分布式事务的笔记

只有mysql5.0以上才能支持xa协议，并不只有innodb才支持xa协议，在使用xa协议的时候必须使用的事务隔离级别是可序列化，对性能影响很大

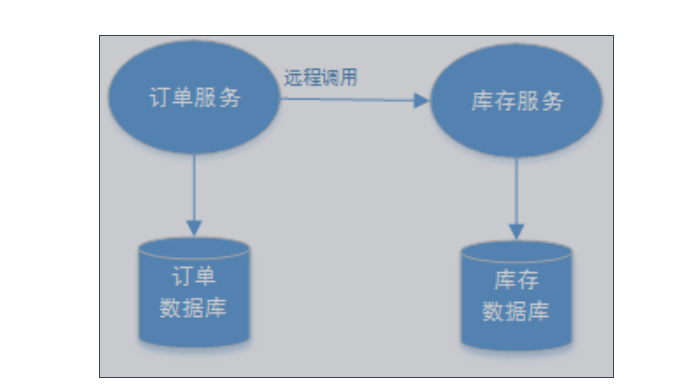
分布式事务产生的例子

张三向李四转账，李四那边已经收到了转账的消息并且成功转账，然而因为网络抖动张三没有收到李四的转账成功的信息，所以张三回滚了本地事务，那么就造成张三没有减钱，李四加了钱，造成分布式不一致的问题。

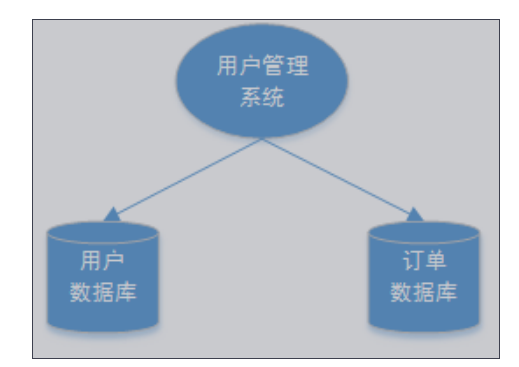
分布式事务产生的场景

只要是两个以上的微服务调用，只要是这次调用采用两个数据库以上都会产生分布式事务，因为两个微服务之间的调用会因为网络的抖动产生分布式事务，两个数据库的话因为单体事务是由一个数据库进行的，两个数据库意味着两个事务，当然会产生分布式事务

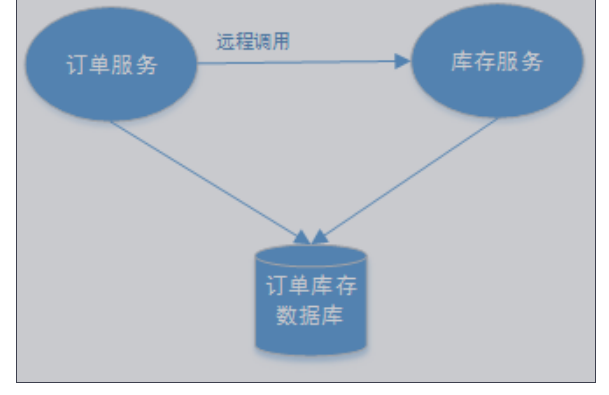
多库多服务



单服务多库



多服务单库



Cap理论

一致性

一致性是指写操作后的读操作可以读取到最新的数据状态，当数据分布在多个节点上，从任意结点读取到的数据都 是最新的状态。

可用性

可用性是指任何事务操作都可以得到响应结果，且不会出现响应超时或响应错误。

分区容错性

通常分布式系统的各各结点部署在不同的子网，这就是网络分区，不可避免的会出现由于网络问题而导致结点之间 通信失败，此时仍可对外提供服务，这叫分区容忍性。

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。BASE理论是对CAP中AP的一个扩展，通过牺牲强一致性来获得可用性，当出现故障允许部分不可用但要保证 核心功能可用，允许数据在一段时间内是不一致的，但最终达到一致状态。满足BASE理论的事务，我们称之为“柔 性事务”。

基本可用:

分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用。如，电商网站交易付款出 现问题了，商品依然可以正常浏览。

软状态:

由于不要求强一致性，所以BASE允许系统中存在中间状态（也叫软状态），这个状态不影响系统可用 性，如订单的"支付中"、“数据同步中”等状态，待数据最终一致后状态改为“成功”状态。

最终一致:

最终一致是指经过一段时间后，所有节点数据都将会达到一致。如订单的"支付中"状态，最终会变 为“支付成功”或者"支付失败"，使订单状态与实际交易结果达成一致，但需要一定时间的延迟、等待。

什么是2pc

2pc是指两阶段提交协议，第一阶段为准备阶段，所有的分布式事务的节点会尝试执行业务代码，但是不提交事务并且会保留锁，第二阶段提交阶段，在这个阶段如果所有节点都成预先执行常规则提交事务完成调用，如果有一个节点时报则进行回滚

基于xa的2pc

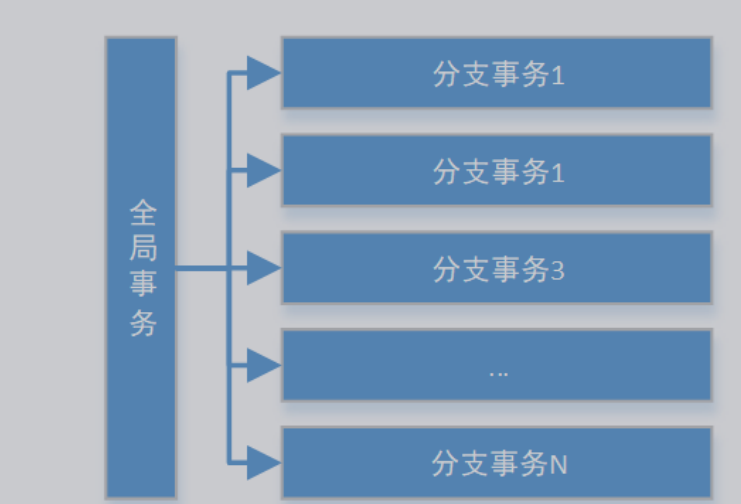
Oracle和mysql的innodb支持分布式事务的xa协议，他的执行流程和2pc一致，他的缺点就是要所有的数据库都支持xa协议，第二就是在事务的进行的两个阶段都会锁定数据，性能较差

Seata 的AT（2pc）

Seata是阿里的开源项目，它的AT（2pc）很好的解决了ax方案里面所资源锁定的问题，并且它工作与应用层，采用aop的方式接入分布式事务，对于我们的业务代码0侵入，实现的成本低

Seata AT方案的解决方式

首先seata将分布式事务拆解为全局事务和分支事务，所以分支事务成功则成功，任何分支事务失败则失败



Seata将分布式事务的控制分为三个角色

TC （trantional coordinate：事务协调器），事务协调器是独立的一个项目，可以通过官方提供的jar包直接启动完成，它负责协调整一个分布式事务，和每个TM和RM通信，接受他们发来的关于事务的开启、分支事务的情况、等信息，然后根据各个节点的情况，进行协调和操作整个分布式事务的进行或者回滚

TM（trantional Manager 事务管理者），事务管理者是位于开启分布式事务的微服务里，也是位于有@grobalTrantional注解上，TM负责整个分布式事务的开启全局事务，回滚全局事务，提交全局事务。在微服务一开始的时候，它就会想TC注册全局事务得到全局事务id，在各个分支事务执行完毕后，因为他是整个调用链的第一个，所以它很容易知道整个调用链是否都成功了，或者某些失败了，因此它在最后如果都成功了则告诉哦tc进行提交全局事务，如果某个节点失败了则回滚全局事务

RM（Resources Manager）数据库资源管理器，传统的2pc里面这个角色就是数据库本身，在seata里面则是在应用层，它通过代理的方式获取数据库连接可以操控数据库，是数据的资源管理者，执行具体的分支事务的提交和回滚

Seata执行分布式事务的具体流程，以转账为例子 见另一个文档《分布式事务专题1.1》的3.3.7

以张三转账李四为例子

首先所有的tm和rm在启动的时候想tc注册

张三Tm开启全局事务获取全局事务id

张三微服务注册分支事务，获取分支事务id

张三微服务执行本地事务

将需要的数据获取全局锁，并且提交分支事务释放本地所

张三以feign的方式携带全局事务id调用李四

李四注册分支事务，

李四执行事务

将需要的数据获取全局锁，并且提交分支事务释放本地所

李四上报处理结果

张三的tm发起全局提交，释放xid对应的所有的全局锁

Over

Seata的注意点

Seata的第一阶段每个分布式事务的提交实际上是真的提交，并且释放锁资源，它的回滚是采用undo\_long里面记录的镜像文件来进行数据的反向执行进行回滚

Seata反向执行的原理

Seata相较于xa的优点

占用锁的事件短，每个分支事务都是真实的提交不必占用锁很长时间，并且基于应用层，业务0侵入，开发成本低

Seata的全局锁的原因

分布式事务中在两个全局事务中可能存在相互的干扰，形成脏写，如分布式事务1的一阶段中改了张三的账户金额增加100，从1000到1100，在不加全局锁的情况之下，由于分支事务提交之后会释放本地锁，所以分布式事务2就可以修改张三的金额，如减掉200到900，此时，如果分布式事务其他分支执行失败回滚，那么金额会改回1000，那么分布式事务的写就被覆盖了，形成更新丢失，所以为了杜绝这种情况使用了分布式锁。

分布式锁与本地所的获取与执行流程（以张三向李四转账为例子）

张三 李四 tc

获取本地锁

执行本地事务-100

然后获取全局锁

事务执行成功后

释放本地锁，不放

全局锁

获取本地锁

执行本地事务+100

然后获取全局锁

事务执行成功后

释放本地锁，不放

全局锁

Tm得知分支事务

执行完毕，

提交全局事务 全局事务提交

释放这个全局

事务id对应的

所有全局锁

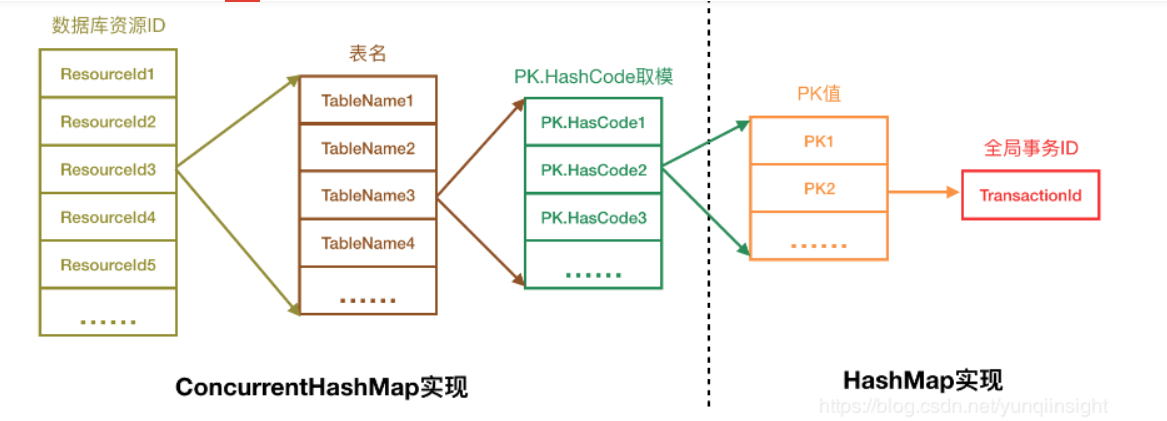
1张三在提交本地事务前必须获取全局锁，并且提交本地事务后释放数据库锁不释放全局锁

2李四在提交本地事务前必须获取全局锁，并且提交本地事务后释放数据库锁不释放全局锁

3全局事务提交之后才释放所有的全局锁

Tc上全局锁的存储方式

currentHashmap<String，CurrentHashmap<Integer,map<string,long>>>



三层hashmap，

第一层的key是库，value表

第二层key是表，value是具体数据行

第三层key是主键，value是全局事务id

加锁

使用需要加锁的数据库主键然后调用get方法如果返回null，则将数据写入hashmap，然后返回true，如果get方法返回不是null，并且他的值不等于当前全局事务id，则表示已经被其他线程加锁，采用cas等待

解锁

将数据从hashmap删除就是释放锁

Seata的全局事务隔离级别是读未提交，所以使用全局锁来防止脏写，这个锁的粒度比较细，但是持有的事件比较长，如果并发的量比较大会影响性能，对于读一致性，一般是允许的，如转账的时候，转走500，然后立即看到账户剩余500这是正常的，如果不允许这种情况出现，seata采用数据库悲观锁来避免脏读。

Seata的全局锁

Seata为了防止全局事务的写隔离，在tc上设置全局锁，每个分支事务再进行commit的时候必须获取全局锁才能commit成功，然后释放本地锁，对于不能获取到全局锁的分布式事务会进行重试，知道超时回滚，全局锁避免了全局事务的脏写，但是也对性能有影响。

Seata　ＡＴ模式实现２ｐｃ的缺点

需要多次的和tc进行连接，进行数据交互，消耗大量网络资源

在每次commit之前都必须获取tc上的全局锁，会有锁竞争问题，

在大概率情况下，分布式事务是可以成功的，但是seata必须为每次分布式事务在undo\_log上面进行数据的插入和删除，不可避免会消耗大量网络资源和数据库资源，拖慢整个调用过程。

Seata实现2PC要点：

1、全局事务开始使用 @GlobalTransactional标识 。

2、每个本地事务方案仍然使用@Transactional标识。

3、每个数据都需要创建undo\_log表，此表是seata保证本地事务一致

Seata的代码原理

TCC原理

也是两个阶段，一阶段try操作进行预执行，以及资源预留，二阶段执行cancel或者conform，一阶段如果成功，那么conform也必须成功，如果不成功则重试以人工接入，一阶段失败，则cancel也必须完成，不成功则重试或者是人工干预

Tcc的两个角色

TM：事务管理器，负责事务调度一阶段的try和二阶段的cancel

RM：每个分支事务的资协调者



实现要点

1每个分支事务的数据库需要建立try\_log，conform\_log，cancel\_log三张表用于记录相关操作

2每个分支事务都要编写try conform cancel三个操作

Tcc常见的一些问题

空回滚

Try没有执行就执行cancel

例子，张三调用李四，李四因为网络延迟没有try，但是张三执行try后出错了，然后事务管理者让所有的分支直接执行cancel，那么李四就是没有执行try就执行回滚，

解决方法，

每执行一条try，就一全局事务id记录一次try记录，执行cancel的时候看一下有没有执行过对应的try，没有就不回滚

幂等性

因为try和cancel，conform均可能会反复执行，

用全局事务id来进行幂等性的校验

悬挂

Cancel执行完了，还执行try

例子：张三转李四，张三这边已经回滚了，tm发起cancel，李四执行了cancel，然后才先收到了try请求，然后执行try，那么这个try预留的资源将没有对应的cancel或者conform造成悬挂

解决方法

有执行过cancel就不执行try，每次执行cancel的时候在cancel表里插入以全局事务id为主键的cancel记录，try的时候查一下这个全局事务id对应的cancel有没有执行，如果执行了就不执行try

Tcc处理张三转账李四的实际操作



Try操作如何预留资源

从上面可以看出try操作是真的操作，如真的扣钱，其实就是冻结资金，保证conform的时候有钱扣，如果失败了就加回去，而不是在数据库层面加锁。

Tcc的优点，

通过自定义try cancel 和comfirm来进行分布式事务的进行和回滚，多一可以精细控制分布式锁的每个步骤，所以可以较为灵活控制共享资源的使用，控制锁的粒度，提高并发量，也是因为tcc的灵活，我们需要编写try cancel 和comfirm自己来实现分布式事务的提交和回滚逻辑，所以业务的侵入性强，并且实现难度较大，成本高

可靠性消息一致性

Ebay方案

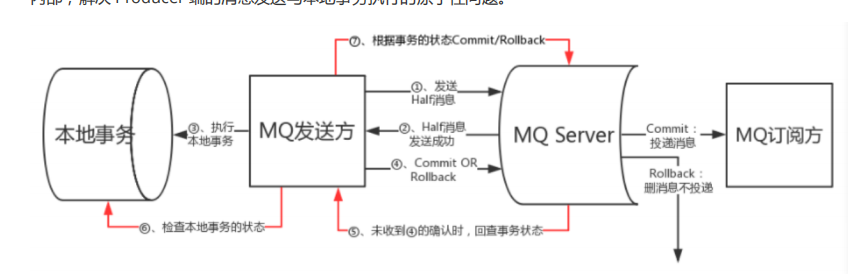
第一步将业务处理和需要发送到mq 的信息在同一个本地事务里去处理

采用定时任务将存在数据库里的需要发mq的消息取出来投递到mq，采用confirm机制保证将消息投递到mq

Mq需要进行队列和exchange的持久化，防止消息丢失

消费者采用手动ack的方式将消息进行消费，需要保证消息被消费，必要的时候进人工介入

基于rocketmq的事务消息解决



1生产者（事务发起者）向mq发送事务消息

2mq收到后告诉生产者发送成功并且回调RocketMQLocalTransactionListener

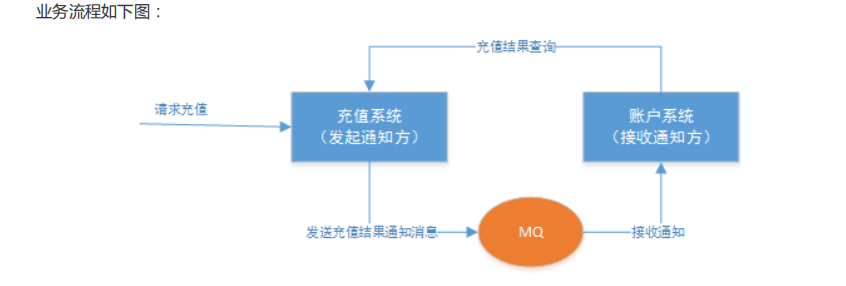
里面的executeLocalTransaction方法进行本地事务的执行，

3commit或者rollback消息，如果生产者一直不回复，mq则会自己调用回查接口查询生产者事务进行情况，即调用RocketMQLocalTransactionListener的checkLocalTransaction进行回查

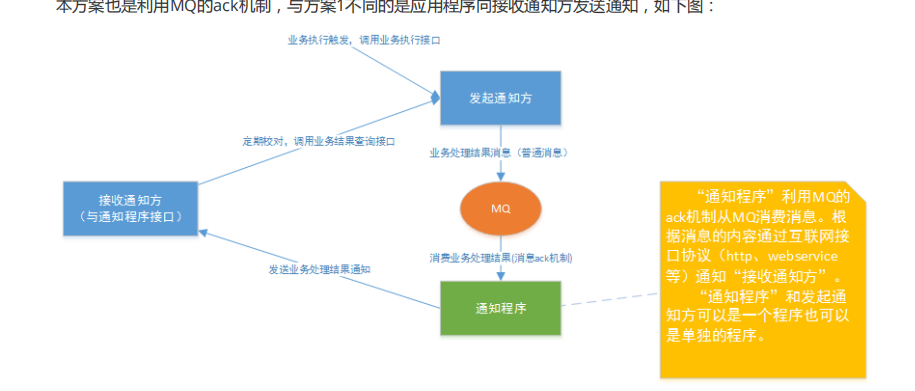
4如果commit则mq会将消息进行投递给消费者，如果rollback则将消息删除

5消费者保证消息消费成功，必要时人工介入

最大努力通知型



改进版



流程

1业务执行方发起请求向发起通知方（充值系统）发起http请求，

2充值系统将充值结果放到mq

3mq会通知业务执行方充值结果，或者设立一个专门的通知程序通知所有需要知道充值结果的程序去知道这个结果，这便是最大努力通知

4充值系统本身提供一个充值结果查询接口，充值方可以通过这个接口知道充值结果，从而修改自己的状态达到最终一致性

总结

2PC 最大的诟病是一个阻塞协议。RM在执行分支事务后需要等待TM的决定，此时服务会阻塞并锁定资源。由于其 阻塞机制和最差时间复杂度高， 因此，这种设计不能适应随着事务涉及的服务数量增加而扩展的需要，很难用于并 发较高以及子事务生命周期较长 (long-running transactions) 的分布式服务中。

如果拿TCC事务的处理流程与2PC两阶段提交做比较，2PC通常都是在跨库的DB层面，而TCC则在应用层面的处 理，需要通过业务逻辑来实现。这种分布式事务的实现方式的优势在于，可以让应用自己定义数据操作的粒度，使 得降低锁冲突、提高吞吐量成为可能。而不足之处则在于对应用的侵入性非常强，业务逻辑的每个分支都需要实现 try、confifirm、cancel三个操作。此外，其实现难度也比较大，需要按照网络状态、系统故障等不同的失败原因实 现不同的回滚策略。典型的使用场景：满，登录送优惠券等。

可靠消息最终一致性事务适合执行周期长且实时性要求不高的场景。引入消息机制后，同步的事务操作变为基于消息执行的异步操作, 避免了分布式事务中的同步阻塞操作的影响，并实现了两个服务的解耦。典型的使用场景：注册送积分，登录送优惠券等。

最大努力通知是分布式事务中要求最低的一种,适用于一些最终一致性时间敏感度低的业务；允许发起通知方处理业 务失败，在接收通知方收到通知后积极进行失败处理，无论发起通知方如何处理结果都会不影响到接收通知方的后 续处理；发起通知方需提供查询执行情况接口，用于接收通知方校对结果。典型的使用场景：银行通知、支付结果 通知等。



2pc和tcc都是一致性较好的，可回滚，实时性强，用于需要强一致性的业务中，如跨行转账，这种是必须要可回滚，但是缺点就是吞吐量不高，如果某个步骤需要花费大量时间的会导致性能问题，如消息发送，用2pc或者tcc则会阻塞

分布式消息一致性则是不可会滚，但是吞吐量高，因为用了消息中间件可进行硝峰，适用于需要高吞吐，实时性不强的业务中，如外卖点餐派单，发送消息等

最大努力通实时性低，一致性不高，一般用于事情做完后的通知，如充值系统通知账户系统充值结果，如注册送积分。