# 第1章 Kafka入门

**学习目标**

* 了解消息队列的应用场景
* 能够搭建Kafka集群
* 能够完成生产者、消费者Java代码编写
* 理解Kafka的架构，以及Kafka的重要概念
* 了解Kafka的事务

## 简介

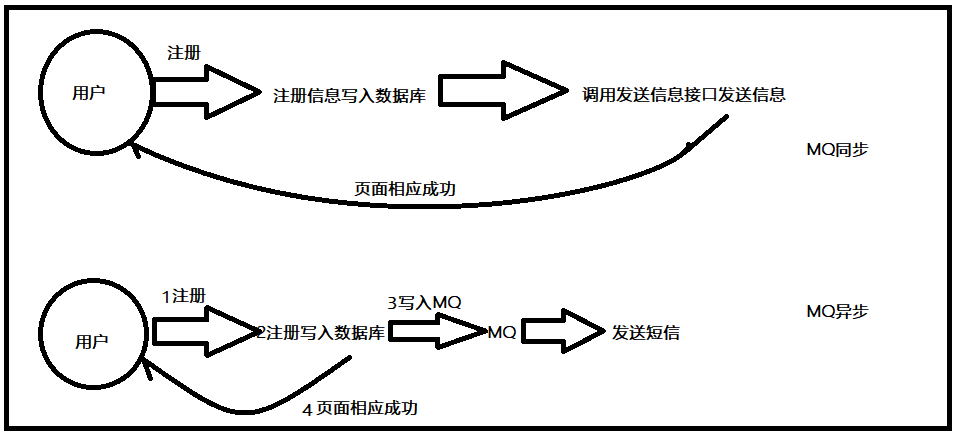
什么是消息？应用之间传递的数据就是消息，比如文本，对象

比如在**用户管理系统和财务管理系统**之间需要传递**用户id**来查询账户(画图分析)

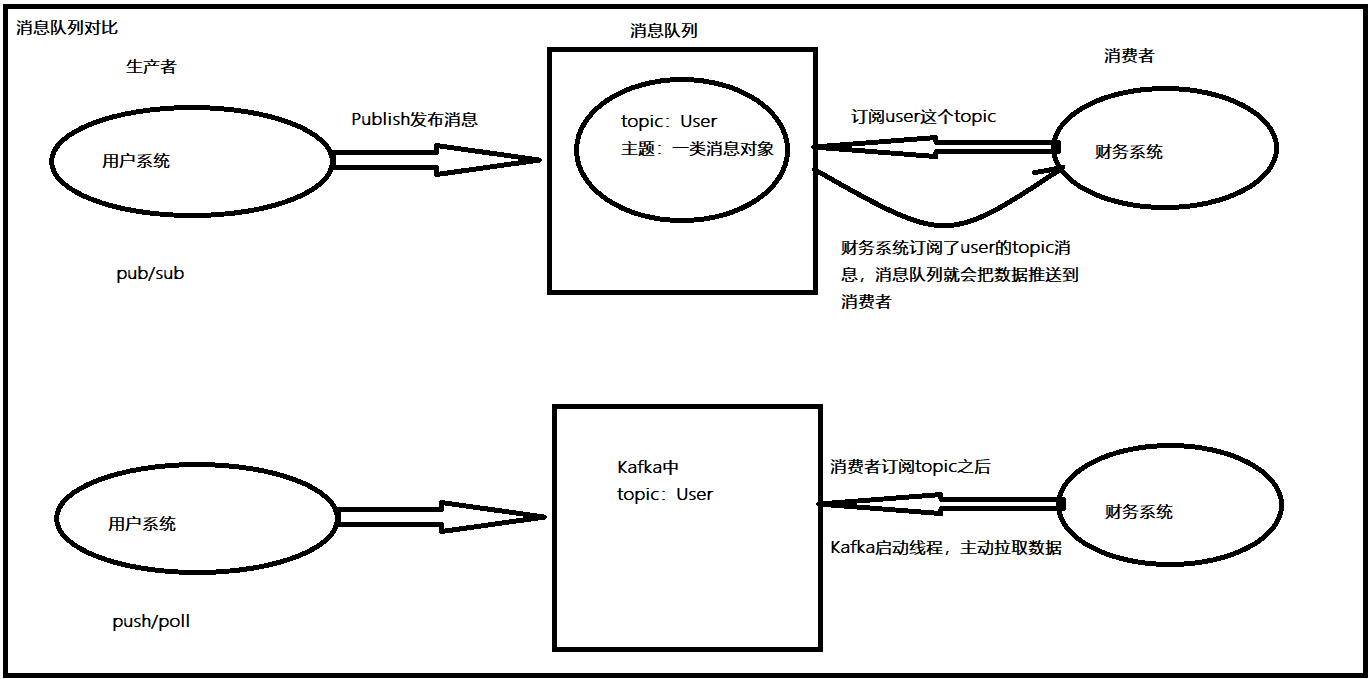
### 消息队列简介

什么是消息队列？应用之间通信方式，有消息系统确保信息可靠**传递。**

比如在**用户系统和财务系统增加MQ**，用户系统发送的username和password到，**MQ然后财务系统从MQ中获取消息**。



#### 传统消息队列



特点：

* 解耦合：MQ隔离两套业务系统
* 异步处理：业务需求可以基于消费者做各种事情，比如新用户可以发新手红包，可以授权认证等
* 限流消峰：访问量激增情况下限制流量
* 消息驱动(日志处理)

#### 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。来看一下下面的代码：

|  |
| --- |
| // 1. 创建一个保存字符串的队列 Queue<String> stringQueue = **new** LinkedList<String>();  // 2. 往消息队列中放入消息 stringQueue.offer(**"hello"**);  // 3. 从消息队列中取出消息并打印 System.**out**.println(stringQueue.poll()); |

上述代码，创建了一个队列，先往队列中添加了一个消息，然后又从队列中取出了一个消息。这说明了队列是可以用来存取消息的。

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**。

#### 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。

##### 为什么叫Kafka呢

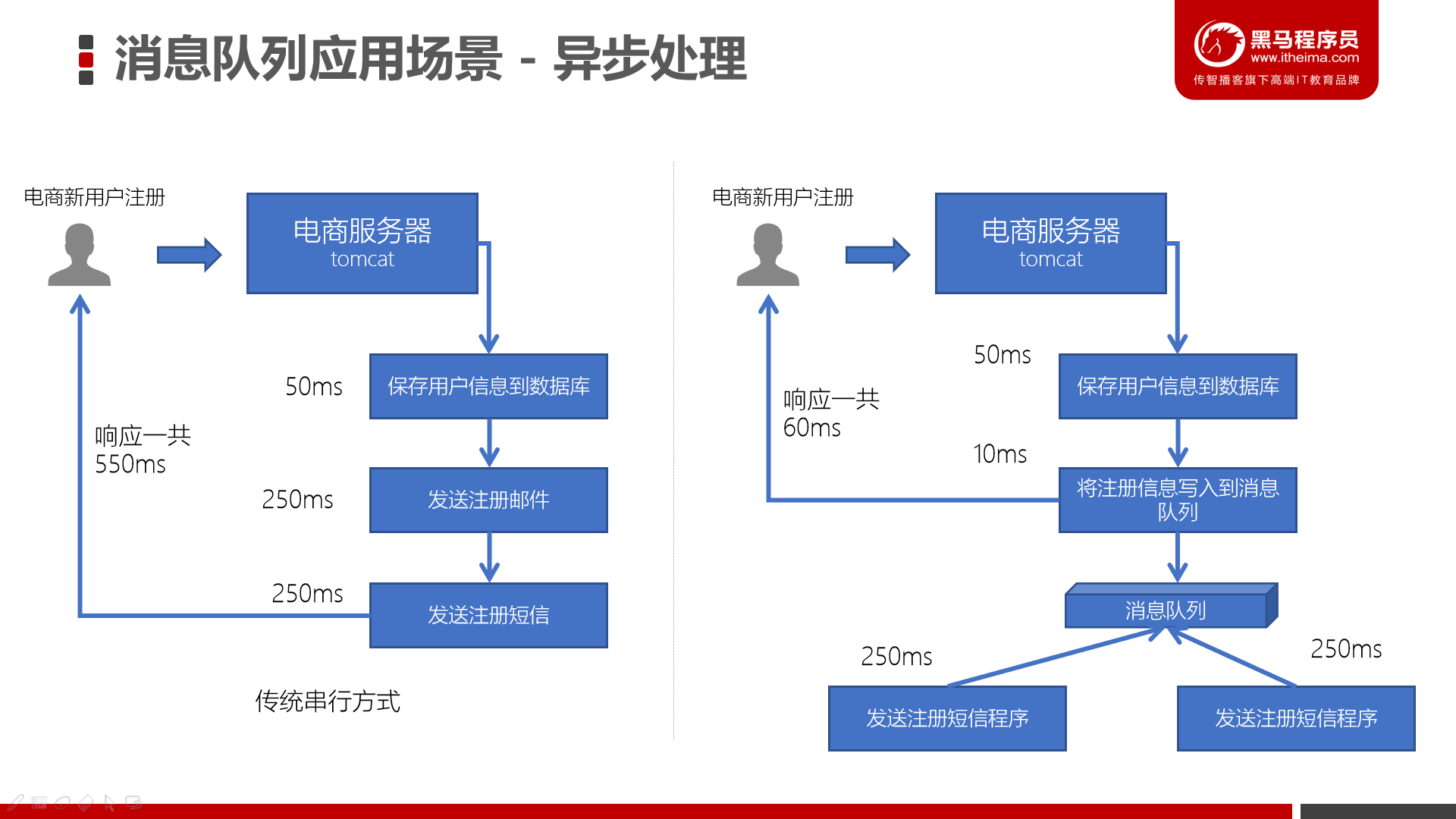
Kafka的架构师jay kreps非常喜欢franz kafka（弗兰兹·卡夫卡）,并且觉得kafka这个名字很酷，因此取了个和消息传递系统完全不相干的名称kafka，该名字并没有特别的含义。

「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」

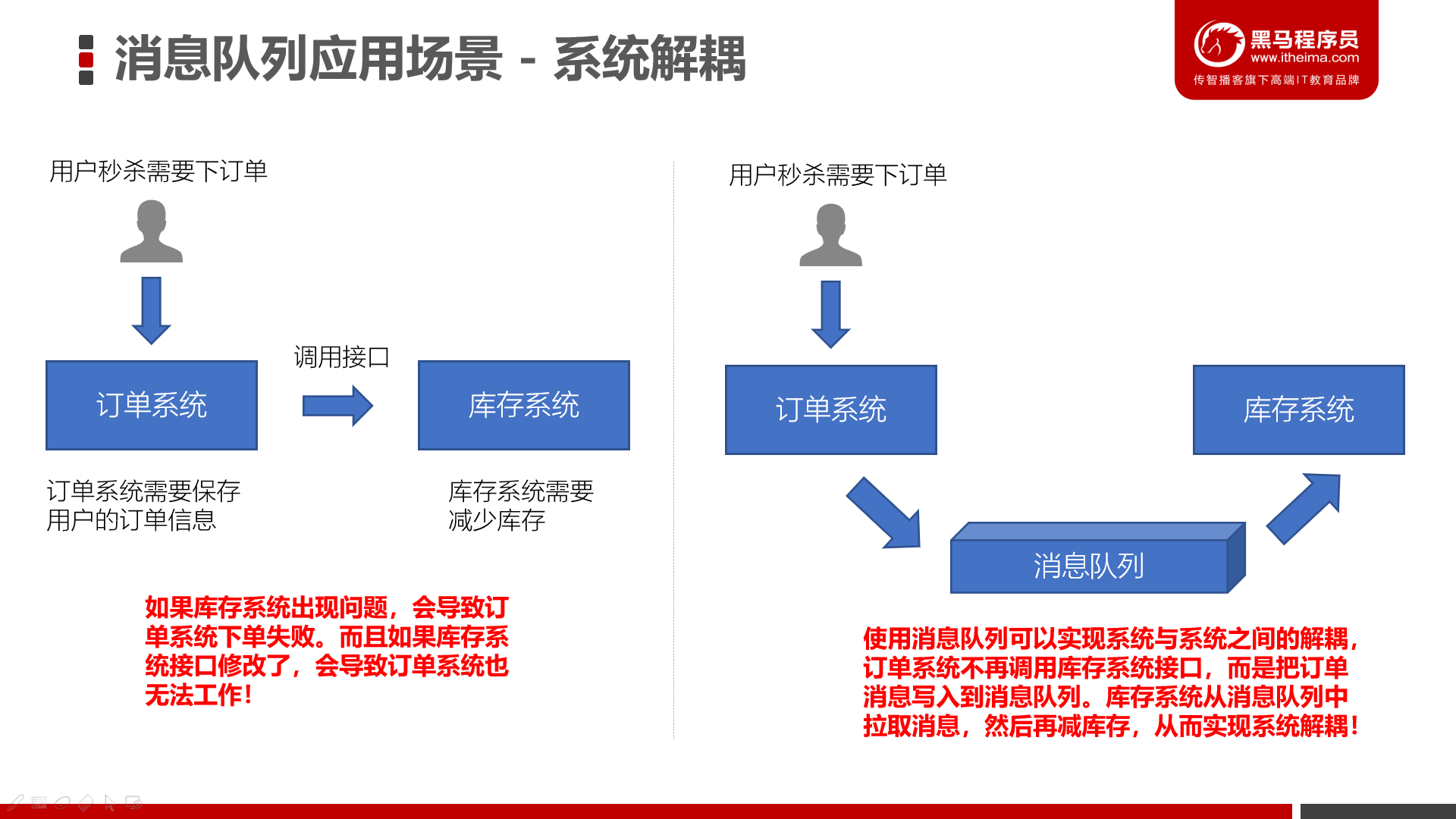
#### 消息队列的应用场景

##### 异步处理

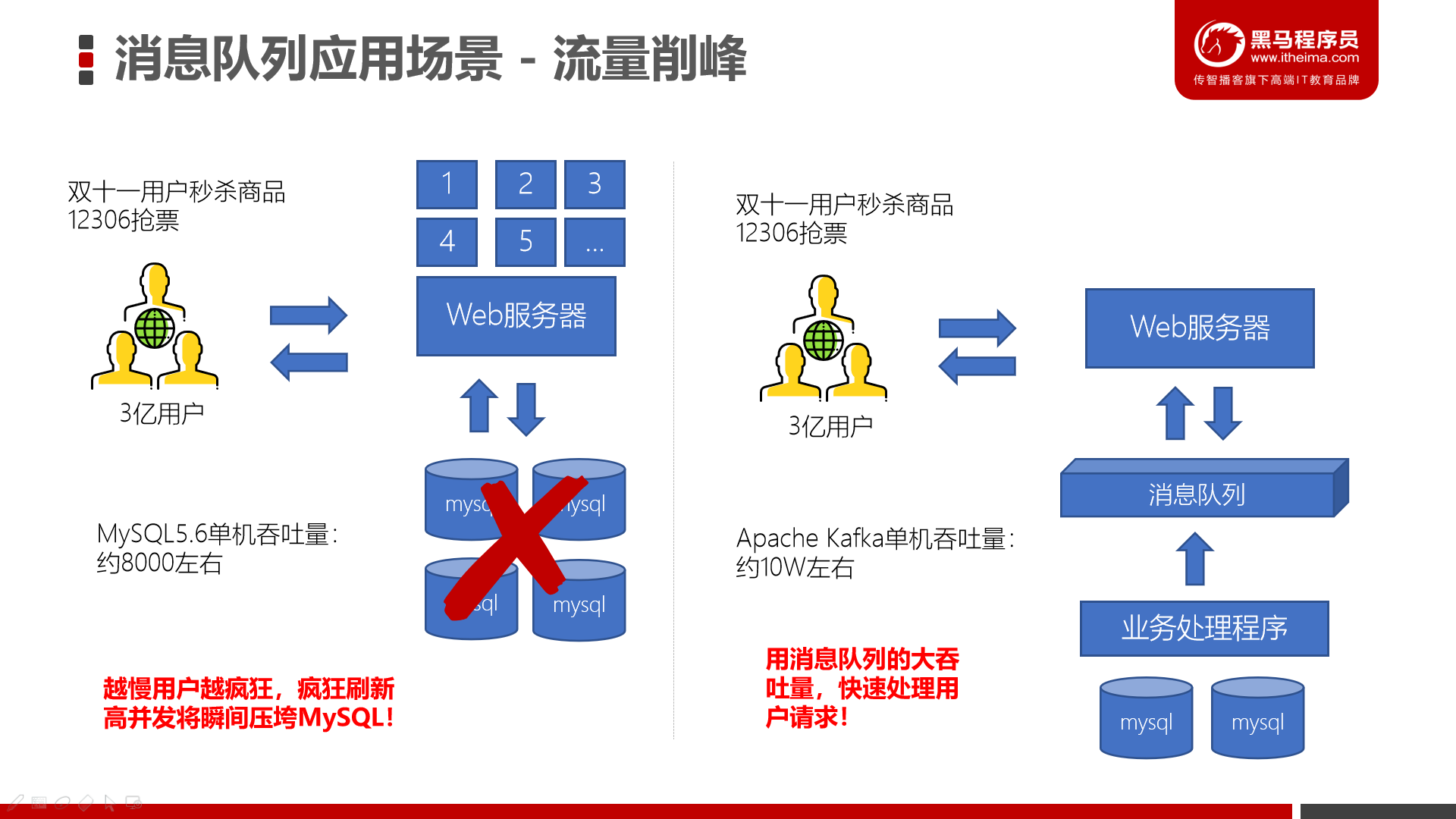
电商网站中，新的用户注册时，需要将用户的信息保存到数据库中，同时还需要额外发送注册的邮件通知、以及短信注册码给用户。但因为发送邮件、发送注册短信需要连接外部的服务器，需要额外等待一段时间，此时，就可以使用消息队列来进行异步处理，从而实现快速响应。



##### 系统解耦

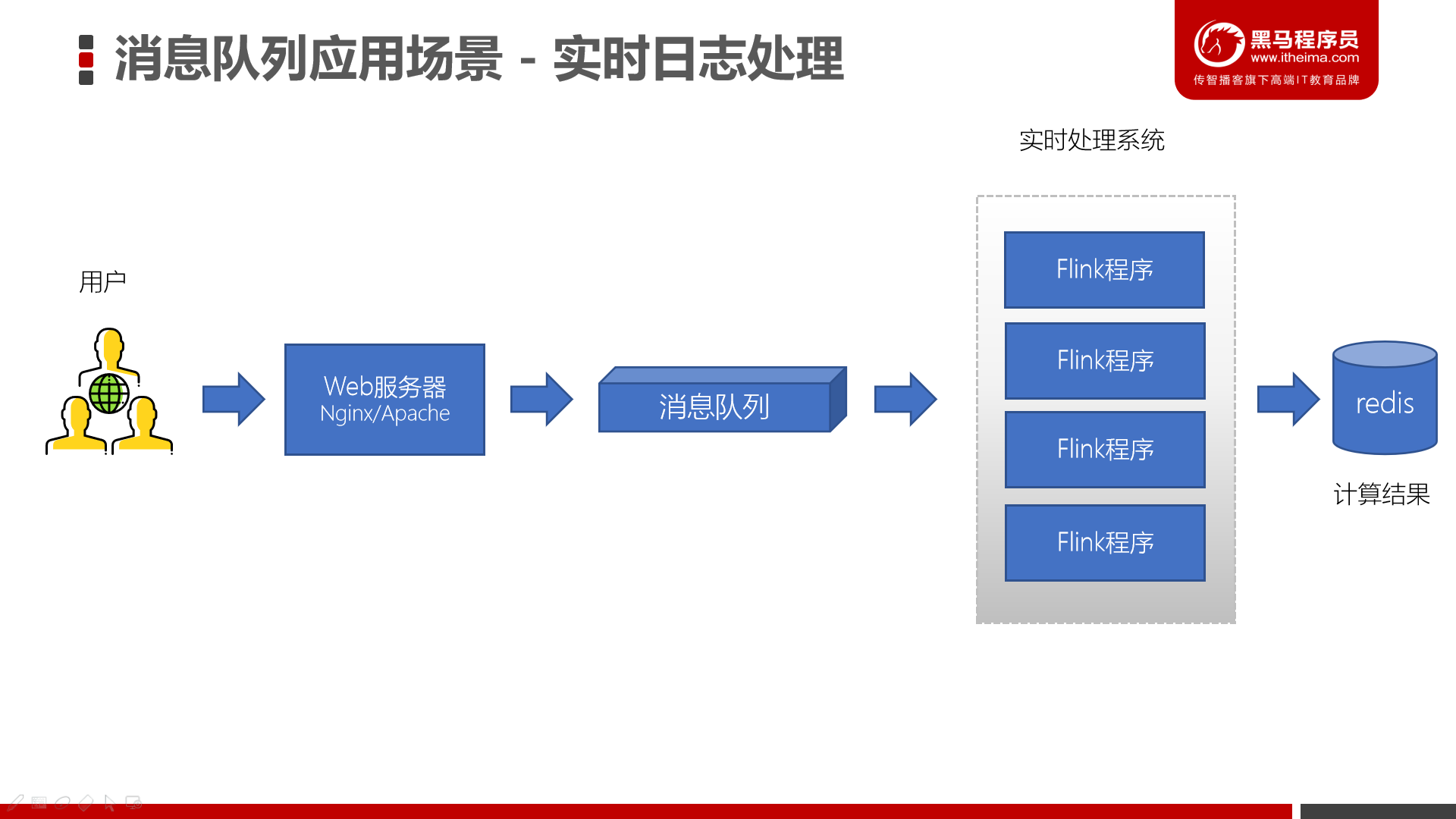


##### 流量削峰



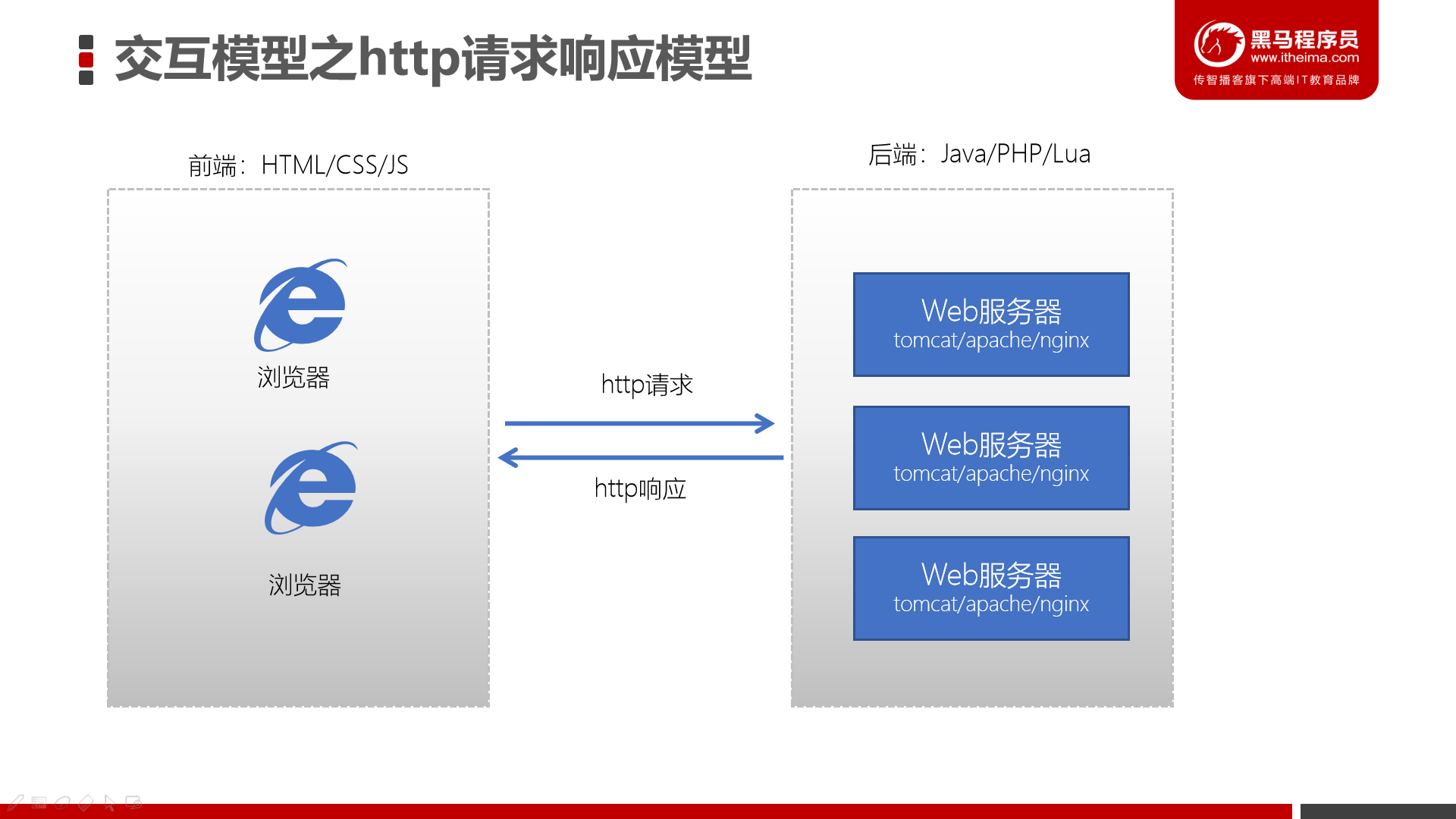
##### 日志处理（大数据领域常见）

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。

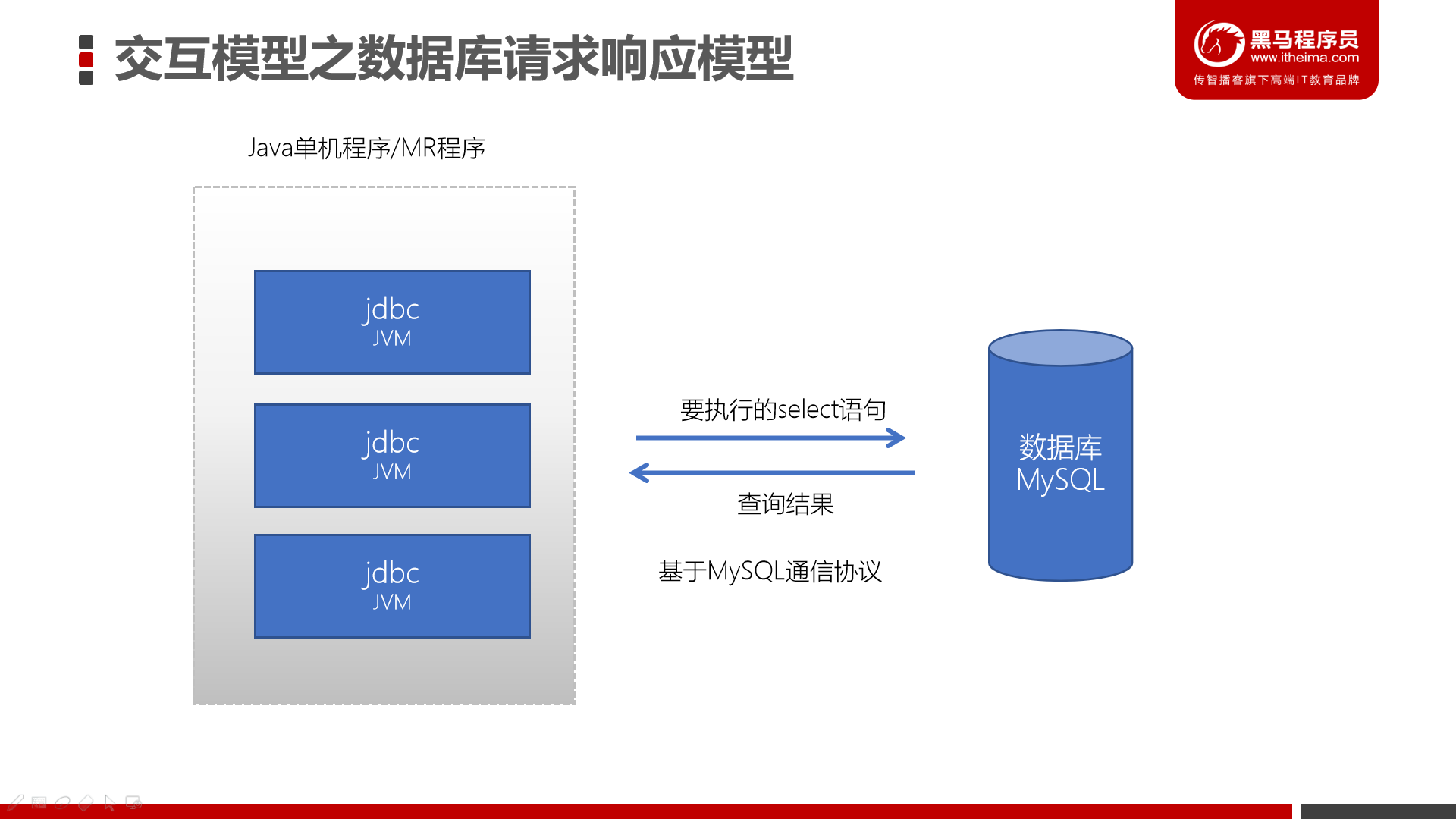


#### 生产者、消费者模型

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：

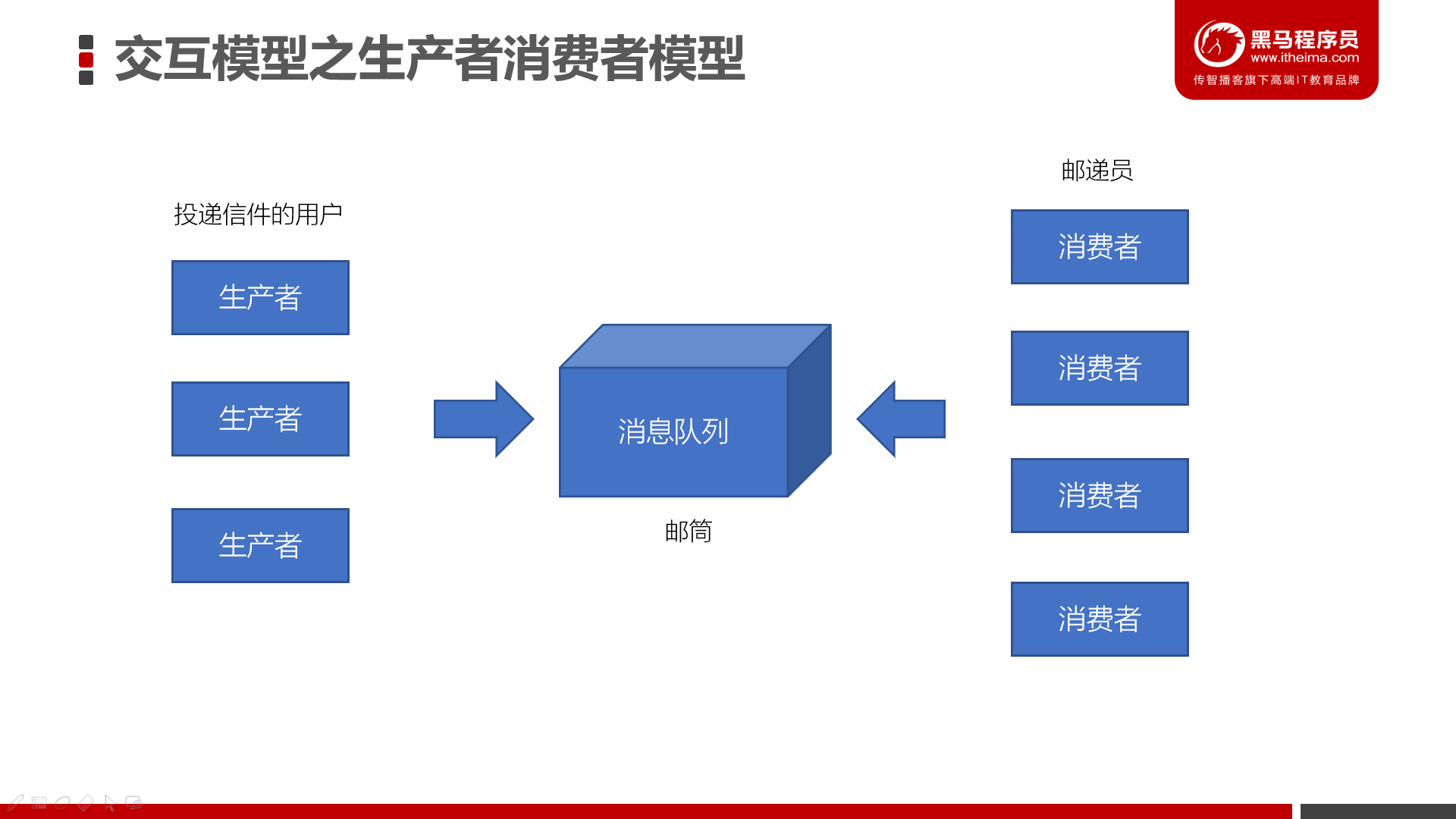


我们之前也学习过使用Java JDBC来访问操作MySQL数据库，它的交互模型是这样的：



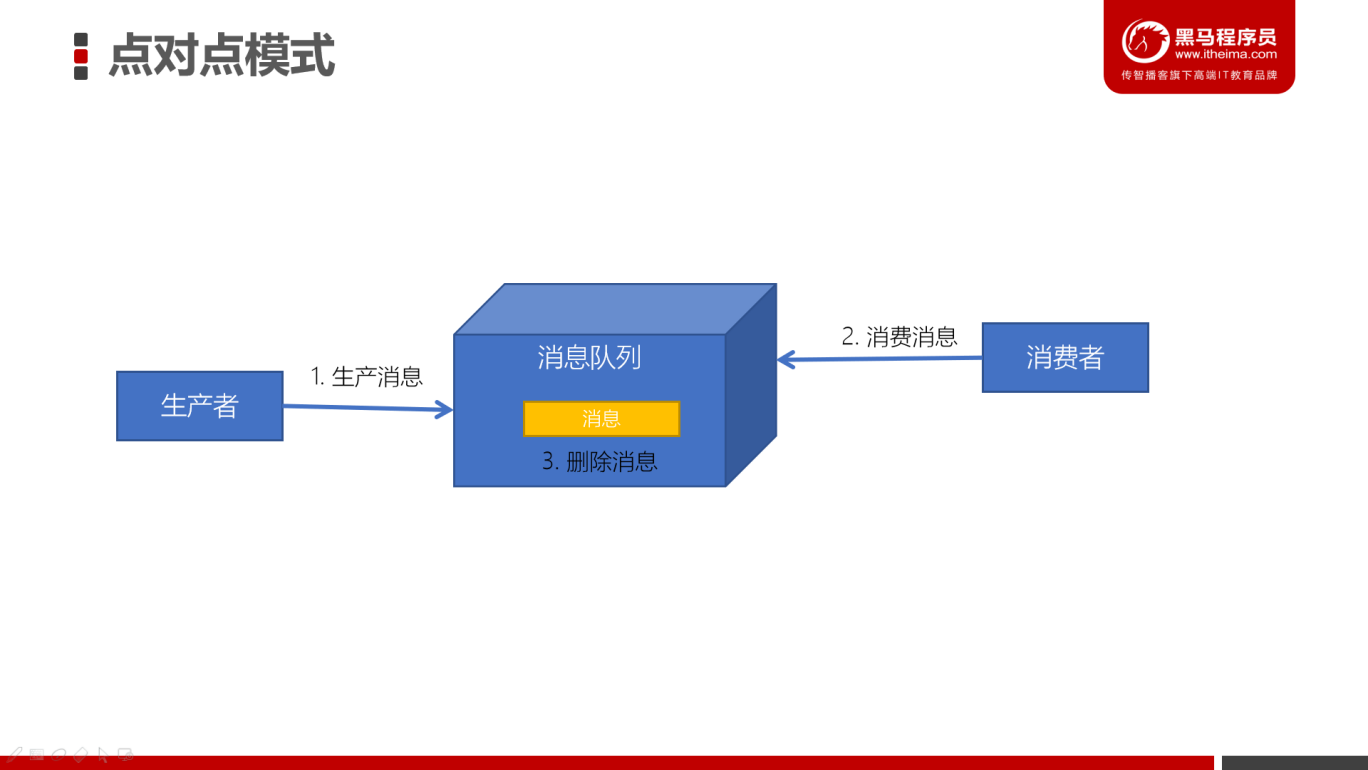
它也是一种**请求响应模型**，只不过它不再是基于http协议，而是基于MySQL数据库的通信协议。

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。



#### 消息队列的两种模式

##### 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；

##### 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；

### Kafka简介

#### 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Kakfa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：

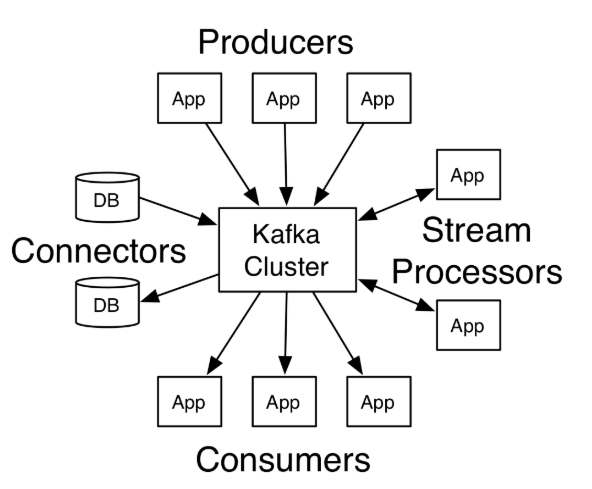
1. **Publish and subscribe：发布与订阅**
2. **Store：存储-🡪文本中存储，保存在本地磁盘上，并在集群中复制以防止数据丢失**
3. **Process：处理🡪指标分析，日志聚合，流式处理**

后续我们的课程主要围绕这三点来讲解。

#### 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。

1. Stream Processors：**流处理器可以Kafka中拉取数据**，也可以将数据写入到Kafka中。

#### Kafka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 |

### 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 |
| 单机呑吐量 | 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，不论成成熟度、社区、性能、可靠性，Kafka都是非常有竞争力的一款产品。

### 哪些公司在使用Kafka

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### 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>



### Kafka版本

本次课程使用的Kafka版本为2.4.1，是2020年3月12日发布的版本。

可以注意到Kafka的版本号为：kafka\_2.12-2.4.1，因为kafka主要是使用scala语言开发的，2.12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。

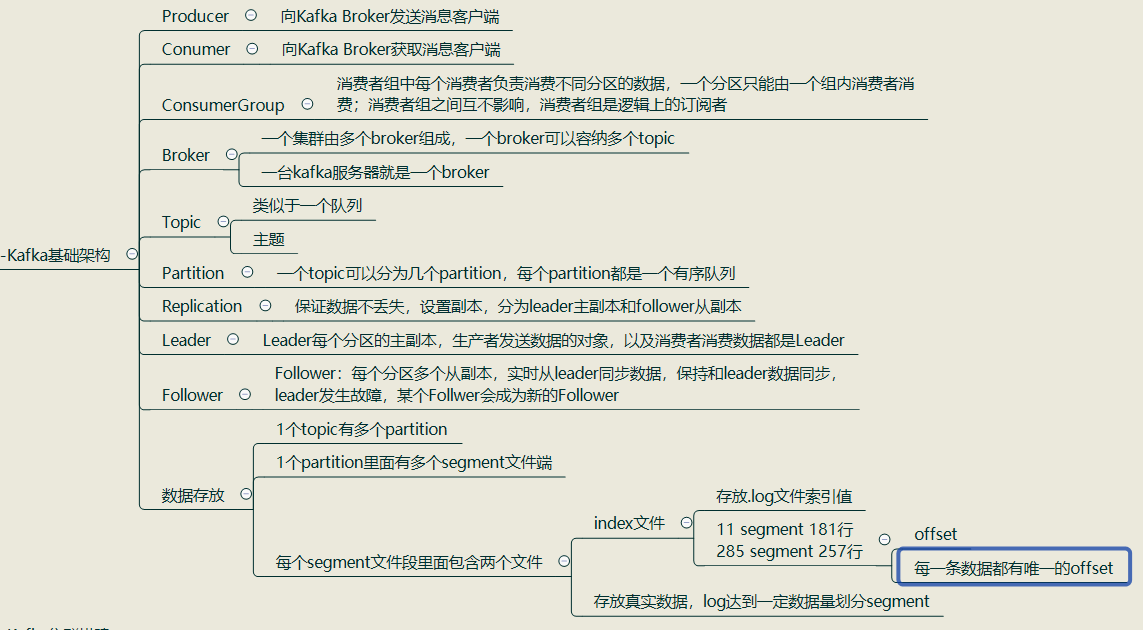
## 环境搭建

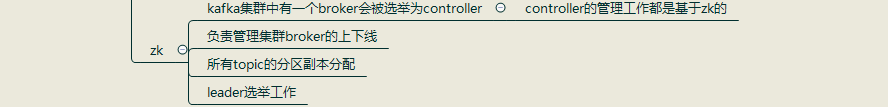
**Kafak（1）Scala开发（2）LinkedIn开发（3）分布式消息队列（4）依赖ZK保存Meta信息**

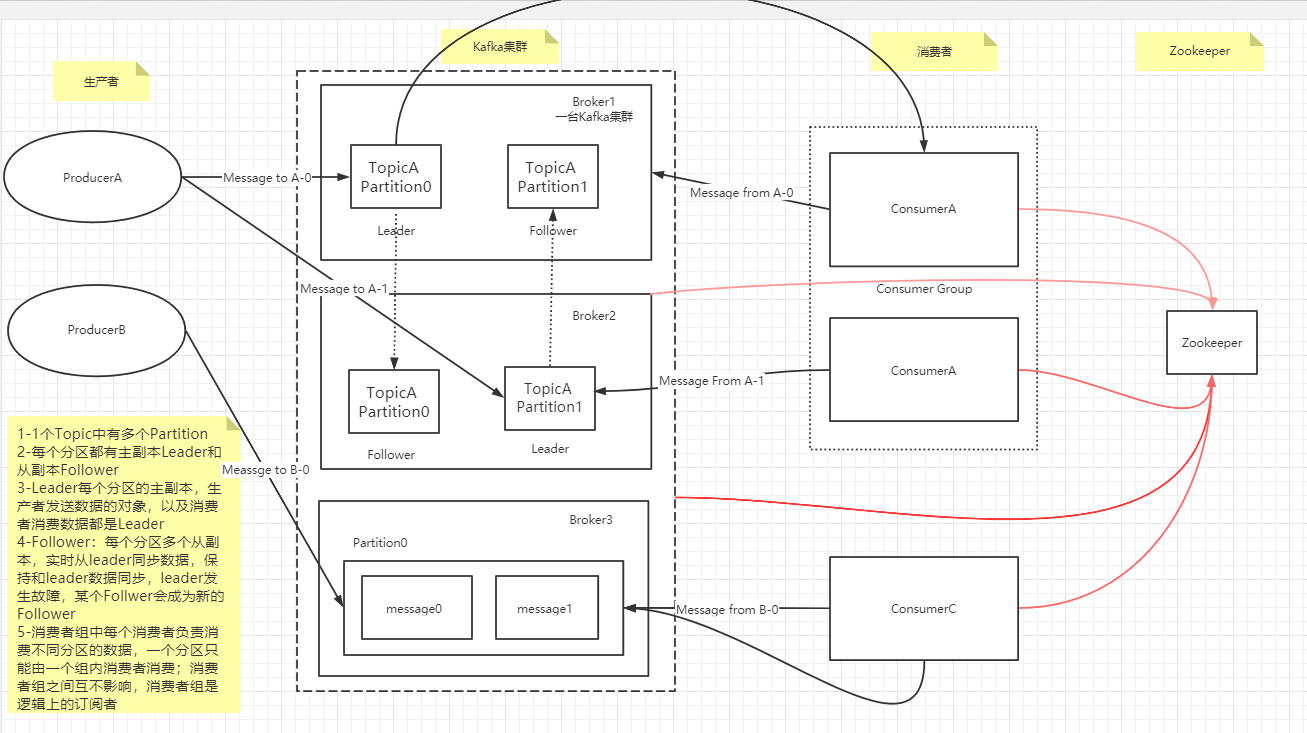
**Kafka将消息保留在磁盘上，并在集群中复制以防止数据丢失**

**Kafka应用于指标分析，日志聚合，流式处理**

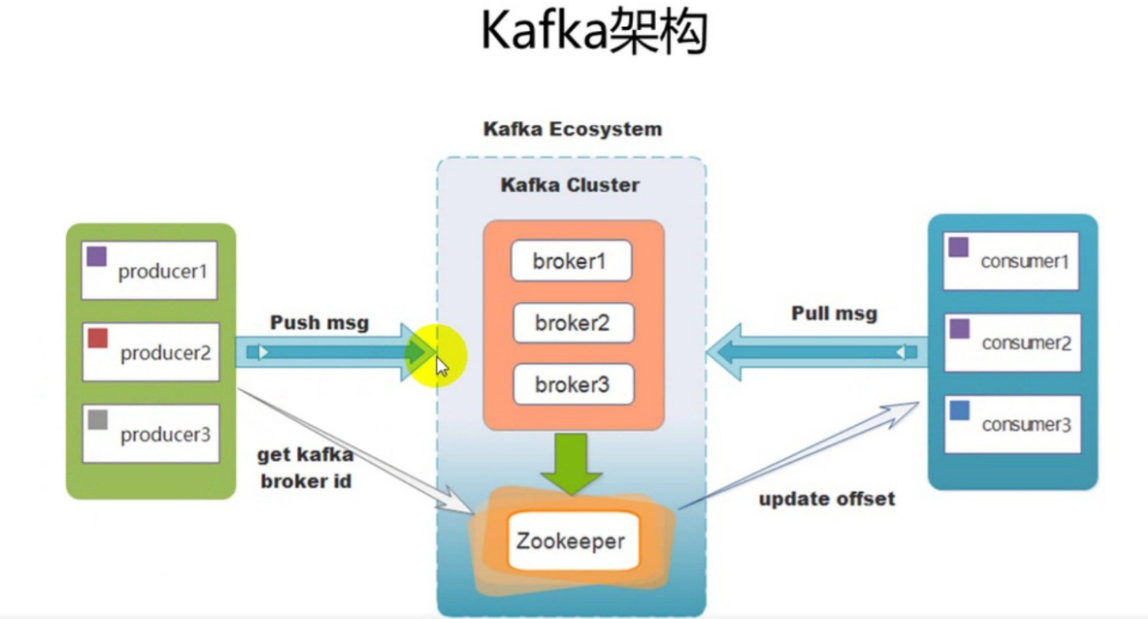
### Kafka数据模型

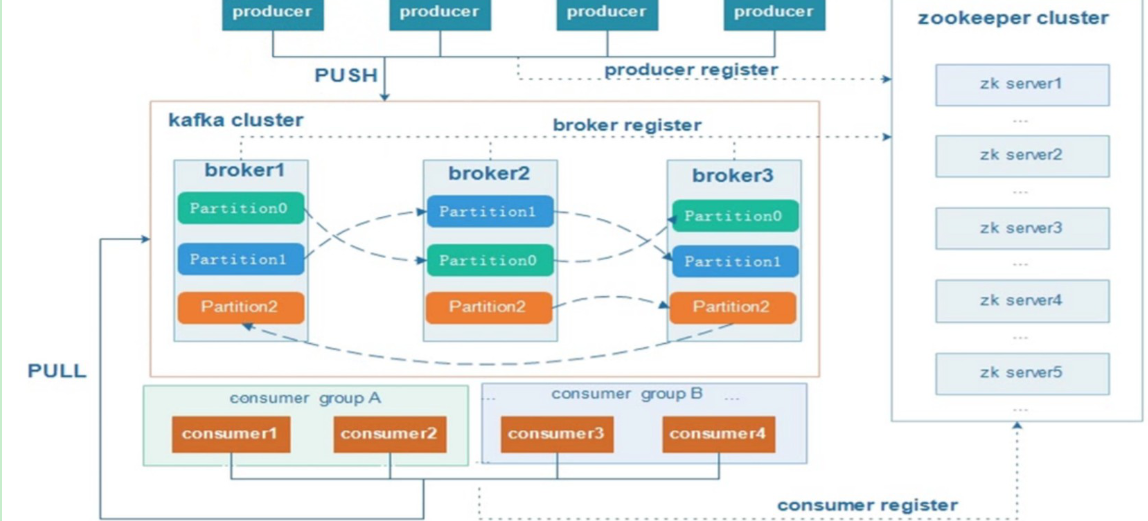






其他图示





### 搭建Kafka集群

1. 将Kafka的安装包上传到虚拟机，并解压

|  |
| --- |
| cd /export/software/  tar -xvzf kafka\_2.12-2.4.1.tgz -C ../server/  cd /export/server/kafka\_2.12-2.4.1/ |

1. 修改 server.properties

|  |
| --- |
| cd /export/server/kafka\_2.12-2.4.1/config  vim server.properties  # 指定broker的id  broker.id=0  # 指定Kafka数据的位置  log.dirs=/export/server/kafka\_2.12-2.4.1/data  # 配置zk的三个节点  zookeeper.connect=node1.itcast.cn:2181,node2.itcast.cn:2181,node3.itcast.cn:2181 |

1. 将安装好的kafka复制到另外两台服务器

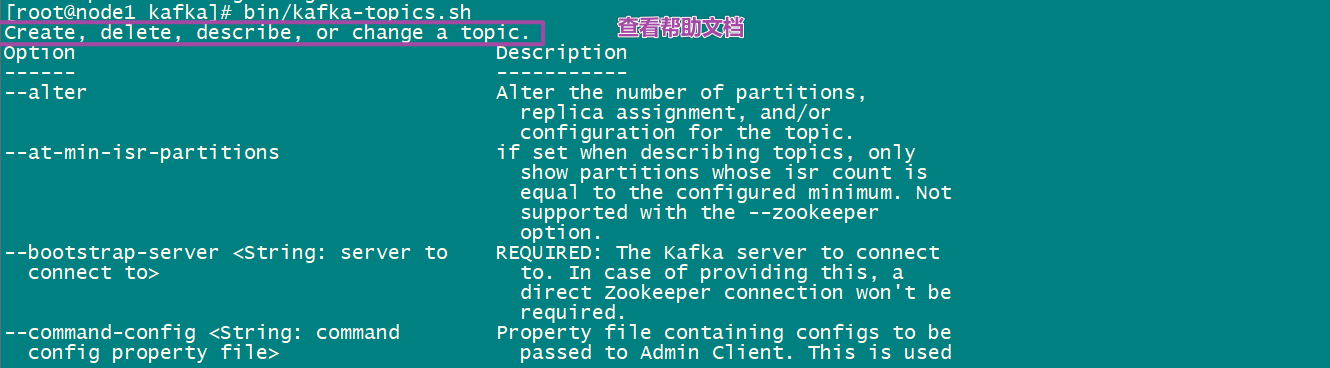
|  |
| --- |
| cd /export/server  scp -r kafka\_2.12-2.4.1/ node2.itcast.cn:$PWD  scp -r kafka\_2.12-2.4.1/ node3.itcast.cn:$PWD  修改另外两个节点的broker.id分别为1和2  ---------node2.itcast.cn--------------  cd /export/server/kafka\_2.12-2.4.1/config  vim erver.properties  broker.id=1  --------node3.itcast.cn--------------  cd /export/server/kafka\_2.12-2.4.1/config  vim server.properties  broker.id=2 |

1. 配置KAFKA\_HOME环境变量

|  |
| --- |
| vim /etc/profile  export KAFKA\_HOME=/export/server/kafka\_2.12-2.4.1  export PATH=:$PATH:${KAFKA\_HOME}vim  分发到各个节点  scp /etc/profile node2.itcast.cn:$PWD  scp /etc/profile node3.itcast.cn:$PWD  每个节点加载环境变量  source /etc/profile |

1. 启动服务器

|  |
| --- |
| # 启动ZooKeeper  nohup bin/zookeeper-server-start.sh config/zookeeper.properties &  # 启动Kafka  cd /export/server/kafka\_2.12-2.4.1  nohup bin/kafka-server-start.sh config/server.properties &  # 测试Kafka集群是否启动成功  bin/kafka-topics.sh --bootstrap-server node1.itcast.cn:9092 --list |



### 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 |

### Kafka一键启动/关闭脚本

为了方便将来进行一键启动、关闭Kafka，我们可以编写一个shell脚本来操作。将来只要执行一次该脚本就可以快速启动/关闭Kafka。

1. 在节点1中创建 /export/onekey 目录

cd /export/onekey

1. 准备slave配置文件，用于保存要启动哪几个节点上的kafka

|  |
| --- |
| node1.itcast.cn  node2.itcast.cn  node3.itcast.cn |

1. 编写start-kafka.sh脚本

|  |
| --- |
| **vim start-kafka.sh**  cat /export/onekey/slave | while read line  do  {  echo $line  ssh $line "source /etc/profile;export JMX\_PORT=9988;nohup ${KAFKA\_HOME}/bin/kafka-server-start.sh ${KAFKA\_HOME}/config/server.properties >/dev/nul\* 2>&1 & "  }&  wait  done |

1. 编写stop-kafka.sh脚本

|  |
| --- |
| **vim stop-kafka.sh**  cat /export/onekey/slave | while read line  do  {  echo $line  ssh $line "source /etc/profile;jps |grep Kafka |cut -d' ' -f1 |xargs kill -s 9"  }&  wait  done |

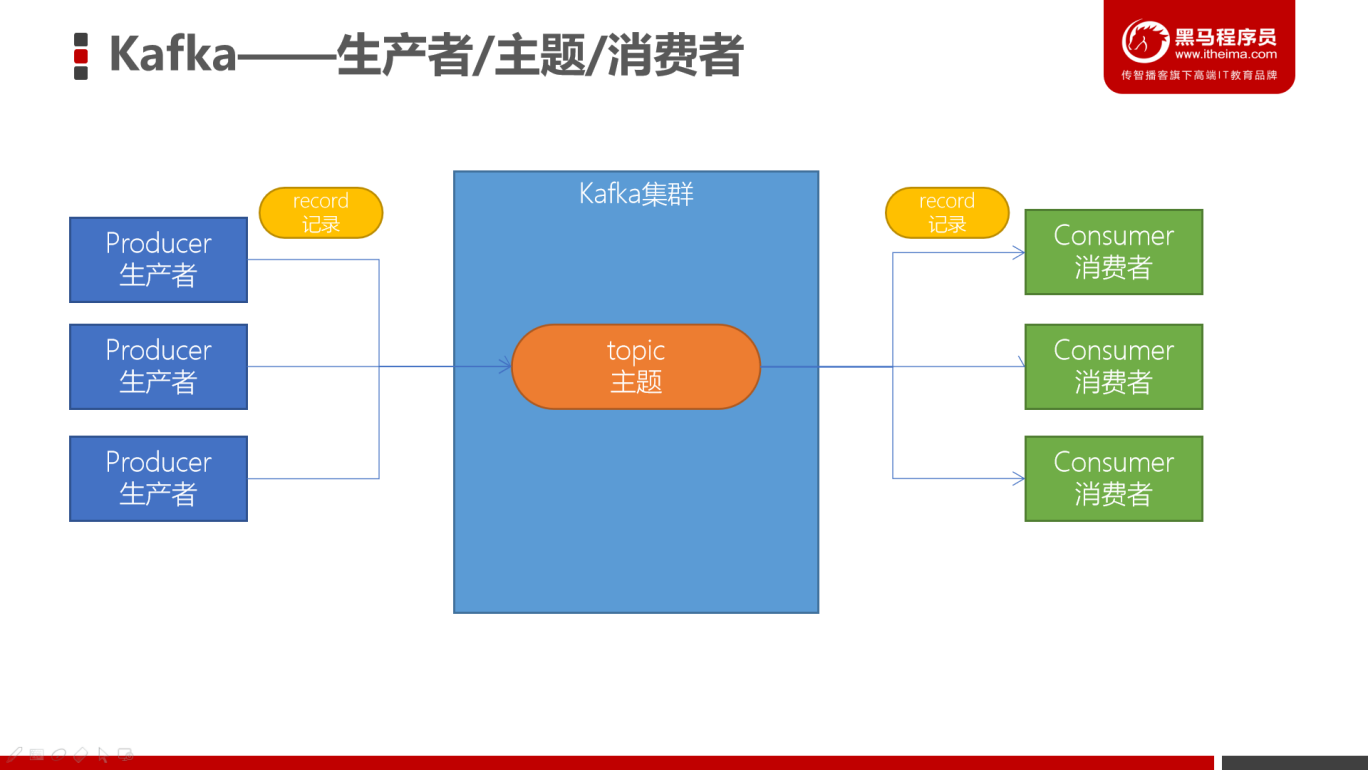
1. 给start-kafka.sh、stop-kafka.sh配置执行权限

|  |
| --- |
| chmod u+x start-kafka.sh  chmod u+x stop-kafka.sh |

1. 执行一键启动、一键关闭

|  |
| --- |
| ./start-kafka.sh  ./stop-kafka.sh |

## 基础操作



### 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 --bootstrap-server node1.itcast.cn:9092 --topic test  # 增加zk,属于过期淘汰的配置，参看kafka-topics.sh –help  bin/kafka-topics.sh –create --zookeeper node1.itcast.cn:2181 --topic test  # 查看目前Kafka中的主题  bin/kafka-topics.sh --list --bootstrap-server node1.itcast.cn:9092 |

查看帮助文- -help即可

案例：创建一个名字为test的主题， 有三个分区，有两个副本

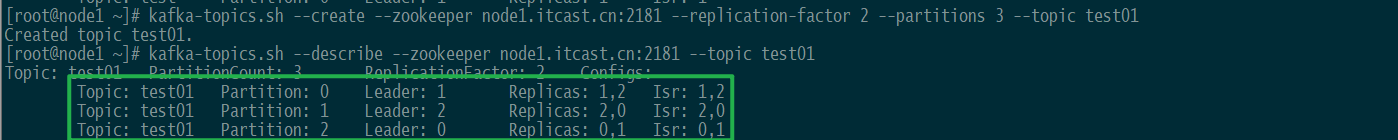
|  |
| --- |
| bin/kafka-topics.sh --create --bootstrap-server node1.itcast.cn:9092 --replication-factor 2 --partitions 3 --topic test |

### 查看Topic

|  |
| --- |
| 建议：  bin/kafka-topics.sh --list --bootstrap-server node1.itcast.cn:9092  淘汰：  bin/kafka-topics.sh --list --zookeeper node1.itcast.cn:2181 |

查看Topic的描述信息

|  |
| --- |
| 建议：  kafka-topics.sh --describe --bootstrap-server node1.itcast.cn:9092 --topic test  淘汰版本：  kafka-topics.sh --describe --zookeeper node1.itcast.cn:2181 --topic test |



结果说明：

​ 这是输出的解释。第一行给出了所有分区的摘要，每个附加行提供有关一个分区的信息。由于我们只有一个分区用于此主题，因此只有一行。

​ “leader”是负责给定分区的所有读取和写入的节点。每个节点将成为随机选择的分区部分的领导者。（因为在kafka中 如果有多个副本的话，就会存在leader和follower的关系，表示当前这个副本为leader所在的broker是哪一个）

​ “replicas”是复制此分区日志的节点列表，无论它们是否为领导者，或者即使它们当前处于活动状态。（所有副本列表 0 ，1,2）

​ “isr”是“同步”复制品的集合。这是副本列表的子集，该列表当前处于活跃状态并且已经被领导者捕获。（可用的列表数）

### 删除topic

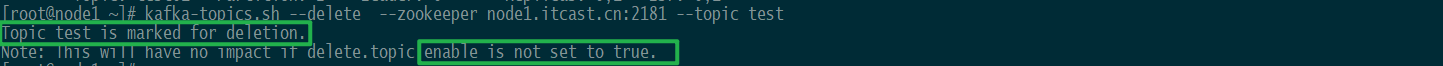
目前删除topic在默认情况下只是打上一个删除的标记，在重新启动kafka后才删除。如果需要立即删除，则需要在

server.properties中配置：

delete.topic.enable=true

然后执行以下命令进行删除topic

|  |
| --- |
| bin/kafka-topics.sh --delete --zookeeper node1.itcast.cn:2181 --topic test |

执行结果截图如下：

### 生产消息到Kafka

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 node1.itcast.cn:9092 --topic test |

### 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 --bootstrap-server node1.itcast.cn:9092 --topic test --from-beginning |

### 扩展其他命令

增加topic分区数

任意kafka服务器执行以下命令可以增加topic分区数

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

增加配置

动态修改kakfa的配置

bin/kafka-topics.sh --zookeeper node1.itcast.cn:2181 --alter --topic test --config flush.messages=1

删除配置

动态删除kafka集群配置

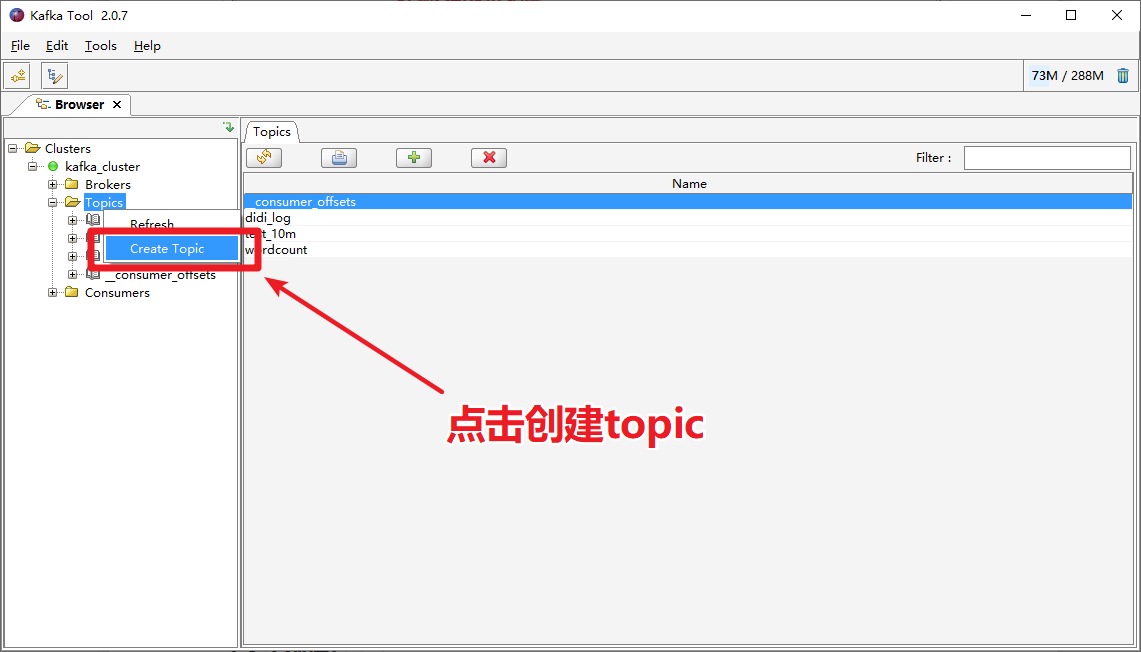
bin/kafka-topics.sh --zookeeper node1.itcast.cn:2181 --alter --topic test --delete-config flush.messages

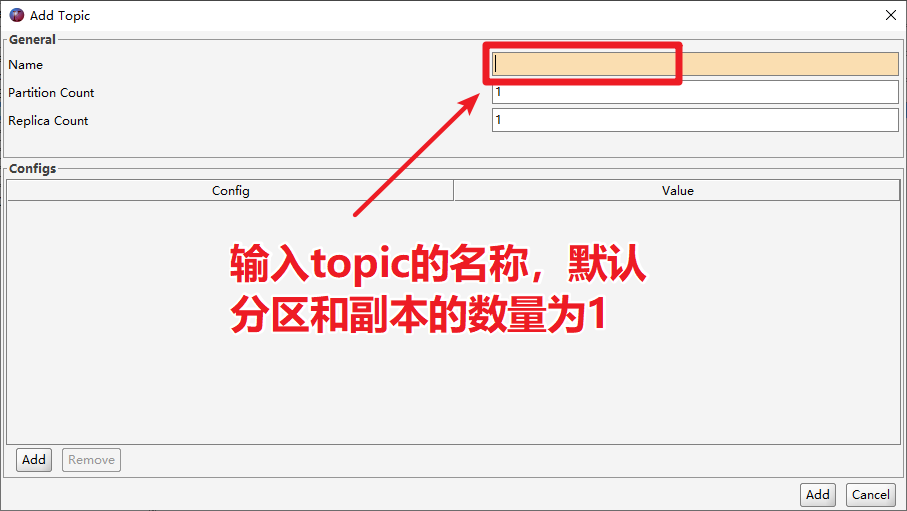
### 使用Kafka Tools操作Kafka

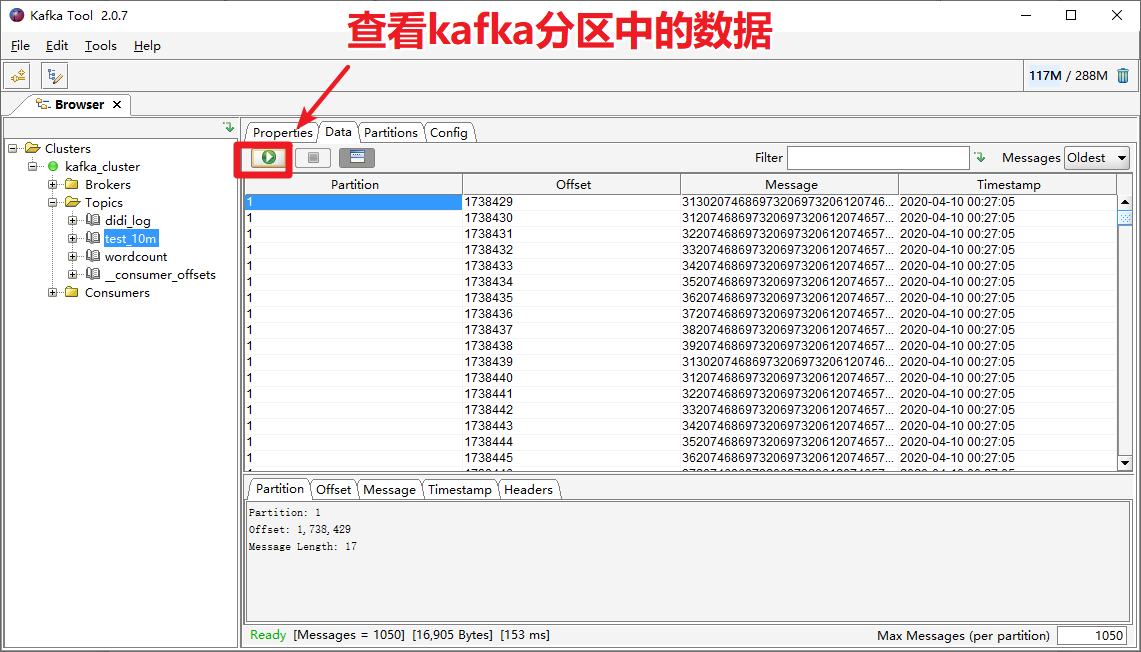
#### 连接Kafka集群

|  |
| --- |
| 安装Kafka Tools后启动Kafka |
|  |
|  |
|  |

#### 创建topic







## Kafka基准测试

### 基准测试

基准[测试](http://www.blogjava.net/qileilove/archive/2012/07/05/382241.html)（benchmark testing）是一种测量和评估软件性能指标的活动。我们可以通过基准测试，了解到软件、硬件的性能水平。主要测试**负载的执行时间、传输速度、吞吐量、资源占用率**等。

#### 基于1个分区1个副本的基准测试

测试步骤：

1. 启动Kafka集群
2. 创建一个1个分区1个副本的topic: benchmark
3. 同时运行生产者、消费者基准测试程序
4. 观察结果

##### 创建topic

|  |
| --- |
| bin/kafka-topics.sh --zookeeper node1.itcast.cn:2181 --create --topic benchmark --partitions 1 --replication-factor 1 |

##### 生产消息基准测试

在生产环境中，推荐使用生产5000W消息，这样会性能数据会更准确些。为了方便测试，课程上演示测试500W的消息作为基准测试。

|  |
| --- |
| 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=1 |

|  |
| --- |
| bin/kafka-producer-perf-test.sh  --topic topic的名字  --num-records 总共指定生产数据量（默认5000W）  --throughput 指定吞吐量——限流（-1不指定）  --record-size record数据大小（字节）  --producer-props bootstrap.servers=192.168.1.20:9092,192.168.1.21:9092,192.168.1.22:9092 acks=1 指定Kafka集群地址，ACK模式 |

测试结果：

|  |  |
| --- | --- |
| 吞吐量 | 93092.533979 records/sec  **每秒9.3W条记录** |
| 吞吐速率 | (88.78 MB/sec)  每秒约89MB数据 |
| 平均延迟时间 | 346.62 ms avg latency |
| 最大延迟时间 | 1003.00 ms max latency |

##### 消费消息基准测试

|  |
| --- |
| bin/kafka-consumer-perf-test.sh --broker-list node1.itcast.cn:9092,node2.itcast.cn:9092,node3.itcast.cn:9092 --topic benchmark --fetch-size 1048576 --messages 5000000 |

|  |
| --- |
| bin/kafka-consumer-perf-test.sh  --broker-list 指定kafka集群地址  --topic 指定topic的名称  --fetch-size 每次拉取的数据大小  --messages 总共要消费的消息个数 |

|  |  |
| --- | --- |
| data.consumed.in.MB  共计消费的数据 | 4768.3716MB |
| MB.sec  每秒消费的数量 | 445.6006  每秒445MB |
| data.consumed.in.nMsg  共计消费的数量 | 5000000 |
| nMsg.sec  每秒的数量 | 467246.0518  每秒46.7W条 |

#### 基于3个分区1个副本的基准测试

被测虚拟机：

|  |  |  |
| --- | --- | --- |
| node1.itcast.cn | node2.itcast.cn | node3.itcast.cn |
| inter i5 8th 8G内存 | inter i5 8th 4G内存 | inter i5 8th 4G内存 |

##### 创建topic

|  |
| --- |
| bin/kafka-topics.sh --zookeeper node1.itcast.cn:2181 --create --topic benchmark --partitions 3 --replication-factor 1  bin/kafka-topics.sh --alter --bootstrap-server node1.itcast.cn:9092 --topic benchmark --partitions 3 |

##### 生产消息基准测试

|  |
| --- |
| 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=1 |

测试结果：

|  |  |  |
| --- | --- | --- |
| **指标** | **3分区1个副本** | **单分区单副本** |
| 吞吐量 | 68755.930199 records/sec | 93092.533979 records/sec  每秒9.3W条记录 |
| 吞吐速率 | 65.57 MB/sec | (88.78 MB/sec)  每秒约89MB数据 |
| 平均延迟时间 | 469.37 ms avg latency | 346.62 ms avg latency |
| 最大延迟时间 | 2274.00 ms max latency | 1003.00 ms max latency |

在虚拟机上，因为都是共享笔记本上的CPU、内存、网络，所以分区越多，反而效率越低。但如果是真实的服务器，分区多效率是会有明显提升的。

##### 消费消息基准测试

|  |
| --- |
| bin/kafka-consumer-perf-test.sh --broker-list node1.itcast.cn:9092,node2.itcast.cn:9092,node3.itcast.cn:9092 --topic benchmark --fetch-size 1048576 --messages 5000000 |

|  |  |  |
| --- | --- | --- |
| **指标** | **单分区3个副本** | **单分区单副本** |
| data.consumed.in.MB  共计消费的数据 | 4768.3716MB | 4768.3716MB |
| MB.sec  每秒消费的数量 | 265.8844MB | 445.6006  每秒445MB |
| data.consumed.in.nMsg  共计消费的数量 | 5000000 | 5000000 |
| nMsg.sec  每秒的数量 | 278800.0446  每秒27.8W | 467246.0518  每秒46.7W |

还是一样，因为虚拟机的原因，多个分区反而消费的效率也有所下降。

#### 基于1个分区3个副本的基准测试

##### 创建topic

|  |
| --- |
| bin/kafka-topics.sh --zookeeper node1.itcast.cn:2181 --create --topic benchmark --partitions 1 --replication-factor 3 |

##### 生产消息基准测试

|  |
| --- |
| 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=1 |

测试结果：

|  |  |  |
| --- | --- | --- |
| **指标** | **单分区3个副本** | **单分区单副本** |
| 吞吐量 | 29899.477955 records/sec | 93092.533979 records/sec  每秒9.3W条记录 |
| 吞吐速率 | 28.51 MB/sec | (88.78 MB/sec)  每秒约89MB数据 |
| 平均延迟时间 | 1088.43 ms avg latency | 346.62 ms avg latency |
| 最大延迟时间 | 2416.00 ms max latency | 1003.00 ms max latency |

同样的配置，副本越多速度越慢。

##### 消费消息基准测试

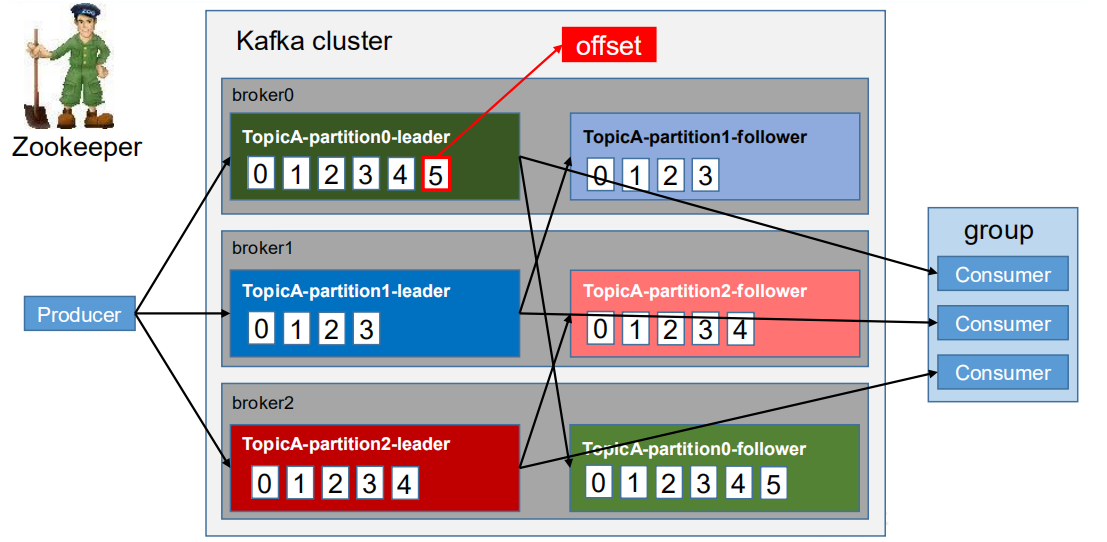
|  |
| --- |
| bin/kafka-consumer-perf-test.sh --broker-list node1.itcast.cn:9092,node2.itcast.cn:9092,node3.itcast.cn:9092 --topic benchmark --fetch-size 1048576 --messages 5000000 |

|  |  |  |
| --- | --- | --- |
| **指标** | **单分区3个副本** | **单分区单副本** |
| data.consumed.in.MB  共计消费的数据 | 4768.3716MB | 4768.3716MB |
| MB.sec  每秒消费的数量 | 265.8844MB  每秒265MB | 445.6006  每秒445MB |
| data.consumed.in.nMsg  共计消费的数量 | 5000000 | 5000000 |
| nMsg.sec  每秒的数量 | 278800.0446  每秒27.8W | 467246.0518  每秒46.7W |

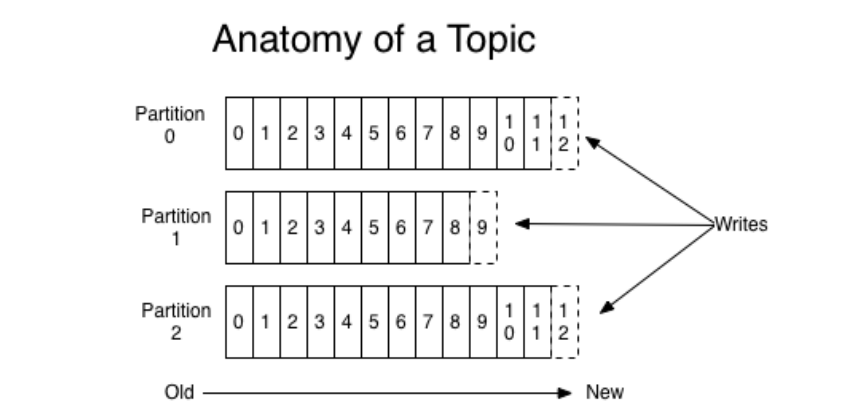
## Kafka的架构分析

### Kafka文件存储机制

Topic---Partition—Segment—log+index

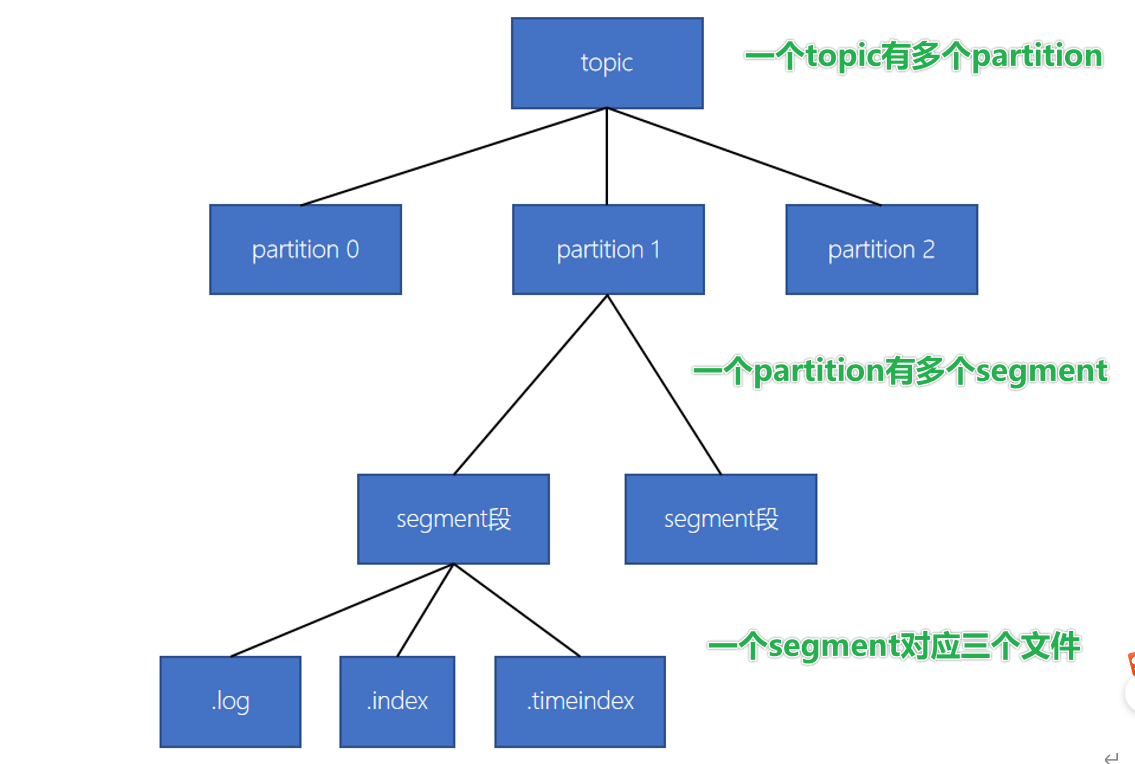


Topic：消息是以 **topic** 进行分类的，生产者生产消息，消费者消费消息，都是面向 topic，如下。



**Topic 是逻辑上的概念，而 partition 是物理上的概念.**

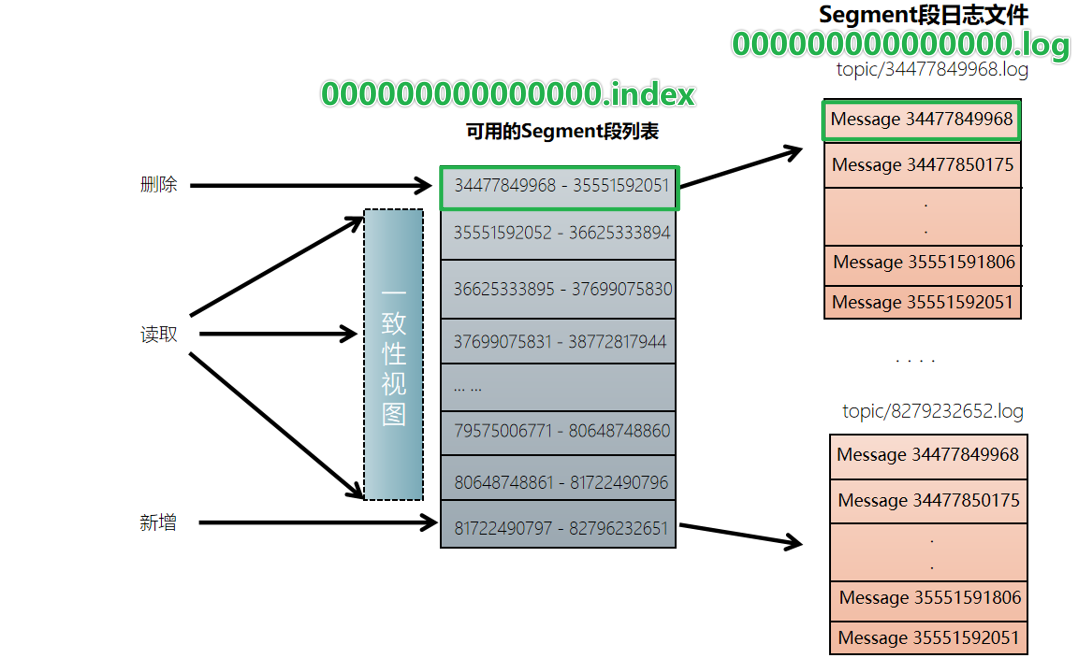
每个 partition 对应于一个 log 文件，该 log 文件中存储的就是 producer 生产的数据。**Producer 生产的数据会被不断追加到该log 文件末端，且每条数据都有自己的 offset**。消费者组中的每个消费者，都会实时记录自己消费到了哪个 offset，以便出错恢复时，从上次的位置继续消费。

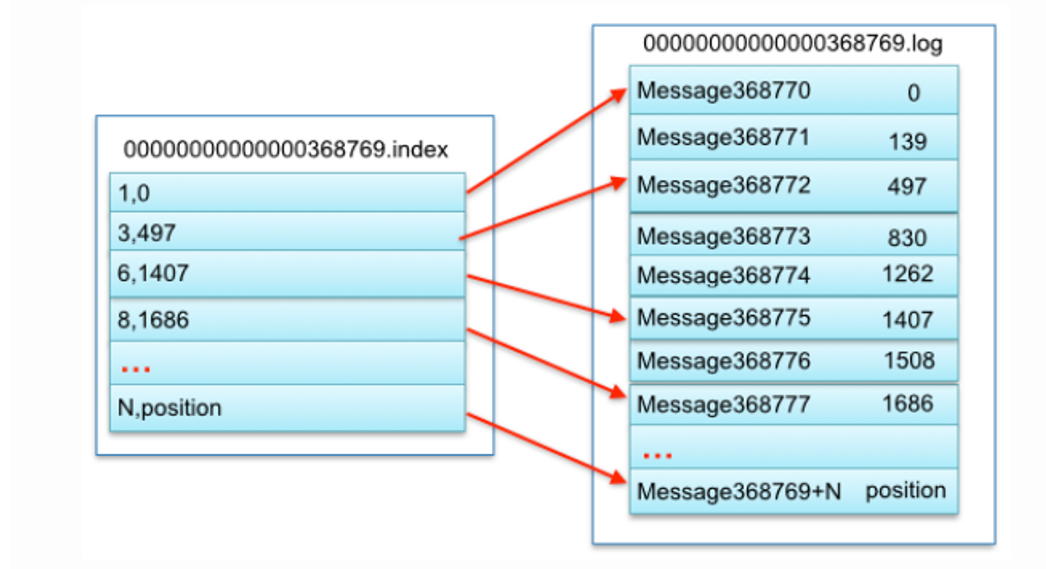


Kafka 采取了**分片和索引**机制，将每个partition 分为多个 segment。每个 segment对应两个文件— “.index”文件和“.log”文件。这些文件位于一个文件夹下，该文件夹的命名

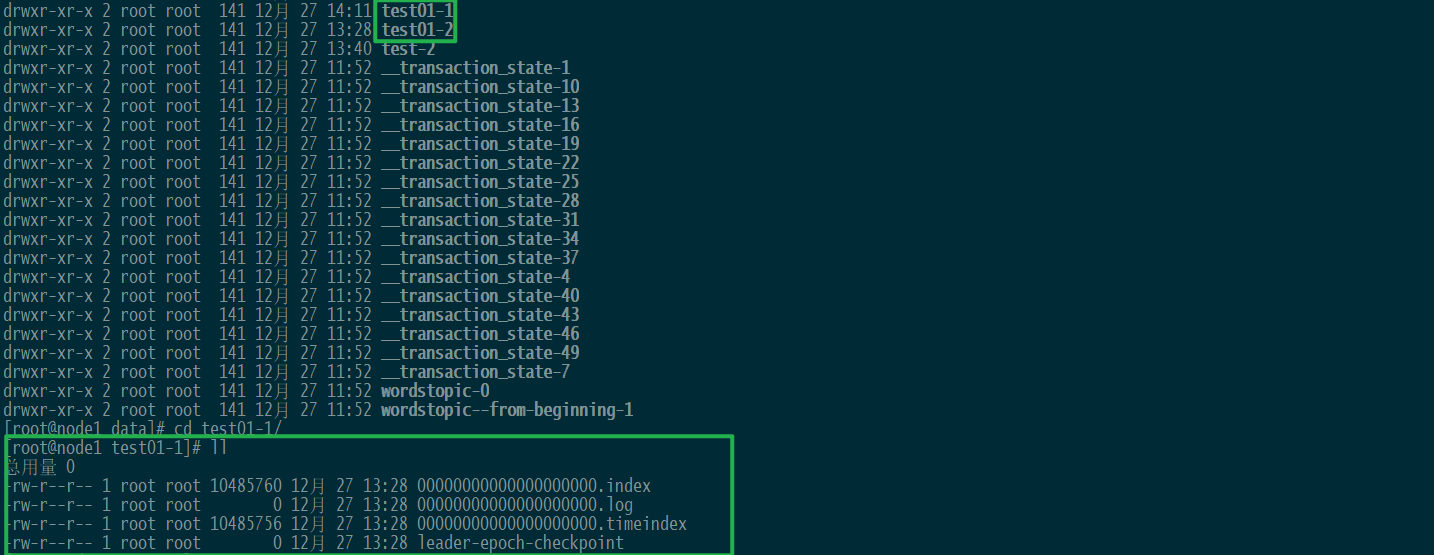
规则为：topic 名称+分区序号。例如，test这个topic有三个分区，则其对应的文件夹为test-0, test-1, test-2。

如下图：**“.index”文件存储大量的索引信息，“.log”文件存储大量的数据**，索引文件中的元数据指向对应数据文件中 message 的物理偏移地址。**Timemindex根据时间索引判断哪些segment数据，可以定期进行数据清理(TTL)**





根据上面的segment的索引段找到offset起始offset和endoffset在寻找对应message



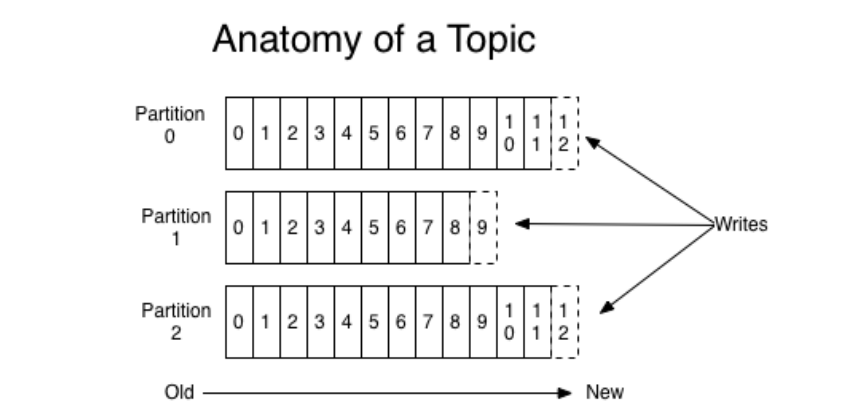
### Kafka生产者

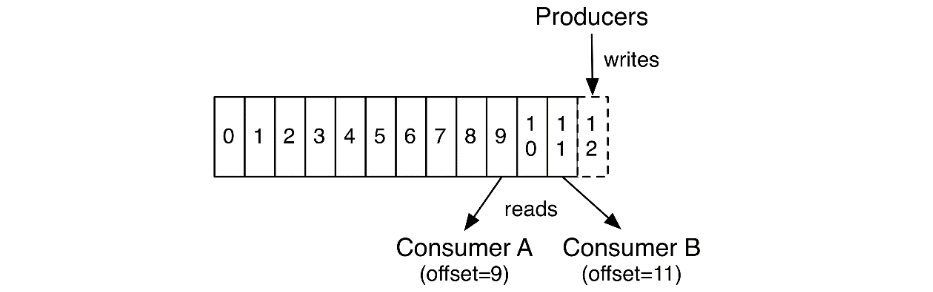
#### 写入方式

producer采用**推（push）模式**将消息发布到broker，**每条消息都被追加（append）到分区（patition）中，属于顺序写磁盘**（顺序写磁盘效率比随机写内存要高，保障kafka吞吐率）。

#### 分区（Partition）

消息发送时都被发送到一个topic，**其本质就是一个目录**，而topic是由一些Partition Logs(分区日志)组成，其组织结构如下图所示：





我们可以看到，每个Partition中的消息都是**有序**的，生产的消息被不断追加到Partition log上，其中的每一个消息都被赋予了一个**唯一的offset值**。

1）分区的原因

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

（2）可以提高并发，因为可以以**Partition**为单位读写了。

2）分区的原则

（1）指定了patition，则直接使用；

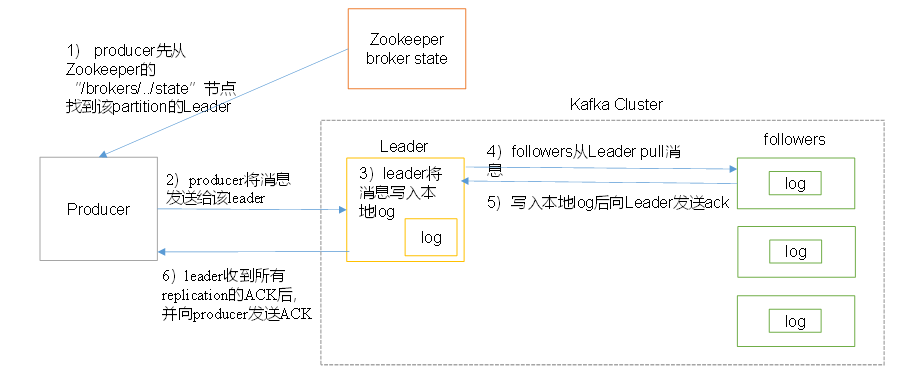
（2）未指定patition但指定key，通过对key的hash值与topic的patition数取余得到partition值；(后面讲解)

（3）patition和key都未指定，使用轮询选出一个patition，第一次随机生成一个整数，该数值在每次调用会自增，将这个值与topic可用的partition总数取余得到partition值，也就是roundrobin算法(后面讲解)

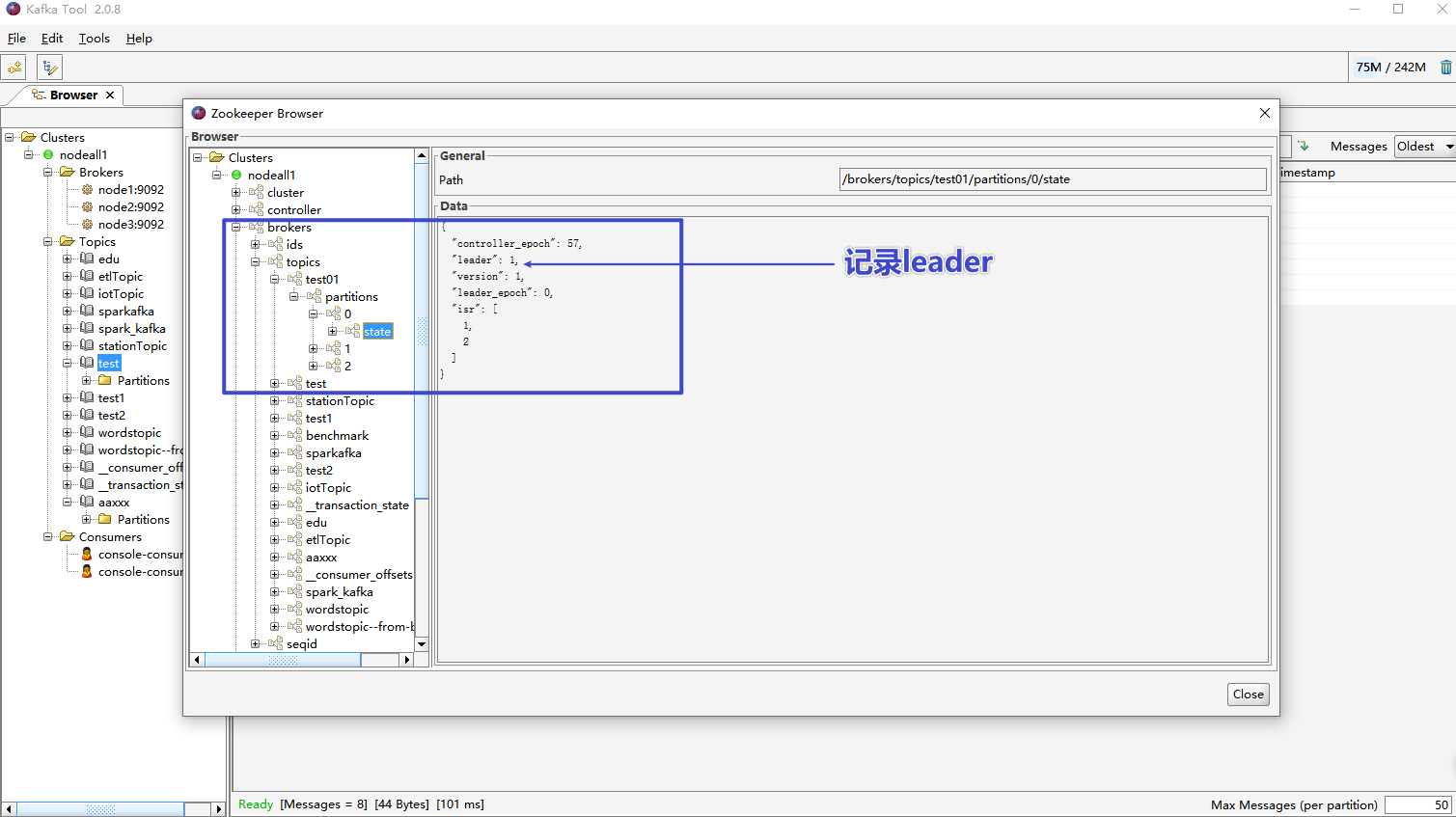
#### 副本（Replication）

同一个partition可能会有多个replication（对应 server.properties 配置中的 default.replication.factor=N）。没有replication的情况下，一旦broker 宕机，其上所有 patition 的数据都不可被消费，同时producer也不能再将数据存于其上的patition。引入replication之后，同一个partition可能会有多个replication，**而这时需要在这些replication之间选出一个leader，producer和consumer只与这个leader交互**，其它replication作为follower从leader 中复制数据。

#### 写入流程分析



如下kafkatool记录state



1）producer先从zookeeper的 "/brokers/.../state"节点找到该partition的leader

2）producer将消息发送给该leader

3）leader将消息写入本地log

4）followers从leader pull消息，写入本地log后向leader发送ACK

5）leader收到所有**ISR同步副本中的replication的ACK**后，**增加HW（high watermark**，最后commit 的offset）并向producer发送ACK

#### 数据可靠性保证

上述的案例我们能看到ACK机制，如何理解？

为保证 producer 发送的数据，能可靠的发送到指定的 topic，topic 的每个 partition 收到

producer 发送的数据后，都需要向 producer 发送 ack（acknowledgement 确认收到），如果

producer 收到 ack，就会进行下一轮的发送，否则重新发送数据。

这里需要理解何时发送ack以及**多少个follwers同步完成后发送ack**?

1. 确保有follower与leader同步完成，leader再发送ack，这样才能保证leader挂掉之后，能在follower中选举出新的leader.
2. 两种方案：全部followers同步完成以及板书以上followers同步完成。



Kafka 选择了第二种方案，原因如下：

1.同样为了容忍 n 台节点的故障，第一种方案需要 2n+1 个副本(如果有n个机器挂了，半数以上指的是n+1)，而第二种方案只需要 n+1个副本，而 Kafka 的每个分区都有大量的数据，第一种方案会造成大量数据的冗余。

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

2）ISR

采用第二种方案之后，设想以下情景：**leader 收到数据，所有 follower 都开始同步数据，**

**但有一个 follower，因为某种故障，迟迟不能与 leader 进行同步，那 leader 就要一直等下去，**

**直到它完成同步，才能发送 ack。这个问题怎么解决呢？**

答案：Leader 维护了一个动态的 in-sync replica set (ISR)，意为和 leader 保持同步的 follower 集

合。当 **ISR 中的 follower 完成数据的同步之后，leader 就会给 follower 发送 ack**。如果 follower

长时间 未 向 leader 同 步 数 据 ， 则 该 follower 将 被 踢 出 ISR ， 该 时 间 阈 值 由**replica.lag.time.max.ms(心跳时间，默认10s)** 参数设定。Leader 发生故障之后，就会从 ISR 中选举新的 leader。

3）ack 应答机制

对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失，

所以没必要等 ISR 中的 follower 全部接收成功。

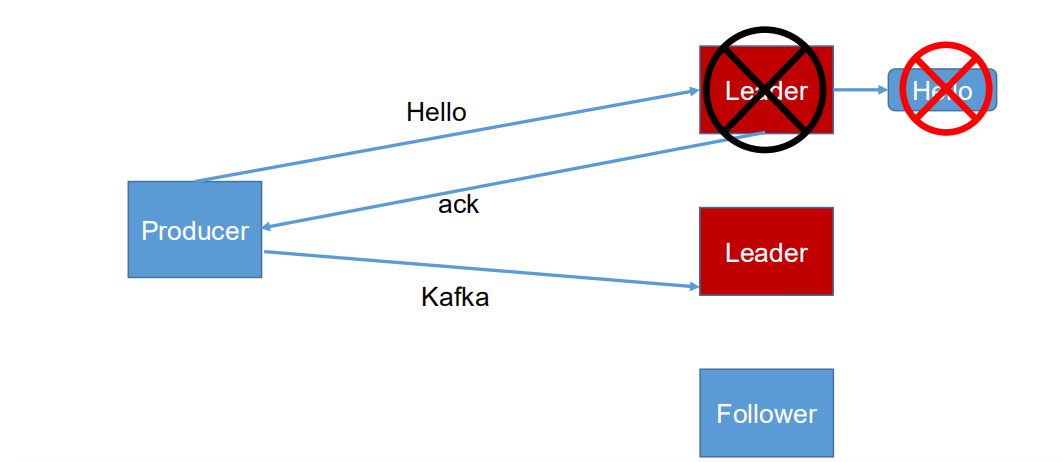
**所以 Kafka 为用户提供了三种可靠性级别，用户根据对可靠性和延迟的要求进行权衡**，

选择以下的配置。

acks 参数配置：

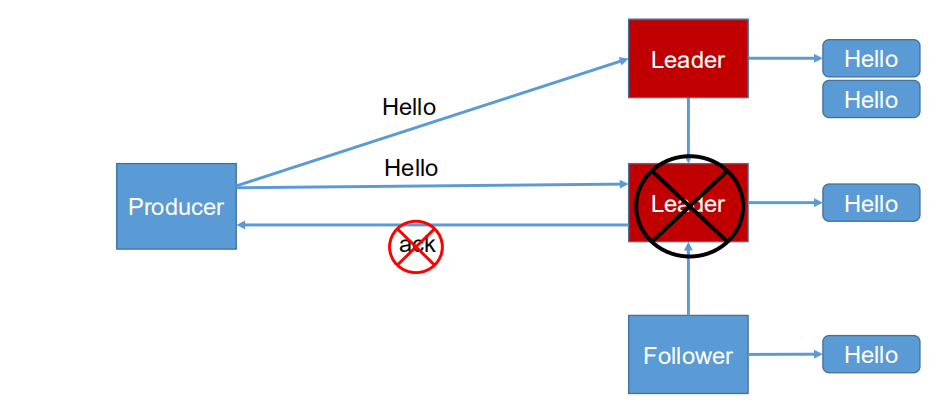
acks： 0：producer 不等待 broker 的 ack，这一操作提供了一个最低的延迟，broker 一接收到还没有写入磁盘就已经返回，当 broker 故障时有可能丢失数据；

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



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

造成数据重复。



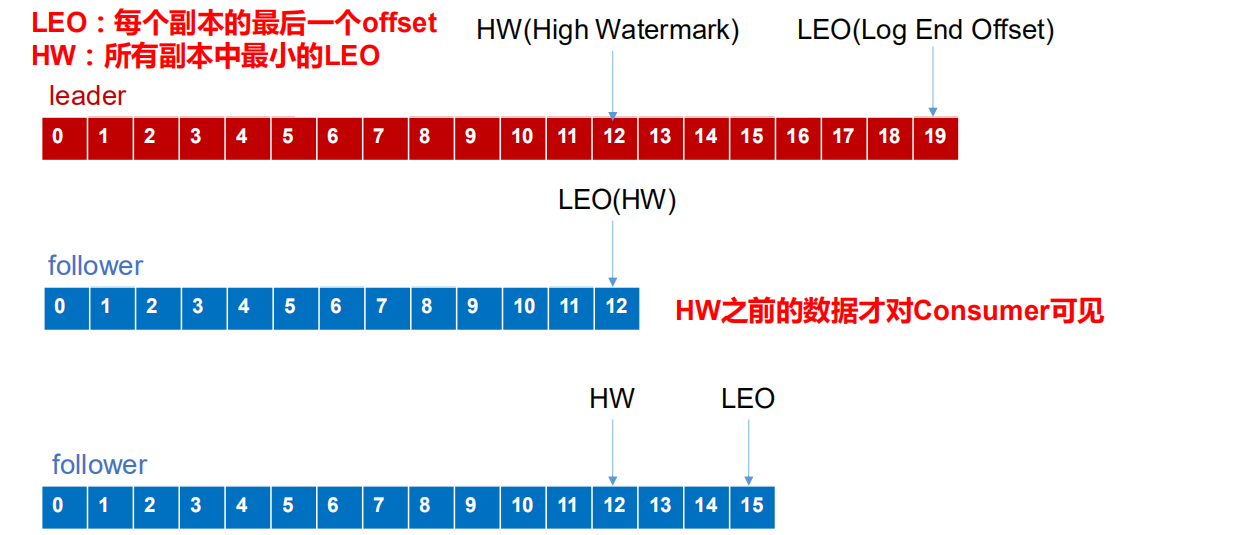
#### 故障处理细节

LEO：指的是每个副本最大的 offset；**(消费者课件的最大offset，保证消费者消费一致性)**

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

思考：假设没有HW，会出现什么情况？

如果没有HW，消费者在leader获取数据可以获取offset为19的数据，如果此时leader挂了，消费者是可以获取16-19之间的数据的，但是下面两个follower都没有16-19的offset的值，所以这里设置HW消费者端可以看到HW=12，能够达到无论谁挂掉获取Offset偏移量都统一。



（1）follower 故障

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

（2）leader 故障

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

同步数据。

注意：这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。

#### Exactly-Once语义

将服务器的ACK级别设置为-1，可以保证Producer到Server之间的不会丢失数据，也就是At LeasetOnce，相对的，如果将服务器ACK级别设置为0，可以保证生产者每条数据只会被发送一次，会出现丢失，就是AtMostOnce。0.11版本kafka引入幂等性。无论Producer向Server发送多少重复数据，Server都只会持久化一条数据。幂等性结合AtLeastOnce实现精确一致。

AtLeastOnce+幂等性=ExactlyOnce

开启幂等性，只需要将Producer的参数中ennable.idompotence设置为true即可。Kafka的幂等性就是将原来下游的需要做的去重放到了数据上游，开启幂等性的Producer在初始化的时候会分配一个PID，发往同一个Partition的消息会附带SequenceNumber。而Broker端会对<PID, Partition,SequenceNumber>做缓存，当具有相同主键提交的时候，Broker会持久化一条数据。

### Broker保存消息

存储方式

物理上把topic分成一个或多个patition（对应 server.properties 中的num.partitions=3配置），每个patition物理上对应一个文件夹（该文件夹存储该patition的所有消息和索引文件），如下：

存储策略

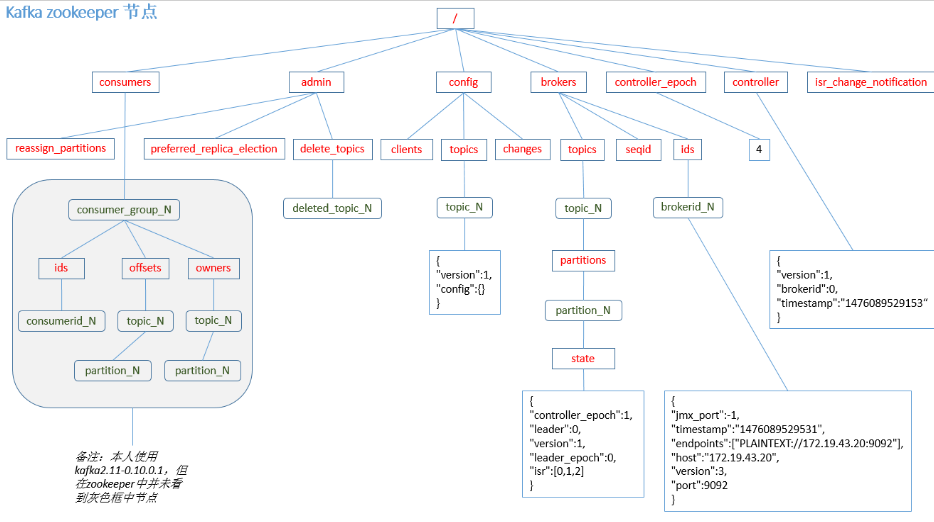
无论消息是否被消费，kafka都会保留所有消息。有两种策略可以删除旧数据：

1）基于时间：log.retention.hours=168

2）基于大小：log.retention.bytes=1073741824

需要注意的是，因为Kafka读取特定消息的时间复杂度为O(1)，即与文件大小无关，所以这里删除过期文件与提高 Kafka 性能无关。

Zk存储结构

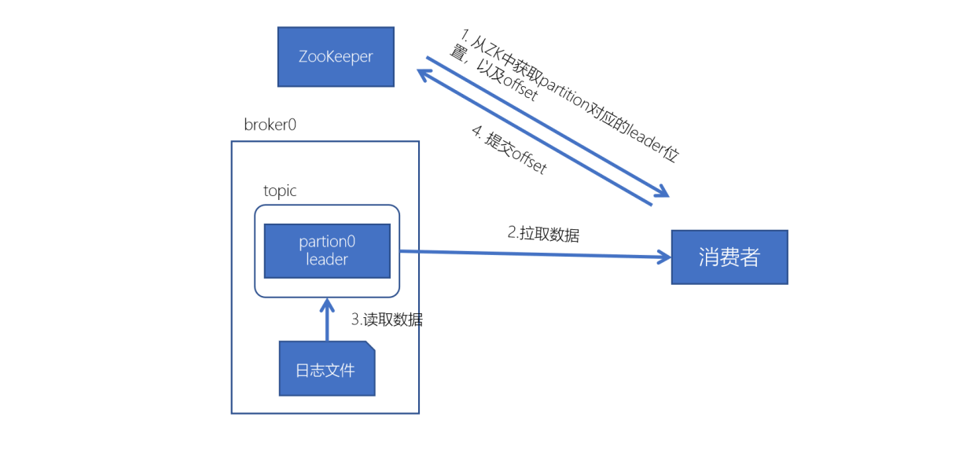


注意：producer不在zk中注册，消费者在zk中注册。

注意：在Kafka 0.9 版本之前，consumer 默认将 offset 保存在 Zookeeper 中，从 0.9 版本开始，consumer 默认将 offset 保存在 Kafka 一个内置的 topic 中，该 topic 为\_\_consumer\_offsets。

### Kafka消费者

#### 消费的流程分析



* 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

#### API介绍

kafka提供了两套consumer API：高级Consumer API和低级Consumer API。

高级API

1）高级API优点

高级API 写起来简单

不需要自行去管理offset，系统通过zookeeper自行管理。

不需要管理分区，副本等情况，.系统自动管理。

消费者断线会自动根据上一次记录在zookeeper中的offset去接着获取数据（**默认设置1分钟更新一下zookeeper中存的offset）**

可以使用**group来区分对同一个topic 的不同程序访问分离**开来（不同的group记录不同的offset，这样不同程序读取同一个topic才不会因为offset互相影响）

2）高级API缺点

不能自行控制offset（对于某些特殊需求来说）

不能细化控制如分区、副本、zk等

低级API

1）低级 API 优点

能够让开发者自己控制offset，想从哪里读取就从哪里读取。

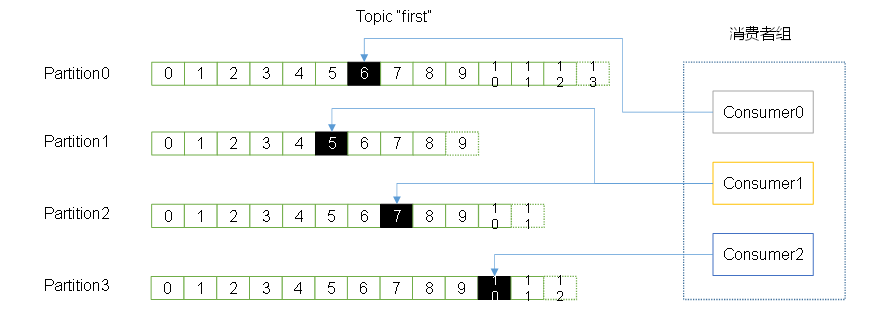
自行控制连接分区，对分区自定义进行负载均衡

对zookeeper的依赖性降低（如：offset不一定非要靠zk存储，自行存储offset即可，比如存在文件或者内存中）

2）低级API缺点

太过复杂，需要自行控制offset，连接哪个分区，找到分区leader 等。

#### 消费者组



消费者是以consumer group消费者组的方式工作，由一个或者多个消费者组成一个组，共同消费一个topic。每个分区在同一时间只能由group中的一个消费者读取，但是多个group可以同时消费这个partition。在图中，**有一个由三个消费者组成的group**，有一个消费者读取主题中的两个分区，另外两个分别读取一个分区。某个消费者读取某个分区，也可以叫做某个消费者是某个分区的拥有者。

在这种情况下，消费者可以通过水平扩展的方式同时读取大量的消息。另外，如果一个消费者失败了，那么其他的group成员会自动负载均衡读取之前失败的消费者读取的分区。

#### 消费方式

consumer采用pull（拉）模式从broker中读取数据。

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

对于Kafka而言，pull模式更合适，它可简化broker的设计，consumer可自主控制消费消息的速率，同时consumer可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

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，一直等待数据到达。为了避免这种情况，我们在我们的拉请求中有参数，允许消费者请求在等待数据到达的“长轮询”中进行阻塞（并且可选地等待到给定的字节数，以确保大的传输大小）。

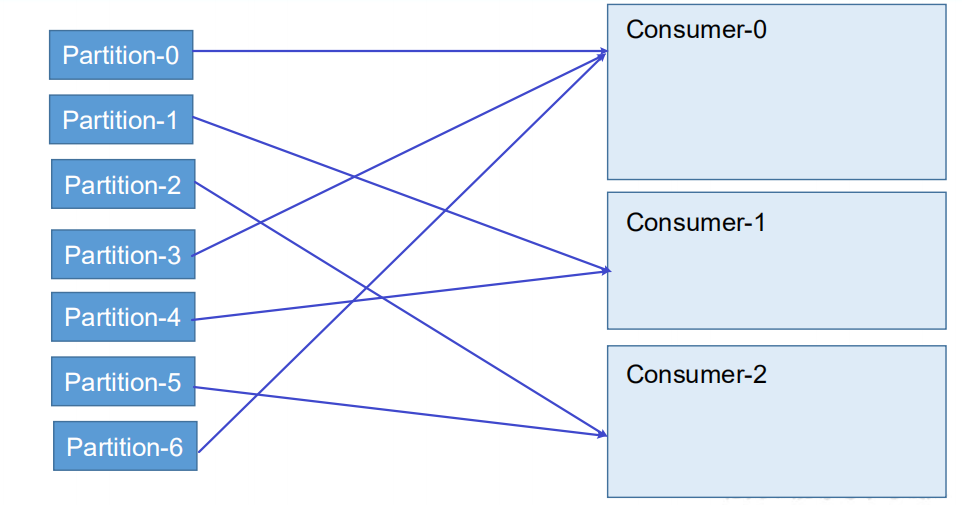
#### 分区策略

一个 consumer group 中有多个 consumer，一个 topic 有多个 partition，所以必然会涉及

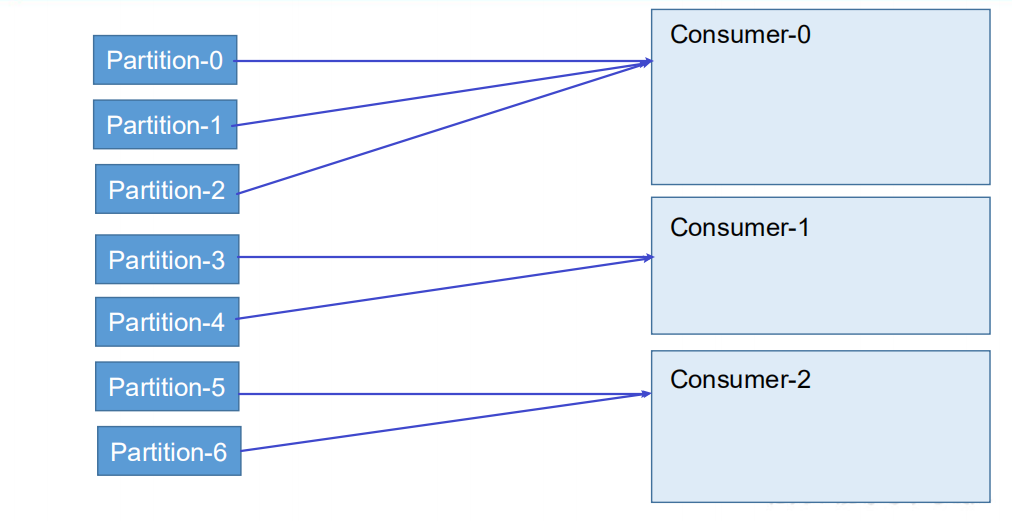
到 partition 的分配问题，即确定那个 partition 由哪个 consumer 来消费。

Kafka 有两种分配策略，一是 RoundRobin，一是 Range。

RoundRobin：



Range：



消费者组案例后置：

需求：测试同一个消费者组中的消费者，同一时刻只能有一个消费者消费。

### Kafka高效读写

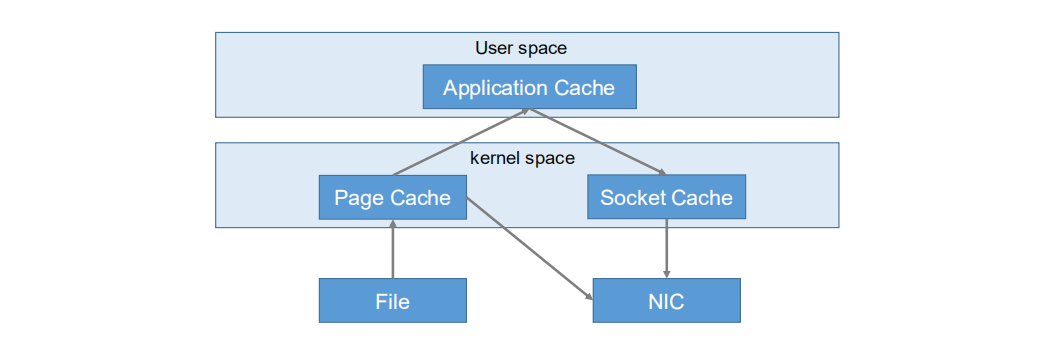
1）顺序写磁盘

Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端，

为顺序写。官网有数据表明，同样的磁盘，顺序写能到 600M/s，而随机写只有 100K/s。这

与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

2）零复制技术



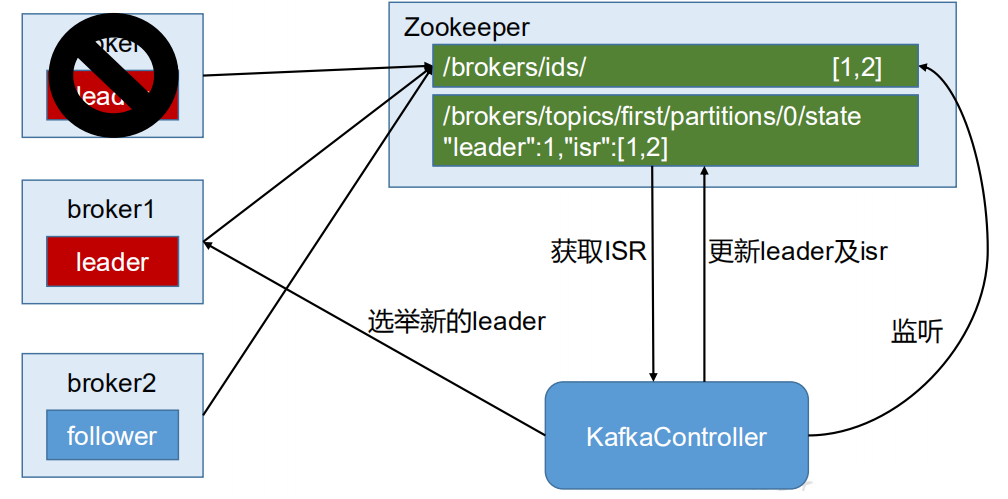
ZK在kafka中作用

Kafka 集群中有一个 broker 会被选举为 Controller，负责管理集群 broker 的上下线，所

有 topic 的分区副本分配和 leader 选举等工作。

Controller 的管理工作都是依赖于 Zookeeper 的。

以下为 partition 的 leader 选举过程



## Java编程操作Kafka

### 同步生产消息到Kafka中

#### 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。

参考官网API：<http://kafka.apache.org/21/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html>

#### 准备工作

##### 导入Maven Kafka POM依赖

|  |
| --- |
| <**repositories**><!-- 代码库 -->  <**repository**>  <**id**>central</**id**>  <**url**>http://maven.aliyun.com/nexus/content/groups/public//</**url**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  <**snapshots**>  <**enabled**>true</**enabled**>  <**updatePolicy**>always</**updatePolicy**>  <**checksumPolicy**>fail</**checksumPolicy**>  </**snapshots**>  </**repository**> </**repositories**>  <**dependencies**>  <!-- kafka客户端工具 -->  <**dependency**>  <**groupId**>org.apache.kafka</**groupId**>  <**artifactId**>kafka-clients</**artifactId**>  <**version**>2.4.1</**version**>  </**dependency**>   <!-- 工具类 -->  <**dependency**>  <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-io</**artifactId**>  <**version**>1.3.2</**version**>  </**dependency**>   <!-- SLF桥接LOG4J日志 -->  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-log4j12</**artifactId**>  <**version**>1.7.6</**version**>  </**dependency**>   <!-- SLOG4J日志 -->  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.16</**version**>  </**dependency**> </**dependencies**>  <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.1</**version**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  </**configuration**>  </**plugin**>  </**plugins**> </**build**> |

##### 导入log4j.properties

将log4j.properties配置文件放入到resources文件夹中

|  |
| --- |
| **log4j.rootLogger**=**INFO,stdout log4j.appender.stdout**=**org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout**=**org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern**=**%5p - %m%n** |

##### 创建包和类

创建包cn.itcast.kafka，并创建KafkaProducerTest类。

#### 代码开发

可以参考以下方式来编写第一个Kafka示例程序

参考以下文档：<http://kafka.apache.org/24/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html>

1. 创建用于连接Kafka的Properties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"); |

1. 创建一个生产者对象KafkaProducer
2. 调用send发送1-100消息到指定Topic test，并获取返回值Future，该对象封装了返回值
3. 再调用一个Future.get()方法等待响应
4. 关闭生产者

参考代码：

|  |
| --- |
| **public class** KafkaProducerTest {  **public static void** main(String[] args) {  // 1. 创建用于连接Kafka的Properties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. 创建一个生产者对象KafkaProducer  KafkaProducer<String, String> producer = **new** KafkaProducer<String, String>(props);   // 3. 调用send发送1-100消息到指定Topic test  **for**(**int** i = 0; i < 100; ++i) {  **try** {  // 获取返回值Future，该对象封装了返回值  Future<RecordMetadata> future = producer.send(**new** ProducerRecord<String, String>(**"test"**, **null**, i + **""**));  // 调用一个Future.get()方法等待响应  future.get();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (ExecutionException e) {  e.printStackTrace();  }  }   // 5. 关闭生产者  producer.close();  } } |

### 从Kafka的topic中消费消息

#### 需求

从 test topic中，将消息都消费，并将记录的offset、key、value打印出来

#### 准备工作

在cn.itcast.kafka包下创建KafkaConsumerTest类

#### 开发步骤

1. 创建Kafka消费者配置

|  |
| --- |
| Properties props = **new** Properties(); props.setProperty(**"bootstrap.servers"**, **"node1.itcast.cn: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"**); |

1. 创建Kafka消费者
2. 订阅要消费的主题
3. 使用一个while循环，不断从Kafka的topic中拉取消息
4. 将将记录（record）的offset、key、value都打印出来

#### 参考代码

|  |
| --- |
| **public class** KafkaProducerTest {  **public static void** main(String[] args) {  // 1. 创建用于连接Kafka的Properties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"**);   // 2. 创建一个生产者对象KafkaProducer  KafkaProducer<String, String> producer = **new** KafkaProducer<String, String>(props);   // 3. 调用send发送1-100消息到指定Topic test  **for**(**int** i = 0; i < 100; ++i) {  **try** {  // 获取返回值Future，该对象封装了返回值  Future<RecordMetadata> future = producer.send(**new** ProducerRecord<String, String>(**"test"**, **null**, i + **""**));  // 调用一个Future.get()方法等待响应  future.get();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (ExecutionException e) {  e.printStackTrace();  }  }   // 5. 关闭生产者  producer.close();  } } |

参考官网API文档：

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

### 异步使用带有回调函数方法生产消息

**如果我们想获取生产者消息是否成功，或者成功生产消息到Kafka中后，执行一些其他动作**。此时，可以很方便地使用带有回调函数来发送消息。

需求：

1. **在发送消息出现异常时，能够及时打印出异常信息**
2. **在发送消息成功时，打印Kafka的topic名字、分区id、offset**

|  |
| --- |
| **public class** KafkaProducerTest {  **public static void** main(String[] args) {  // 1. 创建用于连接Kafka的Properties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"**);   // 2. 创建一个生产者对象KafkaProducer  KafkaProducer<String, String> **producer** = **new** KafkaProducer<String, String>(props);   // 3. 调用send发送1-100消息到指定Topic test  **for**(**int** i = 0; i < 100; ++i) {  // 一、同步方式  // 获取返回值Future，该对象封装了返回值  // Future<RecordMetadata> future = producer.send(new ProducerRecord<String, String>("test", null, i + ""));  // 调用一个Future.get()方法等待响应  // future.get();   // 二、带回调函数异步方式  producer.send(**new** ProducerRecord<String, String>(**"test"**, **null**, i + **""**), **new** Callback() {  @Override  **public void** onCompletion(RecordMetadata metadata, Exception exception) {  **if**(exception != **null**) {  System.***out***.println(**"发送消息出现异常"**);  }  **else** {  String topic = metadata.topic();  **int** partition = metadata.partition();  **long** offset = metadata.offset();   System.***out***.println(**"发送消息到Kafka中的名字为"** + topic + **"的主题，第"** + partition + **"分区，第"** + offset + **"条数据成功!"**);  }  }  });  }   // 5. 关闭生产者  producer.close();  } } |

## 了解KafkaStreaming

KafkaStreaming是Kafka为了占用消费者端，研发的一套消费者API，实际生产环境中还是使用Spark和Flink

kafka StreamsAPI开发

需求：使用StreamAPI获取test这个topic当中的数据，然后将数据全部转为大写，写入到test2这个topic当中去

实现步骤:

1) 创建一个topic

node01服务器使用以下命令来常见一个topic 名称为test2

2) 开发streamAPI

3) 生产数据

执行以下命令，向test这个topic当中生产数据

4) 消费数据

node2执行一下命令消费test2这个topic当中的数据

public class StreamAPI {

public static void main(String[] args) {

Properties props = new Properties();

props.put(StreamsConfig.APPLICATION\_ID\_CONFIG, "wordcount-application");

props.put(StreamsConfig.BOOTSTRAP\_SERVERS\_CONFIG, "node01:9092");

props.put(StreamsConfig.KEY\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass());

props.put(StreamsConfig.VALUE\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass());

KStreamBuilder builder = new KStreamBuilder();

builder.stream("test").mapValues(line -> line.toString().toUpperCase()).to("test2");

KafkaStreams streams = new KafkaStreams(builder, props);

streams.start();

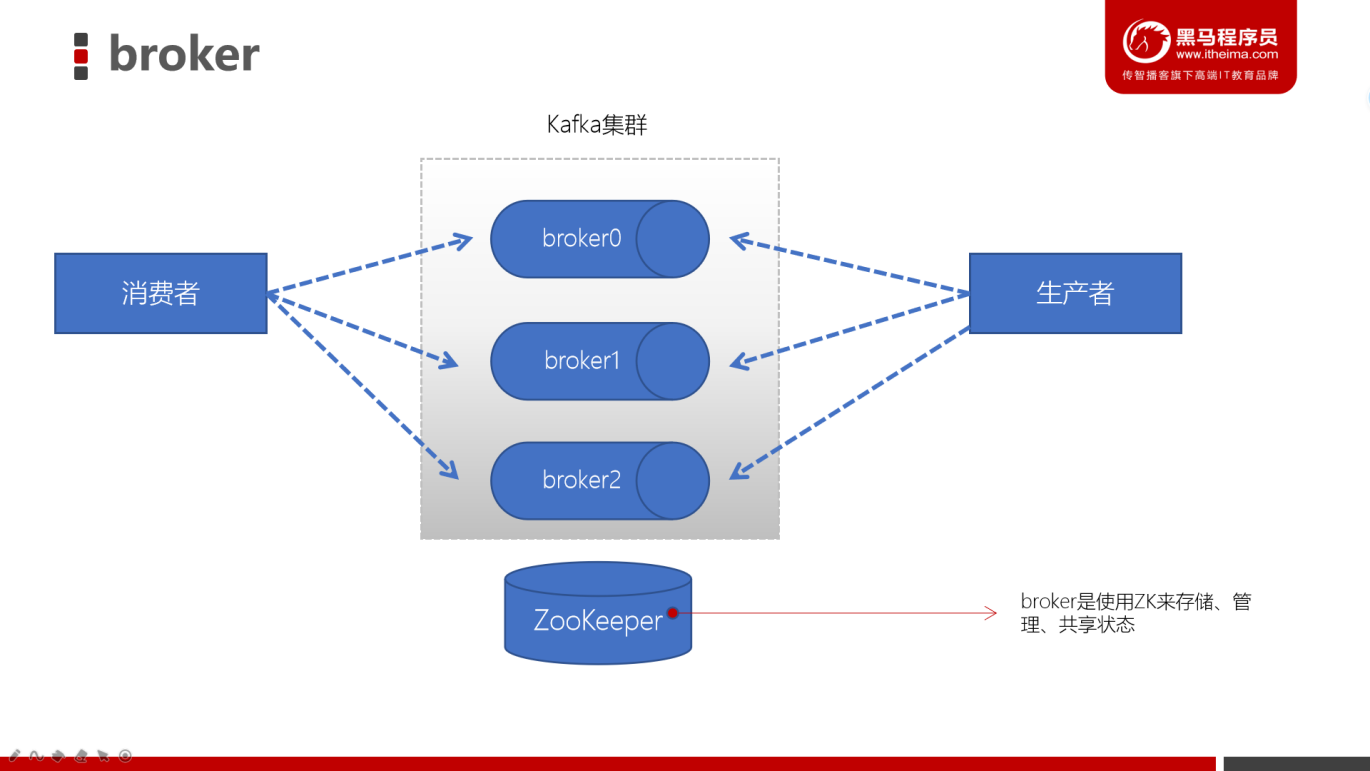
}

}

## 架构

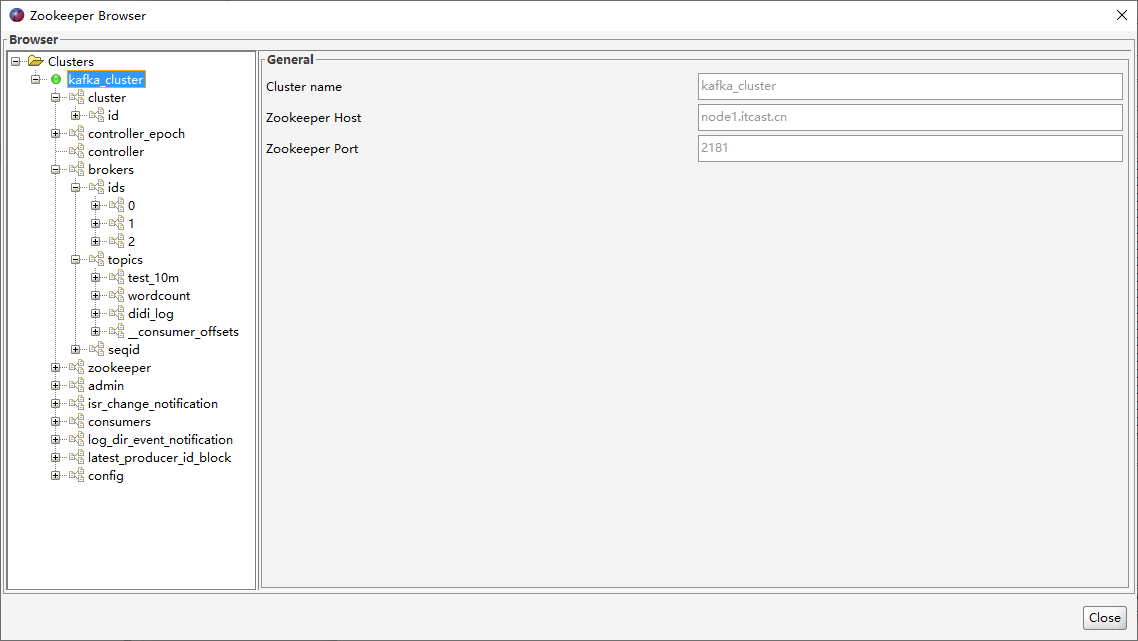
### Kafka重要概念

#### 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（Sateless）**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

#### zookeeper



* ZK用来管理和协调broker，并且存储了Kafka的元数据（例如：有多少topic、partition、consumer）
* 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。

PS：Kafka正在逐步想办法将ZooKeeper剥离，维护两套集群成本较高，社区提出KIP-500就是要替换掉ZooKeeper的依赖。“Kafka on Kafka”——Kafka自己来管理自己的元数据

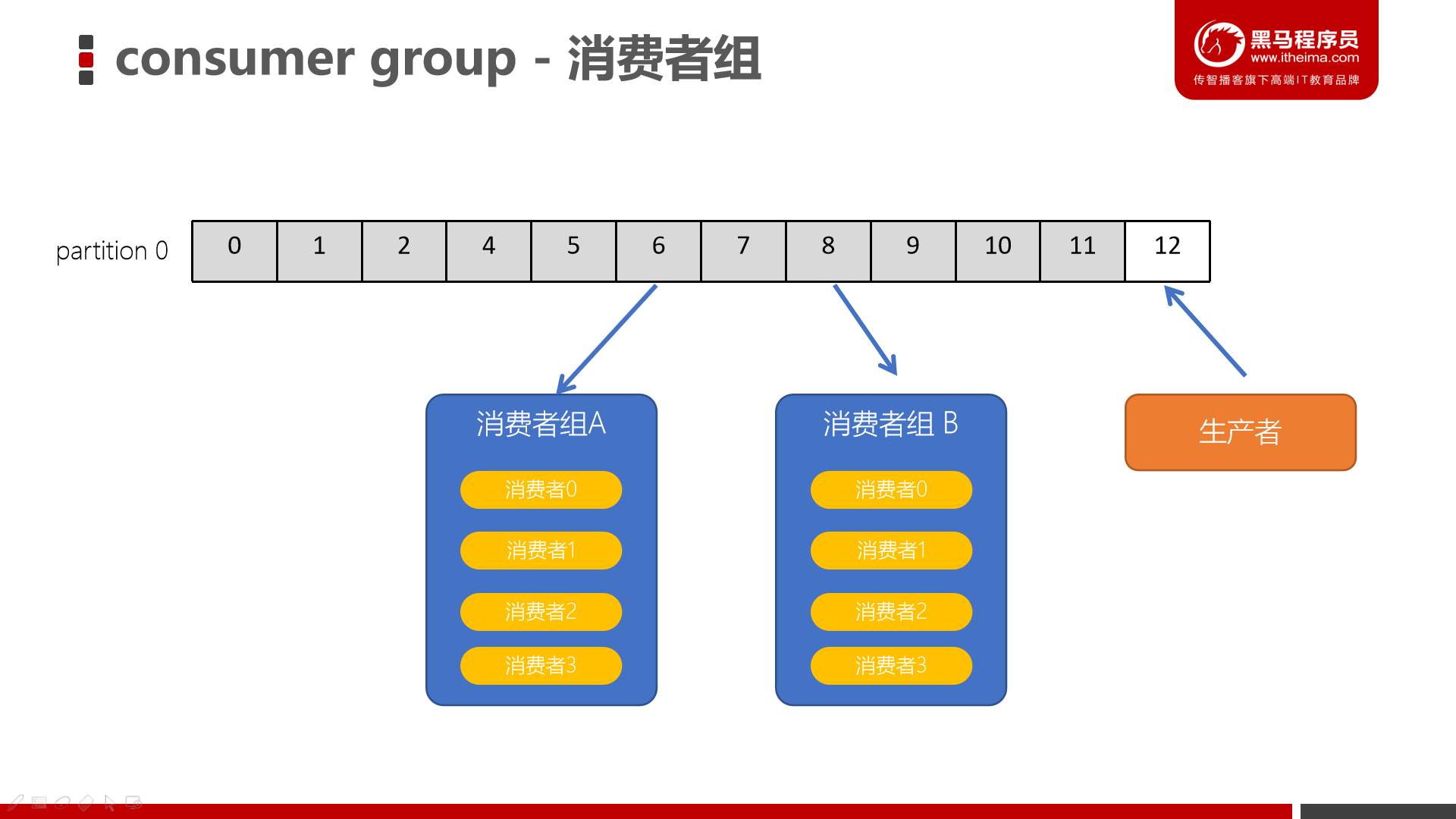
#### producer（生产者）

* 生产者负责将数据推送给broker的topic

#### consumer（消费者）

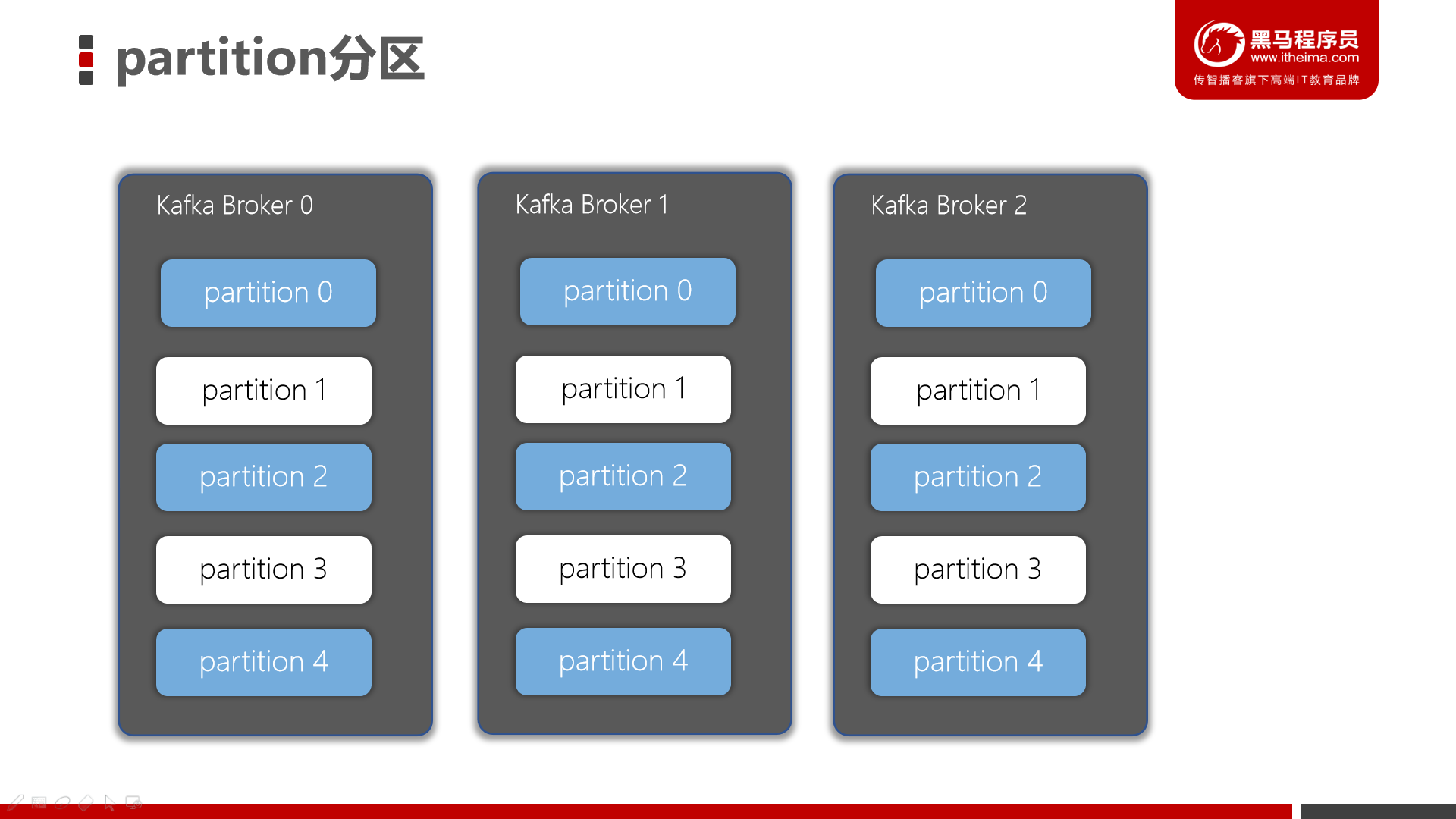
* 消费者负责从broker的topic中拉取数据，并自己进行处理

#### consumer group（消费者组）



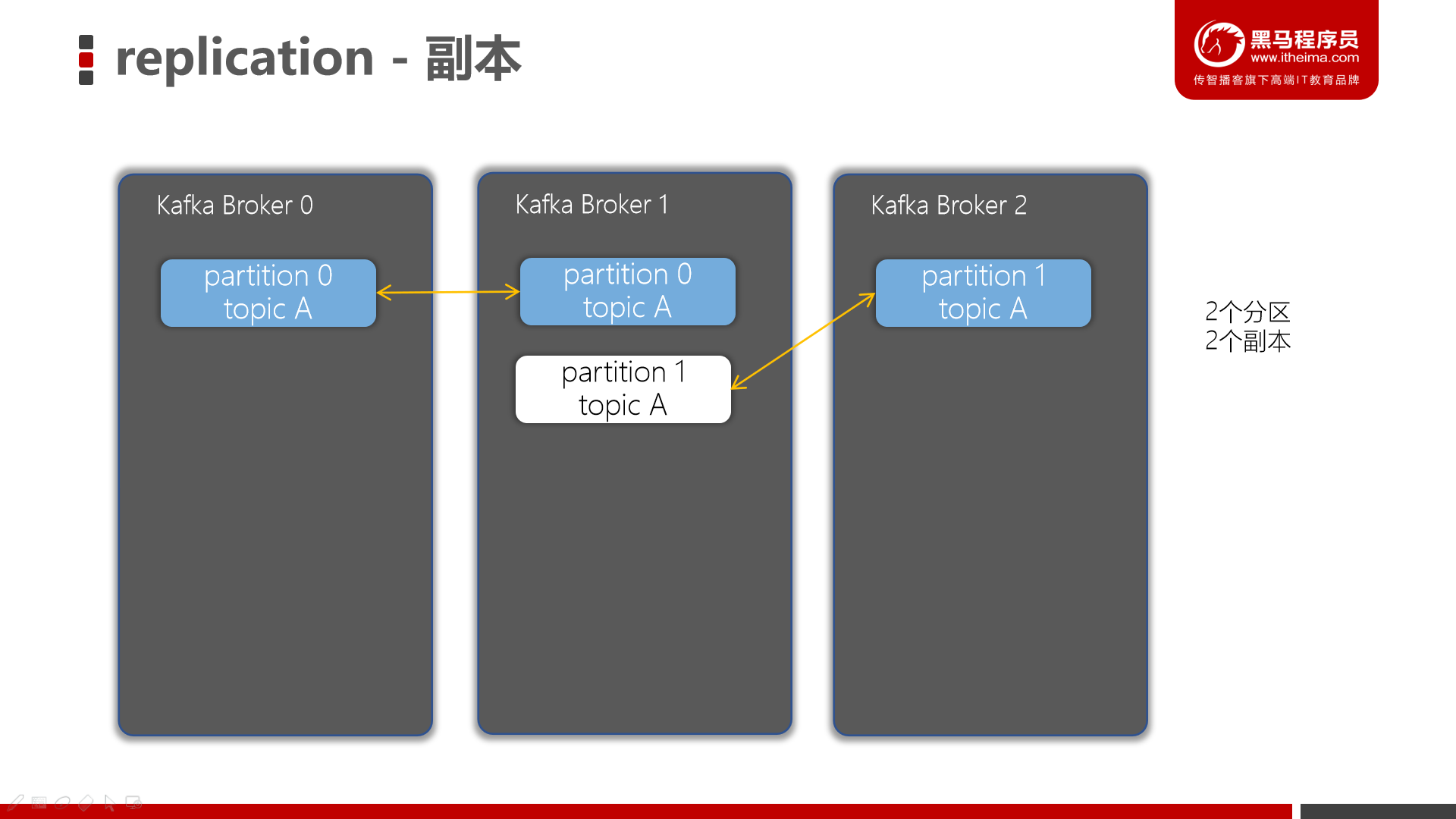
* consumer group是kafka提供的可扩展且具有容错性的消费者机制
* 一个消费者组可以包含多个消费者
* 一个消费者组有一个唯一的ID（group Id）
* 组内的消费者一起消费主题的所有分区数据

#### 分区（Partitions）



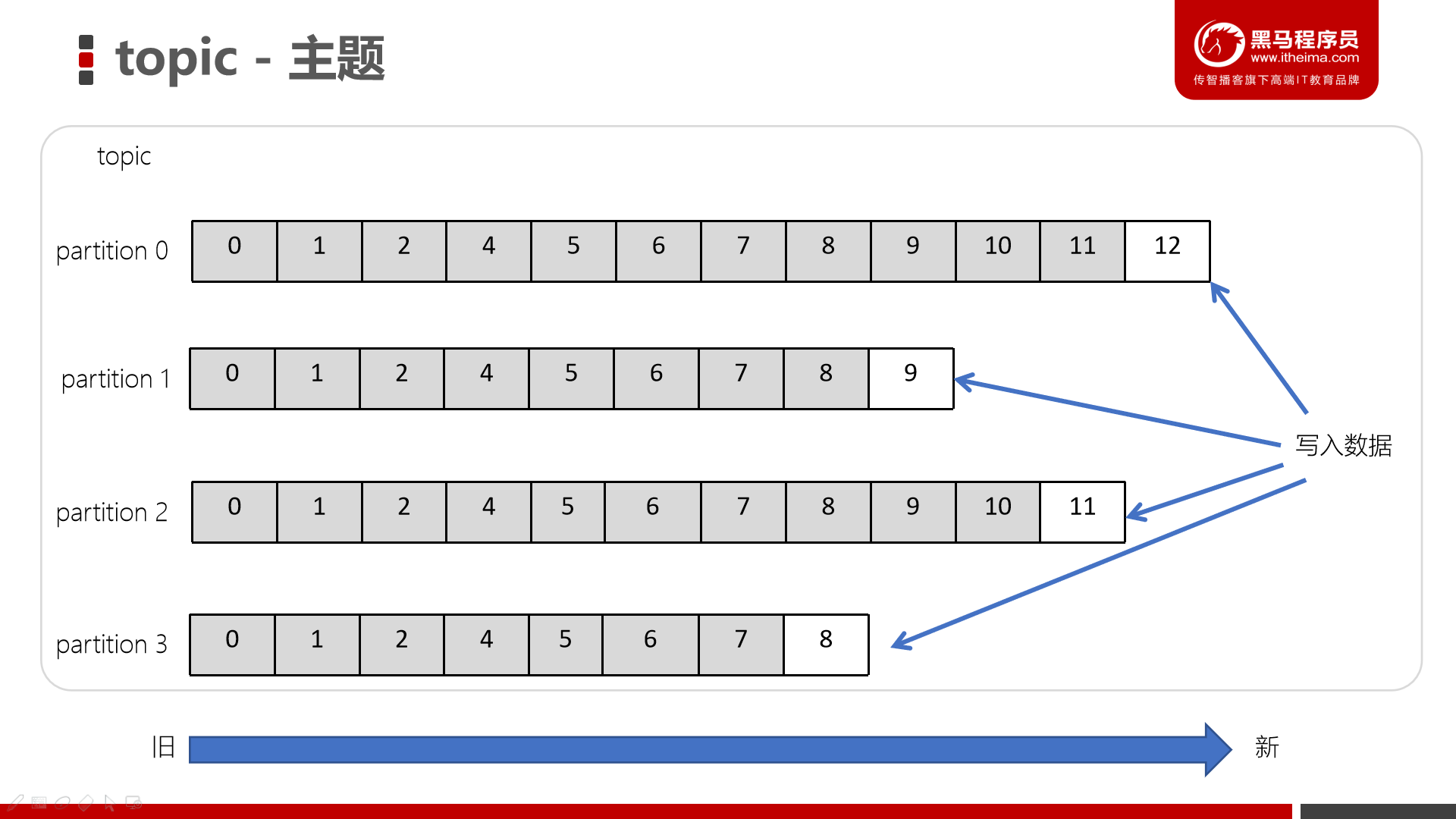
* 在Kafka集群中，主题被分为多个分区

#### 副本（Replicas）



* 副本可以确保某个服务器出现故障时，确保数据依然可用
* 在Kafka中，一般都会设计副本的个数＞1

#### 主题（Topic）



* 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）**

#### 偏移量（offset）



* **offset记录着下一条将要发送给Consumer的消息的序号**
* **默认Kafka将offset存储在ZooKeeper中**
* 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

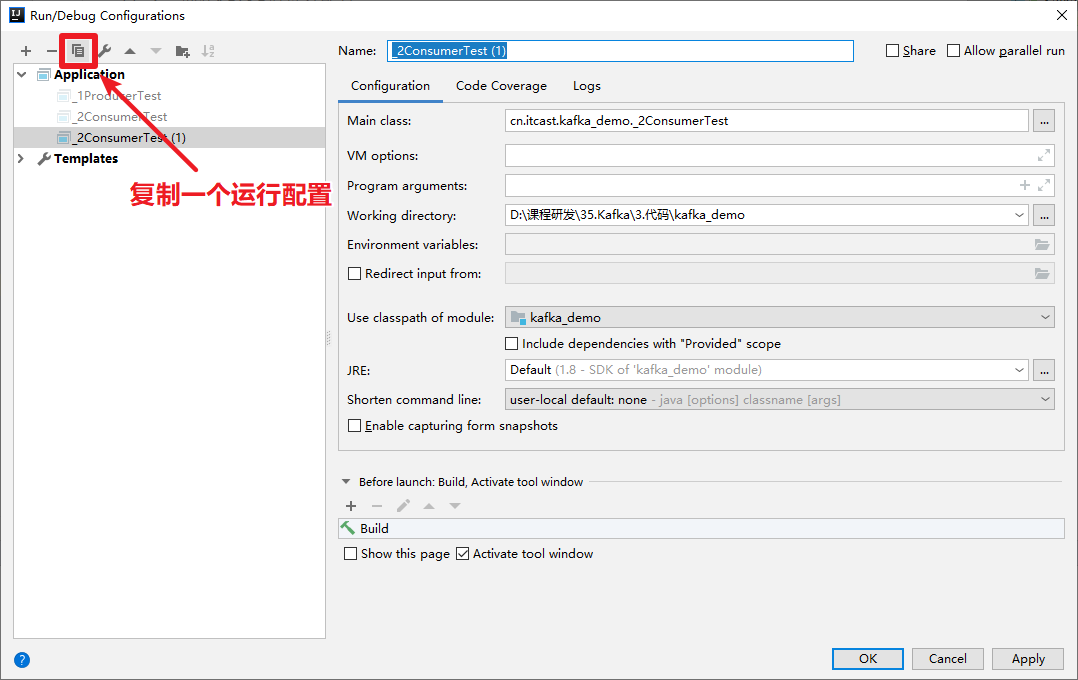
### 消费者组

Kafka支持有**多个消费者同时消费一个主题中的数据**。我们接下来，给大家演示，启动两个消费者共同来消费 test 主题的数据。

1. 首先，修改生产者程序，让生产者每3秒生产1-100个数字。

|  |
| --- |
| // 3. 发送1-100数字到Kafka的test主题中 **while**(**true**) {  **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  // 这样可以让消息发送变得更高效  producer.send(**new** ProducerRecord<>(**"test"**, i + **""**));  }  Thread.*sleep*(3000); } |

1. 接下来，同时运行两个消费者。



1. 同时运行两个消费者，**我们发现，只有1个分区，只有一个消费者程序能够拉取到消息**。

**注意事项：想要让两个消费者同时消费消息，必须要给test主题，添加一个分区。**

|  |
| --- |
| # 设置 test topic为2个分区  bin/kafka-topics.sh --zookeeper 192.168.88.100:2181 -alter --partitions 2 --topic test |

1. 重新运行生产者、两个消费者程序，我们就可以看到两个消费者都可以消费Kafka Topic的数据了

## Kafka生产者幂等性与事务

### 幂等性

解决生产者消息重复问题，通过附带pid和ack机制解决(生产者的seqnum<=partition的seqnum)，**保证发送消息无论发送多少次都能确保仅仅一次发送**。

随机数程序---seed概念---随机数种子---确保每次随机的随机数选择的是一样

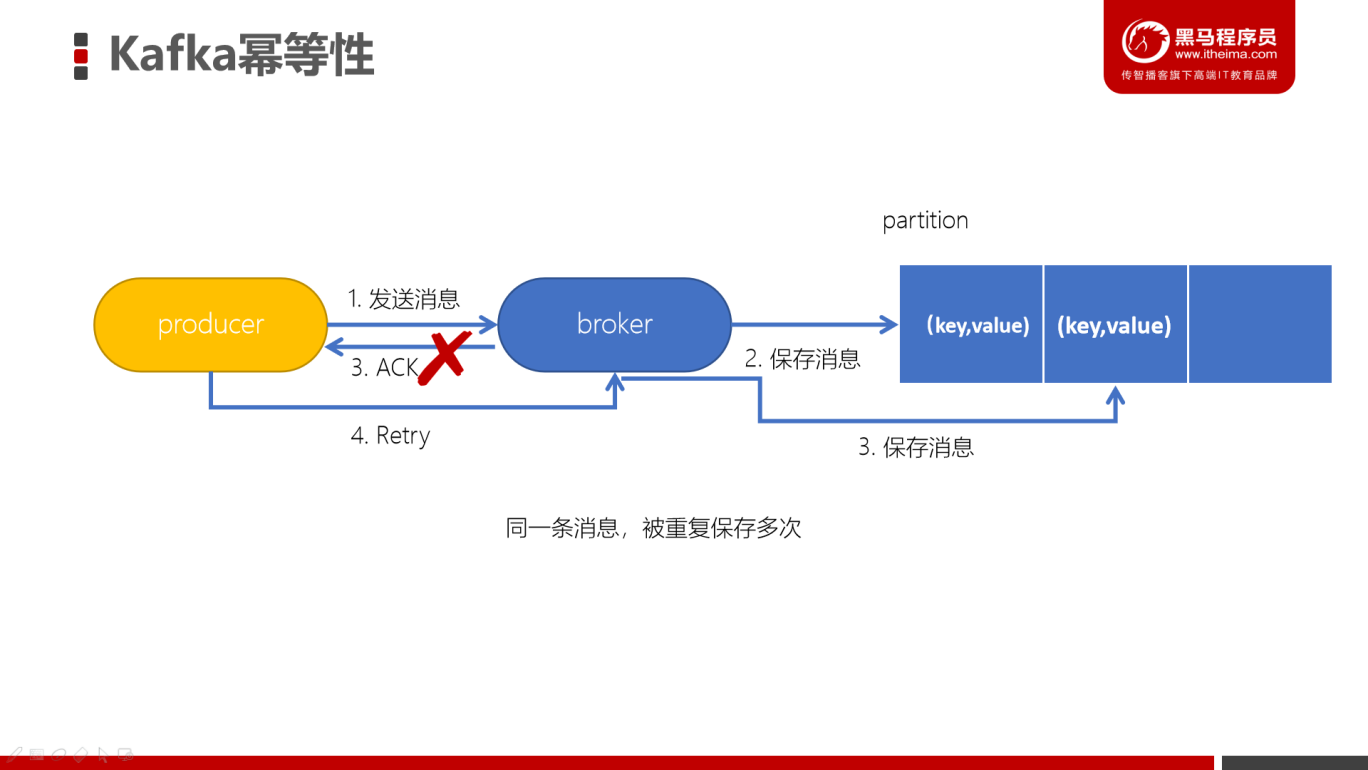
#### 简介

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**。



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。

#### Kafka生产者幂等性



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，就有可能会在partition中保存多条一模一样的消息。

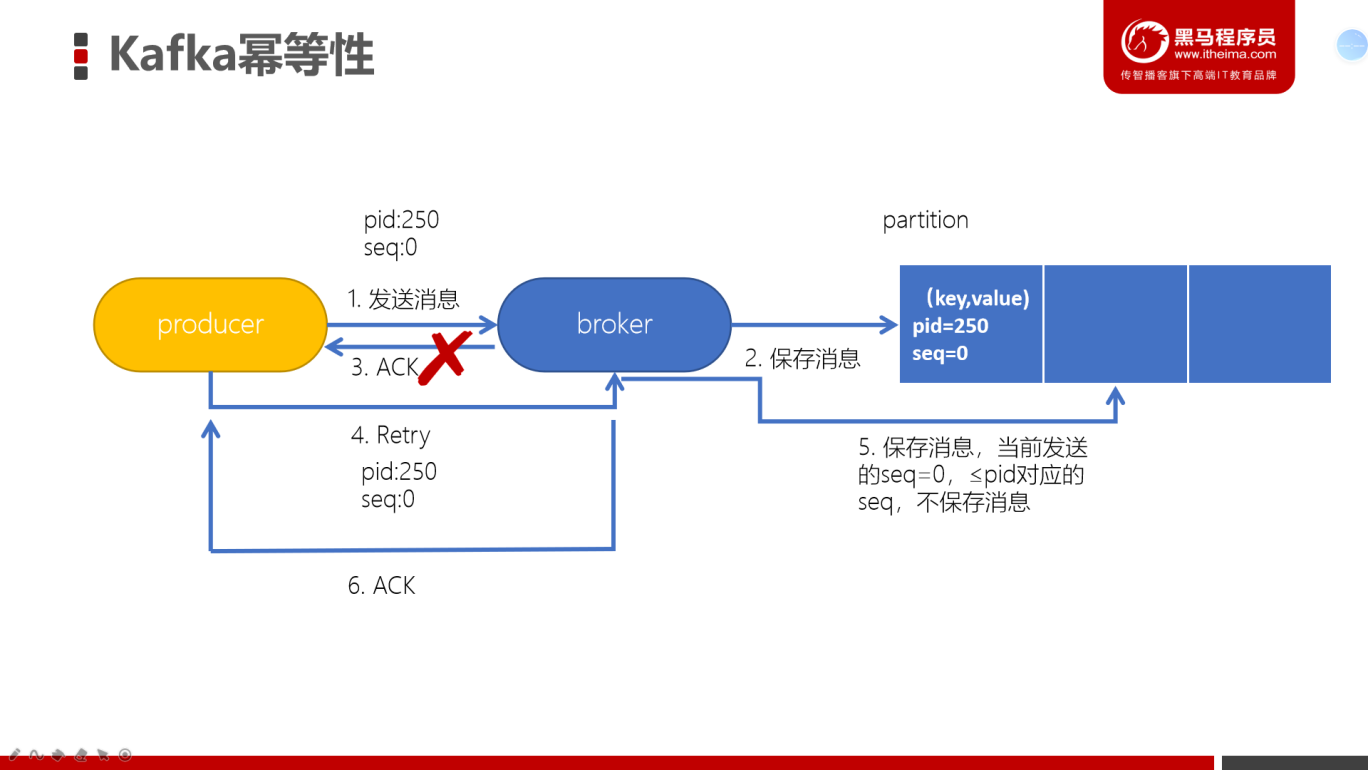
#### 配置幂等性

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

#### 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。

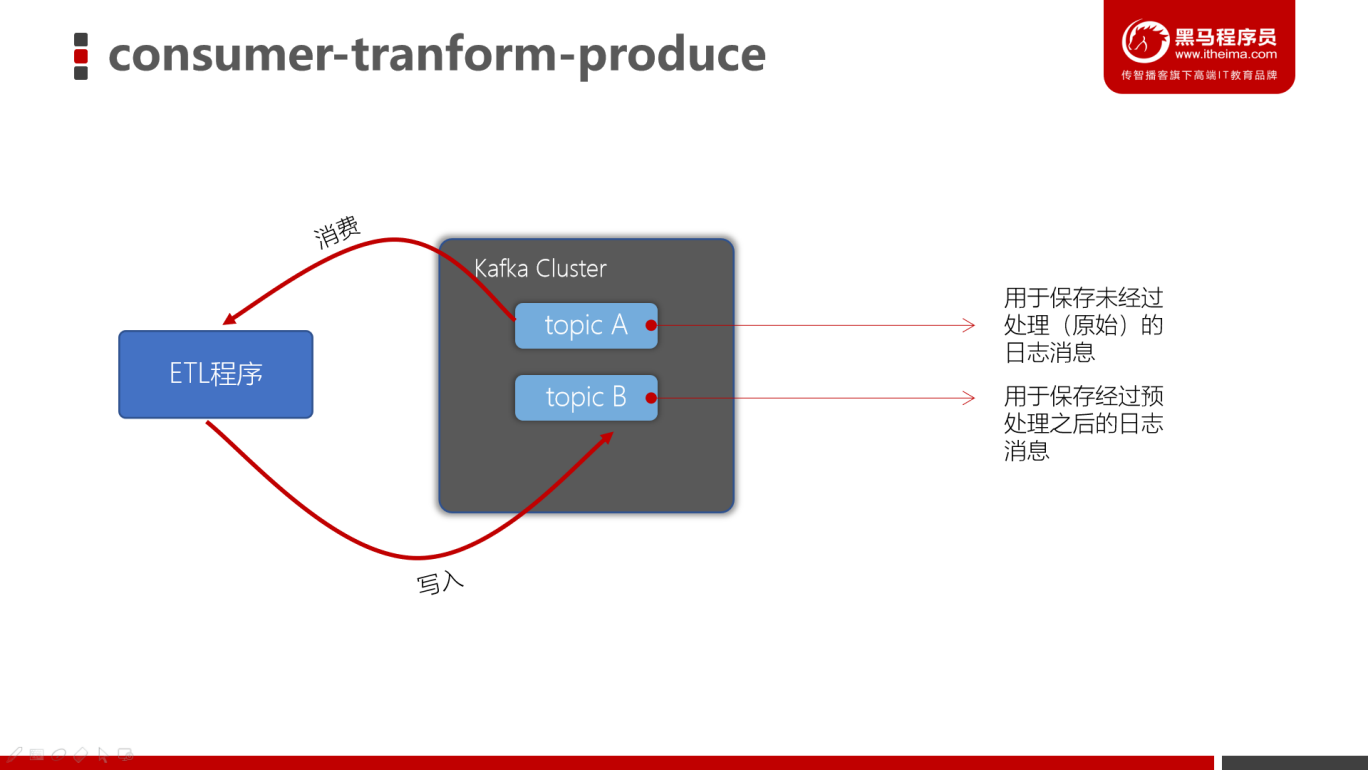
* PID：每个Producer在初始化时，都会分配一个唯一的PID，这个PID对用户来说，是透明的。
* Sequence Number：针对每个生产者（对应PID）发送到指定主题分区的消息都对应一个从0开始递增的Sequence Number。



### Kafka事务

#### 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）



#### 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）：要使用Kafka事务，必须先进行初始化操作
2. beginTransaction（开始事务）：启动一个Kafka事务
3. sendOffsetsToTransaction（提交偏移量）：批量地将分区对应的offset发送到事务中，方便后续一块提交
4. commitTransaction（提交事务）：提交事务
5. abortTransaction（放弃事务）：取消事务

### 【理解】Kafka事务编程

#### 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"**); |

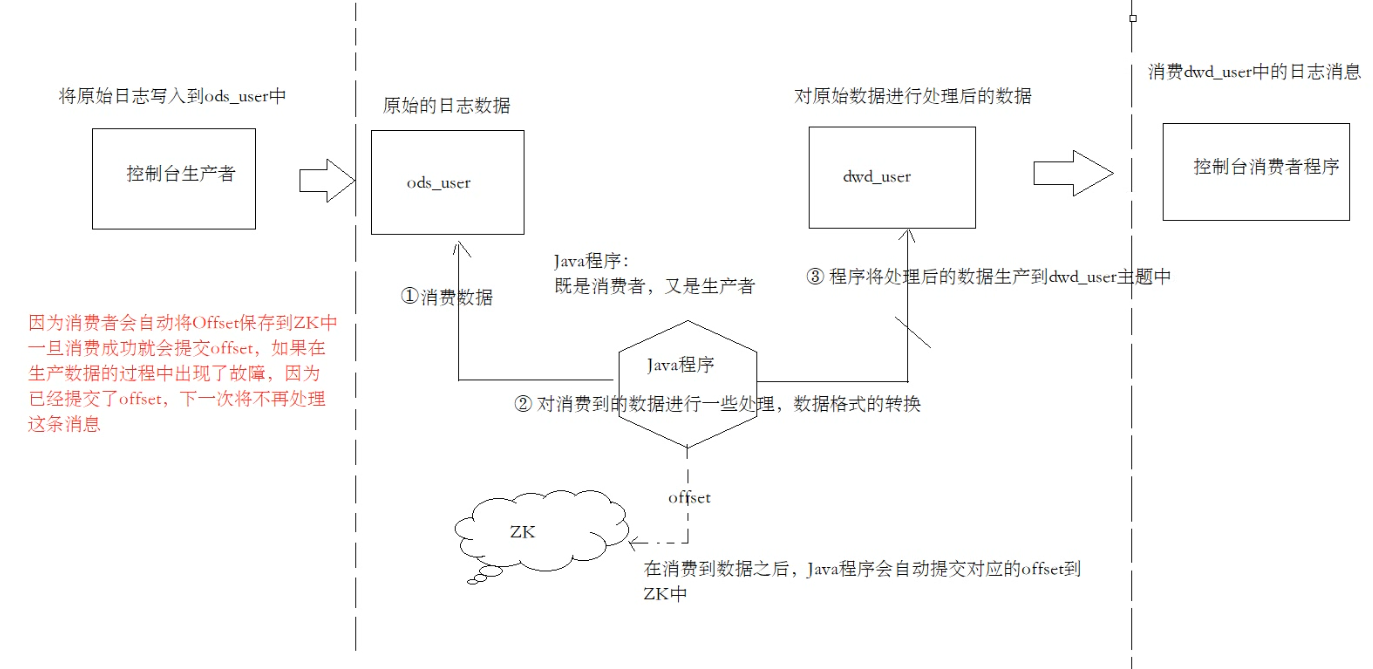
#### Kafka事务编程

##### 需求

在Kafka的topic 「ods\_user」中有一些用户数据，数据格式如下：

|  |
| --- |
| 姓名,性别,出生日期  张三,1,1980-10-09  李四,0,1985-11-01 |

我们需要编写程序，将用户的性别转换为男、女（1-男，0-女），转换后将数据写入到topic 「dwd\_user」中。要求使用事务保障，要么消费了数据同时写入数据到 topic，提交offset。要么全部失败。



##### 启动生产者控制台程序模拟数据

|  |
| --- |
| # 创建名为ods\_user和dwd\_user的主题  bin/kafka-topics.sh --create --bootstrap-server node1.itcast.cn:9092 --topic ods\_user  bin/kafka-topics.sh --create --bootstrap-server node1.itcast.cn:9092 --topic dwd\_user  # 生产数据到 ods\_user  bin/kafka-console-producer.sh --broker-list node1.itcast.cn:9092 --topic ods\_user  # 从dwd\_user消费数据  bin/kafka-console-consumer.sh --bootstrap-server node1.itcast.cn:9092 --topic dwd\_user --from-beginning --isolation-level read\_committed |

##### 编写创建消费者代码

编写一个方法 createConsumer，该方法中返回一个消费者，订阅「ods\_user」主题。注意：需要配置事务隔离级别、关闭自动提交。

【隔离级别】参考：http://kafka.apache.org/documentation/#api

isolation.level控制如何读取事务写入的消息。如果设置为read\_committed，consumer.poll（）将仅返回已提交的事务性消息。如果设置为read\_uncommitted'（默认设置），consumer.poll（）将返回所有消息，甚至是已中止的事务性消息。非事务性消息将以两种方式无条件返回。

实现步骤：

1. 创建Kafka消费者配置

|  |
| --- |
| Properties **props** **=** **new** Properties();  **props**.setProperty(**"bootstrap.servers"**, **"node1.itcast.cn:9092"**);  **props**.setProperty(**"group.id"**, **"ods\_user"**);  **props**.put(**"isolation.level"**,**"read\_committed"**);//配置事务隔离级别  **props**.setProperty(**"enable.auto.commit"**, **"false"**);//关闭自动提交  **props**.setProperty(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  **props**.setProperty(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**); |

1. 创建消费者，并订阅 ods\_user 主题

|  |
| --- |
| *// 1. 创建消费者*  **public** **static** Consumer**<**String, String**>** createConsumer() {  *// 1. 创建Kafka消费者配置*          Properties **props** **=** **new** Properties();  **props**.setProperty(**"bootstrap.servers"**, **"node1.itcast.cn:9092"**);  **props**.setProperty(**"group.id"**, **"ods\_user"**);  **props**.put(**"isolation.level"**,**"read\_committed"**);  **props**.setProperty(**"enable.auto.commit"**, **"false"**);//关闭自动提交，下面代码手动提交  **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(**"ods\_user"**));    **return** consumer;  } |

##### 编写创建生产者代码

编写一个方法 createProducer，返回一个生产者对象。注意：需要配置事务的id，开启了事务会默认开启幂等性。

1. 创建生产者配置

|  |
| --- |
| Properties **props** **=** **new** Properties();  **props**.put(**"bootstrap.servers"**, **"node1.itcast.cn:9092"**);  **props**.put(**"transactional.id"**, **"dwd\_user"**);  **props**.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  **props**.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**); |

1. 创建生产者对象

|  |
| --- |
| **public** **static** Producer**<**String, String**>** createProduceer() {  *// 1. 创建生产者配置*          Properties **props** **=** **new** Properties();  **props**.put(**"bootstrap.servers"**, **"node1.itcast.cn:9092"**);  **props**.put(**"transactional.id"**, **"dwd\_user"**);//配置事务的ID  **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);  **return** producer;      } |

##### 编写代码消费并生产数据

实现步骤：

1. 调用之前实现的方法，创建消费者、生产者对象
2. 生产者调用initTransactions初始化事务
3. 编写一个while死循环，在while循环中不断拉取数据，进行处理后，再写入到指定的topic
   1. 生产者开启事务
   2. 消费者拉取消息
   3. 遍历拉取到的消息，并进行预处理（将1转换为男，0转换为女）
   4. 生产消息到dwd\_user topic中
   5. 提交偏移量到事务中
   6. 提交事务
   7. 捕获异常，如果出现异常，则取消事务

|  |
| --- |
| **public** **static** void main(String[] args) {          Consumer<String, String> **consumer** **=** createConsumer();          Producer<String, String> **producer** **=** createProducer();  *// 初始化事务*  **producer**.initTransactions();  **while**(true) {  **try** {  ***// 1. 开启事务***  **producer**.beginTransaction();  *// 2. 定义Map结构，用于保存分区对应的offset*                  Map<TopicPartition, OffsetAndMetadata> **offsetCommits** **=** **new** HashMap<>();  *// 2. 拉取消息*                  ConsumerRecords<String, String> **records** **=** **consumer**.poll(**Duration**.ofSeconds(2));  //这里需要手动提交Offset  **for** (ConsumerRecord<String, String> **record** **:** records) {  *// 3. 保存偏移量,下面加1表示offset是当前消费记录(消息)对应在partition中的offset，而我们希望下一次能继续从下一个消息，必须加1，能从下一条消息消费*  **offsetCommits**.put(**new** TopicPartition(**record**.topic(), **record**.partition()),**new** OffsetAndMetadata(**record**.offset() + 1));  *// 4. 进行转换处理*                      String[] **fields** **=** **record**.value().split(**","**);                      fields[1] **=** fields[1].equalsIgnoreCase(**"1"**) **?** **"男":"女"**;                      String **message** **=** fields[0] **+** **","** **+** fields[1] **+** **","** **+** fields[2];  *// 5.****生产消息到dwd\_user，这里不能用callback的异步方式，只能用同步方式***  **producer**.send(**new** ProducerRecord<>(**"dwd\_user"**, message));                  }  *// 6. 提交偏移量到事务---Map(TopicPartition,OffSetAndMetaData),CousumerGroup*  **producer**.sendOffsetsToTransaction(offsetCommits, **"ods\_user"**);  *// 7. 提交事务*  **producer**.commitTransaction();              } **catch** (Exception **e**) {  **e.printStackTrace();//打印异常在工资台**  *// 8. 放弃事务*  **producer**.abortTransaction();              }          }      } |

##### 测试

往之前启动的console-producer中写入消息进行测试，同时检查console-consumer是否能够接收到消息：



逐个测试一下消息：

|  |
| --- |
| 张三,1,1980-10-09  李四,0,1985-11-01 |

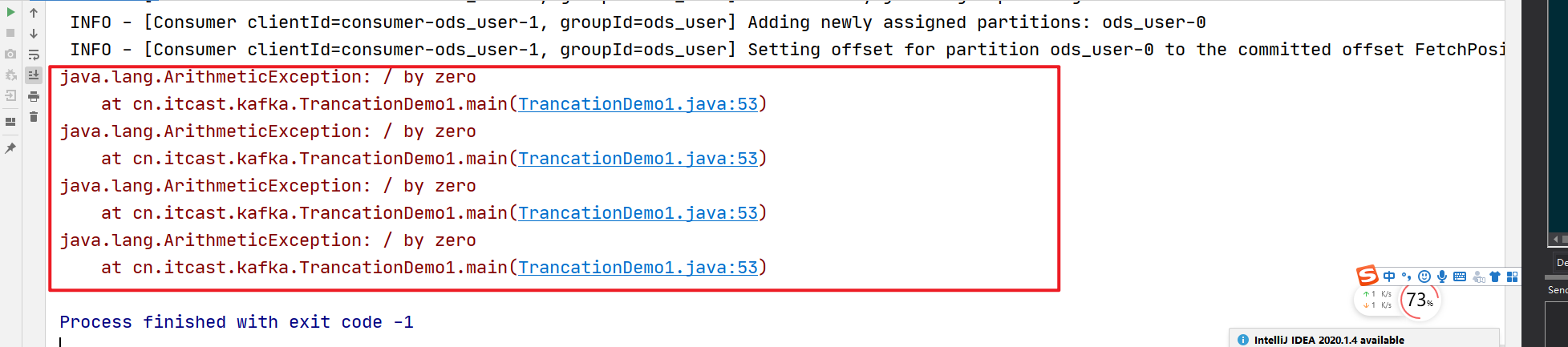
##### 模拟异常测试事务

|  |
| --- |
| *// 3. 保存偏移量*  **offsetCommits**.put(**new** TopicPartition(**record**.topic(), **record**.partition()),  **new** OffsetAndMetadata(**record**.offset() **+** 1));  *// 4. 进行转换处理*  String[] **fields** **=** **record**.value().split(**","**);  fields[1] **=** fields[1].equalsIgnoreCase(**"1"**) **?** **"男":"女"**;  String **message** **=** fields[0] **+** **","** **+** fields[1] **+** **","** **+** fields[2];  *// 模拟异常*  int **i** **=** 1**/**0;  *// 5. 生产消息到dwd\_user*  **producer**.send(**new** ProducerRecord<>(**"dwd\_user"**, message)); |

启动程序一次，抛出异常。

再启动程序一次，还是抛出异常。

直到我们处理该异常为止。



我们发现，可以消费到消息，但如果中间出现异常的话，offset是不会被提交的，除非消费、生产消息都成功，才会提交事务。