19 |如何做好微服务间依赖的治理和分布式事务?

在前两讲里，分别从微服务的对外接口、消息消费以及微服务自身的相关编码规范上阐述了“**防备上游、做好自己**”这两个准则如何落地。

在本讲里，将会讲解为什么要“**怀疑下游**”，以及有哪些手段可以落地此条准则。此外，还会介绍在进行微服务拆分后，调用外部依赖会产生的分布式事务、消息发送等问题的应对方案。

为什么要怀疑下游

首先我们先来回顾一下“第 17 讲”里介绍过的抽象的微服务架构，如下图 1 所示：

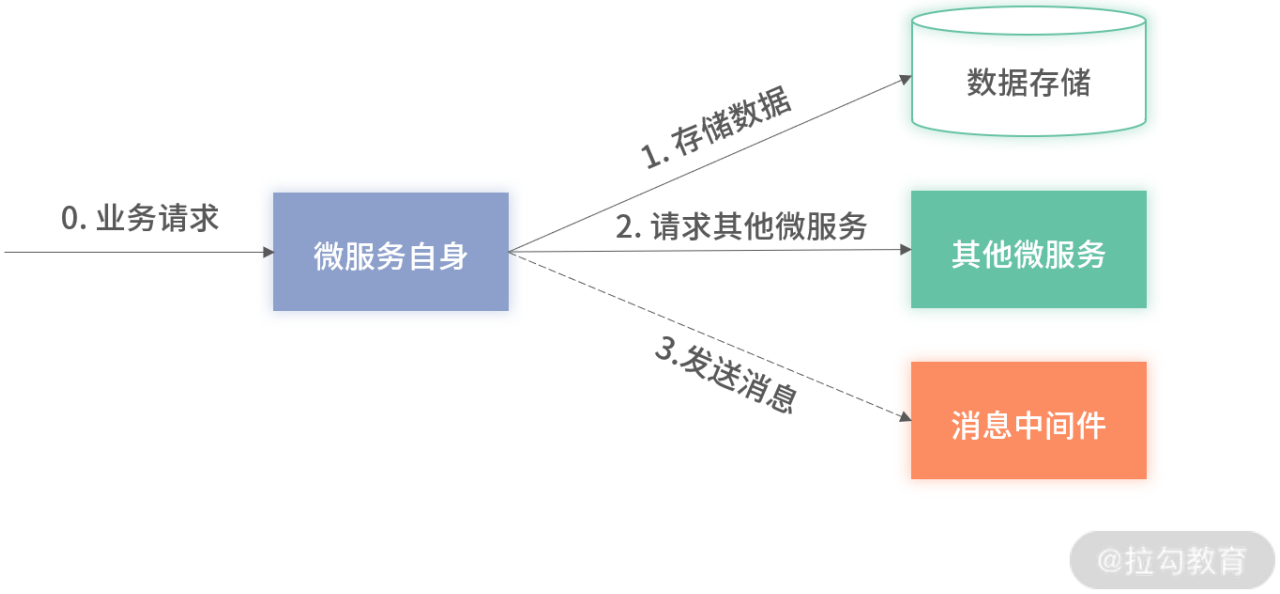

图 1：抽象的架构示意图

从图一中可以看到，微服务会依赖很多其他微服务提供的接口、数据库、缓存，以及消息中间件等，这些接口及存储可能会因为代码 Bug、网络、磁盘故障、上线操作失误等因素引发线上问题。此时，由于依赖不可用，就会导致微服务对外提供的服务受到影响，出现接口可用率下降或者直接宕机的情况。

为了防止上述情况的发生，在构建微服务时，就需要预先考虑微服务所依赖的各项“下游”出现故障时的应对方案。假设下游出现故障及预设计对应的方案的过程，便是在实践“**怀疑下游**”。

如何落地

下面将基于下图 2 所展示的三大类依赖：其他微服务、数据库、消息中间件，逐一介绍可能引发的故障的应对方案和最佳使用准则。

#后期同学美化下图，注意将“其它”改为“其他”

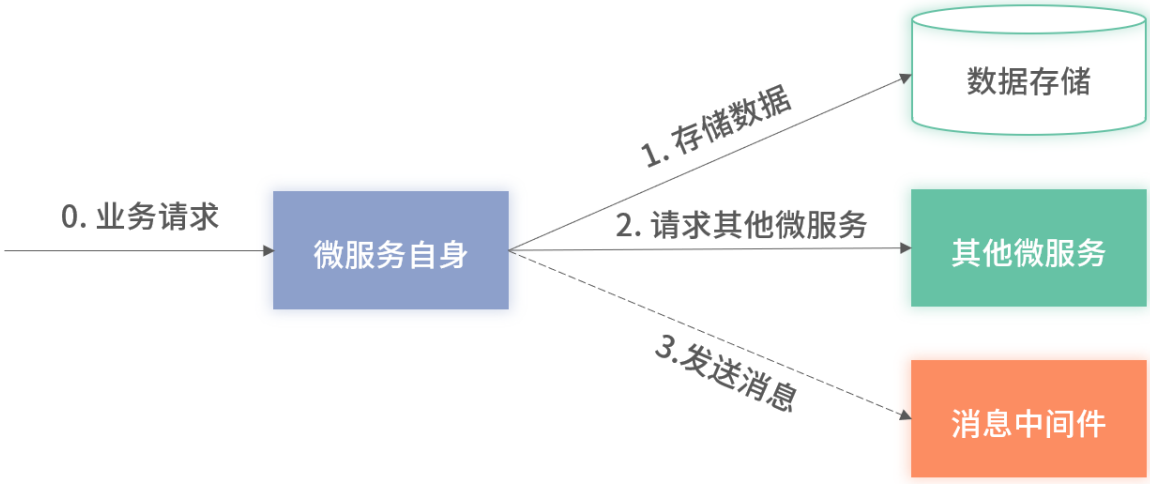

图 2：包含三大类依赖的微服务图

对其他微服务的依赖

在采用了微服务的架构后，各个模块间均通过 RPC 的方式进行依赖，有些模块在完成一项业务流程时可能会依赖多达几十、上百个外部微服务。比如在完成下单的流程里，就需要依赖用户、商品、促销、价格、优惠券等各个微服务提供的接口，这些被依赖的微服务的稳定性直接影响了用户是否能够成功下单。因此，需要对微服务依赖的其他微服务接口进行可用性的治理。

在“第 10 讲”里，已经**从写服务的角度介绍了通过依赖后置、依赖并行化、设置超时和重试、服务降级等手段，来对它的依赖进行治理，进而保障写服务的高可用**。其实这些手段依然可以用在读服务里，此处便不再赘述，你可以回到“第 10 讲”进行复习。

下面将重点讲解在采用微服务架构后，如何应对随之而来的分布式事务。这里以**提单**作为案例，介绍分布式事务的实际场景。在微服务架构下，订单和库存是两个单独的微服务，它们之间的架构如下图 3 所示：

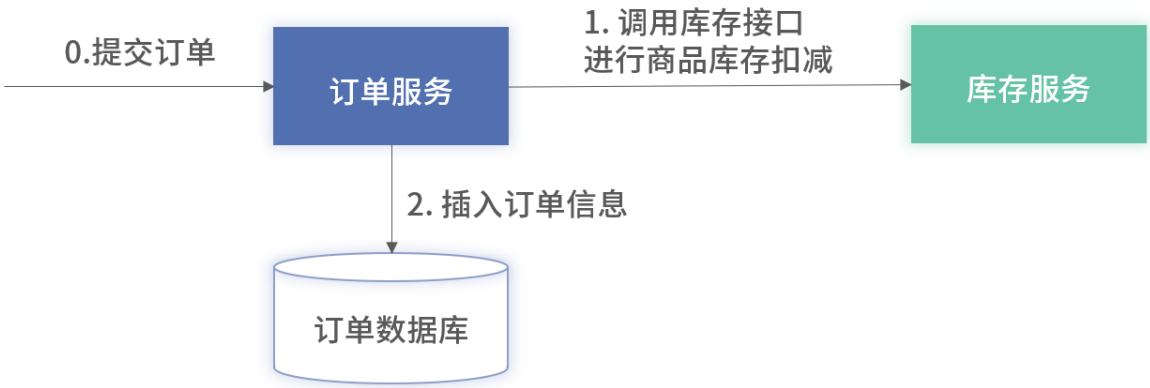

图 3：订单和库存组成的微服务架构图

在提单时，订单模块需要调用库存模块进行商品的扣减，以便判断用户购买的商品是否有货。订单调用库存的扣减接口会有以下几种情况发生。

1. 调用库存接口返回成功且库存数量充足，订单模块便将此用户订单保存至数据库，并返回用户下单成功消息。

2. 调用库存接口返回成功且库存数量充足，但订单模块将此用户订单保存至数据库时出错并进行数据库回滚，同时订单模块返回用户下单失败。

3. 调用库存接口超时，订单模块判断此次调用库存接口失败，返回用户下单失败。

4. ...

在微服务化之后，上述订单模块和库存模块的交互会产生非常多的可能性场景。此处我只罗列了几个，你可以继续向后梳理。其中，上述的第 2 、3 点描述的场景里就存在**分布式事务问题**。在第 2 点里，因为订单模块本地的数据库事务回滚了，但调用库存接口产生的已扣减的商品数量并没有回滚，此时就会导致库存数据少于实际的数据。

有一些基于 TCC 和 Saga 的成熟基础框架可以解决上述分布式事务问题，但理解和接入成本较高。此处介绍一种本质上和 TCC、Saga 理论相类似，但无须借助第三方框架的简单、易落地的解决方案。理解此方案也有助于你理解 TCC 和 Saga 的思想。

此方案的架构图如下图 4 所示，图中订单模块的数据库里除了订单原有的表之外，会增加一张任务表。

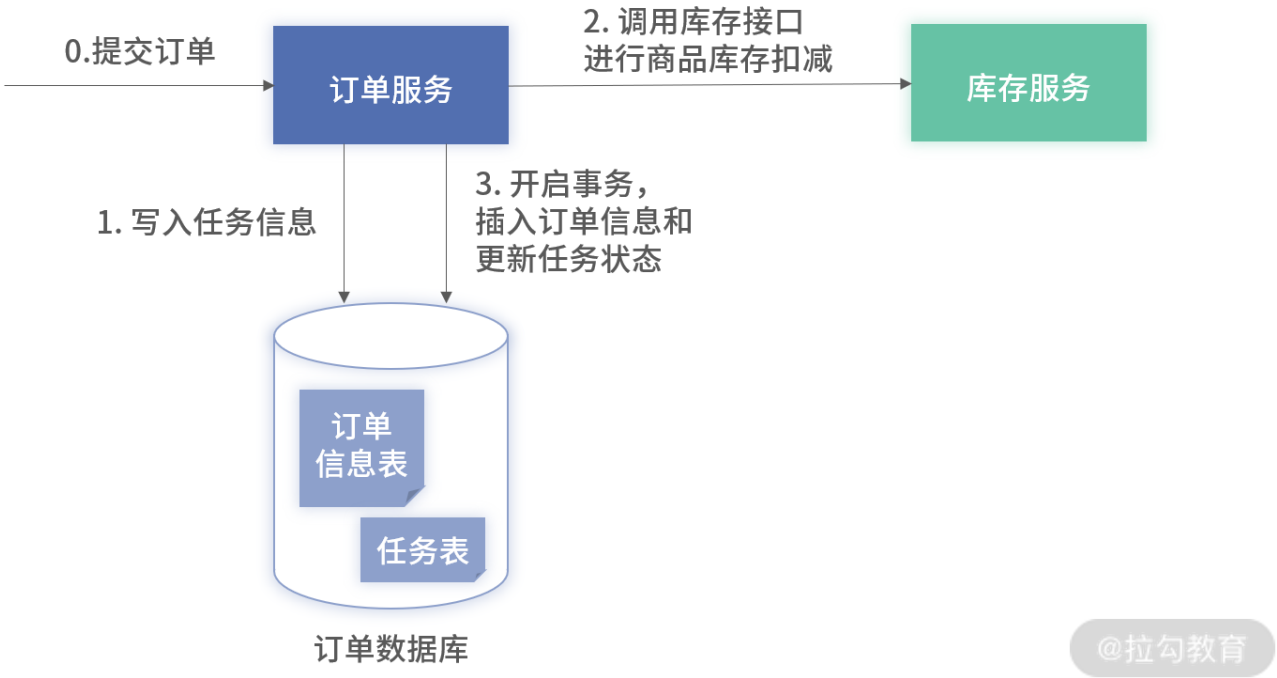

图 4：基于本地数据库的分布式事务架构

基于上述的架构，下单流程变更如下。

1. 在接收到下单请求后，在调用任何外部 RPC 前，先将此订单的相关信息，如此次用户购买的商品、商品数量、用户账号、此次订单的编号等信息写入新增的任务表中。

2. 调用库存的接口进行商品数量的扣减，并根据库存模块的返回值更新订单模块的数据库。这一步，又细分为以下几种场景情况：

（1）如果调用库存接口**成功**，则在同一个事务中，将订单信息写入订单库中，同时更新第一步写入任务的状态为“**已成功**”；

（2）如果调用库存接口明确返回**失败**，则直接更新订单库中的任务状态为“**待回滚**”，并返回用户下单失败；

（3）如果调用库存接口**超时**，则直接更新订单库中的任务状态为“**待回滚**”，并返回用户下单失败；

（4）无论调用库存接口**是成功还是失败**，只要在更新本地订单库时失败，就返回用户下单失败，同时任务库的状态保留为“**初始化**”。

上述介绍的是用户下单的同步流程，完成这两个步骤后，用户下单便结束了。我们再来看看下单后的异步情况。

3. 下单完成后，异步 Worker 功能是扫描订单库新增的任务表，获取状态为“**待回滚**”，任务创建时间距扫描时间点超过一定时间区间（如 5 分钟）仍为“初始化”状态的任务。获取到这些任务之后，会基于任务表中的商品和对应的数量信息，异步地调用库存接口进行商品数据的返还。

通过上述方式，能够将各种失败场景里漏返回的商品数量进行返还，保证库存数量的最终一致性，完成分布式事务。**上**述保障数据最终一致性主要是依赖任务表和订单表在同一个数据库里，可以通过本地事务来保障订单表数据写入成功后，任务表里的任务状态绝对能够更新为“**已成功**”。而当提单失败后，任务表的状态为“**非成功**”状态，再通过类似 TCC 和 Saga 的异步补偿性 Worker 来进行业务回滚即可保证最终最一致性。

在发起分布式事务的业务模块的数据库里创建补偿性任务，基本上可以复用在其他存在分布式事务的场景里。如果你不希望引入更加复杂的 TCC 和 Saga 框架，可以尝试利用此方式来解决架构微服务化之后带来的分布式事务的问题。

对数据库的依赖

除了对其他微服务的依赖，微服务中最常见的便是对数据库的依赖。在使用时，需要遵守以下几点基本原则。

原则一：数据库一定要配置从库，且从库部署的机房需要与主库不同，从而保障数据库具备跨机房灾备的能力。

此外，对于测试环境的数据库依然要配置主从复制，防止某天测试环境的数据库磁盘损坏，需要耗费大量人力恢复测试环境。

原则二：在能够完成功能的前提下，使用的 SQL 要尽可能简单。

因为 SQL 和代码一样，除了完成功能之外，最重要的是清晰简单地表达其自身含义，以供后续研发人员进行维护。我曾经在线上遇到过为了不使用唯一索引，纯使用 SQL 来完成防重的语句，它包含了四层 insert、select、exists、select 的语法嵌套。这一语句因为无法调试（Debug），导致后续一个需求的上线时间延期了 2 天，最终还是痛定思痛地进行了重构。

原则三：在业务需求不断更新迭代的场景里，最好不要使用外键。

大学时期的数据库理论课曾提到，需要使用外键来校验数据完整性。比如，在 A、B 表之间有了外键约束之后，可以设置外键级联删除，当 A 表中的某条数据删除后，自动级联地删除 B 表中的数据。此方式表面上可以极大地简化代码操作，但实则隐藏着巨大风险。因为现今互联网需求的迭代速度非常快，上个月可能 A、B 表中还存在外键关系，到了下个月又因为需求不存在了，或者需要更多字段组合才能形成外键关系。

此外，外键关系是隐藏在数据库的建表语句里的，在新需求开发时，很容易被遗忘、清除或者修改为新的外键关系。在新需求上线后，也可能因此疏漏导致线上数据被误删，进而引发线上问题。

原则四：表结构中尽可能不要创建一个长度为上千或上万的 varchar 类型字段，且用其来存储类似 JSON 格式的数据，因为这会带来并发更新的问题。

假设创建了一个长度一千的 varchar 字段，它存储了如下的信息：

```
{"fieldA":"valueA","fieldB":"valueB"}
```

此时假设有两个请求**同时**对此字段进行修改，A 线程将此字段的值读取后修改了其中 filedA 的值，具体修改如下：

```
{"fieldA":"valueAA",:"fieldB":"valueB"}
```

而 B 线程将此字段的值读取后修改了其中 fieldB 的值，具体修改如下：

```
{"fieldA":"valueA",:"fieldB":"valueBB"}
```

那么，最终数据库中此字段的值会变成什么呢?

答案是不一定。这取决于 A、B 这两个线程的最终修改顺序。但不管顺序如何，最终的结果都是错误的。因为 A、B 两个线程各修改了JSON 内容的其中一个字段，最终期望的结果是 fieldA、fieldB 两个字段都得到更新，但实际只会有一个字段得到更新。

因此，在创建表结构的时候，不建议设置此类型的字段。如果期望这两个字段都得到更新，你需要借助**并发锁**来实现，但这也增加了代码实现的难度。

对消息中间件的依赖

在微服务的架构里，微服务间的通信除了接口调用的方式外，当前最常见的方式便是基于消息中间件（如 RabbitMQ 和 Kafka）的消息通信。同样，在使用消息中间件时，仍有一些基础原则需要你尽可能地遵守。

原则一：数据要先写入数据库或缓存后，再发送消息通知。

因为很多消息接收方在接收到消息通知后，会调用发送消息的微服务的接口进行数据反查，以便获取更多信息来做下一步业务的流转。

假设订单模块在判断用户的下单请求的库存能够满足后， 便向外发送下单成功的消息。此时，如果物流系统监听了此消息，就会在获取到下单成功的通知后，第一时间去反查订单的接口，以便获取更多订单相关信息（如用户期望的收货时间、用户是否为会员等）来辅助判断何时发货。在极端情况下，可能会因为订单模块的数据还未写入数据库，导致反查不到数据，进而影响业务的正常流转。

原则二：发送的消息要有版本号。

有些消息中间件为了提升消息消费的吞吐量，支持乱序消费。但如果发送的消息没有数据变更版本号，消息消费方会因此无法判断数据是否乱序，进而有可能导致数据错乱，产生线上问题。

原则三：消息的数据要尽可能全，进而减少消息消费方的反查。

微服务间使用消息通信的目的就是解耦，但如果消息中包含的信息量太少，消息消费方就无法基于其中的信息处理业务，此时消息消费方便需要反查发送方的接口，来获取更多信息，但这样处理就达不到解耦的目的了，你可以参考第一点物流系统的案例。因此，在可能的情况下，建议发送尽可能全的信息。

原则四：消息中需要包含标记某个字段是否变更的标识。

根据原则三，你可能会发送包含较多字段的消息，有些字段可能在当次消息中并未发生数据变更。如果没有标记字段是否变更，可能会产生无效通知的情况。

比如一个消息包含两个字段（如为 A、B），而某一个消息的接收方（如用户模块）只关心 A 字段是否变更。如果没有标记变更字段，那么 B 字段变更后，消息发送方也会发送消息，这会导致“用户模块”误以为 A 字段发生了变更，进而触发“用户模块”执行一次本不应该执行的业务流程。

总结

本讲介绍了采用微服务架构后，不可避免的分布式事务的解决方案，同时介绍了微服务常见的依赖：数据库、消息中间件的基本治理原则。后续你可以将本讲学习到的内容应用到你所负责的微服务的依赖治理中去。

最后，我再给你留一道讨论题，你所负责的微服务对于它的依赖的使用，有哪些基本原则？欢迎留言区留言，咱们一起讨论。

这一讲就到这里，感谢你学习本次课程，接下来我们将学习20 | 如何通过监控快速发现问题。再见。