**「查缺补漏」，DDD 核心概念梳理**

<https://mp.weixin.qq.com/s/qca9foDun0HUvo7eS03W5g>

**一、概述**

DDD 是什么，DDD 的英文全称是 Domain-Driven Design，翻译过来就是领域驱动设计。

这种设计一般是用在微服务的系统中，当我们聊微服务的时候，争论最多的就是如何进行微服务的拆分，这也是最让人产生争议的地方。

当我们聊微服务也必然会会聊到中台，中台又是什么呢？

**中台**

中台从 2015 年提出，就已经被我们熟知，但是每个人对中台的认识可能都千差万别，有没有一个大家都比较认可的定义呢?

将通用的可复用的业务能力沉淀到中台业务模型，实现企业级能力复用。

因此中台面临的首要问题就是中台领域模型的重构。

而中台落地时，依然会面临微服务设计和拆分的问题。

* **微服务**：中台落地时需要用微服务进行支撑。
* **中台**：复用业务，实现企业级能力复用。
* **DDD**：对中台进行领域建模，实现适合企业发展的中台。

DDD 可以说是微服务和中台的产品经理。我们去写业务功能时，是面向领域的，而不是面向数据库表来实现代码的。

## 二、DDD 是什么？

DDD 的核心思想：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。

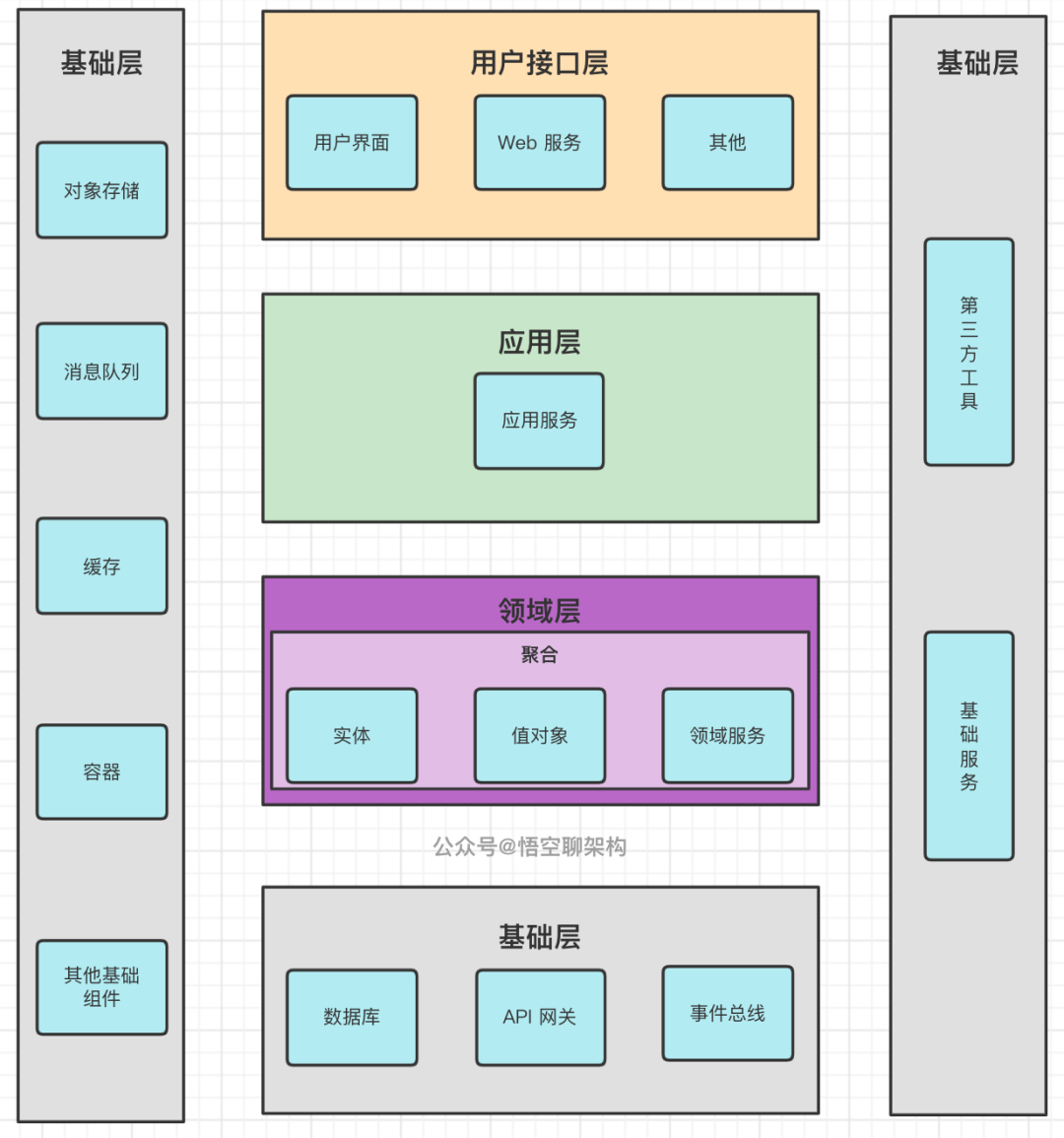
DDD 是一种处理高度复杂领域的设计思想，它试图分离技术实现的复杂性，并围绕业务概念构建领域模型来控制业务的复杂性，以解决软件难以理解，难以演进的问题。

**战略设计**：主要从业务视角出发，建立业务领域模型，划分领域边界，建立通用语言的限界上下文，限界上下文可以作为微服务设计的参考边界。

**战术设计**：则从技术视角出发，侧重于领域模型的技术实现，完成软件开发和落地，包括：聚合根、实体、值对象、领域服务、应用服务和资源库等代码逻辑的设计和实现。

## 三、DDD 架构分层

首先我们来看下架构分层的原理图：



### 用户接口层

用户接口层主要包含用户界面、Web 服务。

用户接口层负责向用户显示信息和解释用户指令。这里的用户可能是：用户、程序、自动化测试和批处理脚本等等。

### 应用层

应用层不应该有业务逻辑。它是很薄的一层，理论上不应该有业务规则或逻辑，主要面向用例和流程相关的操作。

应用服务是在应用层的，它负责服务的组合、编排和转发，负责处理业务用例的执行顺序以及结果的拼装，以粗粒度的服务通过 API 网关向前端发布。还有，应用服务还可以进行安全认证、权限校验、事务控制、发送或订阅领域事件等。

### 领域层

领域层主要实现企业的核心业务逻辑，和之前的三层架构的 Service 层很像。

领域层当中又包含聚合，聚合里面就带有聚合根、实体、值对象、领域服务等领域模型中的领域对象。

领域模型的业务逻辑主要通过实体和领域服务来实现，采用充血模型来时先所有与之相关的业务功能。充血模型后面会解释。当单一实体（或值对象）不能实现时，领域服务就来进行聚合多个实体（或值对象），来实现复杂的业务逻辑。

### 基础层

基础层为其他各层提供通用的技术和基础服务，包括数据库服务、消息中间件、对象存储、缓存服务等。

它是封装了所有的基础服务，当切换基础组件时，只用稍微修改下基础服务就可以了。

比如之前用的对象文件存储组件是阿里的，现在想换成腾讯的了，稍微改下基础服务，切换成腾讯的就可以了，不用去改业务逻辑代码。

这个就是采用了依赖倒置的原则，通过解耦来保持独立的核心业务逻辑。

### 传统三层架构转 DDD 四层架构

传统的三层架构就是 controller->service->model 这种模型，我们的思维习惯就是基于数据库的表来开发业务功能。这种分层架构给开发人员带来了便利，但是如果有其他人过来看你的代码，他会很难从业务角度去理解，因为这些代码都是为操作数据库的表而写。

有了 DDD 之后，代码是面向业务功能的，而不是面向数据库表的。

DDD 分层架构将业务逻辑层的服务拆分到了应用层和领域层。应用层快速响应前端的变化，领域层实现领域模型的能力。

三层架构数据访问采用 DAO 方式；DDD 分层架构的数据库等基础资源访问，采用了仓储（Repository）设计模式，通过依赖倒置实现各层对基础资源的解耦。

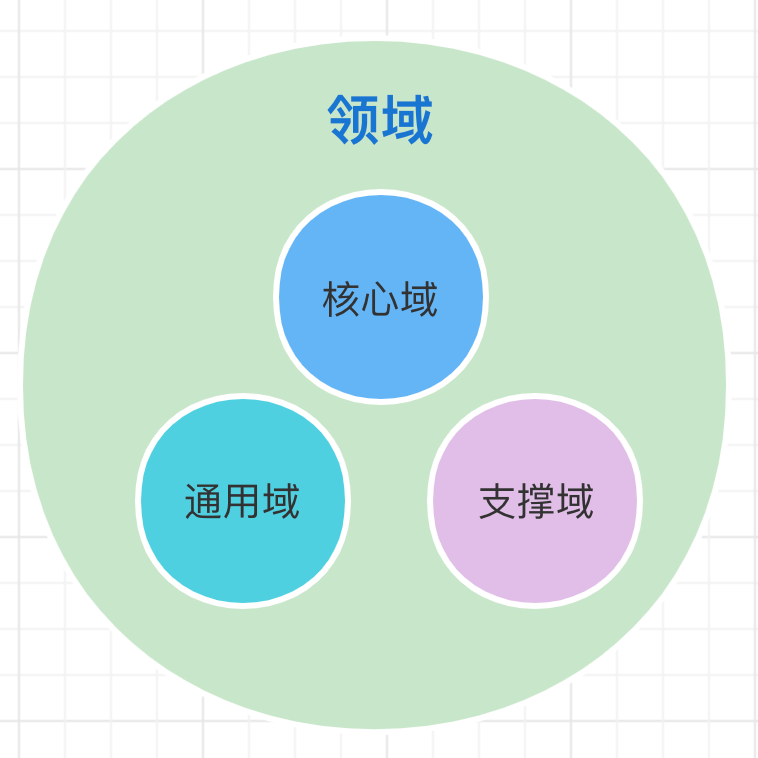
**四、DDD 中各种 Object**

* **数据持久化对象** (Persistent Object， PO)，与数据库结构一一映射，它是数据持久化过程中的数据载体。
* **领域对象**（ Domain Object， DO），微服务运行时核心业务对象的载体， DO 一般包括实体或值对象。
* **数据传输对象**（ Data Transfer Object， DTO），用于前端应用与微服务应用层或者微服务之间的数据组装和传输，是应用之间数据传输的载体。
* **视图对象**（View Object， VO），用于封装展示层指定页面或组件的数据。
* **微服务基础层**的主要数据对象是PO。在设计时，我们需要先建立DO和PO的映射关系。大多数情况下DO和PO是一一对应的。但也有DO和PO多对多的情况。在DO和PO数据转换时，需要进行数据重组。对于DO对象较多复杂的数据转换操作，你可以在聚合用工厂模式来实现。当DO数据需要持久化时，先将DO转换为PO对象，由仓储实现服务完成数据库持久化操作。当DO需要构建和数据初始化时，仓储实现服务先从数据库获取PO对象，将PO转换为DO后，完成DO数据构建和初始化。
* **领域层**主要是DO对象。DO是实体和值对象的数据和业务行为载体，承载着基础的核心业务逻辑，多个依赖紧密的DO对象构成聚合。领域层DO对象在持久化时需要转换为PO对象。
* **应用层**主要对象有DO对象，但也可能会有DTO对象。应用层在进行不同聚合的领域服务编排时，一般建议采用聚合根ID的引用方式，应尽量避免不同聚合之间的DO对象直接引用，避免聚合之间产生依赖。
* 在涉及跨微服务的应用服务调用时，在调用其他微服务的应用服务前，DO会被转换为DTO，完成跨微服务的DTO数据组装，因此会有DTO对象。在前端调用后端应用服务时，用户接口层先完成DTO到DO的转换，然后DO作为应用服务的参数，传导到领域层完成业务逻辑处理。
* 用户接口层主要完成DO和DTO的互转，完成微服务与前端应用数据交互和转换。facade接口服务在完成后端应用服务封装后，会对多个DO对象进行组装，转换为DTO对象，向前端应用完成数据转换和传输。facade接口服务在接收到前端应用传入的DTO后，完成DTO向多个DO对象的转换，调用后端应用服务完成业务逻辑处理。前端应用主要是VO对象。展现层使用VO进行界面展示，通过用户接口层与应用层采用DTO对象进行数据交互。

## 五、领域分类

在研究和解决业务问题时，DDD 会按照一定的规则将业务领域进行细分，当领域细分到一定的程度后，DDD 会将问题范围限定在特定的边界内，在这个边界内建立领域模型，进而用代码实现该领域模型，解决相应的业务问题。简言之，DDD 的领域就是这个边界内要解决的业务问题域。

领域又可以分为多个子域，子域又包含核心域、通用域和支撑域。

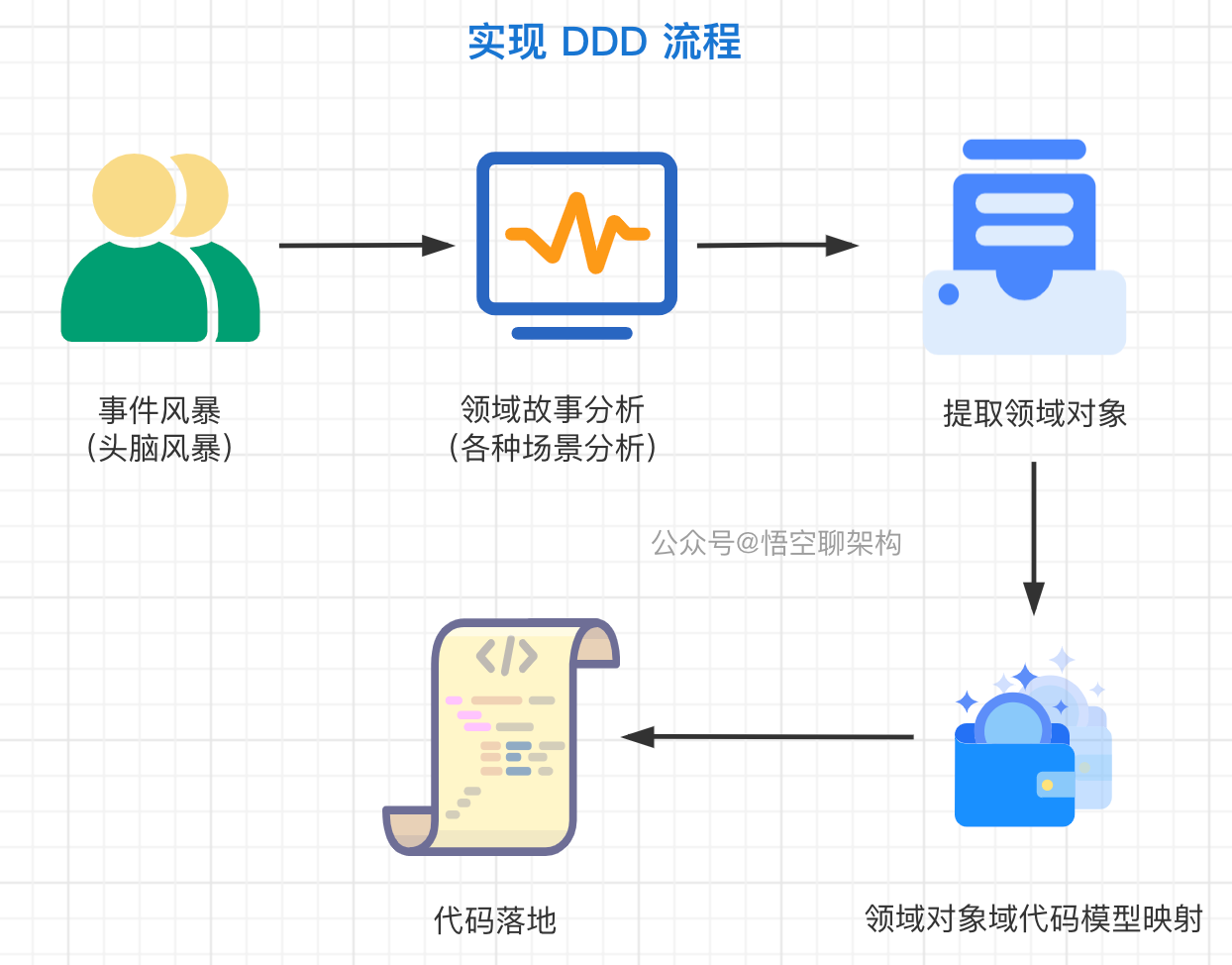


**核心域**：核心业务，决定产品和公司核心竞争力的子域。

**通用域**：同时被多个子域使用的通用功能子域。

**支撑域**：支持其他子域，非核心域和通用域。

## 六、实现 DDD 流程



* **第一步**：事件风暴，这里的风暴可以理解为头脑风暴，领域专家会和设计、开发人员一起建立领域模型。
* **第二步**：对领域中涉及到的场景（用户故事）进行分析。
* **第三步**：分析了场景之后，就要定义领域对象。设计实体、找出聚合根、设计值对象、设计领域事件、设计领域服务、设计仓储。
* **第四步**：领域对象需要包含业务逻辑，所以会形成一个代码模型的映射。
* **第五步**：根据代码模型进行代码落地。

**六、限界上下文和通用语言**

**限界上下文**

领域边界就是通过限界上下文来定义的。

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。

理论上限界上下文就是微服务的边界。我们将限界上下文内的领域模型映射到微服务，就完成了从问题域到软件的解决方案。

如果不考虑技术异构、团队沟通等其它外部因素，一个限界上下文理论上就可以设计为一个微服务。

* 逻辑边界：微服务内聚合之间的边界是逻辑边界。它是一个虚拟的边界，强调业务的内聚，可根据需要变成物理边界，也就是说聚合也可以独立为微服务。
* 物理边界：微服务之间的边界是物理边界。它强调微服务部署和运行的隔离，关注微服务的服务调用、容错和运行等。
* 代码边界：不同层或者聚合之间代码目录的边界是代码边界。它强调的是代码之间的隔离，方便架构演进时代码的重组。

### 通用语言

DDD 分析和设计过程中的每一个环节都需要保证限界上下文内术语的统一，在代码模型设计的时侯就要建立领域对象和代码对象的一一映射，从而保证业务模型和代码模型的一致，实现业务语言与代码语言的统一。

## 七、实体

### 实体概念

实体和值对象是组成领域模型的基础单元。

类包含了实体的属性和方法，通过这些方法实现实体自身的业务逻辑。

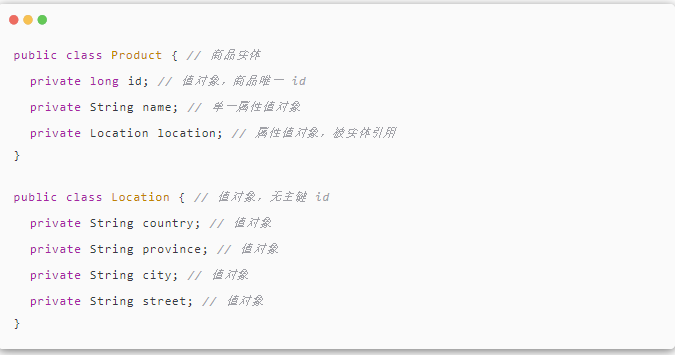
实体以 DO（领域对象）的形式存在，每个实体对象都有唯一的 ID。字段的值可以变。

实体是看得到、摸得着的实实在在的业务对象，实体具有业务属性、业务行为和业务逻辑。

### 实体特点

有 ID 标识，通过 ID 判断相等性，ID 在聚合内唯一。依附于聚合根，生命周期由聚合根管理。实体一般会持久化，但是与数据库持久化对象不一定是一对一的关系。实体可以引用聚合内的聚合根、实体和值对象。

如下代码所示，Product 属于商品实体，有商品唯一 id。Location 属于值对象，后面会讲解值对象。



实体类通常采用充血模型。

### 充血模型和贫血模型的区别

* 贫血模型：数据和业务逻辑分开到不同的类中，比如 Model 类和 Service 类。
* 充血模型：数据和业务逻辑封装在同一个实体类中。

## 八、值对象

### 值对象概念

值对象描述了领域中的一件东西，这个东西是不可变的，它将不同的相关属性组合成了一个概念整体。

值对象是 DDD 领域模型中的一个基础对象，它跟实体一样都来源于事件风暴所构建的领域模型，都包含了若干个属性，它与实体一起构成聚合。

**值对象只是若干个属性的集合，只有数据初始化操作和有限的不涉及修改数据的行为，基本不包含业务逻辑。**值对象的属性集虽然在物理上独立出来了，但在逻辑上它仍然是实体属性的一部分，用于描述实体的特征。

### 值对象的特点

无 ID，不可变，无生命周期，用完就不需要了。值对象之间通过属性值判断相等性。核心本质是值，是一组概念完整的属性组成的集合，用于描述实体的状态和特征，值对象尽量只引用值对象。

## 九、聚合和聚合根

### 聚合

聚合就是由业务和逻辑紧密关联的实体和值对象组合而成。

聚合是数据修改和持久化的基本单元，每一个聚合对应一个仓储，实现数据的持久化。

聚合有一个聚合根和上下文便捷，根据业务单一职责和高内聚原则，定义了聚合内部应该包含哪些实体和值对象，而聚合之间的边界是松耦合的。

聚合属于 DDD 领域层，领域层包含多个聚合，共同实现核心业务逻辑。

聚合内的实体以充血模型实现个体业务能力，以及业务逻辑的高内聚。

跨多个实体的业务逻辑通过领域服务来实现，跨多个聚合的业务逻辑通过应用服务来实现。

**特点**：高内聚、低耦合，它是领域模型中最底层的边界，可以作为拆分微服务的最小单位，但是不建议对微服务过度拆分。一个聚合可以作为一个微服务，以满足版本的高频发布和极致的弹性伸缩能力。一个微服务也可以包含多个聚合，可以进行拆分和组合。

### 聚合根

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

聚合可以比作组织，聚合根就是这个组织的负责人。

外部对象不能直接访问聚合内实体，需要先访问聚合根，再导航到聚合内部实体。

**特点**：聚合根是实体，有实体的特点，具有全局唯一标识，有独立的生命周期。一个聚合只有一个聚合根，聚合根在聚合内对实体和值对象采用直接对象引用的方式进行组织和协调，聚合根和聚合根之间通过 ID 关联的方式实现聚合之间的协同。

## 十、领域事件

领域事件用来表示领域中发生的事件。一个领域事件将导致进一步的业务操作，在实现业务解耦的同时，有助于形成完成的业务闭环。

领域事件驱动设计可以切断领域模型之间的强依赖关系，事件发布完成后，发布方不必关心后续订阅方事件处理是否成功，可以实现领域模型的解耦，维护领域模型的独立性和数据的一致性。微服务之间的数据不必要求强一致性，而是基于事件的最终一致性。

领域事件的执行需要一系列的组件和技术来支撑：事件的构建和发布、事件数据持久化、事件总线、消息中间件、事件接收和处理。