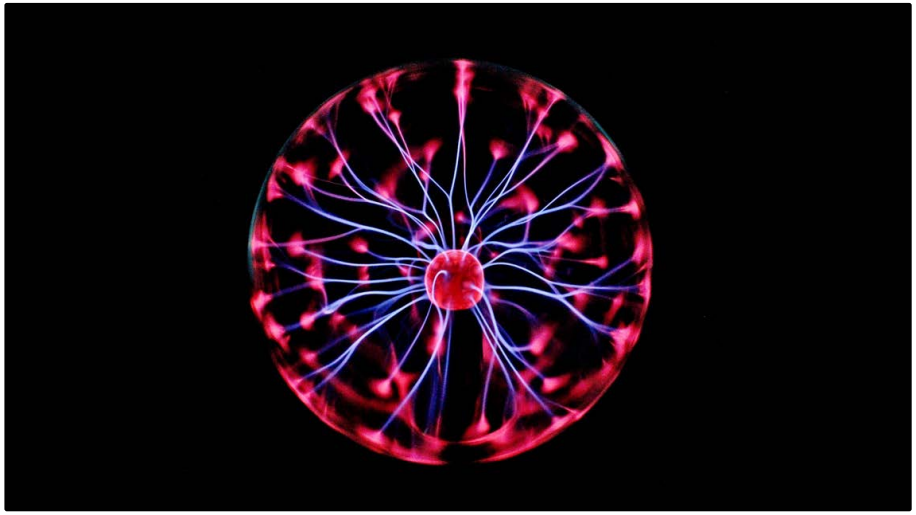
弹力设计篇之“补偿事务”

2018-03-08 陈皓



弹力设计篇之“补偿事务”

陈皓

- 00:00 / 12:39

前面，我们说过，分布式系统有一个比较明显的问题就是，一个业务流程需要组合一组服务。这样的事情在微服务下就更为明显了，因为这需要业务上的一致性的保证。也就是说，如果一个步骤失败了，那么要么回滚到以前的服务调用，要么不断重试保证所有的步骤都成功。

这里，如果需要强一性的需求，那么，在业务层上需要使用“两阶段提交”这样的方式。但是好在我们的很多情况下并不需要这么强的一致性，而且强一致性的最佳保证最好是在底层完成。或是像之前说的那样Stateful的Sticky Session那样在一台机器上完成。在我们接触到的大多数业务，其实只需要最终一致性就好。

ACID 和 BASE

谈到这里，有必要先说一下ACID和BASE的差别。传统关系型数据库系统的事务都有ACID属性，即原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability）。

- 原子性：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
- 一致性：在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。
- 隔离性：两个事务的执行是互不干扰的，一个事务不可能看到其他事务运行时，中间某一时刻的数据。两个事务不会发生交互。
- 持久性：在事务完成以后，该事务对数据库所做的更改便持久地保存在数据库之中，并不会被回滚。

事务的ACID属性保证了数据库的一致性，比如银行系统中，转账就是一个事务，从原账户扣除金额，以及向目标账户添加金额，这两个数据库操作的总和构成一个完整的逻辑过程，是不可拆分的原子操作，从而保证了整个系统中的总金额没有变化。

然而，这对于我们的分布式系统来说，尤其是微服务来说，这样的方式是很难适合高性能的要求的。我们都很熟悉CAP理论——在分布式的服务架构中，一致性（Consistency）、可用性（Availability）、分区容忍性（Partition Tolerance），在现实中不能都满足，最多只能满足其中两个。

所以，为了提高性能，出现了ACID的一个变种BASE。

- **Basic Availability**：基本可用。这意味着，系统可以出现暂时不可用的状态，而后面会快速恢复。
- **Soft-state**：软状态。它是我们前面的“有状态”和“无状态”的服务的一种中间状态。也就是说，为了提高性能，我们可以让服务暂时保存一些状态或数据，这些状态和数据不是强一致性的。
- **Eventual Consistency**：最终一致性，系统在一个短暂的时间段内是不一致的，但最终整个系统看到的数据是一致的。

可以看到，BASE系统是允许或是容忍系统出现暂时性的问题的，这样一来，我们的系统就能更有弹力。因为我们知道，在分布式系统的世界里，故障是不可避免的，我们能做的就是把故障处理当成功能写入代码中，这就是Design for Failure。

BASE的系统倾向于设计出更加有弹力的的系统，这种系统的设计特点是，要保证在短时间内，就算是有数据不同步的风险，我们也应该允许新的交易可以发生，而后面我们在业务上将可能出现问题的事务给处理掉，以保证最终的一致性。

举个例子，网上卖书的场景。

ACID的玩法就是，大家在买同一本书的过程中，每个用户的购买请求都需要把库存锁住，等减完库存后，把锁释放出来，后续的人才能进行购买。于是，在ACID的玩法下，我们在同一时间不可能有多个用户下单，我们的订单流程需要有排队的情况，这样一来，我们就不可能做出性能比较高的系统来。

BASE的玩法是，大家都可以同时下单，这个时候不需要去真正地分配库存，然后系统异步地处理订单，而且是批量的处理。因为下单的时候没有真正去扣减库存，所以，有可能会有超卖的情况。而后台的系统会异步地处理订单时，发现库存没有了，于是才会告诉用户你没有购买成功。

BASE这种玩法，其实就是亚马逊的玩法，因为要根据用户的地址去不同的仓库查看库存，这个操作非常耗时，所以，不想做成异步的都不行。

在亚马逊上买东西，你会收到一封邮件说，系统收到你的订单了，然后过一会儿你会收到你的订单被确认的邮件，这时候才是真正地分配了库存。所以，有某些时候，你会遇到你先收到了下单的邮件，过一会又收到了没有库存的致歉的邮件。

有趣的是，ACID的意思是酸，而BASE却是碱的意思，因此这是一个对立的东西。其实，从本质上来讲，酸（ACID）强调的是一致性（CAP中的C），而碱（BASE）强调的是可用性（CAP中的A）。

业务补偿

有了上面对ACID和BASE的分析，我们知道，在很多情况下，我们是无法做到强一致的ACID的。特别是我们需要跨多个系统的时候，而且这些系统还不是由一个公司所提供的。比如，在我们的日常生活中，我们经常会遇到这样的情况，就是要找很多方协调很多事，而且要保证我们每一件事都成功，否则整件事就做不到。

比如，要出门旅游， 我们需要干这么几件事。第一，向公司请假，拿到相应的假期；第二，订飞机票或是火车票；第三，订酒店；第四，租车。这四件事中，前三件必需完全成功，我们才能出行，而第四件事只是一个锦上添花的事，但第四件事一旦确定，那么也会成为整个事务的一部分。这些事都是要向不同的组织或系统请求。我们可以并行地做这些事，而如果某个事有变化，其它的事都会跟着出现一些变化。

设想下面的几种情况。

1. 我没有订到返程机票，那么我就去不了了。我需要把订到的去程机票，酒店、租到的车都给取消了，并且把请的假也取消了。
2. 如果我假也请好了，机票，酒店也订好了，只是车没租到，那么并不影响我出行这个事，整个事还是可以继续的。
3. 如果我的飞机因为天气原因取消或是晚点了，那么我被迫要去调整和修改我的酒店预订和租车的预订。

从人类的实际生活当中，我们可以看出，上述的这些情况都是天天在发生的事情。所以，我们的分布式系统也是一样的，也是需要处理这样的事情——就是当条件不满足，或是有变化的时候，需要从业务上做相应的整体事务的补偿。

一般来说，业务的事务补偿都是需要一个工作流引擎的。亚马逊是一个超级喜欢工作流引擎的公司，这个工作流引擎把各式各样的服务给串联在一起，并在工作流上做相应的业务补偿，整个过程设计成为最终一致性的。

对于业务补偿来说，首先需要将服务做成幂等性的，如果一个事务失败了或是超时了，我们需要不断地重试，努力地达到最终我们想要的状态。然后，如果我们不能达到这个我们想要的状态，我们需要把整个状态恢复到之前的状态。另外，如果有变化的请求，我们需要启动整个事务的业务更新机制。

所以，一个好的业务补偿机制需要做到下面这几点。

1. 要能清楚地描述出要达到什么样的状态（比如：请假、机票、酒店这三个都必须成功，租车是可选的），以及如果其中的条件不满足，那么，我们要回退到哪一个状态。这就是所谓的整个业务的起始状态定义。
2. 当整条业务跑起来的时候，我们可以串行或并行地做这些事。对于旅游订票是可以并行的，但是对于网购流程（下单、支付、送货）是不能并行的。总之，我们的系统需要努力地通过一系列的操作达到一个我们想要的状态。如果达不到，就需要通过补偿机制回滚到之前的状态。这就是所谓的状态拟合。
3. 对于已经完成的事务进行整体修改。其实可以考虑成一个修改事务。

其实，在纯技术的世界里也有这样的事。比如，线上运维系统需要发布一个新的服务或是对一个已有的服务进行水平扩展，我们需要先找到相应的机器，然后初始化环境，再部署上应用，再做相应的健康检查，最后接入流量。这一系列的动作都要完全成功，所以，我们的部署系统就需要管理好整个过程和相关的运行状态。

业务补偿的设计重点

业务补偿主要做两件事。

1. 努力地把一个业务流程执行完成。
2. 如果执行不下去，需要启动补偿机制，回滚业务流程。

所以，下面是几个重点。

- 因为要把一个业务流程执行完成，需要这个流程中所涉及的服务方支持幂等性。并且在上游有重试机制。
- 我们需要小心维护和监控整个过程的状态，所以，千万不要把这些状态放到不同的组件中，最好是一个业务流程的控制方来做这个事，也就是一个工作流引擎。所以，这个工作流引擎是需要高可用和稳定的。这就好像旅行代理机构一样，我们把需求告诉它，它会帮我们搞定所有的事。如果有问题，也会帮我们回滚和补偿的。
- 补偿的业务逻辑和流程不一定非得是严格反向操作。有时候可以并行，有时候，可能会更简单。总之，设计业务正向流程的时候，也需要设计业务的反向补偿流程。
- 我们要清楚地知道，业务补偿的业务逻辑是强业务相关的，很难做成通用的。
- 下层的业务方最好提供短期的资源预留机制。就像电商中的把货品的库存预先占住等待用户在15分钟内支付。如果没有收到用户的支付，则释放库存。然后回滚到之前的下单操作，等待用户重新下单。

小结

好了，我们来总结一下今天分享的主要内容。首先，我介绍了ACID和BASE两种不同级别的一致性。在分布式系统中，ACID有更强的一致性，但可伸缩性非常差，仅在必要时使用；BASE的一致性较弱，但有很好的可伸缩性，还可以异步批量处理；大多数分布式事务适合BASE。

要实现BASE事务，需要实现补偿逻辑，因为事务可能失败，此时需要协调各方进行撤销。补偿的各个步骤可以根据具体业务来确定是串行还是并行。由于补偿事务是和业务强相关的，所以必须实现在业务逻辑里。下篇文章中，我们讲述重试设计。希望对你有帮助。

也欢迎你分享一下你的分布式服务用到了怎样的一致性？你是怎么实现补偿事务的？

文末给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。

- 弹力设计篇
  - 认识故障和弹力设计
  - 隔离设计Bulkheads
  - 异步通讯设计Asynchronous
  - 幂等性设计Idempotency
  - 服务的状态State
  - 补偿事务Compensating Transaction
  - 重试设计Retry
  - 熔断设计Circuit Breaker
  - 限流设计Throttle
  - 降级设计degradation

- 弹力设计总结

- 管理设计篇
  - 分布式锁Distributed Lock
  - 配置中心Configuration Management
  - 边车模式Sidecar
  - 服务网格Service Mesh
  - 网关模式Gateway
  - 部署升级策略
- 性能设计篇
  - 缓存Cache
  - 异步处理Asynchronous
  - 数据库扩展
  - 秒杀Flash Sales
  - 边缘计算Edge Computing

林超

2018-03-15

期待讲解下工作流引擎的实现

李达

2018-03-09

陈老师好，有个疑问想请教：衡量高可用一般用几个9来衡量，例如4个9的高可用是指一年的服务不可用时间不能超过53分钟，我的疑问有两个：第一，到底怎么样定义服务不可用？例如返回业务的系统忙算不算可用？第二，直观理解，可用性应该是针对单个接口定义的，一个系统有很多接口，那么一个系统的可用性又应该怎么计算呢？
非常感谢！

二康

2018-06-19

期待耗子叔可以讲讲如何设计一个补偿框架，可以讲讲具体地实现过程和相关技术和难点。谢谢

阿拖

2018-04-03

感觉有点像2pc，工作流引擎是cooradinator。

mingshun

2018-05-23

一直觉得补偿事务很烦琐，日常实现也是尽量避免。看了这篇后，发现是没有很好地记录起始状态，总是想着通过目标状态来反推，所以总感觉实现起来很烦琐，逻辑错踪复杂。也许是重度精神洁癖导致吧！总觉得对正向目标没意义的数据都没有记录的必要甚至认为是浪费，然而顾着正向目标却忘了反向目标，而两个方向的目标都同等重要。

neohope

2018-06-21

皓哥， 您好！我们业务上会面临一种情况，就是跨厂商跨系统保持系统间数据同步，无论是通过代码直接操作多个数据库进行同步，还是通过让厂商提供服务来进行同步，最终要么实现效果很差，要么就同步机制弄的略复杂（比如订阅发布，厂商不配合），让计划没法推进。您这边后续有计划说一下类似情况如何处理吗？

说到库存的话，我们行业这边还有一种神奇的操作。这种操作的要求是，“你没货我理解，但是你说有货、我要了、你再说没有那就不行”。所以在我们行业，通常把库存分为实库存和用户库存。用户看到的库存永远小于实库存，由于并发量并不大，只需要用简单的事务控制，也能在很大程度上可以避免超卖。但当库存很低的情况下，代码就要用严格的事务控制，来避免超卖了，执行效率就很低，好在这样的情况很少。

追梦

2018-06-17

求，耗子叔，讲解分布式事务原理��

jackwoo

2018-06-17

希望可以出个工作流引擎介绍

颜忒妥

2018-06-12

在分布式的服务架构中，一致性（Consistency）、可用性（Availability）、分区容忍性（Partition Tolerance），在现实中不能都满足，最多只能满足其中两个。
我觉得这句话有点歧义，我理解的是在不发生网络分区的情况下，CAP都能满足，当发生网络分区的时候，只能在CA中选一个。

王磊

2018-06-11

我的理解是分布式事务为了提高性能，将要做的若干事情记录下来(属于本地事务)，然后再异步去执行这些若干事情(并行或串行)，尽量使其成功(重试)，如果的确不能成功，则需要回滚。这些

若干事情中对于占用资源的事情，需要增加一个占用时间的限制，如果超过此时间但整个事务还没有提交，则释放资源，如如果没有在30分钟内支付，则释放库存。

小伟

2018-04-02

亚马逊的工作流是如何实现呢？

刘勇

2018-03-31

补偿一词不是很理解，为什么回滚叫补偿？

作者回复

2018-04-06

回滚是补偿的子集

流畅

2018-03-12

想再了解下工作流的实现，希望能讲解下

阿凡达

2018-03-09

期待深入讲解文中提到的工作流。