**Kafka**

#### Kafka安装与配置

1. 下载Kafka

http://mirror.bit.edu.cn/apache/kafka/1.1.1/kafka\_2.11-1.1.1.tgz

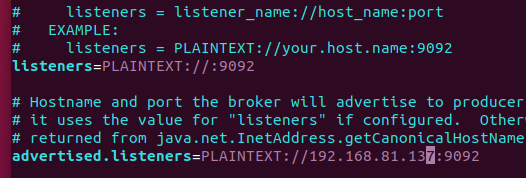
1. 将tar文件通过xftp传到linux上
2. 解压文件
3. 修改根目录config文件下的server.properties



1. 运行命令启动kafka

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

1. 允许远程访问



1. 搭建kafka集群

修改server.properties配置文件



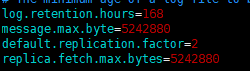
broker.id相当于zookeeper的myid不能相同

在log.retention.hours=168 下面新增下面三项

message.max.byte=5242880

default.replication.factor=2

replica.fetch.max.bytes=5242880



修改端口号

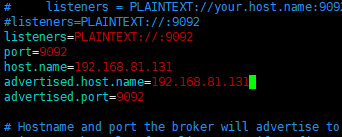
listeners=PLAINTEXT://:9092

port=9092

host.name=192.168.81.131

advertised.host.name=192.168.81.131

advertised.port=9092



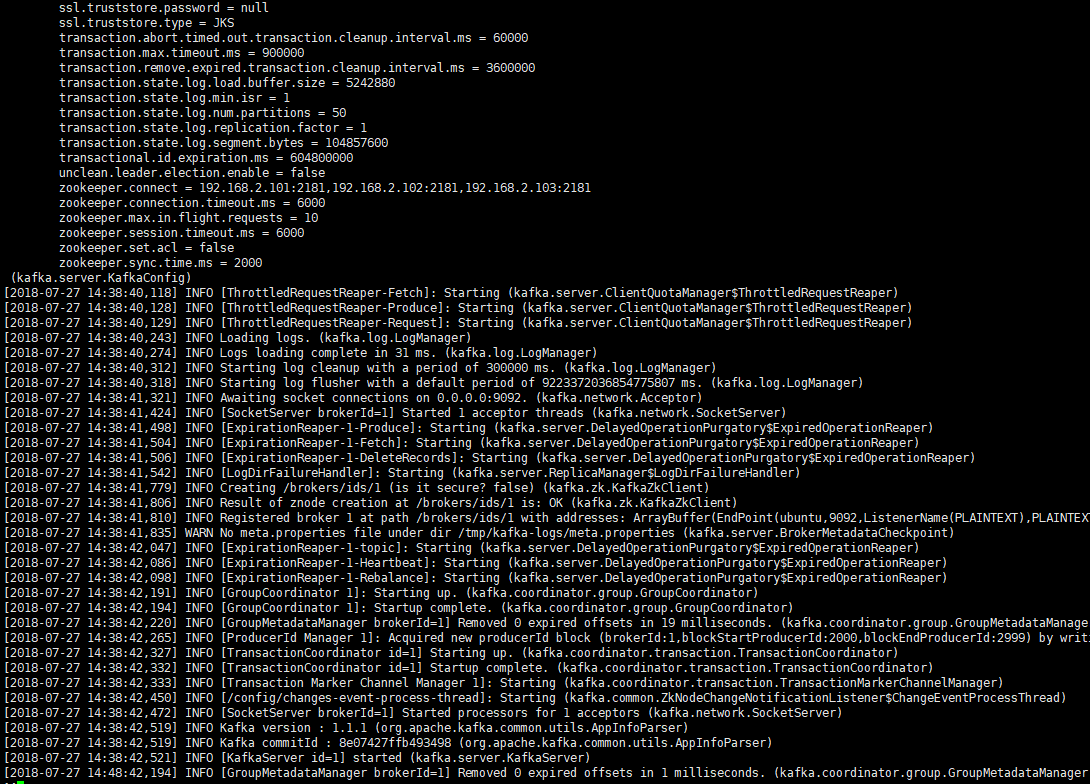
注意：按照官方文档的说法，advertised.host.name 和 advertised.port 这两个参数用于定义集群向 Producer 和 Consumer 广播的节点 host 和 port，如果不定义，会默认使用 host.name 和 port 的定义。但在实际应用中，发现如果不定义 advertised.host.name 参数，使用 Java 客户端从远端连接集群时，会发生连接超时，抛出异常：org.apache.kafka.common.errors.TimeoutException: Batch Expired

经过过 debug 发现，连接到集群是成功的，但连接到集群后更新回来的集群 meta 信息却是错误的。metadata 中的 Cluster 信息中节点的 hostname 是一串字符，而不是实际的ip地址。这串其实是远端主机的 hostname，这说明在没有配置 advertised.host.name 的情况下，Kafka 并没有像官方文档宣称的那样改为广播我们配置的 host.name，而是广播了主机配置的 hostname 。远端的客户端并没有配置 hosts，所以自然是连接不上这个 hostname 的。要解决这一问题，把 host.name 和 advertised.host.name 都配置成绝对的 ip 地址就可以了。

1. 若之前启动过kafka需要修改log.dirs

在/tmp/kafka-logs下修改meta.properties中的broker.id

1. 启动集群—>启动成功，如图所示



1. 注：kafka依赖于zookeeper，所以启动kafka之前需要启动zookeeper
2. 创建topic

kafka-topics.sh --create --zookeeper 192.168.81.131:2181,192.168.81.132:2181,192.168.81.133:2181 --replication-factor 3 --partitions 3 --topic mykafka

#解释

--replication-factor 3 #复制3份

--partitions 3 #创建3个分区

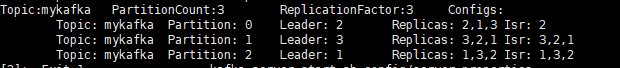
--topic #主题为mykafka





查看topic状态

kafka-topics.sh --describe --zookeeper 192.168.81.131:2181,192.168.81.132:2181,192.168.81.133:2181 --topic mykafka



1. 关闭kafka

kafka-server-stop.sh

1. 后台启动kafka

根目录下运行

Bin/kafka-server-start.sh config/server.properties 1>/dev/null 2>&1 &

其中1>/dev/null 2>&1 是将命令产生的输入和错误都输入到空设备，也就是不输出的意思。

/dev/null代表空设备。

#### Kafka创建生产者与消费者

1. 客户端命令
2. 创建生产者客户端

bin/kafka-console-producer.sh --broker-list 192.168.81.131:9092,192.168.81.132:9092,192.168.81.133:9092 --topic baseTopic

然后就可以向topic写数据



1. 创建消费者客户端

bin/kafka-console-consumer.sh --zookeeper 192.168.81.131:9092,192.168.81.132:9092,192.168.81.133:9092 --topic baseTopic --from-beginning

1. Java API

Maven依赖：

<dependency>

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

<artifactId>kafka-clients</artifactId>

<version>1.1.0</version>

</dependency>

1. 生产者---简单模式

**public static void** main(String[] args) **throws** Exception{  
 Properties pros = **new** Properties();  
 *//设置kafka集群的地址*pros.put(**"bootstrap.servers"**,**"192.168.81.131:9092,192.168.81.132:9092,192.168.81.133:9092"**);  
 *//ack模式，all是最慢但是最安全的* pros.put(**"acks"**,**"-1"**);  
 *//失败重试次数* pros.put(**"retries"**,0);  
 *//每个分区发送消息总字节大小（字节），超过大小就会自动提交到服务端* pros.put(**"batch.size"**,10);  
 *//消息在缓冲区保留的时间，超过时间就会发送到服务器* pros.put(**"linger.ms"**,10000);  
 *//整个produce用到的总内存的大小，如果缓冲区满了会提交数据到服务端  
 //buffer.memory要大于batch.size，不然会报内存不足* pros.put(**"buffer.memory"**,10240);  
 *//序列化器*pros.put(**"key.serializer"**,**"org.apache.kafka.common.serialization.StringSerializer"**);  
pros.put(**"value.serializer"**,**"org.apache.kafka.common.serialization.StringSerializer"**);  
 *//创建producer* Producer<String,String> producer = **new** KafkaProducer<>(pros);  
 *//发送信息* **for**(**int** i=0;i<100;i++){  
 producer.send(**new** ProducerRecord<String,String>(**"baseTopic"**,i+**""**,**"info:"**+i));  
 System.***out***.println(i);  
 }  
 Thread.*sleep*(5000);  
 System.***out***.println(**"success"**);  
 producer.close();  
}

1. 生产者---幂等模式

幂等性：客户端一次或多次操作，最终数据是一致的，比如购买火车票支付时可能显示网络异常，但其实已经扣款成功，用户再次发起扣款不会再触发真正的扣款Kafka只能保证在一个会话中的幂等性

幂等模式只需要将enable.idempotence设置为true，一旦设置了该属性，那么retries默认是Integer.MAX\_VALUE ，acks默认是all。代码的写法和前面例子没什么区别

1. 生产者---事务模式

**public static void** main(String[] args) **throws** Exception{  
 Properties pros = **new** Properties();  
 *//设置kafka集群的地址*pros.put(**"bootstrap.servers"**,**"192.168.81.131:9092,192.168.81.132:9092,192.168.81.133:9092"**);  
pros.put(**"key.serializer"**,**"org.apache.kafka.common.serialization.StringSerializer"**);  
pros.put(**"value.serializer"**,**"org.apache.kafka.common.serialization.StringSerializer"**);  
 *//设置事务id* pros.put(**"transactional.id"**, **"my\_transactional\_id"**);  
 *//创建producer* Producer<String,String> producer = **new** KafkaProducer<>(pros);  
 *//事务初始化* producer.initTransactions();  
 *//开启事务* producer.beginTransaction();  
 *//发送信息* **for**(**int** i=0;i<100;i++){  
 producer.send(**new** ProducerRecord<String,String>(**"baseTopic"**,i+**""**,**"info:"**+i));  
 System.***out***.println(i);  
 }  
 producer.commitTransaction();  
 producer.close();  
}

1. 消费者---偏移量-自动提交

消费者拉取数据之后自动提交偏移量，不关心后续对消息的处理是否正确

优点：消费快，适用于数据一致性弱的业务场景

缺点：消息很容易丢失

**public static void** main(String[] args) {  
 **final** Properties props = **new** Properties();  
 *//设置kafka集群地址* props.put(**"bootstrap.servers"**, **"192.168.81.131:9092,192.168.81.132:9092,192.168.81.133:9092"**);  
 *//设置消费者组，组名字自定义，组名字相同的消费在一个组* props.put(**"group.id"**, **"test"**);  
 *//开启offset自动提交* props.put(**"enable.auto.commit"**, **"true"**);  
 *//自动提交时间间隔* props.put(**"auto.commit.interval.ms"**, **"1000"**);  
 *//序列化器* props.put(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 props.put(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 *//实例化一个消费者* KafkaConsumer<String, String> consumer = **new** KafkaConsumer<>(props);  
 *//消费者订阅主题，可以订阅多个主题* consumer.subscribe(Arrays.*asList*(**"baseTopic"**));  
 *//死循环不停的从broker中拿数据* **while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(1000);  
 **for** (ConsumerRecord<String, String> record : records) {  
 System.***out***.printf(**"offset = %d, key = %s, value = %s%n"**,  
 record.offset(), record.key(), record.value());  
 }  
 }  
}

1. 消费者---偏移量-手动按消费者提交

通常从Kafka拿到的消息是要做业务处理，而且业务处理完成才算真正消费成功，所以需要客户端控制offset提交时间

**public static void** main(String[] args) {  
 **final** Properties props = **new** Properties();  
 *//设置kafka集群地址* props.put(**"bootstrap.servers"**, **"192.168.81.131:9092,192.168.81.132:9092,192.168.81.133:9092"**);  
 *//设置消费者组，组名字自定义，组名字相同的消费在一个组* props.put(**"group.id"**, **"test"**);  
 *//开启offset自动提交* props.put(**"enable.auto.commit"**, **"false"**);  
 *//自动提交时间间隔* props.put(**"auto.commit.interval.ms"**, **"1000"**);  
 *//序列化器* props.put(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 props.put(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 *//实例化一个消费者* KafkaConsumer<String, String> consumer = **new** KafkaConsumer<>(props);  
 *//消费者订阅主题，可以订阅多个主题* consumer.subscribe(Arrays.*asList*(**"baseTopic"**));  
 **final int** minBatchSize = 50;  
 List<ConsumerRecord<String, String>> buffer = **new** ArrayList<>();  
 **while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(100);  
 **for** (ConsumerRecord<String, String> record : records) {  
 buffer.add(record);  
 }  
 **if** (buffer.size() >= minBatchSize) {  
 *//insertIntoDb(buffer);* **for** (ConsumerRecord bf : buffer) {  
 System.***out***.printf(**"offset = %d, key = %s, value = %s%n"**, bf.offset(), bf.key(), bf.value());  
 }  
 consumer.commitSync();  
 buffer.clear();  
 }  
 }  
}

1. 消费者---偏移量-手动按分区提交

在munualCommit的基础上更细粒度的提交数据，按照每个分区手动提交偏移量

这里实现了按照分区取数据，因此可以从分区入手，不同的分区可以做不同的操作，可以灵活实现一些功能

**public static void** main(String[] args) {  
 **final** Properties props = **new** Properties();  
 *//设置kafka集群地址* props.put(**"bootstrap.servers"**, **"192.168.81.131:9092,192.168.81.132:9092,192.168.81.133:9092"**);  
 *//设置消费者组，组名字自定义，组名字相同的消费在一个组* props.put(**"group.id"**, **"test"**);  
 *//开启offset自动提交* props.put(**"enable.auto.commit"**, **"false"**);  
 *//自动提交时间间隔* props.put(**"auto.commit.interval.ms"**, **"1000"**);  
 *//序列化器* props.put(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 props.put(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 *//实例化一个消费者* KafkaConsumer<String, String> consumer = **new** KafkaConsumer<>(props);  
 *//消费者订阅主题，可以订阅多个主题* consumer.subscribe(Arrays.*asList*(**"baseTopic"**));  
 **final int** minBatchSize = 50;  
 List<ConsumerRecord<String, String>> buffer = **new** ArrayList<>();  
 **while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(Long.***MAX\_VALUE***);  
 **for** (TopicPartition partition : records.partitions()) {  
 List<ConsumerRecord<String, String>> partitionRecords = records.records(partition);  
 **for** (ConsumerRecord<String, String> record : partitionRecords) {  
 System.***out***.println(**"partition: "** + partition.partition() + **" , "** + record.offset() + **": "** + record.value());  
 }  
 **long** lastOffset = partitionRecords.get(partitionRecords.size() - 1).offset();  
 */\*  
 提交的偏移量应该始终是您的应用程序将要读取的下一条消息的偏移量。因此，在调用commitSync（）时，  
 offset应该是处理的最后一条消息的偏移量加1  
 为什么这里要加上面不加喃？因为上面Kafka能够自动帮我们维护所有分区的偏移量设置，有兴趣的同学可以看看SubscriptionState.allConsumed()就知道  
 \*/* consumer.commitSync(Collections.*singletonMap*(partition, **new** OffsetAndMetadata(lastOffset + 1)));  
 }  
 }  
}

1. 消费者---从指定分区拉取数据

消费只读取特定分区数据，这种方式比上面的更加灵活，在实际应用场景中会经常使用

因为分区的数据是有序的，利用这个特性可以用于数据到达有先后顺序的业务，比如一个用户将订单提交，紧接着又取消订单，那么取消的订单一定要后于提交的订单到达某一个分区，这样保证业务处理的正确性

​ 一旦指定了分区，要注意以下两点：

​ a.kafka提供的消费者组内的协调功能就不再有效

​ b.这样的写法可能出现不同消费者分配了相同的分区，为了避免偏移量提交冲突，每个消费者实例的group\_id要不重复

**public static void** main(String[] args) {  
 **final** Properties props = **new** Properties();  
 *//设置kafka集群地址* props.put(**"bootstrap.servers"**, **"192.168.81.131:9092,192.168.81.132:9092,192.168.81.133:9092"**);  
 *//设置消费者组，组名字自定义，组名字相同的消费在一个组* props.put(**"group.id"**, **"test"**);  
 *//开启offset自动提交* props.put(**"enable.auto.commit"**, **"false"**);  
 *//自动提交时间间隔* props.put(**"auto.commit.interval.ms"**, **"1000"**);  
 *//序列化器* props.put(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 props.put(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 *//实例化一个消费者* KafkaConsumer<String, String> consumer = **new** KafkaConsumer<>(props);  
 *//消费者订阅主题，并设置要拉取的分区* TopicPartition partition0 = **new** TopicPartition(**"baseTopic"**,0);  
 consumer.assign(Arrays.*asList*(partition0));  
 List<ConsumerRecord<String, String>> buffer = **new** ArrayList<>();  
 **while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(Long.***MAX\_VALUE***);  
 **for** (TopicPartition partition : records.partitions()) {  
 List<ConsumerRecord<String, String>> partitionRecords = records.records(partition);  
 **for** (ConsumerRecord<String, String> record : partitionRecords) {  
 System.***out***.println(**"partition: "** + partition.partition() + **" , "** + record.offset() + **": "** + record.value());  
 }  
 **long** lastOffset = partitionRecords.get(partitionRecords.size() - 1).offset();  
 consumer.commitSync(Collections.*singletonMap*(partition, **new** OffsetAndMetadata(lastOffset + 1)));  
 }  
 }  
}

1. 获取topic的所有分区
2. 获取指定offset的文件
3. 创建以及删除topic