2-Kafka存储机制

重点内容

1.集群

2.参数设置，哪些参数

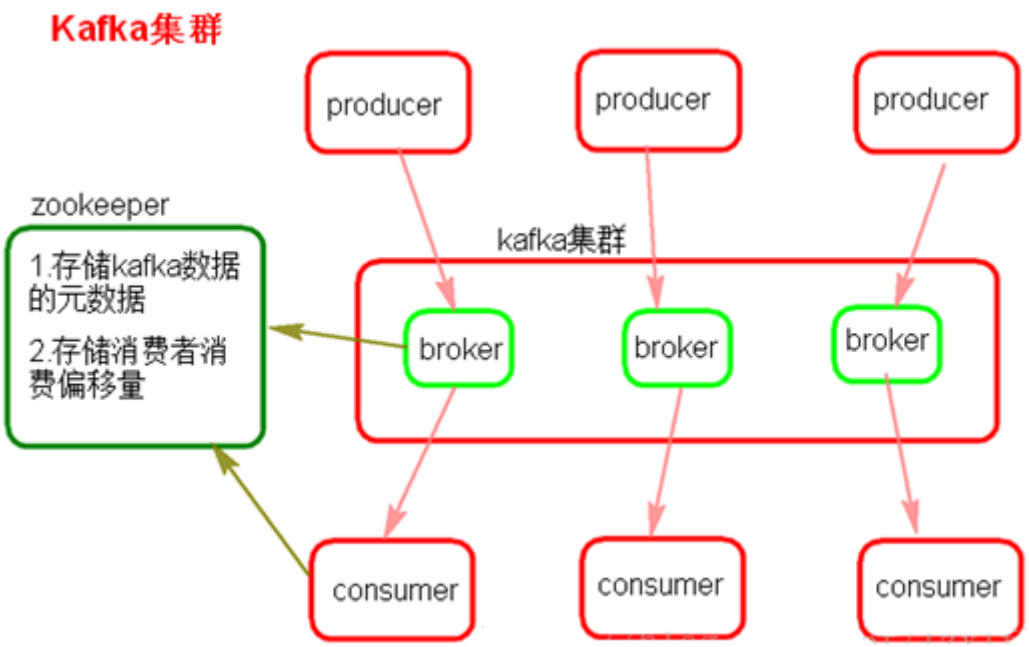
3.存储机制

4.可靠性

5.消费组的reblance(重新负载均衡)

6.高吞吐:存储机制(顺序性/零拷贝)

1 集群



1. Kafka架构是由producer（消息生产者）、consumer（消息消费者）、borker(kafka集群的server，负责处理消息读、写请求，存储消息，在kafka cluster这一层这里，其实里面是有很多个broker)、topic（消息队列/分类相当于队列，里面有生产者和消费者模型）、zookeeper(元数据信息存在zookeeper中，包括：存储消费偏移量，topic话题信息，partition信息) 这些部分组成。
2. kafka里面的消息是有topic来组织的，简单的我们可以想象为一个队列，一个队列就是一个topic，然后它把每个topic又分为很多个partition，这个是为了做并行的，在每个partition内部消息强有序，相当于有序的队列，其中每个消息都有个序号offset，比如0到12，从前面读往后面写。一个partition对应一个broker，一个broker可以管多个partition，比如说，topic有6个partition，有两个broker，那每个broker就管3个partition。这个partition可以很简单想象为一个文件，当数据发过来的时候它就往这个partition上面append，追加就行，消息不经过内存缓冲，直接写入文件，kafka和很多消息系统不一样，很多消息系统是消费完了我就把它删掉，而kafka是根据时间策略删除，而不是消费完就删除，在kafka里面没有一个消费完这么个概念，只有过期这样一个概念。
3. producer自己决定往哪个partition里面去写，这里有一些的策略，譬如如果hash，不用多个partition之间去join数据了。consumer自己维护消费到哪个offset，**每个consumer都有对应的**

group，group内是queue消费模型（各个consumer消费不同的partition，**因此一个消息在group内只消费一次**），**group间是publish-subscribe消费模型，各个group各自独立消费，互不影响**，因此一个消息在被每个group消费一次。

2 重要的参数配置

2.1 broker参数详解

https://kafka.apache.org/documentation.html#brokerconfigs

broker 端参数需要在 Kafka 目录下的 config/server.properties 文件中进行设置。当前对于绝大多数的 broker 端参数而言， Kafka 尚不支持动态修改——这就是说，如果要新增、修改，抑或是删除某些 broker 参数的话，需要重启对应的 broker 服务器。

broker.id

Kafka 使用唯一的一 个整数来标识每个 broker ， 这就是 broker.id ； 该参数默认是-1 。如果不指定，Kafka 会自动生成一个唯一值；总之，不管用户指定什么都必须保证该值在 Kafka 集群中是唯一的，不能与其他 broker 冲突 。

在实际使用中，推荐使用从0开始的数字序列，如0、1、2.....

log.dirs

该参数指定了 Kafka 持久化消息的目录；非常重要的参数!

若不设置该参数， Kafka 默认使用tmp/kafka-logs 作为消息保存的目录。把消息保存在 tmp 录下，生产环境中是不可取的。若待保存的消息数量非常多，那么最好确保该文件夹下有充足的磁盘空间。

该参数可以设置多个目录，以逗号分隔，比如 / home/kafka1，/home/kafka2 。在实际使用过程中，指定多个目录的做法通常是被推荐的，因为这样 Kafka 可以把负载均匀地分配到多个目录下。

若用户机器上有N块物理硬盘，那么设置 个目录（须挂载在不同磁盘上的目录）是一个很好的选择。N 个磁头可以同时执行写操作，极大地提升了吞吐量。

注意 ，这里的“均匀”是根据目录下的分区数进行比较的，而不是根据实际的磁盘空间。

zookeeper.connect

同样是非常重要的参数。主要是在zookeeper中，kafka的配置数据，有一个公共开始的跟节点；该参数没有默认的值，如果不配置，则使用zookeeper的/目录；

例如：zkl :218 l ,zk2:2181,zk3:2181/kafka_clusterl；/kafka_cluster就是kafka的配置的目录；配置了，可以起到很好的隔离效果。这样管理 Kafka 集群将变得更加容易。

listeners

broker 监听器的 csv (comma-separated values)列表，格式是［协议://:主机名:端口]，[协议]:[// 主机名:端口]。

该参数主要用于客户端连接 broker 使用，可以认为是 broker 端开放给 clients的监听端口 如果不指定主机名，则表示绑定默认网卡：如果0.0.0.0，则表示绑定所有网卡。 Kafka 当前支持的协议类型包括 PLAINTEXT、SSL以及 SASL SSL 等。

advertised.listeners

跟 listeners 类似，该参数也是用于发布给 clients 的监听器；不过该参数主要用于 IaaS 环境，比如云上的机器通常都配有多块网卡（私网网卡和公网网卡）。

对于这种机器，用户可以设置该参数绑定公网 IP 供外部 clients 使用，然后配置上面的 listeners 来绑定私网 IP供broker间通信使用。

当然不设置该参数也是可以的，只是云上的机器很容易出现 clients 无法获取数据的问题，原因就是 listeners 绑定的是默认网卡，而默认网卡通常都是绑定私网的。

在实际使用场景中，对于配有多块网卡的机器而言，这个参数通常都是需要配置的。

unclean.leader.election.enable

是否开启 unclean leader 选举。Kafka 社区在1.0.0 版本才正式将该参数默认值调整为 false；

即表明如果发生这种情况， Kafka 不允许从剩下存活的非 ISR 副本中选择一个当 leader。因为如果 true，这样做固然可以让 Kafka 继续提供服务给 clients ，但会造成消息数据的丢失，正式环境中，数据不丢失是基本的业务需求；

番外：ISR解释

ISR 全称是in-sync replica ，翻译过来就是与 leader replica 保持同步的 replica 集合 ;一个partition 可以配置N个replica ，那么这是否就意味着该 partition可以容忍 N-1 replica 失效而不丢失数据呢？答案是“否”!

Kafka partition 动态维护 一个replica 集合。该集合中的所有 replica 保存的消息日志都与leader replica 保持同步状态。只有这个集合中的 replica 才能被选举为 leader ，也只有该集合中所有 replica 都接收到了同 条消息， Kafka 才会将该消息置于“已提交”状态，即认为这条消息发送成功。

Kafka 中只要这个集合中至少存在一个 replica ，那些“已提交”状态的消息就不会丢失；

这句话的两个关键点：

1. ISR 中至少存在一个“活着的”replica；
2. replica “已提 ”消息 。

Kafka 对于没有提交成功的消息不做任何保证，只保证在 ISR 存活的情况下“已提交”的消息不会丢失。正常情况下，partition 的所有 replica （含 leader replica ）都应该与 leader replica 保持同步，即所有 replica 都在 ISR 中。

因为各种各样的原因，一小部分 replica 开始落后于 leader replica的进度 。当滞后 一定程度时， Kafka 会将这些 replica “踢”出 ISR；相反地，当这replica 重新追上leader进度时候，kafka会再次把它们加回 ISR中。

delete.topic.enable

是否允许 Kafka 删除 topic 。

默认情况下， Kafka 集群允许用户删除topic 及其数据。这样当用户发起删除 topic 操作时， broker 端会执行 topic 删除逻辑。

在实际生产环境中我们发现允许 Kafka 删除 opic 其实是一个很方便的功能，再加上自Kafka 0.0 新增的 ACL 权限特性，以往对于误操作和恶意操作的担心完全消失了，因此设置该参数为 true 是推荐的做法。

log.retention. {hours|minutes|ms }

这组参数控制了消息数据的留存时间；默认的留存时间是7天；即 Kafka 只会保存最近 7天的数据，并自动删除 7天前的数据。

当前较新版本的Kafka 会根据消息的时间戳信息进行留存与否的判断；老版本消息格式没有时间戳，Kafka 会根据日志文件的最近修改时间（ last modified time ）进行判断。

三个参数如果同时设置，优先选取ms的设置，minutes次之，hours最后。

log.retention.bytes

这个参数定义了空间维度上的留存策略；参数默认值是-1，表示 Kafka 永远不会根据消息日志文件总大小来删除日志。

对于大小超过该参数的分区日志而言， Kafka 会自动清理该分区的过期日志段文件。

min.insync.replicas

该参数表示kafka存储的最小副本数，producer发送数据的时候，指定了 broker 端必须成功响应 clients 消息发送的最少副本数；

该参数其实是与 producer 端的 acks 参数配合使用的； 并且min.insync.replicas 也只有在 acks=-1 时才有意义;acks=-1 表示 producer端寻求最高等级的持久化保证；

假如 broker 端无法满足该条件，则 clients 的消息发送并不会被视为成功。它与 acks 配合使用可以令 Kafka集群达成最高等级的消息持久化；

举 个例子，假设某个topic 的每个分区的副本数是3 ，那么推荐设置该参数为 2，这样我们就能够容忍一台broker 窑机而不影响服务；若设置参数为3 ，那么只要任何一台 broker 岩机，整个Kafka 集群将无法继续提供服务。

num.network threads

一个 broker 在后台用于处理网络请求的线程数，默认是3 。

broker启动时会创建多个线程处理来自其他broker和clients 发送过来的各种请求。会将接收到的请求转发到后面的处理线程中。在真实的环境中，用户需要不断地监控NetworkProcessorAvgldlePercent JMX 指标; 如果该指标持续低于0 .3 ;建议适当增加该参数的值。

num.io.threads

这个参数就是控制 broker 端实际处理网络请求的线程数，默认值是8；

Kafka broker 默认创建 8个线程以轮询方式不停地监昕转发过来的网络请求并进行实时处理。 Kafka 同样也为请求处理提供了一个 JMX 监控指标 RequestHandler A vgldlePercent。如果发现该指标持续低于 0.3 ， 则可以考虑适当增加该参数的值。

message.max. bytes

Kafka broker 能够接收的最大消息大小，默认是 977kb；还不到1MB ， 可见是非常小的。

在实际使用场景中，突破 1MB 大小的消息十分常见，因此用户有必要综合考虑 Kafka 集群可能处理的最大消息尺寸并设置该参数值

2.2 producer参数详解

https://kafka.apache.org/documentation.html#producerconfigs

bootstrap.servers

该参数指定了一组host:port 对，用于创建向 Kafka broker 服务器的连接，比如:kl:9092,k2:9092,k3:9092。

如果 Kafka 集群中机器数很多，那么只需要指定部分 broker 即可，不需要列出所有的机器。因为不管指定几台机器，producer 都会通过该参数找到并发现集群中所有的 broker；为该参数指定多台机器只是为了故障转移使用。这样即使某一台 broker 挂掉了，producer 重启后依然可以通过该参数指定的其他 broker 连入 Kafka 集群。

另外，如果 broker 端没有显式配置 listeners 使用 IP 地址，那么最好将该参数也配置成主机名，而不是 IP 地址。因为 Kafka 内部使用的就是FQDN (Fully Qualified Domain Name ）。

key.serializer

被发送到 broker 端的任何消息的格式都必须是字节数组，因此消息的各个组件必须首先做序列化，然后才能发送到 broker。

该参数就是为消息的 key 做序列化之用的。这个参数指定的是实现了org.apache.kafka.common.serialization.Serializer接口的类的全限定名称。

Kafka 为大部分的初始类型（ primitive type ）默认提供了现成的序列化器。producer 程序在发送消息时不指定 key，这个参数也是必须要设置的；否则程序会抛出 ConfigException 异常，提示“key.serializer”参数无默认值，必须要配置。

value.serialize

和 key.serializer 类似，只是它被用来对消息体（即消息 value ）部分做序列化，将消息value 部分转换成字节数组 。

value.serializer 和 key. serializer 可以设置相同的值，也可以不同的值 。只要消费端，消费数据的时候，保持一致就可以了；

一定要注意的是，这两个参数都必须是全限定类名，org.apache.kafka.common.serialization.Serializer

acks

acks 参数用于控制 producer 生产消息的持久性（durability）；对于 producer 而言， Kafka在乎的是“已提交”消息的持久性。一旦消息被成功提交，那么只要有任何一个保存了该消息的副本“存活”，这条消息就会被视为“不会丢失的” 。

经常碰到抱怨Kafka的producer会丢消息，其实这里混淆了 一个概念，即那些所谓的“已丢失”的消息其实并没有被成功写入 Kafka 。换句话说，它们并没有被成功提交，因此 Kafka 对这些消息的持久性不做任何保障；

当然 producer API 确实提供了回调机制供用户处理发送失败的情况。具体来说，当 producer 发送一条消息给 Kafka 集群时，这条消息会被发送到指定 topic 分区 leader 所在的 broker 上，producer 等待从该 leader broker 返回消息的写入结果（当然并不是无限等待，是有超时时间的）以确定消息被成功提交。这一切完成后 producer 可以继续发送新的消息 。

Kafka 能够保证的是 consumer 永远不会读取到尚未提交完成的消息；显然， leader broker 何时发送写入结果返还给 producer 就需要仔细考虑的问题了，也会直接影响消息的持久性甚至是 producer 端的吞吐量（producer越快地接收到 leader broker响应，就能发送下一条消息）；producer 端的 acks 参数就是用来控制做这件事情的 。

acks 指定了在给 producer 发送响应前， leader broker 必须要确保已成功写入该消息的副本数 。 当前 acks 有 3 个取值： **0、 1和 all**

- acks = 0：设置成 0 表示 producer 完全不理睬 leader broker 端的处理结果。此时，producer 发送消息后立即开启下一条消息的发送，根本不等待 leader broker 端返回结果 。
    - 由于不接收发送结果，因此在这种情况下 producer.send 的回调也就完全失去了作用，即用户无法通过回调机制感知任何发送过程中的失败，所以 acks=O 时 producer 并不保证消息会被成功发送。
    - 但凡事有利就有弊，由于不需要等待响应结果，通常这种设置下 producer 的吞吐量是最高的 。
- acks = all 或者-1：表示当发送消息时， leader broker 不仅会将消息写入本地日志，同时还会等待 ISR 中所有其他副本都成功写入它们各自的本地日志后，才发送响应结果给 producer。
    - 显然当设置 acks=all 时，只要 ISR中 至少有一个副本是处于"存活"状态的 ，那么这条消息就肯定不会丢失，因而可以达到最高的消息持久性，但通常这种设置下producer 的吞吐量也是最低的 。
- acks = 1 ： 是 0 和 all 折中的方案，也是默认的参数值。

producer发送消息后 leader broker仅将该消息写入本地日志，然后发送响应结果给 producer，而无须等待 ISR中其他副本写入该消息。 那么此时只要该 leader broker 一直存活 ， Kafka 就能够保证这条消息不丢失。这实际上是一种折中方案，既可以达到适当的消息持久性，同时也保证了 producer 端的吞吐量。

总结一下：acks 参数控制 producer 实现不同程度的消息持久性，它有 3 个取值，对应的优缺点以使用场景如表所示。

acks	producer	消息持久性	使用场景
0	最高	最差	1、完全不管理消息是否发送成功； 2、允许消息丢失（比如统计服务器日志等）
1	适中	适中	一般场景
all 或-1	最差	最高	不能容忍消息丢失

buffer.memory

该参数指定了 producer 端用于缓存消息的缓冲区大小，单位是字节，默认值是 33554432,即 32MB 。

如前所述，由于采用了异步发送消息的设计架构， Java 版本 producer 启动时会首先创建一块内存缓冲区用于保存待发送的消息，然后由另 一个专属线程负责从缓冲区中读取消息执行真正的发送。这部分内存空间的大小即是由 buffer.memory 参数指定的。

若 producer 向缓冲区写消息的速度超过了专属 I/0 线程发送消息的速度，那么必然造成该缓冲区空间的不断增大。此时 producer 会停止手头的工作等待 I/0 线程追上来，若一段时间之后 I/0 线程还是无法追上 producer 的进度， 就会抛出异常；若 producer 程序要给很多分区发送消息，那么就需要仔细地设置这个参数，以防止过小的内存缓冲区降低了producer 程序整体的吞吐量。

compression.type

设置 producer 端是否压缩消息，默认值是 none ， 即不压缩消息 。

Kafka 的 producer 端引入压缩后可以显著地降低网络 I/O 传输开销从而提升整体吞吐量，但也会增加 producer 端机器的 CPU 开销。另外，如果 broker 端的压缩参数设置得与 producer 不同， broker 端在写入消息时也会额外使用 CPU 资源对消息进行对应的解压缩 - 重新压缩操作。

目前 Kafka 支持 3 种压缩算法：GZIP、Snappy 和 LZ4。根据实际使用经验来看 producer 结合 LZ4 的性能是最好； LZ4 > Snappy > GZIP；

retries

Kafka broker 在处理写入请求时可能因为瞬时的故障（比如瞬时的leader选举或者网络抖动）导致消息发送失败。这种故障通常都是可以自行恢复的，如果把这些错误封装进回调函数的异常中返还给producer,producer程序也并没有太多可以做的，只能简单地在回调函数中重新尝试发送消息。与其这样，还不如 producer 内部自动实现重试。因此 Java 版本 producer 在内部自动实现了重试，当然前提就是要设置retries参数。

该参数表示进行重试的次数，默认值是 0，表示不进行重试。

在实际使用过程中，设置重试可以很好地应对那些瞬时错误，因此推荐用户设置该参数为一个大于 0 的值 。

只不过在考虑retries 的设置时，有两点需要着重注意 。

1、重试可能造成消息的重复发送；

比如由于瞬时的网络抖动使得 broker 端已成功写入消息但没有成功发送响应给 producer，因此 producer 会认为消息发送失败，从而开启重试机制。为了应对这一风险， Kafka 要求用户在 consumer 端必须执行去重处理 。 令人欣喜 的是，社区已于 0.11.0.0 版本开始支持“精确一次”处理语义，从设计上避免了类似的问题 。

2、重试可能造成消息的乱序；

当前 producer 会将多个消息发送请求（默认是 5 个） 缓存在内存中；如果由于某种原因发生了消息发送的重试，就可能造成消息流的乱序 。为了避免乱序发生， Java 版本 producer 提供了 max.in.flight.requets.per.connection 参数 。一旦用户将此参数设置成 1, producer 将确保某一时刻只能发送一个请求 。

另外， producer 两次重试之间会停顿一段时间，以防止频繁地重试对系统带来冲击 。 这段时间是可以配置的，由参数retry.backff.ms 指定，默认是 100 毫秒 。 由于 leader “换届选举 ”是最常见的瞬时错误，推荐用户通过测试来计算平均 leader 选举时间并根据该时间来设定retries 和retry.backff.ms 的值 。

batch.size

batch.size 是 producer 最重要的参数之一！ 它对于调优 producer 吞吐量和延时性能指标都有着非常重要的作用 。

producer 会将发往同一分区的多条消息封装进一个 batch中，当 batch 满了的时候， producer 会发送 batch 中的所有消息 。不过， producer并不总是等待batch满了才发送消息，很有可能当batch还有很多空闲空间时 producer 就发送该 batch 。显然，batch 的大小就显得非常重要 。

通常来说，一个小的 batch 中包含的消息数很少，因而一次发送请求能够写入的消息数也很少，所以 producer 的吞吐量会很低；一个 batch 非常之巨大，那么会给内存使用带来极大的压力，因为不管是否能够填满，producer 都会为该batch 分配固定大小的内存。

因此batch.size 参数的设置其实是一种时间与空间权衡的体现 。**batch.size 参数默认值是 16384 ，即 16KB** 。这其实是一个非常保守的数字。 在实际使用过程中合理地增加该参数值，通常都会发现 producer 的吞吐量得到了相应的增加 。

linger.ms

参数控制消息发送延时行为的 。 该参数默认值是 0，表示消息需要被立即发送，无须关心 batch 是否已被填满；

大多数情况下这是合理的；毕竟我们总是希望消息被尽可能快地发送；不过这样做会拉低 produc 吞吐量，毕竟 produce 发送的每次请求中包含的消息数越多，produce就越能将发送请求的开销摊薄到更多的消息上 从而提升吞吐量；如果要设置，跟上述参数batch.size 配合使用；针对消息的发送的一种权衡考虑；

max.request.size

官网中给出的解释是，该参数用于控制 producer 发送请求的大小 。实际上该参数控制的是producer 端能够发送的最大消息大小 。

由于请求有一些头部数据结构，因此包含一条消息的请求的大小要比消息本身大 。 不过姑且把它当作请求的最大尺寸是安全的 。如果 producer 要发送尺寸很大的消息 ， 那么这个参数就是要被设置的 。默认的 1048576 字节（1MB）

request.timeout.ms

当 producer 发送请求给broker后，broker 需要在规定的时间范围内将处理结果返还给producer 。 默认是 30 秒 。

如果 broker 在 30 秒内都没有给 producer 发送响应，就会认为该请求超时了，回调函数中显式地抛出 TimeoutException异常 。默认的 30 秒对于一般的情况是足够的，但如果 producer发送的负载很大 ,超时的情况就很容易碰到，此时就应该适当调整该参数值。

2.3 consumer参数详解

https://kafka.apache.org/documentation.html#consumerconfigs

bootstrap.servers

和 producer 相同，这是必须要指定的参数。该参数指定了一组 host:port 对，用于 创建与 Kafka broker 服务器的 Socket 连接 。 可以指定多组，使用逗号分隔，如kafkal :9092,kafka2:9092,kafka3 :9092。在实际生产环境中需要替换成线上 broker 列表。

另外，若 Kafka集群中 broker 机器数很多，我们只需要指定部分 broker 即可，不需要列出完整的 broker 列表 。这是因为不管指定了几台机器， consumer 启动后都能通过这些机器找到完整的 broker 列表，因此为该参数指定多台机器通常只是为了常规的 failover 使用 。 这样即使某一台 broker 挂掉了，consumer 重启后依然能够通过该参数指定的其他 broker 连接 Kafka 集群。

需要注意的是，如果 broker 端没有显式配置 listeners （或 advertised.listeners ）使用 IP 地址的话，那么最好将 bootstrap.servers 配置成主机名而不要使用 IP 地址，因为Kafka 内部使用的是全称域名（ FQDN, Fully Qualified Domain Name ）。倘若不统一， 会出现无法获取元数据的异常。

group.id

该参数指定的是 consumer group 的名字。它能够唯一标识一个 consumer group 细心的读者可能会发现宫网上该参数是有默认值的，即一个空字符串 。 但在开发 consumer 程序时我们依然要显式指定 group.id ， 否则 consumer 端会抛出 InvalidGroupIdException 异常 。

通常为group.id 设置一个有业务意义的名字就可以了。

key.deserializer

consumer 代码从 broker 端获取的任何消息都是宇节数组的格式，因此消息的每个组件都要执行相应的解序列化操作才能“还原”成原来的对象格式 。 这个参数就是为消息的 key 做反序列化的 。

该参数值必须是实现 org.apache .kafka.common.serialization.Deserializer 接口的 Java类的全限定名称。

Kafka 默认为绝大部分的初始类型（ primitive type ）提供了现成的解序列化器。上面代码中使用了 org.apache.kafka.common.serialization.StringDeserializer 类。 该类会将接收到的字节数组转换成 UTF-8 编码的字符串。 consumer 支持用户自定义 deserializer， 这通常都与 producer 端自定义 serializer “遥相呼应”。值得注意的是，不论 consumer 消费的消息是否指定了间， consumer 都必须要设置这个参数，否则程序会抛出 ConfigException，提示“key.deserializer” 没有默认值。

value.deserializer

与 value.deserializer 类似，该参数被用来对消息体（即消息 value ）进行反序列化，从而把消息“还原”回原来的对象类型。

value.deserializer 可以设置成与key.deserializer 不同的值，前提是 key.serializer 与 value.serializer 设置了不同的值 。

在使用过程中，我们一定要谨记 key.deserializer 和 value.deserializer 指定的是类的全限定名，单独指定类名是行不通的 。

session.timeout.ms

非常重要的参数之一！

session.timeout.ms 是 consumer group 检测组内成员发送崩溃的时间 。

假设你设置该参数为 5 分钟，那么当某个 group 成员突然崩攒了（比如被 kill -9 或岩机）， 管理 group 的 Kafka 组件（即消费者组协调者，也称 group coordinator）有可能需要 5 分钟才能感知到这个崩溃。显然我们想要缩短这个时间，让coordinator 能够更快地检测到 consumer 失败 。

这个参数还有另外一重含义： consumer 消息处理逻辑的最大时间。

倘若 consumer 两次 poll 之间的间隔超过了该参数所设置的阈值，那么 coordinator 就会认为这个 consumer 已经追不上组内其他成员的消费进度了，因此会将该 consumer 实例“踢出”组，该 consumer 负责的分区也会被分配给其他 consumer。

在最好的情况下，这会导致不必要的 rebalance，因为 consumer 需要重新加入 group 。更糟的是，对于那些在被踢出 group 后处理的消息， consumer 都无法提交位移——这就意味着这些消息在 rebalance 之后会被重新消费一遍。如果一条消息或一组消息总是需要花费很长的时间处理，那么 consumer 甚至无法执行任何消费，除非用户重新调整参数 。鉴于以上的“窘境”， Kafka 社区于 0 .10.1.0 版本对该参数的含义进行了拆分 。 在该版本及以后的版本中， session.timeout.ms 参数被明确为“ coordinator 检测失败的时间” 。

因此在实际使用中，用户可以为该参数设置一个比较小的值，让 coordinator 能够更快地检测 consumer 崩溃的情况，从而更快地开启 rebalance，避免造成更大的消费滞后（ consumer lag ） 。目前该参数的默认值是 10 秒。

max.poll.interval.ms

Apache官网max.poll.interval.ms 上的解释如下，消费者组中的一员在拉取消息时如果超过了设置的最大拉取时间，则会认为消费者消费消息失败，kafka会重新进行重新负载均衡，以便把消息分配给另一个消费组成员。

在一个典型的 consumer 使用场景中，用户对于消息的处理可能需要花费很长时间。这个参数就是用于设置消息处理逻辑的

最大时间的 。 假设用户的业务场景中消息处理逻辑是把消息、“落地”到远程数据库中，且这个过程平均处理时间是 2 分钟，那么用户仅需要将 max.poll.interval.ms 设置为稍稍大于 2 分钟的值即可，而不必为 session. neout.ms 也设置这么大的值。

通过将该参数设置成实际的逻辑处理时间再结合较低的 session.timeout.ms 参数值，consumer group 既实现了快速的 consumer 崩溃检测，也保证了复杂的事件处理逻辑不会造成不必要的 rebalance 。

auto.offset. reset

指定了无位移信息或位移越界（即 consumer 要消费的消息的位移不在当前消息日志的合理区间范围）时 Kafka 的应对策略 。 特别要注意这里的无位移信息或位移越界，只有满足这两个条件中的任何一个时该参数才有效果 。

关于这一点，我们举一个实际的例子来说明 。 假设你首次运行一个 consumer group 并且指定从头消费 。 显然该 group 会从头消费所有数据，因为此时该 group 还没有任何位移信息 。 一旦该 group 成功提交位移后，你重启了 group，依然指定从头消费 。 此时你会发现该 group 并不会真的从头消费一一因为 Kafka 已经保存了该 group 的位移信息，因此它会无视 auto.offset.reset 的设置。

目前该参数有如下 3 个可能的取值：

- earliest：指定从最早的位移开始消费 。 注意这里最早的位移不一定就是 0 。
- latest：指定从最新处位移开始消费 。
- none ： 指定如果未发现位移信息或位移越界，则抛出异常。笔者在实际使用过程中几乎从未见过将该参数设置为 none 的用法，因此该值在真实业务场景中使用甚少。

enable.auto.commit

该参数指定 consumer 是否自动提交位移 。

若设置为 true，则 consumer 在后台自动提交位移；

否则，用户需要手动提交位移。

对于有较强“精确处理一次”语义需求的用户来说，最好将该参数设置为 false，由用户自行处理位移提交问题 。

fetch.max.bytes

一个经常被忽略的参数。它指定了 consumer 端单次获取数据的最大字节数 。 若实际业务消息很大，则必须要设置该参数为一个较大的值，否则 consumer 将无法消费这些消息 。

max.poll.records

该参数控制单次 poll 调用返回的最大消息数 。

比较极端的做法是设置该参数为 1，那么每次 poll 只会返回 1 条消息。如果用户发现 consumer 端的瓶颈在 poll 速度太慢，可以适当地增加该参数的值。如果用户的消息处理逻辑很轻量，默认的 500 条消息通常不能满足实际的消息处理速度 。

heartbeat.interval.ms

该参数和 request.timeout.ms 、max.poll.interval.ms 参数是最难理解的 consumer 参数 。 前面已经讨论了后两个参数的含义，这里解析一下heartbeat.interval.ms的含义及用法 。

从表面上看，该参数似乎是心跳的问隔时间，但既然已经有了上面的 session.timeout.ms 用于设置超时，为何还要引入这个参数呢?

这里的关键在于要搞清楚 consumer group 的其他成员，如何得知要开启新一轮 rebalance；当 coordinator 决定开启新一轮 rebalance 时，它会将这个决定以 REBALANCE_IN_PROGRESS 异常的形式“塞进” consumer 心跳请求的 response 中，这样其他成员拿到 response 后才能知道它需要重新加入 group。显然这个过程越快越好，而heartbeat. interval.ms 就是用来做这件事情的 。

比较推荐的做法是设置一个比较低的值，让 group 下的其他 consumer 成员能够更快地感知新一轮 rebalance. 开启了。

注意，该值必须小于 session.timeout.ms ！ 这很容易理解，毕竟如果 consumer 在 session.timeout.ms 这段时间内都不发送心跳， coordinator 就会认为它已经 dead，因此也就没有必要让它知晓 coordinator 的决定了。

connections.max.idle.ms

这又是一个容易忽略的参数！经常有用户抱怨在生产环境下周期性地观测到请求平均处理时间在飘升，这很有可能是因为 Kafka 会定期地关闭空闲 Socket 连接导致下次 consumer 处理请求时需要重新创建连向 broker 的 Socket 连接 。

当前默认值是 9 分钟，如果用户实际环境中不在乎这些Socket资源开销，比较推荐设置该参数值为-1，即不要关闭这些空闲连接。

2.4 更多配置

https://kafka.apache.org/documentation.html#topicconfigs

https://kafka.apache.org/documentation.html#configuration

https://kafka.apachecn.org/documentation.html#majordesignelements

3 存储机制

Kafka 是为了解决大数据的实时日志流而生的, 每天要处理的日志量级在千亿规模。**对于日志流的特点主要包括：**

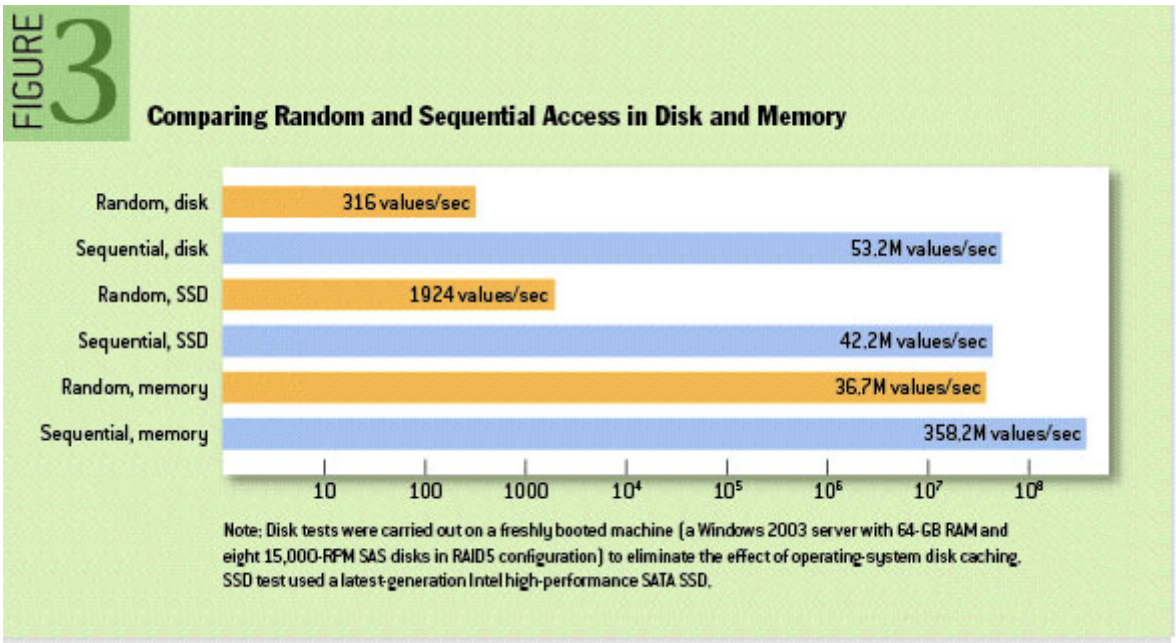
1. 数据实时产生
2. 海量数据存储与处理

所以它必然要面临分布式系统遇到的高并发、高可用、高性能等三高问题。

对于 Kafka 的存储需要保证以下几点：

1. 存储的主要是消息流（可以是简单的文本格式也可以是其他格式，对于 Broker 存储来说，它并不关心数据本身）
2. 要支持海量数据的高效存储、高持久化（保证重启后数据不丢失）
3. 要支持海量数据的高效检索（消费的时候可以通过offset或者时间戳高效查询并处理）
4. 要保证数据的安全性和稳定性、故障转移容错性

3.1 kafka 存储选型



磁盘和内存的 IO 速度对比

从上图性能测试的结果看出普通机械磁盘的顺序I/O性能指标是53.2M values/s，而内存的随机I/O性能指标是36.7M values/s。由此似乎可以得出结论：**磁盘的顺序I/O性能要强于内存的随机I/O性能。**

另外，如果需要较高的存储性能，必然是**提高读速度和写速度**：

1. 提高读速度：利用索引，来提高查询速度，但是有了索引，大量写操作都会维护索引，那么会降低写入效率。常见的如关系型数据库：mysql等
2. 提高写速度：这种一般是采用日志存储, 通过顺序追加写的方式来提高写入速度，因为没有索引，无法快速查询，最严重的只能一行行遍历读取。常见的如大数据相关领域的基本都基于此方式来实现。

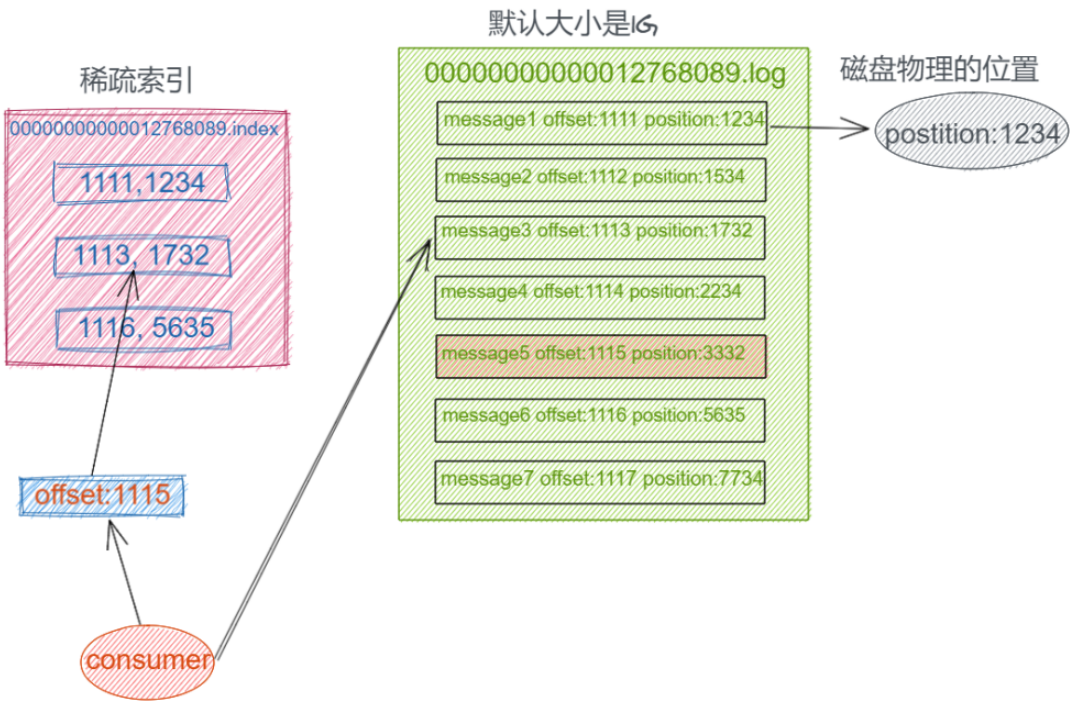
3.2 Kafka 存储方案剖析

对于 Kafka 来说， 它主要用来处理海量数据流，这个场景的特点主要包括：

1. 写操作：写并发要求非常高，基本得达到百万级 TPS，顺序追加写日志即可，无需考虑更新操作
2. 读操作：相对写操作来说，比较简单，只要能按照一定规则高效查询即可（offset或者时间戳）

对于写操作来说，直接采用**顺序追加写日志**的方式就可以满足 Kafka 对于百万TPS写入效率要求。

所以我们重点放在如何解决高效查询这些日志。Kafka采用了**稀疏哈希索引（底层基于Hash Table 实现）**的方式。

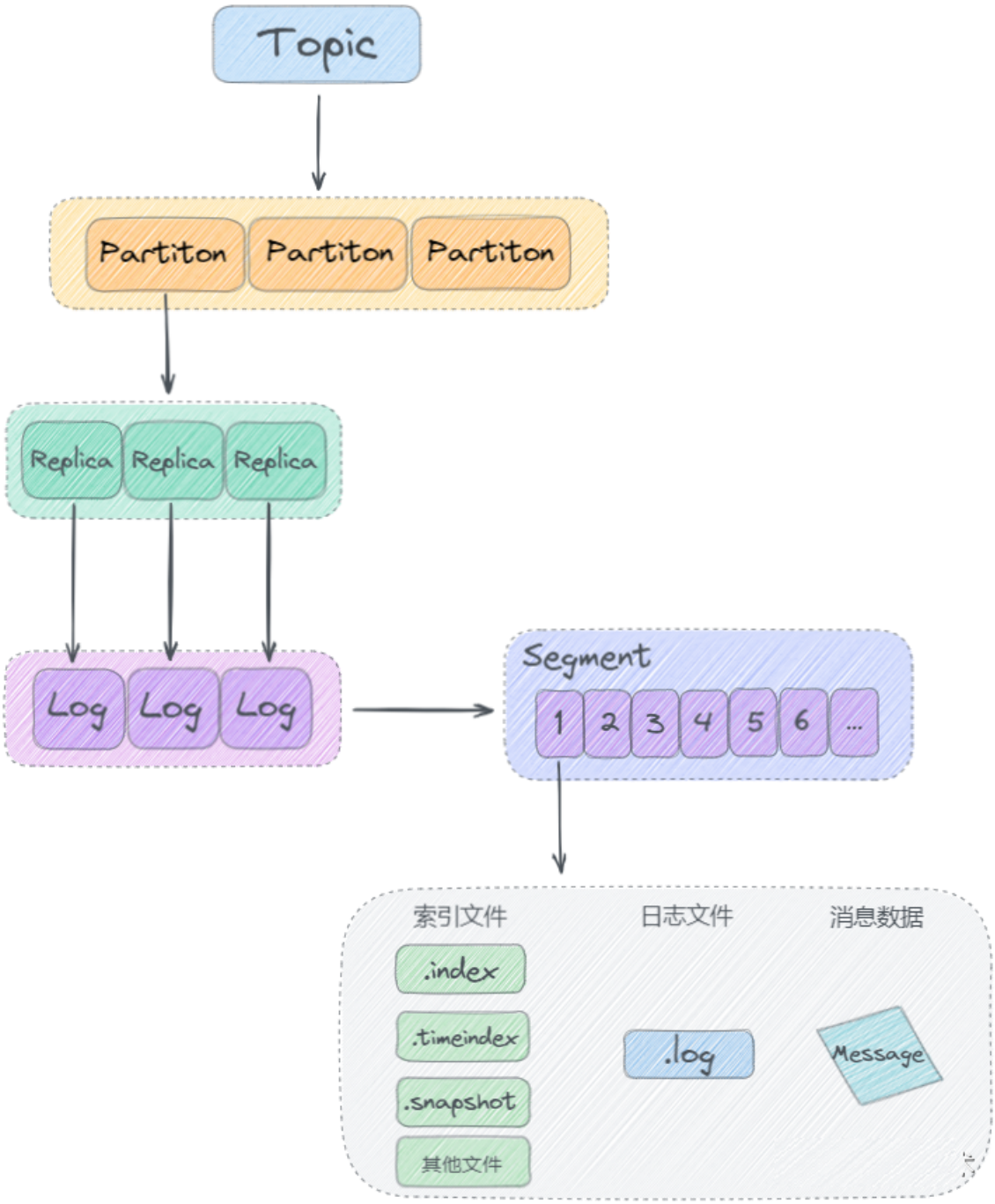


把消息的 Offset 设计成一个有序的字段，这样消息在日志文件中也就有序存放了，也不需要额外引入哈希表结构， 可以直接将消息划分成若干个块，**对于每个块，我们只需要索引当前块的第一条消息的 Offset （类似二分查找算法的原理）**，即先根据 Offset 大小找到对应的块， 然后再从块中顺序查找，这样就可以快速定位到要查找的消息。

3.3 kafka 存储架构设计

从上分析我们可以知道： Kafka 最终的存储实现方案， 即**基于顺序追加写日志 + 稀疏哈希索引。**

Kafka 日志存储结构：



从上图可以看出Kafka 是基于「主题 + 分区 + 副本 + 分段 + 索引」的结构：

1. kafka 中消息是以主题 Topic 为基本单位进行归类的，这里的 Topic 是逻辑上的概念，实际上在磁盘存储是根据分区 Partition 存储的, 即每个 Topic 被分成多个 Partition，分区 Partition 的数量可以在主题 Topic 创建的时候进行指定。
2. Partition 分区主要是为了解决 Kafka 存储的水平扩展问题而设计的， 如果一个 Topic 的所有消息都只存储到一个 Kafka Broker上的话， 对于 Kafka 每秒写入几百万消息的高并发系统来说，这个 Broker 肯定会出现瓶颈， 故障时候不好进行恢复，所以 Kafka 将 Topic 的消息划分成多个 Partition， 然后均衡的分布到整个 Kafka Broker 集群中。
3. Partition 分区内每条消息都会被分配一个唯一的消息 id,即我们通常所说的 偏移量 Offset，因此 kafka 只能保证每个分区内部有序性,并不能保证全局有序性。
4. 然后每个 Partition 分区又被划分成了多个 LogSegment，这是为了防止 Log 日志过大，Kafka 又引入了日志分段(LogSegment)的概念，将 Log 切分为多个 LogSegement，相当于一个巨型文件被

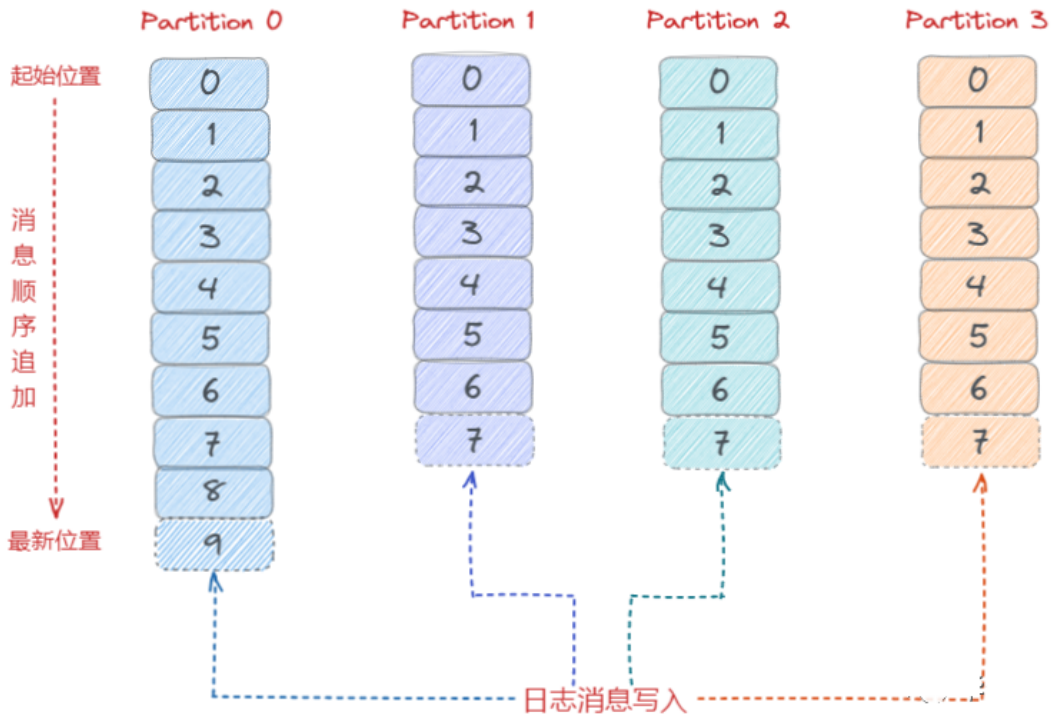
平均分割为一些相对较小的文件，这样也便于消息的查找、维护和清理。这样在做历史数据清理的时候，直接删除旧的 LogSegement 文件就可以了。

5. Log 日志在物理上只是以文件夹的形式存储，而每个 LogSegement 对应磁盘上的一个日志文件和两个索引文件，以及可能的其他文件(比如以".snapshot"为后缀的快照索引文件等)

3.4 kafka 日志系统架构设计

再来研究topic->partition的关系。

kafka 消息是按主题 Topic 为基础单位归类的，各个 Topic 在逻辑上是独立的，每个 Topic 又可以分为一个或者多个 Partition，每条消息在发送的时候会根据分区规则被追加到指定的分区中，如下图所示：



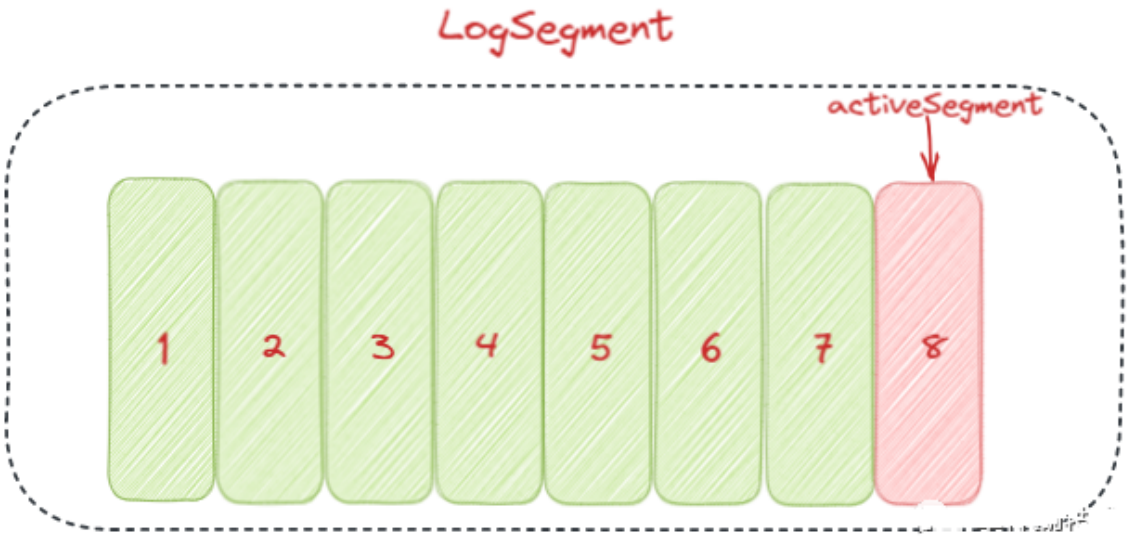
4个分区的主题逻辑结构图

3.4.1 日志目录布局

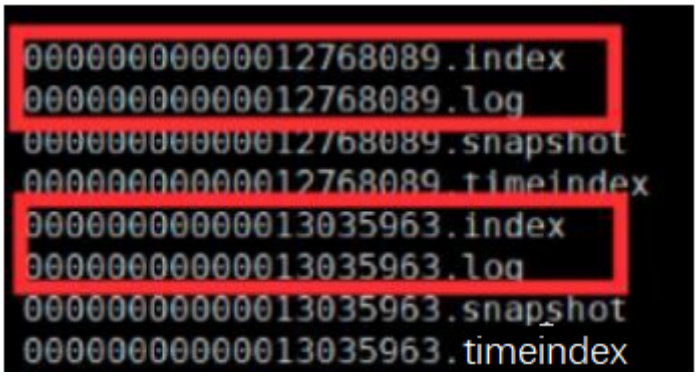
Kafka 消息写入到磁盘的日志目录布局是怎样的?

Log 对应了一个命名为-的文件夹。举个例子，假设现在有一个名为“topic-order”的 Topic，该 Topic 中有4个 Partition，那么在实际物理存储上表现为“topic-order-0”、“topic-order-1”、“topic-order-2”、“topic-order-3” 这4个文件夹。

Log 中写入消息是顺序写入的。**但是只有最后一个 LogSegement 才能执行写入操作**，之前的所有 LogSegement 都不能执行写入操作。为了更好理解这个概念，我们将最后一个 LogSegement 称为**"activeSegement"，即表示当前活跃的日志分段**。随着消息的不断写入，当 activeSegement 满足一定的条件时，就需要创建新的 activeSegement，之后再追加的消息会写入新的 activeSegement。



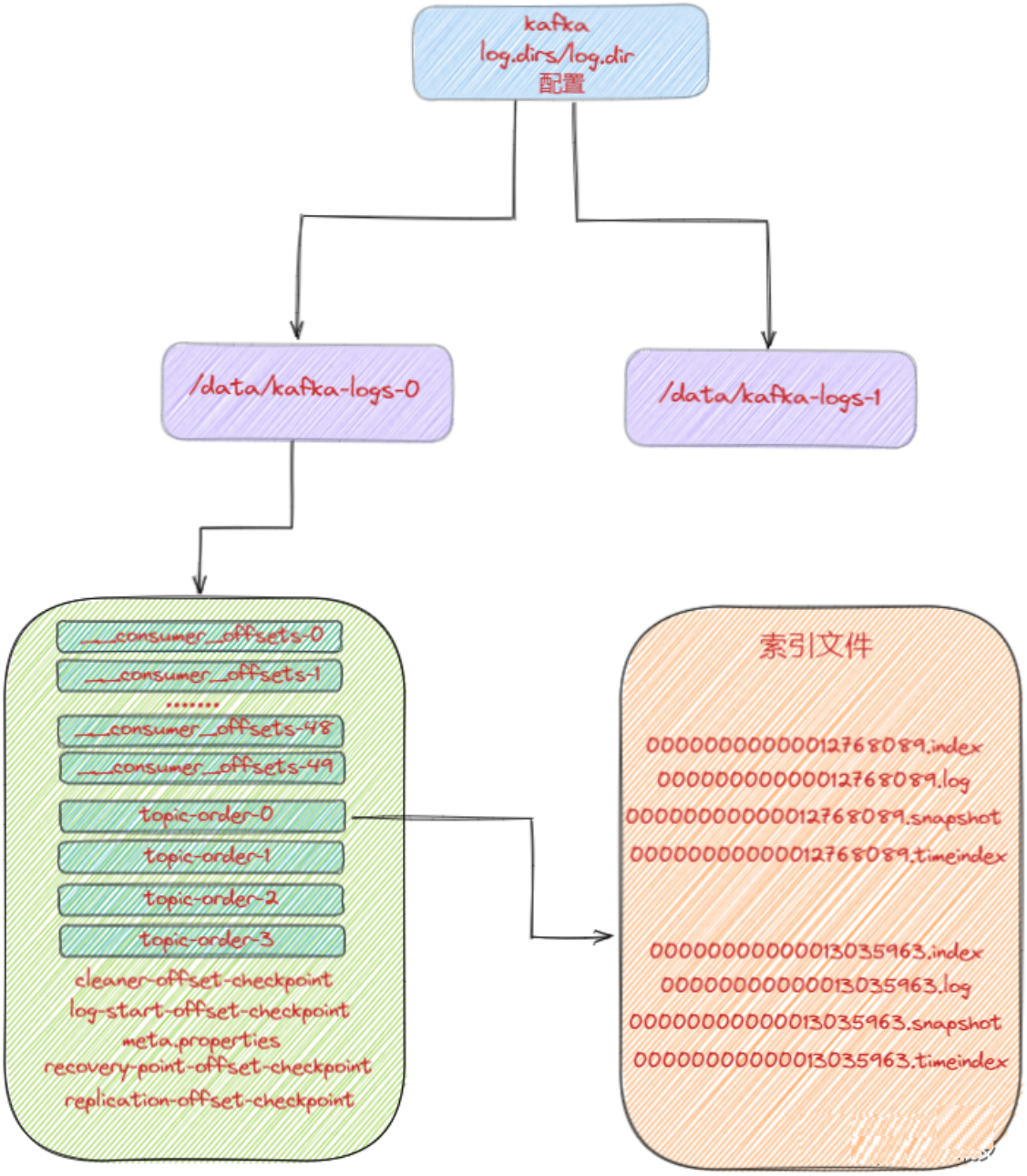
为了更高效的进行消息检索，每个 LogSegment 中的日志文件（以“.log”为文件后缀）都有对应的几个索引文件：**偏移量索引文件（以“.index”为文件后缀）、时间戳索引文件（以“.timeindex”为文件后缀）、快照索引文件 （以“.snapshot”为文件后缀）**。其中每个 LogSegment 都有一个 Offset 来作为基准偏移量（baseOffset），用来表示当前 LogSegment 中第一条消息的 Offset。偏移量是一个64位的 Long 长整型数，日志文件和这几个索引文件都是根据基准偏移量（baseOffset）命名的，名称固定为20位数字，没有达到的位数前面用0填充。比如第一个 LogSegment 的基准偏移量为0，对应的日志文件为00000000000000000000.log。



上面例子中 LogSegment 对应的基准位移是12768089，也说明了当前 LogSegment 中的第一条消息的偏移量为12768089，同时可以说明当前 LogSegment 中共有12768089条消息（偏移量从0至12768089的消息）。

> 注意每个 LogSegment 中不只包含“.log”、“.index”、“.timeindex”这几种文件，还可能包含“.snapshot”、“.txnindex”、“leader-epoch-checkpoint”等文件, 以及 “.deleted”、“.cleaned”、“.swap”等临时文件。

消费者消费的时候，会将提交的位移保存在 Kafka 内部的主题__consumer_offsets中，下面来看一个整体的日志目录结构图：



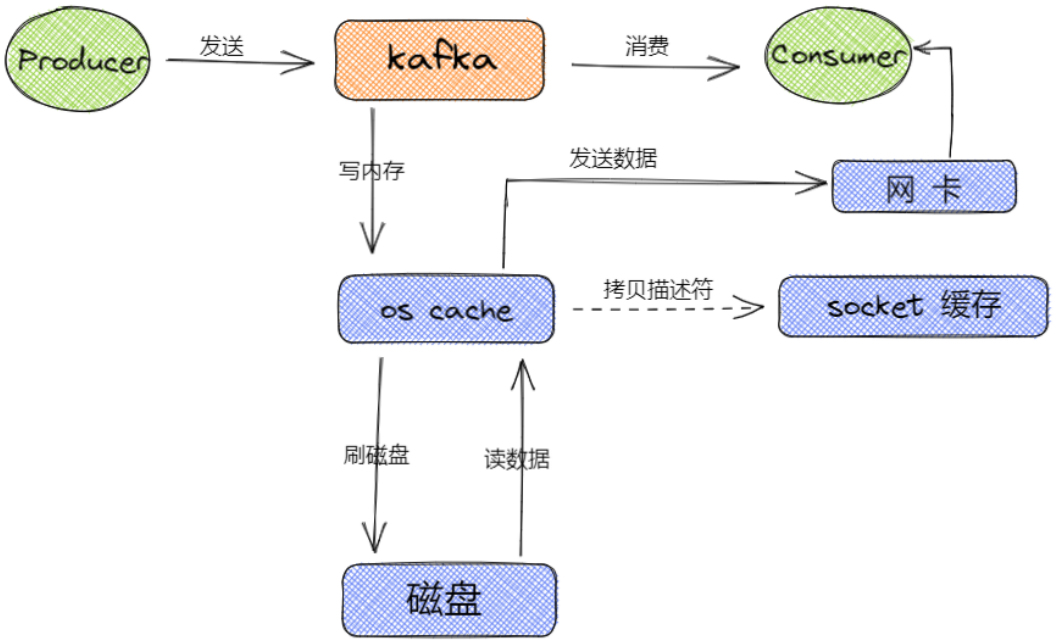
log 整体目录布局示意图

3.4.2 磁盘数据存储

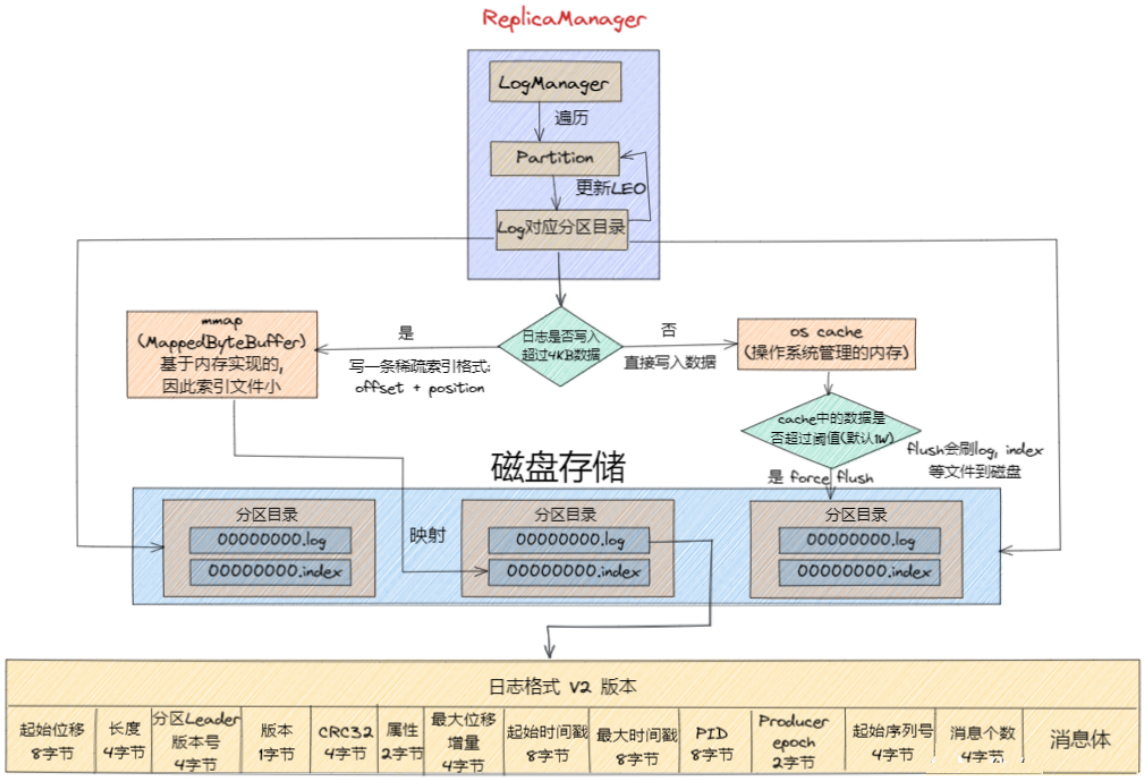
我们知道 Kafka 是依赖文件系统来存储和缓存消息，以及典型的顺序追加写日志操作，另外它使用操作系统的 PageCache 来减少对磁盘 I/O 操作，即将磁盘的数据缓存到内存中，把对磁盘的访问转变为对内存的访问。

在 Kafka 中，大量使用了 PageCache，这也是 Kafka 能实现高吞吐的重要因素之一，当一个进程准备读取磁盘上的文件内容时，操作系统会先查看待读取的数据页是否在 PageCache 中，如果命中则直接返回数据，从而避免了对磁盘的 I/O 操作；如果没有命中，操作系统则会向磁盘发起读取请求并将读取的数据页存入 PageCache 中，之后再将数据返回给进程。同样，如果一个进程需要将数据写入磁盘，那么操作系统也会检查数据页是否在页缓存中，如果不存在，则 PageCache 中添加相应的数据页，最后将数据写入对应的数据页。被修改过后的数据页也就变成了脏页，操作系统会在合适的时间把脏页中的数据写入磁盘，以保持数据的一致性。

除了消息顺序追加写日志、PageCache以外， kafka 还使用了零拷贝（Zero-Copy）技术来进一步提升系统性能， 如下图所示：



消息从生产到写入磁盘的整体过程如下图所示：



4 可靠性

可靠性相关的问题：

- 我发消息的时候，需要等 ack 嘛?
- 我发了消息之后，消费者一定会收到嘛?

- 申请腾讯云的 kafka 实例后，各种参数怎么设置呀?
- 遇到各种故障时，我的消息会不会丢?
- 消费者侧会收到多条消息嘛？消费者 svr 重启后消息会丢失嘛?

4.1 Producer的可靠性保证

回答生产者的可靠性保证，即回答：

1. 发消息之后有么有 ack
2. 发消息收到 ack 后，是不是消息就不会丢失了而 Kafka 通过配置来指定 producer 生产者在发送消息时的 ack 策略：

Request.required.acks=-1 (全量同步确认，强可靠性保证)
Request.required.acks = 1(leader 确认收到, 默认)
Request.required.acks = 0 (不确认，但是吞吐量大)

如果想实现 kafka 配置为 CP(Consistency & Partition tolerance) 系统, 配置需要如下:

request.required.acks=-1
min.insync.replicas = ${N/2 + 1}
unclean.leader.election.enable = false

4.2 Broker 的可靠性保证

消息通过 producer 发送到 broker 之后，还会遇到很多问题：

- Partition leader 写入成功， follower 什么时候同步?
- Leader 写入成功，消费者什么时候能读到这条消息?
- Leader 写入成功后，leader 重启，重启后消息状态还正常嘛?
- Leader 重启，如何选举新的 leader?

这些问题集中在， 消息落到 broker 后，集群通过何种机制来保证不同副本建的消息状态一致性。

4.3 Consumer 的可靠性策略

Consumer 的可靠性策略集中在 consumer 的投递语义上，即：

- 何时消费，消费到什么?
- 按消费是否会丢?
- 消费是否会重复?

这些语义场景，可以通过 kafka 消费者的而部分参数进行配置，简单来说有以下 3 中场景：

1. AutoCommit（at most once, commit 后挂，实际会丢）

enable.auto.commit = true

auto.commit.interval.ms

配置如上的 consumer 收到消息就返回正确给 brocker, 但是如果业务逻辑没有走完中断了，实际上这个消息没有消费成功。这种场景适用于可靠性要求不高的业务。其中 auto.commit.interval.ms 代表了自动提交的间隔。比如设置为 1s 提交 1 次，那么在 1s 内的故障重启，会从当前消费 offset 进行重新消费时，1s 内未提交但是已经消费的 msg, 会被重新消费到。

2. 手动 Commit（at least once, commit 前挂，就会重复, 重启还会丢）

enable.auto.commit = false

配置为手动提交的场景下，业务开发者需要在消费消息到消息业务逻辑处理整个流程完成后进行手动提交。如果在流程未处理结束时发生重启，则之前消费到未提交的消息会重新消费到，即消息显然会投递多次。此处应用与业务逻辑明显实现了幂等的场景下使用。

特别应关注到在 golang 中 sarama 库的几个参数的配置：

sarama.offset.initial （oldest, newest）
offsets.retention.minutes

intitial = oldest 代表消费可以访问到的 topic 里的最早的消息，大于 commit 的位置，但是小于 HW。同时也受到 broker 上消息保留时间的影响和位移保留时间的影响。不能保证一定能消费到 topic 起始位置的消息。

如果设置为 newest 则代表访问 commit 位置的下一条消息。如果发生 consumer 重启且 autocommit 没有设置为 false, 则之前的消息会发生丢失，再也消费不到了。在业务环境特别不稳定或非持久化 consumer 实例的场景下，应特别注意。

一般情况下， offsets.retention.minutes 为 1440s。

3. Exactly once, 很难，需要 msg 持久化和 commit 是原子的

消息投递且仅投递一次的语义是很难实现的。首先要消费消息并且提交保证不会重复投递，其次提交前要完成整体的业务逻辑关于消息的处理。在 kafka 本身没有提供此场景语义接口的情况下，这几乎是不可能有效实现的。一般的解决方案，也是进行原子性的消息存储，业务逻辑异步慢慢的从存储中取出消息进行处理。

5 消费组Reblance

5.1 消费者组

消费组指的是多个消费者组成起来的一个组，它们共同消费 topic 的所有消息，并且一个 topic 的一个 partition 只能被一个 consumer 消费。其实reblance就是为了kafka对提升消费效率做的优化，规定了一个ConsumerGroup下的所有consumer均匀分配订阅 Topic 的每个分区。

例如：某 Group 下有 20 个 consumer 实例，它订阅了一个具有 100 个 partition 的 Topic 。正常情况下，kafka 会为每个 Consumer 平均的分配 5 个分区。这个分配的过程就是 Rebalance。

5.2 rebalance的影响

每次reblance会把所有的消费者重新分配监听topic，会产生一定影响

- 可能重复消费: Consumer被踢出消费组，可能还没有提交offset，Rebalance时会Partition重新分配其它Consumer,会造成重复消费，虽有幂等性限制，但是会增加负担。
- 集群不稳定：Rebalance扩散到整个ConsumerGroup的所有消费者，因为一个消费者的退出，导致整个Group进行了Rebalance，影响面较大
- 影响消费速度：频繁的Rebalance反而降低了消息的消费速度，大部分时间都在重复消费和Rebalance

其实我在流式图表里只遇到过几次reblance的情况，原因是默认超时时间设置的太短了，因为每个topic只有一个consumer，所以这一个consumer group里只有一个consumer，只要合理设置超时时间显然不会发生reblance。

5.3 rebalance处理思路

出发reblance的三种情况分别是

- 新consumer加入
- 组内consumer主动离开
- 组内consumer成员崩溃

前两个一般不会出现，大部分都是由第三种情况导致的，首先明确消费者的四个参数。

- session.timeout.ms 表示 consumer 向 broker 发送心跳的超时时间。例如 session.timeout.ms = 180000 表示在最长 180 秒内 broker 没收到 consumer 的心跳，那么 broker 就认为该 consumer 死亡了，会启动 rebalance。
- heartbeat.interval.ms 表示 consumer 每次向 broker 发送心跳的时间间隔。heartbeat.interval.ms = 60000 表示 consumer 每 60 秒向 broker 发送一次心跳。一般来说，session.timeout.ms 的值是 heartbeat.interval.ms 值的 3 倍以上。
- max.poll.interval.ms 表示 consumer 每两次 poll 消息的时间间隔。简单地说，其实就是 consumer 每次消费消息的时长。如果消息处理的逻辑很重，那么市场就要相应延长。否则如果时间到了 consumer 还么消费完，broker 会默认认为 consumer 死了，发起 rebalance。
- max.poll.records 表示每次消费的时候，获取多少条消息。获取的消息条数越多，需要处理的时间越长。所以每次拉取的消息数不能太多，需要保证在 max.poll.interval.ms 设置的时间内能消费完，否则会发生 rebalance。

总结两点就是可能会导致崩溃的点就是，消费者心跳超时和消费者消费数据超时

解决方法已经很明朗了，就是适当调参，心跳超时就调整session.timeout.ms和heartbeat.interval.ms.

对于心跳超时问题。一般是调高心跳超时时间（session.timeout.ms）和心跳间隔时间（heartbeat.interval.ms）的比例。阿里云官方文档建议超时时间（session.timeout.ms）设置成 25s，最长不超过 30s。那么心跳间隔时间（heartbeat.interval.ms）就不超过 10s。

对于消费处理超时问题。一般是增加消费者处理的时间（max.poll.interval.ms），减少每次处理的消息数（max.poll.records）。阿里云官方文档建议 max.poll.records 参数要远小于当前消费组的消费能力（records < 单个线程每秒消费的条数 * 消费线程的个数 * session.timeout的秒数）。