

Kafka的Zookeeper元数据梳理

- 1、zookeeper整体数据
- 2、Controller Broker选举机制
- 3、Leader Partition选举机制
- 4、Leader Partition自动平衡机制
- 5、Partition故障恢复机制
- 6、HW一致性保障-Epoch更新机制
- 7、章节总结

三、从Zookeeper数据理解Kafka集群工作机制

-- 楼兰

这一部分主要是理解Kafka的服务端重要原理。但是Kafka为了保证高吞吐，高性能，高可扩展的三高架构，很多具体设计都是相当复杂的。如果直接跳进去学习研究，很快就会晕头转向。所以，找一个简单清晰的主线就显得尤为重要。这一部分主要是从可见的存储数据的角度来理解Kafka的Broker运行机制。这对于上一章节建立的简单模型，是一个很好的细节补充。

Kafka依赖很多的存储数据，但是，总体上是有划分的。Kafka会将每个服务的不同之处，也就是状态信息，保存到Zookeeper中。通过Zookeeper中的数据，指导每个Kafka进行与其他Kafka节点不同的业务逻辑。而将状态信息抽离后，剩下的数据，就可以直接存在Kafka本地，所有Kafka服务都以相同的逻辑运行。这种状态信息分离的设计，让Kafka有非常好的集群扩展性。

Kafka的Zookeeper元数据梳理

1、zookeeper整体数据

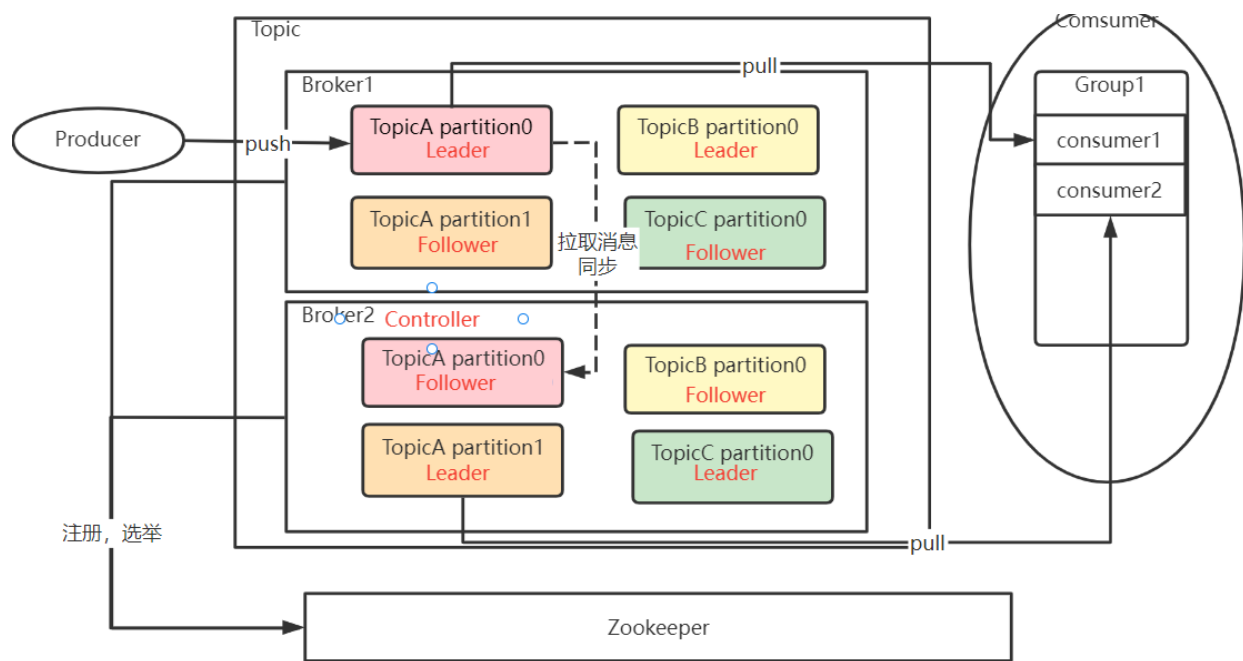
Kafka将状态信息保存在Zookeeper中，这些状态信息记录了每个Kafka的Broker服务与另外的Broker服务有什么不同。通过这些差异化的功能，共同体现出集群化的业务能力。这些数据，需要在集群中各个Broker之间达成共识，因此，需要存储在一个所有集群都能共同访问的第三方存储中。

这些共识数据需要保持强一致性，这样才能保证各个Broker的分工是同步、清晰的。而基于CP实现的Zookeeper就是最好的选择。

另外，Zookeeper的Watcher机制也可以很好的减少Broker读取Zookeeper的次数。

Kafka在Zookeeper上管理了哪些数据呢？这个问题可以先回顾一下Kafka的整体集群状态结构，然后再去Zookeeper上验证。

Kafka的整体集群结构如下图。其中红色字体标识出了重要的状态信息。



Kafka的集群中，最为主要的状态信息有两个。一个是在多个Broker中，需要选举出一个Broker，担任Controller角色。由Controller角色来管理整个集群中的分区和副本状态。另一个是在同一个Topic下的多个Partition中，需要选举出一个Leader角色。由Leader角色的Partition来负责与客户端进行数据交互。

这些状态信息都被Kafka集群注册到了Zookeeper中。Zookeeper数据整体如下图：

▶ admin

▼ brokers

▶ ids

seqid

▼ topics

▶ __consumer_offsets

▶ disTopic

Kafka相关

▼ cluster

id

▶ config

consumers

controller

controller_epoch

feature

isr_change_notification

latest_producer_id_block

log_dir_event_notification

▶ zookeeper zookeeper自己的数据

Zookeeper客户端工具：prettyZoo。下载地址：<https://github.com/vran-dev/PrettyZoo/releases>

对于Kafka往Zookeeper上注册的这些节点，大部分都是比较简明的。比如/brokers/ids下，会记录集群中的所有BrokerId，/topics目录下，会记录当前Kafka的Topic相关的Partition分区等信息。下面就从这些Zookeeper的基础数据开始，来逐步梳理Kafka的Broker端的重要流程。

例如集群中每个Broker启动后，都会往Zookeeper注册一个临时节点/broker/ids/{BrokerId}。可以做一个试验验证一下。如果启动了Zookeeper和Kafka后，服务器非正常关机，这时Zookeeper上的这个临时节点就不会注销。下次重新启动Kafka时，就有可能因为无法注册上这个临时节点而报错。

```
[2023-03-23 11:24:08.542] INFO Creating /brokers/ids/1 (is it secure? false) (kafka.zk.KafkaZkClient)
[2023-03-23 11:24:08.576] ERROR Error while creating ephemeral at /brokers/ids/1, node already exists and owner '72059086999257090' does not match current session '72057611838881792' (kafka.zk.KafkaZkClient$CheckedEphemeral)
[2023-03-23 11:24:08.586] ERROR [KafkaServer id=1] Fatal error during KafkaServer startup. Prepare to shutdown (kafka.server.KafkaServer)
org.apache.zookeeper.KeeperException$NodeExistsException: KeeperErrorCode = NodeExists
    at org.apache.zookeeper.KeeperException.create(KeeperException.java:126)
    at kafka.zk.KafkaZkClient$CheckedEphemeral.getAfterNodeExists(KafkaZkClient.scala:2185)
    at kafka.zk.KafkaZkClient$CheckedEphemeral.create(KafkaZkClient.scala:2123)
    at kafka.zk.KafkaZkClient.checkedEphemeralCreate(KafkaZkClient.scala:2090)
    at kafka.zk.KafkaZkClient.registerBroker(KafkaZkClient.scala:102)
    at kafka.server.KafkaServer.startup(KafkaServer.scala:355)
    at kafka.Kafka$.main(Kafka.scala:115)
    at kafka.Kafka.main(Kafka.scala)
[2023-03-23 11:24:08.590] INFO [KafkaServer id=1] shutting down (kafka.server.KafkaServer)
```

2、Controller Broker选举机制

在Kafka集群进行工作之前，需要选举出一个Broker来担任Controller角色，负责整体管理集群内的分区和副本状态。选举Controller的过程就是通过抢占Zookeeper的/controller节点来实现的。

当一个集群内的Kafka服务启动时，就会尝试往Zookeeper上创建一个/controller临时节点，并将自己的brokerid写入这个节点。节点的内容如下：

```
{"version":1,"brokerid":0,"timestamp":"1661492503848"}
```

Zookeeper会保证在一个集群中，只会有一个broker能够成功创建这个节点。这个注册成功的broker就成了集群当中的Controller节点。

当一个应用在Zookeeper上创建了一个临时节点后，Zookeeper需要这个应用一直保持连接状态。如果Zookeeper长时间检测不到应用的心跳信息，就会删除临时节点。同时，Zookeeper还允许应用监听节点的状态，当应用状态有变化时，会向该节点对应的所有监听器广播节点变化事件。

这样，如果集群中的Controller节点服务宕机了，Zookeeper就会删除/controller节点。而其他未注册成功的Broker节点，就会感知到这一事件，然后开始竞争，再次创建临时节点。这就是Kafka基于Zookeeper的Controller选举机制。

选举产生的Controller节点，就会负责监听Zookeeper中的其他一些关键节点，触发集群的相关管理工作。例如：

- 监听Zookeeper中的/brokers/ids节点，感知Broker增减变化。
- 监听/brokers/topics，感知topic以及对应的partition的增减变化。
- 监听/admin/delete_topic节点，处理删除topic的动作。

另外，Controller还需要负责将元数据推送给其他Broker。

3、Leader Partition选举机制

在Kafka中，一个Topic下的所有消息，是分开存储在不同的Partition中的。在使用kafka-topics.sh脚本创建Topic时，可以通过--partitions 参数指定Topic下包含多少个Partition，还可以通过--replication-factors参数指定每个Partition有几个备份。而在一个Partition的众多备份中，需要选举出一个Leader Partition，负责对接所有的客户端请求，并将消息优先保存，然后再通知其他Follower Partition来同步消息。

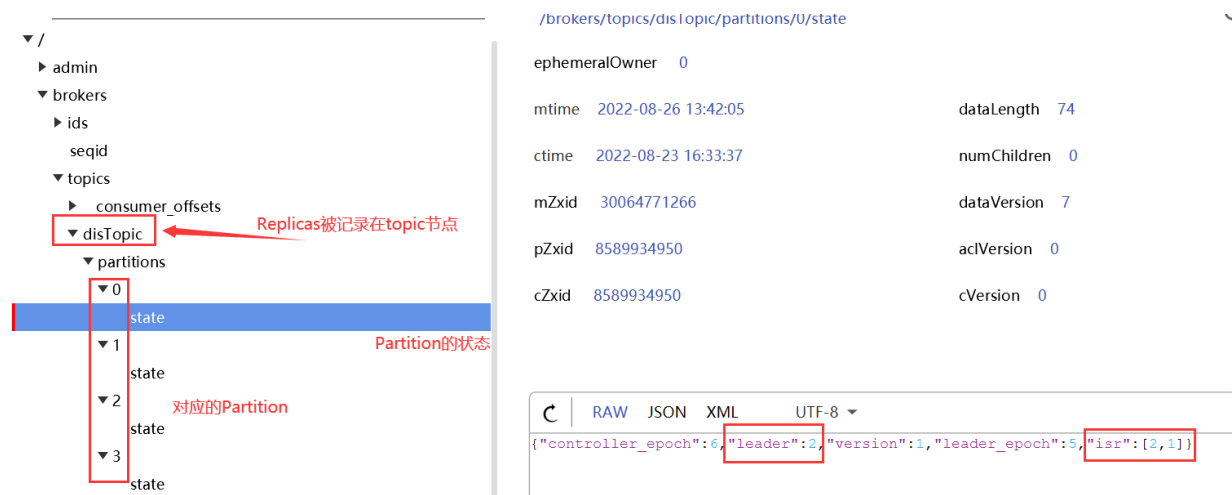
在理解Leader Partition选举机制前，需要了解几个基础的概念：

- AR: Assigned Replicas。表示Kafka分区中的所有副本(存活的不存活的)
- ISR: 表示在所有AR中，服务正常，保持与Leader同步的Follower集合。如果Follower长时间没有向Leader发送通信请求(超时时间由**replica.lag.time.max.ms**参数设定，默认30S)，那么这个Follower就会被提出ISR中。(在老版本的Kafka中，还会考虑Partition与Leader Partition之间同步的消息差值，大于参数replica.lag.max.messages条就会被移除ISR。现在版本已经移除了这个参数。)
- OSR: 表示从ISR中踢出的节点。记录的是那些服务有问题，延迟过多的副本。

其中，AR和ISR比较关键，可以通过kafka-topics.sh的--describe指令查看。

```
[oper@worker1 kafka_2.13-3.2.0]$ bin/kafka-topics.sh --bootstrap-server worker1:9092
--describe --topic disTopic
Topic: disTopic TopicId: vx4ohhIER6aDpDZgTy10tQ PartitionCount: 4
ReplicationFactor: 2   Configs: segment.bytes=1073741824
    Topic: disTopic Partition: 0   Leader: 2       Replicas: 2,1   Isr: 2,1
    Topic: disTopic Partition: 1   Leader: 1       Replicas: 1,0   Isr: 1,0
    Topic: disTopic Partition: 2   Leader: 2       Replicas: 0,2   Isr: 2,0
    Topic: disTopic Partition: 3   Leader: 2       Replicas: 2,0   Isr: 2,0
```

这个结果中，AR就是Replicas列中的Broker集合。而这个指令中的所有信息，其实都是被记录在Zookeeper中的。



接下来，Kafka设计了一套非常简单高效的Leader Partition选举机制。在选举Leader Partition时，会按照AR中的排名顺序，靠前的优先选举。只要当前Partition在ISR列表中，也就是是存活的，那么这个节点就会被选举成为Leader Partition。

例如，我们可以设计一个实验来验证一下LeaderPartiton的选举过程。

#1、创建一个备份因子为3的Topic，每个Partition有3个备份。

```
[oper@worker1 kafka_2.13-3.2.0]$ bin/kafka-topics.sh --bootstrap-server worker1:9092
--create --replication-factor 3 --partitions 4 --topic secondTopic
Created topic secondTopic.
```

#2、查看Topic的Partition情况 可以注意到，默认的Leader就是ISR的第一个节点。

```
[oper@worker1 kafka_2.13-3.4.0]$ bin/kafka-topics.sh --bootstrap-server worker1:9092
--describe --topic secondTopic
Topic: secondTopic   TopicId: W3mXDtj1RsWmsEhQrZjN5g PartitionCount: 4
ReplicationFactor: 3   Configs:
    Topic: secondTopic   Partition: 0   Leader: 1       Replicas: 1,0,2 Isr:
1,0,2
    Topic: secondTopic   Partition: 1   Leader: 0       Replicas: 0,2,1 Isr:
0,1,2
    Topic: secondTopic   Partition: 2   Leader: 2       Replicas: 2,1,0 Isr:
1,0,2
    Topic: secondTopic   Partition: 3   Leader: 1       Replicas: 1,2,0 Isr:
1,0,2
```

#3、在worker3上停掉brokerid=2的kafka服务。

```
[oper@worker3 kafka_2.13-3.2.0]$ bin/kafka-server-stop.sh
#4、再次查看SecondTopic上的Partition分区情况
[oper@worker1 kafka_2.13-3.4.0]$ bin/kafka-topics.sh --bootstrap-server worker1:9092
--describe --topic secondTopic
Topic: secondTopic      TopicId: W3mXDtj1RSWmsEhQrZjN5g PartitionCount: 4
ReplicationFactor: 3    Configs:
      Topic: secondTopic Partition: 0    Leader: 1      Replicas: 1,0,2 Isr:
1,0
      Topic: secondTopic Partition: 1    Leader: 0      Replicas: 0,2,1 Isr:
0,1
      Topic: secondTopic Partition: 2    Leader: 1      Replicas: 2,1,0 Isr:
1,0
      Topic: secondTopic Partition: 3    Leader: 1      Replicas: 1,2,0 Isr:
1,0
```

从实验中可以看到，当BrokerId=2的kafka服务停止后，2号BrokerId就从所有Partition的ISR列表中剔除了。然后，Partition2的Leader节点原本是Broker2，当Broker2的Kafka服务停止后，都重新进行了Leader选举。Partition2预先评估的是Replicas列表中Broker2后面的Broker1，Broker1在ISR列表中，所以他被最终选举成为Leader。

当Partition选举完成后，Zookeeper中的信息也被及时更新了。

```
/brokers/topics/secondTopic: {"partitions":{"0":[1,0,2],"1":[0,2,1],"2":[2,1,0],"3":
[1,2,0]},"topic_id":"W3mXDtj1RSWmsEhQrZjN5g","adding_replicas":
{},"removing_replicas":{},"version":3}
/brokers/topics/secondTopic/partitions/0/state:
{"controller_epoch":20,"leader":1,"version":1,"leader_epoch":2,"isr":[1,0]}
```

Leader Partition选举机制能够保证每一个Partition同一时刻有且仅有一个Leader Partition。**但是，是不是只要分配好了Leader Partition就够了呢？**

4、Leader Partition自动平衡机制

在一组Partition中，Leader Partition通常是比较繁忙的节点，因为他要负责与客户端的数据交互，以及向Follower同步数据。默认情况下，**Kafka会尽量将Leader Partition分配到不同的Broker节点上**，用以保证整个集群的性能压力能够比较平均。

但是，经过Leader Partition选举后，这种平衡就有可能会被打破，让Leader Partition过多的集中到同一个Broker上。这样，这个Broker的压力就会明显高于其他Broker，从而影响到集群的整体性能。

为此，Kafka设计了Leader Partition自动平衡机制，当发现Leader分配不均衡时，自动进行Leader Partition调整。

Kafka在进行Leader Partition自平衡时的逻辑是这样的：他会认为AR当中的第一个节点就应该是Leader节点。这种选举结果成为**preferred election 理想选举结果**。Controller会定期检测集群的Partition平衡情况，在开始检测时，Controller会依次检查所有的Broker。当发现这个Broker上的不平衡的Partition比例高于**leader.imbalance.per.broker.percentage**阈值时，就会触发一次Leader Partition的自平衡。

这是官方文档的部分截图。

Balancing leadership

Whenever a broker stops or crashes, leadership for that broker's partitions transfers to other replicas. When the broker is restarted it will only be a follower for all its partitions, meaning it will not be used for client reads and writes.

To avoid this imbalance, Kafka has a notion of preferred replicas. If the list of replicas for a partition is 1,5,9 then node 1 is preferred as the leader to either node 5 or 9 because it is earlier in the replica list. By default the Kafka cluster will try to restore leadership to the preferred replicas. This behaviour is configured with:

```
1 | auto.leader.rebalance.enable=true
```

You can also set this to false, but you will then need to manually restore leadership to the restored replicas by running the command:

```
1 | > bin/kafka-leader-election.sh --bootstrap-server broker_host:port --
```

这个机制涉及到Broker中server.properties配置文件中的几个重要参数:

#1 自平衡开关。默认true

auto.leader.rebalance.enable

Enables auto leader balancing. A background thread checks the distribution of partition leaders at regular intervals, configurable by ``leader.imbalance.check.interval.seconds``. If the leader imbalance exceeds ``leader.imbalance.per.broker.percentage``, leader rebalance to the preferred leader for partitions is triggered.

Type: boolean

Default: true

Valid values:

Importance: high

Update Mode: read-only

#2 自平衡扫描间隔

leader.imbalance.check.interval.seconds

The frequency with which the partition rebalance check is triggered by the controller

Type: long

Default: 300

Valid values: [1,...]

Importance: high

Update Mode: read-only

#3 自平衡触发比例

leader.imbalance.per.broker.percentage

The ratio of leader imbalance allowed per broker. The controller would trigger a leader balance if it goes above this value per broker. The value is specified in percentage.

Type: int
Default: 10
Valid Values:
Importance: high
Update Mode: read-only

这几个参数可以到broker的server.properties文件中修改。但是注意要修改集群中所有broker的文件，并且要重启Kafka服务才能生效。

另外，你也可以通过手动调用kafka-leader-election.sh脚本，触发一次自平衡。例如：

```
# 启动worker3上的Kafka服务，Broker2上线。
# secondTopic的partition2不是理想状态。理想的leader应该是2
[oper@worker1 kafka_2.13-3.4.0]$ bin/kafka-topics.sh --bootstrap-server worker1:9092
--describe --topic secondTopic
Topic: secondTopic      TopicId: W3mXDTj1RSWmsEhQrZjN5g PartitionCount: 4
ReplicationFactor: 3    Configs:
      Topic: secondTopic  Partition: 0    Leader: 1      Replicas: 1,0,2 Isr:
1,0,2
      Topic: secondTopic  Partition: 1    Leader: 0      Replicas: 0,2,1 Isr:
0,1,2
      Topic: secondTopic  Partition: 2    Leader: 1      Replicas: 2,1,0 Isr:
1,0,2
      Topic: secondTopic  Partition: 3    Leader: 1      Replicas: 1,2,0 Isr:
1,0,2
# 手动触发所有Topic的Leader Partition自平衡
[oper@worker1 bin]$ ./kafka-leader-election.sh --bootstrap-server worker1:9092 --
election-type preferred --topic secondTopic --partition 2
Successfully completed leader election (PREFERRED) for partitions secondTopic-2
# 自平衡后secondTopic的partition2就变成理想状态了。
[oper@worker1 kafka_2.13-3.4.0]$ bin/kafka-topics.sh --bootstrap-server worker1:9092
--describe --topic secondTopic
Topic: secondTopic      TopicId: W3mXDTj1RSWmsEhQrZjN5g PartitionCount: 4
ReplicationFactor: 3    Configs:
      Topic: secondTopic  Partition: 0    Leader: 1      Replicas: 1,0,2 Isr:
1,0,2
      Topic: secondTopic  Partition: 1    Leader: 0      Replicas: 0,2,1 Isr:
0,1,2
      Topic: secondTopic  Partition: 2    Leader: 2      Replicas: 2,1,0 Isr:
1,0,2
      Topic: secondTopic  Partition: 3    Leader: 1      Replicas: 1,2,0 Isr:
1,0,2
```

但是要注意，这样Leader Partition自平衡的过程是一个非常重的操作，因为要涉及到大量消息的转移与同步。并且，在这个过程中，会有丢消息的可能。所以在很多对性能要求比较高的线上环境，会选择将参数auto.leader.rebalance.enable设置为false，关闭Kafka的Leader Partition自平衡操作，而用其他运维的方式，在业务不繁忙的时间段，手动进行Leader Partiton自平衡，尽量减少自平衡过程对业务的影响。

至于为什么会丢消息。下一章节就会给出答案。

5、Partition故障恢复机制

Kafka设计时要面对的就是各种不稳定的网络以及服务环境。如果Broker的服务不稳定，随时崩溃，Kafka集群要怎么保证数据安全呢？

当一组Partition中选举出了一个Leader节点后，这个Leader节点就会优先写入并保存Producer传递过来的消息，然后再同步给其他Follower。当Leader Partition所在的Broker服务发生宕机时，Kafka就会触发Leader Partition的重新选举。但是，在选举过程中，原来Partition上的数据是如何处理的呢？

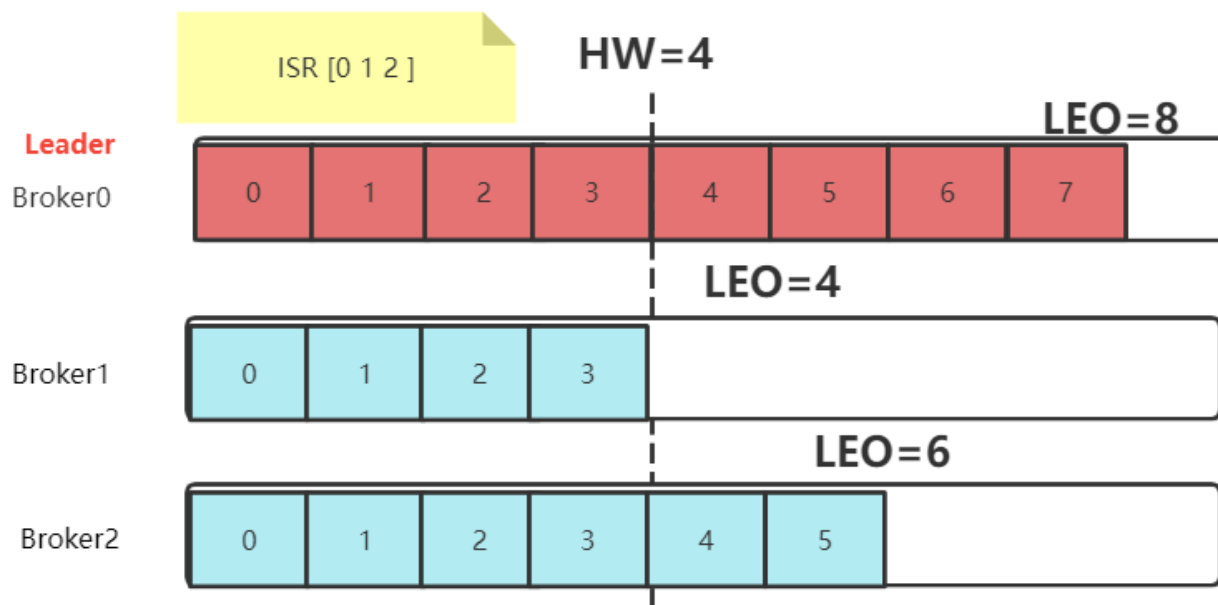
Kafka为了保证消息能够在多个Partition中保持数据同步，内部记录了两个关键的数据：

- LEO(Log End Offset): 每个Partition的最后一个Offset

这个参数比较好理解，每个Partition都会记录自己保存的消息偏移量。leader partition收到并记录了生产者发送的一条消息，就将LEO加1。而接下来，follower partition需要从leader partition同步消息，每同步到一个消息，自己的LEO就加1。通过LEO值，就知道各个follower partition与leader partition之间的消息差距。

- HW(High Watermark): 一组Partition中最小的LEO。

follower partition每次往leader partition同步消息时，都会同步自己的LEO给leader partition。这样leader partition就可以计算出这个HW值，并最终会同步给各个follower partition。leader partition认为这个HW值以前的消息，都是在所有follower partition之间完成了同步的，是安全的。这些安全的消息就可以被消费者拉取过去了。而HW值之后的消息，就是不安全的，是可能丢失的。这些消息如果被消费者拉取过去消费了，就有可能造成数据不一致。

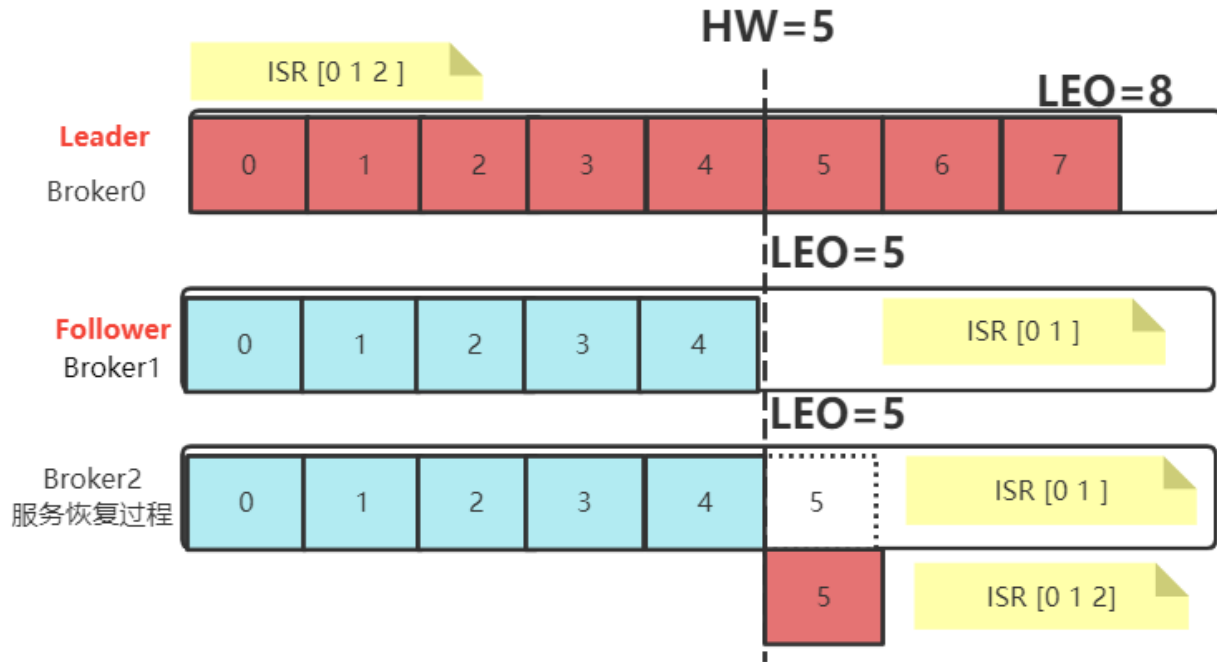


也就是说，在所有服务都正常的情况下，当一个消息写入到Leader Partition后，并不会立即让消费者感知。而是会等待其他Follower Partition同步。这个过程中就会推进HW。当HW超过当前消息时，才会让消费者感知。比如在上图中，4号往后的消息，虽然写入了Leader Partition，但是消费者是消费不到的。

这跟生产者的acks应答参数是不一样的

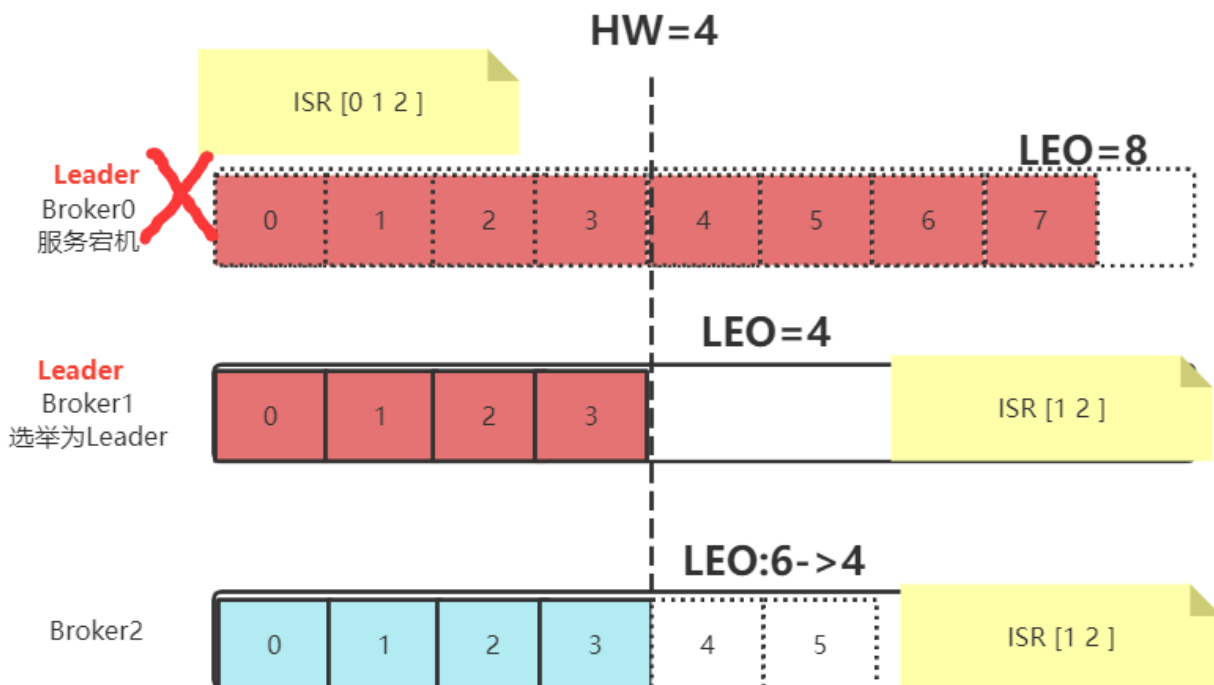
当服务出现故障时，如果是Follower发生故障，这不会影响消息写入，只不过是少了一个备份而已。处理相对简单一点。Kafka会做如下处理：

1. 将故障的Follower节点临时提出ISR集合。而其他Leader和Follower继续正常接收消息。
2. 出现故障的Follower节点恢复后，不会立即加入ISR集合。该Follower节点会读取本地记录的上一次的HW，将自己的日志中高于HW的部分信息全部删除掉，然后从HW开始，向Leader进行消息同步。
3. 等到该Follower的LEO大于等于整个Partiton的HW后，就重新加入到ISR集合中。这也就是说这个Follower的消息进度追上了Leader。



如果是Leader节点出现故障，Kafka为了保证消息的一致性，处理就会相对复杂一点。

1. Leader发生故障，会从ISR中进行选举，将一个原本是Follower的Partition提升为新的Leader。这时，消息有可能没有完成同步，所以新的Leader的LEO会低于之前Leader的LEO。
2. Kafka中的消息都只能以Leader中的备份为准。其他Follower会将各自的Log文件中高于HW的部分全部清理掉，然后从新的Leader中同步数据。
3. 旧的Leader恢复后，将作为Follower节点，进行数据恢复。



在这个过程中，Kafka注重的是保护多个副本之间的数据一致性。但是这样，消息的安全性就得不到保障。例如在上述示例中，原本Partition0中的4，5，6，7号消息就被丢失掉了。

从这个角度来看，在服务极端不稳定的极端情况下，Kafka为了保证高性能，其实是牺牲了数据安全性的。Kafka并没有保证消息绝对安全。而RocketMQ在这一方面做了改善，优先保证数据安全。后续学习RocketMQ时，可以对比Kafka理解一下。

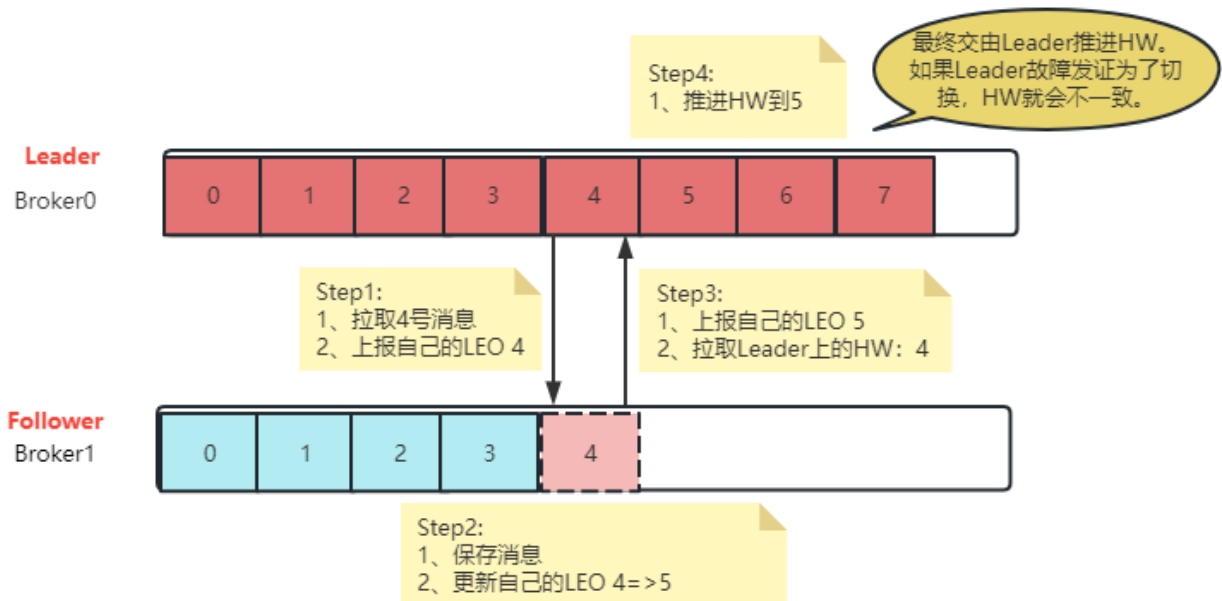
在这里你或许会有一个疑问，这个机制中有一个很重要的前提，就是各个Broker中记录的HW是一致的。但是HW和LEO同样是一个分布式的值，怎么保证HW在多个Broker中是一致的呢？

6、HW一致性保障-Epoch更新机制

有了HW机制后，各个Partiton的数据都能够比较好的保持统一。但是，实际上，**HW值在一组Partition里并不是总是一致的。**

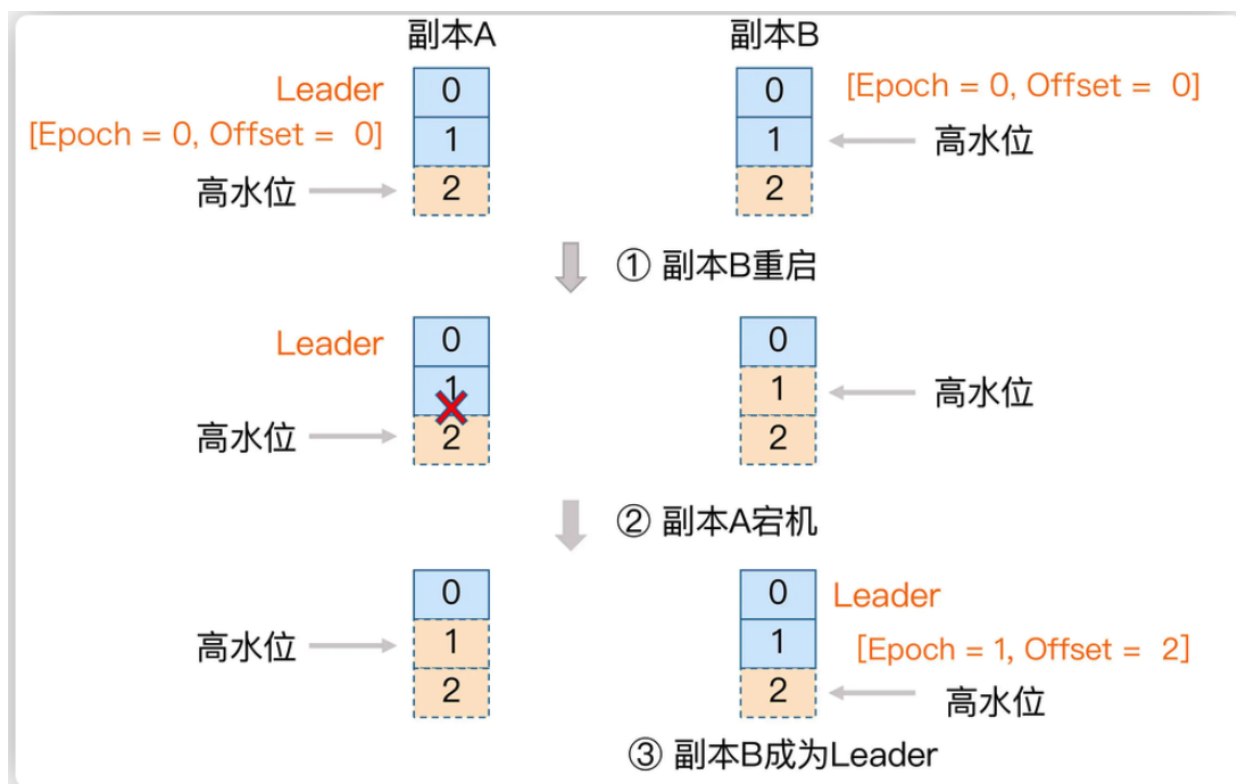
Leader Partition需要计算出HW值，就需要保留所有Follower Partition的LEO值。

但是，对于Follower Partition，他需要先将消息从Leader Partition拉取到本地，才能向Leader Partition上报LEO值。所有Follower Partition上报后，Leader Partition才能更新HW的值，然后Follower Partition在下次拉取消息时，才能更新HW值。所以，Leader Partiton的LEO更新和Follower Partition的LEO更新，在时间上是有延迟的。这也导致了Leader Partition上更新HW值的时刻与Follower Partition上跟新HW值的时刻，是会出现延迟的。这样，如果有多个Follower Partition，这些Partition保存的HW的值是不统一的。当然，如果服务一切正常，最终Leader Partition还是会正常推进HW，能够保证HW的最终一致性。但是，**当Leader Partition出现切换，所有的Follower Partition都按照自己的HW进行数据恢复，就会出现数据不一致的情况。**



因此，Kafka还设计了Epoch机制，来保证HW的一致性。

1. Epoch是一个单调递增的版本号，每当Leader Partition发生变更时，该版本号就会更新。所以，当有多个Epoch时，只有最新的Epoch才是有效的，而其他Epoch对应的Leader Partition就是过期的，无用的Leader。
2. 每个Leader Partition在上任之初，都会新增一个新的Epoch记录。这个记录包含更新后端的epoch版本号，以及当前Leader Partition写入的第一个消息的偏移量。例如(1,100)。表示epoch版本号是1，当前Leader Partition写入的第一条消息是100。Broker会将这个epoch数据保存到内存中，并且会持久化到本地一个leader-epoch-checkpoint文件当中。
3. 这个leader-epoch-checkpoint会在所有Follower Partition中同步。当Leader Partition有变更时，新的Leader Partition就会读取这个Epoch记录，更新后添加自己的Epoch记录。
4. 接下来其他Follower Partition要更新数据时，就可以不再依靠自己记录的HW值判断拉取消息的起点。而可以根据这个最新的epoch条目来判断。



这个关键的leader-epoch-checkpoint文件保存在Broker上每个partition对应的本地目录中。这是一个文本文件，可以直接查看。他的内容大概是这样样子的：

```
[oper@worker1 distopic-0]$ cat leader-epoch-checkpoint
0
1
29 2485991681
```

其中

第一行版本号

第二行表示下面的记录数。这两行数据没有太多的实际意义。

从第三行开始，可以看到两个数字。这两个数字就是epoch 和 offset。epoch就是表示leader的epoch版本。从0开始，当leader变更一次epoch就会+1。offset则对应该epoch版本的leader写入第一条消息的offset。可以理解为用户可以消费到的最早的消息offset。

7、章节总结

Kafka其实天生就是为了集群而生，即使单个节点运行Kafka，他其实也是作为一个集群运行的。而Kafka为了保证在各种网络抽风，服务器不稳定等复杂情况下，保证集群的高性能，高可用，高可扩展三高，做了非常多的设计。而这一章节，其实是从可见的Zookeeper注册信息为入口，理解Kafka的核心集群机制。回头来看今天总结的这些集群机制，其实核心都是为了保持整个集群中Partition内的数据一致性。有了这一系列的数据一致性保证，Kafka集群才能在复杂运行环境下保持高性能、高可用、高可扩展三高特性。而这其实也是我们去理解互联网三高问题最好的经验。