6.1 可靠性保证

Kafka可以保证分区消息的顺序。

只有当消息被写入分区的所有同步副本时(但不一定要写入磁盘)，它才被认为是“ 已 提交”的。

只要还有一个副本是活跃的，那么已经提交的消息就不会丢失 。

消费者只能读取已经提交的悄息。

6.2 复制

Kafka的主题被分为多个分区，分区是基本的数据块。分区存储在单个磁盘上，Kafka可以 保证分区里的事件是有序的，分区可以在线(可用)，也可以离线(不可用) 。每个分区可 以有多个副本，其中 一个副本是首领。所有的事件都直接发送给首领副本，或者直接从首 领副本读取事件。其他副本只需要与首领保持同步，并及时复制最新的事件。当首领副本 不可用时，其中一个同步副本将成为新首领。

分区首领是同步副本，而对于跟随者副本来说，它需要满足以下条件才能被认为是同步的。

与 Zookeeper 之间有 一 个活跃的会话，也就是说，它在过去的“(可配 置)内向 Zookeeper 发送过心跳。

- 在过去的 10s 内(可配置)从首领那里获取过消息。
- 在过去的 10s 内从首领那里获取过最新的消息。光从首领那里获取消息是不够的，它还

必须是儿乎零延迟的。

如果跟随者副本不能满足以上任何一点，比如与 Zookeeper断开连接，或者不再获取新消 息，或者获取消息滞后了 10s 以上，那么它就被认为是不同步的。 一个不同步的副本通过 与 Zookeeper重新建立连接，并从首领那里获取最新消息，可以重新变成同步的。这个过 程在网络出现临时问题并很快得到修复的情况下会很快完成，但如果 broker发生崩愤就需 要较长的时间。

（第五章内容）

6. 3 broker配置

应用在 broker级别，用于控制所有主题的行为，

应用在主题级别，用于控制个别主题的行为。

6.3.1 复制系数

主 题级别的配置参数是 replication.factor，而在 broker级别则可以通过 default. replication.factor 来配置自动创建的主题。

6.3.2 不完全的首领选举

unclean.leader.election 只能是 broker级别

在首领不可用时其他副本都是不同步的，面临两难选择
·如果不同步的副本不能被提升为新首领，那么分区在旧首领(最后一个同步副本)恢复 之前是不可用的。
如果不同步的副本可以被提升为新首领，那么在这个副本变为不同步之后写入旧首领的
消息、会全部丢失，导致数据不一致

银行等对数据质量和数据 一致性要求较高的系统，会禁用这种情况，在对可用性要求较高的系统里，比如实 时点击 流分析系统， 一般会启用不完全的首领选举。

6.3.3 最少同步副本

min.insync.replicas
如果同步副本少于这个值 broker就会停止接受生产者的请求。尝试发送数据的生产 者会收到 NotEnoughReplicasExcepti.on异常。

6.4 在可靠的系统里使用生产者

根据可靠性需求配置恰当的 acks值。
在参数配置和代码里正确处理错误。

6.4.1 发送确认

acks=0 意味着如果生产者能够通过网络把消息发送出去，那么就认为消息已成功 写入 Kafka。
acks=1 意味若首领在收到消息并把它写入到分区数据文件(不一定同步到磁盘上)时 会返回确认或错误响应。
acks=all 意味着首领在返回确认或错误响应之前，会等待所有同步副本都收到消息。

6.4.2 配置生产者的重试参数

生产者需要处理的错误包括两部分：一部分是生产者可以自动处理的错误，还有一部分是
需要开发者手动处理的错误。
生产者向 broker发送消息时， broker可以返回 一个成功晌应码或者一个错误响应码。错民晌应码可以 分为两种， 一种是在重试之后可以解决的，还有一种是无法通过重试解决的。例如，如果 broker返回的是 LEADER_NOT_AVAILABLE 错误，生产者可以尝试重新发送消息。如果 broker返回的是 INVALID_CONFIG错误，即使通过重试 也无能改变配置选项，所以这样的重试是没有意义的。

6.4.3 额外的错误处理

不可重试的 broker 错误，例如消息大小错误、认证错误等 3
在消息发送之前发生的错误，例如序列化错误:
在生产者达到重试次数上限时或者在消息占用的内存达到上限时发生的错误。
我们在第 3 章讨论了如何为同步发送消息和异步发送悄息编写错误处理器。这些错误处理
器的代码逻辑与具体的应用程序及其目标有关。丢弃“不合陆的消息”?把错误记录下
来?把这些消息保存在本地磁盘上?恒|调另 一个应用程序?具体使用哪一种逻辑要根据具 体的架构来决定。只要记住，如果错误处理只是为了重试发送消息，那么最好还是使用生 产者内置的重试机制。

6.5 在可靠的系统里使用消费者

6.5.1 消费者的可靠性配置

4个非常重要的配置参数。
第一个是group.id 如果两个消费者具有相同的 9「oup.Id， 并且订阅了同一个主题，那么每个消费者会分到主题分区的一个子集， 也就是 说它们只能读 到所有消息的 一 个子集
第二个是auto.offset.reset 在没有偏移量可提交时/请求的偏移量在 broker上不存在时，=earliest 消费者会 从分区的开始位置读取数据 =latest 消费者从分区的末尾开始读取数据
第三个是enable.auto.commit 基于任务调度自动提交偏移量
第四个配置是 auto.commit.interval.ms 选择了自动提交偏移盘，可以通过该参数配置提交的频度， 默认值是每 5秒钟提交一次。 一般来说，频繁提交会增加

额外的开销，但也会降低重复处理消息的概率。

6.5.2 显式提交偏移量

如果希望能够更多地控 制偏移量提交 的时间点，那么就要仔细想想该如何提交偏移量了 ——要么是为了减少重复 处理消息 ，要么是因为把消息处理逻辑放在了轮询之外。

1. 总是在处理完事件后再提交偏移量:所有 的处理都是在轮询里完成，并且不需要在轮询之间维护状态

2.提交频度是性能和重复消息数量之间的权衡

3. 确保对提交的偏移量心里有数

4.再均衡:(第四章)一般要在分区被撤销之前提交偏移量，并在分配到新分区时清理之前的状态。

5. 消费者可能需要重试

第一种模式，在遇到可重试错误时，提交最后一个处理成功的偏移量，然后把还没有处理

好的消息保存到缓冲区里(这样下一个轮询就不会把它们覆盖掉)，调用消费者的 pause() 方怯来确保其他的轮询不会返回数据(不需要担心在重试时缓冲区隘出)，在保持轮询的同时尝试重新处理(关于为什么不能停止轮询，请参考第 4 章)。如果重试成功，或者重 试次数达到上限并决定放弃，那么把错误记录下来并丢弃消息，然后调用 resul'le()方能让 消费者继续从轮询里获取新数据。

第二种模式，在遇到可重试错误时，把错误写入一个独立的主题，然后继续。一个独立的 消费者群组负责从该主题上读取错误消息，并进行重试，或者使用其中的一个消费者同时从该主题上读取错误消息并进行重试，不过在重试时需要暂停该主题。这种模式有点像其 他消息系统里的 dead-letter-queue。

6. 消费者可能需要维护状态

有时候你希望在多个轮询之间维护状态，例如，你想计算消息的移动平均数，希望 在首次 轮询之后计算平均数，然后在后续的轮询中更新这个结果。如果进程重启 ，你不仅需要从 上一个偏移量开始处理数据，还要恢复移动平均数。有一种办怯是在提交偏移量的同时把 最近计算的平均数写到一个“结果”主题上。消费者线程在重新启动之后，它就可以拿到 最近的平均数并接着计算。不过这并不能完全地解决问题，因为 Kafka并没有提供事务支 持。消费者有可能在写入平均数之后来不及提交偏移量就崩溃了，或者反过来也一样。这是一个很复杂的问题，你不应该尝试自己去解决这个问题，建议尝试一下 Kafk:aStreams这 个类库 ，它为聚合、连接、时间窗和其他复杂的分析提供了高级的 DSLAPI。

7. 长时间处理：
有时候 处理数据需要很长时间:你可能会从发生阻塞的外部系统获取信息，或者把数据 写到外部系统，或者进行一个非常复杂的计算。要记住，暂停轮i询的时间不能超过几秒 钟。即使不想获取更多的数据，也要保持轮询，这样客户端才能往 broker发送心跳。在 这种情况下， 一种常见的做桂是使用一个线程地来处理数据，因为使用多个线程可以进 行并行处理，从而加快处理速度 。在把数据移交给线程地去处理之后，你就可以暂停消 费者，然后保持轮询，但不获取新数据，直到工作线程处理完成。在工作线程处理完成 之后 ，可以让消费者继续获取新数据。因为消费者一直保持轮询，心跳会正常发送，就 不会发生再均衡。

8. 仅一次传递：方法一：写到一个支持唯一键的系统里(事等性写入) 方法二：使用关系型数据 库,我们把消 息和偏移量放在同一个事务里 ，这样它们就能保持同步。在消费者启动时，它会获取最近 处理过 的消息偏移量 ，然后调用 seek() 方也从该偏移量位置继续读取数据。第四章有相应例子

6.6 验证系统可靠性

6.6.1 配置验证

org.apache.kafka.tools 包 里的 VerifiableProducer 和 VerifiableConsumer 这两个类，其思想是， VerifiableProducer 生成一系列消息，这些消息包含从 1 到你指定的某个数 字。你可以使用与生产者相同的方式来配置VerifiableProducer，比如配置相同的 acks、 重试次数和消息生成速度。在运行VerifiableProducer 时，它会把每个消息是否成功发送 到 broker的结果打印出来。
VerifiableConsumer 执行的是另一个检查——它读取事件(由 VerifiableProducer 生成)并按顺序打印出这些事件。它也会打印出已提交的偏移量和再 均衡的相关信息。
然后你从中选择一个场景，启动VerifiableProducer和 VerifiableConsumer 并开始测试这 个场景，例如，停掉正在接收消息的分区首领。如果期望在一个短暂的暂停之后状态恢复 正常并且没有任何数据丢失，那么只要确保生产者生成的数据个数与消费者读取的数据个 数是匹配的就可以了。
Kafka 的代码库里包含了大量测试用例。它们大部分都遵循相同的 准则 —— 使用 VerifiableConsumer 和 VerifiableConsumer 来确保迭代的版本能够正常工作。

6.6.2 应用程序验证

考虑一下情况:

客户端从服务器断开连接(系统管理员可以帮忙模拟网络故障);

首领选举

依次重启 broker;

依次重启消费者;

依次重启生产者。

6.6.3 在生产环境监控可靠性

我们将会在第 9章详细介绍如何监控 Kafka集群，不过除了监控 集群的健康状况之外，监控客户端和数据流也是很重要的。

1.Kafka 的 Java 客户端包含了 JMX 度量指标,除了最重要的两个可靠性指标是消息的已汀error-rate 和 retry-rate，还要监控生产者日 志——发送消息的错误日志被设为 WARN 级别

2.消费者 最重要的指标是 consumer-lag，该指标表 明 了消费者的处理速度与最近 提交到分区里的偏移量之间还有多少差距。理想情况下，该指标总是为 0，消费者总能读 到最新的消息。不过在实际当中，因为 poll() 方邑会返回很多消息，消费者在获取更多数 据之前需要花一些时间来处理它们，所以该指标会有些波动。关键是要确保消费者最终会 赶上去 ，而不是越落越远。因为该指标会正常波动，所以在告警系统里配置该指标有一定 难度。

3.监控数据流是为了确保所有生成的数据会被及时地读取(你的需求决定了“及时”的具体 含义)。 为了确保数据能够被及时读取，你需要知道数据是什么时候生成的。 0.10.0 版本 的 Kafka在消息里增加了时间戳，表明了消息的生成时间。如果你使用的是更早版本的 客户端 ，我们建议自己在消息里加入时间戳、应用程序的名字和机器名，这样有助于将 来诊断问题。

为了确保所有消息能够在合理的时间内被读取，应用程序需要记录生成消息的数量(一般 用每秒多少个消息来表示)，而消费者需要记录已读取消息的数量(也用每秒多少个悄息 来表示) 以及消息生成时间(生成消息的时间)到当前时间(读取消息的时间)之间的时

间差。然后，你需要使用工具来比较生产者和消费者记录的消息数量(为了 确保没有丢失 消息)，确保这两者之间的时间 差不会超出我们允许的范围 。为了做到更好 的监控， 我们 可以增加一个“监控消费者”，这个消费者订阅一个特别的主题，它只进行消息的计数操 作，并把数值与生成的消息、数量进行对比，这样我们就可以在没有消费者的情况下仍然能 够准确地监控生产者。

6.7 总结

可靠性并不只是 Kafka单方面的事情。我们应该从整个系统 层面来考虑可靠性问题，包括应用程序的架构、生产者和悄费者 API的使用方式、生产 者和消费者的配置、主题的配置以及 broker 的配置。系统的可靠性需要在许多方面作出权衡，比如复杂性、性能、可用性和磁盘空间的使用。