## **限界上下文**

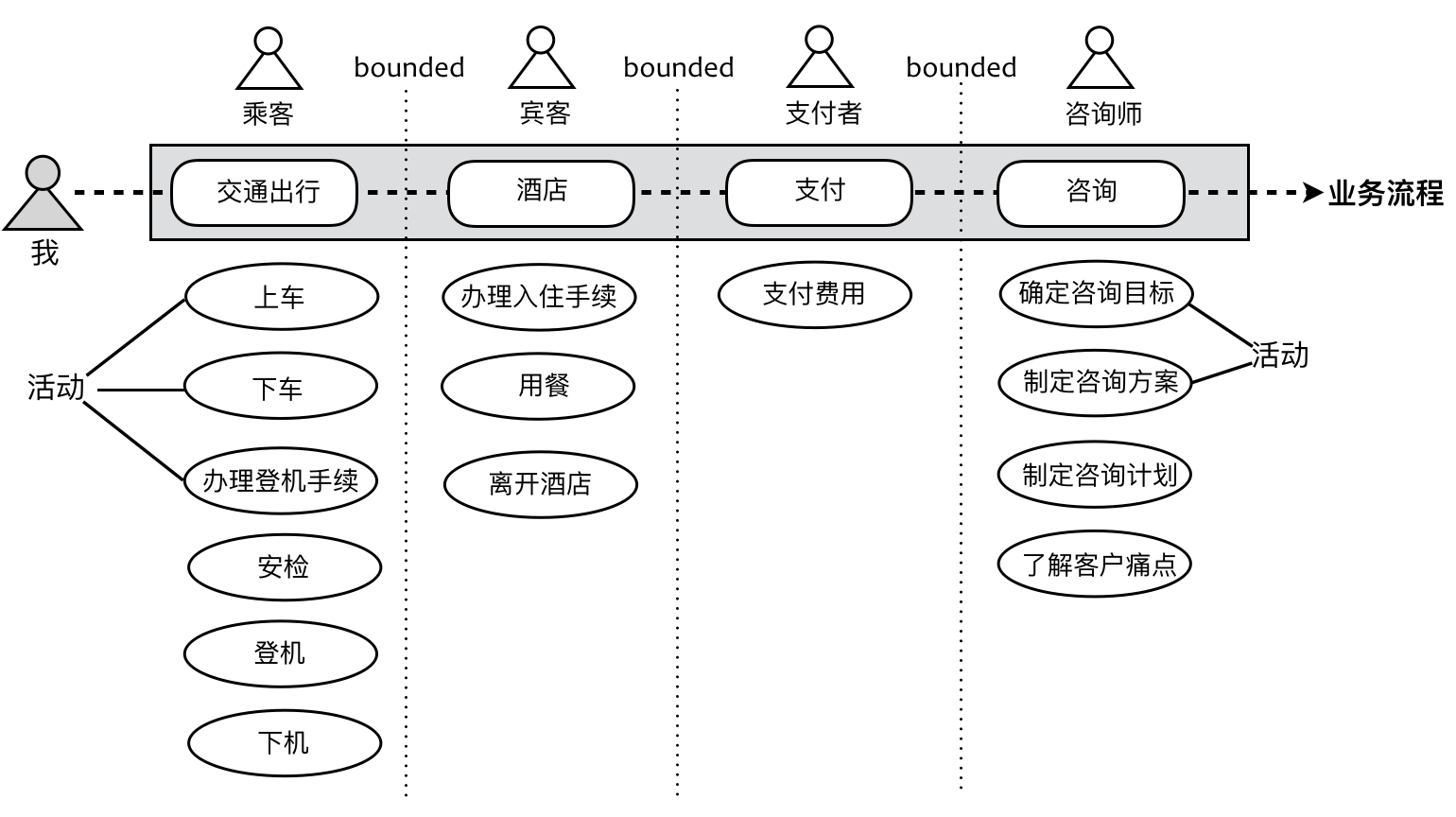
语言的极限就是世界的极限。——维特根斯坦

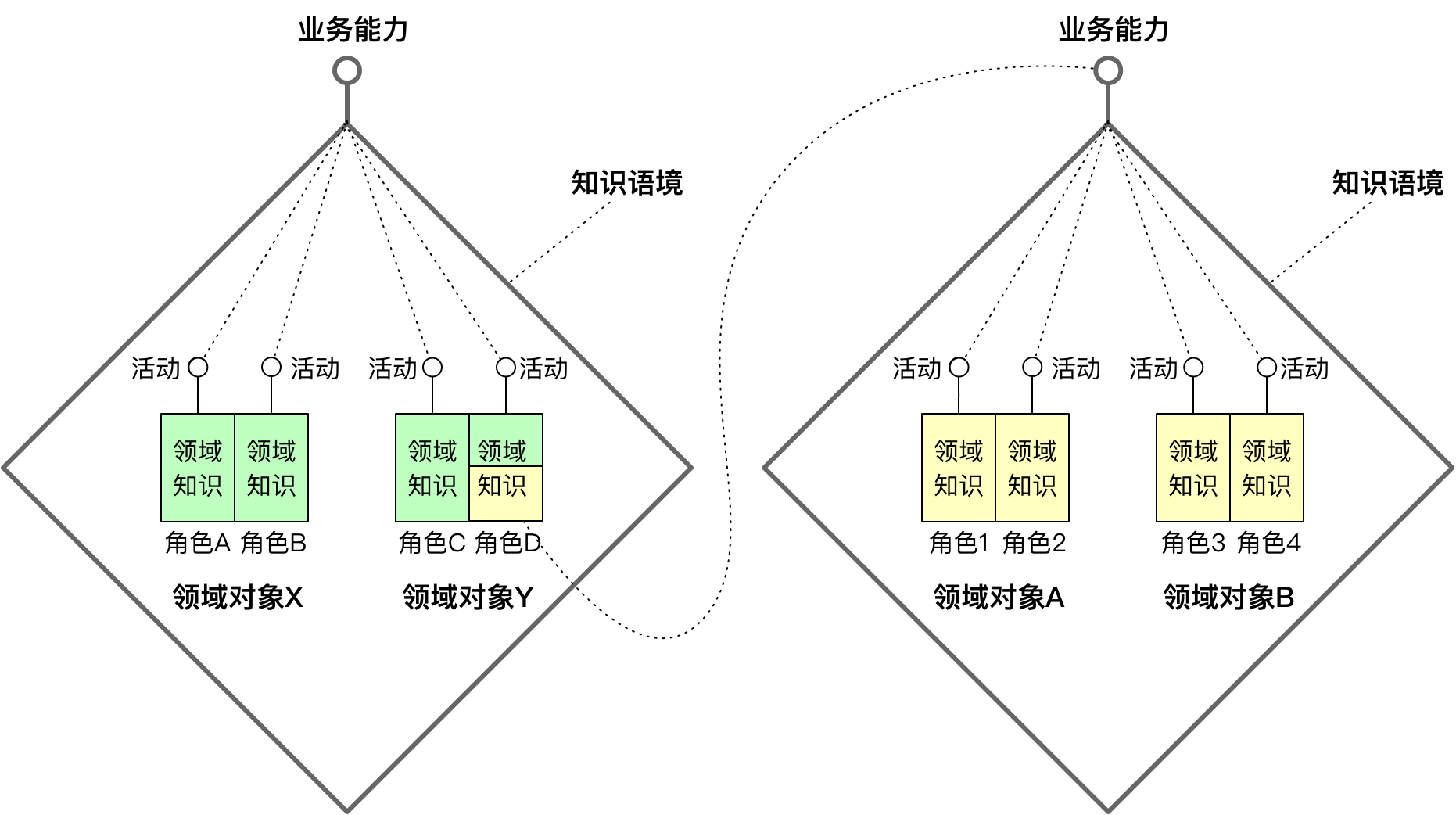
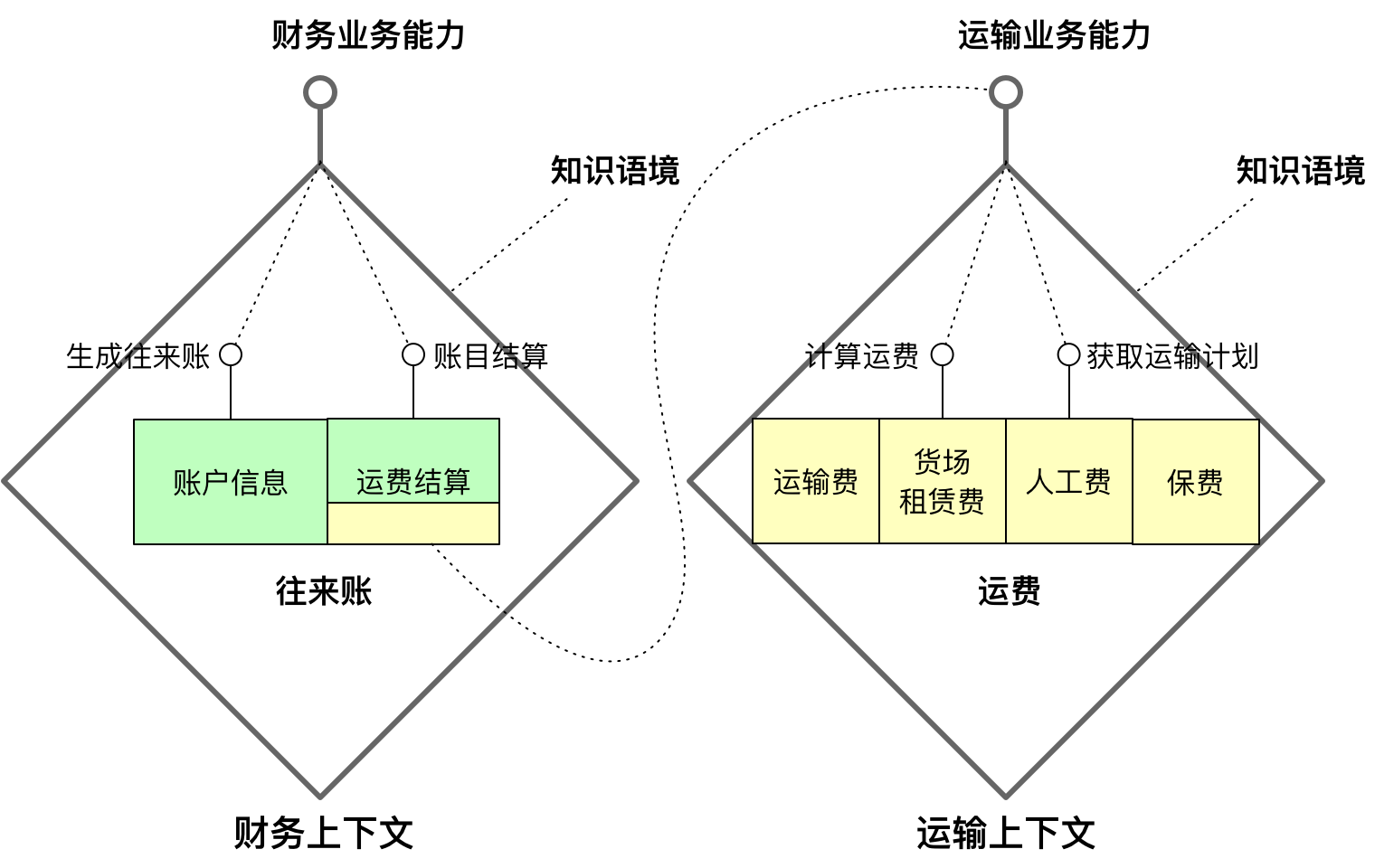
### **限界上下文的定义**

什么是限界上下文（Bounded Context）？我认为，要明确限界上下文的定义，需要从**Bounded**与**Context**这两个单词的含义来理解。**Context**表现了业务流程的场景片段。整个业务流程由诸多具有时序的活动组成，随着流程的进行，不同的活动需要不同的角色参与，并导致上下文因为某个活动的执行发生切换。因而，**上下文（Context）其实是动态的业务流程被边界（Bounded）静态切分的产物**。

假设这样一个业务场景。我作为一名咨询师从成都出发前往深圳为客户做领域驱动设计的咨询。无论是从家乘坐地铁到达成都双流机场，还是乘坐飞机到达深圳宝安机场，再从宝安机场乘坐出租车到达酒店，我的身份都是一名乘客（Passenger），虽然因为交通工具的不同，参与的活动也不尽相同，但无论上车下车，还是办理登机手续、安检、登机以及下机等活动，都与交通出行有关。

那么，是否我坐在交通工具上，就一定代表我属于这个上下文？未必！注意，其实交通出行上下文模糊了“我”而强调了“乘客”这个概念。这一概念代表了参与到该上下文的“角色”，或者说“身份”。我坐在飞机上，忽然想起给客户提供的咨询方案还需要完善。于是我拿出电脑，在一万米高空上继续完善我的领域驱动设计咨询方案。此时的我虽然还在飞机上，身份却切换成了一名咨询师，执行的业务活动也与咨询内容有关，当前的上下文也就从出行上下文切换到了咨询上下文。当我作为乘客乘坐出租车前往酒店，并至前台办理入住手续时，我又“撕下了乘客的面具”，摇身一变而为酒店上下文的宾客角色，上下文随之切换为住宿上下文。次日清晨，我离开酒店前往客户公司。随着我走出酒店这一活动的发生，住宿上下文又切换回交通出行。当我到达客户所在地时，面对客户，我开始以一名咨询师身份与客户团队交谈，了解他们的咨询目标与现有痛点。我制定咨询计划与方案，并与客户一起评审咨询方案，上下文又切换为咨询工作了。无论是交通出行还是入住酒店，都需要支付费用。支付的费用虽然不同，支付的行为也有所差别，需要用到的领域知识却是相同的，因此支付活动又可以归为支付上下文。

上下文在流程中的切换犹如电影画面的场景切换，相同的人物扮演了不同的角色，在不同的上下文参与了不同的活动。由于活动的目标发生了改变，履行的职责亦有所不同。上述场景如下图所示：   
  
整个业务流程由诸多活动组成，这些活动有着不同的角色参与。在每个上下文中，角色与角色之间通过活动产生协作，以满足业务流程的需求。这些活动是分散的，活动的目标也不相同，但**在同一个上下文中，这些活动却是为同一个目标提供服务**。

显然，每个限界上下文各自代表了不同的**业务能力（Business Capability）**，以满足参与到当前上下文中的各个角色的目标。这些角色只会执行满足当前限界上下文业务能力的活动，这是因为限界上下文划定了**知识**的边界，不同的限界上下文需要不同的领域知识，形成了各自的**知识语境**。业务能力与领域知识存在业务相关性，要提供该业务能力，需要具备对应的领域知识。领域知识由限界上下文的领域模型对象所拥有，或者说，这些领域模型对象共同提供了符合当前知识语境的业务能力，并被分散到对象扮演的角色之上，由它履行的活动来体现。如果该角色执行该活动却不具备对应的领域知识，说明对活动的分配不合理；如果该活动的目标与该限界上下文保持一致，却缺乏相应知识，说明该活动需要与别的限界上下文协作：   
  
一个物流运输系统能够支持集装箱在铁路运输与公路运输的多式联运。系统需要计算每次多式联运的运费，以管理公司与委托公司之间的往来账。那么，运费计算应该放在财务上下文吗？如果从“知识”和“能力”的角度去理解，财务上下文的领域模型对象并不具备计算运费的业务知识，它们并不了解运输过程中的各种费率，如运输费、货站租赁费、货物装卸人工费、保费，也不了解运输费用的计算规则。缺乏这些知识，自然也就不具备计算运费的能力。财务上下文其实只需要获得与往来账有关的结算费用，而不是具体的运费计算过程。既然不具备计算运费的能力，就不应该将运费计算活动放到财务上下文中，而是考虑放到运输上下文（或者建立一个单独的运费上下文）。由于财务需要运费计算的结果，说明财务上下文需要与运输上下文协作。结合前面对限界上下文的理解，生成运输委托往来账的业务场景就可体现为两个限界上下文业务能力的协作：   
  
**领域模型拥有的领域知识表达了履行活动的角色的业务能力，从而决定了该活动的上下文边界，这就是限界上下文的根本意义。**上下文（Context）体现了业务能力，限界（Bounded）则是保护和隔离上下文的边界，形成了领域模型的知识语境，维护了领域概念的一致性与完整性。

### **限界上下文的特征**

根据限界上下文的定义，我们可以明确它的业务特征与设计特征。在识别限界上下文时，必须满足如下业务特征：

* 它是领域模型的知识语境
* 它体现了业务能力的纵向切分

在设计限界上下文时，必须满足如下设计特征：

* 它是自治的架构单元

#### **领域模型的知识语境**

让我们来读一个句子：

wǒ yǒu kuài dì

到底是什么意思？   

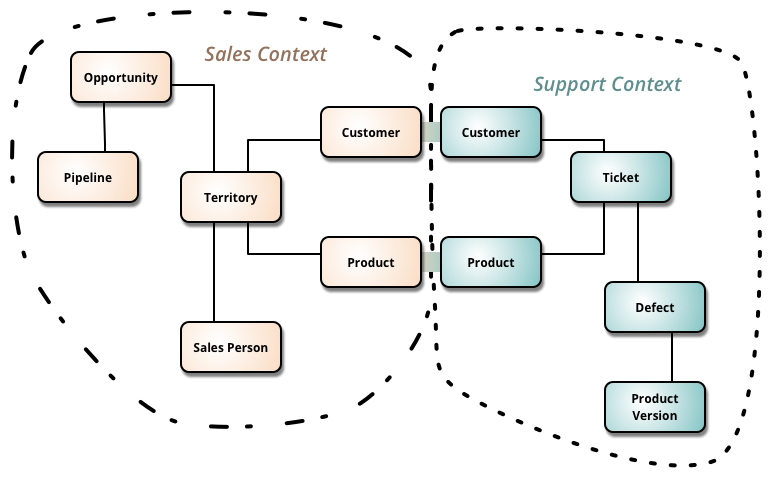

我们能确定到底是哪个意思呢？确定不了，或者说即使确定了也可能引起误解！我们必须结合说话人的语气与语境理解。例如：

* wǒ yǒu kuài dì，zǔ shàng liú xià lái de ——> 我有块地，祖上留下来的。
* wǒ yǒu kuài dì，shùn fēng de ——> 我有快递，顺丰的。

在日常对话中，说话的语气与语境就是帮助我们理解对话含义的**上下文（Context）**。当我们在理解系统的领域需求时，同样需要借助这样的上下文，形成能够达成共识的知识语境【注：实际上英文单词Context本身就可以翻译为“语境”。】。限界上下文（Bounded Context）就是用一个清晰可见的**边界（Bounded）**将领域语言所在的上下文勾勒出来，并在边界内维持领域模型的一致性与完整性。

限界上下文形成的这种知识语境就好似对领域模型指定了“修饰定语”，翻译为代码，就是领域类所属的命名空间。例如，当我们谈论“合同”时，它的语义是模棱两可的，在引入“员工招聘”上下文后，“合同”这一领域概念才变得明朗了，它隐含地表示了“员工招聘的合同”这一概念，代码体现为recruitingcontext.Contract，在熟悉相关领域知识的人看来，也就是员工与公司签订的“劳务合同”。如此就保证了“合同”概念的清晰表达，不会与统一系统的其他“合同”概念混淆，例如属于营销上下文的“合同”，其本质含义是“销售合同”。

在没有引入限界上下文时，由于缺乏边界保护，建立的领域模型面向的是整个系统乃至整个企业，要保证领域概念的一致性，就需要为那些出现知识冲突的领域概念添加显式的定语修饰，如“合同”概念就需要明确细分，分别命名为“销售合同”、“租赁合同”、“培训合同”以及“劳务合同”。毫无疑问，这些领域概念的命名都是在统一语言的指导下进行的，关键在于，当目标系统的问题空间变得规模庞大时，对应的统一语言也将变得规模庞大。一个目标系统需要多个团队共同协作完成，即使明确了这种显式定语修饰的规则，一个团队在不了解其他团队需要面对的领域知识的情况下，团队成员往往意识不到这种概念的冲突，就会选择适合自己团队的命名，不会刻意地保持与相似概念的不同。如果没有限界上下文的界定，就可能悄无声息地出现了领域概念的冲突而不自知。所以Eric Evans就提到：“在整个企业系统中保持这种水平的统一是一件得不偿失的事情。在系统的各个不同部门中开发多个模型是很有必要的，但我们必须慎重地选择系统的哪些部分可以分开，以及它们之间是什么关系。……大型系统领域模型的完全统一既不可行，也不划算。[13]”

这种“得不偿失”还体现在界定领域知识的难度上。要知道，领域概念的一致性与完整性并不仅仅体现在领域模型的命名上，它蕴含的业务规则也必须保证一致而完整。许多时候，在目标系统表达相同领域概念的模型对象，在不同上下文中，需要关注的领域知识也可能并不相同。Martin Fowler在解释限界上下文的文章[47]中，就给出了如下所示的领域模型图：   
  
图来自[47]

销售人员和售后人员面对的客户（Customer）是同一个领域概念，机缘巧合下，甚至可能是同一个人，扮演的也是同一个角色。面对的产品（Product）也如此。然而，销售人员与售后人员因为工作内容和工作性质的不同，需要了解客户和产品的领域知识自然存在非常大的差异：销售人员为了精准营销，需要掌握客户的信息越详细越好，包括客户的职业、收入、消费习惯等，而售后人员为了提供售后服务，只需掌握客户的联系方式与联系地址就足矣。如果没有限界上下文引入的知识语境，要不就生搬硬造出一个个细小的具有修饰语的领域概念，要不就创造出一个合并了各种属性的庞大“上帝”类。

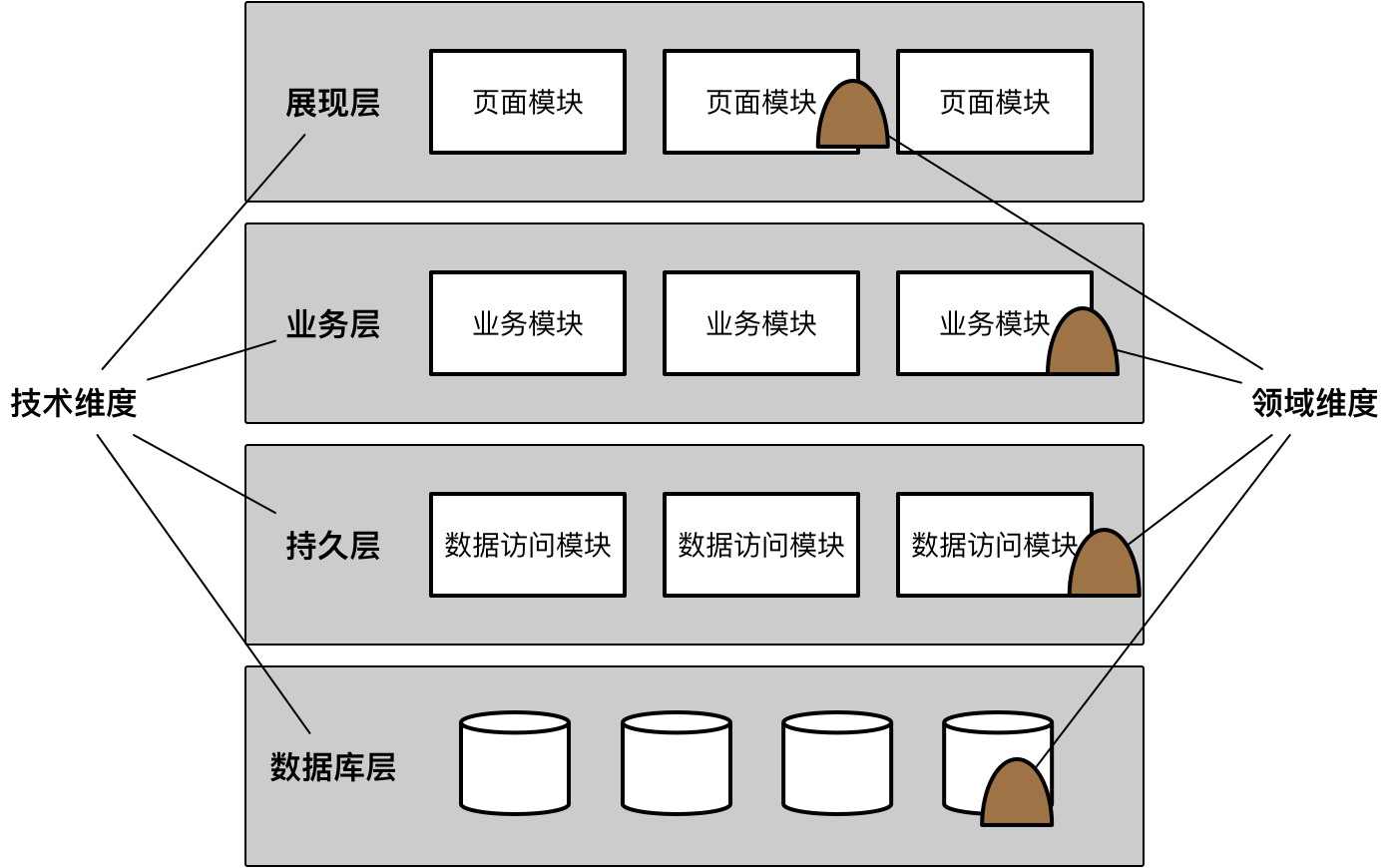
在问题空间，我们需要统一语言来形成团队对领域知识的共识。与其他设计元模式不同，统一语言是贯穿领域驱动设计统一过程始终的，它在架构映射阶段与领域建模阶段起到的作用就是维护领域模型的一致性。Eric Evans将模型的一致性视为模型的最基本要求：“模型最基本的要求是它应该保持内部一致，术语总具有相同的意义，并且不包含互相矛盾的规则：虽然我们很少明确地考虑这些要求。模型的内部一致性又叫做统一，在这种情况下，每个术语都不会有模棱两可的意义，也不会有规则冲突。除非模型在逻辑上是一致的，否则它就没有意义。[13]”显然，限界上下文的边界就是领域模型的边界，它的目的就在于维护领域模型的一致性，这一目的与统一语言的作用重合，因此可以认为：统一语言在解空间的作用域针对每个限界上下文。

#### **业务能力的纵向切分**

要理解所谓“业务能力的纵向切分”，就要明确一个问题，为何领域驱动设计不使用模块（module）、服务（service）、库（libarary）或组件（component）这些耳熟能详的概念来表现业务能力？

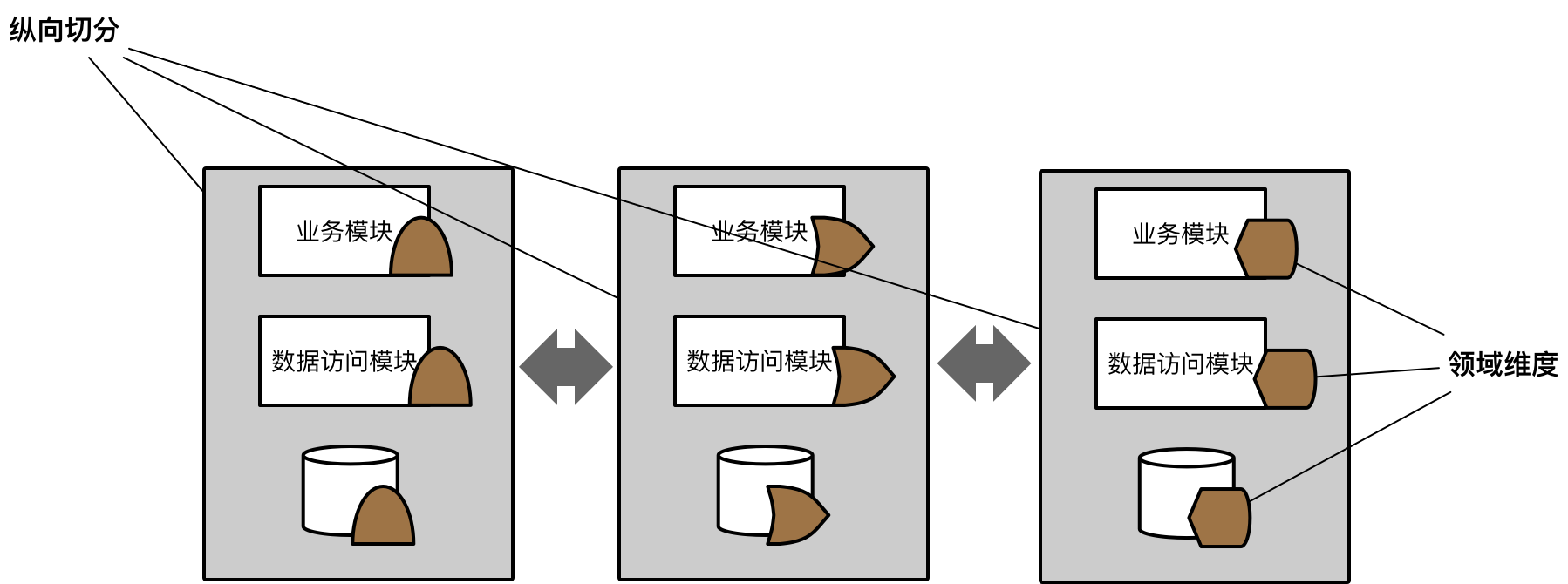
要回答这一疑问，需要先弄清这些概念的真正含义，事实上，虽则我们频繁使用这些概念，对其定义还是懵懂不明的。Neal Ford等人在《演进式架构》中言简意赅地给出了它们的定义和区别[46，P39]：“模块意味着逻辑分组，而组件意味着物理划分。”组件有两种物理划分形式，分别为库和服务。“库……往往和调用代码在相同的内存地址内运行，通过编程语言的函数调用机制进行通信。……服务倾向于在自己的地址空间中运行，通过低级网络协议（比如TCP/IP）、更高级的网络协议（比如SOAP），或REST进行通信。”

显然，模块属于逻辑架构视图的软件元素，组件（库和服务）属于物理架构视图的软件元素。模块与组件的区别体现了观察视图的不同，它们之间并不存在必然的映射关系。模块虽然是逻辑分组的结果，却不仅仅针对业务逻辑，某些基础设施的功能如文件操作、文件传输、网络通信等，也可以视为逻辑划分，并在逻辑架构视图中被定义为模块，然后，在物理架构视图中根据具体的质量属性要求实现为库或者服务。为了与限界上下文做对比，不妨将体现了业务逻辑划分的模块称之为“业务模块”。

那么，业务模块是否就是限界上下文呢？非也！因为模块作为体现职责内聚性的设计概念，它缺乏一套完整的架构体系支撑，边界是模糊不清的。业务模块是从业务角度针对纯粹的业务逻辑的归类与组织，仅此而已，它缺乏自顶向下端对端的独立架构，使得自身无法支撑业务能力的实现，如下图所示：   


【注：该图参考了《演进式架构》图4-11 P56】

图中所示的架构首先从技术维度进行关注点切分，形成了一个分层架构；然后，业务模块又在此基础上针对业务层进行了领域维度[46, P56]的再度切分，封装了纯粹的领域逻辑。业务模块不具备独立的业务能力，只有把分散在各层中与对应领域维度有关的业务模块、数据访问模块以及数据库层的数据库或数据表整合起来，才能为展现层的页面模块提供完整的业务能力支撑。这正是业务模块的致命缺陷。分散在分层架构各个层次的领域维度切片也说明了**模块的划分没有按照同一个业务变化方向进行**，一旦该领域维度的业务逻辑发生变化，就需要更改整个系统的每一层。这正是我所谓的“模块缺乏一套完整架构体系支撑”的原因所在。

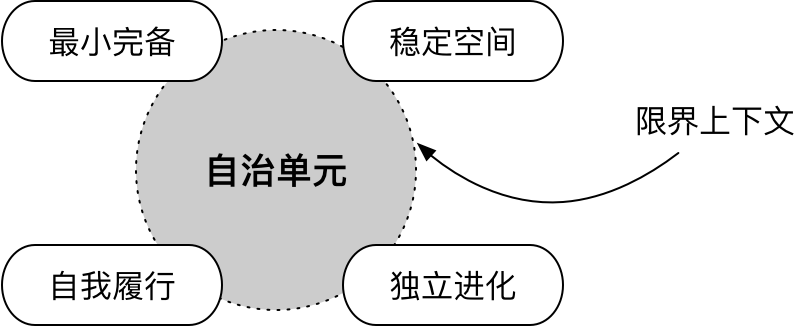
限界上下文与之相反。Eric Evans指出：“根据团队的组织、软件系统的各个部分的用法以及物理表现（代码和数据库模式等）来设置模型的边界。在这些边界中严格保持模型的一致性，而不要受到边界之外问题的干扰和混淆。[13]”这意味着限界上下文边界的控制力不只限于业务，还包括实现业务能力的技术内容，包括代码与数据库模式，它是对目标系统架构的纵向切分，切分的依据则是从业务进行考虑的领域维度。为了完整地满足业务能力的要求，在根据领域维度进行切分时，还需要考虑支撑业务能力的基础设施实现，如与给业务相关的数据访问逻辑，以及将领域知识进行持久化的数据库模型，形成纵向的逻辑边界，即限界上下文的边界。然后，在边界保护下从技术维度考虑内部的横向关注点切分，将业务逻辑与技术实现分离，形成限界上下文内部的独立架构。当然，考虑到前后端分离的架构以及用户体验的特殊性，通常限界上下文并不包含对展现层的纵向切分。形成的架构如下所示：   
  
对外，以限界上下文为整体，就能避免与之相关的领域维度切片被分散到各个地方，如此就能更好地响应业务变化，当与该领域维度相关的业务发生变化时，能将变化影响的内容收敛在一处；对内，将限界上下文视为一个小型系统，按照关注点分离原则进行横向切分，把蕴含了业务逻辑的领域层单独剥离出来，形成了清晰的结构，隔离了业务复杂度与技术复杂度。

与业务模块相比，限界上下文拥有自己的内部架构【注】，提供了对业务能力而非业务逻辑的纵向切分，而业务模块太脆弱了，根本支撑不起限界上下文的重量。在形成这样的业务能力纵向切分之后，整个目标系统的业务架构、应用架构与数据架构才能统一起来，而限界上下文的内部架构又确定了业务与技术的边界，实现了对技术架构的解耦。因此，限界上下文是领域驱动设计战略层面最重要且又最基本的架构设计单元。

【注：我会在后面章节结合整洁架构思想，引入分层架构与六边形架构发展出属于限界上下文的内部架构模式——菱形对称架构。】

#### **自治的架构单元**

限界上下文作为一个基本的架构设计单元，既要体现领域模型的知识语境，又要能独立提供业务能力，这就要求它具有自治性，形成自治的架构单元。

什么是自治的架构单元？一个设计良好的限界上下文只有满足四个特征：最小完备、自我履行、稳定空间和独立进化，方可称为自治的架构单元：   


**最小完备**是实现限界上下文自治的基本条件。所谓“完备”，是指限界上下文在履行属于自己的业务能力时，拥有的领域知识是完整的，无需针对自己的信息去求助别的限界上下文，这就避免了不必要的领域模型依赖。简言之，限界上下文的完备性，就是领域模型的完备性，当然就是领域知识的完备性。当然，仅追求限界上下文的完备性是不够的，要知道，一个大而全的领域模型必然是完备的。为了避免领域模型被盲目扩大，就必须通过“最小”加以限制，避免将不必要的职责错误地添加到当前限界上下文。最小完备体现了限界上下文对领域知识的封装。

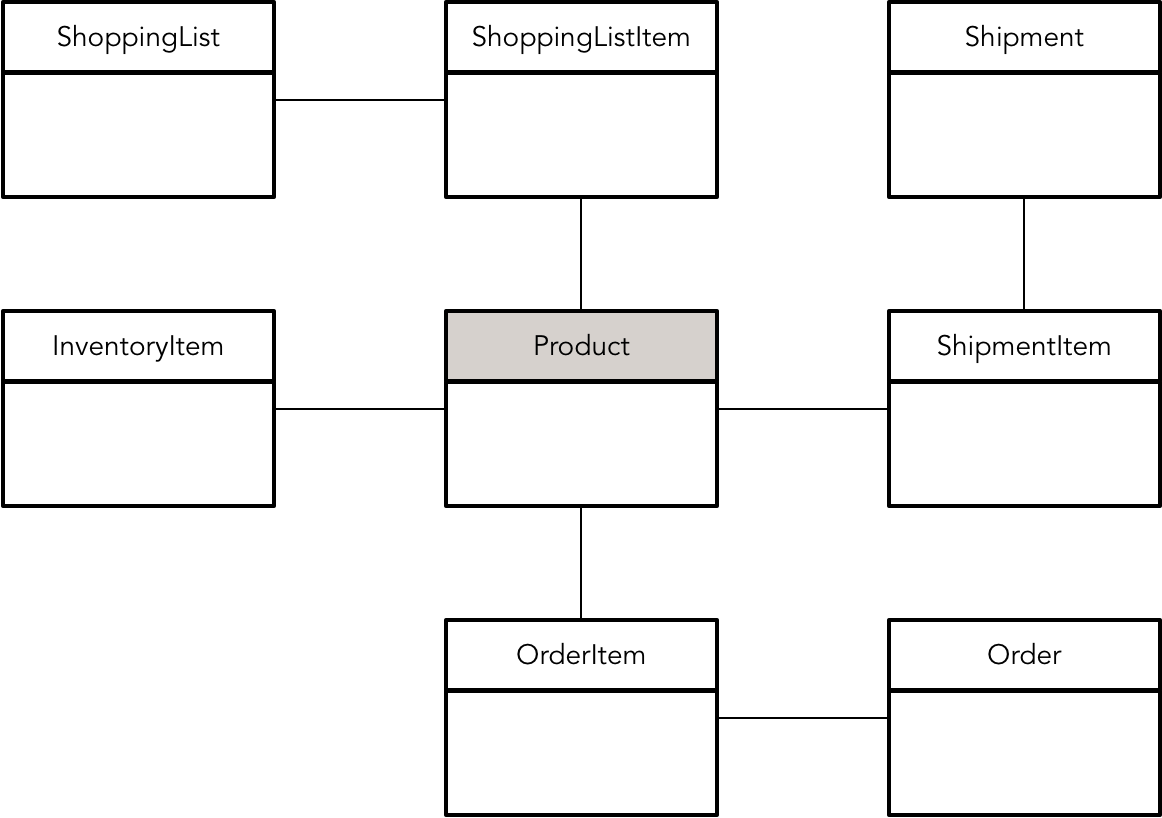
**自我履行**意味着由限界上下文自己决定要做什么，就好似拥有了智能，能够根据自我拥有的知识对外部请求做出符合自身利益的明智判断。分配业务功能时，设计者就应该化身为限界上下文，模拟它进行思考：“我拥有足够的领域知识来履行这一职责吗？”如果做不到，在领域知识的分配是合理的情况下，就说明这一职责不该由当前限界上下文承担，那就要果断放弃。由于“履行”一词表达了职责意识，而非单纯地对数据或信息的拥有与传递，这就隐含地说明，当一个限界上下文具有自我履行的意识时，它就不会轻易突破边界，企图重用别人的领域模型，甚至绕过限界上下文直接访问不属于它的数据，而会优先以行为协作的方式履行自己的职责。自我履行体现了限界上下文对业务能力的承担。

**稳定空间**要求限界上下文必须防止和降低外部变化带来的影响。在满足了“最小完备”与“自我履行”特征的前提下，一个限界上下文已经拥有了必备的领域知识，即使这些领域知识代表的逻辑发生了变化，也是可控的。只有发生在限界上下文外界的变化，才鞭长莫及，力不从心。因此，要保证内部空间的稳定性，就是要解除或降低对外部软件元素的依赖，包括必须访问的环境资源如数据库、文件、消息队列等，也包括当前限界上下文之外的其他限界上下文或伴生系统。解决之道就是通过抽象降低耦合，只要保证访问接口的稳定性，外界的变化就不会产生影响。

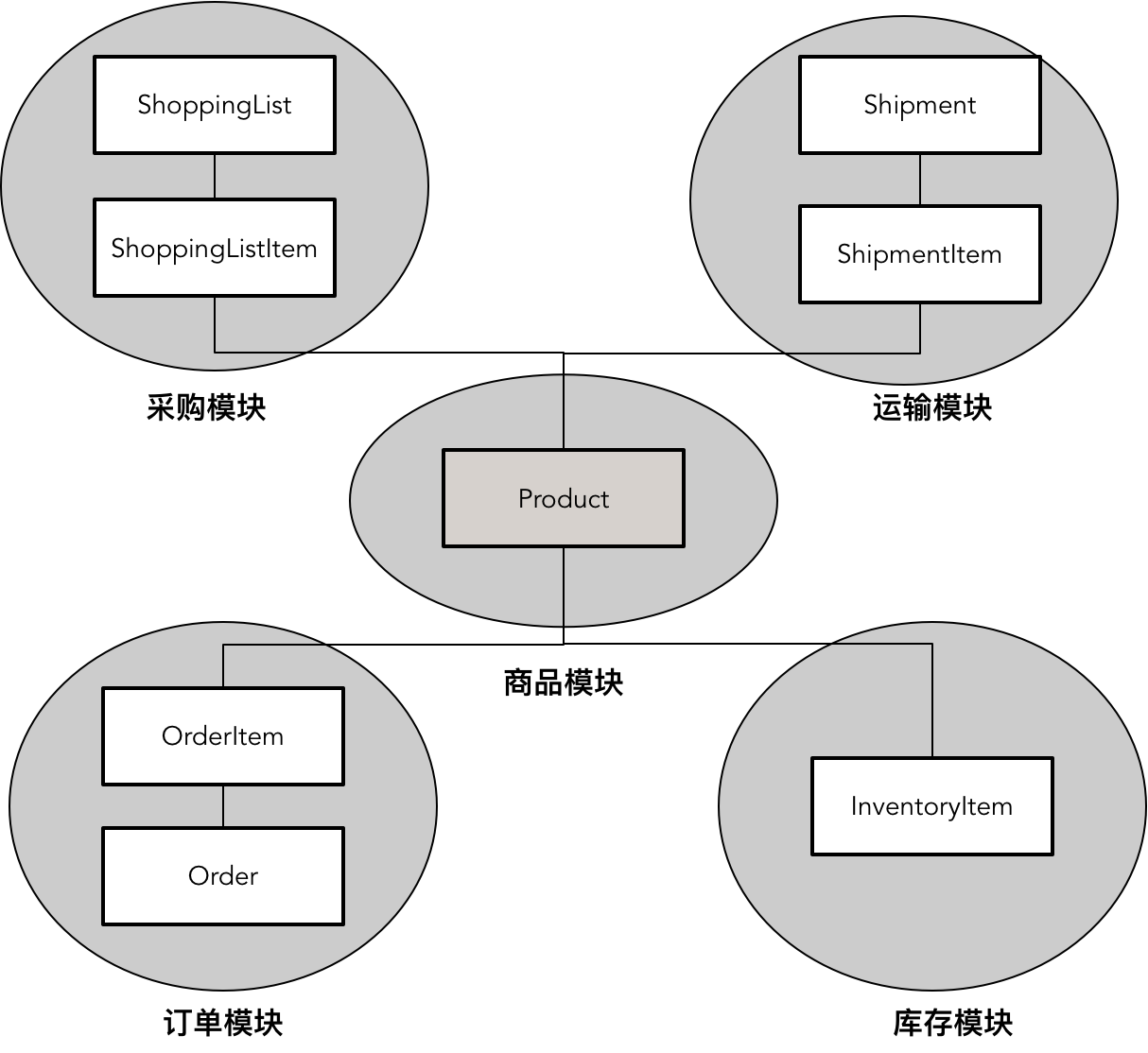
**稳定空间**指的是减少外界变化对限界上下文内部的影响，**独立进化**则相反，指的是减少当限界上下文内部变化对外界产生的影响。这体现了边界的控制力，需要维持一个具有守护能力的“边疆”，对外公开稳定的接口，而将内部领域模型的变化封装在限界上下文的内部。显然，满足独立进化的核心思想是封装。抽象与封装都要求限界上下文划分合理而清晰的内外层次，在其边界内部形成独立的架构空间，即通过菱形对称架构【参考章节】满足限界上下文响应变化的能力。

限界上下文的自治四要素相辅相成。最小完备是基础，只有赋予了限界上下文足够的知识，才能保证它的自我履行。稳定空间对内，独立进化对外，二者都是对变化的有效应对，而它们又是通过最小完备和自我履行来保证限界上下文受到变化的影响最小。遵循自治特性的限界上下文构成了整个系统的架构单元，成为响应业务变化与技术变化的关键支撑点。

#### **案例：供应链的商品模型**

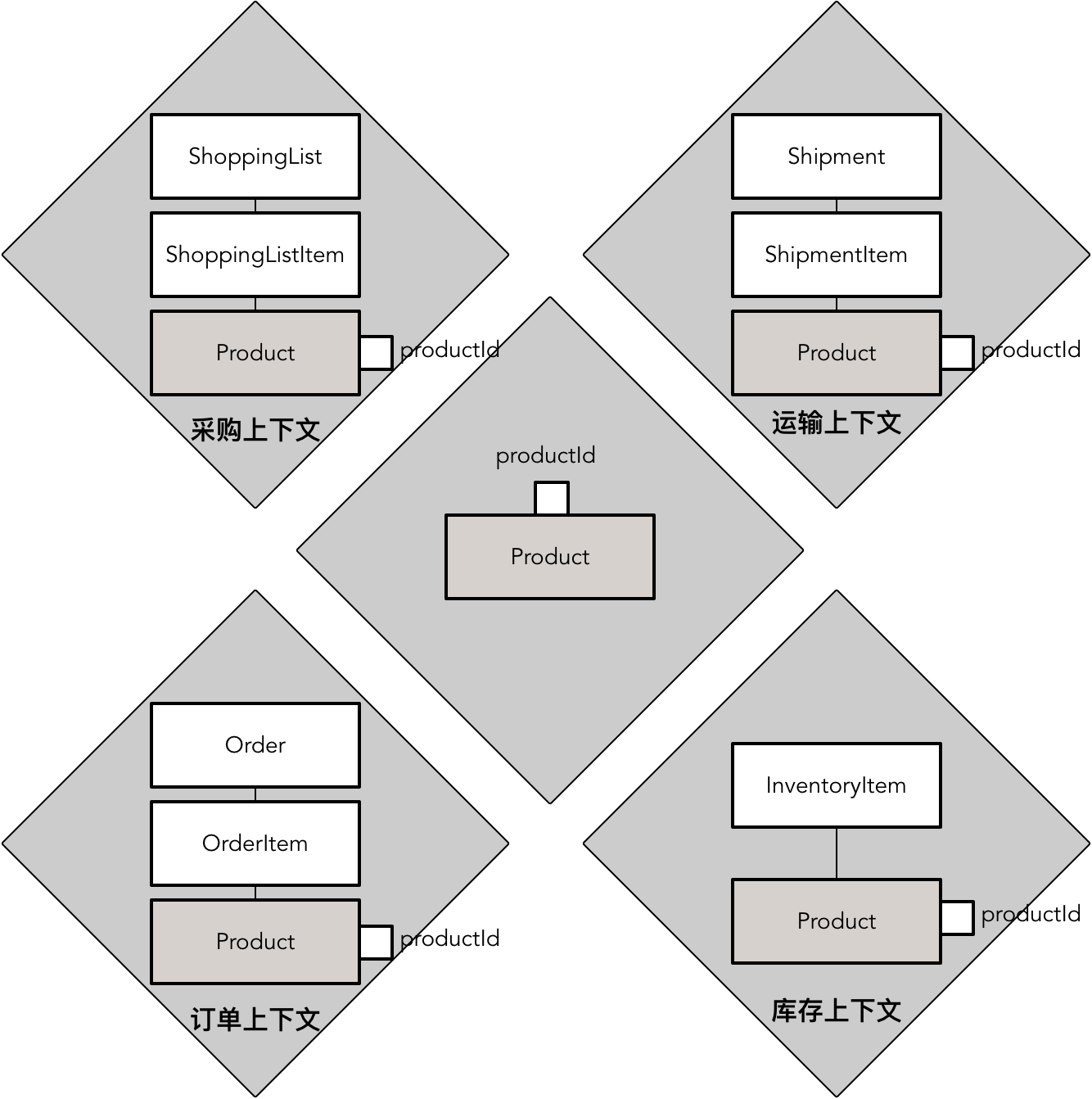
让我们通过供应链的一个案例，深刻体会自治的限界上下文与模块的不同之处。供应链系统的一个核心资源是商品（Product），无论是采购、订单、运输还是仓储，都需要用到商品的信息，因而需要在供应链系统的领域模型中定义“商品（Product）”模型。在未引入限界上下文边界之前，建立的领域模型如下所示：   
  
这里列出的领域模型仅是供应链系统的冰山一角，却已经出现了模型膨胀的苗头。再来看模型中的商品概念，Product类确乎唯一表达了商品这一领域知识在现实世界中的商品概念；但在采购、报价、订单、运输以及仓储等不同视角中，商品却呈现了不同的面貌。例如，采购员在采购商品时，并不需要了解与运输相关的商品知识，如商品的高度、宽度与深度；运输商品时，配送人员并不关心商品的进价、最小起订量与供货周期。在没有边界的领域模型中，Product类若要完整呈现这些差异性，就必须包含与之相关的领域知识，使得Product领域模型变得越发的臃肿：   
   
Product类包含了整个供应链系统范围的商品知识，由于没有边界限定这些知识，就可能因为模棱两可的定义引起领域概念的冲突。例如，管理仓储时，需要通过获得商品的宽度、高度与深度计算商品的体积，由此确定仓库的存放空间。于是，仓储团队在建模时为Product定义了width、height与depth三个属性，分别代表商品的宽度、高度与深度。运输商品时，运输团队也需要知道商品的高度、宽度与深度，目的是为了计算每个包裹的占用空间。同样都是宽度、高度与深度，在仓储和运输这两个场景中，代表了不同的含义。前者是真实的商品高度、宽度与深度，用以计算存放空间，后者则抽象为规则长方体的高度、宽度与深度，用于计算运输空间，如果将它们混为一谈，就会引起计算错误。要在同一模型下避免这种冲突，就只能为属性添加定语修饰，如将宽度分别定义为product width和shipping width。

如此庞大的Product类必然违背了“单一职责原则[48]"，它包含了多个引起它变化的原因。当采购功能对商品的需求发生变化时，需要修改它；当运输功能对商品的需求发生变化时，也需要修改它……这就使得它成为了一个极不稳定的“热点（Hot Spot）”。由于这样的类需要为不同的业务场景公开不同的信息，封装遭到了破坏，依赖变得更多，它就像一块巨大的磁铁，产生了强大的吸力，将与之相关的模块或类吸附其上，造成了业务逻辑的强耦合。

为供应链系统引入业务模块是否能解决这些问题呢？业务模块是对业务逻辑的划分，强调了对领域知识的归类，上述领域模型就可以分别放到采购模块、运输模块、仓储模块、订单模块与商品模块中。由于商品模块的Product类与各模块的领域模型存在关联关系，就会形成如下的模块结构：   
  
这一模块结构看起来满足了封装领域模型的要求，但它并没有做到模型的统一，Product类被定义在商品模块中，并没有清晰表达它在各个业务场景中的意义与差异。由于模块内部缺乏一个层次清晰的架构来支撑，使得模块边界的控制能力被削弱了。如图中所示，各个模块的领域模型直接依赖了商品模块的Product类，根据共同重用原则（Common Reuse Principle，CRP）[48]，因为重用了Product类的定义，调用Product的业务模块都需要依赖商品业务模型。

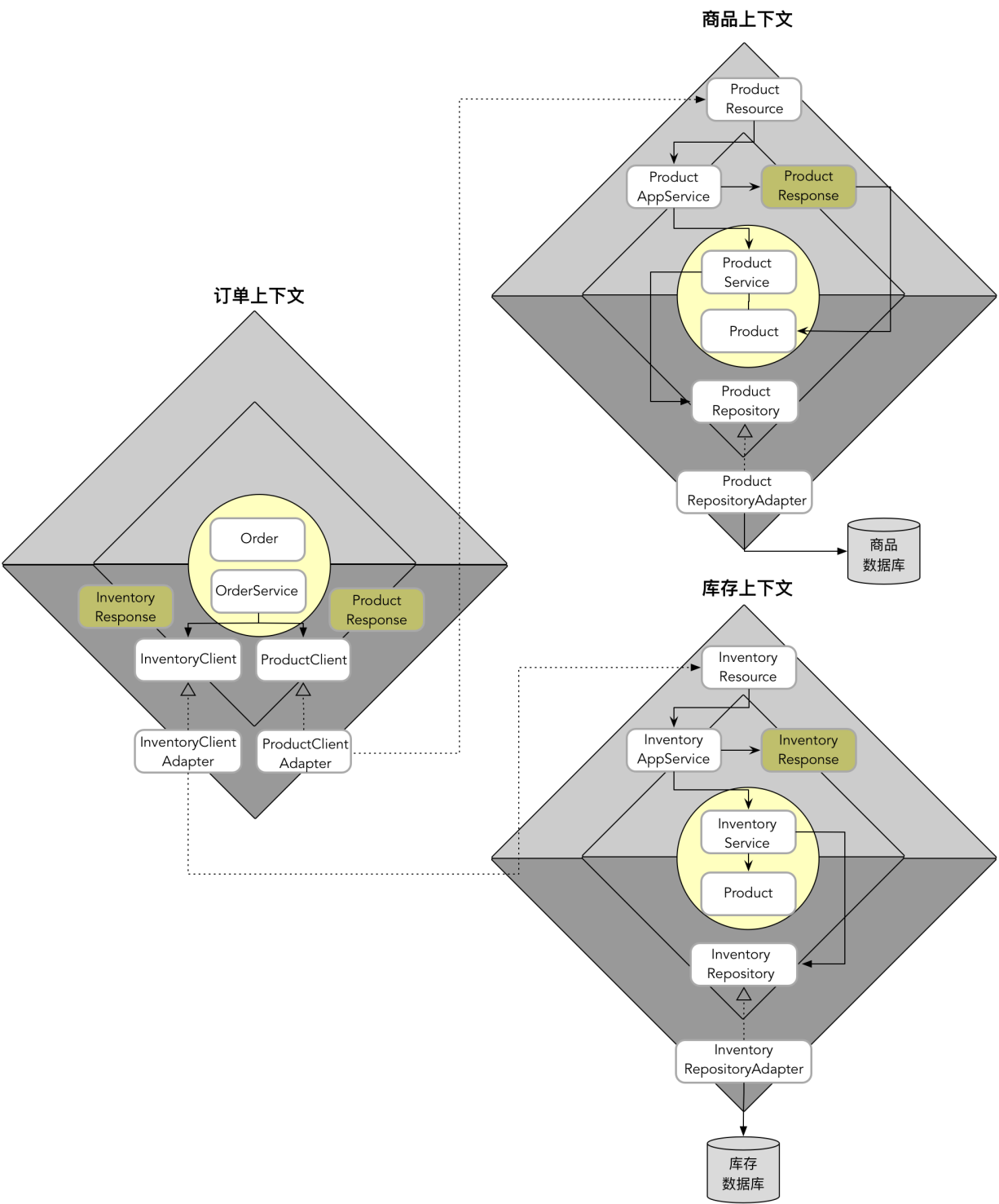
当代表不同业务的业务模块被紧紧耦合在一起时，随着需求不断的变化，业务模块的边界就会变得越来越模糊。模块之间存在的若有若无的依赖，原本内聚的磁力缺乏了边界的有效隔离，就会慢慢吸附上诸多灰尘颗粒，渐渐填补了模块之间的空隙，变成了一个“大泥球”。一个直观的现象就是庞大的Product类在各个模块之间传来传去，而在Product类的实现中，随处可见采购、订单、运输与仓储等业务逻辑的踪影，形成了“你中有我、我中有你”的狎昵关系[9]。目标系统的架构因为缺乏空隙变得没有弹性，无法响应业务变化，架构的演化也会变得举步蹒跚。

显然，业务模块并没有针对一个完整业务进行领域维度的纵向切分，自治能力的缺失使得它的边界控制能力太弱，无法满足大型项目响应业务变化的架构需求。

限界上下文首先需要满足“最小完备”的自治特征，这就要求从知识完备的角度根据不同的知识语境划分专属于自己的领域模型。观察Product类蕴含的领域知识，对于不同的业务场景，具有共同特征的领域知识构成了高内聚的业务单元，各个业务单元之间的联系则是松散的：   
  
例如，如果我们不将商品的运输高度、宽度、是否装箱等领域知识赋给运输上下文，运输上下文就缺乏“完备性”，可要是将商品进价、最小起订量与供货周期也一股脑儿提供给它，就破坏了“最小性”对知识完备的约束了。因此，“最小完备”要求限界上下文对领域模型各取所需，拥有自己专属的领域模型，根据知识语境定义独立的Product类，形成限界上下文内部对商品的唯一表示：   
  
虽然不同的限界上下文都定义了Product类，但由于有了限界上下文作为边界，使得我们在理解领域模型时，是基于当前所在的上下文作为知识语境的，如ShoppingListItem关联的Product类表达了与采购相关的商品领域知识。倘若要确认多个限界上下文的商品是否属于同一件商品，就由商品上下文统一维护商品的唯一身份标识，限界上下文之间对Product类的依赖就被解除了，改为对productId的依赖，并由它维持商品的唯一性。

基于“自我履行”的要求，一个限界上下文应该根据自己拥有的信息判断职责该由谁来履行。例如，采购上下文希望获得商品的基本信息，这是它不具备的信息，只有交给拥有商品基本信息的商品上下文来履行。遵循“自我履行”的原则，一个限界上下文不会轻易突破自己的职责边界，也不会越俎代庖履行本该由别的限界上下文承担的职责。例如，订单上下文希望检查订单中商品的库存量是否满足购买需求，由于它拥有的信息不足以履行该职责，它就不能绕开本该履行检查库存量职责的库存上下文，直接访问存储了商品库存量的数据表，否则它就破坏了“自我履行”原则。

为了确保限界上下文“独立进化”的能力，还需要定义开放主机服务（OHS，参考章节）隔离位于限界上下文内部的领域模型。菱形对称架构【参考章节】的网关层保护了领域模型，不允许领域模型“穿透”限界上下文的边界。如果一个限界上下文想要获得另一个限界上下文的领域知识，需调用北向网关的服务，返回符合请求的消息契约模型。这一设计增加了消息契约模型与领域模型的转换成本，却避免了因重用领域模型带来的依赖，在一定程度上保留了独立进化的能力。例如，商品上下文自我履行的“获取商品基本信息”职责，库存上下文自我履行的“检查库存量”职责，都需要公开为服务的形式，并分别返回ProductResponse与InventoryResponse响应对象。

对内而言，如果限界上下文想要维持自己的“稳定空间”，就需要通过防腐层（ACL，参考章节）来抵御外界变化带来的影响。例如，订单上下文需要调用库存上下文公开的“检查库存量”服务，为了抵御该服务可能的变化，就需要通过菱形对称架构的南向网关建立抽象的客户端端口，将变化封装在适配器的实现中。库存上下文为了检查库存量，需要访问库存数据库。由于数据库属于外部的环境资源，为了不让它的变化影响领域模型，亦需要通过南向网关定义抽象的资源库端口，商品上下文获取商品信息同样如此：   
  
对比业务模块，限界上下文更加淋漓尽致地展现了“自治”的特征。它通过其边界维持了各自领域模型的一致性，避免出现一个庞大而臃肿的领域模型，并利用内部的菱形对称架构保证了限界上下文之间的松散耦合，支撑了它对业务能力的实现，建立了保证领域模型不受污染的边界屏障。

### **识别限界上下文**

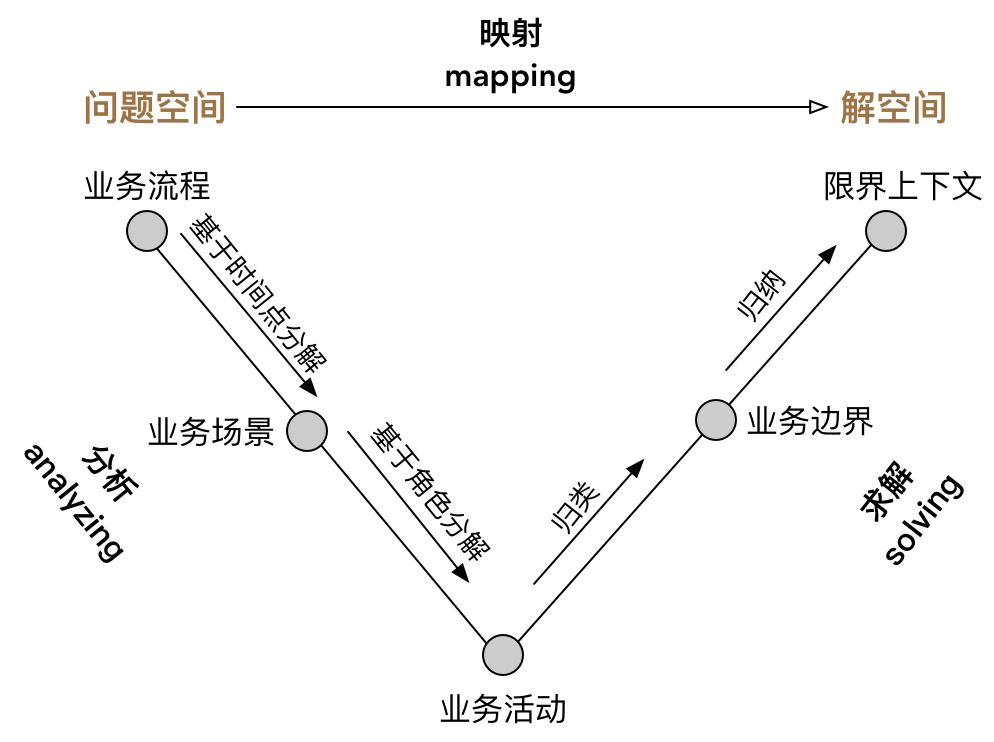
不少领域驱动设计专家都非常重视限界上下文，越来越多的实践者也看到了它的重要性。Mike Mogosanu认为：“限界上下文是领域驱动设计中最难解释的原则，但或许也是最重要的原则。可以说，没有限界上下文，就不能做领域驱动设计。在了解聚合根（Aggregate Root）、聚合（Aggregate）、实体（Entity）等概念之前，需要先了解限界上下文。”经过前面对限界上下文的分析，我认为足以了解限界上下文；然而，限界上下文不会自我呈现，我们需要像猎人一般去探寻它的痕迹，捕捉到它的身影。遗憾的是，就连领域驱动设计之父Eric Evans对如何识别限界上下文，亦是语焉不详。

识别限界上下文的质量直接制约我们的设计，而高明的架构师雅擅于此，一个个限界上下文跃然于纸上，当我们惊叹于概念的准确性和边界的合理性时，问其如何获得，他们却笑称这是妙手偶得。软件设计亦是一门艺术，似乎需要凭借一种妙至毫巅、心领神会的神秘力量，刹那之间迸发灵感，于是，设计的奇思妙想就产生了。实情当然不是这样，若要说真的存在秘而不宣的神奇力量，那就是架构师们千锤百炼造就的“经验”。

Andy Hunt分析了德雷福斯模型（Dreyfus Model）的5个成长阶段：新手、高级新手、胜任者、精通者和专家。对于最高阶段的“专家”，Andy Hunt得出一个有趣的结论：“专家根据直觉工作（Experts work from intuition），而不需要理由。[54]”这一结论充满了神秘主义，反复回味，却又理所当然，专家的“直觉”实际就是通过不断的项目实践磨炼出来的，正所谓厚积薄发。当然，经验的累积过程需要方法，否则所谓数年经验不过是相同的经验重复多次，没有价值。Andy Hunt认为：需要给新手提供**某种形式的规则**去参照，之后，高级新手会逐渐形成一些**总体原则**，然后通过系统思考和自我纠正，建立或者遵循**一套体系方法**，就能从高级新手慢慢成长为胜任者、精通者乃至于专家。因此，从新手到专家是一个量变引起质变的过程，在没有能够养成直觉的经验之前，我们需要一套方法。

Edward Crawley谈到系统思考者在对复杂系统进行分析或综合时可以运用的技巧：“自顶向下（top-down）和自底向上（bottom-up）是思考系统时的两种方向。……我们先从系统的目标开始，然后思考概念及高层架构。在制定架构时，我们会反复地对架构进行细化，并在我们所关注的范围内，把架构中的实体分解到最小。……自底向上……就是先思考工件、能力或服务等最底层的实体，然后沿着这些实体向上构建，以预测系统的涌现物。除了这两种方式外，还有一种方法是同时从顶部和底部向中间行进，这叫做由外向内（outer-in）的思考方式。”[21，P44]在识别限界上下文时，可以借鉴这一思路。

#### **业务维度**

识别限界上下文的过程，是将问题空间的业务需求映射到解空间限界上下文的过程。全局分析阶段的**业务分析工作流**采取自顶向下的分析方法，问题空间中的业务流程根据时间维度切分为各个相对独立的业务场景，再根据角色维度将业务场景分解为业务活动；然后，架构映射阶段的**业务级映射工作流**又采取自底向上的求解方法，从业务活动逆行而上，通过逐步的归类与归纳获得体现业务能力的限界上下文。问题空间的分析过程与解空间的求解过程共同组成了**V型的限界上下文映射过程**：   
  
整个映射过程从分解、归类到归纳形成了一套相对固化的映射过程，可以在一定程度上消解我们对经验的依赖。同时作为分析过程终点与求解过程起点的业务活动，成为了V型映射过程的交叉点。它是一个抽象的概念，代表了由角色参与的具有业务价值的领域行为。在分析阶段，对应于全局分析的业务分析，如果采用用例建模，业务活动就是系统用例；如果采用事件风暴，业务活动就是触发事件的决策命令，由于事件与决策命令是一一对应的，可以直接用事件代表业务活动。到了求解阶段，就无所谓用例还是事件，本质上都是按照语义相关性与功能相关性将业务活动归类到同一个业务边界，再通过归纳形成共同的业务主题，命名，即获得了解空间的限界上下文。

**业务知识的归类**

通过业务需求分析获得的业务活动，是表达业务知识的最基本元素，按照相关性对其进行归类，就是按照“高内聚”原则划定业务知识的边界。这种相关性主要体现为：

* 语义相关性
* 功能相关性

角色在业务场景中的业务活动采用动宾短语的简洁形式进行描述，动词代表领域行为，宾语代表角色所要操作的对象，体现了该业务活动的语义特征。如果不同的业务活动操作了相同的对象，即可认为它们存在**语义相关性**。例如，对于文学阅读产品的业务活动而言，诸如查询作品、收藏作品、分享作品、阅读作品都具有“作品”的语义，根据语义的共同特征可将其归为同一类。

一份高质量的业务分析输出成果，可以帮助我们更好地识别限界上下文。质量的优劣除了体现为对业务活动的分解之外，还体现在对业务活动的命名。命名传递的是业务知识，它既要体现执行该业务活动带给角色的业务价值，又要就描述中的概念在团队达成一致，准确地表达统一语言。这就要求我们在描述业务活动时须字斟句酌，把握每个动词和名词的精确性。命名的过程其实也是提炼统一语言的过程，提炼出来的统一语言会作为对限界上下文进行领域建模的主要参考。

例如，在“管理项目”的业务场景中，项目经理需要查看任务的完成情况，该业务活动最初被描述为“查看任务完成情况”，这一描述并没有体现该活动为项目经理提供的业务价值，他之所以要查看任务的完成情况，是需要跟踪任务的进度。在对软件项目进行管理时，“任务”这一词语又存在歧义，它到底代表了用户故事，还是用户故事分解获得的子任务，甚至是与用户故事平级的表达技术型工作的任务？如果任务与用户故事平级，难道还有一个业务活动被描述为“查看用户故事完成情况”吗？假设项目管理的需求体系中还引入了史诗故事（Epic），是否还需要引入“查看史诗故事完成情况”业务活动？显然，对于项目经理而言，他要了解的进度并不会区分任务、用户故事还是史诗故事，这就需要定义这些概念的统一语言，例如统一命名为“问题（Issue）”，该业务活动就应该描述为“跟踪问题进度（Tracking Issue Progress）”。“问题”统一了语义，避免陷入任务、用户故事、史诗故事诸概念的混乱之中。

**功能相关性**体现为业务活动领域行为的相关性，这种相关性并非设计意义上领域行为之间的功能依赖，为了避免判断错误，需要正确识别业务活动。

一个正确识别的业务活动应由角色驱动，系统作为整体对角色期望的服务价值进行响应。按照这一原则识别和分解业务活动，就不会受困于业务活动粒度粗细的问题，更不会将业务活动的执行步骤视为一个独立的业务活动，排除了那些看似存在功能相关性的错误的业务活动。例如，“提交订单”与“验证订单库存”看似具有功能相关性，但实际上，“验证订单库存”仅仅是“提交订单”业务活动的一个执行步骤，它自身并不提供服务价值，也没有任何角色参与到这个步骤。

如何判断业务活动是否正确识别呢？潘加宇认为：“建模人员只能根据目标涉众心中对系统的期望来确定系统应该提供什么样的服务。[27，P162]”如上所述的“验证订单库存”就没有满足顾客对电商系统的期望，顾客使用该电商系统，显然不是为了验证订单库存来的，也就是没有提供服务价值。

在识别业务场景的参与角色时，也要谨慎地避免引入“系统”这个角色。因为广而推之，哪个业务功能不是由系统执行的呢？它代表的广泛意义很容易被扩大化，形成一个万能的参与角色，而产生大量不恰当的业务活动。例如，作者在文学阅读平台中发布作品时，系统会对待发布内容做违规检查。因为引入了“系统”这个角色，就会自然而然地认为系统的业务活动为“执行违规检查”，但事实上它只是发布作品的一个执行步骤。只有当一个业务活动由系统“主动”执行，它才会成为业务场景的参与角色。例如，业务规则要求顾客在提交订单后的15分钟内完成支付，若在规定时间内支付未完成，则取消订单。那么“取消订单”业务活动的参与角色就是两个：顾客与系统。在这个业务场景下，其实更准确的角色命名应该为“定时器（timer）”。

一旦正确识别了业务活动，就不存在相互依赖相互调用的业务活动，因为这种依赖性实际应转换为业务活动的执行步骤。因此，这里所指的功能相关性，表达的含义为**业务活动是否都是为了同一个业务目标**。如果是，就认为这样的业务活动是功能相关的。例如，在人力资源管理系统中，员工执行“请假”业务活动，人力专员执行“提交考勤记录”业务活动，二者虽然并不存在相互依赖，但都是为了考勤这一共同业务目标，即可认为二者是功能相关的。

可以同时结合语义相关性和功能相关性共同对业务活动进行归类。只要业务活动的名称准确地体现了业务知识，通过语义相关性进行归纳更为容易，可以认作是一种表象的分析；分析功能相关性则需要明确它们的业务目标，若无深入理解，较不容易进行归类。

**业务知识的归纳**

无论语义相关性还是功能相关性，都是业务活动类别的判断标准。业务活动进行了归类，自然就划定了业务边界。接下来，就需要对边界内的所有业务活动进行业务知识的归纳。归纳的过程也是抽象的过程，需要概括所有业务活动所述的主题特征，以最准确的名词表达该主题。故而归纳的过程也是对业务边界命名的过程。业务边界的命名遵循单一职责原则，即这个名称只能代表唯一的最能体现其特征的领域概念。倘若对业务活动的归类欠妥当，命名就会变成一件不可能完成的任务。只要不是抽象能力不具备，当发现命名的工作无以为继时，就该反过来思考之前的归类是否合理。显然，对业务边界的命名可算作是对限界上下文识别的一种检验手段。

在文学阅读平台中，我们根据语义相关性和功能相关性将如下业务活动归类为同一个业务边界：

* 建立读者群
* 加入读者群
* 发布群内消息
* 实时聊天
* 发送离线消息
* 一对一私聊
* 发送私信

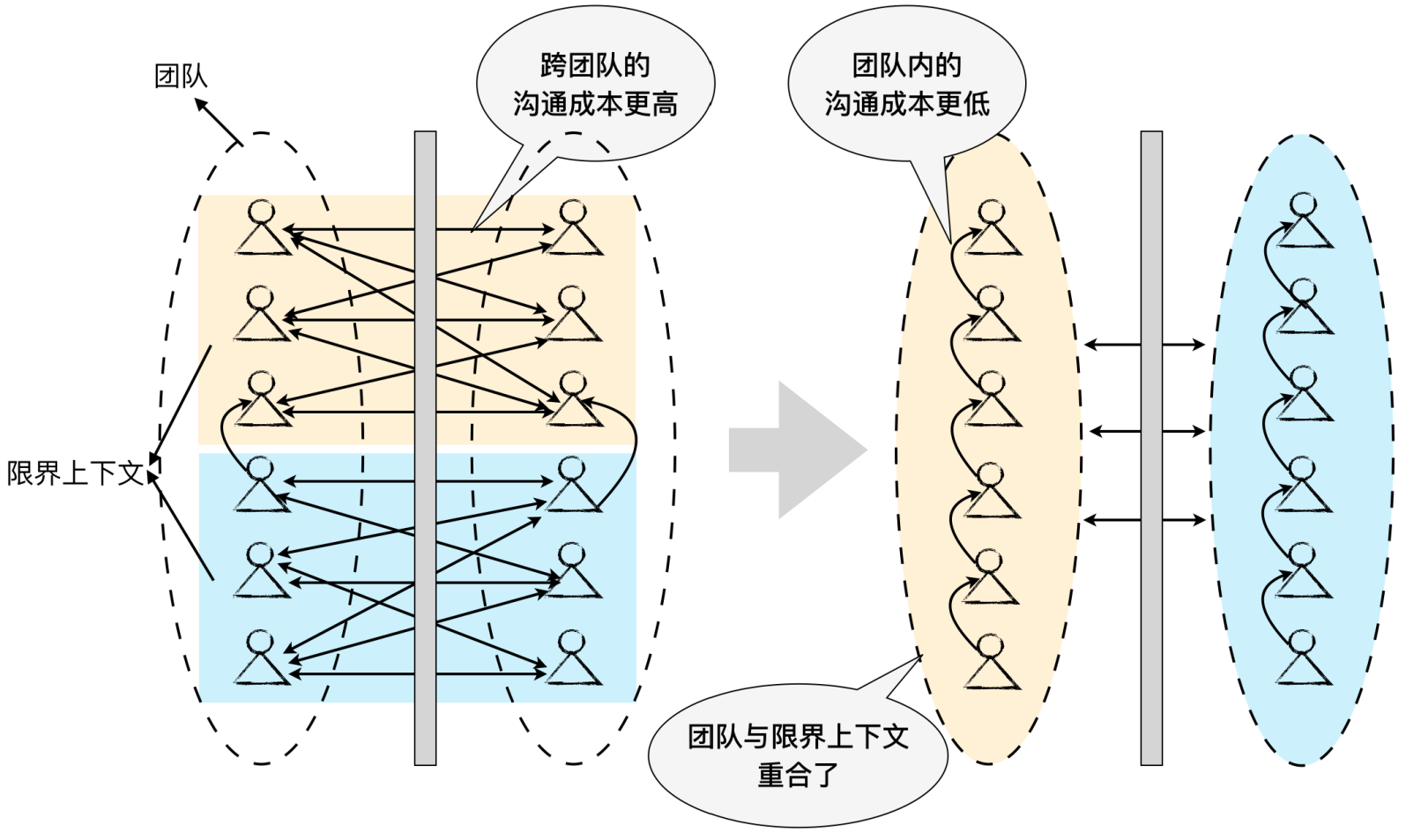
它们都是为了支持作者和读者之间的交流，满足“社交”这一主题特征，故而可以初步获得“社交”限界上下文。如果我们将阅读作品、收藏作品与关注作者、查看作者信息放在一个业务边界内，命名就变得棘手了，这些业务活动实际存在两个交叉的主题，分别为“作品”和“作者”，我们不能强行将其命名为“作品与作者”限界上下文。换言之，当你发现限界上下文的名称包含了“和（and）”与“或（or）”等字眼儿时，其实就是在传递业务边界识别不合理的信号！

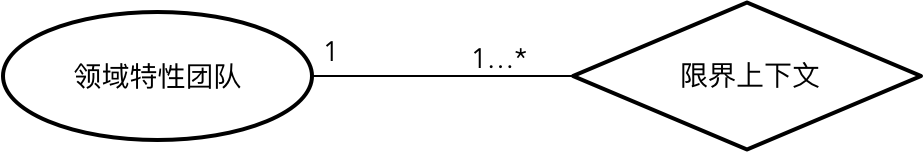
整体而言，识别限界上下文的根本在于业务知识。在理解领域逻辑时，不能出现偏差和不一致，错误的业务理解会影响限界上下文的识别。领域驱动设计强调领域专家与开发团队的紧密合作，识别限界上下文也将是一个充分讨论和分析的过程，尤其不能缺少领域专家的参与。它还是一个迭代的过程，不要奢望一蹴而就。很多时候，我们没有真正去实现限界上下文，就不可能完全正确地理解它。当我们距离真正理解业务还有一段距离时，不妨先草率地规划它，待到一切都明朗起来，再寻机调整。

#### **管理维度**

正如架构设计需要多个视图全方位体现架构的诸多要素，我们也应借助更多的维度全方位分析限界上下文。如果说从业务维度识别限界上下文更多的是从业务相关性（内聚）判断业务的归属，那么基于团队合作划分限界上下文的边界则是从管理维度思考和确定限界上下文合理的工作粒度。

管理层次的同构系统实现了架构系统与管理系统的映射，其中扮演关键作用的是限界上下文与领域特性团队之间的映射。这一映射的理论基础来自于康威定律。

在康威定律中，起到关键杠杆作用的是沟通成本。假设我们将同一个限界上下文的工作交给了两个不同的团队分工完成，为了合力解决问题，就必然需要这两个团队进行密切的沟通，导致了成本更高的跨团队沟通。团队的工作边界与限界上下文的业务边界形成了一种“阻抗不匹配”关系：必须调整工作分配方式，一个限界上下文的工作只能分配给一个团队，如此就能显著降低沟通成本，提高开发效率：   
  
那么，这是否意味着限界上下文与领域特性团队之间的关系就是一对一的关系呢？如果是，就意味着团队的工作边界与限界上下文的边界重合，自然是最佳的选择；可惜，限界上下文与团队的划分标准并不一致。如果目标系统为软件产品，领域特性团队可以在很长一段时间保证其稳定性，若为项目，参与研发的领域特性团队就很难做到稳定了。此外，团队的规模是可以控制的，限界上下文的粒度却要受到业务因素、技术因素以及时间因素的影响，要满足二者边界的完全吻合，实在有些勉为其难。

所谓“二者相害取其轻”，如果限界上下文无法与领域特性团队做到一对一的映射，至少要避免出现将一个限界上下文分配给两个或多个团队的情形。映射时，团队的工作边界又会反过来给划分限界上下文以启发，引发设计者以深思：“限界上下文的粒度是过大，还是过小？”当一个限界上下文的粒度过大，以至于计划中的功能特性完全超出了领域特性团队的工作量，就应该考虑继续分解限界上下文。因此，领域特性团队与限界上下文的映射关系应如下图所示：   
  
一个领域特性团队与限界上下文形成一对一或者一对多的关系，意味着项目经理需要将一个或多个限界上下文分配给大约7~10人的领域特性团队。因此，对限界上下文的粒度识别就变成了对团队工作量的估算。倘若以遵循INVEST原则[50]的用户故事作为待办项来衡量工作量，考虑用户故事之间存在的业务依赖和技术依赖，则每个限界上下文必然存在一个**最大并行开发度（Max Degree of Parallel Development，MDPD）**。领域特性团队受人数限制也存在一个**最大并行开发度**，其值可以借鉴精益看板提出的WIP Limits[51]（WIP即Work in Progress，在制品限制），则一个领域特性团队与多个限界上下文之间需要满足如下公式：

∑BC(MDPD)≈WIPLimits

以音乐网站的研发为例，如果一个领域特性团队同时承担了音乐播放、电台和KTV三个限界上下文的开发工作。假设该团队的WIP Limits为5，音乐播放上下文的最大并行开发度为4，电台上下文的最大并行开发度为2，KTV上下文的最大并行开发度为3，根据公式计算，这三个限界上下文的最大并行开发度之和为9，远远大于了团队的WIP Limits。若按照这一比例分配工作，就会导致限界上下文的开发周期延长；反过来，如果我们只将电台上下文分配给该团队，又会造成团队开发人员的浪费。

**边栏：**我在给某音乐网站进行领域驱动设计时，通过识别业务相关性划分了如下限界上下文：

* Media Player(online & offline)：提供音频和视频文件的播放功能，区分在线播放与离线播放
* Music：与音乐相关的业务，包括乐库、歌单、歌词。
* FM Radio：电台
* Live：直播
* MV：短视频和MV
* Singer：歌手
* Musician：音乐人，注意音乐人与歌手的区别
* Music Community：音乐社区
* File Sharing：包括下载和传歌等与文件有关的功能
* Tag：支持标签管理，包括音乐的分类如最新、话题等分类标签还有歌曲标签
* Loyalty：与提高用户粘度有关的功能，例如关注、投票、收藏、歌单等功能
* Utilities：音乐工具，包括音效增强等功能
* Recommendation：推荐
* Search：对整个音乐网站内容的搜索，包括对人、歌曲、视频等内容的搜索
* Activity：音乐网站组织的活动
* Advertisement：推广与广告
* Payment：支付

在识别限界上下文时，我将直播（Live）视为与音乐、电台、MV短视频同等层次的业务分类。然而，殊不知该音乐网站直播模块的开发团队已经随着功能的逐渐增强发展到了接近200人规模的大团队。这显然不是一个限界上下文边界可以控制的规模。即使属于直播业务的业务活动都与直播领域知识有关，我们也应该基于2PTs原则对直播限界上下文作进一步分解，以满足团队管理以及团队成员充分沟通的需要。

无论是限界上下文，还是领域特性团队，都会根据时过境迁发生动态的演化。康威就认为：“大多数情况下，最先产生的设计都不是最完美的，主导的系统设计理念可能需要更改。因此，组织的灵活性对于有效的设计有着举足轻重的作用。必须找到可以鼓励设计经理保持他们的组织精简与灵活的方法。”根据二者的映射关系，ThoughtWorks的技术雷达提出了“康威逆定律（Inverse Conway Maneuver）[52]”，即“围绕业务领域而非技术分层组建跨功能团队[53]“。这里所谓的“业务领域”，在领域驱动设计的语境中，就是指限界上下文。换言之，就是首先确定限界上下文的边界，再由此组织与之对应的领域特性团队。倘若限界上下文的边界发生演进，则领域特性团队也随之演进，以保证二者的匹配度。由于限界上下文与领域特性团队的边界相互影响，意味着在管理层面，每个领域特性团队的负责人也需要判断新分配的任务是否属于限界上下文的边界。不当的任务分配会导致团队边界的模糊，进而导致限界上下文边界的模糊，影响它对领域模型的控制。利用康威定律与康威逆定律，就能将限界上下文与领域特性团队结合起来，二者相互影响，形成了对限界上下文边界动态的持续改进。

领域特性团队对识别限界上下文的促进不只体现为团队规模传递的分解信号，它的组建原则同样有助于加深我们对限界上下文边界的认识。Jurgen Appelo认为，一个高效的团队需要满足两点要求[2]：

* 共同的目标
* 团队的边界

虽然Jurgen Appelo在提及边界时，是站在团队结构的角度来分析的；可在确定团队的工作边界时，恰恰与限界上下文的控制边界暗暗相合。建立一个良好的领域特性团队，需要保证：

* **团队成员应对团队的边界形成共识。**这意味着团队成员需要了解自己负责的限界上下文边界，以及该限界上下文如何与外部的资源以及其他限界上下文进行通信；同样，限界上下文内的领域模型本身就是在统一语言指导下达成的共识。
* **团队的边界不能太封闭（拒绝外部输入），也不能太开放（失去内聚力），即所谓的“渗透性边界[2]”。**这种渗透性边界恰恰与“高内聚松耦合”的设计原则完全契合。

针对这种“渗透性边界”，团队成员需要对自己负责开发的需求“抱有成见”。在识别限界上下文时，“任劳任怨”的好员工并不是真正的好员工。一个好的员工明确地知道团队的职责边界，他（她）应该学会勇于承担属于团队边界内的需求开发任务，也要敢于推辞职责范围之外强加于他（她）的需求。通过团队每个人的主观能动，渐渐形成组织结构的“自治单元”，进而催生出架构设计上的“自治单元”。同理，“任劳任怨”的好团队也不是真正的好团队。团队对自己的边界已经达成了共识，为什么还要违背这个共识去承接不属于自己边界内的工作呢？这并非团队之间的“恶性竞争”，也不是工作上的互相推诿；恰恰相反，这实际上是一种良好的合作，表面上是在维持自己的利益，然而在一个组织下，如果每个团队都以这种方式维持自我利益，反而会形成一种“互利主义”。

这种“你给我搔背，我也替你抓抓痒”的互利主义最终会形成团队之间的良好协作。如果团队领导者与团队成员能够充分认识到这一点，就可以从团队层面思考限界上下文。此时，限界上下文就不仅仅是架构师局限于一孔之见去完成甄别，而是每个团队成员自发组织的内在驱动力。当每个人都在思考这项工作该不该我做时，变相地就是在思考职责的分配是否合理，限界上下文的划分是否合理。

#### **技术维度**

架构是什么？Martin Fowler认为：架构是重要的东西，是不容易改变的决策[32]。如果我们未曾预测到系统存在的风险，不幸它又发生了，带给系统架构的改变可能是灾难性的。利用限界上下文的边界，就可以将这种风险带来的影响控制在一个极小的范围。为什么说限界上下文是领域驱动设计中最重要的元素，答案也在这里。

【边栏】我曾经负责开发一款基于大数据平台的BI产品，在架构设计时，对性能的评估方案存在问题。我们当时考虑了符合生产规模的数据量，并以一个相对可行的硬件与网络环境，对Spark+Parquet的技术选型进行测试。测试结果满足了设定的响应时间值。然而，两个因素的缺失为我们的架构埋下了祸根。在测试时，我们没有考虑并发访问量，测试的业务场景也过于简单。我们怀着一种鸵鸟心态，在理论上分析这种决策（Spark是当时最快速的基于内存的数据分析平台，Parquet是列式存储，尤为适合统计分析）是对的，然后就按照我们期望的形式去测试，实际上是将风险悄悄地埋藏起来。

当产品真正销售给客户使用时，我们才发现客户的业务场景非常复杂，对性能的要求也更加苛刻。例如它要求在较高的并发访问量前提下，对海量数据进行实时地统计分析与指标运算，以最低延迟的要求获得分析结果；同时，客户所能提供的Spark集群资源却是有限度的。事实上，基于Spark的driver-worker架构，它本身并不擅长完成**高并发**的数据分析任务。对于一个分析任务，Spark可以利用集群的力量由多个worker同时并行地执行成百上千个task，但瓶颈在driver端，一旦上游同时有多个请求涌入，响应能力就不足了。最终，我们的产品在真正的压力测试下一败涂地。

幸而，我们划定了限界上下文，并由此建立了数据分析微服务。针对客户高并发的实时统计分析需求，在保证REST API不变的情况下，我们更改了技术选型，选择基于ElasticSearch的数据分析微服务替换旧服务。这种改变几乎不影响产品的其他模块与功能，前端代码仅仅做了少量修改。3个人的团队在近一个月的周期内基本完成了这部分数据分析功能，及时掐断了炸药的导火线。

**质量属性**

从技术维度看限界上下文，我们需要关注的其实是质量属性（Quality Attributes）。

架构映射阶段虽然是以“领域”为中心的问题求解过程，但这并不意味着在整个过程中可以完全不考虑质量需求、技术因素和实现手段。对于一名架构师而言，考虑系统的质量属性应该成为一种工作习惯。John Klein、David Weiss就认为：“软件架构师的首要关注点不是系统的功能。……你关注的是需要满足的品质（即质量属性）。品质关注点指明了功能必须以何种方式交付，才能被系统的利益相关人所接受，系统的结果包含这些人的既定利益。[8]”。

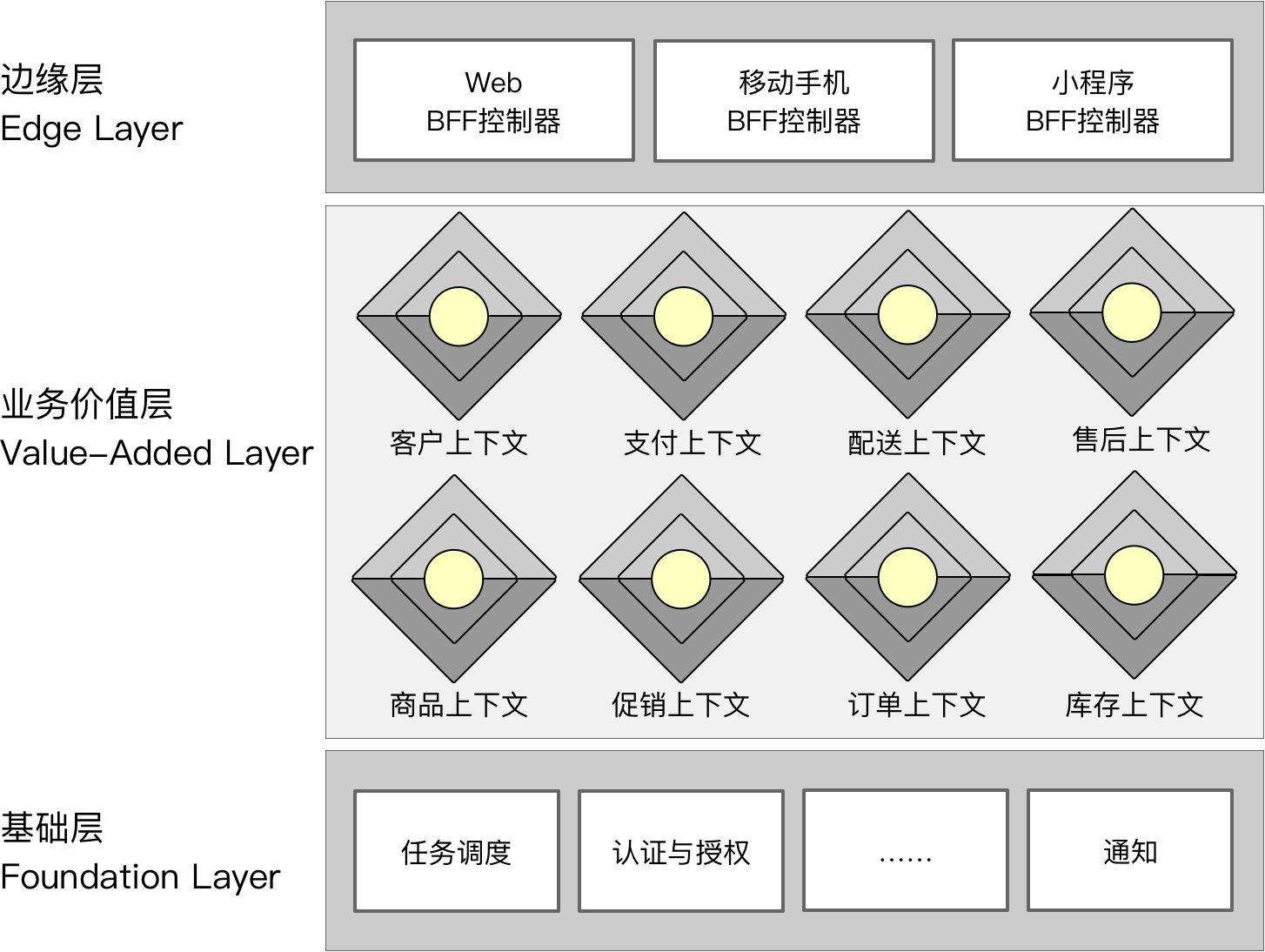
既要面向领域进行架构映射，又要确保质量属性得到满足，同时还不能让业务复杂度与技术复杂度混淆在一起，该如何兼顾？领域驱动架构风格要求：

* 识别限界上下文时，应以业务需求对边界的影响为第一序位，在限界上下文满足了业务需求之后，再考虑质量属性的影响
* 技术因素在影响限界上下文的边界时，仍然要保持领域模型的完整性与一致性

在确定了系统上下文即解空间的边界之后，业务级映射是从领域维度对整个目标系统进行的纵向切分，由限界上下文实现业务之间的正交性[4]。例如，订单业务与商品业务之间，它们各自业务的变化互不影响，只有协作关系会成为垂直相交的交点。在限界上下文内部，通过菱形对称架构【参考章节】定义了内部的领域层与外部的网关层，完成对限界上下文内部架构的内外切分，实现了业务功能与技术实现的正交性。例如，订单业务与订单数据库的操作，彼此的变化互不影响，它们之间的调用关系通过端口的解耦，形成了唯一依赖的垂直交点。

如果分析认为某个限界上下文的部分业务功能不能满足质量属性需求，就需要调整限界上下文的边界。虽然影响的因素是质量属性，但影响到的内容却是对应的业务功能，为了不破坏设计的正交性，就应该按照业务变化的方向进行切分，也就是通过纵向切分，将质量因素影响的那部分业务功能完整地分解出来，形成一个纵向的业务切片，而在限界上下文内部，仍然保持菱形对称架构，以隔离业务功能与技术实现，并在一个更小的范围维持领域模型的统一性和一致性。

如果质量因素影响的内容不仅限于原有的单个限界上下文，考虑到影响因素是一致的，说明引起变化的原因也是一致的，那么变化的频率也将保持一致，故而在纵向切分时，就需要对所有影响到的限界上下文进行纵向分割，形成一个个纵向的业务切片，再把这些切片聚集在一起，定义一个新的限界上下文。

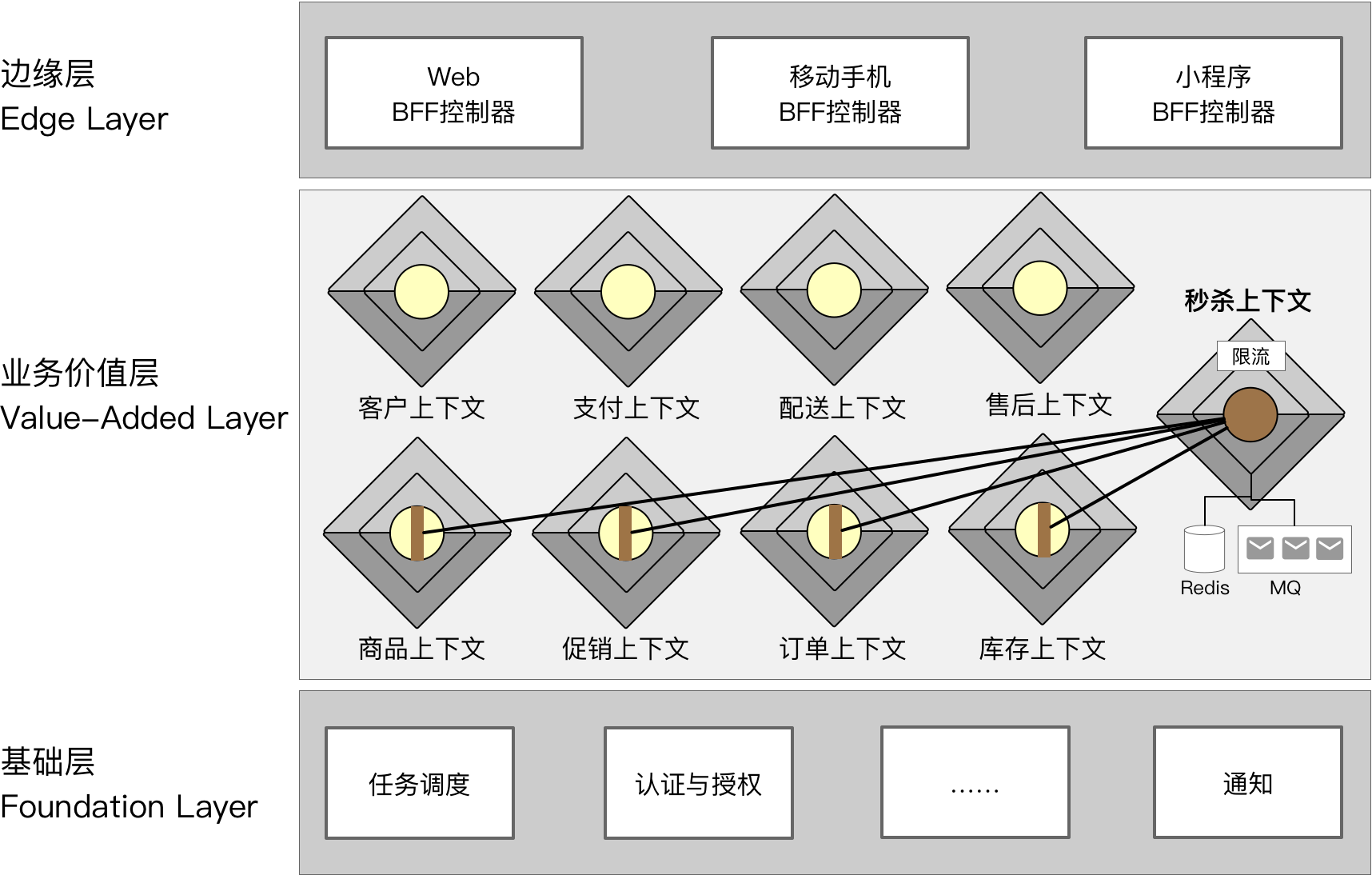
考虑电商平台开展的秒杀业务。一种秒杀方式同时规定了秒杀数量和秒杀时间，如果商品秒杀完毕又或者达到规定时间，就会结束秒杀活动；另一种秒杀是超低价的限量抢购，如一元抢购，价格只需一元，限量一件商品。无论秒杀采用什么样的业务规则，其本质仍然属于一种电商的营销行为，业务流程仍然还是下订单、扣除库存、支付订单、配送和售后，只是部分流程根据秒杀业务的特性进行了精简。遵循领域驱动架构风格，假定我们设计了如下由限界上下文构成的电商平台架构：   
  
从业务需求角度考虑，由于秒杀业务属于促销规则的一种，故而与秒杀有关的领域模型被定义在促销上下文内。买家完成秒杀之后，提交的订单、扣减的库存以及购买商品的配送，与其他营销行为并无任何区别，与秒杀相关的订单模型、库存模型以及配送模型也都放在各自的限界上下文内。当电商平台引入了新的秒杀业务后，原有架构的订单上下文、库存上下文与配送上下文都不需要调整，唯一要做的改变是修改促销上下文的领域模型，使其能够表达秒杀的领域知识，这正是正交性的体现。

然而，秒杀业务的增加不止于领域逻辑的变化，还包括其它特性：

* 瞬时并发量大：秒杀时会有大量用户在同一时间进行抢购，瞬时并发访问量突增数十乃至数百倍
* 库存量少：参与秒杀活动的商品库存量通常很少，只有极少数用户能够成功购买，这样才能增加秒杀的刺激性，并保证价格超低的情况下不至于影响销售利润

瞬时并发量大，就要求系统能在极高峰值的并发访问下，既保证系统的高可用，又要满足低延迟，确保用户的访问体验不受影响；库存量少，就要求系统能在极高峰值的并发访问下，在不影响正常购买的情况下避免超卖。在保证秒杀业务的高可用时，还必须满足其他业务功能的正常访问，不因为秒杀业务的高并发占用其他业务服务的资源。在高并发访问要求下，可扩展性、高可用性、资源独占性、数据一致性等多个质量属性决定了秒杀业务对已有架构产生了影响，除了必要的技术手段，如限流、削峰、异步、缓存外，还有一种根本的手段，就是分离秒杀业务与已有业务。从领域驱动设计的角度讲，秒杀业务的技术因素影响了电商平台的限界上下文划分。

遵循领域驱动架构风格，虽然这些技术因素看起来影响了系统的技术实现层面，但我们对现有架构的改造并不是提炼出与技术因素有关的模块，如异步处理模块、缓存模块，因为这些模块从变化频率与方向看，不只是为秒杀业务提供支撑，而是面向了整个系统，属于系统分层架构定义的基础层内容。

分析引起架构调整的这些质量属性需求，导致变化的主因还是因为秒杀业务，自然就该在各个限界上下文中提炼出与秒杀业务相关的功能，形成纵向的秒杀业务切片。这些业务切片仍然按照领域层与网关层的内外层次组织，形成一个专有的限界上下文——秒杀上下文，如下图所示：   
  
秒杀上下文相当于为秒杀业务建立了一个独立王国。提供秒杀业务的商品、定义秒杀业务的促销规则、秒杀订单与库存都在秒杀上下文的领域模型中。当秒杀请求从客户端传递到秒杀限界上下文时，可对北向网关的远程服务进行技术调整，如增加专门的限流功能，只允许少部分流量进入服务端；在提交订单请求时，领域层会调用南向网关的资源库端口，也可以对端口的资源库适配器进行技术调整，并不直接访问秒杀的订单数据库，而是通过访问Redis缓存数据库，提供访问效率。客户在支付秒杀订单时，仍然发生在秒杀限界上下文内部，可选择由北向网关的本地服务发布PaymentRequested事件，引入消息队列作为事件总线完成异步支付，以减轻秒杀服务端的压力。最关键之处在于当我们引入了这样一个纵向切片的秒杀上下文后，就可以将秒杀的整体业务从已有系统中独立出来，为其分配单独的资源，保证资源的隔离性以及秒杀服务自身的可伸缩性。

【注：这里针对秒杀系统的架构设计是从领域驱动设计的角度进行思考的。因为存在高并发访问这一风险，从而驱动我们为秒杀定义专门的限界上下文。在确定了秒杀上下文的边界后，再针对秒杀业务进行领域建模，同时运用诸如限流、削峰、缓存等技术手段，才是我认为的最佳方式。具体的技术手段，可以参考曹林华的文章《秒杀架构设计》（<https://blog.51cto.com/13527416/2085258>）】

**重用和变化**

无论是重用领域逻辑还是技术实现，都是在设计层面上我们必须考虑的因素。需求变化更是影响设计策略的关键因素。基于限界上下文自治性的四个特征，可以认为这个自治的单元其实就是逻辑重用和封装变化的设计单元。这时，对限界上下文边界的考虑，更多是出于技术设计因素，而非业务因素。

运用重用原则分离出来的限界上下文往往对应于支撑子领域，它作为支撑功能可以同时服务于多个限界上下文。我曾经为一个多式联运管理系统提供领域驱动设计咨询，通过与领域专家的沟通，我注意到他在描述运输、货站以及堆场的相关业务时，都提到了作业和指令的概念。虽然属于不同的领域，但指令的收发、作业的制订与调度都是相同的，区别只在于作业与指令的内容，以及作业调度的周期。为了避免在运输、货站与堆场各自的限界上下文中重复设计与实现作业与指令等领域模型，我们可以将作业与指令单独划分到一个专门的限界上下文中。它作为上游限界上下文，提供对运输、货站与堆场的业务支撑。

限界上下文对变化的应对，其实是“单一职责原则”的体现，即一个限界上下文不应该存在两个引起它变化的原因。还是这个物流联运管理系统，最初团队的设计人员将运费计算与账目、结账等功能放在了财务上下文中。当国家的企业征税策略发生变化时，会引起财务上下文的变化，引起变化的原因是财务规则与政策的调整。倘若运费计算的规则也发生了变化，同样会引起财务上下文的变化，但引起变化的原因却是物流运输的业务需求。如果我们将运费计算单独从财务上下文中分离出来，就可以独立演化，符合限界上下文的自治原则，实现了两种不同关注点的分离。

**遗留系统**

限界上下文自治原则的唯一例外是遗留系统，因为领域驱动设计往往建议将整个遗留系统视为一个限界上下文。那么，什么是遗留系统？根据维基百科的定义，它是一种旧的方法、旧的技术、旧的计算机系统或应用程序。这个定义并不能解释遗留系统的真相。我认为，系统之所以成为遗留系统，关键在于**知识的缺乏**。文档不够全面真实，掌握系统知识的团队成员泰半离开，系统的代码可能是一个大泥球。因此，我对遗留系统的定义是“**一个还在运行和使用，但已步入软件生命衰老期的缺乏足够知识的软件系统**”。

如果目标系统需要与遗留系统协作（注意，它并非系统上下文识别的伴生系统），通常需要为它单独建立一个限界上下文。无论该遗留系统是否定义了领域模型，都可以通过限界上下文的边界作为屏障，以避免遗留系统的混乱结构对系统整体架构的污染，也可以避免开发人员在开发过程中陷入遗留系统庞大代码库的泥沼。

【边栏】我曾经参与了一套医疗内容管理系统的开发，当时，该系统已经在北美多家医疗机构运行了数年时间。我们的任务是为现有系统增加一个全新的Find & Replace模块，以便于为系统中的医疗内容提供针对医疗术语、药品以及药品成分的查询与替换。查询和替换的过程有着非常复杂的与医疗行业有关的业务逻辑。从领域驱动设计的角度讲，Find & Replace模块就是我们的目标系统，原有系统可视为与之集成的遗留系统。在多年的系统发展下，原有系统已经定义了属于自己的领域模型。刚开始开发时，我们尝试去重用这些领域模型，确实为我们省去了不少定义领域模型类的繁琐工作，但随着开发工作的逐渐推移，我们发现这些领域模型不能完全符合Find & Replace的业务要求，使得我们的开发变得束手束脚。当我们决定抛开这些已有的领域模型后，突然有一种退一步海阔天空的感觉，豁然觉得前面的路程反而更加明朗了。

系统之所以要将现有遗留系统当做一个限界上下文，要么是因为还要继续维护遗留系统，满足新增需求，要么是系统的一部分业务功能需要与遗留系统集成。对于前者，遗留系统限界上下文定义了一个独立进化的自治边界，它能小心翼翼控制新增需求的代码，并以适合遗留系统特性的方式自行选择开发与设计模式；对于后者，与之集成的限界上下文由于遵循菱形对称架构风格【参考章节】，即可通过南向网关的客户端端口来固定当前限界上下文不受遗留系统的影响，甚至可以通过此方式，慢慢对遗留系统进行重构。在重构的过程中，仍然需要遵循自治原则，站在调用者的角度观察遗留系统，考虑如何与它集成，然后逐步对其进行抽取与迁移，形成自治的限界上下文。

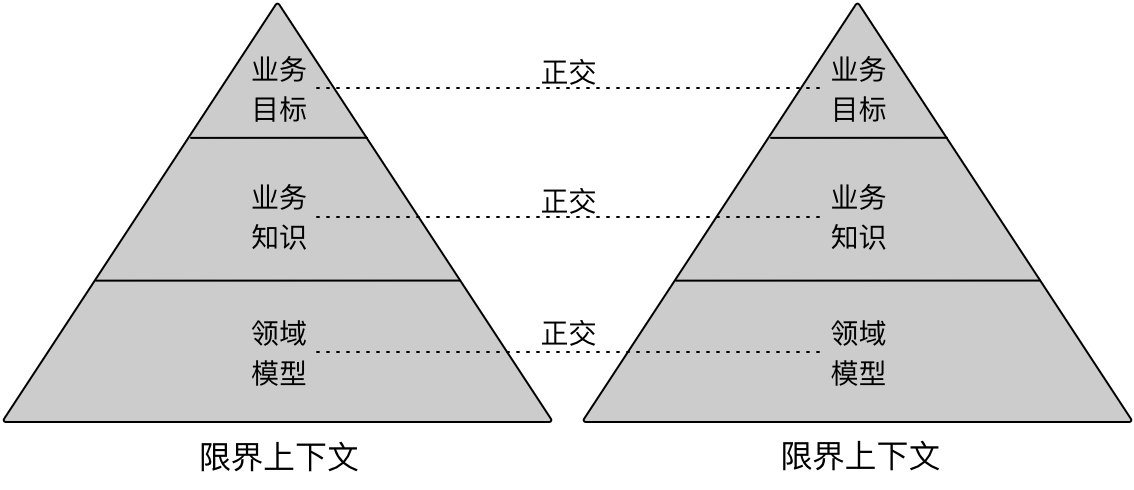
### **限界上下文的验证原则**

从业务维度、团队维度与技术维度识别限界上下文，就会形成一种层层推进的设计驱动力，可以让我们对限界上下文的设计变得更加准确，边界的控制变得更加合理，毕竟，限界上下文的识别对于整个系统的架构至关重要。如此，我们也还未到额手相庆的时候，目前获得的限界上下文只能说是候选限界上下文。我们还需要遵循设计原则对限界上下文边界的合理性进行验证。验证的设计原则包括：正交原则与单一抽象层次原则。

#### **正交原则**

正交性要求：“如果两个或更多事物中的一个发生变化，不会影响其他事物，这些事物就是正交的。[4]【注】”变化的影响主要体现在变化的传递性，即一个事物的变化会传递到另一个事物引起它的变化，但这个变化影响并不包含彼此正交的点。例如，限界上下文之间存在调用关系，当被调用的限界上下文公开的接口发生变化，自然会影响调用方。这一影响是合理的，也是软件设计很难避免的依赖。故而限界上下文存在正交性，指的是各自边界封装的业务知识不存在变化的传递性。

【注：“正交性”是从几何学中借来的术语。如果两条直线相交成直角，它们就是正交的。用向量术语说，这两条直线互不依赖。沿着一条直线移动，你投影到另一条直线上的位置不变。[49]】

要破除变化的传递性，就要保证每个限界上下文有且仅为一个业务目标服务；要让各自服务的业务目标不能出现雷同，就要保证完成该业务目标所需要的业务知识不能出现交叉；要让业务知识不能出现交叉，就要保证封装了业务知识的领域模型不能出现重叠。领域模型、业务知识、业务目标，三者之间存在层次的递进关系，无论是自顶向下去推演，还是自底向上来概括，都不允许同一层次之间存在非正交的事物：   
  
领域模型的重复，意味着各自定义的领域模型对象表达了同一个领域概念。注意，限界上下文展现的领域概念具有知识语境，不能因为领域概念名称相同就认为领域概念出现了重复。判断领域模型的重复性，必须将限界上下文作为修饰，将二者组合起来共同评判。例如，在供应链系统中，商品限界上下文、运输限界上下文与库存限界上下文的领域模型都定义了Product类，但结合各自的知识语境，这一领域模型类实际代表了不同的领域概念；在保险系统，车险限界上下文、寿险限界上下文的领域模型都定义了Customer类，但购买车险和寿险的人可以是同一个客户，代表了相同的领域概念，导致领域模型的重复。

业务知识的重复，代表了业务问题的解决方案出现了重复，通常包含了领域行为与业务规则，例如在电商系统中，运费计算的规则不能同时存在于多个限界上下文，在订单上下文中计算了该订单的运费，在配送上下文同样计算了配送物品的运费。这样的重复会导致解决方案蔓延在系统各处，一旦运费计算规则发生变化，就需要同时修改多个限界上下文，遗漏了修改还会引入潜在的缺陷。

业务目标体现了限界上下文在解空间中存在的必要性，属于业务价值的体现。如果业务目标重复了，就意味着一个限界上下文可以替换另一个限界上下文的全部或部分工作。例如，物流系统的地图上下文与导航上下文的业务目标都是为配送路线提供向导功能，导航上下文不仅支持地图导航，还包含了语音导航，地图上下文就可以被导航上下文替代。而在地图系统中，地图上下文的目标是提供地理位置定位与地图指引与搜索服务，它就不再与导航上下文的业务目标产生冲突了，二者就可以并存。

判断限界上下文是否遵循正交性，没有客观明确的标准。依据领域模型、业务知识与业务目标的层次关系，在识别限界上下文的架构映射阶段，可以尝试为每个限界上下文定义一个清晰的业务目标，最好能用一句话概括，就可通过业务目标快速确定正交性。业务目标一旦确定，就可以反向判断放在边界内的业务活动是否都满足这一最高层次的业务目标。如果一个限界上下文中包含的业务活动存在主次之分，只要那些次要的业务活动是为整体的业务目标提供了服务支撑，就没有违背正交性；当然，也可以将所有支撑该业务目标的业务活动整合起来，为其单独创建一个限界上下文，因为它们具有共同的次要业务目标。例如，计算距离最短路径、计算时间最短路径、计算费用最少路径都是为导航业务目标提供支撑，将它们放在导航限界上下文，并不算是业务目标的偏离，只是显得有些主次不分罢了，但可以将它们视为导航限界上下文的隐私，形成内部的能力支撑；考虑到这些支撑型的业务活动都与路径规划有关，也可以考虑将它们整合起来，单独创建路径规划限界上下文。此时，导航限界上下文提供优质的导航服务，路径规划限界上下文提供高效准确的路径规划能力，二者的业务目标就是正交的，虽然后者是为前者提供了支撑。

如果出现了左右摇摆的业务活动，就需要辨别它们与限界上下文关联的强弱程度。判断强弱度，可以结合变化，当变化发生时，对谁的影响更大，就意味着关联更强。例如，统计实时值机人数、实时登机人数、实时安检人数等业务活动，究竟该放在航班限界上下文，还是旅客限界上下文？从变化分析，例如当实时值机人数发生了变更，究竟对当前航班影响更大，还是对某位旅客的信息影响更大？毫无疑问，实时值机人数是与具体的航班粘连在一起的，应划分到航班限界上下文。

#### **单一抽象层次原则**

单一抽象层次原则（Single Level of Abstraction Principle，SLAP）来自Kent Beck的编码实践，他在组合方法（Composed Method）模式中指出：保证一个方法中的所有操作都在同一个抽象层次[55，P24]；不过这一原则却是由Gleann Vanderburg在理解了这一概念之后提炼出来的[56，P172]。识别限界上下文时，归纳业务知识的过程就是抽象的过程，限界上下文的名称代表了一个抽象的概念，因此，我们可以将单一抽象层次原则运用到限界上下文的识别过程中。

要理解单一抽象层次原则，就需要了解什么是概念的抽象层次？

首先，抽象这个词的拉丁文为abstractio，原意为排除、抽出。中文对这个词语的翻译也很巧妙，可以顾名思义理解为抽出具体形象的东西。例如人是一个抽象的概念，一个具体的人有性别、年龄、身高、相貌、社会关系等具体特征，而抽象的人就是不包含这些具体特征的一个概念。抽象概念指代一类事物，因此，抽象实际上并非真正抽出这些具体特征，而是对一类具有共同特征的事物进行归纳，从而抹掉了具体类型之间的差异。

抽象层次则与概念的内涵有关，概念的内涵即为事物的特征。内涵越小，意味着抽象的特征越少，抽象的层次就越高，外延也越大，反之亦然。例如男人和女人有性别特征的具体值，人抽象了性别特征，使得该概念的内涵要少于男人或女人，而外延的范围却更大，抽象层次也就更高。同理，生物的概念层次要高于人，物质的概念层次又要高于生物。

倘若限界上下文没有遵循单一抽象层次原则，概念层次就会出现混乱。一个高抽象层次的概念由于内涵更小，使得它的外延更大，就有可能包含低抽象层次的概念，使得位于不同抽象层次的限界上下文存在概念上的包含关系，也就违背了正交原则。例如，在一个集装箱多式联运系统中，商务限界上下文与合同限界上下文就不在一个抽象层次上，因为商务的概念实际涵盖了合同、客户、项目等更低抽象层次的概念；运输、堆场、货站限界上下文遵循了单一抽象层次原则，运输限界上下文是对运输计划和路线的抽象，堆场限界上下文是对铁路运输场区概念的抽象，货站限界上下文则是对公路运输站点工作区域相关概念的抽象，它们关注的业务维度可能并不相同，但不影响它们的抽象层次位于同一条水平线上。

限界上下文的概念位于同一个抽象层次，并不意味着业务目标位于同一个层次。业务目标存在主次之分，然而主要的业务目标可以与次要的业务目标位于同一个抽象层次，因为抽象层次与重要程度无关。仍然是在集装箱多式联运系统，运输、堆场以及货站等限界上下文都需要作业和作业指令，区别在于操作的作业内容不同。提炼出来的作业上下文为运输、堆场以及货站限界上下文提供了业务功能的支撑，但并不影响它们作为系统的限界上下文共同存在。

限界上下文作为高层的抽象机制，体现了我们在软件构建过程中对领域思考的本质，它是架构映射阶段的核心模式。因此，限界上下文的识别直接影响了领域驱动设计的架构质量。通过分解、归类与归纳到最后的验证，若我们对识别出来的限界上下文的准确性依然心存疑虑，比较务实的做法是**保证限界上下文具备一定的粗粒度**。这正是奥卡姆剃刀原理的体现，即“切勿浪费较多东西去做用较少的东西同样可以做好的事情”，更文雅的说法就是“如无必要，勿增实体”[57, P352]，也就是当我们没有寻找到必须切分限界上下文的必要证据时，就不要增加新的限界上下文。倘若觉得功能的边界不好把握分寸，可以考虑将这些模棱两可的功能放在同一个限界上下文中。待到该限界上下文变得越来越庞大，以至于一个领域特性团队无法完成交付目标；又或者违背了限界上下文的自治原则，或者质量属性要求它的边界需要再次切分时，再对该限界上下文进行分解，增加新的限界上下文。这才是设计的实证主义态度。