尚硅谷大数据技术之 Kafka

(作者：尚硅谷大数据研发部)

版本：V2.0

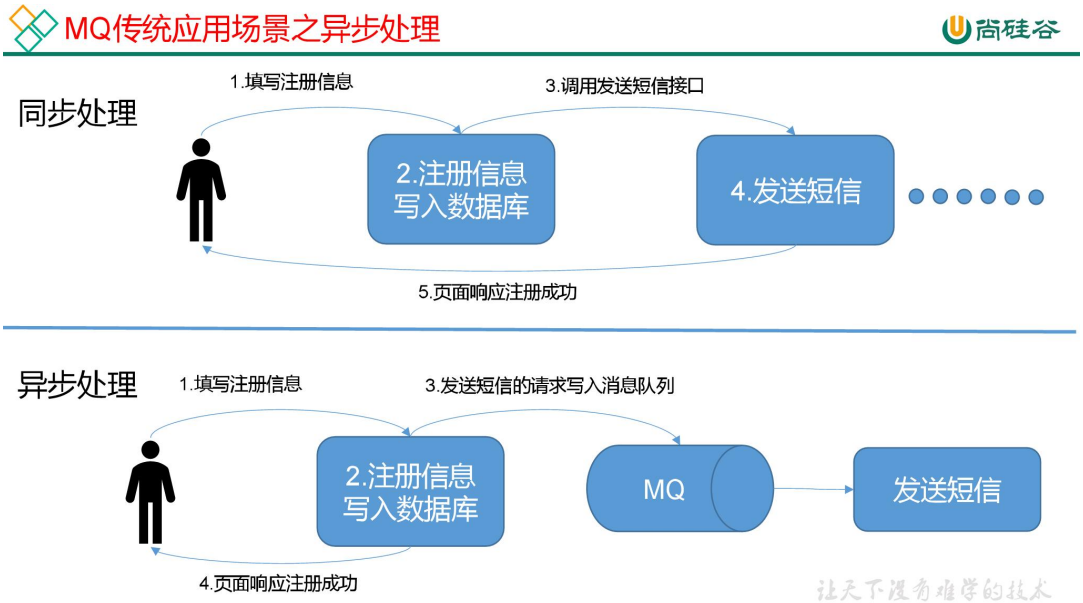
第 1 章 Kafka 概述

1.1 定义

Kafka 是一个分布式的基于发布/订阅模式的消息队列（Message Queue），主要应用于大数据实时处理领域。

1.2 消息队列

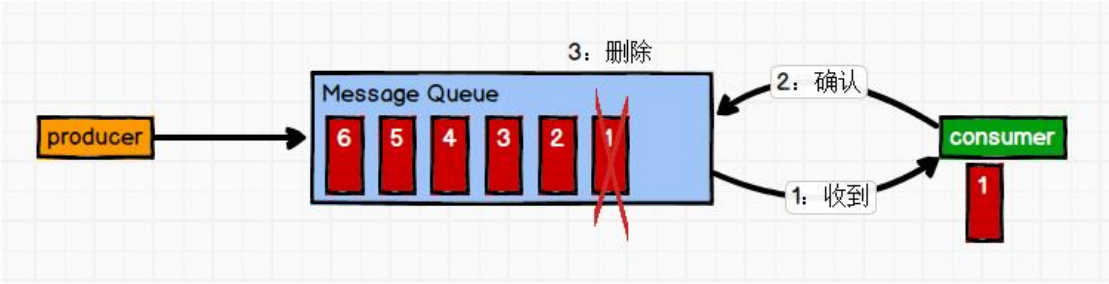
1.2.1 传统消息队列的应用场景



1.2.2 消息队列的两种模式

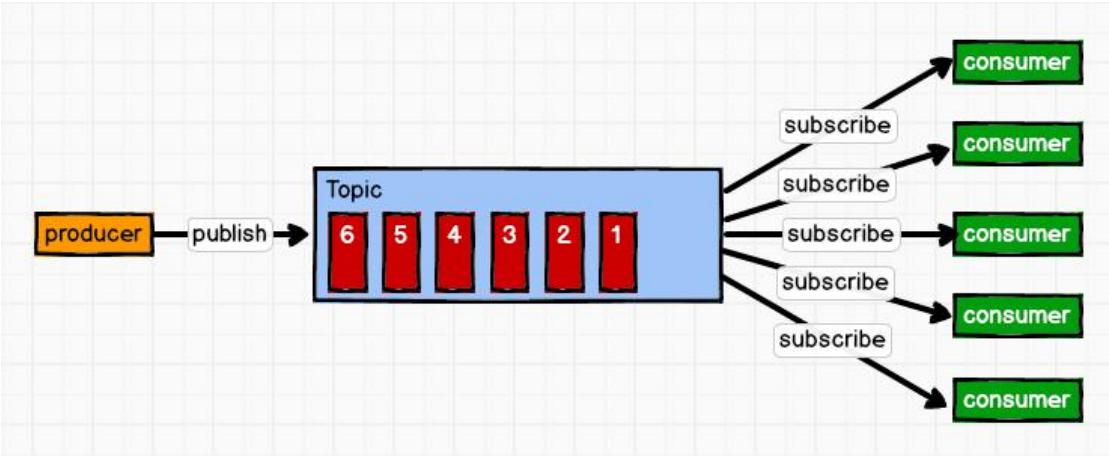
（1）点对点模式（一对一，消费者主动拉取数据，消息收到后消息清除）

消息生产者生产消息发送到 Queue 中，然后消息消费者从 Queue 中取出并且消费消息。消息被消费以后，queue 中不再有存储，所以消息消费者不可能消费到已经被消费的消息。Queue 支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。



（2）发布/订阅模式（一对多，消费者消费数据之后不会清除消息）

消息生产者（发布）将消息发布到 topic 中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到 topic 的消息会被所有订阅者消费。



1.3 Kafka 简介

1.3.1 什么是 Kafka

Kafka 是一个分布式的数据流式传输平台。

在流式计算中，Kafka 一般用来缓存数据，Spark 通过消费 Kafka 的数据进行计算。

1）Apache Kafka 是一个开源消息系统，由 Scala 写成。是由 Apache 软件基金会开发的一个开源消息系统项目。

2）Kafka 最初是由 LinkedIn 公司开发，并于 2011 年初开源。2012 年 10 月从 Apache Incubator 毕业。该项目的目标是为处理实时数据提供一个统一、高通量、低等待的平台。

3）**Kafka 是一个分布式消息队列。**Kafka 对消息保存时根据 Topic 进行归类，发送消息者称为 Producer，消息接受者称为 Consumer，此外 kafka 集群有多个 kafka 实例组成，每个实例(server)称为 broker。

4）无论是 kafka 集群，还是 consumer 都依赖于 **zookeeper** 集群保存一些 meta 信息，来保证系统可用性。

1.3.4 Kafka 的特点

作为一个数据流式传输平台，kafka 有以下三大特点：

- 类似于消息队列和商业的消息系统，kafka 提供对流式数据的发布和订阅
- kafka 提供一种持久的容错的方式存储流式数据
- kafka 拥有良好的性能，可以及时地处理流式数据

基于以上三种特点，kafka 在以下两种应用之间流行：

①需要在多个应用和系统间提供高可靠的实时数据通道

②一些需要实时传输数据及及时计算的应用

此外，kafka 还有以下特点：

- Kafka 主要集群方式运行在一个或多个可跨多个数据中心的服务器上
- Kafka 集群将数据按照类别记录存储，这种类别在 kafka 中称为主题
- 每条记录由一个键，一个值和一个时间戳组成

1.4 Kafka 核心概念

1.4.0 Broker

一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。

1.4.1 Topic

Topic 就是数据主题，kafka 建议根据业务系统将不同的数据存放在不同的 topic 中！Kafka 中的 Topics 总是多订阅者模式，一个 topic 可以拥有一个或者多个消费者来订阅它的数据。一个大的 Topic 可以分布式存储在多个 kafka broker 中！Topic 可以类比为数据库中的库！

1.4.2 Partition

每个 topic 可以有多个分区，通过分区的设计，topic 可以不断进行扩展！即一个 Topic 的多个分区分布式存储在多个 broker!

此外通过分区还可以让一个 topic 被多个 consumer 进行消费！以达到并行处理！分区可以类比为数据库中的表！

kafka 只保证按一个 partition 中的顺序将消息发给 consumer，不保证一个 topic 的整体（多个 partition 间）的顺序。

1.4.3 Offset

数据会按照时间顺序被不断第追加到分区的一个结构化的 commit log 中！每个分区中

存储的记录都是有序的，且顺序不可变！

这个顺序是通过一个称之为 offset 的 id 来唯一标识！因此也可以认为 offset 是有序且不可变的！

在每一个消费者端，会唯一保存的元数据是 offset（偏移量）,即消费在 log 中的位置. 偏移量由消费者所控制。通常在读取记录后，消费者会以线性的方式增加偏移量，但是实际上，由于这个位置由消费者控制，所以消费者可以采用任何顺序来消费记录。例如，一个消费者可以重置到一个旧的偏移量，从而重新处理过去的数据；也可以跳过最近的记录，从"现在"开始消费。

这些细节说明 Kafka 消费者是非常廉价的—消费者的增加和减少，对集群或者其他消费者没有多大的影响。比如，你可以使用命令行工具，对一些 topic 内容执行 tail 操作，并不会影响已存在的消费者消费数据。

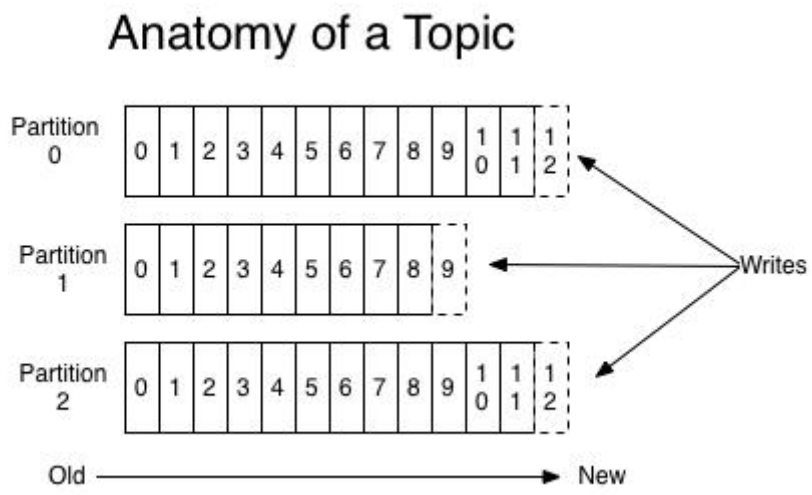


图 1 Topic 拓扑结构

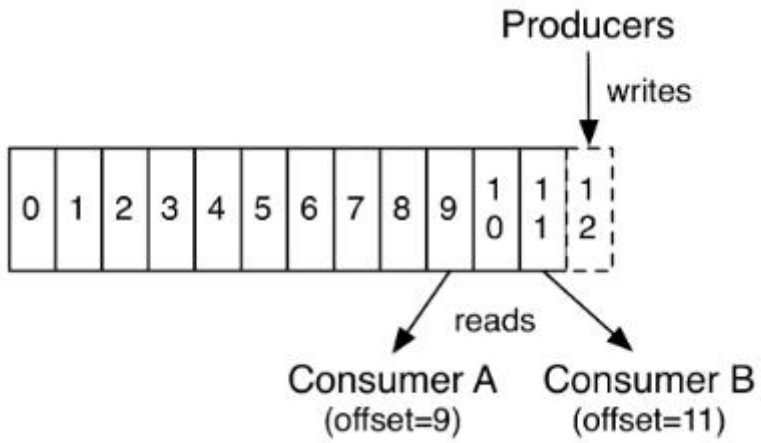


图 2 数据流

1.4.4 持久化

Kafka 集群保留所有发布的记录—无论他们是否已被消费—并通过一个可配置的参数——保留期限来控制。举个例子， 如果保留策略设置为 2 天，一条记录发布后两天内，可以随时被消费，两天过后这条记录会被清除并释放磁盘空间。

Kafka 的性能和数据大小无关，所以长时间存储数据没有什么问题。

1.4.5 副本机制

日志的分区 partition （分布）在 Kafka 集群的服务器上。每个服务器在处理数据和请求时，共享这些分区。每一个分区都会在已配置的服务器上进行备份，确保容错性。

每个分区都有一台 server 作为 “leader”，零台或者多台 server 作为 follwers 。leader server 处理一切对 partition （分区）的读写请求，而 follwers 只需被动的同步 leader 上的数据。当 leader 宕机了，followers 中的一台服务器会自动成为新的 leader。通过这种机制，既可以保证数据有多个副本，也实现了一个高可用的机制！

基于安全考虑，每个分区的 Leader 和 follower 一般会错在在不同的 broker!

1.4.6 Producer

消息生产者，就是向 kafka broker 发消息的客户端。生产者负责将记录分配到 topic 的指定 partition（分区）中

1.4.7 Consumer

消息消费者，向 kafka broker 取消息的客户端。每个消费者都要维护自己读取数据的 offset。低版本 0.9 之前将 offset 保存在 Zookeeper 中，0.9 及之后保存在 Kafka 的 “__consumer_offsets”主题中。

1.4.8 Consumer Group

每个消费者都会使用一个消费组名称来进行标识。同一个组中的不同的消费者实例，可以分布在多个进程或多个机器上！

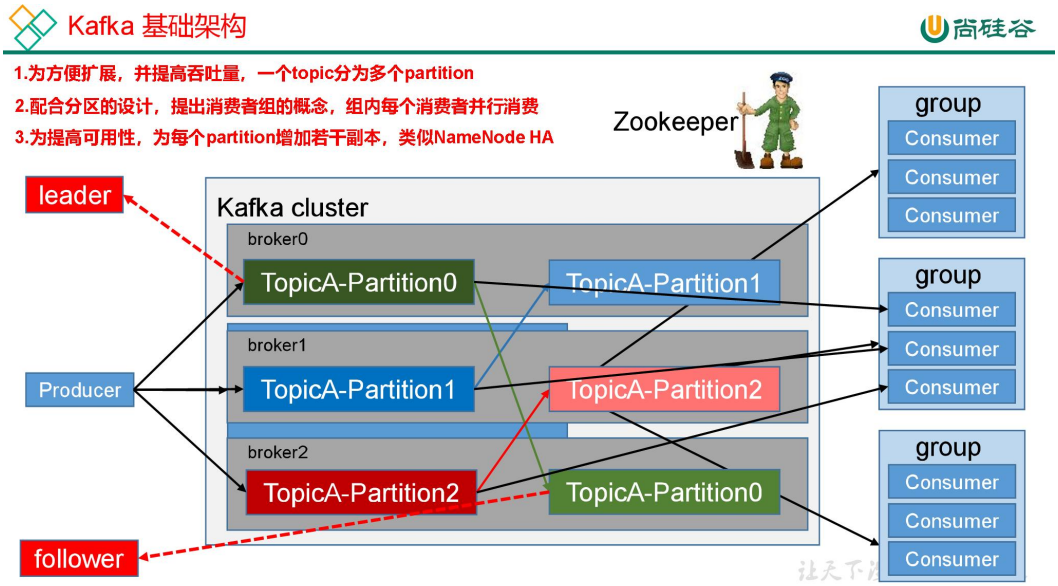
如果所有的消费者实例在同一消费组中，消息记录会负载平衡到每一个消费者实例（单播）。即每个消费者可以同时读取一个 topic 的不同分区！

如果所有的消费者实例在不同的消费组中，每条消息记录会广播到所有的消费者进程(广播)。

如果需要实现广播，只要每个 consumer 有一个独立的组就可以了。要实现单播只要所有的 consumer 在同一个组。

一个 topic 可以有多个 consumer group。topic 的消息会复制（不是真的复制，是概念上的）到所有的 CG，但每个 partion 只会把消息发给该 CG 中的一个 consumer。

1.5 Kafka 基础架构



第 2 章 Kafka 快速入门

2.1 安装部署

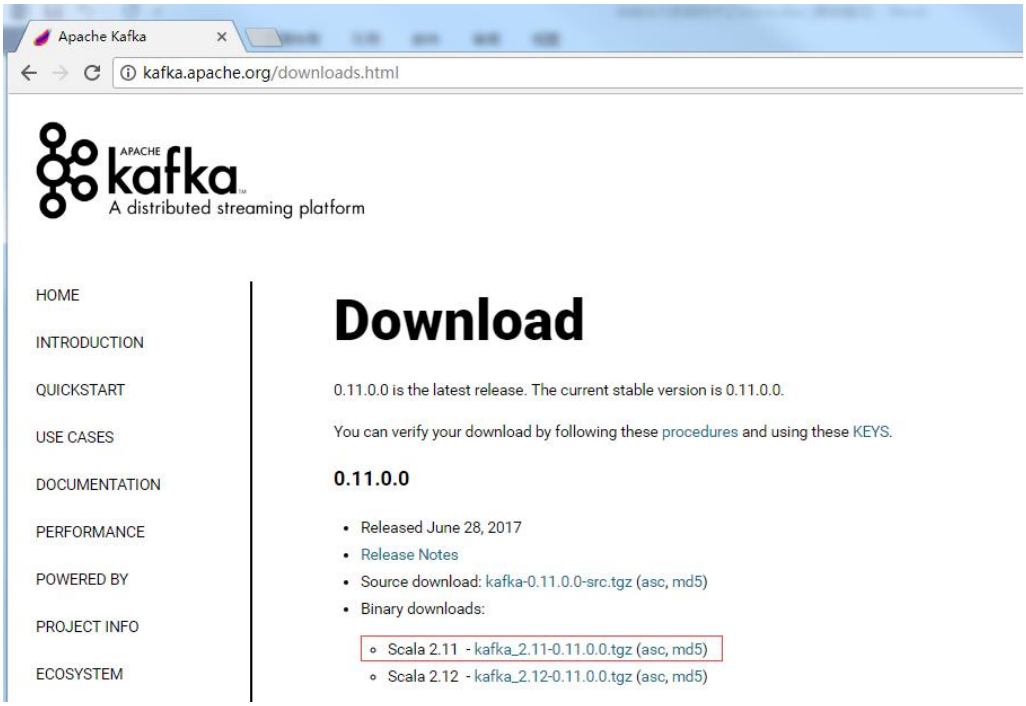
2.1.1 集群规划

hadoop102	hadoop103	hadoop104
zk	zk	zk

kafka kafka kafka

2.1.2 jar 包下载

http://kafka.apache.org/downloads.html



2.1.3 集群部署

1）解压安装包

```
[atguigu@hadoop102 software]$ tar -zxvf kafka_2.11-0.11.0.0.tgz -C /opt/module/
```

2）修改解压后的文件名称

```
[atguigu@hadoop102 module]$ mv kafka_2.11-0.11.0.0/ kafka
```

3）在/opt/module/kafka 目录下创建 logs 文件夹

```
[atguigu@hadoop102 kafka]$ mkdir logs
```

4）修改配置文件

```
[atguigu@hadoop102 kafka]$ cd config/
[atguigu@hadoop102 config]$ vi server.properties
```

输入以下内容：

```
#broker 的全局唯一编号，不能重复
broker.id=0
#删除 topic 功能使能
delete.topic.enable=true
#处理网络请求的线程数量
num.network.threads=3
#用来处理磁盘 IO 的现成数量
num.io.threads=8
#发送套接字的缓冲区大小
socket.send.buffer.bytes=102400
```

```
#接收套接字的缓冲区大小
socket.receive.buffer.bytes=102400
#请求套接字的缓冲区大小
socket.request.max.bytes=104857600
#kafka 运行日志存放的路径
log.dirs=/opt/module/kafka/logs
#topic 在当前 broker 上的分区个数
num.partitions=1
#用来恢复和清理 data 下数据的线程数量
num.recovery.threads.per.data.dir=1
#segment 文件保留的最长时间，超时将被删除
log.retention.hours=168
#配置连接 Zookeeper 集群地址
zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181
```

5）配置环境变量

```
[atguigu@hadoop102 module]$ sudo vi /etc/profile

#KAFKA_HOME
export KAFKA_HOME=/opt/module/kafka
export PATH=$PATH:$KAFKA_HOME/bin

[atguigu@hadoop102 module]$ source /etc/profile
```

6）分发安装包

```
[atguigu@hadoop102 module]$ xsync kafka/
```

注意：分发之后记得配置其他机器的环境变量

7）分别在 hadoop103 和 hadoop104 上修改配置文件/opt/module/kafka/config/server.properties 中的 broker.id=1、broker.id=2

注：broker.id 不得重复

8）启动集群

依次在 hadoop102、hadoop103、hadoop104 节点上启动 kafka

```
[atguigu@hadoop102 kafka]$ bin/kafka-server-start.sh -daemon
 config/server.properties
[atguigu@hadoop103 kafka]$ bin/kafka-server-start.sh -daemon
 config/server.properties
[atguigu@hadoop104 kafka]$ bin/kafka-server-start.sh -daemon
 config/server.properties
```

9）关闭集群

```
[atguigu@hadoop102 kafka]$ bin/kafka-server-stop.sh stop
[atguigu@hadoop103 kafka]$ bin/kafka-server-stop.sh stop
[atguigu@hadoop104 kafka]$ bin/kafka-server-stop.sh stop
```

10）kafka 群起脚本

```
for i in hadoop102 hadoop103 hadoop104
do
echo "========== $i =========="
ssh $i '/opt/module/kafka/bin/kafka-server-start.sh -daemon
 /opt/module/kafka/config/server.properties'
echo $?
```

```
done
```

2.2 Kafka 命令行操作

1）查看当前服务器中的所有 topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper
hadoop102:2181 --list
```

2）创建 topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper
hadoop102:2181 --create --replication-factor 3 --partitions 1 --topic
first
```

选项说明：

--topic 定义 topic 名

--replication-factor 定义副本数

--partitions 定义分区数

3）删除 topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper
hadoop102:2181 --delete --topic first
```

需要 server.properties 中设置 delete.topic.enable=true 否则只是标记删除。

4）发送消息

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh
--broker-list hadoop102:9092 --topic first
>hello world
>atguigu atguigu
```

5）消费消息

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh \
--zookeeper hadoop102:2181 --topic first

[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh \
--bootstrap-server hadoop102:9092 --topic first

[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh \
--bootstrap-server hadoop102:9092 --from-beginning --topic first
```

--from-beginning：会把主题中以往所有的数据都读取出来。

6）查看某个 Topic 的详情

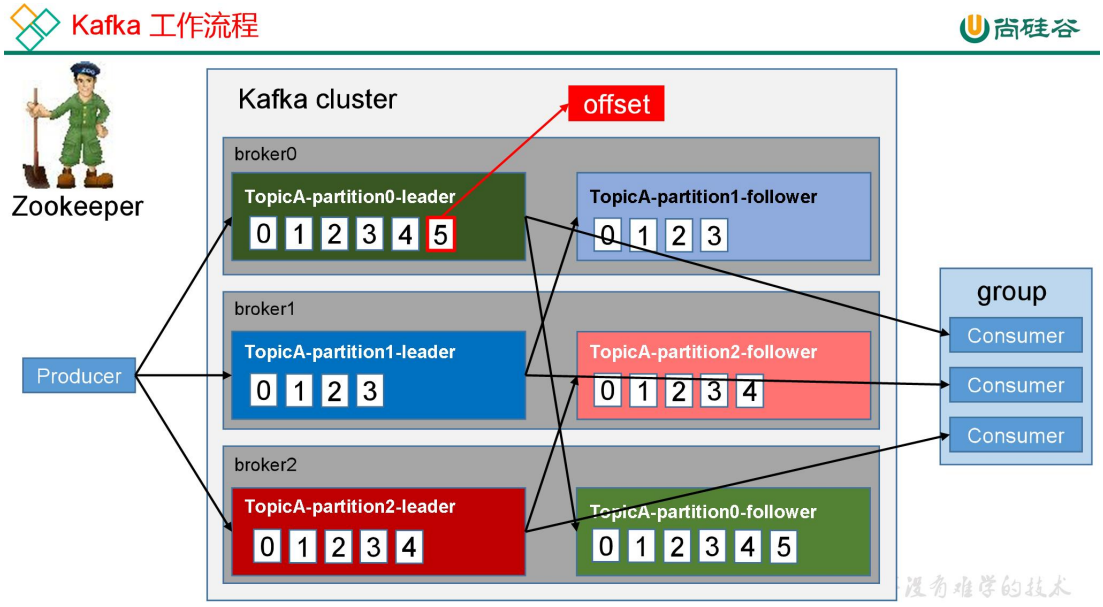
```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper
hadoop102:2181 --describe --topic first
```

7）修改分区数

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper
hadoop102:2181 --alter --topic first --partitions 6
```

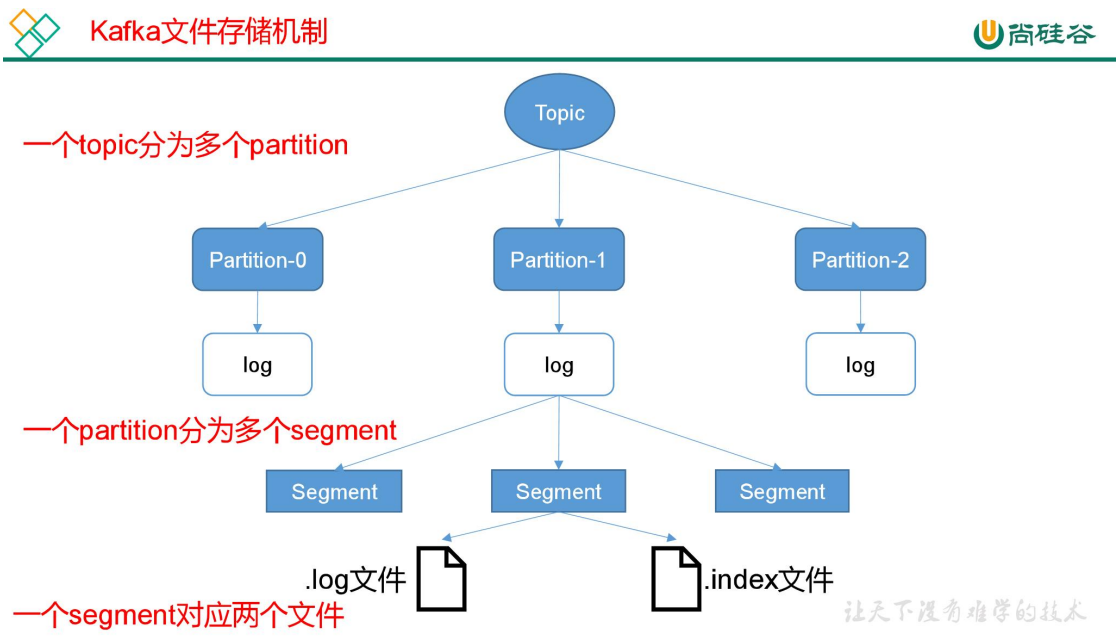
第 3 章 Kafka 架构深入

3.1 Kafka 工作流程及文件存储机制



Kafka 中消息是以 **topic** 进行分类的，生产者生产消息，消费者消费消息，都是面向 topic 的。

topic 是逻辑上的概念，而 partition 是物理上的概念，每个 partition 对应于一个 log 文件，该 log 文件中存储的就是 producer 生产的数据。Producer 生产的数据会被不断追加到该 log 文件末端，且每条数据都有自己的 offset。消费者组中的每个消费者，都会实时记录自己消费到了哪个 offset，以便出错恢复时，从上次的位置继续消费。



由于生产者生产的消息会不断追加到 log 文件末尾，为防止 log 文件过大导致数据定位效率低下，Kafka 采取了**分片**和**索引**机制，将每个 partition 分为多个 segment。

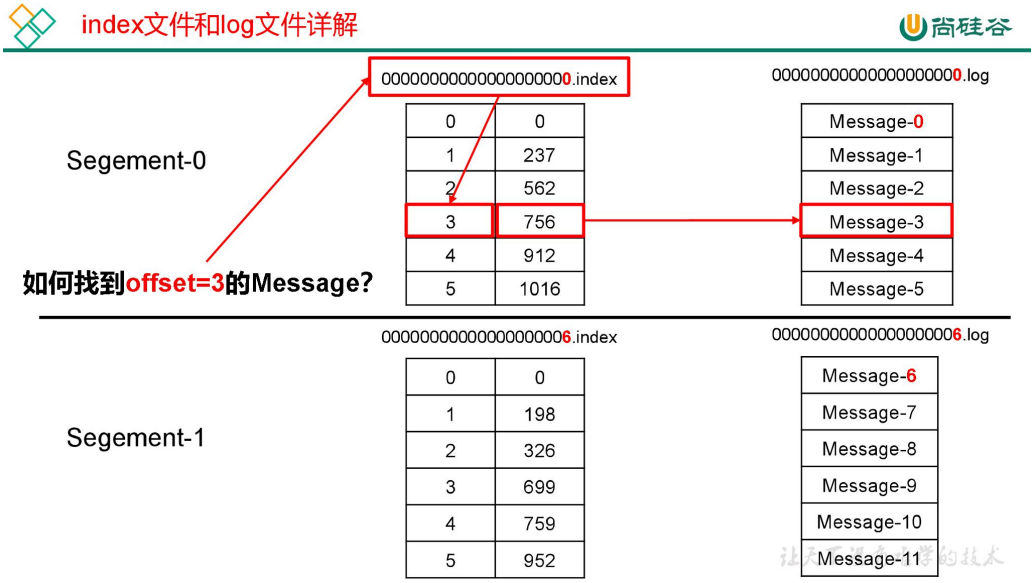
每个 partition(目录)相当于一个巨型文件被平均分配到多个大小相等的 segment(段)数据文件中（每个 segment 文件中消息数量不一定相等），这种特性也方便 old segment 的删除，即方便已被消费的消息的清理，提高磁盘的利用率。每个 partition 只需要支持顺序读写就行，segment 的文件生命周期由服务端配置参数（log.segment.bytes，log.roll.{ms,hours}等若干参数）决定。

每个 segment 对应两个文件——“.index”文件和“.log”文件。分别表示为 segment 索引文件和数据文件（引入索引文件的目的就是便于利用二分查找快速定位 message 位置）。这两个文件的命令规则为：partition 全局的第一个 segment 从 0 开始，后续每个 segment 文件名为上一个 segment 文件最后一条消息的 offset 值，数值大小为 64 位，20 位数字字符长度，没有数字用 0 填充。

这些文件位于一个文件夹下（partition 目录），该文件夹的命名规则为：topic 名称+分区序号。例如，first 这个 topic 有三个分区，则其对应的文件夹为 first-0,first-1,first-2。

```
00000000000000000000.index
00000000000000000000.log
00000000000000170410.index
00000000000000170410.log
00000000000000239430.index
00000000000000239430.log
```

index 和 log 文件以当前 segment 的第一条消息的 offset 命名。下图为 index 文件和 log 文件的结构示意图。



“.index”文件存储大量的索引信息，“.log”文件存储大量的数据，索引文件中的元数

据指向对应数据文件中 message 的物理偏移地址。

3.2 Kafka 生产者

3.2.1 分区策略

1）分区的原因

（1）**方便在集群中扩展**，每个 Partition 可以通过调整以适应它所在的机器，而一个 topic 又可以有多个 Partition 组成，因此整个集群就可以适应任意大小的数据了；

（2）**可以提高并发**，因为可以以 Partition 为单位读写了。

2）分区的原则

我们需要将 producer 发送的数据封装成一个 **ProducerRecord** 对象。

```
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value)
ProducerRecord(@NotNull String topic, String key, String value)
ProducerRecord(@NotNull String topic, String value)
```

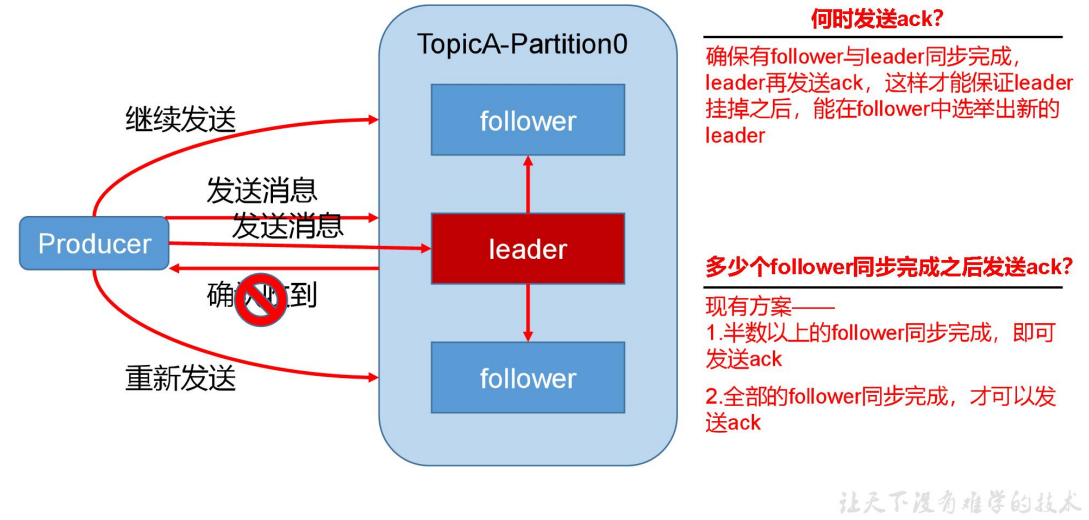
（1）指明 partition 的情况下，直接将指明的值直接作为 partiton 值；

（2）没有指明 partition 值但有 key 的情况下，将 key 的 hash 值与 topic 的 partition 数进行取余得到 partition 值；

（3）既没有 partition 值又没有 key 值的情况下，第一次调用时随机生成一个整数（后面每次调用在这个整数上自增），将这个值与 topic 可用的 partition 总数取余得到 partition 值，也就是常说的 round-robin 算法。

3.2.2 数据可靠性保证

为保证 producer 发送的数据，能可靠的发送到指定的 topic，topic 的每个 partition 收到 producer 发送的数据后，都需要向 producer 发送 ack（acknowledgement 确认收到），如果 producer 收到 ack，就会进行下一轮的发送，否则重新发送数据。



1）副本数据同步策略

方案	优点	缺点
半数以上完成同步，就发送 ack	延迟低	选举新的 leader 时，容忍 n 台节点的故障，需要 2n+1 个副本
全部完成同步，才发送 ack	选举新的 leader 时，容忍 n 台节点的故障，需要 n+1 个副本	延迟高

Kafka 选择了第二种方案，原因如下：

1.同样为了容忍 n 台节点的故障，第一种方案需要 2n+1 个副本，而第二种方案只需要 n+1 个副本，而 Kafka 的每个分区都有大量的数据，第一种方案会造成大量数据的冗余。

2.虽然第二种方案的网络延迟会比较高，但网络延迟对 Kafka 的影响较小。

2）ISR

采用第二种方案之后，设想以下情景：leader 收到数据，所有 follower 都开始同步数据，但有一个 follower，因为某种故障，迟迟不能与 leader 进行同步，那 leader 就要一直等下去，直到它完成同步，才能发送 ack。这个问题怎么解决呢？

Leader 维护了一个动态的 in-sync replica set (ISR)，意为和 leader 保持同步的 follower 集合。当 ISR 中的 follower 完成数据的同步之后，leader 就会给 follower 发送 ack。如果 follower

长时间未向 leader 同步数据，则该 follower 将被踢出 ISR，该时间阈值由 **replica.lag.time.max.ms** 参数设定。Leader 发生故障之后，就会从 ISR 中选举新的 leader。

3）ack 应答机制

对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失，所以没必要等 ISR 中的 follower 全部接收成功。

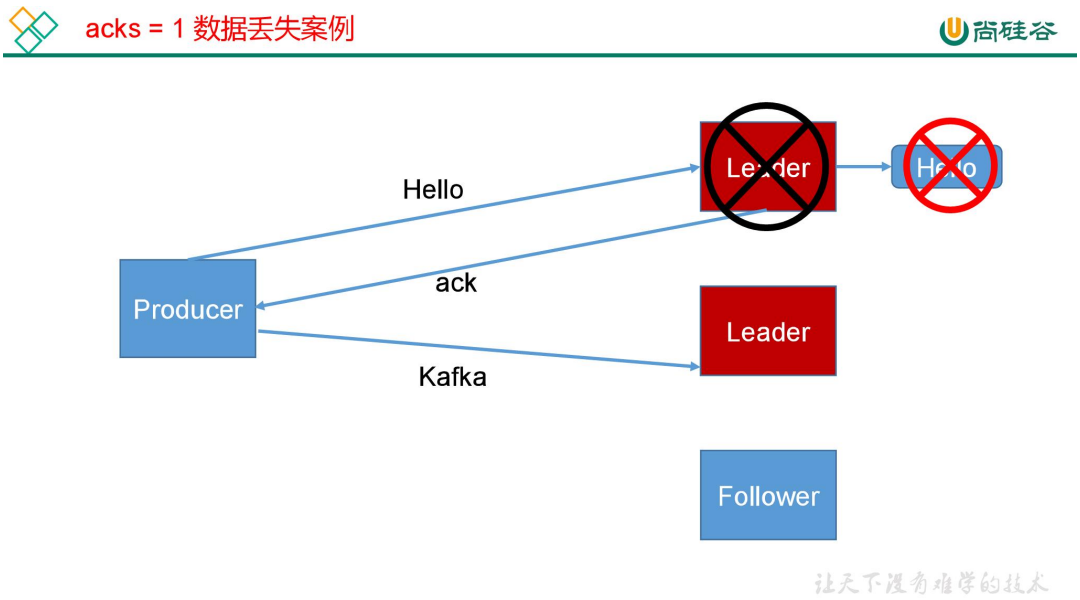
所以 Kafka 为用户提供了三种可靠性级别，用户根据对可靠性和延迟的要求进行权衡，选择以下的配置。

acks 参数配置：

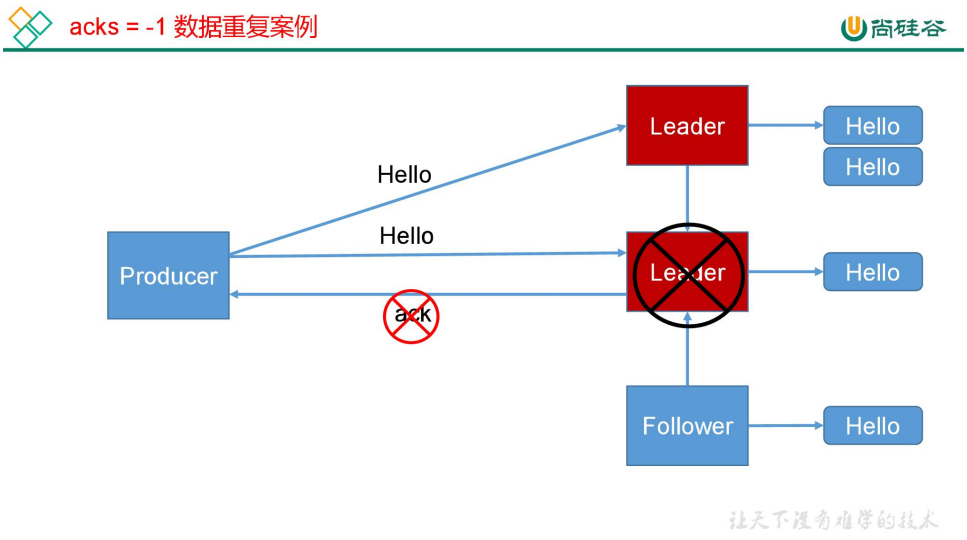
acks：

0：producer 不等待 broker 的 ack，这一操作提供了一个最低的延迟，broker 一接收到还没有写入磁盘就已经返回，当 broker 故障时有可能**丢失数据**；

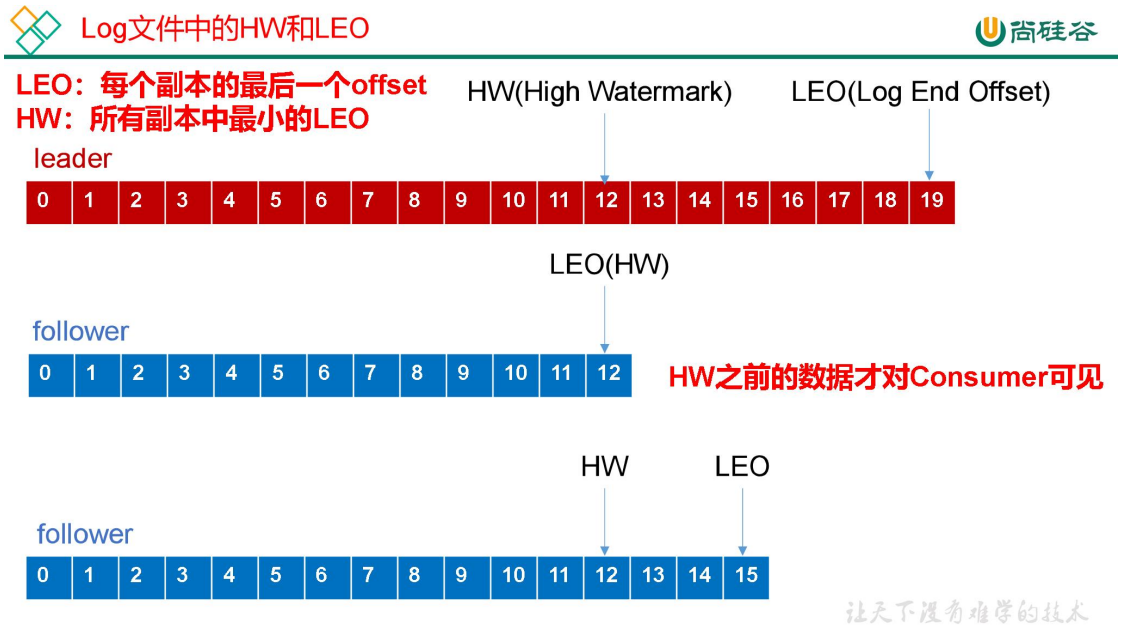
1：producer 等待 broker 的 ack，partition 的 leader 落盘成功后返回 ack，如果在 follower 同步成功之前 leader 故障，那么将会**丢失数据**；



-1（all）：producer 等待 broker 的 ack，partition 的 leader 和 follower 全部落盘成功后才返回 ack。但是如果在 follower 同步完成后，broker 发送 ack 之前，leader 发生故障，那么会造成**数据重复**。



4）故障处理细节



LEO：指的是每个副本最大的 offset；

HW：指的是消费者能见到的最大的 offset，ISR 队列中最小的 LEO。

（1）follower 故障

follower 发生故障后会被临时踢出 ISR，待该 follower 恢复后，follower 会读取本地磁盘记录的上次的 HW，并将 log 文件高于 HW 的部分截取掉，从 HW 开始向 leader 进行同步。等该 **follower 的 LEO 大于等于该 Partition 的 HW**，即 follower 追上 leader 之后，就可以重新加入 ISR 了。

（2）leader 故障

leader 发生故障之后，会从 ISR 中选出一个新的 leader，之后，为保证多个副本之间的数据一致性，其余的 follower 会先将各自的 log 文件高于 HW 的部分截掉，然后从新的 leader 同步数据。

注意：这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。

3.2.3 Exactly Once 语义

对于某些比较重要的消息，我们需要保证 exactly once 语义，即保证每条消息被发送且仅被发送一次。

在 0.11 版本之后，Kafka Producer 引入了幂等性机制（idempotent），配合 acks = -1 时的 at least once 语义，实现了 producer 到 broker 的 exactly once 语义。

idempotent + at least once = exactly once

使用时，只需将 enable.idempotence 属性设置为 true，kafka 自动将 acks 属性设为-1，并将 retries 属性设为 Integer.MAX_VALUE。

3.3 Kafka 消费者

3.3.1 消费方式

consumer 采用 pull（拉）模式从 broker 中读取数据。

push（推）模式很难适应消费速率不同的消费者，因为消息发送速率是由 broker 决定的。它的目标是尽可能以最快速度传递消息，但是这样很容易造成 consumer 来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而 pull 模式则可以根据 consumer 的消费能力以适当的速率消费消息。

pull 模式不足之处是，如果 kafka 没有数据，消费者可能会陷入循环中，一直返回空数据。针对这一点，Kafka 的消费者在消费数据时会传入一个时长参数 timeout，如果当前没有数据可供消费，consumer 会等待一段时间之后再返回，这段时长即为 timeout。

3.3.2 分区分配策略

一个 consumer group 中有多个 consumer，一个 topic 有多个 partition，所以必然会涉及到 partition 的分配问题，即确定那个 partition 由哪个 consumer 来消费。

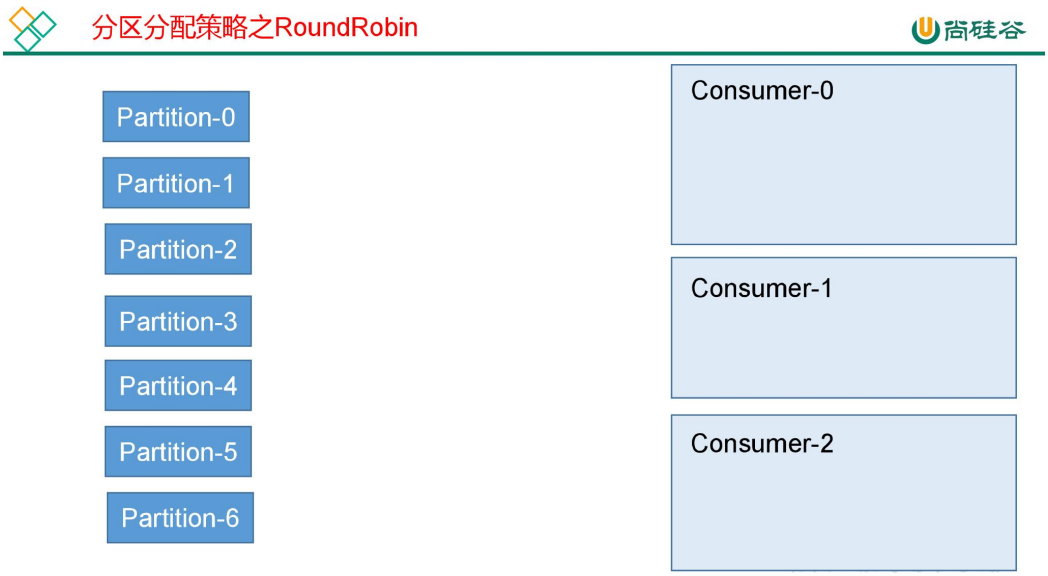
Kafka 有两种分配策略，一是 RoundRobin，一是 Range。

将分区的所有权从一个消费者移到另一个消费者称为重新平衡（rebalance）。当以下事

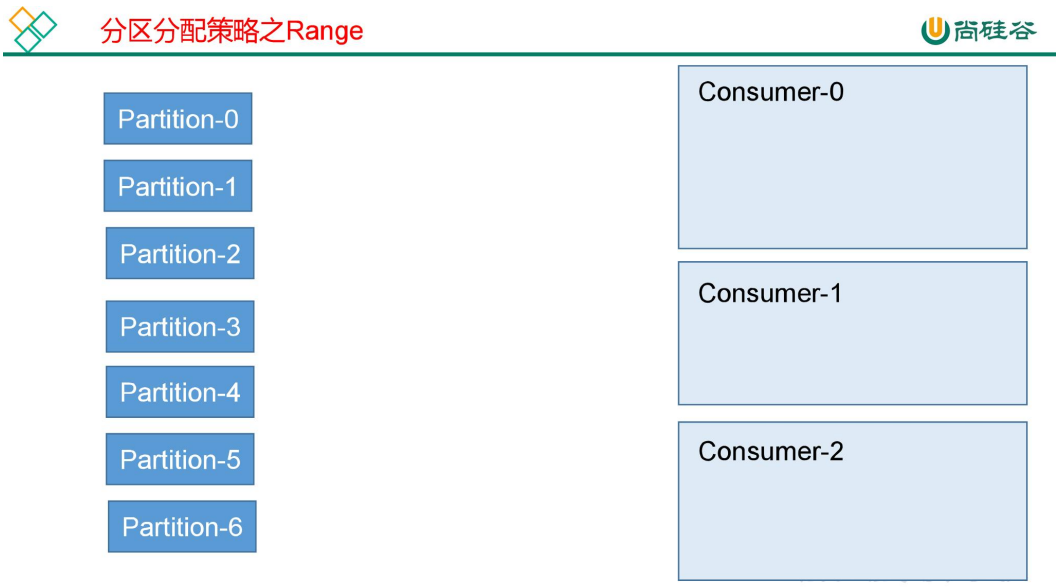
件发生时，Kafka 将会进行一次分区分配：

- 同一个 Consumer Group 内新增消费者
- 消费者离开当前所属的 Consumer Group，包括 shuts down 或 crashes
- 订阅的主题新增分区

1）RoundRobin

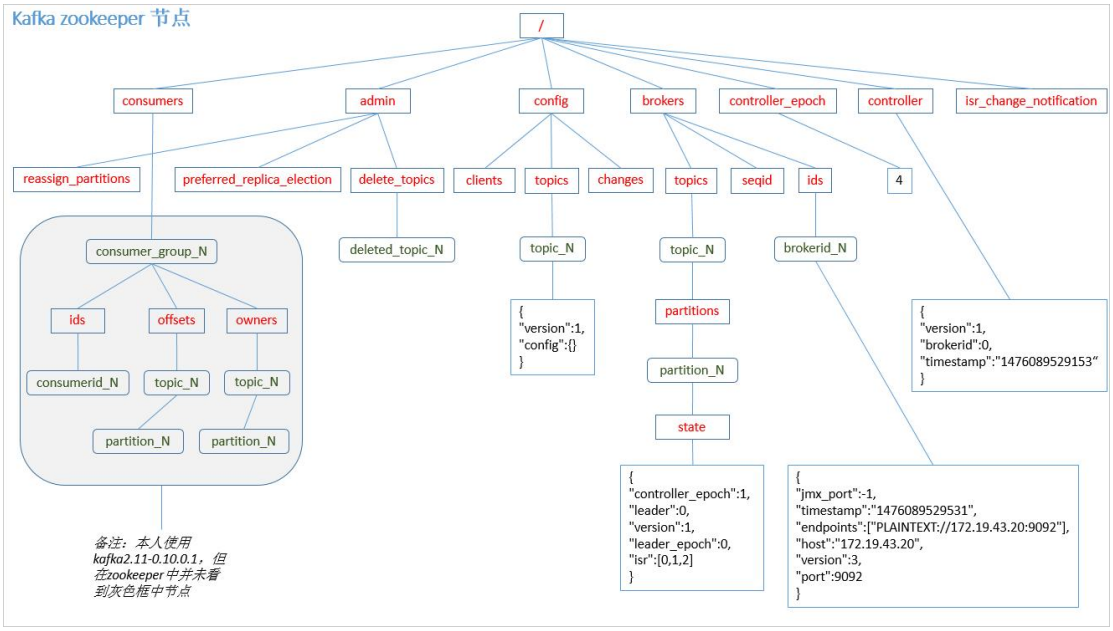


2）Range



3.3.3 offset 的维护

由于 consumer 在消费过程中可能会出现断电宕机等故障，consumer 恢复后，需要从故障前的位置的继续消费，所以 consumer 需要实时记录自己消费到了哪个 offset，以便故障恢复后继续消费。



Kafka 0.9 版本之前，consumer 默认将 offset 保存在 Zookeeper 中，从 0.9 版本开始，consumer 默认将 offset 保存在 Kafka 一个内置的 topic 中，该 topic 为**__consumer_offsets**。

1）修改配置文件 consumer.properties

```
exclude.internal.topics=false
```

2）读取 offset

0.11.0.0 之前版本:

```
bin/kafka-console-consumer.sh --topic __consumer_offsets
--zookeeper hadoop102:2181 --formatter
"kafka.coordinator.GroupMetadataManager\$OffsetsMessageFormatter"
--consumer.config config/consumer.properties --from-beginning
```

0.11.0.0 之后版本(含):

```
bin/kafka-console-consumer.sh --topic __consumer_offsets
--zookeeper hadoop102:2181 --formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageForm
atter" --consumer.config config/consumer.properties
--from-beginning
```

3.3.4 消费者组案例

1）需求：测试同一个消费者组中的消费者，同一时刻只能有一个消费者消费。

2）案例实操

（1）在 hadoop102、hadoop103 上修改/opt/module/kafka/config/consumer.properties 配置文件中的 group.id 属性为任意组名。

```
[atguigu@hadoop103 config]$ vi consumer.properties
group.id=atguigu
```

（2）在 hadoop102、hadoop103 上分别启动消费者

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh \
--zookeeper hadoop102:2181 --topic first --consumer.config
config/consumer.properties
```

```
[atguigu@hadoop103 kafka]$ bin/kafka-console-consumer.sh
--bootstrap-server hadoop102:9092 --topic first --consumer.config
config/consumer.properties
```

（3）在 hadoop104 上启动生产者

```
[atguigu@hadoop104 kafka]$ bin/kafka-console-producer.sh \
--broker-list hadoop102:9092 --topic first
>hello world
```

（4）查看 hadoop102 和 hadoop103 的接收者。

同一时刻只有一个消费者接收到消息。

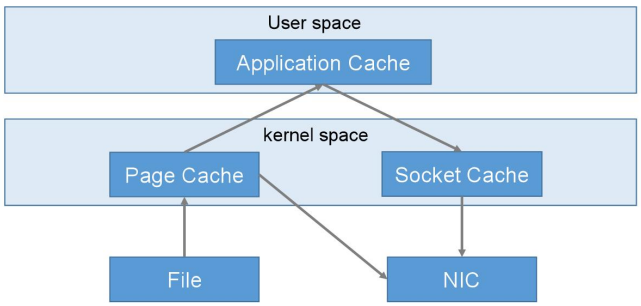
3.4 Kafka 高效读写数据

1）顺序写磁盘

Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端，为顺序写。官网有数据表明，同样的磁盘，顺序写能到 600M/s，而随机写只有 100K/s。这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

2）零复制技术

零拷贝

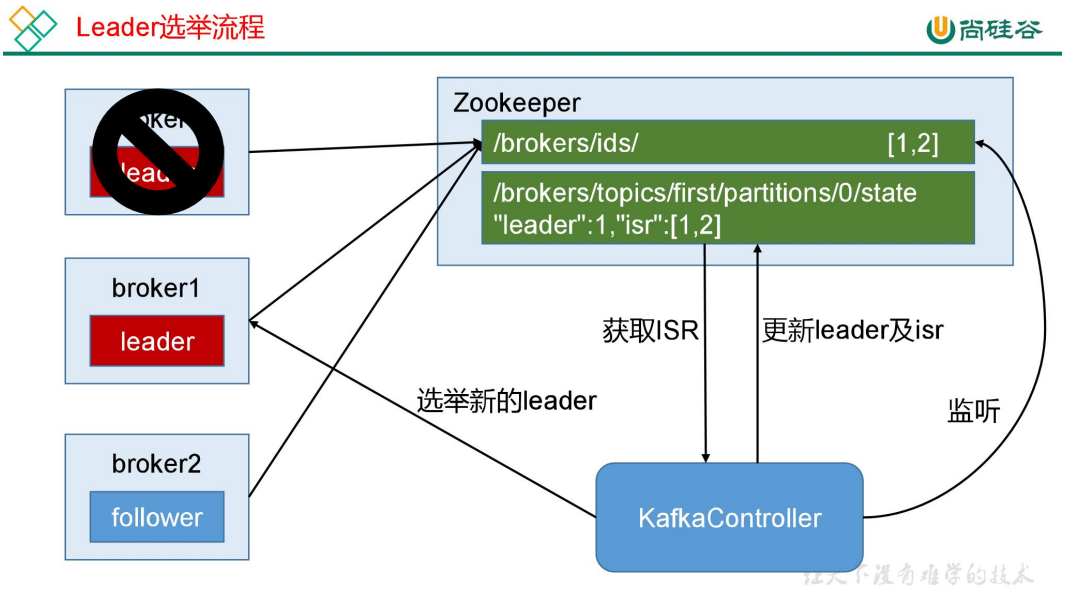


3.5 Zookeeper 在 Kafka 中的作用

Kafka 集群中有一个 broker 会被选举为 Controller，负责管理集群 broker 的上下线，所有 topic 的分区副本分配和 leader 选举等工作。

Controller 的管理工作都是依赖于 Zookeeper 的。

以下为 partition 的 leader 选举过程：

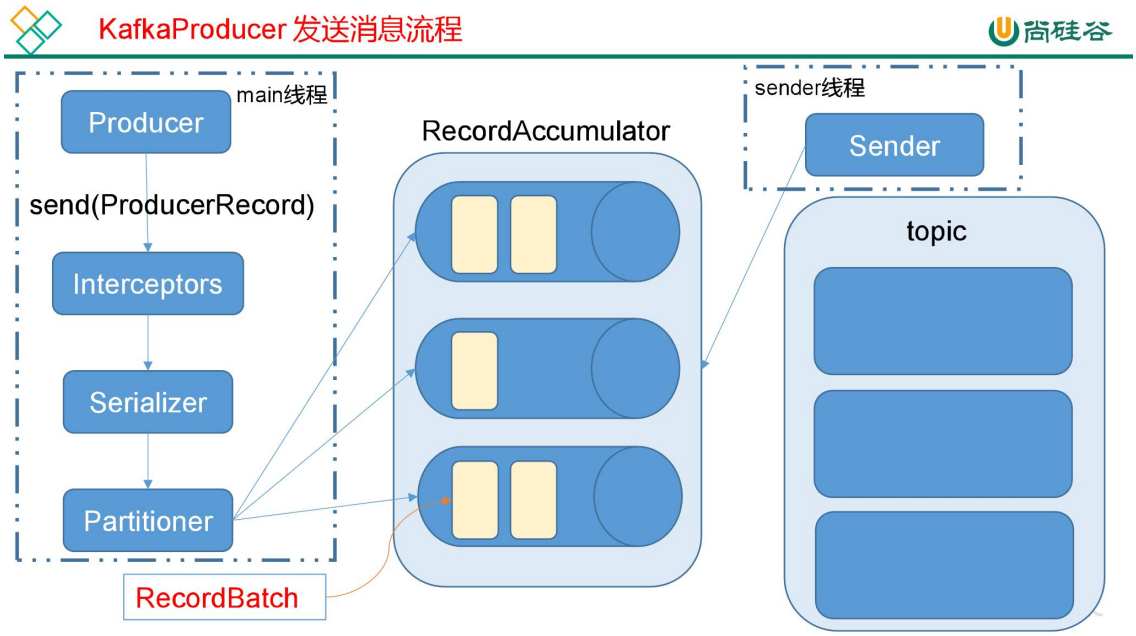


第 4 章 Kafka API

4.1 Producer API

4.1.1 消息发送流程

Kafka 的 Producer 发送消息采用的是**异步发送**的方式。在消息发送的过程中，涉及到了**两个线程——main 线程和 Sender 线程**，以及**一个线程共享变量——RecordAccumulator**。main 线程将消息发送给 RecordAccumulator，Sender 线程不断从 RecordAccumulator 中拉取消息发送到 Kafka broker。



相关参数：

batch.size：只有数据积累到 batch.size 之后，sender 才会发送数据。

linger.ms：如果数据迟迟未达到 batch.size，sender 等待 linger.time 之后就会发送数据。

4.1.2 异步发送 API

1）导入依赖

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

2）编写代码

需要用到的类：

KafkaProducer：需要创建一个生产者对象，用来发送数据

ProducerConfig：获取所需的一系列配置参数

ProducerRecord：每条数据都要封装成一个 ProducerRecord 对象

1.不带回调函数的 API

```
package com.atguigu.kafka;

import org.apache.kafka.clients.producer.*;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class CustomProducer {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        Properties props = new Properties();

        //kafka 集群，broker-list
        props.put("bootstrap.servers", "hadoop102:9092");

        props.put("acks", "all");

        //重试次数
        props.put("retries", 1);

        //批次大小
        props.put("batch.size", 16384);

        //等待时间
        props.put("linger.ms", 1);

        //RecordAccumulator 缓冲区大小
        props.put("buffer.memory", 33554432);
```

```
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String,       String>       producer       =       new
KafkaProducer<>(props);

        for (int i = 0; i < 100; i++) {
            producer.send(new ProducerRecord<String, String>("first",
Integer.toString(i), Integer.toString(i)));
        }

        producer.close();
    }
}
```

2.带回调函数的 API

回调函数会在 producer 收到 ack 时调用，为异步调用，该方法有两个参数，分别是 RecordMetadata 和 Exception，如果 Exception 为 null，说明消息发送成功，如果 Exception 不为 null，说明消息发送失败。

注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试。

```
package com.atguigu.kafka;

import org.apache.kafka.clients.producer.*;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class CustomProducer {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {

        Properties props = new Properties();

        props.put("bootstrap.servers", "hadoop102:9092");//kafka 集群，
broker-list

        props.put("acks", "all");

        props.put("retries", 1);//重试次数

        props.put("batch.size", 16384);//批次大小

        props.put("linger.ms", 1);//等待时间

        props.put("buffer.memory", 33554432);//RecordAccumulator 缓冲
区大小

        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
```

```
"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String,       String>       producer       =       new
KafkaProducer<>(props);

        for (int i = 0; i < 100; i++) {
            producer.send(new ProducerRecord<String, String>("first",
Integer.toString(i), Integer.toString(i)), new Callback() {

                //回调函数，该方法会在 Producer 收到 ack 时调用，为异步调用
                @Override
                public  void  onCompletion(RecordMetadata  metadata,
Exception exception) {
                    if (exception == null) {
                        System.out.println("success->"                +
metadata.offset());
                    } else {
                        exception.printStackTrace();
                    }
                }
            });
        }
        producer.close();
    }
}
```

4.1.3 同步发送 API

同步发送的意思就是，一条消息发送之后，会阻塞当前线程，直至返回 ack。

由于 send 方法返回的是一个 Future 对象，根据 Futrue 对象的特点，我们也可以实现同步发送的效果，只需在调用 Future 对象的 get 方发即可。

```
package com.atguigu.kafka;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class CustomProducer {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {

        Properties props = new Properties();

        props.put("bootstrap.servers", "hadoop102:9092");//kafka 集群，
broker-list

        props.put("acks", "all");

        props.put("retries", 1);//重试次数

        props.put("batch.size", 16384);//批次大小
```

```
        props.put("linger.ms", 1);//等待时间

        props.put("buffer.memory", 33554432);//RecordAccumulator 缓冲
区大小

        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new
KafkaProducer<>(props);
        for (int i = 0; i < 100; i++) {
            producer.send(new ProducerRecord<String, String>("first",
Integer.toString(i), Integer.toString(i))).get();
        }
        producer.close();
    }
}
```

4.2 Consumer API

Consumer 消费数据时的可靠性是很容易保证的，因为数据在 Kafka 中是持久化的，故不用担心数据丢失问题。

由于 consumer 在消费过程中可能会出现断电宕机等故障，consumer 恢复后，需要从故障前的位置的继续消费，所以 consumer 需要实时记录自己消费到了哪个 offset，以便故障恢复后继续消费。

所以 offset 的维护是 Consumer 消费数据是必须考虑的问题。

4.2.1 自动提交 offset

1）导入依赖

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

2）编写代码

需要用到的类：

KafkaConsumer：需要创建一个消费者对象，用来消费数据

ConsumerConfig：获取所需的一系列配置参数

ConsuemrRecord：每条数据都要封装成一个 ConsumerRecord 对象

为了使我们能够专注于自己的业务逻辑，Kafka 提供了自动提交 offset 的功能。

自动提交 offset 的相关参数：

enable.auto.commit：是否开启自动提交 offset 功能

auto.commit.interval.ms：自动提交 offset 的时间间隔

以下为自动提交 offset 的代码：

```
package com.atguigu.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Arrays;
import java.util.Properties;

public class CustomConsumer {

    public static void main(String[] args) {

        Properties props = new Properties();

        props.put("bootstrap.servers", "hadoop102:9092");

        props.put("group.id", "test");

        props.put("enable.auto.commit", "true");

        props.put("auto.commit.interval.ms", "1000");

        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new
KafkaConsumer<>(props);

        consumer.subscribe(Arrays.asList("first"));

        while (true) {

            ConsumerRecords<String, String> records =
consumer.poll(100);

            for (ConsumerRecord<String, String> record : records)

                System.out.printf("offset = %d, key = %s, value = %s%n",
record.offset(), record.key(), record.value());
        }
    }
}
```

4.2.2 手动提交 offset

虽然自动提交 offset 十分简介便利，但由于其是基于时间提交的，开发人员难以把握 offset 提交的时机。因此 Kafka 还提供了手动提交 offset 的 API。

手动提交 offset 的方法有两种：分别是 commitSync（同步提交）和 commitAsync（异步提交）。两者的相同点是，都会将**本次 poll 的一批数据最高的偏移量提交**；不同点是，commitSync 阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；而 commitAsync 则没有失败重试机制，故有可能提交失败。

1）同步提交 offset

由于同步提交 offset 有失败重试机制，故更加可靠，以下为同步提交 offset 的示例。

```
package com.atguigu.kafka.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Arrays;
import java.util.Properties;

public class CustomComsumer {

    public static void main(String[] args) {

        Properties props = new Properties();

        //Kafka 集群
        props.put("bootstrap.servers", "hadoop102:9092");

        //消费者组，只要 group.id 相同，就属于同一个消费者组
        props.put("group.id", "test");

        props.put("enable.auto.commit", "false");//关闭自动提交 offset

        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new
KafkaConsumer<>(props);

        consumer.subscribe(Arrays.asList("first"));//消费者订阅主题

        while (true) {

            //消费者拉取数据
            ConsumerRecords<String, String> records =
consumer.poll(100);

            for (ConsumerRecord<String, String> record : records) {

                System.out.printf("offset = %d, key = %s, value = %s%n",
record.offset(), record.key(), record.value());

            }
```

```
            //同步提交，当前线程会阻塞直到 offset 提交成功
            consumer.commitSync();
        }
    }
}
```

2）异步提交 offset

虽然同步提交 offset 更可靠一些，但是由于其会阻塞当前线程，直到提交成功。因此吞吐量会收到很大的影响。因此更多的情况下，会选用异步提交 offset 的方式。

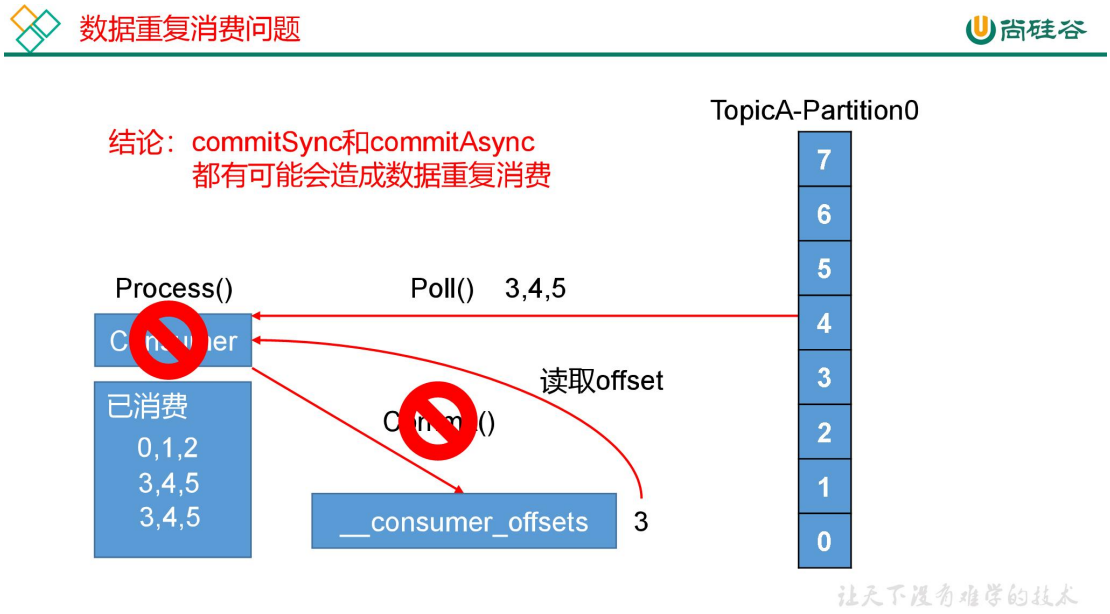
以下为异步提交 offset 的示例：

```
package com.atguigu.kafka.consumer;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;

import java.util.Arrays;
import java.util.Map;
import java.util.Properties;


public class CustomConsumer {

    public static void main(String[] args) {

        Properties props = new Properties();

        //Kafka 集群
        props.put("bootstrap.servers", "hadoop102:9092");

        //消费者组，只要 group.id 相同，就属于同一个消费者组
        props.put("group.id", "test");

        //关闭自动提交 offset
        props.put("enable.auto.commit", "false");

        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new
KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("first"));//消费者订阅主题

        while (true) {
            ConsumerRecords<String, String> records =
consumer.poll(100);//消费者拉取数据
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("offset = %d, key = %s, value = %s%n",
record.offset(), record.key(), record.value());
            }

            //异步提交
```

```
            consumer.commitAsync(new OffsetCommitCallback() {
                @Override
                public      void      onComplete(Map<TopicPartition,
OffsetAndMetadata> offsets, Exception exception) {
                    if (exception != null) {
                        System.err.println("Commit   failed   for"   +
offsets);
                    }
                }
            });
        }
    }
}
```

3） 数据漏消费和重复消费分析

无论是同步提交还是异步提交 offset，都有可能会造成数据的漏消费或者重复消费。先提交 offset 后消费，有可能造成数据的漏消费；而先消费后提交 offset，有可能会造成数据的重复消费。



4.2.3 自定义存储 offset

Kafka 0.9 版本之前，offset 存储在 zookeeper，0.9 版本之后，默认将 offset 存储在 Kafka 的一个内置的 topic 中。除此之外，Kafka 还可以选择自定义存储 offset。

offset 的维护是相当繁琐的，因为需要考虑到消费者的 Rebalace。

当有新的消费者加入消费者组、已有的消费者推出消费者组或者所订阅的主题的分区发生变化，就会触发到分区的重新分配，重新分配的过程叫做 Rebalance。

消费者发生 Rebalance 之后，每个消费者消费的分区就会发生变化。因此消费者要首先获取到自己被重新分配到的分区，并且定位到每个分区最近提交的 offset 位置继续消费。

要实现自定义存储 offset，需要借助 **ConsumerRebalanceListener**，以下为示例代码，其中提交和获取 offset 的方法，需要根据所选的 offset 存储系统自行实现。

```
package com.atguigu.kafka.consumer;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;

import java.util.*;

public class CustomConsumer {

   private static Map<TopicPartition, Long> currentOffset = new
HashMap<>();

   public static void main(String[] args) {

       //创建配置信息
       Properties props = new Properties();

       //Kafka 集群
       props.put("bootstrap.servers", "hadoop102:9092");

       //消费者组，只要 group.id 相同，就属于同一个消费者组
       props.put("group.id", "test");

       //关闭自动提交 offset
       props.put("enable.auto.commit", "false");

       //Key 和 Value 的反序列化类
       props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
       props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

       //创建一个消费者
       KafkaConsumer<String,     String>     consumer     =     new
KafkaConsumer<>(props);

       //消费者订阅主题
       consumer.subscribe(Arrays.asList("first"),                new
ConsumerRebalanceListener() {

           //该方法会在 Rebalance 之前调用
           @Override
           public                                                 void
onPartitionsRevoked(Collection<TopicPartition> partitions) {
               commitOffset(currentOffset);
           }

           //该方法会在 Rebalance 之后调用
           @Override
           public                                                 void
onPartitionsAssigned(Collection<TopicPartition> partitions) {
               currentOffset.clear();
               for (TopicPartition partition : partitions) {
```

```
                    consumer.seek(partition, getOffset(partition));//
定位到最近提交的 offset 位置继续消费
                }
            }
        });

        while (true) {
            ConsumerRecords<String, String> records =
consumer.poll(100);//消费者拉取数据
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("offset = %d, key = %s, value = %s%n",
record.offset(), record.key(), record.value());
                currentOffset.put(new TopicPartition(record.topic(),
record.partition()), record.offset());
            }
            commitOffset(currentOffset);//异步提交
        }
    }

    //获取某分区的最新 offset
    private static long getOffset(TopicPartition partition) {
        return 0;
    }

    //提交该消费者所有分区的 offset
    private static void commitOffset(Map<TopicPartition, Long>
currentOffset) {

    }
}
```

4.3 自定义 Interceptor

4.3.1 拦截器原理

Producer 拦截器(interceptor)是在 Kafka 0.10 版本被引入的，主要用于实现 clients 端的定制化控制逻辑。

对于 producer 而言，interceptor 使得用户在消息发送前以及 producer 回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer 允许用户指定多个 interceptor 按序作用于同一条消息从而形成一个拦截链(interceptor chain)。Intercetpor 的实现接口是 org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括：

（1）configure(configs)

获取配置信息和初始化数据时调用。

（2）onSend(ProducerRecord)：

该方法封装进 KafkaProducer.send 方法中，即它运行在用户主线程中。Producer 确保在消息被序列化以及计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好

保证不要修改消息所属的 topic 和分区，否则会影响目标分区的计算。

（3）onAcknowledgement(RecordMetadata, Exception)：

该方法会在消息从 RecordAccumulator 成功发送到 Kafka Broker 之后，或者在发送过程中失败时调用。并且通常都是在 producer 回调逻辑触发之前。onAcknowledgement 运行在 producer 的 IO 线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢 producer 的消息发送效率。

（4）close：

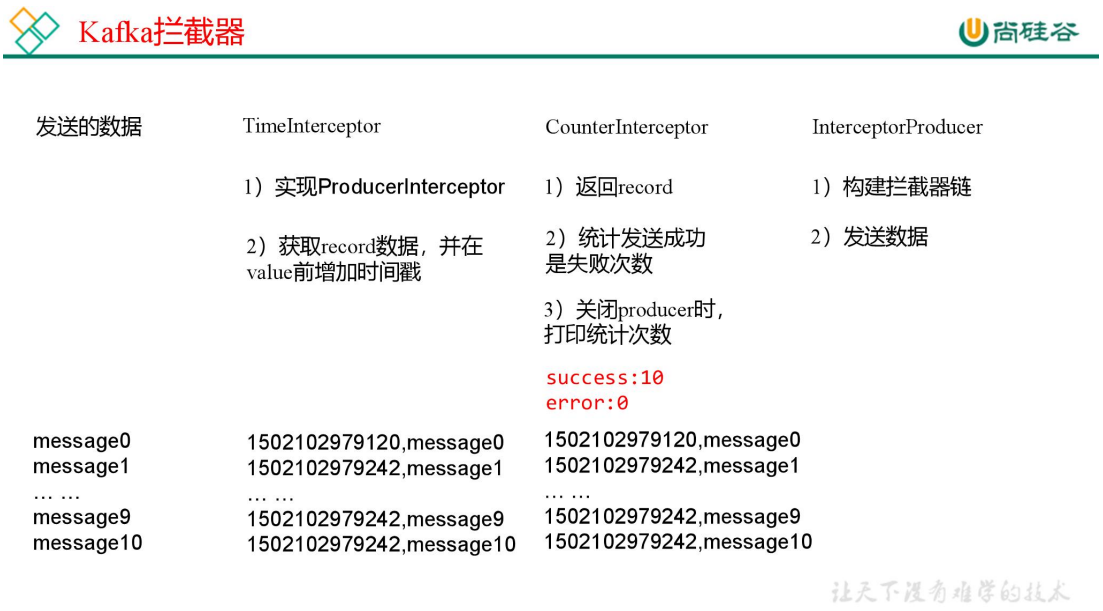
关闭 interceptor，主要用于执行一些资源清理工作

如前所述，interceptor 可能被运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外倘若指定了多个 interceptor，则 producer 将按照指定顺序调用它们，并仅仅是捕获每个 interceptor 可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中要特别留意。

4.3.2 拦截器案例

1）需求：

实现一个简单的双 interceptor 组成的拦截链。第一个 interceptor 会在消息发送前将时间戳信息加到消息 value 的最前部；第二个 interceptor 会在消息发送后更新成功发送消息数或失败发送消息数。



2）案例实操

（1）增加时间戳拦截器

```
package com.atguigu.kafka.interceptor;
```

```
import java.util.Map;
import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

public class TimeInterceptor implements
ProducerInterceptor<String, String> {

    @Override
    public void configure(Map<String, ?> configs) {

    }

    @Override
    public ProducerRecord<String, String>
onSend(ProducerRecord<String, String> record) {

        // 创建一个新的 record，把时间戳写入消息体的最前部
        return new ProducerRecord(record.topic(),
record.partition(), record.timestamp(), record.key(),
                System.currentTimeMillis() + "," +
record.value().toString());
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata,
Exception exception) {

    }

    @Override
    public void close() {

    }
}
```

（2）统计发送消息成功和发送失败消息数，并在 producer 关闭时打印这两个计数器

```
package com.atguigu.kafka.interceptor;
import java.util.Map;
import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

public class CounterInterceptor implements
ProducerInterceptor<String, String>{
    private int errorCounter = 0;
    private int successCounter = 0;

    @Override
    public void configure(Map<String, ?> configs) {

    }

    @Override
    public ProducerRecord<String, String>
onSend(ProducerRecord<String, String> record) {
        return record;
    }
```

```
    @Override
    public void onAcknowledgement(RecordMetadata metadata,
Exception exception) {
        // 统计成功和失败的次数
        if (exception == null) {
            successCounter++;
        } else {
            errorCounter++;
        }
    }

    @Override
    public void close() {
        // 保存结果
        System.out.println("Successful sent: " + successCounter);
        System.out.println("Failed sent: " + errorCounter);
    }
}
```

（3）producer 主程序

```
package com.atguigu.kafka.interceptor;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;

public class InterceptorProducer {

    public static void main(String[] args) throws Exception {
        // 1 设置配置信息
        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop102:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        // 2 构建拦截链
        List<String> interceptors = new ArrayList<>();

    interceptors.add("com.atguigu.kafka.interceptor.TimeInterce
ptor");
    interceptors.add("com.atguigu.kafka.interceptor.CounterInte
rceptor");
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
interceptors);

        String topic = "first";
```

```
        Producer<String,      String>      producer      =      new
KafkaProducer<>(props);

        // 3 发送消息
        for (int i = 0; i < 10; i++) {

                ProducerRecord<String,  String>  record  =  new
ProducerRecord<>(topic, "message" + i);
            producer.send(record);
        }

        // 4 一定要关闭 producer，这样才会调用 interceptor 的 close 方法
        producer.close();
    }
}
```

3）测试

（1）在 kafka 上启动消费者，然后运行客户端 java 程序。

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh \
--bootstrap-server hadoop102:9092 --from-beginning --topic first

1501904047034,message0
1501904047225,message1
1501904047230,message2
1501904047234,message3
1501904047236,message4
1501904047240,message5
1501904047243,message6
1501904047246,message7
1501904047249,message8
1501904047252,message9
```

第 5 章 Kafka 监控

5.1 Kafka Manager

1.上传压缩包 kafka-manager-1.3.3.15.zip 到集群

2.解压 kafka-manager-1.3.3.15.zip

```
[atguigu@hadoop102 module]$ unzip kafka-manager-1.3.3.15.zip
```

3.修改配置文件

conf/application.conf

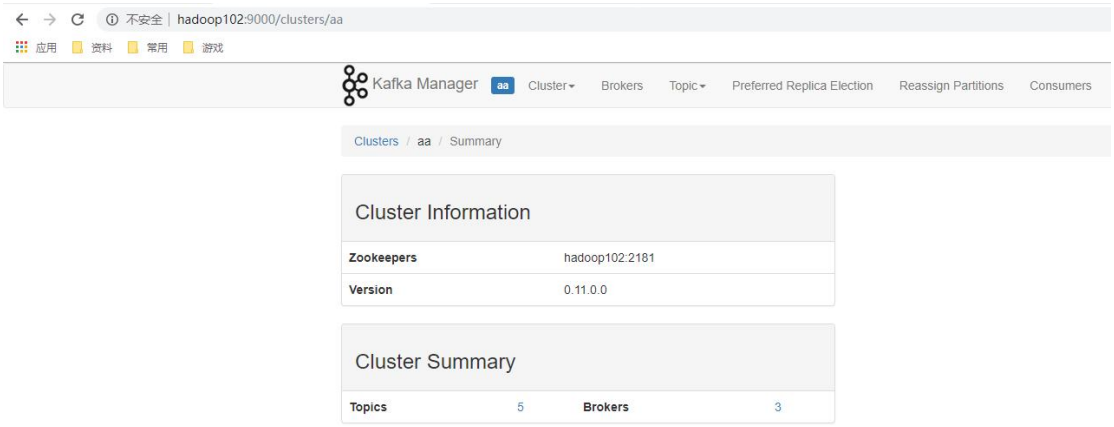
```
kafka-manager.zkhosts="kafka-manager-zookeeper:2181"
修改为：
kafka-manager.zkhosts="hadoop102:2181,hadoop103:2181,hadoop104:21
81"
```

4.启动

修改文件的权限

```
[atguigu@hadoop102 bin]$ chmod 777 kafka-manager

[atguigu@hadoop102 kafka-manager-1.3.3.15]$ bin/kafka-manager
```

5.登录 hadoop102:9000 页面查看详细信息



5.2 Kafka Monitor

1.上传 jar 包 KafkaOffsetMonitor-assembly-0.4.6.jar 到集群

2.在/opt/module/下创建 kafka-offset-console 文件夹

3.将上传的 jar 包放入刚创建的目录下

4.在/opt/module/kafka-offset-console 目录下创建启动脚本 start.sh，内容如下：

```
#!/bin/bash
java -cp KafkaOffsetMonitor-assembly-0.2.0.jar \
com.quantifind.kafka.offsetapp.OffsetGetterWeb \
--offsetStorage kafka \
--kafkaBrokers hadoop102:9092,hadoop103:9092,hadoop104:9092 \
--kafkaSecurityProtocol PLAINTEXT \
--zk hadoop102:2181,hadoop103:2181,hadoop104:2181 \
--port 8086 \
--refresh 10.seconds \
--retain 2.days \
--dbName offsetapp_kafka &
```

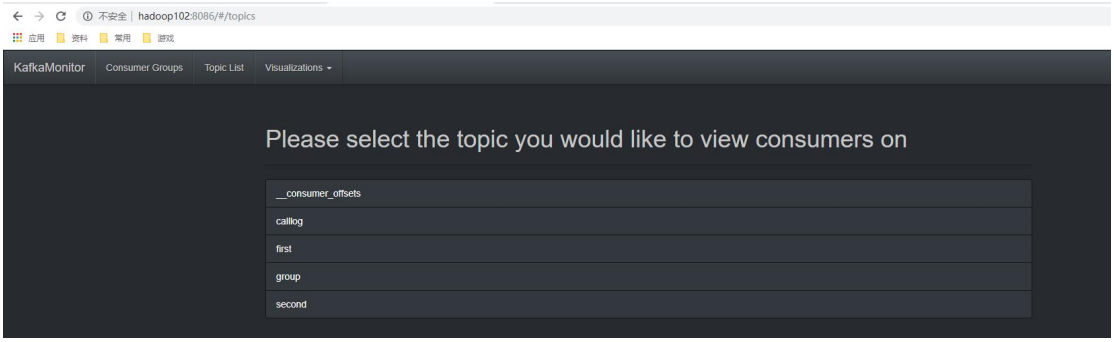
5.在/opt/module/kafka-offset-console 目录下创建 mobile-logs 文件夹

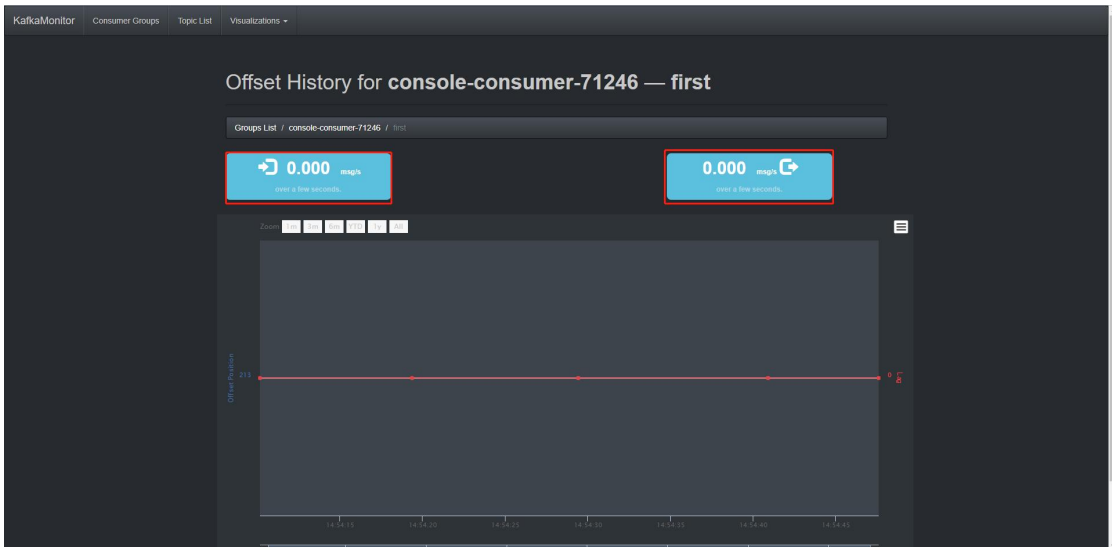
```
mkdir /opt/module/kafka-offset-console/mobile-logs
```

6.启动 KafkaMonitor

```
./start.sh
```

7.登录页面 hadoop102:8086 端口查看详情





第 6 章 Flume 对接 Kafka

1）配置 flume(flume-kafka.conf)

```
# define
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# source
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F -c +0 /opt/module/datas/flume.log
a1.sources.r1.shell = /bin/bash -c

# sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.bootstrap.servers                                    =
hadoop102:9092,hadoop103:9092,hadoop104:9092
a1.sinks.k1.kafka.topic = first
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1

# channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# bind
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

2） 启动 kafkaIDEA 消费者

3） 进入 flume 根目录下，启动 flume

```
$ bin/flume-ng agent -c conf/ -n a1 -f jobs/flume-kafka.conf
```

4） 向 /opt/module/datas/flume.log 里追加数据，查看 kafka 消费者消费情况

```
$ echo hello >> /opt/module/datas/flume.log
```

第 7 章 Kafka 面试题

7.1 面试问题

1.Kafka 中的 ISR(InSyncRepli)、OSR(OutSyncRepli)、AR(AllRepli)又代表什么？

2.Kafka 中的 HW、LEO 等分别代表什么？

3.Kafka 中是怎么体现消息顺序性的？

4.Kafka 中的分区器、序列化器、拦截器是否了解？它们之间的处理顺序是什么？

5.Kafka 生产者客户端的整体结构是什么样子的？使用了几个线程来处理？分别是什么？

6.“消费组中的消费者个数如果超过 topic 的分区，那么就会有消费者消费不到数据”这句话是否正确？

7.消费者提交消费位移时提交的是当前消费到的最新消息的 offset 还是 offset+1？

8.有哪些情形会造成重复消费？

9.那些情景会造成消息漏消费？

10.当你使用 kafka-topics.sh 创建（删除）了一个 topic 之后，Kafka 背后会执行什么逻辑？

1）会在 zookeeper 中的/brokers/topics 节点下创建一个新的 topic 节点，如：/brokers/topics/first

2）触发 Controller 的监听程序

3）kafka Controller 负责 topic 的创建工作，并更新 metadata cache

11.topic 的分区数可不可以增加？如果可以怎么增加？如果不可以，那又是为什么？

12.topic 的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么？

13.Kafka 有内部的 topic 吗？如果有是什么？有什么所用？

14.Kafka 分区分配的概念？

15.简述 Kafka 的日志目录结构？

16.如果我指定了一个 offset，Kafka Controller 怎么查找到对应的消息？

17.聊一聊 Kafka Controller 的作用？

18.Kafka 中有那些地方需要选举？这些地方的选举策略又有哪些？

19.失效副本是指什么？有那些应对措施？

20.Kafka 的哪些设计让它有如此高的性能？

7.2 参考答案

Kafka相关面试题及答案.docx