

## 今天任务

- 1、kafka介绍
- 2、kafka的单机版搭建
- 3、kafka的集群搭建
- 4、kafka的常用命令

## 教学目标

- 1、kafka概念
- 2、Jms概念
- 3、kafka集群搭建
- 4、kafka常用命令

# 第一章 Kafka简介

在学习kafka之前，咱们先来了解JMS，了解完这个一定程度上能帮助咱们更加深入学习kafka。

## 1.1 JMS规范介绍

### 1.1.1 JMS简介

jms(全称java message service)即java消息服务应用程序接口，是一个Java平台中关于面向消息中间件(MOM-分布式系统的集成)的API，用于在两个应用程序之间或者分布式系统中发送消息，进行异步通信。

JMS是一种与厂商无关的 API，用来访问消息收发系统消息，它类似于JDBC(Java Database Connectivity)。

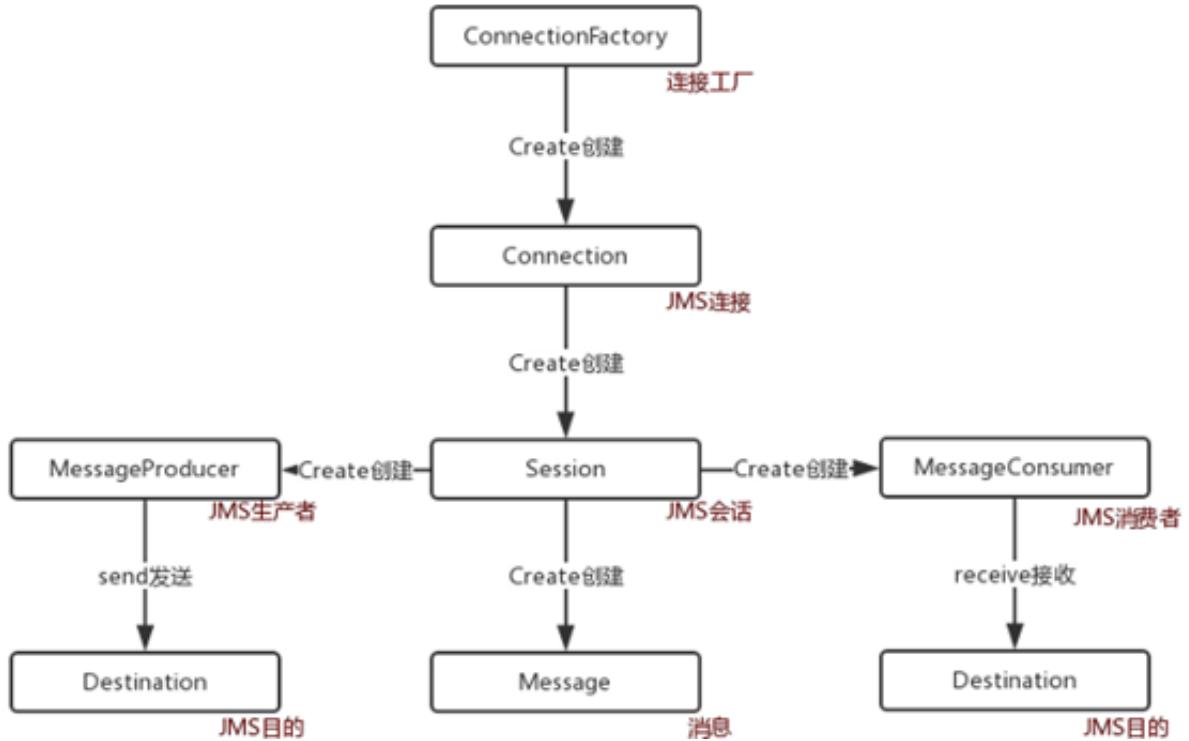
JMS可以自己使用java代码或者别的代码来编写，开源的实现有ActiveMQ、阿里的[RocketMQ](#)(已经贡献给apache)、kafka等。

### 1.1.2 JMS核心组件

构成JMS的常用组件和作用如下表：

组件名	组件作用
JMS提供者	连接面向消息中间件的，JMS接口的一个实现。提供者可以是Java平台的JMS实现，也可以是非Java平台的面向消息中间件的适配器。
JMS客户	生产或消费基于消息的Java的应用程序或对象。
JMS生产者	创建并发送消息的JMS客户。
JMS消费者	接收消息的JMS客户。
JMS消息	包括可以在JMS客户之间传递的数据的对象。
JMS队列	一个容纳那些被发送的等待阅读的消息的区域。与队列名字所暗示的意思不同，消息的接受顺序并不一定要与消息的发送顺序相同。一旦一个消息被阅读，该消息将被从队列中移走。
JMS主题	一种支持发送消息给多个订阅者的机制。

**JMS对象模型：**



组件及其作用：

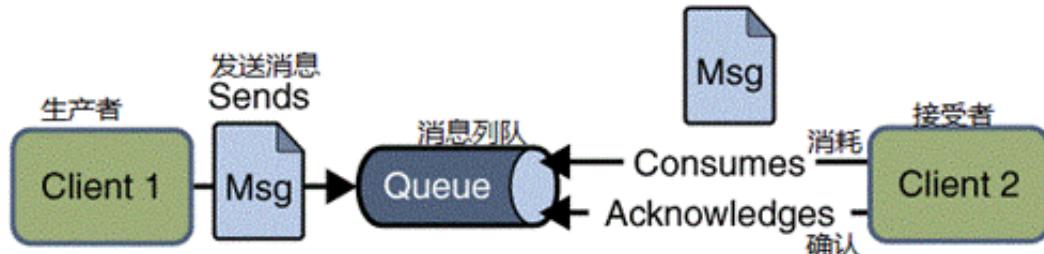
<b>ConnectionFactory</b>	创建 <b>Connection</b> 对象的工厂，针对两种不同的JMS消息模型，分别有 <b>QueueConnectionFactory</b> 和 <b>TopicConnectionFactory</b> 两种。可以通过JNDI来查找 <b>ConnectionFactory</b> 对象。
<b>Connection</b>	<b>Connection</b> 表示在客户端和JMS系统之间建立的链接（对TCP/IP socket的包装）。 <b>Connection</b> 可以产生一个或多个 <b>Session</b> 。跟 <b>ConnectionFactory</b> 一样， <b>Connection</b> 也有两种类型： <b>QueueConnection</b> 和 <b>TopicConnection</b> 。
<b>Session</b>	<b>Session</b> 是操作消息的接口。可以通过 <b>session</b> 创建生产者、消费者、消息等。 <b>Session</b> 提供了事务的功能。当需要使用 <b>session</b> 发送/接收多个消息时，可以将这些发送/接收动作放到一个事务中。同样，也分 <b>QueueSession</b> 和 <b>TopicSession</b> 。
<b>MessageProducer</b>	消息生产者由 <b>Session</b> 创建，并用于将消息发送到 <b>Destination</b> 。同样，消息生产者分两种类型： <b>QueueSender</b> 和 <b>TopicPublisher</b> 。可以调用消息生产者的方法（ <b>send</b> 或 <b>publish</b> 方法）发送消息。
<b>MessageConsumer</b>	消息消费者由 <b>Session</b> 创建，用于接收被发送到 <b>Destination</b> 的消息。两种类型： <b>QueueReceiver</b> 和 <b>TopicSubscriber</b> 。可分别通过 <b>session</b> 的 <b>createReceiver(Queue)</b> 或 <b>createSubscriber(Topic)</b> 来创建。当然，也可以 <b>session</b> 的 <b>createDurableSubscriber</b> 方法来创建持久化的订阅者。
<b>Destination</b>	<b>Destination</b> 的意思是消息生产者的消息发送目标或者说消息消费者的的消息来源。对于消息生产者来说，它的 <b>Destination</b> 是某个队列（ <b>Queue</b> ）或某个主题（ <b>Topic</b> ）；对于消息消费者来说，它的 <b>Destination</b> 也是某个队列或主题（即消息来源）。

### 1.1.3 JMS消息传输模型

在JMS标准中，有两种消息模型PTP (Point to Point) ,Publish/Subscribe(Pub/Sub)。

- PTP模式-点对点消息传送模型

在点对点消息传送模型中，应用程序由消息队列，发送者，接收者组成。每一个消息发送给一个特殊的消息队列，该队列保存了所有发送给它的消息(除了被接收者消费掉的和过期的消息)。



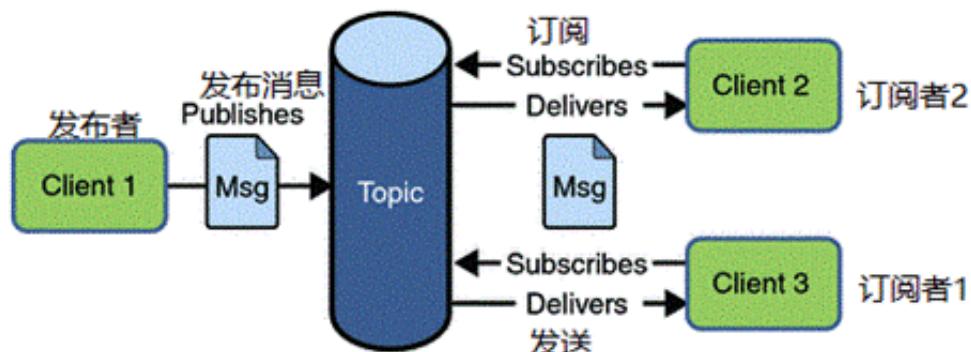
PTP的特点:

1. 每个消息只有一个消费者 (Consumer) (即一旦被消费，消息就不再在消息队列中)。
2. 发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列。
3. 接收者在成功接收消息之后需向队列发送确认收到通知 (acknowledgement) 。

- Pub/Sub-发布/订阅消息传递模型

在发布/订阅消息模型中，发布者发布一个消息，该消息通过topic传递给所有的客户端。在这种模型中，发布者和订阅者彼此不知道对方，是匿名的且可以动态发布和订阅topic。

在发布/订阅消息模型中，目的地被称为主题 (topic) ，topic主要用于保存和传递消息，且会一直保存消息直到消息被传递给客户端。



Pub/Sub特点:

1. 每个消息可以有多个消费者。
2. 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，它必须创建一个或多个订阅者之后，才能消费发布者的消息，而且为了消费消息，订阅者必须保持运行的状态。
3. 为了缓和这样严格的时间相关性，JMS允许订阅者创建一个可持久化的订阅。这样，即使订阅者没有被激活 (运行)，它也能接收到发布者的消息。

#### 消息接收

在JMS中，消息的接收可以使用以下两种方式：

同步	使用同步方式接收消息的话，消息订阅者调用 <b>receive()</b> 方法。在 <b>receive()</b> 中，消息未到达或在到达指定时间之前，方法会阻塞，直到消息可用。
异步	使用异步方式接收消息的话，消息订阅者需注册一个消息监听者，类似于事件监听器，只要消息到达，JMS服务提供者会通过调用监听器的 <b>onMessage()</b> 递送消息。

## JMS消息结构 (Message)

Message主要由三部分组成，分别是Header，Properties，Body，详细如下：

<b>Header</b>	消息头，所有类型的这部分格式都是一样的
Properties	属性，按类型可以分为应用设置的属性，标准属性和消息中间件定义的属性
Body	消息正文，指我们具体需要消息传输的内容。

JMS介绍到此结束，对消息及队列有一个大致的了解。

## 1.2 kafka简介

- Apache Kafka是一个开源**消息**系统、一个开源分布式流平台，由Scala写成。是由Apache软件基金会开发的一个开源消息系统项目。
- Kafka最初是由LinkedIn开发，并于2011年初开源。2012年10月从Apache Incubator毕业。该项目设计目标是为处理实时数据提供一个**统一、高吞吐量、低等待**的平台。
- Kafka是一个分布式消息队列：生产者、消费者的功能。它提供了类似于JMS的特性，但是在设计实现上完全不同，此外它并不是JMS规范的实现。
- Kafka对消息保存时根据Topic进行归类，发送消息者称为Producer,消息接受者称为Consumer,此外kafka集群有多个kafka实例组成，每个实例(server)称为broker。
- 无论是kafka集群，还是producer和consumer都依赖于**zookeeper**集群保存一些meta信息，来保证系统可用性。

## 1.3 kafka的发展历史

- 2010年底，开源到github，初始版本为0.7.0；
- 2011年7月因为备受关注，kafka正式捐赠给apache进行孵化；
- 2012年10月，kafka从apache孵化器项目毕业，成为apache顶级项目；
- 2014年，jay kreps,neha narkhede,jun rao离开linkedin,成立confluent,此后linkedin和confluent成为kafka的核心贡献组织，致力于将kafka推广应用；
- 至今是kafka最新版本为2.3.0，增加诸多新特性，比如集成了分发、存储和计算的“流式数据平台”，至此，kafka不再是一个简单的分布式消息系统。

## 1.4 kafka的常见应用场景

**日志收集：**一个公司可以用Kafka可以收集各种服务的log，通过kafka以统一接口服务的方式开放给各种consumer，例如hadoop、Hbase、Solr等。

**消息系统：**解耦和生产者和消费者、缓存消息等。

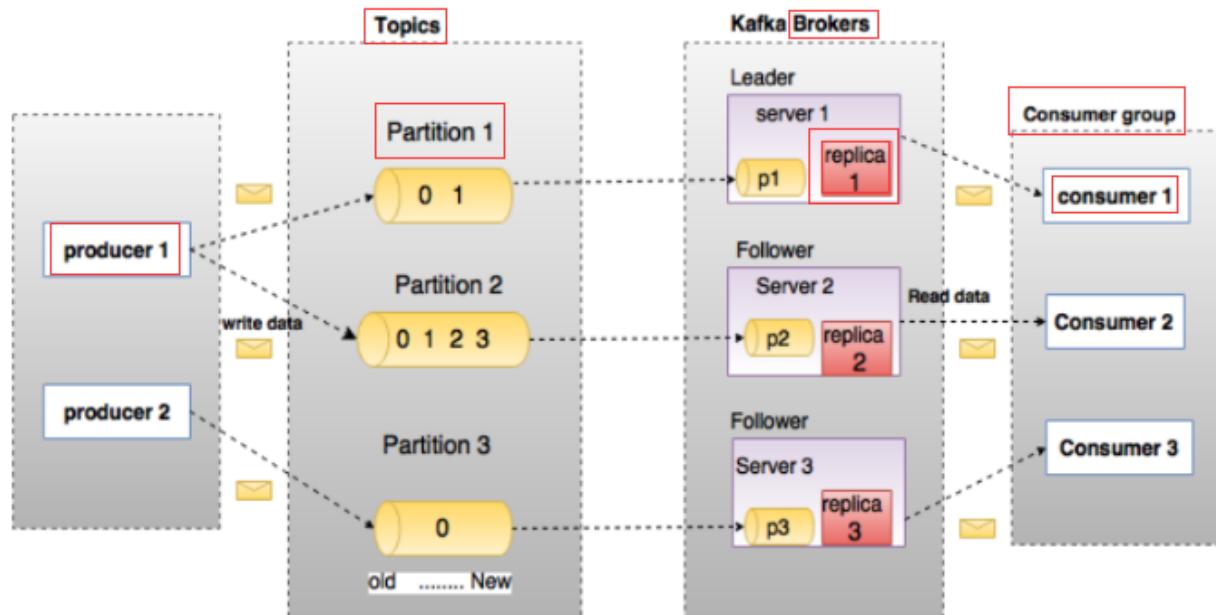
**用户活动跟踪:** Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时的监控分析，或者装载到hadoop、数据仓库中做离线分析和挖掘。

**运营指标:** Kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告。

**流式处理:** 比如spark streaming、flink和storm等整合应用。

## 1.5 kafka的核心概念

如图中，红色方框里面的都是kafka相对较核心概念，具体阐述如下：



**Broker(代理):** Kafka 节点，一个 Kafka 节点就是一个 broker，多个 broker 可以组成一个 Kafka 集群,每一个broker可以有多个topic。

**Producer(生产者):** 生产 message (数据)发送到 topic。

**Consumer(消费者):** 订阅 topic 消费 message， consumer 作为一个线程来消费。

**Consumer Group(消费组):** 一个 Consumer Group 包含多个 consumer，这个是预先在配置文件中配置好的。

**Topic(主题):** 一种类别，每一条发送到kafka集群的消息都可以有一个类别，这个类别叫做topic,不同的消息会进行分开存储，如果topic很大，可以分布到多个broker上，例如 page view 日志、click 日志等都可以以 topic的形式存在，Kafka 集群能够同时负责多个 topic 的分发。也可以这样理解：topic被认为是一个队列，每一条消息都必须指定它的topic，可以说我们需要明确把消息放入哪一个队列。

**Partition(分区):** topic 物理上的分组，一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列。

**Replicas(副本):** 每一个分区，根据副本因子N，会有N个副本。比如在broker1上有一个topic，分区为topic-1, 副本因子为2，那么在两个broker的数据目录里，就都有一个topic-1,其中一个一个是leader，一个replicas。

**Segment:** partition 物理上由多个 segment 组成，每个 Segment 存着 message 信息。

## 1.6 kafka的四大核心API简介

- The [Producer API](#) allows an application to publish a stream of records to one or more Kafka topics.

Producer API(生产者API)允许一个应用程序去推送流式记录到一个或者多个kafka的topic中。

- The [Consumer API](#) allows an application to subscribe to one or more topics and process the stream of records produced to them.

Consumer API(消费者API)允许一个应用程序去订阅消费一个或者多个主题，并处理生产给他们的流式记录。

- The [Streams API](#) allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

Streams API(流式API)允许应用程序作为一个流处理器，消费一个或多个主题的输入流，并生成一个或多个输出主题到输出流，从而有效地将输入流转换为输出流。

- The [Connector API](#) allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

Connector API(连接器API)允许构建和运行将Kafka主题连接到已经存在应用程序或数据系统的可重用生产者或消费者。例如，到关系数据库的连接器可能捕获对表的每个更改。

## 第二章 kafka环境部署

### 2.1 kafka单机版安装

假设你是第一次开始学习，并且之前没有的Kafka或ZooKeeper数据。由于基于unix和Windows平台的Kafka控制脚本不同，所以在Windows平台上使用bin\ Windows \而不是bin/，并将脚本扩展名更改为.bat。

单机版安装简单，适用于快速简单测试或者是学习。

参考1.1.x官网：<http://kafka.apache.org/11/documentation.html>

#### 2.1.1 下载安装包

##### 下载

下载地址：[http://archive.apache.org/dist/kafka/1.1.1/kafka\\_2.11-1.1.1.tgz](http://archive.apache.org/dist/kafka/1.1.1/kafka_2.11-1.1.1.tgz)

##### 解压

将下载的压缩包上传到服务器的任何一个目录(只要你记住即可)。然后解压到/usr/local目录下

```
[root@hadoop01 home]# tar -zxvf /home/kafka_2.11-1.1.1.tgz -C /usr/local/
[root@hadoop01 home]# cd /usr/local/kafka_2.11-1.1.1/
```

#### 2.1.2 启动服务

## 启动zookeeper服务

kafka严重依赖zookeeper，所以我们需要启动zookeeper服务， kafka提供给咱们快速使用的zk服务，即解压开压缩包，其根目录下的bin目录下提供zk的启停服务，但仅适用于单机模式(zk也是单实例)。多broker模式强烈建议使用自行部署的zookeeper。

kafka自带的zookeeper的启停脚本如下：

```
[root@hadoop01 kafka_2.11-1.1.1]# ll ./bin/zookeeper-server-st*
-rwxr-xr-x. 1 root root 1393 Jul  7 2018 ./bin/zookeeper-server-start.sh
-rwxr-xr-x. 1 root root 1001 Jul  7 2018 ./bin/zookeeper-server-stop.sh
```

使用kafka自带的zookeeper脚本启动zookeeper服务：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/zookeeper-server-start.sh
./config/zookeeper.properties &      ### &表示挂载后台启动
```

查看kafka自带的zookeeper服务是否启动成功：

```
[root@hadoop01 kafka_2.11-1.1.1]# jps
```

```
[root@hadoop01 kafka_2.11-1.1.1]# jps
5603 Jps
4966 QuorumPeerMain
```

## 启动kafka的服务

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-server-start.sh
./config/server.properties &      ###同样，挂载后台启动
```

查看kafka是否启动成功：

```
[root@hadoop01 kafka_2.11-1.1.1]# jps
5636 Kafka
5957 Jps
4966 QuorumPeerMain
```

到此为止， kafka单机版已经启动成功。

### 2.1.3 创建topic测试

创建一个单分区和单副本的名为"test"的topic：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor 1 --partitions 1 --topic test      ###创建一个
副本和一个分区的topic， topic名称为test
```

查看topic是否创建成功：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --list --zookeeper
localhost:2181    ### 列出现有的topic
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --list --zookeeper localhost:2181
[2019-11-12 14:43:28,658] INFO Accepted socket connection from /0:0:0:0:0:0:1:52895 (org.apache.zookeeper.server.NIOServerCnxnFactory)
[2019-11-12 14:43:28,662] INFO Client attempting to establish new session at /0:0:0:0:0:0:1:52895 (org.apache.zookeeper.server.ZooKeeperServer)
[2019-11-12 14:43:28,664] INFO Established session 0x16e5dc6e5050002 with negotiated timeout 30000 for client /0:0:0:0:0:0:1:52895 (org.apache.zookeeper.server.ZooKeeper)
test
```

注:另外，你可以配置自动创建主题的配置，即当创建一个不存在主题时将会自动创建而不是手动创建。

## 2.1.4 发送消息

Kafka自带一个命令行客户端(client)，它将从文件或标准输入中获取输入，并将其作为消息发送到Kafka集群。**默认情况下，每一行都将作为单独的消息发送。**

运行producer，然后输入一些信息到控制台并发送到服务端：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
>this is a message      ###发送的第一条消息
>this is anothor message  ###发送的第二条消息
### --broker-list是指定broker的服务器地址，默认端口是9092
```

## 2.1.5 消费消息

kafka有一个消费者客户端命令，它可以 转存消息到标准输出：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning    ###--bootstrap-server 指定kafkaserver地址 --from-beginning从开始位置开始消费，默认情况使用from-beginning
```

消费的信息如下：

```
>^C[root@hadoop01 kafka_2.11-1.1.1]#
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
[2019-11-12 15:40:48,317] INFO [GroupCoordinator 0]: Preparing to rebalance group console-consumer-48232 with old generation 0 (_consumer_offsets-20) (kafka.coordinator.group.GroupCoordinator)
[2019-11-12 15:40:48,318] INFO [GroupCoordinator 0]: Stabilized group console-consumer-48232 generation 1 (_consumer_offsets-20) (kafka.coordinator.group.GroupCoordinator)
[2019-11-12 15:40:48,323] INFO [GroupCoordinator 0]: Assignment received from leader for group console-consumer-48232 for generation 1 (kafka.coordinator.group.GroupCoordinator)
[2019-11-12 15:40:48,323] INFO Updated PartitionLeaderEpoch. New: {epoch:0, offset:0}, Current: {epoch:-1, offset:-1} for Partition: _consumer_offsets-20. Cache now contains 0 entries. (kafka.server.epoch.LeaderEpochFileCache)
this is a message
this is anothor message
```

**注意：**

如果有一个producer生产 消息终端，另外有多个consumer终端，那么多个consumer将会同时立即消费。(不在贴测试图)

所有命令行都有更多的额外参数，可以参考文档查看更多的options。

到此为止，单机版kafka的安装测试完成。

## 2.2 kafka多broker集群安装

到目前为止，我们如上运行的是单个broker，但是这并不好。对于kafka，单个broker仅仅是集群规模就一个，而多broker就是启动多个broker的实例(推荐在多台节点上启动)，其它的什么都不改变。为咯更好理解kafka，我们扩展我们的集群为三台节点(任然使用我们本地机器)。

### 2.2.1 集群规划

集群规模较庞大时，建议都需要提前规划。以便我们清晰服务和主机的映射关系，以便问题的追踪和后续的维护等等。

ip	主机名	kafka角色	备注
192.168.216.111	hadoop01	broker	启动broker角色
192.168.216.112	hadoop02	broker	如上
192.168.216.113	hadoop03	broker	如上

## 2.2.2 准备工作

安装jdk ---参考hdfs.md文档

安装zookeeper并启动 ---参考zookeeper.md文档

## 2.2.3 集群配置

### 2.2.3.1 下载

下载地址: [http://archive.apache.org/dist/kafka/1.1.1/kafka\\_2.11-1.1.1.tgz](http://archive.apache.org/dist/kafka/1.1.1/kafka_2.11-1.1.1.tgz)

### 2.2.3.2 解压

将下载的压缩包上传到服务器的任何一个目录(只要你记住即可)。然后解压到/usr/local目录下。

```
[root@hadoop01 ~]# tar -zxvf /home/kafka_2.11-1.1.1.tgz -C /usr/local/  
[root@hadoop01 ~]# cd /usr/local/kafka_2.11-1.1.1/
```

查看其目录：

```
[root@hadoop01 kafka_2.11-1.1.1]# ll  
total 56  
drwxr-xr-x. 3 root root 4096 Jul 7 2018 bin ← 启动停止脚本、（主题、消息）操作脚本  
drwxr-xr-x. 2 root root 4096 Jul 7 2018 config ← 配置目录，主要是server.properties文件  
drwxr-xr-x. 2 root root 4096 Nov 12 11:52 libs ← 依赖的一下jar包库  
-rw-r--r--. 1 root root 28824 Jul 7 2018 LICENSE ← 默认的日志目录，包括数据操作记录  
drwxr-xr-x. 2 root root 4096 Nov 12 21:06 logs ← 学习文档  
-rw-r--r--. 1 root root 336 Jul 7 2018 NOTICE ← 学习文档  
drwxr-xr-x. 2 root root 4096 Jul 7 2018 site-docs ← 学习文档
```

### 2.2.3.3 配置环境变量

```
[root@hadoop01 kafka_2.11-1.1.1]# vi /etc/profile
```

在文件末尾追加如下内容：

```
export KAFKA_HOME=/usr/local/kafka_2.11-1.1.1/  
export PATH=$PATH:$KAFKA_HOME/bin:
```

```
[root@hadoop01 kafka_2.11-1.1.1]# source /etc/profile
```

## 配置

需要配置的文件主要有3个：server.properties、producer.properties、consumer.properties。

选择在hadoop01中配置./config/server.properties文件

### 2.2.3.4 配置broker(server.properties)

```
[root@hadoop01 kafka_2.11-1.1.1]# vi ./config/server.properties
```

该配置文件覆盖如下内容：

```
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements. See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License. You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# see kafka.server.KafkaConfig for additional details and defaults

#####
# Server Basics #####
#####

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1

#####
# Socket Server Settings #####
#####

# The address the socket server listens on. It will get the value returned
# from
# java.net.InetAddress.getCanonicalHostName() if not configured.
# FORMAT:
#     listeners = listener_name://host_name:port
# EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
#listeners=PLAINTEXT://:9092

# Hostname and port the broker will advertise to producers and consumers. If
# not set,
# it uses the value for "listeners" if configured. Otherwise, it will use the
# value
```

```

# returned from java.net.InetAddress.getCanonicalHostName().
#advertised.listeners=PLAINTEXT://your.host.name:9092

# Maps listener names to security protocols, the default is for them to be the
# same. See the config documentation for more details
#listener.security.protocol.map=PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SAS
#L_PLAINTEXT,SASL_SSL:SASL_SSL

# The number of threads that the server uses for receiving requests from the
# network and sending responses to the network
num.network.threads=3

# The number of threads that the server uses for processing requests, which
# may include disk I/O
num.io.threads=8

# The send buffer (SO_SNDBUF) used by the socket server
socket.send.buffer.bytes=102400

# The receive buffer (SO_RCVBUF) used by the socket server
socket.receive.buffer.bytes=102400

# The maximum size of a request that the socket server will accept (protection
# against OOM)
socket.request.max.bytes=104857600

#####
##### Log Basics #####
#####

# A comma separated list of directories under which to store log files
log.dirs=/home/log/kafka-logs

# The default number of log partitions per topic. More partitions allow
# greater
# parallelism for consumption, but this will also result in more files across
# the brokers.
num.partitions=1

# The number of threads per data directory to be used for log recovery at
# startup and flushing at shutdown.
# This value is recommended to be increased for installations with data dirs
# located in RAID array.
num.recovery.threads.per.data.dir=1

#####
##### Internal Topic Settings #####
#####

# The replication factor for the group metadata internal topics
"__consumer_offsets" and "__transaction_state"

```

```

# For anything other than development testing, a value greater than 1 is
recommended for to ensure availability such as 3.
offsets.topic.replication.factor=1
transaction.state.log.replication.factor=1
transaction.state.log.min_isr=1

##### Log Flush Policy #####
# Messages are immediately written to the filesystem but by default we only
fsync() to sync
# the OS cache lazily. The following configurations control the flush of data
to disk.
# There are a few important trade-offs here:
#   1. Durability: Unflushed data may be lost if you are not using
replication.
#   2. Latency: Very large flush intervals may lead to latency spikes when
the flush does occur as there will be a lot of data to flush.
#   3. Throughput: The flush is generally the most expensive operation, and a
small flush interval may lead to excessive seeks.
# The settings below allow one to configure the flush policy to flush data
after a period of time or
# every N messages (or both). This can be done globally and overridden on a
per-topic basis.

# The number of messages to accept before forcing a flush of data to disk
#log.flush.interval.messages=10000

# The maximum amount of time a message can sit in a log before we force a
flush
#log.flush.interval.ms=1000

##### Log Retention Policy #####
# The following configurations control the disposal of log segments. The
policy can
# be set to delete segments after a period of time, or after a given size has
accumulated.
# A segment will be deleted whenever *either* of these criteria are met.
Deletion always happens
# from the end of the log.

# The minimum age of a log file to be eligible for deletion due to age
log.retention.hours=168

# A size-based retention policy for logs. Segments are pruned from the log
unless the remaining
# segments drop below log.retention.bytes. Functions independently of
log.retention.hours.

```

```

#log.retention.bytes=1073741824

# The maximum size of a log segment file. When this size is reached a new log
segment will be created.
log.segment.bytes=1073741824

# The interval at which log segments are checked to see if they can be deleted
according
# to the retention policies
log.retention.check.interval.ms=300000

##### Zookeeper #####
# zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=hadoop01:2181,hadoop02:2181,hadoop03:2181/kafka

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=6000

#####
# Group Coordinator Settings
#####

# The following configuration specifies the time, in milliseconds, that the
GroupCoordinator will delay the initial consumer rebalance.
# The rebalance will be further delayed by the value of
group.initial.rebalance.delay.ms as new members join the group, up to a
maximum of max.poll.interval.ms.
# The default value for this is 3 seconds.
# We override this to 0 here as it makes for a better out-of-the-box
experience for development and testing.
# However, in production environments the default value of 3 seconds is more
suitable as this will help to avoid unnecessary, and potentially expensive,
rebalances during application startup.
group.initial.rebalance.delay.ms=0

```

### 2.2.3.5 配置生产者(producer.properties)

```
[root@hadoop01 kafka_2.11-1.1.1]# vi ./config/producer.properties
```

覆盖如下内容：

```
# Licensed to the Apache Software Foundation (ASF) under one or more
```

```
# contributor license agreements. See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License. You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# see org.apache.kafka.clients.producer.ProducerConfig for more details

#####
# Producer Basics #####
#####

# list of brokers used for bootstrapping knowledge about the rest of the
# cluster
# format: host1:port1,host2:port2 ...
bootstrap.servers=hadoop01:9092,hadoop02:9092,hadoop03:9092

# specify the compression codec for all data generated: none, gzip, snappy,
# lz4
compression.type=none

# name of the partitioner class for partitioning events; default partition
# spreads data randomly
#partitioner.class=

# the maximum amount of time the client will wait for the response of a
# request
#request.timeout.ms=

# how long `KafkaProducer.send` and `KafkaProducer.partitionsFor` will block
# for
#max.block.ms=

# the producer will wait for up to the given delay to allow other records to
# be sent so that the sends can be batched together
#linger.ms=

# the maximum size of a request in bytes
#max.request.size=

# the default batch size in bytes when batching multiple records sent to a
# partition
#batch.size=
```

```
# the total bytes of memory the producer can use to buffer records waiting to  
be sent to the server  
#buffer.memory=
```

### 2.2.3.6 配置消费者(consumer.properties)

```
[root@hadoop01 kafka_2.11-1.1.1]# vi ./config/consumer.properties
```

覆盖内容如下：

```
# Licensed to the Apache Software Foundation (ASF) under one or more  
# contributor license agreements. See the NOTICE file distributed with  
# this work for additional information regarding copyright ownership.  
# The ASF licenses this file to You under the Apache License, Version 2.0  
# (the "License"); you may not use this file except in compliance with  
# the License. You may obtain a copy of the License at  
#  
#     http://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.  
# see org.apache.kafka.clients.consumer.ConsumerConfig for more details  
  
# list of brokers used for bootstrapping knowledge about the rest of the  
cluster  
# format: host1:port1,host2:port2 ...  
bootstrap.servers=hadoop01:9092,hadoop02:9092,hadoop03:9092  
  
# consumer group id  
group.id=group-test  
  
# What to do when there is no initial offset in Kafka or if the current  
# offset does not exist any more on the server: latest, earliest, none  
#auto.offset.reset=
```

### 2.2.3.7 分发

分发已经配置好的kafka目录到hadoop02和hadoop03节点的/usr/local目录下。

```
[root@hadoop01 kafka_2.11-1.1.1]# scp -r /usr/local/kafka_2.11-1.1.1/  
hadoop02:/usr/local/  
[root@hadoop01 kafka_2.11-1.1.1]# scp -r /usr/local/kafka_2.11-1.1.1/  
hadoop03:/usr/local/
```

修改hadoop02节点上server.properties中的id和host:

```
[root@hadoop02 ~]# cd /usr/local/kafka_2.11-1.1.1/
[root@hadoop02 kafka_2.11-1.1.1]# vi ./config/server.properties
修改如下内容:
broker.id=2    ##修改
host.name=hadoop02    ##修改
```

修改hadoop03节点上server.properties中的id和host:

```
[root@hadoop03 ~]# cd /usr/local/kafka_2.11-1.1.1/
[root@hadoop03 kafka_2.11-1.1.1]# vi ./config/server.properties
修改如下内容:
broker.id=3    ##修改
host.name=hadoop03    ##修改
```

到此为止， kafka的集群配置完成。

## 2.2.4 集群启停

必须先启动zookeeper集群。

```
[root@hadoop01 kafka_2.11-1.1.1]# zkServer.sh start
[root@hadoop02 kafka_2.11-1.1.1]# zkServer.sh start
[root@hadoop03 kafka_2.11-1.1.1]# zkServer.sh start
```

依次再每个节点启动kafka的broker

```
[root@hadoop01 kafka_2.11-1.1.1]# nohup ./bin/kafka-server-start.sh
./config/server.properties > /var/log/kafka.log 2>&1 &
[root@hadoop02 kafka_2.11-1.1.1]# nohup ./bin/kafka-server-start.sh
./config/server.properties > /var/log/kafka.log 2>&1 &
[root@hadoop03 kafka_2.11-1.1.1]# nohup ./bin/kafka-server-start.sh
./config/server.properties > /var/log/kafka.log 2>&1 &
```

## 2.2.5 集群测试

依次测试每台节点的进程是否启动

```
====hadoop01
[root@hadoop01 kafka_2.11-1.1.1]# jps
3109 Jps
2713 QuorumPeerMain
2780 Kafka

====hadoop02
[root@hadoop02 kafka_2.11-1.1.1]# jps
2643 Kafka
2967 Jps
```

```
2603 QuorumPeerMain

====hadoop03
[root@hadoop03 kafka_2.11-1.1.1]# jps
2690 Kafka
2658 QuorumPeerMain
3014 Jps
```

## 关闭集群

法一：脚本

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-server-stop.sh
```

注：

法一有的版本需要修改停止脚本，原内容如下：

```
PIDS=$(ps ax | grep -i 'kafka\.Kafka' | grep java | grep -v grep | awk '{print $1}')
```

修改后如下：

```
PIDS=$(ps ax | grep -i 'kafka' | grep java | grep -v grep | awk '{print $1}')
```

法二：直接kill -9 pid

```
[root@hadoop01 kafka_2.11-1.1.1]# jps
4262 Kafka
2713 QuorumPeerMain
4590 Jps
[root@hadoop01 kafka_2.11-1.1.1]# kill -9 4262
```

# 第三章 kafka操作

## 3.1 主题帮助命令

主题相关操作命令格式

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181 --操作 [--options]
```

主题帮助命令

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --help
Command must include exactly one action: --list, --describe, --create, --alter
or --delete
Option                                Description
-----                                 -----
--alter                               修改topic的分区数量、副本、配置等信息
```

--config <String: name=value> 被创建或者修改的主题的相关配置.如下是一些可用的配置列表:

```

        configurations:
            cleanup.policy
            compression.type
            delete.retention.ms
            file.delete.delay.ms
            flush.messages
            flush.ms

follower.replication.throttled.

replicas
    index.interval.bytes

leader.replication.throttled.replicas
    max.message.bytes
    message.format.version

message.timestamp.difference.max.ms
    message.timestamp.type
    min.cleanable.dirty.ratio
    min.compaction.lag.ms
    min.insync.replicas
    preallocate
    retention.bytes
    retention.ms
    segment.bytes
    segment.index.bytes
    segment.jitter.ms
    segment.ms
    unclean.leader.election.enable

更多的配置参考topic的更全的config配置信息.

--create 创建新的topic, 后紧跟topic名称. 如: --
topic test

--delete 删除topic, 后紧跟topic的名称. 如: topic
test

--delete-config <String: name> A topic configuration override to be
removed for an existing topic (see
the list of configurations under

the
    --config option).

--describe 列出给定topics的描述信息, 多个用逗号分隔
--disable-rack-aware 关闭机架感知副本分配
--force 强制执行
--help 打印使用帮助信息
--if-exists 如果更新或者删除topic, 这些操作被执行仅仅当
存在
--if-not-exists 如果创建topic时, 仅仅当不存在时该创建操作被
执行

```

--list	列出所有有效的topics
--partitions <Integer: # of partitions>	指定被创建或者修改的topic的分区数量
(WARNING:	If partitions are increased <b>for</b> a topic that has a key, the partition logic or ordering of the messages will be affected
--replica-assignment <String: broker_id_for_part1_replica1 : broker_id_for_part1_replica2 , broker_id_for_part2_replica1 : broker_id_for_part2_replica2 , ...>	A list of manual partition-to-broker assignments <b>for</b> the topic being created or altered.
--replication-factor <Integer: replication factor>	被创建主题的每一个分区的副本因子
--topic <String: topic>	创建、修改、描述的主题。创建操作时支持正则
--topics-with-overrides 的配置	如果设置上，当查看主题描述时，仅仅显示被覆盖
--unavailable-partitions	<b>if set</b> when describing topics, only show partitions whose leader is not available
--under-replicated-partitions	<b>if set</b> when describing topics, only show under replicated partitions
--zookeeper <String: hosts> 以使用多个主机	zookeeper的连接地址，格式: host:port. 可以地址来达到失败转移

## 3.2 列出所有主题

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --list --zookeeper  
hadoop01:2181,hadoop02:2181,hadoop03:2181
```

## 3.3 创建主题

创建主题会在zk中的brokers目录下创建对应的znode节点，同时也在消息存储目录下创建对应的"topic-分区编号"的目录。

```
#如下的参数缺一不可  
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --create --zookeeper  
hadoop01:2181,hadoop02:2181,hadoop03:2181 --replication-factor 1 --partitions  
1 --topic test
```

## 3.4 查看主题

--describe查看出来的信息，第一行是所以有分区的总记录、配置等信息。接下来的其它行，每一行代表一个分区信息。

- "leader" node节点是负责给定分区的所有读和写操作. Each node will be the leader for a randomly selected portion of the partitions。他后面跟的数字n就是broker的唯一id。
- "replicas" 是复制此分区的日志的节点列表，而不管这些节点是leader节点还是当前活动节点。后

续跟的数字也是broker的唯一id。

- "isr" 是一系列“在同步”的副本。这是副本列表的子集，该列表当前处于活动状态并与leader节点保持联系。它后续跟的数字任然是broker的唯一id。

```
#查看test主题的详细信息
```

```
[root@hadoop03 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --describe --zookeeper  
hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test  
Topic:test      PartitionCount:1          ReplicationFactor:1      Configs:      #  
总记录  
          Topic: test      Partition: 0      Leader: 2      Replicas: 2      Isr: 2  
#第一个分区
```

```
#查看test, test1的详细信息
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --describe --zookeeper  
hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test,test1  
Topic:test      PartitionCount:1          ReplicationFactor:1      Configs:  
          Topic: test      Partition: 0      Leader: 2      Replicas: 2      Isr: 2  
Topic:test1     PartitionCount:1          ReplicationFactor:1      Configs:  
          Topic: test1     Partition: 0      Leader: 2      Replicas: 2      Isr: 2
```

## 3.5 主题修改

可以修改主题分区和配置属性等信息，不能修改副本因子，同时也不能将分区数越改越小，例如 test\_topic当前分区3，修改为1，这将会报错。

```
#删除test主题的flush.ms配置，如没有该属性将会显示已经更新。可考虑先增加再删除
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --alter --zookeeper  
hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test --delete-config  
flush.ms
```

```
#修改增加配置
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --alter --zookeeper  
hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test --config flush.ms=1000
```

```
#修改后查看主题是否增加成功
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --describe --zookeeper  
hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test  
Topic:test      PartitionCount:1          ReplicationFactor:1  
Configs:flush.ms=1000  
          Topic: test      Partition: 0      Leader: 2      Replicas: 2      Isr: 2
```

```
#删除刚增加的配置，然后自行查看是否删除成功
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --alter --zookeeper  
hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test --delete-config  
flush.ms
```

```
#修改分，注意不能修改副本因子：--replication-factor；同时不能修改成相同分区数。
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --alter --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181 --partitions 3 --topic test

#查看修改分区后的topic
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --describe --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test
Topic:test      PartitionCount:3      ReplicationFactor:1
Configs:max.message.bytes=128000
    Topic: test      Partition: 0      Leader: 2      Replicas: 2      Isr: 2
    Topic: test      Partition: 1      Leader: 3      Replicas: 3      Isr: 3
    Topic: test      Partition: 2      Leader: 1      Replicas: 1      Isr: 1
```

## 3.5 删除主题

删除主题时，需要停止该删除topic相关的生产和消费者，否则会有意向不到问题。同时如果 delete.topic.enable的设置不是true，则将是标记删除，也就不是真正删除，是会list出来的。而设置为true时，将会真正将create时，所创建的信息都会给删除掉，即zk中brokers的topics的znode节点删除，删除对应的分区目录。

思考：能否删除指定topic的指定分区呢？

```
#删除test1主题，可以--list查看是否能出来
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --delete --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test1
Topic test1 is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

### 3.5.1 彻底删除主题数据

第一步：

如果需要被删除topic此时正在被程序 produce和consume，则这些生产和消费程序需要停止。

因为如果有程序正在生产或者消费该topic，则该topic的offset信息一致会在broker更新。调用kafka delete命令则无法删除该topic。

同时，需要设置 auto.create.topics.enable = false，默认设置为true。如果设置为true，则produce或者fetch不存在的topic也会自动创建这个topic。这样会给删除topic带来很多意向不到的问题。

所以，这一步很重要，必须设置auto.create.topics.enable = false，并认真把生产和消费程序彻底全部停止。

第二步：

server.properties 设置 **delete.topic.enable=true**

如果没有设置 delete.topic.enable=true，则调用kafka 的delete命令无法真正将topic删除，而是显示（marked for deletion）。

第三步：

调用命令删除topic：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --delete --zookeeper hadoop01:2181,hadoop02:2181,hadoop03:2181 --topic test1
```

#### 第四步：

删除kafka存储目录（server.properties文件log.dirs配置，默认为"/home/log/kafka-logs"）相关topic的数据目录。

注意：如果kafka 有多个 broker，且每个broker 配置了多个数据盘（比如 /data/kafka-logs,/data1/kafka-logs ...），且topic也有多个分区和replica，则需要对所有broker的所有数据盘进行扫描，删除该topic的所有分区数据。

做好第一二三四步后，一般都会将其topic彻底删除。如果还有顽固分子，那就再将kafka保持在zk中的主题信息手动删除。

#### 手动删除kafka在zk中的相关信息步骤：

#### 第五步：

找一台部署了zk的服务器，使用命令：

```
bin/zkCli.sh -server 【zookeeper server:port】
```

登录到zk shell，然后找到topic所在的目录：ls /brokers/topics，找到要删除的topic，然后执行命令：

```
rmr /brokers/topics/ 【topic name】
```

即可，此时topic被彻底删除。

如果topic 是被标记为 marked for deletion，则通过命令 ls /admin/delete\_topics，找到要删除的topic，然后执行命令：

```
rmr /admin/delete_topics/ 【topic name】
```

#### 备注：

网络上很多其它文章还说明，需要删除topic在zk上面的消费节点记录、配置节点记录，比如：

```
rmr /consumers/ 【consumer-group】  
rmr /config/topics/ 【topic name】
```

其实正常情况是不需要进行这两个操作的，如果需要，那都是由于操作不当导致的。比如第一步停止生产和消费程序没有做，第二步没有正确配置。也就是说，正常情况下严格按照“第一步 -- 第五步”的步骤，是一定能够正常删除topic的。

#### 第六步：

完成之后，调用命令：

```
./bin/kafka-topics.sh --list --zookeeper 【zookeeper server:port】
```

查看现在kafka的topic信息。正常情况下删除的topic就不会再显示。

但是，如果还能够查询到删除的topic，则重启zk和kafka即可。

到此为止，topic彻底删除也彻底结束。

## 3.6 生产消息

下面让咱们来生产一些消息到我们的主题吧。使用命令行生产消息使用脚本为`./bin/kafka-console-producer.sh`。

如果broker没有接收到生产者的消息，默认将会重试3次。

```
#查看该脚本的使用帮助
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh
```

```
#使用脚本生产消息
```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic test
>my test message 1
>my test message 2
>this is ok
```

## 3.7 消费消息

3.6步已经向topic test生产咯一些消息，现在咱们可以开始消费topic的消息咯。消费脚本：`./bin/kafka-console-consumer.sh`

```
#消费kafka的test主题消息，--from-beginning：从开始位置开始消费
```

```
[root@hadoop02 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-server hadoop01:9092,hadoop02:9092,hadoop03:9092 --from-beginning --topic test
my test message 1
my test message 2
this is ok
```

```
#可以边生产，边消费咯
```

## 3.8 创建消费者组

消费者组的帮组命令：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-consumer-groups.sh
```

消费时指定消费者组：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-server hadoop01:9092,hadoop02:9092,hadoop03:9092 --from-beginning --topic test --group my_group
```

查看消费者组详情：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-consumer-groups.sh --describe --bootstrap-server hadoop01:9092,hadoop02:9092,hadoop03:9092 --group my_group
```

注：

如果不指定--group,将会随机生成消费者组，并分配到该组里面。

同时消费者指定--group时，如果不存在，则创建，存在不创建。

当使用api创建时，不存在则也会自动创建。

## 3.9 消费者组列表

每一个消费者都属于一个消费者组。如果没有指定，则将会生成一个。

查看consumer group列表有新、旧两种命令，分别查看新版(信息保存在broker中)consumer列表和老版(信息保存在zookeeper中)consumer列表，因而需要区分指定bootstrap--server和zookeeper参数：

#查看3.7步的消费者所属组，使用新版命令

```
[root@hadoop02 kafka_2.11-1.1.1]# ./bin/kafka-consumer-groups.sh --bootstrap-server hadoop01:9092,hadoop02:9092,hadoop03:9092 --list  
Note: This will not show information about old Zookeeper-based consumers.  
console-consumer-12958
```

#查看3.7步的消费者所属组，使用老板版命令。该命令需要zk中记录有才有

```
[root@hadoop03 kafka_2.11-1.1.1]# ./bin/kafka-consumer-groups.sh --list --zookeeper hadoop01:2181  
Note: This will only show information about consumers that use ZooKeeper (not those using the Java consumer API).
```

## 3.10 删除消费者组

```
#删除不存在的组将会报不存在错误，老板命令要求zk中存在
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-consumer-groups.sh --delete --
zookeeper hadoop01:2181,hadoop02:2181,hadoop03:2181 --group testgroup

#删除存在的就不会报错，新版命令
[root@hadoop03 kafka_2.11-1.1.1]# ./bin/kafka-consumer-groups.sh --delete --
bootstrap-server hadoop01:9092,hadoop02:9092,hadoop03:9092 --group console-
consumer-12958
Note: This will not show information about old Zookeeper-based consumers.
Deletion of requested consumer groups ('console-consumer-12958') was
successful.
```

### 注意:

delete消费组时，删除只有在分组元数据存储在zookeeper的才可用。

当使用新消费者API（broker协调处理分区和重新平衡）时，当该组的最后一个提交的偏移到期时，该组被删除。

## 3.11 查看消费位置

可以通过脚本查看其消费位置，本质是调用kafka中的一个类。使用脚本为:[./bin/kafka-run-class.sh](#)

```
#0.9.x版本的查看消费位置
[root@hadoop02 kafka_2.11-1.1.1]# ./bin/kafka-run-class.sh
kafka.tools.ConsumerOffsetChecker --bootstrap-server
hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic test

#随便开启一个消费者(不重复演示开启消费者)，找到其消费id，即消费组。查看其消费位置信息。如果消
费者组是没被启用，将不会显示CONSUMER-ID 及以后的信息
[root@hadoop03 kafka_2.11-1.1.1]# ./bin/kafka-consumer-groups.sh --bootstrap-
server hadoop01:9092,hadoop02:9092,hadoop03:9092 --describe --group console-
consumer-3904
Note: This will not show information about old Zookeeper-based consumers.

TOPIC          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
CONSUMER-ID                               HOST          CLIENT-ID
test            0           1              1              0
consumer-1-13dcba9d-0c81-4af5-85d1-5a1151267371 /192.168.216.112 consumer-1
test            1           2              2              0
consumer-1-13dcba9d-0c81-4af5-85d1-5a1151267371 /192.168.216.112 consumer-1
test            2           2              2              0
consumer-1-13dcba9d-0c81-4af5-85d1-5a1151267371 /192.168.216.112 consumer-1
test            3           1              1              0
consumer-1-13dcba9d-0c81-4af5-85d1-5a1151267371 /192.168.216.112 consumer-1
```

### 注意:

1、kafka的0.9.0.0版本后，`kafka.tools.ConsumerOffsetChecker`已经不支持了。你应该使用`kafka.admin.ConsumerGroupCommand`(或`bin/kafka-consumer-groups.sh`脚本)来管理消费者组，包括用新消费者API创建的消费者。

2、如果你使用老的高级消费者并存储分组元数据在zookeeper (即。`offsets.storage=zookeeper`) 通过`--zookeeper`，而不是`bootstrap-server`。

## 3.9 平衡leader

当一个broker停止或崩溃时，这个broker中所有分区的leader将转移给其他副本。这意味着在默认情况下，当这个broker重启之后，它的所有分区都将仅作为follower，不再用于客户端的读写操作。

不过，[Kafka](#)中有一个被称为优先副本(preferred replicas)的概念。如果一个分区有3个副本，且这3个副本的优先级别分别为1, 5, 9，根据优先副本的概念，第一个副本1会作为5和9副本的leader。为了使kafka集群恢复默认的leader，可以选择手动和自动两种平衡方式。

### 手动平衡leader命令

```
[root@hadoop03 kafka_2.11-1.1.1]# ./bin/kafka-preferred-replica-election.sh --zookeeper hadoop01:2181
```

### 自动平衡leader操作

Kafka为我们提供了一个参数，可以使得Kafka集群自动平衡Leader，我们只需要在`server.properties`文件中配置如下设置：

```
auto.leader.rebalance.enable=true
```

注:这个值默认就是打开的。

## 3.10 kafka自带压测命令

### 测试意义

验证台服务器上Kafka写入消息和消费消息的能力，根据测试结果评估当前Kafka集群模式是否满足上亿级别的消息处理能力。

### 测试方法

在服务器上使用Kafka自带的测试脚本，分别模拟10w、100w和1000w的消息写入请求，查看Kafka处理不同数量级的消息数时的处理能力，包括每秒生成消息数、吞吐量、消息延迟时间。Kafka消息吸入创建的topic命名为`test_perf`，使用命令发起消费该topic的请求，查看Kafka消费不同数量级别的消息时的处理能力。

### 测试命令

测试项	压测消息数 (单位:w)	测试命令
写入MQ消息	10	./bin/kafka-producer-perf-test.sh --topic test --num-records 100000 --record-size 1000 --throughput 2000 --producer-props bootstrap.servers=hadoop01:9092,hadoop02:9092,hadoop03:9092
	100	./bin/kafka-producer-perf-test.sh --topic test --num-records 1000000 --record-size 2000 --throughput 5000 --producer-props bootstrap.servers=hadoop01:9092,hadoop02:9092,hadoop03:9092
	1000	./bin/kafka-producer-perf-test.sh --topic test --num-records 10000000 --record-size 2000 --throughput 5000 --producer-props bootstrap.servers=hadoop01:9092,hadoop02:9092,hadoop03:9092
消费MQ消息	10	./bin/kafka-consumer-perf-test.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic test --fetch-size 1048576 --messages 100000 --threads 1
	100	./bin/kafka-consumer-perf-test.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic test --fetch-size 1048576 --messages 1000000 --threads 1
	1000	./bin/kafka-consumer-perf-test.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic test --fetch-size 1048576 --messages 10000000 --threads 1

### kafka-producer-perf-test.sh 脚本命令的参数解析（以100w写入消息为例）：

```
--topic topic名称, 本例为test
--num-records 总共需要发送的消息数, 本例为100000
--record-size 每个记录的字节数, 本例为1000
--throughput 每秒钟发送的记录数, 本例为5000
--producer-props bootstrap.servers=hadoop01:9092,hadoop02:9092,hadoop03:9092
(发送端的配置信息, 本次测试取集群服务器中的一台作为发送端, 可在kafka的config目录, 以该项目为例: /usr/local/kafka_2.11-1.1.1/config; 查看server.properties中配置的zookeeper.connect的值, 默认端口: 9092)
```

### kafka-consumer-perf-test.sh 脚本命令的参数为：

```
--broker-list 指定kafka的链接信息, 本例为hadoop01:9092,hadoop02:9092,hadoop03:9092
--topic 指定topic的名称, 本例为test, 即3.10中写入的消息;
--fetch-size 指定每次fetch的数据的大小, 本例为1048576, 也就是1M
--messages 总共要消费的消息个数, 本例为1000000, 100w
```

### 测试结果

写入10W条的压力测试结果：

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-producer-perf-test.sh --topic test --num-records 100000 --record-size 1000 --throughput 2000 --producer-props bootstrap.servers=hadoop01:9092,hadoop02:9092,hadoop03:9092
9990 records sent, 1997.2 records/sec (1.90 MB/sec), 20.6 ms avg latency,
342.0 max latency.
10089 records sent, 2016.6 records/sec (1.92 MB/sec), 8.3 ms avg latency,
184.0 max latency.
10021 records sent, 2004.2 records/sec (1.91 MB/sec), 5.1 ms avg latency,
179.0 max latency.
9888 records sent, 1362.0 records/sec (1.30 MB/sec), 8.8 ms avg latency,
2317.0 max latency.
14575 records sent, 2913.8 records/sec (2.78 MB/sec), 278.3 ms avg latency,
2979.0 max latency.
9741 records sent, 1923.2 records/sec (1.83 MB/sec), 3.9 ms avg latency, 280.0
max latency.
10456 records sent, 2091.2 records/sec (1.99 MB/sec), 12.0 ms avg latency,
281.0 max latency.
10010 records sent, 2001.6 records/sec (1.91 MB/sec), 57.3 ms avg latency,
1009.0 max latency.
10002 records sent, 2000.4 records/sec (1.91 MB/sec), 2.3 ms avg latency,
135.0 max latency.
100000 records sent, 1998.840672 records/sec (1.91 MB/sec), 52.60 ms avg
latency, 2979.00 ms max latency, 2 ms 50th, 331 ms 95th, 954 ms 99th, 2885 ms
99.9th.
```

消费10W条的压力测试结果：

```
[root@hadoop03 kafka_2.11-1.1.1]# ./bin/kafka-consumer-perf-test.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic test --fetch-size 1048576 --messages 100000 --threads 1
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg,
nMsg.sec, rebalance.time.ms, fetch.time.ms, fetch.MB.sec, fetch.nMsg.sec
2019-11-16 17:00:27:380, 2019-11-16 17:00:33:570, 95.3675, 15.4067, 100006,
16156.0582, 3024, 3166, 30.1224, 31587.4921
```

## 3.11 副本管理

default.replication.factor

每个Partition有几个副本，默认是1，即只有1个副本(即不做备份)。创建时可以指定。但是创建后不能使用alter进行修改。

## 3.12 kafka导入导出数据命令

从控制台写入数据并将其写回控制台是一个容易开始的事，但你可能需要使用其它数据源或者将数据从Kafka导出到其他系统。针对很多这样的系统，你可以使用Kafka Connect来导入或导出数据，而不是写自定义的集成代码。

Kafka Connect是Kafka的一个工具，它可以将数据导入和导出到Kafka。它是一种可扩展工具，通过运行connectors（连接器），使用自定义逻辑来实现与外部系统的交互。在本文中，我们将看到如何使用简单的connectors来运行Kafka Connect，这些connectors可以将文件中的数据导入到Kafka topic中，并可从kafka中topic导出数据到一个文件。

### 场景：

1、数据源是一堆文件，想让咱们将其放到kafka的某个主题中。

2、将某个topic中的数据导出到文件中。

3、更多的数据源系统数据转移。

### 案例思路：

我们需要使用kafka，使用kafka的生产者读取test.txt文件中的内容存储到connect-test的topic中，然后将该topic

中的数据消费存储到test.sink.txt文件中。

### 案例步骤：

1、创建测试种子数据

```
[root@hadoop01 kafka_2.11-1.1.1]# echo -e "foo\nbar" > test.txt
```

接下来，我们将启动两个以独立模式运行的连接器，这意味着它们运行在一个单独的、本地的、专用的进程中。我们提供了三个配置文件作为参数。第一种始终是Kafka连接进程的配置，包含常见的配置，比如要连接的Kafka broker和数据的序列化格式。使用重命名的配置文件来创建连接器，连接器任何其它需要的配置文件需要有唯一的连接器名，连接器类需要用来实例化。

官网案例格式(如下命令不需要在咱们服务器上执行)

```
> bin/connect-standalone.sh config/connect-standalone.properties  
config/connect-file-source.properties config/connect-file-sink.properties
```

这些案例配置文件，包含kafka，使用您先前启动的默认本地集群配置，并创建两个连接器：

第一个是源(source)连接器，负责从输入文件中按行读并将生成每一行到kafka的topic中。

第二个是下沉(sink)连接器，负责从kafka的topic中读消息并将其按行生成到输出文件中。

2、配置connect-standalone.properties

```
[root@hadoop01 kafka_2.11-1.1.1]# vi ./config/connect-standalone.properties
```

修改内容如下(后面有注释的修改过)：

```
# Licensed to the Apache Software Foundation (ASF) under one or more  
# contributor license agreements. See the NOTICE file distributed with
```

```
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License. You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# These are defaults. This file just demonstrates how to override some
settings.

bootstrap.servers=hadoop01:9092,hadoop02:9092,hadoop03:9092      #修改成自己的kafka
集群地址

# The converters specify the format of data in Kafka and how to translate it
into Connect data. Every Connect user will
# need to configure these based on the format they want their data in when
loaded from or stored into Kafka
key.converter=org.apache.kafka.connect.json.JsonConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
# Converter-specific settings can be passed in by prefixing the Converter's
setting with the converter we want to apply
# it to
key.converter.schemas.enable=true
value.converter.schemas.enable=true

# The internal converter used for offsets and config data is configurable and
must be specified, but most users will
# always want to use the built-in default. Offset and config data is never
visible outside of Kafka Connect in this format.
internal.key.converter=org.apache.kafka.connect.json.JsonConverter
internal.value.converter=org.apache.kafka.connect.json.JsonConverter
internal.key.converter.schemas.enable=false
internal.value.converter.schemas.enable=false

offset.storage.file.filename=/tmp/connect.offsets      #偏移量存储目录
# Flush much faster than normal, which is useful for testing/debugging
offset.flush.interval.ms=10000      #偏移量刷新间隔, 越小越有利于测试和调试

# Set to a list of filesystem paths separated by commas (,) to enable class
loading isolation for plugins
# (connectors, converters, transformations). The list should consist of top
level directories that include
# any combination of:
```

```
# a) directories immediately containing jars with plugins and their
dependencies
# b) uber-jars with plugins and their dependencies
# c) directories immediately containing the package directory structure of
classes of plugins and their dependencies
# Note: symlinks will be followed to discover dependencies or plugins.
# Examples:
#
plugin.path=/usr/local/share/java,/usr/local/share/kafka/plugins,/opt/connecto
rs,
#plugin.path=
```

### 3、配置connect-file-source.properties

```
[root@hadoop01 kafka_2.11-1.1.1]# vi ./config/connect-file-source.properties
修改内容如下(后面有注释的修改过):
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements. See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License. You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=/usr/local/kafka_2.11-1.1.1/test.txt    #指定生产者将要用于生产的数据文件
topic=connect-test    #将test.txt文件按行读出来后，然后存储到connect-test的topic中。没有该主题会自动创建
```

### 4、配置connect-file-sink.properties

```
[root@hadoop01 kafka_2.11-1.1.1]# vi ./config/connect-file-sink.properties
修改内容如下(后面有注释的修改过):
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements. See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License. You may obtain a copy of the License at
```

```
#  
#      http://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.  
  
name=local-file-sink  
connector.class=FileStreamSink  
tasks.max=1  
file=/usr/local/kafka_2.11-1.1.1/test.sink.txt    #消费的信息输出到该文件中  
topics=connect-test  #需要和第3步中的topic一样，否则不能正确消费
```

## 5、测试(开始导入导出)

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/connect-standalone.sh  
../config/connect-standalone.properties ../config/connect-file-source.properties  
../config/connect-file-sink.properties
```

## 6、查看是否成功创建主题(前提是沒有connect-test的topic)

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --list --zookeeper  
hadoop01:2181  
__consumer_offsets  
connect-test      #该主题就是自动创建的  
test  
test_perf
```

## 7、消费connect-test主题的内容

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-  
server hadoop01:9092 --from-beginning --topic connect-test  
{ "schema":{ "type": "string", "optional":false}, "payload": "foo" }  #文件中的两行内容  
已经被消费  
{ "schema":{ "type": "string", "optional":false}, "payload": "bar" }
```

## 8、检测输出文件存在性及内容

```
[root@hadoop01 kafka_2.11-1.1.1]# cat ./test.sink.txt  
foo      #数据内容能对应上，文件肯定存在  
bar
```

到此，整个导入导出数据案例完成。

## 9、注意事项

a、如果topic=connect-test不是个干净的topic，即有历史数据并且数据格式不是json格式，则在下一步消费数据到文件会报如下异常。

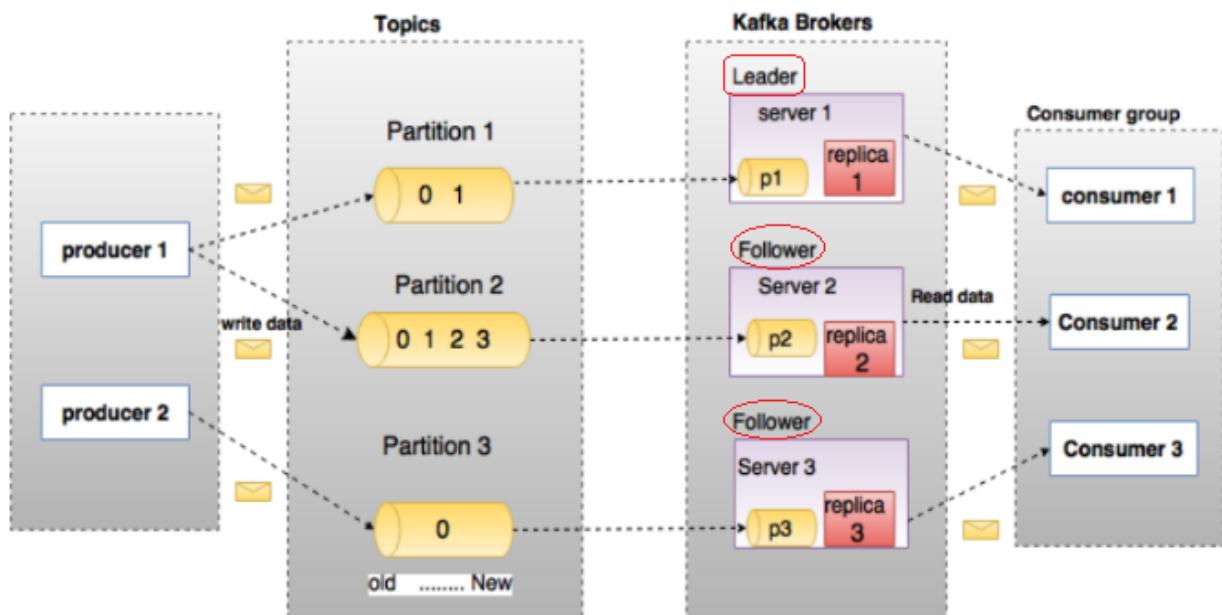
```
JsonParseException: Unrecognized token 'xxx': was expecting ('true', 'false' or 'null')
```

b、如果启动过一次 ./bin/connect-standalone.sh ./config/connect-standalone.properties ... 然后我们这里又更改了connect-file-source.properties里的topic名称，配置成了新的topic名称则需要删除connect.offsets即可，步骤1中的配置里可以找到:offset.storage.file.filename。

## 第四章 kafka的架构

### 4.1 kafka的术语介绍

在介绍kafka的架构前，我们先通过下面这张图了解下kafka相关术语。



**broker**、**topic**、**producer**、**partition**、**consumer**、**consumergroup**、**replica**、**Segment**等相關概念请参考：第一节的1.5 kafka的核心概念。

#### Leader

每个partition有多个副本，其中有且仅有一个作为Leader，Leader是当前负责数据的读写的partition。

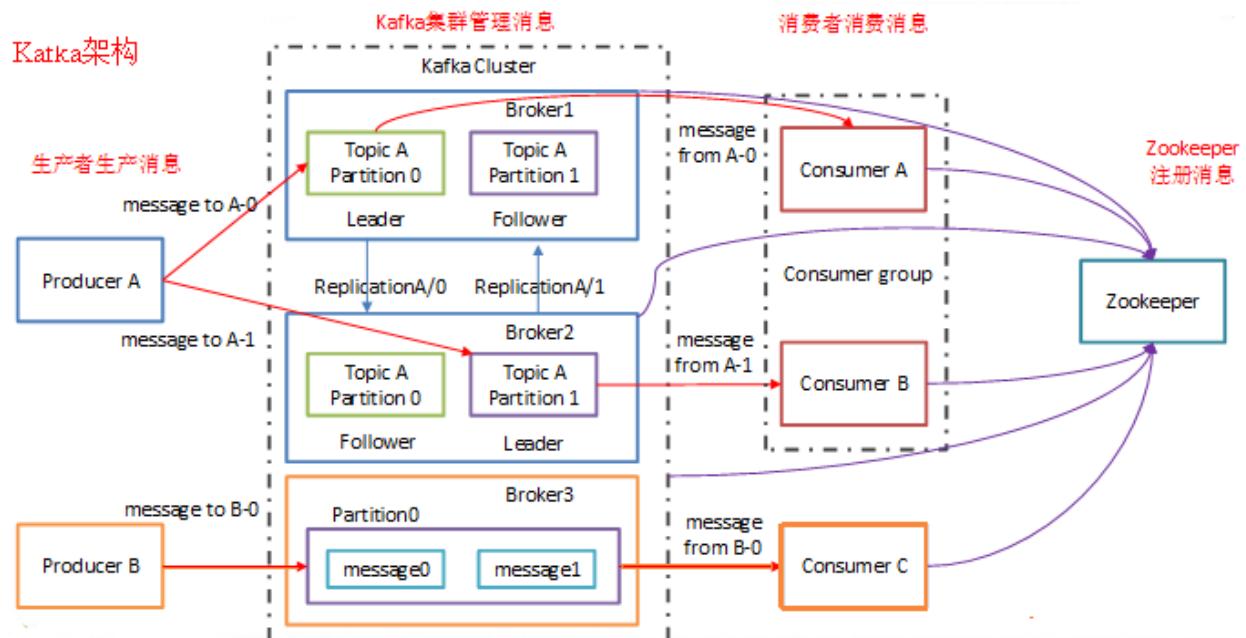
#### Follower

Follower跟随Leader，所有写请求都通过Leader路由，数据变更会广播给所有Follower，Follower与Leader保持数据同步。如果Leader失效，则从Follower中选举出一个新的Leader。当Follower与Leader挂掉、卡住或者同步太慢，leader会把这个follower从“in sync replicas” (ISR) 列表中删除，重新创建一个Follower。

#### Offset

kafka的存储文件都是按照offset.kafka来命名，用offset做名字的好处是方便查找。例如你想找位于2049的位置，只要找到2048.kafka的文件即可。当然the first offset就是00000000000.kafka

## 4.2 kafka的架构



通常，一个典型的Kafka集群中包含若干Producer（可以是web前端产生的Page View，或者是服务器日志，系统CPU、Memory等），若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高），若干Consumer Group，以及一个Zookeeper集群。Kafka通过Zookeeper管理集群配置，选举leader，以及在Consumer Group发生变化时进行rebalance。Producer使用push模式将消息发布到broker，Consumer使用pull模式从broker订阅并消费消息。

## 4.3 kafka的分布式模型

kafka分布式主要是指分区被分布在多台server上，同时每个分区都有leader和follower(不是必须)，即老大和小弟的角色，这儿是老大负责处理，小弟负责同步，小弟可以变成老大，形成分布式模型。

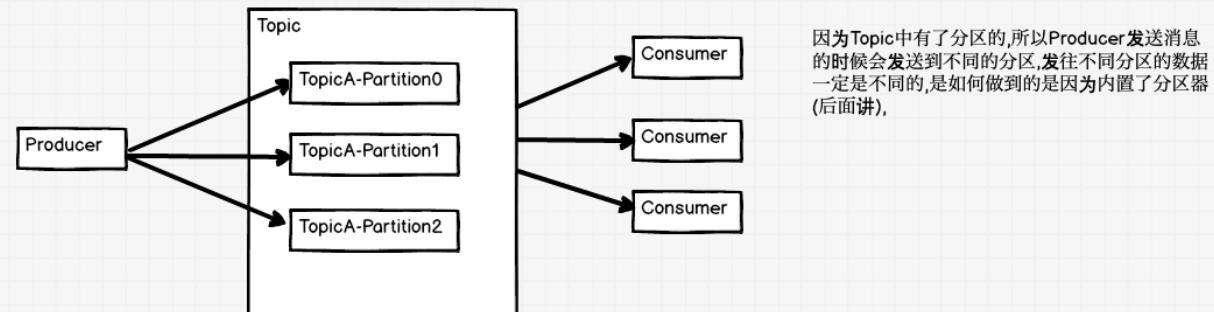
kafka的分区日志分布在kafka集群的服务器上，每一个服务器处理数据和共享分区请求。每一个分区是被复制到一系列配置好的服务器上来进行容错。

每个分区有一个server节点来作为leader和零个或者多个server节点来作为followers。leader处理指定分区的所有读写请求，同时follower被动复制leader。如果leader失败，followers中的一个将会自动地变成一个新的leader。每一个服务器都能作为分区的一个leader和作为其它分区的follower，因此kafka集群能被很好地平衡。

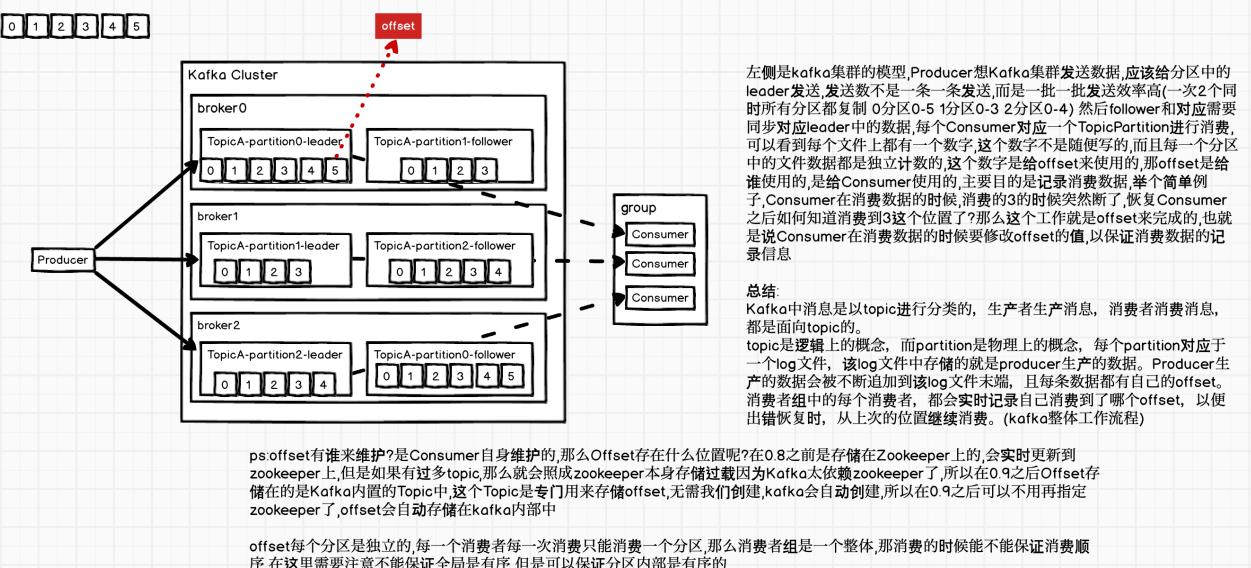
如上信息参考官网:[http://kafka.apache.org/intro.html#intro\\_distribution](http://kafka.apache.org/intro.html#intro_distribution)



为了方便扩展提高吞吐量,一个topic分为多个Partition



## 4.4 kafka的文件存储



- 在kafka集群中,分单个broker和多个broker。每个broker中有多个topic, topic数量可以自己设定。在每个topic中又有0到多个partition, 每个partition为一个分区。kafka分区命名规则为topic的名称+有序序号,这个序号从0开始依次增加。
- 每个partition中有多个segment file。创建分区时,默认会生成一个segment file, kafka默认每个segment file的大小是1G。当生产者往partition中存储数据时,内存中存不下了,就会往segment file里面刷新。在存储数据时,会先生成一个segment file,当这个segment file到1G之后,再生成第二个segment file以此类推。每个segment file对应两个文件,分别是.log结尾的数据文件和.index结尾的索引文件。在服务器上,每个partition是一个目录,每个segment是分区目录下的一个文件。
- 每个segment file也有自己的命名规则,每个名字有20个字符,不够用0填充。每个名字从0开始命名,下一个segment file文件的名字就是,上一个segment file中最后一条消息的索引值。在.index文件中,存储的是key-value格式的, key代表在.log中按顺序开始第n条消息, value代表该消息的位置偏移。但是在.index中不是对每条消息都做记录,它是每隔一些消息记录一次,避免占用太多内存。即使消息不在index记录中,在已有的记录中查找,范围也大大缩小了。

## 4.5 topic中的partition

### 为什么要分区

可以想象，如果一个topic就一个分区，要是这个分区有1T数据，那么kafka就想把大文件划分到更多的目录来管理，这就是kafka所谓的分区。

### 分区的好处

- 方便在集群中扩展。因为一个topic由一个或者多个partition构成，而每个节点中通常可以存储多个partition，这样就方便分区存储于移动，也就增加其扩展性。同时也可以增加其topic的数据量。
- 可以提高并发。因为一个主题多个partition，而每个主题读写数据时，其实就是读写不同的partition，所以增加其并发。

### 单节点partition的存储分布

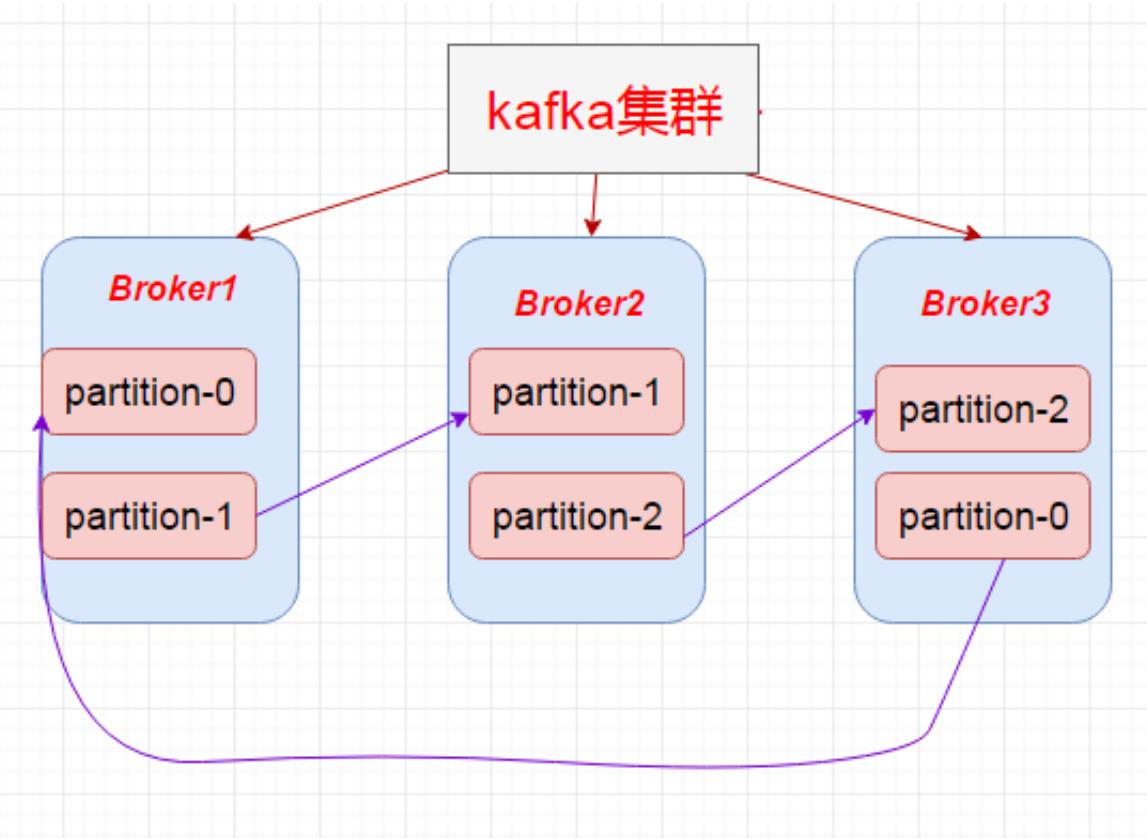
Kafka集群只有一个broker，/var/log/kafka-log为数据文件存储根目录，在Kafka broker中server.properties文件配置(参数log.dirs=/var/log/kafka-log)，例如创建2个topic名称分别为test-1、test-2, partitions数量都为partitions=4

存储路径和目录规则为：

```
|--test-1-0  
|--test-1-1  
|--test-1-2  
|--test-1-3  
|--test-2-0  
|--test-2-1  
|--test-2-2  
|--test-2-3
```

在Kafka文件存储中，同一个topic下有多个不同partition，**每个partition为一个目录**，partiton命名规则为:**topic名称+分区编号(有序)**，第一个partiton序号从0开始，序号最大值为partitions数量减1。

### 多节点partition存储分布



#### 4.5.1 分区分配策略

1. 将所有broker (n个) 和partition排序
2. 将第*i*个Partition分配到第 (*i* mode n) 个broker上

##### 分区策略举例

test3的topic, 4个分区, 2个副本。

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --describe --zookeeper hadoop01:2181 --topic test3
Topic:test3      PartitionCount:4          ReplicationFactor:2        Configs:
          Topic: test3      Partition: 0      Leader: 1        Replicas: 1,3    Isr:
          1,3
          Topic: test3      Partition: 1      Leader: 2        Replicas: 2,1    Isr:
          1,2
          Topic: test3      Partition: 2      Leader: 3        Replicas: 3,2    Isr:
          2,3
          Topic: test3      Partition: 3      Leader: 1        Replicas: 1,2    Isr:
          1,2
```

第1个Partition分配到第 (1 mode 3) = 1个broker上  
 第2个Partition分配到第 (2 mode 3) = 2个broker上  
 第3个Partition分配到第 (3 mode 3) = 3个broker上。  
 第4个Partition分配到第 (4 mode 3) = 1个broker上

#### 4.5.2 副本分配策略

- 在Kafka集群中, 每个Broker都有均等分配Partition的Leader机会。

- 上述图Broker Partition中，箭头指向为副本，以Partition-0为例：broker3中partition-0为Leader，Broker1中Partition-0为副本。
- 上述图中每个Broker(按照BrokerId有序)依次分配主Partition，下一个Broker为副本，如此循环迭代分配，多副本都遵循此规则。

### 副本分配算法：

- 将所有N Broker和待分配的i个Partition排序。
- 将第i个Partition分配到第( $i \bmod n$ )个Broker上，第1个副本也存储在该broker上。
- 将第i个Partition的第j个副本分配到第( $(i + j) \bmod n$ )个Broker上，从第2个副本开始。

### 分区及副本分配举例

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --describe --zookeeper
hadoop01:2181 --topic test1
```

```
Topic:test1      PartitionCount:3      ReplicationFactor:2
```

#### Configs:

Topic: test1	Partition: 0	Leader: 3	Replicas: 3,1	Isr: 3,1
Topic: test1	Partition: 1	Leader: 1	Replicas: 1,2	Isr: 1,2
Topic: test1	Partition: 2	Leader: 2	Replicas: 2,3	Isr: 2,3

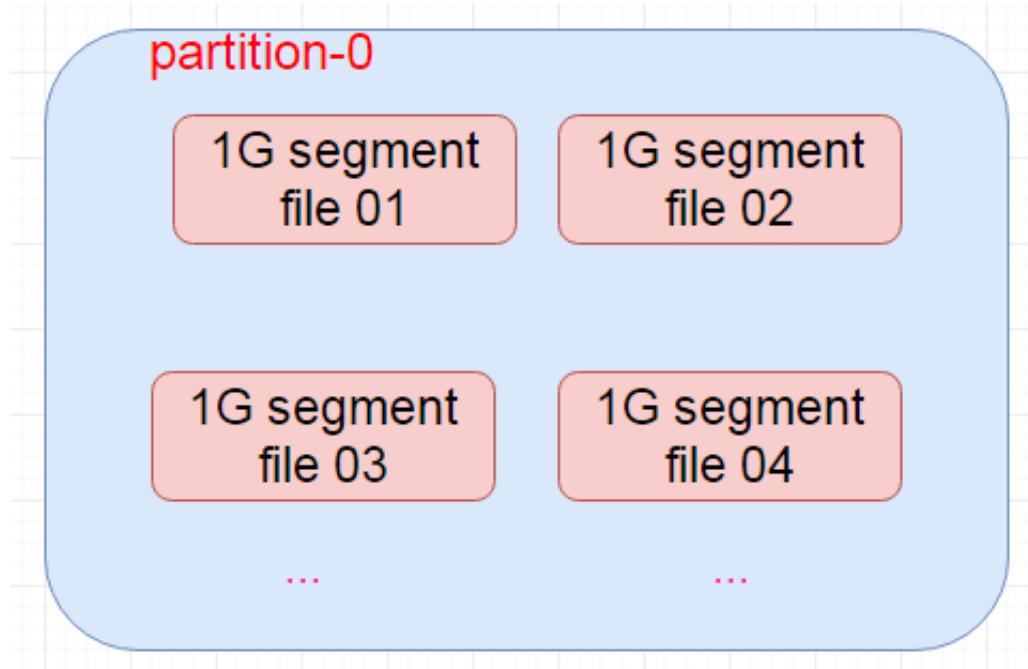
第0个partition分配到第( $0 \% 3$ )个broker上，即分配到第1个broker上。0分区的第一个副本就在该broker上；0分区的第二个副本在( $(0+2) \% 3$ )=2个broker

第1个partition分配到第( $1 \% 3$ )个broker上，即分配到第2个broker上。1分区的第一个副本就在该broker上；1分区的第二个副本在( $(1+2) \% 3$ )=0个broker

第2个partition分配到第( $2 \% 3$ )个broker上，即分配到第3个broker上。2分区的第一个副本就在该broker上；2分区的第二个副本在( $(2+2) \% 3$ )=1个broker

## 4.6 partition中文件存储

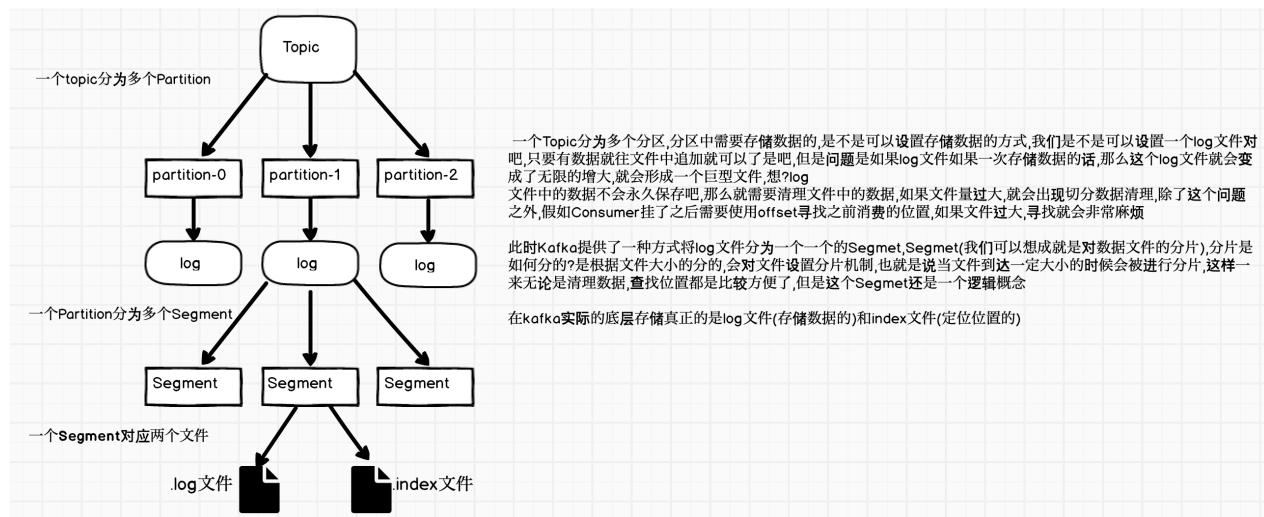
如下是一个partition-0的一个存储示意图。

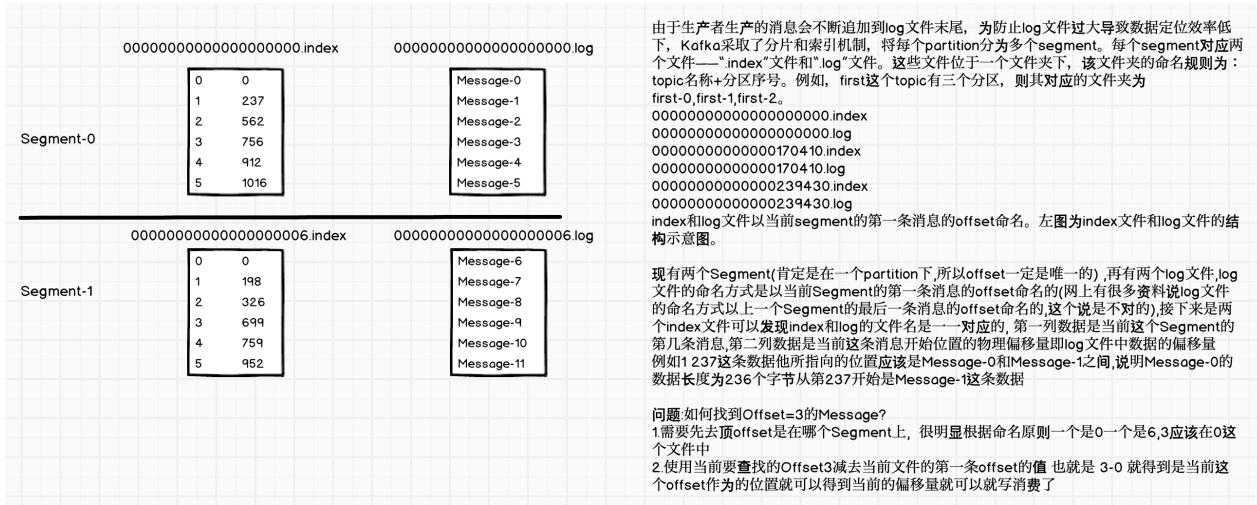


- 每个分区一个目录，该目录中是一堆segment file(默认一个segment是1G)，该目录和file都是物理存储于磁盘。
- 每个partition(目录)相当于一个巨型文件被平均分配到多个大小相等segment(段)数据文件中。但每个段segment file消息数量不一定相等，这种特性方便old segment file快速被删除。
- 每个partition只需要支持顺序读写就行了，segment文件生命周期由服务端配置参数决定。
- 这样做的好处就是能快速删除无用文件，有效提高磁盘利用率。

## 4.7 kafka分区中的segement

通过4.6节我们知道Kafka文件系统是以partition方式存储，下面深入分析partition中segment file组成和物理结构。





通过上面两张图，我们已经知道topic、partition、segment、.log、.index等文件的关系，下面深入看看segment相关组成原理。

### segment file组成：

由2大部分组成，分别为index file和log file(即数据文件)，这2个文件一一对应，成对出现，后缀".index"和".log"分别表示为segment索引文件、数据文件。

### segment文件命名规则：

partition全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值。数值最大为64位long大小，20位数字字符长度，不够的左边用0填充。

### 验证：

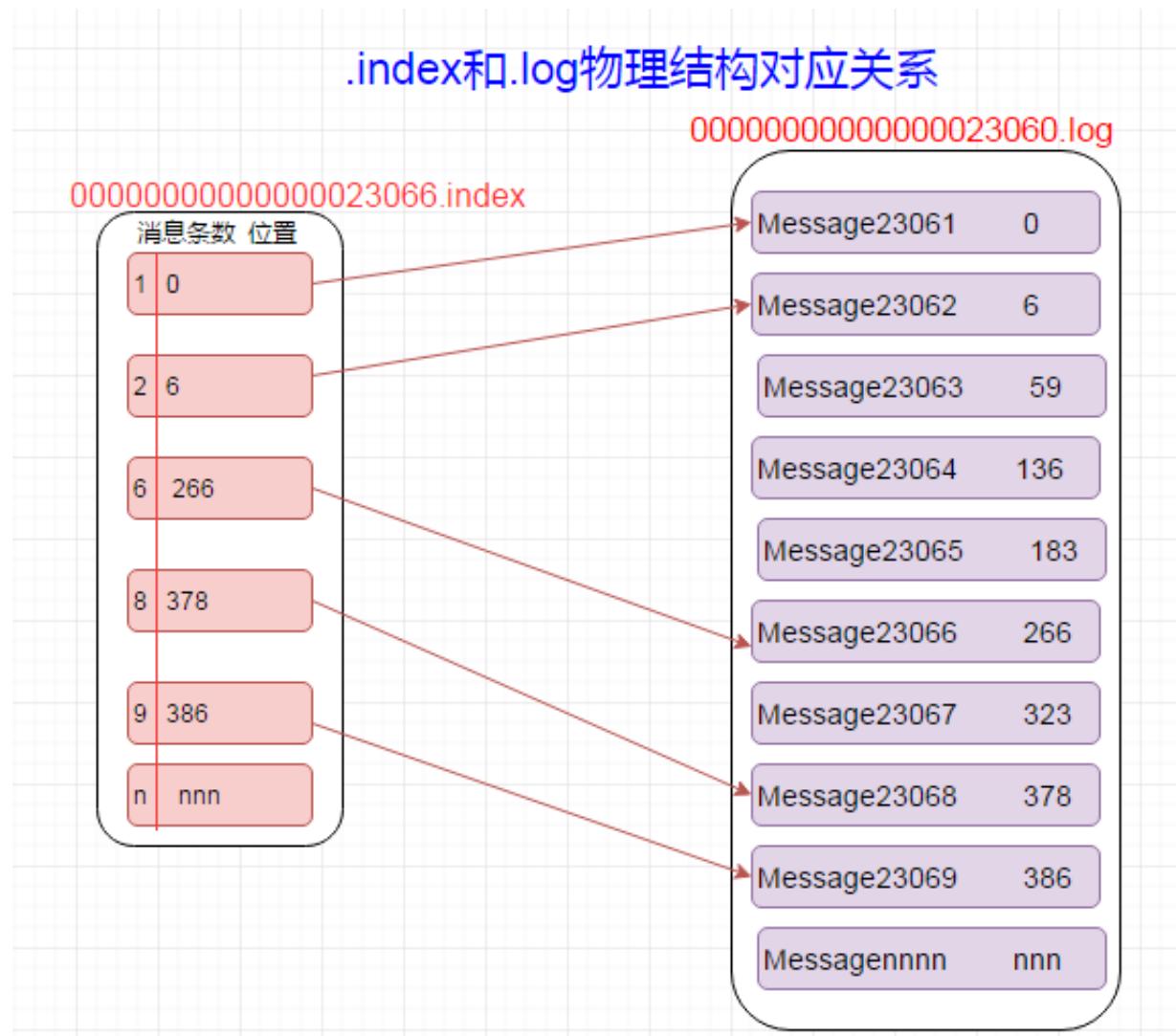
创建一个topic为test5包含1 partition，设置每个segment大小为1G，并启动producer向Kafka broker写入大量数据。

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --create --zookeeper hadoop01:2181,hadoop02:2181,hadoop03:2181 --replication-factor 1 --partitions 1 --topic test5
Created topic "test5".
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic test5
>nihao beijing
>nihao qianfeng
>124
>123456789
>098765
>111
>222
>666
>999
>laowang
>goudan
不断写很多.....
```

### 查看segment文件列表：

```
[root@hadoop03 kafka_2.11-1.1.1]# ll /var/log/kafka-log/test5-0/ #查看分区目录
total 8
-rw-r--r--. 1 root root 10485760 Nov 21 10:50 00000000000000000000.index
#segment文件索引文件
-rw-r--r--. 1 root root 1073761826 Nov 21 10:53 00000000000000000000.log
#segemnt的log文件
-rw-r--r--. 1 root root 10485760 Nov 21 10:50 000000000000000023060.index
-rw-r--r--. 1 root root 892 Nov 21 10:53 000000000000000023060.log
-rw-r--r--. 1 root root 10485756 Nov 21 10:50 00000000000000003268.timeindex
-rw-r--r--. 1 root root 8 Nov 21 10:52 leader-epoch-checkpoint
```

**segment文件的物理结构：**



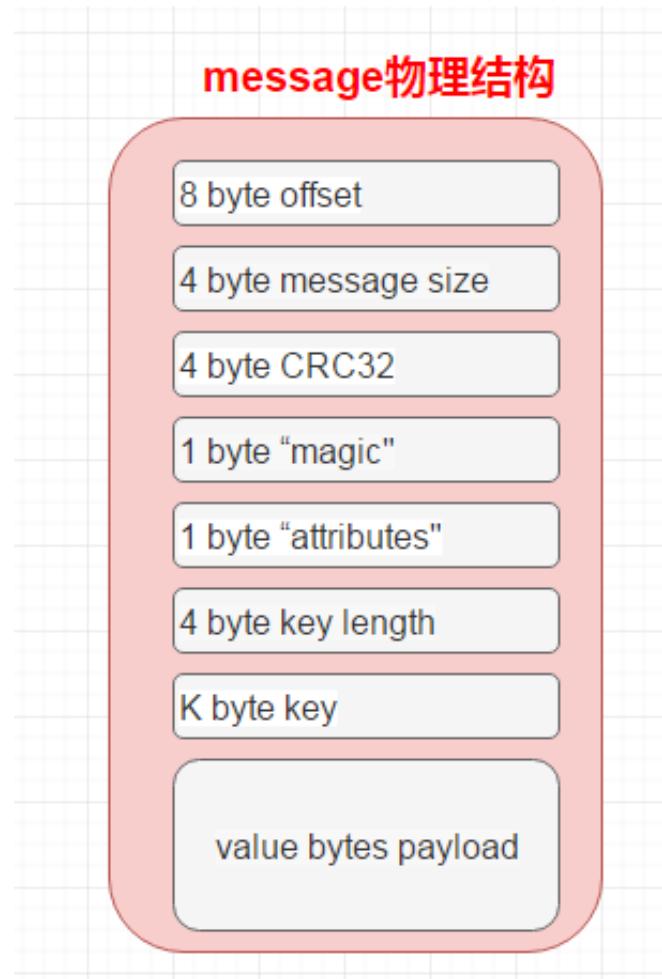
如上图，索引文件存储大量元数据，数据文件存储大量消息，索引文件中元数据指向对应数据文件中 message 的物理偏移地址。

### 举例

上述索引文件中元数据 6 266 为例，依次在数据文件中表示第6个message(在全局partiton表示第20366个message)、以及该消息的物理偏移地址为266。

### message物理结

一个segment data file由许多message组成，一个message物理结构具体如下：



具体参数详解：

关键字	解释说明
8 byte offset	在partition(分区)内的每条消息都有一个有序的id号，这个id号被称为偏移(offset)，它可以唯一确定每条消息在partition(分区)内的位置。即offset表示partiion的第多少message
4 byte message size	message大小
4 byte CRC32	用crc32校验message
1 byte "magic"	表示本次发布Kafka服务程序协议版本号
1 byte "attributes"	表示为独立版本、或标识压缩类型、或编码类型。
4 byte key length	表示key的长度,当key为-1时， K byte key字段不填
K byte key	可选
value bytes payload	表示实际消息数据。

## 4.8 kafka中消息查找流程

### 举例

查找offset=23066的message，需要通过如下2个步骤查找：

#### 第一步查找segment file

```
00000000000000000000.index
00000000000000000000.log
000000000000000023060.index
000000000000000023060.log
```

根据.index和.log物理结构对应关系图可知，其中00000000000000000000.index表示最开始的文件，起始偏移量(offset)为0.第二个文件000000000000000023060.index的消息量起始偏移量为23060 = 23059 + 1.同样，其他后续文件依次类推，以起始偏移量命名并排序这些文件，只要根据offset 二分查找文件列表，就可以快速定位到具体文件。

当offset=23066时定位到000000000000000023060.index和log文件。

#### 第二步通过segment file查找message

通过第一步定位到segment file，当offset=23066时，依次定位到000000000000000023060.index的元数据物理位置和000000000000000023060.log的物理偏移地址，然后再通过000000000000000023060.log顺序查找直到offset=23066为止。

segment index file采取稀疏索引存储方式，即<偏移量、位置>，它减少索引文件大小，通过map可以直接内存操作，稀疏索引为数据文件的每个对应message设置一个元数据指针，它比稠密索引节省了更多的存储空间，但查找起来需要消耗更多的时间。

## 4.9 kafka高效读写数据

### 1) 顺序写磁盘

Kafka的producer生产数据，要写入到log文件中，写的过程是一直追加到文件末端，为顺序写。官网有数据表明，同样的磁盘，顺序写能到到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

是参照Linux系统完成的

Linux总会把系统中还没被应用使用的内存挪来给Page Cache

当写操作发生时，它只是将数据写入Page Cache中，并将该页置上dirty标志。

当读操作发生时，它会首先在Page Cache中查找，如果有就直接返回了，没有的话就会从磁盘读取文件写入Page Cache再读取。

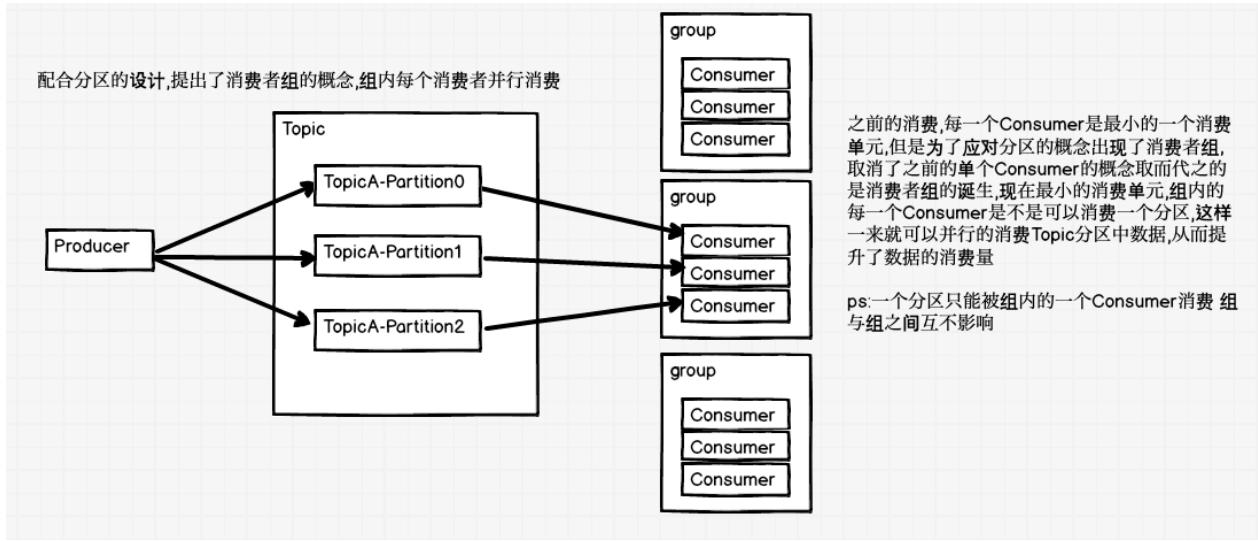
可见，只要生产者与消费者的速度相差不大，消费者会直接读取之前生产者写入Page Cache的数据，大家在内存里完成接力，根本没有磁盘访问。

而比起在内存中维护一份消息数据的传统做法，这既不会重复浪费一倍的内存，Page Cache又不需要GC(可以放心使用60G内存了)，而且即使Kafka重启了，Page Cache还依然在。

## 4.10 Consumer Group架构

consumer group是kafka提供的可扩展且具有容错性的消费者机制。既然是一个组，那么组内必然可以有多个消费者或消费者实例(consumer instance)，它们共享一个公共的ID，即group ID。组内的所有消费者协调在一起消费订阅主题(subscribed topics)的所有分区(partition)。当然，每个分区只能由同一个消费组内的一个consumer来消费。理解consumer group记住下面这三个特性就好了：

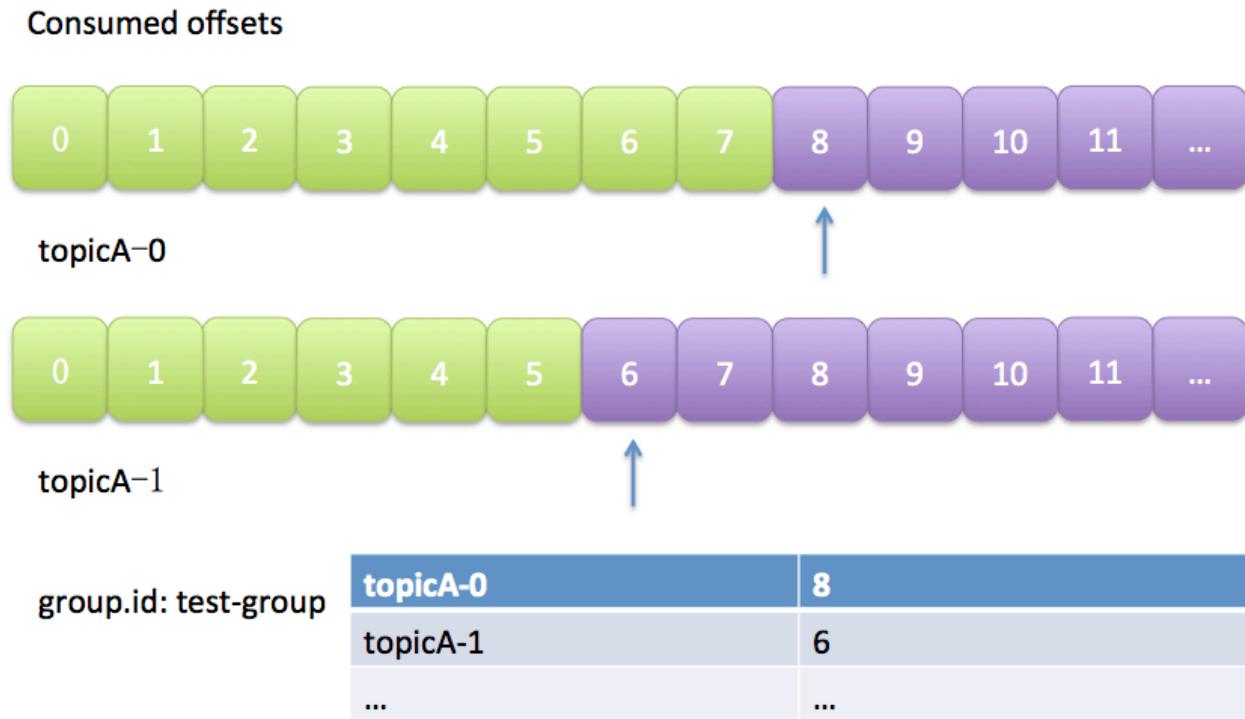
- consumer group下可以有一个或多个consumer instance，consumer instance可以是一个进程，也可以是一个线程
- group.id是一个字符串，唯一标识一个consumer group
- consumer group下订阅的topic下的每个分区只能分配给某个group下的一个consumer(当然该分区还可以被分配给其他group)



## 4.11 offset的维护

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

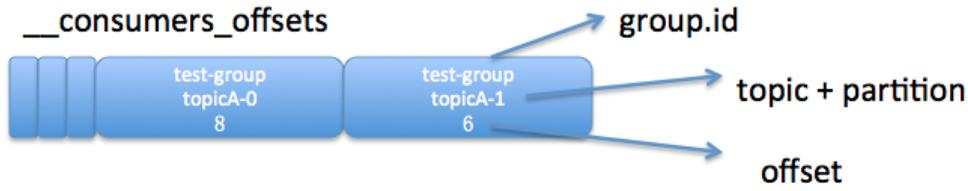
Kafka默认是定期帮你自动提交位移的(enable.auto.commit = true), 你当然可以选择手动提交位移实现自己控制。另外kafka会定期把group消费情况保存起来, 做成一个offset map, 如下图所示:



上图表示, 有一个test-group的消费者组, 该组消费两个分区分别为topicA-0和topicA-1, 对topicA-0分区消费到offset为8这个位置, 而对topicA-1分区消费到offset为6这个位置。

Kafka 0.9版本之前, consumer默认将offset保存在Zookeeper中, zk中的目录结构是: /consumers/[group.id](<http://group.id/>)/offsets//。但是zookeeper其实并不适合进行大批量的读写操作, 尤其是写操作。因此从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。该topic的格式大概如下:

group.id:分组id, 唯一。



high level 和low level

- 将zookeeper维护offset的方式称为 low level API
- 将kafka broker 维护offset的方式称为high level API

使用high level API 更新offset具体设置

consumer中设置可以在代码中设这个属性

- 自动提交，设置`enable.auto.commit=true`, 更新的频率根据参数【`auto.commit.interval.ms`】来定。这种方式也被称为【`at most once`】，fetch到消息后就可以更新offset，无论是否消费成功。默认就是`true`
- 手动提交，设置`enable.auto.commit=false`, 这种方式称为【`at least once`】。fetch到消息后，等消费完成再调用方法【`consumer.commitSync()`】，手动更新offset；如果消费失败，则offset也不会更新，此条消息会被重复消费一次。

## 4.12 kafka中push和pull

一个较早问题是我们应该考虑是消费者从broker中pull数据还是broker将数据push给消费者。kakfa遵守传统设计和借鉴很多消息系统，这儿kafka选择producer向broker去push消息，并由consumer从broker pull消息。一些ogging-centric system，比如Facebook的Scribe和Cloudera的Flume,采用非常不同的push模式。事实上，push模式和pull模式各有优劣。

**push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。** push模式的目标是尽可能以最快速度传递消息，但是这样很容易造成consumer来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而pull模式则可以根据consumer的消费能力以适当的速率消费消息。

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

ps:`timeout`官方案例是100毫秒

## 4.13 kafka中数据发送保障

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。



## 副本数据同步策略

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

Kafka选择了第二种方案, 原因如下:

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

2.虽然第二种方案的网络延迟会比较高, 但网络延迟对Kafka的影响较小。

## 2) ISR

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

Leader维护了一个动态的in-sync replica set (ISR), 意为和leader保持同步的follower集合。当ISR中的follower完成数据的同步之后, leader就会给follower发送ack。如果follower长时间未向leader同步数据, 则该follower将被踢出ISR, 该时间阈值由\**replica.lag.time.max.ms*\*参数设定。Leader发生故障之后, 就会从ISR中选举新的leader。

注:

- 生产者发送到特定主题分区的消息是将按照发送的顺序来追加。也就是说, 如果消息M1和消息M2由相同的生产者发送, 并且M1是先发送的, 那么M1的偏移量将比M2低, 并出现在日志的前面。
- 消费者是按照存储在日志中记录顺序来查询消息。
- 对于具有n个副本的主题, 我们将容忍最多N-1个服务器失败故障, 从而不会丢失提交到日志的任何消息记录。

## 4.14 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。

#### ACK应答机制

ack可以在kafka中进行配置acks参数配置

acks = 0

producer不等的borker的ack,这样的操作是一种低延迟的操作,broker一接受到还没有写出成功就返回,当broker故障的时候可能会出现数据丢失(相当于异步发送)



acks=1

producer等待broker的ack,partition的leader将数据落地磁盘后返回ack,如果在follower同步成功之前leader故障,这个时候数据会丢失



此时Follower需要去同步消息,但是leader挂了,此时Kafka集群就是重新对leader进选举

leader中数据是没有进行同步的,所以会出现数丢失问题

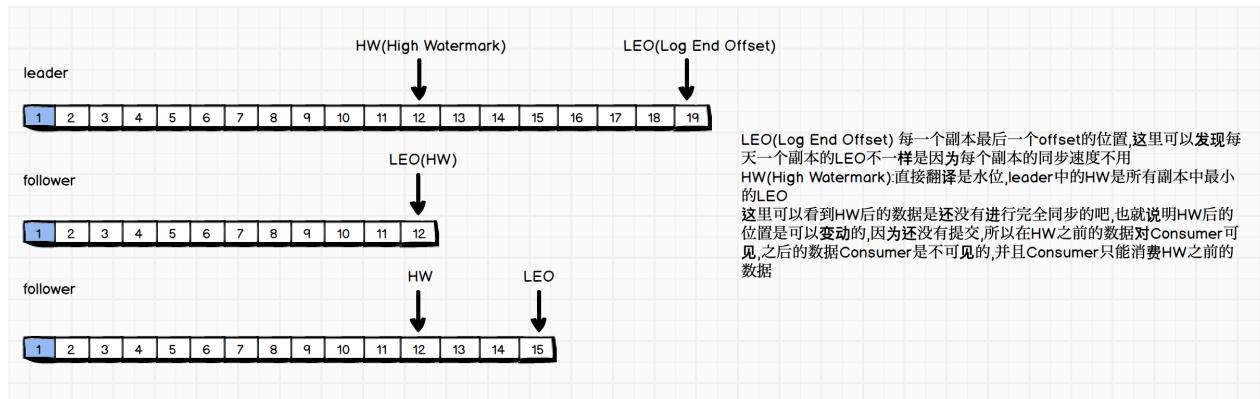
acks=-1(all)

producer等待broker的ack,partition的leader和follower完成全部的数据同步才返回ack(即会触发ISR),但是如果在follower同步完成后, broker发送ack之前,leader挂了,会出现 数据重复



正常Producer向Leader发送Hello信息,收到信息后,落地磁盘成功,等到所有follower同步完成后,此时会向Producer发送ack,就在发送ack的一瞬间Leader挂了,Producer没有接收到ack,Kafka集群会重新选择Leader,但是Producer没有收到ack所以会向新的leader重新再一次发送hello信息,所以此时就会造成数据重复

## 故障处理细节



### (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。\*

## 4.15 Exactly Once(一次正好)语义

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

在0.11版本之后，Kafka引入了幂等性机制（idempotent（id泡深思）），配合acks = -1时的at least once（最少一次）语义，实现了producer到broker的exactly once语义。

#### **\*idempotent + at least once = exactly once\***

使用时，只需将enable.idempotence属性设置为true（在生产者的位置），kafka自动将acks属性设为-1。

ps：幂等性机制是什么意思，幂等简单说1的几次幂都等于1，也就是说一条消息无论发几次都只算一次，无论多少条消息但只实例化一次

kafka完成幂等性其实就是在给消息添加了唯一ID，这个ID的组成是PID（ProducerID）这样保证每一个Producer发送的时候是唯一的，还会为Producer中每天消息添加一个消息ID，也就是说当前Producer中生产的消息会加入Producer的ID和消息ID这样就能保证消息唯一了，这个消息发送到Kafka中的时候会暂时缓存ID，写入数据后没有收到ack，那么会从新发送这个消息，新消息过来的时候会和缓存中ID进行比较如果发现已经存在就不会再次接受了。

#### **详细解析：**

为实现Producer的幂等性，Kafka引入了Producer ID（即PID）和Sequence Number。

PID。每个新的Producer在初始化的时候会被分配一个唯一的PID，这个PID对用户是不可见的。

Sequence Number。（对于每个PID，该Producer发送数据的每个<Topic, Partition>都对应一个从0开始单调递增的Sequence Number）

Kafka可能存在多个生产者，会同时产生消息，但对Kafka来说，只需要保证每个生产者内部的消息幂等就可以了，所有引入了PID来标识不同的生产者。

对于Kafka来说，要解决的是生产者发送消息的幂等问题。也即需要区分每条消息是否重复。

Kafka通过为每条消息增加一个Sequence Number，通过Sequence Number来区分每条消息。每条消息对应一个分区，不同的分区产生的消息不可能重复。所有Sequence Number对应每个分区

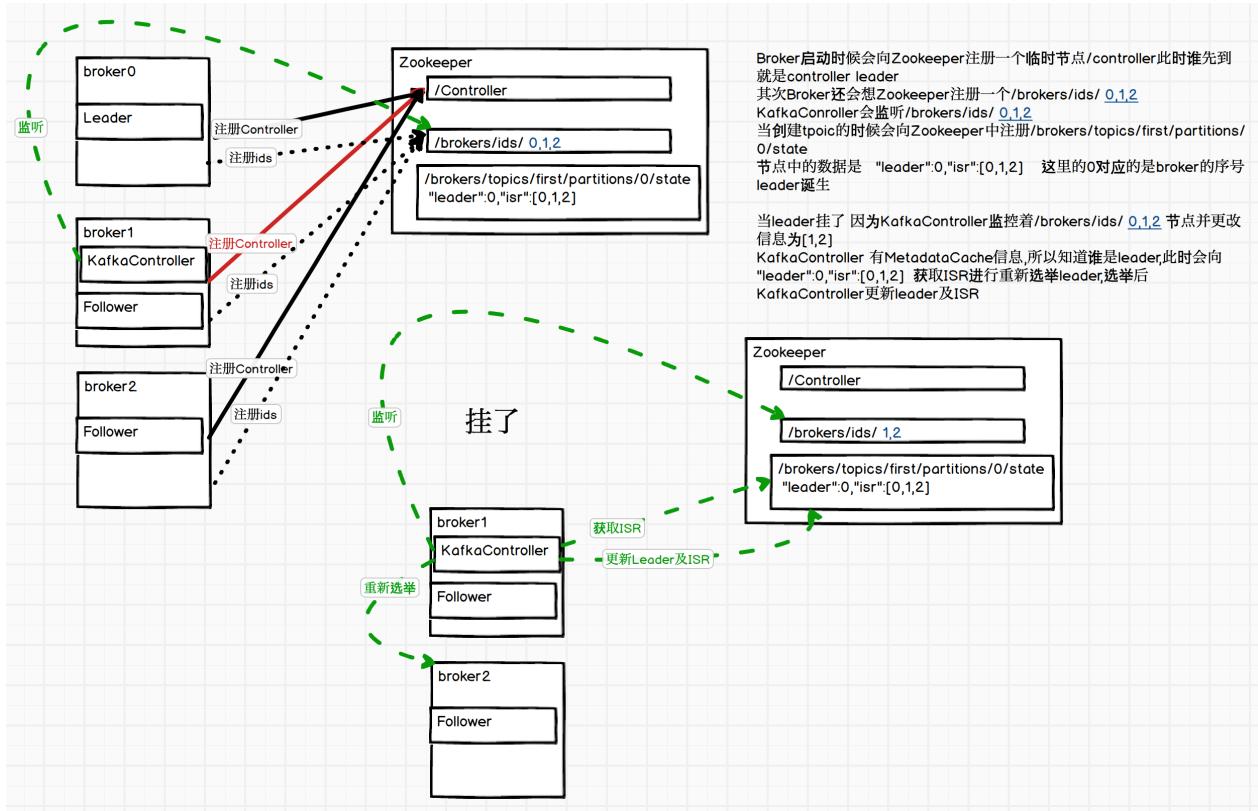
Broker端在缓存中保存了这seq number，对于接收的每条消息，如果其序号比Broker缓存中序号大于1则接受它，否则将其丢弃。这样就可以实现了消息重复提交了。但是，只能保证单个Producer对于同一个<Topic, Partition>的Exactly Once语义。不能保证同一个Producer一个topic不同的partition幂等。

## **4.16 Zookeeper在Kafka中的作用**

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

Controller的管理工作都是依赖于Zookeeper的。

以下为partition和Controller的leader选举过程：



只有KafkaController Leader会向zookeeper上注册Watcher，其他broker几乎不用监听zookeeper的状态变化。

Kafka集群中多个broker，有一个会被选举为controller leader(谁先到就是谁)，负责管理整个集群中分区和副本的状态，比如partition的leader副本故障，由controller负责为该partition重新选举新的leader副本；当检测到ISR列表发生变化，有controller通知集群中所有broker更新其MetadataCache信息；或者增加某个topic分区的时候也会由controller管理分区的重新分配工作

当broker启动的时候，都会创建KafkaController对象，但是集群中只能有一个leader对外提供服务，这些每个节点上的KafkaController会在指定的zookeeper路径下创建临时节点，只有第一个成功创建的节点的KafkaController才可以成为leader，其余的都是follower。当leader故障后，所有的follower会收到通知，再次竞争在该路径下创建节点从而选举新的leader