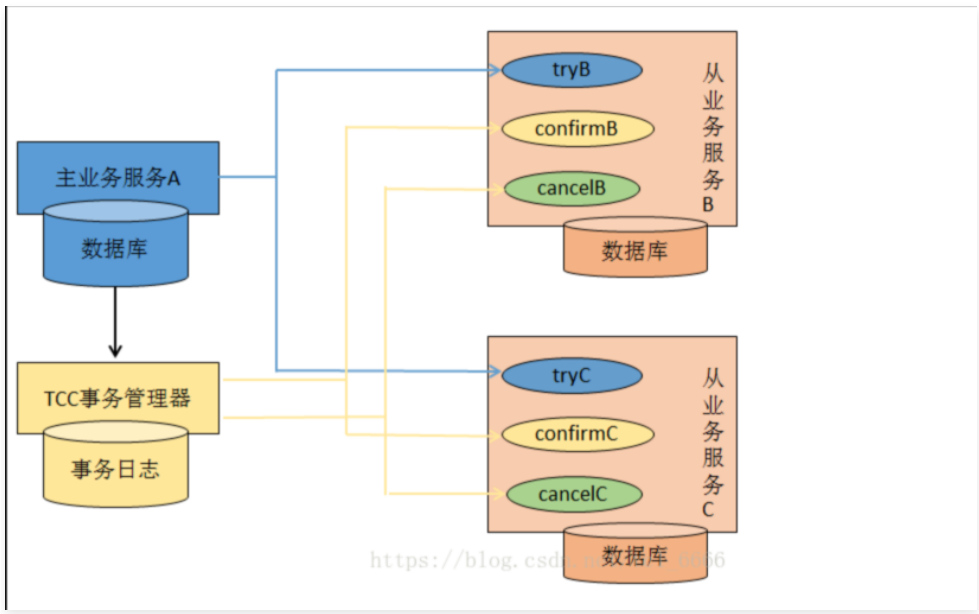
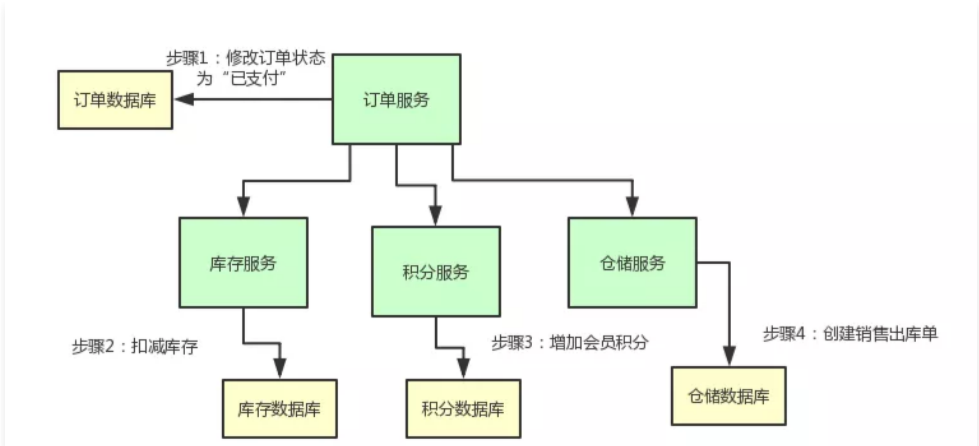
bobby · 更新于 2022-11-16

上一节 4. 两_三阶段提交 6. 基于本地消息... 下一节



一个订单支付之后，我们需要做下面的步骤：

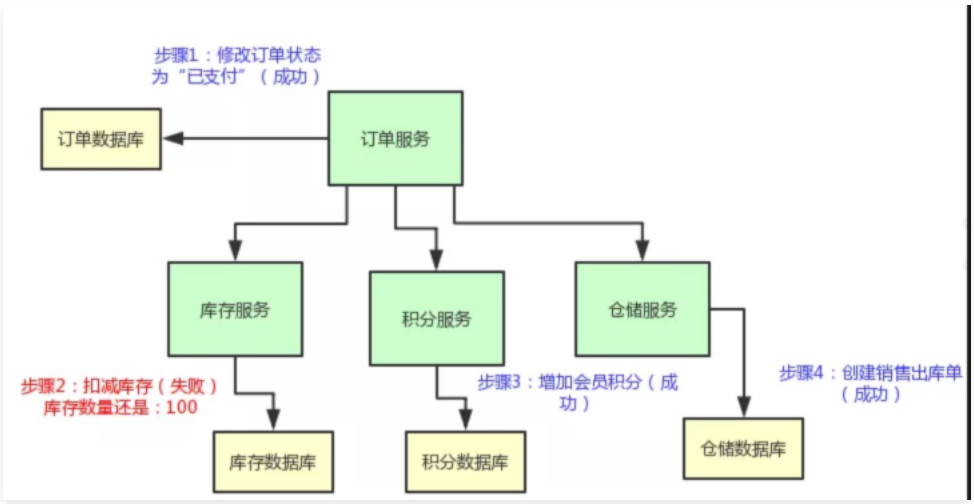
- 更改订单的状态为“已支付”
- 扣减商品库存
- 给会员增加积分
- 创建销售出库单通知仓库发货



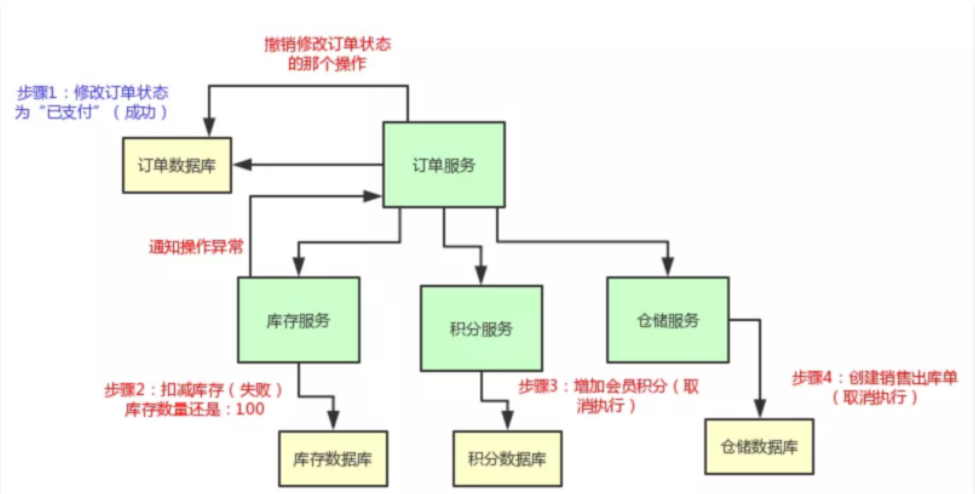
好，业务场景有了，现在我们要更进一步，实现一个 TCC 分布式事务的效果。
什么意思呢？也就是说：
[1] 订单服务-修改订单状态

[3] 积分服务-增加积分
[4] 仓储服务-创建销售出库单。

上述这几个步骤，要么一起成功，要么一起失败，必须是一个整体性的事务。
举个例子，现在订单的状态都修改为“已支付”了，结果库存服务扣减库存失败。那个商品的库存原来是 100 件，现在卖掉了 2 件，本来应该是 98 件了。
结果呢？由于库存服务操作数据库异常，导致库存数量还是 100。这不是在坑人么，当然不能允许这种情况发生了！
但是如果你不用 TCC 分布式事务方案的话，就用个 go开发这么一个微服务系统，很有可能会干出这种事儿来。
我们来看看下面的这个图，直观的表达了上述的过程：



所以说，我们有必要使用 TCC 分布式事务机制来保证各个服务形成一个整体性的事务。
上面那几个步骤，要么全部成功，如果任何一个服务的操作失败了，就全部一起回滚，撤销已经完成的操作。
比如说库存服务要是扣减库存失败了，那么订单服务就得撤销那个修改订单状态的操作，然后得停止执行增加积分和通知出库两个操作。
说了那么多，老规矩，给大家上一张图，大伙儿顺着图来直观的感受一下：



落地实现 TCC 分布式事务

那么现在到底要如何来实现一个 TCC 分布式事务，使得各个服务，要么一起成功？要么一起失败呢？
大家稍安勿躁，我们这就来一步一步的分析一下。咱们就以一个 go开发系统作为背景来解释。

首先，订单服务那儿，它的代码大致来说应该是这样子的：

索引目录

代码块

```
type OrderService struct{
    CreditSrvClient proto.CreditClient //用户积分
    WmsSrvClient proto.WmsClient //记录仓库的变动信息
    InventorySrvClient proto.InventoryClient //库存确认扣减
}

func NewOrderService() *OrderService {
    return &OrderService{
        CreditSrvClient: proto.CreditClient{},
        WmsSrvClient: proto.WmsClient{},
        InventorySrvClient: proto.InventoryClient{},
    }
}

func (o OrderService) UpdateOrderStatus() error {
    return nil
}

func (o OrderService) Notify() error {
    o.UpdateOrderStatus() //更新订单的状态
    o.CreditSrvClient.AddCredit() //增加积分
    o.InventorySrvClient.ReduceStock() //库存确认扣减
    o.WmsClient.SaleDelivery() //记录仓库变更记录
    return nil
}
```

其实就是订单服务完成本地数据库操作之后，通过grpc 来调用其他的各个服务罢了。

但是光是凭借这段代码，是不足以实现 TCC 分布式事务的啊？！兄弟们，别着急，我们对这个订单服务修改点儿代码好不好。

首先，上面那个订单服务先把自己的状态修改为：TRADE_SUCCESS。

这是啥意思呢？也就是说，在 pay() 那个方法里，你别直接把订单状态修改为已支付啊！你先把订单状态修改为 UPDATING，也就是修改中的意思。

这个状态是个没有任何含义的这么一个状态，代表有人正在修改这个状态罢了。

然后呢，库存服务直接提供的那个 reduce_stock() 接口里，也别直接扣减库存啊，你可以是冻结掉库存。

举个例子，本来你的库存数量是 100，你别直接 100 - 2 = 98，扣减这个库存！

你可以把可销售的库存：100 - 2 = 98，设置为 98 没问题，然后在一个单独的冻结库存的字段里，设置一个 2。也就是说，有 2 个库存是给冻结了。

积分服务的 add_credit() 接口也是同理，别直接给用户增加会员积分。你可以先在积分表里的一个预增加积分字段加入积分。

比如：用户积分原本是 1190，现在要增加 10 个积分，别直接 1190 + 10 = 1200 个积分啊！

你可以保持积分为 1190 不变，在一个预增加字段里，比如说 prepare_add_credit 字段，设置一个 10，表示有 10 个积分准备增加。

仓储服务的 sale_delivery() 接口也是同理啊，你可以先创建一个销售出库单，但是这个销售出库单的状态是“UNKNOWN”。

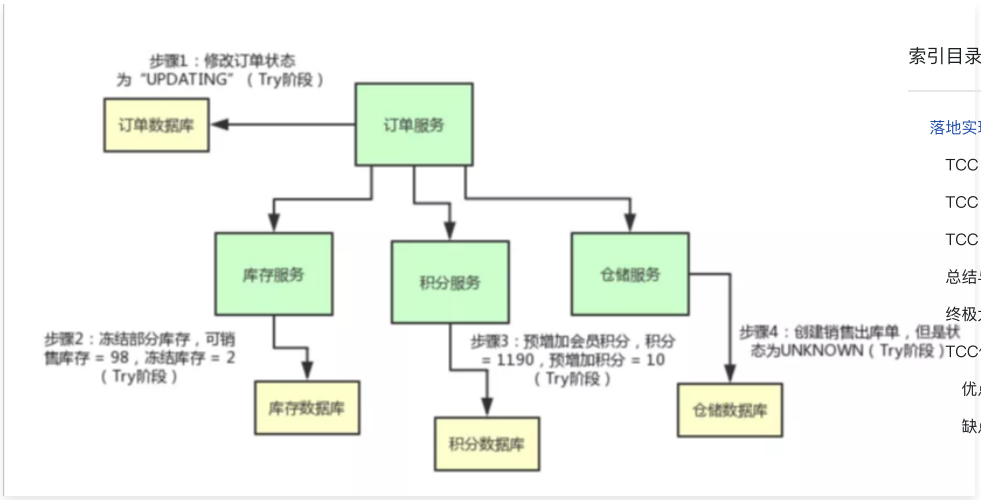
也就是说，刚刚创建这个销售出库单，此时还不确定它的状态是什么呢！

上面这套改造接口的过程，其实就是所谓的 TCC 分布式事务中的第一个 T 字母代表的阶段，也就是 Try 阶段。

总结上述过程，如果你要实现一个 TCC 分布式事务，首先你的业务的主流程以及各个接口提供的业务含义，不是说直接完成那个业务操作，而是完成一个 Try 的操作。

这个操作，一般都是锁定某个资源，设置一个预备类的状态，冻结部分数据，等等，大概都是这类操作。

咱们来一起看看下面这张图，结合上面的文字，再来捋一捋整个过程：



TCC 实现阶段二：Confirm

然后就分成两种情况了，第一种情况是比较理想的，那就是各个服务执行自己的那个 Try 操作，都执行成功了，Bingo！

这个时候，就需要依靠 TCC 分布式事务框架来推动后续的执行了。这里简单提一句，如果你要玩儿 TCC 分布式事务，必须引入一款 TCC 分布式事务框架，比如java国内开源的 seata、ByteTCC、Himly、TCC-transaction。

否则的话，感知各个阶段的执行情况以及推进执行下一个阶段的这些事情，不太可能自己手写实现，太复杂了。

如果你在各个服务里引入了一个 TCC 分布式事务的框架，订单服务里内嵌的那个 TCC 分布式事务框架可以感知到，各个服务的 Try 操作都成功了。

此时，TCC 分布式事务框架会控制进入 TCC 下一个阶段，第一个 C 阶段，也就是 Confirm 阶段。

为了实现这个阶段，你需要在各个服务里再加入一些代码。比如说，订单服务里，你可以加入一个 Confirm 的逻辑，就是正式把订单的状态设置为“已支付”了，大概是类似下面这样子：

代码块

```
1    func (o OrderService) Pay() error {
2        gorm.UpdateStatus("TRADE_SUCCESS")
3    }
```

库存服务也是类似的，你可以有一个 InventoryServiceConfirm 类，里面提供一个 reduce_stock() 接口的 Confirm 逻辑，这里就是将之前冻结库存字段的 2 个库存扣掉变为 0。

这样的话，可销售库存之前就已经变为 98 了，现在冻结的 2 个库存也没了，那就正式完成了库存的扣减。

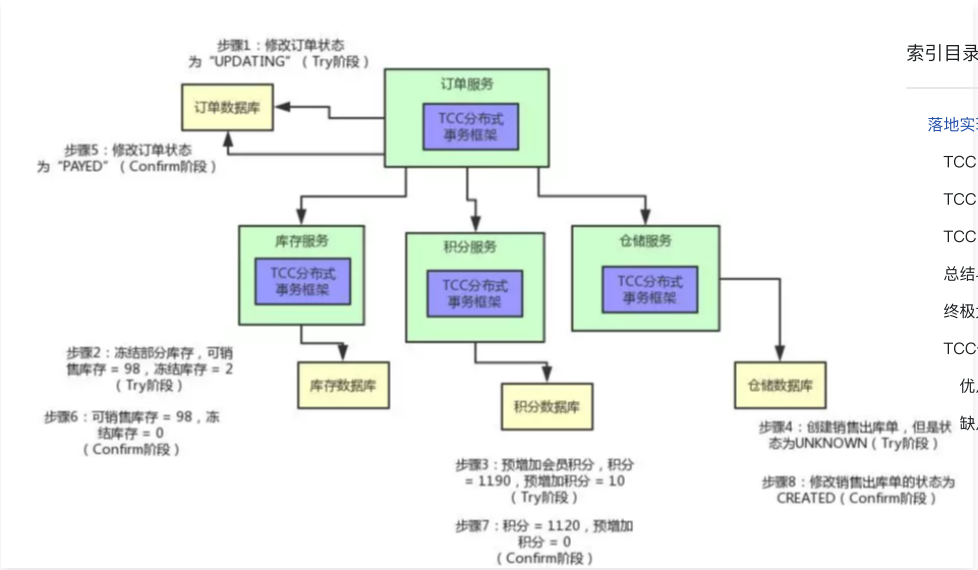
积分服务也是类似的，可以在积分服务里提供一个 CreditServiceConfirm 类，里面有一个 addCredit() 接口的 Confirm 逻辑，就是将预增加字段的 10 个积分扣掉，然后加入实际的会员积分字段中，从 1190 变为 1120。

仓储服务也是类似，可以在仓储服务中提供一个 WmsServiceConfirm 类，提供一个 sale_delivery() 接口的 Confirm 逻辑，将销售出库单的状态正式修改为“已创建”，可以供仓储管理人员查看和使用，而不是停留在之前的中间状态“UNKNOWN”了。

好了，上面各种服务的 Confirm 的逻辑都实现好了，一旦订单服务里面的 TCC 分布式事务框架感知到各个服务的 Try 阶段都成功了以后，就会执行各个服务的 Confirm 逻辑。

订单服务内的 TCC 事务框架会负责跟其他各个服务内的 TCC 事务框架进行通信，依次调用各个服务的 Confirm 逻辑。然后，正式完成各个服务的所有业务逻辑的执行。

同样，给大家来一张图，顺着图一起来看看整个过程：



TCC 实现阶段三：Cancel

好，这是比较正常的一种情况，那如果是异常的一种情况呢?

举个例子：在 Try 阶段，比如积分服务吧，它执行出错了，此时会怎么样?

那订单服务内的 TCC 事务框架是可以感知到的，然后它会决定对整个 TCC 分布式事务进行回滚。

也就是说，会执行各个服务的第二个 C 阶段，Cancel 阶段。同样，为了实现这个 Cancel 阶段，各个服务还得加一些代码。

首先订单服务，它得提供一个 OrderServiceCancel 的类，在里面有一个 pay() 接口的 Cancel 逻辑，就是可以将订单的状态设置为“CANCELED”，也就是这个订单的状态是已取消。

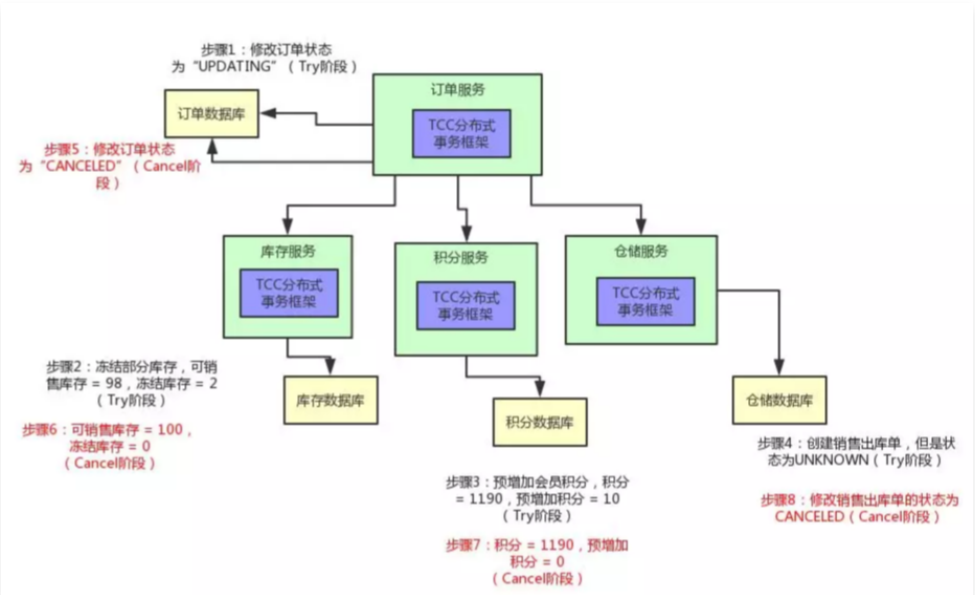
库存服务也是同理，可以提供 reduce_stock() 的 Cancel 逻辑，就是将冻结库存扣减掉 2，加回到可销售库存里去，98 + 2 = 100。

积分服务也需要提供 addCredit() 接口的 Cancel 逻辑，将预增加积分字段的 10 个积分扣减掉。

仓储服务也需要提供一个 sale_delivery() 接口的 Cancel 逻辑，将销售出库单的状态修改为“CANCELED”设置为已取消。

然后这个时候，订单服务的 TCC 分布式事务框架只要感知到了任何一个服务的 Try 逻辑失败了，就会跟各个服务内的 TCC 分布式事务框架进行通信，然后调用各个服务的 Cancel 逻辑。

大家看看下面的图，直观的感受一下：



总结一下，你要玩儿TCC分布式事务的话：

1. 首先需要选择某种TCC分布式事务框架，各个服务里就会有这个TCC分布式事务框架在运行。
2. 然后你原本的一个接口，要改造为3个逻辑，Try-Confirm-Cancel。

- 先是服务调用链路依次执行Try逻辑
- 如果都正常的话，TCC分布式事务框架推进执行Confirm逻辑，完成整个事务
- 如果某个服务的Try逻辑有问题，TCC分布式事务框架感知到之后就会推进执行各个服务的Cancel逻辑，撤销之前执行的各种操作。
- 这就是所谓的TCC分布式事务。
- TCC分布式事务的核心思想，说白了，就是当遇到下面这些情况时，

3. 某个服务的数据库宕机了
4. 某个服务自己挂了
5. 那个服务的redis、elasticsearch、MQ等基础设施故障了
6. 某些资源不足了，比如说库存不够这些

- 先来Try一下，不要把业务逻辑完成，先试试看，看各个服务能不能基本正常运转，能不能先冻结我需要的资源。
- 如果Try都ok，也就是说，底层的数据库、redis、elasticsearch、MQ都是可以写入数据的，并且你保留好了需要使用的一些资源（比如冻结了一部分库存）。
- 接着，再执行各个服务的Confirm逻辑，基本上Confirm就可以很大概率保证一个分布式事务的完成了。
- 那如果Try阶段某个服务就失败了，比如说底层的数据库挂了，或者redis挂了，等等。
- 此时就自动执行各个服务的Cancel逻辑，把之前的Try逻辑都回滚，所有服务都不要执行任何设计的业务逻辑。保证大家要么一起成功，要么一起失败。

终极大招

- 如果有一些意外的情况发生了，比如说订单服务突然挂了，然后再次重启，TCC分布式事务框架是如何保证之前没执行完的分布式事务继续执行的呢?
- TCC事务框架都是要记录一些分布式事务的活动日志的，可以在磁盘上的日志文件里记录，也可以在数据库里记录。保存下来分布式事务运行的各个阶段和状态。
- 万一某个服务的Cancel或者Confirm逻辑执行一直失败怎么办呢?
- 那也很简单，TCC事务框架会通过活动日志记录各个服务的状态。
- 举个例子，比如发现某个服务的Cancel或者Confirm一直没成功，会不停的重试调用他的Cancel或者Confirm逻辑，务必要他成功!
- 当然了，如果你的代码没有写什么bug，有充足的测试，而且Try阶段都基本尝试了一下，那么其实一般Confirm、Cancel都是可以成功的!
- 如果实在解决不了，那么这个一定是很小概率的事件，这个时候发邮件通知人工处理
- seata、 go-seata

TCC优缺点

优点：

1.解决了跨服务的业务操作原子性问题，例如组合支付，订单减库存等场景非常实用
2.TCC的本质原理是把数据库的二阶段提交上升到微服务来实现，从而避免了数据库2阶段中锁冲突的长

事务低性能风险。

3.TCC异步高性能，它采用了try先检查，然后异步实现confirm，真正提交的是在confirm方法中。

缺点：

1.对微服务的侵入性强，微服务的每个事务都必须实现try，confirm，cancel等3个方法，开发成本高，维护改造的成本也高。

2.为了达到事务的一致性要求，try，confirm、cancel接口必须实现等幂性操作。（定时器+重试）

3.由于事务管理器要记录事务日志，必定会损耗一定的性能，并使得整个TCC事务时间拉长，建议采用redis的方式来记录事务日志。

4. tcc需要通过锁来确保数据的一致性，会加锁导致性能不高

4. 两_三阶段提交 上一节 下一节 6. 基于本地消息表的最终一致性

索引目录