6 - 图片优化 - 智能协同云图库项目教程 - 编程导航教程

> 本节重点之前我们已经完成了本项目的功能开发。

本节重点

之前我们已经完成了本项目的功能开发。由于本项目功能丰富、代码量大，如果是在企业中维护开发的项目，传统的 MVC 架构可能会让后续的开发协作越来越困难。所以本节鱼皮要从 0 带大家学习一种新的架构设计模式 —— DDD 领域驱动设计。

大纲：

- 软件架构模式的演进
- DDD 领域驱动设计概念
- DDD 架构设计
- 项目 DDD 重构

通过本节，你将掌握 DDD 领域驱动架构设计，掌握快速的、标准的、通用的重构传统 MVC 项目为 DDD 架构项目的方法，学会之后几乎任何项目都能轻松改造为 DDD 项目。

一、软件架构模式的演进

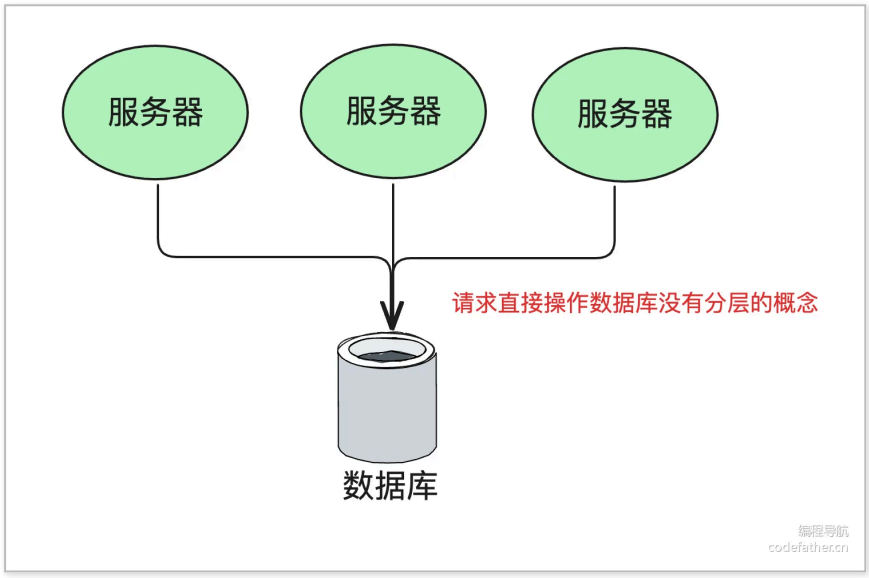
在学习 DDD 之前，我们需要先知晓软件架构模式的演进之路。

为了应对软件系统日益复杂化的需求，从最初的简单传统单体架构到如今复杂的分布式和微服务架构，软件架构的演变经历了多个阶段，主要有以下三个典型阶段：

传统单体架构

所有的应用功能都集成在一个单一的应用程序中，所有模块和组件都在同一个进程内运行，请求直接操作数据库，不进行代码分层，易于开发和部署，尤其适合小型或简单的应用。

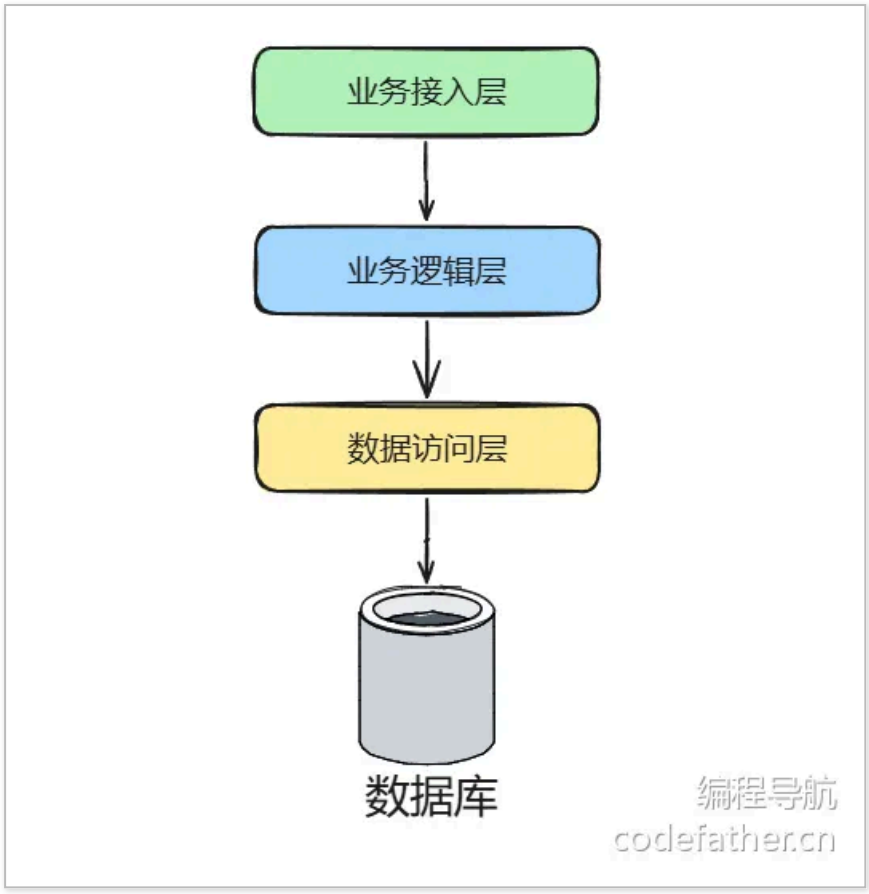
随着业务增长和需求变更，单体架构变得难以扩展和维护。不同功能的模块耦合在一起，导致更新某个功能可能影响到整个系统。



分层架构

应用被划分为不同的层（如业务接入层、业务逻辑层、数据访问层等），每一层负责特定的功能，层与层之间通过接口进行交互，促进了模块化和职责分离，便于管理和维护。

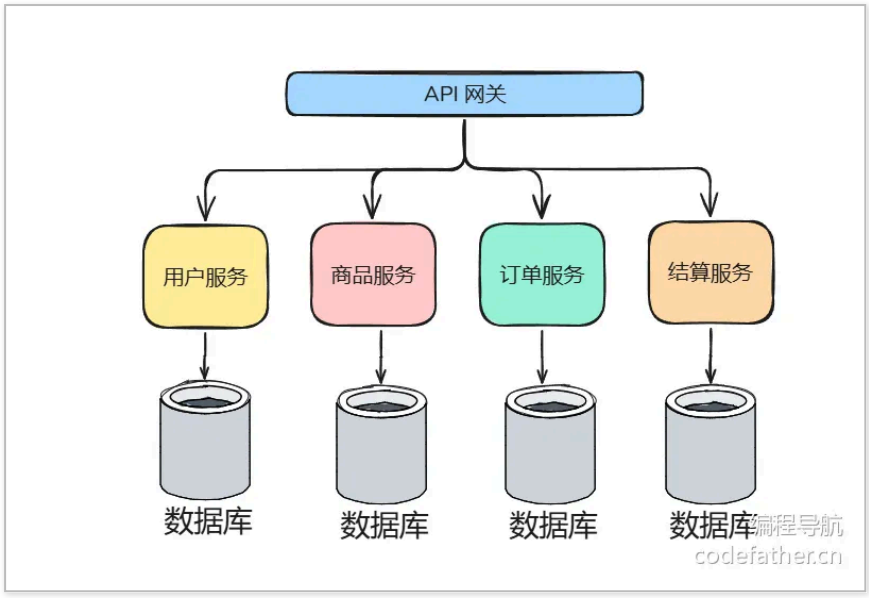
但层与层之间的紧密耦合限制了灵活性，且随着系统的复杂度增加，可能导致性能下降和维护难度增加，并且它的可扩展性和弹性伸缩性差。



微服务架构

将系统拆分为多个小而独立的服务，每个服务负责处理一组特定的功能，每个服务通常由独立的团队开发、部署和维护，服务之间通过轻量级协议（如 HTTP、自定义协议或消息队列）进行通信。

服务之间独立，易于扩展和维护。每个微服务都可以独立部署、开发和扩展，且易于使用不同的技术栈。



二、DDD 领域驱动设计概念

什么是 DDD?

DDD（领域驱动设计，Domain-Driven Design） **是一种软件开发方法论和设计思想**。DDD 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的边界**，保证业务模型和代码模型的一致性。

因为 DDD 主要应用在微服务架构场景，所以想要更好的理解 DDD 的概念，需要结合微服务架构来看：

- DDD 是一种设计思想，**确定业务和应用的边界**
- 微服务架构需要 **将系统拆分为多个小而独立的服务**

是不是有点感觉了？已经知道 DDD 是用来做什么的了？

微服务的拆分一直是业界的一个难题：微服务拆分的粒度应该多大？服务到底应该如何拆分？服务之间的边界如何定义？

有人可能认为，微服务不就是拆就完事了？不需要管那么多！实际上微服务的拆分是门 “艺术”：

- 服务拆分的太细，项目复杂度会过高，接口的调用成本、服务运维成本大幅上升。

- 服务拆分的太粗，业务边界变得模糊，服务的耦合度还是过高，失去了微服务的优势。

而 DDD 就是一个方法论，指导我们根据领域模型确定业务的边界，从而划分出应用的边界，最终落实成服务的边界、代码的边界。

本课程虽然没有涉及到微服务，但是不妨碍利用 DDD 思想拆分代码架构。最终想要变成微服务架构仅需抽离包中的代码独立部署即可。

DDD 的目标

1. 通过领域模型实现业务需求：开发者与领域专家共同理解业务需求，形成共享语言并构建模型。
2. 提高系统的灵活性与可维护性：通过合理划分限界上下文，减少系统的耦合度，使得不同模块或子系统可以独立演化。
3. 支持复杂业务逻辑的表达：通过深入的业务建模，使得复杂的业务逻辑能够清晰、准确地反映在代码中。

总结一下，就是让系统更贴合业务，让大型系统更利于独立建设和维护。

DDD 的适用场景

- 业务复杂的系统：如金融系统、电商平台等，涉及的业务逻辑复杂且频繁变化。
- 需要与多个部门或团队合作的项目：DDD 强调跨部门协作，适用于多方参与的大型项目。
- 长周期、长期维护的项目：DDD 强调可维护性与演化，适合需要长期维护和扩展的系统。

总结一下，大型的、跨部门协作的、长期维护的复杂项目。

DDD 的建设

DDD 会先建立领域模型，根据业务划分领域边界，进而确定微服务的边界，然后再根据领域分块编码实现。

实际上 DDD 的建设包括 **战略设计** 和 **战术设计** 两部分。

下面这些内容对没有参加过企业工作的同学来说会有些难理解，学习时可以跳过。

战略设计

从业务出发，建立领域模型，统一限界上下文。

设计时，需要先进行事件风暴（类似于头脑风暴），邀请领域专家、架构师、开发人员、测试人员、产品经理、项目经理等团队人员一起参加讨论。

描述个场景，大家在会议室里，搞一个大白板，参与者们将自己的想法和意见写在贴纸里并罗列到白板上，大家 **先发散思维** 进行讨论、记录。

主要讨论的内容是：系统会涉及哪些业务，哪个业务动作会触发另一个业务的什么动作，其间的输入是什么？输出是什么？

通过这类分析把所有的业务、业务行为、业务结果都罗列出来，拆分出领域模型中的事件、命令、实体等领域对象。然后梳理这些领域对象之间的关系，从不同维度进行聚类，形成聚合、聚合根、限界上下文等，这个过程就是 **收敛。** 限界上下文可以简单理解为微服务的边界，将其映射到代码模型，就完成了微服务的拆分。

💡 事件风暴实际上会利用常见的产品设计和用户体验分析方法，比如：

- 用例分析：对系统功能需求进行描述，以确定系统如何与外部参与者（即用户或其他系统）进行交互
- 场景分析：通过设定具体的情境或情景，来探讨用户如何在不同的环境下使用产品或系统
- 用户旅程分析：从用户的角度，描绘用户在使用产品或服务的过程中，从开始到结束的一系列步骤或行为

战术设计

从技术实现出发，将领域模型和代码模型进行映射

这个阶段就是完成代码落地，包括聚合、聚合根、实体、值对象等代码逻辑的设计与实现。

DDD 体系名词解析

1、领域

领域指系统关注的业务领域或问题空间，具体的领域与公司或组织的核心业务有关。

实际上在 DDD 中 领域就是用来确定范围，而范围就是边界。

一个领域又可以分为多个子领域，每个子领域代表系统的一部分业务。

而子域根据重要程度和功能特性，可划分为：

- 通用域：指系统中一些通用的、不特定于某一业务的领域，它们在多个不同领域或系统中都有应用。（例如支付、日志管理）
- 支撑域：指在系统中起到支持作用，但并不是直接驱动业务价值的部分（例如网关）
- 核心域：指系统中最关键的部分，是业务的核心竞争力所在，能够为企业带来最大的价值

💡 这里需要注意，在不同业务（公司中）三类子域是有区别的，例如在普通公司中需要调用第三方支付，那么支付是通用域，但是对于支付公司（例如支付宝）来说支付是它们的核心域。

2、限界上下文

是指一个明确的边界，规定了某个子领域的业务模型和语言，确保在该上下文内的术语、规则、模型不与其他上下文冲突。

在事件风暴讨论过程中，我们需要完成通用语言的统一。例如电商场景下，我们统一叫物品为商品、将用户购买商品的行为叫下单。

我们都知道语言需要有语义环境。不同语义环境下，同一个语言表达的意思是不同的。比如：

- “我吃得很饱，现在不能动了”：这里的 “吃得很饱” 表示的是 “吃到肚子很满”，字面意思是 “我已经吃得很饱了，吃不下了”
- “我吃得很饱，今天的演讲让人充实”：这里的 “吃得很饱” 并非字面上的 “吃得饱”，而是比喻 “得到了很大的满足”，表现出内心的充实感。

而限界上下文实际上就类似于语义环境。通用语言需要业务边界，限界上下文就是定义了业务的边界，也就是领域的边界。

电商语义下称之为商品的东西，到运输语义下它就变成了货物。因此我们需要明确限界上下文，在这个上下文中团队内部人员对某一领域对象、领域事件的认知是一致的、没有歧义的。

3、实体

一般业务对象，且具有唯一标识对象都是实体。在代码中所谓的唯一标识就是 ID，例如，订单有订单 ID，用户有用户 ID，它们都是典型的实体。

实体的关键点就在于唯一标识，随着生命周期的变化，实体中的属性可能会改变，例如订单可以从未完成变成已完成，但是其 ID 不会改变。

实体映射到代码中就是实体类。通常采用 充血模型 来实现，即与这个实体相关的所有业务逻辑都写在实体类中。

如果需要跨多个实体才能完成的业务逻辑，会写在领域服务中。

4、值对象

值对象没有唯一标识，创建后就不允许修改了，只能用另外一个值对象来进行 整体替换。通常用于描述对象的属性，用于对实体的状态和特征进行描述。

非常典型的值对象就是地址。比如用户实体对象有地址这个属性，那么这个地址就是值对象，它没有唯一标识，且创建后就不允许修改其本身的值。如果用户需要修改地址，那么这个属性是被整体替换的（换新的地址值对象）。

拥有这样特性的对象，就是值对象。

💡 实体和值对象并不是一成不变的，比如对电脑主机来说，显卡是一个值对象，显卡坏了就换一个，而对显卡厂商来说，显卡是实体，它们有编号需要追踪和管理的。

5、聚合

实体和值对象是基础的领域对象，聚合将多个实体和值对象组合成一个整体，实现高内聚低耦合。

简单来说实体和值对象是个体，个体与个体之间的合作需要被“领导”，而聚合就是将它们组织起来协同工作，这样才能保证聚合内数据的一致性（组织统一口径）。它可以作为微服务拆分的最小单位。

聚合还是数据修改和持久化的基本单位，实现数据的持久化存储。

6、聚合根

聚合根就好比聚合内的带头人，聚合内的多个实体不会直接对外提供接口访问，而是由聚合根统一提供对外接口。

一个聚合内只会有一个聚合根，聚合根通过对象引用的方式组织聚合内的实体和值对象，聚合根之间的合作是通过 ID 关联的。

这里需要注意：聚合根也是一个实体，也具有业务属性和业务逻辑和唯一标识。

例如订单域内只有订单和订单子项两个实体，那个订单就是这个域中的聚合根。

7、领域服务

聚合根可以实现跨多个实体的复杂业务行为，但是为了实现高内聚和低耦合，聚合根内部应该更聚焦与自身强关联的业

务行为，复杂的跨多实体的业务可以放在领域服务中实现。

领域服务是指那些 **不能归属于某个单一实体或值对象，但又属于领域模型的一部分** 的业务逻辑。领域服务封装了对领域对象进行操作的核心业务规则，通常用于处理跨多个实体的操作，或者当业务逻辑无法直接归属于某个特定聚合时。

例如一个订单系统，需要处理订单支付功能，而支付涉及订单、用户账户、支付信息等多个实体，这个支付操作不太好归属某个实体，这样的逻辑就可以放到领域服务中。

```
public class PaymentService {
    public void processPayment(Order order, PaymentDetails paymentDet


    }
}
```

那聚合根更适合怎样的跨实体的业务呢?

例如你有一个 "订单" 聚合，其中包含订单条目、支付信息等，Order 作为聚合根，负责管理订单条目和确保订单的完整性。你不能直接访问订单条目（如 OrderItem），必须通过 Order 聚合根来进行操作。

```
public class Order {
    private List<OrderItem> items;
    private PaymentDetails paymentDetails;

    public void addItem(OrderItem item) {

        this.items.add(item);
    }

    public List<OrderItem> getOrderItems() {

    }


}
```

DDD 建模总结

结合上面的名词解析，我们回顾一下 DDD 建模的流程。

首先我们需要领域建模，此时会进行事件风暴，通过用例分析、场景分析等方式列出所有的业务行为与事件，找出产生这些行为的领域对象，包括实体与值对象。梳理这些领域对象之间的关系，从实体中找出聚合根，再根据聚合根的业务，找寻与其业务紧密关联其它实体与值对象，从而形成聚合。多个聚合之间根据业务相关性又可以划出限界上下文。

可以通过 “开公司” 的比喻来帮助大家理解 DDD。领域就像公司的行业，决定了公司所从事的核心业务；限界上下文是公司内部的各个部门，每个部门有独立的职责和规则；实体是公司中的员工，具有唯一标识和生命周期；值对象是员工的地址或电话等属性，只有值的意义，没有独立的身份；聚合是部门，由多个实体和值对象组成，聚合根（如部门经理）是部门的入口，确保部门内部的一致性；领域服务则是跨部门的职能服务，比如 HR 或 IT 服务，为各部门提供支持和协作。

三、DDD 架构设计

充血模型和贫血模型

贫血模型和充血模型是两种面向对象设计模式，用于描述对象的职责划分和对象是否包含行为逻辑。

我们常见的对象内部的实现非常简单，仅包含数据属性和简单的 `getter` / `setter` 方法，换句话说，这些对象是一个纯粹的 “数据容器”，它仅负责保存数据，而不包含任何业务行为。

从领域模型设计角度来说，这样的设计称为贫血模型，偏向于传统分层架构的设计；与之对应的是充血模型，强调面向对象的系统设计。

两种模型的分类本质是对领域对象中 “数据与行为的职责划分” 的不同理解。反映了在软件设计中，如何组织领域对象的数据和行为，以及如何分配业务逻辑的不同设计思路。

充血模型是指领域对象不仅包含数据（属性），还包含处理这些数据的业务逻辑。换句话说，充血模型的领域对象是 “充血” 的，它们不仅有状态（数据），还有行为（业务方法）。

贫血模型则是指领域对象仅包含数据，不包含任何业务逻辑，所有的业务逻辑都放在单独的服务类中（通常是应用层或领域服务层）。领域对象本身是“贫血”的，只有状态，没有行为。

总结来看：

- 充血模型 适合复杂业务，业务逻辑和数据紧密结合，符合面向对象设计的原则。
- 贫血模型 适合简单业务，关注点分离，数据和业务逻辑分开，领域对象仅负责存储数据，服务类负责业务逻辑。

下面用代码举例，大家就知道它们的区别了。

代码示例

假设我们有一个订单系统，Order 是领域对象，包含了订单的状态和相关的业务逻辑。

1）充血模型代码示例

在充血模型中，Order 对象包含了业务逻辑（如 pay 和 cancel 方法），这些方法对订单的状态进行操作，直接将数据和行为结合在一起。

```
public class Order {
    private String orderId;
    private double totalAmount;
    private boolean isPaid;

    public Order(String orderId, double totalAmount) {
        this.orderId = orderId;
        this.totalAmount = totalAmount;
        this.isPaid = false;
    }

    public void pay() {
        if (this.isPaid) {
            throw new IllegalStateException("Order is already paid");
        }
        this.isPaid = true;
    }

    public void cancel() {
        if (this.isPaid) {
            throw new IllegalStateException("Cannot cancel a paid ord
        }

    }
```

```
    public boolean isPaid() {
        return isPaid;
    }

    public double getTotalAmount() {
        return totalAmount;
    }
}
```

2）贫血模型代码示例

在贫血模型中，Order 对象只包含数据（状态），而所有的业务逻辑（如 payOrder 和 cancelOrder）都被移到了外部的 OrderService 服务类中。

```
public class Order {
    private String orderId;
    private double totalAmount;
    private boolean isPaid;

    public Order(String orderId, double totalAmount) {
        this.orderId = orderId;
        this.totalAmount = totalAmount;
        this.isPaid = false;
    }

    public String getOrderId() {
        return orderId;
    }

    public double getTotalAmount() {
        return totalAmount;
    }

    public boolean isPaid() {
        return isPaid;
    }

    public void setPaid(boolean paid) {
        isPaid = paid;
    }
}

public class OrderService {
    public void payOrder(Order order) {
        if (order.isPaid()) {
            throw new IllegalStateException("Order is already paid");
        }
        order.setPaid(true);
    }

    public void cancelOrder(Order order) {
        if (order.isPaid()) {
            throw new IllegalStateException("Cannot cancel a paid ord
        }
```

```
    }
}
```

二者对比

特点	贫血模型	充血模型
封装性	数据和逻辑分离	数据和逻辑封装在同一对象内
职责分离	服务类负责业务逻辑，对象负责数据	对象同时负责数据和自身的业务逻辑
适用场景	简单的增删改查、DTO 传输对象	复杂的领域逻辑和业务建模
优点	简单易用，职责清晰	高内聚，符合面向对象设计思想
缺点	服务层臃肿，领域模型弱化	复杂度增加，不适合简单场景
面向对象原则	违反封装原则	符合封装原则

在实际项目中，贫血模型和充血模型并非互相排斥。通常可以结合两者的优点：

- 使用充血模型作为领域模型，封装复杂的业务逻辑。
- 使用贫血模型作为数据传输对象（DTO），在系统之间传输数据。

扩展知识 - 缺血模型和涨血模型

1）缺血模型

上面贫血模型的示例可以视为缺血模型的一种表现形式。缺血模型实际上是贫血模型的进一步简化或极端化版本。

在缺血模型中，不仅对象没有业务逻辑，甚至服务层也缺乏真正的业务逻辑，系统的整体设计趋向于 CRUD（增删改

查）开发，会将所有逻辑转移到外部。

需要注意的是，领域对象不包含任何业务逻辑即可称为贫血模型，无需刻意强调是否属于缺血模型，除非是在贫血模型与缺血模型对比的语境中。

2）涨血模型

涨血模型则是充血模型的极端化表现，不仅将所有核心业务逻辑集中于领域模型中，甚至连非核心逻辑（如数据库事务处理、权限校验等）也全部包含其中。

在实际应用中，**缺血模型和涨血模型并不常用，这里仅做扩展了解**。我们通常只需关注贫血模型和充血模型的设计取舍即可。

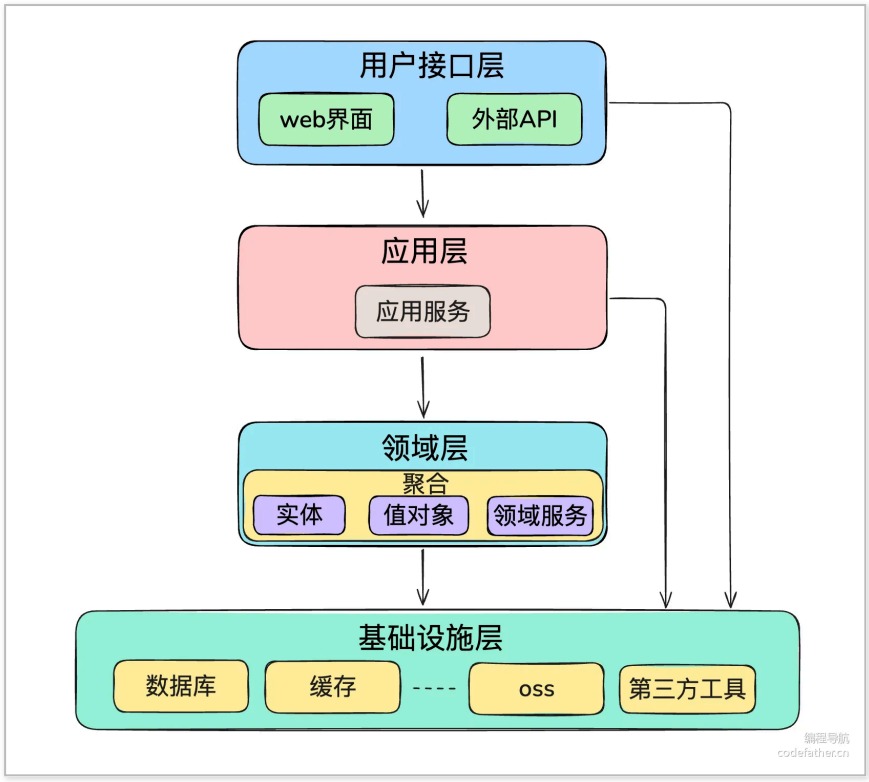
DDD 的分层架构

在领域驱动设计（DDD）中，分层架构模型是一种常见的设计模式，用于组织和管理系统的复杂性。通过将应用分为不同的层次，每一层都有清晰的责任和角色，从而促进了代码的高内聚、低耦合和可维护性。

DDD 的分层架构主要有四层：**用户接口层**、**应用层**、**领域层**、**基础设施层**。每层负责不同的职责，协调工作以实现系统的整体功能。

除基础设施层外，严格来说每层只能与 **直接下层** 产生依赖，即领域层只能被应用层调用，应用层只能被用户接口层调用。

当然也有 **松散分层架构**，层与层之间的依赖和交互更加灵活，不严格分隔。适用于快速开发，但随着系统复杂度的增加，可能变得难以维护。



1）用户接口层

也叫表示层或 Web 层，主要负责与外部（用户、API 等）的交互。它的主要职责是接收用户输入并返回系统的输出。表示层不包含业务逻辑，而是将用户的请求转发到应用层处理，并将处理结果返回给用户。

2）应用层

应用层主要用来协调领域层的逻辑和基础设施层的资源。应用层不包含业务规则或业务逻辑，但会调用领域层的服务进行服务编排与组合，来实现特定的业务。

如果有对其他服务的远程调用，也放在这层实现。除此之外，权限校验、事务、事件等操作也都可以放在这层进行实现。

3）领域层

领域层是整个架构的核心，包含了应用的业务逻辑、规则和策略。它定义了核心的领域模型，包括聚合根、实体、值对象、领域服务等。

领域层的目的是将业务需求转化为代码，并确保业务规则在应用中得以执行。该层的设计强调与业务领域的紧密耦合，是 DDD 中的重点。

4）基础设施层

基础设施层提供技术支持和持久化服务，采用依赖倒置设计，封装基础资源。负责与外部系统（如数据库、消息队列、缓存等）的交互。基础设施层的主要职责是实现应用层和领域层所需要的技术服务，如数据存储、邮件发送、日志记录等等。

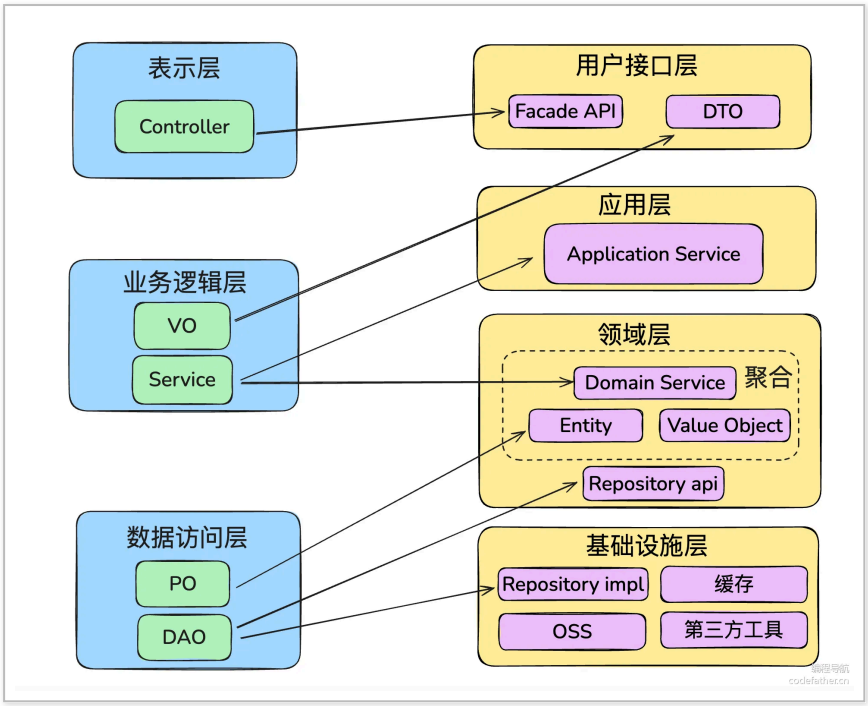
依赖倒置设计实际上指的是各层对基础资源（如数据库）仅依赖其接口而不是具体的实现，假设后续替换基础资源（数据库），仅需替换具体实现，不需要修改各层依赖的代码。

三层架构到 DDD 四层架构的转化

三层架构是传统的架构模式，结合 SpringMVC 通常由以下三层组成：

- 表示层（Controller 层）：处理 HTTP 请求，调用业务层的服务，返回视图或数据。
- 业务逻辑层（Service 层）：封装核心业务逻辑，执行业务操作。
- 数据访问层（Repository 层）：负责与数据库交互，执行数据的持久化和查找。

转化 DDD 四层架构映射关系如下图所示：



主要改造点就是业务逻辑层的 Service，根据聚合拆分到应用层的应用服务与领域层的领域服务，部分业务逻辑还会以充血模型下沉到 Entity 中。

接着就是数据访问层的改造，根据依赖倒置原则，**数据库的访问接口会被放到领域层中（因为属于行为）**，具体的访问实现则是在基础设施层内（为行为提供支持）。除此之外，第三方工具、Common、Config 等都放在基础设施层中。

DDD 代码架构

首先明确一点，**DDD 代码架构并没有统一的标准**，不同公司的架构都是不一样的！但是核心的思想都是大差不差的，仅一些细节有调整。

按照四层架构，我们可以建立 interfaces（用户接口层）、application（应用层）、domain（领域层）、infrastructure（基础设施层） 这 4 个包。

interface 是 Java 关键字，因此包名加了个 s。

1、interface

该层主要负责与外部系统交互，包括用户界面（UI）、API 接口、请求的接收和响应的返回等。它作为领域层与外部世界的接口，确保领域逻辑的解耦。

存放的代码：

- 控制器（Controller）：处理 HTTP 请求，负责路由和请求的转发。
- REST API 接口：定义暴露给外部系统的服务接口。
- 请求和响应对象：用于与外部系统交换数据。

2、application

该层负责协调多个领域对象的操作，完成应用级的任务。它充当领域层与用户接口层之间的桥梁，调用领域层中的业务逻辑，并将结果返回给用户接口层。应用层的职责是实现具体用例，而不包含业务规则。

存放的代码：

- 应用服务（Application Service）：负责组织和协调领域对象，处理跨多个聚合的操作，通常表示应用中的具体功能，如 “下订单” 或“注册用户”。

3、domain

该层包含核心业务逻辑，它是系统的核心部分，负责模型的定义和业务规则的实现。领域层中的模型代表着业务概念，

通常会包括聚合、实体和值对象。这个层不依赖于任何外部技术或框架，它专注于业务本身。

存放的代码：

- 聚合：一个聚合由多个实体和值对象构成，它们之间有着一致的业务规则，**一般包名就代表一个聚合**。
- 实体：具有唯一标识符（ID）的对象。
- 值对象：没有身份标识且是不可变的对象，通常用于表示某个概念的属性。
- 领域服务：当某个业务逻辑无法归属到某个实体或聚合时，使用领域服务来封装这些业务逻辑。
- 领域事件：表示领域中发生的某个重要事件，如 “订单已支付”。
- 仓储接口：定义资源访问的接口
- 持久化对象：PO（数据库查询逻辑不复杂时，可以省略）

4、infrastructure

该层提供技术支持，是所有其他层的基础设施。它包含数据库操作、消息队列、缓存、文件存储等第三方依赖。基础设施层实现了与外部系统的交互，但不包含业务逻辑。

存放的代码：

- 持久化：如使用 JPA 或 MyBatis 等技术实现数据库的访问。
- 外部系统集成：与外部服务或系统的通信，如调用文件存储。
- 工具类和基础设施组件：提供诸如日志、定时任务、邮件发送等功能。

项目目录结构示例

main/java 包下：

- application（应用层）
- domain（领域层）
- order（订单聚合）
- entity（实体）
- valueObject（值对象）
- event（事件）
- repository（仓储）
- service（领域服务）
- user（用户聚合）
- infrastructure（基础设施层）
- api（外部接口）
- config（配置）
- mq（消息队列）
- repository（仓储实现）
- facade（仓储接口）
- po（持久化对象）
- util（工具类）
- interfaces（用户接口层）
- assembler（对象转化类）
- dto（传输对象）
- controller（提供给用户界面、外部服务的接口）
- shared（共享模块）
- Application 项目主类（或启动类）

此外，实现 DDD 的过程中，还可能会用到工厂和仓储模式。

- 工厂：用于创建聚合和实体，因为聚合根与聚合内的实体、值对象关系比较复杂，为了确保对象创建的一致性和完整性会使用工厂模式来创建领域对象（通常从数据库获取 PO 持久化对象后，通过工厂模式创建 DO 领域对象）。

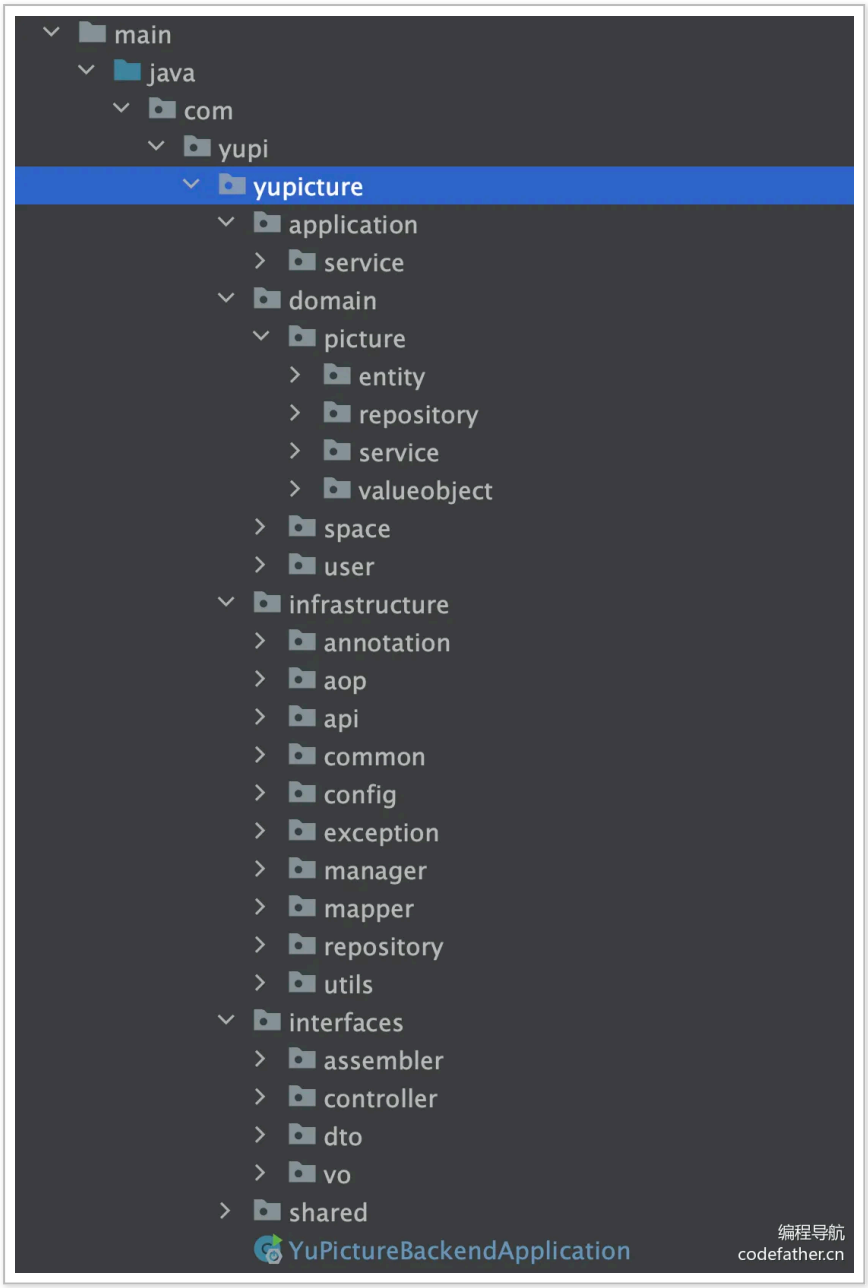
- 仓储：用于持久化领域对象（如实体和聚合），它封装了数据库操作，使得业务逻辑与数据存储分离。

四、项目 DDD 重构

下面我们要将项目重构为 DDD 模式，这个过程不仅涉及到目录结构的改造，还涉及到大量方法的重构、代码的改造等。

在开始之前明确一点：**DDD 项目的改造没有一个绝对的标准！** 一定要根据实际项目的需求和复杂度综合评估改造的逻辑。

来看下改造后的项目包结构，有个印象即可，下面带大家依次实战：



改造方案

1、领域划分

首先，从系统的功能点出发，并且考虑到要利于拆分，将系统划分为以下 3 个领域：

1. 用户领域（User Domain），用户注册、登录、获取个人信息等等用户相关功能放在这个领域中。

2. 图片领域（Picture Domain），包括图片上传、删除、编辑、URL 上传、批量管理等功能。

3. 空间领域（Space Domain），包括空间创建、空间管理、空间分析、空间成员管理等。

2、改造方案

一般项目的重构都要有序进行，所以我们要先 **浅层改造**，也就是将原有代码移动到不同的目录中，但是尽量不改变代码内容本身。有些博主就是这么做的，其实是一种省事儿的方法，不能说这样改造就错了，但效果就是 “项目看起来像是 DDD 架构设计”，实际上缺少灵魂。

所以在划分目录后，还要 **深层改造**，比如将原有的 Service 层服务进行拆分，将对象转换类代码移动到 interfaces 层的 assembler 中、将简单的业务逻辑移动到 domain.entity 实体类中、将跨领域调用的方法移动到 application.service 应用服务中等等。

大家思考一下，如果让你来改造 DDD 项目，你具体会怎么执行呢？

是先把 DDD 目录结构建好，分为 4 个层，然后依次一层一层地完成 infrastructure、domain、application、interface 层的代码么？

这其实是传统的正向思维，按照目标的目录结构来重构。但是这样重构可能会出现一个问题，比如我在开发 domain 层的时候，有些 service 的方法可能要移动到 application 层或者 interface 层，这就会导致我们开发时经常要在各层的目录中进行跳转，增加了复杂度。

所以这里鱼皮结合自己的经验，给大家分享一种又快速、又轻松、又规范的改造方法。让我们使用 **逆向思维**，还原我们最初从 0 开发本项目的流程，**根据现有代码进行拆分，而不是按照特定的分层一层一层拆**。

举个例子，拆分原项目 model 包的时候，可以把 entity 放到 domain 层中，把 dto 和 vo 放到 interface 层中。

这样不仅思路清晰，不容易遗漏代码，而且按照 model => mapper => service => controller 的顺序拆分，每一层都不会缺少对下一层的依赖，不会出现类不存在的情况，能够大幅提高效率。

此外，建议大家一个领域一个领域地重构，而不是一次性把多个领域的代码同时改造，这样出了问题就不好还原了。

💡 DDD 重构的思路都是一致的，完整重构整个项目至少需要好几个小时，性价比不高，大家只需要重点学习一个领域的重构即可。

下面我们进入项目重构。

新建项目

首先基于原有的项目复制一个新的项目，然后新建一个根包，而不是改造原有的包。接下来我们可以持续将原有包的代码移动到新包中，从而提高重构效率。

需要先将 Spring Boot 的启动类移动到新包中，后续才能启动项目。

基础设施层

infrastructure 层是存放基础设施的代码，也就是通用的代码，所以要优先重构，步骤如下：

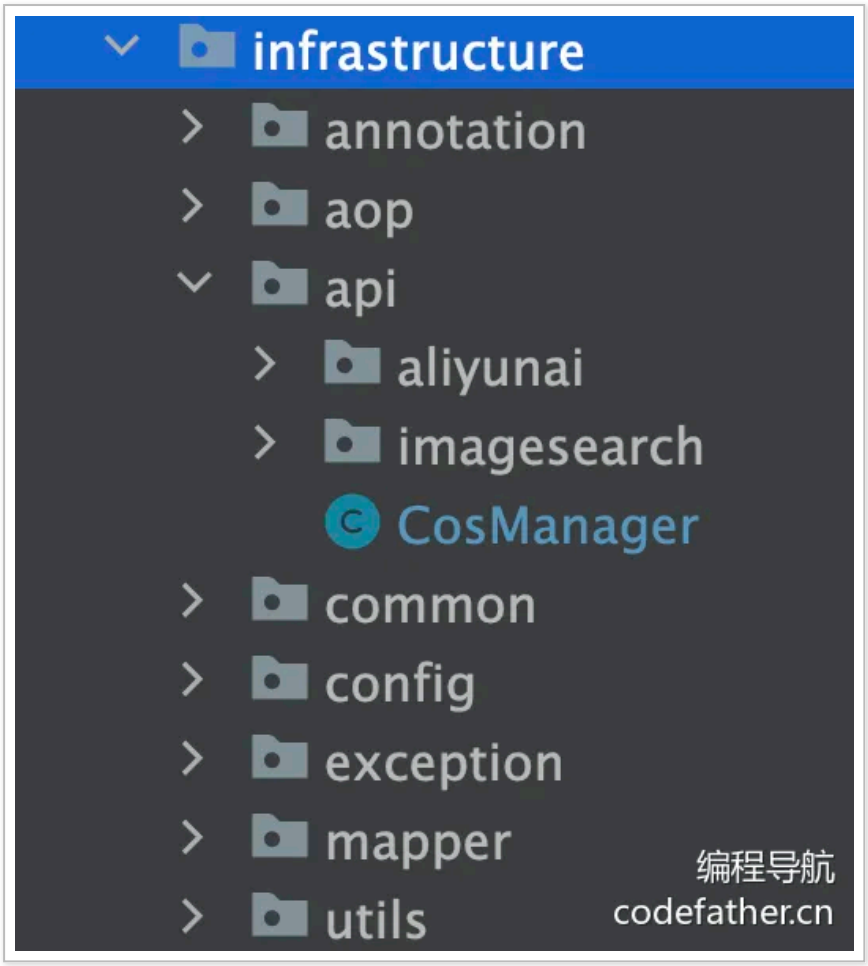
1）移动通用代码：先把 annotation、aop、common、config、exception 包放到 `infrastructure` 包下

2）移动数据访问层 mapper 包。**注意，要同步修改 MyBatisPlusConfig 扫描 mapper 的包名！**

```
@MapperScan("com.yupi.yupicture.infrastructure.mapper")
public class MyBatisPlusConfig {}
```

3）将 CosManager 移动到 api 包中，因为该类主要是负责调用第三方对象存储 API，和业务无关（可以改名为 CosApi）。

这样原包的最外层就只有 constant、model、service、controller、manager 包了，重构后的 `infrastructure` 包结构如图：



💡 为什么 Mapper 应该放在 infrastructure 层?

1. 职责划分：domain 层是业务逻辑的核心，应该专注于领域模型和业务规则，避免引入任何技术实现的细节。infrastructure 层是用来实现技术细节的，包括数据库访问、第三方服务集成、缓存实现等。而 MyBatis Plus 的 Mapper 类就是一种数据库访问的实现细节。

2. 与 DDD 的设计原则保持一致：在 DDD 中，domain 层的职责是独立于技术实现的，不能直接依赖具体的框架或持久化技术。将 Mapper 放在 infrastructure 层，可以避免技术细节 “污染” 领域层，保持领域模型的纯粹性。

用户领域

下面我们先拆分项目的核心模块 —— 用户领域，这个领域我会拆分地相对细一些，带大家学习标准的 DDD 重构方法。学会这一个领域之后，其他的领域重构就很简单了。

1、重构 model 包

按照下面的规则，将原始 model 包中的代码移动到对应的新位置：

原始包	重构后的包	备注
model.entity	domain.user.entity	User 类
model.enums	domain.user.valueobject	UserRoleEnum 枚举类
model.dto.user	interfaces.dto.user	请求封装类
model.vo	interfaces.vo.user	响应封装类 LoginUserVO、UserVO

2、重构 constant 包

原始包	重构后的包	备注
constant	domain.user.constant	UserConstant 类

3、重构数据访问层

根据前面讲过的依赖倒置原则，在领域包下新建 `repository` 包，定义与数据库交互的接口，然后在 `infrastructure.repository` 中写相应的实现。

由于我们的项目中使用了 MyBatis Plus 框架，可以让接口直接继承其提供的 IService 接口，接口的实现继承 ServiceImpl 类，这样就直接拥有了一批操作数据库的方法，简化开发。

新增 UserRepository 接口：

```
package com.yupi.yupicture.domain.user.repository;

public interface UserRepository extends IService<User> {
}
```

新增 UserRepositoryImpl 实现类：

```
package com.yupi.yupicture.infrastructure.repository;
```

```
@Service
public class UserRepositoryImpl extends ServiceImpl<UserMapper, User>
}
```

UserMapper 之前已经移动到了 infrastructure 包中，作为实现中的一部分。

4、重构 Service

Service 层的重构是相对最麻烦的，但我们可以利用一些小技巧大幅提高重构效率。

1）首先，直接在 IDE 中移动 Service 接口和实现类到应用服务层。

原始类	重构后的类
service.UserService	application.service.UserApplicationService
service.impl.UserServiceImpl	application.service.impl.UserApplicationServi

为什么要这么做呢？因为应用服务层是可供其他领域调用的，而之前的 Service 也是可供其他 Service 调用的。直接移动后，IDE 会 自动重构代码，将对原始服务接口的调用改为新应用服务接口的调用，减少了手动修改的代码量。

2）复制 Service 接口和实现类为领域服务层：

原始类	重构后的类
service.UserService	domain.user.service.UserDomainService
service.impl.UserServiceImpl	domain.user.service.impl.UserDomainService

为什么要这么做呢？因为领域服务层是编写核心业务逻辑的位置，也需要被应用服务层调用，所以先把原来的 Service 接口和实现类复制过来，便于等会儿按需保留代码或拆分代码。

3）重构应用服务层

application 层主要做领域服务的编排，事务一般也交由 application 层来控制。

应用服务层遵循的原则：

- 将业务逻辑下沉到 领域服务或实体类 中，应用服务层需要调用领域服务或实体类来完成业务逻辑。
- 如果某个方法需要调用其他应用服务（在单个领域内无法完成），那么该方法不能放到领域服务中，而是保留在应用服务中，因为原则上领域服务不应该调用应用服务。
- 负责为接口层提供调用支持，因为原则上接口层只能调用应用服务层。

比如用户注册方法，包含了校验和执行注册两部分业务逻辑。校验逻辑不涉及调用数据库，是对实体本身的校验，所以可以下沉到 User 实体中；执行注册需要操作数据库，可以下沉到领域服务 UserDomainService 中。而应用服务层要做的就是组合这些调用，并且 **增加事务** 等特性，得到完整的应用服务方法。用户登录方法同理。

给 User 实体补充方法：

```
public static void validUserRegister(String userAccount, String userP

    if (StrUtil.hasBlank(userAccount, userPassword, checkPassword)) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "参数为空"
    }
    if (userAccount.length() < 4) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "用户账号过
    }
    if (userPassword.length() < 8 || checkPassword.length() < 8) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "用户密码过
    }
    if (!userPassword.equals(checkPassword)) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "两次输入的
    }
}


public static void validUserLogin(String userAccount, String userPass
    if (StrUtil.hasBlank(userAccount, userPassword)) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "参数为空"
    }
    if (userAccount.length() < 4) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "账号错误"
    }
    if (userPassword.length() < 8) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "密码错误"
    }
}
```

应用服务层的代码如下，补充了很多 interfaces 层需要调用的方法（比如 getUserById）：

```
@Service
@Slf4j
public class UserApplicationServiceImpl implements UserApplicationSer

    @Resource
    private UserDomainService userDomainService;

    @Override
    @Transactional
    public long userRegister(UserRegisterRequest userRegisterRequest)
        ThrowUtils.throwIf(userRegisterRequest == null, ErrorCode.PAR
```

```
    String userAccount = userRegisterRequest.getUserAccount();
    String userPassword = userRegisterRequest.getUserPassword();
    String checkPassword = userRegisterRequest.getCheckPassword()

    User.validUserRegister(userAccount, userPassword, checkPasswo
    return userDomainService.userRegister(userAccount, userPasswo
}

@Override
public LoginUserVO userLogin(UserLoginRequest userLoginRequest, H
    ThrowUtils.throwIf(userLoginRequest == null, ErrorCode.PARAMS
    String userAccount = userLoginRequest.getUserAccount();
    String userPassword = userLoginRequest.getUserPassword();

    User.validUserLogin(userAccount, userPassword);
    return userDomainService.userLogin(userAccount, userPassword,
}


@Override
public User getLoginUser(HttpServletRequest request) {
    return userDomainService.getLoginUser(request);
}


@Override
public boolean userLogout(HttpServletRequest request) {
    ThrowUtils.throwIf(request == null, ErrorCode.PARAMS_ERROR);
    return userDomainService.userLogout(request);
}

@Override
public LoginUserVO getLoginUserVO(User user) {
    return userDomainService.getLoginUserVO(user);
}

@Override
public UserVO getUserVO(User user) {
    return userDomainService.getUserVO(user);
}

@Override
public List<UserVO> getUserVOList(List<User> userList) {
    return userDomainService.getUserVOList(userList);
}

@Override
public QueryWrapper<User> getQueryWrapper(UserQueryRequest userQu
    return userDomainService.getQueryWrapper(userQueryRequest);
}

@Override
public long addUser(User user) {
    return userDomainService.addUser(user);
}

@Override
public User getUserById(long id) {
    ThrowUtils.throwIf(id <= 0, ErrorCode.PARAMS_ERROR);
    User user = userDomainService.getById(id);
    ThrowUtils.throwIf(user == null, ErrorCode.NOT_FOUND_ERROR);
    return user;
}
```

```
    @Override
    public UserVO getUserVOById(long id) {
        return userDomainService.getUserVO(getUserById(id));
    }

    @Override
    public boolean deleteUser(DeleteRequest deleteRequest) {
        if (deleteRequest == null || deleteRequest.getId() <= 0) {
            throw new BusinessException(ErrorCode.PARAMS_ERROR);
        }
        return userDomainService.removeById(deleteRequest.getId());
    }

    @Override
    public void updateUser(User user) {
        boolean result = userDomainService.updateById(user);
        ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
    }

    @Override
    public Page<UserVO> listUserVOByPage(UserQueryRequest userQueryRe
        ThrowUtils.throwIf(userQueryRequest == null, ErrorCode.PARAMS
        long current = userQueryRequest.getCurrent();
        long size = userQueryRequest.getPageSize();
        Page<User> userPage = userDomainService.page(new Page<>(curre
                userDomainService.getQueryWrapper(userQueryRequest));
        Page<UserVO> userVOPage = new Page<>(current, size, userPage.
        List<UserVO> userVO = userDomainService.getUserVOList(userPag
        userVOPage.setRecords(userVO);
        return userVOPage;
    }

    @Override
    public List<User> listByIds(Set<Long> userIdSet) {
        return userDomainService.listByIds(userIdSet);
    }

    @Override
    public String getEncryptPassword(String userPassword) {
        return userDomainService.getEncryptPassword(userPassword);
    }
}
```

💡 小技巧：只要发现不调用其他应用服务的方法、并且不调用“当前类中依赖其他应用服务”的方法，就可以改为调用领域服务；否则该方法需要在应用服务中实现。

4）重构领域服务层

领域服务层遵循的原则：

- 需要调用数据库服务（repository）或基础设施层（infrastructure）来完成业务逻辑

- 可以根据需要，将和实体强相关的业务逻辑下沉到 实体类中

比如用户注册和用户登录方法，无需再包含校验逻辑（已经下沉到了 User 实体类中），只需要调用 UserRepository 执行数据库操作即可。

像 `isAdmin` 这样根据 User 对象进行判断的方法，可以下沉到 User 实体类中：

```
public boolean isAdmin() {
    return UserRoleEnum.ADMIN.getValue().equals(this.getUserRole());
}
```

领域服务层的代码如下，补充了很多应用服务层需要调用的方法（比如 getById）：

```
@Service
@Slf4j
public class UserDomainServiceImpl implements UserDomainService {

    @Resource
    private UserRepository userRepository;

    @Override
    public long userRegister(String userAccount, String userPassword,

        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("userAccount", userAccount);
        long count = userRepository.getBaseMapper().selectCount(query
        if (count > 0) {
            throw new BusinessException(ErrorCode.PARAMS_ERROR, "账号
        }

        String encryptPassword = getEncryptPassword(userPassword);

        User user = new User();
        user.setUserAccount(userAccount);
        user.setUserPassword(encryptPassword);
        user.setUserName("无名");
        user.setUserRole(UserRoleEnum.USER.getValue());
        boolean saveResult = userRepository.save(user);
        if (!saveResult) {
            throw new BusinessException(ErrorCode.SYSTEM_ERROR, "注册
        }
        return user.getId();
    }

    @Override
    public LoginUserVO userLogin(String userAccount, String userPassw

        String encryptPassword = getEncryptPassword(userPassword);
```

```
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.eq("userAccount", userAccount);
    queryWrapper.eq("userPassword", encryptPassword);
    User user = userRepository.getBaseMapper().selectOne(queryWra

    if (user == null) {
        log.info("user login failed, userAccount cannot match use
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "用户
    }

    request.getSession().setAttribute(USER_LOGIN_STATE, user);

    StpKit.SPACE.login(user.getId());
    StpKit.SPACE.getSession().set(USER_LOGIN_STATE, user);
    return this.getLoginUserVO(user);
}


@Override
public User getLoginUser(HttpServletRequest request) {

    Object userObj = request.getSession().getAttribute(USER_LOGIN
    User currentUser = (User) userObj;
    if (currentUser == null || currentUser.getId() == null) {
        throw new BusinessException(ErrorCode.NOT_LOGIN_ERROR);
    }

    long userId = currentUser.getId();
    currentUser = userRepository.getById(userId);
    if (currentUser == null) {
        throw new BusinessException(ErrorCode.NOT_LOGIN_ERROR);
    }
    return currentUser;
}


@Override
public boolean userLogout(HttpServletRequest request) {

    Object userObj = request.getSession().getAttribute(USER_LOGIN
    if (userObj == null) {
        throw new BusinessException(ErrorCode.OPERATION_ERROR, "未
    }

    request.getSession().removeAttribute(USER_LOGIN_STATE);
    StpKit.SPACE.logout();
    return true;
}

@Override
public LoginUserVO getLoginUserVO(User user) {
    if (user == null) {
        return null;
    }
    LoginUserVO loginUserVO = new LoginUserVO();
    BeanUtils.copyProperties(user, loginUserVO);
    return loginUserVO;
}

@Override
public UserVO getUserVO(User user) {
    if (user == null) {
```

```
        return null;
    }
    UserVO userVO = new UserVO();
    BeanUtils.copyProperties(user, userVO);
    return userVO;
}

@Override
public List<UserVO> getUserVOList(List<User> userList) {
    if (CollUtil.isEmpty(userList)) {
        return new ArrayList<>();
    }
    return userList.stream().map(this::getUserVO).collect(Collect
}

@Override
public QueryWrapper<User> getQueryWrapper(UserQueryRequest userQu
    if (userQueryRequest == null) {
        throw new BusinessException(ErrorCode.PARAMS_ERROR, "请求
    }
    Long id = userQueryRequest.getId();
    String userAccount = userQueryRequest.getUserAccount();
    String userName = userQueryRequest.getUserName();
    String userProfile = userQueryRequest.getUserProfile();
    String userRole = userQueryRequest.getUserRole();
    String sortField = userQueryRequest.getSortField();
    String sortOrder = userQueryRequest.getSortOrder();
    QueryWrapper<User> queryWrapper = new QueryWrapper<>();
    queryWrapper.eq(ObjUtil.isNotNull(id), "id", id);
    queryWrapper.eq(StrUtil.isNotBlank(userRole), "userRole", use
    queryWrapper.like(StrUtil.isNotBlank(userAccount), "userAccou
    queryWrapper.like(StrUtil.isNotBlank(userName), "userName", u
    queryWrapper.like(StrUtil.isNotBlank(userProfile), "userProfi
    queryWrapper.orderBy(StrUtil.isNotEmpty(sortField), sortOrder
    return queryWrapper;
}

@Override
public String getEncryptPassword(String userPassword) {

    final String SALT = "yupi";
    return DigestUtils.md5DigestAsHex((SALT + userPassword).getBy
}

@Override
public Long addUser(User user) {

    final String DEFAULT_PASSWORD = "12345678";
    String encryptPassword = this.getEncryptPassword(DEFAULT_PASS
    user.setUserPassword(encryptPassword);
    boolean result = userRepository.save(user);
    ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
    return user.getId();
}

@Override
public Boolean removeById(Long id) {
    return userRepository.removeById(id);
}

@Override
public boolean updateById(User user) {
    return userRepository.updateById(user);
```

```
    }

    @Override
    public User getById(long id) {
        return userRepository.getById(id);
    }

    @Override
    public Page<User> page(Page<User> userPage, QueryWrapper<User> qu
        return userRepository.page(userPage, queryWrapper);
    }

    @Override
    public List<User> listByIds(Set<Long> userIdSet) {
        return userRepository.listByIds(userIdSet);
    }
}
```

💡 小技巧

1. 修改领域服务时，如果发现某个方法没被 application 调用（IDE 显示灰色），就可以直接移除掉。
2. 如果想节省重复编写增删改查等样板代码的时间，应用服务或领域服务也可以直接继承 MyBatis Plus 的接口和实现类，这样虽然 DDD 目录结构不是 100% 标准，但是能大幅减少开发成本。

5、重构 Controller

1）首先将原始 UserController 移动为 `interfaces.controller.UserController` 类。

2）为保证接口层的精简，需要将其中的代码下沉到 **转换类和应用服务** 中。首先编写转换类 `interfaces.assembler.UserAssembler` ，负责将 DTO 转为实体类：

```
public class UserAssembler {

    public static User toUserEntity(UserAddRequest request) {
        User user = new User();
        BeanUtils.copyProperties(request, user);
        return user;
    }

    public static User toUserEntity(UserUpdateRequest request) {
        User user = new User();
        BeanUtils.copyProperties(request, user);
        return user;
    }
```

```
}
```

3）将 Controller 的代码下沉到应用服务中，调用应用服务和 Assembler 来处理请求。可能会涉及到应用服务方法的参数修改，代码如下：

```
@RestController
@RequestMapping("/user")
public class UserController {

    @Resource
    private UserApplicationService userApplicationService;




    @PostMapping("/register")
    public BaseResponse<Long> userRegister(@RequestBody UserRegisterR
        long result = userApplicationService.userRegister(userRegiste
        return ResultUtils.success(result);
    }


    @PostMapping("/login")
    public BaseResponse<LoginUserVO> userLogin(@RequestBody UserLogin
        LoginUserVO loginUserVO = userApplicationService.userLogin(us
        return ResultUtils.success(loginUserVO);
    }


    @PostMapping("/logout")
    public BaseResponse<Boolean> userLogout(HttpServletRequest reques
        boolean result = userApplicationService.userLogout(request);
        return ResultUtils.success(result);
    }


    @GetMapping("/get/login")
    public BaseResponse<LoginUserVO> getLoginUser(HttpServletRequest
        User user = userApplicationService.getLoginUser(request);
        return ResultUtils.success(userApplicationService.getLoginUse
    }







    @PostMapping("/add")
    @AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
    public BaseResponse<Long> addUser(@RequestBody UserAddRequest use
        ThrowUtils.throwIf(userAddRequest == null, ErrorCode.PARAMS_E
        User userEntity = UserAssembler.toUserEntity(userAddRequest);
        return ResultUtils.success(userApplicationService.addUser(use
    }


    @GetMapping("/get")
```

```
    @AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
    public BaseResponse<User> getUserById(long id) {
        User user = userApplicationService.getUserById(id);
        return ResultUtils.success(user);
    }


    @GetMapping("/get/vo")
    public BaseResponse<UserVO> getUserVOById(long id) {
        return ResultUtils.success(userApplicationService.getUserVOBy
    }


    @PostMapping("/delete")
    @AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
    public BaseResponse<Boolean> deleteUser(@RequestBody DeleteReques
        boolean b = userApplicationService.deleteUser(deleteRequest);
        return ResultUtils.success(b);
    }


    @PostMapping("/update")
    @AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
    public BaseResponse<Boolean> updateUser(@RequestBody UserUpdateRe
        ThrowUtils.throwIf(userUpdateRequest == null, ErrorCode.PARAM
        User userEntity = UserAssembler.toUserEntity(userUpdateReques
        userApplicationService.updateUser(userEntity);
        return ResultUtils.success(true);
    }


    @PostMapping("/list/page/vo")
    @AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
    public BaseResponse<Page<UserVO>> listUserVOByPage(@RequestBody U
        Page<UserVO> userVOPage = userApplicationService.listUserVOBy
        return ResultUtils.success(userVOPage);
    }


}
```

这样一来，接口的代码保持了极致的精简。

💡 前面也提到了，如果觉得一层一层补充调用方法过于麻烦，可以直接给应用服务或领域服务继承 MyBatis Plus 的 IService 和 ServiceImpl，便于上一层调用。

6、其他代码兼容

尝试启动项目，应该会出现编译错误，我们根据报错提示依次解决即可。比如修改下面几个问题：

1）修改 isAdmin 的调用，改为调用对象的方法：

```
原始：userApplicationService.isAdmin(loginUser)
改为：loginUser.isAdmin()
```

2）给用户应用服务 UserApplicationService 补充其他应用服务需要的方法，比如 listByIds。

除非考虑到开发时间成本的问题，否则其他应用服务尽量调用应用服务层的方法，而不是领域服务层。

最终，尝试启动项目，只要不报编译错误，就算是重构完成了，即使项目启动不起来也不用在意，因为我们有些服务还没有重构完。

图片领域

通过用户领域，相信大家已经学会领域的拆分方法了，接下来图片领域和空间领域就不带大家拆分得那么细节了，我们简单将项目进行重构即可。

1、重构 model 包

按照下面的规则，将原始 model 包中的代码移动到对应的新位置：

原始包	重构后的包	备注
model.entity	domain.picture.entity	Picture 类
model.enums	domain.picture.valueobject	PictureReviewStatusEnu 枚举类
model.dto.picture	interfaces.dto.picture	请求封装类
model.vo	interfaces.vo.picture	响应封装类 PictureVO、PictureTagCategory

2、重构数据访问层

根据前面讲过的依赖倒置原则，在领域包下新建 `repository` 包，定义与数据库交互的接口，然后在 `infrastructure.repository` 中写相应的实现。

由于我们的项目中使用了 MyBatis Plus 框架，可以让接口直接继承其提供的 IService 接口，接口的实现继承 ServiceImpl 类，这样就直接拥有了一批操作数据库的方法，简化开发。

新增 PictureRepository 接口：

```
package com.yupi.yupicture.domain.picture.repository;

public interface PictureRepository extends IService<Picture> {
}
```

新增 PictureRepositoryImpl 实现类：

```
package com.yupi.yupicture.infrastructure.repository;

@Service
public class PictureRepositoryImpl extends ServiceImpl<PictureMapper,
}
```

PictureMapper 之前已经移动到了 infrastructure 包中，作为实现中的一部分。

3、重构 Service

Service 层的重构是相对最麻烦的，但我们可以利用一些小技巧大幅提高重构效率。

1）首先，直接在 IDE 中移动 Service 接口和实现类到应用服务层。

原始类	重构后的类
service.PictureService	application.service.PictureApplicationServi

原始类	重构后的类
service.impl.PictureServiceImpl	application.service.impl.PictureApplication

为什么要这么做呢？因为应用服务层是可供其他领域调用的，而之前的 Service 也是可供其他 Service 调用的。直接移动后，IDE 会 自动重构代码，将对原始服务接口的调用改为新应用服务接口的调用，减少了手动修改的代码量。

2）复制 Service 接口和实现类为领域服务层：

原始类	重构后的类
service.PictureService	domain.user.service.PictureDomainService
service.impl.PictureServiceImpl	domain.user.service.impl.PictureDomainSe

为什么要这么做呢？因为领域服务层是编写核心业务逻辑的位置，也需要被应用服务层调用，所以先把原来的 Service 接

口和实现类复制过来，便于等会儿按需保留代码或拆分代码。

3）重构应用服务层

application 层主要做领域服务的编排，如果，事务一般也交由 application 层来控制。

应用服务层遵循的原则：

- 将业务逻辑下沉到 领域服务或实体类 中，应用服务层需要调用领域服务或实体类来完成业务逻辑。
- 如果某个方法需要调用其他应用服务（在单个领域内无法完成），那么该方法不能放到领域服务中，而是保留在应用服务中，因为原则上领域服务不应该调用应用服务。
- 负责为接口层提供调用支持，因为原则上接口层只能调用应用服务层。

遵循原则，将 getPictureVO、getPictureVOPage 方法的实现保留在 PictureApplicationServiceImpl 中，因为它们都调用了其他应用服务 userApplicationService。其他方法可以下沉到领域服务中，应用服务层的代码如下：

```
@Service
@Slf4j
public class PictureApplicationServiceImpl extends ServiceImpl<Pictur

    @Resource
    private PictureDomainService pictureDomainService;

    @Resource
    private UserApplicationService userApplicationService;

    @Override
    public PictureVO uploadPicture(Object inputSource, PictureUploadR
        return pictureDomainService.uploadPicture(inputSource, pictur
    }

    @Override
    public void validPicture(Picture picture) {
        pictureDomainService.validPicture(picture);
    }

    @Override
    public QueryWrapper<Picture> getQueryWrapper(PictureQueryRequest
        return pictureDomainService.getQueryWrapper(pictureQueryReque
    }
```

```
@Override
public PictureVO getPictureVO(Picture picture, HttpServletRequest

    PictureVO pictureVO = PictureVO.objToVo(picture);

    Long userId = picture.getUserId();
    if (userId != null && userId > 0) {
        User user = userApplicationService.getUserById(userId);
        UserVO userVO = userApplicationService.getUserVO(user);
        pictureVO.setUser(userVO);
    }
    return pictureVO;
}


@Override
public Page<PictureVO> getPictureVOPage(Page<Picture> picturePage
    List<Picture> pictureList = picturePage.getRecords();
    Page<PictureVO> pictureVOPage = new Page<>(picturePage.getCur
    if (CollUtil.isEmpty(pictureList)) {
        return pictureVOPage;
    }

    List<PictureVO> pictureVOList = pictureList.stream().map(Pict

    Set<Long> userIdSet = pictureList.stream().map(Picture::getUs
    Map<Long, List<User>> userIdUserListMap = userApplicationServ
            .collect(Collectors.groupingBy(User::getId));

    pictureVOList.forEach(pictureVO -> {
        Long userId = pictureVO.getUserId();
        User user = null;
        if (userIdUserListMap.containsKey(userId)) {
            user = userIdUserListMap.get(userId).get(0);
        }
        pictureVO.setUser(userApplicationService.getUserVO(user))
    });
    pictureVOPage.setRecords(pictureVOList);
    return pictureVOPage;
}

@Override
public void doPictureReview(PictureReviewRequest pictureReviewReq
    pictureDomainService.doPictureReview(pictureReviewRequest, lo
}

@Override
public void fillReviewParams(Picture picture, User loginUser) {
    pictureDomainService.fillReviewParams(picture, loginUser);
}

@Override
public int uploadPictureByBatch(PictureUploadByBatchRequest pictu
    return pictureDomainService.uploadPictureByBatch(pictureUploa
}

@Override
public void clearPictureFile(Picture oldPicture) {
    pictureDomainService.clearPictureFile(oldPicture);
}

@Override
```

```
    public void deletePicture(long pictureId, User loginUser) {
        pictureDomainService.deletePicture(pictureId, loginUser);
    }

    @Override
    public void checkPictureAuth(User loginUser, Picture picture) {
        pictureDomainService.checkPictureAuth(loginUser, picture);
    }

    @Override
    public void editPicture(Picture picture, User loginUser) {
        pictureDomainService.editPicture(picture, loginUser);
    }

    @Override
    public List<PictureVO> searchPictureByColor(Long spaceId, String
        return pictureDomainService.searchPictureByColor(spaceId, pic
    }

    @Override
    public void editPictureByBatch(PictureEditByBatchRequest pictureE
        pictureDomainService.editPictureByBatch(pictureEditByBatchReq
    }

    @Override
    public CreateOutPaintingTaskResponse createPictureOutPaintingTask
        return pictureDomainService.createPictureOutPaintingTask(crea
    }
}
```

由于 interfaces 层要调用应用服务层来实现功能，为了方便，可以直接让图片应用服务继承 MyBatis Plus 的接口和实现类，减少样板增删改查方法的编写（比如 getById）。

💡 小技巧：只要发现不调用其他应用服务的方法、并且不调用“当前类中依赖其他应用服务”的方法，就可以改为调用领域服务；否则该方法需要在应用服务中实现。

4）重构领域服务层

领域服务层遵循的原则：

- 需要调用数据库服务（repository）或基础设施层（infrastructure）来完成业务逻辑
- 可以根据需要，将和实体强相关的业务逻辑下沉到 实体类 中

遵循原则编写领域服务层的代码，由于代码量较大，下面只列举关键修改：

```
@Service
@Slf4j
public class PictureDomainServiceImpl
        implements PictureDomainService {

    @Resource
    private PictureRepository pictureRepository;


    @Override
    public PictureVO uploadPicture(Object inputSource, PictureUploadR


        if (pictureId != null) {
            Picture oldPicture = pictureRepository.getById(pictureId)
            ThrowUtils.throwIf(oldPicture == null, ErrorCode.NOT_FOUN
        }
        transactionTemplate.execute(status -> {
            boolean result = pictureRepository.saveOrUpdate(picture);
            ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR, "
        });
    }


    @Override
    public void validPicture(Picture picture) {
    }


    @Override
    public QueryWrapper<Picture> getQueryWrapper(PictureQueryRequest
    }


    @Override
    public void doPictureReview(PictureReviewRequest pictureReviewReq

        Picture oldPicture = pictureRepository.getById(id);
        boolean result = pictureRepository.updateById(updatePicture);
        ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
    }

    @Override
    public void fillReviewParams(Picture picture, User loginUser) {
    }

    @Override
    public int uploadPictureByBatch(PictureUploadByBatchRequest pictu
    }

    @Async
    @Override
    public void clearPictureFile(Picture oldPicture) {
        long count = pictureRepository.lambdaQuery()
                .eq(Picture::getUrl, pictureUrl)
                .count();
    }

    @Override
    public void deletePicture(long pictureId, User loginUser) {

        Picture oldPicture = pictureRepository.getById(pictureId);
```

```
        ThrowUtils.throwIf(oldPicture == null, ErrorCode.NOT_FOUND_ER

        transactionTemplate.execute(status -> {

            boolean result = pictureRepository.removeById(pictureId);
            return true;
        });
    }


    @Override
    public void checkPictureAuth(User loginUser, Picture picture) {
    }

    @Override
    public void editPicture(Picture picture, User loginUser) {
        Picture oldPicture = pictureRepository.getById(id);

        boolean result = pictureRepository.updateById(picture);
        ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
    }

    @Override
    public List<PictureVO> searchPictureByColor(Long spaceId, String

        List<Picture> pictureList = pictureRepository.lambdaQuery()
                .eq(Picture::getSpaceId, spaceId)
                .isNotNull(Picture::getPicColor)
                .list();
    }



    @Override
    @Transactional(rollbackFor = Exception.class)
    public void editPictureByBatch(PictureEditByBatchRequest pictureE

        List<Picture> pictureList = pictureRepository.lambdaQuery()
                .select(Picture::getId, Picture::getSpaceId)
                .eq(Picture::getSpaceId, spaceId)
                .in(Picture::getId, pictureIdList)
                .list();
    }


    private void fillPictureWithNameRule(List<Picture> pictureList, S
    }

    @Override
    public CreateOutPaintingTaskResponse createPictureOutPaintingTask
        Picture picture = Optional.ofNullable(pictureRepository.getBy
            .orElseThrow(() -> new BusinessException(ErrorCode.NOT_FO
    }
}
```

注意，其实 validPicture 方法是可以移动到 Picture 实体类中的，大家可自行操作。

💡 小技巧

1. 修改领域服务时，如果发现某个方法没被 application 调用（IDE 显示灰色），就可以直接移除掉。

2. 如果想节省重复编写增删改查等样板代码的时间，应用服务或领域服务也可以直接继承 MyBatis Plus 的接口和实现类，这样虽然 DDD 目录结构不是 100% 标准，但是能大幅减少开发成本。

4、重构 Controller

1）首先将原始 PictureController 移动为 `interfaces.controller.PictureController` 类。

2）为保证接口层的精简，需要将其中的代码下沉到 **转换类和应用服务** 中。首先编写转换类 `interfaces.assembler.PictureAssembler` ，负责将 DTO 转为实体类：

```
public class PictureAssembler {

    public static Picture toPictureEntity(PictureEditRequest request)
        Picture picture = new Picture();
        BeanUtils.copyProperties(request, picture);

        picture.setTags(JSONUtil.toJsonStr(request.getTags()));
        return picture;
    }

    public static Picture toPictureEntity(PictureUpdateRequest reques
        Picture picture = new Picture();
        BeanUtils.copyProperties(request, picture);

        picture.setTags(JSONUtil.toJsonStr(request.getTags()));
        return picture;
    }
}
```

3）将 Controller 的代码下沉到应用服务中，调用应用服务和 Assembler 来处理请求，可能会涉及到应用服务方法的参数修改。其中 updatePicture、editPicture 是改造的重点，需要调用 Assembler 和应用服务层完成功能，下面只列举修改的关键代码：

```
@RestController
@RequestMapping("/picture")
@Slf4j
public class PictureController {

    @Resource
    private PictureApplicationService pictureApplicationService;

    @Resource
    private UserApplicationService userApplicationService;




    @PostMapping("/upload")
    @SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTU
    public BaseResponse<PictureVO> uploadPicture(@RequestPart("file")
        User loginUser = userApplicationService.getLoginUser(request)
        PictureVO pictureVO = pictureApplicationService.uploadPicture
        return ResultUtils.success(pictureVO);
    }


    @PostMapping("/upload/url")
    @SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTU
    public BaseResponse<PictureVO> uploadPictureByUrl(@RequestBody Pi
        User loginUser = userApplicationService.getLoginUser(request)
        String fileUrl = pictureUploadRequest.getFileUrl();
        PictureVO pictureVO = pictureApplicationService.uploadPicture
        return ResultUtils.success(pictureVO);
    }


    @PostMapping("/delete")
    @SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTU
    public BaseResponse<Boolean> deletePicture(@RequestBody DeleteReq
        User loginUser = userApplicationService.getLoginUser(request)
        pictureApplicationService.deletePicture(deleteRequest.getId()
        return ResultUtils.success(true);
    }


    @PostMapping("/update")
    @AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
    public BaseResponse<Boolean> updatePicture(@RequestBody PictureUp

        Picture picture = PictureAssembler.toPictureEntity(pictureUpd

        pictureApplicationService.validPicture(picture);

        long id = pictureUpdateRequest.getId();
        Picture oldPicture = pictureApplicationService.getById(id);
        ThrowUtils.throwIf(oldPicture == null, ErrorCode.NOT_FOUND_ER

        User loginUser = userApplicationService.getLoginUser(request)
        pictureApplicationService.fillReviewParams(picture, loginUser

        boolean result = pictureApplicationService.updateById(picture
        ThrowUtils.throwIf(!result, ErrorCode.OPERATION_ERROR);
        return ResultUtils.success(true);
    }
```

```
@GetMapping("/get")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Picture> getPictureById(long id, HttpServletR

    Picture picture = pictureApplicationService.getById(id);
}


@GetMapping("/get/vo")
public BaseResponse<PictureVO> getPictureVOById(long id, HttpServ

    Picture picture = pictureApplicationService.getById(id);

    User loginUser = userApplicationService.getLoginUser(request)
    List<String> permissionList = spaceUserAuthManager.getPermiss
    PictureVO pictureVO = pictureApplicationService.getPictureVO(
    pictureVO.setPermissionList(permissionList);

    return ResultUtils.success(pictureVO);
}


@PostMapping("/list/page")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Page<Picture>> listPictureByPage(@RequestBody

    Page<Picture> picturePage = pictureApplicationService.page(ne
    return ResultUtils.success(picturePage);
}


@PostMapping("/list/page/vo")
public BaseResponse<Page<PictureVO>> listPictureVOByPage(@Request

    Page<Picture> picturePage = pictureApplicationService.page(ne

    return ResultUtils.success(pictureApplicationService.getPictu
}


@PostMapping("/edit")
@SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTU
public BaseResponse<Boolean> editPicture(@RequestBody PictureEdit
    User loginUser = userApplicationService.getLoginUser(request)

    Picture picture = PictureAssembler.toPictureEntity(pictureEdi
    pictureApplicationService.editPicture(picture, loginUser);
    return ResultUtils.success(true);
}




@GetMapping("/tag_category")
public BaseResponse<PictureTagCategory> listPictureTagCategory()
}


@PostMapping("/review")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Boolean> doPictureReview(@RequestBody Picture
    User loginUser = userApplicationService.getLoginUser(request)
```

```
        pictureApplicationService.doPictureReview(pictureReviewReques
        return ResultUtils.success(true);
    }


    @PostMapping("/upload/batch")
    @AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
    public BaseResponse<Integer> uploadPictureByBatch(@RequestBody Pi
        User loginUser = userApplicationService.getLoginUser(request)
        Integer uploadCount = pictureApplicationService.uploadPicture
        return ResultUtils.success(uploadCount);
    }


    @PostMapping("/search/picture")
    public BaseResponse<List<ImageSearchResult>> searchPictureByPictu
        Picture oldPicture = pictureApplicationService.getById(pictur
        ThrowUtils.throwIf(oldPicture == null, ErrorCode.NOT_FOUND_ER
        List<ImageSearchResult> resultList = ImageSearchApiFacade.sea
        return ResultUtils.success(resultList);
    }


    @PostMapping("/search/color")
    @SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTU
    public BaseResponse<List<PictureVO>> searchPictureByColor(@Reques
        User loginUser = userApplicationService.getLoginUser(request)
        List<PictureVO> result = pictureApplicationService.searchPict
        return ResultUtils.success(result);
    }


    @PostMapping("/edit/batch")
    @SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTU
    public BaseResponse<Boolean> editPictureByBatch(@RequestBody Pict
        User loginUser = userApplicationService.getLoginUser(request)
        pictureApplicationService.editPictureByBatch(pictureEditByBat
        return ResultUtils.success(true);
    }


    @PostMapping("/out_painting/create_task")
    @SaSpaceCheckPermission(value = SpaceUserPermissionConstant.PICTU
    public BaseResponse<CreateOutPaintingTaskResponse> createPictureO
            @RequestBody CreatePictureOutPaintingTaskRequest createPi
            HttpServletRequest request) {
        User loginUser = userApplicationService.getLoginUser(request)
        CreateOutPaintingTaskResponse response = pictureApplicationSe
        return ResultUtils.success(response);
    }


    @GetMapping("/out_painting/get_task")
    public BaseResponse<GetOutPaintingTaskResponse> getPictureOutPain
    }
}
```

这样一来，接口的代码更加精简。

💡 前面也提到了，如果觉得一层一层补充调用方法过于麻烦，可以直接给应用服务或领域服务继承 MyBatis Plus 的 IService 和 ServiceImpl，便于上一层调用。

5、其他代码兼容

尝试启动项目，可能会出现编译错误，我们根据报错提示依次解决即可。

最终，尝试启动项目，只要不报编译错误，就算是重构完成了，即使项目启动不起来也不用在意，因为我们有些服务还没有重构完。

空间领域

包括空间、空间分析、空间成员管理这 3 类核心功能，我们简单将项目进行重构即可。

1、重构 model 包

按照下面的规则，将原始 model 包中的代码移动到对应的新位置：

原始包	重构后的包	备注
model.entity	domain.space.entity	Space、SpaceUser 类
model.enums	domain.space.valueobject	SpaceLevelEnum、SpaceRoleEnum、SpaceTypeEnum 枚举类
model.dto.space	interfaces.dto.space	请求封装类
model.vo	interfaces.vo.space	响应封装类 SpaceVO、SpaceUserVO

2、重构数据访问层

根据前面讲过的依赖倒置原则，在领域包下新建 `repository` 包，定义与数据库交互的接口，然后在

`infrastructure.repository` 中写相应的实现。

由于我们的项目中使用了 MyBatis Plus 框架，可以让接口直接继承其提供的 IService 接口，接口的实现继承 ServiceImpl 类，这样就直接拥有了一批操作数据库的方法，简化开发。

新增 SpaceRepository 和 SpaceUserRepository 接口：

```
package com.yupi.yupicture.domain.space.repository;

public interface SpaceRepository extends IService<Space> {
}
```

```
package com.yupi.yupicture.domain.space.repository;

public interface SpaceUserRepository extends IService<SpaceUser> {
}
```

新增 SpaceRepositoryImpl 和 SpaceUserRepositoryImpl 实现类：

```
package com.yupi.yupicture.infrastructure.repository;

@Service
public class SpaceRepositoryImpl extends ServiceImpl<SpaceMapper, Spa
}
```

```
package com.yupi.yupicture.infrastructure.repository;

@Service
public class SpaceUserRepositoryImpl extends ServiceImpl<SpaceUserMap
}
```

SpaceMapper 和 SpaceUserMapper 之前已经移动到了 infrastructure 包中，作为实现中的一部分。

3、重构 Service

Service 层的重构是相对最麻烦的，但我们可以利用一些小技巧大幅提高重构效率。

1）首先，直接在 IDE 中移动 Service 接口和实现类到应用服务层，包括 3 个接口和实现类：

原始类	重构后的类
service.SpaceService	application.service.SpaceApplicatio
service.SpaceUserService	application.service.SpaceUserApplic
service.SpaceAnalyzeService	application.service.SpaceAnalyzeAp
service.impl.SpaceServiceImpl	application.service.impl.SpaceAppli
service.impl.SpaceUserServiceImpl	application.service.impl.SpaceUserA

原始类	重构后的类
service.impl.SpaceAnalyzeServiceImpl	application.service.impl.SpaceAnaly

为什么要这么做呢？因为应用服务层是可供其他领域调用的，而之前的 Service 也是可供其他 Service 调用的。直接移动后，IDE 会 自动重构代码，将对原始服务接口的调用改为新应用服务接口的调用，减少了手动修改的代码量。

2）复制 Service 接口和实现类为领域服务层，包括空间服务和空间成员服务。不需要 SpaceAnalayzeDomainService，因为实现分析功能依赖的是 Space 和 Picture 应用服务，而不是依赖 SpaceAnalayzeRepository（根本没有空间分析表）。

原始类	重构后的类
service.SpaceService	domain.user.service.SpaceDomainServ
service.SpaceUserService	domain.user.service.SpaceUserDomain

原始类	重构后的类
service.impl.SpaceServiceImpl	domain.user.service.impl.SpaceDomain
service.impl.SpaceUserServiceImpl	domain.user.service.impl.SpaceUserDo

为什么要这么做呢？因为领域服务层是编写核心业务逻辑的位置，也需要被应用服务层调用，所以先把原来的 Service 接口和实现类复制过来，便于等会儿按需保留代码或拆分代码。

3）重构应用服务层

application 层主要做领域服务的编排，如果，事务一般也交由 application 层来控制。

应用服务层遵循的原则：

- 将业务逻辑下沉到 领域服务或实体类 中，应用服务层需要调用领域服务或实体类来完成业务逻辑。
- 如果某个方法需要调用其他应用服务（在单个领域内无法完成），那么该方法不能放到领域服务中，而是保留在应用服务中，因为原则上领域服务不应该调用应用服务。
- 负责为接口层提供调用支持，因为原则上接口层只能调用应用服务层。

遵循原则，将 getSpaceUserVOList、getSpaceUserVO、validSpaceUser、addSpaceUser、getSpaceVOPage、getSpaceVO、addSpace 以及空间分析服务方法的实现保留在 ApplicationServiceImpl 中，因为它们都调用了其他应用服务（比如 userApplicationService）。其他方法可以下沉到领域服务中，以 SpaceApplicationService 为例，应用服务层的代码如下：

```
@Service
public class SpaceApplicationServiceImpl extends ServiceImpl<SpaceMap

    @Resource
    private SpaceDomainService spaceDomainService;

    @Resource
    private TransactionTemplate transactionTemplate;

    @Resource
    private UserApplicationService userApplicationService;

    @Resource
    @Lazy
    private SpaceUserApplicationService spaceUserApplicationService;

    @Override
    public long addSpace(SpaceAddRequest spaceAddRequest, User loginU

    }


    @Override
    public QueryWrapper<Space> getQueryWrapper(SpaceQueryRequest spac
        return spaceDomainService.getQueryWrapper(spaceQueryRequest);
    }


    @Override
    public SpaceVO getSpaceVO(Space space, HttpServletRequest request

    }


    @Override
    public Page<SpaceVO> getSpaceVOPage(Page<Space> spacePage, HttpSe

    }

    @Override
    public void fillSpaceBySpaceLevel(Space space) {
        spaceDomainService.fillSpaceBySpaceLevel(space);
    }

    @Override
    public void checkSpaceAuth(User loginUser, Space space) {
        spaceDomainService.checkSpaceAuth(loginUser, space);
    }
```

```
}
```

由于 interfaces 层要调用应用服务层来实现功能，为了方便，可以直接让空间应用服务继承 MyBatis Plus 的接口和实现类，减少样板增删改查方法的编写（比如 getById）。

💡 小技巧：只要发现不调用其他应用服务的方法、并且不调用“当前类中依赖其他应用服务”的方法，就可以改为调用领域服务；否则该方法需要在应用服务中实现。

4）重构领域服务层

领域服务层遵循的原则：

- 需要调用数据库服务（repository）或基础设施层（infrastructure）来完成业务逻辑
- 可以根据需要，将和实体强相关的业务逻辑下沉到 实体类 中

比如 validSpace 方法可以下沉到实体类中，因为校验逻辑不涉及调用数据库，是对实体本身的校验。

遵循原则编写领域服务层的代码，以 SpaceDomainServiceImpl 为例：

```
@Service
public class SpaceDomainServiceImpl extends ServiceImpl<SpaceMapper,


    @Override
    public QueryWrapper<Space> getQueryWrapper(SpaceQueryRequest spac

    }

    @Override
    public void fillSpaceBySpaceLevel(Space space) {

        SpaceLevelEnum spaceLevelEnum = SpaceLevelEnum.getEnumByValue
        if (spaceLevelEnum != null) {
            long maxSize = spaceLevelEnum.getMaxSize();
            if (space.getMaxSize() == null) {
                space.setMaxSize(maxSize);
            }
            long maxCount = spaceLevelEnum.getMaxCount();
            if (space.getMaxCount() == null) {
                space.setMaxCount(maxCount);
```

```
            }
        }
    }


    @Override
    public void checkSpaceAuth(User loginUser, Space space) {

    }
}
```

其实上述代码中，还可以进一步将方法下沉到实体类中哦，应该下沉哪个方法呢？

💡 小技巧

1. 修改领域服务时，如果发现某个方法没被 application 调用（IDE 显示灰色），就可以直接移除掉。
2. 如果想节省重复编写增删改查等样板代码的时间，应用服务或领域服务也可以直接继承 MyBatis Plus 的接口和实现类，这样虽然 DDD 目录结构不是 100% 标准，但是能大幅减少开发成本。

4、重构 Controller

1）首先将原始的空间相关的 3 个 Controller 移动到 `interfaces.controller` 包中。

2）为保证接口层的精简，需要将其中的代码下沉到 **转换类和应用服务** 中。首先编写转换类 `interfaces.assembler.SpaceAssembler` 和 `SpaceUserAssembler`，负责将 DTO 转为实体类：

```
public class SpaceAssembler {

    public static Space toSpaceEntity(SpaceAddRequest request) {
        Space space = new Space();
        BeanUtils.copyProperties(request, space);
        return space;
    }

    public static Space toSpaceEntity(SpaceUpdateRequest request) {
        Space space = new Space();
        BeanUtils.copyProperties(request, space);
        return space;
    }
```

```
    public static Space toSpaceEntity(SpaceEditRequest request) {
        Space space = new Space();
        BeanUtils.copyProperties(request, space);
        return space;
    }
}
```

```
public class SpaceUserAssembler {

    public static SpaceUser toSpaceUserEntity(SpaceUserAddRequest req
        SpaceUser spaceUser = new SpaceUser();
        BeanUtils.copyProperties(request, spaceUser);
        return spaceUser;
    }

    public static SpaceUser toSpaceUserEntity(SpaceUserEditRequest re
        SpaceUser spaceUser = new SpaceUser();
        BeanUtils.copyProperties(request, spaceUser);
        return spaceUser;
    }
}
```

3）将 Controller 的代码下沉到应用服务中，调用应用服务和 Assembler 来处理请求，可能会涉及到应用服务方法的参数修改。其中 updateSpace、editSpace 是改造的重点，需要调用 Assembler 和应用服务层完成功能，下面只列举修改的关键代码：

```
@RestController
@RequestMapping("/space")
@Slf4j
public class SpaceController {

    @Resource
    private SpaceApplicationService spaceApplicationService;

    @Resource
    private UserApplicationService userApplicationService;

    @Resource
    private SpaceUserAuthManager spaceUserAuthManager;
```

```
    @PostMapping("/add")
    public BaseResponse<Long> addSpace(@RequestBody SpaceAddRequest s

        User loginUser = userApplicationService.getLoginUser(request)

        long newSpaceId = spaceApplicationService.addSpace(spaceAddRe
    }
```

```
@PostMapping("/delete")
public BaseResponse<Boolean> deleteSpace(@RequestBody DeleteReque
    User loginUser = userApplicationService.getLoginUser(request)
    long id = deleteRequest.getId();

    Space oldSpace = spaceApplicationService.getById(id);
    ThrowUtils.throwIf(oldSpace == null, ErrorCode.NOT_FOUND_ERRO

    spaceApplicationService.checkSpaceAuth(loginUser, oldSpace);

    boolean result = spaceApplicationService.removeById(id);
}


@PostMapping("/update")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Boolean> updateSpace(@RequestBody SpaceUpdate

    Space space = SpaceAssembler.toSpaceEntity(spaceUpdateRequest

    spaceApplicationService.fillSpaceBySpaceLevel(space);

    space.validSpace(false);

    long id = spaceUpdateRequest.getId();
    Space oldSpace = spaceApplicationService.getById(id);
    ThrowUtils.throwIf(oldSpace == null, ErrorCode.NOT_FOUND_ERRO

    boolean result = spaceApplicationService.updateById(space);
}



@GetMapping("/get")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Space> getSpaceById(long id, HttpServletReque

    Space space = spaceApplicationService.getById(id);
}


@GetMapping("/get/vo")
public BaseResponse<SpaceVO> getSpaceVOById(long id, HttpServletR

    Space space = spaceApplicationService.getById(id);
    ThrowUtils.throwIf(space == null, ErrorCode.NOT_FOUND_ERROR);
    SpaceVO spaceVO = spaceApplicationService.getSpaceVO(space, r
    User loginUser = userApplicationService.getLoginUser(request)
}


@PostMapping("/list/page")
@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)
public BaseResponse<Page<Space>> listSpaceByPage(@RequestBody Spa

    Page<Space> spacePage = spaceApplicationService.page(new Page
            spaceApplicationService.getQueryWrapper(spaceQueryReq
}


@PostMapping("/list/page/vo")
```

```
    public BaseResponse<Page<SpaceVO>> listSpaceVOByPage(@RequestBody
                                                         HttpServletR

        Page<Space> spacePage = spaceApplicationService.page(new Page
                spaceApplicationService.getQueryWrapper(spaceQueryReq

        return ResultUtils.success(spaceApplicationService.getSpaceVO
    }


    @PostMapping("/edit")
    public BaseResponse<Boolean> editSpace(@RequestBody SpaceEditRequ

        Space space = SpaceAssembler.toSpaceEntity(spaceEditRequest);

        space.setEditTime(new Date());

        space.validSpace(false);
        User loginUser = userApplicationService.getLoginUser(request)

        long id = spaceEditRequest.getId();
        Space oldSpace = spaceApplicationService.getById(id);
        ThrowUtils.throwIf(oldSpace == null, ErrorCode.NOT_FOUND_ERRO

        spaceApplicationService.checkSpaceAuth(loginUser, oldSpace);

        boolean result = spaceApplicationService.updateById(space);
    }



    @GetMapping("/list/level")
    public BaseResponse<List<SpaceLevel>> listSpaceLevel() {

    }

}
```

这样一来，接口的代码更加精简。其实还可以进一步规范 DDD 的架构，比如上面有的方法的实现还可以进一步下沉，是哪些方法呢？

💡 前面也提到了，如果觉得一层一层补充调用方法过于麻烦，可以直接给应用服务或领域服务继承 MyBatis Plus 的 IService 和 ServiceImpl，便于上一层调用。

5、其他代码兼容

尝试启动项目，可能会出现编译错误，我们根据报错提示依次解决即可。

最终，尝试启动项目，只要不报编译错误，就算是重构完成了，即使项目启动不起来也不用在意，因为我们有些服务还没有重构完。

公共服务

现在只剩下公共服务 manager 包的代码没有拆分了，接下来的目标就是对 manager 包的代码进行重构。

重构前，我们要先理解公共服务的本质：

- 跨领域：公共服务通常适用于多个领域，如鉴权、日志、通知等。
- 可复用性：不应该绑定到单一的领域模型或用例。
- 无业务含义：与具体的业务无关，仅提供通用的技术能力。

注意，具体情况具体分析，如果某个服务被各个领域或应用调用，那么它也不能和任何一个领域绑定，处理方式也可以和公共服务类似。

建议根据服务**和业务的结合程度**（通用程度）决定将 manager 包的代码移动到哪个位置。如果公共服务不依赖其他领域或应用服务，可以放到 `infrastructure.common` 包中；但如果依赖这些服务，可以放到根包下的 `shared` 包中，以供所有层使用。这种方式能更好地支持模块化管理和解耦。

回归到本项目，步骤如下：

1. 可以先将 `model.dto.file` 和 `FileManager` 移动到 `manager.upload` 包中，由于该包不依赖任何应用服务，可以直接移动到 `infrastructure.manager` 包中，作为基础设施。
2. auth、websocket、sharding 包依赖多个应用服务（或者和多个领域逻辑相关），因此将这些“公共服务”作为独立的 `shared` 包，放到根包下。

剩余代码

对其他剩余代码进行整理，比如将 FileController、MainController 等代码移动到新的 controller 包中。

** 重构完成后，注意将代码中的原包名全部改为新的包名。
** 比如 Mapper 扫描路径、配置文件指定的分库分表算法路径、接口文档路径等。

五、总结

通过上述 DDD 理论的学习，以及项目重构实战，相信大家已经对 DDD 有了一定的理解。建议大家先回顾一下鱼皮分享的重构方法（根据现有代码拆分 + 逐个领域拆分 + 利用好 IDE 重构 + 方法下沉），然后自己跟着教程实操一遍 DDD 重构，并且可以尝试进一步对图片领域和空间领域进行拆分。

但其实大家应该也感受到了，其实 DDD 并不是多么“高大上”的知识，有点类似于在传统分层架构的基础上多增加了一层“应用服务层”，对于非大型项目来说，反而增加了额外的编码。因此虽然大家学会了 DDD，实际的应用场景也并不多，一定要按需使用。

至此，本项目的编码部分全部完成，之后我们就将本项目部署上线。

全文完