

第1章 Kafka入门

1. 简介

1.1 消息队列简介

1.1.1 什么是消息队列

消息队列，英文名：Message Queue，经常缩写为MQ。从字面上来理解，消息队列是一种用来存储消息的队列。

我们可以简单理解消息队列就是**将需要传输的数据存放在队列中**。

1.1.2 消息队列中间件

消息队列中间件就是用来存储消息的软件（组件）。举个例子来理解，为了分析网站的用户行为，我们需要记录用户的访问日志。这些一条条的日志，可以看成是一条条的消息，我们可以将它们保存到消息队列中。将来有一些应用程序需要处理这些日志，就可以随时将这些消息取出来处理。

目前市面上的消息队列有很多，例如：Kafka、RabbitMQ、ActiveMQ、RocketMQ、ZeroMQ等。

1.1.2.1 什么叫Kafka呢

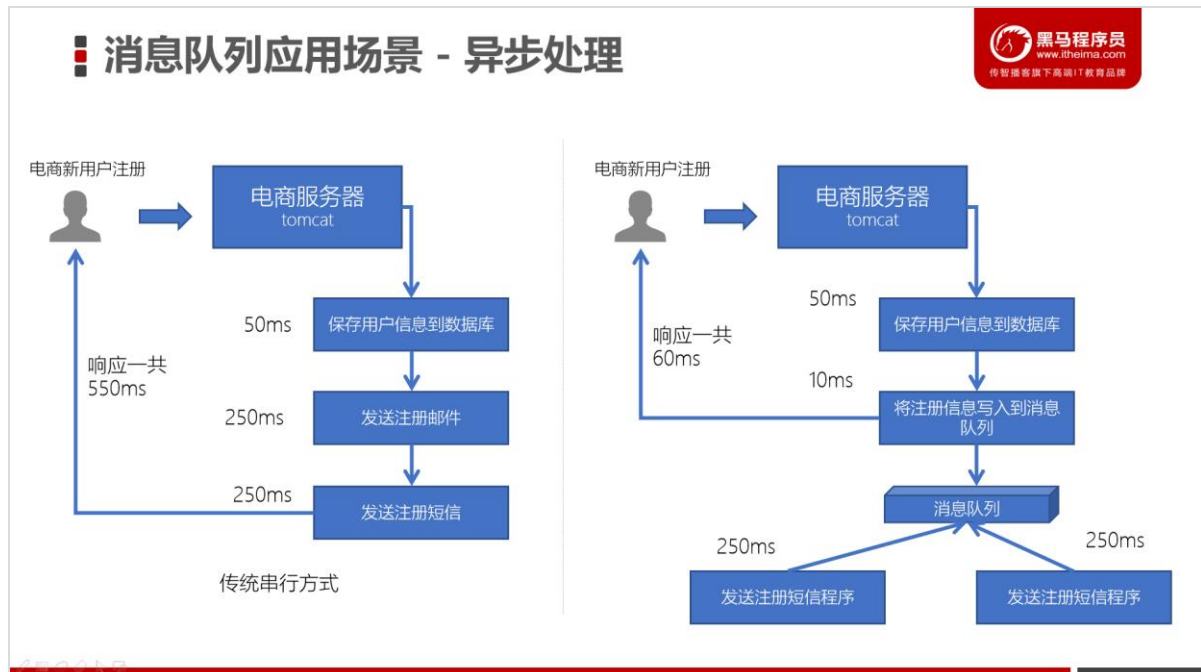
Kafka的架构师jay kreps非常喜欢franz kafka（弗兰兹·卡夫卡），并且觉得kafka这个名字很酷，因此取了个和消息传递系统完全不相干的名称kafka，该名字并没有特别的含义。

「也就是说，你特别喜欢尼古拉斯赵四，将来你做一个项目，也可以把项目的名字取名为：尼古拉斯赵四，然后这个项目就火了」

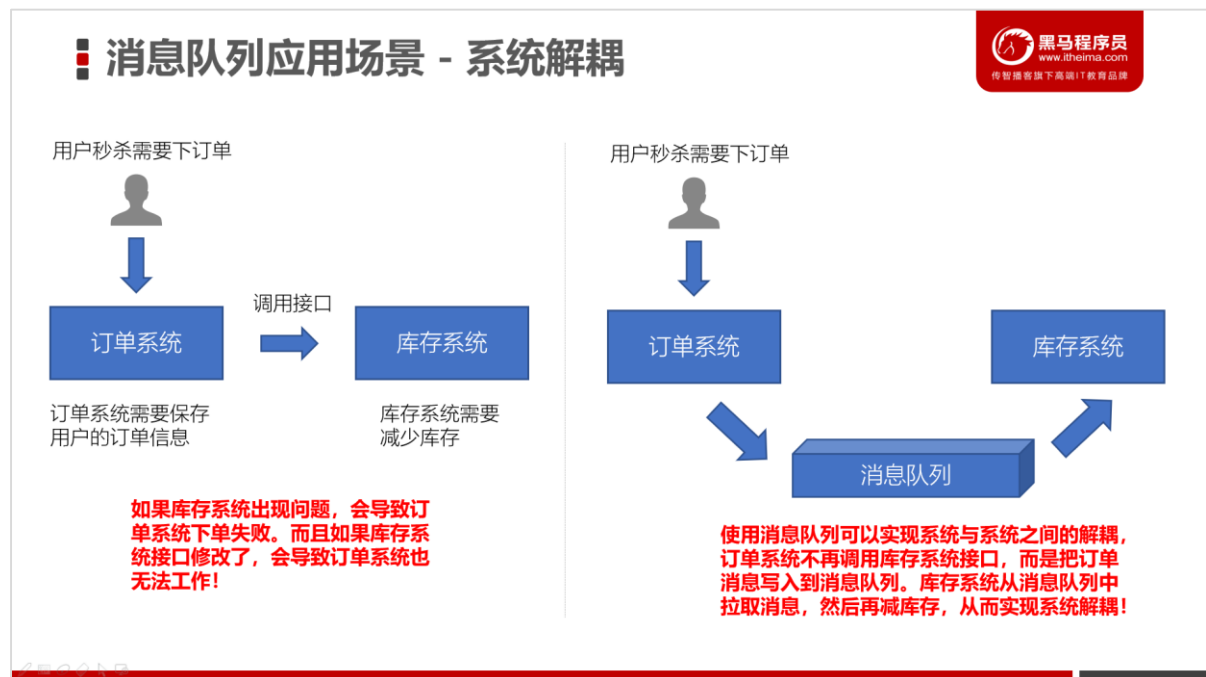
1.1.3 消息队列的应用场景

1.1.3.1 异步处理

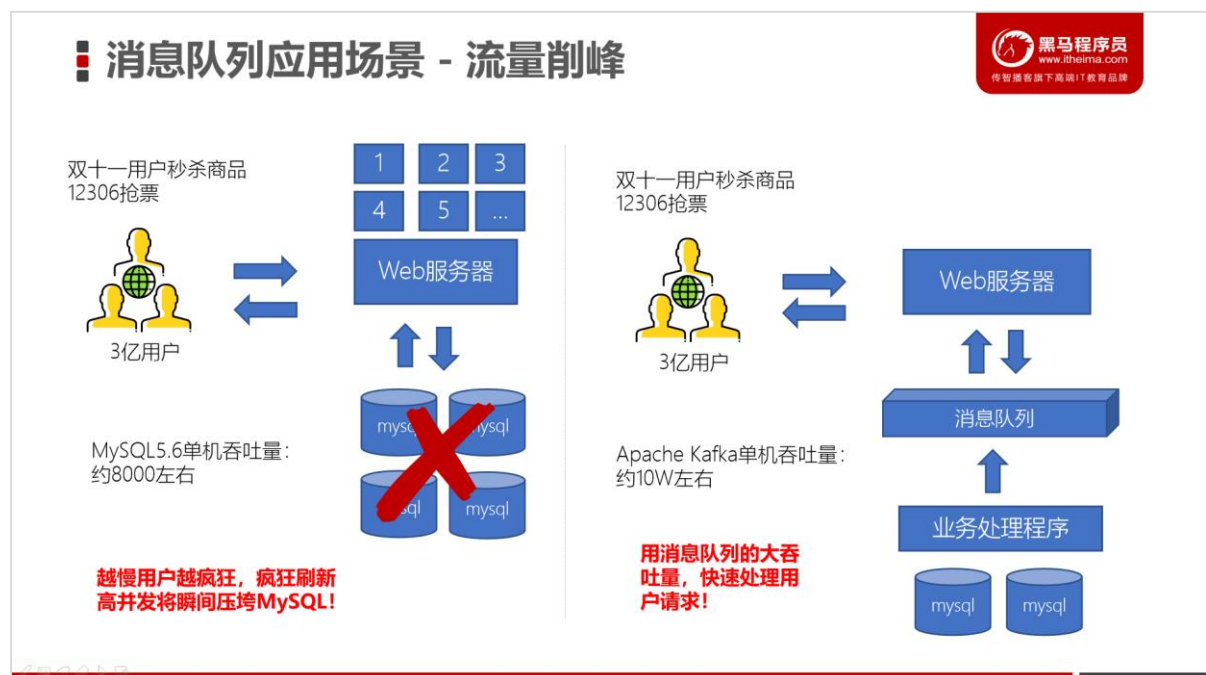
电商网站中，新的用户注册时，需要将用户的信息保存到数据库中，同时还需要额外发送注册的邮件通知、以及短信注册码给用户。但因为发送邮件、发送注册短信需要连接外部的服务器，需要额外等待一段时间，此时，就可以使用消息队列来进行异步处理，从而实现快速响应。



1.1.3.2 系统解耦

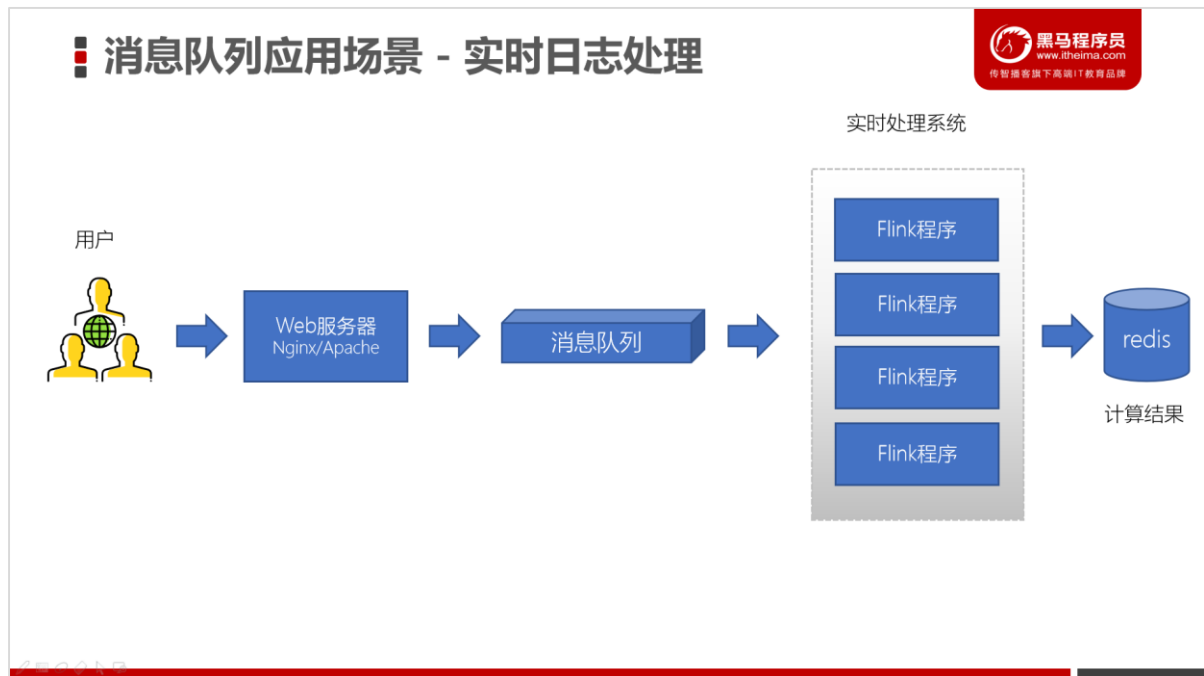


1.1.3.3 流量削峰



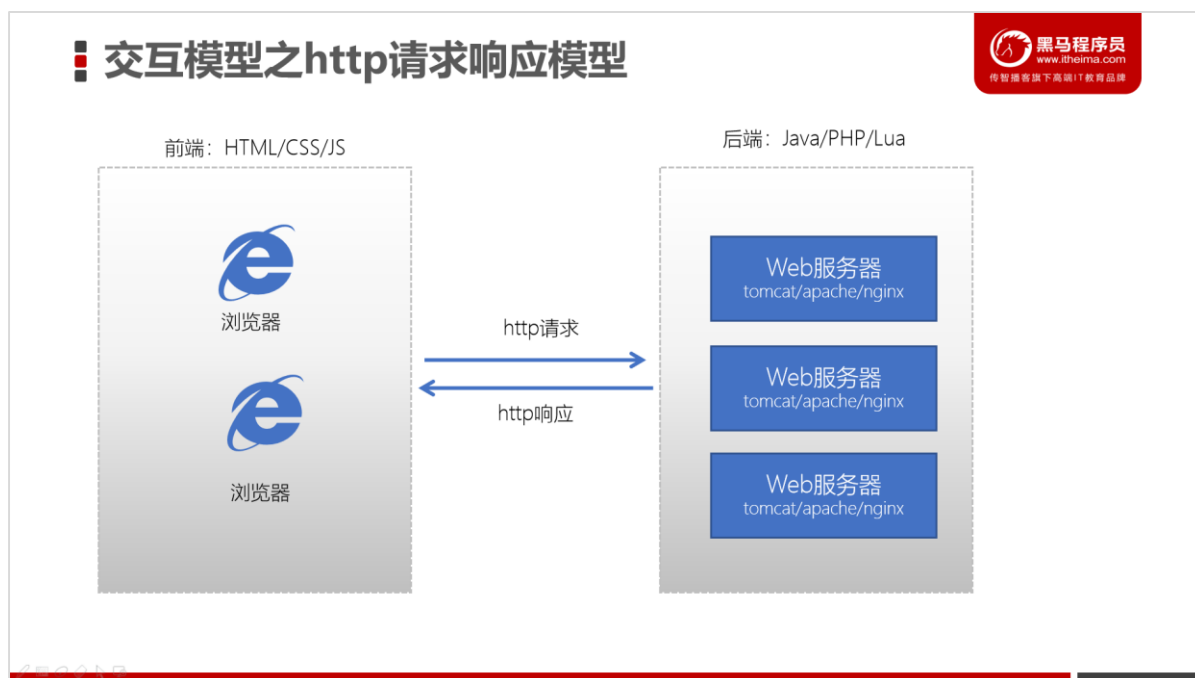
1.1.3.4 日志处理（大数据领域常见）

大型电商网站（淘宝、京东、国美、苏宁...）、App（抖音、美团、滴滴等）等需要分析用户行为，要根据用户的访问行为来发现用户的喜好以及活跃情况，需要在页面上收集大量的用户访问信息。

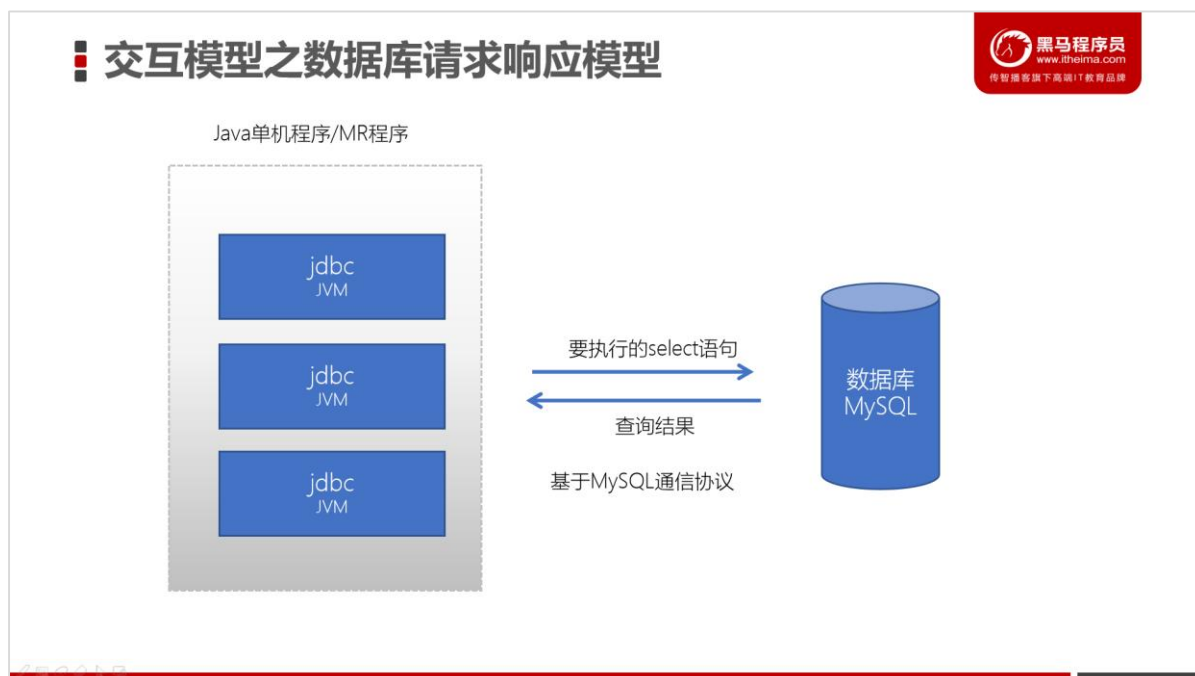


1.1.4 生产者、消费者模型

我们之前学习过Java的服务器开发，Java服务器端开发的交互模型是这样的：



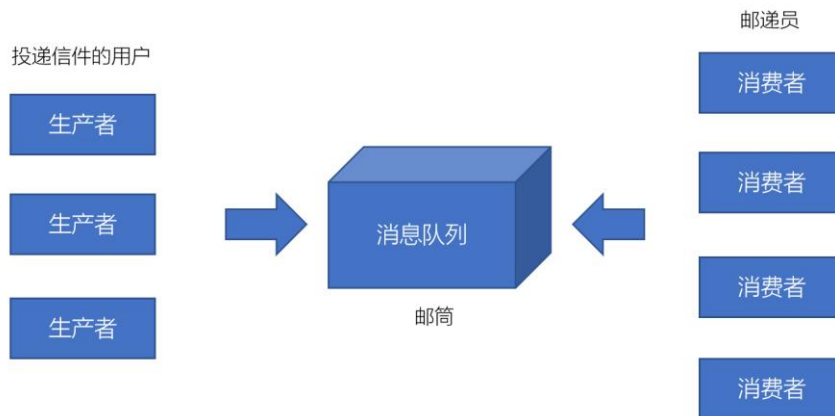
我们之前也学习过使用Java JDBC来访问操作MySQL数据库，它的交互模型是这样的：



它也是一种请求响应模型，只不过它不再是基于http协议，而是基于MySQL数据库的通信协议。

而如果我们基于消息队列来编程，此时的交互模式成为：生产者、消费者模型。

交互模型之生产者消费者模型



1.1.5 消息队列的两种模式

1.1.5.1 点对点模式

点对点模式

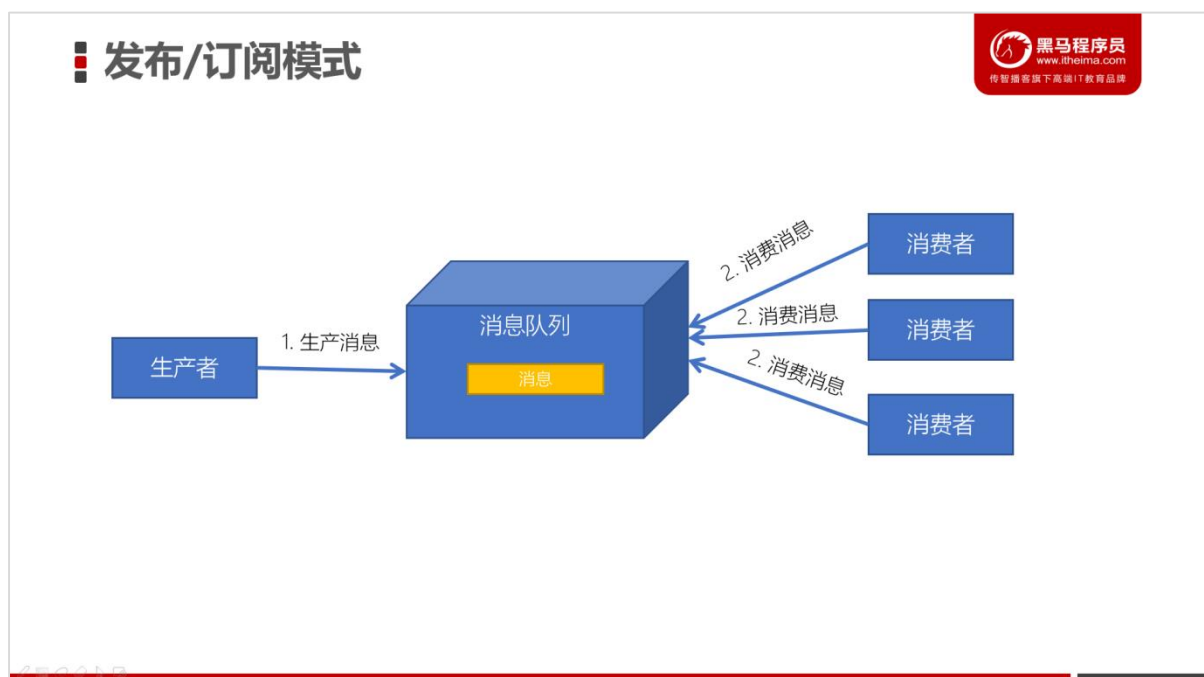


消息发送者生产消息发送到消息队列中，然后消息接收者从消息队列中取出并且消费消息。消息被消费以后，消息队列中不再有存储，所以消息接收者不可能消费到已经被消费的消息。

点对点模式特点：

- 每个消息只有一个接收者（Consumer）（即一旦被消费，消息就不再在消息队列中）
- 发送者和接收者间没有依赖性，发送者发送消息之后，不管有没有接收者在运行，都不会影响到发送者下次发送消息；
- 接收者在成功接收消息之后需向队列应答成功，以便消息队列删除当前接收的消息；

1.1.5.2 发布订阅模式



发布/订阅模式特点：

- 每个消息可以有多个订阅者；
- 发布者和订阅者之间有时间上的依赖性。针对某个主题（Topic）的订阅者，它必须创建一个订阅者之后，才能消费发布者的消息。
- 为了消费消息，订阅者需要提前订阅该角色主题，并保持在线运行；

1.2 Kafka简介

1.2.1 什么是Kafka



Kafka是由Apache软件基金会开发的一个开源流平台，由Scala和Java编写。Kafka的Apache官网是这样介绍Kafka的。

Apache Kafka是一个分布式流平台。一个分布式的流平台应该包含3点关键的能力：

1. 发布和订阅流数据流，类似于消息队列或者是企业消息传递系统
2. 以容错的持久化方式存储数据流
3. 处理数据流

英文原版

- **Publish and subscribe** to streams of records, similar to a message queue or enterprise messaging system.
- **Store** streams of records in a fault-tolerant durable way.
- **Process** streams of records as they occur.

更多参考：<http://kafka.apache.org/documentation/#introduction>

我们重点关键三个部分的关键词：

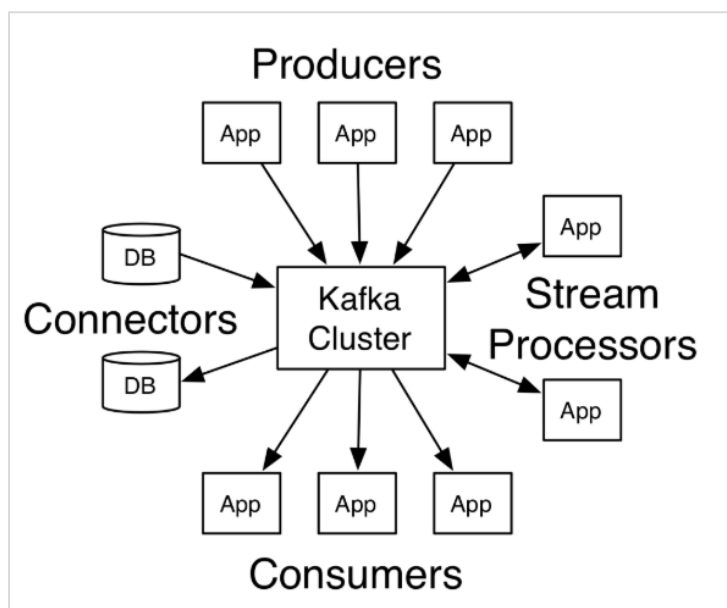
1. Publish and subscribe：**发布与订阅**
2. Store：**存储**
3. Process：处理

课程主要围绕消息队列、存储讲解。

1.2.2 Kafka的应用场景

我们通常将Apache Kafka用在两类程序：

1. 建立实时数据管道，以可靠地在系统或应用程序之间获取数据
2. 构建实时流应用程序，以转换或响应数据流



上图，我们可以看到：

1. Producers：可以有很多的应用程序，将消息数据放入到Kafka集群中。
2. Consumers：可以有很多的应用程序，将消息数据从Kafka集群中拉取出来。
3. Connectors：Kafka的连接器可以将数据库中的数据导入到Kafka，也可以将Kafka的数据导出到数据库中。
4. Stream Processors：流处理器可以从Kafka中拉取数据，也可以将数据写入到Kafka中。

1.2.3 Kafka诞生背景

1.2.3.1 Kafka的诞生背景

kafka的诞生，是为了解决linkedin的数据管道问题，起初linkedin采用了ActiveMQ来进行数据交换，大约是在2010年前后，那时的ActiveMQ还远远无法满足linkedin对数据传递系统的要求，经常由于各种缺陷而导致消息阻塞或者服务无法正常访问，为了能够解决这个问题，linkedin决定研发自己的消息传递系统，当时linkedin的首席架构师jay kreps便开始组织团队进行消息传递系统的研发。

提示：

1. Linkedin还是挺牛逼的
2. Kafka比ActiveMQ牛逼得多

1.3 Kafka的优势













前面我们了解到，消息队列中间件有很多，为什么我们要选择Kafka？

特性	ActiveMQ	RabbitMQ	Kafka	RocketMQ
所属社区/公司	Apache	Mozilla Public License	Apache	Apache/Ali
成熟度	成熟	成熟	成熟	比较成熟
生产者-消费者模式	支持	支持	支持	支持
发布-订阅	支持	支持	支持	支持
REQUEST-REPLY	支持	支持	-	支持
API完备性	高	高	高	低(静态配置)
多语言支持	支持JAVA优先	语言无关	支持，JAVA优先	支持
单机吞吐量	万级(最差)	万级	十万级	十万级(最高)
消息延迟	-	微秒级	毫秒级	-
可用性	高(主从)	高(主从)	非常高(分布式)	高
消息丢失	-	低	理论上不会丢失	-
消息重复	-	可控制	理论上会有重复	-
事务	支持	不支持	支持	支持

文档的完备性	高	高	高	中
提供快速入门	有	有	有	无
首次部署难度	-	低	中	高

在大数据技术领域，一些重要的组件、框架都支持Apache Kafka，不论成熟度、社区、性能、可靠性，Kafka都是非常有竞争力的一款产品。

1.4 哪些公司在使用Kafka

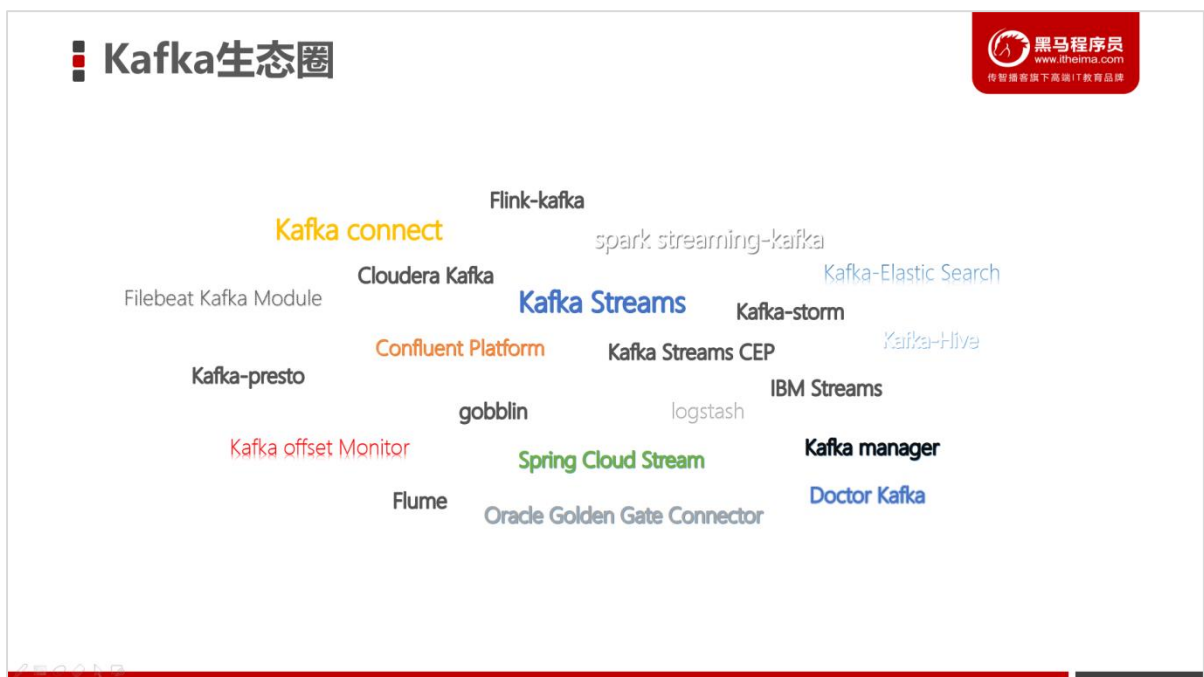
 <small>Google</small>	 Tencent Cloud <small>Tencent</small>	 <small>Facebook</small>
 <small>Pineapple Fund</small>	 Microsoft	 <small>Amazon Web Services</small>
 <small>Comcast</small>		 <small>Verizon Media</small>
 <small>reliable hosting</small>	 <small>ARM</small>	 <small>Bloomberg</small>



1.5 Kafka生态圈介绍

Apache Kafka这么多年的发展，目前也有一个较庞大的生态圈。

Kafka生态圈官网地址：<https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem>



1.6 Kafka版本

本次课程使用的Kafka版本为1.0.0。

可以注意到Kafka的版本号为：`kafka_2.11-1.0.0`，因为kafka主要是使用scala语言开发的，2.11为scala的版本号。<http://kafka.apache.org/downloads>可以查看到每个版本的发布时间。

2. 环境搭建

2.1 搭建Kafka集群

1、3台机器创建文件夹，用于保存kafka数据

```
mkdir -p /export/data/kafka
```

2、上传安装包、解压

```
tar -zxvf kafka_2.11-1.0.0.tgz -C /export/servers/
```

```
cd /export/servers/
```

```
mv kafka_2.11-1.0.0 kafka
```

3、修改配置文件

```
cd /export/servers/kafka/config/
```

```
vi server.properties
```

主要修改以下6个地方：

1) broker.id 需要保证每一台kafka都有一个独立的broker

2) log.dirs 数据存放的目录

3) zookeeper.connect zookeeper的连接地址信息

4) delete.topic.enable 是否直接删除topic

5) host.name 主机的名称

6) 修改: listeners=PLAINTEXT://node-1:9092



4、将安装包scp给其他机器

```
cd /export/servers
```

```
scp -r kafka/ node2:$PWD
```

```
scp -r kafka/ node3:$PWD
```

5、拷贝后，需要修改每一台的broker.id 和 host.name和listeners

6、kafka集群启动

在kafka启动前，一定要让zookeeper启动起来

```
cd /export/servers/kafka
```

#前端启动

```
bin/kafka-server-start.sh config/server.properties
```

#后台启动:

```
bin/kafka-server-start.sh -daemon config/server.properties
```

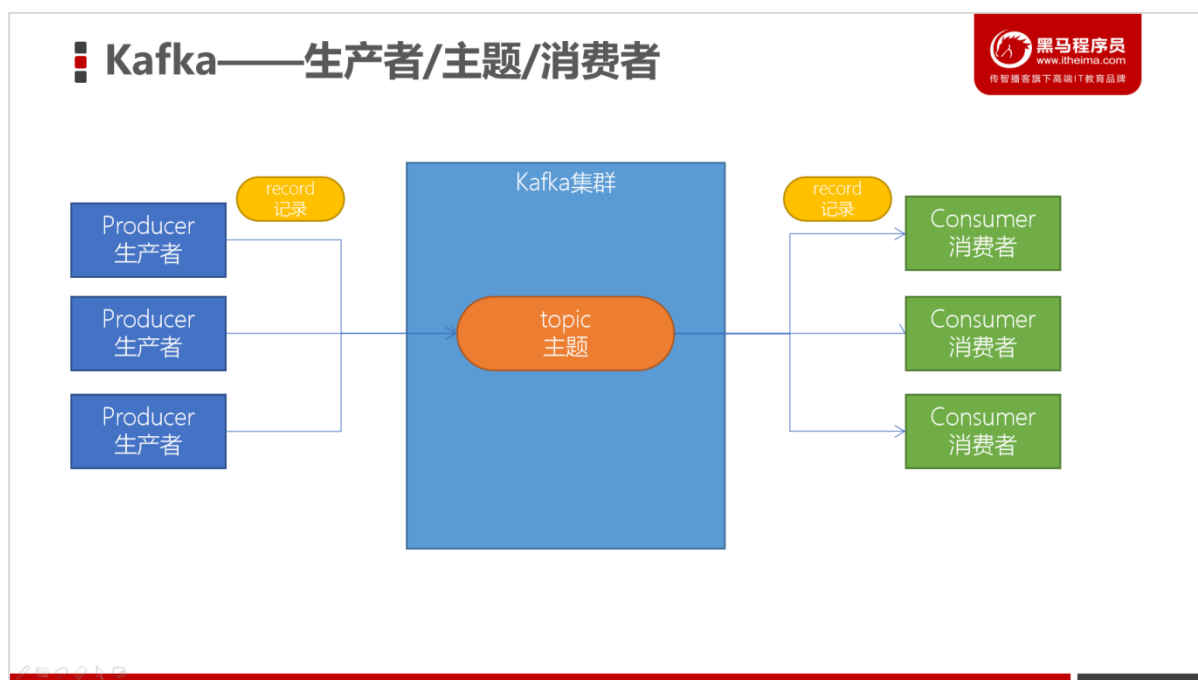
#kafka停止

```
bin/kafka-server-stop.sh
```

2.2 目录结构分析

目录名称	说明
bin	Kafka的所有执行脚本都在这里。例如：启动Kafka服务器、创建Topic、生产者、消费者程序等等
config	Kafka的所有配置文件
libs	运行Kafka所需要的所有JAR包
logs	Kafka的所有日志文件，如果Kafka出现一些问题，需要到该目录中去查看异常信息
site-docs	Kafka的网站帮助文件

3. 基础操作



3.1 创建topic

创建一个topic (主题)。Kafka中所有的消息都是保存在主题中，要生产消息到Kafka，首先必须要有一个确定的主题。

```
# 创建名为test的主题
bin/kafka-topics.sh --create --zookeeper node01:2181 --replication-factor 2 --partitions 3 --topic test
# 查看目前Kafka中的主题
bin/kafka-topics.sh --list --zookeeper node01:2181
```

3.2 生产消息到Kafka

1. 使用Kafka内置的测试程序，生产一些消息到Kafka的test主题中。

```
bin/kafka-console-producer.sh --broker-list node01:9092 --topic test
```

3.3 从Kafka消费消息

使用下面的命令来消费 test 主题中的消息。

```
bin/kafka-console-consumer.sh --from-beginning --topic test --zookeeper node01:2181
```

3.4 查看主题信息

运行describe查看topic的相关详细信息

```
bin/kafka-topics.sh --describe --zookeeper node01:2181 --topic test
```

3.5 增加topic分区

```
bin/kafka-topics.sh --zookeeper zkhost:port --alter --topic topicName --partitions 8
```

3.6 增加、删除配置

```
bin/kafka-topics.sh --zookeeper node01:2181 --alter --topic test --config flush.messages=1
```

```
bin/kafka-topics.sh --zookeeper node01:2181 --alter --topic test --delete-config flush.messages
```

将flush.messages设置为1,那么每一条消息都会刷盘。

3.7 删除topic

目前删除topic在默认情况下只是打上一个删除的标记，在重新启动kafka后才删除。如果需要立即删除，则需要

server.properties中配置：

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

然后执行以下命令进行删除topic

```
kafka-topics.sh --zookeeper zkhost:port --delete --topic topicName
```


4. Java编程操作Kafka

4.1 生产消息到Kafka中

4.1.1 需求

接下来，我们将编写Java程序，将1-100的数字消息写入到Kafka中。

4.1.2 开发步骤

1. 导入Maven Kafka POM依赖
2. 使用以下链接来编写第一个Kafka示例程序
3. 创建用于连接Kafka的Properties配置
4. 创建一个生产者对象
5. 发送消息到指定 Topic
6. 关闭生产者

参考代码：

```
/**
 * 生产者程序：将1-100的数字消息写入到Kafka中
 */

public class _1ProducerTest {

    public static void main(String[] args) {

        // 1. 创建生产者配置

        Properties props = new Properties();

        props.put("bootstrap.servers", "192.168.88.100:9092");

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

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

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



```
// 2. 创建生产者

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

// 3. 发送1-100数字到Kafka的test主题中

for (int i = 1; i <= 100; ++i) {

    // 注意：send方法是一个异步方法，它会将要发送的数据放入到一个buffer中，然后立即
    返回

    // 这样可以使消息发送变得更高效

    producer.send(new ProducerRecord<>("test", i + ""));

}

// 4. 关闭生产者连接

producer.close();

}

}
```

参考以下文档：

<http://kafka.apache.org/24/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html>

4.1.3 生产者客户端参数

为了避免用户手动编写配置名称出错的问题 kafka封装了一个配置类 ProducerConfig

bootstrap.servers //连接kafka集群的broker地址

key.serializer //消息中key的序列化类

value.serializer //消息中value的序列化类

buffer.memory //默认32M(33554432B) 客户端缓存消息的大小 缓存的目的是可以批量发送 减少网络资源提升性能 如果生产者发送消息的速度超过发送到服务器的速度 会造成空间不足

此时 sender方法要么被阻塞 要么抛出异常 阻塞时间由下述参数指定

max.block.ms //生产者发送消息的阻塞时间 默认值60s

acks //指定消息分区中必须要有多少个副本接收到这个消息

1 (默认值) leader副本成功

0 生产者不需要等待任何服务器响应

-1, all 等待所有ISR中副本都成功

max.request.size //生产者能够发送消息的最大值 默认1MB

retries //生产者的重试次数 默认0

retry.backoff.ms //两次重试之间的时间间隔 默认100ms

4.1.4 生产者发送消息的3种方式

4.1.4.1 异步Async (发后即忘)

只管发送消息到kafka 并不关心消息是否到达。性能最高，可靠性最差。

```
kafkaProducer.send(record);
```

4.1.4.2 同步发送Sync

客户端阻塞等待kafka的响应，直到消息发送成功，或者发送异常。

```
RecordMetadata metadata = kafkaProducer.send(record).get();
```

```
System.out.println(metadata.topic()+"-"+metadata.partition()+"-"+metadata.offset());
```

//get方法就是一个阻塞的方法 可以获取一个recordmetadata对象 包含消息的元信息

4.1.4.3 异步发送Async (带回调函数)

在send方法中指定一个callback回调函数，来实现异步的消息发送确认。



```
//生产者配置参数

Properties props = new Properties();

props.put("bootstrap.servers", "node-1:9092");

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

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


//创建生产者客户端实例 用于向kafka 主题生产消息

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


//todo 在kafka中 消息是用ProducerRecord 来封装消息的

for (int i = 0; i < 10; i++) {

    ProducerRecord<String, String> record = new ProducerRecord<>("test1",
"allen:--" + i);

    //发送消息

    kafkaProducer.send(record, new Callback() {

        @Override

        public void onCompletion(RecordMetadata metadata, Exception exception) {

            if (exception != null) {

                exception.printStackTrace();

            } else {

                System.out.println(metadata.topic() + "-" + metadata.partition() + "-"

+ metadata.offset());

            }

        }

    });

    Thread.sleep(3000);

}

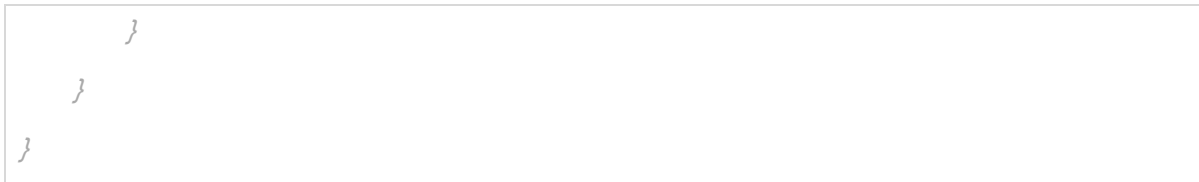
kafkaProducer.close();
```



4.2 从Kafka的topic中消费消息

4.2.1 自动提交消费偏移offset

```
/ public class ConsumerExample {  
    public static void main(String[] args) {  
        //创建kafka消费者配置  
        Properties props = new Properties();  
        props.put("bootstrap.servers", "node-1:9092");//kafka集群broker地址  
        props.put("group.id", "test");//消费者隶属的消费者组名称  
        props.put("enable.auto.commit", "true");//消费偏移offset自动提交  
        props.put("auto.commit.interval.ms", "1000");//自动提交消费偏移offset时间间隔  
        props.put("key.deserializer",  
            "org.apache.kafka.common.serialization.StringDeserializer");//消息中key反序列化类  
        props.put("value.deserializer",  
            "org.apache.kafka.common.serialization.StringDeserializer");//消息中value反序列化类  
  
        //创建kafka消费者实例  
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);  
        //消费者订阅的主题 支持订阅多个  
        consumer.subscribe(Arrays.asList("test3"));  
  
        //使用一个while循环，不断从Kafka的topic中拉取消息  
        while (true) {  
            ConsumerRecords<String, String> records = consumer.poll(100);//定义100ms超时时间  
            for (ConsumerRecord<String, String> record : records)  
                //占位符 %d数字类型 %s字符串类型 $n换行符  
                System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(),  
                    record.key(), record.value());  
        }  
    }  
}
```

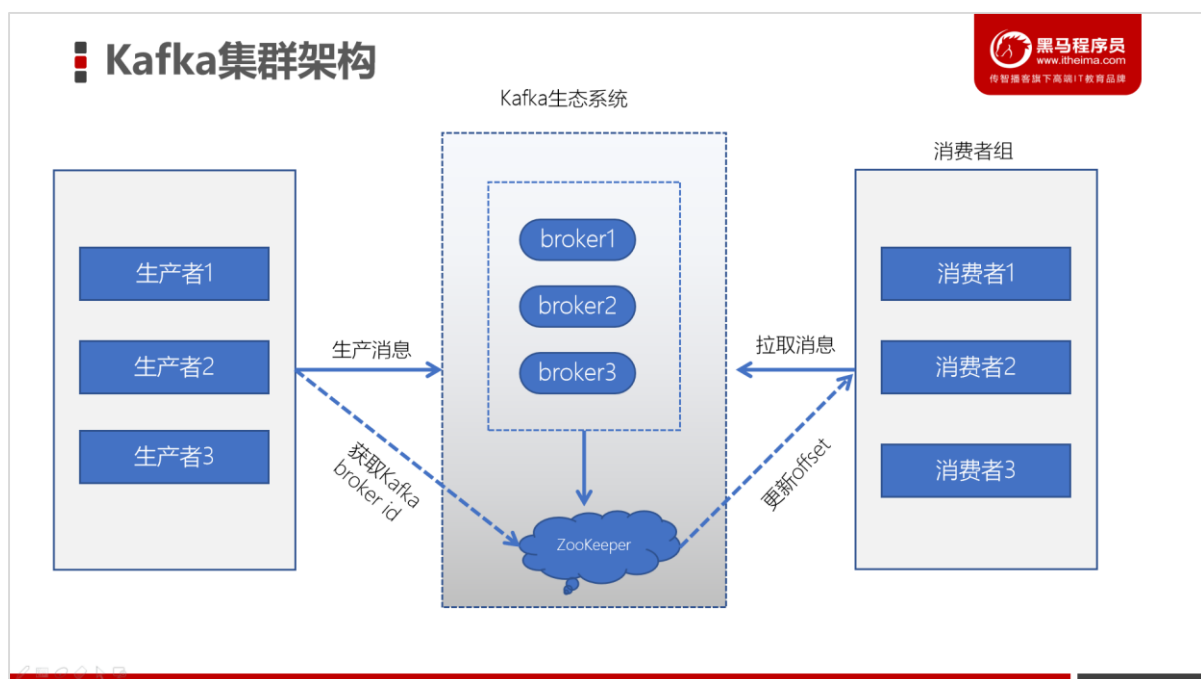


参考官网API文档：

<http://kafka.apache.org/24/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html>

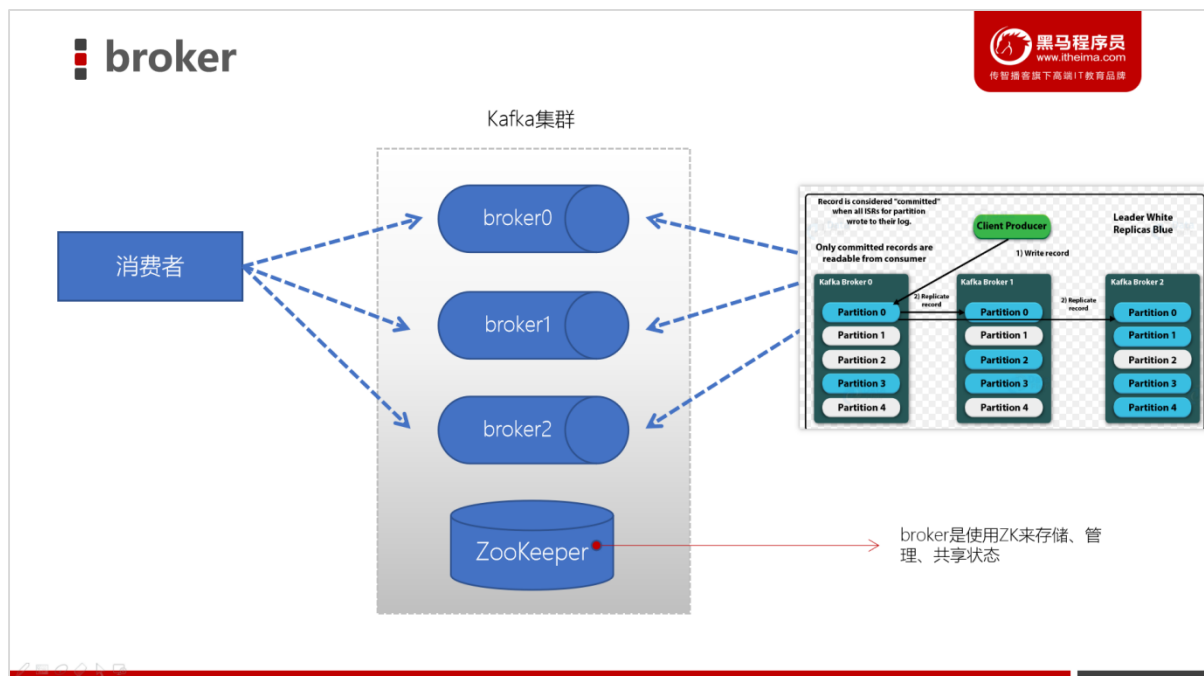
5. 架构

5.1 Kafka集群结构



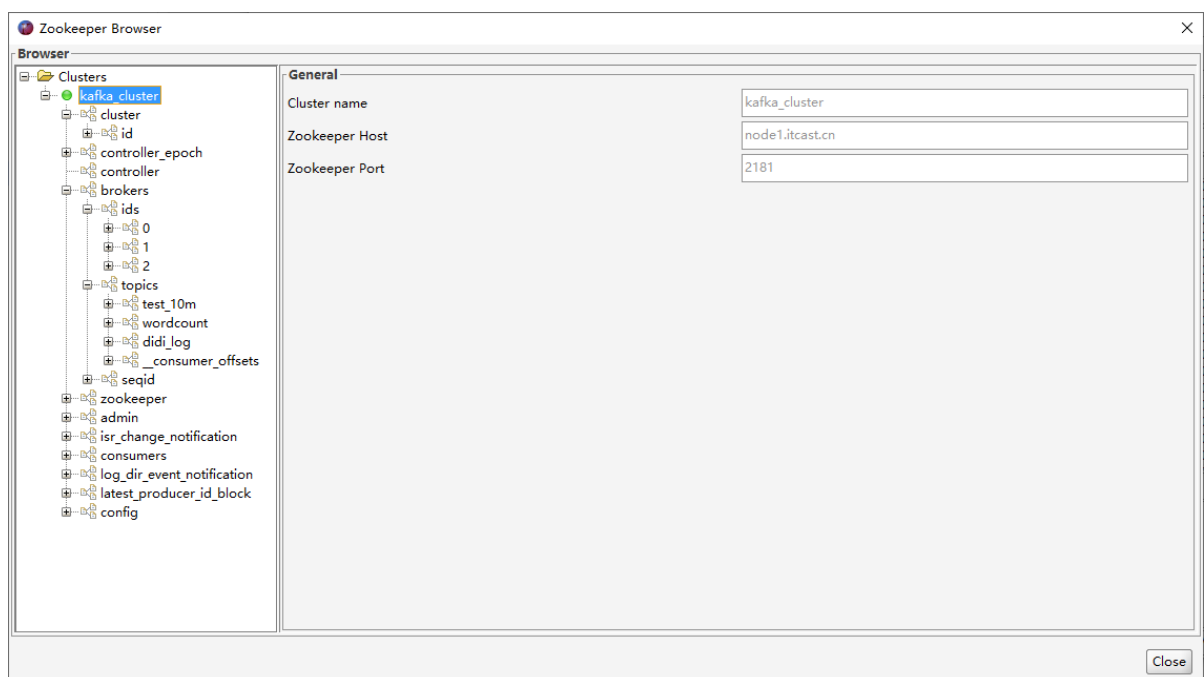
组件说明

5.1.1 broker



- 一个Kafka的集群通常由多个broker组成，这样才能实现负载均衡、以及容错
- broker是**无状态 (Stateless)** 的，它们是通过ZooKeeper来维护集群状态
- 一个Kafka的broker每秒可以处理数十万次读写，每个broker都可以处理TB消息而不影响性能
- broker的leader选举是由ZooKeeper完成的

5.1.2 zookeeper



- ZK用来管理和协调broker。
- ZK服务主要用于通知生产者和消费者Kafka集群中有新的broker加入、或者Kafka集群中出现故障的broker。
- 生产者和消费者接收到ZK的通知后，由生产者和消费者来决定如何处理。

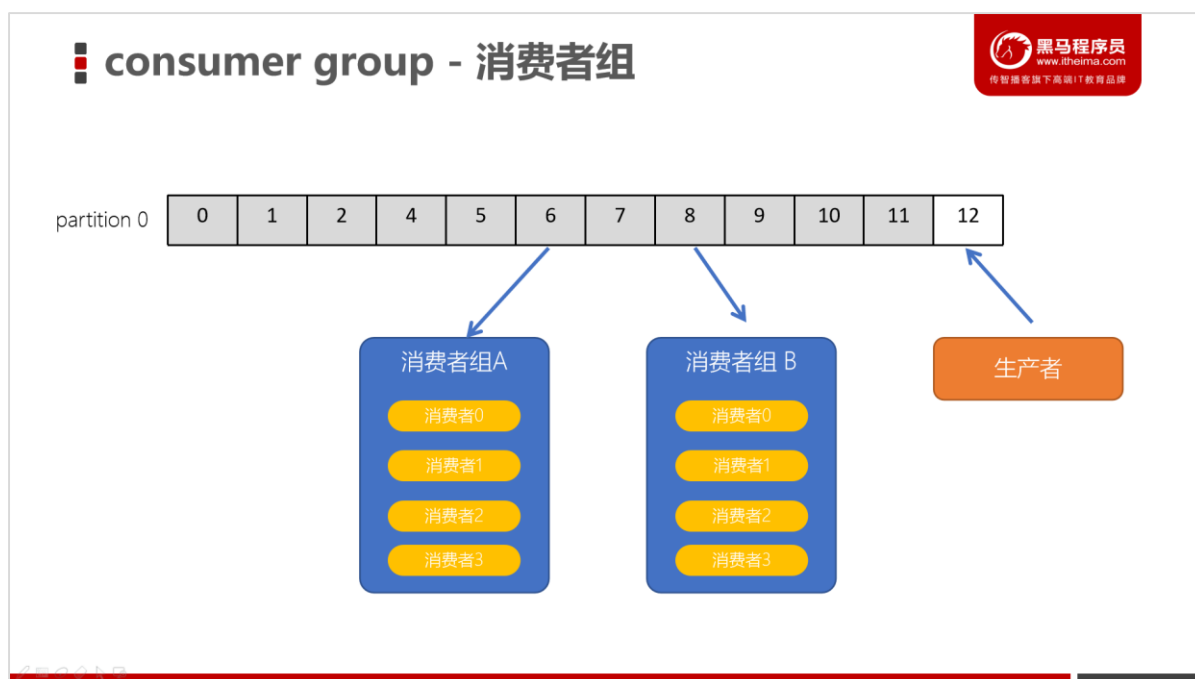
5.1.3 producer (生产者)

- 生产者负责将数据推送给broker
- 当有新的broker加入到集群中，所有的生产者都会自动向新的broker发送消息
- kafka producer发送消息可以不等待broker的确认，尽可能快速地发送消息

5.1.4 consumer (消费者)

- 由于Kafka broker是无状态的，消费者必须使用分区的偏移量 (offset) 来记录消费了多少消息
- 消费者通过向broker发送异步拉取消息的请求，将数据放入到缓冲区
- 消费者只需要提供一个偏移量 (offset)，是可以回滚或调到任意的分区中
- ZK中会记录、通知消费者的偏移量

5.1.5 consumer group (消费者组)



- consumer group是kafka提供的可扩展且具有容错性的消费者机制
- 一个消费者组可以包含多个消费者
- 一个消费者组有一个唯一的ID (group Id)
- 组内的消费者一起消费主题的所有分区数据

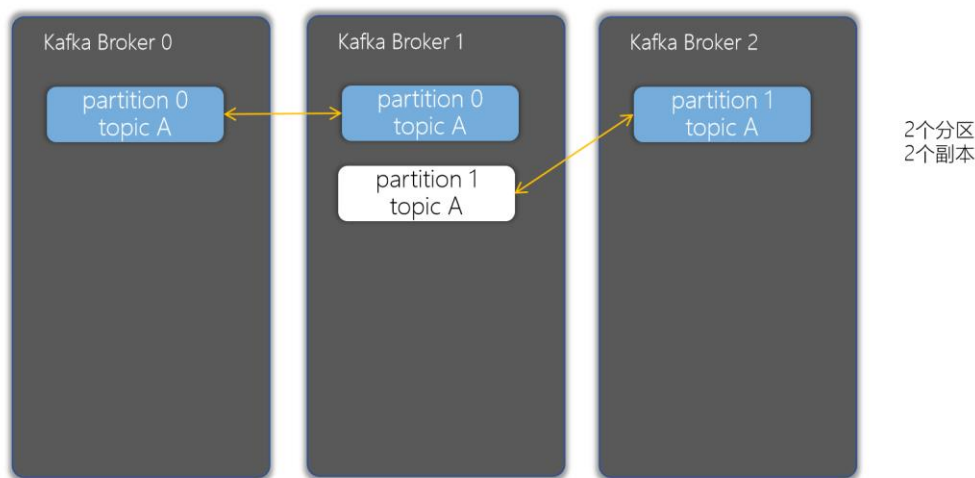
5.1.6 分区 (Partitions)



- 在Kafka集群中，主题被分为多个分区，并且分区会在broker之间复制、同步
- 如果在生产消息的时候，指定一个key，可以确保同一个key的消息，总是在一个分区中
- 基于上述的分区特点，Kafka是可以保证消息的有序性的。但如果不带key，是无法保证消息的有序性的，Kafka会以随机的方式写入到分区中

5.1.7 副本 (Replicas)

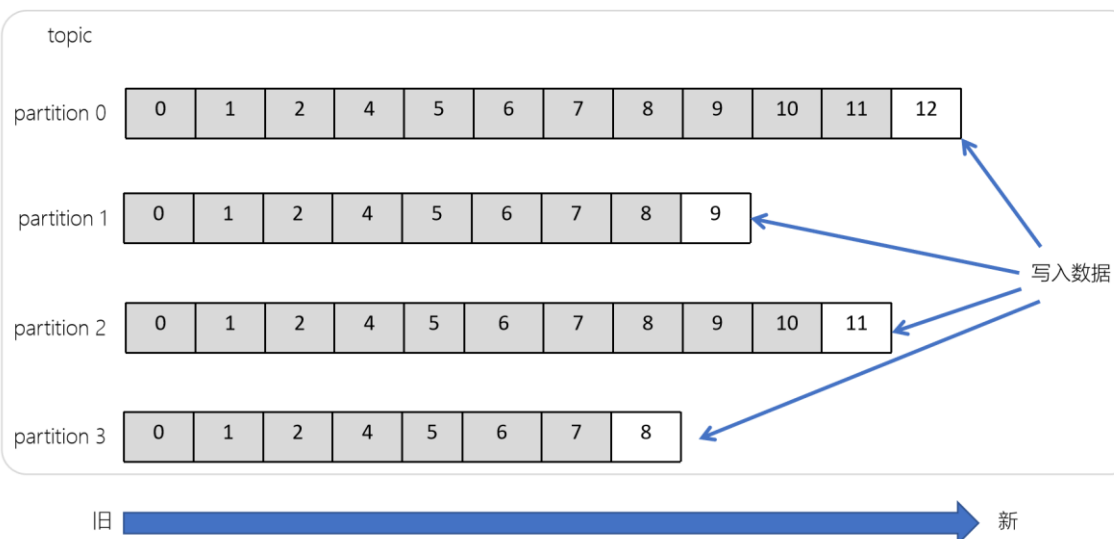
■ replication - 副本



- 如果分区的副本因子为1，那么一旦有任意分区不可用，将会导致该test将失效
- 在Kafka中，一般都会设计副本的个数 > 1。在某个broker崩溃时，分区的副本就可以发挥作用了。
- 举个例子：假设创建Topic的时候，指定分区副本数为2，我们有一个3个broker的Kafka集群，那么broker0中会保存一个分区，broker1由会保存一个分区

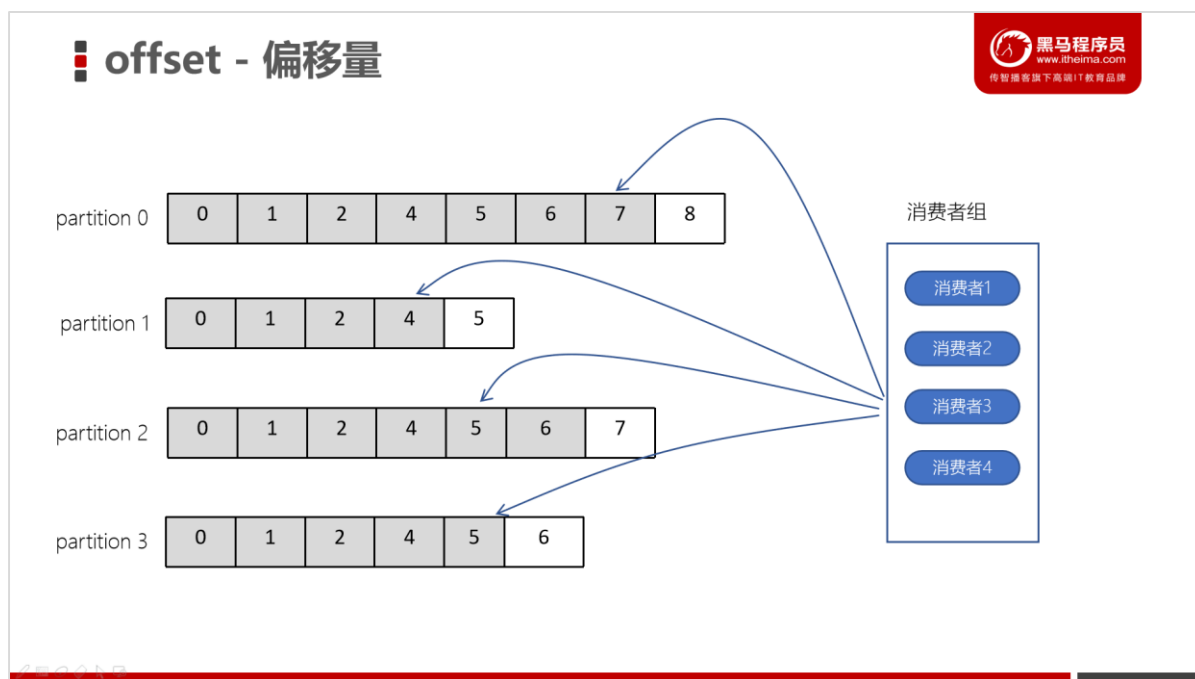
5.1.8 主题 (Topic)

■ topic - 主题



- 主题是一个逻辑概念，用于生产者发布数据，消费者拉取数据
- Kafka中的主题必须要有标识符，而且是唯一的，Kafka中可以有任意数量的主题，没有数量上的限制
- 在主题中的消息是有结构的，一般一个主题包含某一类消息
- 一旦生产者发送消息到主题中，这些消息就不能被更新（更改）

5.1.9 偏移量 (offset)



- offset记录着下一条将要发送给Consumer的消息的序号
- 在一个分区中，消息是有顺序的方式存储着，每个在分区的消费都是有一个递增的id。这个就是偏移量offset
- 偏移量在分区中才是有意义的。在分区之间，offset是没有任何意义的

5.2 分区和副本机制

5.2.1 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

5.2.1.1 轮询策略

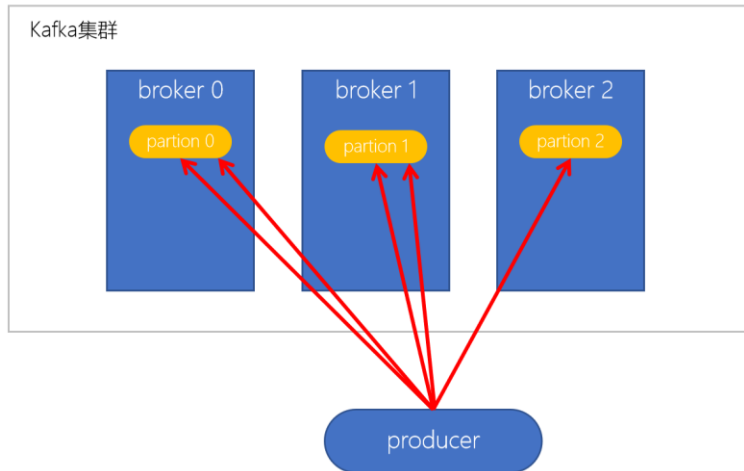


- 默认的策略，也是使用最多的策略，可以最大限度保证所有消息平均分配到一个分区
- 如果在生产消息时，key为null，则使用轮询算法均衡地分配分区

5.2.1.2 随机策略（不用）

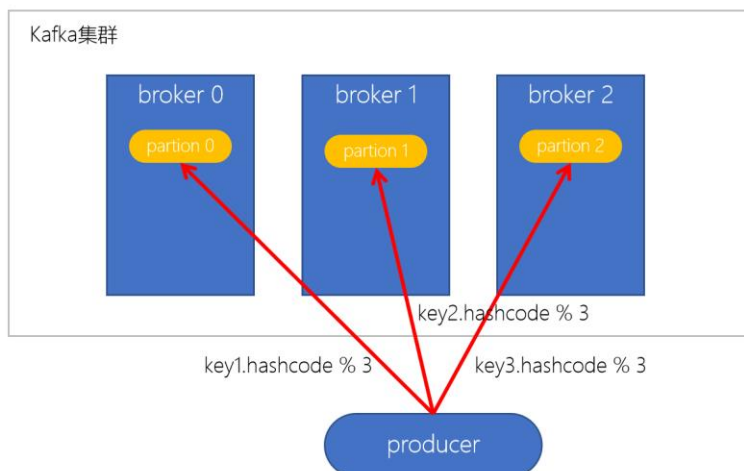
随机策略，每次都随机地将消息分配到每个分区。在较早的版本，默认的分区策略就是随机策略，也是为了将消息均衡地写入到每个分区。但后续轮询策略表现更佳，所以基本上很少会使用随机策略。

■ 随机分配策略



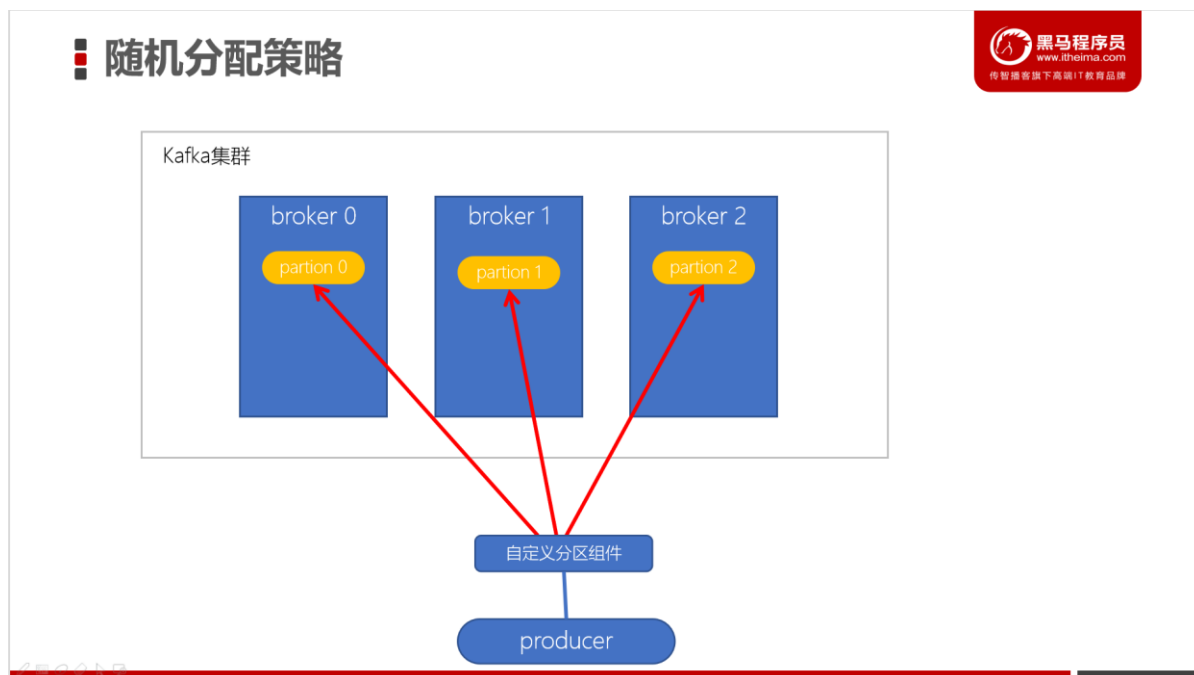
5.2.1.3 按key分配策略

■ 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。

5.2.1.4 自定义分区策略



要实现自定义分区，需要实现 Partitioner 接口。例如：以下的自定义分区，将带有key值的消息，使用随机的方式分配到不同的分区中。

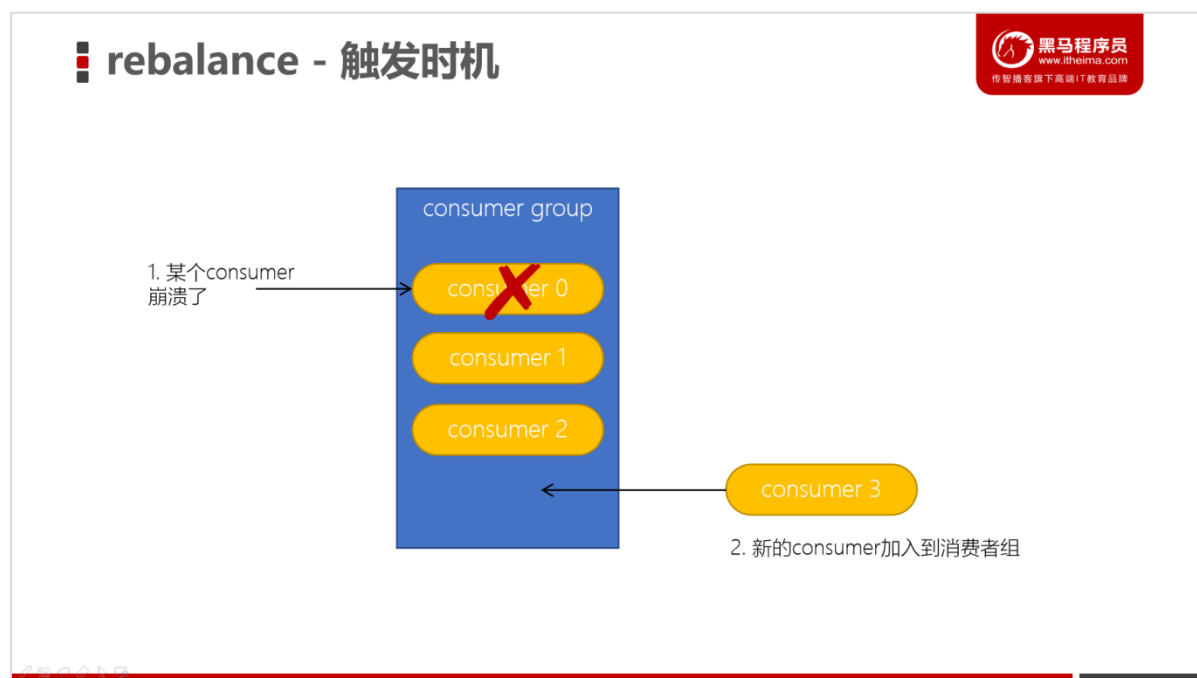
5.2.2 消费者分区分配策略

5.2.2.1 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。

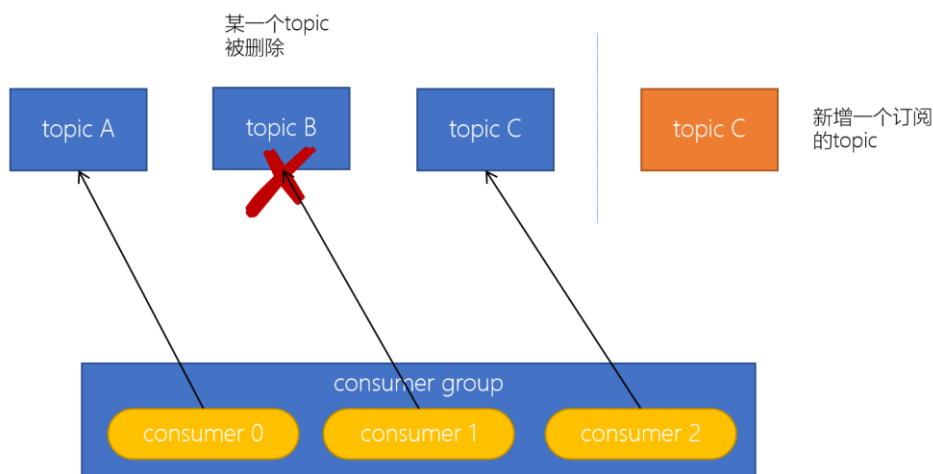
Rebalance触发的时机有：

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。



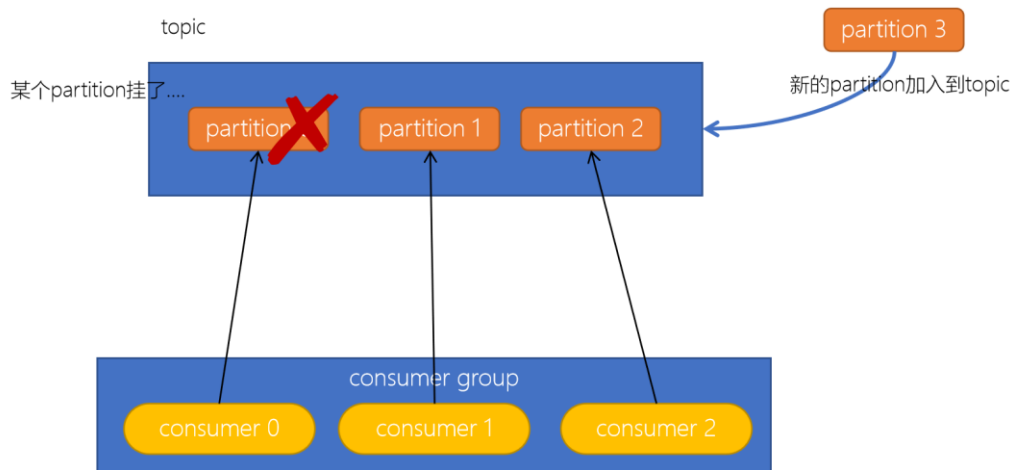
2. 订阅的topic个数发生变化

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

rebalance - 触发时机 - 分区数量变化



5.2.2.2 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

5.2.2.3 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。

对于每一个topic，RangeAssignor策略会将消费组内所有订阅这个topic的消费者按照名称的字典序排序，然后为每个消费者划分固定的分区范围，如果不够平均分配，那么字典序靠前的消费者会被多分配一个分区。

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

假设消费组内有2个消费者C0和C1，都订阅了主题t0和t1，并且每个主题都有4个分区，那么所订阅的所有分区可以标识为：t0p0、t0p1、t0p2、t0p3、t1p0、t1p1、t1p2、t1p3。最终的分配结果为：

消费者C0：t0p0、t0p1、t1p0、t1p1

消费者C1：t0p2、t0p3、t1p2、t1p3

我们再来看下另外一种情况。假设上面例子中2个主题都只有3个分区，那么所订阅的所有分区可以标识为：t0p0、t0p1、t0p2、t1p0、t1p1、t1p2。最终的分配结果为：

消费者C0：t0p0、t0p1、t1p0、t1p1

消费者C1：t0p2、t1p2

5.2.2.4 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，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。

如果同一个消费组内所有的消费者的订阅信息都是相同的，那么RoundRobinAssignor策略的分区分配会是均匀的。举例，假设消费组中有2个消费者C0和C1，都订阅了主题t0和t1，并且每个主题都有3个分区，那么所订阅的所有分区可以标识为：t0p0、t0p1、t0p2、t1p0、t1p1、t1p2。最终的分配结果为：

消费者C0：t0p0、t0p2、t1p1

消费者C1：t0p1、t1p0、t1p2

如果同一个消费组内的消费者所订阅的信息是不相同的，那么在执行分区分配的时候就不是完全的轮询分配，有可能会造成分区分配的不均匀。如果某个消费者没有订阅消费组内的某个topic，那么在分配分区的时候此消费者将分配不到这个topic的任何分区。

举例，假设消费组内有3个消费者C0、C1和C2，它们共订阅了3个主题：t0、t1、t2，这3个主题分别有1、2、3个分区，即整个消费组订阅了t0p0、t1p0、t1p1、t2p0、t2p1、t2p2这6个分区。具体而言，消费者C0订阅的是主题t0，消费者C1订阅的是主题t0和t1，消费者C2订阅的是主题t0、t1和t2，那么最终的分配结果为：

消费者C0：t0p0

消费者C1：t1p0

消费者C2：t1p1、t2p0、t2p1、t2p2

5.2.2.5 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rebalcen时，Striky粘性分配策略和RoundRobin分配策略类似。

Striky粘性分配策略

topicA: 4个分区



topicB: 4个分区



consumer0	topicA:p0 topicA:p3 topicB:p2
consumer1	topicA:p1 topicB:p0 topicB:p3
consumer2	topicA:p2 topicB:p1

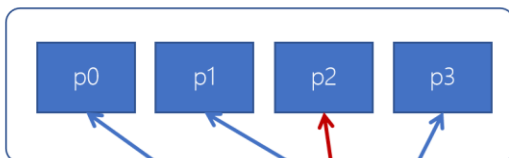


consumer group: 3个消费者共同消费8个分区

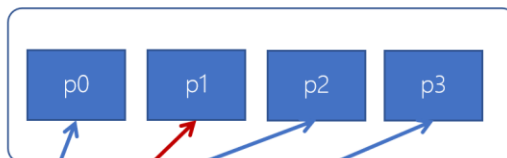
上面如果consumer2崩溃了，此时需要进行rebalance。

Striky粘性分配策略

topicA: 4个分区



topicB: 4个分区



rebalance前

consumer0	topicA:p0 topicA:p3 topicB:p2
consumer1	topicA:p1 topicB:p0 topicB:p3
consumer2	topicA:p2 topicB:p1



consumer group: 2个消费者共同消费8个分区

rebalance后

consumer0	topicA:p0 topicA:p3 topicB:p2 topicA:p2
consumer1	topicA:p1 topicB:p0 topicB:p3 topicB:p1
consumer2	

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）

5.2.3 副本机制

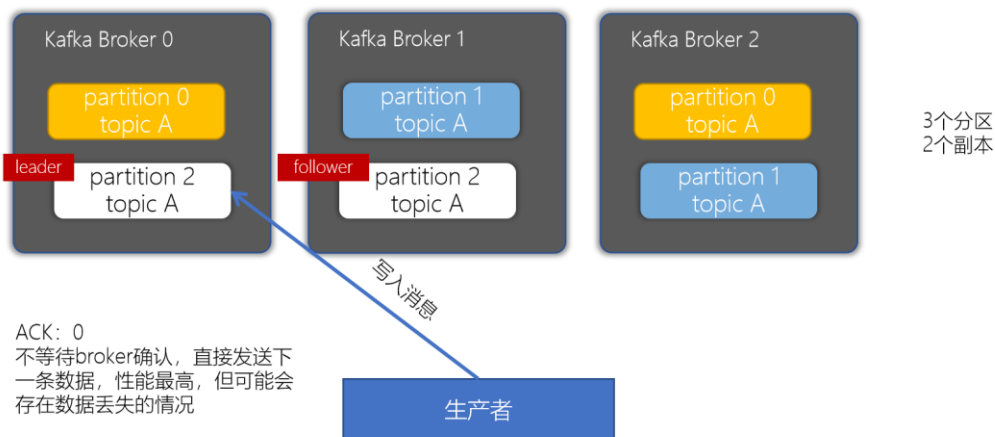
副本的目的就是冗余备份，当某个Broker上的分区数据丢失时，依然可以保障数据可用。因为在其他的Broker上的副本是可用的。

5.2.3.1 producer的ACKs参数

对副本关系较大的就是，producer配置的acks参数了，它决定了生产者如何在性能和可靠性之间做取舍。

```
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");
```

■ 副本机制 - ACKs (0)



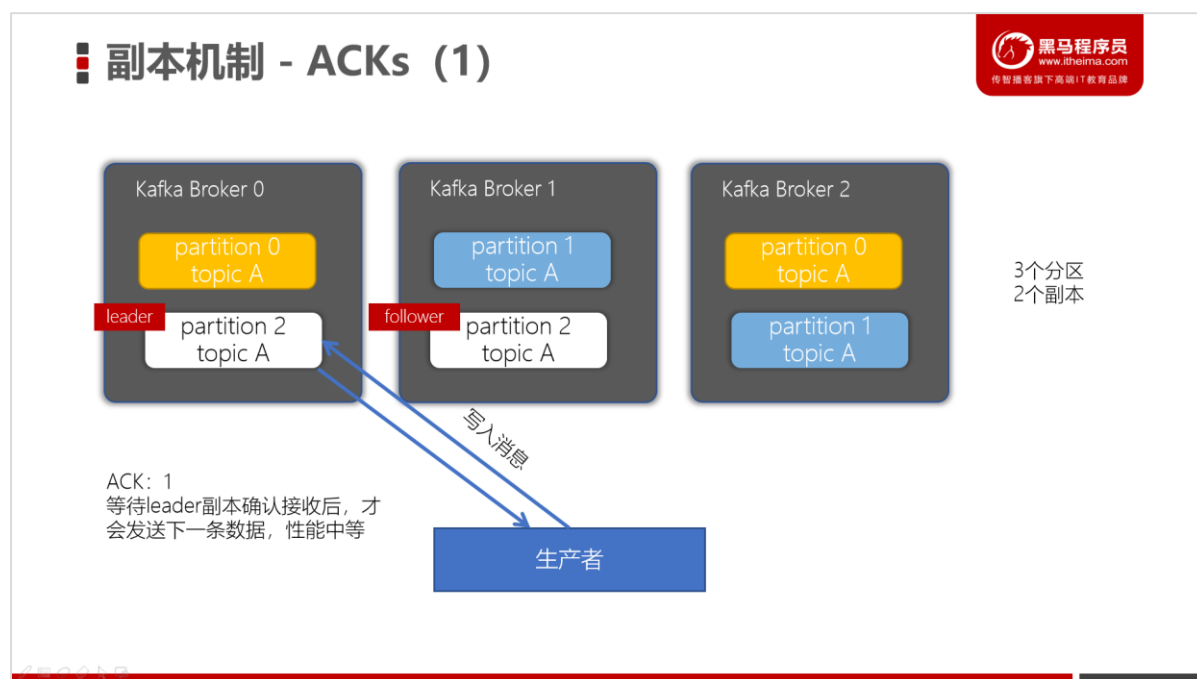
ACK为0，基准测试：

```
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
```

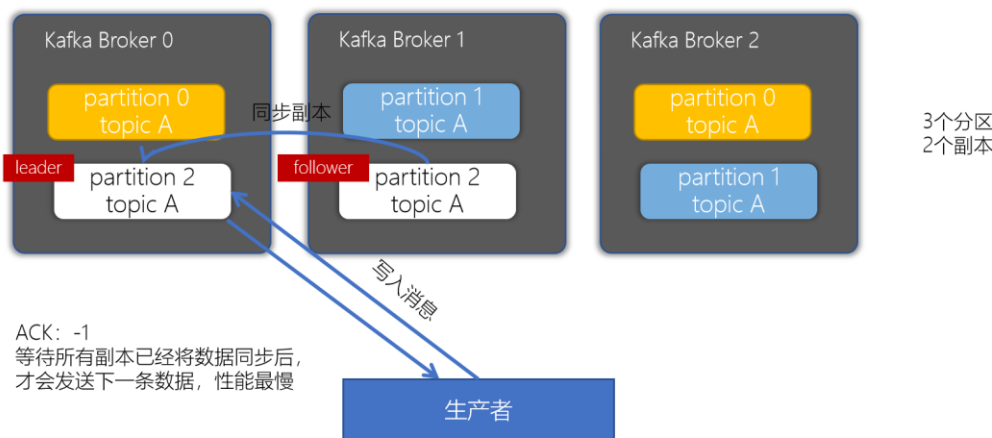
测试结果：

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



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。

■ 副本机制 - ACKs (-1)



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

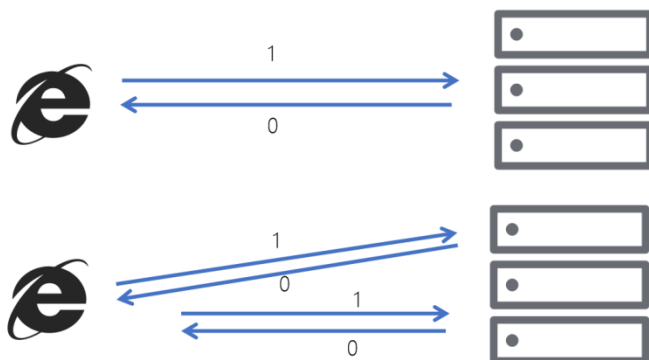
6. Kafka生产者幂等性与事务

6.1 幂等性

6.1.1 简介

拿http举例来说，一次或多次请求，得到地响应是一致的（网络超时等问题除外），换句话说，就是执行多次操作与执行一次操作的影响是一样的。

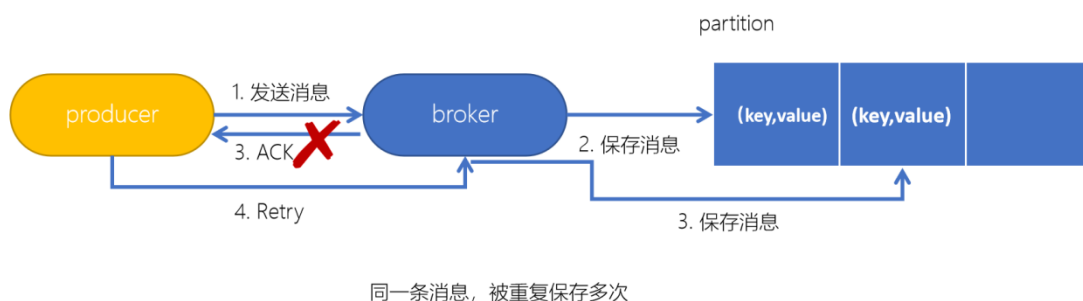
■ 幂等性



如果，某个系统是不具备幂等性的，如果用户重复提交了某个表格，就可能会造成不良影响。例如：用户在浏览器上点击了多次提交订单按钮，会在后台生成多个一模一样的订单。

6.1.2 Kafka生产者幂等性

■ Kafka幂等性



在生产者生产消息时，如果出现retry时，有可能会一条消息被发送了多次，如果Kafka不具备幂等性的，就有可能在partition中保存多条一模一样的消息。

6.1.3 配置幂等性

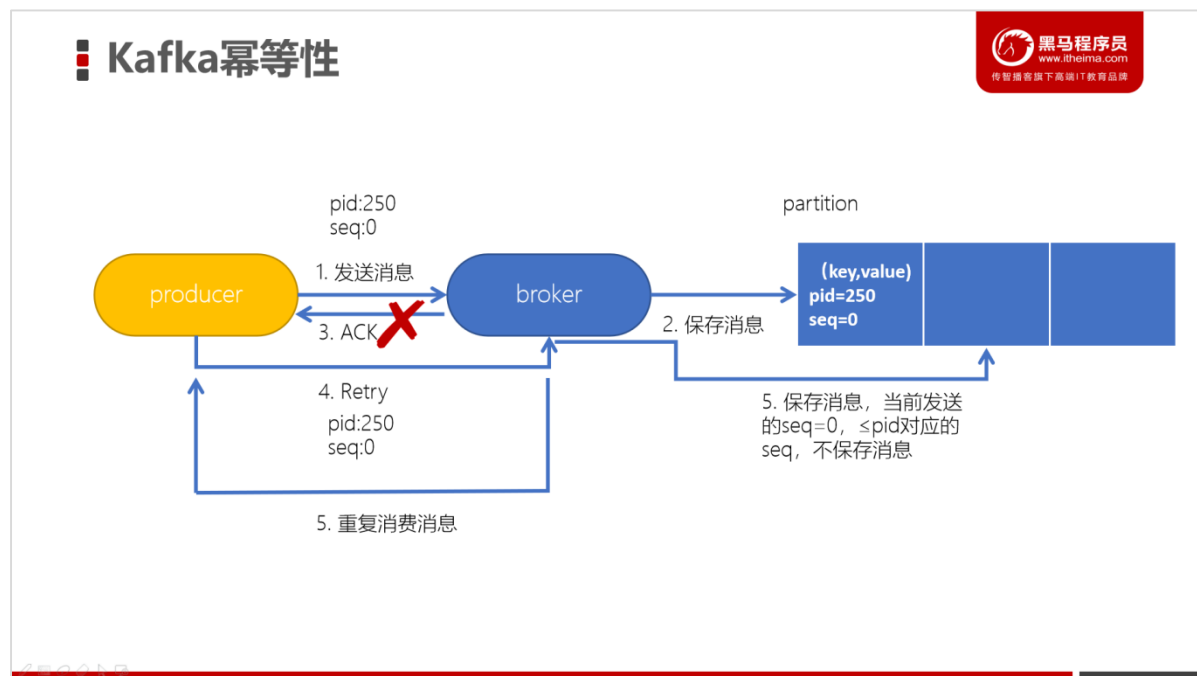
```
props.put("enable.idempotence",true);
```

如果幂等性配置为true，那此时默认会把acks设置为all，所以一旦设置了幂等性，就不再需要配置ACK了。

6.1.4 幂等性原理

为了实现生产者的幂等性，Kafka引入了 Producer ID (PID) 和 Sequence Number的概念。

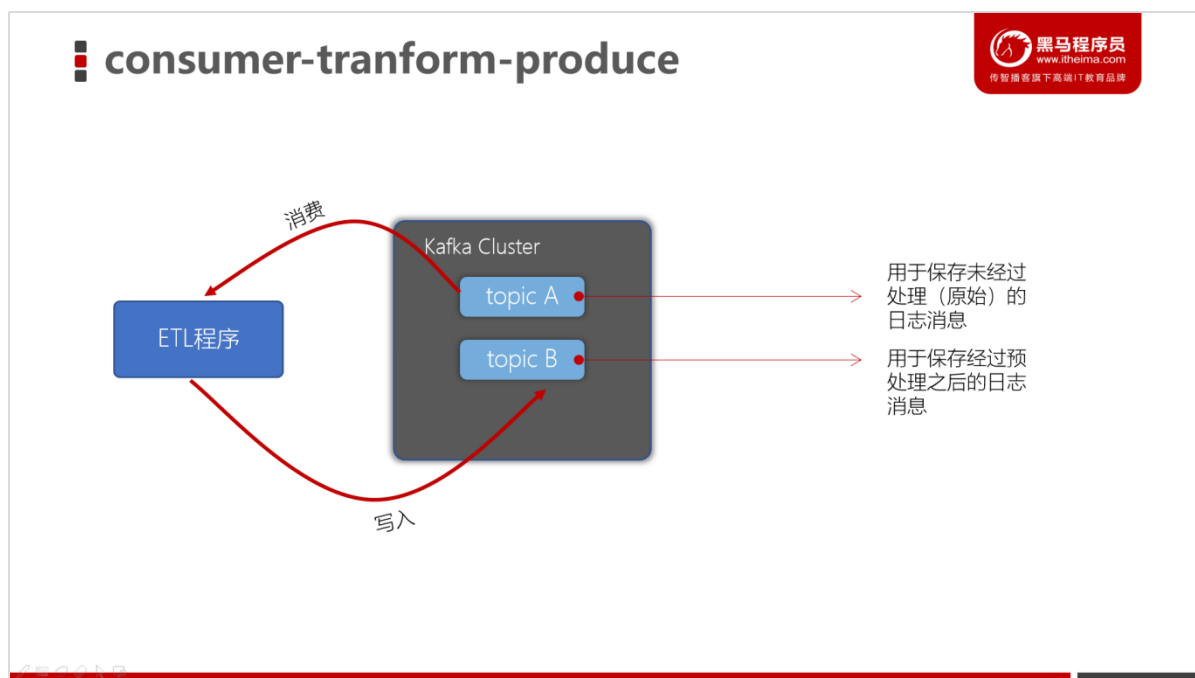
- PID：每个Producer在初始化时，都会分配一个唯一的PID，这个PID对用户来说，是透明的。
- Sequence Number：针对每个生产者（对应PID）发送到指定主题分区的信息都对应一个从0开始递增的Sequence Number。



6.2 Kafka事务

6.2.1 简介

Kafka事务是2017年Kafka 0.11.0.0引入的新特性。类似于数据库的事务。Kafka事务指的是生产者生产消息以及消费者提交offset的操作可以在一个原子操作中，要么都成功，要么都失败。尤其是在生产者、消费者并存时，事务的保障尤其重要。（consumer-transform-producer模式）



例如：生产者一次可以发送多个消息，要么都成功，要么都失败。或者消费者提交偏移量之前就挂了，此时执行re-balance时，其他消费者会重复消费消息。

6.2.2 事务操作API

Producer接口中定义了以下5个事务相关方法：

1. initTransactions (初始化事务)
2. beginTransaction (开始事务)
3. sendOffsets (提交偏移量)
4. commitTransaction (提交事务)
5. abortTransaction (放弃事务)

6.3 Kafka事务编程

6.3.1 事务相关属性配置

```
// 生产者：  
  
// 配置事务的id，开启了事务会默认开启幂等性
```



```
props.put("transactional.id", "first-transactional");
```

// 消费者

// 1. 消费者需要设置隔离级别

```
props.put("isolation.level", "read_committed");
```

// 2. 关闭自动提交

```
props.put("enable.auto.commit", "false");
```

// 3. 在代码中不再使用 commitSync()或者 commitAsync() 手动提交偏移量，统一由事务管理

// consumer.commitSync()