**面向领域的微服务架构实现**

[什么是领域驱动设计](#9952-1616553923118)

[为什么需要领域驱动设计](#9049-1616557200127)

[领域驱动设计优势](#9470-1617335151685)

[开发中的问题](#6144-1616658123125)

[如何解决](#9951-1616565637442)

[如何实践领域驱动设计](#3858-1617335304673)

[领域驱动设计的两个阶段](#4030-1616658172959)

[领域驱动设计和领域模型之间的关系](#7693-1617335221920)

[领域驱动设计分层架构](#9613-1617335365440)

[什么是领域模型](#7011-1617337257057)

[为什么要建模](#4073-1617337294213)

[怎么建模才合理](#2399-1617335400624)

[四种领域模型](#1497-1617337347416)

[设计领域模型的步骤](#3786-1617335296134)

[架构原则](#4326-1616565663938)

[架构思想](#1135-1616566018089)

[领域驱动设计（DDD)](#9816-1616652431904)

[领域驱动设计的经典分层架构](#8174-1616662439388)

[领域驱动设计过程中使用的模式](#4055-1616663397921)

[Clean架构（洋葱架构）](#9028-1616669545951)

[六边形架构](#7582-1616666473675)

[CQRS架构](#4498-1616565664468)

[TMF架构](#6620-1616724634126)

[架构实现](#1182-1616551028218)

[商城功能架构](#6179-1616723939051)

[面向领域的微服务架构](#2850-1617333550101)

[DOMA架构与传统架构对比](#3799-1617333374048)

[工程架构分层方案](#8064-1617102151087)

[领域服务层：领域服务项目架构方案](#5394-1617000336031)

[基础服务层：业务基础服务项目架构方案](#2843-1617102374770)

[应用服务层：web应用服务项目架构方案](#4773-1617333481021)

[web应用服务项目架构（单体应用，应用不分层）](#8058-1617170676153)

[项目模块依赖关系](#7344-1617171087172)

[不同领域跨服务调用](#7517-1616551792041)

[技术开发](#8372-1617334213528)

[问题整理](#3147-1616658596538)

[知识扩展](#4096-1616575901512)

[领域驱动设计](#6881-1616572762478)

[领域和领域模型](#8056-1616669315359)

[领域](#9130-1617339407383)

[上下文映射图](#4211-1617339361793)

[边界是核心](#2818-1616572422013)

[纪律是关键](#5470-1617338959395)

[领域驱动设计能力评估模型](#4092-1617276532902)

**什么是领域驱动设计**

领域驱动设计(DDD:Domain-Driven Design)：

是一套综合软件系统分析和设计的面向对象建模方法。

DDD是解决复杂中大型软件的一套行之有效方式。

一个好的领域模型是控制复杂问题的关键。

领域模型的价值在于提供一种通用的语言，使得领域专家、产品经理和软件技术人员联系在一起，沟通无歧义。

领域驱动设计的核心是建立正确的领域模型。

**为什么需要领域驱动设计**

开发工程师是通过软件来解决问题，编写代码只是其中的一部分工作，设计和交流同样重要。领域驱动设计的目的是让软件系统在实现时准确的基于对真实业务过程的建模并根据真实的业务过程的调整而调整。

**领域驱动设计优势**

1.**以业务必须为导向**

2.**整个团队使用的一组通用术语和定义**

3.**让跟踪变得容易** ：使用相同的术语，跟踪需求实现变得非常简单

**4.更好的代码**：更易读的代码并减少重复，易维护。模型可重复使用，避免重复的模型。

**5.有效地隔离了业务复杂度与技术复杂度**：将领域层作为整个系统稳定而内聚的核心，分离变与不变

领域驱动设计（DDD）的最大好处是：

1.接触到需求**第一步就是考虑领域模型**，而不是将其切割成数据和行为，然后数据用数据库实现，行为使用服务实现，最后造成需求的首肢分离。

2.领域驱动设计让你**首先考虑的是业务语言，而不是数据**。

**开发中的问题**

设计不足（例如：过程式编写代码）主要表现为冗余、耦合、注释覆盖率高

过度设计主要表现为复杂、臃肿、代码闲置率高。

难题：难以修改、难以测试、难以阅读。

现有架构：UI+Service+DataBase的多层SOA架构，这种服务+表模型的架构易使服务变得囊肿，难于维护拓展，伸缩性能差。

**如何解决**

架构升级

**面向领域的微服务架构（DOMA）**，采用**DDD**+**SOA微服务**的**事件驱动的CQRS读写分离**架构，应付复杂业务逻辑，以聚合模型替代数据表模型，以并发的事件驱动替代串联的消息驱动。真正实现以业务实体为核心的灵活拓展。

**如何实践领域驱动设计**

统一语言：业务术语、技术术语→ 领域模型术语

形式：领域专家、技术专家一起头脑风暴，画草图讲解，讨论

**领域驱动设计的两个阶段**

1. 领域通用语言

以一种领域专家、设计人员、开发人员都能理解的通用语言作为相关交流的工具。在交流的过程中发现领域概念，并将这些概念设计成一个领域模型。

2. 将领域模型转换为代码

由领域模型驱动软件设计，用代码来实现领域模型，从而确定业务边界和应用边界，保证业务模型和代码模型的一致性

**领域驱动设计和领域模型之间的关系**

领域驱动设计的核心是建立正确的领域模型。

**领域驱动设计分层架构**

展现层（Interface）又叫UI层，用户界面。负责与前端交互进行界面展示。主要用来处理用户发送的请求，并将数据传递给应用层。数据的组装、数据传输格式以及Facade（“门面”）接口等代码都会放在这层目录里。

应用层（Application Layer）:负责业务流程，但不包含业务逻辑

领域层（Domain Layer）:负责表达业务概念，业务状态信息以及业务规则，是业务软件的核心

基建层（Infrastructure Laye）:又叫基础设施层，负责为其他层提供通用的技术能力，提供层间的通信，为领域层实现持久化基础。

总之，基础设施层可以通过架构和框架来支持其他层的技术需求；

分类的依据是：越往上，预期变动越频察；越往下，预期变动越少。

**什么是领域模型**

又叫概念模型，领域对象模型，分析对象模型。

领域模型是一套完整，详尽的方法论。是对领域内的概念类或现实世界中对象的可视化表示。

目的：专注于分析问题领域本身，发掘重要的业务领域概念，并建立业务领域概念之间的关系。如从如何需求沟通（构建领域知识），到高层设计（战略建模），详细设计（战术建模），细致到代码的实现风格都给出了示例。

**为什么要建模**

通用”是建模的第一步。“复用”将模型达成一致。
第一： 把心智模型提取出来，显性化，让不同的人对业务的理解达成一致。
第二： 建模本质上是一种抽象。抽象就是归类。目的是减轻认知负担，避免重复的思考和工作，提升人的计算能力。归类复用，避免重复的工作，让人可以关注更高层面的事务

**怎么建模才合理**

判断模型好坏的重要依据是它的使用效率（扩展度、灵活度、与组织的对应度）

**四种领域模型**

**失血模型**：模型仅仅包含数据的定义和getter/setter方法，业务逻辑和应用逻辑都放到服务层中。

**贫血模型**：贫血模型中包含了一些业务逻辑，但**不包含依赖持久层的业务逻辑**。这部分依赖于持久层的业务逻辑将会放到服务层中。

**充血模型**：充血模型中包含了所有的业务逻辑，**包括依赖于持久层的业务逻辑**。所以，使用充血模型的领域层是依赖于持久层，简单表示就是 UI层->服务层->领域层<->持久层。

**胀血模型**：胀血模型就是把和业务逻辑不想关的其他应用逻辑（如授权、事务等）都放到领域模型中。

**设计领域模型的步骤**

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。

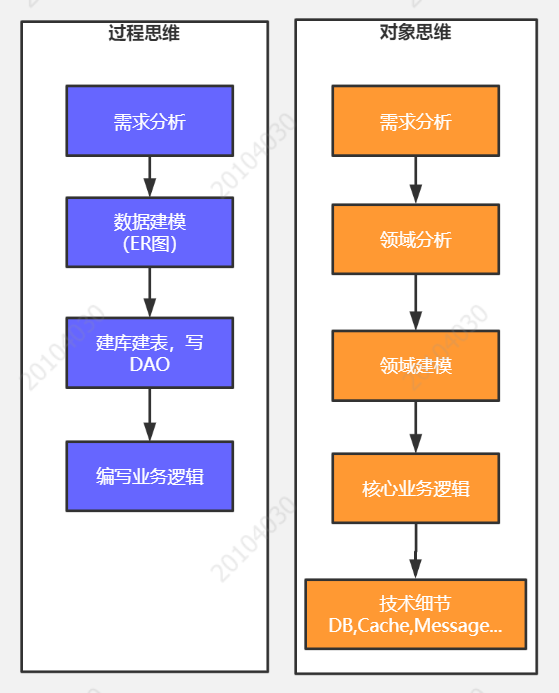
进一步分析每个上下文内部，识别出哪些是实体，哪些是值对象。

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。

采用对象思维的开发过程



**架构原则**

高内聚与低耦合原则，单一职责原则，依赖倒置原则，开闭原则（Extension组件），简单设计原则，领域防腐（防腐层：防腐重任由infrastructure层RepositoryImpl担当）

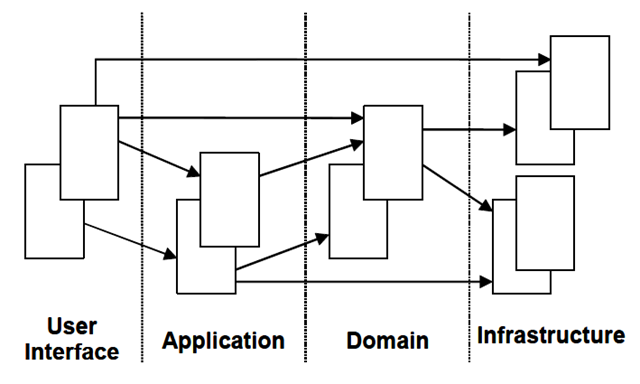
**架构思想**

整合DDD（领域驱动设计）、Clean架构（整洁架构）、六边形架构（适配器架构）、 CQRS（命令查询的责任分离架构）、TMF架构等思想。

**领域驱动设计（DDD)**

分治、抽象、知识

**领域驱动设计的经典分层架构**



**用户界面/展现层**

负责向用户展现信息以及解释用户命令。更细的方面来讲就是：

请求应用层以获取用户所需要展现的数据；

发送命令给应用层要求其执行某个用户命令；

**应用层**

很薄的一层，定义软件要完成的所有任务。对外为展现层提供各种应用功能（包括查询或命令），对内调用领域层（领域对象或领域服务）完成各种业务逻辑，应用层不包含业务逻辑。

**领域层**

负责表达业务概念，业务状态信息以及业务规则，领域模型处于这一层，是业务软件的核心。

**基础设施层**

本层为其他层提供通用的技术能力；提供了层间的通信；为领域层实现持久化机制；总之，基础设施层可以通过架构和框架来支持其他层的技术需求；

**领域驱动设计过程中使用的模式**

**关联的设计**

遵循如下的一些原则：

关联尽量少；否则不利于理解和维护单个对象，很难划分对象与对象之间的边界；

关联尽量保持单向的关联；

在建立关联时，挖掘是否存在关联的限制条件，限制条件能将关联化繁为简，可以将多对多简化为1对多，或将1对多简化为1对1；

**实体（Entity）**

实体就是领域中需要唯一标识的领域概念，即领域对象。领域对象包含：属性、行为、状态。

不要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。

**值对象（Value Object）**

基本对象类型，String、Integer、Double、Boolean、Object、Collection

**领域服务（Domain Service）**

领域中的一些概念不太适合建模为对象，例如一些操作，动作。这些操作或动作往往会涉及到多个领域对象，并且需要协调这些领域对象共同完成这个操作或动作。如果强行将这些操作职责分配给任何一个对象，则被分配的对象就是承担一些不该承担的职责，从而会导致对象的职责不明确很混乱。

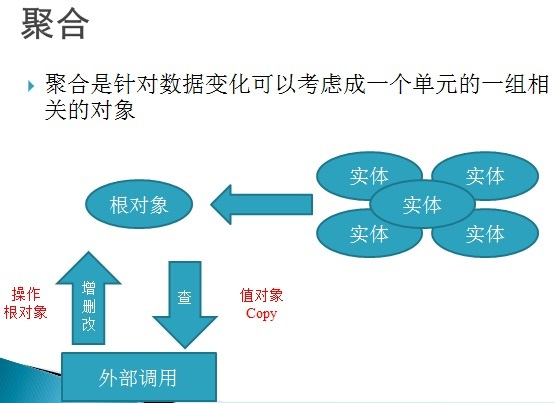
领域服务是以动词开头来命名的，比如资金转帐服务可以命名为MoneyTransferService。

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。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。

功能：避免领域逻辑泄露到应用层。类似Façade模式（门面模式）通过调用领域服务提供的简单易懂但意义明确的接口。

**聚合及聚合根（Aggregate，Aggregate Root）**

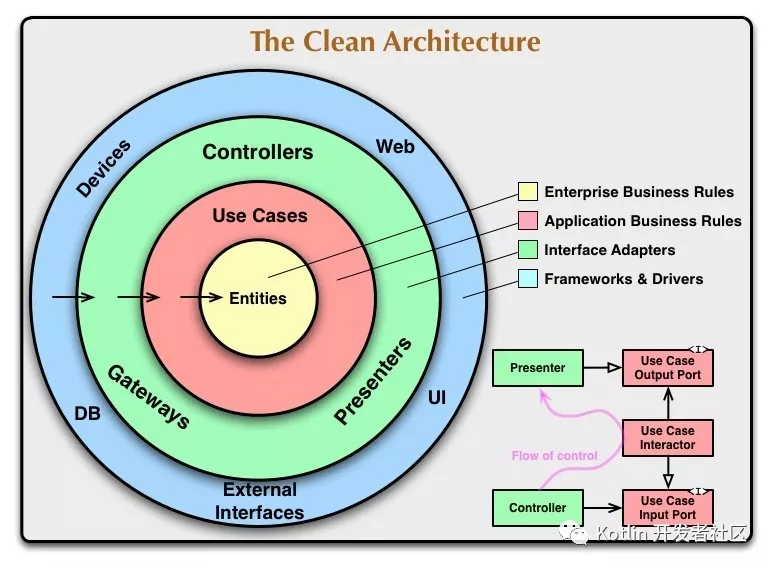
聚合是用来定义领域对象所有权和边界的领域模式，帮助简化模型对象间的关系。

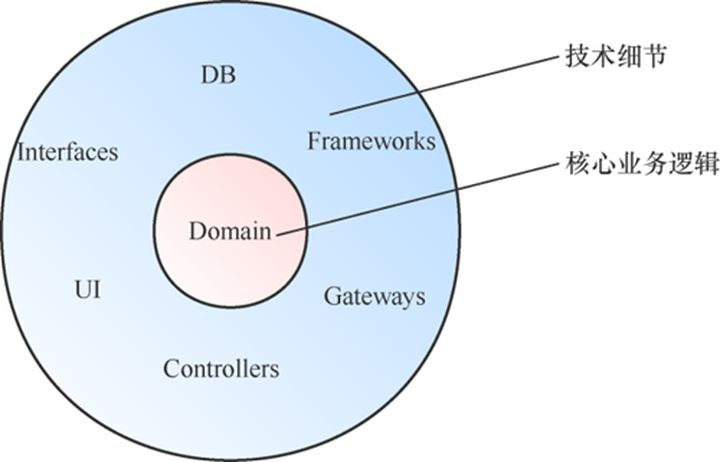


**Clean架构（洋葱架构）**

核心原则：依赖原则、抽象、层之间的通信

从最抽象的核心到细节的边缘，内层包含业务逻辑、外层包含实现细节

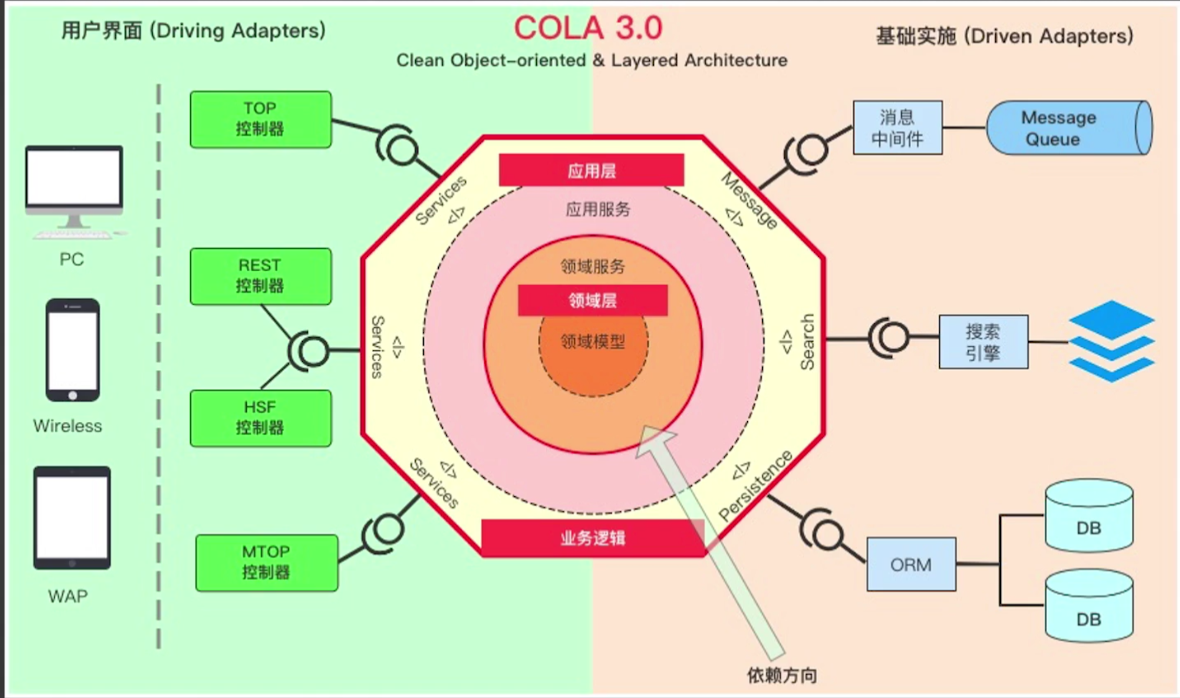




**六边形架构**

端口或适配器架构

端口和适配器架构的内核是将业务逻辑与技术实现隔离，是[单一职责原则](https://link.zhihu.com/?target=https%3A//zh.wikipedia.org/wiki/%25E5%258D%2595%25E4%25B8%2580%25E5%258A%259F%25E8%2583%25BD%25E5%258E%259F%25E5%2588%2599)的一种体现，从而更有效地将变更控制在某一个范围内，更有信心地应对变化。



**CQRS架构**

命令查询的责任分离Command Query Responsibility Segregation (简称CQRS)模式是一种架构体系模式，能够使改变模型的状态的命令和模型状态的查询实现分离。包括**状态校验，状态转换**。

核心思想是将应用程序的查询部分和命令部分完全分离，这两部分可以用完全不同的模型和技术去实现，利于实现技术上的读写分离。

好处：

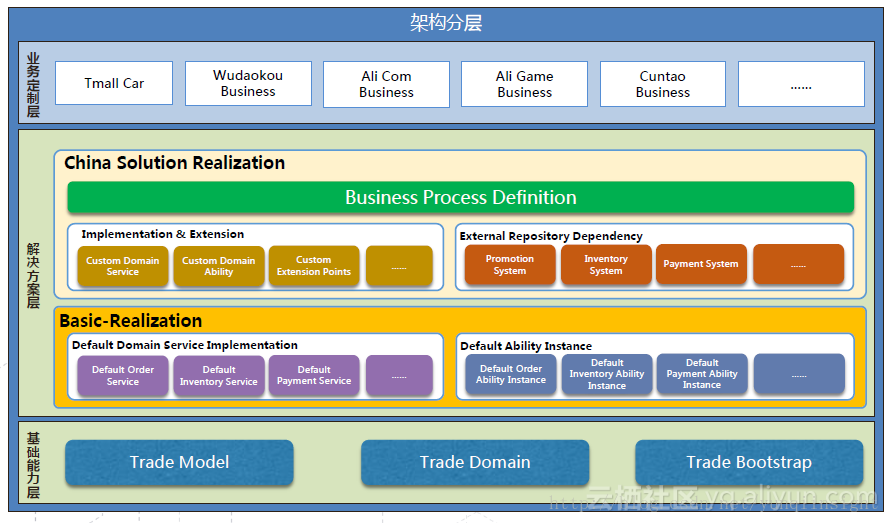
实现命令部分的领域模型不用经常为了领域对象可能会被如何查询而做一些折中处理；

由于命令和查询是完全分离的，所以这两部分可以用不同的技术架构实现，包括数据库设计都可以分开设计，每一部分可以充分发挥其长处；

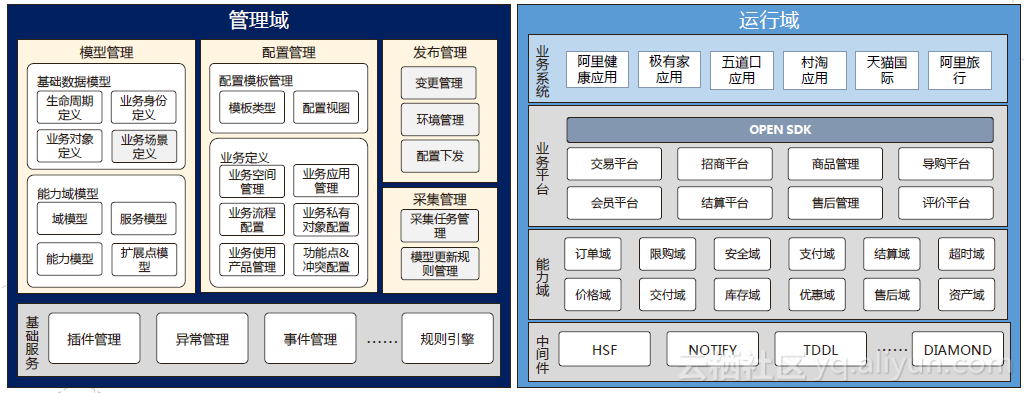
高性能，命令端因为没有返回值，可以像消息队列一样接受命令，放在队列中，慢慢处理；处理完后，可以通过异步的方式通知查询端，这样查询端可以做数据同步的处理；

**TMF架构**

业务定制包与平台分离的架构



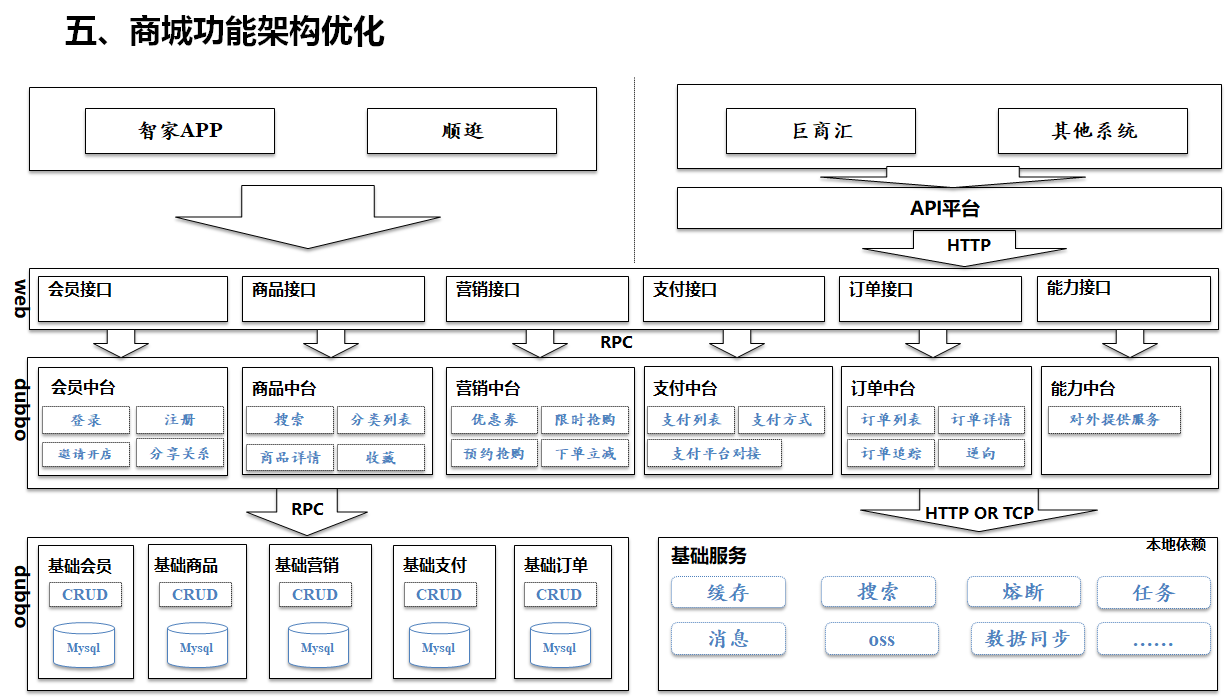
业务管理域与运行域分离的框架



**架构实现**

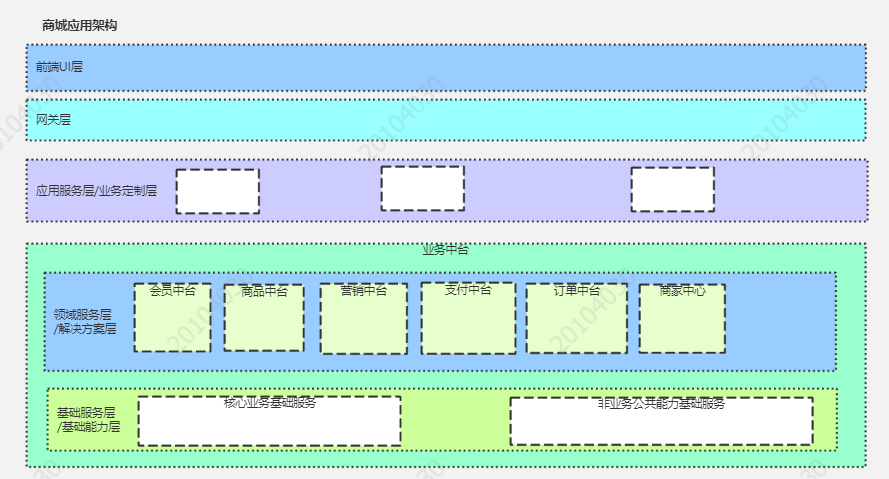
**商城功能架构**

首先引用一下商城功能架构优化方案（@Author：涛哥）



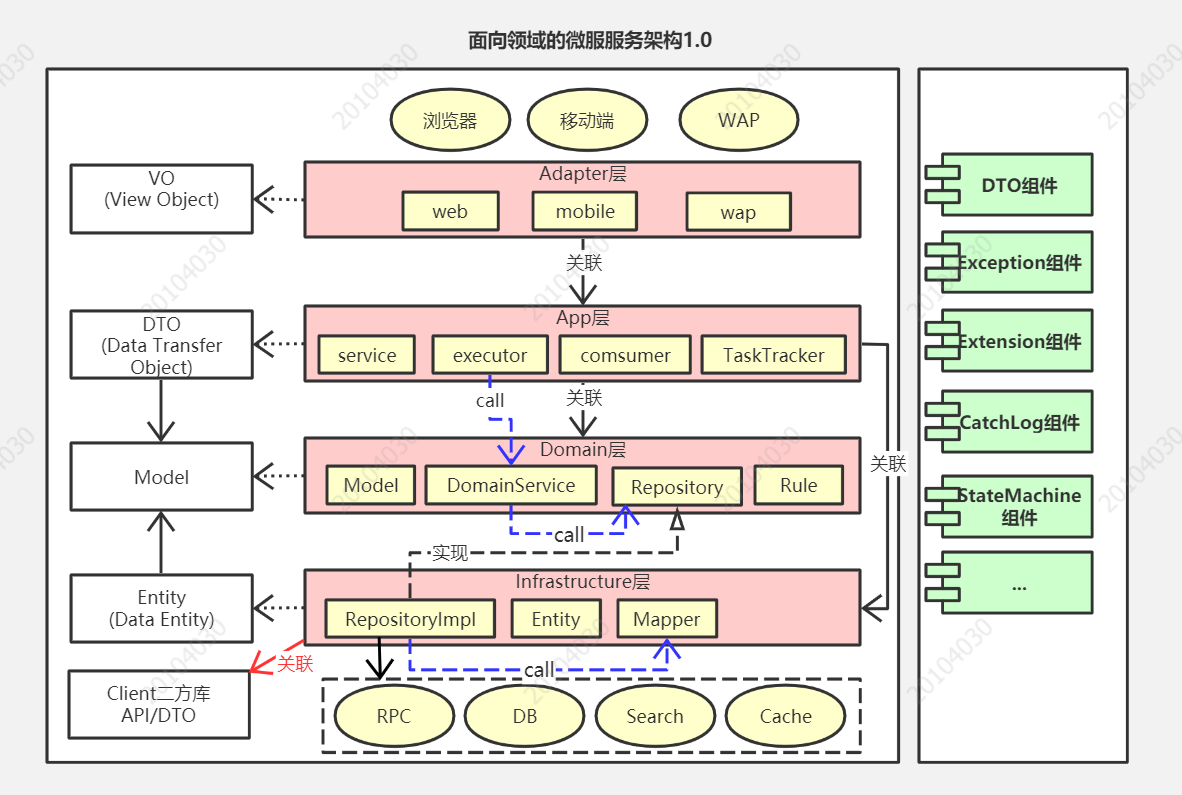
应用架构功能分层

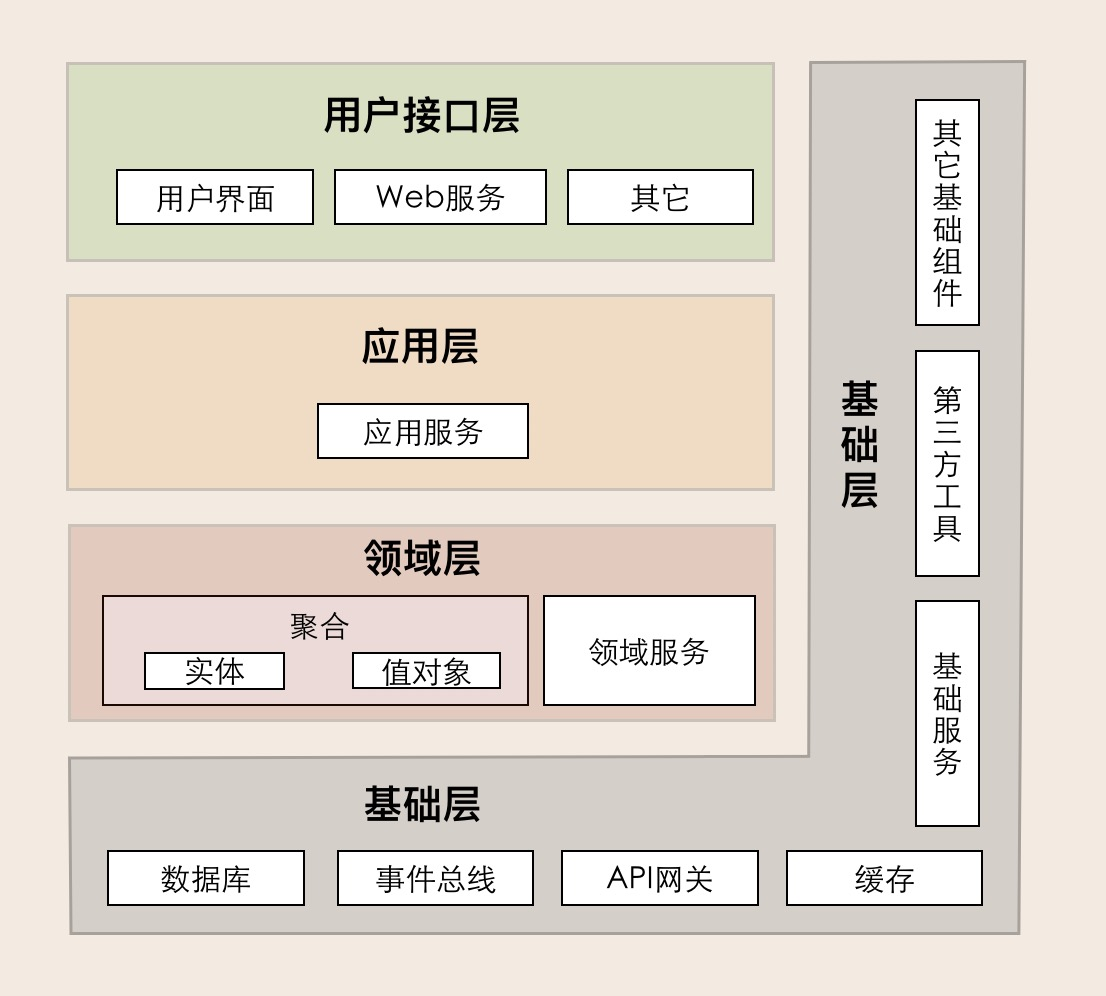
其中领域服务层不是一个严格意义上的上下分层，不同领域服务间可以相互调用，应遵循单向依赖原则，如果业务实现需要存在服务间循环调用，建议抽离出一个新的领域服务，理想状态下服务网络拓扑图应该是树状图，而不是环状图。



**面向领域的微服务架构**

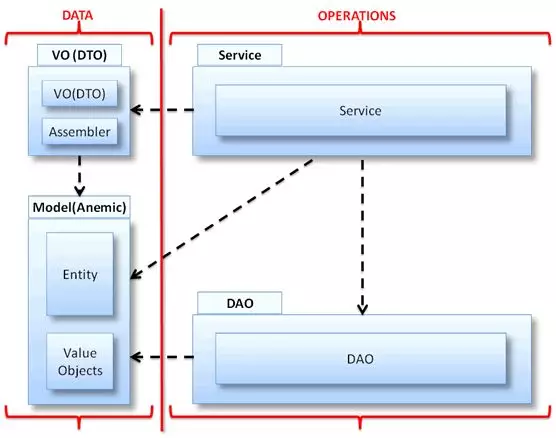
面向领域的微服务架构Domain Oriented Microservice Architecture (DOMA)



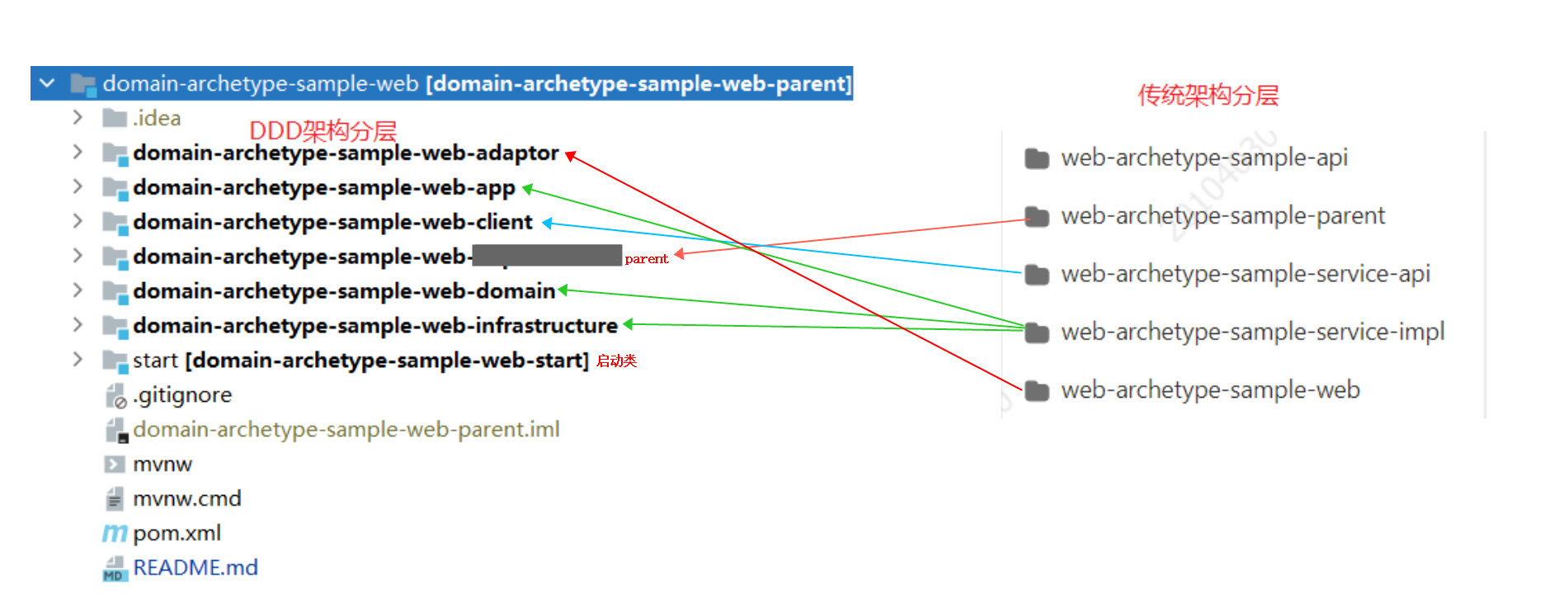


**DOMA架构与传统架构对比**

**依赖关系对比**

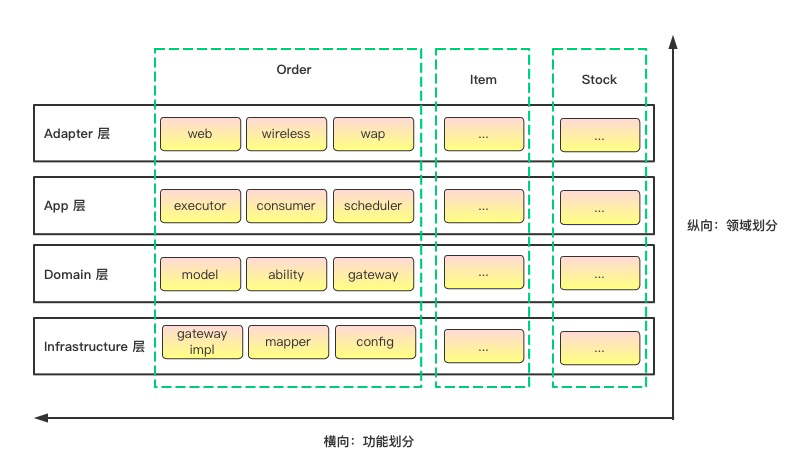


**分层关系对比**

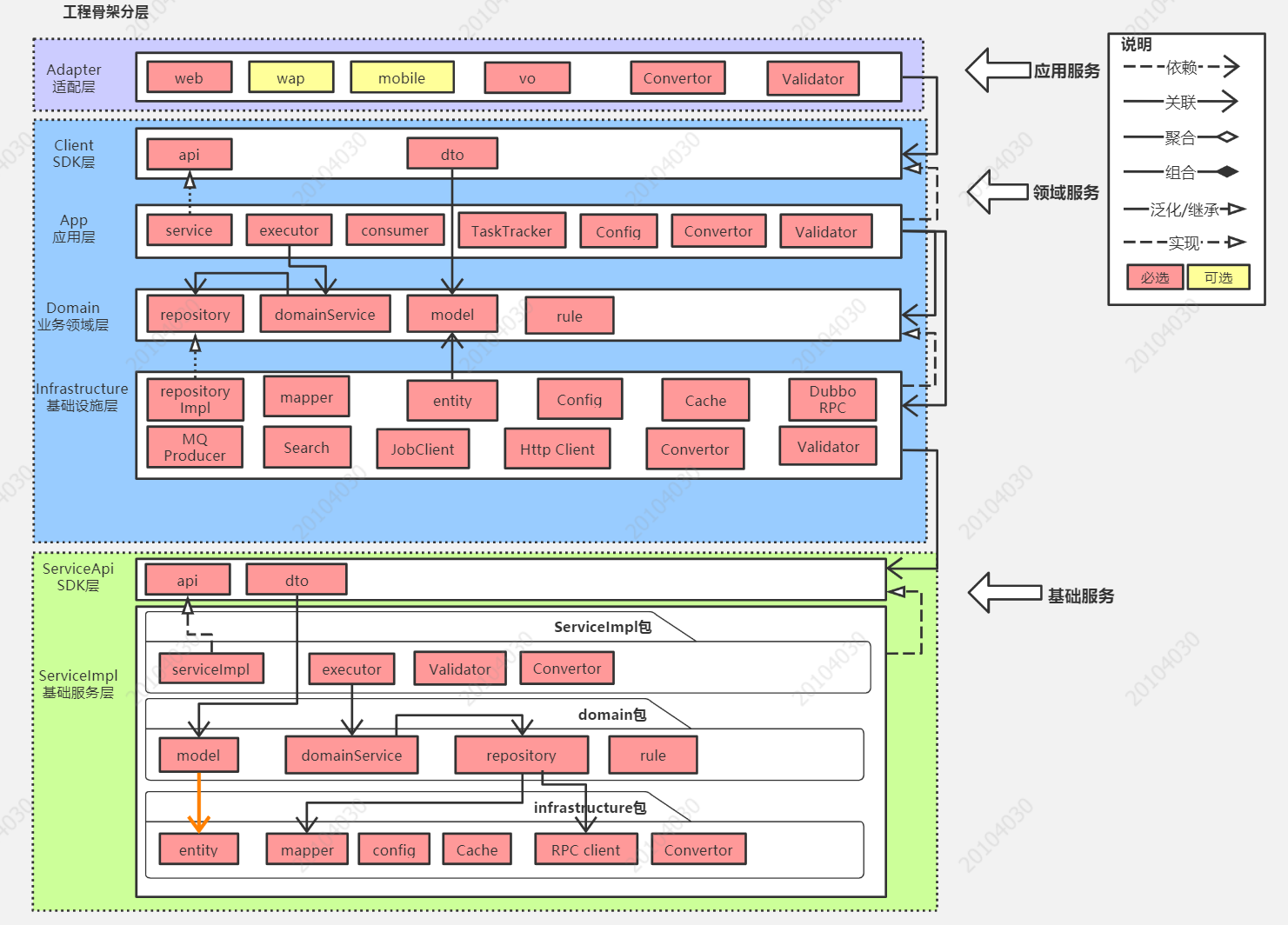


**工程分层**

项目实现：横向功能分层、纵向领域分包



**工程架构分层方案**



1）**适配层（Adapter Layer）**：负责对前端展示（web，wireless，wap）的路由和适配，对于传统B/S系统而言，adapter就相当于MVC中的controller；

web: 处理页面请求的Controller，暴露Http接口，大部分接口功能都应当在web包下。

wirless: 处理无线端特有功能适配。

wap:处理wap端的特有功能适配。

convertor:对象转换器，例如：VO与DTO对象之间的相互转换。

validator:参数校验器，例如：VO请求参数、DTO请求参数非空等其他合法性校验。

2) **客户端层（Client Layer)**：负责对外提供服务的api调用SDK。

api:服务对外提供的api，主要是Dubbo服务的api。

dto:服务对外提供的DTO对象，分为clientobject和data两类。

data:指用于传输服务返回结果的对象。

clientobject:指用于接收请求参数的客户端对象，根据命令查询责任分离（CQRS）架构模式，又可分为Query和Command类型。

3）**应用层（Application Layer）**：主要负责获取输入，组装上下文，参数校验，处理Request，包括Query和Command。调用领域层做业务处理，如果需要的话，发送消息通知、消息消费等。**层次是隔离的，应用层只允许访问领域层，不允许访问基础实施层**。(例如鉴权、事务控制等在应用层实现）

service：应用服务的具体实现，实现服务api接口。提供业务相关服务和任务调度服务。具体服务逻辑实现采用命令模式，由对应的executor来实现。service类只做简单的命令调用。

executor：命令执行器，依据命令command、查询query职责分离架构（CQRS），可分类为xxxCmdExe类和xxxQryExe类，分别处理命令操作和查询类操作。注意：分布式事务处理在executor类中实现，可通过seata框架的@GlobalTransactional注解方式实现。

consumer：消息事件消费，处理外部message。

scheduler：处理定时任务，各种Runner，如：TaskTracker。

convertor: 对象转换器，例如：DTO与Model对象之间的相互转换。

validator: 参数校验器，例如：DTO请求参数、clientobject请求参数非空等其他合法性校验。同一功能不同业务校验规则不一致时，可以使用@Extension组件根据业务、用例、场景三个维度随意组合，实现业务多样性规则校验。

3）**领域层（Domain Layer）**：主要是封装了核心业务逻辑，并通过领域服务（Domain Service）和领域对象（Domain Model）的方法**对App层提供业务实体和业务逻辑计算**。领域是应用的核心，**不依赖项目内部任何其他层次；不依赖任何其他项目二方库Client**。

目前架构采用**贫血模型**，领域模型包括**数据**和**行为能力**，数据由领域模型Model对象负责，同时承担基础的行为能力，行为能力由DomainService、Rule对象负责，承担着绝大部分的模型行为能力。将行为能力分为两类：规则类，Rule负责模型数据校验；行为类，DomainService负责本领域模型业务逻辑处理，整合项目内部其他子领域模型能力，依赖持久层的业务逻辑处理。

领域模型（Domain Model）：领域模型对象（实体、值对象），包括数据和基本行为能力，本领域内部属性定义，提供基本的getter、setter、属性判断、简单业务状态判断。领域模型对象是有状态对象，是有实例变量（数据成员）的对象，可以保存数据，是非线程安全的，需要手动创建对象（实例化）。

领域能力（ability)：由领域对象（Domain Model)、领域规则（Domain Rule)、领域服务（Domain Service)组成。领域能力是无状态对象，没有实例变量的对象。不能保存数据，是不变类，是线程安全的。对象生命周期交由Spring容器管理。

领域对象（Domain Model）：提供基本的getter、setter、constructor、属性判断能力。如果领域能力的动作是一些动词，且隶属于某个实体或值对象，则该能力由对应的领域模型Model提供。

领域服务（Domain Service）:有些领域服务中的动作是一些动词，看上去并不属于任何对象，通过子领域聚合实现领域的业务逻辑,提供各种复杂的领域动作、操作、行为能力，如业务数据重复判断、业务逻辑合法性判断、业务状态流转、业务动作处理、业务操作命令处理。

识别领域服务，主要看他是否满足以下3个特征。

服务执行的操作代表了一个领域概念，这个领域概念无法自然的隶属于一个实体或值对象。

被执行的操作涉及领域中的其他对象

操作是无状态的。

命名规范：以动词开头来命名，动作+Service。例如在银行转账的例子中，转账（transfer）这个行为是一个非常重要的领域概念，但是它发生在两个账号之间，归属账号Entity并不合适，因为一个账号Entity没有必要关联它需要转账的账号Entity。在这种情况下，使用MoneyTransferDomainService就比较合适了。

重复业务逻辑或动作，抽离为组件，能力复用。

领域规则（Domain Rule）：复杂规则校验、多业务场景规则校验，可结合Extension组件实现。

领域仓库（Domain Repository [rɪˈpɒzətri]）：解耦利器，解耦业务逻辑与技术实现细节，本领域的基础设施层能力抽象定义，比如持久层数据库、搜索引擎、文件系统、分布式服务的RPC的能力抽象。**可以依赖DubboClient二方库（调用逻辑在基础设施层RepositoryImpl实现）。**

**领域服务（Domain Service)与领域仓库（Domain Repository）的区别**：

领域服务：聚合项目内部多个领域的能力，实现本领域的业务逻辑，如需调用持久层或外部RPC服务能力，则通过调用领域仓库实现。

**有状态和无状态对象**

　　有状态就是有数据存储功能。有状态对象(Stateful Bean)，就是有实例变量的对象 ，可以保存数据，是非线程安全的。在不同方法调用间不保留任何状态。其实就是有数据成员的对象。

　　无状态就是一次操作，不能保存数据。无状态对象(Stateless Bean)，就是没有实例变量的对象。不能保存数据，是不变类，是线程安全的。具体来说就是只有方法没有数据成员的对象，或者有数据成员但是数据成员是可读的对象。

4）**基础实施层（Infrastructure Layer）**：主要负责技术细节问题的处理，比如数据库的CRUD、搜索引擎、文件系统、分布式服务的RPC等。此外，领域防腐的重任也落在这里，外部依赖需要通过repository的转义处理，才能被上面的App层和Domain层使用。

RepositoryImpl:领域仓库实现，防腐层。整合数据库CRUD、DubboRPC、Cache、search、HttpClient、mq、jobclient等操作。

database:数据库持久层操作

entity:数据库表实体模型

mapper:mybatis数据库映射

cache:缓存操作

config：配置信息

search：搜索引擎

Dubbo RPC：不需要特殊分包，由RepositoryImpl直接调用即可

client: 远程Http接口调用客户端，例如FeignClient调用文件服务

mq: 消息生产者，发布消息事件

JobClient: 提交任务（一次性任务、重复任务、定时任务、cron任务）

**反腐层（Anti-corruption layer）**

反腐层（Anti-corruption layer，简称 ACL）介于新应用和旧应用之间，用于确保新应用的设计不受老应用的限制。是一种在不同应用间转换的机制。

创建一个反腐层，以根据客户端自己的域模型为客户提供功能。该层通过其现有接口与另一个系统进行通信，几乎不需要对其进行任何修改。因此，反腐层隔离不仅是为了保护你的系统免受异常代码的侵害，还在于分离不同的域并确保它们在将来保持分离。

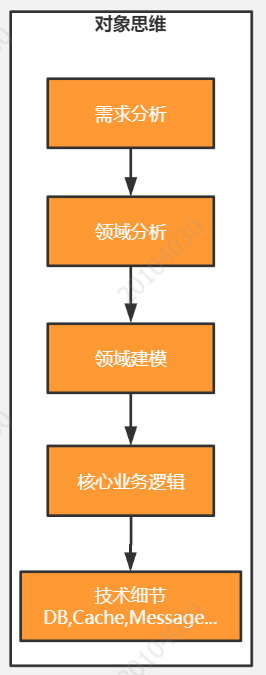
反腐层是将一个域映射到另一个域，这样使用第二个域的服务就不必被第一个域的概念“破坏”。

如果您的应用程序需要处理**数据库或其他应用程序**，这些应用程序的模型不合适或不适用于您自己的应用程序中所需的模型，请使用反腐败层转换为该模型或从您的模型转换为您的模型

我们必须**将这些外部系统或数据存储库视为不同的有界上下文**，当然，它将拥有自己的模型，它将与我们的模型建立关系。

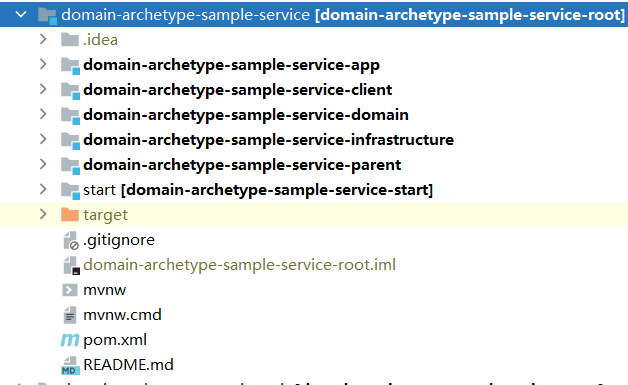
**开发方式**

采用对象思维方式



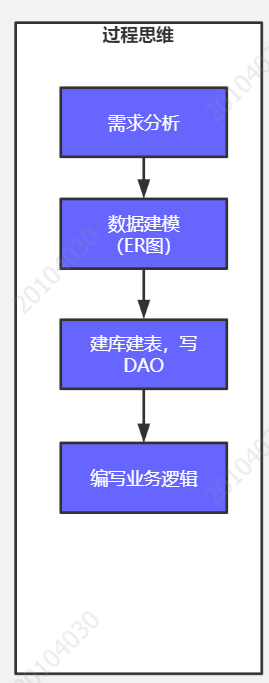
**领域服务层：领域服务项目架构方案**





**基础服务层：业务基础服务项目架构方案**

基础服务架构属于类面向领域驱动设计的微服务架构，不是真正意义上的面向领域驱动设计方式。鉴于基础服务的功能定位是数据库持久层技术实现，DTO对象与业务领域模型最接近，基础服务内部的Model对象与Entity对象更贴近，因此开发方式继续沿用过程思维方式。



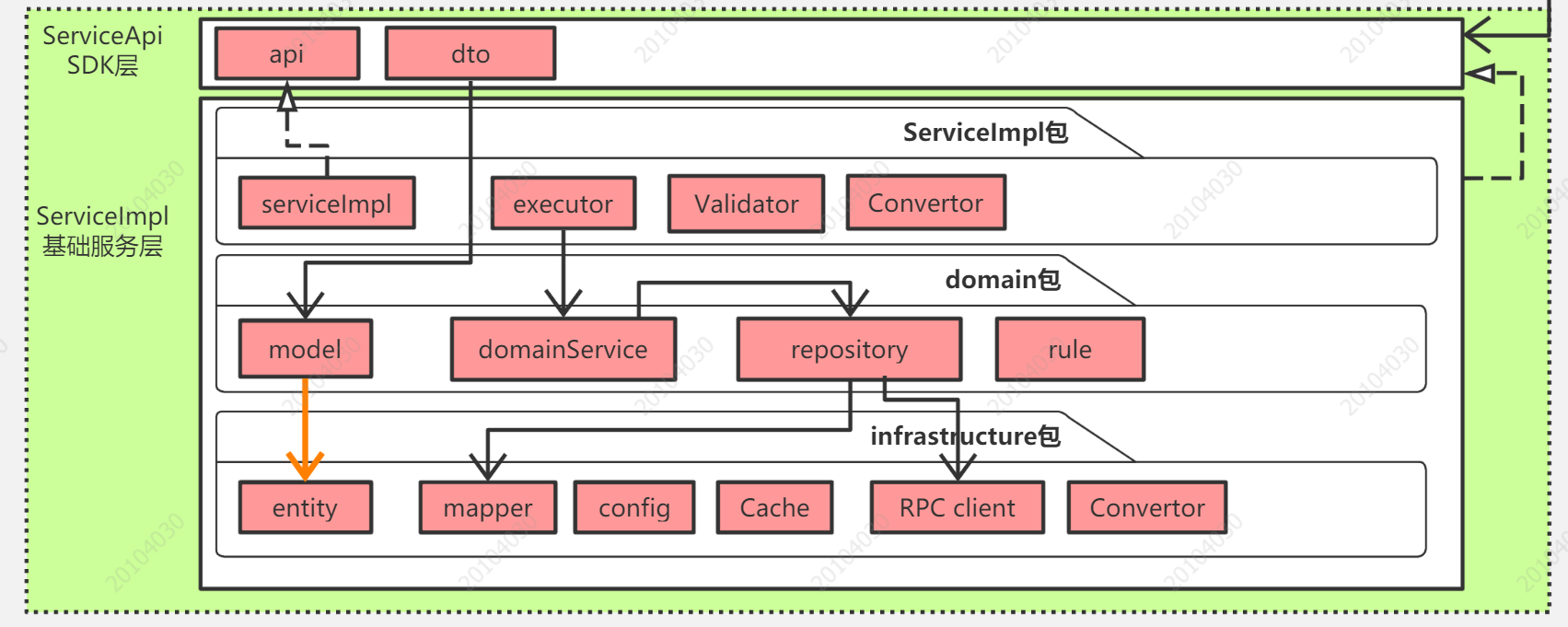
简化版基础服务项目架构，分ServiceApi层和ServiceImpl层，ServiceImpl层横向功能分包+纵向领域分包相结合的方案，横向功能分包：app包、domain包、infrastructure包；纵向在每个功能包下按照子业务领域分包。整合原来领域层Repository接口和基础设施层RepositoryImpl实现类为一个Repository类，位于Domain包下，负责处理聚合基础设施层的能力。

**领域仓库（Domain Repository)与领域仓库实现（Domain RepositoryImpl)的区别**：

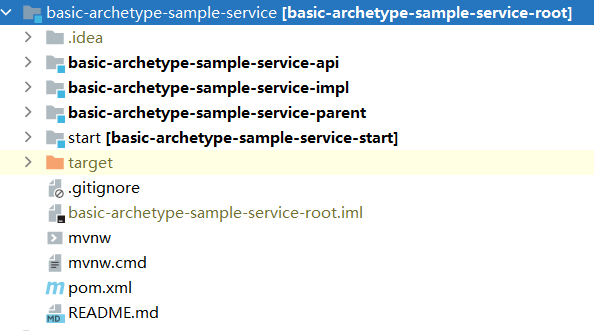
领域服务架构下，domain层Domain Repository 和infrastructure层Domain RepositoryImpl作为领域防腐层，隔离业务领域和技术领域及内部领域与外部领域。基础服务架构下，整合了领域仓库（Domain Repository 和RepositoryImpl)为领域仓库实现（Domain Repository)，位于domain包，不再严格隔离业务领域和技术领域，处理数据库持久层、远程RPC、Cache、MQ等功能。

**领域服务（Domain Service)与领域仓库（Domain Repository）的区别**：

领域服务：聚合项目内部多个领域的能力，实现本领域的业务逻辑，如需调用持久层或外部RPC服务能力，则通过调用领域仓库实现。



**模块结构**



**包结构**

本项目的子领域内部功能分包

├─app

│ ├─convertor

│ │ └─extension

│ ├─executor

│ │ └─query

│ ├─service

│ └─validator

│ └─extension

├─domain

│ ├─domainservice

│ ├─model

│ ├─repository

│ └─rule

└─infrastructure

├─database

│ ├─entity

│ └─mapper

└─mq

**VO、DTO、Model、Entity对象之间关系**

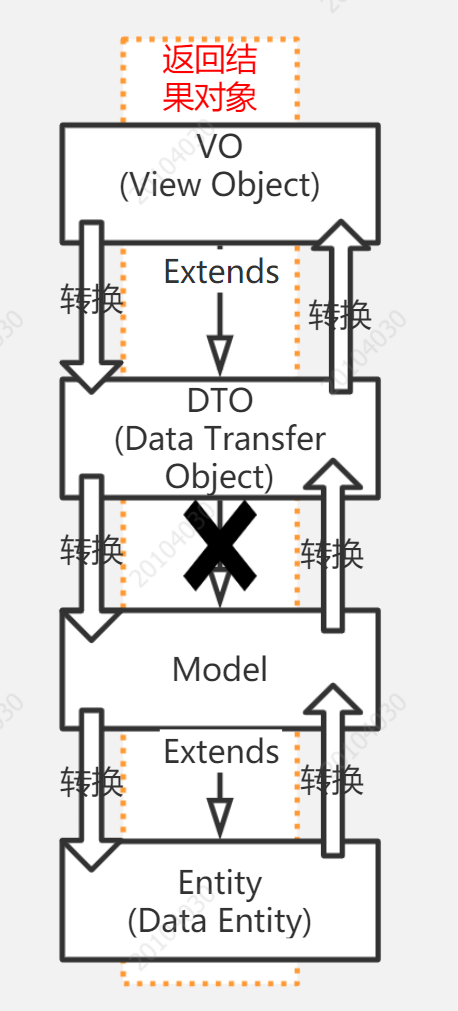
在商城功能架构设计中，领域服务不可以直接访问数据库，基础服务可以访问数据库。因此在基础服务工程中会存在DTO、Model、Entity对象之间的关系。

举例说明：

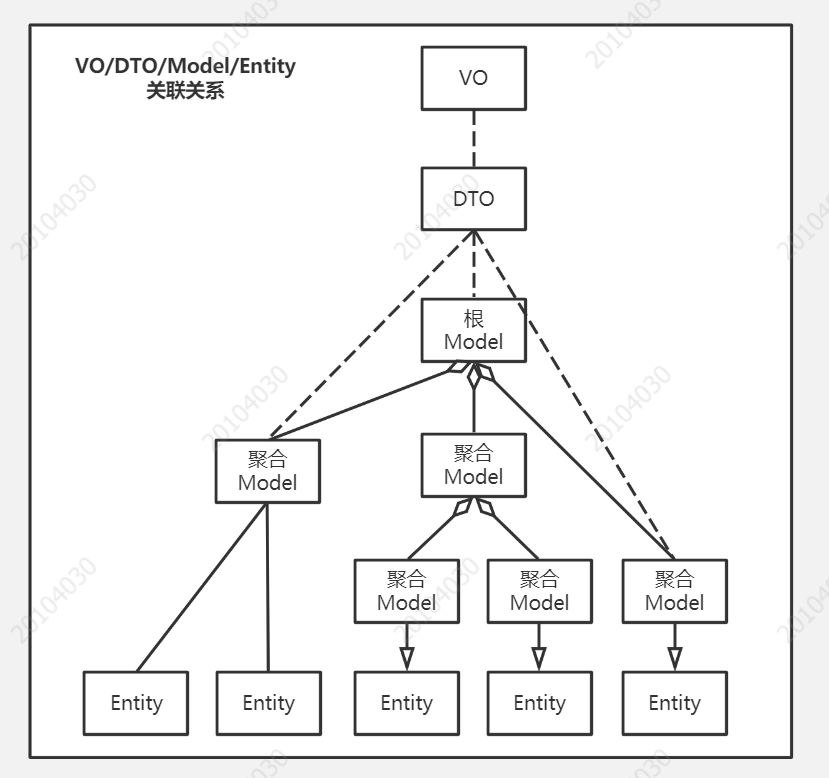
针对单个表的Entity和Model对象之间可以存在继承关系，**Model继承Entity**。但是**DTO和Model对象之间没有关联关系**，即**不可以将Model当DTO来用**。

Model会详细地表示领域模型对象的所有属性，Entity会详细地表示数据库表的所有属性，而DTO是根据应用层暴露的服务特殊定制的数据格式，不可以将领域模型Model直接返回给应用层避免领域数据泄露。

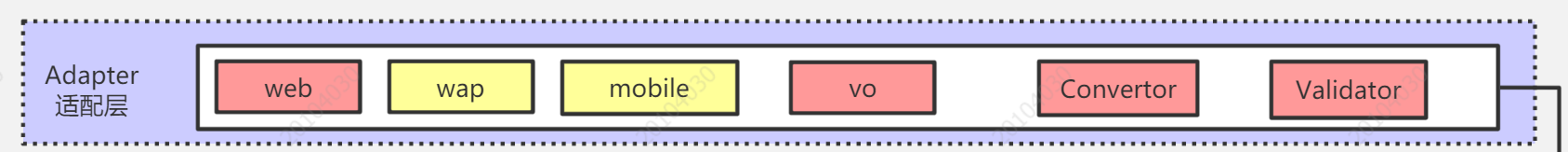
只针对基础服务工程



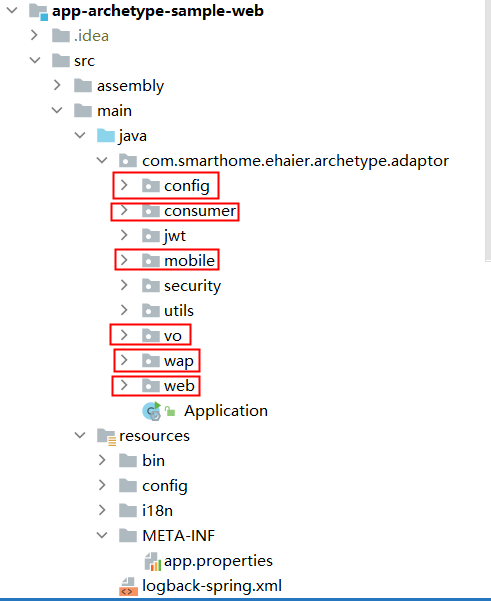
只针对基础服务工程



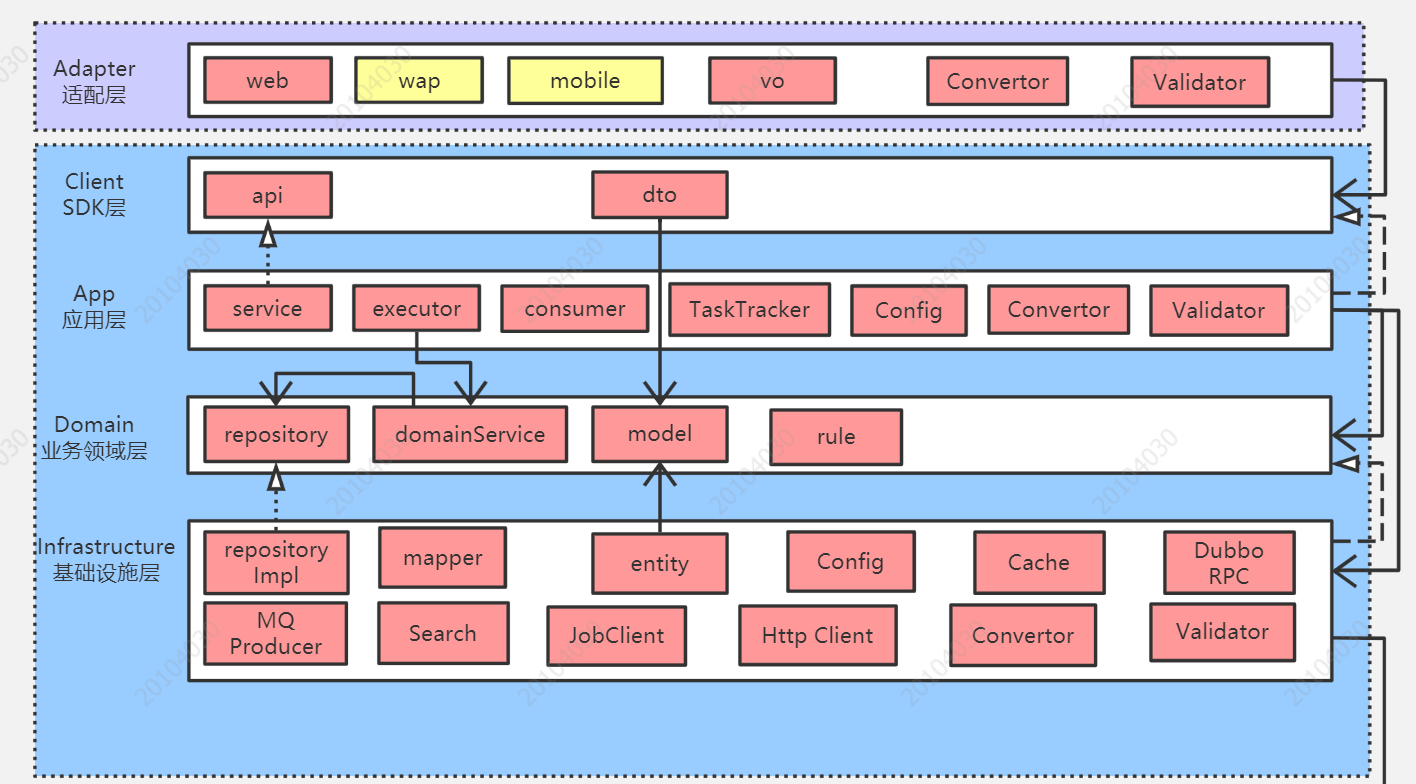
**应用服务层：web应用服务项目架构方案**



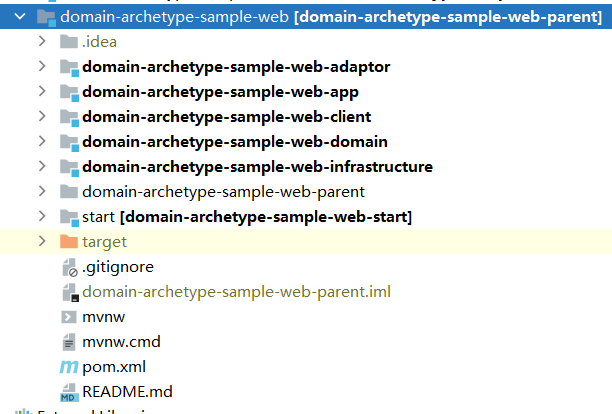
包结构



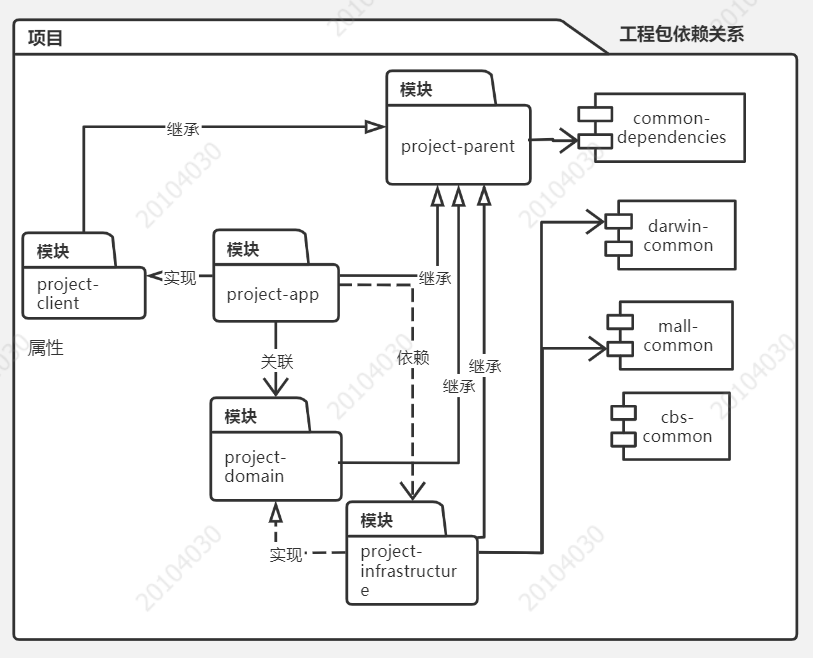
**web应用服务项目架构（单体应用，应用不分层）**



模块结构

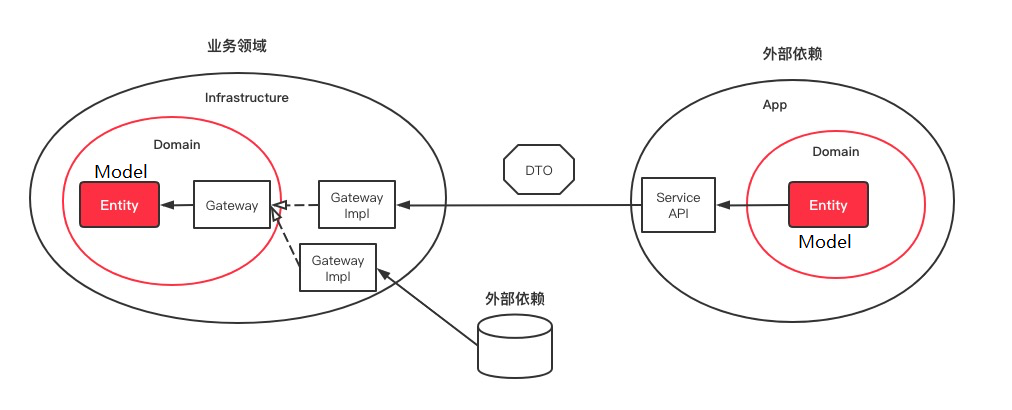


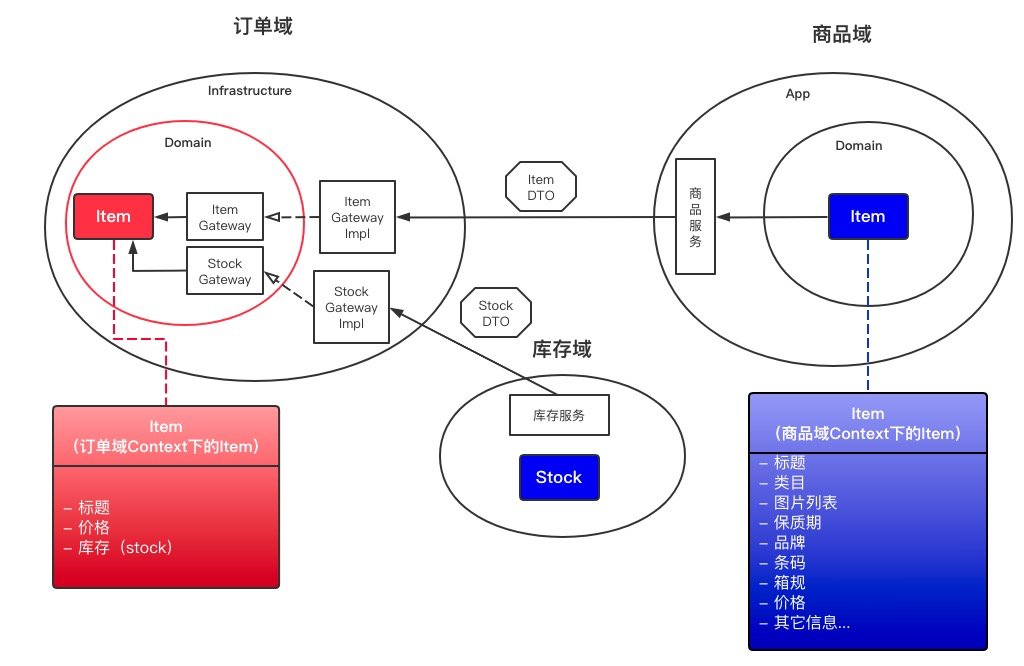
**项目模块依赖关系**



**不同领域跨服务调用**

以下单业务场景为例





**技术开发**

横向分层、纵向领域分包。

横向分层：适配层、应用层、领域层、基础设施层。

纵向分包：采用领域分包方式，取代传统的功能分包。

1. 将应用程序的业务逻辑在领域层实现，技术细节实现在基础设施层实现，服务暴露在应用层实现。

(1)将代码以逻辑方式切割，**服务层只关注应用逻辑**，**领域模型只关注业务逻辑**，**基础设施层只关注技术细节**。

(2)业务逻辑只存在一个地方，容易发现修改，降低维护成本。

(3)服务层的源代码是整洁的，不包含任何复制粘贴代码。

2. 将每个实体服务切割为单一目标的更小的服务。

每个服务类中有一个逻辑组职责。每个服务类的依赖较少，这意味着他们不再是紧耦合的源头。他们是较小的和松耦合的组件。服务类更容易理解，维护和重用。

举个例子：

有一个单一服务类，提供对人员和用户账户的CRUD操作，我们应该将它分为两个独立的服务类：

第一个是对人员的提供CRUD操作

第二个是提供与用户账户相关的操作。

**问题整理**

1、领域模型架构方案：失血模型、贫血模型、充血模型、涨血模型。

解答：目前架构采用**充血模型**，领域模型包括**数据**和**行为能力**，数据由Model对象负责，同时承担基础的行为能力，行为能力由DomainService、Rule对象负责，承担着绝大部分的模型行为能力。将行为能力分为两类：规则类，Rule负责模型数据校验；行为类，DomainService负责本领域模型业务逻辑处理，整合项目内部其他子领域模型能力，依赖持久层的业务逻辑处理。

**失血模型**：模型仅仅包含数据的定义和getter/setter方法，业务逻辑和应用逻辑都放到服务层中。

**贫血模型**：贫血模型中包含了一些业务逻辑，但**不包含依赖持久层的业务逻辑**。这部分依赖于持久层的业务逻辑将会放到服务层中。

**√充血模型**：充血模型中包含了所有的业务逻辑，**包括依赖于持久层的业务逻辑**。所以，使用充血模型的领域层是依赖于持久层，简单表示就是 适配层->应用服务层->领域层<->基础设施层。

**胀血模型**：胀血模型就是把和业务逻辑不想关的其他应用逻辑（如授权、事务等）都放到领域模型中。

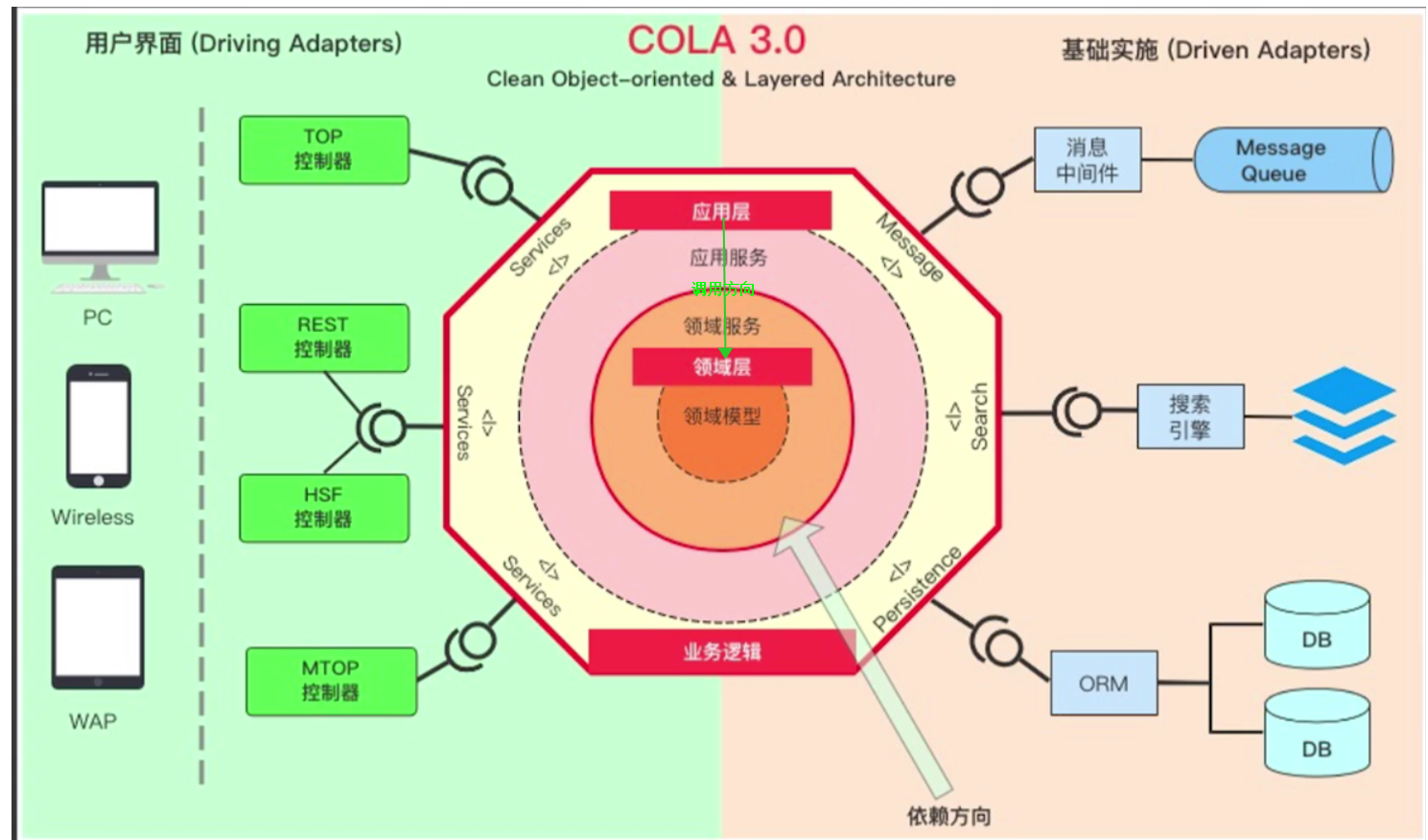
2、应用层、领域层、基础设施层之间的关系，领域层内部各组件的关系？

解答：层次间是隔离的，应用层只允许访问领域层，不允许访问基础设施层，领域层不能访问其他任何层，基础设施层不能被任何层直接调用。即 适配层->应用服务层->领域层。

领域仓库作为防腐层，实现领域限界上下文解耦合、业务逻辑与技术细节解耦合。

领域层各组件层次关系，从上到下为：领域服务、领域规则、领域仓库，只允许从上到下调用，调用关系：领域服务->领域规则->领域仓库，领域仓库不允许调用领域层其他任何组件。同一领域内允许调用其他子领域的领域服务、领域规则、领域仓库。

参考COLA3.0架构



3.Domain模型层说明

解答：Domain模型对外部是黑”盒“，项目内部特有的，它不依赖项目其他任何层，也不依赖任何二方库client，由基础设施层依赖二方库client，RPC调用实现在基础设施层repositoryImpl中实现。依赖外部服务时，不同服务之间通过DTO对象传输数据。项目内部不同子领域之间可以存在关联关系，遵循以下原则：

1）关联尽量少；否则不利于理解和维护单个对象，很难划分对象与对象之间的边界；

2）关联尽量保持单向的关联；

3）在建立关联时，挖掘是否存在关联的限制条件，限制条件能将关联化繁为简，可以将多对多简化为1对多，或将1对多简化为1对1

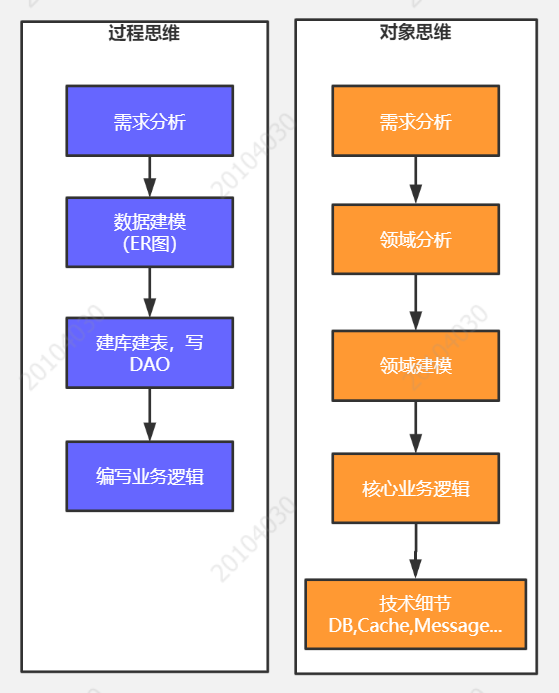
4.业务逻辑聚合是在应用层实现还是在领域层实现？

解答：DDD设计模式的出发点就是分离业务复杂度和技术复杂度，适合业务繁杂的大型系统。该架构方案原则是业务逻辑聚合也是业务实现，在领域层实现，这一点有别于传统的贫血模型架构。

5.基础设施层数据库持久层对象Entity是否可以复用领域层Model对象？

解答：在面向领域的微服务架构（DOMA架构）中数据库持久层对象Entity 可以单向依赖领域层Model对象，但决不允许领域层Model对象反向依赖基础设施层数据库Entity对象。原因：反向依赖违背了业务领域与技术领域隔离的初衷。开发时采用对象思维方式。

特殊说明：基础服务架构有别与严格意义上的领域驱动设计，这里更像是面向对象设计和面向过程设计结合的方式，编程方式更像过程思维方式。因此Model对象可以关联继承Entity对象。



6、命名方案：领域层领域模型对象命名方案？基础设施层数据访问对象命名方案？适配层无线端/APP端适配命名方案？

领域层领域模型对象命名方案：

**Model（模型）** xxx+Model后缀

基础设施层数据访问对象命名方案：

**Entity（实体bean，ORM对象关系映射）** xxx+Entity后缀/或无后缀

适配层无线端/APP端适配命名方案：

**mobile(移动互联网终端）**

领域网关/领域仓库命名方案:

**领域仓库Repository/RepositoryImpl**

Model模型的id命名方案：

**id**

7.领域模型关联关系的表示形式

解答：使用领域模型关联。优点：模型抽象更贴近业务领域中的业务实体对象，容易理解领域模型定义和子领域间依赖关系。缺点：DTO、Entity、Model对象之间转换工作复杂度可能会增加。

引申：使用领域模型ID（主外键）关联。优点：更贴近技术实体对象，弱关联，减少对子领域模型的依赖度，减少在DTO、Entity、Model对象之间转换的工作量。缺点：模型之间依赖关系不是非常清晰，增加了模型理解难度。这里不采用。

8.领域服务命名方案，提供哪些能力？

解答:以动词开头+Service，只提供某一组逻辑的领域能力。例如：PlaceOrderService，AddUserService，QueryOrderService。以名词开头+Service,例如TmallOrderService,OrderService,ItemListService,ItemDetailService,UserService,LoginService。两种方案相结合，原则是逻辑相关性强的一组能力组合，分离相关性不高的能力。当类内的方法太多时或者总代码行数超过1000时，需要做能力拆分，避免领域服务成为业务过度耦合的重灾区，造成代码易读性降低。

The End

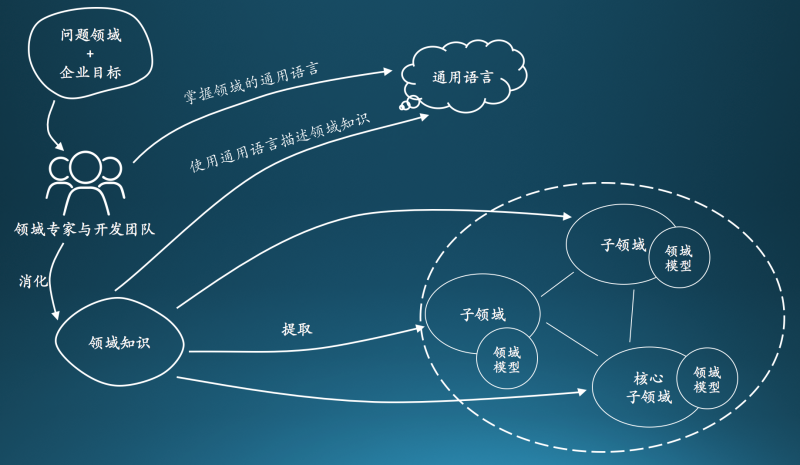
**知识扩展**

**领域驱动设计**

2004年Eric Evans 发表Domain-Driven Design –Tackling Complexity in the Heart of Software （领域驱动设计），简称Evans DDD。领域驱动设计分为两个阶段：

以一种领域专家、设计人员、开发人员都能理解的通用语言作为相互交流的工具，在交流的过程中发现领域概念，然后将这些概念设计成一个领域模型；

由领域模型驱动软件设计，用代码来实现该领域模型；



**领域和领域模型**

领域Domain属于问题空间，领域模型Domain Model属于解决方案空间

**领域**

**Sub Domain子领域**

一个领域（Domain）会拆分为多个子领域（Sub Domain）

**Core Domain核心领域**

**Supporting Subdomain支撑子领域**

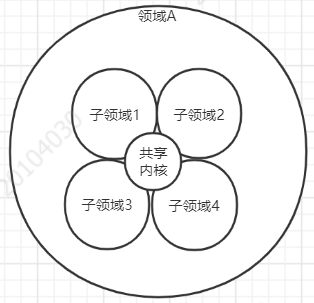
子领域中最核心叫Core Domain，其他的是支撑子域

**Generic Subdomain通用子域**

**Shared Kernel共享内核**

有些支撑子域比较特殊，因为它解决的是一类通用问题，比如账号和权限；这类子域我们叫做通用子域（Generic Subdomain）；通常，通用子域对应的Bounded Context限界上下文，会跨域多个子域；

多个子领域有时会有相交的部分，我们称作共享内核（Shared Kernel）；体现到代码上，就是同一份代码，在两个领域模型中复用；



**Bounded Context限界上下文**

**上下文映射图**

**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。

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。

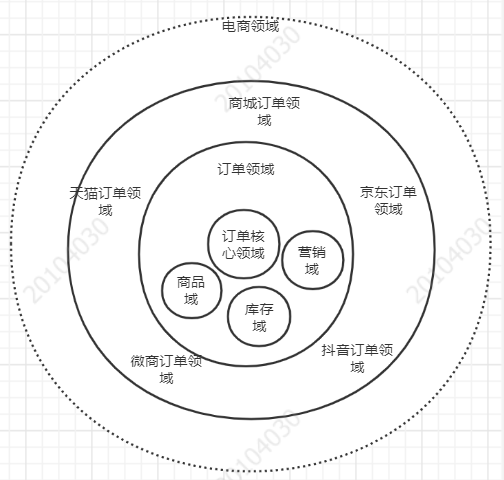
发布语言（Published Language）：通常与OHS一起使用，用于定义开放主机的协议。

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。

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，无论是上下文内部交互（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。

业务领域示例图



**边界是核心**

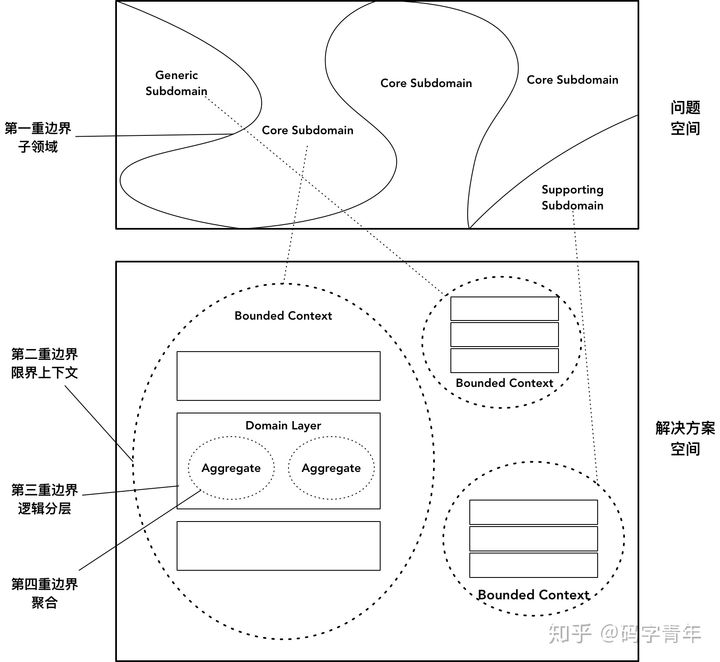
**第一重边界**:从分析需求一开始，我们就需要通过确定项目的愿景与目标，划定问题空间，由此确定核心子领域、通用子领域与支撑子领域。

**第二重边界**:进入解决方案空间,战略设计获得的限界上下文。通过它可以有效地降低系统规模。暴露在限界上下文边界外部的是远程服务或应用服务，每个服务都提供了完整的业务价值。微服务设计。

**第三重边界**：在限界上下文内部，基础设施层、应用层与领域层之间的隔离。

六边形架构为例，有效地隔离了业务复杂度与技术复杂度，将领域层作为整个系统稳定而内聚的核心，分离变与不变，逐渐将这个“领域内核”演化为企业的重要资产。

**第四重边界**：若要维持领域内核的稳定性，高内聚与低耦合是其根本要则。总是在调整与修改的领域模型无法维护领域概念的完整性和一致性；为此，领域模型引入了聚合这一最小的设计单元，它从完整性与一致性对领域模型进行了有效的隔离。聚合设计原则要求聚合之间通过ID进行关联，避免了聚合根实体之间的引用依赖，也不会受到限界上下文边界变化的影响。



**纪律是关键**

三大纪律

领域专家与开发团队工作在一起

领域模型必须遵循统一语言

时刻坚守四重设计边界

八项注意

子领域与限界上下文不要混为一谈

一个限界上下文不能由多个团队开发

跨进程协作通过远程服务，进程内协作通过应用服务

保证领域分析模型、领域设计模型与领域实现模型的一致

不要将领域模型暴露在应用层之外

不要让数据模型干扰领域模型的设计

聚合之间只能通过聚合根ID引用

聚合不能依赖访问外部资源的仓库

**领域驱动设计能力评估模型**

能力评估模型针对的能力维度包括：

敏捷迭代能力

领域建模能力

架构设计能力

整洁编码能力

成熟级能力模型：

**敏捷迭代能力**

自组织的特性团队，团队成员定期轮换，形成知识共享建立了故事地图、建立了史诗故事、特性与用户故事的需求体系建立了可视化的看板，由下游拉动需求的开发，消除浪费。

**领域建模能力**

采用事件风暴、四色建模等建模方法，由领域专家与开发团队一起围绕核心子领域开展领域建模。

**架构设计能力**

建立了系统层次与限界上下文层次的系统架构，遵循了整洁架构，建立了清晰的限界上下文与领域层边界，建立了以聚合为核心的领域设计模型，职责合理地分配给聚合、资源库与领域服务。

**整洁编码能力**

具有清晰表达的领域含义。方法和类的命名都遵循了统一语言，可读性高。采用测试驱动开发编写领域代码，遵循简单设计原则具有明确的测试战略，单元测试先行。

The End