11-无消息丢失配置怎么实现？

你好，我是胡夕。今天我要和你分享的主题是：如何配置Kafka无消息丢失。

一直以来，很多人对于Kafka丢失消息这件事情都有着自己的理解，因而也就有着自己的解决之道。在讨论具体的应对方法之前，我觉得我们首先要明确，在Kafka的世界里什么才算是消息丢失，或者说Kafka在什么情况下能保证消息不丢失。这点非常关键，因为很多时候我们容易混淆责任的边界，如果搞不清楚事情由谁负责，自然也就不知道由谁来出解决方案了。

那Kafka到底在什么情况下才能保证消息不丢失呢？

一句话概括，Kafka只对“已提交”的消息（committed message）做有限度的持久化保证。

这句话里面有两个核心要素，我们一一来看。

第一个核心要素是“**已提交的消息**”。什么是已提交的消息？当Kafka的若干个Broker成功地接收到一条消息并写入到日志文件后，它们会告诉生产者程序这条消息已成功提交。此时，这条消息在Kafka看来就正式变为“已提交”消息了。

那为什么是若干个Broker呢？这取决于你对“已提交”的定义。你可以选择只要有一个Broker成功保存该消息就算是已提交，也可以是令所有Broker都成功保存该消息才算是已提交。不论哪种情况，Kafka只对已提交的消息做持久化保证这件事情是不变的。

第二个核心要素就是“**有限度的持久化保证**”，也就是说Kafka不可能保证在任何情况下都做到不丢失消息。举个极端点的例子，如果地球都不存在了，Kafka还能保存任何消息吗？显然不能！倘若这种情况下你依然还想要Kafka不丢消息，那么只能在别的星球部署Kafka Broker服务器了。

现在你应该能够稍微体会出这里的“有限度”的含义了吧，其实就是说Kafka不丢消息是有前提条件的。假如你的消息保存在N个Kafka Broker上，那么这个前提条件就是这N个Broker中至少有1个存活。只要这个条件成立，Kafka就能保证你的这条消息永远不会丢失。

总结一下，Kafka是能做到不丢失消息的，只不过这些消息必须是已提交的消息，而且还要满足一定的条件。当然，说明这件事并不是要为Kafka推卸责任，而是为了在出现该类问题时我们能够明确责任边界。

“消息丢失”案例

好了，理解了Kafka是怎样做到不丢失消息的，那接下来我带你复盘一下那些常见的“Kafka消息丢失”案例。注意，这里可是带引号的消息丢失哦，其实有些时候我们只是冤枉了Kafka而已。

案例1：生产者程序丢失数据

Producer程序丢失消息，这应该算是被抱怨最多的数据丢失场景了。我来描述一个场景：你写了一个Producer应用向Kafka发送消息，最后发现Kafka没有保存，于是大骂：“Kafka真烂，消息发送居然都能丢失，而且还不告诉我？！”如果你有过这样的经历，那么请先消消气，我们来分析下可能的原因。

目前Kafka Producer是异步发送消息的，也就是说如果你调用的是producer.send(msg)这个API，那么它通常会立即返回，但此时你不能认为消息发送已成功完成。

这种发送方式有个有趣的名字，叫“fire and forget”，翻译一下就是“发射后不管”。这个术语原本属于导弹制导领域，后来被借鉴到计算机领域中，它的意思是，执行完一个操作后不去管它的结果是否成功。调用producer.send(msg)就属于典型的“fire and forget”，因此如果出现消息丢失，我们是无法知晓的。这个发送方式挺不靠谱吧，不过有些公司真的就是在使用这个API发送消息。

如果用这个方式，可能会有哪些因素导致消息没有发送成功呢？其实原因有很多，例如网络抖动，导致消息压根就没有发送到Broker端；或者消息本身不合格导致Broker拒绝接收（比如消息太大了，超过了Broker的承受能力）等。这么来看，让Kafka“背锅”就有点冤枉它了。就像前面说过的，Kafka不认为消息是已提交的，因此也就没有Kafka丢失消息这一说了。

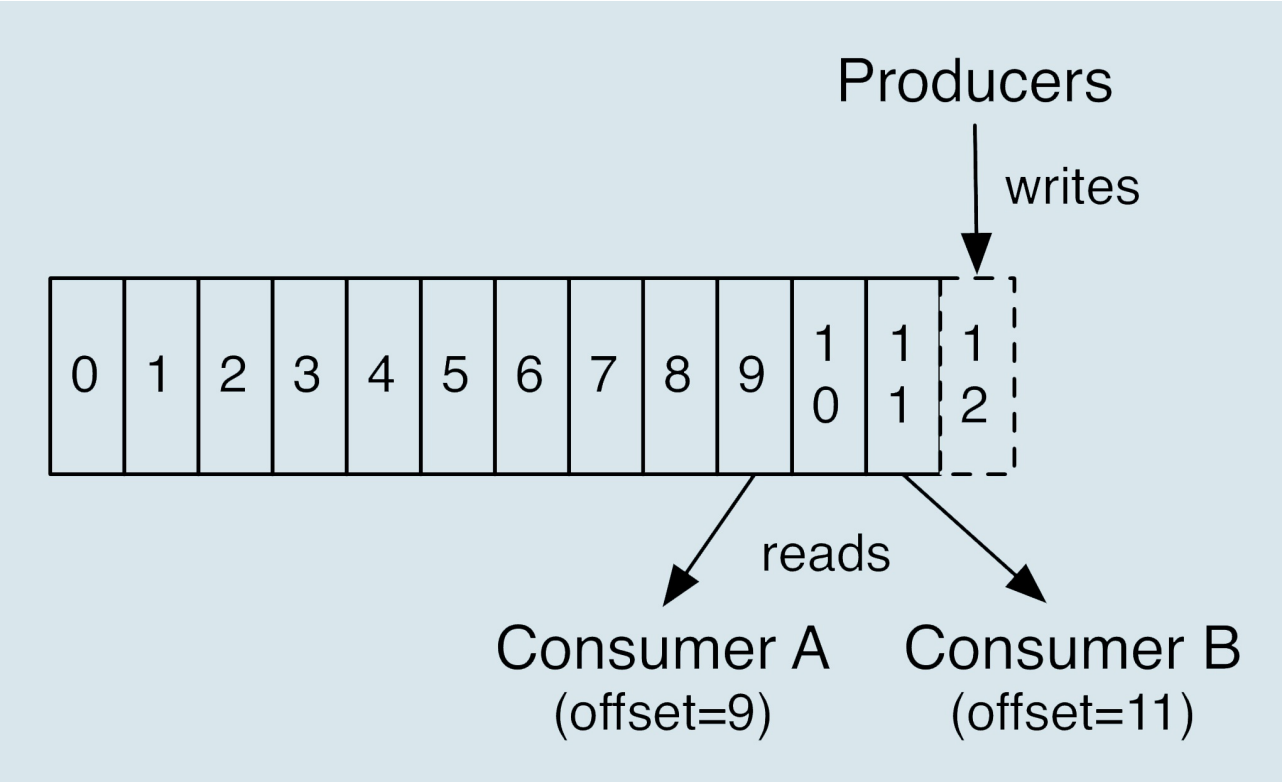
不过，就算不是Kafka的“锅”，我们也要解决这个问题吧。实际上，解决此问题的方法非常简单：**Producer永远要使用带有回调通知的发送API，也就是说不要使用producer.send(msg)，而要使用producer.send(msg, callback)**。不要小瞧这里的callback（回调），它能准确地告诉你消息是否真的提交成功了。一旦出现消息提交失败的情况，你就可以有针对性地进行处理。

举例来说，如果是因为那些瞬时错误，那么仅仅让Producer重试就可以了；如果是消息不合格造成的，那么可以调整消息格式后再次发送。总之，处理发送失败的责任在Producer端而非Broker端。

你可能会问，发送失败真的没可能是由Broker端的问题造成的吗？当然可能！如果你所有的Broker都宕机了，那么无论Producer端怎么重试都会失败的，此时你要做的是赶快处理Broker端的问题。但之前说的核心论据在这里依然是成立的：Kafka依然不认为这条消息属于已提交消息，故对它不做任何持久化保证。

案例2：消费者程序丢失数据

Consumer端丢失数据主要体现在Consumer端要消费的消息不见了。Consumer程序有个“位移”的概念，表示的是这个Consumer当前消费到的Topic分区的位置。下面这张图来自于官网，它清晰地展示了Consumer端的位移数据。



比如对于Consumer A而言，它当前的位移值就是9；Consumer B的位移值是11。

这里的“位移”类似于我们看书时使用的书签，它会标记我们当前阅读了多少页，下次翻书的时候我们能直接跳到书签页继续阅读。

正确使用书签有两个步骤：第一步是读书，第二步是更新书签页。如果这两步的顺序颠倒了，就可能出现这样的场景：当前的书签页是第90页，我先将书签放到第100页上，之后开始读书。当阅读到第95页时，我临时有事中止了阅读。那么问题来了，当我下次直接跳到书签页阅读时，我就丢失了第96～99页的内容，即这些消息就丢失了。

同理，Kafka中Consumer端的消息丢失就是这么一回事。要对抗这种消息丢失，办法很简单：**维持先消费消息（阅读），再更新位移（书签）的顺序**即可。这样就能最大限度地保证消息不丢失。

当然，这种处理方式可能带来的问题是消息的重复处理，类似于同一页书被读了很多遍，但这不属于消息丢失的情形。在专栏后面的内容中，我会跟你分享如何应对重复消费的问题。

除了上面所说的场景，其实还存在一种比较隐蔽的消息丢失场景。

我们依然以看书为例。假设你花钱从网上租借了一本共有10章内容的电子书，该电子书的有效阅读时间是1天，过期后该电子书就无法打开，但如果在1天之内你完成阅读就退还租金。

为了加快阅读速度，你把书中的10个章节分别委托给你的10个朋友，请他们帮你阅读，并拜托他们告诉你主旨大意。当电子书临近过期时，这10个人告诉你说他们读完了自己所负责的那个章节的内容，于是你放心地把该书还了回去。不料，在这10个人向你描述主旨大意时，你突然发现有一个人对你撒了谎，他并没有看完他负责的那个章节。那么很显然，你无法知道那一章的内容了。

对于Kafka而言，这就好比Consumer程序从Kafka获取到消息后开启了多个线程异步处理消息，而Consumer程序自动地向前更新位移。假如其中某个线程运行失败了，它负责的消息没有被成功处理，但位移已经被更新了，因此这条消息对于Consumer而言实际上是丢失了。

这里的关键在于Consumer自动提交位移，与你没有确认书籍内容被全部读完就将书归还类似，你没有真正地确认消息是否真的被消费就“盲目”地更新了位移。

这个问题的解决方案也很简单：**如果是多线程异步处理消费消息，Consumer程序不要开启自动提交位移，而是要应用程序手动提交位移**。在这里我要提醒你一下，单个Consumer程序使用多线程来消费消息说起来容易，写成代码却异常困难，因为你很难正确地处理位移的更新，也就是说避免无消费消息丢失很简单，但极易出现消息被消费了多次的情况。

最佳实践

看完这两个案例之后，我来分享一下Kafka无消息丢失的配置，每一个其实都能对应上面提到的问题。

1. 不要使用producer.send(msg)，而要使用producer.send(msg, callback)。记住，一定要使用带有回调通知的send方法。
2. 设置acks = all。acks是Producer的一个参数，代表了你对“已提交”消息的定义。如果设置成all，则表明所有副本Broker都要接收到消息，该消息才算是“已提交”。这是最高等级的“已提交”定义。
3. 设置retries为一个较大的值。这里的retries同样是Producer的参数，对应前面提到的Producer自动重试。当出现网络的瞬时抖动时，消息发送可能会失败，此时配置了retries > 0的Producer能够自动重试消息发送，避免消息丢失。

4. 设置unclean.leader.election.enable = false。这是Broker端的参数，它控制的是哪些Broker有资格竞选分区的Leader。如果一个Broker落后原先的Leader太多，那么它一旦成为新的Leader，必然会造成消息的丢失。故一般都要将该参数设置成false，即不允许这种情况的发生。
5. 设置replication.factor >= 3。这也是Broker端的参数。其实这里想表述的是，最好将消息多保存几份，毕竟目前防止消息丢失的主要机制就是冗余。
6. 设置min.insync.replicas > 1。这依然是Broker端参数，控制的是消息至少要被写入到多少个副本才算是“已提交”。设置成大于1可以提升消息持久性。在实际环境中千万不要使用默认值1。
7. 确保replication.factor > min.insync.replicas。如果两者相等，那么只要有一个副本挂机，整个分区就无法正常工作了。我们不仅要改善消息的持久性，防止数据丢失，还要在不降低可用性的基础上完成。推荐设置成replication.factor = min.insync.replicas + 1。
8. 确保消息消费完成再提交。Consumer端有个参数enable.auto.commit，最好把它设置成false，并采用手动提交位移的方式。就像前面说的，这对于单Consumer多线程处理的场景而言是至关重要的。

小结

今天，我们讨论了Kafka无消息丢失的方方面面。我们先从什么是消息丢失开始说起，明确了Kafka持久化保证的责任边界，随后以这个规则为标尺衡量了一些常见的数据丢失场景，最后通过分析这些场景，我给出了Kafka无消息丢失的“最佳实践”。总结起来，我希望你今天能有两个收获：

- 明确Kafka持久化保证的含义和限定条件。
- 熟练配置Kafka无消息丢失参数。

开放讨论

其实，Kafka还有一种特别隐秘的消息丢失场景：增加主题分区。当增加主题分区后，在某段“不凑巧”的时间间隔后，Producer先于Consumer感知到新增加的分区，而Consumer设置的是“从最新位移处”开始读取消息，因此在Consumer感知到新分区前，Producer发送的这些消息就全部“丢失”了，或者说Consumer无法读取到这些消息。严格来说这是Kafka设计上的一个小缺陷，你有什么解决的办法吗？

欢迎写下你的思考和答案，我们一起讨论。如果你觉得有所收获，也欢迎把文章分享给你的朋友。

精选留言：

- 阳明 2019-06-27 00:13:08
  总结里的的第二条ack=all和第六条的说明是不是有冲突 [6赞]

  作者回复2019-06-27 08:41:31
  其实不冲突。如果ISR中只有1个副本了，acks=all也就相当于acks=1了，引入min.insync.replicas的目的就是为了做一个下限的限制：不能只满足于ISR全部写入，还要保证ISR中的写入个数不少于min.insync.replicas。

- 明翼 2019-06-27 07:25:18
  这个问题我想个办法就是程序停止再增加分区，如果不能停止那就找个通知机制了。请教一个问题min.insync.replicas这个参数如果设置成3，假设副本数设置为4，那岂不是只支持一台broker坏掉的情况？本来支持三台坏掉的，老师我理解的对不对 [2赞]

  作者回复2019-06-27 08:55:46
  嗯嗯，是的。本来就是为了更强的消息持久化保证，只能牺牲一点高可用性了~~

- cricket1981 2019-06-27 06:54:35
  consumer改用"从最早位置"读解决新加分区造成的问题 [2赞]

- 空知 2019-06-27 10:56:47
  老师问下
  第7条 一个副本挂掉 整个分区不能用了 是因为每次都必须保证可用副本个数 必须跟提交时候一致 才可以正常使用,又没有冗余副本导致的嘛? [1赞]

- 没事走两步 2019-06-27 09:45:16
  如果consumer改用"从最早位置"读解决新加分区造成的问题，那会不会导致旧的分区里的已被消费过的消息重新全部被消费一次 [1赞]

- QQ怪 2019-06-27 23:12:09
  不知道是不是可以这样，生产者感知到了有新分区加入立即通知broke端下的消费者不能消费消息，直到消费端都感应到了加入的新分区之后，生产者和消费者才继续工作

- 光辉 2019-06-27 22:28:06
  老师，你好！
  producer.send(msg, callback)中callback主要用来做什么？可以用来重新发送数据么？如果可以的话，跟producer的配置retries是不是功能重复了

- 光辉 2019-06-27 22:23:07
  producer.send(msg, callback)中callback主要用来做什么？ 可以用来重发数据么，如果可以的话，跟producer的retire

- z.l 2019-06-27 22:14:37
  另外想请教下，单个 Consumer 程序使用多线程来消费消息的情况下，应该怎样自动提交offest？以java客户端为例，我把消息放到一个线程池中异步处理，此时consumer也不知道线程池中的任务是否执行成功，如果用future+同步提交又会阻塞consumer线程。所以是不是用多consumer线程同步消费+同步提交的方式比较合理？？

- z.l 2019-06-27 21:50:09

在“producer.send(msg, callback)的callback方法中重试，和设置retries参数重试，会不会冲突？2个都设置以哪个为准？

- guoyinbo2019 2019-06-27 21:31:23
胡老师，我有以下两个疑问
一、关于最佳实践
2、设置ack=all，这里的all指的任一时刻下所有存活的follower 么?比如开始follower=3，当有一台副本挂掉是，ack=all意味着follower=2
5、 replication.factor >= 3，设置副本个数吧，如果由于副本所在broker挂了，这个参数检测会自动减少么？
例如：设置replication.factor = 3，有3台broker，若有一台broker挂了，kafka会如何处理，会按照replication.factor = 2进行同步数据么， 还是当发现副本数< replication.factor的时候就不能正常工作， 还是其他什么策略呢？
6、min.insync.replicas > 1 这个会和上面参数矛盾么
以上几个 参数都参与的“已提交”的定义，当参数不能全部满足的时候，这个“已提交”是怎么定义的呢，
各参数在“已提交”定义中是否有优先级呢，如有，优先级是什么呢？
二、关于单consumer，多线程消费问题
对于老师说的“避免无消费消息丢失很简单，但极易出现消息被消费了多次的情况。”有些不理解
我认为单consumer，多线程消费时候容易产生消息丢失
例如一个consumer2个线程，thread1消费offset=1 消息，thread2消费offset=2 消息，thread2很快处理完了消息并成功提交了offset, 但是thread1由于某种原因处理失败了，此时offset=1 的消息对于consumer来说就丢了。
不知道理解的对不对，还请老师解答。

- Geek_986289 2019-06-27 18:45:11
设置 acks = all。表明所有副本 Broker 都要接收到消息，该消息才算是“已提交”。如果所有的Broker都要收到消息才能算作已提交，会不会对系统的吞吐量影响很大？另外这里的副本指的是不是仅仅是ISR?

- Xiao 2019-06-27 18:19:08
胡老师，broker在持久化的时候理论上也存在数据丢失的情况吧。

- 永光 2019-06-27 16:24:44
看了评论区回答还是不太理解，第二条ack=all与第六条min.insync.replicas 怎样协调工作的，总感觉是有冲突的。
问题是：
第二条的“已提交”和第六条的“已提交”是同一个意思吗？如果是同一个意思，那定义为什么不一样呀？

- 振超 2019-06-27 15:10:59
设置从最新 offset 处开始消费，实际上默认就接受允许丢失一部分消息，这应该不算是缺陷

- 张庆 2019-06-27 14:36:27
胡夕大拿 您好，关于消费者数据丢失的第一种情况，我不是很明白，在什么情况下读书和更新书签的顺序是颠倒的啊？我理解的如果设置为自动提交了，就是定时间隔的位移提交，在这个间隔时间里面，如果正在消费数据，此时宕机了，还没有提交位移，如果重启了，可能会出现重复消费。不理解消息丢失的场景是什么样子的？您能详细解释一下吗？

- lmtoo 2019-06-27 13:31:52
  最后一个问题，难道新增分区之后，producer先感知并发送数据，消费者后感知，消费者的offset会定位到新分区的最后一条消息？消费者没有提交offset怎么会从最后一条开始的呢？

- Icedmaze 2019-06-27 10:34:09
  目前想到的方法有两种
  一种是在消费端初始化的时候配置“从新位移处开始消费”，当各个分区记录了位移信息后，再将配置改为“从最早位移处开始消费”策略，利用已产生了标志位的分区不受消费策略影响的特性，保证在之后的业务中，新增分区都不会产生“丢失”问题。
  第二种方法是当新增分区后，利用消费端会产生rebalance事件的特性，可以在rebalance的时候判断该分区有无消费位移，如果无位移位，则强制将位移位设置为0从头开始消费。（该方法纯属猜测）

- 趙衍 2019-06-27 10:29:57
  老师可以贴出官方社区对这个问题的解答吗，我想更深入地了解一下这个问题

- 刚子 2019-06-27 10:14:41
  broker的配置、producer和consumer的配置如此多和复杂，不容易记忆。需要后期根据业务需求进行相应调整，是否有模版的配置参数或者自感应的功能，可以帮助开发和运维快速正确的使用Kafka