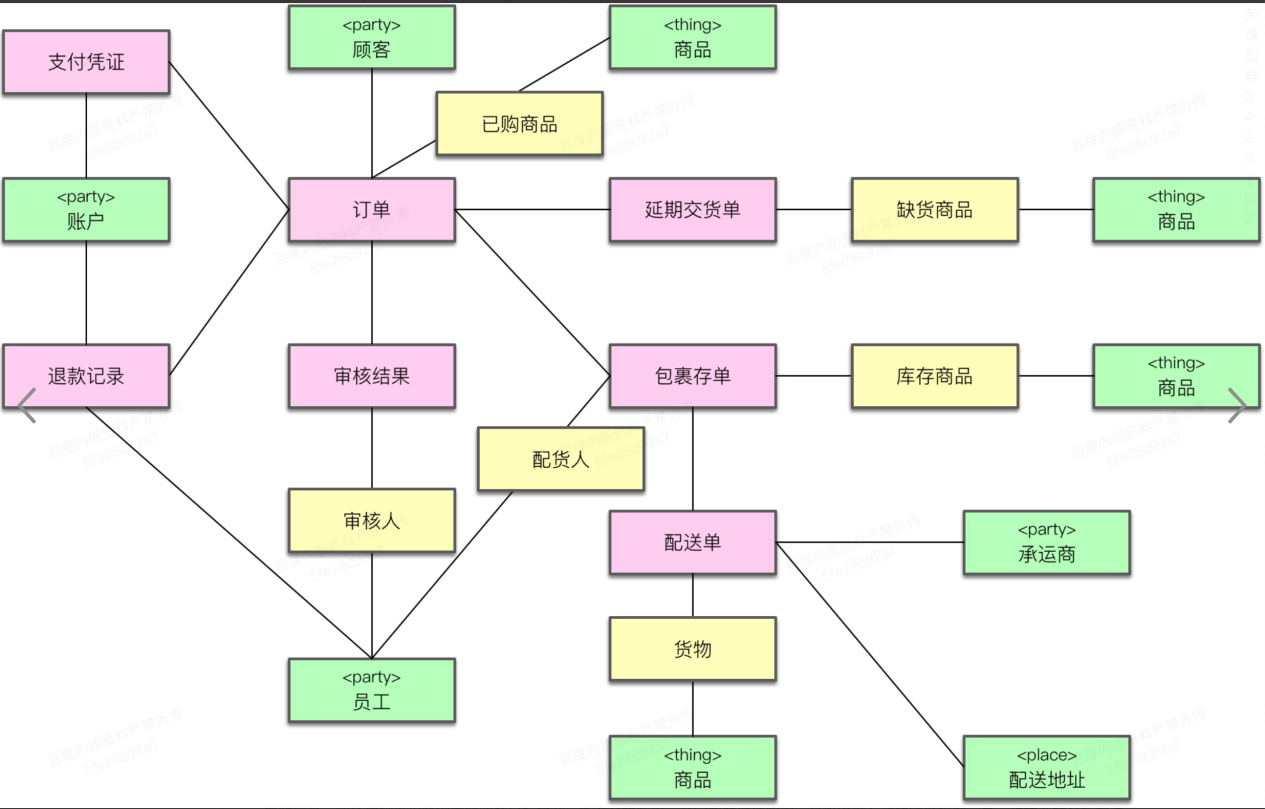
下图的就都是MI对象



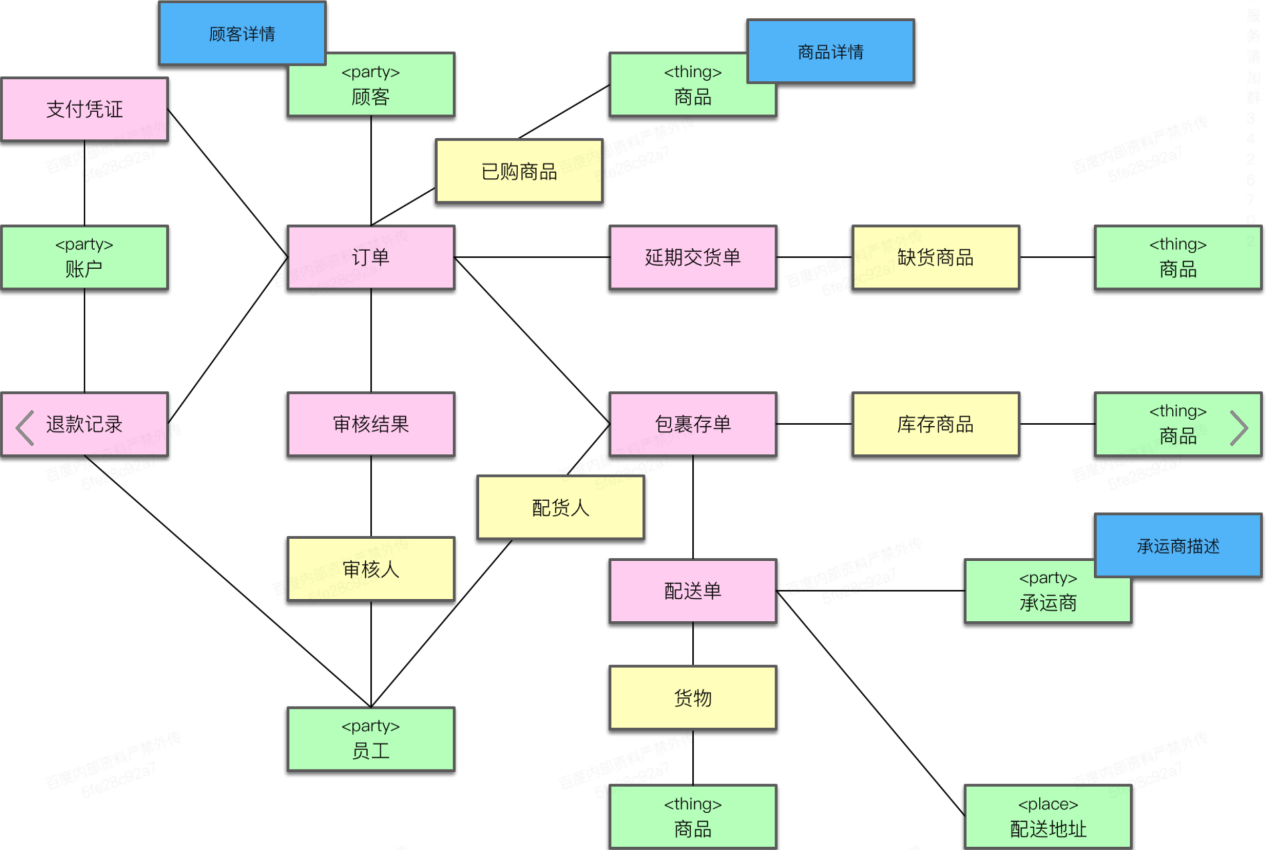
识别出上述MI对象涉及到的参与方、地点、物品。比如上述订单对象，需要有下订单的人，订单里的商品，人和商品就是PPT对象。



根据上一步识别出来的MI对象和PPT对象，找出它们之间的关系，抽取出角色就是Role对象。比如订单和人之间的角色是客户，发货凭证和人之间是库存管理员，这是不同的角色。



PPT对象往往还需要更多的补充信息，比如商品这个PPT对象可能还有供应商信息，上下架状态等，这些就是描述对象。



小结：业务流程是为了满足业务系统的目标，业务系统的目标是要要支撑企业的日常运营、商业决策。而运营管理过程中每个动作都会留下痕迹，增删改查每个动作都会有相应的记录或凭证。根据运营流程中最关心的东西，留下重要的那些记录、凭证，往往这就是MI对象。一旦识别出了业务流程中的MI对象就形成了模型分析的主干，再围绕这个主干延伸出PPT对象，根据MI对象和PPT对象的关系识别出角色对象，最后识别出需要补充PPT对象的信息。四色建模流程紧密围绕业务目标展开，能充分提炼出一个完整的领域知识，最终得到有效的领域模型，是一种推荐的分析方法。

### 模型设计阶段

设计阶段是开发人员跟PM、领域专家弄清楚了领域知识之后，实际编码之前进行的领域元素设计，起着承上启下作用。设计阶段在DDD的战术层面非常重要，因为对编码影响很大，对下一次业务需求迭代效率奠定基础。

目标：基于领域模型分析图，进一步简化对象关系、明确对象职责、设计出以聚合为核心的设计模型，划出第四重边界

参与角色：领域专家、开发、测试

何时发生：迭代开发周期内

产出：1. 维护领域设计模型图 2. 可能会修改新增本限界上下文的通用语言表

方法：基于模型分析图，做实体+值对象设计，识别聚合，各自理清依赖关系。辅助以时序图可以识别出领域服务。如何设计具体的领域对象，详见[DDD的设计原则](http://wiki.baidu.com/pages/viewpage.action?pageId=1233535631)。注意：得出设计模型图后需要返回需求，走查一遍具体需求，验证设计的领域模型是否合理。

领域模型设计图示例



上面这种由类图表达的领域设计模型仍有不足之处，它没有明确每个对象的职责范围大小合适不合适，需不需要领域服务。如果针对某个具体场景画时序图的话可以更直观看出哪个对象承担的职责过多或者过少，从而更合理的确定出对象协作关系。

特别要注意编码先后顺序的问题，分析阶段的顺序是自上而下，从主干流程、业务场景到领域模型，编码阶段是自下而上。1. 先实现领域层的逻辑，不必考虑技术细节比如存储细节，rpc调用。2. 领域层完成后，实现应用层，编排业务场景。3.最后是接口层和基础设施层，接口层其实也是基础设施层的一部分。

## DEMO

### 规划阶段

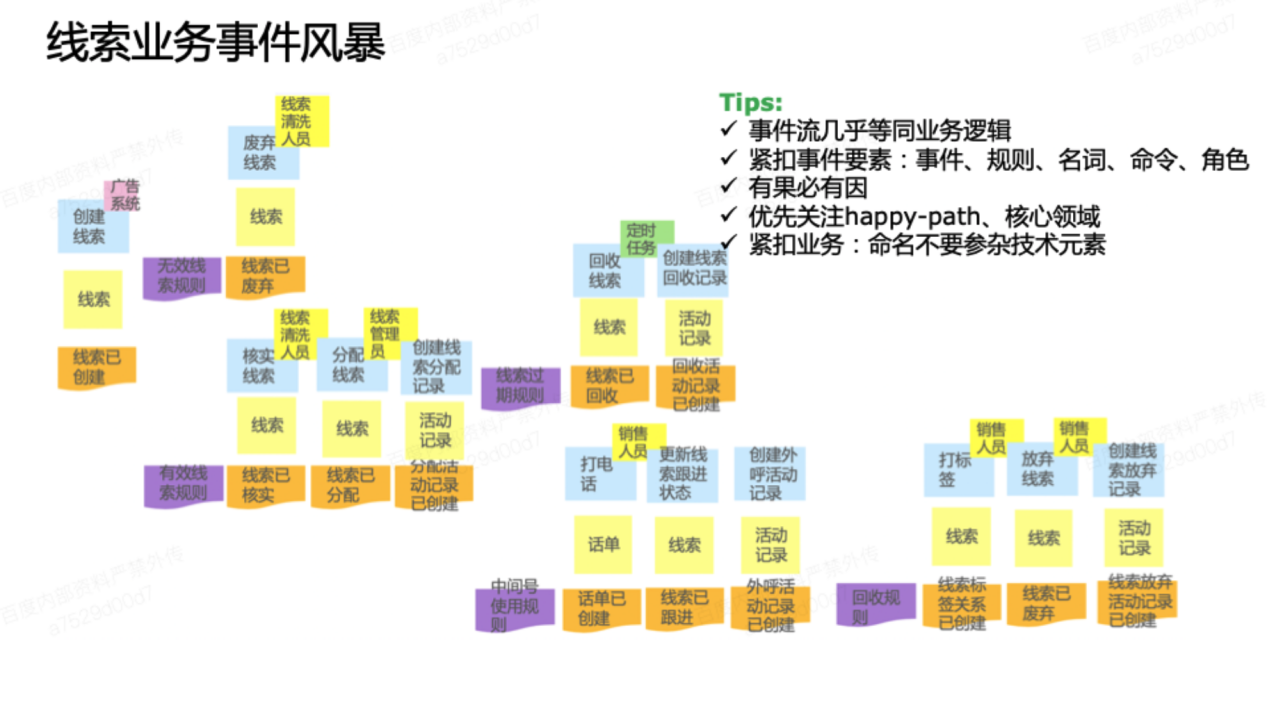
战略规划阶段在DEMO整个产品或业务层面：定义了产品的PMF(Product Market Fit)，产品愿景，划分了多个业务域，每个业务域制定了衡量产品成功的北极星指标。

### 分析阶段

分析阶段要对业务进行梳理抽象并可视化出来，让大家都理解要解决的问题和涉及的业务场景、业务流程、业务概念。比较有效的方法有事件风暴、四色建模。

事件风暴使用门槛低，该方法认为事件流很⼤程度上反映了现实业务逻辑，参与人员基于领域事件发生的时间线，把事件的前因后果逐步罗列出来。整个过程包含识别领域事件、决策命令、领域名词三个步骤。通过尝试回答这几个问题：这个业务涉及的系统产生了什么变化？变化由哪个角色通过什么方式触发的？系统变化产生了哪些结果？

#### 事件风暴



### 设计阶段

设计阶段把事件风暴的结果作为输入，用各领域对象分别承载业务场景、流程、概念、规则等。下图以销售域的线索模块举例。

