## TODO

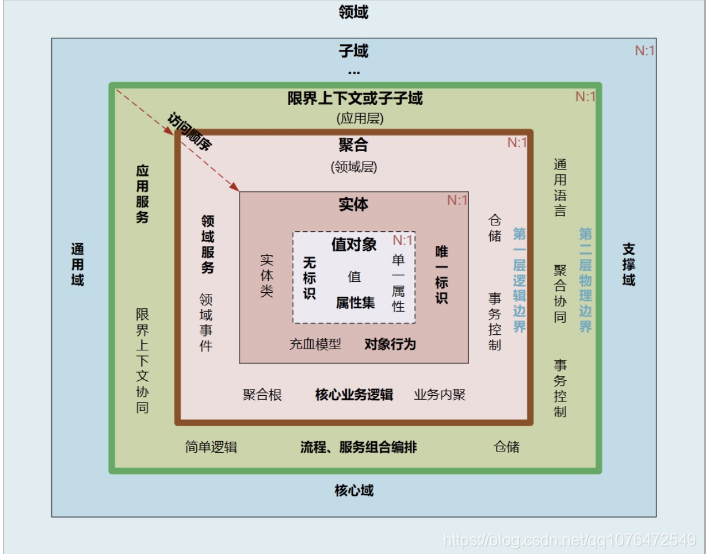
https://blog.csdn.net/weixin\_38370441/article/details/115620797?spm=1001.2101.3001.6650.3&utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-3-115620797-blog-115654613.pc\_relevant\_multi\_platform\_whitelistv1\_exp2&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-3-115620797-blog-115654613.pc\_relevant\_multi\_platform\_whitelistv1\_exp2&utm\_relevant\_index=6



梳理从1到最后所有笔记

## 基础篇

### 核心知识体系



### 设计思路

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

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

3.用三步来划定领域模型和微服务的边界：

一步： 在事件风暴中梳理业务过程中的用户操作、事件以及外部依赖关系等，根据这些要素梳理出领域实体等领域对象。

第二步： 务紧密相关的实体进行组合形成聚合，同时确定聚合中的聚合根、值对象和实体。

第三步： 根据业务及语义边界等因素，将一个或者多个聚合划定在一个限界上下文内，形成领域模型。在这个图根据领域实体之间的业务关联性，将业里，限界上下文之间的边界是第二层边界，这一层边界可能就是未来微服务的边界，不同限界上下文内的领域逻辑被隔离在不同的微服务实例中运行，物理上相互隔离

### 3.DDD 与微服务的关系：

DDD 主要关注：从业务领域视角划分领域边界，构建通用语言进行高效沟通，通过业务抽象，建立领域模型，维持业务和代码的逻辑一致性。

微服务主要关注：运行时的进程间通信、容错和故障隔离，实现去中心化数据管理和去中心化服务治理，关注微服务的独立开发、测试、构建和部署

### 4.核心域、支撑域和通用域

核心域: 决定产品和公司核心竞争力的子域是，它是业务成功的主要因素和公司的核心竞争力。

通用域: 没有太多个性化的诉求，同时被多个子域使用的通用功能子域是。

支撑域: 还有一种功能子域是必需的，但既不包含决定产品和公司核心竞争力的功能，也不包含通用功能的子域，它就是支撑域

### 5.限定上下文：

通用语言确定了项目团队内部交流的统一语言，而这个语言所在的语义环境则是由限界上下文来限定的，以确保语义的唯一性。

而领域专家、架构师和开发人员的主要工作就是 通过事件风暴来划分限界上下文。限界上下文确定了微服务的设计和拆分方向，是微服务设计和拆分的主要依据。如果不考虑技术异构、团队沟通等其它外部因素，一个限界上下文理论上就可以设计为一个微服务。

### 6.实体和值对象：

实体和值对象的目的都是抽象聚合若干属性以简化设计和沟通，有了这一层抽象，我们在使用人员实体时，不会产生歧义，在引用地址值对象时，不用列举其全部属性，在同一个限界上下文中，大幅降低误解、缩小偏差，两者的区别如下：

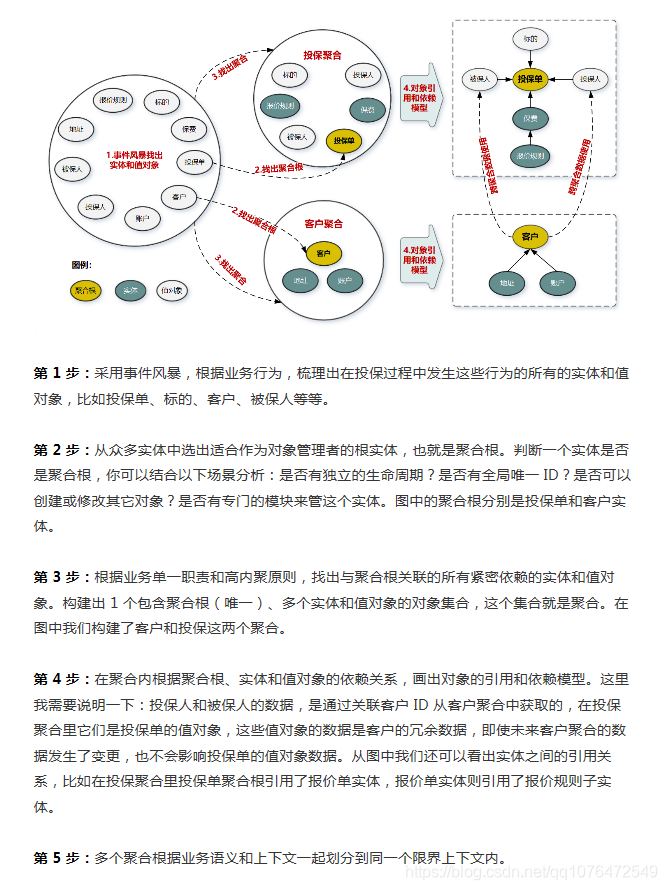
①两者都经过属性聚类形成， 实体有唯一性，值对象没有。在本文案例的限界上下文中，人员有唯一性，一旦某个人员被系统纳入管理，它就被赋予了在事件、流程和操作中被唯一识别的能力，而值对象没有也不必具备唯一性。

②实体着重唯一性和延续性，不在意属性的变化，属性全变了，它还是原来那个它；值对象着重描述性，对属性的变化很敏感，属性变了，它就不是那个它了。

③战略上的思考框架稳定不变，战术上的模型设计却灵活多变，实体和值对象也有可能随着系统业务关注点的不同而更换位置。比如，如果换一个特殊的限界上下文，这个上下文更关注地址，而不那么关注与这个地址产生联系的人员，那么就应该把地址设计成实体，而把人员设计成值对象。

有些领域对象可以设计为值对象，也可以设计为实体，我们需要根据具体情况来分析。如果这个领域对象在其它聚合内维护生命周期，且在它依附的实体对象中只允许整体替换，我们就可以将它设计为值对象。如果这个对象是多条且需要基于它做查询统计，我建议将它设计为实体

### ****7.如何设计一个聚合：****



### 8.聚合的设计原则：

聚合内的实体对象数据一致性，边界之外的数据和聚合内的互不影响，也就是说一个聚合内的数据是一起的

设计小聚合，大聚合会导致实体太多，高频操作导致事务大，并发等问题

聚合之间的关联通过聚合根ID进行关联

通过应用层进行领域编排或者通过事件机制

### 9.领域事件：

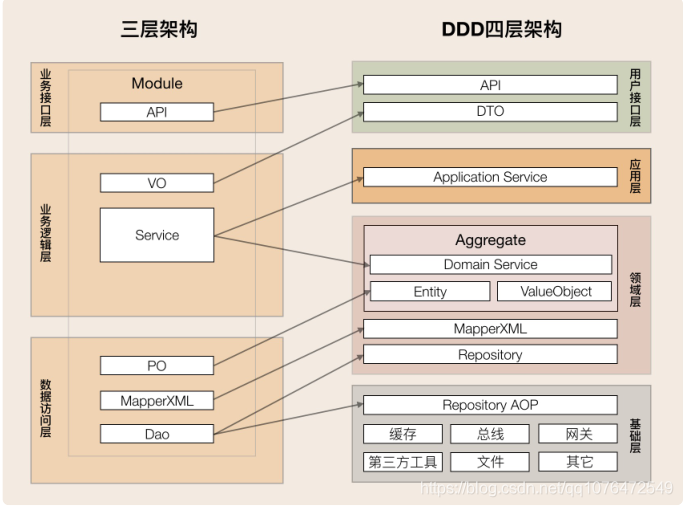
  为了多个聚合之间的交互方式：1.领域事件2通过应用层编排

    1.领域内的事件:通过本地事件表存储，通过eventbus发布事件

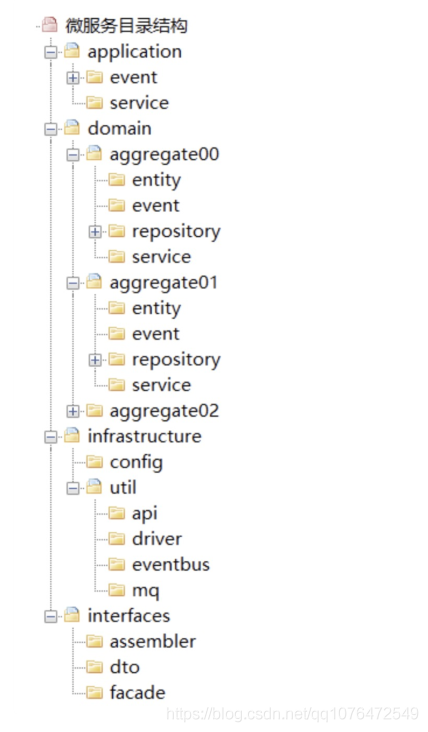
    2.领域外的事件:1.通过消息队列2.通过公共的事件库通信

## 二、进阶篇

### 传统MVC架构转基础的ddd架构

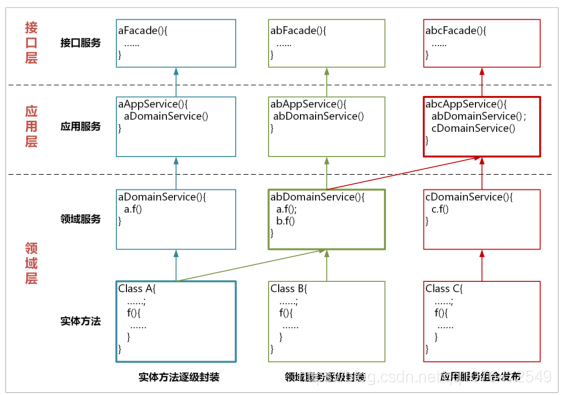


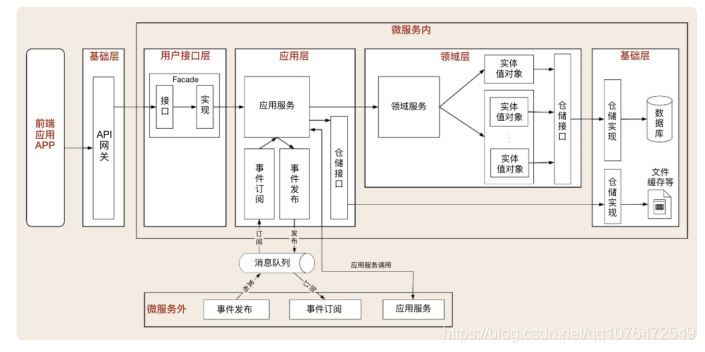
### 微服务代码结构



### 应用层需要跨领域调用怎么办

各层之间通过接口调用





第一种是应用服务调用并组装领域服务。此时领域服务会组装实体和实体方法，实现核心领域逻辑。领域服务通过仓储服务获取持久化数据对象，完成实体数据初始化。

第二种是应用服务直接调用仓储服务。这种方式主要针对像缓存、文件等类型的基础层数据访问。这类数据主要是查询操作，没有太多的领域逻辑，不经过领域层，不涉及数据库持久化对象

## 领域事件

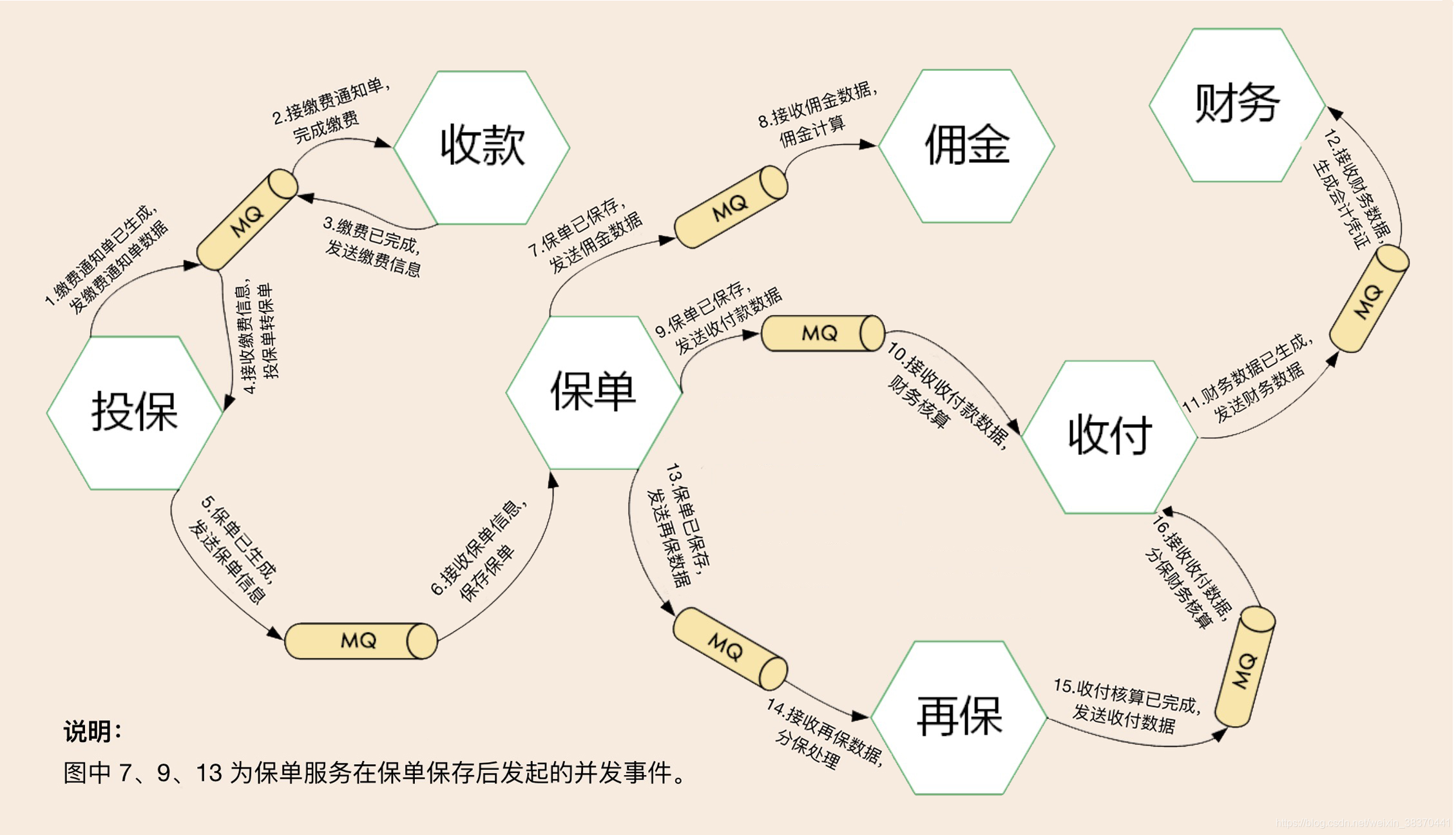
### 参考：

<https://blog.csdn.net/weixin_38370441/article/details/115654613>

### 领域事件：领域模型/微服务之间的事件

领域事件：捕捉业务、需求人员或领域专家口中的关键词：“如果发生……，则……”“当做完……的时候，请通知……”“发生……时，则……”等。在这些场景中，如果发生某种事件后，会触发进一步的操作，那么这个事件很可能就是领域事件。

举例：领域事件可以是业务流程的一个步骤，比如投保业务缴费完成后，触发投保单转保单的动作；也可能是定时批处理过程中发生的事件，比如批处理生成季缴保费通知单，触发发送缴费邮件通知操作；或者一个事件发生后触发的后续动作，比如密码连续输错三次，触发锁定账户的动作。



### 领域事件驱动设计

领域事件一般用在微服务/领域模型之间，切断领域模型之间的强依赖关系，通过领域事件来驱动业务的流转。领域事件需要保证一次事务最多只能更改一个聚合/领域的状态。如果一次业务操作涉及多个聚合/领域状态的更改，应采用领域事件的最终一致性。

跨微服务的事件机制要总体考虑事件构建、发布和订阅、事件数据持久化、消息中间件，甚至事件数据持久化时还可能需要考虑引入分布式事务机制等：微服务之间的访问可以采用应用服务直接调用的方式（RPC等），实现数据和服务的实时访问，弊端就是跨微服务的数据同时变更需要引入分布式事务，以确保数据的一致性。分布式事务机制会影响系统性能，增加微服务之间的耦合，所以我们还是要尽量避免使用分布式事务。

一个微服务/领域模型在事件发布完成时，即完成了事件实体构建和事件数据持久化。领域模型内发布方聚合将事件发布到事件总线。身份为订阅方的领域模型在接收到事件后触发后续业务操作。发布方不必关心后续订阅方事件处理是否成功，实现了领域模型之间的解耦，各个领域模型各自维护领域内数据的强一致性（一个领域事件只涉及到一个聚合的修改，聚合内部的强一致性必须得以维护），通过领域事件实现领域之间数据的最终一致性（涉及到其他聚合的修改时，通过发布事件的方式通知订阅方，订阅方在接收到消息后完成自己内部数据一致性的维护，从而实现了不同领域之间的最终一致性）。

无论是微服务内部还是之间，只要业务涉及到多个聚合/领域的更新，就可能会借助领域事件，引入领域事件则必然意味着聚合之间不再遵守强一致性，而是遵守最终一致性。

在业务操作和事件发布之间我们依然需要采用强一致性，也即这两者的发生应该是原子的，要么全部成功，要么全部失败，否则最终一致性根本无从谈起。以“订单积分”为例，如果客户下单成功，但是事件发送失败，此时就应该整体回退，告知客户操作失败。要保证业务操作和事件发布之间的原子性，如果MQ与业务操作不共享数据底层，那么最直接的方法便是采用XA事务（分布式事务）进行两阶段提交，即本地业务操作成功并且写入数据库的同时，也要保证MQ消息的写入，一方失败另一方都要回滚。

### 微服务内部的领域事件

微服务内大部分事件都发生在同一个进程内，因此不一定需要引入消息中间件。

如果业务需要同时更新多个聚合时，按照 DDD“一次事务只更新一个聚合”的原则，此时就要考虑是否引入事件总线。

微服务内的应用服务，即在应用服务中通过跨聚合的服务编排和组合，以服务调用的方式完成跨聚合的访问，这种方式通常应用于实时性和数据一致性要求高的场景。这个过程会用到分布式事务，以保证发布方和订阅方涉及到同一业务各自聚合内的数据实体同时更新成功（一个聚合对应一个仓储）。

在微服务内，不是少用领域事件，而是不建议使用事件总线。在DDD中是以聚合为单位进行数据管理，若一次操作会修改同一微服务内的多个聚合的数据，就需保证多个聚合的数据一致性，为了解耦不同聚合，可以采用分布式事务或事件总线两种方式。用事件总线不太方便管理服务和数据的关系，可用类似saga之类的分布式事务技术。总之需要确保不同聚合的业务规则和数据一致性，尽量减少系统建设复杂度。

### 微服务之间的领域事件

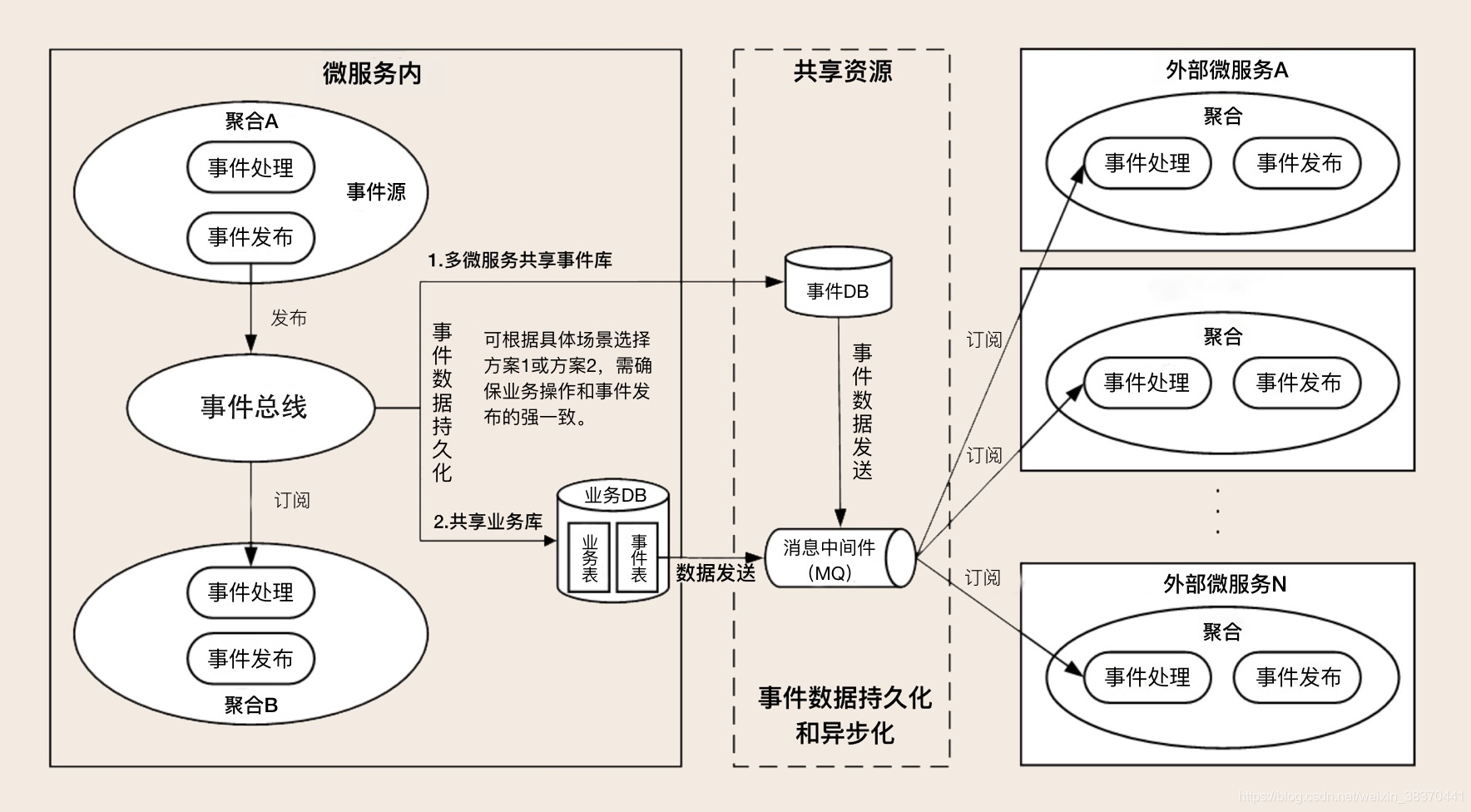
跨微服务的领域事件会在不同的限界上下文或领域模型之间实现业务协作，其主要目的是实现微服务解耦，减轻微服务之间实时服务访问的压力。

领域事件发生在微服务之间的场景比较多。事件机制要总体考虑事件构建、发布和订阅、事件数据持久化、消息中间件，甚至事件数据持久化时还可能需要考虑引入分布式事务机制等来保证数据表与MQ之间的一致性。

也可以采用RPC直接调用的方式，实现数据和服务的实时访问，弊端就是跨微服务的数据同时变更需要引入分布式事务，以确保同一业务不同聚合之间数据的一致性。实际上在事务中实现了不同微服务之间的同步调用。分布式事务机制会影响系统性能，增加微服务之间的耦合，所以我们还是要尽量避免使用分布式事务。

### 领域事件总体架构

领域事件处理包括：事件构建和发布、事件数据持久化、事件总线、消息中间件、事件接收和处理等。



注：保证事件发布和业务操作的强一致性！  
事件总线是引擎，是所有MQ的入口。MQ则是针对不同订阅者而言。事件总线是单服务的，MQ则是跨服务的。

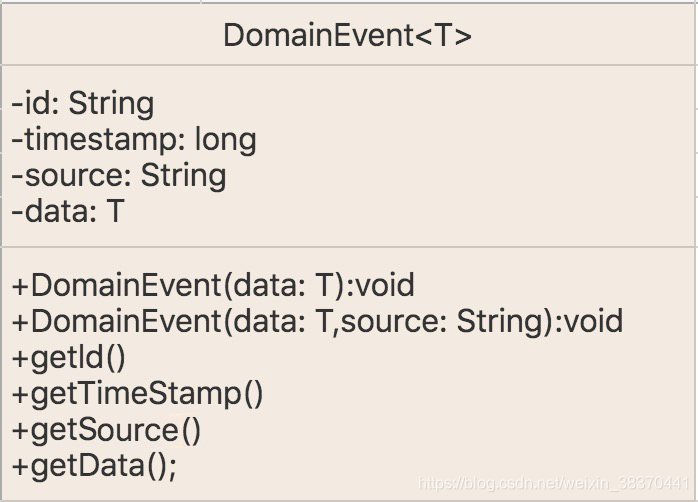
### 事件构建和发布

事件基本属性至少包括：事件唯一标识、发生时间、事件类型和事件源。其中事件唯一标识应该是全局唯一的，以便事件能够无歧义地在多个限界上下文中传递。事件基本属性主要记录事件自身以及事件发生背景的数据。

事件中还有一项更重要，那就是业务属性，用于记录事件发生那一刻的业务数据，这些数据会随事件传输到订阅方，以开展下一步的业务操作。

事件基本属性和业务属性一起构成事件实体，事件实体依赖聚合根。领域事件发生后，事件中的业务数据不再修改，因此业务数据可以以序列化值对象的形式保存，这种存储格式在消息中间件中也比较容易解析和获取。

为了保证事件结构的统一，我们还会创建事件基类 DomainEvent（参考下图），子类可以扩充属性和方法。由于事件没有太多的业务行为，实现方法一般比较简单。



事件发布的方式有很多种，你可以通过应用服务或者领域服务发布到事件总线或者消息中间件，也可以从事件表中利用定时程序或数据库日志捕获技术获取增量事件数据，发布到消息中间件。这种技术可以避免分布式事务的使用。

事件数据持久化

事件数据持久化可用于系统之间的数据对账，或者实现发布方和订阅方事件数据的审计。当遇到消息中间件、订阅方系统宕机或者网络中断，在问题解决后仍可继续后续业务流转，保证数据的一致性。

事件数据持久化有两种方案：

持久化到本地业务数据库的事件表中，利用本地事务保证业务表和事件数据的一致性。

持久化到共享的事件数据库中。这里需要注意的是：业务数据库和事件数据库不在一个数据库中，它们的数据持久化操作会跨数据库，因此需要分布式事务机制来保证业务和事件数据的强一致性，结果就是会对系统性能造成一定的影响。

事件总线

事件总线提供事件分发和接收等服务。事件总线是进程内模型，它会在微服务内聚合之间遍历订阅者列表，采取同步或异步的模式传递数据。事件分发流程大致如下：

如果是微服务内的订阅者（其它聚合），则直接分发到指定订阅者；

如果是微服务外的订阅者，将事件数据保存到事件库（表）并异步发送到消息中间件；

如果同时存在微服务内和外订阅者，则先分发到内部订阅者，将事件消息保存到事件库（表），再异步发送到消息中间件。

消息中间件

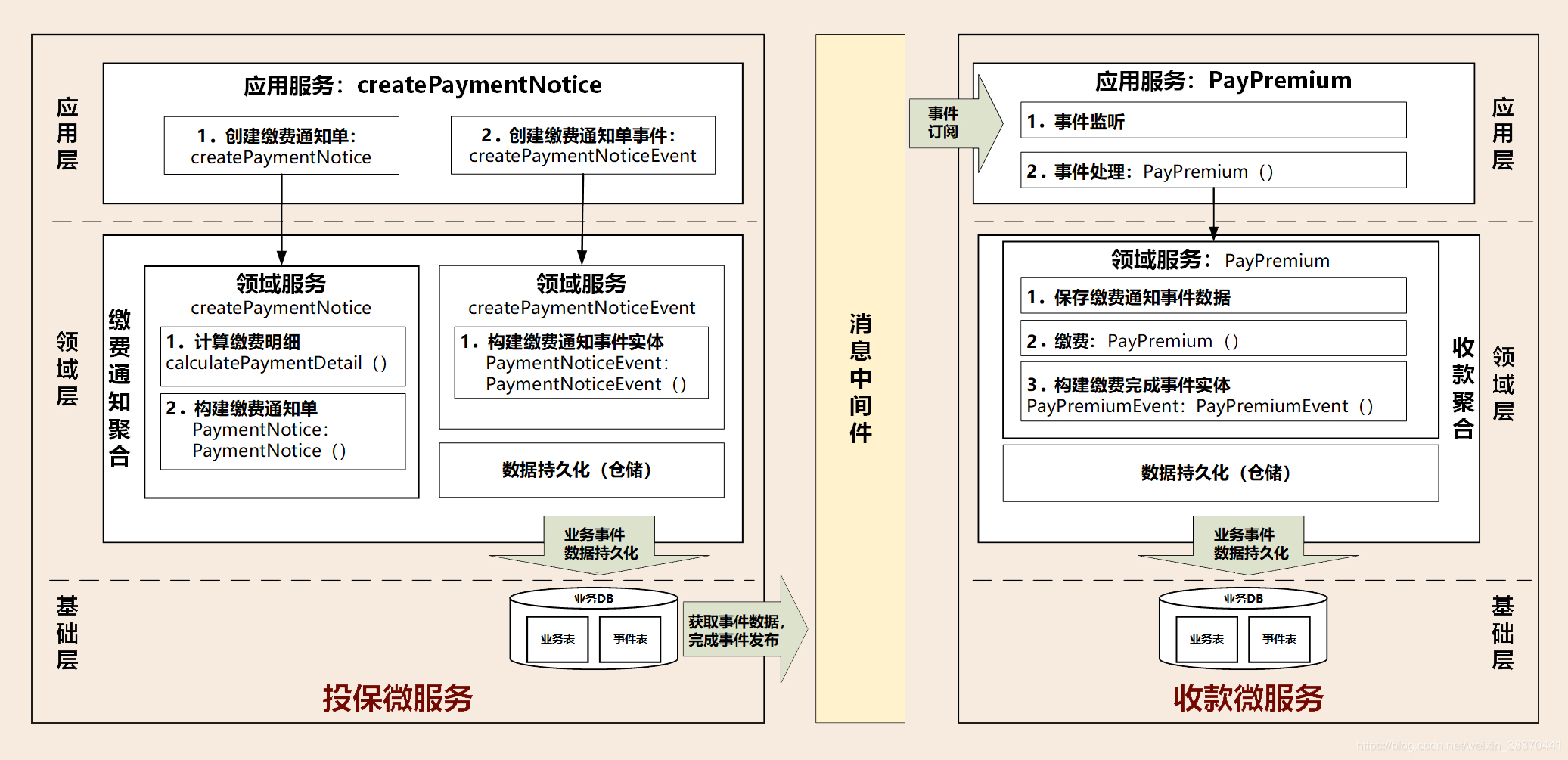
kafka或者RabbitMQ等。

事件接收和处理

微服务订阅方在应用层采用监听机制，接收消息队列中的事件数据，完成事件数据的持久化后，就可以开始进一步的业务处理。领域事件处理可在领域服务中实现。

具体实例

领域事件：缴费通知单生成，触发下一步业务操作：缴费。

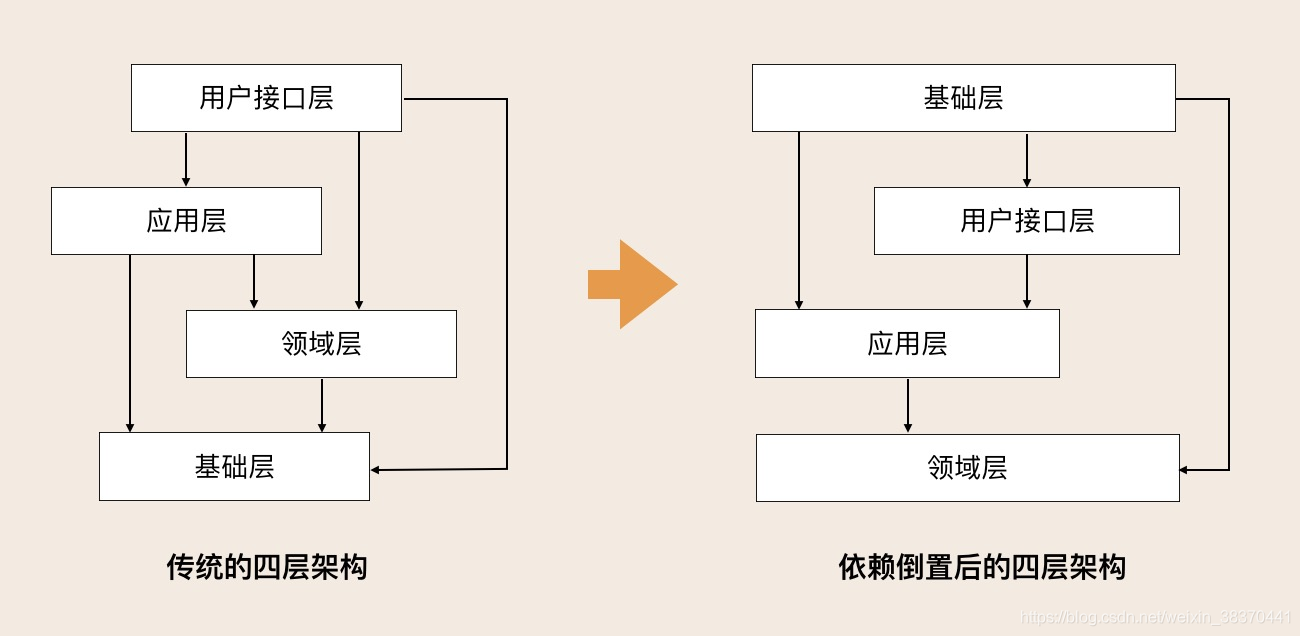


## 07 | DDD分层架构：各层职责建议

所有架构模型的核心都是：高内聚低耦合

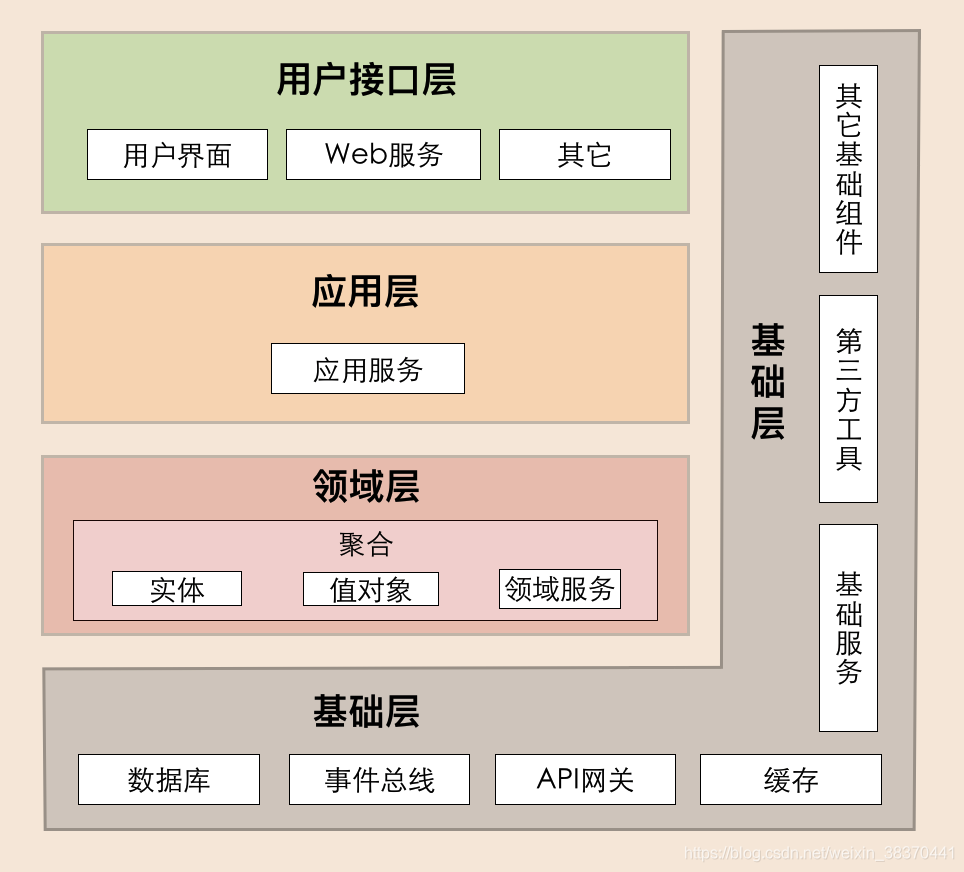
DDD分层架构

传统架构 -》 依赖倒置后的四层架构 -》 领域层与应用层新增上下文环境的五层架构



最早的传统四层架构中，基础层是被其它层依赖的，它位于最核心的位置，但实际上领域层才是软件的核心，所以这种依赖是有问题的。

后来采用了依赖倒置（Dependency inversion principle,DIP）的设计，优化了传统的四层架构，实现了各层对基础层的解耦，同时将核心定位为领域层。领域层发布仓储接口，仓储实现类需要实现领域层的仓储接口。



用户接口层

用户接口层负责向用户显示信息和解释用户指令。

这里的用户可能是：用户、程序、自动化测试和批处理脚本等等。面向前端提供服务适配，面向资源层提供资源适配。这一层聚集了接口适配相关的功能。

应用层 & 应用服务

应用层是很薄的一层，理论上不应该有领域业务规则或逻辑，主要面向用例和流程相关的操作，实现服务组合和编排，适应业务流程快速变化的需求。这一层聚集了应用服务和事件相关的功能。

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

通过应用服务对外暴露微服务的内部功能，这样就可以隐藏领域层核心业务逻辑的复杂性以及内部实现机制。应用层的主要服务形态有：应用服务、事件发布和订阅服务。

为了实现微服务内聚合之间的解耦，聚合之间的服务调用和数据交互应通过应用服务来完成。原则上我们应该禁止聚合之间的领域服务直接调用和聚合之间的数据表关联。

注意：不要将本该放在领域层的业务逻辑放到应用层中实现。因为庞大的应用层会使领域模型失焦，时间一长你的微服务就会演化为传统的三层架构，业务逻辑会变得混乱。

领域层 & 领域服务

领域层主要体现领域模型的业务能力，它用来表达业务概念、业务状态和业务规则。领域层包含聚合根、实体、值对象、领域服务等领域模型中的领域对象。当领域中的某些功能，单一实体（或者值对象）不能实现时，领域服务就会出马，它可以组合聚合内的多个实体（或者值对象），实现复杂的业务逻辑。

实体采用充血模型，在实体类内部实现实体相关的所有业务逻辑，实现的形式是实体类中的方法。实体是微服务的原子业务逻辑单元。在设计时我们主要考虑实体自身的属性和业务行为，实现领域模型的核心基础能力。不必过多考虑外部操作和业务流程，这样才能保证领域模型的稳定性。

基础层

基础层是贯穿所有层的，它的作用就是为其它各层提供通用的技术和基础服务，包括第三方工具、驱动、消息中间件、网关、文件、缓存以及数据库等。比较常见的功能还是提供数据库持久化。

基础层包含基础服务，它采用依赖倒置设计，封装基础资源服务，实现应用层、领域层与基础层的解耦，降低外部资源变化对应用的影响。

基础层的服务形态主要是仓储服务。仓储服务包括接口和实现两部分。仓储接口服务供应用层或者领域层服务调用，仓储层实现具体的服务实现，通过自动注入的方式供其它层调用，领域层与应用层通过调用接口完成领域对象的持久化或数据初始化。从而避免实现与具体仓储的绑定。（待确认？？？）

最重要的原则

在《实现领域驱动设计》一书中，DDD 分层架构有一个重要的原则：每层只能与位于其下方的层发生耦合。

架构根据耦合的紧密程度又可以分为两种：严格分层架构和松散分层架构。优化后的 DDD 分层架构模型就属于严格分层架构，任何层只能对位于其直接下方的层产生依赖。

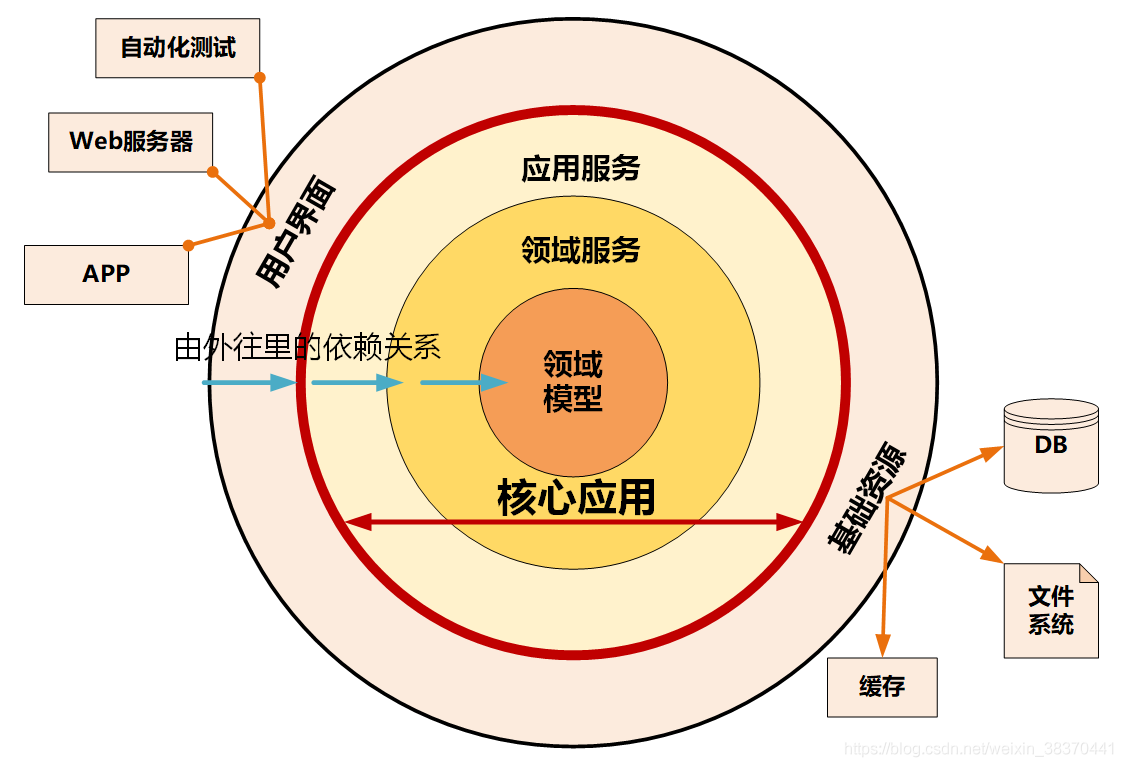
在严格分层架构中，\*\*领域服务只能被应用服务调用，而应用服务只能被用户接口层调用，服务是逐层对外封装或组合的，依赖关系清晰。\*\*在松散分层架构中，领域服务可以同时被应用层或用户接口层调用，服务的依赖关系比较复杂且难管理，甚至容易使核心业务逻辑外泄。

由于层间松耦合，我们可以专注于本层的设计，而不必关心其它层，也不必担心自己的设计会影响其它层。可以说，DDD 成功地降低了层与层之间的依赖。

## 08 | 微服务架构模型：几种常见模型的对比和分析

DDD分层架构、整洁架构、六边形架构之间的对比分析。

### **整洁架构**

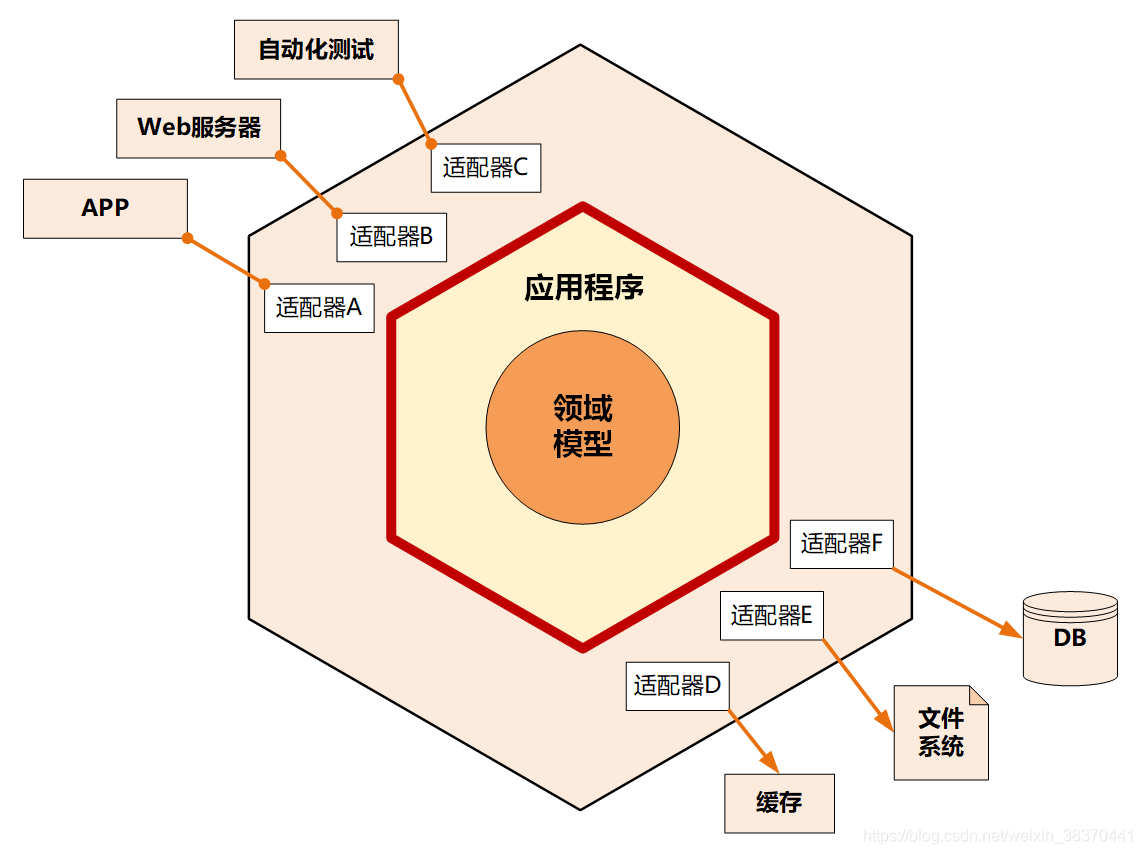


整洁架构最主要的原则是依赖原则，它定义了各层的依赖关系，越往里依赖越低，代码级别越高，越是核心能力。外圆代码依赖只能指向内圆，内圆不需要知道外圆的任何情况。

最外层主要提供适配的能力，适配能力分为主动适配和被动适配。主动适配主要实现外部用户、网页、批处理和自动化测试等对内层业务逻辑访问适配。被动适配主要是实现核心业务逻辑对基础资源访问的适配，比如数据库、缓存、文件系统和消息中间件等。

与四层架构相近，只是用户接口层与仓储层合成了用户界面层与基础资源而已。

### **六边形架构/端口适配器架构**



核心理念是：应用是通过端口与外部进行交互的。

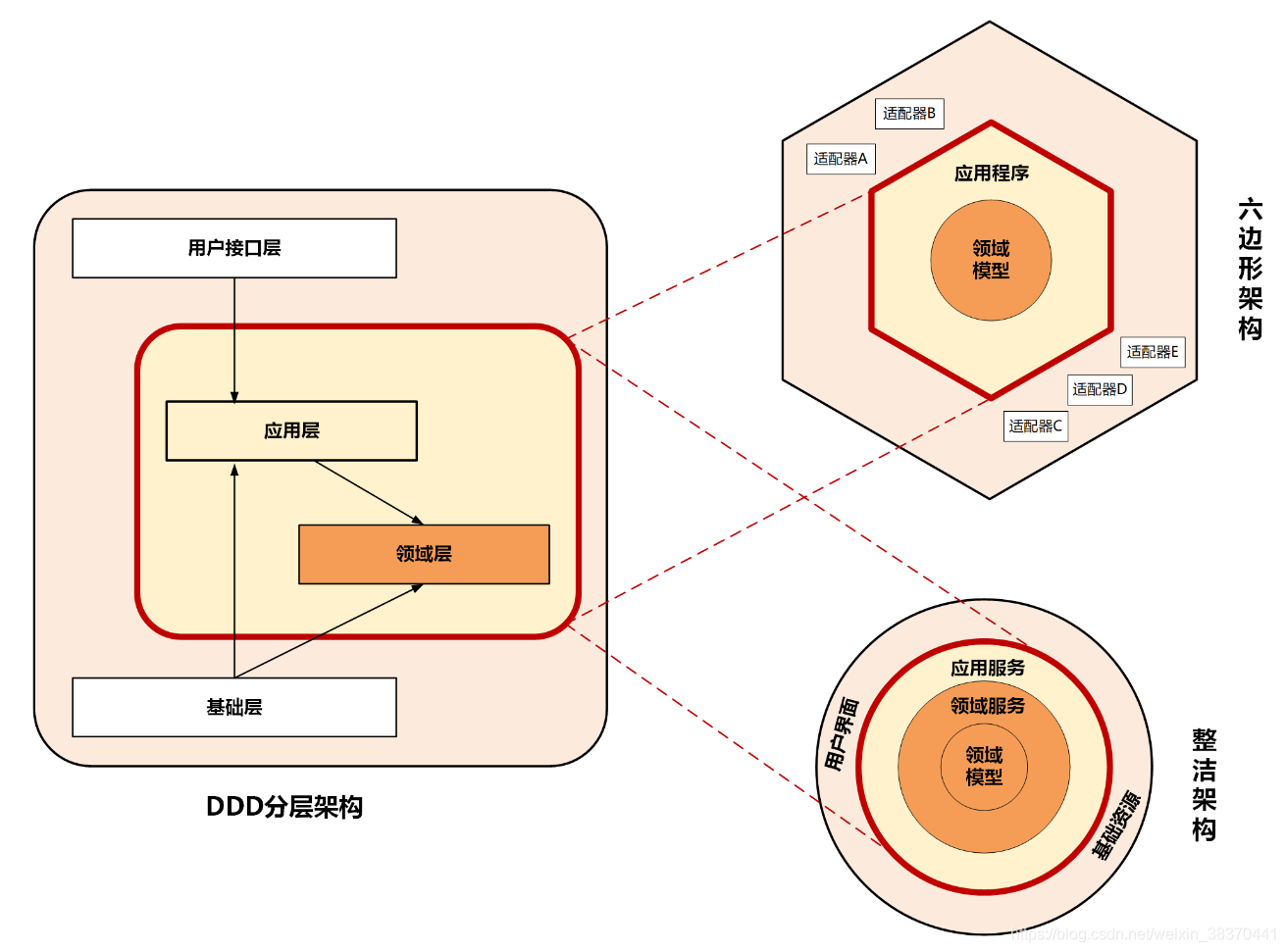
六边形架构将系统分为内六边形和外六边形两层，这两层的职能划分如下：

红圈内的六边形实现应用的核心业务逻辑；

外六边形完成外部应用、驱动和基础资源等的交互和访问，对前端应用以 API 主动适配的方式提供服务，对基础资源以依赖倒置被动适配的方式实现资源访问。

与四层架构相近，只是用户接口层与仓储层合成了各种适配器。

### **三种架构的对比分析**



这三种架构都考虑了前端需求的变与领域模型的不变。通过分层的方式来控制需求变化从外到里对系统的影响，从外向里受需求影响逐步减小。面向用户的前端可以快速响应外部需求进行调整和发布，灵活多变，应用层通过服务组合和编排来实现业务流程的快速适配上线，减少传导到领域层的需求，使领域层保持长期稳定

从三种架构模型看中台和微服务设计

中台本质上是领域的子域，它可能是核心域，也可能是通用域或支撑域。通常大家认为阿里的中台对应 DDD 的通用域，将通用的公共能力沉淀为中台，对外提供通用共享服务。

中台作为子域还可以继续分解为子子域，在子域分解到合适大小，通过事件风暴划分限界上下文以后，就可以定义微服务了，微服务用来实现中台的能力。

### 两种类型的微服务

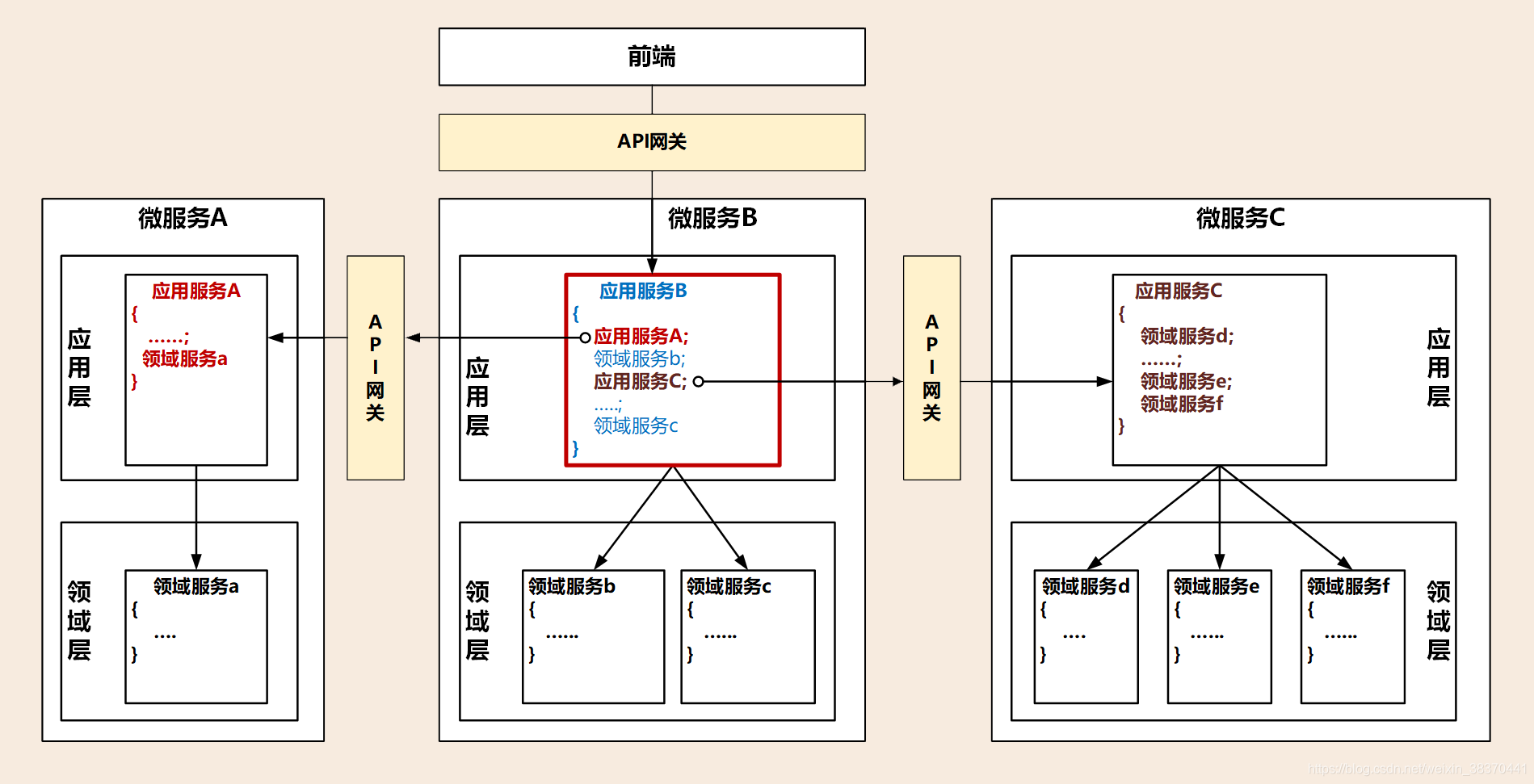
微服务要有合理的架构分层

微服务设计要有分层的设计思想，不要把与领域无关的逻辑放在领域层实现，保证领域层的纯洁和领域逻辑的稳定，避免污染领域模型。

也不要把领域模型的业务逻辑放在应用层，这样会导致应用层过于庞大，最终领域模型会失焦。如果实在无法避免，我们可以引入防腐层，进行新老系统的适配和转换，过渡期完成后，可以直接将防腐层代码抛弃。

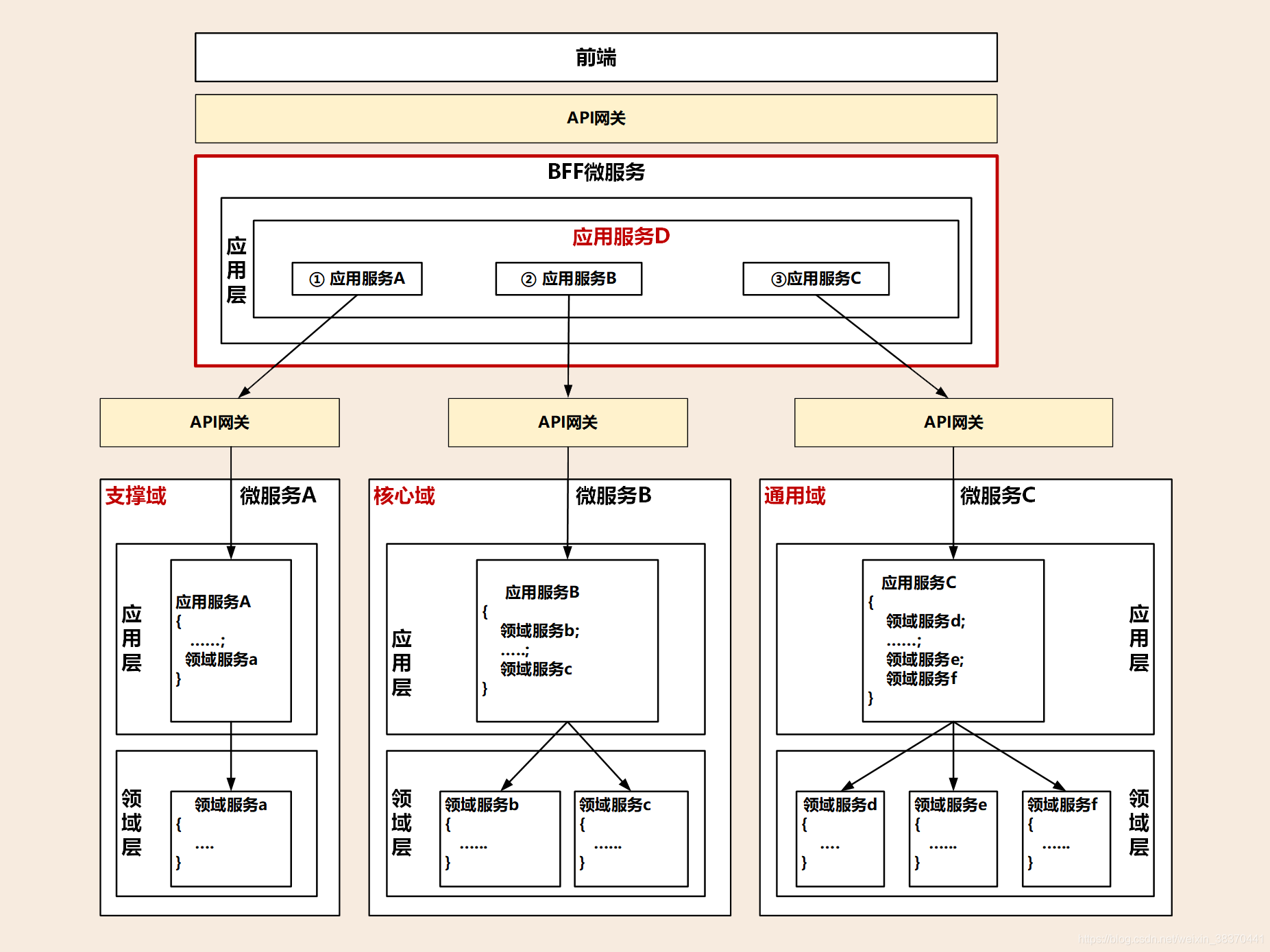
有的微服务可以与前端应用集成，一起完成特定的业务，这是项目级微服务。而有的则是某个职责单一的中台微服务，企业级的业务流程需要将多个这样的微服务组合起来才能完成，这是中台微服务。

### 项目级微服务



通常项目级微服务之间的集成，发生在微服务的应用层，由应用服务调用其它微服务发布在 API 网关上的应用服务。

企业级中台微服务



企业级微服务不能像项目级微服务那样，在某一个微服务内部完成服务的组合与编排。可以在微服务之上再加一层，即BFF（服务于前端的后端）。

BFF的主要职能就是处理跨中台微服务的服务组合和编排，以及微服务之间的协调，它还可以完成前端不同渠道应用的适配。如果再将它的业务范围扩大一些，我可以将它做成一个面向不同行业和渠道的服务平台。

BFF 微服务与其它微服务存在较大的差异，就是它没有领域模型，因此这个微服务内也不会有领域层。BFF 微服务可以承担应用层和用户接口层的主要职能，完成各个中台微服务的服务组合和编排，可以适配不同前端和渠道的要求。

核心：依赖倒置原则的应用

在微服务架构中，应用层、领域层和基础层解耦是通过仓储模式，采用依赖倒置的设计方法来实现的。在应用设计中，我们会同步考虑和基础资源的代码适配，那么一旦基础设施资源出现变更（比如换数据库），就可以屏蔽资源变更对业务代码的影响，切断业务逻辑对基础资源的依赖，最终降低资源变更对应用的影响。避免了对数据库、缓存、文件系统等基础资源的严重依赖。