课程目标

1、掌握 Kafka Broker 文件存储结构分析

2、掌握 Kafka 消息保留和清理机制

3、掌握 Kafka 高可用及选举机制

4、掌握 Kafka 数据同步原理及故障处理

1. Kafka Broker 存储原理

1.1 文件的存储结构

配置文件：config/server.properties

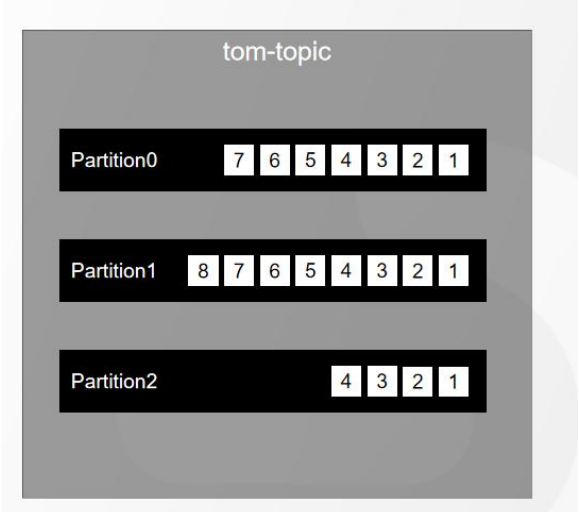
logs.dir 配置

默认/tmp/kafka-logs

1.1.1 Partition 分区

为了实现横向扩展，把不同的数据存放在不同的 Broker 上，同时降低单台服务器的访问压力，我们把一个 Topic 中的数据分隔成多个 Partition。

一个 Partition 中的消息是有序的，顺序写入，但是全局不一定有序。



在服务器上，每个 Partition 都有一个物理目录，Topic 名字后面的数字标号即代表分区。

```
a4part2repq-1
a4part2repq-2
a4part2repq-3
ass5part-0
ass5part-3
```

1.1.2 Replica 副本

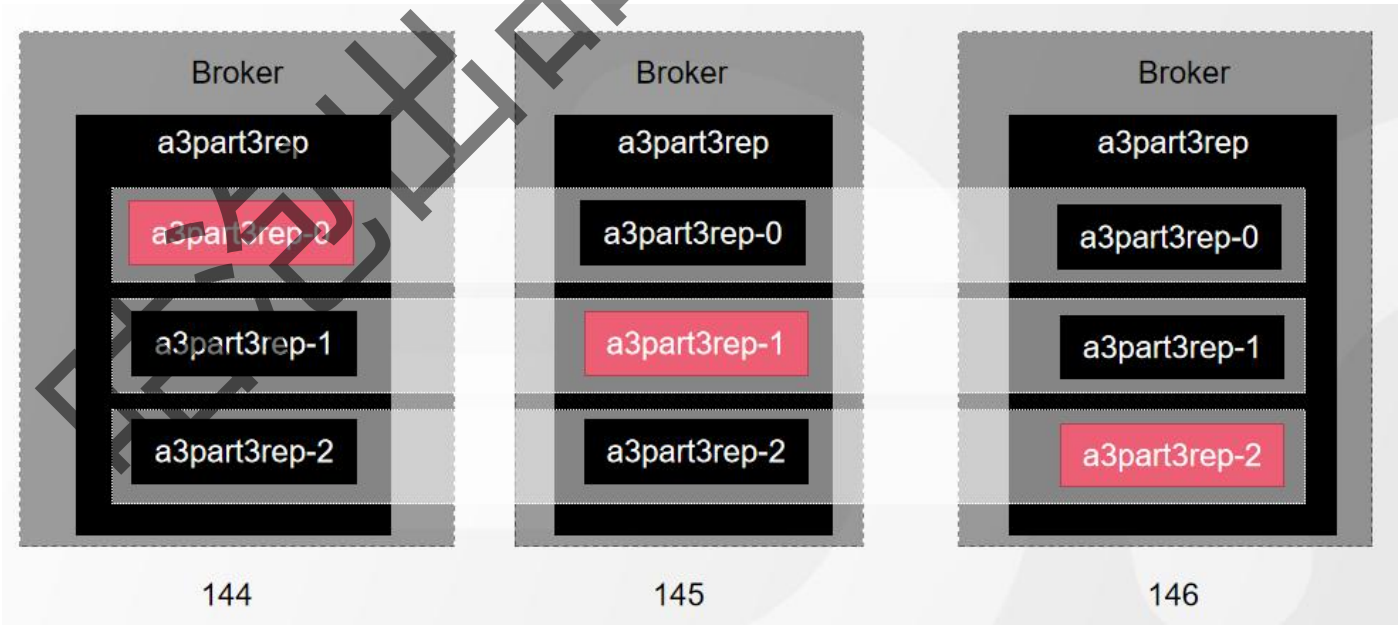
为了提高分区的可靠性，Kafka 又设计了副本机制。

创建 Topic 的时候，通过指定 replication-factor 确定 Topic 的副本数。

注意：副本数必须小于等于节点数，而不能大于 Broker 的数量，否则会报错。

```
./kafka-topics.sh --create --bootstrap-server 192.168.8.146:9092 --replication-factor 4 --partitions 1 --topic overrep
```

这样就可以保证，绝对不会有一个分区的两个副本分布在同一个节点上，不然副本机制也失去了备份的意义了。



这些所有的副本分为两种角色，Leader 对外提供读写服务。Follower 唯一的任务就是从 Leader 异步拉取数据。

思考：为什么不能像 MySQL 一样实现读写分离？写操作都在 Leader 上，读操作

都在 Follower 上。

这个是设计思想的不同。读写都发生在 Leader 节点，就不存在读写分离带来的一致性问题了，这个就叫做单调读一致性。

1.1.3 Leader 在哪里？

问题来了，如果分区有多个副本，哪一个节点上的副本是 Leader 呢？

```
./kafka-topics.sh --create --bootstrap-server 192.168.8.146:9092 --replication-factor 3 --partitions 3 --topic a3part3rep
```

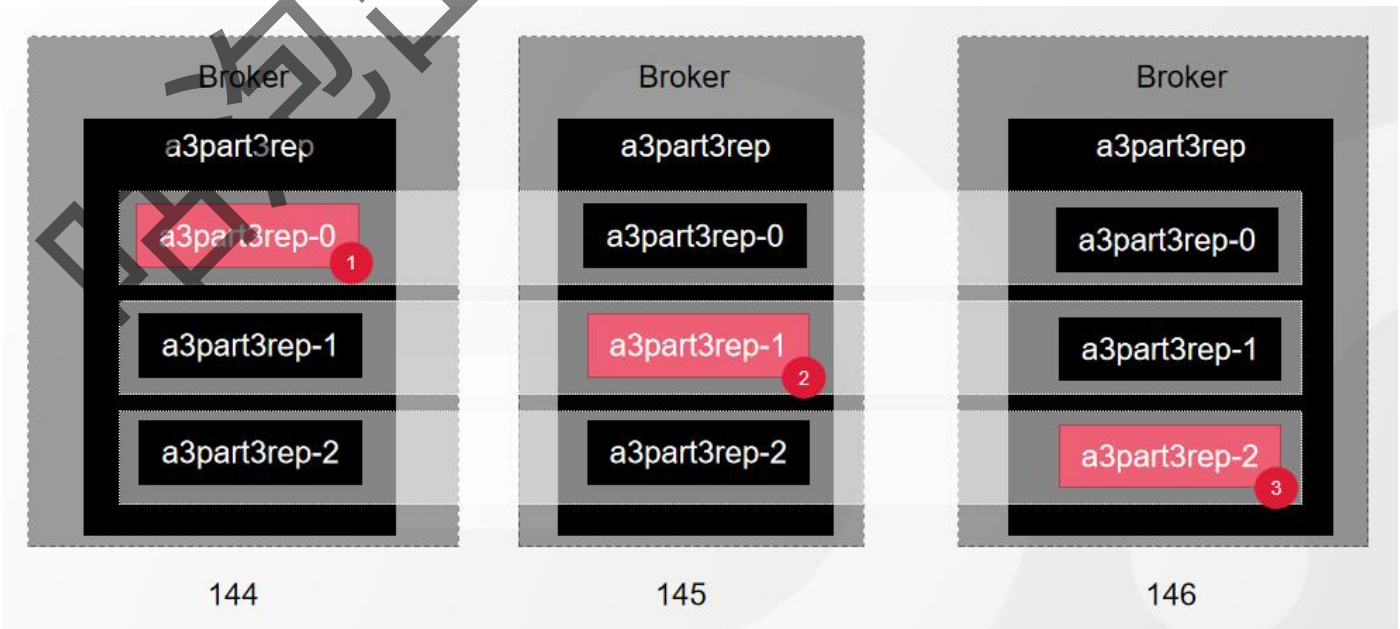
怎么查看所有的副本中谁是 Leader？（在集群上看，单机没有意义）

```
./kafka-topics.sh --topic a3part3rep --describe --bootstrap-server 192.168.8.146:9092
```

```
Topic: a3part3rep       TopicId: _eT5-babRc2hZcbRC51s5A PartitionCount: 3       ReplicationFactor: 3    Configs:
 segment.bytes=1073741824
        Topic: a3part3rep       Partition: 0    Leader: 146     Replicas: 146,144,145   Isr: 146,144,145
        Topic: a3part3rep       Partition: 1    Leader: 144     Replicas: 144,145,146   Isr: 144,145,146
        Topic: a3part3rep       Partition: 2    Leader: 145     Replicas: 145,146,144   Isr: 145,146,144
```

解释：这个 Topic 有 3 个分区 3 个副本。

第一个分区的 3 个副本在 146、144、145，ISR 中的信息对应也是 146、144、145。第一个副本是 Leader。



假设 Topic 有 4 个分区 2 个副本呢？

```
./kafka-topics.sh --create --bootstrap-server 192.168.8.146:9092 --replication-factor 2 --partitions 4 --topic a4part2rep
```

查看：

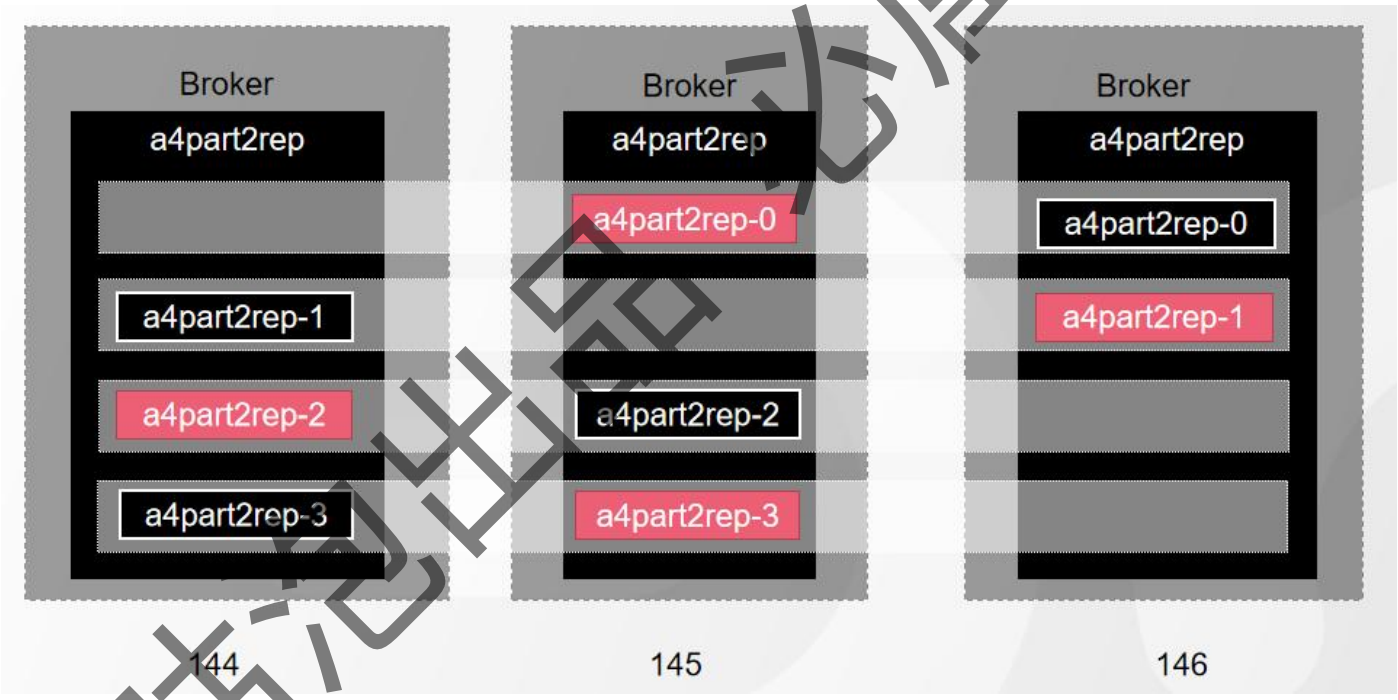
```
./kafka-topics.sh --topic a4part2rep --describe --bootstrap-server 192.168.8.146:9092
```

```
Topic: a4part2rep       TopicId: 1xUDqEUKSvmh0WdDsS2giQ PartitionCount: 4       ReplicationFactor: 2    Configs:
 segment.bytes=1073741824
        Topic: a4part2rep       Partition: 0    Leader: 145     Replicas: 145,146       Isr: 145,146
        Topic: a4part2rep       Partition: 1    Leader: 146     Replicas: 146,144       Isr: 146,144
        Topic: a4part2rep       Partition: 2    Leader: 144     Replicas: 144,145       Isr: 144,145
        Topic: a4part2rep       Partition: 3    Leader: 145     Replicas: 145,144       Isr: 145,144
```

解释：这个 topic 有 4 个分区 2 个副本。

第一个分区的 2 个副本编号 145、146，ISR 同步的也是 145、146。第 2 个副本是 Leader。

以此类推。图片表示出来是这样的：



副本 Leader 怎么选举后面再说。

咦，为什么第一个分区两个副本选择 145,146 的 Broker；第二个分区两个副本选择 146,144 的 Broker；第三个分区两个副本选择 144,145 的 Broker，都不一样呢？

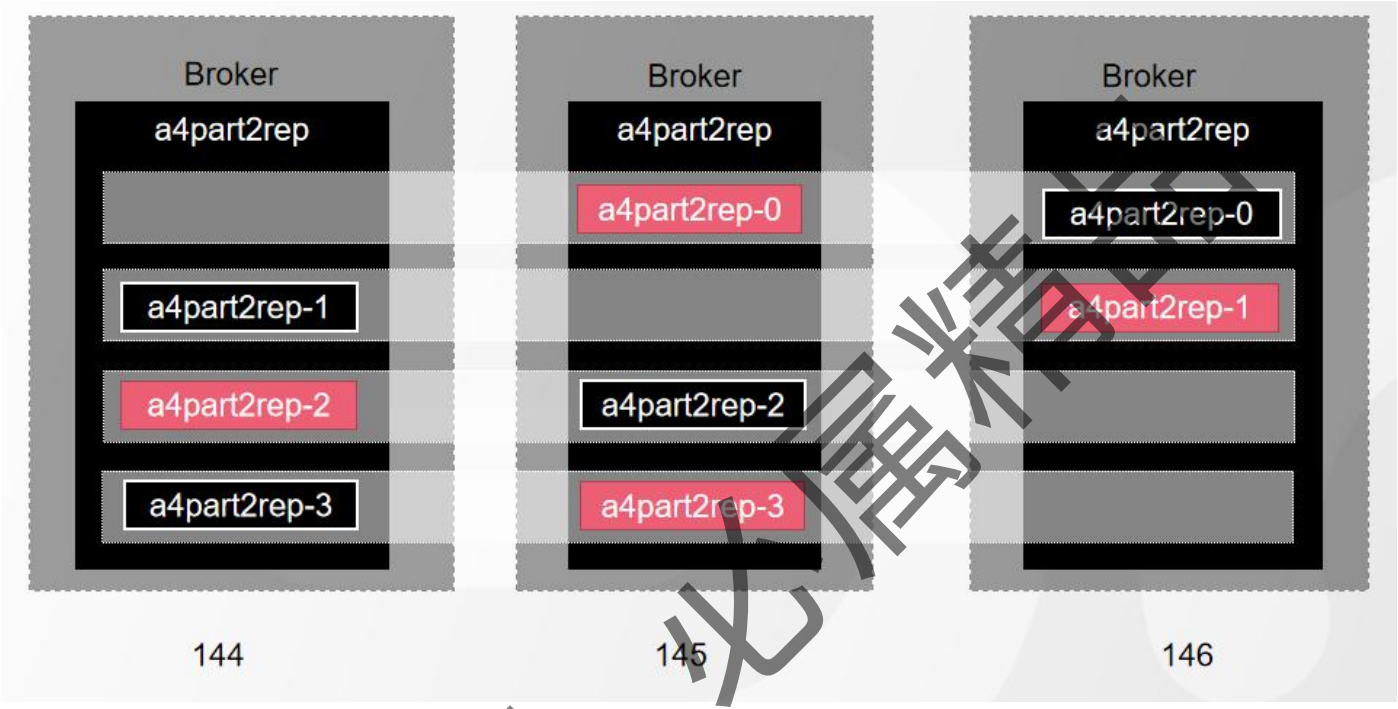
1.1.4 副本在 Broker 的分布

副本在 Broker 的分布有什么规则吗？

a4part2rep 这个 Topic，4 个分区每个 2 个副本，一共 8 份副本，怎么分布到 3 台机器?

理想情况肯定是 3、3、2，谁那么幸运有 2 个副本呢?

上面我们已经看到结果了:



第一个数据目录（/tmp/kafka-logs1）：

```
a4part2rep-1
a4part2rep-2
a4part2rep-3
```

第二个数据目录（/tmp/kafka-logs2）：

```
a4part2rep-0
a4part2rep-2
a4part2rep-3
```

第三个数据目录（/tmp/kafka-logs3）：

```
a4part2rep-0
a4part2rep-1
```

结果是确定的吗？可以换个名字再试一把（注意，如果要删除一个 Topic，要开启永久删除开关，否则只是标记删除）：

```
delete.topic.enable=true
```

删除：

```
./bin/kafka-topics.sh --delete --bootstrap-server 192.168.8.146:9092 --topic a4part2rep
```

实际上，分配策略是由 AdminUtils.scala 的 assignReplicasToBrokers 函数决定的。

规则如下：

1）firt of all，副本因子不能大于 Broker 的个数；

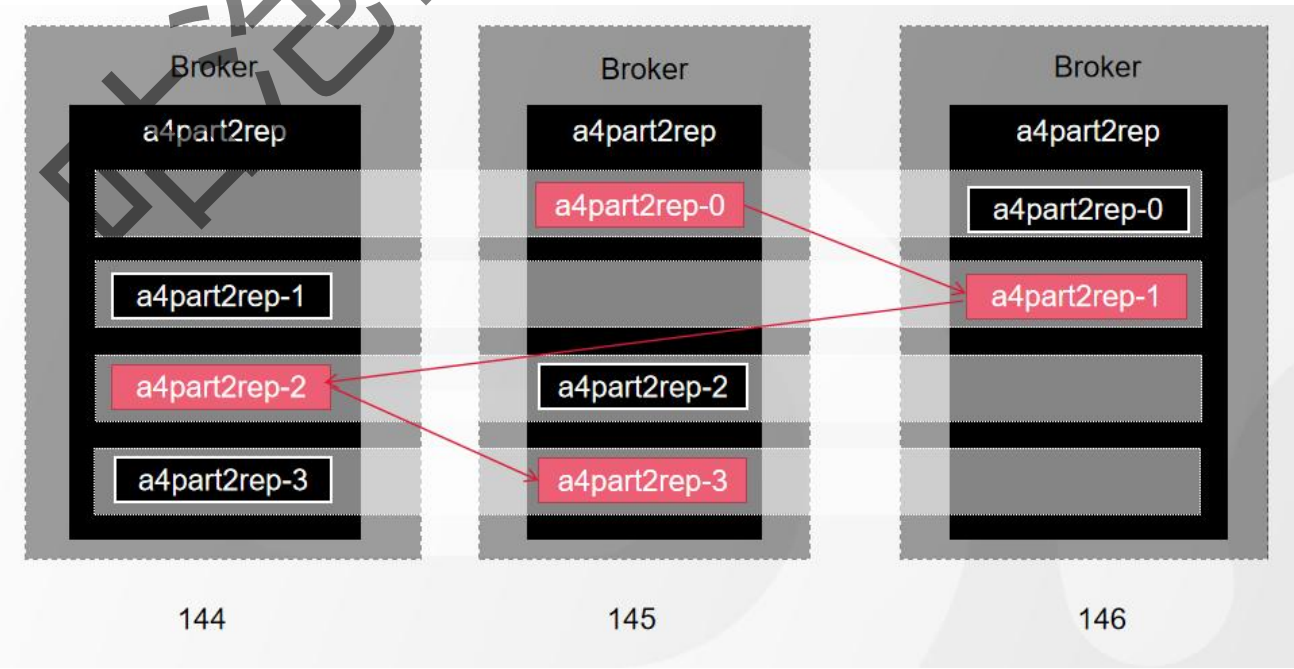
2）第一个分区（编号为 0 的分区）的第一个副本放置位置是随机从 brokerList 选择的（Broker2 的副本）；

3）其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。

也就是说：如果我们有 5 个 Broker，5 个分区，假设第 1 个分区的第 1 个副本放在第四个 Broker 上，那么第 2 个分区的第 1 个副本将会放在第五个 Broker 上；第三个分区的第 1 个副本将会放在第一个 Broker 上；第四个分区的第 1 个副本将会放在第二个 Broker 上，依次类推（蛇形走位）；

4）每个分区剩余的副本相对于第 1 个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的。

用箭头解释这个流程：



这样设计可以提高容灾能力。怎么讲?

在每个分区的第一个副本错开之后，一般第一个分区的第一个副本（按 Broker 编号排序）都是 Leader。Leader 是错开的，不至于一挂影响太大。

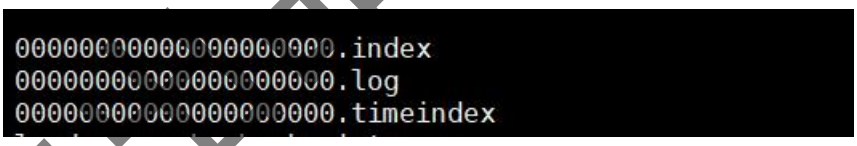
bin 目录下的 kafka-reassign-partitions.sh 可以根据 Broker 数量变化情况重新分配分区。

理一理，分区存储，分区副本分布我们都弄清楚了。那一个分区是不是只有一个文件呢？也就是时候，消息日志文件是不是会无限地变大?

1.1.5 Segment

为了防止 Log 不断追加导致文件过大，导致检索消息效率变低，一个 Partition 又被划分成多个 Segment 来组织数据（MySQL 也有 Segment 的逻辑概念，叶子节点就是数据段，非叶子节点就是索引段）。

在磁盘上，每个 Segment 由一个 log 文件和 2 个 index 文件组成。



这三个文件是成套出现的（其他的文件可以先忽略）。

leader-epoch-checkpoint 中保存了每一任 Leader 开始写入消息时的 Offset。

（1）.log 日志文件（日志就是数据）

在一个 Segment 文件里面，日志是追加写入的。如果满足一定条件，就会切分日志文件，产生一个新的 Segment。什么时候会触发 Segment 的切分呢?

第一种是根据日志文件大小。当一个 Segment 写满以后，会创建一个新的 Segment，用最新的 Offset 作为名称。这个例子可以通过往一个 Topic 发送大量消息

产生。

Segment 的默认大小是 1073741824 bytes（1G），由这个参数控制：

```
log.segment.bytes
```

第二种是根据消息的最大时间戳，和当前系统时间戳的差值。

有一个默认的参数，168 个小时（一周）：

```
log.roll.hours=168
```

意味着：如果服务器上次写入消息是一周之前，旧的 Segment 就不写了，现在要创建一个新的 Segment。

还可以从更加精细的时间单位进行控制，如果配置了毫秒级别的日志切分间隔，会优先使用这个单位。否则就用小时的。

```
log.roll.ms
```

Segment 日志切分的第三种情况：

Offset 索引文件或者 timestamp 索引文件达到了一定的大小，默认是 10485760 字节（10M）。如果要减少日志文件的切分，可以把这个值调大一点。

```
log.index.size.max.bytes
```

亦即：索引文件写满了，数据文件也要跟着拆分，不然这一套东西对不上。

另外两个是索引文件，单独来看。

（2）.index 偏移量（Offset）索引文件

（3）.timeindex 时间戳（timestamp）索引文件

1.1.6 索引

由于一个 Segment 的文件里面可能存放很多消息，如果要根据 Offset 获取消息，必须要有一种快速检索消息的机制。这个就是索引。在 Kafka 中设计了两种索引。

偏移量索引文件记录的是 Offset 和消息物理地址（在 Log 文件中的位置）的映射关系。时间戳索引文件记录的是时间戳和 Offset 的关系。

当然，内容是二进制的文件，不能以纯文本形式查看。bin 目录下有 dumplog 工具。

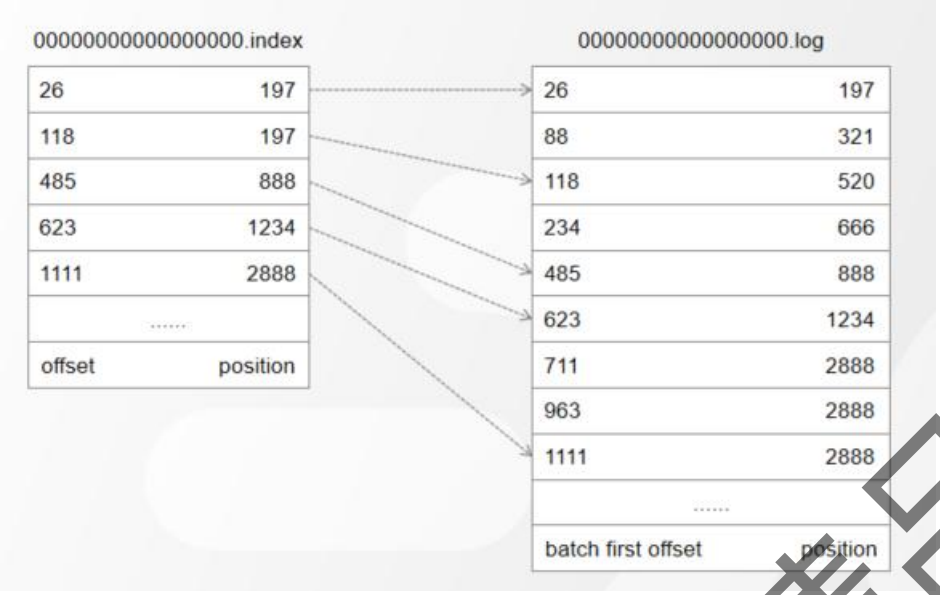
查看最后 10 条 Offset 索引（160 服务器）：

```
./kafka-dump-log.sh --files
/tmp/kafka-logs/mytopic-0/00000000000000000000.index|head -n 10
```

```
offset: 1495549 position: 12785491
offset: 1496368 position: 12801868
offset: 1497187 position: 12818245
offset: 1498006 position: 12834622
offset: 1498825 position: 12850999
offset: 1499644 position: 12867376
offset: 1500463 position: 12883753
offset: 1501282 position: 12900130
offset: 1502101 position: 12916507
offset: 1502129 position: 12932884
```

为什么索引里面记录的 Offset 不是连续的呢？不应该是一条消息一条索引记录吗？

注意 Kafka 的索引并不是每一条消息都会建立索引，而是一种稀疏索引 Sparse Index（DB2 和 MongDB 中都有稀疏索引）。



所以问题就来了，这个稀疏索引到底有多稀疏？也就是说，隔几条消息才产生一个索引记录？或者隔多久？或者隔多少大小的消息？

实际上是用消息的大小来控制的，默认是 4KB：

```
log.index.interval.bytes=4096
```

只要写入的消息超过了 4KB，偏移量索引文件.index 和时间戳索引文件.timeindex 就会增加一条索引记录（索引项）。

这个值设置越小，索引越密集。值设置越大，索引越稀疏。

相对来说，越稠密的索引检索数据更快，但是会消耗更多的存储空间。

越的稀疏索引占用存储空间小，但是插入和删除时所需的维护开销也小。

Kafka 索引的时间复杂度为 O(log2n)+O(m),n 是索引文件里索引的个数，m 为稀疏程度。

第二种索引类型是时间戳索引。

为什么会有时间戳索引文件呢？光有 Offset 索引还不够吗？会根据时间戳来查找消息吗？

首先消息是必须要记录时间戳的。客户端封装的 ProducerRecord 和 ConsumerRecord 都有一个 long timestamp 属性。

为什么要记录时间戳呢？

1、如果要基于时间切分日志文件，必须要记录时间戳；

2、如果要基于时间清理消息，必须要记录时间戳。

好吧，既然你都已经记录时间戳了，干脆我设计一个时间戳索引，可以根据时间戳查询。

注意时间戳有两种，一种是消息创建的时间戳，一种是消费在 Broker 追加写入的时间。到底用哪个时间呢？由一个参数来控制：

```
log.message.timestamp.type=CreateTime
```

默认是创建时间。如果要改成日志追加时间，则修改为 LogAppendTime。

查看最早的 10 条时间戳索引：

```
./kafka-dump-log.sh --files /tmp/kafka-logs/mytopic-0/00000000000000000000.timeindex|head -n 10
```

```
timestamp: 1594885590279 offset: 851364
timestamp: 1594885590280 offset: 852183
timestamp: 1594885590282 offset: 853821
timestamp: 1594885590284 offset: 854640
timestamp: 1594885590301 offset: 855459
timestamp: 1594885590309 offset: 856278
timestamp: 1594885590312 offset: 857097
timestamp: 1594885590313 offset: 857916
timestamp: 1594885590320 offset: 858735
```

Kafka 如何基于索引快速检索消息？比如我要检索偏移量是 10000666 的消息。

1、消费的时候是能够确定分区的，所以第一步是找到在哪个 Segment 中。Segment 文件是用 Base Offset 命名的，所以可以用二分法很快确定（找到名字不小于 10000666 的 Segment）。

2、这个 Segment 有对应的索引文件，它们是成套出现的。所以现在要在索引文件中根据 Offset 找 Position。

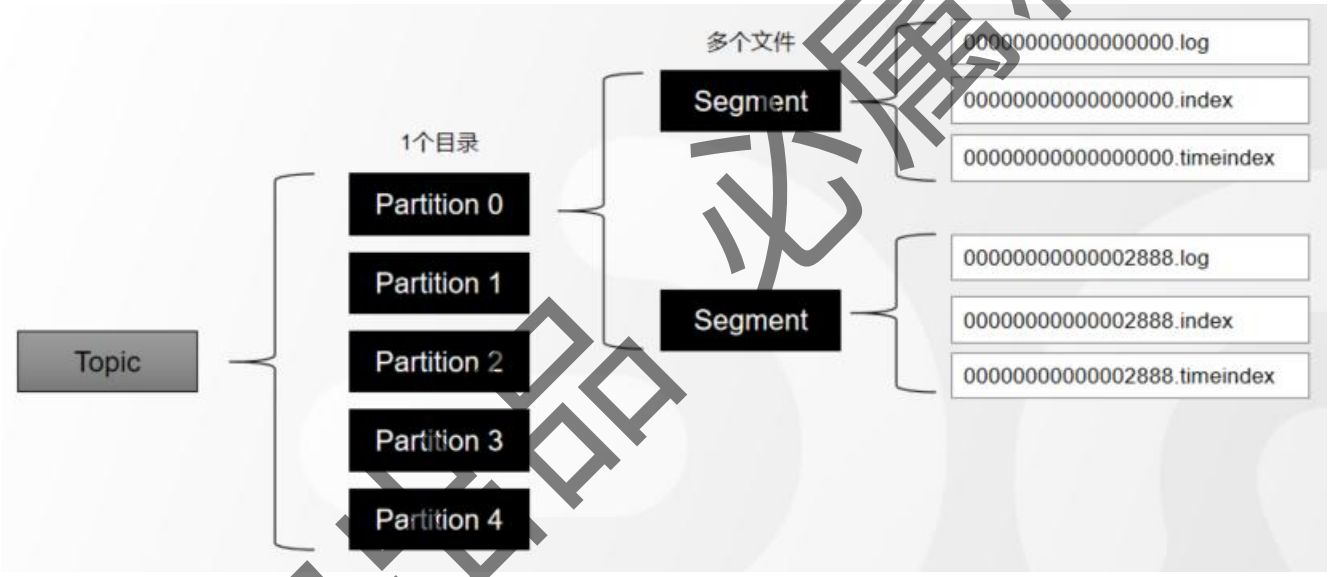
3、得到 Position 之后，到对应的 Log 文件开始查找 Offset，和消息的 Offset 进行比较，直到找到消息

思考一个比较刁钻的面试问题：为什么 Kafka 不用 B+Tree?

Kafka 是写多，查少。如果 Kafka 用 B+Tree，首先会出现大量的 B+Tree，大量插入数据带来的 B+Tree 的调整会非常消耗性能。

1.1.7 总结

总体结构：



好了，我知道你会分段了，但是还有一个问题。

前面我们说了，因为 Kafka 的日志（数据）在消费以后不删除，所以可以顺序追加写入。但是，如果把几个月几年以前的日志全部保留的话，磁盘肯定会被撑爆，而这些数据我们可能根本用不到。

1.2 消息保留（清理）机制

1.2.1 开关与策略

所以对于这些 82 年的日志，我们需要有一定的清理策略。

（大家可以想一下 Log4j 的日志文件的清理策略；MySQL InnoDB Buffer Pool 清理；Redis 内存淘汰）开关默认是开启的：

```
log.cleaner.enable=true
```

Kafka 里面提供了两种方式，一种是直接删除 Delete，一种是对日志进行压缩 Compact。默认是直接删除。

```
log.cleanup.policy=delete
```

1.2.2 删除策略

如果是删除，什么时候删除呢？是不是选一个黄道吉日？背后有一个勤劳的家伙，像教导主任一样时不时来看看，如果有染头发和长头发的学生就把它抓走。日志删除是通过定时任务实现的。默认 5 分钟执行一次，看看有没有需要删除的数据。

```
log.retention.check.interval.ms=300000
```

删除从哪里开始删呢？肯定是从最老的数据开始删。关键就是对于老数据的定义。什么才是老数据的？（请问，90 后是老男人吗？）

由一个参数来控制，默认：

```
log.retention.hours
```

默认值是 168 个小时（一周），也就是时间戳超过一周的数据才会删除。

Kafka 另外也提供了另外两个粒度更细的配置，分钟和毫秒。

```
log.retention.minutes
```

默认值是空。它的优先级比小时高，如果配置了则用这个。

```
log.retention.ms
```

默认值是空。它的优先级比分钟高，如果配置了则用这个。

这里还有一种情况，假设 Kafka 产生消息的速度是不均匀的，有的时候一周几百万条，有的时候一周几千条，那这个时候按照时间来删除就不是那么合理了，所以第二种删除策略就是根据日志大小删除，先删旧的消息，删到不超过这个大小为止。

```
log.retention.bytes
```

默认值是-1，代表不限制大小，想写多少就写多少。log.retention.bytes 指的是所有日志文件的总大小。也可以对单个 Segment 文件大小进行限制。

```
log.segment.bytes
```

默认值 1073741824 字节（1G）。

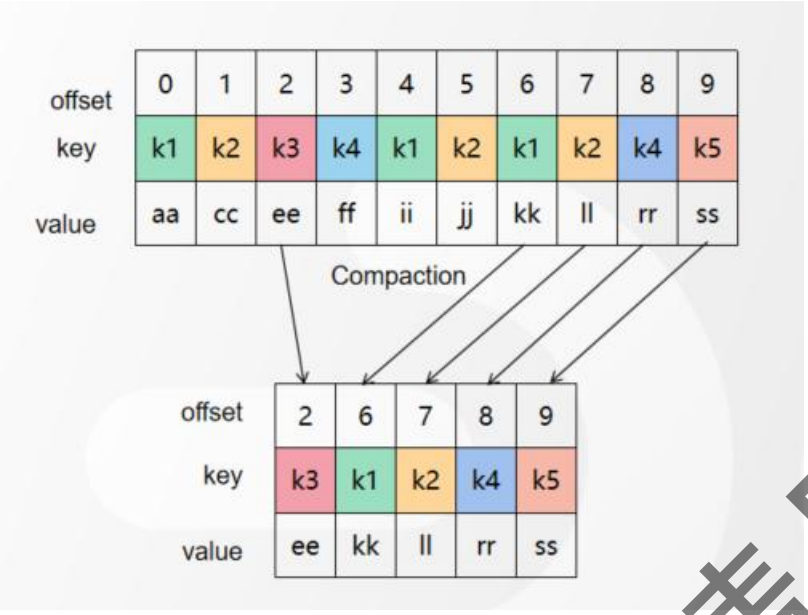
1.2.3 压缩策略

第二种策略是不删除，而是对日志数据进行压缩。

问题：如果同一个 Key 重复写入多次，会存储多次还是会更新?

比如用来存储位移的这个特殊的 topic：__consumer_offsets，存储的是消费者 id 和 Partition 的 Offset 关系，消费者不断地消费消息 commit 的时候，是直接更新原来的 Offset，还是不断地写入新的 Offset 呢？肯定是存储多次，不然怎么能实现顺序写。

当有了这些 Key 相同的 Value 不同的消息的时候，存储空间就被浪费了。压缩就是把相同的 Key 合并为最后一个 Value。



这个压缩跟 Compression 的含义不一样。所以，这里称为压紧更加合适。

Log Compaction 执行过后的偏移量不再是连续的，不过这并不影响日志的查询。

数据存储这一块我们已经了解了，包括分区怎么分布，索引检索数据，日志怎么清理。

接下来有一个很重要的问题是当 Leader 宕机了，怎样在 Follower 中选举出新的 Leader。设计副本不就是为了这个吗？所谓养兵千日用兵一时。

1.3 高可用架构

1.3.1 Controller 选举

当创建添加一个的分区或者分区增加了副本的时候，都要从所有副本中选举一个新的 Leader 出来。

投票怎么玩？是不是所有的 Partition 副本直接发起投票，开始猜拳呢？比如用 ZK 实现。

利用 ZK 怎么实现选举？ ZK 的什么功能可以感知到节点的变化（增加或者减少）？或者说，ZK 为什么能实现加锁和释放锁?

3 个特点： Watch 机制；节点不允许重复写入；临时节点。

这样实现是比较简单，但是也会存在一定的弊端。如果分区和副本数量过多，所有的副本都直接进行选举的话，一旦某个出现节点的增减，就会造成大量的 Watch 事件被触发，ZK 的就会负载过重，不堪重负。

Kafka 早期的版本就是这样做的，后来换了一种实现方式。

不是所有的 Repalica 都参与 Leader 选举，而是由其中的一个 Broker 统一来指挥，这个 Broker 的角色就叫做 Controller（控制器）。

就像 Redis Sentinel 的架构，执行故障转移的时候，必须要先从所有哨兵中选一个负责做故障转移的节点一样。 Kafka 也要先从所有 Broker 中选出唯一的一个 Controller。

所有的 Broker 会尝试在 Zookeeper 中创建临时节点/controller，只有一个能创建成功（先到先得）。

如果 Controller 挂掉了或者网络出现了问题，ZK 上的临时节点会消失。其他的 Broker 通过 Watch 监听到 Controller 下线的消息后，开始竞选新的 Controller。方法跟之前还是一样的，谁先在 ZK 里面写入一个/controller 节点，谁就成为新的 Controller，这个 Controller 就相当于选举委员会的主席。

一个节点成为 Controller 之后，它肩上的责任也比别人重了几份，正所谓劳力越戴，责任越大：

- 监听 Broker 变化。
- 监听 Topic 变化。
- 监听 Partition 变化。
- 获取和管理 Broker、Topic、Partition 的信息。
- 管理 Partiontion 的主从信息。

1.3.2 分区副本 Leader 选举

https://kafka.apache.org/documentation/#replication

https://kafka.apache.org/documentation/#design_replicatedlog

Controller 确定以后，就可以开始做分区选主的事情了（我们叫它选举委员会主席）。下面就是找候选人了。显然，每个 Replica 都想推荐自己，但是所有的 replica 都有竞选资格吗?

并不是。这里要给大家说几个概念。

一个分区所有的副本，叫做 Assigned-Replicas（AR）。所有的皇太子。

这些所有的副本中，跟 Leader 数据保持一定程度同步的，叫做 In-Sync Replicas（ISR）。天天过来参加早会的，有希望继位的皇太子。

跟 Leader 同步滞后过多的副本，叫做 Out-Sync-Replicas（OSR）。天天睡懒觉，不参加早会，没被皇帝放在眼里的皇太子。

AR=ISR+OSR。正常情况下 OSR 是空的，大家都正常同步，AR=ISR。

谁能够参加选举呢？肯定不是 AR，也不是 OSR，而是 ISR。而且这个 ISR 不是固定不变的，还是一个动态的列表。

前面我们说过，如果同步延迟超过 30 秒，就踢出 ISR，进入 OSR；如果赶上来了，就加入 ISR。

默认情况下，当 Leader 副本发生故障时，只有在 ISR 集合中的副本才有资格被选举为新的 Leader。

如果 ISR 为空呢？皇帝突然驾崩，太子们都还小，但是群龙不能无首。在这种情况下，可以让 ISR 之外的副本参与选举。允许 ISR 之外的副本参与选举，叫做 Unclean Leader Election。

```
unclean.leader.election.enable=false
```

把这个参数改成 true（一般情况不建议开启，会造成数据丢失）。

好了，委员会主席有了，候选人也确定了，终于可以选举了吧？根据什么规则确定 Leader 呢？猜拳吗？

首先第一个问题：分布式系统中常见的选举协议有哪些（或者说共识算法）？

ZAB（ZK）、Raft（Redis Sentinel）（他们都是 Paxos 算法的变种），它们的思想归纳起来都是：先到先得、少数服从多数。

但是 Kafka 没有用这些方法，而是用了一种自己实现的算法。

为什么呢？比如 ZAB 这种协议，可能会出现脑裂（节点不能互通的时候，出现多个 Leader）、惊群效应（大量 Watch 事件被触发）。

在这篇文章中：

https://kafka.apache.org/documentation/#design_replicatedlog

提到 Kafka 的选举实现，最相近的是微软的 PacificA 算法。

There are a rich variety of algorithms in this family including ZooKeeper's Zab, Raft, and Viewstamped Replication. The most similar academic publication we are aware of to Kafka's actual implementation is PacificA from Microsoft.

在这种算法中，默认是让 ISR 中第一个 Replica 变成 Leader。比如 ISR 是 1、5、9，优先让 1 成为 Leader。这个跟中国古代皇帝传位是一样的，优先传给皇长子。

Balancing leadership

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.

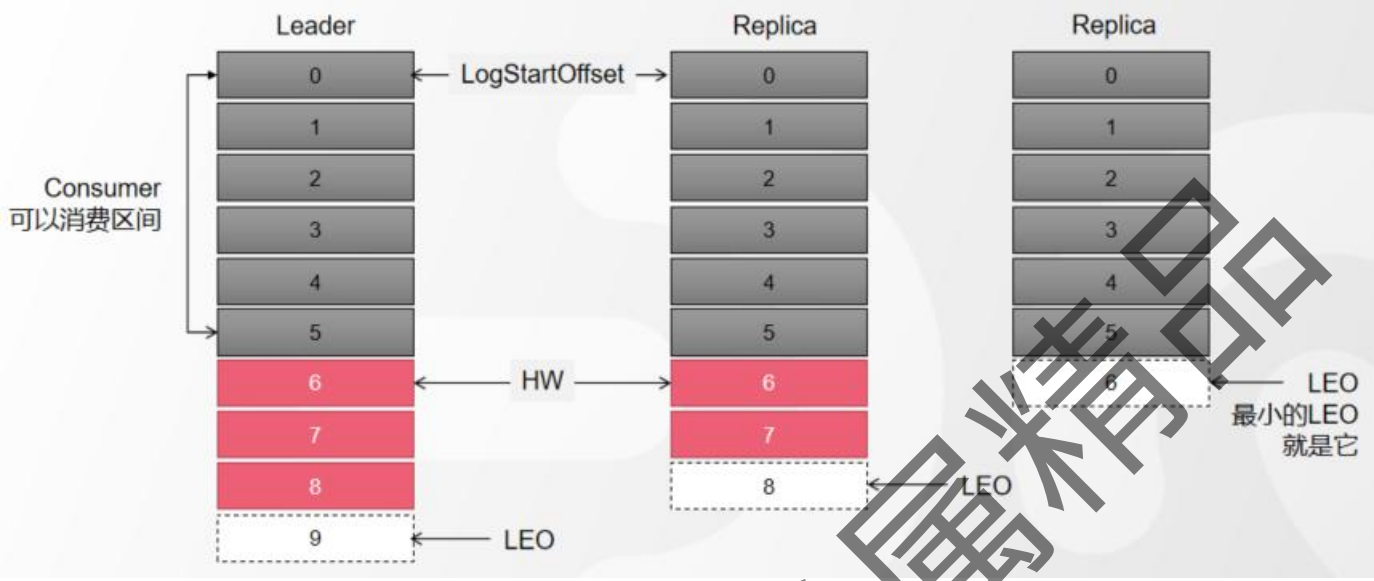
1.3.3 主从同步

Leader 确定之后，客户端的读写只能操作 Leader 节点。Follower 需要向 Leader

同步数据。

不同的 Raplica 的 Offset 是不一样的，同步到底怎么同步呢?

这里又要先讲解几个概念了。



LEO（Log End Offset）：下一条等待写入的消息的 Offset（最新的 Offset + 1），图中分别是 9，8，6。可以用命令看到:

```
./kafka-consumer-groups.sh --bootstrap-server 192.168.8.146:9092 --describe
--group gp-test-group
```

```
PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
0          4              9               5
```

这个命令查看分区对应的 Offset:

```
./kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list 192.168.8.146:9092 --topic 'mytopic' --time -1
```

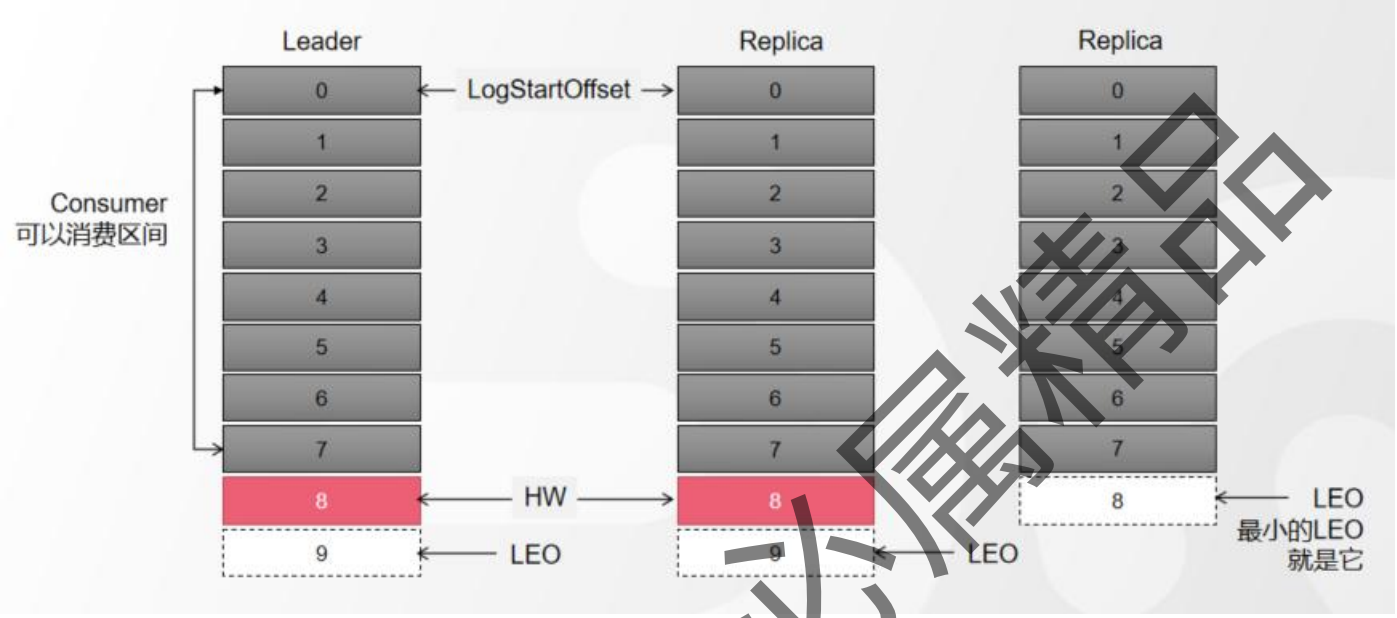
HW（Hign Watermark）：ISR 中最小的 LEO。Leader 会管理所有 ISR 中最小的 LEO 作为 HW，目前是 6。

Consumer 最多只能消费到 HW 之前的位置（消费到 Offset 5 的消息）。也就是说：其他的副本没有同步过去的消息，是不能被消费的。

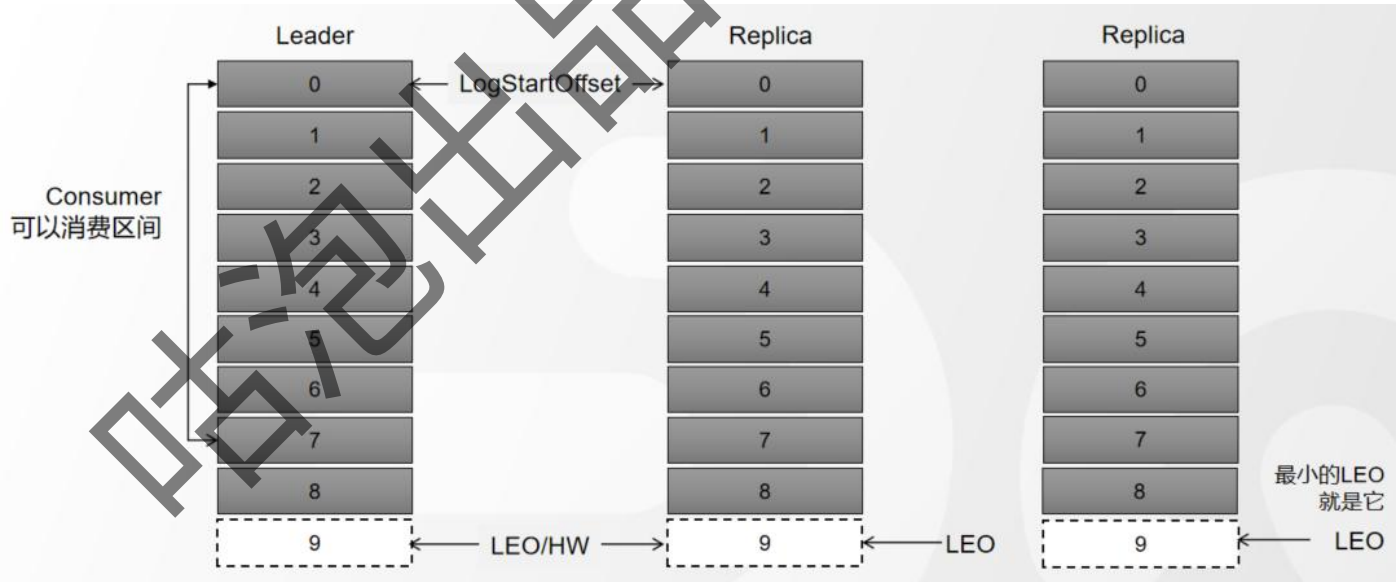
为什么要这样设计呢？ 如果在同步成功之前就被消费了，Consumer Group 的 Offset 会偏大。如果 Leader 崩溃，中间会缺失消息。

有了这两个 Offset 之后，再来看看消息怎么同步。

Follower1 同步了 1 条消息，Follower2 同步了 2 条消息。此时 HW 推进了 2，变成 8。



Follower1 同步了 0 条消息，Follower2 同步了 1 条消息。此时 HW 推进了 1，变成 9。LEO 和 HW 重叠，所有的消息都可以消费了。



这里，我们关注一下，从节点怎么跟主节点保持同步？

1、Follower 节点会向 Leader 发送一个 fetch 请求，Leader 向 Follower 发送数据后，既需要更新 Follower 的 LEO。

2、Follower 接收到数据响应后，依次写入消息并且更新 LEO。

3、Leader 更新 HW（ISR 最小的 LEO）。

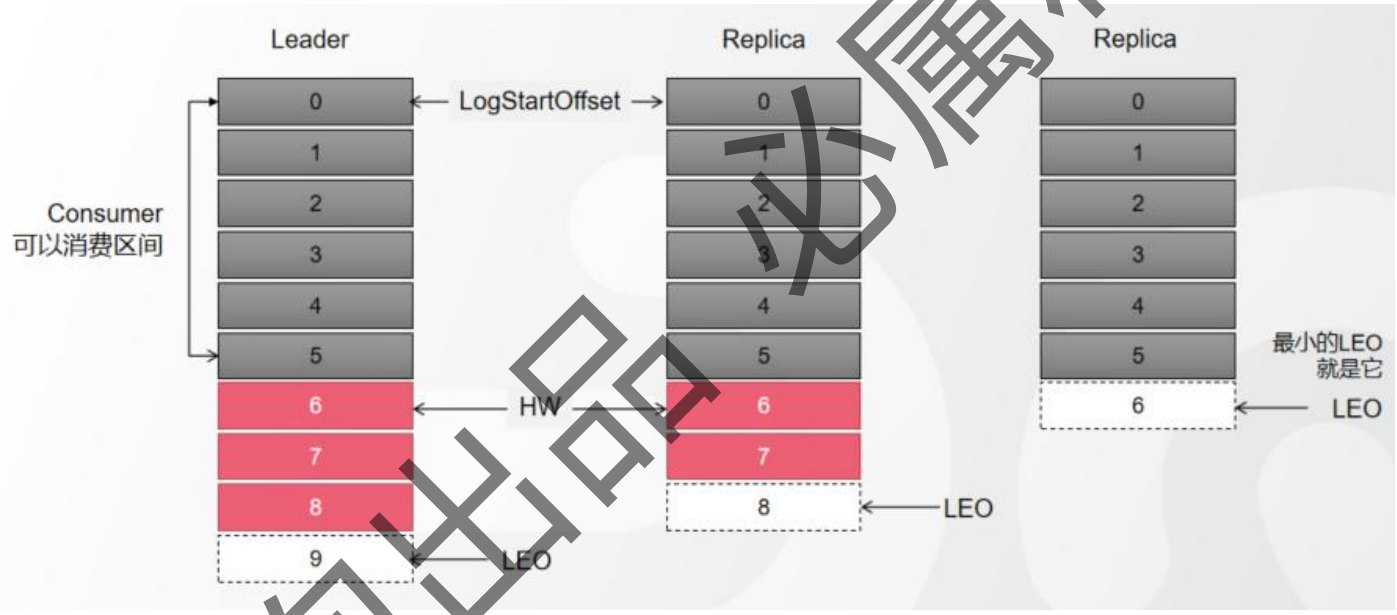
Kafka 设计了独特的 ISR 复制，可以在保障数据一致性情况下又可提供高吞吐量。

1.3.4 Replica 故障处理

Follower 故障

首先 Follower 发生故障，会被先踢出 ISR。

Follower 恢复之后，从哪里开始同步数据呢？假设第 1 个 Replica 宕机（中间这个）。



恢复以后，首先根据之前记录的 HW（6），把高于 HW 的消息截掉（6、7）。然后向 Leader 同步消息。追上 leader 之后（30 秒），重新加入 ISR。

Leader 故障

假设图中 Leader 发生故障。

首先选一个 Leader。因为 Replica 1（中间这个）优先，它成为 Leader。

为了保证数据一致，其他的 Follower 需要把高于 HW 的消息截取掉（这里没有消息需要截取）。

然后 Replica2 同步数据。

注意：这种机制只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。