本文是分布式事务系列文章的第一篇。主线源自eBay架构师 Dan Pritchett 2008 年发表在 ACM 的文章，Base: An Acid Alternative。

对于很多业务系统来说，整个系统其实是由多个独立的系统构成的。这些独立的系统由各自的研发小组进行研发和维护，数据往往也存储在各自独立的数据库中。

我们以下单流程为例。下单往往涉及到订单系统、库存系统、优惠券管理系统、支付系统、物流系统、用户系统等。我们希望下单操作对于这些系统的更新，要么全部成功提交；如果有些步骤失败了，那么所有的改动都能够回滚到最初状态。这个就是我们今天要讨论的分布式事务问题。

问题的产生

对于web应用来说，当访问量日益增大，我们就不得不考虑扩容。扩容主要有两个方向，垂直扩容和水平扩容。

垂直扩容，也就是买性能更强大的服务器，更大的存储。垂直扩容很容易遇到瓶颈，所以几乎没有公司采用这种策略。

水平扩容则能够提供更多的灵活性，但同时也会更复杂，对于数据存储来说，水平扩容包含两个维度。

1. 第一个维度是使用不同的数据库来服务不同的业务数据。比如用户数据用一套数据库，产品数据用一套数据库。这种分库方式能完成业务数据的隔离，各业务数据库独立提供服务，相互之间不会有影响。
2. 第二个维度则是将原本属于同一个业务的数据按片存储在不同的数据库上，也就是我们经常提到的sharding（分片）。分片的方式能帮助我们完成最基本的数据库水平扩展。将原本属于单张表的数据分散到多张表，由多个数据库提供服务，增加了并发能力。

水平扩容后，数据分散在不同的数据库中。当我们希望对这些数据的更新操作以事务的方式来执行时，就产生了分布式事务问题。

CAP理论

CAP理论是由Eric Brewer提出来的。

- Consistency。一致性。写操作后的读操作，必须返回最新写入的值。对于单个节点的系统来说，这点很容易满足。但对于多节点的集群环境来说，如果写操作往节点1写入，而读操作去节点2读取，就无法满足这个要求。
- Availability。可用性。所有的用户请求都能得到响应。
- Partition tolerance。分区容错性。即使在某些节点无法响应的情况下，用户操作能让能正确执行。

按照CAP理论，任何系统只能同时满足上面三个特性中的两个。对于水平扩容机制来说，多存储节点的引入已经设定了分区的大前提。所以我们的系统只能在一致性和可用性之间做折衷性原则。

ACID方案

ACID数据库事务极大简化了研发人员的工作。它提供了下面的保证：

- Atomicity。事务中的所有操作要么都成功，要么都失败。
- Consistency。事务开始前和结束后的数据库状态一致。比如用户A给用户B转账100元，转账前后A账户和B账户总和不变。
- Isolation。事务之间相互隔离。事务执行时，仿佛当前数据库只有它一个事务在执行。
- Durability。事务结束后，操作产生的结果不会被回滚。

为了将ACID事务从单一数据库扩展到多数据库。很多数据库供应商引入了2PC机制。2PC分为如下两个阶段：

- 首先，事务协调器会向所有参与当前事务的数据库节点进行问询，看它们是否准备好提交当前事务。如果所有的数据库都准备好了，则进入下一个阶段。

- 事务协调器让所有数据库提交当前事务。

如果任何一个数据库节点没能成功提交，则所有已经成功提交的数据库节点都需要回滚本次操作。

2PC有什么缺点呢？如果CAP理论时对的，我们在得到一致性的同时，肯定会牺牲可用性。

一个系统的可用性等于所有子系统可用性的乘积。对于2PC来说，假设有两个数据库参与这个2PC事务，每个数据库的可用性是99.9%，那么整个系统的可用性就变成99.8%了。

BASE，ACID的替代方案

BASE代表的是Basically Available，Soft state，Eventually consistent。

BASE和ACID正好相反。ACID强制每个操作后都要满足一致性，BASE则允许出现暂时的不一致。对于一致性的妥协让BASE能够提供比ACID更为强大的扩展性。

那我们如何在具体的项目中来使用BASE呢？

一致性模式

BASE要求我们不要一直绷着一致性那根弦。我们需要去寻找那些可以适当放松一致性警惕的机会。

这些机会往往需要工程师和业务专家一起来寻找，而且我们往往会听到一些反对的声音。比如，一致性对于我们的业务来说至关重要，我们无法向用户隐藏我们的数据不一致等等。下面这个例子可能会给大家一些启发。

假设我们有两张表，user和transaction。user表存储了总的销售量和总的购买量，这些都是统计数据，由transaction表的单笔交易数据统计得来。transaction表则包含了单笔交易。

user表：

- *id*
- *name*
- *amt_sold，总销售量，由transaction的amount统计得来*
- *amt_bought，总购买量，由transaction的amount统计得来*

transaction表：

- *xid*
- *seller_id，卖家id*
- *buyer_id，买家id*
- *amount，数量*

如果我们使用ACID来确保user表和transaction表的一致性，会比较简单：

```
begin transaction
 insert into transaction(xid, seller_id, buyer_id, amount);
 //累加卖家的销售量
 update user set amt_sold = amt_sold + $amount where id = $seller_id;
 //累加买家的购买量
 update user set amt_bought = amt_bought + $amount where id = $buyer_id;
end transaction
```

user表里的总销售量和总购买量，可以看作transaction表的一个缓存。每次查询时，可以直接从user表读取，而不用去transaction统计。基于这样一个前提，我们认为user表里的总销量、总购买量和transaction表里的数据之间的一致性要求可以做一些妥协。

如何修改上面的SQL语句来反映这样一种妥协，依赖于我们如何定义这种统计关系。如果这种统计关系只是一个估计数据，也就是说即使没能准确统计到某些交

易也能接受，那我们很容易就能把上面的SQL改成下面的形式。

```
begin transaction
 insert into transaction(xid, seller_id, buyer_id, amount);
end transaction

begin transaction
 update user set amt_sold = amt_sold + $amount where id = $seller_id;
 update user set amt_bought = amount_bought + $amount where id = $buyer_id;
end transaction
```

这样一种改动可能会导致user表数据和transaction表数据出现永久性的不一致。当其中一个事务出错而另一个成功提交时就会发生这种情况。

如果我们不希望user表的数据只是一个估算数据呢？我们希望user表的数据能够和transaction表的数据一致。

我们可以引入一个消息队列来解决这个问题。实现这个消息队列会有很多种方式，但其中最重要的一点时，要让消息的入队和transaction的写操作能通过一个本地事务而不是2pc来处理，所以最简单的方式就是用一个消息表来存这些消息，这个消息表和transaction在同一个数据库。

```
begin transaction
 insert into transaction(xid, seller_id, buyer_id, amount);
 //入队卖家数据更新消息，第一个字段是balance，第二个字段是id
 queue message "update user("seller", seller_id, amount)";
 //入队买家数据更新消息，第一个字段是balance，第二个字段是id
 queue message "update user("buyer", buyer_id, amount)";
end transaction

for each message in Queue
  begin transaction
  dequeue message; //操作1
  if message.balance == "seller"
  update user set amt_sold = amt_sold + messgae.amount where id = message.id; //操作2
```

```
14  else
15  update user set amt_bought = amt_bought + message.amount where id = mes
sage.id; //操作2
16  end if
17  end transaction
```

上面的解决方案仍然会有一个问题，虽然我们让消息的入队和transaction的写操作可以在一个事务中。但消息的出队（操作1）和user表的更新操作（操作2）则仍然会面临2pc问题，因为user表在另外一个数据库。（transaction和user表因为水平扩展分属不同的数据库）

在介绍具体解决方案之前，我们介绍下幂等性。对于一个操作，如果执行一次和执行多次的结果一样，我们就说这个操作具有幂等性。幂等性操作的一个优点就是它们允许部分失败，对于这种情况，重复执行该操作并不会改变系统的最终状态。

对于上面这个例子，从幂等性角度来看是有问题的。上面的更新操作，每次会给总销售量和总购买量增加一笔交易数据，如果重复执行这个操作，显然会得到错误的结果。

要解决这个问题，关键点是要能够记录哪些交易记录已经反映到user表了，哪些还没有。

这里我们引入一张updates_applied表来跟踪这个更新操作。

update_applied表：

- *trans_id, 对应于transaction.xid*
- *balance, seller or buyer*
- *user_id, 对应于user.id*

然后我们重写上面的逻辑：

```
1 begin transaction
2  insert into transaction(xid, seller_id, buyer_id, amount);
```

```
  //入队卖家数据更新消息，第一个字段是balance，第二个字段是id
  queue message "update user("seller", seller_id, amount)";
  //入队买家数据更新消息，第一个字段是balance，第二个字段是id
  queue message "update user("buyer", buyer_id, amount)";
end transaction

for each message in queue
  peek message
  begin transaction
  //如果当前消息在update_applied表中有记录，说明已经处理过了，否则没有处理过
  select count(*) from update_applied as processed where trans_id = message.trans_id
  and balance = message.balance and user_id = message.user_id;
  if proccessed == 0
  if message.balance == "seller"
  update user set amt_sold = amt_sold + messgae.amount where id = message.id; //操作2
  else
  update user set amt_bought = amt_bought + message.amount where id = message.id; //操作2
  end if
  //插入记录，表示这条消息处理过了
  insert into updates_applied(message.trans_id, message.balance, message.id);
  end if
  end transaction
  if transaction successful
  //如果上面的事务成功处理，则把消息从队列删除。这个操作执行失败也没关系，上面的事务已经变成幂等操作了
  remove message from queue; 。
  end if
end for
```

上面的peek message操作需要说明下，如果这个队列的实现方案是用和transaction表一样的数据库表，那这个peek操作就是一个数据库读操作，最后的出队操作就是一个数据库删除操作。而如果队列的解决方案是其它方式，则需要这种解决方案支持peek操作，也就是可以查看数据，但不会把数据从队列移除。

上面的方案已经解决了user数据库和transaction数据库之间的一致性问题。但在某些情况下，上面的方案还会存在一些问题。

对于user表，假如我们不光是要记录总的销售量和总的购买量，我们还想记录最后一次购买或者销售的发生时间。

我们给user表增加两个字段。

user表：

- *id*
- *name*
- *amt_sold，总销售量*
- *amt_bought，总购买量*
- *last_sale，最后一次销售时间*
- *last_purchage，最后一次购买时间*

transaction表：

- *xid*
- *seller_id，卖家id*
- *buyer_id，买家id*
- *amount，数量*

对于这种情况，上面的解决方案就会存在一个问题，那就是oder的处理顺序不同可能会导致user表的更新操作不再幂等。对于两笔发生时间非常接近的交易，在上面的消息查询操作不能保证消息顺序的情况下，消息可能会乱序执行。那么last_sale和last_purchage的值就可能会错乱。

有两种方案可以处理这个问题：

1. 限制xid递增，处理的时候递增进行处理。
2. 更新user表时，限制transaction的时间必须要大于user的最后一次时间，才能执行更新操作。

采用第二种方案对上述逻辑修改如下：

```
begin transaction
 insert into transaction(xid, seller_id, buyer_id, amount);
 //入队卖家数据更新消息，第一个字段是balance
 queue message "update user("seller", seller_id, amount)";
 //入队买家数据更新消息
 queue message "update user("buyer", buyer_id, amount)";
end transaction
for each message in queue
 peek message
  begin transaction
  //如果当前消息在update_applied表中有记录，说明已经处理过了，否则没有处理过
  select count(*) from update_applied as processed where trans_id = messa
ge.trans_id
  and balance = message.balance and user_id = message.user_id;
  if proccessed == 0
  if message.balance == "seller"
  //增加了last_purchase和trans_date的比较
  update user set amt_sold = amt_sold + messgae.amount, last_purchase = m
essage.trans_date
  where id = message.seller_id and last_purchase < message.trans_date
  else
  //增加了last_sale和trans_date的比较
  update user set amt_bought = amt_bought + message.amount, last_sale = m
essage.trans_date
  where id = message.buyer_id and last_sale < message.trans_date
  end if
  //插入记录，表示这条消息处理过了
  insert into updates_applied(message.trans_id, message.balance,
message.user_id);
  end if
  end transaction
  if transaction successful
  //如果上面的事务成功处理，则把消息从队列删除。如果这个操作失败也没关系，上面的
事务已经变成幂等操作了
  remove message from queue; 。
  end if
 end for
```

当然，这种方案也可能会有一个小问题，就是如果后发生的交易先被处理，那先前的交易就可能会被漏掉，而不会体现在user表中了。

柔性事务/最终一致性

截至目前，我们的讨论主要还是聚焦在如何牺牲一定的一致性来换取可用性。而另一方面，我们需要理解柔性事务和最终一致性对于系统设计的影响。

作为软件工程师，我们习惯于将我们自己的系统看作一个闭环。我们会考虑这个闭环系统的可预测性，亦即可预测的输入产生可预测的输出。这一点对于构建正确的软件系统时很必要的。好消息是，BASE并不会改变软件的可预测性。

比如，考虑一个转账系统，用户可以把一笔钱转给另一个用户。而且基于我们之前的讨论，付款方和收款方可能是不同的银行，所以我们没办法用ACID来处理这个事务。

可能会存在一个时间窗口，钱已经转出去了，但没有到账。在这个时间窗口内，两个用户都不持有这笔钱。

但从用户的角度来考虑，这个延迟是可以接受并且对用户可能是不可见的，除非付款方和收款方做实时沟通。否则他们挺难感知到付款时间和收款时间之间的时差。

事件驱动

假如我们一定要知道什么时候状态变成最终一致了呢？比如我们希望在到账时，给用户发送一条到账短信。一个比较简单的方式就是采用event driven。

对于上面的例子，我们只需要在收款方账户被更新的事务里，同时创建一条event，比如插入event表，然后就可以用这个条event来驱动后续的短信发送流程了。

今天关于BASE的介绍就到这里，希望能给大家带来一些帮助。

参考：https://zhuanlan.zhihu.com/p/95608046