实现领域驱动设计

广州盛原成科技有限公司

提　纲

1. 前言
2. 理论与方法
3. 分层架构
4. 通用原则
5. 最佳实践

一、前言

总述

随着**应用程序的变化**,有时候,为了节省开发时间会**违反一些本应遵守的规则**,使得代码变得复杂且难以维护.

短期来看确实节省了开发时间,但是后期可能需要花费更多的时间为之前的偷懒而**买单.无法对原有的代码进行维护**,导致大量的逻辑都需要进行**重写**.

目标

介绍并解释DDD的架构,概念,原理及构建.

解释ABP框架的分层架构及解决方案结构.

通过案例,介绍实现DDD的一些规则及最佳实践.

展示ABP框架为DDD的实现提供了哪些基础设施.

最后,基于软件开发最佳实践和我们的经验提供建议来创建一个可维护的代码库.

为什么需要DDD？

程序员其实不是在编写代码，而是在摸索业务领域知识

这是很多程序员的真实写照，尤其复杂业务，为什么呢？

很多时候你会遇到这样的情况：**一个函数写了几百行，里面的if-else写了一大堆，计算各种业务规则。另一个人接手之后，分析了好几天，才把业务逻辑彻底理清楚。**

这个问题从表面来看，是代码写的不规范，要重构，把一个几百行的函数拆成一个个小的函数。从根本上来讲，就是**“重要逻辑”隐藏在代码里面，没有“显性”的表达出来**。

这只是一个函数，推而广之，到类、到模块、到系统，是同样的道理，比如：

业务流程隐藏在多个对象的复杂调用关系里面；某个业务核心概念没有提取出来，其职责分摊到了其他几个实体里面；系统耦合，职责边界不清

所以，建模的本质就是把“**重要的东西进行显性化，并进而把这些显性化的构造块，互相串联起来，组成一个体系**”

而DDD就是为了解决这些痛点。

什么是DDD?

领域驱动设计(DDD)是一种将实现与**持续进化**的模型连接在一起来满足**复杂**需求的**软件开发方法**.

DDD适用于**复杂领域**或**较大规模**的系统,而不是简单的CRUD程序.它着重与**核心领域逻辑**,而不是基础架构.这样有助于构建一个**灵活**,模块化,**可维护**的代码库.

领域驱动设计不是架构方法，也并非设计模式。可以把DDD理解为一种方法论，一种构造软件模型的方法论。

OOP & SOLID

实现DDD高度依赖面对对象编程思想(OOP)和SOLID原则.事实上,DDD已经**实现**并**延伸**了这些原则,因此,**深入了解**OOP和SOLID对实施DDD十分有利.

ABP是什么?

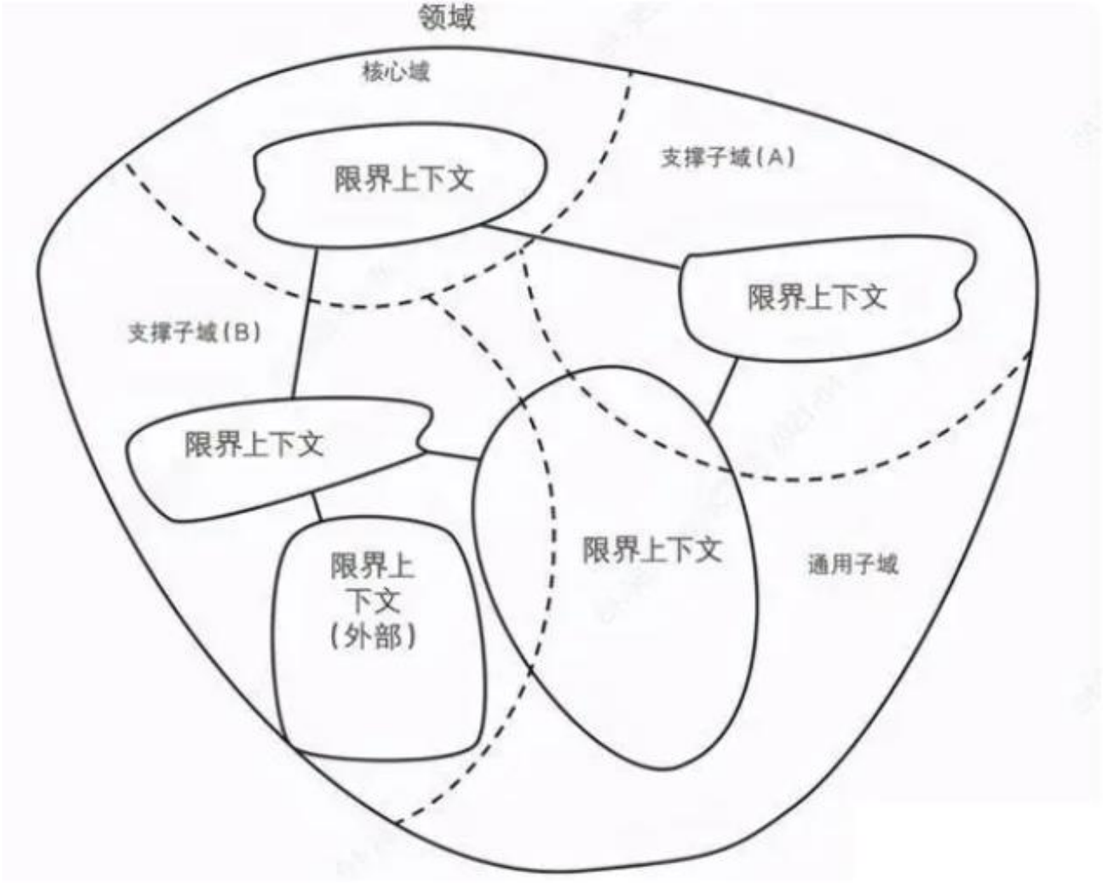
ABP 框架是一个完整的基础架构，用于通过遵循软件 开发最佳实践和约定来创建现代 Web 应用程序。 借助 ABP 的高级框架和生态系统，可以 实现 Don't Repeat Yourself (DRY) 原则并专注于业务代码。

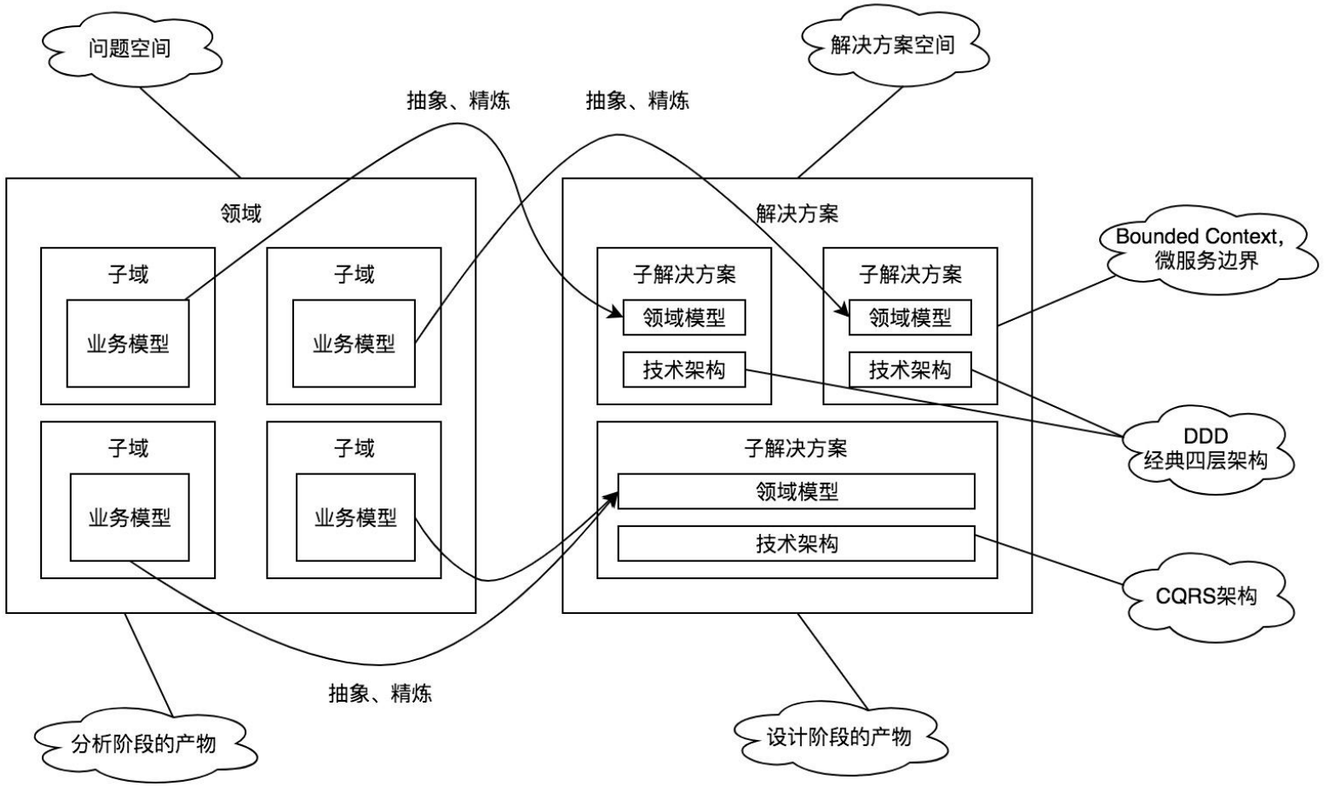
ABP框架提供了基础设施使基于领域驱动设计的开发更易实现.

二、理论与方法

DDD术语介绍

学习DDD前，有很多基本的概念需要理解，主要包括：领域、子域、核心域、通用域、支撑域、实体、值对象、聚合、聚合根、通用语言、限界上下文、事件风暴、领域事件、领域服务、应用服务、工厂、资源库。[一文带你掌握 DDD基础]



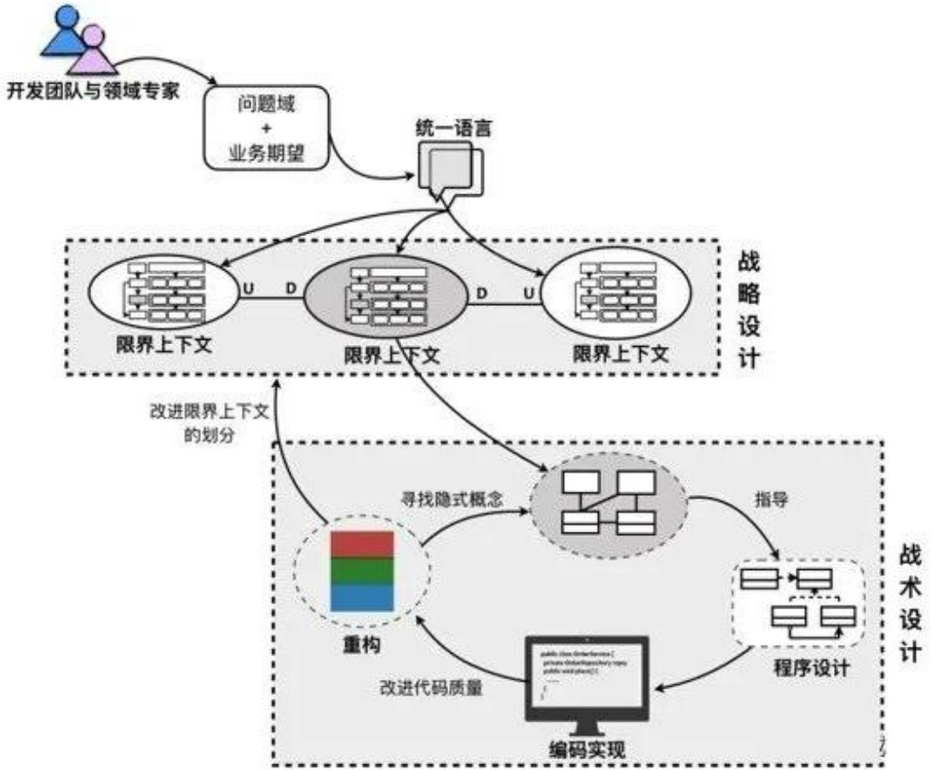


战略战术设计

DDD领域驱动设计通常会包含战略设计和战术设计两部分：

战略设计：重业务建模，以业务视角，拆分领域，通过事件风暴（从发散到收敛过程），梳理业务并构建领域模型，这块过程会涉及业务人员、产品人员、架构师等多方参与

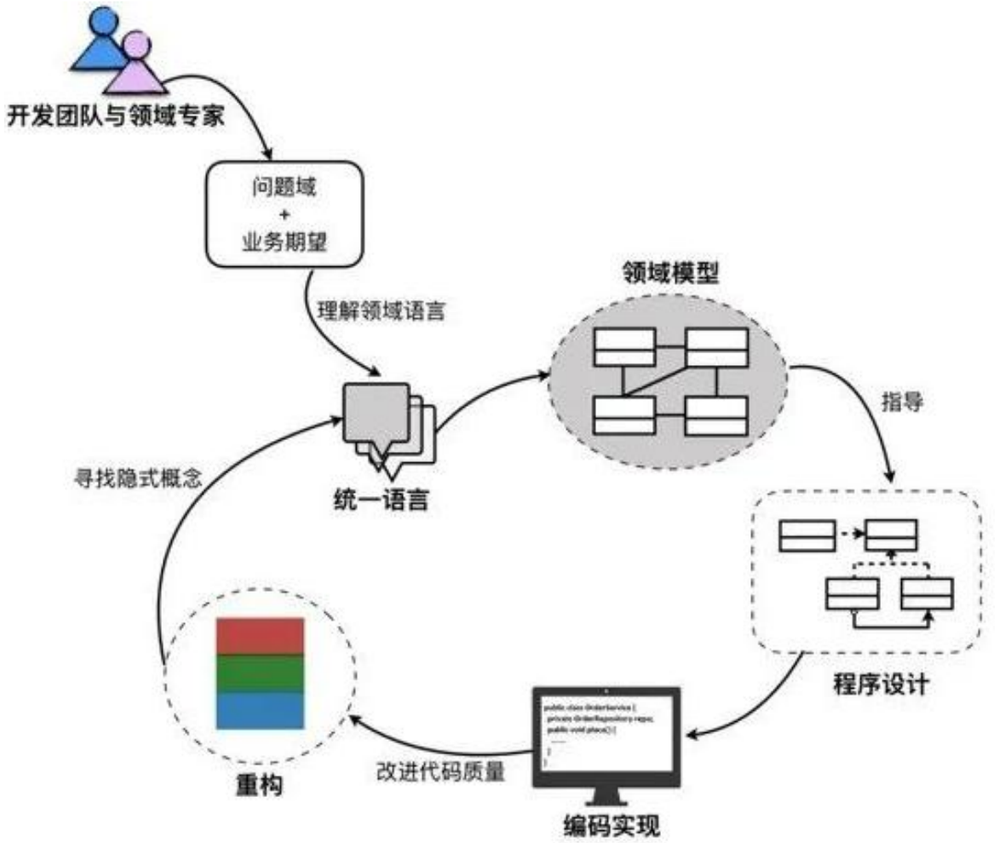
战术设计：重落地实现，以构建的领域模型，建立了领域模型的边界与上下文，也就确认了服务的边界，这个过程会涉及架构师、技术人员参与



宏观层次是战略的层面，微观层次是战术的层面，只有将战略设计与战术设计结合起来，才是完整的领域驱动设计。

领域驱动设计

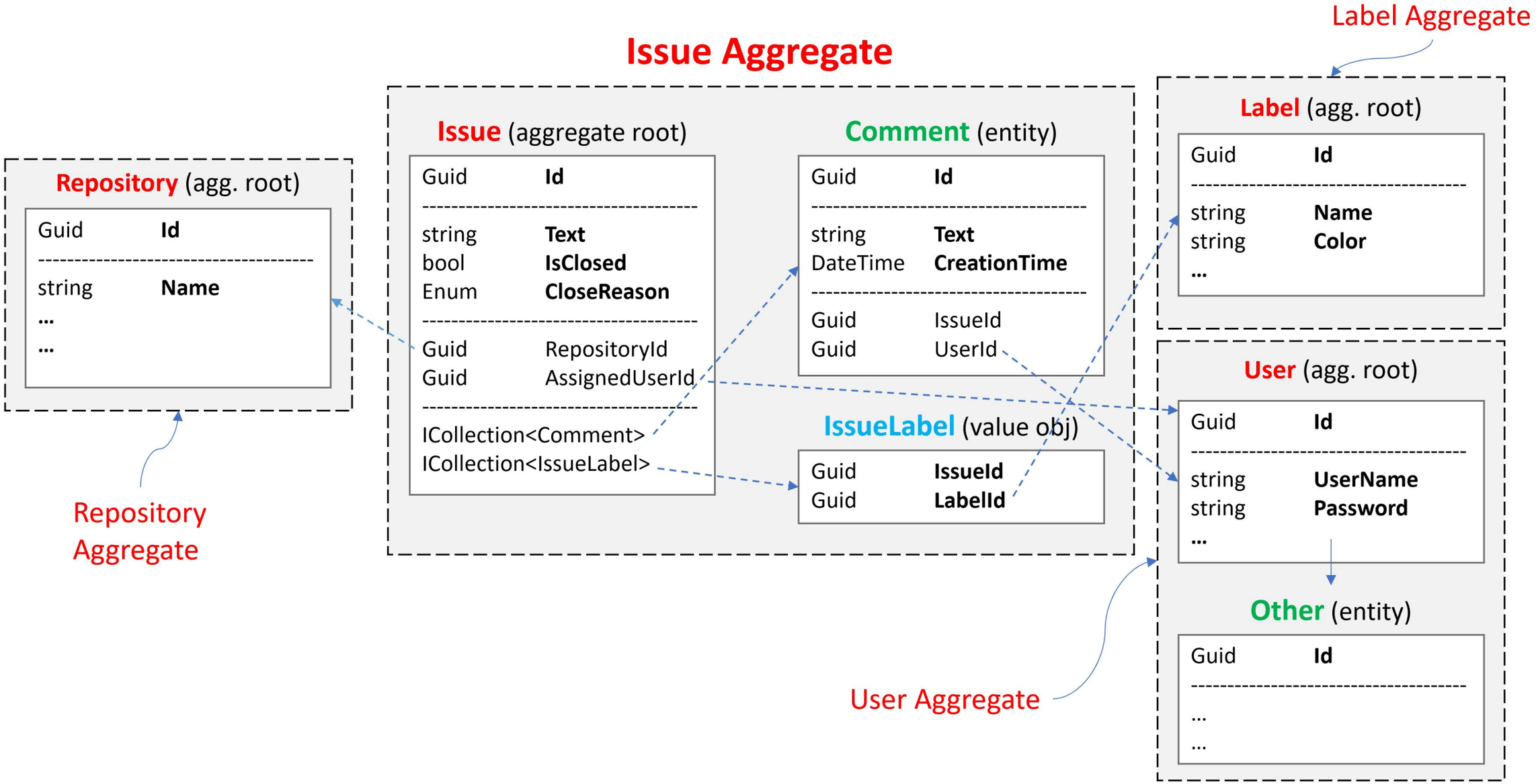
领域驱动设计强调领域模型的重要性，并通过模型驱动设计来保障领域模型与程序设计的一致。从业务需求中提炼出统一语言（Ubiquitous Language），再基于统一语言建立领域模型；
这个领域模型会指导着程序设计以及编码实现；最后，又通过重构来发现隐式概念，并运用设计模式改进设计与开发质量。
这个过程如下图所示：



领域驱动设计贯穿了整个软件开发的生命周期，
包括对需求的分析、建模、架构、设计，甚至最终的编码实现，乃至对编码的测试与重构。

领域模型

下图中展示了一些聚合,聚合根,实体,值对象以及它们之间的关系:Issue(问题), Repository(仓库), Label(标签) 和User(用户).



术语表

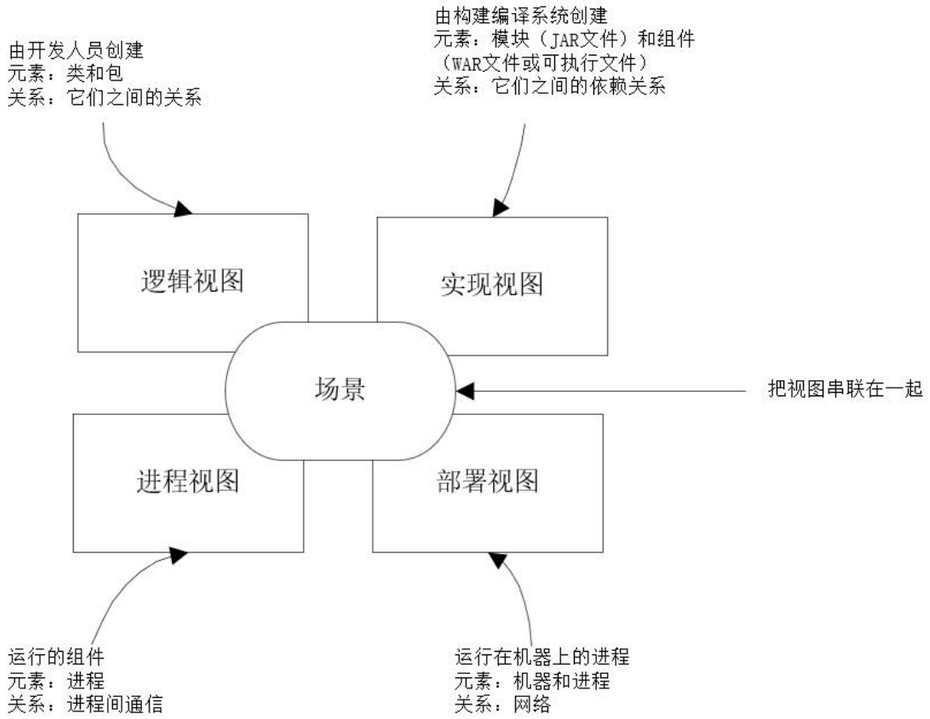
范围	术语名称（中文）	术语名称（英文）	描述
解决方案空间	领域驱动设计	DDD	Domain Driven Design，领域驱动设计，是一种软件设计开发的方法论。
	模型驱动设计	MDD	Model Driven Development，模型驱动开发，属于领域驱动开发的战术设计部分精要
	领域原语	DP	Domain Primitive 领域原语
	限界上下文	Bounded Context	服务职责划分的边界。
	上下文映射	Context Map	上下文映射则建立了限界上下文之间的关系
	聚合	Aggregate	Aggregate(聚合）是一组相关对象的集合，作为一个整体被外界访问，聚合根（Aggregate Root）是这个聚合的根节点。
	聚合根	Aggregate	Aggregate(聚合）是一组相关对象的集合，作为一个整体被外界访问，聚合根（Aggregate Root）是这个聚合的根节点。
	实体	Entity	DDD中定义的抽象业务对象，类似Java开发中的Business Object(BO)
	仓储	Repository	DDD中定义的业务领域模型下的数据源概念、仓储、资源库
	数据传输对象	DTO	Data Transfer Object，用于各个模块间简单数据的传送
	数据访问对象	DAO	Data Acces Object，用于数据处理的接口定义
	持久化对象	PO	Persistent Object，持久对象，这里用于描述数据库中特定table
	数据对象/领域对象	DO	Data Object/Domain Object，数据对象和领域对象都是用这个词，其中Data Object类型PO，domain object包括聚合、实体和值对象
	视图对象	VO	View Object，视图对象，为ui提供数据，用于展示层，它的作用是把某个指定页面（或组件）的所有数据封装起来
	工厂	Factory	工厂，使用它来创建领域对象
	命令查询分离架构	CQRS	命令查询分离架构 Command&Query Responsibility Segregation，读写分离在服务接口上的分类
	命令	Command/Cmd	CQRS架构中的命令，主要是写操作
	查询	Query/Qry	CQRS架构中的查询，主要是读操作
	端口适配器	Port&Adapter	接口和实现的一种设计模式命名，六边形架构的核心概念
	依赖倒置	DIP	SOLID原则之一
	控制反转	IOC	控制反转，一种面向对象编程中的设计原则
	依赖注入	DI	控制反转的实现方式，就是依赖是从外部传入或者查询的
	单例	Singleton	单例模式，一种设计模式
	领域事件	Domain Events	领域事件是一个领域模型中极其重要的部分，用来表示领域中发生的事件。忽略不相关的领域活动
	转换器	Assembler	转换器
	模块	Modules	模块
	防腐层	ACL	防腐层（Anti-Corruption Layer，ACL）
	失血模型	~	Domain Object 只有属性的 getter/setter 方法的纯数据类，所有的业务逻辑完全由 business object 来完成
	贫血模型	anaemic domain model	业务逻辑比较少的在domain层
	充血模型	rich domain model	业务逻辑都集中在domain层
	胀血模型	~	去掉application层，所有的转换和逻辑都在domain层
	Feign	Feign	Feign 是一个HTTP 请求的轻量级客户端框架
	事件溯源	Event Sourcing	事件溯源
	事件驱动架构	EDA	事件驱动架构(Event Driven Architecture，EDA)
问题空间	领域	Domain	切分的服务。领域就是范围。范围的重点是边界。领域的核心思想是将问题逐级细分来减低业务和系统的复杂度，这也是 DDD 讨论的核心
	子域	SubDomain	子服务
	核心域	CoreDomain	核心服务
	支撑域	Supporting (Sub)Domain	企业公共服务。
	通用域	Generic (Sub)Domain	中间件服务或第三方服务。
	统一语言	Ubiquitous Language	统一概念

三、分层架构

什么是架构的风格

架构风格**确定**可以在该风格的实例中使用的**组件**和**连接器**词汇表，以及关于**如何组合它们的一组约束**。应用程序通常使用**多种架构风格**的组合。

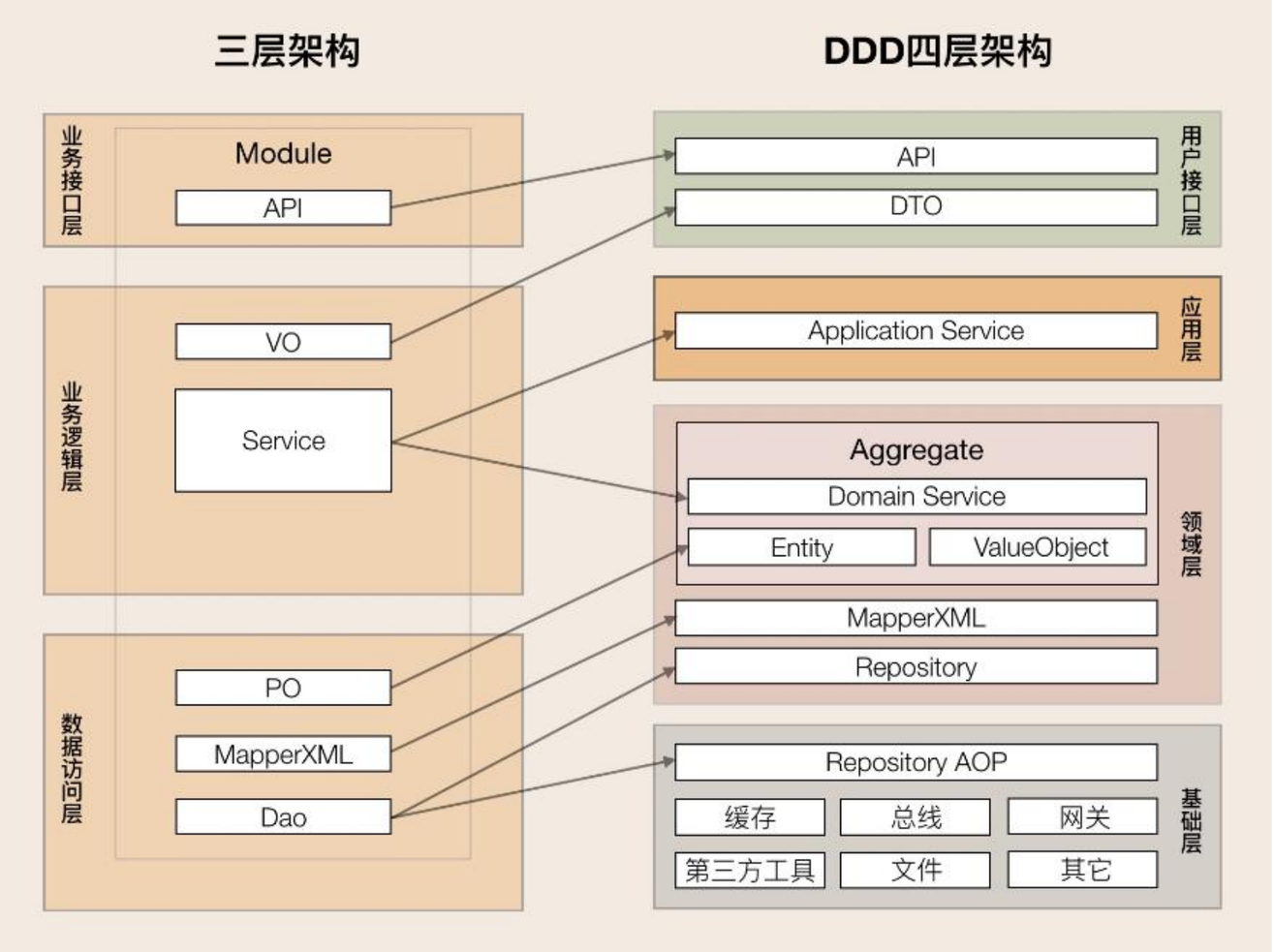
- 分层架构风格
- 六边形架构风格
- 单体架构
- 微服务架构



架构设计的最高原则，即建立一个高内聚松耦合的软件系统架构

分层架构

分层的本质是**关注点分离**，**隔离**对下层的变化。由于层间**松散的耦合**关系，使得我们可以**专注**于本层的设计，而不必关心其他层的设计，也不必担心当前层的设计会影响其它层。



传统的三层架构和 DDD 的四层架构

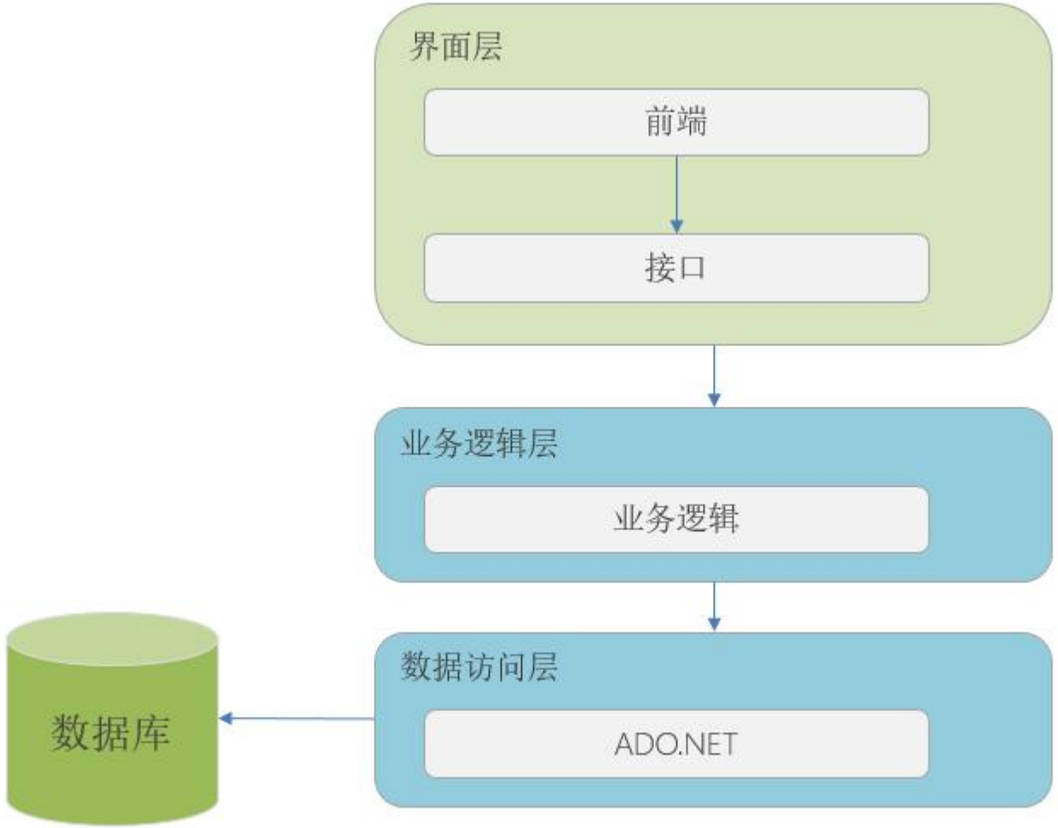
传统三层架构

问题：

过分注重数据访问层，而不重视领域。

业务逻辑直接与数据访问层耦合，与领域为核心的 DDD 思想背道而驰。

没有一系列的模式与方法论指导这种分层架构的开发约束。



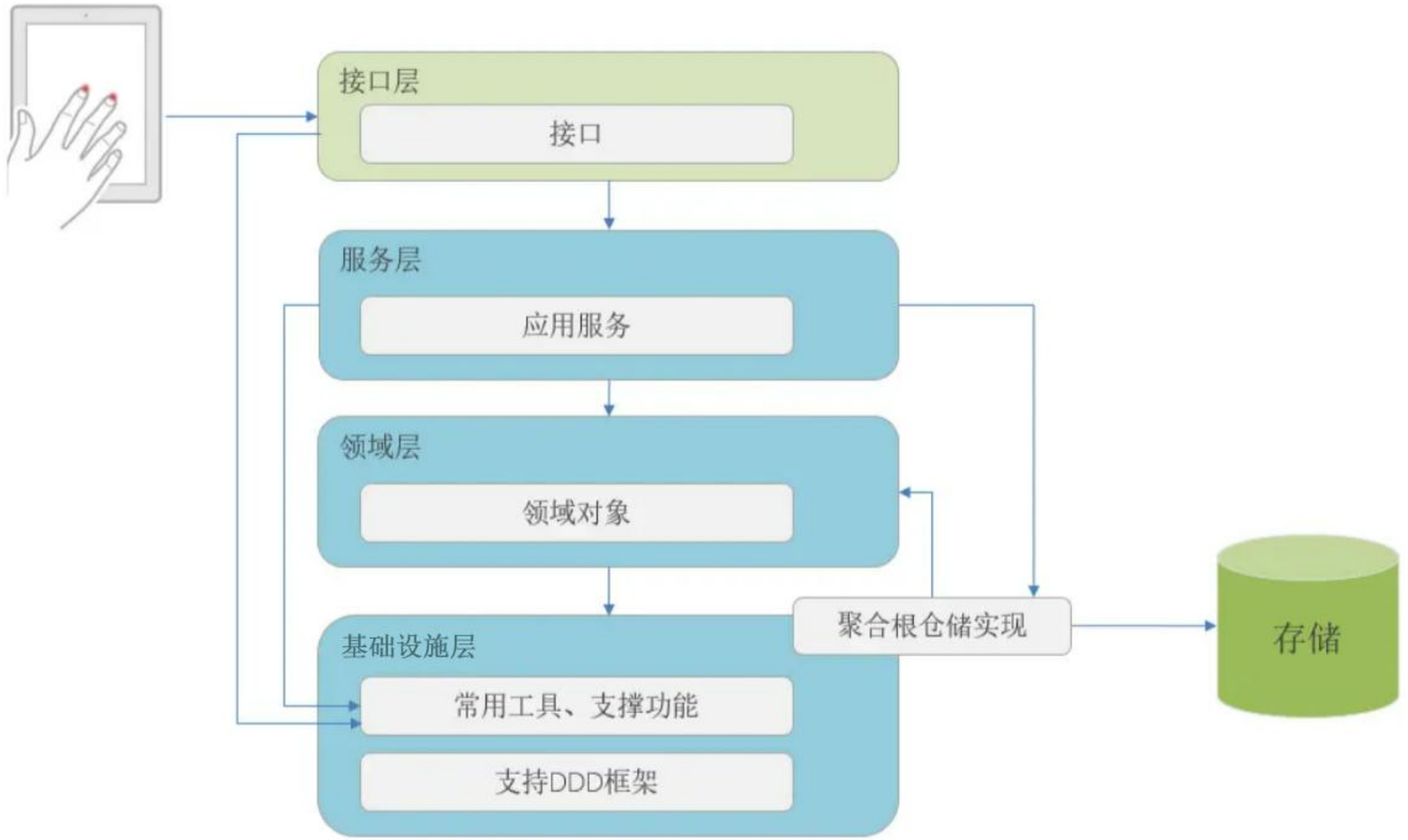
传统三层架构

经典DDD分层

DDD 分层架构包括：展现层、应用层、领域层和基础设施层。

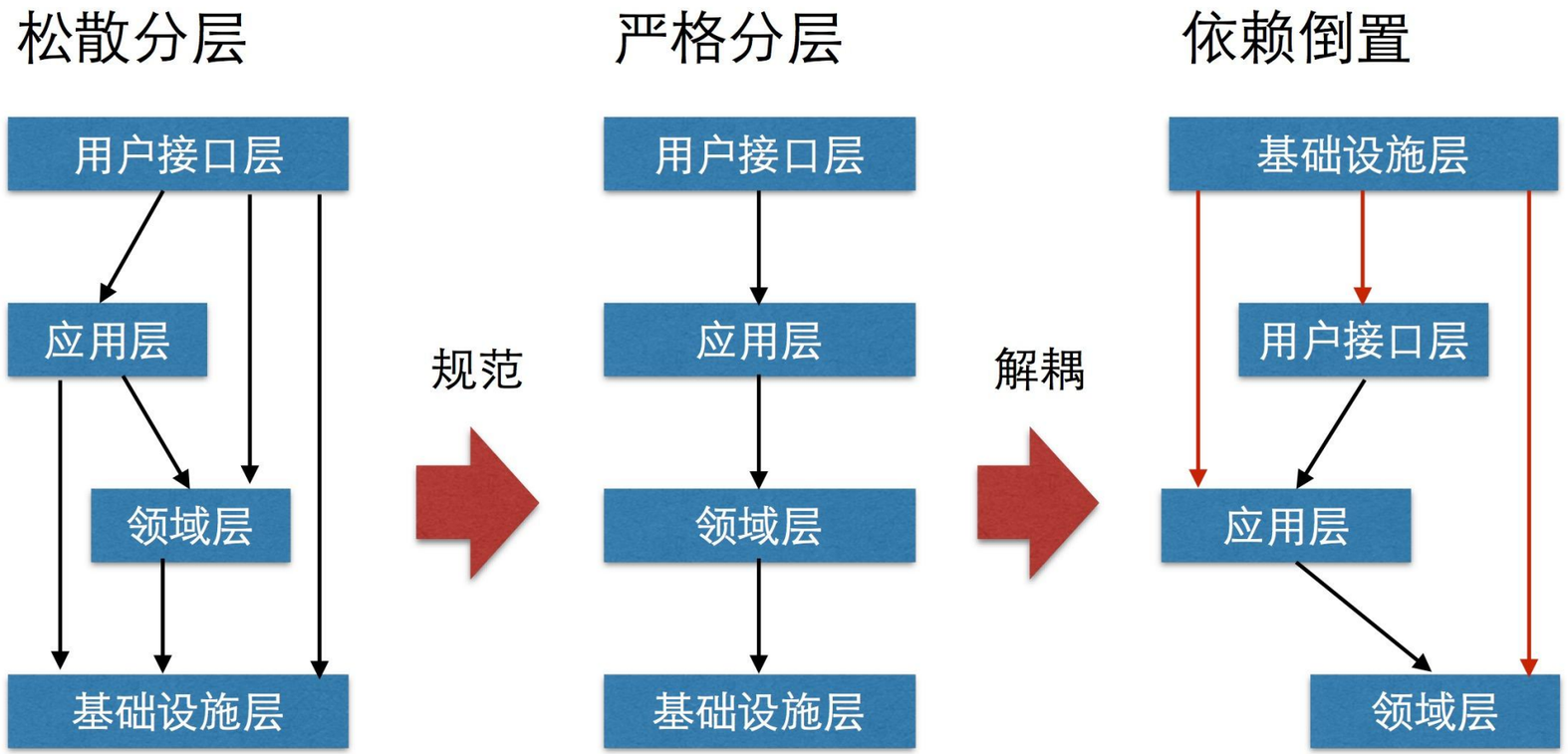
业务逻辑分为两层,分别为 领域(Domain) 层和 应用(Application) 层,它们包含不同类型的业务逻辑.

- **领域层**:只实现领域业务逻辑,与用例无关.
- **应用层**:基于领域层来实现满足用例的业务逻辑.用例可以看作是用户界面(UI)或外部应用程序的交互.
- **用户接口层**:包含应用程序的UI元素、API接口.
- **基础设施层**:通过对第三方库的集成或抽象,来满足其它层的非核心业务逻辑的实现.



严格分层架构和松散分层架构

为了服务的可管理，建议采用**严格分层架构**。
在严格分层架构中，**领域服务只能被应用服务调用**，而**应用服务只能被用户接口层调用**，服务是逐层对外封装或组合的，依赖关系清晰。而在松散分层架构中，领域服务可以同时被应用层或用户接口层调用，服务的依赖关系比较复杂且难管理，甚至容易使核心业务逻辑外泄。



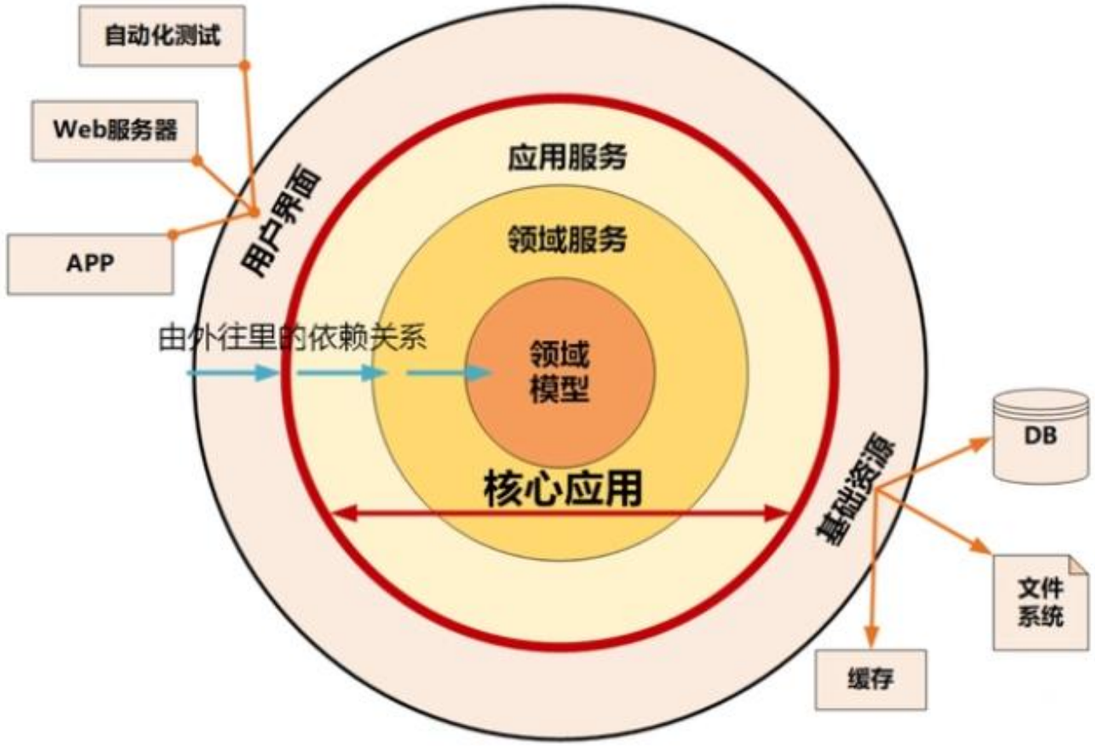
管理混乱

相邻层才可耦合
禁止跨层调用

层之间依赖抽象
层次扁平、松耦合

整洁架构

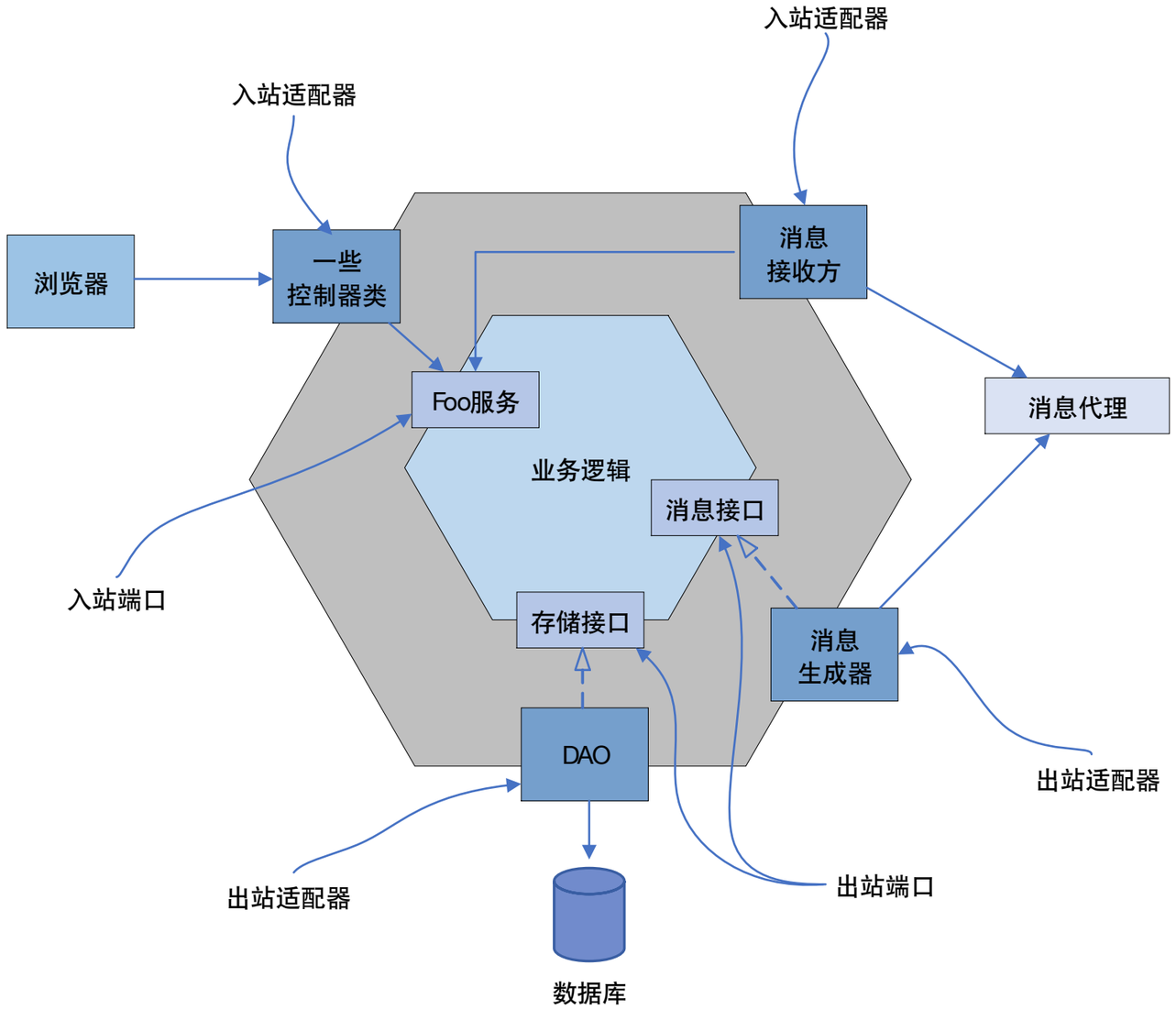
同样的分层架构也可以如下图所示:被称为 **整洁架构**, 又或者称为 **洋葱架构**:



整洁架构最主要的原则是依赖原则，它定义了各层的依赖关系，**越往里依赖越低**，代码级别越高，越是核心能力。

- 最内层的是领域模型,封装了业务规则,**不依赖任务**其他组件.
- 向外都是接口层.
- 领域模型就是业务逻辑的模型，它应该是完全纯粹的，无论你选择什么框架，什么数据库，或者什么通信技术，按照整洁架构的思想都不应该去污染领域模型

六边形架构

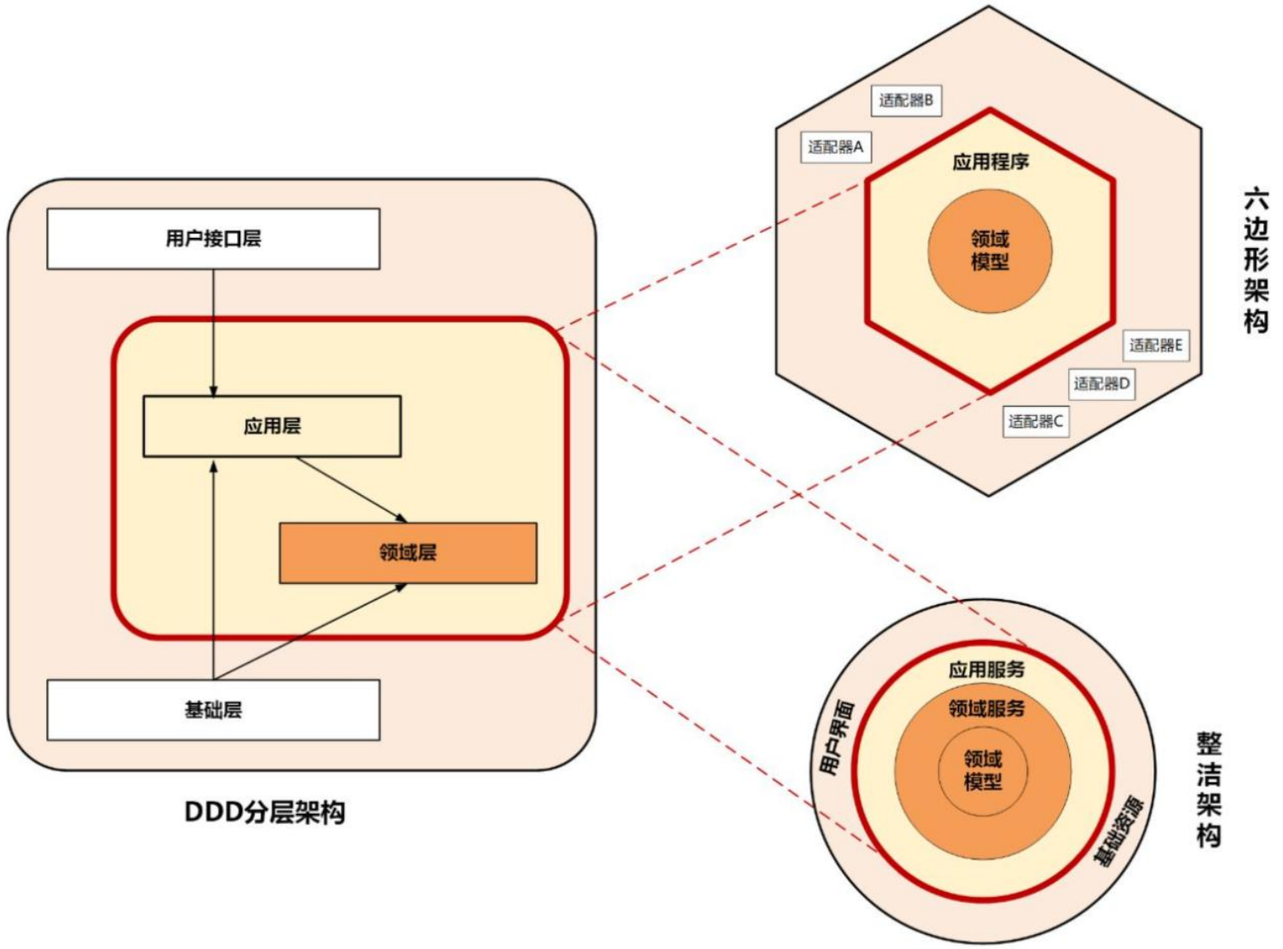


六边形架构的一个示例
它由业务逻辑和一个或多个与外部系统通信的适配器组成。
业务逻辑具有一个或多个端口。
处理来自外部系统请求的入站适配器调用入站端口。
出站适配器实现出站端口，并调用外部系统。

除了最中心的领域模型,数据库等基础设施可看作是南向网关,Controller(REST Services)是北向网关.内层领域模型通过接口和外部交互,业务汇聚在领域模型中.
输入-->领域模型--->输出.

DDD核心构建组成

- DDD的关注点在领域层和应用层上,而展现层和基础设施层则视为细节(这个词原文太抽象,自己体会吧),业务层不应依赖它们.
- 这并不意味着展现层和基础设施层不重要.它们非常重要,但UI框架 和 数据库提供程序 需要你自己定义规则和总结最佳实践.这些不在DDD的讨论范围中.



分层架构、整洁架构、六边形架构的对比和分析

领域层

- **实体(Entity):** 实体是种领域对象,它有自己的属性(状态,数据)和执行业务逻辑的方法.实体由唯一标识符(Id)表示,不同ID的两个实体被视为不同的实体.
- **值对象(Value Object):** 值对象是另外一种类型的领域对象,使用值对象的属性来判断两个值对象是否相同,而非使用ID判断.如果两个值对象的属性值全部相同就被视为同一对象.值对象通常是不可变的,大多数情况下它比实体简单.
- **聚合(Aggregate) 和 聚合根(Aggregate Root):** 聚合是由聚合根包裹在一起的一组对象(实体和值对象).聚合根是一种具有特定职责的实体.
- **仓储(Repository) (接口):** 仓储是被领域层或应用层调用的数据库持久化接口.它隐藏了DBMS的复杂性,领域层中只定义仓储接口,而非实现.
- **领域服务(Domain Service):** 领域服务是一种无状态的服务,它依赖多个聚合(实体)或外部服务来实现该领域的核心业务逻辑.
- **规约(Specification):** 规约是一种强命名,可重用,可组合,可测试的实体过滤器.
- **领域事件(Domain Event):** 领域事件是当领域某个事件发生时,通知其它领域服务的方式,为了解耦领域服务间的依赖.

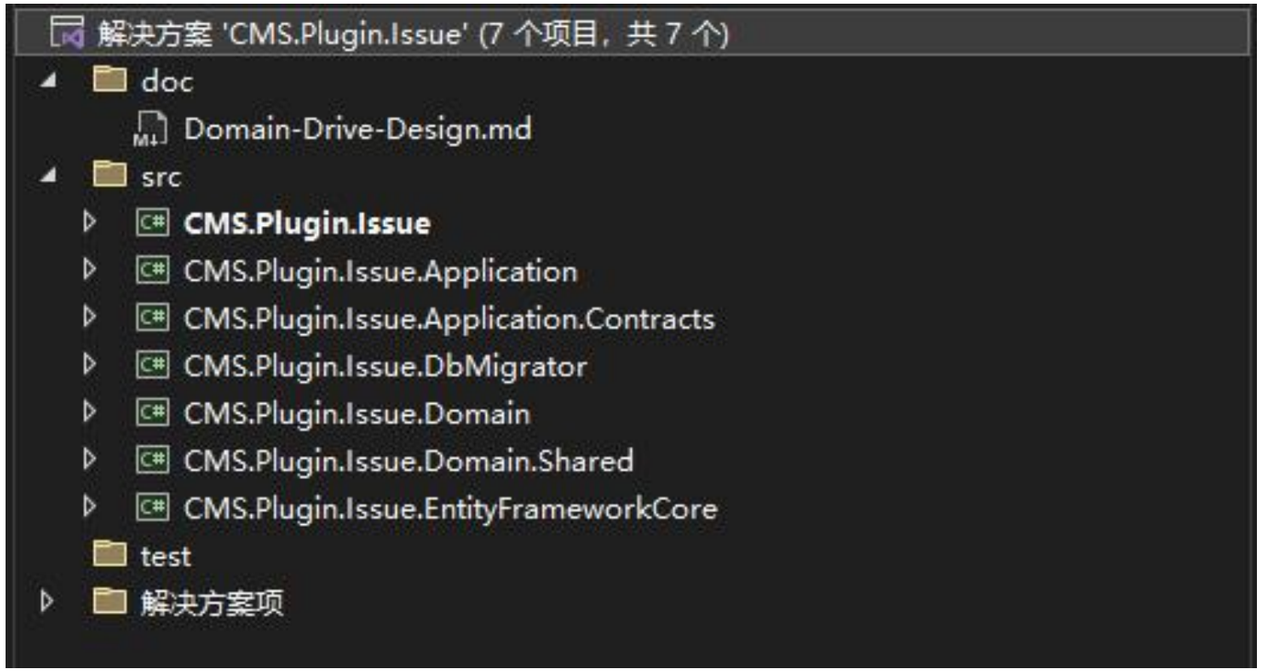
应用层

- **应用服务(Application Service)**：应用服务是为实现用例的无状态服务.展现层调用应用服务获取DTO.应用服务调用多个领域服务实现用例.用例通常被视为一个工作单元.
- **数据传输对象(DTO)**：DTO是一个不含业务逻辑的简单对象,用于应用服务层与展现层间的数据传输.
- **工作单元(UOW)**：工作单元是事务的原子操作.UOW内所有操作,当成功时全部提交,失败时全部回滚.

.NET解决方案分层

下图是使用cms-ddd的启动模板创建的解决方案:

解决方案名为 CMS.Plugin.Issue ,它包含多个项目.该解决方案出于DDD原则及开发和部署的实践来进行分层.



《CMS2.0项目二次开发入门教程.pdf》

.NET解决方案分层

领域层分为两个项目：

CMS.Plugin.Issue.Domain,是**领域层中必需**的,它包含之前介绍的**构建组成**(实体,值对象,领域服务,规约,仓储接口等).

CMS.Plugin.Issue.Domain.Shared,是领域层中**很薄的项目**,它只包含领域层与其它层共享的数据类型的定义.例如,枚举,常量等.

应用层也被分为了两个项目：

CMS.Plugin.Issue.Application.Contracts,包含**接口**的定义及接口依赖的**DTO**,此项目可以被展现层或其它客户端应用程序引用.

CMS.Plugin.Issue.Application,是**应用层中必需**的,它实现了CMS.Plugin.Issue.Application.Contracts项目中定义的接口.

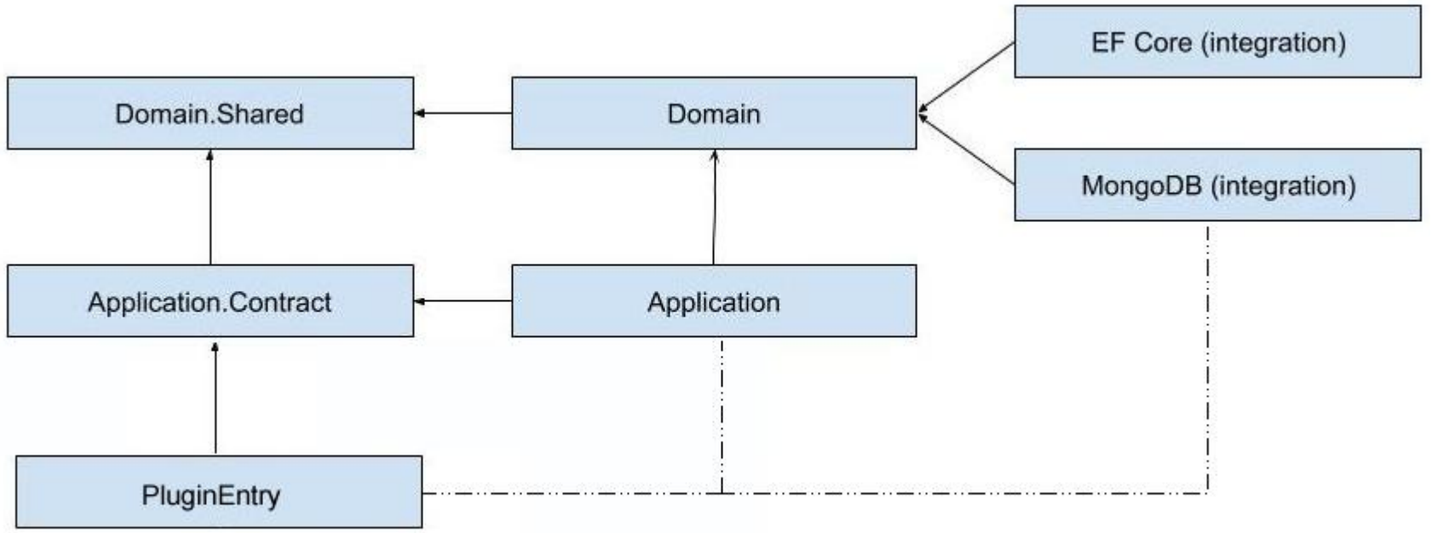
CMS.Plugin.Issue.EntityFrameworkCore,是必需的,因为需要集成EF Core.应用程序的数据库上下文(DbContext),数据库对象映射,仓储接口的实现,以及其它与EF Core相关的内容都位于此项目中.

CMS.Plugin.Issue.DbMigrator,它是一个简单的控制台程序,用来执行数据库迁移,包括**初始化**数据库及创建**种子数据**.这是一个非常实用的应用程序,你可以在开发环境或生产环境中使用它.

CMS.Plugin.Issue,是一个CMS插件入口程序,包含了HTTP API的定义.它通常包含**MVC Controller** 和 **Model**(如果有).因此,你可以在此项目中提供HTTP API.

项目间的依赖关系

下图展示了解决方案中项目间的依赖关系:



Domain.Shared 所有项目直接或间接依赖此项目.此项目中的所有类型都可以被其它项目所引用.

Domain 仅依赖Domain.Shared项目,因为Domain.Shared本就属于领域层的一部分.例如,Domain.Shared项目中的枚举类型 IssueType 被Domain项目中的Issue实体所引用.

Application.Contracts 依赖Domain.Shared项目,可以在DTO中重用Domain.Shared中的类型.例如,Domain.Shared项目中的枚举类型 IssueType 同样被Contracts项目中的CreateIssueDto所引用.

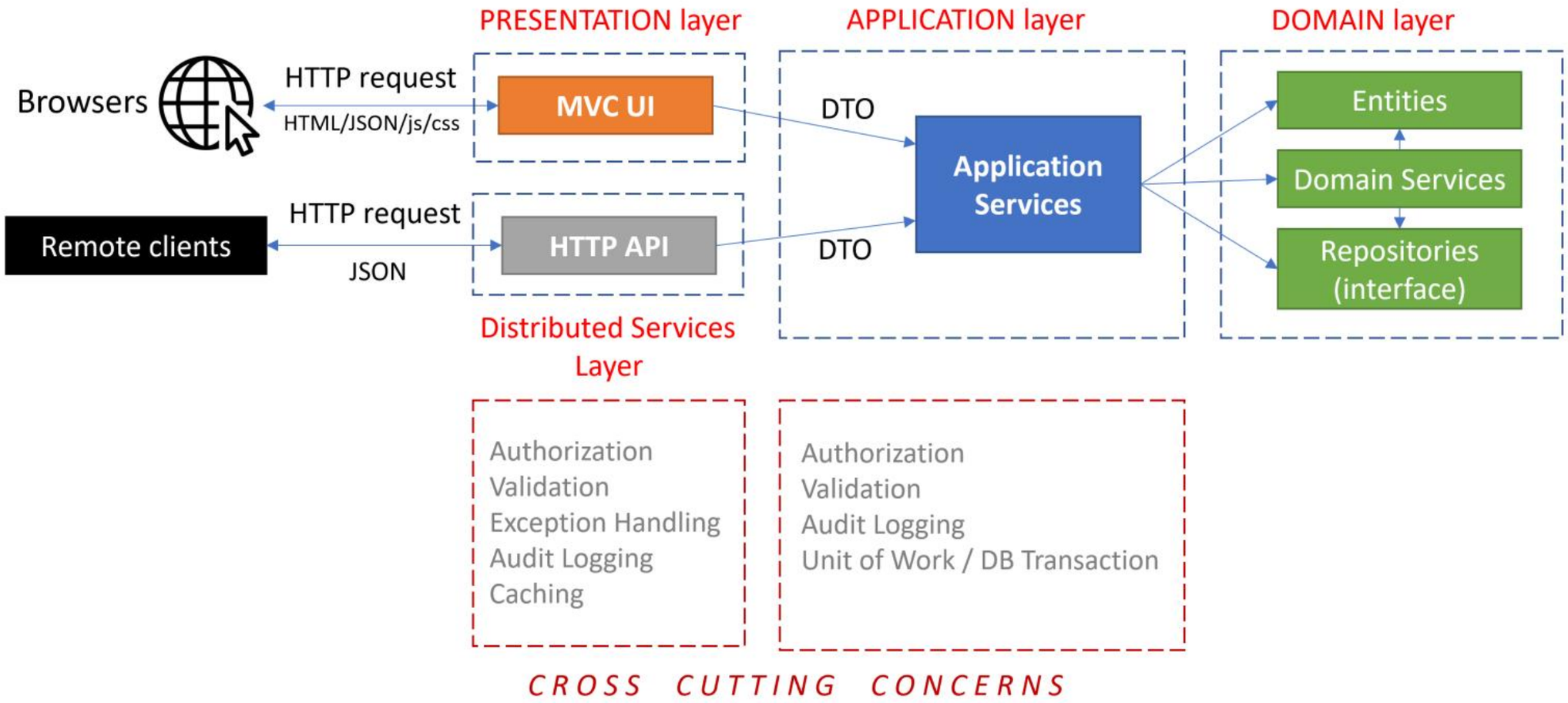
Application 依赖Application.Contracts项目,因为此项目需要实现应用服务的接口及接口使用的DTO.另外也依赖Domain项目,因为应用服务的实现必须依赖领域层中的对象.

EntityFrameworkCore 依赖Domain项目,因为此项目需要将领域对象(实体或值对象)映射到数据库的表,另外还需要实现Domain项目中的仓储接口.

PluginEntry 是CMS基座加载插件的入口，在启动程序注册服务是需要应用服务和仓储的实现类.

DDD模式的应用程序执行顺序

- 通常由UI(用例)发起一个HTTP请求到服务器.
- 由展现层中的一个MVC Controller处理请求,在这个阶段可以执行一些AOP逻辑(授权,验证,异常处理等),MVC Controller调用注入的应用服务接口,并返回其调用后的结果(DTO)..
- 应用服务使用领域层的对象(实体,仓储接口,领域服务等)来实现UI(用例)交互.此阶段同样可以执行一些AOP逻辑(授权,验证等).应用服务中的每个方法应该是一个工作单元,代表它是一次原子性操作.



四、通用原则

数据库提供程序 / ORM 独立原则

领域层和应用层应该与数据库提供程序 / ORM无关.领域层和应用层仅依赖**仓储接口**,并且仓储接口不依赖特定的ORM对象.

原因如下:

- 未来领域层或应用层的基础设施会发生改变,例如,需要支持另外一种数据库类型,因此需要保持领域层或应用层的基础设施是独立的.
- 将基础设施的实现隐藏在仓储中,使得领域层或应用层更专注于业务逻辑代码.
- 可以通过模拟仓储接口,使得自动化测试更为方便.

关于此原则, EntityFrameworkCore项目只被PluginEntry项目所引用,解决方案中其它项目均未引用.

展现层技术无关原则

展现层技术(UI框架)是现代应用程序中最多变的部分之一.**领域层和应用层**应该对展现层所采用的技术或框架**一无所知**.

在某些情况下,你可能需要在应用层和展现层中写重复的逻辑,例如,参数验证和授权检查.展现层检查出于**用户体验**,应用层或领域层检查出于**数据安全性**和**数据完整性**.

关注状态的变化,而不是报表/查询

DDD关注领域对象的**变化和相互作用**,如何创建或修改一个具有数据**完整性,有效性**,符合**业务规则**的实体对象.

DDD忽略**领域对象的数据展示**,这并不意味着它们并不重要,如果应用程序没有精美的看板和报表,谁会愿意用呢?但是报表是另外一个讨论话题,你可以通过使用SQL Server报表功能或ElasticSearch来提供数据展示,又或者使用优化后的SQL查询语句,创建数据库索引或存储过程.唯一的原则是不要将这些内容混入领域的业务逻辑中.

五、最佳实践

聚合 / 聚合根 原则

业务规则：实体负责实现与其自身属性相关的业务规则.同时**聚合根实体**还负责它们的子集合实体.

聚合应该通过领域规则和约束来保证自身的完整性和有效性.这意味着,**实体与DTO是不同的**,实体应该比DTO多了些实现业务逻辑的方法.我们应该尽可能地**在实体上实现业务规则**.

独立单元：应该在一个独立的单元中完成一个聚合的获取及保存,包括自身属性及其子集合.

事务边界：通常认为一个聚合就是一个事务边界.如果用例只涉及单个聚合,那么读取及修改就是一个操作单元.对聚合内所有对象的修改都将作为原子操作一起保存,无需显式创建数据库事务.但是,实际上,可能需要在一个用例中更改**多个聚合对象的实例**,并且还要求创建事务来保证**原子更新**和**数据一致性**.因此,ABP框架提供了为每个用例(应用服务中的方法),可以创建显式事务的功能.有关更多信息,请参见文档工作单元.

序列化：一个聚合(包含聚合根及子集合)可以被序列化或反序列化.例如,MongoDB在保存对象到数据库时,会将聚合序列化为JSON文件,读取时再进行反序列化.使用关系型数据库+ORM时,这个原则不是必须的,但是,这是领域驱动设计的重要实践.

聚合/聚合根规则及最佳实践

聚合间只通过ID相互引用

聚合应该只引用其它聚合的ID,也就是说,不允许定义导航属性关联至其它聚合.

- 该规则遵循了可序列化原则.
- 该规则还可以避免不同聚合彼此间的相互操作以及业务逻辑的暴露.

聚合根 / 实体的主键

- 聚合根通常具有唯一的标识符ID (主键: PK).我们建议使用 Guid作为聚合根的主键类型. (原因请参见Guid生成文档).
- 聚合中的实体(非聚合根)可以使用联合主键.

聚合根 / 实体的构造函数

构造函数是实体生命周期的开始被执行.以下是构造函数的编写建议:

- 将实体的必填属性作为构造函数参数,这样可以创建一个有效(符合规则)的实体.另外,将非必填属性作为构造函数的可选参数.
- 参数必须检查有效性.
- 所有子集合对象必须被初始化.

实体属性访问器和方法

如果我们使用公开的属性,则无法在实体控制数据的有效性和完整性.建议:

- 如果某个属性具有业务逻辑,则将该属性的setter改为私有.
- 定义公开的方法来修改属性.

业务逻辑与实体异常

对实体进行验证,或执行业务逻辑时,通常需要抛出异常:

- 领域中定义的特定的异常.
- 实体方法中抛出的异常.

实体中业务逻辑依赖外部服务时:

仅依赖实体本身的属性执行的业务规非常简单.但是有时候,复杂的业务逻辑会查询数据库或使用依赖注入中的其它服务,这该怎么办?注意:**实体是无法注入服务的.**

实现这种业务逻辑有两种方式:

- 将依赖的服务以方法的参数,传递到实体的业务逻辑方法中.
- 定义一个领域服务.

仓储

仓储是一个类集合的接口,它通常被领域层或应用层调用,负责访问持久化系统(数据库),读取写入业务对象(聚合).

仓储的原则:

- 在领域层中定义仓储接口,因为仓储会被领域层或应用层调用,在基础设施层中实现(EntityFrameworkCore 项目).
- 仓储中不要写任何业务逻辑.
- 仓储接口不依赖 数据库提供程序 / ORM.例如,不要在仓储中返回 DbSet 类型,因为 DbSet是EF Core中的对象.
- 仅为聚合根定义仓储,非聚合根对象不要提供仓储,因为子集合可以通过聚合根来进行持久化.

仓储中不要写任何业务逻辑

我们经常不小心把业务逻辑编写到了仓储层.

规约模式

规约是一种强命名,可重用,可组合,可测试的实体过滤器. 有关ABP框架提供的规约更多信息,请参见规约文档.

例如:规定选择一个18岁以上的顾客

C# Copy

```
using System;
using System.Linq.Expressions;
using Volo.Abp.Specifications;

namespace MyProject
{
    public class Age18PlusCustomerSpecification : Specification<Customer>
    {
        public override Expression<Func<Customer, bool>> ToExpression()
        {
            return c => c.Age >= 18;
        }
    }
}
```

```
public async Task<List<Customer>> GetCustomersCanBuyAlcohol()
{
    var queryable = await _customerRepository.GetQueryableAsync();
    var query = queryable.Where(
        new Age18PlusCustomerSpecification().ToExpression()
    );

    return await AsyncExecuter.ToListAsync(query);
}
```

什么时候不要使用?

- **没有业务含义的表达式:**不要对与业务无关的表达式和操作使用规约.
- **报表:**如果只是创建报表,不要创建规约,而是直接使用 IQueryable 和LINQ表达式.你甚至可以使用普通SQL、视图或其他工具生成报表.DDD不关心报表,因此从性能角度来看,查询底层数据存储的方式可能很重要.

领域服务

领域服务主要用来实现本领域的逻辑:

- 依赖领域对象和仓储.

- 需要使用多个聚合.

- 建议使用 Manager 后缀来命名领域服务.

领域服务和领域对象一起使用.领域服务可以获取并返回实体,值对象等,它们不返回DTO, DTO属于应用层的一部分.

应用服务

应用服务是实现**用例**的**无状态服务**.应用服务通常获取并返回DTO.应用服务被展现层所使用,应用服务调用领域对象(实体,仓储等)来实现用例.

应用服务的通用原则:

- 实现特定用例的**应用程序逻辑**,不要在应用服务内实现**核心领域**的逻辑.
- 应用服务的方法**不要返回实体**.始终只返回DTO.

数据传输对象

DTO是应用层与展现层间传输数据的简单对象.应用服务方法获取并返回Dto.

DTO通用原则和最佳实践

- DTO应该是可被**序列化**的.因为大所数情况下,DTO是通过网络传输的,因此它应该具有**无参的构造函数**.
- 不应该包含任何**业务逻辑**.
- **切勿**继承或引用**实体**.

输入DTO(应用服务方法的参数)与 输出DTO (应用服务方法的返回对象)具有不同的作用,因此,它们应该区别对待.

输入DTO 最佳实践

不要在输入DTO中定义不使用的属性

- 仅在输入DTO中定义用例所需要的属性! 否则,会造成调用应用服务的客户端产生困惑.

不要重用输入DTO

- 为每个用例(应用服务的方法)单独定义一个专属的输入DTO.否则,在一些情况下,会添加一些不被使用的属性,这样就违反上面的规则:**不要在输入DTO中定义不使用的属性**.
- 例外情况: 该规则有一些例外的情况,例如,你想开发两个方法,它们共用相同的输入DTO(通过继承或重用),有一个报表页面有多个过滤条件,多个应用服务使用相同的输入参数返回不同的结果(如,大屏展示数据,Excel报表,csv报表).这种情况下,你是**需要修改一个参数,多个应用服务都应该一起被修改**.

输入DTO中验证逻辑

- 仅在DTO内执行简单验证.使用数据注解验证属性或通过IValidatableObject 方式.
- 不要执行领域验证.例如,不要在DTO中检查用户名是否唯一的验证.

输出DTO最佳实践

保持数量较少的输出DTO,尽可能重用输出DTO(例外:不要将输入DTO作为输出DTO).

输出DTO可以包含比用例需要的属性更多的属性.

针对 Create 和 Update 方法,返回实体的DTO.

输出DTO的建议并不适用于所有情况.出于性能原因,我们可以忽略这些建议,尤其是在返回大量数据,为UI定制,并发量较高时.

在这些情况下,你可以定制仅包含必要信息的DTO.上面的建议只适用于额外多些属性并不会损失太多性能,并关注代码可维护的应用系统.

对象映射到对象

当两个对象具有相同或相似的属性,自动将对象映射到对象是一种将值从一个对象复制到另外一个对象非常有用的方法.
DTO和实体通常具有相同或相似的属性,你经常需要从一个实体创建一个DTO对象.相较于手动映射,基于AutoMapper的ABP对象映射系统,更加方便简单.

- **仅在**实体=>输出DTO的时候使用自动映射.
- 不要在输入DTO=>Entity的时候使用自动映射.

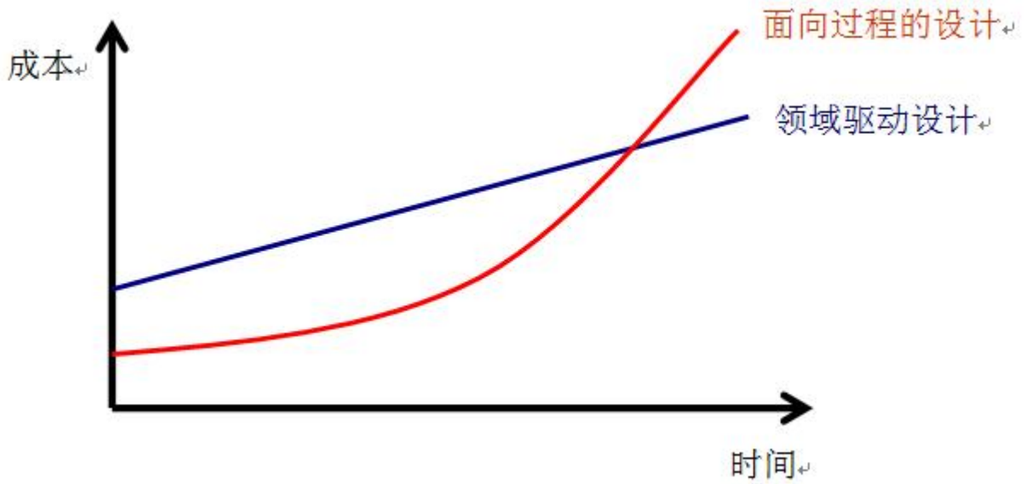
因为以下原因,你不应该在输入DTO=>Entity的时候使用自动映射:

1. 实体类通常具有一个构造函数,该构造函数带有参数,确保创建有效的对象,而自动对象映射通常需要一个**无参构造函数**.
2. 大多数实体中的属性setter是私有的,你只能调用实体上的方法来修改属性.
3. 另外,需要进行对用户或客户端的输入参数进行验证,而不是盲目映射到实体属性上.

总结

有选择性的使用领域驱动设计

所谓设计，就是以你和你的团队的知识、经验和智慧，全面充分的考虑各种内外因素后，在设计方案中作出**合理**的选择的过程。

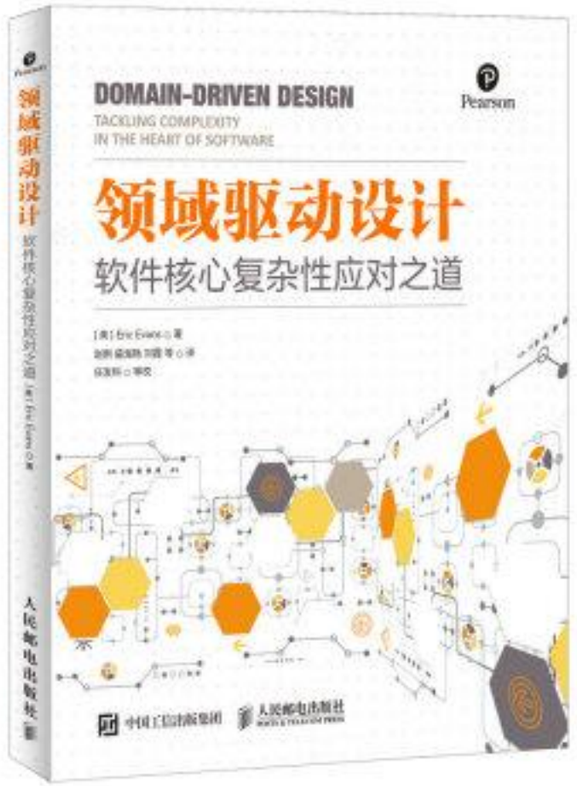


系统的初始阶段，领域驱动设计需要付出更大的成本，但随着时间的推移，领域驱动设计的成本效益优势会逐步显现

使用领域驱动设计，并不代表整个系统的方方面面都必须遵从领域驱动设计的原则，需要根据实际情况，**让适合的部分使用领域驱动设计，让不适合的部分使用面向过程的设计**。（对于一些业务视图，以及报表模块，很明显不适合使用面向对象的方式设计，因为这些“视图”和“报表”，本质上就不是业务实体。）

相关书籍

如果你对领域驱动设计和构建大型系统有兴趣,建议将以下书籍作为参考书籍:



"Domain Driven Design"

by Eric Evans



"Implementing Domain Driven Design" by

Vaughn Vernon



"Clean Architecture"

by Robert C. Marti

谢　谢！