终于有人把“TCC分布式事务”实现原理讲明白了！

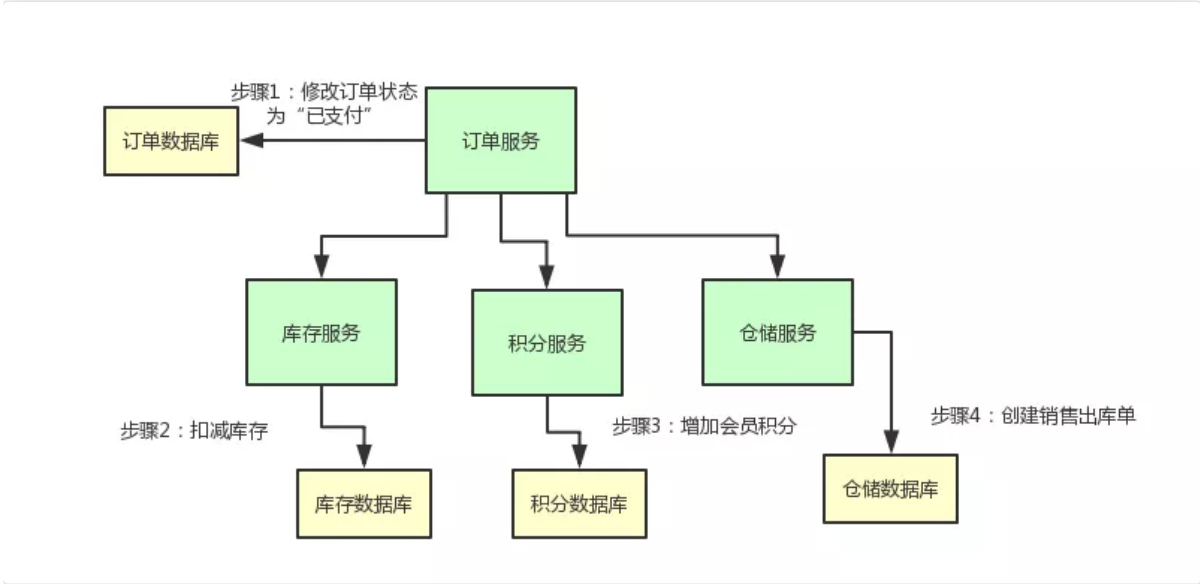
之前网上看到很多写分布式事务的文章，不过大多都是将分布式事务各种技术方案简单介绍一下。很多朋友看了还是不知道分布式事务到底怎么回事，在项目里到底如何使用。

所以这篇文章，就用大白话+手工绘图，并结合一个电商系统的案例实践，来给大家讲清楚到底什么是 TCC 分布式事务。

首先说一下，这里可能会牵扯到一些 Spring Cloud 的原理，如果有不太清楚的同学，可以参考之前的文章：《拜托，面试请不要再问我Spring Cloud底层原理！》。

业务场景介绍

咱们先来看看业务场景，假设你现在有一个电商系统，里面有一个支付订单的场景。



那对一个订单支付之后，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

这是一系列比较真实的步骤，无论大家有没有做过电商系统，应该都能理解。

进一步思考

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，[2] 库存服务-扣减库存，[3] 积分服务-增加积分，[4] 仓储服务-创建销售出库单。

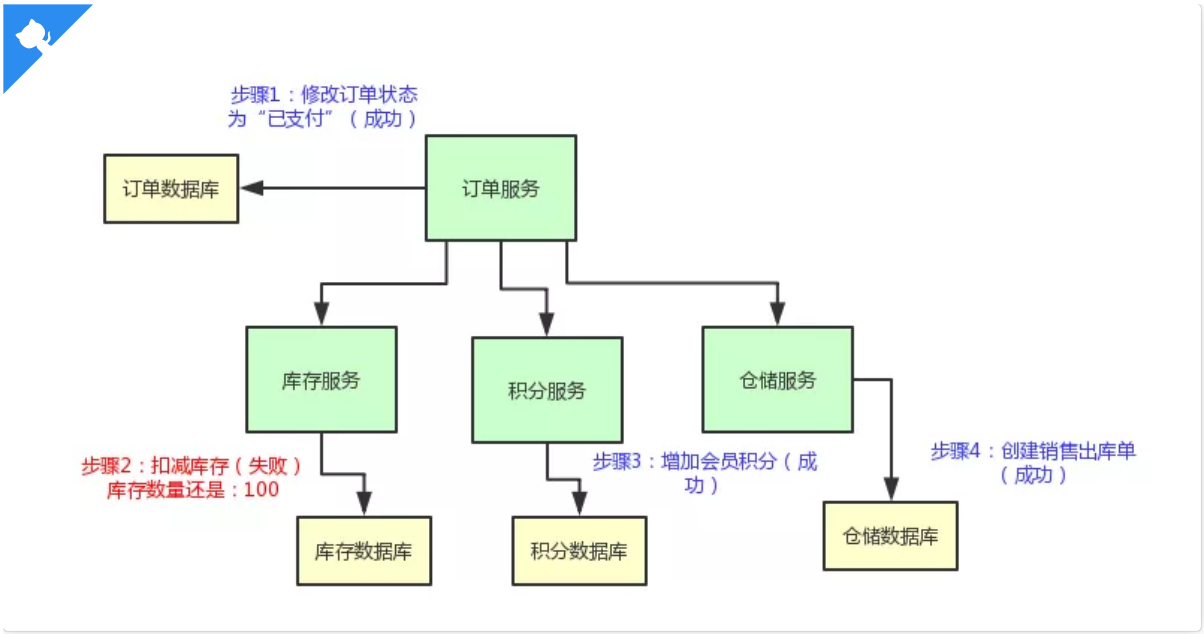
上述这几个步骤，要么一起成功，要么一起失败，必须是一个整体性的事务。

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 Spring Cloud 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：

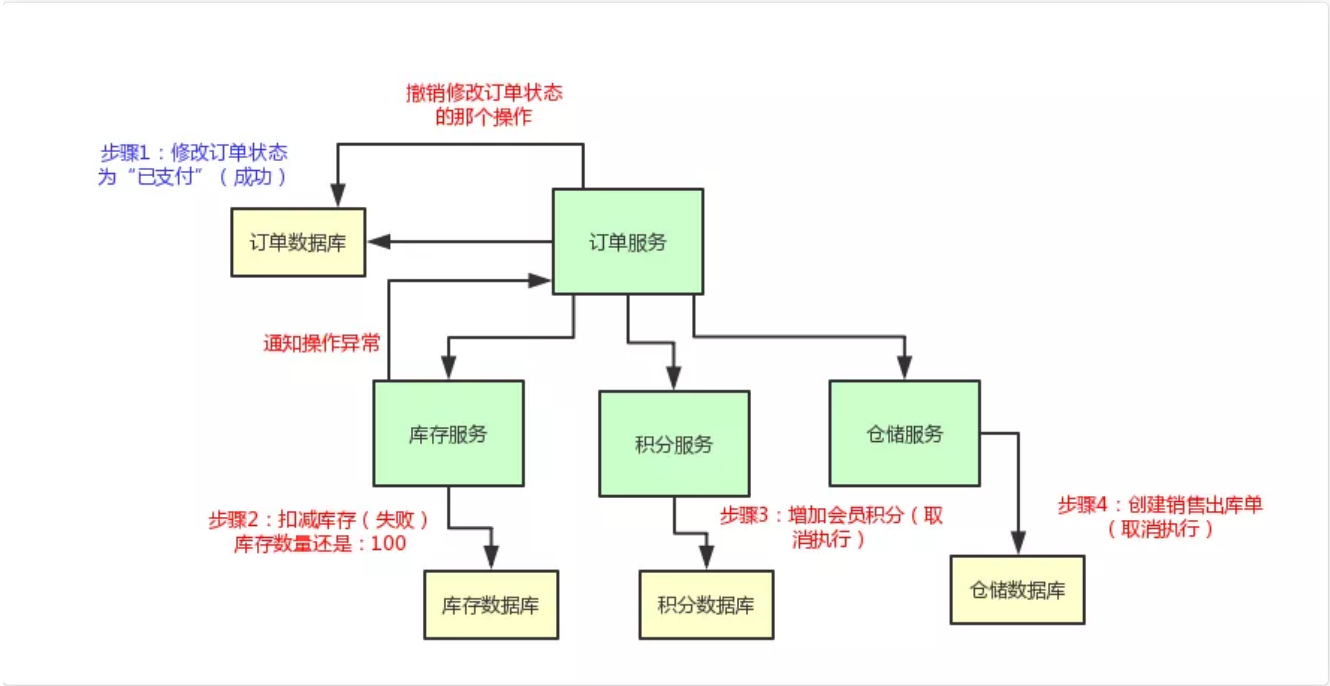


所以说，我们有必要使用 TCC 分布式事务机制来保证各个服务形成一个整体性的事务。

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 Spring Cloud 开发系统作为背景来解释。

TCC 实现阶段一：Try

首先，订单服务那儿，它的代码大致来说应该是这样子的：

```
1  public class OrderService {
2
3      // 库存服务
```

```
    @Autowired
    private InventoryService inventoryService;

    // 积分服务
    @Autowired
    private CreditService creditService;

    // 仓储服务
    @Autowired
    private WmsService wmsService;

    // 对这个订单完成支付
    public void pay(){
        //对本地的的订单数据库修改订单状态为"已支付"
        orderDAO.updateStatus(OrderStatus.PAYED);

        //调用库存服务扣减库存
        inventoryService.reduceStock();

        //调用积分服务增加积分
        creditService.addCredit();

        //调用仓储服务通知发货
        wmsService.saleDelivery();
    }
}
```

如果你之前看过 Spring Cloud 架构原理那篇文章，同时对 Spring Cloud 有一定的了解的话，应该是可以理解上面那段代码的。

其实就是订单服务完成本地数据库操作之后，通过 Spring Cloud 的 Feign 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：OrderStatus.UPDATING。

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 reduceStock() 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 addCredit() 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 saleDelivery() 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”。

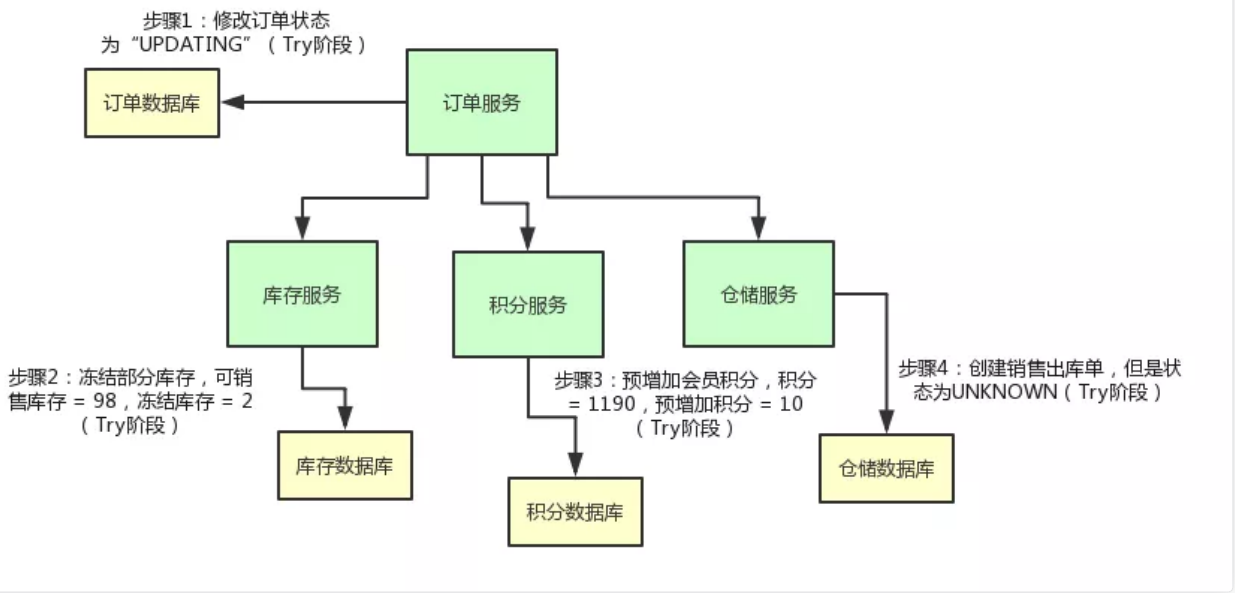
也就是说，刚刚创建这个销售出库单，此时还不确定它的状态是什么呢！

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。

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，比如国内开源的 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：

```
public class OrderServiceConfirm {

    public void pay(){
        orderDao.updateStatus(OrderStatus.PAYED);
    }
}
```

库存服务也是类似的，你可以有一个 InventoryServiceConfirm 类，里面提供一个 reduceStock() 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。

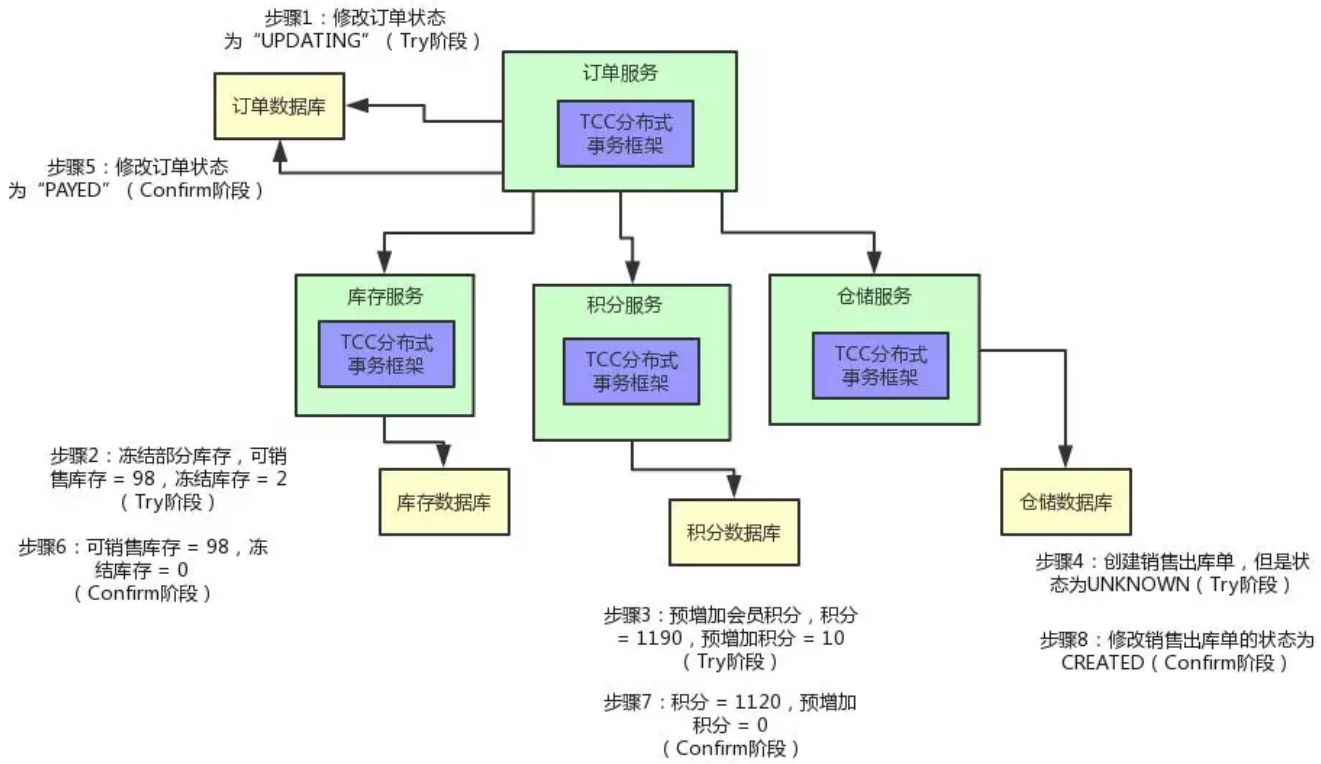
积分服务也是类似的，可以在积分服务里提供一个 CreditServiceConfirm 类，里面有一个 addCredit() 接口的 Confirm 逻辑，就是将预增加字段的 10 个积分扣掉，然后加入实际的会员积分字段中，从 1190 变为 1120。

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 saleDelivery() 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。

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。

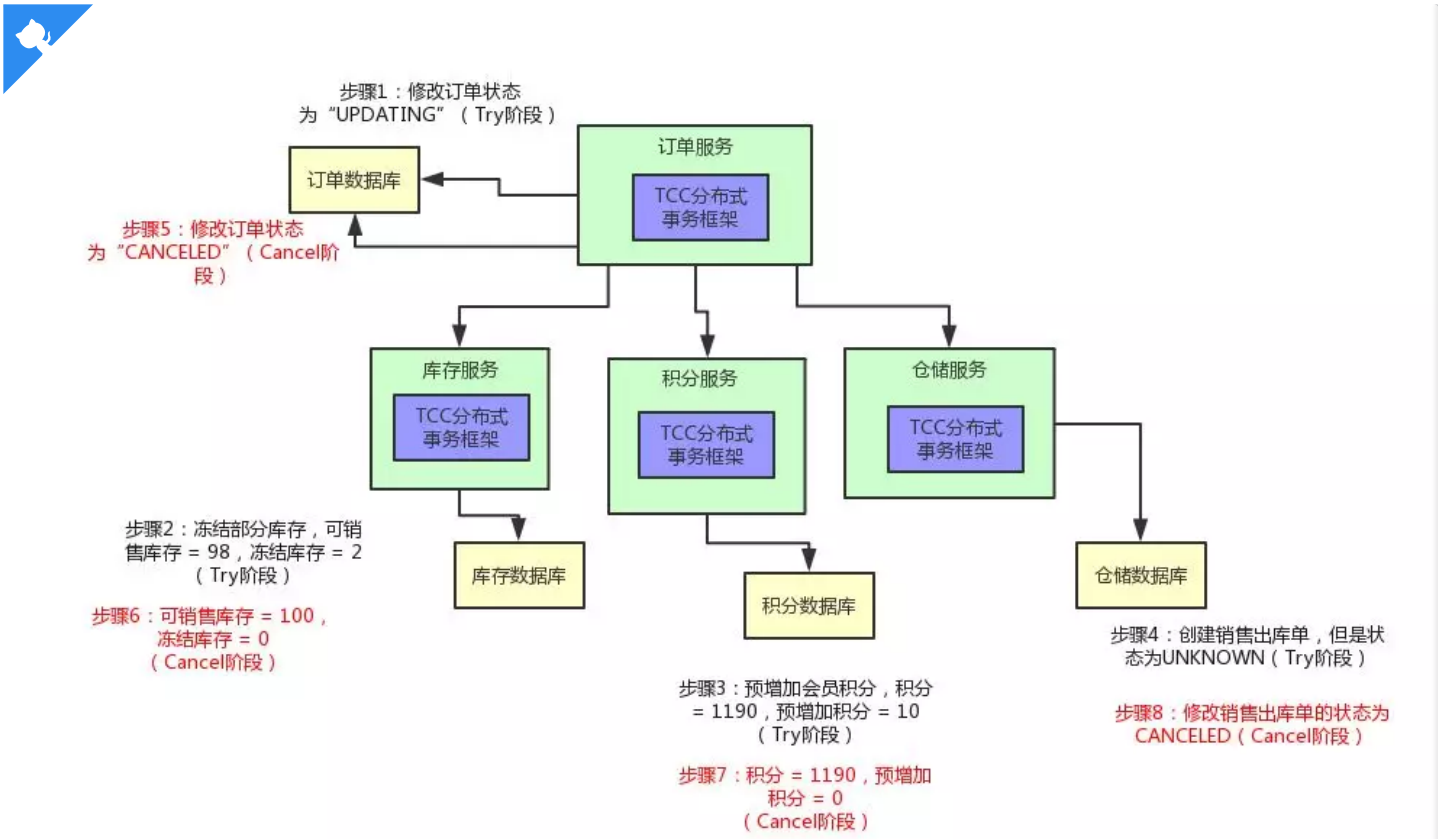
库存服务也是同理，可以提供 reduceStock() 的 Cancel 逻辑，就是将冻结库存扣减掉 2，加回到可销售库存里去，98 + 2 = 100。

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 saleDelivery() 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：



总结与思考

好了，兄弟们，聊到这儿，基本上大家应该都知道 TCC 分布式事务具体是怎么回事了！

总结一下，你要玩儿 TCC 分布式事务的话：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。

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：

- 某个服务的数据库宕机了。
- 某个服务自己挂了。
- 那个服务的 Redis、Elasticsearch、MQ 等基础设施故障了。
- 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。

小伙，你有没有想到一个问题？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？

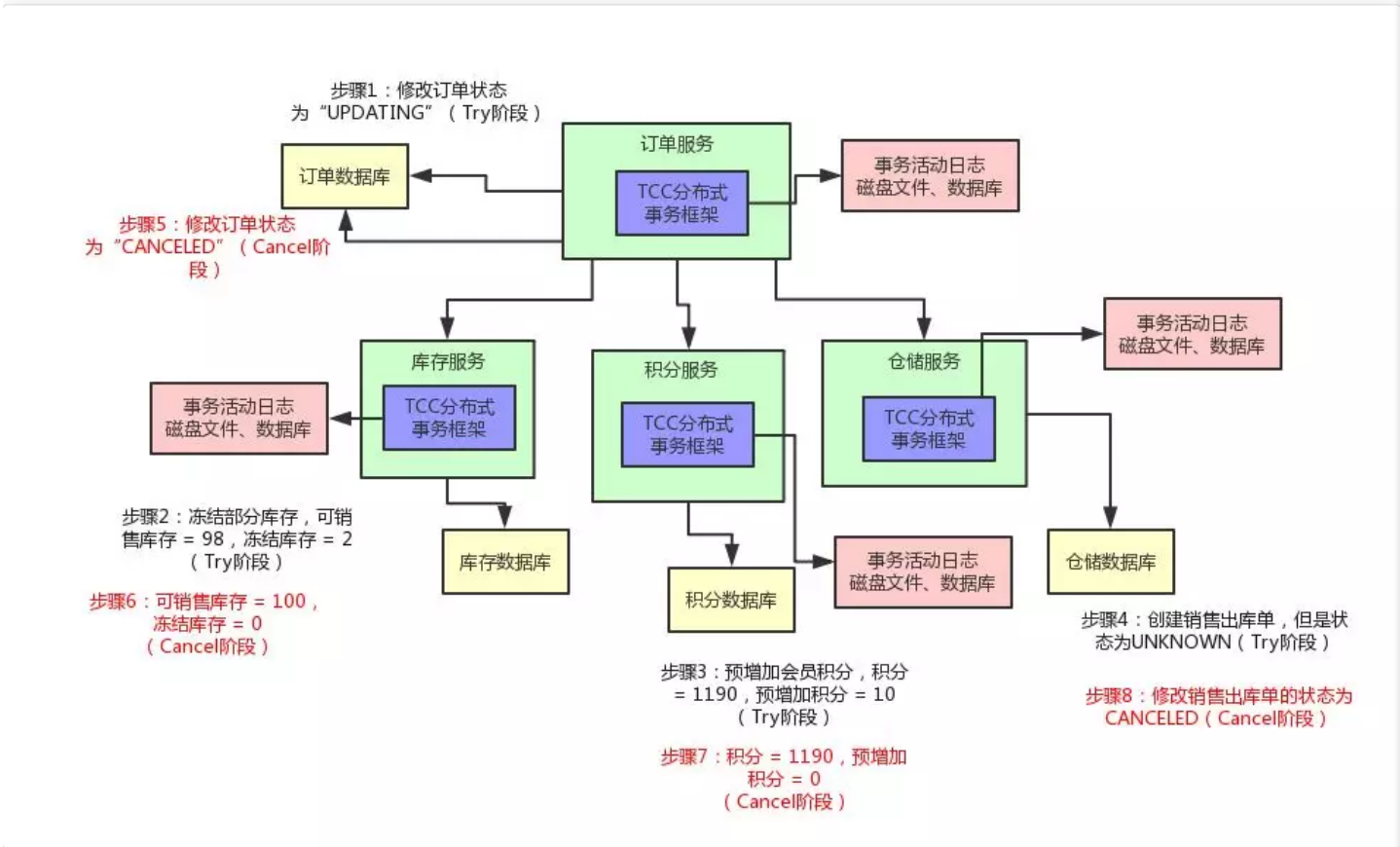
所以，TCC 事务框架都是要记录一些分布式事务的活动日志的，可以在磁盘上的日志文件里记录，也可以在数据库里记录。保存下来分布式事务运行的各个阶段和状态。

问题还没完，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，会不停的重试调用它的 Cancel 或者 Confirm 逻辑，务必要它成功！

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！

最后，再给大家来一张图，来看看给我们的业务，加上分布式事务之后的整个执行流程：



不少大公司里，其实都是自己研发 TCC 分布式事务框架的，专门在公司内部使用，比如我们就是这样。

不过如果自己公司没有研发 TCC 分布式事务框架的话，那一般就会选用开源的框架。

这里笔者给大家推荐几个比较不错的框架，都是咱们国内自己开源出去的：ByteTCC，TCC-transaction，Himly。

大家有兴趣的可以去它们的 GitHub 地址，学习一下如何使用，以及如何跟 Spring Cloud、Dubbo 等服务框架整合使用。

只要把那些框架整合到你的系统里，很容易就可以实现上面那种奇妙的 TCC 分布式事务的效果了。

下面，我们来讲讲可靠消息最终一致性方案实现的分布式事务，同时聊聊在实际生产中遇到的运用该方案的高可用保障架构。

最终一致性分布式事务如何保障实际生产中 99.99% 高可用？

上面咱们聊了聊 TCC 分布式事务，对于常见的微服务系统，大部分接口调用是同步的，也就是一个服务直接调用另外一个服务的接口。

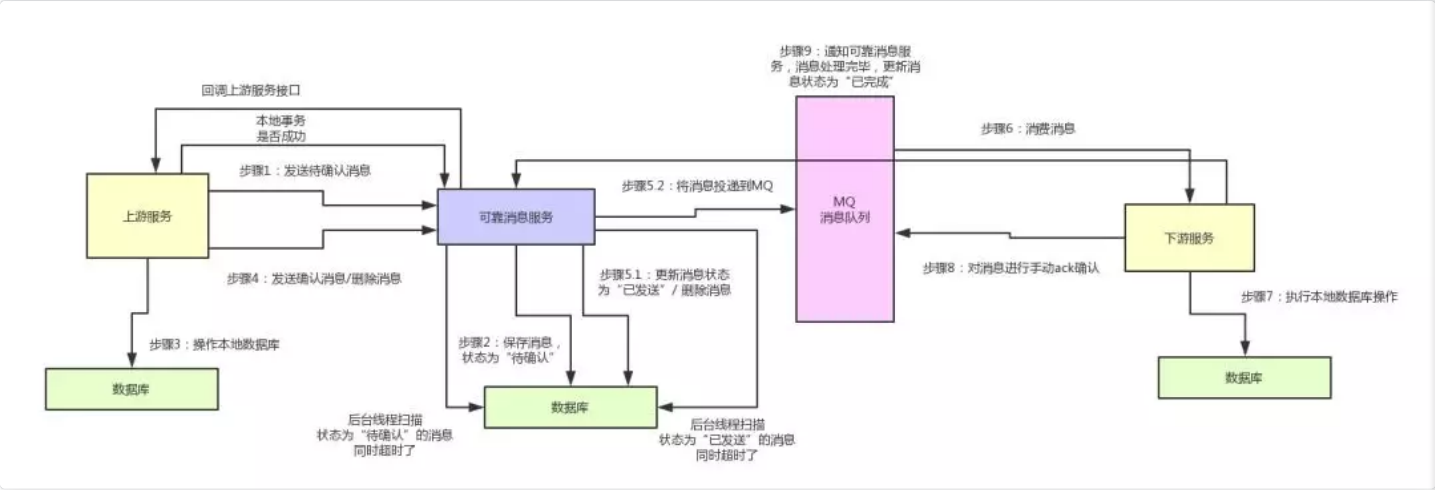
这个时候，用 TCC 分布式事务方案来保证各个接口的调用，要么一起成功，要么一起回滚，是比较合适的。

但是在实际系统的开发过程中，可能服务间的调用是异步的。也就是说，一个服务发送一个消息给 MQ，即消息中间件，比如 RocketMQ、RabbitMQ、Kafka、ActiveMQ 等等。

另外一个服务从 MQ 消费到一条消息后进行处理。这就成了基于 MQ 的异步调用了。

那么针对这种基于 MQ 的异步调用，如何保证各个服务间的分布式事务呢？也就是说，我希望的是基于 MQ 实现异步调用的多个服务的业务逻辑，要么一起成功，要么一起失败。

这个时候，就要用上可靠消息最终一致性方案，来实现分布式事务。



大家看上图，如果不考虑各种高并发、高可用等技术挑战的话，单从“可靠消息”以及“最终一致性”两个角度来考虑，这种分布式事务方案还是比较简单的。

可靠消息最终一致性方案的核心流程

①上游服务投递消息

如果要实现可靠消息最终一致性方案，一般你可以自己写一个可靠消息服务，实现一些业务逻辑。

首先，上游服务需要发送一条消息给可靠消息服务。这条消息说白了，你可以认为是对下游服务一个接口的调用，里面包含了对应的一些请求参数。

然后，可靠消息服务就得把这条消息存储到自己的数据库里去，状态为“待确认”。

接着，上游服务就可以执行自己本地的数据库操作，根据自己的执行结果，再次调用可靠消息服务的接口。

如果本地数据库操作执行成功了，那么就找可靠消息服务确认那条消息。如果本地数据库操作失败了，那么就找可靠消息服务删除那条消息。

此时如果是确认消息，那么可靠消息服务就把数据库里的消息状态更新为“已发送”，同时将消息发送给 MQ。

这里有一个很关键的点，就是更新数据库里的消息状态和投递消息到 MQ。这俩操作，你得放在一个方法里，而且得开启本地事务。

啥意思呢？如果数据库里更新消息的状态失败了，那么就抛异常退出了，就别投递到 MQ；如果投递 MQ 失败报错了，那么就要抛异常让本地数据库事务回滚。这俩操作必须得一起成功，或者一起失败。

如果上游服务是通知删除消息，那么可靠消息服务就得删除这条消息。

②下游服务接收消息

下游服务就一直等着从 MQ 消费消息好了，如果消费到了消息，那么就操作自己本地数据库。

如果操作成功了，就反过来通知可靠消息服务，说自己处理成功了，然后可靠消息服务就会把消息的状态设置为“已完成”。

③如何保证上游服务对消息的 100% 可靠投递?

上面的核心流程大家都看完：一个很大的问题就是，如果在上述投递消息的过程中各个环节出现了问题该怎么办?

我们如何保证消息 100% 的可靠投递，一定会从上游服务投递到下游服务？别着急，下面我们来逐一分析。

上游服务给可靠消息服务发送待确认消息的过程出错了，那没关系，上游服务可以感知到调用异常的，就不用执行下面的流程了，这没问题的。

如果上游服务操作完本地数据库之后，通知可靠消息服务确认消息或者删除消息的时候，出现了问题。

比如：没通知成功，或者没执行成功，或者是可靠消息服务没成功的投递消息到 MQ。这一系列步骤出了问题怎么办？

其实也没关系，因为在这些情况下，那条消息在可靠消息服务的数据库里的状态会一直是“待确认”。

此时，我们在可靠消息服务里开发一个后台定时运行的线程，不停的检查各个消息的状态。

如果一直是“待确认”状态，就认为这个消息出了点什么问题。此时的话，就可以回调上游服务提供的一个接口，问问说，兄弟，这个消息对应的数据库操作，你执行成功了没啊？

如果上游服务答复说，我执行成功了，那么可靠消息服务将消息状态修改为“已发送”，同时投递消息到 MQ。

如果上游服务答复说，没执行成功，那么可靠消息服务将数据库中的消息删除即可。

通过这套机制，就可以保证，可靠消息服务一定会尝试完成消息到 MQ 的投递。

④如何保证下游服务对消息的 100% 可靠接收？

那如果下游服务消费消息出了问题，没消费到？或者是下游服务对消息的处理失败了，怎么办？

其实也没关系，在可靠消息服务里开发一个后台线程，不断的检查消息状态。

如果消息状态一直是“已发送”，始终没有变成“已完成”，那么就说明下游服务始终没有处理成功。

此时可靠消息服务就可以再次尝试重新投递消息到 MQ，让下游服务来再次处理。

只要下游服务的接口逻辑实现幂等性，保证多次处理一个消息，不会插入重复数据即可。

⑤如何基于 RocketMQ 来实现可靠消息最终一致性方案？

在上面的通用方案设计里，完全依赖可靠消息服务的各种自检机制来确保：

- 如果上游服务的数据库操作没成功，下游服务是不会收到任何通知。
- 如果上游服务的数据库操作成功了，可靠消息服务死活都会确保将一个调用消息投递给下游服务，而且一定会确保下游服务务必成功处理这条消息。

通过这套机制，保证了基于 MQ 的异步调用/通知的服务间的分布式事务保障。其实阿里开源的 RocketMQ，就实现了可靠消息服务的所有功能，核心思想跟上面类似。

只不过 RocketMQ 为了保证高并发、高可用、高性能，做了较为复杂的架构实现，非常的优秀。有兴趣的同学，自己可以去查阅 RocketMQ 对分布式事务的支持。

可靠消息最终一致性方案的高可用保障生产实践

背景引入

上面那套方案和思想，很多同学应该都知道是怎么回事儿，我们也主要就是铺垫一下这套理论思想。

在实际落地生产的时候，如果没有高并发场景的，完全可以参照上面的思路自己基于某个 MQ 中间件开发一个可靠消息服务。

如果有高并发场景的，可以用 RocketMQ 的分布式事务支持上面的那套流程都可以实现。

今天给大家分享的一个核心主题，就是这套方案如何保证 99.99% 的高可用。

大家应该发现了这套方案里保障高可用性最大的一个依赖点，就是 MQ 的高可用性。

任何一种 MQ 中间件都有一整套的高可用保障机制，无论是 RabbitMQ、RocketMQ 还是 Kafka。

所以在大公司里使用可靠消息最终一致性方案的时候，我们通常对可用性的保障都是依赖于公司基础架构团队对 MQ 的高可用保

一般来说，大家应该相信兄弟团队，99.99% 可以保障 MQ 的高可用，绝对不会因为 MQ 集群整体宕机，而导致公司业务系统的分布式事务全部无法运行。

但是现实是很残酷的，很多中小型的公司，甚至是一些中大型公司，或多或少都遇到过 MQ 集群整体故障的场景。

MQ 一旦完全不可用，就会导致业务系统的各个服务之间无法通过 MQ 来投递消息，导致业务流程中断。

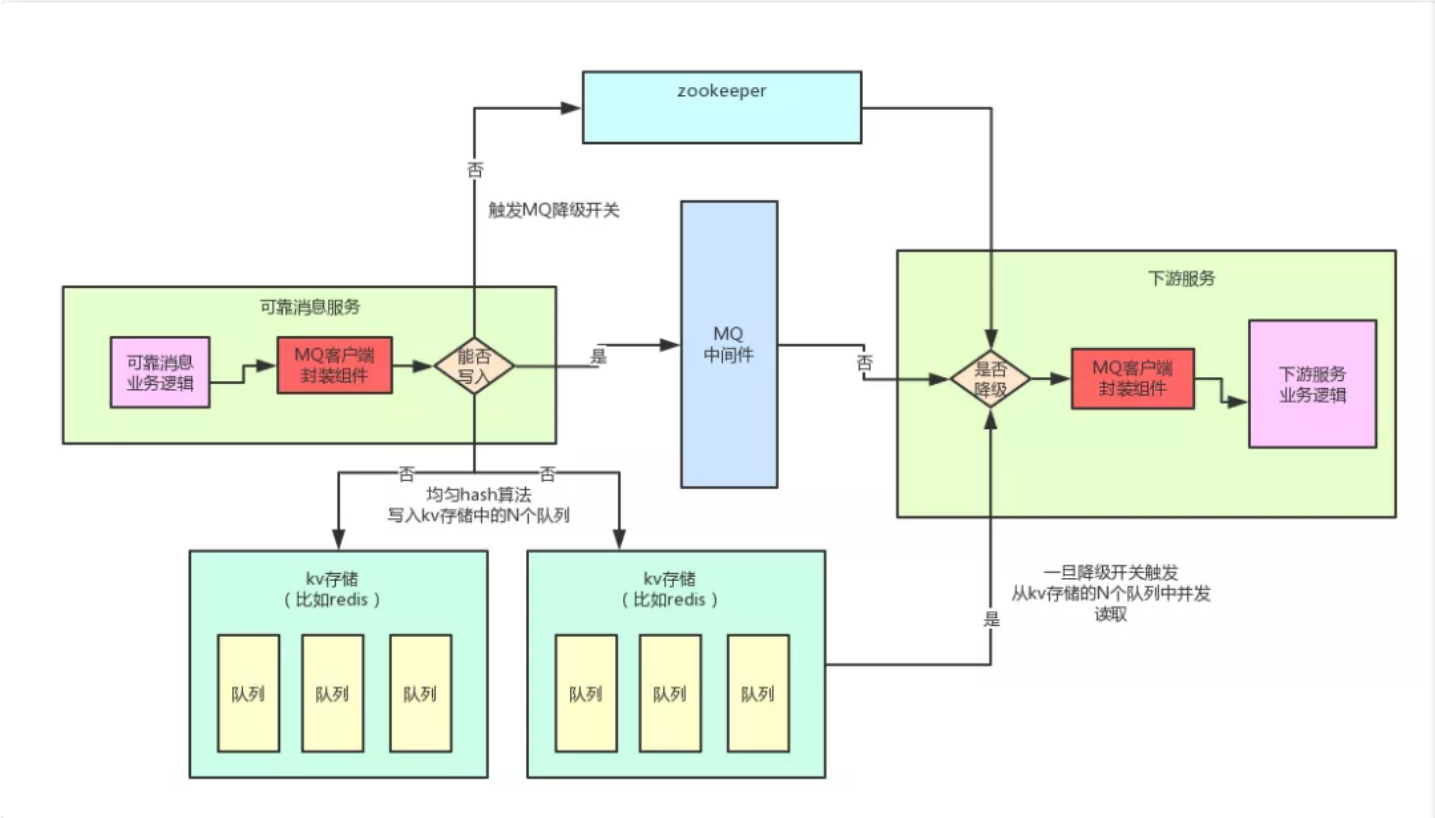
比如最近就有一个朋友的公司，也是做电商业务的，就遇到了 MQ 中间件在自己公司机器上部署的集群整体故障不可用，导致依赖 MQ 的分布式事务全部无法跑通，业务流程大量中断的情况。

这种情况，就需要针对这套分布式事务方案实现一套高可用保障机制。

基于 KV 存储的队列支持的高可用降级方案

大家来看看下面这张图，这是我曾经指导过朋友的一个公司针对可靠消息最终一致性方案设计的一套高可用保障降级机制。

这套机制不算太复杂，可以非常简单有效的保证那位朋友公司的高可用保障场景，一旦 MQ 中间件出现故障，立马自动降级为备用方案。



①自行封装 MQ 客户端组件与故障感知

首先第一点，你要做到自动感知 MQ 的故障接着自动完成降级，那么必须动手对 MQ 客户端进行封装，发布到公司 Nexus 私服上去。

然后公司需要支持 MQ 降级的业务服务都使用这个自己封装的组件来发送消息到 MQ，以及从 MQ 消费消息。

在你自己封装的 MQ 客户端组件里，你可以根据写入 MQ 的情况来判断 MQ 是否故障。

比如说，如果连续 10 次重新尝试投递消息到 MQ 都发现异常报错，网络无法联通等问题，说明 MQ 故障，此时就可以自动感知以及自动触发降级开关。

②基于 KV 存储中队列的降级方案

如果 MQ 挂掉之后，要是希望继续投递消息，那么就必须得找一个 MQ 的替代品。

举个例子，比如我那位朋友的公司是没有高并发场景的，消息的量很少，只不过可用性要求高。此时就可以使用类似 Redis 的 KV 存储中的队列来进行替代。

Redis 本身就支持队列的功能，还有类似队列的各种数据结构，所以你可以将消息写入 KV 存储格式的队列数据结构中去。

PS：关于 Redis 的数据存储格式、支持的数据结构等基础知识，请大家自行查阅了，网上一大堆。

但是，这里有几个大坑，一定要注意一下：

第一个，任何 KV 存储的集合类数据结构，建议不要往里面写入数据量过大，否则会导致大 Value 的情况发生，引发严重的后果。

因此绝不能在 Redis 里搞一个 Key，就拼命往这个数据结构中一直写入消息，这是肯定不行的。

第二个，绝对不能往少数 Key 对应的数据结构中持续写入数据，那样会导致热 Key 的产生，也就是某几个 Key 特别热。

大家要知道，一般 KV 集群，都是根据 Key 来 Hash 分配到各个机器上的，你要是老写少数几个 Key，会导致 KV 集群中的某台机器访问过高，负载过大。

基于以上考虑，下面是笔者当时设计的方案：

- 根据它们每天的消息量，在 KV 存储中固定划分上百个队列，有上百个 Key 对应。
- 这样保证每个 Key 对应的数据结构中不会写入过多的消息，而且不会频繁的写少数几个 Key。
- 一旦发生了 MQ 故障，可靠消息服务可以对每个消息通过 Hash 算法，均匀的写入固定好的上百个 Key 对应的 KV 存储的队列中。

同时需要通过 ZK 触发一个降级开关，整个系统在 MQ 这块的读和写全部立马降级。

③下游服务消费 MQ 的降级感知

下游服务消费 MQ 也是通过自行封装的组件来做的，此时那个组件如果从 ZK 感知到降级开关打开了，首先会判断自己是否还能继续从 MQ 消费到数据？

如果不能了，就开启多个线程，并发的从 KV 存储的各个预设好的上百个队列中不断的获取数据。

每次获取到一条数据，就交给下游服务的业务逻辑来执行。通过这套机制，就实现了 MQ 故障时候的自动故障感知，以及自动降级。如果系统的负载和并发不是很高的话，用这套方案大致是没问题的。

因为在生产落地的过程中，包括大量的容灾演练以及生产实际故障发生时的表现来看，都是可以有效的保证 MQ 故障时，业务流程继续自动运行的。

④故障的自动恢复

如果降级开关打开之后，自行封装的组件需要开启一个线程，每隔一段时间尝试给 MQ 投递一个消息看看是否恢复了。

如果 MQ 已经恢复可以正常投递消息了，此时就可以通过 ZK 关闭降级开关，然后可靠消息服务继续投递消息到 MQ，下游服务在确认 KV 存储的各个队列中已经没有数据之后，就可以重新切换为从 MQ 消费消息。

⑤更多的业务细节

上面说的那套方案是一套通用的降级方案，但是具体的落地是要结合各个公司不同的业务细节来决定的，很多细节多没法在文章里体现。

比如说你们要不要保证消息的顺序性？是不是涉及到需要根据业务动态，生成大量的 Key？等等。

此外，这套方案实现起来还是有一定的成本的，所以建议大家尽可能还是 Push 公司的基础架构团队，保证 MQ 的 99.99% 可用性，不要宕机。

其次就是根据大家公司实际对高可用的需求来决定，如果感觉 MQ 偶尔宕机也没事，可以容忍的话，那么也不用实现这种降级方案。

但是如果公司领导认为 MQ 中间件宕机后，一定要保证业务系统流程继续运行，那么还是要考虑一些高可用的降级方案，比如本文提到的这种。

最后再说一句，真要是一些公司涉及到每秒几万几十万的高并发请求，那么对 MQ 的降级方案会设计的更加的复杂，那就远远不是这么简单可以做到的。

来源：【微信公众号】石杉的架构笔记