主讲老师： Fox

学习本课程基础：
了解本地事务（ACID特性）和分布式事务的区别
了解CAP，BASE理论
了解DTP模型

1. 常见的分布式事务解决方案

	2PC	TCC	可靠消息	最大努力通知
一致性	强一致性	最终一致	最终一致	最终一致
吞吐量	低	中	高	高
实现复杂度	易	难	中	易

2. 2PC（两阶段提交）方案

两阶段提交协议（Two Phase Commit）不是在XA规范中提出，但是XA规范对其进行了优化。而从字面意思来理解，Two Phase Commit，就是将提交(commit)过程划分为2个阶段(Phase)：

阶段1：

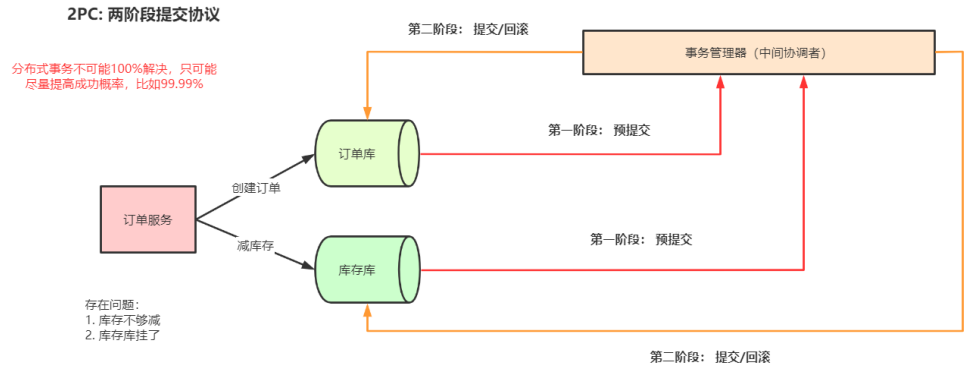
TM通知各个RM准备提交它们的事务分支。如果RM判断自己进行的工作可以被提交，那就对工作内容进行持久化，再给TM肯定答复；要是发生了其他情况，那给TM的都是否定答复。在发送了否定答复并回滚了已经的工作后，RM就可以丢弃这个事务分支信息。

以mysql数据库为例，在第一阶段，事务管理器向所有涉及到的数据库服务器发出prepare"准备提交"请求，数据库收到请求后执行数据修改和日志记录等处理，处理完成后只是把事务的状态改成"可以提交",然后把结果返回给事务管理器。

阶段2

TM根据阶段1各个RM prepare的结果，决定是提交还是回滚事务。如果所有的RM都prepare成功，那么TM通知所有的RM进行提交；如果有RM prepare失败的话，则TM通知所有RM回滚自己的事务分支。

以mysql数据库为例，如果第一阶段中所有数据库都prepare成功，那么事务管理器向数据库服务器发出"确认提交"请求，数据库服务器把事务的"可以提交"状态改为"提交完成"状态，然后返回应答。如果在第一阶段内有任何一个数据库的操作发生了错误，或者事务管理器收不到某个数据库的回应，则认为事务失败，回撤所有数据库的事务。数据库服务器收不到第二阶段的确认提交请求，也会把"可以提交"的事务回撤。



两阶段提交协议(2PC)存在的问题

二阶段提交看起来确实能够提供原子性的操作，但是不幸的是，二阶段提交还是有几个缺点的：

1、同步阻塞问题。

两阶段提交方案下全局事务的ACID特性，是依赖于RM的。一个全局事务内部包含了多个独立的事务分支，这一组事务分支要不都成功，要不都失败。各个事务分支的ACID特性共同构成了全局事务的ACID特性。也就是将单个事务分支的支持的ACID特性提升一个层次到分布式事务的范畴。 即使在本地事务中，如果对操作读很敏感，我们也需要将事务隔离级别设置为SERIALIZABLE。而对于分布式事务来说，更是如此，可重复读隔离级别不足以保证分布式事务一致

性。如果我们使用mysql来支持XA分布式事务的话，那么最好将事务隔离级别设置为SERIALIZABLE，然而SERIALIZABLE(串行化)是四个事务隔离级别中最高的一个级别，也是执行效率最低的一个级别。

2、单点故障。

由于协调者的重要性，一旦协调者TM发生故障，参与者RM会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

3、数据不一致。

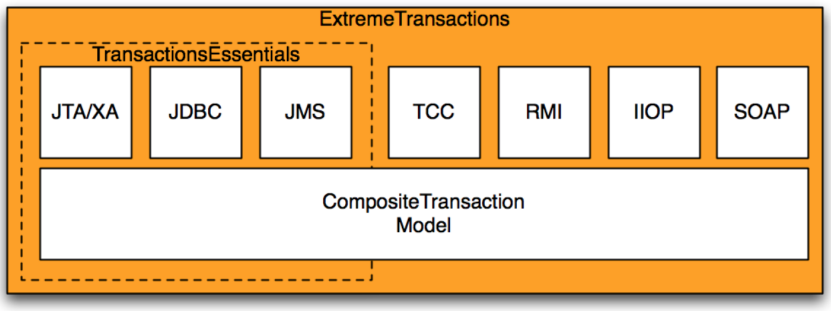
在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这会导致只有一部分参与者接受到了commit请求，而在这部分参与者接到commit请求之后就会执行commit操作，但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象。

2.1 JTA/XA规范实现

开源框架Atomikos：

TransactionEssentials：开源的免费产品

ExtremeTransactions：上商业版，需要收费。



TransactionEssentials：

1、实现了JTA/XA规范中的事务管理器(Transaction Manager)应该实现的相关接口，如：

UserTransaction实现是com.atomikos.icatch.jta.UserTransactionImp，用户只需要直接操作这个类

TransactionManager实现是com.atomikos.icatch.jta.UserTransactionManager

Transaction实现是com.atomikos.icatch.jta.TransactionImp

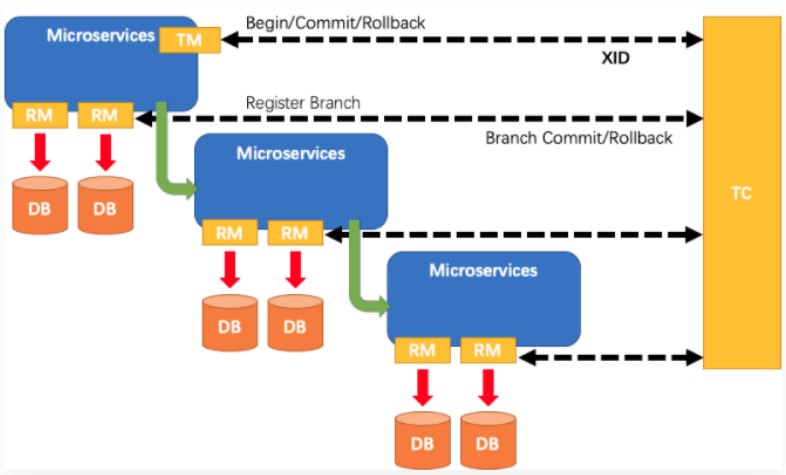
2、针对实现了JDBC规范中规定的实现了XADataSource接口的数据库连接池，以及实现了JMS规范的MQ客户端提供一层封装。

典型的XADataSource实现包括：

- mysql官方提供的com.mysql.jdbc.jdbc2.optional.MysqlXADataSource
- 阿里巴巴开源的druid连接池，对应的实现类为com.alibaba.druid.pool.xa.DruidXADataSource
- tomcat-jdbc连接池提供的org.apache.tomcat.jdbc.pool.XADataSource

2.2 Seata AT模式实现

在 Seata 中，一个分布式事务的生命周期如下：



1.TM 请求 TC 开启一个全局事务。TC 会生成一个 XID 作为该全局事务的编号。XID，会在微服务的调用链路中传播，保证将多个微服务的子事务关联在一起。

2.RM 请求 TC 将本地事务注册为全局事务的分支事务，通过全局事务的 XID 进行关联。

3.TM 请求 TC 告诉 XID 对应的全局事务是进行提交还是回滚。

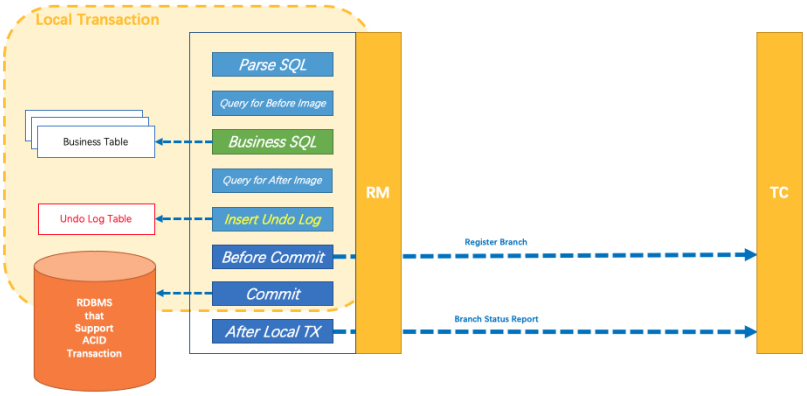
4.TC 驱动 RM 们将 XID 对应的自己的本地事务进行提交还是回滚。

2.2.1 设计思路

AT模式的核心是对业务无侵入，是一种改进后的两阶段提交

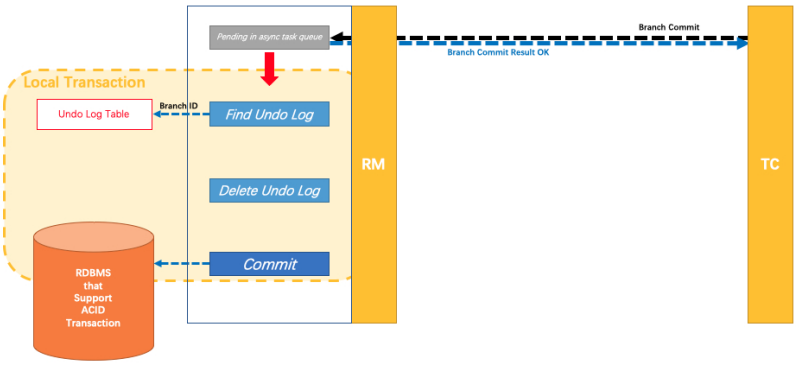
第一阶段

业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源。核心在于对业务sql进行解析，转换成undolog，并同时入库。

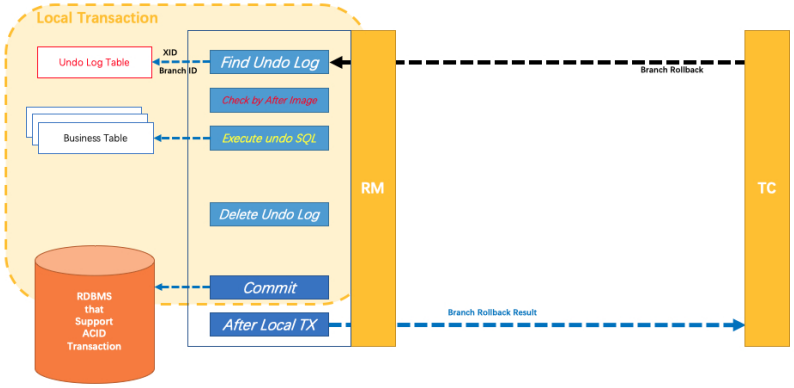


第二阶段

分布式事务操作成功，则TC通知RM异步删除undolog



分布式事务操作失败，TM向TC发送回滚请求，RM 收到协调器TC发来的回滚请求，通过 XID 和 Branch ID 找到相应的回滚日志记录，通过回滚记录生成反向的更新 SQL 并执行，以完成分支的回滚。



2.2.2 设计亮点

相比与其它分布式事务框架，Seata架构的亮点主要有几个:

1. 应用层基于SQL解析实现了自动补偿，从而最大程度的降低业务侵入性；
2. 将分布式事务中TC（事务协调者）独立部署，负责事务的注册、回滚；
3. 通过全局锁实现了写隔离与读隔离。

2.2.3 存在的问题

性能损耗

一条Update的SQL，则需要全局事务xid获取（与TC通讯）、before image（解析SQL，查询一次数据库）、after image（查询一次数据库）、insert undo log（写一次数据库）、before commit（与TC通讯，判断锁冲突），这些操作都需要一次远程通讯RPC，而且是同步的。另外undo log写入时blob字段的插入性能也是不高的。每条写SQL都会增加这么多开销,粗略估计会增加5倍响应时间。

性价比

为了进行自动补偿，需要对所有交易生成前后镜像并持久化，可是在实际业务场景下，这个是成功率有多高，或者说分布式事务失败需要回滚的有多少比率？按照二八原则预估，为了20%的交易回滚，需要将80%的成功交易的响应时间增加5倍，这样的代价相比于让应用开发一个补偿交易是否是值得？

全局锁

热点数据

相比XA，Seata 虽然在一阶段成功后会释放数据库锁，但一阶段在commit前全局锁的判定也拉长了对数据锁的占有时间，这个开销比XA的prepare低多少需要根据实际业务场景进行测试。全局锁的引入实现了隔离性，但带来的问题就是阻塞，降低并发性，尤其是热点数据，这个问题会更加严重。

回滚锁释放时间

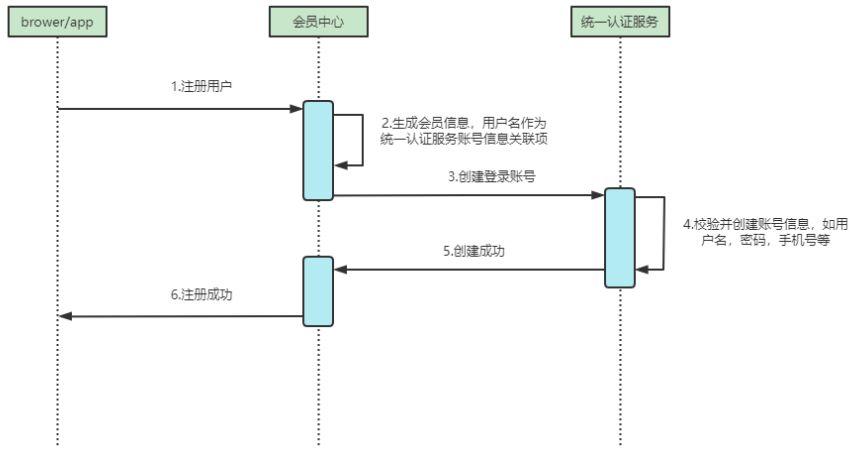
Seata在回滚时，需要先删除各节点的undo log，然后才能释放TC内存中的锁，所以如果第二阶段是回滚，释放锁的时间会更长。

死锁问题

Seata的引入全局锁会额外增加死锁的风险，但如果出现死锁，会不断进行重试，最后靠等待全局锁超时，这种方式并不优雅，也延长了对数据库锁的占有时间。

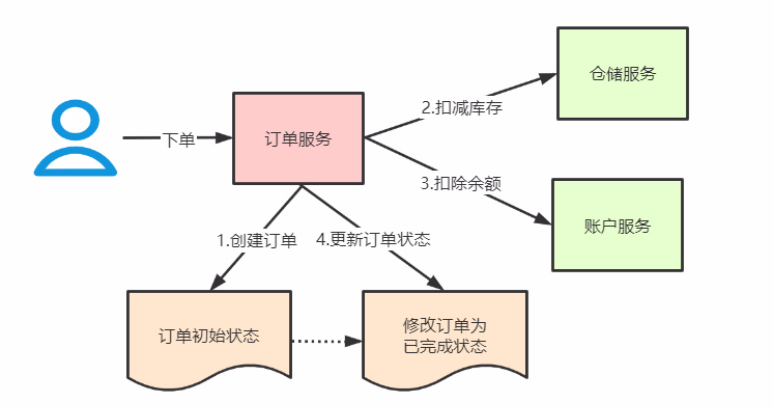
2.2.4 业务场景

场景一：以注册账号为例，采用用户、账号分离设计的方式(这样设计的好处是，当用户的业务信息发生变化时，不会影响的认证、授权等系统机 制)，因此需要保证用户信息与账号信息的一致性。



场景二：用户下单，整个业务逻辑由三个微服务构成：

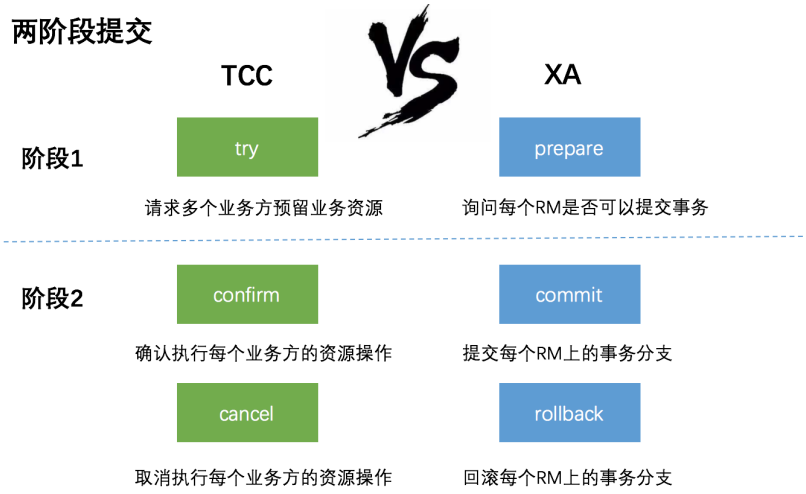
- 仓储服务：对给定的商品扣除库存数量。
- 订单服务：根据采购需求创建订单。
- 帐户服务：从用户帐户中扣除余额。



2.3 柔性事务TCC（Try-Confirm-Cancel）方案实现

TCC是比较常用的一种柔性事务方案。开源的TCC框架:

- **Tcc-Transaction**
- **Hmily**
- **ByteTCC**
- **EasyTransaction**
- **Seata TCC**



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。

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两阶段提交的区别是：

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。

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事务的优缺点：

优点：XA两阶段提交资源层面的，而TCC实际上把资源层面二阶段提交上提到了业务层面来实现。有效了的避免了XA两阶段提交占用资源锁时间过长导致的性能地下问题。

相对于 AT 模式，TCC 模式对业务代码有一定的侵入性，但是 TCC 模式无 AT 模式的全局行锁，TCC 性能会比 AT 模式高很多。

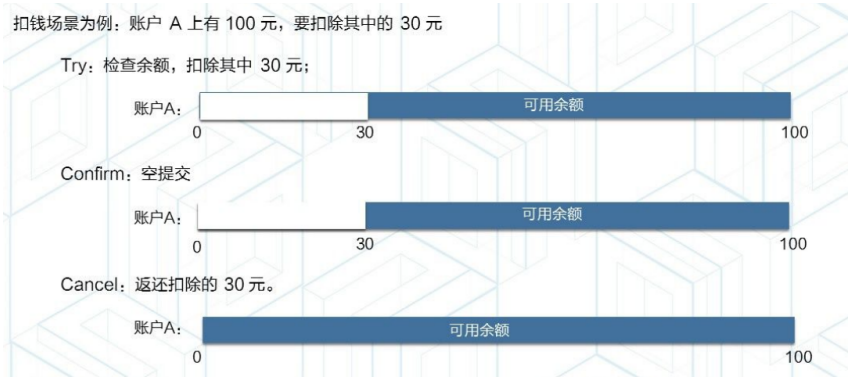
缺点：主业务服务和从业务服务都需要进行改造，从业务方改造成本更高。原来只需要提供一个接口，现在需要改造成try、confirm、canel 3个接口，开发成本高。

2.3.1 TCC设计注意事项

业务模型分 2 阶段设计

用户接入 TCC ，最重要的是考虑如何将自己的业务模型拆成两阶段来实现。

以“扣钱”场景为例，场景为 A 转账 30 元给 B，A和B账户在不同的服务。 在接入 TCC 前，对 A 账户的扣钱，只需一条更新账户余额的 SQL 便能完成；但是在接入 TCC 之后，用户就需要考虑如何将原来一步就能完成的扣钱操作，拆成两阶段，实现成三个方法，并且保证一阶段 Try 成功的话 二阶段 Confirm 一定能成功。



思考： 下面三种方案哪种更合理?

方案1：

账户A

```
try：
 检查余额是否够30元
 扣减30元
confirm：
 空
cancel：
 增加30元
```

账号B

```
try：
 增加30元
confirm：
 空
cancel：
 减少30元
```

方案2

账户A

```
try：
 检查余额是否够30元
 扣减30元
confirm：
 空
cancel：
 增加30元
```

账号B

```
try：
 空
confirm：
 增加30元
cancel：
 空
```

方案3

账户A

```
try：
 try幂等校验
 try悬挂处理
 检查余额是否够30元
 扣减30元
confirm：
 空
cancel：
```

```
cancel幂等校验
cancel空回滚处理
增加可用余额30元
```

账户B

```
try:
 空
confirm:
 confirm幂等校验
 正式增加30元
cancel:
 空
```

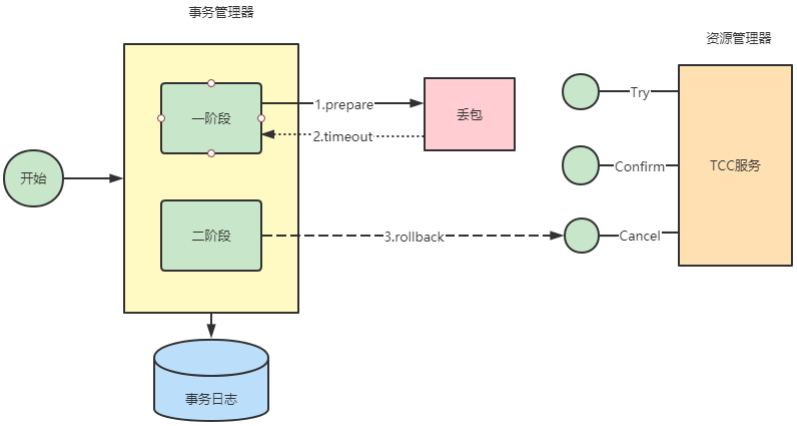
TCC异常控制

在微服务架构下，很有可能出现网络超时、重发，机器宕机等一系列的异常，出现空回滚、幂等、悬挂的问题。

TCC 设计 - 允许空回滚

空回滚：在没有调用 TCC 资源 Try 方法的情况下，调用了二阶段的 Cancel 方法，Cancel 方法需要识别出这是一个空回滚，然后直接返回成功。

空回滚出现的原因：Try超时（丢包），分布式事务回滚触发Cancel，出现未收到Try，收到Cancel的情况

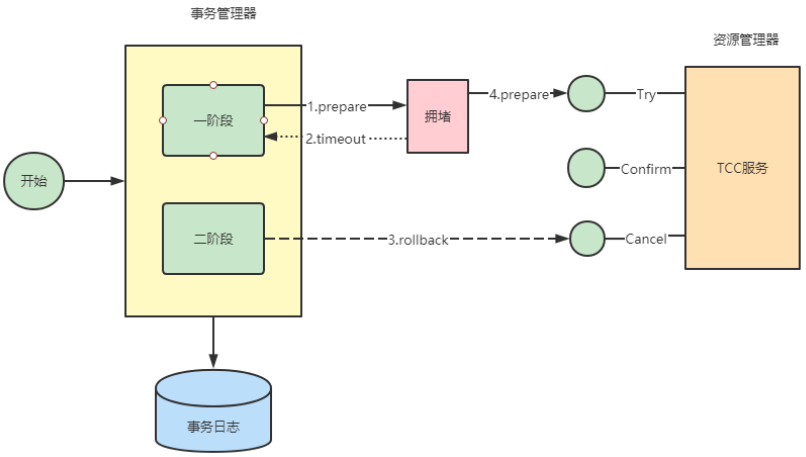


TCC 设计 - 防悬挂控制

要运行空回滚，但要拒绝空回滚之后的Try操作

悬挂：Cancel比Try先执行

悬挂出现的原因：Try超时（拥堵），分布式事务回滚触发Cancel，之后拥堵的Try到达

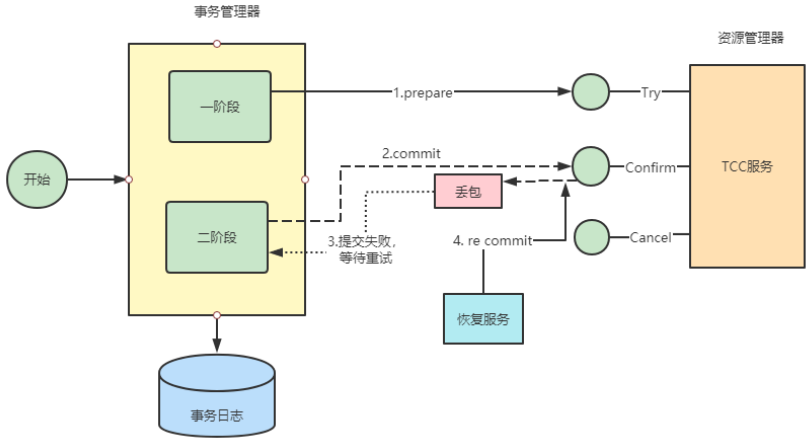


TCC 设计 - 幂等控制

Try，Confirm，Cancel都需要保证幂等性。

因为网络抖动或拥堵可能会超时，事务管理器会对资源进行重试操作，所以很可能一个业务操作会被重复调用，为了不因为重复调用而多次占用资源，需要对服务设计时进行幂等控制，通常我们可以用事务 xid 或业务主键判重来控

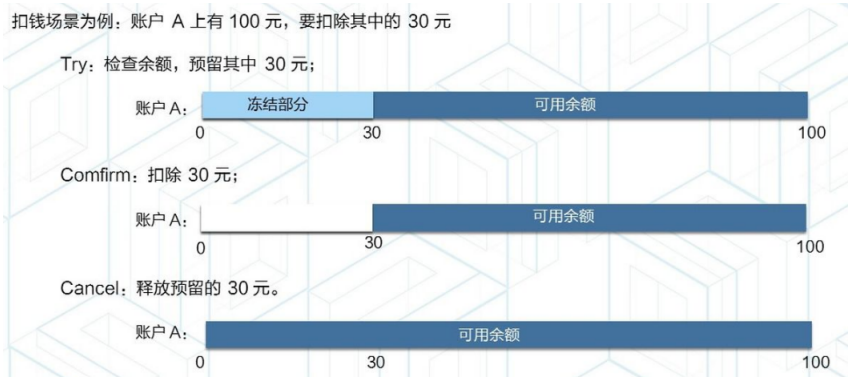
制。



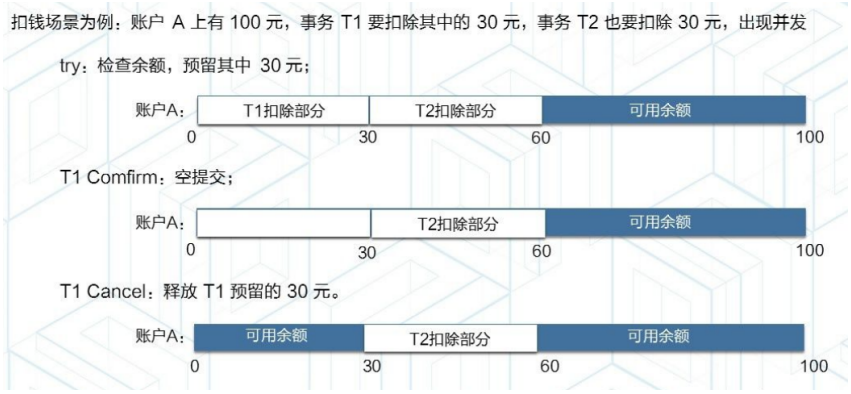
根据自身的业务模型进行控制并发（隔离性）

思考：之前的模型是否存在并发问题?

对业务模型进行优化，在业务模型中增加冻结金额字段，用来表示账户有多少金额处以冻结状态。

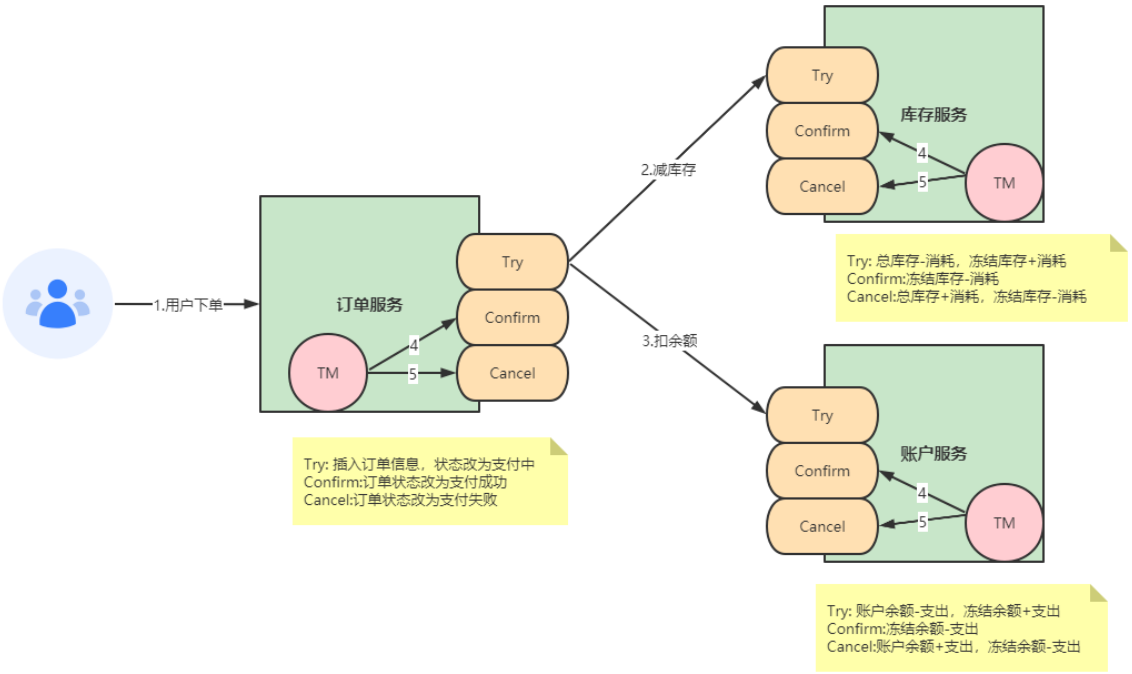


并发场景：



场景二：用户下单，整个业务逻辑由三个微服务构成：

- 仓储服务：对给定的商品扣除库存数量。
- 订单服务：根据采购需求创建订单。
- 帐户服务：从用户帐户中扣除余额。



4. 柔性事务：可靠消息最终一致性方案实现

可靠消息最终一致性方案是指当事务发起执行完成本地事务后并发出一条消息，事务参与方（消息消费者）一定能够接收消息并处理事务成功，此方案强调的是只要消息发给事务参与方最终事务要达到一致。

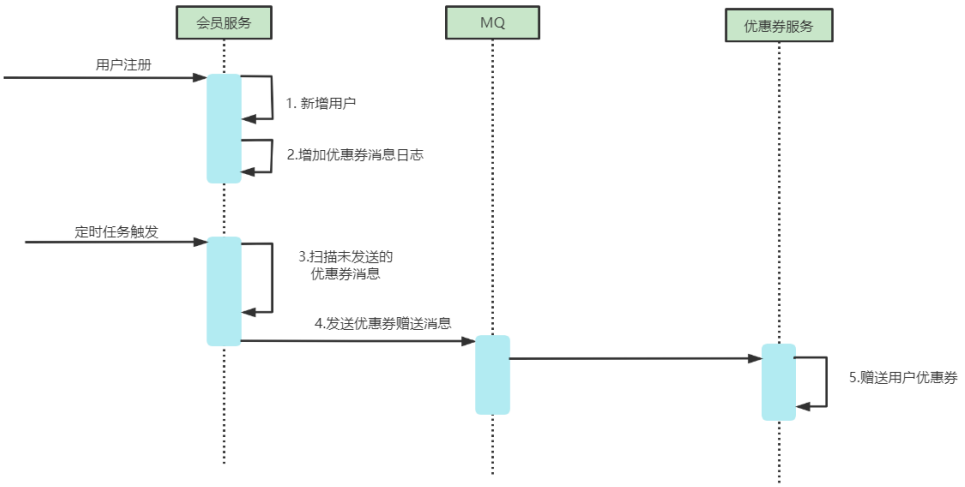
4.1 本地消息表方案

本地消息表这个方案最初是eBay提出的，此方案的核心是通过本地事务保证数据业务操作和消息的一致性，然后通过定时任务将消息发送至消息中间件，待确认消息发送给消费方成功再将消息删除。

下面以注册送优惠券为例来说明：

共有两个微服务交互，会员服务和优惠券服务，用户服务负责添加用户，优惠券服务负责赠送优惠券。

交互流程如下：



1、用户注册

用户服务在本地事务新增用户和增加“优惠券消息日志”。（用户表和消息表通过本地事务保证一致）

下面是伪代码

```
begin transaction;
// 1.新增用户
// 2.存储优惠券消息日志
commit transation;
```

这种情况下，本地数据库操作与存储优惠券消息日志处于同一事务中，本地数据库操作与记录消息日志操作具备原子性。

2、定时任务扫描日志

如何保证将消息发送给消息队列呢?

经过第一步消息已经写到消息日志表中，可以启动独立的线程，定时对消息日志表中的消息进行扫描并发送至消息中间件，在消息中间件反馈发送成功后删除该消息日志，否则等待定时任务下一周期重试。

3、消费消息

如何保证消费者一定能消费到消息呢?

这里可以使用MQ的ack（即消息确认）机制，消费者监听MQ，如果消费者接收到消息并且业务处理完成后向MQ发送ack（即消息确认），此时说明消费者正常消费消息完成，MQ将不再向消费者推送消息，否则消费者会不断重试向消费者来发送消息。

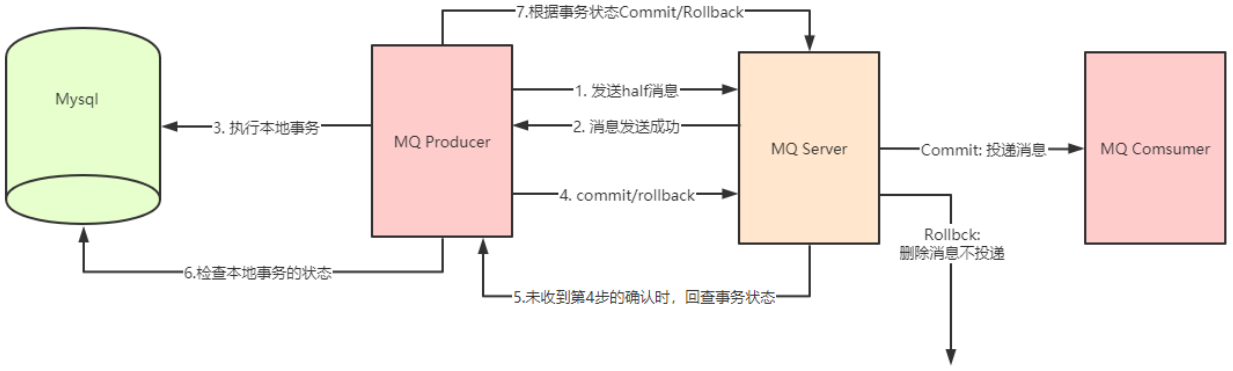
优惠券服务接收到“赠送优惠券”消息，开始赠送用户优惠券，成功后消息中间件回应ack，否则消息中间件将重复投递此消息。

由于消息会重复投递，优惠券服务的“赠送优惠券”功能需要实现幂等性。

4.2 Rocketmq事务消息实现

RocketMQ事务消息设计则主要是为了解决Producer端的消息发送与本地事务执行的原子性问题，RocketMQ的设计中broker与producer端的双向通信能力，使得broker天生可以作为一个事务协调者存在；而RocketMQ本身提供的存储机制为事务消息提供了持久化能力；RocketMQ的高可用机制以及可靠消息设计则为事务消息在系统发生异常时依然能够保证达成事务的最终一致性。

在RocketMQ 4.3后实现了完整的事务消息，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，解决Producer端的消息发送与本地事务执行的原子性问题。



执行流程如下：

为方便理解我们以注册送优惠券的例子来描述整个流程。

Producer即MQ发送方，本例中是用户服务，负责新增用户。MQ订阅方即消息消费方，本例中是优惠券服务，负责新增优惠券。

1、Producer发送事务消息

Producer（MQ发送方）发送事务消息至MQ Server，MQ Server将消息状态标记为Prepared（预览状态），注意此时这条消息消费者（MQ订阅方）是无法消费到的。

2、MQ Server回应消息发送成功

MQ Server接收到Producer发送给的消息则回应发送成功表示MQ已接收到消息。

3、Producer执行本地事务

Producer端执行业务代码逻辑，通过本地数据库事务控制。

本例中，Producer执行添加用户操作。

4、消息投递

若Producer本地事务执行成功则自动向MQ Server发送commit消息，MQ Server接收到commit消息后将“增加优惠券消息”状态标记为可消费，此时MQ订阅方（优惠券服务）即正常消费消息；
若Producer 本地事务执行失败则自动向MQ Server发送rollback消息，MQ Server接收到rollback消息后将删除“增加优惠券消息”。
MQ订阅方（优惠券服务）消费消息，消费成功则向MQ回应ack，否则将重复接收消息。这里ack默认自动回应，即程序执行正常则自动回应ack。
5、事务回查
如果执行Producer端本地事务过程中，执行端挂掉，或者超时，MQ Server将会不停的询问同组的其他Producer来获取事务执行状态，这个过程叫**事务回查**。MQ Server会根据事务回查结果来决定是否投递消息。
以上主干流程已由RocketMQ实现，对用户则来说，用户需要分别实现本地事务执行以及本地事务回查方法，因此只需关注本地事务的执行状态即可。
RocketMQ提供RocketMQLocalTransactionListener接口：

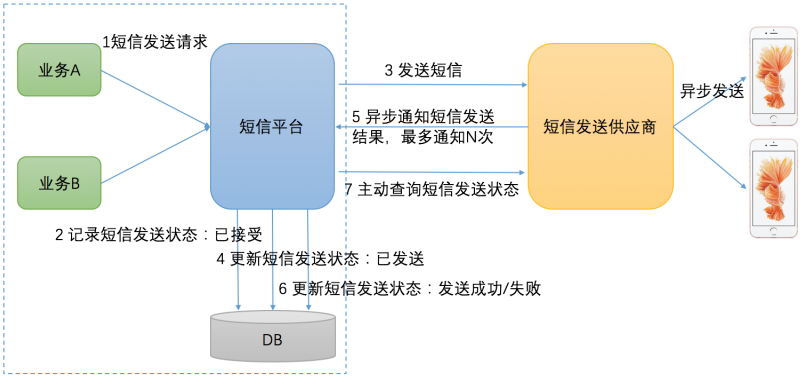
```
public interface RocketMQLocalTransactionListener {
 /**
 发送prepare消息成功此方法被回调，该方法用于执行本地事务
 @param msg 回传的消息，利用transactionId即可获取到该消息的唯一Id
 @param arg 调用send方法时传递的参数，当send时候若有额外的参数可以传递到send方法中，这里能获取到
 @return 返回事务状态，COMMIT ：提交 ROLLBACK ：回滚 UNKNOW ：回调
 */
 RocketMQLocalTransactionState executeLocalTransaction（Message msg，Object arg）；
 /**
  @param msg 通过获取transactionId来判断这条消息的本地事务执行状态
  @return 返回事务状态，COMMIT ：提交 ROLLBACK ：回滚 UNKNOW ：回调
  */
  RocketMQLocalTransactionState checkLocalTransaction（Message msg）；
}
```

以下是RocketMQ提供用于发送事务消息的API：

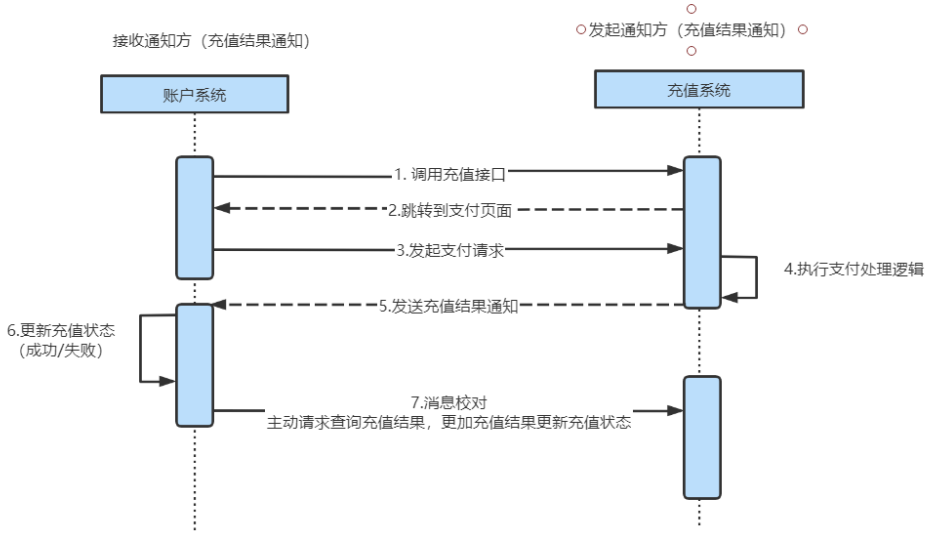
```
TransactionMQProducer producer = new TransactionMQProducer("ProducerGroup");
producer.setNamesrvAddr("127.0.0.1:9876");
producer.start();
// 设置TransactionListener实现
producer.setTransactionListener(transactionListener);
// 发送事务消息
SendResult sendResult = producer.sendMessageInTransaction(msg, null);
```

5. 柔性事务：最大努力通知

最大努力通知型(Best-effort delivery)是最简单的一种柔性事务，是分布式事务中对一致性要求最低的一种，适用于一些最终一致性时间敏感度低的业务，且被动方处理结果 不影响主动方的处理结果。典型的使用场景：如银行通知、商户通知等。
最大努力通知型的实现方案，一般符合以下特点：
1、不可靠消息：业务活动主动方，在完成业务处理之后，向业务活动的被动方发送消息，直到通知N次后不再通知，允许消息丢失(不可靠消息)。
2、定期校对：业务活动的被动方，根据定时策略，向业务活动主动方查询(主动方提供查询接口)，恢复丢失的业务消息。
最大努力通知方案需要实现如下功能：
1、消息重复通知机制。
2、消息校对机制。
场景：比如公司内部实现了短信平台，所有的业务方都接入这个短信平台，来实现发送短信的功能。



场景二： 充值的例子



思考：最大努力通知与可靠消息最终一致性有什么区别?

- 思想不同

可靠消息最终一致性，发起通知方需要保证将消息发出去，并且将消息发到接收通知方，消息的可靠性关键由发起通知方来保证。

最大努力通知，发起通知方尽最大的努力将业务处理结果通知为接收通知方，但是可能消息接收不到，此时需要接收通知方主动调用发起通知方的接口查询业务处理结果，通知的可靠性关键在接收通知方。

- 两者的业务应用场景不同

可靠消息最终一致性关注的是交易过程的事务一致，以异步的方式完成交易。

最大努力通知关注的是交易后的通知事务，即将交易结果可靠的通知出去。

- 技术解决方向不同

可靠消息最终一致性要解决消息从发出到接收的一致性，即消息发出并且被接收到。

最大努力通知无法保证消息从发出到接收的一致性，只提供消息接收的可靠性机制。可靠机制是，最大努力的将消息通知给接收方，当消息无法被接收方接收时，由接收方主动查询消息（业务处理结果）

文档：13 分布式事务解决方案实战.note
链接：http://note.youdao.com/noteshare?id=d5b55c5c8c1a5c516d620aa7492712ae&sub=AC600E21FE0A43C0B1DE94ADDF47B3E1