

## 消息系统避免分布式事务

### 一、本地事务

以支付宝转账余额宝为例，假设有

- 支付宝账户表：A (id, userId, amount)
- 余额宝账户表：B (id, userId, amount)
- 用户的 userId=1;

从支付宝转账 1 万块钱到余额宝的动作分为两步：

• 1) 支付宝表扣除 1 万：update A set amount=amount-10000 where userId=1;

• 2) 余额宝表增加 1 万：update B set amount=amount+10000 where userId=1;

如何确保支付宝余额宝收支平衡呢？有人说这个很简单嘛，可以用事务解决。

Begin transaction

update A set amount=amount-10000 where userId=1;

update B set amount=amount+10000 where userId=1;

End transaction

commit;

非常正确，如果你使用 spring 的话一个注解就能搞定上述事务功能。

@Transactional(rollbackFor=Exception.class)

public void update() {

updateATable(); //更新 A 表

updateBTable(); //更新 B 表

```
}
```

如果系统规模较小，数据表都在一个数据库实例上，上述本地事务方式可以很好地运行，但是如果系统规模较大，比如支付宝账户表和余额宝账户表显然不会在同一个数据库实例上，他们往往分布在不同的物理节点上，这时本地事务已经失去用武之地。既然本地事务失效，分布式事务自然就登上舞台。

## 二、分布式事务—两阶段提交协议

两阶段提交协议（Two-phase Commit, 2PC）经常被用来实现分布式事务。一般分为协调器 C 和若干事务执行者 Si 两种角色，这里的事务执行者就是具体的数据库，协调器可以和事务执行器在一台机器上。

1. 我们的应用程序（client）发起一个开始请求到 TC；
2. TC 先将<prepare>消息写到本地日志，之后向所有的 Si 发起<prepare>消息。以支付宝转账到余额宝为例，TC 给 A 的 prepare 消息是通知支付宝数据库相应账目扣款 1 万，TC 给 B 的 prepare 消息是通知余额宝数据库相应账目增加 1w。为什么在执行任务前需要先写本地日志，主要是为了故障后恢复用，本地日志起到现实生活中凭证 的效果，如果没有本地日志（凭证），出问题容易死无对证；
3. Si 收到<prepare>消息后，执行具体本机事务，但不会进行 commit，如果成功返回<yes>，不成功返回<no>。同理，返回前都应把要返回的消息写到日志里，当作凭证。
4. TC 收集所有执行器返回的消息，如果所有执行器都返回 yes，那么给所有执行器发送 commit 消息，执行器收到 commit 后执行本地事务的 commit 操作；如果有任一个执行器返回 no，那么给所有执行器发送 abort 消息，执行器收到 abort 消息后执行事务 abort 操作。

注：TC 或 Si 把发送或接收到的消息先写到日志里，主要是为了故障后恢复用。如某一 Si 从故障中恢复后，先检查本机的日志，如果已收到<commit>，则提交，如果<abort>则回滚。如果是<yes>，则再向 TC 询问一下，确定下一步。如果什么都没有，则很可能在<prepare>阶段 Si 就崩溃了，因此需要回滚。

现如今实现基于两阶段提交的分布式事务也没那么困难了，如果使用 java，那么可以使用开源软件 atomikos 来快速实现。不过但凡使用过的上述两阶段提交的同学都可以发现性能实在是太差，根本不适合高并发的系统。为什么？

两阶段提交涉及多次节点间的网络通信，通信时间太长！

事务时间相对于变长了，锁定的资源的时间也变长了，造成资源等待时间也增加好多！

正是由于分布式事务存在很严重的性能问题，大部分高并发服务都在避免使用，往往通过其他途径来解决数据一致性问题。

### 三、使用消息队列来避免分布式事务

如果仔细观察生活的话，生活的很多场景已经给了我们提示。比如在北京很有名的姚记炒肝点了炒肝并付了钱后，他们并不会直接把你点的炒肝给你，而是给你一张小票，然后让你拿着小票到出货区排队去取。为什么他们要将付钱和取货两个动作分开呢？原因很多，其中一个很重要的原因是为了使他们接待能力增强（并发量更高）。还是回到我们的问题，只要这张小票在，你最终是能拿到炒肝的。同理转账服务也是如此，当支付宝账户扣除 1 万后，我们只要生成一个凭证（消息）即可，这个凭证（消息）上写着“让余额宝账户增加 1 万”，只要这个凭证（消息）能可靠保存，我们最终是可以拿着这个凭证（消息）让余额宝账户增加 1 万的，即我们能依靠这个凭证（消息）完成最终一致性。

#### 1. 如何可靠保存凭证（消息）

有两种方法：

##### ① 业务与消息耦合的方式

支付宝在完成扣款的同时，同时记录消息数据，这个消息数据与业务数据保存在同一数据库实例里（消息记录表表名为 message）。

Begin transaction

```
update A set amount=amount-10000 where userId=1;
```

```
insert into message(userId, amount,status) values(1, 10000, 1);
```

End transaction

commit;

上述事务能保证只要支付宝账户里被扣了钱，消息一定能保存下来。当上述事务提交成功后，我们通过实时消息服务将此消息通知余额宝，余额宝处理成功后发送回复成功消息，支付宝收到回复后删除该条消息数据。

##### ② 业务与消息解耦方式

上述保存消息的方式使得消息数据和业务数据紧耦合在一起，从架构上看不够优雅，而且容易诱发其他问题。为了解耦，可以采用以下方式。1）支付宝在扣款事务提交之前，向实时消息服务请求发送消息，实时消息服务只记录消息数据，而不真正发送，只有消息发送成功后才会提交事务；2）当支付宝扣款事务被提交成功后，向实时消息服务确认发送。只有在得到确认发送指令后，实时消息服务才真正发送该消息；3）当支付宝扣款事务提交失败回滚后，向实时消息服务

取消发送。在得到取消发送指令后，该消息将不会被发送；4）对于那些未确认的消息或者取消的消息，需要有一个消息状态确认系统定时去支付宝系统查询这个消息的状态并进行更新。为什么需要这一步骤，举个例子：假设在第 2 步支付宝扣款事务被成功提交后，系统挂了，此时消息状态并未被更新为“确认发送”，从而导致消息不能被发送。优点：消息数据独立存储，降低业务系统与消息系统间的耦合；缺点：一次消息发送需要两次请求；业务处理服务需要实现消息状态回查接口。

## 2. 如何解决消息重复投递的问题

还有一个很严重的问题就是消息重复投递，以我们支付宝转账到余额宝为例，如果相同的消息被重复投递两次，那么我们余额宝账户将会增加 2 万而不是 1 万了。为什么相同的消息会被重复投递？比如余额宝处理完消息 msg 后，发送了处理成功的消息给支付宝，正常情况下支付宝应该要删除消息 msg，但如果支付宝这时候悲剧的挂了，重启后一看消息 msg 还在，就会继续发送消息 msg。解决方法很简单，在余额宝这边增加消息应用状态表（message\_apply），通俗来说就是个账本，用于记录消息的消费情况，每次来一个消息，在真正执行之前，先去消息应用状态表中查询一遍，如果找到说明是重复消息，丢弃即可，如果没找到才执行，同时插入到消息应用状态表（同一事务）。

```
for each msg in queue
```

```
Begin transaction
```

```
select count(*) as cnt from message_apply where msg_id=msg.msg_id;
```

```
if cnt==0 then
```

```
update B set amount=amount+10000 where userId=1;
```

```
insert into message_apply(msg_id) values(msg.msg_id);
```

```
End transaction
```

```
commit;
```