答疑：有关3个典型问题的讲解

2019-11-06 00:00:00 欧创新

DDD实战课

你好，我是欧创新。

截至今天这一讲，我们的基础篇和进阶篇的内容就结束了。在这个过程中，我一直有关注大家提的问题。那在实战篇正式开始之前啊，我想针对3个比较典型的问题，做一个讲解，希望你也能同步思考，调动自己已学过的内容，这对我们后面实战篇的学习也是有一定帮助的。

问题1：有关于领域可以划分为核心域、通用域和支撑域，以及子域和限界上下文关系的话题，还有是否有边界划分的量化标准？

我在【第 02 讲】中讲到了，在领域不断划分的过程中，领域会被细分为不同的子域，这个过程实际上是将问题范围不断缩小的过程。

借用读者“密码123456”的总结，他认为：“对于领域问题来说，可以理解为，对一个问题不断地划分，直到划分为我们熟悉的、能够快速处理的小问题。然后再对小问题的处理排列一个优先级。”

这个理解是很到位的。在领域细分到一定的范围后，我们就可以对这个子域进行事件风暴，为这个子域划分限界上下文，建立领域模型，然后就可以基于领域模型进行微服务设计了。

虽然DDD没有明确说明子域和限界上下文的关系。我个人认为，子域的划分是一种比较粗的领域边界的划分，它不考虑子域内的领域对象、对象之间的关系和结构。子域的划分往往按照业务阶段或者功能模块边界进行粗分，其目的就是为了让你能够在一个相对较小的问题空间内，比较方便地用事件风暴来梳理业务场景。

而限界上下文本质上也是子域，限界上下文是在明确的子域内，用事件风暴划分出来的。它体现的是一种详细的设计过程。这个过程设计出了领域模型，明确了领域对象以及领域对象的依赖等关系，有了领域模型，你就可以直接进行微服务设计了。

关于核心域、通用域和支撑域，划分这三个不同类型子域的主要目的是为了区分业务域的优先级，确定IT战略投入。我们会将重要的资源投入在核心域上，确保好钢用在刀刃上。每个企业由于商业模式或者战略方向不一样，核心域会有一些差异，不要用固定的眼光看待不同企业的核心域。

核心域、通用域和支撑域都是业务领域，只不过重要性和功能属性不一样。采用的DDD设计方法和过程，是没有差异的。

从目前来看，还没有可以量化的领域以及限界上下文的划分标准。它主要依赖领域专家经验，以及项目团队在事件风暴过程中不断地权衡和分析。不要奢望一次迭代就能够给复杂的业务，建立一个完美的领域模型。领域模型很多时候也需要多次迭代才能成型，它也需要不断地演进。但如果是用DDD设计出来的领域模型的边界和微服务内聚合的边界非常清晰的话，这个演进过程相对来说会简单很多，不仅时间成本也会很低。

问题2：关于聚合设计的问题？领域层与基础层为什么要依赖倒置（DIP）？

聚合主要实现核心业务逻辑，里面有很多的领域对象，这些领域对象之间需要通过聚合根进行统一的管理，以确保数据的一致性。

在聚合设计时，我们会用到三个重要的设计模式：工厂（Factory）模式和仓储（Repository）模式。如果你有兴趣详细了解的话，推荐你阅读《实现领域驱动设计》一书的第11章和第12章。

那为什么要引入工厂模式呢？

这是因为有些聚合内可能含有非常多的实体和值对象，我们需要确保聚合根以及所有被依赖的对象实例同时被创建。如果都通过聚合根来构造，将会非常复杂。因此我们可以通过工厂模式来封装复杂对象的创建过程，但并不是所有对象的构造都需要用到工厂，如果构造过程不复杂，只是单一对象的构造，你用简单的构造方法就足够了。

又为什么要引入仓储模式？解答这个问题的同时，我也一起将依赖倒置的问题解答一下。

在传统的DDD四层架构中，所有层都是依赖基础层的。这样做有什么不好的地方呢？如果应用逻辑对基础层依赖太大的话，基础层中与资源有关的代码可能会渗透到应用逻辑中。而现在技术组件的更新频率是很快的，一旦出现基础组件的变更，且基础组件的代码被带入到了应用逻辑中，这样会对上层的应用逻辑产生致命的影响。

为了解耦应用逻辑和基础资源，在基础层和上层应用逻辑之间会增加一层，这一层就是仓储层。一个聚合对应一个仓储，仓储实现聚合内数据的持久化。聚合内的应用逻辑通过接口来访问基础资源，仓储实现在基础层实现。这样应用逻辑和基础资源的实现逻辑是分离的。如果变更基础资源组件，只需要替换仓储实现就可以了，不会对应用逻辑产生太大的影响，这样就实现了应用逻辑与基础资源的解耦，也就实现了依赖倒置。

关于聚合设计过程中的一些原则问题。大部分的业务场景我们都可以通过事件风暴，找到聚合根，建立聚合，划分限界上下文，建立领域模型。但也有部分场景，比如数据计算、统计以及批处理业务场景，所有的实体都是独立无关联的，找不到聚合根，也无法建立领域模型。但是它们之间的业务关系是非常紧密的，在业务上是高内聚的。我们也可以将这类场景作为一个聚合处理，除了不考虑聚合根的设计方法外，其它诸如DDD分层架构相关的设计方法都是可以采用的。

一些业务场景，如果复杂度并不高，而用DDD设计会带来不必要的麻烦的话，比如增加复杂度，有些原则也是可以突破的，不要为做DDD而做DDD。即使采用传统的方式也是没有关系的，最终以解决问题为最佳。但必须记住一点，如果采用传统的设计方式，一定要保证领域模型的边界以及微服务内聚合的逻辑边界清晰，这样的话，以后微服务的演进就不会太复杂。

问题3：领域事件采用消息异步机制，发布方和订阅方数据如何保证一致性？微服务内聚合之间领域事件是否一定要用事件总线？

在领域事件设计中，为了解耦微服务，微服务之间数据采用最终一致性原则。由于发布方是在消息总线发布消息以后，并不关心数据是否送达，或者送达后订阅方是否正常处理，因此有些技术人会担心发布方和订阅方数据一致性的问题。

那在对数据一致性要求比较高的业务场景，我们是有相关的设计考虑的。也就是发送方和订阅方的事件数据都必须落库，发送方除了保存业务数据以外，在往消息中间件发布消息之前，会先将要发布的消息写入本地库。而接收方在处理消息之前，需要先将收到的消息写入本地库。然后可以采用定期对发布方和订阅方的事件数据对账的操作，识别出不一致的数据。如果数据出现异常或不一致的情况，可以启动定时程序再次发送，必要时可以转人工操作处理。

关于事件总线的问题。由于微服务内的逻辑都在一个进程内，后端数据库也是一个，微服务内的事务相比微服务之间会好控制一些。在处理微服务内的领域事件时，如果引入事件总线，会增加开发的复杂度，那是否引入事件总线，就需要你来权衡。

个人感觉如果你的场景中，不会出现导致聚合之间数据不一致的情况，就可以不使用事件总线。另外，通过应用服务也可以实现聚合之间的服务和数据协调。

以上就是3个典型问题的答案了，不知你先前是否有同样的疑惑，这些答案又是否与你不谋而合呢？如果你还有其它问题，欢迎在留言区提出，我会一一解答。

今天的内容就到这了，如果有所收获，也可以分享给你的朋友，我们实战篇见！