

ThoughtWorks洞见

# 领域驱动设计



[insights.thoughtworks.cn/tag/ddd](https://insights.thoughtworks.cn/tag/ddd)



## 综述

01

DDD战略篇：架构设计的响应力 ······	2
DDD战术篇：领域模型的应用 ······	7
DDD实战篇：分层架构的代码结构 ······	14
DDD的终极大招——By Experience ······	22

## 通用语言、领域、限界上下文 25

重读领域驱动设计——如何说好一门通用语言 ······	26
当Subdomain遇见Bounded Context ······	30

## 架构

36

从三明治到六边形 ······	37
端口和适配器架构——DDD好帮手 ······	45

## 领域事件

59

识别领域事件 ······	60
在微服务中使用领域事件 ······	63
当提到“事件驱动”时，我们在说什么？ ······	73

# 目录

## 微服务

77

DDD & Microservices .....	78
服务拆分与架构演进 .....	83
溯源微服务：企业分布式应用的一次回顾 .....	94

## 示例实现

100

后端开发实践系列——开发者的第0个迭代 .....	101
后端开发实践系列——领域驱动设计(DDD)编码实践 .....	124
后端开发实践系列——事件驱动架构(EDA)编码实践 .....	151
后端开发实践系列——简单可用的CQRS编码实践 .....	177
用DDD实现打卡系统 .....	196

## 扩展阅读

199

DDD该如何学？ .....	200
领域驱动设计(DDD)实现之路 .....	203
从“四色建模法”到“限界纸笔建模法” .....	211
可视化架构设计——C4介绍 .....	221
从架构可视化入门到抽象坏味道 .....	230
技术债治理的四条原则 .....	235



ThoughtWorks洞见  
领域驱动设计

---

# 综述

# DDD战略篇：架构设计的响应力

作者：肖然

当敏捷宣言的17位签署者在2001年喊出“响应变化胜于遵循计划”这样的口号时，鲜有组织会真正把这句话当回事儿，甚至很多经验丰富的管理者会认为好的计划是成功的一半，遵循计划就是另外一半。然而在时下的第四次工业革命浪潮中，可能很多管理者已经不会简单满足于“响应”，而是选择主动发起变化了。不确定性管理成了这个时代的主旋律，企业的响应力成了成败的关键。

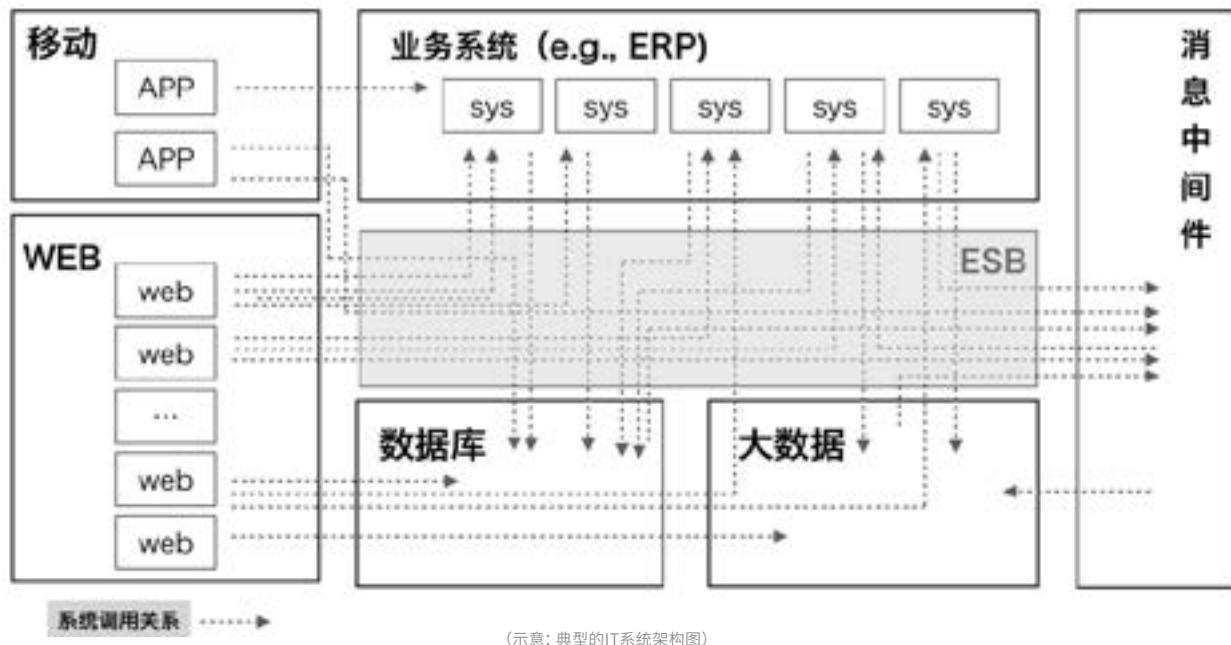
随着这种趋势的深入，架构设计这个技术管理领域也被推到了风暴边缘。“稳定”这个过去我们用来形容好系统的词语似乎已经失去原有的含义，很多人开始用“健壮”这个词语来形容好的系统。比如Netflix公司采用的Chaos Monkey机制随机主动关停线上服务而不

会造成整个服务生态宕机的作法更多的是在测试系统的健壮性，保证不会因为某个局部的问题而造成全身瘫痪。

然而架构的健壮性却比较难于定义和测试，以至于很多时候咱们在架构设计上还是在追求稳定性。在一个典型的企业IT组织里，当你询问一位资深工程师架构设计时，往往会得到一张搭积木一样的“架构图”。

图的底层是各种数据存储（从经典的Oracle到大数据标配的Hadoop），图的中间是类似Kafka这样的消息管道和传统的ESB（消息总线），上层则是各种业务应用（包括各种Web应用和移动的APP）。

仿佛这是一个流行的“稳定”架构设计。



当询问这样的架构是否合理时，不少人会告诉你问题可大了：这不是云时代的的服务化架构。原因是这个架构的大部分组件，如数据存储，都已经可以完全“托管”给云平台了。于是乎，很多企业架构师又开始寻找像过去ESB一样能够对接各种云平台的PaaS了，然后抱怨现在的PaaS没有当年的ESB“稳定”。

两个核心问题却很少被提及：

1. 当年基于ESB集成的SOA服务化架构解耦出的组件不但没有提升效率，反而增加了系统后续修改的复杂度。
2. 看似“以不变应万变”的架构并不能支撑多样化的业务需求，最后各个业务部门仍然有一套自己的IT系统，即
3. 画出来的架构图惊人的相似（多少次有人惊呼“这就是我们之前那个工作流系统~”）。

就这两个核心痛点，让我们一起来谈谈架构设计面临的挑战和应对方式。

## 什么是架构设计？

由于软件设计是一个复杂度很高的活动，“通过组件化完成关注点分离从而降低局部复杂度”很早就成为了咱们这个行业的共识。前面提到的数据存储、消息管道等“模块”在某种意义上都是组件化的产物。这样的好处是在不同系统里遇到同样的功能需求时可以复用。在云服务崛起的今天，这样的组件以“服务”的形式更容易为我们所采用。

当然技术出身的架构师们在架构设计的时候或多或少都有一种“搭积木”的感觉。大家都非常关注Kafka有哪些功能，K8S是不是比Mesos功能更全，以及Akka是不是稳定。就像走进一个家装公司，在选择了“套餐”之后有工程人员给你介绍地砖和木地板用哪个品牌更好。



回到咱们的第二个核心痛点，如果只是这样的搭积木，为什么咱们总是在面对新变化、新需求的时候发现需要新的组装方式或新的组件呢？这样的架构设计对比直接按照需求实现（不考虑架构）有什么优势呢？

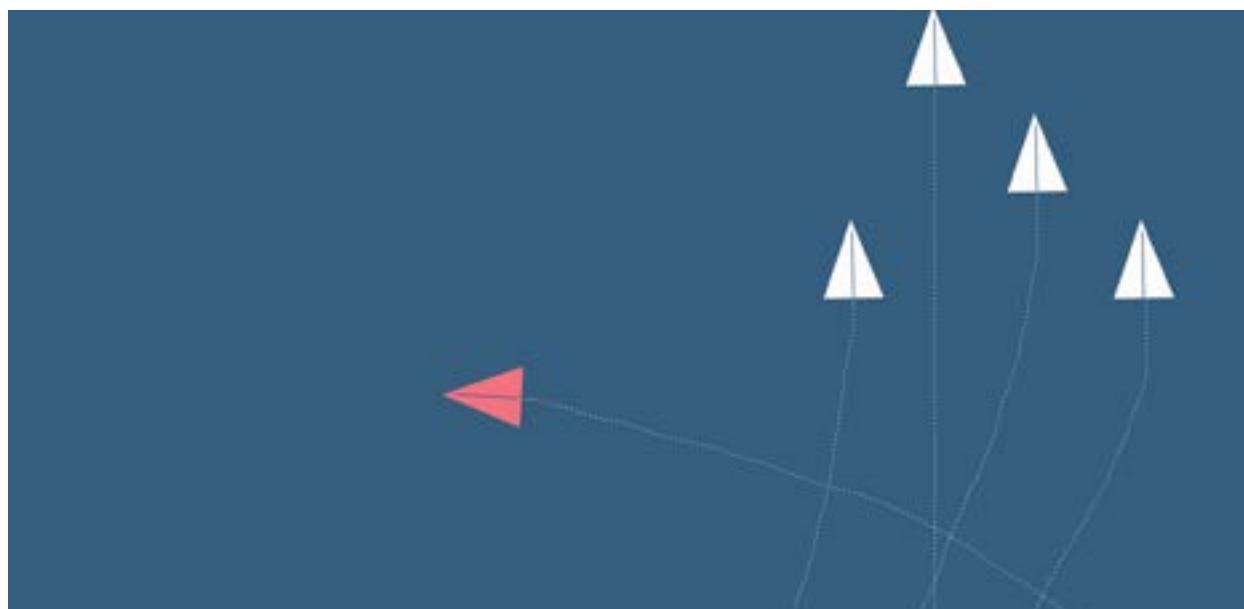
这里我们应该回到架构设计的本质，即为什么我们要在代码实现前做设计。显然如果去掉设计这个过程，大家会说问题这么复杂，如何下手啊？所以设计**首先是要解决问题的复杂度**。于是有人做了一个架构，交给了一个团队去实现，很快发现实现的架构和设计完全是两张皮。当然原因很明确——缺少了交流和沟通，所以设计**其次是要建立团队协作沟通的共识**。

假设我们产生了一个团队都达成共识的架构设计，大家都兢兢业业把设计变成了现实。一个长期困扰软件行业的问题出现了，需求总是在变化，无论预先设计如何“精确”，总是发现下一个坑就在不远处。相信很多技术人员都有这样的经历，结果往往是情况越来越糟糕，也就是我们常说的架构腐化了，最后大家不得不接受重写。这些经历让我们逐步明确了软件架构设计的实质是**让系统能够更快地响应外界业务的变化，并且使得系统能够持续演进**。在遇到变化时不需要从头开始，保证实现成本得到有效控制。

## 面向业务变化而架构

基于上面的架构设计定义，关键因素就是业务变化。显然这个时代的业务变化是很快的，甚至很多业务主动在变，不变则亡是很多行业目前的共识。变化速度给架构设计带来了很大挑战，一个移动APP可能需要在一周内上线，然而为了支撑这个移动APP的后台服务，平台发布窗口是每两个月一次。这样的不匹配在IT领域里是随处可见的现实，我们习惯性地认为后台天然就很重因此很慢，只可能在牺牲质量的情况下满足这样的速度。

然而事实上这样的健壮架构确实是存在的，看看身边现在无处不在的互联网，又有哪一个企业的架构比之复杂呢。互联网系统的组件是一个个网站，每个网站完成着自己的业务功能更新，从新闻发布到在线聊天。而各个站点又是紧密互联的，聊天网站可能把新闻网站拿到的信息实时推送给在线的用户。每个网站都是独立的小单元，面向互联网用户提供着一定的业务服务。好的网站也根据用户的反馈在不停升级和变化，但这样的变化并不影响用户使用其它的网站。



从互联网架构我们可以学到什么呢？从架构设计角度我认为以下三点是关键。

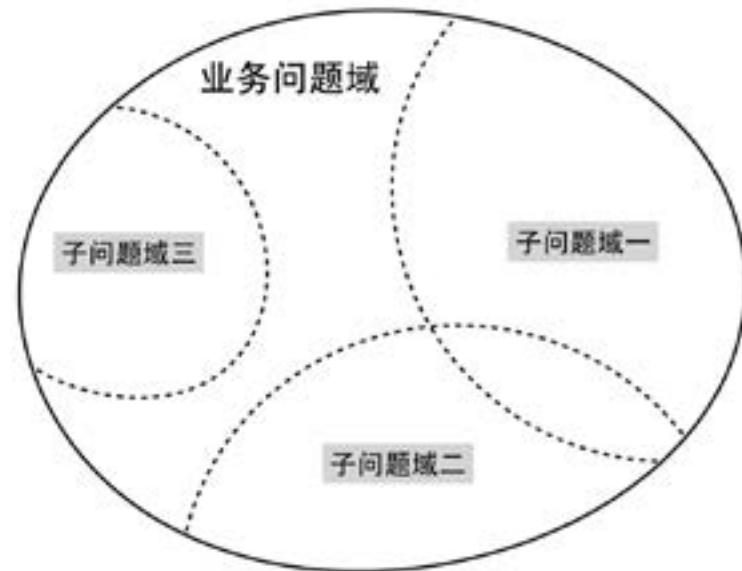
- 1.让我们的组件划分尽量靠近变化的原点，对于互联网来说就是用户和业务，这样的划分能够让我们将变化“隔离”在一定的范围（组件）内，从而帮助我们有效减少改变点。
- 2.组件之间能够互相调用，但彼此之间不应该有强依赖，即各自完成的业务是相对独立的，不会因为一方掉线而牵连另外一方，比如新闻网站挂掉了，聊天网站应该继续正常提供服务，可能提示用户暂时无法提供新闻信息而已。
- 3.组件在业务上是鼓励复用的，正是这样的复用才成就了今天的互联网，我们不会每个网站都去实现一个强大的搜索引擎。而被“复用”最多的网站显然会受到追捧，成为明星业务。当然架构上这样的网站必然是健壮的。

上面的三点毫无疑问都指向了业务，**从业务出发、面向业务变化是我们现代架构设计成功的关键。架构设计的核心实质是保证面对业务变化时我们能够有足够的响应能力。**

这种响应力体现在新需求（变化）的实现速度上，也体现在我们组件的复用上，在实现过程中现有架构和代码变化点的数量也是技术人员能够切身体会到的。面对日新月异的数字化时代，组织的整体关注点都应该集中到变化的原点，即业务上，而架构应该服务于这种组织模式，让这样的模式落地变得自然。

对比之前的传统SOA架构，这个思路的变化是本质性的。类似工业总线（ESB）这样的组件化其实是面向技术的，希望通过技术平台的灵活性来解决业务变化的多样性。虽然短时间能够收到一定的成效，长期看必然把自身做成瓶颈，因为所有业务的变化最后都堆积到了这个技术组件来解决。这也回答了为什么实施了传统SOA架构的企业最后都发现响应速度其实并没有提升起来。

面向业务变化而架构就要求首先理解业务的核心问题，即有针对性地进行关注点分离来找到相对内聚的业务活动形成子问题域。子问题域内部是相对稳定的，即未来的变化频率不会很高，而子问题边界是很容易变化的，比如在一个物流系统中：计算货物从A地到B地的路径是相对固定的，计算包裹的体积及归类也是相对固定的，但根据包裹的体积优化路径却经常会根据业务条件而变化。



(子问题域的划分)

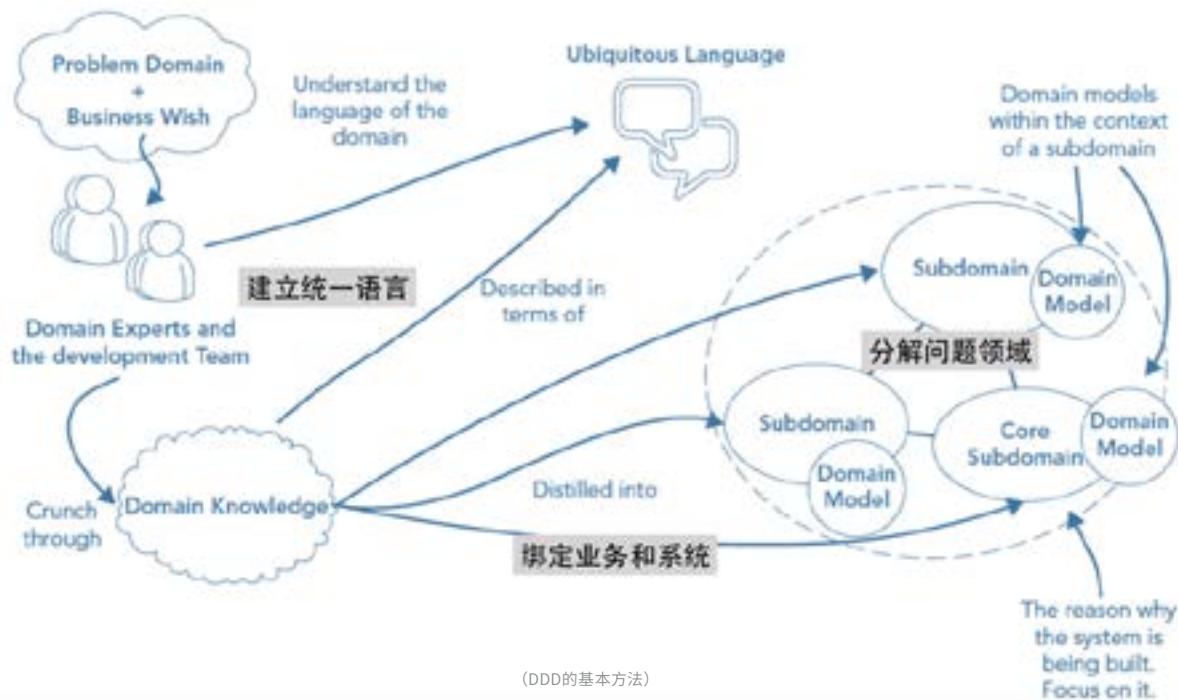
# 打造架构响应力的方法

如果认同了上述现代架构的真正意义，大家一定会问怎么才能打造这样的高响应力架构呢？

领域驱动设计方法DDD (Domain Driven Design) 为我们提供了很好的切入点。这个2003年就总结出来的方法终于在10多年后重新走入了架构师的视野，而这一次大家已经意识到了这种方法在这个快速变化时代的重要性。DDD通过以下两个模式去有效解决了文章开始提到的两大痛点：

- 1.让团队中各个角色（从业务到开发测试）都能够采用统一的架构语言，从而避免组件划分过程中的边界错位。
- 2.让业务架构和系统架构形成绑定关系，从而建立针对业务变化的高响应力架构。

这两点是DDD的核心，也是为什么时下全球架构圈在进一步向DDD这个方向靠拢的原因。DDD明确了业务和系统架构上的绑定关系，并提供了一套元语言来帮助各个角色有效交流架构设计。



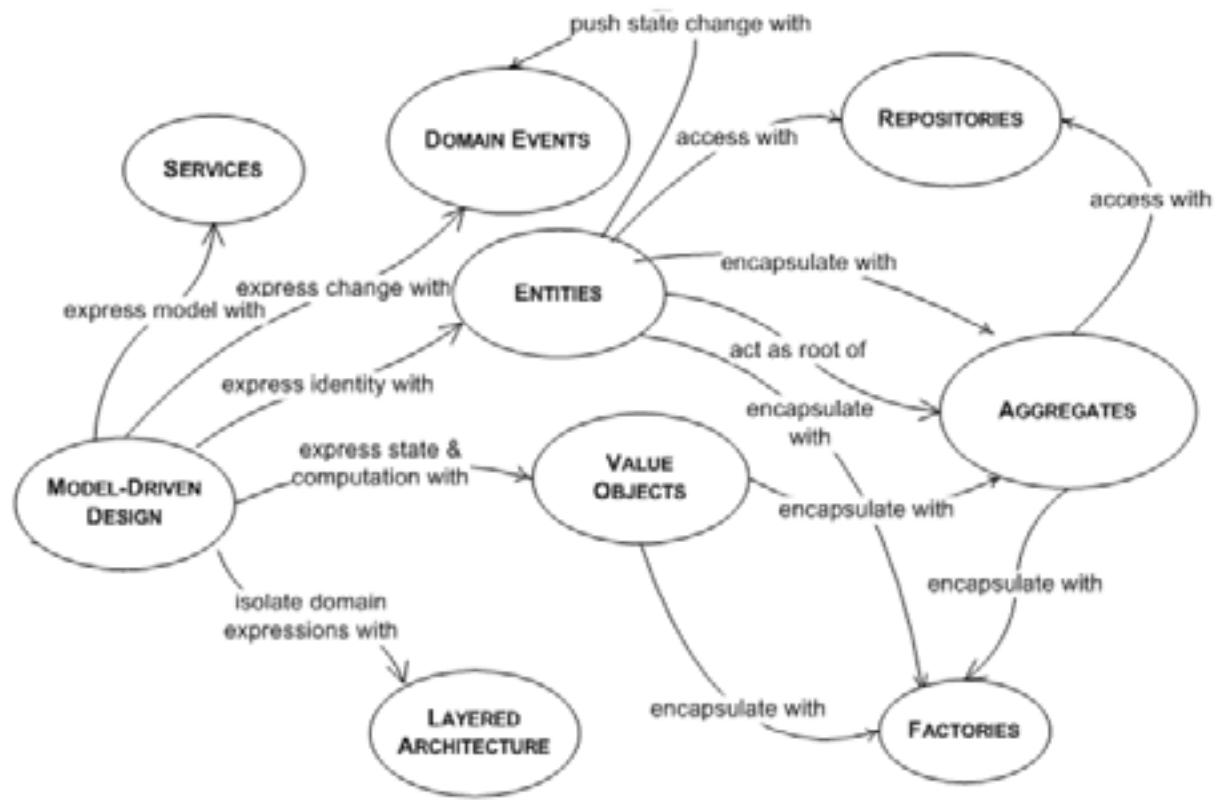
在战略层面，DDD非常强调针对业务问题的分析和分解，通过识别核心问题域来降低分析的复杂度。在战术层面，DDD强调通过识别问题域里的不同业务上下文来进行面向业务需求的组件化。最后在实现层面利用成熟的技术模式屏蔽掉技术细节的复杂度。

在这里我们也希望通过第一届DDD China建立起一个架构设计人员的交流平台。期待更多的中国技术人员能够通过这个平台和世界一流架构大师们建立起沟通的渠道，不仅在战略层面，也在战术层面和所有人一起分享讨论关于DDD的一切。

# DDD战术篇：领域模型的应用

作者：肖然

领域驱动设计DDD在战术建模（后文简称建模，除非特别说明）上提供了一个元模型体系（如下图），通过这个元模型我们会对战略建模过程中识别出来的问题子域进行抽象，而通过抽象来指导最后的落地实现。



(DDD构建的元模型元素脑图)

这里我们谈的战术阶段实际就是这样一个抽象过程。这个抽象过程由于元模型的存在实际是一定程度模式化的。这样的好处是并非只能技术人员参与建模，业务人员经过一定的培训也是完全可以理解的。在带领不少团队实践建模的过程中，业务人员参与战术设计也是我要求的。

由于已经有不少书籍介绍DDD的元模型，这里我们就不再赘述，转而谈谈这个抽象过程中大家经常遇到的一些困惑。这些比较常见的问题可能是DDD元模型未来演进需要解决的，但我们仍然要注意业务问题和架构设计的多样性，不要过度规范，以至于过犹不及。

## 业务对象的抽象

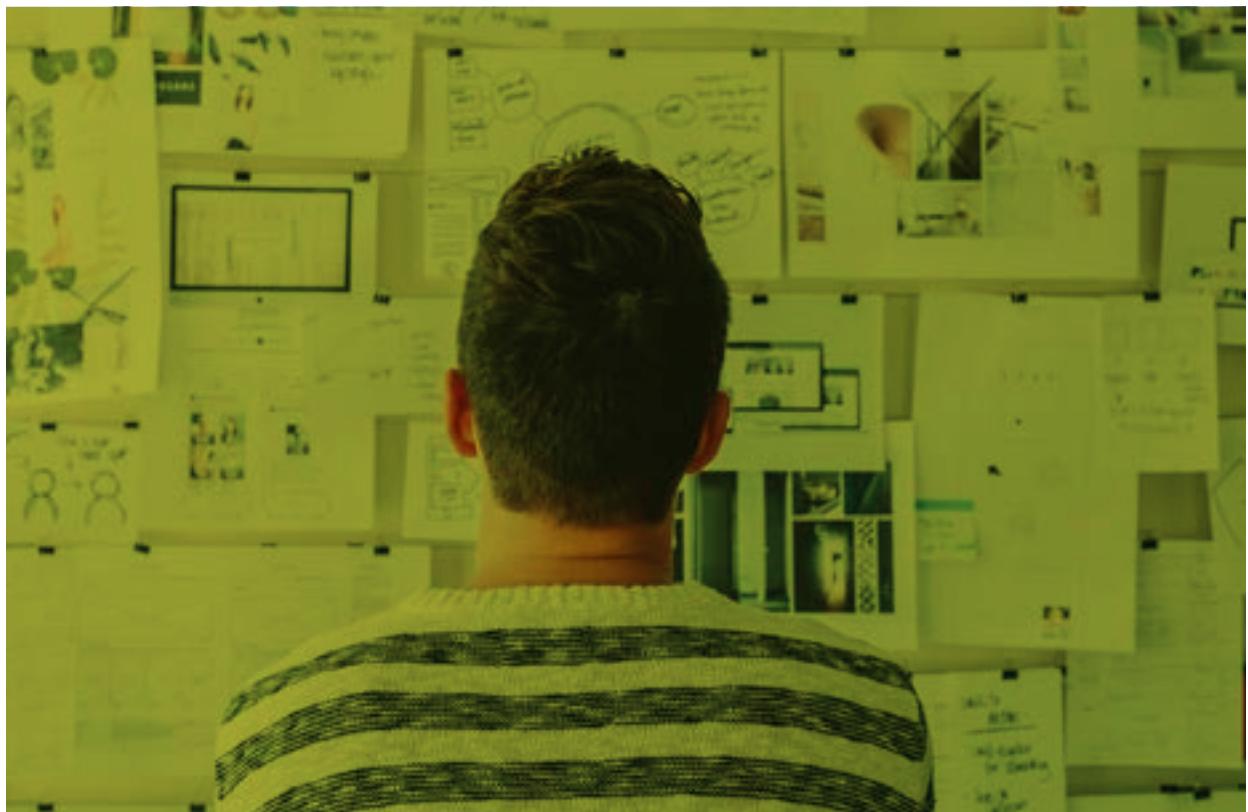
通过对业务问题的子域划分，我们找到了一些关键的业务对象。在开始进行抽象前一个必须的步骤就是“讲故事”！

讲什么故事呢？关于这个子域解决的业务问题或者提供的业务能力的故事。既然是故事，就必须有清晰的业务场景和业务对象之间的交互。这件事情看起来是如此自然和简单，然则一个团队里能够站起来有条不紊陈述清楚的却没有几人。读到这里的读者不妨停下来试试，你是否能够把现在你所做的业务在两三分钟内场景化地描述出来？

这么做显然目的是让我们能够比较完整地思考我们所要提炼和抽象的业务对象有哪些。只有当我们能够“讲”清楚业务场景的时候，才应该开始抽象的步骤。对于一个业务对象，我们常见的抽象可以是“实体”（Entity）和“值对象”（Value Object）。

这两个抽象方式在定义上的区别是，实体需要给予一个唯一标识，而值对象不需要（可以通过属性集合标识）。当然另外一个经常引用的区别是，实体应该是有一个连续的生命周期的，比如我们在一个订单跟踪领域里抽象订单为一个实体，那么每个订单应该有一个唯一识别号，订单也应该有从下单创建到最后交货完成的生命周期。

显然，如果不增加其它约束条件，值对象的抽象是没有意义的，都用实体不就行了？但如果我们稍微思考一下一个实体的管理成本，比如需要保证生命周期中实体状态的一致性，那么我们就会发现值对象变得很简单很可爱。当一个对象在我们（抽象）的世界里不能改变的时候，一切都变得简单了，这个对象被创建后只能被引用，当没有引用时我们可以把它交给垃圾回收自动处理。



随着高并发、分布式系统的普及，实际上我们在对业务对象抽象的第一步思考是能否用值对象。如果大家实现的技术架构采用函数范式的语言（类似Closure），那么首先考虑值对象抽象可能就是一个建模原则了。

对象抽象初步完成后，一定要再重复一次之前的故事来审视一下我们的建模。经历这个抽象过程后，参与讨论的每个人都应该发现自己更清晰业务的需求和需要提供的能力了。

## 聚合的封装

DDD元模型中一个核心概念叫“聚合”（Aggregate）。这个从建筑学来的名词非常形象，建筑学上我们翻译为“骨料”，是形成混凝土的重要元素，也是为什么混凝土如此坚固的基础。

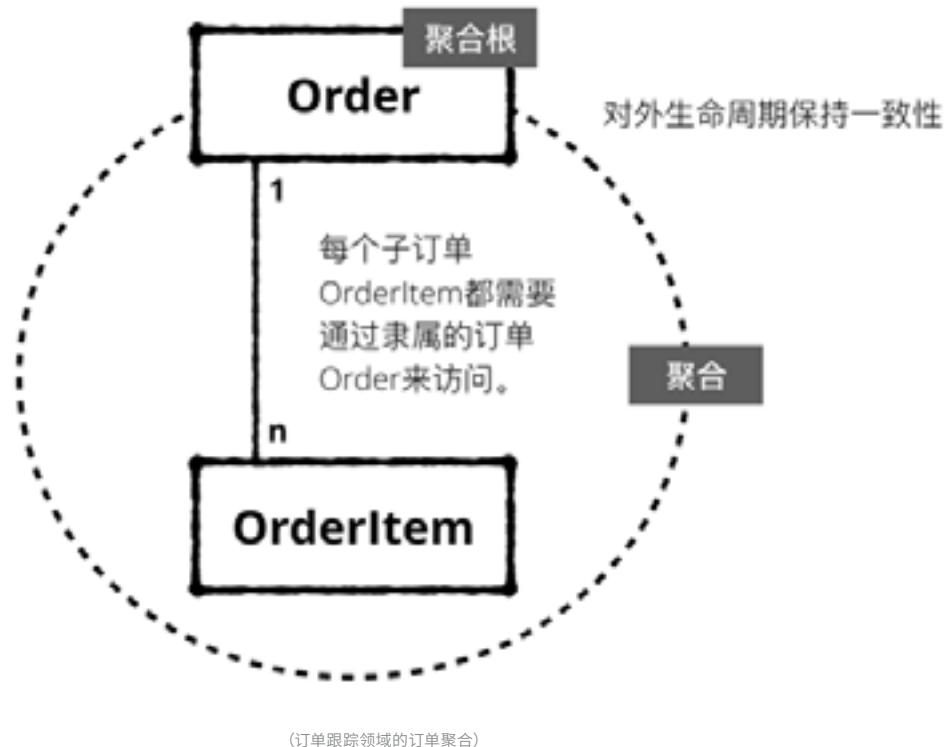


（混凝土里的一种骨料）

同理，在DDD建模中，聚合也是我们构建领域模型的基础，并且每个聚合都是内聚性很高的组合。聚合本身完成了我们对骨干业务规则的封装，减小了我们实现过程中出错的可能。

以上面那个订单跟踪领域为例，假设我们允许一个订单下存在多个子订单，而每个子订单也是可以独立配送的，这种情况下我们抽象出“子订单”这个实体。显然订单和子订单存在业务逻辑上的一致性，没有订单的时候不应该创建子订单，更新子订单的时候应该同时“通知”所属的订单。这个时候如果采用把订单和子订单聚合起来的封装就很有必要了。

采用聚合抽象的结果就是访问每个子订单都需要从相关的订单入口 (i.e., 订单为聚合根) , 存取时我们都是以这个聚合为基本单位, 即包含了订单和订单下面的所有子订单。显然这样的好处是在订单跟踪这个领域模型里, 订单作为一个聚合存在, 我们只需要一次性梳理清楚订单和子订单的逻辑关系, 就不需要在未来每次引用时都考虑这里面的业务规则了。



在建模过程中, 很多团队并没有努力思考聚合的存在。封装这个在技术实现领域的基本原则在建模时却很少被重视起来。开篇提到在战术建模过程中强调业务领域人员的参与也是为了解决这个问题, 聚合的识别实际是针对业务规则的封装, 当我们不理解业务规则的时候是无法做出是否封装的判断的。

一言以蔽之, 识别聚合是认知潜在核心业务规则的过程, 而定义出来的聚合是在大家共识基础上对核心业务规则的封装。

## 领域服务的定义

在最初的元模型定义里, 领域服务让不少人纠结, 一个经典的例子是在账户管理领域里对“转账”这个业务行为的抽象。由于转账本身是作用在至少两个账户上的, 所以把转账作为一个账户的行为显然是不合适的。那么如果我们把转账名词化抽象成一个实体呢? 感觉也是比较别扭, 毕竟转账是依附于账户存在的。

这个时候DDD在元模型里提出了服务 (Service) 这个抽象, 转账被抽象为一个服务感觉就顺畅多了。同样道理, 在

我们上面的订单跟踪领域里，如果跟踪的过程中需要进行短信的通知，一个比较好的建模就是抽象出一个“通知”服务来完成。

我经常会用静态方法来帮助技术人员理解服务的抽象（虽然服务并不一定用静态方法来实现）。服务本身就像一个静态方法一样，拥有一定的逻辑但不持有任何的信息，从整个领域来看也不存在不同“版本”的同一个服务。



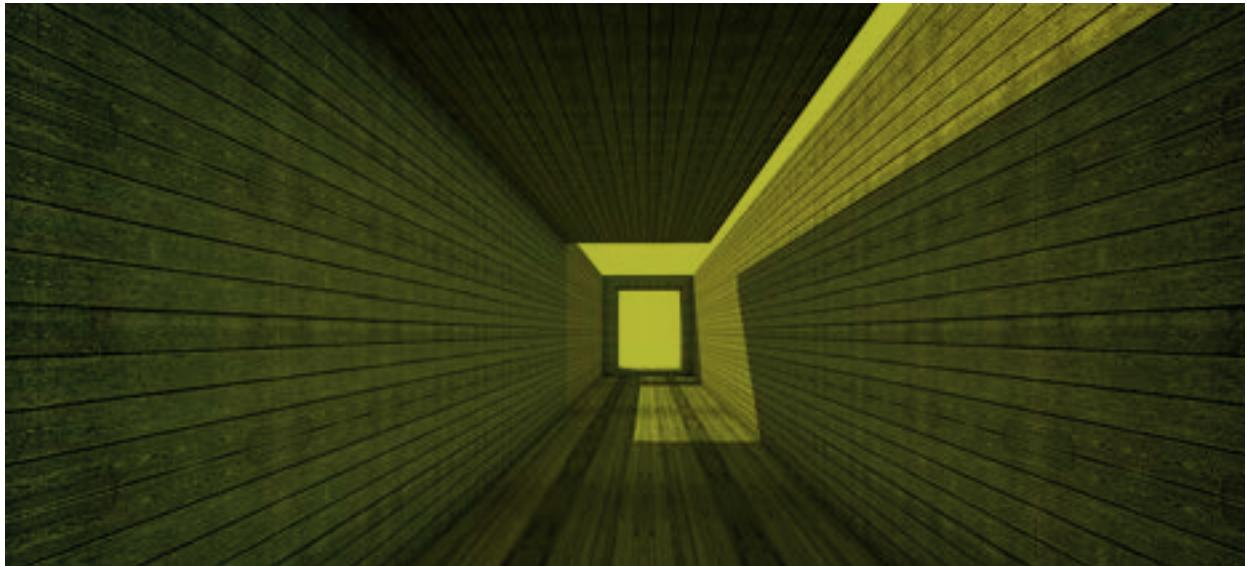
一个经常困扰大家的问题是对Service这个词语的限定，有的分层架构设计里会出现领域服务（Domain Service）和应用服务（Applicaiton Service）。大多数时候应用服务在领域服务的上层，直接对外部提供接口。如果存在这样的分层，那么领域服务就不应该直接对外，而应该通过应用服务。

举个例子，前面的订单消息通知如果是一个领域服务，在完成订单状态变化时创建通知消息，而最后的通知以短信的方式发给设定的人群，这样就应该有一个相应的应用服务，包含了具体的业务场景处理逻辑。之后也可能有一个邮件通知的应用服务，同样调用了这个通知领域服务，但通过邮件渠道来完成最终的业务场景。

由于微服务架构的流行，每个子领域的粒度已经相当细了，很多时候已经没有这样的领域服务和应用服务的区分了。当然从简单性角度出发这是好事情。在整个建模过程中，服务的抽象往往是最不确定的，也是最值得大家反复斟酌的地方。

# Repositories的使用

Repositories是一个非常容易被误解的抽象，很多人会直接联想到具体的数据存储。在初期采用DDD建模的时候，我经常刻意回避这个抽象，避免让大家陷入思考紊乱。



这个抽象概念实际可以追溯到Martin Fowler的Object Query模式。另外一个相关概念是DAO (Data Access Object)，都是用来简化需要存储的数据和对应的业务对象之间的映射关系。不同的是Repositories针对更加粗颗粒度的抽象，在DDD这个方法里我们可以认为映射对象是我们的聚合。针对每个实体在实现时候也可能创造出对应的DAO (比如采用Hibernate这样的ORM框架)，但显然在建模过程中不是我们需要关注的。

那么Repositories的抽象为什么是必要的呢？让我们再回到订单跟踪这个例子，通知订单状态发生变化的服务在发出通知前，需要定位到订单的信息（可能包括订单的相关干系人和子订单的信息）。通知作为一个服务是不应该持有具体订单信息的，这个时候我们就需要通过Repositories的抽象来建立对订单这个聚

合的查询，即有一个订单的repo，而具体的查询逻辑应该在这个repo中。

这样的抽象在需要存储和查询值对象的时候也是必要的。假设我们分析订单查询这个领域，在这个领域里订单记录显然已经不允许修改了，自然的抽象方式就是值对象。同时一个查询的服务来持有具体的查询逻辑（比如按时间或用户）是合理的。外部应用直接调取了查询服务（接口）并给出规定的参数，我们就需要一个订单记录的repo来持有跟存储相关的查询逻辑。当然这并不是说有一个查询就一定有一个repo与之对应，如果查询的逻辑非常简单，未尝不可以让服务直接针对数据存储实现。记住我们抽象的目标是让建模更简单，抽象过程中应该保持灵活。

## 限界上下文的意义

经过最近10多年的演进，我们在如何支撑一个组织的规模化上达成了一些基本的共识。我们知道微服务架构 (Microservices) 能够帮助我们把成百上千的工程师们组织起来，而小团队的自组织性是至关重要的。我们也逐步就如何能够在技术和业务团队之间明确沟通“架构”这个难题上找到了DDD。那么DDD和微服务架构的关系是什么呢？很多人会提到限界上下文 (Bounded Context)。

我曾经就这个话题专门撰文一篇 (DDD&Microservices)。一个限界上下文封装了一个相对独立子领域的领域模型和服务。限界上下文地图描述了各个子领域之间的集成调用关系。这个定义某种意义上和我们的微服务划分不谋而合：以提供业务能力为导向的、自治的、独立部署单元。所以虽然我们不能百分百依据限界上下文划分服务，但限界上下文，或者说是DDD，绝对是我们设计微服务架构的重要方法之一。

如果我们再追溯到DDD的战略设计，我们会发现在问题域上，DDD通过子问题域 (subdomain) 的划分就已经进行了针对业务能力的分解，而限界上下文在解决方案域中完成了进一步分解。当然我们不能完全认为子问题域和限界上下文有严格意义上的一对一关系，但大多数情况下一个子问题域是会被设计成一个或多个限界上下文的。子域 subdomain 和限界上下文某种意义上是互相印证的，重点在区分问题域和解决方案域，这是落地DDD最困难的地方，也是判断一个架构师能力进阶的分水岭。

## 战术建模小结

DDD的建模元素比较简洁，本文中叙述的元模型应该是满足了大多数场景下的建模。毛主席曾经有一句名言“战略上要藐视敌人 战术上要重视敌人”，就架构设计来说我们没有敌人，业务需求是我们的朋友。所以在领域驱动的架构设计方面，咱们需要的是“战略上要重视朋友，战术上要简化建模”。希望这句话能够帮助正在实践DDD的团队重新思考自己在战略问题域的投入和重视程度，不要挥舞着战术模型的大锤到处寻找实际不存在的钉子。

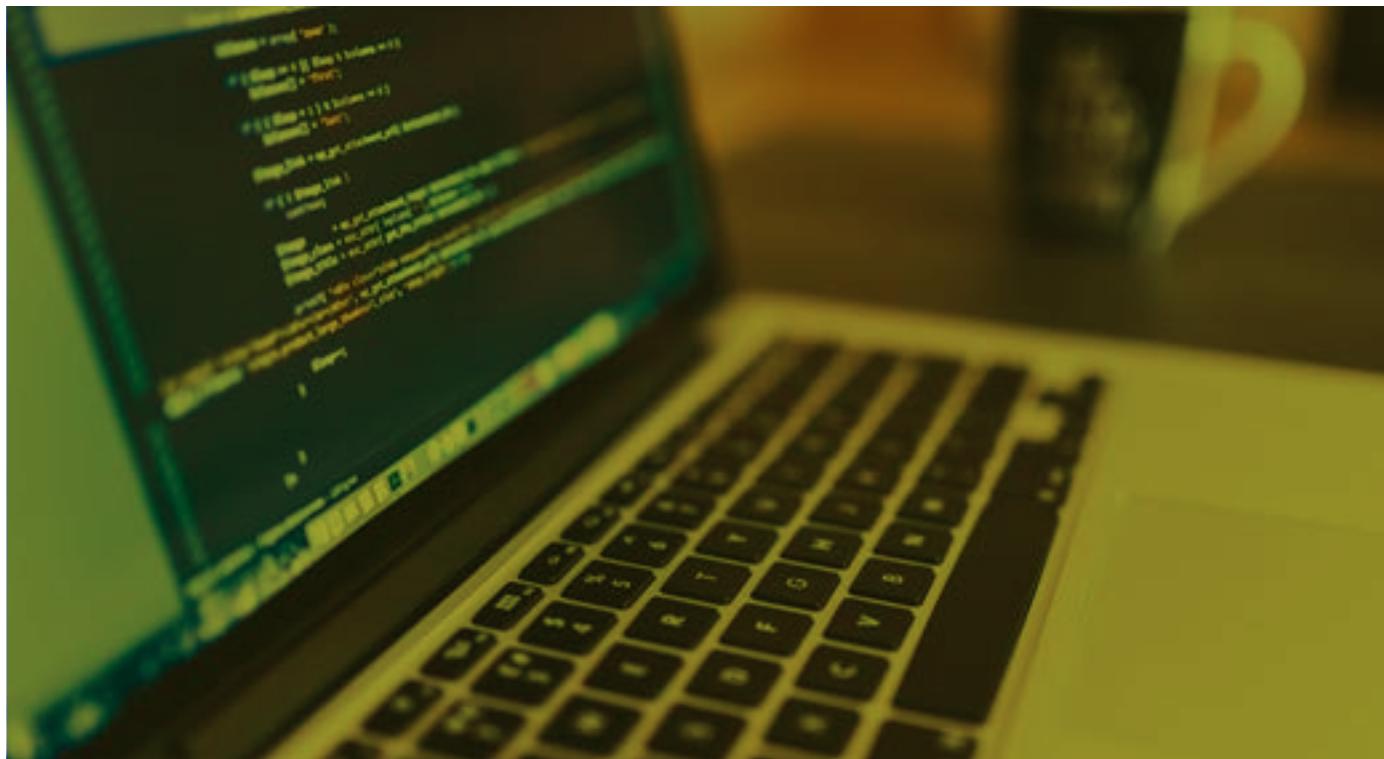
在这里我们也希望通过第一届DDD China建立起一个架构设计人员的交流平台。期待更多的中国技术人员能够通过这个平台和世界一流架构大师们建立起沟通的渠道，不仅在战略层面，也在战术层面和所有人一起分享讨论关于DDD的一切。

# DDD实战篇：分层架构的代码结构

---

作者：肖然

不同于其它的架构方法，领域驱动设计DDD (Domain Driven Design) 提出了从业务设计到代码实现一致性的要求，不再对分析模型和实现模型进行区分。也就是说从代码的结构中我们可以直接理解业务的设计，命名得当的话，非程序员也可以“读”代码。

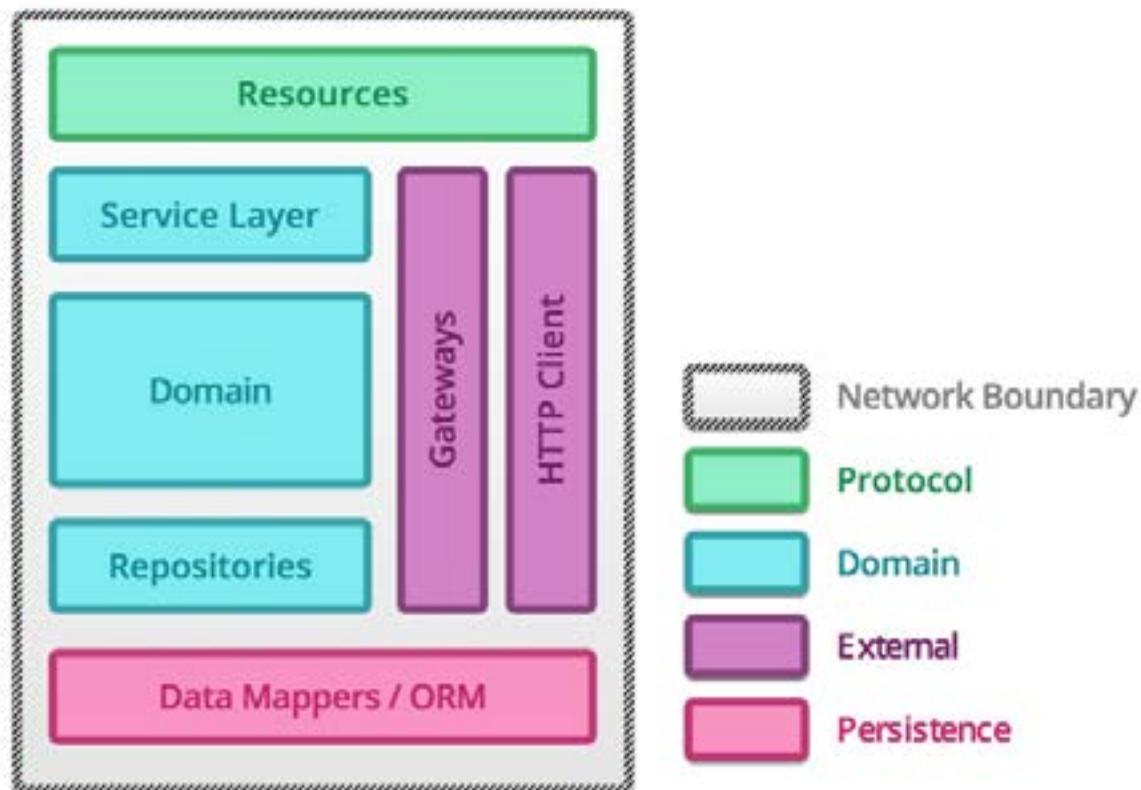


然而在整个DDD的建模过程中，我们更多关注的是核心领域模型的建立，我们认为完成业务的需求就是在领域模型上的一系列操作（应用）。这些操作包括了对核心实体状态的改变，领域事件的存储，领域服务的调用等。在良好的领域模型之上，实现这些应用应该是轻松而愉快的。

笔者经历过很多次DDD的建模工作坊，在经历了数天一轮又一轮激烈讨论和不厌其烦的审视之后，大家欣慰地看着白板上各种颜色纸贴所展示出来的领域模型，成就感写满大家的脸庞。就在这个大功告成的时刻，往往会有问：这个模型我们怎么落地呢？然后大家脸上的愉悦消失了，换上了对细节就是魔鬼的焦虑。但这是我们不可避免的实现细节，DDD的原始方法论中虽然给出了“分层架构”（Layered Architecture）的元模型，但如何分层却没有明确定义。

## 分层架构

在DDD方法提出后的数年里，分层架构的具体实现也经历了几代演进，直到Martin Fowler提炼出下图的分层实现架构后，才逐步为大家所认可。DDD的方法也得到了有效的补充，模型落地的问题也变得更容易，核心领域模型的范围也做出了比较明确的定义：包括了Domain, Service Layer和Repositories。



（Martin Fowler总结提出的分层架构实现，注意“Resources”是基于RESTful架构的抽象，我们也可以理解为更通用的针对外界的接口Interface。而HTTP Client主要是针对互联网的通信协议，Gateways实际才是交换过程中组装信息的逻辑所在。）

我们的核心实体 (Entity) 和值对象 (Value Object) 应该在Domain层，定义的领域服务 (Domain Service) 在

Service Layer，而针对实体和值对象的存储和查询逻辑都应该在Repositories层。值得注意的是，不要把Entity的属性和行为分离到Domain和Service两层中去实现，即所谓的贫血模型，事实证明这样的实现方式会造成很大的维护问题。DDD战术建模中的元模型定义不应该在实现过程中被改变，作为元模型中元素之一的实体本身就应该包含针对自身的行为定义。

基于这个模型，下面我们来谈谈更具体的代码结构。对于这个分层架构还有疑惑的读者可以精读一下Martin的原文。有意思的一点是，这个模型的叙述实际是在微服务架构的测试文章中，其中深意值得大家体会。

这里需要明确的是，我们谈论代码结构的时候，针对的是一个经过DDD建模后的子问题域（参见战略设计篇），这是我们明确的组件化边界。是否进一步组件化，比如按照限界上下文（Bounded Context）模块化，或采用微服务架构服务化，核心实体都是进一步可能采用的组件化方法。从抽象层面讲，老马提炼的分层架构适用于面向业务的服务化架构，所以如果要进一步组件化也是可以按照这个代码结构来完成的。

总体的代码目录结构如下：

```
- DDD-Sample/src/  
  domain  
  gateways  
  interface  
  repositories  
  services
```

这个目录结构一一对应了前文的分层架构图。完整的案例代码请从GitHub下载。

可以看到实际上我们并没有建立外部存储（Data Mappers/ORM）和对外通信（HTTP Client）的目录。从领域模型和应用的角度，这两者都是我们不必关心的，能够验证整个领域模型的输入和输出就足够了。至于什么样的外部存储和外部通信机制是可以被“注入”的。这样的隔离是实现可独立部署服务的基础，也是我们能够测试领域模型实现的要求。



# 模型表达

根据分层架构确立了代码结构后，我们需要首先定义清楚我们的模型。如前面讲到的，这里主要涉及的是从战术建模过程中得到的核心实体和服务的定义。

我们利用C++头文件 (.h文件) 来展示一个Domain模型的定义，案例灵感来源于DDD原著里的集装箱货运例子。

```
namespace domain{
    struct Entity
    {
        int getId();
    protected:
        int id;
    };
    struct AggregateRoot: Entity
    {};
    struct ValueObject
    {};
    struct Provider
    {};
    struct Delivery: ValueObject
    {
        Delivery(int);
        int AfterDays;
    };

    struct Cargo: AggregateRoot
    {
        Cargo(Delivery*, int);
        ~Cargo();
        void Delay(int);
    private:
        Delivery* delivery;
    };
}
```

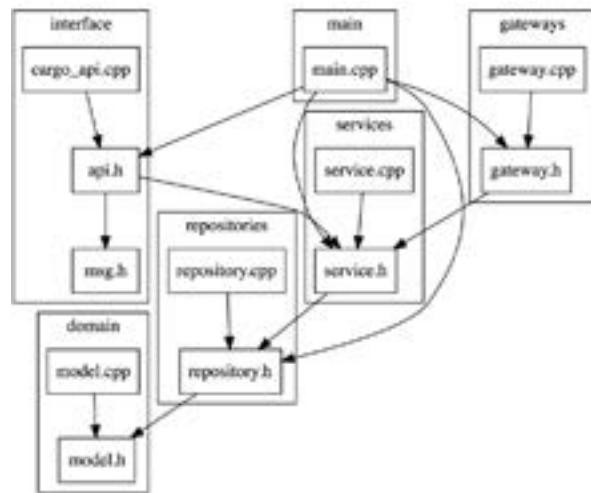
这个实现首先申明了元模型实体 Entity 和值对象 ValueObject。实体一定会有一个标识id。在实体的基础上声明了DDD中的重要元素聚合根AggregateRoot。根据定义，聚合根本身就应该是一个实体，所以 AggregateRoot 继承了 Entity。

这个案例中我们定义了一个实体Cargo，同时也一个聚合根。Delivery是一个值对象。虽然这里为了实现效率采用的是struct，在C++里可以理解为定义一个 class类。

## 依赖关系

代码目录结构并不能表达分层体系中各层的依赖关系，比如Domain层是不应该依赖于其它任何一层的。维护各层的依赖关系是至关重要的，很多团队在实施的过程中都没有能够建立起这样的工程纪律，最后造成代码结构的混乱，领域模型也被打破。

根据分层架构的规则，我们可以看到示例中的代码结构如下图。



Domain是不依赖于任何的其它对象的。Repositories是依赖于Domain的，实现如下：引用了model.h。

```
#include "model.h"
#include

using namespace domain;

namespace repositories {
    struct Repository
    {
    };
    ...
}
```

Services是依赖于Domain和Repositories的，实现如下：引用了model.h和repository.h

```
#include "model.h"
#include "repository.h"

using namespace domain;
using namespace repositories;

namespace services {
    struct CargoProvider : Provider {
        virtual void Confirm(Cargo* cargo){};
    };

    struct CargoService {
        ...
    };
    ...
}
```

```
auto provider = std::make_shared<
    StubCargoProvider>();

api::Api* createApi() {
    ContainerBuilder builder;
    builder.registerType< CargoRepository >().singleInstance();
    builder.registerInstance(provider).as();
    builder.registerType< CargoService >().singleInstance();
    builder.registerType().singleInstance();
```

```
auto container = builder.build();

std::shared_ptr api = container->resolve();

return api.get();
}
```

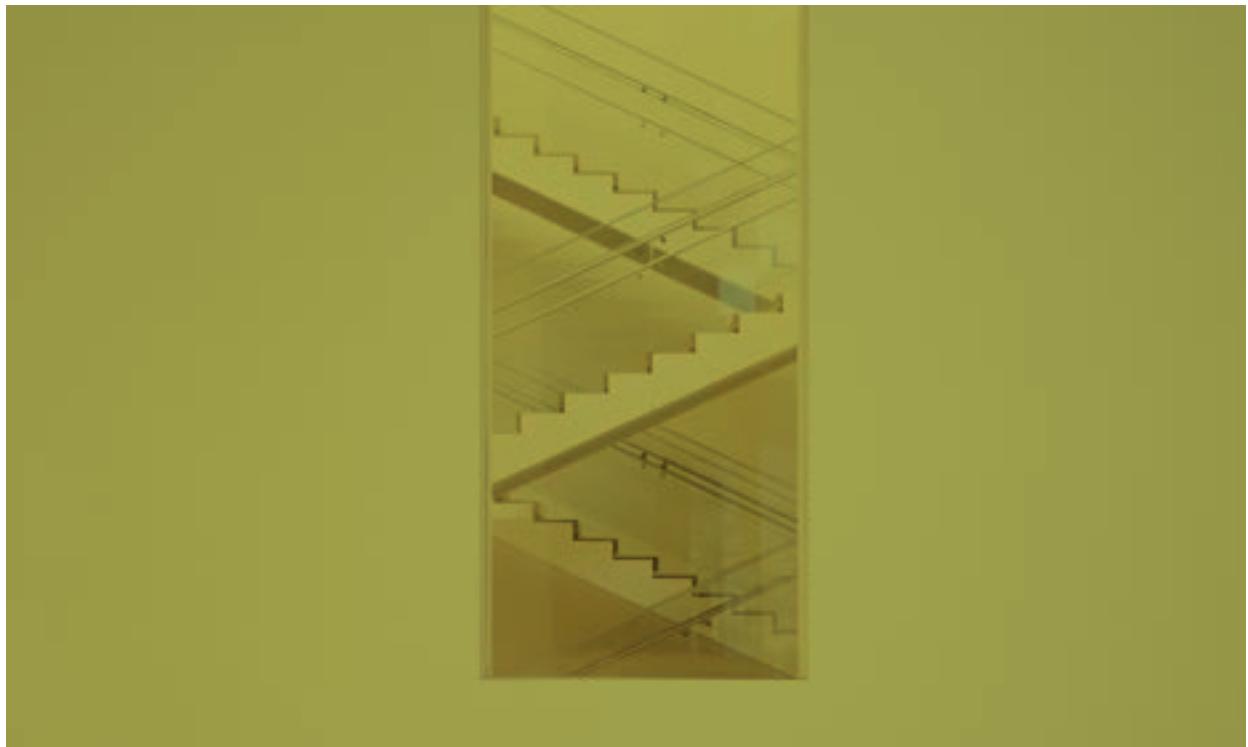
为了维护合理的依赖关系，依赖注入 (Dependency Injection) 是需要经常采用的实现模式，它作为解耦合的一种方法相信大家都不会陌生，具体定义参见[这里](#)。

在测试构建时，我们利用了一个IoC框架（依赖注入的实现）来构造了一个Api，并且把相关的依赖（如CargoService）注入给了这个Api。这样既没有破坏Interface和Service的单向依赖关系，又解决了测试过程中Api的实例化要求。

## 测试实现

有了领域模型，大家自然会想着如何去实现业务应用了，而实现应用的过程中一定会考虑到单元测试的设计。在构建高质量软件过程中，单元测试已经成为了标准规范，但高质量的单元测试却是困扰很多团队的普遍问题。很多时候设计测试比实现应用本身更加困难。

这里很难有一个固定标准来评判某个时间点的单元测试质量，但一个核心的原则是让用例尽量测试业务需求而不是实现方式本身。满足业务需求是我们的目标，实现方式可能有多种，我们不希望需要持续重构的实现代码影响到我们的测试用例。比如针对实现过程中的某个函数进行入参和出参的单元测试，当这个函数发生一点改变（即使 is 重命名），我们也需要改动测试。



测试驱动开发TDD无疑是一种好的实践，如果应用得当，它确实能够实现我们上述的原则，并且能够帮助我们交流业务的需求。比较有意思的是，在基于DDD建立的核心模型之上应用TDD似乎更加顺理成章。类比DDD和TDD虽然是不恰当的，但我们会发现两者在遵循的原则上是一致的，即都是面向业务做分解和设计：DDD就整个业务问题域进行了分解，形成子问题域；TDD就业务需求在实现时进行任务分解，从简单场景到复杂场景逐步通过测试驱动出实现。下面的测试用例展现了在核心模型上的TDD过程。

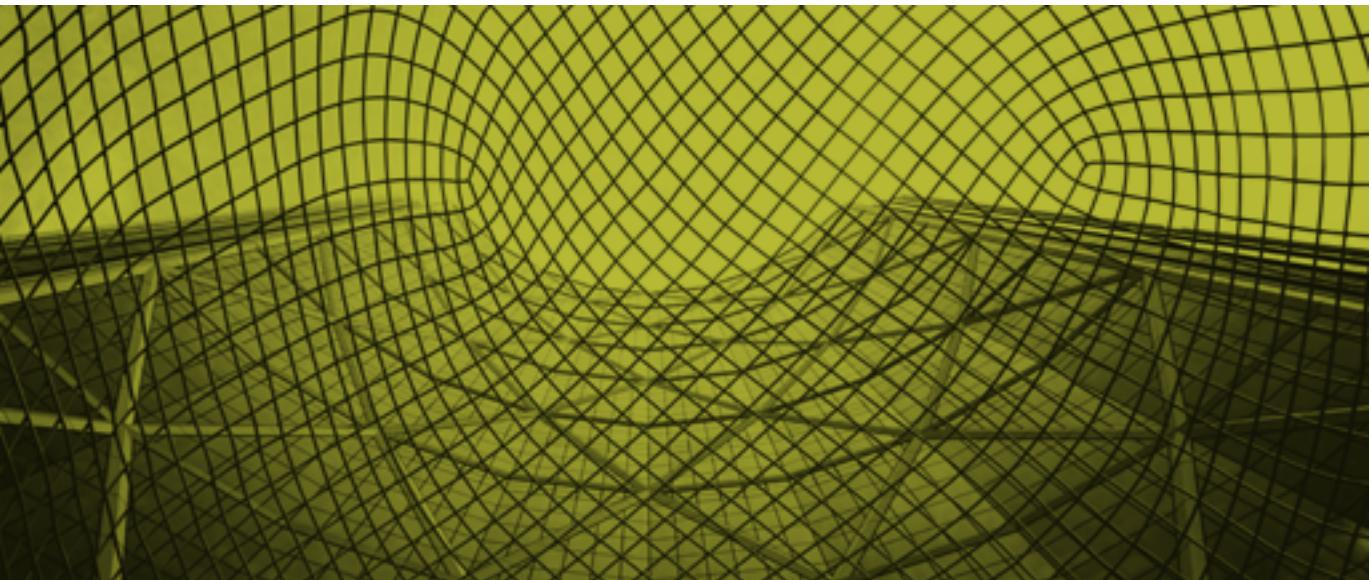
```
TEST(bc_demo_test, create_cargo)
{
    api::CreateCargoMsg* msg = new api::CreateCargoMsg();
    msg->Id = ID;
    msg->AfterDays = AFTER_DAYS;
    createCargo(msg);
    EXPECT_EQ(msg->Id, provider->cargo_id);
    EXPECT_EQ(msg->AfterDays, provider->after_days);
}
```

上面测试了收到一条创建信息后实例化一个Cargo的简单场景，要求创建后的Cargo的标识id跟信息里的一致，并且出货的日期一致。这个测试驱动出来一个Interface的Api::CreateCargo。

下面是另外一个测试推迟delay的场景，同样我们看到了驱动出的Api::Delay的实现。

```
TEST(bc_demo_test, delay_cargo)
{
    api::Api* api = createApi();
    api::CreateCargoMsg* msg = new api::CreateCargoMsg();
    msg->Id = ID;
    msg->AfterDays = AFTER_DAYS;
    api->CreateCargo(msg);
    api->Delay(ID,2);
    EXPECT_EQ(ID, provider->cargo_id);
    EXPECT_EQ(12, provider->after_days);
}
```

长期以来对于TDD这个实践大家都有关于架构设计上的疑惑，很多资深架构师担心完全从业务需求驱动出实现没法形成有效的技术架构，而且每次实现的重构成本都可能很高。DDD的引入从某种程度上解决了这个顾虑，通过前期的战略和战术建模确定了核心领域架构，这个架构是通过预先综合讨论决策的，考虑了更广泛的业务问题，较之TDD应用的业务需求层面更加宏观。在已有核心模型基础上我们也会发现测试用例的设计更容易从应用视角出发，从而降低了测试设计的难度。



## 关于预先设计

如果没有读战略篇直接看本文的读者肯定会提出关于预先设计的顾虑，毕竟DDD是被敏捷开发圈子认可的一种架构方式，其目标应该是构建架构模型的响应力。而这里给大家的更多的是模式化的实现过程，好似从建模到代码一切都预先设计好了。

值得强调的是，我们仍然反对前期设计的大而全 (Big-Design-Up-Front, BDUF)。但我们应该认可前期对核心领域模型的分析和设计，这样能够帮助我们更快地响应后续的业务变化 (即在核心模型之上的应用)。这不代表着核心领域模型未来会一成不变，或者不能改变，而是经过统一建模的核心部分变化频率较之外部应用会低很多。如果核心领域模型也变化剧烈，那么我们可能就要考虑是否业务发生了根本性的变化，需要建立新的模型。

另外不能忘记我们预先定义的模型也是被局限在一个分解出来的核心问题域里的，也就是说我们并不希望一口气把整个复杂的业务领域里的所有模型都建立起来。这种范围的局限某种程度上也限制了我们预先设计的范围，促使我们更多用迭代的方式来对待建模工作本身。

最后显然我们应该有一个核心团队来守护核心领域模型，这不代表着任何模型的设计和改动都必须由这个团队的人做出（虽然有不少的团队确实是这样落地DDD的）。我们期望的是任何对核心模型的改动都能够通过这个核心团队来促进更大范围的交流和沟通。检验一个模型是否落地的唯一标准是应用这个模型的团队能否就模型本身达成共识。在这点上我们看到很多团队持续通过代码走查 (code review) 的方式在线上和线下实践基于核心模型的交流，从而起到了真正意义上的“守护”作用，让模型本身成为团队的共同责任。

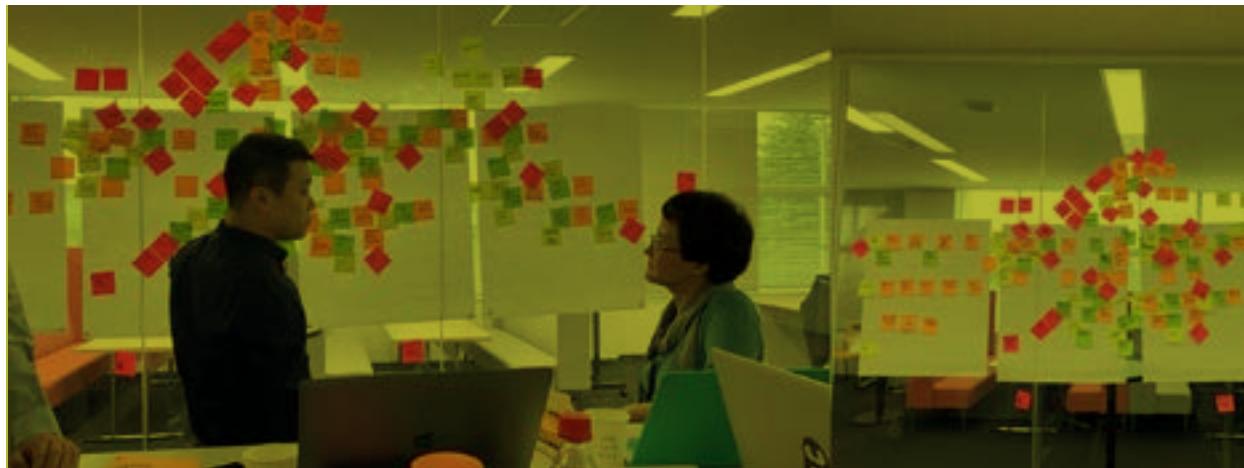
实践DDD时仍然需要遵循“模型是用来交流的”的这一核心原则。我们希望本文介绍的方法及模式能够帮助大家更容易地交流领域模型，也算是对DDD战略和战术设计的一点补充。

# DDD的终极大招——By Experience

作者：肖然

以DDD思想和微服务架构为代表的新的架构时代正在逐步形成，不同方法和工具的涌现让人激动不已，同时这个过程也让人感觉到些许的不安，因为没有一套方法和一套架构能够打遍天下，我们能明确告诉所有组织和团队的，也只是架构设计上应该“响应变化胜过遵循计划”！具体到采用哪一种架构设计思想和方法，仿佛都需要增加一个定语“这取决于……”。

以去年的“明星”方法Event Storming (ES) 为例，今年已经开始被不少人所批判。内行已经开始调侃这就是“糊墙”（不明就里的同学可以感受下图中的ES现场）。而实际上ES创始人Alberto是一位很低调的实践者，仍然在不停地磨练着他发明的这套方法。一年里我也接到了无数类似“我们是xxx领域，有xxx系统，ES感觉好像用不上？”的问题。我的答案往往是：“没事儿，你们先试试，找到具体困难点，咱们再看为啥不好用。”



(一个ES现场，“糊”满各色纸贴的建模过程。)

我相信得到这个答案的部分团队可能真的去尝试了ES，但鲜有人再将他们遇到的具体困难反馈给我 —— 也许ES实践本身就是困难，而不是他们要解决的业务问题。但我的出发点却并非推广ES，而是让团队能够获取“经验”！这点上还是小有成就的，去年我可能还是中国区“糊墙”最多的人，今年很多人都远胜过我了。

不管是在DDD原著，还是后续不少专家的书籍中，都明示或暗示架构设计最后的终极大招还是By Experience——靠经验吃饭。从战略角度的subdomain（子问题域的划分）到战术建模层面Entity、VO的选择，最终的决策很可能不是完全“理性”，经验这个“感性”的东西发挥着很大的作用。

对于一个顾问和教练来说这是绝望的答案，因为我们每次面对的是希望学习，但没有经验的团队，“靠经验吃饭”等于告诉团队这东西没套路、靠感悟。这就迫使我们转换视角，从教大家DDD方法，转换到帮助大家获取DDD经验。下面就让我们来看看怎么有效解决DDD经验获取这个问题。

## 问题、问题、问题

DDD作为一种架构方法，最大的突破应该说是非常明确地区分出了问题域和解决方案域。而认知问题这件事情绝对不是技术人员擅长的，从我们学习编程起，我们就被如设计模式（Design Pattern）这样的解决方案所包围。想当年我自己最得意的事情也是refactor to pattern，也是把解决方案当成了“终极问题”来追求。

这往往是一个痛苦的蜕变，需要有人在你身边不停念叨“你说的问题是什么？”你必须要做到心平气和，即使你认为对方是故意挑衅，有时候挑战更能促进思考上的突破。比如我经历过下面的一段经典对话：

甲：我认为这个子问题域是客户账户管理的问题。

乙：我觉得你已经在说解决方案了。

甲：客户账户管理是问题，我并没有提怎么管理啊！

乙：谁说一定要管理客户？！我还是觉得你说的是解决方案！

甲：（受不了你了……）不管理客户我们做这个系统干啥？

乙：我就是这个意思啊，为啥要做这个系统？我们解决了什么业务问题？

甲：这么说的话那把业务找过来，看他们怎么说。

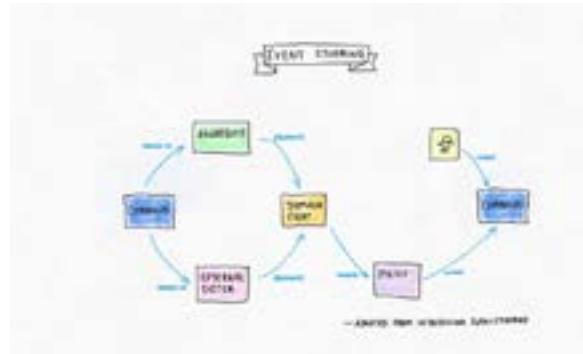
乙：行，反正DDD里说领域专家很重要，业务来了再讨论。

某种意义上这两位技术人员的争论是卓有成效的，最终的发现是业务问题其实并不清楚，远没有达到可以进入解决方案建模讨论的时候。

## 跨领域合作

当然上面的对话还有另外一个有意思的核心观点，即由于问题和解决方案在整个建模过程中是不停深入和迭代的，所以我们必须鼓励，甚至要求从业务到技术跨领域的人员参与和协作。

这点是我为什么仍然认为ES是一个好方法的基础，当然与我相对的观点是，如果有了真正的领域专家，搞那么复杂的协作有必要吗？ES通过对事件（event）的利用，提供了一套业务和技术能够共同理解的协作机制。在我的辅导过程中，很容易让两边的同学都理解如何上手。



(ES的运作机制，很有效的利用了Domain Event; 注意这里的event是业务事件，而非技术实现。我的同事伍斌在自己的简书中详细记录ES的采用过程，欢迎大家查阅。)

当然如果真有经验丰富的领域专家，确实事情就简单了很多。业务问题的分解首先就变得非常流畅，ES的功效也就不那么明显了。然而我个人始终认为“团队共同的学习 胜于 建模本身的正确性”，即使专家也不能完全预见未来，所以团队能够有机会通过某种手段学习专家的知识，也是很有价值的一件事情。

## 从需求到代码

DDD最初吸引我的地方是能够从问题分析一直拉通到代码实现，这有别于很多其它的架构方法，总是在某个链条上产生脱节。所以DDD的经验获取也需要尝试让团队端到端的拉通体验。

然而事实上很多团队仍然在践行着脱节的实践，比如建模后产生的Entity仍然用传统的数据和行为分离的实现方式。这样的实践方式显然是有悖于DDD的初衷，如果不能让业务和系统模型实现绑定关系，很快就会走上各说各话的老路上去。

实践端到端也有一定的技巧，首先应该明确分层架构的原则和规范，比如是否有Application Service存在的必要，Interface的调用规则等等。在此基础上，需要明确守护Domain Model的纪律，时刻保证代码和建模的一致性。最后需要建立分层的测试机制，特别是对Domain层逻辑的守护。

和前两点相比，这真是一个需要全队刻意练习的过程，坚持信念是团队走过开始阵痛期的必要条件。

## 刻意“失败”

之前在辅导团队的时候，一个常见问题就是团队纠结于一个业务概念建模采用Entity，还是VO。经常会听到团队说：“从现在的角度来看，VO应该是完全够用了，但很显然接下来我们马上就需要有业务状态的变

化，很可能VO就没法玩了。”

针对这样的问题，我往往会刻意引导团队从简单的VO建模入手，先不要考虑“未来”的需求，即使有时候这些需求已经相当明确。这样的刻意行为显然会造成团队在接下来的时间里改变模型，VO可能会被重新建模成Entity。短时间有可能是痛苦的，很多技术人员也会跳起来说，你这是“站着说话不腰疼”。

但DDD的核心就在于持续的演进，演进就意味着模型和实现的改变。这样的改变和上面我们刻意安排的“失败”其实是一致的。当我们通过这样的刻意练习获取了演进的经验后，业务和架构未来的变化对我们来说就真的可以by experience了。

## 写在最后

开篇我就提到了一个新的架构时代正在浮现，不同于之前的架构方法，没有一个组织和企业会在这个时代告诉你这就是做架构的正确方式。数字化时代的系统和应用在不停进化着，速度越来越快，想要找到进化过程中正确的元方法是非常困难的。

DDD的终极大招By Experience某种意义上是在持续探索，并要求大家接受在这个探索过程中的不确定性 —— 你的设计有可能在未来被证明是错误的。这可能是未来架构设计最大的挑战，我们必须能够让架构持续演进。

《演进式架构》已于今年问世，带给我们很多这些方面的思考，类比人类社会的演进，数字化世界的构建和发展应该有很多地方可以借鉴和学习。当然就这个问题而言，不管是DDD，还是Microservices，都只是我们探索架构演进的开始，我们还有很多的Experience需要获取！

ThoughtWorks洞见  
领域驱动设计

---

# 通用语言、领域、 界限上下文

# 重读领域驱动设计 —— 如何说好一门通用语言

作者：王岩

结论先行：

在 DDD 中，通用语言是以限界上下文为边界的。如果一个产品或者项目有多个限界上下文，我们就需要为每个限界上下文定义通用语言。

限界上下文提供了一个语义边界，来保持通用语言和领域概念的一一对应关系。

这个约束解决了现实世界中同样的名词在不同场景、时机下对应不同的业务概念所带来的歧义问题，帮助团队在使用通用语言交流的时候可以无歧义沟通。

## 初尝“通用语言”

最初我对于如何构建通用语言的认识，来自于《领域驱动设计》第一章中的案例。这个案例生动的展示了开发人员如何在和领域专家的沟通过程中，建立了双方理解一致的通用语言，并且使用这个语言来进行双方的沟通。基于那个案例，我当时对构建通用语言的理解就是要：

- 技术人员使用业务人员的用语作为开发词汇；
- 划分好聚合，将这些词汇关联到聚合上；
- 技术人员要将这些词汇映射到代码实现中；
- 这些词汇会随着项目的发展一点点扩展；

带着这份理解，我在曾经负责过的小型项目上做了一些实践，效果都很不错。在很长一段时间，团队的开发人员体会到了在和业务人员交流时候心有灵犀、会心一笑的快感；也很少听到“这个东西不是我要的”这类批评了。

# “通用语言”遇到同名词汇时就变得不清不楚了

然而，当我来到ThoughtWorks参与到一些几十号人的项目时，我发现根据这个原则构建起来的通用语言，在遇到同名多义的词汇时，就无法保证团队内部的沟通是无歧义的。而这种歧义又会导致团队成员说着同样的话想着不同的事情的情况出现，例如：

- 同名的业务词汇与实际业务关系不清：“为什么不能给销售订单增加一个是否投诉的字段，界面上都是显示在销售订单上的”——销售订单到底是个什么东西，能干什么不能干什么是怎么确定的？
- 同名的业务词汇与不同的业务词汇关联：“我在销售订单付款后改变了买家信息，为什么我看销售订单的预定里的买家也发生了改变”——这里说的买家信息有几个？
- 同名的业务词汇之间的关系不清楚：“为什么我变更了profile 上的买家地址，销售订单上的买家地址就跟着改变了”——这里说订单上的买家地址和profile 上的买家地址是一个什么关系？

## 通过添加约束消除歧义

下图是 DDD 概念的一个元模型图。从图的左下角，我们可以看到在构建通用语言时，还有两个额外的约束条件：子域和限界上下文。



**在 DDD 中, 软件的核心是其为客户解决领域相关的问题的能力。**

这里的领域, 就是指软件系统要解决的实际问题相关的东西的集合。

例如: 为一个电子商务公司开发一个电商系统, 我们就需要围绕这个盈利模式的运营方式、业务规则, 比如如何进货, 如何促销, 如何物流等等了解这个电子商务公司的盈利模式, 所有和业务相关的东西都属于领域。

领域分为问题域和解决方案域两部分。

为了分解问题域的复杂度, 问题域又会被拆解为多个子域, 每个子域都要明确待解决的业务问题和业务流程, 以及通过解决业务问题为企业带来了什么样的业务价值 (这个是因, 业务流程和要解决的业务问题是果)。

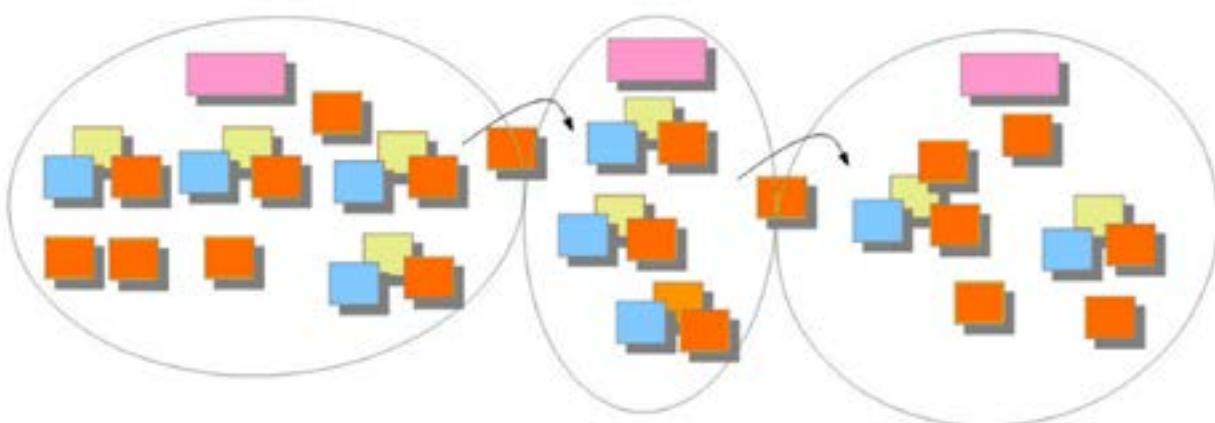
在清晰的定义子域后, 我们就可以建立通用语言来提取该子域的领域知识, 并基于通用语言为解决问题建立领域模型。

一个领域模型会存在于一个限界上下文中。限界上下文在 DDD 中用来定义模型的适用范围、模型的用途、以及在何处保持一致, 限界上下文会让团队明确模型的职责边界是什么。同时, 通用语言被限定在限界上下文中; 限界上下文提供了一个语义边界, 在每个限界上下文内通用语言的每个词汇必须和领域概念一一对应。

理想条件下, 子域和限界上下文是一一对应。但是子域划分的粒度, 遗留系统的现状, 语言的歧义, 团队结构等子域和限界上下文对应可能是 $1:N$  或者  $N:N$  的。

通过限界上下文间的映射, 上下文中的多个模型会协作以满足系统需求。我们也可以了解在不同上下文中的同名词汇是否存在关系, 存在什么样的关系。

对通用语言而言, 子域解释了通用语言和现实世界业务活动的关系; 限界上下文提供了一个语义边界, 来保持通用语言和领域概念的一一对应关系; 上下文映射则提供了不同限界上下文中的通用语言的转换关系。



# 来解决下前文的问题

前文所述的订单及订单的相关概念存在着歧义，我们来看下通过子域、限界上下文和上下文映射是怎么消除这些歧义的：

## 因为同名的业务词汇与实际业务关系不清导致的疑惑

“为什么不能在销售订单中增加一个是否投诉的字段，界面上都是显示在销售订单上的”

假设，这里所说的销售订单存在于销售子域下，那么这个订单应该解决的是销售过程中的问题。订单的生命周期以销售开始到销售终止。一般而言投诉属于售后环节，在销售订单上声明是否投诉字段，意味着销售订单的职能突破了销售子域。UI 上的销售订单展示了聚合的信息，和同名的领域模型不一定保持一致。

## 因为同名的业务词汇与不同的业务词汇关联导致的疑惑

“我在订单付款后改变了买家信息，为什么我看订单的预定里的买家也发生了改变”

在订单上有两种买家信息，可以通过在不同的上下文中隔离来区别这两个拥有相同含义但却是不同词汇的词汇。在销售子域中建立两个上下文，分别为预定有界上下文和购买上下文，把订单领域模型拆分到这两个上下文中。在不同的上下文中，订单都有自己的买家信息，就解决了“在订单付款后改变了买家信息，为什么我看订单的预定里的买家也发生了改变”这个问题。

## 因为同名的业务词汇之间的关系不清楚导致的疑惑

“为什么我变更了profile 上的买家地址，订单上的买家地址就跟着改变了”

订单存在于购买上下文，profile 存在于身份信息上下文中，购买上下文和身份信息上下文存在映射关系，在订单创建时候从身份信息上下文复制买家地址，在订单中单独保存。这样就解决了“为什么我变更了profile 上的买家地址，订单上的买家地址就跟着改变了”的问题。

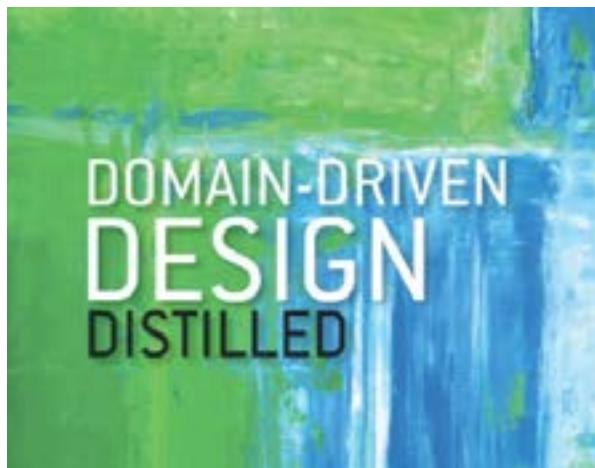
## 引用：

- 1.《领域驱动设计》
- 2.《实现领域驱动设计》
- 3.当Subdomain遇见Bounded Context
- 4.DDD的终极大招——By Experience
- 5.《领域驱动设计学习：领域、子域、限界上下文》

# 当Subdomain遇见Bounded Context

作者：肖然

《实现领域驱动设计》的作者Vernon根据过去几年DDD的实战经验又写了一本《领域驱动设计精粹》，日前已经在中国翻译出版。去年底出版社找到我时，读完英文原著最终还是放弃了翻译，推荐给了其他同事，并告诉他们出版后准备接受炮火洗礼。



不得不承认Vernon的新书在构建DDD落地体系方面较之上一本有了很大的进步，全书读起来很连贯，有一定实践基础的团队或个人均可直接上手书中很多的实践。并且通过一个案例完整叙述了从需求分析开始到最后的团队迭代开发。当然迭代运作过程中的工作量估计方式，在我看来过于简单粗暴，虽然强化了架构的最终代码落地，但却可能造成一系列的僵化。

本文主要针对Vernon一直以来对Subdomain和Bounded Context的一对一映射关系进行讨论。目标是让更多同学意识到这个方面的不同声音，从而能够加深对这两个概念存在意义的理解，并建立自己的判断。

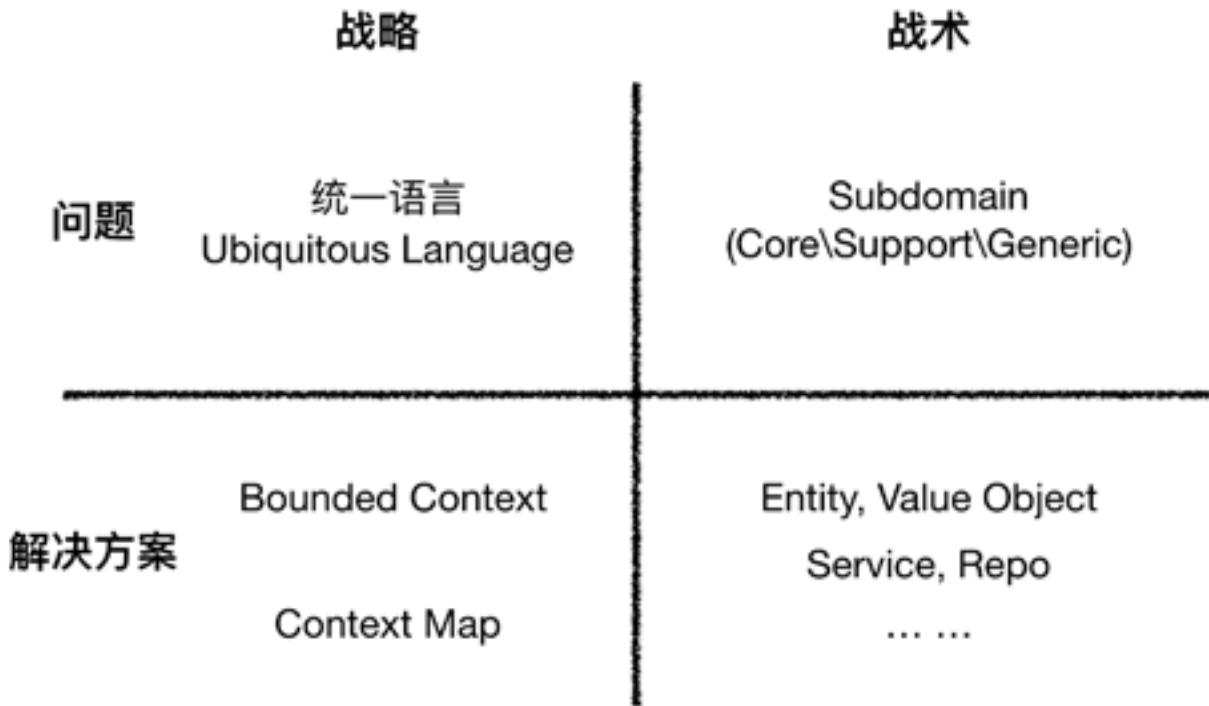
## 区分问题和解决方案是个老大难问题

问题和解决方案总是像一对难以分辨的孪生兄弟，一个人看到的哥哥可能就是另一个人认为的弟弟。好像程序员在开发Story时，Story成了我们要解决的问题，具体的代码实现成了解决方案；但当BA在分析同样一个Story时，问题就成了对应的业务需求，Story只是分析出的解决方案的描述。

当然这个区分有时候可能并没有那么重要，Story到底是一个问题，还是一个解决方案，其实我们在迭代过程中并不是很关心。但有时候不做问题和解决方案的区分确是十分危险的，甚至会造成整个产品的失败。这样的例子当然是一抓一大把的，比如我经常提及的为税务审计人员提供屏幕上多记录的翻页功能，就是我职业生涯中记忆最深刻的一次失误，想当然地采用了“通用”解决方案。

Eric Evan在构建DDD的体系时显然是思考了问题和解决方案这两个维度的，我相信这个过程也是十分痛苦的，以至于最后呈现在书里的实践并没有做非常明确地划分。对于后面的实践者，包括我们自己，都存在着不一样的解读。我们曾经讨论过一个DDD实践的象限划分，但由于这样的划分太过主观，结果是一组很长的邮件讨论。

象限如下图所示，这是一个如同“PHP是世界上最好语言”般的讨论，建议大家慎入，以免上火。



(从问题/解决方案和战略/战术维度分析DDD元模型的元素)

这样的象限分类确实有点简单粗暴，但Subdomain和Bounded Context却是Eric明确定义的两个核心模型概念。Subdomain是对问题域的分解，而Bounded Context是对解决方案域的分解。这两个核心概念构建起了DDD处理真实世界复杂度的根基。

建模过程中很多同学其实是忽略Subdomain的，反正目标是Bounded Context。当问题相对简单时，Subdomain的划分确实给人感觉是自寻烦恼，划出Bounded Context后反过来推Subdomain似乎更容易上手。读《领域驱动设计精粹》时你会发现相似的逻辑，配合书中敏捷项目管理工具的案例（问题也挺简单）还是挺好用的。

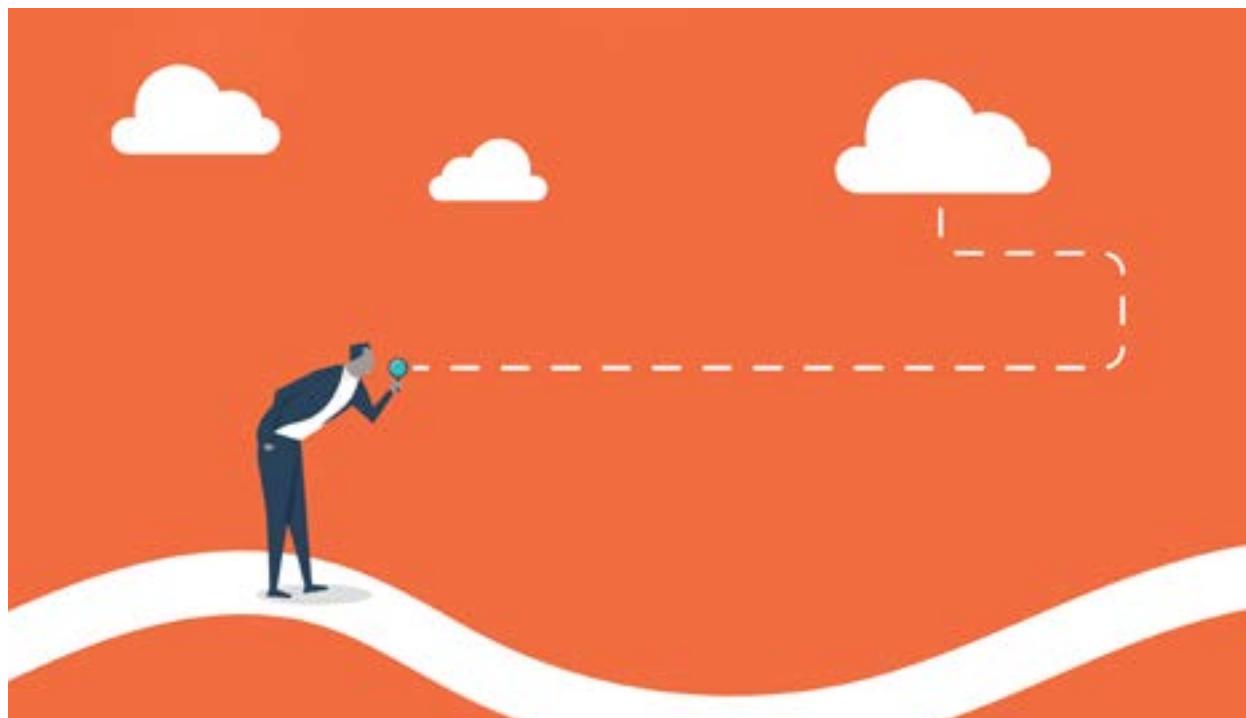
那么为什么我们还要关注Subdomain，还要去区分什么Core Domain、Support Domain和Generic Domain呢？是否和Story一样，留给业务和BA就好，程序员还是应该抓紧搞完Bounded Context，然后开写微服务比较务实呢？

## 区分Subdomain的必要性

在帮助一个长期合作伙伴构建大规模DDD应用时，我写了一个“xx阶xx步”的体系。也成了很多咱们同事体系性学习DDD的开始。

一年半以后这个团队组织了所有的技术专家和主管让我又讲了一次这个体系。这次我花了一天时间让大家体会问题和解决方案的区别，加入了Subdomain的概念。参加团建时，我问了几个专家和主管他们怎么看之前的设计，得到更多是务实的“赞赏”。其实我并不在意具体落地时的裁剪，但希望白纸黑字时应该明确原委，这也是我为什么拒绝了《领域驱动设计精要》翻译的原因。

我经常用电商的案例让大家快速认识到Subdomain划分的重要性。大浪淘沙之后我们发现淘宝和京东依然是霸主。当年马爸爸嘲笑强哥构建人肉物流网的寓言也并没有发生，反而很多人爱上了京东自有物流的速度。当然站在马爸爸当年对电商问题的认知角度，自建物流是可笑的，毕竟他要解决的核心问题是如何让琳琅满目的中小供应商能够直接对接千千万万的用户，让用户能够更容易的发现适合的商品。



所以从一开始淘宝和京东定义的Core Subdomain就是不一样的，正是问题认知的区别让两家都活了下来，并且活得很好。我们可以看到在线物品展示，吸引消费者方面淘宝一直在引领；而行业里如果你有机会接触电商领域，会发现京东物流系统还是蛮厉害的。

这是我们多年后的今天看到的结果呈现，但其实真正决定命运和格局的确是多年前两家电商对自身核心问题的理解。这个认知驱动出了两家完全不同的成功电商。

很多同学会说这玩意儿是商业模式，也轮不到我们搞研发的参与。我们拿到的都是既定问题了，再识别Subdomain也没啥意义了。这个论断有两方面问题：

- 作为产品和服务的实现者，如果不参与和关注问题本身的划分及核心子问题的认知，那么你很可能在浪费自己的时间，开发出未来被边缘化，甚至淘汰的系统。这不是危言耸听，在我的最近咨询过程中已经鉴证了很多次，比如在这个移动优先的时代去强化PC应用的技术架构。
- 其次在这个软件应用空前发展的时代，始终抱着所有模块都必须是“自研”，所有代码都必须自己写的思想，毫无疑问只能成为“小作坊”。构建现代的复杂系统已经逐步成为一个生态工程，随着数字化服务的普及，识别哪些领域应该直接外购使用也成为了开发团队的重要能力，构建一个典型的移动应用应该没有人再去重头写一个二维码扫描模块，而是学会从市场上选择适合的软件包。

那么什么地方应该建，什么地方应该买，应该如何决定呢？这时候我们会发现Subdomain的划分就非常有指导意义了。类似二维码扫描这样的Generic领域显然应该是外购的，而当年京东对电商的理解来看物流系统是要自建的。同样道理还有上次DDD China大会来分享的盒马生鲜，半年时间已经重写了三次核心ERP系统。不去思考问题划分的同学们会觉得盒马疯了，ERP在外部看来是多么成熟的软件包啊~但事实上盒马生鲜的本质就在如何解决生鲜食品的高效配送上，也可以说是一家特殊的物流公司。



小结一下，即使区分问题和解决方案很抽象，划分子问题很烧脑，我们还是必须认识到分析问题本身的重要性和必要性。借用雷布斯的成名句“不要用战术上的勤奋掩盖战略上的懒惰”！

# Subdomain和Bounded Context的对应关系？

探讨了Subdomain的必要性，自然我们需要分析和解决方案这边Bounded Context分解的关系。第一次看Eric构建的DDD模型脑图（如下）时，我一直认为少画了Subdomain和Bounded Context的对应关系。最早采用DDD时，个人认知是一个Subdomain下应该有多个Bounded Context，即当我们分析出了一个子问题后在针对建模的解决方案进行分解，成为多个Bounded Context。所以Subdomain: Bounded Context应该是1: N的关系。



(Eric构建的DDD模型脑图)

然而Vernon一直以来的实践方式隐含着1: 1的对应关系。这样的对应关系并非没有道理，如果咱们从一个Bounded Context出发，我们会发现每个Bounded Context必然应该是“解决”部分问题的，而这个部分问题是否就应该是一个Subdomain呢？

当我们拿着这个差异去跟Event Storming的发明者Alberto Brandolini讨论时，发现对方委婉地表达了N: N的理解。简而言之没有直接的对应关系。当然这种理解隐含了一个Bounded Context是可以服务于多个Subdomain子问题的。比如“产品展示” Bounded Context的模型可能服务于产品销售和产品评论两个Subdomain子问题。



这三个对应关系的理解暴露出了大家对问题和解决方案这个老大难问题的纠结~ 当然最简单的是能够建立一对一的映射，作为解决方案高手的程序员们显然是非常喜欢这个模式的。以至于很多用DDD建模的程序员直接就跳过Subdomain搞起了Bounded Context。当然这也是我坚决反对这样简单化映射关系的重要原因。

出于对方法实操性的考虑，我仍然认为一对多的映射是最优的选择。诚然在我们的现实世界里，问题和解决方案是没有必然对应关系的，他山之石可以攻玉也是古来有之的。但软件设计本身就是一个问题抽象的过程，这个抽象一定会选取一个视角，也就会放弃部分信息。在这样的认知下，其实我并不介意在不同子问题的解决方案里存在一定的重复。

所以，如果让我来站队Subdomain和Bounded Context的对应关系，我仍然会选择一对多。在准确性和易用性之间寻求一个平衡，并保证大家能够更多的关注问题本身。

## 坚持持续认知问题

Subdomain和Bounded Context的讨论随着DDD实践的深入会进一步被大家所讨论，不论大家是否能够共识，这样的讨论都是有好处的。作为软件开发的从业者，在面对这个越来越多不确定性的数字化时代，认知问题本身将越来越重要。

Subdomain和Bounded Context在实际认知过程中一定也是相辅相成，逐步清晰的两个概念。Bounded Context建立一定是针对Subdomain的；而Subdomain的划分又会通过Bounded Context的模型得到持续地验证。



ThoughtWorks洞见  
领域驱动设计

---

# 架构

# 从三明治到六边形

作者: 邱俊涛



## 软件项目的套路

如果你平时的工作是做各种项目（而不是产品），而且你工作的时间足够长，那么自然见识过很多不同类型的项目。在切换过多次上下文之后，作为程序员的你，自然而然的会感到一定程度的重复：稍加抽象，你会发现所有的业务系统都几乎做着同样的事情：

- 从某种渠道与用户交互，从而接受输入（Native App, Mobile Site, Web Site, 桌面应用等等）。
- 将用户输入的数据按照一定规则进行转换，然后保存起来（通常是关系型数据库）。
- 将业务数据以某种形式展现（列表，卡片，地图上的Marker，时间线等）。

稍加简化，你会发现大部分业务系统其实就是对某种形式的资源进行管理。所谓管理，也无非是增删查改（CRUD）操作。比如知乎是对“问题”这种资源的管理，LinkedIn是对“Profile”的管理，Jenkins对构建任务的管理等等，粗略的看起来都是这一个套路（当然，每个系统管理的资源类型可能不止一种，比如知乎还有时间线，Live，动态等等资源的管理）。

这些情况甚至会给开发者一种错觉：世界上所有的信息管理系统都是一样的，不同的仅仅是技术栈和操作的业务对象而已。如果写好一个模板，几乎都可以将开发过程自动化起来。事实上，有一些工具已经支持通过配置文件（比如yaml或者json/XML）的描述来生成对应的代码的功能。

如果真是这样的话，软件开发就简单多了，只需要知道客户业务的资源，然后写写配置文件，最后执行了一个命令来生成应用程序就好了。不过如果你和我一样生活在现实世界的话，还是趁早放弃这种完全自动化的想法吧。

## 复杂的业务

现实世界的软件开发是复杂的，复杂性并不体现在具体的技术栈上。如Java, Spring, Docker, MySQL等等具体的技术是可以学习很快就熟练掌握的。软件真正复杂的部分，往往是业务本身，比如航空公司的超售策略，在超售之后Remove乘客的策略等；比如亚马逊的打折策略，物流策略等。

用软件模型如何优雅而合理的反应复杂的业务（以便在未来业务发生变化时可以更快速，更低错误的作出响应）本身也是复杂的。要将复杂的业务规则转换成软件模型是软件活动中非常重要的一环，也是信息传递往往失真的一环。业务人员说的A可能被软件开发者理解成Z，反过来也一样。

举个例子，我给租来的房子买了1年的联通宽带。可是过了6个月后，房东想要卖房子把我赶了出来，在搬家之后，我需要通知联通公司帮我做移机服务。

如果纯粹从开发者的角度出发，写出来的代码可能看起来是这样的：

```
public class Customer {  
    private String address;  
  
    public void setAddress(String address) {  
        this.address = address;  
    }  
  
    public String getAddress() {  
        return this.address;  
    }  
}
```

中规中矩，一个简单的值对象。作为对比，通过与领域专家的交流之后，写出来的代码会是这样：

```
public class Customer {  
    private String address;  
  
    public void movingHome(String address) {  
        this.address = address;  
    }  
}
```

通过引入业务场景中的概念movingHome，代码就变得有了业务含义，除了可读性变强之外，这样的代码也便于和领域专家进行交流和讨论。Eric在领域驱动设计 (Domain Driven Design) 中将统一语言视为实施 DDD 的先决条件。

## 层次架构 (三明治)

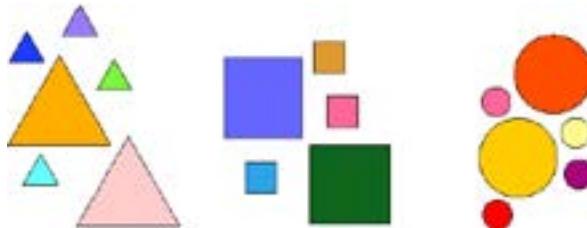
All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections.

— David Wheeler

上文提到，业务系统对外的呈现是对某种资源的管理，而且，现实世界里的业务系统往往要对多种资源进行管理。这些资源还会互相引用，互相交织。比如一个看板系统中的泳道、价值流、卡片等；LinkedIn中的公司，学校，个人，研究机构，项目，项目成员等，它们往往会有嵌套、依赖等关系。

为了管理庞大的资源种类和繁复的引用关系，人们自然而然的将做同样事情的代码放在了统一的地方。将不同职责的事物分类是人类在处理复杂问题时自然使用的一种方式，将复杂的、庞大的问题分解、降级成可以解决的问题，然后分而治之。

- 展现层
- 应用层
- 数据访问层



(图片来自: <http://t.cn/RSNienv>)

比如在实践中，展现部分的代码只负责将数据渲染出来，应用部分的代码只负责序列化/反序列化、组织并协调对业务服务的调用，数据访问层则负责屏蔽底层关系型数据库的差异，为上层提供数据。这就是**层级架构**的由来：上层的代码直接依赖于临近的下层，一般不对间接的下层产生依赖，层次之间通过精心设计的API来通信（依赖通常也是单向的）。

以现代的眼光来看，层次架构的出现似乎理所应当、自然而然，其实它也是经过了很多次的演进而来的。以JavaEE世界为例，早期人们会把应用程序中负责请求处理、文件IO、业务逻辑、结果生成都放在servlet中；后来发明了可以被Web容器翻译成servlet的JSP，这样数据和展现可以得到比较好的分离（当然中间还有一些迂回，比如JSTL、taglib的滥用又导致很多逻辑被泄露到了展现层）；数据存储则从JDBC演化到了各种ORM框架，最后再到JPA的大一统。

如果现在把一个Spring-Boot写的RESTful后端，和SSH (Spring-Struts-Hibernate) 流行的年代的后端来做对比，除了代码量上会少很多以外，层次结构上基本上并无太大区别。不过当年在SSH中复杂的配置，比如大量的XML变成了代码中的注解，容器被内置到应用中，一些配置演变成了惯例，大致来看，应用的层次基本还是保留了：

在有些场景下，应用层内还可能划分出一个服务层。



## 前后端分离

随着智能设备的大爆发，移动端变成了展现层的主力，如何让应用程序很容易的适配新的展现层变成了新的挑战。这个新的挑战驱动出了前后端分离方式，即后端只提供数据（JSON或者XML），前端应用来展现这些数据。甚至很多时候，前端会成为一个独立的应用程序，有自己的MVC/MVP，只需要有一个HTTP后端就可以独立工作。



前后端分离可以很好的解决多端消费者的问题，后端应用现在不区分前端的消费者到底是谁，它既可以是通过4G网络连接的iOS上的Native App，也可以是iMac桌面上的Chrome浏览器，还可以是Android上的猎豹浏览器。甚至它还可以是另一个后台的应用程序：总之，只要可以消费HTTP协议的文本就可以了！

这不得不说是一个非常大的进步，一旦后端应用基本稳定，频繁改变的用户界面不会影响后端的发布计划，手机用户的体验改进也与后端的API设计没有任何关系，似乎一切都变的美好起来了。

## 业务与基础设施分离

不过，如果有一个消费者（一个业务系统），它根本不使用HTTP协议怎么办？比如使用消息队列，或者自定义的Socket协议来进行通信，应用程序如何处理这种场景？这种情况就好比你看到了这样一个函数：

```
httpService(request, response);
```

作为程序员，自然会做一次抽象，将协议作为参数传入：

```
service(request, response, protocol);
```

更进一步, protocol可以在service之外构造, 并注入到应用中, 这样代码就可以适配很多种协议 (比如消息队列, 或者其他自定义的Socket协议)。比如:

```
public interface Protocol {  
    void transform(Request request, Response response);  
}  
  
public class HTTP implements Protocol {  
}  
  
public class MyProtocol implements Protocol {  
}  
  
public class Service {  
    public Service(Protocol protocol) {  
        this.protocol = protocol;  
    }  
  
    public void service(request, response) {  
        //business logic here  
        protocol.transfrom(request, response);  
    }  
}
```

类似的, 对于数据的持久化, 也可以使用同样的原则。对于代码中诸如这样的代码:

```
persisteToDatabase(data);
```

在修改之后会变成： persistenceTo(data, repository);

应用依赖倒置原则，我们会写出这样的形式：

```
public class DomainService {  
    public BusinessLogic(Repository repository) {  
        this.repository = repository  
    }  
  
    public void perform() {  
        //perform business logic  
        repository.save(record);  
    }  
}
```

对于Repository可能会有多种实现。根据不同的需求，我们可以自由的在各种实现中切换：

```
public class InMemoryRepository implements Repository {}  
public class RDBMSRepository implements Repository {}
```

这样业务逻辑和外围的传输协议、持久化机制、安全、审计等等都隔离开来了，应用程序不再依赖具体的传输细节，持久化细节，这些具体的实现细节反过来会依赖于应用程序。

通过将传统内置在层次架构中的数据库访问层、通信机制等部分的剥离，应用程序可以简单的分为内部和外部两大部分。内部是业务的核心，也就是DDD (Domain Driven Design) 中强调的领域模型（其中包含领域服务，对业务概念的建立的模型等）；外部则是类似RESTful API, SOAP, AMQP, 或者数据库，内存，文件系统，以及自动化测试。

这种架构风格被称为六边形架构，也叫端口适配器架构。

# 六边形架构 (端口适配器)

六边形架构最早由Alistair Cockburn提出。在DDD社区得到了发展和推广，然后IDDD (《实现领域驱动设计》)一书中，作者进行了比较深入的讨论。

## 02 Overview



(图片来自: slideshare.net)

简而言之，在六边形架构风格中，应用程序的内部（中间的橙色六边形）包含业务规则，基于业务规则的计算，领域对象，领域事件等。这部分是企业应用的核心：比如在线商店里什么样的商品可以打折，对那种类型的用户进行80%的折扣；取消一个正在执行的流水线会需要发生什么动作，删除一个已经被别的Job依赖的Stage又应该如何处理。

而外部的，也是我们平时最熟悉的诸如REST, SOAP, NoSQL, SQL, Message Queue等，都通过一个端口接入，然后在内外之间有一个适配器组成的层，它负责将不同端口来的数据进行转换，翻译成领域内部可以识别的概念（领域对象，领域事件等）。

内部不关心数据从何而来，不关心数据如何存储，不关心输出时JSON还是XML，事实上它对调用者一无所知，它可以处理的数据已经是经过适配器转换过的领域对象了。

## 六边形架构的优点

- 业务领域的边界更加清晰
- 更好的可扩展性
- 对测试的友好支持
- 更容易实施DDD

要新添加一种数据库的支持，或者需要将RESTful的应用扩展为支持SOAP，我们只需要定义一组端口-适配器即可，对于业务逻辑部分无需触碰，而且对既有的端口-适配器也不会有影响。

由于业务之外的一切都属于外围，所以应用程序是真的跑在了Web容器中还是一个Java进程中其实是无所谓的，这时候自动化测试会容易很多，因为测试的重点：业务逻辑和复杂的计算都是简单对象，也无需容器，数据库之类的环境问题，单元级别的测试就可以覆盖大部分的业务场景。

这种架构模式甚至可能影响到团队的组成，对业务有深入理解的业务专家和技术专家一起来完成核心业务领域的建模及编码，而外围的则可以交给新人或者干脆外包出去。

在很多情况下，从开发者的角度进行的假设都会在事后被证明是错误的。人们在预测软件未来演进方向时往往会展开很多错误的决定。比如对关系型数据库的选用，对前端框架的选用，对中间件的选用等等，六边形架构可以很好的帮助我们避免这一点。

## 小结

软件的核心复杂度在于业务本身，我们需要对业务本身非常熟悉才可能正确的为业务建模。通过统一的语言我们可以编写出表意而且易于和业务人员交流的模型。

另一方面模型应该尽可能的和基础设施（比如JSON/XML的，数据库存储，通信机制等）分离开。这样一来可以很容易用mock的方式来解耦模型和基础设施，从而更容易测试和修改，二来我们的领域模型也更独立，更精简，在适应新的需求时修改也会更容易。

# 端口和适配器架构——DDD好帮手

作者：周宇刚



- 本文源自2018领域驱动设计中国峰会《领域驱动设计与演进式架构专题》的Session之一，是其博客版。
- 在实践领域驱动设计时，可以挑选一些方法互为参照，端口和适配器架构概念简单，容易掌握，适合作为实践领域驱动设计的辅助方法。

大概一个月前，在做2018年领域驱动设计大会预告的时候，上一届大会的主题演讲者肖然提出这样的担忧：工具和方法似乎没有很好地解决“落地难”的挑战。

1. 没有一套方法能够打遍天下，具体到采用哪一种方案，仿佛都需要增加一个定语“这取决于……”。
2. 不管是在DDD原著，还是后续不少专家的书籍中，都暗示、甚至明示架构设计的终极大招还是By Experience—靠经验吃饭。
3. 从战略角度的子领域划分，到战术建模层面实体、值对象的选择，最终的决策很可能不是完全“理性”的，经验这个“感性”的东西发挥着很大的作用”。

所以，推动领域驱动设计实践的方向是否应该从介绍方法转变为介绍如何累积经验？

看了这篇文章后，我放弃了之前准备的话题《CQRS和Event Sourcing, 从入门到放弃》，因为可能你一年都不会遇到一个需要使用这两种方法才能解决的复杂项目。

如何快速获取经验？无非就是多练，但是练了要讨论和总结，我遇到过这样的对话，我将它称为“两小儿辩DDD”：

A: 我觉得你这里不该使用实体，应该使用值对象。  
B: 我觉得你这个接口不是领域服务，它其实是应用服务，你这样做不DDD。  
A: 你的实体不应该调用Repository，你这样做也不DDD。  
B: (看着我)你来评评理，我们谁说的对。  
我: 僻也不知道，这取决于…

这样的复盘方式效果欠佳，我建议不妨从DDD中跳出，找一种方法互为参照和检验，比如“端口和适配器架构”。

## 什么是端口和适配器架构

套用流行的提问方式：当我们在说架构时，我们在说什么？在本文中我们不是在讨论微服务架构，也不是讨论基础设施架构，这里的架构指：

1. 在单个应用（进程）中。
2. 代码是如何组织起来实现一个端到端的用户请求的。
3. 它与框架无关，不管你是使用ORM框架或是JDBC，这不是架构的关键差异点。

一个例子是三层架构，展现层负责接收用户指令、渲染视图；业务逻辑层负责处理“业务逻辑”；数据层负责和数据库打交道，保存和读取数据。

### “经典”的三层架构

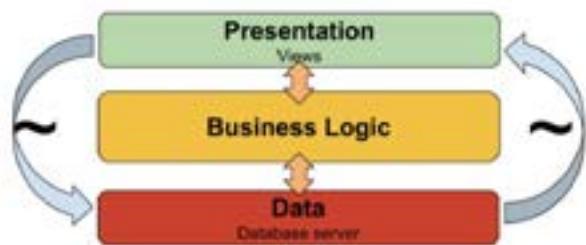
三层（或多层）架构仍然是目前最普遍的架构，但它也有缺点：

1. 架构被过分简化，如果解决方案中包含发送邮件通知，代码应该放置在哪些层里？
2. 它虽然提出了业务逻辑隔离，但没有明确的架构元素指导我们如何隔离。

因此，在实际落地时，业务逻辑容易泄漏到展示层中，导致当应用需要一种新的使用方式时（例如开放API），原有的业务逻辑层可能不能快速重用，同样的问题也发生在数据层和业务逻辑层之间。

那么有没有替代的方案？Alistair Cockburn是敏捷运动的早期推动者之一，他于2005年在其博客中提出了端口和适配器架构，他对该架构的一句话定义是：

“应用应能**平等**地被用户、其他程序、自动化测试或脚本**驱动**，也可以**独立**于其最终的运行时设备和数据库进行**开发和测试**”。



原文为“Allow an application to **equally be driven** by users, programs, automated test or batch scripts, and to be developed and tested in **isolation** from its eventual run-time devices and databases.”

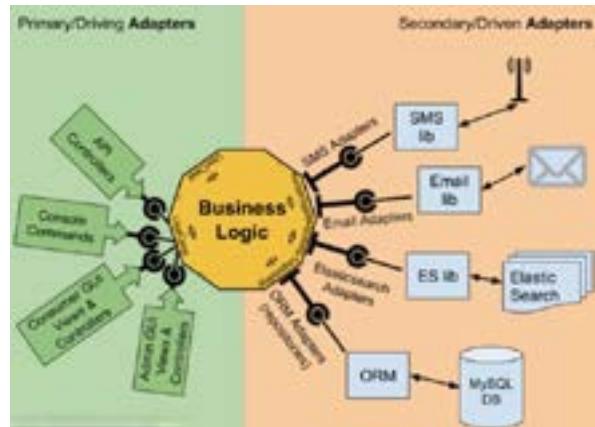
该架构由端口和适配器组成，所谓端口是应用的入口和出口，在许多语言中，它以接口的形式存在。例如以取消订单为例，“发送订单取消通知”可以被认为是一个出口端口，订单取消的业务逻辑决定了何时调用该端口，订单信息决定了端口的输入，而端口为预订流程屏蔽了通知发送方式的实现细节。

而适配器分为两种，主适配器（别名 **Driving Adapter**）代表用户如何使用应用，从技术上来说，它们接收用户输入，调用端口并返回输出。Rest API 是目前最常见的应用使用方式，以取消订单为例，该适配器实现 Rest API 的 Endpoint，并调用入口端口 CancelOrderService。同一个端口可能被多种适配器调用，例如 CancelOrderService 也可能会被实现消息协议的 Driving Adapter 调用以便异步取消订单。

次适配器（别名 **Driven Adapter**）实现应用的出口端口，向外部工具执行操作，例如

- 向MySQL执行SQL，存储订单
- 使用Elasticsearch的API搜索产品
- 使用邮件/短信发送订单取消通知

若将其可视化，Driving Adapter 和 Driven Adapter 基于端口围绕着应用形成左右结构，有别于传统的分层形象，形成一个六边形，因此也会称作六边形架构。



### 可视化端口和适配器架构

如果到此我已经成功地把你讲晕了，请不要担心，我们接下来通过一个案例体验一下这个架构。

## 端口和适配器架构有什么好处

DDD邮轮，有咨询公司的报告显示，在接下来的几年内，邮轮游作为国人出游形式的比例会大幅上升，在这样一个大背景下，DDD Cruise，一家中国的邮轮公司，正在研发新一代的预订系统，尝试在线邮轮预订。

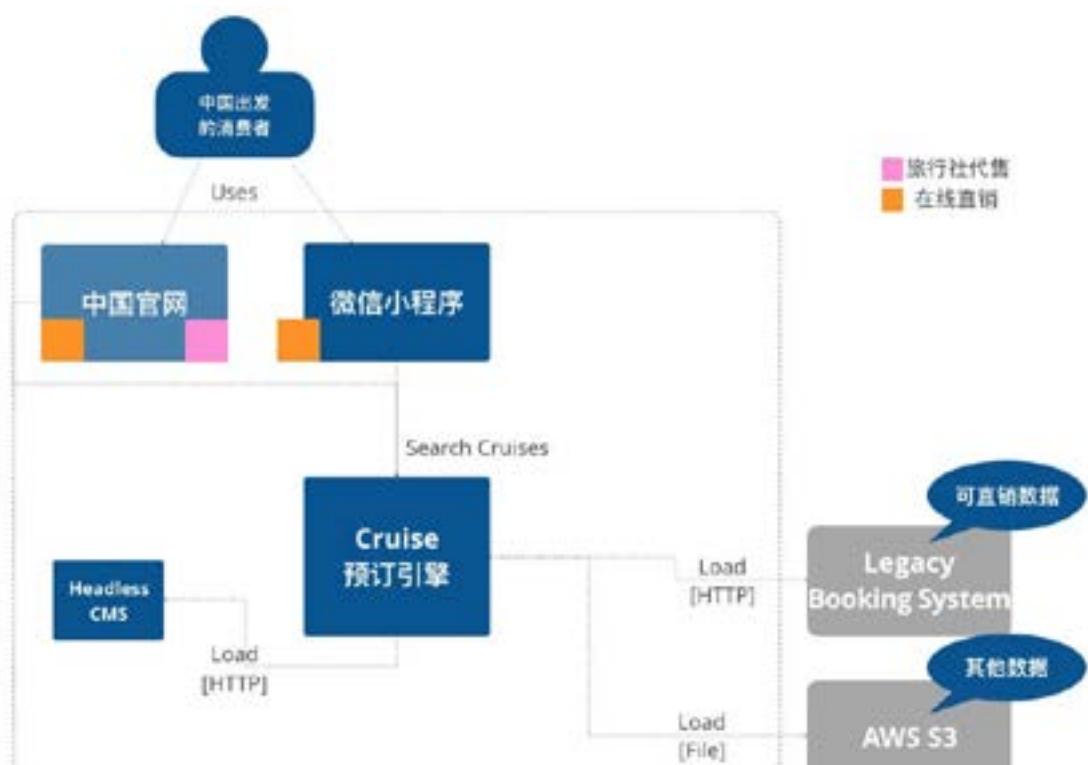
目前计划中有两个触点应用：

### 1. 微信小程序

提供邮轮搜索、邮轮预订的核心体验。

### 2. 中国区官网

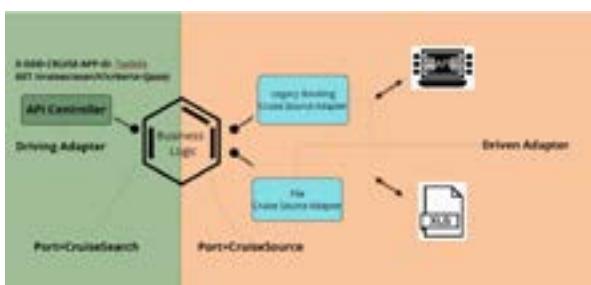
这原是一个包含几个HTML页面的遗留应用，本次希望可以提供邮轮搜索的功能，值得注意的是，有部分邮轮是承包给旅行社销售，在网站上也需要展示以便做市场宣传。



#### C4 Model——System Context Diagram

在这两个触点背后，是这次的主角，预订引擎1.0，计划以一个单体应用起步，为触点应用提供API，实现邮轮搜索、邮轮预订。邮轮有多个数据来源，一部分来自一个遗留的预订系统，一部分来自业务部门的Excel表格，存放在AWS S3对象存储中。最后还有一个小型的Headless CMS为市场人员提供邮轮描述，吸引眼球。

现在让我们代入端口和适配器：

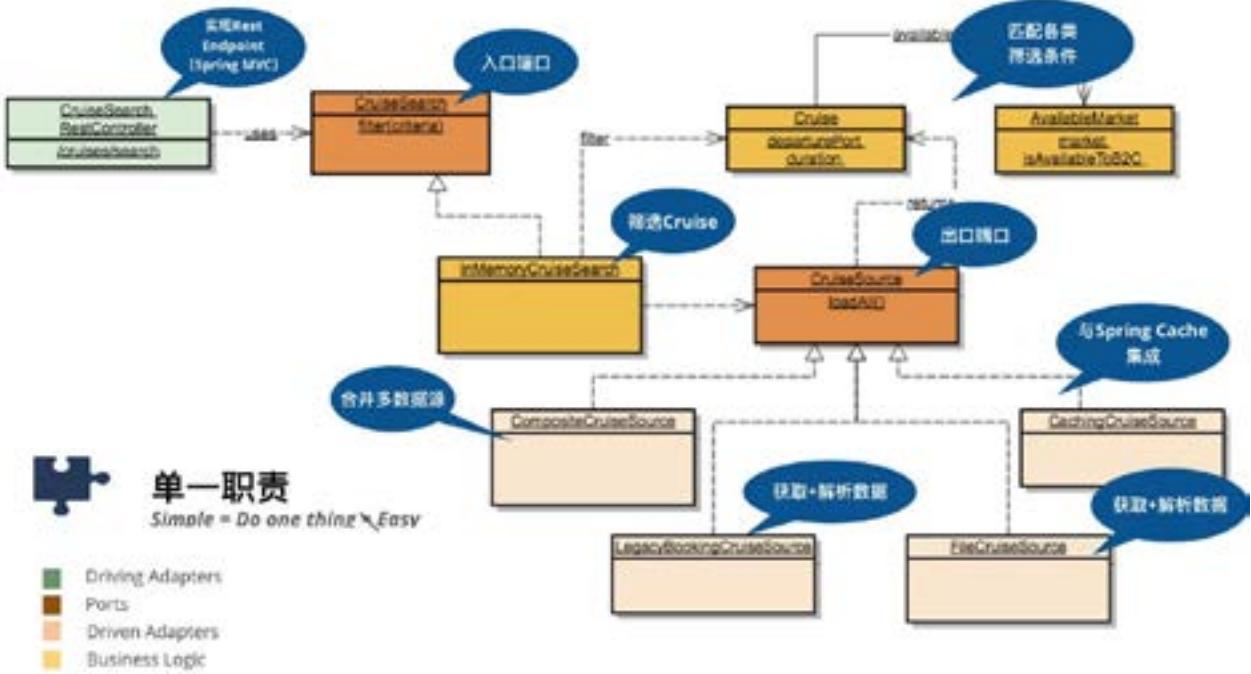


上“套路”，Driving Adapter一个，端口两个，Driven Adapter两个，连线少许。

1. API Controller是一个典型的Driving Adapter，它实现Rest API的Endpoint，调用入口端口CruiseSearch。
2. CruiseSearch作为应用的入口，向Driving Adapter屏蔽了邮轮搜索的实现。
3. 在另一边，出口端口CruiseSource要求返回全量的Cruise数据，为应用隐藏了外部数据源的集成方案：从遗留预订系统或AWS S3上的文件中抽取Cruise。

## 促进单一职责原则

那么我们接下来在这个架构的基础上，进行概要设计，组件很自然地分为了三个部分：



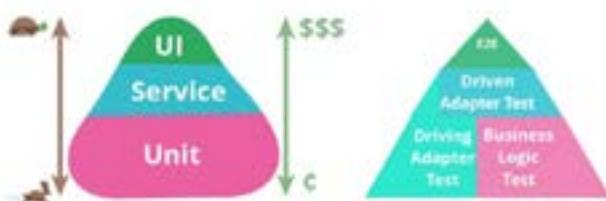
## 概要设计类图

1. 绿色是Driving Adapter, 如果你对Java-Spring技术栈, 可以从命名发现他是一个RestController。
2. 黄色是Cruise Search的实现, 这里的概念只和邮轮相关, 你在这里不应该看到技术术语。
3. 粉色部分则是Driven Adapter, 除了与处理从数据源获取Cruise的Adapter, 我们还需要。
  - a. CompositeCruiseSource, 它不直接与数据源打交道, 但它负责合并多个数据源并根据规则去除重复的Cruise。
  - b. CachingCruiseSource, 它也不直接与数据源打交道, 负责缓存Cruise。

从架构角度来看，这些组件很简单。请注意，简单 (Simple) 并不代表着容易 (Easy)，简单说的是只做一件事 (或一种事)，而容易是指做一件事的难度，例如如果使用Spring MVC实现Driving Adapter，利用注解寥寥几行代码就可以实现。由于这些组件要么实现业务逻辑，要么实现对某种技术的适配，符合单一职责原则，你可以更有效地将变更控制在某一个范围内，更有信心地应对变化。

### 澄清测试策略

应对变化的另一个有效手段是自动化测试，测试金字塔是最常被提及的测试策略，它建议自动化测试集应该由大量单元测试作为基础，它们编写容易、运行速度快，应该只包含少量的用UI驱动的测试，由于需要处理测试数据冲突、外部依赖准备，它们编写困难、运行速度也较慢。但中层的服务/集成测试的测试目标是什么，它们和单元测试有什么区别呢？



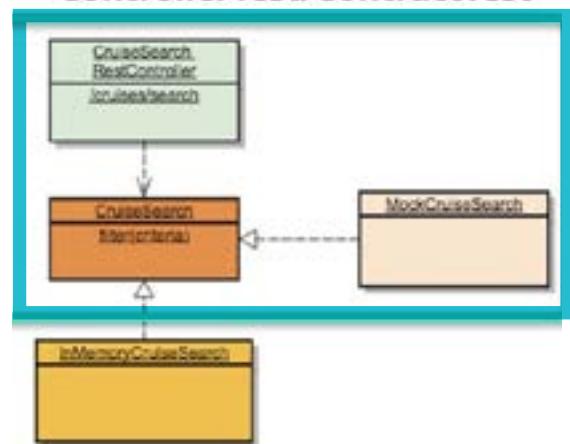
### 测试金字塔——端口和适配器版

如果你也有此困惑，不妨按照端口和适配器架构来重新解读，金字塔应该包含大量的Driving Adapter测试、业务逻辑测试、Driven Adapter测试。

1. Driving Adapter测试，目标是验证API能正确地解析输入、按预期的参数调用了入口端口并生成输出。由于

Driving Adapter不关心入口端口的实现，在测试中，可以通过Mock方便地构造测试场景，并提升测试速度。

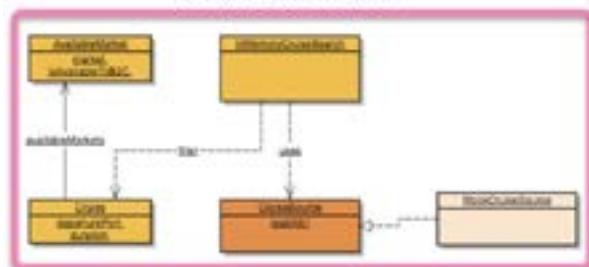
### ControllerTest/ContractTest



近两年开始流行的契约测试也可以认为是Driving Adapter测试的扩展

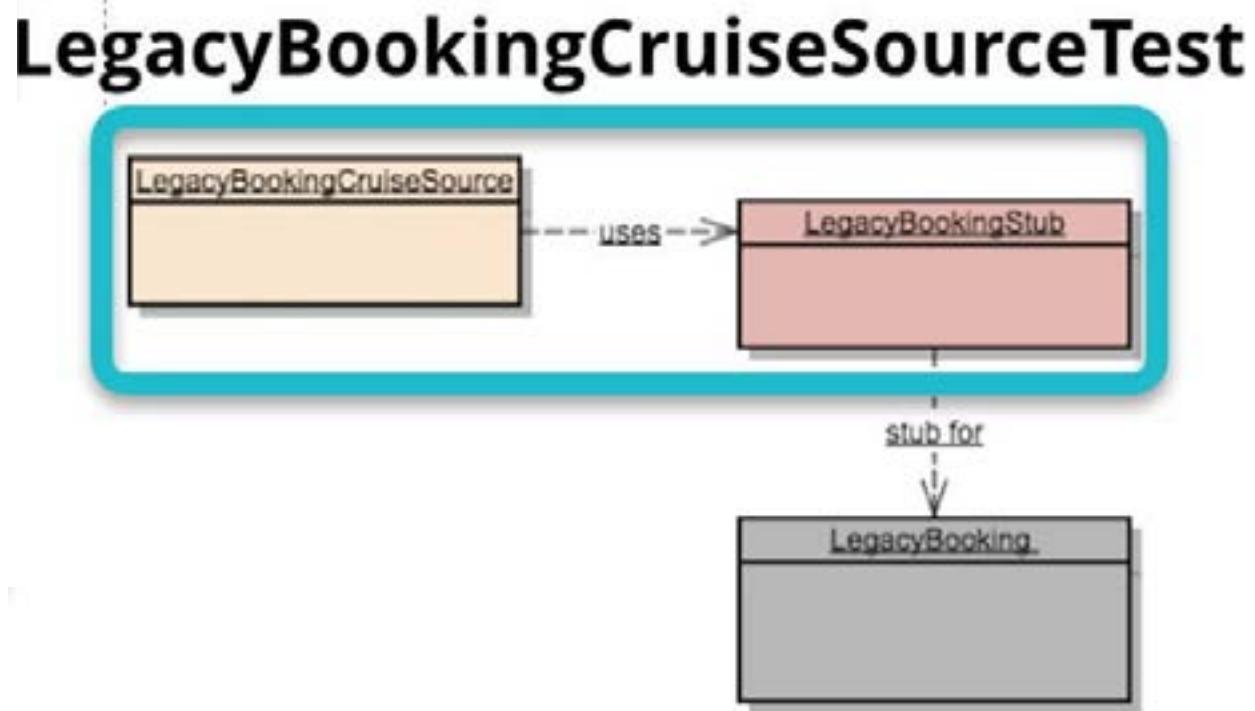
2. 业务逻辑测试，通过Mock出口端口，同样可以方便地构造测试数据，而且这里应该都是Plain Object，测试可以完全在内存中运行，速度是最快的。

### CruiseSearchTest



## 传统的单元测试

3. Driven Adapter测试，目标是验证按预期的方式操作了外部工具、下游服务、数据库。传统上，涉及这些外部依赖的测试编写难度大，运行速度慢，但如果出口端口和Driven Adapter设计得当，它们就不涉及业务逻辑，从而需要测试用例会大大减少，通过引入内存数据库、Stub Server等技术，其测试场景的构建难度会改善不少，整体执行时间也会相应减少。



单一职责的Driven Adapter也降低了测试难度，不过测试速度仍然相对较慢

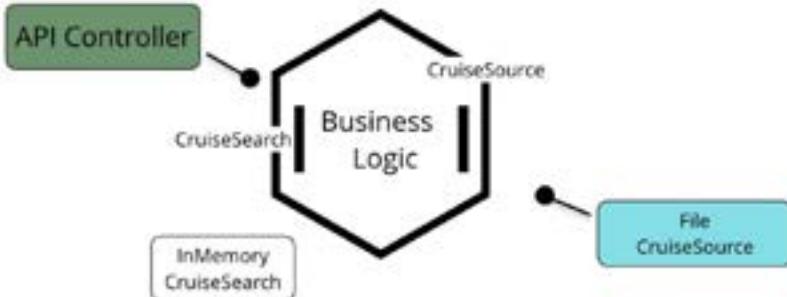
需要注意的是以上测试都是在技术上检测组件是否符合预期，可以考虑适当加入E2E Test来验证这些组件集成起来可用，业务上符合预期，一般覆盖关键功能的Happy Path场景即可。

## 促进增量开发

端口和适配器架构可能还能给我们一些灵感，实施增量开发，不妨看一下这个用户故事分解的例子：

由于旅行社代售的邮轮都来自于Excel表格，只要确定了表格字段含义，我们就可以开始集成，我们选择这张卡来搭建脚手架：

作为中文官网用户。  
我应该可以找到所有非B2C渠道开放的邮轮  
以便我联系代售旅行社进行预订

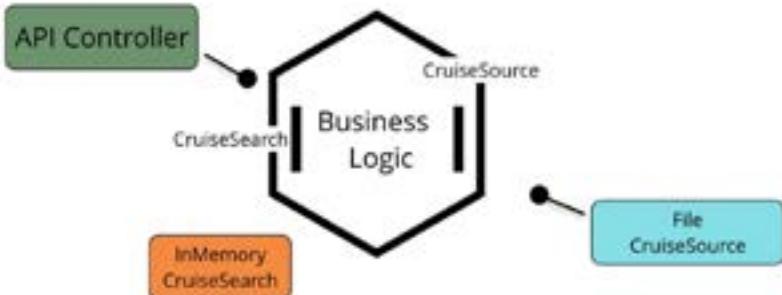


如果业务优先级允许，选择技术实现最简单的卡搭建脚手架

接下来在InMemoryCruiseSearch中实现筛选：

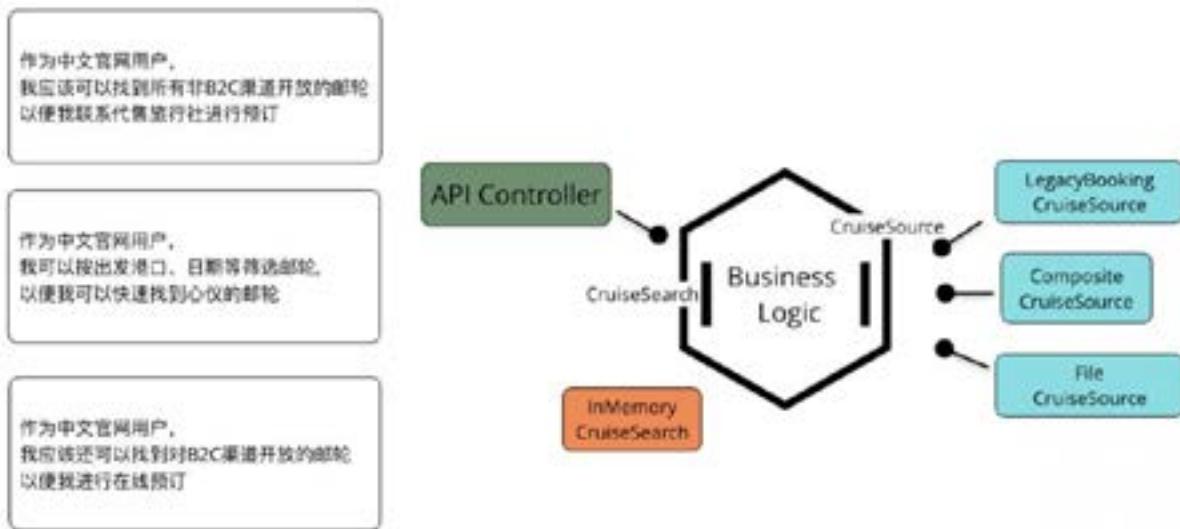
作为中文官网用户。  
我应该可以找到所有非B2C渠道开放的邮轮  
以便我联系代售旅行社进行预订

作为中文官网用户。  
我可以按出发港口、日期等筛选邮轮  
以便我可以快速找到心仪的邮轮



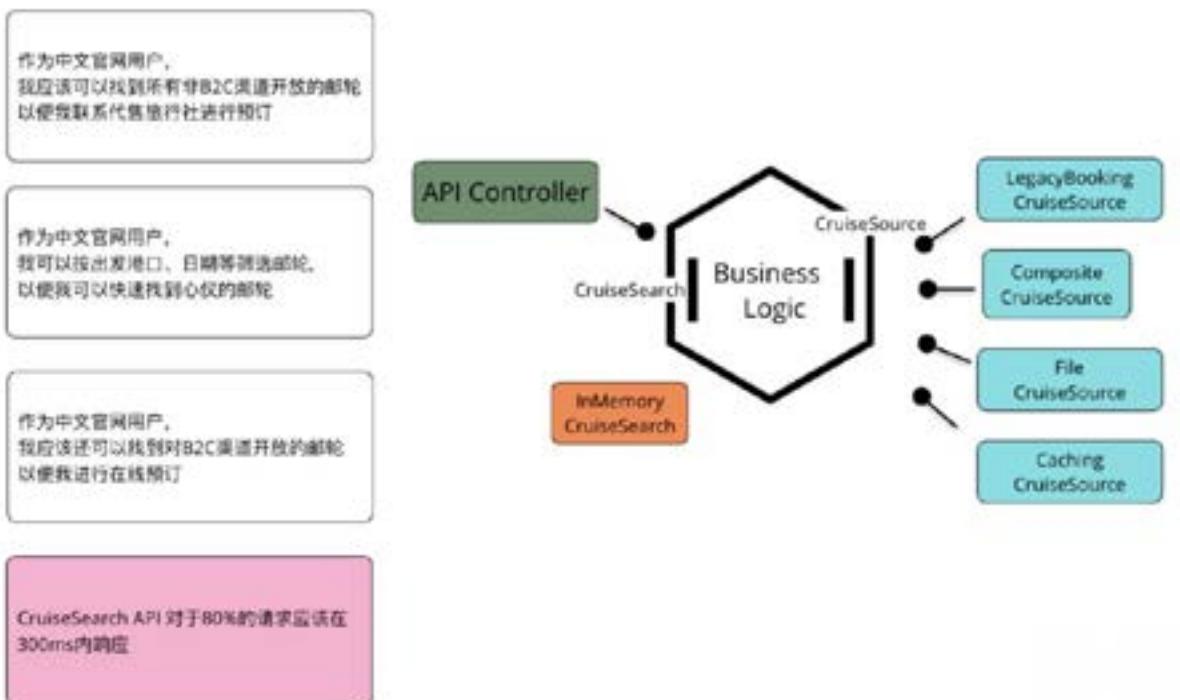
实现筛选功能

引入LegacyBookingCruiseSource和CompositeCruiseSource



扩展数据源，另外还需要扩展Cruise销售渠道的筛选条件实现

最后，可以引入一张技术卡：



加入CachingCruiseSource, 提升Cruise读取速度

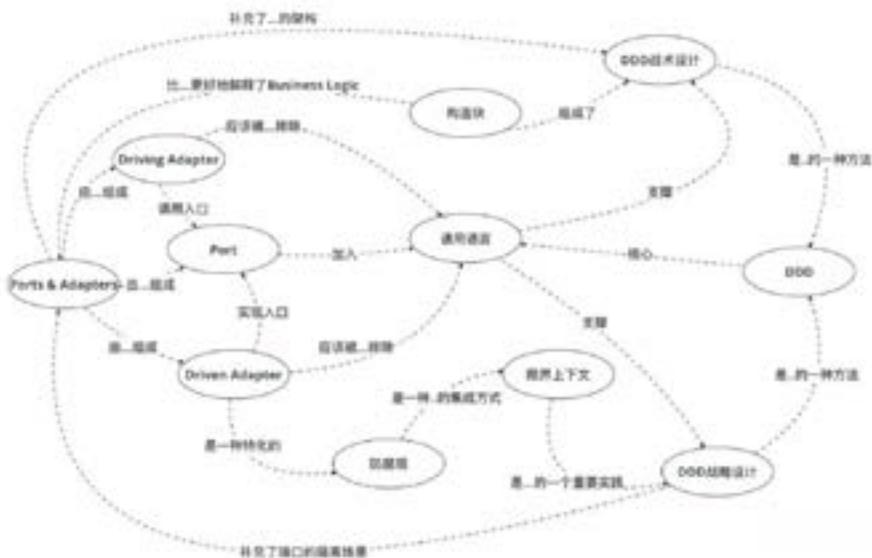
到这里，我们不妨小结一下：



端口和适配器架构的组成元素及它的好处

## 与领域驱动设计的协同增效

由于概念简单、易于掌握，端口和适配器架构很适合作为DDD的入门辅导工具，而领域驱动设计的诸多方法也能够补充端口和适配器架构的空白，形成合力。



端口和适配器架构与领域驱动设计的协同增效

## 校验“通用语言”

通用语言是领域驱动设计的核心精髓，它建议各方（无论是领域专家和还是开发人员）对于同一件事都使用相同的词汇。这可以防止各方在沟通领域问题、制定解决方案时不会由于不同的专业背景产生误解，最终促进了识别正确的问题，采用正确的解决方案。甚至有激进的观点认为“领域模型就是通用语言本身”。

端口和适配器虽然不能直接帮助我们找到领域模型或通用语言，但它有助于我们从通用语言中快速剔除技术概念：凡是用于实现适配器的技术细节都应该被排除。让我们回到DDD Cruise的例子：

开发团队：中文小程序上可以展示哪艘邮轮

业务负责人：所有从遗留预订系统来的邮轮都可以。目前该系统返回的邮轮都是从中国出发的，而且支持B2C预订

开发团队：理解。那么中文官网呢？

业务负责人：这会复杂一些，一部分也是来自遗留预订系统，除此之外，它还需要展示由旅行社代售的邮轮

开发团队：好的，那我们从哪里可以得到这些邮轮？

业务负责人：这有些麻烦。目前还没有办法直接由某个系统提供这些数据，但我可以从内部系统中导出一份Excel表格，你觉得这可行吗？

开发团队：不是很理想，但我们可以解析这份表格，等有更成熟的来源时再替换

作为中文小程序用户

我希望看到所有~~所有~~遗留预订系统的邮轮

作为中文小程序用户

我希望看到所有从中国出发且对B2C开放的邮轮

作为中文官网用户

我希望看到~~所有~~存在于AWS S3上的Excel中的邮轮

作为中文官网用户

我还希望由旅行社代售的邮轮

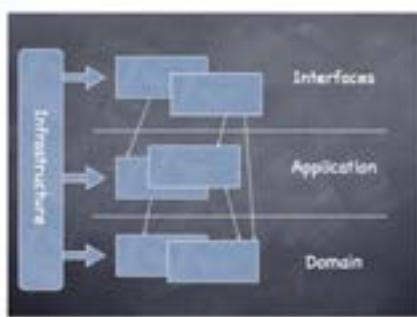
- 领域知识丢失，充斥着实现细节
- 可能误导设计。万一技术实现变化？

- 领域知识、业务规则得到保留
- 将技术细节交给开发者处理

对话片段，注意绿色字体都和Driven Adapter有关，它们应该被通用语言排除。

## 作为“DDD战术设计”的脚手架

领域驱动设计于2004年横空出世，一年后端口和适配器被提出，在战术设计层面，我们可以发现诸多相似点，互为呼应。以架构为例，DDD原著中提出的架构很有意思：乍看之下，以为是传统的分层架构，但却强调了Infrastructure对各层的实现。



*DDD (2004)*

*Interfaces*

*Infrastructure*

*Application + Domain = Business Logic*

*Ports and Adapters(2005)*

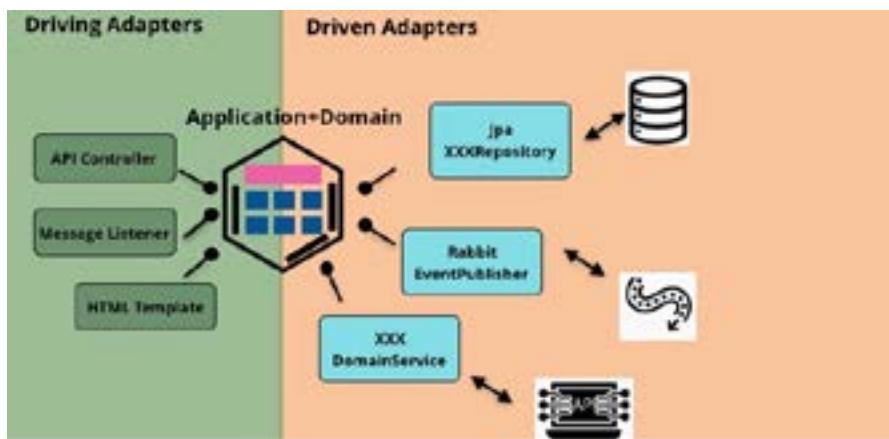
= *Driving Adapters*

= *Driven Adapters*

技术设计  
在此处收敛

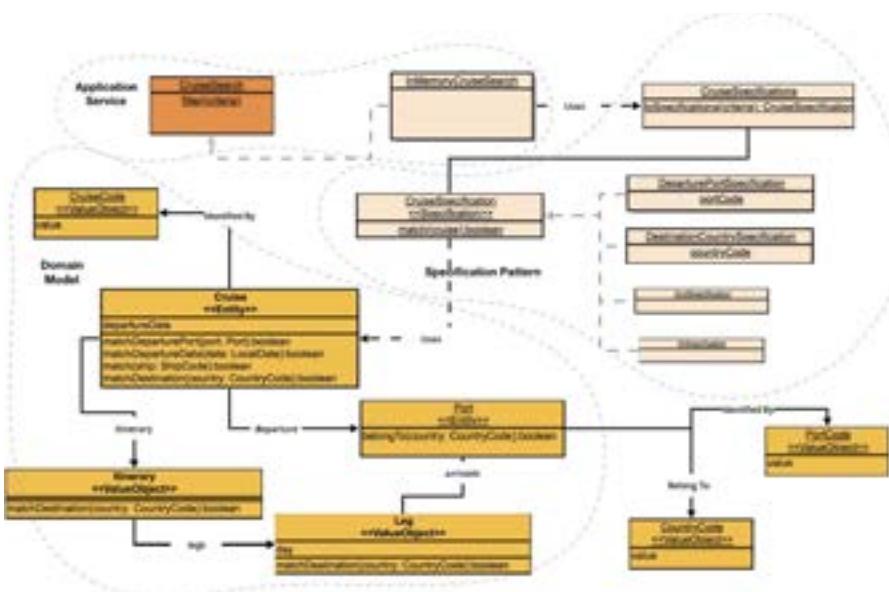
如果我们做一下职责分析，你会发现这不就是端口和适配器嘛？

端口和适配器的优势是突出了分层不是重点，技术实现隔离才是关键，让你不再纠结是否允许组件跨层调用。而 DDD 原著架构的优势是用 Application 和 Domain 进一步澄清了业务逻辑这个模糊的概念。不妨合二为一：



值得一提的是，Application 和 Domain 甚至可以是声明式的，作为端口存在，例如 DDD 构建块中的 ApplicationService 是一个典型的入口端口，而 Repository 则是一个典型的出口端口。

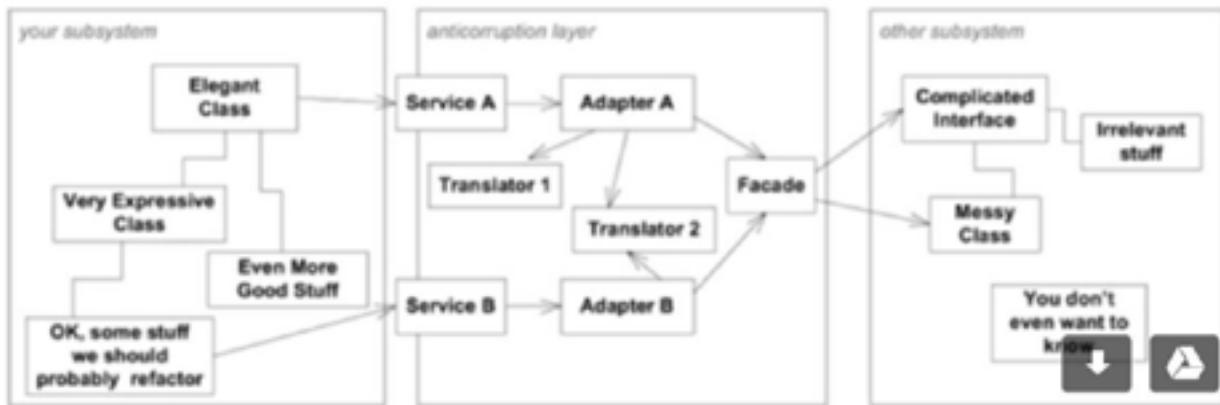
让我们回到 DDD Cruise，细化 Cruise 的领域模型：**CruiseSearch**（应用服务），但实际的筛选逻辑会交给 **Cruise**（实体）及其值对象 **Itinerary**, **Leg** 实现，你甚至可以引入 DDD 书中提到的规格模式，进一步强化单一职责，将筛选条件与领域模型筛选方法的映射工作从 **InMemoryCruiseSearch** 中剥离，使其完全只负责步骤协调。



将应用服务、领域模型代入 Cruise Search

# 让“DDD战略设计”指导隔离实施

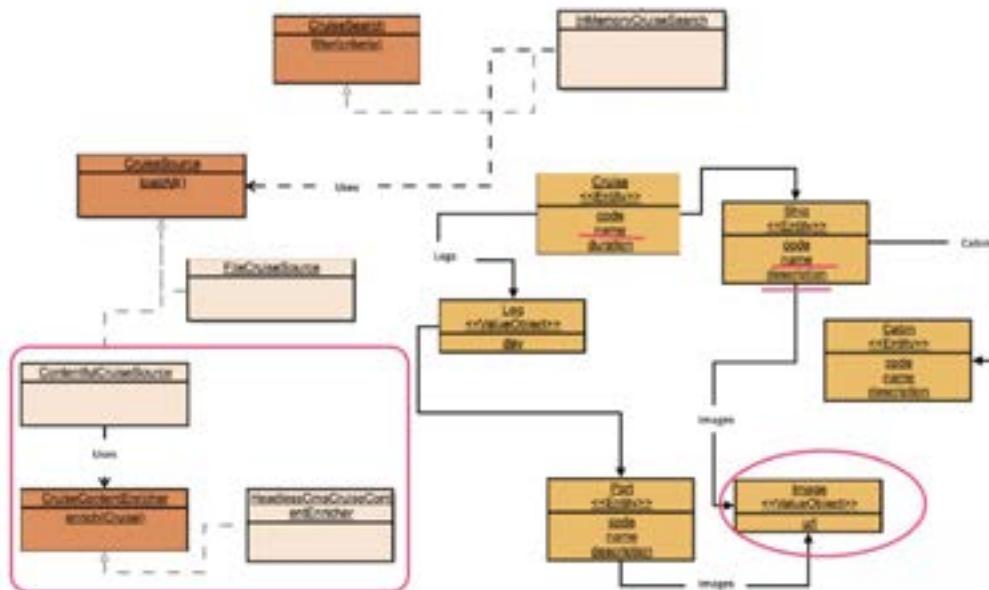
实施战略设计时候，有一个重要的实践是限界上下文的识别，当存在多个限界上下文的时候，很有可能需要集成，防腐层是常见的集成手段。来看这个示意：Service A 是左侧限界上下文暴露出来的接口，通过适配器调用右侧限界上下文的接口。



## 防腐层

这是不是很眼熟？这不正是端口和Driven Adapter吗？你可以认为它们是一种特化的防腐层。那么当一个单体应用中有多个限界上下文时，它们之间也应该用端口隔离，用适配器集成。如果你使用微服务来隔离限界上下文，端口和适配器架构则适用于其中每个服务。

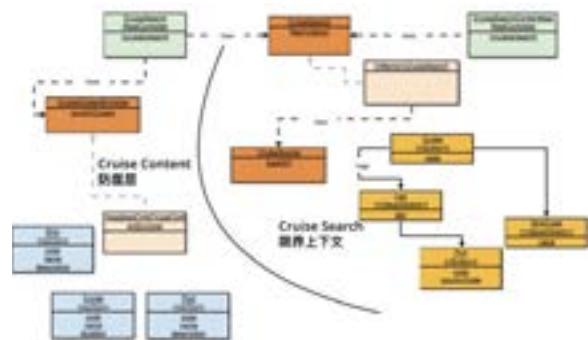
回到DDD Cruise，还记得我们需要集成Headless CMS吗，由于在当前阶段，我们工作在单体应用中，CruiseSearch的API需要返回包含邮轮描述的信息。



没有识别限界上下文，虽然引入了端口和Driven Adapter，但不够理想。

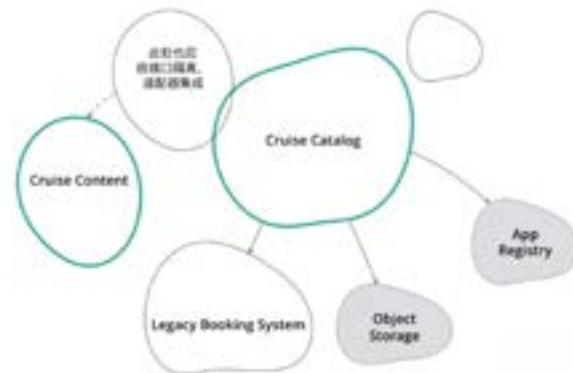
一种方案是将这些描述信息加入到领域模型中，由于已有的两个数据源都无法提供这些信息，我们又引入了**ContentfulCruiseSource**及另一个出口端口**CruiseContentEnricher**及其Driven Adapter以便填充这些信息。但这个方案不够理想：

1. 在邮轮搜索的筛选实现中，描述信息并没有实际作用，领域模型变得更臃肿了，甚至造成了干扰。
2. 在邮轮搜索的测试中，我们并不关心这些描述信息，但却可能需要构造一些Dummy数据，避免可能的空指针误报。



在限界上下文指导下找到更稳定的端口

## 总结



将限界上下文引入DDD Cruise

在限界上下文概念指导下的另一种方案，引入**CruiseContentEnricher**既作为入口端口、同时也作为出口端口，保持邮轮搜索上下文不被干扰，这个方案的好处是，假设邮轮搜索引擎进行微服务改造，很有可能将描述信息填充的职责分离到单独的服务中去，这时，只需要再提供一个输入、输出不含描述信息的Driving Adapter就可以了。

我们介绍了端口和适配器架构，它简单易掌握，和领域驱动设计又合拍，希望它能帮助你快速积累DDD经验！

ThoughtWorks洞见  
领域驱动设计

# 领域事件

# 识别领域事件

作者: 黄亮

随着微服务架构的兴起，微服务设计与拆分的最佳实践DDD已然成为大家讨论与实践的热点，整个行业都在探索如何用DDD建模来实现微服务设计。事件风暴作为最接地气的实践，在不同的项目中野蛮生长，不断演进，今天已经渐渐成熟。作为事件风暴的灵魂——领域事件，值得我们投入更多的精力去设计与打磨。

领域事件是用特定方式(已发生的时态)表达发生在问题域中的重要事情，是领域通用语言(UL)的一部分。为了方便理解这个概念，这里举一个宠物的例子：如果做为宠物主人，你的问题域是如何养好一只猫，那么是不是已经打了疫苗，给宠物饲喂食物等将成为你关注的事情，领域事件会有：疫苗已注射，猫粮已饲喂等。如果你是宠物医生，问题域是如何治好宠物的病，关注的事情是宠物的身体构成，准确的诊断宠物病情，对症下药，领域事件会有：病情已确诊，药方已开治。虽说二者关注的都是宠物，在不同的问题域下领域事件是不同的。



DDD的提出者和圈内的大师先后提到领域事件在领域建模中的价值，前沿实践者们已经开始应用领域事件来表达业务全景。在DDD建模过程中，以领域事件为线索逐步得到领域模型已经成为了主流的实践，即：事件风暴。

事件风暴是以更专注的方式发现与提取领域事件，并将以领域事件为中心的概念模型逐渐演化成以聚合为中心的领域模型，以快速可落地的方式实现了DDD建模。

对于高质量的事件风暴，首先要解决识别领域事件的问题，理想的情况下领域专家和研发团队一起参加事件风暴，从业务的视角去分析涉众关心的领域事件，短时间内高度可视化交流，集中思考，快速发散收敛形成所有参与者一致认可的领域事件集合。我在多个项目上实现事件风暴后，总结了一些坑和应对办法，供大家参考：

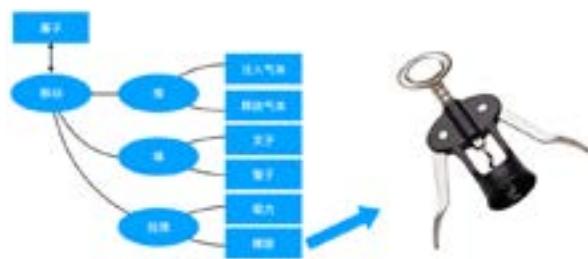
## 1. 组织没有领域专家

对问题域有深刻见解的主题专家称为领域专家，在大多数组织中没有这个角色，当DDD建模需要领域专家支持时，组织往往找业务部门的业务人员，BA，产品经理或在这个领域有多年开发经验的DEV来充当。

这些一线业务人员和开发团队都清楚有什么功能，但往往不清楚为什么有这些功能。举个例子：如果我们的问题是打开一瓶红酒，你去调研每天都会打开酒瓶的waiter，给你的答案是：开瓶器。但换做领域专家的视角来看，会回归问题的本质，如果我们希望打开酒

瓶，需要把瓶塞移除，移除瓶塞的方式有多种，包括推，撬与拉拽，对于拉拽可能基于吸力或螺旋拉拽，下面右图的开瓶器只不过是螺旋拉拽的一种解决方案。领域专家应该对问题域及其中的各种可行方案有更深入的理解。

在辅导团队的过程中，为了弥补这部分视角的缺失，往往会在事件风暴之前，组织业务愿景和场景分析，与被指派的业务干系人对齐业务愿景，一起分析业务场景背后的问题域，找到问题域的本质后再展开事件风暴。

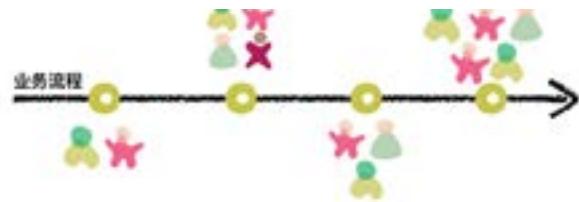


## 2. 面向复杂业务系统的事件风暴

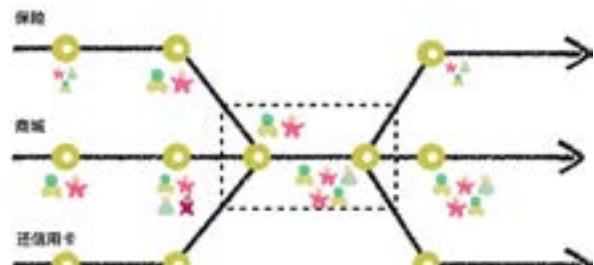
高效事件风暴的规模推荐5-8人，超过8人的事件风暴就会出现讨论时间过长，部分成员参与度不高，业务之间的相关度弱等问题。在一个以支付中台为主题的事风暴中，对于电商商城的支付与理财产品的支付相关性就很弱，各自关心的是自己的业务，让这两组人在一起讨论，在得到同样产出的情况下，会花费双倍的时间。

在处理复杂问题时，一个有效又好用的方法就是分而治之，对于复杂的事件风暴也是同样如此。在业务干系人达到一定规模后，将业务干系人分成多组，组织多轮事件风暴，迭代演进领域模型也是一种不错的选择。

分组的基本原则应以业务线为线索，如果目标系统的业务干系人在同一个业务主线上，每一组人代表业务主线上的一个环节（如下图），这种情况按照业务结点进行分组即可。对于业务相对简单的结点，可以将其与相邻结点合并组织事件风暴。



当目标系统是多条业务线上的某几个公共结点，一般业务中台会出现这种情况，如支付中台要为不同的业务部门（保险，商城，还信用卡等）提供支付服务，如下图中的虚线部分。这类业务往往结点之间的边界并没有那么清楚，系统做什么与不做什么只有在梳理完整条业务线才能确认下来，这种情况按每条业务线分组展开事件风暴，然后针对多组产出结果进行统一业务概念抽象，建立系统边界内的统一事件流。



## 3. 业务代表或领域专家用自己的语言表达业务

事件风暴的第一个环节是让参与者头脑风暴，各自找出业务干系人关注的领域事件，对于业务干系人来讲，往往不适应把自己理解的业务按领域事件的方式表达出来，他们看到一串领域事件，也不觉得这种表达方式

比传统方式直观，在这种情况下，我们就需要考虑如何引导业务共同输出领域事件。留心领域专家在表达需求过程中的一些模式：

1. 当…
2. 如果发生…
3. 当…的时候请通知我
4. 发生…时

通过模式中的关键字转换成领域事件，按时间顺序排序后，基于商业模式与价值定位与领域专家讨论领域事件，以统一的语言与统一的业务视角修正并验证领域事件。高质量的领域事件定义自然是清楚的，是可以找到问题域中的某个actor是关注它的，通过讲述领域事件是可以体现商业价值的。

系统建模同理，我们不关注所有事件，仅关注对干系人解决特定问题有价值的事件，并且这个特定问题应该已经在项目初期，业务愿景梳理的过程中在组织内达成了共识，就像上述投资者关注的问题一样清楚，在业务场景梳理与事件风暴的过程中，不断还原具体过程，以确保识别出的活动或事件真正可以解决业务问题。所以在事件风暴的过程中，并不需要担心是不是找出所有领域事件，只要真正解决了业务问题就好了。

另外，当开始采用新的方法论时，实践过程与角度都有差别，旧有体系的交付物不适用是常有的情况，重点关注的新的方法会不会以更简洁的方式解决实际问题。在存疑的风险处，活学活用新方法的交付物能够让组织更顺利的落地，当然必要的开发过程与交付物改进也是需要的，即可以更高效的完成设计工作，也能够让团队更专注在问题上。

## 总结

有人说微服务的设计与拆分是一门艺术，经验性的成份占了很大比重。当我们准备基于经验来做微服务的设计决策时，结合业务愿景，找出问题域内所有业务干系人真正关心的领域事件，展开完整的事件风暴，循序渐进的让场景变得更加具体，让经验与艺术在生动的问题域之中得到最大的发挥。

另一方面，有效地识别领域事件，既统一了语言，又助力在模型中体现出业务价值部分，为设计关注业务价值的领域模型打下了坚实的基础。

## 4. 事件风暴可能识别不出来所有领域事件

通过事件风暴可以快速把整个问题域主线梳理出来，这样的产出是相当的高效和有价值，但对于正在尝试用事件风暴成果代替传统交付物的组织，往往会质疑事件风暴是否可以发现所有领域事件。

试考虑一个投资者，为一座摩天大楼的建造提供资金，投资者未必对建造过程的细节感兴趣，材料的选择及各种工程细节会议对于建造者来说是很重要的活动，对于投资者来讲，感兴趣的是良好的投资回报，保护投资免受风险，较为务实的投资者会设立明确的里程碑，每个里程碑通过后再做下一次注资。例如，在项目开始时，提供适量资金进行建筑设计工作。当建造事宜被批准时，再为项目提供较多的资金以进行设计工作。在设计通过评审通过后，才拔给更大量的资金，以便建造者破土动工。梳理得到事件如下：

# 在微服务中使用领域事件

作者: 滕云



稍微回想一下计算机硬件的工作原理我们便不难发现，整个计算机的工作过程其实就是一个对事件的处理过程。当你点击鼠标、敲击键盘或者插上U盘时，计算机便以中断的形式处理各种外部事件。在软件开发领域，事件驱动架构 (EventDriven Architecture, EDA) 早已被开发者用于各种实践，典型的应用场景比如浏览器对用户输入的处理、消息机制以及SOA。最近几年重新进入开发者视野的响应式编程 (Reactive Programming) 更是将事件作为该编程模型中的一等公民。可见，“事件”这个概念一直在计算机科学领域中扮演着重要的角色。

## 认识领域事件

领域事件 (DomainEvents) 是领域驱动设计 (Domain Driven Design, DDD) 中的一个概念，用于捕获我们所建模的领域中所发生过的事情。领域事件本身也作为通用语言 (Ubiquitous Language) 的一部分成为包括领域专家在内的所有项目成员的交流用语。比如，在用户注册过程中，我们可能会说“当用户注册成功之后，发送一封欢迎邮件给客户。”，此时的“用户已经注册”便是一个领域事件。

当然，并不是所有发生过的事情都可以成为领域事件。一个领域事件必须对业务有价值，有助于形成完整的业务闭环，也即一个领域事件将导致进一步的业务操作。举个咖啡厅建模的例子，当客户来到前台时将产生“客户已到达”的事件，如果你关注的是客户接待，比如需要为客户预留位置等，那么此时的“客户已到达”便是一个典型的领域事件，因为它将用于触发下一步——“预留位置”操作；但是如果你建模的是咖啡结账系统，那么此时的“客户已到达”便没有多大存在的必要——你不可能在用户到达时就立即向客户要钱对吧，而“客户已下单”才是对结账系统有用的事件。

在微服务 (Microservices) 架构实践中，人们大量地借用了DDD中的概念和技术，比如一个微服务应该对应DDD中的一个限界上下文 (Bounded Context)；在微服务设计中应该首先识别出DDD中的聚合根 (Aggregate Root)；还有在微服务之间集成时采用DDD中的防腐层 (Anti-Corruption Layer, ACL)；我们甚至可以说DDD和微服务有着天生的默契。更多有关DDD的内容，请参考笔者的另一篇文章或参考《领域驱动设计》及《实现领域驱动设计》。

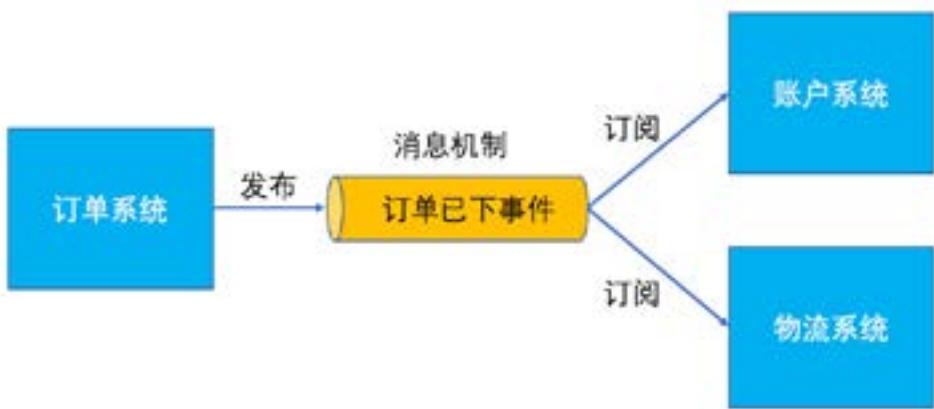
**在DDD中有一条原则：一个业务用例对应一个事务，一个事务对应一个聚合根，也即在一次事务中，只能对一个聚合根进行操作。**但是在实际应用中，我们经常发现一个用例需要修改多个聚合根的情况，并且不同的聚合根还处于不同的限界上下文中。比如，当你在电商网站上买了东西之后，你的积分会相应增加。这里的购买行为可能被建模为一个订单 (Order) 对象，而积分可以建模成账户 (Account) 对象的某个属性，订单和账户均为聚合根，并且分别属于订单系统和账户系统。显然，我们需要在订单和积分之间维护数据一致性，通常的做法是在同一个事务中同时更新两者，但是这会存在以下问题：

- 违背DDD中“单个事务修改单个聚合根”的设计原则；
- 需要在不同的系统之间采用重量级的分布式事务 (Distributed Transaction, 也叫XA事务或者全局事务)；
- 在不同系统之间产生强耦合。

通过引入领域事件，我们可以很好地解决上述问题。总的来说，领域事件给我们带来以下好处：

- 解耦微服务（限界上下文）；
- 帮助我们深入理解领域模型；
- 提供审计和报告的数据来源；
- 迈向事件溯源 (Event Sourcing) 和CQRS等。

还是以上面的电商网站为例，当用户下单之后，订单系统将发出一个“用户已下单”的领域事件，并发布到消息系统中，此时下单便完成了。账户系统订阅了消息系统中的“用户已下单”事件，当事件到达时进行处理，提取事件中的订单信息，再调用自身的积分引擎（也有可能是另一个微服务）计算积分，最后更新用户积分。可以看到，此时的订单系统在发送了事件之后，整个用例操作便结束了，根本不用关心是谁收到了事件或者对事件做了什么处理。事件的消费方可以是账户系统，也可以是任何一个对事件感兴趣的第三方，比如物流系统。由此，各个微服务之间的耦合关系便解开了。**值得注意的一点是，此时各个微服务之间不再是强一致性，而是基于事件的最终一致性。**



## 事件风暴 (Event Storming)

事件风暴是一项团队活动，旨在通过领域事件识别出聚合根，进而划分微服务的限界上下文。在活动中，团队先通过头脑风暴的形式罗列出领域中所有的领域事件，整合之后形成最终的领域事件集合，然后对于每一个事件，标注出导致该事件的命令 (Command)，再然后为每个事件标注出命令发起方的角色，命令可以是用户发起，也可以是第三方系统调用或者是定时器触发等。最后对事件进行分类整理出聚合根以及限界上下文。事件风暴还有一个额外的好处是可以加深参与人员对领域的认识。需要注意的是，在事件风暴活动中，领域专家是必须在场的。更多有关事件风暴的内容，请参考[这里](#)。



## 创建领域事件

领域事件应该回答“什么人什么时候做了什么事情”这样的问题，在实际编码中，可以考虑采用层超类型(Layer Supertype)来包含事件的某些共有属性：

```
public abstract class Event {  
    private final UUID id;  
    private final DateTime createdTime;  
  
    public Event() {  
        this.id = UUID.randomUUID();  
        this.createdTime = new DateTime();  
    }  
}
```

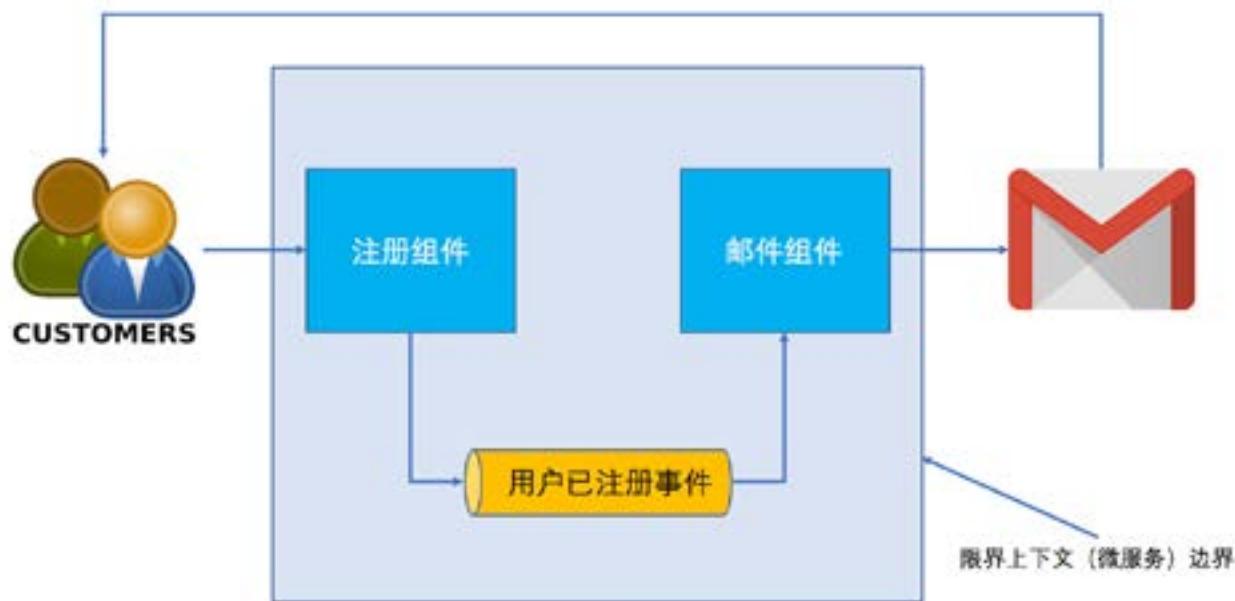
可以看到，领域事件还包含了ID，但是该ID并不是实体(Entity)层面的ID概念，而是主要用于事件追溯和日志。另外，由于领域事件描述的是过去发生的事情，我们应该将领域事件建模成不可变的(Immutable)。从DDD概念上讲，领域事件更像一种特殊的值对象(Value Object)。对于上文中提到的咖啡厅例子，创建“客户已到达”事件如下：

```
public final class CustomerArrivedEvent extends Event {  
    private final int customerNumber;  
  
    public CustomerArrivedEvent(int customerNumber) {  
        super();  
        this.customerNumber = customerNumber;  
    }  
}
```

在这个CustomerArrivedEvent事件中，除了继承自Event的属性外，还自定义了一个与该事件密切关联的业务属性——客户人数(customerNumber)——这样后续操作便可预留相应数目的座位了。另外，我们将所有属性以及CustomerArrivedEvent本身都声明成了final，并且不向外暴露任何可能修改这些属性的方法，这样便保证了事件的不变性。

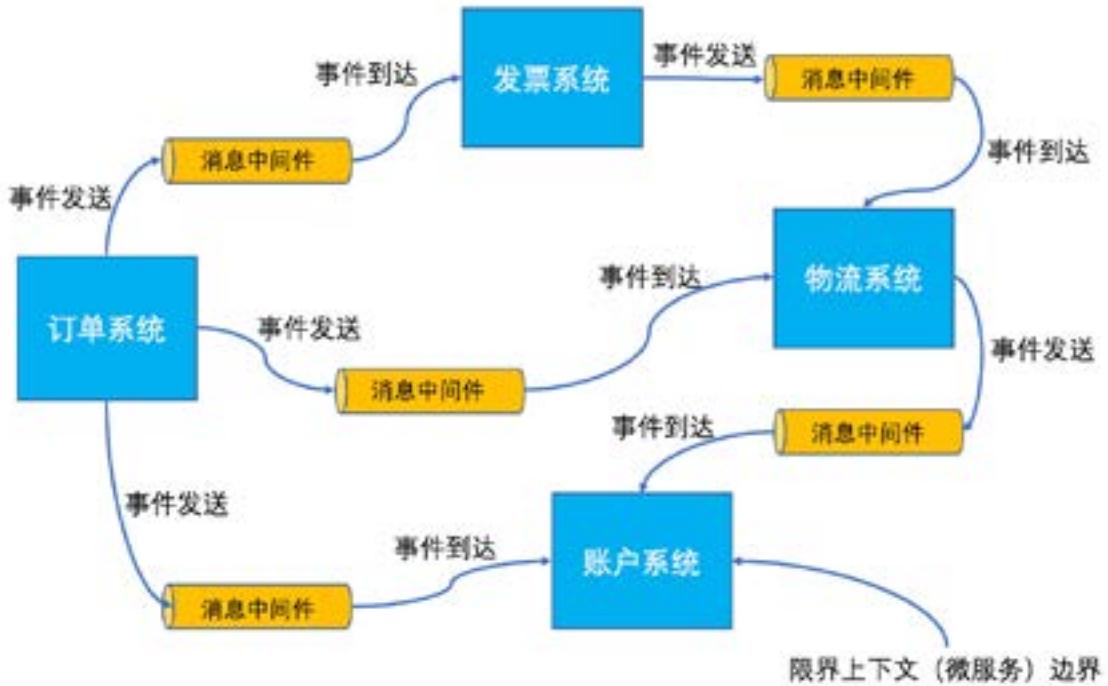
## 发布领域事件

在使用领域事件时，我们通常采用“发布-订阅”的方式来集成不同的模块或系统。在单个微服务内部，我们可以使用领域事件来集成不同的功能组件，比如在上文中提到的“用户注册之后向用户发送欢迎邮件”的例子中，注册组件发出一个事件，邮件发送组件接收到该事件后向用户发送邮件。



在微服务内部使用领域事件时，我们不一定非得引入消息中间件（比如ActiveMQ等）。还是以上面的“注册后发送欢迎邮件”为例，注册行为和发送邮件行为虽然通过领域事件集成，但是他们依然发生在同一个线程中，并且是同步的。另外需要注意的是，在限界上下文之内使用领域事件时，我们依然需要遵循“一个事务只更新一个聚合根”的原则，违反之往往意味着我们对聚合根的拆分是错的。即便确实存在这样的情况，也应该通过异步的方式（此时需要引入消息中间件）对不同的聚合根采用不同的事务，此时可以考虑使用后台任务。

除了用于微服务的内部，领域事件更多的是被用于集成不同的微服务，如上文中的“电商订单”例子。



通常，领域事件产生于领域对象中，或者更准确的说是产生于聚合根中。在具体编码实现时，有多种方式可用于发布领域事件。

一种直接的方式是在聚合根中直接调用发布事件的Service对象。以上文中的“电商订单”为例，当创建订单时，发布“订单已创建”领域事件。此时可以考虑在订单对象的构造函数中发布事件：

```

public class Order {
    public Order(EventPublisher eventPublisher) {
        //create order
        //...
        eventPublisher.publish(new OrderPlacedEvent());
    }
}

```

注：为了把焦点集中在事件发布上，我们对Order对象做了简化，Order对象本身在实际编码中不具备参考性。

可以看到，为了发布OrderPlacedEvent事件，我们需要将Service对象EventPublisher传入，这显然是一种API污染，即Order作为一个领域对象只需要关注和业务相关的数据，而不是诸如EventPublisher这样的基础设施对象。另一种方法是由NServiceBus的创始人Udi Dahan提出来的，即在领域对象中通过调用EventPublisher上的静态方法发布领域事件：

```
public class Order {  
    public Order() {  
        //create order  
        //...  
        EventPublisher.publish(new OrderPlacedEvent());  
    }  
}
```

这种方法虽然避免了API污染，但是这里的publish()静态方法将产生副作用，对Order对象的测试带来了难处。此时，我们可以采用“在聚合根中临时保存领域事件”的方式予以改进：

```
public class Order {  
  
    private List<Event> events;  
  
    public Order() {  
        //create order  
        //...  
        events.add(new OrderPlacedEvent());  
    }  
  
    public List<Event> getEvents() {  
        return events;  
    }  
  
    public void clearEvents() {  
        events.clear();  
    }  
}
```

在测试Order对象时，我们便你可以通过验证events集合保证Order对象在创建时的确发布了OrderPlacedEvent事件：

```
@Test  
public void shouldPublishEventWhenCreateOrder() {  
    Order order = new Order();  
    List<Event> events = order.getEvents();  
    assertEquals(1, events.size());  
    Event event = events.get(0);  
    assertTrue(event instanceof OrderPlacedEvent);  
}
```

在这种方式中，聚合根对领域事件的保存只能是临时的，在对该聚合根操作完成之后，我们应该将领域事件发布出去并及时清空events集合。可以考虑在持久化聚合根时进行这样的操作，在DDD中即为资源库（Repository）：

```
public class OrderRepository {  
    private EventPublisher eventPublisher;  
  
    public void save(Order order) {  
        List<Event> events = order.getEvents();  
        events.forEach(event -> eventPublisher.publish(event));  
        order.clearEvents();  
  
        //save the order  
        //...  
    }  
}
```

除此之外，还有一种与“临时保存领域事件”相似的做法是“在聚合根方法中直接返回领域事件”，然后在Repository中进行发布。这种方式依然有很好的可测性，并且开发人员不用手动清空先前的事件集合，不过还是得记住在Repository中将事件发布出去。另外，这种方式不适合创建聚合根的场景，因为此时的创建过程既要返回聚合根本身，又要返回领域事件。

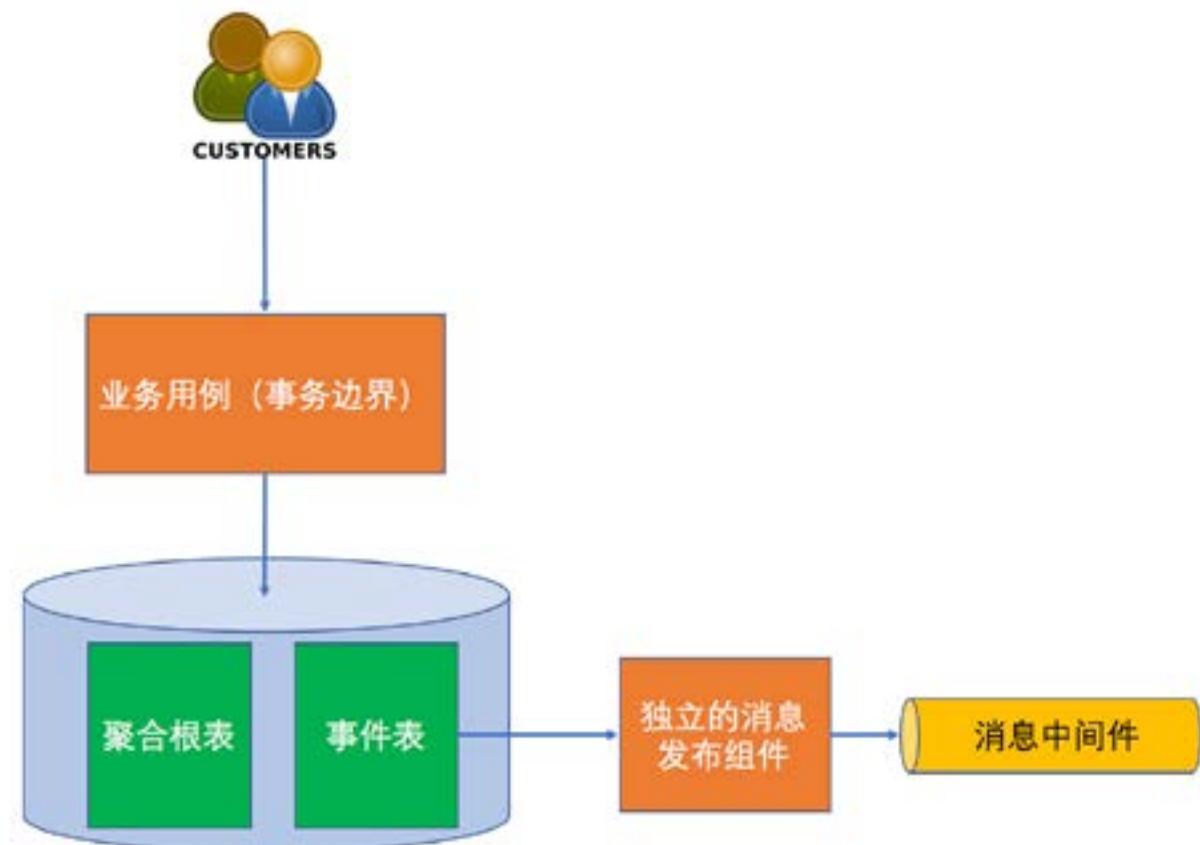
这种方式也有不好的地方，比如它要求开发人员在每次更新聚合根时都必须记得清空events集合，忘记这么做将为程序带来严重的bug。不过虽然如此，这依然是笔者比较推荐的方式。

## 业务操作和事件发布的原子性

虽然在不同聚合根之间我们采用了基于领域事件的最终一致性，但是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。以上文中“订单积分”为例，如果客户下单成功，但是事件发送失败，下游的账户系统便拿不到事件，导致最终客户的积分并不增加。

要保证业务操作和事件发布之间的原子性，最直接的方法便是采用XA事务，比如Java中的JTA，这种方式由于其重量级并不被人们所看好。但是，对于一些对性能要求不那么高的系统，这种方式未尝不是一个选择。一些开发框架已经能够支持独立于应用服务器的XA事务管理器（如Atomikos和Bitronix），比如Spring Boot作为一个微服务框架便提供了对Atomikos和Bitronix的支持。

如果JTA不是你的选项，那么可以考虑采用事件表的方式。这种方式首先将事件保存到聚合根所在的数据库中，由于事件表和聚合根表同属一个数据库，整个过程只需要一个本地事务就能完成。然后，在一个单独的后台任务中读取事件表中未发布的事件，再将事件发布到消息中间件中。



这种方式需要注意两个问题，第一个是由于发布了事件之后需要将表中的事件标记成“已发布”状态，即依然涉及到对数据库的操作，因此发布事件和标记“已发布”之间需要原子性。当然，此时依旧可以采用XA事务，但是这违背了采用事件表的初衷。**一种解决方法是将事件的消费方创建成幂等的，即消费方可以多次消费同一个事件而不污染系统数据。**这个过程大致为：整个过程中事件发送和数据库更新采用各自的事务管理，此时有可能发生的情况是事件发送成功而数据库更新失败，这样在下一次事件发布操作中，由于先前发布过的事件在数据库中依然是“未发布”状态，该事件将被重新发布到消息系统中，导致事件重复，但由于事件的消费方是幂等的，因此事件重复不会存在问题。

另外一个需要注意的问题是持久化机制的选择。其实对于DDD中的聚合根来说，NoSQL是相比于关系型数据库更合适的选择，比如用MongoDB的Document保存聚合根便是种很自然的方式。但是多数NoSQL是不支持ACID的，也就是说不能保证聚合更新和事件发布之间的原子性。还好，关系型数据库也在向NoSQL方向发展，比如新版本的PostgreSQL(版本9.4)和MySQL(版本5.7)已经能够提供具备NoSQL特征的JSON存储和基于JSON的查询。此时，我们可以考虑将聚合根序列化成JSON格式的数据进行保存，从而避免了使用重量级的ORM工具，又可以在多个数据之间保证ACID，何乐而不为？

## 总结

领域事件主要用于解耦微服务，此时各个微服务之间将形成最终一致性。事件风暴活动有助于我们对微服务进行拆分，并且有助于我们深入了解某个领域。领域事件作为已经发生过的历史数据，在建模时应该将其创建为不可变的特殊值对象。存在多种方式用于发布领域事件，其中“在聚合中临时保存领域事件”的方式是值得推崇的。另外，我们需要考虑到聚合更新和事件发布之间的原子性，可以考虑使用XA事务或者采用单独的事件表。为了避免事件重复带来的问题，最好的方式是将事件的消费方创建为幂等的。

# 当提到“事件驱动”时，我们在说什么？

作者: Martin Fowler



文/Martin Fowler

译/梅雪松

去年年底（译者注：2016年底），我和ThoughtWorks同事一起参加了一个研讨会，讨论“事件驱动”的本质。在过去的几年里，我们构建的很多系统都大量使用了事件。对于这些系统，人们常常赞誉有加，但批评的声音也不绝于耳。我们的北美办公室组织了一次峰会，来自世界各地的ThoughtWorks资深开发者出席会议并分享了他们的想法。

这次峰会的最大认识是到当人们谈论“事件”时，实际上说的是完全不同的东西，所以我们花了很多时间来梳理一些有用的模式。本文简要总结我们的成果。

## 事件通知

当领域内有变化发生时，发送事件消息来通知其它系统。事件通知的一个关键点是源系统并不关心外部系统的响应。通常它根本不期待任何结果，即使有也是间接的。发送事件的逻辑流与响应该事件的逻辑流之间会有显著的隔离。

事件通知非常有用，因为它意味着低耦合，并且结构也非常简单。但是，当逻辑处理流跨越各种事件通知时，它也可能成为问题。因为没有任何代码显式地描述这个流程，所以这个流程是不可见的。通常，唯一的办法是通过监控系统来观察它。这会导致调试和修改流程变得很困难。这里的危险在于，当你使用事件通知来优雅地做系统解耦时，没有意识到更大规模的流程，而这会让你在未来几年中陷入困境。不论如何，此模式仍然非常有用，但你必须小心陷阱。

举个例子，将事件用作被动操控型命令(Passive-aggressive command)就属于这种陷阱。它指的是源系统期待接收方执行一个动作，此时本该使用命令消息(Command message)来展现此意图，然而却使用了事件。



事件不需要包含太多数据，通常只有一些ID信息和一个指向发送方、可供查询更多信息的链接。接收方知道它已发生变化，并且接收到关于变化的最少信息，随后会向发送方发出请求，以决定下一步该做什么。

# 事件携带的状态转移 (Event-Carried State Transfer)

采用此模式时，可以在不需要访问源系统的情况下，更新客户端的信息。客户管理系统可能在客户修改自己的详细信息（如地址）时抛出事件，事件包含了详细的修改数据。因此，接收方无需与客户管理系统通信，就可以更新自己的客户数据副本，以进行下一步的操作。

这种模式的一个明显缺点是，有很多冗余数据和副本。但在存储很便宜的时代，这不是一个问题。我们获得了更好的弹性，因为即使客户管理系统不可用时，接收方系统仍然可以正常工作。我们减少了延迟，因为访问客户信息不需要远程调用。我们也不必担心所有来自消费端的查询给客户管理系统带来的负载。但它确实给事件接收端带来了更多复杂性，因为它必须维护所有状态，而如果它直接访问事件发送方查询信息，通常会更加容易。

## 事件溯源

事件溯源(Event Sourcing)的核心思想是，每当系统状态发生变化时，都将状态更改记录为事件，这样我们就有信心在任何时间都能够通过重新处理事件来重建系统状态。事件库成为事实的主要来源，系统状态完全来源于它。对于程序员来说，最好的例子就是版本控制系统。所有的提交日志就是事件库，源码树的工作副本是系统状态。

事件溯源会引入很多问题，但我不会在这里讨论，我想强调一些常见的误解。事件处理不必是异步的，以更新本地Git库为例，这完全是一个同步操作，就像更新Subversion这样的集中式版本控制系统一样。当然拥有所有这些提交允许你做各种有趣的事情，Git就是一个很好的例子，但核心提交从根本上说是一个简单的动作。

另一个常见错误是，假定使用事件溯源系统的每个人都应该理解并访问事件日志以确定有用的数据，但实际上他们很可能对事件日志只具备有限的了解。我正在使用编辑器写这篇文章，编辑器不知道我的源代码树中的所有提交，它只是假设磁盘上有一个文件。在基于事件溯源的系统中，很多处理可以基于一个有效的工作副本。只有当真正需要事件日志中的信息时才必须处理它。如果需要的话，我们可以有多个不同Schema的工作副本，但通常应该在领域处理和通过事件日志派生工作副本之间做明确区分。



使用事件日志时，构建工作副本的快照通常很有用，这样你就不必在每次需要工作副本时都从头开始处理所有事件。实际上这里存在二元性，我们可以将事件日志视为变更列表或状态列表。 我们可以从一个派生出另一个。 版本控制系统通常在事件日志中混合快照和增量变更，以获得最佳性能。<sup>[1]</sup>

考虑一下版本控制系统带来的价值，就很容易明白事件溯源有许多有趣的收益。事件日志提供了强大的审计功能（账户交易是帐户余额的事件溯源）。我们可以重放事件日志到某个点来重新创建历史状态。在重放时注入假设事件可以探索不一样的历史。事件溯源使得非持久化的工作副本（例如Memory Image）变得合理可行。

事件溯源也有自己的问题。当结果依赖于与外部系统的交互时，重放事件就会成为问题。随着时间的推移，我们必须清楚如何处理事件Schema的变化。许多人发现事件处理给系统增加了很多复杂性（尽管我很想知道，主要原因是不是工作副本派生组件和领域处理组件之间糟糕的隔离）。

## CQRS

命令查询职责分离（CQRS）是指读取和写入分别拥有单独的数据结构。严格地说，CQRS跟事件没有关系，因为你完全不需要任何事件就可以使用CQRS。但通常人们会将CQRS与之前的模式结合起来，因此我们在峰会上就此进行了讨论。

使用CQRS的理由是，在复杂领域中，使用单一模型处理读取和写入过于复杂，我们可以通过分离模型来简化。当访问模式有区别时（例如大量读取和非常少的写入），这一点尤其具有吸引力。但是，需要注意平衡CQRS的收益和分离模型所带来的额外复杂度。我发现很多同事对使用CQRS非常警惕，发现它经常被滥用。

## 理解这些模式

作为一名热衷于收集样本的“软件植物学家”，我发现这是一个棘手的地带，主要问题在于不同模式的混淆。在某个项目中，一位能力很强，经验丰富的项目经理告诉我，事件溯源是一场灾难，任何变化都需要两倍的时间来修改读和写模型。在他这句话中，可以发现事件溯源和CQRS之间可能存在混淆，我们如何找出哪个是罪魁祸首？该项目的技术主管声称主要问题是大量的异步通信，这当然是一个已知的复杂性助推器，但异步通信不是事件溯源或CQRS的必要组成部分。总的来说，我们必须要注意这些模式在对的地方都很好，反之则很糟糕。但是当我们混淆了这些模式时，很难弄清楚哪里是对的地方。

ThoughtWorks洞见  
领域驱动设计

# 微服务

# DDD & Microservices

作者: 肖然

Microservices (微服务架构) 和DDD (领域驱动设计) 是时下最炙手可热的两个技术词汇。在最近两年的咨询工作中总是会被不同的团队和角色询问，由此也促使我思考为什么这两个技术词汇被这么深入人心的绑定，它们之间的关系是什么呢？



然后用Microservices来定义了拥有这些特质的架构。之后由于Google、Netflix、Amazon等一系列明星企业都对号入座，Microservices开始风靡整个软件业。这时候很多人会问微服务架构是怎么设计出来的，业界人士会说DDD是一个好方法，其中包括微服务定义者Martin Fowler，毕竟DDD原书的序是他给著的；）于是乎DDD开始在被定义10年后火了。

从我个人角度来看，如果真的需要找到因果关系的话，最根本的驱动力来自于**科技时代对软件系统（数字化）响应力要求的不断提升，而系统的复杂度却随着业务的多元化而与日俱增**。如何驾驭这样的高复杂度成了每个企业必须面对的挑战，以至于业界开始把这种模型总结为**响应力企业** (ResponsiveEnterprise)，而模型中总结的大部分原则都是为了更好的适应环境不确定性带来的高复杂度。

## 服务于更高的业务响应力

首先从两个词汇的发明来看它们是没有因果关系的。DDD是Eric Evans于2003年出版的书名，同时也是这个架构设计方法名的起源。DDD的想法是让我们的软件实现和一个演进的架构模型保持一致，而这个演进的模型来自于我们的业务需求。这种演进式设计方法在当时看来还是比较挑战的，更为流行的解决架构设计复杂度的方法是分层：比如数据架构、服务架构、中间件架构等。MVC在互联网应用开发领域也基本成为了标配。

时间很快过了10年，Martin Fowler和ThoughtWorks英国架构师James Lewis坐下来一起分析了好几个能够持续演进的大型复杂系统，总结出了9大核心特质，



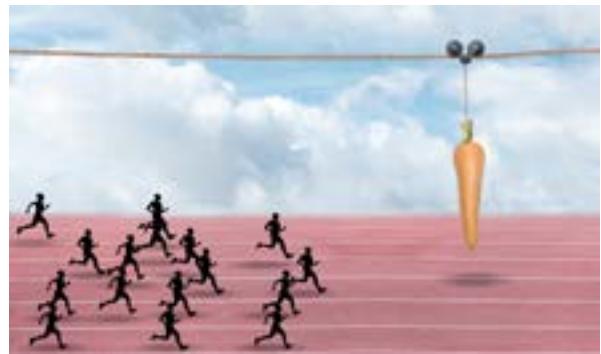
# 从业务视角分离复杂度

每个人能够认知的复杂度都是有限的，在面对高复杂度的时候我们会做关注点分离，这是一个最基本的哲学原则。显然在针对复杂业务场景进行建模时，我们也会应用此原则。这个时候去分离关注点一般可以从两个维度出发：

- 技术维度分离，类似MVC这样的分层思想是我们广泛接受的。
- 业务维度分离，根据不同的业态划分系统，比如按售前、销售、售后划分。

以上两个维度没有孰优孰劣之分，在处理复杂问题的时候一定都会用上，但为了能够高效响应业务的变化，**微服务的架构更强调业务维度的关注点分离来应对高复杂度**。这是显著区别于传统SOA架构的特质之一，比如诞生于传统SOA时代的ESB（工业服务总线）就是一个典型的技术关注点分离出来的中间件。随着业务的变化，我们也看到ESB成为了一个架构上的反模式，即大量的业务规则和流程被封装在了ESB里，让ESB成为了不可驾驭的复杂度之源，以至于破坏了SOA架构之前承诺的各种优势。当然Microservices架构并非是新一代SOA架构这么简单，已经有不少文章在讨论这个话题，本文就不在展开了。

如果这个时代你还觉得自己的架构不需要这种响应力，我建议你问问身边维护3年以上系统的的朋友或同事们，他们会告诉你这是怎样的一种痛苦。实际上很多企业对这种响应力的追求已经很“疯狂”了，这也是微服务的两位定义者可能都始料未及的。



他们在定义文章中带着很强警告语气让大家慎用，但在这个科技时代，微服务架构实施的可能风险对比高响应力在未来可能带来的市场机会几乎可以忽略不计。一个Netflix的成功就足以让大部分企业毫不犹豫的选择微服务作为自身的架构风格。

## 业务和技术渐进统一的架构设计

如果Microservices和DDD在目标上达成了上文的统一，那么在具体做法上和以前有什么不同呢？

为了解释清楚这个问题让我们极简化架构设计为以下三个层面工作：

- 业务架构：根据业务需求设计业务模块及交互关系。
- 系统架构：根据业务需求设计系统和子系统的模块。
- 技术架构：根据业务需求决定采用的技术及框架。

显然这三者在具体一个架构设计活动中应该是有先后顺序的，但并非一定是孰先孰后，比如一个简单的web应用，很多人会说MVC是标配了（首先确定了系统架构），或者有人说用RoR快（首先确定了技术架构）。在给定的业务场景里，也许这样的顺序是合理的。



这个时候咱们增加**复杂业务需求和快速市场变化**这两个环境变量，这个顺序就变得很有意思了。于是我们听到不少走出初创期的互联网服务平台开始“重写”他们的系统（从PHP到Java），很多文章开始反思MVC带来的僵化（臃肿的展现层）。

经历了这样变迁的架构师们都会感同身受的出来为DDD站台，其原因就是“跳过”（或“后补”）业务架构显然表明设计出来的架构关注点并不在业务的响应力上，因为业务的可能变化点并没有被分析出来指导系统和技术架构的设计。

**DDD的核心诉求就是能够让业务架构和系统架构形成绑定关系，从而当我们去响应业务变化调整业务架构时，系统架构的改变是随之自发的。**

这个变化的结果有两个：

- 业务架构的梳理和系统架构的梳理是同步渐进的，其结果是划分出的业务上下文和系统模块结构是绑定的。
- 技术架构是解耦的，可以根据划分出来的业务上下文的系统架构选择最合适的实现技术。

第一点显然也是我们产生微服务划分所必须遵循的，因为微服务追求的是业务层面的复用，所以设计出来的系统必须是跟业务一致的。第二点更是微服务架构的特质：“去中心化”的治理技术和数据管理。作为架构设计的方法，DDD的各种实践，包括最近流行的Event Storming（事件风暴）实际上都是促进业务和系统架构梳理的渐进式认知。



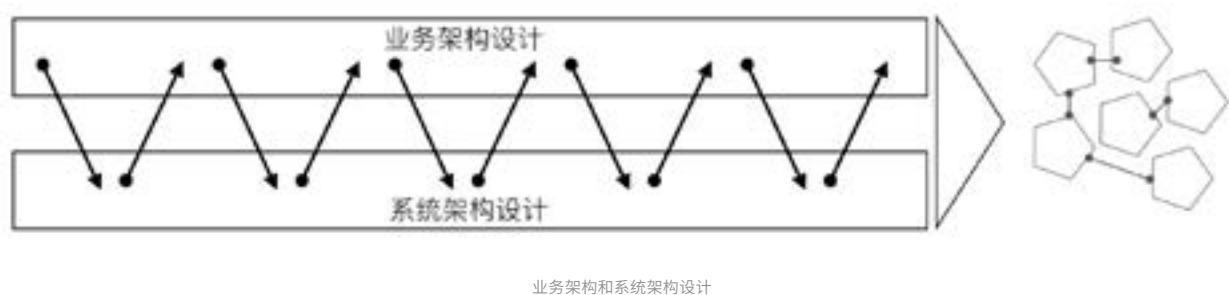
在一次DDD工作坊中，一位同事给出了“你们连业务故事都讲不清楚，还有必要继续做架构设计吗？”这样的经典评论。而DDD的整个方法也没有涉及具体的技术架构实现，这个选型的权利很多时候被“下放”给了真正的开发团队。

值得一提的是采用DDD这种架构设计方法并不一定就产生Microservices这种架构风格，往往会推荐用大颗粒度的服务来包含业务分析过程中发现的不确定点，以避免拆分后变化过度频繁带来的双向修改成本。

## 跨职能协作的架构设计

业务和系统的渐进认知改变了很多之前的架构工作模式，在采用DDD的过程中，很容易感受到业务专家的重要性。而如果还有人寄希望让业务能够一次性给架构师讲清楚需求，那我建议抱有这样希望的同学去亲身参加一次自己不熟悉业务领域的架构设计讨论。你会很容易得出结论“原来业务也不懂他要什么”。当然业务人员听说要参加某种（软件）架构设计方法时心里也一定是抵触的。

DDD成功运用的基础就是创造让业务和系统这两种不同认知模型逐步统一的环境。



所以“不幸”的是如果你不能建立一个跨业务和技术的新型架构设计小组，你的DDD实践就没有成功的基础，继而采用微服务架构可能就会是一场灾难。幸运的是这种跨职能组织结构已经是前文中“采用”微服务架构企业（如Amazon）的标配，你不必再论证这件事情的可实施性。剩下的关键就是如何能够让不同背景的人们协作起来。这也是大家可以看到DDD领域的下一个热点，类似Event Storming这样的模式化协作工作坊会更多的出现在大家的视线里。

## 永无终止的DDD和演进的Microservices

DDD是容易上瘾的，当大家发现原来通过这个建模过程业务专家更了解服务划分（系统模块），架构设计更懂业务需求，这种协作会成为常态。在这个tech@core的时代，这样的融合将成为企业的核心竞争力。

当然刚开始采用DDD方法的时候，请不要认为每个系统搞一次所谓的DDD工作坊就能够找到最佳的服务划分了。业务的变化是持续的，而每次业务架构变化必然牵动系统架构的变化。良好的领域架构绑定了业务和系统，让双方人员能够用统一语言交流，这件事情建立不易，而持续运作更难。

**成功的DDD方法运用是贯穿系统的整个生命周期的，这个过程中业务和技术的协作是持续发生的。**

Microservices的最后一个特质：“演进式”设计 – 也明确了设计是一种持续的活动。DDD提供了一种符合这个微服务特质的工作方法，让演进能够落地。值得一提的是就笔者最近的经验，这个演进过程中最难认知到变化的就是

DDD里最显而易见的“统一语言”。当大家形成了一个业务概念-“客户”后，少有团队能够持续审视这个“客户”是否随着市场的变化而发生了含义的变迁。

对比传统的SOA，微服务的拆分也是动态的，禚娴静在自己的文章中描述一个系统采用微服务架构历程中服务拆分的演变。这里不会有ESB来以不变应万变，这种幻想在过去的10年里已经被数次打脸。DDD的好处是让业务和技术人员都能够在合作中理解这种变化，而不至于陷入业务人员抱怨技术架构不知所谓，技术人员觉得业务人员朝三暮四的尴尬。

## 你需要成为那个高个子！

Martin Fowler在Microservices的定义文章中画了下面的图，评论“你必须有那个高度”来隐喻微服务实施的能力要求。就架构建模方面来说我认为DDD应该是一个团队必须去掌握的，包括这个团队的业务人员和产品设计人员。



微服务前置条件示意

很有意思的是目前**Service Design**也是全球用户体验设计领域的一个热门话题，从用户视角出发去设计整个服务链条。比如时下热门的共享单车，一个成功的服务设计应该是从用户开始有用车需求触发到最后完成骑行缴费离开，而不仅仅是去设计一辆能够互联网解锁的自行车。

我们可以找到很多Service Design和DDD在原则上的相似之处，比如用户中心和协同设计。借用上面的高个子说法：在业务需求认知和跨职能协作方面你一定需要成为高个子！

# 服务拆分与架构演进

作者：禚娴静

领域驱动设计和服务自演进能力是内功。

## 前言

《微服务的团队应对之道》提到，微服务帮助企业提升其响应力，而企业需要从DevOps、服务构建、团队和文化四点入手，应对微服务带来的复杂度和各种挑战，从而真正获益。如果说运维能力是微服务的加油站，服务则是其核心。



企业想要实施微服务架构，经常问到的第一个问题是，怎么拆？如何从单体到服务化的结构？第二个问题是拆完后业务变了增加了怎么办？另外，我们想要改变的系统往往已经成功上线，并有着活跃的用户。那么对其拆分还需要考虑现有的系统运行，如何以安全最快最低成本的方式拆分也是在这个过程中需要回答的问题。

本文会针对以上问题，介绍我们团队在服务拆分和演进过程中的实践和经验总结。

## 我们项目架构的演化历程



该项目始于2009年，到现在已有7年的时间。在这7年中覆盖的业务线不断扩大，从工单、差旅、计费、文件、报表、增值业务等；业务流程从部分节点到用户端的全线延伸；7年间打造多个产品，架构经历了多次调整，从单体架构、RPC、服务化、规模化到微服务。

主要架构变迁如下图所示：



在这7年架构演进路上，我们遇到的主要挑战如下：

- 如何拆？即如何正确理解业务，将单体结构拆分为服务化架构？
- 拆完后业务变了增加了怎么办？即在业务需求不断发展变化的前提下，如何持续快速地演进？
- 如何安全地持续地拆？即如何在不影响当下系统运行状态的前提下，持续安全地演进？
- 如何保证拆对了？
- 拆完了怎么保证不被破坏？

## 问题1：如何将单体结构拆分为服务化架构？

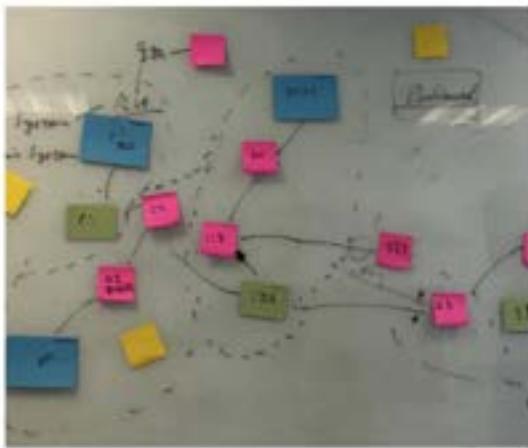
就如庖丁解牛一样，拆分需要摸清内部的构造脉络，在筋骨缝隙处下刀。那么微服务架构中，我们认为服务是业务能力的代表，需要围绕业务进行组织。拆分的关键在于正确理解业务，识别单体内部的业务领域及其边界，并按边界进行拆分。

### 1. 识别业务领域及边界。

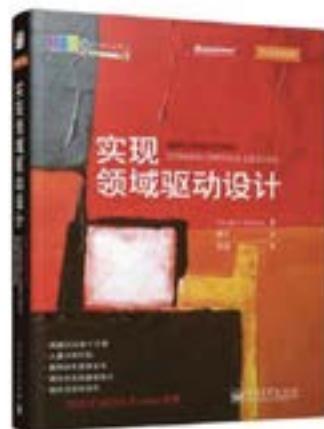
首先需要将客户、体验设计师、业务分析师、技术人员集结在一起对业务需求进行沟通，随后对其进行领域划分，确定限界上下文(Boundary Context)，也称战略建模。

以我们经常使用的方法和参考的红蓝宝书：

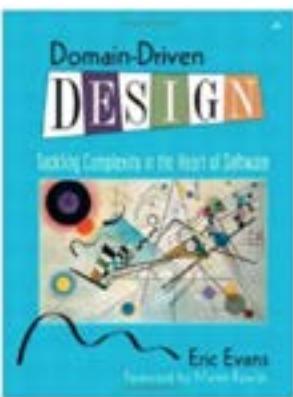
- Inception-> User Journey | Scenarios，用于梳理业务流程，由粗粒度到细粒度逐一场景分析。
- 四色建模，用于提取核心概念、关键数据项和业务约束。
- 领域驱动设计-战略设计，用于划分领域及边界、进行技术验证。
- Eventstorming，用于提取领域中的业务事件，便于正确建模。



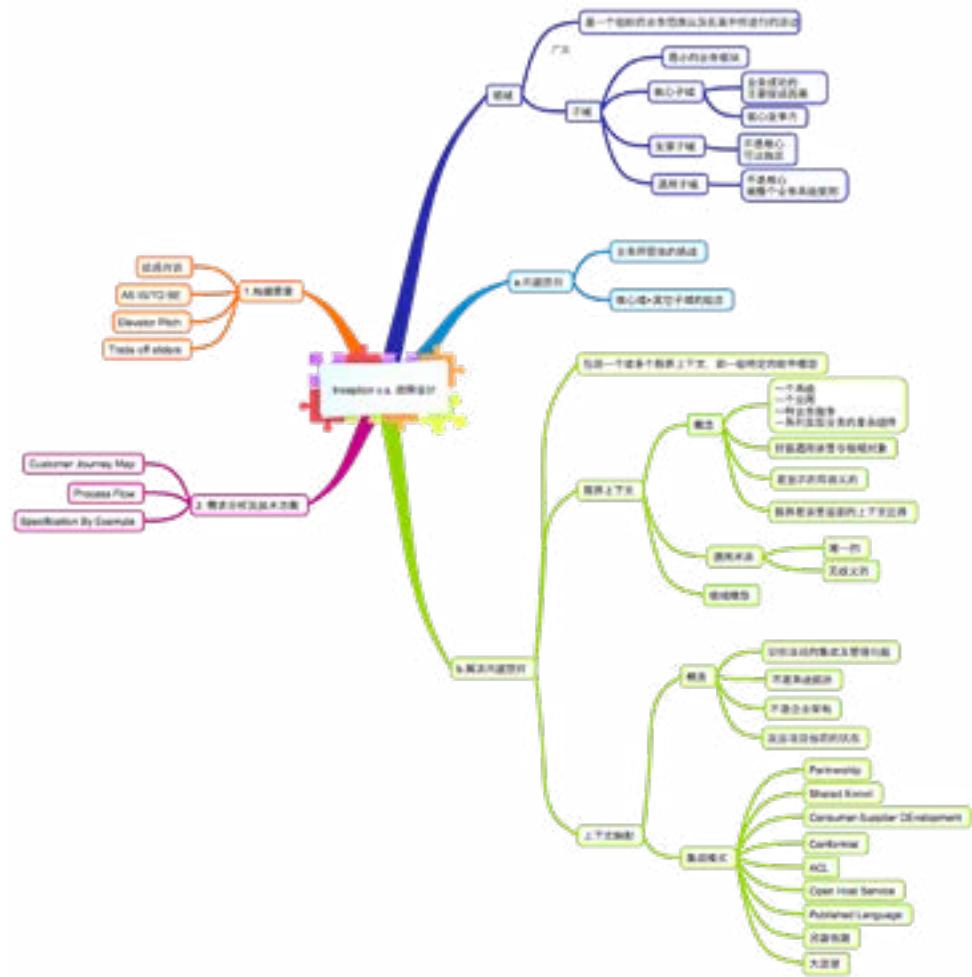
业务流程与四色建模



统一语言 领域/子域 限界上下文



Inception与DDD战略设计的对比：



一个业务领域或子域是一个企业中的业务范围以及在其中进行的活动，核心子域指业务成功的主要促成因素，是企业的核心竞争力；通用子域不是核心，但被整个业务系统所使用；支撑子域不是核心，不被整个系统使用，该能力可从外部购买。一个业务领域和子域可以包括多个业务能力，一个业务能力对应一个服务。领域的边界即限界上下文，也是服务的边界，它封装了一系列的领域模型。

一个业务流程代表了企业的一个业务领域，业务流程所涉及的数据或角色或是通用子域，或是支撑子域，由其在企业的核心竞争力的角色所决定。比如企业有统一身份认证，决策不同部门负责不同的流程任务，那么身份认证子域并不产生业务价值，不是业务成功的促成因素，但是所有流程的入口，因而为通用子域，可为单独服务；而部门负责的业务则为核心子域。

举个例子

工单业务流程：

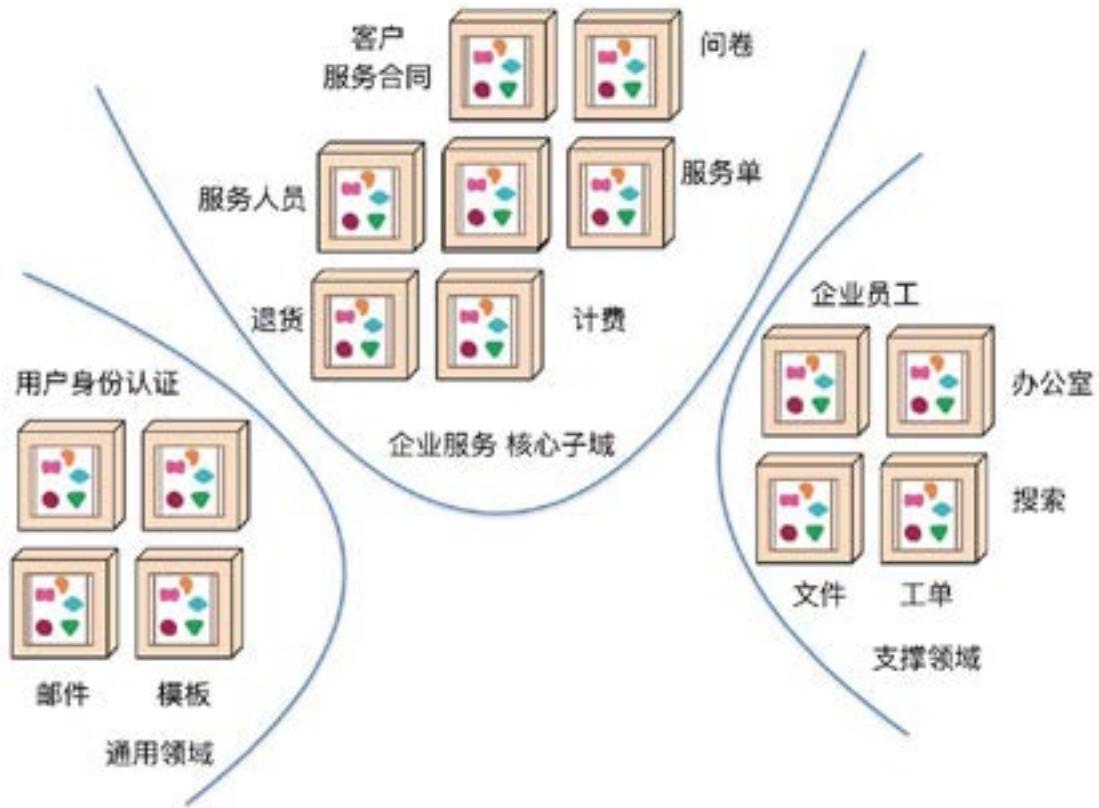
某企业为服务人员提供工单服务的业务流程简化如下。首先搜索服务人员，选取服务人员购买的服务，基于目标国家的工单流程，向服务人员收取资料，对其进行审计，最后发送结果。

## 工单服务流程



识别的领域：

其中服务为其核心竞争能力，包括该企业对全球各国的政策理解，即法律流程，服务资料（问卷），计算服务，资料审计服务，相比其他竞争对手的服务（价位/效率等），这些都为改企业提供核心的业务价值，自然也是核心子域。而其他用于统计改企业员工工作的工单，组织结构和员工为支撑子域，并不直接产生业务价值。



### 领域划分的原则

在划分的过程中，经常纠结的一个问题是：这个模型（概念或数据）看起来放这个领域合适，放另一个也合适，如何抉择呢？

- 第一，依据该模型与边界内其他模型或角色关系的紧密程度。比如，是否当该模型变化时，其他模型也需要进行变化；该数据是否通常由当前上下文中的角色在当前活动范围内使用。
- 第二，服务边界内的业务能力职责应单一，不是完成同一业务能力的模型不放在同一个上下文中。
- 第三，划分的子域和服务需满足正交原则。领域名字代表的自然语言上下文保持互相独立。
- 第四，读写分离的原则。例如报表需有单独报表子域。核心子域的划分更多基于来自业务价值的产生方，而非不产生价值的报表系统。
- 第五，模型在很多业务操作中同时被修改和更新。
- 第六，组织中业务部分的划分也是一种参考，一个业务部门的存在往往有其独特的业务价值。

简单打个比方，同一个领域上下文中的模型要保持近亲关系，五福以内，同一血统（业务）。

### 领域划分的误区和建议

- **业务能力还是计算能力？** 在划分一些貌似通用的领域时，其实只是用到了通用的计算能力而不是业务能力，只需采用通用库的方式进行封装，而无需使用服务的方式。如我们系统的模板服务，是构建通用的模板服务，服务于整个平台的服务；还是每个服务拥有独立的模板模块？
- **尽早识别剥离通用领域。** 如身份认证与鉴权领域，是企业系统中最复杂、有相对多变的领域，需要及早隔离它对核心业务的干扰。
- **时刻促成技术人员与客户、业务人员的对话。** 业务领域的划分离不开对业务意图的真正理解。而需求人员和体验设计师对于UserJourney的使用更熟悉，而技术人员、架构师对领域驱动设计、Eventstorming更熟悉。不管哪种方法都要求跨角色的群体协同工作，即客户人员、业务分析师、体验设计师与技术人员、架构师。而现实的情况中，User Journey更多的在Inception，在需求阶段进行，而领域驱动设计、Eventstorming更多的在开发设计阶段被使用，故而需求阶段经常缺失技术人员，而开发设计阶段经常缺失客户、业务人员的参与。另一个常见的现象是，Inception的参与人员和真正的开发团队有可能不是同一个群体，那么Inception中的业务沟通往往以UI的方式作为传递，因此在开发中经常只能通过UI设计来理解业务的真正意图。所以要想将正确的理解业务，做对软件，需要时刻促成技术人员与客户、业务人员的对话。

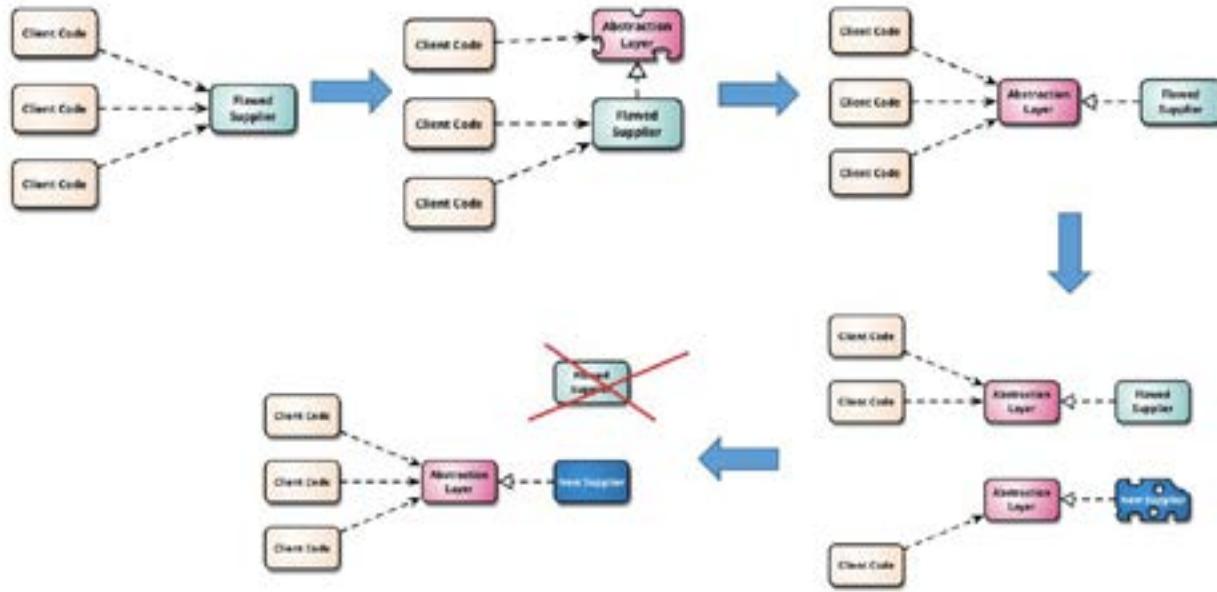
识别了被拆对象的结构和边界，下一步需要决定拆分的策略和拆分的步骤。

## 2.拆分方法与策略

拆分方法需要根据遗留系统的状态，通常分为绞杀者与修缮者两种模式。

- **绞杀者模式：** 指在遗留系统外围，将新功能用新的方式构建为新的服务。随着时间的推移，新的服务逐渐“绞杀”老的遗留系统。对于那些老旧庞大难以更改的遗留系统，推荐采用绞杀者模式。
- **修缮者模式：** 就如修房或修路一样，将老旧待修缮的部分进行隔离，用新的方式对其进行单独修复。修复的同时，需保证与其他部分仍能协同功能。

我们过去所做的拆分中多为修缮者模式，其基本原理来自Martin Fowler的branch by abstraction的重构方法，如下图所示：



就如我们团队所总结的16字重构箴言，我觉得十分的贴切：

“旧的不变，新的创建，一步切换，旧的再见”。

通过识别内部的被拆模块，对其增加接口层，将旧的引用改为新接口调用；随后将接口封装为API，并将对接口的引用改为本地API调用；最后将新服务部署为新进程，调用改为真正的服务API调用。

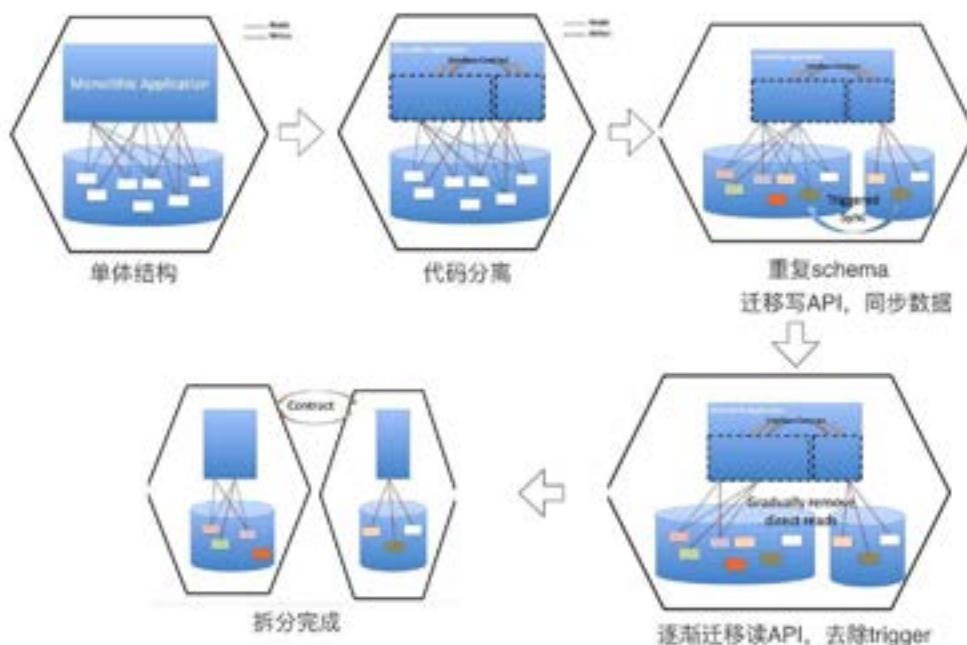
同时，拆分建议从业务相对独立、耦合度最小的地方开始。待团队获取相应经验和基础设施平台构建完善后，再进行核心应用迁移和大规模的改造。另外，核心通用服务尽量先行，如身份认证服务。

### 3. 拆分步骤

对于模块的拆分包括两部分：数据库与业务代码，可以先数据库后业务代码，亦可先业务代码后数据库。然而我们的项目拆分中遇到的最大挑战是数据层的拆分。在2015年的拆分中发现，数据库层由于当时系统性能调优的驱动，在代码中出现了跨模块的数据库连表查询。这导致后期服务的拆分非常的困难。因此在拆分步骤上我们更多的推荐数据库先行。

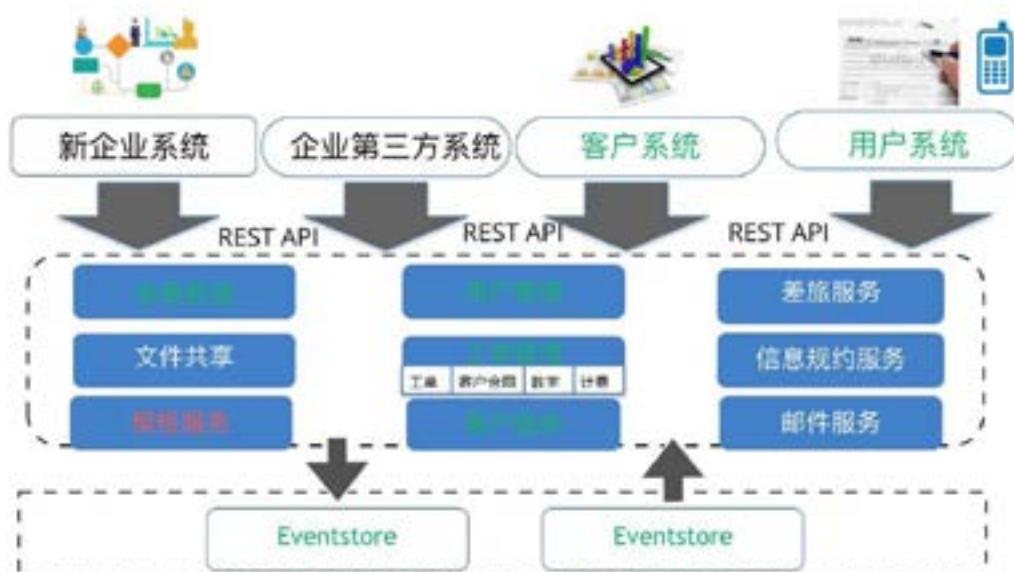
#### 4. 数据库拆分

我们借鉴了重构数据库一书中提到的方法，通过重复schema同步数据，对数据库的读写操作分别进行迁移。如下图所示：



#### 5. 我们的结果：

系统架构图：



## 问题2：拆分后业务变了增加了怎么办？

随着客户业务的变化，我们的服务也在持续的增加，而其中碰到了一个特大的服务。服务的大小如何衡量呢？该服务生产代码7万行+，测试代码14万行+，测试运行时间2个小时。团队中7个stream每天50%工作需要对这个服务进行更改，使得团队间的依赖非常严重，独立功能无法单独快速前行，交付速度及质量都受到了影响。

### 我们的总结：

客户的业务是在变化的，我们对业务的认知也是逐渐的过程，所以Martin Fowler在他的文章中提出，系统的初期建议以单体结构开始，随业务发展决定其是否被拆分或合并。那么这也意味着这样构建的服务在它的生命周期中必然会持续被拆分或合并。那么为了实现这样一个目标，使系统拥有快速的响应力，也要求这样的拆分必然是高效的低成本的。

因此，服务的设计需要满足如下的原则：

- 服务要有明确的业务边界，以单体开始并不意味着没有边界。服务要有边界，即使以单体开始也要定义单体时期的边界。我们系统中有一个名为“Monkey”的服务，是在中国虎年启动的，由此它并不是一个业务概念。当这个服务的名字为MonkeyAPI时，可以想象5年来它变成了什么？几乎所有和这个产品相关的功能都放入了这个服务中。脱离平台来看这一个产品的系统，其实它只是做了前后端分离而已。这个例子告诉我们，没有边界就会导致大杂烩，之后对其进行整理和重造的代价很大，可能需要花费“几代人”的努力。
- 服务要有明确清晰的契约设计，即对外提供的业务能力。
- 服务内部要保持高度模块化，才能够容易的被拆分。
- 可测试。

## 问题3：如何安全地持续地拆？

就如前言中提到的，系统已经上线大量的用户正在使用，如何在不影响当下系统运行状态的前提下，持续安全地演进？其实持续演进就是一场架构层次的重构，在这样的路上同样需要：

- 坏味道驱动，架构的坏味道是代码坏味道在更高层次的展现，也就意味着架构的混乱程度同样反映了该系统代码层的质量问题。
- 安全小步的重构。
- 有足够的测试进行保护——契约测试。
- 持续验证演进的方向。

## 真正有挑战的问题4: 如何保证拆对了?

拆分不能没有目标，尤其在具有风险的架构层次拆分更需谨慎。那么我们如何验证拆分的结果和收益？或许它可以提高开发效率，交付速度快，上线快，宕机时间也短，还能提高开发质量，可扩展性好，稳定，维护成本低，新人成长快，团队容易掌握等等。然而软件开发是一个复杂的事情，拆分可以引起多个维度的变化，度量的难度在于如何准确定位由拆分这一单一因素引起的价值的变化（增加或降低）。

其实要回答这个问题，还是要回到拆分之初：为什么而拆？我所见过的案例中有因为政治原因拆的、业务发展需要的、系统集成驱动的等等；有因之而成功的，也有因之而失败的。拆并不是一件容易的事，有诸多的因素。我认为不管表象是什么，拆之前需要弄清拆分的价值所在，这也是我们可以保证拆分结果的源头。

## 总结

系统可由单体结构开始，不断的演进。而团队需要对业务保持敏感，与客户、业务人员进行业务对话，不断修炼领域驱动设计和重构的能力。

在拆分的路上，我们的经验显示其最大的障碍来自意大利面一样的系统。不管我们是什么样的架构风格，高内聚低耦合的模块化代码内部质量仍然是我们架构演进的基石。具有夯实领域驱动设计和重构功底的团队才可以应对这些挑战，持续演进，保持其生命力。而架构变迁之前需要弄清背后的变动因与价值，探索性前进，及时反馈验证，才是正解。那么我们如何保证架构不被破坏呢？这个问题会在后续的文章中持续探讨。

最后，勿忘初心，且行且演进。

# 溯源微服务：企业分布式应用的一次回顾

作者：刘尚奇

微服务作为架构风格几乎成为云时代企业级应用的事实标准，构成微服务的技术元素本身却并非革命性。跨平台的分布式通信框架、地址无关的服务注册与发现、智能路由与编排等技术早已在CORBA、SOA时代实现了一遍又一遍，我们不禁好奇，微服务有什么不同？本文是对企业分布式应用的一次回顾，与前微服务时代相比，我们究竟在哪些领域吸取了教训，哪些方面持续搞砸。

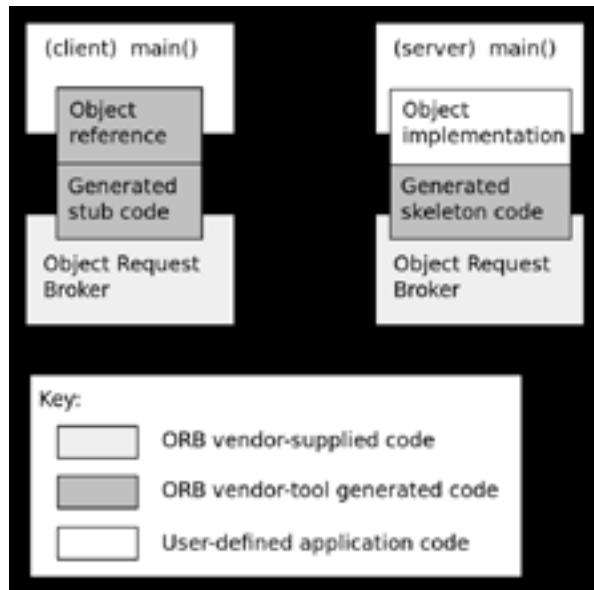
## 我们在重新界定抽象边界上取得了进展...

架构的关键在于构造合理的封装抽象。良好的抽象构造如进程，由操作系统接管CPU调度、内存地址空间分配和I/O，程序员的心智从此解放，得以聚焦在业务逻辑上。糟糕的抽象往往引向万丈深渊，大量精力被浪费在抽象泄露带来的问题上。

在分布式系统中我们关注组件、组件间的通信以及伴随的工程实践，微服务在企业应用的上下文中就技术约束和业务价值间达成了更好的平衡点。

## RPC? 不, 是API!

让我们从组件间的通信开始，最初人们认为这只是需要被解决的技术要素。



(图片来自: <https://upload.wikimedia.org/wikipedia/en/thumb/f/f0/Orb.svg/802px-Orb.svg.png>)

关于如何实现跨平台的分布式通信，30年前诞生的CORBA架构在今天来看仍然非常漂亮：通过定义IDL/ORB/API我们可以将内存对象任意分布于网络中。只要共享IDL，对象可以由C++/Java等不同的语言实现，其互相调用就像本地方法一样简单。然而实践经验告诉我们，分布式系统总是会出现本地调用不会发生的各种问题：网络的开销、传输的延迟、消息的超时和丢包、远端系统的崩溃……物理世界的技术约束是无法被忽略的，我们没有办法把分布式调用抽象成简单的

本地方法。因此Martin Fowler在他的<企业应用架构模式>里提出了著名分布式对象第一定律：“不要分布式你的对象”。相反，你应该把尽可能多的操作置于进程之内，通过replicate整个应用的方式来实现系统的scale。

由分析师们发起的SOA运动从另一个角度看待这个问题，Web Service应该是对企业资产和业务能力的封装。我们开始站在更高的维度，远过程调用不再只是技术意义上的集成。WSDL不仅是通信调用的接口，更是服务间的契约；UDDI不仅是服务描述、发现、集成的中心，更是企业业务与服务的黄页。WS-\*在厂商的裹挟下发展成包罗万象，却也没几个人能掌握。开发者们抱怨花了太多时间写冗余的XML制定所谓的规范，WSDL生成的客户端也将不同服务耦合在一起。是否有更加轻量敏捷的方式，让我们快点开始写第一行生产代码？

于是我们看到REST的兴起。起初是作为反叛，用更加轻量级的方式(http+json)使用Web。然后我们发现”企业级”应用并非需要ESB这样昂贵的专有中间件，由”消费级”技术组成的万维网是世界上最大规模的分布式网络，我们应该向其学习如何构建健壮、可演化的系统。Roy Fielding那篇论文所提出的无状态、可缓存等特征已经深入人心，而狭义上的REST API(基于资源的URI、HTTP动词和状态码的标准接口)也成为API设计的最佳实践。

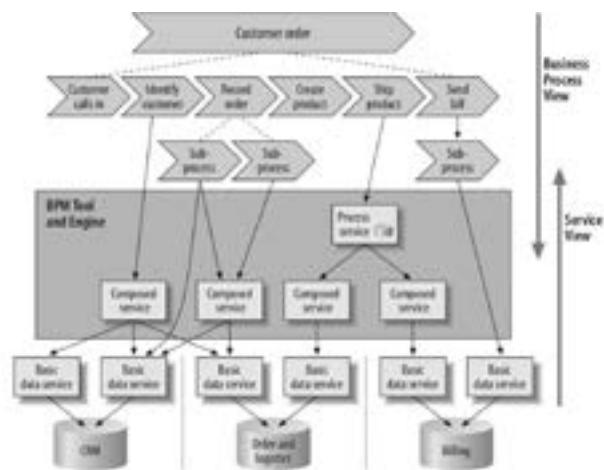
既然API和网站一样都是基于通用Web技术，API是否可以像网站一样作为产品提供呢(APIs as product)?于是越来越多的企业开始将自己的业务能力封装成API，提供给消费者，随之而来的是更弹性的商业应用和更灵活的计费方式。很多组织也着手构建自己的API市场，把内部IT能力整合、复用，并为孵化外部产品做准备。API已经成为商业价值主张的一部分。

我们从聚焦实现细节的rpc出发，来到了更具价值导向的REST API。即使构建内部系统，以消费者驱动的方式，也总是能帮助我们设计出更加松耦合和易于演进的API。

## 技术组件？不，是业务服务！

编程语言中的组件构造(如Java中的jar, C#中的dll)是软件架构师们封装可复用单元的最常用武器。组件作为理论上的最小部署单元，在工程实践中却不容易独立变更。一般应用程序需要将多个组件打包成一个部署单元(如war包)，链接在内存地址中进行调用。对单个组件的热更新往往对组件间耦合和对象状态管理有很高的要求，重新部署整个应用一般是默认选项。以进程为边界构建可独立部署的服务成为架构师的另一项选择。

早期的服务只是单纯的技术构件，大多数组织从纯粹的技术实现角度考虑服务的划分。SOA的推动者们指出企业的信息资产应该被复用，信息孤岛应该被打通。通过将不同的服务编排组合，我们应该能够实现IT对业务更加灵活的支撑。

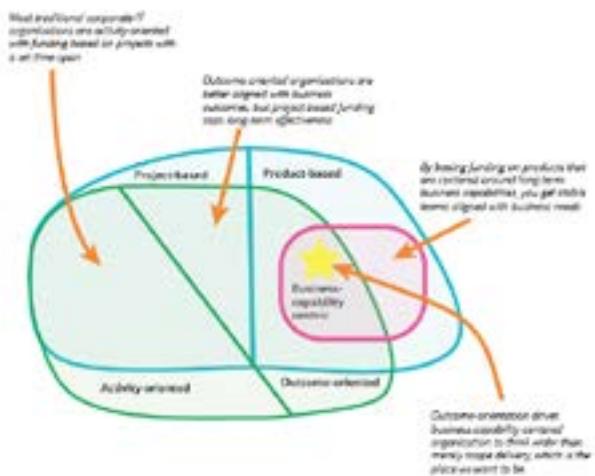


(图片来自: OSOA in practice, Nicolai Josuttism, 2009)

SOA的服务建模一般采用业务流程驱动的方式。一个典型的SOA设计是由业务分析师自顶向下地对企业

现有业务流程进行分析，通过BPM引擎对流程进行建模，向下分解成组合服务，并进一步拆分成数据访问服务(很多可怜的SOA实现中数据的访问被拆分成不同的读服务和写服务)。然而这带来的问题是，服务跟服务间的耦合非常严重。当我的业务发生了变化，可能会需要修改很多不同的服务，涉及到多个团队的沟通和协调。在运行时层面，服务器间的通信非常频繁，用户在界面上的一次点击按钮，对应的后台多层服务间的级联通信。这给系统性能和稳定性也带来了巨大的挑战。SOA式的服务建模从分析型思维出发，却往往低估了分布式系统和跨团队协调的复杂度，导致服务拆分粒度过细。

微服务的名字常常让人误解，但实施正确的微服务粒度可能并不”微”。Martin Fowler与James Lewis在开创微服务定义的一文中已经指出微服务应该围绕完整的业务能力。今天我们在做微服务设计时，常常利用领域驱动设计中的Bounded Context来进行服务边界的划分。假设你的库存管理是一个独立的业务子域，针对库存的维护和操作应该被放到通过一个上下文和微服务中，由一个团队进行开发维护。多数业务变更都发生在上下文内部，不涉及跨团队协调。单个 codebase 内的重构和部署让发布更加容易。维护库存所需要的信息查询的调用多发生在进程中，更好的性能，同时无需处理额外的一致性问题。



微服务的另一个特点在于Product over Project, 这需要不同于传统投资组合的预算管理与团队组建。传统的项目制将预算分配在相对短期的服务开发过程中，项目团队关注的是如何将业务范围(scope)实现，开发结束后服务转交运维团队进行维护，项目团队则被解散进行其他项目的开发。将微服务作为产品运营则需要建立业务结果导向的稳定产品团队。服务的设计不只聚焦于当下需求，更需要考虑价值定位和产品愿景。工程团队则需要思考如何用有限成本支撑非线性的业务接入增长。.

<h2>Coordination</h2> <ul style="list-style-type: none"><li>• Shared customers, products, or suppliers</li><li>• Impact on other business unit transactions</li><li>• Operationally unique business units or functions</li><li>• Autonomous business management</li><li>• Business unit control over business process design</li><li>• Shared customer/supplier/product data</li><li>• Consensus processes for designing IT infrastructure services; IT application decisions made in business units</li></ul>	<h2>Unification</h2> <ul style="list-style-type: none"><li>• Customers and suppliers may be local or global</li><li>• Globally integrated business processes often with support of enterprise systems</li><li>• Business units with similar or overlapping operations</li><li>• Centralized management often applying functional/process/business unit matrices</li><li>• High-level process owners design standardized processes</li><li>• Centrally mandated databases</li><li>• IT decisions made centrally</li></ul>
<h2>Diversification</h2> <ul style="list-style-type: none"><li>• Few, if any, shared customers or suppliers</li><li>• Independent transactions</li><li>• Operationally unique business units</li><li>• Autonomous business management</li><li>• Business unit control over business process design</li><li>• Few data standards across business units</li><li>• Most IT decisions made within business units</li></ul>	<h2>Replication</h2> <ul style="list-style-type: none"><li>• Few, if any, shared customers</li><li>• Independent transactions aggregated at a high level</li><li>• Operationally similar business units</li><li>• Autonomous business unit leaders with limited discretion over processes</li><li>• Centralized (or federal) control over business process design</li><li>• Standardized data definitions but data locally owned with some aggregation at corporate</li><li>• Centrally mandated IT services</li></ul>

(图片来自: Enterprise Architecture as Strategy, Ross et al, 2006)

如今我们对服务的定义已经超越了技术组件，领先的组织已经在尝试将design thinking, business operating model应用到微服务设计中。

# 解耦服务就足够了吗? 我们需要去中心化一切!

即使有了设计合理的服务于API, 我们仍然需要与之匹配的工程实践才能将其顺利实施。

今天仍有很多企业使用集中式的应用服务器部署应用: 开发团队将软件包构建出来, 再统一安装到应用服务器中。对应用团队来说, 这往往意味着漫长的反馈周期和痛苦的自动化。我们很早就推荐用Jetty这样内嵌式的应用容器部署软件, 启动更快, 测试环境更接近生产。one Tomcat per VM的部署方式虽然运行时开销较大, 却是前容器时代隔离性最好的服务部署模式。Docker将这个实践更进一步, 除了更轻量级的隔离, 我们第一次可以将软件和所依赖的环境本身打包成版本化的artifact, 彻底统一开发和生产环境。容器技术的成熟让我们可以将部署去中心化, 开发团队可以独立部署一个服务。

数据库耦合是影响服务独立变更的另一重要因素。相比代码构成的应用软件, 数据库schema更加难以变动。因为难以测试、难以兼顾性能优化和耦合的发布周期等因素, 服务间以数据库集成成为臭名昭著的反模式。服务间的集成应该依赖封装好的显示接口, 而不是数据库这种实现细节。我们应该在兼顾数据一致性的情况下, 为每个微服务分配独立的db schema甚至db instance。如果说十年前数据几乎等同于关系数据库。如今数据则可能呈现出各种形态: 键值、文档、时间序列、图...我们完全可以采用更加合适的技术, 以去中心化的方式进行微服务的数据治理。

即使将这一切都解耦, 如果将交给一个集中的团队去实施, 很有可能最终还是得到一个耦合的架构。这就是著名的康威定律。康威定律告诉我们“设计系统的架构受制于产生这些设计的组织的沟通结构”。但同样我们可以将康威定律反转应用: 如果你想达成一个目标架构, 则必须对团队结构进行调整, 使之和目标架构对齐。相比单体系统, 微服务在运行时监控和运维所带来的挑战更大。”you build it, you run it”的DevOps文化成为必须。监控运维不再是Ops部门的事情, 产品团队必须对微服务的整个生命周期负责。授权的去中心化自治团队是实施微服务的必要条件。

## 我们干得还不错, 但也在持续搞砸一些事情...

我们在很多方向的确取得了进展。但即使在微服务时代, 很多问题仍然在轮回发生着, 似乎我们总是无法吸取历史的教训。让我们看一看那些挥之不去的反模式阴云。

一个例子是开发者对强类型RPC代码生成的依恋。尽管历史经验已经证明同步的rpc无法为分布式通信提供足够好的封装, 伪装成本地方法调用的客户端往往鼓励程序员做出糟糕的接口设计: 细粒度的频繁调用、缺少缓存和容错处理。IDL生成客户端也会导致服务间耦合, 每次变更接口都需要升级数个相关服务。如果用可演进的REST API(如HATEOS)和tolerant reader模式, 则可以优雅地解决这个问题。然而新一代的开发者们还是经常“重新”发现rpc的这些能力并陷入依赖——更快的序列化反序列化、类型安全和来自IDE的智能提示、通过spec反向生成代码...分布式计算先驱Vinoski不禁感叹“开发人员的便利性是否真的胜过正确性, 可扩展性, 性能, 关注点分离, 可扩展性和意外复杂性?”

另一个挥之不去的阴影是ESB。ESB在将异构的应用wire在一起有着关键的作用。然而当越来越多的职责被加入: 数据报文的裁剪转换、难以测试和版本控制的编排(orchection)逻辑、服务发现智能路由监控治理分布式事务等

All in One的solution将ESB变成了一个可怕的单点梦魇。所以微服务发出了“智能终端哑管道”的呐喊：我们只是需要一个不那么智能的代理处理可靠消息传输，将灵活的逻辑交给服务本身去编配(choreography)吧。

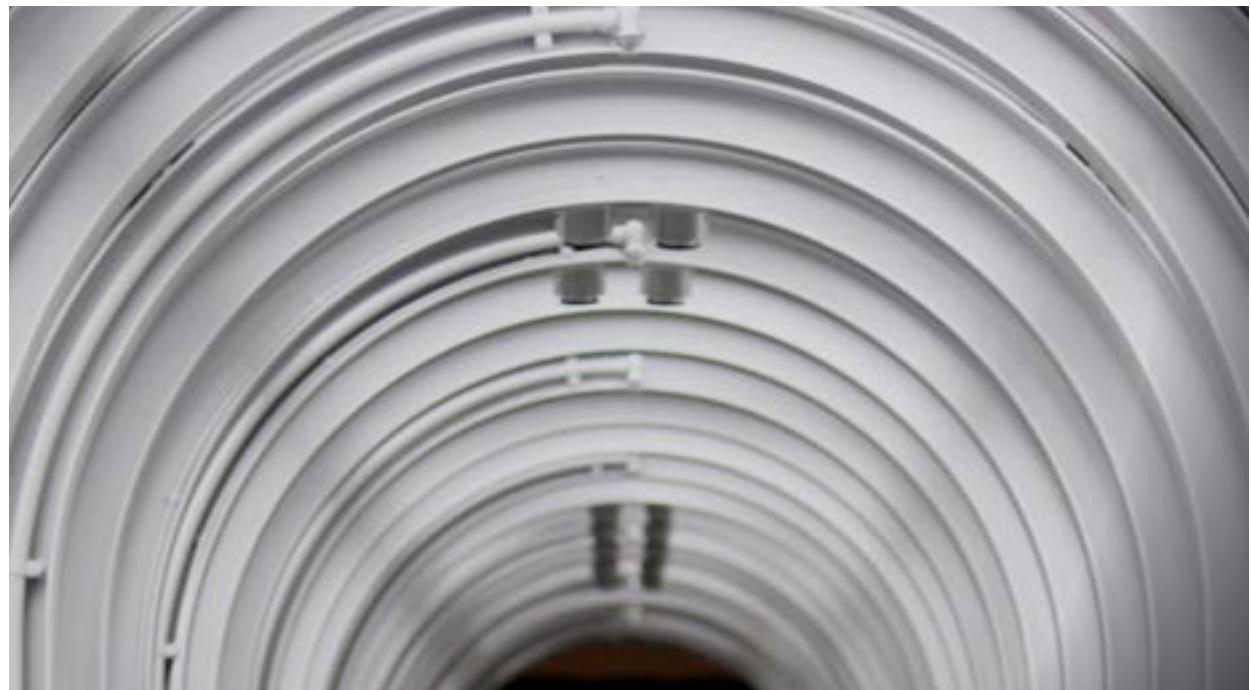
于是在典型的微服务架构里，负载均衡、服务注册发现、分布式追踪等组件以Unix way的方式各司其职。然而在利益诱惑和特性竞争压力之下，很多厂商不断将更多的功能放进他们的中间件，其中为代表的Overambitious API gateways俨然要重新实现占据中心的ESB。如果API gateway只是处理鉴权、限流等横切层逻辑没有问题，如果API gateway开始处理数据转换和业务逻辑编排，你应该提高警惕！

尽管行业在不断发展，但很多时候人们仍然沿用旧的思维，用新的技术去一遍遍重新实现这些旧的反模式。

## 如何更进一步

你总是可以在技术雷达里追踪微服务的state of art，如今这个领域的前沿方向是什么，Service Mesh, Chaos Engineering, 还是Observability as Code? 然而历史告诉我们，新的技术在解决一些问题的同时，也可能会产生新的问题。更糟糕的是，我们永远无法记住历史，用新的工具更高效地重现旧日问题。

Technologies come and go, Principles stay forever。好在那些架构和实践背后的原则是经久不变的。从操作系统到移动应用都会需要高内聚低耦合的架构，任何软件开发都需要版本控制、自动化构建等实践。谨记这些核心原则、谨记软件被创造出来是为了解决有价值的问题，可以帮我们更好的借鉴历史的经验，理解和采纳新的技术。





ThoughtWorks洞见  
领域驱动设计

---

# 示例实现

# 后端开发实践系列——开发者的第0个迭代

作者: 滕云



在ThoughtWorks, 我从零开始搭建了不少软件项目, 其中包含了基础代码框架和持续集成基础设施等, 这些内容在敏捷开发中通常被称为“第0个迭代”要做的事情。但是, 当项目运行了一段时间之后再来反观, 我总会发现一些不足的地方, 要么测试分类没有分好, 要么基本的编码架子没有考虑周全。

另外, 我在工作中也会接触到很多既有项目, 公司内部和外部的都有, 多数项目的编码实践我都是不满意的。比如, 我曾经新加入一个项目的时候, 前前后后请教了3位同事才把该项目在本地运行起来; 又比如在另一项目中, 我发现前端请求对应的Java类命名规范不统一, 有被后缀为Request的, 也有被后缀为Command的。

再者, 工作了这么多年之后, 我越来越发现基础知识以及系统性学习的重要性。诚然, 技术框架的发展使得我们可以快速地实现业务功能, 但是当软件出了问题之后有时却需要将各方面的知识融会贯通并在大脑里综合反应才能找到解决思路。

基于以上, 我希望整理出一套公共性的项目模板出来, 旨在尽量多地包含日常开发之所需, 减少开发者的重复性工作以及提供一些最佳实践。对于后端开发而

言, 我选择了当前被行业大量使用的Spring Boot, 基于此整理出了一套公共的、基础性的实践方式, 在结合了自己的经验以及其他项目的优秀实践之后, 总结出本文以飨开发者。

本文以一个简单的电商订单系统为例, 源代码请访问:

<https://github.com/e-commerce-sample/order-backend>

所使用的技术栈主要包括: Spring Boot、Gradle、Junit 5、Rest Assured、Docker等。

## 第一步: 从写好README开始

一份好的 README 可以给人以项目全景概览, 可以使新人快速上手项目, 可以降低沟通成本。同时, README 应该简明扼要, 条理清晰, 建议包含以下方面:



- **项目简介:** 用一两句话简单描述该项目所实现的业务功能；
- **技术选型:** 列出项目的技术栈，包括语言、框架和中间件等；
- **本地构建:** 列出本地开发过程中所用到的工具命令；
- **领域模型:** 核心的领域概念，比如对于示例电商系统来说有Order、Product等；
- **测试策略:** 自动化测试如何分类，哪些必须写测试，哪些没有必要写测试；
- **技术架构:** 技术架构图；
- **部署架构:** 部署架构图；
- **外部依赖:** 项目运行时所依赖的外部集成方，比如订单系统会依赖于会员系统；
- **环境信息:** 各个环境的访问方式，数据库连接等；

- **编码实践:** 统一的编码实践，比如异常处理原则、分页封装等；
- **FAQ:** 开发过程中常见问题的解答。

需要注意的是，README中的信息可能随着项目的演进而改变（比如引入了新的技术栈或者加入了新的领域模型），因此也是需要持续更新的。虽然我们知道，软件文档的一个痛点便是无法与项目实际进展保持同步，但是就README这点信息来讲，还是建议开发者们不要吝啬那一点点敲键盘的时间。

此外，除了保持README的持续更新，一些重要的架构决定可以通过示例代码的形式记录在代码库中，新开发者可以通过直接阅读这些示例代码快速了解项目的通用实践方式以及架构选择，请参考ThoughtWorks的技术雷达。

## 一键式本地构建

为了避免诸如前文中所提到的“请教了3位同事才本地构建成功”的尴尬，为了减少“懒惰”的程序员们的手动操作，也为了为所有开发者提供一种一致的开发体验，我们希望用一个命令就可以完成所有的事情。这里，对于不同的场景我总结出了以下命令：

- 生成IDE工程: **idea.sh**, 生成IntelliJ工程文件并自动打开IntelliJ。
- 本地运行: **run.sh**, 本地启动项目, 自动启动本地数据库, 监听调试端口5005。
- 本地构建: **local-build.sh**, 只有本地构建成功才能提交代码。
- 以上3个命令基本上可以完成日常开发之所需, 此时, 对于新人的开发流程。

以上3个命令基本上可以完成日常开发之所需, 此时, 对于新人的开发流程大致为:

- 1. 拉取代码;
- 2. 运行**idea.sh**, 自动打开IntelliJ;
- 3. 编写代码, 包含业务代码和自动化测试;
- 4. 运行**run.sh**, 进行本地调试或必要的手动测试(本步骤不是必需);
- 5. 运行**local-build.sh**, 完成本地构建;
- 6. 再次拉取代码, 保证**local-build.sh**成功, 提交代码。

事实上, 这些命令脚本的内容非常简单, 比如**run.sh**文件内容为:

```
#!/usr/bin/env bash  
./gradlew clean bootRun
```

然而，这种显式化的命令却可以减少新人的恐惧感，因为他们只需要知道运行这3个命令就可以搞开发了。另外，一个小小的细节：本地构建的`local-build.sh`命令本来可以重命名为更简单的`build.sh`，但是当我们在命令行中使用Tab键自动补全的时候，会发现自动补全到了`build`目录，而不是`build.sh`命令，并不方便，因此命名为了`local-build.sh`。细节虽小，但是却体现了一个宗旨，即我们希望给开发者一种极简的开发体验，我把这些看似微不足道的东西称作是对程序员的“人文关怀”。

## 目录结构

Maven所提倡的目录结构当前已经成为事实上的行业标准，Gradle在默认情况下也采用了Maven的目录结构，这对于多数项目来说已经足够了。此外，除了Java代码，项目中还存在其他类型的文件，比如Gradle插件的配置、工具脚本和部署配置等。无论如何，项目目录结构的原则是简单而有条理，不要随意地增加多余的文件夹，并且也需要及时重构。

在示例项目中，顶层只有2个文件夹，一个是用于放置Java源代码和项目配置的`src`文件夹，另一个是用于放置所有Gradle配置的`gradle`文件夹，此外，为了方便开发人员使用，将上文提到的3个常用脚本直接放到根目录下：

```
└—— order-backend
    ├── gradle // 文件夹，用于放置所有Gradle配置
    ├── src // 文件夹，Java源代码
    ├── idea.sh // 生成IntelliJ工程
    ├── local-build.sh // 提交之前的本地构建
    └—— run.sh // 本地运行
```

对于`gradle`而言，我们刻意地将Gradle插件脚本与插件配置放到了一起，比如Checkstyle：

```
├—— gradle
|   ├── checkstyle
|   |   ├── checkstyle.gradle
|   |   └—— checkstyle.xml
```

事实上，在默认情况下Checkstyle插件会从项目根目录下的**config**目录查找**checkstyle.xml**配置文件，但是这一方面增加了多余的文件夹，另一方面与该插件相关的设施分散在了不同的地方，违背了广义上的内聚原则。

## 基于业务分包

早年的Java分包方式通常是基于技术的，比如与domain包平级的有controller包、service包和infrastructure包等。这种方式当前并不被行业所推崇，而是应该首先基于业务分包。比如，在订单示例项目中，有两个重要的领域对象**Order**和**Product**（在DDD中称为聚合根），所有的业务都围绕它们展开，因此分别创建order包和product包，再分别在包下创建与之相关的各个子包。此时的order包如下：

```
|--- order
|   |--- OrderApplicationService.java
|   |--- OrderController.java
|   |--- OrderNotFoundException.java
|   |--- OrderRepository.java
|   |--- OrderService.java
|   |--- model
|   |   |--- Order.java
|   |   |--- OrderFactory.java
|   |   |--- OrderId.java
|   |   |--- OrderItem.java
|   |   |--- OrderStatus.java
```

可以看到，在order包下我们直接放置了**OrderController**和**OrderRepository**等类，而没有必要再为这些类划分单独的子包。而对于领域模型Order来讲，由于包含了多个对象，因此基于内聚性原则将它们归到model包中。但是这并不是一个必须，如果业务足够简单，我们甚至可以将所有类直接放到业务包下，product包便是如此：

```
└── product
    ├── Product.java
    ├── ProductApplicationService.java
    ├── ProductController.java
    ├── ProductId.java
    └── ProductRepository.java
```

在编码实践中，我们总是基于一个业务用例来实现代码，在技术分包场景下，我们需要在分散的各包中来回切换，增加了代码导航的成本；另外，代码提交的变更内容也是散落的，在查看代码提交历史时，无法直观的看出该次提交是关于什么业务功能的。在业务分包下，我们只需要在单个统一的包下修改代码，减少了代码导航成本；另外一个好处是，如果哪天我们需要将某个业务迁移到另外的项目（比如识别出了独立的微服务），那么直接整体移动业务包即可。

当然，基于业务分包并不意味着所有的代码都必须囿于业务包下，这里的逻辑是：优先进行业务分包，然后对于一些不隶属于任何业务的代码可以单独分包，比如一些util类、公共配置等。比如我们依然可以创建一个common包，下面放置了Spring公共配置、异常处理框架和日志等子包：

```
└── common
    ├── configuration
    ├── exception
    ├── loggin
    └── utils
```

## 自动化测试分类

在当前的微服务和前后端分离的开发模式下，后端项目仅提供纯粹的业务API，而不包含UI逻辑，因此后端项目不会再包含诸如WebDriver的重量级端到端测试。同时，后端项目作为向外提供业务功能的独立运行单元，在API级别也应该有相应的测试。

此外，程序中有些框架性代码，要么是诸如Controller之类的技术性框架代码，要么是基于某种架构风格的代码（比如DDD实践中的ApplicationService），这些代码一方面并不包含业务逻辑，一方面是很薄的一个抽象层（即实现相对简单），用单元测试来覆盖显得没有必要，因此笔者的观点是可以不为此编写单独的单元测试。再者，程序中有些重要的组件性代码，比如访问数据库的Repository或者分布式锁，使用单元测试实际上“测不到点上”，而使用API测试又显得在分类逻辑上不合理，为此我们可以专门创建一种测试类型谓之组件测试。

基于以上，我们可以对自动化测试做个分类：

- 单元测试：核心的领域模型，包括领域对象（比如Order类），Factory类，领域服务类等；
- 组件测试：不适合写单元测试但是又必须测试的类，比如Repository类，在有些项目中，这种类型测试也被称为集成测试；
- API测试：模拟客户端测试各个API接口，需要启动程序。

Gradle在默认情况下只提供`src/test/java`目录用于测试，对于以上3种类型的测试，我们需要将它们分开以便于管理（也是职责分离的体现）。为此，可以通过Gradle提供的SourceSets对测试代码进行分类：

```
sourceSets {  
    componentTest {  
        compileClasspath += sourceSets.main.output + sourceSets.test.output  
        runtimeClasspath += sourceSets.main.output + sourceSets.test.output  
    }  
  
    apiTest {  
        compileClasspath += sourceSets.main.output + sourceSets.test.output  
        runtimeClasspath += sourceSets.main.output + sourceSets.test.output  
    }  
}
```

到此，3种类型的测试可以分别编写在以下目录：

- 单元测试: `src/test/java`
- 组件测试: `src/componentTest/java`
- API测试: `src/apiTest/java`

需要注意的是，这里的API测试更多强调的是对业务功能的测试，有些项目中可能还会存在契约测试和安全测试等，虽然从技术上讲都是对API的访问，但是这些测试都是单独的关注点，因此建议分开对待。

值得一提的是，由于组件测试和API测试需要启动程序，也即需要准备好本地数据库，我们采用了Gradle的docker-compose插件(或者jib插件)，该插件会在运行测试之前自动运行Docker容器(比如MySQL)：

```
apply plugin: 'docker-compose'

dockerCompose {
    useComposeFiles = [ 'docker/mysql/docker-compose.yml' ]
}

bootRun.dependsOn composeUp
componentTest.dependsOn composeUp
apiTest.dependsOn composeUp
```

更多的测试分类配置细节，比如JaCoCo测试覆盖率配置等，请参考本文的示例项目代码。对Gradle不熟悉的读者可以参考笔者的Gradle学习系列文章。

# 日志处理

在日志处理中，除了完成基本配置外，还有2个需要考虑的点：

- 在日志中加入请求标识，便于链路追踪。在处理一个请求的过程中有时会输出多条日志，如果每条日志都共享统一的请求ID，那么在日志追踪时会更加方便。此时，可以使用Logback原生提供的MDC(Mapped Diagnostic Context)功能，创建一个RequestIdMdcFilter：

```
protected void doFilterInternal(HttpServletRequest request,
                               HttpServletResponse response,
                               FilterChain filterChain)
        throws ServletException, IOException {
    //request id in header may come from Gateway, eg. Nginx
    String headerRequestId = request.getHeader(H HEADER_X_REQUEST_ID);
    MDC.put(REQUEST_ID, isNullOrEmpty(headerRequestId) ? newUuid() : headerRequestId);
    try {
        filterChain.doFilter(request, response);
    } finally {
        clearMdc();
    }
}
```

- 集中式日志管理，在多节点部署的场景下，各个节点的日志是分散的，为此可以引入诸如ELK之类的工具将日志统一输出到ElasticSearch中。本文的示例项目使用了RedisAppender将日志输出到Logstash：

```
<appender name="REDIS" class="com.cwbase.logback.RedisAppender">
    <tags>ecommerce-order-backend-${ACTIVE_PROFILE}</tags>
    <host>elk.yourdomain.com</host>
    <port>6379</port>
    <password>whatever</password>
    <key>ecommerce-ordder-log</key>
    <mdc>true</mdc>
    <type>redis</type>
</appender>
```

当然，统一日志的方案还有很多，比如Splunk和Graylog等。

## 异常处理

在设计异常处理的框架时，需要考虑以下几点：

- 向客户端提供格式统一的异常返回。
- 异常信息中应该包含足够多的上下文信息，最好是结构化的数据以便于客户端解析。
- 不同类型的异常应该包含唯一标识，以便客户端精识别。

异常处理通常有两种形式，一种是层级式的，即每种具体的异常都对应了一个异常类，这些类最终继承自某个父异常；另一种是单一式的，即整个程序中只有一个异常类，再以一个字段来区分不同的异常场景。层级式异常的好处是能够显式化异常含义，但是如果层级设计不好可能导致整个程序中充斥着大量的异常类；单一式的好处是简单，而其缺点在于表意性不够。

本文的示例项目使用了层级式异常，所有异常都继承自一个AppException：

```
public abstract class AppException extends RuntimeException {  
    private final ErrorCode code;  
    private final Map<String, Object> data = newHashMap();  
}
```

这里，**ErrorCode**枚举中包含了异常的唯一标识、HTTP状态码以及错误信息；而data字段表示各个异常的上下文信息。在示例系统中，在没有找到订单时抛出异常：

```
public class OrderNotFoundException extends AppException {  
    public OrderNotFoundException(OrderId orderId) {  
        super(ErrorCode.ORDER_NOT_FOUND, ImmutableMap.of("orderId", orderId.toString()));  
    }  
}
```

在返回异常给客户端时，通过一个ErrorDetail类来统一异常格式：

```
public final class ErrorDetail {  
    private final ErrorCode code;  
    private final int status;  
    private final String message;  
    private final String path;  
    private final Instant timestamp;  
    private final Map<String, Object> data = newHashMap();  
}
```

最终返回客户端的数据为：

```
{  
    requestId: "d008ef46bb4f4cf19c9081ad50df33bd",  
    error: {  
        code: "ORDER_NOT_FOUND",  
        status: 404,  
        message: "没有找到订单",  
        path: "/order",  
        timestamp: 1555031270087,  
        data: {  
            orderId: "123456789"  
        }  
    }  
}
```

可以看到，**ORDER\_NOT\_FOUND**与**data**中的数据结构是一一对应的，也即对于客户端来讲，如果发现了**ORDER\_NOT\_FOUND**，那么便可确定**data**中一定存在**orderId**字段，进而完成精确的结构化解析。

## 后台任务与分布式锁

除了即时完成客户端的请求外，系统中通常会有一些定时性的例行任务，比如定期地向用户发送邮件或者运行数据报表等；另外，有时从设计上我们会对请求进行异步化处理。此时，我们需要搭建后台任务相关基础设施。Spring原生提供了任务处理(TaskExecutor)和任务计划(TaskScheduler)机制；而在分布式场景下，还需要引入分布式锁来解决并发冲突，为此我们引入一个轻量级的分布式锁框架ShedLock。

启用Spring任务配置如下：

```
@Configuration  
@EnableAsync  
@EnableScheduling  
public class SchedulingConfiguration implements SchedulingConfigurer {  
  
    @Override  
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {  
        taskRegistrar.setScheduler(new ScheduledThreadPool(10));  
    }  
  
    @Bean(destroyMethod = "shutdown")  
    @Primary  
    public TaskExecutor taskExecutor() {  
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
        executor.setCorePoolSize(2);  
        executor.setMaxPoolSize(5);  
        executor.setQueueCapacity(10);  
        executor.setTaskDecorator(new LogbackMdcTaskDecorator());  
        executor.initialize();  
        return executor;  
    }  
}
```

然后配置Shedlock:

```
@Configuration  
@EnableSchedulerLock(defaultLockAtMostFor = "PT30S")  
public class DistributedLockConfiguration {  
  
    @Bean  
    public LockProvider lockProvider(DataSource dataSource) {  
        return new JdbcTemplateLockProvider(dataSource);  
    }  
  
    @Bean  
    public DistributedLockExecutor distributedLockExecutor(LockProvider lockProvider) {  
        return new DistributedLockExecutor(lockProvider);  
    }  
}
```

实现后台任务处理:

```
@Scheduled(cron = "0 0/1 * * * ?")  
@SchedulerLock(name = "scheduledTask", lockAtMostFor = THIRTY_MIN, lockAtLeastFor =  
ONE_MIN)  
public void run() {  
    logger.info("Run scheduled task.");  
}
```

为了支持代码直接调用分布式锁，基于Shedlock的LockProvider创建DistributedLockExecutor：

```
public class DistributedLockExecutor {  
    private final LockProvider lockProvider;  
  
    public DistributedLockExecutor(LockProvider lockProvider) {  
        this.lockProvider = lockProvider;  
    }  
  
    public <T> T executeWithLock(Supplier<T> supplier, LockConfiguration configuration) {  
        Optional<SimpleLock> lock = lockProvider.lock(configuration);  
        if (!lock.isPresent()) {  
            throw new LockAlreadyOccupiedException(configuration.getName());  
        }  
  
        try {  
            return supplier.get();  
        } finally {  
            lock.get().unlock();  
        }  
    }  
}
```

使用时在代码中直接调用：

```
public String doBusiness() {  
    return distributedLockExecutor.executeWithLock(() -> "Hello World." ,  
        new LockConfiguration("key", Instant.now().plusSeconds(60)));  
}
```

本文的示例项目使用了基于JDBC的分布式锁，事实上任何提供原子操作的机制都可用于分布式锁，Shedlock还提供基于Redis、ZooKeeper和Hazelcast等的分布式锁实现机制。

## 统一代码风格

除了Checkstyle统一代码格式之外，项目中有些通用的公共的编码实践方式也需要在整个开发团队中进行统一，包括但不限于以下方面：

- 客户端的请求数据类统一使用相同后缀，比如Command。
- 返回给客户端的数据统一使用相同后缀，比如Representation。
- 统一对请求处理的流程框架，比如采用传统的3层架构或者DDD战术模式。
- 提供一致的异常返回（请参考“异常处理”小节）。
- 提供统一的分页结构类。
- 明确测试分类以及统一的测试基础类（请参考“自动化测试分类”小节）。

## 静态代码检查

静态代码检查主要包含以下Gradle插件，具体配置请参考本文示例代码：

- Checkstyle：用于检查代码格式，规范编码风格
- Spotbugs：Findbugs的继承者
- Dependency check：OWASP提供的Java类库安全性检查
- Sonar：用于代码持续改进的跟踪

# 健康检查

健康检查主要用于以下场景：

- 我们希望初步检查程序是否运行正常
- 有些负载均衡软件会通过一个健康检查URL判断节点的可达性

此时，可以实现一个简单的API接口，该接口不受权限管控，可以公开访问。如果该接口返回HTTP的200状态码，便可初步认为程序运行正常。此外，我们还可以在该API中加入一些额外的信息，比如提交版本号、构建时间、部署时间等。

启动本文的示例项目：

```
./run.sh
```

然后访问健康检查API：<http://localhost:8080/about>，结果如下：

```
{
  requestId: "698c8d29add54e24a3d435e2c749ea00",
  buildNumber: "unknown",
  buildTime: "unknown",
  deployTime: "2019-04-11T13:05:46.901+08:00[Asia/Shanghai]",
  gitRevision: "unknown",
  gitBranch: "unknown",
  environment: "[local]"
}
```

以上接口在示例项目中用了一个简单的Controller实现，事实上Spring Boot的Acuator框架也能够提供相似的功能。

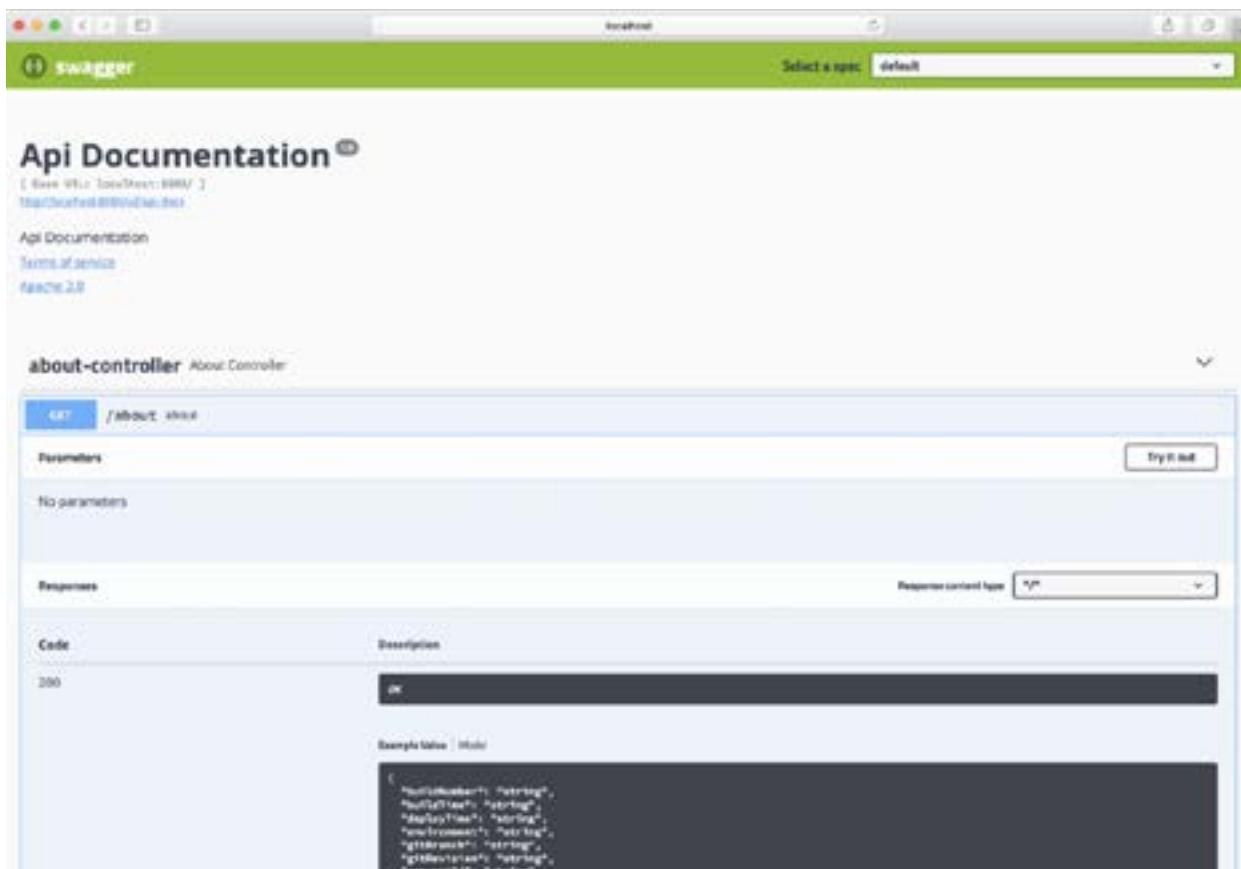
# API文档

软件文档的难点不在于写，而在于维护。多少次，当我对照着项目文档一步一步往下走时，总得不到正确的结果，问了同事之后得到回复“哦，那个已经过时了”。本文示例项目所采用的Swagger在一定程度上降低了API维护的成本，因为Swagger能自动识别代码中的方法参数、返回对象和URL等信息，然后自动地实时地创建出API文档。

配置Swagger如下：

```
@Configuration  
@EnableSwagger2  
@Profile(value = { "local", "dev" })  
public class SwaggerConfiguration {  
  
    @Bean  
    public Docket api() {  
        return new Docket(SWAGGER_2)  
            .select()  
            .apis(basePackage( "com.ecommerce.order" ))  
            .paths(any())  
            .build();  
    }  
}
```

启动本地项目，访问<http://localhost:8080/swagger-ui.html>：



## 数据库迁移

在传统的开发模式中，数据库由专门的运维团队或者DBA来维护，要对数据库进行修改需要向DBA申请，告之迁移内容，最后由DBA负责数据库变更实施。在持续交付和DevOps运动中，这些工作逐步提前到开发过程，当然并不是说不需要DBA了，而是这些工作可以由开发者和运维人员一同完成。另外，在微服务场景下，数据库被包含在单个服务的边界之内，因此基于内聚性原则（咦，这好像是本文第三次提到内聚原则了，可见其在软件开发中的重要性），数据库的变更最好也与项目代码一道维护在代码库中。

本文的示例项目采用了Flyway作为数据库迁移工具，加入了Flyway依赖后，在[src/main/sources/db/migration](#)目录下创建迁移脚本文件即可：

```
resources/
|   ├── db
|   |   └── migration
|   |       ├── V1__init.sql
|   |       └── V2__create_product_table.sql
```

迁移脚本的命名需要遵循一定的规则以保证脚本执行顺序，另外迁移文件生效之后不要任意修改，因为Flyway会检查文件的checksum，如果checksum不一致将导致迁移失败。

## 多环境构建

在软件的开发流程中，我们需要将软件部署到多个环境，经过多轮验证后才能最终上线。在不同的阶段中，软件的运行态可能是不一样的，比如本地开发时可能将所依赖的第三方系统stub掉；持续集成构建时可能使用的是测试用的内存数据库等等。为此，本文的示例项目推荐采用以下环境：

- local: 用于开发者本地开发
- ci: 用于持续集成
- dev: 用于前端开发联调
- qa: 用于测试人员
- uat: 类生产环境，用于功能验收(有时也称为staging环境)
- prod: 正式的生产环境
- CORS

# CORS

在前后端分离的系统中，前端单独部署，有时连域名都和后端不同，此时需要进行跨域处理。传统的做法可以通过JSONP，但这是一种比较“trick”的做法，当前更通用的实践是采用CORS机制，在Spring Boot项目中，启用CORS配置如下：

```
@Configuration
public class CorsConfiguration {
    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping(“/**”);
            }
        };
    }
}
```

对于使用Spring Security的项目，需要保证CORS工作于Spring Security的过滤器之前，为此Spring Security专门提供了相应配置：

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // by default uses a Bean by the name of corsConfigurationSource
            .cors().and()
            ...
    }

    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("https://example.com"));
        configuration.setAllowedMethods(Arrays.asList("GET", "POST"));
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}
```

# 常用第三方类库

这里列出一些比较常见的第三方库，开发者们可以根据项目所需引入：

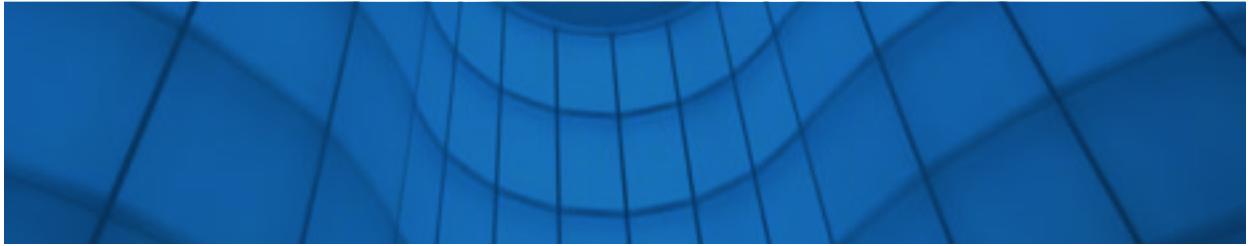
- Guava: 来自Google的常用类库
- Apache Commons: 来自Apache的常用类库
- Mockito: 主要用于单元测试的mock
- DBUnit: 测试中管理数据库测试数据
- RestAssured: 用于Rest API测试
- Jackson 2: Json数据的序列化和反序列化
- jjwt: Jwt token认证
- Lombok: 自动生成常用Java代码，比如equals()方法等；
- Feign: 声明式Rest客户端
- Tika: 用于准确检测文件类型
- itext: 生成Pdf文件等
- zxing: 生成二维码
- Xstream: 比JAXB更轻量级的XML处理库

# 总结

本文通过一个示例项目谈及到了项目之初开发者搭建后端工程的诸多方面，其中的绝大多数实践均在笔者的项目中真实落地。读完本文之后你可能会发现，文中的很多内容都是很基础很简单的。没错，的确没有什么难的东西，但是要系统性地搭建好后端项目的基础框架却不见得是每个开发团队都已经做到的事情，而这恰恰是本文的目的。最后，需要提醒的是，本文提到的实践方式只是一个参考，一方面依然存在考虑不周的地方，另一方面示例项目中用到的技术工具还存在其他替代方案，请根据自己项目的实际情况进行取舍。

# 后端开发实践系列：领域驱动设计(DDD)编码实践

作者：滕云



Martin Fowler在《企业应用架构模式》一书中写道：

I found this(business logic) a curious term because there are few things that are less logical than business logic.

初略翻译过来可以理解为：业务逻辑是很没有逻辑的逻辑。

的确，很多时候软件的业务逻辑是无法通过推理而得到的，有时甚至是被臆想出来的。这样的结果使得原本已经很复杂的业务变得更加复杂而难以理解。而在具体编码实现时，除了应付业务上的复杂性，技术上的复杂性也不能忽略，比如我们要讲究技术上的分层，要遵循软件开发的基本原则，又比如要考虑到性能和安全等等。

在很多项目中，技术复杂度与业务复杂度相互交错纠缠不清，这种火上浇油的做法成为不少软件项目无法继续往下演进的原因。然而，在合理的设计下，技术和业务是可以分离开来或者至少它们之间的耦合度是可以降低的。在不同的软件建模方法中，领域驱动设计(Domain Driven Design, DDD)尝试通过其自有的原则与套路来解决软件的复杂性问题，它将研发者的目光首先聚焦在业务本身上，使技术架构和代码实现成为软件建模过程中的“副产品”。

## DDD总览

DDD分为战略设计和战术设计。在战略设计中，我们讲求的是子域和限界上下文(Bounded Context, BC)的划分，以及各个限界上下文之间的上下游关系。当前如此火热的“在微服务中使用DDD”这个命题，究其最初的逻辑无外乎是“DDD中的限界上下文可以用于指导微服务中的服务划分”。事实上，限界上下文依然是软件模块化的一种体现，与我们一直以来追求的模块化原则的驱动力是相同的，即通过一定的手段使软件系统在人的大脑中更加有条理地呈现，让作为“目的”的人能够更简单地了解进而掌控软件系统。

如果说战略设计更偏向于软件架构，那么战术设计便更偏向于编码实现。DDD战术设计的目的是使得业务能够从技术中分离并突显出来，让代码直接表达业务的本身，其中包含了聚合根、应用服务、资源库、工厂等概念。虽然DDD不一定通过面向对象(OO)来实现，但是通常情况下在实践DDD时我们采用的是OO编程范式，行业中甚至有种说法是“DDD是OO进阶”，意思是面向对象中的基本原则(比如SOLID)在DDD中依然成立。本文主要讲解DDD的战术设计。

本文以一个简单的电商订单系统为例，通过以下方式可以获取源代码：

```
git clone https://github.com/e-commerce-sample/order-backend  
git checkout a443dace
```

## 实现业务的3种常见方式

在讲解DDD之前，让我们先来看一下实现业务代码的几种常见方式，在示例项目中有个“修改Order中Product的数量”的业务需求如下：

可以修改Order中Product的数量，但前提是Order处于未支付状态，Product数量变更后Order的总价(totalPrice)应该随之更新。

### 1. 基于“Service + 贫血模型”的实现

这种方式当前被很多软件项目所采用，主要的特点是：存在一个贫血的“领域对象”，业务逻辑通过一个Service类实现，然后通过setter方法更新领域对象，最后通过DAO(多数情况下可能使用诸如Hibernate之类的ORM框架)保存到数据库中。实现一个OrderService类如下：

```
@Transactional  
public void changeProductCount(String id, ChangeProductCountCommand command) {  
    Order order = DAO.findById(id);  
    if (order.getStatus() == PAID) {
```

```
        throw new OrderCannotBeModifiedException(id);

    }

    OrderItem orderItem = order.getOrderItem(command.getProductId());
    orderItem.setCount(command.getCount());
    order.setTotalPrice(calculateTotalPrice(order));
    DAO.saveOrUpdate(order);
}
```

这种方式依然是一种面向过程的编程范式，违背了最基本的OO原则。另外的问题在于职责划分模糊不清，使本应该内聚在**Order**中的业务逻辑泄露到了其他地方(**OrderService**)，导致**Order**成为一个只是充当数据容器的贫血模型(Anemic Model)，而非真正意义上的领域模型。在项目持续演进的过程中，这些业务逻辑会分散在不同的Service类中，最终的结果是代码变得越来越难以理解进而逐渐丧失扩展能力。

## 2. 基于事务脚本的实现

在上一种实现方式中，我们会发现领域对象(**Order**)存在的唯一目的其实是为了让ORM这样的工具能够一次性地持久化，在不使用ORM的情况下，领域对象甚至都没有必要存在。于是，此时的代码实现便退化成了事务脚本(Transaction Script)，也就是直接将Service类中计算出的结果直接保存到数据库(或者有时都没有Service类，直接通过SQL实现业务逻辑)：

```
@Transactional

public void changeProductCount(String id, ChangeProductCountCommand command) {
    OrderStatus orderStatus = DAO.getOrderStatus(id);
    if (orderStatus == PAID) {
        throw new OrderCannotBeModifiedException(id);
    }
    DAO.updateProductCount(id, command.getProductId(), command.getCount());
    DAO.updateTotalPrice(id);
}
```

可以看到，DAO中多出了很多方法，此时的DAO不再只是对持久化的封装，而是也会包含业务逻辑。

另外，**DAO.updateTotalPrice(id)**方法的实现中将直接调用SQL来实现Order总价的更新。与“Service+贫血模型”方式相似，事务脚本也存在业务逻辑分散的问题。

事实上，事务脚本并不是一种全然的反模式，在系统足够简单的情况下完全可以采用。但是：一方面“简单”这个度其实并不容易把握；另一方面软件系统通常会在不断的演进中加入更多的功能，使得原本简单的代码逐渐变得复杂。因此，事务脚本在实际的应用中使用得并不多。

### 3. 基于领域对象的实现

在这种方式中，核心的业务逻辑被内聚在行为饱满的领域对象(**Order**)中，实现**Order**类如下：

```
public void changeProductCount(ProductId productId, int count) {  
    if (this.status == PAID) {  
        throw new OrderCannotBeModifiedException(this.id);  
    }  
    OrderItem orderItem = retrieveItem(productId);  
    orderItem.updateCount(count);  
}
```

然后在Controller或者Service中，调用**Order.changeProductCount()**：

```
@PostMapping(“/order/{id}/products”)  
public void changeProductCount(@PathVariable(name = “id”) String id, @RequestBody @  
Valid ChangeProductCountCommand command) {  
    Order order = DAO.byId(orderId(id));  
    order.changeProductCount(productId.productId(command.getProductId()), command.  
    getCount());  
    order.updateTotalPrice();  
    DAO.saveOrUpdate(order);  
}
```

可以看到，所有业务（“检查Order状态”、“修改Product数量”以及“更新Order总价”）都被包含在了**Order**对象中，这些正是**Order**应该具有的职责。（不过示例代码中有个地方明显违背了内聚性原则，下文会讲到，作为悬念读者可以先行尝试着找一找）

事实上，这种方式与本文要讲的DDD战术模式已经很相近了，只是DDD抽象出了更多的概念与原则。

## 基于业务的分包

在本系列的上一篇:Spring Boot项目模板文章中，其实我已经讲到了基于业务的分包，结合DDD的场景，这里再简要讨论一下。所谓基于业务分包即通过软件所实现的业务功能进行模块化划分，而不是从技术的角度划分(比如首先划分出service和infrastructure等包)。在DDD的战略设计中，我们关注于从一个宏观的视角俯视整个软件系统，然后通过一定的原则对系统进行子域和限界上下文的划分。在战术实践中，我们也通过类似的提纲挈领的方法进行整体的代码结构的规划，所采用的原则依然逃离不了“内聚性”和“职责分离”等基本原则。此时，首先映入眼帘的便是软件的分包。

在DDD中，聚合根(下文会讲到)是主要业务逻辑的承载体，也是“内聚性”原则的典型代表，因此通常的做法便是基于聚合根进行顶层包的划分。在示例电商项目中，有两个聚合根对象**Order**和**Product**，分别创建**order**包和**product**包，然后在各自的顶层包下再根据代码结构的复杂程度划分子包，比如对于**product**包：

```
└── product
    ├── CreateProductCommand.java
    ├── Product.java
    ├── ProductApplicationService.java
    ├── ProductController.java
    ├── ProductId.java
    ├── ProductNotFoundException.java
    ├── ProductRepository.java
    └── representation
        ├── ProductRepresentationService.java
        └── ProductSummaryRepresentation.java
```

可以看到，**ProductRepository**和**ProductController**等多数类都直接放在了**product**包下，而没有单独分包；但是展现类**ProductSummaryRepresentation**却做了单独分包；这里的原则是：在所有类已经被内聚在了**product**包下的情况下，如果代码结构足够的简单，那么没有必要再次进行子包的划分，**ProductRepository**和**ProductController**便是这种情况；而如果多个类需要做再次的内聚，那么需要另行分包，比如通过REST API接口返回Product数据时，代码中涉及到了两个对象**ProductRepresentationService**和**ProductSummaryRepresentation**，这两个对象是紧密关联的，因此将他们放在**representation**子包下。而对于更加复杂的Order，分包如下：

```
|—— order
|   |—— OrderApplicationService.java
|   |—— OrderController.java
|   |—— OrderPaymentProxy.java
|   |—— OrderPaymentService.java
|   |—— OrderRepository.java
|   |—— command
|   |   |—— ChangeAddressDetailCommand.java
|   |   |—— CreateOrderCommand.java
|   |   |—— OrderItemCommand.java
|   |   |—— PayOrderCommand.java
|   |   |—— UpdateProductCountCommand.java
|   |—— exception
|   |   |—— OrderCannotBeModifiedException.java
|   |   |—— OrderNotFoundException.java
|   |   |—— PaidPriceNotSameWithOrderPriceException.java
|   |   |—— ProductNotInOrderException.java
|   |—— model
|   |   |—— Order.java
|   |   |—— OrderFactory.java
```

```
|   |   ├── OrderId.java  
|   |   ├── OrderIdGenerator.java  
|   |   ├── OrderItem.java  
|   |   └── OrderStatus.java  
|   └── representation  
|       ├── OrderItemRepresentation.java  
|       ├── OrderRepresentation.java  
|       └── OrderRepresentationService.java
```

可以看到，我们专门创建了一个model包用于放置所有与**Order**聚合根相关的领域对象；另外，基于同类型相聚原则，创建**command**包和**exception**包分别用于放置请求类和异常类。

## 领域模型的门面——应用服务

UML中有用例(Use Case)的概念，表示的是软件向外提供业务功能的基本逻辑单元。在DDD中，由于业务被提到了第一优先级，那么自然地我们希望对业务的处理能够显现出来，为了达到这样的目的，DDD专门提供了一个名为应用服务(ApplicationService)的抽象层。ApplicationService采用了门面模式，作为领域模型向外提供业务功能的总出入口，就像酒店的前台处理客户的不同需求一样。



在编码实现业务功能时，通常用2种工作流程：

- 自底向上：先设计数据模型，比如关系型数据库的表结构，再实现业务逻辑。我在与不同的程序员结对编程的时候，总会是听到这么一句话：“让我先把数据库表的字段设计出来吧”。这种方式将关注点优先放在了技术性的数据模型上，而不是代表业务的领域模型，是DDD之反。
- 自顶向下：拿到一个业务需求，先与客户方确定好请求数据格式，再实现Controller和ApplicationService，然后实现领域模型(此时的领域模型通常已经被识别出来)，最后实现持久化。

在DDD实践中，自然应该采用自顶向下的实现方式。ApplicationService的实现遵循一个很简单的原则，即一个业务用例对应ApplicationService上的一个业务方法。比如，对于上文提到的“修改Order中Product的数量”业务需求实现如下：

实现OrderApplicationService：

```
@Transactional  
public void changeProductCount(String id, ChangeProductCountCommand command) {  
    Order order = orderRepository.byId(orderId(id));  
    order.changeProductCount(productId.productId(command.getProductId()), command.  
    getCount());  
    orderRepository.save(order);  
}
```

OrderController调用OrderApplicationService：

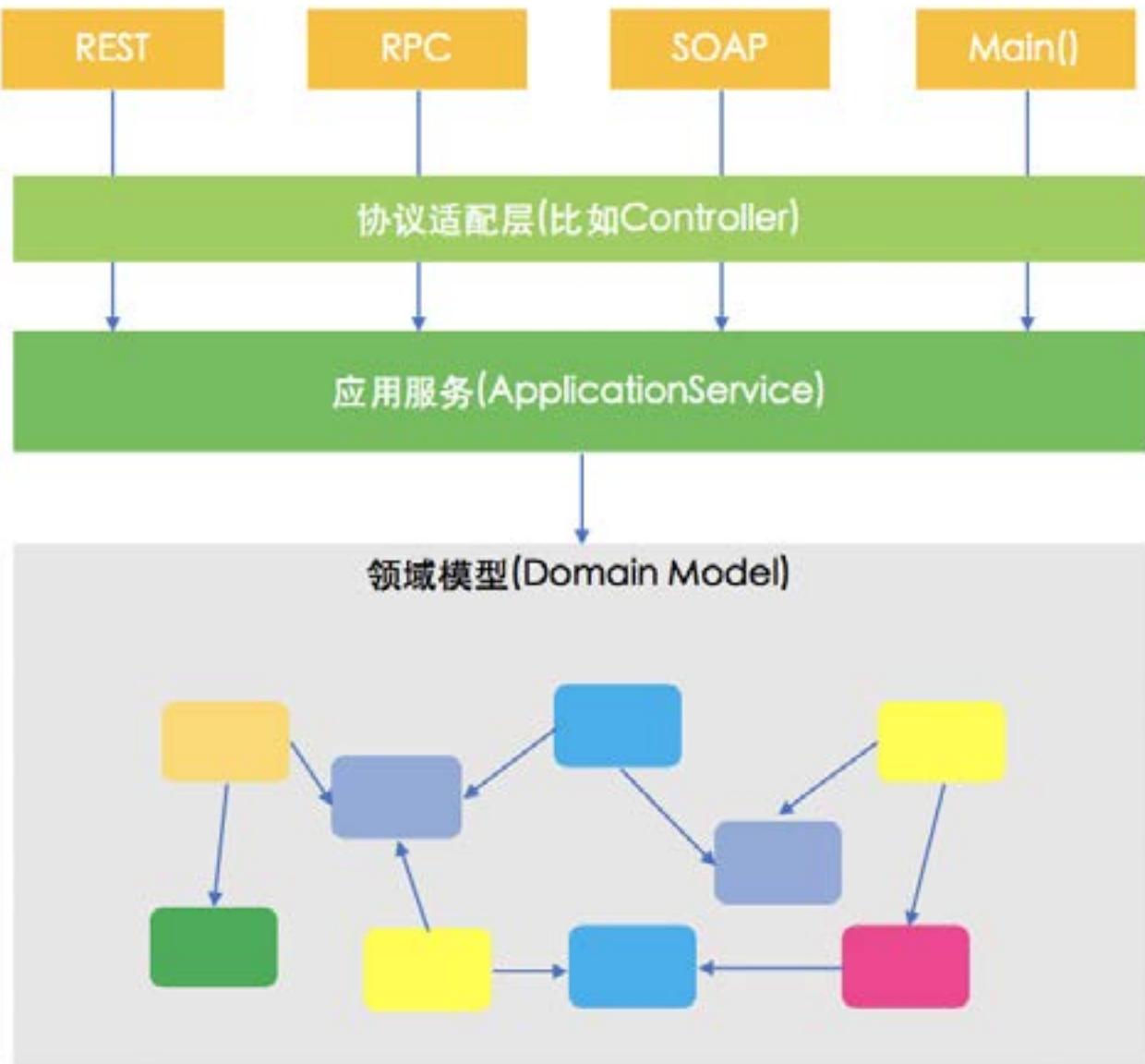
```
@PostMapping(“/{id}/products”)  
public void changeProductCount(@PathVariable(name = “id”) String id, @RequestBody @  
Valid ChangeProductCountCommand command) {  
    orderApplicationService.changeProductCount(id, command);  
}
```

此时，`order.changeProductCount()`和`orderRepository.save()`都没有必要实现，但是由`OrderController`和`OrderApplicationService`所构成的业务处理的架子已经搭建好了。

可以看到，“修改Order中Product的数量”用例中的`OrderApplicationService.changeProductCount()`方法实现中只有不多的3行代码，然而，如此简单的ApplicationService却存在很多讲究。

ApplicationService需要遵循以下原则：

- 业务方法与业务用例一一对应：前面已经讲到，不再赘述。
- 业务方法与事务一一对应：也即每一个业务方法均构成了独立的事务边界，在本例中，`OrderApplicationService.changeProductCount()`方法标记有Spring的`@Transactional`注解，表示整个方法被封装到了一个事务中。
- 本身不应该包含业务逻辑：业务逻辑应该放在领域模型中实现，更准确的说是放在聚合根中实现，在本例中，`order.changeProductCount()`方法才是真正实现业务逻辑的地方，而ApplicationService只是作为代理调用`order.changeProductCount()`方法，因此，ApplicationService应该是很薄的一层。
- 与UI或通信协议无关：ApplicationService的定位并不是整个软件系统的门面，而是领域模型的门面，这意味着ApplicationService不应该处理诸如UI交互或者通信协议之类的技术细节。在本例中，Controller作为ApplicationService的调用者负责处理通信协议(HTTP)以及与客户端的直接交互。
- 这种处理方式使得ApplicationService具有普适性，也即无论最终的调用方是HTTP的客户端，还是RPC的客户端，甚至一个Main函数，最终都统一通过ApplicationService才能访问到领域模型。接受原始数据类型：ApplicationService作为领域模型的调用方，领域模型的实现细节对其来说应该是个黑盒子，因此ApplicationService不应该引用领域模型中的对象。此外，ApplicationService接受的请求对象中的数据仅仅用于描述本次业务请求本身，在能够满足业务需求的条件下应该尽量的简单。因此，ApplicationService通常处理一些比较原始的数据类型。在本例中，`OrderApplicationService`所接受的Order ID是Java原始的String类型，在调用领域模型中的Repository时，才被封装为`OrderId`对象。



## 业务的载体——聚合根

接地气一点地讲，聚合根(Aggregate Root, AR)就是软件模型中那些最重要的以名词形式存在的领域对象，比如本文示例项目中的**Order**和**Product**。又比如，对于一个会员管理系统，会员(Member)便是一个聚合根；对于报销系统，报销单(Expense)便是一个聚合根；对于保险系统，保单(Policy)便是一个聚合根。聚合根是主要的业务逻辑载体，DDD中所有的战术实现都围绕着聚合根展开。

然而，并不是说领域模型中的所有名词都可以建模为聚合根。所谓“聚合”，顾名思义，即需要将领域中高度内聚的概念放到一起组成一个整体。至于哪些概念才能聚到一起，需要我们对业务本身有很深刻的认识，这也是为什么DDD强调开发团队需要和领域专家一起工作的原因。近年来流行起来的事件风暴建模活动，究其本意也是通过罗列出领域中发生的所有事件可以让我们全面的了解领域中的业务，进而识别出聚合根。

对于“更新Order中Product数量”用例，聚合根**Order**的实现如下：

```
public void changeProductCount(ProductId productId, int count) {
    if (this.status == PAID) {
        throw new OrderCannotBeModifiedException(this.id);
    }

    OrderItem orderItem = retrieveItem(productId);
    orderItem.updateCount(count);
    this.totalPrice = calculateTotalPrice();
}

private BigDecimal calculateTotalPrice() {
    return items.stream()
        .map(OrderItem::totalPrice)
        .reduce(ZERO, BigDecimal::add);
}

private OrderItem retrieveItem(ProductId productId) {
    return items.stream()
        .filter(item -> item.getProductId().equals(productId))
        .findFirst()
        .orElseThrow(() -> new ProductNotInOrderException(productId, id));
}
```

在本例中，**Order**中的品项(**orderItems**)和总价(**totalPrice**)是密切相关的，**orderItems**的变化会直接导致**totalPrice**的变化，因此，这二者自然应该内聚在**Order**下。此外，**totalPrice**的变化是**orderItems**变化的必然结果，这种因果关系是业务驱动出来的，为了保证这种“必然”，我们需要在**Order.changeProductCount()**方法中同时实现“因”和“果”，也即聚合根应该保证业务上的一致性。在DDD中，业务上的一致性被称为不变条件(**Invariants**)。

还记得上文中提到的“违背内聚性的悬念”吗？当时调用**Order**上的业务方式如下：

```
.....
order.changeProductCount(productId,
    productId(command.getProductId()),
    command.getCount());
order.updateTotalPrice();
....
```

为了实现“更新Order中Product数量”业务功能，这里先后调用了**Order**上的两个**public**方法**changeProductCount()**和**updateTotalPrice()**。虽然这种做法也能正确地实现业务逻辑，但是它将保证业务一致性的职责交给了**Order**的调用方(上文中的**Controller**)而不是**Order**自身，此时调用方需要确保在调用了**changeProductCount()**之后必须调用**updateTotalPrice()**方法，这一方面是**Order**中业务逻辑的泄露，另一方面调用方并不承担这样的职责，而**Order**才最应该承担这样的职责。

对内聚性的追求会自然地延伸出聚合根的边界。在DDD的战略设计中，我们已经通过限界上下文的划分将一个大的软件系统拆分为了不同的“模块”，在这样的前提下，再在某个限界上下文中来讨论内聚性将比在大泥球系统中讨论变得简单得多。

对聚合根的设计需要提防上帝对象(God Object)，也即用一个大而全的领域对象来实现所有的业务功能。上帝对象的背后存在着一种表面上看似合理的逻辑：既然要内聚，那么让我们把所有相关的东西都聚到一起吧，比如用一个**Product**类来应付所有的业务场景，包括订单、物流、发票等等。这种机械的方式看似内聚，实则恰恰是内聚性的反面。要解决这样的问题依然需要求助于限界上下文，不同限界上下文使用各自的通用语言(Ubiquitous Language)，通用语言要求一个业务概念不应该有二义性，在这样的原则下，不同的限界上下文可能都有自己的**Product**类，虽然名字相同，却体现着不同的业务。



除了内聚性和一致性，聚合根还有以下特征：

- 聚合根的实现应该与框架无关：既然DDD讲求业务复杂度和技术复杂度的分离，那么作为业务主要载体的聚合根应该尽量少地引用技术框架级别的设施，最好是POJO。试想一下，如果你的项目哪天需要从Spring迁移到Play，而你可以自信地给老板说，直接将核心Java代码拷贝过去即可，这将是一种多么美妙的体验。又或者说，很多时候技术框架会有“大步”的升级，这种升级会导致框架中API的变化并且不再支持向后兼容，此

时如果我们的领域模与框架无关，那么便可做到在框架升级的过程中幸免于难。

- 聚合根之间的引用通过ID完成：在聚合根边界设计合理的情况下，一次业务用例只会更新一个聚合根，此时你在该聚合根中去引用另外聚合根的整体有什么好处呢？  
在本文示例中，一个Order下的OrderItem引用了ProductId，而不是整个Product。
- 聚合根内部的所有变更都必须通过聚合根完成：为了保证聚合根的一致性，同时避免聚合根内部逻辑向外泄露，客户方只能将整个聚合根作为统一调用入口。
- 如果一个事务需要更新多个聚合根，首先思考一下自己的聚合根边界处理是否出了问题，因为在设计合理的情况下通常不会出现一个事务更新多个聚合根的场景。如果这种情况的确是业务所需，那么考虑引入消息机制和事件驱动架构，保证一个事务只更新一个聚合根，然后通过消息机制异步更新其他聚合根。
- 聚合根不应该引用基础设施。
- 外界不应该持有聚合根内部的数据结构。
- 尽量使用小聚合。

聚合根一定是实体对象，但是并不是所有实体对象都是聚合根，同时聚合根还可以拥有其他子实体对象。聚合根的ID在整个软件系统中全局唯一，而其下的子实体对象的ID只需在单个聚合根下唯一即可。在本文示例项目中，OrderItem是聚合根Order下的子实体对象：

```
public class OrderItem {  
    private ProductId productId;  
    private int count;  
    private BigDecimal itemPrice;  
}
```

可以看到，虽然OrderItem使用了ProductId作为ID，但是此时我们并没有享受ProductId的全局唯一性，事实上多个Order可以包含相同ProductId的OrderItem，也即多个订单可以包含相同的产品。

区分实体和值对象的一个很重要的原则便是根据相等性来判断，实体对象的相等性是通过ID来完成的，对于两个实体，如果他们的所有属性均相同，但是ID不同，那么他们依然两个不同的实体，就像一对长得一模一样的双胞胎，他们依然是两个不同的自然人。对于值对象来说，相等性的判断是通过属性字段来完成的。比如，订单下的送货地址Address对象便是一个典型的值对象：

## 实体 vs 值对象

软件模型中存在实体对象(Entity)和值对象(Value Object)之说，这种划分方式事实上并不是DDD的专属，但是在DDD中我们非常强调这两者之间的区别。

实体对象表示的是具有一定生命周期并且拥有全局唯一标识(ID)的对象，比如本文中的Order和Product，而值对象表示用于起描述性作用的，没有唯一标识的对象，比如Address对象。

```
public class Address {  
    private String province;  
    private String city;  
    private String detail;  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) {  
            return true;  
        }  
        if (o == null || getClass() != o.getClass()) {  
            return false;  
        }  
        Address address = (Address) o;  
        return province.equals(address.province) &&  
               city.equals(address.city) &&  
               detail.equals(address.detail);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(province, city, detail);  
    }  
}
```

在**Address**的**equals()**方法中，通过判断**Address**所包含的所有属性(**province**, **city**, **detail**)来决定两个**Address**的相等性。

值对象还有一个特点是不变的(Immutable), 也就说一个值对象一旦被创建出来了便不能对其进行变更, 如果要变更, 必须重新创建一个新的值对象整体替换原有的。比如, 示例项目有一个业务需求:

“在订单未支付的情况下, 可以修改订单送货地址的详细地址(detail)”

由于**Address**是**Order**聚合根中的一个对象, 对**Address**的更改只能通过**Order**完成, 在**Order**中实现**changeAddressDetail()**方法:

```
public void changeAddressDetail(String detail) {  
    if (this.status == PAID) {  
        throw new OrderCannotBeModifiedException(this.id);  
    }  
  
    this.address = this.address.changeDetailTo(detail);  
}
```

可以看到, 通过调用**address.changeDetailTo()**方法, 我们获取到了一个全新的**Address**对象, 然后将新的**Address**对象整体赋值给**address**属性。此时**Address.changeDetailTo()**的实现如下:

```
public Address changeDetailTo(String detail) {  
    return new Address(this.province, this.city, detail);  
}
```

这里的**changeDetailTo()**方法使用了新的详细地址**detail**和未发生变更的**province**、**city**重新创建出了一个**Address**对象。

值对象的不变性使得程序的逻辑变得更加简单, 你不用去维护复杂的状态信息, 需要的时候创建, 不要的时候直接扔掉即可, 使得值对象就像程序中的过客一样。在DDD建模中, 一种受推崇的做法便是将业务概念尽量建模为值对象。

对于OrderItem来说，由于我们的业务需要对OrderItem的数量进行修改，也即拥有生命周期的意味，因此本文将OrderItem建模为了实体对象。但是，如果没有这样的业务需求，那么将OrderItem建模为值对象应该更合适一些。

另外，需要指明的是，实体和值对象的划分并不是一成不变的，而应该根据所处的限界上下文来界定，相同一个业务名词，在一个限界上下文中可能是实体，在另外的限界上下文中可能是值对象。比如，订单Order在采购上下文中应该建模为一个实体，但是在物流上下文中便可建模为一个值对象。

## 聚合根的家——资源库

通俗点讲，资源库(Repository)就是用来持久化聚合根的。从技术上讲，Repository和DAO所扮演的角色相似，不过DAO的设计初衷只是对数据库的一层很薄的封装，而Repository是更偏向于领域模型。另外，在所有的领域对象中，只有聚合根才“配得上”拥有Repository，而DAO没有这种约束。

实现Order的资源库OrderRepository如下：

```
public void save(Order order) {  
    String sql = "INSERT INTO ORDERS (ID, JSON_CONTENT) VALUES (:id, :json) " +  
        "ON DUPLICATE KEY UPDATE JSON_CONTENT=:json;" ;  
    Map<String, String> paramMap = of("id", order.getId().toString(), "json", objectMapper.  
        writeValueAsString(order));  
    jdbcTemplate.update(sql, paramMap);  
}  
  
public Order getById(OrderId id) {  
    try {  
        String sql = "SELECT JSON_CONTENT FROM ORDERS WHERE ID=:id;" ;  
        return jdbcTemplate.queryForObject(sql, of("id", id.toString()), mapper());  
    } catch (EmptyResultDataAccessException e) {  
        throw new OrderNotFoundException(id);  
    }  
}
```

在OrderRepository中，我们只定义了**save()**和**byId()**方法，分别用于保存/更新聚合根和通过ID获取聚合根。这两个方法是Repository中最常见的方法，有的DDD实践者甚至认为一个纯粹的Repository只应该包含这两个方法。

读到这里，你可能会有些疑问：为什么OrderRepository中没有更新和查询等方法？事实上，Repository所扮演的角色只是向领域模型提供聚合根而已，就像一个聚合根的“容器”一样，这个“容器”本身并不关心客户端对聚合根的操作到底是新增还是更新，你给一个聚合根对象，Repository只是负责将其状态从计算机的内存同步到持久化机制中，从这个角度讲，Repository只需要一个类似**save()**的方法便可完成同步操作。当然，这个是从概念的出发点得出的设计结果，在技术层面，新增和更新还是需要区别对待，比如SQL语句有**insert**和**update**之分，只是我们将这样的技术细节隐藏在了**save()**方法中，客户方并无需知道这些细节。在本例中，我们通过MySQL的ON DUPLICATE KEY UPDATE特性同时处理对数据库的新增和更新操作。当然，我们也可以通过编程判断聚合根在数据库中是否已经存在，如果存在则**update**，否则**insert**。另外，诸如Hibernate这样的持久化框架自动提供**saveOrUpdate()**方法可以直接用于对聚合根的持久化。

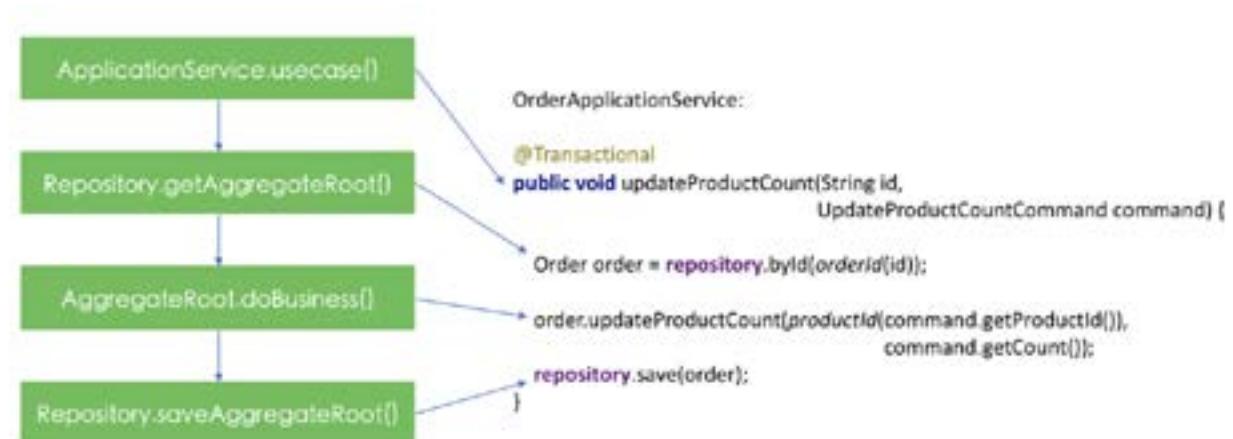
对于查询功能来说，在Repository中实现查询本无不合理之处，然而项目的演进可能导致Repository中充斥着大量的查询代码“喧宾夺主”似的掩盖了Repository原本的目的。事实上，DDD中读操作和写操作是两种很不一样的过程，笔者的建议是尽量将此二者分开实现，由此查询功能将从Repository中分离出去，在下文中我将详细讲到。

在本例中，我们在技术实现上使用到了Spring的**JdbcTemplate**和JSON格式持久化**Order**聚合根，其实Repository并不与某种持久化机制绑定，一个被抽象出来的Repository向外暴露的功能“接口”始终是向领域模型提供聚合根对象，就像“聚合根的家”一样。

好了，至此让我们来做个回顾，上文中我们以“更新Order中的Product数量”业务需求为例，讲到了应用服务、聚合根和资源库，对该业务需求的处理流程体现了DDD处理业务需求的最常见最典型的形式：

**应用服务作为总体协调者，先通过资源库获取到聚合根，然后调用聚合根中的业务方法，最后再次调用资源库保存聚合根。**

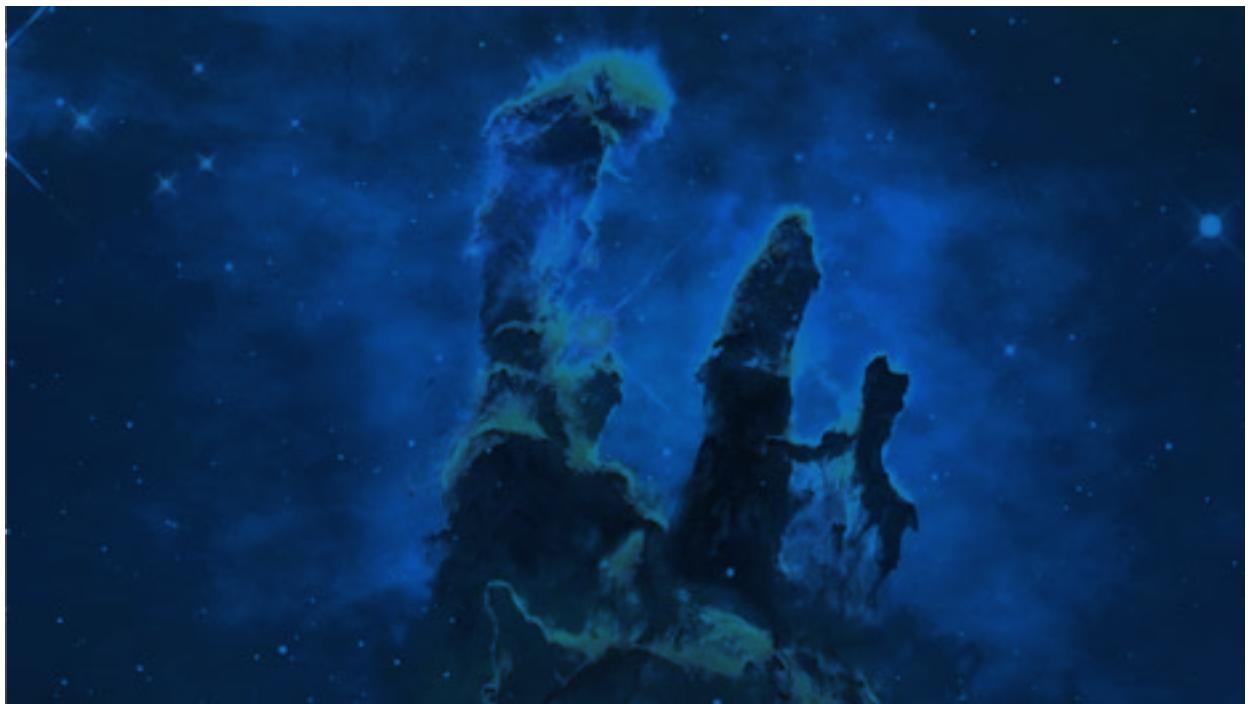
流程示意图如下：



## 创生之柱——工厂

稍微提炼一下，我们便知道软件里面的写操作要么是修改既有数据，要么是新建数据。对于前者，DDD给出的答案已经在上文中讲到，接下来我们讲讲在DDD中如何新建聚合根。

创建聚合根通常通过设计模式中的工厂(Factory)模式完成，这一方面可以享受到工厂模式本身的好处，另一方面，DDD中的Factory还具有将“聚合根的创建逻辑”显现出来的效果。



聚合根的创建过程可简单可复杂，有时可能直接调用构造函数即可，而有时却存在一个复杂的构造流程，比如需要调用其他系统获取数据等。通常来讲，Factory有两种实现方式：

- 直接在聚合根中实现Factory方法，常用于简单的创建过程
- 独立的Factory类，用于有一定复杂度的创建过程，或者创建逻辑不适合放在聚合根上

让我们先演示一下简单的Factory方法，在示例订单系统中，有个业务用例是“创建Product”：

创建Product，属性包括名称(name)，描述(description)和单价(price)，ProductId为UUID

在**Product**类中实现工厂方法**create()**:

```
public static Product create(String name, String description, BigDecimal price) {  
    return new Product(name, description, price);  
}
```

```
private Product(String name, String description, BigDecimal price) {  
    this.id = ProductId.newProductId();  
    this.name = name;  
    this.description = description;  
    this.price = price;  
    this.createdAt = Instant.now();  
}
```

这里，**Product**中的**create()**方法并不包含创建逻辑，而是将创建过程直接代理给了**Product**的构造函数。你可能觉得这个**create()**方法有些多此一举，然而这种做法的初衷依然是：我们希望将聚合根的创建逻辑突显出来。构造函数本身是一个非常技术的东西，任何地方只要涉及到在计算机内存中新建对象都需要使用构造函数，无论创建的初始原因是业务需要，还是从数据库加载，亦或是从JSON数据反序列化。因此程序中往往存在多个构造函数用于不同的场景，而为了将业务上的创建与技术上的创建区别开来，我们引入了**create()**方法用于表示业务上的创建过程。

“创建Product”所设计到的Factory的确简单，让我们再来看看另外一个例子：“创建Order”：

创建Order，包含用户选择的Product及其数量，OrderId必须调用第三方的OrderIdGenerator获取

这里的**OrderIdGenerator**是具有服务性质的对象(即下文中的领域服务)，在DDD中，聚合根通常不会引用其他服务类。另外，调用OrderIdGenerator生成ID应该是一个业务细节，如前文所讲，这种细节不应该放在ApplicationService中。此时，可以通过Factory类来完成Order的创建：

```
@Component
public class OrderFactory {
    private final OrderIdGenerator idGenerator;

    public OrderFactory(OrderIdGenerator idGenerator) {
        this.idGenerator = idGenerator;
    }

    public Order create(List<OrderItem> items, Address address) {
        OrderId orderId = idGenerator.generate();
        return Order.create(orderId, items, address);
    }
}
```

## 必要的妥协——领域服务

前面我们提到，聚合根是业务逻辑的主要载体，也就是说业务逻辑的实现代码应该尽量地放在聚合根或者聚合根的边界之内。但有时，有些业务逻辑并不适合于放在聚合根上，比如前文的**OrderIdGenerator**便是如此，在这种“迫不得已”的情况下，我们引入领域服务(Domain Service)。

还是先来看一个例子，对于Order的支付有以下业务用例：

通过支付网关OrderPaymentService完成Order的支付。

在**OrderApplicationService**中，直接调用领域服务**OrderPaymentService**:

```
@Transactional  
public void pay(String id, PayOrderCommand command) {  
    Order order = orderRepository.byId(orderId(id));  
    orderPaymentService.pay(order, command.getPaidPrice());  
    orderRepository.save(order);  
}
```

然后实现**OrderPaymentService**:

```
public void pay(Order order, BigDecimal paidPrice) {  
    order.pay(paidPrice);  
    paymentProxy.pay(order.getId(), paidPrice);  
}
```

这里的**PaymentProxy**与**OrderIdGenerator**相似，并不适合于放在**Order**中。可以看到，在**OrderApplicationService**中，我们并没有直接调用**Order**中的业务方法，而是先调用**OrderPaymentService.pay()**，然后在**OrderPaymentService.pay()**中完成调用支付网关**PaymentProxy.pay()**这样的业务细节。

到此，再来反观在通常的实践中我们编写的Service类，事实上这些Service类将DDD中的ApplicationService和DomainService糅合在了一起，比如在“基于Service + 贫血模型”的实现“小节中的OrderService便是如此。在DDD中，ApplicationService和DomainService是两个很不一样的概念，前者是必须有的DDD组件，而后者只是一种妥协的结果，因此程序中的DomainService应该越少越好。

# Command对象

通常来说，DDD中的写操作并不需要向客户端返回数据，在某些情况下(比如新建聚合根)可以返回一个聚合根的ID，这意味着Application Service或者聚合根中的写操作方法通常返回**void**即可。比如，对于**OrderApplicationService**，各个方法签名如下：

```
public OrderId createOrder(CreateOrderCommand command);  
public void changeProductCount(String id, ChangeProductCountCommand command);  
public void pay(String id, PayOrderCommand command);  
public void changeAddressDetail(String id, String detail);
```

在多数情况下我们使用了后缀为**Command**的对象传给Application Service。比如**CreateOrderCommand**和**ChangeProductCountCommand**。Command即命令的意思，也即写操作表示的是外部向领域模型发起的一次命令操作。事实上，从技术上讲，Command对象只是一种类型的DTO对象，它封装了客户端发过来的请求数据。在Controller中所接收的所有写操作都需要通过Command进行包装，在Command比较简单(比如只有1-2个字段)的情况下Controller可以将Command解开之后，将其中的数据直接传递给Application Service，比如**changeAddressDetail()**便是如此；而在Command中数据字段比较多时，可以直接将Command对象传递给Application Service。当然，这并不是DDD中需要严格遵循的一个原则，比如无论Command的简繁程度，统一将所有Command从Controller传递给Application Service，也不存在太大的问题，更多的只是一个编码习惯上的选择。不过有一点需要强调，即前文提到的“Application Service需要接受原始数据类型而不是领域模型中的对象”，在这里意味着Command对象中也应该包含原始的数据类型。

统一使用Command对象还有个好处是，我们通过查找所有后缀为**Command**的对象，便可以概览性地了解软件系统向外提供的业务功能。

阶段性小结一下，以上我们主要围绕着软件的“写操作”在DDD中的实现进行讨论，并且讲到了3种场景，分别是：

- 通过聚合根完成业务请求
- 通过Factory完成聚合根的创建
- 通过DomainService完成业务请求

以上3种场景大致上涵盖了DDD完成业务写操作的基本方面，总结下来3句话：创建聚合根通过Factory完成；业务逻辑优先在聚合根边界内完成；聚合根中不合适放置的业务逻辑才考虑放到DomainService中。



## DDD中的读操作

软件中的读模型和写模型是很不一样的，我们通常所讲的业务逻辑更多的时候是在写操作过程中需要关注的东西，而读操作更多关注的是如何向客户端返回恰当的数据展现。

在DDD的写操作中，我们需要严格地按照“应用服务 -> 聚合根 -> 资源库”的结构进行编码，而在读操作中，采用与写操作相同的结构有时不但得不到好处，反而使整个过程变得冗繁。这里介绍3种读操作的方式：

- 基于领域模型的读操作
- 基于数据模型的读操作
- CQRS

首先，无论哪种读操作方式，都需要遵循一个原则：领域模型中的对象不能直接返回给客户端，因为这样领域模型的内部便暴露给了外界，而对领域模型的修改将直接影响到客户端。因此，在DDD中我们通常为读操作专门创建相应的模型用于数据展现。在写操作中，我们通过Command后缀进行请求数据的统一，在读操作中，我们通过Representation后缀进行展现数据的统一，这里的Representation也即REST中的“R”。

## 基于领域模型的读操作

这种方式将读模型和写模型糅合到一起，先通过资源库获取到领域模型，然后将其转换为Representation对象，这也是当前被大量使用的方式，比如对于“获取Order详情的接口”，OrderApplicationService实现如下：

```
@Transactional(readOnly = true)
public OrderRepresentation byId(String id) {
    Order order = orderRepository.byId(orderId(id));
    return orderRepresentationService.toRepresentation(order);
}
```

我们先通过**orderRepository.byId()**获取到**Order**聚合根对象，然后调用**orderRepresentationService.toRepresentation()**将**Order**转换为展现对象**OrderRepresentation**，**OrderRepresentationService.toRepresentation()**实现如下：

```
public OrderRepresentation toRepresentation(Order order) {
    List<OrderItemRepresentation> itemRepresentations = order.getItems().stream()
        .map(orderItem -> new OrderItemRepresentation(orderItem.getProductId().toString(),
            orderItem.getCount(),
            orderItem.getItemPrice()))
        .collect(Collectors.toList());

    return new OrderRepresentation(order.getId().toString(),
        itemRepresentations,
        order.getTotalPrice(),
        order.getStatus(),
        order.getCreatedAt());
}
```

这种方式的优点是非常直接明了，也不用创建新的数据读取机制，直接使用Repository读取数据即可。然而缺点也很明显：一是读操作完全束缚于聚合根的边界划分，比如，如果客户端需要同时获取Order及其所包含的Product，那么我们需要同时将Order聚合根和Product聚合根加载到内存再做转换操作，这种方式既繁琐又低效；二是在读操作中，通常需要基于不同的查询条件返回数据，比如通过Order的日期进行查询或者通过Product的名称进行查询等，这样导致的结果是Repository上处理了太多的查询逻辑，变得越来越复杂，也逐渐偏离了Repository本应该承担的职责。

### 基于数据模型的读操作

这种方式绕开了资源库和聚合，直接从数据库中读取客户端所需要的数据，此时写操作和读操作共享的只是数据库。比如，对于“获取Product列表”接口，通过一个专门的ProductRepresentationService直接从数据库中读取数据：

```
@Transactional(readOnly = true)

public PagedResource<ProductSummaryRepresentation> listProducts(int pageIndex, int
pageSlice) {

    MapSqlParameterSource parameters = new MapSqlParameterSource();
    parameters.addValue("limit", pageSlice);
    parameters.addValue("offset", (pageIndex - 1) * pageSlice);

    List<ProductSummaryRepresentation> products = jdbcTemplate.query(SELECT_SQL,
parameters,
        (rs, rowNum) -> new ProductSummaryRepresentation(rs.getString("ID"),
        rs.getString("NAME"),
        rs.getBigDecimal("PRICE")));
}

int total = jdbcTemplate.queryForObject(COUNT_SQL, newHashMap(), Integer.class);
return PagedResource.of(total, pageIndex, products);
}
```

然后在Controller中直接返回：

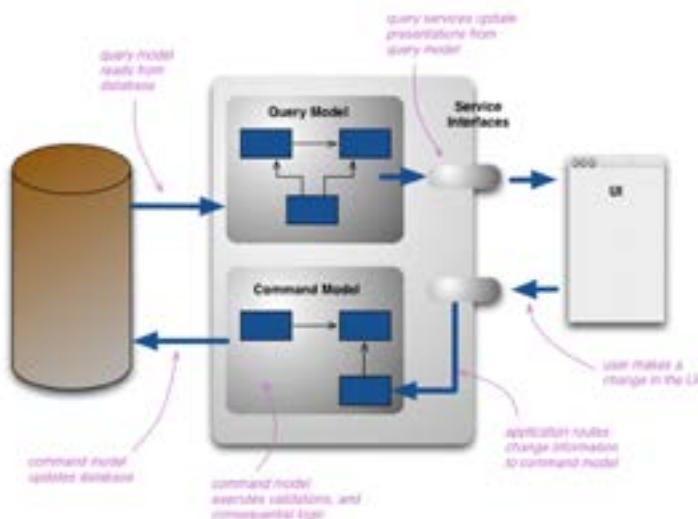
```
@GetMapping  
public PagedResource<ProductSummaryRepresentation> pagedProducts(@RequestParam(required  
= false, defaultValue = "1") int pageIndex,  
@RequestParam(required = false, defaultValue  
= "10") int pageSize){  
    return productRepresentationService.listProducts(pageIndex, pageSize);  
}
```

可以看到，真个过程并没有使用到**ProductRepository**和**Product**，而是将SQL获取到的数据直接新建为**ProductSummaryRepresentation**对象。

这种方式的优点是读操作的过程不用囿于领域模型，而是基于读操作本身的需求直接获取需要的数据即可，一方面简化了整个流程，另一方面大大提升了性能。但是，由于读操作和写操作共享了数据库，而此时的数据库主要是对应于聚合根的结构创建的，因此读操作依然会受到写操作的数据模型的牵制。不过这种方式是一种很好的折中，微软也提倡过这种方式，更多细节请参考微软官网。

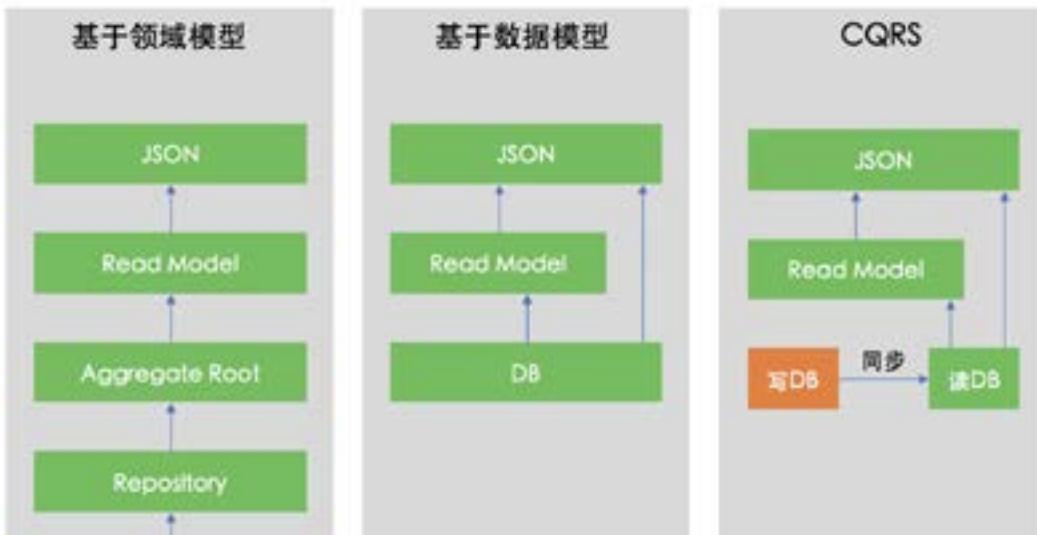
## CQRS

CQRS(Command Query Responsibility Segregation)，即命令查询职责分离，这里的命令可以理解为写操作，而查询可以理解为读操作。与“基于数据模型的读操作”不同的是，在CQRS中写操作和读操作使用了不同的数据库，数据从写模型数据库同步到读模型数据库，通常通过领域事件的形式同步变更信息。



这样一来，读操作便可以根据自身所需独立设计数据结构，而不用受写模型数据结构的牵制。CQRS本身是一个很大的话题，已经超出了本文的范围，读者可以自行研究。

到此，DDD中的读操作可以大致分为3种实现方式：



## 总结

本文主要介绍了DDD中的应用服务、聚合、资源库和工厂等概念以及与它们相关的编码实践，然后着重讲到了软件的读写操作在DDD中的实现方式，其中写操作的3种场景为：

- 通过聚合根完成业务请求，这是DDD完成业务请求的典型方式。
- 通过Factory完成聚合根的创建，用于创建聚合根。
- 通过DomainService完成业务请求，当业务放在聚合根中不合适时才考虑放在DomainService中。

对于读操作，同样给出了3种方式：

- 基于领域模型的读操作(读写操作糅合在了一起，不推荐)。
- 基于数据模型的读操作(绕过聚合根和资源库，直接返回数据，推荐)。
- CQRS(读写操作分别使用不同的数据库)。

以上“3读3写”基本上涵盖了程序员完成业务功能的日常开发之所需，原来DDD就这么简单，不是吗？

# 后端开发实践系列：事件驱动架构(EDA)编码实践

---

作者：滕云

在本系列的前两篇文章中，笔者分别讲到了后端项目的代码模板和DDD编码实践，在本文中，我将继续以编码实践的方式分享如何落地事件驱动架构。

单纯地讲事件驱动架构(Event Driven Architecture, EDA)，那是几十年前就出现了的话题；单纯地讲领域事件，那也是这些年被大量提及并讨论得快熟透了的软件用语。然而，就笔者的观察看，事件驱动架构远没有想象中那样普遍地被开发团队所接受。即便搞微服务的人都知道除了同步的HTTP还有异步的消息机制，即便搞DDD的人都知道领域事件是其中的一等公民，事件驱动架构所带来的优点并没有相应地转化为软件从业者的青睐。

我尝试着去思考其中的原因，总结出了两点：第一是事件驱动可能是客观世界的运作方式，但不是人的自然思考问题的方式；第二是事件驱动架构在给软件带来好处的同时，又会增加额外的复杂性，比如调试的困难性，又比如并不直观的最终一致性。

当然，事实上有不少软件项目都使用了消息队列，但是这里需要明确的是，对消息队列的使用并不意味着你的项目就一定是事件驱动架构，很多项目只是由于技术方面的驱动，小范围地采用了某些消息队列（比如RabbitMQ和Kafka等）的产品而已。偌大一个系统，如果你的消息队列只是用作邮件发送的通知，那么这样系统自然谈不上采用了事件驱动架构。

放到当下，微服务兴起，DDD重现，在采用事件驱动架构时，我们需要考虑业务的建模、领域事件的设计、DDD的约束、限界上下文的边界以及更多技术方面的因素，这一个系统工程应该如何从头到尾的落地，是需要经过思考和推敲的。还是那句话，有讲究的编程并不是一件易事。

诚然，用好事件驱动架构存在实践上的难处，然而它的优点也委实诱人，本文希望形成一定的“条理”和“套路”，让事件驱动架构能够更简单的落地。

本文主要分为两大部分，第一部分独立于具体的消息队列实现来讲解通用的对领域事件的建模，第二部分以一个真实的微服务系统为例，采用RabbitMQ作为消息队列，并以此分享完整的事件驱动架构落地实践。

本文以DDD为基础进行编码，其中会涉及到DDD中的不少概念，比如聚合根、资源库和应用服务等，对DDD不熟悉的读者可以参考笔者的DDD编码实践文章。

本文的示例代码请参考github上的e-commerce-sample项目。

# 第一部分：领域事件的建模

领域事件是DDD中的一个概念，表示的是在一个领域中所发生的一次对业务有价值的事情，落到技术层面就是在  
一个业务实体对象(通常来说是聚合根)的状态发生了变化之后需要发出一个领域事件。虽然事件驱动架构中的“  
事件”不一定指“领域事件”，但本文由于密切结合DDD，因此当提到事件时，我们特指“领域事件”。

## 创建领域事件

关于领域事件的基础知识，请参考笔者的在微服务中使用领域事件文章，本文直接进入编码实践环节。

在建模领域事件时，首先需要记录事件的一些通用信息，比如唯一标识ID和创建时间等，为此创建事件基类  
**DomainEvent**：

```
public abstract class DomainEvent {  
    private final String _id;  
    private final DomainEventType _type;  
    private final Instant _createdAt;  
}
```

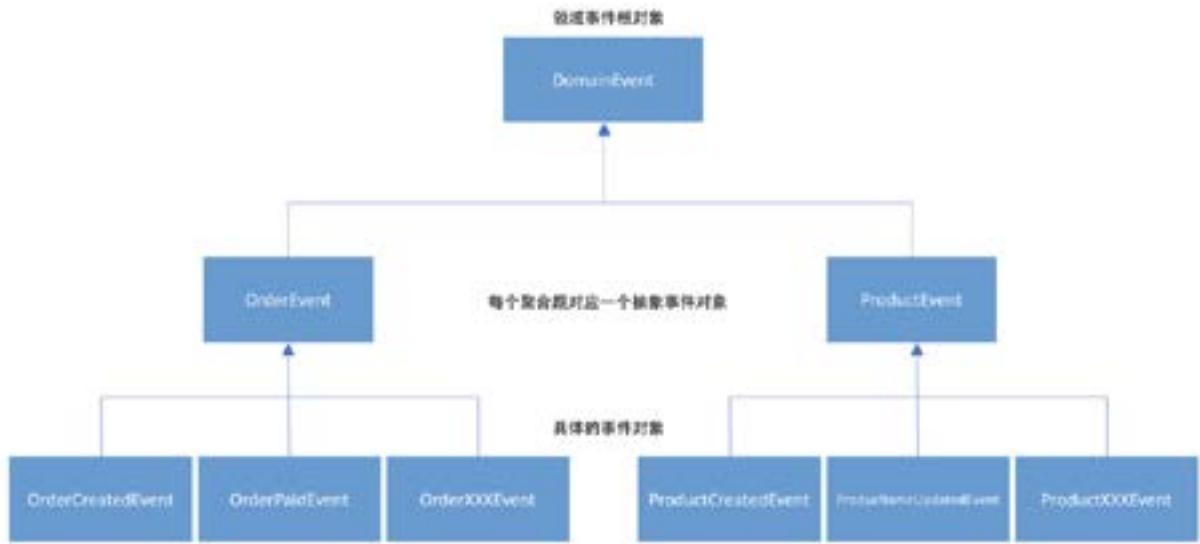
在DDD场景下，领域事件一般随着聚合根状态的更新而产生，另外，在事件的消费方，有时我们希望监听发生在某  
个聚合根下的所有事件，为此笔者建议为每一个聚合根对象创建相应的事件基类，其中包含聚合根的ID，比如对于  
订单(Order)类，创建**OrderEvent**：

```
public abstract class OrderEvent extends DomainEvent {  
    private final String orderId;  
}
```

然后对于实际的Order事件，统一继承自**OrderEvent**，比如对于创建订单的**OrderCreatedEvent**事件：

```
public class OrderCreatedEvent extends OrderEvent {  
    private final BigDecimal price;  
    private final Address address;  
    private final List<OrderItem> items;  
    private final Instant createdAt;  
}
```

领域事件的继承链如下：



在创建领域事件时，需要注意2点：

- 领域事件本身应该是不变的(Immutable);
- 领域事件应该携带与事件发生时相关的上下文数据信息，但是并不是整个聚合根的状态数据，例如，在创建订单时可以携带订单的基本信息，而对于产品(Product)名称更新的 **ProductNameUpdatedEvent**事件，则应该同时包含更新前后的产品名称：

```
public class ProductNameUpdatedEvent extends ProductEvent {  
    private String oldName; //更新前的名称  
    private String newName; // 更新后的名称  
}
```

## 发布领域事件

发布领域事件有多种方式，比如可以在应用服务(ApplicationService)中发布，也可以在资源库(Repository)中发布，还可以引入事件表的方式，这3种发布方式的详细比较可以参考笔者的在微服务中使用领域事件文章。笔者建议采用事件表方式，这里展开讨论一下。

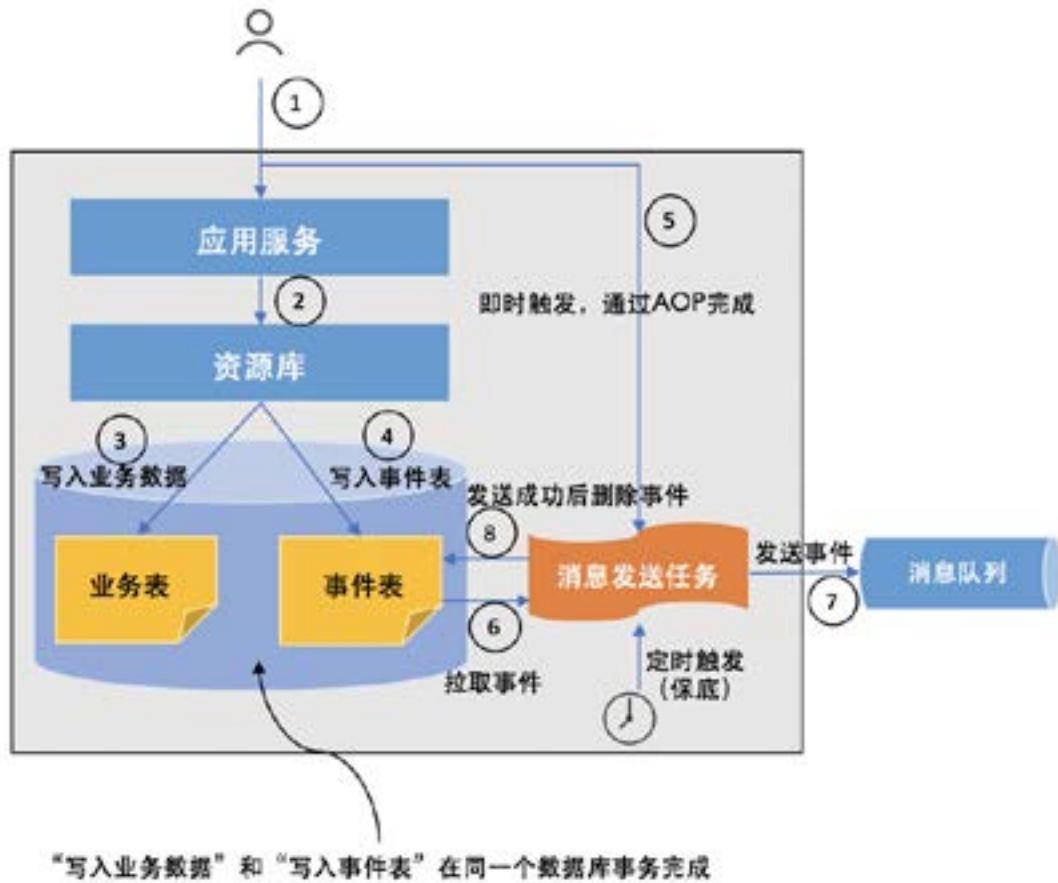
通常的业务处理过程都会更新数据库然后发布领域事件，这里一个比较重要的点是：我们需要保证数据库更新和事件发布之间的原子性，也即要么二者都成功，要么都失败。在传统的实践方式中，全局事务(Global Transaction/XA Transaction)通常用于解决此类问题。然而，全局事务本身的效果是很低的，另外，一些技术框架并不提供对全局事务的支持。当前，一种比较受推崇的方式是引入事件表，其流程大致如下：

- 1.在更新业务表的同时，将领域事件一并保存到数据库的事件表中，此时业务表和事件表在同一个本地事务中，即保证了原子性，又保证了效率。
- 2.在后台开启一个任务，将事件表中的事件发布到消息队列中，发送成功之后删除掉事件。

但是，这里又有一个问题：在第2步中，我们如何保证发布事件和删除事件之间的原子性呢？答案是：我们不用保证它们的原子性，我们需要保证的是“至少一次投递”，并且保证消费方幂等。此时的大致场景如下：

- 代码中先发布事件，成功后再从事件表中删除事件；
- 发布消息成功，事件删除也成功，皆大欢喜；
- 如果消息发布不成功，那么代码中不会执行事件删除逻辑，就像事情没有发生一样，一致性得到保证；
- 如果消息发布成功，但是事件删除失败，那么在第二次任务执行时，会重新发布消息，导致消息的重复发送。然而，由于我们要求了消费方的幂等性，也即消费方多次消费同一条消息是ok的，整个过程的一致性也得到了保证。

发布领域事件的整个流程如下：



1. 接受用户请求；
2. 处理用户请求；
3. 写入业务表；
4. 写入事件表，事件表和业务表的更新在同一个本地数据库事务中；
5. 事务完成后，即时触发事件的发送（比如可以通过Spring AOP的方式完成，也可以定时扫描事件表，还可以借助诸如MySQL的binlog之类的机制）；
6. 后台任务读取事件表；
7. 后台任务发送事件到消息队列；
8. 发送成功后删除事件。

更多有关事件表的介绍,请参考Chris Richardson的“Transaction Outbox模式”和Udi Dahan的“在不使用分布式事务条件下如何处理消息可靠性”的视频。

在事件表场景下,一种常见的做法是将领域事件保存到聚合根中,然后在Repository保存聚合根的时候,将事件保存到事件表中。这种方式对于所有的Repository/聚合根都采用的方式处理,因此可以创建对应的抽象基类。

创建所有聚合根的基类**DomainEventAwareAggregate**如下:

```
public abstract class DomainEventAwareAggregate {  
    @JsonIgnore  
    private final List<DomainEvent> events = newArrayList();  
  
    protected void raiseEvent(DomainEvent event) {  
        this.events.add(event);  
    }  
  
    void clearEvents() {  
        this.events.clear();  
    }  
  
    List<DomainEvent> getEvents() {  
        return Collections.unmodifiableList(events);  
    }  
}
```

这里的**raiseEvent()**方法用于在具体的聚合根对象中产生领域事件,然后在Repository中获取到事件,与聚合根对象一起完成持久化,创建**DomainEventAwareRepository**基类如下:

```
public abstract class DomainEventAwareRepository<AR extends DomainEventAwareAggregate> {
```

```
@Autowired  
private DomainEventDao eventDao;  
  
public void save(AR aggregate) {  
    eventDao.insert(aggregate.getEvents());  
    aggregate.clearEvents();  
    doSave(aggregate);  
}  
protected abstract void doSave(AR aggregate);  
}
```

具体的聚合根在实现业务逻辑之后调用[raiseEvent\(\)](#)方法生成事件，以“更改Order收货地址”业务过程为例：

```
public class Order extends DomainEventAwareAggregate {  
  
    //.....  
  
    public void changeAddressDetail(String detail) {  
        if (this.status == PAID) {  
            throw new OrderCannotBeModifiedException(this.id);  
        }  
        this.address = this.address.changeDetailTo(detail);  
        raiseEvent(new OrderAddressChangedEvent(getId().toString(), detail, address.getDetail()));  
    }  
    //.....  
}
```

在保存Order的时候，只需要处理Order自身的持久化即可，事件的持久化已经在[DomainEventAwareRepository](#)基类中完成：

```
@Component
public class OrderRepository extends DomainEventAwareRepository<Order> {

    //......

    @Override
    protected void doSave(Order order) {
        String sql = "INSERT INTO ORDERS (ID, JSON_CONTENT) VALUES (:id, :json) " +
                    "ON DUPLICATE KEY UPDATE JSON_CONTENT=:json;" ;
        Map<String, String> paramMap = of("id", order.getId().toString(), "json", objectMapper.writeValueAsString(order));
        jdbcTemplate.update(sql, paramMap);
    }

    //......

}
```

当业务操作的事务完成之后，需要通知消息发送设施即时发布事件到消息队列。发布过程最好做成异步的后台操作，这样不会影响业务处理的正常返回，也不会影响业务处理的效率。在Spring Boot项目中，可以考虑采用AOP的方式，在HTTP的POST/PUT/PATCH/DELETE方法完成之后统一发布事件：

```
@Aspect  
@Component  
public class DomainEventPublishAspect {  
  
//.....  
    @After(“@annotation(org.springframework.web.bind.annotation.PostMapping) || “+  
           “@annotation(org.springframework.web.bind.annotation.PutMapping) || “+  
           “@annotation(org.springframework.web.bind.annotation.PatchMapping) || “+  
           “@annotation(org.springframework.web.bind.annotation.DeleteMapping) ||” )  
    public void publishEvents(JoinPoint joinPoint) {  
        logger.info(“Trigger domain event publish process.”);  
        taskExecutor.execute(() -> publisher.publish());  
    }  
    //.....  
}
```

以上，我们使用了TaskExecutor在后台开启新的线程完成事件发布。

实际的发布由[RabbitDomainEventPublisher](#)完成：

```
@Component  
public class DomainEventPublisher {  
  
// .....,  
    public void publish() {
```

```
Instant now = Instant.now();

LockConfiguration configuration = new LockConfiguration("domain-event-publisher", now.
plusSeconds(10), now.plusSeconds(1));

distributedLockExecutor.execute(this::doPublish, configuration);

}

//.....
}
```

这里，我们使用了分发布锁来处理并发发送的情况，**doPublish()**方法将调用实际的消息队列(比如RabbitMQ/Kafka等)API完成消息发送。更多的代码细节，请参考本文的示例代码。

## 消费领域事件

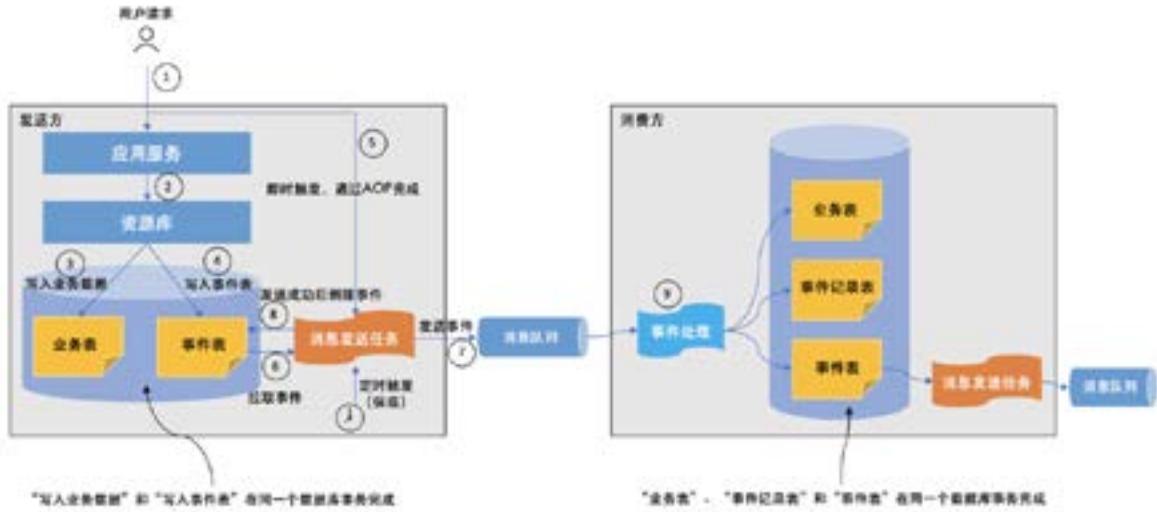
在事件消费时，除了完成基本的消费逻辑外，我们需要重点关注以下两点：

1. 消费方的幂等性
2. 消费方有可能进一步产生事件

对于“消费方的幂等性”，在上文中我们讲到事件的发送机制保证的是“至少一次投递”，为了能够正确地处理重复消息，要求消费方是幂等的，即多次消费事件与单次消费该事件的效果相同。为此，在消费方创建一个事件记录表，用于记录已经消费过的事件，在处理事件时，首先检查该事件是否已经被消费过，如果是则不做任何消费处理。

对于第2点，我们依然沿用前文讲到的事件表的方式。事实上，无论是处理HTTP请求，还是作为消息的消费方，对于聚合根来讲都是无感知的，领域事件由聚合根产生进而由Repository持久化，这些过程都与具体的业务操作源头无关。

综上，在消费领域事件的过程中，程序需要更新业务表、事件记录表以及事件发送表，这3个操作过程属于同一个本地事务，此时整个事件的发布和消费过程如下：



在编码实践时，可以考虑与事件发布过程相同的AOP方式完成对事件的记录，以Spring和RabbitMQ为例，可以将@[RabbitListener](#)通过AOP代理起来：

```

@Aspect
@Component
public class DomainEventRecordingConsumerAspect {
    //.....
    @Around("@annotation(org.springframework.amqp.rabbit.annotation.RabbitHandler) || "+
            "@annotation(org.springframework.amqp.rabbit.annotation.RabbitListener)")
    public Object recordEvents(ProceedingJoinPoint joinPoint) throws Throwable {
        return domainEventRecordingConsumer.recordAndConsume(joinPoint);
    }
    //.....
}

```

然后在代理过程中通过[DomainEventRecordingConsumer](#)完成事件的记录：

```
@Component
public class DomainEventRecordingConsumer {

    //.....
    @Transactional
    public Object recordAndConsume(ProceedingJoinPoint joinPoint) throws Throwable {
        Object[] args = joinPoint.getArgs();
        Optional<Object> optionalEvent = Arrays.stream(args)
            .filter(o -> o instanceof DomainEvent)
            .findFirst();

        if (optionalEvent.isPresent()) {
            DomainEvent event = (DomainEvent) optionalEvent.get();
            try {
                dao.recordEvent(event);
            } catch (DuplicateKeyException dke) {
                logger.warn("Duplicated {} skipped.", event);
                return null;
            }
            return joinPoint.proceed();
        }
        return joinPoint.proceed();
    }
    //.....
}
```

这里的**DomainEventRecordingConsumer**通过直接向事件记录表中插入事件的方式来判断消息是否重复，如果发生重复主键异常**DuplicateKeyException**，即表示该事件已经在记录表中存在了因此直接**return null**;而不再执行业务过程。

需要特别注意的一点是，这里的封装方法**recordAndConsume()**需要打上**@Transactional**注解，这样才能保证对事件的记录和业务处理在同一个事务中完成。

此外，由于消费完毕后也需要即时发送事件，因此需要在发布事件的AOP配置**DomainEventPublishAspect**中加入**@RabbitListener**:

```
@Aspect
@Component
public class DomainEventPublishAspect {

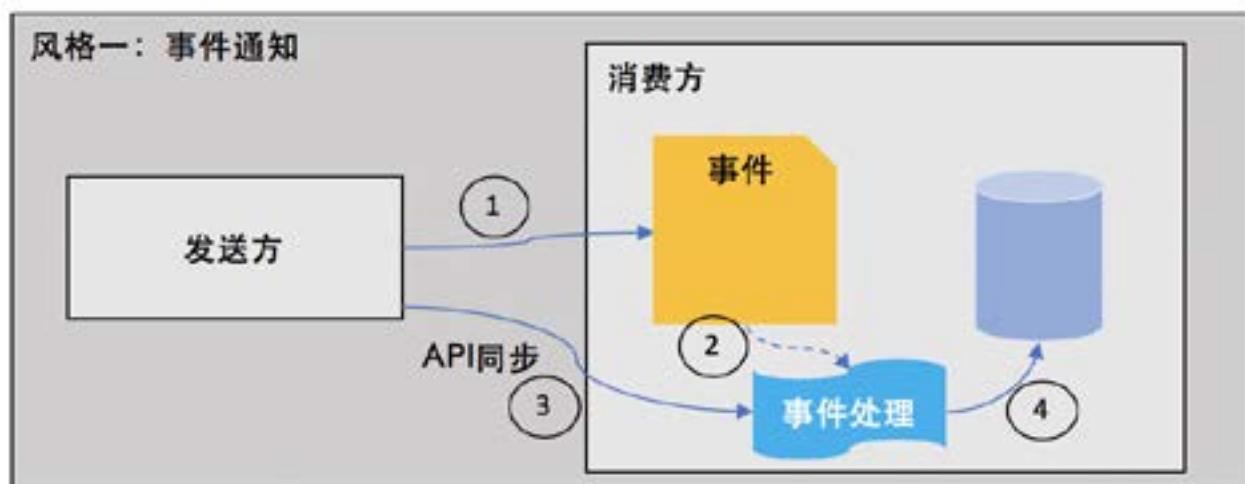
    //.....
    @After( "@annotation(org.springframework.web.bind.annotation.PostMapping) || " +
            "@annotation(org.springframework.web.bind.annotation.PutMapping) || " +
            "@annotation(org.springframework.web.bind.annotation.PatchMapping) || " +
            "@annotation(org.springframework.web.bind.annotation.DeleteMapping) ||" +
            "@annotation(org.springframework.amqp.rabbit.annotation.RabbitListener) ||" )
    public void publishEvents(JoinPoint joinPoint) {
        logger.info( "Trigger domain event publish process." );
        taskExecutor.execute(() -> publisher.publish());
    }
    //.....
}
```

## 事件驱动架构的2种风格

事件驱动架构存在多种风格，本文就其中的2种主要风格展开讨论，它们是：

1. 事件通知
2. 事件携带状态转移(Event-Carried State Transfer)

在“事件通知”风格中，事件只是作为一种信号传递到消费方，消费方需要的数据需要额外API请求从源事件系统获取，如图：

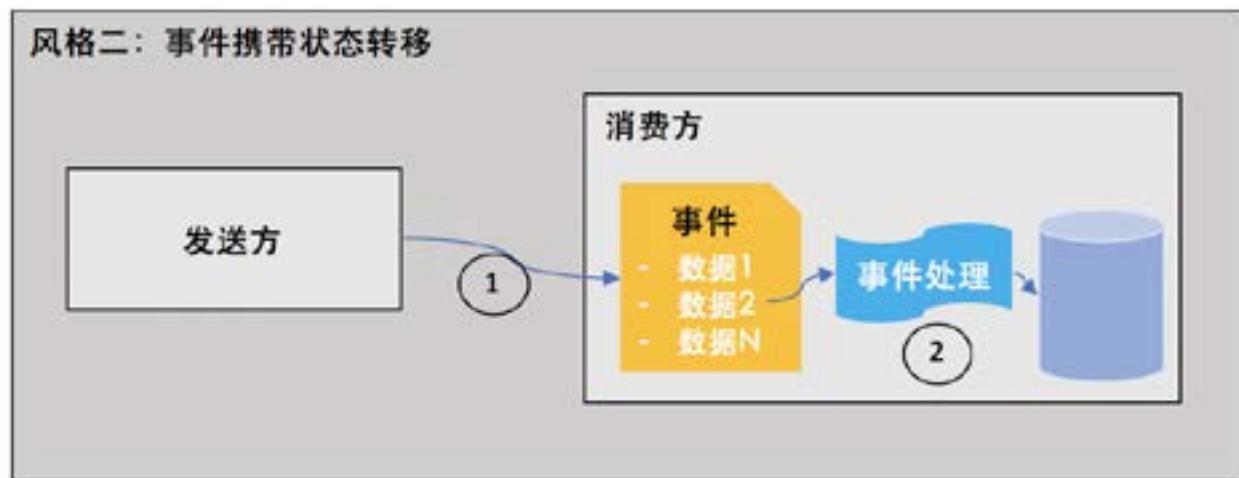


在上图的事件通知风格中，对事件的处理流程如下：

1. 发布方发布事件
2. 消费方接收事件并处理
3. 消费方调用发布方的API以获取事件相关数据
4. 消费方更新自身状态

这种风格的好处是，事件可以设计得非常简单，通常只需要携带聚合根的ID即可，由此进一步降低了事件驱动系统中的耦合度。然而，消费方需要的数据依然需要额外的API调用从发布方获取，这又从另一个角度增加了系统之间的耦合性。此外，如果源系统宕机，消费方也无法完成后续操作，因此可用性会受到影响。

在“事件携带状态转移”中，消费方所需要的数据直接从事件中获取，因此不需要额外的API请求：



这种风格的好处在于，即便发布方系统不可用，消费方依然可以完成对事件的处理。

笔者的建议是，对于发布方来说，作为一种数据提供者的“自我修养”，事件应该包含足够多的上下文数据，而对于消费方来讲，可以根据自身的实际情况确定具体采用哪种风格。在同一个系统中，同时采用2种风格是可以接受的。比如，对于基于事件的CQRS而言，可以采用“事件通知”，此时的事件只是一个“触发器”，一个聚合下的所有事件所触发的结果是一样的，即都是告知消费方需要从源系统中同步数据，因此此时的消费方可以对聚合下的所有事件一并处理，而不用为每一种事件单独开发处理逻辑。

事实上，事件驱动还存在第3种风格，即事件溯源，本文不对此展开讨论。更多有关事件驱动架构不同风格的介绍，请参考Martin Fowler的“事件风格”文章。

## 第二部分：基于RabbitMQ的示例项目

领域事件是DDD中的一个概念，表示的是在一个领域中所发生的一次对业务有价值的事情，落到技术层面就是在本部分中，我将以一个简单的电商平台微服务系统为例，采用RabbitMQ作为消息机制讲解事件驱动架构落地的全过程。

该电商系统包含3个微服务，分别是：

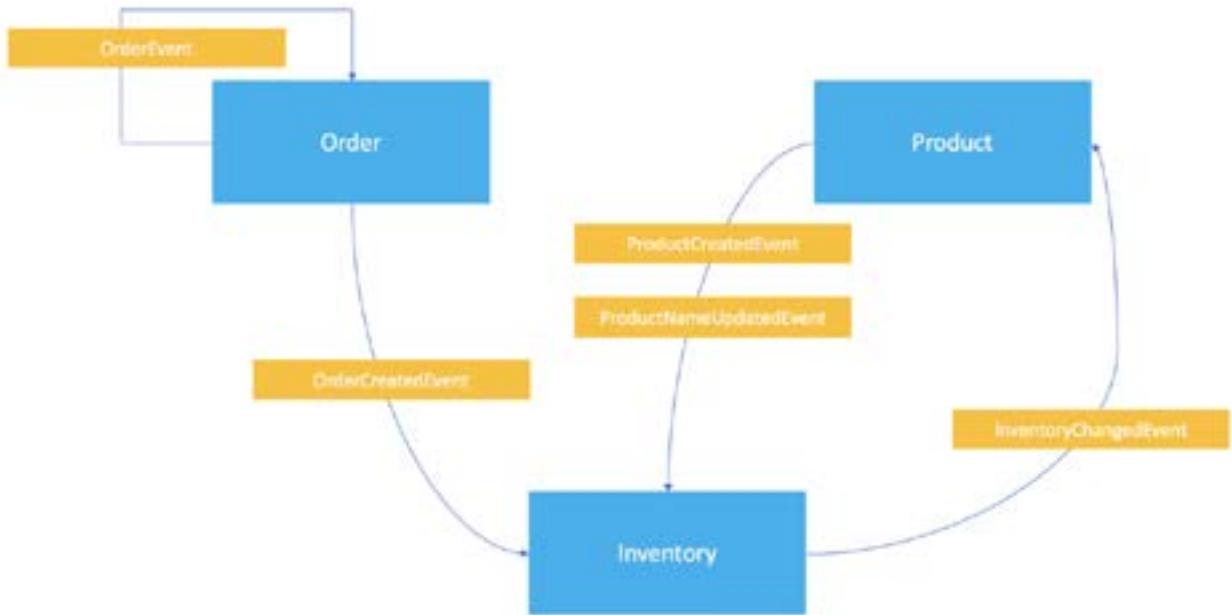
- 订单(Order)服务: 用于用户下单
- 产品(Product)服务: 用于管理/展示产品信息
- 库存(Inventory)服务: 用于管理产品对应的库存

整个系统包含以下代码库:

代码库	用途	地址
order-backend	Order服务	<a href="https://github.com/e-commerce-sample/order-backend">https://github.com/e-commerce-sample/order-backend</a>
product-backend	Product服务	<a href="https://github.com/e-commerce-sample/product-backend">https://github.com/e-commerce-sample/product-backend</a>
inventory-backend	Inventory服务	<a href="https://github.com/e-commerce-sample/inventory-backend">https://github.com/e-commerce-sample/inventory-backend</a>
common	共享依赖包	<a href="https://github.com/e-commerce-sample/common">https://github.com/e-commerce-sample/common</a>
devops	基础设施	<a href="https://github.com/e-commerce-sample/devops">https://github.com/e-commerce-sample/devops</a>

其中，**common**代码库包含了所有服务所共享的代码和配置，包括所有服务中的所有事件（请注意，这种做法只是笔者为了编码上的便利，并不是一种好的实践，一种更好的实践是各个服务各自管理自身产生的事件），以及RabbitMQ的通用配置（即每个服务都采用相同的方式配置RabbitMQ设施），同时也包含了异常处理和分布式锁等配置。**devops**库中包含了RabbitMQ的Docker镜像，用于在本地测试。

整个系统中涉及到的领域事件如下：



整个系统中涉及到的领域事件如下：

- Order服务自己消费了自己产生的所有**OrderEvent**用于CQRS同步读写模型；
- Inventory服务消费了Order服务的**OrderCreatedEvent**事件，用于在下单之后即时扣减库存；
- Inventory服务消费了Product服务的**ProductCreatedEvent**和**ProductNameChangedEvent**事件，用于同步产品信息；
- Product服务消费了Inventory服务的**InventoryChangedEvent**用于更新产品库存。

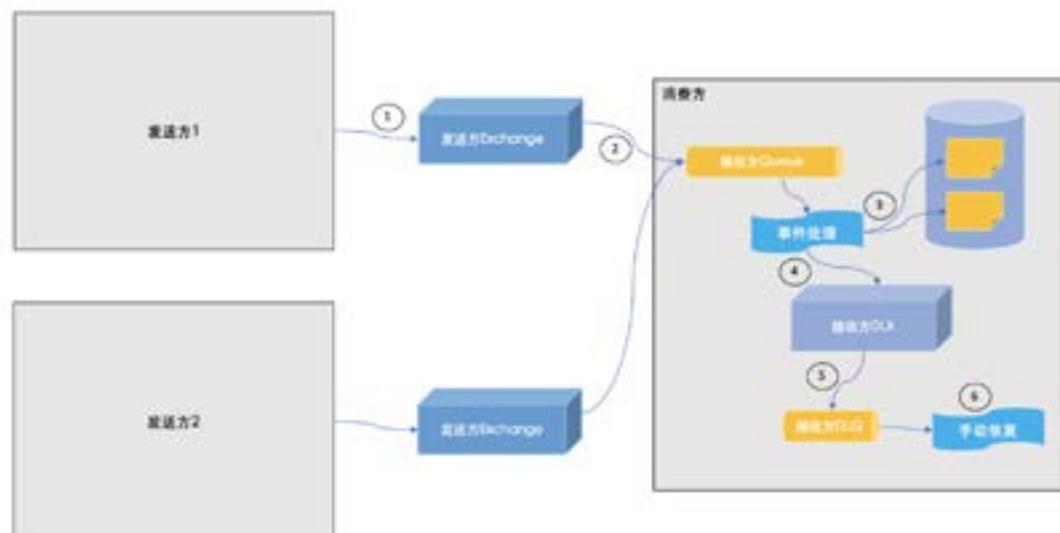
## 配置RabbitMQ

阅读本小节需要熟悉RabbitMQ中的基本概念，建议不熟悉RabbitMQ的读者事先参考[RabbitMQ入门文章](#)。

这里介绍2种RabbitMQ的配置方式，一种简单的，一种稍微复杂的。两种配置过程中会反复使用到以下概念，读者可以先行熟悉：

概念	概念		概念	概念
发送方 Exchange	Exchange	用于接收某个微服务中所有消息的Exchange, 一个服务只有一个 <b>发送方Exchange</b>	xxx-publish-x	order-publish-x
发送方 DLX	Exchange	用于接收发送方无法路由的消息	xxx-publish-x-dlx	order-publish-x-dlx
发送方 DLQ	Queue	用于存放 <b>发送方DLX</b> 的消息	xxx-publish-x-dlp	order-publish-x-dlq
接受方 Queue	Queue	用于接收发送方 <b>Exchange</b> 的消息, 一个服务只有一个 <b>接收方Queue</b> 用于接收所有外部消息	xxx-receive-q	product-receive-q
接受方 DLX	Exchange	死信Exchange, 用于接收消费失败的消息	xxx-receive-dlx	product-receive-dlx
接受方 DLQ	Queue	死信队列, 用于存放 <b>接收方DLX</b> 的消息	xxx-receive-dlq	product-receive-dlq
接受方恢复 Exchange	Exchange	用于接收从 <b>接收方DLQ</b> 中手动恢复的消息, <b>接收方Queue</b> 应该绑定到 <b>接收方恢复Exchange</b>	xxx-receive-recover-x	product-receive-recover-x

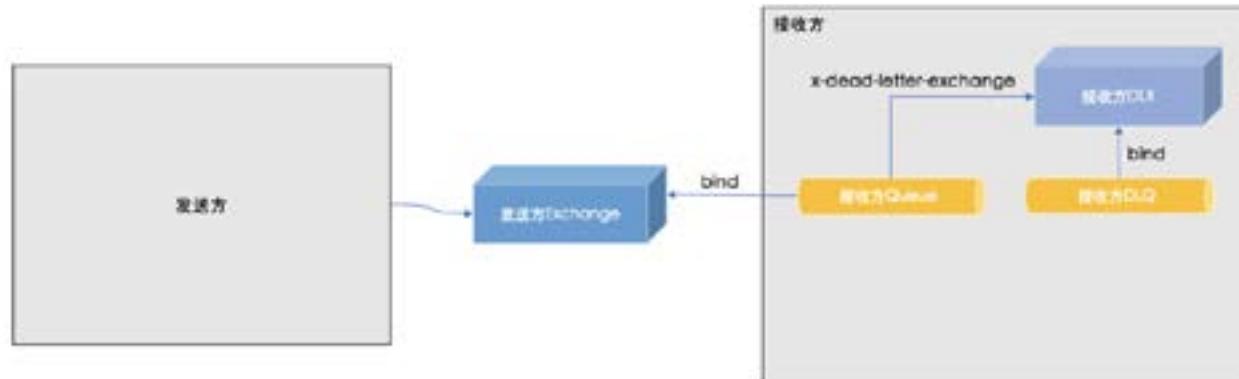
在简单配置方式下, 消息流向图如下:



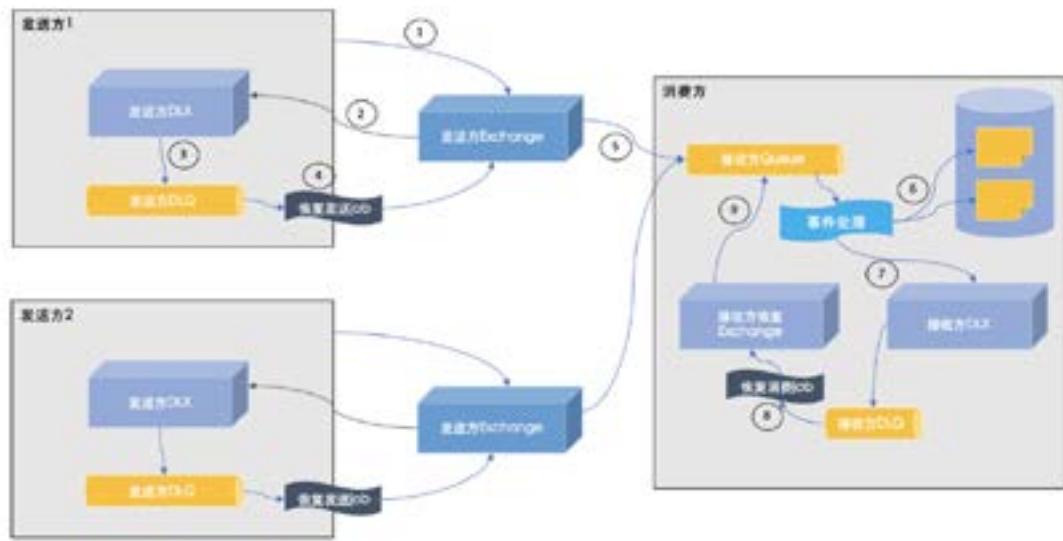
1. 发送方发布事件到发送方Exchange。
2. 消息到达消费方的接收方Queue。
3. 消费成功处理消息，更新本地数据库。
4. 如果消息处理失败，消息被放入接收方DLX。
5. 消息到达死信队列接收方DLQ。
6. 对死信消息做手工处理（比如作日志记录等）。

对于发送方而言，事件驱动架构提倡的是“发送后不管”机制，即发送方只需要保证事件成功发送即可，而不用关心是谁消费了该事件。因此在配置发送方的RabbitMQ时，可以简单到只配置一个发送方Exchange即可，该Exchange用于接收某个微服务中所有类型的事件。在消费方，首先配置一个接收方Queue用于接收来自所有发送方Exchange的所有类型的事件，除此之外对于消费失败的事件，需要发送到接收方DLX，进而发送到接收方DLQ中，对于接收方DLQ的事件，采用手动处理的形式恢复消费。

在简单方式下的RabbitMQ配置如下：

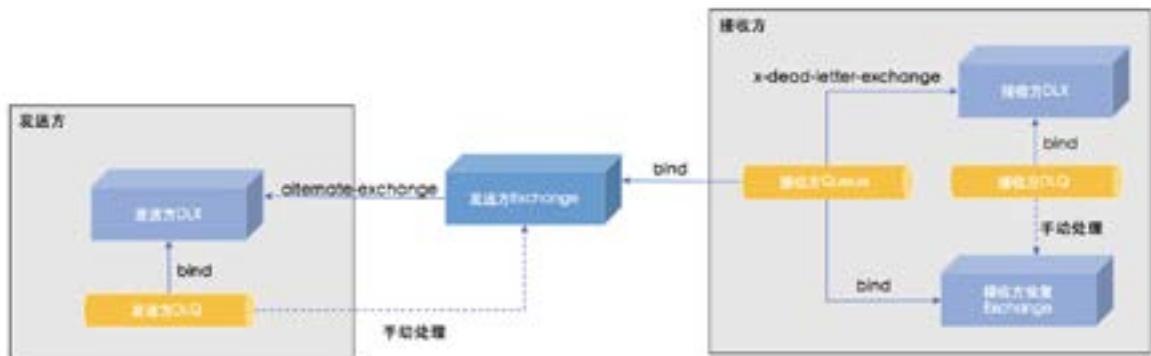


在第2种配置方式稍微复杂一点，其建立在第1种基础之上，增加了发送方的死信机制以及消费方用于恢复消费的Exchange，此时的消息流向如下：



1. 发送方发布事件。
2. 事件发布失败时被放入死信Exchange**发送方DLX**。
3. 消息到达死信队列**发送方DLQ**。
4. 对于发送方DLQ中的消息进行人工处理，重新发送。
5. 如果事件发布正常，则会到达**接收方Queue**。
6. 正常处理事件，更新本地数据库。
7. 事件处理失败时，发到**接收方DLX**，进而路由到**接收方DLQ**。
8. 手工处理死信消息，将其发到**接收方恢复Exchange**，进而重新发到接收方**Queue**。

此时的RabbitMQ配置如下：



在以上2种方式中，我们都启用了RabbitMQ的“发送方确认”和“消费方确认”，另外，发送方确认也可以通过 RabbitMQ的事务(不是分布式事务)替代，不过效率更低。更多关于RabbitMQ的知识，可以参考笔者的Spring AMQP学习笔记和RabbitMQ最佳实践。

## 系统演示

- 启动RabbitMQ，切换到[ecommerce-sample/devops/local/rabbitmq](#)目录，运行：

```
./start-rabbitmq.sh
```

- 启动Order服务：切换到[ecommerce-sample/order-backend](#)项目，运行：

```
./run.sh //监听8080端口，调试5005端口
```

- 启动Product服务：切换到[ecommerce-sample/product-backend](#)项目，运行：

```
./run.sh //监听8082端口，调试5006端口
```

- 启动Inventory服务：切换到[ecommerce-sample/inventory-backend](#)项目，运行：

```
./run.sh //监听8083端口，调试5007端口
```

- 创建Product：

```
curl -X POST \
http://localhost:8082/products \
-H ‘Content-Type: application/json’ \
-H ‘cache-control: no-cache’ \
-d ‘{
    “name” :”好吃的苹果”，
    “description” :”原生态的苹果”，
    “price” :10.0
}’
```

此时返回Product ID:

```
{ "id" : "3c11b3f6217f478fbdb486998b9b2fee" }
```

- 查看Product:

```
curl -X GET \
http://localhost:8082/products/3c11b3f6217f478fbdb486998b9b2fee \
-H 'cache-control: no-cache'
```

返回如下:

```
{
  "id": {
    "id": "3c11b3f6217f478fbdb486998b9b2fee"
  },
  "name": "好吃的苹果",
  "price": 10,
  "createdAt": 1564361781956,
  "inventory": 0,
  "description": "原生态的苹果"
}
```

可以看到，新创建的Product的库存([inventory](#))默认为0。

- 创建Product时，会创建ProductCreatedEvent，Inventory服务接收到该事件后会自动创建对应的Inventory，日志如下：

```
2019-07-29 08:56:22.276 -- INFO [taskExecutor-1] c.e.i.i.InventoryEventHandler
: Created inventory[5e3298520019442b8a6d97724ab57d53] for
product[3c11b3f6217f478fbdb486998b9b2fee].
```

- 增加Inventory为10:

```
curl -X POST \
  http://localhost:8083/inventories/5e3298520019442b8a6d97724ab57d53/increase \
-H 'Content-Type: application/json' \
-H 'cache-control: no-cache' \
-d '{ \
  "increaseNumber":10 \
}'
```

- 增加Inventory之后，会发送InventoryChangedEvent，Product服务接收到该事件后会自动同步自己的库存，再次查看Product:

```
curl -X GET \
  http://localhost:8082/products/3c11b3f6217f478fdbdb486998b9b2fee \
-H 'cache-control: no-cach'
```

返回如下:

```
{ \
  "id":{ \
    "id": "3c11b3f6217f478fdbdb486998b9b2fee" \
  }, \
  "name": "好吃的苹果", \
  "price": 10, \
  "createdAt": 1564361781956, \
  "inventory": 10, \
  "description": "原生态的苹果" \
}
```

可以看到，Product的库存已经更新为10。

- 至此，Product和Inventory都准备好了，让我们下单吧：

```
curl -X POST \
http://localhost:8080/orders \
-H 'Content-Type: application/json' \
-H 'cache-control: no-cache' \
-d '{
  "items": [
    {
      "productId": "3c11b3f6217f478fdbb486998b9b2fee",
      "count": 2,
      "itemPrice": 10
    }
  ],
  "address": {
    "province": "四川",
    "city": "成都",
    "detail": "天府软件园1号"
  }
}'
```

返回Order ID：

```
{
  "id": "d764407855d74ff0b5bb75250483229f"
}
```

- 创建订单之后，会发送OrderCreatedEvent，Inventory服务接收到该事件会自动扣减相应库存：

```
2019-07-29 09:11:31.202 -- INFO [taskExecutor-1] c.e.i.i.InventoryEventHandler  
:Inventory[5e3298520019442b8a6d97724ab57d53] decreased to 8 due to  
order[d764407855d74ff0b5bb75250483229f] creation.
```

同时，Inventory将发送InventoryChangedEvent，Product服务接收到该事件会自动更新Product的库存，再次查看Product：

```
curl -X GET \  
http://localhost:8082/products/3c11b3f6217f478fbdb486998b9b2fee \  
-H 'cache-control: no-cache'
```

返回如下：

```
{  
  "id": {  
    "id": "3c11b3f6217f478fbdb486998b9b2fee"  
  },  
  "name": "好吃的苹果",  
  "price": 10,  
  "createdAt": 1564361781956,  
  "inventory": 8,  
  "description": "原生态的苹果"  
}
```

可以看到，Product的库存从10减少到了8，因为先前下单时我们选了2个Product。

## 总结

本文首先独立于消息队列的技术实现，讲到了事件驱动架构在落地过程中的诸多方面以及问题，包括领域事件的建模、通过聚合根暂存事件然后由Repository完成存储，再由后台任务读取事件表完成事件的实际发布。在消费方，通过幂等性解决在“至少一次投递”的情况下所带来的重复消费问题。另外，还讲到了事件驱动架构的2种常见风格，即事件通知和事件携带状态转移，以及他们之间的优劣势。在第二部分，以RabbitMQ为例，分享了如何在一个微服务化的系统中落地事件驱动架构。

# 后端开发实践系列：简单可用的CQRS编码实践

作者：滕云

本文只讲了一件事情：软件模型中存在读模型和写模型之分，CQRS便为此而生。

20多年前，Bertrand Meyer在他的《Object-Oriented Software Construction》一书中提出了CQS (Command Query Separation, 命令查询分离) 的概念，指出：

Every method should either be a command that performs an action, or a query that returns data to the caller, but never both. (一个方法要么作为一个“命令”执行一个操作，要么作为一次“查询”向调用方返回数据，但两者不能共存。)

这里的“命令”可以理解为更新软件状态的写操作，Martin Fowler将此称为“Modifier”；而“查询”即为读操作，是无副作用的。这种分离的好处在于使程序变得更容易推理与维护，由于查询操作不会更新软件状态，在编码时我们将更加有信心。试想，如果程序中出了一个bug，如果这个bug出现在查询过程中，那么我们至少可以消除这个bug可能给软件带来脏数据的恐惧。

后来，Greg Young在此基础上提出了CQRS (Command Query Responsibility Segregation, 命令查询职责分离)，将CQS的概念从方法层面提升到了模型层面，即“命令”和“查询”分别使用不同的对象模型来表示。

采用CQRS的驱动力除了从CQS那里继承来的好处之外，还旨在解决软件中日益复杂的查询问题，比如有时我们希望从不同的维度查询数据，或者需要将各种数据进行组合后返回给调用方。此时，将查询逻辑与业务逻辑糅合在一起会使软件迅速腐化，诸如逻辑混乱、可读性变差以及可扩展性降低等等一些列问题。

## 一个例子

设想电商系统中的订单 (Order) 对象，一开始其对应的OrderRepository类可以简单到只包含2个方法：

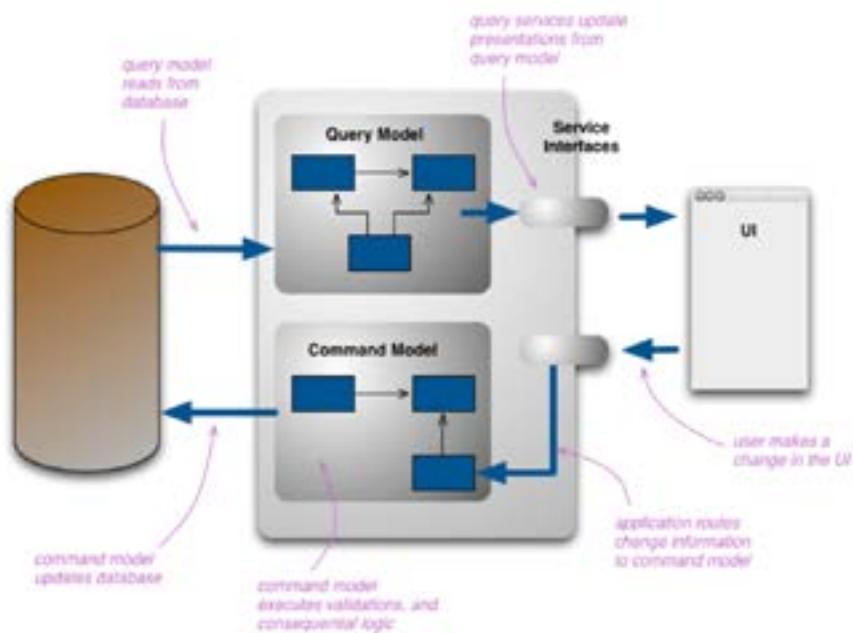
```
public interface OrderRepository {  
    void save(Order order);  
    Order getById(String id);  
}
```

在项目的演进中，你可能需要依次实现以下需求：

1. 查询某个Order详情，详情中不用包含Order的某些字段；
2. 查询Order列表，列表中所展示的数据比Order详情更少；
3. 根据时间、类别和金额等多种筛选条件查询Order列表；
4. 展示Order中的产品（Product）概要信息，而Product属于另一个业务实体；
5. 展示Order下单人的昵称，下单人信息属于另一个单独的账户系统，用户修改昵称之后，Order下单人昵称也需要相应更新；
6. .....

当这些需求实现完后，你可能会发现OrderRepository和领域模型已经被各种“查询”功能淹没了。什么？OrderRepository不是给领域模型提供Order聚合根对象的吗，为什么却充斥着如此多的查询逻辑？

CQRS通过单独的读模型解决上述问题，其大致的架构图如下：



对于Command侧，主要的讲究是将业务用例建模成对应的Command对象，然后在对Command的处理流程中应用核心的业务逻辑，其中最重要的是领域模型的建模，关于此的内容请参考笔者的《领域驱动设计(DDD)编码实践》文章，本文着重介绍Query侧的编码实践。

在本文中，查询模型（Query Model）也被表达为读模型（Read Model）；命令模型（Command Model）也被表达为写模型（Write Model）。

# CQRS实现模式概览

## 常见误解

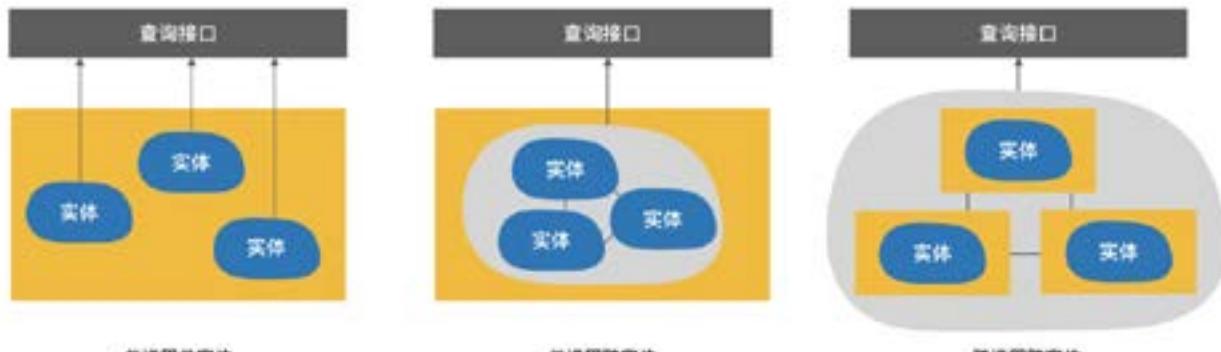
在网上搜索一番，你会发现很多关于CQRS的文章都将CQRS与Event Sourcing（事件溯源）结合起来使用，这容易让人觉得采用CQRS就一定需要同时使用Event Sourcing，事实上这是一种误解。CQRS究其本意只是要求“读写模型的分离”，并未要求使用Event Sourcing；再者，Event Sourcing会极大地增加软件的复杂度，而本文追求的是“简单可用的CQRS”，因此本文将不会涉及Event Sourcing相关内容。更多内容，请参考简化版CQRS的文章。

另外需要指出的是，读写模型的分离并不一定意味着数据存储的分离，不过在实际应用中，数据存储分离是一种常见的CQRS实践模式，在这种模式中，写模型的数据会同步到读模型数据存储中，同步过程通常通过消息机制完成，在DDD场景下，消息通常承载的是领域事件（Domain Event）。

## 查询模型的数据来源

无论是单体还是微服务，所读数据的唯一正确来源（Single Source of Truth）最终都来自于业务实体（Entity）对象（比如DDD中的聚合根），基于此，所读数据的来源形式大致分为以下几种：

- 所读数据来源于同一个进程空间的单个实体（后文简称“**单进程单实体**”），这里的进程空间指某个单体应用或者单个微服务；
- 所读数据来源于同一个进程空间中的多个实体（后文简称“**单进程跨实体**”）；
- 所读数据来源于不同进程空间中的多个实体（后文简称“**跨进程跨实体**”）。

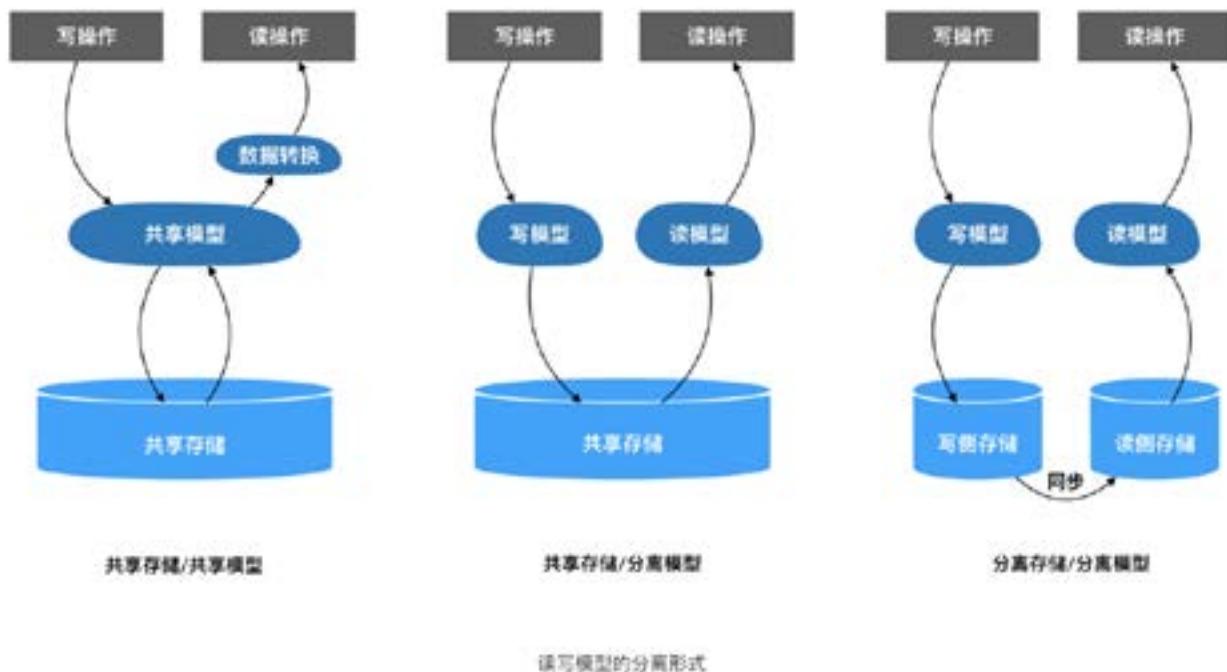


查询模型的数据来源

## 读写模型的分离形式

CQRS中的读写分离存在2个层次，一层是代码中的模型是否需要分离，另一层是数据存储是否需要分离，总结下来有以下几种：

- **共享存储/共享模型**: 读写模型共享数据存储（即同一个数据库），同时也共享代码模型，数查询据通过模型转换后返回给调用方，事实上这不能算CQRS，但是对于很多中小型项目而言已经足够；
- **共享存储/分离模型**: 共享数据存储，代码中分别建立写模型和读模型，读模型通过最适合于查询的方式进行建模；
- **分离存储/分离模型**: 数据存储和代码模型都是分离的，这种方式通常用于需要聚合查询多个子系统的情况，比如微服务系统。



将以上“查询模型的数据来源”与“读写模型的分离形式”相组合，我们可以得到以下不同的CQRS模式及其适用范围：

数据来源形式	模型分离形式	适用范围
单进程单实体	共享存储/共享模型	其实算不上CQRS, 但对于很多中小型项目已经足够
单进程单实体	共享存储/分离模型	适用于单实体查询比较复杂或者对查询效率要求较高的场景
单进程单实体	不同存储/分离模型	适用于对单个实体的查询非常复杂的场景
单进程跨实体	共享存储/共享模型	不适用
单进程跨实体	共享存储/分离模型	适用于查询比较复杂的场景, 比如需要做多表join操作
单进程跨实体	分离存储/分离模型	适用于复杂查询或者对查询效率要求较高的情况
跨进程跨实体	共享存储/共享模型	不适用
跨进程跨实体	共享存储/分离模型	不适用
跨进程跨实体	分离存储/分离模型	主要用于微服务中需要对多个服务进行聚合查询的场景

总结下来，有以下几种常见做法：

- 单进程单实体 + 共享存储/共享模型
- 单进程单实体 + 共享存储/分离模型
- 单进程跨实体 + 共享存储/分离模型
- 单进程跨实体 + 分离存储/分离模型
- 跨进程跨实体 + 分离存储/分离模型

接下来，针对以上几种常见做法，本文将依次给出编码示例。

## CQRS编码实践

本文的示例是一个简单的电商系统，其中包含以下微服务：

服务	用途	所含实体	Git地址
订单服务	用于用户下单	Order	ecommerce-order-service
订单查询服务	用于订单的CQRS查询操作	无	ecommerce-order-query-service
产品服务	用于管理/展示产品信息	Product Category(产品目录)	ecommerce-product-service
库存服务	用于管理产品对应的库存	Inventory	ecommerce-inventory-service

示例代码请参考：

<https://github.com/e-commerce-sample>

请注意，本文的示例电商项目只是一个虚构出来的简单项目，仅仅用于演示CQRS的各种编码模式，并不具备实际参考价值。

针对以上各种CQRS模式组合，本文将使用电商系统中的以下业务用例进行演示：

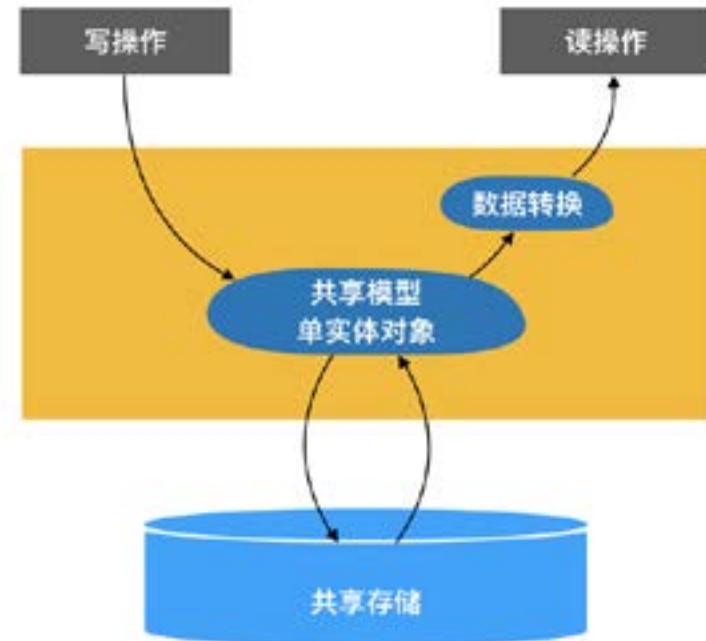
针对以上各种CQRS模式组合，本文将使用电商系统中的以下业务用例进行演示：

CQRS模式	业务查询用例	所属服务
单进程单实体 + 共享存储/共享模型	Inventory详情查询	库存服务
单进程单实体 + 共享存储/分离模型	Product摘要查询	产品服务
单进程跨实体 + 共享存储/分离模型	Product详情查询 (包含Category信息)	产品服务
单进跨单实体 + 分离存储/分离模型	Product详情查询 (包含Category信息)	产品服务
跨进程跨实体 + 分离存储/分离模型	Order详情查询 (包含Product信息)	订单查询服务

## 1. 单进程单实体 + 共享存储/共享模型

对于简单的单体或者微服务应用，这种方式是最自然最直接的方式，事实上我们并不需要太多设计上的思考便能想到这种方式。在这种方式中，存在单个领域实体模型同时用于读写操作，在向调用方返回查询数据时，需要针对性地对领域模型进行转换，转换的目的在于：

- 调用方所需的数据模型与领域模型可能不一致；
- 有些敏感信息是不能返回给调用方的，需要屏蔽；
- 从设计上讲，领域模型不能直接返回给调用方，否则会产生领域模型的泄露
- 将领域模型直接返回给调用方会在领域模型与对外接口间产生强耦合，不利于领域模型自身的演进。



单进程单实体 + 共享存储/共享模型

这里，我们以“库存(Inventory)详情查询”为例进行演示，[Inventory](#)领域模型定义如下：

```
public class Inventory{  
    private String id;  
    private String productId;  
    private String productName;  
    private int remains;  
    private Instant createdAt;  
}
```

在获取Inventory详情时，我们并不需要返回领域模型中的productId和createdAt字段，于是在Inventory中创建相应的转换方法如下：

```
public InventoryRepresentation toRepresentation() {  
    return new InventoryRepresentation(this.id,  
        this.productName,  
        this.remains);  
}
```

这里的InventoryRepresentation即表示读模型，后缀Representation取自REST中的“R”，表示读模型是一种数据展现，下文将沿用这种命名形式。在InventoryApplicationService服务中返回InventoryRepresentation：

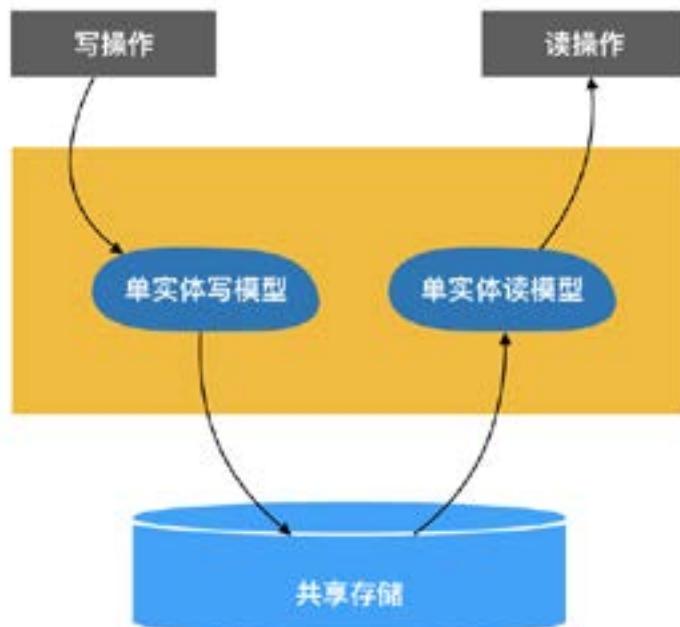
```
public InventoryRepresentation byId(String inventoryId) {  
    return repository  
        .byId(inventoryId)  
        .toRepresentation();  
}
```

值得一提的是，在查询Inventory时，我们使用了应用服务（ApplicationService）-InventoryApplicationService，此时的InventoryApplicationService同时承担了读操作和写操作的业务入口，在实践中也可以将此二者分离开来，即让InventoryApplicationService只负责写操作，而另行创建InventoryRepresentationService专门用于读操作。

另外，抛开CQRS，为了保证每一个聚合根实体自身的完备性，即便在没有调用方查询的情况下，笔者也建议为每一个聚合根提供一个Representation并对外暴露查询接口。因此每一个聚合根中都会有一个toRepresentation()方法，该方法仅仅返回当前聚合根的状态，而不会关联其他实体对象（比如下文提到的“单进程跨实体”）。

## 2. 单进程单实体 + 共享存储/分离模型

有时，即便是对于单个实体，其查询也会变得复杂，为了维护读写过程彼此的清晰性，我们可以对读模型和写模型分别建模，事实上这也是CQRS的本意。



单进程单实体 + 共享存储/分离模型

在Product服务中，需要返回Product的摘要信息，并对返回列表进行分页处理，为此独立于ApplicationService创建**ProductRepresentationService**，直接从数据库读取数据构建**ProductSummaryRepresentation**。

```
@Transactional(readOnly = true)

    public PagedResource<ProductSummaryRepresentation> listProducts(int pageIndex, int
pageSize) {

        MapSqlParameterSource parameters = new MapSqlParameterSource();
        parameters.addValue( "limit" , pageSize);
        parameters.addValue( "offset" , (pageIndex - 1) * pageSize);

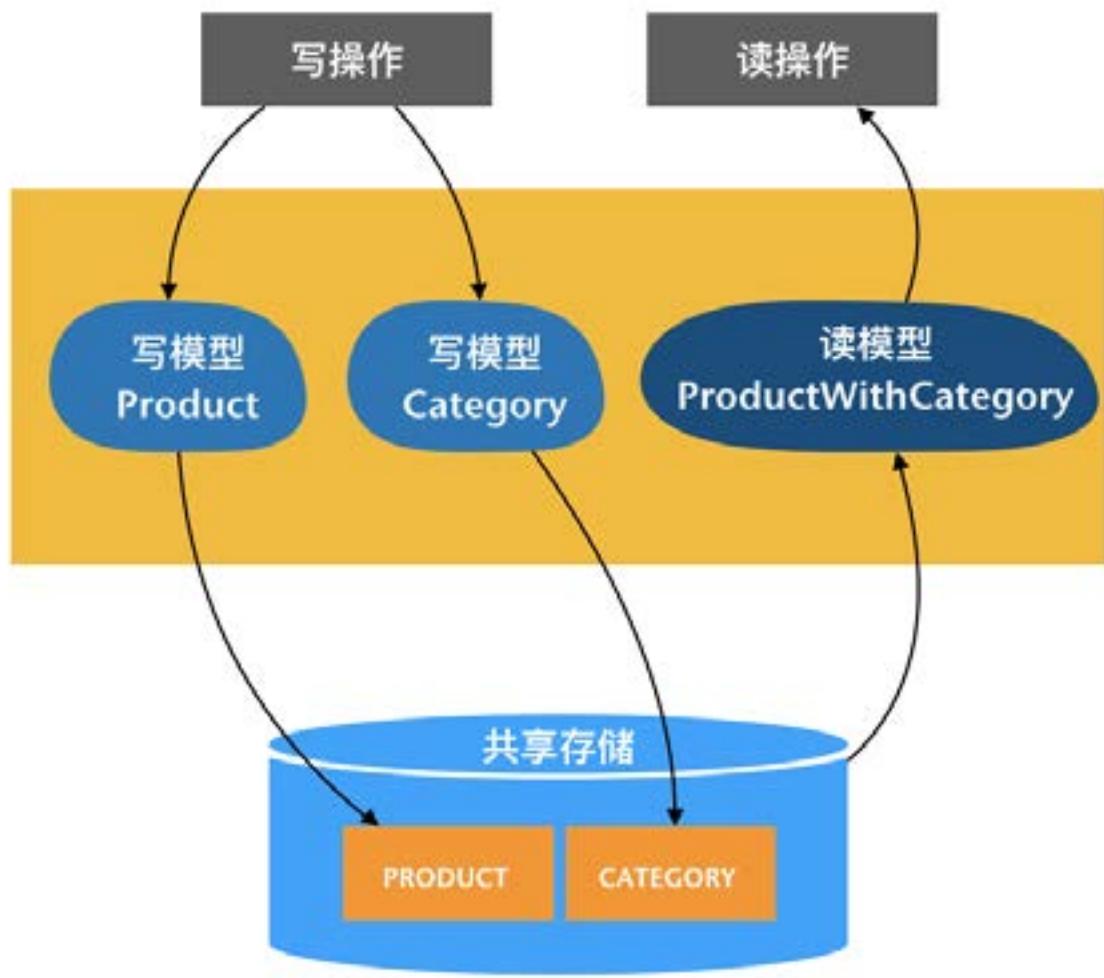
        List<ProductSummaryRepresentation> products = jdbcTemplate.query(SELECT_SQL,
parameters,
        (rs, rowNum) -> new ProductSummaryRepresentation(rs.getString( "ID" ),
rs.getString( "NAME" ),
rs.getBigDecimal( "PRICE" )));

        int total = jdbcTemplate.queryForObject(COUNT_SQL, newHashMap(), Integer.class);
        return PagedResource.of(total, pageIndex, products);
    }
}
```

这里，我们绕过了领域模型Product，也绕过了其对应的ProductRepository，以最快速的方式从数据库中直接获取数据。

### 3. 单进程跨实体 + 共享存储/分离模型

既然单个实体都有必要使用分离模型，那么在同一个进程空间中的跨实体查询更有理由使用分离模型的形式。对于简单形式跨实体查询，还用不着使用分离的存储，只需要做一些join联合查询即可。



### 单进程跨实体 + 共享存储/分离模型

在[Product](#)服务中，存在[Product](#)和[Category](#)两个聚合根对象， 在查询[Product](#)时，我们希望一并带上[Category](#)的信息，为此创建[ProductWithCategoryRepresentation](#)如下：

```
@Value  
public class ProductWithCategoryRepresentation {  
    private String id;  
    private String name;  
    private String categoryId;  
    private String categoryName;  
}
```

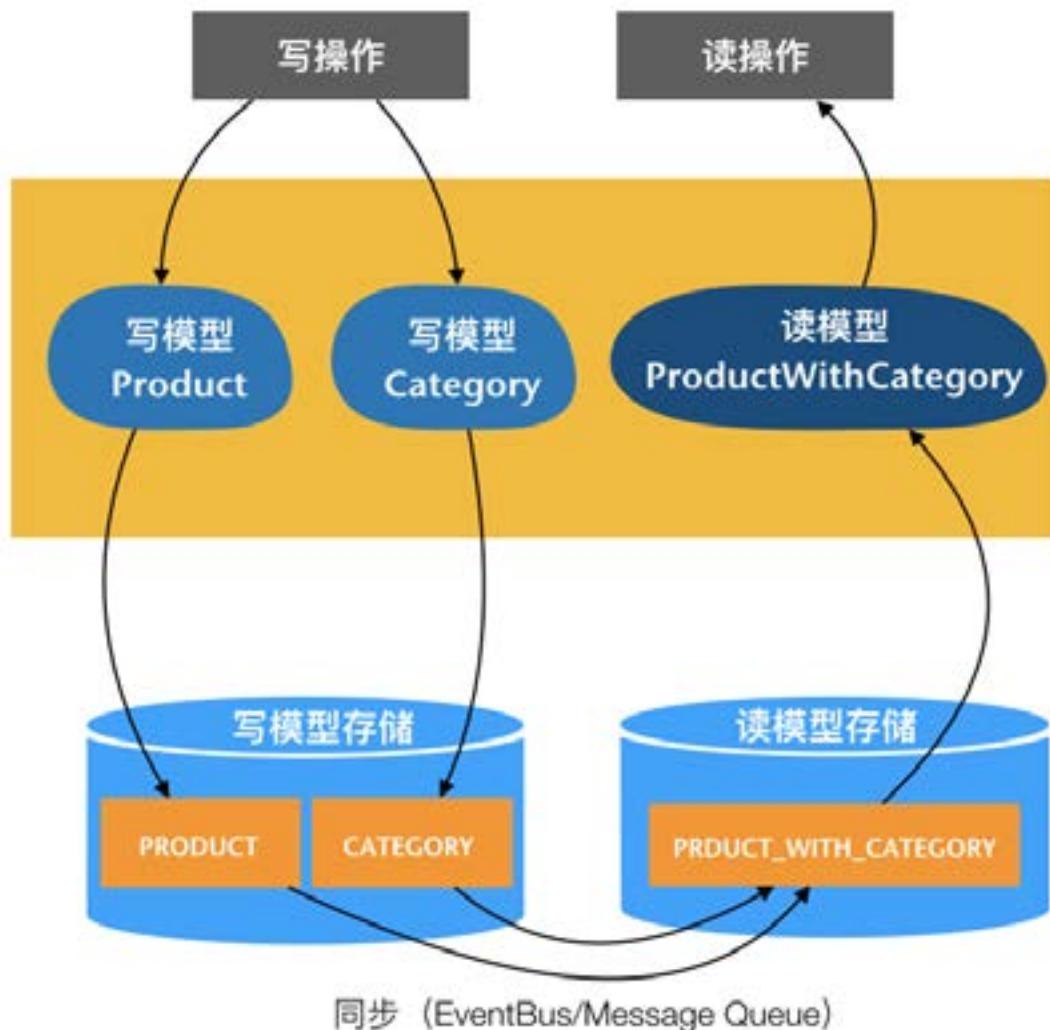
在ProductRepresentationService中，直接从数据库获取**Product**和**Category**数据，此时需要对**PRODUCT**和**CATEGORY**两张表做join操作：

```
@Transactional(readOnly = true)  
public ProductWithCategoryRepresentation productWithCategory(String id) {  
    String sql = "SELECT PRODUCT.ID, PRODUCT.NAME, CATEGORY.ID AS CATEGORY_ID,  
    CATEGORY.NAME AS CATEGORY_NAME FROM PRODUCT JOIN CATEGORY ON PRODUCT.  
    CATEGORY_ID=CATEGORY.ID WHERE PRODUCT.ID=:productId;" ;  
    return jdbcTemplate.queryForObject(sql, of(“productId”, id),  
        (rs, rowNum) -> new ProductWithCategoryRepresentation(rs.getString(“ID”),  
            rs.getString(“NAME”),  
            rs.getString(“CATEGORY_ID”),  
            rs.getString(“CATEGORY_NAME”)));  
}
```

需要注意的是，如果join的级联太多，那么会大大影响查询的效率，并且使程序变得更加复杂。一般来讲，如果join次数达到了3次及其以上，建议考虑采用分离存储的形式。

#### 4. 单进程跨实体 + 分离存储/分离模型

依然以返回[ProductWithCategoryRepresentation](#)为例，假设我们认为先前的join操作太复杂或者太低效了，需要采用专门的数据库来简化查询提升效率。



为此创建单独的读模型数据库表**PRODUCT\_WITH\_CATEGORY**:

```
CREATE TABLE PRODUCT_WITH_CATEGORY
(
    PRODUCT_ID  VARCHAR(32) NOT NULL,
    PRODUCT_NAME VARCHAR(100) NOT NULL,
    CATEGORY_ID  VARCHAR(32) NOT NULL,
    CATEGORY_NAME VARCHAR(100) NOT NULL,
    PRIMARY KEY (PRODUCT_ID)
) CHARACTER SET utf8mb4
    COLLATE utf8mb4_unicode_ci;
```

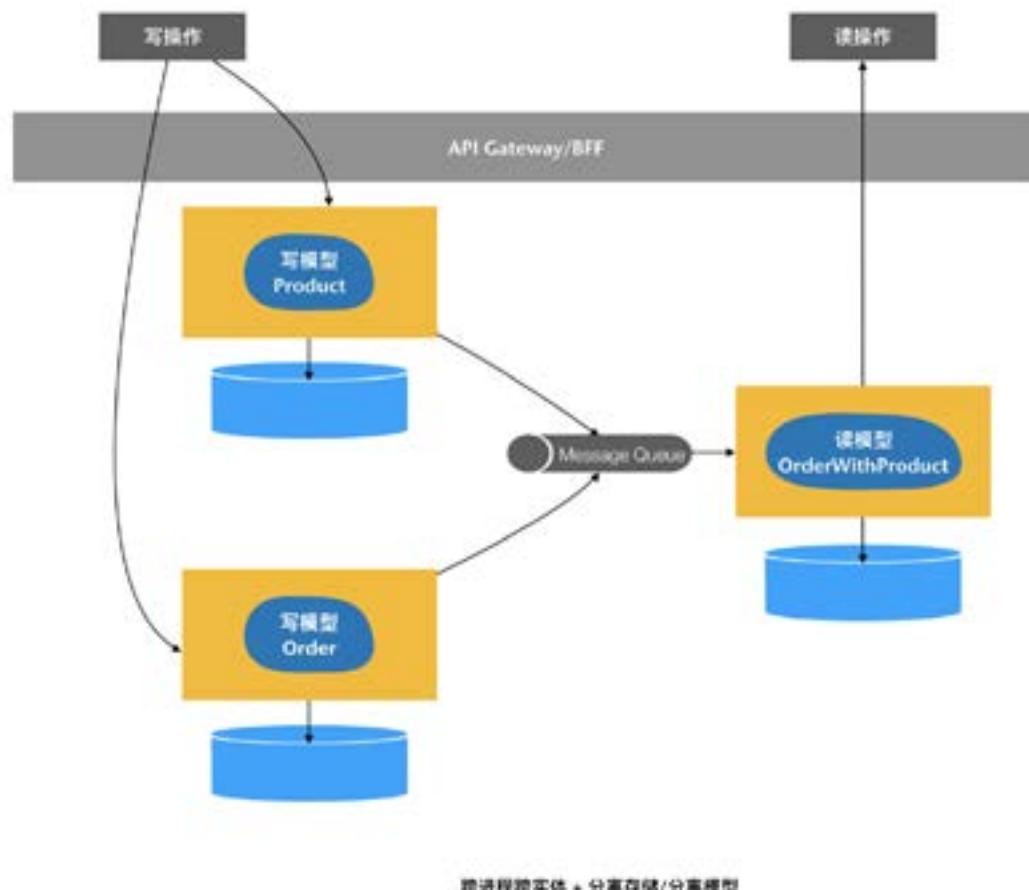
读写同步通常通过领域事件的形式完成，由于是在同一个进程空间中，因此读写同步相比于跨进程的同步来说，可以有更多的选择：

- 使用进程内事件机制（比如Guava的EventBus），在与写操作相同的事务中同步，这种方式的好处是可以保证写操作与同步操作的原子性进而确保读写间的数据一致性，缺点是在写操作过程中存在额外的数据库同步开销进而增加了写操作的延迟时间；
- 使用进程内事件机制，独立事务同步（比如Guava的AsyncEventBus），这种方式的好处是写操作和同步操作彼此独立互不影响，缺点是无法保证二者的原子性进而可能使系统产生脏数据；
- 使用独立的消息机制（比如RabbitMQ/Kafka等），独立事务同步，可以将查询功能分离为单独的子系统，事实上这种方式已经与“跨进程跨实体 + 分离存储/分离模型”相似，因此请参考“5. 跨进程跨实体 + 分离存储/分离模型”小节。

## 5. 跨进程跨实体 + 分离存储/分离模型

这种方式在微服务中最常见，因为微服务系统首先是多进程的，每个服务都内聚性地管理自身的聚合根对象，另外，微服务的数据存储通常也是独占式的，意味着在微服务系统中数据存储一定是分离的，在这种场景下，跨微服务之间的查询通常采用“API Compositon”模式或者本文的CQRS模式。

在“跨进程跨实体 + 分离存储/分离模型”中，存在一个单独的查询服务用于CQRS的读操作，查询所需数据通常通过事件机制从不同的其他业务服务中同步而来，读操作所返回的数据通过API Gateway或者BFF向外暴露，示意图如下：



在本文的示例电商项目中，需要在查询Order的时候同时带上Product的信息，但是由于Order和Product分别属于不同的服务，为此创建[ecommerce-order-query-service](#)查询服务，该服务负责接收Order和Product服务发布的领域事件以同步其自身的读模型[OrderWithProductRepresentation](#)。

在`ecommerce-order-query-service`服务中，在接收到`OrderEvent`事件后，`OrderQueryRepresentationService`负责分别调用Order和Product的接口完成数据同步：

```
public void cqrsSync(OrderEvent event) {  
    String orderUrl = "http://localhost:8080/orders/{id}";  
    String productUrl = "http://localhost:8082/products/{id}";  
  
    OrderRepresentation orderRepresentation = restTemplate.getForObject(orderUrl,  
    OrderRepresentation.class, event.getOrderId());  
    List<Product> products = orderRepresentation.getItems().stream().map(orderItem -> {  
        ProductRepresentation productRepresentation = restTemplate.getForObject(productUrl,  
        ProductRepresentation.class,  
        orderItem.getProductId());  
        return new Product(productRepresentation.getId(),  
            productRepresentation.getName(),  
            productRepresentation.getDescription());  
    }).collect(Collectors.toList());  
    OrderWithProductRepresentation order = new OrderWithProductRepresentation(  
        orderRepresentation.getId(),  
        orderRepresentation.getTotalPrice(),  
        orderRepresentation.getStatus(),  
        orderRepresentation.getCreatedAt(),  
        orderRepresentation.getAddress(),  
        products  
    );  
    dao.save(order);  
    log.info("CQRS synced order {}.", orderId);  
}
```

在本例中，**ecommerce-order-query-service**查询服务使用了关系型数据库，但在实际应用中应该根据项目所需选择适当的数据存储机制。例如，对于海量数据的查询，可以选择诸如MongoDB或者Cassandra之类的NoSQL数据库；而对于需要进行全文搜索的场景，可以采用Elasticsearch等。

事实上，在接收并处理事件时，存在2中风格，一种是本例中的仅将事件作为消息通知，然后调用其他服务的API接口完成同步，另一种是直接使用事件所携带的数据进行同步，更多关于这2种风格的比较，请参考笔者的《事件驱动架构(EDA)编码实践》文章。

事件驱动架构总是意味着异步，它将给软件带来以下方面的影响：

- 读模型和写模型之间不再是强事务一致性，而是最终一致性。
- 从用户体验上讲，用户发起操作之后将不再立即返回结果数据，此时要么需要调用方（比如前端）进行轮询查询，要么需要在用户体验上做些权衡，比如使用确认页面延迟用户对查询数据的获取。

## 关于Representation对象的命名

命名总是一件令开发者头疼的事情，特别对于需要返回多种数据形式的查询接口来说。为此，笔者自己采用以下方式命名不同的**Representation**对象，以Order为例：

- **OrderRepresentation**: 仅仅包含聚合根实体自身状态详情，一种常见的形式是通过Order。.
- **toRepresentation()**方法获得。
- **OrderSummaryRepresentation**: 用于返回聚合根的列表，仅仅包含Order本身的状态。
- **OrderWithProductRepresentation**: 用于返回带有Product数据的Order详情。
- **OrderWithProductSummaryRepresentation**: 用于返回带有Product数据的Order列表。

当然，命名是一件见仁见智的事情，以上也绝非最佳方式，不过总的原则是要一致、清晰、可读。



## 什么时候该采用CQRS

事实上，不管是Martin Fowler、Udi Dahan还是Chris Richardson，都提醒到需要慎用CQRS，因为它会带来额外的复杂性；而另有人（比如Gabriel Schenker）却提到，当前很多软件逻辑复杂性能低下恰恰是因为没有选择CQRS造成的。

的确，不管在架构层面还是编码层面，采用CQRS的都会增加程序的复杂度和代码量，不过，这种复杂性可以在很大程度上被其所带来的“条理性”所抵消，“有条理的多”恰恰是为了简单。因此，当你的项目正在承受本文一开始的“一个例子”小节中所提到的“痛楚”时，不妨试一试本文提到的几种简化版的CQRS实践。

## 总结

本文本着“简单可用的CQRS”的目的讲到了不同的CQRS实现模式，其中包含如何在单体和微服务架构中进行不同的CQRS落地实践。可以看出，CQRS并不像人们想象中的那么难，通过适当的设计与选择，CQRS可以在很大程度上将程序架构变得更加的有条理，进而使软件项目在CQRS上的付出变成一件值得做的事情。

# 用DDD实现打卡系统

作者: 裳娴静

“这是我们DDD workshop的作业, 仅供练习。

## 案例1. 一家咨询服务公司的Timesheet系统

### 需求

- 1.公司的所有员工能够登陆到系统填写每周工作的时间、内容。
- 2.公司有两部分员工, 一类是办公室人员, 一类是咨询人员;
- 3.咨询人员是为某个项目工作, 在每个项目里的角色不尽相同; 每个项目的Timesheet要求也不同, 根据角色的不同有不同的定义, 比如开发人员要求填写工作的story号等。每个项目的PM可以批准项目成员填写的Timesheet内容。
- 4.办公室人员的工作是与办公室的事情相关, 有很多的工作项目, 每个办公室要求的内容不同。每个办公室经理可以批准办公室人员填写的Timesheet内容。

## 问题空间与子域

### 问题空间

一家咨询服务公司的Timesheet系统。

### 子域:

领域: 这个公司的Timesheet

子域: 这家公司的Timesheet业务及为其服务的一系列活动。

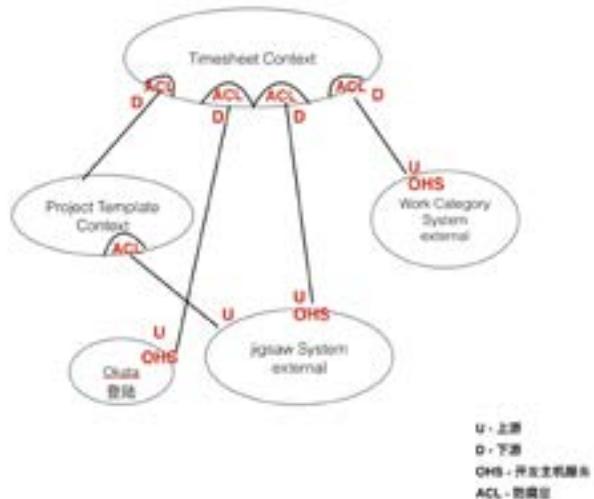
其业务活动分析如下:

PM定义项目Timesheet填写模版, 项目成员登陆系统, 选定项目, 展示模版, 填写内容。PM批准其填写的Timesheet。

办公室管理人员定义办公室Timesheet填写模版，办公室人员登陆系统，选定工作项目，展示模版，填写内容。

- Timesheet子域 - 核心子域
- 项目模版资源 - 支撑子域
- 办公室模版子域- 支撑子域
- 项目及人员信息子域- 支撑子域
- 办公室及人员信息子域- 支撑子域
- 用户登陆身份子域- 通用子域
- 解决方案与限界上下文

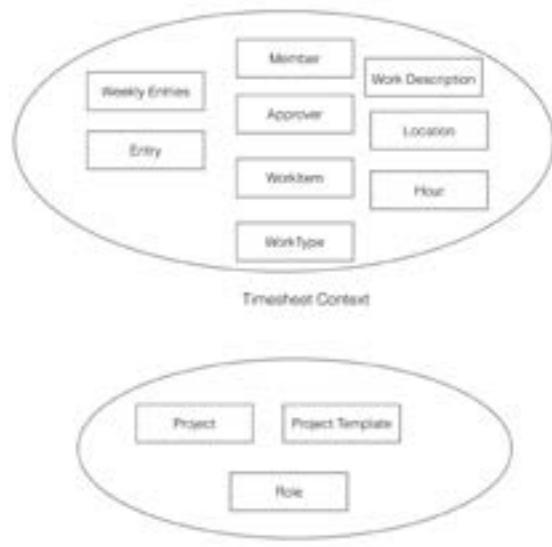
### 限界上下文映射



### 解决方案与限界上下文

限界上下文是解空间的内容，要求限界上下文中的术语是通用的无歧义的。在Project Template域中存在项目经理，而在Timesheet中则没有项目经理这个概念，只有timesheet批准人这么一个术语。

### 通用术语



# 如何划分上下文

这里面最重要的依然是两个概念，子域与限界上下文。如何定义和划分限界上下文是一个需要根据当前需求和客户一起讨论确定的。

限界上下文的不同决定了解空间的不同，限界上下文不易过早缩小，否则会带来很多小系统或者小模块，使得集成方式变得过于复杂。限界上下文过大，有可能会导致大而全的解空间，使得问题过于复杂，领域混乱。那么如果限界上下文中的术语没有歧义，是否就可以选择大而为之呢？

这一题感觉可以将Project TemplateContext 与 Timesheet Context合并，减少复杂性。



ThoughtWorks洞见  
领域驱动设计

## 扩展阅读

# DDD该如何学？

作者：姚琪琳

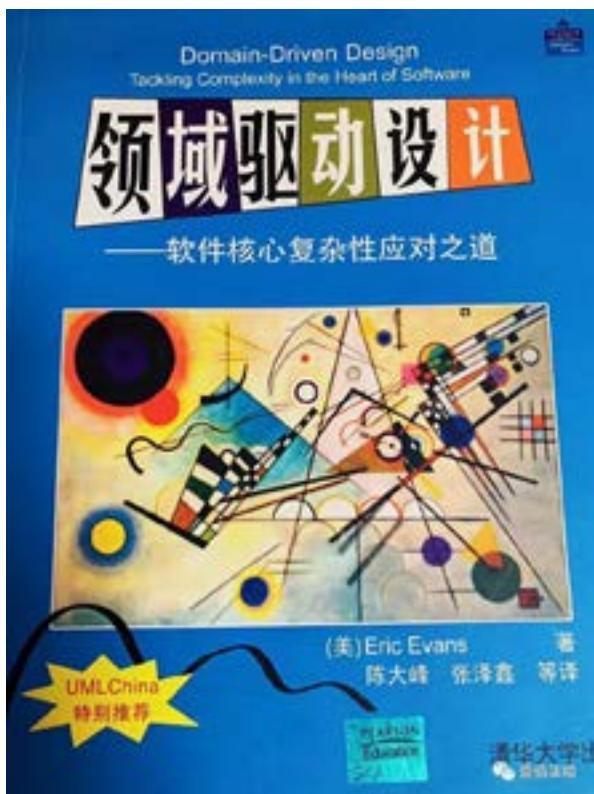
2006年，国内互联网才刚刚萌芽，大家甚至还不习惯网购，大多数在校生都在宿舍里刷魔兽世界副本。但企业软件开发却得到了蓬勃发展，各大公司和事业单位都纷纷进行信息化转型。

然而大家很快发现，企业应用业务逻辑的复杂度要远远高于技术本身，且企业IT人员很难描述清楚他们真正的业务，广大程序员也普遍缺乏挖掘真正需求的能力。整个开发过程更多的是瀑布式，开发人员一次性收集需求，可能半年后才会和业务人员再次沟通。大多数企业软件就是在这样的环境下硬着头皮上线的，其质量可想而知。

随着《领域驱动设计》中文版的首次发布，DDD (Domain-Driven Design, 领域驱动设计) 的概念正式进入中国。当时业界普大喜奔，认为它能指导程序员更精准地收集领域知识，进行更合理的设计，企业应用的银弹出现了。

当时的我正处于多层架构的启蒙阶段，挣扎于企业系统的泥潭，又刚刚被Martin Fowler的《企业应用架构》洗了一遍脑，自然也随波逐流地买了一本，捧在手里翻来翻去，但反反复复就是看不懂。当时以为在贫血模型里面加几个方法就是领域模型了，把DAL或DAO改名成Repository就是资源库了。而且身为程序员，自然愿意去关注那些能指导我们写代码的战术设计方法，对那些真正能帮助我们进行合理设计的战略设计方法，则视而不见（可能是因为看也看不懂）。

多年过去，这本书仍然作为我的镇宅之宝戳在书架显眼的位置，希望能有识货的朋友来访时能一眼瞧见，伸出大拇指羡慕嫉妒地说“这么老的书你都有”。或者偶尔拿出来拍张照片在朋友圈晒晒，以炫耀自己当初的见识。顺便翻开一页，把鼻子凑上去闻一闻来自12年前的墨香。



7年之后 Vaughn Vernon 出版了 Implement Domain-Driven Design, 简称 IDDD。一年之后由同事翻译的中文版《实现领域驱动设计》也相应出版，当时被看做是能让 DDD 落地的书（毕竟书名里有个“实现”嘛）。然而我在项目技术负责人的带领下，在众多有经验的架构师的指导下，仍然没有弄明白。之前看过的相关知识均已遗忘殆尽。限界上下文、上下文映射这些名词只是似曾相识。

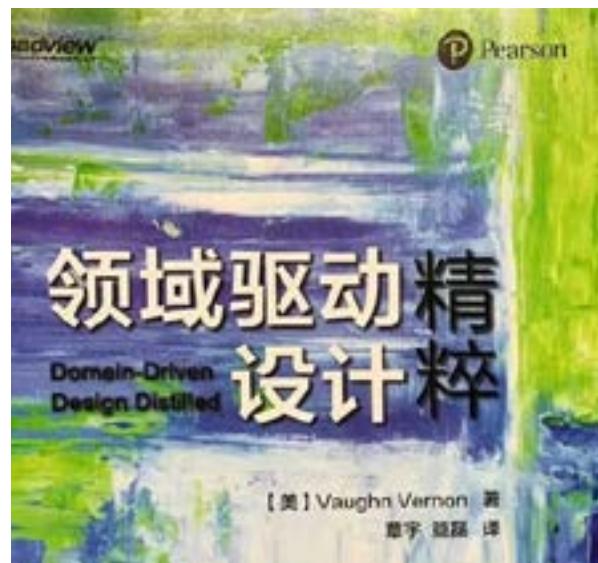
两年之后《领域驱动设计模式、原理与实践》问世，简称 PPPDDD。社区对这本书的评价非常之高，甚至认为在 IDDD 之上。只可惜这本书的翻译质量并不高，我翻了几页之后又束之高阁了。

今年年初，项目上的架构小组又开始组织学习 DDD。所使用的“教材”是英文版的 PPPDDD。在同事的激励下，我开始重整旗鼓，啃这本英文版大部头。开始精读之后，才发现这是一本很水的好书。说它水是因为

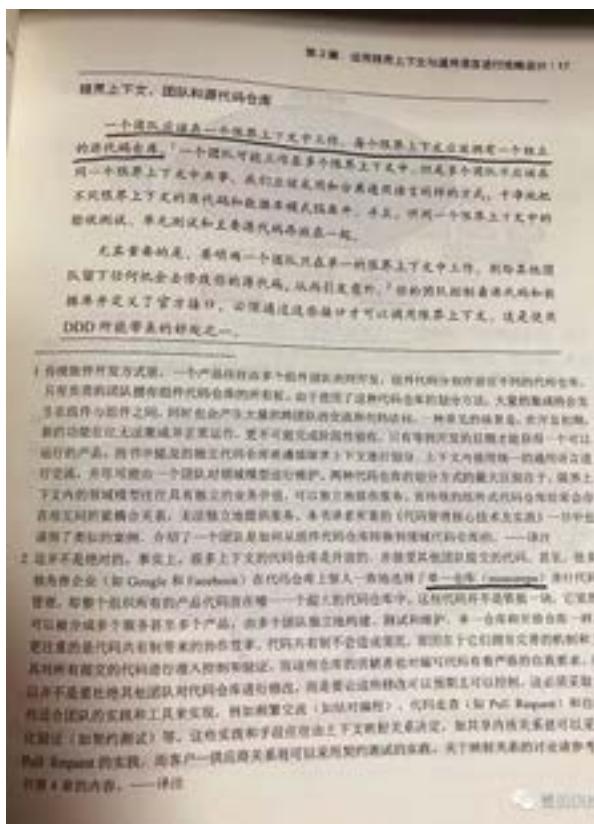
它的编排并不足够细心，甚至有不同章节的两段文字完全相同的现象，还会花 30 页的篇幅去介绍一个基于 NHibernate 的资源库实现。说它好是因为面面俱到，把所有战略模式和战术模式都介绍了个遍，还有大量代码去帮你实现各种战术模式，可以说相当落地。

在学习的过程中，我常常翻阅 IDDD 中的相关章节进行补充阅读，发现当初晦涩难懂的概念慢慢变得容易起来。应用服务和领域服务不再傻傻分不清楚，不同的上下文映射方式也能在工作中找到对应的例子。对于 DDD，感觉快要开始入门了。

与此同时，IDDD 的精华版 DDDD (Domain-Driven Design Distilled) 也出版了。作者总结了过去几年在 DDD 方面的实战经验，将 IDDD 中的诸多内容精简升华。在很多概念处都标注了 IDDD 中的相关章节，可以算是 IDDD 的一个索引。



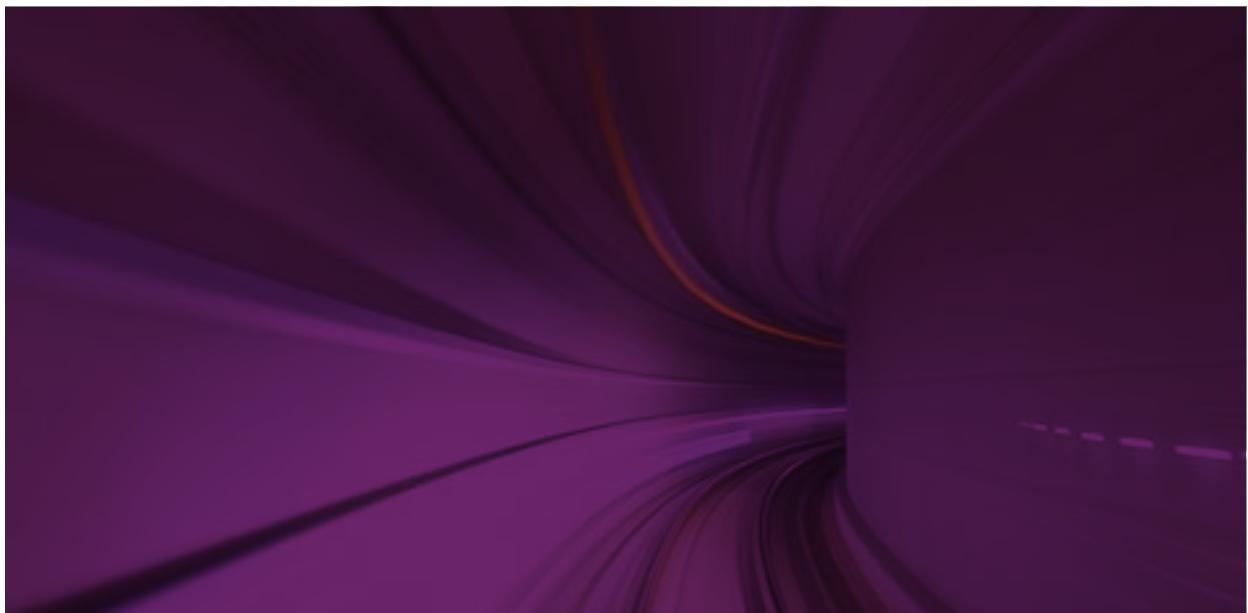
其中文版《领域驱动设计精粹》由 ThoughtWorks 同事 覃宇 和 章磊 合作翻译。这是我读过的最良心的一本书籍，因为它包含了大量译者注解，解释了很多书中没有解释清楚的概念（毕竟是精粹本）。还有些有争议的观点，译者也毫不客气地给出自己的看法。



像这样注解超过原文的情况在其他书中是很少见的。每一处注解都倾注了译者的心血和精力，这背后势必包括大量资料的查阅和研究，而且很多解释都夹带了浓浓的ThoughtWorks特色，使得这样一本薄薄的书变得丰满充实起来。

如果你读书快的话，可能两个小时就读完这样一本书。但如果把原书和注解中的推荐文章和书籍读完，恐怕要一个月。我顺着书中的指引，找到了ThoughtWorks洞见上的所有DDD文章，读完之后，世界观崩塌了，感觉自己刚要入门就要放弃了。具体原因请参考链接中的文章。

其实，不管是DDD、IDDD、PPPPP还是DDDD，讲的都是理论，充其量会附加一些作者杜撰的示例。相信我，光学习理论是没有用的，你必须将其应用于实践，在自己的真实项目里演练DDD。这时你才会发现，那些白纸黑字的概念，在读书时似乎搞清楚了，但一使用起来，反而更迷惑了。就像最基本的子域和限界上下文的关系问题，ThoughtWorks的首席咨询师肖然就和Vaughn Vernon的理解就相去甚远。到底该“信谁”？那就似乎更要通过实践来出真知了。



# 领域驱动设计 (DDD) 实现之路

作者: 滕云

2004年,当Eric Evans的那本《领域驱动设计——软件核心复杂性应对之道》(后文简称《领域驱动设计》)出版时,我还在念高中,接触到领域驱动设计 (DDD) 已经是8年后的的事情了。那时,我正打算在软件开发之路上更进一步,经同事介绍,我开始接触DDD。

我想,多数有经验的程序开发者都应该听说过DDD,并且尝试过将其应用在自己的项目中。不知你是否遇到过这样的场景:你创建了一个资源库 (Repository),但一段时间之后发现这个资源库和传统的DAO越来越像了,你开始反思自己的实现方式是正确的吗?或者,你创建了一个聚合,然后发现这个聚合是如此的庞大,它为什么引用了如此多的对象,难道又是我做错了吗?

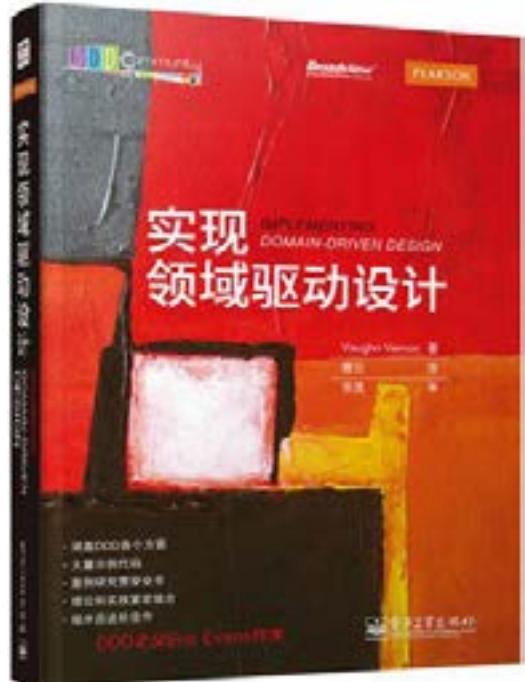
其实你并不孤单,我相信多数同仁都曾遇到过相似的问题。前不久,我一个同事给我展示了他在2007年买的那本已经被他韦编三绝过的《领域驱动设计》,他告诉我,读过好几遍后,他依然不知道如何将DDD付诸实践。Eric那本书固然是好,无可否认,但是我们程序员总希望看到一些实际的例子能够切实将DDD落地以指导我们的日常开发。

于是,在Eric的书出版将近10年之后,我们有了 Vaughn Vernon 的《实现领域驱动设计》,作为该书的译者,我有幸通读了本书,受益匪浅,得到的结论是:好的软件就应该是DDD的。

就像在微电子领域有知识产权核 (Intellectual Property) 一样,DDD将一个软件系统的业务功能集中在一个核心域里面,其中包含了实体、值对象、领域服务、资源库和聚合等概念。在此基础上,DDD

提出了一套完整的支撑这样的核心领域的基础设施。此时,DDD已经不再是“面向对象进阶”那么简单了,而是演变成了一个系统工程。

所谓领域,即是一个组织的业务开展方式,业务价值便体现在其中。长久以来,我们程序员都是很好的技术型思考者,我们总是擅长从技术的角度来解决项目问题。但是,一个软件系统是否真正可用是通过它所提供的业务价值体现出来的。因此,与其每天钻在那些永远也学不完的技术中,何不将我们的关注点向软件系统所提供的业务价值方向思考思考,这也正是DDD所试图解决的问题。



在DDD中，代码就是设计本身，你不再需要那些繁文缛节的并且永远也无法得到实时更新的设计文档。编码者与领域专家再也不需要翻译才能理解对方所表达的意思。DDD有战略设计和战术设计之分。战略设计主要从高层“俯视”我们的软件系统，帮助我们精准地划分领域以及处理各个领域之间的关系；而战术设计则从技术实现的层面教会我们如何具体地实施DDD。

## DDD之战略设计

需要指出的是，DDD绝非一套单纯的技术工具集，但是我所看到的很多程序员却的确是这么认为的，并且也是怀揣着这样的想法来使用DDD的。过于拘泥于技术上的实现将导致DDD-Lite。简单来讲，DDD-Lite将导致劣质的领域对象，因为我们忽略了DDD战略建模所带来的好处。DDD的战略设计主要包括领域/子域、通用语言、限界上下文和架构风格等概念。

## 领域和子域 (Domain/Subdomain)

既然是领域驱动设计，那么我们主要的关注点理所当然应该放在如何设计领域模型上，以及对领域模型的划分。

领域并不是多么高深的概念，比如，一个保险公司的领域中包含了保险单、理赔和再保险等概念；一个电商网站的领域包含了产品名录、订单、发票、库存和物流的概念。这里，我主要讲讲对领域的划分，即将一个大的领域划分成若干个子域。

在日常开发中，我们通常会将一个大型的软件系统拆分成若干个子系统。这种划分有可能是基于架构方面的考虑，也有可能是基于基础设施的。但是在DDD中，我们对系统的划分是基于领域的，也即是基于业务的。

于是，问题也来了：首先，哪些概念应该建模在哪些子系统里面？我们可能会发现一个领域概念建模在子系统A中是可以的，而建模在子系统B中似乎也合乎情理。第二个问题是，各个子系统之间的应该如何集成？有人可能会说，这不简单得就像客户端调用服务端那么简单吗？问题在于，两个系统之间的集成涉及到基

础设施和不同领域概念在两个系统之间的翻译，稍不注意，这些概念就会对我们精心创建好的领域模型造成污染。

如何解决？答案是：限界上下文和上下文映射图。

## 限界上下文 (Bounded Context)

在一个领域/子域中，我们会创建一个概念上的领域边界，在这个边界中，任何领域对象都只表示特定于该边界内部的确切含义。这样边界便称为限界上下文。限界上下文和领域具有一对一的关系。

举个例子，同样是一本书，在出版阶段和出售阶段所表达的概念是不同的，出版阶段我们主要关注的是出版日期，字数，出版社和印刷厂等概念，而在出售阶段我们则主要关心价格，物流和发票等概念。我们应该怎么办呢，将所有这些概念放在单个Book对象中吗？这不是DDD的做法，DDD有限界上下文将这两个不同的概念区分开来。

从物理上讲，一个限界上下文最终可以是一个DLL(.NET)文件或者JAR(Java)文件，甚至可以是一个命名空间（比如Java的package）中的所有对象。但是，技术本身并不应该用来界分限界上下文。将一个限界上下文中的所有概念，包括名词、动词和形容词全部集中在一起，我们便为该限界上下文创建了一套通用语言。通用语言是一个团队所有成员交流时所使用的语言，业务分析人员、编码人员和测试人员都应该直接通过通用语言进行交流。

对于上文中提到的各个子域之间的集成问题，其实也是限界上下文之间的集成问题。在集成时，我们主要关心的是领域模型和集成手段之间的关系。比如需要与一个REST资源集成，你需要提供基础设施（比如Spring 中的RestTemplate），但是这些设施并不是你核心领域模型的一部分，你应该怎么办呢？答案是防腐层，该层负责与外部服务提供方打交道，还负责将外部概念翻译成自己的核心领域能够理解的概念。当然，防腐层只是限界上下文之间众多集成方式的一种，另外还有共享内核、开放主机服务等，具体细节请参考

《实现领域驱动设计》原书。限界上下文之间的集成关系也可以理解为是领域概念在不同上下文之间的映射关系，因此，限界上下文之间的集成也称为上下文映射图。

## 架构风格 (Architecture)

DDD并不要求采用特定的架构风格，因为它是对架构中立的。你可以采用传统的三层式架构，也可以采用REST架构和事件驱动架构等。但是在《实现领域驱动设计》中，作者比较推崇事件驱动架构和六边形 (Hexagonal) 架构。

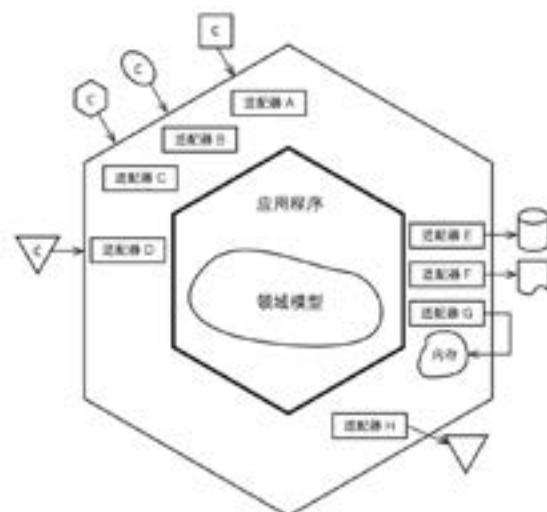
当下，面向接口编程和依赖注入原则已经在颠覆着传统的分层架构，如果再进一步，我们便得到了六边形架构，也称为端口和适配器 (Ports and Adapters)。在六边形架构中，已经不存在分层的概念，所有组件都是平等的。这主要得益于软件抽象的好处，即各个组件的之间的交互完全通过接口完成，而不是具体的实现细节。正如Robert C. Martin所说：

抽象不应该依赖于细节，细节应该依赖于抽象。

采用六边形架构的系统中存在着很多端口和适配器的组合。端口表示的是一个软件系统的输入和输出，而适配器则是对每一个端口的访问方式。比如，在一个Web应用程序中，HTTP协议可以作为一个端口，它向用户提供HTML页面并且接受用户的表单提交；而Servlet（对于Java而言）或者Spring中的Controller则是相对应于HTTP协议的适配器。再比如，要对数据进行持久化，此时的数据库系统则可看成是一个端口，而访问数据库的Driver则是相对于数据库的适配器。如果要为系统增加新的访问方式，你只需要为该访问方式添加一个相应的端口和适配器即可。

那么，我们的领域模型又如何与端口和适配器进行交互呢？

上文已经提到，软件系统的真正价值在于提供业务功能，我们会将所有的业务功能分解为若干个业务用例，每一次业务用例都表示对软件系统的一次原子操作。所以首先，软件系统中应该存在这样的组件，他们的作用即以业务用例为单位向外界暴露该系统的业务功能。在DDD中，这样的组件称为应用层 (Application Layer)。



在有了应用层之后，软件系统和外界的交互便变成了适配器和应用层之间的交互，如上图所示。

从图中可以看出，领域模型位于应用层的核心部分，外界与领域模型的交互都通过应用层完成，应用层是领域模型的直接客户。然而，应用层中不应该包含有业务逻辑，否则就造成了领域逻辑的泄漏，而应该是很薄的一层，主要起到协调的作用，它所做的只是将业务操作代理给我们的领域模型。同时，如果我们的业务操作有事务需求，那么对于事务的管理应该放在应用层上，因为事务也是以业务用例为单位的。

应用层虽然很薄，但却非常重要，因为软件系统的领域逻辑都是通过它暴露出去的，此时的应用层扮演了系统门面 (Facade) 的角色。

## DDD之战术设计

战略设计为我们提供一种高层视野来审视我们的软件系统，而战术设计则将战略设计进行具体化和细节化，它主要关注的是技术层面的实施，也是对我们程序员来得最实在的地方。

### 行为饱满的领域对象

我们希望领域对象能够准确地表达出业务意图，但是多数时候，我们所看到的却是充满getter和setter的领域对象，此时的领域对象已经不是领域对象了，而是Martin Fowler所称之为的贫血对象。

放到Java世界中，多年以来，Java Bean规范都引诱着程序员们以“自然而然又合乎情理”的方式创建着无数的贫血对象，而一些框架也规定对象必须提供getter和setter方法，比如Hibernate的早期版本。那么，贫血对象到底有什么坏处呢？来看一个例子：要修改一个客户（Customer）的邮箱地址，在使用setter方法时为：

```
public class Customer {  
    private String email;  
  
    public void setEmail(String email) {  
        this.email = email;  
    }  
}
```

虽然以上代码可以完成“修改邮箱地址”的功能，但是当你读到这段代码时，你能否推测出系统中就一定存在着一个“修改邮箱地址”的业务用例呢？

你可能会说，可以在另一个Service类里面创建一个changeCustomerEmail()方法，再在该方法中调用Customer的setEmailAddress()方法，这样业务意图不就明了了吗？问题在于，修改邮箱地址这样的职责本来就应该放在Customer上，而不应该由Service和Customer共同完成。遵循诸如信息封装这样的基本面向对象原则是在实施DDD时最基本的素养。

要创建行为饱满的领域对象并不难，我们需要转变一下思维，将领域对象当做是服务的提供方，而不是数据容器，多思考一个领域对象能够提供哪些行为，而不是数据。

近几年又重新流行起来的函数式编程也能够帮助我们编写更加具有业务表达力的业务代码，比如C#和Java 8都

提供了Lambda功能，同时还包括多数动态语言（比如Ruby和Groovy等）。再进一步，我们完全可以通过领域特定语言（DSL）的方式实现领域模型。

笔者曾经设想过这么一个软件系统：它的核心功能完全由一套DSL暴露给外界，所有业务操作都通过这套DSL进行，这个领域的业务规则可以通过一套规则引擎进行配置，于是这套DSL可以像上文提到的知识产权核一样拿到市面上进行销售。此时，我们的核心域被严严实实地封装在这套DSL之内，不容许外界的任何污染。

### 实体vs值对象 (Entity vs Value Object)

在一个软件系统中，实体表示那些具有生命周期并且会在其生命周期中发生改变的东西；而值对象则表示起描述性作用的并且可以相互替换的概念。同一个概念，在一个软件系统中被建模成了实体，但是在另一个系统中则有可能是值对象。例如货币，在通常交易中，我们都将它建模成了一个值对象，因为我们花了20元买了一本书，我们只是关心货币的数量而已，而不是关心具体使用了哪一张20元的钞票，也即两张20元的钞票是可以互换的。但是，如果现在中国人民银行需要建立一个系统来管理所有发行的货币，并且希望对每一张货币进行跟踪，那么此时的货币便变成了一个实体，并且具有唯一标识（Identity）。在这个系统中，即便两张钞票都是20元，他们依然表示两个不同的实体。

具体到实现层面，值对象是没有唯一标识的，他的equals()方法（比如在Java语言中）可以用它所包含的描述性属性字段来实现。但是，对于实体而言，equals()方法便只能通过唯一标识来实现了，因为即便两个实体所拥有的状态是一样的，他们依然是不同的实体，就像两个人的名字都叫张三，但是他们却是两个不同的人的个体。

我们发现，多数领域概念都可以建模成值对象，而非实体。值对象就像软件系统中的过客一样，具有“创建后不管”的特征，因此，我们不需要像关心实体那样去关心诸如生命周期和持久化等问题。

### 聚合 (Aggregate)

聚合可能是DDD中最难理解的概念，之所以称之为聚合，是因为聚合中所包含的对象之间具有密不可分的联系，他们是内聚在一起的。比如一辆汽车（Car）包含了引擎（Engine）、车轮（Wheel）和油箱（Tank）等组件，缺一不可。一个聚合中可以包含多个实体和值对象，因此聚合也被称为根实体。聚合是持久化的基本单位，它和资源库（请参考下文）具有一一对应的关系。

既然聚合可以容纳其他领域对象，那么聚合应该设计得多大呢？这也是设计聚合的难点之一。比如在一个博客（Blog）系统中，一个用户（User）可以创建多个Blog，而一个Blog又可以包含多篇博文（Post）。在建模时，我们通常的做法是在User对象中包含一个Blog的集合，然后在每个Blog中又包含了一个Post的集合。你真的需要这么做吗？如果你需要修改User的基本信息，在加载User时，所有的Blog和Post也需要加载，这将造成很大的性能损耗。诚然，我们可以通过延迟加载的方式解决问题，但是延迟加载只是技术上的实现方式而已。导致上述问题的深层原因其实就在我们的设计上，我们发现，User更多的是和认证授权相关的概念，而与Blog关系并不大，因此完全没有必要在User中维护Blog的集合。在将User和Blog分离之后，Blog也和User一样成为了一个聚合，它拥有自己的资源库。问题又来了：既然User和Blog分离了，那么如果需要在Blog中引用User又该怎么办呢？在一个聚合中直接引用另外一个聚合并不是DDD所鼓励的，但是我们可以通过ID的方式引用另外的聚合，比如在Blog中可以维护一个userId的实例变量。

User作为Blog的创建者，可以成为Blog的工厂。放到DDD中，创建Blog的功能也只能由User完成。

综上，对于“创建Blog”的用例，我们可以通过以下方法完成：

```
public class BlogApplicationService {  
  
    @Transactional  
    public void createBlog(String blogName, String userId) {  
        User user = userRepository.userById(userId);  
        Blog blog = user.createBlog(blogName);  
        blogRepository.save(blog);  
    }  
}
```

在上例中，业务用例通过BlogApplicationService应用服务完成，在用例方法createBlog()中，首先通过User的资源库得到一个User，然后调用User中的工厂方法createBlog()方法创建一个Blog，最后通过BlogRepository对Blog进行持久化。整个过程构成了一次事务，因此createBlog()方法标记有@Transactional作为事务边界。

使用聚合的首要原则为在一次事务中，最多只能更改一个聚合的状态。如果一次业务操作涉及到了对多个聚合状态的更改，那么应该采用发布领域事件（参考下文）的方式通知相应的聚合。此时的数据一致性便从事务一致性变成了最终一致性（Eventual Consistency）。

### 领域服务 (Domain Service)

你是否遇到过这样的问题：想建模一个领域概念，把它放在实体上不合适，把它放在值对象上也不合适，然后你冥思苦想着自己的建模方式是不是出了问题。恭喜你，祝贺你，你的建模手法完全没有问题，只是你还没有接触到领域服务（Domain Service）这个概念，因为领域服务本来就是来处理这种场景的。比如，要对密码进行加密，我们便可以创建一个PasswordEncryptService来专门负责此事。

值得一提的是，领域服务和上文中提到的应用服务是不同的，领域服务是领域模型的一部分，而应用服务不是。应用服务是领域服务的客户，它将领域模型变成对外界可用的软件系统。领域服务不能滥用，因为如果我们将太多的领域逻辑放在领域服务上，实体和值对象上将变成贫血对象。

## 资源库 (Repository)

资源库用于保存和获取聚合对象，在这一点上，资源库与DAO多少有些相似之处。但是，资源库和DAO是存在显著区别的。DAO只是对数据库的一层很薄的封装，而资源库则更加具有领域特征。另外，所有的实体都可以有相应的DAO，但并不是所有的实体都有资源库，只有聚合才有相应的资源库。

资源库分为两种，一种是基于集合的，一种是基于持久化的。顾名思义，基于集合的资源库具有编程语言中集合的特征。举个例子，Java中的List，我们从一个List中取出一个元素，在对该元素进行修改之后，我们并不用显式地将该元素重新保存到List里面。因此，面向集合的资源库并不存在save()方法。比如，对于上文中的User，其资源库可以设计为：

```
public interface CollectionOrientedUserRepository {  
    public void add(User user);  
    public User userById(String userId);  
    public List allUsers();    public void remove(User user);  
}
```

对于面向持久化的资源库来说，在对聚合进行修改之后，我们需要显式地调用save()方法将其更新到资源库中。依然是User，此时的资源库如下：

```
public interface PersistenceOrientedUserRepository {  
    public void save(User user);  
    public User userById(String userId);  
    public List<User> allUsers();  
    public void remove(User user);  
}
```

在以上两种方式所实现的资源库中，虽然只是将add()方法改成了save()方法，但是在使用的时候却是不一样的。在使用面向集合资源库时，add()方法只是用来将新的聚合加入资源库；而在面向持久化的资源库中，save()方法不仅用于添加新的聚合，还用于显式地更新既有聚合。

### 领域事件 (Domain Event)

在Eric的《领域驱动设计》中并没有提到领域事件，领域事件是最近几年才加入DDD生态系统的。

在传统的软件系统中，对数据一致性的处理都是通过事务完成的，其中包括本地事务和全局事务。但是，DDD的一个重要原则便是一次事务只能更新一个聚合实例。然而，的确存在需要修改多个聚合的业务用例，那么此时我们应该怎么办呢？

另外，在最近流行起来的微服务 (Micro Service) 的架构中，整个系统被分成了很多个轻量的程序模块，他们之间的数据一致性不容易通过事务一致性完成，此时我们又该怎么办呢？

在DDD中，领域事件便可以用于处理上述问题，此时最终一致性取代了事务一致性，通过领域事件的方式达到各个组件之间的数据一致性。

领域事件的命名遵循英语中的“名词+动词过去分词”格式，即表示的是先前发生过的一件事情。比如，购买者提交商品订单之后发布OrderSubmitted事件，用户更改邮箱地址之后发布EmailAddressChanged事件。

需要注意的是，既然是领域事件，他们便应该从领域模型中发布。领域事件的最终接收者可以是本限界上下文中的组件，也可以是另一个限界上下文。

领域事件的额外好处在于它可以记录发生在软件系统中所有的重要修改，这样可以很好地支持程序调试和商业智能化。另外，在CQRS架构的软件系统中，领域事件还用于写模型和读模型之间的数据同步。再进一步发展，事件驱动架构可以演变成事件源 (Event Sourcing)，即对聚合的获取并不是通过加载数据库中的瞬时状态，而是通过重放发生在聚合生命周期中的所有领域事件完成。

## 总结

DDD存在战略设计和战术设计之分，过度地强调DDD的技术性将使我们错过由战略设计带来的好处。因此，在实现DDD时，我们应该将战略设计也放在一个重要的位置加以对待。战略设计帮助我们从一个宏观的角度观察和审视软件系统，其中的限界上下文和上下文映射图帮助我们正确地界分各个子域（系统）。DDD的战术设计则更加侧重于技术实现，它向我们提供了一整套技术工具集，包括实体、值对象、领域服务和资源库等。虽然DDD的概念已经提出近10年了，但是在如何实现DDD上，我们依然有很长的路要走。

# 从“四色建模法”到“限界纸笔建模法”

作者: 伍斌



对于领域专家、程序员和测试工程师来说，领域建模所构建的概念模型是撑起软件开发系统的骨架。领域建模的方法有很多种，ThoughtWorks的同事们经常使用经过徐昊改编的“四色建模法”<sup>[1]</sup>来进行建模。而本文所描述的“限界纸笔建模法”，在“四色建模法”的“时标对象”的基础上确定“限界上下文”<sup>[2]</sup>与“聚集”<sup>[3]</sup>的概念，再使用“纸和笔来管理”的方法，力图在建模过程中实现“分而治之”，增强数据的完整性，并避免过度设计。

为了便于说明，下文将以“小画笔”绘画课外班的业务为例，首先描述同事亢江妹用四色建模法对其所建的模型，然后再描述如何在其“时标对象”的基础上，运用限界纸笔建模法进行建模。

## “小画笔”绘画课外班

“小画笔”是一家面向2-10岁小朋友的绘画培训机构。乐乐老师是这个培训机构的主管。她毕业于北京师范学院，有一些认识的同学和老师愿意在业余时间授课来获得一定收入。

“小画笔”目前有三个班：

- 美术预科：适合2-3岁孩子，从看、摸、闻、听、尝培养艺术感；每期课程8次，每周一次，周末上；每班最少6个孩子开课，最多10个孩子。
- 书法：适合3-6岁孩子，学习字体结构、笔画线条；每期课程8次，每周一次，周末或晚上上；每班最少6个孩子开课，最多12个孩子。
- 儿童绘本：适合年龄5-10岁，用文字和图画表达；每期课程8次，每周一次，周末或晚上上；每班最少6个孩子开课，最多12个孩子。

“小画笔”有三个教室：达芬奇，毕加索和梵高。

每两个月开始新期课程，8次课结束后重新开始。

乐乐老师的主要工作是：

- 根据现有老师能够讲授的课程及时间，做好课程表安排，提前两个月把下期课程介绍做好，印成宣传彩页；
- 在附近社区里给家长宣传，招募生源；
- 接受家长报名（电话或面对面），记录和追踪报名情况；
- 收取学费，学费支付采用现金付款，或银联卡刷卡支付；
- 争取在每一期课程开始前都有足够学员报名，可以准时开班；
- 保证课程质量和安全，保证营收，维持培训机构正常运转。

乐乐老师需要处理的三个核心问题有：

- 下一期的课程如何安排？ - 排课
- 如何招生和管理报名？ - 管理报名
- 如何保证课程质量？ - 管理课堂

挑战：

请为乐乐老师设计一个软件工具的概念模型，能让乐乐老师方便地管理排课、报名和学员考勤，维护“小画笔”的正常运营。

在日常运营中，乐乐老师经常碰到下列问题：

- 课程开始后发现有孩子没来上课，也没有请假，需要查找家长联系电话来询问。
- 有时孩子请假会造成缺课，家长会要求在后期课程中补上。乐乐老师需要记录孩子的上课次数，在可以补上课程的时候通知孩子来参加。
- 需要查看还有哪个家长没有缴费，通知他们缴费。
- 需要查看某个孩子的缴费记录，以确认家长是否缴费。
- 经常有家长要求先试一次课，再确定报名交钱。但每班只能在报名不满额的情况下才能试听，且最多只能允许3个孩子试听。所以乐乐需要追踪每班报名多少孩子、试学多少孩子、还有没有试学名额。
- 需要经常查看每班是否报满，可否开课；如果不满，需要再去小区做宣传。
- 需要提前跟上课老师确认，确保他们能按时授课。
- 有些课程可能会有一位主任老师和另一位见习老师来共同授课，所以这些课程每门课需要管理多位老师的信息。
- 需要跟踪教学质量，确保老师按时按质授课，及正常教学安全和秩序；

请检查你所设计的概念模型能否帮乐乐老师解决上述问题。

## 用“四色建模法”进行建模

### 第一步：寻找要追溯的事件

1. 谁，在什么时候，为谁，报名了什么课程
2. 谁，在什么时候，为谁，支付了多少学费
3. 谁，在什么时候，上了什么课程

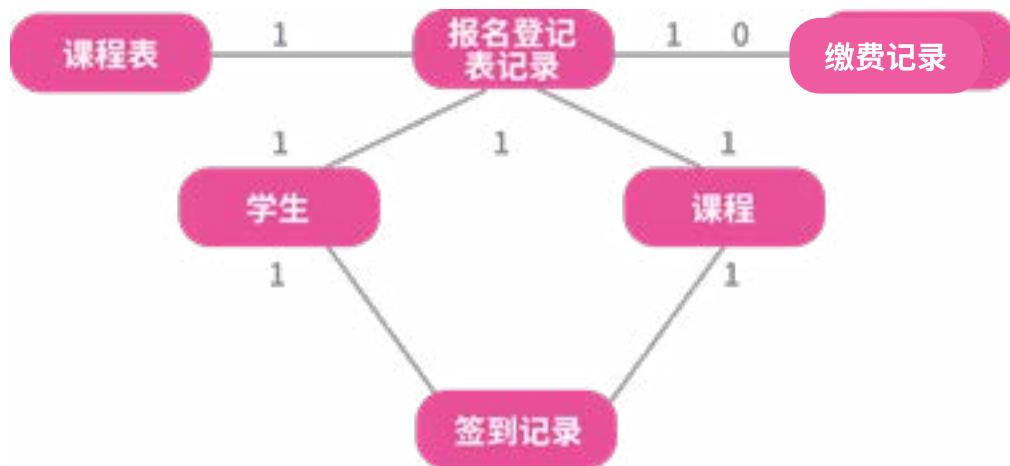
## 第二步：识别“时标对象” [4]

按时间发展的先后顺序，用红色所表示的起到“追溯单据”作用的“时标”概念，如下图所示：



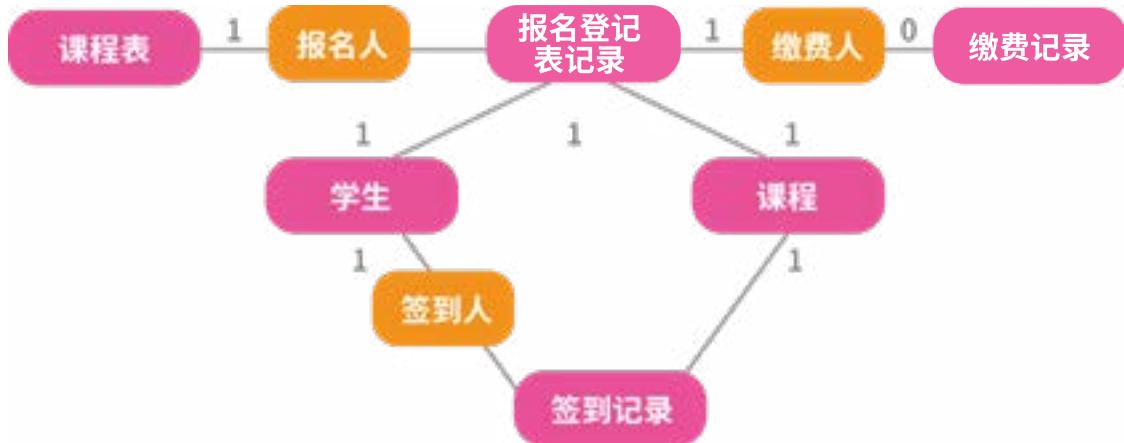
## 第三步：寻找时标对象周围的“人、地、物” [5]

在“时标”对象周围的用绿色所表示的“人、地、物”概念，如下图所示：



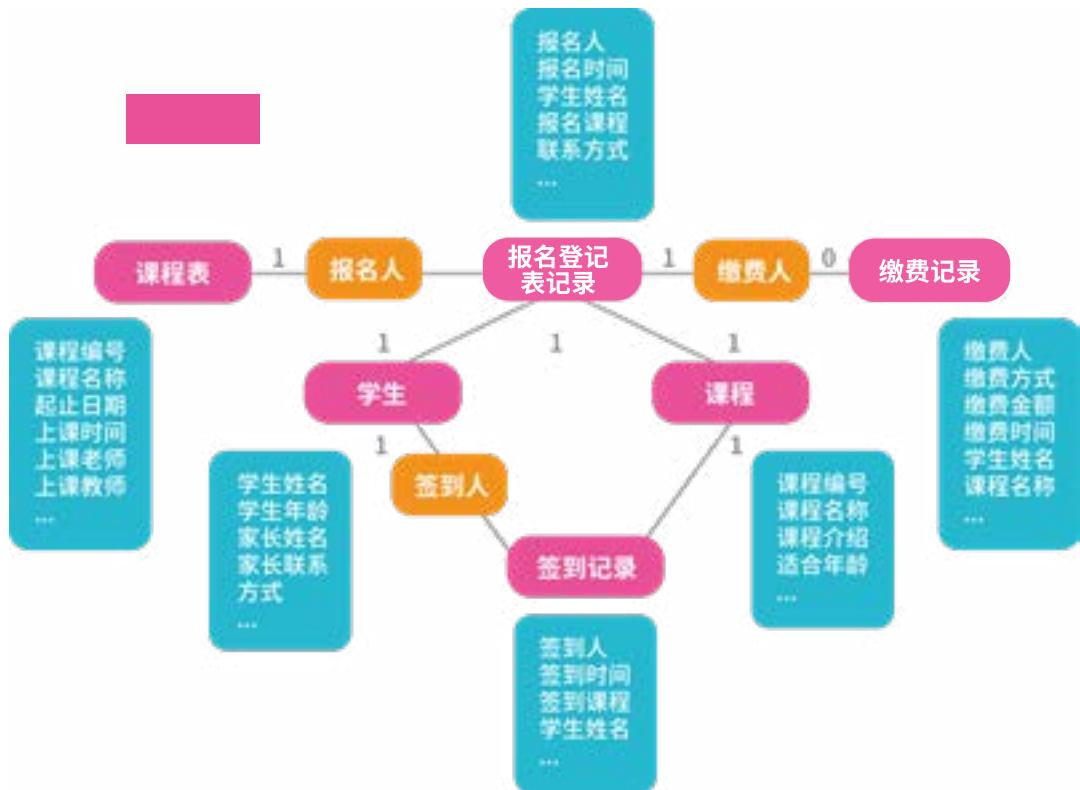
#### 第四步：抽象“角色” [6]

在上图中插入用黄色所表示的“角色”概念，如下图所示：



#### 第五步：补充“描述”信息[7]

在上图中插入用蓝色所表示的“描述”概念，如下图所示：



# 用“限界纸笔建模法”进行建模

上面用四色建模法对“小画笔”绘画课外班的软件系统所做的建模过程，最大的亮点是按时间发展的先后顺序，识别出起到“追溯单据”作用的“时标”概念。这种识别方法直达业务核心数据，简便有效。

此外，限界纸笔建模法可以继续进行以下三项建模工作：

1. 划分限界上下文，避免模型发展成“大泥球架构”。
2. 强调“聚集根”[8]的概念，更好地保证数据的完整性。
3. 寻找“恰好够用”的概念，避免过度设计，降低所建模型的复杂性。

一个“聚集根”好比一个经理，这个“聚集”所汇聚的所有彼此相关的概念好比这个经理手下管理的所有员工。当另一个“聚集”的经理，要访问该“聚集”中的某个员工时，必须要通过该“聚集”的“聚集根”这个经理的同意和安排才行，不能直接去找。这样设计能够将管理数据的访问职责缩小到一些“经理”那里，既方便定位职责，也有助于增强数据的完整性。

下面尝试用限界纸笔建模法来对“小画笔”绘画课外班的软件系统建模。

## 第一步：根据“追溯单据”的价值识别核心领域[9]

首先以“小画笔”绘画课外班的业务“追溯单据”为线索，列出这些追溯单据为乐乐老师所提供价值，并合并其中一些价值相同的单据。比如对于一个只有三间教室的小课外培训机构，只有交费成功的“报名”才视作有效报名。所以“报名登记表”和“交费纪录”可以合并。然后确定这些价值所对应的核心领域。如下表所示：

追溯单据	为乐乐老师所提供的价值	核心领域
课程表	管理课程描述、适合孩子、上课时间、老师、学费、教室等信息	排课 (Scheduling)
报名登记表、交费记录	管理交费成功的孩子所报课程、孩子、家长联系方式、交费日期及金额等信息	报名 (Enrolment)
签到记录	管理孩子所上课程、授课老师、出勤日期、上课孩子等信息	签到 (Attendance)

## 第二步：确定核心领域之间的依赖关系

排课、报名与签到三个核心领域的依赖关系如下所示：

排课 <---- 报名 <---- 签到

排课 <---- 报名 <---- 签到

## 第三步：用纸和笔画表格并写实例

先选择一个核心领域，然后开始在其所对应的“限界上下文”中，开始对其建模。假设时光回退100年，那时没有电脑，只有纸和笔。用纸和笔画表格并写实例的方法，来管理该核心领域的“恰好够用”的数据，来达成乐乐老师在此核心领域所期望的价值。

首先选择“排课”。下图是用纸和笔画出的表格，并有一条实例数据：

Scheduling 排课

课程名称	课程经理	适合年龄	上课时间	起始日期	次数	老师	老师手机	学费/教室
基础设计 入手	从零摸爬滚打 听读等感觉	2~3岁	周三晚 6:47:30	2015 2.11	12	张 哥	18921 22222 222	3000 起步

## 第四步：确定“聚集根”

给表中所有的列找一个“经理”来作为“聚集根”。这里选择“课程”作为“聚集根”，它是一种具有唯一标识（即有唯一的课程编号）的Entity[10]概念，然后把表中所有的列名都抄写到表下“聚集根”的右侧，并用括号括起来，如下图所示：

## Scheduling 排课

课程名称	课程介绍	适合年龄	上课时间	起始日期	次数	老师	老师手机	学费	备注
美术设计 从看、摸、闻、 听、尝等感觉 入手	认识颜色、 形状、大小、 冷热等概念	2~3岁	周三晚 6:27:30	2015 2.11	12	张 墨	18921 22222 222	3000	送5节课

Entity  
课程 Course (名称, 介绍, 年龄, 上课时间, 起始日期, 次数, 老师, 老师手机, 学费, 备注)

### 第五步：以“人以群分”的原则抽取新的“聚集”

观察所抄写的“聚集根”右侧所有聚集在一起的各个概念，提取出总是“一起玩儿”的多个概念，形成一个新的聚集，并确定这个新聚集的名字。比如，“上课时间”、“起始日期”、“次数”这三个概念总是一起出现，它们决定了这门课程的学时，所以就提取“学时”这个新聚集，并把这三个概念从“课程”右侧划掉。“学时”这个概念不需要有唯一标识，所以是Value Object[11]概念。它和“课程”是一对一的关系。用相同的方法可以提取“老师”这个新聚集，它是Entity概念，“课程”与它是一对多的关系。如下图所示：

## Scheduling 排课

课程名称	课程介绍	适合年龄	上课时间	起始日期	次数	老师	老师手机	学费	备注
美术设计 从看、摸、闻、 听、尝等感觉 入手	认识颜色、 形状、大小、 冷热等概念	2~3岁	周三晚 6:27:30	2015 2.11	12	张 墨	18921 22222 222	3000	送5节课

Entity  
课程 Course (名称, 介绍, 年龄, 上课时间, 起始日期, 次数, 老师, 老师手机, 学费, 备注)

学时 DateRange (时间, 起始, 结束)  
Value Object

\*  
老师 Teacher (名字, 手机)  
Entity

对于那些只“自己单独玩儿”的概念，如“学费”、“教室”等，就先暂时放到“课程”这个聚集根下面，等随着将来业务演进出现了新的“能一起玩儿的伙伴”后，再提取新的聚集不迟。

至此，使用限界纸笔建模法对“排课”这个核心领域所进行的建模告一段落。下图是使用这种方法对“报名”这个核心领域所建的模型，“签到”核心领域的建模略去不讨论。



这里有一个问题，“排课”这个限界上下文中有“课程”这个聚集根，而“报名”这个限界上下文中也有“课程”这个聚集根，这两者是同一个概念吗？

从DDD的观点来看，这两者是不同的概念，而且在数据库中也应该是两张不同的表。前者是“排课.课程”，后者是“报名.课程”。两者的属性也各不相同，前者是：排课.课程（名称、介绍、适合年龄、学费、教室），后者是：报名.课程（名称）。当然，这两个概念的“课程名称”和课程编号都是一致的。当“报名”限界上下文需要访问“排课”限界上下文中的“课程”概念来获取“课程介绍”等信息时，需要通过两者之间的“翻译器”来根据上述一致的课程编号来进行翻译和转换。

## 限界纸笔建模法的3点优势

1. 划分核心领域有助于“分而治之”：一旦确定了核心领域，限界上下文也就确定了，不同的限界上下文之间通过“翻译器”来彼此沟通并屏蔽干扰，这样就避免了“大泥球”的设计，并有助于演进到微服务架构。

- 2.“聚集根”有助于数据完整性: 每个限界上下文都有一个“聚集根”的概念, 外界对其下属概念的访问都必须通过它来进行, 这样既方便定位职责, 也有助于增强数据的完整性。
- 3.用“纸和笔”画恰好够用的概念有助于避免过度设计: 每个限界上下文中要管理的概念, 都是通过“倒退到没有电脑而用纸和笔的时代如何管理”来引导出来的, 用纸和笔来记录, 能促使人避免写过多的信息, 而只写限界上下文中恰好够用的概念。

## 总结

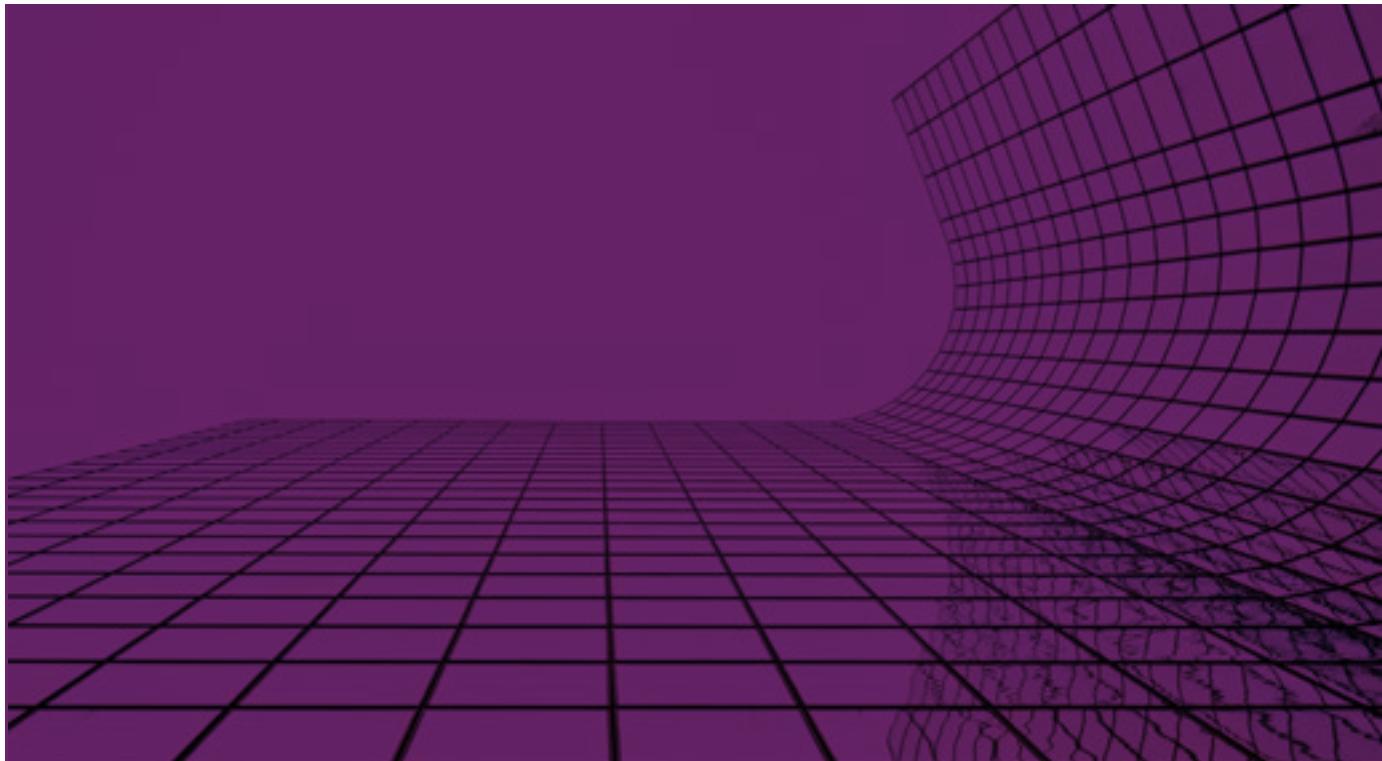
- 四色建模法最大的亮点是按时间发展的先后顺序, 识别起“追溯单据”作用的“时标”概念, 从而能把握业务核心数据, 简便有效。
- 限界纸笔建模法, 使用了DDD中的”限界上下文”与“聚集”的概念以及“纸和笔来管理”方法, 来实现“分而治之, 增强数据的完整性, 避免过度设计。

注:

- [1]四色建模法: 经过徐昊改编的“四色建模法”参见: <http://www.infoq.com/cn/articles/xh-four-color-modeling>。
- [2] 限界上下文: Bounded Context, DDD概念, 与“核心领域”(Core Domain)一一对应, 它被程序员所关注, 包含其所对应的核心领域的概念模型。
- [3] 聚集: Aggregates, DDD概念, 指某一簇彼此相关联的概念集合。
- [4] 时标对象: 指四色建模法中的”Moment, Interval”概念, 用红色表示。近似于DDD中的Repositories概念。
- [5] “人、地、物”: 指四色建模法中的“Party, Place, Thing”概念, 用绿色表示。近似于DDD中的Entities概念。
- [6] 角色: 指四色建模法中的“Role”概念, 用黄色表示。近似于DDD中的Services概念。
- [7] 描述信息: 指四色建模法中的“Description”概念, 用蓝色表示。近似于DDD中的Value Objects概念。
- [8] 聚集根: Aggregate Root, DDD概念, 指对其所属的聚集内的相关从属概念进行统一访问控制的那个概念。
- [9] 核心领域: Core Domain, DDD概念, 与“限界上下文”(Bounded Context)一一对应, 指某个业务专家所关注的、具有差异化竞争优势的、相对独立的业务领域。
- [10] Entity: DDD概念, 表示具有唯一标识的概念。
- [11] Value Object: DDD概念, 表示不必有唯一标识的概念。

# 可视化架构设计—C4介绍

作者: 全键



好多年前，同事徐昊说过的一句话给了我很大启发，他说“纸上的不是架构，每个人脑子里的才是”。这句话告诉我们，即便是天天工作在一个团队里的人，对架构的认识也可能是不一样的。每个人嘴上说的是类似的话，但心里想象的画面仍然是不一样的。在多年的工作中，我越来越认可这句话所揭示出的道理。软件开发是一个团队协作的工作，混乱的理解会造成架构的无意义腐化、技术债的无意识积累、维护成本的无价值上升。

最近听到一句话，“那些精妙的方案之所以落不了地，是因为没有在设计上兼容人类的愚蠢”。话糙理不糙，虽然最终人们选择的方案的思想都是在十年前甚至几十年前就已经存在的，然而在技术升级到足以“兼容”人类的愚蠢之前，这些思想只能在学术的故纸堆里睡大觉。当然话糙确实也会有一个问题，将一个思想性问题转化成了情绪性问题。人们容易把一些糟心的事情归因到人类的愚蠢，在宣泄完不满情绪后就停止思考了。作为知识工作者，我们的思维不能停步，我们需要思考到底人类有哪些愚蠢，分别用什么方法去避免或者“兼容”。

可以肯定彼此明明对自己开发的软件有不一样的认识却天天在一起讨论问题并试图把软件做好是一件愚蠢的事情，为了兼容这种愚蠢我们需要采用可视化的方法。

为什么需要可视化呢，主要还是语言不靠谱。人类语言真的是太随意了，只要你想，你可以说你见过一个方形的圆，并为此与别人辩论。但是无论如何你也画不出来一个方形的圆，这就是我们需要可视化的原因。

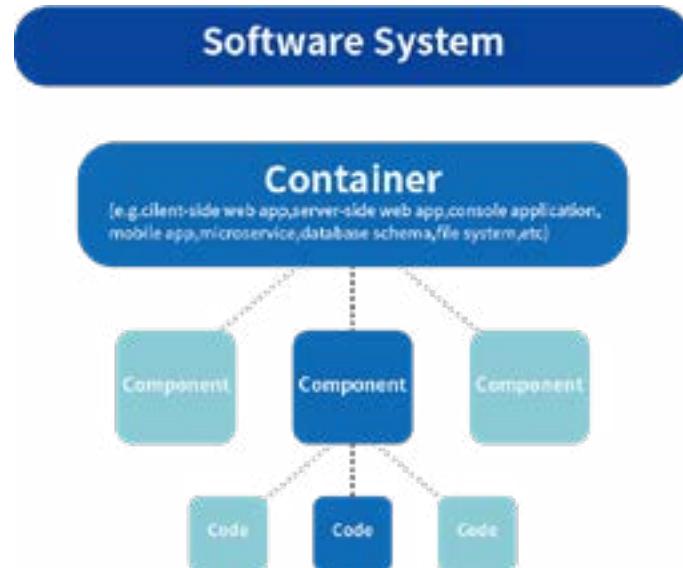
今天我们介绍一个工具，叫做C4 model，这是我近几年见到的一个比较难得跟我的认知有大量共鸣的工具。

该工具的作者在多年的咨询中经常发现，很多个人画出来的架构图都是不一样的，但也不是说谁画错了，而是每个人的抽象层次不一样。抽象层次这种东西，说起来好像存在，但真要说清楚还挺难，于是作者类比地图，提出了缩放的概念。（两年前我在教学生的时候提过同样的概念）如下图：

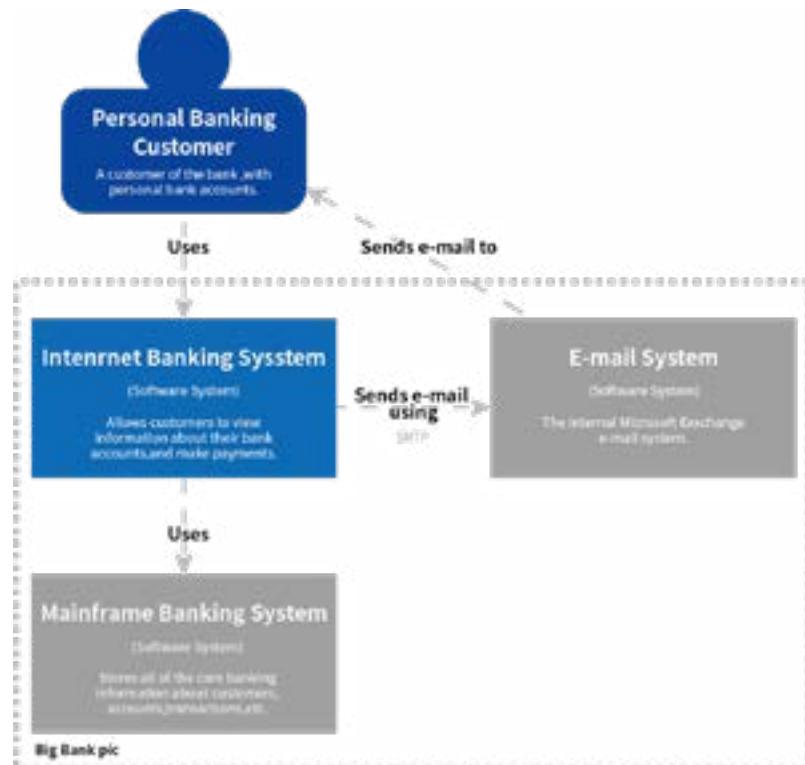


上面的四张地图就是想说明，当我们看待真实世界的“架构图”时候，也是要不停的缩放，在每一个层次刻意忽略一些细节才能表达好当前抽象层次的信息。所以他类比着把架构也提出了四个抽象层次：

从上到下依次是系统System、容器Container、组件Component和代码Code。(咦，那为什么叫C4呢，因为系统的图叫System Context，系统上下文图。为了凑四个C也是够拼的。) 基于这四个层次的抽象，C4模型由4张核心图和3张附属图组成，分别用于描述不同的场景，下面我们一一介绍一下。



## 四张核心图·系统上下文图



如上图所示，这个图表达的是你所开发的系统和它的用户以及它所依赖的系统之间的关系。从这个图上我们已经看出来C4图形的几个关键图形：

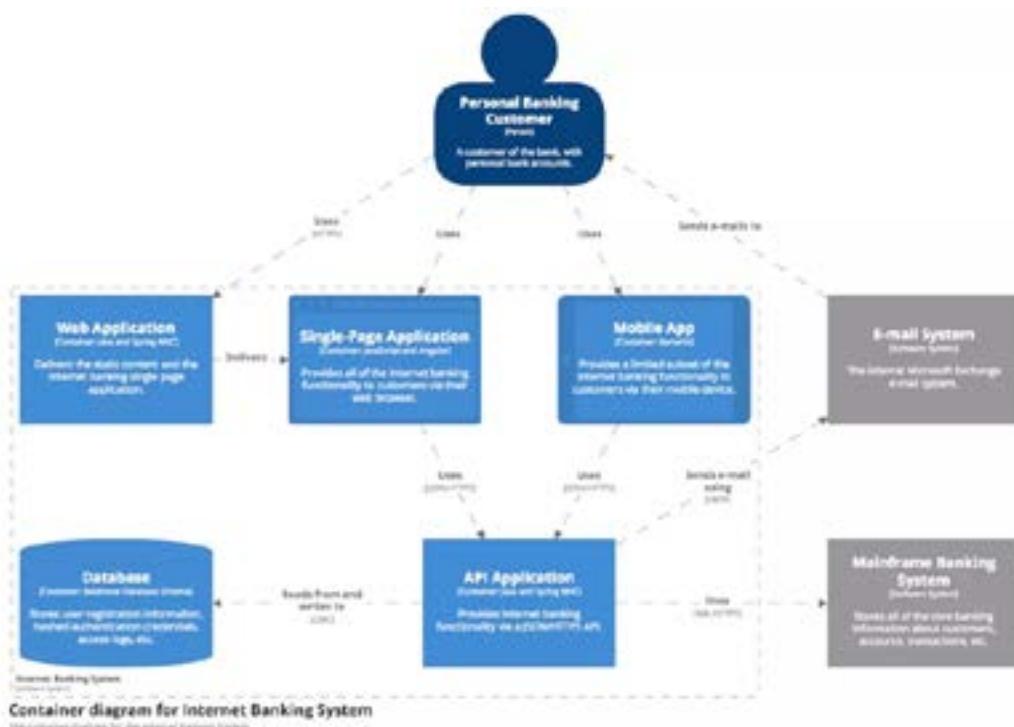


C4说穿了就是几个要素：关系——带箭头的线、元素——方块和角色、关系描述——线上的文字、元素的描述——方块和角色里的文字、元素的标记——方块和角色的颜色、虚线框（在C4里面虚线框的表达力被极大的限制了，我觉得可以给虚线框更大的扩展空间）。

通过在不同的抽象层次上，重新定义方块和虚线框的含义来将我们的表达限制在一个抽象层次上，从而避免在表达的时候产生抽象层次混乱的问题。

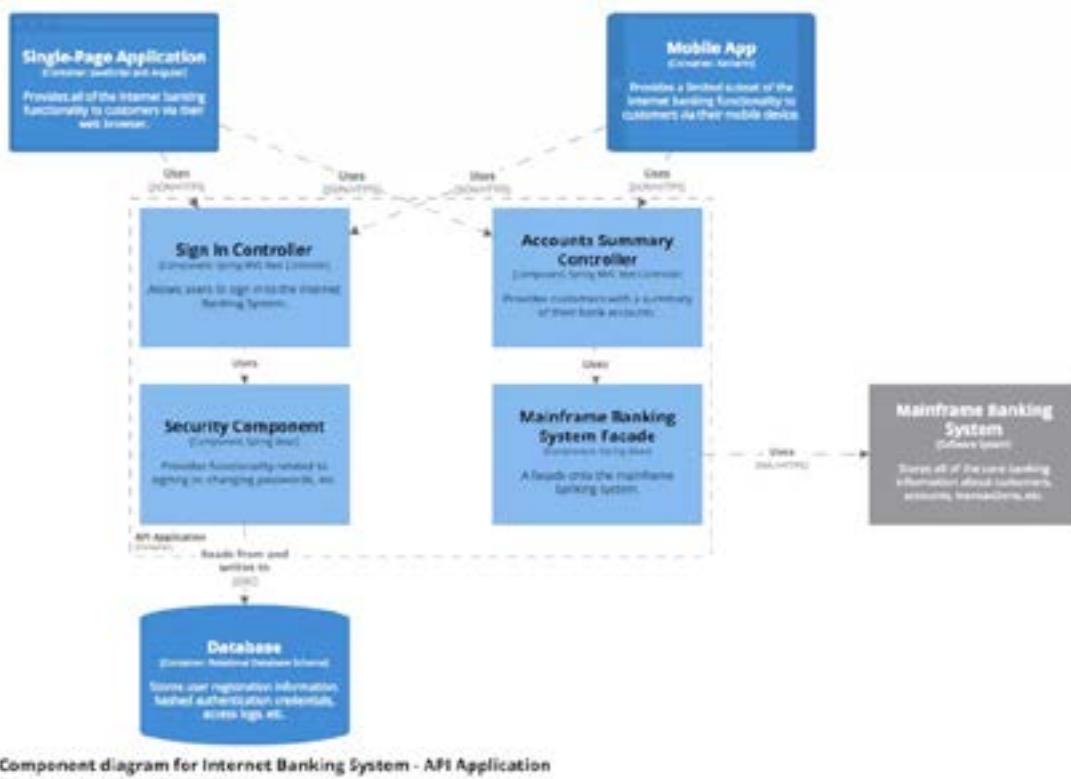
那么在系统上下文图里，方块指代的是软件系统，蓝色表示我们聚焦的系统，也就是我开发的系统（也可能是我分析的系统，取决于我是谁），灰色表示我们直接依赖的系统，虚线框表示的是企业的边界。通过这些图形化的元素表达我们可以看出来各个系统彼此之间的关系。

## 容器图

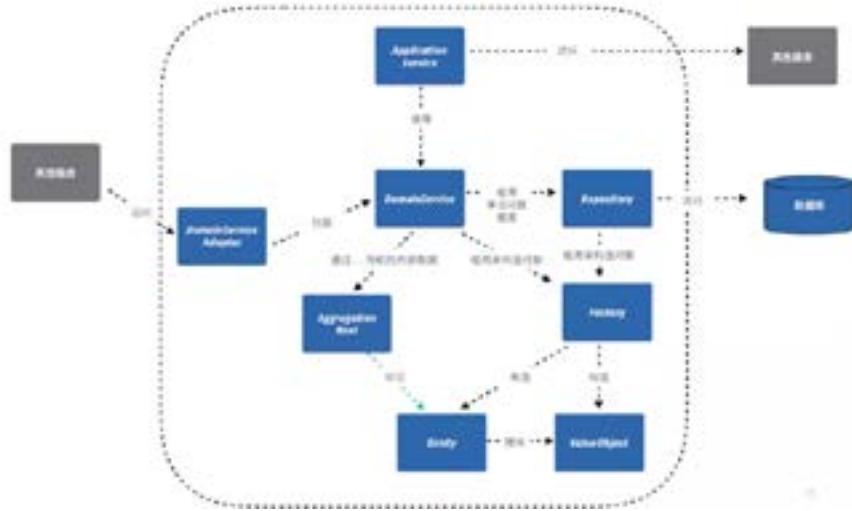


当我们放大一个系统，就会看到容器，如上图所示，C4模型认为系统是由容器组成的。我个人认为，容器是C4模型最大的创举，尤其是在这个单体架构快速崩塌的时代。所谓容器，既不是Docker的容器，也不是JavaEE里的容器，而是借用了进程模型，代指有自己独立进程空间的一种存在。不管是在服务器上的单独进程空间，还是在浏览器里的单独进程空间，只要是单独的进程空间就可以看作一个容器。当然如果你容器化做得好，Docker的Container和这个Container可以一一对应。有了这个概念的存在我们就可以更清晰的去表达我们的架构，而不是总是用一些模糊的东西。

## 组件图



当我们放大一个容器，我们就会看到组件，如上图所示。组件在这里面很好的把接口和它的实现类打包成一个概念来表达关系。我个人觉得有时候一些存在于代码中，但又不是接口的某些东西，比如Service、Controller、Repository之类也可以用组件图来表达，如果你学了一些没有明确抽象层次的架构知识或者一些单体时代的遗留经验的时候，你可以画出来一些组件图，来印证自己的理解，如下图，是我画的自己对DDD战术设计里面的一些概念的理解：



比起模糊的堆砌在一起的文字，这种表达要清晰的很多，哪怕我的理解是不对的，也容易指出和讨论。

### 代码图

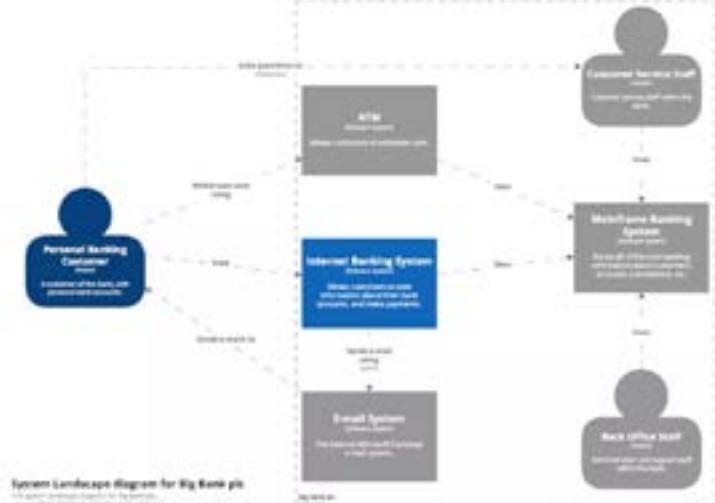
代码图没什么可说的，就是UML里的类图之类很细节的图。一般是不画的，都是代码生成出来。除非非常重要的且还没有写出代码的组件才画代码图。

以上就是C4的核心图，我们可以看到四种不同的抽象层次的定义会让我们更容易固定住我们讨论的层次，这点上我觉得C4是非常有价值的。

## 三张扩展图

架构设计设计要考虑的维度很多，仅四张核心图是不够的，所以作者又提供了三张扩展图，可以让我们关注更多的维度。

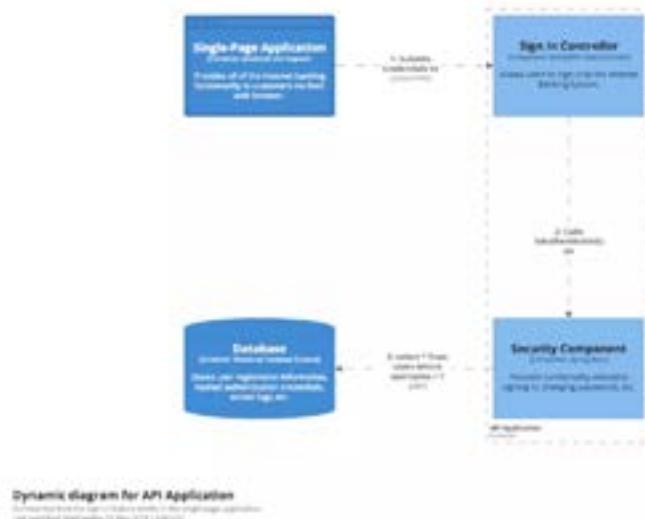
### 系统景观图



看得出来，系统景观图是比上下文图更丰富的系统级别的表达。不像上下文图只关注聚焦系统和它的直接关系，连一些间接相关的系统都会标示出来，那些系统的用户以及用户之间的关系也会标示出来，只是内部的用户会用灰色标记。

这个图有什么用呢？在我们分析一个企业的时候，我们需要一个工具帮助我们把一家公司给挖个底掉，做到完全穷尽，才能看到企业的全景图，从而理解局部的正确定位以做好局部设计为全局优化服务。之前我试过以四色建模的红卡、事件风暴的事件两种工具来教人掌握这种能力，一般来说，程序员学员都无法快速掌握这种顺藤摸瓜的分析技巧，毕竟跟程序员的思维还是有些差异的。但是用了系统景观图之后，学员就毫不费力的掌握了这种分析能力，所以我后来都是用这个图来教程序员探索企业的数字化全景图，效果极好，推荐给大家。

## 动态图

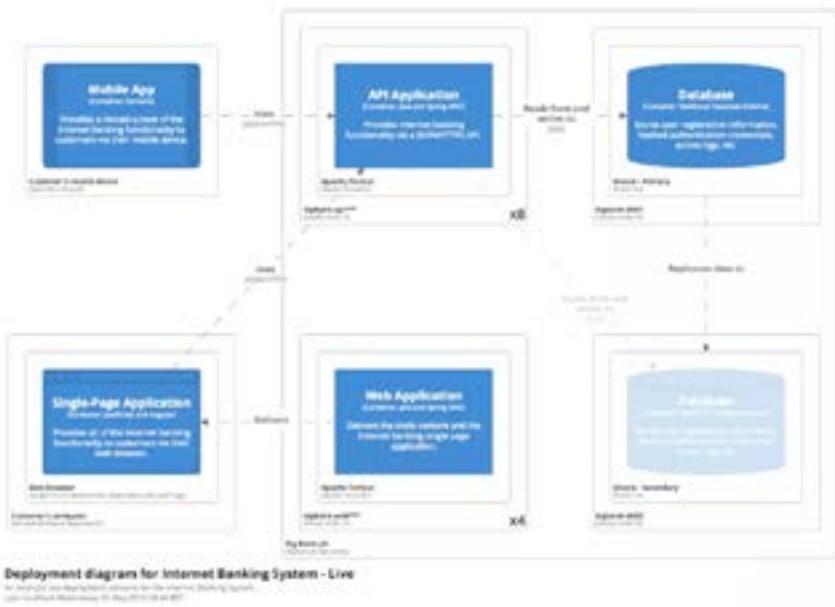


动态图不同于其他表达静态关系的图，它是用来表达动态关系的，也就是不同的元素之间是如何调用完成一个业务的。所以动态图不仅仅适用于一个层面上，它在系统级、容器级和组件级都可以画，表达的目标是不一样的。

我之前曾经写过名为《像机器一样思考》的一系列文章，在文中也发明了类似的图，不同于本文中关系线上标注的是调用的方法、函数，我更关注的是数据，使用效果也很好。

什么时候用动态图呢？举个小例子，我之前做一个内部的小系统，团队中只有一个有经验的工程师带着十多个毕业生，我便要求他们在开始工作之前都画出动态图来，交由有经验的工程师去评估他们的思路是否正确，如果有问题，就在开始之前就扼杀掉烂设计。不管是毕业生还是初级工程师，改代码的能力都比写代码的能力要差很多，所以将烂设计扼杀在实现之前还是有帮助的。

## 部署图



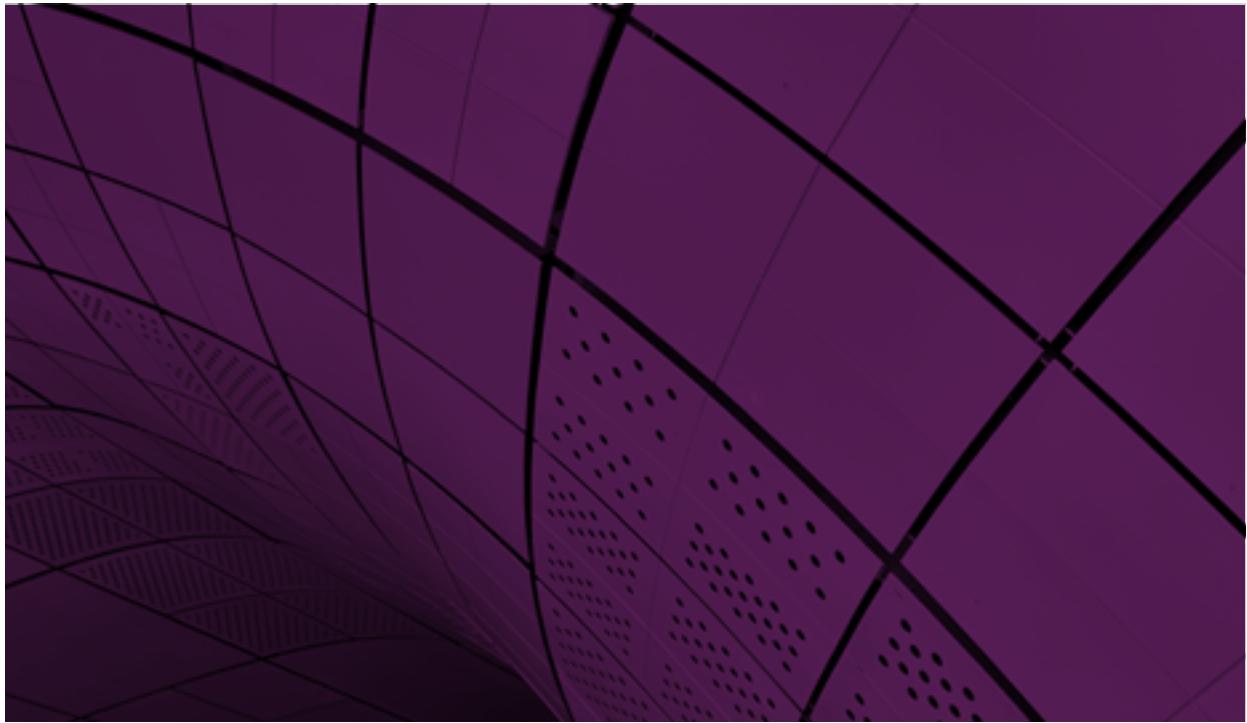
前面的几张图都是站在开发的角度思考，但是一个没有充分思考过部署的架构很容易变成一个运维的灾难。所以作者提供了一个部署图。考虑到DevOps运动如火如荼，这个图可以变成很好的Dev和Ops之间沟通的桥梁。我们在实操中发现，Dev和Ops关注点的不同、语言的不一致，在这张图上表现得非常清楚。

图上最大的的实线框不同于虚线框，它表达的是数据中心，当你开始考虑异地载备的时候它就有了意义。数据的同步、实例的数量都会影响部署图的内容。部署图基本都是容器级的，它能很好的表达出来容器到底部署了几个实例，部署在什么样的操作系统上，一个节点部署了几个容器之类，我们在实际使用中，发现需要考虑的信息太多，自己就抽象出了类似于亚马逊上实例规格的Small、Large之类的术语来表达机器配置，增进了开发和运维之间的交流准确性。

## 为什么C4值得推荐

够直观，对于程序员来说容易理解，容易使用。

我们在开头的时候说过，只有每个人脑子里的才是架构图，如果我们使用一个本身就很难达成一致理解的工具，那成员就会陷入理解的死循环。经过尝试教授不同工具，发现C4模型是最容易理解、最容易使用的工具。可能它的概念是复用了程序员已有的一些认知模型，程序员在学习后都可以迅速的使用起来，并问出一些高质量的问题。



## 总结

在思维的世界里，我们都是盲人，很多东西我们以为自己知道，实际上画出来之后，才发现很多东西没想到，或者想的是乱的，同时别人也才可以给我们反馈。

有了上面的这个工具，我们就可以开始可视化的架构设计之路了，但路上还有一个心魔需要战胜。在我们的文化里，出错是一件很丢人的事情，所以我们喜欢用一些模糊的描述避免被别人挑战，而可视化是让我们精确的描述出自己的理解，来欢迎别人的挑战。这一个坎不太容易跨过去，但是一旦跨过去、大家形成正向的互动之后，我们的进步速度会变得很快，从而把封闭的人远远的甩在后面，获得组织级的成长推力。我自己就在跟别人的交流之后获得了更深入的洞见，本文已经分享了一些，还有一些内容后续再跟大家分享。

# 从架构可视化入门到抽象坏味道

---

作者: 全键

## 抽象的坏味道

上文说过, C4说穿了就是几个东西: 关系-线、元素-方块和角色 (角色不过是图形不同的方块)、关系表述-线上的文字、元素的描述-方块里的文字, 虚线框 (如前文所说, 在C4里面虚线框的表达力被极大的限制了)。

这些东西一点都不新, 我们自己随便找个白板, 无非也是用这几个东西来表达架构, 它的优点在于引进了一些分层, 帮助我们理清思路、也有利于可视化给别人看。

换言之, C4不能帮你做好架构设计, 但是它能暴露出你设计中的问题, 以便于被自己或其他人纠正。

可视化的威力就在这里, 但根据我的经验, 即便你用上了C4也不见得就能表达清楚, 不过好消息是, 我们终于可以聊一些高级的表达问题了。

可视化之后, 我们能看到自己的表达问题, 大概的问题有两个: 抽象层次和抽象粒度。这个是表达方面永恒的问题, 也就是软件设计永恒的问题, 没有万灵丹, 但是用上了可视化手段之后还是有机会让生活更美好一点的。

这两个问题可能太抽象了, 不容易意识到, 那我们可以看图, 从图上的具体表现来发现坏味道。一般会有几个迹象表明我们有可视化的坏味道:

1. 一张图上过分密密麻麻的线。
2. 一张图上太多元素 (也就是方块)。
3. 一张图上太少的元素, 比如角色特别少。
4. 每个图上文字表达不契合, 有的太泛泛, 有的太细节。
5. 无限制的画更多张图, 基本上也就失去了使用图形化表达的意义。

那么对应的手段就有:

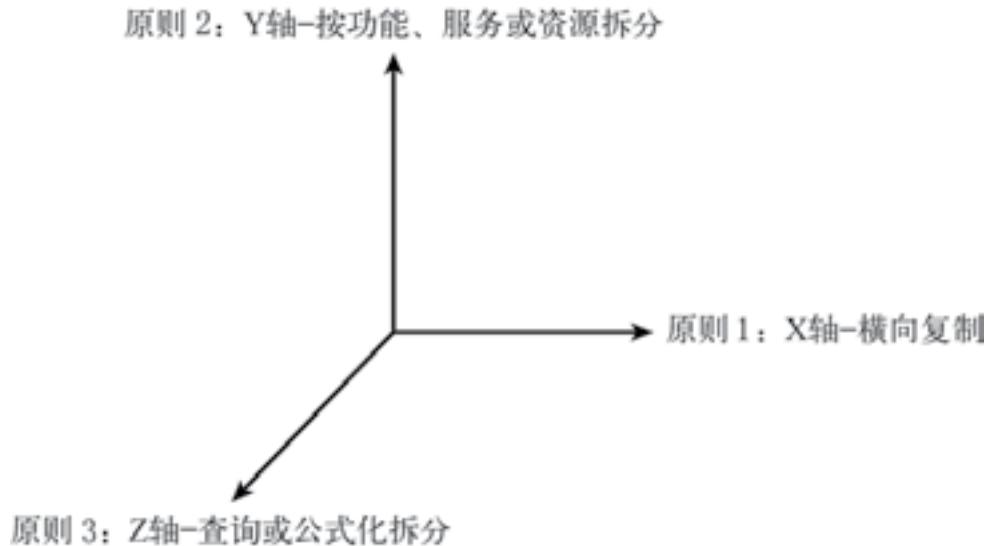
# 合成更大的元素

当我们意识到有密密麻麻的线、太多的元素，闻到这个味道的时候，可以考虑是不是该把里面的一些元素合成更大的元素了。Component可以合成Container, Container可以合成System, 这样就会分成更多的图，每张图就变得没那么多线和元素了。

紧接着会面临下一个问题：怎么合成一个更大的系统，Container是明确的，所以Component合成Container不是问题，问题是Container怎么合成一个系统，为什么是这些Container合成这个系统，而不是另外几个？或者多加几个、减几个？

这个问题没有标准答案，但是有一些其他的框架可以提供一些思考的维度。

比如可以结合akf扩展立方来思考。

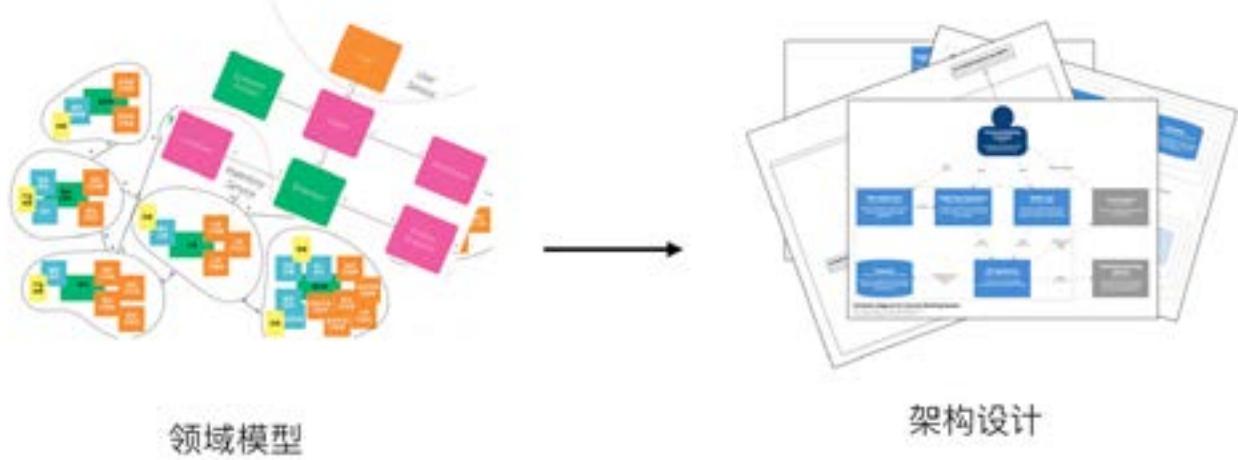


(akf扩展立方)

X轴就比较容易，一方面从容器本身的描述来看设计上是不是支持横向复制的，另一方面则是看部署图。

Z轴相对难一些，只是比较偏技术。比如当技术上有性能瓶颈，则需要注意这一个维度，有时不得不搞出一些特殊的容器出来，有时已经存在这些容器了，他们可能单独属于一个系统（类似于大数据分析的系统），或者一个系统的某一个局部（这就是前面提到的虚线框表达力被限制的地方）。

Y轴给人的感觉是最容易操作的，但实际上却是最难做好的，Y轴的背后是业务，往往我们认为就按业务切成多张图就好了。这种想法就表现出我们往往意识不到业务的真正难度，于是总在这里出问题。如果你能跨过这个心理障碍，决定去认真做一下，那么也有一些工具可以帮助我们做好。



(领域模型与架构设计)

最经典的工具组合就是求助于DDD，结合康威定律和步速，考虑维护的团队、使用的角色、变化的节奏，这块展开就复杂了，有机会再聊。

这里说一个最简单的做法：按照用户角色分。同一种角色，由于其在公司里的职能、职责都是已经被定好的，天然在系统上就有一种隔离性。比如招聘专员、会计、出纳，他们使用的系统肯定是不一样。

但说简单，其实也不简单。我见过一些图，上面的角色只有两个，内部用户和外部用户。而另一些图，细化到了个人的级别，或者把职级都放上去了。所以无论再简单的原则，最后都会掉进抽象的坑。

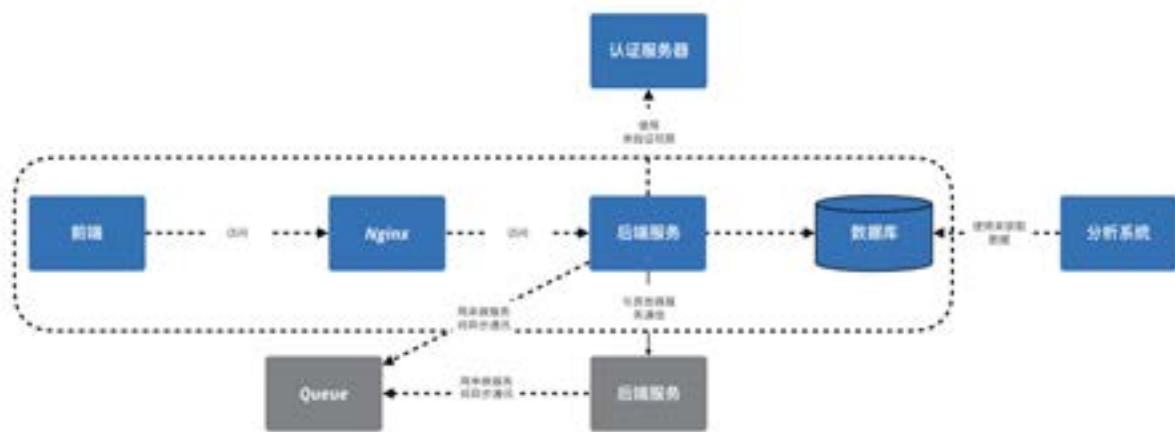
## 画一些共识图来忽略掉一些通用的元素

有时候合成了更大的元素，元素依然很多，线条依然很密。画多张图也不够切分的。这个时候我们可以求助于共识。

人与人交流，如果已经有一些共识存在就可以少废很多话，共识多到一定程度只需要确认一个眼神就完成交流了。所以毫无疑问，做好共识管理，可以大幅简化我们的架构图。

所以在我们做架构可视化的时候，经常会先画一个技术共识图，比如以一个能力建设的数字平台为例，我们就画了一个下面这样的技术共识图：

## 技术共识图

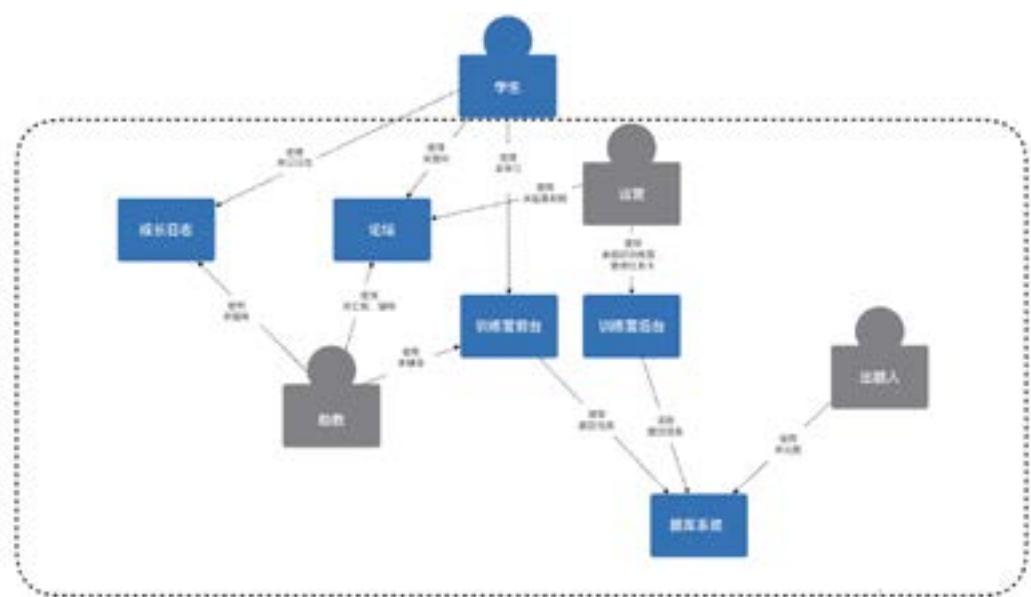


(技术共识图)

在后面画具体的图时，就可以省略掉一些共识的元素，像nginx和数据库就没有了，可以更关注在业务上，而不是技术上。

## 通过制定主题，限制文字的抽象层次

其实上面的技术共识图就是类似的做法，只是用于技术方面，如果用于业务方面，我们可以用一些抽象的名词或动词来代替一类业务，比如下图：



(数字平台系统景观图)

上图是一个系统景观图。当前这个主题是希望人们能一眼看清楚这个系统里面的相关角色都在使用什么系统，他们关注什么，职责是什么。至于具体学什么，怎么学的，都不是那么重要。所以我们就用学习一词代表了一系列的业务。

当主题确定的时候，很多纷杂的信息就没有了。一定要克制住自己试图在一张图上表达足够多信息的冲动。

## 只画重要的图，剩下的交流的时候再画

除了像上面说的，不要试图在一张图上给他足够的信息。同时也不要试图把所有的信息都表达出来。

绝大多数的图可能只在交流具体业务的时候才画，推荐使用动态图。

## 总结

即便有了C4这么，好用的可视化工具。我们依然会看到，自己会掉进抽象的坑。所以在使用的时候一定要注意坏味道，经常自查是不是犯了抽象层次和抽象力度的错，才能做好可视化。这件事上，没有谁能幸免，所以要时常自省，与诸君共勉。



# 技术债治理的四条原则

---

作者：杨政权

“技术债”是 Ward Cunningham 在1992年提出的，它主要用来描述理想中的解决方案和当前解决方案中间的差距所隐含的潜在成本。这种隐喻和金融债务非常类似，这也是这个隐喻的高明之处：为了解决短期的资金压力，获得短期收益，个人或企业向银行或他人借款，从而产生债务，这种债务需要付出的额外代价是利息。

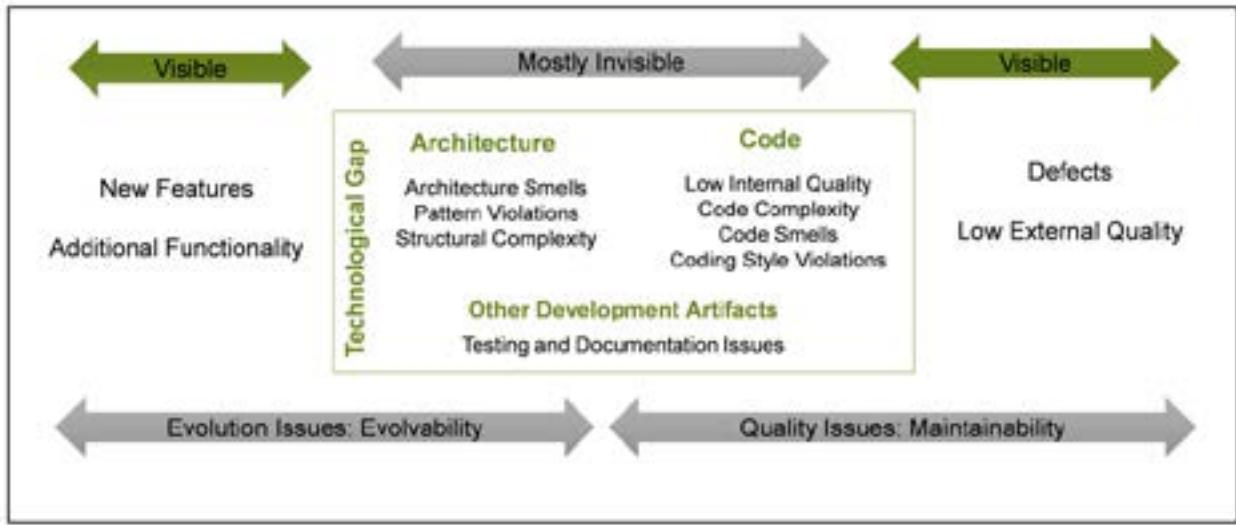
如果短期商业的投资所带来的收益大于利息，这也许是一种明智的做法，但如果入不敷出，收益不及债务产生的利息就会导致资产受损。虽然长期来看这种投资仍然有可能扭亏转盈，但是整个过程风险很大，随时会导致个人或企业破产。

如果把技术债的产生也看做一种投资，那么获得的短期收益可能是快速上线带来的商业利益，比如新的功能吸引了更多的付费用户，解决了短期之内的资金缺口问题；赶在竞争对手之前上线了杀手级应用，并快速地抢占了市场。

不可否认，技术债的存在的确有很多积极的意义，但是我们经常会过度关注积极的因素，而忽略了技术债长期存在所导致的“利息”问题。

# 技术债全景图

卡内基-梅龙大学软件工程研究所 (SEI) 的Robert Nord在《The Future of Managing Technical Debt》提出了“技术债务全景图”(Tech Debt Landscape)的概念，我们可以借助于这个模型定性或者定量分析技术债务所产生的“利息”：



(图1：来自 Robert Nord 的《The Future of Managing Technical Debt》)

这张全景图主要从两个方向来分析技术债对于软件的影响：

可维护性 (Maintainability)、可演进性 (Evolvability)，同时结合问题的可见性 (Visibility) 分析技术债对于软件开发过程的影响。

这里的可维护性 (Maintainability) 主要指的是狭义上的代码问题，即代码本身可读性如何、是否容易被他人所理解、是否有明显的代码坏味道、是否容易扩展和增强。

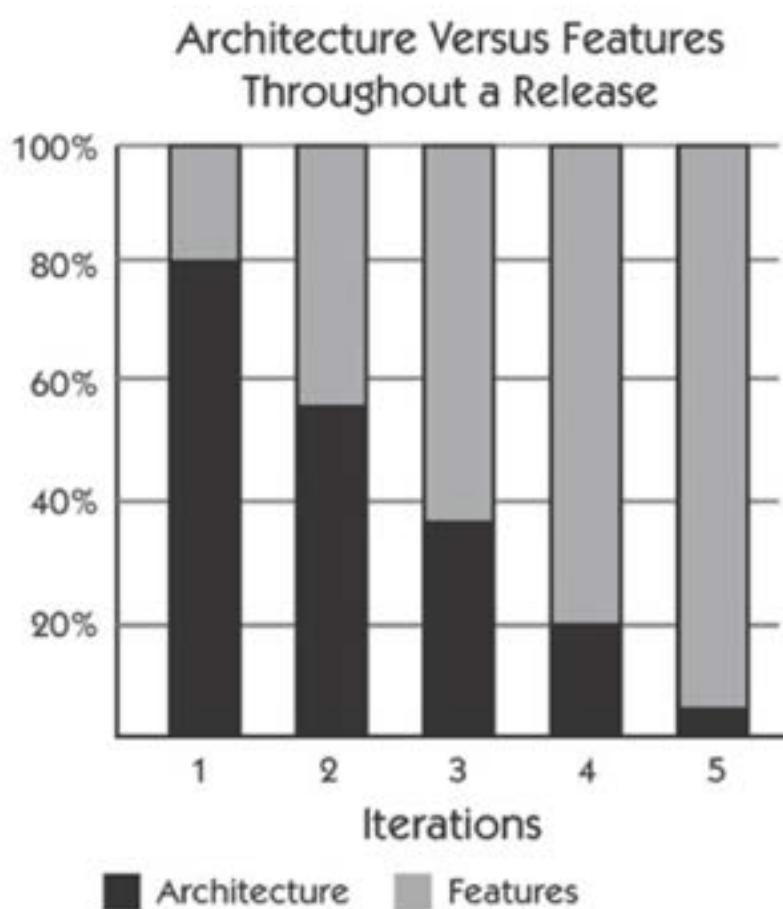
其中可演进性 (Evolvability) 指的是系统适应变化的能力。在生物学中它指的是种群产生适应性的遗传多样性，从而通过自然选择进化的能力。对软件系统来说，可演进性 (Evolvability) 本质上一种架构的元特征 (Meta-Characteristic)，描述的是软件架构趋于目标演进的能力，演进目标并不仅局限于支撑功能快速迭代 (Iteration) 的灵活性 (Flexibility)，也可以是其他的架构属性 (Quality Attribute)，比如高可用性、可扩展性。

针对可见性的分析可以依赖于外部视角：对于最终用户来说，软件功能、设计和用户体验等方面的缺陷，导致用户无法顺利完成既定的业务流程，那么对于用户不可见的代码问题就升级为了可见的质量问题；对于需求提供方来说，臃肿的技术架构、散落各处的业务逻辑导致产品无法快速响应需求变化，导致交付延期，那么对于无技术背景的业务人员来说，难以理解的、不可见的架构问题就升级为了可见的软件交付风险。

## 技术债治理的困境

技术债全景图的分类方法可以帮助我们更全面地了解技术债导致的问题，在可演进性和可维护性这两个维度，我们都可以提取出一些指标来量化“利息”，但是这些指标和业务功能相比终究显得太过苍白无力，并不足以说服主要的业务端干系人并获得对于技术改进的支持。

Mike Cohn (《Scrum敏捷软件开发》作者) 曾经使用下面这张图来阐述这样一种情况：在第一个迭代需要花费大量的时间和精力来进行架构的设计，在后续的迭代中对于架构方面的投资不断降低，期望可以一直延续之前的架构设计并从中持续受益。



(图2：来自 Brent Barton 的《Managing Software Debt: Building for Inevitable Change》)

很多技术管理者或多或少地都会遇到上图中描述的困境：项目开始进入正常的开发节奏之后，技术方面的投资逐渐降低，尤其是在面临交付压力的时候，面对技术债更心有余而力不足，自己的意见总是被忽视，长期积压的技术问题迟迟得不到解决。我相信很多团队在治理技术债时都遇到过同样的阻力。

我们也曾经在团队中多次大刀阔斧地进行技术债治理的变革，团队以头脑风暴 (Brainstorm) 的方式收集技术债，添加到敏捷项目管理工具中统一管理，然后对于这些技术问题进行全局的优先级排序。

在业务端我们也积极地进行了技术债相关理论知识的导入，陈述技术债的产生原因和危害。业务端负责人非常认同技术债治理的意义，甚至主动提出要把技术债放到每个迭代中治理、追踪。虽然在接下来的几个迭代中情况得到了一些好转，但是在1~2个月之后，排到迭代计划中的技术改进比例又恢复了原状，能够真正得到治理的少之又少。

究其原因，团队在反思之后觉得主要有这几点：

### 1. 团队对于技术改进缺少战略思考

在今年的1月份，客户所处的行业竞争不断加剧，新进入者对于客户的威胁越来越大，客户的投资重心也对应地发生了转变，把资源更多地分配给了另一个产品。在技术改进方面团队并没有及时调整技术改进的优先级，让改进的方向和业务的战略方向保持一致。

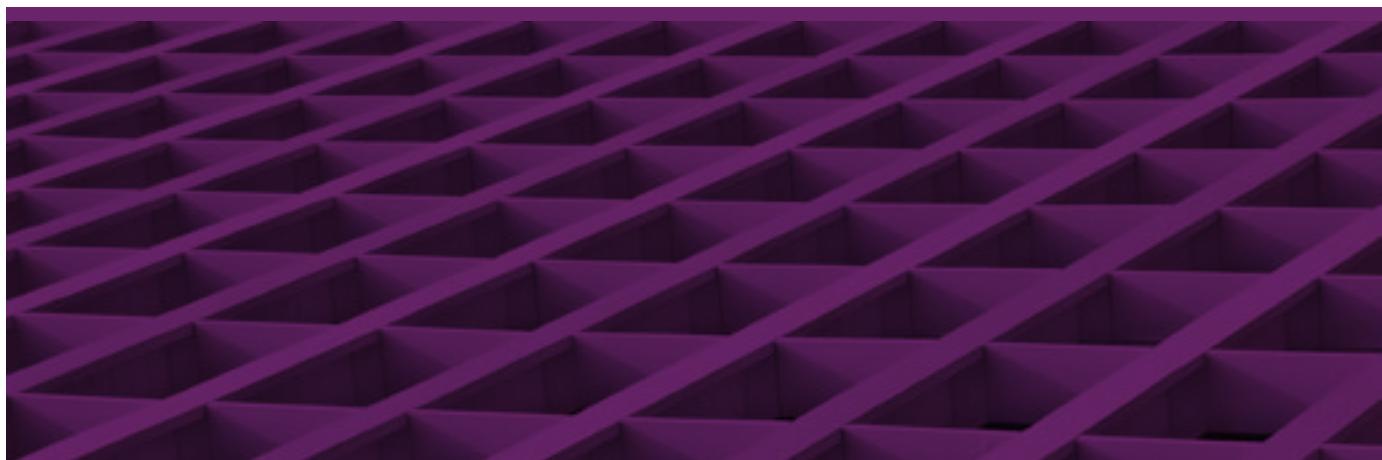
### 2. 代码可维护性问题很难说服客户买单

技术债的影响和收益是难以衡量的，对于这种代码级别的问题更是这样，对于没有技术背景的客户来说，很难用数字量化代码重构的直接收益。况且我们一直给客户承诺的是交付高质量、可工作的软件，除了性能、安全等非功能需求之外，代码质量本身也应该是内建的交付物之一，那代码达到什么水准才能体现出我们在这方面的专业性？

### 3. 效果不明显，客户信心不足

一个典型的例子是应用程序的性能优化，面对一些技术债导致的性能问题时只是隔靴搔痒。虽然加几个索引、调整一下SQL、增加过滤条件或者配置一下延迟加载 (Lazy Load) 可以使问题得到一些缓解，但是并没有触及本质的问题。随着数据量或者并发用户的增加，之前的问题又再次暴露了出来。也许下次在游刃余地解决问题的同时，也可以多质疑一下：领域模型设计是否合理？为什么一定要把这个一对多的关系放到某个实体上？花更多的时间讨论、探索不同的设计区别是什么，评估它们优劣的标准是什么？

基于这几方面原因并结合团队在技术债治理的实践经验，我们总结出了技术债治理的四条原则，也许可以为缓解这个困境提供一些不同的视角和思路。



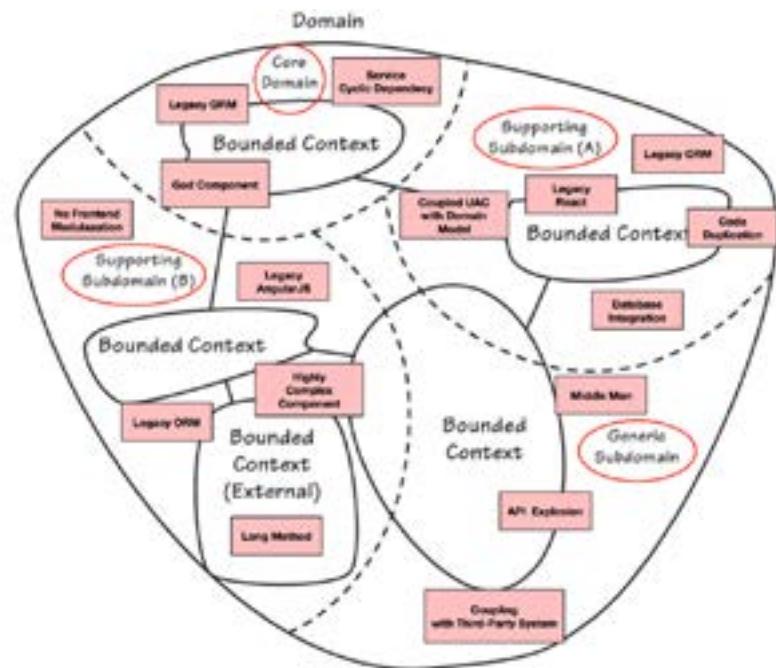
# 技术债治理的四条原则

## 1. 核心领域优于其他子域

识别领域、子域是DDD战略设计的重要步骤，在识别子域之后我们还需要进一步分析哪些是核心域（Core Domain），哪些是支撑子域（Supporting SubDomain）和通用子域（Generic Subdomain）。核心域在业务上至关重要，它提供了区别于行业竞争对手的差异化优势，承载了业务背后最核心的基础理念。《领域驱动设计》的作者、DDD概念的提出者 Eric Evans 是这样描述核心域的：

“The Core Domain should deliver about 20% of the total value of the entire system, be about 5% of the code base, and take about 80% of the effort.

我们可以借助于这种战略建模方式，根据解决技术债之后所产生的收益，将其放置于领域图中的不同位置，可以得到类似这样的可视化结果。在建立关联之后，需要遵循“核心域优先、其他子域次之”的原则来选择技术债。也许我们可以把这种评估技术债优先级的方式叫做“Technical Debt Mapping”。



(图3: Technical Debt Mapping – 基于《领域驱动设计》中的 Bounded Context 修改)

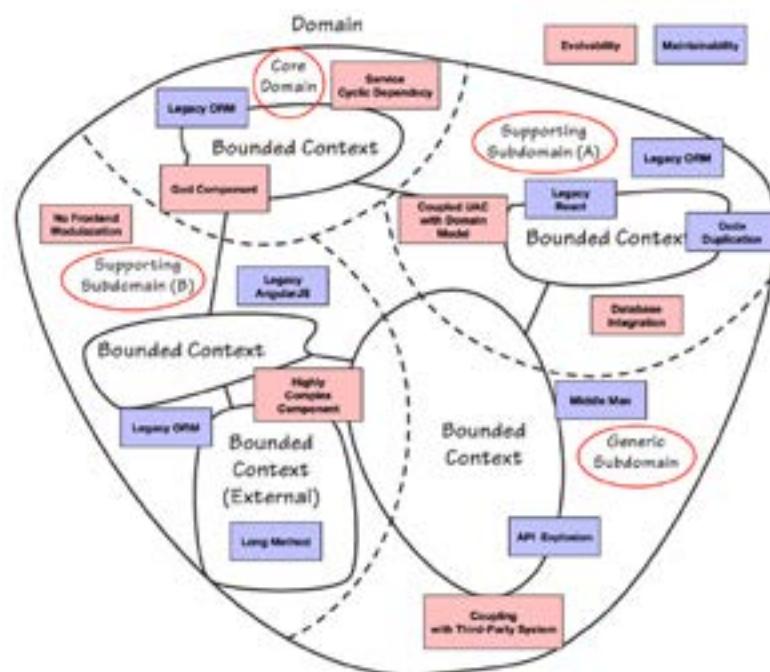
## 2. 可演进性优于可维护性

技术债导致的可演进性问题大多和架构相关，比如服务和服务之间的循环依赖、模块和模块之间的过度耦合、缺少模块化和服务边界的“大泥球”组件等，在添加新的功能时，这些架构的坏味道会给产品功能的迭代造成不少麻烦。比如服务之间如果存在循环依赖的问题，当你对系统进行少量更改时，它可能会对其他模块产生连锁反应，这些模块可能会产生意想不到的错误或者异常。此外，如果两个模块过度耦合、相互依赖，则单个模块的重用也变得极其困难。

可演进性问题可能会直接导致开发速度滞后，功能无法按期交付，使项目出现重大的交付风险。而且问题发生的时候往往已经“积重难返”，引入的技术债务没有在合适的时间得到解决，其产生的影响会像“滚雪球”一样越滚越大。在我所经历过的项目中有一个不太合理的模型设计，由于错过了最佳的纠正时间，为了实现新的业务功能最终不得不做服务拆分时，发现需要修改的调用点竟有1000多处，而且这些修改点很难借助于IDE或者重构工具来一次性解决，不但增加了团队的负担还直接导致了后续功能需求的交付延期。

和可演进性问题相比，高复杂度、霰弹式修改等代码级别问题也很重要，但是相对来说我们更加关注软件适应变化的能力，通过提升软件系统的适应性减少软件最终交付价值的前置时间，快速收集真实用户的反馈，持续不断迭代产品、完善设计。

所以我们在治理技术债时坚持的另外一个原则是“可演进性优于可维护性”。如果把上文提到的可维护性和可演进性使用不同的颜色来标识的话（红色表示可演进性问题、蓝色表示可维护性问题），我们可以得到这样的结果：



(图4: Classified Technical Debt Mapping – 基于《领域驱动设计》中 Bounded Context 修改)

### 3. 明确清晰的责任定义优于松散无序的任务分配

如果我们深入分析一下技术债产生的过程，很容易发现“交付压力”是一个频繁被提及的原因，这也许也是技术债这个隐喻本身存在的一些问题，原本应该体现为内建质量的工作被当做可以取舍、可以之后偿还的债务，导致必要的工作被滞后、被遗忘。有时候浮现式设计 (Emergent Design) 反而成了一种心理安慰的借口：“我知道这里有问题，但是我觉得这个变更需要通过需求来驱动”。

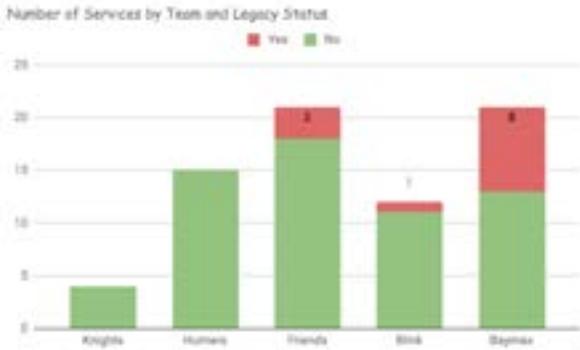
诚然，在开发阶段彻底消除技术债是需要付出额外成本的，在真实的项目中也很难明确定义出这样的边界：哪些部分应该是用刻意设计 (Intentional Design)，哪些应该是用浮现式设计 (Emergent Design)？在 Bob 大叔的新作《架构整洁之道》中也提到这种类似的情形：一开始的设计阶段，如果要划分出完美的架构边界，让两个组件在编译期和部署期相互独立，既要考虑动态的接口抽象、输入 (Input) 和输出 (Output) 数据结构定义，又需要做好组件之间的依赖管理，这些都增加了不少额外的工作量，可以采用一些妥协的做法比如共享组件、策略模式、外观模式 (Facade) 等。

那对应不同的组件、模块或者服务，谁可以来决定采用哪一种设计方法？两个模块之间应该采用完全隔离还是部分隔离？如果采用部分隔离应该采用哪一种方式？团队在针对技术债的治理过程中也经过一段很长时期的混乱期，团队中每个小组并不是一个独立的作战单元，而是一个个特性工厂 (Feature Factory)，每个小组没有清晰的业务职责边界，分配功能特性的时候由各个小组根据兴趣、意愿等主观因素选择功能开发；针对线上问题分配也比较随机，更多是基于团队当时的忙碌程度和带宽，这些都导致了业务和技术上下文的割裂，原本在开发初期的架构设计原则逐

渐退化，刻意选择要在未来偿还的技术债务在各个小组切换功能的过程中也逐渐被遗忘。

理想的情况下每个小组应该是一个价值趋向 (Outcome Oriented) 的团队，负责一个或者多个业务能力，原则上每个业务能力应该有且仅有一个团队负责。然而这种理想的情况在实际项目中又很难落地，即使业务能力和团队对齐，但客户对于不同业务能力的投资并不是均等固定的，如果客户一开始希望把百分之五十的预算花在某核心业务能力的构建上，而且要在3个月之内交付，之后的投资重心又转向了其他方面，在这种情况下要如何保持这种稳定的结构？不过这种业务能力和团队之间映射关系还是必要的，在多个小组合作开发时，可以由所负责的小组驱动必要的合作对话和相关的技术治理活动，触发关于业务上下文和技术上下文的跨团队分享，制定并推广代码规范、架构设计原则同时监督各个团队实施的效果。

但有时候业务能力和技术模块是无法一一对应的，现存的单体应用可能是横跨了多个上下文、提供了多种业务能力，在分配特性、技术债务或者线上问题的时候，高度抽象的业务能力无法提供有效的指导。所以我们采用了一种比较折中的方案——服务责任人制度 (Service Owner)，一个小组负责一个或者多个微服务，每个微服务只由一个小组负责，在分配特性功能、技术债务和线上问题时，需要把服务责任人制度作为首要遵循的原则。由于业务知识和技术上下文的相对集中，在解决具体的软件缺陷时不再浮于表面，团队成员可以更加深入地从需求、技术方案、软件架构等方面着手解决根本问题；针对线上问题通过清晰、明确的责任制度，倒逼团队在开发阶段主动关注软件的内建质量，谨慎判断是否引入技术债。



(图5: 服务责任人制度 - 横纵坐标分别为小组名称和服务数量, Yes / No表示服务活跃状态)

#### 4. 主动预防优于被动响应

这个原则本质上是缩短反馈周期，提前发现潜在问题，除了必要的代码审查流程 (Code Review)、提升团队能力之外还可以借助于自动化工具来提前发现问题。

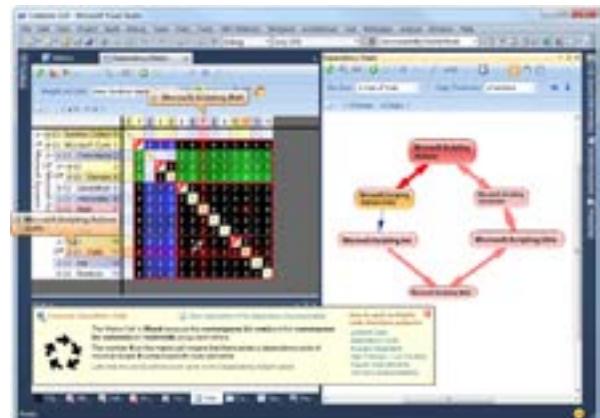
对于代码可维护性方面，很多比较成熟的静态代码扫描工具都可以自动识别这类问题，比如 SonarQube、checkstyle 等，但是仅仅在持续集成上 (Continuous Integration) 运行还不够，需要和团队一起自定义扫描规则，并把检查代码扫描报告作为代码审查的一部分，逐步形成一种正向的反馈机制。

那我们应该如何提前发现不可见的可演进性问题呢？在Neal Ford、Rebecca Parsons 等合著的《Building Evolutionary Architecture》中提出了“架构适应度函数” (Architecture Fitness Function) 的概念，可以给我们发现潜在的架构问题提供一些思路。

“适应度函数”这个概念来源于遗传算法 (Genetic Algorithm)，用计算机模拟仿自然界生物进化机制，

适应度函数用于评价个体的优劣程度，适应度越大个体越好，反之适应度越小则个体越差。在软件系统的不断增量迭代过程中，我们可以基于架构的演进目标，定义出软件架构的适应度函数，来衡量增量的代码是否会导致架构偏离这个目标。

在工具方面使用方面，我们可以借助于 ArchUnit 和 NDepend 帮助我们定义自己项目中的“适应度”规则，这是一个借助于ndepend自动识别组件循环依赖的例子：



(图6: 来自 NDepend 官方文档中的依赖矩阵图)

在技术债治理的过程中，实践可以剪裁，甚至原则也可以妥协，因为比这几条原则更重要的是获得关键干系人的支持。作为技术人员或者技术领导者，不仅要有前瞻性的技术洞察力、锐意变革的魄力，还需要以“旁观者”视角，置身事外地观察自己所处的环境，思考技术改进究竟对于自己、他人、团队、公司和客户究竟产生了什么价值。

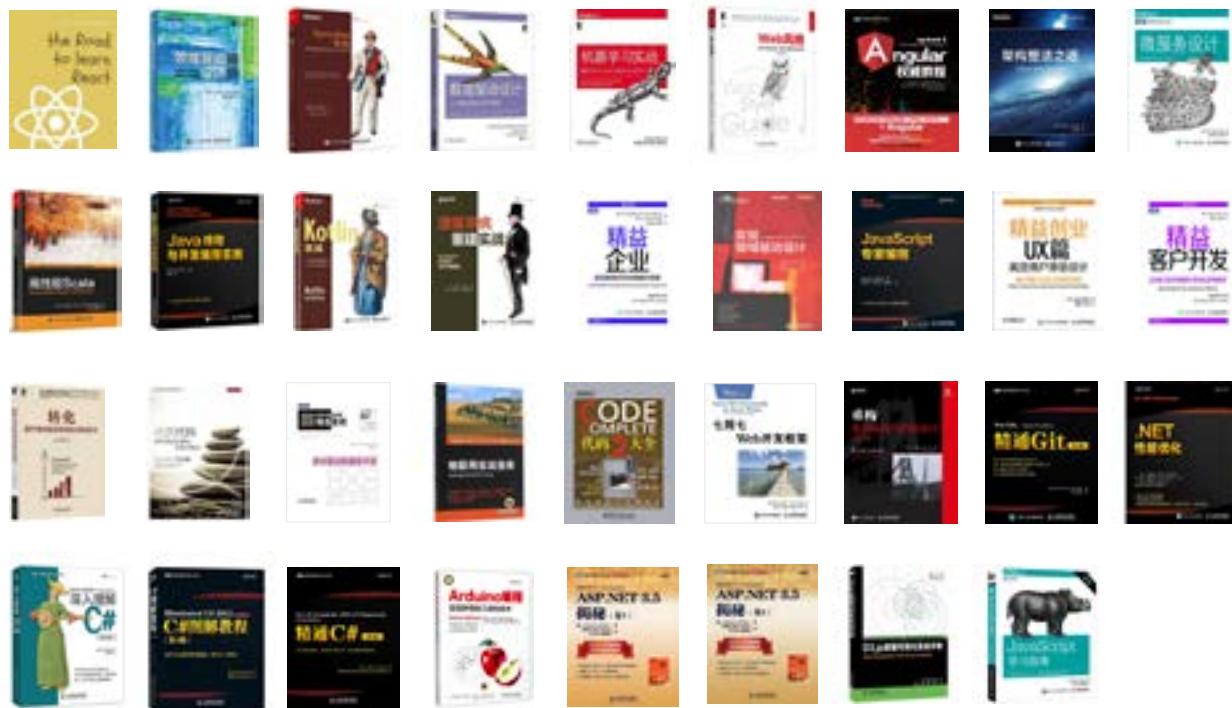
# ThoughtWorks著书/译书

ThoughtWorks员工编著、翻译了大量技术专著，广泛参与开源软件项目，下图展示的书籍均为中国区同事撰写或翻译。

## 著书



## 译书



# 《ThoughtWorks技术雷达》

为了体现技术卓越, ThoughtWorks全球技术委员会(TAB)定期讨论技术战略, 分析对行业产生重大影响的最新技术趋势, 这便是我们看到的自2010年起每年两度的《ThoughtWorks技术雷达》。



技术雷达官网:

<https://www.thoughtworks.com/cn/radar>



技术雷达官网

ThoughtWorks是一家软件咨询公司, 也是一个充满热情、以目标为导向的社区。我们帮助客户以技术为核心, 推动其商业变革, 与他们并肩作战解决最核心的技术问题。我们致力于积极变革, 希望能够通过软件技术创造更美好的社会, 与此同时我们也与许多志向相投的组织合作。

创办25年以来, ThoughtWorks已经从一个小团队, 成长为现在拥有超过7000人, 分布于全球14个国家、拥有43间办公室的全球企业。这14个国家是: 澳大利亚、巴西、加拿大、智利、中国、厄瓜多尔、德国、印度、意大利、新加坡、西班牙、泰国、英国、美国。

ThoughtWorks®

