

一、Kafka的Log日志梳理

- 1、Topic下的消息是如何存储的？
 - 1 log文件追加记录所有消息
 - 2 index和timeindex加速读取log消息日志。
- 2、文件清理机制
- 3、Kafka的文件高效读写机制
 - 1、Kafka的文件结构
 - 2、顺序写磁盘
 - 3、零拷贝
- 4、合理配置刷盘频率
- 5、客户端消费进度管理

二、Kafka生产调优实践

- 1、搭建Kafka监控平台
- 2、合理规划Kafka部署环境
- 3、合理优化Kafka集群配置
- 4、优化Kafka客户端使用方式
- 5、生产环境常见问题分析
 - 1、消息零丢失方案
 - 2、消息积压如何处理
 - 3、如何保证消息顺序

三、课程总结

四、Kafka日志索引详解以及生产常见问题分析与总结

-- 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。比如RocketMQ就直接借鉴了Kafka的日志文件记录机制。当然，是借鉴，不是照抄

一、Kafka的Log日志梳理

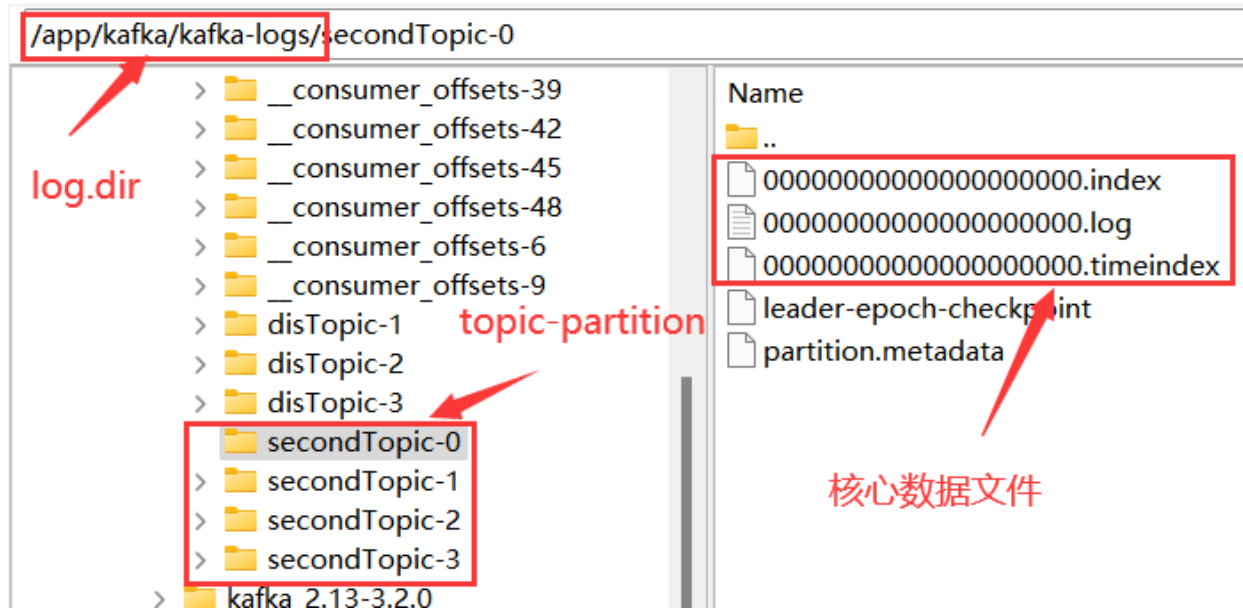
这一部分数据主要包含当前Broker节点的消息数据(在Kafka中称为Log日志)。这是一部分无状态的数据，也就是说每个Kafka的Broker节点都是以相同的逻辑运行。这种无状态的服务设计让Kafka集群能够比较容易的进行水平扩展。比如你需要用一个新的Broker服务来替换集群中一个旧的Broker服务，那么只需要将这部分无状态的数据从旧的Broker上转移到新的Broker上就可以了。

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

1、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)，写满后会新增一个新的文件。一个文件也成为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

```
[oper@worker1 bin]$ ./kafka-dump-log.sh --files /app/kafka/kafka-logs/secondTopic-0/00000000000000000000.timeindex
Dumping /app/kafka/kafka-logs/secondTopic-0/00000000000000000000.timeindex
timestamp: 1661753911323 offset: 61
timestamp: 1661753976084 offset: 119
timestamp: 1661753977822 offset: 175
```

#2、查看index文件

```
[oper@worker1 bin]$ ./kafka-dump-log.sh --files /app/kafka/kafka-logs/secondTopic-0/00000000000000000000.index
Dumping /app/kafka/kafka-logs/secondTopic-0/00000000000000000000.index
offset: 61 position: 4216
```

```
offset: 119 position: 8331
offset: 175 position: 12496
#3、查看log文件
[oper@worker1 bin]$ ./kafka-dump-log.sh --files /app/kafka/kafka-logs/secondTopic-0/00000000000000000000.log
Dumping /app/kafka/kafka-logs/secondTopic-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 1 count: 2 baseSequence: 0 lastSequence: 1 producerId: 7000 producerEpoch: 0 partitionLeaderEpoch: 11 isTransactional: false isControl: false deleteHorizonMs: OptionalLong.empty position: 0 CreateTime: 1661753909195 size: 99 magic: 2 compresscodec: none crc: 342616415 invalid: true
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: 2 lastSequence: 2 producerId: 7000 producerEpoch: 0 partitionLeaderEpoch: 11 isTransactional: false isControl: false deleteHorizonMs: OptionalLong.empty position: 99 CreateTime: 1661753909429 size: 80 magic: 2 compresscodec: none crc: 3141223692 invalid: true
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: 3 lastSequence: 3 producerId: 7000 producerEpoch: 0 partitionLeaderEpoch: 11 isTransactional: false isControl: false deleteHorizonMs: OptionalLong.empty position: 179 CreateTime: 1661753909524 size: 80 magic: 2 compresscodec: none crc: 1537372733 invalid: 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。对这里面的各个属性理解得越详细，也就表示对Kafka的消息日志处理机制理解得越详细。

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新的消息日志。position就是消息记录的起点，size就是消息序列化后的长度。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：

0000.index		00000.log (从0条消息开始)					
offset	position	索引位置					
61	4216	baseOffset	lastOffset	count	position	size	
		0	1	2	0	99	
119	8331	2	2	1	99	80	
175	12496	3	3	1	179	80	
00550.log		4	4	1	259	80	
		5	5	1	339	80	
		6	6	1	419	82	
		7	7	1	501	82	
		8	8	1	583	82	
		9	9	1	665	82	
		
		00550.log (从550条消息开始)					
		baseOffset	lastOffset	count	position	size	
		550	550	1	0	100	

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。

2、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，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。

3、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。

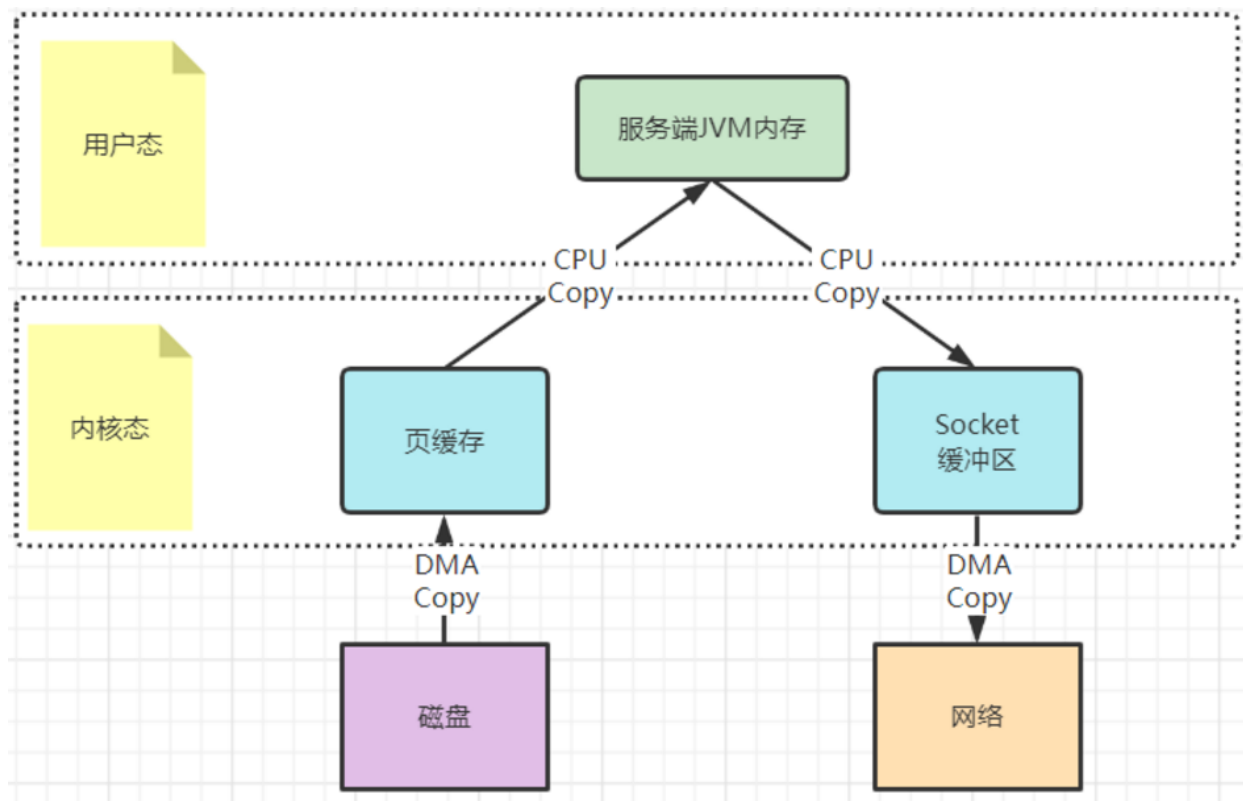
对每个Log文件，Kafka会提前规划固定的大小，这样在申请文件时，可以提前占据一块连续的磁盘空间。然后，Kafka的log文件只能以追加的方式往文件的末端添加(这种写入方式称为顺序写)，这样，新的数据写入时，就可以直接往直前申请的磁盘空间中写入，而不用再去磁盘其他地方寻找空闲的空间(普通的读写文件需要先寻找空闲的磁盘空间，再写入。这种写入方式称为随机写)。由于磁盘的空闲空间有可能并不是连续的，也就是说有很多文件碎片，所以磁盘写的效率会很低。

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：



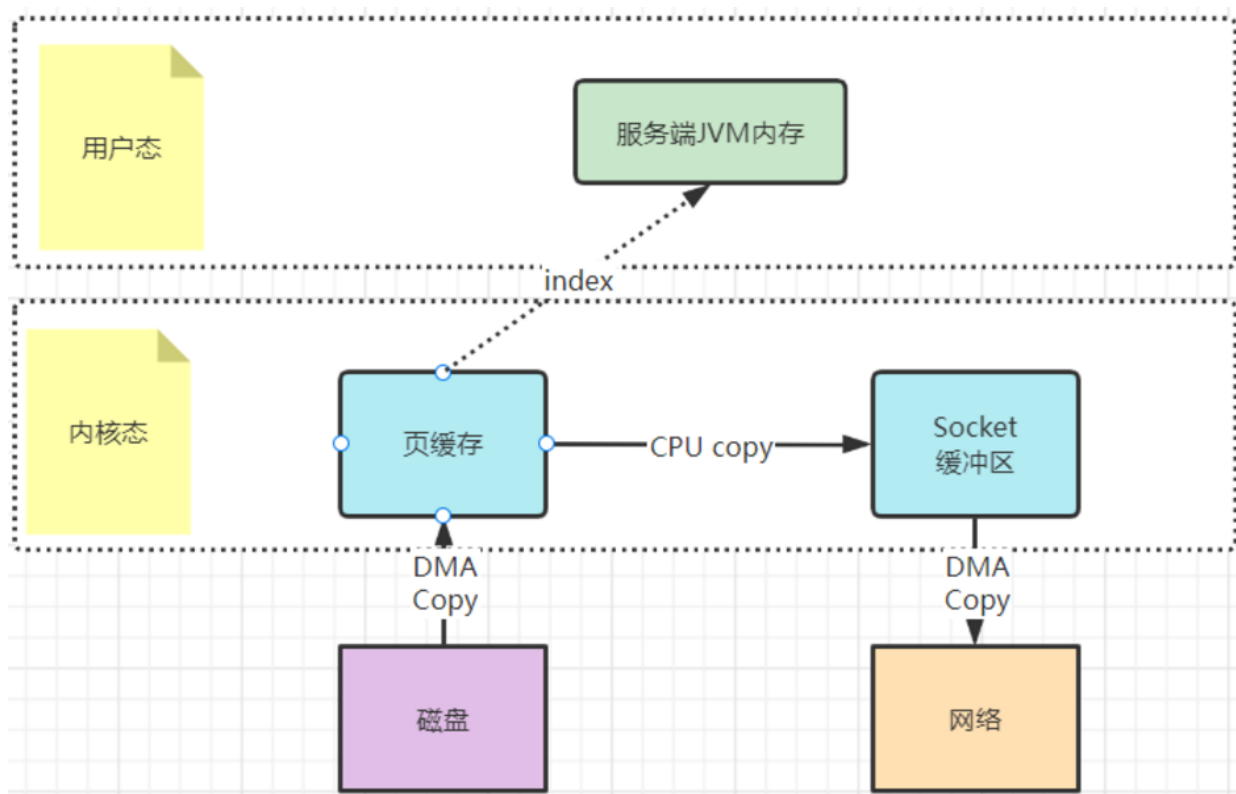
其中，内核态的内容复制是在内核层面进行的，而零拷贝的技术，重点是要配合内核态的复制机制，减少用户态与内核态之间的内容拷贝。

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。



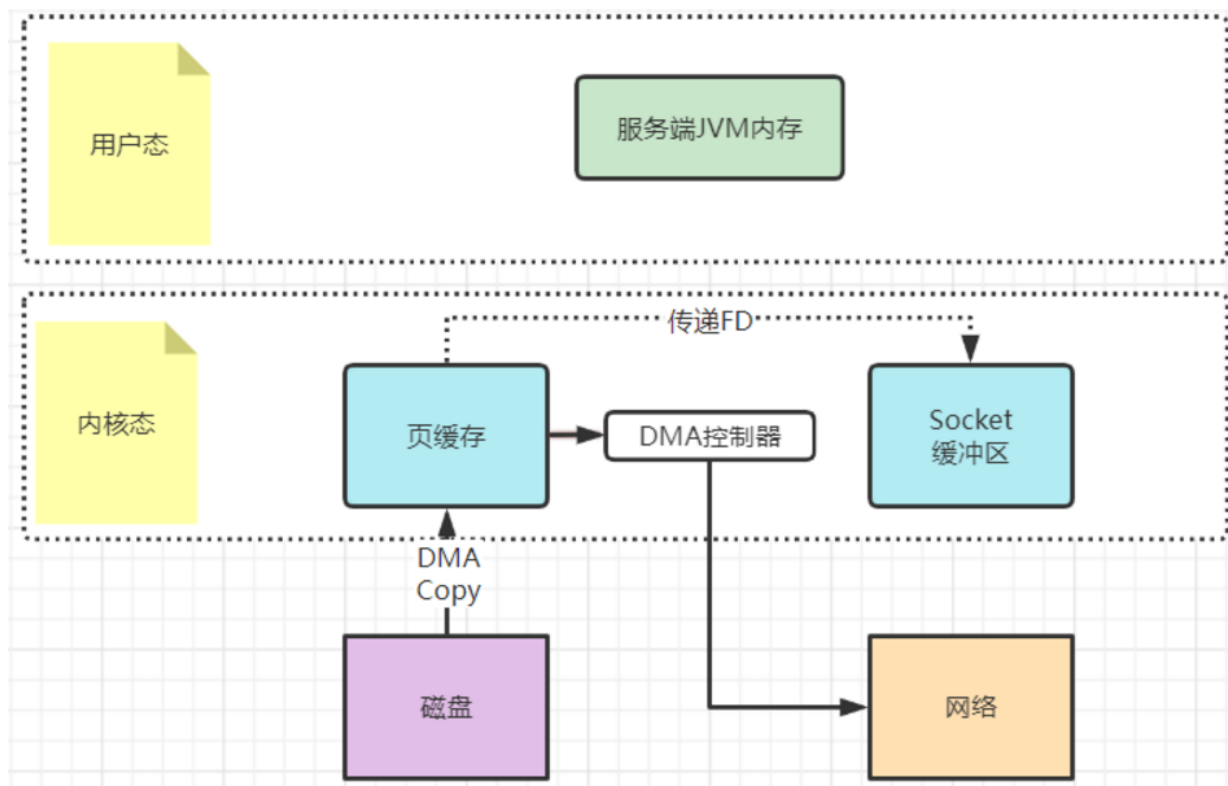
mmap文件映射机制是操作系统提供了一种文件操作机制，可以使用`man 2 mmap`查看。实际上在Java程序执行过程当中就会被大量使用。

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。



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Partition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。

渔与鱼 从这里也能看到, Kafka并没有实现写一个消息就进行一次刷盘的“同步刷盘”操作。但是在RocketMQ中却支持了这种同步刷盘机制。但是如果真的每来一个消息就调用一次刷盘操作, 这是任何服务器都无法承受的。那么RocketMQ是怎么实现同步刷盘的呢? 日后可以关注一下。

5、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。

/app/kafka/kafka-logs

Name	
..	内置Topic
__consumer_offsets-0	
__consumer_offsets-12	
__consumer_offsets-15	
__consumer_offsets-18	
__consumer_offsets-21	
__consumer_offsets-24	
__consumer_offsets-27	
__consumer_offsets-3	
__consumer_offsets-30	
__consumer_offsets-33	
__consumer_offsets-36	
__consumer_offsets-39	
__consumer_offsets-42	
__consumer_offsets-45	
__consumer_offsets-48	
__consumer_offsets-6	
__consumer_offsets-9	
disTopic-1	

同时，Kafka也会将这些消费进度的状态信息记录到Zookeeper中。

`/brokers/topics/__consumer_offsets/partitions/0/state`

ephemeralOwner 0

mtime 2022-08-29 14:18:04

dataLength 74

ctime 2021-09-25 15:25:31

numChildren 0

mZxid 42949673032

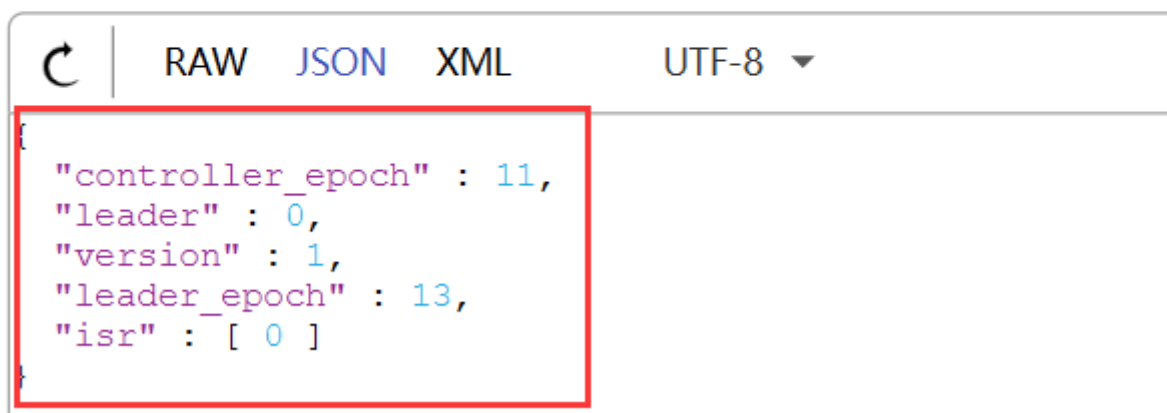
dataVersion 13

pZxid 4294967479

aclVersion 0

cZxid 4294967479

cVersion 0



```
{
  "controller_epoch" : 11,
  "leader" : 0,
  "version" : 1,
  "leader_epoch" : 13,
  "isr" : [ 0 ]
}
```

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。

```
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。

渔与鱼：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;
```

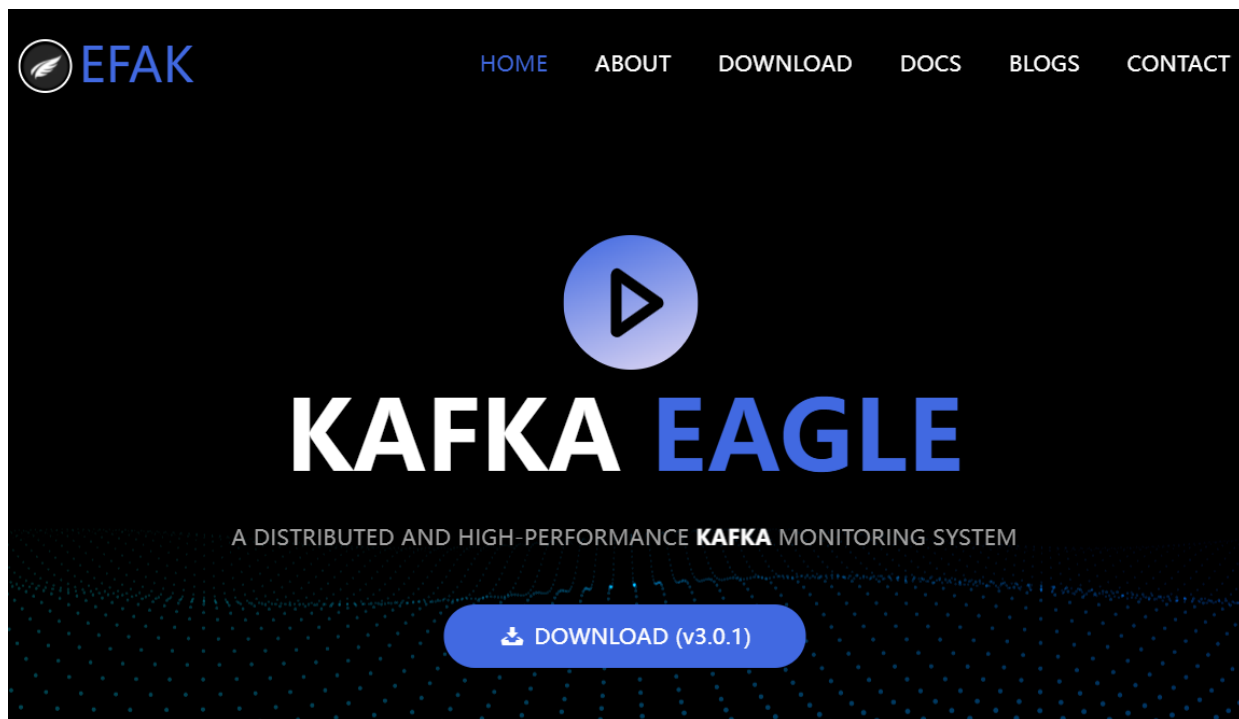
这个参数简单测试了一下，在当前版本是没有用的。

二、Kafka生产调优实践

通常在生产环境中，Kafka都是用来应对整个项目中最高峰的流量的。这种极高的请求流量，对任何服务都是一个很大的负担，因此如果在生产环境中部署Kafka，也可以从以下几个方面进行一些优化。

1、搭建Kafka监控平台

生产环境通常会对Kafka搭建监控平台。而Kafka-eagle就是一个可以监控Kafka集群整体运行情况的框架，在生产环境经常会用到。官网地址：<https://www.kafka-eagle.org/> 以前叫做Kafka-eagle，现在用了个简写，EFAK（Eagle For Apache Kafka）



环境准备：

在官网的DownLoad页面可以下载EFAK的运行包，efak-web-3.0.2-bin.tar.gz。

另外，EFAK需要依赖的环境主要是Java和数据库。其中，数据库支持本地化的SQLite以及集中式的MySQL。生产环境建议使用MySQL。在搭建EFAK之前，需要准备好对应的服务器以及MySQL数据库。

略过MySQL服务搭建过程。

数据库不需要初始化，EFAK在执行过程中会自己完成初始化。

安装过程：以Linux服务器为例。

1、将efak压缩包解压。

```
tar -zxvf efak-web-3.0.2-bin.tar.gz -C /app/kafka/eagle
```

2、修改efak解压目录下的conf/system-config.properties。这个文件中提供了完整的配置，下面只列出需要修改的部分。

```
#####
# multi zookeeper & kafka cluster list
# Settings prefixed with 'kafka.eagle.' will be deprecated, use 'efak.' instead
#####
# 指向Zookeeper地址
efak.zk.cluster.alias=cluster1
cluster1.zk.list=worker1:2181,worker2:2181,worker3:2181
```

```
#####
# zookeeper enable acl
#####
# Zookeeper权限控制
cluster1.zk.acl.enable=false
cluster1.zk.acl.schema=digest
cluster1.zk.acl.username=test
cluster1.zk.acl.password=test123

#####
# kafka offset storage
#####
# offset选择存在kafka中。
cluster1.efak.offset.storage=kafka
cluster2.efak.offset.storage=zookeeper

#####
# kafka mysql jdbc driver address
#####
#指向自己的MySQL服务。库需要提前创建
efak.driver=com.mysql.cj.jdbc.Driver
efak.url=jdbc:mysql://worker1:3306/ke?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
efak.username=root
efak.password=root
```

3、配置EFAK的环境变量

```
vi ~/.bash_profile
-- 配置KE_HOME环境变量，并添加到PATH中。
    export KE_HOME=/app/kafka/eagle/efak-web-3.0.2
    PATH=$PATH:$KE_HOME/bin:$HOME/.local/bin:$HOME/bin
--让环境变量生效
source ~/.bash_profile
```

4、启动EFAK

配置完成后，先启动Zookeeper和Kafka服务，然后调用EFAK的bin目录下的ke.sh脚本启动服务

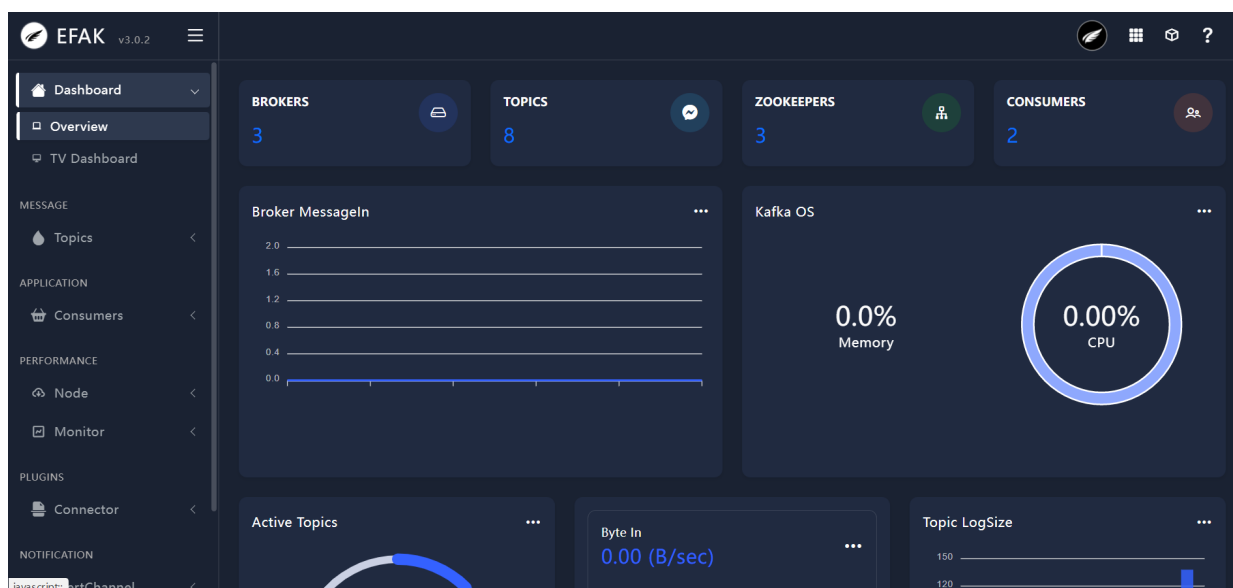
```
[oper@worker1 bin]$ ./ke.sh start
-- 日志很长，看到以下内容表示服务启动成功
[2023-06-28 16:09:43] INFO: [Job done!]
welcome to

  _ _ _ _ _
 / _ _ \ / _ _ \ / _ | / _ _ \
 / _ \ / _ \ / _ | / _ \ ,<
 / _ _ \ / _ _ \ / _ _ | / _ |
 / _ _ \ / _ \ / _ \ | _ \ | _ \
( Eagle For Apache Kafka® )
```

```
Version v3.0.2 -- Copyright 2016-2022
*****
* EFAK Service has started success.
* Welcome, Now you can visit 'http://192.168.232.128:8048'
* Account:admin ,Password:123456
*****
* <Usage> ke.sh [start|status|stop|restart|stats] </Usage>
* <Usage> https://www.kafka-eagle.org/ </Usage>
*****
```

5、访问EFAK管理页面

接下来就可以访问EFAK的管理页面。<http://192.168.232.128:8048>。默认的用户名是admin，密码是123456



关于EFAK更多的使用方式，比如EFAK服务如何集群部署等，可以参考官方文档。

2、合理规划Kafka部署环境

机械硬盘：对于准备部署Kafka服务的服务器，建议配置大容量机械硬盘。Kafka顺序读写的实现方式不太需要SSD这样高性能的磁盘。同等容量SSD硬盘的成本比机械硬盘要高出非常多，没有必要。将SSD的成本投入到MySQL这类的服务更合适。

大内存：在Kafka的服务启动脚本bin/kafka-start-server.sh中，对于JVM内存的规划是比较小的，可以根据之前JVM调优专题中的经验进行适当优化。

脚本中的JVM内存参数默认只申请了1G内存。

```
[oper@worker1 bin]$ cat kafka-server-start.sh
.....
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"
fi
.....
```

对于主流的16核32G服务器，可以适当扩大Kafka的内存。例如：


```
export KAFKA_HEAP_OPTS="-Xmx16G -Xms16G -Xmn10G -XX:MetaspaceSize=256M -XX:+UseG1GC  
-XX:MaxGCPauseMillis=50 -XX:G1HeapRegionSize=16M"
```

高性能网卡：Kafka本身的服务性能非常高，单机就可以支持百万级的TPS。在高流量冲击下，网络非常有可能优先于服务，成为性能瓶颈。并且Kafka集群内部也需要大量同步消息。因此，对于Kafka服务器，建议配置高性能的网卡。成本允许的话，尽量选择千兆以上的网卡。

3、合理优化Kafka集群配置

合理配置Partition数量：Kafka的单个Partition读写效率是非常高的，但是，Kafka的Partition设计是非常碎片化的。如果Partition文件过多，很容易严重影响Kafka的整体性能。

控制Partition文件数量主要有两个方面：1、尽量不要使用过多的Topic，通常不建议超过3个Topic。过多的Topic会加大索引Partition文件的压力。2、每个Topic的副本数不要设置太多。大部分情况下，将副本数设置为2就可以了。

至于Partition的数量，最好根据业务情况灵活调整。partition数量设置多一些，可以一定程度增加Topic的吞吐量。但是过多的partition数量还是同样会带来partition索引的压力。因此，需要根据业务情况灵活进行调整，尽量选择一个折中的配置。

Kafka提供了一个生产者的性能压测脚本，可以用来衡量集群的整体性能。

```
[oper@worker1 bin]$ ./kafka-producer-perf-test.sh --topic test --num-record 1000000  
--record-size 1024 --throughput -1 --producer-props bootstrap.servers=worker1:9092  
acks=1  
94846 records sent, 18969.2 records/sec (18.52 MB/sec), 1157.4 ms avg latency,  
1581.0 ms max latency.  
133740 records sent, 26748.0 records/sec (26.12 MB/sec), 1150.6 ms avg latency,  
1312.0 ms max latency.  
146760 records sent, 29346.1 records/sec (28.66 MB/sec), 1051.5 ms avg latency,  
1164.0 ms max latency.  
137400 records sent, 27480.0 records/sec (26.84 MB/sec), 1123.7 ms avg latency,  
1182.0 ms max latency.  
158700 records sent, 31740.0 records/sec (31.00 MB/sec), 972.1 ms avg latency,  
1022.0 ms max latency.  
158775 records sent, 31755.0 records/sec (31.01 MB/sec), 963.5 ms avg latency,  
1055.0 ms max latency.  
1000000 records sent, 28667.259123 records/sec (28.00 MB/sec), 1030.44 ms avg  
latency, 1581.00 ms max latency, 1002 ms 50th, 1231 ms 95th, 1440 ms 99th, 1563 ms  
99.9th.
```

其中num-record表示要发送100000条压测消息，record-size表示每条消息大小1KB，throughput表示限流控制，设置为小于0表示不限流。properducer-props用来设置生产者的参数。

另外，在我们课程中，介绍了大量的Kafka参数。这些参数都不是凭空冒出来给你介绍的，而是都到源码或者官方文档中查的。经常光顾下这些参数，能够发现很多其他对性能提升有帮助的配置。

例如**合理对数据进行压缩**

在生产者的ProducerConfig中，有一个配置 COMPRESSION_TYPE_CONFIG，是用来对消息进行压缩的。

```
/** <code>compression.type</code> */
public static final String COMPRESSION_TYPE_CONFIG = "compression.type";
private static final String COMPRESSION_TYPE_DOC = "The compression type for all
data generated by the producer. The default is none (i.e. no compression). valid "
+ " values are
<code>none</code>, <code>gzip</code>, <code>snappy</code>, <code>lz4</code>, or
<code>zstd</code>. "
+ "Compression is of full
batches of data, so the efficacy of batching will also impact the compression ratio
(more batching means better compression).";
```

生产者配置了压缩策略后，会对生产的每个消息进行压缩，从而降低Producer到Broker的网络传输，也降低了Broker的数据存储压力。

从介绍中可以看到，Kafka的生产者支持四种压缩算法。这几种压缩算法中，zstd算法具有最高的数据压缩比，但是吞吐量不高。lz4在吞吐量方面的优势比较明显。在实际使用时，可以根据业务情况选择合适的压缩算法。但是要注意下，压缩消息必然增加CPU的消耗，如果CPU资源紧张，就不要压缩了。

关于数据压缩机制，在Broker端的broker.conf文件中，也是可以配置压缩算法的。正常情况下，Broker从Producer端接收到消息后不会对其进行任何修改，但是如果Broker端和Producer端指定了不同的压缩算法，就会产生很多异常的表现。

```
compression.type
Specify the final compression type for a given topic. This configuration accepts the
standard compression codecs ('gzip', 'snappy', 'lz4', 'zstd'). It additionally
accepts 'uncompressed' which is equivalent to no compression; and 'producer' which
means retain the original compression codec set by the producer.

Type:      string
Default:    producer
Valid values: [uncompressed, zstd, lz4, snappy, gzip, producer]
Server Default Property:    compression.type
Importance: medium
```

如果开启了消息压缩，那么在消费者端自然是要进行解压缩的。在Kafka中，消息从Producer到Broker再到Consumer会一直携带消息的压缩方式，这样当Consumer读取到消息集合时，自然就知道了这些消息使用的是哪种压缩算法，也就可以自己进行解压了。但是这时要注意的是应用中使用的Kafka客户端版本和Kafka服务端版本是否匹配。

4、优化Kafka客户端使用方式

在使用Kafka时，也需要根据业务情况灵活进行调整，选择最合理的Kafka使用方式。

1、合理保证消息安全

在生产者端最好从以下几个方面进行优化。

- 设置好发送者应答参数：主要涉及到两个参数。

- 一个是生产者的ACKS_CONFIG配置。acks=0，生产者不关心Broker端有没有将消息写入到Partition，只发送消息就不管了。acks=all or -1，生产者需要等Broker端的所有Partition(Leader Partition以及其对应的Follower Partition都写完了才能得到返回结果，这样数据是最安全的，但是每次发消息需要等待更长的时间，吞吐量是最低的。acks设置成1，则是一种相对中和的策略。Leader Partition在完成自己的消息写入后，就向生产者返回结果。

其中acks=1是应用最广的一种方案。但是，如果结合服务端的min.insync.replicas参数，就可以配置更灵活的方式。

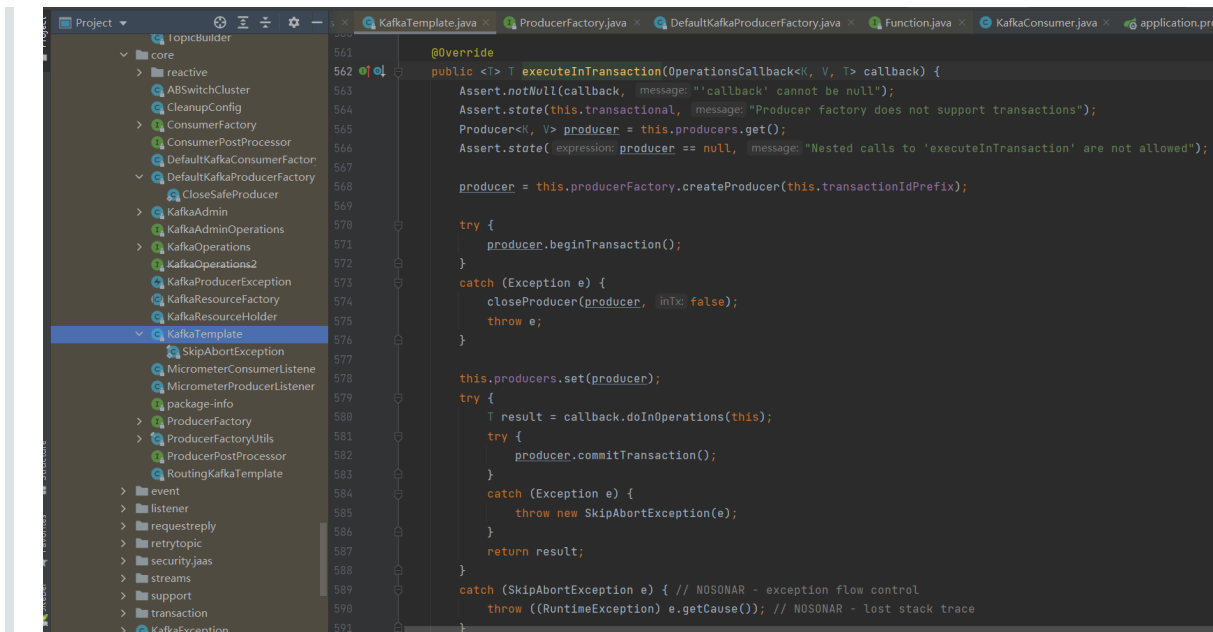
- min.insync.replicas参数表示如果生产者的acks设置为-1或all，服务端并不是强行要求所有Partition都完成写入再返回，而是可以配置多少个Partition完成消息写入后，再往Producer返回消息。比如，对于一个Topic，设置他的备份因子replication factor为3，然后将min.insync.replicas参数配置为2，而生产者端将ACKS_CONFIG设定为-1或all，这样就能在消息安全性和发送效率之间进行灵活选择。
- 打开生产者端的幂等性配置：ENABLE_IDEMPOTENCE_CONFIG。生产者将这个参数设置为true后，服务端会根据生产者实例以及消息的目标Partition，进行重复判断，从而过滤掉生产者一部分重复发送的消息。
- 使用生产者事务机制发送消息：

在打开幂等性配置后，如果一个生产者实例需要发送多条消息，而你能够确定这些消息都是发往同一个Partition的，那么你就不需要再过多考虑消息安全的问题。但是如果你不确定这些消息是不是发往同一个Partition，那么尽量使用异步发送消息机制加上事务消息机制进一步提高消息的安全性。

生产者事务机制主要是通过以下一组API来保证生产者往服务端发送消息的事务性。

```
// 1 初始化事务
void initTransactions();
// 2 开启事务
void beginTransaction() throws ProducerFencedException;
// 3 提交事务
void commitTransaction() throws ProducerFencedException;
// 4 放弃事务（类似于回滚事务的操作）
void abortTransaction() throws ProducerFencedException;
```

尤其在与Spring框架整合使用时，通常会将Producer作为一个单例放入到Spring容器中，这时候就更需要注意事务消息使用。实际上SpringBoot集成Kafka时使用的KafkaTemplate就是使用事务消息机制发送的消息。



然后在消费者端。Kafka消费消息是有重试机制的，如果消费者没有主动提交事务(自动提交或者手动提交)，那么这些失败的消息是可以交由消费者组进行重试的，所以正常情况下，消费者这一端是不会丢失消息的。但是如果消费者要使用异步方式进行业务处理，那么如果业务处理失败，此时消费者已经提交了Offset，这个消息就无法重试了，这就会造成消息丢失。

因此在消费者端，尽量不要使用异步处理方式，在绝大部分场景下，就能够通过Kafka的消费者重试机制，保证消息安全处理。此时，在消费者端，需要更多考虑的问题，就变成了消费重试机制造成的消息重复消费的问题。

2、消费者防止消息重复消费

回顾一下消费者的实现步骤，通常都是这样的处理流程：

```
while (true) {  
    //拉取消息  
    ConsumerRecords<String, String> records =  
    consumer.poll(Duration.ofNanos(100));  
    //处理消息  
    for (ConsumerRecord<String, String> record : records) {  
        //do business ...  
    }  
    //提交offset，消息就不会重复推送。  
    consumer.commitSync(); //同步提交，表示必须等到offset提交完毕，再去消费下一批数据。  
}
```

在大部分的业务场景下，这不会有什么问题。但是在一些大型项目中，消费者的业务处理流程会很长，这时就会带来一些问题。比如，一个消费者在正常处理这一批消息，但是时间需要很长。Broker就有可能认为消息消费失败了，从而让同组的其他消费者开始重试这一批消息。这就给消费者端带来不必要的幂等性问题。

消费者端的幂等性问题，当然可以交给消费者自己进行处理，比如对于订单消息，消费者根据订单ID去确认一下这个订单消息有没有处理过。这种方式当然是可以的，大部分的业务场景下也都是这样处理的。但是这样会给消费者端带来更大的业务复杂性。

但是在很多大型项目中，消费者端的业务逻辑有可能是非常复杂的。这时候要进行幂等性判断，，因此会更希望以一种统一的方式处理幂等性问题，让消费者端能够专注于处理自己的业务逻辑。这时，在大型项目中有一种比较好的处理方式就是将Offset放到Redis中自行进行管理。通过Redis中的offset来判断消息之前是否处理过。伪代码如下：

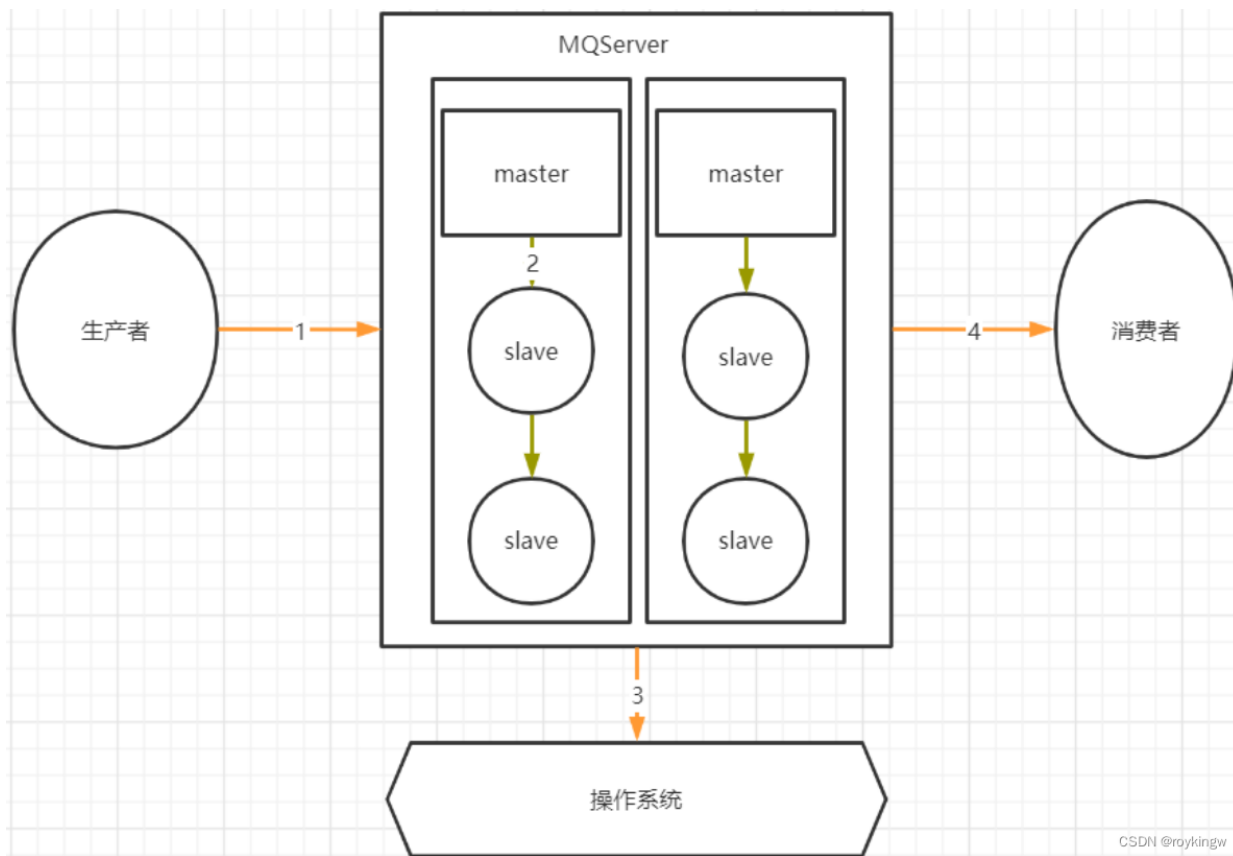
```
while(true){
    //拉取消息
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
    records.partitions().forEach(partition ->{
        //从redis获取partition的偏移量
        String redisKafkaOffset = redisTemplate.opsForHash().get(partition.topic(),
"" + partition.partition()).toString();
        Long redisOffset = StringUtils.isEmpty(redisKafkaOffset)?
-1:Long.valueOf(redisKafkaOffset);
        List<ConsumerRecord<String, String>> partitionRecords =
records.records(partition);
        partitionRecords.forEach(record ->{
            //redis记录的偏移量>=kafka实际的偏移量，表示已经消费过了，则丢弃。
            if(redisOffset >= record.offset()){
                return;
            }
            //业务端只需要实现这个处理业务的方法就可以了，不用再处理幂等性问题
            doMessage(record.topic(),record.value());
        });
    });
    //处理完成后立即保存Redis偏移量
    Long saveRedisOffset = partitionRecords.get(partitionRecords.size() -
1).offset();
    redisTemplate.opsForHash().put(partition.topic(),"" +
partition.partition(),saveRedisOffset);
    //异步提交。消费业务多时，异步提交有可能造成消息重复消费，通过Redis中的Offset，就可以过滤掉这
一部分重复的消息。。
    consumer.commitAsync();
}
```

将这段代码封装成一个抽象类，具体的业务消费者端只要继承这个抽象类，然后就可以专注于实现doMessage方法，处理业务逻辑即可，不用再过多关心幂等性的问题。

5、生产环境常见问题分析

1、消息零丢失方案

MQ的全链路消息不丢失问题，需要的通常都是两个元凶：网络+缓存。因此Kafka也同样需要从这几个方面来考虑。



1、生产者发送消息到Broker不丢失

Kafka的消息生产者Producer，支持定制一个参数，`ProducerConfig.ACKS_CONFIG`。

- acks配置为0：生产者只负责往Broker端发消息，而不关注Broker的响应。也就是说，不关心Broker端有没有收到消息。性能高，但是数据会有丢失消息的可能。
- acks配置为1：当Broker端的Leader Partition接收到消息后，只完成本地日志文件的写入，然后就给生产者答复。其他Partition异步拉取Leader Partition的消息文件。这种方式如果其他Partition拉取消息失败，也有可能丢失消息。
- acks配置为-1或者all：Broker端会完整所有Partition的本地日志写入后，才会给生产者答复。数据安全性最高，但是性能显然是最低的。

对于KafkaProducer，只要将acks设置成1或-1，那么Producer发送消息后都可以拿到Broker的反馈RecordMetadata，里面包含了消息在Broker端的partition, offset等信息。通过这些信息可以判断消息是否发送成功。如果没有发送成功，Producer就可以根据情况选择重新进行发送。

2、Broker端保存消息不丢失

首先，合理优化刷盘频率，防止服务异常崩溃造成消息未刷盘。Kafka的消息都是先写入操作系统的PageCache缓存，然后再刷盘写入到硬盘。PageCache缓存中的消息是断电即丢失的。如果消息只在PageCache中，而没有写入硬盘，此时如果服务异常崩溃，这些未写入硬盘的消息就会丢失。Kafka并不支持写一条消息就刷一次盘的同步刷盘机制，只能通过调整刷盘的执行频率，提升消息安全。主要涉及几个参数：

- `flush.ms`：多长时间进行一次强制刷盘。
- `log.flush.interval.messages`：表示当同一个Partition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。

- 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。

然后，配置多备份因子，防止单点消息丢失。在Kafka中，可以给Topic配置更大的备份因子replication-factors。配置了备份因子后，Kafka会给每个Partition分配多个备份Partition。这些Partiton会尽量平均的分配到多个Broker上。并且，在这些Partiton中，会选举产生Leader Partition和Follower Partition。这样，当Leader Partition发生故障时，其他Follower Partition上还有消息的备份。就可以重新选举产生Leader Partition，继续提供服务。

如果按照上一章节的方式分析Kafka中Partition故障恢复的过程，那么在服务经常崩溃的极端环境下，Kafka其实是为了高性能牺牲了消息安全性的。这样看来，kafka的消息又不是太安全的。

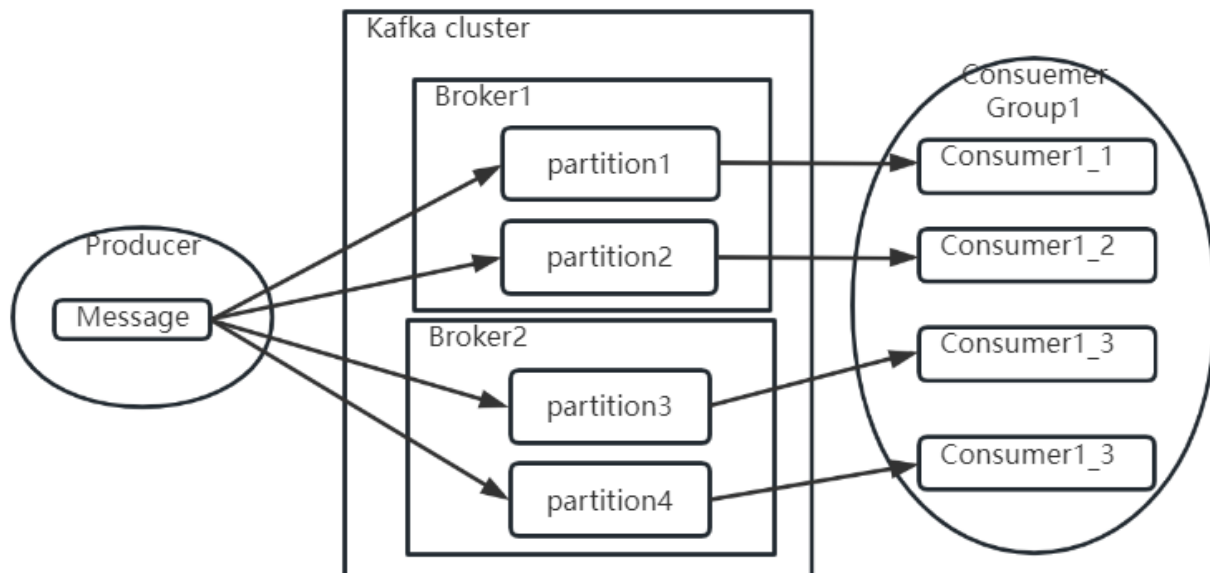
3、消费者端防止异步处理丢失消息

消费者端由于有消息重试机制，正常情况下是不会丢消息的。每次消费者处理一批消息，需要在处理完后给Broker应答，提交当前消息的Offset。Broker接到应答后，会推进本地日志的Offset记录。如果Broker没有接到应答，那么Broker会重新向同一个消费者组的消费者实例推送消息，最终保证消息不丢失。这时，消费者端采用手动提交Offset的方式，相比自动提交会更容易控制提交Offset的时机。

消费者端唯一需要注意的是，不要异步处理业务逻辑。因为如果业务逻辑异步进行，而消费者已经同步提交了Offset，那么如果业务逻辑执行过程中出现了异常，失败了，那么Broker端已经接收到了消费者的应答，后续就不会再重新推送消息，这样就造成了业务层面的消息丢失。

2、消息积压如何处理

Kafka的粗略消息模型是这样的：



通常情况下，Kafka本身是能够存储海量消息的，他的消息积压能力是很强的。但是，如果发现消息积压问题已经影响了业务处理进度，这时就需要进行一定的优化。

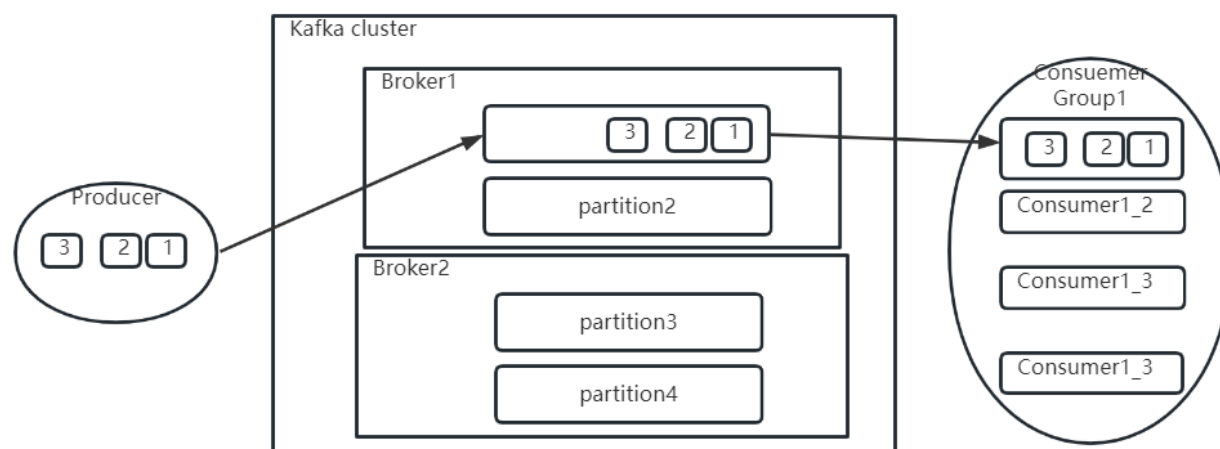
1、如果业务运行正常，只是因为消费者处理消息过慢，造成消息积压。那么可以增加Topic的Partition分区数，将消息拆分到更多的Partition。然后增加消费者个数，最多让消费者个数=Partition分区数，让一个Consumer负责一个分区，将消费进度提升到最大。

另外，在发送消息时，还是要尽量保证消息在各个Partition中的分布比较均匀。比如，在原有Topic下，可以调整Producer的分区策略，让Producer将后续的消息更多的发送到新增的Partition里，这样可以让各个Partition上的消息能够趋于平衡。如果你觉得这样太麻烦，那就新增一个Topic，配置更多的Partition以及对应的消费者实例。然后启动一批Consumer，将消息从旧的Topic搬运到新的Topic。这些Consumer不处理业务逻辑，只是做消息搬运，所以他们的性能是很高的。这样就能让新的Topic下的各个Partition数量趋于平衡。

2、如果是消费者的业务问题导致消息阻塞了，从而积压大量消息，并影响了系统正常运行。比如消费者序列化失败，或者业务处理全部异常。这时可以采用一种降级的方案，先启动一个Consumer将Topic下的消息先转发到其他队列中，然后再慢慢分析新队列里的消息处理问题。类似于死信队列的处理方式。

3、如何保证消息顺序

这也是一个常见的面试题。有时候业务上会需要消息按照顺序进行处理。例如QQ的聊天记录，一问一答必须有顺序，要是顺序乱了，就没法看了。这时应该怎么做？这个问题要交由Kafka来处理是很麻烦的，因为我们一直强调过，kafka设计的最优先重点是海量吞吐，所以他对于传统MQ面临的这些问题，处理是比较粗犷的。如果面试中遇到这样的问题，很多朋友都会想当然的按照其他MQ的思路随意回答。比如最典型的单partition，单Consumer组合就可以了。但是，经过Kafka这一轮详细的梳理后，我希望你能更成熟，更理智的分析这个问题。至少，你能够深入Kafka做一些个性化的分析，体现一点Kafka独特的东西。



这问题要分两个方面来考虑：

1、因为kafka中各个Partition的消息是并发处理的，所以要保证消息顺序，对于Producer，要保证将一组有序的消息发到同一个Partition里。因为Partition的数据是顺序写的，所以自然就能保证消息是按顺序保存的。

2、接下来对于消费者，需要能够按照1,2,3的顺序处理消息。

接下来我们详细分析一下这两个目标要怎么达成。

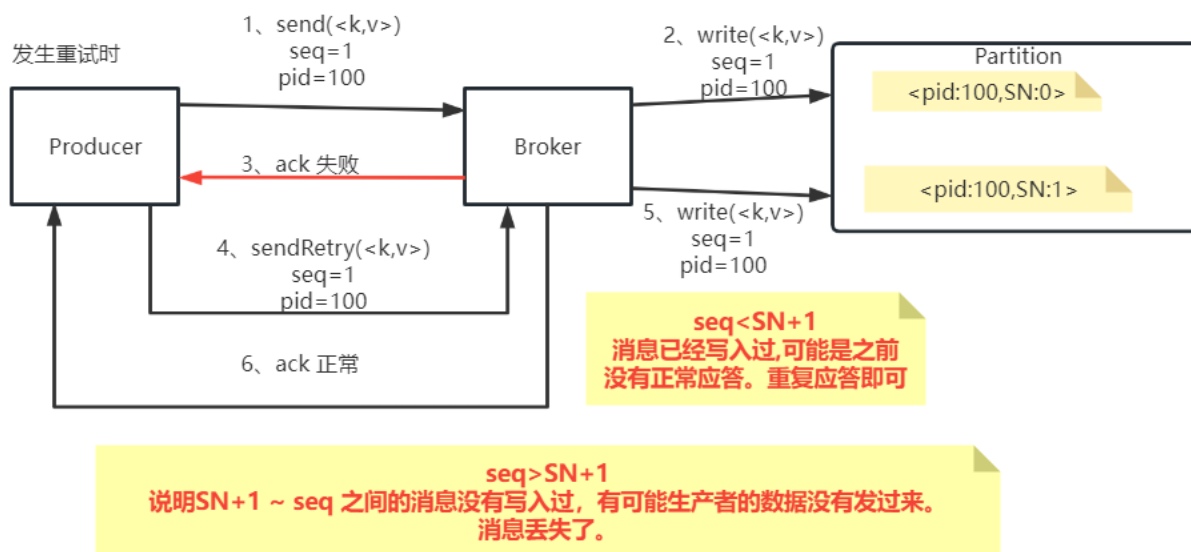
问题一、如何保证Producer发到Partition上的消息是有序的

首先，要保证Producer将消息都发送到一个Partition上，其实有两种方法。一种简答粗暴的想法就是给Topic只配一个Partition，没有其他Partition可选了，自然所有消息都到同一个Partition上了。但是，这表示从创建Topic时就放弃了多Partition带来的吞吐量便利，从一开始就是不现实的。现实一点的想法是，Topic依然配置多个Partition，但是通过定制Producer的Partition分区器，将消息分配到同一个Partition上。这样对于某一些要求局部有序的场景是至少是可行的。例如在电商场景，我可能只是需要保证同一个订单相关的

多条消息有序，但是并不要求所有消息有序。这样就可以通过自定义分区路由器，将订单相同的多条消息发送到同一个Partition。

然后，是不是Producer都将消息往同一个Partition发，就能保证消息顺序呢？如果只追求答案，那么结果肯定是正确的，因为Partition就是FIFO的队列结构。但是，稍微深入想想怎么实现的，就没这么简单了。因为消息可能发送失败。比如Producer依次发送1,2,3，三条消息。如果消息1因为网络原因发送失败了，2和3发送成功了，这样消息顺序就乱了。那怎么办呢？有人就会想到把producer的acks参数设置成1或-1，这样每次发送消息后，可以根据Broker的反馈判断消息是否成功。如果发送失败，就重试。直到1发送成功了，再发送2。2发送成功了，再发送3。这样的思路是可行的，但是你会想到，照这样的思路，写出来的代码就没法看了。重试多少次？要发送多少个消息？这都很难处理。并且每次只发送一个消息，这发送消息的性能基本就低到没法用了。那kafka是如何考虑这个事情的呢？

其实消息顺序这事，Kafka是有考量的。回顾一下之前对于生产者消息幂等性的设计，是这样的：



之前说过，Kafka的这个sequenceNumber是单调递增的。如果只是为了消息幂等性考虑，那么只要保证sequenceNumber唯一就行了，为什么要设计成单调递增呢？其实Kafka这样设计的原因就是可以通过sequenceNumber来判断消息的顺序。也就是说，在Producer发送消息之前就可以通过sequenceNumber定制好消息的顺序，然后Broker端就可以按照顺序来保存消息。与此同时，SequenceNumber单调递增的特性不光保证了消息是有顺序的，同时还保证了每一条消息不会丢失。一旦Kafka发现Producer传过来的SequenceNumber出现了跨越，那么就意味着中间有可能消息出现了丢失，就会往Producer抛出一个OutOfOrderSequenceException异常。

Kafka到底有没有这么做？老规矩，不要拍脑袋想。先去Kafka里找一找。在生产者的配置类ProducerConfig中很快能找到很多和消息顺序ordering的描述：

```

    public static final String MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION =
"max.in.flight.requests.per.connection";
    private static final String MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION_DOC = "The
maximum number of unacknowledged requests the client will send on a single
connection before blocking."

+ " Note

that if this configuration is set to be greater than 1 and
<code>enable.idempotence</code> is set to false, there is a risk of"

+ "

message reordering after a failed send due to retries (i.e., if retries are
enabled); "

+ " if

retries are disabled or if <code>enable.idempotence</code> is set to true, ordering
will be preserved."

+ "

Additionally, enabling idempotence requires the value of this configuration to be
less than or equal to " + MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION_FOR_IDEMPOTENCE +
"."

+ " If

conflicting configurations are set and idempotence is not explicitly enabled,
idempotence is disabled. ";

```

另外还有很多参数都有关于ordering的描述。

```

    public static final String RETRIES_CONFIG = CommonClientConfigs.RETRIES_CONFIG;
    private static final String RETRIES_DOC = "Setting a value greater than zero
will cause the client to resend any record whose send fails with a potentially
transient error."

+ " Note that this retry is no different than if the client resent the
record upon receiving the error."

+ " Produce requests will be failed before the number of retries has
been exhausted if the timeout configured by"

+ " <code>" + DELIVERY_TIMEOUT_MS_CONFIG + "</code> expires first before
successful acknowledgement. Users should generally"

+ " prefer to leave this config unset and instead use <code>" +
DELIVERY_TIMEOUT_MS_CONFIG + "</code> to control"

+ " retry behavior."

+ "<p>"

+ "Enabling idempotence requires this config value to be greater than
0."

+ " If conflicting configurations are set and idempotence is not
explicitly enabled, idempotence is disabled."

+ "<p>"

+ "Allowing retries while setting <code>enable.idempotence</code> to
<code>>false</code> and <code>" + MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION + "</code> to
1 will potentially change the"

+ " ordering of records because if two batches are sent to a single
partition, and the first fails and is retried but the second"

+ " succeeds, then the records in the second batch may appear first.";

```

稍微扩展一下，你会发现，这种机制其实不光适合Kafka这样的消息发送场景，如果你的微服务对请求顺序敏感，这也是一种非常值得借鉴的方案。

问题二：Partition中的消息有序后，如何保证Consumer的消费顺序是有序的

现在消息已经有序的保存到Partition上了，那么Consumer是不是会乖乖的按照Partition上的消息按顺序一批一批处理呢？这问题你不用去研究源码，按照课程的思路，去ConsumerConfig中找找有没有相关的描述：

```
public static final String FETCH_MAX_BYTES_CONFIG = "fetch.max.bytes";
private static final String FETCH_MAX_BYTES_DOC = "The maximum amount of data
the server should return for a fetch request. " +
    "Records are fetched in batches by the consumer, and if the first record
batch in the first non-empty partition of the fetch is larger than " +
    "this value, the record batch will still be returned to ensure that the
consumer can make progress. As such, this is not a absolute maximum. " +
    "The maximum record batch size accepted by the broker is defined via
<code>message.max.bytes</code> (broker config) or " +
    "<code>max.message.bytes</code> (topic config). Note that the consumer
performs multiple fetches in parallel.";
public static final int DEFAULT_FETCH_MAX_BYTES = 50 * 1024 * 1024;
```

这里明确提到Consumer其实是每次并行的拉取多个Batch批次的消息进行处理的。也就是说Consumer拉取过来的多批消息并不是串行消费的。所以在Kafka提供的客户端Consumer中，是没有办法直接保证消费的消息顺序。其实这也比较好理解，因为Kafka设计的重点是高吞吐量，所以他的设计是让Consumer尽最大的能力去消费消息。而只要对消费的顺序做处理，就必然会影响Consumer拉取消息的性能。

所以这时候，我们能做的就是Consumer的处理逻辑中，将消息进行排序。比如将消息按照业务独立性收集到一个集合中，然后在集合中对消息进行排序。

那么针对消费者顺序消费的问题，有没有其他的处理思路呢？在RocketMQ中提供了一个比较好的方式。RocketMQ中提供了顺序消息的实现。他的实现原理是先锁定一个队列(在RocketMQ中称为MessageQueue，类似于Kafka中的Partition，都是实际存储消息的队列结果)，消费完这一个队列后，才开始锁定下一个队列，并消费队列中的消息。再结合MessageQueue中的消息有序性，就能保证整体消息的消费顺序是有序的。

三、课程总结

Kafka的设计重点是在网络不稳定，服务也不稳定的复杂分布式环境下，如何保持高性能，高可用，高可扩展的三高架构。在这方面，Kafka的设计是很复杂也很完善的，是业内公认的老大哥。因此网上的解读文章也是最多的。但是这些解读的文章如果没有一条主线串起来，那永远都只是一些零散的，过目既忘的东西。

在这几期的课程中，一直在尝试以可见的方式去深入理解Kafka的各种主要设计思想。内容非常多，节奏也非常紧凑，但是梳理出了客户端、Zookeeper数据、Log日志三条可以看得见的数据主线。希望通过这些可见的东西为主线，带你串联起Kafka最核心的设计思想。但是，如果你希望强行去记忆这些各种各样的配置，那就完蛋了。你会觉得这么多参数，有客户端的，有服务端的，看过就忘了。最后一回顾，几节课的内容全在官方文档里。离开官方文档，你啥都没学会。

这是好的，也是不好的。习惯去梳理官方文档，至少让你面对互联网上海量的Kafka文章和视频，有了基础的鉴别能力。但同时这也有不好的地方。海量的参数容易让你在开发过程中陷入配置海洋。这么多配置，不知道要配哪些。有些配置的作用根本无法理会。实际开发时不知从何下手。因此，对于Kafka，你更应该从各个角度建立起一个完整的数据流转的模型，理解一些Kafka处理三高问题的核心思想。这样一些核心的参数配置可以通过这个模型整合起来。后续如果你有时间，再去回顾补充Kafka的一些重要设计细节，并且尝试去进行验证。这样才能真正提升自己对于kafka的理解。

有道云笔记链接：<https://note.youdao.com/s/au4nS1l8>