Dubbo、ZooKeeper、Spring Cloud、Redis都能否做分布式事务控制？

创建时间： *2017/4/6 16:23*
来源： *https://mp.weixin.qq.com/s?__biz=MzIyNjE4NjI2Nw==&mid=2652558473&idx=1&sn=62609fbb96012d8bd4091fe6120f8fcd&chksm=f39a31fdc4edb8eb42d48e0bab1ccb5132b1b68a778f49b9b0b9c1bbb5a2a4513ec2193b5d2b&mpshare=1&sce*

Dubbo、ZooKeeper、Spring Cloud、Redis都能否做分布式事务控制？

2017-04-01 itegel

分布式一致性

一、写在前面

现今互联网界，分布式系统和微服务架构盛行。

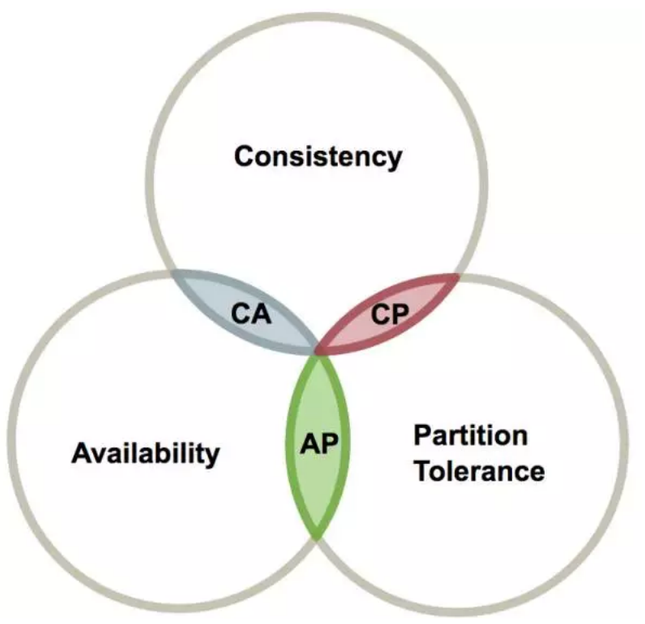
一个简单操作，在服务端非常可能是由多个服务和数据库实例协同完成的。

在互联网金融等一致性要求较高的场景下，多个独立操作之间的一致性问题显得格外棘手。

基于水平扩容能力和成本考虑，传统的强一致的解决方案（e.g.单机事务）纷纷被抛弃。其理论依据就是响当当的CAP原理。

我们往往为了可用性和分区容错性，忍痛放弃强一致支持，转而追求最终一致性。大部分业务场景下，我们是可以接受短暂的不一致的。

本文主要讨论一些最终一致性相关的实现思路。



二、最终一致性解决方案

这个时候一般都会去举一个例子：A给B转100元。

当然，A跟B很不幸的被分在了不同的数据库实例上。甚者这两个人可能是在不同机构开的户。

下面讨论基本都是围绕这个场景的。更复杂的场景需要各位客官发挥下超人的想象力和扩展能力了。

谈到最终一致性，人们首先想到的应该是2PC解决方案。

1. 两阶段提交

两阶段提交需要有一个协调者，来协调两个操作之间的操作流程。当参与方为更多时，其逻辑其实就比较复杂了。

而参与者需要实现两阶段提交协议。Pre commit阶段需要锁住相关资源，commit或rollback时分别进行实际提交或释放资源。

看似还不错。但是考虑到各种异常情况那就比较痛苦了。

举个例子：如下图，执行到提交阶段，调用A的commit接口超时了，协调者该如何做？

我们一般会假设预提交成功后，提交或回滚肯定是成功的（由参与者保障）。

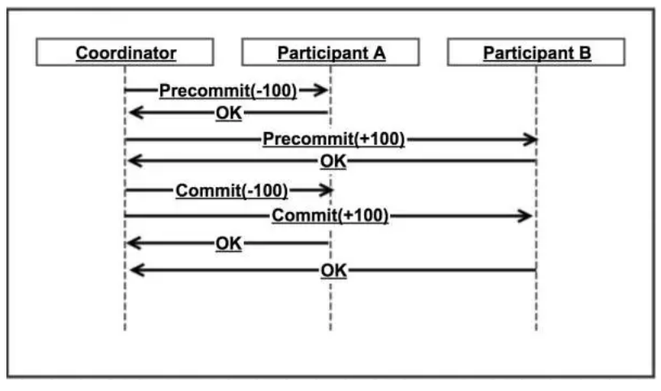
上述情况协调者只能选择继续重试。这也就要求下游接口必须实现幂等（关于幂等的实现下面我们单独再讨论下）。

一般，下游出现故障，不是短时重试能解决的。所以，我们一般也需要有定时去处理中间状态的逻辑。

这个地方，其实如果有个支持重试的MQ，可以扔到MQ。在实践中，我们曾经也尝试自己实现了一个基于MySQL的重试队列。下面还会聊到这一点。

另外，我们也利用了一些外部重试机制。比如支付场景，微信和支付宝都有非常可靠的通知机制。

我们在通知处理接口中做一些重试策略。如果重试失败，就返回微信或支付宝失败。
这样第三方还会接着回调我们(怀疑他们可能发现了我厂回调成功率比其他商户要低^_^)，不过作为小厂，利用一些大厂成熟的机制还是可取的。



2. 异步确保（没有事务消息）

“异步确保”这个词不一定是准确的，还没找到更合适的词，抱歉。

异步化不只是为了一致性，有时候更多的考虑响应时间，下游稳定性等因素。本节只讨论通过异步方案，如何实现最终一致性。

该方案关键是要有个消息表。另外，一般会有个队列，而且我们一般都会假设这个MQ不丢消息。不过很不幸此MQ还不支持事务消息。

基本思路就是：

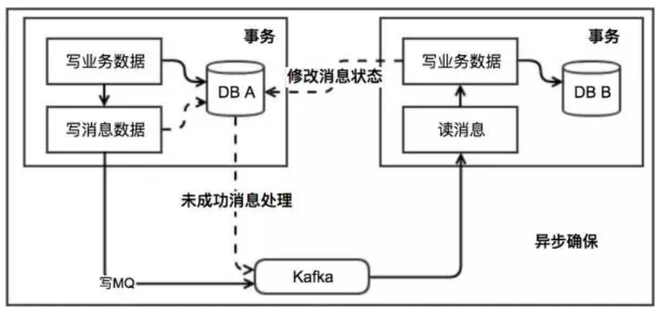
1. 消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交。实现时为了简单，可以只是增加一个字段。新增字段会跟业务强耦合，新增表处理起来不同交易数据可以通用处理。不过因为消息表跟业务需要在一个事务里，所以存储耦合在所难免。
2. 消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，那发送给生产方一个confirm消息，表明已经处理成功了。如果处理失败，该消息还是需要放回MQ的。如果MQ支持重试，那就省事儿了。如果不支持，可以考虑把该消息放回队尾或另建一个队列特殊处理。当然非要处理成功才能继续，那只能block在这条消息了（估计一般人不会这么做）。Kafka lowlevel接口是支持自己设置offset

的，所以可以实现block。

3. 生产方定时扫描本地消息表，把还没处理完成的消息由发送一遍。如果有靠谱的自动对账补账逻辑，其实这一步也可以省略。在实践中，丢消息或者下游处理失败这种场景还是非常少的。这里要看业务上能不能容忍不一致到一个对账补账周期。

当然如果进一步简化，那么MQ也可以不要的。直接用一个脚本处理，一些低频场景，也没啥大问题。当然离线扫表这个事情，总让人不爽。业务量不大且也出初期相信很多人干活儿这事儿。

另外，对一致性要求不高的或者有其他兜底方案的场景（比如较为频繁的对账补账机制），我们就不需要关心消息的confirm等情况，只要扔给消息，就认为万事大吉，一般也是可取的。



上面我们除了处理业务逻辑，还做了很多繁琐的事情。把这些杂活儿都扔给一个中间件多好！这就是阿里等大厂做的事务消息中间件了（比如Notify，RockitMQ的事务消息，请看下节）。

3. 异步确保（事务消息）

事务消息实际上是一个很理想的想法。

理想是：我们只要把消息扔到MQ，那么这个消息肯定会被消费成功。生产方不用担心消息发送失败，也不用担心消息会丢失。【RabbitMQ就支持啊】

回到现实，消费方，如果消息处理失败了，还有机会继续消费，直到成功为止（消费方逻辑bug导致消费失败情况不在本文讨论范围内）。

但遗憾的是市面上大部分MQ都不支持事务消息，其中包括看起来可以一统江湖的kafka。
RocketMQ号称支持，但是还没开源。阿里云据说免费提供，没玩过（羡慕下阿里等大厂内部猿类们）。不过从网上公开的资料看，用起来还是有些不爽的地方。这是后话了，毕竟解决了很多问题。

事务消息，关键一点是把上小节中繁琐的消息状态和重发等用中间件形式封装了。

我厂目前还没提供成熟的支持事务消息的MQ。下面以网传RMQ为例，说明事务消息大概是怎么玩的：

RMQ的事务消息相对于普通MQ，相当于提供了2PC的提交接口。

生产方需要先发送一个prepared消息给RMQ。如果操作1失败，返回失败。

然后执行本地事务，如果成功了需要发送Confirm消息给RMQ。2失败，则调用RMQ cancel接口。

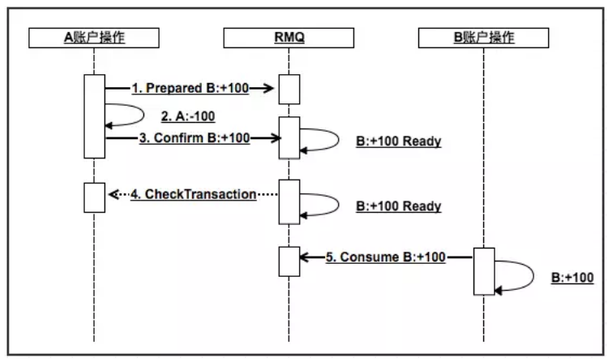
那问题是3失败了（或者超时）该如何处理呢？

别急，RMQ考虑到这个问题了。 RMQ会要求你实现一个check的接口。生产方需要实现该接口，并告知RMQ自己本地事务是否执行成功（第4步）。RMQ会定时轮训所有处于pre状态的消息，并调用对应的check接口，以决定此消息是否可以提交。

当然第5步也可能会失败。这时候需要RMQ支持消息重试。处理失败的消息隔断时间再进行重试，直到成功为止（超过重试次数后会进死信队列，可能得人肉处理了，因为没用过所以细节不是很了解）。

支持消息重试，这一点也很重要。消息重试机制也不仅仅在这里能用到，还有其他一些特殊的场景，我们会依赖他。下一小节，我们简单探讨一下这个问题。

RMQ还是很强大的。我们认为这个程度的一致性已经能够满足绝大部分互联网应用场景。代价是生产方做了不少额外的事情，但相比没有事务消息情况，确实解放了不少劳动力。



P.S. 据说阿里内部因为历史原因，用notify比RMQ要多，他们俩基本原理类似。

4. 补偿交易（Compensating Transaction）

补偿交易，其核心思想是:针对每个操作，都要注册一个与其对应的补偿操作。一般来说操作本身和其补偿（撤销）操作会在一个事务里完成。

当其后续操作失败后，需要按相反顺序完成前面注册的所有撤销操作。

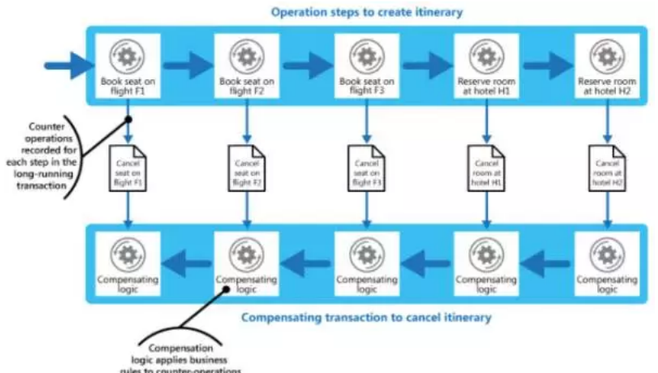
跟2PC比，他的核心价值应该是少了锁资源的代价。流程也相对简单一点。但实际操作中，补偿操作不太好定义，其中间状态处理也会比较棘手。

比如A:-100(补偿为A:+100), B:+100。那么如果B:+100失败后就需要执行A:+100。

曾经有位大牛同事(也是我灰常崇拜的一位技术控)一直热衷于这个思路，相信有些场景用补偿交易模式也是个不错的选择。

他更多是不断思考如何让补偿看起来跟注册个单库事务一样简单。做到业务无感知。

因为本人没有相关实战经验，所以留个链接在这里（https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction），供大家扩展阅读。偷懒了，截个此文中的一张图。



5. 消息重试

上面多次提到消息重试。如果说事务消息重点解决了生产者和MQ之间的一致性问题，那么重试机制对于确保消费者和MQ之间的一致性是至关重要的。

重试可以是pull模式，也可以是push模式。我厂目前已经提供push模式的消息重试，这个还是要赞一下的！

消息重试，重试顾名思义是要解决消息一次性传递过程中的失败场景。举个例子，支付宝回调商户，然后商户系统挂了，怎么办？答案是重试！

一般来说，消息如果消费失败，就会被放到重试队列。如果是延迟时间固定（比如每次延迟2s），那么只需要按失败的顺序进队列就好了，然后对队首的消息，只有当延迟时间到达才能被消费。

这里会有个水位的概念。如果按时间作为水位，那么期望执行时间大于当前时间的消息才是高于水位以上的。其他消息对consumer不可见。

如果要实现每个消息延迟时间不一样，之前想过一种基于队列的方案是，按秒的维度建多个队列。按执行时间入到不同的队列，一天86400个队列（一般丑陋）。然后cosumer按时间消费不同队列。

当然如果不依赖队列可以有更灵活的方案。

之前做支付时候，做了个基于DB的延时队列。每次消息进去时候，都会把下次执行时间设置一下。再对这个时间做个索引....

略土，but it works。毕竟失败的消息不该很多，所以DB容量也不用太在意。很多时候，能跑起来的，简单的架构会得到更多人喜爱。

我厂提供了一种基于redis的延时队列，可以支持消息重试。用到的主要数据结构是redis的zset，按消息处理时间排序。

当然实现起来也没说的那么简单。MQ遇到的持久化问题，内存数据丢失问题，重试次数控制，消息追溯等等都需要有一些额外的开发量。

综上，如果MQ能够提供消息重试特性，那就不要自己折腾了。这里还是有不少坑的。

6. 幂等（接口支持重入）

即使没有MQ，重试也是无处不在的。所以幂等问题不是因为用到MQ后引入的，而是老问题。

幂等怎么做？

如果是单条insert操作，我们一般会依赖唯一键。如果一个事务里包含一个单条insert，那也可以依赖这条insert做幂等，当insert抛异常就回滚事务。

如果是update操作，那么状态机控制和版本控制异常重要。这里要多加小心。

再复杂点的，可以考虑引入一个log表。该log对操作id（消息id？）进行唯一键控制。然后整个操作用事务控制。当插入log失败时整个事务回滚就好了。

有人会说先查log表或者利用redis等缓存，加锁。我想说的是这个基本上都不work。除非在事务里进行查寻。所以建议，索性让代码简单点，直接插入，依赖数据库唯一键冲突回滚掉就好了。

用唯一键挡重入是目前为止个人觉得最有安全感的方式。当然对数据库会有一些额外性能损耗。问题就变成了有多大的并发，其中又有多大是需要重试的？

我相信Fasion IO卡+分库分表之后，想达到数据库性能瓶颈还是有点难度的（主要是针对金融类场景）。

事务消息

人们一般用MQ只是实现了最大努力通知模型。

我们最近尝试实现一种事务消息解决方案。

我们希望这个方案是轻量级的，能实际解决目前业务痛点。

接下来几乎看不到任何理论分析，只谈做什么，怎么做。因为涉及到一些内部资料，有所省略。

实现事务消息核心是需要有一个支持消息重试的MQ。经调研，我厂有个自研的基于Redis的延时MQ服务。我们暂且叫他DelayQ吧。目前我厂还没有类似RMQ的事务消息方案。
据称RocketMQ和ActiveMQ等均有相似特性。您猜对了，我都没用过。请各位客官帮忙告知下还有哪些有此特性。

DelayQ

DelayQ主要功能是支持不同的消息延迟策略。提供两个核心接口：addMsg， deleteMsg.

1. addMsg负责将一个重试消息注册到DelayQ中。其核心参数有：

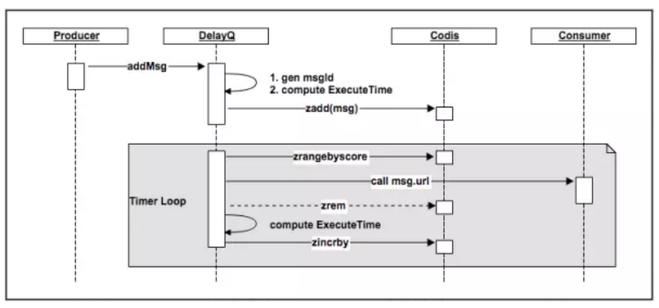
最大重试次数，重试间隔， 生效时间，下游Url级对应业务参数。

2. deleteMsg负责主动将上面注册的消息删除掉。

大致流程是：producer通过addMsg注册一条延迟消息，DelayQ负责在生效时间点将次消息push给下游consumer。如果下游返回成功，则不再继续发送。否则，会每隔重试间隔尝试发送，直到最大重试次数为止。

在一些特殊场景，producer或consumer可以通过deleteMsg接口主动删除该消息。

我厂DelayQ是基于Redis的zset实现的。我们只是个业务团队，所以也没有参与其开发。下面根据个人理解，简单说明下他大概是如何工作的（对显示器发誓，此图只是本人yy之作。如有雷同切莫当做泄露公司机密。我确实也没看到详细的设计文档。正因此也是仅供参考，实操性不大）。



如上所述，关键数据结构就是zset。Redis集群方案选用Codis。

每次addMsg时，都会给该消息生成一个唯一id。然后计算其下次尝试时间。这里就是DelayQ要实现的核心策略部分。比如，我们可以每次间隔相同时间，也可以是指数衰减的间隔去尝试。为了实现这些，我们还需要在消息体上面额外记录一些信息。比如，上次尝试时间，总共已尝试次数等信息。这些都搞定了就将消息和其他一些必要信息写到zset中。zset以ExecuteTime作为Score排序。

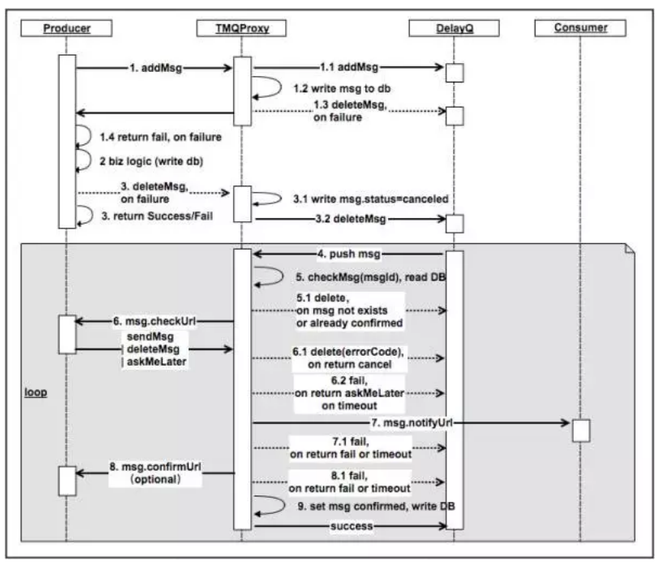
消息进到zset后，DelayQ会通过timer触发(比如秒级)，fork相应的消费线程去处理zset里ExecuteTime大于当前时间的消息。DelayQ拿到一条消息后，解析其中的callbackurl，并组装参数，push业务消息给Consumer。

Consumer返回处理成功，那么zrem Codis里的消息。如果处理失败，则计算其下次尝试时间，并更新其ExecuteTime。

YY主要流程如上。相信实际实现过程中，还有很多问题需要解决。比如，定时处理，如何提高并发度？考虑到redis丢消息情况，还需要做些啥？回调接口超时和限流问题等等。因为理解不深，所以不敢继续写了，怕露馅^_^

事务消息

基于上面的DelayQ,我们尝试提出一种可靠的消息传递机制（事务消息）。核心思想抄袭RMQ。直接上图。



我们在DelayQ前面增加一层Proxy，暂且就叫TMQProxy吧。producer将通过TMQProxy跟DelayQ交互，不会直接跟DelayQ进行交互。

TMQ相对于DelayQ要求Producer多提供两个信息，一个是checkUrl, 一个是confirmUrl(可选)。producer需要分别实现这两个接口。

check接口：主要是告知TMQProxy，当前消息是否可以发送。对应上面步骤6.

confirm接口：可选接口。TMQProxy将消息处理成功后会通过该主动通知Producer消息已经处理成功。如果producer对此结果不感兴趣，就可以不必实现。

详细过程如上面交互图。就不一一解释了。如果对实现感兴趣，还是建议稍微仔细看下上图，画这个还是费了点脑细胞的。相信大家还是能看明白大致思想。

我们引入TMQProxy还有一个目的是，不想让内部业务使用DelayQ上太过花哨。比如其内部topic这些，我们都屏蔽掉了。重试策略这些也不希望业务层玩得太灵活，所以只能提供枚举的策略。

另外一个考虑是，DelayQ可能也只是个阶段性方案，后续如果切换其他MQ。我们希望尽量做到业务方无感知。直接通过Proxy屏蔽底层具体的MQ实现。

TMQProxy提供的接口大致如下（示例）：

参数名	含义	必填	备注
producer	String	是	
msgType	消息类型	否	细分策略和定位问题用
strategy	重试策略	否	TMQProxy提供可选策略
transactionId	业务方交易id	否	业务方交易Id,定位问题用
checkUrl	check消息地址	是	get方式，需要把需要的参数拼好
confirmUrl	消息处理成功后通知接口	否	get方式，需要把需要的参数拼好
remoteUrl	下游push地址	是	
method	get/post	是	
headers	下游headers	否	
charset	请求下游字符集	否	
params	请求下游实际参数	是	透传
timeout	请求下游超时时间，connectior	否	

稍微啰嗦下timeout。timeout设置合理性还是很重要的。这里check和confirm的超时时间，我们会限制比较小的时间。如果不满足，则可能拒绝接入TMQProxy。而下游的处理时间一般都会比较长。这里，我们可以容忍第一次超时。但是处理成功后第二次请求还耗很长时间是接受不了的。所以需要下游该加结果cache加结果cache，该优化幂等算法优化幂等算法。如果连续timeout多次，其实这条消息可以考虑丢弃。

简化

细心的朋友可能发现上一节的图里多出了个DB，主要是存储消息的一些状态。引入DB目的是完全代替业务侧的消息表。可以在消息状态查询，故障恢复等中起到兜底作用。当然，也可以有效减少查询check接口的次数。

实际应用中，我们认为这个DB的引入有点过重。因为Redis和DelayQ稳定性已然经受住了更大业务量的考验。目前已经趋于成熟了，也没发生过大面积丢消息等严重事故。

所以我们在TMQProxy中，把DB先抛弃了。上面所有DB相关操作实际上，我们是没有实现的。所以消息Id实际上也是用了DelayQ返回的消息Id。消息Id，主要用于delete。这里有点小坑，就不扩展了。实际应用中，因为我们通过同步接口中返回码控制消息是否要删除掉，所以饶过了这个坑。

所以，很多时候，我们设计时候尽量考虑全，实际实现和应用的时候会做很多tradeoff。据说这个是架构师关键素养。这话题小的就不敢扩展了。

还有啥

一个系统，如果只是画几张图，把代码写完就万事大吉该多好？可惜，我们想让他上线，还需要考虑很多很多问题（考虑的永远都不够多）。因为有点跑题，所以就不再扩展，只是蜻蜓点水，纯属凑字数了。

1. 限流。对于一个通用服务，因为接入方杂七杂八，所以不管是producer请求频次控制，还是对下游的保护，都离不开限流措施。可以在不同层进行限流。接口层，我们考虑按producer+msgType等多个维度进行限流。做全局限流还有点麻烦。假设loadbalancer足够均匀，我们只对单机做限流基本就够了。这里推荐com.google.common.util.concurrent.RateLimiter，非常好用。感谢肖少早前推荐。
2. 监控。不怕出问题，就怕出了问题不知道。监控非常重要。关键是监控些啥？不展开了，这个真的非常非常重要。相信大部分厂子都有自己的一套相对完整的监控系统。
3. 降级和兜底。首先Producer可以把TMQProxy当做直接调用下游失败后的补充手段。这样就不会对TMQProxy产生强依赖。TMQProxy自身没有存储，所以除了逻辑错误，最大可能就是下游DelayQ故障了。此时，因为没有存储，只能通过日志进行恢复了。我们需要规范日志打印，并准备好处理脚本。把故障期间消费失败的消息重新处理一遍。理论上只要DelayQ不丢消息，我们也可以等待DelayQ恢复后重试。不过如果add就失败，那根据上面流程，业务上肯定是会有损失的。所以我们对DelayQ的稳定性，要求至少是4个9。所以更多的降级，应该在producer侧结合业务去考虑。对于DelayQ丢消息场景，目前大部分业务场景只能通过对账进行发现和补账。当然，有些业务自己会实现主动查询机制。通过定期或人为（可能是用户查询，也可能是后台查询）触发查询下游，并同步状态。
4. 部署。如果有条件，最好独立部署。作为proxy，可能性能瓶颈在网络IO上。所以消息体不应过大。一种常见做法是Producer发给下游的只是一个Id，下游获得该消息后，还得查一下producer才能拿到全部信息。实际情况还是看量。
5. 压测。线上线下压测还是要经常做一做的。上线前做一次可能不够。因为逻辑可能会有变化。线下压测可能也不够，因为线上应用场景跟线下模拟的可能不同。线上线下机器配置可能也不一致。所以有条件做全链路压测，那就圆满了。全链路压测是个浩大的工程，大厂似乎都在玩。比如微信动不动就演练....

> **来源：https://www.zhihu.com/people/itegel/**
>
> https://zhuanlan.zhihu.com/p/25933039