大数据技术之Kafka

# 第1章 Kafka概述

## 1.2 消息队列（Message Queue）

**1.2.1 传统消息队列的应用场景**

****

**1.2.2 消息队列的两种模式**

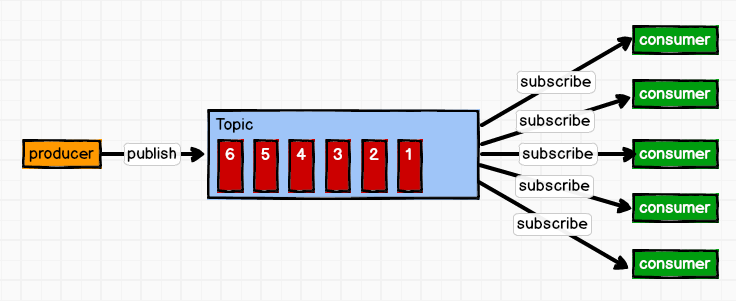
**（1）点对点模式**（一对一，消费者主动拉取数据，消息收到后消息清除）

消息生产者生产消息发送到Queue中，然后消息消费者从Queue中取出并且消费消息。

消息被消费以后，queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。

**（2）发布/订阅模式**（一对多，消费者消费数据之后不会清除消息）

消息生产者（发布）将消息发布到topic中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到topic的消息会被所有订阅者消费。



## 1.1定义

Kafka是一个分布式的基于发布/订阅模式的**消息队列，**主要应用于大数据实时处理领域。

## 1.3 Kafka基础架构



**1）Producer ：**消息生产者，就是向kafka broker发消息的客户端；

**2）Consumer ：**消息消费者，向kafka broker取消息的客户端；

**3）Consumer Group （CG）：**消费者组，由多个consumer组成。**消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个消费者消费；消费者组之间互不影响。**所有的消费者都属于某个消费者组，即**消费者组是逻辑上的一个订阅者**。

**4）Broker ：**一台kafka服务器就是一个broker。一个集群由多个broker组成。一个broker可以容纳多个topic。

**5）Topic ：**可以理解为一个队列，**生产者和消费者面向的都是一个topic**；

**6）Partition：**为了实现扩展性，一个非常大的topic可以分布到多个broker（即服务器）上，**一个topic可以分为多个partition**，每个partition是一个有序的队列；

**7）Replica：**副本，为保证集群中的某个节点发生故障时，该节点上的partition数据不丢失，且kafka仍然能够继续工作，kafka提供了副本机制，一个topic的每个分区都有若干个副本，一个**leader**和若干个**follower**。

**8）leader：**每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是leader。

**9）follower：**每个分区多个副本中的“从”，实时从leader中同步数据，保持和leader数据的同步。leader发生故障时，某个follower会成为新的leader。

# 第2章 Kafka快速入门

## 2.1 安装部署

**2.1.1 集群规划**

hadoop102 hadoop103 hadoop104

zk zk zk

kafka kafka kafka

**2.1.2 jar包下载**

<http://kafka.apache.org/downloads>

**2.1.3 集群部署**

1）解压安装包

[atguigu@hadoop102 software]$ tar -zxvf kafka\_2.11-2.4.1.tgz -C /opt/module/

2）修改解压后的文件名称

[atguigu@hadoop102 module]$ mv kafka\_2.11-2.4.1/ kafka

3）在/opt/module/kafka目录下创建logs文件夹

[atguigu@hadoop102 kafka]$ mkdir logs

4）修改配置文件

[atguigu@hadoop102 kafka]$ cd config/

[atguigu@hadoop102 config]$ vi server.properties

输入以下内容：

#broker的全局唯一编号，不能重复

broker.id=0

#删除topic功能使能

delete.topic.enable=true

#处理网络请求的线程数量

num.network.threads=3

#用来处理磁盘IO的现成数量

num.io.threads=8

#发送套接字的缓冲区大小

socket.send.buffer.bytes=102400

#接收套接字的缓冲区大小

socket.receive.buffer.bytes=102400

#请求套接字的缓冲区大小

socket.request.max.bytes=104857600

#kafka运行日志存放的路径

log.dirs=/opt/module/kafka/logs

#topic在当前broker上的分区个数

num.partitions=1

#用来恢复和清理data下数据的线程数量

num.recovery.threads.per.data.dir=1

#segment文件保留的最长时间，超时将被删除

log.retention.hours=168

#配置连接Zookeeper集群地址

zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181/kafka

5）配置环境变量

[atguigu@hadoop102 module]$ sudo vi /etc/profile

#KAFKA\_HOME

export KAFKA\_HOME=/opt/module/kafka

export PATH=$PATH:$KAFKA\_HOME/bin

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

6）分发安装包

[atguigu@hadoop102 module]$ xsync kafka/

注意：分发之后记得配置其他机器的环境变量

7）分别在hadoop103和hadoop104上修改配置文件/opt/module/kafka/config/server.properties中的broker.id=1、broker.id=2

注：broker.id不得重复

8）启动集群

依次在hadoop102、hadoop103、hadoop104节点上启动kafka

[atguigu@hadoop102 kafka]$ kafka-server-start.sh -daemon $KAFKA\_HOME/config/server.properties

[atguigu@hadoop103 kafka]$ kafka-server-start.sh -daemon $KAFKA\_HOME/config/server.properties

[atguigu@hadoop104 kafka]$ kafka-server-start.sh -daemon $KAFKA\_HOME/config/server.properties

9）关闭集群

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

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

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

10）kafka群起脚本

for i in `cat /opt/module/hadoop-2.7.2/etc/hadoop/slaves`

do

echo "========== $i =========="

ssh $i '/opt/module/kafka/bin/kafka-server-start.sh -daemon /opt/module/kafka/config/server.properties'

echo $?

done

## 2.2 Kafka命令行操作

1）查看当前服务器中的所有topic

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181/kafka --list

2）创建topic

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181/kafka \

--create --replication-factor 3 --partitions 1 --topic first

选项说明：

--topic 定义topic名

--replication-factor 定义副本数

--partitions 定义分区数

3）删除topic

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181/kafka \

--delete --topic first

需要server.properties中设置delete.topic.enable=true否则只是标记删除。

4）发送消息

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

--broker-list hadoop102:9092 --topic first

>hello world

>atguigu atguigu

5）消费消息

[atguigu@hadoop103 kafka]$ bin/kafka-console-consumer.sh \

--bootstrap-server hadoop102:9092 --from-beginning --topic first

[atguigu@hadoop103 kafka]$ bin/kafka-console-consumer.sh \

--bootstrap-server hadoop102:9092 --from-beginning --topic first

--from-beginning：会把主题中以往所有的数据都读取出来。

6）查看某个Topic的详情

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --zookeeper hadoop102:2181/kafka \

--describe --topic first

7）修改分区数

[atguigu@hadoop102 kafka]$bin/kafka-topics.sh --zookeeper hadoop102:2181/kafka --alter --topic first --partitions 6

# 第3章 Kafka架构深入

## 3.1 Kafka工作流程及文件存储机制



Kafka中消息是以**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的。

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。



由于生产者生产的消息会不断追加到log文件末尾，为防止log文件过大导致数据定位效率低下，Kafka采取了**分片**和**索引**机制，将每个partition分为多个segment。每个segment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，first这个topic有三个分区，则其对应的文件夹为first-0,first-1,first-2。

00000000000000000000.index

00000000000000000000.log

00000000000000170410.index

00000000000000170410.log

00000000000000239430.index

00000000000000239430.log

index和log文件以当前segment的第一条消息的offset命名。下图为index文件和log文件的结构示意图。



“.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。

## 3.2 Kafka生产者

### 3.2.2 分区策略

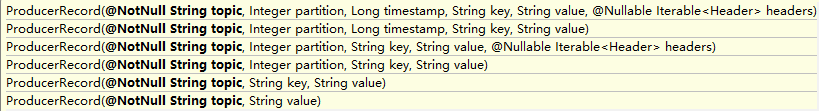
**1）分区的原因**

（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**

我们需要将producer发送的数据封装成一个**ProducerRecord**对象。



（1）指明 partition 的情况下，直接将指明的值直接作为 partiton 值；

（2）没有指明 partition 值但有 key 的情况下，将 key 的 hash 值与 topic 的 partition 数进行取余得到 partition 值；

（3）既没有 partition 值又没有 key 值的情况下，第一次调用时随机生成一个整数（后面每次调用在这个整数上自增），将这个值与 topic 可用的 partition 总数取余得到 partition 值，也就是常说的 round-robin 算法。

### 3.2.3 数据可靠性保证

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。



**1）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producer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。

**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**。



**4）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。**

### 3.2.4 Exactly Once语义

将服务器的ACK级别设置为-1，可以保证Producer到Server之间不会丢失数据，即At Least Once语义。相对的，将服务器ACK级别设置为0，可以保证生产者每条消息只会被发送一次，即At Most Once语义。

At Least Once可以保证数据不丢失，但是不能保证数据不重复；相对的，At Least Once可以保证数据不重复，但是不能保证数据不丢失。但是，对于一些非常重要的信息，比如说交易数据，下游数据消费者要求数据既不重复也不丢失，即Exactly Once语义。在0.11版本以前的Kafka，对此是无能为力的，只能保证数据不丢失，再在下游消费者对数据做全局去重。对于多个下游应用的情况，每个都需要单独做全局去重，这就对性能造成了很大影响。

0.11版本的Kafka，引入了一项重大特性：幂等性。所谓的幂等性就是指Producer不论向Server发送多少次重复数据，Server端都只会持久化一条。幂等性结合At Least Once语义，就构成了Kafka的Exactly Once语义。即：

At Least Once + 幂等性 = Exactly Once

要启用幂等性，只需要将Producer的参数中enable.idompotence设置为true即可。Kafka的幂等性实现其实就是将原来下游需要做的去重放在了数据上游。开启幂等性的Producer在初始化的时候会被分配一个PID，发往同一Partition的消息会附带Sequence Number。而Broker端会对<PID, Partition, SeqNumber>做缓存，当具有相同主键的消息提交时，Broker只会持久化一条。

但是PID重启就会变化，同时不同的Partition也具有不同主键，所以幂等性无法保证跨分区跨会话的Exactly Once。

## 3.3 Kafka消费者

### 3.3.1 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。

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。

### 3.3.3 分区分配策略

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。

**1）roundrobin**

****

**2）range**

****

### 3.3.4 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 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**。

## 3.4 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）应用Pagecache**

Kafka数据持久化是直接持久化到Pagecache中，这样会产生以下几个好处：

* + I/O Scheduler 会将连续的小块写组装成大块的物理写从而提高性能
  + I/O Scheduler 会尝试将一些写操作重新按顺序排好，从而减少磁盘头的移动时间
  + 充分利用所有空闲内存（非 JVM 内存）。如果使用应用层 Cache（即 JVM 堆内存），会增加 GC 负担
  + 读操作可直接在 Page Cache 内进行。如果消费和生产速度相当，甚至不需要通过物理磁盘（直接通过 Page Cache）交换数据
  + 如果进程重启，JVM 内的 Cache 会失效，但 Page Cache 仍然可用

尽管持久化到Pagecache上可能会造成宕机丢失数据的情况，但这可以被Kafka的Replication机制解决。如果为了保证这种情况下数据不丢失而强制将 Page Cache 中的数据 Flush 到磁盘，反而会降低性能。

**3）零复制技术**

`

## 3.5 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的leader选举过程：



## 3.6 Kafka事务

Kafka从0.11版本开始引入了事务支持。事务可以保证Kafka在Exactly Once语义的基础上，生产和消费可以跨分区和会话，要么全部成功，要么全部失败。

### 3.6.1 Producer事务

为了实现跨分区跨会话的事务，需要引入一个全局唯一的Transaction ID，并将Producer获得的PID和Transaction ID绑定。这样当Producer重启后就可以通过正在进行的Transaction ID获得原来的PID。

为了管理Transaction，Kafka引入了一个新的组件Transaction Coordinator。Producer就是通过和Transaction Coordinator交互获得Transaction ID对应的任务状态。Transaction Coordinator还负责将事务所有写入Kafka的一个内部Topic，这样即使整个服务重启，由于事务状态得到保存，进行中的事务状态可以得到恢复，从而继续进行。

### 3.6.2 Consumer事务

上述事务机制主要是从Producer方面考虑，对于Consumer而言，事务的保证就会相对较弱，尤其时无法保证Commit的信息被精确消费。这是由于Consumer可以通过offset访问任意信息，而且不同的Segment File生命周期不同，同一事务的消息可能会出现重启后被删除的情况。

# 第4章 Kafka API

## 4.1 Producer API

### 4.1.1 消息发送流程

Kafka的Producer发送消息采用的是**异步发送**的方式。在消息发送的过程中，涉及到了**两个线程——main线程和Sender线程**，以及**一个线程共享变量——RecordAccumulator**。main线程将消息发送给RecordAccumulator，Sender线程不断从RecordAccumulator中拉取消息发送到Kafka broker。



**相关参数：**

**batch.size：**只有数据积累到batch.size之后，sender才会发送数据。

**linger.ms：**如果数据迟迟未达到batch.size，sender等待linger.time之后就会发送数据。

### 4.1.1 异步发送API

**1）导入依赖**

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>2.4.1</version>

</dependency>

**2）编写代码**

需要用到的类：

**KafkaProducer**：需要创建一个生产者对象，用来发送数据

**ProducerConfig**：获取所需的一系列配置参数

**ProducerRecord**：每条数据都要封装成一个ProducerRecord对象

**1.不带回调函数的API**

package com.atguigu.kafka;

import org.apache.kafka.clients.producer.\*;

import java.util.Properties;

import java.util.concurrent.ExecutionException;

public class CustomProducer {

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

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");//kafka集群，broker-list

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

props.put("retries", 1);//重试次数

props.put("batch.size", 16384);//批次大小

props.put("linger.ms", 1);//等待时间

props.put("buffer.memory", 33554432);//RecordAccumulator缓冲区大小

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

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

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

for (int i = 0; i < 100; i++) {

producer.send(new ProducerRecord<String, String>("first", Integer.toString(i), Integer.toString(i)));

}

producer.close();

}

}

**2.带回调函数的API**

回调函数会在producer收到ack时调用，为异步调用，该方法有两个参数，分别是RecordMetadata和Exception，如果Exception为null，说明消息发送成功，如果Exception不为null，说明消息发送失败。

注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试。

package com.atguigu.kafka;

import org.apache.kafka.clients.producer.\*;

import java.util.Properties;

import java.util.concurrent.ExecutionException;

public class CustomProducer {

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

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");//kafka集群，broker-list

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

props.put("retries", 1);//重试次数

props.put("batch.size", 16384);//批次大小

props.put("linger.ms", 1);//等待时间

props.put("buffer.memory", 33554432);//RecordAccumulator缓冲区大小

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

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

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

for (int i = 0; i < 100; i++) {

producer.send(new ProducerRecord<String, String>("first", Integer.toString(i), Integer.toString(i)), new Callback() {

//回调函数，该方法会在Producer收到ack时调用，为异步调用

@Override

public void onCompletion(RecordMetadata metadata, Exception exception) {

if (exception == null) {

System.out.println("success->" + metadata.offset());

} else {

exception.printStackTrace();

}

}

});

}

producer.close();

}

}

### 4.1.2 同步发送API

同步发送的意思就是，一条消息发送之后，会阻塞当前线程，直至返回ack。

由于send方法返回的是一个Future对象，根据Futrue对象的特点，我们也可以实现同步发送的效果，只需在调用Future对象的get方发即可。

package com.atguigu.kafka;

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

import org.apache.kafka.clients.producer.Producer;

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

import java.util.Properties;

import java.util.concurrent.ExecutionException;

public class CustomProducer {

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

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");//kafka集群，broker-list

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

props.put("retries", 1);//重试次数

props.put("batch.size", 16384);//批次大小

props.put("linger.ms", 1);//等待时间

props.put("buffer.memory", 33554432);//RecordAccumulator缓冲区大小

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

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

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

for (int i = 0; i < 100; i++) {

producer.send(new ProducerRecord<String, String>("first", Integer.toString(i), Integer.toString(i))).get();

}

producer.close();

}

}

## 4.2 Consumer API

Consumer消费数据时的可靠性是很容易保证的，因为数据在Kafka中是持久化的，故不用担心数据丢失问题。

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。

所以offset的维护是Consumer消费数据是必须考虑的问题。

### 4.2.1 自动提交offset

**1）导入依赖**

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>2.4.1</version>

</dependency>

**2）编写代码**

需要用到的类：

**KafkaConsumer**：需要创建一个消费者对象，用来消费数据

**ConsumerConfig**：获取所需的一系列配置参数

**ConsuemrRecord**：每条数据都要封装成一个ConsumerRecord对象

为了使我们能够专注于自己的业务逻辑，Kafka提供了自动提交offset的功能。

自动提交offset的相关参数：

**enable.auto.commit：**是否开启自动提交offset功能

**auto.commit.interval.ms：**自动提交offset的时间间隔

以下为自动提交offset的代码：

package com.atguigu.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import org.apache.kafka.clients.consumer.ConsumerRecords;

import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Arrays;

import java.util.Properties;

public class CustomConsumer {

public static void main(String[] args) {

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");

props.put("group.id", "test");

props.put("enable.auto.commit", "true");

props.put("auto.commit.interval.ms", "1000");

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("first"));

while (true) {

ConsumerRecords<String, String> records = consumer.poll(100);

for (ConsumerRecord<String, String> record : records)

System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

}

}

}

### 4.2.2 手动提交offset

虽然自动提交offset十分简介便利，但由于其是基于时间提交的，开发人员难以把握offset提交的时机。因此Kafka还提供了手动提交offset的API。

手动提交offset的方法有两种：分别是commitSync（同步提交）和commitAsync（异步提交）。两者的相同点是，都会将**本次poll的一批数据最高的偏移量提交**；不同点是，commitSync阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；而commitAsync则没有失败重试机制，故有可能提交失败。

**1）同步提交offset**

由于同步提交offset有失败重试机制，故更加可靠，以下为同步提交offset的示例。

package com.atguigu.kafka.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import org.apache.kafka.clients.consumer.ConsumerRecords;

import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Arrays;

import java.util.Properties;

/\*\*

\* @author liubo

\*/

public class CustomComsumer {

public static void main(String[] args) {

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");//Kafka集群

props.put("group.id", "test");//消费者组，只要group.id相同，就属于同一个消费者组

props.put("enable.auto.commit", "false");//关闭自动提交offset

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("first"));//消费者订阅主题

while (true) {

ConsumerRecords<String, String> records = consumer.poll(100);//消费者拉取数据

for (ConsumerRecord<String, String> record : records) {

System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

}

consumer.commitSync();//同步提交，当前线程会阻塞知道offset提交成功

}

}

}

**2）异步提交offset**

虽然同步提交offset更可靠一些，但是由于其会阻塞当前线程，直到提交成功。因此吞吐量会收到很大的影响。因此更多的情况下，会选用异步提交offset的方式。

以下为异步提交offset的示例：

package com.atguigu.kafka.consumer;

import org.apache.kafka.clients.consumer.\*;

import org.apache.kafka.common.TopicPartition;

import java.util.Arrays;

import java.util.Map;

import java.util.Properties;

/\*\*

\* @author liubo

\*/

public class CustomConsumer {

public static void main(String[] args) {

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");//Kafka集群

props.put("group.id", "test");//消费者组，只要group.id相同，就属于同一个消费者组

props.put("enable.auto.commit", "false");//关闭自动提交offset

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("first"));//消费者订阅主题

while (true) {

ConsumerRecords<String, String> records = consumer.poll(100);//消费者拉取数据

for (ConsumerRecord<String, String> record : records) {

System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

}

consumer.commitAsync(new OffsetCommitCallback() {

@Override

public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception exception) {

if (exception != null) {

System.err.println("Commit failed for" + offsets);

}

}

});//异步提交

}

}

}

1. **数据漏消费和重复消费分析**

无论是同步提交还是异步提交offset，都有可能会造成数据的漏消费或者重复消费。先提交offset后消费，有可能造成数据的漏消费；而先消费后提交offset，有可能会造成数据的重复消费。



### 4.2.3 自定义存储offset

Kafka 0.9版本之前，offset存储在zookeeper，0.9版本之后，默认将offset存储在Kafka的一个内置的topic中。除此之外，Kafka还可以选择自定义存储offset。

Offset的维护是相当繁琐的，因为需要考虑到消费者的Rebalace。

当有新的消费者加入消费者组、已有的消费者推出消费者组或者所订阅的主题的分区发生变化，就会触发到分区的重新分配，重新分配的过程叫做Rebalance。

消费者发生Rebalance之后，每个消费者消费的分区就会发生变化。因此消费者要首先获取到自己被重新分配到的分区，并且定位到每个分区最近提交的offset位置继续消费。

要实现自定义存储offset，需要借助**ConsumerRebalanceListener**，以下为示例代码，其中提交和获取offset的方法，需要根据所选的offset存储系统自行实现。

package com.atguigu.consumer;

import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import org.apache.kafka.clients.consumer.ConsumerRecords;

import org.apache.kafka.clients.consumer.KafkaConsumer;

import org.apache.kafka.common.TopicPartition;

import java.io.\*;

import java.util.\*;

/\*\*

\* 自定义保存Offset

\*/

public class ConsumerManual {

private static Map<TopicPartition, Long> offset = new HashMap<TopicPartition, Long>();

private static String file = "d:/offset";

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

//1. 实例化Consumer对象

Properties properties = new Properties();

properties.load(Consumer.class.getClassLoader().getResourceAsStream("consumer1.properties"));

final KafkaConsumer<String, String> consumer =

new KafkaConsumer<String, String>(properties);

//2. 订阅话题，拉取消息

consumer.subscribe(Collections.singleton("first"),

new ConsumerRebalanceListener() {

//分区分配之前做的事情

public void onPartitionsRevoked(Collection<TopicPartition> partitions) {

//提交旧的offset

commit();

}

//分区分配之后做的事情

public void onPartitionsAssigned(Collection<TopicPartition> partitions) {

//获取新的offset

readOffset(partitions);

for (TopicPartition partition : partitions) {

Long os = offset.get(partition);

if (os == null) {

consumer.seek(partition,0);

} else {

consumer.seek(partition, os);

}

}

}

});

while (true) {

ConsumerRecords<String, String> records = consumer.poll(2000);

//原子绑定

{

for (ConsumerRecord<String, String> record : records) {

//消费

System.out.println(record);

offset.put(

new TopicPartition(record.topic(), record.partition()),

record.offset());

}

commit();

}

}

}

/\*\*

\* 从自定义介质读取offset到缓存

\* @param partitions

\*/

private static void readOffset(Collection<TopicPartition> partitions) {

Map<TopicPartition, Long> temp = readFile();

//从全部分区offset中读取我们分配到的分区的offset

for (TopicPartition partition : partitions) {

offset.put(partition, temp.get(partition));

}

}

/\*\*

\* 将缓存中的Offset提交到自定义介质中

\*

\*/

private static void commit() {

//1. 先从文件中读取旧的所哟Offset

Map<TopicPartition, Long> temp = readFile();

//2. 合并我们的Offset

temp.putAll(offset);

//3. 将新的Offset写出去

writeFile(temp);

}

private static Map<TopicPartition, Long> readFile() {

ObjectInputStream objectInputStream = null;

Map<TopicPartition, Long> temp;

try {

objectInputStream = new ObjectInputStream(new FileInputStream(file));

temp = (Map<TopicPartition, Long>) objectInputStream.readObject();

} catch (Exception e) {

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

} finally {

if (objectInputStream != null) {

try {

objectInputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return temp;

}

private static void writeFile(Map<TopicPartition, Long> temp) {

ObjectOutputStream objectOutputStream = null;

try {

objectOutputStream = new ObjectOutputStream(new FileOutputStream(file));

objectOutputStream.writeObject(temp);

} catch (IOException e) {

e.printStackTrace();

} finally {

if (objectOutputStream != null) {

try {

objectOutputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

## 4.3 自定义Interceptor

### 4.3.1 拦截器原理

Producer拦截器(interceptor)是在Kafka 0.10版本被引入的，主要用于实现clients端的定制化控制逻辑。

对于producer而言，interceptor使得用户在消息发送前以及producer回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer允许用户指定多个interceptor按序作用于同一条消息从而形成一个拦截链(interceptor chain)。Intercetpor的实现接口是org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括：

（1）configure(configs)

获取配置信息和初始化数据时调用。

（2）onSend(ProducerRecord)：

该方法封装进KafkaProducer.send方法中，即它运行在用户主线程中。Producer确保在消息被序列化以及计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的topic和分区，否则会影响目标分区的计算。

（3）onAcknowledgement(RecordMetadata, Exception)：

该方法会在消息从RecordAccumulator成功发送到Kafka Broker之后，或者在发送过程中失败时调用。并且通常都是在producer回调逻辑触发之前。onAcknowledgement运行在producer的IO线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢producer的消息发送效率。

（4）close：

关闭interceptor，主要用于执行一些资源清理工作

如前所述，interceptor可能被运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外倘若指定了多个interceptor，则producer将按照指定顺序调用它们，并仅仅是捕获每个interceptor可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中要特别留意。

### 4.3.2 拦截器案例

1）需求：

实现一个简单的双interceptor组成的拦截链。第一个interceptor会在消息发送前将时间戳信息加到消息value的最前部；第二个interceptor会在消息发送后更新成功发送消息数或失败发送消息数。



2）案例实操

（1）增加时间戳拦截器

package com.atguigu.kafka.interceptor;

import java.util.Map;

import org.apache.kafka.clients.producer.ProducerInterceptor;

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

import org.apache.kafka.clients.producer.RecordMetadata;

public class TimeInterceptor implements ProducerInterceptor<String, String> {

@Override

public void configure(Map<String, ?> configs) {

}

@Override

public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {

// 创建一个新的record，把时间戳写入消息体的最前部

return new ProducerRecord(record.topic(), record.partition(), record.timestamp(), record.key(),

System.currentTimeMillis() + "," + record.value().toString());

}

@Override

public void onAcknowledgement(RecordMetadata metadata, Exception exception) {

}

@Override

public void close() {

}

}

（2）统计发送消息成功和发送失败消息数，并在producer关闭时打印这两个计数器

package com.atguigu.kafka.interceptor;

import java.util.Map;

import org.apache.kafka.clients.producer.ProducerInterceptor;

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

import org.apache.kafka.clients.producer.RecordMetadata;

public class CounterInterceptor implements ProducerInterceptor<String, String>{

private int errorCounter = 0;

private int successCounter = 0;

@Override

public void configure(Map<String, ?> configs) {

}

@Override

public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {

return record;

}

@Override

public void onAcknowledgement(RecordMetadata metadata, Exception exception) {

// 统计成功和失败的次数

if (exception == null) {

successCounter++;

} else {

errorCounter++;

}

}

@Override

public void close() {

// 保存结果

System.out.println("Successful sent: " + successCounter);

System.out.println("Failed sent: " + errorCounter);

}

}

（3）producer主程序

package com.atguigu.kafka.interceptor;

import java.util.ArrayList;

import java.util.List;

import java.util.Properties;

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

import org.apache.kafka.clients.producer.Producer;

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

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

public class InterceptorProducer {

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

// 1 设置配置信息

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");

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

props.put("retries", 0);

props.put("batch.size", 16384);

props.put("linger.ms", 1);

props.put("buffer.memory", 33554432);

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

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

// 2 构建拦截链

List<String> interceptors = new ArrayList<>();

interceptors.add("com.atguigu.kafka.interceptor.TimeInterceptor"); interceptors.add("com.atguigu.kafka.interceptor.CounterInterceptor");

props.put(ProducerConfig.INTERCEPTOR\_CLASSES\_CONFIG, interceptors);

String topic = "first";

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

// 3 发送消息

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

ProducerRecord<String, String> record = new ProducerRecord<>(topic, "message" + i);

producer.send(record);

}

// 4 一定要关闭producer，这样才会调用interceptor的close方法

producer.close();

}

}

3）测试

（1）在kafka上启动消费者，然后运行客户端java程序。

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

--bootstrap-server hadoop102:9092 --from-beginning --topic first

1501904047034,message0

1501904047225,message1

1501904047230,message2

1501904047234,message3

1501904047236,message4

1501904047240,message5

1501904047243,message6

1501904047246,message7

1501904047249,message8

1501904047252,message9

# 第5章 Flume对接Kafka

**1）配置flume(flume-kafka.conf)**

# define

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# source

a1.sources.r1.type = exec

a1.sources.r1.command = tail -F -c +0 /opt/module/datas/flume.log

a1.sources.r1.shell = /bin/bash -c

# sink

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.kafka.bootstrap.servers = hadoop102:9092,hadoop103:9092,hadoop104:9092

a1.sinks.k1.kafka.topic = first

a1.sinks.k1.kafka.flumeBatchSize = 20

a1.sinks.k1.kafka.producer.acks = 1

a1.sinks.k1.kafka.producer.linger.ms = 1

# channel

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# bind

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

**2） 启动kafkaIDEA消费者**

**3） 进入flume根目录下，启动flume**

$ bin/flume-ng agent -c conf/ -n a1 -f jobs/flume-kafka.conf

**4） 向 /opt/module/datas/flume.log里追加数据，查看kafka消费者消费情况**

$ echo hello >> /opt/module/datas/flume.log

# 第6章 Kafka监控

## 6.1 Kafka Monitor

1.上传jar包KafkaOffsetMonitor-assembly-0.4.6.jar到集群

2.在/opt/module/下创建kafka-offset-console文件夹

3.将上传的jar包放入刚创建的目录下

4.在/opt/module/kafka-offset-console目录下创建启动脚本start.sh，内容如下：

#!/bin/bash

java -cp KafkaOffsetMonitor-assembly-0.4.6-SNAPSHOT.jar \

com.quantifind.kafka.offsetapp.OffsetGetterWeb \

--offsetStorage kafka \

--kafkaBrokers hadoop102:9092,hadoop103:9092,hadoop104:9092 \

--kafkaSecurityProtocol PLAINTEXT \

--zk hadoop102:2181,hadoop103:2181,hadoop104:2181/kafka \

--port 8086 \

--refresh 10.seconds \

--retain 2.days \

--dbName offsetapp\_kafka &

5.在/opt/module/kafka-offset-console目录下创建mobile-logs文件夹

mkdir /opt/module/kafka-offset-console/mobile-logs

6.启动KafkaMonitor

./start.sh

7.登录页面hadoop102:8086端口查看详情

## 6.2 Kafka Manager

1.上传压缩包kafka-manager-1.3.3.15.zip到集群

2.解压到/opt/module

3.修改配置文件conf/application.conf

kafka-manager.zkhosts="kafka-manager-zookeeper:2181"

修改为：

kafka-manager.zkhosts="hadoop102:2181,hadoop103:2181,hadoop104:2181/kafka"

4.启动kafka-manager

bin/kafka-manager

5.登录hadoop102:9000页面查看详细信息

# 第7章 Kafka面试题

## 7.1 面试问题

1.Kafka中的ISR、AR代表什么？

2.Kafka中的HW、LEO等分别代表什么？

3.Kafka中是怎么体现消息顺序性的？

4.Kafka中的分区器、序列化器、拦截器是否了解？它们之间的处理顺序是什么？

5.Kafka生产者客户端的整体结构是什么样子的？使用了几个线程来处理？分别是什么？

6.“消费者组中的消费者个数如果超过topic的分区，那么就会有消费者消费不到数据”这句话是否正确？

7.消费者提交消费位移时提交的是当前消费到的最新消息的offset还是offset+1？

8.有哪些情形会造成重复消费？

9.那些情景会造成消息漏消费？

10.当你使用kafka-topics.sh创建（删除）了一个topic之后，Kafka背后会执行什么逻辑？

1）会在zookeeper中的/brokers/topics节点下创建一个新的topic节点，如：/brokers/topics/first

2）触发Controller的监听程序

3）kafka Controller 负责topic的创建工作，并更新metadata cache

11.topic的分区数可不可以增加？如果可以怎么增加？如果不可以，那又是为什么？

12.topic的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么？

13.Kafka有内部的topic吗？如果有是什么？有什么作用？

14.Kafka分区分配的概念？

15.简述Kafka的日志目录结构？

16.如果我指定了一个offset，Kafka Controller怎么查找到对应的消息？

17.聊一聊Kafka Controller的作用？

18.Kafka中有那些地方需要选举？这些地方的选举策略又有哪些？

19.失效副本是指什么？有那些应对措施？

20.Kafka的那些设计让它有如此高的性能？

## 7.2 参考答案

