### [Kafka 0.9+Zookeeper3.4.6集群搭建、配置，新Client API的使用要点，高可用性测试，以及各种坑](http://kelgon.iteye.com/blog/2287985)

Kafka 0.9版本对java client的api做出了较大调整，本文主要总结了Kafka 0.9在集群搭建、高可用性、新API方面的相关过程和细节，以及本人在安装调试过程中踩出的各种坑。

关于Kafka的结构、功能、特点、适用场景等，网上到处都是，我就不再赘述了，直接进入正文

Kafka 0.9集群安装配置

操作系统：CentOS 6.5

**1. 安装Java环境**

    Zookeeper和Kafka的运行都需要Java环境，所以先安装JRE，Kafka默认使用G1垃圾回收器，如果不更改垃圾回收器，官方推荐使用 7u51以上版本的JRE。如果你使用老版本的JRE，需要更改Kafka的启动脚本，指定G1以外的垃圾回收器。Java环境的安装过程在此不赘述了。

**2. Zookeeper集群搭建**

    Kafka依赖Zookeeper管理自身集群（Broker、Offset、Producer、Consumer等），所以先要安装 Zookeeper。自然，为了达到高可用的目的，Zookeeper自身也不能是单点，接下来就介绍如何搭建一个最小的Zookeeper集群（3个 zk节点）

    此处选用Zookeeper的版本是3.4.6，此为Kafka0.9中推荐的Zookeeper版本。

    首先解压

tar -xzvf zookeeper-3.4.6.tar.gz

    进入zookeeper的conf目录，将zoo\_sample.cfg复制一份，命名为zoo.cfg，此即为Zookeeper的配置文件

cp zoo\_sample.cfg zoo.cfg

    编辑zoo.cfg

# The number of milliseconds of each tick   
tickTime=2000   
# The number of ticks that the initial   
# synchronization phase can take   
initLimit=10   
# The number of ticks that can pass between   
# sending a request and getting an acknowledgement   
syncLimit=5   
# the directory where the snapshot is stored.   
dataDir=/data/zk/zk0/data   
dataLogDir=/data/zk/zk0/logs   
# the port at which the clients will connect   
clientPort=2181   
server.0=10.0.0.100:4001:4002   
server.1=10.0.0.101:4001:4002   
server.2=10.0.0.102:4001:4002

* dataDir和dataLogDir的路径需要在启动前创建好
* clientPort为zookeeper的服务端口
* server.0/1/2为zk集群中三个node的信息，定义格式为hostname:port1:port2，其中port1是node间通信使用的端口，port2是node选举使用的端口，需确保三台主机的这两个端口都是互通的

    在另外两台主机上执行同样的操作，安装并配置zookeeper

    分别在三台主机的dataDir路径下创建一个文件名为myid的文件，文件内容为该zk节点的编号。例如在第一台主机上建立的myid文件内容是0，第二台是1。

    接下来，启动三台主机上的zookeeper服务：

bin/zkServer.sh start

    3个节点都启动完成后，可依次执行如下命令查看集群状态：

bin/zkServer.sh status

    命令输出如下：

    Mode: leader 或 Mode: follower

    3个节点中，应有1个leader和两个follower

    验证zookeeper集群高可用性：

    假设目前3个zk节点中，server0为leader，server1和server2为follower

    我们停掉server0上的zookeeper服务：

bin/zkServer.sh stop

    再到server1和server2上查看集群状态，会发现此时server1（也有可能是server2）为leader，另一个为follower。

    再次启动server0的zookeeper服务，运行zkServer.sh status检查，发现新启动的server0也为follower

    至此，zookeeper集群的安装和高可用性验证完成。

    附：Zookeeper默认会将控制台信息输出到启动路径下的zookeeper.out中，显然在生产环境中我们不能允许Zookeeper这样做，通过如下方法，可以让Zookeeper输出按尺寸切分的日志文件：

    修改conf/log4j.properties文件，将

    zookeeper.root.logger=INFO, **CONSOLE**

    改为

    zookeeper.root.logger=INFO, **ROLLINGFILE**

    修改bin/zkEnv.sh文件，将

    ZOO\_LOG4J\_PROP="INFO,**CONSOLE**"

    改为

    ZOO\_LOG4J\_PROP="INFO,**ROLLINGFILE**"

    然后重启zookeeper，就ok了

**3. Kafka集群搭建**

    此例中，我们会安装配置一个有两个Broker组成的Kafka集群，并在其上创建一个两个分区的Topic

    本例中使用Kafka最新版本0.9.0.1

    首先解压

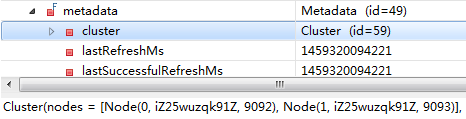
tar -xzvf kafka\_2.11-0.9.0.1.tgz

    编辑config/server.properties文件，下面列出关键的参数

#此Broker的ID，集群中每个Broker的ID不可相同  
broker.id=0  
#监听器，端口号与port一致即可  
listeners=PLAINTEXT://:9092  
#Broker监听的端口  
port=9092  
#Broker的Hostname，填主机IP即可  
host.name=10.0.0.100  
#向Producer和Consumer建议连接的Hostname和port（此处有坑，具体见后）  
advertised.host.name=10.0.0.100  
advertised.port=9092  
#进行IO的线程数，应大于主机磁盘数  
num.io.threads=8  
#消息文件存储的路径  
log.dirs=/data/kafka-logs  
#消息文件清理周期，即清理x小时前的消息记录  
log.retention.hours=168  
#每个Topic默认的分区数，一般在创建Topic时都会指定分区数，所以这个配成1就行了  
num.partitions=1  
#Zookeeper连接串，此处填写上一节中安装的三个zk节点的ip和端口即可  
zookeeper.connect=10.0.0.100:2181,10.0.0.101:2181,10.0.0.102:2181

    配置项的详细说明请见官方文档：http://kafka.apache.org/documentation.html#brokerconfigs

**此处的坑：**

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，如果不定义的话，会默认使用host.name和port的定义。但在实际应用中，我发现如果不定义 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：  
能够看到，metadata中的Cluster信息，节点的hostname是iZ25wuzqk91Z这样的一串数字，而不是实际的ip地址 10.0.0.100和101。iZ25wuzqk91Z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。

    接下来，我们在另一台主机也完成Kafka的安装和配置，然后在两台主机上分别启动Kafka：

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

**此处的坑：**

官方给出的后台启动kafka的方法是：

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

    但用这种方式启动后，只要断开Shell或登出，Kafka服务就会自动shutdown，不知是OS的问题还是SSH的问题还是Kafka自己的问题，总之我改用-daemon方式启动Kafka才不会在断开shell后自动shutdown。

    接下来，我们创建一个名为test，拥有两个分区，两个副本的Topic：

bin/kafka-topics.sh --create --zookeeper 10.0.0.100:2181,10.0.0.101:2181,10.0.0.102:2181 --replication-factor 2 --partitions 2 --topic test

    创建完成后，使用如下命令查看Topic状态：

bin/kafka-topics.sh --describe --zookeeper 10.0.0.100:2181,10.0.0.101:2181,10.0.0.102:2181 --topic test

    输出：

Topic:test PartitionCount:2 ReplicationFactor:2 Configs:  
     Topic: test Partition: 0 Leader: 1 Replicas: 1,0 Isr: 0,1  
     Topic: test Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0,1

    解读：test这个topic，当前有2个分区，分别为0和1，分区0的Leader是1（这个1是broker.id），分区0有两个 Replica（副本），分别是1和0，这两个副本中，Isr（In-sync）的是0和1。分区2的Leader是0，也有两个Replica，同样也 是两个replica都是in-sync状态

至此，Kafka 0.9集群的搭建工作就完成了，接下来我们将介绍新的Java API的使用，以及集群高可用性的验证测试。

**4. 使用Kafka的Producer API来完成消息的推送**

1) Kafka 0.9.0.1的java client依赖：

<dependency>

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

<artifactId>kafka-clients</artifactId>

<version>0.9.0.1</version>

</dependency>

2) 写一个KafkaUtil工具类，用于构造Kafka Client

public class KafkaUtil {

private static KafkaProducer<String, String> kp;

public static KafkaProducer<String, String> getProducer() {

if (kp == null) {

Properties props = new Properties();

props.put("bootstrap.servers", "10.0.0.100:9092,10.0.0.101:9092");

props.put("acks", "1");

props.put("retries", 0);

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

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

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

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

}

return kp;

}

}

  KafkaProducer<K,V>的K代表每条消息的key类型，V代表消息类型。消息的key用于决定此条消息由哪一个partition接收，所以我们需要保证每条消息的key是不同的。

  Producer端的常用配置

* bootstrap.servers：Kafka集群连接串，可以由多个host:port组成
* acks：broker消息确认的模式，有三种：  
  0：不进行消息接收确认，即Client端发送完成后不会等待Broker的确认  
  1：由Leader确认，Leader接收到消息后会立即返回确认信息  
  all：集群完整确认，Leader会等待所有in-sync的follower节点都确认收到消息后，再返回确认信息  
  我们可以根据消息的重要程度，设置不同的确认模式。默认为1
* retries：发送失败时Producer端的重试次数，默认为0
* batch.size：当同时有大量消息要向同一个分区发送时，Producer端会将消息打包后进行批量发送。如果设置为0，则每条消息都独立发送。默认为16384字节
* linger.ms：发送消息前等待的毫秒数，与batch.size配合使用。在消息负载不高的情况下，配置linger.ms能够让Producer在发送消息前等待一定时间，以积累更多的消息打包发送，达到节省网络资源的目的。默认为0
* key.serializer/value.serializer：消息key/value的序列器Class，根据key和value的类型决定
* buffer.memory：消息缓冲池大小。尚未被发送的消息会保存在Producer的内存中，如果消息产生的速度大于消息发送的速度，那么缓冲池满后发送消息的请求会被阻塞。默认33554432字节（32MB）

  更多的Producer配置见官网：http://kafka.apache.org/documentation.html#producerconfigs

  3) 写一个简单的Producer端，每隔1秒向Kafka集群发送一条消息：

public class KafkaTest {

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

Producer<String, String> producer = KafkaUtil.getProducer();

int i = 0;

while(true) {

ProducerRecord<String, String> record = new ProducerRecord<String, String>("test", String.valueOf(i), "this is message"+i);

producer.send(record, new Callback() {

public void onCompletion(RecordMetadata metadata, Exception e) {

if (e != null)

e.printStackTrace();

System.out.println("message send to partition " + metadata.partition() + ", offset: " + metadata.offset());

}

});

i++;

Thread.sleep(1000);

}

}

}

  在调用KafkaProducer的send方法时，可以注册一个回调方法，在Producer端完成发送后会触发回调逻辑，在回调方法的 metadata对象中，我们能够获取到已发送消息的offset和落在的分区等信息。注意，如果acks配置为0，依然会触发回调逻辑，只是拿不到 offset和消息落地的分区信息。

    跑一下，输出是这样的：

message send to partition 0, offset: 28  
message send to partition 1, offset: 26  
message send to partition 0, offset: 29  
message send to partition 1, offset: 27  
message send to partition 1, offset: 28  
message send to partition 0, offset: 30  
message send to partition 0, offset: 31  
message send to partition 1, offset: 29  
message send to partition 1, offset: 30  
message send to partition 1, offset: 31  
message send to partition 0, offset: 32  
message send to partition 0, offset: 33  
message send to partition 0, offset: 34  
message send to partition 1, offset: 32

  乍一看似乎offset乱掉了，但其实这是因为消息分布在了两个分区上，每个分区上的offset其实是正确递增的。

**5. 使用Kafka的Consumer API来完成消息的消费**

1) 改造一下KafkaUtil类，加入Consumer client的构造。

public class KafkaUtil {

private static KafkaProducer<String, String> kp;

private static KafkaConsumer<String, String> kc;

public static KafkaProducer<String, String> getProducer() {

if (kp == null) {

Properties props = new Properties();

props.put("bootstrap.servers", "10.0.0.100:9092,10.0.0.101:9092");

props.put("acks", "1");

props.put("retries", 0);

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

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

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

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

}

return kp;

}

public static KafkaConsumer<String, String> getConsumer() {

if(kc == null) {

Properties props = new Properties();

props.put("bootstrap.servers", "10.0.0.100:9092,10.0.0.101:9092");

props.put("group.id", "1");

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

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

props.put("session.timeout.ms", "30000");

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

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

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

}

return kc;

}

}

  同样，我们介绍一下Consumer常用配置

* bootstrap.servers/key.deserializer/value.deserializer：和Producer端的含义一样，不再赘述
* fetch.min.bytes：每次最小拉取的消息大小（byte）。Consumer会等待消息积累到一定尺寸后进行批量拉取。默认为1，代表有一条就拉一条
* max.partition.fetch.bytes：每次从单个分区中拉取的消息最大尺寸（byte），默认为1M
* group.id：Consumer的group id，同一个group下的多个Consumer不会拉取到重复的消息，不同group下的Consumer则会保证拉取到每一条消息。注意，同一个group下的consumer数量不能超过分区数。
* enable.auto.commit：是否自动提交已拉取消息的offset。提交offset即视为该消息已经成功被消费，该组下的Consumer无法再拉取到该消息（除非手动修改offset）。默认为true
* auto.commit.interval.ms：自动提交offset的间隔毫秒数，默认5000。

  全部的Consumer配置见官方文档：http://kafka.apache.org/documentation.html#newconsumerconfigs

2) 编写Consumer端：

public class KafkaTest {

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

KafkaConsumer<String, String> consumer = KafkaUtil.getConsumer();

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

while(true) {

ConsumerRecords<String, String> records = consumer.poll(1000);

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

System.out.println("fetched from partition " + record.partition() + ", offset: " + record.offset() + ", message: " + record.value());

}

}

}

}

  运行输出：

fetched from partition 0, offset: 28, message: this is message0  
fetched from partition 0, offset: 29, message: this is message2  
fetched from partition 0, offset: 30, message: this is message5  
fetched from partition 0, offset: 31, message: this is message6  
fetched from partition 0, offset: 32, message: this is message10  
fetched from partition 0, offset: 33, message: this is message11  
fetched from partition 0, offset: 34, message: this is message12  
fetched from partition 1, offset: 26, message: this is message1  
fetched from partition 1, offset: 27, message: this is message3  
fetched from partition 1, offset: 28, message: this is message4  
fetched from partition 1, offset: 29, message: this is message7  
fetched from partition 1, offset: 30, message: this is message8  
fetched from partition 1, offset: 31, message: this is message9  
fetched from partition 1, offset: 32, message: this is message13

说明：

* KafkaConsumer的poll方法即是从Broker拉取消息，在poll之前首先要用subscribe方法订阅一个Topic。
* poll方法的入参是拉取超时毫秒数，如果没有新的消息可供拉取，consumer会等待指定的毫秒数，到达超时时间后会直接返回一个空的结果集。
* 如 果Topic有多个partition，KafkaConsumer会在多个partition间以轮询方式实现负载均衡。如果启动了多个 Consumer线程，Kafka也能够通过zookeeper实现多个Consumer间的调度，保证同一组下的Consumer不会重复消费消息。注 意，Consumer数量不能超过partition数，超出部分的Consumer无法拉取到任何数据。
* 可以看出，拉取到的消息并不是完全顺序化的，kafka只能保证一个partition内的消息先进先出，所以在跨partition的情况下，消息的顺序是没有保证的。
* 本 例中采用的是自动提交offset，Kafka client会启动一个线程定期将offset提交至broker。假设在自动提交的间隔内发生故障（比如整个JVM进程死掉），那么有一部分消息是会被 重复消费的。要避免这一问题，可使用手动提交offset的方式。构造consumer时将enable.auto.commit设为false，并在代 码中用consumer.commitSync()来手动提交。

如果不想让kafka控制consumer拉取数据时在partition间的负载均衡，也可以手工控制：

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

KafkaConsumer<String, String> consumer = KafkaUtil.getConsumer();

String topic = "test";

TopicPartition partition0 = new TopicPartition(topic, 0);

TopicPartition partition1 = new TopicPartition(topic, 1);

consumer.assign(Arrays.asList(partition0, partition1));

while(true) {

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

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

System.out.println("fetched from partition " + record.partition() + ", offset: " + record.offset() + ", message: " + record.value());

}

consumer.commitSync();

}

}

 使用consumer.assign()方法为consumer线程指定1个或多个partition。

**此处的坑：**

在测试中我发现，如果用手工指定partition的方法拉取消息，不知为何kafka的自动提交offset机制会失效，必须使用手动方式才能正确提交已消费的消息offset。

  题外话：

在 真正的应用环境中，Consumer端将消息拉取下来后要做的肯定不止是输出出来这么简单，在消费消息时很有可能需要花掉更多的时间。1个 Consumer线程消费消息的速度很有可能是赶不上Producer产生消息的速度，所以我们不得不考虑Consumer端采用多线程模型来消费消息。  
然而KafkaConsumer并不是线程安全的，多个线程操作同一个KafkaConsumer实例会出现各种问题，Kafka官方对于Consumer端的多线程处理给出的指导建议如下：  
  
1. 每个线程都持有一个KafkaConsumer对象  
好处：

* 实现简单
* 不需要线程间的协作，效率最高
* 最容易实现每个Partition内消息的顺序处理

弊端：

* 每个KafkaConsumer都要与集群保持一个TCP连接
* 线程数不能超过Partition数
* 每一batch拉取的数据量会变小，对吞吐量有一定影响

2. 解耦，1个Consumer线程负责拉取消息，数个Worker线程负责消费消息  
好处：

* 可自由控制Worker线程的数量，不受Partition数量限制

弊端：

* 消息消费的顺序无法保证
* 难以控制手动提交offset的时机

个人认为第二种方式更加可取，consumer数不能超过partition数这个限制是很要命的，不可能为了提高Consumer消费消息的效率而把Topic分成更多的partition，partition越多，集群的高可用性就越低。

**6. Kafka集群高可用性测试**

1) 查看当前Topic的状态：

/kafka-topics.sh --describe --zookeeper 10.0.0.100:2181,10.0.0.101:2181,10.0.0.102:2181 --topic test

  输出：

Topic:test PartitionCount:2 ReplicationFactor:2 Configs:  
   Topic: test Partition: 0 Leader: 1 Replicas: 1,0 Isr: 0,1  
   Topic: test Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0,1

  可以看到，partition0的leader是broker1，parition1的leader是broker0

2) 启动Producer向Kafka集群发送消息

  输出：

message send to partition 0, offset: 35  
message send to partition 1, offset: 33  
message send to partition 0, offset: 36  
message send to partition 1, offset: 34  
message send to partition 1, offset: 35  
message send to partition 0, offset: 37  
message send to partition 0, offset: 38  
message send to partition 1, offset: 36  
message send to partition 1, offset: 37

3) 登录SSH将broker0，也就是partition 1的leader kill掉

  再次查看Topic状态：

Topic:test PartitionCount:2 ReplicationFactor:2 Configs:  
  Topic: test Partition: 0 Leader: 1 Replicas: 1,0 Isr: 1  
  Topic: test Partition: 1 Leader: 1 Replicas: 0,1 Isr: 1

  可以看到，当前parition0和parition1的leader都是broker1了

  此时再去看Producer的输出：

[kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.common.network.Selector - Connection with /10.0.0.100 disconnected  
java.net.ConnectException: Connection refused: no further information  
    at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)  
    at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:739)  
    at org.apache.kafka.common.network.PlaintextTransportLayer.finishConnect(PlaintextTransportLayer.java:54)  
    at org.apache.kafka.common.network.KafkaChannel.finishConnect(KafkaChannel.java:72)  
    at org.apache.kafka.common.network.Selector.poll(Selector.java:274)  
    at org.apache.kafka.clients.NetworkClient.poll(NetworkClient.java:256)  
    at org.apache.kafka.clients.producer.internals.Sender.run(Sender.java:216)  
    at org.apache.kafka.clients.producer.internals.Sender.run(Sender.java:128)  
    at java.lang.Thread.run(Thread.java:745)  
[kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.clients.Metadata - Updated cluster metadata version 7 to Cluster(nodes = [Node(1, 10.0.0.101, 9092)], partitions = [Partition(topic = test, partition = 1, leader = 1, replicas = [1,], isr = [1,], Partition(topic = test, partition = 0, leader = 1, replicas = [1,], isr = [1,]])

  能看到Producer端的DEBUG日志显示与broker0的链接断开了，此时Kafka立刻开始更新集群metadata，更新后的metadata表示broker1现在是两个partition的leader，Producer进程很快就恢复继续运行，没有漏发任何消息，能够看出Kafka集群的故障切换机制还是很厉害的

4) 我们再把broker0启动起来

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

  然后再次检查Topic状态：

Topic:test PartitionCount:2 ReplicationFactor:2 Configs:  
   Topic: test Partition: 0 Leader: 1 Replicas: 1,0 Isr: 1,0  
   Topic: test Partition: 1 Leader: 1 Replicas: 0,1 Isr: 1,0

  我们看到，broker0启动起来了，并且已经是in-sync状态（注意Isr从1变成了1,0），但此时两个partition的leader还都是 broker1，也就是说当前broker1会承载所有的发送和拉取请求。这显然是不行的，我们要让集群恢复到负载均衡的状态。

  这时候，需要使用Kafka的选举工具触发一次选举：

bin/kafka-preferred-replica-election.sh --zookeeper 10.0.0.100:2181,10.0.0.101:2181,10.0.0.102:2181

  选举完成后，再次查看Topic状态：

Topic:test PartitionCount:2 ReplicationFactor:2 Configs:  
   Topic: test Partition: 0 Leader: 1 Replicas: 1,0 Isr: 1,0  
   Topic: test Partition: 1 Leader: 0 Replicas: 0,1 Isr: 1,0

  可以看到，集群重新回到了broker0挂掉之前的状态

  但此时，Producer端产生了异常：

org.apache.kafka.common.errors.NotLeaderForPartitionException: This server is not the leader for that topic-partition.

  原因是Producer端在尝试向broker1的parition0发送消息时，partition0的leader已经切换成了broker0，所以消息发送失败。

  此时用Consumer去消费消息，会发现消息的编号不连续了，确实漏发了一条消息。这是因为我们在构造Producer时设定了retries=0，所以在发送失败时Producer端不会尝试重发。

  将retries改为3后再次尝试，会发现leader切换时再次发生了同样的问题，但Producer的重发机制起了作用，消息重发成功，启动Consumer端检查也证实了所有消息都发送成功了。

每 次集群单点发生故障恢复后，都需要进行重新选举才能彻底恢复集群的leader分配，如果嫌每次这样做很麻烦，可以在broker的配置文件（即 server.properties）中配置auto.leader.rebalance.enable=true，这样broker在启动后就会自动进 行重新选举

至此，我们通过测试证实了集群出现单点故障和恢复的过程中，Producer端能够保持正确运转。接下来我们看一下Consumer端的表现：

5) 同时启动Producer进程和Consumer进程

  此时Producer一边在生产消息，Consumer一边在消费消息

6) 把broker0干掉，观察Consumer端的输出：

能看到，在broker0挂掉后，consumer也端产生了一系列INFO和WARN输出，但同Producer端一样，若干秒后自动恢复，消息仍然是连续的，并未出现断点。

7) 再次把broker0启动，并触发重新选举，然后观察输出：

fetched from partition 0, offset: 418, message: this is message48  
fetched from partition 0, offset: 419, message: this is message49  
[main] INFO org.apache.kafka.clients.consumer.internals.ConsumerCoordinator - Offset commit for group 1 failed due to NOT\_COORDINATOR\_FOR\_GROUP, will find new coordinator and retry  
[main] INFO org.apache.kafka.clients.consumer.internals.AbstractCoordinator - Marking the coordinator 2147483646 dead.  
[main] WARN org.apache.kafka.clients.consumer.internals.ConsumerCoordinator - Auto offset commit failed: This is not the correct coordinator for this group.  
fetched from partition 1, offset: 392, message: this is message50  
fetched from partition 0, offset: 420, message: this is message51

  能看到，重选举后Consumer端也输出了一些日志，意思是在提交offset时发现当前的调度器已经失效了，但很快就重新获取了新的有效调度器，恢复 了offset的自动提交，验证已提交offset的值也证明了offset提交并未因leader切换而发生错误。

  如上，我们也通过测试证实了Kafka集群出现单点故障时，Consumer端的功能正确性。