02 | 如何利用“拆分”降低架构复杂度?

今天将开始介绍如何利用“拆分”降低架构复杂度。

上一讲我们从**技术的目的性**这个维度，将业务后台系统的类型归为三大类，读业务、写业务及扣减业务。除了基于技术维度的拆分，在后台架构里还有很多其他形式的拆分，比如上一讲提到的外卖系统架构，整个后台可以拆分为了用户、订单、商品、价格等模块。

通过拆分可以将一个涉及面广、复杂度高的系统拆解为多个小模块，让各个团队单独负责并专项击破。**拆分是架构设计大型复杂系统的第一步，对降低系统复杂性有着决定性的意义，它也是架构师的必备技能之一**。

在本讲，我会给出一个实际业务场景，手把手带你做一次完整的系统拆分，**通过理论和实战两个层面，帮助你快速掌握这项可以有效降低系统复杂度的技能**。

为什么要做拆分

人解决复杂问题的能力是有限的，当问题涉及面广、情况复杂时，自然会去寻找方法提升效率。我们的业务后台系统就是一个“复杂问题”，而解决“这个问题”的方法便是拆分——将复杂问题拆解为多个相对简单的小问题，分而治之、各个击破，这样做极大地提高了解决复杂问题的可能性和效率。

以电商系统为例，建设一个支持千万用户同时使用、日均流量上亿的电商系统也是一个十分庞大、复杂、涉及面广的工程。至少需要上百名研发、产品、测试等不同职能的人员共建。想象一下，如果没有系统拆分，所有人聚集在一起，开会讨论需求和技术方案的场景。

先不说别的，把所有人召集齐都是一项十分费时的事情，应该是一个顶级的项目经理才能做到。其次是沟通的效率，上百人一起开会，七嘴八舌，一天都得不出什么重要结论，更别说梳理出细节流程了。最后，在开发的时候，上百名研发使用同一个代码仓库，每天你至少需要花费几个小时在解决代码冲突的问题上。使用此方式推进项目，上线可能遥遥无期了。即使能够上线，线上质量也是堪忧。

面对这样的情况，我们该如何解决呢?

首先，应该是几个业务专家、技术架构一起对整个系统进行全面的梳理和分析。然后便是对系统进行拆分，将大系统按一定的规则拆分成多个小模块。注意，这里我多说一句，“一定的规则”便是我们前面所讲的“按照不同维度”。

拆分后，上百人的产研测团队就可以认领工作了，各个团队安心处理自己模块内的工作（产品设计、代码开发等）。对于模块间需要交互的地方，每个团队出一个资深的产品和架构师进行沟通即可。沟通成本降低了，效率自然也就提升了。

通过上述案例，可以看出拆分在大型系统开发中的重要作用。**一个好的拆分能够降低各个模块间的耦合性，极大地降低效率消耗，提升系统成功的可能性**。如果你立志成为架构师，那么就必须将此项重要技能收入囊中。

何时拆分

此外，一些系统在建设时并没有预计到最终会发展到什么量级，为了满足快速上线的要求，没有经过拆分就直接采用了单体架构的模式进行落地。对于此类系统，是不是需要立马做拆分呢?

其实并不是。当单体架构存在以下 3 类问题时，建议考虑进行拆分。

1. 系统常年处于多需求并行开发，导致代码维护成本太大

如果一个系统变得非常庞大，它的需求及并行的需求就会非常多。每个并行需求都会有一个开发分支，当开发需求完成陆续上线，那么在代码合并时就会产生冲突，比如你写的一个功能块被别人改了、新增了一个类似的公共类等，此时就可以考虑进行系统拆分。

2. 在开发一个需求时，花费几天时间阅读代码，而开发只要一半甚至更少的时间

在一个过于庞大的系统里，如果你长期处于熟悉代码的时间和实际编写的时间配比不对等的情况，此时就应考虑进行系统拆分。因为信息过载了，你无法完全掌握此系统。如果不拆分，每次重新熟悉代码并进行修改，可能会导致 Bug 频发，线上质量低下。

经过拆分后，你只需负责你熟悉的小模块，其他部分依赖第三方接口即可，让更专业的人保证它的质量。

3. 在一个需求上线前，需要召集上百人对齐上线风险和上线步骤

对于上面的第二个问题，你可能会说，我找不同的人负责系统里的不同部分不就好了。这虽然可以解决熟悉的问题，但耦合度是解决不了的。一个需求由十个人开发一个系统和三波人开发三个系统，这期间的沟通成本、耦合度等是无法比拟的。想象一下，一屋子开会的人你一言我一语和确定好边界后，三波人单独开会，一定是后者更高效。

拆分的前提条件

如果你所负责的系统满足上述三个场景，我想你已经准备好大干一场了。但在开始拆分之前，我有几个点要提醒你。

第一是团队和人员的准备。 拆分的主要原因是信息过载，因此在拆分后，对应的系统也要划分给新增的团队，而不是原有人员继续维护，这样只是换汤不换药，信息依然过载。亚马逊的 CEO 曾提出“两个比萨”理论——一个团队的人员能够吃下两个比萨刚好。因此拆分后，建议以三个人左右为单位进行组织和划分。

第二是在进行拆分后，各个模块不在一个进程内，需要采用接口依赖的形式。此时，对于 RPC 框架及对应的监控、降级等中间件基础设施要求较高了。如果公司的中间件和基础设施跟不上，建议先将此加强再进行拆分，以免拆分后还要填此处留下的坑。

如何进行拆分

接下来我们再来看看到底如何进行拆分。

其实拆分就和切蛋糕类似，将一整块蛋糕切分成多个小块的方法有很多。你可以横着切或者竖着切，不管怎么切，蛋糕最终都将被分成很多小块。只是选择不同的切法，最终得到的蛋糕形状不一样。同样架构的拆分也有很多方式，对一个业务系统采用不同的拆分方法将会得到不同的子模块。

在上一讲中，我们采用了“目的性”这个维度，将后台业务系统拆分或者说归类为三大类，读服务、写服务和扣减服务。这里你可能会有疑问，拆分后可以得到三种类型的模块，难道所有的后台系统都只有这三个模块?

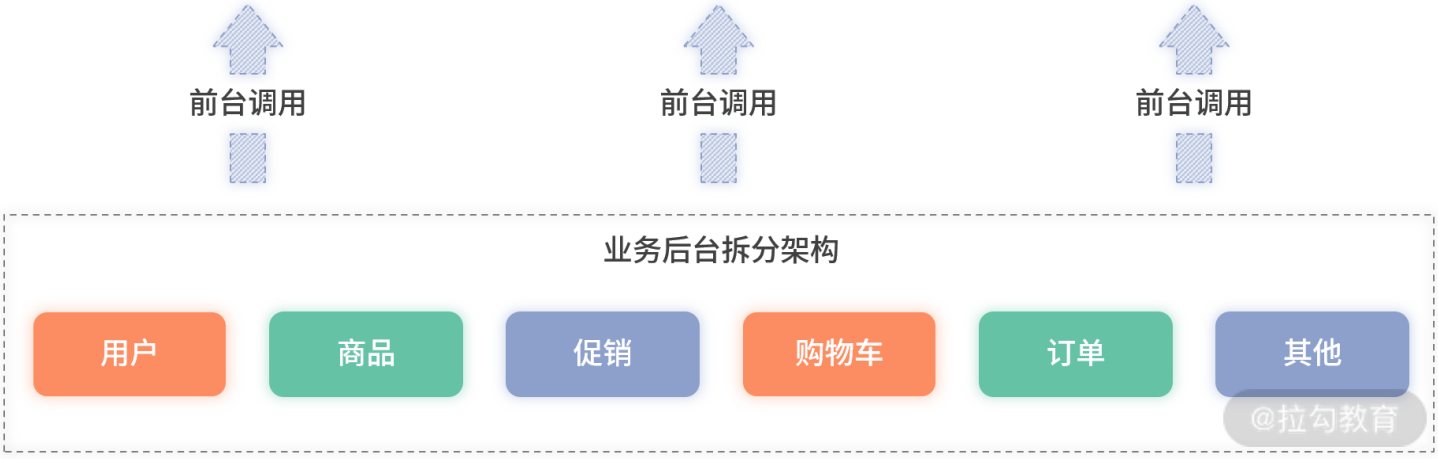

图 1：电商后台拆分架构

以上图 1 的电商后台拆分架构为例，电商后台一般包含用户模块、商品模块、促销模块、购物车模块、订单模块等。其实，这正好验证了我们前面提到的，架构拆分有很多方式，只是不同拆法得到的结果不一样。

“目的性”是一种偏技术视角的拆分方式，而图 1 展示的则是一种偏业务视角的拆分方式。可以发现拆分后的模块和你平时购物的流程极为相似，比如，你在购物时，要先通过认证登录 App，再浏览商品、查看促销活动及价格，确认购买后添加购物车并提交订单。这种按用户使用流程进行拆分的方法叫作**业务流拆分法。**

并没有哪一种架构拆分绝对正确，因为架构是论述题而不是选择题。在实战中，你需要针对不同复杂度的业务类型灵活选择。**不管经过多少次重复拆分和组合，目标只有一个——降低系统复杂度、减少耦合提升效率。**

对于复杂的系统架构，先竖着切一刀，称为垂直拆分。经过垂直拆分后，如果系统还是太复杂，就可以横着再来一刀，称为水平拆分或者分层拆分。在实际场景中，你可以根据需求重复做垂直拆分或者水平拆分。每一次拆分的依据可以不一样，可以依据业务流程，也可以依据纯技术。

虽没有绝对的架构拆分标准，但从我的经验来看，大型系统通常先采用两次垂直拆分再加一次水平拆分。这两次垂直拆分有一定的先后顺序，先按业务维度再按技术维度。接下来，我将以最近经常使用的微信读书作为分析案例，和你一起从零开始对一个陌生的后台架构进行拆分。

首先，我们先来分析一下微信读书的业务流程，然后再开始第一步的垂直拆分。使用微信读书阅读大致经过以下几个步骤：

1. 注册并登录 App；
2. 在书城里查找你想要阅读的图书，并浏览书籍的版本目录和介绍；
3. 确定是你想要的书籍并且价格适合，就会产生购买；
4. 支付完成之后，开始阅读。

为了方便你理解，我制作了一张微信读书业务流程图，如下所示：

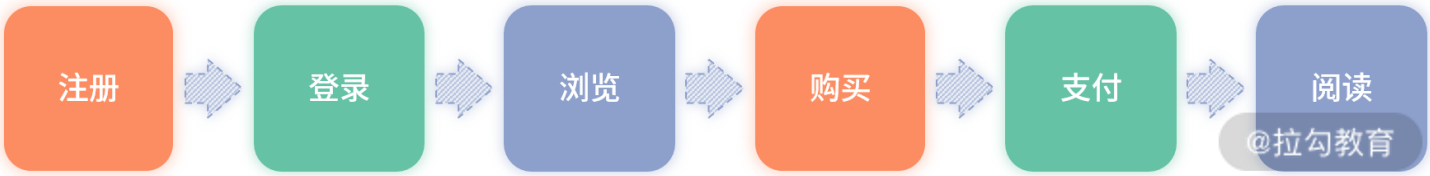

图 2：微信读书业务流程图

结合前面的文字描述与图 2 的流程图展示之后，我想你应该知道第一次垂直拆分的结果了。

1. 注册和登录应该是用户模块。
2. 在微信读书里书籍属于售卖商品，因此浏览书籍称为商品模块。
3. 书籍价格展示区域称为价格模块或者叫促销模块，有些时候价格是根据运营节奏或者用户是否为会员等条件计算出来的，这种经过计算输出价格的区域叫促销模块。
4. 购买是订单模块。
5. 最后完成支付则是支付模块。
6. 支付完成后支持实际阅读的区域是阅读模块，它记录了我每次的阅读进度、书签、笔记等。

至此，我们一起按照业务流程完成了对微信读书后台的第一次拆分，得到了微信读书的六大后台模块。因为第二次按技术维度的垂直拆分较通用，上述每个模块均可以按此进行拆分，所以这里我只选取用户模块和支付模块进行分析。

用户模块对外会提供两个基本功能，一个是用户注册，另一个则是登录验证，可以称为查询，因为除了验证之外，App 里还显示了用户的名称及其他信息。按技术维度进行拆分时，可以将注册和修改归为写业务。

除此之外均为读业务，包含验证、查询基本信息、查询用户是否为会员等。经过第二次拆分后，用户模块一分为二——用户读模块和用户写模块。按此套路，我们可以将用户读模块进一步拆分。这里只是讲解拆分的思想，并不是鼓励你一直拆分下去。拆分太细会出现多个模块，如果团队人手不够，即使利用“拆分”也达不到降低复杂度的目的。

另外，支付模块通常会对外提供余额查询与支付扣减的能力，此处可以将支付模块分为额度查询模块和支付扣减模块，同样完成了支付模块的第二次技术维度的垂直划分。拆分结果如下图 3 所示：

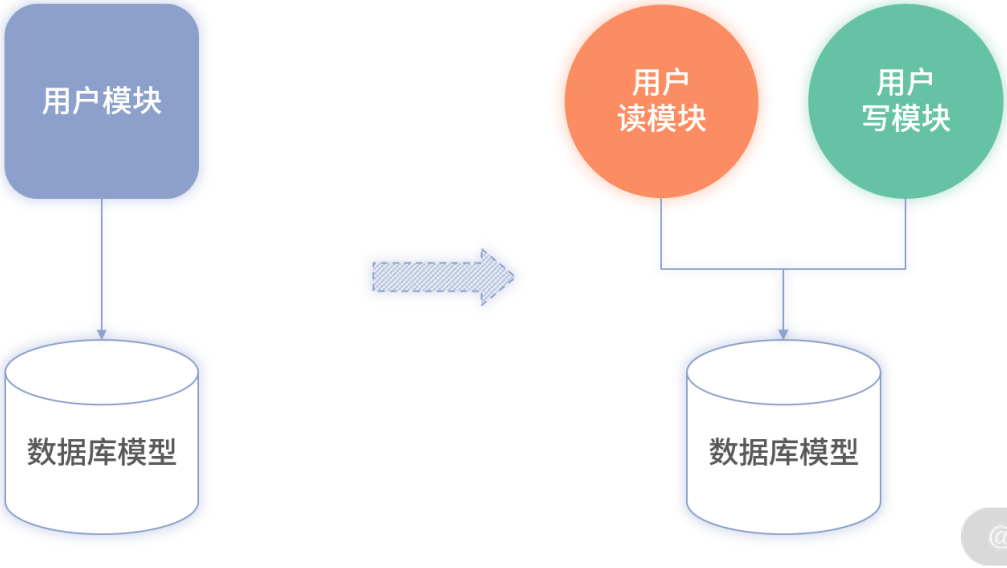

图 3：技术维度的垂直拆分架构

至此，我们已经完成了两次垂直维度的拆分。拆分的结果与后台系统的三大类：读模块、写模块及扣减模块基本一致，只是本讲的模块都带上了具体的业务前缀。这里我再强调一点，**任何技术都是服务于业务的，脱离业务的技术无法发挥它的价值。本专栏讲解时去掉了具体的业务前缀，不是不关注业务，而是希望你能够深入理解这些业务背后的通用点，从而更好地服务于业务。**

完成两次垂直拆分后，就可以做最后一步拆分了，即水平拆分或者叫作分层拆分。水平拆分的依据是按易变度或共性度 。经过水平拆分，上层的称为易变模块，下层的称为非易变模块。越靠下面的模块越稳定、越共性、越不易变化。

拆分后，对于非易变的模块，我们只需要编写、修改一次或者零星几次即可，对于易变的模块则需要投入更多的人力去维护。因为易变与非易变模块已经拆开，易变模块进行需求改造对非易变模块基本上没有任何影响。下面我们以第二步垂直拆分形成的模块作为分层拆分的实战。

经过第二步按技术维度的垂直拆分，形成了用户的读模块、写模块、任务模块等。在设计或者开发时，你会发现这些模块都会连接数据库或者其他存储。对于这些连接数据库的代码，基本都是对象映射，将入参的对象转换为数据库 ER 格式的对象。

如果是 Java 应用，还会包含 MyBatis 或者 Hibernate 相关的数据库 ORM 映射的脚手架代码，其他语言以此类推。这些通用的代码只编写一次即可形成一个模块，比如数据访问模块，可以供用户的读模块和写模块共同使用，这就是水平拆分的结果，它的上层是读写模块、下层为共性的数据访问模块。具体形式见下图 4：

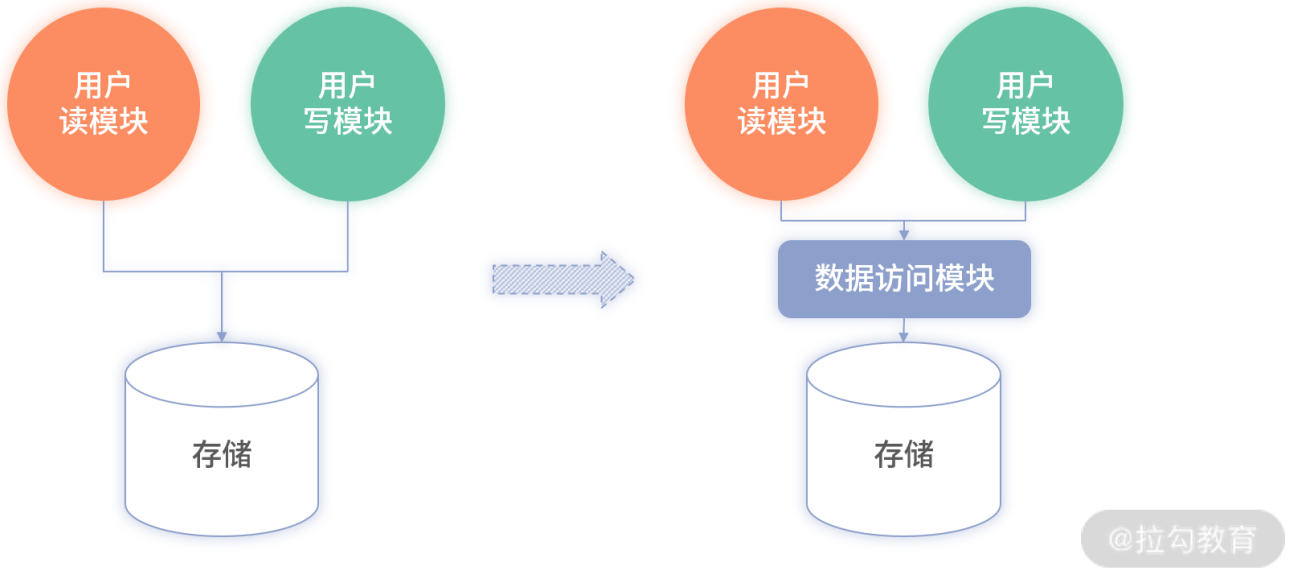

图 4：共性维度的水平拆分架构

如果单独部署水平拆分得到的数据访问模块会因为网络、数据序列化等因素降低读写模块的性能。为了规避此问题，在实际应用中，可以将数据访问模块的代码单独一个工程，但在编译时以动态包的形式链接进用户读模块、用户写模块里，这样就两全其美了。

至此我们一起完成了两次垂直拆分、一次水平拆分的架构拆分实战。在实战中，你可以根据需求选择比此次实践多或少的拆分。另外，架构和历史也是一样，分久必合合久必分，但在分分合合的过程中应遵循上述原则和手段。

最后我想说，架构拆分不是完美无缺的，它也会存在一些问题，比如拆分后带来的分布式事务、调用链路变长、模块变多消耗机器变多等问题。针对这些问题，我将在模块五详细讲解。

总结

在本讲，我向你介绍了为什么要做服务架构的拆分以及哪些情况可以暂不进行拆分。我们再来复习一下系统拆分的几个主要原因。

1. 当需求不断叠加导致并行开发和上线时，通过拆分可以减少相互影响。
2. 当维护一个覆盖范围比较广的业务系统，从而导致研发人员业务专业度不够高时，通过拆分可以适当聚焦，提升专业度。
3. 当一个系统范围较广同时线上 Bug 不断时，就需要适当拆分，逐个击破。

在落地进行拆分时，有几个重要准则需要你牢牢记住。

1. 拆分是按维度逐层进行，从顶层逐步向下。在顶层按业务及业务流程进行垂直拆分，而不是按技术或其他。
2. 在此之后，对于拆分得到的具体模块，可以按读写分离、在线离线分离、快慢分离、场景分离等方式做进一步的水平拆分。
3. 在模块内部的垂直拆分完成之后，可以按易变与稳定、共性与非共性进行水平拆分。需要注意的是，第二步的垂直拆分和最后的水平拆分是交替进行的，并无非常清晰的边界和先手顺序。

通过拆分我们得到了可以独自进行详细架构设计的读服务模块、写服务模块及扣减服务模块，你可以对各个类型的模块进行逐个击破了。

最后，我再留给你一道思考题，**你现在使用的是单体架构还是经过拆分的微服务架构？拆分维度是什么？可以在留言区写下按此维度拆分的原因，我会选取一位同学的内容做点评分析。**

这一讲就到这里，下一模块我们讲解如何构建一个高性能的读服务。实战之旅即将开始，你准备好了吗？