运维加薪技术——微服务拆分规范

原创 运维自习室 运维自习室 2025年01月11日 16:44 上海

微信扫一扫
关注该公众号

一、微服务拆分规范

微服务拆分之后，工程会比较的多，如果没有一定的规范，将会非常混乱，难以维护。

1、高内聚、低耦合

紧密关联的事物应该放在一起，每个服务是针对一个**单一职责**的业务能力的封装，专注做好一件事情（每次只有一个更改它的理由)。如下图：有四个服务A、B、C、D，但是每个服务职责不单一，A可能在做B的事情，B又在做C的事情，C又同时在做A的事情，通过重新调整，将相关的事物放在一起后，可以减少不必要的服务。

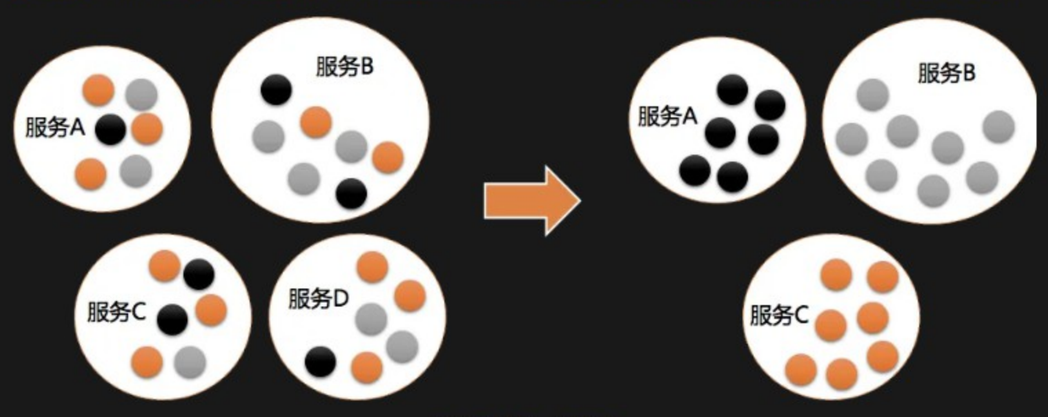

图1. 高内聚、低耦合

通用拆分方式：

1. 先按业务领域拆，如社区、用户、商城、慢病、工单，如果有相同功能需要聚合，则进行下沉（垂直）。
2. 再按功能定位拆（水平），如商城业务复杂度提高之后可进一步拆分为商品、订单、物流、支付。
3. 按重要程度拆，区分核心与非核心，如订单核心，订单非核心。

2、服务拆分正交性原则

两条直线相交成直角，就是正交的。正交也就是两条直线互不依赖。如果一个系统的变化不影响另一个系统这些系统就是正交的。

直接应用正交性原则，构建的系统的质量可以得到很大提高，可以让你的系统易于设计、开发、测试及扩展上线。提高开发效率，降低风险。

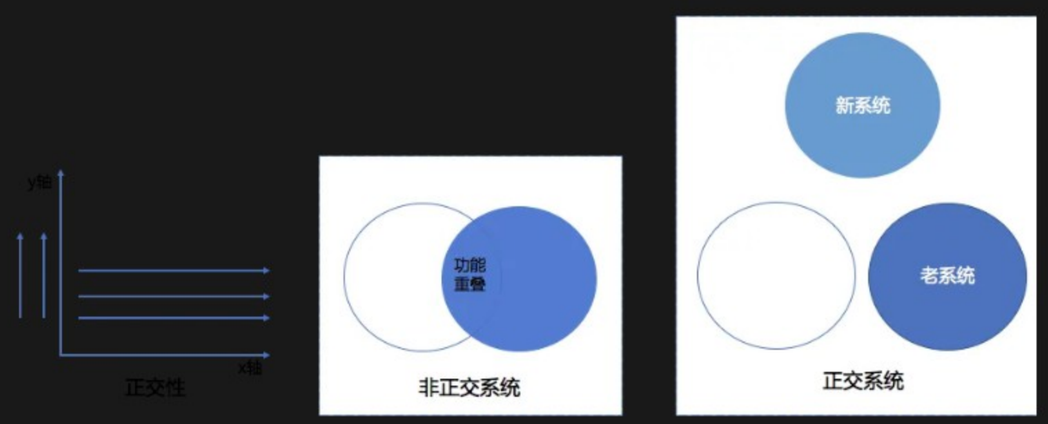

图2. 服务拆分正交性

3、服务拆分层级最多三层

服务拆分是为了横向扩展，因而应该横向拆分，而非纵向拆成一串的。也即应该将商品和订单拆分，而非下单的十个步骤拆分，然后一个调用一个。

人们经常问的一个问题是，服务拆分之后，原来都在一个进程里面的函数调用，现在变成了A调用B调用C调用D调用E，会不会因为调用链路过长而使得相应变慢呢？这里纵向的拆分最多三层，大部分情况下只有两次调用，具体请参考《微服务分层模型》。

4、服务粒度适中、演进式拆分

微服务拆分并不是一步到位的，应当根据实际情况逐步展开。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。

此外，一个微服务需要足够简单，站在微服务角度而言往往只需要1~2人左右可方便快速维护。如果维护的人员过多，要么这个服务过于复杂成为了单体应用；要么是服务边界划分得不够明确；要么是人员组织架构的职责不清。

5、避免环形依赖、双向依赖

服务之间的环形/双向依赖会使得服务间耦合加重，在服务升级的的时候会比较头疼，不知道应该先升级哪个，后升级哪个，难以维护。因此我们需要极力避免服务间的这种复杂依赖关系。

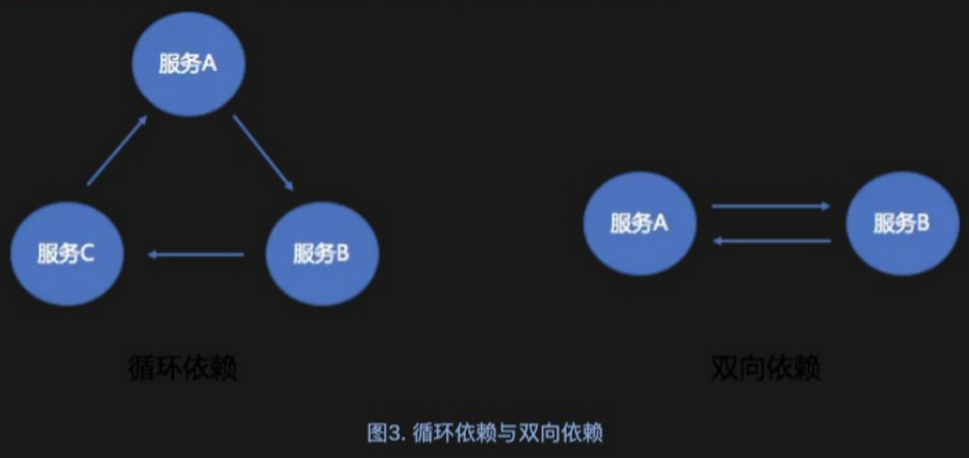

图3. 循环依赖与双向依赖

解决这种复杂依赖关系，我们有两种服务拆分避免方式，根据业务场景合理选择：

- 将共同依赖的服务功能进行下沉，拆分为第三方服务。还要注意下沉的服务维护方需要多方协商确定维护者。
- 如果在业务流程上允许异步解耦的，可以考虑引入消息中间件来处理，消息中间件也是微服务治理中使用较多的核心组件。

6、通用化接口设计，减少定制化设计

提微服务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服务提供一个获取订单列表的接口，那么A服务设计的接口就应该是 `GetOrderList()` ，而不是 `GetOrderListForA()` 。

7、接口设计需要严格保证兼容性

为了保证每个微服务能够独立发布，并且降低发布的兼容性风险，那么接口需要严格保证兼容性。

8、将串行调用改为并行调用，或者异步化

如果有的组合服务处理流程的确很长，需要调用多个外部服务，应该考虑如何通过消息队列，实现异步化和解耦。

例如下单之后，要刷新缓存，要通知仓库等，这些都不需要再下单成功的时候就要做完，而是可以发一个消息给消息队列，异步通知其他服务。

而且使用消息队列的好处是，你只要发送一个消息，无论下游依赖方有一个，还是有十个，都是一条消息搞定，只不过多几个下游监听消息即可。

对于下单必须同时做完的，例如扣减库存和优惠券等，可以进行**并行调用**，这样处理时间会大大缩短，不是多次调用的时间之和，而是最长的那个系统调用时间。

9、接口应该实现幂等性

微服务拆分之后，服务之间的调用当出现错误的时候，往往都会重试，但是为了不要下两次单，支付两次，微服务接口应当实现幂等性。

幂等操作使用状态机，当一个调用到来的时候，往往触发一个状态的变化，当下次调用到来的时候，发现已经不是这个状态，就说明上次已经调用过了。状态的变化需要是一个原子操作，也即并发调用的时候，只有一次可以执行。

10、接口数据定义严禁内嵌，透传

微服务接口之间传递数据，往往通过数据结构，如果数据结构透传，从底层一直到上层使用同一个数据结构，或者上层的数据结构内嵌底层的数据结构，当数据结构中添加或者删除一个字段的时候，波及的面会非常大。

因而接口数据定义，在每两个接口之间约定，严禁内嵌和透传，即便差不多，也应该重新定义，这样接口数据定义的改变，影响面仅仅在调用方和被调用方，当接口需要更新的时候，比较可控，也容易升级。

11、避免服务间共享数据库

数据库包括任意的数据存储服务，例如： `MySQL` 、 `Redis` 、 `MongoDB` 等。如果服务间共享数据库，会造成：

1. **强耦合**：为多个服务提供单个数据库会造成服务间紧密耦合，也会造成服务独立部署困难。
2. **扩展性差**：使用这种设计很难扩展单个服务，因为这样只能选择扩展整个单个数据库。
3. **性能问题**：使用一个共享数据库（不是数据库服务器），在一段时间内，你可能最终会得到一个巨大的表。

因此，对于现有共享庞大数据库的微服务，建议是按照业务维度拆分成多个小的数据库，分开独立维护。**此外，再次提醒，禁止跨库联表查询。**

12、同时应当考虑团队结构

前面讲的都是技术因素的划分原则，其实微服务拆分时也应当考虑团队组织结构。

1. 团队足够轻量级，2 pizza原则，保证团队内部能够高效沟通。
2. 团队的职责足够明确，保证能够独立维护，减少团队间工作耦合度，降低跨团队协作成本。

二、微服务拆分时机

微服务拆分绝非一个大跃进运动，由高层发起，把一个应用拆分的七零八落的，最终大大增加运维成本，但是并不会带来收益。**微服务拆分的过程，应该是一个由痛点驱动的**，是业务真正遇到了快速迭代和高并发的问题，如果不拆分，将对于业务的发展带来影响，只有这个时候，微服务的拆分是有确定收益的，增加的运维成本才是值得的。

1、快速迭代

使用微服务架构的目的就是为了快速迭代，快速上线，这也是微服务架构的最大特点。

2、高并发、性能要求

对于微服务化拆分后的服务，可以轻松地进行水平扩容，进行服务优化，满足更多的并发和性能需求。

在高并发场景下（或者资源紧张的场景下），我们希望一个请求如果不成功，不要占用资源，应该尽快失败，尽快返回，而且希望当一些边角的业务不正常的情况下，主要业务流程不受影响。这就需要熔断策略，也即当A调用B，而B总是不正常的时候，为了让B不要波及到A，可以对B的调用进行熔断，也即A不调用B，而是返回暂时的fallback数据，当B正常的时候，再放开熔断，进行正常的调用。

如果核心业务流程和普通业务流程在同一个服务中，就需要使用大量的 `if-else` 语句，根据下发的配置来判断是否熔断或者降级，这会使得配置异常复杂，难以维护。如果核心业务和普通业务分成两个服务，就可以使用标准的熔断降级策略，配置在某种情况下，放弃对另一个服务的调用，可以进行统一的维护。

3、提交代码频繁出现大量冲突

微服务对于快速迭代的效果，首先是开发独立，如果是一单体应用，几十号人开发一个模块，如果使用GIT做代码管理，则经常会遇到的事情就是代码提交冲突。同样一个模块，你也改，他也改，几十号人根本没办法沟通。所以当你想提交一个代码的时候，发现和别人提交的冲突了，于是因为你是后提交的人，你有责任去merge代码，好不容易merge成功了，等再次提交的时候，发现又冲突了，你是不是很恼火。随着团队规模越大，冲突概率越大。

所以应该拆分成不同的模块，比如每十个人左右维护一个模块，也即一个工程，首先代码冲突的概率小多了，而且有了冲突，一个小组一吼，基本上问题就解决了。每个模块对外提供接口，其他依赖模块可以不用关注具体的实现细节，只需要保证接口正确就可以。

4、小功能要积累到大版本才能上线

微服务对于快速迭代的效果，首先是上线独立。如果没有拆分微服务，每次上线都是一件很痛苦的事情。当你修改了一个边角的小功能，但是你不敢马上上线，因为你依赖的其他模块才开发了一半，你要等他，等他好了，也不敢马上上线，因为另一个被依赖的模块也开发了一半，当所有的模块都耦合在一起，互相依赖，谁也没办法独立上线，而是需要协调各个团队，大家开大会，约定一个时间点，无论大小功能，死活都要这天上线。

这种模式导致上线的时候，单次上线的需求列表非常长，这样风险比较大，可能小功能的错误会导致大功能的上线不正常，将如此长的功能，需要一点点check，非常小心，这样上线时间长，影响范围大。

Captured by FireShot Pro: 14 2月 2025, 14:00:49
https://getfireshot.com

服务拆分后，**在团队职责明确、应用边界明确、接口稳定的情况下**，不同的模块可以独立上线。这样上线的次数增多，单次上线的需求列表变小，可以随时回滚，风险变小，时间变短，影响面小，从而迭代速度加快。对于接口要升级部分，保证灰度，先做接口新增，而非原接口变更，当注册中心中监控到的调用情况，发现接口已经不用了，再删除。