分布式事务：不过是在一致性、吞吐量和复杂度之间，做一个选择

刘相 聊聊架构 2016-09-07

这是一个开撕的话题，我经历过太多的关于分布式事务的需求：“有没有简单的方案，像使用数据库事务那样，解决分布式数据一致性的问题”。特别是微服务架构流行的今天，一次交易需要跨越多个“服务”、多个数据库来实现，传统的技术手段，已经无法应对和满足微服务情况下这些复杂的场景了。针对微服务下的交易业务如何保障数据一致性，本文尽量做到理论结合实践，将**我们在实际产品中用到的分布式事务实现机制**，和大家扒一扒，希望能帮到各位。

谈到分布式事务，必须先把CAP拿出来说说事……，当然还有BASE……

从架构的角度来看，业务拆分（数据分区）、数据一致性、性能（可用性）永远是个平衡的艺术：

- 在微服务架构下，为了获得更高的性能与灵活性，将业务应用拆分为多个，交易跨多个微服务编排，数据一致性的问题产生；
- 为了解决数据一致性问题，需要采用不同的事务机制来保障，这又会产生性能（可用性）问题；

在计算机世界里，为了解决一件事情，另外的问题就会接踵而至，从另一个层面印证了IT架构永远是一种平衡的艺术。

“BASE”其核心思想是根据业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）；在互联网领域，通常需要**牺牲强一致性来换取系统的高可用性**，只需要保证数据的“最终一致”，只是这个最终时间需要在用户可以接受的范围内；但在金融相关的交易领域，仍然需要采用强一致性的方式来保障交易的准确性与可靠性。

接下来为大家介绍业界常见的事务处理模式，包括**两阶段提交、三阶段提交、Sagas长事务、补偿模式、可靠事件模式（本地事件表、外部事件表）、可靠事件模式（非事务消息、事务消息）、TCC**等。不同的事务模型支持不同的数据一致性。如果读者对这几种分布式事务比较熟悉，可以直接参考下图并结合自身业务需求选择合适的事务模型。

	两阶段	三阶段	Sagas长事务	补偿	可靠事件	TCC
一致性	强一致性	强一致性	最终一致	最终一致	最终一致	最终一致
事务	全局	全局	全局	全局	全局	全局
吞吐量	弱	弱	高	中	高	中
实现复杂度	易	易	难	中	难	难

两阶段提交、三阶段提交

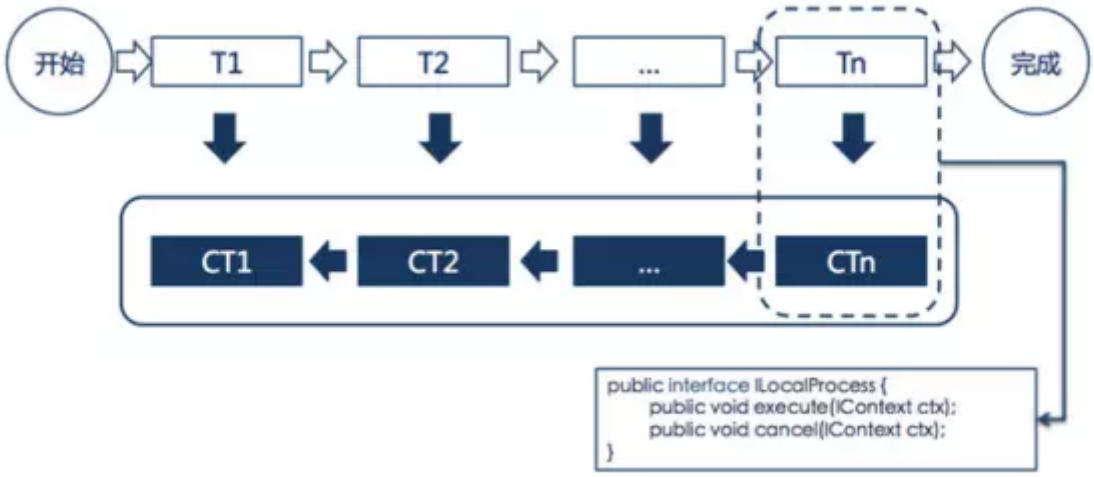
这种分布式事务解决方案目前在各种技术平台上已经比较成熟：JavaEE架构下面的JTA事务（各应用服务器均提供了实现，Tomcat除外）。

但目前两阶段提交、三阶段提交存在如下的局限性，并不适合在微服务架构体系下使用：

- 所有的操作必须是事务性资源（比如数据库、消息队列、EJB组件等），存在使用局限性（微服务架构下多数使用HTTP协议），比较适合传统的单体应用；
- 由于是强一致性，资源需要在事务内部等待，性能影响较大，吞吐率不高，不适合高并发与高性能的业务场景；

Sagas长事务

在Sagas事务模型中，一个长事务是由一个预先定义好执行顺序的子事务集合和他们对应的补偿子事务集合组成的。典型的一个完整的交易由T1、T2、……、Tn等多个业务活动组成，每个业务活动可以是本地操作、或者是远程操作，所有的业务活动在Sagas事务下要么全部成功，要么全部回滚，不存在中间状态。

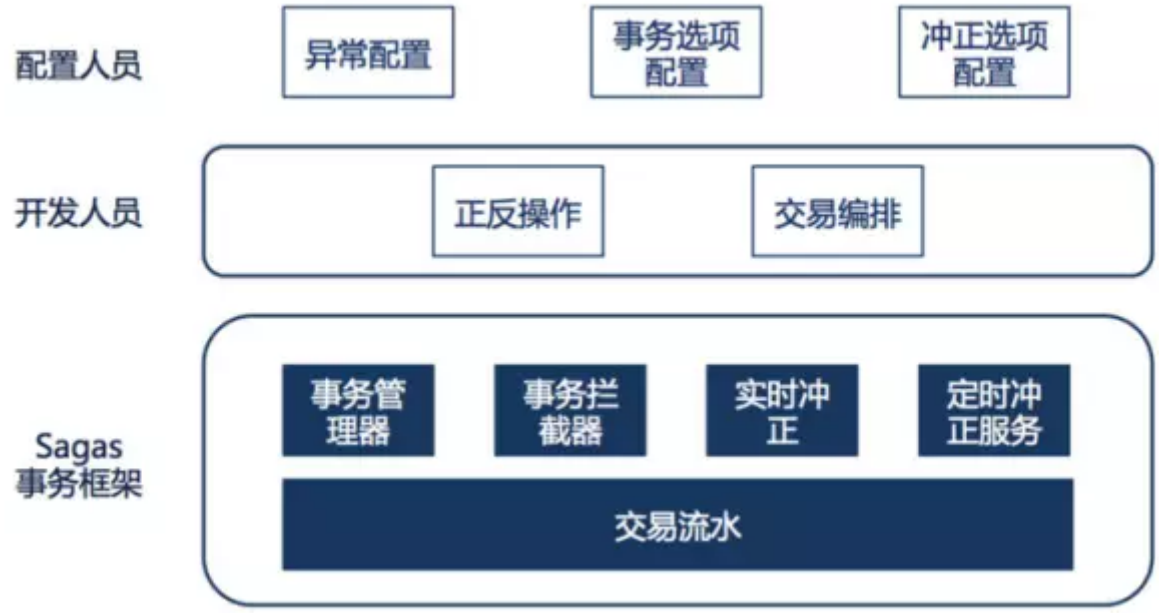


Sagas事务模型的实现机制：

- 每个业务活动都是一个原子操作；
- 每个业务活动均提供正反操作；
- 任何一个业务活动发生错误，按照执行的反顺序，实时执行反操作，进行事务回滚；
- 回滚失败情况下，需要记录待冲正事务日志，通过重试策略进行重试；
- 冲正重试依然失败的场景，提供定时冲正服务器，对回滚失败的业务进行定时冲正；
- 定时冲正依然失败的业务，等待人工干预；

Sagas长事务模型支持对**数据一致性要求比较高的场景比较适用**，由于采用了补偿的机制，每个原子操作都是先执行任务，避免了长时间的资源锁定，能做到实时释放资源，性能相对有保障。

Sagas长事务方式如果由业务去实现，复杂度与难度并存。在我们实际使用过程中，开发了一套支持Sagas事务模型的框架来支撑业务快速交付。

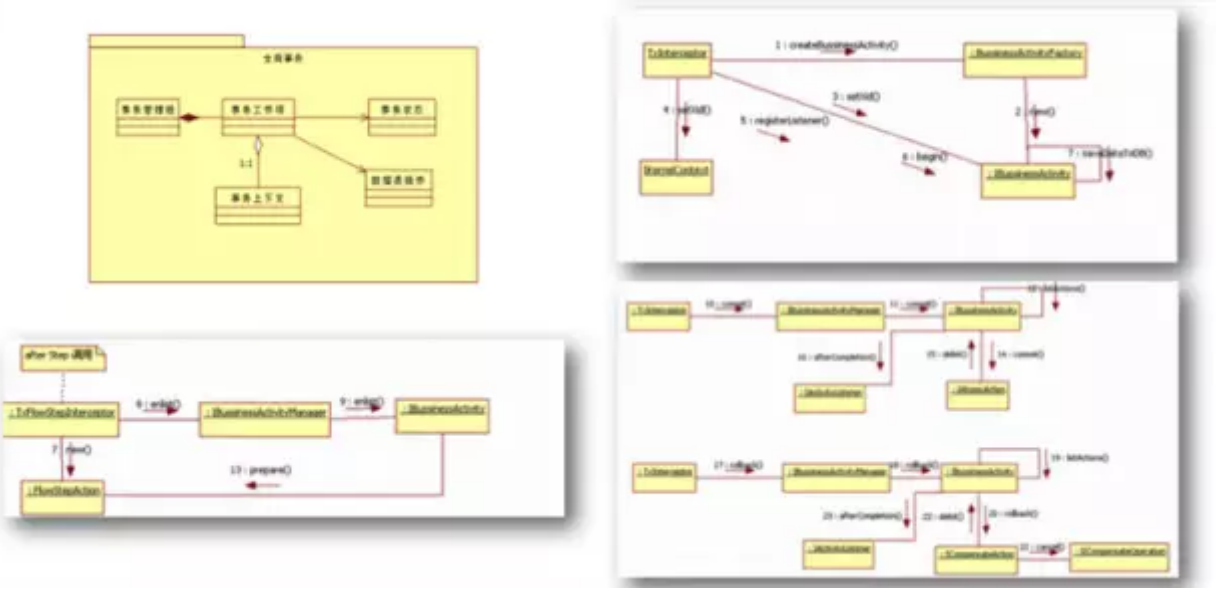


开发人员：业务只需要进行交易编排，每个原子操作提供正反交易；

配置人员：可以针对异常类型设定事务回滚策略（哪些异常纳入事务管理、哪些异常不纳入事务管理）；每个原子操作的流水是否持久化（为了不同性能可以支持缓存、DB、以及扩展其它持久化方式）；以及冲正选项配置（重试次数、超时时间、是否实时冲正、定时冲正等）；

Sagas事务框架：提供事务保障机制，负责原子操作的流水落地，原子操作的执行顺序，提供实时冲正、定时冲正、事务拦截器等基础能力；

Sagas框架的核心是IBusinessActivity、IAtomicAction。IBusinessActivity完成原子活动的enlist()、delist()、prepare()、commit()、rollback()等操作；IAtomicAction主要完成对状态上下文、正反操作执行。



限于文章篇幅，本文不对具体实现做详述；后面找时间可以详细介绍基于Sagas长事务模型具体的实现框架。Sagas长事务需要交易提供反操作，支持事务的强一致性，由于没有在整个事务周期内锁定资源，对性能影响较小，适合对数据要求比较高的场景中使用。

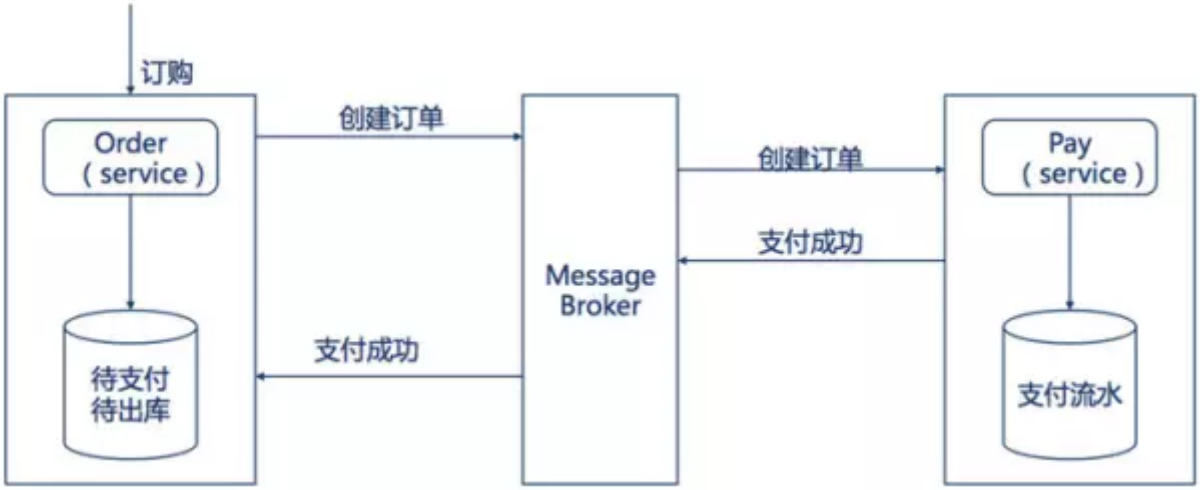
补偿模式

Sagas长事务模型本质上是补偿机制的复杂实现，如果实际业务场景上不需要复杂的Sagas事务框架支撑，可以在业务中实现简单的补偿模式。补偿过程往往也同样需要实现最终一致性，需要保证取消服务至少被调用一次和取消服务必须实现幂等性。补偿模式可以参见同事田向阳的技术文章《微服务架构下数据一致性保证（三）》http://dwz.cn/3TVJaB

补偿机制不推荐在复杂场景（需要多个交易的编排）下使用，优点是非常容易提供回滚，而且依赖的服务也非常少，与Sagas长事务比较来看，使用起来更简便；缺点是会造成代码量庞大，耦合性高，对应无法提供反操作的交易不适合。

可靠事件模式（本地事件表、外地事件表）

可靠事件模式属于事件驱动架构，当某件重要事情发生时，例如更新一个业务实体，微服务会向消息代理发布一个事件。消息代理会向订阅事件的微服务推送事件，当订阅这些事件的微服务接收此事件时，就可以完成自己的业务，也可能会引发更多的事件发布。

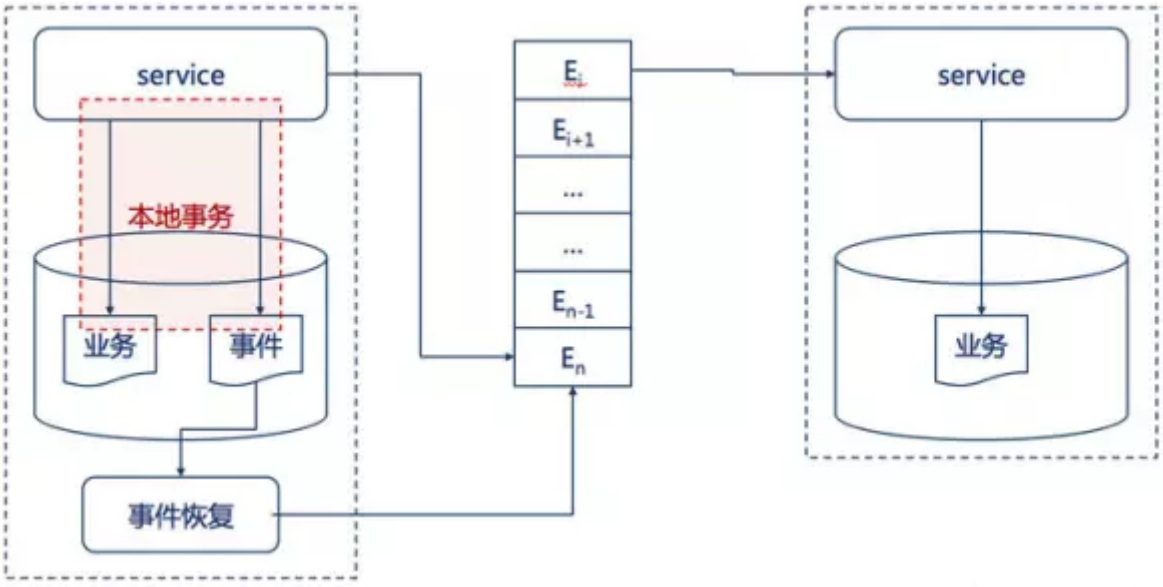


可靠事件模式在于保证可靠事件投递和避免重复消费，靠事件投递定义为:

- 每个服务原子性的业务操作和发布事件;
- 消息代理确保事件传递至少一次；避免重复消费要求服务实现幂等性。

基于事件模式，需要重点考虑的是事件的可靠到达，在我们产品实际支持过程中，通常有本地事件表、外部事件表两种模式：

1. 本地事件表方法将事件和业务数据保存在同一个数据库中，使用一个额外的“事件恢复”服务来恢复事件，由本地事务保证更新业务和发布事件的原子性。考虑到事件恢复可能会有一定的延时，服务在完成本地事务后可立即向消息代理发布一个事件。



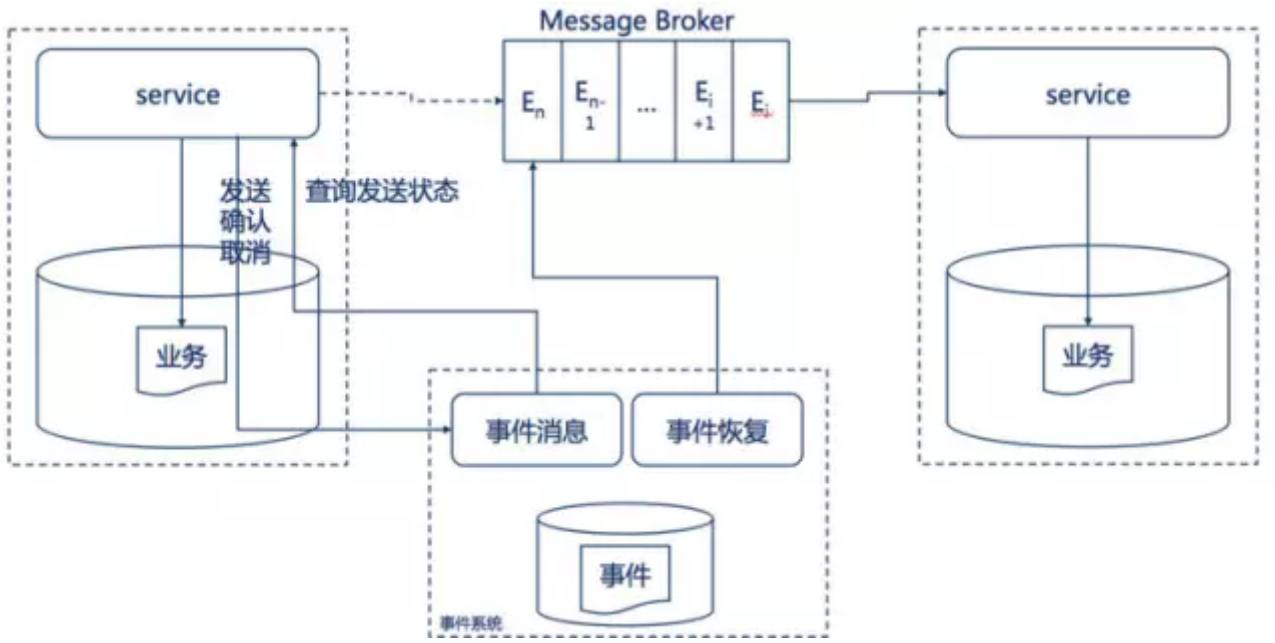
i. 微服务在同一个本地事务中记录业务数据和事件；

ii. 微服务实时发布一个事件立即通知关联的业务服务，如果事件发布成功立即删除记录的事件；

iii. 事件恢复服务定时从事件表中恢复未发布成功的事件，重新发布，重新发布成功才删除记录的事件；

其中第2条的操作主要是为了增加发布事件的实时性，由第三条保证事件一定被发布。本地事件表方式业务系统和事件系统耦合比较紧密，额外的事件数据库操作也会给数据库带来额外的压力，可能成为瓶颈。

2. 外部事件表方法将事件持久化到外部的事件系统，事件系统需提供实时事件服务以接受微服务发布事件，同时事件系统还需要提供事件恢复服务来确认和恢复事件。



i. 业务服务在事务提交前，通过实时事件服务向事件系统请求发送事件，事件系统只记录事件并不真正发送；

ii. 业务服务在提交后，通过实时事件服务向事件系统确认发送，事件得到确认后，事件系统才真正发布事件到消息代理；

iii. 业务服务在业务回滚时，通过实时事件向事件系统取消事件；

iv. 如果业务服务在发送确认或取消之前停止服务了怎么办呢？事件系统的事件恢复服务会定期找到未确认发送的事件向业务服务查询状态，根据业务服务返回的状态决定事件是要发布还是取消；

该方式将业务系统和事件系统独立解耦，都可以独立伸缩。但是这种方式需要一次额外的发送操作，并且需要发布者提供额外的查询接口。

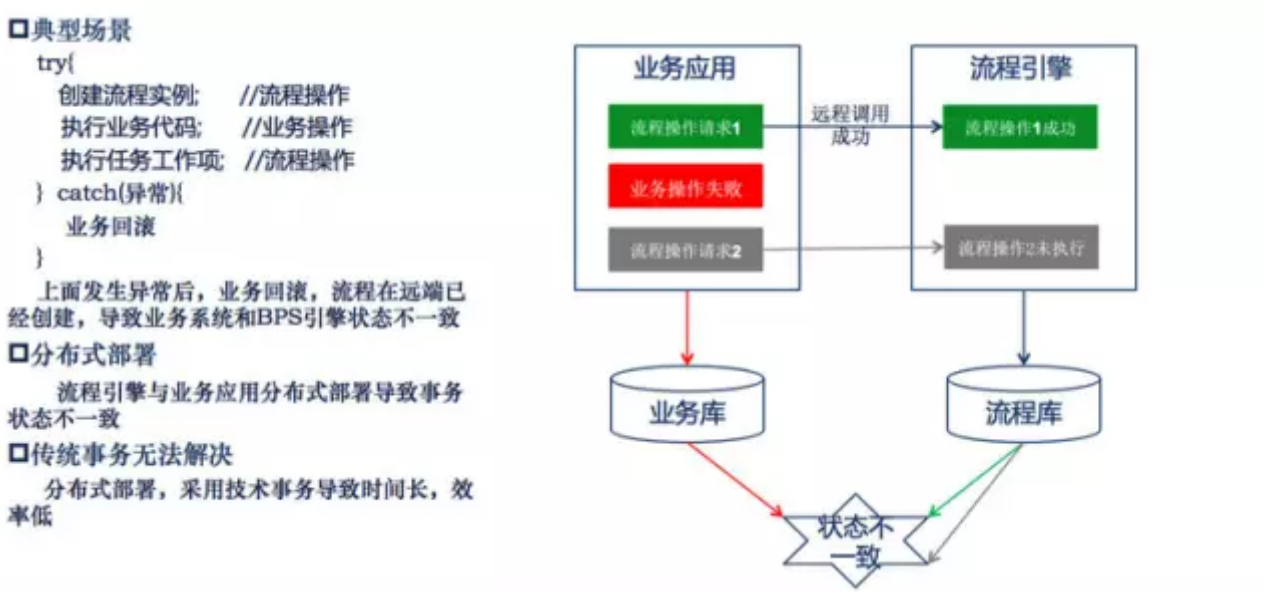
基于可靠事件的事务保障模式可以有很多的变种实现，比如对消息可靠性不高的话，可以将本地表的方式换做缓存方式。为了**提高消息投递的效率**，可以将多次消息合并为投递模

式。为了**提供强一致性的事务保障**，甚至可以将本地消息表持久化（保障发方法消息可靠落地）+远程消息表持久化（保障接收方消息可靠落地）结合的模式。

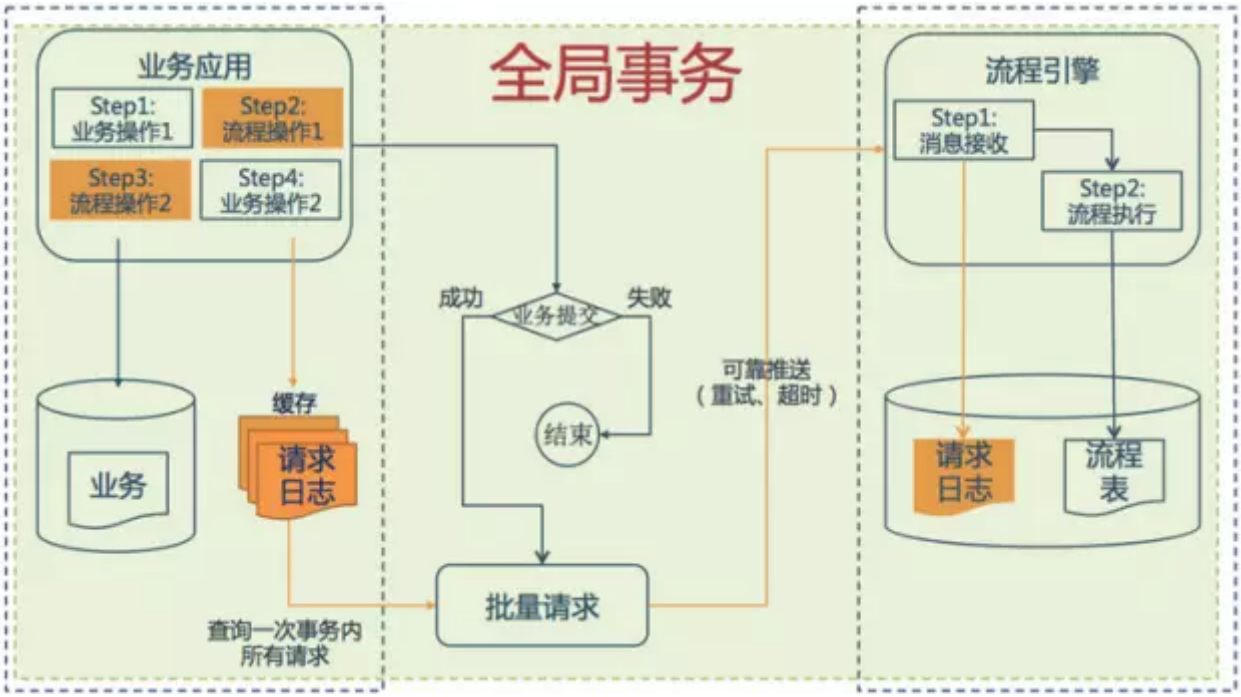
在我们的流程产品中针对业务和流程的分布式事务解决方案就**采用了多次消息合并投递+本地缓存+远程消息表持久化的模式**，接下来为大家介绍具体的使用方式。

使用场景

在实际业务项目中通常采用业务与流程分布式部署的模式，业务系统通过远程接口访问流程引擎，业务数据同流程数据存放到各自的数据库中。



在这种场景中，如果业务系统的流程操作和业务操作交叉在一起，当流程操作成功，而业务操作失败时，就会造成业务回滚，而流程在引擎端已经创建，导致业务系统和流程引擎状态不一致。



在业务应用中对一个事务中的流程操作采用本地缓存+批量投递+远程落地的模式（如果需要在客户端确保消息可靠性，可以将本地缓存换成本地表的方式）；在流程引擎端在消息投递来之后，做了消息表落地的工作，保障可靠执行。在我们流程产品中流程引擎对外提供的客户端提供了统一的分布式事务API，和使用传统本地事务一样进行操作，保证了透明性，简化开发人员的复杂度。分布式事务API支持两种协议模式：

1. HTTP + 二进制序列化的模式
2. WebService模式

后续我们会增加Restful风格的接口。

可靠事件模式在互联网公司中有着较大规模的应用，该方式适合的业务场景非常广泛，而且能够做到数据的最终一致性，缺点是该模式实现难度较大，依赖数据库实现可靠性，在高并发场景下可能存在性能瓶颈，需要在公司层面搭建一套标准的可靠事件框架来支撑。

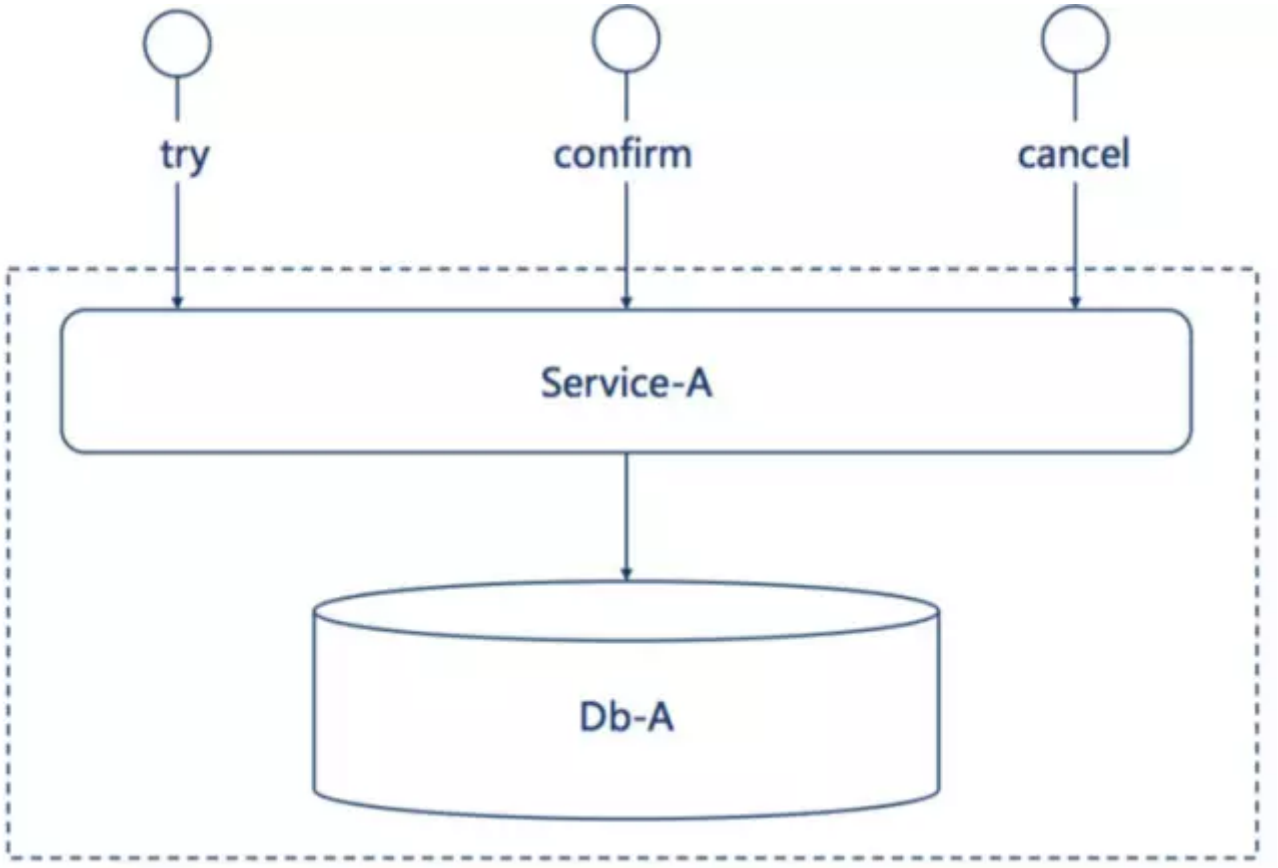
可靠事件模式（非事务消息、事务消息）

可靠事件模式的事件通知可以采用消息的模式来实现，其实现原理和本地事件表、外部事件表一致，本文就不在详述。目前常用的消息框架ActiveMQ、RabbitMQ、Kafka、RocketMQ可以用来作为消息投递的渠道。注意：Kafka通常不适合，因为Kafka的设计存在丢消息的场景。

目前市面上支持事务的消息产品比较少，RocketMQ虽然实现了可靠的事务模式，但并没有开源出来、没有开源出来、没有开源出来，顺便说一下国内的开源有太多需要改进的空间（关键点不开源，开源后没有持续的投入）。

TCC模式

一个完整的TCC业务由一个主业务服务和若干个从业务服务组成，主业务服务发起并完成整个业务活动，TCC模式要求从服务提供三个接口：Try、Confirm、Cancel。



1. Try：完成所有业务检查

   预留必须业务资源

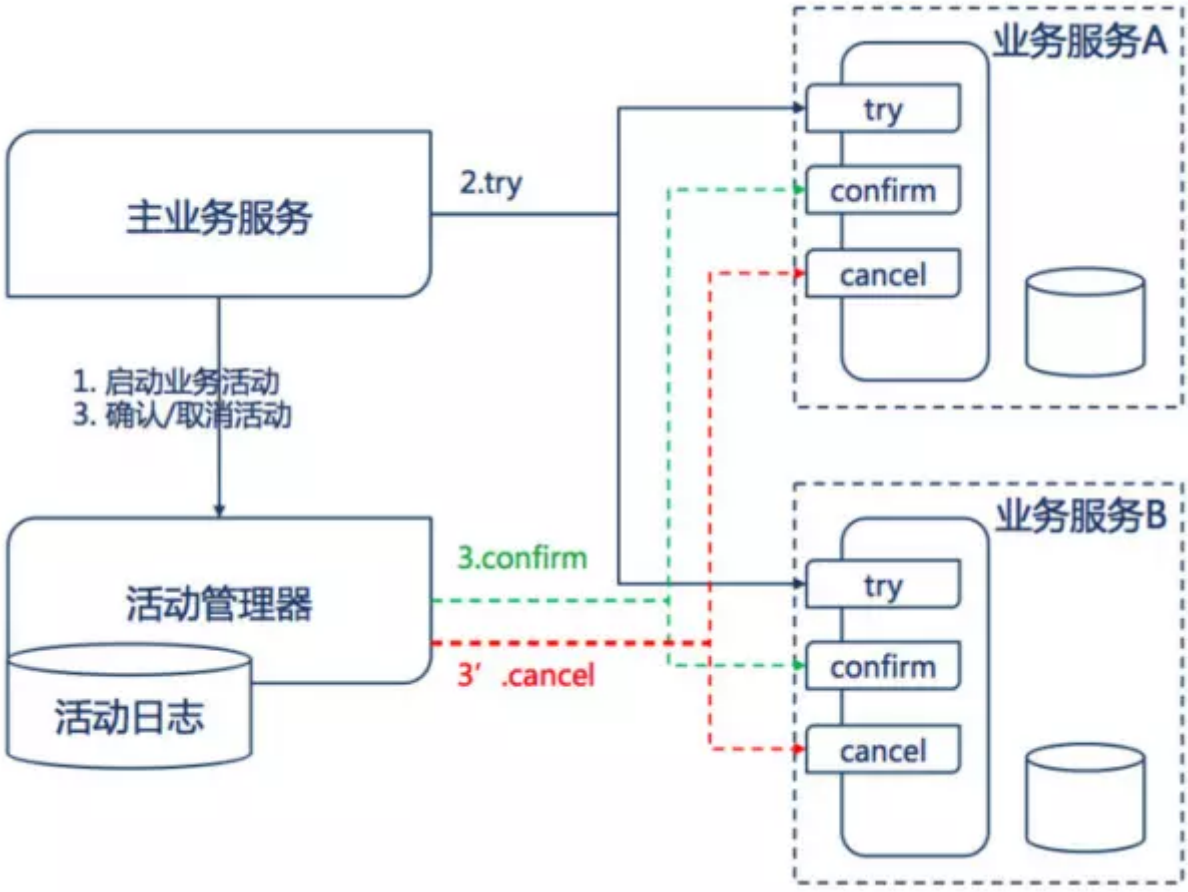
2. Confirm：真正执行业务

   不作任何业务检查；只使用Try阶段预留的业务资源；Confirm操作满足幂等性；

3. Cancel：

   释放Try阶段预留的业务资源；Cancel操作满足幂等性；

整个TCC业务分成两个阶段完成：



第一阶段：主业务服务分别调用所有从业务的try操作，并在活动管理器中登记所有从业务服务。当所有从业务服务的try操作都调用成功或者某个从业务服务的try操作失败，进入第二阶段。

第二阶段：活动管理器根据第一阶段的执行结果来执行confirm或cancel操作。如果第一阶段所有try操作都成功，则活动管理器调用所有从业务活动的confirm操作。否则调用所有从业务服务的cancel操作。

TCC模式的详细描述可以参见同事田向阳的技术文章《微服务架构下数据一致性保证（三）》http://dwz.cn/3TVJaB。

需要注意的是第二阶段confirm或cancel操作本身也是满足最终一致性的过程，在调用confirm或cancel的时候也可能因为某种原因（比如网络）导致调用失败，所以需要活动管理支持重试的能力，同时这也就要求confirm和cancel操作具有幂等性。

总结

六种分布式事务的实现模式从数据一致性、事务级别、吞吐量、实现的复杂度各有优劣，下图为大家提供选择依据。

	两阶段	三阶段	Sagas长事务	补偿	可靠事件	TCC
一致性	强一致性	强一致性	最终一致	最终一致	最终一致	最终一致
事务	全局	全局	全局	全局	全局	全局
吞吐量	弱	弱	高	中	高	中
实现复杂度	易	易	难	中	难	难

站在架构设计的角度，针对数据一致性需要把业务因素考虑进来，这有利于团队在技术上作出更合理的选择。根据具体业务场景，评估出业务对事务的优先级，更有利于作出架构上的取舍。我们经常接触的证券、金融、支付等行业，对数据一致性要求极高，需要严格的实时保证要求；但对于基于社交类的应用场景，可以采用局部实时一致，最终全局一致的能力。因此大家在实践过程中，一定要把技术与业务结合，选择适合自身业务的技术方案。

作者介绍

刘相，来自普元，十年IT行业经验，专注于企业软件平台。在SOA、分布式计算、企业架构设计等领域有一定了解。著有《SpringBatch批处理框架》一书。

欢迎扫描二维码加入由刘相主持的“普元云计算研发开放群”，讨论更多微服务、DevOps、容器相关内容，加群暗号“事务”。