为什么说传统分布式事务不再适用于微服务架构?

创建时间： *2017/4/6 16:56*
来源： *http://www.36dsj.com/archives/74029*

为什么说传统分布式事务不再适用于微服务架构?

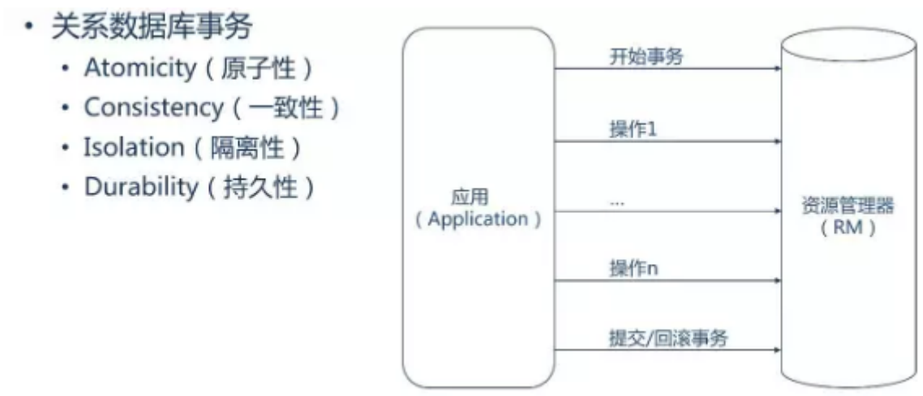
文 | 朱江

传统应用使用本地事务和分布式事务保证数据一致性，但是在微服务架构中数据都是服务私有的，需要通过服务提供的API来访问，所以分布式事务不再适用微服务架构。那么微服务架构又该如何保证数据一致性呢?本文就来谈谈这个话题。

1. 传统分布式事务不是微服务中数据一致性的最佳选择
2. 微服务架构中应满足数据最终一致性原则
3. 微服务架构实现最终一致性的三种模式
4. 对账是最后的终极防线

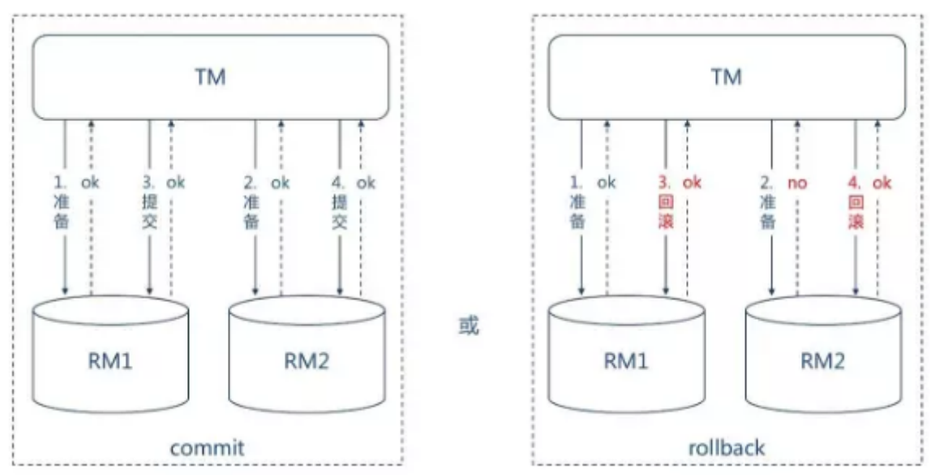
传统分布式事务

我们先来看下第一部分，传统使用本地事务和分布式事务保证一致性。

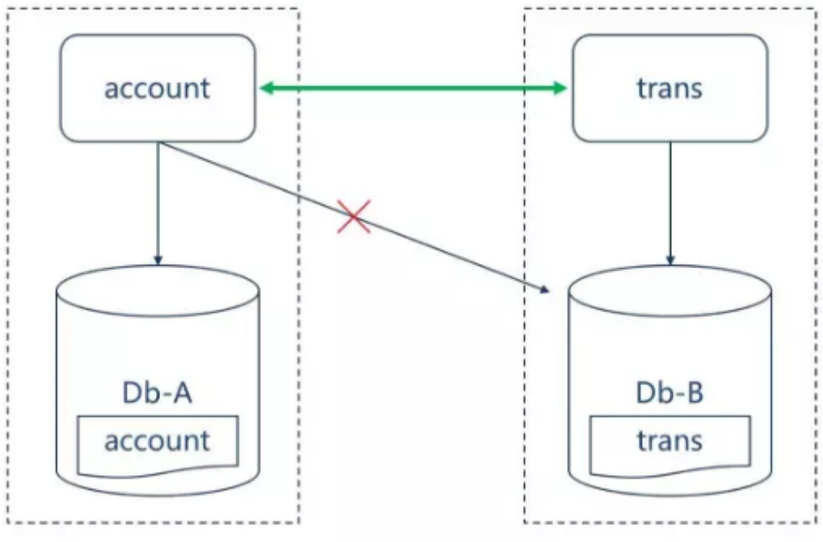


传统单机应用一般都会使用一个关系型数据库，好处是应用可以使用ACID。为保证一致性我们只需要：开始一个事务，改变(插入，删除，更新)很多行，然后提交事务(如果有异常时回滚事务)。更进一步，借助开发平台中的数据访问技术和框架(如 Spring)，我们需要做的事情更少，只需要关注数据本身的改变。

随着组织规模不断扩大，业务量不断增长，单机应用和数据库已经不足以支持庞大的业务量和数据量，这个时候需要对应用和数据库进行拆分，这就出现了一个应用需要同时访问两个或两个以上的数据库情况。开始我们用分布式事务来保证一致性，也就是我们常说的两阶段提交协议(2PC)。



本地事务和分布式事务现在已经非常成熟，相关介绍很丰富，此处不再讨论。我们下面来谈谈为什么分布式事务不适用于微服务架构。



首先，对于微服务架构来说，数据访问变得更加复杂，这是因为数据都是微服务私有的，唯一可访问的方式就是通过 API。这种打包数据访问方式使得微服务之间松耦合，并且彼此之间独立，更容易进行性能扩展。

其次，不同的微服务经常使用不同的数据库。应用会产生各种不同类型的数据，关系型数据库并不一定是最佳选择。

例如，某个产生和查询字符串的应用采用 Elasticsearch 的字符搜索引擎;某个产生社交图片数据的应用可以采用图数据库，例如Neo4j。

基于微服务的应用一般都使用 SQL 和 NoSQL 结合的模式。但是这些非关系型数据大多数并不支持 2PC。

可见在微服务架构中已经不能选择分布式事务了。

最终一致性原则

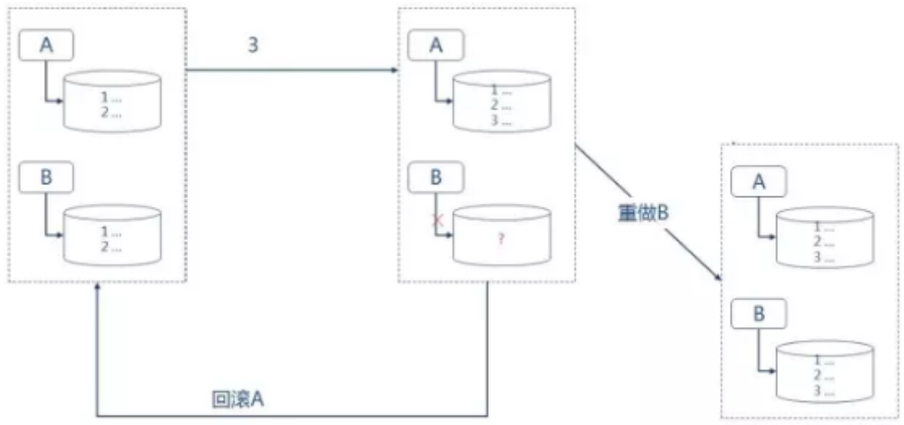
依据 CAP 理论，必须在可用性(availability)和一致性(consistency)之间做出选择。如果选择提供一致性需要付出在满足一致性之前阻塞其他并发访问的代价。这可能持续一个不确定的时间，尤其是在系统已经表现出高延迟时或者网络故障导致失去连接时。

依据目前的成功经验，可用性一般是更好的选择，但是在服务和数据库之间维护数据一致性是非常根本的需求，微服务架构中应选择满足最终一致性。

最终一致性是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。

当然选择了最终一致性，就要保证到最终的这段时间要在用户可接受的范围之内。那么我们怎么实现最终一致性呢?

从一致性的本质来看，是要保证在一个业务逻辑中包含的服务要么都成功，要么都失败。那我们怎么选择方向呢?保证成功还是保证失败呢?

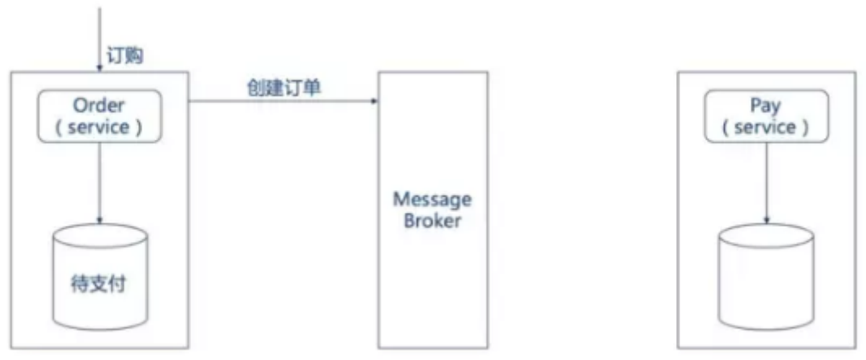


我们说业务模式决定了我们的选择。实现最终一致性有三种模式：可靠事件模式、业务补偿模式、TCC 模式。

可靠事件模式

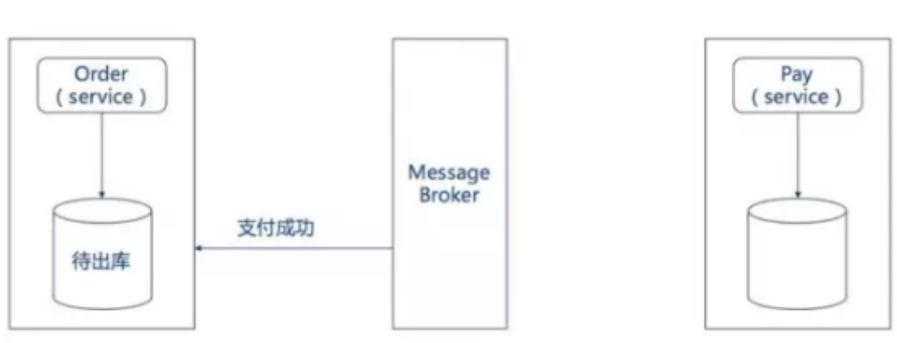
可靠事件模式属于事件驱动架构，当某件重要事情发生时，例如更新一个业务实体，微服务会向消息代理发布一个事件。消息代理会向订阅事件的微服务推送事件，当订阅这些事件的微服务接收此事件时，就可以完成自己的业务，也可能会引发更多的事件发布。

1. 如订单服务创建一个待支付的订单，发布一个“创建订单”的事件。

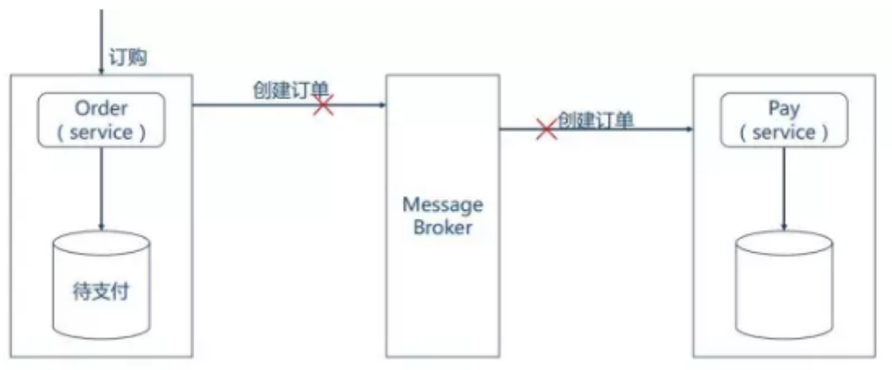


2. 支付服务消费“创建订单”事件，支付完成后发布一个“支付完成”事件。

3. 订单服务消费“支付完成”事件，订单状态更新为待出库。



从而就实现了完成的业务流程。但是这并不是一个完美的流程。



这个过程可能导致出现不一致的地方在于：某个微服务在更新了业务实体后发布事件却失败;虽然微服务发布事件成功，但是消息代理未能正确推送事件到订阅的微服务;接受事件的微服务重复消费了事件。

可靠事件模式在于保证可靠事件投递和避免重复消费，可靠事件投递定义为：

- **每个服务原子性的业务操作和发布事件。**
- **消息代理确保事件传递至少一次。**

避免重复消费要求服务实现幂等性，如支付服务不能因为重复收到事件而多次支付。

因为现在流行的消息队列都实现了事件的持久化和 at least once 的投递模式，『消息代理确保事件投递至少一次』已经满足，今天不做展开。

下面分享的内容主要从可靠事件投递和实现幂等性两方面来讨论，我们先来看可靠事件投递。

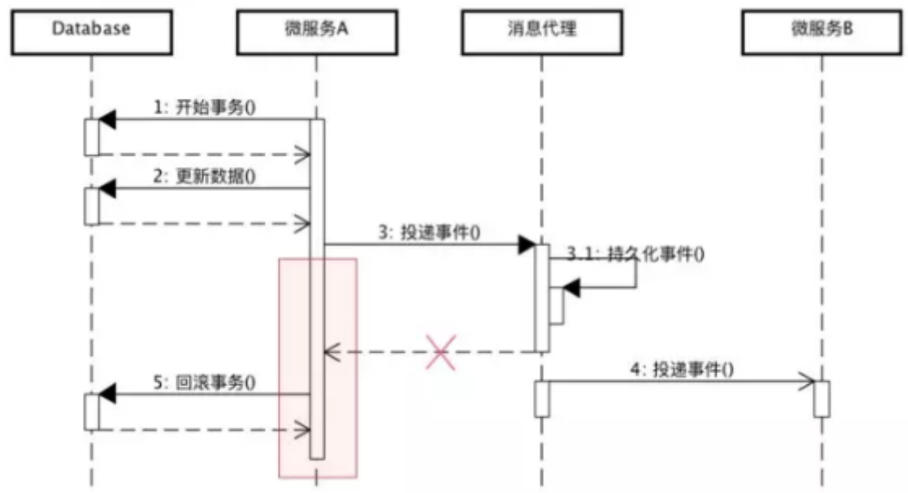
首先我们来看一个实现的代码片段，这是从某生产系统上截取下来的。

```
public void trans(){
  try{
     //1.操作数据库

      bool result = dao.update(model);//操作数据库失败，会抛出异常

     //2.如果第一步成功 则操作消息队列（投递消息）

      if(result){
         mq.append(model); //如果mq.append方法执行失败（投递消息失败），方法内部会抛出异常
      }

  }catch(Exception ex){
     rollback(); //如果发生异常 则回滚
  }
}
```

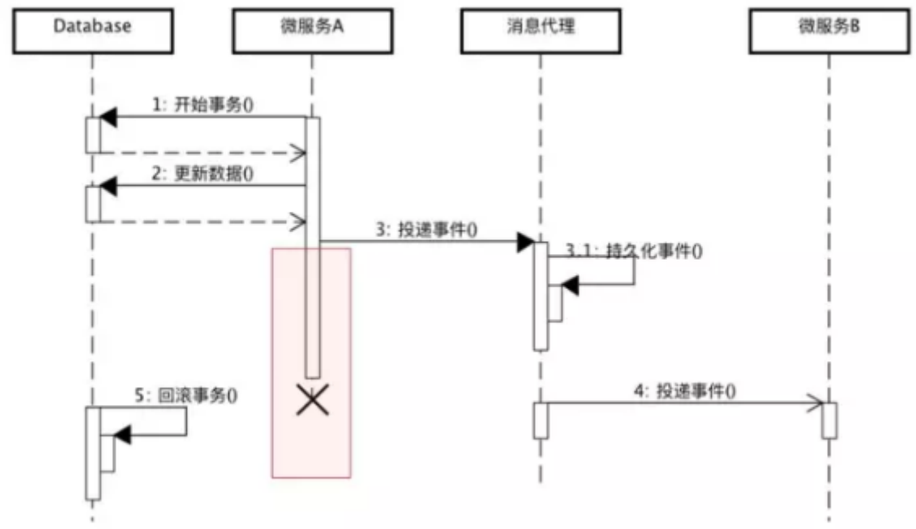
根据上述代码及注释，初看可能出现 3 种情况：

1. 操作数据库成功，向消息代理投递事件也成功。
2. 操作数据库失败，不会向消息代理中投递事件了。
3. 操作数据库成功，但是向消息代理中投递事件时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚。

从上面分析的几种情况来看，貌似没有问题。但是仔细分析不难发现缺陷所在，在上面的处理过程中存在一段隐患时间窗口。



微服务 A 投递事件的时候可能消息代理已经处理成功，但是返回响应的时候网络异常，导致 append 操作抛出异常。最终结果是事件被投递，数据库却被回滚。

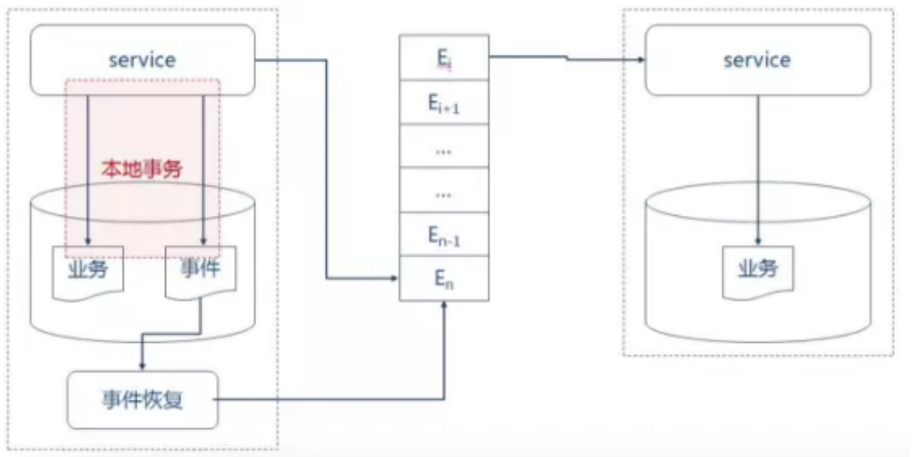


在投递完成后到数据库 commit 操作之间如果微服务 A 宕机也将造成数据库操作因为连接异常关闭而被回滚。最终结果还是事件被投递，数据库却被回滚。这个实现往往运行很长时间都没有出过问题，但是一旦出现了将会让人感觉莫名，很难发现问题所在。

下面给出两种可靠事件投递的实现方式。

1. 本地事件表

本地事件表方法将事件和业务数据保存在同一个数据库中，使用一个额外的“事件恢复”服务来恢复事件，由本地事务保证更新业务和发布事件的原子性。考虑到事件恢复可能会有一定的延时，服务在完成本地事务后可立即向消息代理发布一个事件。



微服务在同一个本地事务中记录业务数据和事件。

微服务实时发布一个事件立即通知关联的业务服务，如果事件发布成功立即删除记录的事件。

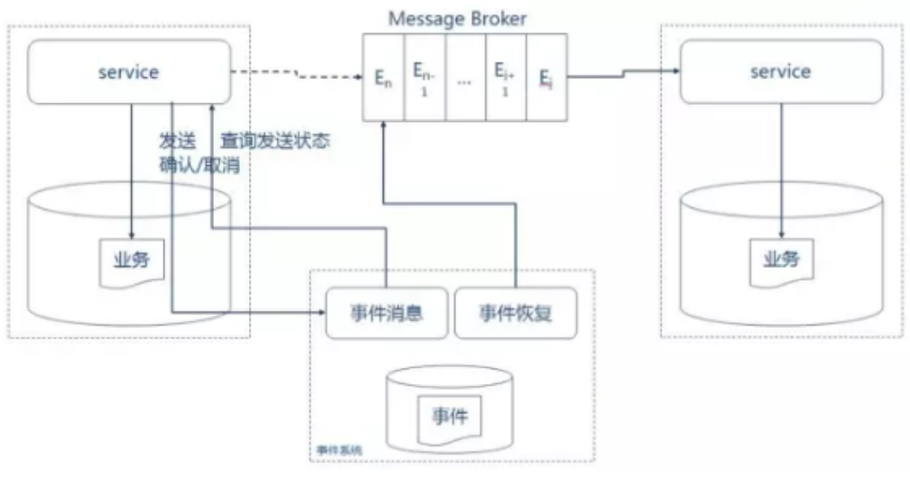
事件恢复服务定时从事件表中恢复未发布成功的事件，重新发布，重新发布成功才删除记录的事件。

其中第Ⅱ条的操作主要是为了增加发布事件的实时性，由第三条保证事件一定被发布。

本地事件表方式业务系统和事件系统耦合比较紧密，额外的事件数据库操作也会给数据库带来额外的压力，可能成为瓶颈。

2. 外部事件表

外部事件表方法将事件持久化到外部的事件系统，事件系统需提供实时事件服务以接受微服务发布事件，同时事件系统还需要提供事件恢复服务来确认和恢复事件。



1. 业务服务在事务提交前，通过实时事件服务向事件系统请求发送事件，事件系统只记录事件并不真正发送。
2. 业务服务在提交后，通过实时事件服务向事件系统确认发送，事件得到确认后事件系统才真正发布事件到消息代理。
3. 业务服务在业务回滚时，通过实时事件向事件系统取消事件。
4. 如果业务服务在发送确认或取消之前停止服务了怎么办呢?事件系统的事件恢复服务会定期找到未确认发送的事件向业务服务查询状态，根据业务服务返回的状态决定事件是要发布还是取消。

该方式将业务系统和事件系统独立解耦，都可以独立伸缩。但是这种方式需要一次额外的发送操作，并且需要发布者提供额外的查询接口。

介绍完了可靠事件投递再来说一说幂等性的实现，有些事件本身是幂等的，有些事件却不是。

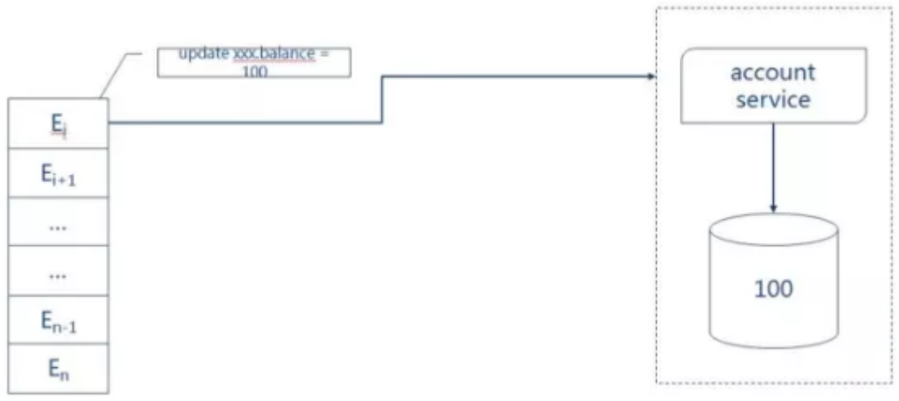
本身具有幂等性的事件需要考虑执行顺序

如果事件本身描述的是某个时间点的固定值(如账户余额为 100)，而不是描述一条转换指令(如余额增加 10)，那么这个事件是幂等的。

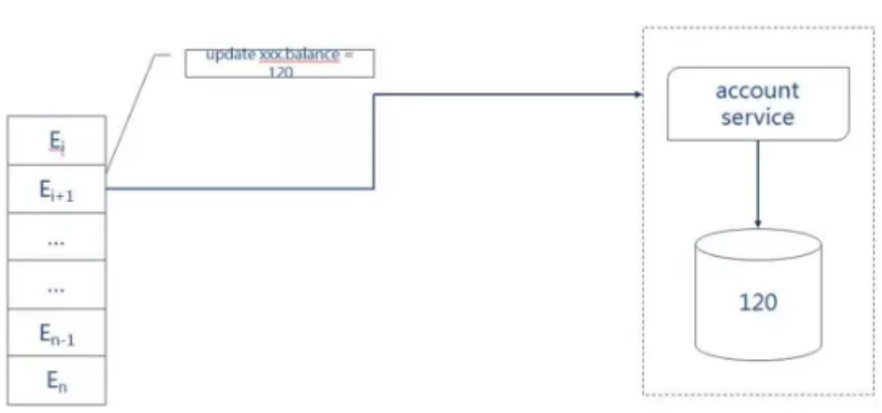
我们要意识到事件可能出现的次数和顺序是不可预测的，需要保证幂等事件的顺序执行，否则结果往往不是我们想要的。

如果我们先后收到两条事件，(1)账户余额更新为100，(2)账户余额更新为120。

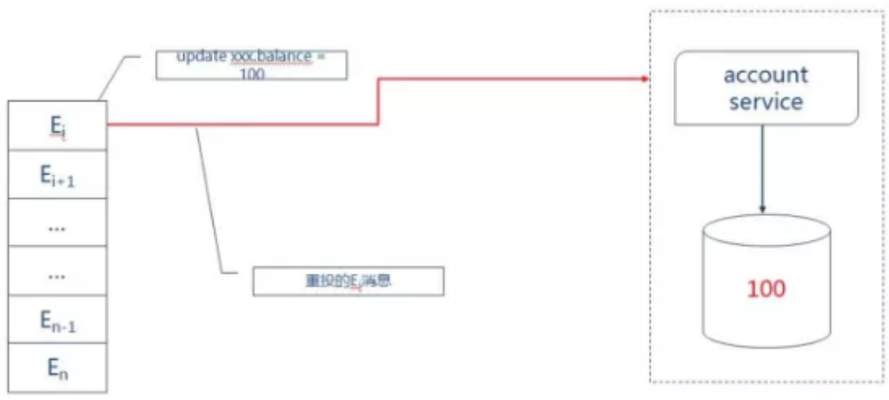
1. 微服务收到事件(1)



2. 微服务收到事件(2)



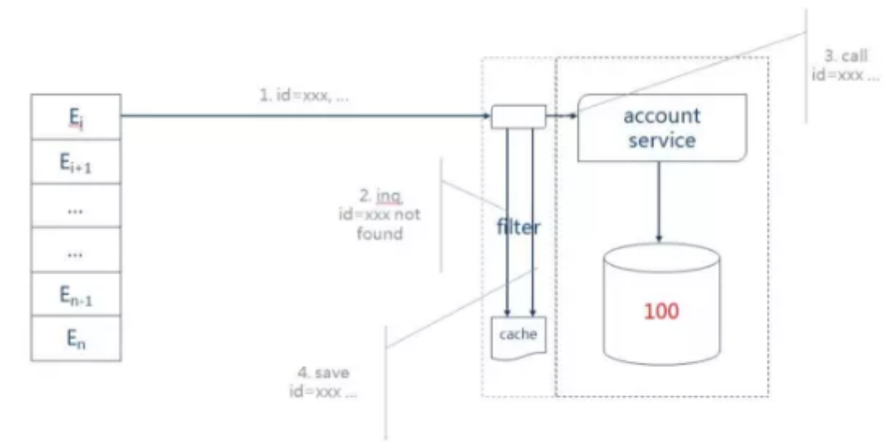
3. 微服务再次收到事件(1)

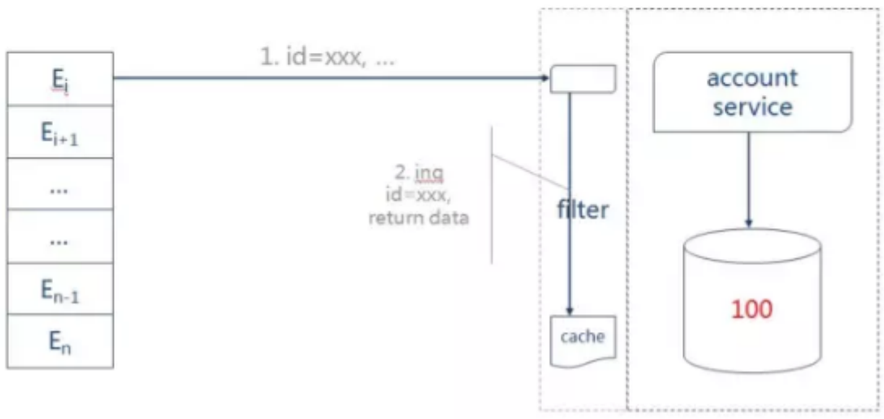


显然结果是错误的，所以我们需要保证事件(2)一旦执行事件(1)就不能再处理，否则账户余额仍不是我们想要的结果。

为保证事件的顺序一个简单的做法是在事件中添加时间戳，微服务记录每类型的事件最后处理的时间戳，如果收到的事件的时间戳早于我们记录的，丢弃该事件。如果事件不是在同一个服务器上发出的，那么服务器之间的时间同步是个难题，更稳妥的做法是使用一个全局递增序列号替换时间戳。

对于本身不具有幂等性的操作，主要思想是为每条事件存储执行结果，当收到一条事件时我们需要根据事件的 ID 查询该事件是否已经执行过，如果执行过直接返回上一次的执行结果，否则调度执行事件。





在这个思想下我们需要考虑重复执行一条事件和查询存储结果的开销。

重复处理开销小的事件

如果重复处理一条事件开销很小，或者可预见只有非常少的事件会被重复接收，可以选择重复处理一次事件，在将事件数据持久化时由数据库抛出唯一性约束异常。

重复处理开销大事件使用事件存储过滤重复事件

如果重复处理一条事件的开销相比额外一次查询的开销要高很多，使用一个过滤服务来过滤重复的事件，过滤服务使用事件存储存储已经处理过的事件和结果。

当收到一条事件时，过滤服务首先查询事件存储，确定该条事件是否已经被处理过，如果事件已经被处理过，直接返回存储的结果;否则调度业务服务执行处理，并将处理完的结果存储到事件存储中。

一般情况下上面的方法能够运行得很好，如果我们的微服务是 RPC 类的服务我们需要更加小心，可能出现的问题在于，(1)过滤服务在业务处理完成后才将事件结果存储到事件存储中，但是在业务处理完成前有可能就已经收到重复事件，由于是 RPC 服务也不能依赖数据库的唯一性约束;(2)业务服务的处理结果可能出现位置状态，一般出现在正常提交请求但是没有收到响应的时候。

对于问题(1)可以按步骤记录事件处理过程，比如事件的记录事件的处理过程为“接收”、“发送请求”、“收到应答”、“处理完成”。好处是过滤服务能及时的发现重复事件，进一步还能根据事件状态作不同的处理。

对于问题(2)可以通过一次额外的查询请求来确定事件的实际处理状态，要注意额外的查询会带来更长时间的延时，更进一步可能某些 RPC 服务根本不提供查询接口。此时只能选择接收暂时的不一致，时候采用对账和人工接入的方式来保证一致性。

补偿模式

为了描述方便，这里先定义两个概念：

1. 业务异常：业务逻辑产生错误的情况，比如账户余额不足、商品库存不足等。
2. 技术异常：非业务逻辑产生的异常，如网络连接异常、网络超时等。

补偿模式使用一个额外的协调服务来协调各个需要保证一致性的微服务，协调服务按顺序调用各个微服务，如果某个微服务调用异常(包括业务异常和技术异常)就取消之前所有已经调用成功的微服务。

补偿模式建议仅用于不能避免出现业务异常的情况，如果有可能应该优化业务模式，以避免要求补偿事务。如账户余额不足的业务异常可通过预先冻结金额的方式避免，商品库存不足可要求商家准备额外的库存等。

我们通过一个实例来说明补偿模式，一家旅行公司提供预订行程的业务，可以通过公司的网站提前预订飞机票、火车票、酒店等。

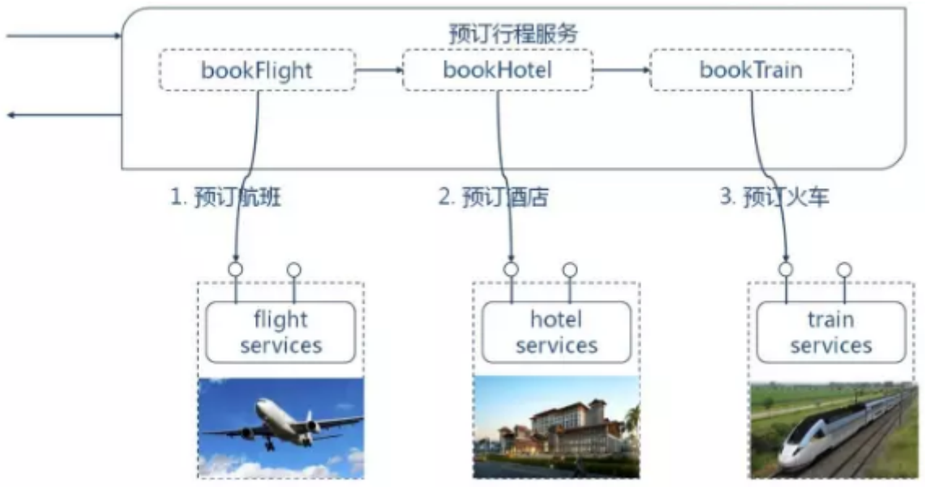
假设一位客户规划的行程是:

上海-北京6月19日9点的某某航班。

某某酒店住宿3晚。

北京-上海6月22日17点火车。

在客户提交行程后，旅行公司的预订行程业务按顺序串行的调用航班预订服务、酒店预订服务、火车预订服务。最后的火车预订服务成功后整个预订业务才算完成。



如果火车票预订服务没有调用成功，那么之前预订的航班、酒店都得取消。取消之前预订的酒店、航班即为补偿过程。

为了降低开发的复杂性和提高效率，协调服务实现为一个通用的补偿框架。补偿框架提供服务编排和自动完成补偿的能力。

要实现补偿过程，我们需要做到两点：

首先要确定失败的步骤和状态，从而确定需要补偿的范围。

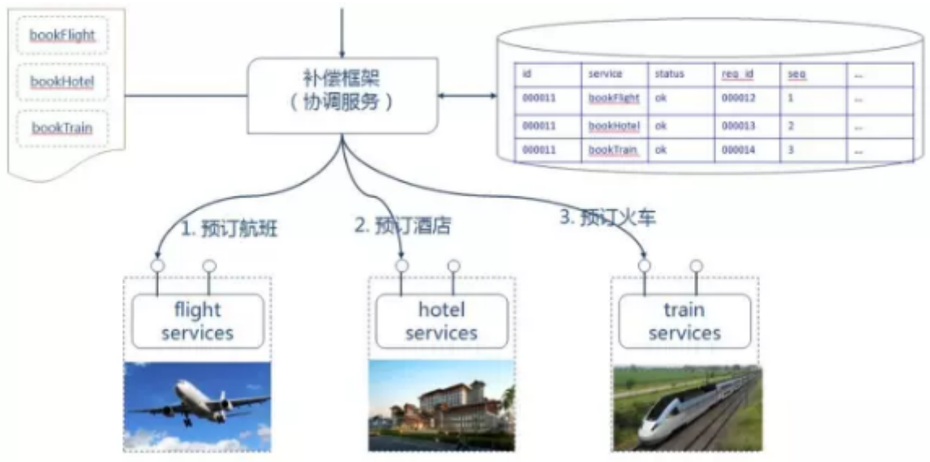
在上面的例子中我们不仅要知道第 3 个步骤(预订火车)失败，还要知道失败的原因。如果是因为预订火车服务返回无票，那么补偿过程只需要取消前两个步骤就可以了;但是如果失败的原因是因为网络超时，那么补偿过程除前两个步骤之外还需要包括第 3 个步骤。

其次要能提供补偿操作使用到的业务数据。

比如一个支付微服务的补偿操作要求参数包括支付时的业务流水 id、账号和金额。理论上说实际完成补偿操作可以根据唯一的业务流水 id 就可以，但是提供更多的要素有益于微服务的健壮性，微服务在收到补偿操作的时候可以做业务的检查，比如检查账户是否相等，金额是否一致等等。

实现补偿模式的关键在于业务流水的记录

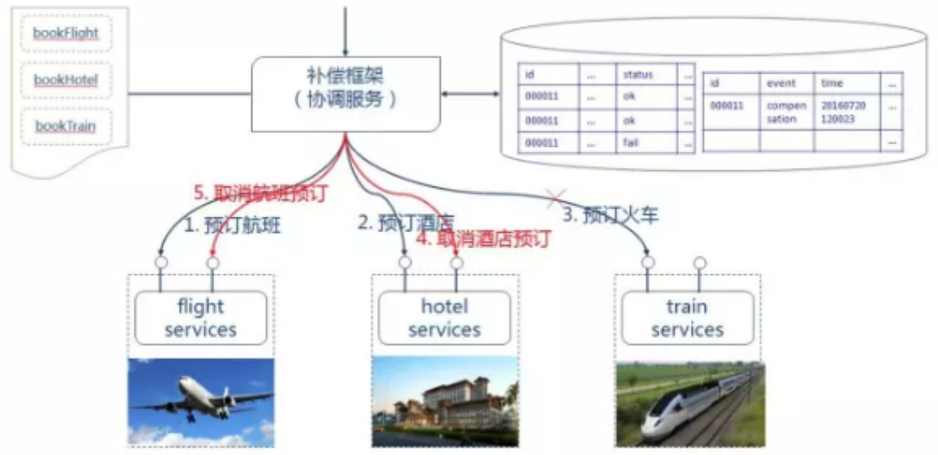
做到上面两点的办法是记录完整的业务流水，可以通过业务流水的状态来确定需要补偿的步骤，同时业务流水为补偿操作提供需要的业务数据。



当客户的一个预订请求达到时，协调服务(补偿框架)为请求生成一个全局唯一的业务流水号，并在调用各个工作服务的同时记录完整的状态。

1. 记录调用 bookFlight 的业务流水，调用 bookFlight 服务，更新业务流水状态。
2. 记录调用 bookHotel 的业务流水，调用 bookHotel 服务，更新业务流水状态。
3. 记录调用 bookTrain 的业务流水，调用 bookTrain 服务，更新业务流水状态。

当调用某个服务出现异常时，比如第 3 步骤(预订火车)异常。



协调服务(补偿框架)同样会记录第 3 步的状态，同时会另外记录一条事件，说明业务出现了异常。然后就是执行补偿过程了，可以从业务流水的状态中知道补偿的范围，补偿过程中需要的业务数据从记录的业务流水中获取。

对于一个通用的补偿框架来说，预先知道微服务需要记录的业务要素是不可能的。那么就需要一种方法来保证业务流水的可扩展性，这里介绍两种方法：大表和关联表。

trace

ID	service	seq	status	col1	col2	col3	...
1	bookFlight	1	send	2	10000	xxxx	

大表

关联表

platform_trace

ID	...	b_tbname	b_keys	b_keys_v
1		bank_trace	id	2

bank_trace

id	amount	card	time	...
2	10000	xxxx	xxxx	

大表顾明思议就是设计时除必须的字段外，还需要预留大量的备用字段，框架可以提供辅助工具来帮助将业务数据映射到备用字段中。

关联表，分为框架表和业务表，技术表中保存为实现补偿操作所需要的技术数据，业务表保存业务数据，通过在技术表中增加业务表名和业务表主键来建立和业务数据的关联。

大表对于框架层实现起来简单，但是也有一些难点，比如预留多少字段合适，每个字段又需要预留多少长度。另外一个难点是如果向从数据层面来查询数据，很难看出备用字段的业务含义，维护过程不友好。

关联表在业务要素上更灵活，能支持不同的业务类型记录不同的业务要素;但是对于框架实现上难度更高，另外每次查询都需要复杂的关联动作，性能方面会受影响。

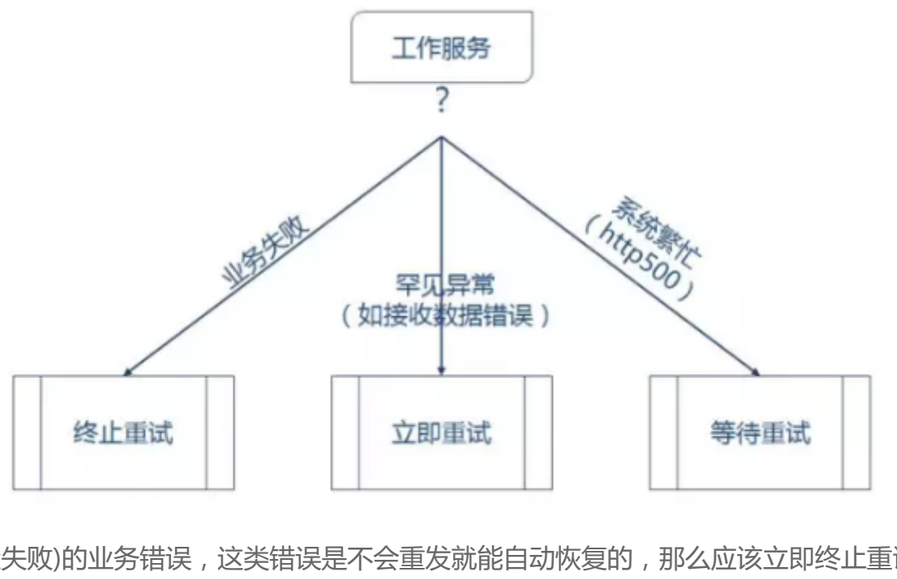
有了上面的完整的流水记录，协调服务就可以根据工作服务的状态在异常时完成补偿过程。但是补偿由于网络等原因，补偿操作并不一定能保证 100%成功，这时候我们还要做更多一点。

通过重试保证补偿过程的完整，从而满足最终一致性

补偿过程作为一个服务调用过程同样存在调用不成功的情况，这个时候需要通过重试的机制来保证补偿的成功率。当然这也就要求补偿操作本身具备幂等性。

关于幂等性的实现在前面做过讨论。

如果只是一味的失败就立即重试会给工作服务造成不必要的压力，我们要根据服务执行失败的原因来选择不同的重试策略。



1) 如果失败的原因不是暂时性的，由于业务因素导致(如业务要素检查失败)的业务错误，这类错误是不会重发就能自动恢复的，那么应该立即终止重试。

2) 如果错误的原因是一些罕见的异常，比如因为网络传输过程出现数据丢失或者错误，应该立即再次重试，因为类似的错误一般很少会再次发生。

3) 如果错误的原因是系统繁忙(比如 HTTP 协议返回的 500 或者另外约定的返回码)或者超时，这个时候需要等待一些时间再重试。

重试操作一般会指定重试次数上线，如果重试次数达到了上限就不再进行重试了。这个时候应该通过一种手段通知相关人员进行处理。

对于等待重试的策略如果重试时仍然错误，可逐渐增加等待的时间，直到达到一个上限后，以上限作为等待时间。

如果某个时刻聚集了大量需要重试的操作，补偿框架需要控制请求的流量，以防止对工作服务造成过大的压力。

另外关于补偿模式还有几点补充说明。

微服务实现补偿操作不是简单的回退到业务发生时的状态，因为可能还有其他的并发的请求同时更改了状态。一般都使用逆操作的方式完成补偿。

补偿过程不需要严格按照与业务发生的相反顺序执行，可以依据工作服务的重用程度优先执行，甚至是可以并发的执行。

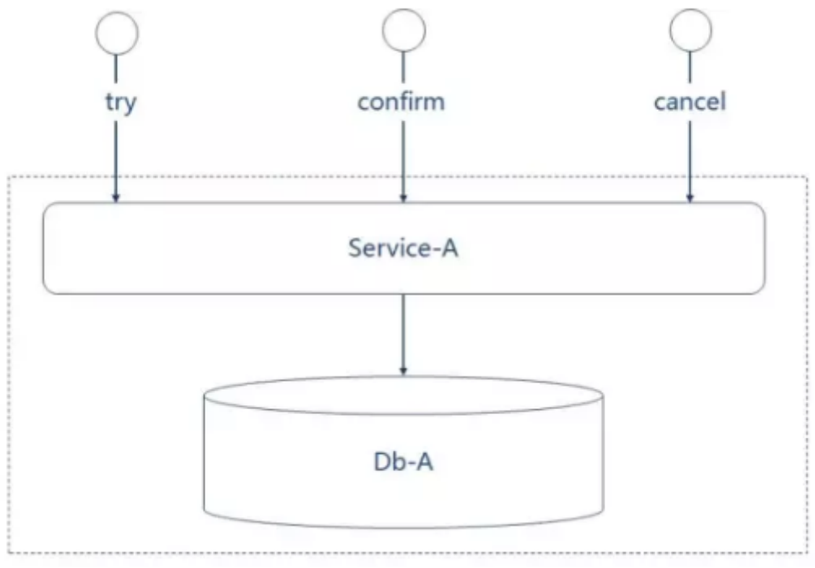
有些服务的补偿过程是有依赖关系的，被依赖服务的补偿操作没有成功就要及时终止补偿过程。

如果在一个业务中包含的工作服务不是都提供了补偿操作，那我们编排服务时应该把提供补偿操作的服务放在前面，这样当后面的工作服务错误时还有机会补偿。

设计工作服务的补偿接口时应该以协调服务请求的业务要素作为条件，不要以工作服务的应答要素作为条件。因为还存在超时需要补偿的情况，这时补偿框架就没法提供补偿需要的业务要素。

TCC模式(Try-Confirm-Cancel)

一个完整的 TCC 业务由一个主业务服务和若干个从业务服务组成，主业务服务发起并完成整个业务活动，TCC 模式要求从服务提供三个接口：Try、Confirm、Cancel。



1. Try

完成所有业务检查

预留必须业务资源

2. Confirm

真正执行业务

不作任何业务检查

只使用 Try 阶段预留的业务资源

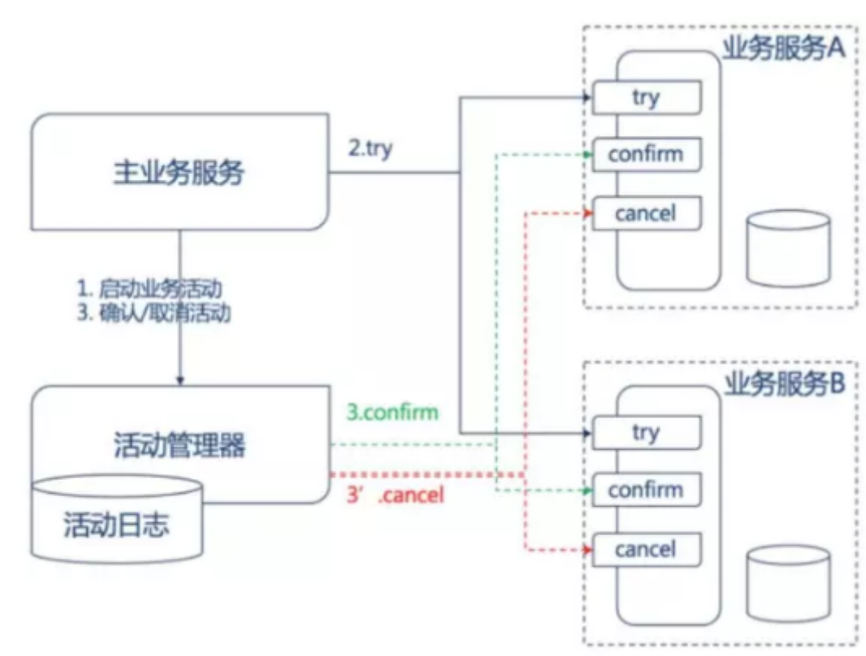
Confirm 操作满足幂等性

3. Cancel：

释放 Try 阶段预留的业务资源

Cancel 操作满足幂等性

整个 TCC 业务分成两个阶段完成。



第一阶段：主业务服务分别调用所有从业务的 try 操作，并在活动管理器中登记所有从业务服务。当所有从业务服务的 try 操作都调用成功或者某个从业务服务的 try 操作失败，进入第二阶段。

第二阶段：活动管理器根据第一阶段的执行结果来执行 confirm 或 cancel 操作。

如果第一阶段所有 try 操作都成功，则活动管理器调用所有从业务活动的 confirm操作。否则调用所有从业务服务的 cancel 操作。

需要注意的是第二阶段 confirm 或 cancel 操作本身也是满足最终一致性的过程，在调用 confirm 或 cancel 的时候也可能因为某种原因(比如网络)导致调用失败，所以需要活动管理支持重试的能力，同时这也就要求 confirm 和 cancel 操作具有幂等性。

在补偿模式中一个比较明显的缺陷是，没有隔离性。从第一个工作服务步骤开始一直到所有工作服务完成(或者补偿过程完成)，不一致是对其他服务可见的。另外最终一致性的保证还充分的依赖了协调服务的健壮性，如果协调服务异常，就没法达到一致性。

TCC模式在一定程度上弥补了上述的缺陷，在TCC模式中直到明确的confirm动作，所有的业务操作都是隔离的(由业务层面保证)。另外工作服务可以通过指定 try 操作的超时时间，主动的 cancel 预留的业务资源，从而实现自治的微服务。

TCC模式和补偿模式一样需要需要有协调服务和工作服务，协调服务也可以作为通用服务一般实现为框架。与补偿模式不同的是 TCC 服务框架不需要记录详细的业务流水，完成 confirm 和 cancel 操作的业务要素由业务服务提供。

在第4步确认预订之前，订单只是pending状态，只有等到明确的confirm之后订单才生效。

如果3个服务中某个服务try操作失败，那么可以向TCC服务框架提交cancel，或者什么也不做由工作服务自己超时处理。

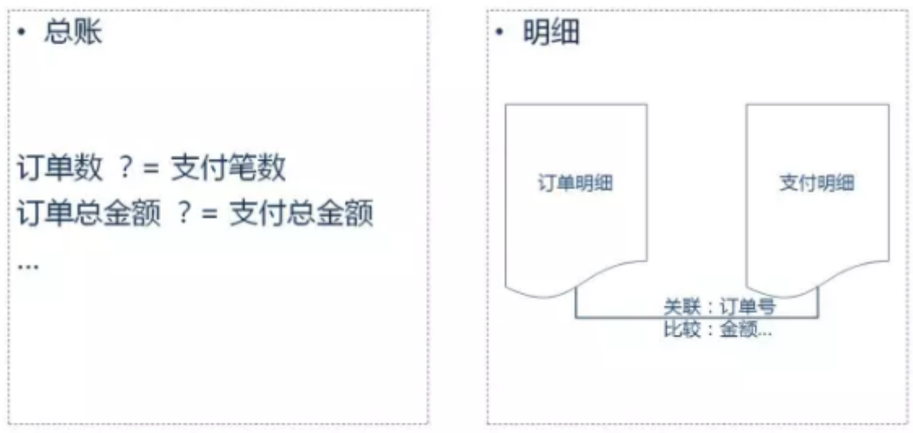
TCC 模式也不能百分百保证一致性，如果业务服务向 TCC 服务框架提交 confirm后，TCC 服务框架向某个工作服务提交 confirm 失败(比如网络故障)，那么就会出现不一致，一般称为 heuristic exception。

需要说明的是为保证业务成功率，业务服务向 TCC 服务框架提交 confirm 以及TCC 服务框架向工作服务提交 confirm/cancel 时都要支持重试，这也就要confirm/cancel 的实现必须具有幂等性。如果业务服务向 TCC 服务框架提交confirm/cancel 失败，不会导致不一致，因为服务最后都会超时而取消。

另外 heuristic exception 是不可杜绝的，但是可以通过设置合适的超时时间，以及重试频率和监控措施使得出现这个异常的可能性降低到很小。如果出现了heuristic exception 是可以通过人工的手段补救的。

对账是最后的终极防线

如果有些业务由于瞬时的网络故障或调用超时等问题，通过上文所讲的 3 种模式一般都能得到很好的解决。但是在当今云计算环境下，很多服务是依赖于外部系统的可用性情况，在一些重要的业务场景下还需要周期性的对账来保证真实的一致性。比如支付系统和银行之间每天日终是都会有对账过程。



作者介绍

朱江，普元解决方案中心总经理。2003 年毕业于河海大学，2007 年加入普元信息技术股份有限公司，长期致力于金融软件平台建设实践。主持完成多个金融行业平台产品及解决方案的研发工作，包括业务集成平台、集中监控平台、统一渠道接入平台等。对金融行业技术架构有较深刻理解，曾主持中国工商银行、建设银行、中信银行、国开银行等多家大型企业的技术平台规划与落地，目前负责普元解决方案中心，负责基于普元标准产品打造行业解决方案。

36大数据(www.36dsj.com)成立于2013年5月，是中国访问量最大的大数据网站。36大数据(微信号:dashuju36)以独立第三方的角度，为大数据产业生态图谱上的需求商 、应用商、服务商、技术解决商等相关公司及从业人员提供全球资讯、商机、案例、技术教程、项目对接、创业投资及专访报道等服务。

End.

转载请注明来自36大数据（36dsj.com)：36大数据 » 为什么说传统分布式事务不再适用于微服务架构?