我们前面已经讲了很多 DDD 的设计方法和实践案例。虽然 DDD 的设计思想和方法很好，但由于企业发展历程以及企业技术和文化的不同，DDD 和微服务的实施策略也会有差异。那么面对这种差异，我们应该如何落地 DDD 和微服务呢？今天我们就来聊聊微服务的设计原则和演进策略。

微服务的演进策略

在从单体向微服务演进时，演进策略大体分为两种：绞杀者策略和修缮者策略。

1. **绞杀者策略**

绞杀者策略是一种逐步剥离业务能力，用微服务逐步替代原有单体系统的策略。它对单体系统进行领域建模，根据领域边界，在单体系统之外，将新功能和部分业务能力独立出来，建设独立的微服务。新微服务与单体系统保持松耦合关系。

随着时间的推移，大部分单体系统的功能将被独立为微服务，这样就慢慢绞杀掉了原来的单体系统。绞杀者策略类似建筑拆迁，完成部分新建筑物后，然后拆除部分旧建筑物。

2. **修缮者策略**

修缮者策略是一种维持原有系统整体能力不变，逐步优化系统整体能力的策略。它是在现有系统的基础上，剥离影响整体业务的部分功能，独立为微服务，比如高性能要求的功能，代码质量不高或者版本发布频率不一致的功能等。

通过这些功能的剥离，我们就可以兼顾整体和局部，解决系统整体不协调的问

题。修缮者策略类似古建筑修复，将存在问题的部分功能重建或者修复后，重新加入到原有的建筑中，保持建筑原貌和功能不变。一般人从外表感觉不到这个变化，但是建筑物质量却得到了很大的提升。

其实还有第三种策略，就是另起炉灶，顾名思义就是将原有的系统推倒重做。建设期间，原有单体系统照常运行，一般会停止开发新需求。而新系统则会组织新的项目团队，按照原有系统的功能域，重新做领域建模，开发新的微服务。在完成数据迁移后，进行新旧系统切换。

对于大型核心系统我一般不建议采用这种策略，这是因为系统重构后的不稳定性、大量未知的潜在技术风险和新的开发模式下项目团队磨合等不确定性因素，会导致项目实施难度大大增加。

不同场景下的领域建模策略

由于企业内情况千差万别，发展历程也不一样，有遗留单体系统的微服务改造，也有全新未知领域的业务建模和系统设计，还有遗留系统局部优化的情况。不同场景下，领域建模的策略也会有差异。下面我们就分几类场景来看看如何进行领域建模。

1. 新建系统

新建系统又分为简单和复杂领域建模两种场景。

简单领域建模

简单的业务领域，一个领域就是一个小的子域。在这个小的问题域内，领域建模过程相对简单，直接采用事件风暴的方法构建领域模型就可以了。

复杂领域建模

对于复杂的业务领域，领域可能需要多级拆分后才能开始领域建模。领域拆分为子域，甚至子域还需要进一步拆分。比如：保险它需要拆分为承保、理赔、收付费和再保等子域，承保子域再拆分为投保、保单管理等子子域。复杂领域如果不做进一步细分，由于问题域太大，领域建模的工程量会非常浩大。你不太容易通过事件风暴，完成一个很大的领域建模，即使勉强完成，效果也不一定好。

对于复杂领域，我们可以分三步来完成领域建模和微服务设计。

第一步，拆分子域建立领域模型

根据业务领域的特点，参考流程节点边界或功能聚合模块等边界因素。结合领域专家和项目团队的讨论，将领域逐级分解为大小合适的子域，针对子域采用事件风暴，划分聚合和限界上下文，初步确定子域内的领域模型。

第二步，领域模型微调

梳理领域内所有子域的领域模型，对各子域领域模型进行微调。微调的过程重点考虑不同领域模型中聚合的重组。同步考虑领域模型和聚合的边界，服务以及事件之间的依赖关系，确定最终的领域模型。

第三步，微服务的设计和拆分

根据领域模型和微服务拆分原则，完成微服务的拆分和设计。

2. **单体遗留系统**

如果我们面对的是一个单体遗留系统，只需要将部分功能独立为微服务，而其余仍为单体，整体保持不变，比如将面临性能瓶颈的模块拆分为微服务。我们只需要将这一特定功能，理解为一个简单子领域，参考简单领域建模的方式就可以了。在微服务设计中，我们还要考虑新老系统之间服务和业务的兼容，必要时可引入防腐层。

DDD 使用的误区

很多人在接触微服务后，但凡是系统，一概都想设计成微服务架构。其实有些业务场景，单体架构的开发成本会更低，开发效率更高，采用单体架构也不失为好的选择。同样，虽然 DDD 很好，但有些传统设计方法在微服务设计时依然有它的用武之地。下面我们就来聊聊 DDD 使用的几个误区。

1. **所有的领域都用 DDD**

很多人在学会 DDD 后，可能会将其用在所有业务域，即全部使用 DDD 来设计。DDD 从战略设计到战术设计，是一个相对复杂的过程，首先企业内要培养 DDD 的文化，其次对团队成员的设计和技术能力要求相对比较高。在资源有限的情况下，应聚焦核心域，建议你先从富领域模型的核心域开始，而不必一下就在全业务域推开。

2. 全部采用 DDD 战术设计方法

不同的设计方法有它的适用环境，我们应选择它最擅长的场景。DDD 有很多

的概念和战术设计方法，比如聚合根和值对象等。聚合根利用仓储管理聚合内实体数据之间的一致性，这种方法对于管理新建和修改数据非常有效，比如在修改订单数据时，它可以保证订单总金额与所有商品明细金额的一致，但它并不擅长较大数据量的查询处理，甚至有延迟加载进而影响效率的问题。

而传统的设计方法，可能一条简单的 SQL 语句就可以很快地解决问题。而很多贫领域模型的业务，比如数据统计和分析，DDD 很多方法可能都用不上，或用得并不顺手，而传统的方法很容易就解决了。

因此，在遵守领域边界和微服务分层等大原则下，在进行战术层面设计时，我们应该选择最适合的方法，不只是 DDD 设计方法，当然还应该包括传统的设计方法。这里要以快速、高效解决实际问题为最佳，不要为做 DDD 而做 DDD。

3. **重战术设计而轻战略设计**

很多 DDD 初学者，学习 DDD 的主要目的，可能是为了开发微服务，因此更看重 DDD 的战术设计实现。殊不知 DDD 是一种从领域建模到微服务落地的全方位的解决方案。

战略设计时构建的领域模型，是微服务设计和开发的输入，它确定了微服务的边界、聚合、代码对象以及服务等关键领域对象。领域模型边界划分得清不清晰，领域对象定义得明不明确，会决定微服务的设计和开发质量。没有领域模型的输入，基于 DDD 的微服务的设计和开发将无从谈起。因此我们不仅要重视战术设计，更要重视战略设计。

4. **DDD 只适用于微服务**

DDD 是在微服务出现后才真正火爆起来的，很多人会认为 DDD 只适用于微服务。在 DDD 沉默的二十多年里，其实它一直也被应用在单体应用的设计中。

具体项目实施时，要吸取 DDD 的核心设计思想和理念，结合具体的业务场景和团队技术特点，多种方法组合，灵活运用，用正确的方式解决实际问题。

微服务设计原则

微服务设计原则中，如高内聚低耦合、复用、单一职责等这些常见的设计原则在此就不赘述了，我主要强调下面这几条：

第一条：要领域驱动设计，而不是数据驱动设计，也不是界面驱动设计。

微服务设计首先应建立领域模型，确定逻辑和物理边界以及领域对象后，然后才开始微服务的拆分和设计。而不是先定义数据模型和库表结构，也不是前端界面需要什么，就去调整核心领域逻辑代码。在设计时应该将外部需求从外到内逐级消化，尽量降低对核心领域层逻辑的影响。

第二条：要边界清晰的微服务，而不是泥球小单体。

微服务上线后其功能和代码也不是一成不变的。随着需求或设计变化，领域模型会迭代，微服务的代码也会分分合合。边界清晰的微服务，可快速实现微服务代码的重组。微服务内聚合之间的领域服务和数据库实体原则上应杜绝相互依赖。你可通过应用服务编排或者事件驱动，实现聚合之间的解耦，以便微服务的架构演进。

第三条：要职能清晰的分层，而不是什么都放的大箩筐。

分层架构中各层职能定位清晰，且都只能与其下方的层发生依赖，也就是说只能从外层调用内层服务，内层通过封装、组合或编排对外逐层暴露，服务粒度也由细到粗。应用层负责服务的组合和编排，不应有太多的核心业务逻辑，领域层负责核心领域业务逻辑的实现。各层应各司其职，职责边界不要混乱。在服务演进时，应尽量将可复用的能力向下层沉淀。

第四条：要做自己能 hold 住的微服务，而不是过度拆分的微服务。

微服务过度拆分必然会带来软件维护成本的上升，比如：集成成本、运维成本、监控和定位问题的成本。企业在微服务转型过程中还需要有云计算、DevOps、自动化监控等能力，而一般企业很难在短时间内提升这些能力，如果项目团队没有这些能力，将很难 hold 住这些微服务。

如果在微服务设计之初按照 DDD 的战略设计方法，定义好了微服务内的逻辑边界，做好了架构的分层，其实我们不必拆分太多的微服务，即使是单体也未尝不可。随着技术积累和能力提升，当我们有了这些能力后，由于应用内有清晰的逻辑边界，我们可以随时轻松地重组出新的微服务，而这个过程不会花费太多的时间和精力。

微服务拆分需要考虑哪些因素？

理论上一个限界上下文内的领域模型可以被设计为微服务，但是由于领域建模主要从业务视角出发，没有考虑非业务因素，比如需求变更频率、高性能、安全、团队以及技术异构等因素，而这些非业务因素对于领域模型的系统落地也

会起到决定性作用，因此在微服务拆分时我们需要重点考虑它们。我列出了以下主要因素供你参考。

1. **基于领域模型**

基于领域模型进行拆分，围绕业务领域按职责单一性、功能完整性拆分。

2. **基于业务需求变化频率**

识别领域模型中的业务需求变动频繁的功能，考虑业务变更频率与相关度，将业务需求变动较高和功能相对稳定的业务进行分离。这是因为需求的经常性变动必然会导致代码的频繁修改和版本发布，这种分离可以有效降低频繁变动的敏态业务对稳态业务的影响。

3. **基于应用性能**

识别领域模型中性能压力较大的功能。因为性能要求高的功能可能会拖累其它功能，在资源要求上也会有区别，为了避免对整体性能和资源的影响，我们可以把在性能方面有较高要求的功能拆分出去。

4. 基于组织架构和团队规模

除非有意识地优化组织架构，否则微服务的拆分应尽量避免带来团队和组织架构的调整，避免由于功能的重新划分，而增加大量且不必要的团队之间的沟通成本。拆分后的微服务项目团队规模保持在 10～12 人左右为宜。

5. **基于安全边界**

有特殊安全要求的功能，应从领域模型中拆分独立，避免相互影响。

6. **基于技术异构等因素**

领域模型中有些功能虽然在同一个业务域内，但在技术实现时可能会存在较大的差异，也就是说领域模型内部不同的功能存在技术异构的问题。由于业务场景或者技术条件的限制，有的可能用.NET，有的则是 Java，有的甚至大数据架构。对于这些存在技术异构的功能，可以考虑按照技术边界进行拆分。