

尚硅谷大数据技术之 Kafka

(作者：尚硅谷研究院)

版本：V3.0.0

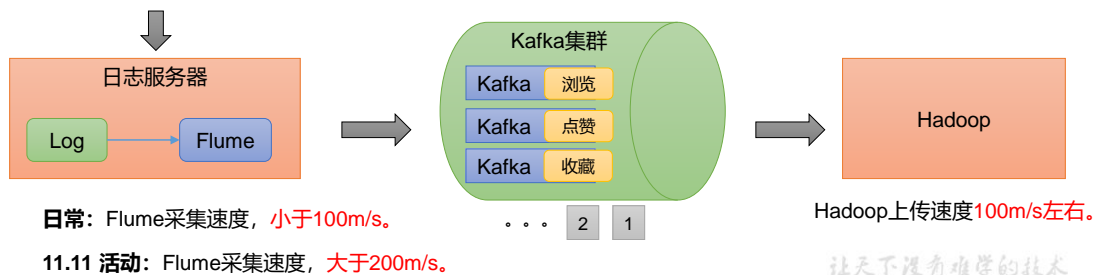
第 1 章 Kafka 概述

1.1 定义

Kafka定义



前端埋点记录用户购买海狗人参丸的行为数据（浏览、点赞、收藏、评论等）。



1.2 消息队列

目前企业中比较常见的消息队列产品主要有 Kafka、ActiveMQ、RabbitMQ、RocketMQ 等。

在大数据场景主要采用 Kafka 作为消息队列。在 JavaEE 开发中主要采用 ActiveMQ、RabbitMQ、RocketMQ。可以关注尚硅谷教育公众号回复 java，免费获取相关资料。

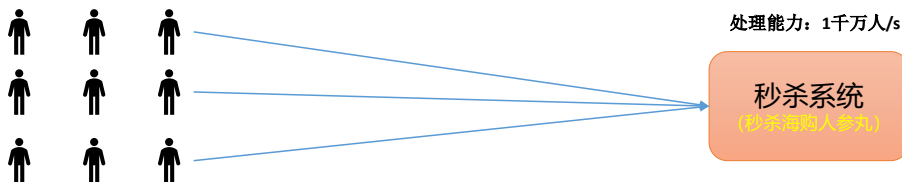
1.2.1 传统消息队列的应用场景

传统的消息队列的主要应用场景包括：缓存/消峰、解耦和异步通信。

消息队列的应用场景——缓冲/消峰

缓冲/消峰：有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。

双十一参与用户：10亿人/s



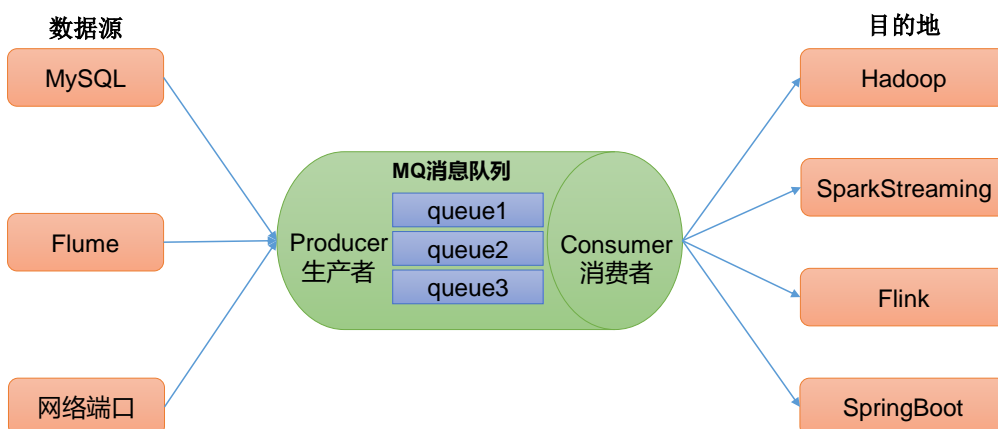
双十一参与用户：10亿人/s



让天下没有难学的技术

消息队列的应用场景——解耦

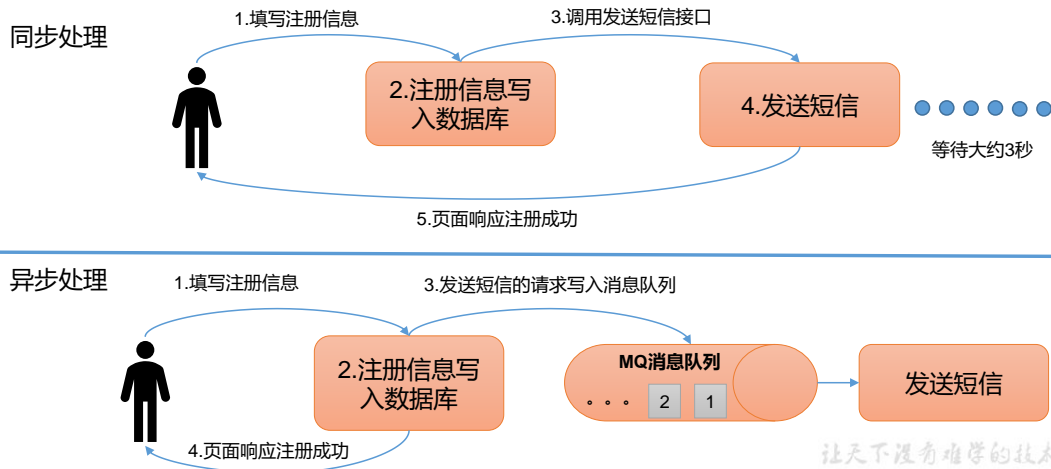
解耦：允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。



让天下没有难学的技术

消息队列的应用场景——异步通信

异步通信：允许用户把一个消息放入队列，但并不立即处理它，然后在需要的时候再去处理它们。



1.2.2 消息队列的两种模式

消息队列的两种模式

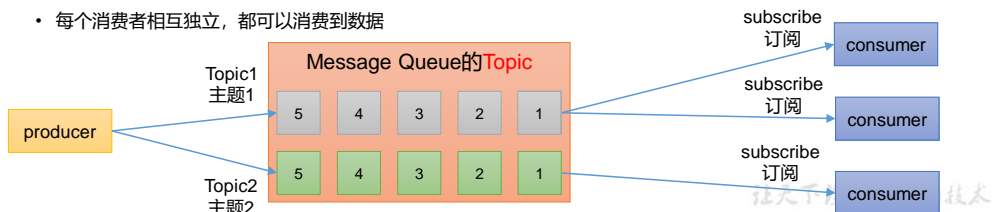
1) 点对点模式

- 消费者主动拉取数据，消息收到后清除消息



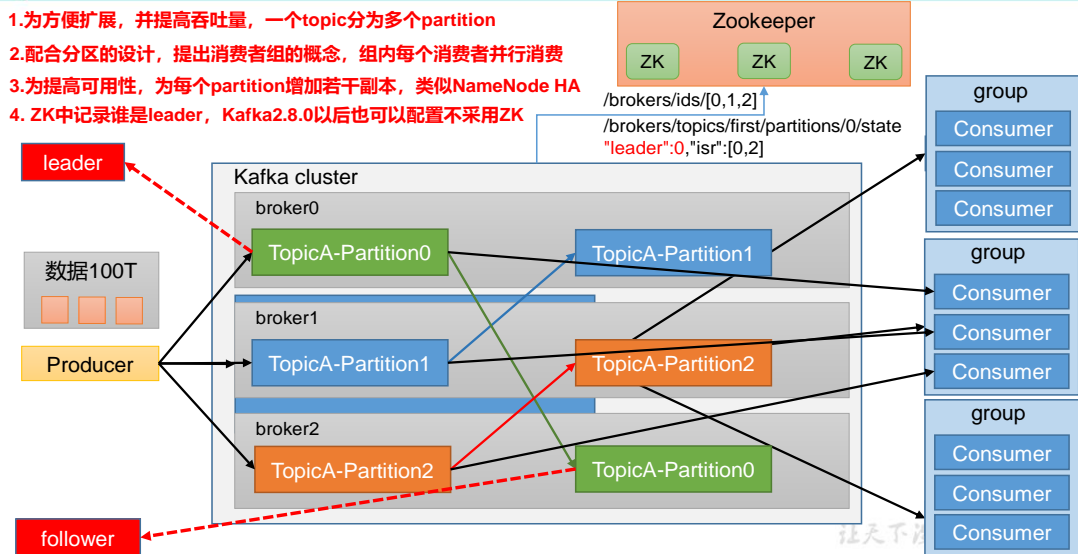
2) 发布/订阅模式

- 可以有多个topic主题（浏览、点赞、收藏、评论等）
- 消费者消费数据之后，不删除数据
- 每个消费者相互独立，都可以消费到数据



1.3 Kafka 基础架构

Kafka 基础架构



- (1) **Producer:** 消息生产者，就是向 Kafka broker 发消息的客户端。
- (2) **Consumer:** 消息消费者，向 Kafka broker 取消息的客户端。
- (3) **Consumer Group (CG):** 消费者组，由多个 consumer 组成。消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费；消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。
- (4) **Broker:** 一台 Kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker 可以容纳多个 topic。
- (5) **Topic:** 可以理解为一个队列，生产者和消费者面向的都是一个 topic。
- (6) **Partition:** 为了实现扩展性，一个非常大的 topic 可以分布到多个 broker（即服务器）上，一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列。
- (7) **Replica:** 副本。一个 topic 的每个分区都有若干个副本，一个 Leader 和若干个 Follower。
- (8) **Leader:** 每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是 Leader。
- (9) **Follower:** 每个分区多个副本中的“从”，实时从 Leader 中同步数据，保持和 Leader 数据的同步。Leader 发生故障时，某个 Follower 会成为新的 Leader。

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 集群部署

0) 官方下载地址: <http://kafka.apache.org/downloads.html>

1) 解压安装包

```
[atguigu@hadoop102 software]$ tar -zxvf kafka_2.12-3.0.0.tgz -C /opt/module/
```

2) 修改解压后的文件名称

```
[atguigu@hadoop102 module]$ mv kafka_2.12-3.0.0/ kafka
```

3) 进入到/opt/module/kafka 目录, 修改配置文件

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

输入以下内容:

```
#broker 的全局唯一编号, 不能重复, 只能是数字。  
broker.id=0  
#处理网络请求的线程数量  
num.network.threads=3  
#用来处理磁盘 IO 的线程数量  
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 (数据) 存放的路径, 路径不需要提前创建, kafka 自动帮你创建, 可以配置多个磁盘路径, 路径与路径之间可以用", "分隔  
log.dirs=/opt/module/kafka/datas  
# 若是使用云服务器, 需要配置下面两行, 才能从外网访问 kafka  
# 允许外部端口连接  
listeners=PLAINTEXT://0.0.0.0:9092  
# 外部代理地址  
advertised.listeners=PLAINTEXT://本机 IP:9092  
#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  
# 每个 topic 创建时的副本数, 默认时 1 个副本
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载, 可百度访问: 尚硅谷官网

```
offsets.topic.replication.factor=1
#segment 文件保留的最长时间，超时将被删除
log.retention.hours=168
#每个 segment 文件的大小，默认最大 1G
log.segment.bytes=1073741824
# 检查过期数据的时间，默认 5 分钟检查一次是否数据过期
log.retention.check.interval.ms=300000
#配置连接 Zookeeper 集群地址（在 zk 根目录下创建/kafka，方便管理）
zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181/kafka
```

4) 分发安装包

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

5) 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，整个集群中唯一。

```
[atguigu@hadoop103 module]$ vim kafka/config/server.properties
修改：
# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1

[atguigu@hadoop104 module]$ vim kafka/config/server.properties
修改：
# The id of the broker. This must be set to a unique integer for each broker.
broker.id=2
```

记得修改外网访问 IP

6) 配置环境变量

(1) 在/etc/profile.d/my_env.sh 文件中增加 kafka 环境变量配置

```
[atguigu@hadoop102 module]$ sudo vim /etc/profile.d/my_env.sh
```

增加如下内容：

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

(2) 刷新一下环境变量。

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

(3) 分发环境变量文件到其他节点，并 source。

```
[atguigu@hadoop102 module]$ sudo /home/atguigu/bin/xcsync /etc/profile.d/my_env.sh
[atguigu@hadoop103 module]$ source /etc/profile
[atguigu@hadoop104 module]$ source /etc/profile
```

7) 启动集群

(1) 先启动 Zookeeper 集群，然后启动 Kafka。

```
[atguigu@hadoop102 kafka]$ zk.sh start
```

(2) 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
```

注意：配置文件的路径要能够到 `server.properties`。

若 `kafka` 启动后一直自动关闭，查看 `log` 日志发现是 `clusterID` 出错，请先确保 `zk` 集群配置完毕，然后启动 `kafka` 前将每个 `kafka` 下 `datas` 文件夹下的 `meta.properties` 删除。

8) 关闭集群

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

2.1.3 集群启停脚本

1) 在 `/home/atguigu/bin` 目录下创建文件 `kf.sh` 脚本文件

```
[atguigu@hadoop102 bin]$ vim kf.sh
```

脚本如下：

```
#!/bin/bash

case $1 in
"start"){
    for i in hadoop102 hadoop103 hadoop104
    do
        echo " -----启动 $i Kafka-----"
        ssh $i "/opt/module/kafka/bin/kafka-server-start.sh -daemon /opt/module/kafka/config/server.properties"
    done
};;
"stop"){
    for i in hadoop102 hadoop103 hadoop104
    do
        echo " -----停止 $i Kafka-----"
        ssh $i "/opt/module/kafka/bin/kafka-server-stop.sh "
    done
};;
esac
```

2) 添加执行权限

```
[atguigu@hadoop102 bin]$ chmod +x kf.sh
```

3) 启动集群命令

```
[atguigu@hadoop102 ~]$ kf.sh start
```

4) 停止集群命令

```
[atguigu@hadoop102 ~]$ kf.sh stop
```

注意：停止 `Kafka` 集群时，一定要等 `Kafka` 所有节点进程全部停止后再停止 `Zookeeper`

集群。因为 Zookeeper 集群当中记录着 Kafka 集群相关信息，Zookeeper 集群一旦先停止，Kafka 集群就没有办法再获取停止进程的信息，只能手动杀死 Kafka 进程了。

2.2 Kafka 命令行操作

2.2.1 主题命令行操作

1) 查看操作主题命令参数

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh
```

参数	描述
--bootstrap-server <String: server to connect to>	连接的 Kafka Broker 主机名称和端口号。
--topic <String: topic>	操作的 topic 名称。
--create	创建主题。
--delete	删除主题。
--alter	修改主题。
--list	查看所有主题。
--describe	查看主题详细描述。
--partitions <Integer: # of partitions>	设置分区数。
--replication-factor <Integer: replication factor>	设置分区副本。
--config <String: name=value>	更新系统默认的配置。

2) 查看当前服务器中的所有 topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --list
```

3) 创建 first topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --create --partitions 1 --replication-factor 3 --topic first
```

选项说明：

--topic 定义 topic 名

--replication-factor 定义副本数

--partitions 定义分区数

4) 查看 first 主题的详情

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic first
```


5) 修改分区数（注意：分区数只能增加，不能减少）

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server
hadoop102:9092 --alter --topic first --partitions 3
```

6) 再次查看 first 主题的详情

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server
hadoop102:9092 --describe --topic first
```

7) 删除 topic（学生自己演示）

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server
hadoop102:9092 --delete --topic first
```

2.2.2 生产者命令行操作

1) 查看操作生产者命令参数

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh
```

参数	描述
--bootstrap-server <String: server to connect to>	连接的 Kafka Broker 主机名称和端口号。
--topic <String: topic>	操作的 topic 名称。

2) 发送消息

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

2.2.3 消费者命令行操作

1) 查看操作消费者命令参数

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh
```

参数	描述
--bootstrap-server <String: server to connect to>	连接的 Kafka Broker 主机名称和端口号。
--topic <String: topic>	操作的 topic 名称。
--from-beginning	从头开始消费。
--group <String: consumer group id>	指定消费者组名称。

2) 消费消息

（1）消费 first 主题中的数据。

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

（2）把主题中的所有数据都读取出来（包括历史数据）。

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

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

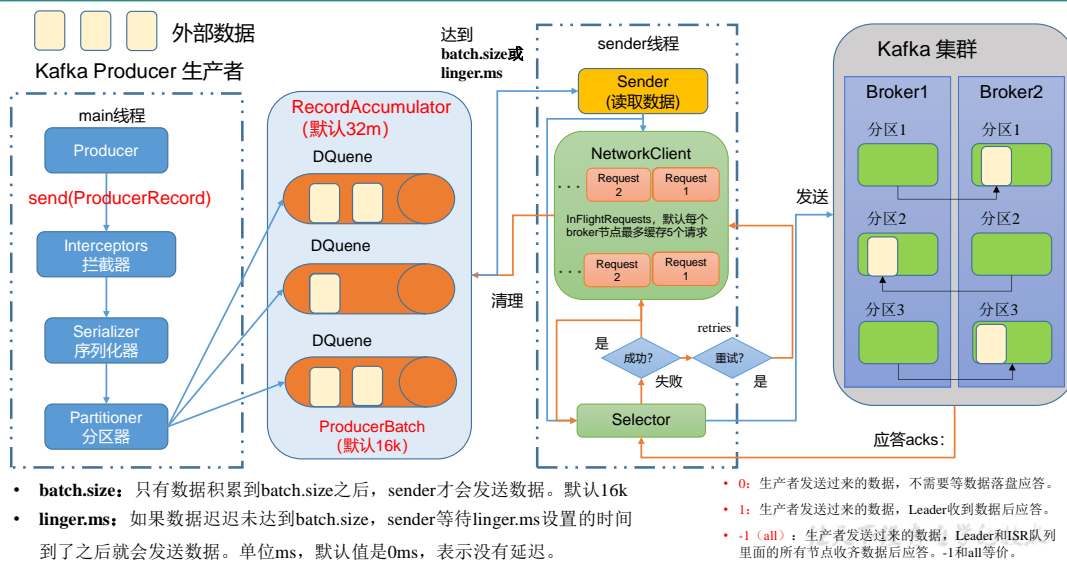
第 3 章 Kafka 生产者

3.1 生产者消息发送流程

3.1.1 发送原理

在消息发送的过程中，涉及到了**两个线程——main 线程和 Sender 线程**。在 main 线程中创建了一个**双端队列 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。

发送流程



3.1.2 生产者重要参数列表

参数名称	描述
bootstrap.servers	生产者连接集群所需的 broker 地址清单。例如 hadoop102:9092,hadoop103:9092,hadoop104:9092，可以设置 1 个或者多个，中间用逗号隔开。注意这里并非需要所有的 broker 地址，因为生产者从给定的 broker 里查找到其他 broker 信息。
key.serializer 和 value.serializer	指定发送消息的 key 和 value 的序列化类型。一定要写全类名。
buffer.memory	RecordAccumulator 缓冲区总大小，默认 32m。
batch.size	缓冲区一批数据最大值，默认 16k。适当增加该值，可以提高吞吐量，但是如果该值设置太大，会导致数据传输延迟增加。

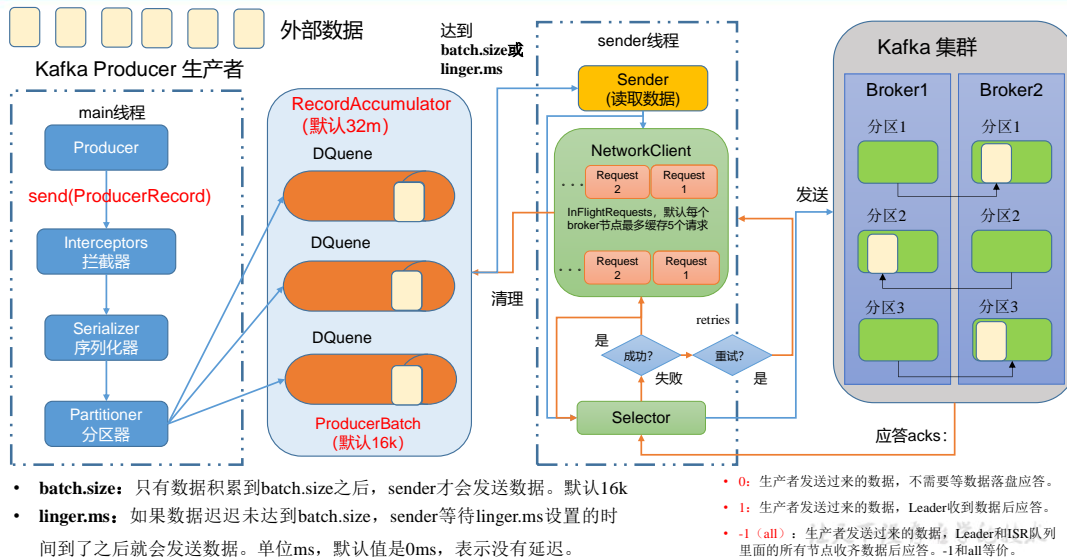
linger.ms	如果数据迟迟未达到 batch.size, sender 等待 linger.time 之后就会发送数据。单位 ms, 默认值是 0ms, 表示没有延迟。生产环境建议该值大小为 5-100ms 之间。
acks	0: 生产者发送过来的数据, 不需要等数据落盘应答。 1: 生产者发送过来的数据, Leader 收到数据后应答。 -1 (all): 生产者发送过来的数据, Leader+和 isr 队列里面的所有节点收齐数据后应答。默认值是 1, -1 和 all 是等价的。
max.in.flight.requests.per.connection	允许最多没有返回 ack 的次数, 默认为 5, 开启幂等性要保证该值是 1-5 的数字。
retries	当消息发送出现错误的时候, 系统会重发消息。retries 表示重试次数。默认是 int 最大值, 2147483647。 如果设置了重试, 还想保证消息的有序性, 需要设置 MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION=1 否则在重试此失败消息的时候, 其他的消息可能发送成功了。
retry.backoff.ms	两次重试之间的时间间隔, 默认是 100ms。
enable.idempotence	是否开启幂等性, 默认 true, 开启幂等性。
compression.type	生产者发送的所有数据的压缩方式。默认是 none, 也就是不压缩。 支持压缩类型: none、gzip、snappy、lz4 和 zstd。

3.2 异步发送 API

3.2.1 普通异步发送

1) 需求: 创建 Kafka 生产者, 采用异步的方式发送到 Kafka Broker

异步发送流程



2) 代码编写

(1) 创建工程 kafka

(2) 导入依赖

```
<dependencies>
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.0.0</version>
  </dependency>
</dependencies>
```

(3) 创建包名: com.atguigu.kafka.producer

(4) 编写不带回调函数的 API 代码

```
package com.atguigu.kafka.producer;

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

import java.util.Properties;

public class CustomProducer {

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

        // 1. 创建 kafka 生产者的配置对象
        Properties properties = new Properties();

        // 2. 给 kafka 配置对象添加配置信息: bootstrap.servers
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // key,value 序列化 (必须): key.serializer, value.serializer
```

```
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");

        // 3. 创建 kafka 生产者对象
        KafkaProducer<String, String> kafkaProducer = new
KafkaProducer<String, String>(properties);

        // 4. 调用 send 方法, 发送消息
        for (int i = 0; i < 5; i++) {

            kafkaProducer.send(new
ProducerRecord<>("first", "atguigu " + i));

        }

        // 5. 关闭资源
        kafkaProducer.close();
    }
}
```

测试:

①在 hadoop102 上开启 Kafka 消费者。

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

②在 IDEA 中执行代码, 观察 hadoop102 控制台中是否接收到消息。

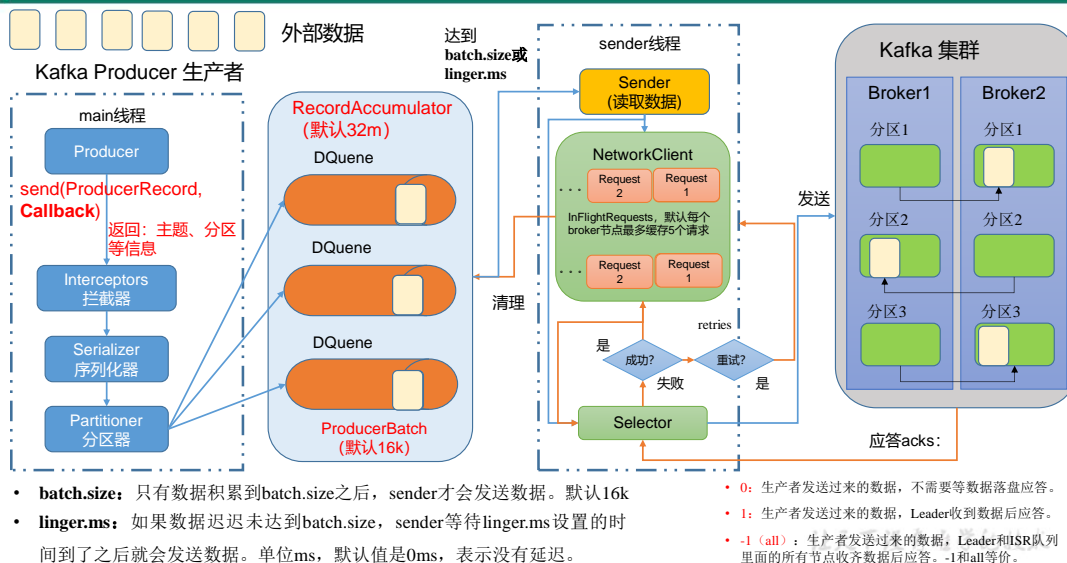
```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh --
bootstrap-server hadoop102:9092 --topic first

atguigu 0
atguigu 1
atguigu 2
atguigu 3
atguigu 4
```

3.2.2 带回调函数的异步发送

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, 分别是元数据信息 (RecordMetadata) 和异常信息 (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。

带回调函数的异步发送流程



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.producer;

import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class CustomProducerCallback {

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

        // 1. 创建 kafka 生产者的配置对象
        Properties properties = new Properties();

        // 2. 给 kafka 配置对象添加配置信息
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // key,value 序列化 (必须): key.serializer, value.serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());

        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());

        // 3. 创建 kafka 生产者对象
        KafkaProducer<String, String> kafkaProducer = new
            KafkaProducer<String, String>(properties);

        // 4. 调用 send 方法, 发送消息
        for (int i = 0; i < 5; i++) {

            // 添加回调
            kafkaProducer.send(new ProducerRecord<>("first",
                "atguigu " + i), new Callback() {
```

```
        // 该方法在 Producer 收到 ack 时调用，为异步调用
        @Override
        public void onCompletion(RecordMetadata metadata,
Exception exception) {

            if (exception == null) {
                // 没有异常,输出信息到控制台
                System.out.println(" 主 题 : " +
metadata.topic() + "->" + "分区: " + metadata.partition());
            } else {
                // 出现异常打印
                exception.printStackTrace();
            }
        }
    });

    // 延迟一会会看到数据发往不同分区
    Thread.sleep(2);
}

// 5. 关闭资源
kafkaProducer.close();
}
```

测试:

①在 hadoop102 上开启 Kafka 消费者。

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

②在 IDEA 中执行代码，观察 hadoop102 控制台中是否接收到消息。

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

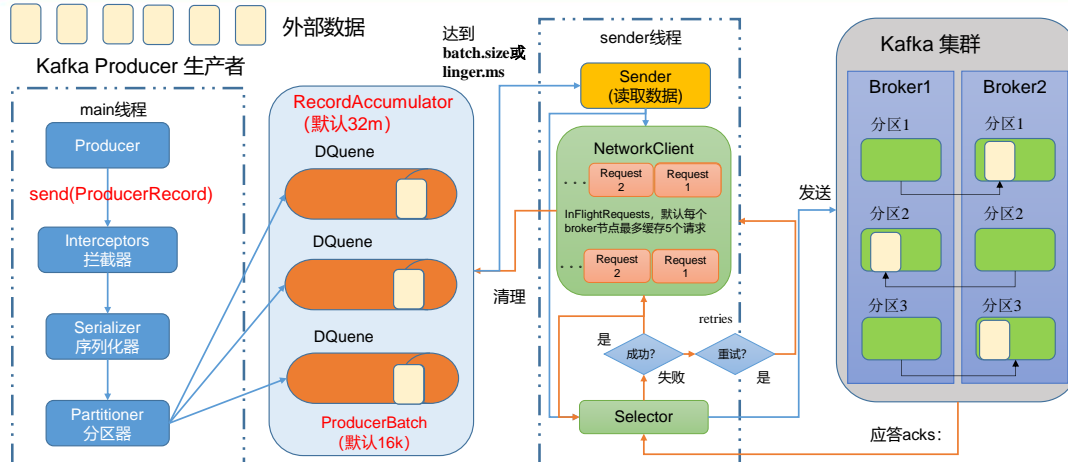
atguigu 0
atguigu 1
atguigu 2
atguigu 3
atguigu 4
```

③在 IDEA 控制台观察回调信息。

```
主题: first->分区: 0
主题: first->分区: 0
主题: first->分区: 1
主题: first->分区: 1
主题: first->分区: 1
```

3.3 同步发送 API

同步发送流程



- **batch.size**: 只有数据积累到 batch.size 之后，sender 才会发送数据。默认 16k
- **linger.ms**: 如果数据迟迟未达到 batch.size，sender 等待 linger.ms 设置的时间到了之后就会发送数据。单位 ms，默认值是 0ms，表示没有延迟。

- **0**: 生产者发送过来的数据，不需要等数据落盘应答。
- **1**: 生产者发送过来的数据，Leader 收到数据后应答。
- **-1 (all)**: 生产者发送过来的数据，Leader 和 ISR 队列里面的所有节点收齐数据后应答。-1 和 all 等价。

只需在异步发送的基础上，再调用一下 `get()` 方法即可。

```
package com.atguigu.kafka.producer;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class CustomProducerSync {

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

        // 1. 创建 kafka 生产者的配置对象
        Properties properties = new Properties();

        // 2. 给 kafka 配置对象添加配置信息

        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "hadoop102:9092");

        // key,value 序列化 (必须): key.serializer, value.serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        // 3. 创建 kafka 生产者对象
        KafkaProducer<String, String> kafkaProducer = new KafkaProducer<String, String>(properties);
```



```
// 4. 调用 send 方法, 发送消息
for (int i = 0; i < 10; i++) {

    // 异步发送 默认
    kafkaProducer.send(new
ProducerRecord<>("first", "kafka" + i));

    // 同步发送
    kafkaProducer.send(new
ProducerRecord<>("first", "kafka" + i)).get();

}

// 5. 关闭资源
kafkaProducer.close();

}
}
```

测试:

①在 hadoop102 上开启 Kafka 消费者。

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

②在 IDEA 中执行代码, 观察 hadoop102 控制台中是否接收到消息。

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

atguigu 0
atguigu 1
atguigu 2
atguigu 3
atguigu 4
```

3.4 生产者分区

3.4.1 分区好处

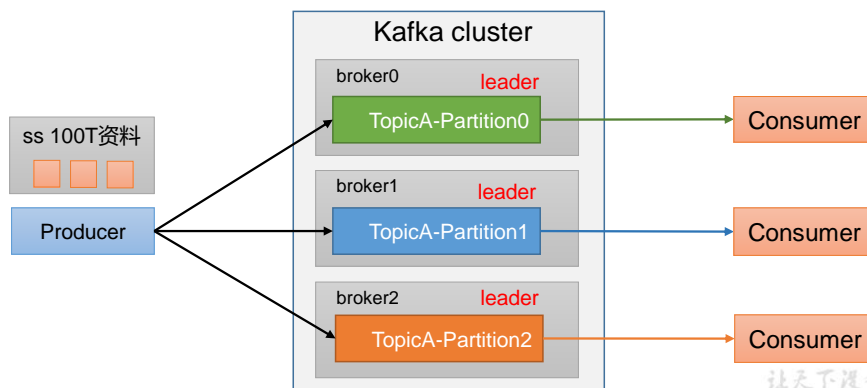


Kafka 分区好处



(1) **便于合理使用存储资源**, 每个Partition在一个Broker上存储, 可以把海量的数据按照分区切割成一块一块数据存储在多台Broker上。合理控制分区的任务, 可以实现**负载均衡**的效果。

(2) **提高并行度**, 生产者可以以分区为单位**发送数据**; 消费者可以以分区为单位进行**消费数据**。



让天下没有难学的技术

3.4.2 生产者发送消息的分区策略

1) 默认的分区器 DefaultPartitioner

在 IDEA 中 ctrl +n，全局查找 DefaultPartitioner。

```
/**
 * The default partitioning strategy:
 * <ul>
 * <li>If a partition is specified in the record, use it
 * <li>If no partition is specified but a key is present choose a
 * partition based on a hash of the key
 * <li>If no partition or key is present choose the sticky
 * partition that changes when the batch is full.
 * </ul>
 * See KIP-480 for details about sticky partitioning.
 */
public class DefaultPartitioner implements Partitioner {

    ... ..
}
```



Kafka 原则



在IDEA中全局查找（ctrl +n） **ProducerRecord**类，在类中可以看到如下构造方法：

```
public ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value, Iterable<Header> headers) {
    ... ..
}
public ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value) {
    ... ..
}
public ProducerRecord(String topic, Integer partition, K key, V value, Iterable<Header> headers) {
    ... ..
}
public ProducerRecord(String topic, Integer partition, K key, V value) {
    ... ..
}
public ProducerRecord(String topic, K key, V value) {
    ... ..
}
public ProducerRecord(String topic, V value) {
    ... ..
}
```

(1) 指明partition的情况下，直接将指明的值作为partition值；
例如partition=0，所有数据写入分区0

(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；
例如：key1的hash值=5，key2的hash值=6，topic的partition数=2，那么key1对应的value1写入1号分区，key2对应的value2写入0号分区。

(3) 既没有partition值又没有key值的情况下，Kafka采用Sticky Partition（黏性分区器），会随机选择一个分区，并尽可能一直使用该分区，待该分区的batch已满或者已完成，Kafka再随机一个分区进行使用（和上一次的分区不同）。
例如：第一次随机选择0号分区，等0号分区当前批次满了（默认16k）或者linger.ms设置的时间到，Kafka再随机一个分区进行使用（如果还是0会继续随机）。

2) 案例一

将数据发往指定 partition 的情况下，例如，将所有数据发往分区 1 中。

```
package com.atguigu.kafka.producer;

import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class CustomProducerCallbackPartitions {

    public static void main(String[] args) {

        // 1. 创建 kafka 生产者的配置对象
        Properties properties = new Properties();
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

```
// 2. 给 kafka 配置对象添加配置信息

properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "hadoop102:9092");

// key,value 序列化 (必须): key.serializer, value.serializer
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

KafkaProducer<String, String> kafkaProducer = new
KafkaProducer<>(properties);

for (int i = 0; i < 5; i++) {
    // 指定数据发送到 1 号分区, key 为空 (IDEA 中 ctrl + p 查看参数)
    kafkaProducer.send(new ProducerRecord<>("first",
1, "", "atguigu " + i), new Callback() {
        @Override
        public void onCompletion(RecordMetadata metadata,
Exception e) {
            if (e == null){
                System.out.println(" 主 题 : " +
metadata.topic() + "->" + "分区: " + metadata.partition()
);
            }else {
                e.printStackTrace();
            }
        }
    });
}

kafkaProducer.close();
}
```

测试:

①在 hadoop102 上开启 Kafka 消费者。

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

②在 IDEA 中执行代码, 观察 hadoop102 控制台中是否接收到消息。

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

atguigu 0
atguigu 1
atguigu 2
atguigu 3
atguigu 4
```

③在 IDEA 控制台观察回调信息。

```
主题: first->分区: 1
```

```
主题: first->分区: 1
主题: first->分区: 1
主题: first->分区: 1
主题: first->分区: 1
```

3) 案例二

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。

```
package com.atguigu.kafka.producer;

import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class CustomProducerCallback {

    public static void main(String[] args) {

        Properties properties = new Properties();

        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "hadoop102:9092");

        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(properties);

        for (int i = 0; i < 5; i++) {
            // 依次指定 key 值为 a,b,f ，数据 key 的 hash 值与 3 个分区求余，分别发往 1、2、0
            kafkaProducer.send(new ProducerRecord<>("first", "a", "atguigu " + i), new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception e) {
                    if (e == null){
                        System.out.println(" 主 题 : " + metadata.topic() + "->" + "分区: " + metadata.partition());
                    }else {
                        e.printStackTrace();
                    }
                }
            });
        }

        kafkaProducer.close();
    }
}
```

测试：

①key="a"时，在控制台查看结果。

```
主题: first->分区: 1
主题: first->分区: 1
主题: first->分区: 1
主题: first->分区: 1
主题: first->分区: 1
```

②key="b"时，在控制台查看结果。

```
主题: first->分区: 2
主题: first->分区: 2
主题: first->分区: 2
主题: first->分区: 2
主题: first->分区: 2
```

③key="f"时，在控制台查看结果。

```
主题: first->分区: 0
主题: first->分区: 0
主题: first->分区: 0
主题: first->分区: 0
主题: first->分区: 0
```

3.4.3 自定义分区器

如果研发人员可以根据企业需求，自己重新实现分区器。

1) 需求

例如我们实现一个分区器实现，发送过来的数据中如果包含 `atguigu`，就发往 0 号分区，不包含 `atguigu`，就发往 1 号分区。

2) 实现步骤

(1) 定义类实现 `Partitioner` 接口。

(2) 重写 `partition()` 方法。

```
package com.atguigu.kafka.producer;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;

import java.util.Map;

/**
 * 1. 实现接口 Partitioner
 * 2. 实现 3 个方法:partition,close,configure
 * 3. 编写 partition 方法,返回分区号
 */
public class MyPartitioner implements Partitioner {

    /**
```

```
* 返回信息对应的分区
* @param topic      主题
* @param key        消息的 key
* @param keyBytes   消息的 key 序列化后的字节数组
* @param value      消息的 value
* @param valueBytes 消息的 value 序列化后的字节数组
* @param cluster    集群元数据可以查看分区信息
* @return
*/
@Override
public int partition(String topic, Object key, byte[]
keyBytes, Object value, byte[] valueBytes, Cluster cluster) {

    // 获取消息
    String msgValue = value.toString();

    // 创建 partition
    int partition;

    // 判断消息是否包含 atguigu
    if (msgValue.contains("atguigu")) {
        partition = 0;
    } else {
        partition = 1;
    }

    // 返回分区号
    return partition;
}

// 关闭资源
@Override
public void close() {

}

// 配置方法
@Override
public void configure(Map<String, ?> configs) {

}
}
```

(3) 使用分区器的方法，在生产者的配置中添加分区器参数。

```
package com.atguigu.kafka.producer;

import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class CustomProducerCallbackPartitions {

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

```
Properties properties = new Properties();

properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "hadoop102:9092");

properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

// 添加自定义分区器
properties.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "com.atguigu.kafka.producer.MyPartitioner");

KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(properties);

for (int i = 0; i < 5; i++) {

    kafkaProducer.send(new ProducerRecord<>("first", "atguigu " + i), new Callback() {
        @Override
        public void onCompletion(RecordMetadata metadata, Exception e) {
            if (e == null){
                System.out.println(" 主 题 : " + metadata.topic() + "->" + "分区: " + metadata.partition());
            }else {
                e.printStackTrace();
            }
        }
    });
}

kafkaProducer.close();
}
```

(4) 测试

①在 hadoop102 上开启 Kafka 消费者。

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

②在 IDEA 控制台观察回调信息。

```
主题: first->分区: 0
主题: first->分区: 0
主题: first->分区: 0
主题: first->分区: 0
主题: first->分区: 0
```

3.5 生产经验——生产者如何提高吞吐量

生产经验——生产者如何提高吞吐量

ss家仓库,默认32m



一次拉一个，
来了就走



broker



- batch.size: 批次大小, 默认16k
- linger.ms: 等待时间, 修改为5-100ms
- compression.type: 压缩snappy
- RecordAccumulator: 缓冲区大小, 修改为64m



```
package com.atguigu.kafka.producer;

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

import java.util.Properties;

public class CustomProducerParameters {

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

        // 1. 创建 kafka 生产者的配置对象
        Properties properties = new Properties();

        // 2. 给 kafka 配置对象添加配置信息: bootstrap.servers
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // key,value 序列化 (必须): key.serializer, value.serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // batch.size: 批次大小, 默认 16K
        properties.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);

        // linger.ms: 等待时间, 默认 0
        properties.put(ProducerConfig.LINGER_MS_CONFIG, 1);

        // RecordAccumulator: 缓冲区大小, 默认 32M: buffer.memory
        properties.put(ProducerConfig.BUFFER_MEMORY_CONFIG,
```



```
33554432);

        // compression.type: 压缩, 默认 none, 可配置值 gzip、snappy、
        lz4 和 zstd
        properties.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");

        // 3. 创建 kafka 生产者对象
        KafkaProducer<String, String> kafkaProducer = new
        KafkaProducer<String, String>(properties);

        // 4. 调用 send 方法, 发送消息
        for (int i = 0; i < 5; i++) {

            kafkaProducer.send(new
            ProducerRecord<>("first", "atguigu " + i));

        }

        // 5. 关闭资源
        kafkaProducer.close();
    }
}
```

测试:

①在 hadoop102 上开启 Kafka 消费者。

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

②在 IDEA 中执行代码, 观察 hadoop102 控制台中是否接收到消息。

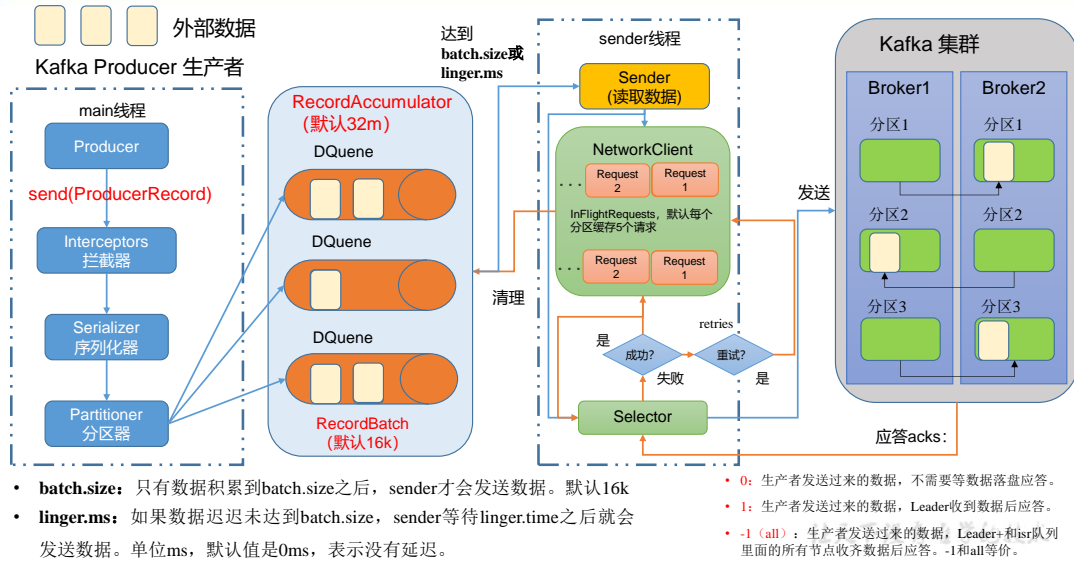
```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh --
bootstrap-server hadoop102:9092 --topic first

atguigu 0
atguigu 1
atguigu 2
atguigu 3
atguigu 4
```

3.6 生产经验——数据可靠性

0) 回顾发送流程

发送流程

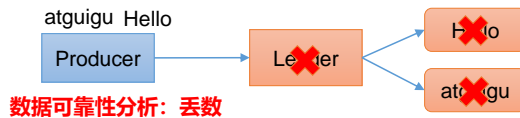


1) ack 应答原理

ACK 应答级别

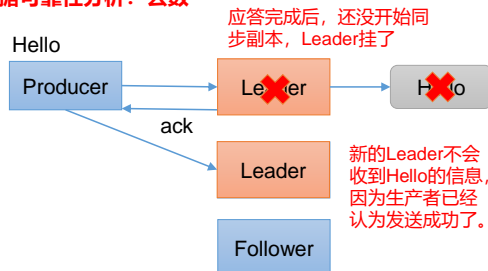
acks:

0: 生产者发送过来的数据，不需要等数据落盘应答

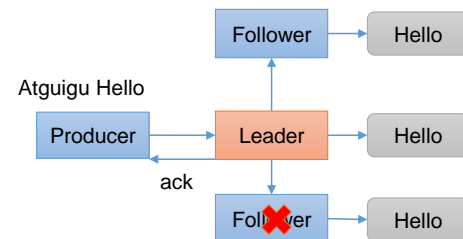


1: 生产者发送过来的数据，Leader 收到数据后应答。

数据可靠性分析：丢数



-1 (all): 生产者发送过来的数据，Leader 和 ISR 队列里面的所有节点收齐数据后应答。



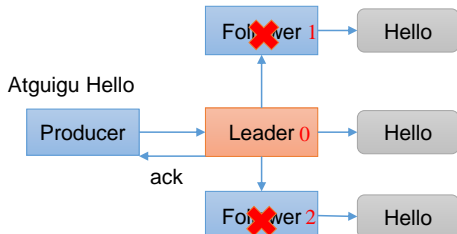
思考: Leader 收到数据，所有 Follower 都开始同步数据，但有一个 Follower，因为某种故障，迟迟不能与 Leader 进行同步，那这个问题怎么解决呢？

让天下没有难学的技术

ACK应答级别

acks:

-1 (all): 生产者发送过来的数据, Leader和ISR队列里面的所有节点收齐数据后应答。



思考: 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+Leader集合(leader: 0, isr:0,1,2)。

如果Follower长时间未向Leader发送通信请求或同步数据, 则该Follower将被踢出ISR。该时间阈值由replica.lag.time.max.ms参数设定, 默认30s。例如2超时, (leader:0, isr:0,1)。

这样就不用等长期联系不上或者已经故障的节点。

数据可靠性分析:

如果分区副本设置为1个, 或者ISR里应答的最小副本数量 (min.insync.replicas 默认为1) 设置为1, 和ack=1的效果是一样的, 仍然有丢数的风险 (leader: 0, isr:0)。

- 数据完全可靠条件 = ACK级别设置为-1 + 分区副本大于等于2 + ISR里应答的最小副本数量大于等于2

让天下没有难学的技术

ACK应答级别

可靠性总结:

acks=0, 生产者发送过来数据就不管了, 可靠性差, 效率高;

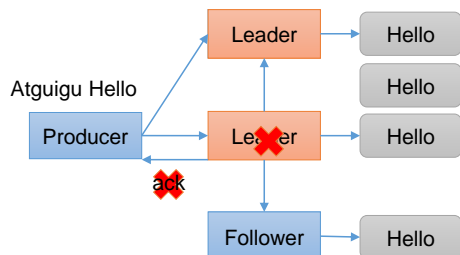
acks=1, 生产者发送过来数据Leader应答, 可靠性中等, 效率中等;

acks=-1, 生产者发送过来数据Leader和ISR队列里面所有Follower应答, 可靠性高, 效率低;

在生产环境中, acks=0很少使用; acks=1, 一般用于传输普通日志, 允许丢个别数据; acks=-1, 一般用于传输和钱相关的数据, 对可靠性要求比较高的场景。

数据重复分析:

acks: -1 (all): 生产者发送过来的数据, Leader和ISR队列里面的所有节点收齐数据后应答。



接收了两份Hello数据, 导致数据重复

具体如何解决数据重复? 下回分解。

让天下没有难学的技术

2) 代码配置

```
package com.atguigu.kafka.producer;

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

import java.util.Properties;

public class CustomProducerAck {

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

        // 1. 创建 kafka 生产者的配置对象
        Properties properties = new Properties();

        // 2. 给 kafka 配置对象添加配置信息: bootstrap.servers
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载, 可百度访问: 尚硅谷官网

```
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"hadool02:9092");

        // key,value 序列化 (必须): key.serializer, value.serializer
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

        // 设置 acks
properties.put(ProducerConfig.ACKS_CONFIG, "all");

        // 重试次数 retries, 默认是 int 最大值, 2147483647
properties.put(ProducerConfig.RETRIES_CONFIG, 3);

        // 3. 创建 kafka 生产者对象
        KafkaProducer<String, String> kafkaProducer = new
KafkaProducer<String, String>(properties);

        // 4. 调用 send 方法, 发送消息
        for (int i = 0; i < 5; i++) {

            kafkaProducer.send(new
ProducerRecord<>("first", "atguigu " + i));

        }

        // 5. 关闭资源
        kafkaProducer.close();
    }
}
```

3.7 生产经验——数据去重

3.7.1 数据传递语义

数据传递语义



- 至少一次 (At Least Once) = ACK级别设置为-1 + 分区副本大于等于2 + ISR里应答的最小副本数量大于等于2
 - 最多一次 (At Most Once) = ACK级别设置为0
 - 总结:
 - At Least Once可以保证数据不丢失, 但是不能保证数据不重复;
 - At Most Once可以保证数据不重复, 但是不能保证数据不丢失。
 - 精确一次 (Exactly Once): 对于一些非常重要的信息, 比如和钱相关的数据, 要求数据既不能重复也不丢失。
- Kafka 0.11版本以后, 引入了一项重大特性: 幂等性和事务。

让天下没有难学的技术

3.7.2 幂等性

1) 幂等性原理

幂等性原理

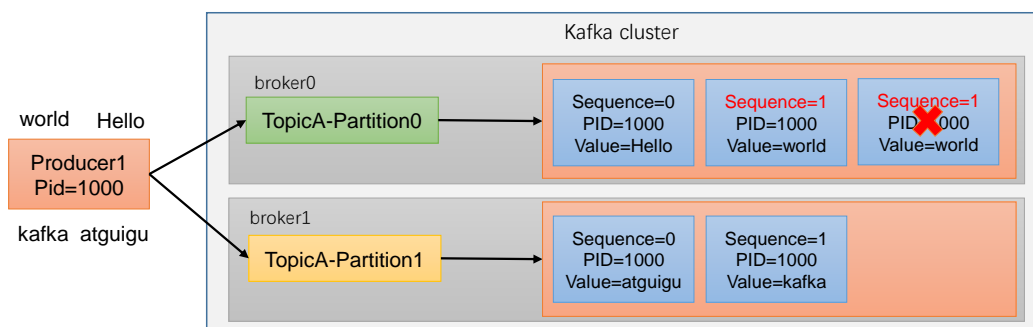


幂等性就是指Producer不论向Broker发送多少次重复数据, Broker端都只会持久化一条, 保证了不重复。

精确一次 (Exactly Once) = 幂等性 + 至少一次 (ack=-1 + 分区副本数>=2 + ISR最小副本数量>=2)。

重复数据的判断标准: 具有<PID, Partition, SeqNumber>相同主键的消息提交时, Broker只会持久化一条。其中PID是Kafka每次重启都会分配一个新的; Partition 表示分区号; Sequence Number是单调自增的。

所以幂等性只能保证的是在单分区单会话内不重复。



2) 如何使用幂等性

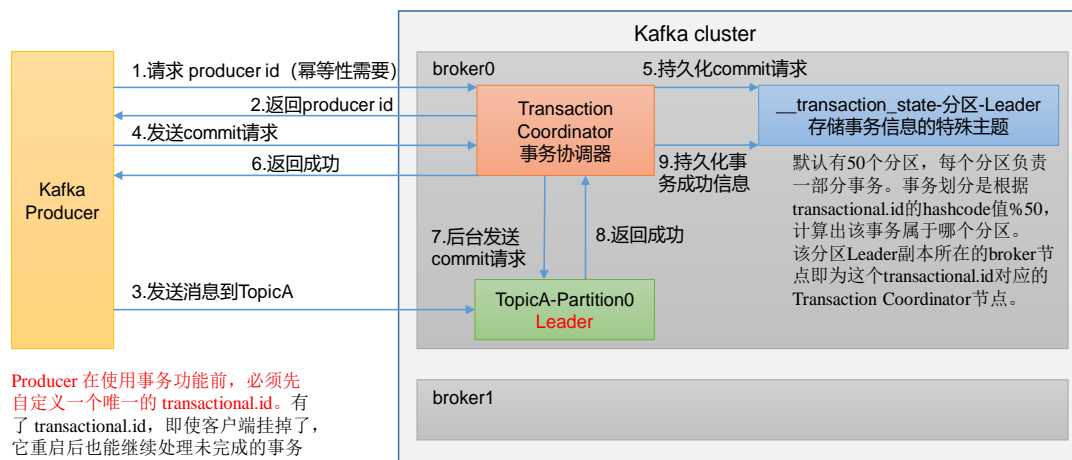
开启参数 `enable.idempotence` 默认为 true, false 关闭。

3.7.3 生产者事务

1) Kafka 事务原理

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)

说明：开启事务，必须开启幂等性。



让天下没有难学的技术

2) Kafka 的事务一共有如下 5 个 API

```
// 1 初始化事务
void initTransactions();

// 2 开启事务
void beginTransaction() throws ProducerFencedException;

// 3 在事务内提交已经消费的偏移量（主要用于消费者）
void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets,
                               String consumerGroupId) throws
ProducerFencedException;

// 4 提交事务
void commitTransaction() throws ProducerFencedException;

// 5 放弃事务（类似于回滚事务的操作）
void abortTransaction() throws ProducerFencedException;
```

3) 单个 Producer，使用事务保证消息的仅一次发送

```
package com.atguigu.kafka.producer;

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

import java.util.Properties;

public class CustomProducerTransactions {

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

        // 1. 创建 kafka 生产者的配置对象
        Properties properties = new Properties();

        // 2. 给 kafka 配置对象添加配置信息
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
"hadoop102:9092");

    // key,value 序列化
    properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

    // 设置事务 id (必须), 事务 id 任意起名
    properties.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,
"transaction_id_0");

    // 3. 创建 kafka 生产者对象
    KafkaProducer<String, String> kafkaProducer = new
KafkaProducer<String, String>(properties);

    // 初始化事务
    kafkaProducer.initTransactions();
    // 开启事务
    kafkaProducer.beginTransaction();
    try {
        // 4. 调用 send 方法, 发送消息
        for (int i = 0; i < 5; i++) {
            // 发送消息
            kafkaProducer.send(new ProducerRecord<>("first",
"atguigu " + i));
        }

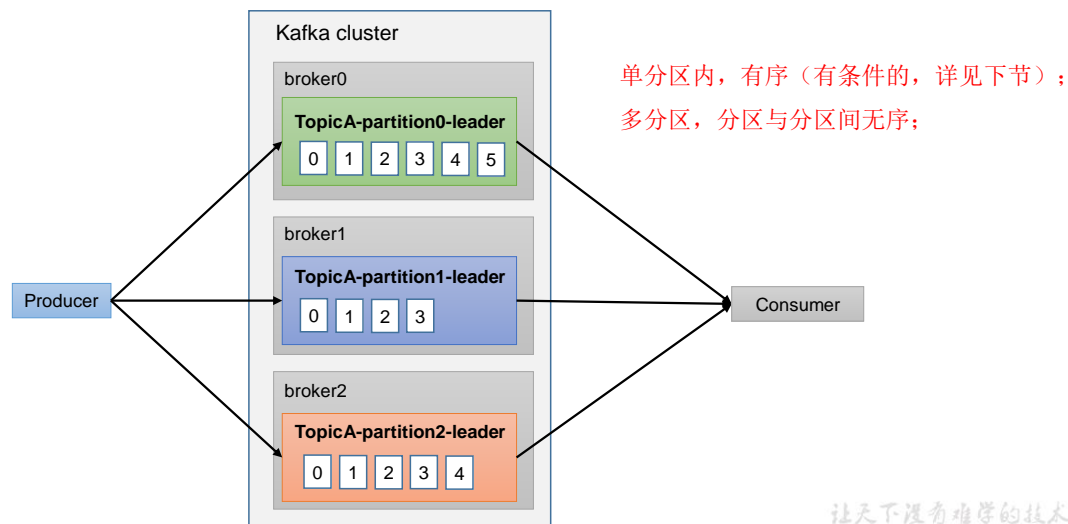
        //          int i = 1 / 0;

        // 提交事务
        kafkaProducer.commitTransaction();

    } catch (Exception e) {
        // 终止事务
        kafkaProducer.abortTransaction();
    } finally {
        // 5. 关闭资源
        kafkaProducer.close();
    }
}
```

3.8 生产经验——数据有序

生产经验——数据有序



3.9 生产经验——数据乱序

生产经验——数据乱序

1) kafka在1.x版本之前保证数据单分区有序，条件如下：

max.in.flight.requests.per.connection=1（不需要考虑是否开启幂等性）。

2) kafka在1.x及以后版本保证数据单分区有序，条件如下：

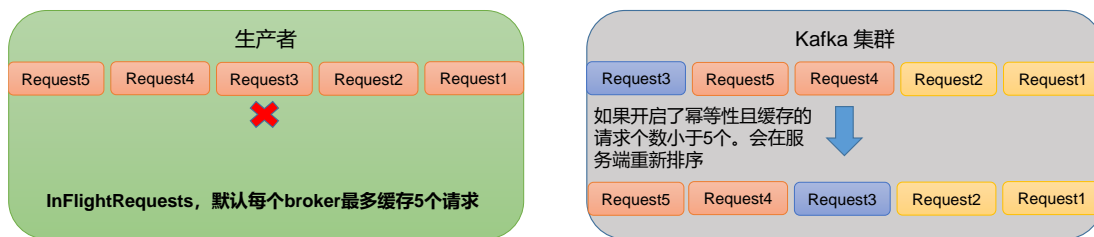
(1) 未开启幂等性

max.in.flight.requests.per.connection需要设置为1。

(2) 开启幂等性

max.in.flight.requests.per.connection需要设置小于等于5。

原因说明：因为在kafka1.x以后，启用幂等后，kafka服务端会缓存producer发来的最近5个request的元数据，故无论如何，都可以保证最近5个request的数据都是有序的。



第 4 章 Kafka Broker

4.1 Kafka Broker 工作流程

4.1.1 Zookeeper 存储的 Kafka 信息

(1) 启动 Zookeeper 客户端。

```
[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkCli.sh
```

(2) 通过 ls 命令可以查看 kafka 相关信息。

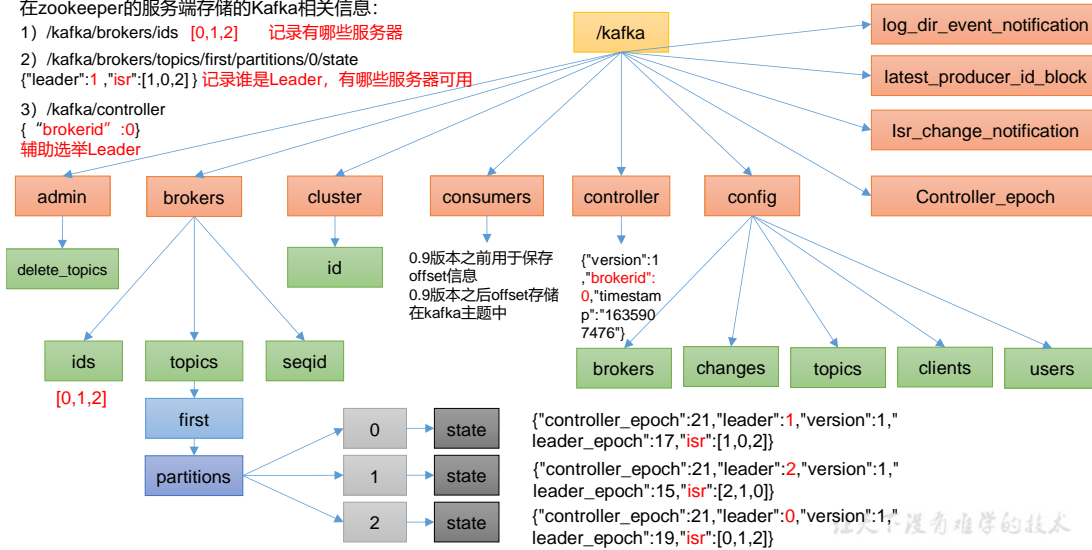
更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网


```
[zk: localhost:2181(CONNECTED) 2] ls /kafka
```

Zookeeper中存储的Kafka 信息

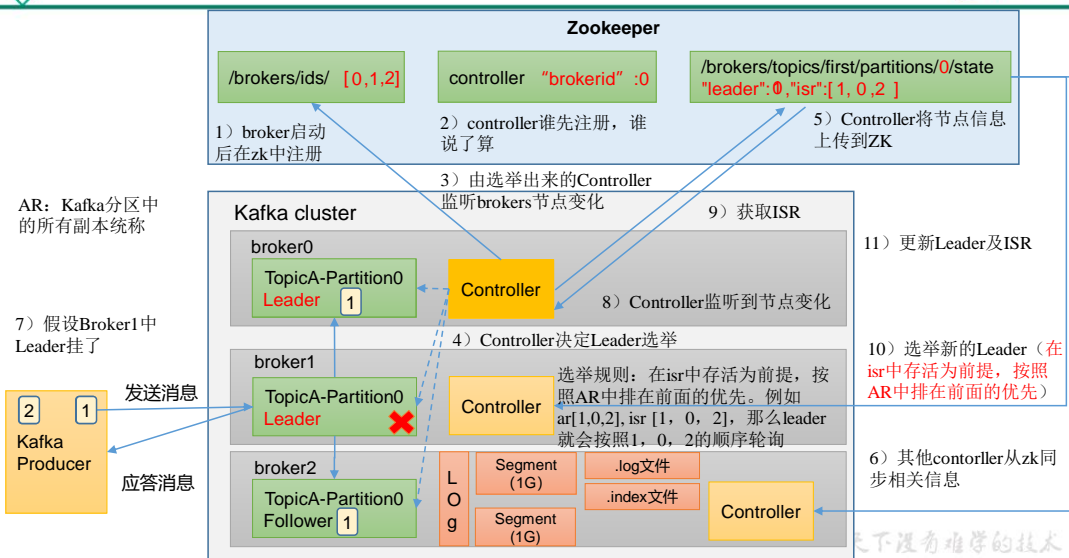
在zookeeper的服务端存储的Kafka相关信息:

- 1) /kafka/brokers/ids [0,1,2] 记录有哪些服务器
- 2) /kafka/brokers/topics/first/partitions/0/state {"leader":1,"isr":[1,0,2]} 记录谁是Leader, 有哪些服务器可用
- 3) /kafka/controller {"brokerid":0} 辅助选举Leader



4.1.2 Kafka Broker 总体工作流程

Kafka Broker总体工作流程



1) 模拟 Kafka 上下线，Zookeeper 中数据变化

- (1) 查看/kafka/brokers/ids 路径上的节点。

```
[zk: localhost:2181(CONNECTED) 2] ls /kafka/brokers/ids
```

```
[0, 1, 2]
```

- (2) 查看/kafka/controller 路径上的数据。

```
[zk: localhost:2181(CONNECTED) 15] get /kafka/controller
```

```
{"version":1,"brokerid":0,"timestamp":"1637292471777"}
```

- (3) 查看/kafka/brokers/topics/first/partitions/0/state 路径上的数据。

```
[zk: localhost:2181(CONNECTED) 16] get /kafka/brokers/topics/first/partitions/0/state
{"controller_epoch":24,"leader":0,"version":1,"leader_epoch":18,"isr":[0,1,2]}
```

(4) 停止 hadoop104 上的 kafka。

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

(5) 再次查看 /kafka/brokers/ids 路径上的节点。

```
[zk: localhost:2181(CONNECTED) 3] ls /kafka/brokers/ids
[0, 1]
```

(6) 再次查看 /kafka/controller 路径上的数据。

```
[zk: localhost:2181(CONNECTED) 15] get /kafka/controller
{"version":1,"brokerid":0,"timestamp":"1637292471777"}
```

(7) 再次查看 /kafka/brokers/topics/first/partitions/0/state 路径上的数据。

```
[zk: localhost:2181(CONNECTED) 16] get /kafka/brokers/topics/first/partitions/0/state
{"controller_epoch":24,"leader":0,"version":1,"leader_epoch":18,"isr":[0,1]}
```

(8) 启动 hadoop104 上的 kafka。

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

(9) 再次观察 (1)、(2)、(3) 步骤中的内容。

4.1.3 Broker 重要参数

参数名称	描述
replica.lag.time.max.ms	ISR 中，如果 Follower 长时间未向 Leader 发送通信请求或同步数据，则该 Follower 将被踢出 ISR。该时间阈值，默认 30s。
auto.leader.rebalance.enable	默认是 true。自动 Leader Partition 平衡。
leader.imbalance.per.broker.percentage	默认是 10%。每个 broker 允许的不平衡的 leader 的比率。如果每个 broker 超过了这个值，控制器会触发 leader 的平衡。
leader.imbalance.check.interval.seconds	默认值 300 秒。检查 leader 负载是否平衡的间隔时间。
log.segment.bytes	Kafka 中 log 日志是分成一块块存储的，此配置是指 log 日志划分成块的大小，默认值 1G。
log.index.interval.bytes	默认 4kb，kafka 里面每当写入了 4kb 大小的日志 (.log)，然后就往 index 文件里面记录一个索引。

log.retention.hours	Kafka 中数据保存的时间，默认 7 天。
log.retention.minutes	Kafka 中数据保存的时间，分钟级别，默认关闭。
log.retention.ms	Kafka 中数据保存的时间，毫秒级别，默认关闭。
log.retention.check.interval.ms	检查数据是否保存超时的间隔，默认是 5 分钟。
log.retention.bytes	默认等于 -1，表示无穷大。超过设置的所有日志总大小，删除最早的 segment。
log.cleanup.policy	默认是 delete，表示所有数据启用删除策略；如果设置值为 compact，表示所有数据启用压缩策略。
num.io.threads	默认是 8。负责写磁盘的线程数。整个参数值要占总核数的 50%。
num.replica.fetchers	副本拉取线程数，这个参数占总核数的 50% 的 1/3
num.network.threads	默认是 3。数据传输线程数，这个参数占总核数的 50% 的 2/3。
log.flush.interval.messages	强制页缓存刷写到磁盘的条数，默认是 long 的最大值，9223372036854775807。一般不建议修改，交给系统自己管理。
log.flush.interval.ms	每隔多久，刷数据到磁盘，默认是 null。一般不建议修改，交给系统自己管理。

4.2 生产经验——节点服役和退役

4.2.1 服役新节点

1) 新节点准备

(1) 关闭 hadoop104，并右键执行克隆操作。

(2) 开启 hadoop105，并修改 IP 地址。

```
[root@hadoop104 ~]# vim /etc/sysconfig/network-scripts/ifcfg-ens33

DEVICE=ens33
TYPE=Ethernet
ONBOOT=yes
BOOTPROTO=static
NAME="ens33"
IPADDR=192.168.10.105
PREFIX=24
GATEWAY=192.168.10.2
DNS1=192.168.10.2
```

(3) 在 hadoop105 上, 修改主机名称为 hadoop105。

```
[root@hadoop104 ~]# vim /etc/hostname  
hadoop105
```

(4) 重新启动 hadoop104、hadoop105。

(5) 修改 hadoop105 中 kafka 的 broker.id 为 3。

(6) 删除 hadoop105 中 kafka 下的 datas 和 logs。

```
[atguigu@hadoop105 kafka]$ rm -rf datas/* logs/*
```

(7) 启动 hadoop102、hadoop103、hadoop104 上的 kafka 集群。

```
[atguigu@hadoop102 ~]$ zk.sh start  
[atguigu@hadoop102 ~]$ kf.sh start
```

(8) 单独启动 hadoop105 中的 kafka。

```
[atguigu@hadoop105 kafka]$ bin/kafka-server-start.sh -  
daemon ./config/server.properties
```

2) 执行负载均衡操作

(1) 创建一个要均衡的主题。

```
[atguigu@hadoop102 kafka]$ vim topics-to-move.json  
  
{  
  "topics": [  
    {"topic": "first"}  
  ],  
  "version": 1  
}
```

(2) 生成一个负载均衡的计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --  
bootstrap-server hadoop102:9092 --topics-to-move-json-file  
topics-to-move.json --broker-list "0,1,2,3" --generate
```

```
Current partition replica assignment  
{  
  "version": 1, "partitions": [  
    {"topic": "first", "partition": 0, "replicas": [0, 2, 1], "log_dirs": ["any", "any", "any"]},  
    {"topic": "first", "partition": 1, "replicas": [2, 1, 0], "log_dirs": ["any", "any", "any"]},  
    {"topic": "first", "partition": 2, "replicas": [1, 0, 2], "log_dirs": ["any", "any", "any"]} ]  
}
```

Proposed partition reassignment configuration

```
{  
  "version": 1, "partitions": [  
    {"topic": "first", "partition": 0, "replicas": [2, 3, 0], "log_dirs": ["any", "any", "any"]},  
    {"topic": "first", "partition": 1, "replicas": [3, 0, 1], "log_dirs": ["any", "any", "any"]},  
    {"topic": "first", "partition": 2, "replicas": [0, 1, 2], "log_dirs": ["any", "any", "any"]} ]  
}
```

(3) 创建副本存储计划 (所有副本存储在 broker0、broker1、broker2、broker3 中)。

```
[atguigu@hadoop102 kafka]$ vim increase-replication-factor.json
```

输入如下内容:

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)

```
{ "version": 1, "partitions": [ { "topic": "first", "partition": 0, "replicas": [ 2, 3, 0 ], "log_dirs": [ "any", "any", "any" ] }, { "topic": "first", "partition": 1, "replicas": [ 3, 0, 1 ], "log_dirs": [ "any", "any", "any" ] }, { "topic": "first", "partition": 2, "replicas": [ 0, 1, 2 ], "log_dirs": [ "any", "any", "any" ] } ] }
```

(4) 执行副本存储计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --reassignment-json-file increase-replication-factor.json --execute
```

(5) 验证副本存储计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --reassignment-json-file increase-replication-factor.json --verify
```

```
Status of partition reassignment:
Reassignment of partition first-0 is complete.
Reassignment of partition first-1 is complete.
Reassignment of partition first-2 is complete.

Clearing broker-level throttles on brokers 0,1,2,3
Clearing topic-level throttles on topic first
```

4.2.2 退役旧节点

1) 执行负载均衡操作

先按照退役一台节点，**生成执行计划**，然后按照服役时操作流程**执行负载均衡**。

(1) 创建一个要均衡的主题。

```
[atguigu@hadoop102 kafka]$ vim topics-to-move.json

{
  "topics": [
    { "topic": "first" }
  ],
  "version": 1
}
```

(2) 创建执行计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --topics-to-move-json-file topics-to-move.json --broker-list "0,1,2" --generate
```

```
Current partition replica assignment
{"version":1,"partitions":[{"topic":"first","partition":0,"replicas": [2,0,1],"log_dirs":["any","any","any"]}, {"topic":"first","partition":1,"replicas": [3,1,2],"log_dirs":["any","any","any"]}, {"topic":"first","partition":2,"replicas": [0,2,3],"log_dirs":["any","any","any"]} ] }
```

```
Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"first","partition":0,"replicas": [2,0,1],"log_dirs":["any","any","any"]}, {"topic":"first","partition":1,"replicas": [3,1,2],"log_dirs":["any","any","any"]}, {"topic":"first","partition":2,"replicas": [0,2,3],"log_dirs":["any","any","any"]} ] }
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
as":[2,0,1],"log_dirs":["any","any","any"]},{ "topic":"first","partition":1,"replicas":[0,1,2],"log_dirs":["any","any","any"]},{ "topic":"first","partition":2,"replicas":[1,2,0],"log_dirs":["any","any","any"]}]}}
```

(3) 创建副本存储计划（所有副本存储在 broker0、broker1、broker2 中）。

```
[atguigu@hadoop102 kafka]$ vim increase-replication-factor.json

{"version":1,"partitions":[{"topic":"first","partition":0,"replicas":[2,0,1],"log_dirs":["any","any","any"]},{ "topic":"first","partition":1,"replicas":[0,1,2],"log_dirs":["any","any","any"]},{ "topic":"first","partition":2,"replicas":[1,2,0],"log_dirs":["any","any","any"]}]}}
```

(4) 执行副本存储计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --reassignment-json-file increase-replication-factor.json --execute
```

(5) 验证副本存储计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --reassignment-json-file increase-replication-factor.json --verify

Status of partition reassignment:
Reassignment of partition first-0 is complete.
Reassignment of partition first-1 is complete.
Reassignment of partition first-2 is complete.

Clearing broker-level throttles on brokers 0,1,2,3
Clearing topic-level throttles on topic first
```

2) 执行停止命令

在 hadoop105 上执行停止命令即可。

```
[atguigu@hadoop105 kafka]$ bin/kafka-server-stop.sh
```

4.3 Kafka 副本

4.3.1 副本基本信息

(1) Kafka 副本作用：提高数据可靠性。

(2) Kafka 默认副本 1 个，生产环境一般配置为 2 个，保证数据可靠性；太多副本会增加磁盘存储空间，增加网络上数据传输，降低效率。

(3) Kafka 中副本分为：Leader 和 Follower。Kafka 生产者只会把数据发往 Leader，然后 Follower 找 Leader 进行同步数据。

(4) Kafka 分区中的所有副本统称为 AR（Assigned Replicas）。

AR = ISR + OSR

ISR，表示和 Leader 保持同步的 Follower 集合。如果 Follower 长时间未向 Leader 发送

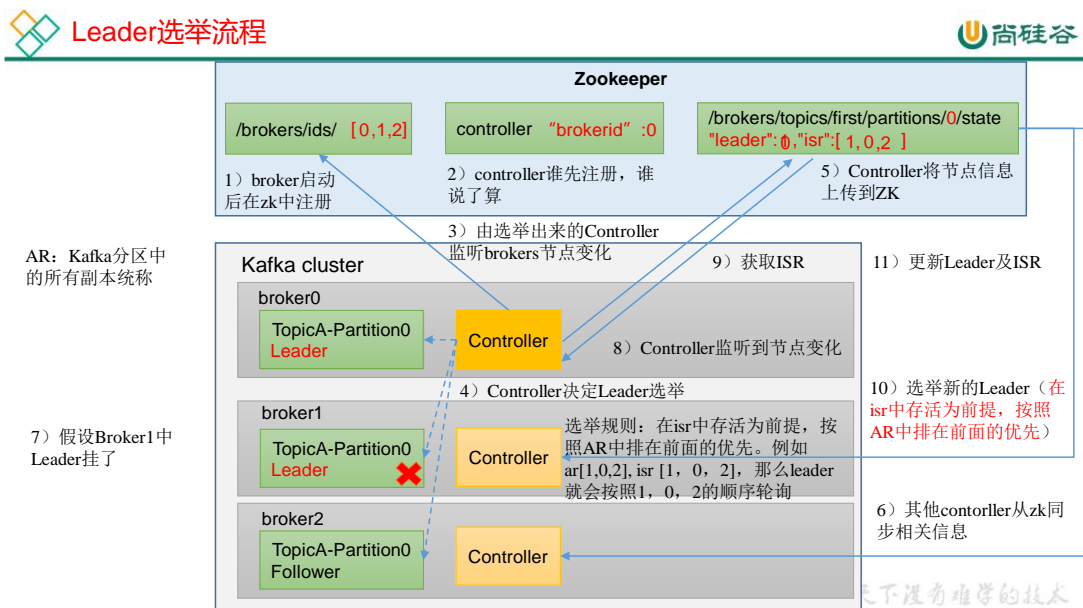
通信请求或同步数据，则该 Follower 将被踢出 ISR。该时间阈值由 `replica.lag.time.max.ms` 参数设定，默认 30s。Leader 发生故障之后，就会从 ISR 中选举新的 Leader。

OSR，表示 Follower 与 Leader 副本同步时，延迟过多的副本。

4.3.2 Leader 选举流程

Kafka 集群中有一个 broker 的 Controller 会被选举为 Controller Leader，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 的信息同步工作是依赖于 Zookeeper 的。



(1) 创建一个新的 topic，4 个分区，4 个副本

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server
hadoop102:9092 --create --topic atguigu1 --partitions 4 --replication-factor
4
Created topic atguigu1.
```

(2) 查看 Leader 分布情况

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe
--topic atguigu1
Topic: atguigu1 TopicId: awpgX_7WR-OX3Vl6HE8sVg PartitionCount: 4 ReplicationFactor: 4
Configs: segment.bytes=1073741824
Topic: atguigu1 Partition: 0 Leader: 3 Replicas: 3,0,2,1 Isr: 3,0,2,1
Topic: atguigu1 Partition: 1 Leader: 1 Replicas: 1,2,3,0 Isr: 1,2,3,0
Topic: atguigu1 Partition: 2 Leader: 0 Replicas: 0,3,1,2 Isr: 0,3,1,2
Topic: atguigu1 Partition: 3 Leader: 2 Replicas: 2,1,0,3 Isr: 2,1,0,3
```

(3) 停止掉 hadoop105 的 kafka 进程，并查看 Leader 分区情况

```
[atguigu@hadoop105 kafka]$ bin/kafka-server-stop.sh

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe
--topic atguigu1
Topic: atguigu1 TopicId: awpgX_7WR-OX3Vl6HE8sVg PartitionCount: 4 ReplicationFactor: 4
Configs: segment.bytes=1073741824
Topic: atguigu1 Partition: 0 Leader: 0 Replicas: 3,0,2,1 Isr: 0,2,1
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网


```
Topic: atguigu1 Partition: 1 Leader: 1 Replicas: 1,2,3,0 Isr: 1,2,0
Topic: atguigu1 Partition: 2 Leader: 0 Replicas: 0,3,1,2 Isr: 0,1,2
Topic: atguigu1 Partition: 3 Leader: 2 Replicas: 2,1,0,3 Isr: 2,1,0
```

(4) 停止掉 hadoop104 的 kafka 进程，并查看 Leader 分区情况

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

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic atguigu1
Topic: atguigu1 TopicId: awpgX_7WR-OX3Vl6HE8sVg PartitionCount: 4 ReplicationFactor: 4
Configs: segment.bytes=1073741824
Topic: atguigu1 Partition: 0 Leader: 0 Replicas: 3,0,2,1 Isr: 0,1
Topic: atguigu1 Partition: 1 Leader: 1 Replicas: 1,2,3,0 Isr: 1,0
Topic: atguigu1 Partition: 2 Leader: 0 Replicas: 0,3,1,2 Isr: 0,1
Topic: atguigu1 Partition: 3 Leader: 1 Replicas: 2,1,0,3 Isr: 1,0
```

(5) 启动 hadoop105 的 kafka 进程，并查看 Leader 分区情况

```
[atguigu@hadoop105 kafka]$ bin/kafka-server-start.sh -daemon config/server.properties

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic atguigu1
Topic: atguigu1 TopicId: awpgX_7WR-OX3Vl6HE8sVg PartitionCount: 4 ReplicationFactor: 4
Configs: segment.bytes=1073741824
Topic: atguigu1 Partition: 0 Leader: 0 Replicas: 3,0,2,1 Isr: 0,1,3
Topic: atguigu1 Partition: 1 Leader: 1 Replicas: 1,2,3,0 Isr: 1,0,3
Topic: atguigu1 Partition: 2 Leader: 0 Replicas: 0,3,1,2 Isr: 0,1,3
Topic: atguigu1 Partition: 3 Leader: 1 Replicas: 2,1,0,3 Isr: 1,0,3
```

(6) 启动 hadoop104 的 kafka 进程，并查看 Leader 分区情况

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

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic atguigu1
Topic: atguigu1 TopicId: awpgX_7WR-OX3Vl6HE8sVg PartitionCount: 4 ReplicationFactor: 4
Configs: segment.bytes=1073741824
Topic: atguigu1 Partition: 0 Leader: 0 Replicas: 3,0,2,1 Isr: 0,1,3,2
Topic: atguigu1 Partition: 1 Leader: 1 Replicas: 1,2,3,0 Isr: 1,0,3,2
Topic: atguigu1 Partition: 2 Leader: 0 Replicas: 0,3,1,2 Isr: 0,1,3,2
Topic: atguigu1 Partition: 3 Leader: 1 Replicas: 2,1,0,3 Isr: 1,0,3,2
```

(7) 停止掉 hadoop103 的 kafka 进程，并查看 Leader 分区情况

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

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic atguigu1
Topic: atguigu1 TopicId: awpgX_7WR-OX3Vl6HE8sVg PartitionCount: 4 ReplicationFactor: 4
Configs: segment.bytes=1073741824
Topic: atguigu1 Partition: 0 Leader: 0 Replicas: 3,0,2,1 Isr: 0,3,2
Topic: atguigu1 Partition: 1 Leader: 2 Replicas: 1,2,3,0 Isr: 0,3,2
Topic: atguigu1 Partition: 2 Leader: 0 Replicas: 0,3,1,2 Isr: 0,3,2
Topic: atguigu1 Partition: 3 Leader: 2 Replicas: 2,1,0,3 Isr: 0,3,2
```

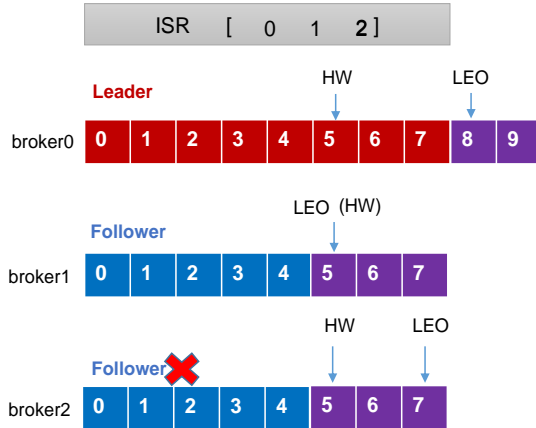

4.3.3 Leader 和 Follower 故障处理细节

Follower故障处理细节



LEO (Log End Offset) : 每个副本的最后一个offset, LEO其实就是最新的offset + 1。

HW (High Watermark) : 所有副本中最小的LEO。



1) Follower故障

- (1) Follower发生故障后会被临时踢出ISR
- (2) 这个期间Leader和Follower继续接收数据
- (3) 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。
- (4) 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。

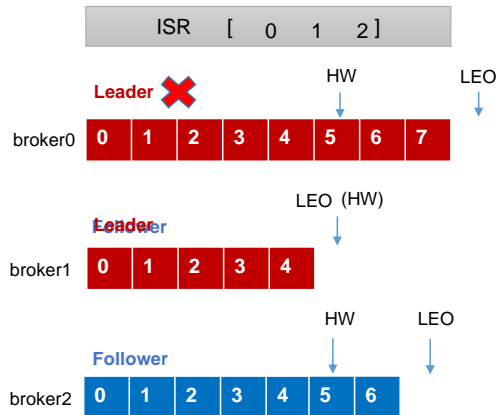
让天下没有难学的技术

Leader故障处理细节



LEO (Log End Offset) : 每个副本的最后一个offset, LEO其实就是最新的offset + 1

HW (High Watermark) : 所有副本中最小的LEO



1) Leader故障

- (1) Leader发生故障之后, 会从ISR中选出一个新的Leader
- (2) 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。

让天下没有难学的技术

4.3.4 分区副本分配

如果 kafka 服务器只有 4 个节点, 那么设置 kafka 的分区数大于服务器台数, 在 kafka 底层如何分配存储副本呢?

1) 创建 16 分区, 3 个副本

- (1) 创建一个新的 topic, 名称为 second。

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --create --partitions 16 --replication-factor 3 --topic second
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载, 可百度访问: 尚硅谷官网

(2) 查看分区和副本情况。

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server
hadoop102:9092 --describe --topic second
```

```
Topic: second4 Partition: 0 Leader: 0 Replicas: 0,1,2 Isr: 0,1,2
Topic: second4 Partition: 1 Leader: 1 Replicas: 1,2,3 Isr: 1,2,3
Topic: second4 Partition: 2 Leader: 2 Replicas: 2,3,0 Isr: 2,3,0
Topic: second4 Partition: 3 Leader: 3 Replicas: 3,0,1 Isr: 3,0,1

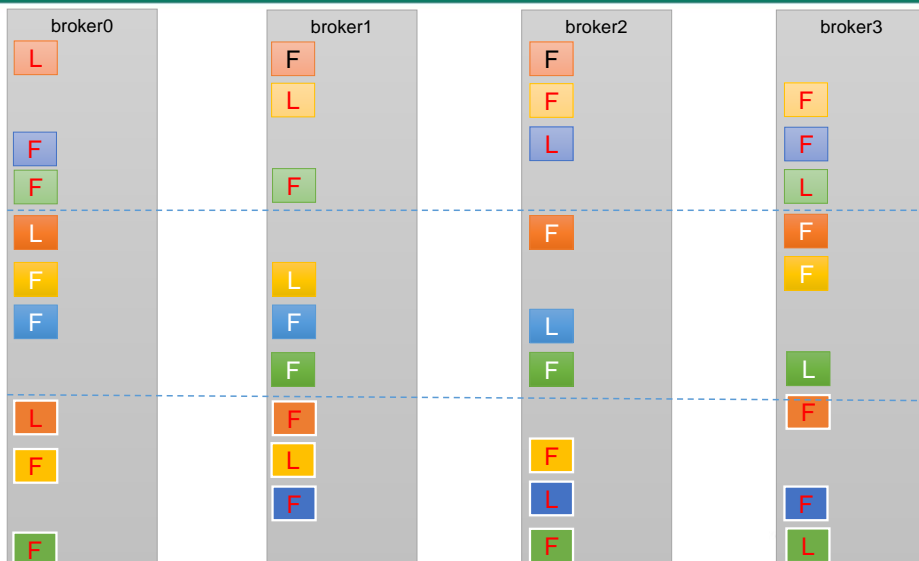
Topic: second4 Partition: 4 Leader: 0 Replicas: 0,2,3 Isr: 0,2,3
Topic: second4 Partition: 5 Leader: 1 Replicas: 1,3,0 Isr: 1,3,0
Topic: second4 Partition: 6 Leader: 2 Replicas: 2,0,1 Isr: 2,0,1
Topic: second4 Partition: 7 Leader: 3 Replicas: 3,1,2 Isr: 3,1,2

Topic: second4 Partition: 8 Leader: 0 Replicas: 0,3,1 Isr: 0,3,1
Topic: second4 Partition: 9 Leader: 1 Replicas: 1,0,2 Isr: 1,0,2
Topic: second4 Partition: 10 Leader: 2 Replicas: 2,1,3 Isr: 2,1,3
Topic: second4 Partition: 11 Leader: 3 Replicas: 3,2,0 Isr: 3,2,0

Topic: second4 Partition: 12 Leader: 0 Replicas: 0,1,2 Isr: 0,1,2
Topic: second4 Partition: 13 Leader: 1 Replicas: 1,2,3 Isr: 1,2,3
Topic: second4 Partition: 14 Leader: 2 Replicas: 2,3,0 Isr: 2,3,0
Topic: second4 Partition: 15 Leader: 3 Replicas: 3,0,1 Isr: 3,0,1
```



分区副本分配



4.3.5 生产经验——手动调整分区副本存储

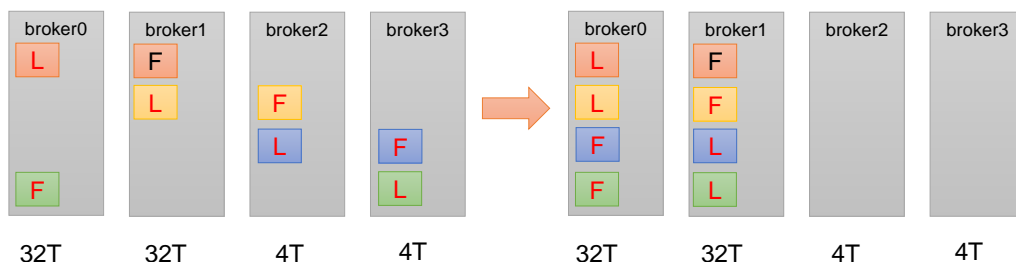


生产经验——手动调整分区副本存储



在生产环境中，每台服务器的配置和性能不一致，但是Kafka只会根据自己的代码规则创建对应的分区副本，就会导致个别服务器存储压力较大。所有需要手动调整分区副本的存储。

需求：创建一个新的topic，4个分区，两个副本，名称为three。将该topic的所有副本都存储到broker0和broker1两台服务器上。



让天下没有难学的技术

手动调整分区副本存储的步骤如下：

(1) 创建一个新的 topic，名称为 three。

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --create --partitions 4 --replication-factor 2 --topic three
```

(2) 查看分区副本存储情况。

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --describe --topic three
```

(3) 创建副本存储计划（所有副本都指定存储在 broker0、broker1 中）。

```
[atguigu@hadoop102 kafka]$ vim increase-replication-factor.json
```

输入如下内容：

```
{
  "version":1,
  "partitions":[{"topic":"three","partition":0,"replicas":[0,1]},
               {"topic":"three","partition":1,"replicas":[0,1]},
               {"topic":"three","partition":2,"replicas":[1,0]},
               {"topic":"three","partition":3,"replicas":[1,0]}]
}
```

(4) 执行副本存储计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --reassignment-json-file increase-replication-factor.json --execute
```

(5) 验证副本存储计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --reassignment-json-file increase-replication-factor.json --verify
```

(6) 查看分区副本存储情况。

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server
hadoop102:9092 --describe --topic three
```

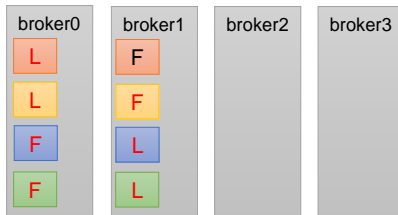
4.3.6 生产经验——Leader Partition 负载均衡



Leader Partition自动平衡



正常情况下，Kafka本身会自动把Leader Partition均匀分散在各个机器上，来保证每台机器的读写吞吐量都是均匀的。但是如果某些broker宕机，会导致Leader Partition过于集中在其他少部分几台broker上，这会导致少数几台broker的读写请求压力过高，其他宕机的broker重启之后都是follower partition，读写请求很低，造成集群负载不均衡。



- auto.leader.rebalance.enable, 默认是true。自动Leader Partition 平衡
- leader.imbalance.per.broker.percentage, 默认是10%。每个broker允许的不平衡的leader的比率。如果每个broker超过了这个值，控制器会触发leader的平衡。
- leader.imbalance.check.interval.seconds, 默认值300秒。检查leader负载是否平衡的间隔时间。

下面拿一个主题举例说明，假设集群只有一个主题如下图所示：

```
Topic: atguigu1 Partition: 0 Leader: 0 Replicas: 3,0,2,1 Isr: 3,0,2,1
Topic: atguigu1 Partition: 1 Leader: 1 Replicas: 1,2,3,0 Isr: 1,2,3,0
Topic: atguigu1 Partition: 2 Leader: 2 Replicas: 0,3,1,2 Isr: 0,3,1,2
Topic: atguigu1 Partition: 3 Leader: 3 Replicas: 2,1,0,3 Isr: 2,1,0,3
```

针对broker0节点，分区2的AR优先副本是0节点，但是0节点却不是Leader节点，所以不平衡数加1，AR副本总数是4，所以broker0节点不平衡率为1/4>10%，需要再平衡。

broker2和broker3节点和broker0不平衡率一样，需要再平衡。
Broker1的不平衡数为0，不需要再平衡

让天下没有难学的技术

自动 leader partitions 负载均衡推荐设置为 false，因为上例中实际 leader 就是 0 1 2 3，不需要改变，但按照算法需要改变，或者把比率上调

参数名称	描述
auto.leader.rebalance.enable	默认是 true。自动 Leader Partition 平衡。生产环境中，leader 重选举的代价比较大，可能会带来性能影响，建议设置为 false 关闭。
leader.imbalance.per.broker.percentage	默认是 10%。每个 broker 允许的不平衡的 leader 的比率。如果每个 broker 超过了这个值，控制器会触发 leader 的平衡。
leader.imbalance.check.interval.seconds	默认值 300 秒。检查 leader 负载是否平衡的间隔时间。

4.3.7 生产经验——增加副本因子

在生产环境当中，由于某个主题的重要等级需要提升，我们考虑增加副本。副本数的增加需要先制定计划，然后根据计划执行。

1) 创建 topic

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server
hadoop102:9092 --create --partitions 3 --replication-factor 1 --
topic four
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

2) 手动增加副本存储

(1) 创建副本存储计划（所有副本都指定存储在 broker0、broker1、broker2 中）。

```
[atguigu@hadoop102 kafka]$ vim increase-replication-factor.json
```

输入如下内容：

```
{"version":1,"partitions":[{"topic":"four","partition":0,"replicas":[0,1,2]},{ "topic":"four","partition":1,"replicas":[0,1,2]},{ "topic":"four","partition":2,"replicas":[0,1,2]}]}
```

(2) 执行副本存储计划。

```
[atguigu@hadoop102 kafka]$ bin/kafka-reassign-partitions.sh --bootstrap-server hadoop102:9092 --reassignment-json-file increase-replication-factor.json --execute
```

4.4 文件存储

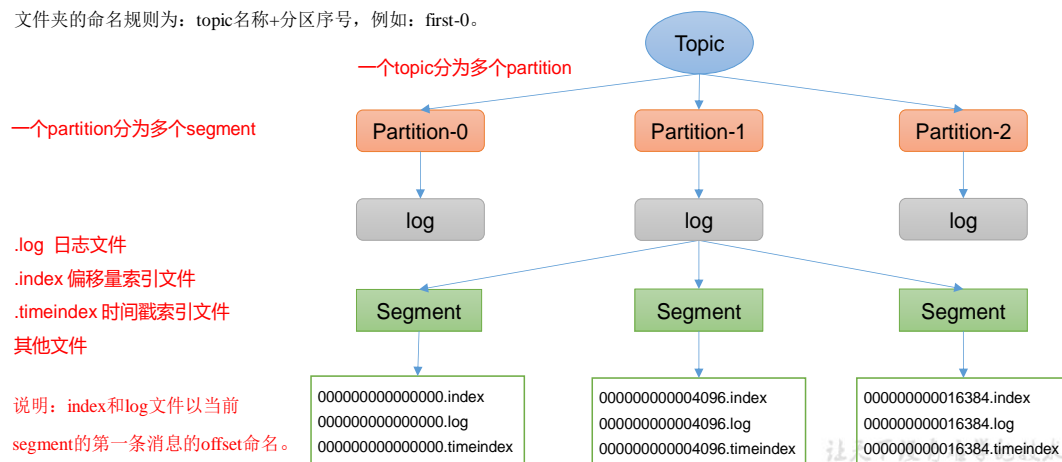
4.4.1 文件存储机制

1) Topic 数据的存储机制

Kafka文件存储机制



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，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segment包括：“.index”文件、“.log”文件和“.timeindex”等文件。这些文件位于一个文件夹下，该文件夹的命名规则为：topic名称+分区序号，例如：first-0。



2) 思考：Topic 数据到底存储在什么位置？

(1) 启动生产者，并发送消息。

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

(2) 查看 hadoop102（或者 hadoop103、hadoop104）的/opt/module/kafka/datas/**first-1**

(first-0、first-2) 路径上的文件。

```
[atguigu@hadoop104 first-1]$ ls
00000000000000000092.index
00000000000000000092.log
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

```
000000000000000000092.snapshot
000000000000000000092.timeindex
leader-epoch-checkpoint
partition.metadata
```

(3) 直接查看 log 日志，发现是乱码。

```
[atguigu@hadoop104 first-1]$ cat 000000000000000000092.log
\CYnF|©|©yhello world
```

(4) 通过工具查看 index 和 log 信息。

```
[atguigu@hadoop104 first-1]$ kafka-run-class.sh kafka.tools.DumpLogSegments
--files ./00000000000000000000.index
```

```
Dumping ./00000000000000000000.index
offset: 3 position: 152
```

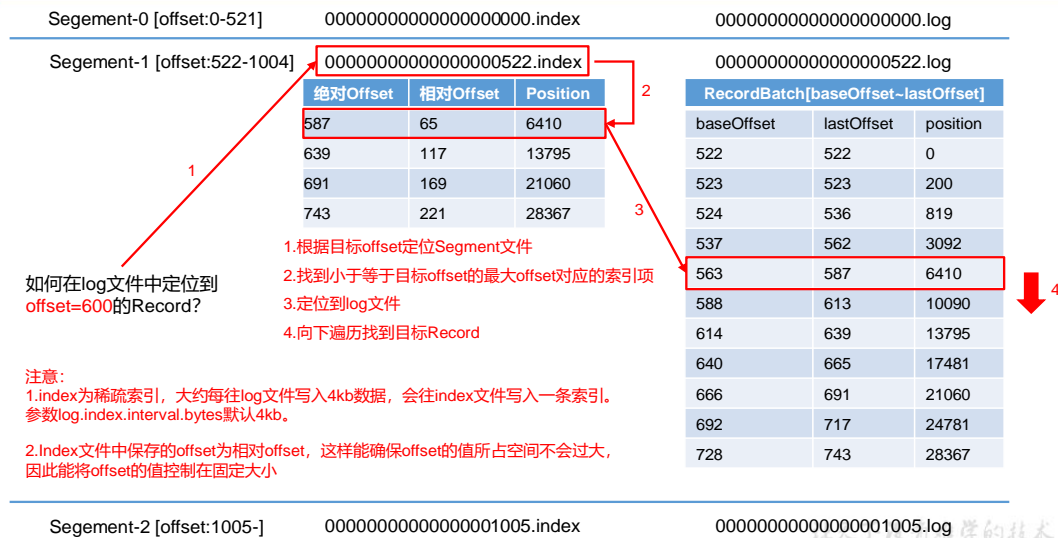
```
[atguigu@hadoop104 first-1]$ kafka-run-class.sh kafka.tools.DumpLogSegments
--files ./00000000000000000000.log
```

```
Dumping datas/first-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 1 count: 2 baseSequence: -1 lastSequence: -1 producerId: -1
producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position:
0 CreateTime: 1636338440962 size: 75 magic: 2 compresscodec: none crc: 2745337109 invalid:
true
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1 producerId: -1
producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position:
75 CreateTime: 1636351749089 size: 77 magic: 2 compresscodec: none crc: 273943004 invalid:
true
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1 producerId: -1
producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position:
152 CreateTime: 1636351749119 size: 77 magic: 2 compresscodec: none crc: 106207379 invalid:
true
baseOffset: 4 lastOffset: 8 count: 5 baseSequence: -1 lastSequence: -1 producerId: -1
producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position:
229 CreateTime: 1636353061435 size: 141 magic: 2 compresscodec: none crc: 157376877 invalid:
true
baseOffset: 9 lastOffset: 13 count: 5 baseSequence: -1 lastSequence: -1 producerId: -1
producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position:
370 CreateTime: 1636353204051 size: 146 magic: 2 compresscodec: none crc: 4058582827 invalid:
true
```

3) index 文件和 log 文件详解



Log文件和Index文件详解



说明：日志存储参数配置

参数	描述
log.segment.bytes	Kafka 中 log 日志是分成一块块存储的，此配置是指 log 日志划分成块的大小，默认值 1G。
log.index.interval.bytes	默认 4kb，kafka 里面每当写入了 4kb 大小的日志 (.log)，然后就往 index 文件里面记录一个索引。稀疏索引。

4.4.2 文件清理策略

Kafka 中默认的日志保存时间为 7 天，可以通过调整如下参数修改保存时间。

- log.retention.hours，最低优先级小时，默认 7 天。
- log.retention.minutes，分钟。
- log.retention.ms，最高优先级毫秒。
- log.retention.check.interval.ms，负责设置检查周期，默认 5 分钟。

那么日志一旦超过了设置的时间，怎么处理呢？

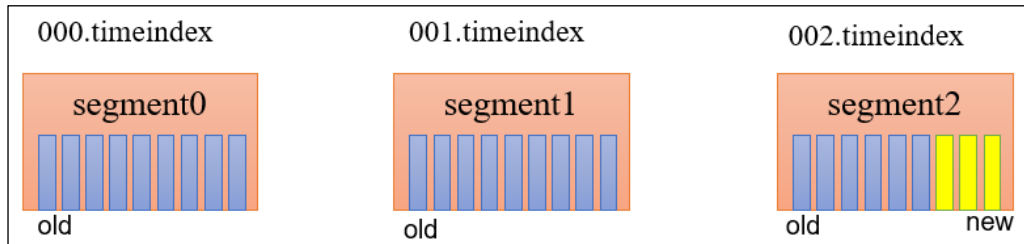
Kafka 中提供的日志清理策略有 delete 和 compact 两种。

1) delete 日志删除：将过期数据删除

- log.cleanup.policy = delete 所有数据启用删除策略
- (1) 基于时间：默认打开。以 segment 中所有记录中的最大时间戳作为该文件时间戳。
- (2) 基于大小：默认关闭。超过设置的所有日志总大小，删除最早的 segment。

log.retention.bytes，默认等于-1，表示无穷大。

思考：如果一个 segment 中有一部分数据过期，一部分没有过期，怎么处理？



2) compact 日志压缩



compact 日志压缩



compact 日志压缩：对于相同key的不同value值，只保留最后一个版本。

- log.cleanup.policy = compact 所有数据启用压缩策略

Offset	0	1	2	3	4	5	6	7	8
key	K1	K2	K1	K1	K3	K4	K5	K5	K2
value	V1	V2	V3	V4	V5	V6	V7	V8	V9

压缩之前的数据

Offset	3	4	5	7	8
keys	k1	K3	K4	K5	K2
values	V4	V5	V6	V8	V9

压缩之后的数据

压缩后的offset可能是不连续的，比如上图中没有6，当从这些offset消费消息时，将会拿到比这个offset大的offset对应的消息，实际上会拿到offset为7的消息，并从这个位置开始消费。

这种策略只适合特殊场景，比如消息的key是用户ID，value是用户的资料，通过这种压缩策略，整个消息集里就保存了所有用户最新的资料。

让天下没有难学的技术

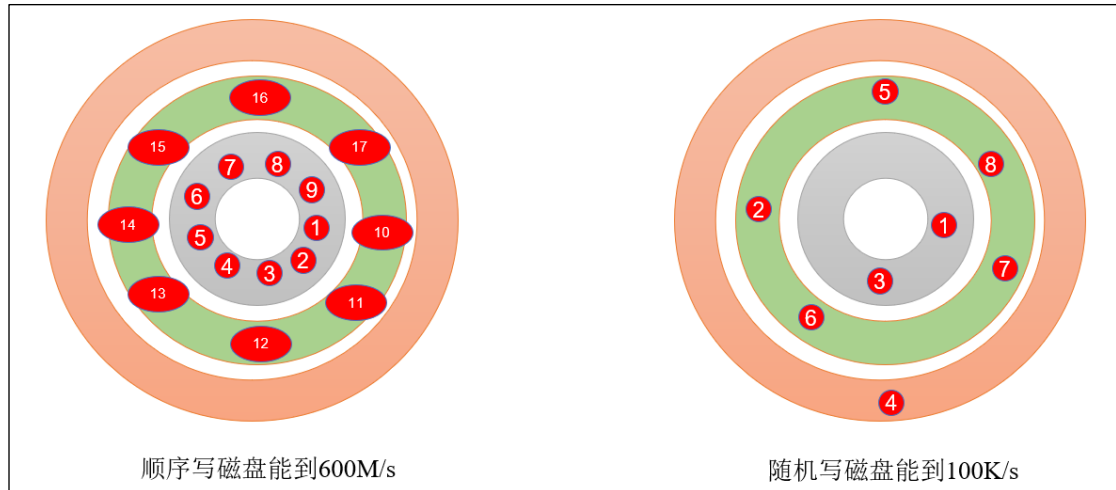
4.5 高效读写数据

1) Kafka 本身是分布式集群，可以采用分区技术，并行度高

2) 读数据采用稀疏索引，可以快速定位要消费的数据

3) 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。



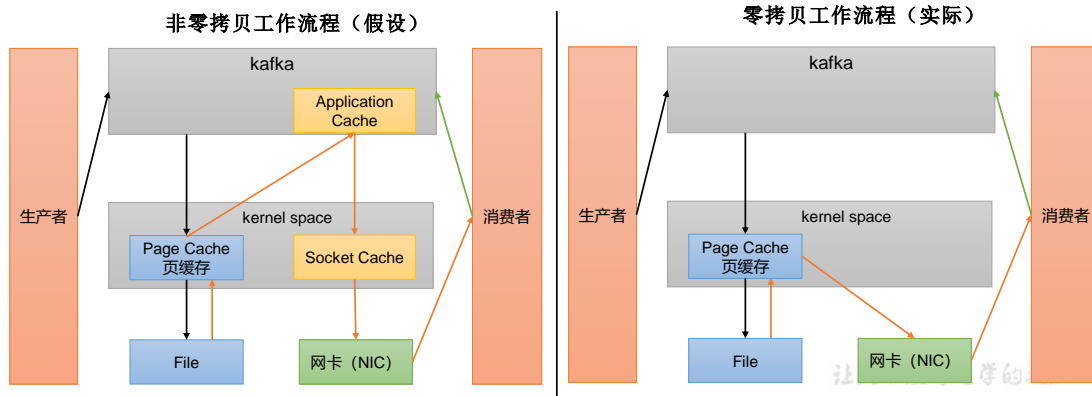
4) 页缓存 + 零拷贝技术



页缓存 + 零拷贝技术

零拷贝：Kafka的数据加工处理操作交由Kafka生产者和Kafka消费者处理。Kafka Broker应用层不关心存储的数据，所以就不用走应用层，传输效率高。

PageCache页缓存：Kafka重度依赖底层操作系统提供的PageCache功能。当上层有写操作时，操作系统只是将数据写入PageCache。当读操作发生时，先从PageCache中查找，如果找不到，再去磁盘中读取。实际上PageCache是把尽可能多的空闲内存都当做了磁盘缓存来使用。



参数	描述
log.flush.interval.messages	强制页缓存刷写到磁盘的条数，默认是 long 的最大值，9223372036854775807。一般不建议修改，交给系统自己管理。
log.flush.interval.ms	每隔多久，刷数据到磁盘，默认是 null。一般不建议修改，交给系统自己管理。

第 5 章 Kafka 消费者

5.1 Kafka 消费方式

Kafka 消费方式

➤ pull（拉）模式：

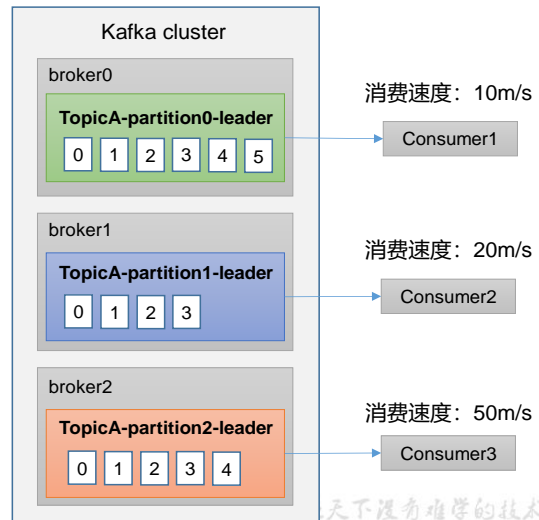
consumer采用从broker中主动拉取数据。

Kafka采用这种方式。

➤ push（推）模式：

Kafka没有采用这种方式，因为由broker决定消息发送速率，很难适应所有消费者的消费速率。例如推送的速度是50m/s，Consumer1、Consumer2就来不及处理消息。

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。

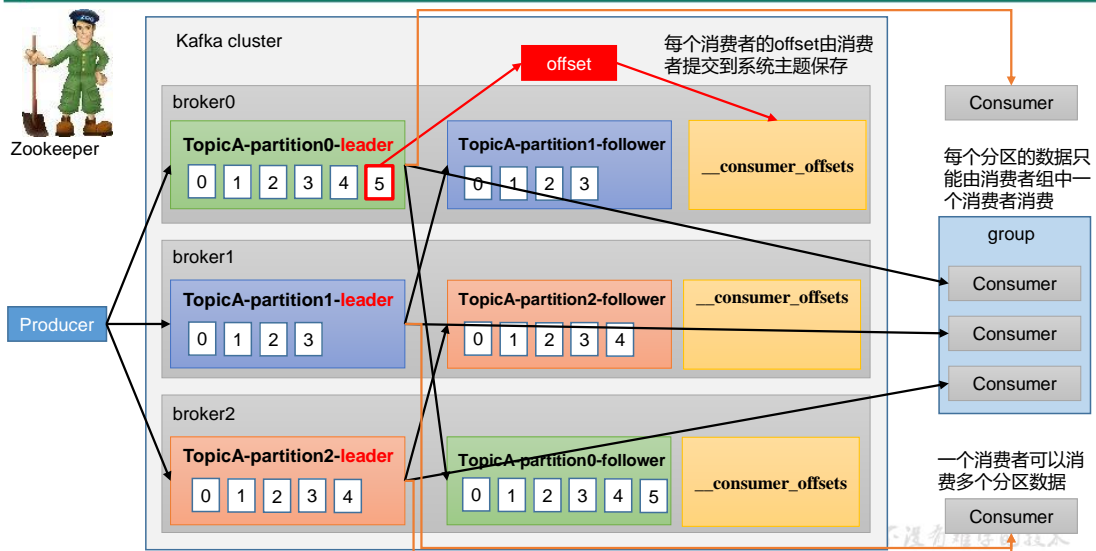


天下没有难学的技术

5.2 Kafka 消费者工作流程

5.2.1 消费者总体工作流程

Kafka 消费者总体工作流程



天下没有难学的技术

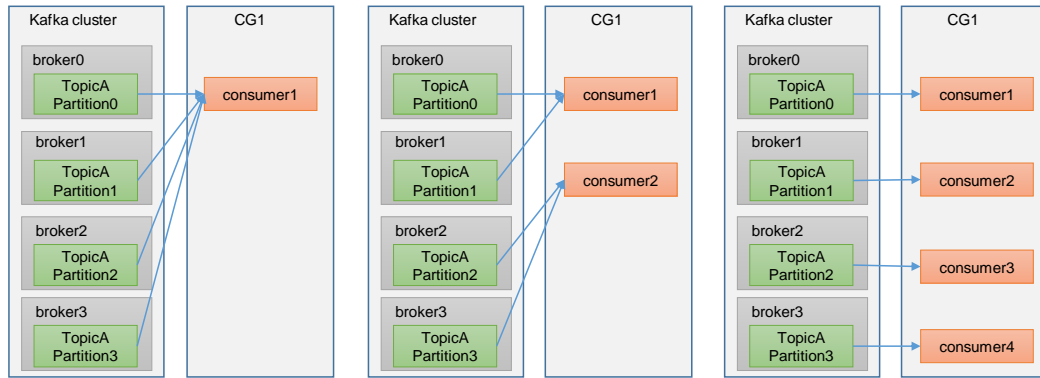
5.2.2 消费者组原理

消费者组



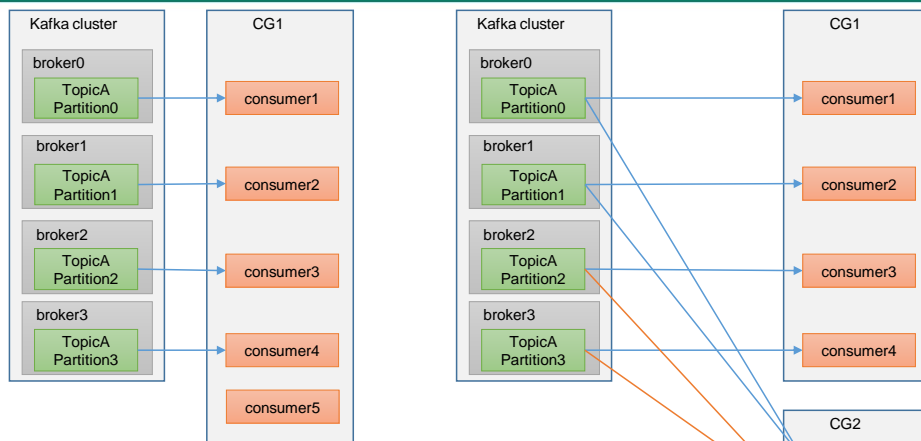
Consumer Group (CG)：消费者组，由多个consumer组成。形成一个消费者组的条件，是所有消费者的groupid相同。

- 消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费。
- 消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。



让天下没有难学的技术

消费者组



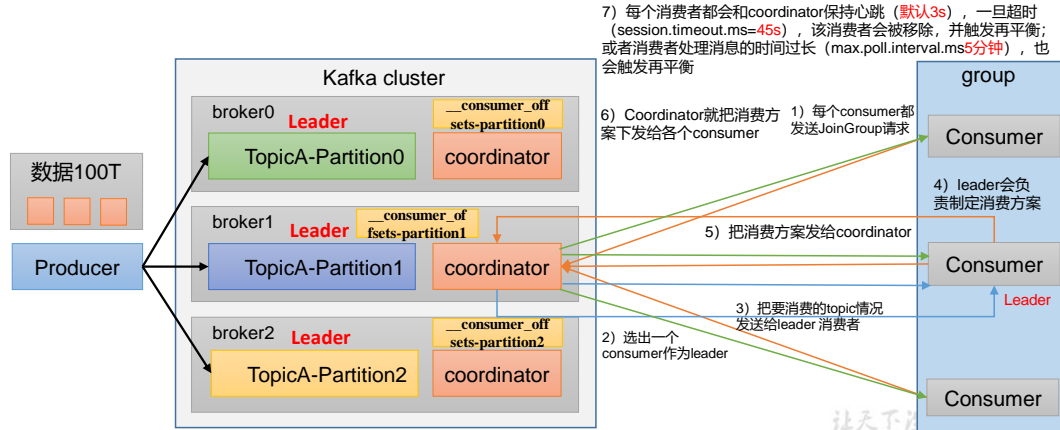
- 如果向消费组中添加更多的消费者，超过主题分区数量，则有一部分消费者就会闲置，不会接收任何消息。

- 消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

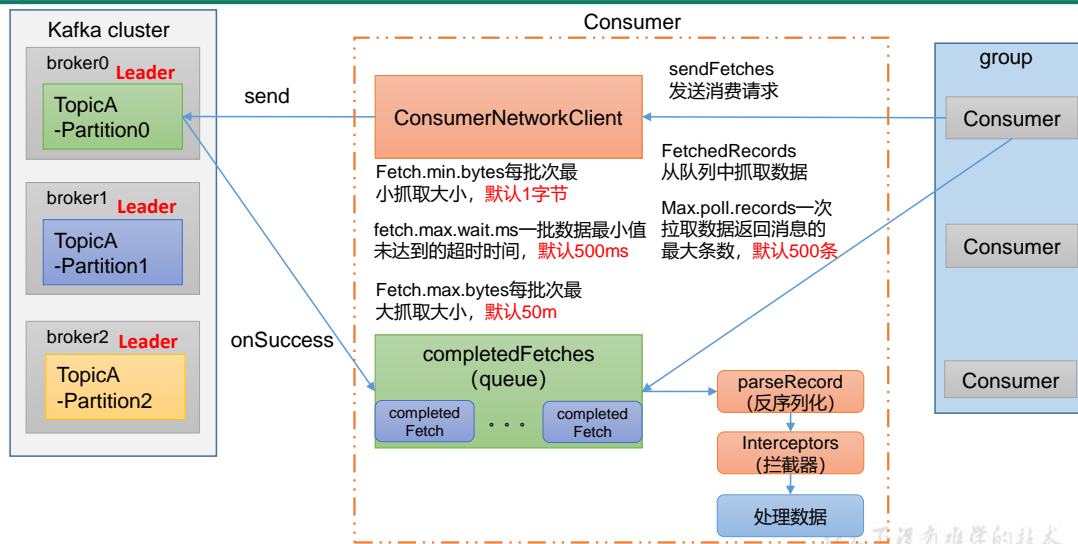
让天下没有难学的技术

消费者组初始化流程

- 1、coordinator: 辅助实现消费者组的初始化和分区的分配。
coordinator节点选择 = $\text{groupid} \text{的} \text{hashCode值} \% 50$ ($__\text{consumer_offsets}$ 的分区数量)
例如: $\text{groupid} \text{的} \text{hashCode值} = 1$, $1 \% 50 = 1$, 那么 $__\text{consumer_offsets}$ 主题的1号分区, 在哪个broker上, 就选择这个节点的coordinator 作为这个消费者组的老大。消费者组下的所有的消费者提交offset的时候就往这个分区去提交offset。



消费者组详细消费流程



5.2.3 消费者重要参数

参数名称	描述
bootstrap.servers	向 Kafka 集群建立初始连接用到的 host/port 列表。
key.deserializer 和 value.deserializer	指定接收消息的 key 和 value 的反序列化类型。一定要写全类名。
group.id	标记消费者所属的消费者组。
enable.auto.commit	默认值为 true ，消费者会自动周期性地向服务器提交偏移量。
auto.commit.interval.ms	如果设置了 enable.auto.commit 的值为 true，则该值定义了

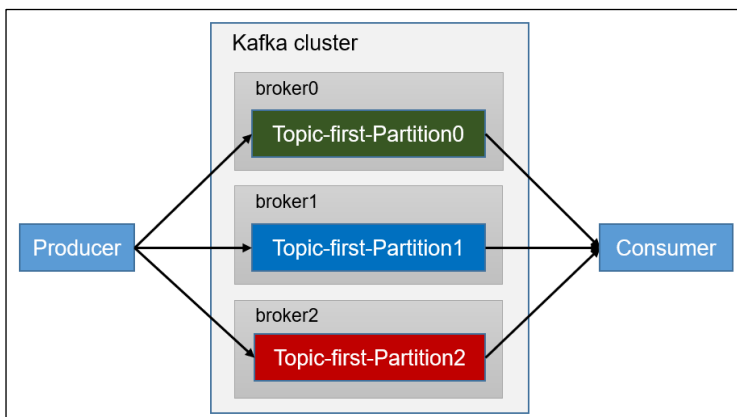
	消费者偏移量向 Kafka 提交的频率，默认 5s。
auto.offset.reset	当 Kafka 中没有初始偏移量或当前偏移量在服务器中不存在（如，数据被删除了），该如何处理？ earliest: 自动重置偏移量到最早的偏移量。 latest: 默认，自动重置偏移量为最新的偏移量。 none: 如果消费组原来的（previous）偏移量不存在，则向消费者抛异常。 anything: 向消费者抛异常。
offsets.topic.num.partitions	__consumer_offsets 的分区数，默认是 50 个分区。
heartbeat.interval.ms	Kafka 消费者和 coordinator 之间的心跳时间，默认 3s。 该条目的值必须小于 session.timeout.ms，也不应该高于 session.timeout.ms 的 1/3。
session.timeout.ms	Kafka 消费者和 coordinator 之间连接超时时间，默认 45s。 超过该值，该消费者被移除，消费者组执行再平衡。
max.poll.interval.ms	消费者处理消息的最大时长，默认是 5 分钟。超过该值，该消费者被移除，消费者组执行再平衡。
fetch.min.bytes	默认 1 个字节。消费者获取服务器端一批消息最小的字节数。
fetch.max.wait.ms	默认 500ms。如果没有从服务器端获取到一批数据的最小字节数。该时间到，仍然会返回数据。
fetch.max.bytes	默认 Default: 52428800 (50 m)。消费者获取服务器端一批消息最大的字节数。如果服务器端一批次的数据大于该值（50m）仍然可以拉取回来这批数据，因此，这不是一个绝对最大值。一批次的大小受 message.max.bytes（broker config）or max.message.bytes（topic config）影响。
max.poll.records	一次 poll 拉取数据返回消息的最大条数，默认是 500 条。

5.3 消费者 API

5.3.1 独立消费者案例（订阅主题）

1) 需求：

创建一个独立消费者，消费 first 主题中数据。



注意：在消费者 API 代码中必须配置消费者组 id。命令行启动消费者不填写消费者组 id 会被自动填写随机的消费者组 id。

2) 实现步骤

(1) 创建包名：com.atguigu.kafka.consumer

(2) 编写代码

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

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

import java.time.Duration;
import java.util.ArrayList;
import java.util.Properties;

public class CustomConsumer {

    public static void main(String[] args) {

        // 1.创建消费者的配置对象
        Properties properties = new Properties();

        // 2.给消费者配置对象添加参数
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // 配置序列化 必须
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());

        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());

        // 配置消费者组（组名任意起名） 必须
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
```

```
// 创建消费者对象
KafkaConsumer<String, String> kafkaConsumer = new
KafkaConsumer<String, String>(properties);

// 注册要消费的主题（可以消费多个主题）
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);

// 拉取数据打印
while (true) {
    // 设置 1s 中消费一批数据
    ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));

    // 打印消费到的数据
    for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
        System.out.println(consumerRecord);
    }
}
}
```

3) 测试

(1) 在 IDEA 中执行消费者程序。

(2) 在 Kafka 集群控制台，创建 Kafka 生产者，并输入数据。

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

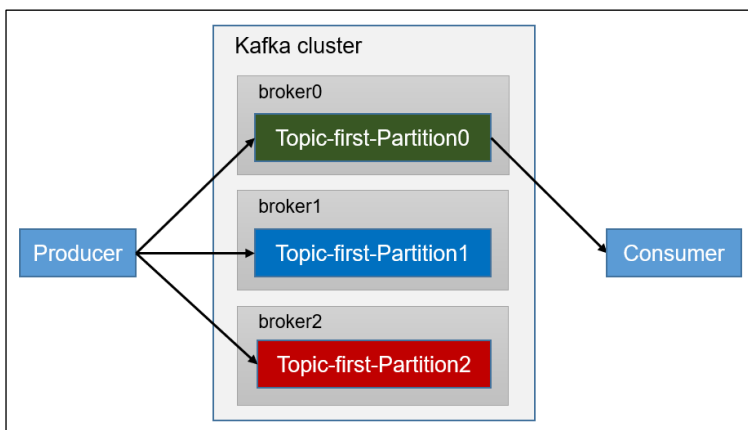
>hello
```

(3) 在 IDEA 控制台观察接收到的数据。

```
ConsumerRecord(topic = first, partition = 1, leaderEpoch = 3,
offset = 0, CreateTime = 1629160841112, serialized key size = -1,
serialized value size = 5, headers = RecordHeaders(headers = [],
isReadOnly = false), key = null, value = hello)
```

5.3.2 独立消费者案例（订阅分区）

1) 需求：创建一个独立消费者，消费 first 主题 0 号分区的数据。



2) 实现步骤

(1) 代码编写。

```

package com.atguigu.kafka.consumer;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;

import java.time.Duration;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Properties;

public class CustomConsumerPartition {

    public static void main(String[] args) {

        Properties properties = new Properties();

        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "hadoop102:9092");

        // 配置序列化 必须
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

        // 配置消费者组（必须），名字可以任意起
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");

        KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>(properties);

        // 消费某个主题的某个分区数据
        ArrayList<TopicPartition> topicPartitions = new

```



```
ArrayList<>();
    topicPartitions.add(new TopicPartition("first", 0));
    kafkaConsumer.assign(topicPartitions);

    while (true){

        ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));

        for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
            System.out.println(consumerRecord);
        }
    }
}
```

3) 测试

(1) 在 IDEA 中执行消费者程序。

(2) 在 IDEA 中执行生产者程序 CustomProducerCallback()在控制台观察生成几个 0 号分区的数据。

```
first 0 381
first 0 382
first 2 168
first 1 165
first 1 166
```

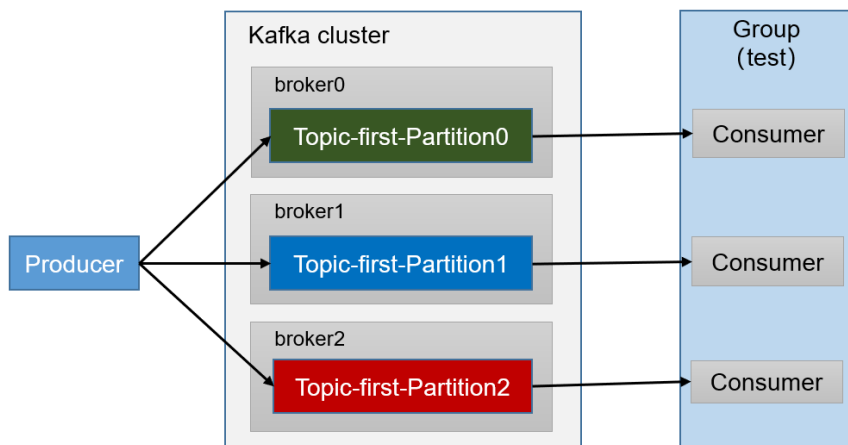
(3) 在 IDEA 控制台，观察接收到的数据，只能消费到 0 号分区数据表示正确。

```
ConsumerRecord(topic = first, partition = 0, leaderEpoch = 14,
offset = 381, CreateTime = 1636791331386, serialized key size = -
1, serialized value size = 9, headers = RecordHeaders(headers =
[], isReadOnly = false), key = null, value = atguigu 0)

ConsumerRecord(topic = first, partition = 0, leaderEpoch = 14,
offset = 382, CreateTime = 1636791331397, serialized key size = -
1, serialized value size = 9, headers = RecordHeaders(headers =
[], isReadOnly = false), key = null, value = atguigu 1)
```

5.3.3 消费者组案例

1) 需求：测试同一个主题的分区数据，只能由一个消费者组中的一个消费。



2) 案例实操

(1) 复制一份基础消费者的代码，在 IDEA 中同时启动，即可启动同一个消费者组中的两个消费者。

```

package com.atguigu.kafka.consumer;

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

import java.time.Duration;
import java.util.ArrayList;
import java.util.Properties;

public class CustomConsumer1 {

    public static void main(String[] args) {

        // 1.创建消费者的配置对象
        Properties properties = new Properties();

        // 2.给消费者配置对象添加参数
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // 配置序列化 必须
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());

        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());

        // 配置消费者组 必须
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");

        // 创建消费者对象
        KafkaConsumer<String, String> kafkaConsumer = new
            KafkaConsumer<String, String>(properties);

```

更多 Java - 大数据 - 前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
// 注册主题
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);

// 拉取数据打印
while (true) {
    // 设置 1s 中消费一批数据
    ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));

    // 打印消费到的数据
    for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
        System.out.println(consumerRecord);
    }
}
}
```

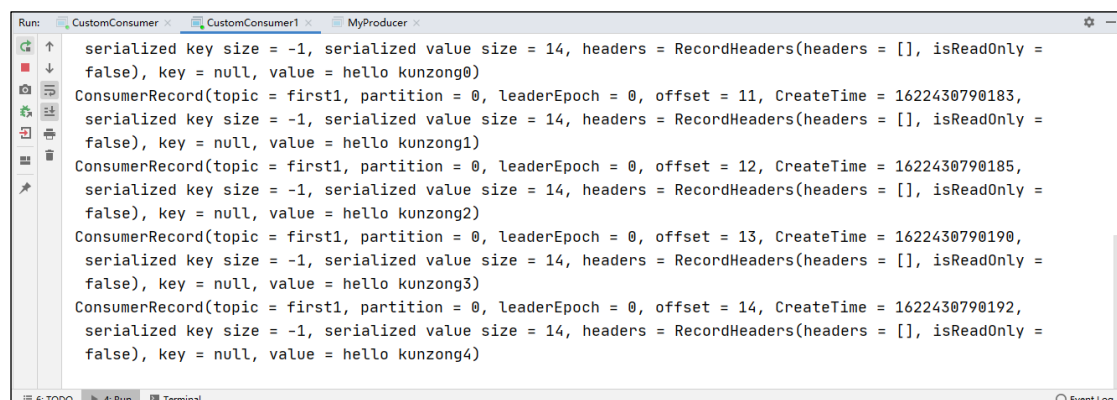
(2) 启动代码中的生产者发送消息，在 IDEA 控制台即可看到两个消费者在消费不同分区的数据（如果只发生到一个分区，可以在发送时增加延迟代码 `Thread.sleep(2);`）。

```
ConsumerRecord(topic = first, partition = 0, leaderEpoch = 2,
offset = 3, CreateTime = 1629169606820, serialized key size = -1,
serialized value size = 8, headers = RecordHeaders(headers = [],
isReadOnly = false), key = null, value = hello1)

ConsumerRecord(topic = first, partition = 1, leaderEpoch = 3,
offset = 2, CreateTime = 1629169609524, serialized key size = -1,
serialized value size = 6, headers = RecordHeaders(headers = [],
isReadOnly = false), key = null, value = hello2)

ConsumerRecord(topic = first, partition = 2, leaderEpoch = 3,
offset = 21, CreateTime = 1629169611884, serialized key size = -1,
serialized value size = 6, headers = RecordHeaders(headers = [],
isReadOnly = false), key = null, value = hello3)
```

(3) 重新发送到一个全新的主题中，由于默认创建的主题分区数为 1，可以看到只能有一个消费者消费到数据。



```
Run: CustomConsumer x CustomConsumer1 x MyProducer x
serialized key size = -1, serialized value size = 14, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = hello kunzong0)
ConsumerRecord(topic = first1, partition = 0, leaderEpoch = 0, offset = 11, CreateTime = 1622430790183,
serialized key size = -1, serialized value size = 14, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = hello kunzong1)
ConsumerRecord(topic = first1, partition = 0, leaderEpoch = 0, offset = 12, CreateTime = 1622430790185,
serialized key size = -1, serialized value size = 14, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = hello kunzong2)
ConsumerRecord(topic = first1, partition = 0, leaderEpoch = 0, offset = 13, CreateTime = 1622430790190,
serialized key size = -1, serialized value size = 14, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = hello kunzong3)
ConsumerRecord(topic = first1, partition = 0, leaderEpoch = 0, offset = 14, CreateTime = 1622430790192,
serialized key size = -1, serialized value size = 14, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = hello kunzong4)
```



5.3.4 消费者的重试机制

当某个消息消费异常，Kafka 消费者仍然能够继续消费后续的消息，不会一直卡在当前消息，保证了业务的正常进行。

1 默认配置下，消费异常会进行重试，重试次数是多少，重试是否有时间间隔？

Kafka 消费者在默认配置下会进行最多 **10 次** 的重试，每次重试的时间间隔为 **0**，即立即进行重试。如果在 10 次重试后仍然无法成功消费消息，则不再进行重试，消息将被视为消费失败。

2 如何自定义重试次数以及时间间隔？

```
@Bean
public KafkaListenerContainerFactory
kafkaListenerContainerFactory(ConsumerFactory<String, String>
consumerFactory) {
    ConcurrentKafkaListenerContainerFactory factory = new
ConcurrentKafkaListenerContainerFactory();
    // 自定义重试时间间隔以及次数
    FixedBackOff fixedBackOff = new FixedBackOff(1000, 5);
    factory.setCommonErrorHandler(new DefaultErrorHandler(fixedBackOff));
    factory.setConsumerFactory(consumerFactory);
    return factory;
}
```

或者：

```
// 重试 5 次，重试间隔 100 毫秒，最大间隔 1 秒
@RetryableTopic(
    attempts = "5",
    backoff = @Backoff(delay = 100, maxDelay = 1000)
)
@KafkaListener(topics = {KafkaConst.TEST_TOPIC}, groupId = "apple")
```

3 重试失败后的数据如何再次处理？

死信队列（Dead Letter Queue，简称 DLQ）是消息中间件中的一种特殊队列。它主要用于处理无法被消费者正确处理的消息，通常是因为消息格式错误、处理失败、消费超时等情况导致的消息被“丢弃”或“死亡”的情况。当消息进入队列后，消费者会尝试处理它。如果处理失败，或者超过一定的重试次数仍无法被成功处理，消息可以发送到死信队列中，而不是被永久性地丢弃。在死信队列中，可以进一步分析、处理这些无法正常消费的消息，以便定位问题、修复错误，并采取适当的措施。

对于死信队列的处理，既可以用 `@DltHandler` 处理，也可以使用 `@KafkaListener` 重新消费。

5.3.5 @KafkaListener 批量消费数据

配置:

```
@Configuration
public class BatchConfig {

    @Bean
    KafkaListenerContainerFactory<?> batchFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String>
factory = new
        ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(new
DefaultKafkaConsumerFactory<>(consumerConfigs()));
        factory.setBatchListener(true); // 开启批量监听
        return factory;
    }

    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        // props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "true");
        props.put(ConsumerConfig.BootstrapServersConfig,
"hahhome:9092");
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100); //设置
每次接收 Message 的数量
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG,
"100");
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 120000);
        props.put(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG, 180000);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        return props;
    }
}
```

使用:

```
@KafkaListener(topics = xxx, groupId = xxx, containerFactory =
"batchFactory")
public void consumeTopic(List<ConsumerRecord<?, ?>> records) {
    // 批量拉取消息进行处理
    for (ConsumerRecord<?, ?> record : records) {
```

5.4 生产经验——分区的分配以及再平衡

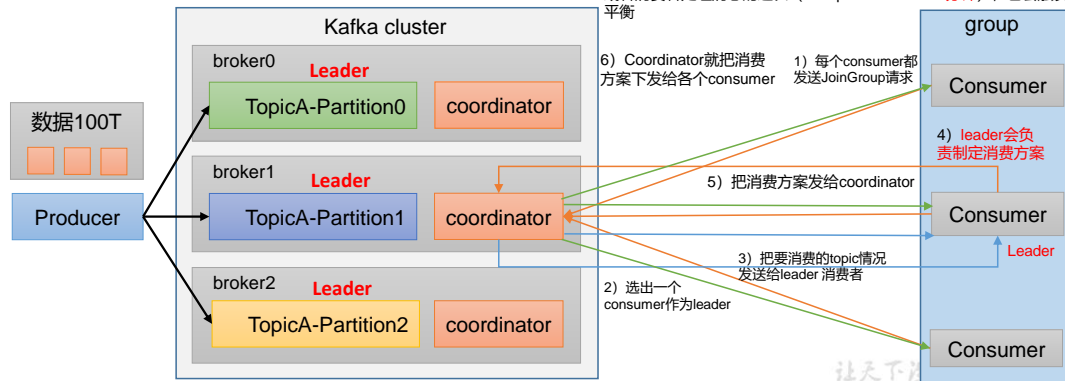
生产经验——分区的分配以及再平衡



1、一个consumer group中有多个consumer组成，一个topic有多个partition组成，现在的问题是，到底由哪个consumer来消费哪个partition的数据。

2、Kafka有四种主流的分区分配策略：Range、RoundRobin、Sticky、CooperativeSticky。可以通过配置参数partition.assignment.strategy，修改分区的分配策略。默认策略是Range + CooperativeSticky。Kafka可以同时使用多个分区分配策略。

7) 每个消费者都会和coordinator保持心跳（默认3s），一旦超时（session.timeout.ms=45s），该消费者会被移除，并触发再平衡；或者消费者处理消息的过长（max.poll.interval.ms5分钟），也会触发再平衡



参数名称	描述
heartbeat.interval.ms	Kafka 消费者和 coordinator 之间的心跳时间，默认 3s。 该条目的值必须小于 session.timeout.ms，也不应该高于 session.timeout.ms 的 1/3。
session.timeout.ms	Kafka 消费者和 coordinator 之间连接超时时间，默认 45s。超过该值，该消费者被移除，消费者组执行再平衡。
max.poll.interval.ms	消费者处理消息的最大时长，默认是 5 分钟。超过该值，该消费者被移除，消费者组执行再平衡。
partition.assignment.strategy	消费者分区分配策略，默认策略是 Range + CooperativeSticky。Kafka 可以同时使用多个分区分配策略。 可以选择的策略包括：Range、RoundRobin、Sticky、CooperativeSticky

5.4.1 Range 以及再平衡

1) Range 分区策略原理



分区分配策略之Range



Range 是对每个 topic 而言的。

首先对同一个 topic 里面的分区分按照序号进行排序，并对消费者按照字母顺序进行排序。

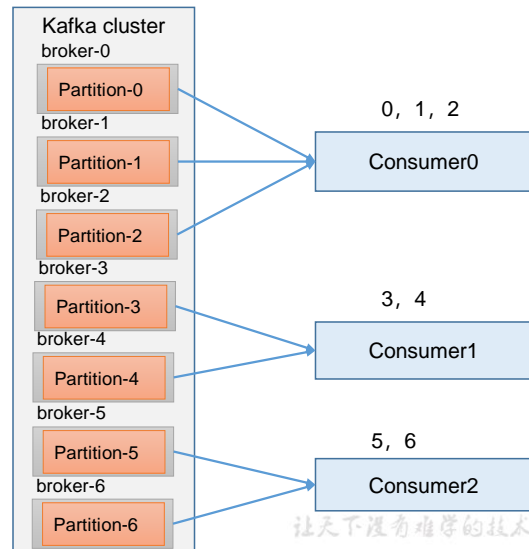
假如现在有 7 个分区，3 个消费者，排序后的分区将会是 0,1,2,3,4,5,6；消费者排序完之后将会是 C0,C1,C2。

通过 **partitions数/consumer数** 来决定每个消费者应该消费几个分区。如果除不尽，那么前面几个消费者将会多消费 1 个分区。

例如， $7/3 = 2$ 余 1，除不尽，那么消费者 C0 便会多消费 1 个分区。 $8/3 = 2$ 余 2，除不尽，那么 C0 和 C1 分别多消费一个。

注意：如果只是针对 1 个 topic 而言，C0 消费者多消费 1 个分区影响不是很大。但是如果有 N 多个 topic，那么针对每个 topic，消费者 C0 都将多消费 1 个分区，topic 越多，C0 消费的分区分会比其他消费者明显多消费 N 个分区。

容易产生数据倾斜！



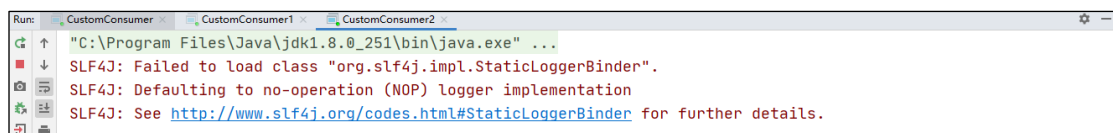
2) Range 分区分配策略案例

(1) 修改主题 first 为 7 个分区。

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server hadoop102:9092 --alter --topic first --partitions 7
```

注意：分区分数可以增加，但是不能减少。

(2) 复制 CustomConsumer 类，创建 CustomConsumer2。这样可以由三个消费者 CustomConsumer、CustomConsumer1、CustomConsumer2 组成消费者组，组名都为“test”，同时启动 3 个消费者。



(3) 启动 CustomProducer 生产者，发送 500 条消息，随机发送到不同的分区。

```
package com.atguigu.kafka.producer;

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

import java.util.Properties;

public class CustomProducer {

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

        Properties properties = new Properties();

        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
```



```
"hadoop102:9092");

        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        KafkaProducer<String, String> kafkaProducer = new
KafkaProducer<>(properties);

        for (int i = 0; i < 7; i++) {
            kafkaProducer.send(new ProducerRecord<>("first", i,
"test", "atguigu"));
        }

        kafkaProducer.close();
    }
}
```

说明：Kafka 默认的分区分配策略就是 Range + CooperativeSticky，所以不需要修改策略。

(4) 观看 3 个消费者分别消费哪些分区的数据。

```
Run: CustomConsumer x CustomConsumer1 x CustomConsumer2 x CustomProducer (1) x
Serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu481)
ConsumerRecord(topic = first, partition = 1, leaderEpoch = 6, offset = 409, CreateTime = 1622448997128,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu484)
ConsumerRecord(topic = first, partition = 2, leaderEpoch = 4, offset = 412, CreateTime = 1622448997134,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu486)
ConsumerRecord(topic = first, partition = 0, leaderEpoch = 15, offset = 429, CreateTime = 1622448997143,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu490)

Run: CustomConsumer x CustomConsumer1 x CustomConsumer2 x CustomProducer (1) x
false), key = null, value = atguigu480)
ConsumerRecord(topic = first, partition = 3, leaderEpoch = 0, offset = 76, CreateTime = 1622448997126,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu483)
ConsumerRecord(topic = first, partition = 3, leaderEpoch = 0, offset = 77, CreateTime = 1622448997139,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu488)
ConsumerRecord(topic = first, partition = 3, leaderEpoch = 0, offset = 78, CreateTime = 1622448997146,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu491)
ConsumerRecord(topic = first, partition = 4, leaderEpoch = 0, offset = 70, CreateTime = 1622448997154,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu494)

Run: CustomConsumer x CustomConsumer1 x CustomConsumer2 x CustomProducer (1) x
false), key = null, value = atguigu487)
ConsumerRecord(topic = first, partition = 5, leaderEpoch = 0, offset = 75, CreateTime = 1622448997141,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu489)
ConsumerRecord(topic = first, partition = 6, leaderEpoch = 0, offset = 63, CreateTime = 1622448997149,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu492)
```

3) Range 分区分配再平衡案例

(1) 停止掉 0 号消费者，快速重新发送消息观看结果（45s 以内，越快越好）。

1 号消费者：消费到 3、4 号分区数据。

2 号消费者：消费到 5、6 号分区数据。

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

0号消费者的任务会整体被分配到1号消费者或者2号消费者。

说明：0号消费者挂掉后，消费者组需要按照超时时间45s来判断它是否退出，所以需要等待，时间到了45s后，判断它真的退出就会把任务分配给其他broker执行。

(2) 再次重新发送消息观看结果(45s以后)。

1号消费者：消费到0、1、2、3号分区数据。

2号消费者：消费到4、5、6号分区数据。

说明：消费者0已经被踢出消费者组，所以重新按照range方式分配。

5.4.2 RoundRobin 以及再平衡

1) RoundRobin 分区策略原理

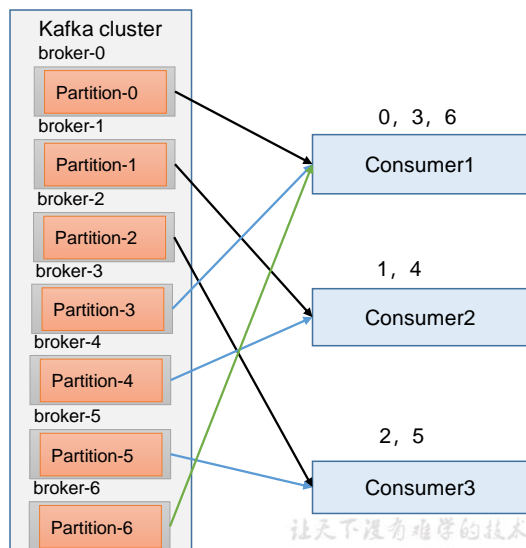


分区分配策略之RoundRobin



RoundRobin 针对集群中所有Topic而言。

RoundRobin 轮询分区策略，是把所有的 partition 和所有的 consumer 都列出来，然后按照 hashcode 进行排序，最后通过轮询算法来分配 partition 给到各个消费者。



2) RoundRobin 分区分配策略案例

(1) 依次在 CustomConsumer、CustomConsumer1、CustomConsumer2 三个消费者代码中修改分区分配策略为 RoundRobin。

```
// 修改分区分配策略
properties.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG, "org.apache.kafka.clients.consumer.RoundRobinAssignor");
```

(2) 重启3个消费者，重复发送消息的步骤，观看分区结果。

```
Run: CustomConsumer x CustomConsumer1 x CustomConsumer2 x CustomProducer (1) x
ConsumerRecord(topic = first, partition = 3, leaderEpoch = 0, offset = 156, CreateTime = 1622449331774,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu482)
ConsumerRecord(topic = first, partition = 3, leaderEpoch = 0, offset = 156, CreateTime = 1622449331996,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu483)
ConsumerRecord(topic = first, partition = 6, leaderEpoch = 0, offset = 123, CreateTime = 1622449332001,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu485)
ConsumerRecord(topic = first, partition = 0, leaderEpoch = 15, offset = 498, CreateTime = 1622449332004,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu486)
```

```
Run: CustomConsumer x CustomConsumer1 x CustomConsumer2 x CustomProducer (1) x
ConsumerRecord(topic = first, partition = 2, leaderEpoch = 4, offset = 490, CreateTime = 1622449332025,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu494)
ConsumerRecord(topic = first, partition = 2, leaderEpoch = 4, offset = 491, CreateTime = 1622449332027,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu495)
ConsumerRecord(topic = first, partition = 5, leaderEpoch = 0, offset = 147, CreateTime = 1622449332031,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu497)
```

```
Run: CustomConsumer x CustomConsumer1 x CustomConsumer2 x CustomProducer (1) x
ConsumerRecord(topic = first, partition = 4, leaderEpoch = 0, offset = 145, CreateTime = 1622449331999,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu484)
ConsumerRecord(topic = first, partition = 1, leaderEpoch = 6, offset = 477, CreateTime = 1622449332012,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu489)
```

3) RoundRobin 分区分配再平衡案例

(1) 停止掉 0 号消费者，快速重新发送消息观看结果（45s 以内，越快越好）。

1 号消费者：消费到 2、5 号分区数据

2 号消费者：消费到 4、1 号分区数据

0 号消费者的任务会按照 RoundRobin 的方式，把数据轮询分成 0、6 和 3 号分区数据，分别由 1 号消费者或者 2 号消费者消费。

说明：0 号消费者挂掉后，消费者组需要按照超时时间 45s 来判断它是否退出，所以需要等待，时间到了 45s 后，判断它真的退出就会把任务分配给其他 broker 执行。

(2) 再次重新发送消息观看结果（45s 以后）。

1 号消费者：消费到 0、2、4、6 号分区数据

2 号消费者：消费到 1、3、5 号分区数据

说明：消费者 0 已经被踢出消费者组，所以重新按照 RoundRobin 方式分配。

5.4.3 Sticky 以及再平衡

粘性分区定义：可以理解为分配的结果带有“粘性的”。即在执行一次新的分配之前，考虑上一次分配的结果，尽量少的调整分配的变动，可以节省大量的开销。

粘性分区是 Kafka 从 0.11.x 版本开始引入这种分配策略，**首先会尽量均衡的放置分区到消费者上面**，在出现同一消费者组内消费者出现问题的时候，**会尽量保持原有分配的分区不变化**。

1) 需求

设置主题为 first，7 个分区；准备 3 个消费者，采用粘性分区策略，并进行消费，观察消费分配情况。然后再停止其中一个消费者，再次观察消费分配情况。

2) 步骤

(1) 修改分区分配策略为粘性。

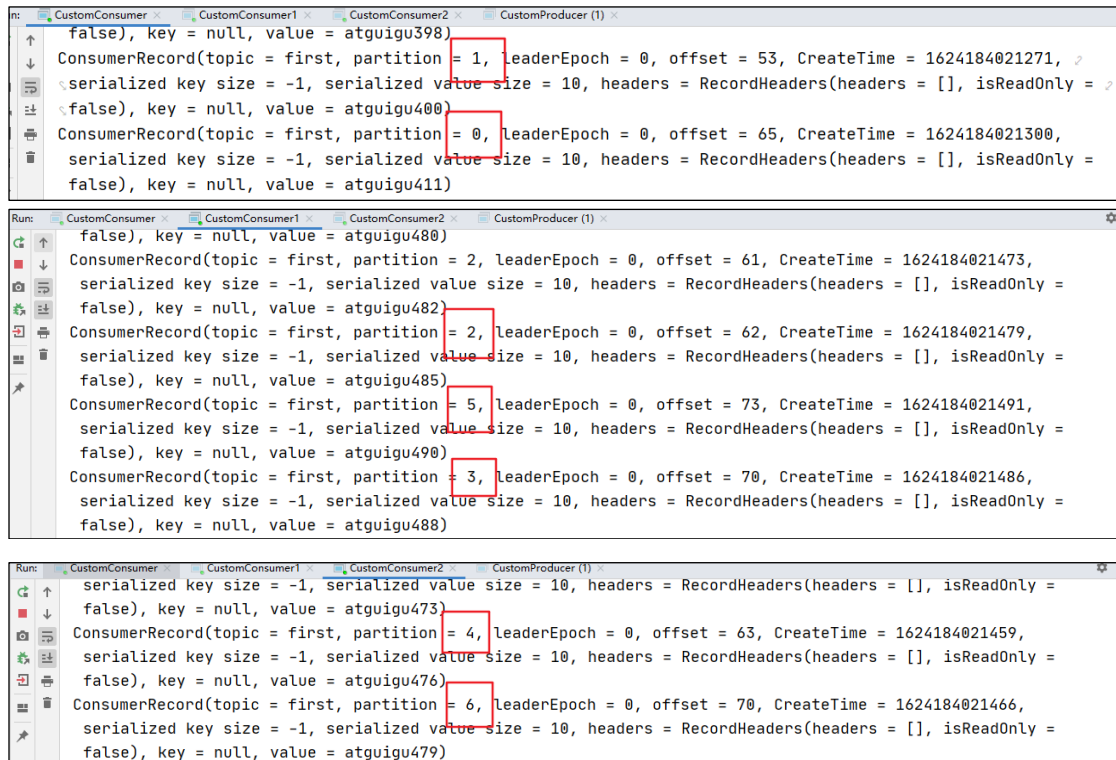
注意：3 个消费者都应该注释掉，之后重启 3 个消费者，如果出现报错，全部停止等会再重启，或者修改为全新的消费者组。

```
// 修改分区分配策略
ArrayList<String> startegys = new ArrayList<>();
startegys.add("org.apache.kafka.clients.consumer.StickyAssignor");

properties.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
startegys);
```

(2) 使用同样的生产者发送 500 条消息。

可以看到会尽量保持分区的个数近似划分分区。



```
CustomConsumer × CustomConsumer1 × CustomConsumer2 × CustomProducer (1) ×
false), key = null, value = atguigu398)
ConsumerRecord(topic = first, partition = 1, leaderEpoch = 0, offset = 53, CreateTime = 1624184021271,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu400)
ConsumerRecord(topic = first, partition = 0, leaderEpoch = 0, offset = 65, CreateTime = 1624184021300,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu411)

Run: CustomConsumer × CustomConsumer1 × CustomConsumer2 × CustomProducer (1) ×
false), key = null, value = atguigu480)
ConsumerRecord(topic = first, partition = 2, leaderEpoch = 0, offset = 61, CreateTime = 1624184021473,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu482)
ConsumerRecord(topic = first, partition = 2, leaderEpoch = 0, offset = 62, CreateTime = 1624184021479,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu485)
ConsumerRecord(topic = first, partition = 5, leaderEpoch = 0, offset = 73, CreateTime = 1624184021491,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu490)
ConsumerRecord(topic = first, partition = 3, leaderEpoch = 0, offset = 70, CreateTime = 1624184021486,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu488)

Run: CustomConsumer × CustomConsumer1 × CustomConsumer2 × CustomProducer (1) ×
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu473)
ConsumerRecord(topic = first, partition = 4, leaderEpoch = 0, offset = 63, CreateTime = 1624184021459,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu476)
ConsumerRecord(topic = first, partition = 6, leaderEpoch = 0, offset = 70, CreateTime = 1624184021466,
serialized key size = -1, serialized value size = 10, headers = RecordHeaders(headers = [], isReadOnly =
false), key = null, value = atguigu479)
```

3) Sticky 分区分配再平衡案例

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

(1) 停止掉 0 号消费者，快速重新发送消息观看结果（45s 以内，越快越好）。

1 号消费者：消费到 2、5、3 号分区数据。

2 号消费者：消费到 4、6 号分区数据。

0 号消费者的任务会按照粘性规则，尽可能均衡的随机分成 0 和 1 号分区数据，分别由 1 号消费者或者 2 号消费者消费。

说明：0 号消费者挂掉后，消费者组需要按照超时时间 45s 来判断它是否退出，所以需要等待，时间到了 45s 后，判断它真的退出就会把任务分配给其他 broker 执行。

(2) 再次重新发送消息观看结果（45s 以后）。

1 号消费者：消费到 2、3、5 号分区数据。

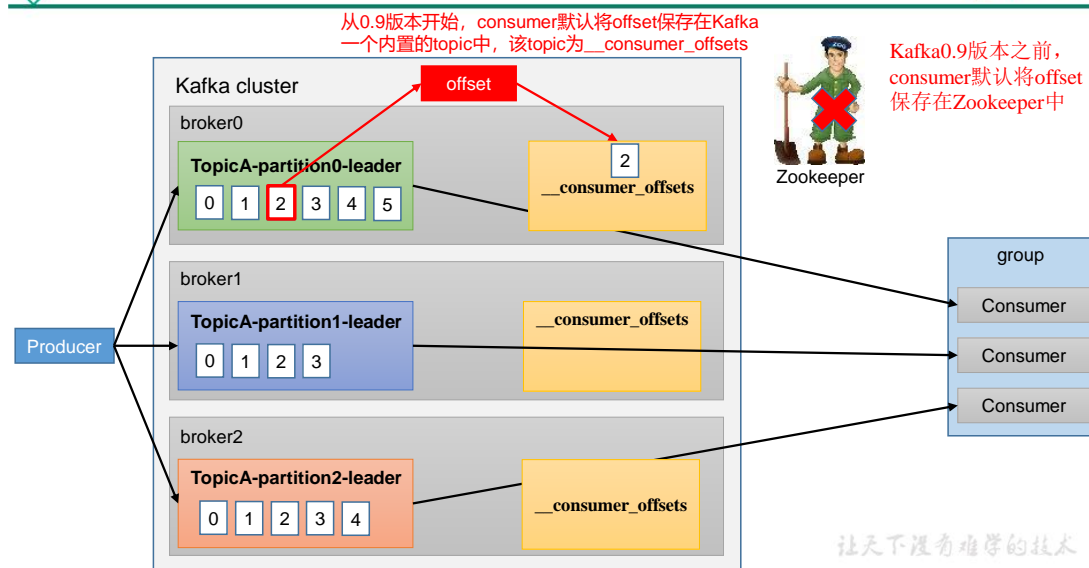
2 号消费者：消费到 0、1、4、6 号分区数据。

说明：消费者 0 已经被踢出消费者组，所以重新按照粘性方式分配。

5.5 offset 位移

5.5.1 offset 的默认维护位置

offset 的默认维护位置



`__consumer_offsets` 主题里面采用 key 和 value 的方式存储数据。key 是 `group.id+topic+分区号`，value 就是当前 offset 的值。每隔一段时间，kafka 内部会对这个 topic 进行 compact，也就是每个 `group.id+topic+分区号` 就保留最新数据。

1) 消费 offset 案例

(0) 思想：`__consumer_offsets` 为 Kafka 中的 topic，那就可以通过消费者进行消费。

(1) 在配置文件 `config/consumer.properties` 中添加配置 `exclude.internal.topics=false`，默认是 `true`，表示不能消费系统主题。为了查看该系统主题数据，所以该参数修改为 `false`。

(2) 采用命令行方式，创建一个新的 topic。

```
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server
hadoop102:9092 --create --topic atguigu --partitions 2 --
replication-factor 2
```

(3) 启动生产者往 `atguigu` 生产数据。

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh --topic
atguigu --bootstrap-server hadoop102:9092
```

(4) 启动消费者消费 `atguigu` 数据。

```
[atguigu@hadoop104 kafka]$ bin/kafka-console-consumer.sh --
bootstrap-server hadoop102:9092 --topic atguigu --group test
```

注意：指定消费者组名称，更好观察数据存储位置（key 是 `group.id+topic+分区号`）。

(5) 查看消费者消费主题 `_consumer_offsets`。

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh --topic
_consumer_offsets --bootstrap-server hadoop102:9092 --
consumer.config config/consumer.properties --formatter
"kafka.coordinator.group.GroupMetadataManager\${OffsetsMessageForm
atter}" --from-beginning

[offset,atguigu,1]::OffsetAndMetadata(offset=7,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1622442520203,
expireTimestamp=None)
[offset,atguigu,0]::OffsetAndMetadata(offset=8,
leaderEpoch=Optional[0], metadata=, commitTimestamp=1622442520203,
expireTimestamp=None)
```

5.5.2 自动提交 offset

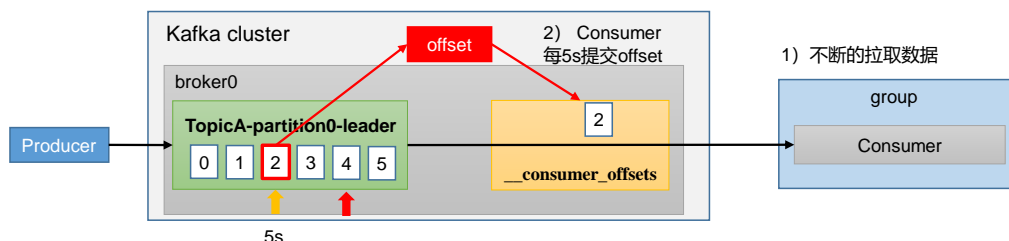
自动提交offset



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，默认是true
- **auto.commit.interval.ms**：自动提交offset的时间间隔，默认是5s



让天下没有难学的技术

参数名称	描述
------	----

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

enable.auto.commit	默认值为 true ，消费者会自动周期性地向服务器提交偏移量。
auto.commit.interval.ms	如果设置了 enable.auto.commit 的值为 true ，则该值定义了消费者偏移量向 Kafka 提交的频率，默认 5s 。

1) 消费者自动提交 offset

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

import org.apache.kafka.clients.consumer.ConsumerConfig;
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 CustomConsumerAutoOffset {

    public static void main(String[] args) {

        // 1. 创建 kafka 消费者配置类
        Properties properties = new Properties();

        // 2. 添加配置参数
        // 添加连接
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // 配置序列化 必须
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");

        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");

        // 配置消费者组
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");

        // 是否自动提交 offset
        properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
            true);

        // 提交 offset 的时间周期 1000ms，默认 5s
        properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG,
            1000);

        //3. 创建 kafka 消费者
        KafkaConsumer<String, String> consumer = new
            KafkaConsumer<>(properties);

        //4. 设置消费主题 形参是列表
        consumer.subscribe(Arrays.asList("first"));

        //5. 消费数据
        while (true){
```

```
// 读取消息
ConsumerRecords<String, String> consumerRecords =
consumer.poll(Duration.ofSeconds(1));

// 输出消息
for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
    System.out.println(consumerRecord.value());
}
}
```

5.5.3 手动提交 offset

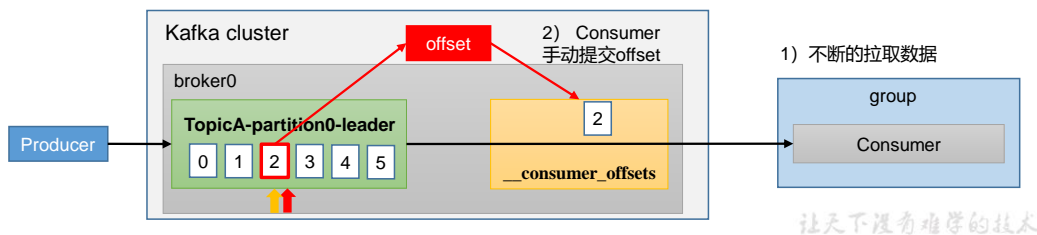
手动提交offset



虽然自动提交offset十分简单便利，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，都会将本次提交的一批数据最高的偏移量提交；不同点是，同步提交阻塞当前线程，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）；而异步提交则没有失败重试机制，故有可能提交失败。

- `commitSync`（同步提交）：必须等待offset提交完毕，再去消费下一批数据。
- `commitAsync`（异步提交）：发送完提交offset请求后，就开始消费下一批数据了。



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，但是由于一直等待提交结果，提交的效率比较低。以下为同步提交 offset 的示例。

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

import org.apache.kafka.clients.consumer.ConsumerConfig;
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 CustomConsumerByHandSync {

    public static void main(String[] args) {
        // 1. 创建 kafka 消费者配置类
        Properties properties = new Properties();
        // 2. 添加配置参数
```



```
// 添加连接
properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"hadoop102:9092");

// 配置序列化 必须
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");

// 配置消费者组
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");

// 是否自动提交 offset
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
false);

//3. 创建 kafka 消费者
KafkaConsumer<String, String> consumer = new
KafkaConsumer<>(properties);

//4. 设置消费主题 形参是列表
consumer.subscribe(Arrays.asList("first"));

//5. 消费数据
while (true){

    // 读取消息
    ConsumerRecords<String, String> consumerRecords =
consumer.poll(Duration.ofSeconds(1));

    // 输出消息
    for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
        System.out.println(consumerRecord.value());
    }

    // 同步提交 offset
    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。因此吞吐量会受到很大的影响。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 CustomConsumerByHandAsync {

    public static void main(String[] args) {

        // 1. 创建 kafka 消费者配置类
        Properties properties = new Properties();

        // 2. 添加配置参数
        // 添加连接
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // 配置序列化 必须
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");

        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");

        // 配置消费者组
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test");

        // 是否自动提交 offset
        properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
            "false");

        //3. 创建 Kafka 消费者
        KafkaConsumer<String, String> consumer = new
            KafkaConsumer<>(properties);

        //4. 设置消费主题 形参是列表
        consumer.subscribe(Arrays.asList("first"));

        //5. 消费数据
        while (true){

            // 读取消息
            ConsumerRecords<String, String> consumerRecords =
                consumer.poll(Duration.ofSeconds(1));

            // 输出消息
            for (ConsumerRecord<String, String> consumerRecord :
                consumerRecords) {
                System.out.println(consumerRecord.value());
            }

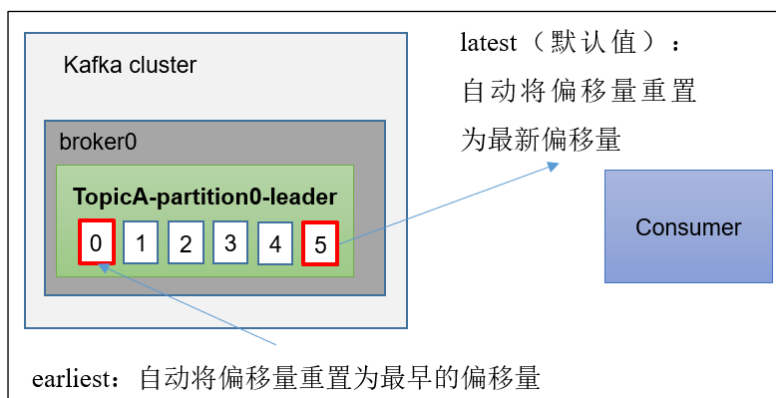
            // 异步提交 offset
            consumer.commitAsync();
        }
    }
}
```

5.5.4 指定 Offset 消费

`auto.offset.reset = earliest | latest | none` 默认是 `latest`。

当 Kafka 中没有初始偏移量（消费者组第一次消费）或服务器上不再存在当前偏移量时（例如该数据已被删除），该怎么办？

- (1) `earliest`: 自动将偏移量重置为最早的偏移量，`--from-beginning`。
- (2) `latest` (默认值): 自动将偏移量重置为最新偏移量。
- (3) `none`: 如果未找到消费者组的先前偏移量，则向消费者抛出异常。



- (4) 任意指定 offset 位移开始消费

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

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Properties;
import java.util.Set;

public class CustomConsumerSeek {

    public static void main(String[] args) {

        // 0 配置信息
        Properties properties = new Properties();

        // 连接
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // key value 反序列化
```

```
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test2");

        // 1 创建一个消费者
        KafkaConsumer<String, String> kafkaConsumer = new
KafkaConsumer<>(properties);

        // 2 订阅一个主题
        ArrayList<String> topics = new ArrayList<>();
        topics.add("first");
        kafkaConsumer.subscribe(topics);

        Set<TopicPartition> assignment= new HashSet<>();

        while (assignment.size() == 0) {
            kafkaConsumer.poll(Duration.ofSeconds(1));
            // 获取消费者分区分配信息（有了分区分配信息才能开始消费）
            assignment = kafkaConsumer.assignment();
        }

        // 遍历所有分区，并指定 offset 从 1700 的位置开始消费
        for (TopicPartition tp: assignment) {
            kafkaConsumer.seek(tp, 1700);
        }

        // 3 消费该主题数据
        while (true) {

            ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));

            for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
                System.out.println(consumerRecord);
            }
        }
    }
}
```

注意：每次执行完，需要修改消费者组名；

5.5.5 指定时间消费

需求：在生产环境中，会遇到最近消费的几个小时数据异常，想重新按照时间消费。

例如要求按照时间消费前一天的数据，怎么处理？

操作步骤：

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

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.time.Duration;
import java.util.*;

public class CustomConsumerForTime {

    public static void main(String[] args) {

        // 0 配置信息
        Properties properties = new Properties();

        // 连接
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop102:9092");

        // key value 反序列化

        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());

        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());

        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "test2");

        // 1 创建一个消费者
        KafkaConsumer<String, String> kafkaConsumer = new
            KafkaConsumer<>(properties);

        // 2 订阅一个主题
        ArrayList<String> topics = new ArrayList<>();
        topics.add("first");
        kafkaConsumer.subscribe(topics);

        Set<TopicPartition> assignment = new HashSet<>();

        while (assignment.size() == 0) {
            kafkaConsumer.poll(Duration.ofSeconds(1));
            // 获取消费者分区分配信息（有了分区分配信息才能开始消费）
            assignment = kafkaConsumer.assignment();
        }

        HashMap<TopicPartition, Long> timestampToSearch = new
            HashMap<>();

        // 封装集合存储，每个分区对应一天前的数据
        for (TopicPartition topicPartition : assignment) {
            timestampToSearch.put(topicPartition,
                System.currentTimeMillis() - 1 * 24 * 3600 * 1000);
        }

        // 获取从 1 天前开始消费的每个分区的 offset
        Map<TopicPartition, OffsetAndTimestamp> offsets =
            kafkaConsumer.offsetsForTimes(timestampToSearch);
```

```
// 遍历每个分区，对每个分区设置消费时间。
for (TopicPartition topicPartition : assignment) {
    OffsetAndTimestamp offsetAndTimestamp =
offsets.get(topicPartition);

    // 根据时间指定开始消费的位置
    if (offsetAndTimestamp != null){
        kafkaConsumer.seek(topicPartition,
offsetAndTimestamp.offset());
    }
}

// 3 消费该主题数据
while (true) {

    ConsumerRecords<String, String> consumerRecords =
kafkaConsumer.poll(Duration.ofSeconds(1));

    for (ConsumerRecord<String, String> consumerRecord :
consumerRecords) {
        System.out.println(consumerRecord);
    }
}
}
```

5.5.6 漏消费和重复消费

重复消费：已经消费了数据，但是 offset 没提交。

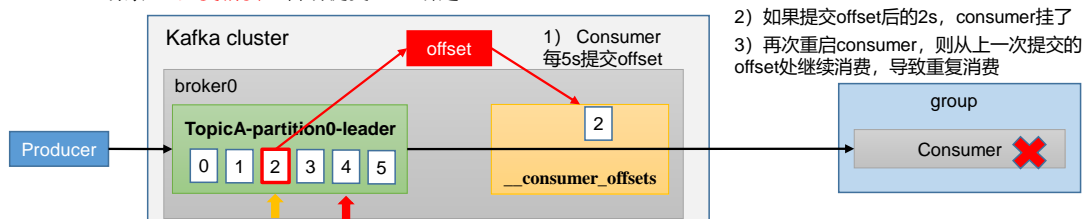
漏消费：先提交 offset 后消费，有可能会造成数据的漏消费。



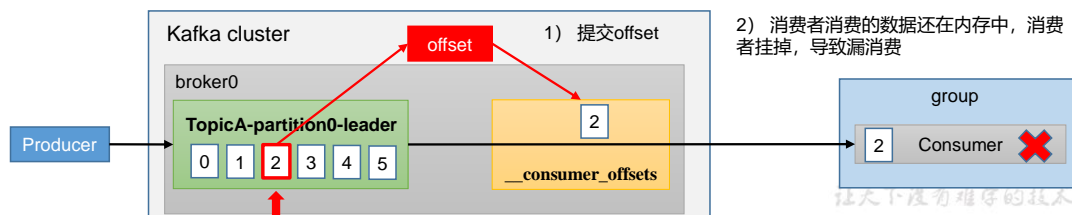
重复消费与漏消费



(1) 场景1: **重复消费**。自动提交offset引起。



(2) 场景1: **漏消费**。设置offset为手动提交，当offset被提交时，数据还在内存中未落盘，此时刚好消费者线程被kill掉，那么offset已经提交，但是数据未处理，导致这部分内存中的数据丢失。



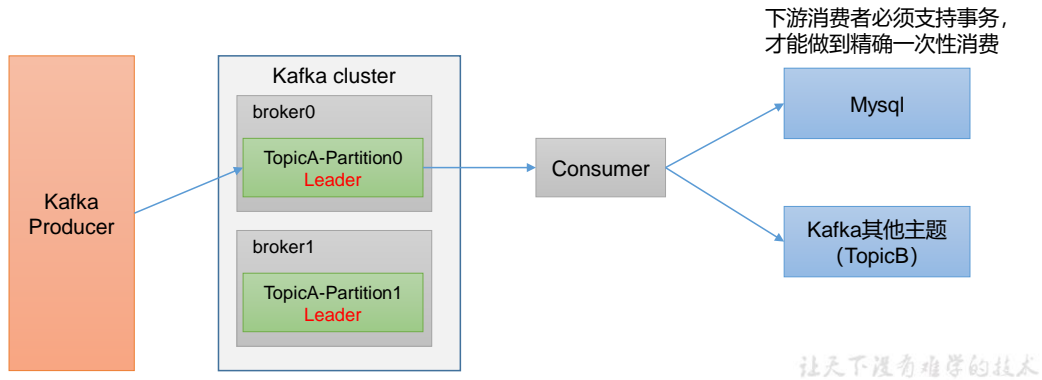
思考：怎么能做到既不漏消费也不重复消费呢？详看消费者事务。

5.6 生产经验——消费者事务

生产经验——消费者事务



如果想完成Consumer端的精准一次性消费，那么需要Kafka消费端将消费过程和提交offset过程做原子绑定。此时我们需要将Kafka的offset保存到支持事务的自定义介质（比如MySQL）。这部分知识会在后续项目部分涉及。



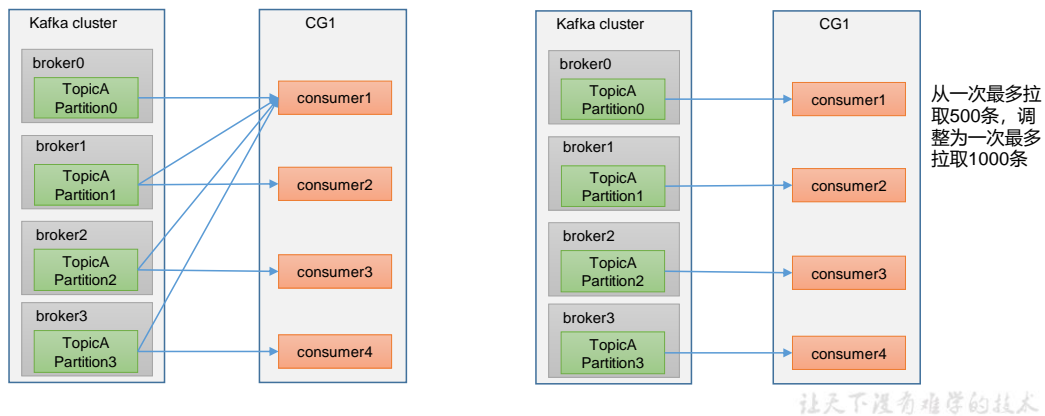
5.7 生产经验——数据积压（消费者如何提高吞吐量）

生产经验——数据积压（消费者如何提高吞吐量）



1) 如果是Kafka消费能力不足，则可以考虑增加Topic的分区数，并且同时提升消费组的消费者数量，消费者数 = 分区数。（两者缺一不可）

2) 如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间 < 生产速度），使处理的数据小于生产的数据，也会造成数据积压。



参数名称	描述
fetch.max.bytes	默认 Default: 52428800 (50 m) 。消费者获取服务器端一批消息最大的字节数。如果服务器端一批次的数据大于该值（50m）仍然可以拉取回来这批数据，因此，这不是一个绝对最大值。一批次的大小受 message.max.bytes（broker config）or max.message.bytes（topic config）影响。
max.poll.records	一次 poll 拉取数据返回消息的最大条数，默认是 500 条

第 6 章 Kafka-Eagle 监控

Kafka-Eagle 框架可以监控 Kafka 集群的整体运行情况，在生产环境中经常使用。

6.1 MySQL 环境准备

Kafka-Eagle 的安装依赖于 MySQL，MySQL 主要用来存储可视化展示的数据。如果集群中之前安装过 MySQL 可以跨过该步。



尚硅谷大数据技术
之MySQL安装.doc

6.2 Kafka 环境准备

1) 关闭 Kafka 集群

```
[atguigu@hadoop102 kafka]$ kf.sh stop
```

2) 修改/opt/module/kafka/bin/kafka-server-start.sh 命令中

```
[atguigu@hadoop102 kafka]$ vim bin/kafka-server-start.sh
```

修改如下参数值：

```
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"
fi
```

为

```
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-server -Xms2G -Xmx2G -
XX:PermSize=128m -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -
XX:ParallelGCThreads=8 -XX:ConcGCThreads=5 -
XX:InitiatingHeapOccupancyPercent=70"
    export JMX_PORT="9999"
    #export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"
fi
```

注意：修改之后在启动 Kafka 之前要分发之其他节点

```
[atguigu@hadoop102 bin]$ xsync kafka-server-start.sh
```

6.3 Kafka-Eagle 安装

0) 官网：<https://www.kafka-eagle.org/>

1) 上传压缩包 kafka-eagle-bin-2.0.8.tar.gz 到集群/opt/software 目录

2) 解压到本地

```
[atguigu@hadoop102 software]$ tar -zxvf kafka-eagle-bin-2.0.8.tar.gz
```

3) 进入刚才解压的目录

```
[atguigu@hadoop102 kafka-eagle-bin-2.0.8]$ ll
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

```
总用量 79164
```

```
-rw-rw-r--. 1 atguigu atguigu 81062577 10 月 13 00:00 efak-web-2.0.8-bin.tar.gz
```

4) 将 efak-web-2.0.8-bin.tar.gz 解压至/opt/module

```
[atguigu@hadoop102 kafka-eagle-bin-2.0.8]$ tar -zxvf efak-web-2.0.8-bin.tar.gz -C /opt/module/
```

5) 修改名称

```
[atguigu@hadoop102 module]$ mv efak-web-2.0.8/ efak
```

6) 修改配置文件 /opt/module/efak/conf/system-config.properties

```
[atguigu@hadoop102 conf]$ vim system-config.properties

#####
# multi zookeeper & kafka cluster list
# Settings prefixed with 'kafka.eagle.' will be deprecated, use 'efak.' instead
#####
efak.zk.cluster.alias=cluster1
cluster1.zk.list=hadoop102:2181,hadoop103:2181,hadoop104:2181/kafka

#####
# zookeeper enable acl
#####
cluster1.zk.acl.enable=false
cluster1.zk.acl.schema=digest
cluster1.zk.acl.username=test
cluster1.zk.acl.password=test123

#####
# broker size online list
#####
cluster1.efak.broker.size=20

#####
# zk client thread limit
#####
kafka.zk.limit.size=32
#####
# EFAK webui port
#####
efak.webui.port=8048

#####
# kafka jmx acl and ssl authenticate
#####
cluster1.efak.jmx.acl=false
cluster1.efak.jmx.user=keadmin
cluster1.efak.jmx.password=keadmin123
cluster1.efak.jmx.ssl=false
cluster1.efak.jmx.truststore.location=/data/ssl/certificates/kafka.truststore
cluster1.efak.jmx.truststore.password=ke123456

#####
# kafka offset storage
#####
# offset 保存在 kafka
cluster1.efak.offset.storage=kafka
```



```
#####
# kafka jmx uri
#####
cluster1.efak.jmx.uri=service:jmx:rmi:///jndi/rmi://%s/jmxrmi

#####
# kafka metrics, 15 days by default
#####
efak.metrics.charts=true
efak.metrics.retain=15

#####
# kafka sql topic records max
#####
efak.sql.topic.records.max=5000
efak.sql.topic.preview.records.max=10

#####
# delete kafka topic token
#####
efak.topic.token=keadmin

#####
# kafka sasl authenticate
#####
cluster1.efak.sasl.enable=false
cluster1.efak.sasl.protocol=SASL_PLAINTEXT
cluster1.efak.sasl.mechanism=SCRAM-SHA-256
cluster1.efak.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramL
oginModule required username="kafka" password="kafka-eagle";
cluster1.efak.sasl.client.id=
cluster1.efak.blacklist.topics=
cluster1.efak.sasl.cgroup.enable=false
cluster1.efak.sasl.cgroup.topics=
cluster2.efak.sasl.enable=false
cluster2.efak.sasl.protocol=SASL_PLAINTEXT
cluster2.efak.sasl.mechanism=PLAIN
cluster2.efak.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainL
oginModule required username="kafka" password="kafka-eagle";
cluster2.efak.sasl.client.id=
cluster2.efak.blacklist.topics=
cluster2.efak.sasl.cgroup.enable=false
cluster2.efak.sasl.cgroup.topics=

#####
# kafka ssl authenticate
#####
cluster3.efak.ssl.enable=false
cluster3.efak.ssl.protocol=SSL
cluster3.efak.ssl.truststore.location=
cluster3.efak.ssl.truststore.password=
cluster3.efak.ssl.keystore.location=
cluster3.efak.ssl.keystore.password=
cluster3.efak.ssl.key.password=
cluster3.efak.ssl.endpoint.identification.algorithm=https
cluster3.efak.blacklist.topics=
cluster3.efak.ssl.cgroup.enable=false
cluster3.efak.ssl.cgroup.topics=

#####
# kafka sqlite jdbc driver address
```

```
#####
# 配置mysql 连接
efak.driver=com.mysql.jdbc.Driver
efak.url=jdbc:mysql://hadoop102:3306/ke?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
efak.username=root
efak.password=000000

#####
# kafka mysql jdbc driver address
#####
#efak.driver=com.mysql.cj.jdbc.Driver
#efak.url=jdbc:mysql://127.0.0.1:3306/ke?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
#efak.username=root
#efak.password=123456
```

7) 添加环境变量

```
[atguigu@hadoop102 conf]$ sudo vim /etc/profile.d/my_env.sh

# kafkaEFAK
export KE_HOME=/opt/module/efak
export PATH=$PATH:$KE_HOME/bin
```

注意: source /etc/profile

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

8) 启动

(1) 注意: 启动之前需要先启动 ZK 以及 KAFKA。

```
[atguigu@hadoop102 kafka]$ kf.sh start
```

(2) 启动 efak

```
[atguigu@hadoop102 efak]$ bin/ke.sh start
Version 2.0.8 -- Copyright 2016-2021
*****
* EFAK Service has started success.
* Welcome, Now you can visit 'http://192.168.10.102:8048'
* Account:admin , Password:123456
*****
* <Usage> ke.sh [start|status|stop|restart|stats] </Usage>
* <Usage> https://www.kafka-eagle.org/ </Usage>
*****
```

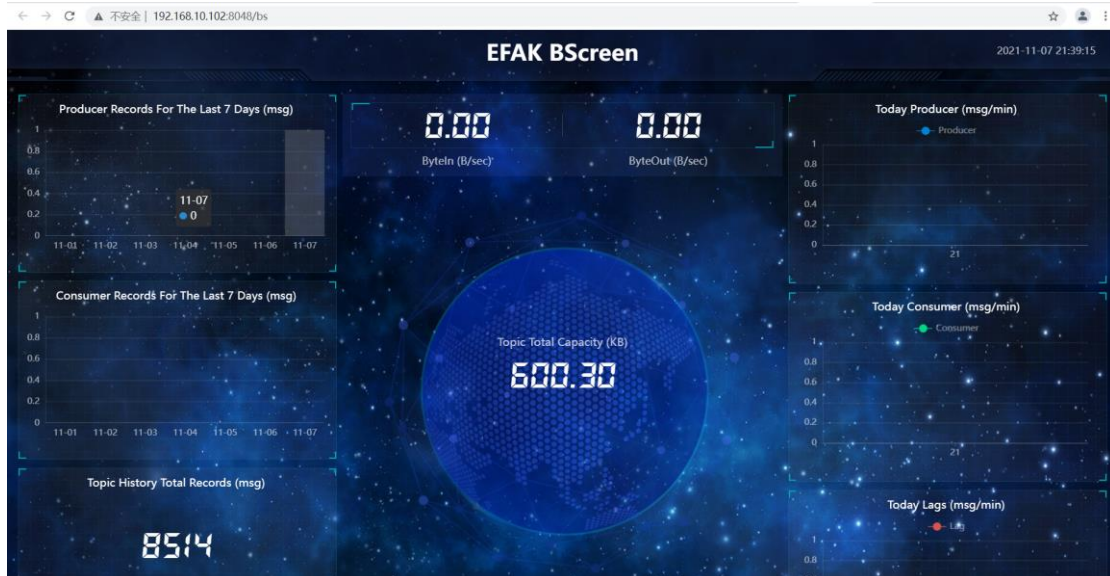
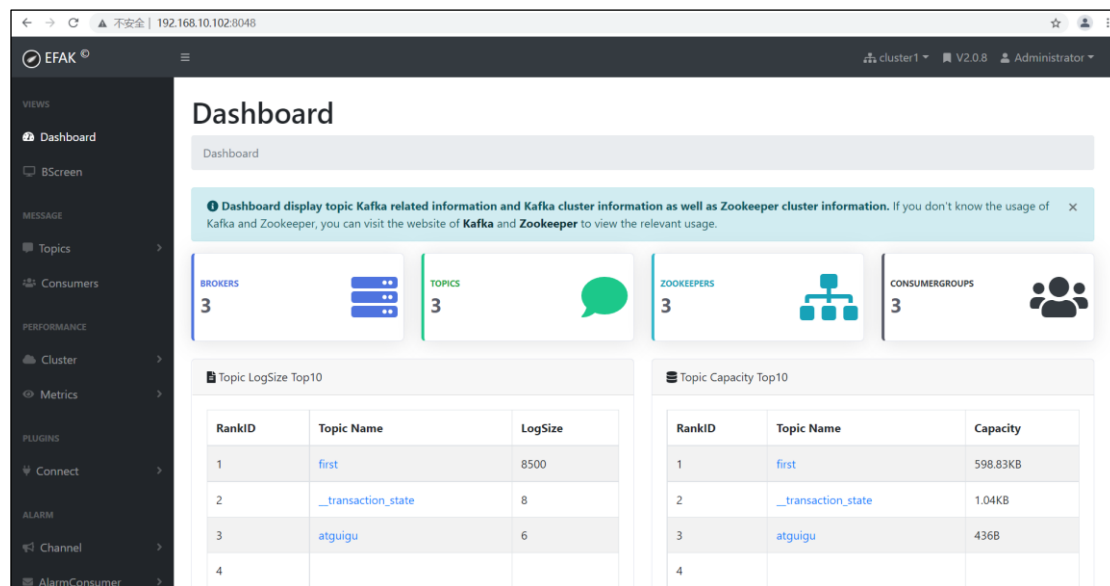
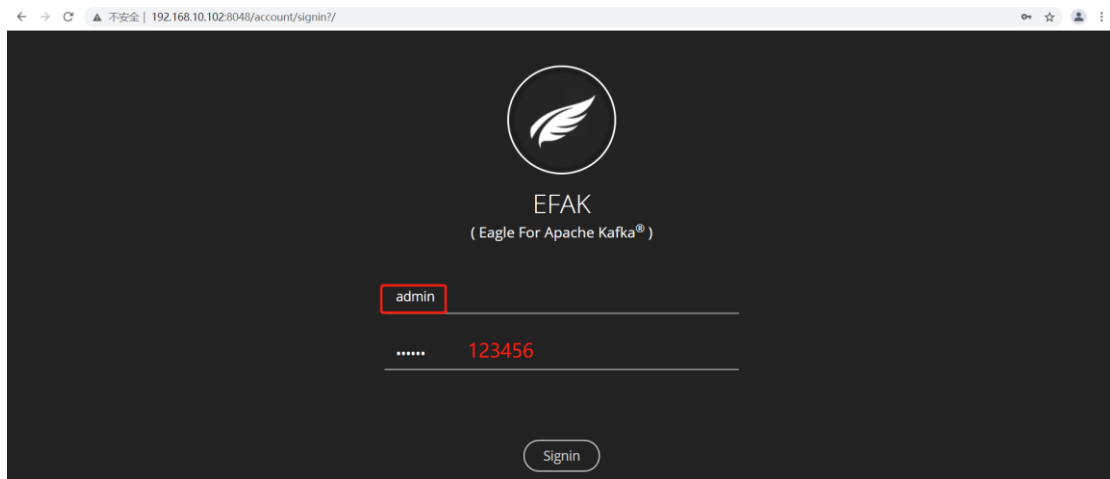
说明: 如果停止 efak, 执行命令。

```
[atguigu@hadoop102 efak]$ bin/ke.sh stop
```

6.4 Kafka-Eagle 页面操作

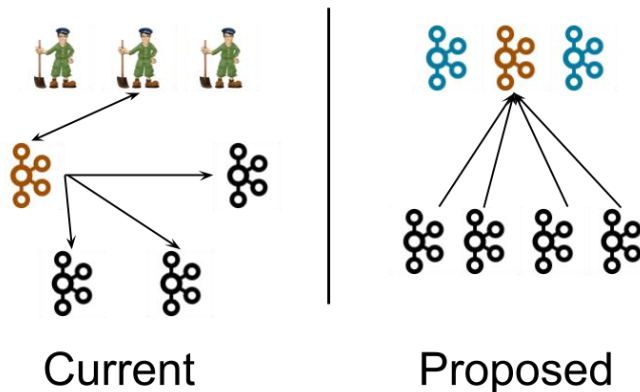
1) 登录页面查看监控数据

<http://192.168.10.102:8048/>



第 7 章 Kafka-Kraft 模式

7.1 Kafka-Kraft 架构



左图为 Kafka 现有架构，元数据在 zookeeper 中，运行时动态选举 controller，由 controller 进行 Kafka 集群管理。右图为 kraft 模式架构（实验性），不再依赖 zookeeper 集群，而是用三台 controller 节点代替 zookeeper，元数据保存在 controller 中，由 controller 直接进行 Kafka 集群管理。

这样做的好处有以下几个：

- Kafka 不再依赖外部框架，而是能够独立运行；
- controller 管理集群时，不再需要从 zookeeper 中先读取数据，集群性能上升；
- 由于不依赖 zookeeper，集群扩展时不再受到 zookeeper 读写能力限制；
- controller 不再动态选举，而是由配置文件规定。这样我们可以有针对性的加强 controller 节点的配置，而不是像以前一样对随机 controller 节点的高负载束手无策。

7.2 Kafka-Kraft 集群部署

1) 再次解压一份 kafka 安装包

```
[atguigu@hadoop102 software]$ tar -zxvf kafka_2.12-3.0.0.tgz -C /opt/module/
```

2) 重命名为 kafka2

```
[atguigu@hadoop102 module]$ mv kafka_2.12-3.0.0/ kafka2
```

3) 在 hadoop102 上修改/opt/module/kafka2/config/kraft/server.properties 配置文件

```
[atguigu@hadoop102 kraft]$ vim server.properties
```

#kafka 的角色（controller 相当于主机、broker 节点相当于从机，主机类似 zk 功能）

```
process.roles=broker, controller
```

```
#节点 ID
```

```
node.id=2
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

```
#controller 服务协议别名
controller.listener.names=CONTROLLER
#全 Controller 列表
controller.quorum.voters=2@hadoop102:9093,3@hadoop103:9093,4@hadoop104:9093
#不同服务器绑定的端口
listeners=PLAINTEXT://:9092,CONTROLLER://:9093
#broker 服务协议别名
inter.broker.listener.name=PLAINTEXT
#broker 对外暴露的地址
advertised.listeners=PLAINTEXT://hadoop102:9092
#协议别名到安全协议的映射
listener.security.protocol.map=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
#kafka 数据存储目录
log.dirs=/opt/module/kafka2/data
```

4) 分发 kafka2

```
[atguigu@hadoop102 module]$ xsync kafka2/
```

- 在 hadoop103 和 hadoop104 上需要对 **node.id** 相应改变，值需要和 controller.quorum.voters 对应。
- 在 hadoop103 和 hadoop104 上需要根据各自的主机名称，修改相应的 **advertised.listeners** 地址。

5) 初始化集群数据目录

(1) 首先生成存储目录唯一 ID。

```
[atguigu@hadoop102 kafka2]$ bin/kafka-storage.sh random-uuid
J7s9e8PPTKOO47PxzI39VA
```

(2) 用该 ID 格式化 kafka 存储目录（三台节点）。

```
[atguigu@hadoop102 kafka2]$ bin/kafka-storage.sh format -t
J7s9e8PPTKOO47PxzI39VA -c
/opt/module/kafka2/config/kraft/server.properties
```

```
[atguigu@hadoop103 kafka2]$ bin/kafka-storage.sh format -t
J7s9e8PPTKOO47PxzI39VA -c
/opt/module/kafka2/config/kraft/server.properties
```

```
[atguigu@hadoop104 kafka2]$ bin/kafka-storage.sh format -t
J7s9e8PPTKOO47PxzI39VA -c
/opt/module/kafka2/config/kraft/server.properties
```

6) 启动 kafka 集群

```
[atguigu@hadoop102 kafka2]$ bin/kafka-server-start.sh -daemon
config/kraft/server.properties
```

```
[atguigu@hadoop103 kafka2]$ bin/kafka-server-start.sh -daemon
config/kraft/server.properties
```

```
[atguigu@hadoop104 kafka2]$ bin/kafka-server-start.sh -daemon config/kraft/server.properties
```

7) 停止 kafka 集群

```
[atguigu@hadoop102 kafka2]$ bin/kafka-server-stop.sh  
[atguigu@hadoop103 kafka2]$ bin/kafka-server-stop.sh  
[atguigu@hadoop104 kafka2]$ bin/kafka-server-stop.sh
```

7.3 Kafka-Kraft 集群启动停止脚本

1) 在/home/atguigu/bin 目录下创建文件 kf2.sh 脚本文件

```
[atguigu@hadoop102 bin]$ vim kf2.sh
```

脚本如下：

```
#!/bin/bash  
  
case $1 in  
"start"){  
    for i in hadoop102 hadoop103 hadoop104  
    do  
        echo " -----启动 $i Kafka2-----"  
        ssh $i "/opt/module/kafka2/bin/kafka-server-start.sh -  
daemon /opt/module/kafka2/config/kraft/server.properties"  
    done  
};;  
"stop"){  
    for i in hadoop102 hadoop103 hadoop104  
    do  
        echo " -----停止 $i Kafka2-----"  
        ssh $i "/opt/module/kafka2/bin/kafka-server-stop.sh "  
    done  
};;  
esac
```

2) 添加执行权限

```
[atguigu@hadoop102 bin]$ chmod +x kf2.sh
```

3) 启动集群命令

```
[atguigu@hadoop102 ~]$ kf2.sh start
```

4) 停止集群命令

```
[atguigu@hadoop102 ~]$ kf2.sh stop
```