* [首页](https://km.woa.com/?kmref=km_header)
* [发现](https://km.woa.com/discovery?kmref=km_header)
* [悦读](https://km.woa.com/read?kmref=km_header)
* [乐问](https://km.woa.com/q?kmref=km_header)
* [直播](https://tencent.lexiangla.com/lives?company_from=tencent&kmref=km_header)
* [应用](javascript:void(0))
* [我的K吧](javascript:void(0))
* [[](https://km.woa.com/user/rickyrqzhao)](https://km.woa.com/user/rickyrqzhao)

窗体顶端



窗体底端

[](https://km.woa.com/group/cdc)[用户研究与体验设计部](https://km.woa.com/group/628)

[加入K吧](javascript:void(0);)

* [首页](https://km.woa.com/group/628)
* [微博](https://km.woa.com/group/628/twitters)
* [讨论](https://km.woa.com/group/628/topics)
* [文章](https://km.woa.com/group/628/articles)
* [活动](https://km.woa.com/group/628/events)
* [投票](https://km.woa.com/group/628/surveys)
* [相册](https://km.woa.com/group/628/photos/albums)
* [视频](https://km.woa.com/group/628/videos)
* [文集](https://km.woa.com/group/628/knowledges)

React语境下前端DDD的思考

* [tisontang](https://km.woa.com/user/tisontang)

* 2021年08月31日 16:05

* 浏览(735)

* [收藏(45)](javascript:void(0);)

* [评论(13)](javascript:void(0);)

* [分享](javascript:void(0);)

| 导语 笔者在react项目中尝试使用DDD方法论，为业务对象建模，以更好的理解和把握业务。本文从对业务的思考、react项目的特征出发，阐述笔者在项目中进行的前端DDD探索，以抛砖引玉，和读者一起思考前端DDD的理解和方法。

**前言**

笔者所处的业务团队服务于腾讯投资M&A部门，其中系统涉及的投资相关业务在国内具有典型意义，是覆盖一级市场、二级市场、基金的多品类全流程的投资系统，其中包含了对投资项目本身的业务处理，也包含了投资流程的工作处理（类比OA系统），还包括了其他大部分系统需要考虑的技术建设（例如基于安全性考虑的数据、合同文件、电子签等）。这些方方面面的建设使得这套系统在除了技术本身之外，对业务的掌握能力非常重要。虽然技术团队跟随产品团队完成技术开发，但是在一些具体问题上，如果对业务不熟悉，很难真正处理好开发过程中的某些具体的逻辑代码，而对于投资系统而言，准确实现业务是最基本的要求，否则带来的风险可想而知。这也给我们技术团队带来了巨大挑战，如何在如此复杂的系统中，比较合理的掌握每一个技术细节背后的业务逻辑，确保业务实现的准确呢？

我把目光瞄准了DDD，这个由Eric Evans在《Domain-Driven Design: Tackling Complexity in the Heart of Software》（2004）总结提出的一整套概念和方法论。在阅读人民邮电出版社出版的翻译版本之后，我开始收集各种资料学习和思考，探索在前端践行DDD的可能性。我想，基于DDD的理念，可以帮助我所处的团队合理的形成良好的业务沟通规范和业务逻辑沉淀流程，通过DDD在技术方面的指导，帮助我们构建更加稳固的业务系统。基于这样的想法，我开启了相关的探索。

**什么是DDD？**

Domain-Driven Design是从实际业务出发，站在解决领域问题的角度去思考和设计系统的方法论。在我看来，它包含了两个方面的内容：沟通方法论和研发方法论。

Eric从另一个角度让我们技术人员重新思考自己的工作方式，对于面临复杂业务的系统开发的技术人员，不能一上来就开始进行系统设计和代码实现，他们要做的第一步，应该是和领域专家一道，基于某种大家都能理解的专业语言，构建出一套领域模型，这也就是我所讲的“沟通方法论”。而这个步骤，是我们现在的很多开发团队完全没有考虑过的事情。首先，这里的“领域”（Domain）是指一类事物的集合，比如我们常讲的金融领域、通信领域、数学领域等等。你可以很明显的感知到，“领域”意味着“边界”，意味着某些“共性”和某些“特性”。领域专家就是这些特定业务领域的资深工作者，对该领域的业务非常了解，是该领域的行家。要让技术人员和领域专家坐在一起构建出对应业务的领域模型，必须摒弃他们各自在自己工作范围内的狭义概念，大家找到一种相互都可以理解的概念表达方式，来最终确认各自提出的问题和回答对方都能准确理解。这在我看来，就是世界上最高效的沟通方式，没有之一。

我们可以使用DSL（领域专用语言）来完成这一沟通，领域专家和技术专家（姑且这么叫吧～）基于统一语言，构建领域模型，可想而知，这一模型是无法直接作为代码运行的。开发人员要做的，是基于该领域模型，用代码将其准确实现。而Eric充分考虑到一个系统在技术实现时，和现实世界存在差异，因此，提出了非常多的技术建模方案（Scheme）和配套的架构理念，让技术专家在和领域专家构建领域模型时，有意无意的引导领域专家一起构建出更加符合后期技术实现的模型架构体系。

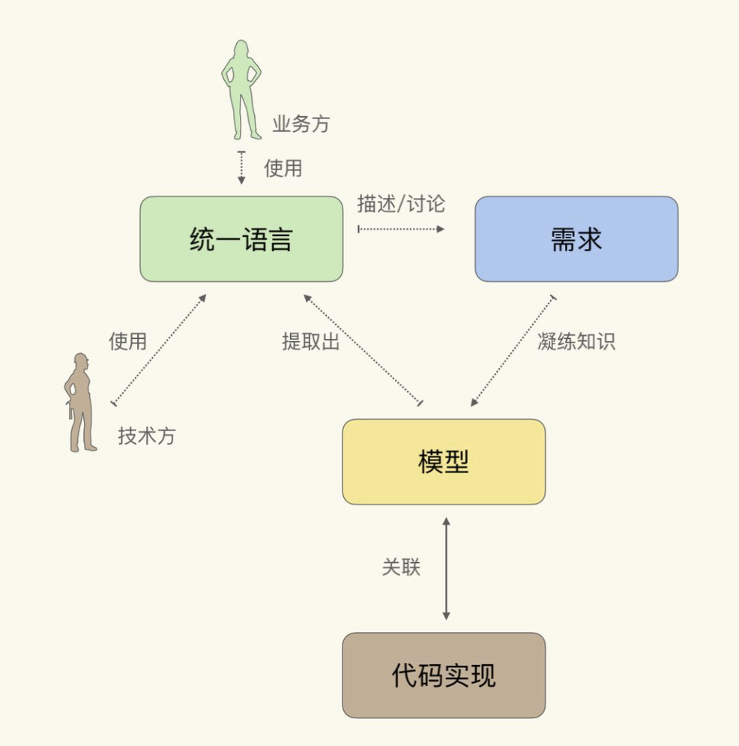


图1 统一语言、领域模型与代码实现的关系

因此，DDD不是某种架构，而是一种设计，不同的业务团队，可以基于这种设计实现一种符合DDD的架构，来帮助自己更好的构建自己的系统。阅读上书过程中，我反复意识到，这本书是写给技术人员的看上去都是技术字眼但实际上是在传播工作方法的一本方法论书籍，如果有这样的想法，那么在阅读此书，特别是后面的章节时，就不会越来越混乱，因为你不再是在这本书中去找到某些具体的实现或架构，而是在思考如何去发现规律。它不能帮助你完成某个架构，但是它可以帮助你思考如何设计这个架构。

现在，回到我们的业务中，为什么DDD正好可以帮助我解决复杂业务的系统设计呢？

**业务的技术语义**

技术上讲，什么是业务？

业务（Business） 专指商业活动，是实现企业生产到利益回收的一个环节，它的总和，构成了该企业盈利活动的整个流程。一般而言，我们所指的业务是企业商业活动中的一个部分，有的甚至小到一个环节，例如“结算”这个环节。业务系统则是辅助这些商业活动的计算机在线系统，以信息化的形式管理和决策企业的商业活动（理论上没有业务系统企业也能运转，但信息化社会没有业务系统会让企业寸步难行）。

业务模块 是以业务系统的建设者（领域专家、系统工程师等）的角度看待业务系统时，将庞大的业务系统，按照某个业务活动的边界，进行划分的某个单元。但是技术上，一般一个模块还是粒度比较大的单元。一般而言，业务模块囊括了系统关于该业务的所有内容，且和其他业务有明显的界限，理论上，可以在脱离了其他业务模块的情况下独立运行。

业务逻辑 是只用代码实现的真实业务的规则映射。简单说，一个业务中，存在什么逻辑，可以通过在纸上画出不同业务对象之间的联系和约束，并将这些联系和约束一条条列出来，形成一个列表，而这列表中的每一条，就是一条规则，这些规则的总和，就是这个业务的全部业务逻辑。既然是一条条的规则，那么我们就可以在代码层面对规则进行管理。对于前端开发者来说，最熟悉的规则管理，莫过于路由管理。

业务系统 本质上是基于人机交互的管理工具。不同角色在系统中管理的内容不同，但总体上，是进行基于业务上下游数据进行职权范围内的业务管理活动。

而从我个人的感官上，业务系统的开发是最复杂的，为什么呢？

我们前端开发对不同的前端应用的开发有非常明显的感知差别，我把前端应用归纳为3类，分别是业务系统、通用应用、工具类应用，它们虽然都是前端应用，但是在开发工作中呈现出来的关注度却非常不同。

表1 业务系统、通用应用和工具类应用的横向对比



由于开发过程中，关注的重点的不同，以至于在开发时，解决问题的思路也迥异。

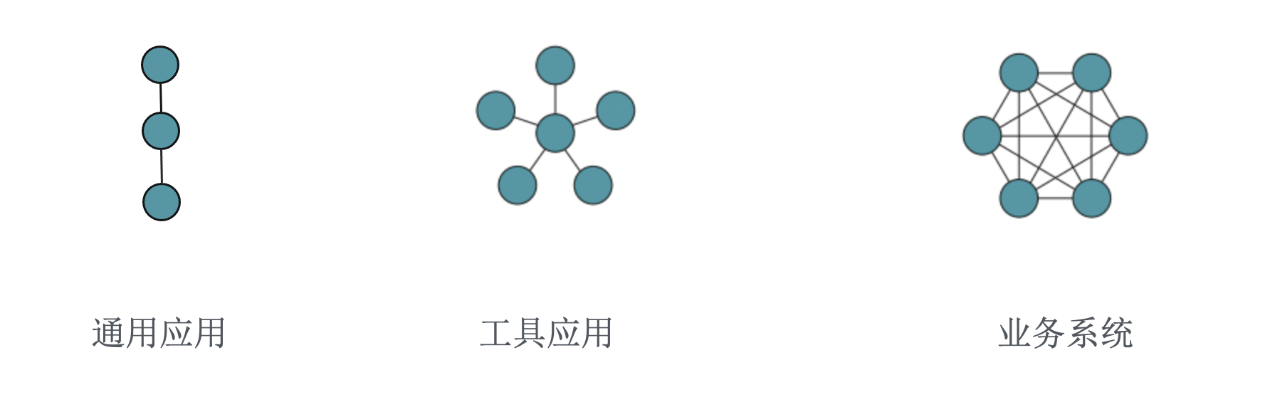


图2 业务系统、通用应用和工具类应用开发时解决问题的思路对比

这种解决问题的思路差异，决定了我们在开发工作中方法论的差异。而业务系统涉及对象多，联系密，维度广等特征，导致在实际开发时，其复杂度比其他类应用高很多。（但在难度上往往并不高，业务系统常常不需要实现非常多需要基于特殊技术才能实现的功能。）

总结起来，业务系统之所以复杂的因素，我认为主要有以下一些方面：

* UI交互的不复杂与反复实现的矛盾
* 特定业务逻辑对原有架构的挑战
* 可持续维护（长期稳定性）与破坏性共存
* 模块与模块间数据、事件通知耦合
* 业务准确性要求高与需求多变之间的时间争夺
* 前后端耦合

基于这些因素，我需要谨慎对待我所开发的每一个功能，因为对于业务而言，这一功能可能会在5年甚至10年后还要继续使用，而如果功能不够健壮，或者设计之初没有充分考虑业务发展后所需要的扩展能力，很有可能在之后的迭代中给自己挖下巨大的坑。我想，这是很多开发普适性强的大众类产品的同学无法体会到的。

**前端语境下DDD的价值主张**

前端需要DDD吗？

这个问题可以细化为，前端需要与业务方领域专家进行沟通吗？在设计系统或功能时，需要基于沟通结构的领域模型展开完成模块的搭建吗？我们需要在前端建模吗？我们需要在前端分层吗？等等。我无法给出标准答案，但我在思考，如今的前端开发，特别是类似和我一样的业务系统项目中，如果我们仍然是按照设计稿完成实现，是否能准确满足需求？或者说，对于前端开发的同学而言，我们是否有必要掌握业务细节，通过一整套的方法论，在代码中将这些细节管理起来？

我想从另一个角度为前端的同学描绘他的工作场景：

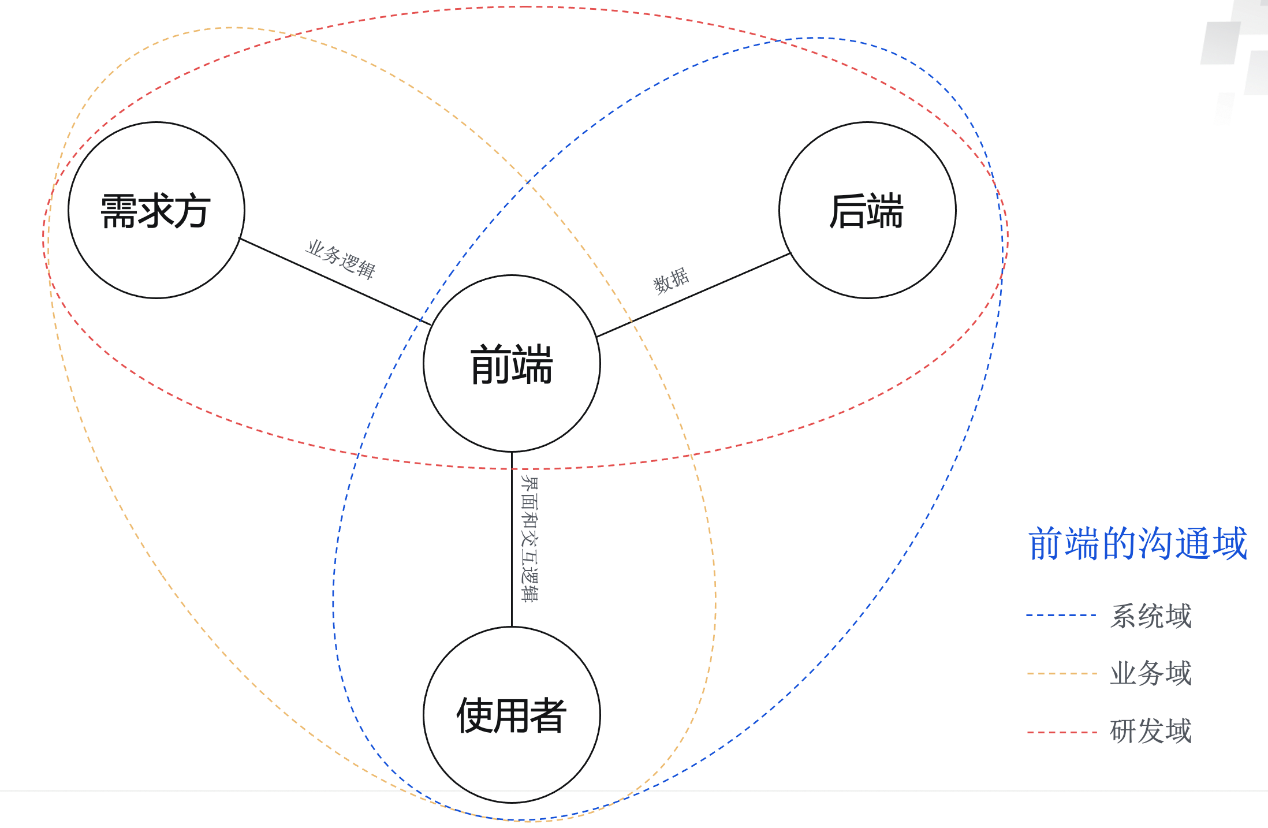


图3 前端开发人员的沟通域

上图中，我描述了前端开发的同学在整个业务开发过程中所处的境地。黑色圆圈表示在这过程中出现的角色，黑色实线表示与前端沟通时两个角色的联系，虚线圈表示当问题发生时参与该问题讨论的相关角色和内容。可以看到，前端的同学在与不同的角色进行沟通时，常常需要切换思维和角度，他所要面临的问题，所处的境地，如果没有一套方法论支撑，很难在繁杂的工作中不迷失。

前端可以DDD吗？

包括Eric著作在内，讨论DDD多半是在后端环境下。作为思想武器，DDD可以帮我们思考如何去设计业务系统架构，前端是否只需要基于后端接口输出渲染界面就可以了呢？或许，前端根本无法按照DDD进行设计吧。

但随着业务的深入，我发现如今的业务系统（基于B/S架构），渐渐的靠近重客户端的方向（类似C/S架构中的C端）。前端代码中的业务逻辑逐渐越来越丰富，甚至很多逻辑只能由前端完成，后端无能为力，或者前后端一起推进时成本更高。运行在C端的前端代码，承载的业务逻辑与UI混杂在一起，导致组件或controller的代码被撑的越来越大，越来越无法维护。我曾指出过，基于vue写的应用，当业务足够复杂时，你根本无法区分vue组件中哪些是业务的，哪些是交互的。而这种情况持续下去，便是组件的不可维护，代码的腐化。

作为前端开发者，我们需要思考，我们如何让我们的代码组织更健壮，更能体现业务的核心逻辑。基于沟通域的思考，我认为，前端所关注的主要内容包含：业务模型、数据服务、UI交互组件体系。

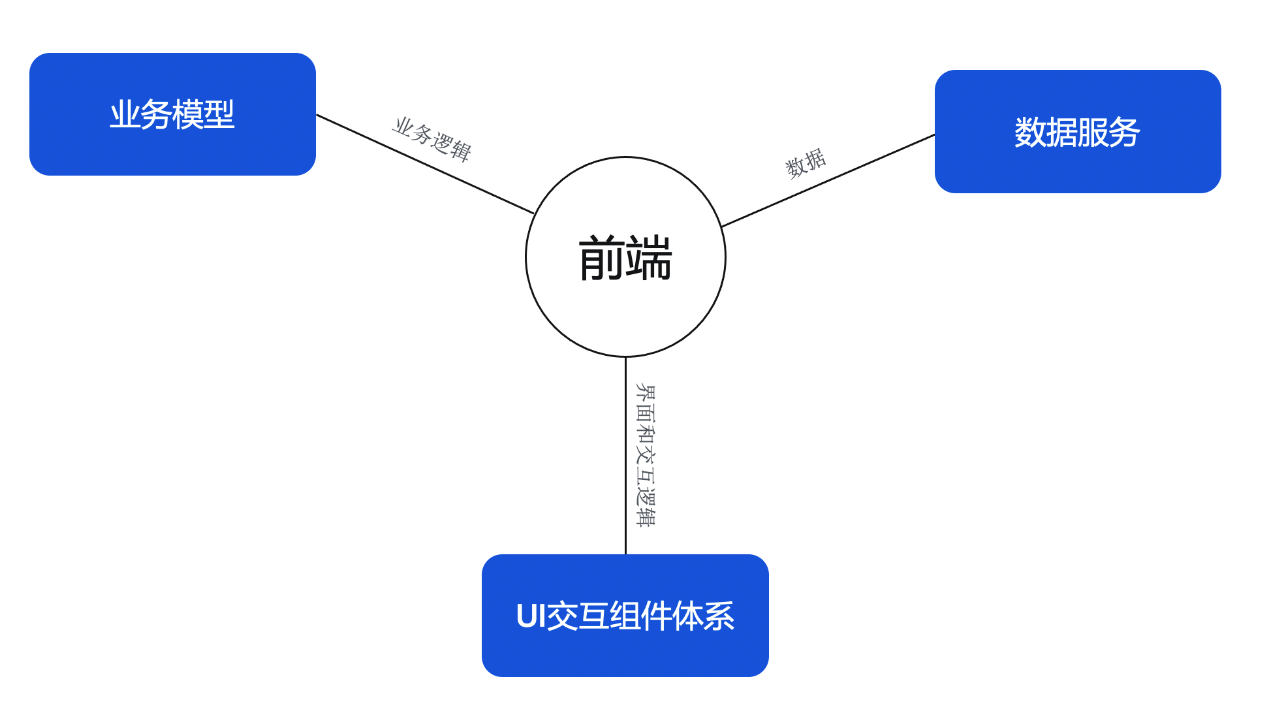


图4 前端关注的主要内容

前端开发注定不可能只关注业务模型，在前端，特别是web领域，除业务之外关注由后端吐出的数据和界面交互是一件必须的事。甚至有这样一种情况，产品文档中指出，“用户点击该按钮时，需要弹出二次确认窗口，点击确认后该签署流转为已完成”。在这个描述中，用户点击按钮弹出确认对话框，是否属于业务逻辑呢？从传统后端的角度，当然不属于，但是，如果去掉这个交互逻辑，能否完整表达这一业务过程呢？这是一个值得思考的问题，前端如果要实施DDD，不可能照搬后端的实践。前后端要解决的问题大不相同：

表2 前后端要解决的问题对比



这也就意味着，在前端语境下，我们关注的内容范畴比后端还要大。

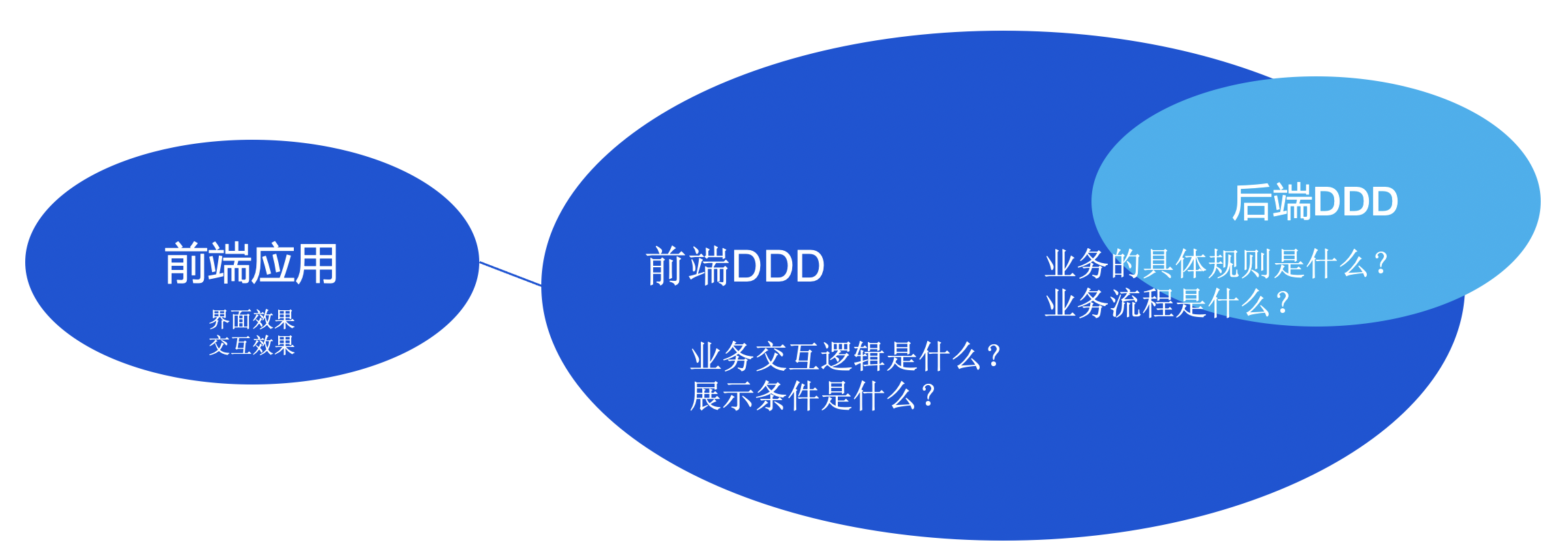


图5 前后端DDD范畴对比

这些思考反过来促使我问自己：对于前端而言，DDD的价值是什么？前端工程师需要重新审视自己的这一职位，从视觉交互的实现者陷阱中跳出来，正视自己作为工程师的一面。解决业务项目中的关键问题，持续的与后端一道提供稳定可靠的业务服务，这才是前端工程师的本职。而这些都是我从DDD中发现的，对于前端而言，实施DDD有肉眼可见的好处：

* 稳定的业务知识体系
* 可传承的代码体系
* 脱离UI的单元测试
* 跨端开发、多端共用的便捷性
* 明确的团队分工
* 需求变更的快速响应
* 持续敏捷

这些好处对于需要持续迭代的项目团队而言，非常有价值，特别是需要持续支撑业务团队在下线完成更多以前无法完成的任务的价值，是业务系统最不可替代的。

**前端领域建模**

我们在腾讯投资系统TIM中践行前端建模已经超过两年，这期间的收益与之前一股脑通过数据绑定的方式相比，可谓天差地别。在此前的开发中，我们虽然觉得麻烦，但是到底能把功能堆砌出来。但是随着时间的推移，一些稍微老一点的逻辑，就再也不敢动，所谓牵一发而动全身，一点改动，重则可能带来整个业务流程的瘫痪，轻则影响相关模块的正确呈现。归根结底，在于我们的系统是线性的（可以回头看上文线性思维属于哪种类型的应用），我们开发团队的知识是不可复制的，一位同学负责一个业务功能，只有他知道这业务里面的逻辑到底有哪些，其他同学即使重新研读代码，也很难梳理出准确全面的业务逻辑，如果需要重构，只能按照代码逻辑慢慢复制。而且由于技术债务，我们的单元测试几乎处于没有的状态，重构的成本和风险可想而知。

两年前，我们开始思考我们所遇到的大部分业务场景的共性，结合开发中遇到的痛点，我们急需有一种方式，可以对我们在应用界面的切换、数据的流转中，有一种把握业务核心逻辑的能力。我们开始了建模探索。经过两年的积累，我现在才能将这些经验总结出来。

**思考单一业务的核心与边界**

我们在开始用代码建模之前，我们需要和领域专家（也就是业务方，或者与我们对接的产品人员）开会讨论某一业务的核心概念有哪些、有哪些可能的事件会发生、它怎么和其他概念发生联系？我们在记事本上梳理出与要开发的业务发生联系的各个概念以及它们之间的连线，这样我们基本掌握了有关这一业务的大部分知识。

当开始用代码表达这些知识的时候，我遇到的首要问题是：哪些是必须的，哪些是与该业务关联但非必须的，我该以何种形式表达这些概念？

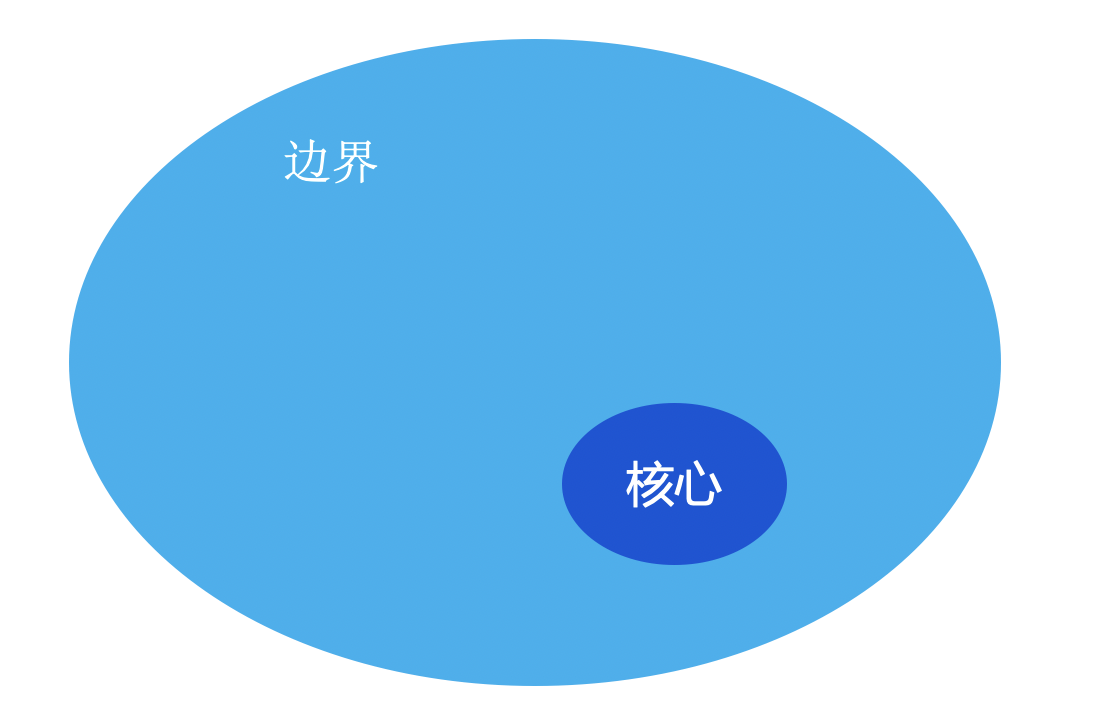


图6 前端领域建模的首要问题是划清核心与边界

使用OOP的范式进行建模是比较常见且直接的方式，通过建立形形色色的class来创建一个又一个的对象。关键的问题在于，这些对象的核心是什么，边界又在哪里？这些都是我们必须在一开始考虑清楚的。

**建模方法**

DDD给我们提供了一些具体的建模方案，例如ENTITY、VALUE OBJECT、SERVICE、AGGREGATE、REPOSITORY、FACTORY等等。

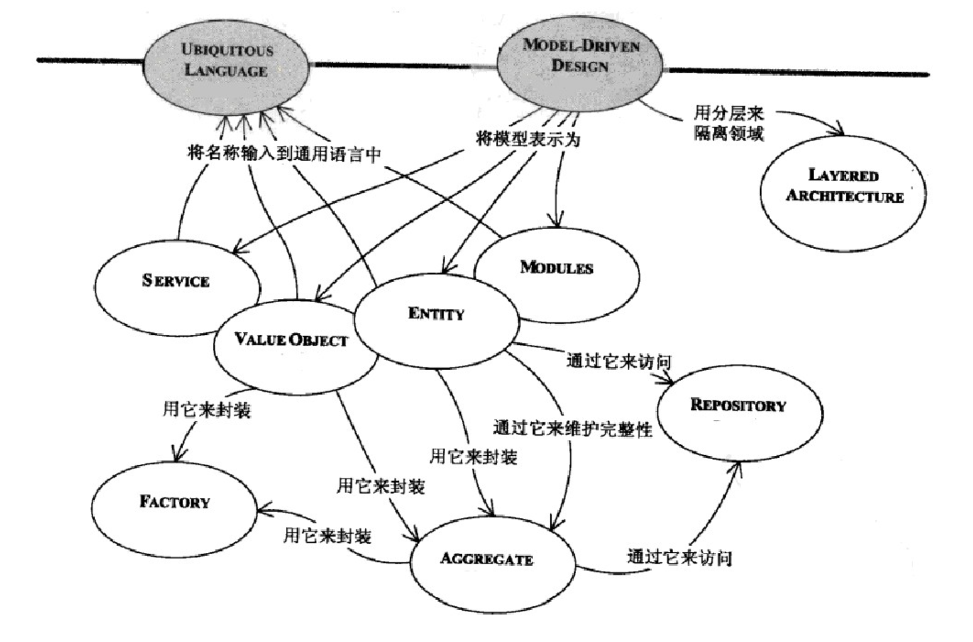


图7 DDD的建模方案

在建立对象模型时，根据对象在业务中所表达的意义，选择其中对应的方案来进行建模。例如，我们为一个投资对象进行建模，首先需要区分，投资对象的边界在哪里，比如一个投资中我们可能要走一些审批流程，会产生一些特殊数据，它们是否属于该投资对象的核心？其次，我们需要了解一个投资它都由哪些资源构成，比如其中有一个字段叫做“投资主体”，投资主体即在投资中作为合同上付钱的那一家公司出现，它自己本身也具有自己的一些属性，那么投资主体是否是投资这个对象的核心资源呢？我们应该处理投资主体？是用一个普通的JS对象来表达，还是创建一个子模型来管理？类似这样的思考，在我们的建模过程中常常发生，而且，有的时候，并不能一击即中，有的时候需要在迭代更新中进行调整。

在长期摸索实践中，我们形成了一套标准的建模步骤（如下图）。在这一标准步骤中，我们抽象出了建模的基类，基于这些基类进行extends，根据每个对象的特征，选择性使用某些建模手段。具体如下：

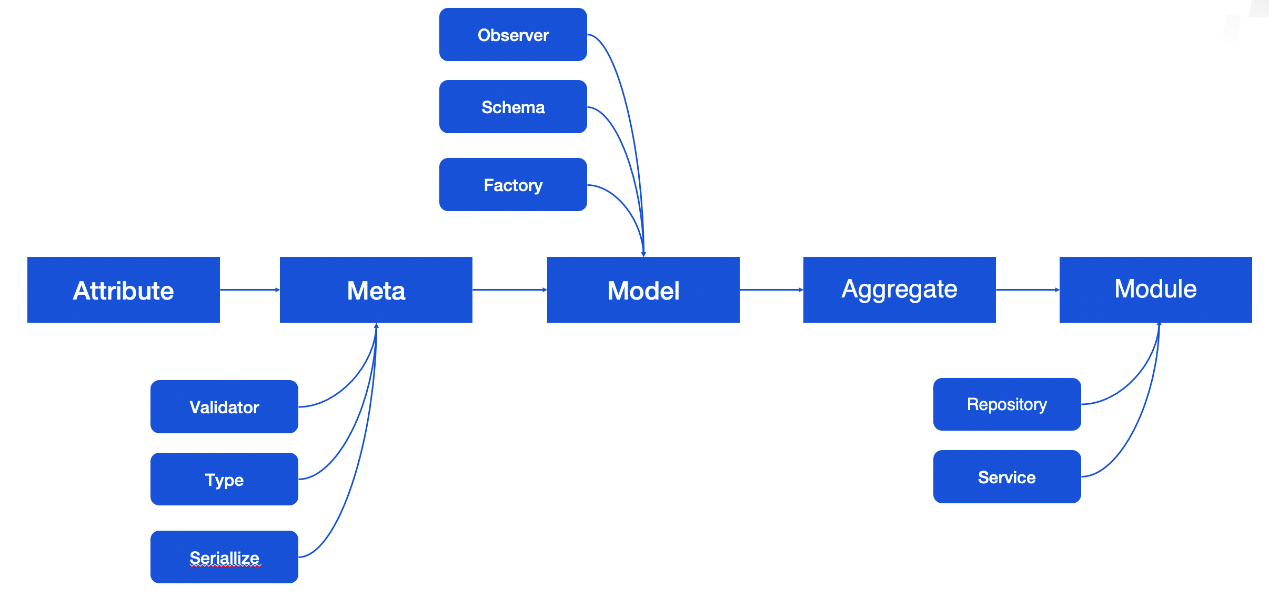


图8 前端领域建模步骤

在上图中，完整呈现了我们在代码层面实现领域模型的所有素材。建模的最小单位是Attribute，即原子属性，原子属性描述一个字段（Field）或属性（Property）的具体属性，即元数据中的某一项。Meta是最小的模型，由Attribute组成，即关于单个字段的模型。Attribute和Meta不单独存在，它们是颗粒度最小的素材，Attribute一般不可复用（或者说没必要），Meta可限制性复用。

在上图中，Model和本文讲的领域模型概念稍有不同，Model是代码层面可以表达完整业务对象的最小单位，它由Meta和其他资源组成。其他资源包括代码层面的方法（概念上的Factory）、静态属性（概念上的Value Object）等。另外，和后端模型最大的不同，前端模型必须为UI视图层留足空间，我们为Model提供了响应式能力，以便在与UI结合时，可以观察到模型内变化，以触发界面的更新。具体从代码层面，我们用Mobx进行举例。



图9 两个简易模型

上图中，我们创建了两个模型Todo和TodoList，其中TodoList是对Todo的聚合，我们会在下文讲。这是两个再普通不过的class进行建模，现在的问题是，如果我们在前端使用这两个模型，我们无法与我们已有的UI框架配合使用，比如在react中或vue中使用。有没有一种方法，可以让我们可以以最小的代价扩展模型的能力呢？我们使用mobx这个库对模型进行改造。

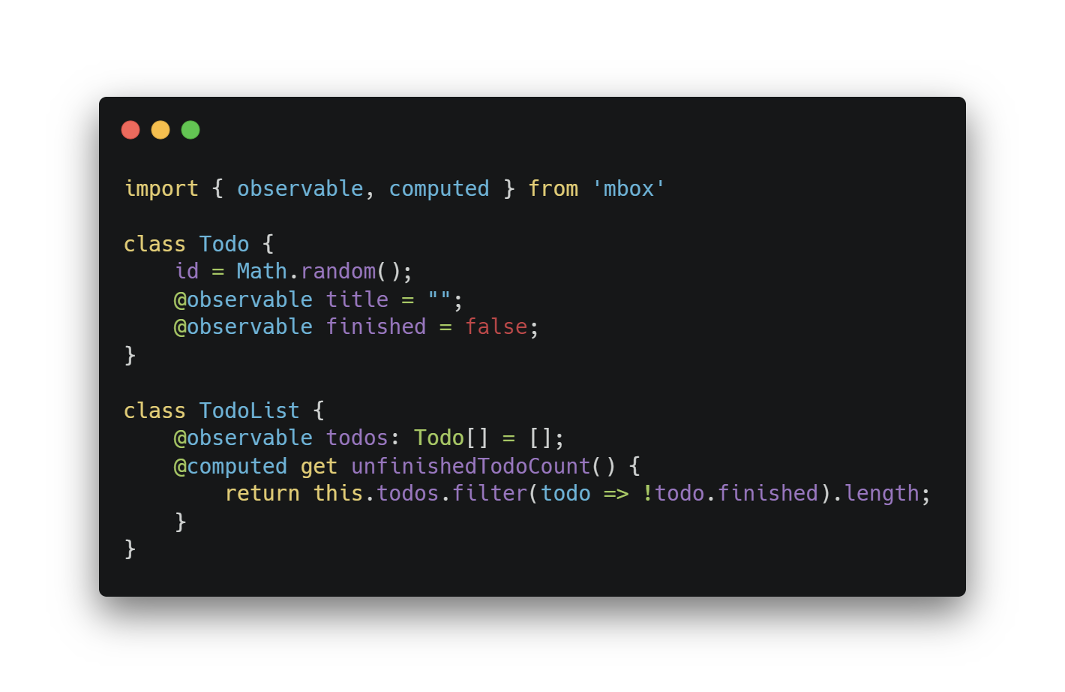


图10 基于Mobx的简易模型

使用mobx提供的装饰器，我们以最少的改动增强模型。这一改动几乎不会对原始模型的原有阅读产生任何干扰，但却使得该模型是可被观察的。再配合mobx的工具，就可以与UI框架配合，就可以在应用中无缝对接。



图11 基于Mobx的模型和UI对接

与后端框架不同，后端以Controller作为入口，对模型和视图进行消费。而在前端，主要的消费者是UI框架，视图是入口，模型需要被实例化为UI状态后消费。因此，像Mobx这样，提供与UI框架对接的工具，是比较合理的设计方式。不过，这已经超出建模本身的话题，此处只是延伸。

我们回到建模步骤，在前端建模过程中，我认为最难的一点在于，如何划定一个模型的边界。很多场景下，一个业务对象的某个逻辑，又依赖于另外一个业务对象的某个信息，跨模型的关系如何去描述？实践中，我们通过嵌套模型来表达，也就是聚合Aggregate。通过Aggregate，我们把高于业务对象本身的逻辑进行梳理。一般而言，一个聚合根囊括了该业务的所有实体，也就是说，它界定了该业务在核心实体上的边界，该业务只在这些对象之间运转。

但业务的运转不仅只包含实体，也包含业务的流转逻辑，即概念上的领域事件，以及背后的领域服务，说人话就是业务对象在该业务运转过程中发生变化。对于前端而言，与后端对接数据也是很关键的一个点，我们基于Respository方案建立可响应式的接口数据管理方案，创建Service单例来在整个应用中响应事件，流转业务对象的状态，从而让界面跟着业务的流转而发生变化。

以上的建模成果被组织在一个Module（模块）中，一个模块不是一个单一文件，它是有特定目录结构的一系列文件组成的功能单元。不过需要注意，这里的Module和我们应用开发中通常讲的Module概念上稍有不同，这里的Module主要是对领域模型完成代码实现后组成的组织单元，本质上，它还是建模的一部分，和我们通常意义上的业务模块（包含UI等）有所区别。

**在React项目中设计业务模块**

作为优秀的视图驱动库，react实现了完美的从数据到视图的映射。对比vue的优势在于，它对数据本身的形式没有要求，在vue中，你给定的数据包含getter(Object.defineProperty)或由Proxy创建抑或由特定的class实例化而来，会导致vue丢失部分响应式的能力，而react中没有这层限制，因此，在我们实施DDD的过程中，更有亲和力。

但react虽然解决了数据到视图的映射，却没有在反过来的视图到数据的映射上提供方案。我们在react中，会在onClick等基于它内置的合成事件系统中执行回调函数来完成视图到数据的处理，然而这种处理显然是不利于建模的。因此，在react本身之外，我创建了一套基于RxJS的单例服务来处理来自交互的事件与模型层的绑定。在具体的react组件中，我只暴露给组件它渲染和交互需要用到的数据（状态）和事件接口。

我称这种独立于react组件本身之外的体系为“无视图交互模型”。该模型在撰写时，站在视图层的角度，处理业务模型的实例化、修改等，处理来自视图层的交互事件等等，但是，在该模型中，没有任何具体的UI实现。

如果你还有印象，你可能还记得我在前文问到“用户点击按钮弹出确认对话框，是否属于业务逻辑呢？”，在“无视图交互模型”的设计下，我们可以将“用户点击按钮弹出对话框”这一交互转化为模型的一个部分，在该模型中，它提供了用户点击动作的接口，而该接口处理时会流转模型内持有的其他具体业务模型，进而达到需求文档中所描述的这一要求。但在具体UI中，这个按钮长什么样子，在哪个位置，弹出框样式，都需要在UI层（组件中）具体去实现，而在实现时，开发者不需要考虑该按钮点击事件的具体效果，只需要调用模型接口即可。

截止到这一步，你会发现，在我们的业务模块中，还没有任何有关UI的具体实现，但是，我们几乎已经把需求文档中有关业务的实体对象、某些与业务流转相关的交互，都用代码表达出来了。

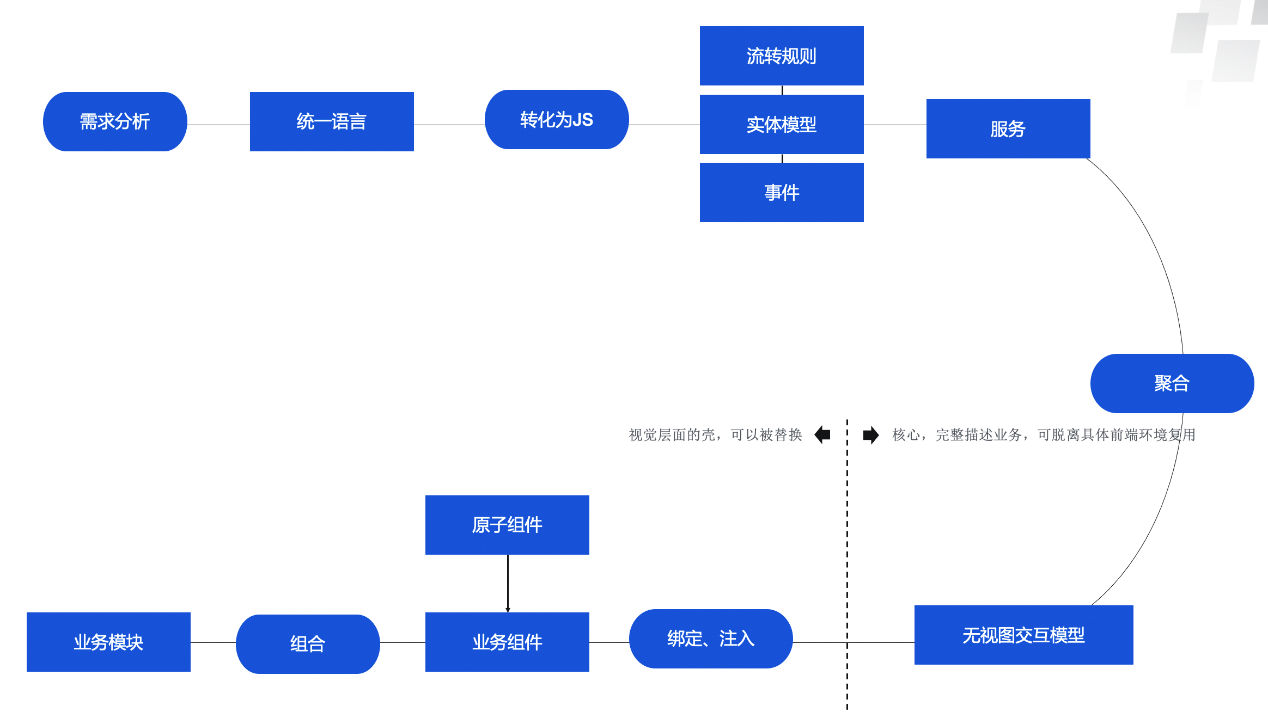


图12 项目中实现业务模块的具体流程

从需求分析，基于统一语言建立领域模型，到具体代码去实现领域模型，再到建立无视图交互模型，到这里，我们有关业务的建模基本完成，但我们还没有任何的界面效果。对于以前我们拿起react就开撸的开发方式而言，简直不可思议，怎么界面都还没有就已经一大堆代码了？

我们的系统包含PC端、APP端和微信内嵌H5，APP端和H5端交互基本相同，稍有些细节差异，但PC和移动端的差异大的就不止一点点。然而，对于某一业务模块而言，两端的交互虽然不同，却在业务逻辑上（包含基于业务逻辑的交互逻辑）是必须一致的，否则业务本身就会出现问题。

上图中，虚线的右侧即我们多端共用的部分，业务模型在各端复用并非完全为了代码重用，更多的是为了保证业务的一致性。而虚线的左侧，则是我们常见的react组件体系的编写。对于不同端，我们的组件很少有能复用的，基本上各自一套，但却能够保证在业务逻辑上没有出入。一旦业务（数据和交互）被业务模型规范下来，对于不同端的视图层，就是换皮操作（有点夸张）。

这一体系给了我们很多想象的空间，虽然我们团队目前并没有再深入，但是我个人的角度而言，我们可以在这一模式基础上，做到更容易的业务单元测试、基于后端输出的统一UI DSL、业务模块服务化、小程序等等。

**基于DDD的前端应用架构分层**

分层也是DDD的重要思想，在Eric原书中，他对系统分层做了梳理，同时，在行业中，后续继承者们对该分层进行了扩展，主要包括：依赖倒置四层架构、六边形架构、Clean Architecture等。四层架构典型示意描述如图：

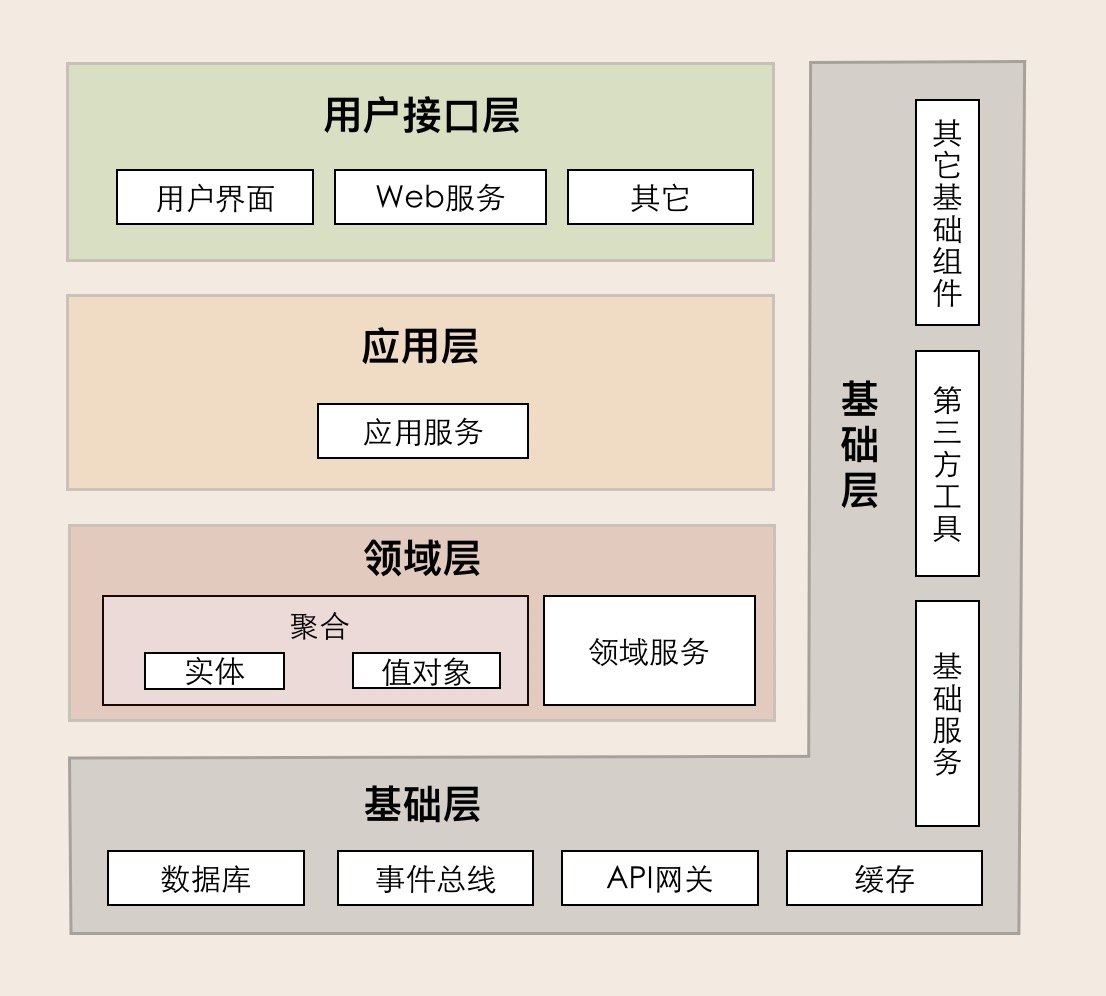


图13 遵循DDD的系统分层架构示意图

对应到前端，与后端不同的是，前端是胖UI瘦数据，前端不需要去考虑数据的存储读写所带来的一系列问题。但是，在整体的分层划分上，前后端具有一致性。



图14 遵循DDD的前端应用分层示意图

在我以前的前端应用开发中，也有分层的思想，但是分层实践很微弱，大部分情况下还是把所有逻辑杂糅到react组件和状态管理中。这也是我们以前的代码在开发完一段时间后可维护性就极速变弱的原因。而按照DDD的设计，我们将代码分层组织和管理。其中的核心层是领域层，这一层决定了整个应用的大部分代码逻辑，虽然在控制层和UI层也有一些逻辑，但是它们多是处理交互的，与具体的业务无关。它完整的描述了这一系统所反应的业务面貌。也正因如此，在多端复用（例如PC和APP两端写同一个业务）时，只需要重写UI层，而不需要再写其他层。

**结语**

复杂的业务系统无论前端还是后端，都面临着巨大的挑战，除了系统本身的功能之外，最大的风险在于业务逻辑需要被准确实现的同时，给研发团队的实现时间却很紧张，紧随而来的系统的代码在长时间迭代中，越来越难以维护。DDD从沟通和研发两个角度为我们提供方法论，是面对复杂业务时的思想工具。它启发了我们，让我们在工作方式上不再莽撞的马上开始写代码，而是先与业务方深入沟通，掌握业务的知识网络，基于统一语言进行领域建模，之后再来考虑如何在react中结合，开发整个应用。在前端语境下，由于前端关注的内容的异质性，我们不可能直接照搬后端的DDD实践，不得不探索前端DDD的特殊途径。基于DDD的设计，我们的架构剖离出不同的分层，在领域层和控制层完完全全描述了业务需求，因此，可以不考虑UI层就可以进行业务编程，并实施有效的业务单元测试。而对于跨端复用而言，就是换UI壳的处理，内在的业务逻辑代码可直接复用。

以上是我在前端DDD话题下的探索，大部分是概念上的梳理，没有具体的实践代码，很大的原因在于DDD的实践没有统一标准，一旦需要整理，就需要把很多的代码牵扯出来，也不是很好举例。DDD也不是唯一的选择，它给了我们启发，也允许我们在它的基础上持续颠覆。本文的理解也仅限本人在工作中的总结与思考，并不一定正确，如有不足之处，欢迎大家批评指正。



最后更新于 2021-08-31 17:21

如果觉得我的文章对您有用，请随意赞赏

赏

[1人已赞赏](javascript:void();)

[](https://km.woa.com/user/v_gxzhang)

仅供内部学习与交流，未经公司授权切勿外传

分类：[技术开发](https://km.woa.com/group/cdc/articles?cat_id=46139" \t "_blank) 标签：[ddd](https://km.woa.com/group/cdc/articles?tag_id=38662" \o "ddd" \t "_blank)(4) [前端建模](https://km.woa.com/group/cdc/articles?tag_id=217111)(2) [领域模型](https://km.woa.com/group/cdc/articles?tag_id=5966)(2) [react](https://km.woa.com/group/cdc/articles?tag_id=95464)(1)



本文专属二维码，扫一扫还能分享朋友圈

想要微信公众号推广本文章？[点击获取链接](javascript:void(0);)

我顶 (34)

收藏 (45)

* [转载](javascript:void(0);)

* [收录](javascript:void(0);)
* [评论](javascript:void(0);)(13)
* [反馈](javascript:void(0);)

分享到

大家评论

[](https://km.woa.com/user/artisan)

[artisan](https://km.woa.com/user/artisan)

2021-08-31 17:24:08

大佬啥时候开课？

 顶 (1) [回复](javascript:void(0);)

[](https://km.woa.com/user/v_gxzhang)

[v\_gxzhang](https://km.woa.com/user/v_gxzhang)

2021-08-31 17:24:14

tql

 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/zawaliang)

[zawaliang](https://km.woa.com/user/zawaliang)

2021-09-01 16:26:53

tql

 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/bobbli)

[bobbli](https://km.woa.com/user/bobbli)

2021-09-01 18:51:04



 顶 [回复](javascript:void(0);)

[](https://km.woa.com/user/jerizhang)

[jerizhang](https://km.woa.com/user/jerizhang)

2021-09-02 10:05:16

DDD 第一个D讲的是领域，数据展现是领域是表示 ，存储是领域内容序列化的形式，在界面上用的比较多的是状态机视图，可以看看《软件方法》一书

 顶 [回复 (1)](javascript:void(0);)

[](https://km.woa.com/user/tisontang)

[tisontang](https://km.woa.com/user/tisontang) (楼主)

2021-09-02 12:28:42

向大佬学习

[回复](javascript:void(0);)

[](https://km.woa.com/user/sunlei)

[sunlei](https://km.woa.com/user/sunlei)

2021-09-02 10:20:47

个人觉得DDD模式下 前端接入微前端可能会有一定效果

 顶 [回复 (1)](javascript:void(0);)

[](https://km.woa.com/user/tisontang)

[tisontang](https://km.woa.com/user/tisontang) (楼主)

2021-09-02 12:33:21

这是另外一个话题，我在组内分享过，只是没有在km上发表。  
简单讲，康威定律告诉我们，技术的架构和组织的结构总是匹配的，大的架构总趋向于拆分成小的组成，而微前端和微服务给了这种拆分实践的可能。

[回复](javascript:void(0);)

[](https://km.woa.com/user/justynchen)

[justynchen](https://km.woa.com/user/justynchen)

2021-09-02 10:32:10

“我曾指出过，基于vue写的应用，当业务足够复杂时，你根本无法区分vue组件中哪些是业务的，哪些是交互的。” 其他框架也没差吧，怎么把业务和交互区分开是个好课题

 顶 (1) [回复 (2)](javascript:void(0);)

[](https://km.woa.com/user/kennthpeng)

[kennthpeng](https://km.woa.com/user/kennthpeng)

2021-09-02 11:14:26

我也觉得这个似乎和框架无关。作者能举个例子进一步讲解一下不？

[回复](javascript:void(0);)

[](https://km.woa.com/user/tisontang)

[tisontang](https://km.woa.com/user/tisontang) (楼主)

2021-09-02 12:27:59

可以深入讨论一下

[回复](javascript:void(0);)

[](https://km.woa.com/user/sheepluo)

[sheepluo](https://km.woa.com/user/sheepluo)

2021-09-02 10:51:15

很想和作者进行进一步沟通，更加深入的理解逻辑和 UI 的分层

 顶 [回复 (1)](javascript:void(0);)

[](https://km.woa.com/user/tisontang)

[tisontang](https://km.woa.com/user/tisontang) (楼主)

2021-09-02 12:28:25

可以再交流哈

[回复](javascript:void(0);)

[](https://km.woa.com/user/rickyrqzhao)



[切换到更多功能](javascript:void(0);)

关于作者

[](https://km.woa.com/user/tisontang)

[tisontang(唐霜)](https://km.woa.com/user/tisontang)

TEG\用户研究与体验设计部\墨子工作室员工

作者文章

* [基于quickjs为后端提供js工具包](https://km.woa.com/posts/show/520844?kmref=author_post)
* [React语境下前端DDD的思考](https://km.woa.com/posts/show/519558?kmref=author_post)
* [Formast动态表单框架1.0正式版，性能杀疯了!!!](https://km.woa.com/posts/show/516508?kmref=author_post" \o "Formast动态表单框架1.0正式版，性能杀疯了!!!" \t "_blank)
* [前后端统一接口DSL标准](https://km.woa.com/posts/show/505595?kmref=author_post)
* [记一次React性能优化：更小粒度更新](https://km.woa.com/posts/show/502029?kmref=author_post)

Copyright©1998-2021 Tencent Inc. All Rights Reserved

腾讯公司研发管理部 版权所有

[广告申请](https://km.woa.com/chartlets/km/) [反馈问题](javascript:void(0);)

[463/480/499 ms]