

Implementing Domain Driven Design

*A practical guide for implementing the Domain Driven Design
with the ABP Framework*

Halil İbrahim Kalkan

- 作者: Halil Ibrahim Kalkan
- 发布时间: 六月, 2021 (第一版)
- 页数: 109

《实现领域驱动设计》

—— 基于 ABP Framework 实现领域驱动设计实用指南

- 版本：v1.0
- 生成时间：2021.6.29
- 字数：4W+

前言

翻译缘由

自 ABP vNext 1.0 开始学习和使用该框架，被其优雅的设计和实现吸引，适逢 ABP Framework 4.3 版本发布，官网将实现DDD部分的帮助文档，整理成电子书《[Implementing Domain Driven Design](#)》发布，标志着ABP对DDD开发支持趋于完善。

参看照英文版电子书，基于对该框架的理解，边学边译，希望让更多人了解、学习和掌握 ABP Framework，为该优秀的开源项目，贡献绵薄之力。

英文版电子书需要注册或发送邮件下载，不想注册的朋友可加 ABP Framework 研习社-QQ 群：726299208 共享文件中查找：Implementing_Domain_Driven_Design_V1.0.pdf 直接下载。

译者简介

- 网名：**iEricLee**
- 博客：[编程悟道](#) 以码传心，以软制道，知行合一！
- 个人QQ：**2900571998**（用于技术合作、项目开发、企业培训等事项沟通，其他勿扰！）

本书解决的现实问题

对于大多数开发者，苦于学习了DDD开发的理论和指导原则，却在项目或代码层面没有与DDD理论相配套的支持框架，这一点成为很多开发者实施DDD的障碍。

DDD落地实用指南，有助于更好地理解 ABP Framework 和更好地实现 DDD。译者在以前使用 ABP Framework 时的一些疑惑和问题，也在学习过程中豁然开朗。

需要说明的是：

1. 本书侧重DDD实现的最佳实践和原则，完整的DDD理论学习可以补充阅读参考书。

2. 这是一份实施领域驱动设计（DDD）的实用指南。虽然实施细节是基于 ABP Framework 基础设施，但基本概念、原则和模式可以应用于任何解决方案。

学习帮助

为了更好地帮助大家在使用ABP框架实践DDD开发过程中，遇到问题时，讨论、交流！创建 **ABP Framework 研习社（QQ群：726299208）**

专注 **ABP Framework** 技术分析、讨论交流、资料共享、示例源码等，欢迎加入！



ABP Framework 研...

群号: 726299208



扫一扫二维码，入群聊。

目录

- 前言
 - 翻译缘由
 - 译者简介
 - 本书解决的现实问题
 - 学习帮助
 - 目录
- 简介
 - 目标
 - 简单代码
- 什么是领域驱动设计？
 - OOP 和 SOLID
 - DDD 和 Clean Architecture
 - 核心构件
 - 领域层构件
 - 应用层构件
- 全景图
 - 项目分层
 - 领域层
 - 应用层
 - 展示层
 - 远程服务层

- 基础层
 - 其他项目
- 项目依赖关系
- DDD应用程序的执行流程
- 通用原则
- 构件
 - 领域示例
 - 聚合
 - 聚合和聚合根原则
 - 聚合和聚合根最佳实践
 - 仓储
 - 仓储的通用原则
 - 仓储中不包含领域逻辑
 - 规约
 - 在实体中使用规约
 - 在仓储中使用规约
 - 组合规约
 - 领域服务
 - 应用服务
 - 数据传输对象
 - 输入DTO最佳实践
 - 不要在输入DTO中定义不使用的属性
 - 不要重用输入DTO
 - 输入DTO中验证逻辑
 - 输出DTO最佳实践
 - 对象映射
- 实体创建和更新
 - 实体创建
 - 实体创建时应用领域规则
 - 实体更新
- 领域逻辑和应用逻辑
 - 多应用层
 - 示例：正确区分应用逻辑和领域逻辑

简介

这是一份实施领域驱动设计（DDD）的实用指南。虽然实施细节依赖于 ABP 框架 基础设施，但核心概念、原则和模式适用于任何类型的解决方案，即使它不是一个 .NET 解决方案。

目标

本书的目标是：

- 介绍并解释DDD架构、概念、原则、模式和构建模块。
- 解释ABP框架提供的分层架构和解决方案结构
- 通过给出具体的例子，介绍实现DDD模式和最佳实践的**明确规则**。
- 展示ABP Framework 为您提供了哪些基础设施，以便以适当的方式实施DDD。
- 最后，提供创建可维护的代码库的软件开发最佳实践和建议。

简单代码

Playing football is very simple, but playing simple football is the hardest thing there is. — Johan Cruyff

踢足球非常简单，但踢简单的足球是最难的事情。

如果我们把这句名言用于编程，我们可以说：

编写代码非常简单，但编写简单的代码是最难的事情。

一旦你的应用程序扩展，就很难遵循这些规则。有时候，你会为了节省时间而打破规则，当然在短期节省的时间会导致中后期消耗更多的时间。项目中的代码变得**复杂难懂**且难以维护，因为维护成本太高，导致宁愿选择重新开发新的业务程序。

如果您遵循规则和最佳实践，代码将变得更加简单和易于维护，应用程序对需求变更响应更快。

本书在后面的内容中，将介绍**易于实现的简单规则**。

ABP Framework将编写简单代码作为开发目标之一。

什么是领域驱动设计？

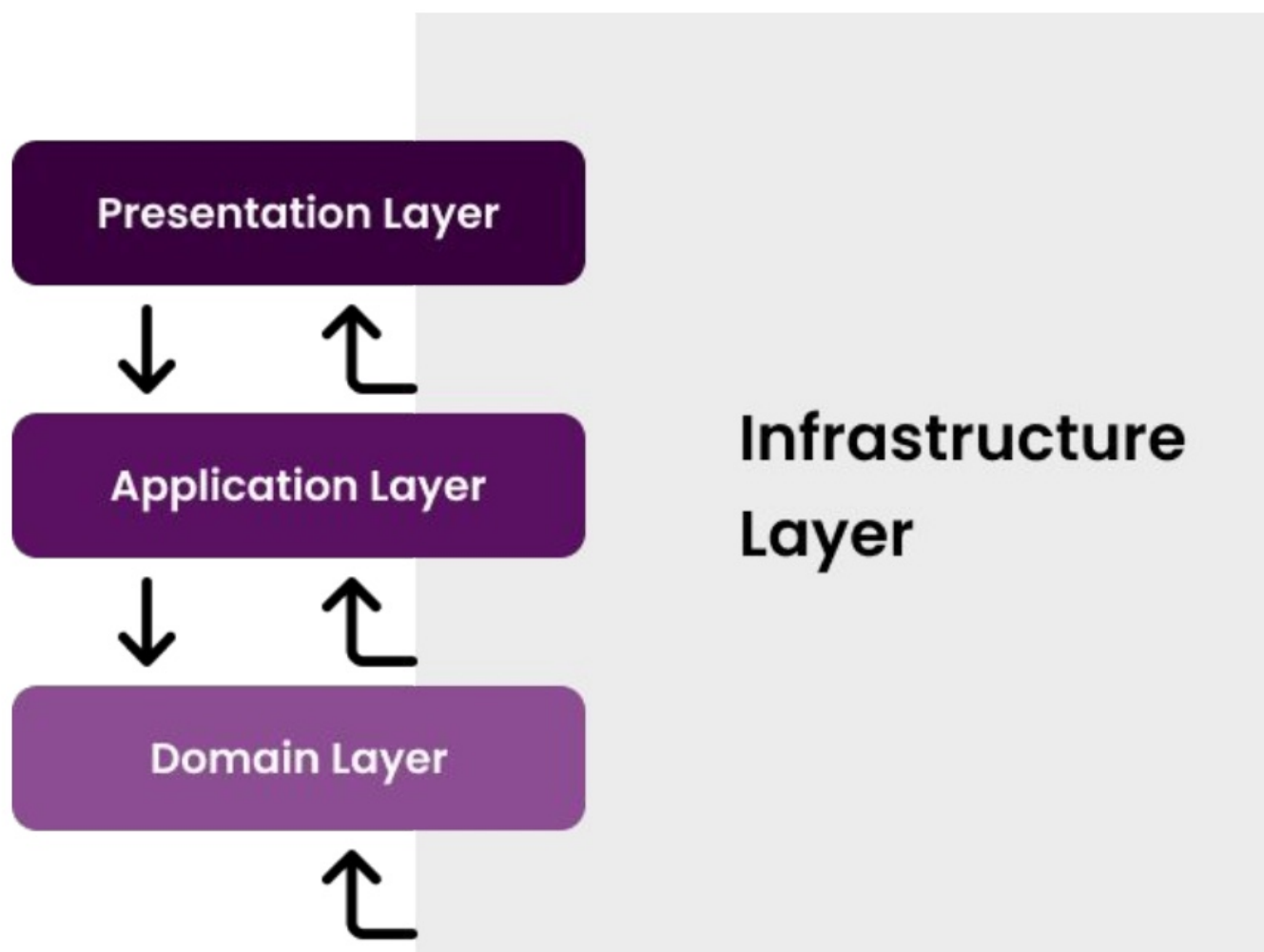
领域驱动设计（简称：DDD）是一种针对**复杂需求**的软件开发方法。将软件实现与不断发展的**模型**联系起来，专注于**核心领域逻辑**，而不是基础设施细节。DDD适用于**复杂领域**和**大规模应用**，而不是简单的CRUD应用。它有助于建立一个**灵活**、**模块化**和**可维护**的代码库。

OOP 和 SOLID

DDD实现高度依赖**面向对象编程思想（OOP）**和**SOLID原则**。实际上，实现并扩展了这些原则。因此，在真正实施DDD时，对OOP和SOLID的良好理解将对您有很大帮助。

DDD 和 Clean Architecture

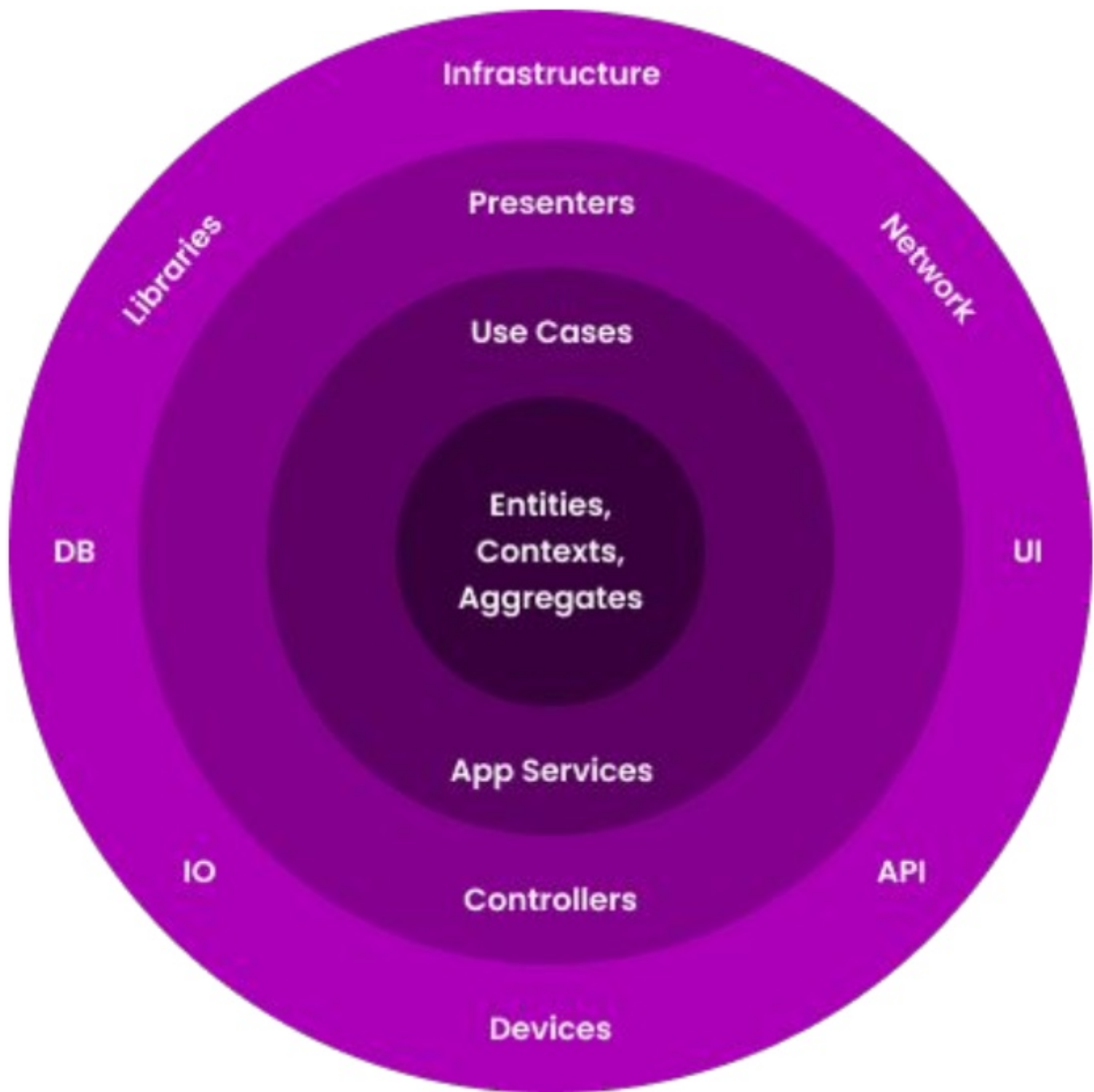
一个基于领域驱动的方案有四个基本层：



业务逻辑分布在两个层中：**领域层**（Domain Layer）和 **应用层**（Application Layer），分别包含不同类型的业务逻辑：

- 领域层：实现领域（或系统）中的用例独立的核心业务逻辑。
- 应用层：基于领域的应用程序用例，应用程序用例可以看作是用户界面上的用户交互。
- 展示层：包含应用程序UI元素（页面、组件等）。
- 基础层：支持层，通过对第三方类库的调用或系统的抽象和集成来实现对其他层的支持。

简洁架构（Clean Architecture） 是与之相同的分层架构，又称为洋葱架构（Onion Architecture）。



从架构图可以看出，每一层只**直接依赖**于它内部的层，最独立的层是**领域层**，显示在最内圈中。

核心构件

DDD主要关注**领域层**和**应用层**，展示层和基础层被看作是细节，业务层不应该依赖于它们，但这并不意味着展示层和基础层不重要，它们也非常重要。展示层中的UI框架和基础层中的数据提供程序有他们自己的实现规则和最佳实践，需要了解和应用。然而，这些并不在DDD的主题中，我们重点来看领域层和应用层的基本构件。

领域层构件

- **实体 (Entity)**：一个实体是一个对象，该对象包含自己的属性和方法，属性用于存储数据和描述状态；方法结合属性实现业务逻辑。一个实体使用唯一标识 (ID) 来表示，两个实体对象ID不同则是为不同的实体。
- **值对象 (Value Object)**：值对象是另一种类型的领域对象，该对象由其属性而不是唯一ID来标识。意思是说，只有全部属性相同才会被认为是同一个对象。值对象通常被实现为**不可变的**，而且大多比实体简单得多。
- **聚合和聚合根**：聚合根是一个特定类型的实体，具有额外的职责。聚合是以聚合根为中心绑定在一起的一组对象，对象包括实体和值对象。
- **仓储 (接口)**：仓储是一个类似集合的接口，被领域层和应用层用来访问数据持久化系统（数据库）。它将数据库的复杂性从业务代码中隐藏起来。领域层包含仓储接口。
- **领域服务**：领域服务是**无状态服务**，实现核心领域业务规则。用于实现依赖于多个聚合（实体）或外部服务的领域逻辑。
- **规约**：用于为实体和其他业务对象定义可命名的、可重用的和可组合的过滤器。
- **领域事件**：领域事件是一种低耦合的通知方式，当一个特定的领域事件发生时，会通知其他服务。

应用层构件

- **应用服务**：应用服务是**无状态服务**，实现应用程序用例。一个应用服务通常获取和返回数据传输对象 (DTOs)，用于展示层。调用领域对象来实现用例。一个用例通常被认为是一个工作单元。
- **数据传输对象 (DTO)**：DTO是简单对象，不包含任何业务逻辑，只用于在应用层和展示层传递数据。
- **工作单元**：一个工作单元是一个原子工作。在工作单元中的所有操作统一提交，要么全部成功，失败则全部回滚。

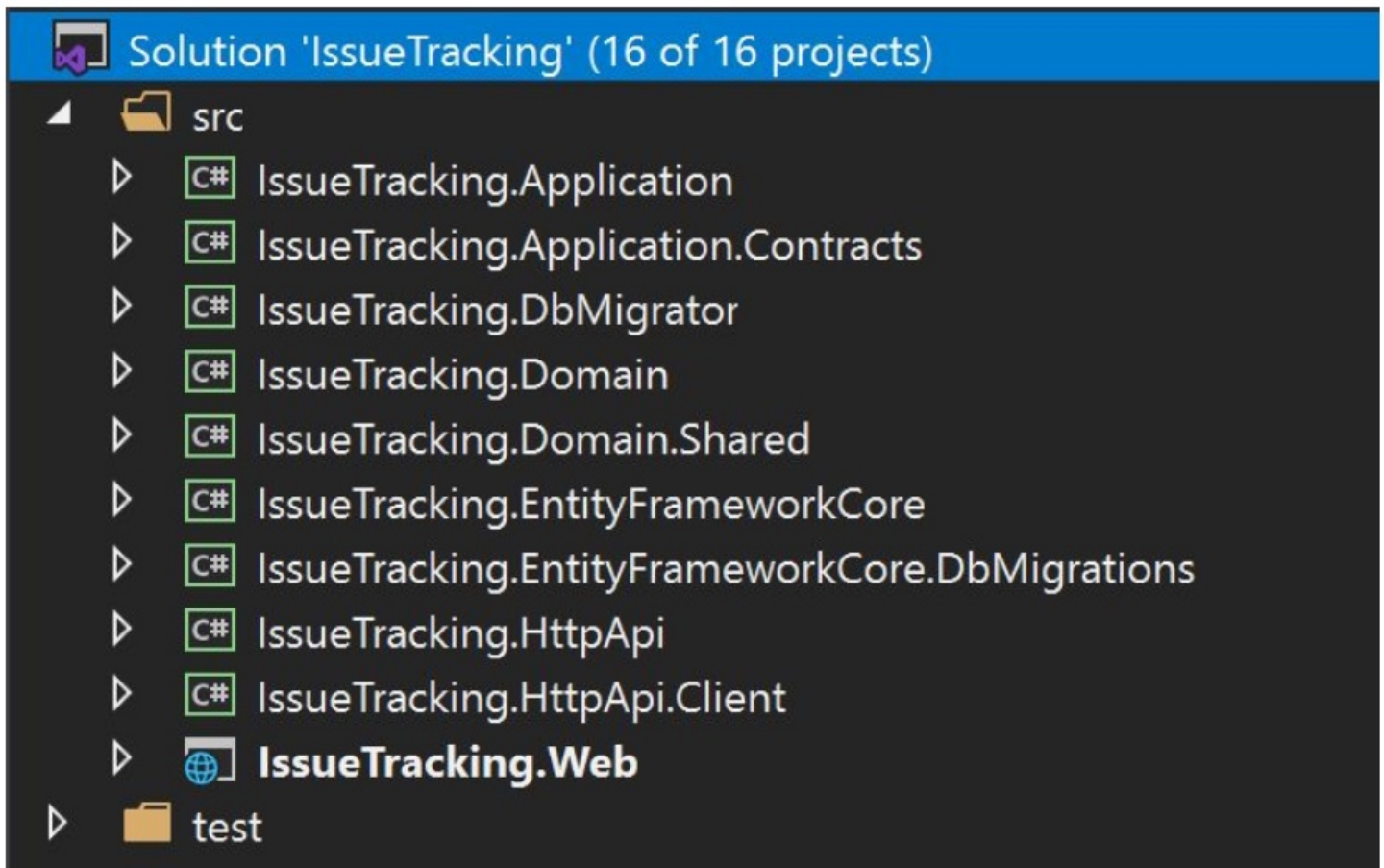
全景图

为了更好地帮助大家在使用ABP框架实践DDD开发过程中，遇到问题时，讨论、交流！创建 **ABP Framework 研习社 (QQ群：726299208)**

专注 **ABP Framework** 技术分析、讨论交流、资料共享、示例源码等，欢迎加入！

项目分层

下图是在 .Net解决方案 (Visual Studio) ，基于 ABP 应用程序启动模板创建的解决方案结构：



解决方案名称为： `IssueTracking` 。解决方案的项目分层考虑到DDD原则，同时兼顾开发和部署实践而划分。

如果选择不同的UI或数据库提供程序，则解决方案结构可能略有不同。但是，领域层和应用层是相同的。如果您想了解更多关于解决方案结构的信息，请参阅应用程序启动模板文档。

示例项目业务场景参考 [GitHub 问题追踪](#)，这个场景比较通用，使用过Git的开发人员都了解。

领域层

领域层拆分为两个项目：

- `IssueTracking.Domain`：**领域层**，该项目包含所有领域层构件，比如：实体、值对象、领域服务、规约、仓储接口等。
- `IssueTracking.Domain.Shared`：**领域共享层**，包含属于领域层，但是与其他层共享的类型。举个例子：定义的常量和枚举，既在领域对象中使用，也要在其他层中使用，放在该项目中。

应用层

应用层拆分为两个项目：

- `IssueTracking.Application.Contracts`: **应用契约层**, 包含应用服务接口和数据传输对象 (用于接口), 该项目被应用程序客户端引用, 比如: WEB项目、API客户端项目。
- `IssueTracking.Application`: **应用层**, 实现在 `Contracts` 项目中定义的接口。

展示层

- `IssueTracking.Web`: 可执行程序, 调用应用服务或APIs, 当前解决方案中是 `ASP.NET Core MVC/Razor Pages` 应用。

ABP框架提供不同类型的UI框架, 比如: Angular和Blazor。如果采用这种UI框架, 解决方案为前后端分离架构, 解决方案中不包含 `IssueTracking.Web` 项目, 而是通过 `IssueTracking.HttpApi.Host` 项目作为一个独立的端点提供 HTTP API 服务, 供客户端调用。

远程服务层

- `IssueTracking.HttpApi`: **远程服务层**, 该项目用于定义 HTTP APIs, 通常包含 MVC Controller 及相关的模型。

大多数时候, API Controller 只是应用服务的包装器, 以便将它们公开给远程客户端。因为ABP框架提供根据应用服务接口自动生成API Controller, 实现**自动配置并将你的应用服务公开为API控制器**, 所以通常不会在这个项目中创建控制器。

- `IssueTracking.HttpApi.Client`: **远程服务代理层**, 客户端应用程序引用该项目, 将直接通过依赖注入使用远程应用服务, 该项目基于ABP Framework**动态C#客户端API代理系统**实现。在C#项目中需要调用HTTP APIs时, 会非常有用。

在解决方案的 `test` 文件夹中有一个控制台应用程序, 名为 `IssueTracking.HttpApi.Client.ConsoleTestApp`。它只是使用 `IssueTracking.HttpApi.Client` 项目来消费应用程序所暴露的API。它只是一个演示应用程序, 可以安全地删除它。如果认为不需要, 甚至可以删除 `IssueTracking.HttpApi.Client` 项目。

基础层

实现DDD时, 可以使用一个**基础层项目**来实现所有的集成和抽象, 当然也可以为不同依赖创建不同项目。

建议折中处理, 为**核心基础依赖**创建单独项目, 比如: Entity Framework Core; 另外创建一个**公共基础项目**存放其他基础设施。

启动模板中包含两个项目对 Entity Framework Core 进行集成:

- IssueTracking.EntityFrameworkCore: **EF Core核心基础依赖项目**，包含：数据上下文、数据库映射、EF Core仓储实现等。
- IssueTracking.EntityFrameworkCore.DbMigrations: **数据迁移项目**，是一个特殊的工具项目，用于管理 Code First 数据迁移。项目中有独立的数据上下文，用于数据迁移。除了需要创建新的数据库迁移或添加应用程序模块增加相应的表时，需要创建一个新的数据库迁移之外，通常不会涉及这个项目。

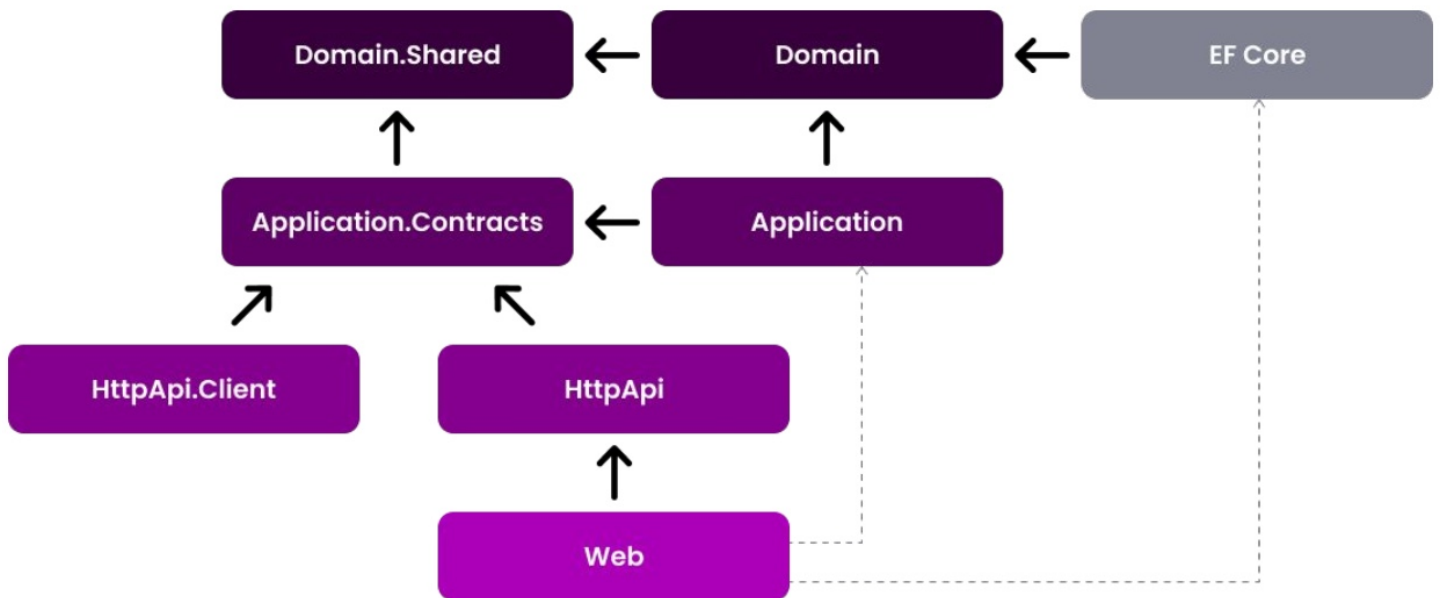
可能你会疑惑为什么集成EF Core创建了两个项目，因为**模块化**的需要。每一个模块有其独立的 `DbContext`，应用程序也有一个 `DbContext`。`DbMigration` 项目包含用于跟踪和应用单个迁移模块的联合。虽然大多数时候您不需要了解它，但您可以查看 EF Core迁移文档，以获得更多信息。

其他项目

还有一个项目，`IssueTracking.DbMigrator`，一个简单的控制台应用程序，当你执行它时，会迁移数据库结构并初始化种子数据。这是一个有用的实用程序，可以在开发和生产环境中使用它。

项目依赖关系

下图是解决方案中项目引用（依赖）关系



前面我们讲解了各个项目的作用，接下来梳理项目之前的关系：

- `Domain.Shared` 其他项目直接或间接引用，项目中定义的类型在所有项目中共享。
- `Domain` 只引用 `Domain.Shared`，比如：在 `Domain.Shared` 中定义的 `IssueType` 枚举类型需要在 `Domain` 项目中 `Issue` 实体中用到。

- `Application.Contracts` 依赖 `Domain.Shared`，这样我们可以在 DTOs 中使用这些共享类型。比如：`CreateIssueDto` 中可以直接使用 `IssueType` 枚举。
- `Application` 依赖 `Application.Contracts`，因为 `Application` 实现 `Application.Contracts` 中定义的服务接口和使用 DTO 对象。同时，引用 `Domain` 项目，在应用服务中使用**仓储接口或领域对象**。
- `EntityFrameworkCore` 依赖 `Domain`，映射 `Domain` 对象（实体和值类型）到数据库表（ORM）并实现在 `Domain` 中定义的仓储接口。
- `HttpApi` 依赖 `Application.Contract`，在控制器在内部对应用服务接口进行依赖注入。
- `HttpApi.Client` 依赖 `Application.Contract` 消费应用服务
- `Web` 依赖 `HttpApi`，发布里面定义的 HTTP APIs。另外，通过这种方式，它间接地依赖于 `Application.Contracts` 项目，可以在页面/组件中使用应用服务。

虚拟依赖

当你仔细查看解决方案依赖关系图时，会看到还有两个依赖关系，在上图中用虚线表示。`Web` 项目依赖于 `Application` 和 `EntityFrameworkCore` 项目，理论上不应该是这样，但实际上是这样。

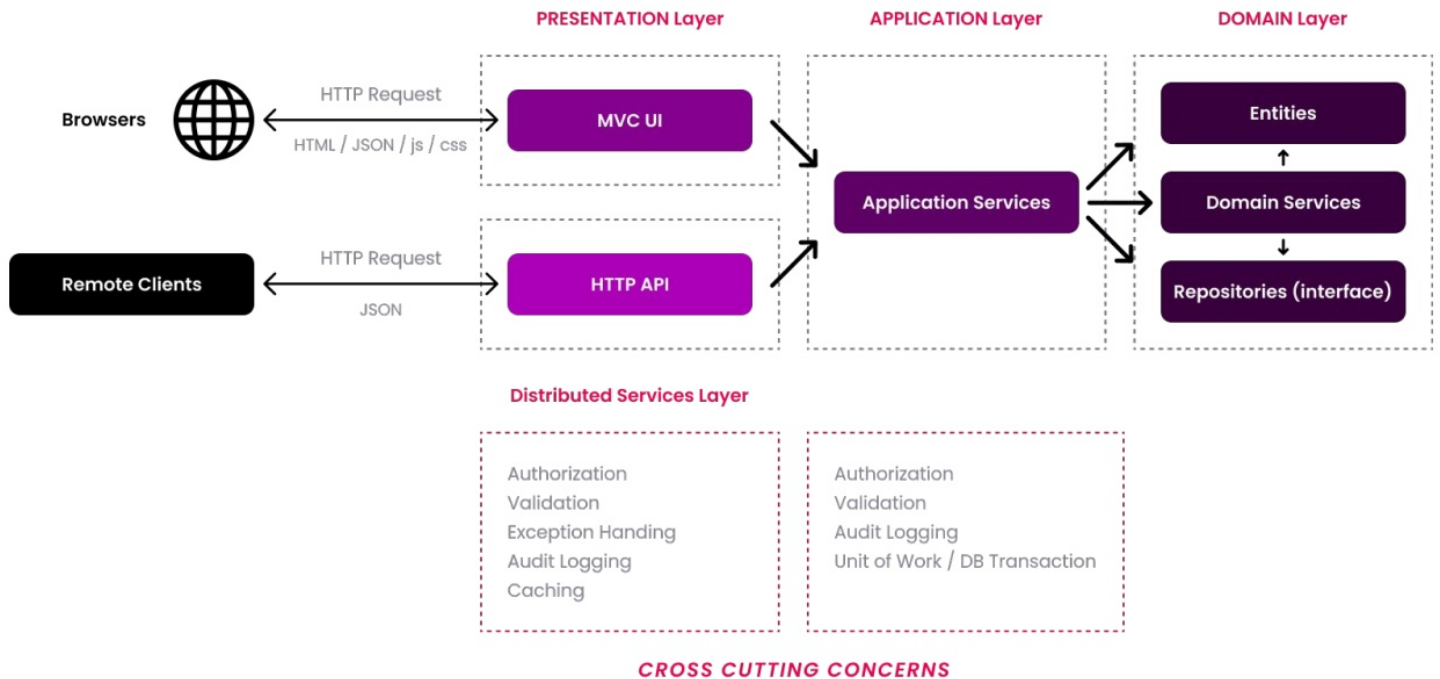
这是因为 `Web` 是运行和托管应用程序的最终项目，应用程序**在运行时需要应用服务和仓储的实现**。

这个设计决定有可能让你在展示层中使用**实体和EF Core 对象**，但这应该是严格避免的。然而，我们发现替代设计过于复杂。在这里，如果你想消除这种依赖性，有两个备选方案：

- 将 `Web` 项目转换为 Razor 类库类型，然后创建新项目，比如：`Web.Host`，引用 `Web` 项目、`Application` 和 `EntityFrameworkCore` 项目。在新项目中，不需要编写任何 UI 代码，只用来做承载项目。
- 从 `Web` 项目中移除 `Application` 和 `EntityFrameworkCore` 项目引用，作为 ABP **插件模块** 在应用初始化时加载程序集。

DDD应用程序的执行流程

下图显示基于DDD模式开发的Web应用请求的基本流程：



- 通过UI用户交互（可以看做是一个用例）发起HTTP请求到服务器
- 在展示层 MVC Controller（HTTP API）或 Razor Page Handler（Razor Pages）接收并处理请求，在此阶段执行**横切关注点**，如：授权、输入验证、异常处理、审计日志、缓存等。Controller或Page在构造函数中注入应用服务接口，调用方法发送和接收DTO对象。
- 应用服务使用领域对象（实体、仓储接口、领域服务等）实现用例。在此阶段，应用层执行**横切关注点**，如：授权、验证、审计日志、工作单元等。一个应用服务方法是一个工作单元，具有原子性。

大多数横切关注点在ABP框架中自动实现或按照约定实现，无需额外编写代码。

通用原则

在进入DDD之前，让我们梳理下DDD通用原则。

数据库（Database Provider / ORM）独立性原则

领域层和应用层不知道项目中使用的 ORM 和 Database Provider。只依赖于仓储接口，并且仓储接口**不适合使用任何 ORM 特殊对象**。

这一原则的主要原因是：

1. 使领域层和应用层与基础层独立，因为基础层将来可能更改，或者你可能需要支持其他类型数据库。
2. 使领域和应用聚焦在业务代码上，通过将基础设施实现细节隐藏于仓储之后，使您的领域和应用服务专注于业务代码。

3. 易于自动化测试，因为可以通过仓储接口模拟仓储数据。

根据这一原则，除**启动应用程序**外，解决方案中的任何项目都没有引用 EntityFrameworkCore 项目。

关于数据库独立性原则的讨论

尤其是**原因1**会深深地影响你的领域对象设计（比如，实体关系）和应用层代码。假设你当前使用 Entity Framework Core 操作关系型数据库，后期希望切换为 MongoDB，这就决定你不能使用 EF Core 中独有功能，因为在MongoDB中不被支持。

举个例子：

- 不能使用**更改跟踪（Change Tacking）**，因为 MongoDB 不支持。所以，需要显式更改实体。
- 不能在实体中使用**导航属性（Navigation Properties）**或集合关联其他聚合，因为可能在文档数据库中不支持。

那么如何解决实体关联的问题？记住规则：仅通过**Id**引用其他聚合。

如果你认为这些功能对你很重要，而且你永远不会弃用 EF Core，我们认为这个原则是可以有**弹性的**，但是我们仍然建议使用**仓储模式**来隐藏基础设施的实现细节。

ABP Framework 为仓储接口 IRepository 提供获取 IQueryable 对象的扩展方法 GetQueryableAsync()，使我们在使用仓储时可以直接使用**标准LINQ扩展方法**。

展示技术无关性原则

展示层技术（UI框架）是应用程序中变化最多的部分，将领域层和应用层设计成**完全不知道**展示层技术或框架是非常重要的。

这一原则相对容易实现，而ABP的启动模板使其更加容易实现，选择不同UI框架自动生成对应的启动模板项目。

在某些场景下，你可能需要在应用层和展示层使用相同的逻辑。举例，你可能需要在两个层中进行验证和授权。在UI层检测是为了提高用户体验，在应用层和领域层是出安全和数据有效性考虑。这是非常正常和必要的。

聚焦状态变化，而不是性能优化

DDD聚焦**领域对象如何变化和如何交互**；如何创建实体和改变属性，并且保持数据的完整性、有效性；如何创建方法，实现业务规则。

DDD没有考虑**报表和大规模查询**等需要高性能的业务场景，如果你的应用程序中没有花哨的仪表盘或报表功能，谁会去考虑呢？意思是我们需要自己考虑性能问题。

性能优化或技术选型，只要**不影响到业务逻辑**，可以自由使用 SQL Server 全部功能，比如：查询优化、索引、存储过程等技术；甚至使用一个其他数据源，如：ElasticSearch，来负责报表功能。

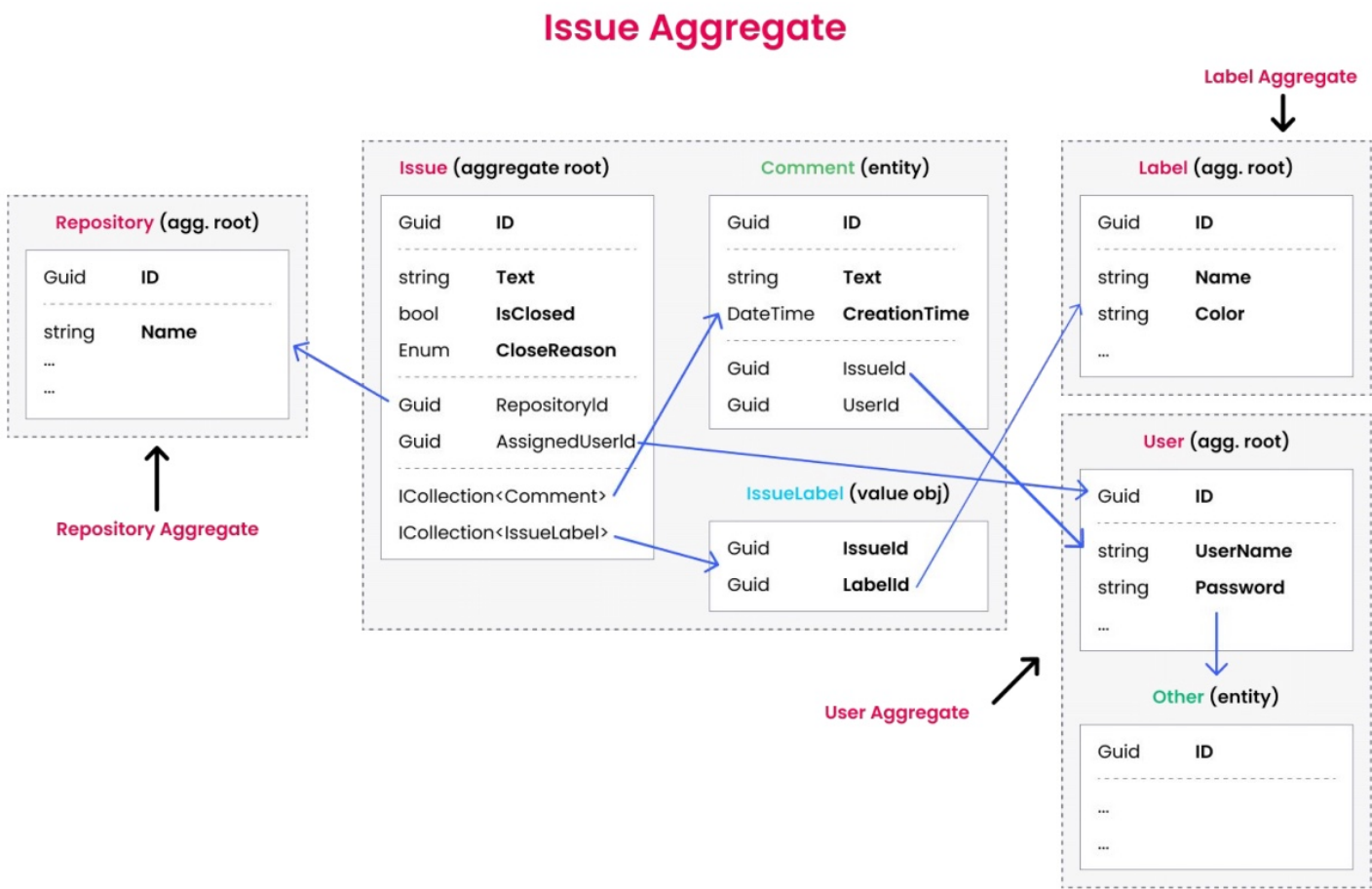
构件

这是本指南的最重要部分。我们将通过例子介绍和解释一些**显式规则**。在实现领域驱动设计时，应该遵循这些规则并将其应用到解决方案中。

领域示例

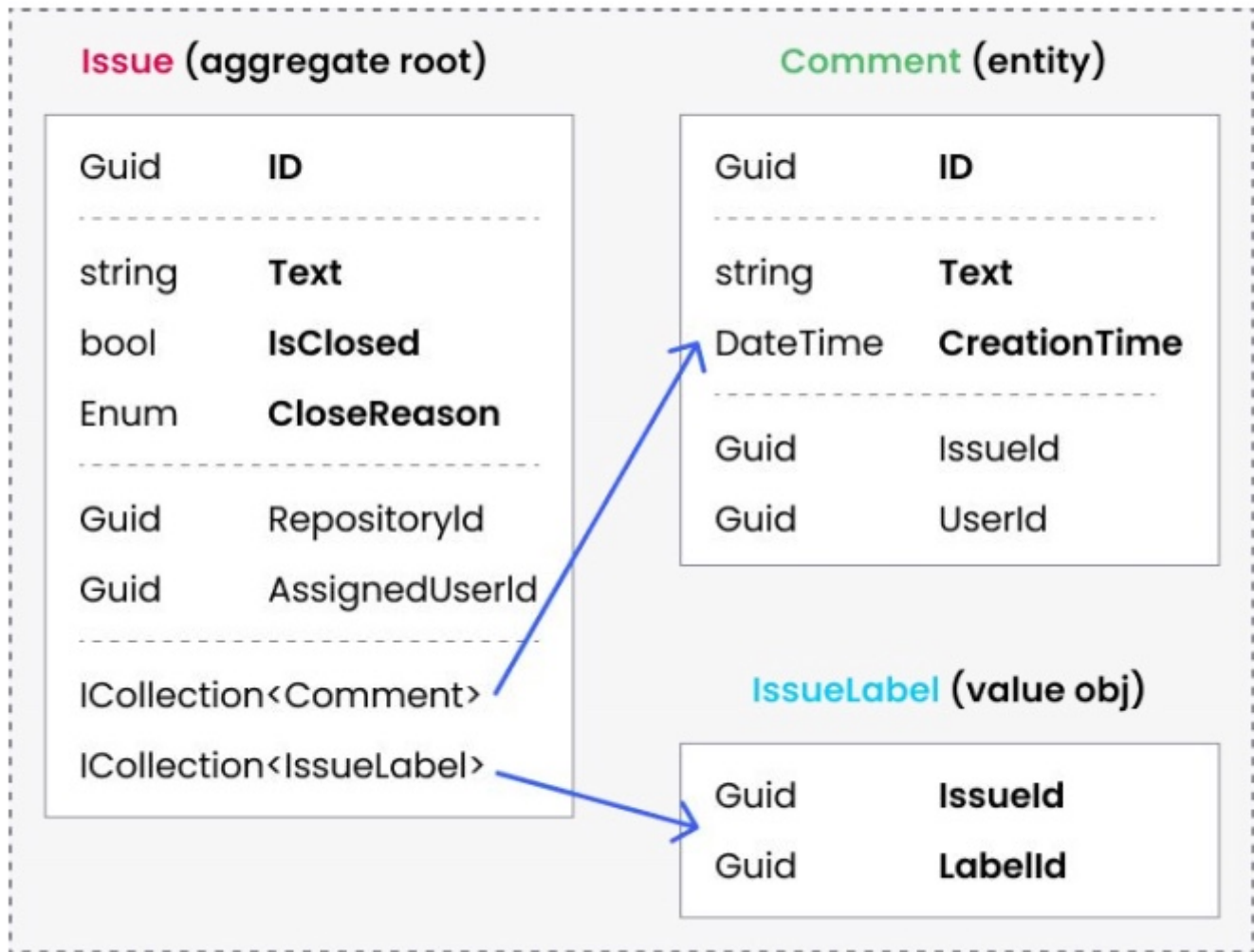
例子中会用到 GitHub 的一些概念，如：**Issue**（建议）、**Repository**（代码仓库）、**Label**（标签）和**User**（用户），作为开发者的你可能已经很熟悉了。

下图显示了业务场景对应的聚合、聚合根、实体、值对象以及它们之间的关系。



Issue 聚合是由 **Issue**（聚合根）、**Comment**（实体）和 **IssueLabel**（值对象）组成的集合。因为其他聚合相对简单，所以我们重点分析 **Issue 聚合**。

Issue Aggregate



聚合

正如前面所讲，一个聚合是一系列对象（实体和值对象）的集合，通过聚合根将所有关联对象绑定在一起。本节将介绍与聚合相关的最佳实践和原则。

我们对**聚合根**和**子集合实体**都使用**实体**这个术语，除非明确写出聚合根或子集合实体。

聚合和聚合根原则

包含业务原则

- 实体负责实现与其自身属性相关的业务规则。
- 聚合根还负责其子集合实体状态管理。

- 聚合应该通过实现**领域规则**和**规约**来保持自身的**完整性和有效性**。这意味着，与数据传输对象（DTO）不同，实体具有**实现业务逻辑的方法**。实际上，我们应该**尽可能在实体中实现业务规则**。

单个单元原则

聚合及其所有子集合，作为单个单元被检索和保存。例如：如果向 **Issue** 添加 **Comment**，需要这样做：

- 从数据库中获取 **Issue** 包含所有子集合：**Comments**（该问题的评论列表）和 **IssueLabels**（该问题的标签集合）。
- 在 **Issue** 类中调用方法添加一个新的 **Comment**，比如：`Issue.AddCommnet(...)`
- 作为一个单一的数据库更新操作，将 **Issue**（包括所有子集合）保存到数据库。

对于习惯使用 EF Core 和 关系数据的开发者来说，这看起来似乎有些奇怪。获取 **Issue** 的所有数据是**没有必要且低效**的。为什么我们不直接执行一个**SQL插入**命令到数据库，而不查询任何数据呢？

答案是，我们应该在代码中**实现业务规则并保持数据的一致性和完整性**。如果我们有一个业务规则，如：*用户不能对**锁定**的 **Issue** 进行评论*，我们如何不通过检索数据库中数据的情况下，检查 **Issue** 的锁定状态呢？所以，只有当应用程序代码中的相关对象可用时，即获取到聚合及其所有子集合数据时，我们才能执行该业务规则。

另一方面，MongoDB开发者会发现这个规则非常自然。因为在 MongoDB 中，一个聚合对象（包括子集合）被保存在数据库中的一个**集合**中，而在关系型数据库中，它被分布在数据库中**几个表**中。因此，当你得到一个聚合时，所有的子集合已经作为查询的一部分被检索出来了，不需要任何额外配置。

ABP框架有助于在您的应用程序中实现这一原则。

示例：添加 **Comment** 到 **Issue**

```
public class IssueAppService : ApplicationService ,IIssueAppService
{
    private readonly IRepository<Issue,Guid> _issueRepository;
    public IssueAppService(IRepository<Issue,Guid> issueRepository)
    {
        _issueRepository = issueRepository;
    }
    [Authorize]
    public async Task CreateCommentAsync(CreateCommentDto input)
    {
        var issue = await _issueRepository.GetAsync(input.IssueId);
        issue.AddComment(CurrentUser.GetId(),input.Text);
        await _issueRepository.UpdateAsynce(issue);
    }
}
```

`_issueRepository.GetAsync(...)` 方法默认作为单个单元检索 Issue 对象并包含所有子集合。对于 MongoDB 来说这个操作开箱即用，但是使用 EF Core 需要配置聚合与数据库映射，配置后 EF Core 仓储实现会自动处理。`_issueRepository.GetAsync(...)` 方法提供一个可选参数 `includeDetails`，可以传递值 `false` 禁用该行为，不包含子集合对象，只在需要时启用它。

关于配置，请参阅 **EF Core 集成** 帮助文档中[加载关联实体](#)一节。

`Issue.AddComment(...)` 传递参数 `userId` 和 `text`，表示用户ID和评论内容，添加到 Issue 的 `Comments` 集合中，并实现必要的业务逻辑验证。

最后，使用 `_issueRepository.UpdateAsync(...)` 保存更改到数据库。

EF Core 提供 变更跟踪 (Change Tracking) 功能，实际上你不需要调用

`_issueRepository.UpdateAsync(...)` 方法，会自动进行保存。这个功能是由 **ABP 工作单元系统** 提供，应用服务的方法作为一个单独的工作单元，在执行完之后会自动调用

`DbContext.SaveChanges()`。当然，如果使用 MongoDB 数据库，则需要显式地更新已经更改的实体。

所以，如果你想要编写独立于数据库提供程序的代码，应该总是为要更改的实体调用 `UpdateAsync()` 方法。

事务边界原则

一个聚合通常被认为是一个事务边界。如果用例使用单个聚合，读取并保存为单个单元，那么对聚合对象所做的所有更改，将作为原子操作保存，而不需要显式地使用数据库事务。

当然，我们可能需要处理将多个聚合实例作为单一用例更改的场景，此时需要使用数据库事务确保更新操作的原子性和数据一致性。正因为如此，ABP 框架为一个用例（即一个应用程序服务方法）显式地使用数据库事务，一个应用程序服务方法，就是一个工作单元。参看：[工作单元帮助文档](#)获取更多帮助信息。

可序列化原则

聚合（包含根实体和子集合）应该是可序列化的，并且可以作为单个单元在网络上进行传输。举个例子，MongoDB 序列化聚合为 Json 文档保存到数据库，反序列化从数据库中读取的 Json 数据。

当您使用关系数据库和 ORM 时，没有必要这样做。然而，它是领域驱动设计的一个重要实践。

聚合和聚合根最佳实践

以下最佳实践确保实现上述原则。

只通过ID引用其他聚合

一个聚合应该只通过其他**聚合的ID**引用聚合，这意味着你不能添加导航属性到其他聚合。

- 这条规则使得实现可序列化原则得以实现。
- 可以防止不同聚合相互操作，以及将聚合的业务逻辑泄露给另一个聚合。

我们来看一个例子，两个聚合根： `GitRepository` 和 `Issue`：

```
public class GitRepository:AggregateRoot<Guid>
{
    public string Name {get;set;}
    public int StarCount{get;set;}
    public Collection<Issue> Issues {get;set;} //错误代码示例
}

public class Issue:AggregateRoot<Guid>
{
    public string Text{get;set;}
    public GitRepository Repository{get;set;} //错误代码示例
    public Guid RepositoryId{get;set;} //正确示例
}
```

- `GitRepository` 不应该包含 `Issue` 集合，他们是不同聚合。
- `Issue` 不应该设置导航属性关联 `GitRepository`，因为他们是不同聚合。
- `Issue` 使用 `RepositoryId` 关联 `Repository` 聚合，正确。

当你有一个 `Issue` 需要关联的 `GitRepository` 时，那么可以从数据库通过 `RepositoryId` 直接查询。

用于 EF Core 和 关系型数据库

在 MongoDB 中，自然不适合有这样的导航属性/集合。如果这样做，在源集合的数据库集合中会保存目标集合对象的**副本**，因为它在保存时被序列化为JSON，这样可能会导致持久化数据的不一致。

然而，EF Core 和关系型数据库的开发者可能会发现这个限制性的规则是不必要的，因为 EF Core 可以在数据库的读写中处理它。

但是我们认为这是一条重要的规则，有助于降低领域的复杂性防止潜在的问题，我们强烈建议实施这条规则。然而，如果你认为忽略这条规则是切实可行的，请参阅前面关于**数据库独立性原则**的讨论部分。

保持聚合根足够小

一个好的做法是保持一个**简单而小**的聚合。这是因为一个聚合体将作为一个**单元**被加载和保存，读/写一个大对象会导致性能问题。

请看下面的例子：

```

public class UserRole:ValueObject
{
    public Guid UserId{get;set;}
    public Guid RoleId{get;set;}
}

public class Role:AggregateRoot<Guid>
{
    public string Name{get;set;}
    public Collection<UserRole> Users{get;set;} //错误示例：角色对应的用户是不断增加的
}
public class User:AggregateRoot<Guid>
{
    public string Name{get;set;}
    public Collection<UserRole> Roles{get;set;}//正确示例：一个用户拥有的角色数量是有限的
}

```

Role聚合 包含 UserRole 值对象集合，用于跟踪分配给此角色的用户。注意，UserRole 不是另一个聚合，对于规则仅通过引用其他聚合没有冲突。

然而，实际却存在一个问题。在现实生活中，一个角色可能被分配给数以千计（甚至数以百万计）的用户，每当你从数据库中查询一个角色时，加载数以千计的数据项是一个重大的性能问题。**记住：聚合是由它们的子集合作为一个单一单元加载的。**

另一方面，用户可能有角色集合，因为实际情况中用户拥有的角色数量是有限的，不会太多。当您使用用户聚合时，拥有一个角色列表可能会很有用，且不会影响性能。

如果你仔细想想，当使用非关系型数据库（如MongoDB）时，当 Role 和 User 都有关系列表时还有一个问题：在这种情况下，相同的信息会在不同的集合中重复出现，将很难保持数据的一致性，每当你 User.Roles 中添加一个项，你也需要将它添加到 Role.Users 中。

因此，根据以下因素来确定聚合边界和大小：

- 考虑对象关联性，是否需要在一起使用。
- 考虑性能，查询（加载/保存）性能和内存消耗。
- 考虑数据的完整性、有效性和一致性。

而实际：

- 大多数聚合根本没有子集合。
- 一个子集合最多不应该包含超过**100-150个条目**。如果您认为集合可能有更多项时，请不要定义集合作为聚合的一部分，应该考虑为集合内的实体提取为另一个聚合根。

聚合根/实体中的主键

- 一个聚合根通常有一个**ID属性**作为其标识符（主键，Primark Key: PK）。推荐使用 **Guid** 作为聚合根实体的PK（参见 [Guid 生成](#) 帮助文档以了解原因）。
- 聚合中的实体（不是聚合根）可以使用**复合主键**。

示例：聚合根和实体

```
//聚合根：单个主键
public class Organization
{
    public Guid Id{get;set;}
    public string Name{get;set;}
    //...
}
//实体：复合主键
public class OrganizationUser
{
    public Guid OrganizationId{get;set;} //主键
    public Guid UserId{get;set;} //主键
    public bool IsOwner{get;set;}
    //...
}
```

- Organization 包含 Guid 类型主键 Id
- OrganizationUser 是 Organization 中的子集合，有复合主键：OrganizationId 和 UserId。

这并不意味着子集合实体应该总是有复合主键，只有当需要时设置；通常是单一的ID属性。

复合主键实际上是关系型数据库的一个概念，因为子集合实体有自己的表，需要一个主键。另一方面，例如：在MongoDB中，你根本不需要为子集合实体定义主键，因为它们是作为聚合根的一部分来存储的。

聚合根/实体构造函数

构造函数是实体的生命周期开始的地方。一个设计良好的构造函数，担负以下职责：

- 获取**所需的实体属性参数**，来**创建一个有效的实体**。应该强制只传递必要的参数，并可以将非必要的属性作为**可选参数**。
- 检查参数的有效性。
- 初始化子集合。

示例：Issue（聚合根）构造函数


```

using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using Volo.Abp;
using Volo.Abp.Domain.Entities;

namespace IssueTracking.Issues
{
    public class Issue:AggregateRoot<Guid>
    {
        public Guid RepositoryId{get;set;}
        public string Title{get;set;}
        public string Text{get;set;}
        public Guid? AssignedUserId{get;set;}
        public bool IsClosed{get;set;}
        public IssueCloseReason? CloseReason{get;set;} //枚举
        public ICollection<IssueLabel> Labels {get;set;}

        public Issue(
            Guid id,
            Guid repositoryId,
            string title,
            string text=null,
            Guid? assignedUserId = null
        ):base(id)
        {
            //属性赋值
            RepositoryId=repositoryId;
            //有效性检测
            Title=Check.NotNullOrWhiteSpace(title,nameof(title));

            Text=text;
            AssignedUserId=assignedUserId;
            //子集合初始化
            Labels=new Collection<IssueLabel>();
        }
        private Issue(){/*反序列化或ORM 需要*/}
    }
}

```

- **Issue** 类通过其构造函数参数，获得属性所需的值，以此创建一个正确有效的实体。
- 在构造函数中验证输入参数的有效性，比如：`Check.NotNullOrWhiteSpace(...)` 当传递的值为空时，抛出异常 `ArgumentException` 。
- 初始化子集合，当使用 `Labels` 集合时，不会获取到空引用异常。
- 构造函数将参数 `id` 传递给 `base` 类，不在构造函数中生成 `Guid`，可以将其委托给另一个 `Guid`生成服务，作为参数传递进来。
- 无参构造函数对于ORM是必要的。我们将其设置为私有，以防止在代码中意外地使用它。

参看帮助文档 [实体](#) 章节，学习更多关于创建实体内容。

实体属性访问器和方法

上面的示例代码，看起来可能很奇怪。比如：在构造函数中，我们强制传递一个不为 null 的 Title 。但是，我们可以将 Title 属性设置为 null ，而对其没有进行任何有效性控制。这是因为示例代码关注点暂时只在构造函数。

如果我们用 `public` **设置器**声明所有的属性，就像上面的 Issue 类中的属性例子，我们就不能在实体的生命周期中强制保持其有效性和完整性。所以：

- 当需要在设置属性时，执行任何逻辑，请将属性设置为私有 `private` 。
- 定义**公共方法**来操作这些属性。

示例：通过方法修改属性

```
namespace IssueTracking.Issues
{
    public Guid RepositoryId {get; private set;} //不更改
    public string Title { get; private set; } //更改，需要非空验证
    public string Text{get;set;} //无需验证
    public Guid? AssignedUserId{get;set;} //无需验证
    public bool IsClosed { get; private set; } //需要和 CloseReason 一起更改
    public IssueCloseReason? CloseReason { get;private set;} //需要和 IsClosed 一起更改

    public class Issue:AggregateRoot<Guid>
    {
        //...
        public void SetTitle(string title)
        {
            Title=Check.NotNullOrWhiteSpace(title,nameof(title));
        }

        public void Close(IssueCloseReason reason)
        {
            IsClosed = true;
            CloseReason =reason;
        }

        public void ReOpen()
        {
            IsClosed=false;
            CloseReason=null;
        }
    }
}
```

- RepositoryId 设置器设置为**私有** private ，因为 Issue 不能将 Issue 移动到另一个 Repository 中，该属性创建之后无需更改。
- Title 设置器设置为**私有**，当需要更改时，可以使用 SetTitle 方法，这是一种**可控**的方式。
- Text 和 AssignedUserId 都有公共设置器，因为这两个字段并没有约束，可以是 null 或任何值。我们认为没有必要定义单独的方法来设置它们。如果以后需要，可以添加更改方法并将其设置器设置为私有。**领域层是内部项目，并不会暴露给客户端使用，所以这种更改不会有问题。**
- IsClosed 和 IssueCloseReason 是成对修改的属性，分别定义 Close 和 ReOpen 方法一起修改他们。通过这种方式，可以防止在没有任何理由的情况下关闭一个问题。

业务逻辑和实体中的异常处理

当你在实体中进行验证和实现业务逻辑，经常需要管理异常：

- 创建特定领域异常。
- 必要时在实体方法中抛出这些异常。

示例：

```

public class Issue:AggregateRoot<Guid>
{
    //..
    public bool IsLocked {get;private set;}
    public bool IsClosed{get;private set;}
    public IssueCloseReason? CloseReason {get;private set;}

    public void Close(IssueCloseReason reason)
    {
        IsClose = true;
        CloseReason =reason;
    }
    public void ReOpen()
    {
        if(IsLocked)
        {
            throw new IssueStateException("不能打开一个锁定的问题! 请先解锁! ");
        }
        IsClosed=false;
        CloseReason=null;
    }
    public void Lock()
    {
        if(!IsClosed)
        {
            throw new IssueStateException("不能锁定一个关闭的问题! 请先打开! ");
        }
    }
    public void Unlock()
    {
        IsLocked = false;
    }
}

```

这里有两个业务规则：

- 锁定的 Issue 不能重新打开
- 不能锁定一个关闭的 Issue

Issue 类在这些业务规则中抛出异常 IssueStateException 。

```
namespace IssueTracking.Issues
{
    public class IssueStateException : Exception
    {
        public IssueStateException(string message)
            :base(message)
        {

        }
    }
}
```

抛出此类异常有两个潜在问题:

1. 在这种异常情况下，终端用户是否应该看到异常(错误)消息？如果是，如何实现本地化异常消息？
因为不能在实体中注入和使用 `IStringLocalizer`，导致不能使用本地化系统。
2. 对于 Web 应用程序或 HTTP API，应该给客户端返回什么 HTTP Status Code？

ABP框架 **Exception Handling 系统**处理了这些问题。

示例：抛出业务异常

```
using Volo.Abp;
namespace IssuTracking.Issues
{
    public class IssueStateException : BusinessException
    {
        public IssueStateExcetipn(string code)
            : base(code)
        {

        }
    }
}
```

- `IssueStateException` 类继承 `BusinessException` 类。ABP框架在请求禁用时默认返回 403 HTTP 状态码；发生内部错误是返回 500 HTTP 状态码。
- `code` 用作本地化资源文件中的一个键，用于查找本地化消息。

现在，我们可以修改 `ReOpen` 方法：

```
public void ReOpen()
{
    if(IsLocked)
    {
        throw new IssueStateException("IssueTracking:CanNotOpenLockedIssue");
    }
    IsClosed=false;
    CloseReason=null;
}
```

建议：使用常量代替魔术字符串 "IssueTracking:CanNotOpenLockedIssue" 。

然后在本地化资源中添加一个条目，如下所示:

```
"IssueTracking:CanNotOpenLockedIssue": "不能打开一个锁定的问题！请先解锁！"
```

- 当抛出异常时，ABP自动使用这个本地化消息(基于当前语言)向终端用户显示。
- 异常Code ("IssueTracking:CanNotOpenLockedIssue") 被发送到客户端，因此它可以以编程方式处理错误情况。

对于当前示例，可以直接抛出 BusinessException ，返回的结果是一样的。参看：[异常处理](#)文档获取更多帮助。

实体中业务逻辑需要用到外部服务

当业务逻辑只使用该实体的属性时，在实体方法中实现业务规则是很简单的。如果业务逻辑需要查询数据库或使用任何应该从依赖注入系统中获取的外部服务时，该怎么办？请记住，**实体不能注入服务**。

有两个方式实现：

- 在实体方法上实现业务逻辑，并将外部依赖项作为方法的参数。
- 创建**领域服务** (Domain Service)

领域服务在后面介绍，现在让我们看看如何在实体类中实现它。

示例：业务规则：一个用户不能同时分配超过3个未解决的问题

```

public class Issue:AggregateRoot<Guid>
{
    //..
    public Guid? AssignedUserId{get;private set;}
    //问题分配方法
    public async Task AssignToAsync(AppUser user,IUserIssueService userIssueService)
    {
        var openIssueCount = await userIssueService.GetOpenIssueCountAsync(user.Id);
        if(openIssueCount >=3 )
        {
            throw new BusinessException("IssueTracking:CanNotOpenLockedIssue");
        }
        AssignedUserId=user.Id;
    }
    public void CleanAssignment()
    {
        AssignedUserId=null;
    }
}

```

- AssignedUserId 属性设置器设置为**私有**，通过 AssignToAsync 和 CleanAssignment 方法进行修改。
- AssignToAsync 获取一个 AppUser 实体，实际上只用到 user.Id ，传递实体是为了**确保参数值是一个存在的用户**，而不是一个随机值。
- IUserIssueService 是一个任意的服务，用于获取分配给用户的问题数量。如果业务规则不满足，则抛出异常。所有规则满足，则设置 AssignedUserId 属性值。

此方法完全实现了应用业务逻辑，然而，它有一些问题：

- 实体变得复杂，因为实体类依赖外部服务。
- 实体变得难用，调用方法时需要注入依赖的外部服务 IUserIssueService 作为参数。

实现此业务逻辑的另一种方法是引入**领域服务**，稍后将对此进行解释。

仓储

仓储（接口）是一组集合的接口，被领域层和应用层用来访问数据持久化系统（数据库），以读写业务对象，业务对象通常是**聚合**。

仓储的通用原则

- 在领域层中定义**仓储接口**，在基础层中实现仓储接口（比如： EntityFrameworkCore 项目或 MongoDB 项目）

- 仓储不包含业务逻辑，专注数据处理。
- 仓储接口应该保持 **数据提供程序/ORM 独立性**。举个例子，仓储接口定义的方法不能返回 `DbSet` 对象，因为该对象由 EF Core 提供，如果使用 MongoDB 数据库则无法实现该接口。
- **为聚合根创建对应仓储**，而不是所有实体。因为子集合实体（聚合）应该通过聚合根访问。

仓储中不包含领域逻辑

虽然这个规则一开始看起来很好理解，但在实际开发过程中，很容易在不经意间将业务逻辑放到仓储中。

示例：从仓储中获取 `inactive` 状态的 `Issue`

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Volo.Abp.Domain.Repositories;

namespace IssueTracking.Issues
{
    public interface IIssueRepository : IRepository<Issue, Guid>
    {
        Task<List<Issue>> GetInactiveIssuesAsync();
    }
}
```

`IIssueRepository` 继承 `IRepository<Issue, Guid>` 接口，添加了 `GetInactiveIssuesAsync()` 方法。与之对应的聚合根类型是 `Issue` 类：

```
public class Issue : AggregateRoot<Guid>, IHasCreationTime
{
    public bool IsClosed { get; private set; }
    public Guid? AssignedUserId { get; private set; }
    public DateTime CreationTime { get; private set; }
    public DateTime? LastCommentTime { get; private set; }
}
```

规则要求我们：仓储不应该知道业务规则，那么问题来了：“什么是 **inactive Issue**（未激活的问题）？”，这是业务规则。

为了更好地理解，我们继续看看接口方法的实现：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using IssueTracking.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Volo.Abp.Domain.Repositories.EntityFrameworkCore;
using Volo.Abp.EntityFrameworkCore;

namespace IssueTracking.Issues
{
    public class EfCoreIssueRepository :
        EfCoreRepository<IssueTrackingDbContext, Issue, Guid>,
        IIssueRepository
    {
        public EfCoreIssueRepository(
            IDbContextProvider<IssueTrackingDbContext> dbContextProvider
        ) : base(dbContextProvider)
        {}

        public async Task<List<Issue>> GetInactiveIssueAsync()
        {
            var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));

            var dbSet = await GetDbSetAsync();
            return await dbSet.Where(i =>
                // 打开状态
                !i.IsClosed &&
                // 无分配人
                i.AssignedUserId == null &&
                // 创建时间在30天前
                i.CreationTime < daysAgo30 &&
                // 没有评论或最后一次评论在30天前
                (i.LastCommentTime == null || i.LastCommentTime < daysAgo30)
            ).ToListAsync();
        }
    }
}

```

在 `GetInactiveIssueAsync` 实现方法中，对于**未激活的Issue** 这条业务规则，需要满足条件：打开状态、未分配给任何人、创建超过30天、最近30天没有评论。

如果我们将业务规则隐含在仓储中，当我们需要重复使用这个业务逻辑时，问题就出现了。

举个例子，在 `Issue` 实体中希望添加一个方法 `bool IsInactive()`，用于检测 `Issue` 是否未激活状态。

看看如何实现：


```

public class Issue:AggregateRoot<Guid>,IHasCreationTime
{
    public bool IsClosed {get;private set;}
    public Guid? AssignedUserId{get;private set;}
    public DateTime CreationTime{get;private set;}
    public DateTime? LastCommentTime{get;private set;}
    //...
    public bool IsInactive(){
        var daysAgo30=DateTime.Now.Subtract(TimeSpan.FromDays(30));
        return
            //打开状态
            !IsClosed &&
            //无分配人
            AssignedUserId ==null &&
            //创建时间在30天前
            CreationTime < daysAgo30 &&
            //无评论或最后一次评论在30天前
            (LastCommentTime == null || LastCommentTime < daysAgo30 );
    }
}

```

我们不得不复制、粘贴、修改代码。如果**对未激活的Issue 规则**改变了怎么办？我们应该记得同时更新这两个地方。这是业务逻辑重复，代码的坏味道，是相当危险的。

这个问题的一个很好的解决方案就是**规约**。

规约

规约是一个命名的、**可重用的**、**可组合的**和可测试的类，用于根据业务规则**过滤领域对象**。

ABP框架提供了必要的基础设施，以轻松创建规约并在你的应用程序代码中使用。让我们把 `inactive Issue` 非活动问题业务规则实现为一个规约类。

```

using System;
using System.Linq.Expressions;
using Vollo.Abp.Specifications;

namespace IssueTracking.Issues
{
    public class InActiveIssueSpecification:Specification<Issue>
    {
        public override Expression<Func<Issue,bool>> ToExpression()
        {
            var daysAgo30=DateTime.Now.Subtract(TimeSpan.FromDays(30));
            return i =>
                //打开状态
                !i.IsClosed &&
                //无分配人
                i.AssingedUserId ==null &&
                //创建时间超过30天
                i.CreationTime < daysAgo30 &&
                //没有评论或最后评论超过30天
                (i.LastCommentTime == null || i.LastCommentTime < daysAgo30)
        }
    }
}

```

Specification<T> 基类可以帮助我们简单地创建规约类，我们可以将仓储中的表达式移到规约中。

现在，可以在 Issue 实体和 EfCoreIssueRepository 类中使用 InActiveIssueSpecification 规约。

在实体中使用规约

Specification 类提供了一个 IsSatisfiedBy 方法，如果给定的对象（实体）满足该规范，则返回 true 。我们可以重新编写 Issue.IsInactive 方法，如下所示：

```

public class Issue:AggregateRoot<Guid>,IHasCreationTime
{
    public bool IsClosed{get;private set;}
    public Guid? AssignedUserId{get;private set;}
    public DateTime CreationTiem{get;private set;}
    public DateTime? LastCommentTime{get;private set;}
    //...
    public bool IsInactive()
    {
        return new InActiveIssueSpecification().IsSatisfiedBy(this);
    }
}

```

创建一个 `InActiveIssueSpecification` 新实例，使用其 `IsSatisfiedBy` 方法，进行规约验证。

在仓储中使用规约

首先，修改仓储接口：

```
public interface IIssueRepository: IRepository<Issue, Guid>
{
    Task<List<Issue>> GetIssuesAsync(ISpecification<Issue> spec);
}
```

将方法名 `GetInActiveIssuesAsync` 改为 `GetIssuesAsync` (命名更加简洁)，接收一个规约对象参数。将规约判断的代码逻辑从仓储中移出之后，我们不再需要定义不同的方法来获取不同条件下的 `Issue`，比如：`GetAssignedIssues(...)` 获取已有分配人的问题列表，`GetLockedIssues(...)` 获取已锁定问题列表 等。

修改仓储的实现：

```
public class EfCoreIssueRepository:
    EfCoreRepository<IssueTrackingDbContext, Issue, Guid>,
    IIssueRepository
{
    public EfCoreIssueRepository(
        IDbContextProvider<IssueTrackingDbContext> dbContextProvider
    ):base(dbContextProvider)
    {}
    public async Task<List<Issue>> GetIssuesAsync(ISpecification<Issue> spec)
    {
        var dbSet = await GetDbSetAsync();
        return await dbSet
            .Where(spec.ToExpression())
            .ToListAsync();
    }
}
```

`ToExpression()` 方法返回一个**表达式**，可以直接作为 `Where` 方法的参数传递，实现实体过滤。

最后，我们将规约实例，传递给 `GetIssuesAsync` 方法：

```

public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IIssueRepository _issueRepository;
    public IssueAppService (IIssueRepository issueRepository)
    {
        _issueRepository = issueRepository;
    }
    public async Task DoItAsync()
    {
        var issues = await _issueRepository.GetIssuesAsync(
            new InActiveIssueSpecification());
    }
}

```

默认仓储

实际上，你不需要创建自定义仓储就能使用规约。标准的 `IRepository` 接口已经扩展 `IQueryable` 接口，所以你可以直接使用**标准的LINQ扩展方法**。（非常帅气!!!）

```

public class IssueAppService : ApplicationService, IIssueAppService
{
    private readonly IRepository<Issue, Guid> _issueRepository;
    public IssueAppService (IRepository<Issue, Guid> issueRepository)
    {
        _issueRepository = issueRepository;
    }
    public async Task DoItAsync()
    {
        var queryable = await _issueRepository.GetQueryableAsync();
        var issues = AsyncExecuter.ToListAsync(
            queryable.Where(new InActiveIssueSpecification()))
    }
}

```

`AsyncExecuter` 是ABP框架提供的一个工具类，用于使用异步LINQ扩展方法（比如这里的 `ToListAsync`），而不依赖于**EF Core NuGet 包**。

更多信息，请参见[Repositories](#)文档。

组合规约

规范的一个强大的地方是它们是可以**组合使用**的。假设我们有另一个规约，当问题 **Issue** 处于指定里程碑中时返回 `true`。

```

public class MilestoneSpecification : Specification<Issue>
{
    public Guid MilestoneId{get;}
    public MilestoneSpecification (Guid milestoneId)
    {
        MilestoneId = milestoneId;
    }
    public override Expression<Func<Issue,bool>> ToExpression()
    {
        return i => i.MilestoneId == MilestoneId;
    }
}

```

我们新定义了一个新的参数化规约，和前面定义 `InActiveIssueSpecification` 不同。那么如何组合两个规约，获取指定里程碑中未激活的 **Issue**（问题）呢？

```

public class IssueAppService : ApplicationService,IIssueAppService
{
    private readonly IRepository<Issue,Guid> _issueRepository;
    public IssueAppService (IRepository<Issue,Guid> issueRepository)
    {
        _issueRepository = issueRepository;
    }
    public async Task DoItAsync(Guid milestonesId)
    {
        var queryable = await _issueRepository.GetQueryableAsync();
        var issues = AsyncExecuter.ToListAsync(
            queryable.Where(new InActiveIssueSpecification()
                .Add(new MilestoneSpecification(milestoneId))
                .ToExpression()
            )
        );
    }
}

```

示例中使用 `Add` 扩展方法组合规约，还有更多的扩展方法，比如：`Or(...)` `AndNot(...)`。

参看：规约文档，获取更多使用帮助。

领域服务

领域服务实现领域逻辑，它：

- 依赖于**服务**和**仓储**。
- 需要多个聚合，以实现单个聚合无法处理的逻辑。

领域服务与领域对象一起使用，其方法可以获取和返回**实体**、**值对象**、原始类型等。然而，它并不获取/返回**DTOs**，DTOs属于应用层。

示例：将问题分配给用户

回想一下，我们之前是如何实现将问题分配给用户的

```
public class Issue:AggregateRoot<Guid>
{
    //..
    //问题关联的用户ID
    public Guid? AssignedUserId{get;private set;}
    //分配方法
    public async Task AssignToAsync(AppUser user,IUserIssueService userIssueService)
    {
        var openIssueCount = await userIssueService.GetOpenIssueCountAsync(user.Id);
        if(openIssueCount >=3 )
        {
            throw new BusinessException("IssueTracking:CanNotOpenLockedIssue");
        }
        AssignedUserId=user.Id;
    }
    public void CleanAssignment()
    {
        AssignedUserId=null;
    }
}
```

现在，我们将逻辑迁移到领域服务中。首先，修改 Issue 类：

```
public class Issue:AggregateRoot<Guid>
{
    //...
    public Guid? AssignedUserId{get;internal set;}
}
```

- 在聚合中移除 AssignToAsync 方法（因为需要在对应的领域服务中实现该方法。）
- 将 AssignedUserId 属性设置器从私有改为内部 internal，以允许从领域服务中设置它。

接下来，创建一个领域服务 IssueManager 定义方法 AssignToAsync 将指定 Issue 分配给指定用户。

```

public class IssueManager:DomainService
{
    private readonly IRepository<Issue,Guid> _issueRepository;
    public IssueManager(IRepository<Issue,Guid> issueRepository)
    {
        _issueRepository=issueRepository;
    }
    public async Task AssignToAsync(Issue issue,AppUser user)
    {
        //获取关联用户处于打开状态问题的数量
        var openIssueCount=await _issueRepository.CountAsync(
            i=>i.AssingedUserId==user.Id && !i.IsClosed
        );
        //超过3个，则抛出异常
        if(openIssueCount>=3)
        {
            throw new BusinessException("IssueTracking:ConcurrentOpenIssueLimit");
        }
        issue.AssignedUserId=user.Id;
    }
}

```

IssueManager 在构造函数中注入需要的仓储，用于查询分配给用户处于打开状态的Issue。

建议使用 Manager 后缀命名来命名领域服务。

这种设计的唯一问题是： Issue.AssignedUserId 现在是 public ，可以在任何外部类中设置。然而，它不应该是公共的，访问范围应该是程序集内部 internal ，只有在同一个程序集（ IssueTracking.Domain ）项目中才可以调用。

这个例子的解决方案就是如此，我们认为这很合理：

- 领域层开发者在使用 IssueManager 时，已经熟知领域规则。
- 应用层开发者强制使用 IssueManager，因此无法直接修改实体。

以上我们展示了将问题分配给用户的两种实现方式，两种方式权衡之下，我们更加推荐**当业务逻辑需要与外部服务协同工作时，创建领域服务**。

如果没有一个充分的理由，我们认为没有必要去为领域服务创建接口，比如：为 IssueManager 创建 IIssueManger 接口。

应用服务

应用服务是无状态服务，实现应用程序**用例**。一个应用服务通常使用领域对象实现用例，获取或返回数据传输对象DTOs，被展示层调用。

应用服务通用原则：

- 实现特定用例的应用逻辑，不能在应用服务中实现领域逻辑（需要理清应用逻辑和领域逻辑二者的区别）。
- **应用服务方法不能返回实体**，因为这样会打破领域层的封装性，始终只返回DTO。

示例：分配问题给用户

```
using System;
using System.Threading.Tasks;
using IssueTracking.Users;
using Microsoft.AspNetCore.Authorization;
using Volo.Abp.Application.Services;
using Volo.Abp.Domain.Repositories;

namespace IssueTracking.Issues
{
    public class IssueAppService : ApplicationService.IIssueAppService
    {
        private readonly IssueManager _issueManager;
        private readonly IRepository<Issue, Guid> _issueRepository;
        private readonly IRepository<AppUser, Guid> _userRepository;

        public IssueAppService(
            IssueManager issueManager,
            IRepository<Issue, Guid> issueRepository,
            IRepository<AppUser, Guid> userRepository
        )
        {
            _issueManager = issueManager;
            _issueRepository = issueRepository;
            _userRepository = userRepository;
        }

        [Authorize]
        public async Task AssignAsync(IssueAssignDto input)
        {
            var issue = await _issueRepository.GetAsync(input.IssueId);
            var user = await _userRepository.GetAsync(input.UserId);
            await _issueManager.AssignToAsync(issue, user);
            await _issueRepository.UpdateAsync(issue); // 没有对issue做任何修改，为什么要更新？在IssueManager
        }
    }
}
```

一个应用服务方法通常有三个步骤：

- 从数据库获取关联的领域对象
- 使用领域对象（领域服务、实体等）执行业务逻辑

- 在数据库中更新实体（如果已修改）

当时使用EF Core时，最后的 Update 更新操作并不是必须的，应为有 状态变更跟踪。但是建议显式调用，适配其他数据库提供程序。

示例中 IssueAssignDto 是一个简单的 DTO 类：

```
using System;
namespace IssueTracking.Issues
{
    public class IssueAssignDto
    {
        public Guid IssueId{get;set;}
        public Guid UserId{get;set;}
    }
}
```

数据传输对象

DTO 是简单对象，用于在应用层和展示层传递状态数据。所以，应用服务方法返回 DTO。

DTO原则和最佳实践：

- DTO应该可序列化，因为大多数时候，需要网络传输。
- 应该有一个无参构造函数
- 不能包含任何业务逻辑
- 不能继承或引用实体

输入DTO和**输出DTO**在本质上不同：一个用于给应用服务方法传递参数，一个作为应用服务方法的返回值，根据业务需要区别对待。

输入DTO最佳实践

不要在输入DTO中定义不使用的属性

只定义需要用的属性，否则，无用的属性只会让客户端在使用应用服务方法时感到困惑。当然可以定义**可选属性**，但是确保当客户端在使用时，不应该影响到用例的工作方式。

这条规则看起来没什么必要，谁会为方法仓储（输入DTO）添加不使用的属性呢？但是，它经常发生，尤其是当你想**重用**输入DTO对象时，会将多个DTO属性放在一个DTO对象中。

不要重用输入DTO

为每个用例（应用服务方法）定义特定的输入DTO，否则，在某些情况下不会添加一些不被使用的属性，这就违反了上面定义的规则。

有时候，在两个不同的用例中使用相同的DTO似乎很有吸引力，因为他们如此相似。甚至，当前是一模一样，可能后面随着业务变化才会有可能不同，此时也应该不要重用输入DTO。因为**和用例间的耦合相比，代码复制可能是更好的做法。**

重用DTO的另一种方式是：DTO继承，这同样会产生上面描述的问题。

示例：用户应用服务

```
public interface IUserAppService:IApplicationService
{
    Task CreateAsync(UserDto input);
    Task UpdateAsync(UserDto input);
    Task ChangePasswordAsync(UserDto input);
}
```

IUserAppService 在所有方法（用例）使用 UserDto 作为输入DTO， UserDto 定义如下：

```
public class UserDto
{
    public Guid Id{get;set;}
    public string UserName{get;set;}
    public string Email{get;set;}
    public string Password{get;set;}
    public DateTime CreationTime{get;set;}
}
```

- Id 在 Create 方法中不被使用，因为 Id 由服务器生成。
- Password 在 Update 方法中不使用，因为有修改密码的单独方法。
- CreationTime 未被使用，且不应该由客户端发送给服务端，应该在服务端设置创建时间。

正确的实现，如下：

```
public interface IUserAppService:IApplicationService
{
    Task CreateAsync(UserCreationDto input);
    Task UpdateAsync(UserUpdateDto input);
    Task ChangePasswordAsync(UserChangePasswordDto input);
}
```

然后定义对应的DTO类：

```
public class UserCreationDto
{
    public string UserName {get;set;}
    public string Email{get;set;}
    public string Password{get;set;}
}

public class UserUpdateDto
{
    public Guid Id{get;set;}
    public string UserName{get;set;}
    public string Email{get;set;}
}

public class UserChangePasswordDto
{
    public Guid Id{get;set;}
    public string Password{get;set;}
}
```

尽管需要编写更多的代码，但是这是一种更易维护的方法。

****特殊情况：**举个例子，如果你有一个报表页，页面中有多个过滤条件，对应多个应用服务方法（显示报表、导出Excel、导出CSV），此时应该使用相同的输入DTO参数，返回不同的结果。因为当页面过滤条件改变时，修改一个DTO而对整个页面对应的应用服务方法参数生效。

输入DTO中验证逻辑

- 仅在DTO内部执行**简单验证**，使用**数据注解特性**或实现 `IValidatableObject` 接口
- **不要执行领域验证**，举个例子，不要在DTO中检测用户名是否唯一的验证。

示例：使用数据注解特性

```
using System.ComponentModel.DataAnnotations;

namespace IssueTracking.Users
{
    public class UserCreationDto
    {
        [Required]
        [StringLength(UserConsts.MaxUserNameLength)]
        public string Username {get;set;}

        [Required]
        [EmailAddress]
        [StringLength(UserConsts.MaxEmailLength)]
        public string Email{get;set;}
        [Required]
        [StringLength(UserConsts.MaxEmailLength,MinimumLength=UserConsts.MinPasswordLength)]
        public string Password{get;set;}
    }
}
```

ABP框架自动验证输入DTO，验证失败则抛出 `AbpValidationException` 异常，返回 400 HTTP 状态码。

某些开发者认为将验证规则和DTO类分离可能会更好。我们认为声明式（数据注解）是实用的，不会导致任何设计问题。当然，ABP支持 `FluentValidation`集成。

输出DTO最佳实践

- 保持输出DTO数量最小，尽可能重用，但是不能将输入DTO作为输出DTO使用。
- 输出DTO可以包含比用例需要的更多属性
- `Create` 和 `Update` 方法中返回DTO

以上建议的主要原因是：

- 使客户端代码易于开发和扩展
 - 在客户端端处理**不同但相似的**DTO容易混淆
 - 输入DTO中的更多属性可能未来会在UI/客户端中被使用，返回实体的所有属性(已经考虑过安全性和特殊情况)使客户端代码易于改进，而不需要修改后端代码。
 - 如果是通过API暴露给第三方客户端，避免不同需求返回不同DTO
- 使服务端代码易于开发和扩展
 - 更少的类，易于理解和维护
 - 可以重用实体到DTO（`AutoMapper`）的对象映射代码
 - 不同方法返回相同类型，使**添加新方法**变得简单明了。

示例：从不同方法返回不同DTO

```
public interface IUserAppService:IApplicationService
{
    UserDto Get(Guid id);
    List<UserNameAndEmailDto> GetUserNameAndEmail(Guid id);
    List<string> GetRoles(Guid id);
    List<UserListDto> GetList();
    UserCreateResultDto Create(UserCreationDto input);
    UserUpdateResultDto Update(UserUpdateDto input);
}
```

示例中没有使用异步方法，在实际开发时应该是异步方法。

上面的示例代码中，为每个方法返回不同DTO类型，这样会导致我们需要处理非常多的数据查询，映射实体到DTO的重复代码。

按照以下方式定义就简单多了：

```
public interface IUserAppService:IApplicationService
{
    UserDto Get(Guid id);
    List<UserDto> GetList();
    UserDto Create(UserCreationDto input);
    UserDto Update(UserUpdateDto input);
}
```

使用一个输出DTO：

```
public class UserDto
{
    public Guid Id{get;set;}
    public string UserName{get;set;}
    public string Email{get;set;}
    public DateTime CreationTime{get;set;}
    public List<string> Roles{get;set;}
}
```

- 移除 GetUserNameAndEmail 和 GetRoles 方法，因为 Get 方法已经返回足够需要的信息。
- GetList 返回对象与 Get 相同
- Create 和 Update 同样返回 UserDto

由此可见，返回相同DTO更加简洁。

为什么创建或更新之后要返回DTO? 想象一个用例场景，在页面中显示**表格数据**，当更新之后，获取返回对象，并对表格数据源进行更新，这样就不需要再次调用 `GetList` 方法，这是我们建议在 `Create` 和 `Update` 方法中返回 DTO 的原因。

讨论

以上关于输出DTO的建议,并不适用所有场景。

出于性能考虑，这些建议可以被忽略，特别是当存在大型数据集返回结果时，或者用户界面需要发起很多并发请求时，此时应该创建特定的输出DTO，只包含尽可能少的信息。

可维护性和性能，需要开发者权衡，上面的建议适用于性能损失可忽略不计的应用。

对象映射

自动对象映射是一个非常有用的工具，两个对象的属性相同或相似，将一个对象的值复制给另一个对象。

DTO和实体类通常具有相同或相似的属性，通常需要根据实体和业务需求来创建DTO对象。ABP框架对象映射基于 `AutoMapper`，相比手动赋值，效率更高。

- 仅对**实体到输出DTO**使用自动对象映射。
- **输入DTO到实体**，**不适用**自动对象映射。

不使用**输入DTO到实体**自动映射的原因：

1. 实体类通常有构造函数，接收参数并在创建时，进行参数验证。自动对象映射操作通常需要**无参构造函数**创建对象。
2. 实体属性设置器大多是私有的，应该**使用方法设置属性值**。
3. 通常需要**仔细验证和处理**用户/客户端输入，而不是盲目地映射到实体属性。

虽然其中一些问题可以通过映射配置来解决（例如，`AutoMapper`允许定义自定义映射规则），但它使你的**业务逻辑隐含/隐藏，并与基础设施紧密耦合**。我们认为业务代码应该是明确的、清晰的、容易理解的。

实体创建和更新

本节将演示一些用例，并讨论替代场景。

为了更好地帮助大家在使用ABP框架实践DDD开发过程中，遇到问题时，讨论、交流！创建 **ABP Framework 研习社（QQ群：726299208）**

实体创建

创建实体（或聚合根类）的实例对象是实体生命周期的开始。在聚合和聚合根最佳实践和规则章节部分建议：为**实体类**创建一个**主构造函数**，确保实体的有效性。所以，无论何时我们在需要创建该实体的实例时，都统一使用该构造函数。

我们来看看 Issue 聚合根类：

```
public class Issue:AggregateRoot<Guid>
{
    public Guid RepositoryId{get;private set;}
    public string Title{get;private set;}
    public string Text{get;set;}
    public Guid? AssignedUserId{get;internal set;}

    public Issue(
        Guid id,
        Guid repositoryId,
        string title,
        string text=null
    ): base(id)
    {
        RepositoryId=repositoryId;
        Title=Check.NotNullOrWhiteSpace(title,nameof(title));
        Text=text; //允许为空或null
    }

    private Issue(){}

    public void SetTitle(string title)
    {
        Title=Check.NotNullOrWhiteSpace(title,nameof(title));
    }
    //...
}
```

- 该类通过构造函数确保创建实体的有效性
- 如果需要在创建后修改 Title 属性，使用 SetTitle 方法，确保 Title 是有效状态。
- 如果要将当前 Issue 分配给一个用户，需要使用 IssueManager。
- Text属性有一个公共设置器，因为其接收 null 或 任何内容，而不需要任何验证。在构造函数中是可选参数。

让我们看看在应用服务方法中如何创建一个 Issue：

```

public class IssueAppService :ApplicationService.IIssueAppService
{
    private readonly IssueManager _issueManager;
    private readonly IRepository<Issue,Guid> _issueRepository;
    private readonly IRepository<AppUser,Guid> _userRepository;

    public IssueAppService(
        IssueManager issueManager,
        IRepository<Issue,Guid> issueRepository,
        IRepository<AppUser,Guid> userRepository
    )
    {
        _issueManager=issueManager;
        _issueRepository=issueRepository;
        _userRepository=userRepository;
    }

    public async Task<IssueDto> CreateAsync(IssueCreationDto input)
    {
        //创建有效实体
        var issue=new Issue(
            GuidGenerator.Create(),
            input.RepositoryId,
            input.Title,
            input.Text
        );
        //应用其他领域操作
        if(input.AssignedUserId.HasValue)
        {
            var user =await _userRepository.GetAsync(input.AssignedUserId.Value);
            await _issueManager.AssignToAsync(issue,user);
        }
        //保存
        await _issueRepository.InsertAsync(issue);
        //返回DTO
        return ObjectMapper.Map<Issue,IssueDto>(issue);
    }
}

```

CreateAsync 方法:

- 使用 Issue 构造函数创建, 使用 IGuidGenerator 服务生成 Id, 并没有使用自动对象映射。
- 在 Issue 对象创建同时, 需要指定给某个用户, 使用 IssueManager 来执行必要的业务逻辑。
- 保存实体到数据库
- 最后使用 IMapper 从新创建的 Issue 实体自动映射到 IssueDto 对象, 并返回。

实体创建时应用领域规则

Issue 实体在创建时并没有业务规则，只是在构造函数中进行了常规验证。当然，我们可能需要处理在实体创建时进行其他业务规则检测的场景。

举个例子，在创建 Issue 时，不允许创建已经存在的 Title，如何实现该规则？不应该在应用服务中实现该规则，因为它是**领域业务规则**，所以应该在领域服务 IssueManager 中实现。所以，我们需要强制应用层使用 IssueManager 来创建新的 Issue 实例。

现在，我们要将 Issue 的构造函数从 public 设置为 internal：

```
public class Issue:AggregateRoot<Guid>
{
    //...
    internal Issue(
        Guid id,
        Guid repositoryId,
        string title,
        string text=null
    ): base(id)
    {
        RepositoryId=repositoryId;
        Title=Check.NotNullOrWhiteSpace(title,nameof(title));
        Text=text;//allow empty/null
    }
    //...
}
```

这样就阻止了应用服务直接调用实体构造函数创建实体，而必须使用领域服务 IssueManager 创建，添加 CreateAsync 方法：

```

public class IssueManager:DomainService
{
    private readonly IRepository<Issue,Guid> _issueRepository;
    public IssueManager(IRepository<Issue,Guid> issueRepository)
    {
        _issueRepository=issueRepository;
    }
    public async Task<Issue> CreateAsync(
        Guid repositoryId,
        string title,
        string text=null
    )
    {
        if(await _issueRepository.AnyAsync(i=>i.Title==title))
        {
            throw new BusinessException("IssueTracking:IssueWithSameTitleExists");
        }
        retur new Issue(
            GuidGenerator.Create(),
            repositoryId,
            title,
            text
        );
    }
}

```

- CreateAsync 方法检测是否已存在相同 Title 的 Issue，存在则抛出异常；如果标题没有重复的，则创建并返回一个新的 Issue 对象。

IssueAppService 应用服务中需要修改，以调用 IssueManager.CreateAsync 方法，创建实体。

```

public class IssueAppService :ApplicationService.IIssueAppService
{
    private readonly IssueManager _issueManager;
    private readonly IRepository<Issue,Guid> _issueRepository;
    private readonly IRepository<AppUser,Guid> _userRepository;

    public IssueAppService(
        IssueManager issueManager,
        IRepository<Issue,Guid> issueRepository,
        IRepository<AppUser,Guid> userRepository
    )
    {
        _issueManager=issueManager;
        _issueRepository=issueRepository;
        _userRepository=userRepository;
    }

    public async Task<IssueDto> CreateAsync(IssueCreationDto input)
    {
        //创建有效实体
        var issue=await _issueManager.CreateAsync(
            input.RepositoryId,
            input.Title,
            input.Text
        );
        //应用其他领域操作
        if(input.AssignedUserId.HasValue)
        {
            var user =await _userRepository.GetAsync(input.AssignedUserId.Value);
            await _issueManager.AssignToAsync(issue,user);
        }
        //保存
        await _issueRepository.InsertAsync(issue);
        //返回DTO
        return ObjectMapper.Map<Issue,IssueDto>(issue);
    }
}

public class IssueCreationDto
{
    public Guid RepositoryId{get;set;}
    [Required]
    public string Title {get;set;}
    public Guid? AssignedUserId{get;set;}
    public string Text {get;set;}
}

```

讨论：为什么不在 **IssueManager** 中保存 **Issue** 数据？

我们认为这是**应用服务**的责任。因为，应用服务可能需要在保存 Issue 对象之前，对其进行额外的更改操作。如果**领域服务**保存，那么保存操作会重复的。

- 因为触发两次数据库操作，会导致性能损失。
- 需要对两个操作使用显式的数据库事务，确保数据一致性。
- 如果额外的操作，因为业务规则而取消了实体的创建，应该在数据库中回滚事务，取消所有操作。

当你检查 `IssueAppService` 时，你会发现在 `IssueManager.CreateAsync` 中不把 Issue 保存到数据库的好处。否则，我们将需要在 `IssueManager` 中执行一次插入操作，在 `IssueAppService` 赋值之后执行一次更新操作。

讨论：**为什么标题重复检测不在应用服务中实现？**

简单地说“因为它是**核心领域逻辑**，应该在领域层中实现。”这会引发一个新的问题“如何判断它是核心领域逻辑，而不是应用逻辑？”（我们会在下一个章节重点讨论）

在当前示例中，一个简单的问题有助于我们更加清晰地思考：“如果我们还有另一种方式来创建 Issue，是否还需要应用相同的规则？这个规则是否应该一直被实现？”，你可能会想“为什么会有第二种方式创建 Issue 呢？”，在真实的环境下，可能碰到：

- 应用程序终端用户通过标准UI创建Issue。
- 集成到办公应用后台，被你的员工使用，提供一种方式创建 Issue。
- 通过 HTTP API，提供给第三方终端使用，创建 Issue。
- 运行后台任务，当检测到问题时自动提交 Issue，这种方式不涉及任何用户交互。
- 提供一键转换操作，将讨论的问题转换成 Issue 。

我们还可以给出更多的例子，这些例子通过不同的应用服务方法来实现，但是他们都应该遵循规则：新的 Issue 不能与现有 Issue 同名。由此，我们判定这个规则是**核心领域规则**，应该放在领域层实现，而不是放在应用服务的方法中去重复定义。

实体更新

实体一旦被创建，通过用例来更新和维护，直到被系统删除。有不同类型的用例直接或间接地修改实体。

本章节，我们将讨论实体的更新操作，修改 Issue 的多个属性。

从 Update DTO 开始：

```

public class UpdateIssueDto
{
    [Required]
    public string Title {get;set;}
    public string Text{get;set;}
    public Guid? AssignedUserId{get;set;}
}

```

对比 IssueCreationDto 会发现没有 RepositoryId 属性。因为系统设定为不允许在不同 Repository 中移动 Issue 。只有 Title 属性是必须的，其他属性都是可选的。

让我们看看 IssueAppService 中 Update 实现：

```

public class IssueAppService :ApplicationService.IIssueAppService
{
    private readonly IssueManager _issueManager;
    private readonly IRepository<Issue,Guid> _issueRepository;
    private readonly IRepository<AppUser,Guid> _userRepository;

    public IssueAppService(
        IssueManager issueManager,
        IRepository<Issue,Guid> issueRepository,
        IRepository<AppUser,Guid> userRepository
    )
    {
        _issueManager=issueManager;
        _issueRepository=issueRepository;
        _userRepository=userRepository;
    }
    public async Task<IssueDto> UpdateAsync(Guid id,UpdateIssueDto input)
    {
        //从数据库中获取实体
        var issue = await _issueRepository.GetAsync(id);
        //设置Title
        await _issueManager.ChangeTitleAsync(issue,input.Title);
        //设置所属用户
        if(input.AssignedUserId.HasValue)
        {
            var user=await _userRepository.GetAsync(input.AssignedUserId.Value);
            await _issueManager.AssignToAsync(issue,user);
        }
        issue.Text=input.Text;
        //保存更新
        await _issueRepository.UpdateAsync(issue);
        //返回DTO
        return ObjectMapper.Map<Issue,IssueDto>(issue);
    }
}

```

- `UpdateAsync` 方法接收单独的参数 `id` , `UpdateIssueDto` 中不包含该字段。这个设计决策, 有助于ABP框架在将该服务方法**自动暴露**为HTTP API端点时, 正确定义HTTP路由, 这与DDD没有关系。
- 首先从数据库**获取** `Issue` 实体
- 使用 `IssueManager.ChangeTitleAsync` 代替直接调用 `Issue.SetTitle(...)` , 因为需要实现 `Title` 同名检测的规则, 与实体创建时保持一致。
- 使用 `IssueManager.AssignToAsync` 方法, 设置所属用户。
- 直接设置 `Issue.Text` , 该字段没有业务规则约束, 如果后期需要, 可进行重构。
- 保存更改到数据库, 保存实体更改是应用服务方法的职责。如果 `IssueManager` 内部在 `ChangeTitleAsync` 和 `AssignToAsync` 方法中保存, 则会存在双重数据库操作的问题。
- 最后, 使用 `IObjectMapper` 映射 `Issue` 实体为 `IssueDto` , 作为方法返回值返回。

如前所述, 我们需要对 `Issue` 和 `IssueManager` 类做一些修改。

首先, 在 `Issue` 类中将 `SetTitle` 方法设置为 `internal`

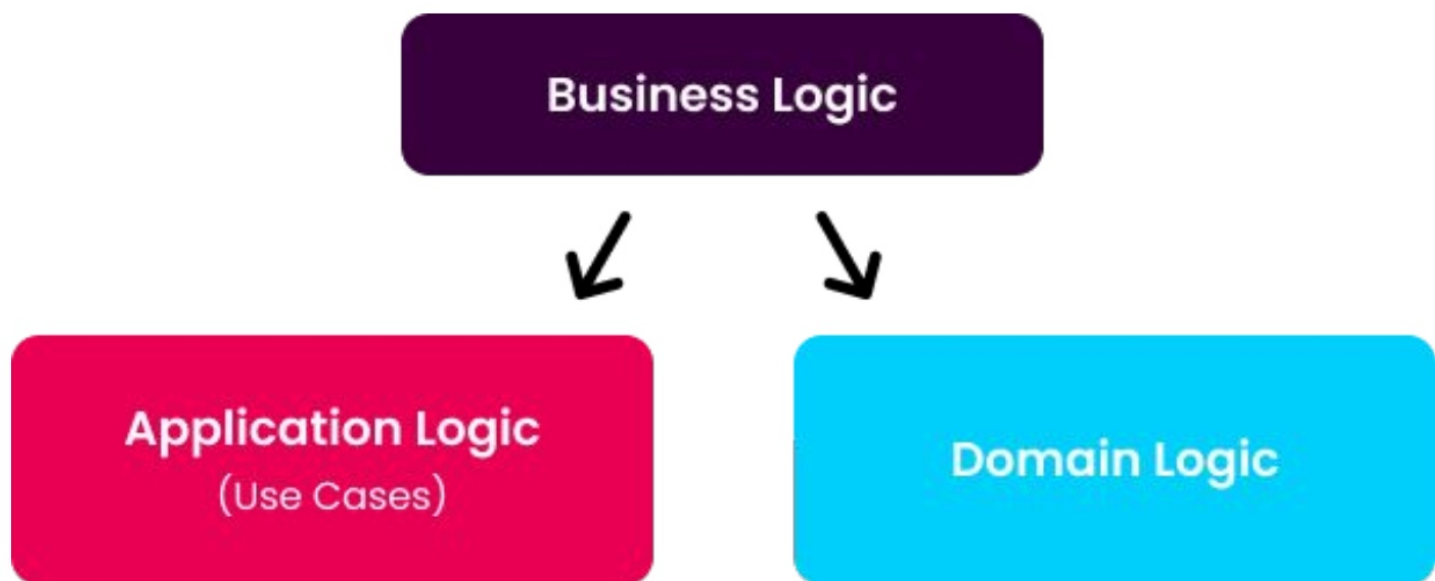
```
internal void SetTitle(string title)
{
    Title=Check.NotNullOrWhiteSpace(title,nameof(title));
}
```

在 `IssueManager` 中添加新方法:

```
public async Task ChangeTitleAsync(Issue issue,string title)
{
    if(issue.Title==title)
    {
        return;
    }
    if(await _issueRepository.AnyAsync(i=>i.Title==title))
    {
        throw new BusinessException("IssueTracking:IssueWithSameTitleExists");
    }
    issue.SetTitle(title);
}
```

领域逻辑和应用逻辑

正如前面提到的, 领域驱动设计中的**业务逻辑**拆分为两部分: **领域逻辑**和**应用逻辑**。



领域逻辑由系统的核心领域规则组成，而应用程序逻辑实现特定于应用程序的用例。

虽然定义很清楚，但实现可能并不容易，常常无法决定哪些代码应该放在应用层，哪些代码应该放在领域层。本节试图解释这些差异。

为了更好地帮助大家在使用ABP框架实践DDD开发过程中，遇到问题时，讨论、交流！创建 **ABP Framework 研习社（QQ群：726299208）**

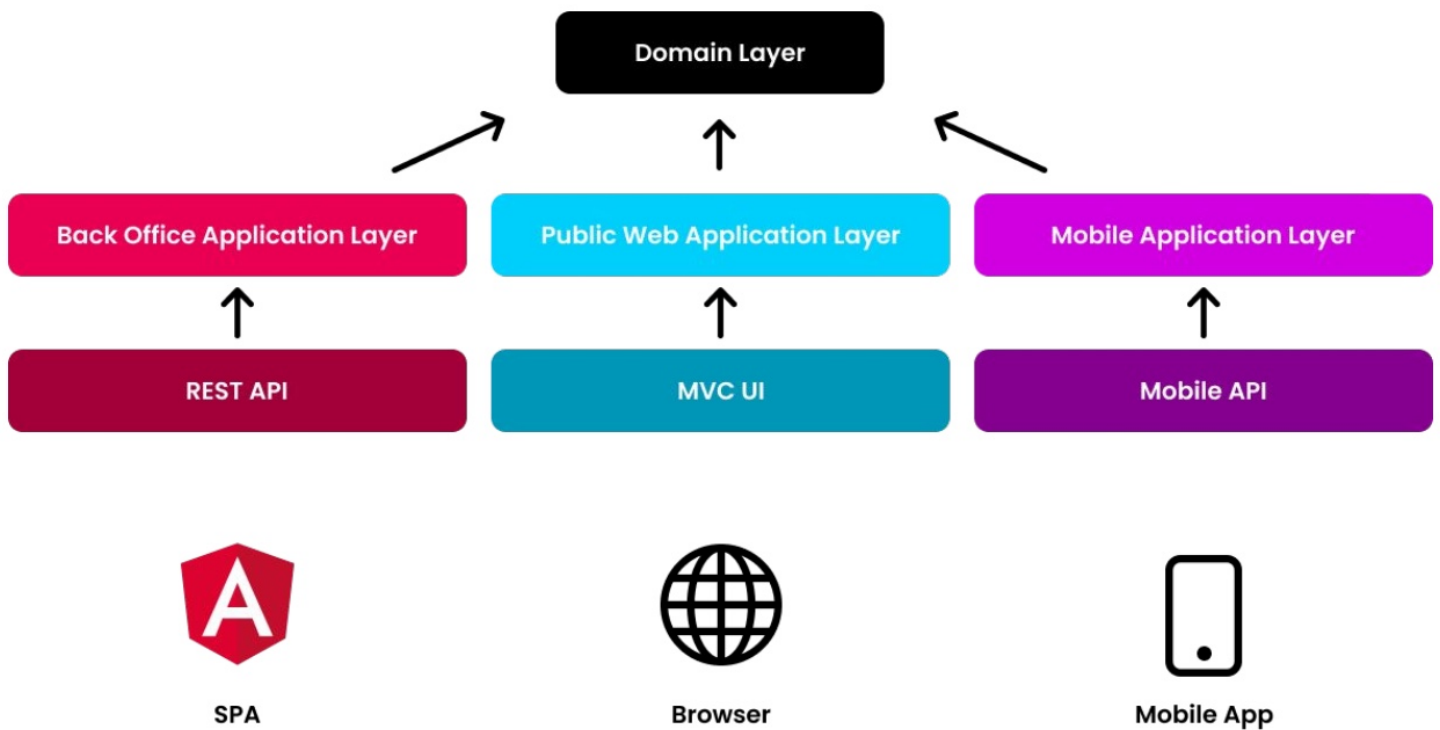
专注 **ABP Framework** 技术分析、讨论交流、资料共享、示例源码等，欢迎加入！

多应用层

当你的系统很大时，DDD有助于处理复杂性问题。特别是，在一个领域中开发多个应用，那么**领域逻辑与应用逻辑的分离**就变得更加重要。

假设你正在构建一个有多个应用程序的系统：

- 一个**Web应用程序**，使用 ASP .NET Core MVC，展示产品给用户。浏览产品，不需要进行身份验证；只有当用户执行某些操作时，比如向购物车中添加产品，才会要求登陆。
- 一个**后台管理应用程序**，使用Angular UI+ REST APIs构建。此应用由公司办公人员做系统管理，比如：编辑产品描述。
- 一个**移动应用程序**，和 Web应用程序 相比UI更加简单，通过REST APIs或其他技术（如：TCP/Socket）与服务器通信。



每一个应用都需要解决不同的需求，实现不同用例（应用服务方法），不同DTO，不同验证和授权规则等等。

将所有这些逻辑混合到一个应用层中，将使你的服务包含太多的判断条件和复杂的业务逻辑，使代码更难开发、维护和测试，并导致潜在的Bug。

如果你有单个领域关联多个应用程序：

- 为每个应用程序或客户端创建单独的应用层，在这些单独层中实现特定于应用的业务逻辑。
- 使用**单个领域层共享核心领域逻辑**。

这样的设计使得区分**领域逻辑**和**应用逻辑**变得更加重要。

为了更清楚地实现，可以为每种应用程序类型创建不同的项目(`.csproj`)。

例如：

- 后台管理应用创建 `IssueTracker.Admin.Application` 和 `IssueTracker.Admin.Application.Contracts` 项目
- WEB应用创建 `IssueTracker.Public.Application` 和 `IssueTracker.Public.Application.Contracts`
- 移动应用创建 `IssueTracker.Mobile.Application` 和 `IssueTracker.Mobile.Application.Contracts`

示例：正确区分应用逻辑和领域逻辑

本节包含一些应用服务和领域服务示例，讨论如何决定在这些服务中放置业务逻辑。

示例：在领域服务中创建 Organization（组织）

```
public class OrganizationManager:DomainService
{
    private readonly IRepository<Organization> _organizationRepository;
    private readonly ICurrentUser _currentUser;
    private readonly IAuthorizationService _authorizationService;
    private readonly IEmailSender _emailSender;

    public OrganizationManager(
        IRepository<Organization> organizationRepository,
        ICurrentUser currentUser,
        IAuthorizationService authorizationService,
        IEmailSender emailSender
    )
    {
        _organizationRepository=organizationRepository;
        _currentUser=currentUser;
        _authorizationService=authorizationService;
        _emailSender=emailSender;
    }
    //创建组织
    public async Task<Organization> CreateAsync(string name)
    {
        //检测是否存在同名组织，存在则抛出异常。
        if(await _organizationRepository.AnyAsync(x=>x.Name==name))
        {
            throw new BusinessException("IssueTracking:DuplicateOrganizationName");
        }
        //检测是否拥有创建权限
        await _authorizationService.CheckAsync("OrganizationCreationPermission");
        //记录日志
        Logger.LogDebug($"Creating organization {name} by {_currentUser.UserName}");
        //创建组织实例
        var organization = new Organization();
        //发送提醒邮件
        await _emailSender.SendAsync(
            "systemadmin@issuetracking.com",
            "新组织",
            "新组织名称: "+name
        );
        //返回组织实例
        return organization;
    }
}
```

让我们一步一步来分析 CreateAsync 方法中的代码是否都应该放在领域服务中：

- 正确：**组织名重复检测**，存在重复名称则抛出异常。该检测与核心领域规则相关，不允许重名。
- 错误：**领域服务不应该进行权限验证**。权限验证应该放在应用层。
- 错误：**记录日志包含当前用户的用户名**。领域服务不应该依赖当前用户，当前用户（Session）应该是展示层或应用层中的相关概念。
- 错误：**创建新组织发送邮件**，我们仍然认为这是业务逻辑。可以能会根据用例来创建不同类型邮件。

示例：在应用层创建新组织

```

public class OrganizationAppService:ApplicationService
{
    private readonly OrganizationManager _organizationManager;
    private readonly IPaymentService _paymentService;
    private readonly IEmailSender _emailSender;

    public OrganizaitonAppService(
        OrganizationManager organizationManager,
        IPaymentService paymentService,
        IEmailSender emailSender
    )
    {
        _organizationManager=organizationManager;
        _paymentService=paymentService;
        _emailSender=emailSender;
    }
    [UnitOfWork]
    [Authorize("OrganizationCreationPermission")]
    public async Task<Organization> CreateAsync(CreateOrganizationDto input)
    {
        //支付组织费用
        await _paymentService.ChargeAsync(
            CurrentUser.Id,
            GetOrganizationPrice()
        );
        //创建组织实例
        var organization = await _organizationManager.CreateAsync(input.Name);
        //保存组织到数据库
        await _organizationManager.InsertAsync(organization);
        //发送提醒邮件
        await _emailSender.SendAsync(
            "systemadmin@issuetracking.com",
            "新组织",
            "新组织名称: "+name
        );
        //返回实例
        return organization;
    }
    private double GetOrganizationPrice()
    {
        return 42; //Gets form somewhere...
    }
}

```

让我们看看 CreateAsync 方法，一步一步讨论其中的代码是否应该放在应用服务：

- 正确：应用服务方法应该是工作单元，ABP框架工作单元系统自动实现，可以不用添加 [UnitOfWork] 特性。
- 正确：权限验证应该放在应用层，可以使用 [Authorize] 特性。

- 正确：在我们的业务逻辑中创建组织是付费服务，当前操作调用基础设施服务进行支付操作。
- 正确：应用服务方法负责保存变更到数据库。
- 正确：给系统管理员发送邮件通知。
- 错误：不能返回实体，应该返回DTO。

讨论：为什么我们不应该将支付逻辑放在领域服务中？

你可能想知道为什么付款代码不在 `OrganizationManager` 里面。这是一件很重要的事情，我们绝不希望付款出错。

然而，**业务的重要性并不意味着要将其视为核心业务逻辑**。我们可能有其他的支付用例，在这些用例中，我们不收取费用来创建一个新的组织。

例如：

- 后台办公系统中的管理员用户可以创建新组织，不用考虑支付。
- 系统数据导入、整合、同步，也可能需要在没有任何支付操作的情况下，创建组织。

如您所见，支付不是创建一个有效组织的必要操作。它是一个特定于用例的应用逻辑。

示例：**CRUD**操作

```
public class IssueAppService
{
    private readonly IssueManager _issueManager;
    public IssueAppService(IssueManager issueManager)
    {
        _issueManager=issueManager;
    }
    public async Task<IssueDto> GetAsync(Guid id)
    {
        return await _issueManager.GetAsync(id);
    }
    public async Task CreateAsync(IssueCreationDto input)
    {
        await _issueManager.CreateAsync(input);
    }
    public async Task UpdateAsync(UpdateIssueDto input)
    {
        await _issueManager.UpdateAsync(input);
    }
    public async Task DeleteAsync(Guid id)
    {
        await _issueManager.DeleteAsync(id);
    }
}
```

应用服务并没有做任何事情，而是委托给领域服务来处理。只接收DTO参数，并传递给 IssueManger。

- 不要创建只实现简单 CRUD 操作的领域服务方法，而不带任何领域逻辑。
- 不要传递 DTO 给领域服务，或领域服务方法返回 DTO。

应用服务可以直接使用仓储，实现查询、创建、更新或删除数据，除非执行这些操作时需要处理领域逻辑，这种情况下，创建领域服务方法，但只针对那些真正需要的方法。

不要因为将来可能会需要这些CRUD领域服务方法，就去提前创建这些方法! 当需要时再去创建，并重构现有的代码。由于抽象了应用层，重构领域层不会影响到UI层和其他客户端。