**基于服务的分布式事务(上)**

**传统数据库事务**

在传统单体应用架构下，我们通常会将业务数据存储在一个数据库中，应用各模块直接对数据库进行操作业务数据。由数据库提供基于ACID[1]的事务保证。

* A是Atomic 原子性：事务作为整体来执行，要么全部执行，要么都不执行。
* C是Consistency 一致性：事务应确保数据从一个一致的状态转变为另一个一致的状态。
* I是 Isolation 隔离性：多个事务并发执行时，一个事务的执行不应影响其他事务的执行。
* D是Durability 持久性：已提交的事务修改数据会被持久保持。

例如一个电商的下单操作，就涉及到用户系统、库存系统、支付系统以及配送系统等一系列的协同操作。我们在执行下单操作的过程中，如果出现库存短缺，或者用户账户余额不足的情况，这个下单操作就会涉及到一系列的相关业务系统调用。如果这些子系统连接同一个数据库，我们可以通过数据库提供的事务原子性机制将库存数量校验以及用户余额校验的工作，和执行具体的下单业务操作组合成为一个数据库事务操作。通过数据库事务原子性来保证系统各个模块的调用要么都成功，要么都失败（取消）。 同时，由于数据库提供一致性和持久性保证，保证了如果事务执行成功并提交，本次业务操作的数据在立即生效的同时不会产生异议。 同时数据库提供了不同级别的数据锁机制保证应用多个线程同时读取或者更新数据的过程中不会相互影响，从而来保证业务操作的隔离性。

**微服务的分布式事务**

随着微服务架构的流行，很多大型的业务流程被拆分成为了多个功能单一的基础服务，大家会根据业务的诉求在这些基础服务之上编写一些组合调用服务来满足业务诉求。为了保证微服务能够独立开发部署运行，通常我们会采用一个微服务对应一个数据库的架构，将内部数据经微服务封装之后，以服务的方式对外暴露。这样以往基于数据库来实现的数据操作，就变成了多个对外提供服务的微服务系统的协同完成操作。因为单个微服务只知道自己的服务执行情况，为了保证分布事务的一致性，参与分布式事务的微服务通常会依托协调器完成相关的一致性协调操作[2]。

在十多年前分布式事务的实现方案有CORBA的 Object Transaction Service（OTS)、J2EE的 Java Transaction API 以及 Java Transaction Service。这些事务管理以及事务服务的技术都是建立在ACID事务的概念上的。协调器依托于底层的资源交互协议实现资源的占用以及提交的操作，通过两阶段提交[3]的方式实现分布式事务的强一致操作。两阶段提交将分布式事务操作分为准备和提交两个阶段：系统在准备解决阶段完成资源操作， 如果准备阶段中出现问题，支持回滚操作，但是在提交阶段是不允许出错的。两阶段在保证事务原子性上做了很多工作，但是两阶段提交最大的问题是在分布式事务执行过程中， 所有参与事务的节点资源都是被锁定的，系统不允许其他节点访问锁定的资源，在这种执行下很难进一步提升系统的执行效率。

如上所述，在ACID的事务执行过程中，为了保证事务的隔离性，通常我们会采用读写加锁的方式，通过串行处理数据方式，保证多个事务在同时执行的过程中不会相互影响。也就是说只有当事务提交并且保存修改记录或者回退取消修改记录之后，其他的事务才能继续执行。然而对于由多个事务组成的长时间运行的事务来说，如果在整个事务的执行过程都采用这种机制来保证事务的隔离性是一种很低效的解决方案。

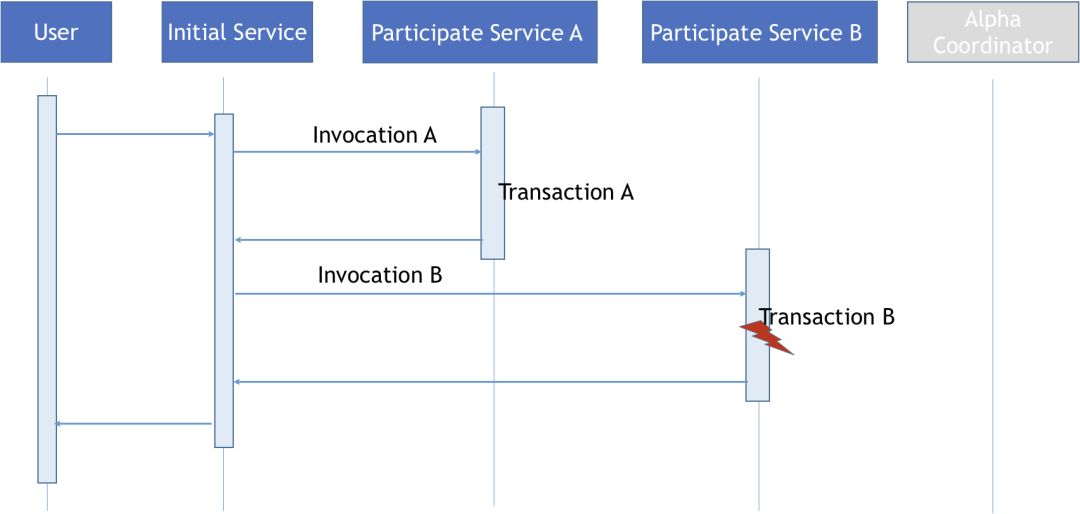
那我们有什么办法即提高系统运行效率，又能保证事务的数据一致性呢？

答案是采用补偿的方式来解决这一问题。

**基于补偿的事务实现**

补偿是指我们将一个事务分成一个本地执行的正常操作事务和一个逻辑上对之前的操作进行补偿的事务。这样采用补偿事务的方式，我们可以把一个长时间运行的事务分化成若干个可以立即提交的本地事务调用，而不是一个长时间占用锁资源的巨型事务。 这样做的最大好处就是极大降低锁占用的时间。作为代价，补偿方式的取消操作和以往的实现方式有很大的不同，我们需要执行一个单独的ACID事务来完成对之前已提交的事务的逻辑补偿。

下图展示了一个典型的分布式事务调用， 用户请求触发事务初始服务， 事务初始服务会顺序调用两个事务参与服务（服务A，服务B）。由于这两个事务参与服务之间没有联系，当事务参与服务执行出现了问题，需要一个协调器参与相关的恢复操作。



这里我们可以根据补偿执行的不同将其分成两组不同的补偿方式：

* 不完美补偿 - 反向操作会留下之前原始事务操作的痕迹，一般来说我们是会在原始事务记录中设置取消状态。
* 完美补偿 - 反向逻辑会彻底清理之前的原始事务操作，一般来说是不会保留原始事务交易记录，用户是感知不到事务取消之前的状态信息的。

对于采用不完美的补偿方式的系统（Saga实现[4]）来说：

我们的补偿事务逻辑和其他的事务逻辑相比没有什么不同， 系统只需要像执行其他业务逻辑一样执行相关的补偿操作即可，无需设置特殊的处理逻辑来恢复事务执行之前的状态。

以我们常见的银行ATM取款业务为例，银行账户预先进行扣减的操作，如果取款不成功，其逻辑恢复操作就是通过冲正的方式将预先扣减的款项打回到用户账户，我们可以通过查看账户的交易记录找到扣减和冲正的记录信息。

下图展示的内容就是当初始服务调用分别调用服务A和服务B，服务B执行出现错误，这个时候我们事务协调器会调用服务A的冲正方法将系统状态恢复到执行服务调用之前的状态。

对于采用完美补偿方式的系统（Try-Cancel/Confirm[5]实现)来说：

为了让系统能够在补偿操作彻底清除事务执行的情况，我们会借助两阶段提交协议来完成这部分的功能。

在TCC方式下，cancel补偿显然是在第二阶段需要执行业务逻辑来取消第一阶段产生的后果。try是在第一阶段执行相关的业务操作，完成相关业务资源的占用，例如预先分配票务资源，或者检查并刷新用户的账户信用额度。在cancel阶段释放相关的业务资源，例如释放预先分配的票务资源或者恢复之前占用的用户信用额度。

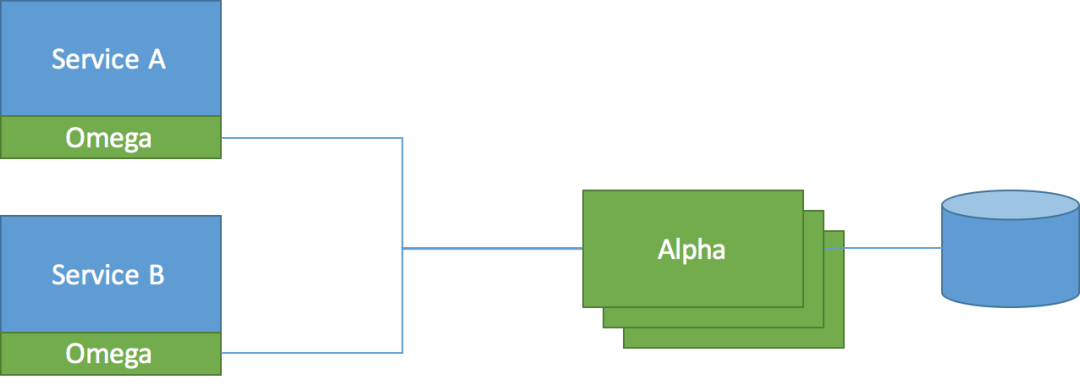
那我们为什么还要加入confirm操作呢？

这需要从业务资源的使用生命周期来入手。在try过程中，我们只是占用的业务资源，相关的执行操作只是出于待定状态，只有在确认操作执行完毕之后，业务资源才能真正被确认。例如订票业务的try操作，我们只是占用了相关的票务资源。目的是防止票务资源被其他用户占用，但是业务还没有执行完毕，票务提供方还不能将被占用的票务资源统计为已售出票务。 只有相关票务资源被确认售出的之后，票务提供方才能将其统计为已售出票务资源。

**ServiceComb Pack架构介绍**

通过上面的分析我们可以发现一个有意思的现象，每一步事务的操作都有可能会根据业务的执行情况提供一个补偿操作，通过一个事务管理系统来协调这个补偿操作可以帮我们大大降低业务流程建模的复杂度。在分布式事务实现过程中， 协调器的作用非常重要， 各个事务的参与方需要跟协调器建立好良好的沟通， 由协调器统一调度完成相关事务的执行或者取消的操作。

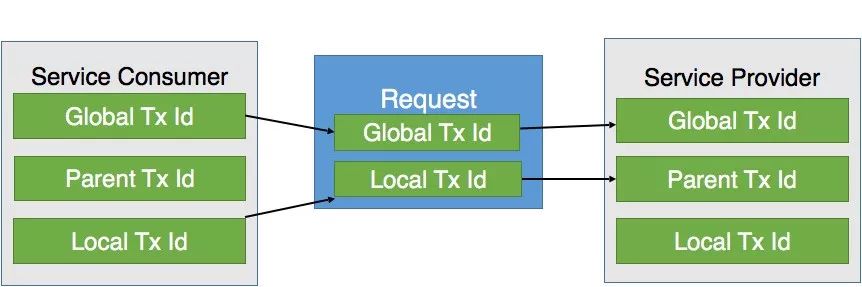
ServiceComb Pack架构如下图所示，主要包含两个组件，即Alpha和Omega，其中：



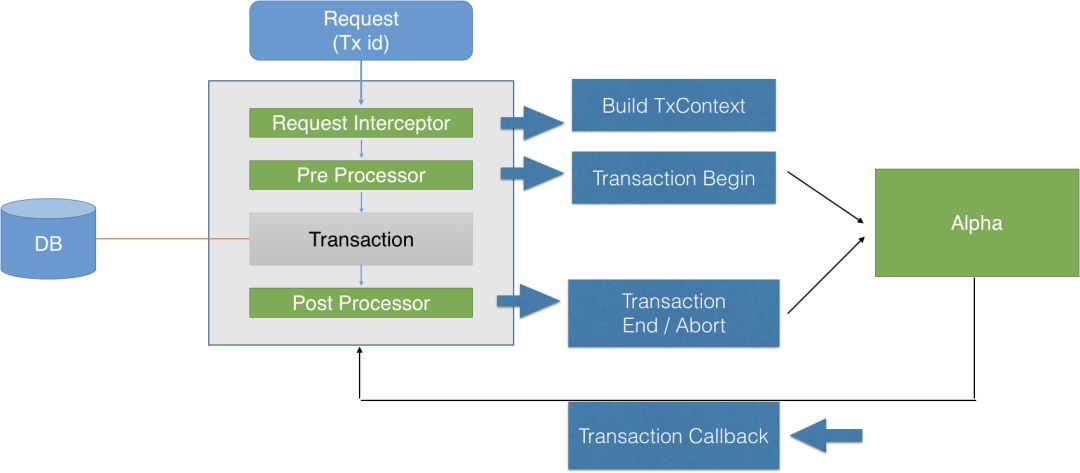
Alpha充当协调者的角色，主要负责对事务的事件进行持久化存储以及协调子事务的状态，使其最终得以与全局事务的状态保持一致，即保证事务中的子事务要么全执行，要么全不执行。

Omega是微服务中内嵌的一个agent，负责对监控本地事务执行情况并向Alpha上报事务执行事件，并在异常情况下根据alpha下发的指令执行相应的补偿或重试操作。

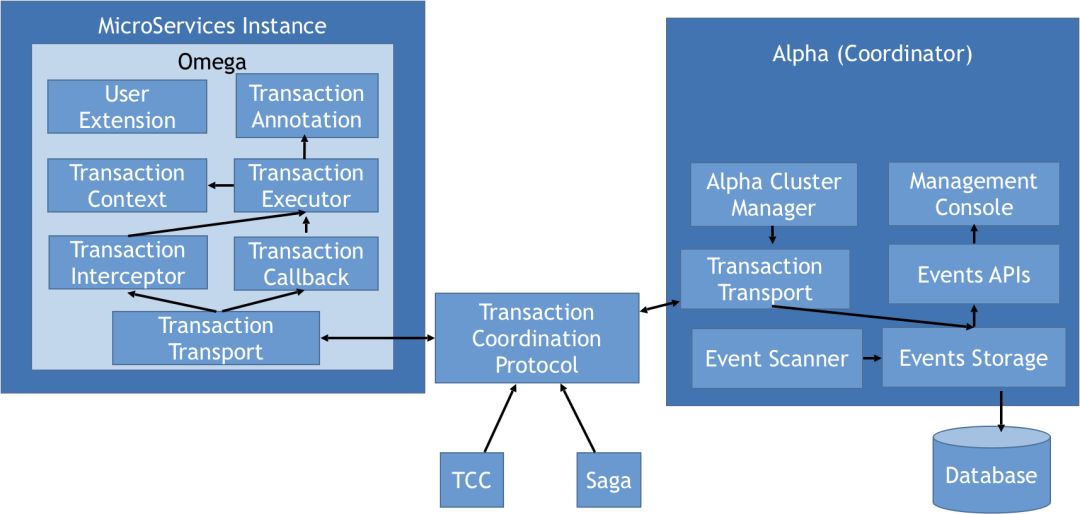
Omega可以通过向Alpha发送消息的方式向Alpha实时传递事务执行的进展，但是Alpha怎么知道这些Omega上传的消息是相互关联的呢？我们通过在服务调用过程中插入唯一的全局事务ID，并在后续的调用其它服务过程中传递这个全局事务ID。通过全局事务ID可以从汇总到Alpha事件中找到事件与之相关联的所有事件，通过对这些事件信息进行分析，我们可以完整地追踪到与分布式事务执行情况。



Omega会以切面编程的方式向应用程序注入相关的处理模块，帮助我们构建分布式事务调用的上下文。Omega在事务处理初始阶段处理事务的相关准备的操作，并在事务执行完毕后做一些清理的操作，例如创建分布式事务起始事件，以及相关的子事件，根据事务执行的成功或者失败生成相关的事务终止或者失败事件。这样带来的好处是用户的代码只需要添加几个annotation来描述分布式事务执行范围，以及与本地的事务处理恢复的相关函数信息，Omega就能通过切面注入的代码追踪本地事务的执行情况。Omega会将本地事务执行的情况以事件的方式通知给Alpha。由于单个Omega不可能知晓一个分布式事务下其他参与服务的执行情况，如此一来，就需要Alpha扮演一个十分重要的协调者的角色。Alpha将收集到的分布式事务事件信息整理汇总，通过分析这些事件之间的关系可以了解到分布式事务的执行情况。Alpha通过向Omega下发相关的执行指令由Omega执行相关提交或恢复操作，实现分布式事务的最终一致性。



在了解过Pack实现的部分细节之后， 我们可以从下图进一步了解ServiceComb Pack架构下，Alpha与Omega内部各模块之间的关系图。



整个架构分为三个部分：

|  |  |
| --- | --- |
| 1 | 一个是Alpha协调器（支持多个实例提供高可用支持）。 |
| 2 | 二是注入到微服务实例中的Omega。 |
| 3 | 三是Alpha与Omega之间的交互协议。 |
| 目前ServiceComb Pack支持Saga 以及TCC两种分布式事务协调协议实现。 | |

Omega包含了与分析用户分布式事务逻辑相关的模块：

* 事务注解模块（Transaction Annotation）
* 事务拦截器（Transaction Interceptor）
* 分布式事务执行相关的事务上下文（Transaction Context）
* 事务回调（Transaction Callback）
* 事务执行器（Transaction Executor）

以及负责与Alpha进行通讯的事务传输（Transaction Transport）模块。

**事务注解模块**是分布式事务的用户界面，用户将这些标注添加到自己的业务代码之上用以描述与分布式事务相关的信息，这样Omega就可以按照分布式事务的协调要求进行相关的处理。如果大家扩展自己的分布式事务，也可以通过定义自己的事务标注来实现。

**事务拦截器**这个模块我们可以借助AOP手段，在用户标注的代码基础上添加相关的拦截代码，获取到与分布式事务以及本地事务执行相关的信息，并借助事务传输模块与Alpha进行通讯传递事件。

事务上下文为Omega内部提供了一个传递事务调用信息的一个手段，借助前面提到的全局事务ID以及本地事务ID的对应关系，Alpha可以很容易检索到与一个分布式事务相关的所有本地事务事件信息。

**事务执行器**主要是为了处理事务调用超时设计的模块。由于Alpha与Omega之间的连接有可能不可靠，Alpha端很难判断Omega本地事务执行超时是由Alpha与Omega直接的网络引起的还是Omega自身调用的问题，因此设计了事务执行器来监控Omega的本地的执行情况，简化Omega的超时操作。目前Omega的缺省实现是直接调用事务方法，由Alpha的后台服务通过扫描事件表的方式来确定事务执行时间是否超时。

**事务回调**在Omega与Alpha建立连接的时候就会向Alpha进行注册，当Alpha需要进行相关的协调操作的时候，会直接调用Omega注册的回调方法进行通信。 由于微服务实例在云化场景启停会很频繁，我们不能假设Alpha一直能找到原有注册上的事务回调， 因此我们建议微服务实例是无状态的，这样Alpha只需要根据服务名就能找到对应的Omega进行通信。

**事务传输模块**负责Omega与Alpha之间的通讯，在具体的实现过程中，Pack通过定义相关的Grpc描述接口文件定义了TCC 以及Saga的事务交互方法， 同时也定义了与交互相关的事件。我们借助了Grpc所提供的双向流操作接口实现了Omega与Alpha之间的相互调用。 Omega和Alpha的传输建立在Grpc多语言支持的基础上，为实现多语言版本的Omega奠定了基础。

Alpha为了实现其事务协调的功能，首先需要通过事务传输（Transaction Transport）接收Omega上传的事件， 并将事件存在事件存储（Event Store）模块中，Alpha通过事件API（Event API）对外提供事件查询服务。Alpha会通过事件扫描器（Event Scanner）对分布式事务的执行事件信息进行扫描分析，识别超时的事务，并向Omega发送相关的指令来完成事务协调的工作。由于Alpha协调是采用多个实例的方式对外提供高可用架构， 这就需要Alpha集群管理器（Alpha Cluster Manger）来管理Alpha集群实例之前的协调。用户可以通过管理终端（Manage console）对分布式事务的执行情况进行监控。

目前Alpha的事件存储是构建在数据库基础之上的。为了降低系统实现的复杂程度，Alpha集群的高可用架构是建立在数据库集群基础之上的。 为了提高数据库的查询效率，我们会根据全局事务的执行情况将数据存储分成了在线库以及存档库，将未完成的分布式事务事件存储在在线库中， 将已经完成的分布式事务事件存储在存档库中。

**事件API**是Alpha对外暴露的Restful事件查询服务。 该模块功能首先应用在Pack的验收测试[6]中，通过事件API验收测试代码可以很方便的了解Alpha内部接收的事件。验收测试通过模拟各种分布式事务执行异常情况（错误或者超时），比对Alpha接收到的事务事件来验证相关的其他事务协调功能是否正确。

**管理终端**是一个js的前端界面， 管理终端通过访问事件API提供的Rest服务，向用户提供分布式事务执行情况的统计分析，并且可以追踪单个全局事务的执行情况，找出事务的失败的根源。在Pack 0.3.0 中实现了一部分功能，后续还需要进一步完善，欢迎大家参与进来。

**Alpha集群管理器**负责Alpha实例注册工作，管理Alpha中单个服务的执行情况， 并且为Omega提供一个及时更新的服务列表。 通过集群管理器用户可以轻松实现Alpha服务实例的启停操作，以及Alpha服务实例的滚动升级功能。目前这部分的模块还在设计开发中，欢迎对此有兴趣的朋友加入到我们的开发队伍中来。

**小结**

本文从分布式事务需要解决的问题入手，向大家介绍了建立在补充基础之上的基于服务的分布式事务的解决思路。接下来我们结合具体的示例介绍了完美的补偿（TCC）和非完美补偿（Saga）两种分布式事务协调协议，最后结合ServiceComb Pack的实现原理详细介绍了ServiceComb Pack的架构实现。

在基于服务的分布式事务下篇中，我们将结合具体的示例向大家介绍TCC以及Saga分布式事务协调协议的交互细节，以及如何使用ServiceComb Pack编写TCC 以及Saga 应用。

**参考链接详情**

[1]ACID

https://en.wikipedia.org/wiki/ACID

[2]一致性协调操作https://www.enterpriseintegrationpatterns.com/patterns/conversation/CoordinatedAgreement.html

[3]两阶段提交

https://zh.wikipedia.org/wiki/%E4%BA%8C%E9%98%B6%E6%AE%B5%E6%8F%90%E4%BA%A4

[4]Saga

https://microservices.io/patterns/data/saga.html

[5]Try-Cancel/Confirm

https://dzone.com/articles/transactions-for-the-rest-of-us

[6]Pack的验收测试

https://github.com/apache/servicecomb-pack/tree/master/acceptance-tests