链滴

分布式事务终极解决方案探讨

作者：vvsuperman
原文链接：https://ld246.com/article/1515511635962
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

一 小小推广

讲座

本话题已收入视频讲座《Spring Cloud分布式事务解决方案》大家不妨围观下

开源项目

我们利用消息队列实现了分布式事务的最终一致性解决方案，请大家围观。可以参考Github CoolM源码，项目支持网站: http://rabbitmq.org.cn 最新文章或实现会更新在上面

二 前言

阿里2017云栖大会《破解世界性技术难题！GTS让分布式事务简单高效》中，阿里声称提出了一种破解**界性难题**之分布式事务的终极解决方案，无论是可靠性、还是处理速率都领先于市面上所有的技术。令人遗憾的是一来项目**未开源**，二来还必须**依赖阿里云**的分布式数据库。毕竟，**吃饭的家伙可不能轻示人嘛。**

虽然如此，但《世界难题...》一文中对事务还是归纳的还是蛮到位的:“一个看似简单的功能，内部可需要调用多个“服务”并操作多个数据库或分片来实现，单一技术手段和解决方案已无法满足这些复应用场景。因此，分布式系统架构中分布式事务是一个绕不过去的挑战。

什么是分布式事务？简单的说，就是一次大操作由不同小操作组成，这些小操作**分布在不同服务器上分布式事务需要保证这些小操作要么全部成功，要么全部失败。**”

举个栗子

你上Taobao买东西，需要先扣钱，然后商品库存-1吧。但扣款和库存分别属于两个服务，这两个服中间要经过网络、网关、主机等一系列中间层，万一任何一个地方出了问题，比如网络抖动、突发异等待，都会导致不一致，比如扣款成功了，但是库存没-1，就会出现超卖的现象，而这就是分布式事需要解决的问题

三 2阶段提交(2PC, 3PC等)

2阶段提交是分布式事务传统解决方案，先进为止还广泛存在。当一个事务跨越多个节点时，为了保事务ACID特性，需要引入一个作为**协调者**来统一掌控所有节点(称作参与者)的操作结果并最终指示这节点是否要把操作结果进行真正的提交(比如将更新后的数据写入磁盘等等)。因此，二阶段提交的算思路可以概括为：**参与者**将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与是否要提交操作还是中止操作。

以开会为例

甲乙丙丁四人要组织一个会议，需要确定会议时间，不妨设甲是协调者，乙丙丁是参与者。

投票阶段

（1）甲发邮件给乙丙丁，周二十点开会是否有时间；

（2）甲回复有时间；

（3）乙回复有时间；

（4）丙迟迟不回复，此时对于这个活动，甲乙丙均处于阻塞状态，算法无法继续进行；

（5）丙回复有时间（或者没有时间）；

提交阶段

（1）协调者甲将收集到的结果反馈给乙丙丁（什么时候反馈，以及反馈结果如何，在此例中取决与的时间与决定）；

（2）乙收到；

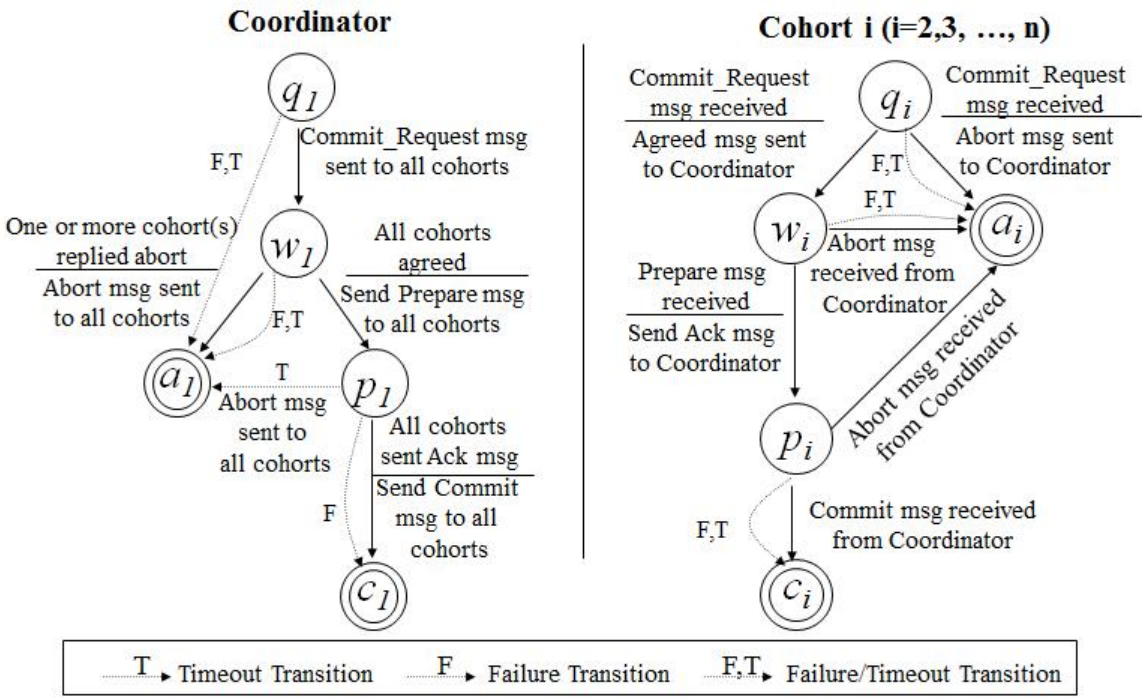
（3）丙收到；

（4）丁收到；

不仅要锁住参与者的所有资源，而且要锁住协调者资源，开销大。一句话总结就是：2PC效率很低，高并发很不友好。

引用《世界性难题...》一文原话 "国外具有几十年历史和技术沉淀的基于XA模型的商用分布式事务产，在相同软硬件条件下，开启分布式事务后吞吐经常有数量级的下降。"

此外还有三阶段提交



大家有兴趣的不妨研究下

四 柔性事务

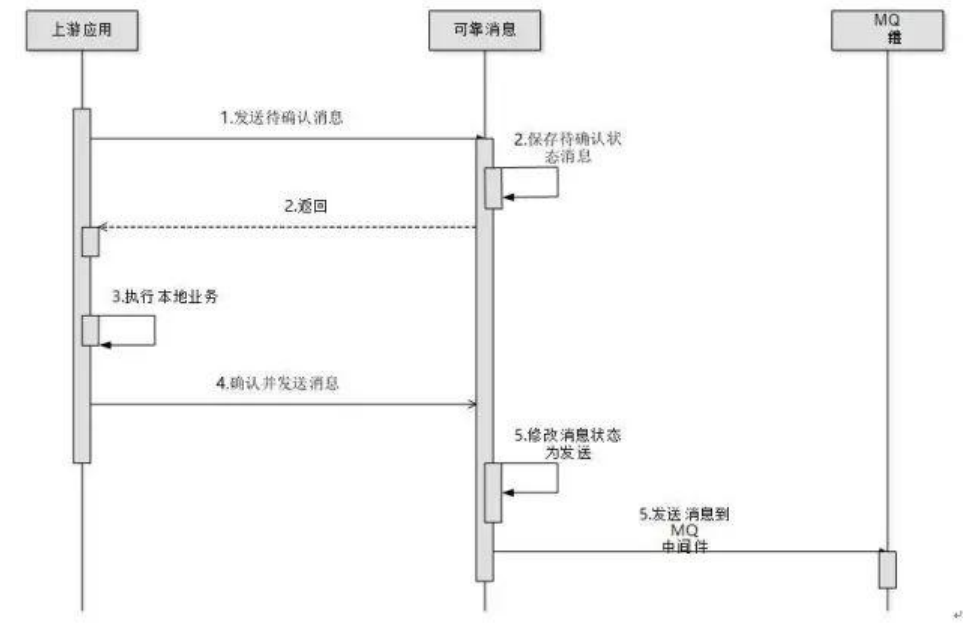
所谓**柔性事务**是相对强制锁表的刚性事务而言。流程入下：服务器A的事务如果执行顺利，那么事务就先行提交，如果事务B也执行顺利，则事务B也提交，整个事务就算完成。但是如果事务B执行失败事务B本身回滚，这时事务A已经被提交，所以需要执行一个补偿操作，将已经提交的事务A执行的操作反操作，恢复到未执行前事务A的状态。

缺点是业务侵入性太强，还要补偿操作，缺乏普遍性，没法大规模推广。

五 消息最终一致性解决方案之RocketMQ

目前基于消息队列的解决方案有阿里的RocketMQ，它实现了半消息的解决方案,有点类似于Paxos算法具体流程如下

第一阶段：上游应用执行业务并发送MQ消息



1.上游应用发送待确认消息到可靠消息系统

2.可靠消息系统保存待确认消息并返回

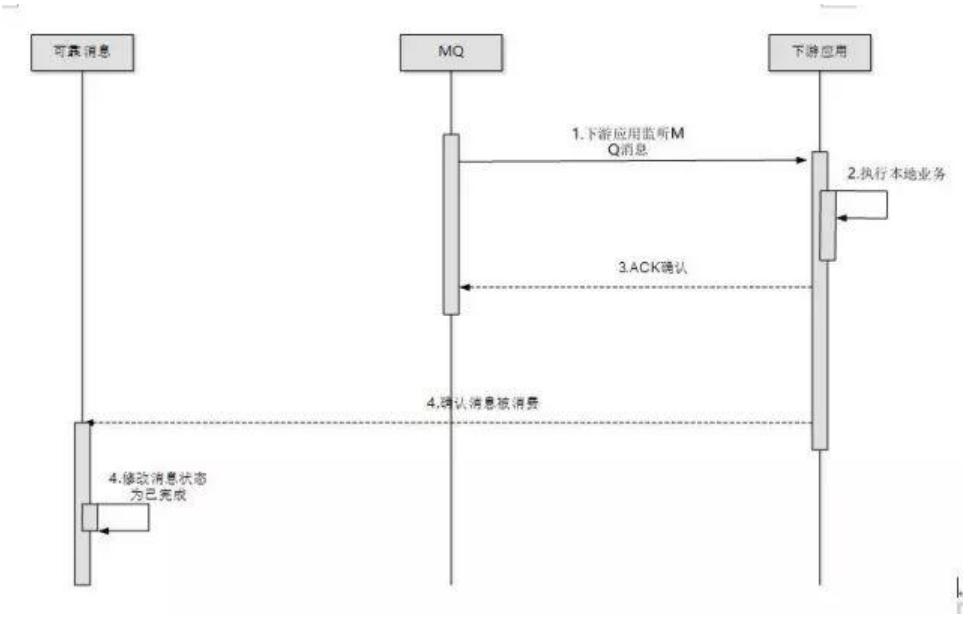
3.上游应用执行本地业务

4.上游应用通知可靠消息系统确认业务已执行并发送消息。

5.可靠消息系统修改消息状态为发送状态并将消息投递到 MQ 中间件

第二阶段：下游应用监听 MQ 消息并执行业务

下游应用监听 MQ 消息并执行业务，并且将消息的消费结果通知可靠消息服务。



1. 下游应用监听 MQ 消息组件并获取消息

2. 下游应用根据 MQ 消息体信息处理本地业务

3. 下游应用向 MQ

原文链接：分布式事务终极解决方案探讨

4. 确认消息被消费

5. 下游应用通知可靠消息系统消息被成功消费，可靠消息将该消息状态更改为已完成

RocketMQ貌似是一种先进的实现方案了，但问题是缺乏文档，无论是在Apache项目主页，还是在里的页面上，最多只告诉你如何用，而原理性或者指导性的东西非常缺乏。

当然，如果你在阿里云上专门购买了RocketMQ服务，想必是另当别论了。但如果你试图在自己的服环境中部署和使用，想必要历经相当大的学习曲线。毕竟是人家吃饭的家伙嘛

六 消息最终一致性解决方案之RabbitMQ实现

RabbitMQ遵循了AMQP规范，用消息确认机制来保证：**只要消息发送，就能确保被消费者消费**来做了消息最终一致性。而且开源，文档还异常丰富，貌似是实现分布式事务的良好载体

6.1 RabbitMQ消息确认机制

rabbitmq的整个发送过程如下

1. **生产者**发送消息到**消息服务**

2. 如果消息 **落地持久化完成**，则返回一个标志给生产者。生产者拿到这个**确认**后，才能放心的说消终于成功发到消息服务了。否则进入异常处理流程。

```
rabbitTemplate.setConfirmCallback((correlationData, ack, cause) -> {
if (!ack) {
//try to resend msg
} else {
//delete msg in db
}
});
```

3. **消息服务**将消息发送给**消费者**

4. 消费者接受并处理消息，如果处理成功则 **手动确认**。当消息服务拿到这个确认后，才放心的说终消费完成了。否则重发，或者进入异常处理。

```
final Consumer consumer = new DefaultConsumer(channel) {
@Override
public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties pr
perties, byte[] body) throws IOException {
String message = new String(body, "UTF-8");

System.out.println(" [x] Received '" + message + "'");
try {
  doWork(message);
} finally {
   //确认收到消息
  channel.basicAck(envelope.getDeliveryTag(), false);
    }
  }
```

```
};
```

6.2 异常

我们来看看可能发送异常的四种

1. **直接无法到达消息服务**

网络断了，抛出异常，业务直接回滚即可。如果出现connection closed错误，直接增加 connection 即可

```
connectionFactory.setChannelCacheSize(100);
```

2. **消息已经到达服务器，但返回的时候出现异常**

rabbitmq提供了确认ack机制，可以用来确认消息是否有返回。因此我们可以在发送前在db中(内存关系型数据库)先存一下消息，如果ack异常则进行重发

```
/**confirmcallback用来确认消息是否有送达消息队列*/
rabbitTemplate.setConfirmCallback((correlationData, ack, cause) -> {
if (!ack) {
    //try to resend msg
} else {
    //delete msg in db
}
});
 /**若消息找不到对应的Exchange会先触发returncallback */
rabbitTemplate.setReturnCallback((message, replyCode, replyText, tmpExchange, tmpRoutin
Key) -> {
    try {
        Thread.sleep(Constants.ONE_SECOND);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    log.info("send message failed: " + replyCode + " " + replyText);
    rabbitTemplate.send(message);
});
```

3. 消息送达后，消息服务自己挂了

如果设置了消息持久化，那么ack= true是在**消息持久化**完成后，就是存到硬盘上之后再发送的，确消息已经存在硬盘上，万一消息服务挂了，消息服务恢复是能够再重发消息

4. 未送达消费者

消息服务收到消息后，消息会处于"UNACK"的状态，直到客户端确认消息

```
channel.basicQos(1); // accept only one unack-ed message at a time (see below)
final Consumer consumer = new DefaultConsumer(channel) {
  @Override
  public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties
roperties, byte[] body) throws IOException {
    String message = new String(body, "UTF-8");
```

```
System.out.println(" [x] Received '" + message + "'");
try {
  doWork(message);
} finally {
   //确认收到消息
  channel.basicAck(envelope.getDeliveryTag(), false);
}
  }
};
boolean autoAck = false;
channel.basicConsume(TASK_QUEUE_NAME, autoAck, consumer);
```

5. **确认消息丢失**

消息返回时假设确认消息丢失了，那么消息服务会重发消息。注意，如果你设置了autoAck= false，又没应答 channel.baskAck也没有应答 channel.baskNack，那么会导致非常严重的错误：消息队列被堵塞住，所以，无论如何都必须应答

6. **消费者业务处理异常**

消息监听接受消息并处理，假设抛异常了，第一阶段事物已经完成，如果要配置回滚则过于麻烦，即做事务补偿也可能事务补偿失效的情况，所以这里可以做一个重复执行，比如guava的retry，设置一指数时间来循环执行，如果n次后依然失败，发邮件、短信，用人肉来兜底。

七 总结

《世界性难题...》一文中对分布式事务的几种实现方式进行了形象归纳

你每天上班，要经过一条10公里的只有两条车道的马路到达公司。这条路很堵，经常需要两三个小时上班时间没有保证，这是2PC的问题-慢。

选择一条很绕，长30公里但很少堵车的路，这是选b。上班时间有保证，但是必须早起，付出足够的间和汽油。这是柔性事务的问题，必须用具体业务来回滚，很难模块化

选择一条有点绕，长20公里的山路，路不平，只有suv可以走，这是事务消息最终一致性问题。引入新的消息中间件，需要额外的开发成本。但我司开发的CoolMQ已经对组件进行了封装，只需要发送接受，就能满足事务的要求。目前还有该方案的讲座，大家可以根据自己的需要选用。

最后是GTS，GTS修了一条拥有4条车道的高架桥，没有绕路，还是10公里。不堵车，对事务来说是性能；不绕路，对事务来说是简单易用，对业务无侵入，不用为事务而重构；没有车型限制，对事务说是没有功能限制，提供强一致事务。在没有高架桥的时代，高架桥出现对交通来说就是一个颠覆性新，很多以前看来无解的问题就迎刃而解了，同样的，GTS希望通过创新改变数据一致性处理的行业状。但遗憾的是并未开源，而且需要结合阿里云服务来使用。

123456

原文链接：分布式事务终极解决方案探讨