Kafka是一个分布式的流媒体平台，因为能应用到的比较多，比如消息队列，日志收集

1.**高吞吐量**：能处理TB级的数据

消息持久化，将消息传入**硬盘**里，硬盘便宜，**空间大，处理海量数据**的前提（要先有海量数据）虽在硬盘中，但并不是说在硬盘的读取就一定慢了，**kafka的硬盘读取仍然很快**，他是**通过硬盘的顺序读取Segment 数据文件**，且消息消费后不会删除（删除策略是针对过期的 Segment 文件，而非随机读取，随机慢，但顺序读取甚至**能快过内存。**

2.**高扩展**：集群不够了，**加一台服务器**

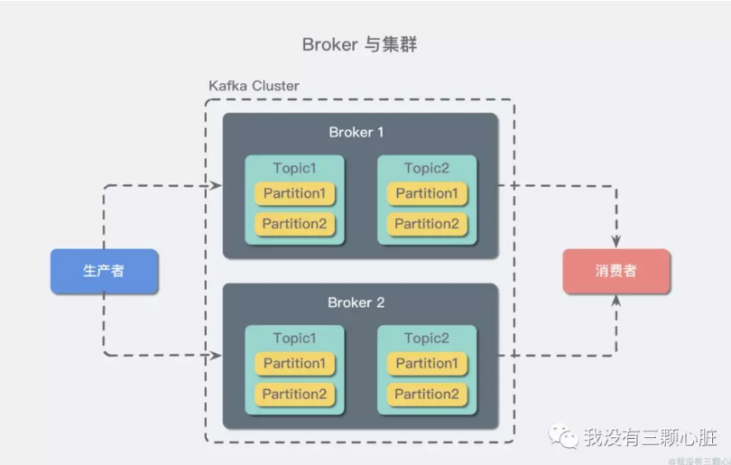
3.**高可靠**：通过**分布式**保证，可以**做集群部署**，一台服务器挂了还有别的服务器

**高可用：副本机制**。每个partition的数据都会同步到其他机器上，形成自己的副本。然后所有的副本会选一个leader出来，那么生产者消费者都和这个leader打交道，其他的副本就是follower。写的时候，leader会把数据同步到所有follower上面去，读的时候直接从leader上面读取即可

如果某个broker宕机了，那个broker的partition在其他机器上有副本，如果这上面有某个partition的leader，那么此时会重新选举出一个现代leader出来，继续读写这个新的leader即可

**写消息：**写数据的时候，生产者就写给leader，然后leader将数据落到磁盘上之后，然后其他follower自己从leader来pull数据。所有follower同步好数据就会发ack确认给leader，leader收到所有的follower的ack之后，就会返回 写成功 的消息给消息生产者。（这只是一种模式，可以调整）。

**读(消费)数据**:只会从leader进行消费。但是只有一个消息已经被所有follower都同步成功返回ack的时候，这个消息才会被消费者读到



**Broker**：消息中间件处理节点，一个Kafka服务器，把生产出的消息放到磁盘然后会让消费者pull(因为push目标是尽可能快地传输数据，会和消费者消费速度不适应造成来不及消费--网络阻塞，消费者pull允许消费者根据自己消费数据能力用适当的速率消费数据)。

**Topic**(只是逻辑上的一个东西)：主题，一类消息(代表发消息的位置和消息类别，比如点赞、关注的主题)，消息队列MQ的实现方式有两种：

1. 一种是**点对点**，**阻塞队列**就是点对点，每个数据被一个消费者消费；
2. **发布订阅模式**：生产者将消息放到每一个位置，可以有**很多消费者同时订阅消息**，这个消息可以被多个消费者读到（同时或先后）

**kafka**用的第二种发布订阅消息，**生产者将消息发送到的位置就是topic**

**Partition**：topic的物理上的分区，一个topic分多个partition，每个partition是一个有序队列。可以采用**多线程**的方式，同时向多个分区写数据，增强并发能力，每个分区按从前往后写入数据。比如加了A再加B，就会先消费A --> 这样也保证了数据的可靠

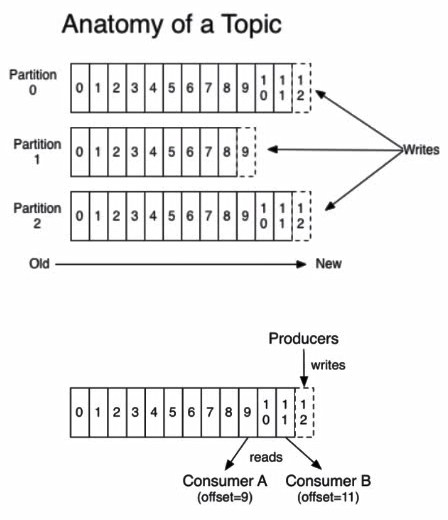
序号从0开始，序号最大值为partitions数量减1。

**Segment：**partition物理上由多个segment组成，每个partition(目录)相当于一个巨型文件被平均分配到多个大小相等segment(段)数据文件中。但每个段segment file消息数量不一定相等，这种特性方便old segment file快速被删除。默认保留7天的数据。

**Segment File = index file(索引文件".index") + data file(数据文件".log" )总是成对出现**

索引文件用**稀疏索引**存储的方式，每隔一定字节的数据建立一条索引，它减少了索引文件大小，使得能够把 index 映射到内存，降低了查询时的磁盘 IO 开销，同时也并没有给查询带来太多的时间消耗，所以index 文件并不是从0开始，也不是每次递增1的

**Offset**：**消息在分区内的索引，消息的序列号，用于分区里标识消息**



**Cluster**(集群)：多个Broker组成，某个Broker会被选为集群控制器来管理集群，监控故障等。一个分区也可以被复制到多个Broker，这样一个Broker有故障可以用其他Broker

**多集群：**用MirrorMaker(一个消费者和生产者)工具实现集群直接消息生产和消费。

好处：1.基于数据和安全的隔离 2.多数据中心(容灾)

**Zookeeper**(独立引用)：用于管理集群,比如选出分区的leader故障了,选个broker作为控制器

**Retention(日志保留)：**可以对某个主题单独设置日志保留时间/日志大小，老消息就被删除

**Replica副本：**因为**kafka是分布式**消息引擎，**将数据存放多份 -> 数据更可靠**

**Leader Replica**(主副本): 从分区**读数据**时，主副本可以做出响应

**Follower Replica**(从副本)只用来**存数据**，主副本挂掉时候集群会从副本中选一个作为leader

**怎么保证顺序消费**

生产者在写的时候，其实可以指定一个 key，比如说我们指定了某个订单 id 作为 key，那么这个订单相关的数据，一定会被分发到同一个 partition 中去，而且这个 partition 中的数据一定是有顺序的。  
消费者从 partition 中取出来数据的时候，也一定是有顺序的。到这里，顺序还是 ok 的，没有错乱。

**没顺序因为**：然后我们在消费者里可能会搞多个线程来并发处理消息。因为如果消费者是单线程消费处理，而处理比较耗时的话，比如处理一条消息耗时几十 ms，那么 1 秒钟只能处理几十条消息，这吞吐量太低了。而多个线程并发跑的话，顺序可能就乱掉了

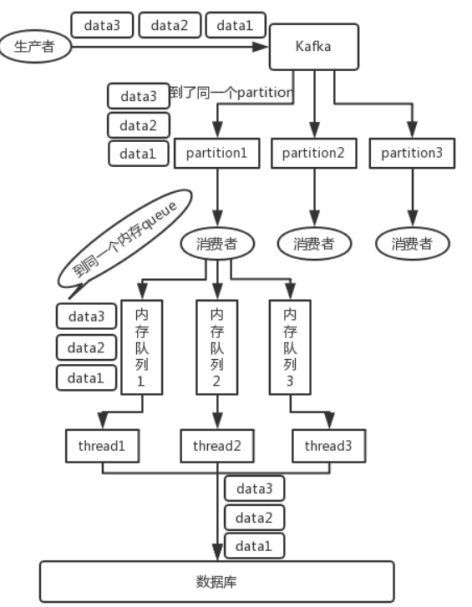
**保证顺序消费：**

**全局有序：**1.一个 topic只设置一个 partition，一个 consumer，内部单线程消费，一个Topic下的所有消息都需要按照生产顺序消费。但单线程吞吐量太低，一般不会用这个。

发送消息可指定(topic, partition, key) 3个参数，保证同一类消息只发送到同一个partition

**局部有序：**在发消息的时候指定Partition Key，Kafka再对key来Hash计算，根据计算结果决定放到哪个Partition。这样Partition Key相同的消息会放在同一个Partition。此时，Partition的数量仍然可以设置多个，提升Topic的整体吞吐量。

在不增加partition数量的情况下想**提高消费速度**，对于每个partition的消费者可以有 N 个内存 queue，每个queue用一个线程处理，然后对消息再hash他们的唯一标识（例如订单orderId）到不同的queue，也就是用不同线程来处理，这样就是多个消费者线程并发处理消息（依旧可以保证局部有序）。



4.如果一个发送失败了，后边的就不能继续发了，不然重发的那个肯定就乱序了

5.生产者在收到发送成功的反馈之前，不能发下一条数据(但我感觉生产者是一个流，阻塞正产者感觉业务不可行，怎么会因为一条消息发出去没收到反馈，就阻塞生产者)

也就是用同步发送模式：发出消息后，必须阻塞等待收到通知后，才发送下一条消息

而异步发送模式：一直往缓冲区写，然后一把写到队列中去

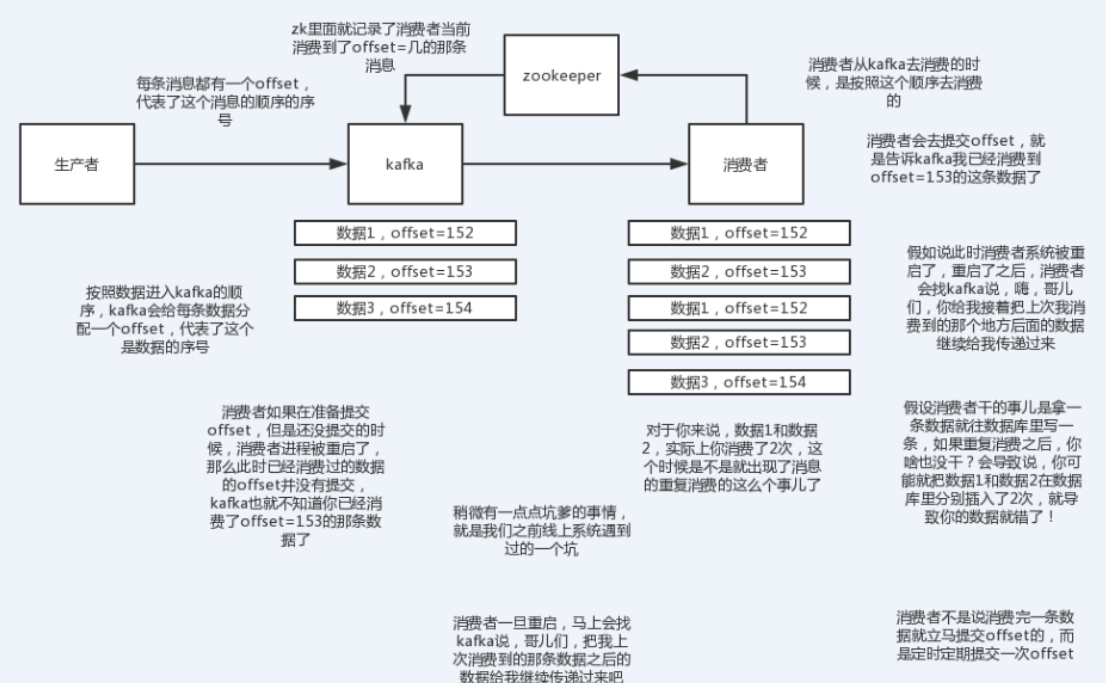
两者都是各有利弊：  
同步发送模式虽然吞吐量小，但是发一条收到确认后再发下一条，既能保证不丢失消息，又能保证顺序

**保证消息不重复**

**为什么会重复消费消息：**每个消息写进的时候都有个offset代表序号，consumer消费后，每隔一段时间会把自己消费过的消息的offset提交一下，代表已经消费过了，下次重启的话就从上次消费到的offset开始消费。但如果重启不当，比如直接kill进程再重启，会导致consumer把有些消息处理了但没提交offset，这就导致了有些消费被再消费一次。

重复消费不要紧，但幂等性(一个请求来多次，对应数据/结构也不变)不能保证就不行

如果消费消息是直接把消息插入到数据库，那么重复消费就会在数据库插入多条重复数据



**解决重复**：1.插入数据库的时候先拿主键查一下，数据已经有了就只update。如果是Redis 用set

2.情景复杂点的可以让生产者发消息的时候加个全局唯一的id，消费的时候去Redis查 有没有消费过这个id，没消费过才消费

3.还可以用数据库的唯一键，之后如果插入重复数据就会报错

生产端重复发送：这个不重要，消费端消费之前从去重表中判重就可以

消费端重复消费：建去重表

**保证消息不丢失**

**Producer端解决方案：**

Producer是异步发消息，如果用的发后即焚的方式发，也就是调用Producer.send(msg)立即返回，那么因为没有回调可能会因为网络原因导致Broker没收到消息，这个时候就丢失了。

**1.更换调用方式**：

不用调用发后即焚的方式，用带回调通知函数的方法来发消息，也就是Producer.send(msg, callback), 这样一旦发现发送失败，就可以做针对性处理：

（1）网络抖动导致消息丢失，Producer就可以重试。

（2）消息大小不合格，可以调整Broker接收消息的阈值，超过设置值Broker不能接收，所以要衡量消息的大小，合理设置这两个配置项。max.message.bytes控制精度更细，可以针对每个Topic设置正常接收消息最大值，符合Broker承受范围再发送。

这两种方式可以保证最大限度消息能发送成功。

**2.ACK 确认机制**：

这个参数代表对"已提交"消息的定义。

需要把request.required.acks 设置成-1或者all，-1或者all表示有多少个副本Broker全部收到消息，才认为是消息提交成功的标识。

针对 **acks = -1/ all**, 有两种情况：

（1）如果数据发到Leader Partition，而且所有的ISR成员全部同步完数据，这个时候Leader Partition异常Crash掉，那么会选举新的Leader Partition，数据不会丢失，

（2）如果数据发送到 Leader Partition，部分ISR成员同步完成，这个时候Leader Partition 异常Crash， 剩下的Follower Partition都可能被选举成新的Leader Partition，会给 Producer 端发送失败标识，后续会重新发送数据，数据**可能会重复**

**3.重试次数 retries**：

这个参数表示Producer端发送消息的重试次数。

需要把retries 设置为>0的数，在 Kafka 2.4 版本中默认设置为Integer.MAX\_VALUE。

如果要保证消息的顺序性，就要配置

retries = Integer.MAX\_VALUE 和 max.in.flight.requests.per.connection = 1

这样 Producer 端就会一直进行重试直到Broker端返回 ACK 标识，同时只有一个连接向 Broker发送数据保证了消息的**顺序性**。

**4.重试时间 retry.backoff.ms：**

这个参数表示消息发送超时后两次重试之间的间隔时间，避免无效的频繁重试，默认值为100ms,  推荐设置为300ms。

**Broker端解决方案：**

Broker端通过异步批量刷盘的策略，先把数据存到PageCache，再异步刷盘，因为没有提供同步刷盘策略， 所以Kafka是通过多分区多副本的方式来最大限度的保证数据不丢失。

可以通过参数配合来保证：

如果ISR里的副本都挂了，可以设置broke里的unclean.leader.election.enable为true，也就是启用不在ISR里的副本作为最后选的leader，但是不在ISR里的副本备份数据比较慢，数据不全，所以会有数据丢失的问题。所以最好还是false

**所以还需要其他broker配置来保证：replication.factor >= 2和min.insync.replicas > 1**

**min.insync.replicas控制的是消息至少要被写入到多少个副本才算是“已提交”。前提是生产者把确认设置为“all”（或“-1”）的情况下。**

**在使用acks设置为all时，所有的同步副本（ISR）集合全部接收消息后，就认为是“已提交”，现在设置了min.insync.replicas，相当于增加了对“已提交”的确认条件，那就是还需保证消息至少要被写入了多少个副本。**

**1 unclean.leader.election.enable**：表示有哪些 Follower 可以有资格被选举为 Leader , 如果一个Follower的数据落后 Leader 太多，那么一旦它被选举为新的 Leader，数据就会丢失，所以我们要把他设置为成false，防止这个情况发生。

**2 replication.factor：**表示分区副本的个数。建议设置 replication.factor >=3, 这样如果 Leader 副本异常 Crash 掉，Follower 副本会被选举为新的 Leader 副本继续提供服务。

**3 min.insync.replicas**：表示消息至少要被写入成功到 ISR 多少个副本才算"已提交"，建议设置min.insync.replicas > 1, 这样才可以提升消息持久性，保证数据不丢失。

另外我们还需要确保一下 replication.factor > min.insync.replicas, 如果相等，只要有一个副本异常 Crash 掉，整个分区就无法正常工作了，因此推荐设置成： replication.factor = min.insync.replicas +1, 最大限度保证系统可用性。

**Consumer端解决方案：**

**过程**变成为：拉取数据、业务逻辑处理、提交消费 Offset 位移信息。

**参数**：1、enable.auto.commit，自动提交？。默认是true

2.auto.commit.interval.ms，如果 enable.auto.commit 设置为 true，消费者偏移量自动提交到 Kafka 的频率（以毫秒为单位），默认5s

**方案1**：enable.auto.commit=true 自动提交

poll 方法是先提交上一批消息的位移，再处理下一批消息，所以能保证不丢消息，但可能会重复消费消息。

**方案2：**enable.auto.commit=false 手动提交，使用同步提交方法commitSync()去提交Consumer offset

commitSync()方法是在处理完poll的所有消息之后调用。

缺点: 调用 commitSync() 时，Consumer 程序会处于阻塞状态

**方案3**：enable.auto.commit=false 手动提交，使用异步提交方法commitAsync() 去提交Consumer offset

调用 commitAsync() 之后，它会立即返回，不会阻塞，所以不会影响 Consumer 应用的TPS。

缺点：出现问题时它不会自动重试。异步提交的重试没有意义，提交失败后，自动重试时提交的位移值可能早就过期。

**方案4**：enable.auto.commit=false ，同时采用同步提交和异步提交两种方式

该方案既不影响 TPS，又支持自动重试，改善 Consumer 应用的高可用性。同时做到了消息处理的不丢不重。

还要设置参数 enable.auto.commit = false来采用手动提交位移的方式。

对于消费消息重复的情况，业务自己保证幂等性, 保证只成功消费一次就行。

**3端消息丢失原因**

**Producer丢消息原因：**

**1.网络原因：**由于网络抖动导致数据没发到Broker端。

**2.数据原因：**消息体太大超出Broker承受范围导致Broker拒收消息。

**3.可以通过配置来确认消息是否生产成功**

Producer 端发送数据有ACK机制，acks默认配置是1，级别是 at least once, 不能保证exactly once

**(1)acks = 0：**发送后就自认为发送成功，producer完全不管broker的处理结果，没有回调，如果发生网络抖动，Produce不会校验 ACK, 所以就丢了，而且没办法重试。不能保证消息成功发送，但是这种吞吐量最高。实现at most once

**(2)acks = 1：默认的值，**消息发了之后到Leader Parition接收成功表示发送成功，不会等待ISR其他的副本写入，只要Leader Partition存活就可以保证Leader Partition不丢数据，但如果 Leader Partition异常Crash掉了，Follower Partition还没同步完数据而且没有ACK，这个时候就会丢数据。一定程度保证了不丢失，也保证了吞吐量。但有种情况是消息成功写入，而这个时候由于网络问题producer没有收到写入成功的响应，producer就会开启重试的操作，直到网络恢复，消息就发送了多次。这就是at least once，所以默认的是at least once

**(3)acks = -1 或者 all：** 消息发送后leader要等待ISR(同步副本)里所有Follower Partition都确认收到消息才返回给producer确认通知, 可靠性最高, 这种**只要ISR有副本存活就肯定不会丢失**，但吞吐量最低。但也**不能保证不丢数据,** 比如当ISR只剩下 Leader Partition了, 这样就变成acks = 1的情况了。

**ISR:**Leader与Followers之间的同步时存在延时的，Leader挂掉后要从Followers中选出新的Leader，但是延时高的Follower去做Leader是不合理的，所以Kafka动态的维护着同步副本（ISR）集合，消息接收延时高的follwer会被踢出这个集合，只有这个集合的成员才有资格被选举为leader。ISR集合在发生更改时会持久保存到 ZooKeeper。 所以ISR里任何副本都有资格被选为leader。

**实现exactly once**：后面版本的Kafka有idempotent(幂等 ai ding p tent) producer机制，这个机制里同一消息可能被producer发送多次，但是在broker端只会写入一次，他给每一条消息编号去重，而且对kafka开销影响不大。在多分区的情况，用**事务**保证**原子性地**写入多个分区，也就是写入到多个分区的消息要么全部成功，要么全部回滚。

consumer端可能无法消费事务中所有消息，而且消息可能被删除，所以可能要自己管理offset的提交，不要自动提交

**Broker端持久化存储丢失消息场景：**

kafka通过多Partition(分区)多Replica(副本)机制已经可以最大限度的保证数据不丢失，**如果数据已经写入PageCache里但是还没来得及刷写到磁盘，此时如果所在 Broker 突然宕机挂掉或者停电，极端情况还是会造成数据丢失**

Broker集群接收到数据后会把数据持久化存到磁盘，为了提高吞吐量和性能，采用的是**异步批量刷盘**的策略，也就是按照一定的消息量和间隔时间刷盘。

步骤：先把数据存到PageCache里，再由操作系统根据自己的策略决定什么时候把Cache里的数据刷盘或者调用fsync命令强制刷盘。如果这个时候Broker宕机Crash掉，而且选举了一个落后Leader Partition很多的Follower Partition成为新的Leader Partition，那么落后的消息数据就会丢失。

因为Kafka没有提供同步刷盘的方式，所以从单个Broker来看是很有可能丢数据的。

**Consumer端丢消息原因**

可能使用的**自动提交Offset**方式：

1. 拉消息后如果**先提交 Offset，后处理消息**，如果这个时候处理消息的时候宕机，因为Offset已经提交了,  等Consumer重启后，会从之前已提交的Offset下一个位置重新开始消费，之前未处理完成的消息不会被再次处理，对于这个Consumer来说消息就丢失了

2. 拉消息后如果**先处理消息，在进行提交 Offset**，如果这个时候在提交之前发生异常宕机，因为没有提交成功Offset，等下次Consumer重启后还会从上次的Offset重新拉消息，不会出现消息丢失的情况，但会出现重复消费的情况，这里只能业务自己保证幂等性。

**消息为什么会丢失**：1.使用同步模式的时候，有3种状态保证消息被安全生产，在配置为1（只保证写入leader成功）的话，如果刚好leader partition挂了，数据就会丢失。

2.使用异步模式的时候，当缓冲区满了，如果配置为0（还没有收到确认的情况下，缓冲池一满，就清空缓冲池里的消息），数据就会被立即丢弃掉。

**解决丢失**：1.同步模式的时候，确认机制设置为-1，也就是让消息写入leader和所有的副本。

2.在异步模式下，如果消息发出去了，但还没有收到确认的时候，缓冲池满了，在配置文件中设置成不限制阻塞超时的时间，也就说让生产端一直阻塞，这样也能保证数据不会丢失。

在数据消费时，如果使用了storm，要开启storm的ackfail机制；如果没有使用storm，确认数据被完成处理之后，再更新offset值。低级API中需要手动控制offset值。

总结：

生产端重复发送：这个不重要，消费端消费之前从去重表中判重就可以

生产端丢失数据：1.异步方式缓冲区满了，就阻塞在那，等着缓冲区可用，不能清空缓冲区

2.发送消息之后回调函数，发送成功就发送下一条，发送失败就记在日志 中，等着定时脚本来扫描

消费端重复消费：建去重表

消费端丢失数据：关闭自动提交offset，处理完消息之后再移位

**消息传递语义**

**at most once(最多一次)**：消息可能会丢，但绝不会重复传递

**at least once(最少一次)**：消息绝不会丢，但可能会重复传递

**exactly once(精准传递一次)**：消息只会被精准地传递一次，不丢失不重复

**(1)**Producer向Broker发数据后会commit，如果 commit 成功，由于副本机制消息不会丢失，但Producer发数据给Broker后，如果遇到网络问题而造成通信中断，那么 Producer就无法准确判断该消息是否已经被提交（commit），这就可能造成**at least once**语义。

**(2)**在 Kafka 0.11.0.0版本之前， 如果 Producer没有收到消息commit响应结果，它只能重发消息，确保消息已经到Broker，重发的时候会把消息再次写入日志。

**实现exactly once**：后面版本的Kafka有idempotent(幂等 ai ding p tent) producer机制，这个机制里同一消息可能被producer发送多次，但是在broker端只会写入一次，他给每一条消息编号去重，而且对kafka开销影响不大。在多分区的情况，用**事务**保证**原子性地**写入多个分区，也就是写入到多个分区的消息要么全部成功，要么全部回滚。

consumer端可能无法消费事务中所有消息，而且消息可能被删除，所以可能要自己管理offset的提交，不要自动提交

**启用幂等传递**的方法配置：enable.idempotence = true。

**启用事务支持**的方法配置：设置属性 transcational.id = "指定值"。

**(3)**从Consumer角度看, 因为Offset是由Consumer自己维护的, 如果Consumer收到消息后更新 Offset，如果这时Consumer因为异常crash掉，那么新的Consumer接管后再次重启消费，就会造成**at most once**语义（消息会丢，但不重复）

**(4)**如果Consumer消费消息完成后,再更新 Offset，如果这时Consumer crash掉，那么新的 Consumer接管后重新用这个Offset拉取消息， 这时就会造成**at least once**语义（消息不丢，但被多次重复处理）。

**总结：默认 Kafka 提供 at least once**语义的消息传递，**允许用户通过在处理消息之前保存 Offset 的方式提供 at most once**语义。如果我们可以自己**实现消费幂等**，理想情况下这个系统的消息传递就是严格的**exactly once**, 也就是保证不丢失、且只会被精确的处理一次，但是这样是很难做到的。

**三次消息传递的过程**

1）Producer端发消息给Kafka Broker端

2）Kafka Broker把消息同步并且持久化数据

3）Consumer端从 Kafka Broker拉取消息来消费

每一步都可能丢失数据

**Kafka 只对已提交的消息做最大限度的持久化保证不丢失**

**对这句话的理解：**

**1.已提交的消息定义**:比如Kafka里 N 个Broker成功收到一条消息并且把他写到日志文件后，它们会告诉Producer端这条消息已成功提交了，这时这个消息在Kafka里就是已提交消息

**已提交**的定义：可以选择只要一个Broker成功保存这个消息就算已提交，也可以是所有 Broker都成功保存这个消息才算是已提交。

**2.最大限度的持久化保证不丢失：**Kafka不能保证在任何情况下都能做到数据不丢失。也就是Kafka不丢失数据是有前提条件的。假如消息保存在 N 个 Broker上，那么前提条件就是这N个Broker至少有1个是存活的，就可以保证消息不丢失。

也就是说 Kafka 是能做到不丢失数据的， 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才可以。

**发送流程**

1） Producer 端是直接与 Broker 中的 Leader Partition 交互的，所以在 Producer 端初始化中就需要通过 Partitioner 分区器从 Kafka 集群中获取到相关 Topic 对应的 Leader Partition 的元数据 。

2）待获取到 Leader Partition 的元数据后直接将消息发送过去。

3）Kafka Broker 对应的 Leader Partition 收到消息会先写入 Page Cache，定时刷盘进行持久化（顺序写入磁盘）。

4) Follower Partition 拉取 Leader Partition 的消息并保持同 Leader Partition 数据一致，待消息拉取完毕后需要给 Leader Partition 回复 ACK 确认消息。

5）待 Kafka Leader 与 Follower Partition 同步完数据并收到所有 ISR 中的 Replica 副本的 ACK 后，Leader Partition 会给 Producer 回复 ACK 确认消息。

**Producer 端为了提升发送效率，减少IO操作，发送数据的时候是将多个请求合并成一个个 RecordBatch，并将其封装转换成 Request 请求「异步」将数据发送出去（也可以按时间间隔方式，达到时间间隔自动发送），所以 Producer 端消息丢失更多是因为消息根本就没有发送到 Kafka Broker 端。**

**消费流程**

1）Consumer拉数据之前跟 Producer发送数据一样要通过订阅关系得到集群元数据, 找到相 关Topic对应的Leader Partition的元数据。

2）Consumer通过Pull模式主动去Kafka集群拉取消息。

3）多个Consumer可以组成一个消费者组(Consumer Group)，每个消费者组都有一个Group-Id。 同一个Consumer Group中的Consumer可以消费同一个Topic下不同分区的数据，但是 不会出现多个Consumer去消费同一个分区的数据。

4）拉到消息后处理业务逻辑，处理完后ACK确认，也就是提交Offset消费位移进度记录。

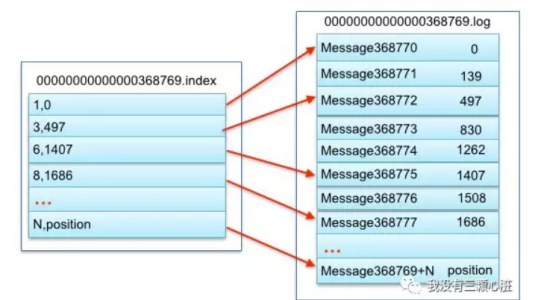
5）最后Offset会被保存到Kafka Broker集群中的 \_\_consumer\_offsets这个Topic里，而且每 个Consumer保存自己的Offset进度。

## **Kafka 中的底层存储设计**

假设我们现在 Kafka 集群只有一个 Broker，我们创建 2 个 Topic 名称分别为：「topic1」和「topic2」，Partition 数量分别为 1、2，那么我们的根目录下就会创建如下三个文件夹：

    | --topic1-0  
    | --topic2-0  
    | --topic2-1

现在假设我们设置每个 Segment 大小为 500 MB，并启动生产者向 topic1 中写入大量数据，topic1-0 文件夹中就会产生类似如下的一些文件：



索引文件中元数据 <3, 497>  = 数据文件中表示第 3 个 message（在全局 Partition 表示第 368769 + 3 = 368772 个 message）+ 该消息的物理偏移地址为 497。

**文件名 = 上一个 Segment 最后一条消息的 offset**

**找一个指定offset的message:**通过在所有 segment 的文件名中进行二分查找找到它归属的 segment，再在他 index索引文件种找到它对应到文件上的物理位置，就能拿出这个message

**Kafka如何知道 message的偏移**：因为在Kafka定义了标准的数据存储结构，在 Partition中的每一条message都包含了三个属性：

**1.offset：**表示 message 在当前 Partition 中的偏移量，是一个逻辑上的值，唯一确定了Partition 中的一条 message，可以简单的认为是一个**id**；

**2.MessageSize**：表示 message 内容 data 的大小；

**3.data：**message 的具体内容

## **Kafka 存储在文件系统上 --- 消息存储策略**

现代的操作系统针对磁盘的读写已经做了一些优化方案来加快磁盘的访问速度。比如，**预读**会提前将一个比较大的磁盘快读入内存。**后写**会将很多小的逻辑写操作合并起来组合成一个大的物理写操作。并且，操作系统还会将主内存剩余的所有空闲内存空间都用作**磁盘缓存**，所有的磁盘读写操作都会经过统一的磁盘缓存（除了直接 I/O 会绕过磁盘缓存）。综合这几点优化特点，**如果是针对磁盘的顺序访问，某些情况下它可能比随机的内存访问都要快，甚至可以和网络的速度相差无几。**

**上述的 Topic 其实是逻辑上的概念，面相消费者和生产者，物理上存储的其实是 Partition**，每一个 Partition 最终对应一个目录，里面存储所有的消息和索引文件。默认情况下，每一个Topic 在创建时如果不指定 Partition 数量时只会创建 1 个 Partition。比如，我创建了一个 Topic 名字为 test ，没有指定 Partition 的数量，那么会默认创建一个 test-0 的文件夹，这里的命名规则是：<topic\_name>-<partition\_id>

任何发布到 Partition 的消息都会被追加到 **Partition 数据文件的尾部**，这样的顺序写磁盘操作让 Kafka 的**效率非常高**（经验证，顺序写磁盘效率比随机写内存还要高，这是 Kafka 高吞吐率的一个很重要的保证）。

每一条消息被发送到 Broker 中，会根据 Partition 规则选择被存储到哪一个 Partition。如果 Partition 规则设置的合理，所有消息可以均匀分布到不同的 Partition中。

## **生产者设计概要**

每条消息都是很关键且不能容忍丢失么？偶尔重复消息可以么？我们关注的是消息延迟还是写入消息的吞吐量？

举个例子，有一个**信用卡交易处理系统**，当交易发生时会发送一条消息到 Kafka，另一个服务来读取消息并根据规则引擎来检查交易是否通过，将结果通过 Kafka 返回。对于这样的业务，**消息既不能丢失也不能重复**，由于交易量大因此吞吐量需要尽可能大，延迟可以稍微高一点。

再举个例子，假如我们需要收集用户在**网页上的点击数据**，对于这样的场景，**少量消息丢失或者重复是可以容忍的**，延迟多大都不重要只要不影响用户体验，吞吐则根据实时用户数来决定。

不同的业务需要使用不同的写入方式和配置。

**生产者写消息流程：**

1.创建一个ProducerRecord，这个对象需要包含消息的主题（topic）和值（value），可以选择性指定一个键值（key）或者分区（partition）。

2.发送消息时，生产者会对键值和值序列化成字节数组，然后发送到分配器（partitioner）。

3.如果我们指定了分区，那么分配器返回该分区即可；否则，分配器将会基于键值来选择一个分区并返回。

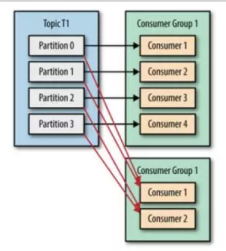
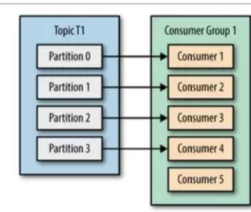
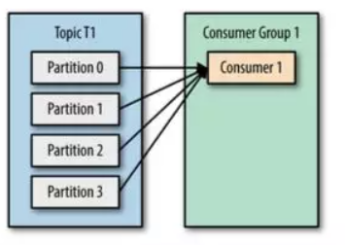
4.选择完分区后，生产者知道了消息所属的主题和分区，它将这条记录添加到相同主题和分区的批量消息中，另一个线程负责发送这些批量消息到对应的Kafka broker。

5.当broker接收到消息后，如果成功写入则返回一个包含消息的主题、分区及位移的RecordMetadata对象，否则返回异常。

6.生产者接收到结果后，对于异常可能会进行重试。

## **消费者设计概要**

一个消费者不够就在消费者组里加消费者



第三个图：Kafka很重要的特性：只要写入一次消息，可以支持任意多的应用读取这个消息

所以如果应用要读全量消息，就设置个消费组，如果消费能力不足就加消费者

**重平衡(rebalance)：**消费者离开后他负责的分区会分给其他分区

期间所有消费者都不能消费消息，所以会造成整个消费组不可用

消费者通过定期发送**心跳**（hearbeat）到一个作为组协调者（group coordinator）的 broker 来保持在消费组内存活

如果消费者超过一定时间没有发送心跳，那么它的会话（session）就会过期，组协调者会认为该消费者已经宕机，然后触发重平衡。可以看到，从消费者宕机到会话过期是有一定时间的，这段时间内该消费者的分区都不能进行消息消费；通常情况下，我们可以进行优雅关闭，这样消费者会发送离开的消息到组协调者，这样组协调者可以立即进行重平衡而不需要等待会话过期。

在 0.10.1 版本，Kafka 对心跳机制进行了修改，将发送心跳与拉取消息进行分离，这样使得发送心跳的频率不受拉取的频率影响。另外更高版本的 Kafka 支持配置一个消费者多长时间不拉取消息但仍然保持存活，这个配置可以避免活锁（livelock）。活锁，是指应用没有故障但是由于某些原因不能进一步消费。

## **Kafka 如何保证可靠性**

关系型数据库的可靠性保证是ACID，也就是原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。

Kafka 中的**可靠性**保证有如下四点：

1.对于一个分区来说，它的消息是有序的。如果一个生产者向一个分区先写入消息A，然后写入消息B，那么消费者会先读取消息A再读取消息B。

2.当消息写入所有in-sync状态的副本后，消息才会认为已提交（committed）。这里的写入有可能只是写入到文件系统的缓存，不一定刷新到磁盘。生产者可以等待不同时机的确认，比如等待分区主副本写入即返回，后者等待所有in-sync状态副本写入才返回。

3.一旦消息已提交，那么只要有一个副本存活，数据不会丢失。

4.消费者只能读取到已提交的消息。

**为什么用Kafka不用别的**：

数据传输块，吞吐量（每秒传的数据量）是百万级别的，其他的只有十万级或者万级。而且可靠性更高，比如有副本保证数据安全，有跳表思想设计log和index文件还有稀松索引来提升效率，零拷贝减少数据在不同环节来回复制，提升读写速度。但如果要保证高并发高性能高吞吐，那么Kafka有数据丢失风险，比如跟金额相关的要保证高可靠，丢失任何一个订单都不可以。

**Kafka快分为写入和读取**

**写入快**因为1.顺序写入 2.MMFile内存映射文件

1.顺序写入磁盘（因为每次读写都会寻址然后写入）读写速度大于内存随机读写，但不会删除数据，只会每个topic里每个消费者用offset表示读到第几条消息，解决是靠顺序写入基于时间和分区文件大小

2.顺序写入硬盘访问速度还是赶不上内存，所以用内存来非实时写入硬盘。MMF利用操作系统的page来实现文件到物理内存的直接映射，映射之后对物理内存的操作会被同步到硬盘上。但不可靠因为不会真正写到硬盘，要os的flush，但可以配置写入mmf就flush

**读取快**：1.基于sendfile实现零拷贝2.批量压缩

1.sendfile系统调用，文件数据被copy到内核中socket相关的缓冲区

再copy到协议引擎，比传统的减少了copy次数

2.把多个消息一起压缩

总结：它把所有的消息都变成一个批量的文件，并且进行合理的批量压缩，减少网络IO损耗，通过mmap提高I/O速度，写入数据的时候由于单个Partition是末尾添加所以速度最优；读取数据的时候配合sendfile直接暴力输出

1. **消息队列的延时和过期失效问题, 消息队列满了, 很多数据挤压很久怎么办**

可能消费端出问题，消息队列集群的磁盘快写满了也没人消费，然后积压太久导致设置了过期时间的消息过期了(RabbitMQ里才能设置)怎么办？

常见场景是：消费端每次消费之后要写 mysql，结果 mysql 挂了，消费端卡那儿了，不动了；或者是消费端出了个什么岔子，导致消费速度极其慢

**解决消息队列满了：**临时紧急扩容

1.先修复 consumer 的问题，确保其恢复消费速度，然后将现有 consumer 都停掉。

2.新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。

3.然后写一个临时的分发数据的consumer 程序，这个程序部署上去消费积压的数据，消费 之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。

4.接着临时征用10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的 数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍 速度来消费数据。

5.等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消 费消息。

**解决MQ中消息过期**：批量重导--在用户量少的时候，比如半夜，写临时程序把丢失的数据 重新灌入到MQ

1. **架构/设计一个MQ消息队列**

1.支持可伸缩性也就是能快速扩容，这样就可以增加吞吐量和容量，实现的话是设计个分布式的系统，比如按照kafka 的设计，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，就给 topic 增加 partition，然后做数据迁移，增加机器，这样就可以存放更多数据，提供更高的吞吐量

2. 数据落磁盘，这样才能保证进程挂了数据还在。落磁盘是用顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，也就是 kafka 的思路

3.要保证可用性，也就是要用多副本 -> leader & follower -> broker 挂了后重新选leader就可以继续对外服务。

4.支持数据不丢失

基本命令

先创建主题 发消息

