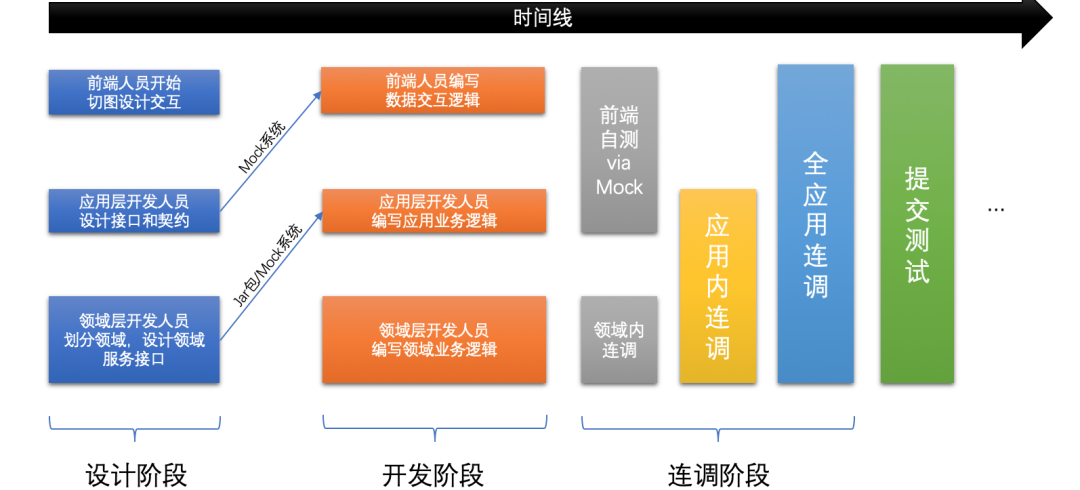
**浅谈微服务体系中的分层设计和领域划分**

汤波/甘盘 分布式实验室  1周前

K8s已经成为一线大厂分布式平台的标配技术。你是不是还在惆怅怎么掌握它？来这里，大型互联网公司一线工程师亲授，不来虚的，直接上手实战，3天时间带你搭建K8s平台，快速学会K8s，点击下方图片可了解培训详情。

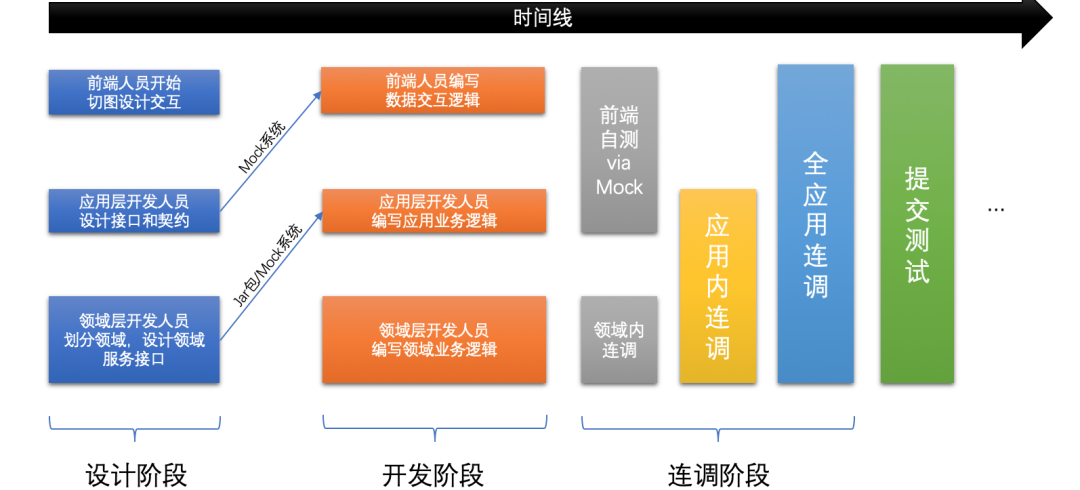
[](https://mp.weixin.qq.com/s?__biz=MzA5OTAyNzQ2OA==&mid=2649709888&idx=2&sn=712f5ce411434f5112790d581d389f10&scene=21#wechat_redirect)

引言



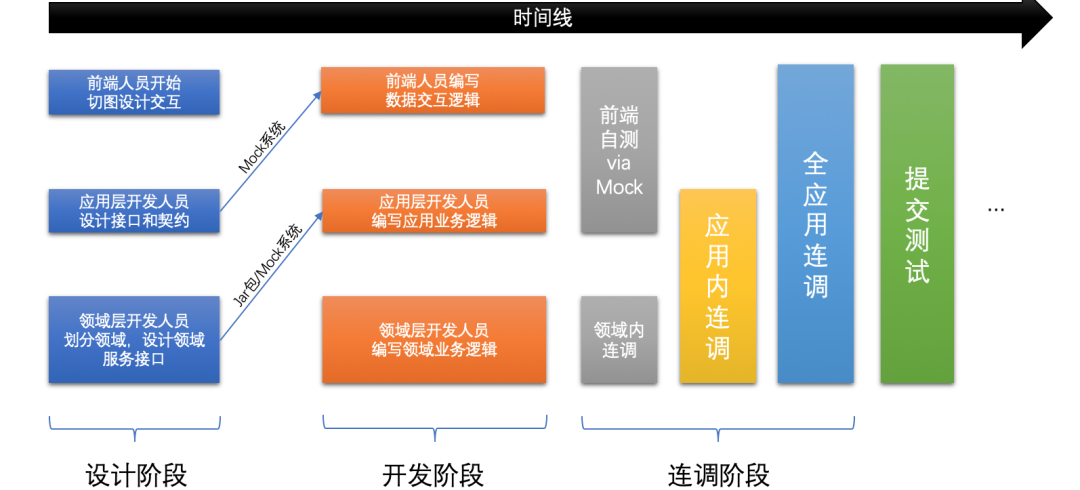
***本文获得阿里巴巴《第二届研发效能征文》优秀文章奖，并在阿里第二届能效峰会上展出。***  
看标题感觉这个东西很理论，比起“高并发、多线程”、“分布式CAP、一致性、Paxos”、“高可用SLA”等具体的干货技术点，软件体系知识显得很“湿”，似乎人人都有自己的认识，但又很少有人能说完整，有一点可以确定的是，如果你未来需要独立设计一个复杂的系统中台，并使之未来能快速应对各种需求变化的话，科学合理的领域划分和边界界定需要我们“处女座级”的坚持下去，这对防止人力失控、减少项目烂尾很有帮助。合理的界定了边界后，即便某个微服务很糟糕，也可以就输入输出以很少的人力投入进行重构，相反的就是牵一发而动全身，加上业务需求频繁而来，很容易烂尾或是达不到如期的效果。  
其实很多技术大神都是某一个技术点的好手，但可能在整体软件体系上思考并不多，每个人都有自己的设计方法，大部分容易想到的设计方法处理一般的系统已经够了，后面发生问题慢慢打补丁就行了，当我们面对各种需求变化陷入开发困境的时候我们就该想想了，咱们系统的体系设计上是否出了问题？本文不打算涉及领域建模和设计模式等代码级别的详述，而是探讨如何将一个复杂的大系统进行分层和拆分，这是设计一个优美系统的第一步，相信对各BU同事们快速搭建系统中台也是很有参考意义的。  
文中的一些例子大家也可能遇到过，大家如果在开发中遇到困境，可以多来圈子交流和发表问题，大家一起学习进步。大概知道内容背景的可以直接跳到第3部分。想了解一个大项目如何进行科学人员安排的可以直接看5.4部分。如果你的组里还有人把数据库模型当接口契约用，可以建议他看下5.1部分。假如你在开发过程中遇到一些别人的开发设计习惯，你觉得不是很好，但是又不知道如何说服他，都可以到评论区聊聊，大家一起讨论讨论。

1. 摘要



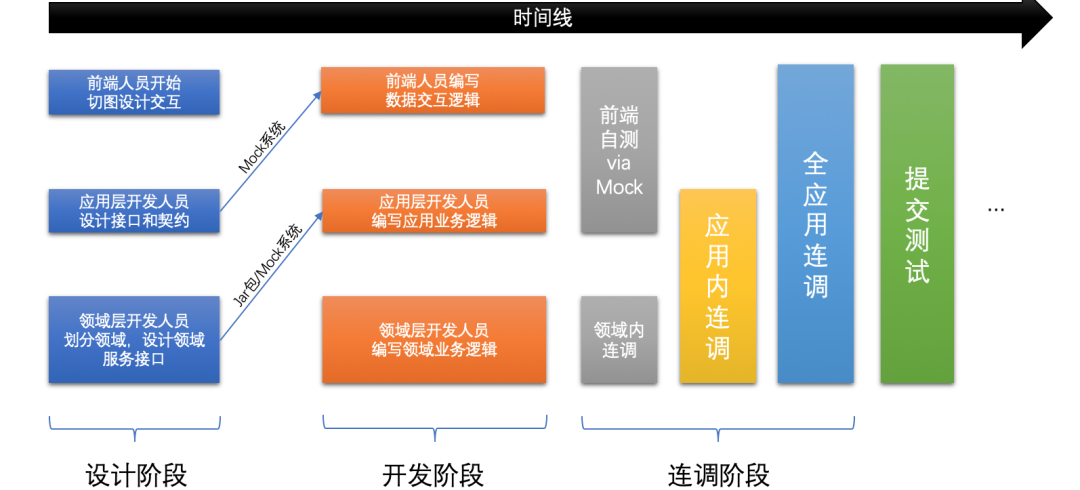
本文阐述了一种将分层设计和DDD领域设计思想应用于微服务体系架构的方案实践，也是个人的最佳实践。对于大部分互联网公司来说，我们主张将其Web服务架构分为五层：基础设施层、领域服务层、应用服务层、网关层和用户界面层（表示层）。  
领域服务层和应用服务层均可以采用微服务设计进行拆分，其中领域服务层将按照DDD领域设计进行领域划分，设计为一个个领域模块微服务，每个微服务高度内聚，仅关注自己的业务，领域服务间通过接口调用进行松耦合。这种设计方案可以大大简化大系统，并且在后期的维护中优势会日渐凸显，然而把大系统分而治之拆成微服务同时也对架构师和开发人员提出了更高的要求。第2部分介绍了相关背景，接着第3部分探讨了分层设计以及每一层的功能，第4部分结合微服务和DDD对领域服务层进行服务模块划分和设计。第5部分则就分层设计和DDD领域设计中常见的问题进行了整理。

2. 背景介绍

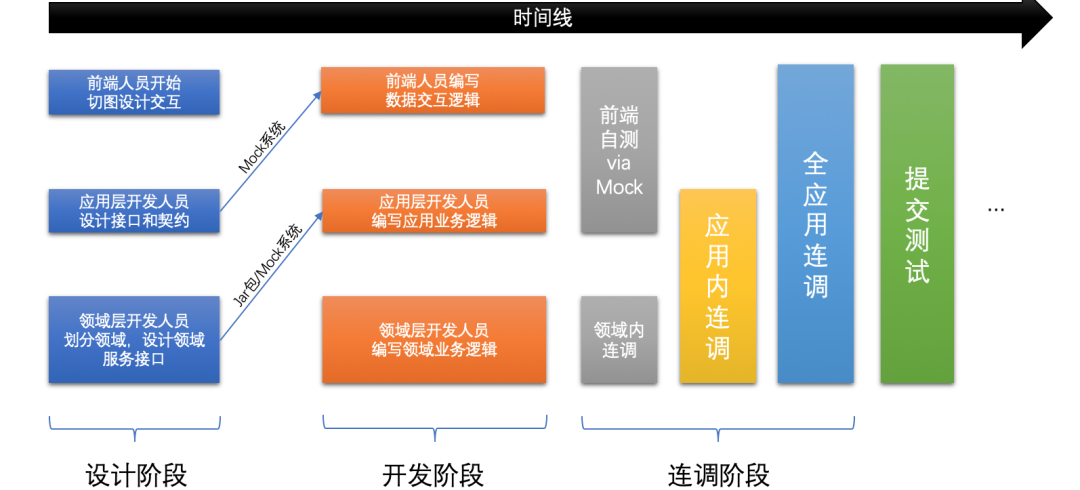


想写这样一篇文章很久了，虽然本科学的是软件工程，但碍于自己能力有限，从08年写代码以来一直断断续续的思考，始终对项目模块设计和分层结构设计没有一个可以让自己觉得满意且无纠结点的答案，假设了某个设计，很快在实践中又会发现其存在着一些问题。直到2014年毕业工作了解了DDD领域驱动设计后，才有了相对清晰的方向。实际上早在2004年，Eric Envas的《领域驱动设计：软件核心复杂性应对之道》就已出版，毕竟软件开发自计算机普及以来已经存在很长一段时间了，早期国外程序员对软件开发理论的研究也十分兴盛，如今成熟后反而研究的相对少了，基本上依葫芦画瓢即可。  
DDD领域驱动设计对软件设计各个环节的人员都有较高的要求，用《领域驱动设计》一书的话来说它需要一个“领域驱动团队”，它要求从分析阶段，产品经理、项目经理、架构师以及开发工程师就使用统一的模型语言（Ubiquitous Language）来进行沟通，并且他们都懂一些代码、产品和建模相关的知识，事实上这在国内很难实施，国内的产品经理约等于需求整理工，对其计算机基础的要求是少之又少，在我所从事的公司里，也曾发生过产品经理直接指导开发，以至于后面双方理解的同一个词有着不同含义的情况。  
所以本文不打算去阐述DDD领域内部建模代码级别的实践，甚至本文并不认为贫血模型是不好的，本文主要探讨领域之间的划分和分层设计，正如引言说提到的，这是设计优美系统的第一步。另外提一句：其实合理设计的微服务体系中的服务本身就是功能单一边界清晰的小应用，届时贫血也好、DDD领域建模也好，其实都可以胜任。  
近年来，随着分布式的发展，传统中小型机集中式服务器已经不在流行，所以微服务体系也成为了各大互联网公司主流的选择。直观的感受下微服务和DDD两者，似乎一个是微系统，另一个则是大系统的设计方法，似乎两者天生互斥，微服务化的小系统也用不着DDD，其实并不是，DDD是针对整个复杂的软件解决方案的一种科学设计方法，微服务化也是把复杂的大系统拆分为小系统，方便维护和管理，所以两者都有一个特点——为复杂的大系统服务。下面咱们就来探讨下，如何把DDD的领域设计和其主张的分层设计应用到微服务体系架构中。需要说明的是本文主要是个人多年来的一点总结，未必适合所有场景，有更好通用性更为广泛的方案请不吝赐教。

3. 分层设计



准确的说分层设计（Layered Architecture）跟DDD没有必然的联系，我最早接触分层设计是在携程网，当时内部使用的应该只是简单的业务层（Biz）和表示层，数据库访问之类的也是放在各自的业务包下的。后来接触和学习了《领域驱动设计：软件核心复杂性应对之道》，书的第4章“分离领域”中说到了四层分层设计，即：基础设施层、领域层、应用层和用户界面层（表示层）。DDD产生的年代微服务还未流行，当时甚至基于浏览器的Web应用都比较少，更多的是PC软件和EJB等网络应用，所以作者更多的是想表达对复杂系统的逻辑分层，并不在意每个领域是单独的系统还是一个软件系统内不同的模块。  
所以为了跟其做区分，我们建议的四层为在其基础上引入“服务”两个字，即：基础设施层、领域服务层、应用服务层和用户界面层。这样做的意图是让开发人员立刻可以了解到——每个领域模块即一个微服务（一个领域可以对应一个或者多个模块Module）。  
摘要中提到我们主张的分层体系中还有一个层，即网关层，这又是什么鬼呢。刚刚提到的DDD的时代背景，PC软件系统或者企业内部使用的网络应用系统是根本没有网关层（有也是网络网关设备）这一说的，而现如今互联网公司产品的输出形式无外乎Web应用（网站、或者网络服务），并且为了更好的适配PC站和App，一般会采用前后端分离的应用设计方案，这时候会产生一个需求——内部网络应用系统如何把自己的服务输出到互联网上，供外部系统或者浏览器网页访问。  
最直接的方式就是把应用层直接暴露在公网上，但我们不建议这么做，应用层服务更多的是关注业务应用，对网络级的系统安全性（防DDOS、钓鱼、跨域等）、请求监控等缺乏考虑，这些工作交给网关层统一管理会轻松很多（比如淘宝的TOP平台）。  
这时候我们在Web应用系统中引入网关层用于衔接表示层和应用层 ，因为这样可以更好的划分各层的职能。网关层也可以看作是应用服务层的对外包装层。如果一定要把网关层做到应用服务层里理论上也是可行的，比如针对于Spring Cloud这种框架下的微服务体系，可以考虑直接暴露应用层，只需辅助一些运维手段进行统一的安全验证和监控即可。假设我们选择引入网关层，那么我们就得到了以下网络应用系统分层体系：



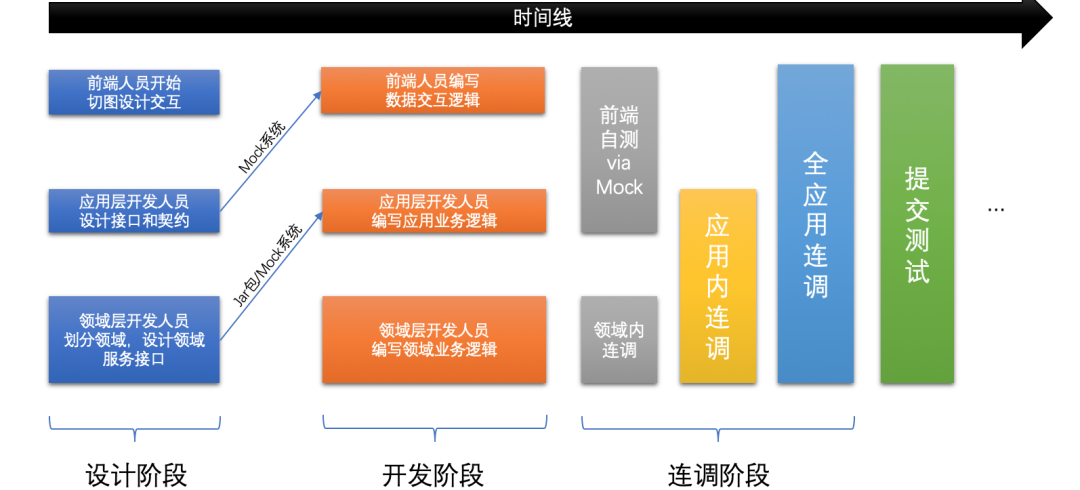
其中，各层的职能和作用为：

* 用户界面层：负责向用户显示和解释用户指令。这里指的用户可以是另一个计算机系统，不一定是使用用户界面的人（比如外部应用调用对应接口）。
* 网关层：负责提供对外的HTTP服务或者其他应用层协议（这里是指OSI七层协议中的应用层，别混淆了哈）服务。
* 应用服务层：定义软件要完成的任务，并且指挥表达领域概念的对象来解决问题。这一层所负责的工作对业务来说意义重大，也是与其他系统的应用层进行交互的必要渠道。应用层要尽量简单，不包含业务规则或者知识，而只为下一层中的领域对象协调任务，分配工作，使他们互相协作。它没有反应业务情况的状态，但是却可以具有另外一种状态，为用户或者程序显示某个任务的进度。
* 领域服务层：负责表达业务概念，业务状态信息以及业务规则。尽管保存业务状态的技术细节是由基础设施层实现的，但是反应业务情况的状态是由本层控制并且使用的。领域层是业务软件的核心。
* 基础设施层：为上面各层提供通用的技术能力，为应用层传递消息，为领域层提供持久化机制，为用户界面层绘制屏幕组件（PS：这个在互联网应用中几乎用不到）等等。互联网Web应用系统中基础设施包含了数据持久化服务，中间件服务（数据库，Redis，Memcached，ZooKeeper，ELK等等）以及第三方服务等。

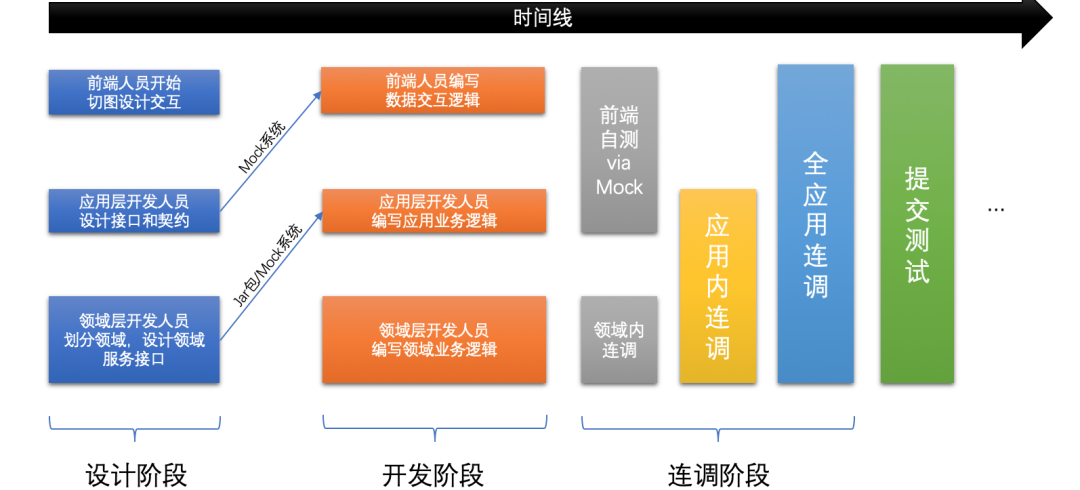
各层除了实现自己的功能外，还需要遵守以下原则：

1. 每一层设计保持内聚，并且只依赖于它的下方的层。
2. 下层向上层发起的通信只能通过中间件等间接方式进行。
3. 上层和下层只能有松散耦合（各自为独立个体，通过简单引用关联）。在某些微服务框架比如Dubbo中，可以把api包提供给上层引用即可。而Spring Cloud的上下层耦合更为松散，通过契约约定即可。前者的优点是调用者可以直接使用提供方定义好的契约和方法。后者的优点则在于最大限度的降低了耦合，避免在上层无限制的进行下层包引入。

这里重点说明应用服务层和领域服务层之间的关系。举一个我经常跟部门其他开发举的一个例子：有一家上市企业A公司，靠卖水果发家，其首席架构师科学合理的按照DDD搭建了一套基于微服务体系的卖水果应用，其架构图如下：

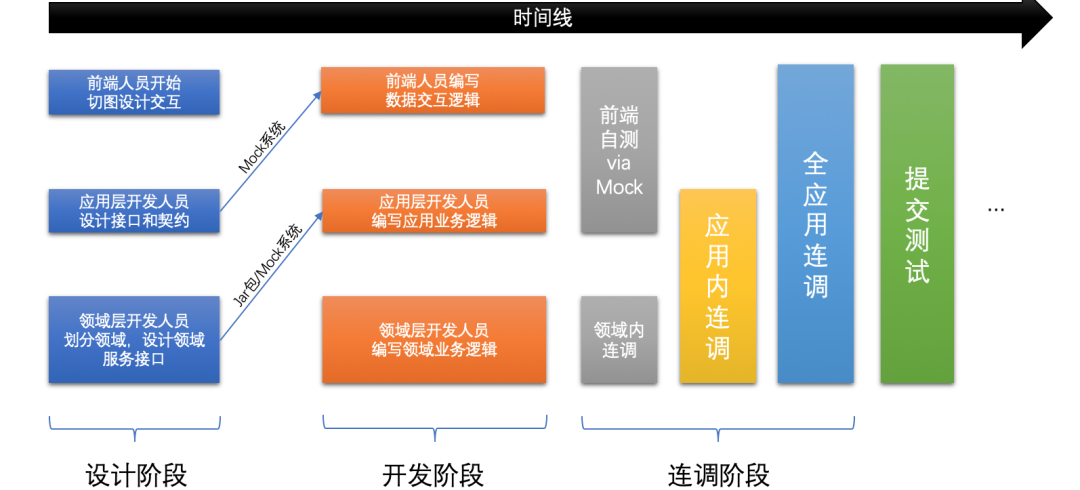


今年水果行情一般，而房地产十分火热，A公司高层发现房地产带动的五金行业也十分火热，于是下达任务给技术部，要求其立即着手搭建五金销售系统，货源已经谈好。得益于首席架构师之前优秀的架构设计，他发现只需要做一个卖五金的网站以及另外对微服务进行微量的调整即可满足老板的需求——因为卖五金和卖水果并无本质区别，他们涉及的环节几乎一致。加入五金售卖的系统架构图如下：



可见应用服务层代表是某一个业务应用，它代表的更多的是从需求出发的应用定义，而领域服务层则是业务领域按照自身的边界进行设计的一个高内聚的服务体。应用层通过协调和组合各个领域服务即可形成一个新的应用服务。《领域驱动设计》中明确指出，在设计领域服务时无需考虑表示层和持久层服务的东西。我在现实开发中总是遇到大量工程师按照产品的设计稿一溜烟的从上至下设计应用层服务和领域层服务，完全没有考虑业务领域的概念，导致后面微服务数量膨胀，功能重复度高。这种开发习惯代表的是《领域驱动设计》作者极力吐槽的一种模式——SMART UI “反模式”。

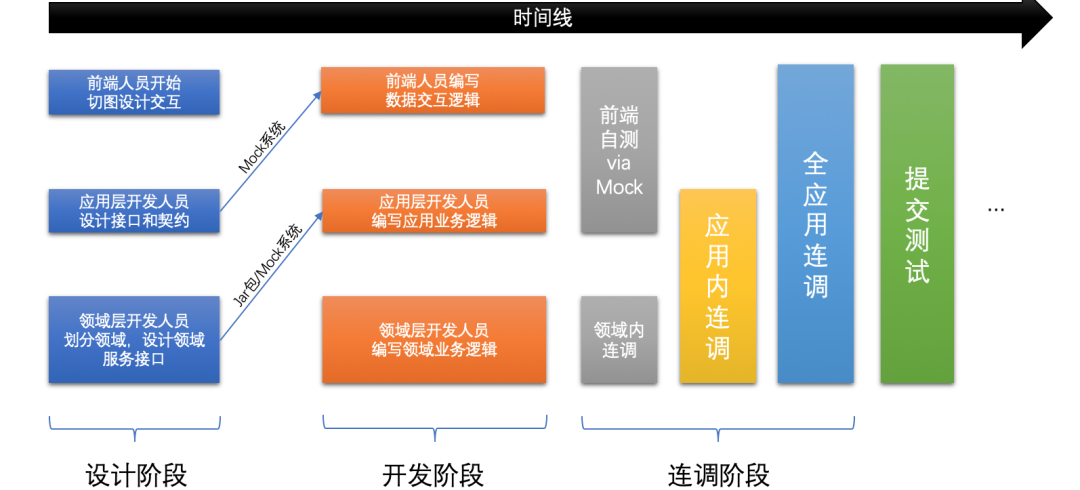
4. 领域划分和微服务化



根据DDD理论，领域建模主要发生在领域服务层，各领域模块都应该是高内聚低耦合的，具有清晰的业务边界。本文不打算讨论具体的DDD建模（服务，工厂，仓库，实体，值对象，聚合等），这需要对DDD有较深入的研究，就目前所从事过的公司来看，似乎没有一家真正严格按照DDD进行项目代码设计的，就像摘要中说的，这对整个软件工程链路上的人员都有较高的要求。有机会可以单独写一篇关于自己对DDD建模的思考和建议，本文更多的是讨论高视角下的领域服务拆分，从而搭建一个低耦合高内聚的微服务体系。如果一定要将微服务和DDD联系起来的话，领域层的微服务就对应了DDD中的领域模块Module，每个Module由多个Service模式对象以及对应的模型对象（实体, 值对象以及它们的聚合）组成。  
从《领域驱动设计：软件核心复杂性应对之道》中我学到的主要有两块：领域设计思想和领域建模模式。本文更多的是对前者的运用，后者的对立模式是贫血模型，大家日常用到的也都是贫血模型，我也觉得贫血模型有存在的必要性，所以本文我们主要从其中借鉴一下领域设计思想。本文所描述的设计理念，并不影响具体的模型设计方法，我们仍然可以在每个微服务中使用DDD领域建模。  
如何切分领域模块并没有一个明确的规则，不同的场景下可能相同的业务块边界也不尽相同。这里提几点领域划分的个人心得：

* 领域设计一定要有清晰的功能边界。一个领域服务对应了一个功能集合，这些功能一定是有一些共性的。比如，订单服务，那么创建订单、修改订单、查询订单列表，一般是订单域的功能集合。
* 领域拆分并不是一步到位的，应当根据实际情况逐步展开。从单体应用到微服务体系的拆分过程能很好的说明这个问题，一上来拆的很细的改造方案一定会死的很惨。所以如果一开始不知道应该划分多细，完全可以先粗粒度划分，然后随着需要，初步拆分。比如一个电商一开始索性可以拆分为商品服务和交易服务，一个负责展示商品，一个负责购买支付。随后随着交易服务越来越复杂，就可以逐步的拆分成订单服务和支付服务。
* 领域拆分并不是一成不变的，应当具体情况具体分析。2015年在大众点评的时候，其订单服务就拆分为了order-service和order-query-service，一来为了读写分离，二来order-query-service作为单独应用可以按需水平扩容。
* 领域可以是多个子领域的一个虚拟集合，换句话说多个微服务也可以形成一个大域，不必纠结于领域和微服务之间的数量对应关系。我们在做架构设计PPT的时候可能就把订单域作为一个领域，代表了这个域就是关于订单的，具体该有几个微服务，这需要更细的详细设计来提供。
* 领域层服务设计应当是调用者无关的。这一点有点像第一点，但是它强调的是领域层服务的设计不应该受调用者的影响，这个观点在《领域驱动设计：软件核心复杂性应对之道》这本书里也可以找得到。领域层服务开发和设计的理念是关注自己的域，一旦边界划分清楚了，开发所需要考虑的永远都只是输入和输出，提供的服务一定是尽可能通用的，面向功能来开发的，而不是面向调用方来开发的。比如某个调用方提出了一个需求：调用方B希望A服务提供一个买汽车的接口，那么A服务设计的接口就应该是buyCar()，而不是buyCarForA()。

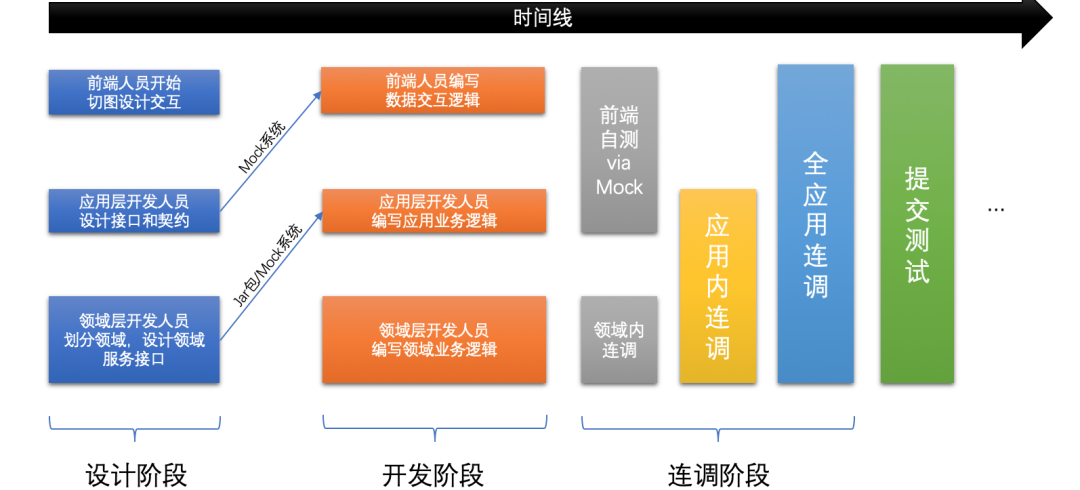
5. Q&A



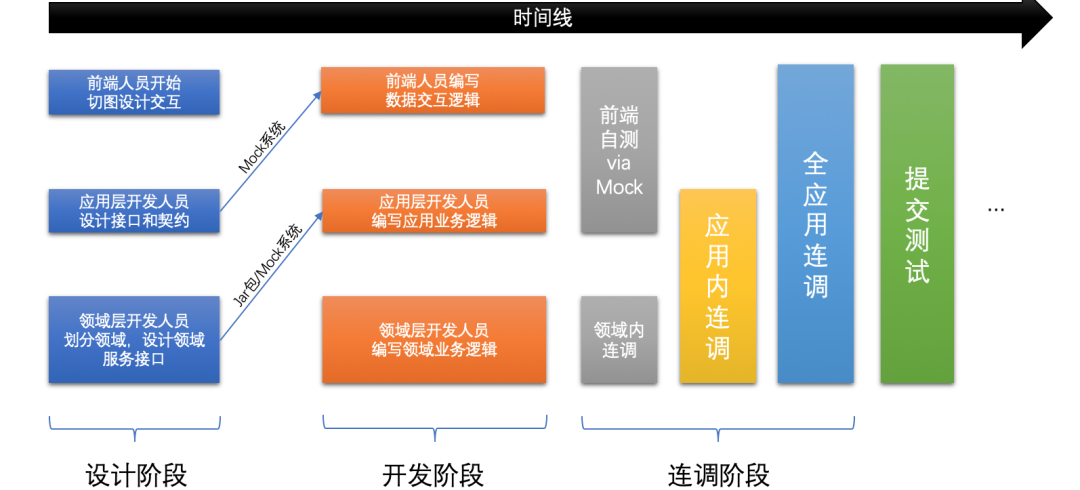
**5.1 能不能在所有层使用数据持久层模型，简单快捷？**  
大家一定听说过不同层的数据模型的叫法不同的概念，比如数据持久层的模型对象叫DBO（database object）或者DPO（data persistence object），领域层的模型对象叫DMO（Domain Model Object）或者就叫Model，数据传输层的模型对象叫DTO（Data Transfer Object）。  
那为啥要这么多模型呢，直接使用Mybatis等ORM框架生成DBO，然后一路吐给前端不是更爽（还真有同事尝试立项写Mybatis插件来实现这种所谓的代码自动化）。我个人建议如果您真的是要搭建一个复杂的大系统，大平台，一定不要偷这种懒，最好的就是做到“一层一模型”（网关层使用应用层模型即可）。各层之间采用手动的数据赋值（getter，setter）来完成，并且拒绝使用BeanUtils.copyProperties()这种工具类，因为这样的工具类会让“一层一模型”形同虚设。下面我们来细谈下不能在每一层都是用数据持久层模型的具体原因：

* 应用层对接网关层，是向面向C端或者调用者的一个数据出口，但是调用者只需要这个出口输出用户感兴趣的数据，并且有些敏感数据不能吐出去。如果应用层（面向调用者）使用的仍然是数据库模型，而开发人员没有在应用层把无关值置空的话（相信我，需求一多，工作一忙，鬼才会在意这些细节），那么数据库里的整条记录就作为接口输出吐出去了。比如订单记录，用户订单列表可能只需要订单ID，商品名称，订单金额。而像商家结算价这种就不能吐出去，万一被有心人察觉到了，用户一定会投诉——你跟商家结算价200，卖给我400？
* 前端或者接口调用方会很痛苦，一个接口契约多一两个无关字段是没关系的，但是一个契约里给的30个字段，我只用到5个，前端会骂娘的，我亲眼见过这种事，设计原则里有个原则叫“迪米特法则”，也叫最小知识原则，接口设计也可以参考这个原则，尽量让你的调用方知道尽可能少的信息点就能完成相关的任务。
* “一层一模型”本质是解耦模型依赖。我在上家公司做架构师时为了兼顾开发的感受，决定让他们可以在领域层和基础设施层都是用数据持久层模型，而只需要在应用层做数据控制（解决第一个问题），然而我的妥协也慢慢露出弊端，开发有时候觉得某个数据库字段命名不合适修改之后，整个引用了该模型的微服务都需要修改，如果一层一模型的话，只需要关联数据库访问的服务修改下DPO和DM的映射就行了，其他上层微服务都是依赖DM的。虽然我们不鼓励随意改动数据库字段，但设计框架上最好能支持这种情况。

刚开始推广“一层一模型”的时候，会有耍小聪明的开发去把下一层的模型POJO直接拷贝过来改个名字，然后用BeanUtils.copyProperties()完成赋值，这样跟直接使用数据持久层模型就没有区别了，所以要杜绝这种情况的发生。  
**5.2 为啥需要应用层，领域层微服务直接通过网关暴露不就行了吗？**  
对于习惯了单体应用开发者来说，一个微服务很可能就直观对应成了一个个垂直的应用服务，每个服务间的关系是这样的：



其实这样的体系本质上仍然不能解决软件的复杂性，这只是把系统简单粗暴的拆分了，耦合问题仍然很严重，甚至这很有可能比原来的单体应用更复杂（多对多依赖），如果使用微服务体系来处理复杂系统，其服务体系应当是这样的：



这两幅图的区别在于，其实第一幅图中的每个服务都包含了完整的2～3层，所以不再需要单独的应用层。而第二幅图各个领域模块互相协作，对外提供服务时，则需要有一层直面用户需求的应用层。  
达成了微服务体系是解决复杂系统的出路之一这个共识后，我们再来看“应用层服务存在的必要性”有哪些理由：

* 统一权限校验：如上文所说，网关层只负责网络级的安全防护，业务层的权限校验则需要应用层来完成，试想一个没有应用层的微服务体系，就意味着每一个微服务都需要加上权限校验逻辑，这不仅编码上困难（可以用过滤器，AOP），而且对于成千上万个微服务（据了解，腾讯目前微服务数量已经超过2万，大众点评有将近千个微服务）来说，这会浪费大量时间，调用链越长，浪费的时间越多。换句话说，微服务体系有一个不突出但是很重要的特征—— 领域间环境安全，领域间的通信应当是可信的 ，否则分布式的缺点（多服务意味着多次通信）会被加剧。
* 业务数据网关：举个例子，一个order-service提供了一个queryOrder的接口，输入起始日期查询对应的订单列表，其有2个消费者：C端网站应用服务 和 报表应用服务 ，C端网站应用服务 只需要知道订单的基本信息如下单时间、商品名称、金额就可以了，而报表应用服务是给管理者看的，需要的订单数据很全，除了C端网站应用服务需要的之外，还需要看平台与商家的结算金额。根据第4部分最后一点的思路，我们肯定不能为调用方写定制接口（写不完的，有的要这个数据，有的要那个数据，每次新增调用方，领域服务还得找人修改）。而如果我们统一使用的全量数据，并且没有应用层（同样的也没有应用层模型DTO了），那么很可能我们吐出去的数据包含了我们与商家的结算价，这会引发很多不必要的麻烦的。所以应用层还充当了业务数据网关的作用，应用层应用服务需要保证仅吐出调用方感兴趣的数据。
* 资源控制和缓存：想象一下双十一高并发的情况，如果查询库存每次都查库是多么恐怖的一件事。所以一般仅在支付的时候做一次库存校验，而在商品展示时查缓存的库存即可。那么问题来了，如果没有应用层，缓存直接放在库存微服务上是否可行呢？首先这会入侵库存领域，库存微服务需要按照调用方的需求做特定时间的缓存，而不是自己想缓存多久就多久，我想库存微服务的开发者也会很不满的，他会提出，让你自己去做缓存。他的方案是科学的，因为还有一些其他服务可能需要实时的数据。这时候就需要有一层来做对其下方微服务返回的数据按照应用自身的需求进行必要的缓存，而不是把这些需求都推给资源提供方，想象一下一个资源提供方有多少需求者，每个需求方都有自己的定制需求，该多痛苦。当然这一点也不是说微服务自身不能做缓存，微服务自身的缓存一定是考虑自身域的合理性后的一个措施（比如订单查询服务会做一个500ms的缓存，因为不会有正常人500ms里点两次查询还必须要求两次都是最新的），而不是由调用方来决定的。
* 资源聚合和加工：其实第2点也有加工的味道在里面，只是这里更多的是描述应用层应用根据自身需求来对下层返回的数据进行聚合和处理的过程。举个例子就能很好的说明这一点：任何APP都有首页，而首页的数据可能是五花八门的，可以有用户昵称、最近下的订单简要信息、最近支出曲线、积分信息等。这4个信息可以来自4个领域微服务，他们是：用户中心、订单中心、支付中心和积分中心。那么有读者会说，直接暴露微服务让前端分别调用4个接口再做聚合不是也行吗？显然这种粗暴的方式是极其不合理的，会额外增加广域网网络调用3次不说，还传输了很多不必要的信息。
* 应用隔离和流控：如果将每个领域服务直接暴露到网关层对外提供服务，那么在多应用场景下，多个应用间是共享这些服务能力的，在服务降级的时候，如果需要按照应用进行降级（比如将优先级不高的应用进行限流），就很难实现。但如果每个应用对应了一个应用层服务，只需要对其暴露的网关接口进行统一限流就行了，或者在应用层做一个开关，将其流量阻止在应用层，而不是拖垮整个领域服务。举个例子，假如我们的平台不仅有自己的网站服务，还有第三方的对接服务，如果某个第三方被攻击而我们直接将领域服务暴露了出去，那么我们就需要在各个领域层服务里去编写对应的开关，这将侵入领域层服务，导致不必要的耦合。而有了应用层这些都不是问题，因为应用层充当了一个调度者的角色，调度者可以很轻松的决定是否调度下层的服务。

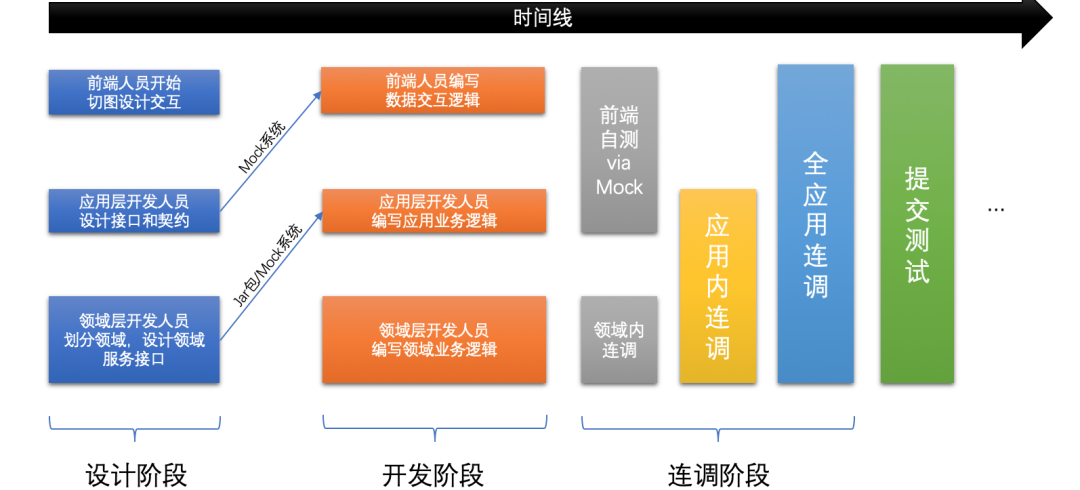
为了加深对应用层的理解，我们举个代码的例子，假如我们写一个很简单的首页应用：

Response getHomeData(Request request){  
  
    String nickName = userService.getNickName(request.getUserId());   
    OrderInfo orderInfo = orderService.queryLatestOrder(request.getUserId());  
    CostTrend costTrend = payService.queryCostTrend(request.getUserId());  
    Integer points = pointService.queryAvailablePoints(request.getUserId());  
    return new Response(nickName, orderInfo, costTrend, points);  
}

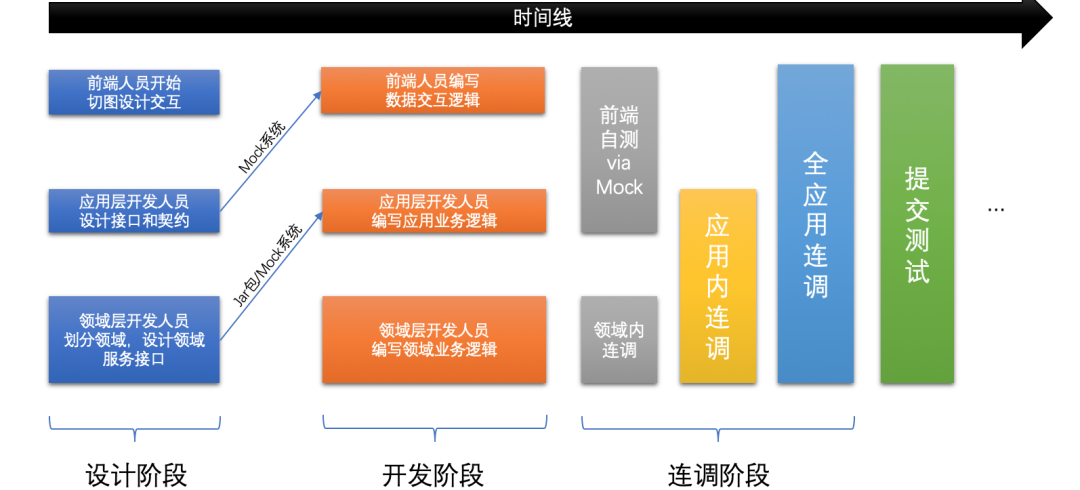
这里的4个服务类实例userService，orderService、payService和pointService如果都是本地的方法，那么这就是一个单体应用，而微服务化后这4个可能都是微服务了，但是应用层应用的结构还是可以不用变化（现在很多的RPC框架都做到了与调用本地方法无差别）。这就是应用层的位置所在。  
**5.3 什么是反模式？**  
这里的反模式是指《领域驱动设计：软件核心复杂性应对之道》这本书里提到的与DDD相违背的模式，也是Eric极其反对的一种模式，即SmartUI模式（注意反模式不等于SmartUI，只是在本书中作为一个反模式的例子而已），这是一种什么样的模式呢，其实我很早之前做C++ Builder（和Delphi很像）的时候还不知道，C++ Builder就是一种SmartUI模式。  
但其实SmartUI并没有错，对于小规模的PC本地应用开发来说也是有很多好处的。举个例子，C++ Builder中在窗体上添加一个按钮，然后双击按钮添加事件，这样就跟实际操作的时候有机的结合了起来。换句话说就是使用界面驱动业务开发。在大型系统的开发上，这种模式是害人精，我很理解Eric为啥这么讨厌它。  
曾有一次我带领着一个团队在浦东一别墅做封闭式开发，在过完产品需求后，家里出了点事我请了假，回来后发现产品经理竟然直接让开发按照UI原型来设计数据库，我审核的时候发现这些开发设计的表有极其多的冗余，而有一些重要的过程变量值却没有考虑到。比如他们会为每个页面建几个表，这显然是行不通的，科学的方法是拆分领域，每个领域自己建立自己的表。UI只是应用层整合了各领域服务的数据并且处理后输出的一种展示。  
**5.4 分层设计的开发步骤是怎样的？**  
假设我们以一个标准的SaaS项目为主，也就是表示层是前端页面（可以是APP，H5，M站，小程序，PC站等），那么高效的一种开发步骤可以是这样的：

1. 业务、产品、开发PM进行需求评审（可行性等）
2. 产品准备好原型
3. 产品、开发（前后端）、架构师（或有架构师能力的资深开发）开会过PRD，了解要做什么
4. 架构师开始设计领域（资深架构师一下午就能搞定），前端开始切图，应用层开发开始按照UI和PRD设计前端每个页面使用的Restful接口（比如直接Springfox代码生成Swagger）
5. 架构师设计完领域后分工给领域层开发，进行领域边界明确，然后领域层开发开始设计数据库表等。
6. 这样前后端开发就同时开工了。
7. 开发初步完成后，自测加连调。
8. 后续就是测试发布了。

具体阶段和时间线可以参考下图：



6. 结语



真实技术开发日常讨论的经常是高并发、多线程、大数据，分布式、RPC，很少有人讨论软件架构设计方面的，架构师文化不应该只是对某个技术点的深入挖掘，也应该多讨论些大型软件设计理念方面的～ 欢迎大家有任何软件/系统/平台设计方法方面的问题一起来留言区探讨～