四、Kafka日志索引详解

-- 楼兰

上一章节Kafka的核心集群机制，重点保证了在复杂运行环境下，整个Kafka集群如何保证Partition内消息的一致性。这就相当于一个军队，有了完整统一的编制。但是，在进行具体业务时，还是需要各个Broker进行分工，各自处理好自己的工作。

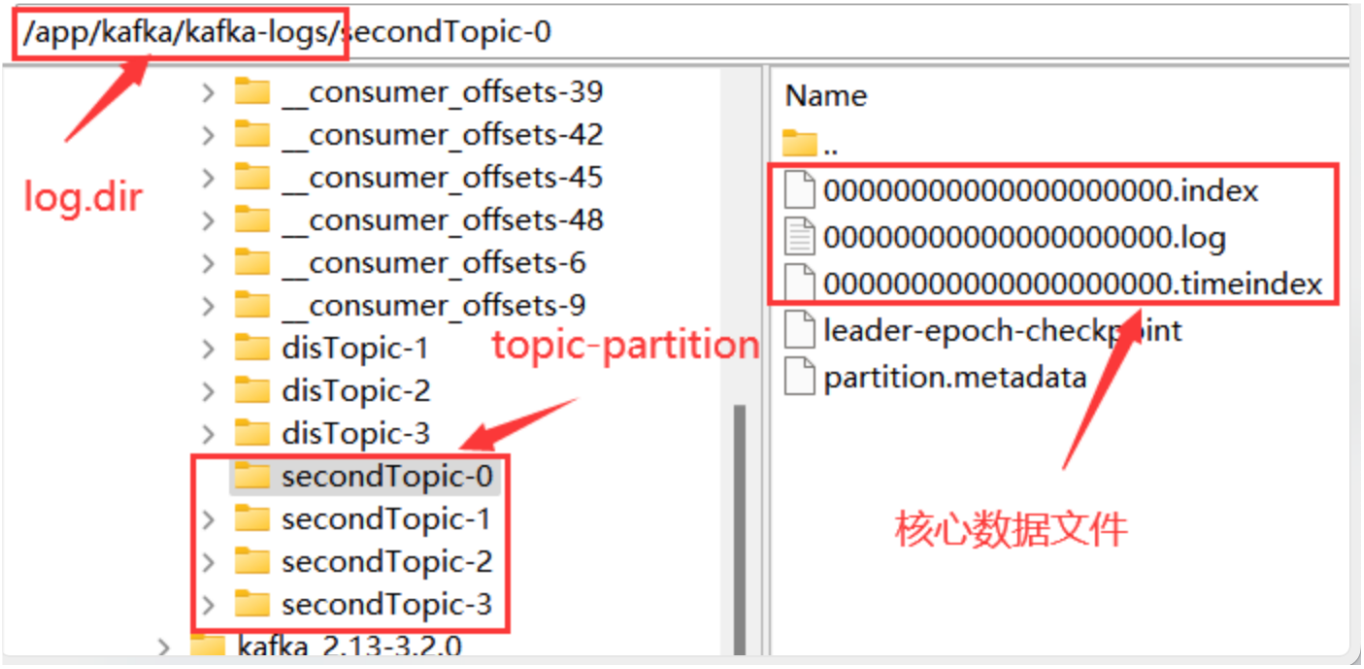
每个Broker如何高效的处理以及保存消息，也是Kafka高性能背后非常重要的设计。这一章节还是按照之前的方式，从可见的Log文件入手，来逐步梳理Kafka是如何进行高效消息流转的。Kafka的日志文件记录机制也是Kafka能够支撑高吞吐、高性能、高可扩展的核心所在。对于业界的影响也是非常巨大的。

这一部分数据主要包含当前Broker节点的消息数据(在Kafka中称为Log日志)。这是一部分无状态的数据，也就是说每个Kafka的Broker节点都是以相同的逻辑运行。这种无状态的服务设计让Kafka集群能够比较容易的进行水平扩展。比如你需要用一个新的Broker服务来替换集群中一个旧的Broker服务，那么只需要将这部分无状态的数据从旧的Broker上转移到新的Broker上就可以了。

> 当然，这里说的的数据转移，并不是复制，粘贴这么简单，因为底层的数据文件中的细节还是非常多的，并且是二进制文件，操作也不容易。
>
> 实际上Kafka也提供了很多工具来协助进行数据迁移，例如bin目录下的 kafka-reassign-partitions.sh都可以帮助进行服务替换。感兴趣可以使用脚本的--help指令了解一下

一、Topic下的消息是如何存储的?

在搭建Kafka服务时，我们在server.properties配置文件中通过log.dir属性指定了Kafka的日志存储目录。实际上，Kafka的所有消息就全都存储在这个目录下。



这些核心数据文件中，.log结尾的就是实际存储消息的日志文件。他的大小固定为1G(由参数log.segment.bytes参数指定)，写满后就会新增一个新的文件。一个文件也成为一个segment文件名表示当前日志文件记录的第一条消息的偏移量。

.index和.timeindex是日志文件对应的索引文件。不过.index是以偏移量为索引来记录对应的.log日志文件中的消息偏移量。而.timeindex则是以时间戳为索引。

> 另外的两个文件，partition.metadata简单记录当前Partition所属的cluster和Topic。leader-epoch-checkpoint文件参见上面的epoch机制。

这些文件都是二进制的文件，无法使用文本工具直接查看。但是，Kafka提供了工具可以用来查看这些日志文件的内容。

```
#1、查看timeIndex文件
[root@192-168-65-112 bin]# ./kafka-dump-log.sh --files /app/kafka/logs/disTopic-
0/00000000000000000000.timeindex
Dumping /app/kafka/logs/disTopic-0/00000000000000000000.timeindex
timestamp: 1723519364827 offset: 50
timestamp: 1723519365630 offset: 99
timestamp: 1723519366162 offset: 148
timestamp: 1723519366562 offset: 197
timestamp: 1723519367013 offset: 246
timestamp: 1723519367364 offset: 295
timestamp: 1723519367766 offset: 344
#2、查看index文件
[root@192-168-65-112 bin]# ./kafka-dump-log.sh --files /app/kafka/logs/disTopic-
0/00000000000000000000.index
Dumping /app/kafka/logs/disTopic-0/00000000000000000000.index
offset: 50 position: 4098
offset: 99 position: 8214
offset: 148 position: 12330
offset: 197 position: 16446
offset: 246 position: 20562
offset: 295 position: 24678
offset: 344 position: 28794
#3、查看log文件
[root@192-168-65-112 bin]# ./kafka-dump-log.sh --files /app/kafka/logs/disTopic-
0/00000000000000000000.log
Dumping /app/kafka/kafka-logs/secondTopic-0/00000000000000000000.log
Starting offset: 0
.....
baseOffset: 350 lastOffset: 350 count: 1 baseSequence: 349 lastSequence: 349 producerId:
5002 producerEpoch: 0 partitionLeaderEpoch: 7 isTransactional: false isControl: false
deleteHorizonMs: OptionalLong.empty position: 29298 CreateTime: 1723519367827 size: 84
magic: 2 compresscodec: none crc: 400306231 isvalid: true
baseOffset: 351 lastOffset: 351 count: 1 baseSequence: 350 lastSequence: 350 producerId:
5002 producerEpoch: 0 partitionLeaderEpoch: 7 isTransactional: false isControl: false
deleteHorizonMs: OptionalLong.empty position: 29382 CreateTime: 1723519367829 size: 84
magic: 2 compresscodec: none crc: 2036034757 isvalid: true
.......
```

这些数据文件的记录方式，就是我们去理解Kafka本地存储的主线。

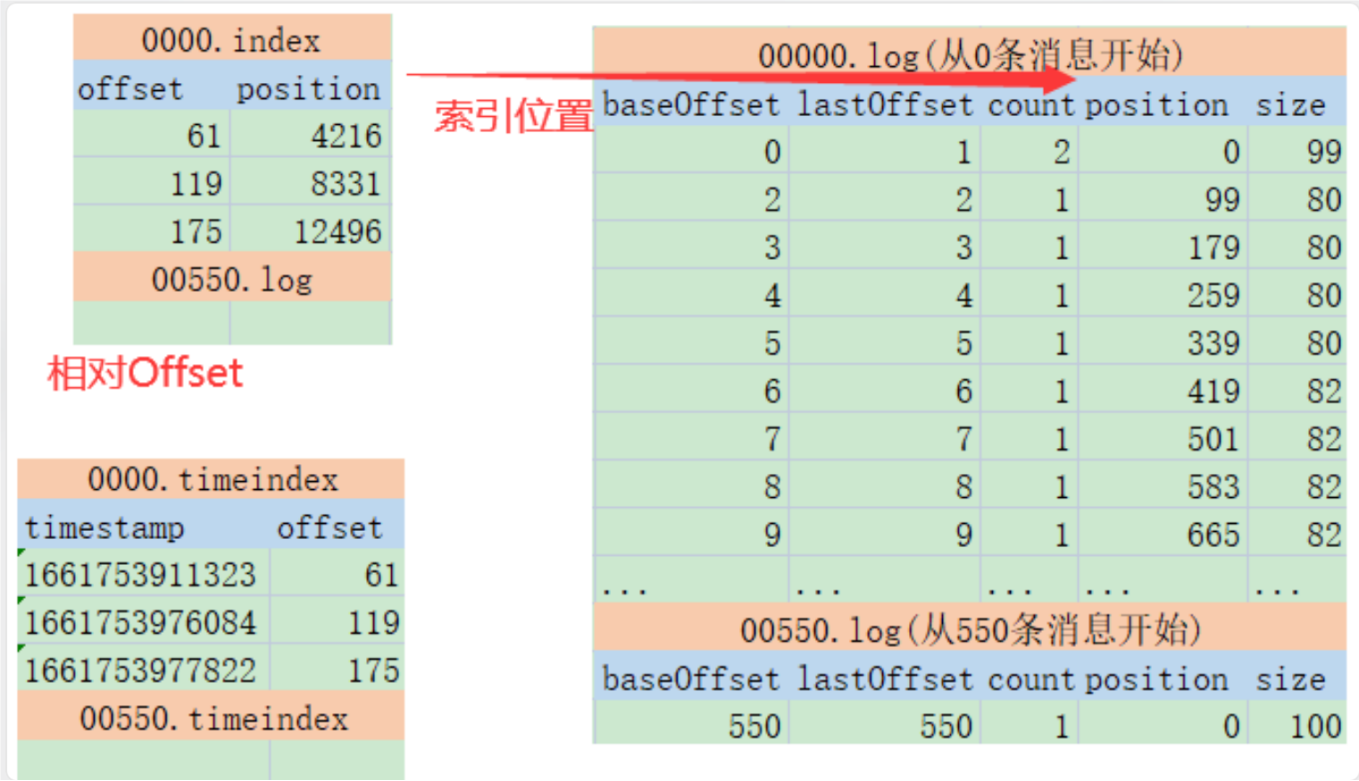
1 log文件追加记录所有消息

首先：在每个文件内部，Kafka都会以追加的方式写入到log日志文件中。Kafka中的消息日志，只允许追加，不支持删除和修改。所以，只有文件名最大的一个log文件是当前写入消息的日志文件，其他文件都是不可修改的历史日志。

然后：每个Log文件都保持固定的大小。如果当前文件记录不下了，就会重新创建一个log文件，并以这个log文件写入的第一条消息的偏移量命名。这种设计其实是为了更方便进行文件映射，加快读消息的效率。

2 index和timeindex加速读取log消息日志。

详细看下这几个文件的内容，就可以总结出Kafka记录消息日志的整体方式：



首先：index和timeindex都是以相对偏移量的方式建立log消息日志的数据索引。比如说 0000.index和0550.index中记录的索引数字，都是从0开始的。表示相对日志文件起点的消息偏移量。而绝对的消息偏移量可以通过日志文件名 + 相对偏移量得到。

然后：这两个索引并不是对每一条消息都建立索引。而是Broker每写入40KB的数据，就建立一条index索引。由参数log.index.interval.bytes定制。

```
log.index.interval.bytes
The interval with which we add an entry to the offset index

Type:   int
Default:    4096 (4 kibibytes)
Valid Values:   [0,...]
Importance: medium
Update Mode:    cluster-wide
```

index文件的作用类似于数据结构中的跳表，他的作用是用来加速查询log文件的效率。而timeindex文件的作用则是用来进行一些跟时间相关的消息处理。比如文件清理。

这两个索引文件也是Kafka的消费者能够指定从某一个offset或者某一个时间点读取消息的原因。

二、文件清理机制

Kafka为了防止过多的日志文件给服务器带来过大的压力，他会定期删除过期的log文件。Kafka的删除机制涉及到几组配置属性：

1、如何判断哪些日志文件过期了

- log.retention.check.interval.ms：定时检测文件是否过期。默认是 300000毫秒，也就是五分钟。
- log.retention.hours ， log.retention.minutes， log.retention.ms 。 这一组参数表示文件保留多长时间。默认生效的是log.retention.hours，默认值是168小时，也就是7天。如果设置了更高的时间精度，以时间精度最高的配置为准。
- 在检查文件是否超时时，是以每个.timeindex中最大的那一条记录为准。

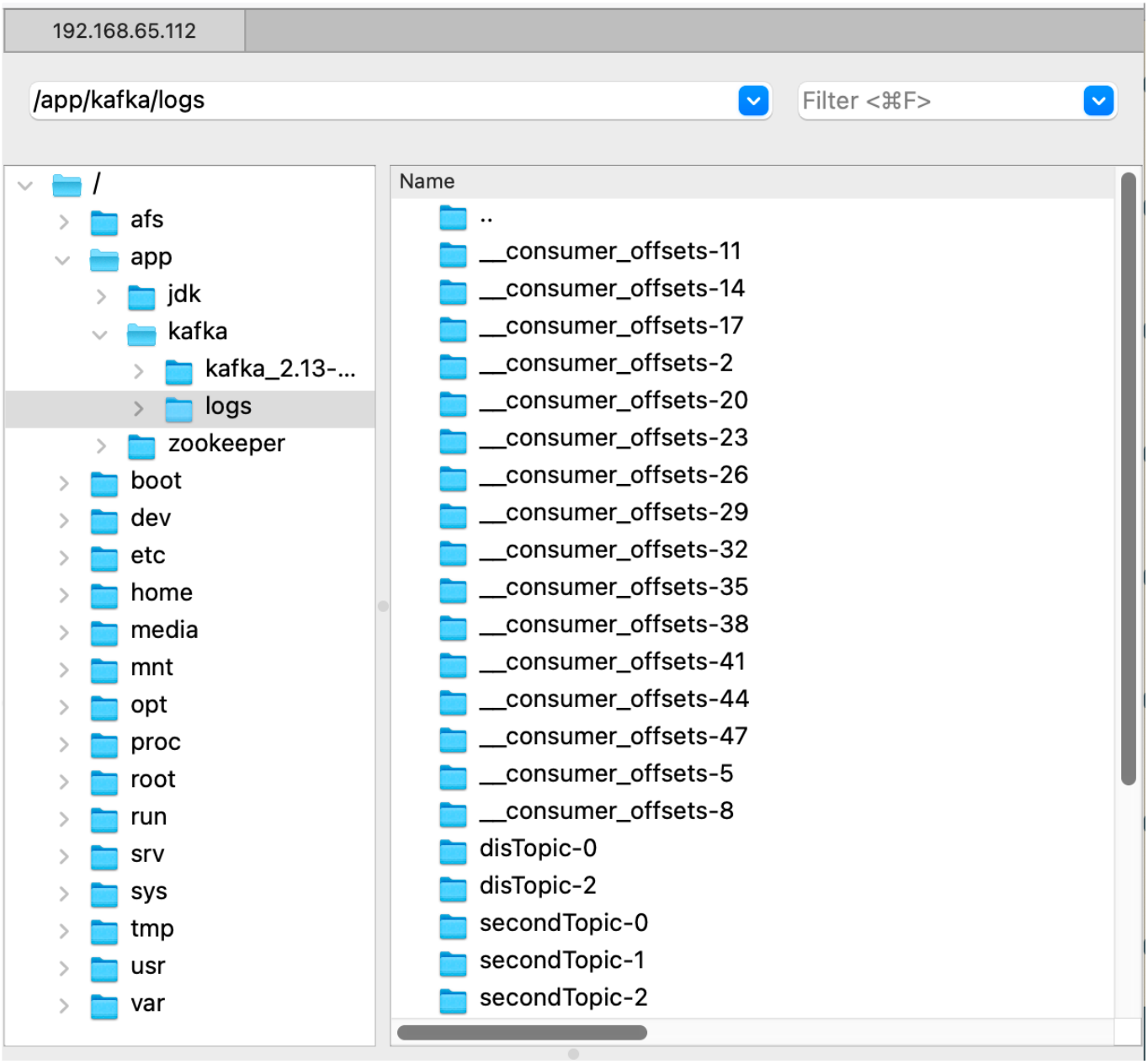
2、过期的日志文件如何处理

- log.cleanup.policy：日志清理策略。有两个选项，delete表示删除日志文件。 compact表示压缩日志文件。
- 当log.cleanup.policy选择delete时，还有一个参数可以选择。log.retention.bytes：表示所有日志文件的大小。当总的日志文件大小超过这个阈值后，就会删除最早的日志文件。默认是-1，表示无限大。

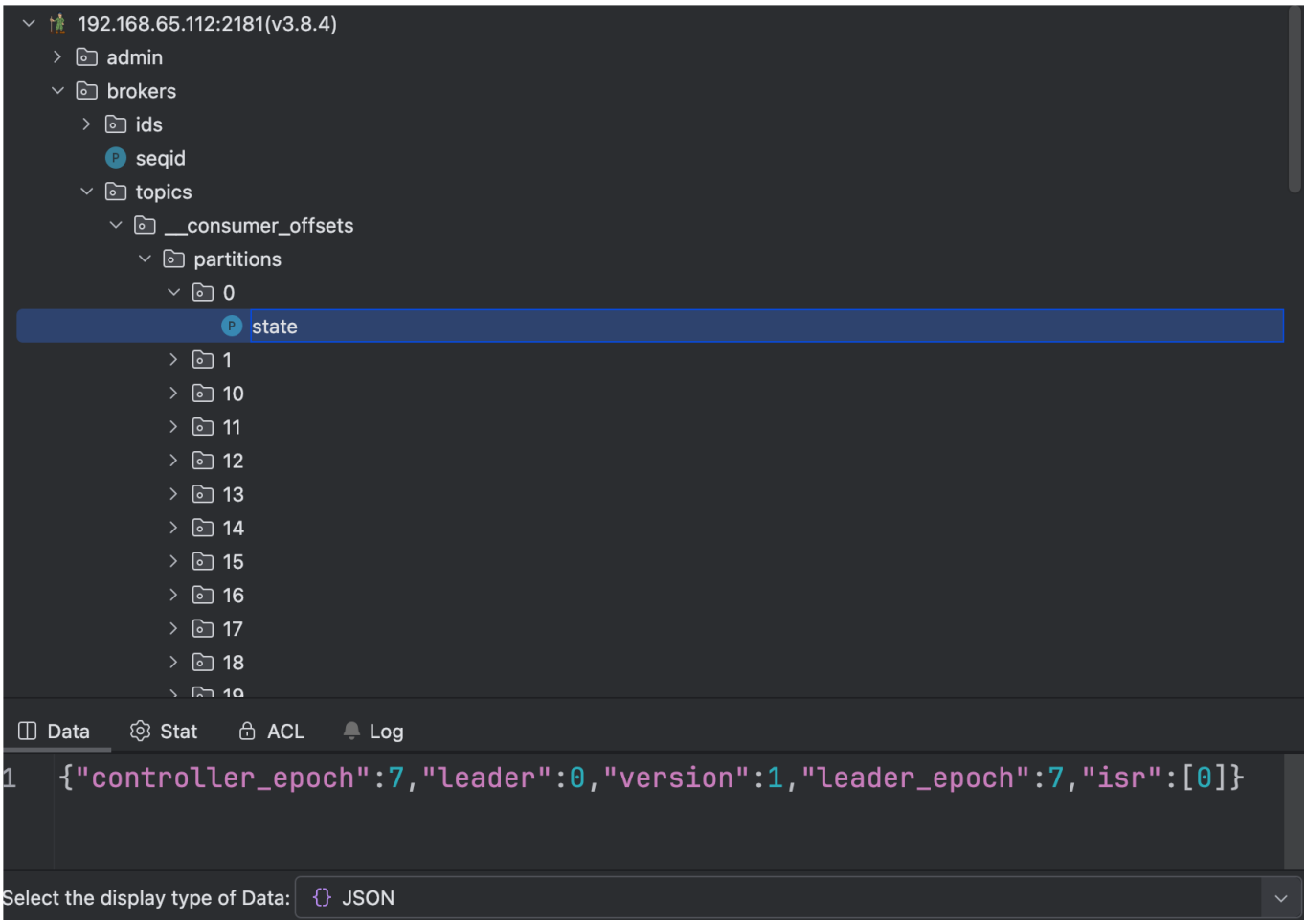
压缩日志文件虽然不会直接删除日志文件，但是会造成消息丢失。压缩的过程中会将key相同的日志进行压缩，只保留最后一条。

三、客户端消费进度管理

kafka为了实现分组消费的消息转发机制，需要在Broker端保持每个消费者组的消费进度。而这些消费进度，就被Kafka管理在自己的一个内置Topic中。这个Topic就是__consumer__offsets。这是Kafka内置的一个系统Topic，在日志文件可以看到这个Topic的相关目录。Kafka默认会将这个Topic划分为50个分区。



同时，Kafka也会将这些消费进度的状态信息记录到Zookeeper中。



这个系统Topic中记录了所有ConsumerGroup的消费进度。那他的数据是怎么保存的呢？在Zookeeper中似乎并没有记载Offset数据啊。

既然他是Kafka的一个Topic，那消费者是不是可以直接消费其中的消息？

这个Topic是Kafka内置的一个系统Topic，可以启动一个消费者订阅这个Topic中的消息。

```
[root@192-168-65-112 kafka_2.13-3.8.0]# bin/kafka-console-consumer.sh --topic
__consumer_offsets --bootstrap-server worker1:9092 --consumer.config
config/consumer.properties --formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter" --from-beginning
```

查看到结果：

```
[test,disTopic,1]::OffsetAndMetadata(offset=3, leaderEpoch=Optional[1], metadata=,
commitTimestamp=1661351768150, expireTimestamp=None)
[test,disTopic,2]::OffsetAndMetadata(offset=0, leaderEpoch=Optional.empty, metadata=,
commitTimestamp=1661351768150, expireTimestamp=None)
[test,disTopic,0]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[2], metadata=,
commitTimestamp=1661351768150, expireTimestamp=None)
[test,disTopic,3]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[3], metadata=,
commitTimestamp=1661351768151, expireTimestamp=None)
[test,disTopic,1]::OffsetAndMetadata(offset=3, leaderEpoch=Optional[1], metadata=,
commitTimestamp=1661351768151, expireTimestamp=None)
[test,disTopic,2]::OffsetAndMetadata(offset=0, leaderEpoch=Optional.empty, metadata=,
commitTimestamp=1661351768151, expireTimestamp=None)
[test,disTopic,0]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[2], metadata=,
commitTimestamp=1661351768151, expireTimestamp=None)
[test,disTopic,3]::OffsetAndMetadata(offset=6, leaderEpoch=Optional[3], metadata=,
commitTimestamp=1661351768153, expireTimestamp=None)
[test,disTopic,1]::OffsetAndMetadata(offset=3, leaderEpoch=Optional[1], metadata=,
commitTimestamp=1661351768153, expireTimestamp=None)
[test,disTopic,2]::OffsetAndMetadata(offset=0, leaderEpoch=Optional.empty, metadata=,
commitTimestamp=1661351768153, expireTimestamp=None)
```

从这里可以看到，Kafka也是像普通数据一样，以Key-Value的方式来维护消费进度。key是groupid+topic+partition，value则是表示当前的offset。

而这些Offset数据，其实也是可以被消费者修改的，在之前章节已经演示过消费者如何从指定的位置开始消费消息。而一旦消费者主动调整了Offset，Kafka当中也会更新对应的记录。

在早期版本中，Offset确实是存在Zookeeper中的。但是Kafka在很早就选择了将Offset从Zookeeper中转移到Broker上。这也体现了Kafka其实早就意识到，Zookeeper这样一个外部组件在面对三高问题时，是不太"靠谱"的，所以Kafka逐渐转移了Zookeeper上的数据。而后续的Kraft集群，其实也是这种思想的延伸。

另外，这个系统Topic里面的数据是非常重要的，因此Kafka在消费者端也设计了一个参数来控制这个Topic应该从订阅关系中剔除。

```
    public static final String EXCLUDE_INTERNAL_TOPICS_CONFIG = "exclude.internal.topics";
    private static final String EXCLUDE_INTERNAL_TOPICS_DOC = "Whether internal topics
matching a subscribed pattern should " +
            "be excluded from the subscription. It is always possible to explicitly
subscribe to an internal topic.";
    public static final boolean DEFAULT_EXCLUDE_INTERNAL_TOPICS = true;
```

这个参数简单测试了一下，在当前版本是没有用的。

四、Kafka的文件高效读写机制

这是Kafka非常重要的一个设计，同时也是面试频率超高的问题。可以分几个方向来理解。

1、Kafka的文件结构

Kafka的数据文件结构设计可以加速日志文件的读取。比如同一个Topic下的多个Partition单独记录日志文件，并行进行读取，这样可以加快Topic下的数据读取速度。然后index的稀疏索引结构，可以加快log日志检索的速度。

2、顺序写磁盘

这个跟操作系统有关，主要是硬盘结构。

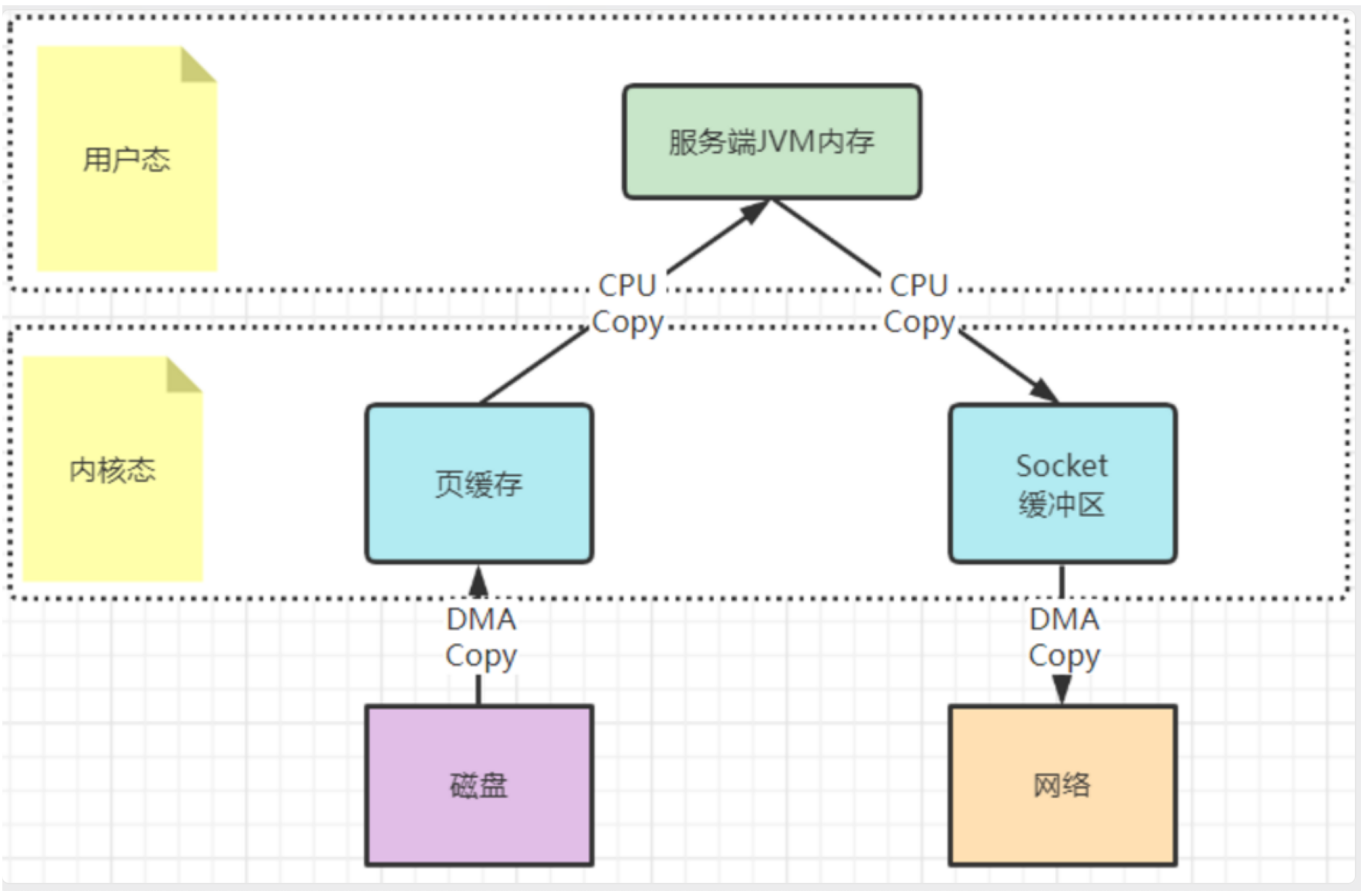
对每个Log文件，Kafka会提前规划固定的大小，这样在申请文件时，可以提前占据一块连续的磁盘空间。然后，Kafka的log文件只能以追加的方式往文件的末端添加(这种写入方式称为顺序写)，这样，新的数据写入时，就可以直接往直前申请的磁盘空间中写入，而不用再去磁盘其他地方寻找空闲的空间(普通的读写文件需要先寻找空闲的磁盘空间，再写入。这种写入方式称为随机写)。由于磁盘的空闲空间有可能并不是连续的，也就是说有很多文件碎片，所以磁盘写的效率会很低。

kafka的官网有测试数据，表明了同样的磁盘，顺序写速度能达到600M/s，基本与写内存的速度相当。而随机写的速度就只有100K/s，差距比加大。

3、零拷贝

零拷贝是Linux操作系统提供的一种IO优化机制，而Kafka大量的运用了零拷贝机制来加速文件读写。

传统的一次硬件IO是这样工作的。如下图所示：



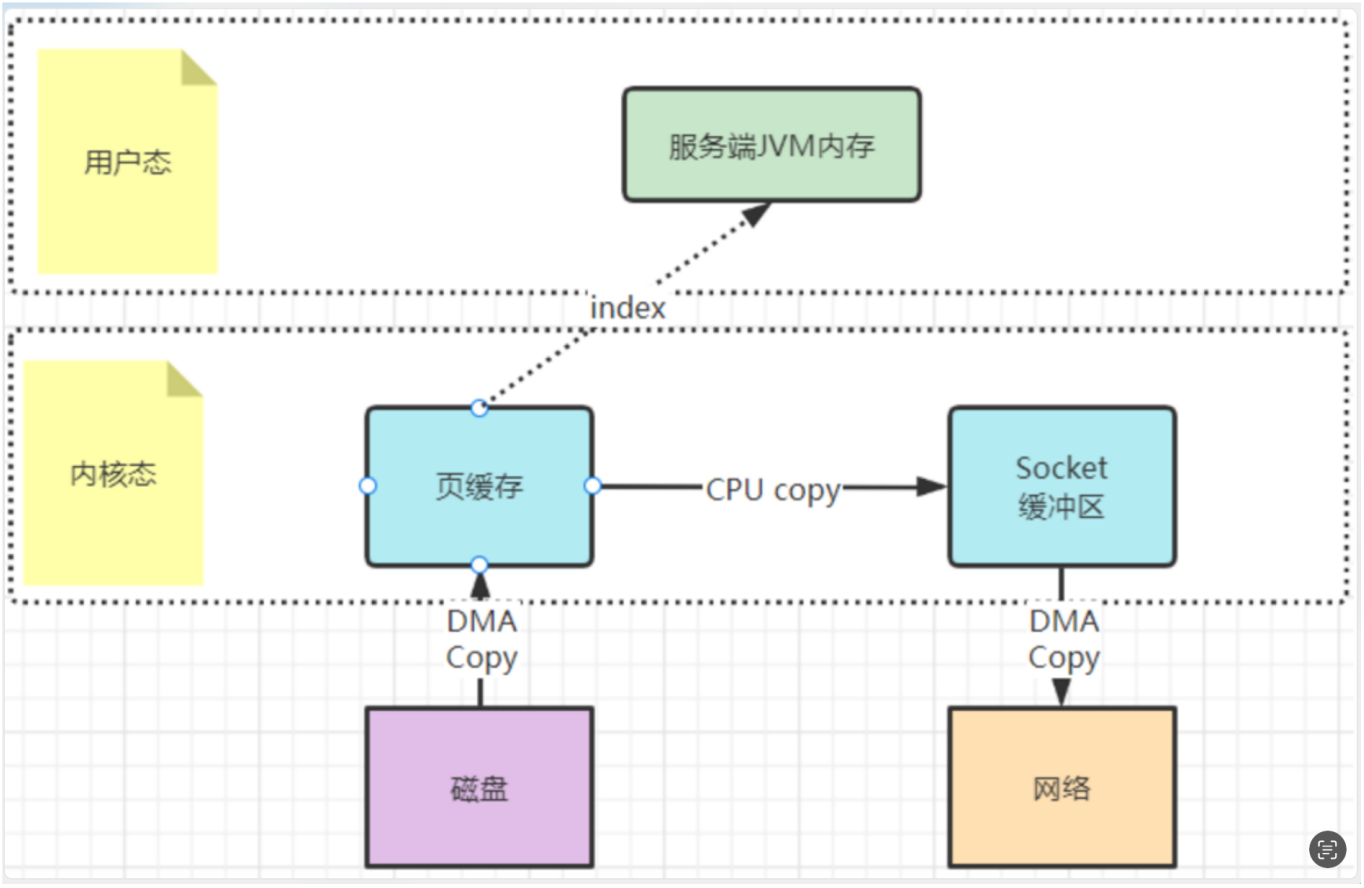
其中，内核态的内容复制是在内核层面进行的，而零拷贝的技术，重点是要配合内核态的复制机制，减少用户态与内核态之间的内容拷贝。

具体实现时有两种方式：

1、mmap文件映射机制

这种方式是在用户态不再缓存整个IO的内容，改为只持有文件的一些映射信息。通过这些映射，"遥控"内核态的文件读写。这样就减少了内核态与用户态之间的拷贝数据大小，提升了IO效率。

> 这都说的是些什么？去参考下JDK中的DirectByteBuffer实现机制吧。



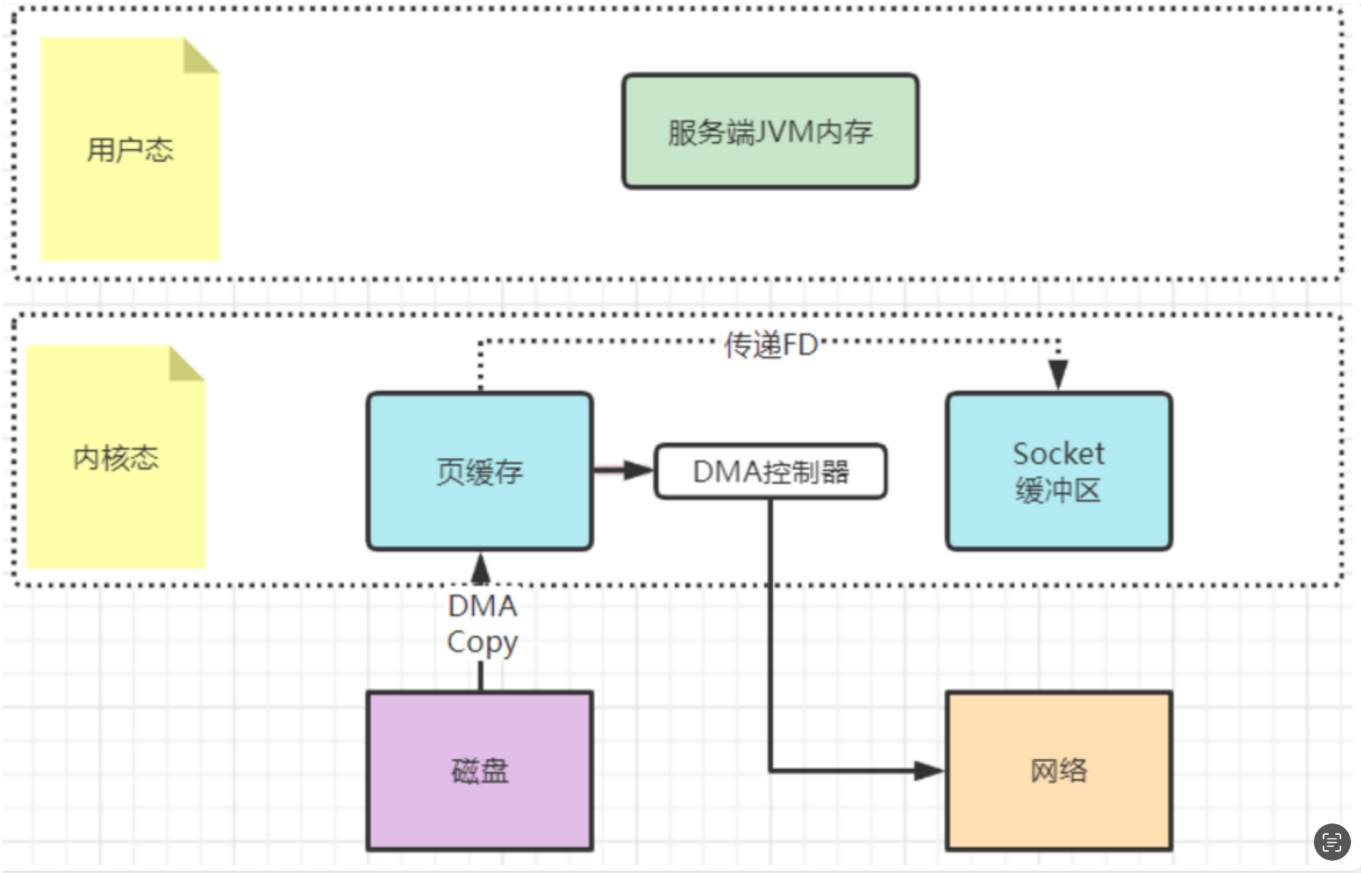
> mmap文件映射机制是操作系统提供的一种文件操作机制，可以使用man 2 mmap查看。实际上在Java程序执行过程当中就会被大量使用。

这种mmap文件映射方式，适合于操作不是很大的文件，通常映射的文件不建议超过2G。所以kafka将.log日志文件设计成1G大小，超过1G就会另外再新写一个日志文件。这就是为了便于对文件进行映射，从而加快对.log文件等本地文件的写入效率。

2、sendfile文件传输机制

这种机制可以理解为用户态，也就是应用程序不再关注数据的内容，只是向内核态发一个sendfile指令，要他去复制文件就行了。这样数据就完全不用复制到用户态，从而实现了零拷贝。

> 相比mmap，连索引都不读了，直接通知操作系统去拷贝就是了。好处，自然是效率更高了。但是坏处是在用户态对文件内容完全无感知，也就是说无法在用户态中对文件内容做解析。



例如在Kafka中，当Consumer要从Broker上poll消息时，Broker需要读取自己本地的数据文件，然后通过网卡发送给Consumer。这个过程当中，Broker只负责传递消息，而不对消息进行任何的加工。所以Broker只需要将数据从磁盘读取出来，复制到网卡的Socket缓冲区，然后通过网络发送出去。这个过程当中，用户态就只需要往内核态发一个sendfile指令，而不需要有任何的数据拷贝过程。Kafka大量的使用了sendfile机制，用来加速对本地数据文件的读取过程。

> 具体细节可以在linux机器上使用man 2 sendfile指令查看操作系统的帮助文件。
>
> ```
> SENDFILE(2)                                                          Linux
> Programmer's Manual
> SENDFILE(2)
> NAME
>     sendfile - transfer data between file descriptors
> SYNOPSIS
>         ......
>     In Linux kernels before 2.6.33, out_fd must refer to a socket.  Since Linux
> 2.6.33 it can be any file.  If it is a regular file,  then  sendfile()  changes  the
> file  offset
>     appropriately.
>
> RETURN VALUE
>     If the transfer was successful, the number of bytes written to out_fd is
> returned.  On error, -1 is returned, and errno is set appropriately.
> ```
>
> JDK中8中java.nio.channels.FileChannel类提供了transferTo和transferFrom方法，底层就是使用了操作系统的sendfile机制。

这些底层的优化机制都是操作系统提供的优化机制，其实针对任何上层应用语言来说，都是一个黑盒，只能去调用，但是控制不了具体的实现过程。而上层的各种各样的语言，也只能根据操作系统提供的支持进行自己的实现。虽然不同语言的实现方式会有点不同，但是本质都是一样的。

4、合理配置刷盘频率

缓存数据断电就会丢失，这是大家都能理解的，所以缓存中的数据如果没有及时写入到硬盘，也就是常说的刷盘，那么当服务突然崩溃，就会有丢消息的可能。所以，最安全的方式是写一条数据，就刷一次盘，成为同步刷盘。刷盘操作在Linux系统中对应了一个fsync的系统调用。

```
FSYNC(2)                                                                  Linux
Programmer's Manual
FSYNC(2)

NAME
       fsync, fdatasync – synchronize a file's in-core state with storage device
```

但是，这里真正容易产生困惑的，是这里所提到的in-core state。这并不是我们平常开发过程中接触到的缓存，而是操作系统内核态的缓存-pageCache。这是应用程序接触不到的一部分缓存。比如我们用应用程序打开一个文件，实际上文件里的内容，是从内核态的PageCache中读取出来的。因为与磁盘这样的硬件交互，相比于内存，效率是很低的。操作系统为了提升性能，会将磁盘中的文件加载到PageCache缓存中，再向应用程序提供数据。修改文件时也是一样的。用记事本修改一个文件的内容，不管你保存多少次，内容都是写到PageCache里的。然后操作系统会通过他自己的缓存管理机制，在未来的某个时刻将所有的PageCache统一写入磁盘。这个操作就是刷盘。比如在操作系统正常关系的过程中，就会触发一次完整的刷盘机制。

说这么多，就是告诉你，其实对于缓存断掉，造成数据丢失，这个问题，应用程序其实是没有办法插手的。他并不能够决定自己产生的数据在什么时候刷入到硬盘当中。应用程序唯一能做的，就是尽量频繁的通知操作系统进行刷盘操作。但是，这必然会降低应用的执行性能，而且，也不是能百分之百保证数据安全的。应用程序在这个问题上，只能取舍，不能解决。

Kafka其实在Broker端设计了一系列的参数，来控制刷盘操作的频率。如果对这些频率进行深度定制，是可以实现来一个消息就进行一次刷盘的“同步刷盘”效果的。但是，这样的定制显然会大大降低Kafka的执行效率，这与Kafka的设计初衷是不符合的。所以，在实际应用时，我们通常也只能根据自己的业务场景进行权衡。

Kafka在服务端设计了几个参数，来控制刷盘的频率：

- flush.ms : 多长时间进行一次强制刷盘。

  flush.ms
  This setting allows specifying a time interval at which we will force an fsync of data written to the log. For example if this was set to 1000 we would fsync after 1000 ms had passed. In general we recommend you not set this and use replication for durability and allow the operating system's background flush capabilities as it is more efficient.

  Type: long
  Default: 9223372036854775807
  Valid Values: [0,...]
  Server Default Property: log.flush.interval.ms
  Importance: medium

- log.flush.interval.messages：表示当同一个Partiton的消息条数积累到这个数量时，就会申请一次刷盘操作。默认是Long.MAX。

  The number of messages accumulated on a log partition before messages are flushed to disk

  Type: long
  Default: 9223372036854775807
  Valid Values: [1,...]
  Importance: high
  Update Mode: cluster-wide

- log.flush.interval.ms：当一个消息在内存中保留的时间，达到这个数量时，就会申请一次刷盘操作。他的默认值是空。如果这个参数配置为空，则生效的是下一个参数。

  log.flush.interval.ms
  The maximum time in ms that a message in any topic is kept in memory before flushed to disk. If not set, the value in log.flush.scheduler.interval.ms is used

  Type: long
  Default: null
  Valid Values:
  Importance: high
  Update Mode: cluster-wide

- log.flush.scheduler.interval.ms：检查是否有日志文件需要进行刷盘的频率。默认也是Long.MAX。

  log.flush.scheduler.interval.ms
  The frequency in ms that the log flusher checks whether any log needs to be flushed to disk

  Type: long
  Default: 9223372036854775807
  Valid Values:
  Importance: high
  Update Mode: read-only

这里可以看到，Kafka为了最大化性能，默认是将刷盘操作交由了操作系统进行统一管理。

另外在这里也能看出，Kafka并没有实现写一个消息依旧进行一次刷盘的“同步刷盘”机制。也就是说，Kafka无法保证非正常断电情况下的消息安全。这其实不光是Kafka面临的问题，而是所有应用程序都需要面临的问题。在RabbitMQ中，官网明确提出，服务端并不完全保证消息不丢失，如果需要提升消息安全性，就只能通过Publisher Confirms机制，让客户端参与验证。而RocketMQ提供了“同步刷盘”的配置选项。但是如果真的每来一个消息就调用一次刷盘操作，那么任何服务器都是无法承受的。RocketMQ是如何实现同步刷盘的呢？日后可以关注一下。

> 【有道云笔记】四、Kafka日志索引详解.md
> https://note.youdao.com/s/RrvuVj94