电商服务开放平台-DDD设计实战一

初识DDD：理解DDD设计思想以及核心四层架构模型

--楼兰

一、电商服务开放平台实战课程内容介绍

到这里，电商项目就暂告一个段落，接下来，我们将开始一段新的实战之旅。开始构建电商的服务开放平台。

> 前置课程：**【图灵课堂】互联网未来架构之道DDD领域驱动设计** https://vip.tulingxueyuan.cn/detail/p_615ea697e4b0dfaf7faa6cdf/6?product_id=p_615ea697e4b0dfaf7faa6cdf

1、项目需求介绍

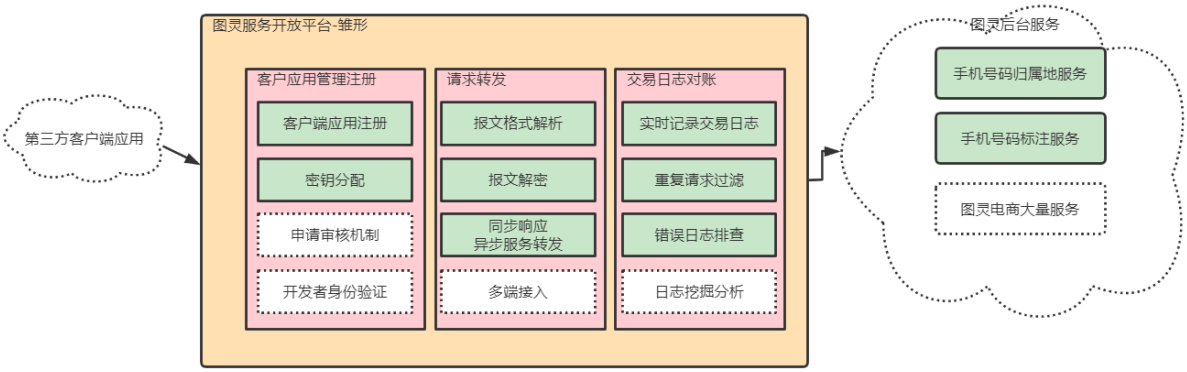
当今互联网讲究互联互通，当企业通过核心产品构建了核心的业务能力后，都希望通过服务开放平台将核心能力暴露出来，通过与其他行业能力互通，打造更完整的产品生态。所以，可以看到，现在基本所有主流的互联网产品，都在构建自己的服务开放平台。比如：

- 微信开放平台： https://open.weixin.qq.com/
- 淘宝开放平台： https://open.taobao.com/
- 支付宝开放平台：https://open.alipay.com/
- 新浪微博开放平台： https://open.weibo.com/
- 百度AI开放平台： https://ai.baidu.com/
- 快递100API开放平台： https://api.kuaidi100.com/notice

而我们的电商项目在构建过程中，也逐渐积累出了很多可以供第三方平台联合调用的业务能力。因此，我们也可以着手构建一个服务开放平台，将电商项目的一些业务接口，对第三方应用开放。

服务开放平台的需求定位是很明确的，就是在未知的第三方应用和内部的电商应用之间构建服务交流的桥梁。对外，降低第三方应用接入电商后端服务的门槛。对内，提供对电商后台服务的安全隔离。

而在技术设计方面，开放平台强调的就是标准。一方面要为第三方应用制定易实现的接入标准。另一方面是要为内部电商应用构建快捷，安全的服务输出标准。



其中，对于与第三方客户端应用之间的接入标准，要保证标准通用性，这就需要降低服务接入门槛。电商项目内部通过基于Nacos的微服务体系提供模块之间的合作能力，如果你想要让互联网第三方应用都能访问电商后台的服务，显然不可能要求互联网的第三方应用都使用微服务架构。而且，Nacos服务本身也不可能向互联网开放。所以，一定是需要基于无状态的HTTP协议来构建，并且需要通过SDK等方式来进一步降低第三方应用的技术要求。让第三方应用只需要关注于业务，而不需要关注于接入的技术。

而对于与后端应用之间的接入标准，则更强调安全性，需要提高服务接入的门槛，毕竟一旦面向互联网开放，面对的将是数不胜数的无数奇葩。所以，对于后端服务，可以设计一些比较复杂的接入标准，比如电商现有的微服务或者要求电商进行一些可以快速实现的改造方案。另外，也要考虑对后端服务进行流量保护，将一些非法的、重复的，或者不合理的请求直接拒绝，防止对后端服务产生大量的无效调用。

2、课程重点-重架构，轻业务！

课程中会带大家实现一个服务开放平台。但是，这毕竟是一个教学课程，在课程开始之前，需要你转换思维方式，将自己从一个学习者转换成为一个思考者。站在项目长期演进的角度，思考如何让项目从起步开始，慢慢发展得更好，而不是像电商项目一样，只是学习已经被别人验证成熟的设计方案。

第一、电商服务开放平台更强调整体设计。

服务开放平台并不提供实际的业务能力，他的设计重点，并不在自身，而是在整个业务流程中。在设计电商服务开放平台时，就不只是需要关注自己本身的功能实现，而应该站在整个服务调用的业务流程中，去考虑外部的第三方应用以及后端的电商服务如何去协作。并且在这个协作过程中，服务开放平台是定义标准的主体，所以需要更关注如何设计建立第三方客户端与企业内部服务的交互方式。相反，对于OAuth、负载均衡、分布式事务、缓存一致性等等这些在其他业务系统最经常考虑的问题，则并不是服务开放平台优先考虑的重点。

这一次，我们并不会按照电商项目的思路，直接按照大厂标准带你去实现一个可以在公网上使用的电商服务平台，甚至很多的功能都会比互联网上那些开放平台简化很多。我也非常鼓励你去对比这些成熟的服务开放平台，去理解我们这次的实战项目有多low，low在什么地方。然后，自己思考如何优雅的把这些功能给补充上。

第二、电商服务平台在设计时，更强调变化。

这个课程的设计，是为了针对一个缺失已久的问题。你应该都已经非常熟悉一个具体的功能点，如何从简单快速的实现逐步往三高架构进行优化的过程。但是如何将这种演进思路从某一个具体的功能点升级到一个整体的项目，这样的开发经历就非常少了，更别说如何将你手中各种各样的Demo系统向互联网的成熟系统一样，长期支持，不断演进下去了。

而这次的课程设计，强调的是变化。会带你做出多个不同版本的设计，并都提供具体的实现版本，带你来真实的理解这些设计思路，从而理解一个整体的项目，应该如何从简单走向复杂，并进而去面向未来无穷无尽的，超出你当前理解范围的需求变化。

比如，你可以想象，之前的电商项目中，开发之初，我们就花费了几十台服务器来搭建各种各样的服务集群。但是，如果我就只需要做一个起步阶段的，规模很小的电商项目，也要一上来就投入这么多服务器吗？电商项目会以Nacos为核心来构建微服务应用，那是不是一上来就先搭建Nacos再说？你有没有想过如何在没有用Nacos之前，对Nacos的可行性做验证，让领导相信Nacos是有用的？再比如，数据库以后想要从MySQL换到Oracle，一次切换肯定换不过来，那如何让项目支持一个模块一个模块的分步切换呢？

另外，在本次课程中，并不是就针对服务开放平台这一个特殊的场景，做一通天花乱坠，花里胡哨，炫技式的设计，而是会按照一套成熟统一的架构理论来进行指导，落地。这套理论方案就是DDD领域驱动设计。毕竟，你要清楚，越是复杂的设计，通常他的适用场景就越是单一，对你的实际开发帮助，就越是有限。例如，针对高并发，基于Redis，你或许了解过好多种优化的实现思路，什么分布式锁啊，防止超卖啊，缓存一致性啊各种方案。但是，如果你的实际项目中没有用Redis呢？是不是Redis的这些解决方案就白学了？而DDD的好处在于，他是一套普适性的架构思想，上到整个企业的软件体系设计，下到每一个功能如何实现，都能提供统一的指导思路。换一种技术实现，也依然可以借鉴其中的解决思路。并且，DDD其实是业内多年实践积累下来的一种思想，你只要能理解DDD，并将DDD良好的落地到项目当中，项目的健壮性就能自然而然的得到很大的提高。这些会在后面的课程中，逐步带你去理解。

二、电商服务开放平台功能演示

1、演示实现效果

> tulingmall-open 功能演示

目前tulingmall-open主要完成了两个主要的流程：

流程1： 客户端应用注册流程

tulingmall-open对外开放服务，肯定不能让所有的人随意访问，所以需要对访问的来源进行鉴权，只有少部分的合法请求才能访问。所以，需要app需要先在tulingmall-open上完成注册，获取到专属的访问凭证。后续需要携带访问凭证才能访问到服务。在tulingmall-open的实现中，是直接在后台完成app的注册。注册时，由tlmall-opem会生成一对专属于app的公私钥，其中，私钥由tulingmall-open自己保存，而公钥则主动分发给app，app需要自行保存注册分配的sysid以及这个公钥，共同作为访问的凭证。

吐槽时间：跟互联网的服务开放平台接入流程差在哪？以后如何优化？这里列出几个很明显的。

一、这个注册的系统ID，未来将作为授权认证的唯一要素，没有过期保护机制。

整个授权机制，虽然没有复杂的审批流程，但是整个业务的核心是充足的。因为在这些服务开放平台的应用API层面，最终都是通过一个类似于token的私密数据来进行授权判断的。只不过，互联网平台会通过申请审批机制、OAuth2.0机制等复杂机制来保证这个token的安全性。而tulingmall-open则是直接用sysid作为授权标志，省掉的是安全流程，保留的是最核心的授权认证机制。

另外，在tulingmall-open中，预留了系统用户名、系统用户密码、系统IP白名单，这几个字段暂时没有用。其作用也是表明未来tulingmall-open需要接入更多的鉴权机制。

> 如果你看不到以后的变化路线，可以回头参看一下OAuth2.0的扩展课程，这基本是服务开放平台的标配。

二、只生成一对公私钥

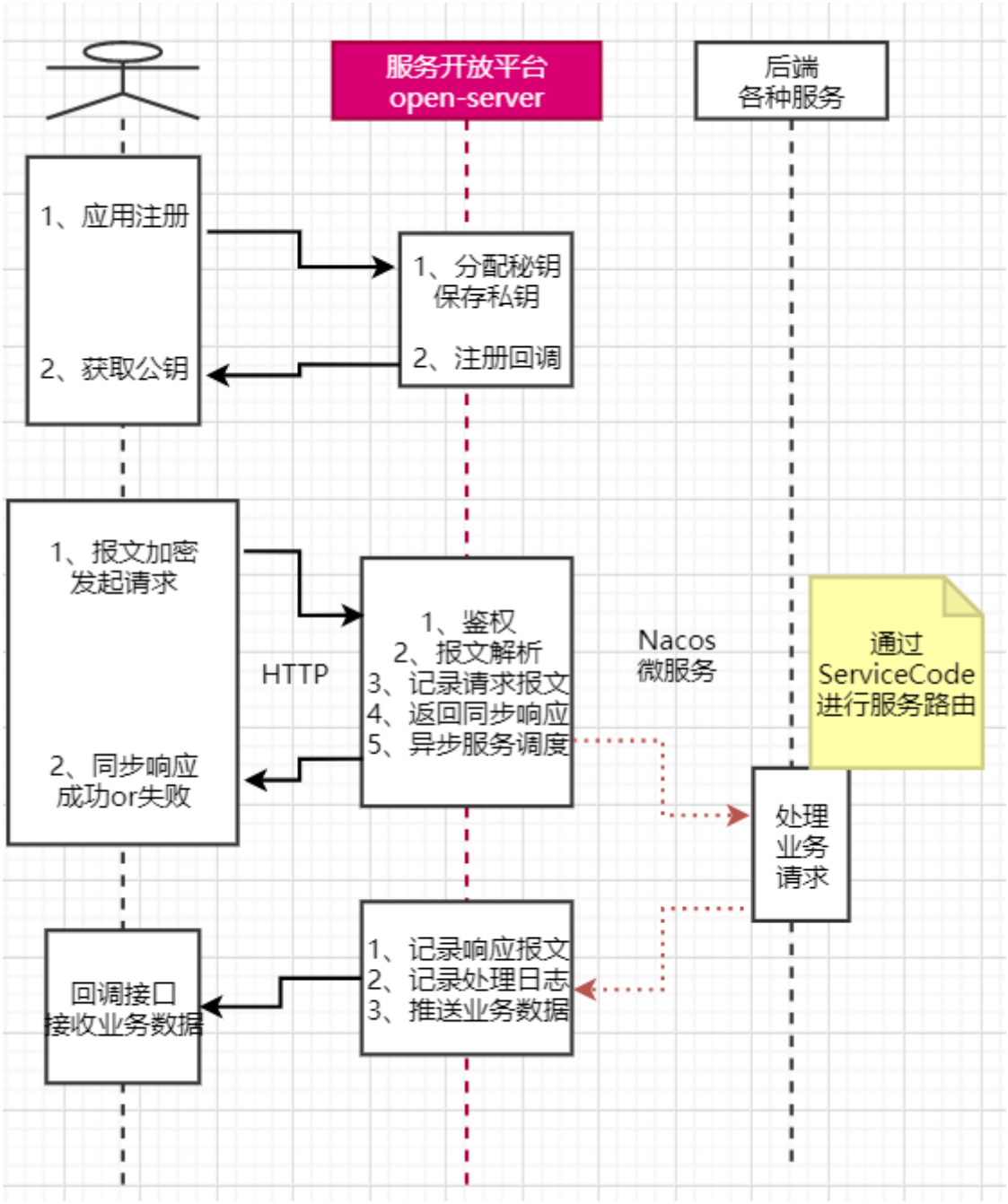
至于密钥，跟其他平台的机制是差不多的，都是使用对报文使用公钥加密后，必须使用对应的私钥才能解密出正确的报文。只不过，像微信开放平台和支付宝开放平台，会要进行双向加密。即app进行注册时，平台会生成一对公私钥，将公钥通过开放平台分发给app。同时，平台会要求app也生成一组公私钥(平台提供生成工具)，私钥由app自己保留，公钥上传个平台。其中，平台生成的公私钥是用来对app提交给平台的请求进行加密，而app自己生成的公私钥则是用来对平台返回给app的业务数据进行加密。

至于消息回调地址，基本上跟其他平台是一样的。在你接入微信开放平台、淘宝开放平台时，也需要注册回调地址。

> 这些差距，是课程完成后需要考虑的问题。
>
> 你还有没有其他的想法？多思考，多吐槽！

流程2：客户端应用请求转发流程

App完成注册后，就可以下载对应的客户端jar包以及示例代码去完成调用了。这一块是模拟的其他服务开放平台的实现流程。tulingmall-open会提供统一的接入方式，然后通过API文档指导App传入正确的业务参数。 其中serviceCode主要表示要调用哪个服务，然后在requestBody中要传入服务对应的参数。



吐槽时间： 这一块与服务平台的差距就没有那么明显了。并且其实各个开放平台的实现方式也是不一样的。

对于数据安全性，tulingmall-open也是与其他开放平台一样，采用同步响应+异步回调的机制。同步响应请求是否接收成功，而真实的业务数据，tulingmall-open会异步往App注册的回调接口进行推送。这样，即使App的sysId和公钥泄漏了，被别有用心的黑客冒用了，黑客也无法拿到业务数据。

然后为了对后端服务进行流量保护，tulingmall-open在实现时，需要App自己传入一个transId，tulingmall-open会根据transId和serviceCode来判断请求的唯一性。如果发现App发送了重复的请求，tulingmall-open就不会再去重复请求后端服务，而是会直接给App推送历史的请求结果。

tulingmall-open与后端服务之间，目前是通过基于Nacos的openFeign接口进行调用的。这样，虽然目前只接了两个简单的服务，但是之前电商项目的所有业务接口，都是可以作为后端服务接入到tulingmall-open的。

> 还有哪些不足的地方？项目后续应该如何改善？请自己补充！

2、回顾设计思路

tulingmall-open是一个典型的MVC的设计思想。在做这种项目时，通常的设计思路都是这样的：

1、优先设计表结构。

通过UML图，时序图这一类设计工具梳理业务流程，将关键业务要素都沉淀到底层的表结构中。设计出足够支撑业务的表结构。然后将表结构映射成项目中的POJO对象。

tulingmall-open中设计了clientInfo表，记录了App的关键信息。hisRequest表记录了App的历史访问日志。而后端服务信息因为是直接从Nacos上获取的，所以不需要建表记录。

2、构建Controller开始与业务场景进行对接。

在设计Controller时，往往会很自然的按照一些业务拆分形成不同的Controller 。例如tulingmall-open中很自然的就会设计HisRequestController负责处理历史日志相关的功能，SysManageController负责处理App管理相关的功能，TransferController负责处理请求转发相关的功能。

3、在Controller的实现过程中，配置需要的Service，DAO实现。

Serivce和DAO的设计，通常就比较随意了。就是需要什么就用什么。

这应该是一种你很熟悉的设计方式。后续就会称为MVC设计方式，用来与DDD设计方式进行对比。

3、总结发现不足

按照上一节课的分析，如果tulingmall-open的需求就这样了，以后没有什么变动了，那设计出这样的实现也就够了，好像看不出太多的根本性的问题。但是，如果考虑到tulingmall-open以后需要往互联网上那些服务开放平台进行演进，问题就会显得有点严重。比如，下面简单列出几个可以预见的问题：

1、鉴权机制不足，后续需要对App进行更复杂的管理。

这就需要对ClientInfo表做大量的修改。但是，一旦修改表结构，就需要修改对应的POJO，Mapper，还有Service等一系列的代码。如果项目足够复杂，这个改动会相当于重构。

在tulingmall-open的设计中，为了防止表结构，也做了一些挣扎。例如，在ClientInfo表中预留了几个有明确含义的authedIp,userName,pwd几个字段，同时还预留了业务含义不明确的reserve1,reserve2两个字段。其目的，就是希望未来引入更多的鉴权机制时，能够尽量少的改动表结构。

但是，你应该能够理解这种挣扎有多无力。因为未来的鉴权机制是tulingmall-open当前没有考虑到的。ClientInfo表中预留的任何一个字段，都只能应对一种可能的鉴权机制，如果未来要上OAuth2.0或者另外一些当前想不到的鉴权机制时，这种设计就会显得很无力。这种行为就会被称为"**过度设计**"。

2、后端服务接入方式过于笨重。

每接入一个新的服务，都需要在tulingmall-open-server项目中配置对应的接口参数，对应的线程实现类。当前接入两个后端服务，感觉还不是很麻烦，但是如果后续要像互联网开放平台一样接入几十上百个服务，工作量就会相当庞大，并且接入的质量也会难以保证。

3、难以实现服务降级处理。

当前设计了三个Controller响应三种主要的业务，但是这些业务的负载压力明显是不平均的。TransferController所要负责的请求转发项目的压力会明显高于另外两个Controller。如果并发量不大，系统的性能充足，还没有什么问题。但是当并发量开始变大，系统的性能紧张时，就希望能够将更多的系统性能提供给TransferController。

这个问题，虽然可以通过引入Nginx，将tulingmall-open进行分布式部署，但是你自然能看到，这并不能解决根本上的问题。加多少服务器合适呢？各个服务器上分配的请求就是不平均的，要怎么办呢？

解决的办法有很多，比如使用Dubbo做降级处理，比如自己实现负载均衡策略等等。但是，在这众多复杂的解决方案中，你可能直接就忽略了一种最简单，也最自然的解决方案，就是将这三个服务进行微服务拆分！如果可以将这三个功能模块抽象成若干个微服务(并不一定要是三个)，那么所谓的降级、负载均衡，还不是自然而然的事情了。把请求转发服务多部署几个实例，其他服务少部署几个实例就可以了。这是不是比那些复杂的策略简单可行很多？

但是，当你真的想要做去拆分时，基于tulingmall-open，你会发现事情并没有那么简单。Controller还很好拆分，service呢？mapper呢？表呢？而在复杂项目中，甚至很多不同的业务还会使用同一个表。微服务拆分的过程就足够让你头大，想要不影响业务进度，完成这样的微服务拆分，基本是不可能的事情。

三、DDD如何应对软件核心复杂性?

总结以上几个问题，会发现这几个问题有一个很明显的共同点。就是这些问题如果提前设计，都不难实现。但是就是因为没有做提前设计，就只能将这些设计上的变革延后到项目建设过程中。而此时，项目带上了沉重的业务包袱，就会让架构调整越来越困难。直到最后，这些业务包袱越来越沉重时，很多项目组就不得不走入重构的深渊。

表面上看，这似乎是技术架构的问题。如果架构师经验足够丰富，就能提前考虑到这些问题。如果开发团队技术足够过硬，就可以最快速度坚决这些问题。但是，如果你将视野放开，放到到整个项目团队。你就会发现，一味的要求技术人员能够未卜先知，这是不现实也是不理智的。尤其对于长期建设的大型项目，更是看如此。难道你要求当年开发淘宝时就要支持好双十一吗？另外，在Eric Evans提出DDD真个思想时，在书本的前言部分，就分享了对他影响很大的三个典型项目。这些项目，是无数软件团队面临问题的缩影。

> 有兴趣去看一下《领域驱动设计-软件核心复杂性应对之道》一书的前言部分。

所以，这一类问题，其实并不是个例，而是软件发展过程中，大家一直都在面临的通用问题。而这些问题，随着微服务架构的不断完善，开始变得越来越严重，也被越来越多的人所重视。其实很多软件团队都在尝试解决这样的问题，你可以看到，业界不断推出像敏捷开发这类的软件工程管理方式，也不断诞生Jira、Jenkins等软件实施工具。这些方式都有助于缓解这些软件的核心复杂性问题。

DDD也是随着这些核心问题逐渐诞生的一种理论体系。特别之处在于，他尝试以一种通用的、统一的理论指导，来应对软件面临的这些核心复杂性问题。以往的这些解决方案，更多的局限于某一部分架构师、某一些具体项目。这些经验很难形成团队的共识。而DDD希望让整个软件团队都用统一的方式思考解决问题，减少架构师、程序员、产品经理、测试人员等各个参与角色之间的分歧。上到企业战略设计，下到每一个功能实现，都能从DDD中获取到足够的指导。

那么DDD是如何应对这些软件核心复杂性呢？固然DDD有非常多的理论工具，这些在后面会带大家继续回顾。但是，更重要的是，DDD有两个非常核心的思想，很容易对大家习惯的软件开发方式形成冲击。

技术主动理解业务。

技术虽然是为了解决业务问题而生，但是，技术却往往并不能忠实的反应业务现状。技术人员总是习惯性的按照Controller、Service等等类似的技术概念来组织整个项目。但是这些概念并不具备实际的业务价值。因此，业务专家设计出来的业务流程，总是需要产品人员进行梳理，及时雨人员设计才能最终落地到项目当中。而这其中，不可避免的会产生信息不对称的情况。尤其在面对频繁变化的复杂业务时，这种问题会更加明显。

以演示项目中的服务降级问题为例，业务专家看到的是消息转发、历史请求管理、应用注册管理这三个业务能力，不管多复杂的业务场景，基于这三个能力进行组织就行。但是在技术人员的视野中，却显然更关注MVC的三层架构。会更注重Controller、Service这样一个个技术层面的逻辑分工与复用。当业务需求发生变化后，业务专家考虑的是如何通过这三种业务能力将整个业务场景贯穿起来。而技术专家往往会更多的考虑新的结构对以往的架构有多大的冲突。如果新的设计对已有的加油造成太大的冲突，就只能考虑重构。而重构，对于业务专家来说，其实是没有什么业务价值的。

要如何协调这种业务与技术的冲突呢？其实业界也有很多的方法。比如像微软的ERP这类成熟的系统，往往会不断推动行业的一体化解决方案。从企业组织架构、管理方式等等各个方面提供一整套的服务，软件只是整个服务体系中的一个环节。其目的，其实也是为了让软件能够更忠实的反应业务场景。只不过，这种方式在国内，略显霸道，对软甲的成熟度要求太高。往往会有点水土不服。

而DDD则是主动强调让技术贴近业务。主动放下技术层面的身段，让软件项目，按照业务方式来构建。业务如何运行，软件就如何构建。实际的业务实施过程中需要哪些角色，软件就构建对应的组件。这些角色之间需要怎样的联系，软件中就构建对应的关联关系。当然，这并不意味着软件可以随着业务场景随心所欲的设计。而是为了让不怎么懂技术的业务专家也能主动参与到软件建设的整个过程当中，而不仅仅是前期的需求设计。

例如DDD鼓励形成团队内部的通用语言，来让领域专家和技术人员能够共同理解软件项目。这种通用语言即不需要过于接地气，也不需要过于学术，只要让团队内部形成统一接口。而DDD中所谓的领域，就是试图建立一种技术人员和领域专家(业务专家)都能够共同理解的业务模型。业务如何组织，软件就如何构建。让领域专家也能参与到软件建设过程当中。然后，DDD会不断将领域划分成更细分的子域，并形成限界上下文等概念来强调领域划分方式。这些想法，其实也是和领域专家将复杂问题逐步拆分、细化的思路是一致的。

在DDD的实践过程当中，有很多技术人员容易陷入以往的技术思维中。将DDD理解成某种技术架构。觉得只要调整一下软件架构，将DDD中的各种基础概念落地实现，就是实现了DDD。比如说，DDD中设计了Repository仓库组件，负责实体的数据持久化与查询工作。而我见过很多人，将MVC中的DAO转换成为OracleRepository，MySQLRepository等等这样的仓库实现。这些组件描述的依然是软件项目的技术实现，而不是业务实现。领域专家看到项目中都是一些这样的实现，绝大概率是一脸懵逼，从而放弃了继续参与软件建设的信心。而相反，OrderRepository，ClientRepository这一类的实现，体现的是更多的业务属性。领域专家看到这样的组件，即便不了解实现细节，也能够知道这些组件是干什么的，从而有可能继续深入的参与软件建设过程。你可以想象，如果在这种氛围中开发软件，即便真的需要重构，也比较容易得到领域专家的理解。

> 反例： https://gitee.com/xtoon/xtoon-boot 不是四层架构就是DDD。
>
> https://gitee.com/izhengyin/ddd-message/tree/master
>
> 反例： https://gitee.com/hypier/barry-ddd 限界上下文落地不够明确，无法强化领域划分

技术主动去理解业务，这是DDD相对于传统MVC最为明显的区别。当然，要想让技术真正能够理解业务，光靠调整软件的包结构肯定是不够的，还需要一系列从战术落地到战略设计的完整理论体系来进行完善。而这，就是接下来我们需要了解的DDD领域驱动设计。

“刚刚好”解决问题。

在以往的设计过程中，为了更多的兼容日后的需求变化，技术人员往往会做很多的“提前设计”。寄希望于这些“提前设计”能够减少未来业务变化对软件架构的冲击。但是，往往项目未来的需求变化并不是技术人员能够把控的。如果未来的业务需求不是按照之前设计的路线演进，这些“提前设计”不但无法起到设计时的作用，反而会增加软件项目的复杂度，从而给软件后续的升级改造带来更多的负担。

> 这方面最明显的例子就是注解。

以演示项目当中的鉴权问题为例。在对gsmanage表进行具体设计时，由于考虑到未来鉴权机制可能出现一些变动，因此，我们会主动在表结构中增加reserve1,reserve2这样的字段，希望这些字段未来可以用来兼容更复杂的鉴权因素。希望未来在与实际企业项目集成的过程当中，除了已有的IP、用户名等鉴权维度外，还可以更轻松的兼容部门、角色等等这样的鉴权维度。但是很可惜，我们分析出了OAuth2.0这样的需求，并没有按照之前的套路出牌。如果真要接入OAuth2.0鉴权体系，这些不用的业务字段，不但不能帮助解决新的需求，反而给权限系统改造增加不必要的复杂度。

而DDD强调的则是让软件可以"刚刚好"的解决眼前的业务问题，同时，通过一系列的手段，为软件保持足够的灵活性。例如同样是以DDD中的Repository仓库组件为例。在DDD设计中，我们会将实体的持久化逻辑全部交由仓库组件负责，在业务流程中不需要考虑持久化相关的逻辑。当我们需要设计更复杂的持久化策略时，就只需要调整仓库层的实现，而不需要考虑业务上有什么影响。比如，往往很多项目在最初的设计中，都会选择将关键业务信息通过MyBatis这样的框架直接存入到数据库。但是如果软件的并发量逐渐增大，往往我们就需要增加一个Redis缓存层，用来保护数据库，提高系统的并发能力。而围绕Redis缓存，各种缓存穿透、缓存击穿、缓存雪崩、缓存预热等等这样的一些业务逻辑就自然而然的冒出来了。这样一个简单的MyBatis就不够用了。如果没有抽象出Repository仓库层，那么这些新增加的逻辑就不可避免的要蔓延到上层的DAO或者Service当中。但是，如果抽象出了Repository仓库层，不管多复杂的持久化逻辑，都只是一个仓库实现而已，不会对业务造成任何影响。这些需求就可以比较独立的进行改造。

关于DDD如何保持软件的灵活性，后面课程中会有大量的内容进行具体的演示。但是在对DDD进行落地实践之前，仔细梳理一下DDD的这两条核心思想，是非常有必要的。

总结

通过这一章节，你或许会意识到，这将是你从未接触过的一个项目实战课。对于这个电商服务开放平台项目，我们并不试图带你去理解各种具体的业务设计，而是重新开始审视软件项目本身。写好的代码，做好的设计，这其实并不是架构师的专属。我们写过的每一个CRUD功能，每一个基础的信息管理系统，其实都是程序员技术成长的营养。所以对于这个电商服务开放平台，我们甚至会故意将很多具体的业务实现进行简化，只保留最核心的主干。逐渐尝试以动态的眼光来看软件项目，来审视自己的技术成长。

这一章节内容，也在带你去发现电商服务开放平台的问题。其实发现问题的能力也是程序员非常重要的一种技能。发现的问题越深，意味着你理解技术也越深。所以，课上带你分析的这些问题，不是终点，相反，是个起点。加入你自己的思考，列出你自己的问题，然后再逐步尝试形成你自己的解决方案。

在章节最后，简单介绍了DDD领域驱动设计的基础思想。你看到，基础的思想其实很朴素。当然，一百个人眼中有一百个哈姆雷特。对于DDD，这里也只是分享我的理解。接下来，我们就会尝试用DDD来对电商项目进行逐步改造升级，希望从这个过程中，你能够形成你自己的DDD设计思想。

> 有道云分享连接：https://note.youdao.com/s/Q8FQPJDk