微服务架构下的分布式事务处理

方 意 朱永强 宫学庆
(华东师范大学计算机与软件工程学院 上海 200062)

摘 要 单体架构下的分布式事务是一个服务内访问多个数据源的分布式事务,可以采用传统分布式事务处理模型——DTP(Distributed Transaction Processing)模型来解决。在微服务的架构下,可能会出现跨服务、跨资源的分布式事务。在解决这类分布式事务时,微服务追求系统的可用性和最终一致性而非数据的强一致性。针对不同的微服务分布式事务场景,介绍不同的分布式事务处理模型,包括可靠消息模型、业务补偿模型和 TCC(Try-Confirm/Cancel)模型,并总结每种模型的处理流程和优缺点。对 TCC 模型在性能上和可移植性上提出优化方案。

关键词 微服务 分布式事务 DTP 可靠消息 业务补偿 TCC

中图分类号 TP3 **文献标识码** A DOI:10.3969/j.issn.1000-386x.2019.01.028

DISTRIBUTED TRANSACTION PROCESSING UNDER MICROSERVICE ARCHITECTURE

Fang Yi Zhu Yongqiang Gong Xueqing
(*School of Computer Science and Software Engineering, East China Normal University, Shanghai 200062, China*)

Abstract The distributed transaction under the monolithic architecture can access multiple data resources within a service. It can be solved by using the traditional distributed transaction processing (DTP) model. Cross-service and cross-resource distributed transactions may occur under the microservice architecture. When solving such distributed transactions, microservices pursue system availability and eventual consistency rather than strong consistency of data. For different microservices distributed transaction scenarios, different distributed transaction processing models were introduced, including reliable message model, business compensation model and Try-Confirm/Cancel (TCC) model. And the process flow and advantages and disadvantages of each model were summarized. We proposed optimization scheme for TCC model in terms of performance and portability.

Keywords Microservice Distributed transaction DTP Reliable message Business compensation TCC

0 引 言

单体应用架构在规模较小的情况下可以很好地满足业务需求,但随着互联网技术的发展,系统规模的持续扩大,单体架构暴露的问题也越来越多。一方面代码量大,逻辑复杂,不利于维护,更新某一个小模块需要重启整个项目;另一方面,不利于项目的横向扩展和按需伸缩。微服务[1]架构可以将不同模块独立开来,使得各个模块之间更加的松耦合。各个模块可以独立部署、运行与更新,可以更加灵活的扩展与维护。此外,微服务与云计算天然契合,使得微服务架构被广泛讨论与采用。

随着微服务概念的兴起,如何在微服务架构下实施分布式事务[2]是一个值得探讨的问题。早期事务的概念一般局限于资源层面,不管是单机事务还是分布式事务,都是交给资源层去做。然而在业务开发阶段,事务的概念上升到应用层。应用层的事务场景有三种,如图 1 所示(AP 表示应用程序,RS 表示资源)。单体架构下的事务包含图 1 中的(a)和(b)两种,即单服务内访问单个数据资源的本地事务和单服务内访问多个数据资源的分布式事务;微服务架构下的分布式事

收稿日期:2018-07-01。**方意**,硕士,主研领域:分布式数据库与 Web 应用开发。**朱永强**,硕士。**宫学庆**,教授。

务除了这两种类型外还会存在更复杂的情况,即图 1 的(c)所描述的跨服务、跨资源的分布式事务。

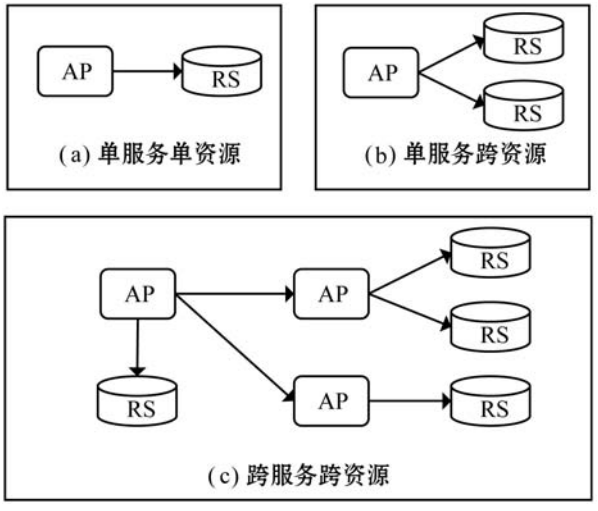

图 1　应用层事务场景

解决应用层的分布式事务,较为经典的方案是 DTP 模型[3]。DTP 模型延续了人们对以往分布式事务的理解,将业务层的分布式事务交由资源层处理,应用层不需要关注事务的执行流程。DTP 模型只能解决图 1 中单服务、跨资源场景,若要解决跨服务、跨资源的分布式事务,比较常见的方案有 TCC 模型、可靠消息模型和业务补偿模型。

1　传统分布式事务处理与微服务场景

1.1　X/Open DTP 模型

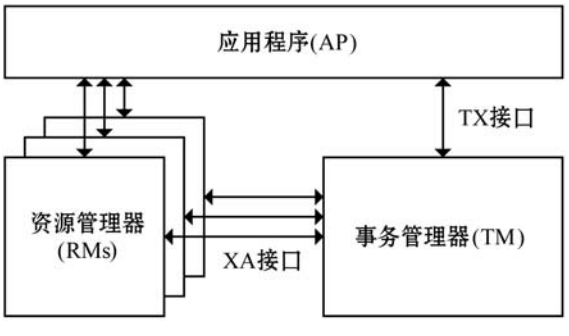

图 2　X/Open DTP 模型[3]

传统单体架构下的分布式事务处理一般采用 DTP 模型。DTP 模型由 X/Open 公司提出,也称 X/Open XA 协议。

应用程序(AP)通常是指单个的应用,在一个服务内访问一到多个资源。资源管理器负责管理一到多个资源域,每个资源管理器实例(RM)管理一个资源域,负责具体的资源操作。事务管理器通过与资源管理器交流来协调全局事务[3]。

首先,AP 通过 TX 接口向 TM 注册一个全局事务并告知 TM 需要操作哪些资源域。然后 TM 通过 XA 接口通知每个管理对应资源域的 RM 开启一个子事务。接着 AP 通过 RM 对资源进行操作,并根据执行结果通知 TM 提交或回滚全局事务。如果所有子事务全部执行成功,则提交全局事务,否则回滚全局事务,即 TM 通知所有 RM 回滚子事务。

1.2　DTP 模型的局限性

不难看出,DTP 模型使用 2PC(Two Phase Commit,两阶段提交)协议[4]来保证分布式事务的原子性和一致性。TM 充当全局事务协调者,RM 充当全局事务参与者。2PC 能够严格保证分布式事务的原子性和一致性,并且由于直接作用于资源层,对业务代码没有过多的侵入性,这使得 DTP 模型具有一定的普适性,满足大部分场景需求。

DTP 模型的缺点在于性能低下,由于事务的隔离性,2PC 一般采用基于锁的并发控制[5]来控制对数据的访问,这意味着资源将被锁定直至事务结束。如果一个分布式事务对非热点数据的访问时间过长,将严重影响对于热点数据的访问,降低系统的并发性能。

此外,就分布式事务应用场景而言,DTP 模型只适用于图 1 中的单服务、跨资源场景[6],不能有效解决跨服务、跨资源场景。而在微服务架构下,跨服务、跨资源分布式事务往往更加常见。

1.3　微服务架构下的分布式事务特性

ACID[7]是传统数据库中事务的设计理念,目的是保证数据的正确性,避免出现脏读、幻读等错误。但是在分布式系统中,尤其在应用层面,最重要的是满足业务需求,而非追求绝对的系统特性。根据 CAP[8]原理,强一致性、可用性和分区容错性不能同时满足。基于 CAP 原理的 BASE[9]理论采取了和 ACID 完全不同的设计思想,BASE 理论通过牺牲强一致性来换取高可用性,但可以通过合适的方法达到最终一致性,这符合现实生活中分布式领域的特点。在此基础上实施分布式事务,事务是在应用层执行的,不仅能够保证数据的最终一致性,也能获取很好的可用性。

在微服务架构下,跨服务、跨资源的分布式事务满足 CAP 原理,所以后面讨论的微服务架构下的分布式事务处理模型,都是在 BASE 理论下解决跨服务、跨资源分布式事务的处理模型。

2　微服务架构下分布式事务处理模型

在单体应用中,各个模块的调用是通过方法或函

数来实现的。而在微服务架构下,服务之间的交互必须通过服务间通信来解决[10]。常用的两种服务间通信机制:基于消息的异步通信和基于请求/响应的同步通信。不同的分布式事务场景可能涉及不同的服务间通信机制,因此需要不同的分布式事务处理模型来解决。

下面通过一个例子来描述现实生活中经常见到的几类分布式事务场景:周末,Tom 想去参加某明星在某地的演唱会,中午先去某家餐馆吃饭,在一个微服务架构平台通过订餐接口完成一次订餐操作;然后通过该平台的订票接口预订好下午的车票和晚上的演唱会门票。大致活动如图 3 所示。

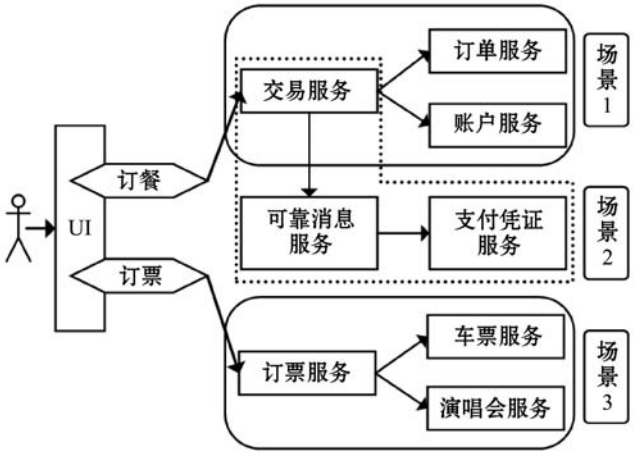

图 3 微服务架构下分布式事务案例

整个微服务架构平台共有 8 个服务(不包括 UI):交易服务完成支付操作,记录交易流水;订单服务为本次交易生成订单,检查原材料是否充足;账户服务记录商家和客户账户,完成转账操作;可靠消息服务包含一个消息中间件,完成消息接收和投递动作;支付凭证服务给消费者发送消费详细记录;订票服务可以预定演唱会门票、机票、酒店等;车票服务模拟火车订票,提供订票和退票两个接口;演唱会服务模拟演唱会订票和退票。整个过程包含三个场景,每个场景下的分布式事务对应于不同的解决方案。本文将以场景 1 为例介绍 TCC 模型,以场景 2 介绍可靠消息模型,以场景 3 介绍业务补偿模型。

2.1 TCC 模型

TCC[11]是 Try-Confirm-Cancel 的缩写,分别对应着三种操作:Try 操作、Confirm 操作和 Cancel 操作,在服务中分别以三种接口的形式存在。图 4 是 TCC 模型图和正确执行时的工作流。

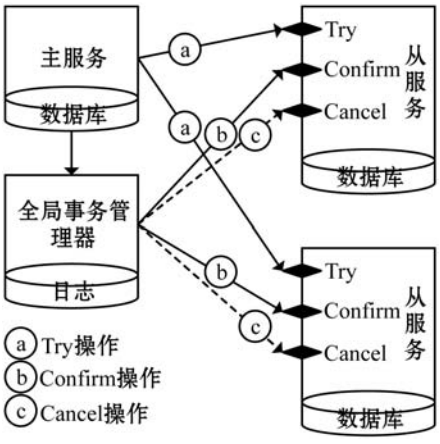

图 4 TCC 模型

Try 操作对业务进行检查,比如检查数据库资源是否充足,然后在业务层隔离业务活动需要的资源;Confirm 操作的前提是 Try 操作成功。这一步才是真正的执行业务操作,不需要检查资源情况,使用的就是 Try 操作预留的资源;Cancel 操作的前提是 Try 操作失败,释放 Try 操作预留的资源。

如图 4 所示,TCC 模型大致分为三个部分:主服务、从服务和全局事务管理器。服务与服务之间通过请求/响应的同步通信机制进行交互。主服务提供对外接口,接受客户端请求,发起一个全局的业务活动并编排所有的事务参与者。从服务是全局事务的参与者,提供 Try、Confirm 和 Cancel 三个接口,通过调用这些接口来使从服务完成分支事务。全局事务管理器是整个分布式事务的协调者,记录全局事务的执行日志和事务状态,并且在 Try 阶段完成后,根据结果成功与否调用从服务的 Confirm 接口或 Cancel 接口。全局事务管理器是一个单独的服务。

结合图 3 中的场景 1 和图 4 来描述 TCC 模型的执行流程,如图 5 所示。(1) 交易服务接受请求开始执行本地事务。(2) 交易服务向事务管理器申请一个分布式事务,并注册需要调用的从服务,即订单服务和账户服务。(3) 交易服务调用订单服务和账户服务的 Try 接口,在业务层锁定业务资源,比如订单服务会预留一份菜品原材料,账户服务会在 Tom 的账户中预留本次的交易额。(4) 当所有的 Try 接口操作成功(菜品原材料充足且 Tom 账户充足),交易服务提交本地事务;如果有 Try 操作失败(比如菜品原料不够或 Tom 账户余额不足),交易服务回滚本地事务。(5) 当所有的 Try 接口调用成功,主服务提交本地事务后,通过事务管理器调用从服务的 Confirm 接口,执行具体的业务逻辑,如扣除一份原材料,削减 Tom 账户余额,增加商家账户余额;若有 Try 接口调用失败,导致主服务本地事务回滚,事务管理器调用从服务 Cancel 接口解锁预留资源。(6) 当所有从服务完成 Confirm 操作或 Can-

cel 操作,分布式事务结束。

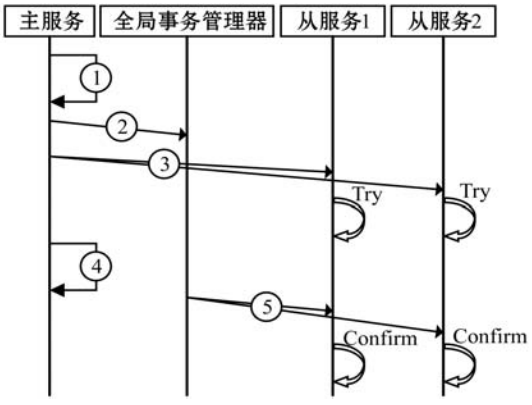

图 5　TCC 模型工作流

这里需要注意两个问题:一是具体设计时可以控制 Try 操作的超时时间,如果没有按时执行 Confirm 操作可以主动执行 Cancel 操作,将资源留给其他事务,这样可以提高服务的自治性;二是在执行 Confirm 操作或者 Cancel 操作时一定要保证幂等性,确保由于任何原因导致多次调用这两个接口时所产生的结果必须和调用一次产生的结果相同,这样才可以安全地实施重试策略。

TCC 模型也是通过两阶段提交协议来保证分布式事务的原子性。在隔离性方面,DTP 模型采用 2PL[12](两阶段封锁协议)来控制并发。与 DTP 模型不同的是,TCC 是在业务层上锁。Try 阶段完成后,隔离本次所需资源,释放掉数据库层面的锁,其他事务可继续以同种方式操作数据库的余下可用资源,这样就减少上锁时间,提高并发性。一致性方面,TCC 模型允许短时间内的不一致。比如 Tom 在付款后商家可能过一会才收到消费金额,但这在一定程度上可以容忍的。

TCC 模型适合于当前的各种微服务框架,业务层的编码可以灵活控制事务。但也正因为如此,TCC 模型需要为正常的业务逻辑添加事务属性来满足分布式事务的处理要求,代码侵入性大,可移植性低,开发成本高。

2.2　可靠消息模型

可靠消息模型的最大特点在于借用了消息中间件来完成分布式事务,天然具备最终一致性的思想。图 6 是可靠消息模型图和正确执行时的工作流。

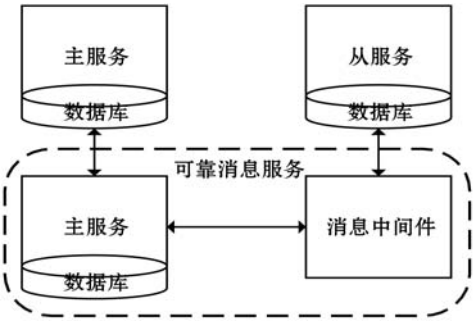

图 6　可靠消息模型

图 6 中,主服务不会直接与从服务交互,中间通过可靠消息服务解耦。从服务不影响主服务的事务处理,只是被动地接受主服务事务处理的结果。服务与服务之间通过消息的发布与订阅进行异步通信。

图 7 中①到⑥步是主服务向可靠消息服务发送消息阶段(生产阶段):(1) 主服务向可靠消息服务发送消息,状态为“待生产”。(2) 可靠消息服务持久化该消息。(3) 向主服务返回消息持久化结果(成功与否)。(4) 主服务处理本地事务(当步骤③返回成功)。(5) 将本地事务处理结果发送至可靠消息服务。(6) 根据主服务的处理结果修改消息状态为“待消费”(当步骤⑤成功)或删除消息(当步骤⑤失败)。

只有①到⑥步顺利执行,即第⑥步成功将消息状态修改为“待消费”,才会执行⑦到⑩步。⑦到⑩步是可靠消息服务向从服务投递消息阶段(消费阶段):(7) 可靠消息服务向从服务投递消息。(8) 从服务执行本地事务。(9) 从服务向可靠消息服务发出确认信息(步骤⑧执行成功)。(10) 可靠消息服务将这条消息删除。

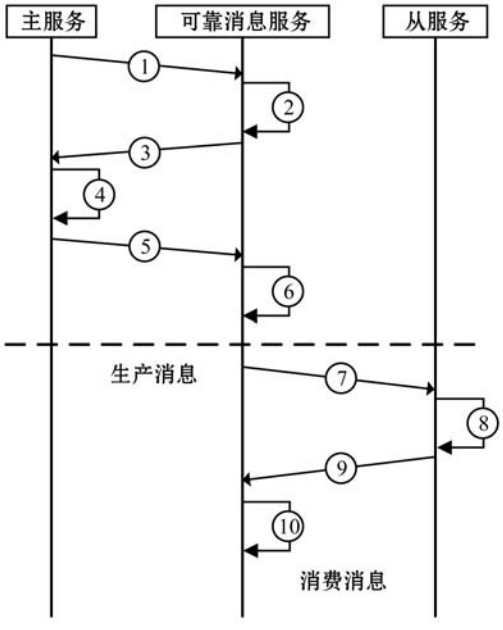

图 7　可靠消息模型工作流

可靠消息模型中,从服务的处理不会影响主服务,分布式事务的一致性体现在如果主服务成功执行,一定要向从服务成功地投递消息,从服务接收到消息后一定准确地完成自身的业务逻辑。结合图 3 中的场景 2,一旦交易确定,Tom 已完成支付操作,那么他一定要收到支付凭证服务给他发送的支付凭证账单。如果 Tom 由于网络原因没有付款成功,则一定不能收到支付凭证,但支付凭证服务对 Tom 的付款操作没有影响,是一个被动的操作。

可靠消息模型中分布式事务的原子性可以细化为

两阶段:一是主服务完成业务逻辑后,必须将结果持久化到可靠消息服务里;二是可靠消息服务根据完成结果进行操作,如果失败,删除消息,如果成功,必须将消息投递到从服务中,并等待从服务的执行结果。

然而,常见的中间件一般只负责接收消息和投递消息[13],不能正确地完成第一阶段(生产阶段)。因为不论是主服务先完成业务而后向中间件发送消息,还是先向中间件发送消息而后执行业务,都可能存在异常导致两边信息不一致。所以需要对常规的中间件进行包装,添加一个额外的服务完成图 7 中的①到⑥步。其中①到③步必不可少,有了这三步才能够保证在各种异常情况下主服务的业务执行结果和可靠消息的消息持久化过程保持一致。第二阶段(消费阶段)大部分的中间件可以实现,异常情况下中间件可能会向从服务多次投递消息。因此,从服务的接口设计需要满足幂等性。若多次投递无效,说明从服务有问题,必要时需要人工介入。

图 6 中的可靠消息模型是单独做一个消息服务来管理消息,这样可以做到独立部署和维护,重用性高,降低业务逻辑和消息管理的耦合性。也可以将这个任务直接交给主服务来做,但是这样做的弊端就是代码入侵,将消息的可靠性保证与正常的业务逻辑混合在一起,可移植性低,每一个类似的场景都需要根据业务重新编码。

2.3 业务补偿模型

相对于 TCC 模型,业务补偿模型[14]的思想比较简单。一旦分布式事务某一步出错,可以利用回滚的思想将已经执行过的业务按照相反的逻辑执行一遍。业务补偿模型的架构和 TCC 模型架构类似,需要主从服务的共同参与。从服务的操作结果也影响主服务,也需要一个全局事务管理器。服务与服务之间通过请求/响应的同步通信机制进行交互。图 8 是业务补偿模型图,图 9 是正确执行时的工作流。

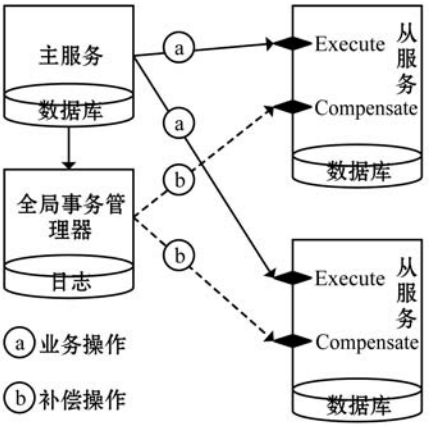

图 8 业务补偿模型

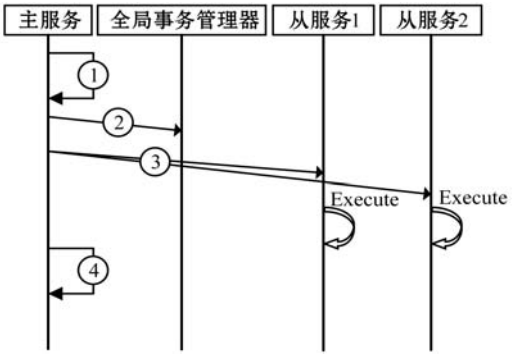

图 9 正确执行时的工作流

业务补偿模型中从服务的接口只有两个:Execute 接口和 Compensate 接口。不同于 TCC 模型,补偿机制事务处理只有一个阶段,即 Execute 阶段,所有的业务活动都在这一阶段完成。分布式事务的正常流程是由主服务调度的,只有在异常情况下才会通过事务管理器进行补偿操作。

结合例子,Tom 通过订票服务订票时是希望两张票能同时预定成功的,只买到门票或只买到车票该行程便无法成行。结合图 3 中的场景 3 可以描述业务补偿的执行过程如下:(1) 订票服务接收到请求后,开始本地事务。(2) 主服务通知事务管理器开始一个分布式事务,并注册需要调用的从服务。(3) 主服务分别调用各从服务的 Execute 接口,执行业务操作,事务管理器记录事务的活动状态。(4) 所有从服务成功完成子事务,主服务提交本地事务。如果出现异常,比如某一步订票失败,事务挂起,主服务回滚本地事务,并通过事务管理器开始执行补偿操作。(5) 所有从服务返回正确结果或者都补偿完成,分布式事务结束。

业务补偿模式是通过事务管理器记录分布式事务的处理流程的,记录的信息越详细,越有利于控制业务补偿的范围。需要注意的是,业务补偿模型中,从服务事务的执行并不是一定要按顺序执行的。如果服务之间存在依赖关系就按照顺序执行(但这违反了微服务的设计理念,服务之间耦合性太大)。如果服务之间不存在依赖(如图 3 场景 3 中预定门票和预定火车票是不存在依赖关系的),服务内的子事务可以按照顺序执行,也可以并发执行。顺序执行会影响效率,但是一旦出错,只需要对已经执行过的业务进行补偿就行。并发执行可以提高效率,但是一旦出错,需要对所有服务进行业务补偿,代价较高。

与 TCC 模型相比,业务补偿模型容易理解,并且代码入侵少,不需要改造正常的业务逻辑,只需要添加一个补偿逻辑即可。但也正因为如此,会有一定的代码冗余,所以业务补偿模型适合于业务逻辑简单的场景。如果业务逻辑繁杂,代码量大,补偿逻辑的实现将变得很复杂。另外一旦业务的正向逻辑发生修改,补

偿逻辑也需要做修改,将变得难以维护。就图 3 中的场景 3 而言,车票服务和演唱会服务只需要提供两个接口:订票和退票。正向流程调用订票接口,补偿流程调用退票接口。

业务补偿模型存在较为明显的两个缺点:一是重复补偿或补偿失败,重复补偿可以通过设置 Compensate 接口的幂等性解决。补偿失败也就是在正向操作失败时,执行补偿操作时再度失败。如果此时没有一个额外的自动处理机制的话,需要人工进行干预[13]。二是隔离性不好,业务补偿模型的事务处理只有一个阶段——Execute 阶段,这一阶段可能出现多个事务对同一资源进行操作的混乱局面。可以借用 TCC 模型的思想,在 Execute 阶段前加上一个 Try 阶段,预定本次事务所需要的资源。

3　模型总结对比

早期的事务是在资源层处理的,涉及到分布式事务时一般采用两阶段提交协议或者三阶段提交协议来解决。然而在微服务的架构下,事务的概念已经从资源层上升到应用层,事务的处理变得复杂多变,如果采用 2PC 或 3PC 来解决,将极大影响系统的性能。在分布式的环境中,不同的场景就要采取不同的分布式事务处理模型。图 1 中列出了微服务架构下的几种事务模型,图 2 给出了标准分布式事务的处理模型,其后给出了几种基于 BASE 理念设计的分布式事务处理模型。不同的模型解决不同场景下的分布式事务,通过表 1 可以清楚地看到各个模型的特性。

表 1　微服务架构下分布式事务处理模型对比

特性	DTP 模型	TCC 模型	可靠消息模型	业务补偿模型
设计思想	ACID	BASE	BASE	BASE
事务场景	单服务跨资源	跨服务跨资源	跨服务跨资源	跨服务跨资源
原子性	2PC	2PC		
一致性	强一致性	最终一致性	最终一致性	最终一致性
隔离性	高	高	高	低
模型代价	协议成本(包括 TX 协议和 XA 协议)	TCC 框架的设计成本(包括全局事务管理器的设计和接设计)	可靠消息服务的设计成本	补偿框架的设计成本(包括全局事务管理器的设计和接口设计)

续表 1

特性	DTP 模型	TCC 模型	可靠消息模型	业务补偿模型
优点	严格的 ACID 特性,保证数据的强一致性	在业务层针对业务需求灵活控制事务,并发性好并且保证了隔离性	代码入侵小,可以独立部署和维护,降低业务与消息的耦合性	业务入侵小,服务架构易于理解
缺点	资源封锁时间长,并发性能差	开发成本高,业务入侵大,可移植性低	消息处理流程复杂,严重依赖网络	代码冗余,隔离性差导致并发性能低
适用场景	并发要求不高,对数据一致性有严格要求	并发量大,一致性要求高,业务执行时间短	业务执行时间不确定,时效性要求不高	并发要求不高,业务简单易于回滚
实施难度	小	大	较大	较大

从表中可以看出,DTP 模型基于 ACID 理念设计,有成熟的框架,实施难度小。但性能较差,且在微服务中只适合于单服务内的多资源场景。另外三种模型基于 BASE 理念设计,适合于跨服务、跨资源场景,但没有具体的标准框架,需要根据业务需求具体定制。其中 TCC 模型由于与业务代码绑定较紧,可以灵活控制,性能较好,但实施难度大;可靠消息模型要借助中间件来完成,需要单独实现可靠消息服务;业务补偿模型不影响原有的业务逻辑,但需额外实现相反的业务逻辑操作,所以也需要根据具体的业务来实施。

现实的业务场景中往往是多种模型交织在一块。比如就本文的例子而言,图 3 中的场景 3 也是要经过一个交易付款流程的,所以也包含场景 1 和场景 2,不过为了简化分析将它单独剥离出来讨论。

4　模型优化与猜想

前面总结了每种模型较为通用的版本,实际上一些模型仍具有相当大的优化空间。比如业务补偿模型的隔离性较差,可以借鉴 TCC 模型的方法,先加资源锁来提高事务的隔离性[14]。TCC 模型采用两阶段的思想来解决分布式事务,与传统分布式事务处理方法两阶段提交协议的思想相近,实际项目中多采用 TCC 模型作为分布式事务解决方案。所以本节内容着重思考如何优化 TCC 模型。

对于 TCC 模型的优化要考虑两个方面:一是性能上的优化,二是可移植性低上的优化。

在 2.1 节介绍的 TCC 模型中,全局事务管理器是独立的,以此来充当两阶段过程中的事务协调者。当

发起一个分布式事务时,主服务和从服务需要多次与事务管理器进行交互,频繁的服务调用将影响性能。如图 10 所示,可以将事务管理器集成在主服务中,事务日志和数据可以在同一个数据库存储,这样可以减少事务提交或者回滚时主服务和事务管理器之间的服务间调用,而可以直接用方法调用。这种方案增加了编码难度,但是可以进一步提高 TCC 模型的性能,具有较好的可实施性。

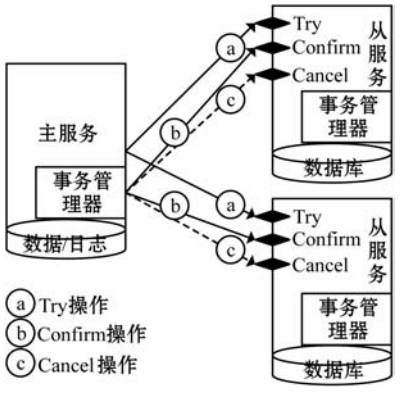

图 10 TCC 性能优化

从 2.1 节可以知道 TCC 模型的最大缺点在于可移植性较差,因为服务需要根据业务具体实现三个接口。不同的项目,不同的公司都需要重新编码实现 TCC 接口,不能做到很好的复用。这里给出一种思路,可以实现一个 TCC 框架,如图 11 所示。主服务和从服务不需要实现 Try、Confirm 和 Cancel 接口,只需要专注于自己的业务逻辑。服务之间是普通的调用。主服务通过全局事务管理器发起一个分布式事务,从服务开始各自的分支事务,在将数据写入数据库之前需要经过 TCC 框架处理。TCC 框架在内部实现 Try、Confirm 和 Cancel 的语义,当然此时的语义可能会发生改变。通过 TCC 框架将服务的业务逻辑与 TCC 操作分离,这样便可以将 TCC 框架作为一个平台复用到其他项目中,大大提高 TCC 模型的可移植性。当然这仅仅是一个猜想,如何实现 TCC 框架仍需要进一步的研究与探索。

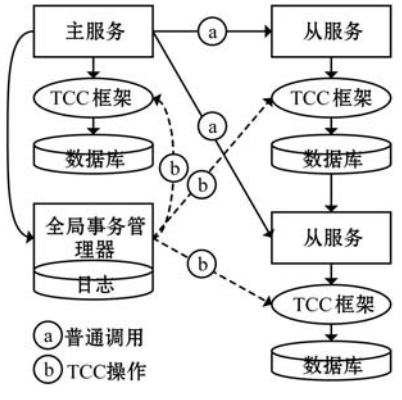

图 11 TCC 可移植性优化

5 结 语

近年来,微服务架构以其众多优点在互联网应用中被广泛采用,但分布式事务是一个不能回避的问题。如何在微服务架构下实施分布式事务具有较大的现实意义。鉴于此,本文分析了微服务架构下的事务模型和分布式事务场景,介绍了各个不同场景下比较有启发意义的分布式事务处理模型,并且对各种模型的特性进行了详细的解读。分布式事务是分布式领域的一个重要问题,现实生活中的场景往往更加复杂,需要做更加深入的研究才能在项目选型时做出最好的选择。

参 考 文 献

[1] Namiot D, Sneps-Sneppe M. On Micro-services Architecture [J]. International Journal of Open Information Technologies, 2014, 2(9):24 - 27.

[2] Surhone L M, Tennoe M T, Henssonow S F. Distributed Transaction[M]. Betascript Publishing, 2010.

[3] Ltd C X C. X/Open CAE specification: distributed transaction processing: CPI-C specification, version 2[M]. Prentice-Hall, Inc. 1996.

[4] Lampson B, Lomet D B. Distributed transaction processing using two-phase commit protocol with presumed-commit without log force: US, US5335343[P]. 1994.

[5] Bernstein P A, Hadzilacos V, Goodman N. Concurrency control and recovery in database systems[J]. 1987, 24(2): 1058 - 1077.

[6] Iwaniak M, Khadzhynov W. Colored Petri net model of X/Open distributed transaction processing environment with single application program[M]//Beyond Databases, Architectures, and Structures. Springer International Publishing, 2014:20 - 29.

[7] Gray J, Reuter A. 事务处理概念与技术[M]. 北京: 机械工业出版社, 2004.

[8] 陈明. 分布系统设计的 CAP 理论[J]. 计算机教育, 2013 (15):109 - 112.

[9] Pritchett D. BASE: an acid alternative[J]. ACM,2008, 6 (3):48 - 55.

[10] 蒋勇. 基于微服务架构的基础设施设计[J]. 软件, 2016, 37(5):93 - 97.

[11] Pardon G, Pautasso C. Atomic distributed transactions: a RESTful design [C]//International Conference on World Wide Web. ACM, 2014:943 - 948.

[12] Thomasian A. Two-phase locking performance and its thrashing behavior[J]. Acm Transactions on Database Systems, 1993, 18(4):579 - 625.

[13] 周世杰, 刘锦德, 秦志光. 消息队列技术研究:综述与一个实例[J]. 计算机科学, 2002, 29(2):84 - 86.

[14] 李晓珍,刘迪,王孟强,等. 微应用架构下分布式事务的处理方法[C]//2016 电力行业信息化年会论文集. 2016.