

第二章 Kafka高级

学习目标

- 理解Kafka的分区副本机制
- 能够搭建Kafka-eagle并查看Kafka集群状态
- 理解分区leader和follower的职责
- 理解分区的ISR
- 理解Kafka消息不丢失机制
- 理解Kafka中数据清理

1. 分区和副本机制

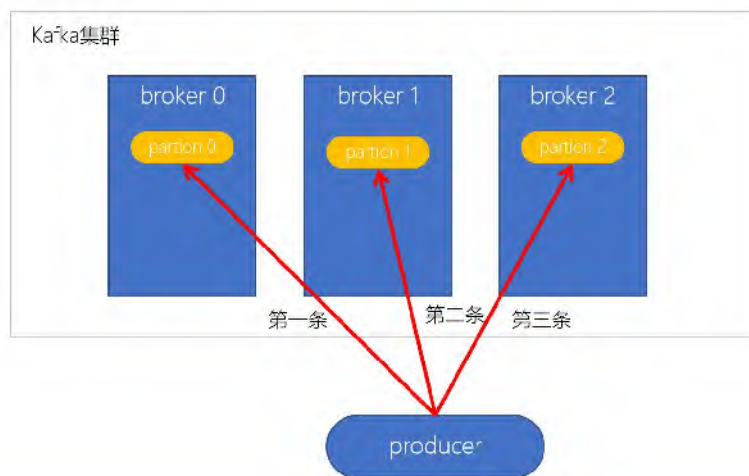
1.1 生产者分区写入策略

生产者写入消息到topic，Kafka将依据不同的策略将数据分配到不同的分区中

1. 轮询分区策略
2. 随机分区策略
3. 按key分区分配策略
4. 自定义分区策略

1.1.1 轮询策略

轮询分配策略

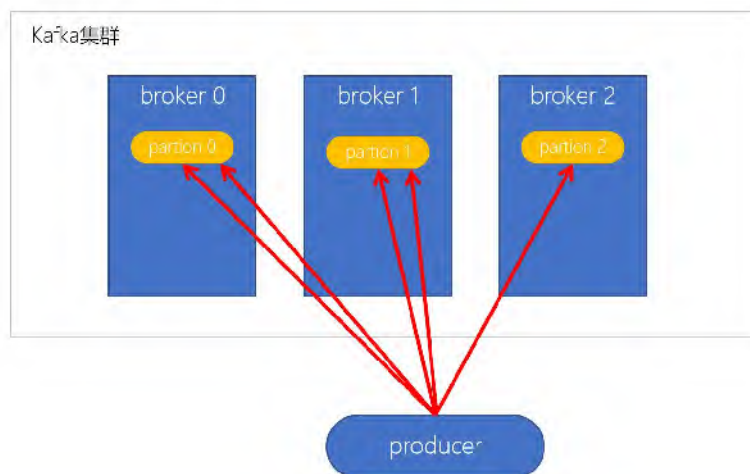


- 默认的策略，也是使用最多的策略，可以最大限度保证所有消息平均分配到一个分区
- 如果在生产消息时，key为null，则使用轮询算法均衡地分配分区

1.1.2 随机策略（不用）

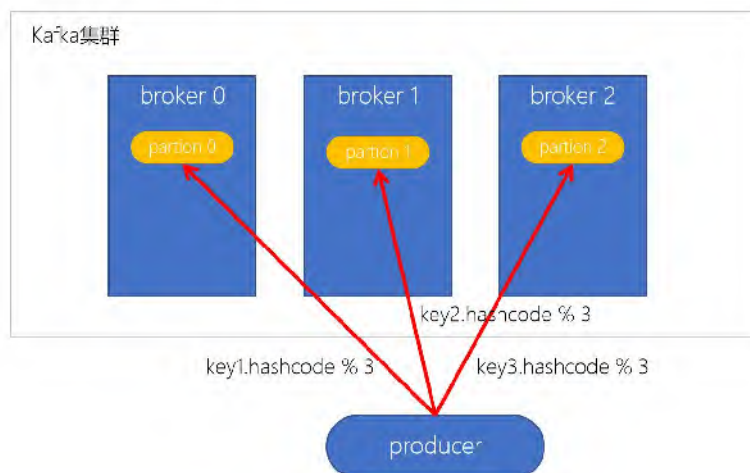
随机策略，每次都随机地将消息分配到每个分区。在较早的版本，默认的分区策略就是随机策略，也是为了将消息均衡地写入到每个分区。但后续轮询策略表现更佳，所以基本上很少会使用随机策略。

■ 随机分配策略



1.1.3 按key分配策略

■ 随机分配策略

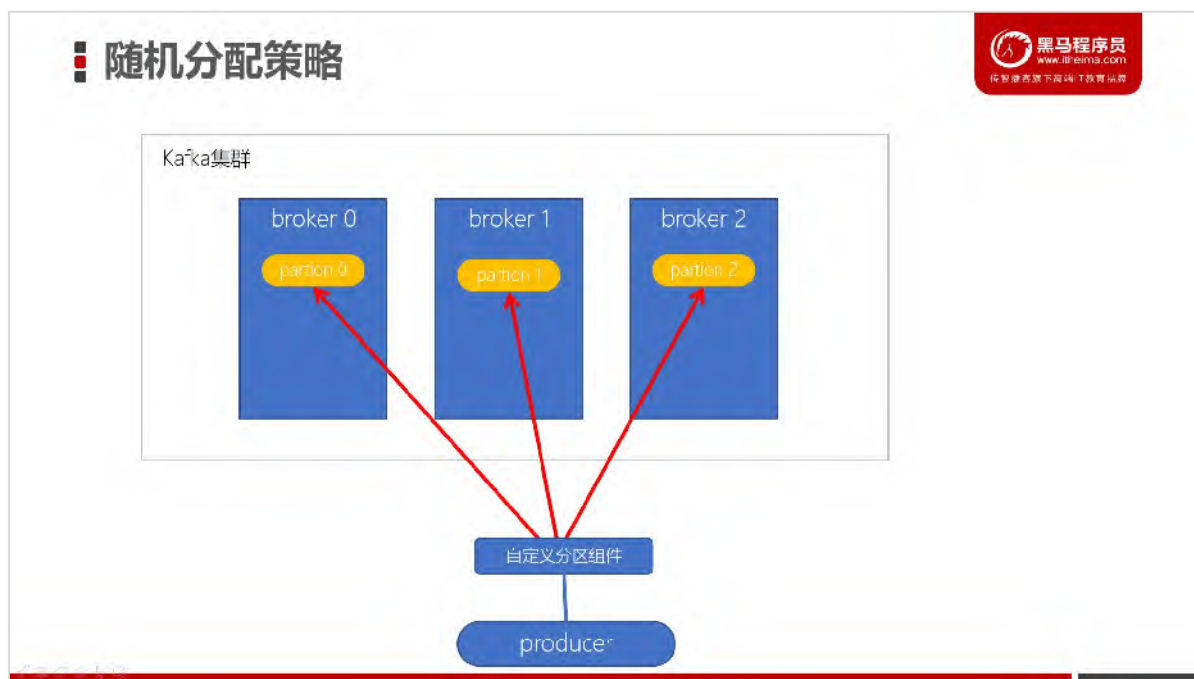


按key分配策略，有可能会出现「数据倾斜」，例如：某个key包含了大量的数据，因为key值一样，所有所有的数据将都分配到一个分区中，造成该分区的信息数量远大于其他的分区。

1.1.4 乱序问题

轮询策略、随机策略都会导致一个问题，生产到Kafka中的数据是乱序存储的。而按key分区可以一定程度上实现数据有序存储——也就是局部有序，但这又可能会导致数据倾斜，所以在实际生产环境中要结合实际情况来做取舍。

1.1.5 自定义分区策略



实现步骤：

1. 创建自定义分区器

```
public class KeyWithRandomPartitioner implements Partitioner {

    private Random r;

    @Override
    public void configure(Map<String, ?> configs) {
        r = new Random();
    }

    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[]
```



```
valueBytes, Cluster cluster) {  
    // cluster.partitionCountForTopic 表示获取指定topic的分区数量  
    return r.nextInt(1000) % cluster.partitionCountForTopic(topic);  
}  
  
@Override  
public void close() {  
}  
}
```

2. 在Kafka生产者配置中，自定义使用自定义分区器的类名

```
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, KeyWithRandomPartitioner.class.getName());
```

1.2 消费者组Rebalance机制

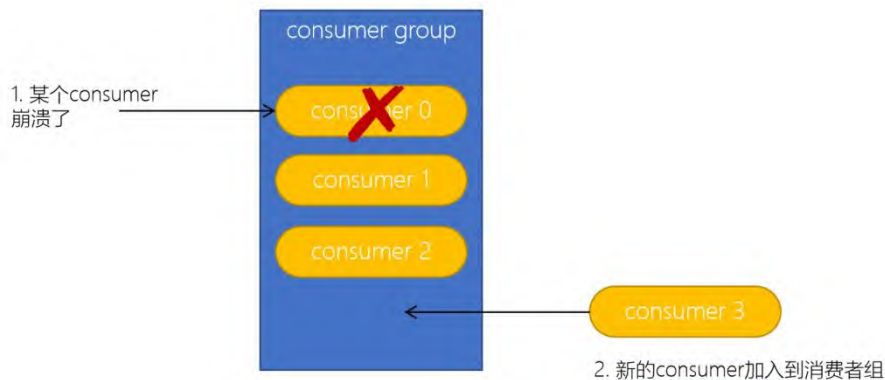
1.2.1 Rebalance再均衡

Kafka中的Rebalance称之为再均衡，是Kafka中确保Consumer group下所有的consumer如何达成一致，分配订阅的topic的每个分区的机制。

Rebalance触发的时机有：

1. 消费者组中consumer的个数发生变化。例如：有新的consumer加入到消费者组，或者是某个consumer停止了。

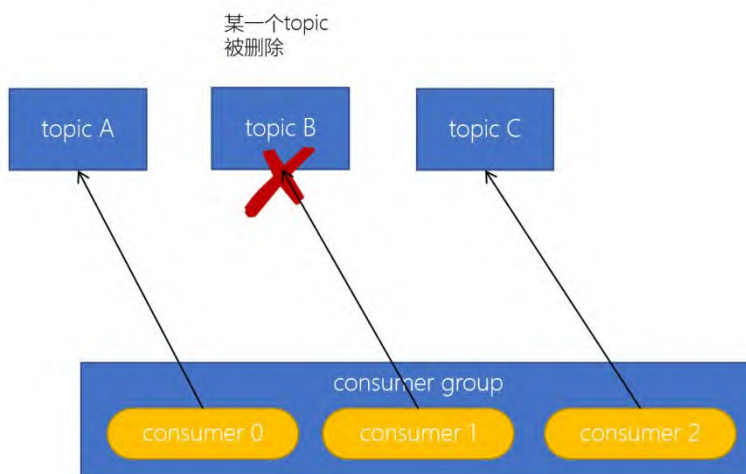
rebalance - 触发时机



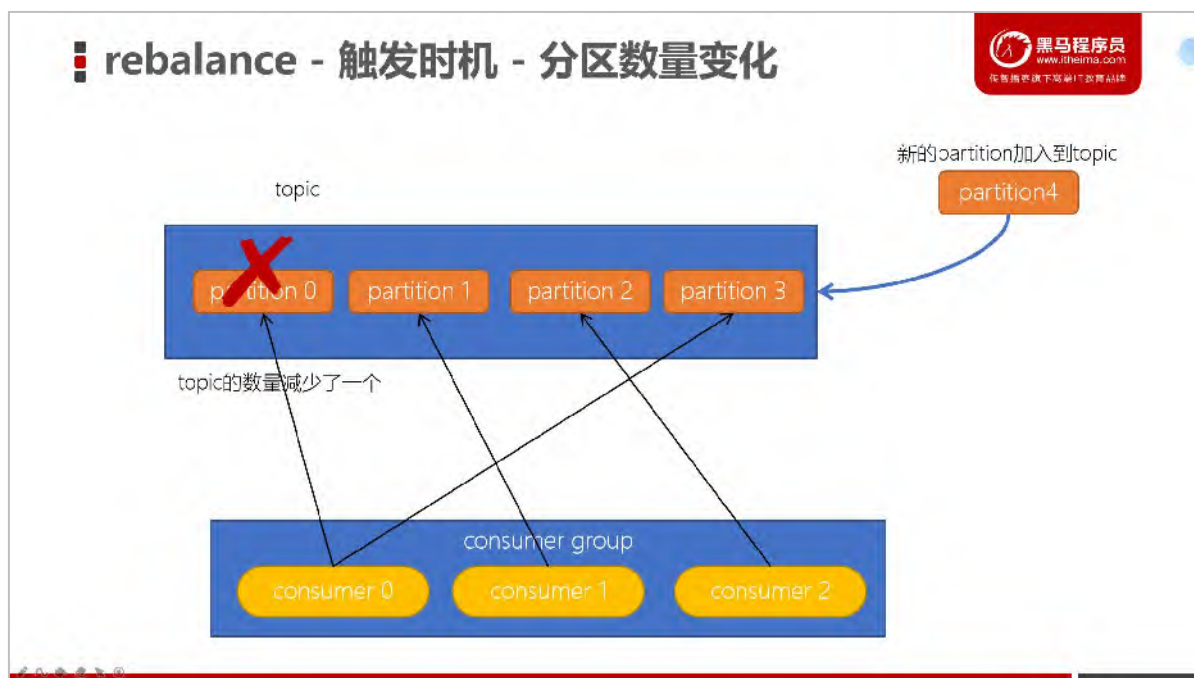
2. 订阅的topic个数发生变化

消费者可以订阅多个主题，假设当前的消费者组订阅了三个主题，但有一个主题突然被删除了，此时也需要发生再均衡。

rebalance - 触发时机 - 订阅主题数量变化



3. 订阅的topic分区数发生变化



1.2.2 Rebalance的不良影响

- 发生Rebalance时，consumer group下的所有consumer都会协调在一起共同参与，Kafka使用分配策略尽可能达到最公平的分配
- Rebalance过程会对consumer group产生非常严重的影响，Rebalance的过程中所有的消费者都将停止工作，直到Rebalance完成

1.3 消费者分区分配策略

1.3.1 Range范围分配策略

Range范围分配策略是Kafka默认的分配策略，它可以确保每个消费者消费的分区数量是均衡的。

注意：Range范围分配策略是针对每个Topic的。

配置

配置消费者的partition.assignment.strategy为
org.apache.kafka.clients.consumer.RangeAssignor。

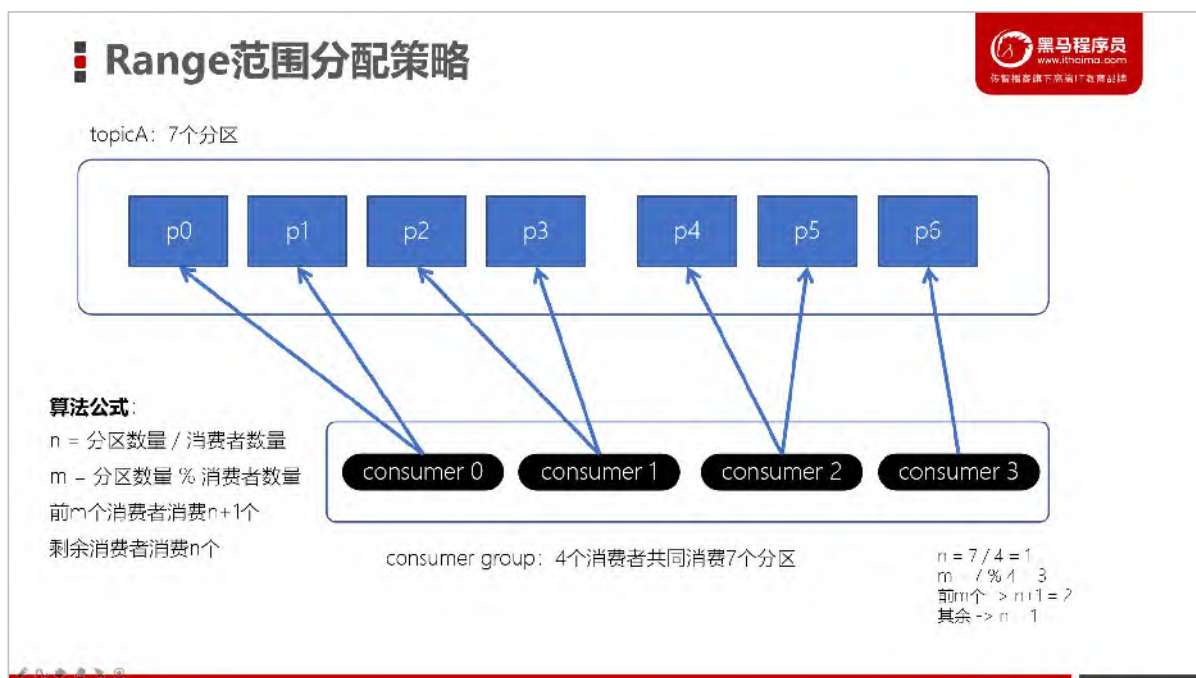
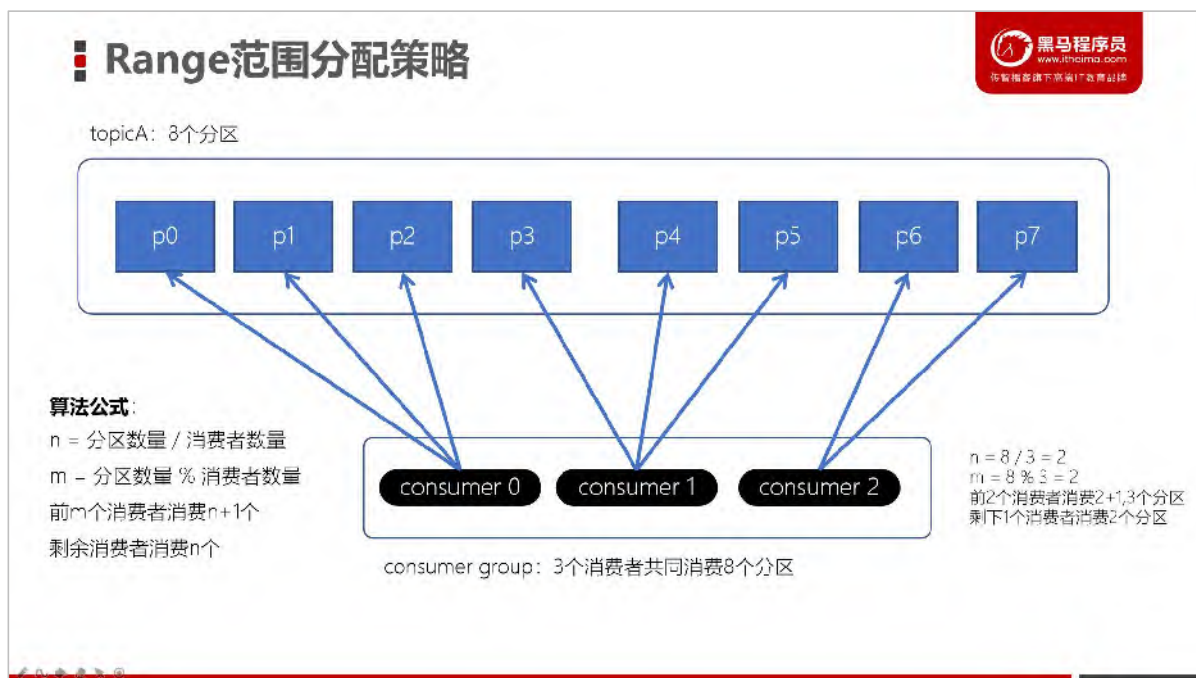
算法公式

$n = \text{分区数量} / \text{消费者数量}$

$m = \text{分区数量} \% \text{消费者数量}$

前m个消费者消费n+1个

剩余消费者消费n个



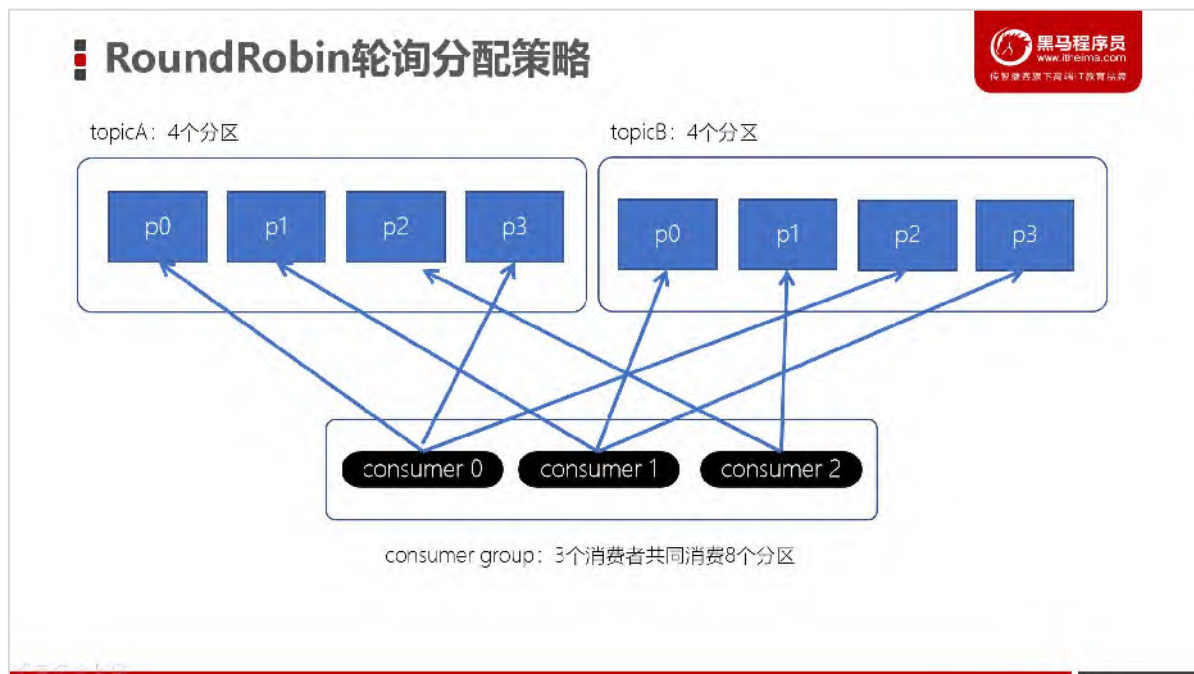
1.3.2 RoundRobin轮询策略

RoundRobinAssignor轮询策略是将消费组内所有消费者以及消费者所订阅的所有topic的partition

按照字典序排序（topic和分区的hashcode进行排序），然后通过轮询方式逐个将分区以此分配给每个消费者。

配置

配置消费者的partition.assignment.strategy为
org.apache.kafka.clients.consumer.RoundRobinAssignor。

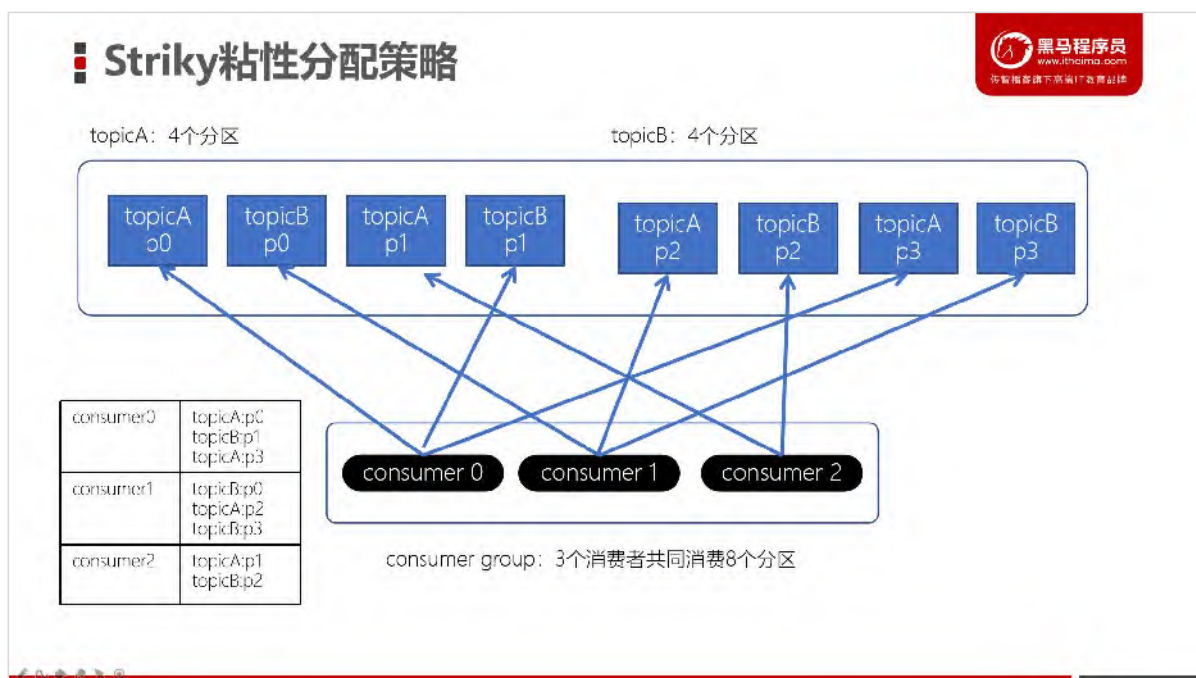


1.3.3 Stricky粘性分配策略

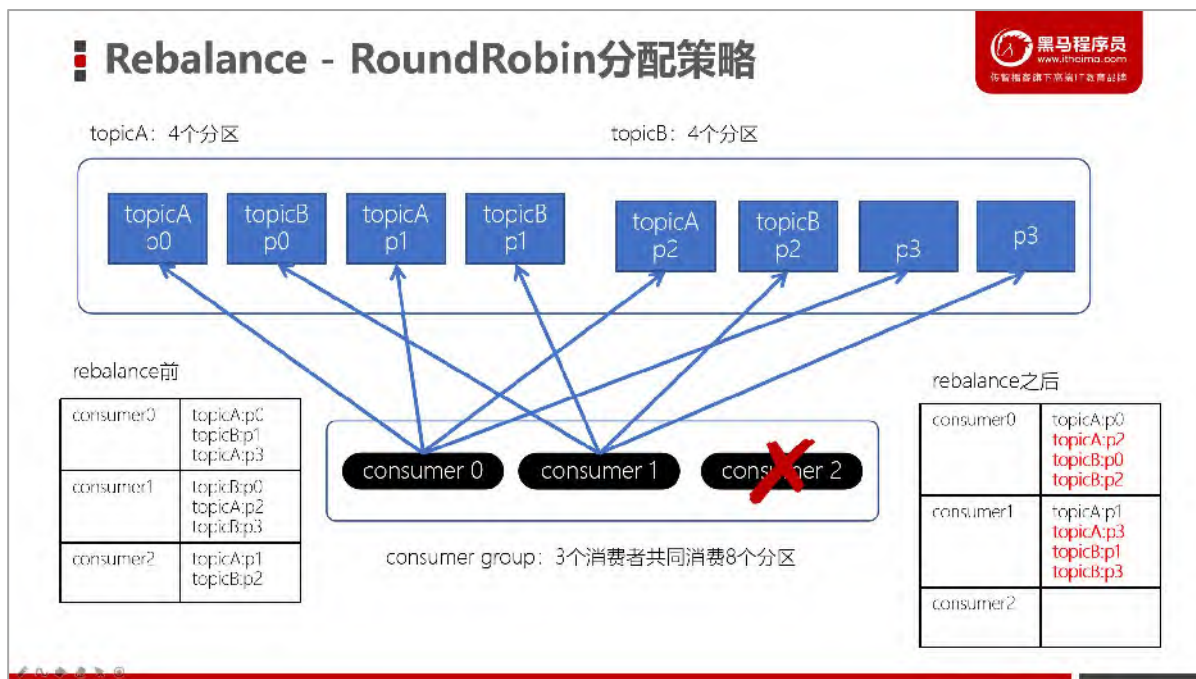
从Kafka 0.11.x开始，引入此类分配策略。主要目的：

1. 分区分配尽可能均匀
2. 在发生rebalance的时候，分区的分配尽可能与上一次分配保持相同

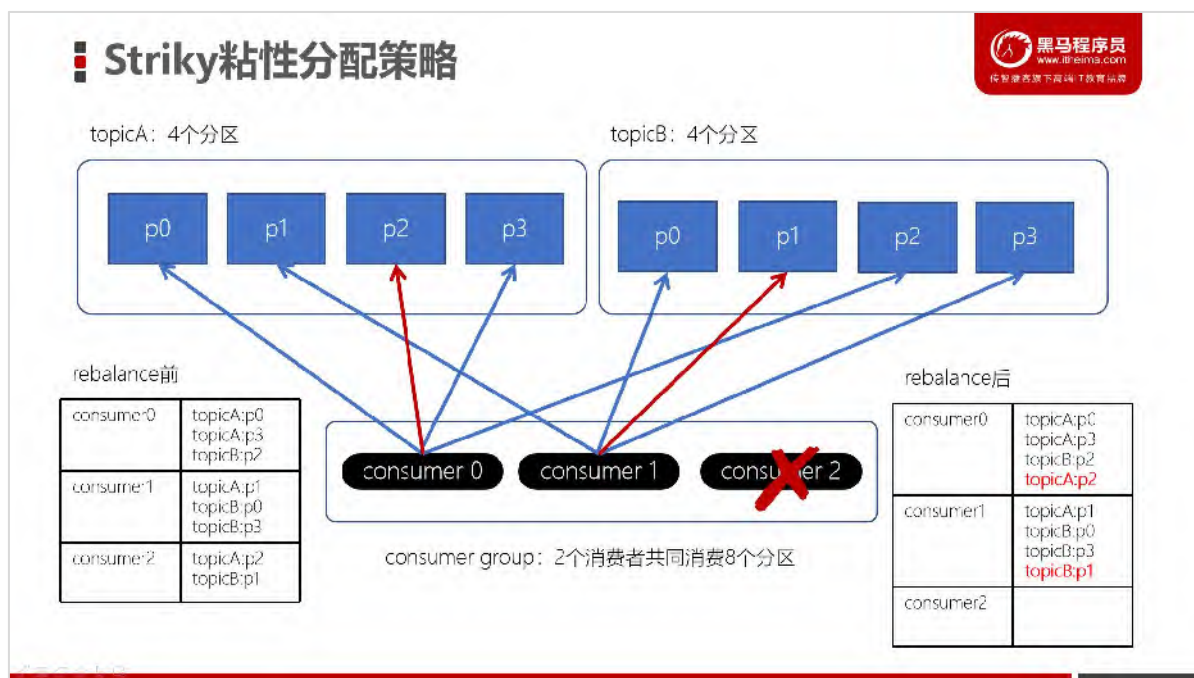
没有发生rebalance时，Stricky粘性分配策略和RoundRobin分配策略类似。



上面如果consumer2崩溃了，此时需要进行rebalance。如果是Range分配和轮询分配都会重新进行分配，例如：



通过上图，我们发现，consumer0和consumer1原来消费的分区大多发生了改变。接下来我们再看下粘性分配策略。



我们发现，Striky粘性分配策略，保留rebalance之前的分配结果。这样，只是将原先consumer2负责的两个分区再均匀分配给consumer0、consumer1。这样可以明显减少系统资源的浪费，例如：之前consumer0、consumer1之前正在消费某几个分区，但由于rebalance发生，导致consumer0、consumer1需要重新消费之前正在处理的分区，导致不必要的系统开销。（例如：某个事务正在进行就必须取消掉了）

1.4 副本机制

副本的目的就是冗余备份，当某个Broker上的分区数据丢失时，依然可以保障数据可用。因为在其他的Broker上的副本是可用的。

1.4.1 producer的ACKs参数

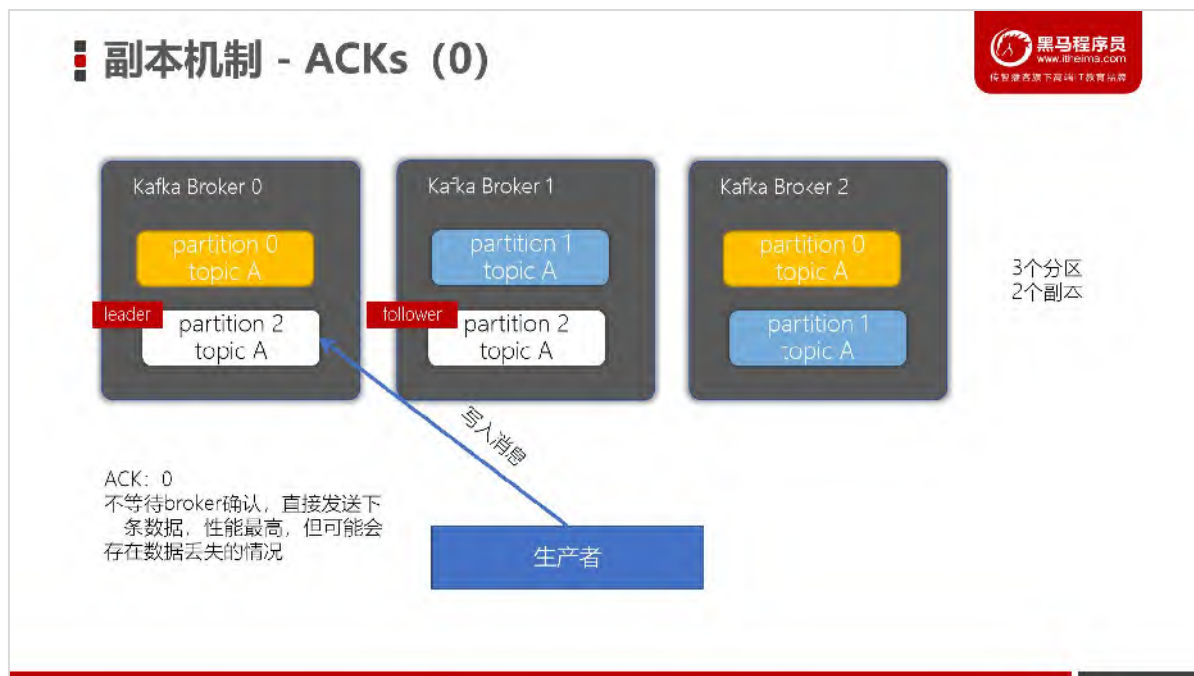
对副本关系较大的就是，producer配置的acks参数了，acks参数表示当生产者生产消息的时候，写入到副本的要求严格程度。它决定了生产者如何在性能和可靠性之间做取舍。

配置：

```
Properties props = new Properties();  
props.put("bootstrap.servers", "node1.itcast.cn:9092");  
props.put("acks", "all");
```

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

1.4.2 acks配置为0



ACK为0，基准测试：

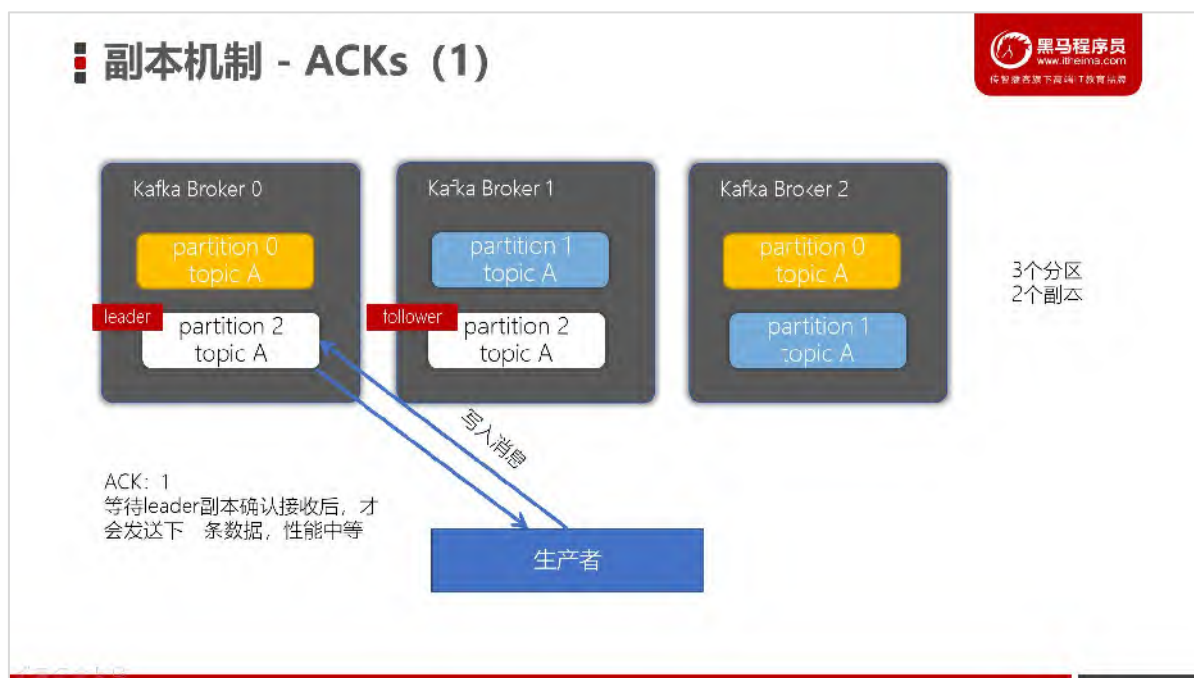
```
bin/kafka-producer-perf-test.sh --topic benchmark --num-records 5000000 --throughput -1  
--record-size 1000 --producer-props  
bootstrap.servers=node1.itcast.cn:9092,node2.itcast.cn:9092,node3.itcast.cn:9092 acks=0
```

测试结果：

指标	单分区单副本 (ack=0)	单分区单副本(ack=1)
吞吐量	165875.991109 records/sec 每秒16.5W条记录	93092.533979 records/sec 每秒9.3W条记录
吞吐速率	158.19 MB/sec 每秒约160MB数据	88.78 MB/sec 每秒约89MB数据

平均延迟时间	192.43 ms avg latency	346.62 ms avg latency
最大延迟时间	670.00 ms max latency	1003.00 ms max latency

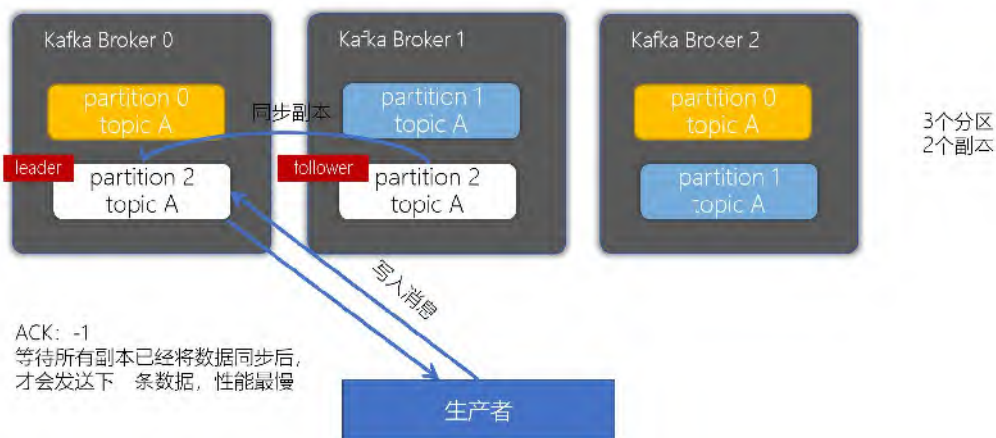
1.4.3 acks配置为1



当生产者的ACK配置为1时，生产者会等待leader副本确认接收后，才会发送下一条数据，性能中等。

1.4.4 acks配置为-1或者all

副本机制 - ACKs (-1)



```
bin/kafka-producer-perf-test.sh --topic benchmark --num-records 5000000 --throughput -1  
--record-size 1000 --producer-props  
bootstrap.servers=node1.itcast.cn:9092,node2.itcast.cn:9092,node3.itcast.cn:9092 acks=all
```

指标	单分区单副本 (ack=0)	单分区单副本(ack=1)	单分区单副本 (ack=-1/all)
吞吐量	165875.991109/s 每秒16.5W条记录	93092.533979/s 每秒9.3W条记录	73586.766156 /s 每秒7.3W条记录
吞吐速率	158.19 MB/sec	88.78 MB/sec	70.18 MB
平均延迟时间	192.43 ms	346.62 ms	438.77 ms
最大延迟时间	670.00 ms	1003.00 ms	1884.00 ms

2. 高级 (High Level) API与低级 (Low Level) API

2.1 高级API

/**

* 消费者程序：从test主题中消费数据



```
*/  
public class _2ConsumerTest {  
    public static void main(String[] args) {  
        // 1. 创建Kafka消费者配置  
        Properties props = new Properties();  
        props.setProperty("bootstrap.servers", "192.168.88.100:9092");  
        props.setProperty("group.id", "test");  
        props.setProperty("enable.auto.commit", "true");  
        props.setProperty("auto.commit.interval.ms", "1000");  
        props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
        props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
  
        // 2. 创建Kafka消费者  
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);  
  
        // 3. 订阅要消费的主题  
        consumer.subscribe(Arrays.asList("test"));  
  
        // 4. 使用一个while循环，不断从Kafka的topic中拉取消息  
        while (true) {  
            // 定义100毫秒超时  
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));  
            for (ConsumerRecord<String, String> record : records)  
                System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());  
        }  
    }  
}
```

- 上面是之前编写的代码，消费Kafka的消息很容易实现，写起来比较简单
- 不需要执行去管理offset，直接通过ZK管理；也不需要管理分区、副本，由Kafka统一管理
- 消费者会自动根据上一次在ZK中保存的offset去接着获取数据
- 在ZK中，不同的消费者组（group）同一个topic记录不同的offset，这样不同程序读取同一个topic，不会受offset的影响

高级API的缺点

- 不能控制offset，例如：想从指定的位置读取
- 不能细化控制分区、副本、ZK等

2.2 低级API

通过使用低级API，我们可以自己来控制offset，想从哪儿读，就可以从哪儿读。而且，可以自己控制连接分区，对分区自定义负载均衡。而且，之前offset是自动保存在ZK中，使用低级API，我们可以将offset不一定要使用ZK存储，我们可以自己来存储offset。例如：存储在文件、MySQL、或者内存中。但是低级API，比较复杂，需要执行控制offset，连接到哪个分区，并找到分区的leader。

2.3 手动消费分区数据

之前的代码，我们让Kafka根据消费组中的消费者动态地为topic分配要消费的分区。但在某些时候，我们需要指定要消费的分区，例如：

- 如果某个程序将某个指定分区的数据保存到外部存储中，例如：Redis、MySQL，那么保存数据的时候，只需要消费该指定的分区数据即可
- 如果某个程序是高可用的，在程序出现故障时将自动重启(例如：后面我们将学习的Flink、Spark程序)。这种情况下，程序将从指定的分区重新开始消费数据。

如何进行手动消费分区中的数据呢？

1. 不再使用之前的 subscribe 方法订阅主题，而使用 「assign」 方法指定想要消费的消息

```
String topic = "test";

TopicPartition partition0 = new TopicPartition(topic, 0);

TopicPartition partition1 = new TopicPartition(topic, 1);

consumer.assign(Arrays.asList(partition0, partition1));
```

2. 一旦指定了分区，就可以就像前面的示例一样，在循环中调用「poll」方法消费消息

注意

1. 当手动管理消费分区时，即使GroupID是一样的，Kafka的组协调器都将不再起作用
2. 如果消费者失败，也将不再自动进行分区重新分配

3. 监控工具Kafka-eagle介绍



3.1 Kafka-Eagle简介

在开发工作中，当业务前提不复杂时，可以使用Kafka命令来进行一些集群的管理工作。但如果业务变得复杂，例如：我们需要增加group、topic分区，此时，我们再使用命令行就感觉很不方便，此时，如果使用一个可视化的工具帮助我们完成日常的管理工作，将会大大提高对于Kafka集群管理的效率，而且我们使用工具来监控消费者在Kafka中消费情况。

早期，要监控Kafka集群我们可以使用Kafka Monitor以及Kafka Manager，但随着我们对监控的功能要求、性能要求的提高，这些工具已经无法满足。

Kafka Eagle是一款结合了目前大数据Kafka监控工具的特点，重新研发的一块开源免费的Kafka集群优秀的监控工具。它可以非常方便的监控生产环境中的offset、lag变化、partition分布、owner等。

官网地址：<https://www.kafka-eagle.org/>

3.2 安装Kafka-Eagle

3.2.1 开启Kafka JMX端口

3.2.1.1 JMX接口

JMX(Java Management Extensions)是一个为应用程序植入管理功能的框架。JMX是一套标准的代理和服务，实际上，用户可以在任何Java应用程序中使用这些代理和服务实现管理。很多的一些软

件都提供了JMX接口，来实现一些管理、监控功能。

3.2.1.2 开启Kafka JMX

在启动Kafka的脚本前，添加：

```
cd ${KAFKA_HOME}
export JMX_PORT=9988
nohup bin/kafka-server-start.sh config/server.properties &
```

3.2.2 安装Kafka-Eagle

1. 安装JDK，并配置好JAVA_HOME。
2. 将kafka_eagle上传，并解压到 /export/server 目录中。

```
cd /export/software/
tar -xvzf kafka-eagle-bin-1.4.6.tar.gz -C ../server/
cd /export/server/kafka-eagle-bin-1.4.6/
tar -xvzf kafka-eagle-web-1.4.6-bin.tar.gz
cd /export/server/kafka-eagle-bin-1.4.6/kafka-eagle-web-1.4.6
```

3. 配置 kafka_eagle 环境变量。

vim /etc/profile

```
export KE_HOME=/export/server/kafka-eagle-bin-1.4.6/kafka-eagle-web-1.4.6
export PATH=$PATH:$KE_HOME/bin
```

source /etc/profile

4. 配置 kafka_eagle。使用vi打开conf目录下的system-config.properties

vim conf/system-config.properties

```
# 修改第4行，配置kafka集群别名
kafka.eagle.zk.cluster.alias=cluster1
# 修改第5行，配置ZK集群地址
cluster1.zk.list=node1.itcast.cn:2181,node2.itcast.cn:2181,node3.itcast.cn:2181
# 注释第6行
#cluster2.zk.list=xdn10:2181,xdn11:2181,xdn12:2181

# 修改第32行，打开图标统计
```

```
kafka.eagle.metrics.charts=true
kafka.eagle.metrics.retain=30

# 注释第69行，取消sqlite数据库连接配置
#kafka.eagle.driver=org.sqlite.JDBC
#kafka.eagle.url=jdbc:sqlite:/hadoop/kafka-eagle/db/ke.db
#kafka.eagle.username=root
#kafka.eagle.password=www.kafka-eagle.org

# 修改第77行，开启mys
kafka.eagle.driver=com.mysql.jdbc.Driver
kafka.eagle.url=jdbc:mysql://node1.itcast.cn:3306/ke?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
kafka.eagle.username=root
kafka.eagle.password=123456
```

5. 配置JAVA_HOME

```
cd /export/server/kafka-eagle-bin-1.4.6/kafka-eagle-web-1.4.6/bin
vim ke.sh

# 在第24行添加JAVA_HOME环境配置
export JAVA_HOME=/export/server/jdk1.8.0_241
```

6. 修改Kafka eagle可执行权限

```
cd /export/server/kafka-eagle-bin-1.4.6/kafka-eagle-web-1.4.6/bin
chmod +x ke.sh
```

7. 启动 kafka_eagle。

```
./ke.sh start
```

8. 访问Kafka eagle，默认用户为admin，密码为：123456

<http://node1.itcast.cn:8048/ke>





查看ZK状态

3 Zookeepers

View Details



查看消费者组

0 Groups

View Details

3.3 Kafka度量指标

3.3.1 topic list

点击Topic下的List菜单，就可以展示当前Kafka集群中的所有topic。

Dashboard
Topic
Create
List
KSQL
Mock
Manager
Consumers
Cluster
Metrics
Alarm
System
BScreen

Topic list

List all topic information.

- Broker Spread: the higher the coverage, the higher the resource usage of kafka broker nodes.
- Broker Skewed: the larger the skewed, the higher the pressure on the broker node of kafka.
- Broker Leader Skewed: the higher the leader skewed, the higher the pressure on the kafka broker leader node.

Topic List Info

ID	Topic Name	Partitions	Broker Spread	Broker Skewed	Broker Leader Skewed
1	topic_view	3	100%	0%	33%
2	didit_log	1	33%	0%	0%
3	test	20	100%	0%	0%
4	__transaction_state	50	100%	0%	0%
5	dwd_user	1	33%	0%	0%
6	test_by_kafka_tool	1	33%	0%	0%
7	ods_user	1	33%	0%	0%
8	benchmark	1	33%	0%	0%

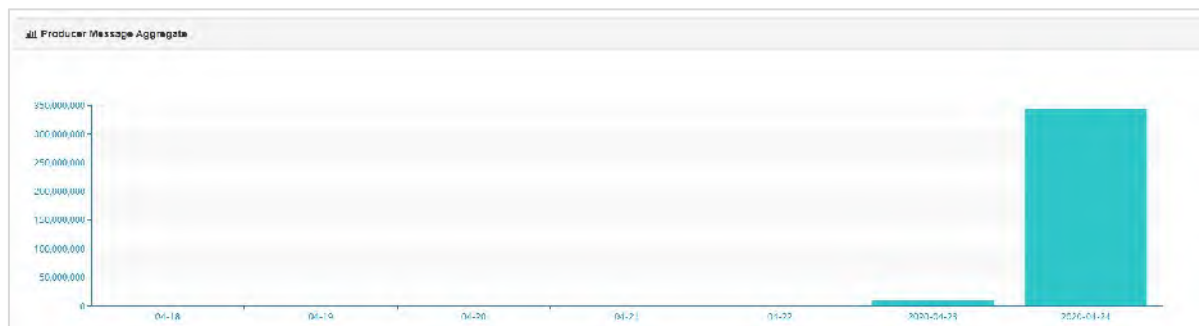
ID	Topic Name	Partitions	Broker Spread	Broker Skewed	Broker Leader Skewed
1	wordcount	1	33%	0%	0%
2	didit_log	3	100%	0%	33%
3	test	3	100%	0%	33%
4	__transaction_state	50	100%	0%	0%
5	test_10m	3	100%	0%	33%
6	dwd_user	1	33%	0%	0%
7	ods_user	1	33%	0%	0%

Showing 1 to 7 of 7 entries

指标	意义
----	----

Brokers Spread	broker使用率
Brokers Skew	分区是否倾斜
Brokers Leader Skew	leader partition是否存在倾斜

3.3.2 生产者消息总计



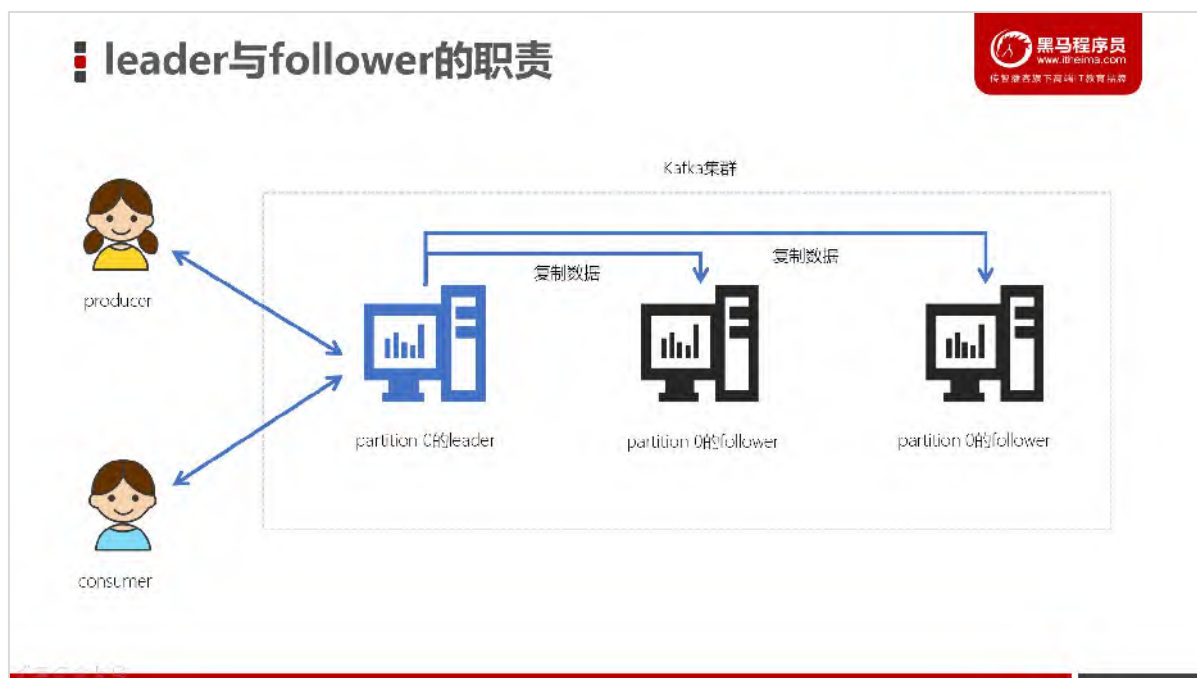
4. Kafka原理

4.1 分区的leader与follower

4.1.1 Leader和Follower

在Kafka中，每个topic都可以配置多个分区以及多个副本。每个分区都有一个leader以及0个或者多个follower，在创建topic时，Kafka会将每个分区的leader均匀地分配在每个broker上。我们正常使用kafka是感觉不到leader、follower的存在的。但其实，所有的读写操作都是由leader处理，而所有的follower都复制leader的日志数据文件，如果leader出现故障时，follower就会被选举为leader。所以，可以这样说：

- **Kafka中的leader负责处理读写操作，而follower只负责副本数据的同步**
- **如果leader出现故障，其他follower会被重新选举为leader**
- **follower像一个consumer一样，拉取leader对应分区的数据，并保存到日志数据文件中**



4.1.2 查看某个partition的leader

使用Kafka-eagle查看某个topic的partition的leader在哪个服务器中。为了方便观察，我们创建一个名为test的3个分区、3个副本的topic。

The screenshot shows the 'Topic create' form in Kafka-eagle. The left sidebar has a 'Create' button highlighted with a red box and an arrow. The form fields are: 'Topic Name (*)' with value 'test', 'Partitions (*)' with value '3', and 'Replication Factor (*)' with value '3'. There are error messages for the topic name and replication factor.

1. 点击「Topic」菜单下的「List」

Topic list

- ① List all topic information.
- ① Broker Spread: the higher the coverage, the higher the resource usage of kafka broker nodes.
- ① Broker Skewed: the larger the skewed, the higher the pressure on the broker node of kafka.
- ① Broker Leader Skewed: the higher the leader skewed, the higher the pressure on the kafka broker leader node.

Topic List Info

ID	Topic Name	Partitions	Broker Spread	Broker Skewed
1	dwd_user	1	33%	0%
2	test	20	100%	0%
3	benchmark	1	33%	0%
4	didi_log	1	33%	0%

2. 任意点击选择一个Topic

Topic Meta Info

Topic	Partition	LogSize	Leader
test	0	4	2
test	1	7	1
test			

分区0的leader在broker2上

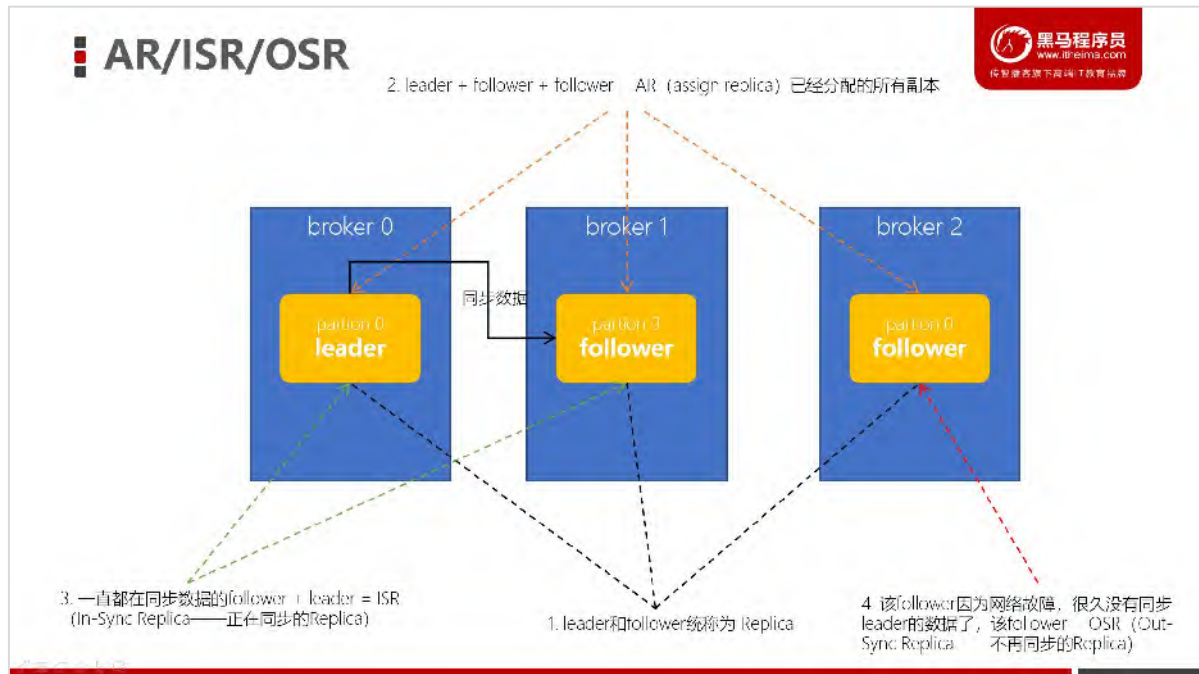
Showing 1 to 3 of 3 entries

4.1.3 AR、ISR、OSR

在实际环境中，leader有可能会出现一些故障，所以Kafka一定会选举出新的leader。在讲解leader选举之前，我们先要明确几个概念。Kafka中，把follower可以按照不同状态分为三类——AR、ISR、OSR。

- 分区的所有副本称为「AR」（Assigned Replicas——已分配的副本）
- 所有与leader副本保持一定程度同步的副本（包括 leader 副本在内）组成「ISR」（In-Sync Replicas——在同步中的副本）

- 由于follower副本同步滞后过多的副本（不包括 leader 副本）组成「OSR」（Out-of-Sync Replias）
- $AR = ISR + OSR$
- 正常情况下，所有的follower副本都应该与leader副本保持同步，即 $AR = ISR$ ，OSR集合为空。



4.1.4 查看分区的ISR

1. 使用Kafka Eagle查看某个Topic的partition的ISR有哪几个节点。

ISR

Topic Meta Info

Topic	Partition	LogSize	Leader	Replicas	In Sync Replicas
test	0	0	0	[0, 2, 1]	[0,2,1]
test	1	0	2	[2, 1, 0]	[2,1,0]
test	2	0	1	[1, 0, 2]	[1,0,2]

Showing 1 to 3 of 3 entries

2. 尝试关闭id为0的broker（杀掉该broker的进程），参看topic的ISR情况。

Topic Meta Info					
Topic	Partition	Log Size	Leader	Replicas	In Sync Replicas
test	0	0	2	[0, 2, 1]	[1,2]
test	1	0	2	[2, 1, 0]	[1,2]
test	2	0	1	[1, 0, 2]	[1,2]

Showing 1 to 3 of 3 entries

broker0已经不在ISR中

4.1.5 Leader选举

leader对于消息的写入以及读取是非常关键的，此时有两个疑问：

1. Kafka如何确定某个partition是leader、哪个partition是follower呢？
2. 某个leader崩溃了，如何快速确定另外一个leader呢？因为Kafka的吞吐量很高、延迟很低，所以选举leader必须非常快

4.1.5.1 如果leader崩溃，Kafka会如何？

使用Kafka Eagle找到某个partition的leader，再找到leader所在的broker。在Linux中强制杀掉该Kafka的进程，然后观察leader的情况。

Topic Meta Info					
Topic	Partition	Log Size	Leader	Replicas	In Sync Replicas
test	0	0	2	[0, 2, 1]	[2,1]
test	1	0	2	[2, 1, 0]	[2,1]
test	2	0	1	[1, 0, 2]	[1,2]

Showing 1 to 3 of 3 entries

重新选举2为leader

通过观察，我们发现，leader在崩溃后，Kafka又从其他的follower中快速选举出来了leader。

4.1.5.2 Controller介绍

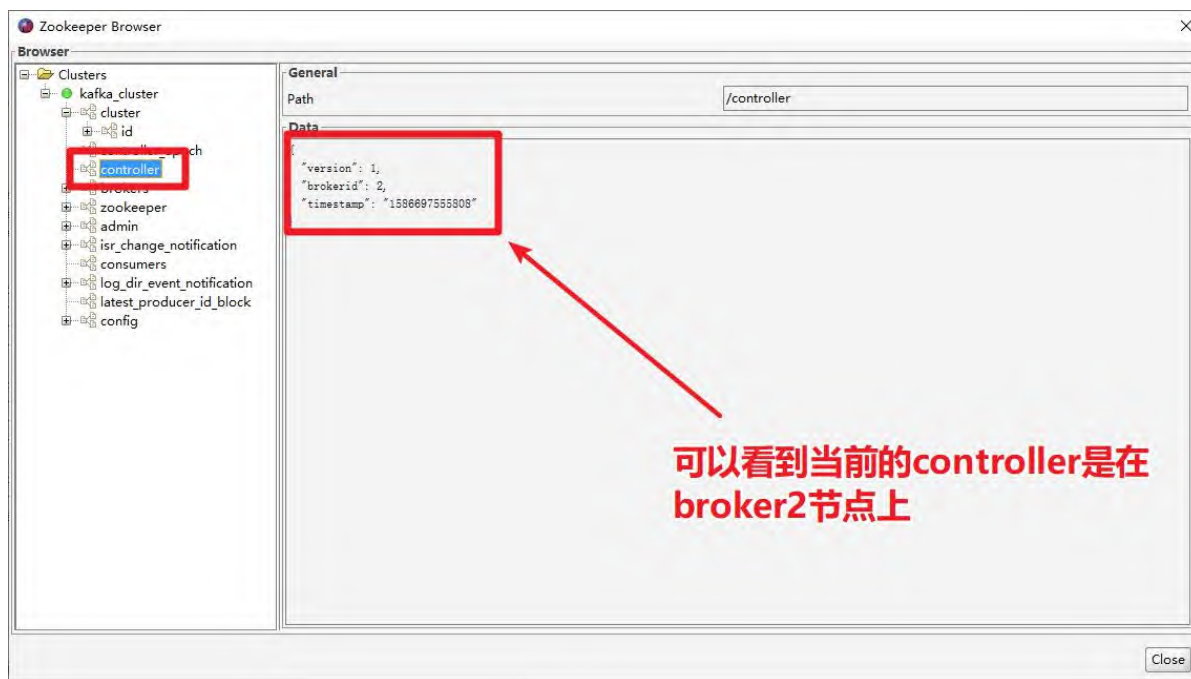
- Kafka启动时，会在所有的broker中选择一个controller
- 前面leader和follower是针对partition，而controller是针对broker的
- 创建topic、或者添加分区、修改副本数量之类的管理任务都是由controller完成的
- Kafka分区leader的选举，也是由controller决定的

4.1.5.3 Controller的选举

- 在Kafka集群启动的时候，每个broker都会尝试去ZooKeeper上注册成为Controller（ZK临时节点）
- 但只有一个竞争成功，其他的broker会注册该节点的监视器
- 一点该临时节点状态发生变化，就可以进行相应的处理
- Controller也是高可用的，一旦某个broker崩溃，其他的broker会重新注册为Controller

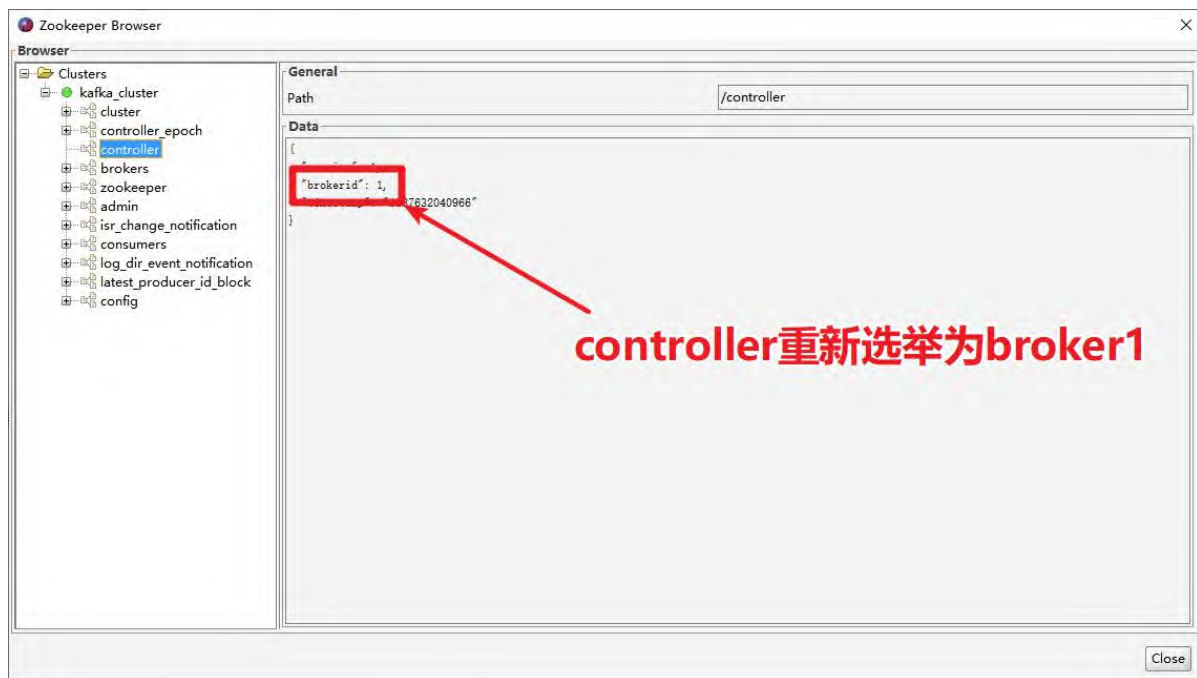
4.1.5.4 找到当前Kafka集群的controller

1. 点击Kafka Tools的「Tools」菜单，找到「ZooKeeper Brower...」
2. 点击左侧树形结构的controller节点，就可以查看到哪个broker是controller了。



4.1.5.5 测试controller选举

通过kafka tools找到controller所在的broker对应的kafka进程，杀掉该进程，重新打开ZooKeeper brower，观察kafka是否能够选举出来新的Controller。



4.1.5.6 Controller选举partition leader

- 所有Partition的leader选举都由controller决定
- controller会将leader的改变直接通过RPC的方式通知需为此作出响应的Broker
- controller读取到当前分区的ISR，只要有一个Replica还幸存，就选择其中一个作为leader否则，则任意选这个一个Replica作为leader
- 如果该partition的所有Replica都已经宕机，则新的leader为-1

为什么不能通过ZK的方式来选举partition的leader?

- Kafka集群如果业务很多的情况下，会有很多的partition
- 假设某个broker宕机，就会出现很多的partition都需要重新选举leader
- 如果使用zookeeper选举leader，会给zookeeper带来巨大的压力。所以，kafka中leader的选举不能使用ZK来实现

4.1.6 leader负载均衡

4.1.6.1 Preferred Replica

- Kafka中引入了一个叫做「preferred-replica」的概念，意思就是：优先的Replica

- 在ISR列表中，第一个replica就是preferred-replica
- 第一个分区存放的broker，肯定就是preferred-replica
- 执行以下脚本可以将preferred-replica设置为leader，均匀分配每个分区的leader。

```
./kafka-leader-election.sh --bootstrap-server node1.itcast.cn:9092 --topic 主题 --partition=1  
--election-type preferred
```

4.1.6.2 确保leader在broker中负载均衡

杀掉test主题的某个broker，这样kafka会重新分配leader。等到Kafka重新分配leader之后，再次启动kafka进程。此时：观察test主题各个分区leader的分配情况。

Topic Meta Info					
Topic	Partition	Log Size	Leader		In Sync Replicas
test	0	0	0	broker1上分配了两个leader	[0, 2, 1]
test	1	0	1		[2, 1, 0]
test	2	0	1		[1, 0, 2]

Showing 1 to 3 of 3 entries

此时，会造成leader分配是不均匀的，所以可以执行以下脚本来重新分配leader:

```
bin/kafka-leader-election.sh --bootstrap-server node1.itcast.cn:9092 --topic test  
--partition=2 --election-type preferred
```

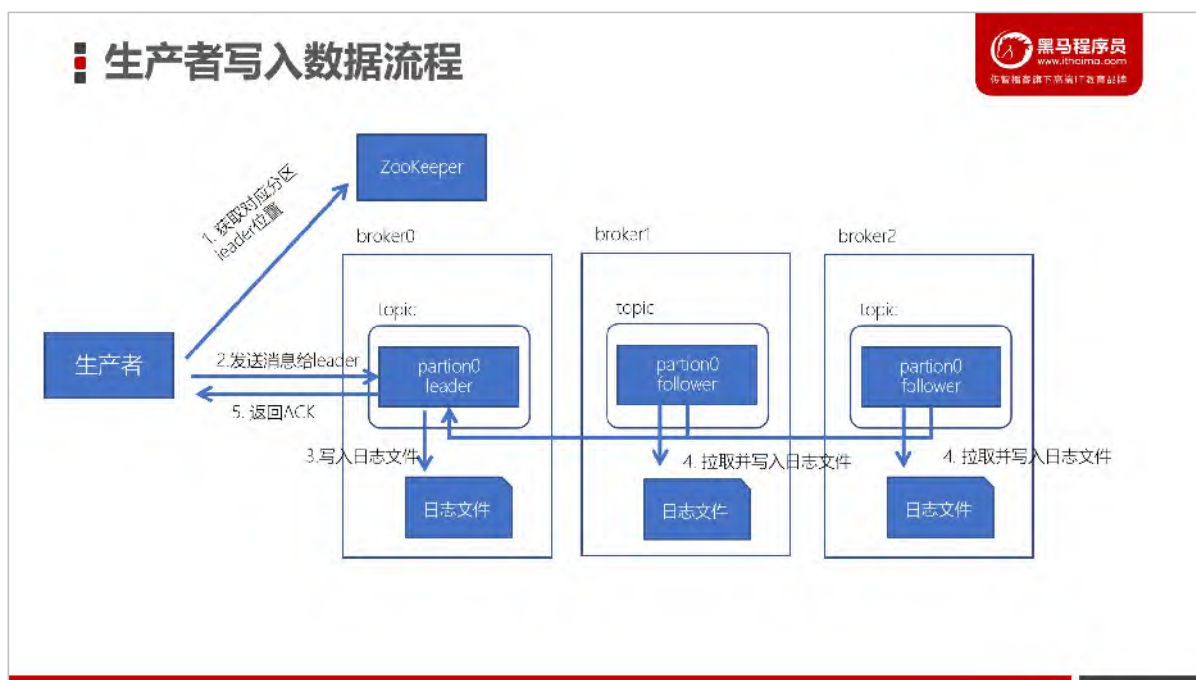
--partition: 指定需要重新分配leader的partition编号

Topic Meta Info					
Topic	Partition	Log Size	Leader	Replicas	
test	0	0	0	再刷新，leader已经重新分配了	[0, 2, 1]
test	1	0	2		[2, 1, 0]
test	2	0	1		[1, 0, 2]

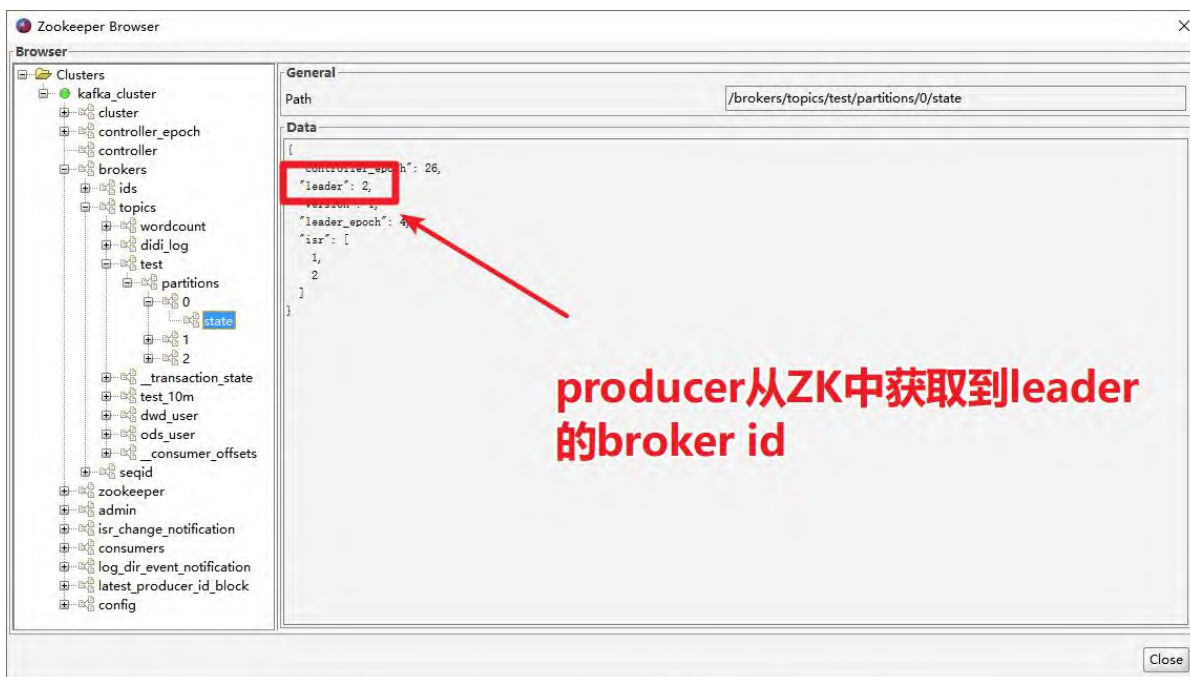
Showing 1 to 3 of 3 entries

4.2 Kafka生产、消费数据工作流程

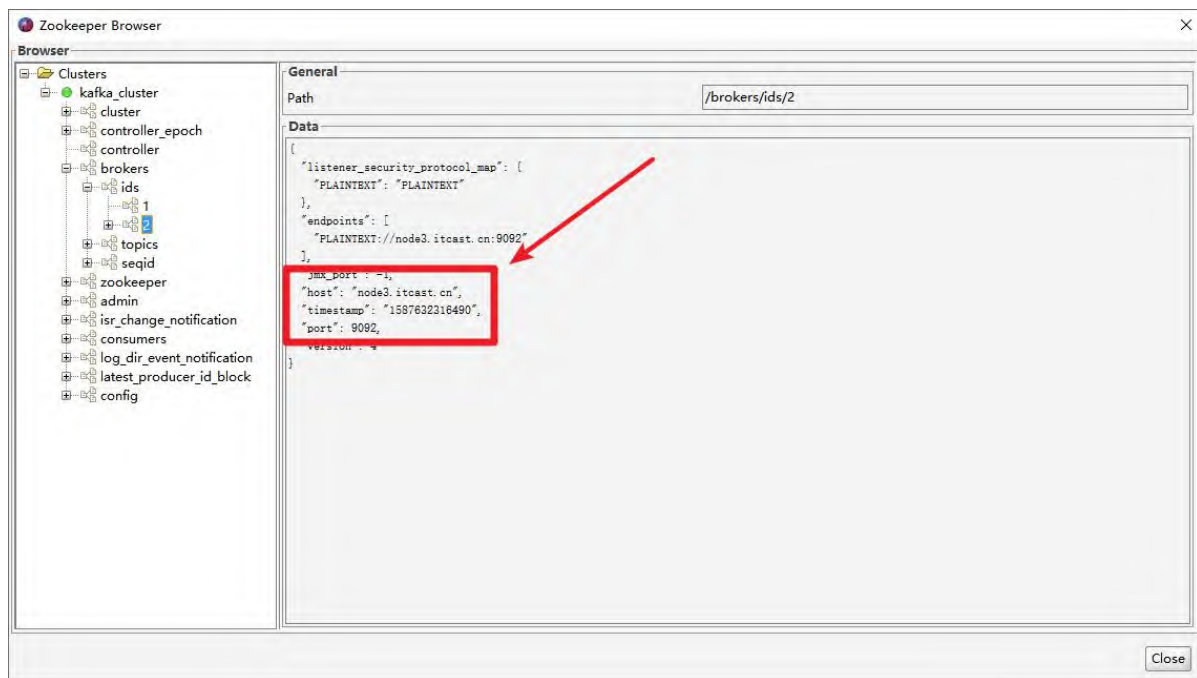
4.2.1 Kafka数据写入流程



- 生产者先从 zookeeper 的 `/brokers/topics/主题名/partitions/分区名/state` 节点找到该 partition 的 leader



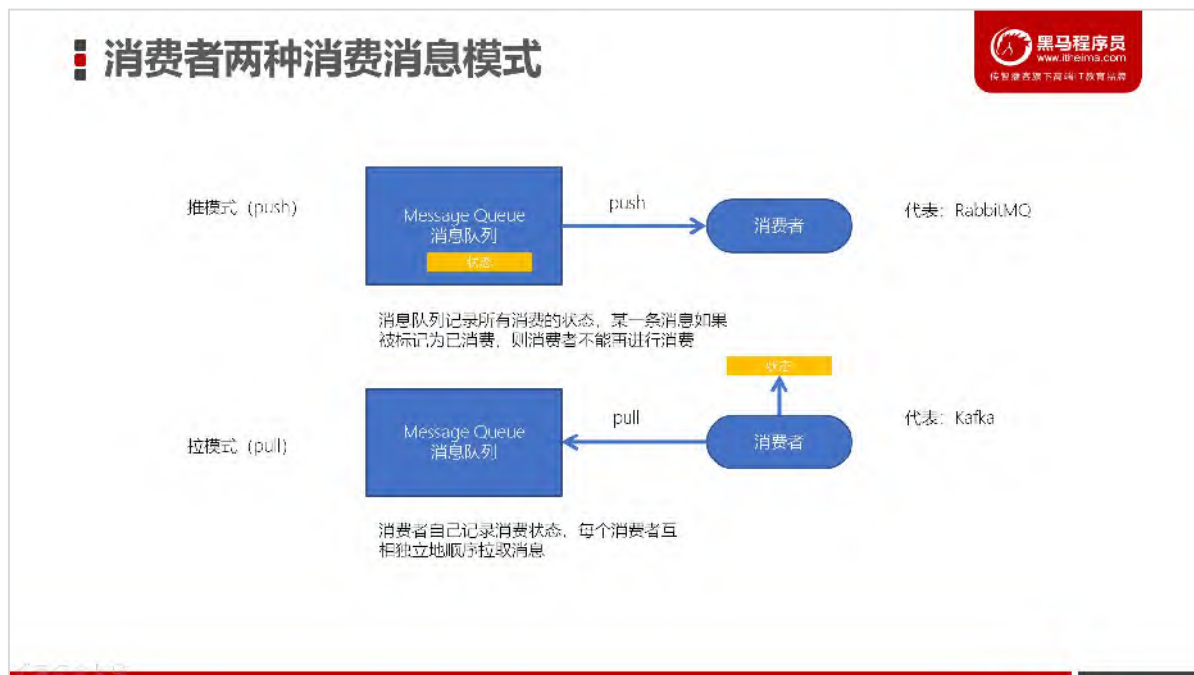
- 生产者在ZK中找到该ID找到对应的broker



- broker进程上的leader将消息写入到本地log中
- follower从leader上拉取消息，写入到本地log，并向leader发送ACK
- leader接收到所有的ISR中的Replica的ACK后，并向生产者返回ACK。

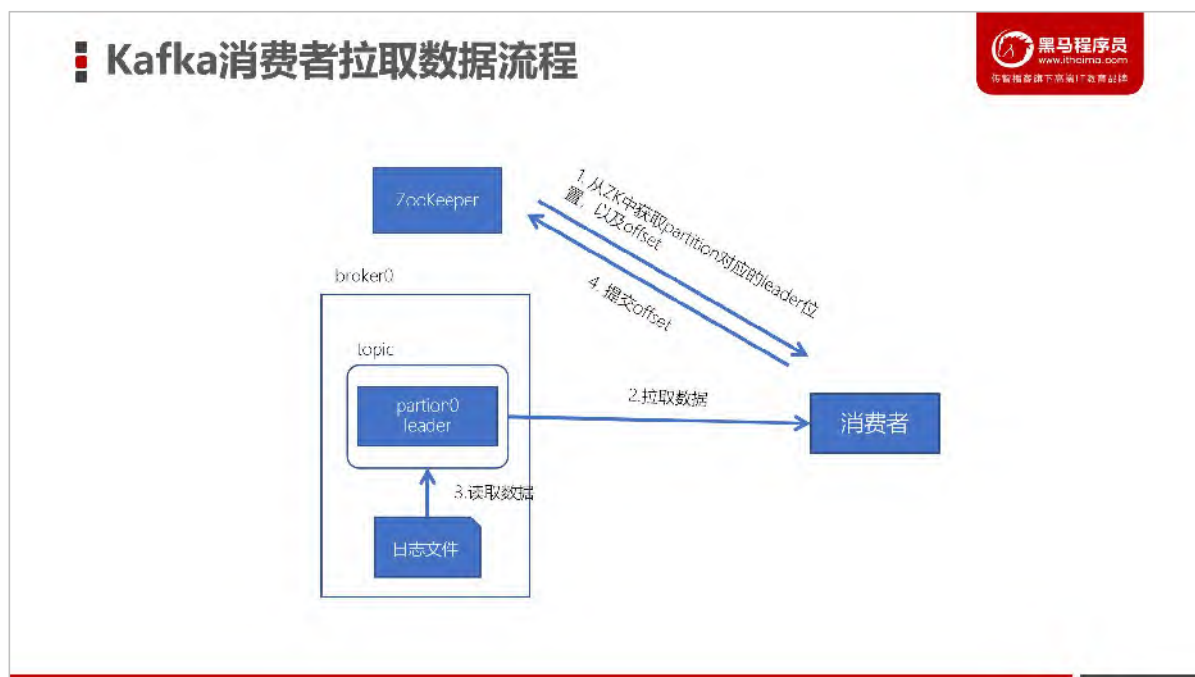
4.2.2 Kafka数据消费流程

4.2.2.1 两种消费模式



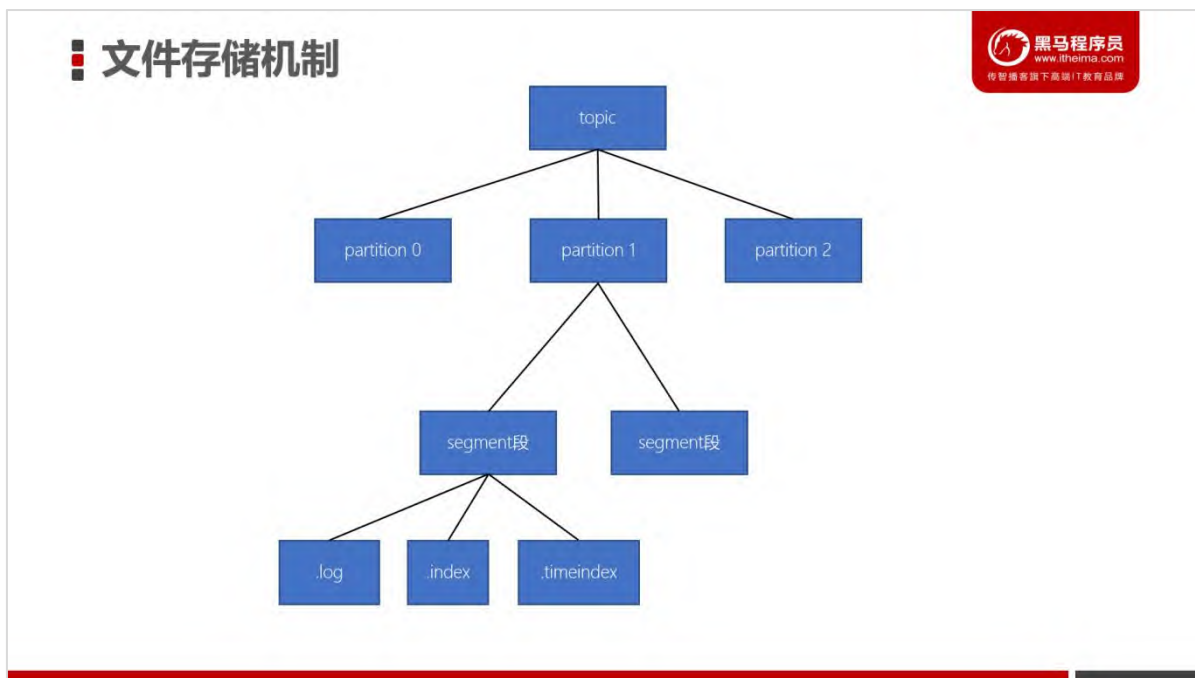
- kafka采用拉取模型, 由消费者自己记录消费状态, 每个消费者互相独立地顺序拉取每个分区的信息
- 消费者可以按照任意的顺序消费消息。比如, 消费者可以重置到旧的偏移量, 重新处理之前已经消费过的消息; 或者直接跳到最近的位置, 从当前的时刻开始消费。

4.2.2.2 Kafka消费数据流程



- 每个consumer都可以根据分配策略（默认RangeAssignor），获得要消费的分区
- 获取到consumer对应的offset（默认从ZK中获取上一次消费的offset）
- 找到该分区的leader，拉取数据
- 消费者提交offset

4.3 Kafka的数据存储形式



- 一个topic由多个分区组成
- 一个分区 (partition) 由多个segment (段) 组成
- 一个segment (段) 由多个文件组成 (log、index、timeindex)

4.3.1 存储日志

接下来，我们来看一下Kafka中的数据到底是如何在磁盘中存储的。

- Kafka中的数据是保存在 /export/server/kafka_2.12-2.4.1/data中
- 消息是保存在以：「主题名-分区ID」的文件夹中的
- 数据文件夹中包含以下内容：

```
-rw-r--r--. 1 root root 10M 4月 9 23:50 00000000000000000000.index
-rw-r--r--. 1 root root 11M 4月 9 23:50 00000000000000000000.log
-rw-r--r--. 1 root root 10M 4月 9 23:50 00000000000000000000.timeindex
-rw-r--r--. 1 root root 8 4月 9 23:43 leader-epoch-checkpoint
```

这些分别对应：

文件名	说明
00000000000000000000.index	索引文件，根据offset查找数据就是通过该索引

- 每个日志文件的文件名为起始偏移量，因为每个分区的起始偏移量是0，所以，分区的日志文件都以00000000000000000000.log开始
- 默认的每个日志文件最大为「log.segment.bytes = 1024*1024*1024」 1G
- 为了简化根据offset查找消息，Kafka日志文件名设计为开始的偏移量

为了方便测试观察，新创建一个topic: 「test 10m」，该topic每个日志数据文件最大为10M

使用之前的生产者程序往「test 10m」主题中生产数据，可以观察到如下：

每个log文件的大小
最大为 10M

```
10 00:26 00000000000000000000.index
10 00:26 00000000000000000000.log
10 00:26 00000000000000000000.timeindex
10 00:26 00000000000000435086.index
10 00:26 00000000000000435086.log
```

该数字说明 00...0.log
文件中有 435086 条消息

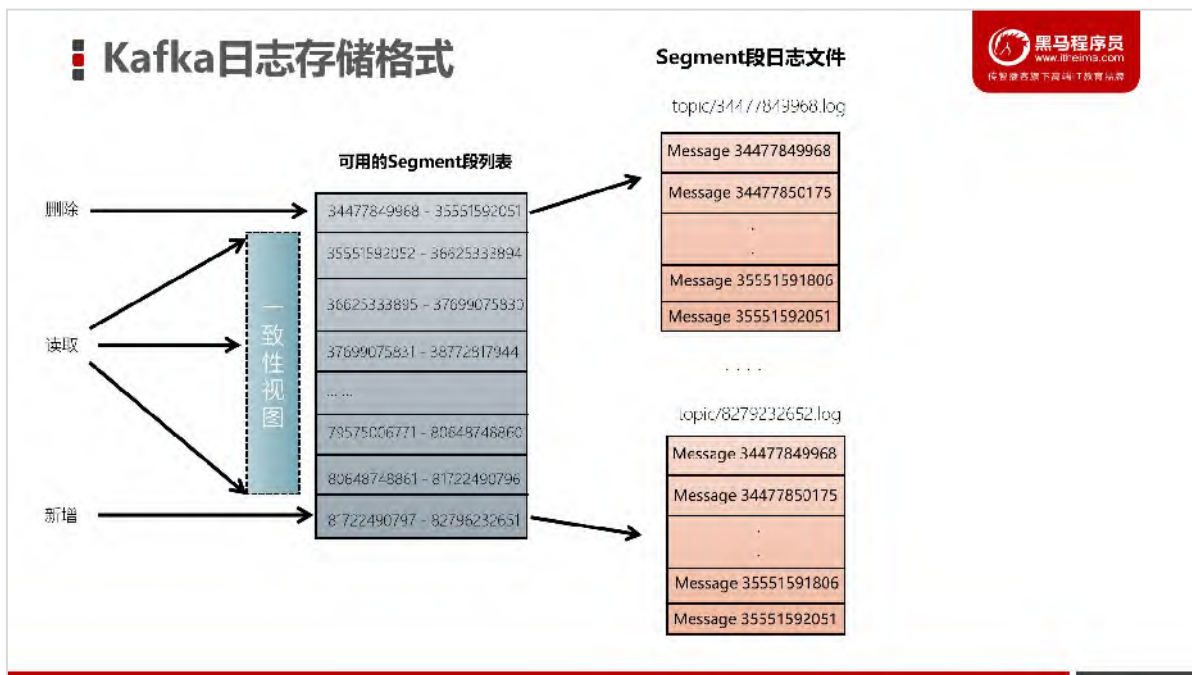
4.3.1.2 写入消息

- 新的消息总是写入到最后的一个日志文件中
- 该文件如果到达指定的大小（默认为：1GB）时，将滚动到一个新的文件中

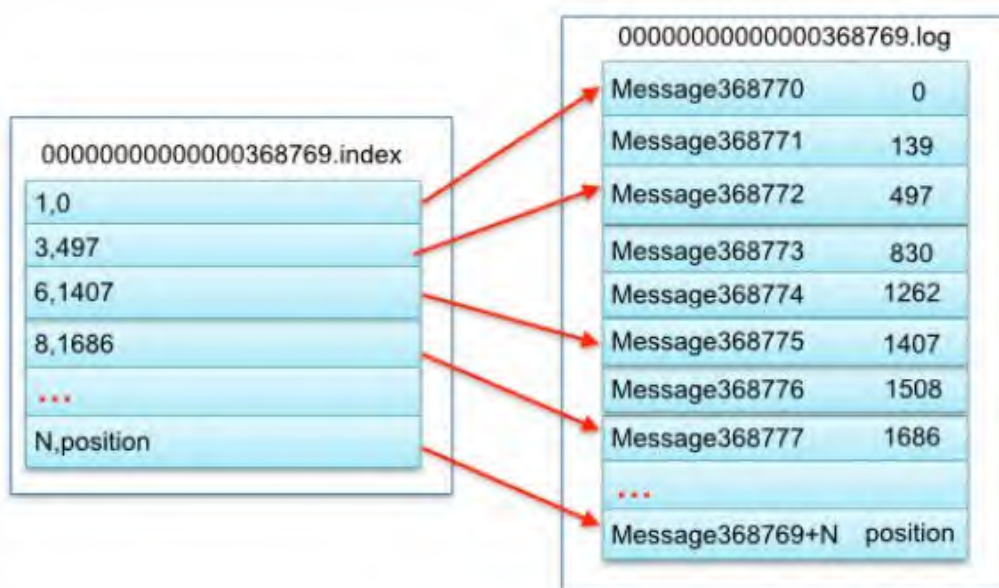
```
5.1K 4月 10 00:26 00000000000000000000.index
10M 4月 10 00:26 00000000000000000000.log
7.5K 4月 10 00:26 00000000000000000000.timeindex
5.0K 4月 10 00:26 00000000000000435086.index
10M 4月 10 00:26 00000000000000435086.log
10 4月 10 00:26 00000000000000435086.snapshot
7.5K 4月 10 00:26 00000000000000435086.timeindex
5.0K 4月 10 00:27 00000000000000869646.index
10M 4月 10 00:27 00000000000000869646.log
10 4月 10 00:26 00000000000000869646.snapshot
7.5K 4月 10 00:27 00000000000000869646.timeindex
5.0K 4月 10 00:27 00000000000001304206.index
10M 4月 10 00:27 00000000000001304206.log
10 4月 10 00:27 00000000000001304206.snapshot
7.5K 4月 10 00:27 00000000000001304206.timeindex
10M 4月 10 00:27 00000000000001738766.index
9.3M 4月 10 00:27 00000000000001738766.log
10 4月 10 00:27 00000000000001738766.snapshot
10M 4月 10 00:27 00000000000001738766.timeindex
```

数据总是写入到最后的
日志文件中

4.3.1.3 读取消息



- 根据「offset」首先需要找到存储数据的 segment 段（注意：offset指定分区的全局偏移量）
- 然后根据这个「全局分区offset」找到相对于文件的「segment段offset」



- 最后再根据 「segment段offset」 读取消息
- 为了提高查询效率，每个文件都会维护对应的范围内存，查找的时候就是使用简单的二分查找

4.3.1.4 删除消息

- 在Kafka中，消息是会被**定期清理**的。一次删除一个segment段的日志文件
- Kafka的日志管理器，会根据Kafka的配置，来决定哪些文件可以被删除

4.4 消息不丢失机制

4.4.1 broker数据不丢失

生产者通过分区的leader写入数据后，所有在ISR中follower都会从leader中复制数据，这样，可以确保即使leader崩溃了，其他的follower的数据仍然是可用的

4.4.2 生产者数据不丢失

- 生产者连接leader写入数据时，可以通过ACK机制来确保数据已经成功写入。ACK机制有三个可选配置
 1. 配置ACK响应要求为 -1 时 —— 表示所有的节点都收到数据(leader和follower都接收到数据)
 2. 配置ACK响应要求为 1 时 —— 表示leader收到数据
 3. 配置ACK影响要求为 0 时 —— 生产者只负责发送数据，不关心数据是否丢失（这种情况可能会产生数据丢失，但性能是最好的）
- 生产者可以采用同步和异步两种方式发送数据
 - 同步：发送一批数据给kafka后，等待kafka返回结果
 - 异步：发送一批数据给kafka，只是提供一个回调函数。

说明：如果broker迟迟不给ack，而buffer又满了，开发者可以设置是否直接清空buffer中的数据。

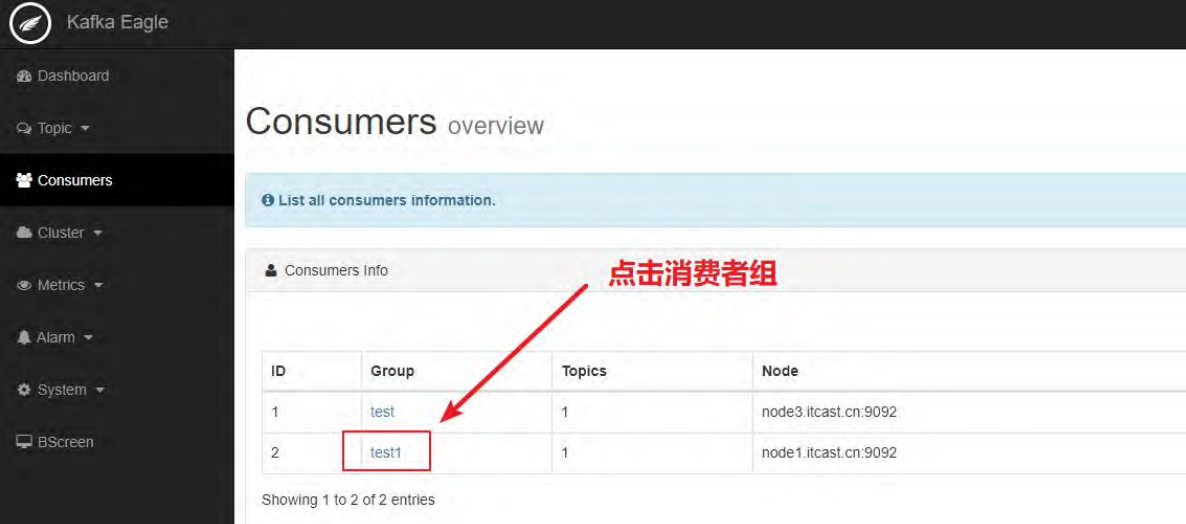
4.4.3 消费者数据不丢失

在消费者消费数据的时候，只要每个消费者记录好offset值即可，就能保证数据不丢失。

4.5 数据积压

Kafka消费者消费数据的速度是非常快的，但如果由于处理Kafka消息时，由于有一些外部IO、或者是产生网络拥堵，就会造成Kafka中的数据积压（或称为数据堆积）。如果数据一直积压，会导致数据出来的实时性受到较大影响。

4.5.1 使用Kafka-Eagle查看数据积压情况

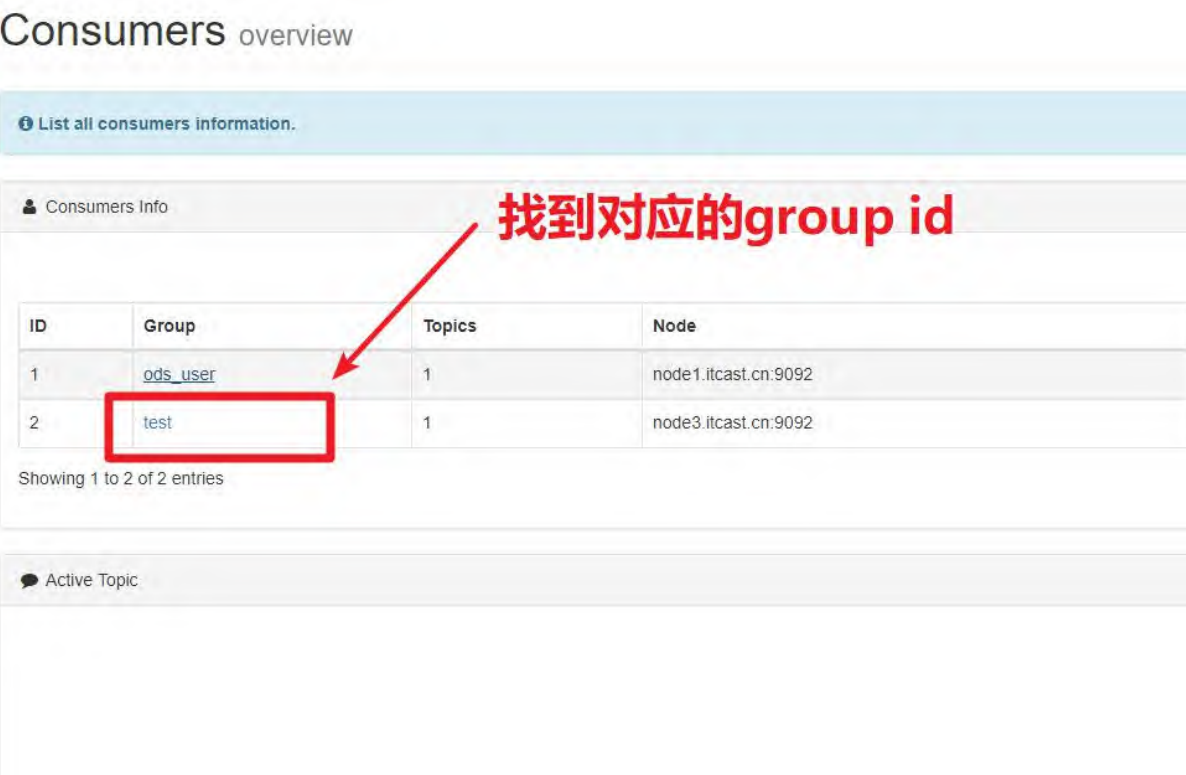


Kafka Eagle interface showing Consumers overview. The table lists consumer groups:

ID	Group	Topics	Node
1	test	1	node3.itcast.cn:9092
2	test1	1	node1.itcast.cn:9092

Showing 1 to 2 of 2 entries

Red arrow points to the 'test' group, labeled: 点击消费者组

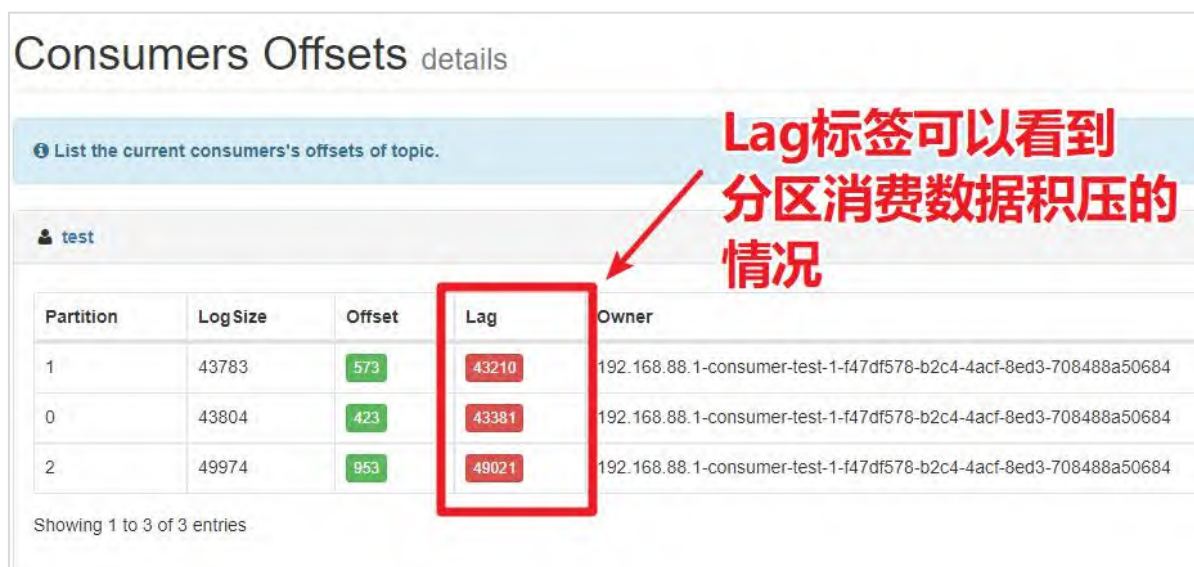
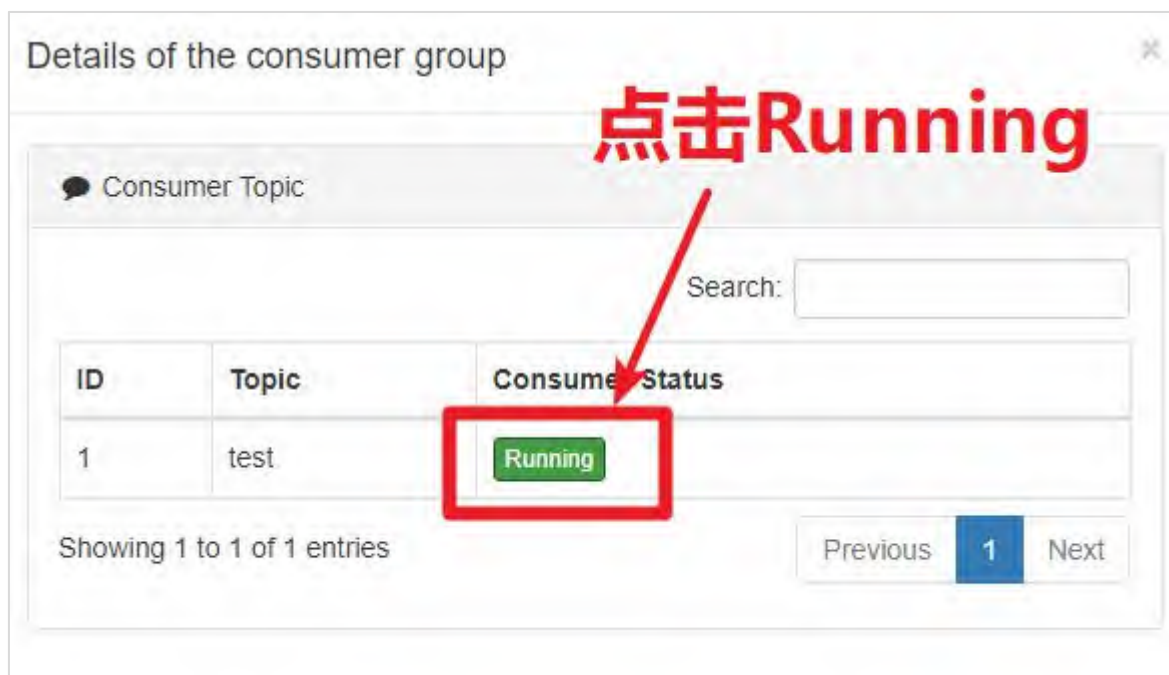


Kafka Eagle interface showing Consumers overview. The table lists consumer groups:

ID	Group	Topics	Node
1	ods_user	1	node1.itcast.cn:9092
2	test	1	node3.itcast.cn:9092

Showing 1 to 2 of 2 entries

Red arrow points to the 'test' group, labeled: 找到对应的group id



4.5.2 解决数据积压问题

当Kafka出现数据积压问题时，首先要找到数据积压的原因。以下是在企业中出现数据积压的几个类场景。

4.5.2.1 数据写入MySQL失败

问题描述

某日运维人员找到开发人员，说某个topic的一个分区发生数据积压，这个topic非常重要，而且开

始有用户投诉。运维非常紧张，赶紧重启了这台机器。重启之后，还是无济于事。

问题分析

消费这个topic的代码比较简单，主要就是消费topic数据，然后进行判断在进行数据库操作。运维通过kafka-eagle找到积压的topic，发现该topic的某个分区积压了几十万条的消息。

最后，通过查看日志发现，由于数据写入到MySQL中报错，导致消费分区的offset一直没有提交，所以数据积压严重。

4.5.2.2 因为网络延迟消费失败

问题描述

基于Kafka开发的系统平稳运行了两个月，突然某天发现某个topic中的消息出现数据积压，大概有几万条消息没有被消费。

问题分析

通过查看应用程序日志发现，有大量的消费超时失败。后查明原因，因为当天网络抖动，通过查看Kafka的消费者超时配置为50ms，随后，将消费的时间修改为500ms后问题解决。

5. Kafka中数据清理 (Log Deletion)

Kafka的消息存储在磁盘中，为了控制磁盘占用空间，Kafka需要不断地对过去的一些消息进行清理工作。Kafka的每个分区都有很多的日志文件，这样也是为了方便进行日志的清理。在Kafka中，提供两种日志清理方式：

- 日志删除 (Log Deletion)：按照指定的策略**直接删除**不符合条件的日志。
- 日志压缩 (Log Compaction)：按照消息的key进行整合，有相同key的但有不同value值，只保留最后一个版本。

在Kafka的broker或topic配置中：

配置项	配置值	说明
log.cleaner.enable	true (默认)	开启自动清理日志功能
log.cleanup.policy	delete (默认)	删除日志
log.cleanup.policy	compaction	压缩日志

log.cleanup.policy	delete,compact	同时支持删除、压缩
--------------------	----------------	-----------

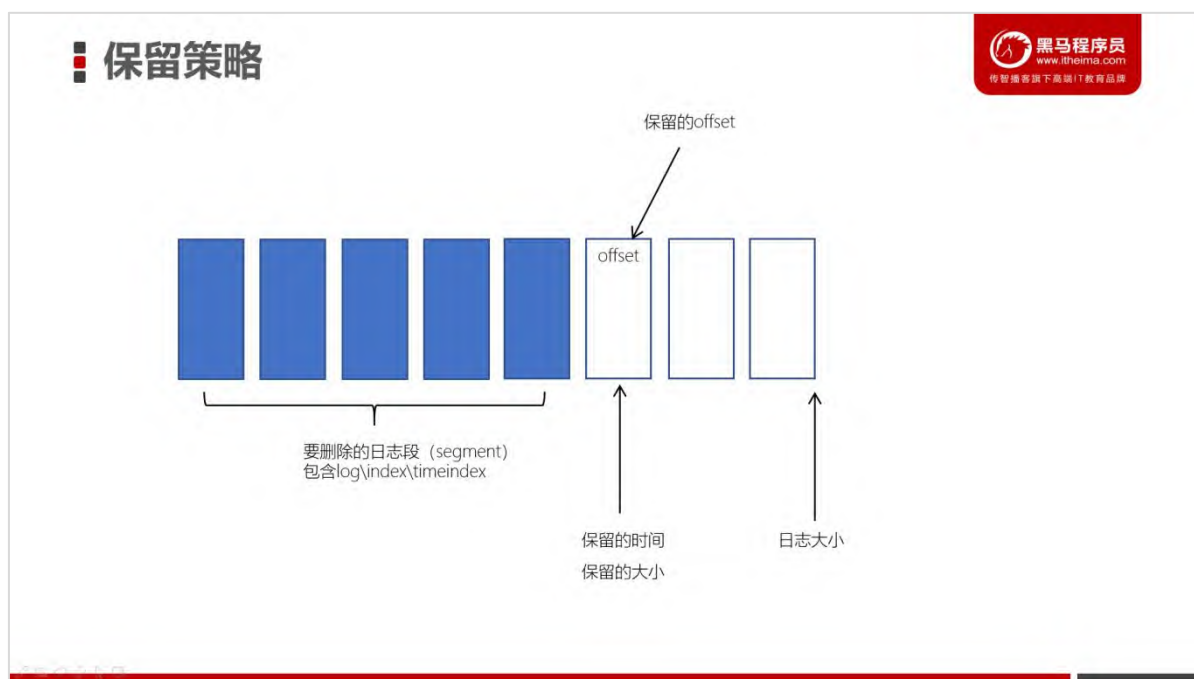
5.1 日志删除

日志删除是以段（segment日志）为单位来进行定期清理的。

5.1.1 定时日志删除任务

Kafka日志管理器中会有一个专门的日志删除任务来定期检测和删除不符合保留条件的日志分段文件，这个周期可以通过broker端参数log.retention.check.interval.ms来配置，默认值为300,000，即5分钟。当前日志分段的保留策略有3种：

1. 基于时间的保留策略
2. 基于日志大小的保留策略
3. 基于日志起始偏移量的保留策略



5.1.2 基于时间的保留策略

以下三种配置可以指定如果Kafka中的消息超过指定的阈值，就会将日志进行自动清理：

- log.retention.hours
- log.retention.minutes

- log.retention.ms

其中，优先级为 log.retention.ms > log.retention.minutes > log.retention.hours。默认情况，在 broker 中，配置如下：

log.retention.hours=168

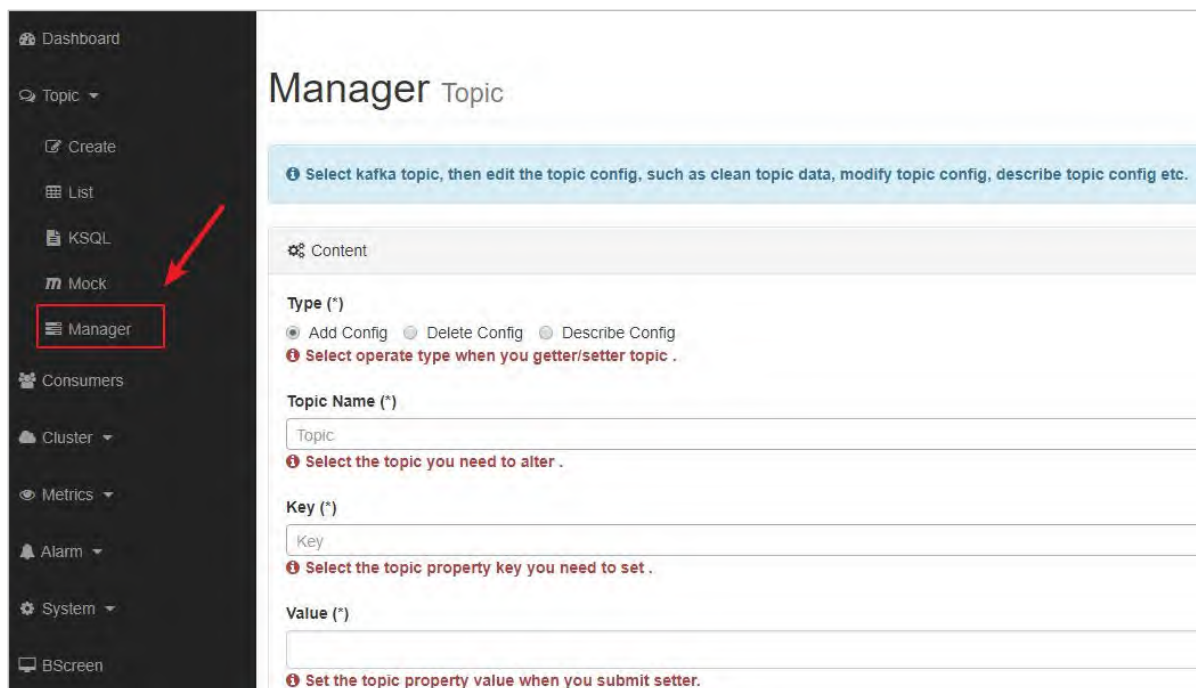
也就是，默认日志的保留时间为168小时，相当于保留7天。

删除日志分段时：

1. 从日志文件对象中所维护日志分段的跳跃表中移除待删除的日志分段，以保证没有线程对这些日志分段进行读取操作
2. 将日志分段文件添加上“.deleted”的后缀（也包括日志分段对应的索引文件）
3. Kafka的后台定时任务会定期删除这些“.deleted”为后缀的文件，这个任务的延迟执行时间可以通过file.delete.delay.ms参数来设置，默认值为60000，即1分钟。

5.1.2.1 设置topic 5秒删除一次

1. 为了方便观察，设置段文件的大小为1M。



key: segment.bytes

value: 1048576



Content

Type (*)

☒ Add Config ☐ Delete Config ☐ Describe Config

Select operate type when you getter/setter topic .

Topic Name (*)

test

Select the topic you need to alter .

Key (*)

segment.bytes

Select the topic property key you need to set .

Value (*)

1048576

Set the topic property value when you submit setter.

Message (*)

{ "type": "ADD", "value": "SUCCESS" }

Get result from server when you getter/setter topic .

Submit

设置segment段的大小为1M

2. 设置topic的删除策略

key: retention.ms

value: 5000

Manager Topic

Select kafka topic, then edit the topic config, such as clean topic data, modify topic config, describe topic config etc.

Content

Type (*)
☒ Add Config ☐ Delete Config ☐ Describe Config
Select operate type when you getter/setter topic .

Topic Name (*)
test
Select the topic you need to alter .

Key (*)
retention.ms
Select the topic property key you need to set .

Value (*)
5000
Set the topic property value when you submit setter.

Message (*)

Get result from server when you getter/setter topic .

Submit

指定topic

指定删除策略

指定保留时间

尝试往topic中添加一些数据，等待一会，观察日志的删除情况。我们发现，日志会定期被标记为删除，然后被删除。

5.1.3 基于日志大小的保留策略

日志删除任务会检查当前日志的大小是否超过设定的阈值来寻找可删除的日志分段的文件集合。可以通过broker端参数 `log.retention.bytes` 来配置，默认值为-1，表示无穷大。如果超过该大小，会自动将超出部分删除。

注意:

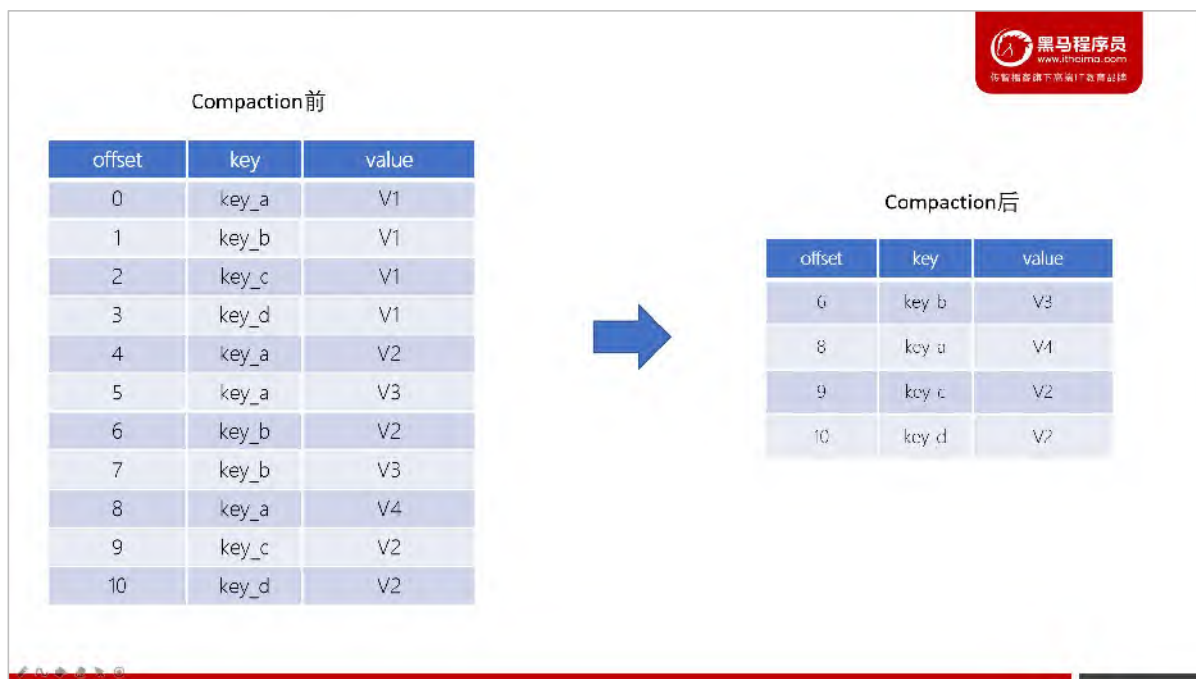
log.retention.bytes 配置的是日志文件的总大小，而不是单个的日志分段的大小，一个日志文件包含多个日志分段。

5.1.4 基于日志起始偏移量保留策略

每个segment日志都有它的起始偏移量，如果起始偏移量小于 logStartOffset，那么这些日志文件将会标记为删除。

5.2 日志压缩 (Log Compaction)

Log Compaction是默认的日志删除之外的清理过时数据的方式。它会将相同的key对应的数据只保留一个版本。



- Log Compaction执行后，offset将不再连续，但依然可以查询Segment
- Log Compaction执行前后，日志分段中的每条消息偏移量保持不变。Log Compaction会生成一个新的Segment文件
- Log Compaction是针对key的，在使用的时候注意每个消息的key不为空
- 基于Log Compaction可以保留key的最新更新，可以基于Log Compaction来恢复消费者的最新状态

6. Kafka配额限速机制 (Quotas)

生产者和消费者以极高的速度生产/消费大量数据或产生请求，从而占用broker上的全部资源，造成网络IO饱和。有了配额 (Quotas) 就可以避免这些问题。Kafka支持配额管理，从而可以对Producer和Consumer的produce&fetch操作进行流量限制，防止个别业务压爆服务器。

6.1 限制producer端速率

为所有client id设置默认值，以下为所有producer程序设置其TPS不超过1MB/s，即1048576/s，命令如下：

```
bin/kafka-configs.sh --zookeeper node1.itcast.cn:2181 --alter --add-config  
'producer_byte_rate=1048576' --entity-type clients --entity-default
```

运行基准测试，观察生产消息的速率

```
bin/kafka-producer-perf-test.sh --topic test --num-records 500000 --throughput -1  
--record-size 1000 --producer-props  
bootstrap.servers=node1.itcast.cn:9092,node2.itcast.cn:9092,node3.itcast.cn:9092 acks=1
```

结果：

50000 records sent, 1108.156028 records/sec (1.06 MB/sec)

6.2 限制consumer端速率

对consumer限速与producer类似，只不过参数名不一样。

为指定的topic进行限速，以下为所有consumer程序设置topic速率不超过1MB/s，即1048576/s。命令如下：

```
bin/kafka-configs.sh --zookeeper node1.itcast.cn:2181 --alter --add-config  
'consumer_byte_rate=1048576' --entity-type clients --entity-default
```

运行基准测试：

```
bin/kafka-consumer-perf-test.sh --broker-list
node1.itcast.cn:9092,node2.itcast.cn:9092,node3.itcast.cn:9092 --topic test --fetch-size
1048576 --messages 500000
```

结果为：

MB.sec: 1.0743

6.3 取消Kafka的Quota配置

使用以下命令，删除Kafka的Quota配置

```
bin/kafka-configs.sh --zookeeper node1.itcast.cn:2181 --alter --delete-config
'producer_byte_rate' --entity-type clients --entity-default

bin/kafka-configs.sh --zookeeper node1.itcast.cn:2181 --alter --delete-config
'consumer_byte_rate' --entity-type clients --entity-default
```