DDD落地指南-架构师眼中的餐厅

京东云开发者 2024-04-16 1,509 阅读23分钟

在去年、我整理了一篇名为《如何做架构设计？》的文章，主要探讨了架构设计的目标和过程，然而、那是一篇概括性的文章，用于启发思路，并不是具体的实践指南，因此、我一直期望给出具体参考案例。

我几乎忘了这件事，如今回顾、我发现并没有合适的案例可供参考，现有的案例要么不完整、要么是与业务耦合的特定场景，要么无法支撑研发落地。所以我决定从实际生活中出发，虚拟一个案例场景，以便能够系统性的阐述这个问题。

正文开始

本案例侧重于DDD的实践，从实际业务场景推导软件架构，将业务元素映射为系统元素，让系统本身成为最好的业务文档。在本案例中，我们选择餐厅作为业务场景，但不在意餐厅实现细节，而是以餐厅为主线故事，系统性的阐述DDD落地方法。希望读者能够从中吸取精华，去其糟粕，全文较长、耐心读完、必有收获。

1、领域设计

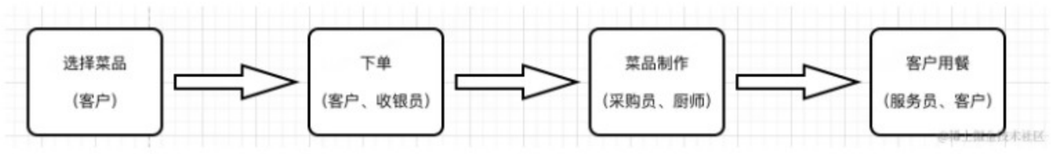
领域设计的核心是业务驱动的分而治之，旨在缩小软件系统与真实业务的差异，从而减少差异带来的问题。

当业务与系统之间存在差异时，我们无法将业务逻辑和程序逻辑对应起来，从而分不清区域，也分不清职责，因此会觉得混乱。就像你平时不会将枕头和被子放在厨房或卫生间一样，你的床上不会放着大米白面，否则你想睡觉是一件很复杂的事情，软件系统也是如此。

所以、首先要把业务分析清楚，然后设计与业务模型对应的软件模型，这就是DDD的核心思想。

1.1 宏观流程

假如我要设计一个餐厅，由于分而治之的需要，我会首先从宏观流程去分析，可以帮我们迅速找到重要的区域（这是功能相关性的初步划分）。



因此会得到几个明确的行为区域，我将餐厅划分为“菜品域”，“订单域”，“厨房域”，“用餐域”，这是宏观级别的领域划分，后续应该针对每个区域单独分析。

产出物是：宏观流程和参与角色

1.2 统一语言

语言贯穿于整个开发过程，从需求分析到设计、从设计到编码，因此好的语言非常重要，好的语言体现了清晰的业务概念。

在这个阶段，我们需要通过梳理，找到业务中都有哪些实体与行为，对其做一些归纳。我们的核心问题是“谁”通过什么“行为”影响了“谁”，其中的三个要素分别是“角色”、“行为”、“实体”，因此我的建议是先找到“角色”、“行为”、“实体”，并对他们归类，我常常关注角色以及具体身份、行为以及包含的重要步骤、实体以及具体实例。

角色：是施事主语、是名词，是主动发起行为的一类实体。

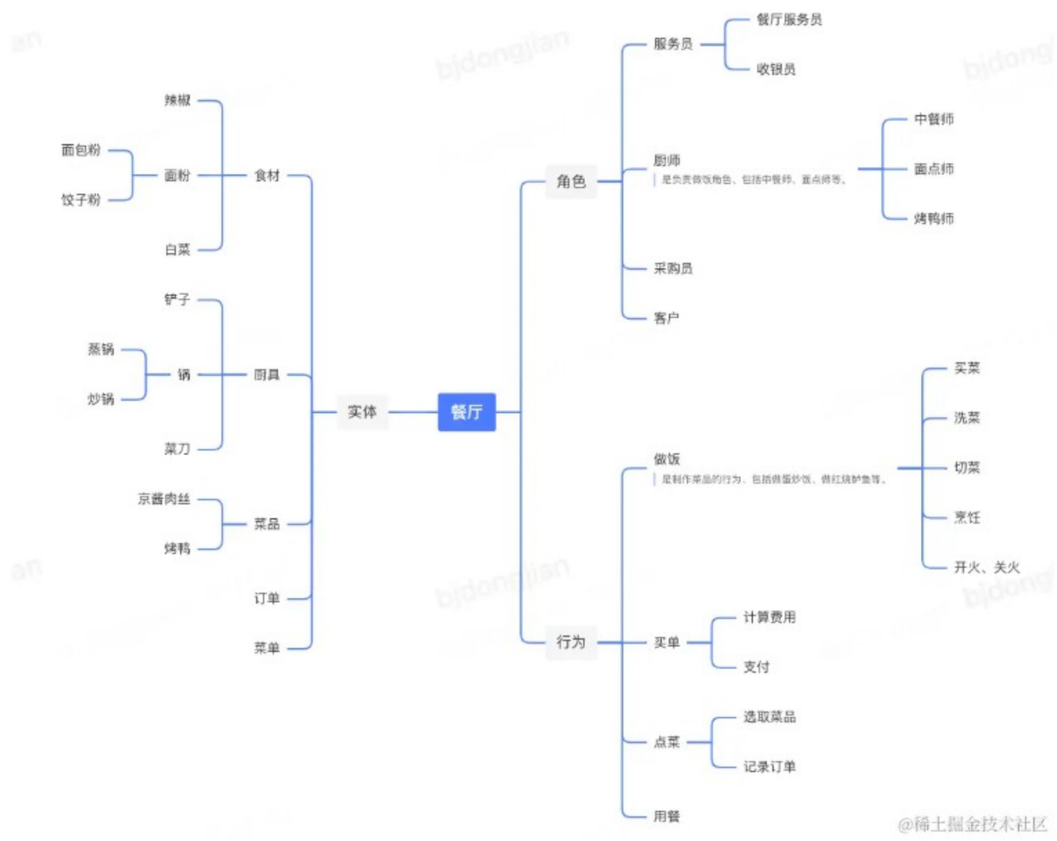
行为：是动词、是做了什么事情，是行为本身。

实体：是名词，是除“角色”之外的其他实体。

京东云开发者 LV.7
技术运营 @京东科技信息技术...
作者榜No.1 优秀作者
1.8k 文章 3.0m 阅读 19k 粉丝
关注 私信

推荐使用脑图画出来，我认为归纳后的脑图有助于我们识别根本要素，有利于抽象。

产出物是：名词、概念定义、相关脑图。

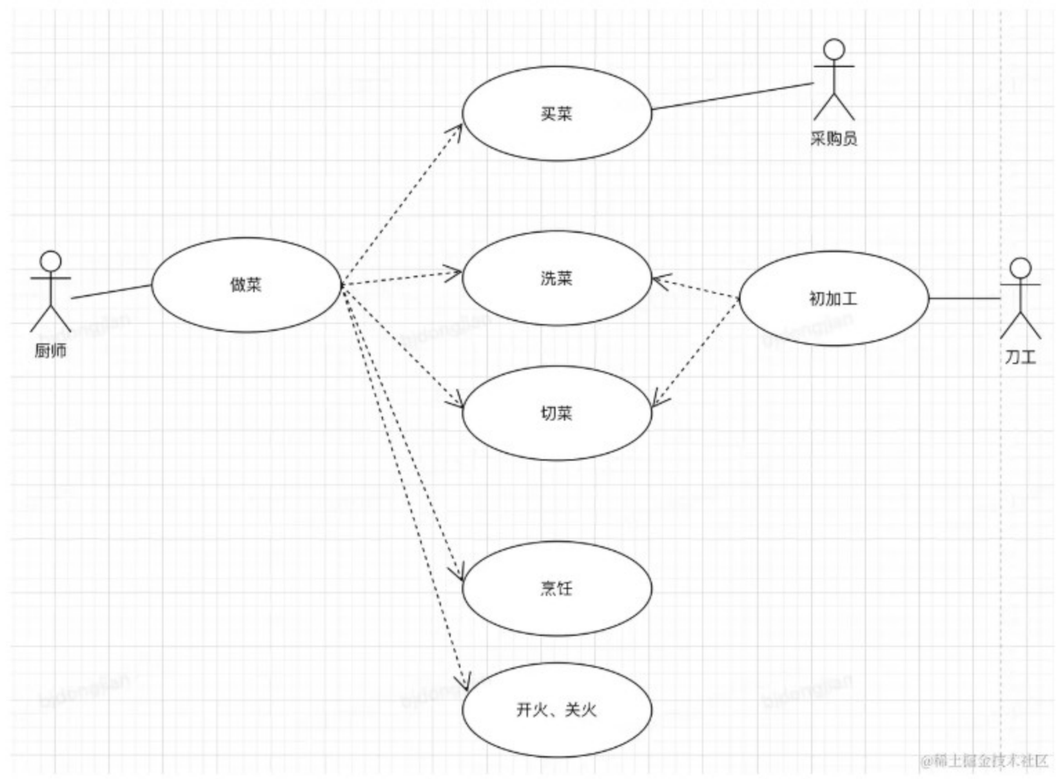


1.3 用例分析

在这一步、我们使用相对宏观的分析，不需要进入用例的细节分析，主要的目的是掌握角色与行为之间的关系，理清谁在做什么，角色的职责目的是什么，用于指导领域划分以及领域服务设计。

产出物：用例图

以做菜为例，如图



1.4 领域划分

我们在分析宏观流程时，划分了几个行为区域，那是宏观级别的。在那基础之上，我们需要拉进某个区域的视角，再结合之前的用例分析，按照“功能相关性”、“角色相关性”进一步划分领域。我们不仅要知道谁做了什么，还需要知道谁“在哪”做了什么。

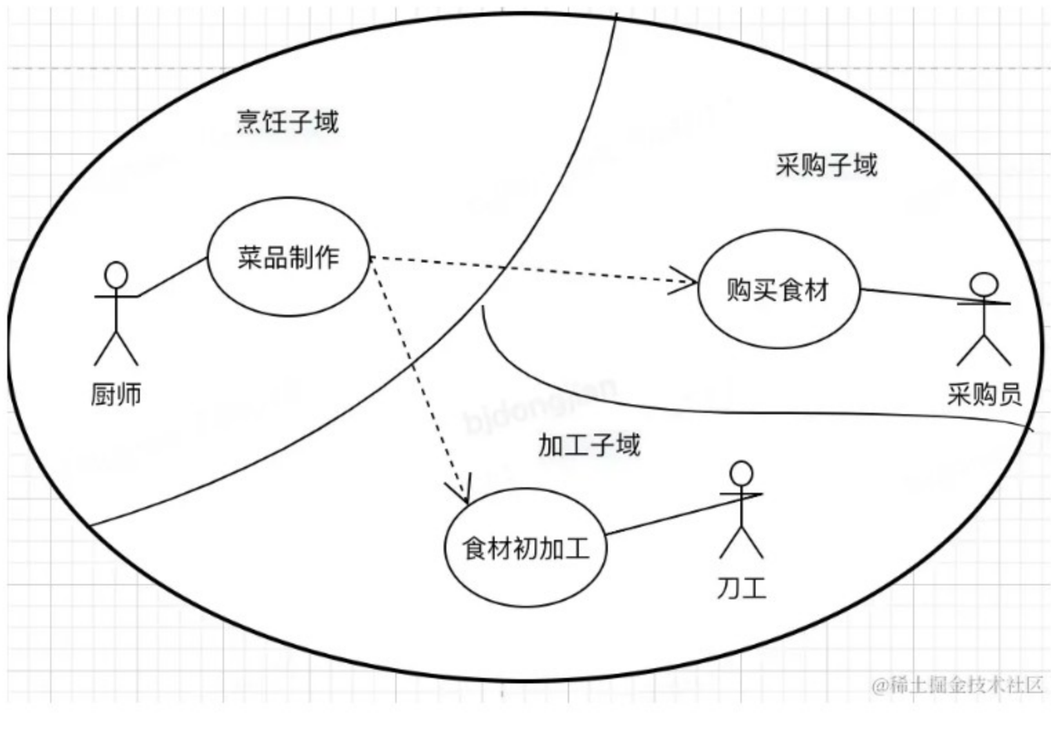
功能相关性：也称为业务相关性，业务是由一套用例组成的，一套用例之间是符合高内聚原则的，一套用例构成了一个问题空间，也就构成了一个领域，所以“功能相关性”是划分领域的黄金标准。例如与做菜相关的用例都应该归属于厨房，所以我们确认了厨房域，这也是很自然的事。在这一步，通过划分领域、梳理领域与用例之间的关系。

角色相关性： 角色相关性不可以作为首要参考因素，在特殊情况下用于划分子域，某个区域涉及多个角色参与，可以按照角色的分工，拆分为多个子域，从而满足不同角色的个性化需要。例如厨房的采购人员负责买菜、刀工负责切菜、大厨负责烹饪。我们就会考虑将厨房划分为“采购子域”、“加工子域”、“烹饪子域”。通常来说，子域不具备独立的问题空间，不会作为独立的领域存在。

划分领域的核心原则是保证领域的自治性（最小完备和自我履行），谨慎使用“实体相关性”划分领域，否则有可能将一个功能打散在多个领域上，违反了自治性原则，如果按照功能相关性划分，更容易实现领域的自治性，并且有助于将功能需要的实体聚合在一起。

产出物：领域、子域、领域与用例的关系

以厨房域为例，如图



在复杂业务时，可以使用事件风暴方法辅助分析，并输出上述产出物。

1.5 领域服务

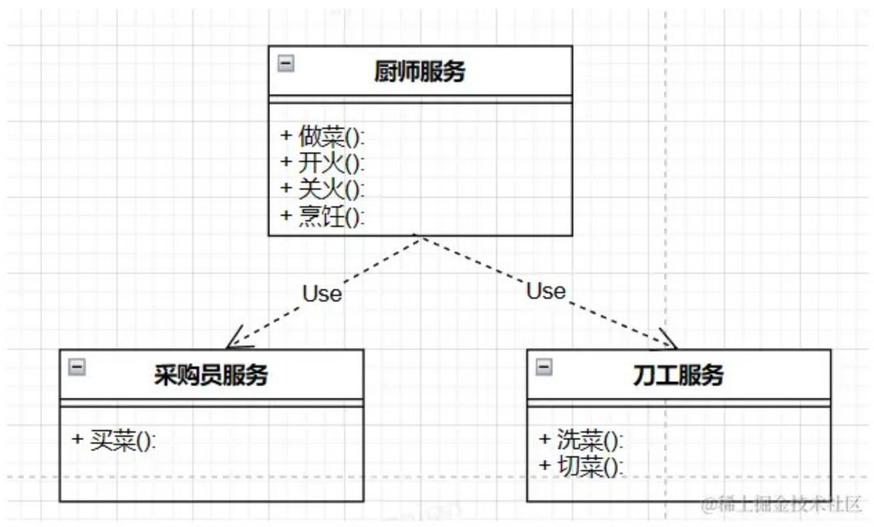
什么是领域服务？一个领域可以有几个领域服务？ 我们如何划分领域服务？标准是什么？

我认为一个领域不只有一个领域服务，我们不应该按照实体划分，也不应该按照聚合划分，也不该按照功能相关性划分。

领域服务用于实现用例功能，我认为应该使用角色划分领域服务。在用例图中，不同的角色发起不一样的用例，不同的领域服务提供不一样的用例，只有这样、才能确保领域服务是用例图的映射，也才能真正体现业务含义。领域服务是面向角色的，在一个领域中、每个角色对应一个领域服务。另外、同一个用例的逻辑差异是与角色的身份有关的，角色的身份对应了服务的泛化，角色的用例对应了服务的方法。对于此观点、我们在后续功能设计的部分也有体现。

例如：厨房域（厨师服务、刀工服务、采购员服务），菜品域（客户服务、管理者服务）。

产出物：领域服务类图



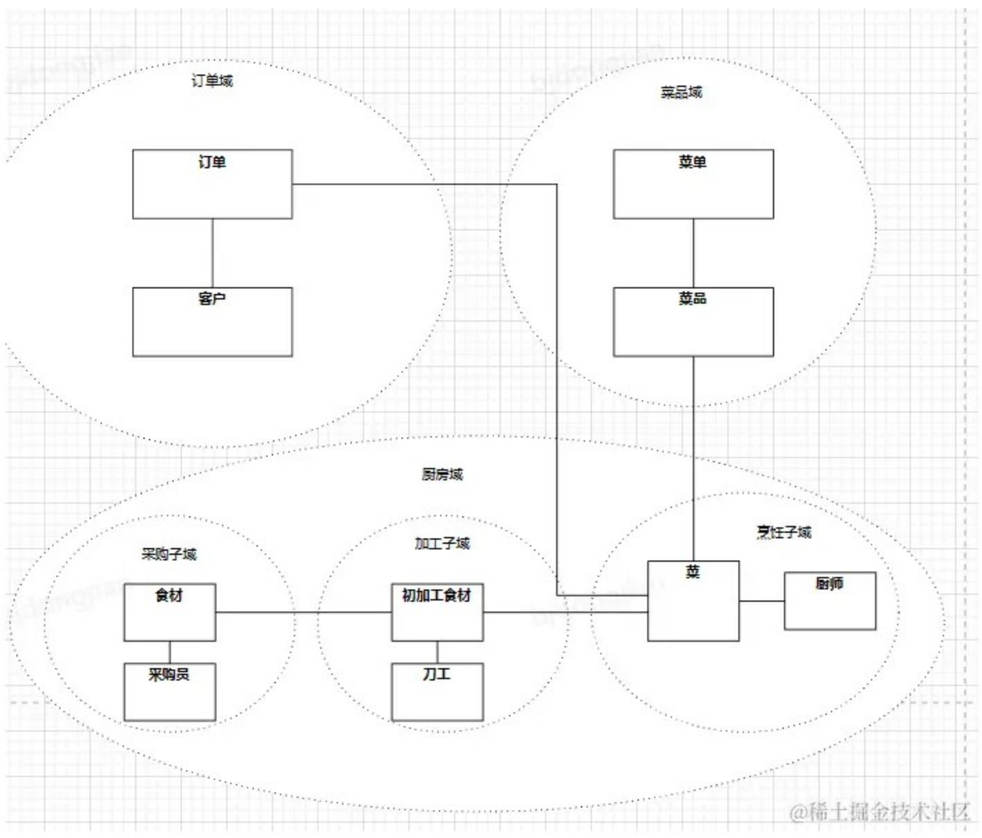
1.6 领域建模

我们思考一下，到底什么才是领域驱动设计？ 例如“厨房域”被称为“菜域”，“厨师”的“做菜”功能被称为“菜服务”的“做菜”功能，也例如“菜品域”有个“菜品服务”，“菜品服务”提供了“增、删、改、查”的功能。我们往往以最核心的实体为中心，误以为业务就是在操作数据，丢掉了业务本质含义，逐渐也就走歪了。

不要学传统的数据模型驱动设计，实体模型驱动设计与前者的本质是一样的，是换汤不换药的，这不是技术问题，而是过度集中在实体上以至于忘记其他元素。我们必须把精力放在业务本身，防止领域驱动设计变成领域模型驱动设计。我们不应该优先思考领域模型，不应该以领域模型命名一切，不应该让领域模型决定业务的实现方式。厨房不只有菜，也有服务员和厨师，我们使用合适的语言对应合适的元素，以确保软件元素是真实业务的映射。例如“厨师在厨房做菜”，这句话中的所有元素都要在系统中得以保留，丢了一个也不行，更何况只剩下菜了。

所以、我们先做领域划分，再做领域服务设计，最后做领域建模，这个顺序很重要，可以避免我们错误的以领域建模为中心。先有用例才有领域，先有领域才为领域建模，实体是为了实现一组用例存在的。而一组用例不一定依赖实体。

回到正题、我们在这一步的重点还是菜的问题，我们分析实体与领域之间关系（领域聚合），实体与实体的关系（OO聚合）。其中OO关系影响了功能的扩展性，需要我们特别关注。实体因一套用例而聚合在一起，我推荐做法是将领域的用例放在一起分析，找到他们的共同性，充分考虑变化，使用兼容性更好的模型解决问题。

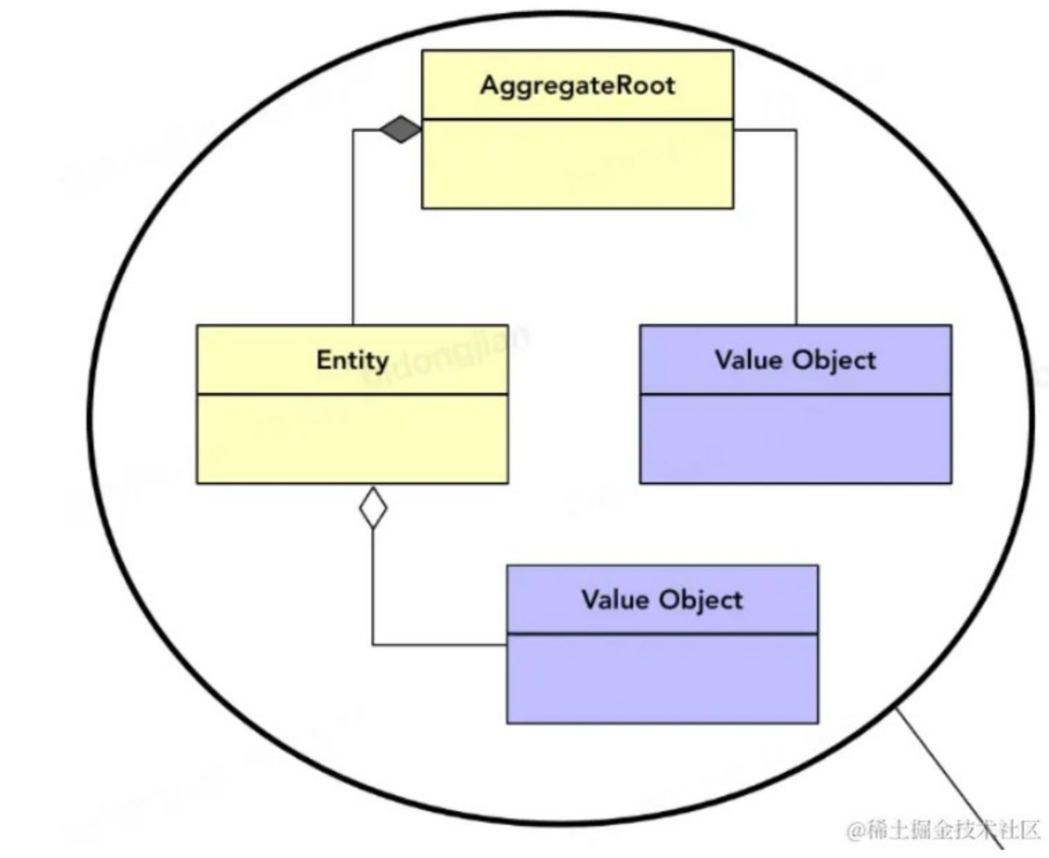


组合、聚合

聚合（aggregation）：聚合关系是一种弱的关系，整体和部分可以相互独立。

组合（composition）：组合关系是一种强的整体和部分的关系，整体和部分具有相同的生命周期。

可以使用如下案例，既能表达领域聚合，又能表达OO聚合的关系。



产出物：聚合、实体、值对象、实体的属性

1.7 领域上下游

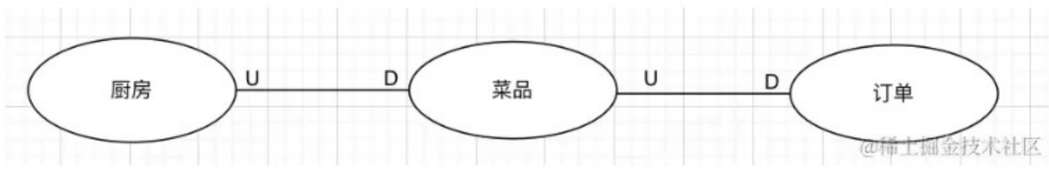
领域上下游关系，不是领域的依赖关系，依赖关系指的是能力的依赖，是共用了某些能力。领域上下游关系，也不是调用关系，调用关系是与用例相关的，不是用于描述领域处境的。

领域上下游关系指的是影响力的关系，上游影响下游，影响力分为“逻辑影响”和“数据影响”，一般说来我们更应该关注“数据影响”，因为上下游的逻辑影响也是靠数据传递的，所以领域上下游关系是一种数据流向的限制，是业务发生的顺序限制，用于规定该领域所使用的数据，是下游领域依赖上游领域“准备就绪”的体现。合理的上下游限制，有助于减少领域之间的不必要依赖和重复的计算。

领域上下游是与场景相关的，并不是一成不变的，不同场景的情况下，存在不同的上下游关系，各场景应该独立说明。

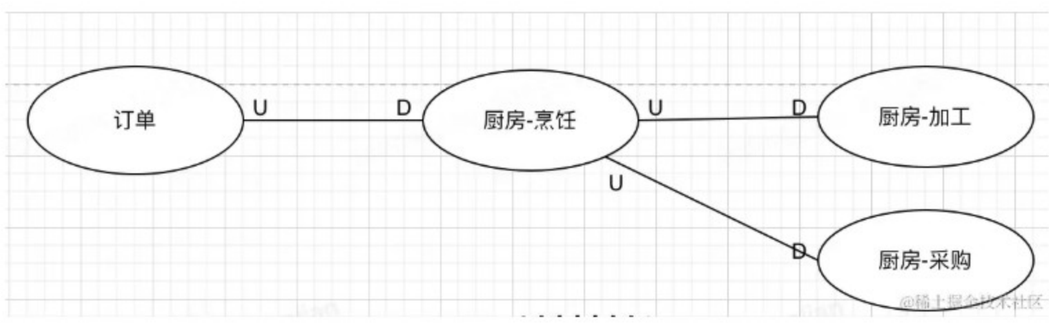
产出物：各场景的上下游说明

例：在【菜品管理】场景下



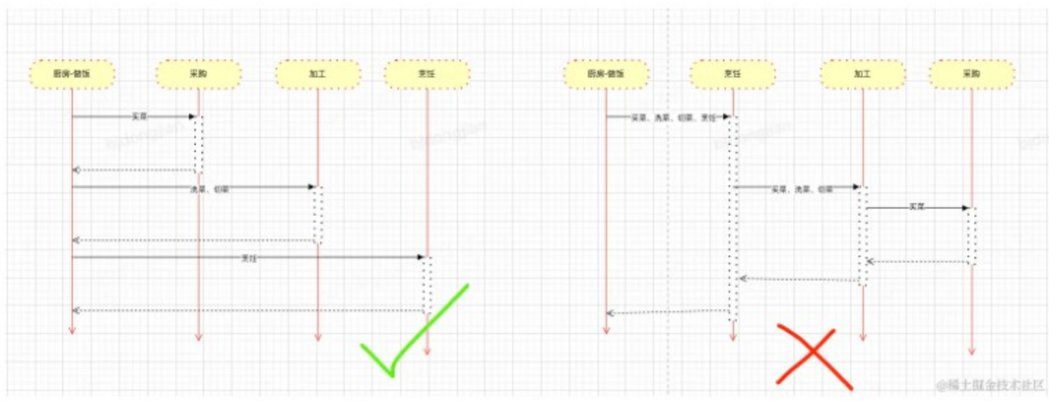
如果厨房的某些食材不足了，或者某个厨师休假了，就会影响到菜品的展示，从而影响到客户的订单。

例：在【客户消费】场景下



客户的订单、影响厨房生产的菜，从而影响刀工的行为，也影响到了采购。

请对比下面两个图，用于理解领域的上下游



实际上，厨师不应该依赖采购人员的采购功能，也不依赖刀工的切菜功能，他只是依赖“初加工食材”而已，而“初加工食材”就是被处理好的数据，厨师在做饭时，“初加工食材”就已经被处理好了，上面的图例只是为了说明一个关于领域上下游的问题，这是业务**发生顺序**以及**数据来源**的问题。

我们常常使用领域事件串联业务流程，在使用领域事件时，不止要关注点对点的解耦，更应该使业务流程符合领域上下游限定，让各个领域独立运行。

顺序发生优于嵌套发生，数据依赖优于功能依赖。

2、架构设计

架构设计是为了解决软件系统复杂度带来的问题，找到系统中的元素并搞清楚他们之间关系。

架构的目标是用于管理复杂性、易变性和不确定性，以确保在长期的系统演化过程中，一部分架构的变化不会对其它部分产生不必要的负面影响。这样做可以确保业务和研发效率的敏捷，让应用的易变部分能够频繁地变化，对应用的其它部分的影响尽可能地小。

架构设计三原则：合适原则、简单原则、演化原则

2.1 分层架构

我们需要按照 接口层、领域层（领域用例层、领域模型层）、依赖层、基础层 构建架构模型。

接口层：为外部提供服务的入口，是适配层的北向网关。不实现任何业务逻辑，也不处理事务，是跨领域的，是流程编排层，是门面服务。

领域用例层： 是领域服务层，是领域用例的实现层、隶属于某个领域、是业务逻辑层，是事务层，业务逻辑应该在这层完整体现，不要分散到其他层级。

领域模型层： 是领域模型（实体、值对象、聚合）的所在位置，专注于领域模型自身的能力，不包含业务功能，可以处理事务，是原子化的能力，是领域对象的自我实现 。

依赖层： 是连接外部服务的出口，是适配层的南向网关。包括仓储，端点、RPC等，主要作用是领域和外部解耦，是跨领域的。

基础层： 与业务无关的，与领域无关的，通用的技术能力，技术组件等。

2.2 架构映射

架构的视角，从大到小依次是：系统->应用（微服务）->模块（包）->子模块 这样的从大到小的层级。

业务领域映射：我们将划分好的领域，按照对应的视角映射为对应的元素，领域模型映射到架构模型时，应该是视角对等的，如果餐厅是系统、那么厨房就是应用，如果餐厅是应用、那么厨房就是模块。也应该是层级匹配的，将用例的实现映射到用例层，将领域模型的实现映射到领域模型层。也应该是名称一致的，将领域名映射为应用名或包名，将实体名映射为实体类名，将角色名映射为领域服务类名，将角色身份名映射为服务类的子类名，将用例名映射为服务类的方法名。

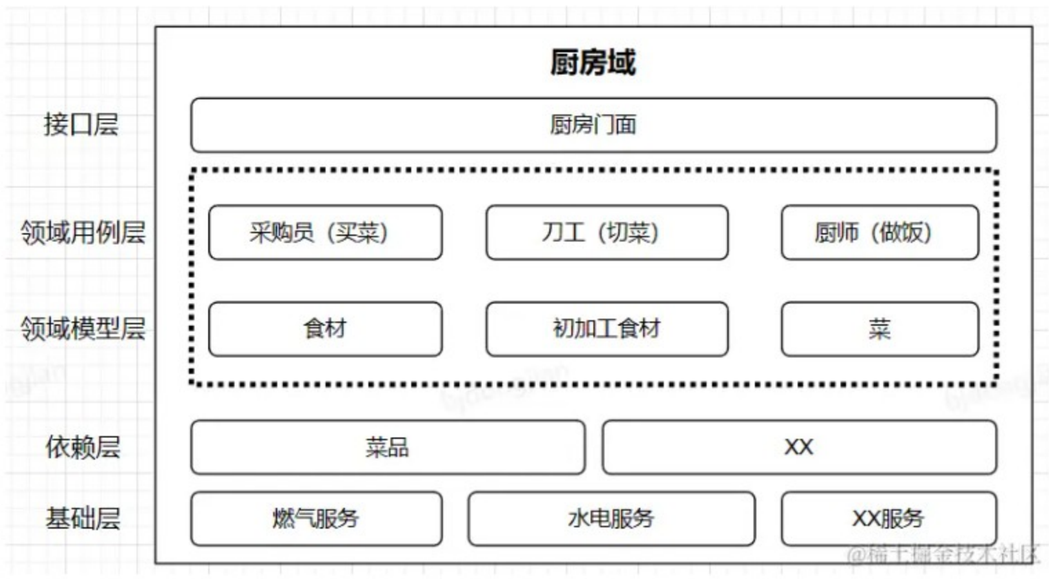
技术和抽象问题：有时候、业务领域分析不能体现那些共性的技术问题，所以需要适当结合技术视角，可能需要对领域模型微调。同时、我们需要找到共同需要的基础能力，例如“水”、“电”、“煤气”等等，将这些作为额外的考虑因素，要做到业务问题与技术问题解耦，不要将技术问题和业务逻辑揉成一团。

领域设计，类似餐厅设计师，他设计餐厅有几个区域，区域的用途是什么。

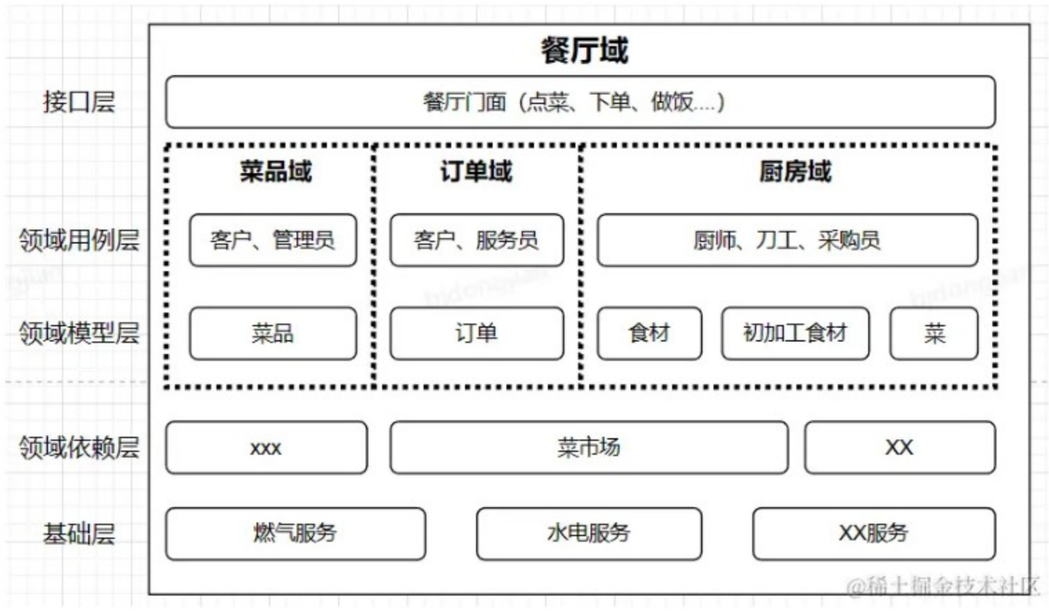
架构设计，类似建筑设计师，他设计如何走水电煤气、如何施工等。

产出物：分层架构图

以厨房为视角，其架构如下



以餐厅为视角，其架构如下



分层架构图，体现逻辑上的层级分布，而不是代表组件的具体含义，组件是应用还是模块、需要结合实际情况而定。

2.3 必要的约束

1、分层架构越往下层就越是稳定的：下层是被上层依赖的，下层不可以反向依赖上层（扩展点除外）。因为分层架构的核心原则是将容易变化的逻辑上浮，将共性的、原子化的、通用的逻辑下沉，被依赖的下层应该是稳定的，这要求上层承接更多业务变化。下层离开上层应该是可以独立存在的，例如在接口层定义的DTO不可以在下层被使用，但领域层定义的实体可以被上层使用。

2、在使用充血模型时，应该符合面向对象编程原则：不要随意的将一些能力都充到领域实体模型中。以“菜”为例，重量和规格是“菜”的自身的属性，激发味蕾是“菜”的能力，“菜”可以维护自身的持久化状态。但是、请注意、“菜”不可以“炒菜”，因为“炒菜”的时候，“菜”还没有出现呢，“菜”不是自己的上帝，“菜”需要被做出来，所以“菜”被做出来之前是没有“菜”的，这是个时间上的概念，不要错把“炒菜”的能力放在“菜”的身上。“炒菜”用到的“水+电+气+食材+调料+厨具”不应该是“菜”的属性范围，这些元素都在“厨房”的范围中，不要让领域的模型包含不属于自身的元素，领域的实体模型只是领域的一部分，只用于实现通用的模型能力。

3、接口层和依赖层是与领域无关的：他们是与技术相关的层级，不属于任何领域，这两层不能包含业务逻辑。有时候我们可以把接口层拆为两层（接口层、应用层），但是我不建议这样做，我们没有必要把很轻的一层再次拆分。我们也可以把依赖层拆分为两个（领域模型依赖、其他依赖），我非常建议这样做，因为领域模型依赖的资源不会被其他领域使用，拆开之后可以有效限制领域模型的依赖，

4、领域层是与环境无关的：无论某个领域是应用还是模块，都应该是完整的。应该具备独立的用例层和独立的模型层，即使多个领域在同一个应用当中，也要按照他们是分别独立去看待，无论某个领域是应用还是模块，领域对外部的交互，不可以绕过依赖层和接口层。

5、领域应该自治性的：把一个领域拆分为子域、子子域..... 无限拆分，子域就不完整了。或者没有按照功能相关性拆分，也可能破坏领域的完整性，不完整的领域不符合自治性原则。所以、不完整的领域不会单独存在，所以、当一个领域的内部子域不具备独立性时，子域之间不必严格解耦，不需要通过依赖层访问本领域的其他子域，他们之间可以直接调用。

6、领域用例层和领域模型层是两个层级：领域用例层指的就是领域服务层，不建议将领域服务与领域模型放在同一层，这可能会导致逻辑的分散（一部分在领域服务层、一部分在领域模型层）。如果将业务逻辑写在领域模型中，会导致业务逻辑进一步下沉，业务逻辑的不确定性太大，是不适合下沉的，是违反分层架构原则的。领域模型对应的是实体、领域服务对应的是用例，分开就是更有效的限制措施。

7、领域用例层只能承接符合自身领域的用例：我们划分出领域的目的，就是为了区分每个领域的职责所在，因此他们必须严格按照职责办事，我们在之前已明确了用例和领域之间的关系，需要严格遵守。 如果出现跨领域的编排，请在接口层串联。如果依赖其他领域的功能，请把被依赖的功能逻辑放在其他领域中。

8、领域模型层遵循最小依赖原则：只可以依赖必要的资源，必要资源指的是领域模型实现自身能力需要的资源，不包括实现业务逻辑依赖的资源。例如领域模型需要依赖DB完成持久化，可以依赖数据访问资源，但不应该依赖其他领域资源、不可以依赖RPC资源等。 最好的做法就是将领域模型依赖的资源单独拿出来，并且与领域模型放在一起。保持领域模型层的独立性，在多个领域应用共享领域模型时，方便使用共享内核的设计模式。

2.4 微服务划分

服务划分以领域划分为参考，主要看我们要拆分到什么粒度，不建议将几个领域放在同一个服务中，不建议把一个完整的领域拆分为几个不完整的微服务。

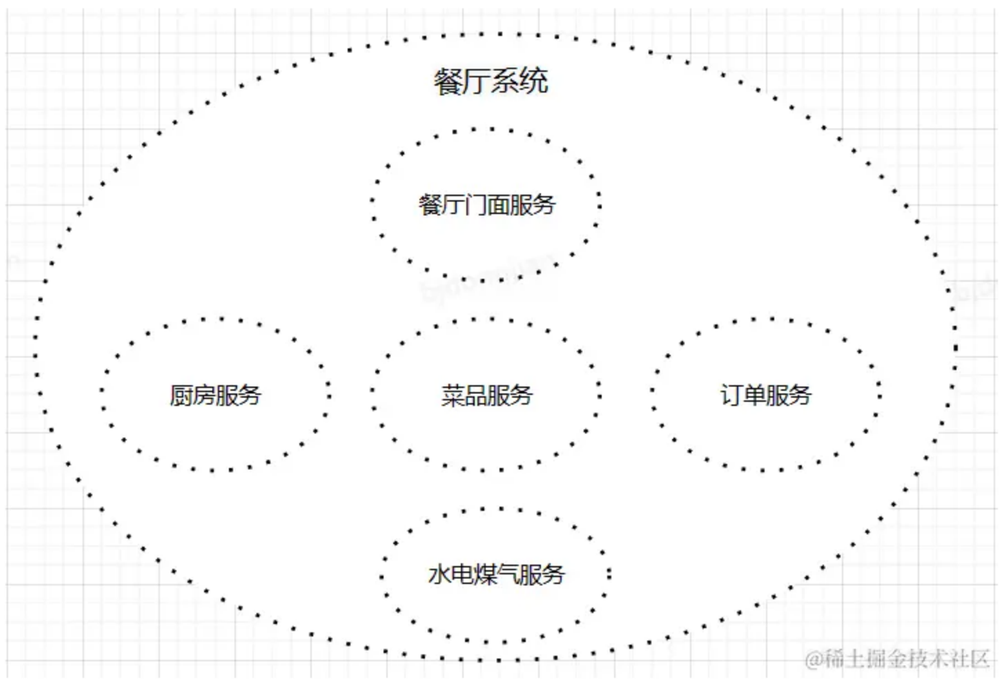
产出物：微服务

例如餐厅：是有必要拆分的，餐厅的“菜品域”，“订单域”，“厨房域”有独立的问题空间，是具备自治性的。

例如厨房：是没有必要拆分的，厨师与刀工的耦合非常高，他们都在做饭，分开之后是不完整的，分开就是没有必要的。

所以餐厅被拆分为：**厨房、菜品、订单**，三个微服务。基于此、我们单独拿出**餐厅门面服务**作为接口层应用，再单独拿出**餐厅基础服务**作为水电煤气的应用。

一般情况下，依赖层不会作为单独的服务提供，会被以组件的形式嵌入到其他服务中。



3、功能设计（用例实现）

如果说领域设计是餐厅的设计师、架构设计是餐厅的建筑师、那么功能设计就是餐厅的厨师。

任何设计都要落地到功能设计，如果厨师不守规则，偏偏要去洗手间洗菜，最后的结果依然是一团乱，最终会导致前面的所有设计泡汤。

功能设计是实现 “面向扩展开放、面向修改关闭” 的途径，

功能设计是为研发提供的落地支撑。

3.1 功能的概念

功能迭代时，功能会发生一些变化，所以他的含义是可能变化的，所以我们需要再次审视功能的概念，及时加以调整。

例如、我们实现了一个“做蛋炒饭”的功能，后来又实现了一个“做辣椒炒蛋”的功能，那么我们应该将功能升级为“炒菜”，甚至是“制作菜品”等。

结合相关功能，系统性思考和抽象，明确功能的概念，是功能设计的前提。

产出物：更新语言库，更新脑图

3.2 用例的位置

我们在1.3用例分析章节，明确了用例与角色的关系，在1.4领域划分章节，明确了用例与领域的关系。

然而一个新功能的加入，我们仍然要再次评估，以确保他处于正确的位置。按照之前的做法,根据功能相关性确认用例的领域，根据角色相关性确认用例的领域服务。

产出物：更新用例图

3.3 事件风暴

事件风暴常用于梳理业务流程，适用于解构跨领域的复杂业务，感兴趣的朋友可以去自行学习。

但是、对于领域内的单功能，稍有复杂的时候，我们可以采取简化版事件风暴的方法，从而获得如下信息：

将功能拆分为多个子功能（步骤）。(在后续使用)

步骤对应的角色+角色身份。(在后续的3.6章节落地)

步骤的串联流程+领域事件。(在后续的3.6章节落地)

步骤依赖的实体。(在后续的3.7章节落地)

产出物：事件风暴模型

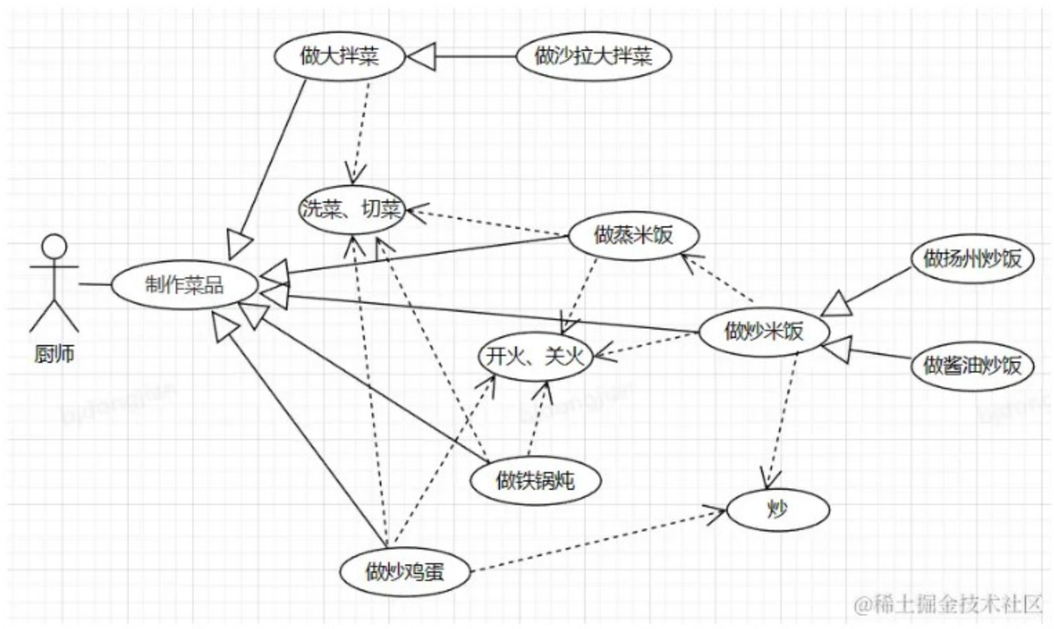
3.4 用例分析

我们暂且收回思路，首先要关注共性和差异问题，以实现功能复用或扩展。

- 确认用例的泛化+差异点，实现功能的扩展。
- 寻找共同包含的步骤，实现逻辑的复用。

产出物：用例分析图

例：制作菜品（做大拌菜、做铁锅炖、做炒鸡蛋、做蒸米饭、做炒米饭）

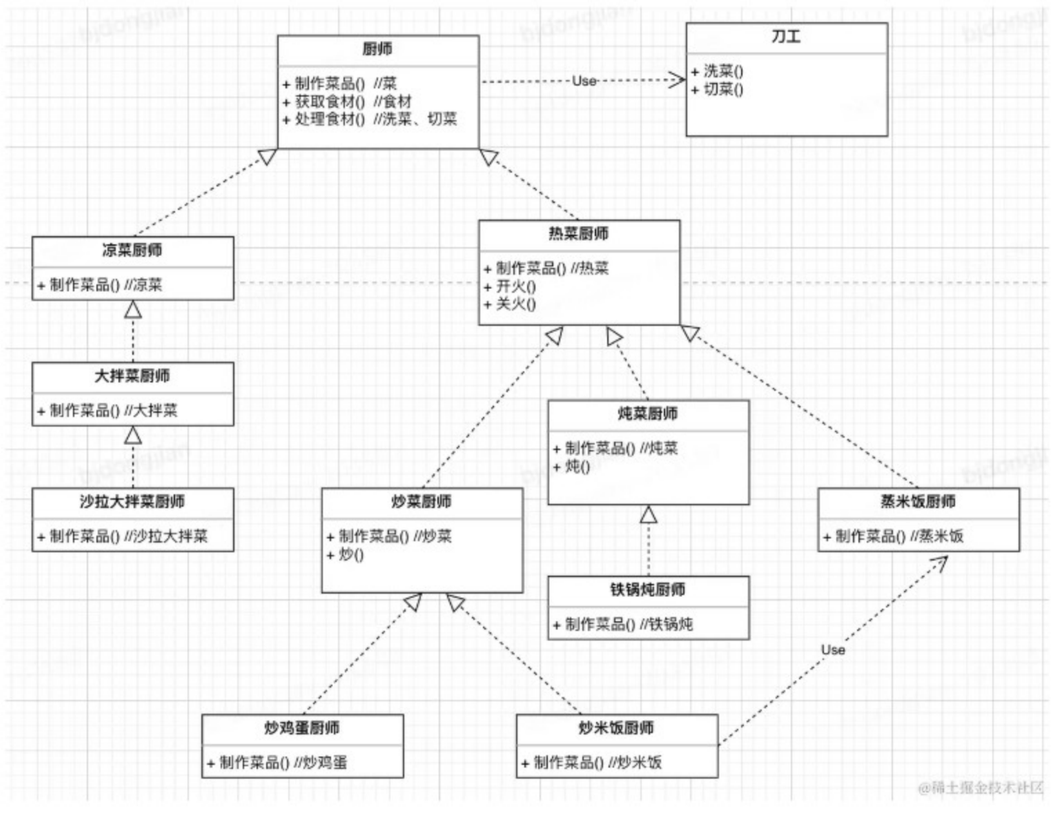


3.5 用例实现类（领域服务类）结构图

首要关注点是领域服务类的结构问题，结构决定了扩展，我们需要先达到“面相修改关闭，面相扩展开放”的目的。

领域服务的类结构图是用例图的映射，服务类结构图反向映射了角色的身份，进一步反向印证了上文的观点。

出物：用例层的类结构图



3.6 用例流程图

我们接回思路，更进一步，将事件风暴模型落实到代码层面。

我们将步骤分配到实现类中、步骤就是该类的一个方法，进一步明确由哪个类和方法来实现该步骤，从而就规定了步骤所在的领域服务。再将步骤和领域事件串联起来，规定了业务实现流程。

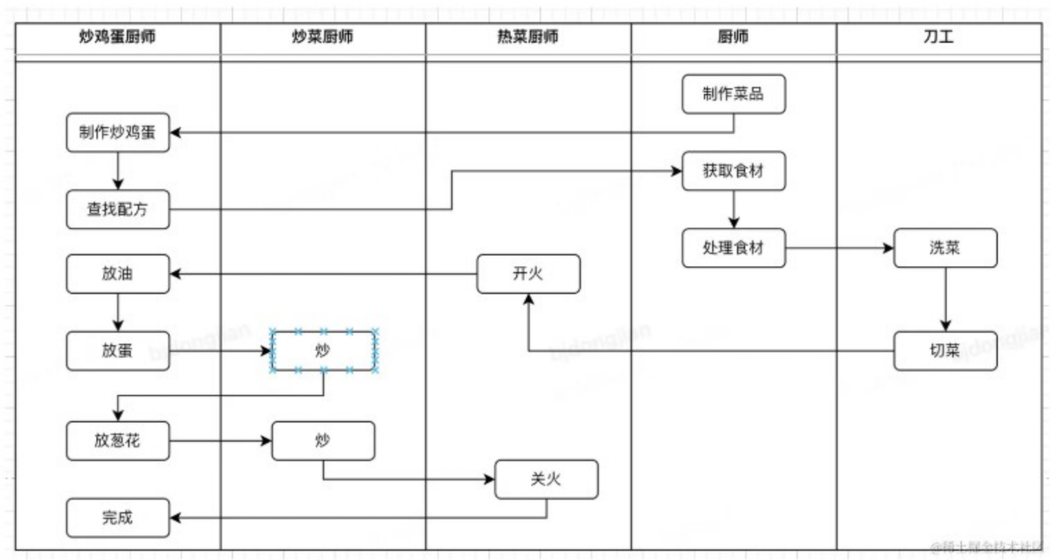
在确认步骤所在位置的时候，根据角色身份相关性定位步骤的具体实现类。

推荐使用泳道图表达上述内容，泳道的纵向组件是领域服务类，领域服务承接了所有子功能，流程图也需要体现所有的步骤，这是用例层的横向交互。

程序流程就是业务流程的映射，步骤分布体现了角色身份的差异。

产出物：用例流程图

以炒鸡蛋为例，其用例流程图如下

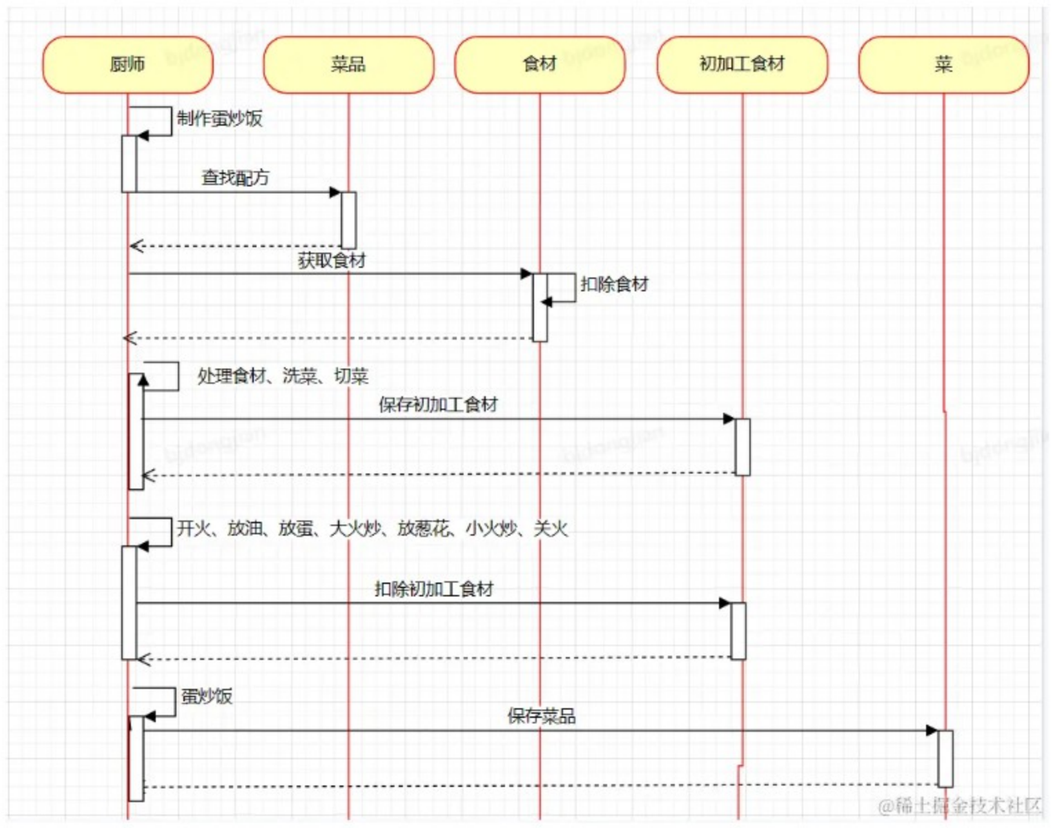


3.7 活动图（时序图）

进一步将事件风暴模型落实到代码层面，我们使用时序图，体现依赖和调用关系，规定了步骤与领域实体模型的关系，说明该步骤影响了谁。

时序图体现了领域服务内部的纵向交互，为了简便、我们可以收起领域服务类（用例层）的泳道。

产出物：时序图、活动图



在本篇文章中，通过三大步骤阐述了映射办法，让软件系统成为真实业务的说明书，软件系统似乎在对我们说"谁？在哪？做了什么事？影响了谁？是怎么做的？有什么差异？"。例如我们画的圈成为了应用名或包名，圈中的领域模型图成为了实体类+数据模型，圈中的用例图成为了领域服务和方法，功能流程成为了程序调用链，功能步骤成为了方法，领域服务类结构反向体现了角色身份，也体现了不同身份的差异..... 系统就是业务、业务就是系统、两者可以相互映射。

DDD的概念有很多，到底什么是DDD？是思想吗？是方法论吗？每个人都有自己的理解。在我看来、DDD是一套系统化的办法，无法用几句话说清楚，故而以此分享DDD的落地模式。

标签： 后端

评论 3

登录 / 注册 即可发布评论！

最热 | 最新

Renounce
写的不错
6月前 点赞 评论 ···

威哥爱编程 华为HDE，公众号：威哥爱编程
好文，想一句话说清楚 DDD 确实不容易
6月前 点赞 评论 ···

用户70758502408
mark
7月前 点赞 评论 ···

为你推荐

DDD落地指南
码农戏码 | 3年前 | 2.2k | 4 | 2 | 领域驱动设计

架构师必备 - DDD之落地实践
二进制狂人 | 1年前 | 61 | 点赞 | 评论 | 后端 | 面试

架构师必备 - DDD之落地实践
Java基尼太美 | 2年前 | 66 | 点赞 | 评论 | 后端 | Java

可落地的DDD(4)-如何利用DDD进行微服务的划分(2)
方丈的寺院 | 5年前 | 3.5k | 24 | 12 | 架构

.NET现代应用的产品设计 - DDD实践
MASA技术团队 | 2年前 | 51 | 1 | 评论 | 领域驱动... | .NET

DDD之5限界上下文-定义领域边界的利器
李福春 | 4年前 | 2.2k | 3 | 评论 | Java

[微服务与DDD]-什么是DDD
Wise技术人生 | 2年前 | 1.8k | 2 | 评论 | 领域驱动设计

【实践篇】教你玩转微服务--基于DDD的微服务架构落地实践之路
京东云开发者 | 1年前 | 6.5k | 36 | 1 | 架构 | 微服务 | 掘金·金...

DDD的领域概念们
luoxn28 | 3年前 | 1.2k | 2 | 评论 | 领域驱...

架构师必备-DDD之落地实践
JavaGPT | 1年前 | 108 | 点赞 | 评论 | Spring Boot

DDD-实战解析
重庆穿山甲 | 1年前 | 139 | 1 | 评论 | 后端

一文带你落地DDD
柏炎 | 3年前 | 30k | 339 | 49 | 领域驱动... | 后端

DDD落地路线图
zh740 | 1年前 | 207 | 点赞 | 评论 | 领域驱动设计

DDD-CQRS的落地案例
方丈的寺院 | 5年前 | 5.0k | 14 | 1 | 架构

可落地的DDD(3)-如何利用DDD进行微服务的划分
方丈的寺院 | 5年前 | 5.2k | 25 | 评论 | 架构