应用的幂等是在分布式系统设计时必须要考虑的一个方面，如果对幂等没有额外的考虑，那么在消息失败重新投递，或者远程服务重试时，可能会出现许多诡异的问题。这一课时一起来看一下，在消息队列应用中，如何处理因为重复投递等原因导致的幂等问题。

对业务幂等的理解

首先明确一下，**幂等并不是问题**，而是业务的一个特性。幂等问题体现在对于不满足幂等性的业务，在消息重复消费，或者远程服务调用失败重试时，出现的数据不一致，业务数据错乱等现象。

幂等最早是一个数学上的概念，幂等函数指的是对一个函数或者方法，使用相同的参数执行多次，数据结果是一致的。

以 HTTP 协议为例，我们知道 HTTP 协议中定义了交互的不同方法，比如 GET 和 POST，以及 PUT、DELETE 等，其中 GET、DELETE 等方法都是幂等的，而 POST 方法不是。

这个很好理解，GET 方法用于获取资源，不管调用多少次接口，结果都不会改变，所以是幂等的，DELETE 等可以类比。

这里有一点需要注意，业务上的幂等指的是操作不影响资源本身，并不是每次读取的结果都保证一致。比如通过 GET 接口查询一条订单记录，在多次查询的时间段内，订单状态可能会有新的更新而发生变化，查询到的数据可能不同，但是读接口本身仍然是一个幂等的操作。

在业务开发中对数据的操作主要是 CRUD，即在做数据处理时的 Create、Read、Update、Delete 这几种操作。很明显，这里的 Create 操作不是幂等的，Update 操作可能幂等也可能不幂等。例如，现在有一个订单表，下面的操作就是幂等的：

```
UPDATE order SET status=1 WHERE id=100
```

下面的这个操作，就不符合幂等性的要求：

```
UPDATE order SET price=price+1 WHERE id=100
```

对应的，Read 和 Delete 操作则是幂等的。

各类中间件对幂等性的处理

幂等处理不好，可能会出现很多问题，比如使用 binlog 分发进行数据同步，如果数据库更新消息被多次消费，可能会导致数据的不一致。

- 远程服务调用的幂等问题

因为存在网络抖动等，远程服务调用出现失败，一般是通过配置重试，保证请求调用成功率，提高整体服务的可用性。

以 Apache Dubbo 为例，我一直觉得 Dubbo 对容错的支持特别全面，它支持多种集群容错的方式，并且可以针对业务特性，配置不同的失败重试机制，包括 Failover 失败自动切换、Failsafe 失败安全、Failfast 快速失败等。比如在 Failover 下，失败会重试两次；在 Failfast 下，失败则不会重试，直接抛出异常。

Dubbo 的容错机制考虑了多种业务场景的需求，根据不同的业务场景，可以选择不同的容错机制，进而有不同的重试策略，保证业务正确性。

Dubbo RPC 的重试和容错机制不是本课时的重点，如果想对 Dubbo 集群容错方式有进一步的了解，可以点击查看 Dubbo 官方文档。

- 消息消费中的重试问题

从本质上来讲，消息队列的消息发送重试，和微服务中的失败调用重试是一样的，都是通过重试的方式，解决网络抖动、传输不稳定等导致的偶发调用失败。这两者其实是一个问题，两个问题的解决方式也可以互相借鉴。

在分布式系统中，要解决这个问题，需从中间件和业务的不同层面，来保证服务调用的幂等性。下面从消息队列投递语义，以及业务中如何处理幂等，两个方面进行拆解。

消息投递的几种语义

为了进一步规范消息的调用，业界有许多消息队列的应用协议，其中也对消息投递标准做了一些约束。

- At most once

消息在传递时，最多会被送达一次，在这种场景下，消息可能会丢，但绝不会重复传输，一般用于对消息可靠性没有太高要求的场景，比如一些允许数据丢失的日志报表、监控信息等。

- At least once

消息在传递时，至少会被送达一次，在这种情况下，消息绝不会丢，但可能会出现重复传输。

绝大多数应用中，都是使用至少投递一次这种方式，同时，大部分消息队列都支持到这个级别，应用最广泛。

- Exactly once

每条消息肯定会被传输一次且仅传输一次，并且保证送达，因为涉及发送端和生产端的各种协同机制，绝对的 Exactly once 级别是很难实现的，通用的 Exactly once 方案几乎不可能存在，可以参考分布式系统的「FLP 不可能定理」。

我觉得消息投递的语义，和数据库的隔离级别很像，不同语义的实现，付出的成本也不一样。上面定义的消息投递语义，主要在消息发送端，在消费端也可以定义类似的消费语义，比如消费端保证最多被消费一次，至少被消费一次等，这两种语义是相对应的，可以认为是同一个级别的两种描述。

不同消息队列支持的投递方式

以 RocketMQ 为例，我们来看下对应的投递支持。

RocketMQ 支持 At least once 的投递语义，也就是保证每个消息至少被投递一次。在 RocketMQ 中，是通过消费端消费的 ACK 机制来实现的：

> 在消息消费过程中，消费端在消息消费完成后，才返回 ACK，如果消息已经 pull 到本地，但还没有消费，则不会返回 ack 响应。

在业务上应用 RcoketMQ 时，也可以根据不同的业务场景实现其他级别的投递语义，比如最多送达一次等，由于篇幅限制这里不展开详细讲解了，感兴趣的同学可以查阅 RocketMQ 相关的源码和文档学习。

业务上如何处理幂等

消息消费的幂等和我们在上一课时中提到的时序性一样，本质上也是一个系统设计的问题。

消息队列是我们为了实现系统目标而引入的手段之一，并且分布式消息队列天然存在消费时序、消息失败重发等问题。所以要保证消息队列的消费幂等，还是要回到业务中，结合具体的设计方案解决。

天然幂等不需要额外设计

参考上面对 HTTP 协议方法的幂等性分析，有部分业务是天然幂等的，这部分业务，允许重复调用，即允许重试，在配置消息队列时，还可以通过合理的重试，来提高请求的成功率。

利用数据库进行去重

业务上的幂等操作可以添加一个过滤的数据库，比如设置一个去重表，也可以在数据库中通过唯一索引来去重。

举一个例子，现在要根据订单流转的消息在数据库中写一张订单 Log 表，我们可以把订单 ID 和修改时间戳做一个唯一索引进行约束。

当消费端消费消息出现重复投递时，会多次去订单 Log 表中进行写入，由于我们添加了唯一索引，除了第一条之外，后面的都会失败，这就从业务上保证了幂等，即使消费多次，也不会影响最终的数据结果。

设置全局唯一消息 ID 或者任务 ID

还记得我们在第 15 课时「分布式调用链跟踪」中，提到的调用链 ID 吗？调用链 ID 也可以应用在这里。我们在消息投递时，给每条业务消息附加一个唯一的消息 ID，然后就可以在消费端利用类似分布式锁的机制，实现唯一性的消费。

还是用上面记录订单状态流转消息的例子，我们在每条消息中添加一个唯一 ID，消息被消费后，在缓存中设置一个 Key 为对应的唯一 ID，代表数据已经被消费，当其他的消费端去消费时，就可以根据这条记录，来判断是否已经处理过。

总结

这一课时分享了消息幂等的知识点，包括对幂等的理解，以及消息队列投递时的不同语义，另外简单介绍了业务上处理幂等的两种方式。

西方有一句谚语：当你有了一个锤子，你看什么都像钉子。在我刚开始学习分布式系统时，学习了各种中间件，每个中间件都希望能用上，这其实脱离了系统设计的初衷。

课程内容到这里，已经展开了许多分布式系统的常用组件，提到这个谚语，主要是希望你在做技术方案，特别是做分布式系统设计方案时，不是为了设计而设计。方案设计的目的是**实现业务目标**，并不是在系统中加入各种高大上的中间件，这个方案就是正确的。

我之前读过一本《系统之美》的图书，从复杂系统的角度来看，系统中的元素越多，为了维持系统的平衡，需要付出的势能必然也越大。

对应到系统设计中，系统拆解的粒度越大，对应各个组件之间的耦合就越小，但是需要解决的组件协同问题也越多，实现数据的一致性也越困难。我们在系统设计时，要避免过度设计，把握技术方案的核心目的，在这个基础上进行针对性设计。

对于这一课时的内容，你可以思考下当前的项目中是如何处理重复消息的，有没有考虑消息处理的幂等性？欢迎留言分享。

精选评论

****伟：**

> 前公司就是为了分布式而分布式，设计与技术不到位，到生产上爆了不少问题

****老王：**

> 如果消息消费完毕了，还没来得及记录唯一id，这时候记录唯一id失败了，那再次消费的时候还是会重复消费呀？这个怎么破？没理解到

讲师回复：

> 你既然想到了，如果记录唯一id失败，不能保证幂等，那么解决问题的关键是不是就是不让它失败？或者写成功以后再去消费？

****峰：**

> 深刻，受益匪浅