事务

四大特性

原子性

原子性是数据库事务不可分割的工作单位。只有事务中所有的数据库操作都成功就算成功,其中一个失败就失败,要么都成功,要么都失败。比如银行的取款流程:

- 登陆ATM机平台,验证密码
- 从远程数据库获取账户信息
- 更新远端数据库
- ATM出款

如果中间任何一个步骤出错了,所有的操作就会失败,如果全部成功才算成功。

一致性

一致性指事务将数据库从一种状态转为下一种一致的状态。比如银行会员系统的用户ID是唯一约束,不能重复,如果一个事务对已有的id修改或者插入一个新的ID,无论事务的提交或者回滚都要保证这个键的唯一约束。

隔离性

隔离性指的是所有事务之间的操作互不影响,一个未提交事务的操作对另一个事务的操作不可见。当前数据库系统中都采用了一种粒度锁的方案,允许事务仅锁住一个实体对象的子集,以此来提高事务的并发度。

持久性

事务一旦提交,其结果就是永久性的。即使发生宕机,数据库也要能恢复,即使磁盘损坏,数据库也能将数据恢复。持久性保证的是数据库的高可靠性,而不是高可用性。

事务的分类

扁平事务

扁平事务是事务中最简单的一种,保证事务中间的操作是原子性的要么都执行,要么都回滚,主要有以下三种情况:

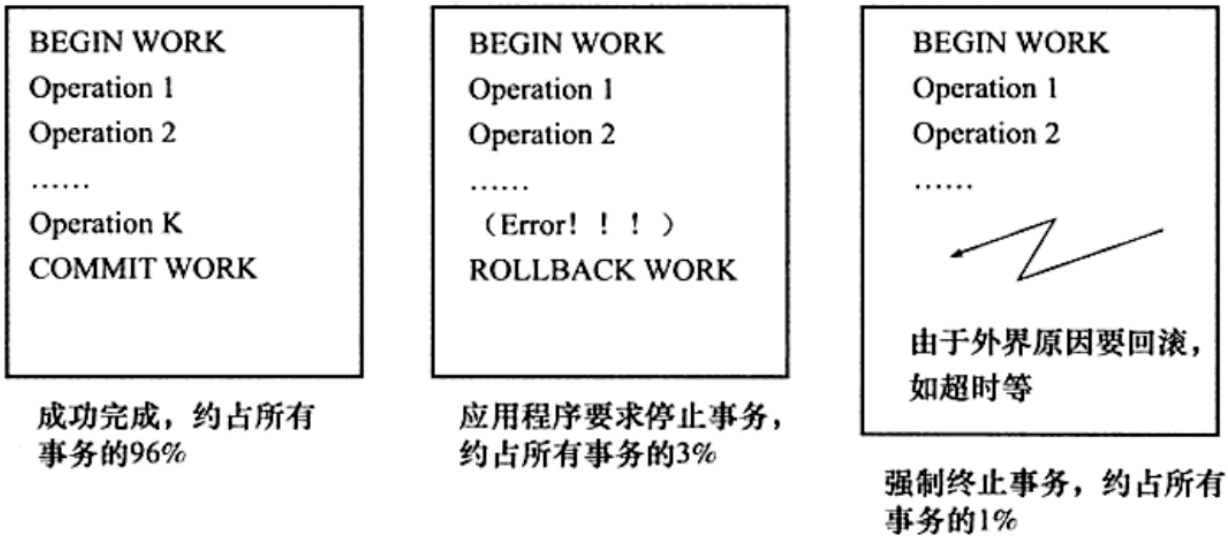


图 7-1　扁平事务的三种情况

带有保存点的扁平事务

顾名思义,带有保存点的扁平事务就是在扁平事务的基础上支持事务执行过程中回滚到同一个事务较早的一个状态(保存点)。这是为了解决扁平事务一次性回滚所有操作这种情况,有些时候只需要回退一部分步骤,然后进行再继续执行操作。

带有保存点的扁平事务可以带有多个保存点,所有的保存点是递增的,当出现问题的时候,根据应用逻辑将事务

回滚到最近的一个保存点还是更早的保存点,具体的例子如下:

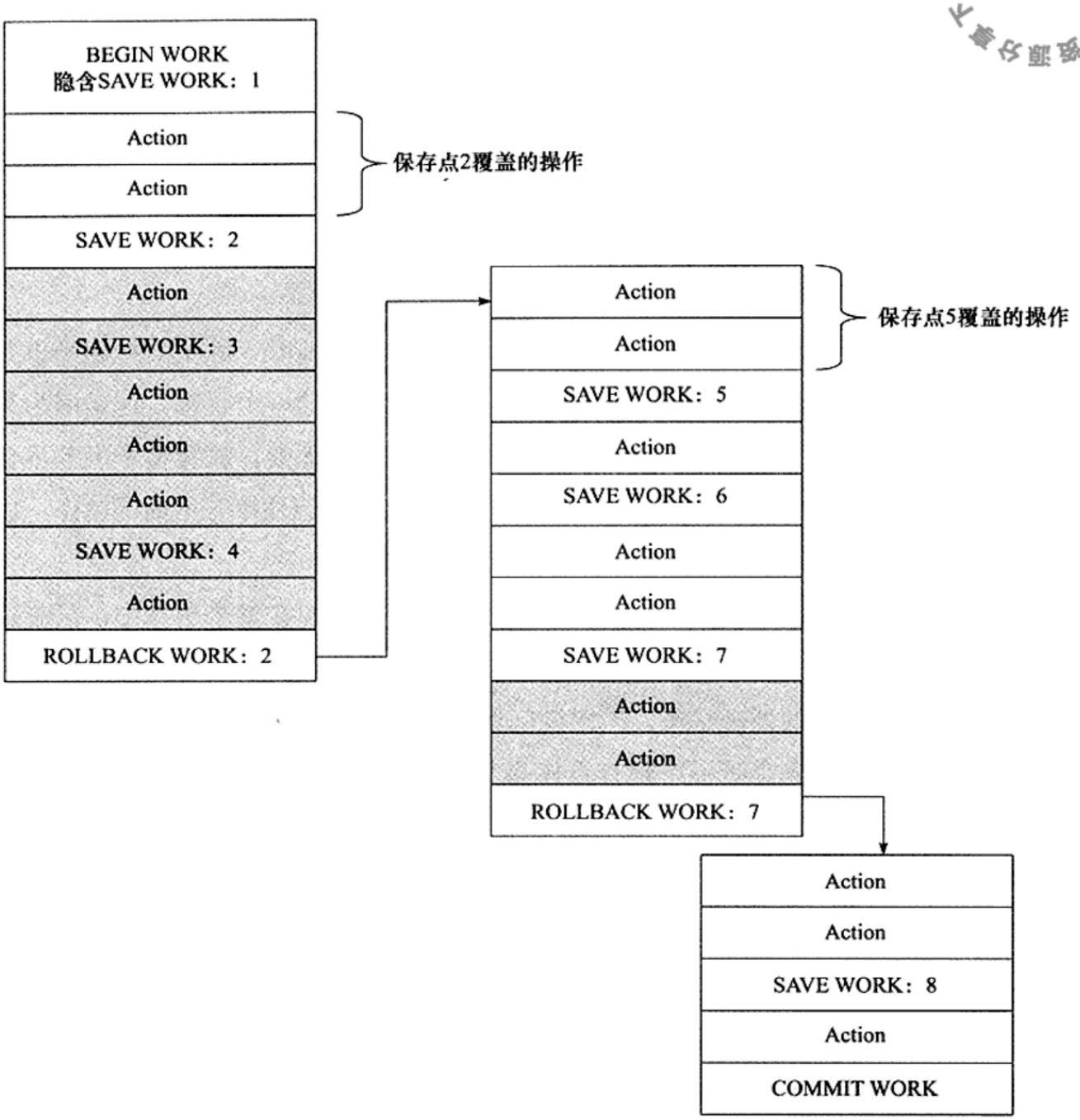


图 7-2　在事务中使用保存点

1. 开启一个事务(默认保存了一个保存点1)
2. 执行两个Action
3. 设置保存点(保存点序号为2)
4. 执行一个Action
5. 设置保存点(保存点序号递增为3)
6. 执行三个Action
7. 设置保存点(保存点序号递增为4)
8. ROLLBACK WORK:2表示回滚到保存点2则保存点2以后的所有请求都不允许被执行
9. 执行两个Action
10. 设置保存点(保存点序号递增为5)
11. 执行一个Action
12. 设置保存点(保存点序号递增为6)

> 13. 执行两个Action
> 14. 设置保存点(保存点序号递增为7)
> 15. 执行两个Action
> 16. ROLLBACK WORK:7
> 17. 执行两个Action
> 18. 设置保存点(保存点序号递增为8)
> 19. 执行一个Action
> 20. COMMIT 事务

链事务

链事务可以看作是带有保存点的扁平事务的一个变种,带有扁平事务当发生系统崩溃的时候,所有的保存点都会消失,因为保存点事易失的,并非持久的,所以系统崩溃后需要恢复则需要从头开始执行。

链事务的思想是:在提交一个对象的时候释放不需要的数据对象,将必要的上下文隐式传给下个对象。链事务和带有保存点的扁平事务还有不同的是链事务只能回退到上一步,而带有保存点的扁平事务可以回退到任意点。执行流程如下:

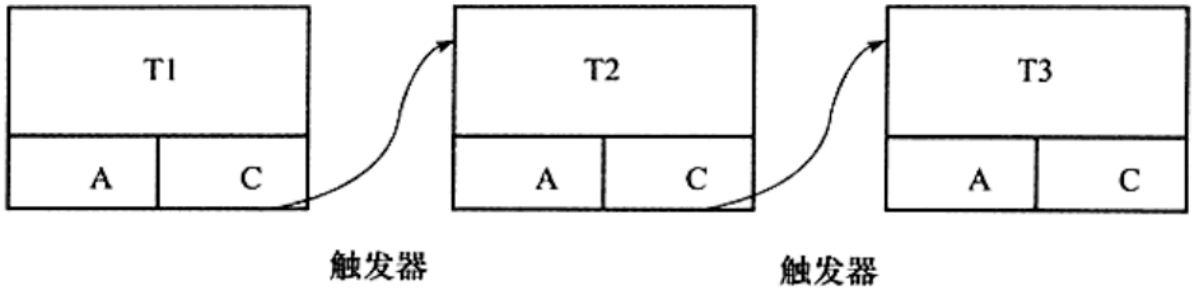


图 7-3　链事务的开始，第一个事务提交触发第二个事务的开始

嵌套事务

嵌套事务是一个层次结构模型(多叉树),由一个根节点控制所有子节点的事务。因为一个事务下可能由两个并行的操作,因此一个事务下面可能同时有多个子事务执行,而这链式事务无法满足这种情形。其层次结构如下

图:

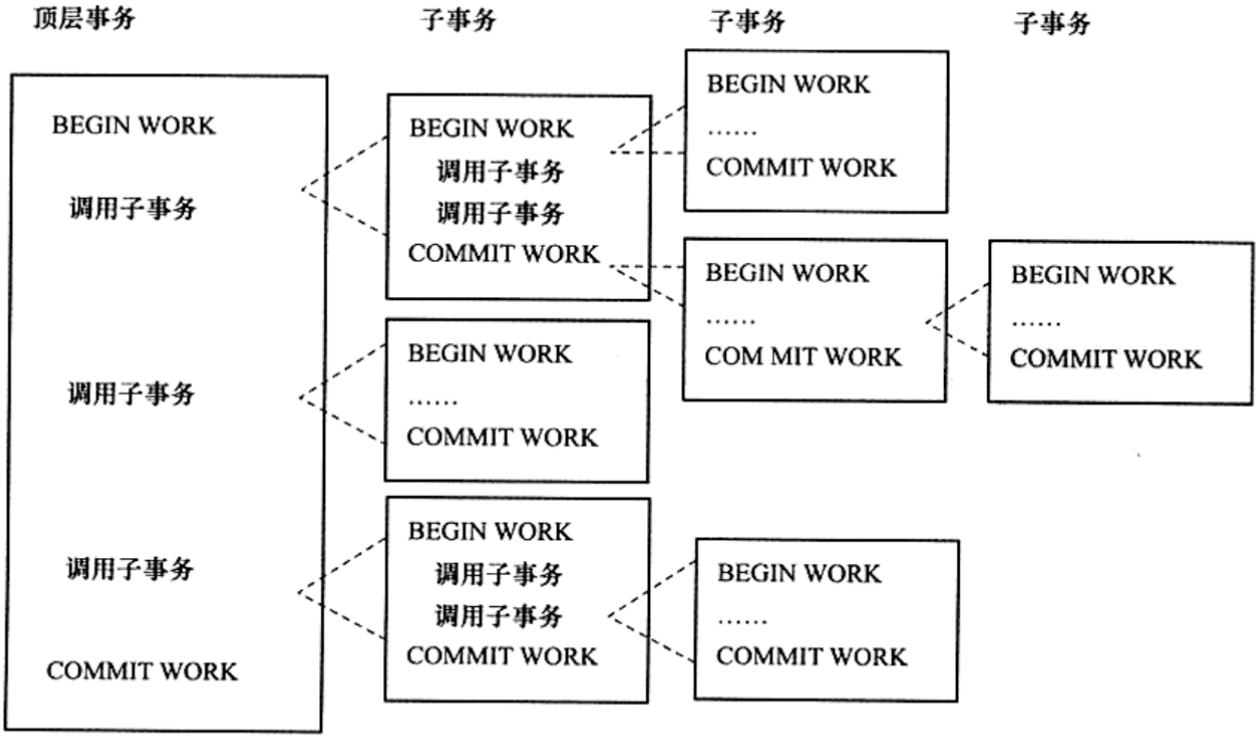


图 7-4 嵌套事务的层次结构

分布式事务

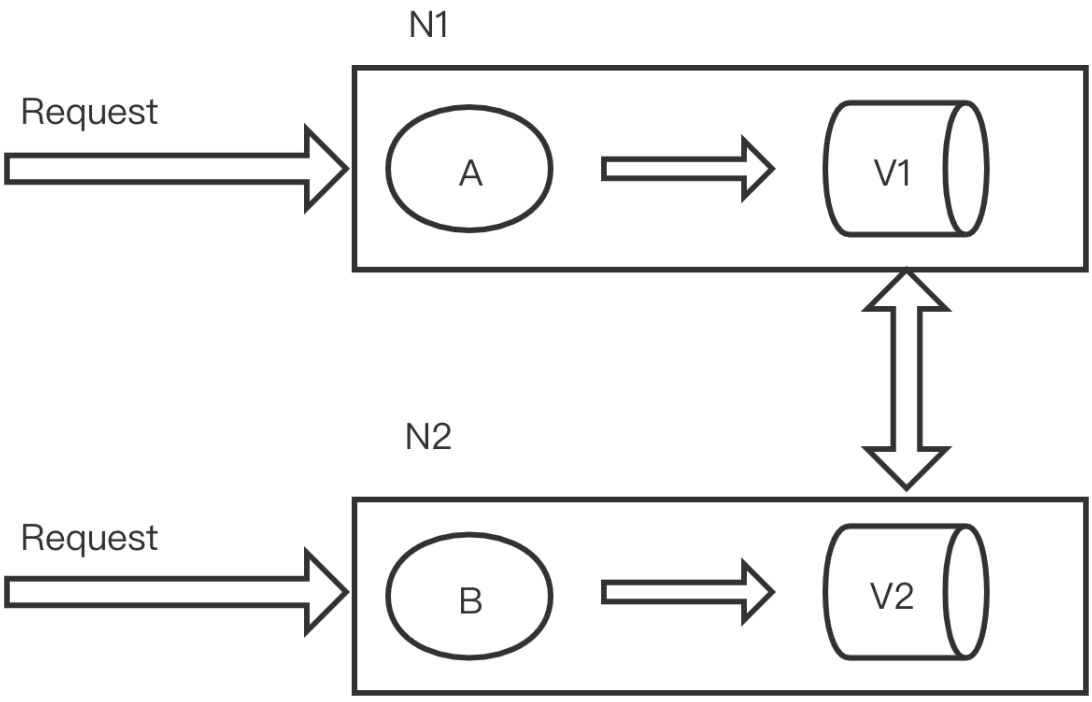
分布式事务就是事务的多个操作分布在不同的机器上面,也称为多机事务。

CAP 定理

CAP定理主要是介绍一个分布式的系统最多只能满足下面三个特性中的两个,不可能全部都满足。

> - C(Consistency):一致性,所有节点访问同一份最新的数据副本,即使在数据更新和删除之后
> - A(Available):可用性,每次请求都能获取到非错的响应——但是不保证获取的数据为最新数据
> - P(Partition Tolerance):分区容忍性,分布式系统在遇到任何网络分区故障的时候,仍然能够对外提供满足一致性和可用性的服务,除非整个网络环境都发生了故障。

CAP定理论证,一个简单的分布式系统如下:



该系统有两个服务端N1、N2,而V1、V2之间是网络互通的,如果请求到N1服务则会通过A服务去修改V1db,而V1会将数据同步给V2。

> 1. 要满足C一致性,则需要V1的数据 = V2的数据
> 2. 要满足A可用性,则无论访问N1或者N2都是可以得到正确的响应
> 3. 要满足P分区容错性,则N1和N2有任何一方宕机,或者网络不通的时候,都不会影响N1和N2彼此之间的正常运作。

那么如果V1和V2之间的数据同步出了问题,怎么保证一致性和可用性?

> 1. 牺牲数据一致性,保证可用性。响应旧的数据V1给用户。
> 2. 牺牲可用性,保证数据一致性。阻塞等待,直到网络连接恢复,数据更新操作M完成之后,再给用户响应最新的数据V2。

CAP如何取舍?

> - CA without P:如果不要求P(不允许分区),则C(强一致性)和A(可用性)是可以保证的。但其实分区不是你想不想的问题,而是始终会存在,因此CA的系统更多的是允许分区后各子系统依然保持CA。
> - CP without A:如果不要求A(可用),相当于每个请求都需要在Server之间强一致,而P(分区)会导致同步时间无限延长,如此CP也是可以保证的。很多传统的数据库分布式事务都属于这种模式。
> - AP without C:要高可用并允许分区,则需放弃一致性。一旦分区发生,节点之间可能会失去联系,为了高可用,每个节点只能用本地数据提供服务,而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

BASE 理论

BASE是Basically Available(基本可用),Soft State(软状态)和Eventually Consistent(最终一致性)三个短语的缩写。既是无法做到强一致性(Strong consistency),但每个应用都可以根据自身的业务特点,采用适当的方式来使系统达到最终一致性(Eventual consistency)。

- Basically Available

> 1. 响应时间上的损失:正常情况下的搜索引擎0.5秒即返回给用户结果,而基本可用的搜索引擎可以在2秒作用返回结果。
> 2. 功能上的损失:在一个电商网站上,正常情况下,用户可以顺利完成每一笔订单。但是到了大促期间,为了保护购物系统的稳定性,部分消费者可能会被引导到一个降级页面。

- Soft State

> 相对于原子性而言,要求多个节点的数据副本都是一致的,这是一种“硬状态”。 软状态指的是:允许系统中的数据存在中间状态,并认为该状态不影响系统的整体可用性,即允许系统在多个不同节点的数据副本存在数据延时。

- Eventually Consistent

> - 因果一致性（Causal consistency）因果一致性指的是:如果节点A在更新完某个数据后通知了节点B,那么节点B之后对该数据的访问和修改都是基于A更新后的值。于此同时,和节点A无因果关系的节点C的数据访问则没有这样的限制。
> - 读己之所写（Read your writes）读己之所写指的是:节点A更新一个数据后,它自身总是能访问到自身更新过的最新值,而不会看到旧值。其实也算一种因果一致性。
> - 会话一致性（Session consistency）会话一致性将对系统数据的访问过程框定在了一个会话当中:系统能保证在同一个有效的会话中实现 “读己之所写” 的一致性,也就是说,执行更新操作之后,客户端能够在同一个会话中始终读取到该数据项的最新值。
> - 单调读一致性（Monotonic read consistency）单调读一致性指的是:如果一个节点从系统中读取出一个数据项的某个值后,那么系统对于该节点后续的任何数据访问都不应该返回更旧的值。
> - 单调写一致性（Monotonic write consistency）单调写一致性指的是:一个系统要能够保证来自同一个节点的写操作被顺序的执行。

分布式事务的解决方案

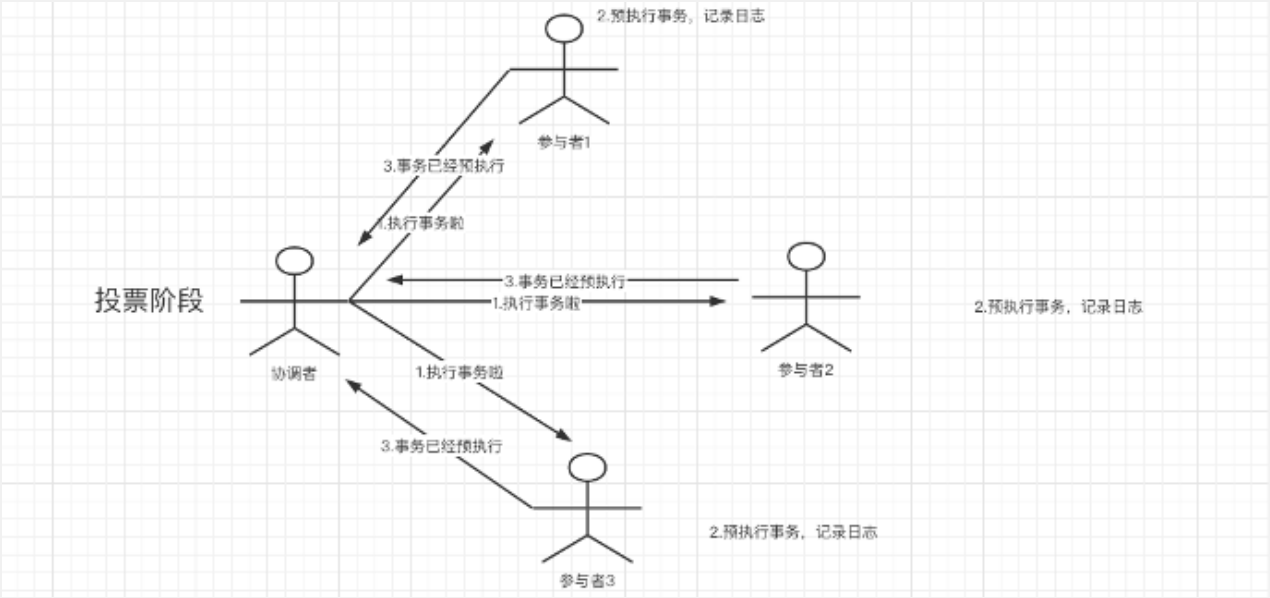
两段式提交

两段式提交主要分为两个投票和提交两个部分；角色有参与者和协调者,参与者是真正操作数据库单点事务的服务(多个),协调者主要是决定各个事务提交的中间服务(1个)。其中体的执行过程如下:

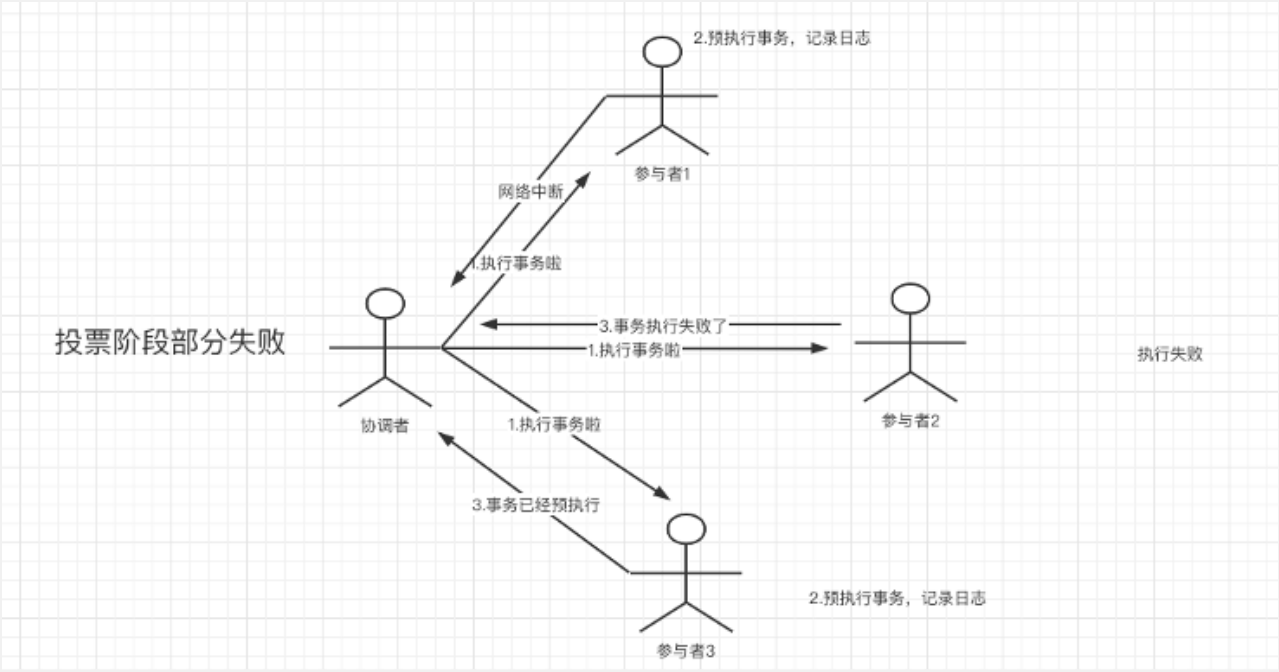
- 投票

> 投票过程主要分为下面三步:
>
> 1. 协调者向所有参与者发送事务执行请求,并等待参与者反馈事务执行结果
> 2. 其次参与者收到请求之后,执行事务但不提交,并记录事务日志
> 3. 最后参与者将执行结果返回给协调者

投票成功示意图如下:



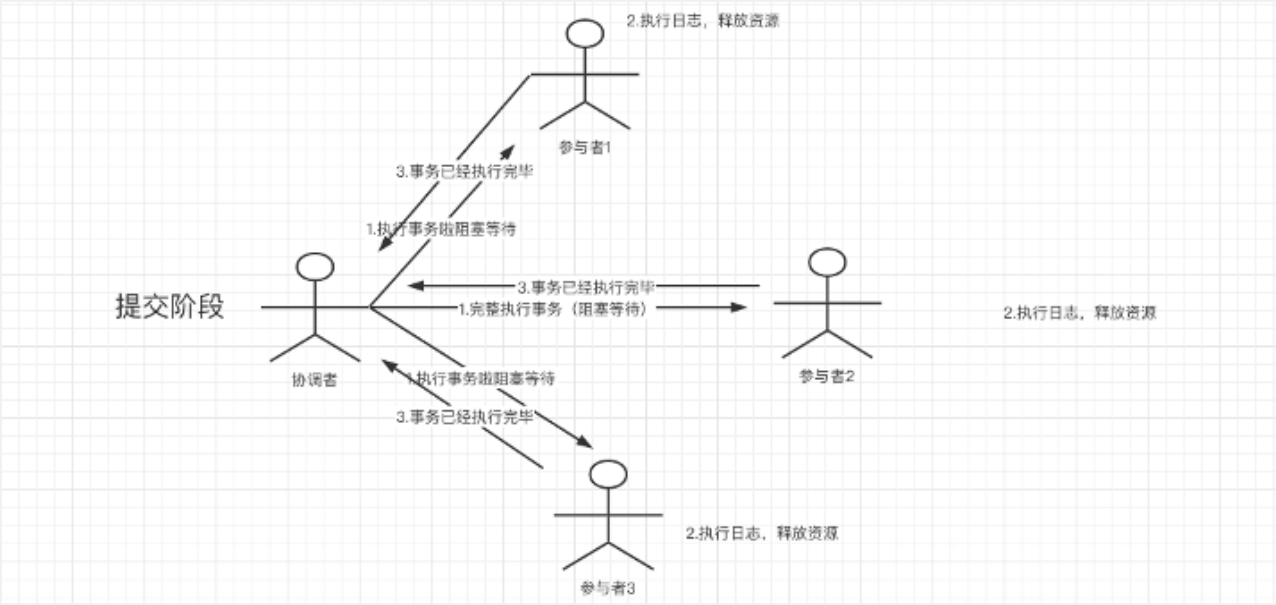
投票部分失败示意图如下:



- 提交

提交阶段会根据投票结果做不同的操作

1. 如果参与者都回复正常则给所有参与者发送提交commit请求,请求提交事务,参与者收到通知后执行commit操作释放所有资源,给协调者返回事务commit信息
2. 如果一个或者多个消费者失败了,协调者给所有参与者发送rollback通知,请求回滚,参与者收到实物回滚通知后执行rollback操作,释放资源；给协调者返回
3. 如果超时了操作同2

> 提交示意图如下:
>
> 
>

提交阶段会出现和投票阶段相同的三种情况,如果提交成功则事务顺利执行,如果提交失败或者协调者宕机则会引入以下几个问题:

> - 单点问题:如果协调者服务挂了就会影响整个集群的正常运行（这个可以通过watchdog解决）。
> - 同步阻塞:所有参与者都由协调者调度,等待响应的期间,参与者处于阻塞状态不能执行其他操作,效率上达不到很好的效果（采用超时机制可以避免长时间等待阻塞）
> - 数据不一致:提交阶段一部分参与者接收到提交指令,一部分没有收到则会产生数据不一致（互询机制可以解决这个问题） 但是存在一种情况是无法避免的,协调者在通知一个参与者后,协作者和参与者同时挂了,这种情况无法确定当初协作者的请求是提交还是回滚。

三段式提交

三段式操作和二段式操作类似,在二段式的基础上增加了一个预询盘阶段,所以总共有以下三个阶段:预询盘(can commit) ,预提交(pre_commit),事务提交阶段。

- 预询盘阶段

> 1. 协调者会询问参与者是否可以执行事务操作并等待回复
> 2. 参与者会根据自身情况确认是否能否进入预备状态看,如果可以则进入预备状态并且返回确认信息,否则返回否定信息。

- 预提交阶段

> 1. 如果参与者都确认则执行与二段式预提交阶段一致的操作
> 2. 如果参与者超时或者等待请求超时则给所有参与者发送abort指令,请求推出预备状态。

- 事务提交阶段

> 同二段式预提交类似,但是等待超时会默认commit操作,可以解决二段式的等待超时阻塞问题,但是还是无法避免数据的不一致。

XA事务

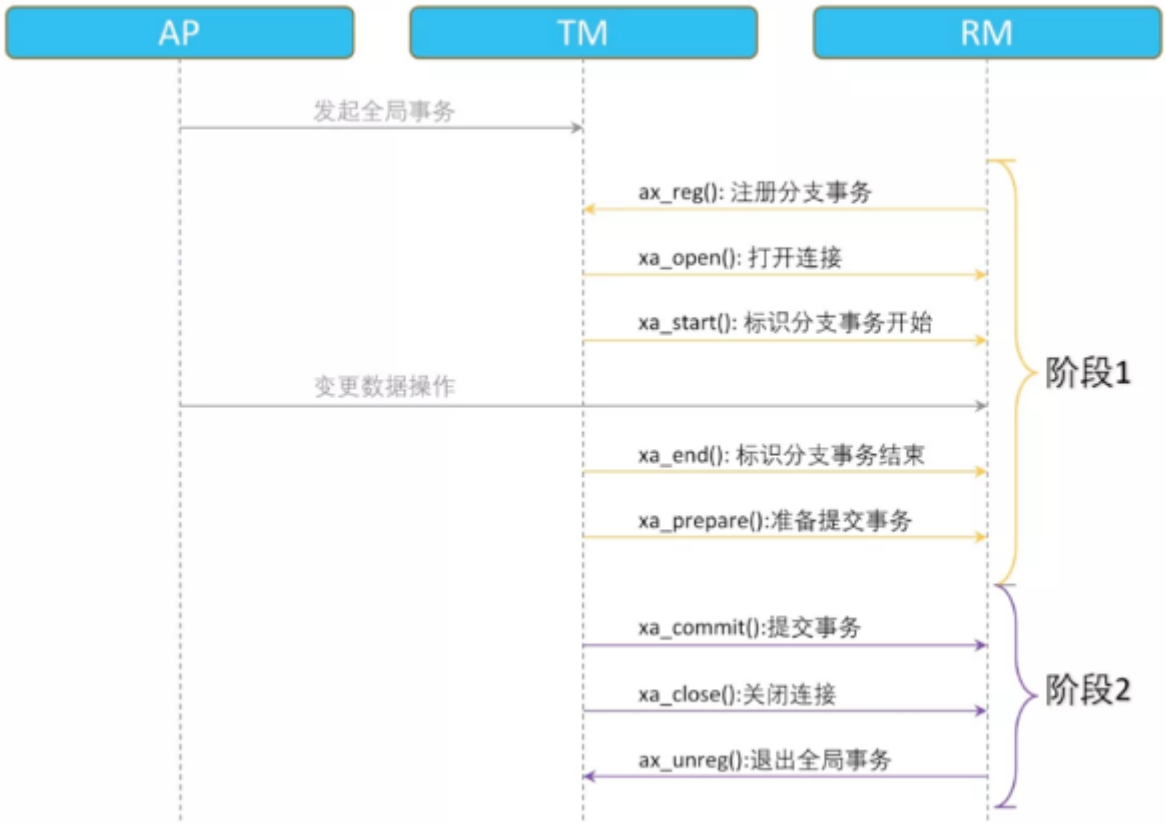
XA事务是基于X/Open DTP事务模型开发的,这个模型主要使用了两段提交(2PC - Two-Phase-Commit)来保证分布式事务的完整性。在这个模型里面,有三个角色:

- AP: Application,应用程序。也就是业务层。哪些操作属于一个事务,就是AP定义的。
- TM: Transaction Manager,事务管理器。接收AP的事务请求,对全局事务进行管理,管理事务分支状态,协调RM的处理,通知RM哪些操作属于哪些全局事务以及事务分支等等。这个也是整个事务调度模型的核心部分。
- RM:Resource Manager,资源管理器。一般是数据库,也可以是其他的资源管理器,如消息队列(如JMS数据源),文件系统等。

XA事务是X/OPEN 提出的分布式事务处理规范。XA则规范了TM与RM之间的通信接口,在TM与多个RM之间形成一个双向通信桥梁,从而在多个数据库资源下保证ACID四个特性。目前知名的数据库,如Oracle, DB2,mysql等,都是实现了XA接口的,都可以作为RM。

> XA是数据库的分布式事务,强一致性,在整个过程中,数据一张锁住状态,即从prepare到commit、rollback的整个过程中,TM一直把持有数据库的锁,如果有其他人要修改数据库的该条数据,就必须等待锁的释放,存在长事务风险。
> 以下的函数使事务管理器可以对资源管理器进行的操作:
>
> 1. xa_open,xa_close:建立和关闭与资源管理器的连接。
> 2. xa_start,xa_end:开始和结束一个本地事务。
> 3. xa_prepare,xa_commit,xa_rollback:预提交、提交和回滚一个本地事务。
> 4. xa_recover:回滚一个已进行预提交的事务。
> 5. ax_开头的函数使资源管理器可以动态地在事务管理器中进行注册,并可以对XID(TRANSACTION IDS)进行操作。
> 6. ax_reg,ax_unreg；允许一个资源管理器在一个TMS(TRANSACTION MANAGER SERVER)中动态注册或撤消注册。

XA事务处理流程如下:

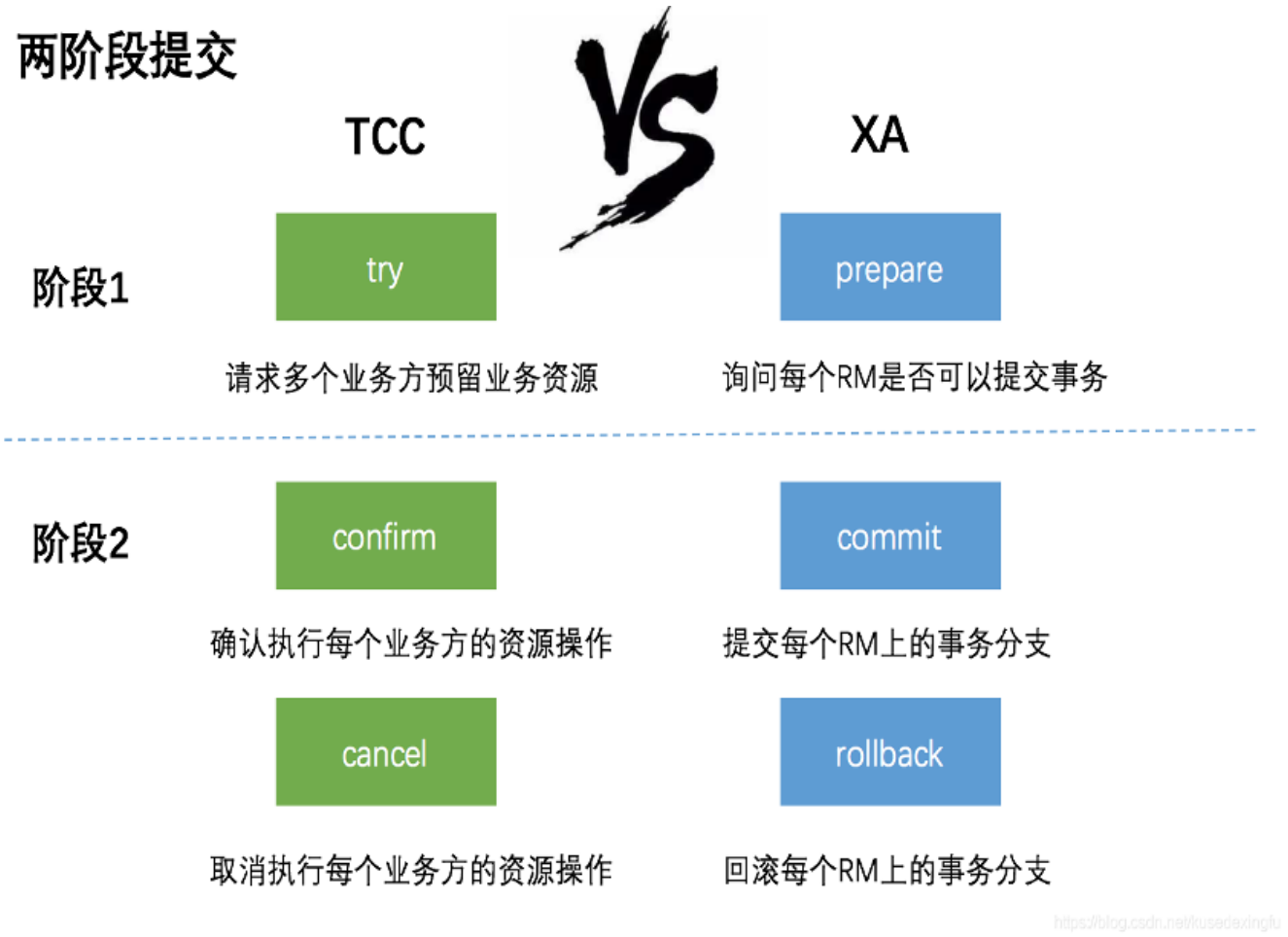


TCC事务

TCC事务主要包括下面几个阶段:

- Try阶段： 完成所有业务检查（一致性）,预留业务资源(准隔离性)
- Confirm阶段： 确认执行业务操作,不做任何业务检查, 只使用Try阶段预留的业务资源。要满足幂等性。
- Cancel阶段： 取消Try阶段预留的业务资源。

TCC与XA两阶段提交有着异曲同工之妙,下图列出了二者之间的对比



- 在阶段1： 在XA中,各个RM准备提交各自的事务分支,事实上就是准备提交资源的更新操作(insert、delete、update等)； 而在TCC中,是主业务活动请求(try)各个从业务服务预留资源。
- 在阶段2： XA根据第一阶段每个RM是否都prepare成功,判断是要提交还是回滚。如果都prepare成功,那么就commit每个事务分支,反之则rollback每个事务分支。在TCC中,如果在第一阶段所有业务资源都预留成功,那么confirm各个从业务服务,否则取消(cancel)所有从业务服务的资源预留请求。

TCC两阶段提交与XA两阶段提交的区别

- XA是资源层面的分布式事务,强一致性,在两阶段提交的整个过程中,一直会持有资源的锁。
- XA事务中的两阶段提交内部过程是对开发者屏蔽的,开发者从代码层面是感知不到这个过程的。而事务管理器在两阶段提交过程中,从prepare到commit/rollback过程中,资源实际上一直都是被加锁的。如果有其他人需要更新这两条记录,那么就必须等待锁释放。
- TCC是业务层面的分布式事务,最终一致性,不会一直持有资源的锁。
- TCC中的两阶段提交并没有对开发者完全屏蔽,也就是说从代码层面,开发者是可以感受到两阶段提交的存在。try、confirm/cancel在执行过程中,一般都会开启各自的本地事务,来保证方法内部业务逻辑的ACID特性。其中：

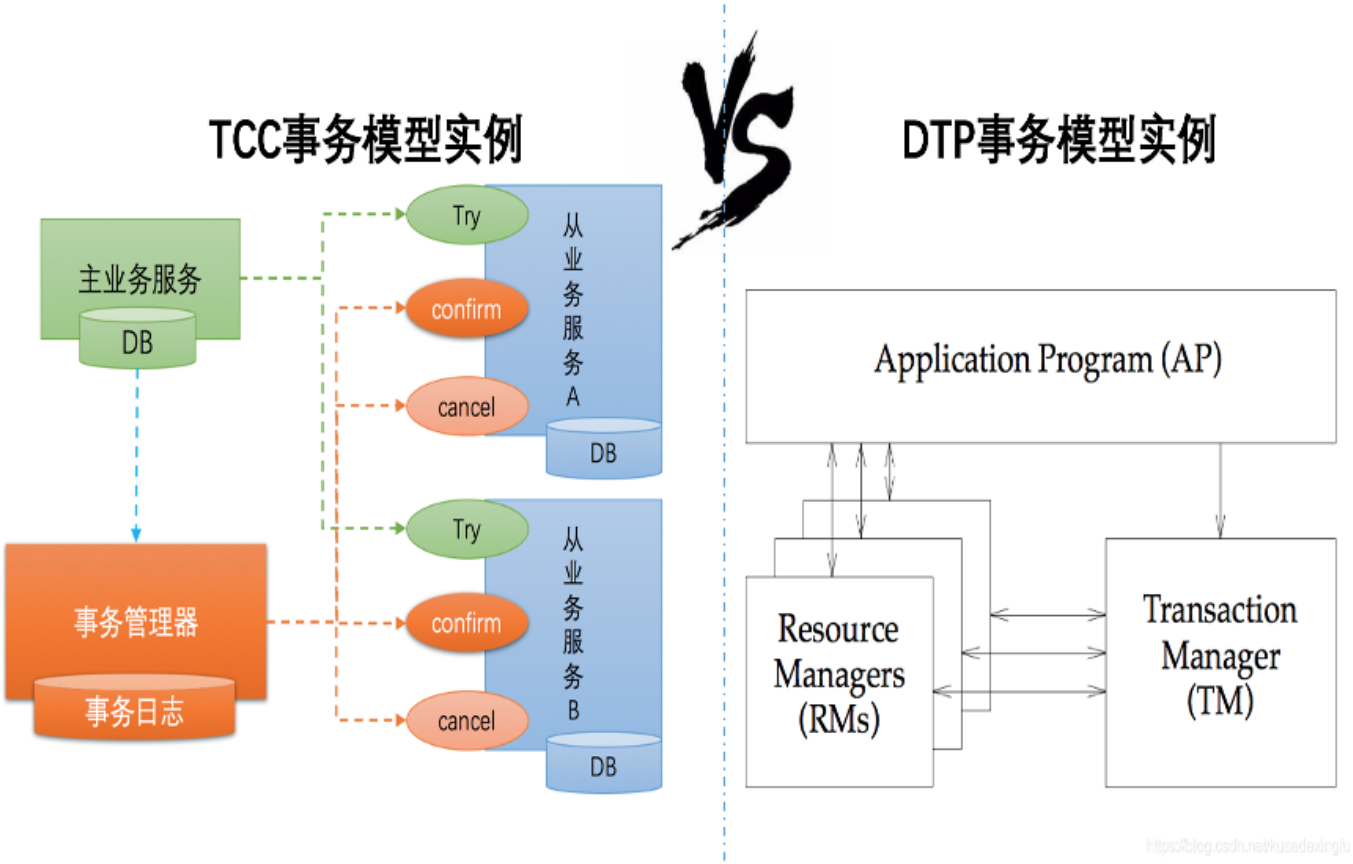
> 1. try过程的本地事务,是保证资源预留的业务逻辑的正确性。
> 2. confirm/cancel执行的本地事务逻辑确认/取消预留资源,以保证最终一致性,也就是所谓的补偿型事务(Compensation-Based Transactions)。由于是多个独立的本地事务,因此不会对资源一直加锁。

另外,这里提到confirm/cancel执行的本地事务是补偿性事务

> 补偿性事务是一个独立的支持ACID特性的本地事务,用于在逻辑上取消服务提供者上一个ACID事务造成的影响,对于一个长事务(long-running transaction),与其实现一个巨大的分布式ACID事务,不如使用基于补偿性的方案,把每一次服务调用当做一个较短的本地ACID事务来处理,执行完就立即提交

TCC事务模型 VS DTP事务模型

比较一下TCC事务模型和DTP事务模型，如下所示：



这两张图看起来差别较大，实际上很多地方是类似的!

1. TCC模型中的主业务服务 相当于 DTP模型中的AP，TCC模型中的从业务服务 相当于 DTP模型中的RM 在DTP模型中，应用AP操作多个资源管理器RM上的资源；而在TCC模型中，是主业务服务操作多个从业务服务上的资源。例如航班预定案例中，美团App就是主业务服务，而川航和东航就是从业务服务，主业务服务需要使用从业务服务上的机票资源。不同的是DTP模型中的资源提供者是类似于Mysql这种关系型数据库，而TCC模型中资源的提供者是其他业务服务。
2. TCC模型中，从业务服务提供的try、confirm、cancel接口相当于DTP模型中RM提供的prepare、commit、rollback接口,XA协议中规定了DTP模型中定RM需要提供prepare、commit、rollback接口给TM调用，以实现两阶段提交。而在TCC模型中，从业务服务相当于RM，提供了类似的try、confirm、cancel接口。
3. 事务管理器 DTP模型和TCC模型中都有一个事务管理器。不同的是： 在DTP模型中，阶段1的(prepare)和阶段2的(commit、rollback)，都是由TM进行调用的。 在TCC模型中，阶段1的try接口是主业务服务调用(绿色箭头)，阶段2的(confirm、cancel接口)是事务管理器TM调用(红色箭头)。这就是 TCC 分布式事务模型的二阶段异步化功能，从业务服务的第一阶段执行成功，主业务服务就可以提交完成，然后再由事务管理器框架异步的执行各从业务服务的第二阶段。这里牺牲了一定的隔离性和一致性的，但是提高了长事务的可用性。

消息队列

本地消息表

将事务拆分为多个小事务，然后存储到本地文本数据库或者消息队列里并发去请求，一旦失败则进行人工重试或者对账修复。如预付费创建机器，生产机器需要进行以下几个步骤：

1. 检查是否有资源
2. 进行扣费；
3. 创建机器以及各种绑定业务

每个业务都部署到不同的机器上，在2步骤扣费后的操作日志存储到数据库或者mq队列里面去执行，如果有失败或者超时的事务则进行重试一般重试需要一个不断变长的时间间隔直到发货成功，如果一直重试不成功则人工根据数据库消息或者mq队列消息进行人工操作或者对账处理，如计费发现创建机器已经多次失败，则取消重试，进行退费。此外，业务接口需要实现幂等性，如创建机器接口就必须要实现幂等性，否则如果第一次请求成功了，但是将本地数据状态修改失败，导致第二次重试则会多生产出一台机器。

MQ事务

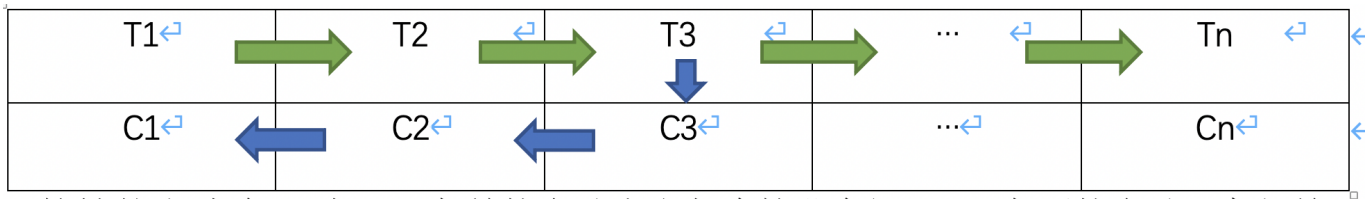
在RocketMQ中实现了分布式事务，实际上其实是对本地消息表的一个封装，将本地消息表移动到了MQ内部，MQ事务的基本流程如下

- 第一阶段Prepared消息，会拿到消息的地址。
- 第二阶段执行本地事务。
- 第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。消息接受者就能使用这个消息。

如果确认消息失败，在RocketMq Broker中提供了定时扫描没有更新状态的消息，如果有消息没有得到确认，会向消息发送者发送消息，来判断是否提交，在rocketmq中是以listener的形式给发送者，用来处理。如果消费超时，则需要一直重试，消息接收端需要保证幂等。如果消息消费失败，这个就需要人工进行处理，因为这个概率较低，如果为了这种小概率时间而设计这个复杂的流程反而得不偿失。

Saga事务

模型：由一串子事务(本地事务)的事务链Tn以及补偿n组成（Tn和Cn）。执行顺序可能是 T1 -> T2 -> T3...-> Tn（最理想情况）也可能T1 -> T2 ->T3...->Tn-1-> Cn-1 ... ->C2 ->C1（最差情况）



C的补偿方式有两种

1. 向前恢复(重试失败的业务)
2. 向后恢复(回滚之前的事务)

要达到事务的完整性，saga对与Tn和Cn必须要有以下几个要求：

1. Tn和Cn是幂等的。
2. Cn必须是能够成功的，如果无法成功则需要人工介入。
3. Tn-Cn和Cn-Tn的执行结果必须是一样的：sub-transaction被撤销。

> - 第一点要求Ti和Ci是幂等的，举个例子，假设在执行Ti的时候超时了，此时我们是不知道执行结果的，如果采用forward recovery策略就会再次发送Ti，那么就有可能出现Ti被执行了两次，所以要求Ti幂等。如果采用backward recovery策略就会发送Ci，而如果Ci也超时了，就会尝试再次发送Ci，那么就有可能出现Ci被执行两次，所以要求Ci幂等。
> - 第二点要求Ci必须能够成功，这个很好理解，因为，如果Ci不能执行成功就意味着整个Saga无法完全撤销，这个是不允许的。但总会出现一些特殊情况比如Ci的代码有bug、服务长时间崩溃等，这个时候就需要人工介入了。
> - 第三点乍看起来比较奇怪，举例说明，还是考虑Ti执行超时的场景，我们采用了backward recovery，发送一个Ci，那么就会有三种情况
>
> 1. Ti的请求丢失了，服务之前没有、之后也不会执行Ti
> 2. Ti在Ci之前执行
> 3. Ci在Ti之前执行

对于第1种情况，容易处理。对于第2、3种情况，则要求Ti和Ci是可交换的（commutative），并且其最终结果都是sub-transaction被撤销。

Saga 模型可以满足事务的三个特性

1. 原子性：Saga 协调器协调事务链中的本地事务要么全部提交，要么全部回滚。
2. 一致性：Saga 事务可以实现最终一致性。
3. 持久性：基于本地事务，所以这个特性可以很好实现。

但是saga模型不支持隔离性，因此需要业务层控制并发，加锁冻结资源等方式去解决问题。 Saga 事务和 TCC 事务一样，都是强依靠业务改造，所以要求业务方在设计上要遵循四个策略：

1. 允许空补偿：网络异常导致事务的参与方只收到了补偿操作指令，因为没有执行过正常操作，因此要进行空补偿。
2. 保持幂等性：事务的正向操作和补偿操作都可能被重复触发，因此要保证操作的幂等性。
3. 防止资源悬挂：原因是网络异常导致事务的正向操作指令晚于补偿操作指令到达，则要丢弃本次正常操作，否则会出现资源悬挂问题。
4. 提供隔离性保证：遵循"宁可长款，不可短款"设计

虽然 Saga 和 TCC 都是补偿事务，但是由于提交阶段不同，所以两者也是有不同的。

- Saga没有Try行为，直接Commit，所以会留下原始事务操作的痕迹，Cancel属于不完美补偿，需要考虑对业务上的影响。TCC Cancel是完美补偿的Rollback，补偿操作会彻底清理之前的原始事务操作，用户是感知不到事务取消之前的状态信息的。
- Saga 的补偿操作通常可以异步执行，TCC的Cancel和Confirm可以跟进需要是否异步化。
- Saga 对业务侵入较小，只需要提供一个逆向操作的Cancel即可；而TCC需要对业务进行全局性的流程改造。
- TCC最少通信次数为2n，而Saga为n（n=子事务的数量）。

关于saga事务详细可查看文章

saga事务1 https://www.jianshu.com/p/e4b662407c66

saga事务2https://servicecomb.apache.org/cn/docs/distributed-transactions-saga-implementation/

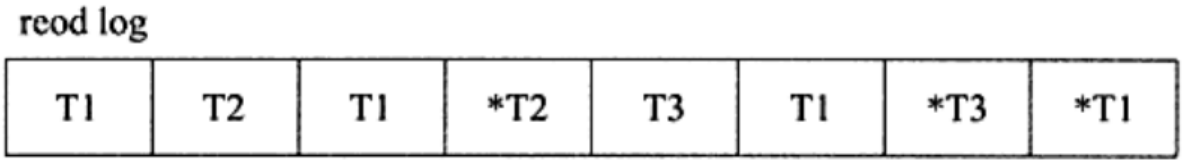
事务的实现

innodb存储引擎中,原子性、一致性、持久性是通过redo log和undo log来实现的。redo log称为重做日志，来保证事务的原子性和持久性。undo log来保证事务的一致性。

redo log、binlog与事务的持久性

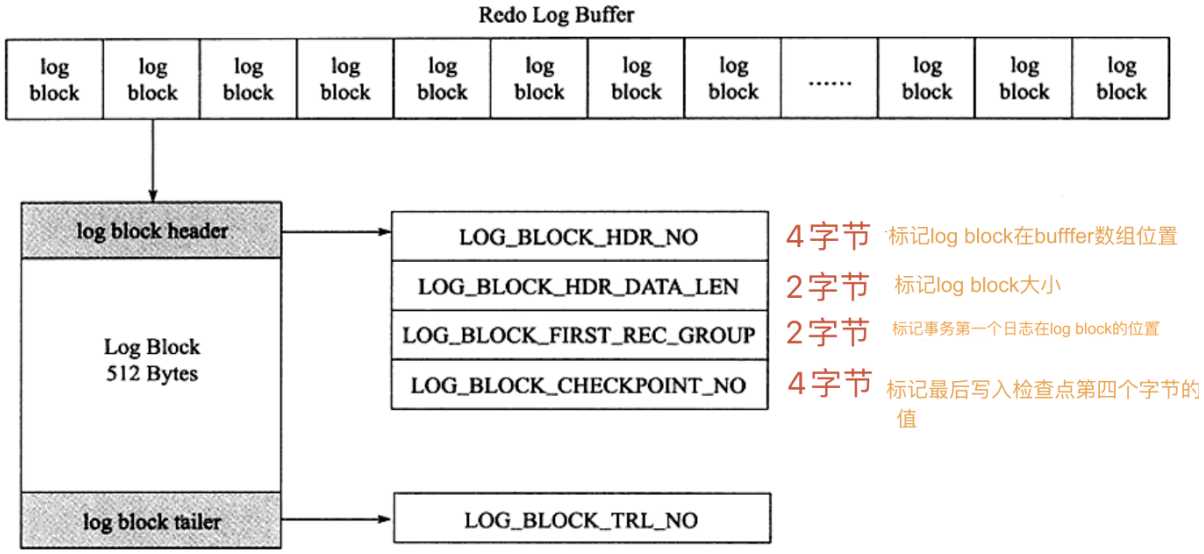
redo log

- 组成：redo log由内存中的重做日志缓冲(易失)和磁盘中的重做日志文件(持久化)两部分组成。
- 性质：redo log是物理格式日志，是存储引擎层产生的，记录的是每个页的修改数据。
- 写入时间点：redo log日志在事务的进行中不断的被写入，如果有多条日志条目，则会并发，这表现为日志并不是随事务提交的顺序一个事务一个事务写入。写入的结果如下图：



- 循环写:redo log有write pos和checkpoint两个指针，当write pos追上checkpoint时，没有空间记录redo log，checkpoint就向前推进将脏页刷入磁盘(刷脏)。
- 结构：redo log 由多个512字节的日志块（log block）组成,而多个redo log则构成一个log group。

1. **log block**
   结构图如下



> - 一个页面产生的log buffer如果超过了512字节则需要存储到多个日志块中
> - log block 由log block header、log body、log block tailer三部分组成。
> - log block header则由 LOG_BLOCK_HDR_NO、LOG_BLOCK_HDR_DATA_LEN、LOG_BLOCK_FIRST_REC_GROUP、LOG_BLOCK_CHECKPOINT_NO组成。
> - 由于重做日志块的大小和磁盘扇区大小一样，所以重做日志写入可以保证原子性，不需要double write技术(详细可以见innodb 刷脏与double write技术)。

2. **log group** 结构图如下

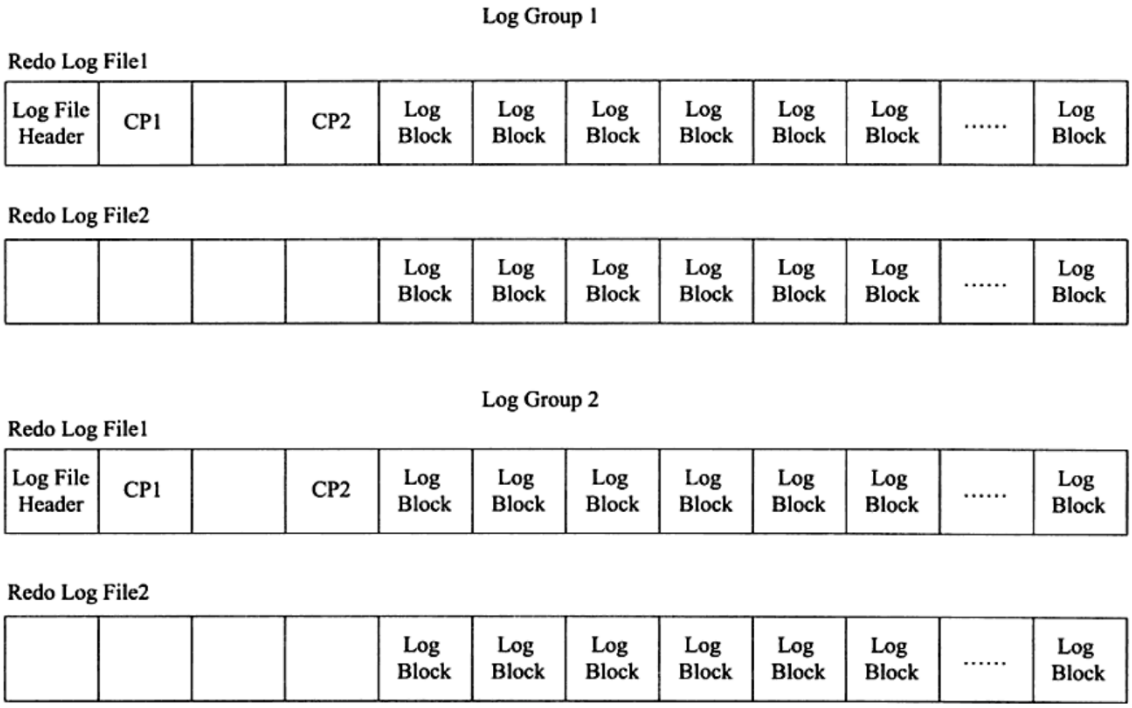
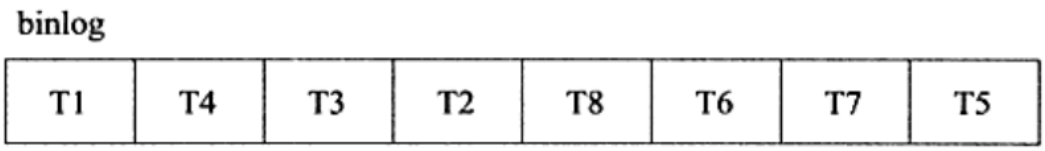


图 7-9 log group 与 redo log file 之间的关系

> - redo log file 前2K空间存储着4个512字节大小的块：log file header、check point1、空、checkpoint2
> - 在log file header 后面check point的值之所以交替写入是为了避免因为介质失败而导致无法找到可用的checkpoint。

binlog

- 用途：用来POINT-IN-TIME(PIT)恢复以及建立主从复制环境。
- 性质：binlog 是逻辑层日志,存储着对应的sql语句,是在数据库上层产生的。
- 写入时间点:该日志是随着事务的提交顺序整体写入的，是一个一个事务写入的，写入的结果如下图:



事务持久化

- 实现原理： 事务的持久性是通Write Ahead logging（WAL）技术实现的，先写日志，再写磁盘。
- 事务提交过程：

> - 开启事务并执行语句但未提交阶段(prepare)主要做几个操作:
>   1. 生成事务ID(xid)信息及redo和undo的内存日志 2. 写事务的undo日志,保存旧数据
>   3. 写事务的redo log但不记录提交（commit）标签
> - 事务commit阶段,主要做几个操作:
>   1. 写binlog(先调用write()将binlog内存日志数据写入文件系统缓存，再调用fsync()将bin

> log文件系统缓存日志数据永久写入磁盘)
> 2. redo log中记录此事务的提交日志（增加commit 标签）

- 奔溃恢复

> 数据库启动时会检查事务的redo log和binlog,判断是否存在下面两种异常情况
>
> 1. 判断此事务的binlog在记录之前和过程中是否发生crash。如果发生了crash,则在数据库在恢复后认为此事务并没有成功提交，则会回滚此事务的操作。与此同时，因为在binlog中也没有此事务的记录，所以从库也不会有此事务的数据修改。
> 2. 判断此事务的binlog是否在记录之后发生crash。如果是则无论redo 日志中是否拥有commit标志，都会认为该事务提交成功，会在redolog中重新写入commit标志，并完成此事务的重做（主库中有此事务的数据修改）。与此同时，因为在binlog中已经有了此事务的记录，所有从库也会有此事务的数据修改。

undo log

- 作用：实现事务的原子性(回滚)和隔离性(MVCC)。
- 实现：undo log是通过对sql语句做反向处理来实现的,并不是通过物理日志实现的,比如事务中insert一条数据到db中,则需要现在undo日志里面先增加一条delete语句,如果事务回滚或者宕机可以通过undo日志去恢复数据。
- 格式：undo log分为insert undo log和update undo log

insert undo log

insert undo log值得是在insert操作中产生的undo log。insert操作只在当前事务本身可见，对其他事务屏蔽因此，insert undo log可以在事务提交后删除，不需要purge。

- insert undo log 结构

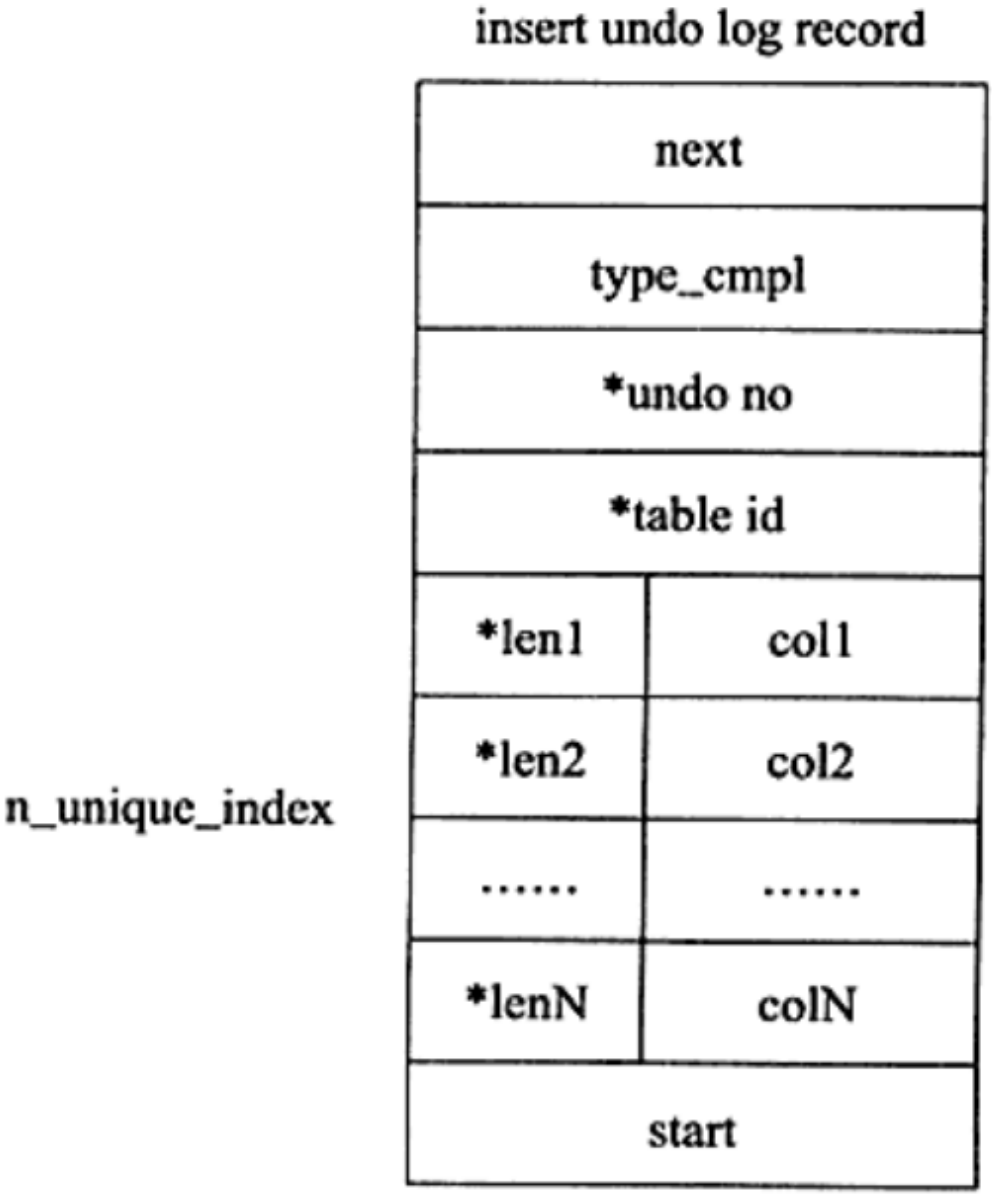


图 7-14　insert undo log 的格式

> - next 两个字节，记录的是下一个undo log的位置
> - type_cmp1 一个字节，记录的是undo log的类型，对于insert undo log该值是11
> - undo_no记录事务的ID
> - table_id 记录表ID
> - n_unique_index表示所有主键的列和值

update undo log

update undo log是对delete和update操作产生的undo log,该undo log可能需要提供MVCC机制,因此只能放入undo log链表,等待purge县城进行删除，而不能提交事务后删除。

- update undo log 结构

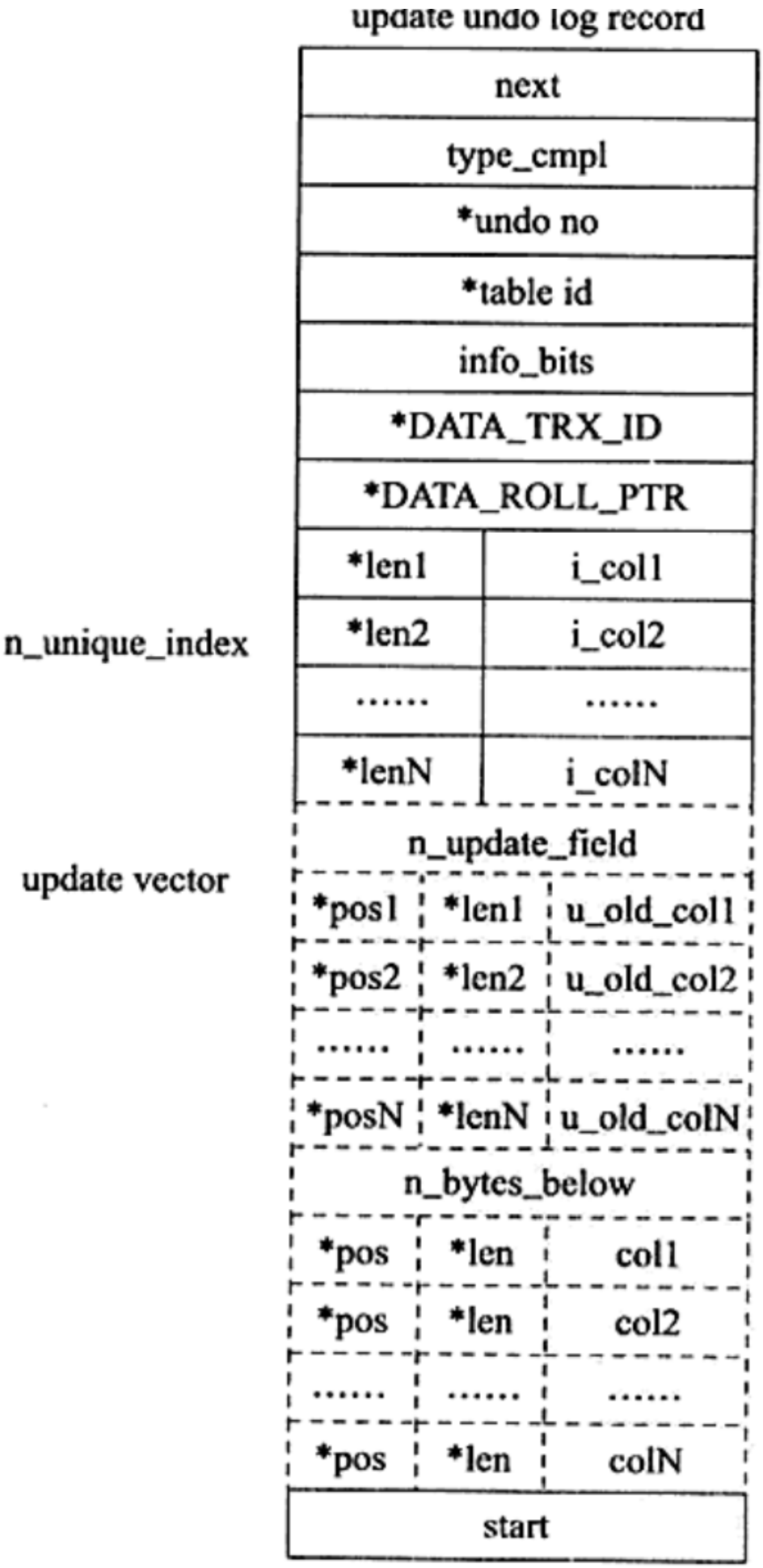


图 7-15　update undo log 格式

> - next 两个字节，记录的是下一个undo log的位置
> - type_cmp1 一个字节，记录的是undo log的类型，对于insert undo log该值是11
> - undo_no记录事务的ID
> - table_id 记录表ID
> - n_unique_index表示所有主键的列和值
> - update_vector 表示update操作导致发生改变的列

MVCC

MVCC 定义

MVCC指多版本并发控制，让普通的select语句直接读取指定版本的值，避免加锁，来提高并发请求时的性能，配合行锁机制，在并发请求下，提高了MYSQL的性能

作用

MVCC使得db读写并行，不阻塞,例如：

1. 在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能
2. 同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题

MVCC实现原理

MVCC主要是依靠表的3个隐藏字段，undo log以及Read View来实现的。

- 表的3个隐藏字段

字段名	字节	说明	作用
DB_TRX_ID	6	最近修改(修改/插入)事务ID	记录创建这条记录/最后一次修改该记录的事务ID
DB_ROLL_PTR	7	回滚指针	指向这条记录的上一个版本（存储于rollback segment里）
DB_ROW_ID	6	隐含的自增ID（隐藏主键）	无主键时作为索引列 |

- Undo log见上面
- ReadView

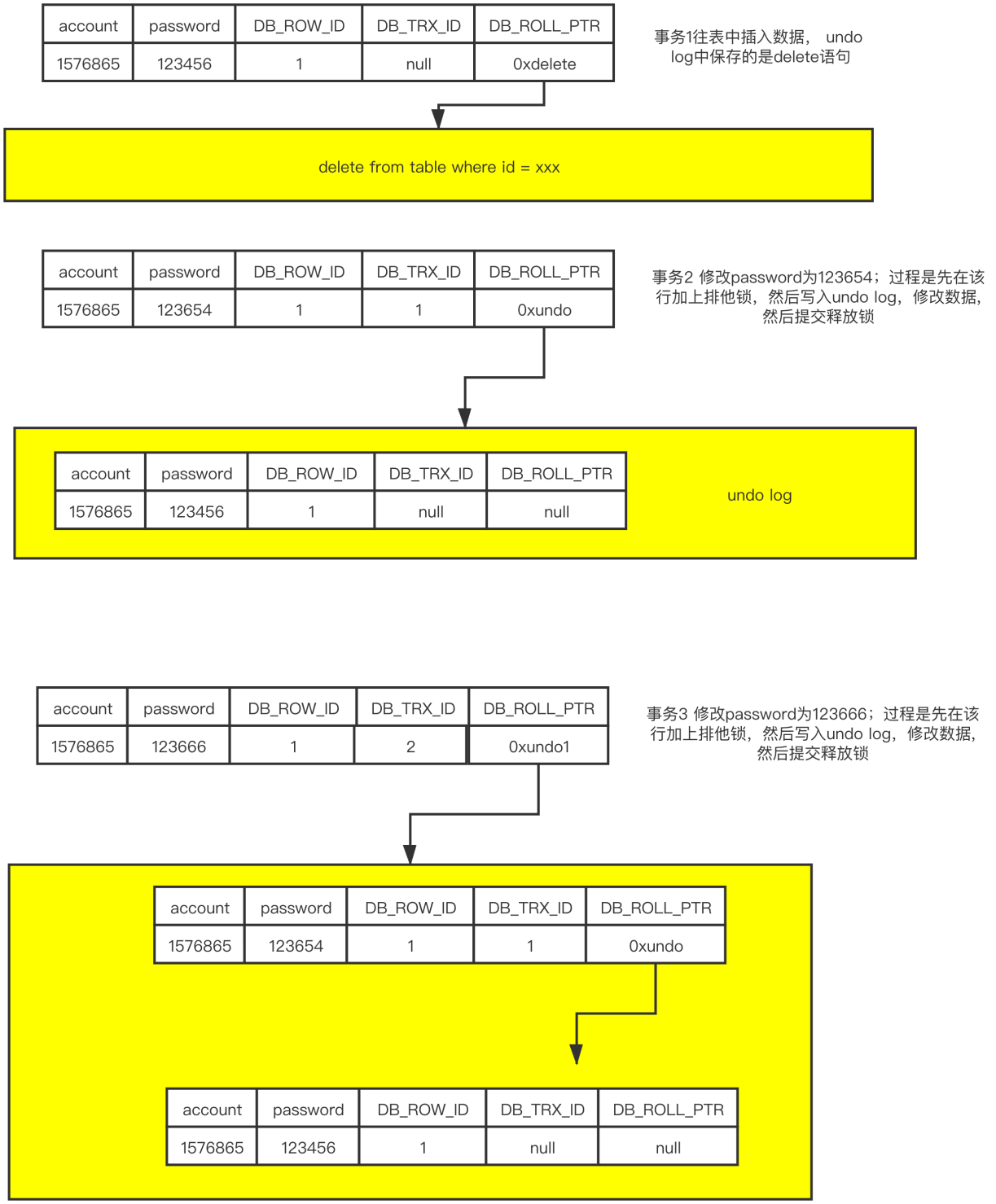
> Read View中 有四个比较重要的内容

名称	说明
m_ids	表示在生成ReadView时当前系统中活跃的读写事务的事务id列表。

名称	说明
min_trx_id	表示在生成ReadView时当前系统中活跃的读写事务中最小的事务id，也就是m_ids中的最小值。
max_trx_id	表示生成ReadView时系统中应该分配给下一个事务的id值。 |
creator_trx_id	表示生成该ReadView的事务的事务id |

> Read View 使用场景
>
> - 如果被访问版本的trx_id属性值与ReadView中的creator_trx_id值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。
> - 如果被访问版本的trx_id属性值小于ReadView中的min_trx_id值，表明生成该版本的事务在当前事务生成ReadView前已经提交，所以该版本可以被当前事务访问。
> - 如果被访问版本的trx_id属性值大于或等于ReadView中的max_trx_id值，表明生成该版本的事务在当前事务生成ReadView后才开启，所以该版本不可以被当前事务访问。
> - 如果被访问版本的trx_id属性值在ReadView的min_trx_id和max_trx_id之间，那就需要判断一下trx_id属性值是不是在m_ids列表中，如果在，说明创建ReadView时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。

- 实现过程



> 执行过程如下
>
> - 事务1插入一条数据到表中，insert undo log中保存着一条delete语句，在事务提交的时候insert undo log就被删除了
> - 事务2修改password为123654, 则事务2需要做以下几个步骤：
>   1. 对该行数据加排他锁
>   2. 拷贝该行数据到update undo log
>   3. 修改改行数据并修改事务id
>   4. 提交事务
> - 事务3 修改passwork为123666

> 1. 对该行数据加排他锁
> 2. 发现该行记录已经有undo log了，那么最新的旧数据作为链表的表头，插在该行记录的undo log最前面
> 3. 修改改行数据并修改事务id
> 4. 提交事务

- 如何提高效率

> 由上图可见，如果一个undo日志很长，而需要查询的数据版本比较早的话需要回溯的链路就很长，从而导致性能很差。而缩短链路的方法就是删除早期的undo log，而删除早期的undo log的条件是undo log中事务id是当前最小的，因此只要保证当前事务中的id都是比较大的就行了，也就是说不要让事务存活时间过长（避免使用长事务）。