https://mp.weixin.qq.com/s/m9lQAGFbg4YU5wN2Imv8fA

**阿里是如何用DDD来拆分微服务的？**

背景

ALIWARE

微服务在最近几年大行其道，很多公司的研发人员都在考虑微服务架构，同时，随着 Docker 容器技术和自动化运维等相关技术发展，微服务变得更容易管理，这给了微服务架构良好的发展机会。  
在做微服务的路上，拆分服务是个很热的话题。我们应该按照什么原则将现有的业务进行拆分？是否拆分得越细就越好？接下来一起谈谈服务拆分的策略和坚持的原则。

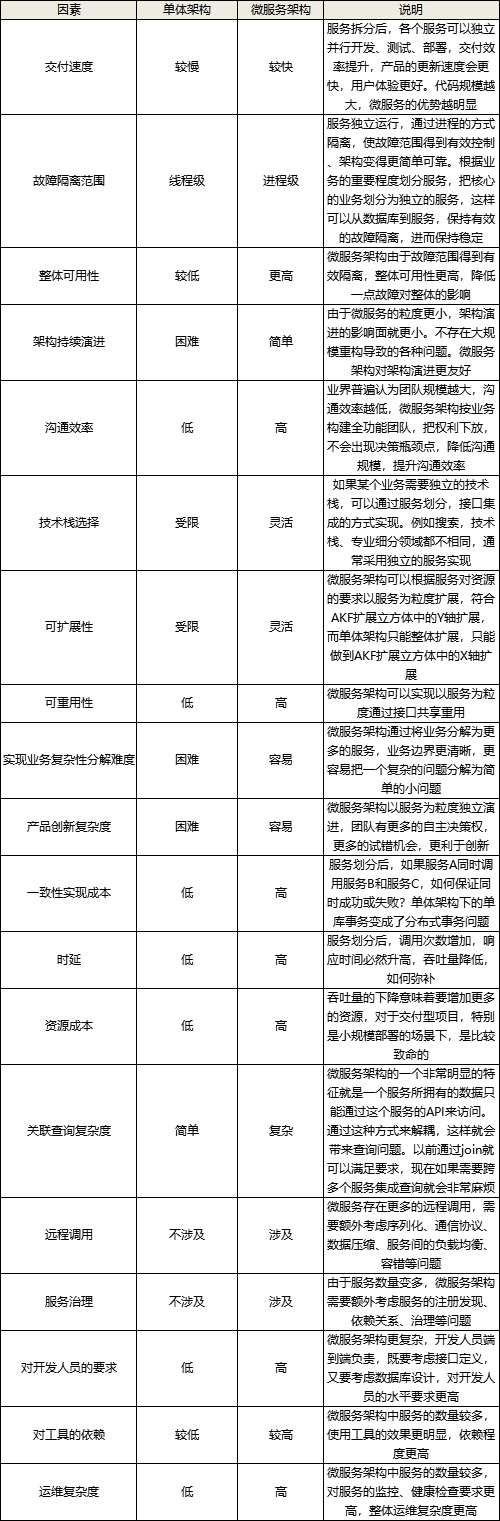
拆分目的是什么？

ALIWARE

在介绍如何拆分之前，我们需要了解下拆分的目的是什么，这样才不会在后续的拆分过程中忘了最初的目的。  
拆分的本质是为了将复杂的问题简单化，那么我们在单体架构阶段遇到了哪些复杂性问题呢？首先来回想下当初为什么选用了单体架构，在电商项目刚启动的时候，我们只希望能尽快地将项目搭建起来，方便将产品更早的投放市场进行快速验证。在开发初期，这种架构确实给开发和运维带来了很大的便捷，主要体现在：

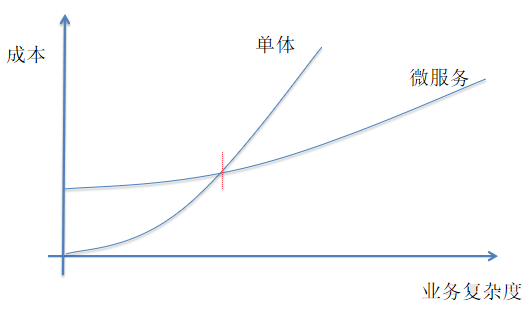
* 开发简单直接，代码和项目集中式管理。
* 排查问题时只需要排查这个应用就可以了，更有针对性。
* 只需要维护一个工程，节省维护系统运行的人力成本。  
  但是随着功能越来越多，开发团队的规模越来越大，单体架构的缺陷慢慢体现出来，主要有以下几个方面：在技术层面，数据库的连接数成为应用服务器扩容的瓶颈，因为连接 MySQL 的客户端数量是有限制的。

**除此之外，单体架构增加了研发的成本抑制了研发效率的提升。**比如公司的垂直电商系统团队会被按业务线拆分为不同的组。当如此多的小团队共同维护一套代码和一个系统时，在配合的过程中就会出现问题。不同的团队之间沟通少，假如一个团队需要一个发送短信的功能，那么有的研发同学会认为最快的方式不是询问其他团队是否有现成的，而是自己写一套，但是这种想法是不合适的，会造成功能服务的重复开发。由于代码部署在一起，每个人都向同一个代码库提交代码，代码冲突无法避免；同时功能之间耦合严重，可能你只是更改了很小的逻辑却导致其它功能不可用，从而在测试时需要对整体功能回归，延长了交付时间。模块之间互相依赖，一个小团队中的成员犯了一个错误，就可能会影响到其它团队维护的服务，对于整体系统稳定性影响很大。  
**最后，单体架构对于系统的运维也会有很大的影响。**想象一下，在项目初期你的代码可能只有几千行，构建一次只需要一分钟，那么你可以很敏捷灵活地频繁上线变更修复问题。但是当你的系统扩充到几十万行甚至上百万行代码的时候，一次构建的过程包括编译、单元测试、打包和上传到正式环境，花费的时间可能达到十几分钟，并且任何小的修改，都需要构建整个项目，上线变更的过程非常不灵活。  
而这些问题都可以通过微服务化拆分来解决。



拆分时机应该如何决策？

产品初期，应该以单体架构优先。因为面对一个新的领域，对业务的理解很难在开始阶段就比较清晰，往往是经过一段时间之后，才能逐步稳定，如果拆分过早，导致边界拆分不合理或者拆的过细，反而会影响生产力。很多时候，从一个已有的单体架构中逐步划分服务，要比一开始就构建微服务简单得多。同时公司的产品并没有被市场验证过，有可能会失败，所以这个投入的风险也会比较高。  
另外，在资源受限的情况下，采用微服务架构很多优势无法体现，性能上的劣势反而会比较明显。如下图所示。当业务复杂度达到一定程度后，微服务架构消耗的成本才会体现优势，并不是所有的场景都适合采用微服务架构，服务的划分应逐步进行，持续演进。产品初期，业务复杂度不高的时候，应该尽量采用单体架构。



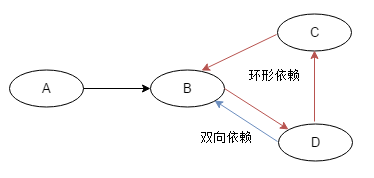
随着公司的商业模式逐渐得到验证，且产品获得了市场的认可，为了能加快产品的迭代效率快速占领市场，公司开始引进更多的开发同学，这时系统的复杂度会变得越来越高，就出现单体应用和团队规模之间出现矛盾，研发效率不升反降。上图中的交叉点表明，业务已经达到了一定的复杂度，单体应用已经无法满足业务增长的需求，研发效率开始下降，而这时就是需要考虑进行服务拆分的时机点。这个点需要架构师去权衡。笔者所在的公司，是当团队规模达到百人的时候，才考虑进行服务化。  
当我们清楚了什么时候进行拆分，就可以直接落地了吗？不是的，微服务拆分的落地还要提前准备好配套的基础设施，如服务描述、注册中心、服务框架、服务监控、服务追踪、服务治理等几大基本组件，以上每个组件缺一不可，每个组件展开又包括很多技术门槛，比如，容器技术、持续部署、DevOps 等相关概念，以及人才的储备和观念的变化。**微服务不仅仅是技术的升级，更是开发方式、组织架构、开发观念的转变。**  
至此，何时进行微服务的拆分，整体总结如下：

1. 业务规模：业务模式得到市场的验证，需要进一步加快脚步快速占领市场，这时业务的规模变得越来越大，按产品生命周期来划分（导入期、成长期、成熟期、衰退期）这时一般在成长期阶段。如果是导入期，尽量采用单体架构。
2. 团队规模：一般是团队达到百人的时候。
3. 技术储备：领域驱动设计、注册中心、配置中心、日志系统、持续交付、监控系统、分布式定时任务、CAP 理论、分布式调用链、API 网关等等。
4. 人才储备：精通微服务落地经验的架构师及相应开发同学。
5. 研发效率：研发效率大幅下降，具体问题参加上面拆分目的里提到的。

拆分时应该坚守哪些指导原则？

**1. 单一服务内部功能高内聚低耦合**

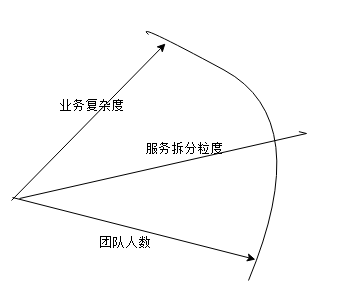
也就是说每个服务只完成自己职责内的任务，对于不是自己职责的功能交给其它服务来完成。  
**2. 闭包原则（CCP）**  
微服务的闭包原则就是当我们需要改变一个微服务的时候，所有依赖都在这个微服务的组件内，不需要修改其他微服务。  
**3. 服务自治、接口隔离原则**  
尽量消除对其他服务的强依赖，这样可以降低沟通成本，提升服务稳定性。服务通过标准的接口隔离，隐藏内部实现细节。这使得服务可以独立开发、测试、部署、运行，以服务为单位持续交付。  
**4. 持续演进原则**  
在服务拆分的初期，你其实很难确定服务究竟要拆成什么样。从微服务这几个字来看，服务的粒度貌似应该足够小，但是服务多了也会带来问题，服务数量快速增长会带来架构复杂度急剧升高，开发、测试、运维等环节很难快速适应，会导致故障率大幅增加，可用性降低，非必要情况，应逐步划分，持续演进，避免服务数量的爆炸性增长，这等同于灰度发布的效果，先拿出几个不太重要的功能拆分出一个服务做试验，如果出现故障，则可以减少故障的影响范围。  
**5. 拆分的过程尽量避免影响产品的日常功能迭代**  
也就是说要一边做产品功能迭代，一边完成服务化拆分。比如优先剥离比较独立的边界服务（如短信服务等），从非核心的服务出发减少拆分对现有业务的影响，也给团队一个练习、试错的机会。同时当两个服务存在依赖关系时优先拆分被依赖的服务。  
**6. 服务接口的定义要具备可扩展性**  
服务拆分之后，由于服务是以独立进程的方式部署，所以服务之间通信就不再是进程内部的方法调用而是跨进程的网络通信了。在这种通信模型下服务接口的定义要具备可扩展性，否则在服务变更时会造成意想不到的错误。比如微服务的接口因为升级把之前的三个参数改成了四个，上线后导致调用方大量报错，推荐做法服务接口的参数类型最好是封装类，这样如果增加参数就不必变更接口的签名，而只需要在类中添加字段就可以了  
**7. 避免环形依赖与双向依赖**  
尽量不要有服务之间的环形依赖或双向依赖，原因是存在这种情况说明我们的功能边界没有化分清楚或者有通用的功能没有下沉下来。



**8. 阶段性合并**  
随着你对业务领域理解的逐渐深入或者业务本身逻辑发生了比较大的变化，亦或者之前的拆分没有考虑的很清楚，导致拆分后的服务边界变得越来越混乱，这时就要重新梳理领域边界，不断纠正拆分的合理性。

拆分的粒度是不是越细越好？

目前很多传统的单体应用再向微服务架构进行升级改造，如果拆分粒度太细会增加运维复杂度，粒度过大又起不到效果，那么改造过程中如何平衡拆分粒度呢？



**三个火枪手原则**

为什么说是三个人分配一个服务是比较理性的？而不是 4 个，也不是 2 个呢？  
首先，从系统规模来讲，3 个人负责开发一个系统，系统的复杂度刚好达到每个人都能全面理解整个系统，又能够进行分工的粒度；如果是 2 个人开发一个系统，系统的复杂度不够，开发人员可能觉得无法体现自己的技术实力；如果是 4 个甚至更多人开发一个系统，系统复杂度又会无法让开发人员对系统的细节都了解很深。  
其次，从团队管理来说，3 个人可以形成一个稳定的备份，即使 1 个人休假或者调配到其他系统，剩余 2 个人还可以支撑；如果是 2 个人，抽调 1 个后剩余的 1 个人压力很大；如果是 1 个人，这就是单点了，团队没有备份，某些情况下是很危险的，假如这个人休假了，系统出问题了怎么办？  
最后，从技术提升的角度来讲，3 个人的技术小组既能够形成有效的讨论，又能够快速达成一致意见；如果是 2 个人，可能会出现互相坚持自己的意见，或者 2 个人经验都不足导致设计缺陷；如果是 1 个人，由于没有人跟他进行技术讨论，很可能陷入思维盲区导致重大问题；如果是 4 个人或者更多，可能有的参与的人员并没有认真参与，只是完成任务而已。  
“三个火枪手”的原则主要应用于微服务设计和开发阶段，如果微服务经过一段时间发展后已经比较稳定，处于维护期了，无须太多的开发，那么平均 1 个人维护 1 个微服务甚至几个微服务都可以。当然考虑到人员备份问题，每个微服务最好都安排 2 个人维护，每个人都可以维护多个微服务。

拆分策略有哪些？

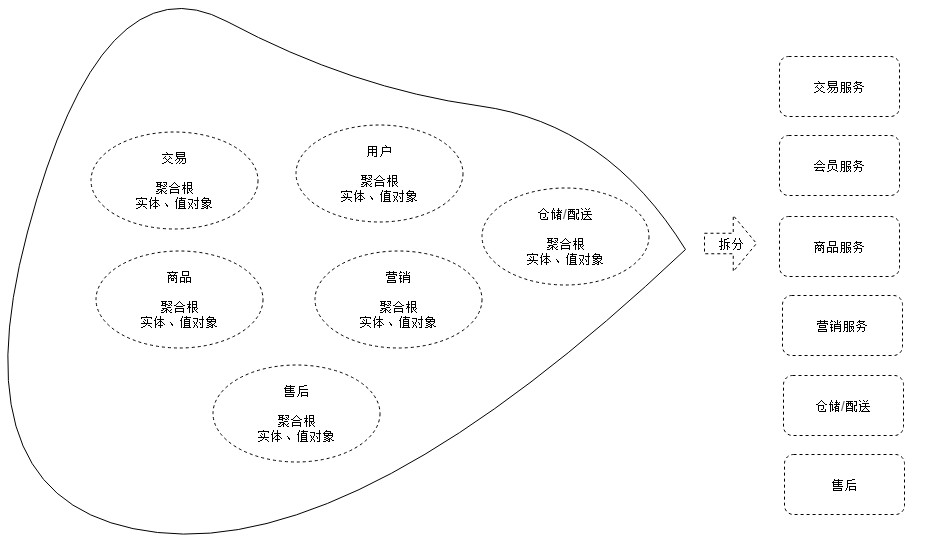
拆分策略可以按功能和非功能维度进行考虑，功能维度主要是划分清楚业务的边界，非功能维度主要考虑六点包括扩展性、复用性、高性能、高可用、安全性、异构性。接下来详细介绍下。

**功能维度**

功能维度主要是划分清楚业务边界，采用的主要设计方法可以利用 DDD（关于 DDD 的理论知识可以参考网上其它资料），DDD 的战略设计会建立领域模型，可以通过领域模型指导微服务的拆分，主要分四步进行：

* 第一步，找出领域实体和值对象等领域对象。
* 第二步，找出聚合根，根据实体、值对象与聚合根的依赖关系，建立聚合。
* 第三步，根据业务及语义边界等因素，定义限界上下文。
* 第四步，每一个限界上下文可以拆分为一个对应的微服务，但也要考虑一些非功能因素。

以电商的场景为例，交易链路划分的限界上下文如下图左半部分，根据一个限界上下文可以设计一个微服务，拆解出来的微服务如下图右侧部分。

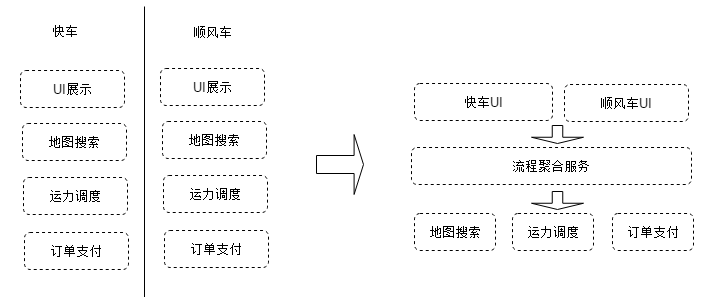


**非功能维度**

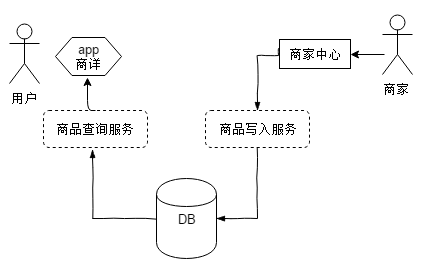
当我们按照功能维度进行拆分后，并不是就万事大吉了，大部分场景下，我们还需要加入其它维度进一步拆分，才能最终解决单体架构带来的问题。

* **扩展性：**区分系统中变与不变的部分，不变的部分一般是成熟的、通用的服务功能，变的部分一般是改动比较多、满足业务迭代扩展性需要的功能，我们可以将不变的部分拆分出来，作为共用的服务，将变的部分独立出来满足个性化扩展需要。同时根据二八原则，系统中经常变动的部分大约只占 20%，而剩下的 80% 基本不变或极少变化，这样的拆分也解决了发布频率过多而影响成熟服务稳定性的问题。

* **复用性：**不同的业务里或服务里经常会出现重复的功能，比如每个服务都有鉴权、限流、安全及日志监控等功能，可以将这些通过的功能拆分出来形成独立的服务，也就是微服务里面的 API 网关。在如，对于滴滴业务，有快车和顺风车业务，其中都涉及到了订单支付的功能，那么就可以将订单支付独立出来，作为通用服务服务好上层业务。如下图：



**高性能：**将性能要求高或者性能压力大的模块拆分出来，避免性能压力大的服务影响其它服务。常见的拆分方式和具体的性能瓶颈有关，例如电商的抢购，性能压力最大的是入口的排队功能，可以将排队功能独立为一个服务。同时，我们也可以基于读写分离来拆分，比如电商的商品信息，在 App 端主要是商详有大量的读取操作，但是写入端商家中心访问量确很少。因此可以对流量较大或较为核心的服务做读写分离，拆分为两个服务发布，一个负责读，另外一个负责写。还有数据一致性是另一个基于性能维度拆分需要考虑的点，对于强一致的数据，属于强耦合，尽量放在同一个服务中（但是有时会因为各种原因需要进行拆分，那就需要有响应的机制进行保证），弱一致性通常可以拆分为不同的服务。



* **高可用：**将可靠性要求高的核心服务和可靠性要求低的非核心服务拆分开来，然后重点保证核心服务的高可用。具体拆分的时候，核心服务可以是一个也可以是多个，只要最终的服务数量满足“三个火枪手”的原则就可以。比如针对商家服务，可以拆分一个核心服务一个非核心服务，核心服务供交易服务访问，非核心提供给商家中心访问。

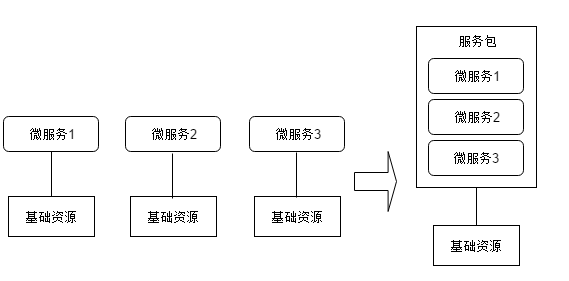
* **安全性：**不同的服务可能对信息安全有不同的要求，因此把需要高度安全的服务拆分出来，进行区别部署，比如设置特定的 DMZ 区域对服务进行分区部署，可以更有针对性地满足信息安全的要求，也可以降低对防火墙等安全设备吞吐量、并发性等方面的要求，降低成本，提高效率。

* **异构性：**对于对开发语言种类有要求的业务场景，可以用不同的语言将其功能独立出来实现一个独立服务。

以上几种拆分方式不是多选一，而是可以根据实际情况自由排列组合。**同时拆分不仅仅是架构上的调整，也意味着要在组织结构上做出相应的适应性优化，以确保拆分后的服务由相对独立的团队负责维护。**

服务都拆了为什么还要合并？

古希腊哲学家赫拉克利特曾经说过：“人不能两次踏进同一条河流。”随着时间的流逝，任何事物的状态都会发生变化。线上系统同样如此，即使一个系统在不同时刻的状况也绝不会一模一样。现在拆分出来的服务粒度也许合适，但谁能保证这个粒度能够一直正确呢。  
  
服务都拆了为什么还要合，就是要不断适应新的业务发展阶段，笔者这里做个类比看大家是否清晰，拆相当于我们开发代码，合相当于重构代码，为什么要重构呢，相信你肯定知道。微服务的合也是一样的道理，随着我们对应用程序领域的了解越来越深，它们可能会随着时间的推移而变化。例如，你可能会发现由于过多的进程间通信而导致特定的分解效率低下，导致你必须把一些服务组合在一起。  
同时因为人员和服务数量的不匹配，导致的维护成本增加，也是导致服务合并的一个重要原因。例如，今年疫情的影响导致很多企业开始大量裁员，人员流失但是服务的数量确没有变，造成服务数量和人员的不平衡，一个开发同学同时要维护至少 5 个服务的开发，效率大幅下降。  
那么如果微服务数量过多和资源不匹配，则可以考虑合并多个微服务到服务包，部署到一台服务器，这样可以节省服务运行时的基础资源消耗也降低了维护成本。需要注意的是，虽然服务包是运行在一个进程中，但是服务包内的服务依然要满足微服务定义，以便在未来某一天要重新拆开的时候可以很快就分离。服务合并到服务包示意图如下：



拆分过程中要注意的风险

**不打无准备之仗**

开发团队是否具备足够的经验，能否驾驭微服务的技术栈，可能是第一个需要考虑的点。这里并不是要求团队必须具备完善的经验才能启动服务拆分，如果团队中有这方面的专家固然是最好的。如果没有，那可能就需要事先进行充分的技术论证和预演，至少不打无准备之仗。避免哪个简单就先拆哪个，哪个新业务要上了，先起一个服务再说。否则可能在一些分布式常见的问题上会踩坑，比如服务器资源不够、运维困难、服务之间调用混乱、调用重试、超时机制、分布式事务等等。  
2  
**不断纠正**

我们需要承认我们的认知是有限的，只能基于目前的业务状态和有限的对未来的预测来制定出一个相对合适的拆分方案，而不是所谓的最优方案，任何方案都只能保证在当下提供了相对合适的粒度和划分原则，要时刻做好在未来的末一个时刻会变得不和时宜、需要再次调整的准备。因此随着业务的演进，需要我们重新审视服务的划分是否合理，如服务拆的太细，导致人员效率反而下降，故障的概率也大大增加，则需要重新划分好领域边界。  
  
3  
**要做行动派，而不是理论派**

在具体怎么拆分上，也不要太纠结于是否合适，不动手怎么知道合不合适呢？如果拆了之后发现真的不合适，在重新调整就好了。你可能会说，重新调整成本比较高。但实际上这个问题的本质是有没有针对服务化架构搭建起一套完成的能力体系，比如服务治理平台、数据迁移工具、数据双写等等，如果有的话，重新调整的成本是不会太高的。