Kafka（讲师：OldJia）

课程介绍

第一部分 Kafka架构与实战

1.1 概念和基本架构

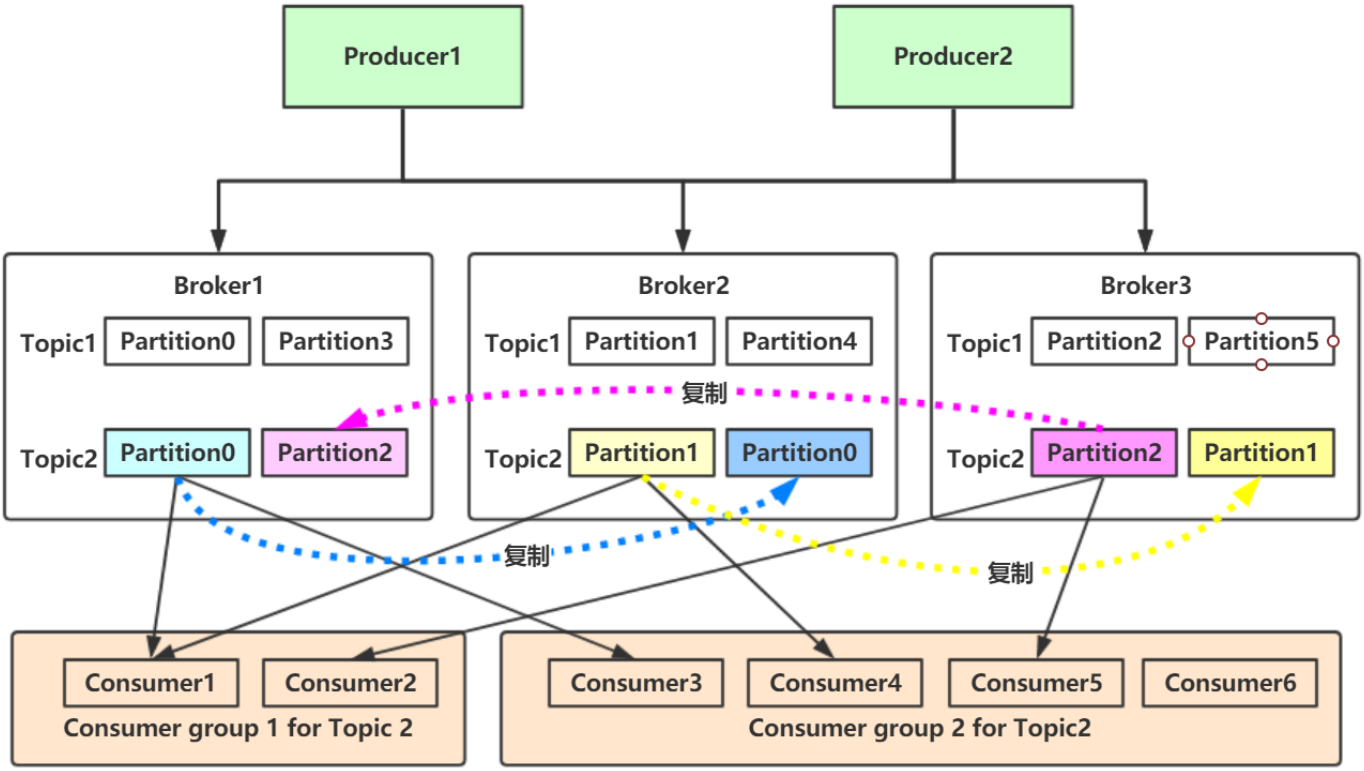
1.1.1 Kafka介绍

Kafka是最初由Linkedin公司开发，是一个分布式、分区的、多副本的、多生产者、多订阅者，基于zookeeper协调的分布式日志系统（也可以当做MQ系统），常见可以用于web/nginx日志、访问日志，消息服务等等，Linkedin于2010年贡献给了Apache基金会并成为顶级开源项目。

主要应用场景是：日志收集系统和消息系统。

Kafka主要设计目标如下：

- 以时间复杂度为O(1)的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间的访问性能。
- 高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒100K条消息的传输。
- 支持Kafka Server间的消息分区，及分布式消费，同时保证每个partition内的消息顺序传输。
- 同时支持离线数据处理和实时数据处理。
- 支持在线水平扩展



有两种主要的消息传递模式：**点对点传递模式、发布-订阅模式**。大部分的消息系统选用发布-订阅模式。**Kafka就是一种发布-订阅模式**。

对于消息中间件，消息分推拉两种模式。Kafka只有消息的拉取，没有推送，可以通过轮询实现消息的推送。

1. Kafka在一个或多个可以跨越多个数据中心的服务器上作为集群运行。
2. Kafka集群中按照主题分类管理，一个主题可以有多个分区，一个分区可以有多个副本分区。
3. 每个记录由一个键，一个值和一个时间戳组成。

Kafka具有四个核心API：

1. Producer API：允许应用程序将记录流发布到一个或多个Kafka主题。
2. Consumer API：允许应用程序订阅一个或多个主题并处理为其生成的记录流。
3. Streams API：允许应用程序充当流处理器，使用一个或多个主题的输入流，并生成一个或多个输出主题的输出流，从而有效地将输入流转换为输出流。
4. Connector API：允许构建和运行将Kafka主题连接到现有应用程序或数据系统的可重用生产者或使用者。例如，关系数据库的连接器可能会捕获对表的所有更改。

1.1.2 Kafka优势

1. 高吞吐量：单机每秒处理几十上百万的消息量。即使存储了许多TB的消息，它也保持稳定的性能。
2. 高性能：单节点支持上千个客户端，并保证零停机和零数据丢失。
3. 持久化数据存储：将消息持久化到磁盘。通过将数据持久化到硬盘以及replication防止数据丢失。
    1. 零拷贝

    2. 顺序读，顺序写
    3. 利用Linux的页缓存
4. 分布式系统，易于向外扩展。所有的Producer、Broker和Consumer都会有多个，均为分布式的。无需停机即可扩展机器。多个Producer、Consumer可能是不同的应用。
5. 可靠性 - Kafka是分布式，分区，复制和容错的。
6. 客户端状态维护：消息被处理的状态是在Consumer端维护，而不是由server端维护。当失败时能自动平衡。
7. 支持online和offline的场景。
8. 支持多种客户端语言。Kafka支持Java、.NET、PHP、Python等多种语言。

1.1.3 Kafka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；

消息系统：解耦生产者和消费者、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，然后消费者通过订阅这些Topic来做实时的监控分析，亦可保存到数据库；

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和Storm。

1.1.4 基本架构

消息和批次

Kafka的数据单元称为消息。可以把消息看成是数据库里的一个“数据行”或一条“记录”。消息由字节数组组成。

消息有键，键也是一个字节数组。当消息以一种可控的方式写入不同的分区时，会用到键。

为了提高效率，消息被分批写入Kafka。批次就是一组消息，这些消息属于同一个主题和分区。

把消息分成批次可以减少网络开销。批次越大，单位时间内处理的消息就越多，单个消息的传输时间就越长。批次数据会被压缩，这样可以提升数据的传输和存储能力，但是需要更多的计算处理。

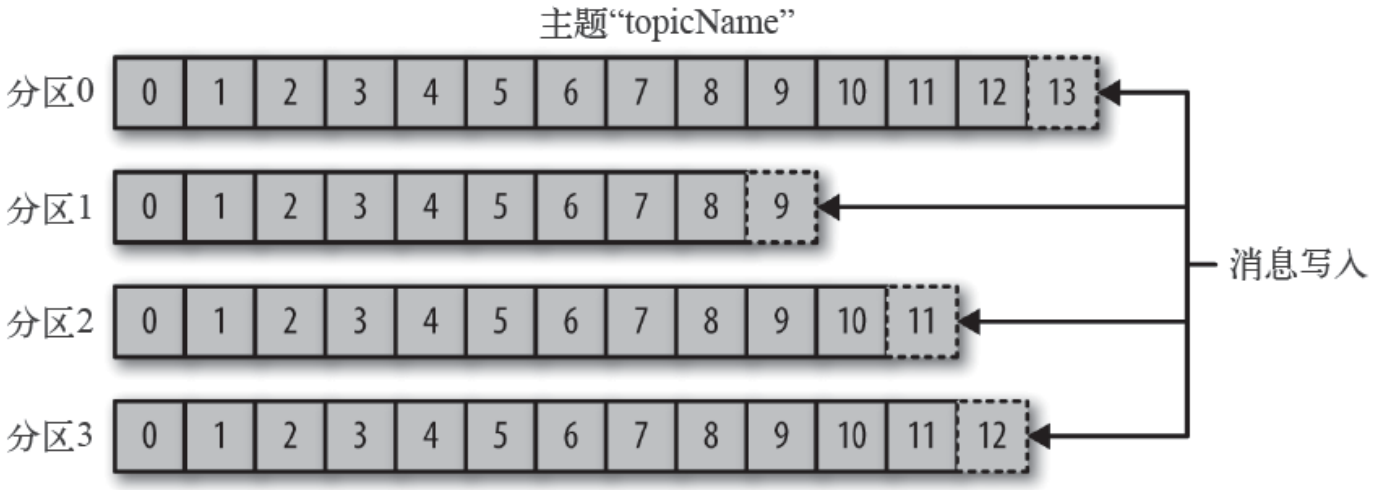
模式

消息模式（schema）有许多可用的选项，以便于理解。如JSON和XML，但是它们缺乏强类型处理能力。Kafka的许多开发者喜欢使用Apache Avro。Avro提供了一种紧凑的序列化格式，模式和消息体分开。当模式发生变化时，不需要重新生成代码，它还支持强类型和模式进化，其版本既向前兼容，也向后兼容。

数据格式的一致性对Kafka很重要，因为它消除了消息读写操作之间的耦合性。

主题和分区

Kafka的消息通过主题进行分类。主题可比是数据库的表或者文件系统里的文件夹。主题可以被分为若干分区，一个主题通过分区分布于Kafka集群中，提供了横向扩展的能力。



生产者和消费者

生产者创建消息。消费者消费消息。

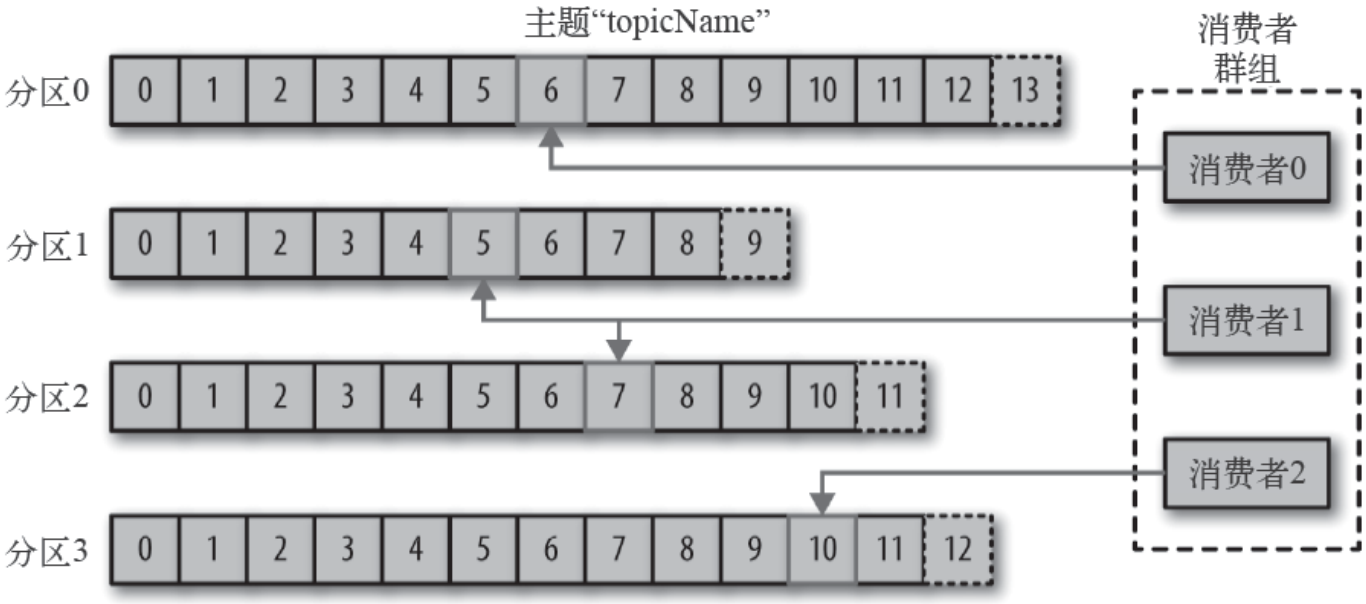
一个消息被发布到一个特定的主题上。

生产者在默认情况下把消息均衡地分布到主题的所有分区上：

1. 直接指定消息的分区
2. 根据消息的key散列取模得出分区
3. 轮询指定分区。

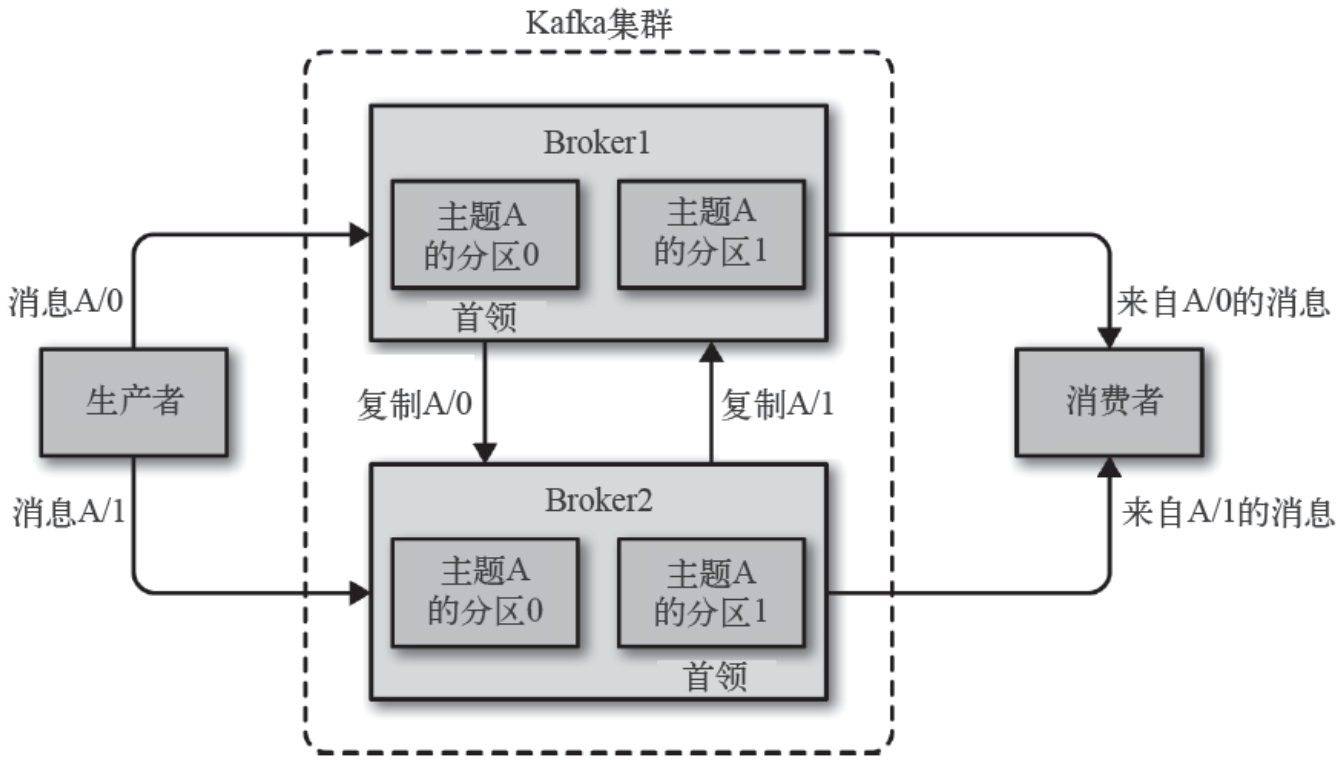
消费者通过偏移量来区分已经读过的消息，从而消费消息。

消费者是消费组的一部分。消费组保证每个分区只能被一个消费者使用，避免重复消费。



broker和集群

一个独立的Kafka服务器称为broker。broker接收来自生产者的消息，为消息设置偏移量，并提交消息到磁盘保存。broker为消费者提供服务，对读取分区的请求做出响应，返回已经提交到磁盘上的消息。**单个broker**可以轻松处理**数千个分区**以及**每秒百万级**的消息量。



每个集群都有一个broker是集群控制器（自动从集群的活跃成员中选举出来）。

控制器负责管理工作：

- 将分区分配给broker
- 监控broker

集群中一个分区属于一个**broker**，该broker称为**分区首领**。

一个分区可以分配给**多个broker**，此时会发生分区复制。

分区的复制提供了**消息冗余，高可用**。**副本分区**不负责处理消息的读写。

1.1.5 核心概念

1.1.5.1 Producer

生产者创建消息。

该角色将消息发布到Kafka的topic中。broker接收到生产者发送的消息后，broker将该消息**追加**到当前用于追加数据的`segment`文件中。

一般情况下，一个消息会被发布到一个特定的主题上。

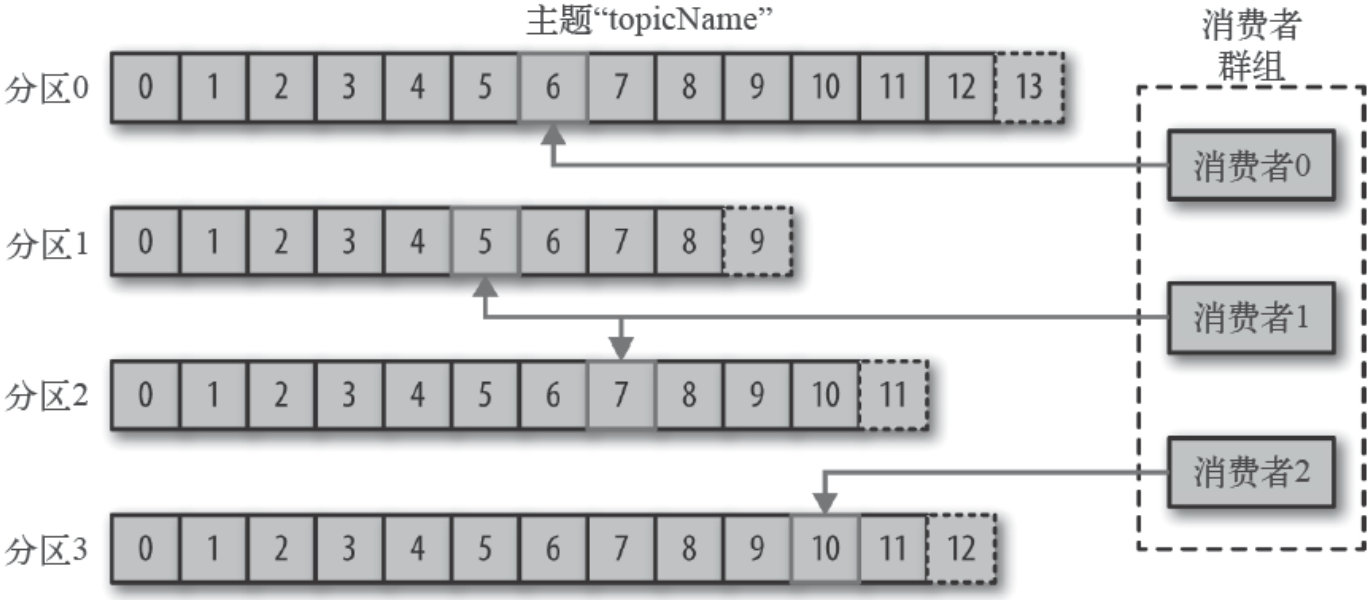
1. 默认情况下通过轮询把消息均衡地分布到主题的所有分区上。
2. 在某些情况下，生产者会把消息直接写到指定的分区。这通常是通过消息键和分区器来实现的，分区器为键生成一个散列值，并将其映射到指定的分区上。这样可以保证包含同一个键的消息会被写到同一个分区上。

3. 生产者也可以使用自定义的分区器，根据不同的业务规则将消息映射到分区。

1.1.5.2 Consumer

消费者读取消息。

1. 消费者订阅一个或多个主题，并按照消息生成的顺序读取它们。
2. 消费者通过检查消息的**偏移量**来区分已经读取过的消息。偏移量是另一种元数据，它是一个**不断递增的整数值**，在创建消息时，Kafka 会把它添加到消息里。在给定的**分区**里，每个消息的偏移量都是**唯一**的。消费者把每个分区最后读取的消息偏移量保存在Zookeeper 或**Kafka** 上，如果消费者关闭或重启，它的读取状态不会丢失。
3. 消费者是**消费组**的一部分。群组保证每个分区只能被一个消费者使用。
4. 如果一个消费者失效，消费组里的其他消费者可以接管失效消费者的工作，再平衡，分区重新分配。



1.1.5.3 Broker

一个独立的Kafka 服务器被称为broker。

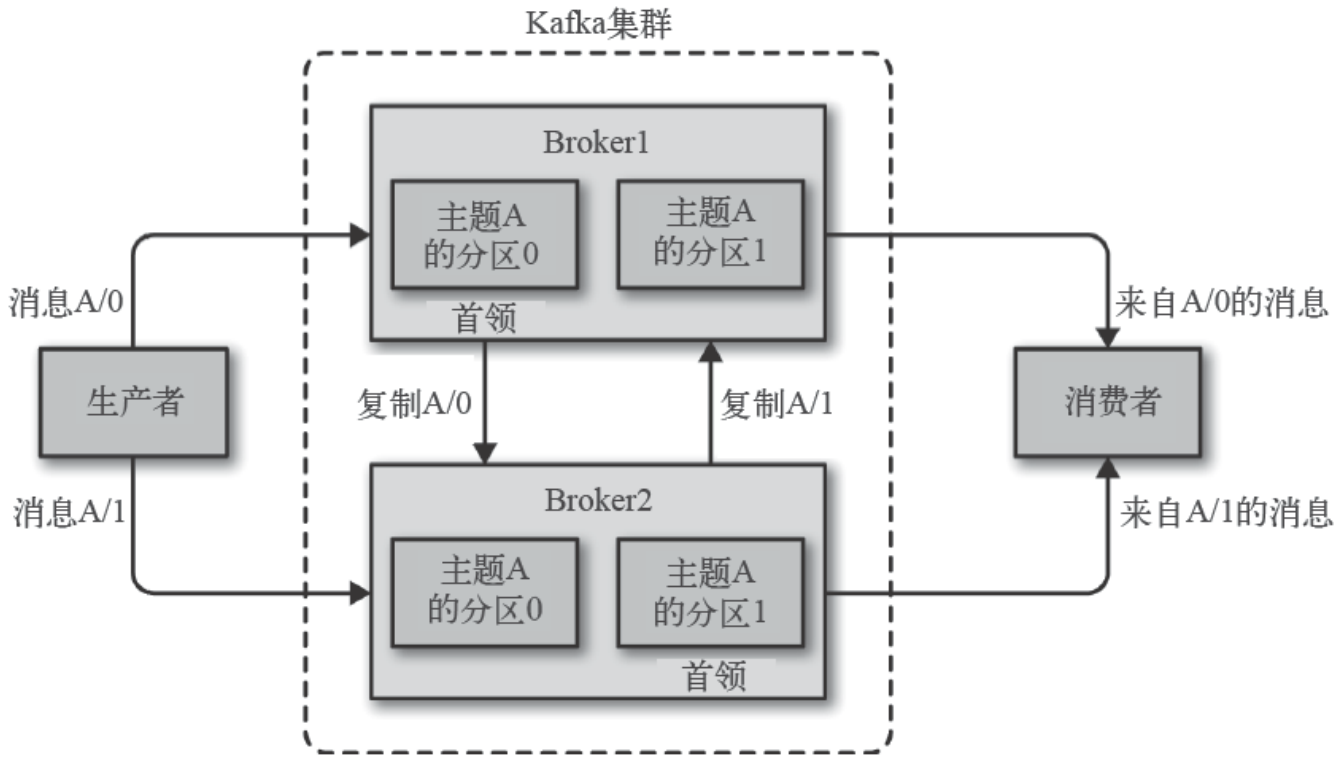
broker 为消费者提供服务，对读取分区的请求作出响应，返回已经提交到磁盘上的消息。

1. 如果某topic有N个partition，集群有N个broker，那么每个broker存储该topic的一个partition。
2. 如果某topic有N个partition，集群有(N+M)个broker，那么其中有N个broker存储该topic的一个partition，剩下的M个broker不存储该topic的partition数据。
3. 如果某topic有N个partition，集群中broker数目少于N个，那么一个broker存储该topic的一个或多个partition。在实际生产环境中，尽量避免这种情况的发生，这种情况容易导致Kafka集群数据不均衡。

broker 是集群的组成部分。每个集群都有一个broker 同时充当了集群控制器的角色（自动从集群的活跃成员中选举出来）。

控制器负责管理工作，包括将分区分配给broker 和监控broker。

在集群中，一个分区从属于一个broker，该broker 被称为分区的首领。



1.1.5.4 Topic

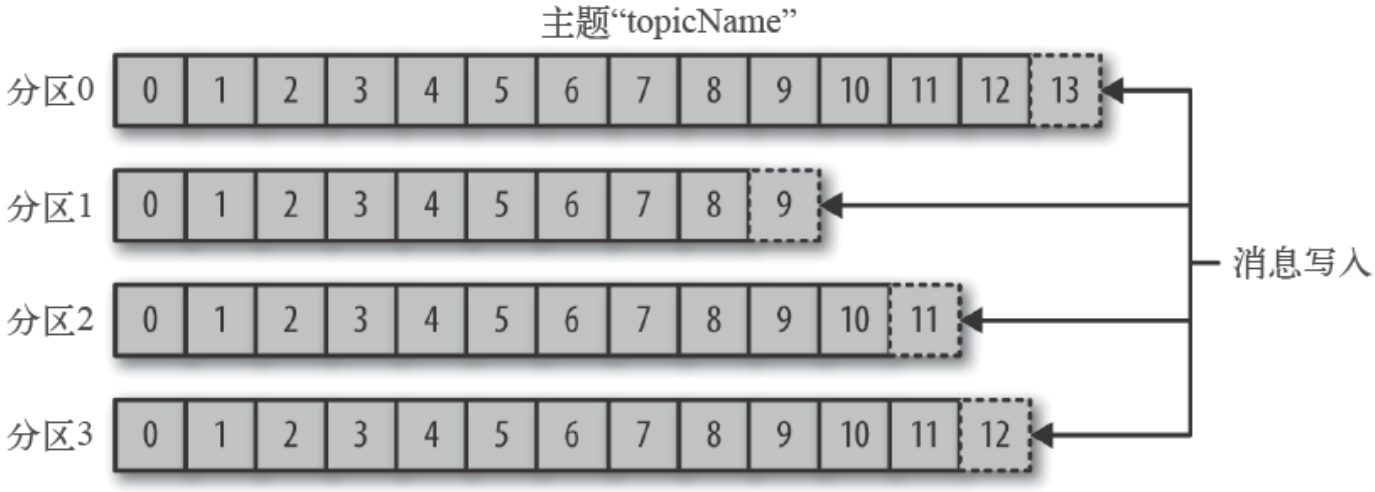
每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic。

物理上不同Topic的消息分开存储。

主题就好比数据库的表，尤其是分库分表之后的逻辑表。

1.1.5.5 Partition

1. 主题可以被分为若干个分区，一个分区就是一个提交日志。
2. 消息以追加的方式写入分区，然后以先入先出的顺序读取。
3. 无法在整个主题范围内保证消息的顺序，但可以保证消息在单个分区内的顺序。
4. Kafka 通过分区来实现数据冗余和伸缩性。
5. 在需要严格保证消息的消费顺序的场景下，需要将partition数目设为1。



1.1.5.6 Replicas

Kafka 使用主题来组织数据，每个主题被分为若干个分区，每个分区有多个副本。那些副本被保存在broker 上，每个broker 可以保存成百上千个属于不同主题和分区的副本。

副本有以下两种类型：

首领副本

每个分区都有一个首领副本。为了保证一致性，所有生产者请求和消费者请求都会经过这个副本。

跟随者副本

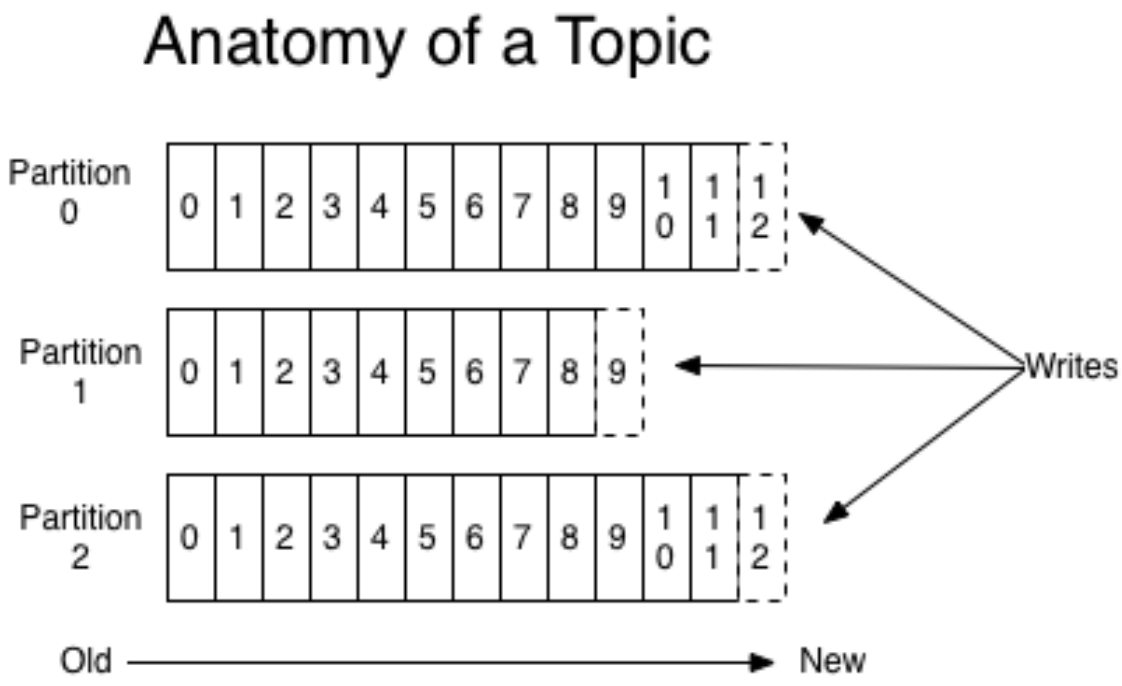
首领以外的副本都是跟随者副本。跟随者副本不处理来自客户端的请求，它们唯一的任务就是从首领那里复制消息，保持与首领一致的状态。如果首领发生崩溃，其中的一个跟随者会被提升为新首领。

1.1.5.7 Offset

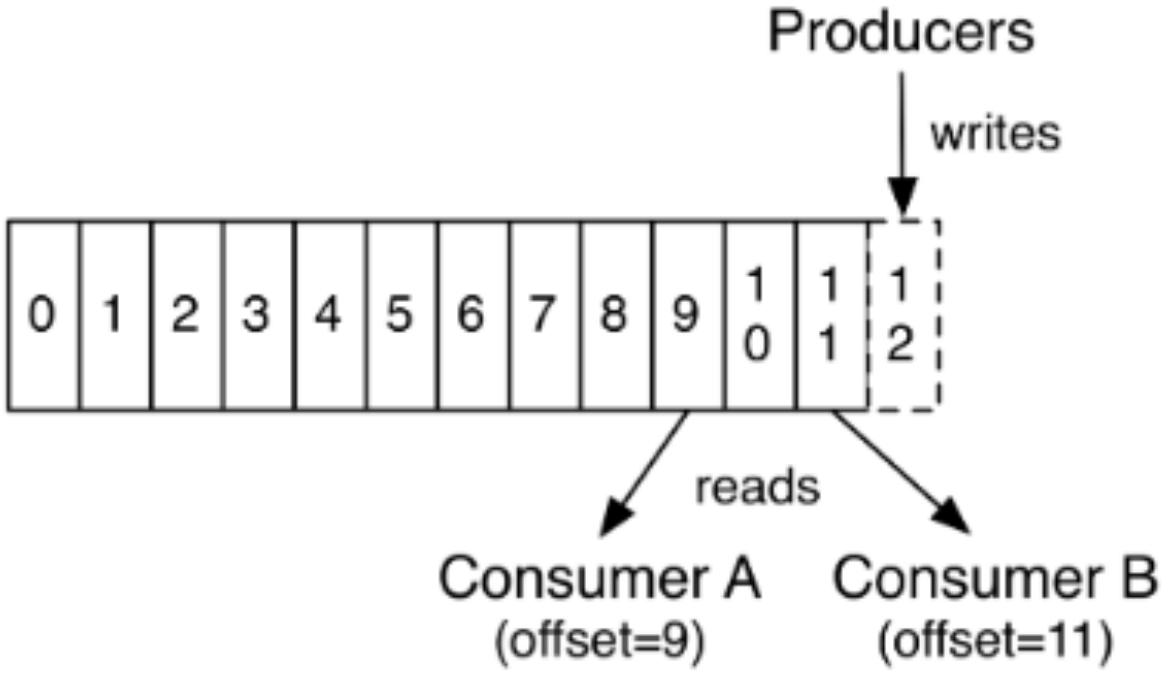
生产者Offset

消息写入的时候，每一个分区都有一个offset，这个offset就是生产者的offset，同时也是这个分区的最新最大的offset。

有些时候没有指定某一个分区的offset，这个工作kafka帮我们完成。



消费者Offset



这是某一个分区的offset情况，生产者写入的offset是最新最大的值是12，而当Consumer A进行消费时，从0开始消费，一直消费到了9，消费者的offset就记录在9，Consumer B就纪录在了11。等下一次他们再来消费时，他们可以选择接着上一次的位置消费，当然也可以选择从头消费，或者跳到最近的记录并从“现在”开始消费。

1.1.5.8 副本

Kafka通过副本保证高可用。副本分为首领副本(Leader)和跟随者副本(Follower)。

跟随者副本包括同步副本和不同步副本，在发生首领副本切换的时候，只有同步副本可以切换为首领副本。

1.1.5.8.1 AR

分区中的所有副本统称为**AR**（Assigned Repllicas）。

AR=ISR+OSR

1.1.5.8.2 ISR

所有与leader副本保持一定程度同步的副本（包括Leader）组成**ISR**（In-Sync Replicas），ISR集合是AR集合中的一个子集。消息会先发送到leader副本，然后follower副本才能从leader副本中拉取消息进行同步，同步期间内follower副本相对于leader副本而言会有一定程度的滞后。前面所说的“一定程度”是指可以忍受的滞后范围，这个范围可以通过参数进行配置。

1.1.5.8.3 OSR

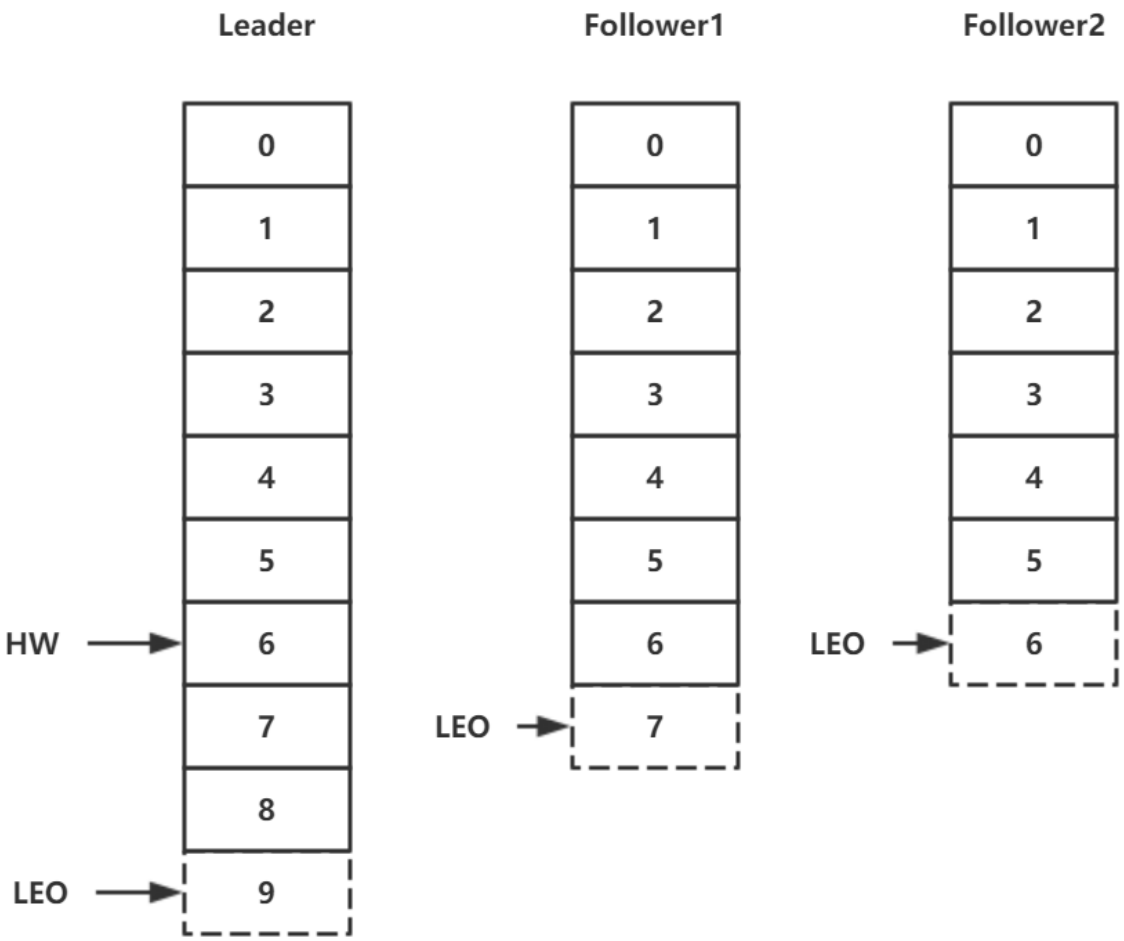
与leader副本同步滞后过多的副本（不包括leader）副本，组成**OSR**(Out-Sync Relipcas)。在正常情况下，所有的follower副本都应该与leader副本保持一定程度的同步，即AR=ISR,OSR集合为空。

1.1.5.8.4 HW

HW是High Watermak的缩写， 俗称高水位，它表示了一个特定消息的偏移量（offset），消费之只能拉取到这个offset之前的消息。

1.1.5.8.5 LEO

LEO是Log End Offset的缩写，它表示了当前日志文件中**下一条待写入**消息的offset。



1.2 Kafka安装与配置

1.2.1 Java环境为前提

1、上传jdk-8u261-linux-x64.rpm到服务器并安装：

```
rpm -ivh jdk-8u261-linux-x64.rpm
```

2、配置环境变量：

```
vim /etc/profile
```

```
78 export JAVA_HOME=/usr/java/jdk1.8.0_261-amd64
79 export PATH=$PATH:$JAVA_HOME/bin
```

```
# 生效
source /etc/profile
# 验证
java -version
```

```
[root@node1 ~]# java -version
java version "1.8.0_261"
Java(TM) SE Runtime Environment (build 1.8.0_261-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.261-b12, mixed mode)
[root@node1 ~]#
```

1.2.2 Zookeeper的安装配置

1、上传zookeeper-3.4.14.tar.gz到服务器

2、解压到/opt：

```
tar -zxf zookeeper-3.4.14.tar.gz -C /opt
cd /opt/zookeeper-3.4.14/conf
# 复制zoo_sample.cfg命名为zoo.cfg
cp zoo_sample.cfg zoo.cfg
# 编辑zoo.cfg文件
vim zoo.cfg
```

3、修改Zookeeper保存数据的目录，dataDir：

- dataDir=/var/lagou/zookeeper/data

4、编辑/etc/profile：

- 设置环境变量ZOO_LOG_DIR，指定Zookeeper保存日志的位置；
- ZOOKEEPER_PREFIX指向Zookeeper的解压目录；
- 将Zookeeper的bin目录添加到PATH中：

```
export ZOOKEEPER_PREFIX=/opt/zookeeper-3.4.14
export PATH=$PATH:$ZOOKEEPER_PREFIX/bin
export ZOO_LOG_DIR=/var/lagou/zookeeper/log
```

5、使配置生效：

```
source /etc/profile
```

6、验证：

```
[root@node1 ~]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.14/bin/../conf/zoo.cfg
Error contacting service. It is probably not running.
[root@node1 ~]#
```

1.2.3 Kafka的安装与配置

1、上传kafka_2.12-1.0.2.tgz到服务器并解压：

```
tar -zxf kafka_2.12-1.0.2.tgz -C /opt
```

2、配置环境变量并生效：

```
vim /etc/profile
```

```
[root@node1 config]# tail -n 2 /etc/profile
export KAFKA_HOME=/opt/kafka_2.12-1.0.2
export PATH=$PATH:$KAFKA_HOME/bin
[root@node1 config]#
```

3、配置/opt/kafka_2.12-1.0.2/config中的server.properties文件：

Kafka连接Zookeeper的地址，此处使用本地启动的Zookeeper实例，连接地址是localhost:2181，后面的`myKafka`是Kafka在Zookeeper中的根节点路径：

```
############################# 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=localhost:2181/myKafka
```

配置kafka存储持久化数据的目录

log.dir=/var/lagou/kafka/kafka-logs

```
mkdir -p /var/lagou/kafka/kafka-logs
```

4、启动Zookeeper：

```
zkServer.sh start
```

5、确认Zookeeper的状态：

```
[root@node1 ~]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.14/bin/../conf/zoo.cfg
Mode: standalone
[root@node1 ~]#
```

6、启动Kafka：

进入Kafka安装的根目录，执行如下命令：

```
[root@node1 kafka_2.12-1.0.2]# pwd
/opt/kafka_2.12-1.0.2
[root@node1 kafka_2.12-1.0.2]# kafka-server-start.sh config/server.properties
```

启动成功，可以看到控制台输出的最后一行的started状态：

```
INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
```

7、查看Zookeeper的节点：

```
[zk: localhost:2181(CONNECTED) 0] ls /
[myKafka, zookeeper]
[zk: localhost:2181(CONNECTED) 1] ls /myKafka
[cluster, controller_epoch, controller, brokers, admin, isr_change_notification, consumers, log_dir_event_notification, latest_producer_id_block, config]
[zk: localhost:2181(CONNECTED) 2]
```

8、此时Kafka是前台模式启动，要停止，使用Ctrl+C。

如果要后台启动，使用命令：

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

```
[root@node1 kafka_2.12-1.0.2]# kafka-server-start.sh
USAGE: /opt/kafka_2.12-1.0.2/bin/kafka-server-start.sh [-daemon] server.properties [--override property=value]*
```

查看Kafka的后台进程：

```
ps aux | grep kafka
```

停止后台运行的Kafka：

```
[root@node1 kafka_2.12-1.0.2]# kafka-server-stop.sh
[root@node1 kafka_2.12-1.0.2]# ps aux | grep kafka
root      40934  0.0  0.0 112708   976 pts/0    R+   00:55   0:00 grep --color=auto kafka
[root@node1 kafka_2.12-1.0.2]#
```

1.2.4 生产与消费

1、kafka-topics.sh 用于管理主题。

```
# 列出现有的主题
[root@node1 ~]# kafka-topics.sh --list --zookeeper localhost:2181/myKafka
# 创建主题，该主题包含一个分区，该分区为Leader分区，它没有Follower分区副本。
[root@node1 ~]# kafka-topics.sh --zookeeper localhost:2181/myKafka --create --topic topic_1 --partitions 1 --replication-factor 1
# 查看分区信息
[root@node1 ~]# kafka-topics.sh --zookeeper localhost:2181/myKafka --list
# 查看指定主题的详细信息
[root@node1 ~]# kafka-topics.sh --zookeeper localhost:2181/myKafka --describe --topic topic_1
# 删除指定主题
[root@node1 ~]# kafka-topics.sh --zookeeper localhost:2181/myKafka --delete --topic topic_1
```

2、kafka-console-producer.sh用于生产消息：

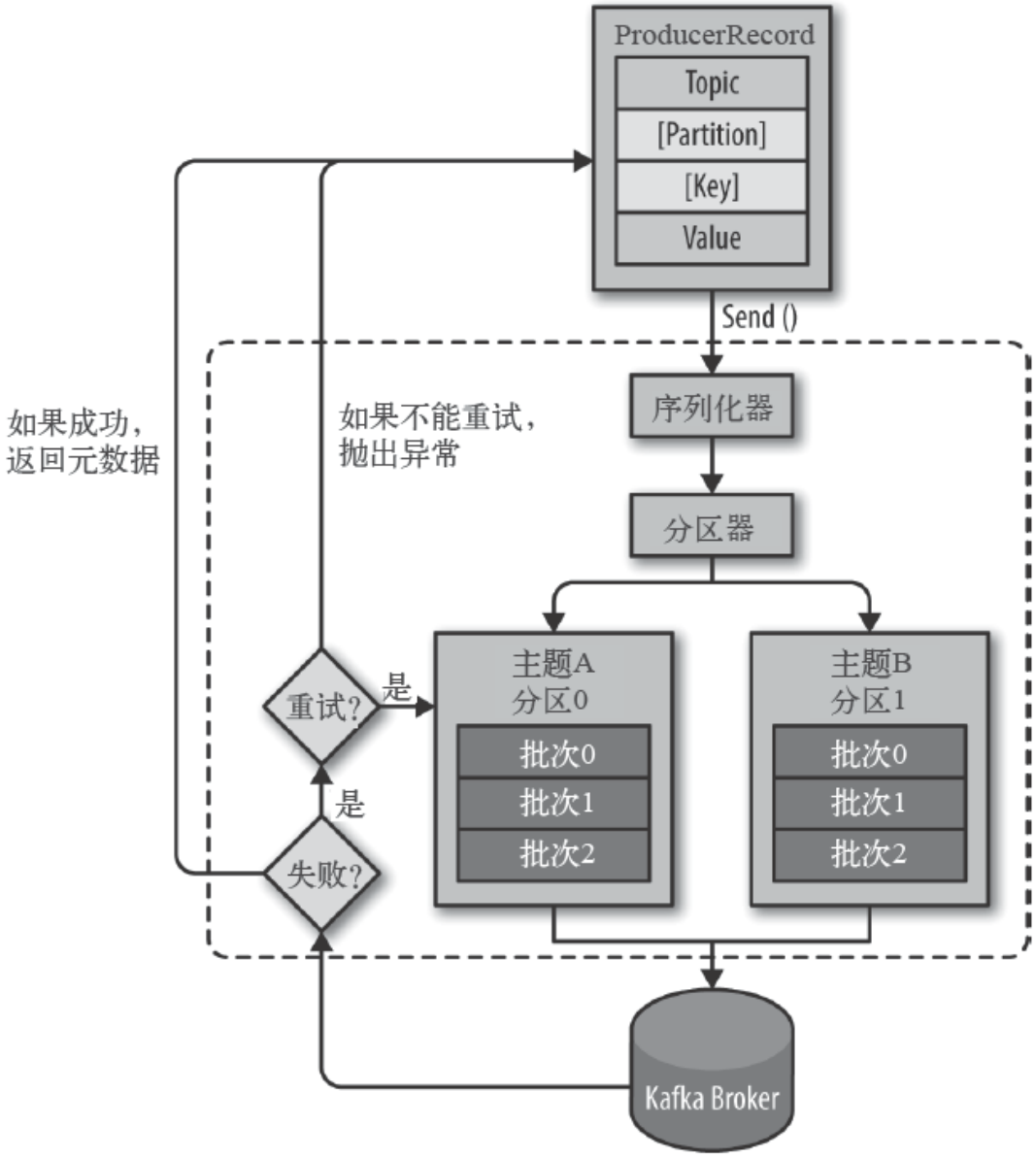
```
# 开启生产者
[root@node1 ~]# kafka-console-producer.sh --topic topic_1 --broker-list localhost:9020
```

3、kafka-console-consumer.sh用于消费消息：

```
# 开启消费者
[root@node1 ~]# kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic topic_1
# 开启消费者方式二，从头消费，不按照偏移量消费
[root@node1 ~]# kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic topic_1 --from-beginning
```

1.3 Kafka开发实战

1.3.1 消息的发送与接收



生产者主要的对象有：`KafkaProducer`，`ProducerRecord`。

其中`KafkaProducer`是用于发送消息的类，`ProducerRecord`类用于封装Kafka的消息。

`KafkaProducer`的创建需要指定的参数和含义：

参数	说明
bootstrap.servers	配置生产者如何与broker建立连接。该参数设置的是初始化参数。如果生产者需要连接的是Kafka集群，则这里配置集群中几个broker的地址，而不是全部，当生产者连接上此处指定的broker之后，在通过该连接发现集群中的其他节点。
key.serializer	要发送信息的key数据的序列化类。设置的时候可以写类名，也可以使用该类的Class对象。
value.serializer	要发送消息的alue数据的序列化类。设置的时候可以写类名，也可以使用该类的Class对象。
acks	默认值：**all**。 acks=0： 生产者不等待broker对消息的确认，只要将消息放到缓冲区，就认为消息已经发送完成。 该情形不能保证broker是否真的收到了消息，retries配置也不会生效。发送的消息的返回的消息偏移量永远是-1。 acks=1 表示消息只需要写到主分区即可，然后就响应客户端，而不等待副本分区的确认。 在该情形下，如果主分区收到消息确认之后就宕机了，而副本分区还没来得及同步该消息，则该消息丢失。 acks=all 首领分区会等待所有的ISR副本分区确认记录。 该处理保证了只要有一个ISR副本分区存活，消息就不会丢失。 这是Kafka最强的可靠性保证，等效于`acks=-1`
retries	retries重试次数 当消息发送出现错误的时候，系统会重发消息。 跟客户端收到错误时重发一样。 如果设置了重试，还想保证消息的有序性，需要设置MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION=1 否则在重试此失败消息的时候，其他的消息可能发送成功了

其他参数可以从`org.apache.kafka.clients.producer.ProducerConfig`中找到。我们后面的内容会介绍到。

消费者生产消息后，需要broker端的确认，可以同步确认，也可以异步确认。

同步确认效率低，异步确认效率高，但是需要设置回调对象。

生产者：

```
1  package com.lagou.kafka.demo.producer;
2
3  import org.apache.kafka.clients.producer.KafkaProducer;
4  import org.apache.kafka.clients.producer.ProducerRecord;
5
6  import java.util.HashMap;
7  import java.util.Map;
8  import java.util.concurrent.ExecutionException;
9  import java.util.concurrent.TimeUnit;
10 import java.util.concurrent.TimeoutException;
11
12 public class MyProducer1 {
13     public static void main(String[] args) throws InterruptedException,
   ExecutionException, TimeoutException {
```

```
14
15          Map<String, Object> configs = new HashMap<>();
16  //          设置连接Kafka的初始连接用到的服务器地址
17  //          如果是集群，则可以通过此初始连接发现集群中的其他broker
18          configs.put("bootstrap.servers", "node1:9092");
19  //          设置key的序列化器
20          configs.put("key.serializer",
    "org.apache.kafka.common.serialization.IntegerSerializer");
21  //          设置value的序列化器
22          configs.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
23          configs.put("acks", "1");
24
25          KafkaProducer<Integer, String> producer = new KafkaProducer<Integer, String>
    (configs);
26
27          // 用于封装Producer的消息
28          ProducerRecord<Integer, String> record = new ProducerRecord<Integer, String>(
29                  "topic_1",    // 主题名称
30                  0,          // 分区编号，现在只有一个分区，所以是0
31                  0,              // 数字作为key
32                  "message 0"  // 字符串作为value
33          );
34
35          // 发送消息，同步等待消息的确认
36          producer.send(record).get(3_000, TimeUnit.MILLISECONDS);
37
38          // 关闭生产者
39          producer.close();
40      }
41  }
```

生产者2：

```
1   package com.lagou.kafka.demo.producer;
2
3   import org.apache.kafka.clients.producer.Callback;
4   import org.apache.kafka.clients.producer.KafkaProducer;
5   import org.apache.kafka.clients.producer.ProducerRecord;
6   import org.apache.kafka.clients.producer.RecordMetadata;
7
8   import java.util.HashMap;
9   import java.util.Map;
10
11  public class MyProducer2 {
12      public static void main(String[] args) {
13          Map<String, Object> configs = new HashMap<>();
14          configs.put("bootstrap.servers", "node1:9092");
```

```
        configs.put("key.serializer",
"org.apache.kafka.common.serialization.IntegerSerializer");
        configs.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        KafkaProducer<Integer, String> producer = new KafkaProducer<Integer, String>
(configs);

        ProducerRecord<Integer, String> record = new ProducerRecord<Integer, String>(
                "topic_1",
                0,
                1,
                "lagou message 2"
        );

//          使用回调异步等待消息的确认
        producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata metadata, Exception exception) {
                if (exception == null) {
                    System.out.println(
                                "主题: " + metadata.topic() + "\n"
                                + "分区: " + metadata.partition() + "\n"
                                + "偏移量: " + metadata.offset() + "\n"
                                + "序列化的key字节: " + metadata.serializedKeySize() +
"\n"
                                + "序列化的value字节: " +
metadata.serializedValueSize() + "\n"
                                + "时间戳: " + metadata.timestamp()
                    );
                } else {
                    System.out.println("有异常: " + exception.getMessage());
                }
            }
        });
        // 关闭连接
        producer.close();
    }
}
```

生产者3:

```
package com.lagou.kafka.demo.producer;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

```

```java
import java.util.HashMap;
import java.util.Map;

public class MyProducer3 {
    public static void main(String[] args) {
        Map<String, Object> configs = new HashMap<>();
        configs.put("bootstrap.servers", "node1:9092");
        configs.put("key.serializer",
"org.apache.kafka.common.serialization.IntegerSerializer");
        configs.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        KafkaProducer<Integer, String> producer = new KafkaProducer<Integer, String>
(configs);


        for (int i = 100; i < 200; i++) {
            ProducerRecord<Integer, String> record = new ProducerRecord<Integer,
String>(
                    "topic_1",
                    0,
                    i,
                    "lagou message " + i
            );
    //         使用回调异步等待消息的确认
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception
exception) {
                    if (exception == null) {
                        System.out.println(
                                "主题: " + metadata.topic() + "\n"
                                + "分区: " + metadata.partition() + "\n"
                                + "偏移量: " + metadata.offset() + "\n"
                                + "序列化的key字节: " +
metadata.serializedKeySize() + "\n"
                                + "序列化的value字节: " +
metadata.serializedValueSize() + "\n"
                                + "时间戳: " + metadata.timestamp()
                        );
                    } else {
                        System.out.println("有异常: " + exception.getMessage());
                    }
                }
            });

        }
        // 关闭连接
        producer.close();
    }
}
```

消息消费流程：

消费者：

```
package com.lagou.kafka.demo.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.lang.reflect.Array;
import java.util.*;
import java.util.regex.Pattern;

public class MyConsumer1 {
    public static void main(String[] args) {

        Map<String, Object> configs = new HashMap<>();
//        指定bootstrap.servers属性作为初始化连接Kafka的服务器。
//        如果是集群，则会基于此初始化连接发现集群中的其他服务器。
        configs.put("bootstrap.servers", "node1:9092");
//        key的反序列化器
        configs.put("key.deserializer",
"org.apache.kafka.common.serialization.IntegerDeserializer");
//        value的反序列化器
        configs.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        configs.put("group.id", "consumer.demo");

//        创建消费者对象
        KafkaConsumer<Integer, String> consumer = new KafkaConsumer<Integer, String>
(configs);

//        final Pattern pattern = Pattern.compile("topic_\\d");
        final Pattern pattern = Pattern.compile("topic_[0-9]");

        // 消费者订阅主题或分区
//        consumer.subscribe(pattern);

//        consumer.subscribe(pattern, new ConsumerRebalanceListener() {

```

```
        final List<String> topics = Arrays.asList("topic_1");
        consumer.subscribe(topics, new ConsumerRebalanceListener() {
            @Override
            public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                partitions.forEach(tp -> {
                    System.out.println("剥夺的分区: " + tp.partition());
                });
            }

            @Override
            public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
                partitions.forEach(tp -> {
                    System.out.println(tp.partition());
                });
            }
        });
//          拉取订阅主题的消息
        final ConsumerRecords<Integer, String> records = consumer.poll(3_000);

        // 获取topic_1主题的消息
        final Iterable<ConsumerRecord<Integer, String>> topic1Iterable =
records.records("topic_1");

//        遍历topic_1主题的消息
        topic1Iterable.forEach(record -> {
            System.out.println("=========================================");
            System.out.println("消息头字段: " +
Arrays.toString(record.headers().toArray()));
            System.out.println("消息的key: " + record.key());
            System.out.println("消息的偏移量: " + record.offset());
            System.out.println("消息的分区号: " + record.partition());
            System.out.println("消息的序列化key字节数: " + record.serializedKeySize());
            System.out.println("消息的序列化value字节数: " +
record.serializedValueSize());
            System.out.println("消息的时间戳: " + record.timestamp());
            System.out.println("消息的时间戳类型: " + record.timestampType());
            System.out.println("消息的主题: " + record.topic());
            System.out.println("消息的值: " + record.value());
        });

        // 关闭消费者
        consumer.close();
    }
}
```

1.3.2 SpringBoot Kafka

1. pom.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.8.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.lagou.kafka.demo</groupId>
    <artifactId>demo-02-springboot</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>demo-02-springboot</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka-test</artifactId>
            <scope>test</scope>
```

```
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

2. application.properties

```
spring.application.name=springboot-kafka-02
server.port=8080

# 用于建立初始连接的broker地址
spring.kafka.bootstrap-servers=node1:9092
# producer用到的key和value的序列化类
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.IntegerSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
# 默认的批处理记录数
spring.kafka.producer.batch-size=16384
# 32MB的总发送缓存
spring.kafka.producer.buffer-memory=33554432

# consumer用到的key和value的反序列化类
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.IntegerDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
# consumer的消费组id
spring.kafka.consumer.group-id=spring-kafka-02-consumer
# 是否自动提交消费者偏移量
spring.kafka.consumer.enable-auto-commit=true
# 每隔100ms向broker提交一次偏移量
spring.kafka.consumer.auto-commit-interval=100
# 如果该消费者的偏移量不存在，则自动设置为最早的偏移量
spring.kafka.consumer.auto-offset-reset=earliest
```

3. Demo02SpringbootApplication.java

```java
package com.lagou.kafka.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Demo02SpringbootApplication {

    public static void main(String[] args) {
        SpringApplication.run(Demo02SpringbootApplication.class, args);
    }

}
```

4. KafkaConfig.java

```java
package com.lagou.kafka.demo.config;

import org.apache.kafka.clients.admin.NewTopic;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class KafkaConfig {

    @Bean
    public NewTopic topic1() {
        return new NewTopic("ntp-01", 5, (short) 1);
    }

    @Bean
    public NewTopic topic2() {
        return new NewTopic("ntp-02", 3, (short) 1);
    }

}
```

5. KafkaSyncProducerController.java

```java
package com.lagou.kafka.demo.controller;

import org.apache.kafka.clients.producer.ProducerRecord;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.util.concurrent.ListenableFuture;
```

```java
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.ExecutionException;

@RestController
public class KafkaSyncProducerController {

    @Autowired
    private KafkaTemplate template;

    @RequestMapping("send/sync/{message}")
    public String sendSync(@PathVariable String message) {
        ListenableFuture future = template.send(
                new ProducerRecord<Integer, String>(
                        "topic-spring-02",
                        0,
                        1,
                        message));

        try {
            // 同步等待broker的响应
            Object o = future.get();
            SendResult<Integer, String> result = (SendResult<Integer, String>) o;

            System.out.println(result.getRecordMetadata().topic()
                    + result.getRecordMetadata().partition()
                    + result.getRecordMetadata().offset());

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        return "success";
    }

}
```

6. KafkaAsyncProducerController

```java
package com.lagou.kafka.demo.controller;

import org.apache.kafka.clients.producer.ProducerRecord;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
```

```java
import org.springframework.util.concurrent.ListenableFuture;
import org.springframework.util.concurrent.ListenableFutureCallback;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class KafkaAsyncProducerController {

    @Autowired
    private KafkaTemplate<Integer, String> template;

    @RequestMapping("send/async/{message}")
    public String asyncSend(@PathVariable String message) {

        ProducerRecord<Integer, String> record = new ProducerRecord<Integer, String>(
                "topic-spring-02",
                0,
                3,
                message
        );

        ListenableFuture<SendResult<Integer, String>> future = template.send(record);
        // 添加回调，异步等待响应
        future.addCallback(new ListenableFutureCallback<SendResult<Integer, String>>
() {
            @Override
            public void onFailure(Throwable throwable) {
                System.out.println("发送失败：" + throwable.getMessage());
            }

            @Override
            public void onSuccess(SendResult<Integer, String> result) {
                System.out.println("发送成功：" +
                        result.getRecordMetadata().topic() + "\t"
                        + result.getRecordMetadata().partition() + "\t"
                        + result.getRecordMetadata().offset());
            }
        });
        return "success";
    }

}
```

7. MyConsumer.java

```java
package com.lagou.kafka.demo.consumer;

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

```java
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

import java.util.Optional;

@Component
public class MyConsumer {
    @KafkaListener(topics = "topic-spring-02")
    public void onMessage(ConsumerRecord<Integer, String> record) {
        Optional<ConsumerRecord<Integer, String>> optional =
Optional.ofNullable(record);
        if (optional.isPresent()) {
            System.out.println(
                    record.topic() + "\t"
                            + record.partition() + "\t"
                            + record.offset() + "\t"
                            + record.key() + "\t"
                            + record.value());
        }
    }
}
```

1.4 服务端参数配置

$KAFKA_HOME/config/server.properties文件中的配置。

1.4.1 zookeeper.connect

该参数用于配置Kafka要连接的Zookeeper/集群的地址。

它的值是一个字符串，使用逗号分隔Zookeeper的多个地址。Zookeeper的单个地址是`host:port`形式的，可以在最后添加Kafka在Zookeeper中的根节点路径。

如：

```
zookeeper.connect=node2:2181,node3:2181,node4:2181/myKafka
```

```
116 ############################# Zookeeper #############################
117
118 # Zookeeper connection string (see zookeeper docs for details).
119 # This is a comma separated host:port pairs, each corresponding to a zk
120 # server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
121 # You can also append an optional chroot string to the urls to specify the
122 # root directory for all kafka znodes.
123 zookeeper.connect=localhost:2181
124
125 # Timeout in ms for connecting to zookeeper
126 zookeeper.connection.timeout.ms=6000
```

1.4.2 listeners

用于指定当前Broker向外发布服务的地址和端口。

与`advertised.listeners`配合，用于做内外网隔离。

```
############################# 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
```

内外网隔离配置：

listener.security.protocol.map

监听器名称和安全协议的映射配置。

比如，可以将内外网隔离，即使它们都使用SSL。

listener.security.protocol.map=INTERNAL:SSL,EXTERNAL:SSL

每个监听器的名称只能在map中出现一次。

inter.broker.listener.name

用于配置broker之间通信使用的监听器名称，该名称必须在advertised.listeners列表中。

inter.broker.listener.name=EXTERNAL

listeners

用于配置broker监听的URI以及监听器名称列表，使用逗号隔开多个URI及监听器名称。

如果监听器名称代表的不是安全协议，必须配置listener.security.protocol.map。

每个监听器必须使用不同的网络端口。

advertised.listeners

需要将该地址发布到zookeeper供客户端使用，如果客户端使用的地址与listeners配置不同。

可以在zookeeper的`get /myKafka/brokers/ids/<broker.id>`中找到。

在IaaS环境，该条目的网络接口得与broker绑定的网络接口不同。

如果不设置此条目，就使用listeners的配置。跟listeners不同，该条目不能使用0.0.0.0网络端口。

advertised.listeners的地址必须是listeners中配置的或配置的一部分。

典型配置：

```
32 listener.security.protocol.map=INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
33 listeners=INTERNAL://192.168.100.101:9092,EXTERNAL://192.168.100.130:9093
34 inter.broker.listener.name=EXTERNAL
35 # Hostname and port the broker will advertise to producers and consumers. If not set,
36 # it uses the value for "listeners" if configured.  Otherwise, it will use the value
37 # returned from java.net.InetAddress.getCanonicalHostName().
38 #advertised.listeners=PLAINTEXT://your.host.name:9092
39 advertised.listeners=EXTERNAL://192.168.100.130:9092
```

1.4.3 broker.id

该属性用于唯一标记一个Kafka的Broker，它的值是一个任意integer值。

当Kafka以分布式集群运行的时候，尤为重要。

最好该值跟该Broker所在的物理主机有关的，如主机名为`host1.lagou.com`，则`broker.id=1`，如果主机名为`192.168.100.101`，则`broker.id=101`等等。

```
18 ############################# Server Basics #############################
19
20 # The id of the broker. This must be set to a unique integer for each broker.
21 broker.id=0
22
```

1.4.4 log.dir

通过该属性的值，指定Kafka在磁盘上保存消息的日志片段的目录。

它是一组用逗号分隔的本地文件系统路径。

如果指定了多个路径，那么broker 会根据“最少使用”原则，把同一个分区的日志片段保存到同一个路径下。

broker 会往拥有最少数目分区的路径新增分区，而不是往拥有最小磁盘空间的路径新增分区。

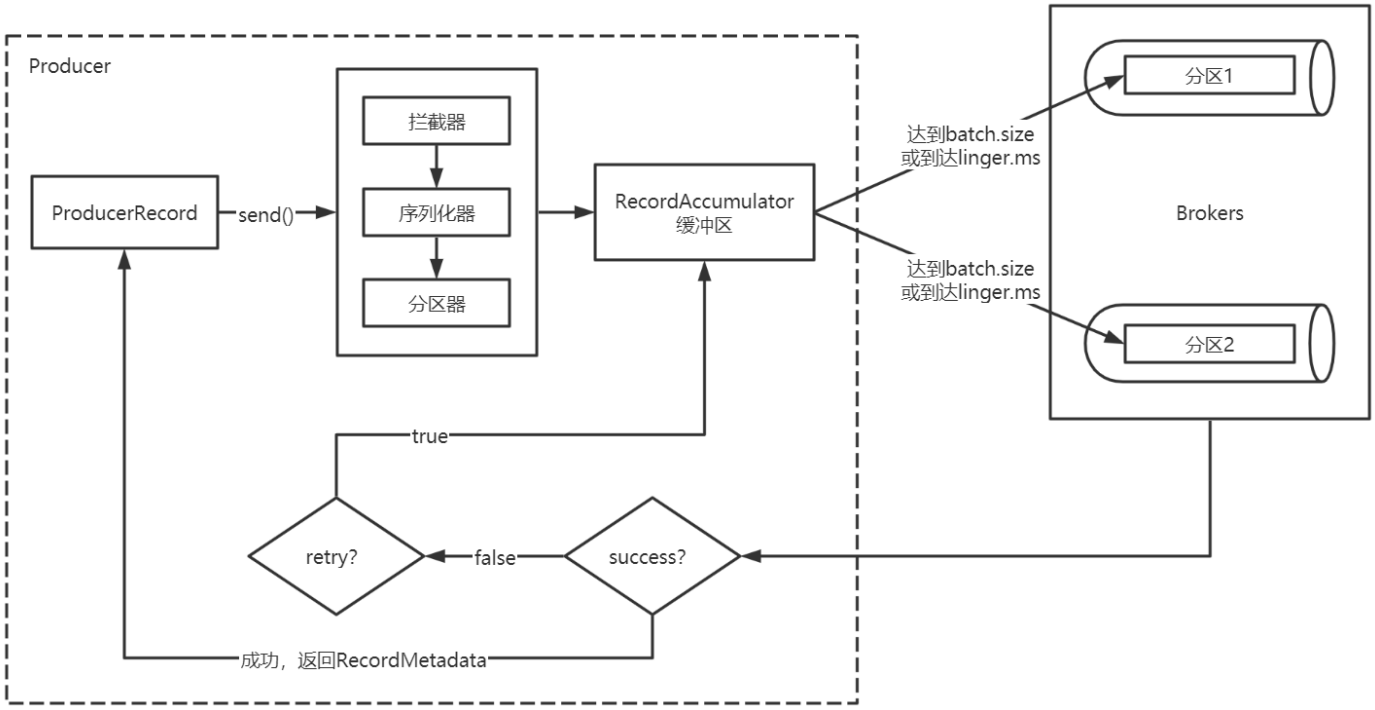
```
############################# Log Basics #############################

# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs
```

第二部分 Kafka高级特性解析

2.1 生产者

2.1.1 消息发送

2.1.1.1 数据生产流程解析



1. Producer创建时，会创建一个Sender线程并设置为守护线程。
2. 生产消息时，内部其实是异步流程；生产的消息先经过拦截器->序列化器->分区器，然后将消息缓存在缓冲区（该缓冲区也是在Producer创建时创建）。
3. 批次发送的条件为：缓冲区数据大小达到batch.size或者linger.ms达到上限，哪个先达到就算哪个。
4. 批次发送后，发往指定分区，然后落盘到broker；如果生产者配置了retrires参数大于0并且失败原因允许重试，那么客户端内部会对该消息进行重试。
5. 落盘到broker成功，返回生产元数据给生产者。
6. 元数据返回有两种方式：一种是通过阻塞直接返回，另一种是通过回调返回。

2.1.1.2 必要参数配置

2.1.1.2.1 broker配置

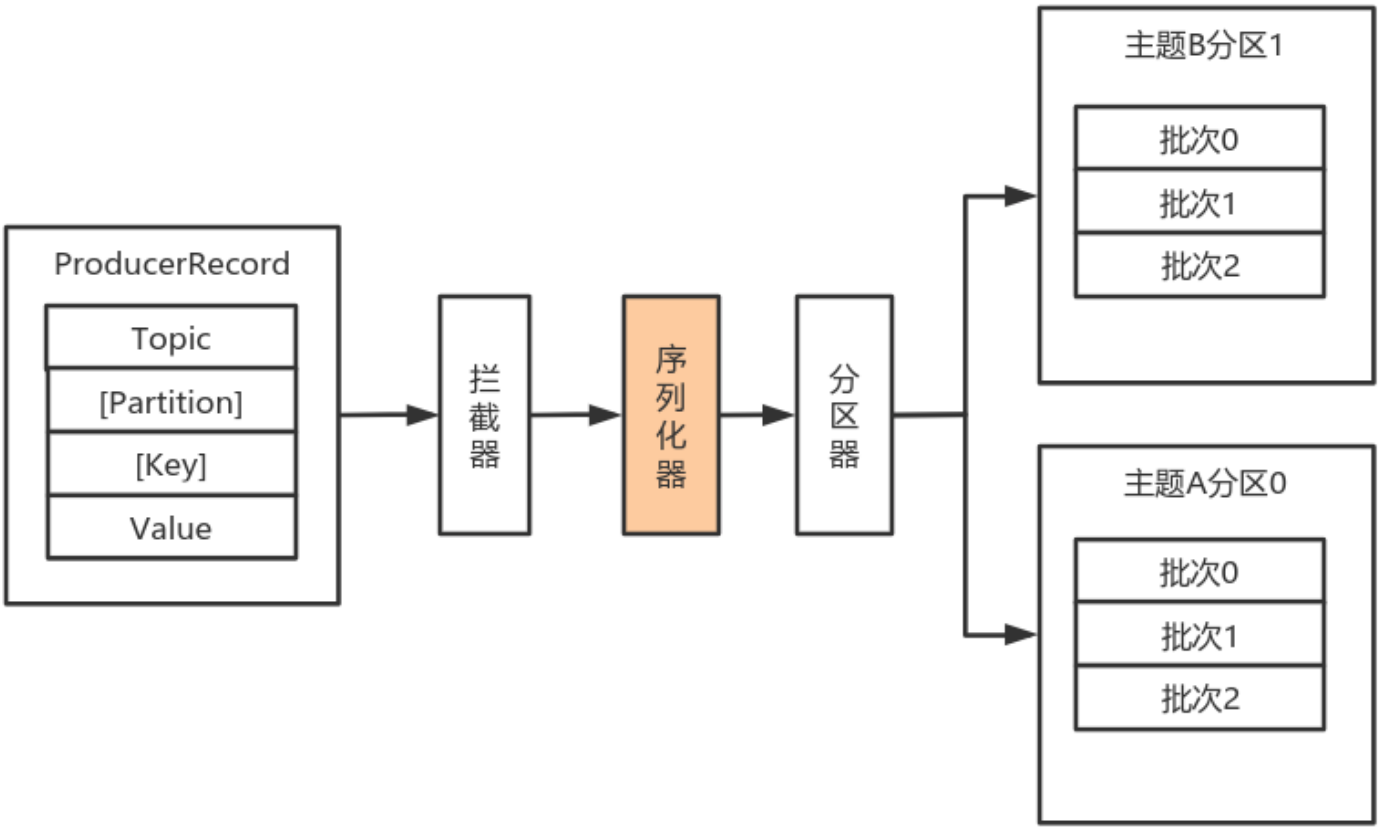
1. 配置条目的使用方式：

```
13          Map<String, Object> configs = new HashMap<>();
14
15          configs.put("bootstrap.servers", "node1:9092,node2:9092");
16  //        configs.put("key.serializer", StringSerializer.class);
17          configs.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
18  //        configs.put("value.serializer", StringSerializer.class);
19          configs.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
20          configs.put("acks", "all");
21          configs.put("compression.type", "gzip");
22          configs.put("retries", 3);
23
24          KafkaProducer<String, String> producer = new KafkaProducer<String, String>(configs);
25
```

2. 配置参数：

属性	说明	重要性
bootstrap.servers	生产者客户端与broker集群建立初始连接需要的broker地址列表，由该初始连接发现Kafka集群中其他的所有broker。该地址列表不需要写全部的Kafka集群中broker的地址，但也不要写一个，以防该节点宕机的时候不可用。形式为：`host1:port1,host2:port2,...`.	high
key.serializer	实现了接口`org.apache.kafka.common.serialization.Serializer`的key序列化类。	high
value.serializer	实现了接口`org.apache.kafka.common.serialization.Serializer`的value序列化类。	high
acks	该选项控制着已发送消息的持久性。 `acks=0`：生产者不等待broker的任何消息确认。只要将消息放到了socket的缓冲区，就认为消息已发送。不能保证服务器是否收到该消息，`retries`设置也不起作用，因为客户端不关心消息是否发送失败。客户端收到的消息偏移量永远是-1。 `acks=1`：leader将记录写到它本地日志，就响应客户端确认消息，而不等待follower副本的确认。如果leader确认了消息就宕机，则可能会丢失消息，因为follower副本可能还没来得及同步该消息。 `acks=all`：leader等待所有同步的副本确认该消息。保证了只要有一个同步副本存在，消息就不会丢失。这是最强的可用性保证。等价于`acks=-1`。默认值为1，字符串。可选值：[all, -1, 0, 1]	high
compression.type	生产者生成数据的压缩格式。默认是none（没有压缩）。允许的值：`none`，`gzip`，`snappy`和`lz4`。压缩是对整个消息批次来讲的。消息批的效率也影响压缩的比例。消息批越大，压缩效率越好。字符串类型的值。默认是none。	high
retries	设置该属性为一个大于1的值，将在消息发送失败的时候重新发送消息。该重试与客户端收到异常重新发送并无二至。允许重试但是不设置`max.in.flight.requests.per.connection`为1，存在消息乱序的可能，因为如果两个批次发送到同一个分区，第一个失败了重试，第二个成功了，则第一个消息批在第二个消息批后。int类型的值，默认：0，可选值：[0,...,2147483647]	high

2.1.1.3 序列化器



由于Kafka中的数据都是字节数组，在将消息发送到Kafka之前需要先将数据序列化为字节数组。

序列化器的作用就是用于序列化要发送的消息的。

Kafka使用`org.apache.kafka.common.serialization.Serializer`接口用于定义序列化器，将泛型指定类型的数据转换为字节数组。

```
package org.apache.kafka.common.serialization;

import java.io.Closeable;
import java.util.Map;

/**
 * 将对象转换为byte数组的接口
 *
 * 该接口的实现类需要提供无参构造器
 * @param <T> 从哪个类型转换
 */
public interface Serializer<T> extends Closeable {

    /**
     * 类的配置信息
     * @param configs key/value pairs
     * @param isKey key的序列化还是value的序列化
     */
    void configure(Map<String, ?> configs, boolean isKey);
```

```
20
21      /**
22       * 将对象转换为字节数组
23       *
24       * @param topic 主题名称
25       * @param data 需要转换的对象
26       * @return 序列化的字节数组
27       */
28      byte[] serialize(String topic, T data);
29
30      /**
31       * 关闭序列化器
32       * 该方法需要提供幂等性，因为可能调用多次。
33       */
34      @Override
35      void close();
36  }
```

系统提供了该接口的子接口以及实现类：

`org.apache.kafka.common.serialization.ByteArraySerializer`

```
21  public class ByteArraySerializer implements Serializer<byte[]> {
22
23      @Override
24      public void configure(Map<String, ?> configs, boolean isKey) {
25          // nothing to do
26      }
27
28      @Override
29      public byte[] serialize(String topic, byte[] data) {
30          return data;
31      }
32
33      @Override
34      public void close() {
35          // nothing to do
36      }
37  }
```

`org.apache.kafka.common.serialization.ByteBufferSerializer`

```java
public class ByteBufferSerializer implements Serializer<ByteBuffer> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public byte[] serialize(String topic, ByteBuffer data) {
        if (data == null)
            return null;

        data.rewind();

        if (data.hasArray()) {
            byte[] arr = data.array();
            if (data.arrayOffset() == 0 && arr.length == data.remaining()) {
                return arr;
            }
        }

        byte[] ret = new byte[data.remaining()];
        data.get(ret, offset: 0, ret.length);
        data.rewind();
        return ret;
    }

    public void close() {
        // nothing to do
    }
}
```

`org.apache.kafka.common.serialization.BytesSerializer`

```java
public class BytesSerializer implements Serializer<Bytes> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public byte[] serialize(String topic, Bytes data) {
        if (data == null)
            return null;

        return data.get();
    }

    public void close() {
        // nothing to do
    }
}
```

`org.apache.kafka.common.serialization.DoubleSerializer`

```java
public class DoubleSerializer implements Serializer<Double> {

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    @Override
    public byte[] serialize(String topic, Double data) {
        if (data == null)
            return null;

        long bits = Double.doubleToLongBits(data);
        return new byte[] {
            (byte) (bits >>> 56),
            (byte) (bits >>> 48),
            (byte) (bits >>> 40),
            (byte) (bits >>> 32),
            (byte) (bits >>> 24),
            (byte) (bits >>> 16),
            (byte) (bits >>> 8),
            (byte) bits
        };
    }

    @Override
    public void close() {
        // nothing to do
    }
}
```

`org.apache.kafka.common.serialization.FloatSerializer`

```java
public class FloatSerializer implements Serializer<Float> {

    @Override
    public void configure(final Map<String, ?> configs, final boolean isKey) {
        // nothing to do
    }

    @Override
    public byte[] serialize(final String topic, final Float data) {
        if (data == null)
            return null;

        long bits = Float.floatToRawIntBits(data);
        return new byte[] {
            (byte) (bits >>> 24),
            (byte) (bits >>> 16),
            (byte) (bits >>> 8),
            (byte) bits
        };
    }

    @Override
    public void close() {
        // nothing to do
    }
}
```

`org.apache.kafka.common.serialization.IntegerSerializer`

```java
public class IntegerSerializer implements Serializer<Integer> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public byte[] serialize(String topic, Integer data) {
        if (data == null)
            return null;

        return new byte[] {
            (byte) (data >>> 24),
            (byte) (data >>> 16),
            (byte) (data >>> 8),
            data.byteValue()
        };
    }

    public void close() {
        // nothing to do
    }
}
```

`org.apache.kafka.common.serialization.StringSerializer`

```java
    public void configure(Map<String, ?> configs, boolean isKey) {
        String propertyName = isKey ? "key.serializer.encoding" : "value.serializer.encoding";
        Object encodingValue = configs.get(propertyName);
        if (encodingValue == null)
            encodingValue = configs.get("serializer.encoding");
        if (encodingValue != null && encodingValue instanceof String)
            encoding = (String) encodingValue;
    }

    @Override
    public byte[] serialize(String topic, String data) {
        try {
            if (data == null)
                return null;
            else
                return data.getBytes(encoding);
        } catch (UnsupportedEncodingException e) {
            throw new SerializationException("Error when serializing string to byte[] due to unsupported encoding " + encoding);
        }
    }

    @Override
    public void close() {
        // nothing to do
    }
}
```

`org.apache.kafka.common.serialization.LongSerializer`

```java
public class LongSerializer implements Serializer<Long> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public byte[] serialize(String topic, Long data) {
        if (data == null)
            return null;

        return new byte[] {
            (byte) (data >>> 56),
            (byte) (data >>> 48),
            (byte) (data >>> 40),
            (byte) (data >>> 32),
            (byte) (data >>> 24),
            (byte) (data >>> 16),
            (byte) (data >>> 8),
            data.byteValue()
        };
    }

    public void close() {
        // nothing to do
    }
}
```

`org.apache.kafka.common.serialization.ShortSerializer`

```java
public class ShortSerializer implements Serializer<Short> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public byte[] serialize(String topic, Short data) {
        if (data == null)
            return null;

        return new byte[] {
            (byte) (data >>> 8),
            data.byteValue()
        };
    }

    public void close() {
        // nothing to do
    }
}
```

2.1.1.3.1 自定义序列化器

数据的序列化一般生产中使用avro。

自定义序列化器需要实现org.apache.kafka.common.serialization.Serializer<T>接口，并实现其中的`serialize`方法。

案例：

实体类：

```java
package com.lagou.kafka.demo.entity;

public class User {
    private Integer userId;
    private String username;

    public Integer getUserId() {
        return userId;
    }

    public void setUserId(Integer userId) {
        this.userId = userId;
    }

```

```
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

序列化类：

```
package com.lagou.kafka.demo.serializer;

import com.lagou.kafka.demo.entity.User;
import org.apache.kafka.common.errors.SerializationException;
import org.apache.kafka.common.serialization.Serializer;

import java.io.UnsupportedEncodingException;
import java.nio.Buffer;
import java.nio.ByteBuffer;
import java.util.Map;

public class UserSerializer implements Serializer<User> {
    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
        // do nothing
    }

    @Override
    public byte[] serialize(String topic, User data) {
        try {
            // 如果数据是null，则返回null
            if (data == null) return null;

            Integer userId = data.getUserId();
            String username = data.getUsername();

            int length = 0;
            byte[] bytes = null;
            if (null != username) {
                bytes = username.getBytes("utf-8");
                length = bytes.length;
            }

            ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + length);
            buffer.putInt(userId);
            buffer.putInt(length);
            buffer.put(bytes);

```

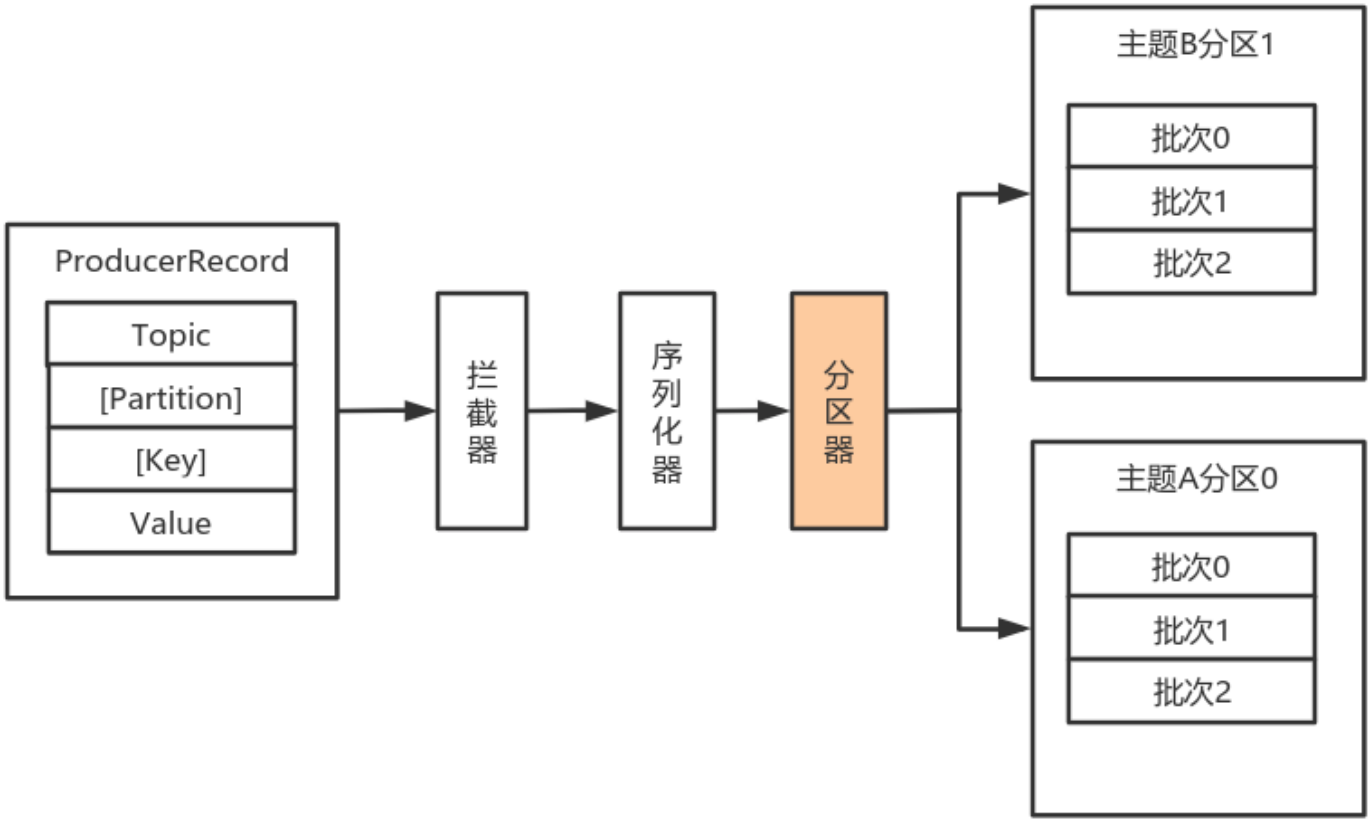
```
            return buffer.array();
        } catch (UnsupportedEncodingException e) {
            throw new SerializationException("序列化数据异常");
        }
    }

    @Override
    public void close() {
        // do nothing
    }
}
```

生产者：

```
package com.lagou.kafka.demo.producer;

import com.lagou.kafka.demo.entity.User;
import com.lagou.kafka.demo.serializer.UserSerializer;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.HashMap;
import java.util.Map;

public class MyProducer {
    public static void main(String[] args) {

        Map<String, Object> configs = new HashMap<>();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        // 设置自定义的序列化类
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
UserSerializer.class);

        KafkaProducer<String, User> producer = new KafkaProducer<String, User>
(configs);

        User user = new User();
        user.setUserId(1001);
        user.setUsername("张三");

        ProducerRecord<String, User> record = new ProducerRecord<>(
                "tp_user_01",
                0,
                user.getUsername(),
                user
        );

```

```
33          producer.send(record, (metadata, exception) -> {
34              if (exception == null) {
35                  System.out.println("消息发送成功: "
36                          + metadata.topic() + "\t"
37                          + metadata.partition() + "\t"
38                          + metadata.offset());
39              } else {
40                  System.out.println("消息发送异常");
41              }
42          });
43
44          // 关闭生产者
45          producer.close();
46      }
47  }
```

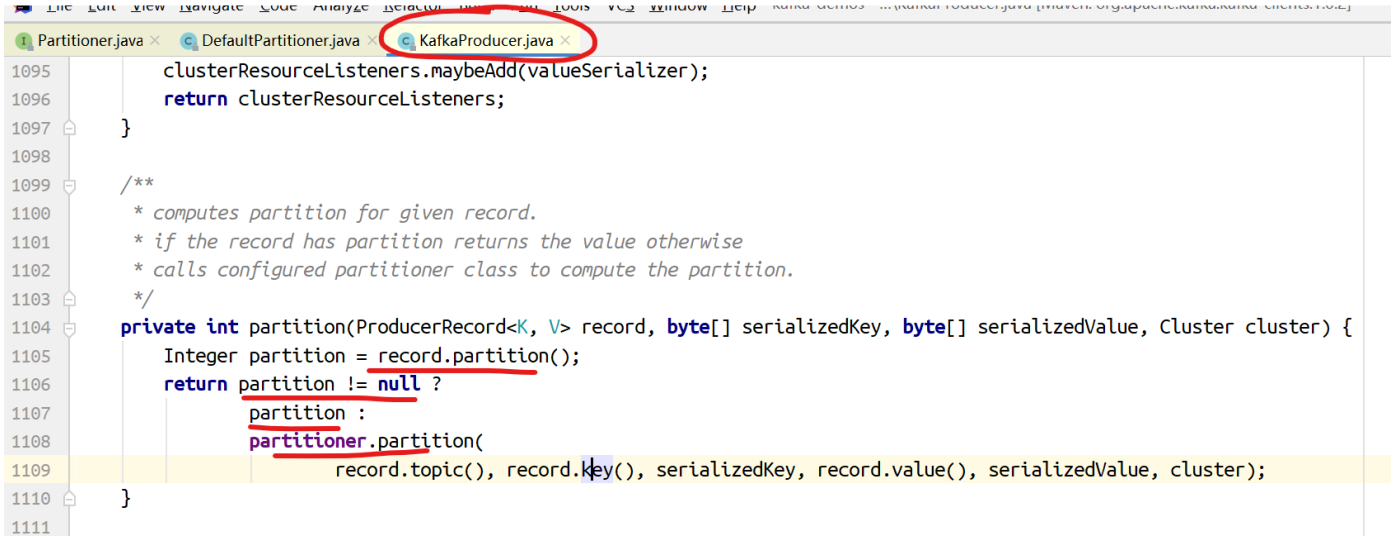
2.1.1.4 分区器

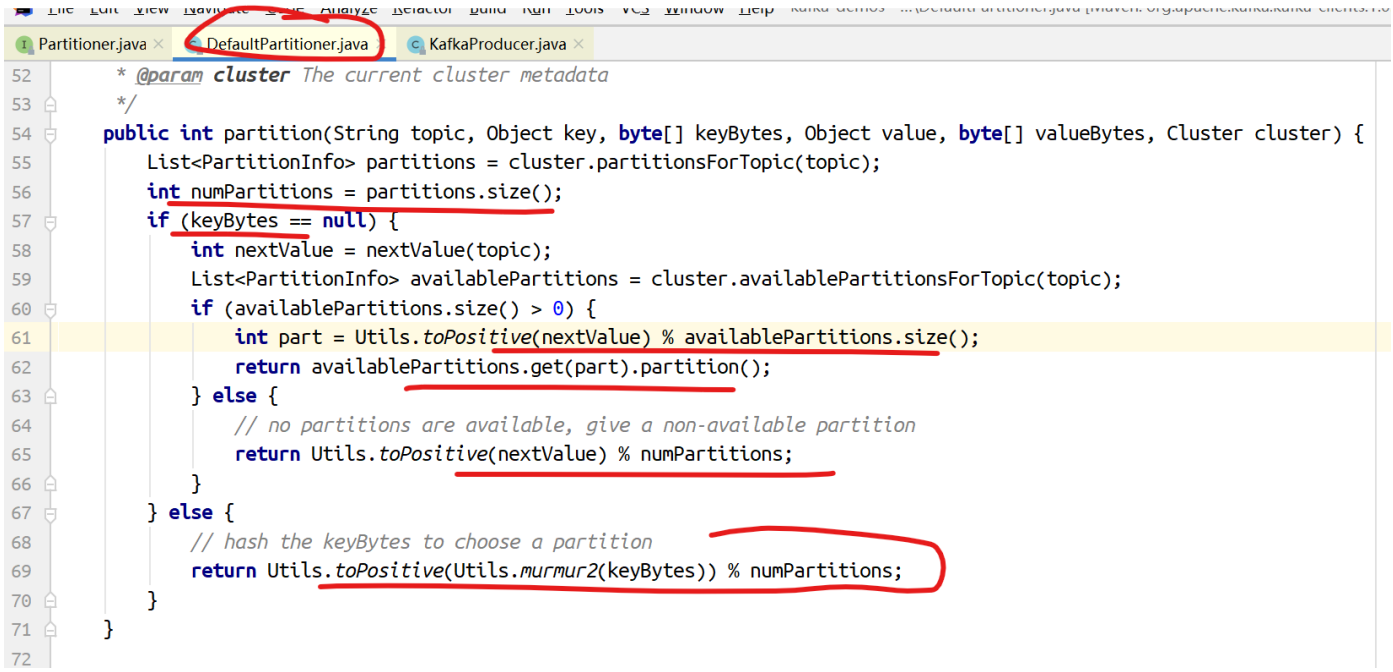


默认（DefaultPartitioner）分区计算：

1. 如果record提供了分区号，则使用record提供的分区号
2. 如果record没有提供分区号，则使用key的序列化后的值的hash值对分区数量取模
3. 如果record没有提供分区号，也没有提供key，则使用轮询的方式分配分区号。

1. 会首先在可用的分区中分配分区号
2. 如果没有可用的分区，则在该主题所有分区中分配分区号。





如果要自定义分区器，则需要

1. 首先开发Partitioner接口的实现类
2. 在KafkaProducer中进行设置：configs.put("partitioner.class", "xxx.xx.Xxx.class")

位于`org.apache.kafka.clients.producer`中的分区器接口：

```
package org.apache.kafka.clients.producer;

import org.apache.kafka.common.Configurable;
import org.apache.kafka.common.Cluster;

```

```java
import java.io.Closeable;

/**
 * 分区器接口
 */

public interface Partitioner extends Configurable, Closeable {

    /**
     * 为指定的消息记录计算分区值
     *
     * @param topic 主题名称
     * @param key 根据该key的值进行分区计算，如果没有则为null。
     * @param keyBytes key的序列化字节数组，根据该数组进行分区计算。如果没有key，则为null
     * @param value 根据value值进行分区计算，如果没有，则为null
     * @param valueBytes value的序列化字节数组，根据此值进行分区计算。如果没有，则为null
     * @param cluster 当前集群的元数据
     */
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster);

    /**
     * 关闭分区器的时候调用该方法
     */
    public void close();

}
```

包`org.apache.kafka.clients.producer.internals`中分区器的默认实现：

```java
package org.apache.kafka.clients.producer.internals;

import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.atomic.AtomicInteger;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.utils.Utils;

/**
 * 默认的分区策略：
 *
 *  如果在记录中指定了分区，则使用指定的分区
 *  如果没有指定分区，但是有key的值，则使用key值的散列值计算分区
```

```
 *  如果没有指定分区也没有key的值，则使用轮询的方式选择一个分区
 */
public class DefaultPartitioner implements Partitioner {

    private final ConcurrentMap<String, AtomicInteger> topicCounterMap = new
ConcurrentHashMap<>();

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

    /**
     * 为指定的消息记录计算分区值
     *
     * @param topic 主题名称
     * @param key 根据该key的值进行分区计算，如果没有则为null。
     * @param keyBytes key的序列化字节数组，根据该数组进行分区计算。如果没有key，则为null
     * @param value 根据value值进行分区计算，如果没有，则为null
     * @param valueBytes value的序列化字节数组，根据此值进行分区计算。如果没有，则为null
     * @param cluster 当前集群的元数据
     */
    public int partition(String topic, Object key, byte[] keyBytes, Object value,
byte[] valueBytes, Cluster cluster) {
        // 获取指定主题的所有分区信息
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        // 分区的数量
        int numPartitions = partitions.size();
        // 如果没有提供key
        if (keyBytes == null) {
            int nextValue = nextValue(topic);
            List<PartitionInfo> availablePartitions =
cluster.availablePartitionsForTopic(topic);
            if (availablePartitions.size() > 0) {
                int part = Utils.toPositive(nextValue) % availablePartitions.size();
                return availablePartitions.get(part).partition();
            } else {
                // no partitions are available, give a non-available partition
                return Utils.toPositive(nextValue) % numPartitions;
            }
        } else {
            // hash the keyBytes to choose a partition
            // 如果有，就计算keyBytes的哈希值，然后对当前主题的个数取模
            return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
        }
    }

    private int nextValue(String topic) {
        AtomicInteger counter = topicCounterMap.get(topic);
        if (null == counter) {
            counter = new AtomicInteger(ThreadLocalRandom.current().nextInt());
            AtomicInteger currentCounter = topicCounterMap.putIfAbsent(topic,
counter);
            if (currentCounter != null) {
```

```
                    counter = currentCounter;
                }
            }
            return counter.getAndIncrement();
        }

        public void close() {}

}
```

```
    /**
     * 计算给定消息的分区号
     * computes partition for given record.
     * if the record has partition returns the value otherwise
     * calls configured partitioner class to compute the partition.
     */
    private int partition(ProducerRecord<K, V> record, byte[] serializedKey, byte[] serializedValue, Cluster cluster) {
        // 获取消息的分区号
        Integer partition = record.partition();
        // 如果指定了，就使用该指定的值，如果没有指定，则使用分区器进行计算
        return partition != null ?
                partition :
                partitioner.partition(
                        record.topic(), record.key(), serializedKey, record.value(), serializedValue, cluster);
    }
```

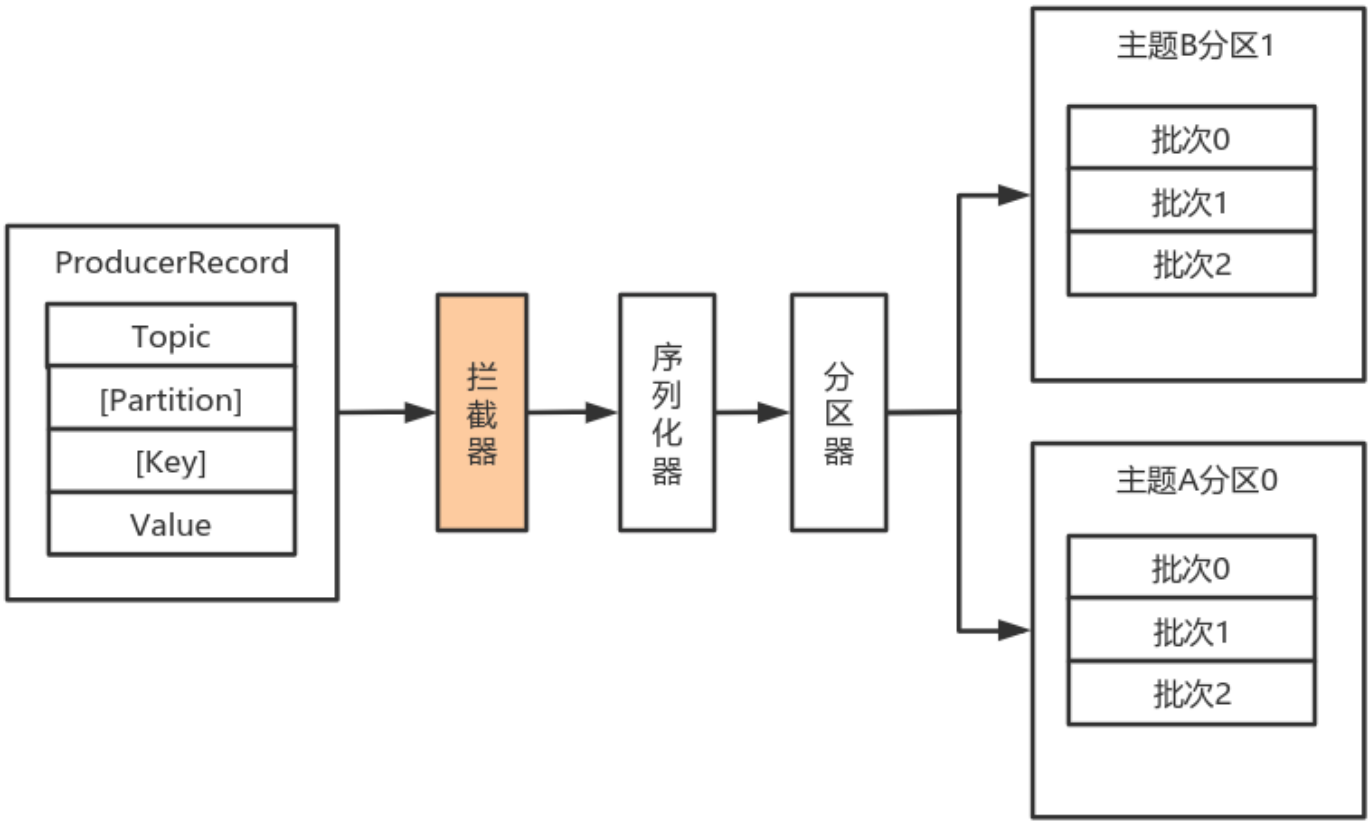
可以实现Partitioner接口自定义分区器：

```
public class MyPartitioner implements Partitioner {
    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
        return 0;
    }

    @Override
    public void close() {

    }

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

    }
}
```

然后在生产者中配置：

```
        // 设置自定义分区器
//        configs.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, MyPartitioner.class);
        configs.put("partitioner.class", "com.lagou.kafka.demo.partitioner.MyPartitioner");
```

2.1.1.5 拦截器



Producer拦截器（interceptor）和Consumer端Interceptor是在Kafka 0.10版本被引入的，主要用于实现Client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：

- onSend(ProducerRecord)：该方法封装进KafkaProducer.send方法中，即运行在用户主线程中。Producer确保在消息被序列化以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。
- onAcknowledgement(RecordMetadata, Exception)：该方法会在消息被应答之前或消息发送失败时调用，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。
- 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。

自定义拦截器：

1. 实现ProducerInterceptor接口
2. 在KafkaProducer的设置中设置自定义的拦截器

```
//      interceptor.classes
      // 如果有多个拦截器，则设置为多个拦截器类的全限定类名，中间用逗号隔开
      configs.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, "com.lagou.kafka.demo.interceptor.InterceptorOne," +
              "com.lagou.kafka.demo.interceptor.InterceptorTwo," +
              "com.lagou.kafka.demo.interceptor.InterceptorThree");
```

案例：

1. 消息实体类：

```
package com.lagou.kafka.demo.entity;

public class User {
    private Integer userId;
    private String username;

    public Integer getUserId() {
        return userId;
    }

    public void setUserId(Integer userId) {
        this.userId = userId;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

2. 自定义序列化器

```
package com.lagou.kafka.demo.serializer;

import com.lagou.kafka.demo.entity.User;
import org.apache.kafka.common.errors.SerializationException;
import org.apache.kafka.common.serialization.Serializer;

import java.io.UnsupportedEncodingException;
import java.nio.Buffer;
import java.nio.ByteBuffer;
import java.util.Map;

public class UserSerializer implements Serializer<User> {
    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
        // do nothing
```

```
    }

    @Override
    public byte[] serialize(String topic, User data) {
        try {
            // 如果数据是null，则返回null
            if (data == null) return null;

            Integer userId = data.getUserId();
            String username = data.getUsername();

            int length = 0;
            byte[] bytes = null;
            if (null != username) {
                bytes = username.getBytes("utf-8");
                length = bytes.length;
            }

            ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + length);
            buffer.putInt(userId);
            buffer.putInt(length);
            buffer.put(bytes);

            return buffer.array();
        } catch (UnsupportedEncodingException e) {
            throw new SerializationException("序列化数据异常");
        }
    }

    @Override
    public void close() {
        // do nothing
    }
}
```

3. 自定义分区器

```
package com.lagou.kafka.demo.partitioner;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;

import java.util.Map;

public class MyPartitioner implements Partitioner {
    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value,
byte[] valueBytes, Cluster cluster) {
        return 2;
```

```
    }

    @Override
    public void close() {

    }

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

    }
}
```

4. 自定义拦截器1

```
package com.lagou.kafka.demo.interceptor;

import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.header.Headers;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Map;

public class InterceptorOne<KEY, VALUE> implements ProducerInterceptor<KEY, VALUE> {

    private static final Logger LOGGER =
LoggerFactory.getLogger(InterceptorOne.class);

    @Override
    public ProducerRecord<KEY, VALUE> onSend(ProducerRecord<KEY, VALUE> record) {
        System.out.println("拦截器1---go");
        // 此处根据业务需要对相关的数据作修改
        String topic = record.topic();
        Integer partition = record.partition();
        Long timestamp = record.timestamp();
        KEY key = record.key();
        VALUE value = record.value();
        Headers headers = record.headers();
        // 添加消息头
        headers.add("interceptor", "interceptorOne".getBytes());

        ProducerRecord<KEY, VALUE> newRecord = new ProducerRecord<KEY, VALUE>(
                topic,
                partition,
                timestamp,
                key,
```

```
                value,
                headers
        );

        return newRecord;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        System.out.println("拦截器1---back");
        if (exception != null) {
            // 如果发生异常，记录日志中
            LOGGER.error(exception.getMessage());
        }
    }

    @Override
    public void close() {

    }

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

    }
}
```

5. 自定义拦截器2

```
package com.lagou.kafka.demo.interceptor;

import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.header.Headers;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Map;

public class InterceptorTwo<KEY, VALUE> implements ProducerInterceptor<KEY, VALUE> {

    private static final Logger LOGGER =
LoggerFactory.getLogger(InterceptorTwo.class);

    @Override
    public ProducerRecord<KEY, VALUE> onSend(ProducerRecord<KEY, VALUE> record) {
        System.out.println("拦截器2---go");
        // 此处根据业务需要对相关的数据作修改
```

```
        String topic = record.topic();
        Integer partition = record.partition();
        Long timestamp = record.timestamp();
        KEY key = record.key();
        VALUE value = record.value();
        Headers headers = record.headers();
        // 添加消息头
        headers.add("interceptor", "interceptorTwo".getBytes());

        ProducerRecord<KEY, VALUE> newRecord = new ProducerRecord<KEY, VALUE>(
                topic,
                partition,
                timestamp,
                key,
                value,
                headers
        );

        return newRecord;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        System.out.println("拦截器2---back");
        if (exception != null) {
            // 如果发生异常，记录日志中
            LOGGER.error(exception.getMessage());
        }
    }

    @Override
    public void close() {

    }

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

    }
}
```

6. 自定义拦截器3

```
package com.lagou.kafka.demo.interceptor;

import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.header.Headers;
```

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Map;

public class InterceptorThree<KEY, VALUE> implements ProducerInterceptor<KEY, VALUE>
{

    private static final Logger LOGGER =
LoggerFactory.getLogger(InterceptorThree.class);

    @Override
    public ProducerRecord<KEY, VALUE> onSend(ProducerRecord<KEY, VALUE> record) {
        System.out.println("拦截器3---go");
        // 此处根据业务需要对相关的数据作修改
        String topic = record.topic();
        Integer partition = record.partition();
        Long timestamp = record.timestamp();
        KEY key = record.key();
        VALUE value = record.value();
        Headers headers = record.headers();
        // 添加消息头
        headers.add("interceptor", "interceptorThree".getBytes());

        ProducerRecord<KEY, VALUE> newRecord = new ProducerRecord<KEY, VALUE>(
                topic,
                partition,
                timestamp,
                key,
                value,
                headers
        );

        return newRecord;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        System.out.println("拦截器3---back");
        if (exception != null) {
            // 如果发生异常，记录日志中
            LOGGER.error(exception.getMessage());
        }
    }

    @Override
    public void close() {

    }

    @Override
```

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

    }
}
```

7. 生产者

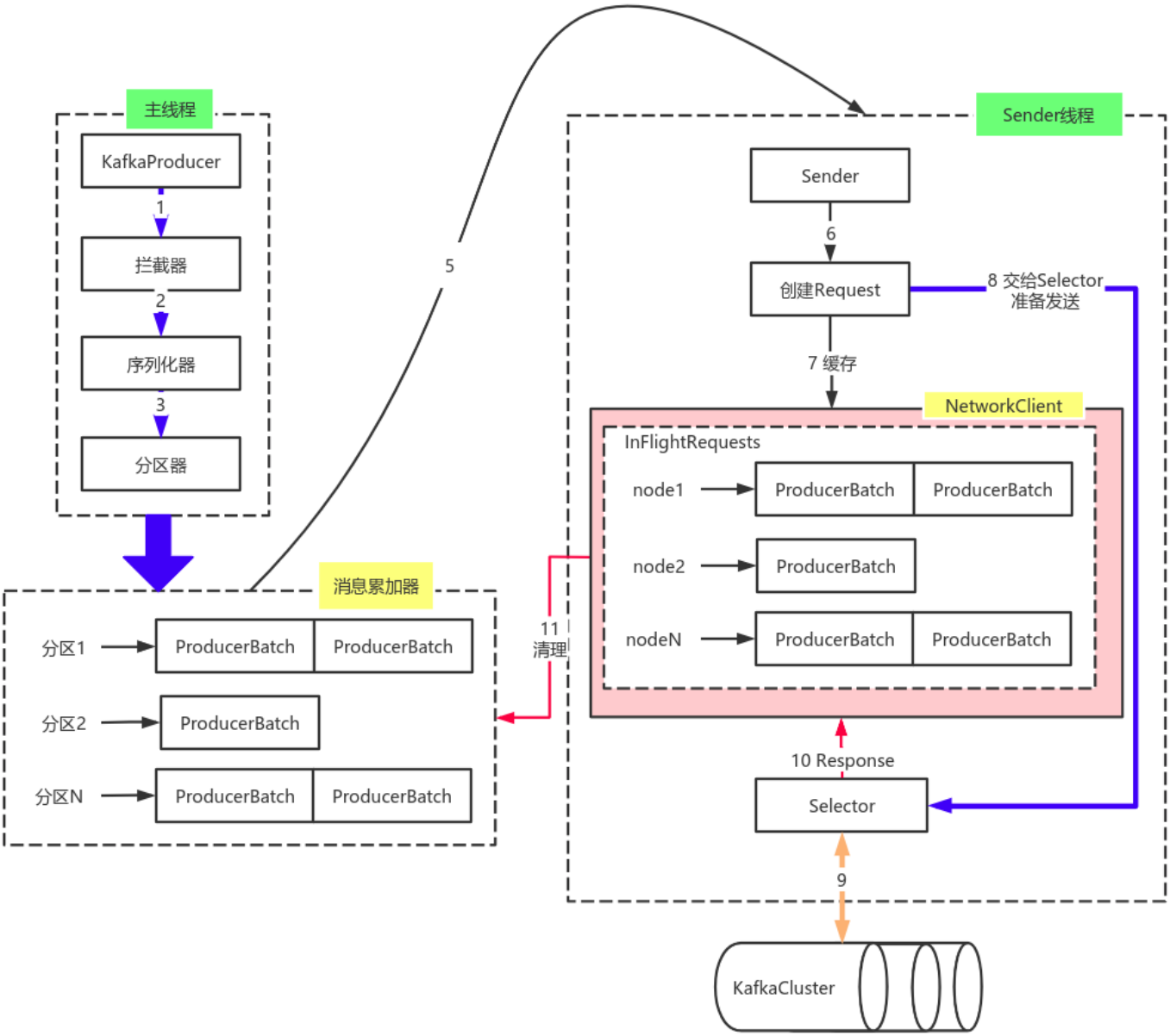
```
package com.lagou.kafka.demo.producer;

import com.lagou.kafka.demo.entity.User;
import com.lagou.kafka.demo.serializer.UserSerializer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.HashMap;
import java.util.Map;

public class MyProducer {
    public static void main(String[] args) {

        Map<String, Object> configs = new HashMap<>();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");

        // 设置自定义分区器
//        configs.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, MyPartitioner.class);
        configs.put("partitioner.class",
"com.lagou.kafka.demo.partitioner.MyPartitioner");

        // 设置拦截器
        configs.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
                "com.lagou.kafka.demo.interceptor.InterceptorOne," +
                        "com.lagou.kafka.demo.interceptor.InterceptorTwo," +
                        "com.lagou.kafka.demo.interceptor.InterceptorThree"
                );


        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        // 设置自定义的序列化类
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
UserSerializer.class);

        KafkaProducer<String, User> producer = new KafkaProducer<String, User>
(configs);

        User user = new User();
        user.setUserId(1001);
```

```
        user.setUsername("张三");

        ProducerRecord<String, User> record = new ProducerRecord<>(
                "tp_user_01",
                0,
                user.getUsername(),
                user
        );

        producer.send(record, (metadata, exception) -> {
            if (exception == null) {
                System.out.println("消息发送成功: "
                        + metadata.topic() + "\t"
                        + metadata.partition() + "\t"
                        + metadata.offset());
            } else {
                System.out.println("消息发送异常");
            }
        });

        // 关闭生产者
        producer.close();
    }
}
```

8. 运行结果：

```
拦截器1---go
拦截器2---go
拦截器3---go
拦截器1---back
拦截器2---back
拦截器3---back
消息发送成功: tp_user_01   0   2
```

2.1.2 原理剖析



由上图可以看出：KafkaProducer有两个基本线程：

- 主线程：负责消息创建，拦截器，序列化器，分区器等操作，并将消息追加到消息收集器RecoderAccumulator中；
    - 消息收集器RecoderAccumulator为**每个分区**都维护了一个 Deque<ProducerBatch> 类型的双端队列。
    - ProducerBatch 可以理解为是 ProducerRecord 的集合，批量发送有利于提升吞吐量，降低网络影响；
    - 由于生产者客户端使用 java.io.ByteBuffer 在发送消息之前进行消息保存，并维护了一个 BufferPool 实现 ByteBuffer 的复用；该缓存池只针对特定大小（ batch.size 指定）的 ByteBuffer 进行管理，对于消息过大的缓存，不能做到重复利用。
    - 每次追加一条ProducerRecord消息，会寻找/新建对应的双端队列，从其尾部获取一个ProducerBatch，判断当前消息的大小是否可以写入该批次中。若可以写入则写入；若不可以写入，则新建一个ProducerBatch，判断该消息大小是否超过客户端参数配置 batch.size 的值，不超过，则以 batch.size建立新的ProducerBatch，这样方便进行缓存重复利用；若超过，则以计算的消息大小建立对应的 ProducerBatch ，缺点就是该内存不能被复用了。
- Sender线程：
    - 该线程从消息收集器获取缓存的消息，将其处理为 <Node, List<ProducerBatch> 的形式， Node

表示集群的broker节点。

- 进一步将<Node, List<ProducerBatch>转化为<Node, Request>形式，此时才可以向服务端发送数据。
- 在发送之前，Sender线程将消息以 Map<NodeId, Deque<Request>> 的形式保存到 InFlightRequests 中进行缓存，可以通过其获取 leastLoadedNode ,即当前Node中负载压力最小的一个，以实现消息的尽快发出。

2.1.3 生产者参数配置补充

1. 参数设置方式：

```
13          Map<String, Object> configs = new HashMap<>();
14
15          configs.put("bootstrap.servers", "node1:9092,node2:9092");
16  //        configs.put("key.serializer", StringSerializer.class);
17          configs.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
18  //        configs.put("value.serializer", StringSerializer.class);
19          configs.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
20          configs.put("acks", "all");
21          configs.put("compression.type", "gzip");
22          configs.put("retries", 3);
23
24          KafkaProducer<String, String> producer = new KafkaProducer<String, String>(configs);
25
```

```
18          // 保证等待确认的消息只有设置的这几个。如果设置为1，则只有一个请求在等待响应
19          // 此时可以保证发送消息即使在重试的情况下也是有序的。
20          configs.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);
21  //        configs.put("max.in.flight.requests.per.connection", 1);
22
```

2. 补充参数：

参数名称	描述
retry.backoff.ms	在向一个指定的主题分区重发消息的时候，重试之间的等待时间。 比如3次重试，每次重试之后等待该时间长度，再接着重试。在一些失败的场景，避免了密集循环的重新发送请求。 long型值，默认100。可选值：[0,...]
retries	retries重试次数 当消息发送出现错误的时候，系统会重发消息。 跟客户端收到错误时重发一样。 如果设置了重试，还想保证消息的有序性，需要设置MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION=1 否则在重试此失败消息的时候，其他的消息可能发送成功了
request.timeout.ms	客户端等待请求响应的最大时长。如果服务端响应超时，则会重发请求，除非达到重试次数。该设置应该比 `replica.lag.time.max.ms` (a broker configuration)要大，以免在服务器延迟时间内重发消息。int类型值，默认：30000，可选值：[0,...]
interceptor.classes	在生产者接收到该消息，向Kafka集群传输之前，由序列化器处理之前，可以通过拦截器对消息进行处理。 要求拦截器类必须实现 `org.apache.kafka.clients.producer.ProducerInterceptor` 接口。 默认没有拦截器。 Map<String, Object> configs中通过List集合配置多个拦截器类名。
	当生产者发送消息之后，如何确认消息已经发送成功了。

acks	支持的值： acks=0： 如果设置为0，表示生产者不会等待broker对消息的确认，只要将消息放到缓冲区，就认为消息已经发送完成。 该情形不能保证broker是否真的收到了消息，retries配置也不会生效，因为客户端不需要知道消息是否发送成功。 发送的消息的返回的消息偏移量永远是-1。 acks=1 表示消息只需要写到主分区即可，然后就响应客户端，而不等待副本分区的确认。 在该情形下，如果主分区收到消息确认之后就宕机了，而副本分区还没来得及同步该消息，则该消息丢失。 acks=all 首领分区会等待所有的ISR副本分区确认记录。 该处理保证了只要有一个ISR副本分区存货，消息就不会丢失。 这是Kafka最强的可靠性保证，等效于`acks=-1`。
batch.size	当多个消息发送到同一个分区的时候，生产者尝试将多个记录作为一个批来处理。批处理提高了客户端和服务器的处理效率。 该配置项以字节为单位控制默认批的大小。 所有的批小于等于该值。 发送给broker的请求将包含多个批次，每个分区一个，并包含可发送的数据。 如果该值设置的比较小，会限制吞吐量（设置为0会完全禁用批处理）。如果设置的很大，又有一点浪费内存，因为Kafka会永远分配这么大的内存来参与到消息的批整合中。
client.id	生产者发送请求的时候传递给broker的id字符串。 用于在broker的请求日志中追踪什么应用发送了什么消息。 一般该id是跟业务有关的字符串。
compression.type	生产者发送的所有数据的压缩方式。默认是none，也就是不压缩。 支持的值：none、gzip、snappy和lz4。 压缩是对于整个批来讲的，所以批处理的效率也会影响到压缩的比例。
send.buffer.bytes	TCP发送数据的时候使用的缓冲区（SO_SNDBUF）大小。如果设置为0，则使用操作系统默认的。
buffer.memory	生产者可以用来缓存等待发送到服务器的记录的总内存字节。如果记录的发送速度超过了将记录发送到服务器的速度，则生产者将阻塞`max.block.ms`的时间，此后它将引发异常。此设置应大致对应于生产者将使用的总内存，但并非生产者使用的所有内存都用于缓冲。一些额外的内存将用于压缩（如果启用了压缩）以及维护运行中的请求。long型数据。默认值：33554432，可选值：[0,...]
connections.max.idle.ms	当连接空闲时间达到这个值，就关闭连接。long型数据，默认：540000
linger.ms	生产者在发送请求传输间隔会对需要发送的消息进行累积，然后作为一个批次发送。一般情况是消息的发送的速度比消息累积的速度慢。有时客户端需要减少请求的次数，即使是在发送负载不大的情况下。该配置设置了一个延迟，生产者不会立即将消息发送到broker，而是等待这么一段时间以累积消息，然后将这段时间之内的消息作为一个批次发送。该设置是批处理的另一个上限：一旦批消息达到了`batch.size`指定的值，消息批会立即发送，如果积累的消息字节数达不到`batch.size`的值，可以设置该毫秒值，等待这么长时间之后，也会发送消息批。该属性默认值是0（没有延迟）。如果设置`linger.ms=5`，则在一个请求发送之前先等待5ms。long型值，默认：0，可选值：[0,...]
max.block.ms	控制`KafkaProducer.send()`和`KafkaProducer.partitionsFor()`阻塞的时长。当缓存满了或元数据不可用的时候，这些方法阻塞。在用户提供的序列化器和分区器的阻塞时间不计入。long型值，默认：60000，可选值：[0,...]
max.request.size	单个请求的最大字节数。该设置会限制单个请求中消息批的消息个数，以免单个请求发送太多的数据。服务器有自己的限制批大小的设置，与该配置可能不一样。int类型值，默认1048576，可选值：[0,...]
partitioner.class	实现了接口`org.apache.kafka.clients.producer.Partitioner`的分区器实现类。默认值为：org.apache.kafka.clients.producer.internals.DefaultPartitioner
receive.buffer.bytes	TCP接收缓存（SO_RCVBUF），如果设置为-1，则使用操作系统默认的值。int类型值，默认32768，可选值：[-1,...]
security.protocol	跟broker通信的协议：PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL. string类型值，默认：PLAINTEXT
max.in.flight.requests.per.connection	单个连接上未确认请求的最大数量。达到这个数量，客户端阻塞。如果该值大于1，且存在失败的请求，在重试的时候消息顺序不能保证。 int类型值，默认5。可选值：[1,...]
reconnect.backoff.max.ms	对于每个连续的连接失败，每台主机的退避将成倍增加，直至达到此最大值。在计算退避增量之后，添加20%的随机抖动以避免连接风暴。 long型值，默认1000，可选值：[0,...]
reconnect.backoff.ms	尝试重连指定主机的基础等待时间。避免了到该主机的密集重连。该退避时间应用于该客户端到broker的所有连接。 long型值，默认50。可选值：[0,...]

2.2 消费者

2.2.1 概念入门

2.2.1.1 消费者、消费组

消费者从订阅的主题消费消息，消费消息的偏移量保存在Kafka的名字是`__consumer_offsets`的主题中。

消费者还可以将自己的偏移量存储到Zookeeper，需要设置offset.storage=zookeeper。

推荐使用Kafka存储消费者的偏移量。因为Zookeeper不适合高并发。

多个从同一个主题消费的消费者可以加入到一个消费组中。

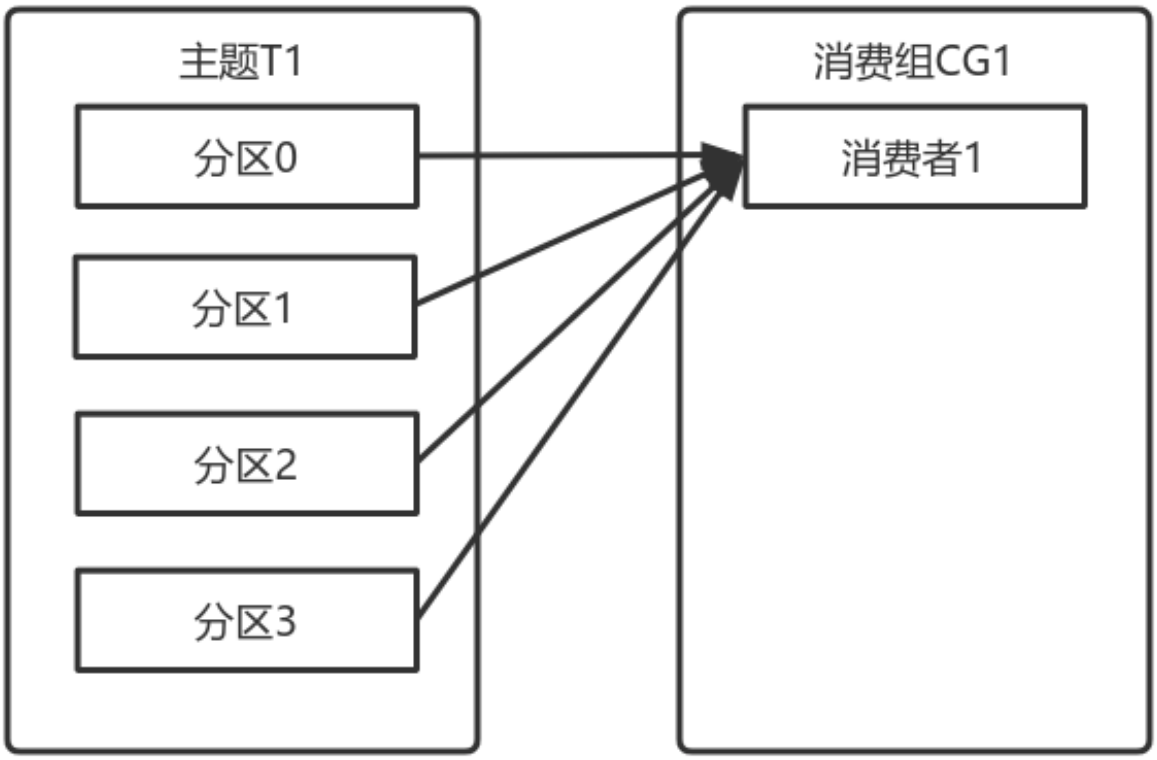
消费组中的消费者共享group_id。

configs.put("group.id", "xxx");

group_id一般设置为应用的逻辑名称。比如多个订单处理程序组成一个消费组，可以设置group_id为"order_process"。

group_id通过消费者的配置指定：`group.id=xxxxx`

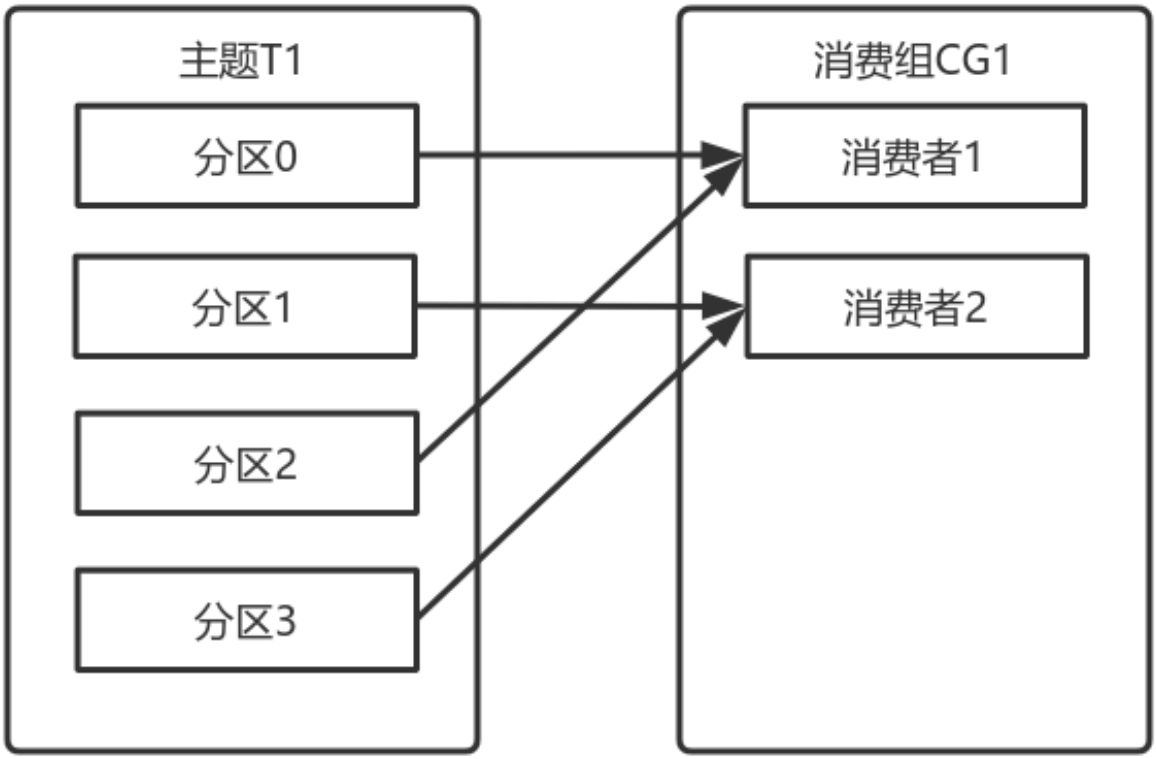
消费组均衡地给消费者分配分区，每个分区只由消费组中一个消费者消费。



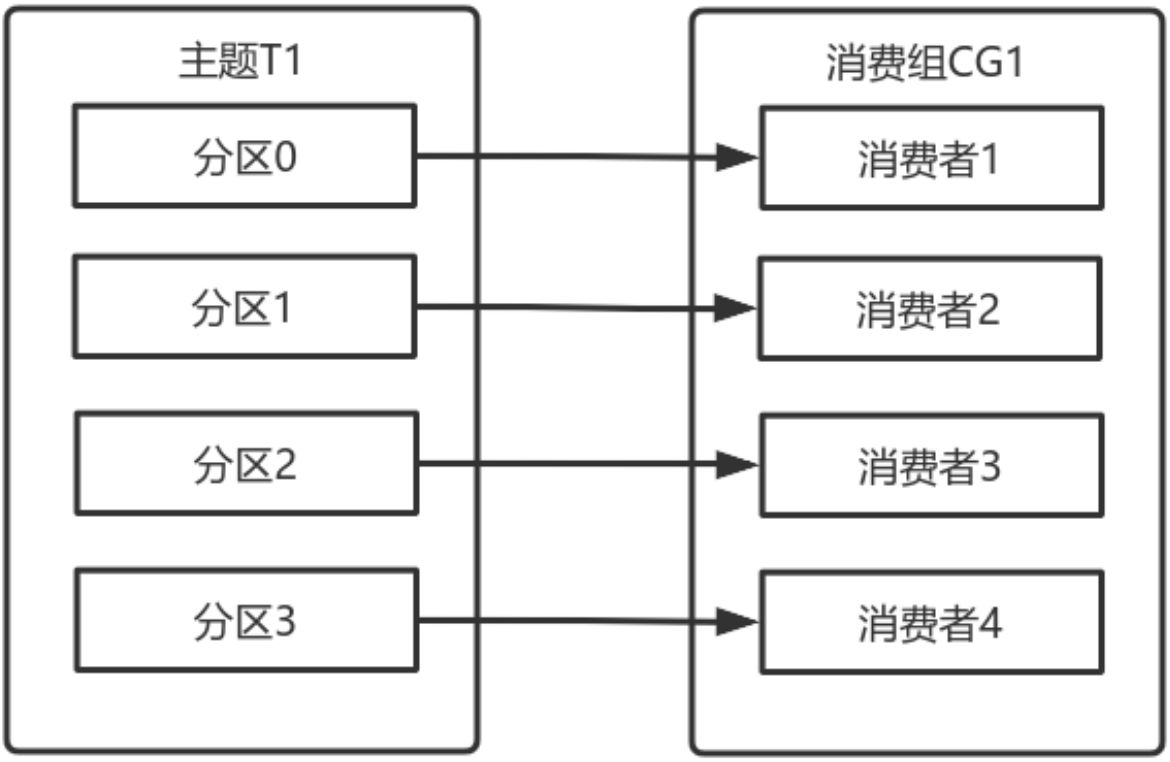
一个拥有四个分区的主题，包含一个消费者的消费组。

此时，消费组中的消费者消费主题中的所有分区。并且没有重复的可能。

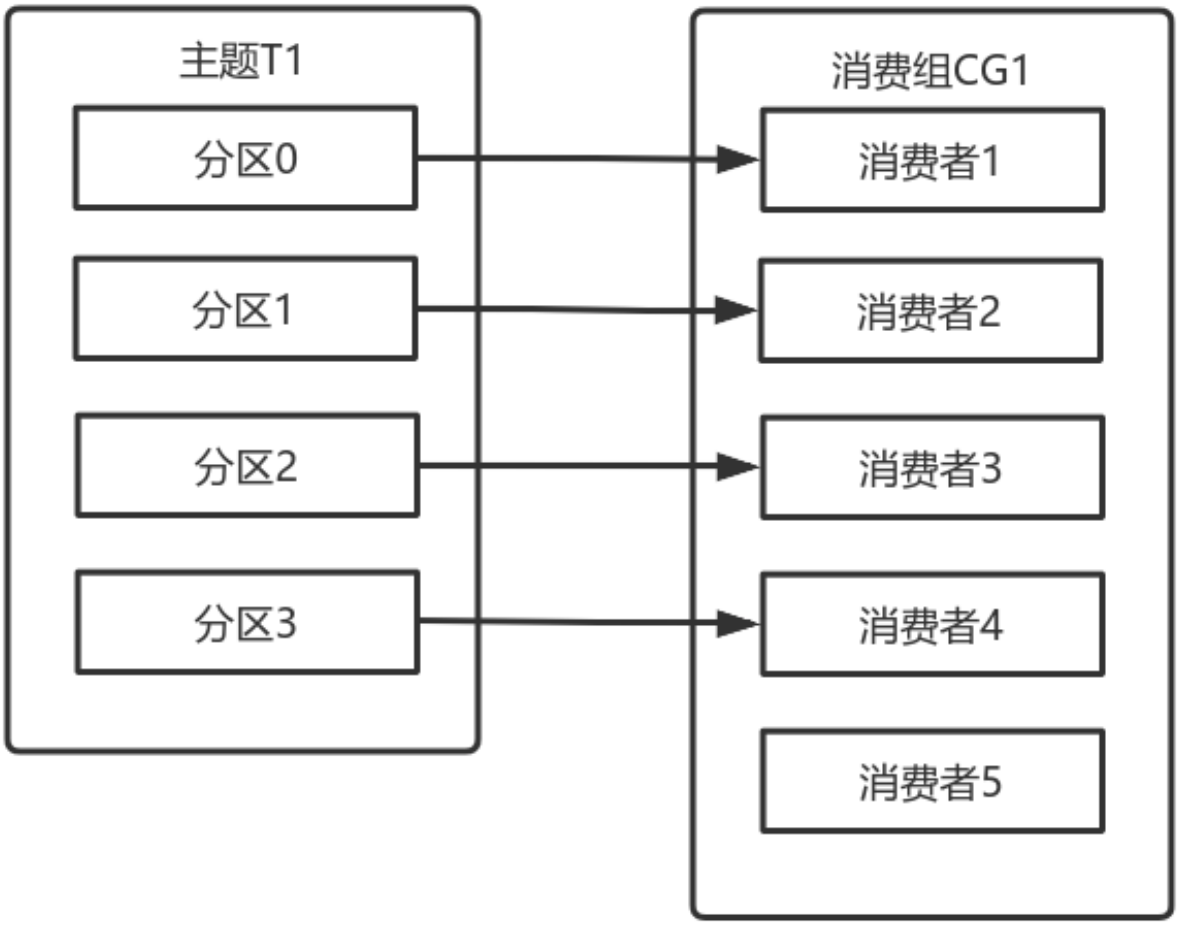
如果在消费组中添加一个消费者2，则每个消费者分别从两个分区接收消息。



如果消费组有四个消费者，则每个消费者可以分配到一个分区。

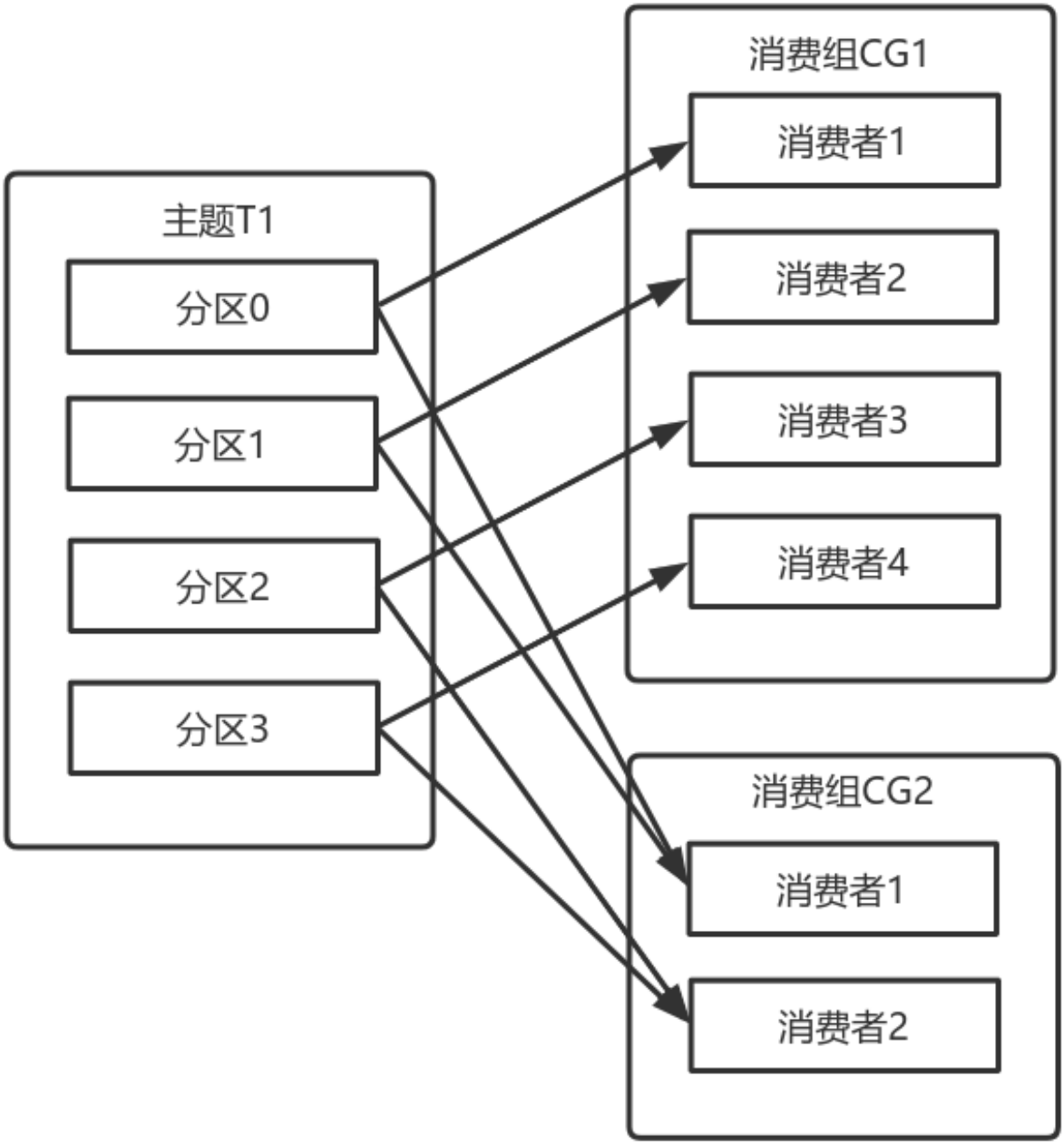


如果向消费组中添加更多的消费者，超过主题分区数量，则有一部分消费者就会闲置，不会接收任何消息。



向消费组添加消费者是横向扩展消费能力的主要方式。

必要时，需要为主题创建大量分区，在负载增长时可以加入更多的消费者。但是不要让消费者的数量超过主题分区的数量。



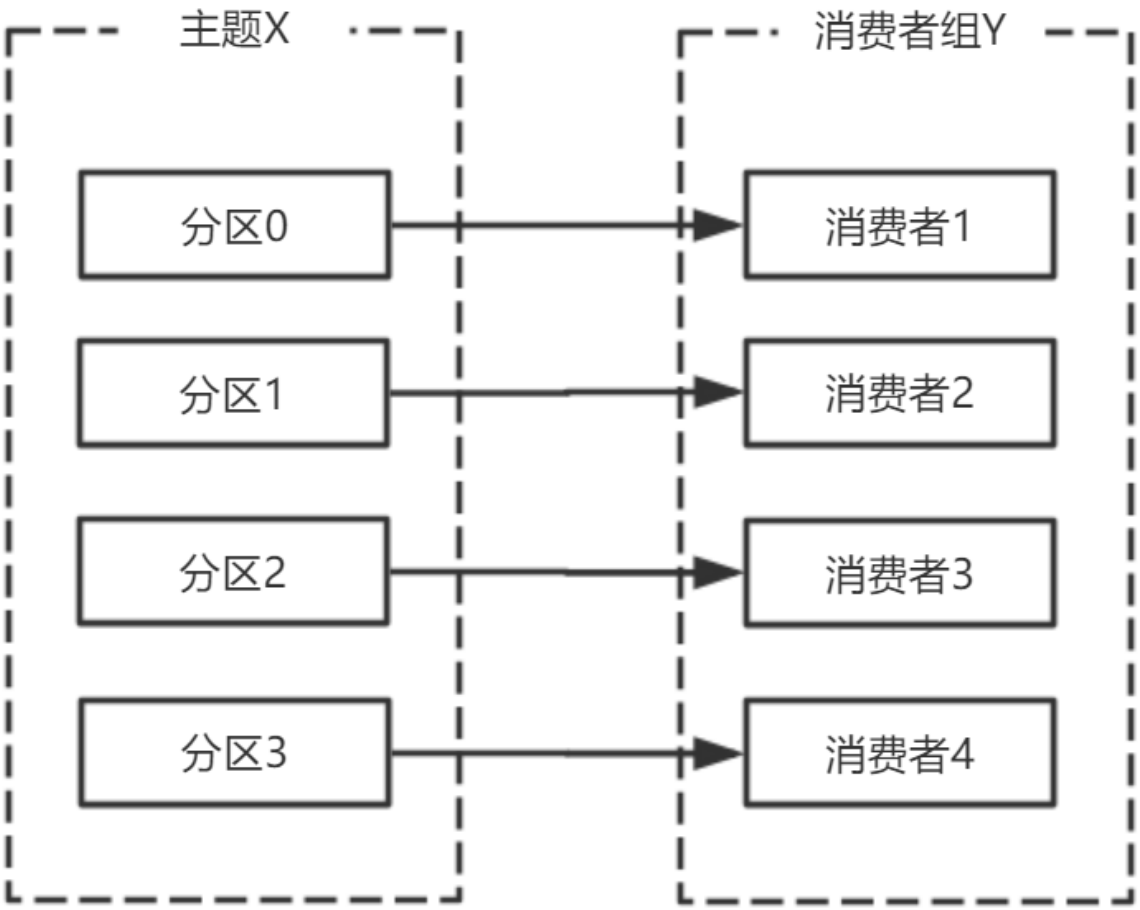
除了通过增加消费者来横向扩展单个应用的消费能力之外，经常出现多个应用程序从同一个主题消费的情况。

此时，每个应用都可以获取到所有的消息。只要保证每个应用都有自己的消费组，就可以让它们获取到主题所有的消息。

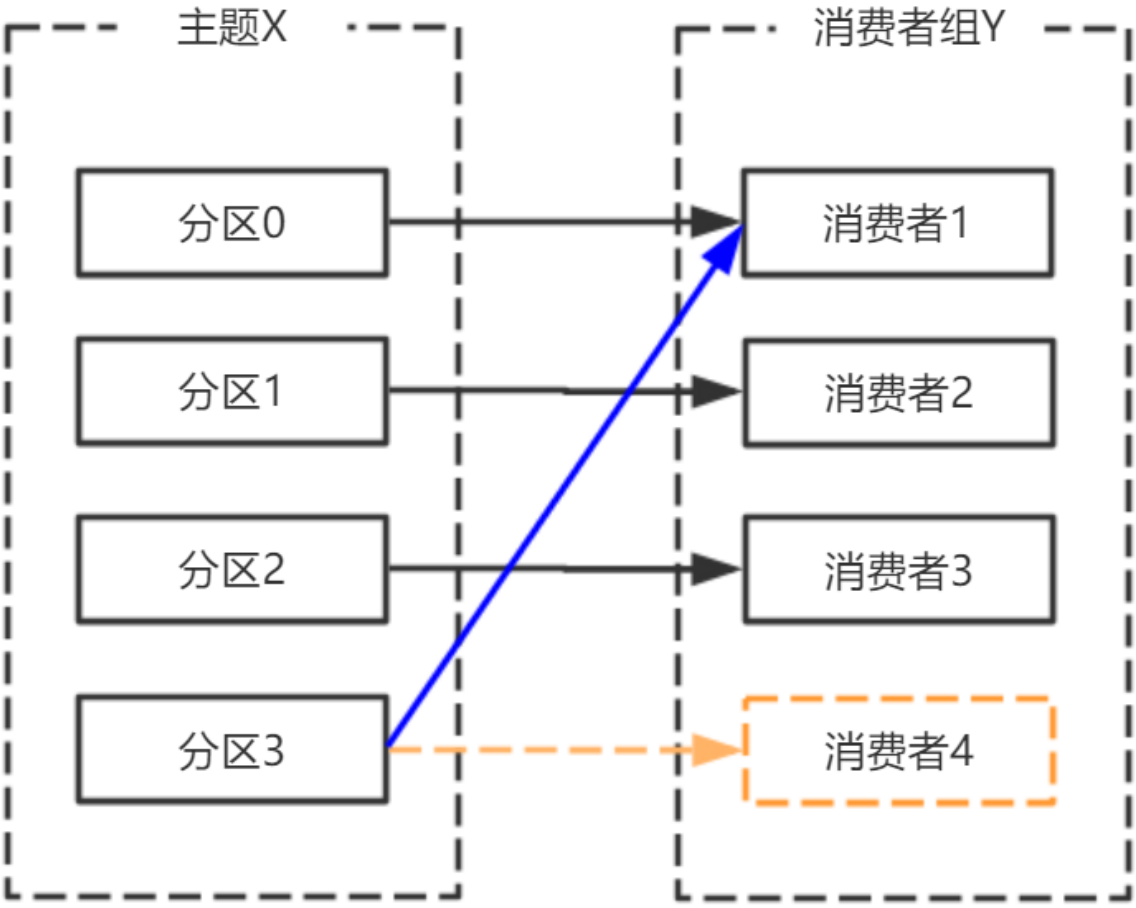
横向扩展消费者和消费组不会对性能造成负面影响。

为每个需要获取一个或多个主题全部消息的应用创建一个消费组，然后向消费组添加消费者来横向扩展消费能力和应用的处理能力，则每个消费者只处理一部分消息。

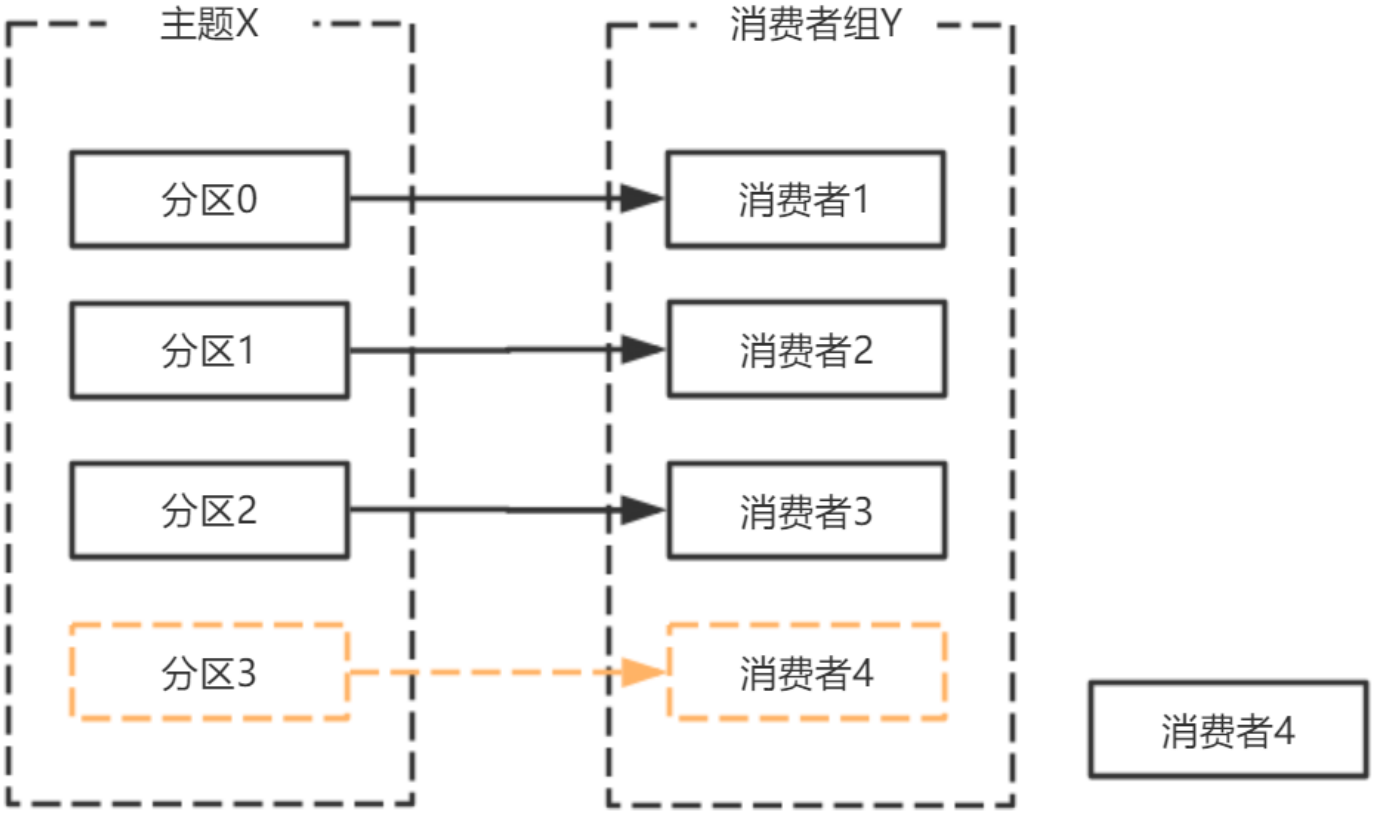
2.2.1.2 心跳机制



消费者宕机，退出消费组，触发再平衡，重新给消费组中的消费者分配分区。



由于broker宕机，主题X的分区3宕机，此时分区3没有Leader副本，触发再平衡，消费者4没有对应的主题分区，则消费者4闲置。



Kafka 的心跳是 Kafka Consumer 和 Broker 之间的健康检查，只有当 Broker Coordinator 正常时，Consumer 才会发送心跳。

Consumer 和 Rebalance 相关的 2 个配置参数：

参数	字段
session.timeout.ms	MemberMetadata.sessionTimeoutMs
max.poll.interval.ms	MemberMetadata.rebalanceTimeoutMs

broker 端，sessionTimeoutMs 参数

broker 处理心跳的逻辑在 `GroupCoordinator` 类中：如果心跳超期， broker coordinator 会把消费者从 group 中移除，并触发 rebalance。

```
private def completeAndScheduleNextHeartbeatExpiration(group: GroupMetadata, member: MemberMetadata) {
    // complete current heartbeat expectation
    member.latestHeartbeat = time.milliseconds()
    val memberKey = MemberKey(member.groupId, member.memberId)
    heartbeatPurgatory.checkAndComplete(memberKey)

    // reschedule the next heartbeat expiration deadline
    // 计算心跳截止时刻
    val newHeartbeatDeadline = member.latestHeartbeat + member.sessionTimeoutMs
    val delayedHeartbeat = new DelayedHeartbeat(this, group, member, newHeartbeatDeadline, member.sessionTimeoutMs)
    heartbeatPurgatory.tryCompleteElseWatch(delayedHeartbeat, Seq(memberKey))
  }

  // 心跳过期
  def onExpireHeartbeat(group: GroupMetadata, member: MemberMetadata, heartbeatDeadline: Long) {
    group.inLock {
      if (!shouldKeepMemberAlive(member, heartbeatDeadline)) {
        info(s"Member ${member.memberId} in group ${group.groupId} has failed, removing it from the group")
        removeMemberAndUpdateGroup(group, member)
      }
    }
  }

  private def shouldKeepMemberAlive(member: MemberMetadata, heartbeatDeadline: Long) =
    member.awaitingJoinCallback != null ||
      member.awaitingSyncCallback != null ||
      member.latestHeartbeat + member.sessionTimeoutMs > heartbeatDeadline
```

consumer 端：sessionTimeoutMs，rebalanceTimeoutMs 参数

如果客户端发现心跳超期，客户端会标记 coordinator 为不可用，并阻塞心跳线程；如果超过了 poll 消息的间隔超过了 rebalanceTimeoutMs，则 consumer 告知 broker 主动离开消费组，也会触发 rebalance

org.apache.kafka.clients.consumer.internals.AbstractCoordinator.HeartbeatThread

```
if (coordinatorUnknown()) {
                            if (findCoordinatorFuture != null ||
lookupCoordinator().failed())
                                // the immediate future check ensures that we backoff
properly in the case that no
                                // brokers are available to connect to.
                                AbstractCoordinator.this.wait(retryBackoffMs);
                        } else if (heartbeat.sessionTimeoutExpired(now)) {
                            // the session timeout has expired without seeing a
successful heartbeat, so we should
                            // probably make sure the coordinator is still healthy.
                            markCoordinatorUnknown();
                        } else if (heartbeat.pollTimeoutExpired(now)) {
                            // the poll timeout has expired, which means that the
foreground thread has stalled
                            // in between calls to poll(), so we explicitly leave the
group.
                            maybeLeaveGroup();
                        } else if (!heartbeat.shouldHeartbeat(now)) {
                            // poll again after waiting for the retry backoff in case
the heartbeat failed or the
                            // coordinator disconnected
                            AbstractCoordinator.this.wait(retryBackoffMs);
                        } else {
                            heartbeat.sentHeartbeat(now);

                            sendHeartbeatRequest().addListener(new
RequestFutureListener<Void>() {
                                @Override
                                public void onSuccess(Void value) {
                                    synchronized (AbstractCoordinator.this) {

 heartbeat.receiveHeartbeat(time.milliseconds());
                                    }
                                }

                                @Override
                                public void onFailure(RuntimeException e) {
                                    synchronized (AbstractCoordinator.this) {
                                        if (e instanceof
RebalanceInProgressException) {
                                            // it is valid to continue heartbeating
while the group is rebalancing. This
```

```
34                                                  // ensures that the coordinator keeps the
    member in the group for as long
35                                                  // as the duration of the rebalance
    timeout. If we stop sending heartbeats,
36                                                  // however, then the session timeout may
    expire before we can rejoin.
37
     heartbeat.receiveHeartbeat(time.milliseconds());
38                                              } else {
39                                                  heartbeat.failHeartbeat();
40
41                                                  // wake up the thread if it's sleeping to
    reschedule the heartbeat
42                                                  AbstractCoordinator.this.notify();
43                                              }
44                                          }
45                                      }
46                                  });
47                          }
```

2.2.2 消息接收

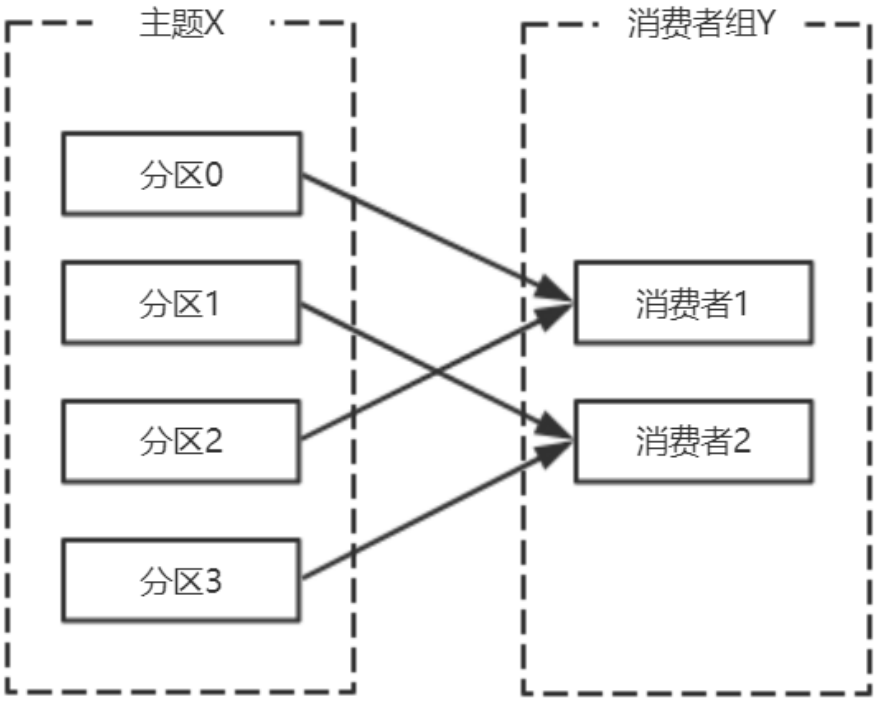
2.2.2.1 必要参数配置

参数	说明
bootstrap.servers	向Kafka集群建立初始连接用到的host/port列表。 客户端会使用这里列出的所有服务器进行集群其他服务器的发现，而不管是否指定了哪个服务器用作引导。 这个列表仅影响用来发现集群所有服务器的初始主机。 字符串形式：host1:port1,host2:port2,... 由于这组服务器仅用于建立初始链接，然后发现集群中的所有服务器，因此没有必要将集群中的所有地址写在这里。 一般最好两台，以防其中一台宕掉。
key.deserializer	key的反序列化类，该类需要实现 `org.apache.kafka.common.serialization.Deserializer` 接口。
value.deserializer	实现了 `org.apache.kafka.common.serialization.Deserializer` 接口的反序列化器， 用于对消息的value进行反序列化。
client.id	当从服务器消费消息的时候向服务器发送的id字符串。在ip/port基础上提供应用的逻辑名称，记录在服务端的请求日志中，用于追踪请求的源。
group.id	用于唯一标志当前消费者所属的消费组的字符串。 如果消费者使用组管理功能如subscribe(topic)或使用基于Kafka的偏移量管理策略，该项必须设置。
auto.offset.reset	当Kafka中没有初始偏移量或当前偏移量在服务器中不存在（如，数据被删除了），该如何处理? earliest：自动重置偏移量到最早的偏移量 latest：自动重置偏移量为最新的偏移量 none：如果消费组原来的（previous）偏移量不存在，则向消费者抛异常 anything：向消费者抛异常
enable.auto.commit	如果设置为true，消费者会自动周期性地向服务器提交偏移量。

2.2.2.2 订阅

2.2.2.2.1 主题和分区

- **Topic**，Kafka用于分类管理消息的逻辑单元，类似与MySQL的数据库。
- **Partition**，是Kafka下数据存储的基本单元，这个是物理上的概念。**同一个topic的数据，会被分散的存储到多个partition中**，这些partition可以在同一台机器上，也可以是在多台机器上。优势在于：有利于水平扩展，避免单台机器在磁盘空间和性能上的限制，同时可以通过复制来增加数据冗余性，提高容灾能力。为了做到均匀分布，通常partition的数量通常是Broker Server数量的整数倍。
- **Consumer Group**，同样是逻辑上的概念，是**Kafka实现单播和广播两种消息模型的手段**。保证一个消费组获取到特定主题的全部的消息。在消费组内部，若干个消费者消费主题分区的消息，消费组可以保证一个主题的每个分区只被消费组中的一个消费者消费。



consumer 采用 pull 模式从 broker 中读取数据。

采用 pull 模式，consumer 可自主控制消费消息的速率， 可以自己控制消费方式（批量消费/逐条消费），还可以选择不同的提交方式从而实现不同的传输语义。

consumer.subscribe("tp_demo_01,tp_demo_02")

2.2.2.3 反序列化

Kafka的broker中所有的消息都是字节数组，消费者获取到消息之后，需要先对消息进行反序列化处理，然后才能交给用户程序消费处理。

消费者的反序列化器包括key的和value的反序列化器。

key.deserializer

value.deserializer

IntegerDeserializer

StringDeserializer

需要实现`org.apache.kafka.common.serialization.Deserializer<T>`接口。

消费者从订阅的主题拉取消息：

consumer.poll(3_000);

在Fetcher类中，对拉取到的消息首先进行反序列化处理。

Fetcher.java

```
910         */
911     private ConsumerRecord<K, V> parseRecord(TopicPartition partition,
912                                              RecordBatch batch,
913                                              Record record) {
914         try {
915             long offset = record.offset();
916             long timestamp = record.timestamp();
917             TimestampType timestampType = batch.timestampType();
918             Headers headers = new RecordHeaders(record.headers());
919             ByteBuffer keyBytes = record.key();
920             byte[] keyByteArray = keyBytes == null ? null : Utils.toArray(keyBytes);
921             K key = keyBytes == null ? null : this.keyDeserializer.deserialize(partition.topic(), headers, keyByteArray);
922             ByteBuffer valueBytes = record.value();
923             byte[] valueByteArray = valueBytes == null ? null : Utils.toArray(valueBytes);
924             V value = valueBytes == null ? null : this.valueDeserializer.deserialize(partition.topic(), headers, valueByteArray);
925             return new ConsumerRecord<>(partition.topic(), partition.partition(), offset,
926                                         timestamp, timestampType, record.checksumOrNull(),
927                                         keyByteArray == null ? ConsumerRecord.NULL_SIZE : keyByteArray.length,
928                                         valueByteArray == null ? ConsumerRecord.NULL_SIZE : valueByteArray.length,
929                                         key, value, headers);
930         } catch (RuntimeException e) {
931             throw new SerializationException("Error deserializing key/value for partition " + partition +
932                     " at offset " + record.offset() + ". If needed, please seek past the record to continue consumption.", e);
933         }
934     }
```

Kafka默认提供了几个反序列化的实现：

`org.apache.kafka.common.serialization`包下包含了这几个实现：

```java
public class ByteArrayDeserializer implements Deserializer<byte[]> {

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    @Override
    public byte[] deserialize(String topic, byte[] data) {
        return data;
    }

    @Override
    public void close() {
        // nothing to do
    }
}
```

```java
public class ByteBufferDeserializer implements Deserializer<ByteBuffer> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public ByteBuffer deserialize(String topic, byte[] data) {
        if (data == null)
            return null;

        return ByteBuffer.wrap(data);
    }

    public void close() {
        // nothing to do
    }
}
```

```java
public class BytesDeserializer implements Deserializer<Bytes> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public Bytes deserialize(String topic, byte[] data) {
        if (data == null)
            return null;

        return new Bytes(data);
    }

    public void close() {
        // nothing to do
    }
}
```

```java
public class DoubleDeserializer implements Deserializer<Double> {

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    @Override
    public Double deserialize(String topic, byte[] data) {
        if (data == null)
            return null;
        if (data.length != 8) {
            throw new SerializationException("Size of data received by Deserializer is not 8");
        }

        long value = 0;
        for (byte b : data) {
            value <<= 8;
            value |= b & 0xFF;
        }
        return Double.longBitsToDouble(value);
    }

    @Override
    public void close() {
        // nothing to do
    }
}
```

```java
public class FloatDeserializer implements Deserializer<Float> {

    @Override
    public void configure(final Map<String, ?> configs, final boolean isKey) {
        // nothing to do
    }

    @Override
    public Float deserialize(final String topic, final byte[] data) {
        if (data == null)
            return null;
        if (data.length != 4) {
            throw new SerializationException("Size of data received by Deserializer is not 4");
        }

        int value = 0;
        for (byte b : data) {
            value <<= 8;
            value |= b & 0xFF;
        }
        return Float.intBitsToFloat(value);
    }

    @Override
    public void close() {
        // nothing to do
    }

}
```

```java
public class IntegerDeserializer implements Deserializer<Integer> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public Integer deserialize(String topic, byte[] data) {
        if (data == null)
            return null;
        if (data.length != 4) {
            throw new SerializationException("Size of data received by IntegerDeserializer is not 4");
        }

        int value = 0;
        for (byte b : data) {
            value <<= 8;
            value |= b & 0xFF;
        }
        return value;
    }

    public void close() {
        // nothing to do
    }
}


public class LongDeserializer implements Deserializer<Long> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public Long deserialize(String topic, byte[] data) {
        if (data == null)
            return null;
        if (data.length != 8) {
            throw new SerializationException("Size of data received by LongDeserializer is not 8");
        }

        long value = 0;
        for (byte b : data) {
            value <<= 8;
            value |= b & 0xFF;
        }
        return value;
    }

    public void close() {
        // nothing to do
    }
}
```

```java
public class ShortDeserializer implements Deserializer<Short> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public Short deserialize(String topic, byte[] data) {
        if (data == null)
            return null;
        if (data.length != 2) {
            throw new SerializationException("Size of data received by ShortDeserializer is not 2");
        }

        short value = 0;
        for (byte b : data) {
            value <<= 8;
            value |= b & 0xFF;
        }
        return value;
    }

    public void close() {
        // nothing to do
    }
}
```

```java
public class StringDeserializer implements Deserializer<String> {
    private String encoding = "UTF8";

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
        String propertyName = isKey ? "key.deserializer.encoding" : "value.deserializer.encoding";
        Object encodingValue = configs.get(propertyName);
        if (encodingValue == null)
            encodingValue = configs.get("deserializer.encoding");
        if (encodingValue != null && encodingValue instanceof String)
            encoding = (String) encodingValue;
    }

    @Override
    public String deserialize(String topic, byte[] data) {
        try {
            if (data == null)
                return null;
            else
                return new String(data, encoding);
        } catch (UnsupportedEncodingException e) {
            throw new SerializationException("Error when deserializing byte[] to string due to unsupported encoding " + encoding);
        }
    }

    @Override
    public void close() {
        // nothing to do
    }
}
```

2.2.2.3.1 自定义反序列化

自定义反序列化类，需要实现`org.apache.kafka.common.serialization.Deserializer<T>`接口。

`com.lagou.kafka.demo.deserializer.UserDeserializer`

```
package com.lagou.kafka.demo.deserializer;

import com.lagou.kafka.demo.entity.User;
import org.apache.kafka.common.serialization.Deserializer;

import java.nio.ByteBuffer;
import java.util.Map;

public class UserDeserializer implements Deserializer<User> {

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {

    }

    @Override
    public User deserialize(String topic, byte[] data) {

        ByteBuffer allocate = ByteBuffer.allocate(data.length);
        allocate.put(data);
        allocate.flip();
        int userId = allocate.getInt();
        int length = allocate.getInt();

        System.out.println(length);

        String username = new String(data, 8, length);

        return new User(userId, username);
    }

    @Override
    public void close() {

    }
}
```

`com.lagou.kafka.demo.consumer.MyConsumer`

```
package com.lagou.kafka.demo.consumer;

import com.lagou.kafka.demo.deserializer.UserDeserializer;
import com.lagou.kafka.demo.entity.User;
```

```java
import org.apache.kafka.clients.consumer.ConsumerConfig;
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.serialization.StringDeserializer;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Consumer;

public class MyConsumer {
    public static void main(String[] args) {

        Map<String, Object> configs = new HashMap<>();
        configs.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
        configs.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        configs.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, UserDeserializer.class);
        configs.put(ConsumerConfig.GROUP_ID_CONFIG, "consumer1");
        configs.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        configs.put(ConsumerConfig.CLIENT_ID_CONFIG, "con1");

        KafkaConsumer<String, User> consumer = new KafkaConsumer<String, User>(configs);

        consumer.subscribe(Collections.singleton("tp_user_01"));

        ConsumerRecords<String, User> records = consumer.poll(Long.MAX_VALUE);

        records.forEach(new Consumer<ConsumerRecord<String, User>>() {
            @Override
            public void accept(ConsumerRecord<String, User> record) {
                System.out.println(record.value());
            }
        });

        // 关闭消费者
        consumer.close();
    }
}
```

2.2.2.4 位移提交

1. Consumer需要向Kafka记录自己的位移数据，这个汇报过程称为`提交位移(Committing Offsets)`
2. Consumer 需要为分配给它的每个分区提交各自的位移数据

3. 位移提交的由Consumer端负责的，Kafka只负责保管。__consumer_offsets
4. 位移提交分为自动提交和手动提交
5. 位移提交分为同步提交和异步提交

2.2.2.4.1 自动提交

Kafka Consumer 后台提交

- 开启自动提交：`enable.auto.commit=true`
- 配置自动提交间隔：Consumer端：`auto.commit.interval.ms`，默认 5s

```
Map<String, Object> configs = new HashMap<>();
configs.put("bootstrap.servers", "node1:9092");
configs.put("group.id", "mygrp");
// 设置偏移量自动提交。自动提交是默认值。这里做示例。
configs.put("enable.auto.commit", "true");
// 偏移量自动提交的时间间隔
configs.put("auto.commit.interval.ms", "3000");
configs.put("key.deserializer", StringDeserializer.class);
configs.put("value.deserializer", StringDeserializer.class);
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(configs);
consumer.subscribe(Collections.singleton("tp_demo_01"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.println(record.topic()
                + "\t" + record.partition()
                + "\t" + record.offset()
                + "\t" + record.key()
                + "\t" + record.value());
    }
}
```

- 自动提交位移的顺序
    - 配置 enable.auto.commit = true
    - Kafka会保证在开始调用poll方法时，提交上次poll返回的所有消息
    - 因此自动提交不会出现消息丢失，但会`重复消费`
- 重复消费举例
    - Consumer 每 5s 提交 offset
    - 假设提交 offset 后的 3s 发生了 Rebalance
    - Rebalance 之后的所有 Consumer 从上一次提交的 offset 处继续消费
    - 因此 Rebalance 发生前 3s 的消息会被重复消费

2.2.2.4.2 异步提交

- 使用 KafkaConsumer#commitSync()：会提交 KafkaConsumer#poll() 返回的最新 offset
- 该方法为同步操作，等待直到 offset 被成功提交才返回

```
while (true) {
    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofSeconds(1));
    process(records); // 处理消息
    try {
        consumer.commitSync();
    } catch (CommitFailedException e) {
        handle(e); // 处理提交失败异常
    }
}
```

- commitSync 在处理完所有消息之后
- 手动同步提交可以控制offset提交的时机和频率
- 手动同步提交会：
    - 调用 commitSync 时，Consumer 处于阻塞状态，直到 Broker 返回结果
    - 会影响 TPS
    - 可以选择拉长提交间隔，但有以下问题
        - 会导致 Consumer 的提交频率下降
        - Consumer 重启后，会有更多的消息被消费

异步提交

- KafkaConsumer#commitAsync()

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(3_000);
    process(records); // 处理消息
    consumer.commitAsync((offsets, exception) -> {
        if (exception != null) {
            handle(exception);
        }
    });
}
```

- commitAsync出现问题不会自动重试
- 处理方式：

```
try {
    while(true) {
    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofSeconds(1));
    process(records); // 处理消息
    commitAysnc(); // 使用异步提交规避阻塞
    }
} catch(Exception e) {
    handle(e); // 处理异常
} finally {
    try {
        consumer.commitSync(); // 最后一次提交使用同步阻塞式提交
    } finally {
        consumer.close();
    }
}
```

2.2.2.5 消费者位移管理

Kafka中，消费者根据消息的位移顺序消费消息。

消费者的位移由消费者管理，可以存储于zookeeper中，也可以存储于Kafka主题__consumer_offsets中。

Kafka提供了消费者API，让消费者可以管理自己的位移。

API如下：KafkaConsumer<K, V>

项目	细节
API	public void assign(Collection<TopicPartition> partitions)
说明	给当前消费者手动分配一系列主题分区。 手动分配分区不支持增量分配，如果先前有分配分区，则该操作会覆盖之前的分配。 如果给出的主题分区是空的，则等价于调用unsubscribe方法。 手动分配主题分区的方法不使用消费组管理功能。当消费组成员变了，或者集群或主题的元数据改变了，不会触发分区分配的再平衡。 手动分区分配assign(Collection)不能和自动分区分配subscribe(Collection, ConsumerRebalanceListener)一起使用。 如果启用了自动提交偏移量，则在新的分区分配替换旧的分区分配之前，会对旧的分区分配中的消费偏移量进行异步提交。
API	public Set<TopicPartition> assignment()
说	获取给当前消费者分配的分区集合。如果订阅是通过调用assign方法直接分配主题分区，则返回相同的集

明	合。如果使用了主题订阅，该方法返回当前分配给该消费者的主题分区集合。如果分区订阅还没开始进行分区分配，或者正在重新分配分区，则会返回none。
API	public Map<String, List<PartitionInfo>> listTopics()
说明	获取对用户授权的所有主题分区元数据。该方法会对服务器发起远程调用。
API	public List<PartitionInfo> partitionsFor(String topic)
说明	获取指定主题的分区元数据。如果当前消费者没有关于该主题的元数据，就会对服务器发起远程调用。
API	public Map<TopicPartition, Long> beginningOffsets(Collection<TopicPartition> partitions)
说明	对于给定的主题分区，列出它们第一个消息的偏移量。 注意，如果指定的分区不存在，该方法可能会永远阻塞。 该方法不改变分区的当前消费者偏移量。
API	public void seekToEnd(Collection<TopicPartition> partitions)
说明	将偏移量移动到每个给定分区的最后一个。 该方法延迟执行，只有当调用过poll方法或position方法之后才可以使用。 如果没有指定分区，则将当前消费者分配的所有分区的消费者偏移量移动到最后。 如果设置了隔离级别为：isolation.level=read_committed，则会将分区的消费偏移量移动到最后一个稳定的偏移量，即下一个要消费的消息现在还是未提交状态的事务消息。
API	public void seek(TopicPartition partition, long offset)
说明	将给定主题分区的消费偏移量移动到指定的偏移量，即当前消费者下一条要消费的消息偏移量。 若该方法多次调用，则最后一次的覆盖前面的。 如果在消费中间随意使用，可能会丢失数据。
API	public long position(TopicPartition partition)
说明	检查指定主题分区的消费偏移量
API	public void seekToBeginning(Collection<TopicPartition> partitions)
说明	将给定每个分区的消费者偏移量移动到它们的起始偏移量。该方法懒执行，只有当调用过poll方法或position方法之后才会执行。如果没有提供分区，则将所有分配给当前消费者的分区消费偏移量移动到起始偏移量。

1. 准备数据

```
# 生成消息文件
[root@node1 ~]# for i in `seq 60`; do echo "hello lagou $i" >> nm.txt; done
# 创建主题，三个分区，每个分区一个副本
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --create --topic tp_demo_01 --partitions 3 --replication-factor 1
# 将消息生产到主题中
[root@node1 ~]# kafka-console-producer.sh --broker-list node1:9092 --topic tp_demo_01 < nm.txt
```

2. API实战

```
package com.lagou.kafka.demo.consumer;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.Node;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.util.*;

/**
 * # 生成消息文件
 * [root@node1 ~]# for i in `seq 60`; do echo "hello lagou $i" >> nm.txt; done
 * # 创建主题，三个分区，每个分区一个副本
 * [root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --create --topic tp_demo_01 --partitions 3 --replication-factor 1
 * # 将消息生产到主题中
 * [root@node1 ~]# kafka-console-producer.sh --broker-list node1:9092 --topic tp_demo_01 < nm.txt
 *
 * 消费者位移管理
 */
public class MyConsumerMgr1 {

    public static void main(String[] args) {
        Map<String, Object> configs = new HashMap<>();
        configs.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
        configs.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        configs.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);

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

        /**
```

```
33          * 给当前消费者手动分配一系列主题分区。
34          * 手动分配分区不支持增量分配，如果先前有分配分区，则该操作会覆盖之前的分配。
35          * 如果给出的主题分区是空的，则等价于调用unsubscribe方法。
36          * 手动分配主题分区的方法不使用消费组管理功能。当消费组成员变了，或者集群或主题的元数据改变了，不会触发分区分配的再平衡。
37          *
38          * 手动分区分配assign(Collection)不能和自动分区分配subscribe(Collection, ConsumerRebalanceListener)一起使用。
39          * 如果启用了自动提交偏移量，则在新的分区分配替换旧的分区分配之前，会对旧的分区分配中的消费偏移量进行异步提交。
40          *
41          */
42  //        consumer.assign(Arrays.asList(new TopicPartition("tp_demo_01", 0)));
43  //
44  //        Set<TopicPartition> assignment = consumer.assignment();
45  //        for (TopicPartition topicPartition : assignment) {
46  //            System.out.println(topicPartition);
47  //        }
48
49          // 获取对用户授权的所有主题分区元数据。该方法会对服务器发起远程调用。
50  //        Map<String, List<PartitionInfo>> stringListMap = consumer.listTopics();
51  //
52  //        stringListMap.forEach((k, v) -> {
53  //            System.out.println("主题: " + k);
54  //            v.forEach(info -> {
55  //                System.out.println(info);
56  //            });
57  //        });
58
59  //        Set<String> strings = consumer.listTopics().keySet();
60  //
61  //        strings.forEach(topicName -> {
62  //            System.out.println(topicName);
63  //        });
64
65  //        List<PartitionInfo> partitionInfos = consumer.partitionsFor("tp_demo_01");
66  //        for (PartitionInfo partitionInfo : partitionInfos) {
67  //            Node leader = partitionInfo.leader();
68  //            System.out.println(leader);
69  //            System.out.println(partitionInfo);
70  //            // 当前分区在线副本
71  //            Node[] nodes = partitionInfo.inSyncReplicas();
72  //            // 当前分区下线副本
73  //            Node[] nodes1 = partitionInfo.offlineReplicas();
74  //        }
75
76          // 手动分配主题分区给当前消费者
77          consumer.assign(Arrays.asList(
78                  new TopicPartition("tp_demo_01", 0),
79                  new TopicPartition("tp_demo_01", 1),
80                  new TopicPartition("tp_demo_01", 2)
```

```
        ));

        // 列出当前主题分配的所有主题分区
//        Set<TopicPartition> assignment = consumer.assignment();
//        assignment.forEach(k -> {
//            System.out.println(k);
//        });

        // 对于给定的主题分区，列出它们第一个消息的偏移量。
        // 注意，如果指定的分区不存在，该方法可能会永远阻塞。
        // 该方法不改变分区的当前消费者偏移量。
//        Map<TopicPartition, Long> topicPartitionLongMap =
consumer.beginningOffsets(consumer.assignment());
//
//        topicPartitionLongMap.forEach((k, v) -> {
//            System.out.println("主题: " + k.topic() + "\t分区: " + k.partition() +
"偏移量\t" + v);
//        });

        // 将偏移量移动到每个给定分区的最后一个。
        // 该方法延迟执行，只有当调用过poll方法或position方法之后才可以使用。
        // 如果没有指定分区，则将当前消费者分配的所有分区的消费者偏移量移动到最后。
        // 如果设置了隔离级别为: isolation.level=read_committed，则会将分区的消费偏移量移动到
        // 最后一个稳定的偏移量，即下一个要消费的消息现在还是未提交状态的事务消息。
//        consumer.seekToEnd(consumer.assignment());

        // 将给定主题分区的消费偏移量移动到指定的偏移量，即当前消费者下一条要消费的消息偏移量。
        // 若该方法多次调用，则最后一次的覆盖前面的。
        // 如果在消费中间随意使用，可能会丢失数据。
//        consumer.seek(new TopicPartition("tp_demo_01", 1), 10);
//
//        // 检查指定主题分区的消费偏移量
//        long position = consumer.position(new TopicPartition("tp_demo_01", 1));
//        System.out.println(position);

        consumer.seekToEnd(Arrays.asList(new TopicPartition("tp_demo_01", 1)));
        // 检查指定主题分区的消费偏移量
        long position = consumer.position(new TopicPartition("tp_demo_01", 1));
        System.out.println(position);

        // 关闭生产者
        consumer.close();
    }

}
```

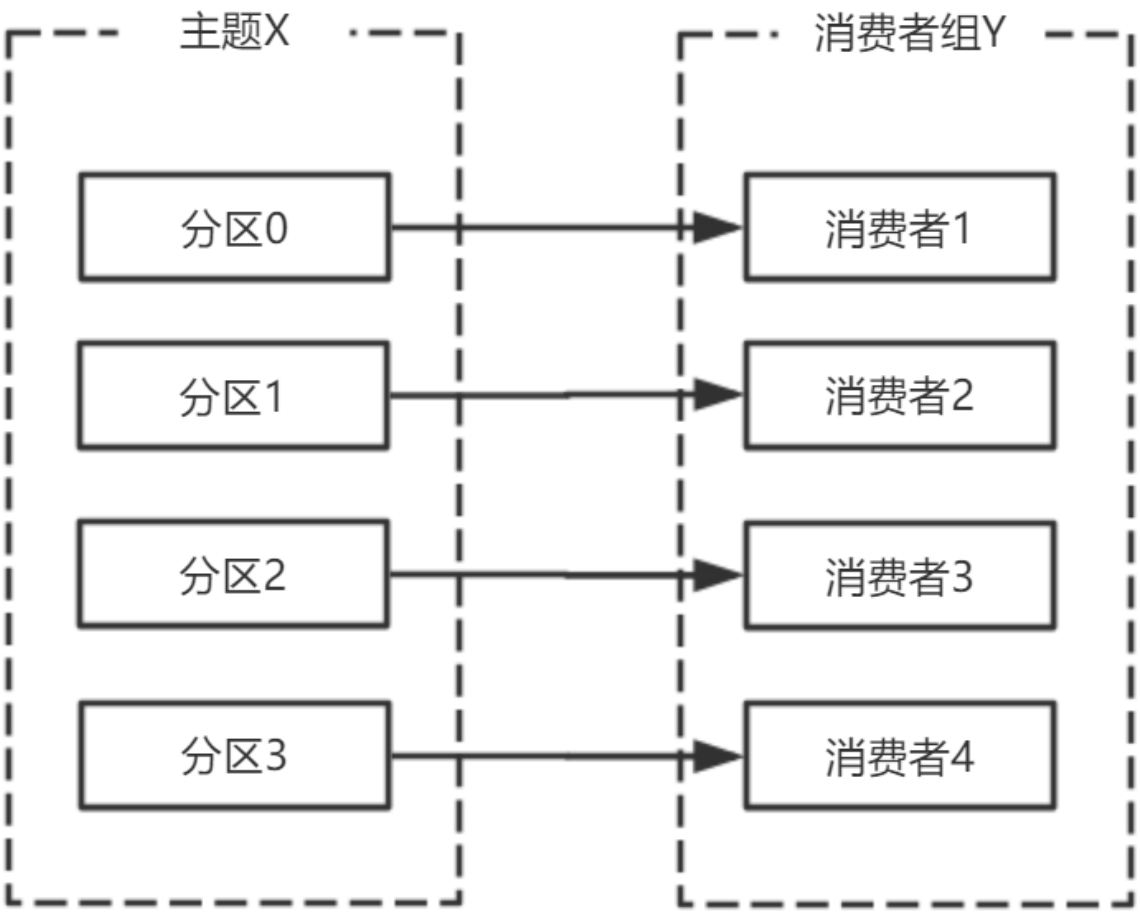
2.2.2.6 再均衡

重平衡可以说是kafka为人诟病最多的一个点了。

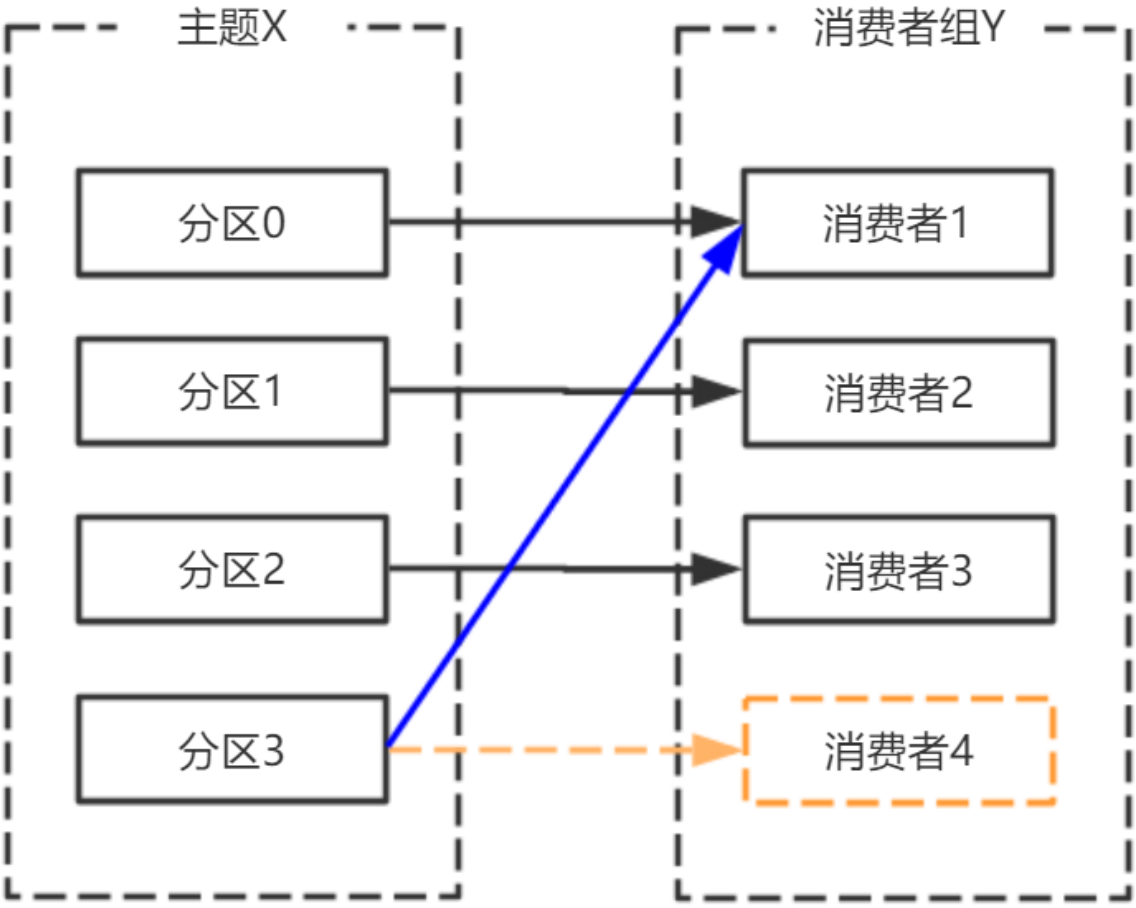
重平衡其实就是一个协议，它规定了如何让消费者组下的所有消费者来分配topic中的每一个分区。比如一个topic有100个分区，一个消费者组内有20个消费者，在协调者的控制下让组内每一个消费者分配到5个分区，这个分配的过程就是重平衡。

重平衡的触发条件主要有三个：

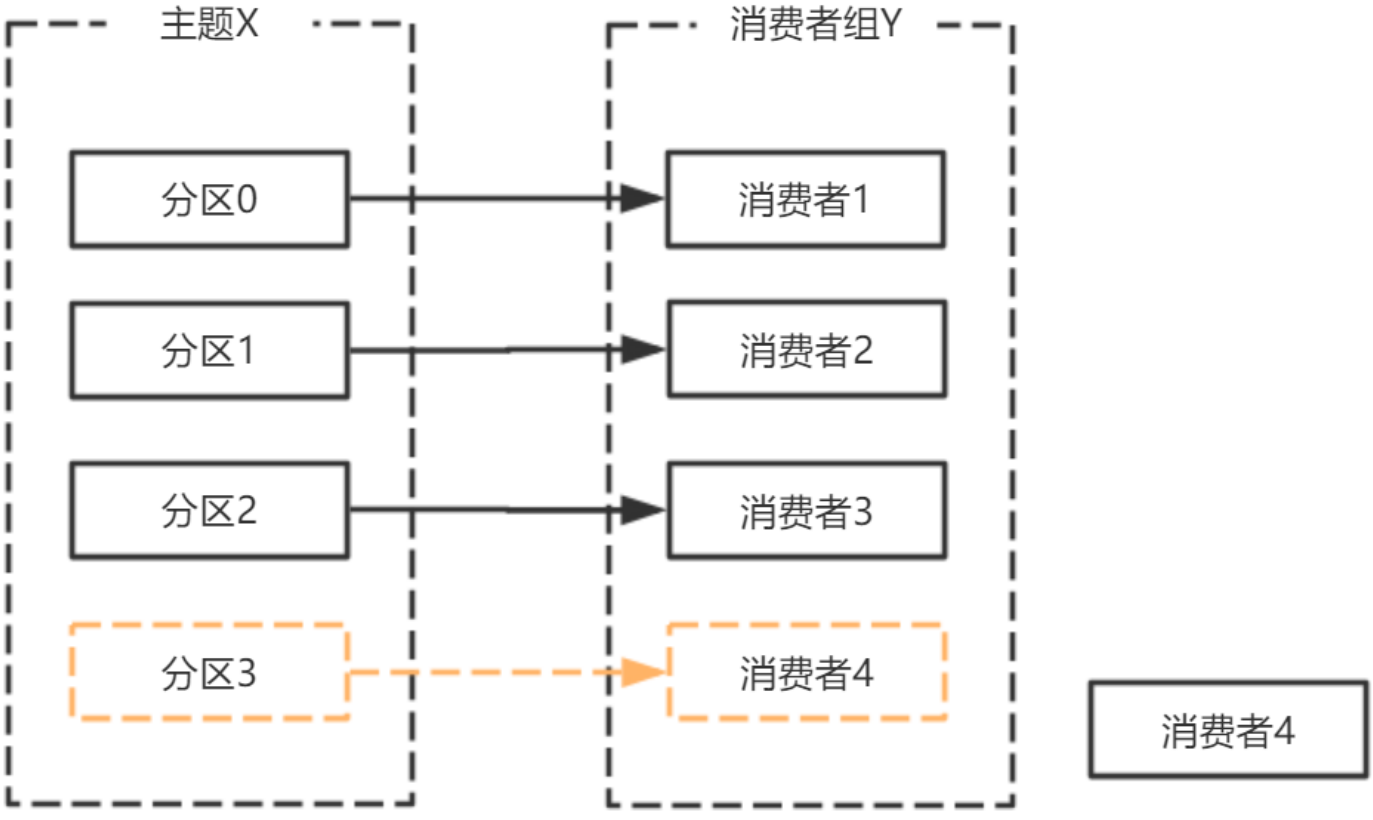
1. 消费者组内成员发生变更，这个变更包括了增加和减少消费者，比如消费者宕机退出消费组。
2. 主题的分区数发生变更，kafka目前只支持增加分区，当增加的时候就会触发重平衡
3. 订阅的主题发生变化，当消费者组使用正则表达式订阅主题，而恰好又新建了对应的主题，就会触发重平衡



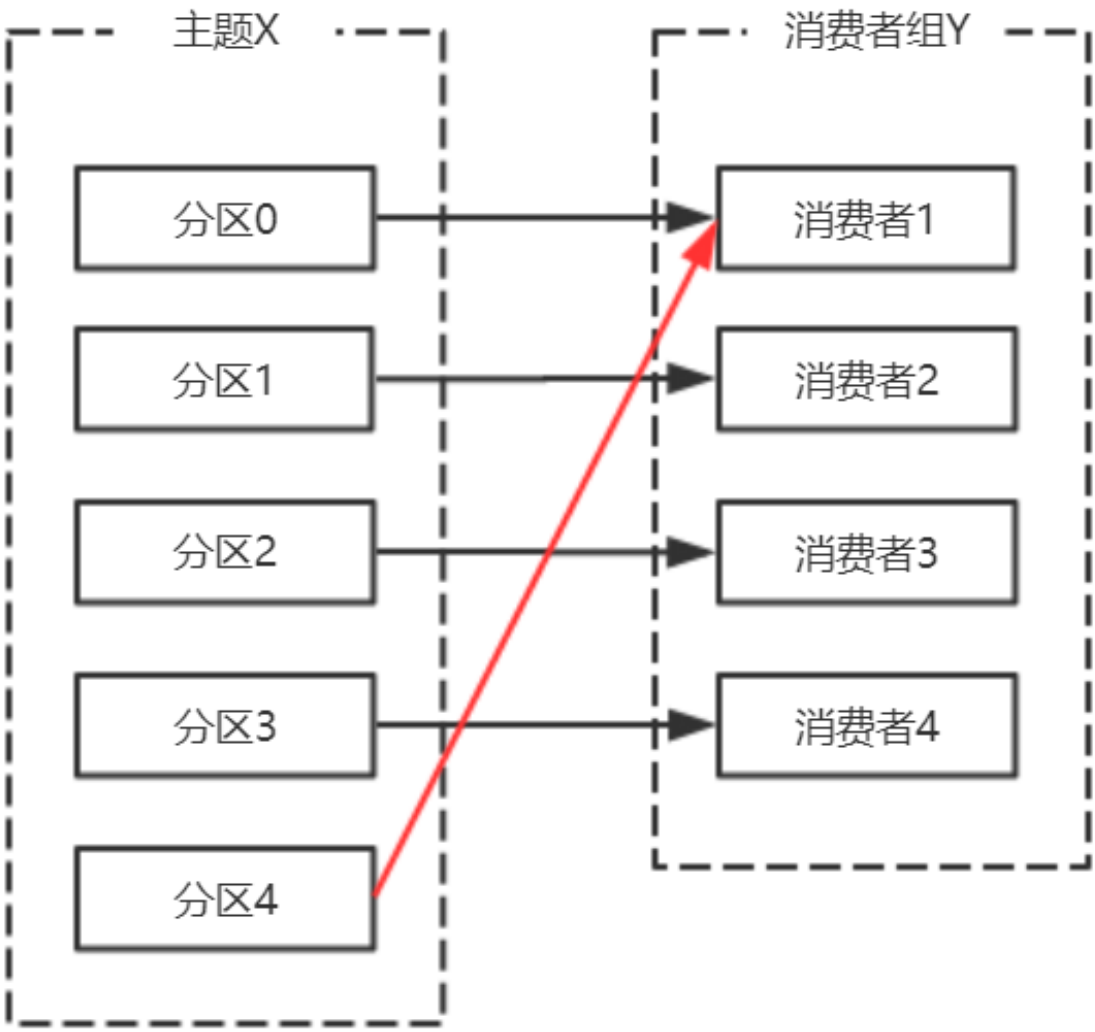
消费者宕机，退出消费组，触发再平衡，重新给消费组中的消费者分配分区。



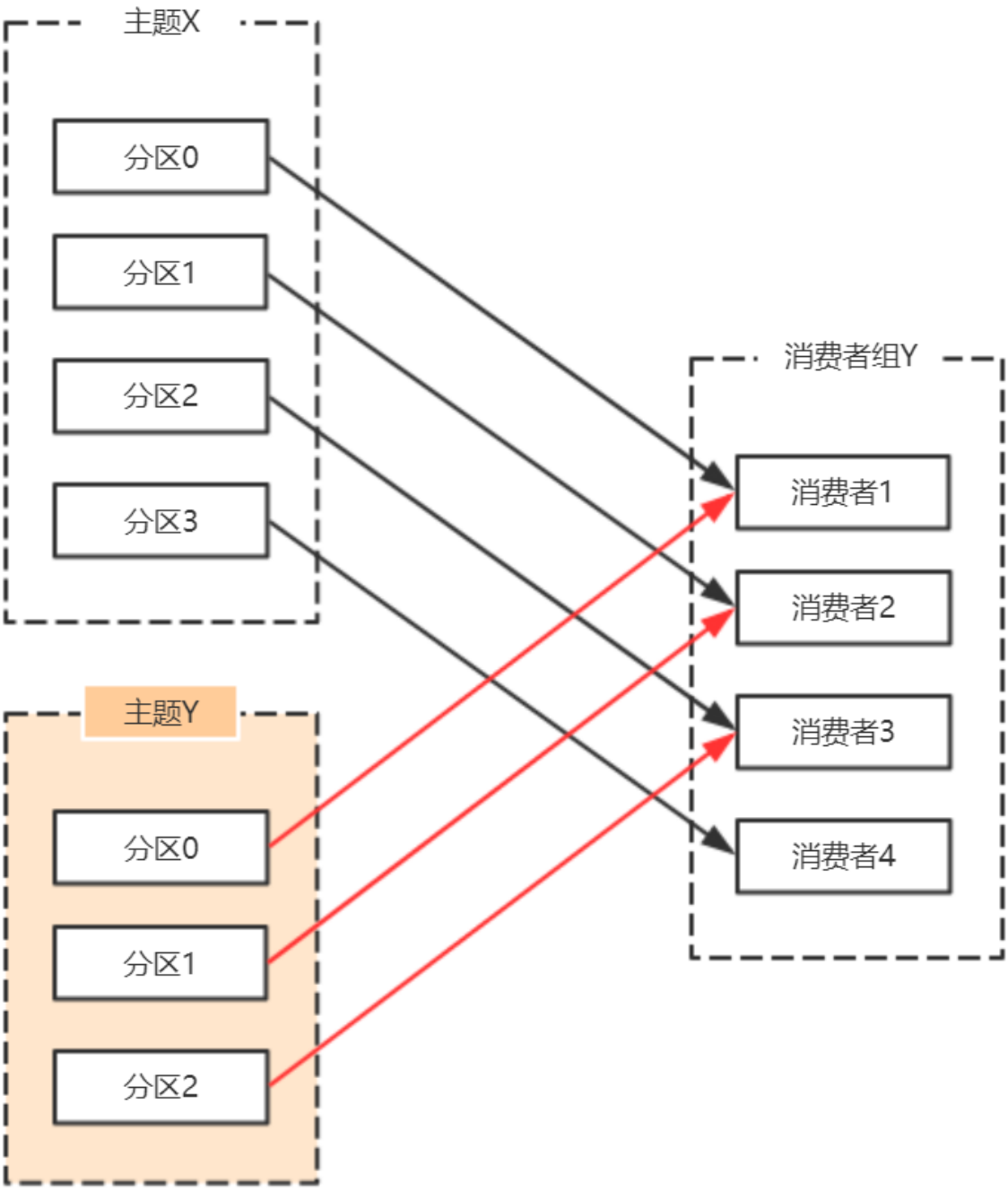
由于broker宕机，主题X的分区3宕机，此时分区3没有Leader副本，触发再平衡，消费者4没有对应的主题分区，则消费者4闲置。



主题增加分区，需要主题分区和消费组进行再均衡。



由于使用正则表达式订阅主题，当增加的主题匹配正则表达式的时候，也要进行再均衡。



为什么说重平衡为人诟病呢？**因为重平衡过程中，消费者无法从kafka消费消息，这对kafka的TPS影响极大，而如果kafka集内节点较多，比如数百个，那重平衡可能会耗时极多。数分钟到数小时都有可能，而这段时间kafka基本处于不可用状态。**所以在实际环境中，应该尽量避免重平衡发生。

避免重平衡

要说完全避免重平衡，是不可能，因为你无法完全保证消费者不会故障。而消费者故障其实也是最常见的引发重平衡的地方，所以我们需要保证**尽力避免消费者故障**。

而其他几种触发重平衡的方式，增加分区，或是增加订阅的主题，抑或是增加消费者，更多的是主动控制。

如果消费者真正挂掉了，就没办法了，但实际中，会有一些情况，**kafka错误地认为**一个正常的消费者已经挂掉了，我们要的就是避免这样的情况出现。

首先要知道哪些情况会出现错误判断挂掉的情况。

在分布式系统中，通常是通过心跳来维持分布式系统的，kafka也不例外。

在分布式系统中，由于网络问题你不清楚没接收到心跳，是因为对方真正挂了还是只是因为负载过重没来得及发生心跳或是网络堵塞。所以一般会约定一个时间，超时即判定对方挂了。**而在kafka消费者场景中，session.timout.ms参数就是规定这个超时时间是多少。**

还有一个参数，**heartbeat.interval.ms，这个参数控制发送心跳的频率**，频率越高越不容易被误判，但也会消耗更多资源。

此外，还有最后一个参数，**max.poll.interval.ms**，消费者poll数据后，需要一些处理，再进行拉取。如果两次拉取时间间隔超过这个参数设置的值，那么消费者就会被踢出消费者组。也就是说，拉取，然后处理，这个处理的时间不能超过`max.poll.interval.ms`这个参数的值。这个参数的默认值是5分钟，而如果消费者接收到数据后会执行耗时的操作，则应该将其设置得大一些。

三个参数，

session.timout.ms控制心跳超时时间，

heartbeat.interval.ms控制心跳发送频率，

max.poll.interval.ms控制poll的间隔。

这里给出一个相对较为合理的配置，如下：

- session.timout.ms：设置为6s
- heartbeat.interval.ms：设置2s
- max.poll.interval.ms：推荐为消费者处理消息最长耗时再加1分钟

2.2.2.7 消费者拦截器

消费者在拉取了分区消息之后，要首先经过反序列化器对key和value进行反序列化处理。

处理完之后，如果消费端设置了拦截器，则需要经过拦截器的处理之后，才能返回给消费者应用程序进行处理。

KafkaConsumer.java

```
1120                client.pollNoWakeup();
1121
1122            // 如果拦截器是null，表示没有配置消费者拦截
1123            if (this.interceptors == null)
1124                return new ConsumerRecords<>(records);
1125            else
1126                // 如果设置了消费端拦截器，则在此时调用消费者拦截器处理完之后，在返回给消费者处理。
1127                return this.interceptors.onConsume(new ConsumerRecords<>(records));
1128        }
```

消费端定义消息拦截器，需要实现`org.apache.kafka.clients.consumer.ConsumerInterceptor<K, V>`接口。

1. 一个可插拔接口，允许拦截甚至更改消费者接收到的消息。首要的用例在于将第三方组件引入消费者应用程序，用于定制的监控、日志处理等。

2. 该接口的实现类通过configre方法获取消费者配置的属性，如果消费者配置中没有指定clientID，还可以获取KafkaConsumer生成的clientId。获取的这个配置是跟其他拦截器共享的，需要保证不会在各个拦截器之间产生冲突。
3. ConsumerInterceptor方法抛出的异常会被捕获、记录，但是不会向下传播。如果用户配置了错误的key或value类型参数，消费者不会抛出异常，而仅仅是记录下来。
4. ConsumerInterceptor回调发生在org.apache.kafka.clients.consumer.KafkaConsumer#poll(long)方法同一个线程。

该接口中有如下方法：

```
package org.apache.kafka.clients.consumer;


import org.apache.kafka.common.Configurable;
import org.apache.kafka.common.TopicPartition;

import java.util.Map;

public interface ConsumerInterceptor<K, V> extends Configurable {

    /**
     *
     * 该方法在poll方法返回之前调用。调用结束后poll方法就返回消息了。
     *
     * 该方法可以修改消费者消息，返回新的消息。拦截器可以过滤收到的消息或生成新的消息。
     * 如果有多个拦截器，则该方法按照KafkaConsumer的configs中配置的顺序调用。
     *
     * @param records 由上个拦截器返回的由客户端消费的消息。
     */
    public ConsumerRecords<K, V> onConsume(ConsumerRecords<K, V> records);

    /**
     * 当消费者提交偏移量时，调用该方法。
     * 该方法抛出的任何异常调用者都会忽略。
     */
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets);

    public void close();
}
```

代码实现：

```
package com.lagou.kafka.demo.consumer;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
```

```java

import java.util.Collections;
import java.util.Properties;

public class MyConsumer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
        props.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        props.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        props.setProperty(ConsumerConfig.GROUP_ID_CONFIG, "mygrp");
//        props.setProperty(ConsumerConfig.CLIENT_ID_CONFIG, "myclient");
        // 如果在kafka中找不到当前消费者的偏移量，则设置为最旧的
        props.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // 配置拦截器
        // One -> Two -> Three，接收消息和发送偏移量确认都是这个顺序
        props.setProperty(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
                        "com.lagou.kafka.demo.interceptor.OneInterceptor" +
                        ",com.lagou.kafka.demo.interceptor.TwoInterceptor" +
                        ",com.lagou.kafka.demo.interceptor.ThreeInterceptor"
                );

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

        // 订阅主题
        consumer.subscribe(Collections.singleton("tp_demo_01"));

        while (true) {
            final ConsumerRecords<String, String> records = consumer.poll(3_000);

            records.forEach(record -> {
                System.out.println(record.topic()
                        + "\t" + record.partition()
                        + "\t" + record.offset()
                        + "\t" + record.key()
                        + "\t" + record.value());
            });

//            consumer.commitAsync();
//            consumer.commitSync();

        }

//        consumer.close();

    }
}
```

```java
package com.lagou.kafka.demo.interceptor;

import org.apache.kafka.clients.consumer.ConsumerInterceptor;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.common.TopicPartition;

import java.util.Map;

public class OneInterceptor implements ConsumerInterceptor<String, String> {
    @Override
    public ConsumerRecords<String, String> onConsume(ConsumerRecords<String, String>
records) {
        // poll方法返回结果之前最后要调用的方法

        System.out.println("One -- 开始");

        // 消息不做处理，直接返回
        return records;
    }

    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets) {
        // 消费者提交偏移量的时候，经过该方法
        System.out.println("One -- 结束");
    }

    @Override
    public void close() {
        // 用于关闭该拦截器用到的资源，如打开的文件，连接的数据库等
    }

    @Override
    public void configure(Map<String, ?> configs) {
        // 用于获取消费者的设置参数
        configs.forEach((k, v) -> {
            System.out.println(k + "\t" + v);
        });
    }
}
```

```java
package com.lagou.kafka.demo.interceptor;

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

```
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.common.TopicPartition;

import java.util.Map;

public class TwoInterceptor implements ConsumerInterceptor<String, String> {
    @Override
    public ConsumerRecords<String, String> onConsume(ConsumerRecords<String, String>
records) {
        // poll方法返回结果之前最后要调用的方法

        System.out.println("Two -- 开始");

        // 消息不做处理，直接返回
        return records;
    }

    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets) {
        // 消费者提交偏移量的时候，经过该方法
        System.out.println("Two -- 结束");
    }

    @Override
    public void close() {
        // 用于关闭该拦截器用到的资源，如打开的文件，连接的数据库等
    }

    @Override
    public void configure(Map<String, ?> configs) {
        // 用于获取消费者的设置参数
        configs.forEach((k, v) -> {
            System.out.println(k + "\t" + v);
        });
    }
}
```

```
package com.lagou.kafka.demo.interceptor;

import org.apache.kafka.clients.consumer.ConsumerInterceptor;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.common.TopicPartition;

import java.util.Map;

public class ThreeInterceptor implements ConsumerInterceptor<String, String> {
    @Override
```

```
12      public ConsumerRecords<String, String> onConsume(ConsumerRecords<String, String>
    records) {
13          // poll方法返回结果之前最后要调用的方法
14
15          System.out.println("Three -- 开始");
16
17          // 消息不做处理，直接返回
18          return records;
19      }
20
21      @Override
22      public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets) {
23          // 消费者提交偏移量的时候，经过该方法
24          System.out.println("Three -- 结束");
25      }
26
27      @Override
28      public void close() {
29          // 用于关闭该拦截器用到的资源，如打开的文件，连接的数据库等
30      }
31
32      @Override
33      public void configure(Map<String, ?> configs) {
34          // 用于获取消费者的设置参数
35          configs.forEach((k, v) -> {
36              System.out.println(k + "\t" + v);
37          });
38      }
39  }
```

2.2.2.8 消费者参数配置补充

配置项	说明
bootstrap.servers	建立到Kafka集群的初始连接用到的host/port列表。 客户端会使用这里指定的所有的host/port来建立初始连接。 这个配置仅会影响发现集群所有节点的初始连接。 形式：host1:port1,host2:port2... 这个配置中不需要包含集群中所有的节点信息。 最好不要配置一个，以免配置的这个节点宕机的时候连不上。
group.id	用于定义当前消费者所属的消费组的唯一字符串。 如果使用了消费组的功能`subscribe(topic)`， 或使用了基于Kafka的偏移量管理机制，则应该配置group.id。
auto.commit.interval.ms	如果设置了`enable.auto.commit`的值为true， 则该值定义了消费者偏移量向Kafka提交的频率。
	如果Kafka中没有初始偏移量或当前偏移量在服务器中不存在 （比如数据被删掉了）：

auto.offset.reset	earliest：自动重置偏移量到最早的偏移量。 latest：自动重置偏移量到最后一个 none：如果没有找到该消费组以前的偏移量没有找到，就抛异常。 其他值：向消费者抛异常。
fetch.min.bytes	服务器对每个拉取消息的请求返回的数据量最小值。 如果数据量达不到这个值，请求等待，以让更多的数据累积， 达到这个值之后响应请求。 默认设置是1个字节，表示只要有一个字节的数据， 就立即响应请求，或者在没有数据的时候请求超时。 将该值设置为大一点儿的数字，会让服务器等待稍微 长一点儿的时间以累积数据。 如此则可以提高服务器的吞吐量，代价是额外的延迟时间。
fetch.max.wait.ms	如果服务器端的数据量达不到`fetch.min.bytes`的话， 服务器端不能立即响应请求。 该时间用于配置服务器端阻塞请求的最大时长。
fetch.max.bytes	服务器给单个拉取请求返回的最大数据量。 消费者批量拉取消息，如果第一个非空消息批次的值比该值大， 消息批也会返回，以让消费者可以接着进行。 即该配置并不是绝对的最大值。 broker可以接收的消息批最大值通过 `message.max.bytes`(broker配置) 或`max.message.bytes`(主题配置)来指定。 需要注意的是，消费者一般会并发拉取请求。
enable.auto.commit	如果设置为true，则消费者的偏移量会周期性地在后台提交。
connections.max.idle.ms	在这个时间之后关闭空闲的连接。
check.crcs	自动计算被消费的消息的CRC32校验值。 可以确保在传输过程中或磁盘存储过程中消息没有被破坏。 它会增加额外的负载，在追求极致性能的场合禁用。
exclude.internal.topics	是否内部主题应该暴露给消费者。如果该条目设置为true， 则只能先订阅再拉取。
isolation.level	控制如何读取事务消息。 如果设置了`read_committed`，消费者的poll()方法只会 返回已经提交的事务消息。 如果设置了`read_uncommitted`(默认值)， 消费者的poll方法返回所有的消息，即使是已经取消的事务消息。 非事务消息以上两种情况都返回。 消息总是以偏移量的顺序返回。 `read_committed`只能返回到达LSO的消息。 在LSO之后出现的消息只能等待相关的事务提交之后才能看到。 结果，`read_committed`模式，如果有为提交的事务， 消费者不能读取到直到HW的消息。 `read_committed`的seekToEnd方法返回LSO。
heartbeat.interval.ms	当使用消费组的时候，该条目指定消费者向消费者协调器 发送心跳的时间间隔。 心跳是为了确保消费者会话的活跃状态， 同时在消费者加入或离开消费组的时候方便进行再平衡。 该条目的值必须小于`session.timeout.ms`，也不应该高于`session.timeout.ms`的1/3。 可以将其调整得更小，以控制正常重新平衡的预期时间。
session.timeout.ms	当使用Kafka的消费组的时候，消费者周期性地向broker发送心跳数据， 表明自己的存在。 如果经过该超时时间还没有收到消费者的心跳， 则broker将消费者从消费组移除，并启动再平衡。 该值必须在broker配置`group.min.session.timeout.ms`和`group.max.session.timeout.ms`之间。
max.poll.records	一次调用poll()方法返回的记录最大数量。
max.poll.interval.ms	使用消费组的时候调用poll()方法的时间间隔。 该条目指定了消费者调用poll()方法的最大时间间隔。 如果在此时间内消费者没有调用poll()方法， 则broker认为消费者失败，触发再平衡， 将分区分配给消费组中其他消费者。
	对每个分区，服务器返回的最大数量。消费者按批次拉取数据。 如果非空分区的第一个记录大于这个值，批处理依然可以返回，

max.partition.fetch.bytes	以保证消费者可以进行下去。 broker接收批的大小由`message.max.bytes`（broker参数）或`max.message.bytes`（主题参数）指定。 `fetch.max.bytes`用于限制消费者单次请求的数据量。
send.buffer.bytes	用于TCP发送数据时使用的缓冲大小（SO_SNDBUF）， -1表示使用OS默认的缓冲区大小。
retry.backoff.ms	在发生失败的时候如果需要重试，则该配置表示客户端 等待多长时间再发起重试。 该时间的存在避免了密集循环。
request.timeout.ms	客户端等待服务端响应的最大时间。如果该时间超时， 则客户端要么重新发起请求，要么如果重试耗尽，请求失败。
reconnect.backoff.ms	重新连接主机的等待时间。避免了重连的密集循环。 该等待时间应用于该客户端到broker的所有连接。
reconnect.backoff.max.ms	重新连接到反复连接失败的broker时要等待的最长时间 （以毫秒为单位）。 如果提供此选项，则对于每个连续的连接失败， 每台主机的退避将成倍增加，直至达到此最大值。 在计算退避增量之后，添加20%的随机抖动以避免连接风暴。
receive.buffer.bytes	TCP连接接收数据的缓存（SO_RCVBUF）。 -1表示使用操作系统的默认值。
partition.assignment.strategy	当使用消费组的时候，分区分配策略的类名。
metrics.sample.window.ms	计算指标样本的时间窗口。
metrics.recording.level	指标的最高记录级别。
metrics.num.samples	用于计算指标而维护的样本数量
interceptor.classes	拦截器类的列表。默认没有拦截器 拦截器是消费者的拦截器，该拦截器需要实现`org.apache.kafka.clients.consumer.ConsumerInterceptor`接口。 拦截器可用于对消费者接收到的消息进行拦截处理。

2.2.3 消费组管理

一、消费者组 (Consumer Group)

1 什么是消费者组

consumer group是kafka提供的可扩展且具有容错性的消费者机制。

三个特性：

1. 消费组有一个或多个消费者，消费者可以是一个进程，也可以是一个线程
2. group.id是一个字符串，唯一标识一个消费组
3. 消费组订阅的主题每个分区只能分配给消费组一个消费者。

2 消费者位移(consumer position)

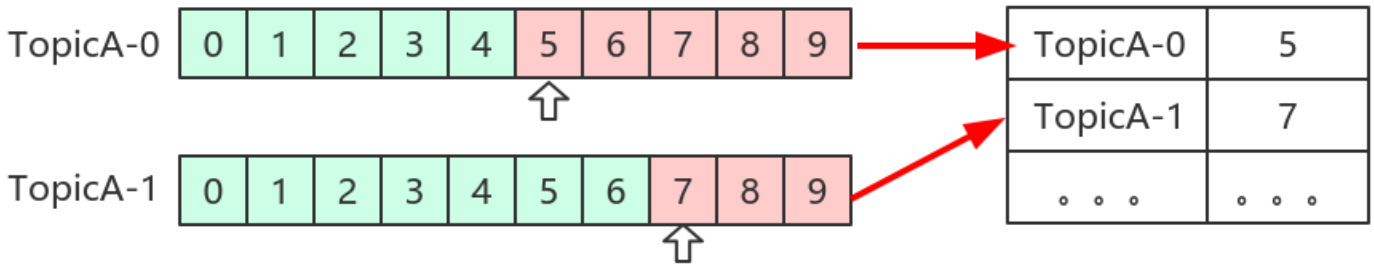
消费者在消费的过程中记录已消费的数据，即消费位移（offset）信息。

每个消费组保存自己的位移信息，那么只需要简单的一个整数表示位置就够了；同时可以引入checkpoint机制定期持久化。

3 位移管理(offset management)

3.1 自动VS手动

Kafka默认定期自动提交位移(`enable.auto.commit = true`)，也手动提交位移。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：



3.2 位移提交

位移是提交到Kafka中的`__consumer_offsets`主题。`__consumer_offsets`中的消息保存了每个消费组某一时刻提交的offset信息。

```
[root@node1 __consumer_offsets-0]# kafka-console-consumer.sh --topic __consumer_offsets --bootstrap-server node1:9092 --formatter "kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter" --consumer.config /opt/kafka_2.12-1.0.2/config/consumer.properties --from-beginning | head
```

```
[root@node1 __consumer_offsets-0]# kafka-console-consumer.sh --topic __consumer_offsets --bootstrap-server node1:9092 --formatter "ka
sMessageFormatter" --consumer.config /opt/kafka_2.12-1.0.2/config/consumer.properties --from-beginning | head
[test-consumer-group,__consumer_offsets,22]::[OffsetMetadata[12,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,30]::[OffsetMetadata[0,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,8]::[OffsetMetadata[7,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,21]::[OffsetMetadata[7,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,4]::[OffsetMetadata[5,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,27]::[OffsetMetadata[4,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,7]::[OffsetMetadata[59,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,9]::[OffsetMetadata[266,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,46]::[OffsetMetadata[0,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
[test-consumer-group,__consumer_offsets,25]::[OffsetMetadata[8,NO_METADATA],CommitTime 1596597941558,ExpirationTime 1596684341558]
Unable to write to standard out, closing consumer.
```

上图中，标出来的，表示消费组为`test-consumer-group`，消费的主题为`__consumer_offsets`，消费的分区是4，偏移量为5。

`__consumers_offsets`主题配置了compact策略，使得它总是能够保存最新的位移信息，既控制了该topic总体的日志容量，也能实现保存最新offset的目的。

4 再谈再均衡

4.1 什么是再均衡?

再均衡（Rebalance）本质上是一种协议，规定了一个消费组中所有消费者如何达成一致来分配订阅主题的每个分区。

比如某个消费组有20个消费组，订阅了一个具有100个分区的主题。正常情况下，Kafka平均会为每个消费者分配5个分区。这个分配的过程就叫再均衡。

4.2 什么时候再均衡?

再均衡的触发条件：

1. 组成员发生变更(新消费者加入消费组组、已有消费者主动离开或崩溃了)
2. 订阅主题数发生变更。如果正则表达式进行订阅，则新建匹配正则表达式的主题触发再均衡。
3. 订阅主题的分区数发生变更

4.3 如何进行组内分区分配?

三种分配策略：RangeAssignor和RoundRobinAssignor以及StickyAssignor。后面讲。

4.4 谁来执行再均衡和消费组管理?

Kafka提供了一个角色：Group Coordinator来执行对于消费组的管理。

Group Coordinator——每个消费组分配一个消费组协调器用于组管理和位移管理。当消费组的第一个消费者启动的时候，它会去和Kafka Broker确定谁是它们组的组协调器。之后该消费组内所有消费者和该组协调器协调通信。

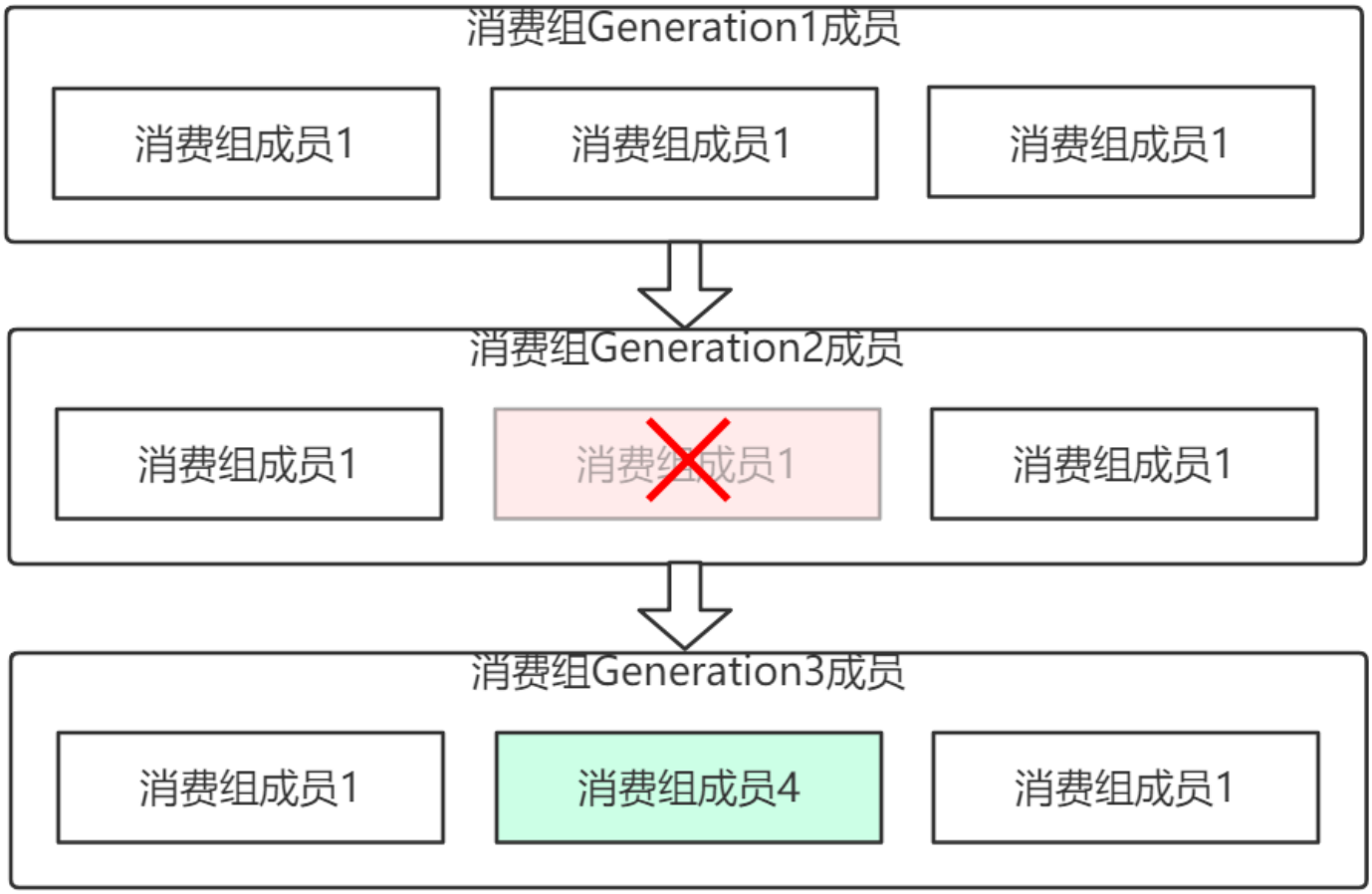
4.5 如何确定coordinator?

两步：

1. 确定消费组位移信息写入`__consumers_offsets`的哪个分区。具体计算公式：
    - __consumers_offsets partition# = Math.abs(groupId.hashCode() % groupMetadataTopicPartitionCount) 注意：groupMetadataTopicPartitionCount由`offsets.topic.num.partitions`指定，默认是50个分区。
2. 该分区leader所在的broker就是组协调器。

4.6 Rebalance Generation

它表示Rebalance之后主题分区到消费组中消费者映射关系的一个版本，主要是用于保护消费组，隔离无效偏移量提交的。如上一个版本的消费者无法提交位移到新版本的消费组中，因为映射关系变了，你消费的或许已经不是原来的那个分区了。每次group进行Rebalance之后，Generation号都会加1，表示消费组和分区的映射关系到了一个新版本，如下图所示： Generation 1时group有3个成员，随后成员2退出组，消费组协调器触发Rebalance，消费组进入Generation 2，之后成员4加入，再次触发Rebalance，消费组进入Generation 3.



4.7 协议(protocol)

kafka提供了5个协议来处理与消费组协调相关的问题：

- Heartbeat请求：consumer需要定期给组协调器发送心跳来表明自己还活着
- LeaveGroup请求：主动告诉组协调器我要离开消费组
- SyncGroup请求：消费组Leader把分配方案告诉组内所有成员
- JoinGroup请求：成员请求加入组
- DescribeGroup请求：显示组的所有信息，包括成员信息，协议名称，分配方案，订阅信息等。通常该请求是给管理员使用

组协调器在再均衡的时候主要用到了前面4种请求。

4.8 liveness

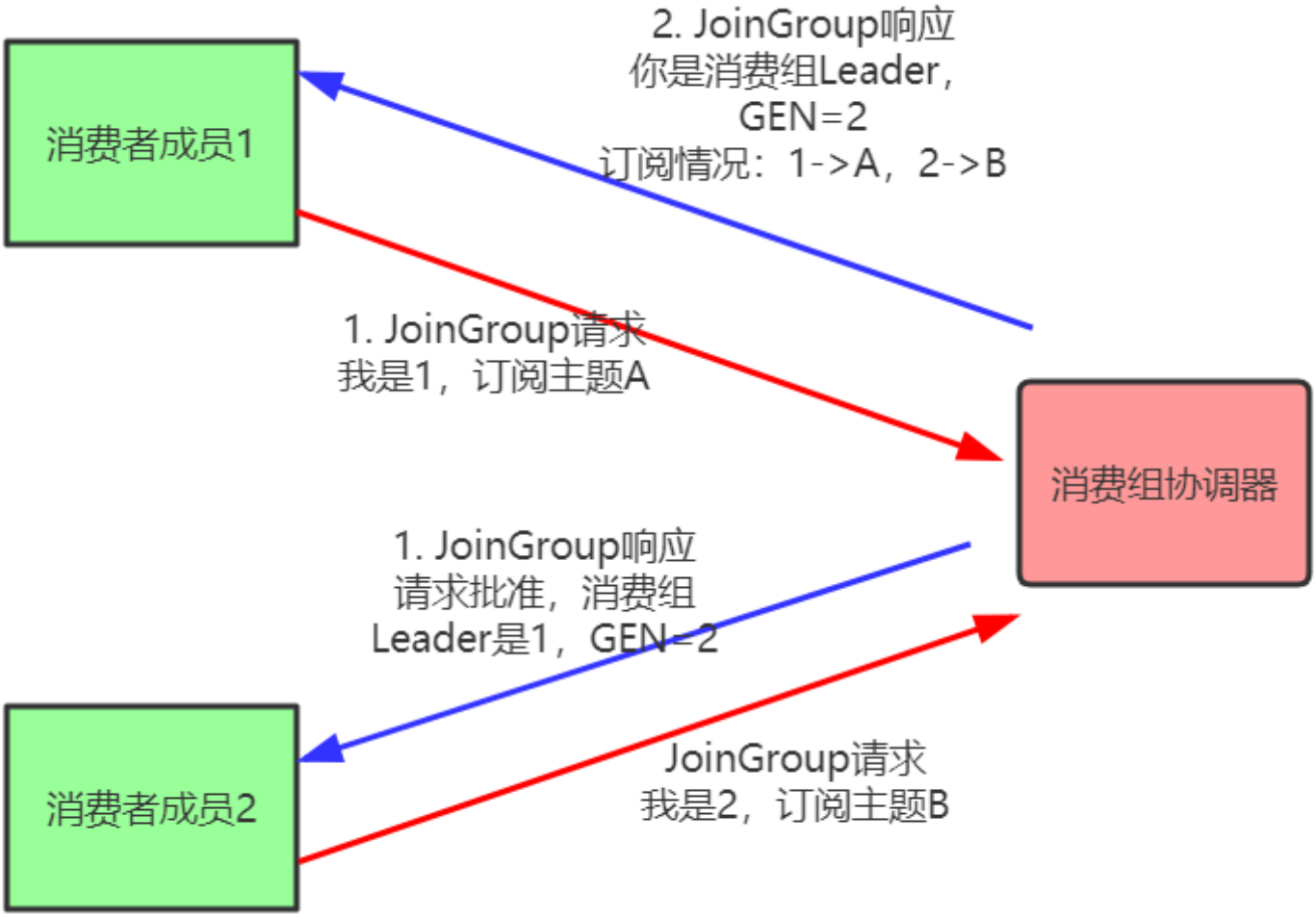
消费者如何向消费组协调器证明自己还活着？ 通过定时向消费组协调器发送Heartbeat请求。如果超过了设定的超时时间，那么协调器认为该消费者已经挂了。一旦协调器认为某个消费者挂了，那么它就会开启新一轮再均衡，并且在当前其他消费者的心跳**响应中**添加“REBALANCE_IN_PROGRESS”，告诉其他消费者：重新分配分区。

4.9 再均衡过程

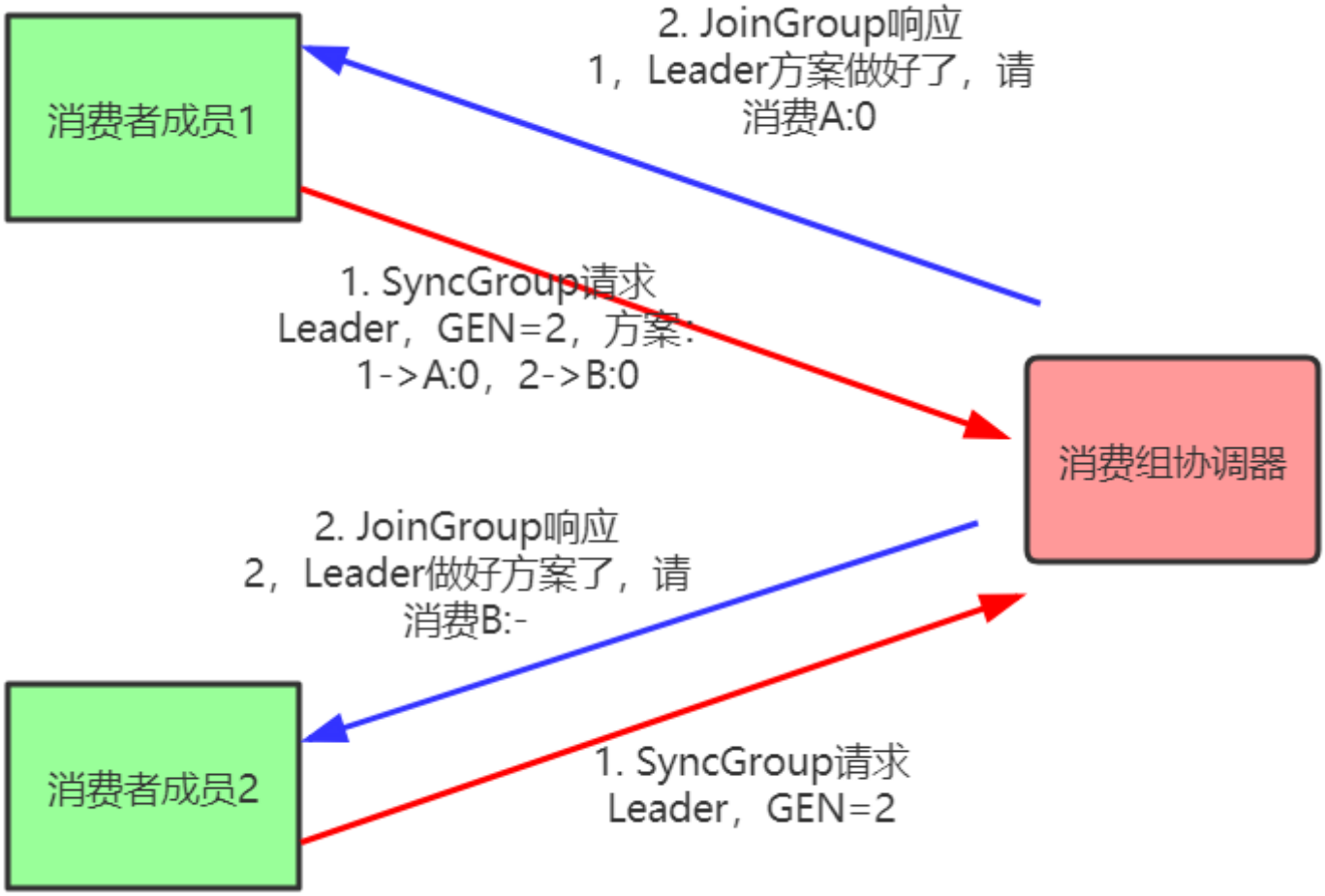
再均衡分为2步：Join和Sync

1. Join， 加入组。所有成员都向消费组协调器发送JoinGroup请求，请求加入消费组。一旦所有成员都发送了JoinGroup请求，协调i器从中选择一个消费者担任Leader的角色，并把组成员信息以及订阅信息发给Leader。
2. Sync，Leader开始分配消费方案，即哪个消费者负责消费哪些主题的哪些分区。一旦完成分配，Leader会将这个方案封装进SyncGroup请求中发给消费组协调器，非Leader也会发SyncGroup请求，只是内容为空。

消费组协调器接收到分配方案之后会把方案塞进SyncGroup的response中发给各个消费者。



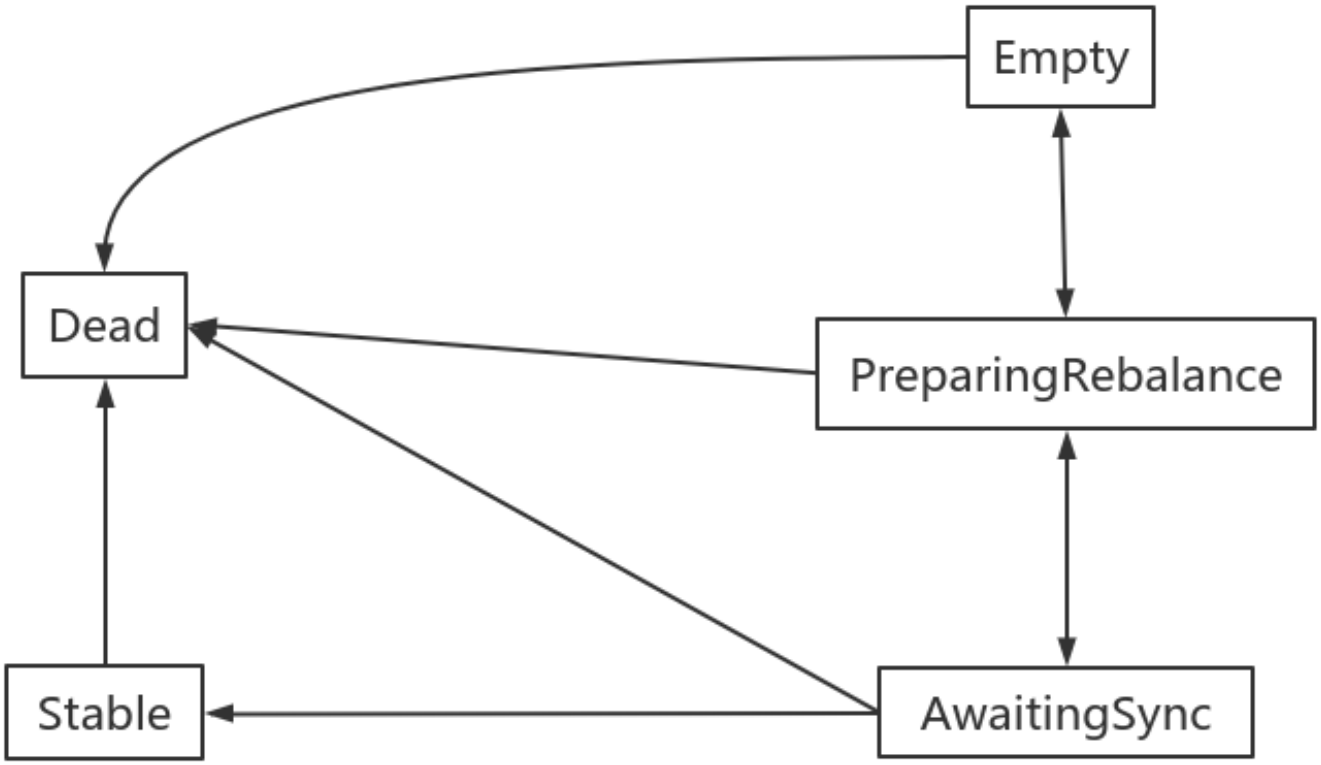
注意：在协调器收集到所有成员请求前，它会把已收到请求放入一个叫purgatory(炼狱)的地方。然后是分发分配方案的过程，即SyncGroup请求：



注意：消费组的分区分配方案在客户端执行。Kafka交给客户端可以有更好的灵活性。Kafka默认提供三种分配策略：range和round-robin和sticky。可以通过消费者的参数：`partition.assignment.strategy`来实现自己分配策略。

4.10 消费组状态机

消费组组协调器根据状态机对消费组做不同的处理：



说明：

1. Dead：组内已经没有任何成员的最终状态，组的元数据也已经被组协调器移除了。这种状态响应各种请求都是一个response： UNKNOWN_MEMBER_ID
2. Empty：组内无成员，但是位移信息还没有过期。这种状态只能响应JoinGroup请求
3. PreparingRebalance：组准备开启新的rebalance，等待成员加入
4. AwaitingSync：正在等待leader consumer将分配方案传给各个成员
5. Stable：再均衡完成，可以开始消费。

2.3 主题

2.3.1 管理

使用kafka-topics.sh脚本：

选项	说明

	为创建的或修改的主题指定配置信息。支持下述配置条目：
	`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`

--config <String: name=value>	`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`

--create	创建一个新主题
--delete	删除一个主题
--delete-config <String: name>	删除现有主题的一个主题配置条目。这些条目就是在 `--config` 中给出的配置条目。
--alter	更改主题的分区数量，副本分配和/或配置条目。
--describe	列出给定主题的细节。
--disable-rack-aware	禁用副本分配的机架感知。
--force	抑制控制台提示信息
--help	打印帮助信息
--if-exists	如果指定了该选项，则在修改或删除主题的时候，只有主题存在才可以执行。
--if-not-exists	在创建主题的时候，如果指定了该选项，则只有主题不存在的时候才可以执行命令。
--list	列出所有可用的主题。
--partitions <Integer: # of partitions>	要创建或修改主题的分区数。
--replica-assignment <String:broker_id_for_part1_replica1 :broker_id_for_part1_replica2 ,broker_id_for_part2_replica1 :broker_id_for_part2_replica2 , ...>	当创建或修改主题的时候手动指定partition-to-broker的分配关系。
--replication-factor <Integer:replication factor>	要创建的主题分区副本数。1表示只有一个副本，也就是Leader副本。
--topic <String: topic>	要创建、修改或描述的主题名称。除了创建，修改和描述在这里还可以使用正则表达式。
--topics-with-overrides	if set when describing topics, only show topics that have overridden configs
--unavailable-partitions	if set when describing topics, only show partitions whose leader is not available
--under-replicated-partitions	if set when describing topics, only show under replicated partitions
--zookeeper <String: urls>	必需的参数：连接zookeeper的字符串，逗号分隔的多个host:port列表。多个URL可以故障转移。

主题中可以使用的参数定义：

属性	默认值	服务器默认属性	说明
cleanup.policy	delete	log.cleanup.policy	要么是”delete“要么是”compact“； 这个字符串指明了针对旧日志部分的利用方式；默认方式（"delete"）将会丢弃旧的部分当他们的回收时间或者尺寸限制到达时。”compact“将会进行日志压缩
compression.type	none		producer用于压缩数据的压缩类型。默认是无压缩。正确的选项值是none、gzip、snappy。压缩最好用于批量处理，批量处理消息越多，压缩性能越好。
delete.retention.ms	86400000 (24 hours)	log.cleaner.delete.retention.ms	对于压缩日志保留的最长时间，也是客户端消费消息的最长时间，通log.retention.minutes的区别在于一个控制未压缩数据，一个控制压缩后的数据。此项配置可以在topic创建时的置顶参数覆盖
flush.ms	None	log.flush.interval.ms	此项配置用来置顶强制进行fsync日志到磁盘的时间间隔；例如，如果设置为1000，那么每1000ms就需要进行一次fsync。一般不建议使用这个选项
flush.messages	None	log.flush.interval.messages	此项配置指定时间间隔：强制进行fsync日志。例如，如果这个选项设置为1，那么每条消息之后都需要进行fsync，如果设置为5，则每5条消息就需要进行一次fsync。一般来说，建议你不要设置这个值。此参数的设置,需要在"数据可靠性"与"性能"之间做必要的权衡.如果此值过大,将会导致每次"fsync"的时间较长(IO阻塞),如果此值过小,将会导致"fsync"的次数较多,这也意味着整体的client请求有一定的延迟.物理server故障,将会导致没有fsync的消息丢失.
index.interval.bytes	4096	log.index.interval.bytes	默认设置保证了我们每4096个字节就对消息添加一个索引，更多的索引使得阅读的消息更加靠近，但是索引规模却会由此增大；一般不需要改变这个选项
max.message.bytes	1000000	max.message.bytes	kafka追加消息的最大尺寸。注意如果你增大这个尺寸，你也必须增大你consumer的fetch 尺寸，这样consumer才能fetch到这些最大尺寸的消息。
min.cleanable.dirty.ratio	0.5	min.cleanable.dirty.ratio	此项配置控制log压缩器试图进行清除日志的频率。默认情况下，将避免清除压缩率超过50%的日志。这个比率避免了最大的空间浪费
min.insync.replicas	1	min.insync.replicas	当producer设置request.required.acks为-1时，min.insync.replicas指定replicas的最小数目（必须确认每一个repica的写数据都是成功的），如果这个数目没有达到，producer会产生异常。
retention.bytes	None	log.retention.bytes	如果使用“delete”的retention 策略，这项配置就是指在删除日志之前，日志所能达到的最大尺寸。默认情况下，没有尺寸限制而只有时间限制
retention.ms	7 days	log.retention.minutes	如果使用“delete”的retention策略，这项配置就是指删除日志前日志保存的时间。
segment.bytes	1GB	log.segment.bytes	kafka中log日志是分成一块块存储的，此配置是指log日志划分成块的大小
segment.index.bytes	10MB	log.index.size.max.bytes	此配置是有关offsets和文件位置之间映射的索引文件的大小；一般不需要修改这个配置
segment.jitter.ms	0	log.roll.jitter.{ms,hours}	The maximum jitter to subtract from logRollTimeMillis.
segment.ms	7 days	log.roll.hours	即使log的分块文件没有达到需要删除、压缩的大小，一旦log 的时间达到这个上限，就会强制新建一个log分块文件
unclean.leader.election.enable	true		指明了是否能够使不在ISR中replicas设置用来作为leader

2.3.1.1 创建主题

```
kafka-topics.sh --zookeeper localhost:2181/myKafka --create --topic topic_x --partitions 1 --replication-factor 1
kafka-topics.sh --zookeeper localhost:2181/myKafka --create --topic topic_test_02 --partitions 3 --replication-factor 1 --config max.message.bytes=1048576 --config segment.bytes=10485760
```

2.3.1.2 查看主题

```
kafka-topics.sh --zookeeper localhost:2181/myKafka --list
kafka-topics.sh --zookeeper localhost:2181/myKafka --describe --topic topic_x
kafka-topics.sh --zookeeper localhost:2181/myKafka --topics-with-overrides --describe
```

2.3.1.3 修改主题

```
kafka-topics.sh --zookeeper localhost:2181/myKafka --create --topic topic_test_01 --partitions 2 --replication-factor 1
kafka-topics.sh --zookeeper localhost:2181/myKafka --alter --topic topic_test_01 --config max.message.bytes=1048576
kafka-topics.sh --zookeeper localhost:2181/myKafka --describe --topic topic_test_01
kafka-topics.sh --zookeeper localhost:2181/myKafka --alter --topic topic_test_01 --config segment.bytes=10485760
kafka-topics.sh --zookeeper localhost:2181/myKafka --alter --delete-config max.message.bytes --topic topic_test_01
```

2.3.1.4 删除主题

```
kafka-topics.sh --zookeeper localhost:2181/myKafka --delete --topic topic_x
```

```
[root@node1 kafka-logs]# kafka-topics.sh --zookeeper localhost:2181/myKafka --list
__consumer_offsets
topic-spring-02
tp_demo_01
tp_test_01
[root@node1 kafka-logs]# kafka-topics.sh --zookeeper localhost:2181/myKafka --delete --topic topic-spring-02
Topic topic-spring-02 is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

给主题添加删除的标记：

```
topic-spring-02-0.fcbed45395bd4e81be37d4af09d6982a-delete
```

要过一段时间删除。

2.3.2 增加分区

通过命令行工具操作，主题的分区只能增加，不能减少。否则报错：

```
ERROR org.apache.kafka.common.errors.InvalidPartitionsException: The number of partitions for a topic can only be increased. Topic myTop1 currently has 2 partitions, 1 would not be an increase.
```

通过--alter修改主题的分区数，增加分区。

```
kafka-topics.sh --zookeeper localhost/myKafka --alter --topic myTop1 --partitions 2
```

2.3.3 分区副本的分配-了解

副本分配的三个目标：

1. 均衡地将副本分散于各个broker上
2. 对于某个broker上分配的分区，它的其他副本在其他broker上
3. 如果所有的broker都有机架信息，尽量将分区的各个副本分配到不同机架上的broker。

在不考虑机架信息的情况下：

1. 第一个副本分区通过轮询的方式挑选一个broker，进行分配。该轮询从broker列表的随机位置进行轮询。
2. 其余副本通过增加偏移进行分配。

分配案例：

broker-0	broker-1	broker-2	broker-3	broker-4	
p0	p1	p2	p3	p4	(1st replica)
p5	p6	p7	p8	p9	(1st replica)
p4	p0	p1	p2	p3	(2nd replica)
p8	p9	p5	p6	p7	(2nd replica)
p3	p4	p0	p1	p2	(3nd replica)
p7	p8	p9	p5	p6	(3nd replica)

```
// 对所有分区进行分配，起始
for (_ <- 0 until nPartitions) {
  // 到broker列表的边界了，则下一个副本位移加1
  if (currentPartitionId > 0 && (currentPartitionId % brokerArray.length == 0))
    nextReplicaShift += 1
  // 计算起始分区第一个副本放置的位置，也就是Leader副本分区的位置
  val firstReplicaIndex = (currentPartitionId + startIndex) % brokerArray.length
  val replicaBuffer = mutable.ArrayBuffer(brokerArray(firstReplicaIndex))
  // 对分区所有副本进行分配
  for (j <- 0 until replicationFactor - 1)
    replicaBuffer += brokerArray(replicaIndex(firstReplicaIndex, nextReplicaShift, j, brokerArray.length))
  ret.put(currentPartitionId, replicaBuffer)
  currentPartitionId += 1
}
```

考虑到机架信息，首先为每个机架创建一个broker列表。如：
三个机架（rack1，rack2，rack3），六个broker（0，1，2，3，4，5）

brokerID -> rack

- 0 -> "rack1", 1 -> "rack3", 2 -> "rack3", 3 -> "rack2", 4 -> "rack2", 5 -> "rack1"

rack1：0，5

rack2：3，4

rack3：1，2

这broker列表为rack1的0，rack2的3，rack3的1，rack1的5，rack2的4，rack3的2

即：0, 3, 1, 5, 4, 2

通过简单的轮询将分区分配给不同机架上的broker：

r1		r2		r3		r1		r2		r3		r1		r2		r3	
0	3	1	5	4	2	0	3	1	5	4	2	0	3	1	5	4	2
0	0	0															
	1	1	1														
		2	2	2													
			3	3	3												
				4	4	4											
					5	5	5										
				6	6	6											
					7	7	7										

每个分区副本在分配的时候在上一个分区第一个副本开始分配的位置右移一位。

六个broker，六个分区，正好最后一个分区的第一个副本分配的位置是该broker列表的最后一个。

如果有更多的分区需要分配，则算法开始对follower副本进行移位分配。

这主要是为了避免每次都得到相同的分配序列。

此时，如果有一个分区等待分配（分区6），这按照如下方式分配：

6 -> 0,4,2 (而不是像分区0那样重复0,3,1)

跟机架相关的副本分配中，永远在机架相关的broker列表中轮询地分配第一个副本。
其余的副本，倾向于机架上没有副本的broker进行副本分配，除非每个机架有一个副本。
然后其他的副本又通过轮询的方式分配给broker。

结果是，如果副本的个数大于等于机架数，保证每个机架最少有一个副本。
否则每个机架最多保有一个副本。
如果副本的个数和机架的个数相同，并且每个机架包含相同个数的broker，可以保证副本在机架和broker之间均匀分布。

```
[zk: localhost:2181(CONNECTED) 0] get /myKafka/brokers/topics/tp_eagle_01/partitions/0/state
{"controller_epoch":9,"leader":1,"version":1,"leader_epoch":3,"isr":[1,2]}
cZxid = 0x200000130
```

上图，tp_eagle_01主题的分区0分配信息：leader分区在broker1上，同步副本分区是1和2，也就是在broker1和broker2上的两个副本分区是同步副本分区，其中一个是leader分区。

2.3.4 必要参数配置

kafka-topics.sh --config xx=xx --config yy=yy

配置给主题的参数。

属性	默认值	服务器默认属性	说明
cleanup.policy	delete	log.cleanup.policy	要么是”delete“要么是”compact“； 这个字符串指明了针对旧日志部分的利用方式；默认方式（"delete"）将会丢弃旧的部分当他们的回收时间或者尺寸限制到达时。”compact“将会进行日志压缩
compression.type	none		producer用于压缩数据的压缩类型。默认是无压缩。正确的选项值是none、gzip、snappy、lz4。压缩最好用于批量处理，批量处理消息越多，压缩性能越好。
max.message.bytes	1000000	max.message.bytes	kafka追加消息的最大字节数。注意如果你增大这个字节数，也必须增大consumer的fetch字节数，这样consumer才能fetch到这些最大字节数的消息。
min.cleanable.dirty.ratio	0.5	min.cleanable.dirty.ratio	此项配置控制log压缩器试图进行清除日志的频率。默认情况下，将避免清除压缩率超过50%的日志。这个比率避免了最大的空间浪费
min.insync.replicas	1	min.insync.replicas	当producer设置request.required.acks为-1时，min.insync.replicas指定replicas的最小数目（必须确认每一个repica的写数据都是成功的），如果这个数目没有达到，producer会产生异常。
retention.bytes	None	log.retention.bytes	如果使用“delete”的retention 策略，这项配置就是指在删除日志之前，日志所能达到的最大尺寸。默认情况下，没有尺寸限制而只有时间限制
retention.ms	7 days	log.retention.minutes	如果使用“delete”的retention策略，这项配置就是指删除日志前日志保存的时间。
segment.bytes	1GB	log.segment.bytes	kafka中log日志是分成一块块存储的，此配置是指log日志划分成块的大小
segment.index.bytes	10MB	log.index.size.max.bytes	此配置是有关offsets和文件位置之间映射的索引文件的大小；一般不需要修改这个配置
segment.jitter.ms	0	log.roll.jitter.{ms,hours}	The maximum jitter to subtract from logRollTimeMillis.
segment.ms	7 days	log.roll.hours	即使log的分块文件没有达到需要删除、压缩的大小，一旦log 的时间达到这个上限，就会强制新建一个log分块文件
unclean.leader.election.enable	true		指明了是否能够使不在ISR中replicas设置用来作为leader

2.3.5 KafkaAdminClient应用

说明

除了使用Kafka的bin目录下的脚本工具来管理Kafka，还可以使用管理Kafka的API将某些管理查看的功能集成到系统中。在Kafka0.11.0.0版本之前，可以通过kafka-core包（Kafka的服务端，采用Scala编写）中的AdminClient和AdminUtils来实现部分的集群管理操作。Kafka0.11.0.0之后，又多了一个AdminClient，在kafka-client包下，一个抽象类，具体的实现是org.apache.kafka.clients.admin.KafkaAdminClient。

功能与原理介绍

Kafka官网：The AdminClient API supports managing and inspecting topics, brokers, acls, and other Kafka objects。

KafkaAdminClient包含了一下几种功能（以Kafka1.0.2版本为准）：

1. 创建主题：
    1. createTopics(final Collection<NewTopic> newTopics, final CreateTopicsOptions options)
2. 删除主题：
    1. deleteTopics(final Collection<String> topicNames, DeleteTopicsOptions options)
3. 列出所有主题：
    1. listTopics(final ListTopicsOptions options)
4. 查询主题：
    1. describeTopics(final Collection<String> topicNames, DescribeTopicsOptions options)
5. 查询集群信息：
    1. describeCluster(DescribeClusterOptions options)
6. 查询配置信息：
    1. describeConfigs(Collection<ConfigResource> configResources, final DescribeConfigsOptions options)
7. 修改配置信息：
    1. alterConfigs(Map<ConfigResource, Config> configs, final AlterConfigsOptions options)
8. 修改副本的日志目录：
    1. alterReplicaLogDirs(Map<TopicPartitionReplica, String> replicaAssignment, final AlterReplicaLogDirsOptions options)
9. 查询节点的日志目录信息：
    1. describeLogDirs(Collection<Integer> brokers, DescribeLogDirsOptions options)
10. 查询副本的日志目录信息：
    1. describeReplicaLogDirs(Collection<TopicPartitionReplica> replicas, DescribeReplicaLogDirsOptions options)
11. 增加分区：
    1. createPartitions(Map<String, NewPartitions> newPartitions, final CreatePartitionsOptions options)

其内部原理是使用Kafka自定义的一套二进制协议来实现，详细可以参见Kafka协议。

用到的参数：

属性	说明	重要性
bootstrap.servers	向Kafka集群建立初始连接用到的host/port列表。 客户端会使用这里列出的所有服务器进行集群其他服务器的发现，而不管是否指定了哪个服务器用作引导。 这个列表仅影响用来发现集群所有服务器的初始主机。 字符串形式：host1:port1,host2:port2,... 由于这组服务器仅用于建立初始链接，然后发现集群中的所有服务器，因此没有必要将集群中的所有地址写在这里。 一般最好两台，以防其中一台宕掉。	high
client.id	生产者发送请求的时候传递给broker的id字符串。 用于在broker的请求日志中追踪什么应用发送了什么消息。 一般该id是跟业务有关的字符串。	medium
connections.max.idle.ms	当连接空闲时间达到这个值，就关闭连接。long型数据，默认：300000	medium
receive.buffer.bytes	TCP接收缓存（SO_RCVBUF），如果设置为-1，则使用操作系统默认的值。 int类型值，默认65536，可选值：[**-1,...**]	medium
request.timeout.ms	客户端等待服务端响应的最大时间。如果该时间超时，则客户端要么重新发起请求，要么如果重试耗尽，请求失败。int类型值，默认：120000	medium
security.protocol	跟broker通信的协议：PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL. string类型值，默认：PLAINTEXT	medium
send.buffer.bytes	用于TCP发送数据时使用的缓冲大小（SO_SNDBUF），-1表示使用OS默认的缓冲区大小。 int类型值，默认值：131072	medium
reconnect.backoff.max.ms	对于每个连续的连接失败，每台主机的退避将成倍增加，直至达到此最大值。在计算退避增量之后，添加20%的随机抖动以避免连接风暴。 long型值，默认1000，可选值：[**0,...**]	low
reconnect.backoff.ms	重新连接主机的等待时间。避免了重连的密集循环。该等待时间应用于该客户端到broker的所有连接。 long型值，默认：50	low
retries	The maximum number of times to retry a call before failing it.重试的次数，达到此值，失败。 int类型值，默认5。	low
retry.backoff.ms	在发生失败的时候如果需要重试，则该配置表示客户端等待多长时间再发起重试。 该时间的存在避免了密集循环。 long型值，默认值：100。	low

主要操作步骤：

客户端根据方法的调用创建相应的协议请求，比如创建Topic的createTopics方法，其内部就是发送CreateTopicRequest请求。

客户端发送请求至Kafka Broker。

Kafka Broker处理相应的请求并回执，比如与CreateTopicRequest对应的是CreateTopicResponse。客户端接收相应的回执并进行解析处理。

和协议有关的请求和回执的类基本都在org.apache.kafka.common.requests包中，AbstractRequest和AbstractResponse是这些请求和响应类的两个父类。

综上，如果要自定义实现一个功能，只需要三个步骤：

1. 自定义XXXOptions;
2. 自定义XXXResult返回值；
3. 自定义Call，然后挑选合适的XXXRequest和XXXResponse来实现Call类中的3个抽象方法。

```java
package com.lagou.kafka.demo;

import org.apache.kafka.clients.admin.*;
import org.apache.kafka.common.KafkaFuture;
import org.apache.kafka.common.Node;
import org.apache.kafka.common.TopicPartitionInfo;
import org.apache.kafka.common.config.ConfigResource;
import org.apache.kafka.common.requests.DescribeLogDirsResponse;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.util.*;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

public class MyAdminClient {

    private KafkaAdminClient client;

    @Before
    public void before() {
        Map<String, Object> conf = new HashMap<>();
        conf.put("bootstrap.servers", "node1:9092");
        conf.put("client.id", "adminclient-1");

        client = (KafkaAdminClient) KafkaAdminClient.create(conf);
    }

    @After
    public void after() {
        client.close();
    }
```

```
37
38      @Test
39      public void testListTopics1() throws ExecutionException, InterruptedException {
40
41          ListTopicsResult listTopicsResult = client.listTopics();
42  //          KafkaFuture<Collection<TopicListing>> listings =
    listTopicsResult.listings();
43  //          Collection<TopicListing> topicListings = listings.get();
44  //
45  //          topicListings.forEach(new Consumer<TopicListing>() {
46  //              @Override
47  //              public void accept(TopicListing topicListing) {
48  //                  boolean internal = topicListing.isInternal();
49  //                  String name = topicListing.name();
50  //                  String s = topicListing.toString();
51  //                  System.out.println(s + "\t" + name + "\t" + internal);
52  //              }
53  //          });
54
55  //          KafkaFuture<Set<String>> names = listTopicsResult.names();
56  //          Set<String> strings = names.get();
57  //
58  //          strings.forEach(name -> {
59  //              System.out.println(name);
60  //          });
61
62  //          KafkaFuture<Map<String, TopicListing>> mapKafkaFuture =
    listTopicsResult.namesToListings();
63  //          Map<String, TopicListing> stringTopicListingMap = mapKafkaFuture.get();
64  //
65  //          stringTopicListingMap.forEach((k, v) -> {
66  //              System.out.println(k + "\t" + v);
67  //          });
68
69          ListTopicsOptions options = new ListTopicsOptions();
70          options.listInternal(false);
71          options.timeoutMs(500);
72          ListTopicsResult listTopicsResult1 = client.listTopics(options);
73
74          Map<String, TopicListing> stringTopicListingMap =
    listTopicsResult1.namesToListings().get();
75          stringTopicListingMap.forEach((k, v) -> {
76              System.out.println(k + "\t" + v);
77          });
78
79          // 关闭管理客户端
80          client.close();
81      }
82
83      @Test
84      public void testCreateTopic() throws ExecutionException, InterruptedException {
```

```
85
86          Map<String, String> configs = new HashMap<>();
87          configs.put("max.message.bytes", "1048576");
88          configs.put("segment.bytes", "1048576000");
89
90          NewTopic newTopic = new NewTopic("adm_tp_01", 2, (short) 1);
91          newTopic.configs(configs);
92
93          CreateTopicsResult topics =
client.createTopics(Collections.singleton(newTopic));
94          KafkaFuture<Void> all = topics.all();
95          Void aVoid = all.get();
96          System.out.println(aVoid);
97      }
98
99      @Test
100     public void testDeleteTopic() throws ExecutionException, InterruptedException {
101
102         DeleteTopicsOptions options = new DeleteTopicsOptions();
103         options.timeoutMs(500);
104
105         DeleteTopicsResult deleteResult =
client.deleteTopics(Collections.singleton("adm_tp_01"), options);
106         deleteResult.all().get();
107     }
108
109     @Test
110     public void testAlterTopic() throws ExecutionException, InterruptedException {
111         NewPartitions newPartitions = NewPartitions.increaseTo(5);
112         Map<String, NewPartitions> newPartitionsMap = new HashMap<>();
113         newPartitionsMap.put("adm_tp_01", newPartitions);
114
115         CreatePartitionsOptions option = new CreatePartitionsOptions();
116         // Set to true if the request should be validated without creating new
partitions.
117         // 如果只是验证，而不创建分区，则设置为true
118 //        option.validateOnly(true);
119
120         CreatePartitionsResult partitionsResult =
client.createPartitions(newPartitionsMap, option);
121         Void aVoid = partitionsResult.all().get();
122
123     }
124
125     @Test
126     public void testDescribeTopics() throws ExecutionException, InterruptedException
{
127
128         DescribeTopicsOptions options = new DescribeTopicsOptions();
129         options.timeoutMs(3000);
130
```

```
        DescribeTopicsResult topicsResult = client.describeTopics(Collections.singleton("adm_tp_01"), options);

        Map<String, TopicDescription> stringTopicDescriptionMap = topicsResult.all().get();
        stringTopicDescriptionMap.forEach((k, v) -> {
            System.out.println(k + "\t" + v);
            System.out.println("=======================================");
            System.out.println(k);
            boolean internal = v.isInternal();
            String name = v.name();
            List<TopicPartitionInfo> partitions = v.partitions();
            String partitionStr = Arrays.toString(partitions.toArray());

            System.out.println("内部的? " + internal);
            System.out.println("topic name = " + name);
            System.out.println("分区: " + partitionStr);

            partitions.forEach(partition -> {
                System.out.println(partition);
            });

        });

    }

    @Test
    public void testDescribeCluster() throws ExecutionException, InterruptedException {
        DescribeClusterResult describeClusterResult = client.describeCluster();
        KafkaFuture<String> stringKafkaFuture = describeClusterResult.clusterId();
        String s = stringKafkaFuture.get();

        System.out.println("cluster name = " + s);

        KafkaFuture<Node> controller = describeClusterResult.controller();
        Node node = controller.get();

        System.out.println("集群控制器: " + node);

        Collection<Node> nodes = describeClusterResult.nodes().get();

        nodes.forEach(node1 -> {
            System.out.println(node1);
        });
    }

    @Test
    public void testDescribeConfigs() throws ExecutionException, InterruptedException, TimeoutException {

```

```
        ConfigResource configResource = new
ConfigResource(ConfigResource.Type.BROKER, "0");

        DescribeConfigsResult describeConfigsResult
                = client.describeConfigs(Collections.singleton(configResource));

        Map<ConfigResource, Config> configMap = describeConfigsResult.all().get(15,
TimeUnit.SECONDS);

        configMap.forEach(new BiConsumer<ConfigResource, Config>() {
            @Override
            public void accept(ConfigResource configResource, Config config) {
                ConfigResource.Type type = configResource.type();
                String name = configResource.name();

                System.out.println("资源名称: " + name);

                Collection<ConfigEntry> entries = config.entries();

                entries.forEach(new Consumer<ConfigEntry>() {
                    @Override
                    public void accept(ConfigEntry configEntry) {
                        boolean aDefault = configEntry.isDefault();
                        boolean readOnly = configEntry.isReadOnly();
                        boolean sensitive = configEntry.isSensitive();
                        String name1 = configEntry.name();
                        String value = configEntry.value();

                        System.out.println("是否默认: " + aDefault + "\t是否只读? " +
readOnly + "\t是否敏感? " + sensitive
                        + "\t" + name1 + " --> " + value);

                    }
                });

                ConfigEntry retries = config.get("retries");
                if (retries != null) {
                    System.out.println(retries.name() + " -->" + retries.value());
                } else {
                    System.out.println("没有这个属性");
                }
            }
        });

    }

    @Test
    public void testAlterConfig() throws ExecutionException, InterruptedException {
        // 这里设置后，原来资源中不冲突的属性也会丢失，直接按照这里的配置设置
        Map<ConfigResource, Config> configMap = new HashMap<>();
```

```
225         ConfigResource resource = new ConfigResource(ConfigResource.Type.TOPIC,
    "adm_tp_01");
226         Config config = new Config(Collections.singleton(new
    ConfigEntry("segment.bytes", "1048576000")));
227         configMap.put(resource, config);
228 
229         AlterConfigsResult alterConfigsResult = client.alterConfigs(configMap);
230         Void aVoid = alterConfigsResult.all().get();
231     }
232 
233     @Test
234     public void testDescribeLogDirs() throws ExecutionException,
    InterruptedException {
235 
236         DescribeLogDirsOptions option = new DescribeLogDirsOptions();
237         option.timeoutMs(1000);
238 
239         DescribeLogDirsResult describeLogDirsResult =
    client.describeLogDirs(Collections.singleton(0), option);
240         Map<Integer, Map<String, DescribeLogDirsResponse.LogDirInfo>> integerMapMap
241                 = describeLogDirsResult.all().get();
242 
243         integerMapMap.forEach(new BiConsumer<Integer, Map<String,
    DescribeLogDirsResponse.LogDirInfo>>() {
244             @Override
245             public void accept(Integer integer, Map<String,
    DescribeLogDirsResponse.LogDirInfo> stringLogDirInfoMap) {
246                 System.out.println("broker.id = " + integer);
247                 stringLogDirInfoMap.forEach(new BiConsumer<String,
    DescribeLogDirsResponse.LogDirInfo>() {
248                     @Override
249                     public void accept(String s, DescribeLogDirsResponse.LogDirInfo
    logDirInfo) {
250                         System.out.println("log.dirs: " + s);
251                         // 查看该broker上的主题/分区/偏移量等信息
252 //                        logDirInfo.replicaInfos.forEach(new
    BiConsumer<TopicPartition, DescribeLogDirsResponse.ReplicaInfo>() {
253 //                            @Override
254 //                            public void accept(TopicPartition topicPartition,
    DescribeLogDirsResponse.ReplicaInfo replicaInfo) {
255 //                                int partition = topicPartition.partition();
256 //                                String topic = topicPartition.topic();
257 //                                boolean isFuture = replicaInfo.isFuture;
258 //                                long offsetLag = replicaInfo.offsetLag;
259 //                                long size = replicaInfo.size;
260 //                                System.out.println("partition:" + partition +
    "\ttopic:" + topic
261 //                                 + "\tisFuture:" + isFuture
262 //                                 + "\toffsetLag:" + offsetLag
263 //                                 + "\tsize:" + size);
264 //                            }
```

```
//                        });
                    }
                });
            }
        });
    }

}
```

2.3.6 偏移量管理

Kafka 1.0.2，__consumer_offsets主题中保存各个消费组的偏移量。

早期由zookeeper管理消费组的偏移量。

查询方法：

通过原生 kafka 提供的工具脚本进行查询。

工具脚本的位置与名称为`bin/kafka-consumer-groups.sh`

首先运行脚本，查看帮助：

参数	说明
`--all-topics`	将所有关联到指定消费组的主题都划归到`reset-offsets`操作范围。
`--bootstrap-server` <String: server to connect to>	**必须**：(基于消费组的新的消费者): 要连接的服务器地址。
`--by-duration` <String: duration>	距离当前时间戳的一个时间段。格式：'PnDTnHnMnS'
`--command-config` <String: command config property file>	指定配置文件，该文件内容传递给Admin Client和消费者。
`--delete`	传值消费组名称，删除整个消费组与所有主题的各个分区偏移量和所有者关系。 如：`--group g1 --group g2`。 传值消费组名称和单个主题，仅删除该消费组到指定主题的分区偏移量和所属关系。 如：`--group g1 --group g2 --topic t1`。 传值一个主题名称，仅删除指定主题与所有消费组分区偏移量以及所属关系。 如：`--topic t1` 注意：消费组的删除仅对基于ZK保存偏移量的消费组有效，并且要小心使用，仅删除不活跃的消费组。

`--describe`	描述给定消费组的偏移量差距（有多少消息还没有消费）。
`--execute`	执行操作。支持的操作：`reset-offsets`。
`--export`	导出操作的结果到CSV文件。支持的操作：`reset-offsets`。
`--from-file` <String: path to CSV file>	重置偏移量到CSV文件中定义的值。
`--group` <String: consumer group>	目标消费组。
`--list`	列出所有消费组。
`--new-consumer`	使用新的消费者实现。这是默认值。随后的发行版中会删除这一操作。

`--reset-offsets`	重置消费组的偏移量。当前一次操作只支持一个消费组，并且该消费组应该是不活跃的。
	有三个操作选项
	1. （默认）plan：要重置哪个偏移量。
	2. execute：执行`reset-offsets`操作。
	3. process：配合`--export`将操作结果导出到CSV格式。
	可以使用如下选项：
	`--to-datetime`
	`--by-period`
	`--to-earliest`
	`--to-latest`
	`--shift-by`
	`--from-file`
	`--to-current`。
	必须选择一个选项使用。
	要定义操作的范围，使用：
	`--all-topics`
	`--topic`。
	必须选择一个，除非使用`--from-file`选项。

`--shift-by` <Long: number-of-offsets>	重置偏移量n个。n可以是正值，也可以是负值。
`--timeout` <Long: timeout (ms)>	对某些操作设置超时时间。 如：对于描述指定消费组信息，指定毫秒值的最大等待时间，以获取正常数据（如刚创建的消费组，或者消费组做了一些更改操作）。默认时间：`5000`。
`--to-current`	重置到当前的偏移量。
`--to-datetime` <String: datetime>	重置偏移量到指定的时间戳。格式：'YYYY-MM-DDTHH:mm:SS.sss'

`--to-earliest`	重置为最早的偏移量
`--to-latest`	重置为最新的偏移量
`--to-offset` <Long: offset>	重置到指定的偏移量。
`--topic` <String: topic>	指定哪个主题的消费组需要删除，或者指定哪个主题的消费组需要包含到`reset-offsets`操作中。对于`reset-offsets`操作，还可以指定分区：`topic1:0,1,2`。其中0，1，2表示要包含到操作中的分区号。重置偏移量的操作支持多个主题一起操作。
`--zookeeper` <String: urls>	**必须**，它的值，你懂的。`--zookeeper node1:2181/myKafka`。

这里我们先编写一个生产者，消费者的例子：

我们先启动消费者，再启动生产者， 再通过 bin/kafka-consumer-groups.sh 进行消费偏移量查询，

由于kafka 消费者记录group的消费偏移量有两种方式：

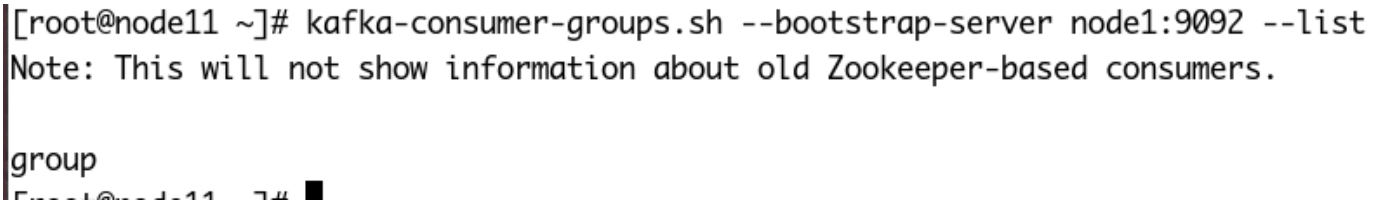
1）kafka 自维护 （新）

2）zookpeer 维护 (旧)，已经逐渐被废弃

所以，脚本只查看由broker维护的，由zookeeper维护的可以将`--bootstrap-server`换成`--zookeeper`即可。

1. 查看有那些 group ID 正在进行消费：

```
[root@node11 ~]# kafka-consumer-groups.sh --bootstrap-server node1:9092 --list
Note: This will not show information about old Zookeeper-based consumers.

group
```



注意：

1. **这里面是没有指定 topic，查看的是所有topic消费者的 group.id 的列表。**
2. **注意： 重名的 group.id 只会显示一次**

2.查看指定group.id 的消费者消费情况

```
[root@node11 ~]# kafka-consumer-groups.sh --bootstrap-server node1:9092 --describe --group group
Note: This will not show information about old Zookeeper-based consumers.


TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID                                   HOST                           CLIENT-ID
tp_demo_02                     0          923             923             0               consumer-1-6d88cc72-1bf1-4ad7-8c6c-060d26dc1c49   /192.168.100.1                 consumer-1
tp_demo_02                     1          872             872             0               consumer-1-6d88cc72-1bf1-4ad7-8c6c-060d26dc1c49   /192.168.100.1                 consumer-1
tp_demo_02                     2          935             935             0               consumer-1-6d88cc72-1bf1-4ad7-8c6c-060d26dc1c49   /192.168.100.1                 consumer-1
[root@node11 ~]#
```

如果消费者停止，查看偏移量信息：

```
[root@node11 ~]# kafka-consumer-groups.sh --bootstrap-server node11:9092 --describe --group group
Note: This will not show information about old Zookeeper-based consumers.

Consumer group 'group' has no active members.

TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID                                   HOST                           CLIENT-ID
tp_demo_02                     1          476             489             13              -                                             -                              -
tp_demo_02                     0          518             536             18              -                                             -                              -
tp_demo_02                     2          546             557             11              -                                             -                              -
[root@node11 ~]#
```

将偏移量设置为最早的：

```
[root@node11 ~]# kafka-consumer-groups.sh --bootstrap-server node1:9092 --reset-offsets --group group --to-earliest --topic tp_demo_02
Note: This will not show information about old Zookeeper-based consumers.


TOPIC                          PARTITION  NEW-OFFSET
tp_demo_02                     2          0
tp_demo_02                     1          0
tp_demo_02                     0          0
[root@node11 ~]#
```

将偏移量设置为最新的：

```
[root@node11 ~]# kafka-consumer-groups.sh --bootstrap-server node1:9092 --reset-offsets --group group --to-latest --topic tp_demo_02
Note: This will not show information about old Zookeeper-based consumers.


TOPIC                          PARTITION  NEW-OFFSET
tp_demo_02                     2          935
tp_demo_02                     1          872
tp_demo_02                     0          923
[root@node11 ~]#
```

分别将指定主题的指定分区的偏移量向前移动10个消息：

```
[root@node11 ~]# kafka-consumer-groups.sh --bootstrap-server node1:9092 --reset-offsets --group group  --topic tp_demo_02:2 --shift-by -10
Note: This will not show information about old Zookeeper-based consumers.


TOPIC                          PARTITION  NEW-OFFSET
tp_demo_02                     2          909
[root@node11 ~]# kafka-consumer-groups.sh --bootstrap-server node1:9092 --reset-offsets --group group  --topic tp_demo_02:1 --shift-by -10
Note: This will not show information about old Zookeeper-based consumers.


TOPIC                          PARTITION  NEW-OFFSET
tp_demo_02                     1          850
[root@node11 ~]# kafka-consumer-groups.sh --bootstrap-server node1:9092 --reset-offsets --group group  --topic tp_demo_02:0 --shift-by -10
Note: This will not show information about old Zookeeper-based consumers.


TOPIC                          PARTITION  NEW-OFFSET
tp_demo_02                     0          900
```

代码：

KafkaProducerSingleton.java

```java
package com.lagou.kafka.demo.producer;

import org.apache.kafka.clients.producer.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
import java.util.Random;

public class KafkaProducerSingleton {

    private static final Logger LOGGER =
LoggerFactory.getLogger(KafkaProducerSingleton.class);
    private static KafkaProducer<String, String> kafkaProducer;
    private Random random = new Random();
    private String topic;
    private int retry;

    private KafkaProducerSingleton() {
    }

    /**
     * 静态内部类
     *
     * @author tanjie
     */
    private static class LazyHandler {
        private static final KafkaProducerSingleton instance = new
KafkaProducerSingleton();
```

```
30      }
31
32      /**
33       * 单例模式,kafkaProducer是线程安全的,可以多线程共享一个实例
34       * @return
35       */
36      public static final KafkaProducerSingleton getInstance() {
37          return LazyHandler.instance;
38      }
39
40      /**
41       * kafka生产者进行初始化
42       *
43       * @return KafkaProducer
44       */
45      public void init(String topic, int retry) {
46          this.topic = topic;
47          this.retry = retry;
48          if (null == kafkaProducer) {
49              Properties props = new Properties();
50              props.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "node1:9092");
51              props.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringSerializer");
52              props.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringSerializer");
53              props.setProperty(ProducerConfig.ACKS_CONFIG, "1");
54
55              kafkaProducer = new KafkaProducer<String, String>(props);
56          }
57      }
58
59      /**
60       * 通过kafkaProducer发送消息
61       * @param message
62       */
63      public void sendKafkaMessage(final String message) {
64
65          ProducerRecord<String, String> record = new ProducerRecord<String, String>(
66                  topic, random.nextInt(3), "", message);
67
68          kafkaProducer.send(record, new Callback() {
69              public void onCompletion(RecordMetadata recordMetadata,
70                                       Exception exception) {
71                  if (null != exception) {
72                      LOGGER.error("kafka发送消息失败:" + exception.getMessage(),
    exception);
73                      retryKakfaMessage(message);
74                  }
75              }
76          });
```

```
77      }
78
79      /**
80       * 当kafka消息发送失败后,重试
81       *
82       * @param retryMessage
83       */
84      private void retryKakfaMessage(final String retryMessage) {
85          ProducerRecord<String, String> record = new ProducerRecord<String, String>(
86                  topic, random.nextInt(3), "", retryMessage);
87          for (int i = 1; i <= retry; i++) {
88              try {
89                  kafkaProducer.send(record);
90                  return;
91              } catch (Exception e) {
92                  LOGGER.error("kafka发送消息失败:" + e.getMessage(), e);
93                  retryKakfaMessage(retryMessage);
94              }
95          }
96      }
97
98      /**
99       * kafka实例销毁
100      */
101     public void close() {
102         if (null != kafkaProducer) {
103             kafkaProducer.close();
104         }
105     }
106
107     public String getTopic() {
108         return topic;
109     }
110
111     public void setTopic(String topic) {
112         this.topic = topic;
113     }
114
115     public int getRetry() {
116         return retry;
117     }
118
119     public void setRetry(int retry) {
120         this.retry = retry;
121     }
122 }
123
```

ProducerHandler.java

```java
package com.lagou.kafka.demo.producer;

public class ProducerHandler implements Runnable {
    private String message;

    public ProducerHandler(String message) {
        this.message = message;
    }

    @Override
    public void run() {
        KafkaProducerSingleton kafkaProducerSingleton =
KafkaProducerSingleton.getInstance();
        kafkaProducerSingleton.init("tp_demo_02", 3);
        int i = 0;

        while (true) {
            try {
                System.out.println("当前线程:" + Thread.currentThread().getName()
                        + "\t获取的kafka实例:" + kafkaProducerSingleton);
                kafkaProducerSingleton.sendKafkaMessage("发送消息: " + message + " " +
(++i));
                Thread.sleep(100);
            } catch (Exception e) {
            }
        }
    }
}

```

MyProducer.java

```java
package com.lagou.kafka.demo.producer;

public class MyProducer {
    public static void main(String[] args){
        Thread thread = new Thread(new ProducerHandler("hello lagou "));
        thread.start();
    }
}

```

KafkaConsumerAuto.java

```java
package com.lagou.kafka.demo.consumer;

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

```

```java

import java.util.Arrays;
import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class KafkaConsumerAuto {
    /**
     * kafka消费者不是线程安全的
     */
    private final KafkaConsumer<String, String> consumer;

    private ExecutorService executorService;

    public KafkaConsumerAuto() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "group");
        // 关闭自动提交
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put("auto.commit.interval.ms", "100");
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "30000");

        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");

        consumer = new KafkaConsumer<String, String>(props);
        // 订阅主题
        consumer.subscribe(Collections.singleton("tp_demo_02"));
    }

    public void execute() throws InterruptedException {
        executorService = Executors.newFixedThreadPool(2);
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(2_000);
            if (null != records) {
                executorService.submit(new ConsumerThreadAuto(records, consumer));
            }
            Thread.sleep(1000);
        }
    }

    public void shutdown() {
        try {
            if (consumer != null) {
                consumer.close();
            }
```

```
            if (executorService != null) {
                executorService.shutdown();
            }
            if (!executorService.awaitTermination(10, TimeUnit.SECONDS)) {
                System.out.println("关闭线程池超时。。。");
            }
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }
}

```

ConsumerThreadAuto.java

```
package com.lagou.kafka.demo.consumer;

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

public class ConsumerThreadAuto implements Runnable {
    private ConsumerRecords<String, String> records;
    private KafkaConsumer<String, String> consumer;

    public ConsumerThreadAuto(ConsumerRecords<String, String> records,
                              KafkaConsumer<String, String> consumer) {
        this.records = records;
        this.consumer = consumer;
    }

    @Override
    public void run() {

        for(ConsumerRecord<String,String> record : records){
            System.out.println("当前线程:" + Thread.currentThread()
                    + "\t主题:" + record.topic()
                    + "\t偏移量:" + record.offset() + "\t分区:" + record.partition()
                    + "\t获取的消息:" + record.value());
        }
    }
}

```

ConsumerAutoMain.java

```
package com.lagou.kafka.demo.consumer;

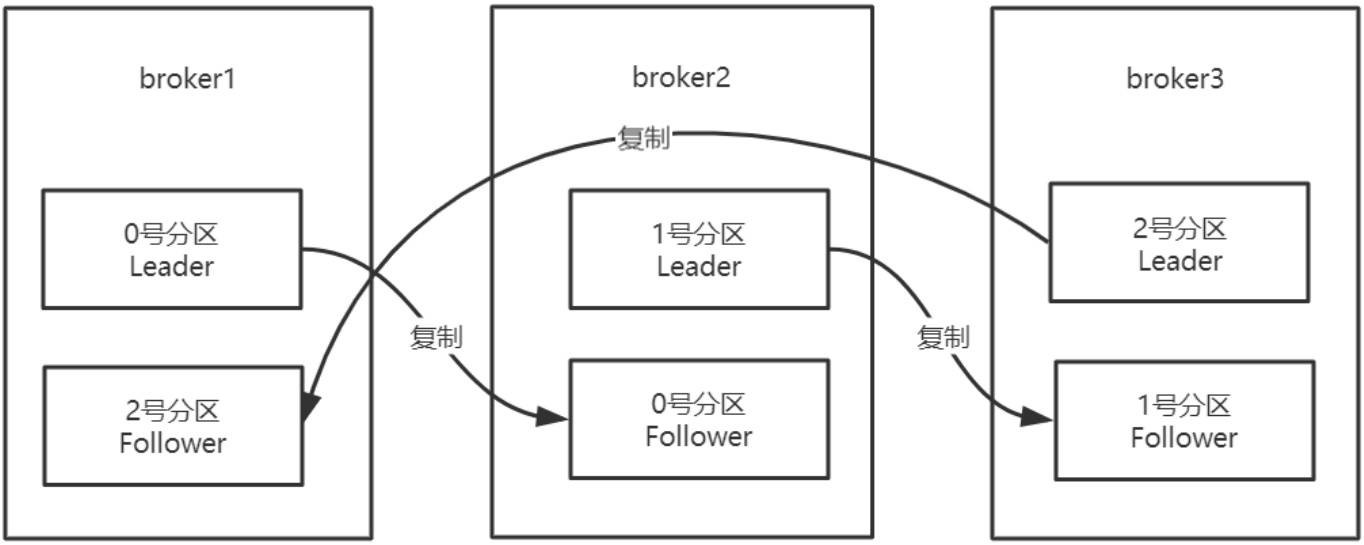
public class ConsumerAutoMain {
    public static void main(String[] args) {
```

```
        KafkaConsumerAuto kafka_consumerAuto = new KafkaConsumerAuto();
        try {
            kafka_consumerAuto.execute();
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            kafka_consumerAuto.shutdown();
        }
    }
}

```

2.4 分区

2.4.1 副本机制



Kafka在一定数量的服务器上对主题分区进行复制。

当集群中的一个broker宕机后系统可以自动故障转移到其他可用的副本上，不会造成数据丢失。

--replication-factor 3 1leader+2follower

1. 将复制因子为1的未复制主题称为复制主题。
2. 主题的分区是复制的最小单元。
3. 在非故障情况下，Kafka中的每个分区都有一个Leader副本和零个或多个Follower副本。
4. 包括Leader副本在内的副本总数构成复制因子。

5. 所有读取和写入都由Leader副本负责。
6. 通常，分区比broker多，并且Leader分区在broker之间平均分配。

Follower分区像*普通的Kafka消费者*一样，消费来自Leader分区的消息，并将其持久化到自己的日志中。

允许Follower对日志条目拉取进行**批处理**。

同步节点定义：

1. 节点必须能够维持与ZooKeeper的会话（通过ZooKeeper的心跳机制）
2. 对于Follower副本分区，它复制在Leader分区上的写入，并且不要延迟太多

Kafka提供的保证是，只要有至少一个同步副本处于活动状态，提交的消息就不会丢失。

宕机如何恢复

（1）少部分副本宕机

当leader宕机了，会从follower选择一个作为leader。当宕机的重新恢复时，会把之前commit的数据清空，重新从leader里pull数据。

（2）全部副本宕机

当全部副本宕机了有两种恢复方式

1、等待ISR中的一个恢复后，并选它作为leader。（等待时间较长，降低可用性）

2、选择第一个恢复的副本作为新的leader，无论是否在ISR中。（并未包含之前leader commit的数据，因此造成数据丢失）

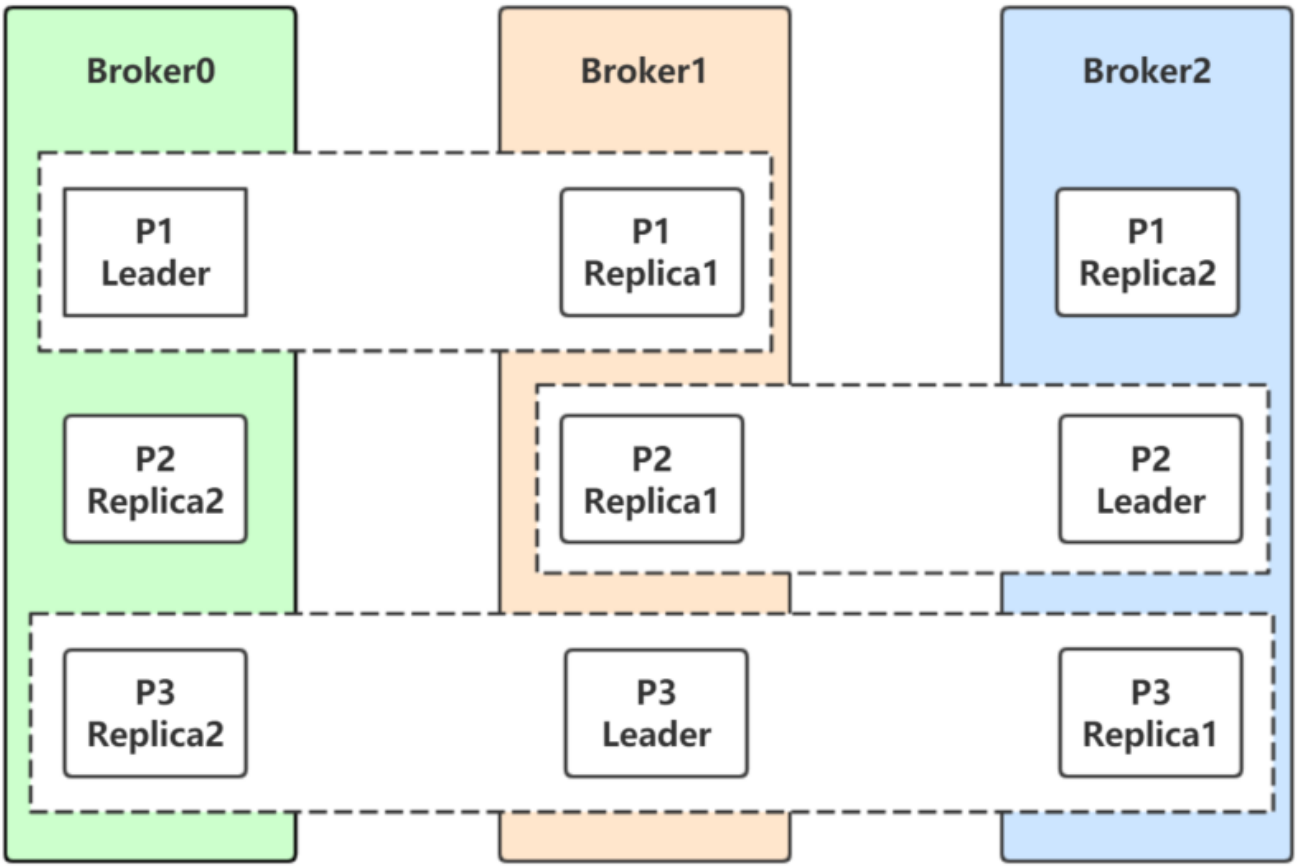
2.4.2 Leader选举

下图中

分区P1的Leader是0，ISR是0和1

分区P2的Leader是2，ISR是1和2

分区P3的Leader是1，ISR是0，1，2。



生产者和消费者的请求都由Leader副本来处理。Follower副本只负责消费Leader副本的数据和Leader保持同步。

对于P1，如果0宕机会发生什么?

Leader副本和Follower副本之间的关系并不是固定不变的，在Leader所在的broker发生故障的时候，就需要进行分区的Leader副本和Follower副本之间的切换，需要选举Leader副本。

如何选举?

如果某个分区所在的服务器除了问题，不可用，kafka会从该分区的其他的副本中选择一个作为新的Leader。之后所有的读写就会转移到这个新的Leader上。现在的问题是应当选择哪个作为新的Leader。

只有那些跟Leader保持同步的Follower才应该被选作新的Leader。

Kafka会在Zookeeper上针对每个Topic维护一个称为ISR（in-sync replica，已同步的副本）的集合，该集合中是一些分区的副本。

只有当这些副本都跟Leader中的副本同步了之后，kafka才会认为消息已提交，并反馈给消息的生产者。

如果这个集合有增减，kafka会更新zookeeper上的记录。

如果某个分区的Leader不可用，Kafka就会从ISR集合中选择一个副本作为新的Leader。

显然通过ISR，kafka需要的**冗余度较低**，可以容忍的失败数比较高。

假设某个topic有N+1个副本，kafka可以容忍N个服务器不可用。

为什么不用少数服从多数的方法

少数服从多数是一种比较常见的一致性算发和Leader选举法。

它的含义是只有超过半数的副本同步了，系统才会认为数据已同步；

选择Leader时也是从超过半数的同步的副本中选择。

这种算法需要较高的冗余度，跟Kafka比起来，浪费资源。

譬如只允许一台机器失败，需要有三个副本；而如果只容忍两台机器失败，则需要五个副本。

而kafka的ISR集合方法，分别只需要两个和三个副本。

如果所有的ISR副本都失败了怎么办?

此时有两种方法可选，

1. 等待ISR集合中的副本复活，
2. 选择任何一个立即可用的副本，而这个副本不一定是在ISR集合中。
    - 需要设置`unclean.leader.election.enable=true`

这两种方法各有利弊，实际生产中按需选择。
如果要等待ISR副本复活，虽然可以保证一致性，但可能需要很长时间。而如果选择立即可用的副本，则很可能该副本并不一致。

总结:

Kafka中Leader分区选举，通过维护一个动态变化的ISR集合来实现，一旦Leader分区丢掉，则从ISR中随机挑选一个副本做新的Leader分区。

如果ISR中的副本都丢失了，则:

1. 可以等待ISR中的副本任何一个恢复，接着对外提供服务，需要时间等待。
2. 从OSR中选出一个副本做Leader副本，此时会造成数据丢失

2.4.3 分区重新分配

向已经部署好的Kafka集群里面添加机器，我们需要从已经部署好的Kafka节点中复制相应的配置文件，然后把里面的broker id修改成全局唯一的，最后启动这个节点即可将它加入到现有Kafka集群中。

问题：新添加的Kafka节点并不会自动地分配数据，无法分担集群的负载，除非我们新建一个topic。

需要手动将部分分区移到新添加的Kafka节点上，Kafka内部提供了相关的工具来重新分布某个topic的分区。

在重新分布topic分区之前，我们先来看看现在topic的各个分区的分布位置:

1. 创建主题:

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --create --topic tp_re_01 --partitions 5 --replication-factor 1
```

2. 查看主题信息：

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_re_01
Topic:tp_re_01  PartitionCount:5    ReplicationFactor:1 Configs:
Topic: tp_re_01 Partition: 0    Leader: 0   Replicas: 0 Isr: 0
Topic: tp_re_01 Partition: 1    Leader: 0   Replicas: 0 Isr: 0
Topic: tp_re_01 Partition: 2    Leader: 0   Replicas: 0 Isr: 0
Topic: tp_re_01 Partition: 3    Leader: 0   Replicas: 0 Isr: 0
Topic: tp_re_01 Partition: 4    Leader: 0   Replicas: 0 Isr: 0
```

3. 在node11搭建Kafka：
    - 拷贝JDK并安装

```
[root@node1 opt]# scp jdk-8u261-linux-x64.rpm node11:~
```

此处不需要zookeeper，切记！！！

```
export JAVA_HOME=/usr/java/jdk1.8.0_261-amd64
export PATH=$PATH:$JAVA_HOME/bin

export KAFKA_HOME=/opt/kafka_2.12-1.0.2
export PATH=$PATH:$KAFKA_HOME/bin
```

让配置生效：

```
. /etc/profile
```

    - 拷贝node1上安装的Kafka

```
[root@node1 opt]# scp -r kafka_2.12-1.0.2/ node11:/opt
```

    - 修改node11上Kafka的配置：

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

```
# zookeeper.connect=localhost:2181
zookeeper.connect=node1:2181/myKafka
```

    - 启动Kafka：

```
[root@node11 ~]# kafka-server-start.sh /opt/kafka_2.12-1.0.2/config/server.properties
```

注意观察node11上节点启动的时候的ClusterId，看和zookeeper节点上的ClusterId是否一致，如果是，证明node11和node1在同一个集群中。

node11启动的Cluster ID：

```
] INFO Cluster ID = EFGahMX4STibP5QHAFca7w (kafka.server.KafkaServer)
```

zookeeper节点上的Cluster ID：

```
[zk: localhost:2181(CONNECTED) 7] get /myKafka/cluster/id
{"version":"1","id":"EFGahMX4STibP5QHAFca7w"}
```

在node1上查看zookeeper的节点信息：

- ```
  [zk: localhost:2181(CONNECTED) 3] ls /myKafka/brokers/ids
  [0, 1]
  [zk: localhost:2181(CONNECTED) 4] get /myKafka/brokers/ids/1
  {"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://node11:9092"],"jmx_port":-1,"host":"node11","timestamp":"1596360457046","port":9092,"version":4}
  cZxid = 0x5b9
  ```

  node11的节点已经加入集群了。

4. 现在我们在现有集群的基础上再添加一个Kafka节点，然后使用Kafka自带的`kafka-reassign-partitions.sh`工具来重新分布分区。该工具有三种使用模式：

   1、generate模式，给定需要重新分配的Topic，自动生成reassign plan（并不执行）
   2、execute模式，根据指定的reassign plan重新分配Partition
   3、verify模式，验证重新分配Partition是否成功

5. 我们将分区3和4重新分布到broker1上，借助kafka-reassign-partitions.sh工具生成reassign plan，不过我们先得按照要求定义一个文件，里面说明哪些topic需要重新分区，文件内容如下：

```
[root@node1 ~]# cat topics-to-move.json
{
  "topics": [
    {
      "topic":"tp_re_01"
    }
  ],
  "version":1
}
```

然后使用`kafka-reassign-partitions.sh`工具生成reassign plan

```
[root@node1 ~]# kafka-reassign-partitions.sh --zookeeper node1:2181/myKafka --topics-to-move-json-file topics-to-move.json --broker-list "0,1" --generate
Current partition replica assignment
{"version":1,"partitions":[{"topic":"tp_re_01","partition":4,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":1,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":2,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":3,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":0,"replicas":[0],"log_dirs":["any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"tp_re_01","partition":4,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":1,"replicas":[1],"log_dirs":["any"]},{"topic":"tp_re_01","partition":2,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":3,"replicas":[1],"log_dirs":["any"]},{"topic":"tp_re_01","partition":0,"replicas":[0],"log_dirs":["any"]}]}
[root@node1 ~]#
```
```

```
[root@node1 ~]# kafka-reassign-partitions.sh --zookeeper node1:2181/myKafka --topics-to-move-json-file topics-to-move.json --broker-list "0,1" --generate
Current partition replica assignment
{"version":1,"partitions":[{"topic":"tp_re_01","partition":4,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":1,"replicas":[0],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":2,"replicas":[0],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":3,"replicas":[0],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":0,"replicas":[0],"log_dirs":["any"]}]}

Proposed partition reassignment configuration
{"version":1,"partitions":[{"topic":"tp_re_01","partition":4,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":1,"replicas":[1],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":2,"replicas":[0],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":3,"replicas":[1],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":0,"replicas":[0],"log_dirs":["any"]}]}
```

Proposed partition reassignment configuration下面生成的就是将分区重新分布到broker 1上的结果。我们将这些内容保存到名为result.json文件里面（文件名不重要，文件格式也不一定要以json为结尾，只要保证内容是json即可），然后执行这些reassign plan：

```
[root@node1 ~]# cat topics-to-execute.json
{"version":1,"partitions":[{"topic":"tp_re_01","partition":4,"replicas":[0],"log_dirs":["any"]},{"topic":"t
e_01","partition":1,"replicas":[1],"log_dirs":["any"]},{"topic":"tp_re_01","partition":2,"replicas":[0],"lo
irs":["any"]},{"topic":"tp_re_01","partition":3,"replicas":[1],"log_dirs":["any"]},{"topic":"tp_re_01","par
ion":0,"replicas":[0],"log_dirs":["any"]}]}
[root@node1 ~]#
```

执行计划：

```
[root@node1 ~]# kafka-reassign-partitions.sh --zookeeper node1:2181/myKafka --reassignment-json-file topics-to-execute.json --execute
Current partition replica assignment

{"version":1,"partitions":[{"topic":"tp_re_01","partition":4,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":1,"replicas":[0],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":2,"replicas":[0],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":3,"replicas":[0],"log_dirs":["any"]},
{"topic":"tp_re_01","partition":0,"replicas":[0],"log_dirs":["any"]}]}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions.
[root@node1 ~]#
```

这样Kafka就在执行reassign plan，我们可以校验reassign plan是否执行完成：

```
[root@node1 ~]# kafka-reassign-partitions.sh --zookeeper node1:2181/myKafka --reassignment-json-file topics-to-execute.json --verify
Status of partition reassignment:
Reassignment of partition tp_re_01-1 completed successfully
Reassignment of partition tp_re_01-4 completed successfully
Reassignment of partition tp_re_01-2 completed successfully
Reassignment of partition tp_re_01-3 completed successfully
Reassignment of partition tp_re_01-0 completed successfully
[root@node1 ~]#
```

查看主题的细节：

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_re_01
Topic:tp_re_01  PartitionCount:5        ReplicationFactor:1     Configs:
        Topic: tp_re_01 Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: tp_re_01 Partition: 1    Leader: 1       Replicas: 1     Isr: 1
        Topic: tp_re_01 Partition: 2    Leader: 0       Replicas: 0     Isr: 0
        Topic: tp_re_01 Partition: 3    Leader: 1       Replicas: 1     Isr: 1
        Topic: tp_re_01 Partition: 4    Leader: 0       Replicas: 0     Isr: 0
[root@node1 ~]#
```

分区的分布的确和操作之前不一样了，broker 1上已经有分区分布上去了。使用`kafka-reassign-partitions.sh`工具生成的reassign plan只是一个建议，方便大家而已。其实我们自己完全可以编辑一个reassign plan，然后执行它，如下：

```
{
    "version": 1,
    "partitions": [{
        "topic": "tp_re_01",
        "partition": 4,
        "replicas": [1],
        "log_dirs": ["any"]
    }, {
        "topic": "tp_re_01",
        "partition": 1,
        "replicas": [0],
        "log_dirs": ["any"]
    }, {
        "topic": "tp_re_01",
        "partition": 2,
        "replicas": [0],
        "log_dirs": ["any"]
    }, {
        "topic": "tp_re_01",
        "partition": 3,
        "replicas": [1],
        "log_dirs": ["any"]
    }, {
```

```
        "topic": "tp_re_01",
        "partition": 0,
        "replicas": [0],
        "log_dirs": ["any"]
    }]
}
```

将上面的json数据文件保存到my-topics-to-execute.json文件中，然后也是执行它：

```
[root@node1 ~]# kafka-reassign-partitions.sh --zookeeper node1:2181/myKafka --reassignment-json-file my-topics-to-execute.json --execute
Current partition replica assignment

{"version":1,"partitions":[{"topic":"tp_re_01","partition":4,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":1,"replicas":[1],"log_dirs":["any"]},{"topic":"tp_re_01","partition":2,"replicas":[0],"log_dirs":["any"]},{"topic":"tp_re_01","partition":3,"replicas":[1],"log_dirs":["any"]},{"topic":"tp_re_01","partition":0,"replicas":[0],"log_dirs":["any"]}]}

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions.
[root@node1 ~]# kafka-reassign-partitions.sh --zookeeper node1:2181/myKafka --reassignment-json-file my-topics-to-execute.json --verify
Status of partition reassignment:
Reassignment of partition tp_re_01-1 completed successfully
Reassignment of partition tp_re_01-4 completed successfully
Reassignment of partition tp_re_01-2 completed successfully
Reassignment of partition tp_re_01-3 completed successfully
Reassignment of partition tp_re_01-0 completed successfully
```

等这个reassign plan执行完，我们再来看看分区的分布：

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_re_01
Topic:tp_re_01  PartitionCount:5    ReplicationFactor:1 Configs:
    Topic: tp_re_01 Partition: 0    Leader: 0   Replicas: 0 Isr: 0
    Topic: tp_re_01 Partition: 1    Leader: 0   Replicas: 0 Isr: 0
    Topic: tp_re_01 Partition: 2    Leader: 0   Replicas: 0 Isr: 0
    Topic: tp_re_01 Partition: 3    Leader: 1   Replicas: 1 Isr: 1
    Topic: tp_re_01 Partition: 4    Leader: 1   Replicas: 1 Isr: 1
[root@node1 ~]#
```

搞定！

2.4.4 自动再均衡

我们可以在新建主题的时候，手动指定主题各个Leader分区以及Follower分区的分配情况，即什么分区副本在哪个broker节点上。

随着系统的运行，broker的宕机重启，会引发Leader分区和Follower分区的角色转换，最后可能Leader大部分都集中在少数几台broker上，由于Leader负责客户端的读写操作，此时集中Leader分区的少数几台服务器的网络I/O，CPU，以及内存都会很紧张。

Leader和Follower的角色转换会引起Leader副本在集群中分布的不均衡，此时我们需要一种手段，让Leader的分布重新恢复到一个均衡的状态。

执行脚本：

```
[root@node11 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --create --topic tp_demo_03 --replica-assignment "0:1,1:0,0:1"
```

上述脚本执行的结果是：创建了主题tp_demo_03，有三个分区，每个分区两个副本，Leader副本在列表中第一个指定的brokerId上，Follower副本在随后指定的brokerId上。

```
[root@node11 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_demo_03
Topic:tp_demo_03        PartitionCount:3        ReplicationFactor:2     Configs:
        Topic: tp_demo_03       Partition: 0    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: tp_demo_03       Partition: 1    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: tp_demo_03       Partition: 2    Leader: 0       Replicas: 0,1   Isr: 0,1
```

然后模拟broker0宕机的情况：

```
# 通过jps找到Kafka进程PID
[root@node1 ~]# jps
54912 Jps
1699 QuorumPeerMain
1965 Kafka
# 直接杀死进程
[root@node1 ~]# kill -9 1965
[root@node1 ~]# jps
1699 QuorumPeerMain
54936 Jps
[root@node1 ~]#
# 查看主题分区信息：
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_demo_03
Topic:tp_demo_03    PartitionCount:3    ReplicationFactor:2 Configs:
    Topic: tp_demo_03   Partition: 0    Leader: 1   Replicas: 0,1   Isr: 1
    Topic: tp_demo_03   Partition: 1    Leader: 1   Replicas: 1,0   Isr: 1
```

```
    Topic: tp_demo_03   Partition: 2    Leader: 1   Replicas: 0,1   Isr: 1
[root@node1 ~]#
# 重新启动node1上的Kafka
[root@node1 ~]# kafka-server-start.sh -daemon /opt/kafka_2.12-1.0.2/config/server.properties
[root@node1 ~]# jps
1699 QuorumPeerMain
55525 Kafka
55557 Jps
[root@node1 ~]#
# 查看主题的分区信息:
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_demo_03
Topic:tp_demo_03    PartitionCount:3    ReplicationFactor:2 Configs:
    Topic: tp_demo_03   Partition: 0    Leader: 1   Replicas: 0,1   Isr: 1,0
    Topic: tp_demo_03   Partition: 1    Leader: 1   Replicas: 1,0   Isr: 1,0
    Topic: tp_demo_03   Partition: 2    Leader: 1   Replicas: 0,1   Isr: 1,0
[root@node1 ~]#
# broker恢复了，但是Leader的分配并没有变化，还是处于Leader切换后的分配情况。
```

是否有一种方式，可以让Kafka自动帮我们进行修改？改为初始的副本分配？

此时，用到了Kafka提供的自动再均衡脚本：`kafka-preferred-replica-election.sh`

先看介绍：

```
[root@node11 ~]# kafka-preferred-replica-election.sh
This tool causes leadership for each partition to be transferred back to the 'preferred replica', it can be used to balance leadership among the servers.
Option                                  Description
------                                  -----------
--path-to-json-file <String: list of    The JSON file with the list of
  partitions for which preferred          partitions for which preferred
  replica leader election needs to be     replica leader election should be
  triggered>                              done, in the following format -
                                        {"partitions":
                                         [{"topic": "foo", "partition": 1},
                                          {"topic": "foobar", "partition": 2}]
                                        }
                                        Defaults to all existing partitions
--zookeeper <String: urls>              REQUIRED: The connection string for
                                          the zookeeper connection in the form
                                          host:port. Multiple URLS can be
                                          given to allow fail-over.
[root@node11 ~]#
```

该工具会让每个分区的Leader副本分配在合适的位置，让Leader分区和Follower分区在服务器之间均衡分配。

如果该脚本仅指定zookeeper地址，则会对集群中所有的主题进行操作，自动再平衡。

具体操作：

1. 创建preferred-replica.json，内容如下：

```
{
    "partitions": [
        {
            "topic":"tp_demo_03",
            "partition":0
        },
```

```
        {
            "topic":"tp_demo_03",
            "partition":1
        },
        {
            "topic":"tp_demo_03",
            "partition":2
        }
    ]
}
```

2. 执行操作：

```
[root@node1 ~]# kafka-preferred-replica-election.sh --zookeeper node1:2181/myKafka --path-to-json-file preferred-replicas.json
Created preferred replica election path with {"version":1,"partitions":[{"topic":"tp_demo_03","partition":0},{"topic":"tp_demo_03","partition":1},{"topic":"tp_demo_03","partition":2}]}
Successfully started preferred replica election for partitions Set(tp_demo_03-0, tp_demo_03-1, tp_demo_03-2)
[root@node1 ~]#
```

3. 查看操作的结果：

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_demo_03
Topic:tp_demo_03    PartitionCount:3    ReplicationFactor:2 Configs:
Topic: tp_demo_03   Partition: 0    Leader: 0   Replicas: 0,1   Isr: 1,0
    Topic: tp_demo_03   Partition: 1    Leader: 1   Replicas: 1,0   Isr: 1,0
    Topic: tp_demo_03   Partition: 2    Leader: 0   Replicas: 0,1   Isr: 1,0
[root@node1 ~]#
```

恢复到最初的分配情况。

之所以是这样的分配，是因为我们在创建主题的时候：

```
--replica-assignment "0:1,1:0,0:1"
```

在逗号分割的每个数值对中排在前面的是Leader分区，后面的是副本分区。那么所谓的preferred replica，就是排在前面的数字就是Leader副本应该在的brokerId。

2.4.5 修改分区副本

实际项目中，我们可能由于主题的副本因子设置的问题，需要重新设置副本因子

或者由于集群的扩展，需要重新设置副本因子。

topic一旦使用又不能轻易删除重建，因此动态增加副本因子就成为最终的选择。

说明：kafka 1.0版本配置文件默认没有default.replication.factor=x， 因此如果创建topic时，不指定–replication-factor 想， 默认副本因子为1. 我们可以在自己的server.properties中配置上常用的副本因子，省去手动调整。例如设置default.replication.factor=3， 详细内容可参考官方文档https://kafka.apache.org/documentation/#replication

原因分析：

假设我们有2个kafka broker分别broker0，broker1。

1. 当我们创建的topic有2个分区partition时并且replication-factor为1，基本上一个broker上一个分区。当一个broker宕机了，该topic就无法使用了，因为两个个分区只有一个能用。
2. 当我们创建的topic有3个分区partition时并且replication-factor为2时，可能分区数据分布情况是
   broker0， partiton0，partiton1，partiton2，
   broker1， partiton1，partiton0，partiton2，

   每个分区有一个副本，当其中一个broker宕机了，kafka集群还能完整凑出该topic的两个分区，例如当broker0宕机了，可以通过broker1组合出topic的两个分区。

1. 创建主题：

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --create --topic tp_re_02 --partitions 3 --replication-factor 1
```

2. 查看主题细节：

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_re_02
Topic:tp_re_02  PartitionCount:3    ReplicationFactor:1 Configs:
Topic: tp_re_02 Partition: 0    Leader: 1   Replicas: 1 Isr: 1
Topic: tp_re_02 Partition: 1    Leader: 0   Replicas: 0 Isr: 0
Topic: tp_re_02 Partition: 2    Leader: 1   Replicas: 1 Isr: 1
[root@node1 ~]#
```

3. 修改副本因子：错误

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --alter --topic tp_re_02 --replication-factor 2
Option "[replication-factor]" can't be used with option"[alter]"
```

4. 使用`kafka-reassign-partitions.sh`修改副本因子：

   -. 创建increment-replication-factor.json

```
{
    "version":1,
    "partitions":[
        {"topic":"tp_re_02","partition":0,"replicas":[0,1]},
        {"topic":"tp_re_02","partition":1,"replicas":[0,1]},
        {"topic":"tp_re_02","partition":2,"replicas":[1,0]}
    ]
}
```

5. 执行分配：

```
[root@node1 ~]# kafka-reassign-partitions.sh --zookeeper node1:2181/myKafka --reassignment-json-file increase-replication-factor.json --execute
Current partition replica assignment

{"version":1,"partitions":[{"topic":"tp_re_02","partition":2,"replicas":[1,0],"log_dirs":["any","any"]},{"topic":"tp_re_02","partition":1,"replicas":[0,1],"log_dirs":["any","any"]},{"topic":"tp_re_02","partition":0,"replicas":[0,1],"log_dirs":["any","any"]}]}

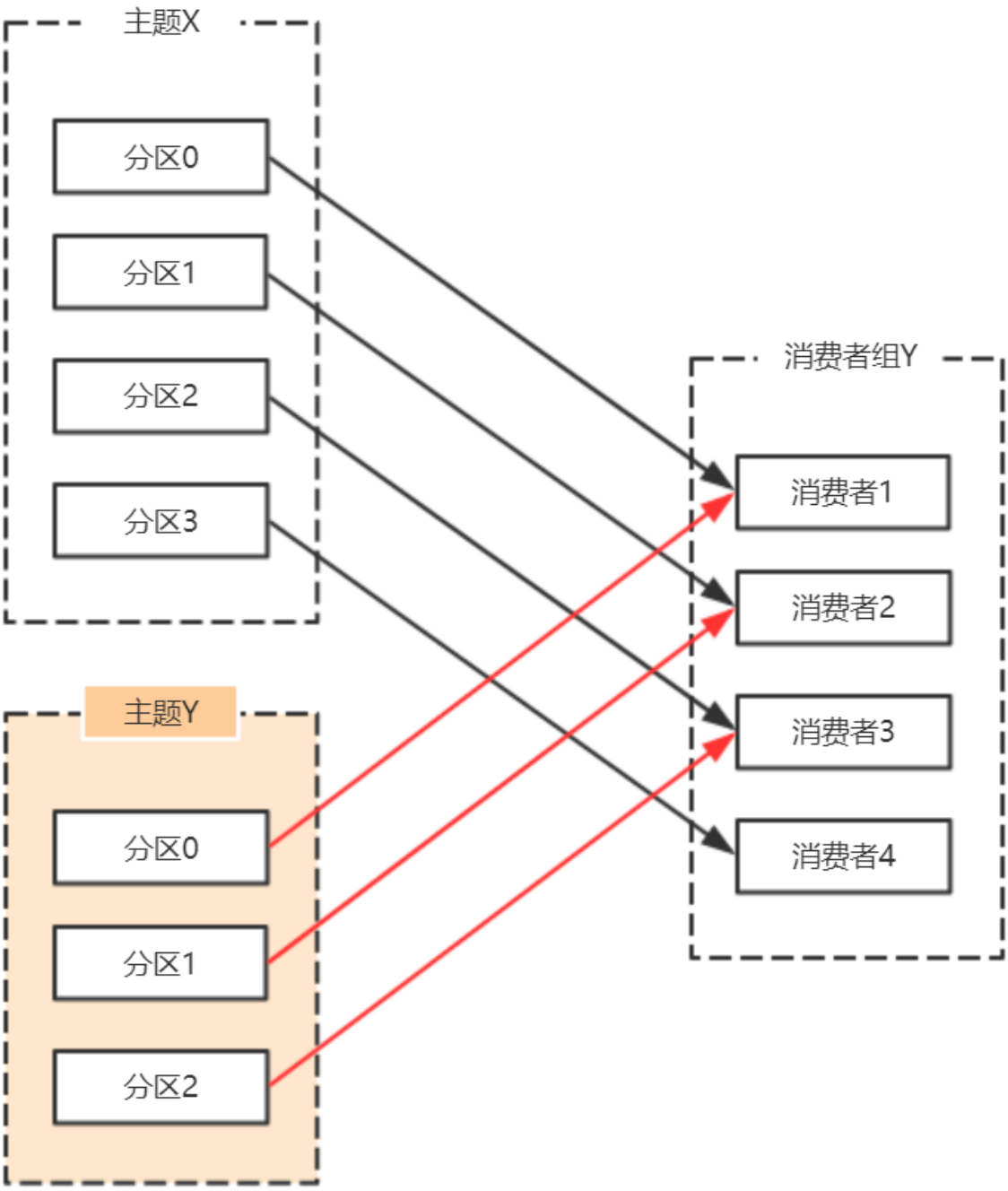
Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions.
```

6. 查看主题细节：

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --describe --topic tp_re_02
Topic:tp_re_02  PartitionCount:3    ReplicationFactor:2 Configs:
    Topic: tp_re_02 Partition: 0    Leader: 1   Replicas: 0,1   Isr: 1,0
    Topic: tp_re_02 Partition: 1    Leader: 0   Replicas: 0,1   Isr: 0,1
    Topic: tp_re_02 Partition: 2    Leader: 1   Replicas: 1,0   Isr: 1,0
[root@node1 ~]#
```

7. 搞定！

2.4.6 分区分配策略



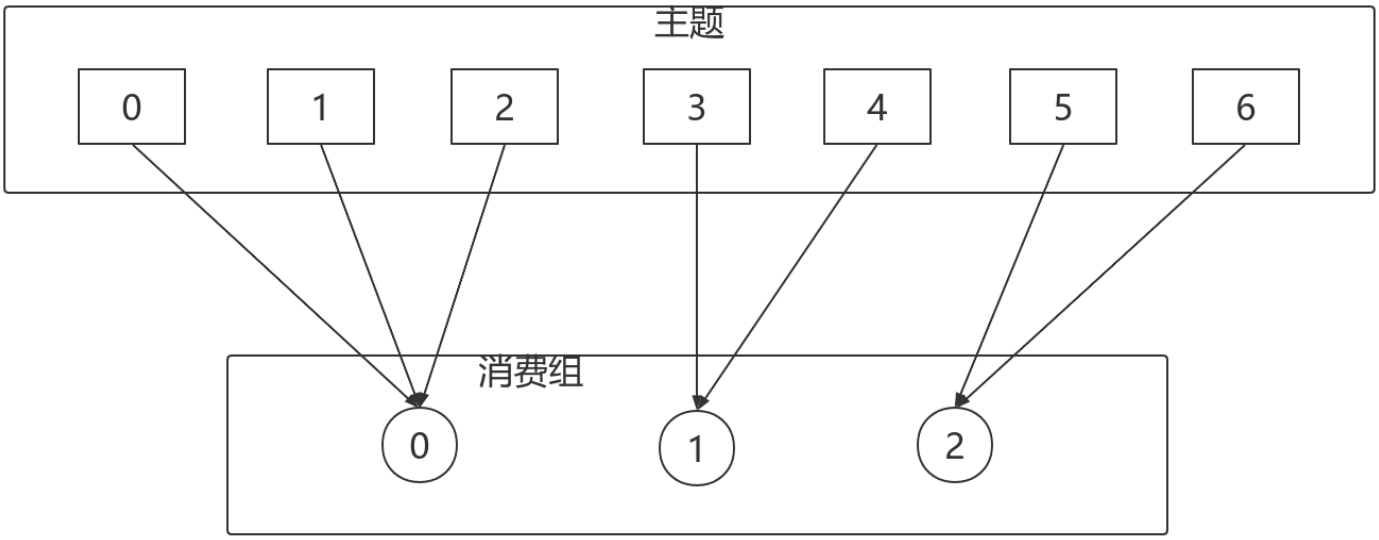
在Kafka中，每个Topic会包含多个分区，默认情况下一个分区只能被一个消费组下面的一个消费者消费，这里就产生了分区分配的问题。Kafka中提供了多重分区分配算法（PartitionAssignor）的实现：RangeAssignor、RoundRobinAssignor、StickyAssignor。

2.4.6.1 RangeAssignor

PartitionAssignor接口用于用户定义实现分区分配算法，以实现Consumer之间的分区分配。

消费组的成员订阅它们感兴趣的Topic并将这种订阅关系传递给作为订阅组协调者的Broker。协调者选择其中的一个消费者来执行这个消费组的分区分配并将分配结果转发给消费组内所有的消费者。**Kafka默认采用RangeAssignor的分配算法。**

RangeAssignor对每个Topic进行独立的分区分配。对于每一个Topic，首先对分区按照分区ID进行数值排序，然后订阅这个Topic的消费组的消费者再进行字典排序，之后尽量均衡的将分区分配给消费者。这里只能是尽量均衡，因为分区数可能无法被消费者数量整除，那么有一些消费者就会多分配到一些分区。



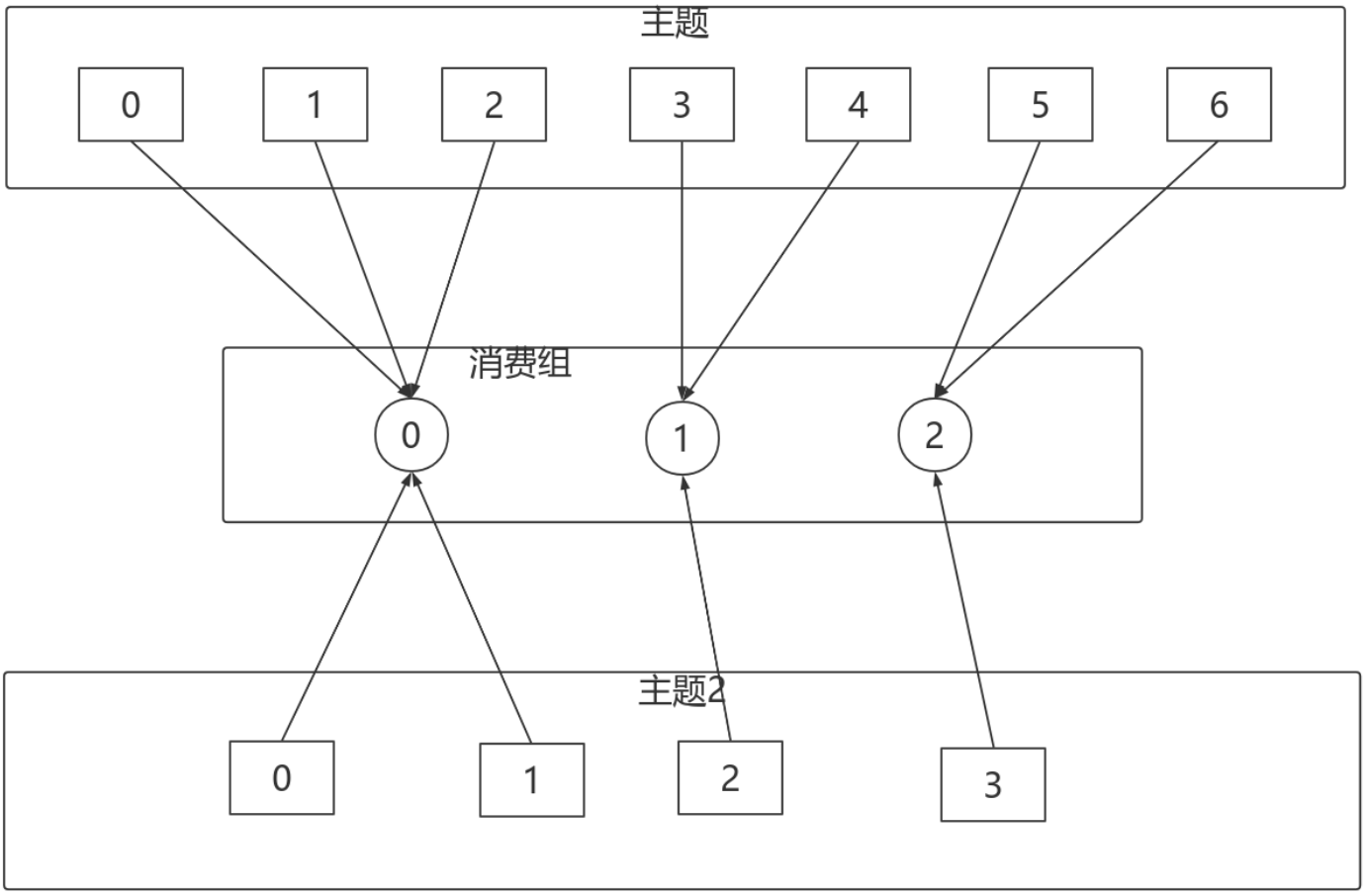
大致算法如下：

```
assign(topic, consumers) {
   // 对分区和Consumer进行排序
   List<Partition> partitions = topic.getPartitions();
   sort(partitions);
   sort(consumers);
   // 计算每个Consumer分配的分区数
   int numPartitionsPerConsumer = partition.size() / consumers.size();
   // 额外有一些Consumer会多分配到分区
   int consumersWithExtraPartition = partition.size() % consumers.size();
   // 计算分配结果
   for (int i = 0, n = consumers.size(); i < n; i++) {
     // 第i个Consumer分配到的分区的index
     int start = numPartitionsPerConsumer * i + Math.min(i,
consumersWithExtraPartition);
     // 第i个Consumer分配到的分区数
     int length = numPartitionsPerConsumer + (i + 1 > consumersWithExtraPartition ? 0
: 1);
     // 分装分配结果
     assignment.get(consumersForTopic.get(i)).addAll(partitions.subList(start, start
+ length));
    }
}
```

RangeAssignor策略的原理是按照消费者总数和分区总数进行整除运算来获得一个跨度，然后将分区按照跨度进行平均分配，以保证分区尽可能均匀地分配给所有的消费者。对于每一个Topic，RangeAssignor策略会将消费组内所有订阅这个Topic的消费者按照名称的字典序排序，然后为每个消费者划分固定的分区范围，如果不够平均分配，那么字典序靠前的消费者会被多分配一个分区。

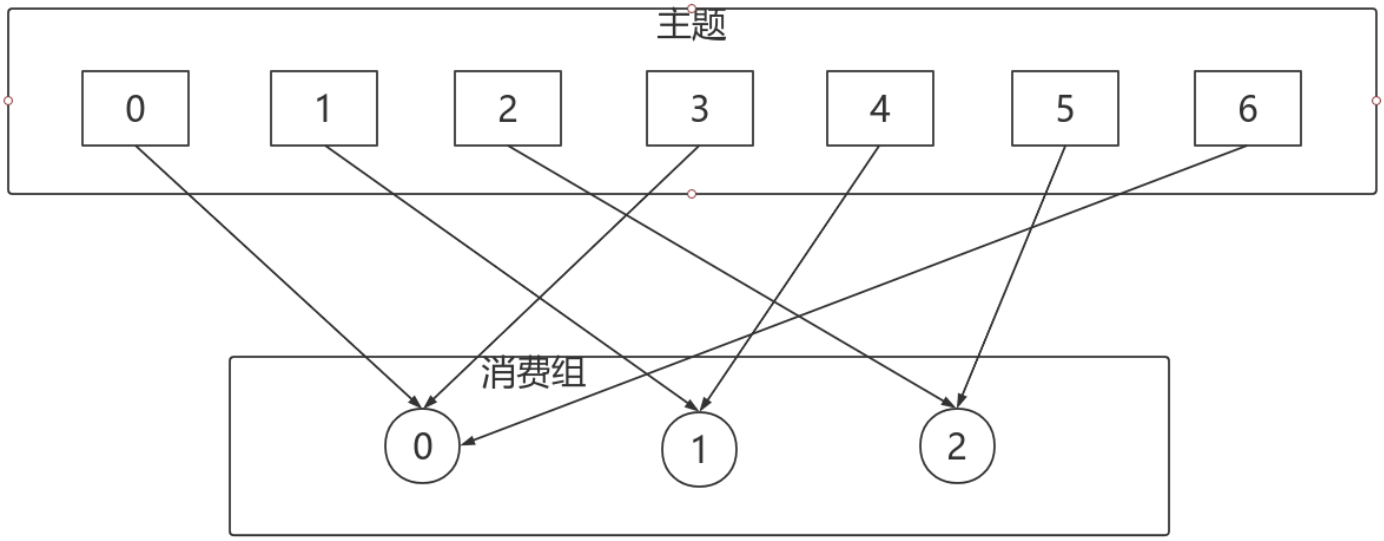
这种分配方式明显的一个问题是随着消费者订阅的Topic的数量的增加，不均衡的问题会越来越严重，比如上图中4个分区3个消费者的场景，C0会多分配一个分区。如果此时再订阅一个分区数为4的Topic，那么C0又会比C1、C2多分配一个分区，这样C0总共就比C1、C2多分配两个分区了，而且随着Topic的增加，这个情况会越来越严重。

字典序靠前的消费组中的消费者比较“**贪婪**”。



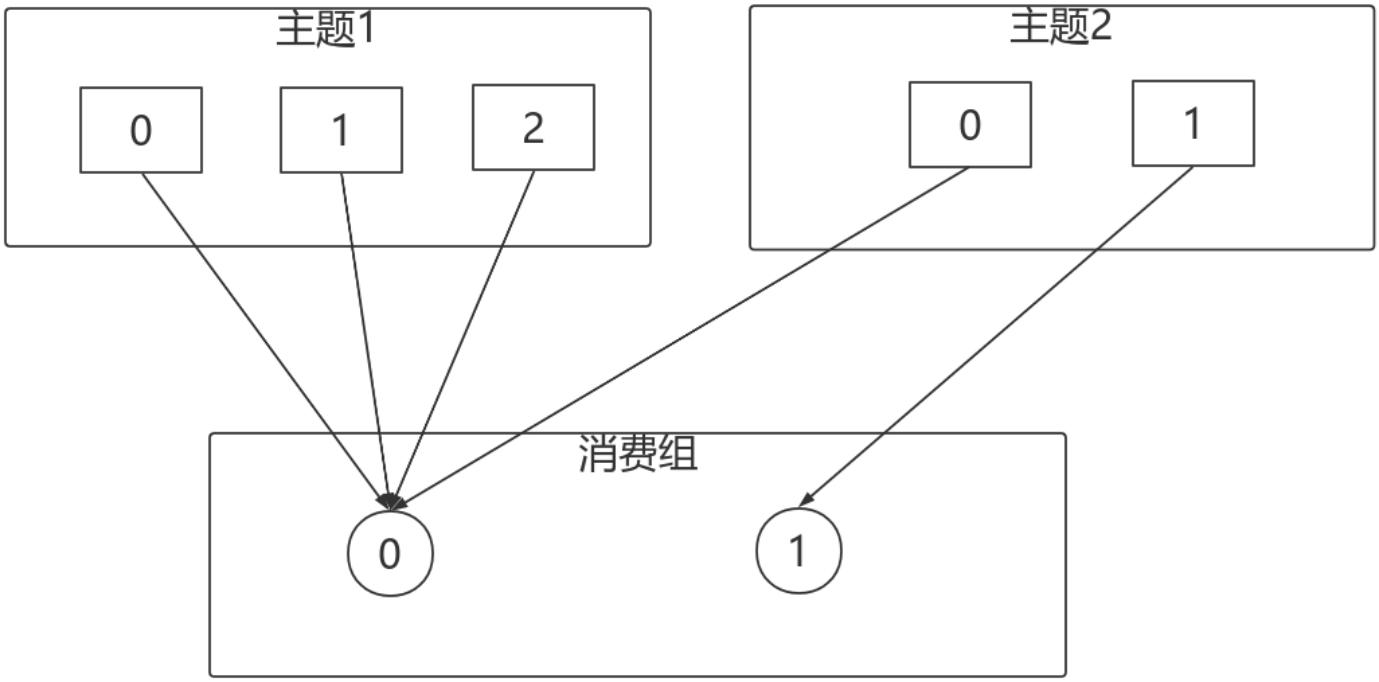
2.4.6.2 RoundRobinAssignor

RoundRobinAssignor的分配策略是将消费组内订阅的所有Topic的分区及所有消费者进行排序后尽量均衡的分配（RangeAssignor是针对单个Topic的分区进行排序分配的）。如果消费组内，消费者订阅的Topic列表是相同的（每个消费者都订阅了相同的Topic），那么分配结果是尽量均衡的（消费者之间分配到的分区数的差值不会超过1）。如果订阅的Topic列表是不同的，那么分配结果是不保证“尽量均衡”的，因为某些消费者不参与一些Topic的分配。



相对于RangeAssignor，在订阅多个Topic的情况下，RoundRobinAssignor的方式能消费者之间尽量均衡的分配到分区（分配到的分区数的差值不会超过1——RangeAssignor的分配策略可能随着订阅的Topic越来越多，差值越来越大）。

对于消费组内消费者订阅Topic不一致的情况：假设有两个个消费者分别为C0和C1，有2个Topic T1、T2，分别拥有3和2个分区，并且C0订阅T1和T2，C1订阅T2，那么RoundRobinAssignor的分配结果如下：



看上去分配已经尽量的保证均衡了，不过可以发现C0承担了4个分区的消费而C1订阅了T2一个分区，是不是把T2P0交给C1消费能更加的均衡呢?

2.4.6.3 StickyAssignor

动机

尽管RoundRobinAssignor已经在RangeAssignor上做了一些优化来更均衡的分配分区，但是在一些情况下依旧会产生严重的分配偏差，比如消费组中订阅的Topic列表不相同的情况下。

更核心的问题是无论是RangeAssignor，还是RoundRobinAssignor，当前的分区分配算法都没有考虑**上一次的分配结果**。显然，在执行一次新的分配之前，如果能考虑到上一次分配的结果，尽量少的调整分区分配的变动，显然是能节省很多开销的。

目标

从字面意义上看，Sticky是“粘性的”，可以理解为分配结果是带“粘性的”：

1. **分区的分配尽量的均衡**
2. **每一次重分配的结果尽量与上一次分配结果保持一致**

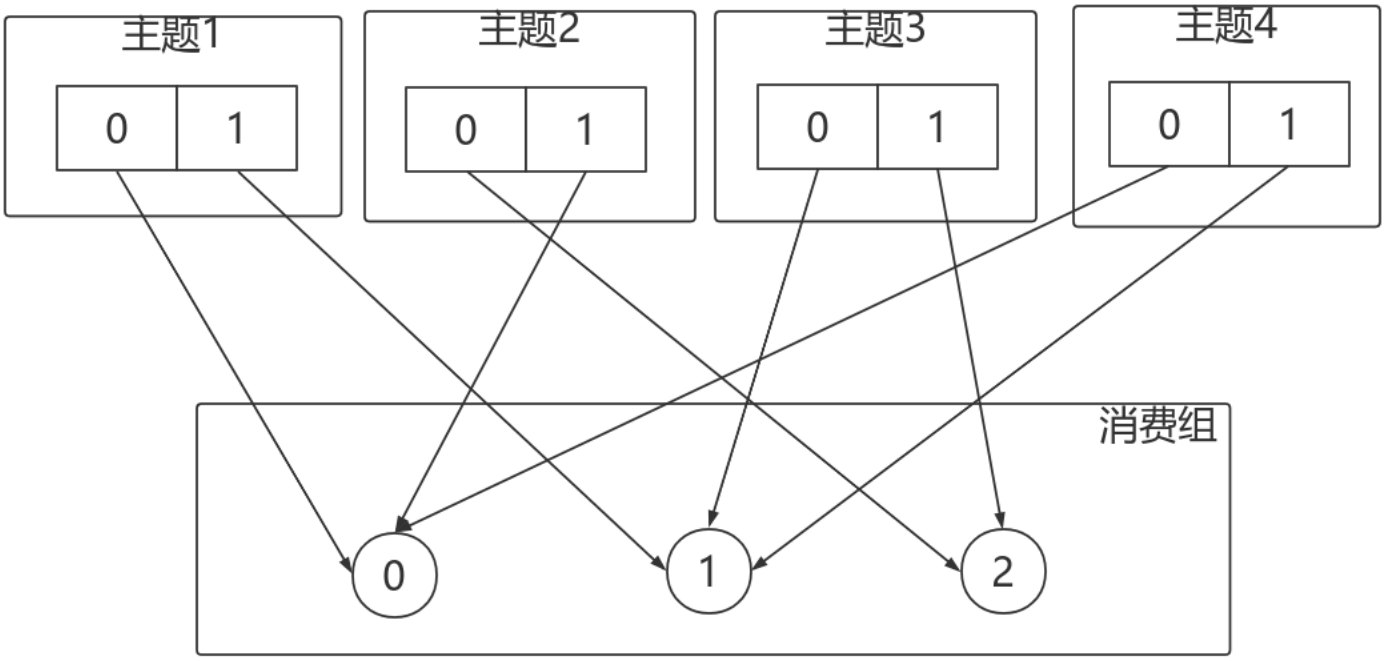
当这两个目标发生冲突时，优先保证第一个目标。第一个目标是每个分配算法都尽量尝试去完成的，而第二个目标才真正体现出StickyAssignor特性的。

我们先来看预期分配的结构，后续再具体分析StickyAssignor的算法实现。

例如：

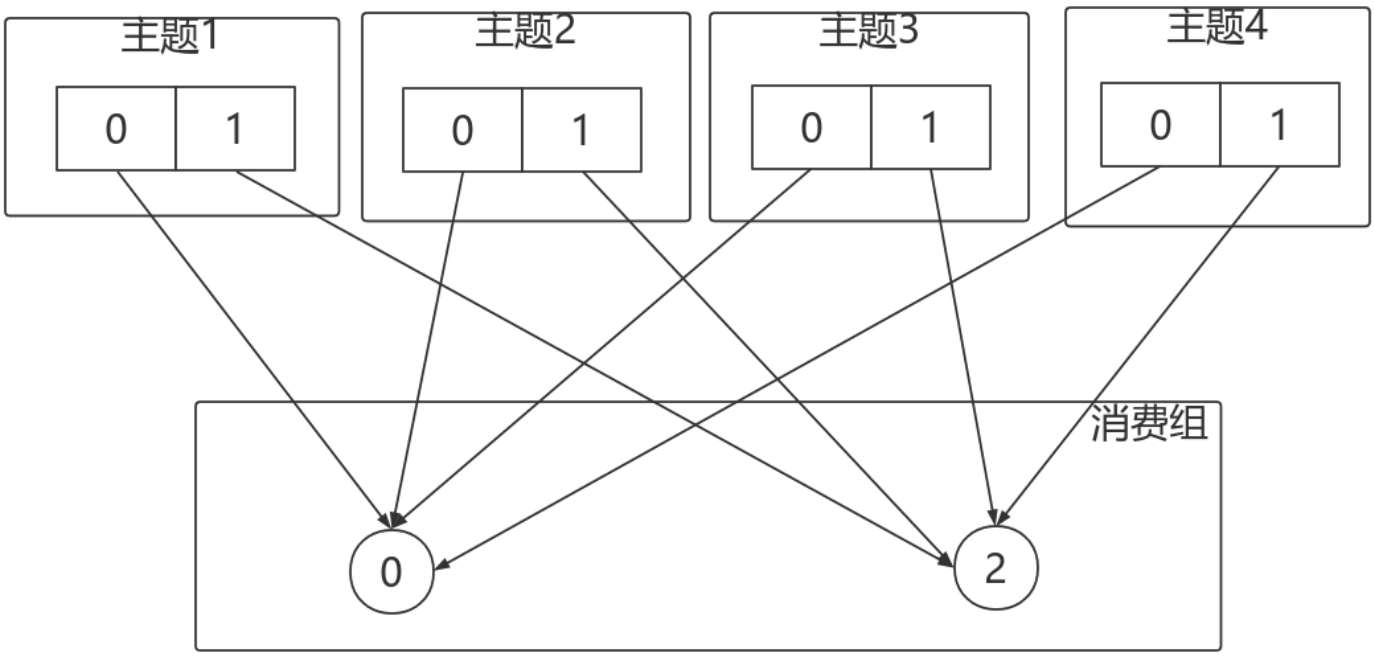
- 有3个Consumer：C0、C1、C2
- 有4个Topic：T0、T1、T2、T3，每个Topic有2个分区
- 所有Consumer都订阅了这4个分区

StickyAssignor的分配结果如下图所示（增加RoundRobinAssignor分配作为对比）：



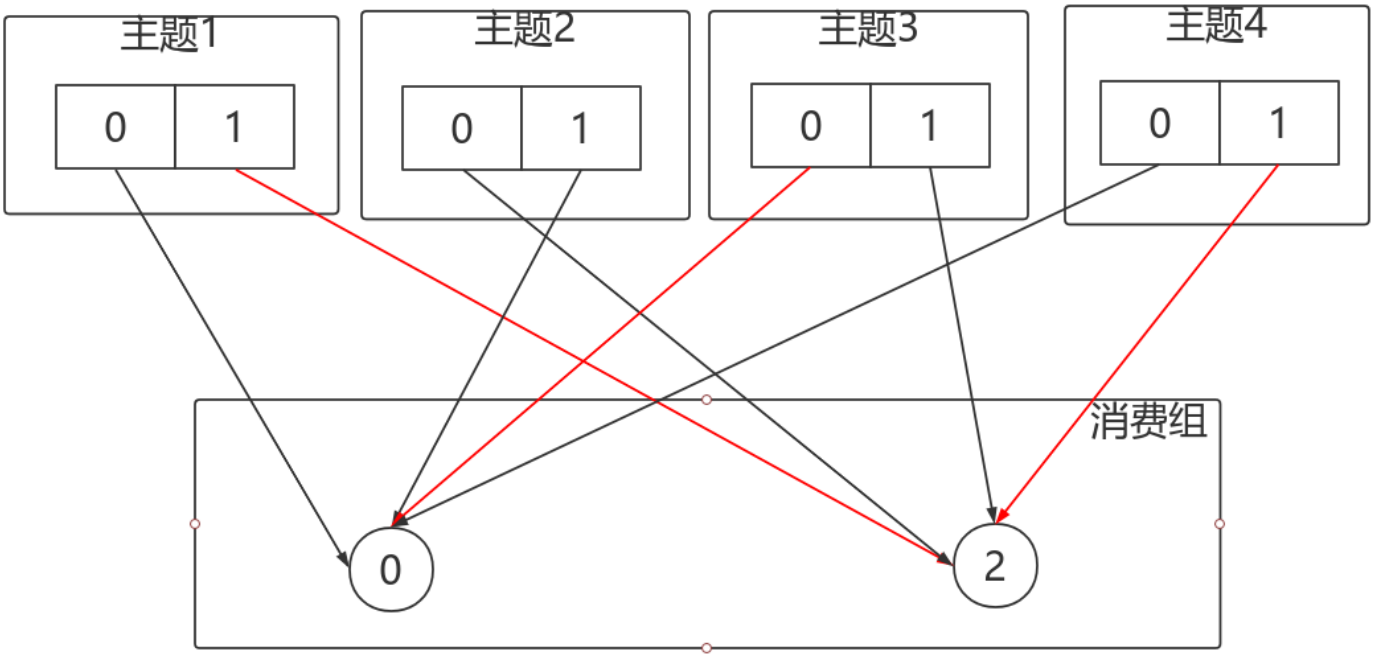
如果消费者1宕机，则按照RoundRobin的方式分配结果如下：

打乱从新来过，轮询分配：



按照Sticky的方式：

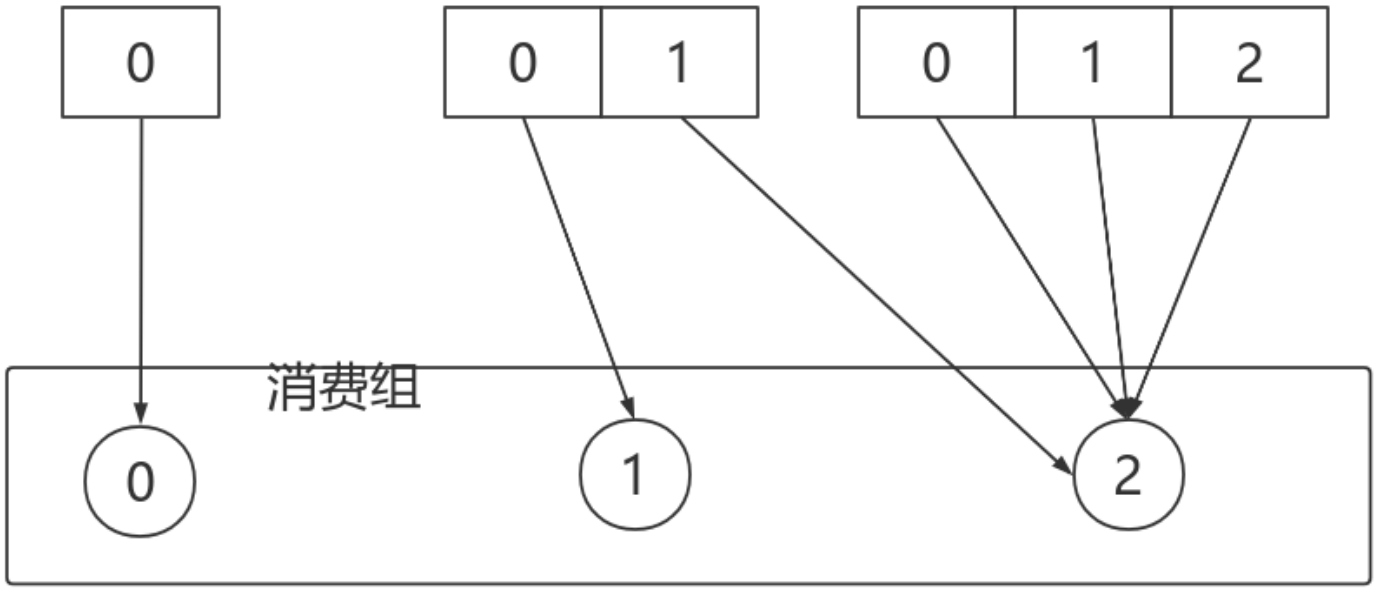
仅对消费者1分配的分区进行重分配，红线部分。最终达到均衡的目的。



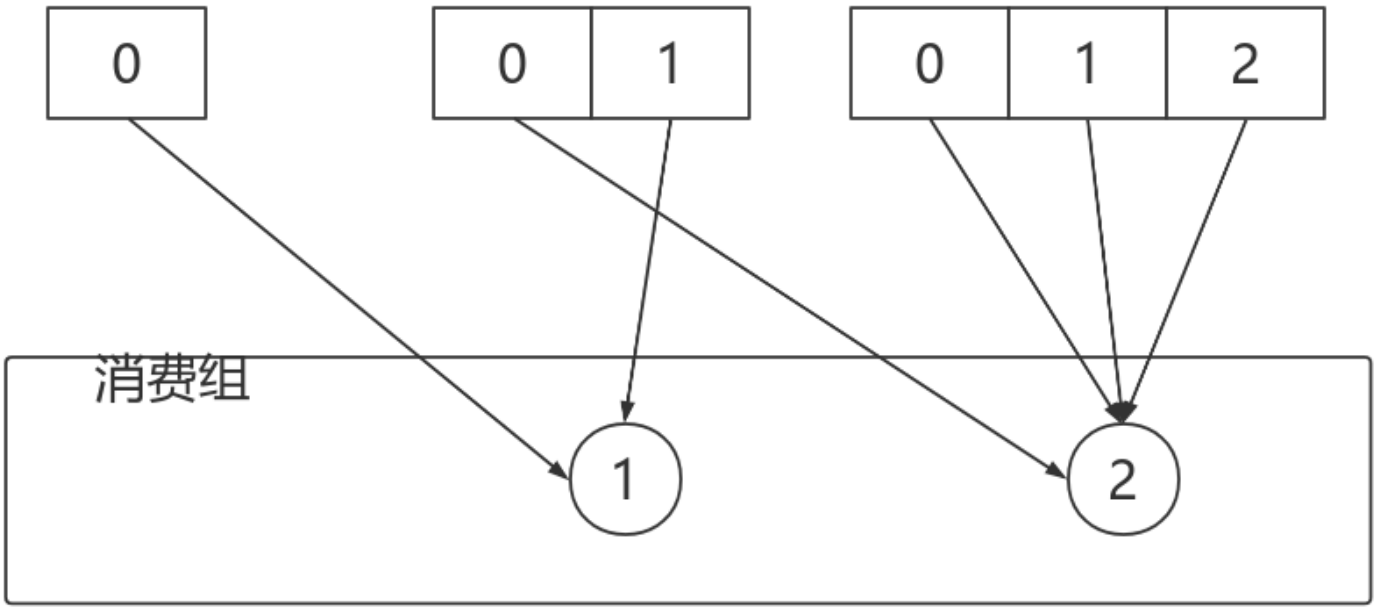
再举一个例子：

- 有3个Consumer：C0、C1、C2
- 3个Topic：T0、T1、T2，它们分别有1、2、3个分区
- C0订阅T0；C1订阅T0、T1；C2订阅T0、T1、T2

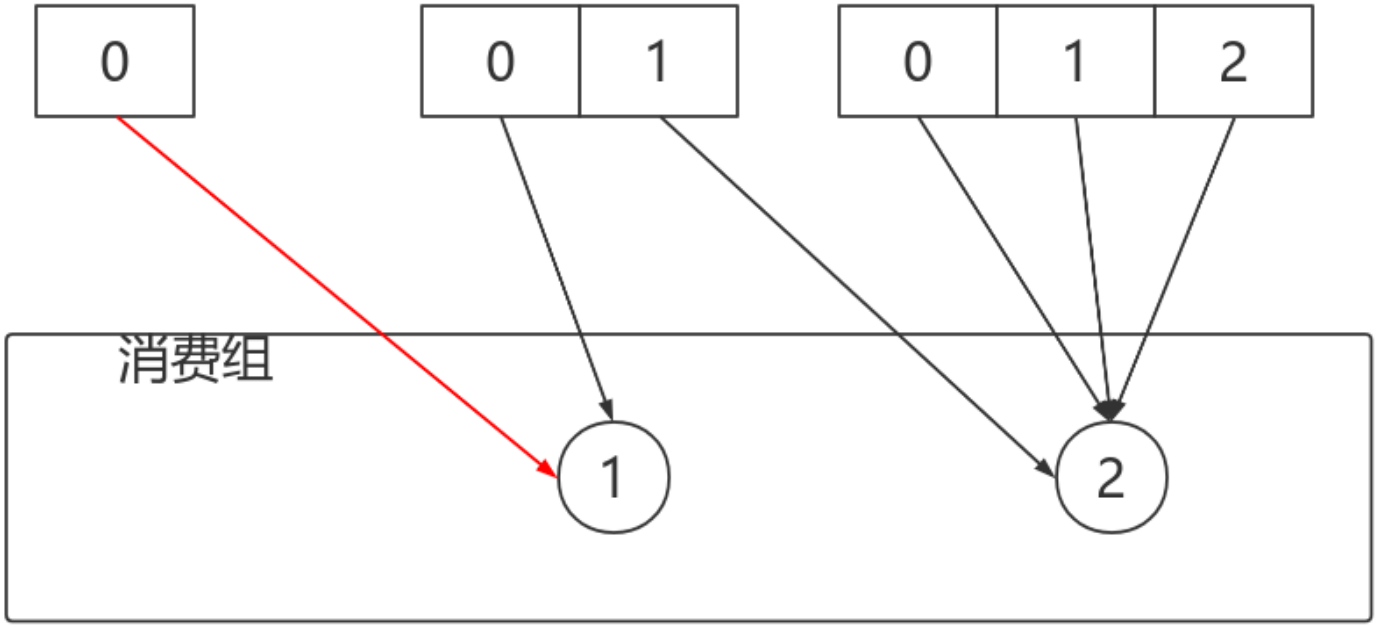
分配结果如下图所示：



消费者0下线，则按照轮询的方式分配：



按照Sticky方式分配分区，仅仅需要动的就是红线部分，其他部分不动。



StickyAssignor分配方式的实现稍微复杂点儿，我们可以先理解图示部分即可。感兴趣的同学可以研究一下。

2.4.6.4 自定义分配策略

自定义的分配策略必须要实现org.apache.kafka.clients.consumer.internals.PartitionAssignor接口。PartitionAssignor接口的定义如下：

```
Subscription subscription(Set<String> topics);

String name();

Map<String, Assignment> assign(Cluster metadata, Map<String, Subscription>
subscriptions);

void onAssignment(Assignment assignment);

class Subscription {
    private final List<String> topics;
    private final ByteBuffer userData;

}

class Assignment {
    private final List<TopicPartition> partitions;
    private final ByteBuffer userData;

}
```

PartitionAssignor接口中定义了两个内部类：Subscription和Assignment。

Subscription类用来表示消费者的订阅信息，类中有两个属性：topics和userData，分别表示消费者所订阅topic列表和用户自定义信息。PartitionAssignor接口通过subscription()方法来设置消费者自身相关的Subscription信息，注意到此方法中只有一个参数topics，与Subscription类中的topics的相互呼应，但是并没有有关userData的参数体现。为了增强用户对分配结果的控制，可以在subscription()方法内部添加一些影响分配的用户自定义信息赋予userData，比如：权重、ip地址、host或者机架（rack）等等。

再来说一下Assignment类，它是用来表示分配结果信息的，类中也有两个属性：partitions和userData，分别表示所分配到的分区集合和用户自定义的数据。可以通过PartitionAssignor接口中的onAssignment()方法是在每个消费者收到消费组leader分配结果时的回调函数，例如在StickyAssignor策略中就是通过这个方法保存当前的分配方案，以备在下次消费组再平衡（rebalance）时可以提供分配参考依据。

接口中的name()方法用来提供分配策略的名称，对于Kafka提供的3种分配策略而言，RangeAssignor对应的protocol_name为“range”，RoundRobinAssignor对应的protocol_name为“roundrobin”，StickyAssignor对应的protocol_name为“sticky”，所以自定义的分配策略中要注意命名的时候不要与已存在的分配策略发生冲突。这个命名用来标识分配策略的名称，在后面所描述的加入消费组以及选举消费组leader的时候会有涉及。

真正的分区分配方案的实现是在assign()方法中，方法中的参数metadata表示集群的元数据信息，而subscriptions表示消费组内各个消费者成员的订阅信息，最终方法返回各个消费者的分配信息。

Kafka中还提供了一个抽象类org.apache.kafka.clients.consumer.internals.AbstractPartitionAssignor，它可以简化PartitionAssignor接口的实现，对assign()方法进行了实现，其中会将Subscription中的userData信息去掉后，在进行分配。Kafka提供的3种分配策略都是继承自这个抽象类。如果开发人员在自定义分区分配策略时需要使用userData信息来控制分区分配的结果，那么就不能直接继承AbstractPartitionAssignor这个抽象类，而需要直接实现PartitionAssignor接口。

```
package org.apache.kafka.clients.consumer;

import org.apache.kafka.clients.consumer.internals.AbstractPartitionAssignor;
import org.apache.kafka.common.TopicPartition;
import java.util.*;

public class MyAssignor extends AbstractPartitionAssignor {

}
```

在使用时，消费者客户端需要添加相应的Properties参数，示例如下：

```
properties.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
MyAssignor.class.getName());
```

2.5 物理存储

2.5.1 日志存储概述

Kafka 消息是以主题为单位进行归类，各个主题之间是彼此独立的，互不影响。

每个主题又可以分为一个或多个分区。

每个分区各自存在一个记录消息数据的日志文件。

```
[root@node1 kafka-logs]# ls
cleaner-offset-checkpoint    recovery-point-offset-checkpoint  tp_demo_01-1  tp_demo_01-4
log-start-offset-checkpoint  replication-offset-checkpoint     tp_demo_01-2  tp_demo_01-5
meta.properties              tp_demo_01-0                      tp_demo_01-3
[root@node1 kafka-logs]# pwd
/var/lagou/kafka/kafka-logs
[root@node1 kafka-logs]#
```

图中，创建了一个 `tp_demo_01` 主题，其存在6个 Parition，对应的每个Parition下存在一个 `[Topic-Parition]` 命名的消息日志文件。在理想情况下，数据流量分摊到各个 Parition 中，实现了负载均衡的效果。在分区日志文件中，你会发现很多类型的文件，比如：`.index、.timestamp、.log、.snapshot` 等。

其中，文件名一致的文件集合就称为 LogSement。

```
[root@node1 kafka-logs]# ls tp_demo_01-0
00000000000000000000.index  00000000000000000000.log  00000000000000000000.timeindex  leader-epoch-checkpoint
[root@node1 kafka-logs]# ls tp_demo_01-1
00000000000000000000.index  00000000000000000000.log  00000000000000000000.timeindex  leader-epoch-checkpoint
[root@node1 kafka-logs]# ls tp_demo_01-3
00000000000000000000.index  00000000000000000000.log  00000000000000000000.timeindex  leader-epoch-checkpoint
[root@node1 kafka-logs]#
```

LogSegment

1. 分区日志文件中包含很多的 LogSegment
2. Kafka 日志追加是顺序写入的
3. LogSegment 可以减小日志文件的大小
4. 进行日志删除的时候和数据查找的时候可以快速定位。
5. ActiveLogSegment 是活跃的日志分段，拥有文件拥有写入权限，其余的 LogSegment 只有只读的权限。

日志文件存在多种后缀文件，重点需要关注 `.index、.timestamp、.log` 三种类型。

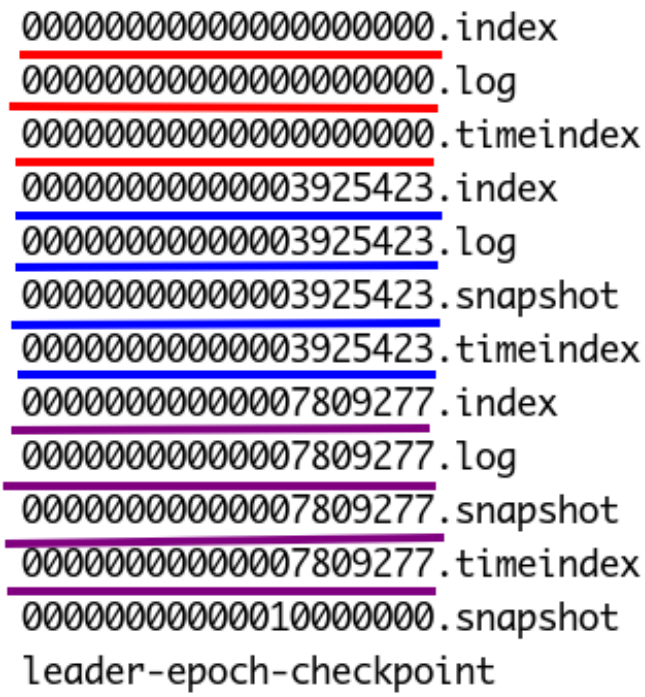
类别作用

后缀名	说明
.index	偏移量索引文件
.timestamp	时间戳索引文件
.log	日志文件
.snapshot	快照文件
.deleted	
.cleaned	日志清理时临时文件
.swap	日志压缩之后的临时文件
leader-epoch-checkpoint	

每个 LogSegment 都有一个基准偏移量，表示当前 LogSegment 中**第一条消息的 offset**。

偏移量是一个 64 位的长整形数，固定是20位数字，长度未达到，用 0 进行填补，索引文件和日志文件都由该作为文件名命名规则（00000000000000000000.index、00000000000000000000.timestamp、00000000000000000000.log）。

如果日志文件名为 `00000000000000000121.log`，则当前日志文件的一条数据偏移量就是 121（偏移量从 0 开始）。



日志与索引文件

配置条目	默认值	说明
log.index.interval.bytes	4096(4K)	增加索引项字节间隔密度，会影响索引文件中的区间密度和查询效率
log.segment.bytes	1073741824(1G)	日志文件最大值
log.roll.ms		当前日志分段中消息的最大时间戳与当前系统的时间戳的差值允许的最大范围，单位毫秒
log.roll.hours	168(7天)	当前日志分段中消息的最大时间戳与当前系统的时间戳的差值允许的最大范围，单位小时
log.index.size.max.bytes	10485760(10MB)	触发偏移量索引文件或时间戳索引文件分段字节限额

配置项默认值说明

偏移量索引文件用于记录消息偏移量与物理地址之间的映射关系。

时间戳索引文件则根据时间戳查找对应的偏移量。

Kafka 中的索引文件是以稀疏索引的方式构造消息的索引，并不保证每一个消息在索引文件中都有对应的索引项。

每当写入一定量的消息时，偏移量索引文件和时间戳索引文件分别增加一个偏移量索引项和时间戳索引项。

通过修改 `log.index.interval.bytes` 的值，改变索引项的密度。

切分文件

当满足如下几个条件中的其中之一，就会触发文件的切分：

1. 当前日志分段文件的大小超过了 broker 端参数 `log.segment.bytes` 配置的值。`log.segment.bytes` 参数的默认值为 1073741824，即 1GB。
2. 当前日志分段中消息的最大时间戳与当前系统的时间戳的差值大于 `log.roll.ms` 或 `log.roll.hours` 参数配置的值。如果同时配置了 `log.roll.ms` 和 `log.roll.hours` 参数，那么 `log.roll.ms` 的优先级高。默认情况下，只配置了 `log.roll.hours` 参数，其值为168，即 7 天。
3. 偏移量索引文件或时间戳索引文件的大小达到 broker 端参数 `log.index.size.max.bytes` 配置的值。`log.index.size.max.bytes` 的默认值为 10485760，即 10MB。
4. 追加的消息的偏移量与当前日志分段的偏移量之间的差值大于 `Integer.MAX_VALUE`，即要追加的消息的偏移量不能转变为相对偏移量。

为什么是 `Integer.MAX_VALUE` ？

1024 * 1024 * 1024=1073741824

在偏移量索引文件中，每个索引项共占用 8 个字节，并分为两部分。

相对偏移量和物理地址。

相对偏移量：表示消息相对与基准偏移量的偏移量，占 4 个字节

物理地址：消息在日志分段文件中对应的物理位置，也占 4 个字节

4 个字节刚好对应 `Integer.MAX_VALUE` ，如果大于 `Integer.MAX_VALUE` ，则不能用 4 个字节进行表示了。

索引文件切分过程

索引文件会根据 `log.index.size.max.bytes` 值进行**预先分配空间**，即文件创建的时候就是最大值

当真正的进行索引文件切分的时候，才会将其裁剪到实际数据大小的文件。

这一点是跟日志文件有所区别的地方。其意义**降低了代码逻辑的复杂性**。

2.5.2 日志存储

2.5.2.1 索引

偏移量索引文件用于记录**消息偏移量与物理地址**之间的映射关系。时间戳索引文件则根据**时间戳查找对应的偏移量**。

文件：

查看一个topic分区目录下的内容，发现有log、index和timeindex三个文件：

1. log文件名是以文件中第一条message的offset来命名的，实际offset长度是64位，但是这里只使用了20位，应付生产是足够的。
2. 一组index+log+timeindex文件的名字是一样的，并且log文件默认写满1G后，会进行log rolling形成一个新的组合来记录消息，这个是通过broker端 `log.segment.bytes` =1073741824指定的。
3. index和timeindex在刚使用时会分配10M的大小，当进行 `log rolling` 后，它会修剪为实际的大小。

```
[root@node1 tp_demo_05-0]# ll
总用量 262888
-rw-r--r-- 1 root root     51256 8月    4 11:57 00000000000000000000.index
-rw-r--r-- 1 root root 104856093 8月    4 11:57 00000000000000000000.log
-rw-r--r-- 1 root root     74544 8月    4 11:57 00000000000000000000.timeindex
-rw-r--r-- 1 root root     51264 8月    4 11:57 00000000000003925423.index
-rw-r--r-- 1 root root 104844831 8月    4 11:57 00000000000003925423.log
-rw-r--r-- 1 root root        10 8月    4 11:57 00000000000003925423.snapshot
-rw-r--r-- 1 root root     75336 8月    4 11:57 00000000000003925423.timeindex
-rw-r--r-- 1 root root  10485760 8月    4 11:57 00000000000007809277.index
-rw-r--r-- 1 root root  59138705 8月    4 11:57 00000000000007809277.log
-rw-r--r-- 1 root root        10 8月    4 11:57 00000000000007809277.snapshot
-rw-r--r-- 1 root root  10485756 8月    4 11:57 00000000000007809277.timeindex
-rw-r--r-- 1 root root         8 8月    4 11:57 leader-epoch-checkpoint
[root@node1 tp_demo_05-0]# pwd
/var/lagou/kafka/kafka-logs/tp_demo_05-0
[root@node1 tp_demo_05-0]#
```

1、创建主题：

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --create --topic tp_demo_05 --partitions 1 --replication-factor 1 --config segment.bytes=104857600
```

2、创建消息文件：

```
[root@node1 ~]# for i in `seq 10000000`; do echo "hello lagou $i" >> nmm.txt; done
```

```
[root@node1 ~]# ll -h nmm.txt
-rw-r--r-- 1 root root 190M 8月    4 11:46 nmm.txt
```

3、将文本消息生产到主题中：

```
[root@node1 ~]# kafka-console-producer.sh --broker-list node1:9092 --topic tp_demo_05 <nmm.txt
```

4、查看存储文件：

```
[root@node1 tp_demo_05-0]# ll
总用量 262888
-rw-r--r-- 1 root root     51256 8月    4 11:57 00000000000000000000.index
-rw-r--r-- 1 root root 104856093 8月    4 11:57 00000000000000000000.log
-rw-r--r-- 1 root root     74544 8月    4 11:57 00000000000000000000.timeindex
-rw-r--r-- 1 root root     51264 8月    4 11:57 00000000000003925423.index
-rw-r--r-- 1 root root 104844831 8月    4 11:57 00000000000003925423.log
-rw-r--r-- 1 root root        10 8月    4 11:57 00000000000003925423.snapshot
-rw-r--r-- 1 root root     75336 8月    4 11:57 00000000000003925423.timeindex
-rw-r--r-- 1 root root  10485760 8月    4 11:57 00000000000007809277.index
-rw-r--r-- 1 root root  59138705 8月    4 11:57 00000000000007809277.log
-rw-r--r-- 1 root root        10 8月    4 11:57 00000000000007809277.snapshot
-rw-r--r-- 1 root root  10485756 8月    4 11:57 00000000000007809277.timeindex
-rw-r--r-- 1 root root         8 8月    4 11:57 leader-epoch-checkpoint
[root@node1 tp_demo_05-0]# pwd
/var/lagou/kafka/kafka-logs/tp_demo_05-0
[root@node1 tp_demo_05-0]#
```

如果想查看这些文件，可以使用kafka提供的shell来完成，几个关键信息如下：

（1）offset是逐渐增加的整数，每个offset对应一个消息的偏移量。

（2）position：消息批字节数，用于计算物理地址。

（3）CreateTime：时间戳。

（4）magic：2代表这个消息类型是V2，如果是0则代表是V0类型，1代表V1类型。

（5）compresscodec：None说明没有指定压缩类型，kafka目前提供了4种可选择，0-None、1-GZIP、2-snappy、3-lz4。

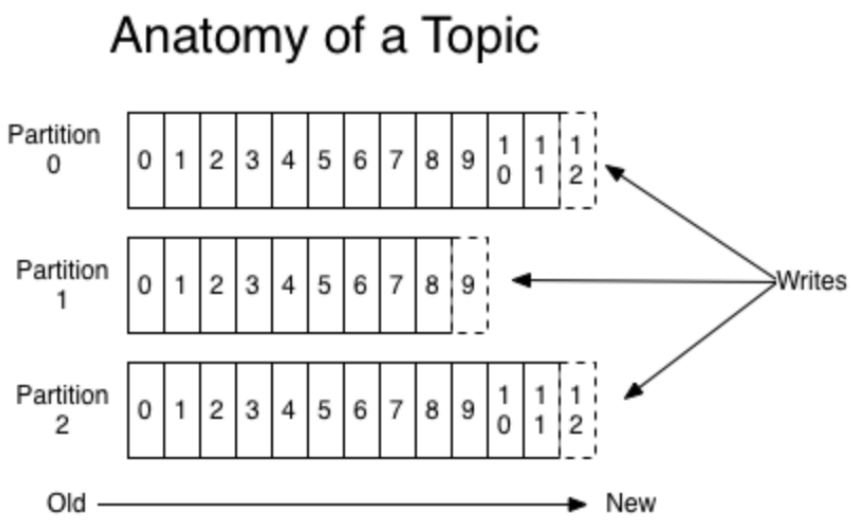
（6）crc：对所有字段进行校验后的crc值。

```
[root@node1 tp_demo_05-0]# kafka-run-class.sh kafka.tools.DumpLogSegments --files 00000000000000000000.log  --print-data-log | head
Dumping 00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 716 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 0 CreateTime: 1596513421661 isvalid: true size: 16380 magic: 2 compresscodec: NONE crc: 2973274901
baseOffset: 717 lastOffset: 1410 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 16380 CreateTime: 1596513421715 isvalid: true size: 16371 magic: 2 compresscodec: NONE crc: 1439993110
baseOffset: 1411 lastOffset: 2092 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 32751 CreateTime: 1596513421747 isvalid: true size: 16365 magic: 2 compresscodec: NONE crc: 3528903590
baseOffset: 2093 lastOffset: 2774 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 49116 CreateTime: 1596513421791 isvalid: true size: 16365 magic: 2 compresscodec: NONE crc: 763876977
baseOffset: 2775 lastOffset: 3456 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 65481 CreateTime: 1596513421795 isvalid: true size: 16365 magic: 2 compresscodec: NONE crc: 2218198476
baseOffset: 3457 lastOffset: 4138 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 81846 CreateTime: 1596513421798 isvalid: true size: 16365 magic: 2 compresscodec: NONE crc: 4018065070
baseOffset: 4139 lastOffset: 4820 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 98211 CreateTime: 1596513421802 isvalid: true size: 16365 magic: 2 compresscodec: NONE crc: 3073882858
baseOffset: 4821 lastOffset: 5502 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 114576 CreateTime: 1596513421819 isvalid: true size: 16365 magic: 2 compresscodec: NONE crc: 207330377
[root@node1 tp_demo_05-0]#
```

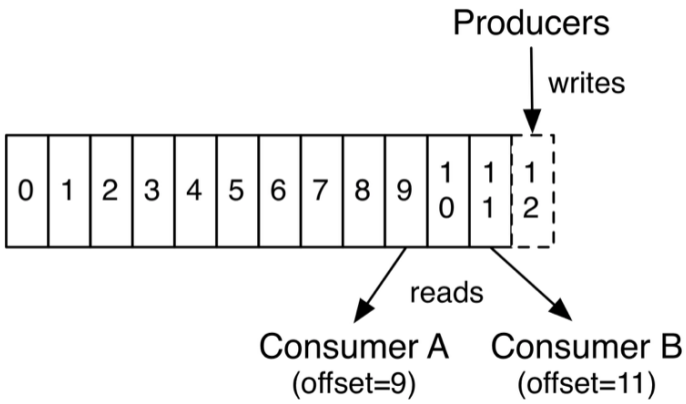
关于消息偏移量：

一、消息存储

1. 消息内容保存在log日志文件中。
2. 消息封装为Record，追加到log日志文件末尾，采用的是**顺序写**模式。
3. 一个topic的不同分区，可认为是queue，顺序写入接收到的消息。



消费者有offset。下图中，消费者A消费的offset是9，消费者B消费的offset是11，不同的消费者offset是交给一个内部公共topic来记录的。



（3）时间戳索引文件，它的作用是可以让用户查询某个时间段内的消息，它一条数据的结构是时间戳（8byte）+相对offset（4byte），如果要使用这个索引文件，首先需要通过时间范围，找到对应的相对offset，然后再去对应的index文件找到position信息，然后才能遍历log文件，它也是需要使用上面说的index文件的。

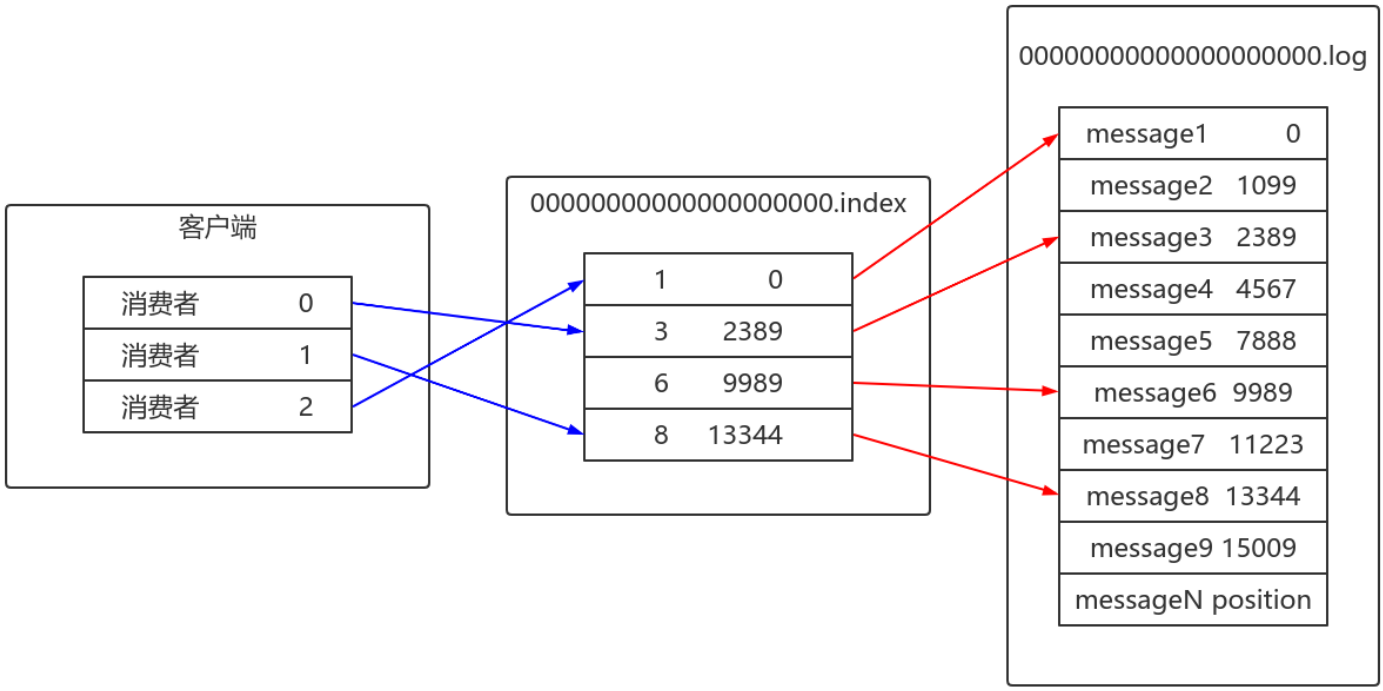
但是由于producer生产消息可以指定消息的时间戳，这可能将导致消息的时间戳不一定有先后顺序，因此**尽量不要生产消息时指定时间戳**。

2.5.2.1.1 偏移量

1. 位置索引保存在index文件中
2. log日志默认每写入4K（log.index.interval.bytes设定的），会写入一条索引信息到index文件中，因此索引文件是稀疏索引，它**不会为每条日志都建立索引信息**。
3. log文件中的日志，是顺序写入的，由message+实际offset+position组成
4. 索引文件的数据结构则是由相对offset（4byte）+position（4byte）组成，由于保存的是相对第一个消息的相对offset，只需要4byte就可以了，可以节省空间，在实际查找后还需要计算回实际的offset，这对用户是透明的。

稀疏索引，索引密度不高，但是offset有序，二分查找的时间复杂度为O(lgN)，如果从头遍历时间复杂度是O(N)。

示意图如下：



偏移量索引由相对偏移量和物理地址组成。

偏移量索引项格式

可以通过如下命令解析`.index`文件

```
kafka-run-class.sh kafka.tools.DumpLogSegments --files 00000000000000000000.index --print-data-log | head
```

注意：offset 与 position 没有直接关系，因为会删除数据和清理日志。

```
[root@node1 tp_demo_05-0]# kafka-run-class.sh kafka.tools.DumpLogSegments --files 00000000000003925423.index  --print-data-log | head
Dumping 00000000000003925423.index
offset: 3926029 position: 16359
offset: 3926635 position: 32718
offset: 3927241 position: 49077
offset: 3927847 position: 65436
offset: 3928453 position: 81795
offset: 3929059 position: 98154
offset: 3929665 position: 114513
offset: 3930271 position: 130872
offset: 3930877 position: 147231
[root@node1 tp_demo_05-0]#
```

```
[root@node1 tp_demo_05-0]# kafka-run-class.sh kafka.tools.DumpLogSegments --files 00000000000003925423.log  --print-data-log | head
Dumping 00000000000003925423.log
Starting offset: 3925423
baseOffset: 3925423 lastOffset: 3926028 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 0 CreateTime: 1596513434779 isvalid: true size: 16359 magic: 2 compresscodec: NONE crc: 4049330741
baseOffset: 3926029 lastOffset: 3926634 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 16359 CreateTime: 1596513434786 isvalid: true size: 16359 magic: 2 compresscodec: NONE crc: 2290699169
baseOffset: 3926635 lastOffset: 3927240 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 32718 CreateTime: 1596513434787 isvalid: true size: 16359 magic: 2 compresscodec: NONE crc: 368995405
baseOffset: 3927241 lastOffset: 3927846 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 49077 CreateTime: 1596513434788 isvalid: true size: 16359 magic: 2 compresscodec: NONE crc: 143415655
baseOffset: 3927847 lastOffset: 3928452 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 65436 CreateTime: 1596513434789 isvalid: true size: 16359 magic: 2 compresscodec: NONE crc: 572340120
baseOffset: 3928453 lastOffset: 3929058 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 81795 CreateTime: 1596513434790 isvalid: true size: 16359 magic: 2 compresscodec: NONE crc: 1029643347
baseOffset: 3929059 lastOffset: 3929664 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 98154 CreateTime: 1596513434791 isvalid: true size: 16359 magic: 2 compresscodec: NONE crc: 2163818250
baseOffset: 3929665 lastOffset: 3930270 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false position: 114513 CreateTime: 1596513434792 isvalid: true size: 16359 magic: 2 compresscodec: NONE crc: 3747213735
[root@node1 tp_demo_05-0]#
```

在偏移量索引文件中，索引数据都是顺序记录 offset，但时间戳索引文件中每个追加的索引时间戳必须大于之前追加的索引项，否则不予追加。在 `Kafka 0.11.0.0` 以后，消息元数据中存在若干的时间戳信息。如果 broker 端参数 `log.message.timestamp.type` 设置为 LogAppendTIme，那么时间戳必定能保持单调增长。反之如果是 CreateTime 则无法保证顺序。

注意：timestamp文件中的 offset 与 index 文件中的 relativeOffset 不是一一对应的。因为数据的写入是各自追加。

思考：如何查看偏移量为23的消息?

Kafka 中存在一个 `ConcurrentSkipListMap` 来保存在每个日志分段，通过跳跃表方式，定位到在 `00000000000000000000.index` ，通过二分法在偏移量索引文件中找到不大于 23 的**最大索引项**，即 offset 20 那栏，然后从日志分段文件中的物理位置为320 开始顺序查找偏移量为 23 的消息。

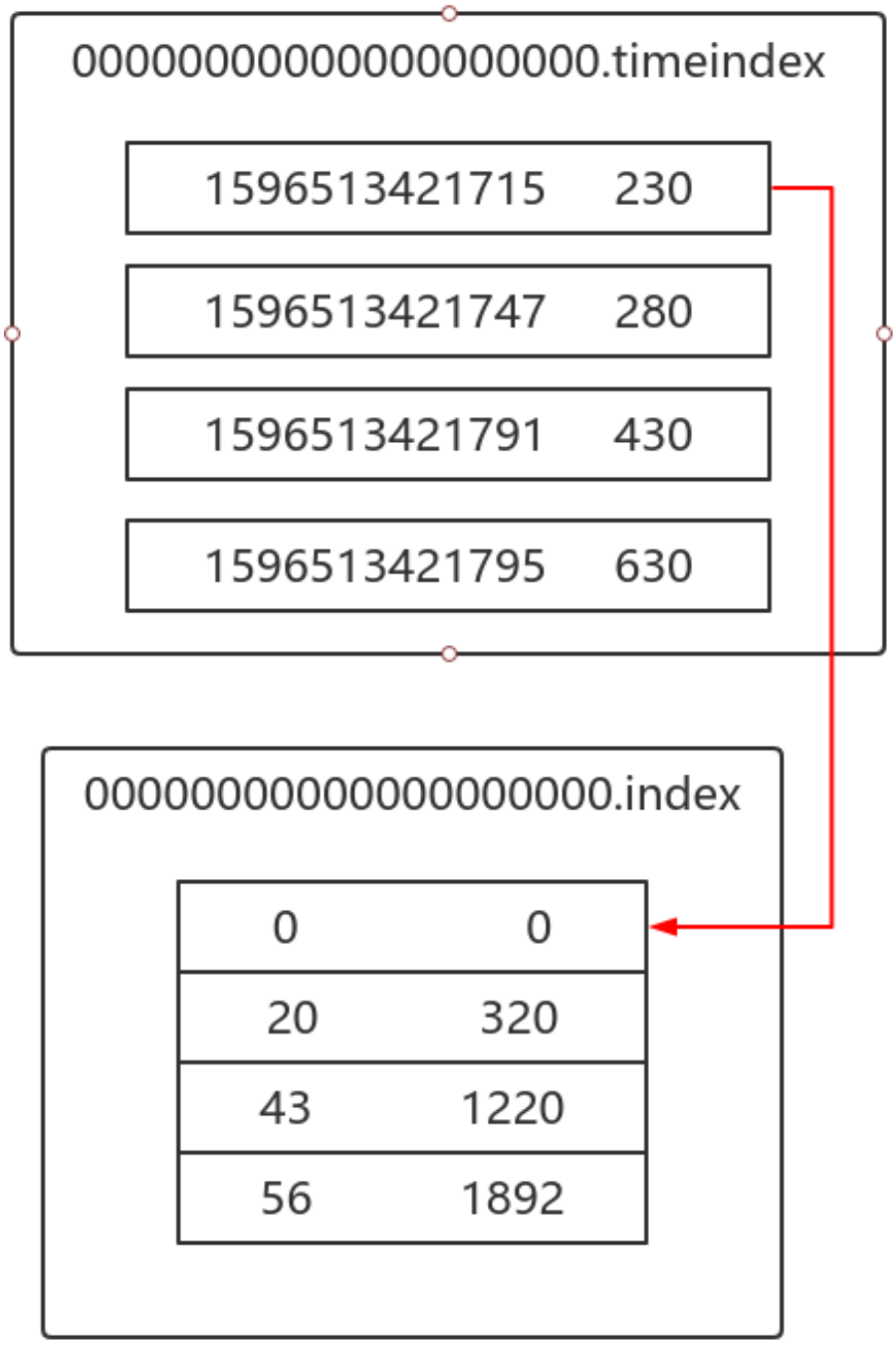
2.5.2.1.2 时间戳

在偏移量索引文件中，索引数据都是顺序记录 offset ，但时间戳索引文件中每个追加的索引时间戳必须大于之前追加的索引项，否则不予追加。在 `Kafka 0.11.0.0` 以后，消息信息中存在若干的时间戳信息。如果 broker 端参数 `log.message.timestamp.type` 设置为 LogAppendTIme ，那么时间戳必定能保持单调增长。反之如果是 CreateTime 则无法保证顺序。

通过时间戳方式进行查找消息，需要通过查找时间戳索引和偏移量索引两个文件。

时间戳索引索引格式：前八个字节表示**时间戳**，后四个字节表示**偏移量**。

时间戳索引项格式



思考：查找时间戳为 1557554753430 开始的消息？

1. 查找该时间戳应该在哪个日志分段中。将1557554753430和每个日志分段中最大时间戳largestTimeStamp逐一对比，直到找到不小于1557554753430所对应的日志分段。日志分段中的largestTimeStamp的计算是：先查询该日志分段所对应**时间戳索引文件**，找到**最后一条索引项**，若最后一条索引项的时间戳字段值大于0，则取该值，否则取该日志分段的最近修改时间。
2. 查找该日志分段的偏移量索引文件，查找该偏移量对应的物理地址。
3. 日志文件中从 320 的物理位置开始查找不小于 1557554753430 数据。

注意：timestamp文件中的 offset 与 index 文件中的 relativeOffset 不是一一对应的，因为数据的写入是各自追加。

2.5.2.2 清理

Kafka 提供两种日志清理策略：

日志删除：按照一定的删除策略，将不满足条件的数据进行数据删除

日志压缩：针对每个消息的 Key 进行整合，对于有相同 Key 的不同 Value 值，只保留最后一个版本。

Kafka 提供 `log.cleanup.policy` 参数进行相应配置，默认值：`delete`，还可以选择 `compact`。

主题级别的配置项是 `cleanup.policy` 。

2.5.2.2.1 日志删除

基于时间

日志删除任务会根据 `log.retention.hours/log.retention.minutes/log.retention.ms` 设定日志保留的时间节点。如果超过该设定值，就需要进行删除。默认是 7 天，`log.retention.ms` 优先级最高。

Kafka 依据日志分段中最大的时间戳进行定位。

首先要查询该日志分段所对应的时间戳索引文件，查找时间戳索引文件中最后一条索引项，若最后一条索引项的时间戳字段值大于 0，则取该值，否则取最近修改时间。

为什么不直接选最近修改时间呢?

因为日志文件可以有意无意的被修改，并不能真实的反应日志分段的最大时间信息。

删除过程

1. 从日志对象中所维护日志分段的跳跃表中移除待删除的日志分段，保证没有线程对这些日志分段进行读取操作。
2. 这些日志分段所有文件添加 上 `.delete` 后缀。
3. 交由一个以 `"delete-file"` 命名的延迟任务来删除这些 `.delete` 为后缀的文件。延迟执行时间可以通过 `file.delete.delay.ms` 进行设置

如果活跃的日志分段中也存在需要删除的数据时?

Kafka 会先切分出一个新的日志分段作为活跃日志分段，该日志分段不删除，删除原来的日志分段。

先腾出地方，再删除。

基于日志大小

日志删除任务会检查当前日志的大小是否超过设定值。设定项为 `log.retention.bytes` ，单个日志分段的大小由 `log.segment.bytes` 进行设定。

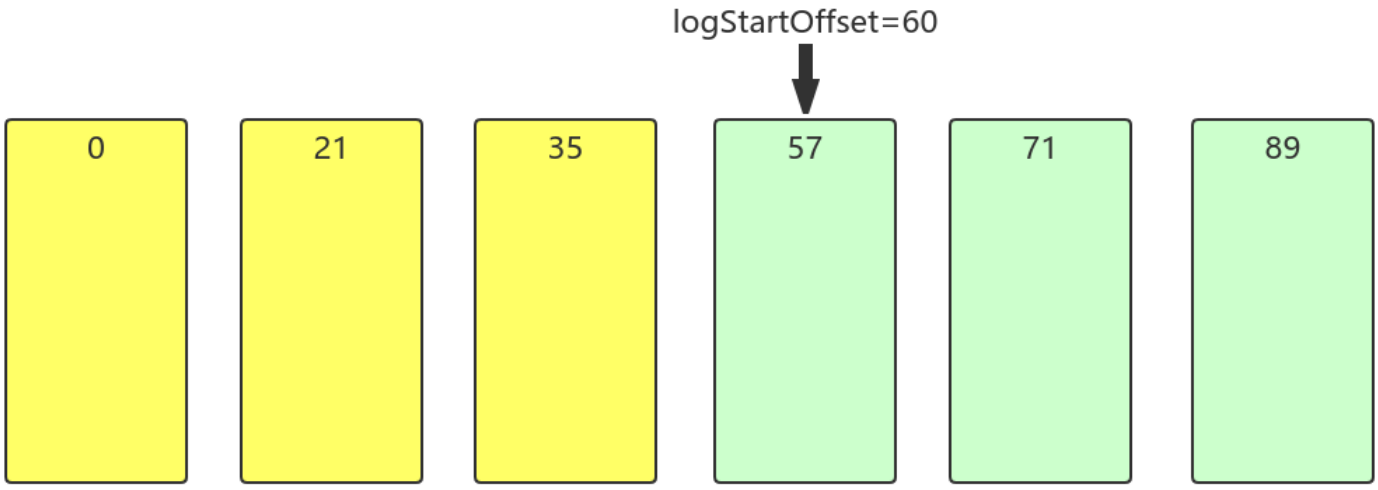
删除过程

1. 计算需要被删除的日志总大小 (当前日志文件大小（所有分段）减去retention值)。
2. 从日志文件第一个 LogSegment 开始查找可删除的日志分段的文件集合。
3. 执行删除。

基于偏移量

根据日志分段的下一个日志分段的起始偏移量是否大于等于日志文件的起始偏移量，若是，则可以删除此日志分段。

注意：日志文件的起始偏移量并不一定等于第一个日志分段的基准偏移量，存在数据删除，可能与之相等的那条数据已经被删除了。



删除过程

1. 从头开始遍历每个日志分段，日志分段1的下一个日志分段的起始偏移量为21，小于logStartOffset，将日志分段1加入到删除队列中
2. 日志分段 2 的下一个日志分段的起始偏移量为35，小于 logStartOffset，将 日志分段 2 加入到删除队列中
3. 日志分段 3 的下一个日志分段的起始偏移量为57，小于logStartOffset，将日志分段3加入删除集合中
4. 日志分段4的下一个日志分段的其实偏移量为71，大于logStartOffset，则不进行删除。

2.5.2.2.2 日志压缩策略

1. 概念

日志压缩是Kafka的一种机制，可以提供较为细粒度的记录保留，而不是基于粗粒度的基于时间的保留。

对于具有相同的Key，而数据不同，只保留最后一条数据，前面的数据在合适的情况下删除。

2. 应用场景

日志压缩特性，就实时计算来说，可以在异常容灾方面有很好的应用途径。比如，我们在Spark、Flink中做实时计算时，需要长期在内存里面维护一些数据，这些数据可能是通过聚合了一天或者一周的日志得到的，这些数据一旦由于异常因素（内存、网络、磁盘等）崩溃了，从头开始计算需要很长的时间。一个比较有效可行的方式就是定时将内存里的数据备份到外部存储介质中，当崩溃出现时，再从外部存储介质中恢复并继续计算。

使用日志压缩来替代这些外部存储有哪些优势及好处呢？这里为大家列举并总结了几点：

- Kafka即是数据源又是存储工具，可以简化技术栈，降低维护成本
- 使用外部存储介质的话，需要将存储的Key记录下来，恢复的时候再使用这些Key将数据取回，实现起来有一定的工程难度和复杂度。使用Kafka的日志压缩特性，只需要把数据写进Kafka，等异常出现恢复任务时再读回到内存就可以了
- Kafka对于磁盘的读写做了大量的优化工作，比如磁盘顺序读写。相对于外部存储介质没有索引查询等工作量的负担，可以实现高性能。同时，Kafka的日志压缩机制可以充分利用廉价的磁盘，不用依赖昂贵的内存来处理，在性能相似的情况下，实现非常高的性价比（这个观点仅仅针对于异常处理和容灾的场景来说）

2.3 日志压缩方式的实现细节

主题的`cleanup.policy`需要设置为compact。

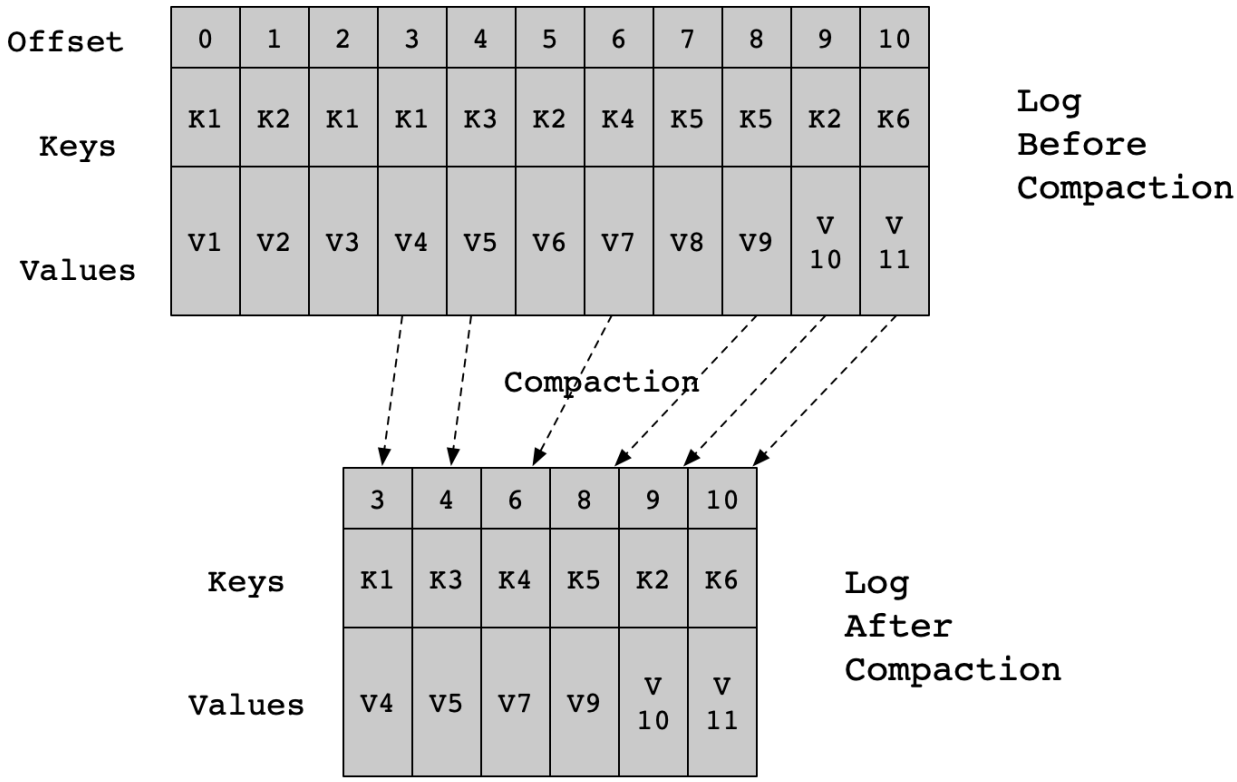
Kafka的后台线程会定时将Topic遍历两次：

1. 记录每个key的hash值最后一次出现的偏移量
2. 第二次检查每个offset对应的Key是否在后面的日志中出现过，如果出现了就删除对应的日志。

日志压缩允许删除，除最后一个key之外，删除先前出现的所有该key对应的记录。在一段时间后从日志中清理，以释放空间。

注意：日志压缩与key有关，**确保每个消息的key不为null**。

压缩是在Kafka后台通过**定时重新打开**Segment来完成的，Segment的压缩细节如下图所示：



日志压缩可以确保：

1. 任何保持在日志头部以内的使用者都将看到所写的每条消息，这些消息将具有顺序偏移量。可以使用Topic的min.compaction.lag.ms属性来保证消息在被压缩之前必须经过的最短时间。也就是说，它为每个消息在（未压缩）头部停留的时间提供了一个下限。可以使用Topic的max.compaction.lag.ms属性来保证从收到消息到消息符合压缩条件之间的最大延时

- 消息始终保持顺序，压缩永远不会重新排序消息，只是删除一些而已
- 消息的偏移量永远不会改变，它是日志中位置的永久标识符
- 从日志开始的任何使用者将至少看到所有记录的最终状态，按记录的顺序写入。另外，如果使用者在比Topic的log.cleaner.delete.retention.ms短的时间内到达日志的头部，则会看到已删除记录的所有delete标记。保留时间默认是24小时。

默认情况下，启动日志清理器，若需要启动特定Topic的日志清理，请添加特定的属性。配置日志清理器，这里为大家总结了以下几点：

1. `log.cleanup.policy`设置为`compact`，Broker的配置，影响集群中所有的Topic。
2. `log.cleaner.min.compaction.lag.ms`，用于防止对更新超过最小消息进行压缩，如果没有设置，除最后一个Segment之外，所有Segment都有资格进行压缩

- `log.cleaner.max.compaction.lag.ms`，用于防止低生产速率的日志在无限制的时间内不压缩。

Kafka的日志压缩原理并不复杂，就是定时把所有的日志读取两遍，写一遍，而CPU的速度超过磁盘完全不是问题，只要日志的量对应的读取两遍和写入一遍的时间在可接受的范围内，那么它的性能就是可以接受的。

2.5.3 磁盘存储

2.5.3.1 零拷贝

kafka高性能，是多方面协同的结果，包括宏观架构、分布式partition存储、ISR数据同步、以及“无所不用其极”的高效利用磁盘/操作系统特性。

零拷贝并不是不需要拷贝，而是减少不必要的拷贝次数。通常是说在IO读写过程中。

nginx的高性能也有零拷贝的身影。

传统IO

比如：读取文件，socket发送

传统方式实现：先读取、再发送，实际经过1~4四次copy。

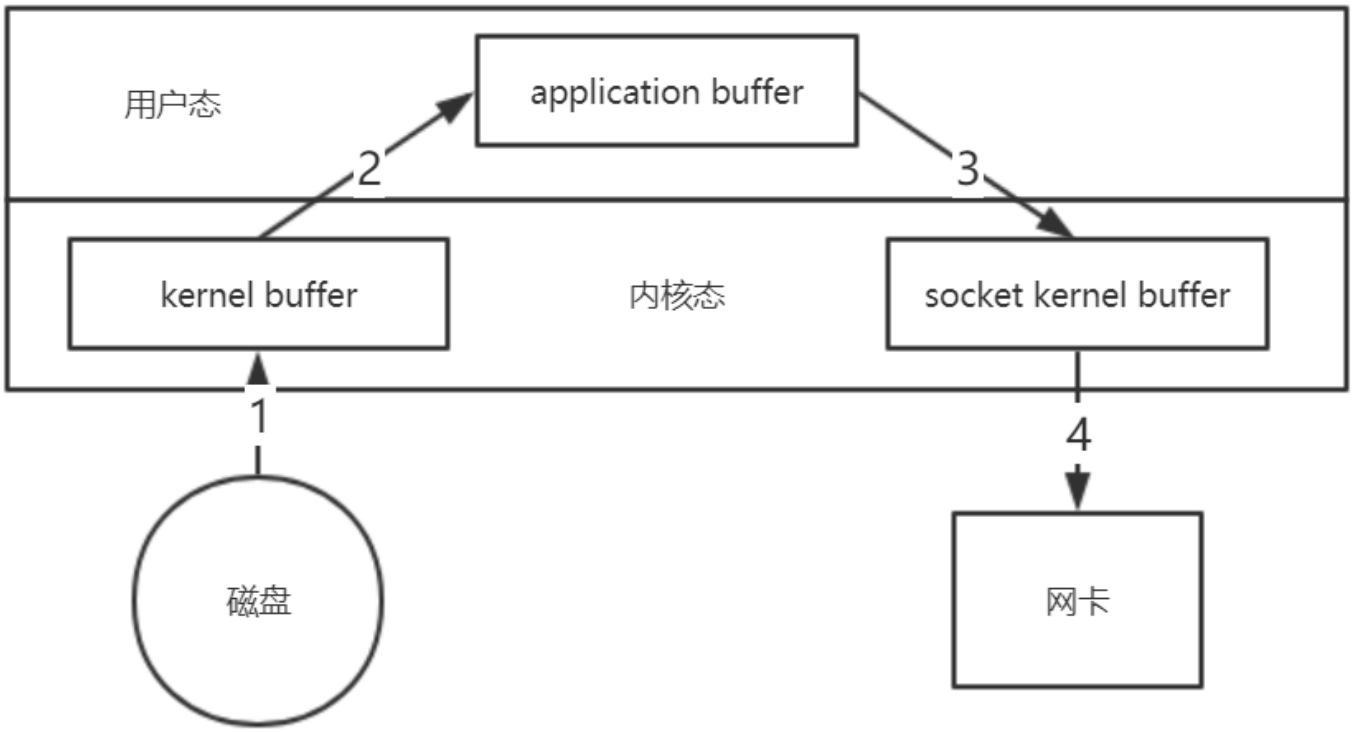
```
buffer = File.read
Socket.send(buffer)
```

1、第一次：将磁盘文件，读取到操作系统内核缓冲区；

2、第二次：将内核缓冲区的数据，copy到application应用程序的buffer；

3、第三步：将application应用程序buffer中的数据，copy到socket网络发送缓冲区(属于操作系统内核的缓冲区)；

4、第四次：将socket buffer的数据，copy到网络协议栈，由网卡进行网络传输。



实际IO读写，需要进行IO中断，需要CPU响应中断(内核态到用户态转换)，尽管引入DMA(Direct Memory Access，直接存储器访问)来接管CPU的中断请求，但四次copy是存在“不必要的拷贝”的。

实际上并不需要第二个和第三个数据副本。数据可以直接从读缓冲区传输到套接字缓冲区。

kafka的两个过程：

1、网络数据持久化到磁盘 (Producer 到 Broker)

2、磁盘文件通过网络发送（Broker 到 Consumer）

数据落盘通常都是非实时的，Kafka的数据并不是实时的写入硬盘，它充分利用了现代操作系统分页存储来利用内存提高I/O效率。

磁盘文件通过网络发送（Broker 到 Consumer）

磁盘数据通过DMA(Direct Memory Access，直接存储器访问)拷贝到内核态 Buffer

直接通过 DMA 拷贝到 NIC Buffer(socket buffer)，无需 CPU 拷贝。

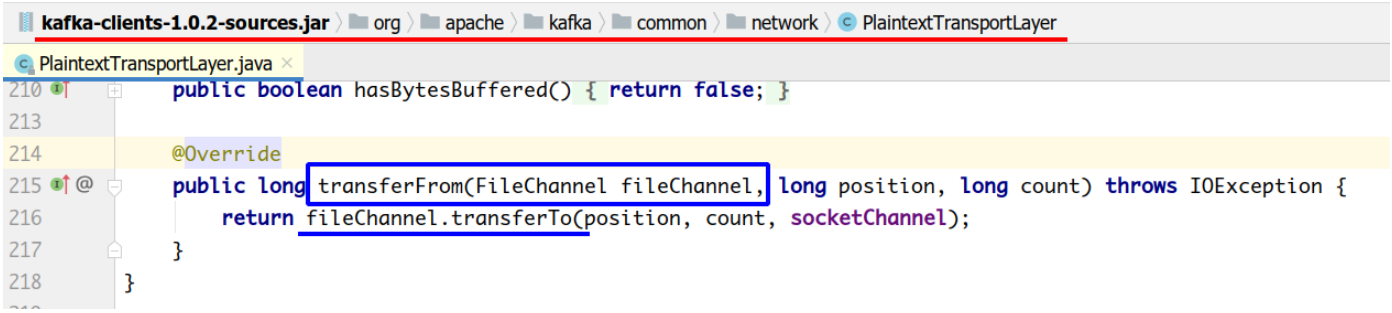
除了减少数据拷贝外，整个读文件 ==> 网络发送由一个 sendfile 调用完成，整个过程只有两次上下文切换，因此大大提高了性能。

Java NIO对sendfile的支持就是FileChannel.transferTo()/transferFrom()。

fileChannel.transferTo(position, count, socketChannel);

把磁盘文件读取OS内核缓冲区后的fileChannel，直接转给socketChannel发送；底层就是sendfile。消费者从broker读取数据，就是由此实现。

具体来看，Kafka 的数据传输通过 TransportLayer 来完成，其子类 PlaintextTransportLayer 通过Java NIO 的 FileChannel 的 transferTo 和 transferFrom 方法实现零拷贝。



注： transferTo 和 transferFrom 并不保证一定能使用零拷贝，需要操作系统支持。

Linux 2.4+ 内核通过 sendfile 系统调用，提供了零拷贝。

2.5.3.2 页缓存

页缓存是操作系统实现的一种主要的磁盘缓存，以此用来减少对磁盘 I/O 的操作。

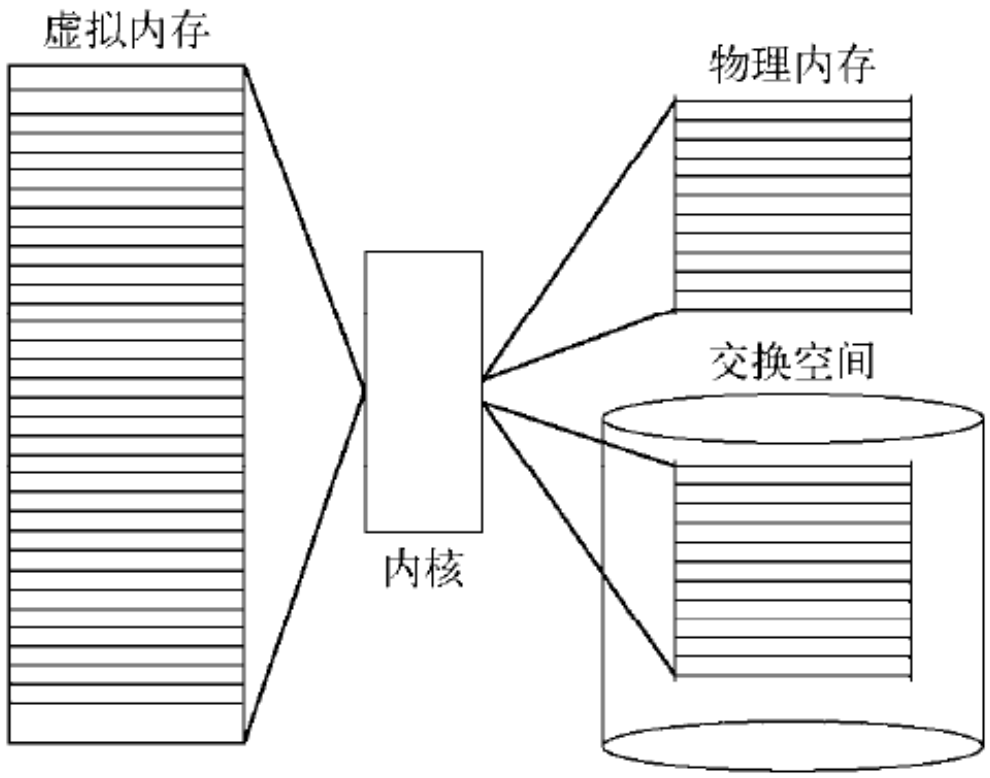
具体来说，就是把磁盘中的数据缓存到内存中，把对磁盘的访问变为对内存的访问。

Kafka接收来自socket buffer的网络数据，应用进程不需要中间处理、直接进行持久化时。可以使用mmap内存文件映射。

Memory Mapped Files

简称mmap，简单描述其作用就是：将磁盘文件映射到内存, 用户通过修改内存就能修改磁盘文件。

它的工作原理是直接利用操作系统的Page来实现磁盘文件到物理内存的直接映射。完成映射之后你对物理内存的操作会被同步到硬盘上（操作系统在适当的时候）。



通过mmap，进程像读写硬盘一样读写内存（当然是虚拟机内存）。使用这种方式可以获取很大的I/O提升，省去了用户空间到内核空间复制的开销。

mmap也有一个很明显的缺陷：不可靠，写到mmap中的数据并没有被真正的写到硬盘，操作系统会在程序主动调用flush的时候才把数据真正的写到硬盘。

Kafka提供了一个参数`producer.type`来控制是不是主动flush；

如果Kafka写入到mmap之后就立即flush然后再返回Producer叫同步(sync)；

写入mmap之后立即返回Producer不调用flush叫异步(async)。

Java NIO对文件映射的支持

Java NIO，提供了一个MappedByteBuffer 类可以用来实现内存映射。

MappedByteBuffer只能通过调用FileChannel的map()取得，再没有其他方式。

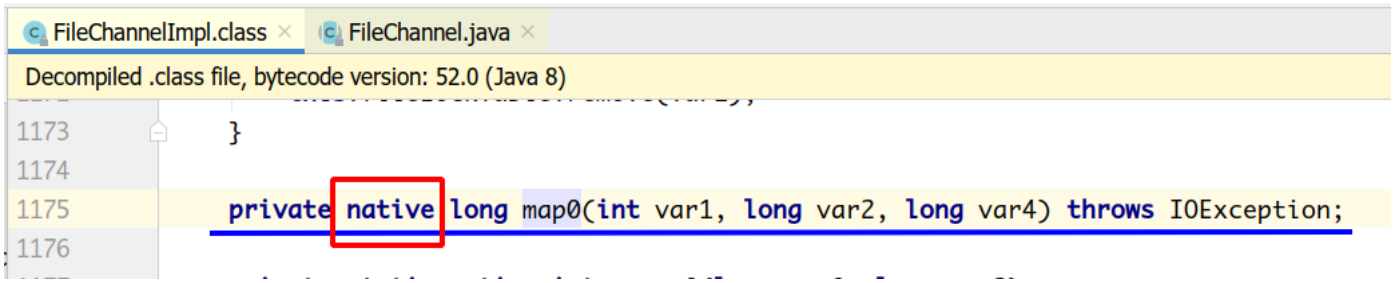
FileChannel.map()是抽象方法，具体实现是在 FileChannelImpl.map()可自行查看JDK源码，其map0()方法就是调用了Linux内核的mmap的API。

FileChannelImpl.class × FileChannel.java ×

Decompiled .class file, bytecode version: 52.0 (Java 8)

```
864 @    public MappedByteBuffer map(MapMode var1, long var2, long var4) throws IOException {
865          this.ensureOpen();
866          if (var1 == null) {
867              throw new NullPointerException("Mode is null");
868          } else if (var2 < 0L) {
```

```
try {
    var7 = this.map0(var6, var36, var10);
} catch (OutOfMemoryError var31) {
```



使用 MappedByteBuffer类要注意的是

- mmap的文件映射，在full gc时才会进行释放。当close时，需要手动清除内存映射文件，可以反射调用sun.misc.Cleaner方法。

当一个进程准备读取磁盘上的文件内容时：

1. 操作系统会先查看待读取的数据所在的页 (page)是否在页缓存(pagecache)中，如果存在(命中)则直接返回数据，从而避免了对物理磁盘的 I/O 操作；
2. 如果没有命中，则操作系统会向磁盘发起读取请求并将读取的数据页存入页缓存，之后再将数据返回给进程。

如果一个进程需要将数据写入磁盘：

1. 操作系统也会检测数据对应的页是否在页缓存中，如果不存在，则会先在页缓存中添加相应的页，最后将数据写入对应的页。
2. 被修改过后的页也就变成了脏页，操作系统会在合适的时间把脏页中的数据写入磁盘，以保持数据的一致性。

对一个进程而言，它会在进程内部缓存处理所需的数据，然而这些数据有可能还缓存在操作系统的页缓存中，因此同一份数据有可能被缓存了两次。并且，除非使用Direct I/O的方式， 否则页缓存很难被禁止。

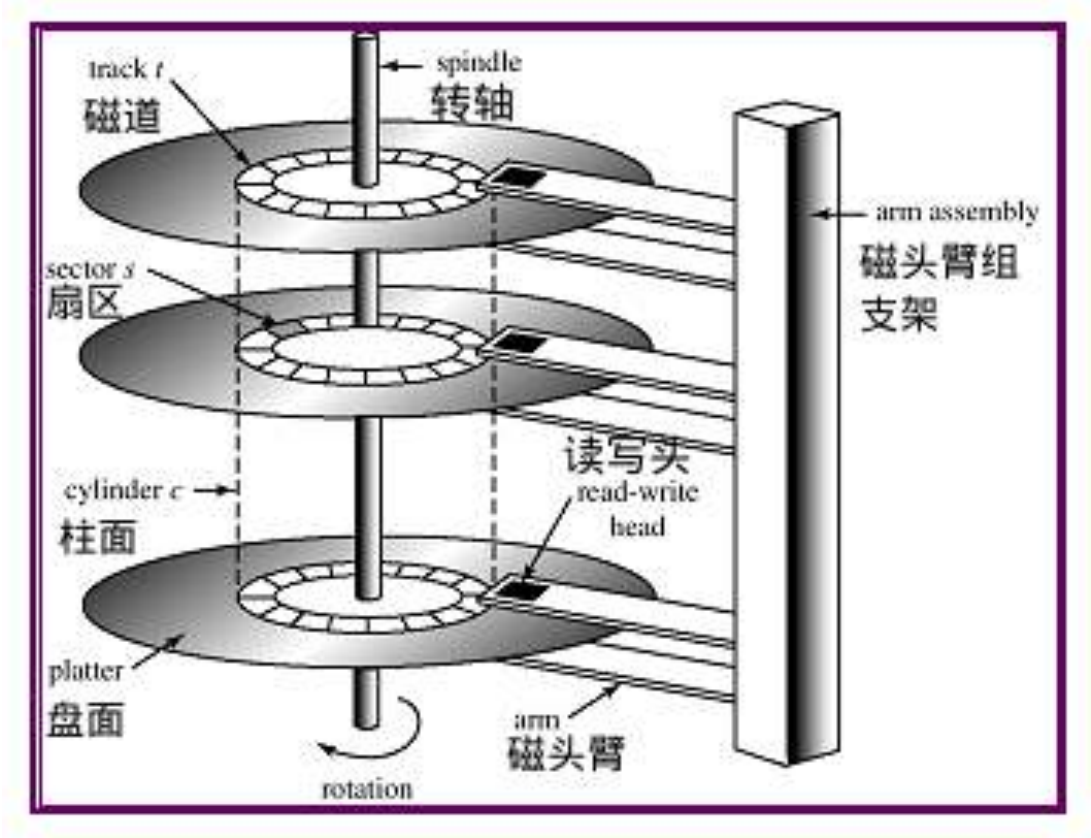
当使用页缓存的时候，即使Kafka服务重启， 页缓存还是会保持有效，然而进程内的缓存却需要重建。这样也极大地简化了代码逻辑，因为维护页缓存和文件之间的一致性交由操作系统来负责，这样会比进程内维护更加安全有效。

Kafka中大量使用了页缓存，这是 Kafka 实现高吞吐的重要因素之一。

消息先被写入页缓存，由操作系统负责刷盘任务。

2.5.3.3 顺序写入

操作系统可以针对线性读写做深层次的优化，比如预读(read-ahead，提前将一个比较大的磁盘块读入内存) 和后写(write-behind，将很多小的逻辑写操作合并起来组成一个大的物理写操作)技术。



Kafka 在设计时采用了文件追加的方式来写入消息，即只能在日志文件的尾部追加新的消 息，并且也不允许修改已写入的消息，这种方式属于典型的顺序写盘的操作，所以就算 Kafka 使用磁盘作为存储介质，也能承载非常大的吞吐量。

mmap和sendfile：

1. Linux内核提供、实现零拷贝的API；
2. sendfile 是将读到内核空间的数据，转到socket buffer，进行网络发送；
3. mmap将磁盘文件映射到内存，支持读和写，对内存的操作会反映在磁盘文件上。
4. RocketMQ 在消费消息时，使用了 mmap。kafka 使用了 sendFile。

Kafka速度快是因为：

1. partition顺序读写，充分利用磁盘特性，这是基础；
2. Producer生产的数据持久化到broker，采用mmap文件映射，实现顺序的快速写入；
3. Customer从broker读取数据，采用sendfile，将磁盘文件读到OS内核缓冲区后，直接转到socket buffer进行网络发送。

2.6 稳定性

2.6.1 事务

一、事务场景

1. 如producer发的多条消息组成一个事务这些消息需要对consumer同时可见或者同时不可见 。
2. producer可能会给多个topic，多个partition发消息，这些消息也需要能放在一个事务里面，这就形成了一个典型的分布式事务。
3. kafka的应用场景经常是应用先消费一个topic，然后做处理再发到另一个topic，这个consume-transform-produce过程需要放到一个事务里面，比如在消息处理或者发送的过程中如果失败了，消费偏移量也不能提交。
4. producer或者producer所在的应用可能会挂掉，新的producer启动以后需要知道怎么处理之前未完成的事务 。
5. 在一个原子操作中，根据包含的操作类型，可以分为三种情况，**前两种情况是事务引入的场景**，最后一种没用。
    1. 只有Producer生产消息；
    2. 消费消息和生产消息并存，**这个是事务场景中最常用的情况**，就是我们常说的`consume-transform-produce`模式
    3. 只有consumer消费消息，这种操作其实没有什么意义，跟使用手动提交效果一样，而且也不是事务属性引入的目的，所以一般不会使用这种情况

二、几个关键概念和推导

1. 因为producer发送消息可能是分布式事务，所以引入了常用的2PC，所以有事务协调者(Transaction Coordinator)。Transaction Coordinator和之前为了解决脑裂和惊群问题引入的Group Coordinator在选举上类似。
2. 事务管理中事务日志是必不可少的，kafka使用一个内部topic来保存事务日志，这个设计和之前使用内部topic保存偏移量的设计保持一致。事务日志是Transaction Coordinator管理的状态的持久化，因为不需要回溯事务的历史状态，所以事务日志只用保存最近的事务状态。`__transaction_state`
3. 因为事务存在commit和abort两种操作，而客户端又有read committed和read uncommitted两种隔离级别，所以消息队列必须能标识事务状态，这个被称作Control Message。
4. producer挂掉重启或者漂移到其它机器需要能关联的之前的未完成事务所以需要有一个唯一标识符来进行关联，这个就是TransactionalId，一个producer挂了，另一个有相同TransactionalId的producer能够接着处理这个事务未完成的状态。kafka目前没有引入全局序，所以也没有transaction id，这个TransactionalId是用户提前配置的。
5. TransactionalId能关联producer，也需要避免两个使用相同TransactionalId的producer同时存在，所以引入了producer epoch来保证对应一个TransactionalId只有一个活跃的producer

三、事务语义

2.1. 多分区原子写入

事务能够保证Kafka topic下每个分区的原子写入。事务中所有的消息都将被成功写入或者丢弃。

首先，我们来考虑一下原子`读取-处理-写入`周期是什么意思。简而言之，这意味着如果某个应用程序在某个topic tp0的偏移量X处读取到了消息A，并且在对消息A进行了一些处理（如B = F（A））之后将消息B写入topic tp1，则只有当消息A和B被认为被成功地消费并一起发布，或者完全不发布时，整个读取过程写入操作是原子的。

现在，只有当消息A的偏移量X被标记为已消费，消息A才从topic tp0消费，消费到的数据偏移量（record offset）将被标记为提交偏移量（Committing offset）。在Kafka中，我们通过写入一个名为offsets topic的内部Kafka topic来记录offset commit。消息仅在其offset被提交给offsets topic时才被认为成功消费。

由于offset commit只是对Kafkatopic的另一次写入，并且由于消息仅在提交偏移量时被视为成功消费，所以跨多个主题和分区的原子写入也启用原子`读取-处理-写入`循环：提交偏移量X到offset topic和消息B到tp1的写入将是单个事务的一部分，所以整个步骤都是原子的。

2.2. 粉碎“僵尸实例”

我们通过为每个事务Producer分配一个称为transactional.id的唯一标识符来解决僵尸实例的问题。在进程重新启动时能够识别相同的Producer实例。

API要求事务性Producer的第一个操作应该是在Kafka集群中显示注册transactional.id。 当注册的时候，Kafka broker用给定的transactional.id检查打开的事务并且完成处理。 Kafka也增加了一个与transactional.id相关的epoch。Epoch存储每个transactional.id内部元数据。

一旦epoch被触发，任何具有相同的transactional.id和旧的epoch的生产者被视为僵尸，Kafka拒绝来自这些生产者的后续事务性写入。

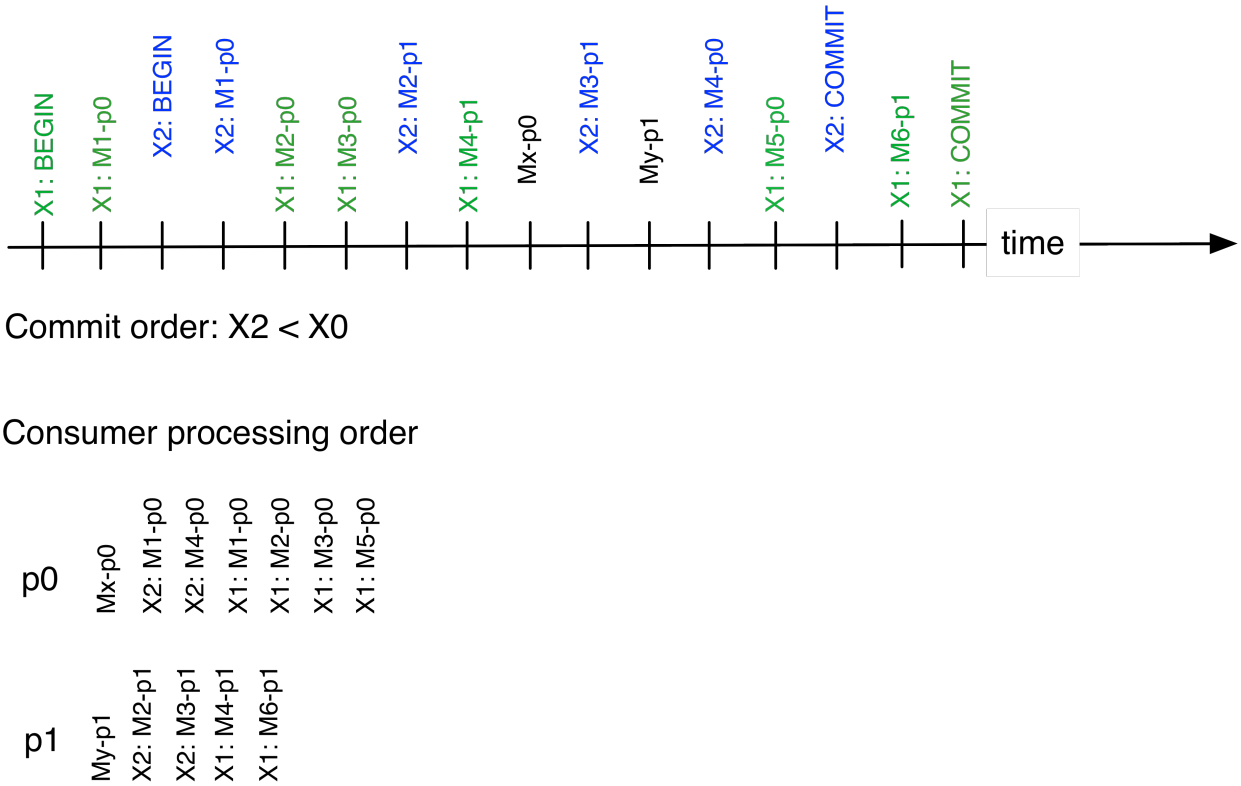
简而言之：Kafka可以保证Consumer最终只能消费非事务性消息或已提交事务性消息。它将保留来自未完成事务的消息，并过滤掉已中止事务的消息。

2.3 事务消息定义

生产者可以显式地发起事务会话，在这些会话中发送（事务）消息，并提交或中止事务。有如下要求：

1. 原子性：消费者的应用程序不应暴露于未提交事务的消息中。
2. 持久性：Broker不能丢失任何已提交的事务。
3. 排序：事务消费者应在每个分区中以原始顺序查看事务消息。
4. 交织：每个分区都应该能够接收来自事务性生产者和非事务生产者的消息
5. 事务中不应有重复的消息。

如果允许事务性和非事务性消息的交织，则非事务性和事务性消息的相对顺序将基于附加（对于非事务性消息）和最终提交（对于事务性消息）的相对顺序。



在上图中，分区p0和p1接收事务X1和X2的消息，以及非事务性消息。时间线是消息到达Broker的时间。由于首先提交了X2，所以每个分区都将在X1之前公开来自X2的消息。由于非事务性消息在X1和X2的提交之前到达，因此这些消息将在来自任一事务的消息之前公开。

四、事务配置

1、创建消费者代码，需要：

- 将配置中的自动提交属性（auto.commit）进行关闭
- 而且在代码里面也不能使用手动提交commitSync( )或者commitAsync( )
- 设置isolation.level：READ_COMMITTED或READ_UNCOMMITTED

2、创建生成者，代码如下,需要：

- 配置transactional.id属性
- 配置enable.idempotence属性

五、事务概览

生产者将表示事务开始/结束/中止状态的事务控制消息发送给使用多阶段协议管理事务的高可用事务协调器。生产者将事务控制记录（开始/结束/中止）发送到事务协调器，并将事务的消息直接发送到目标数据分区。消费者需要了解事务并缓冲每个待处理的事务，直到它们到达其相应的结束（提交/中止）记录为止。

- 事务组
- 事务组中的生产者
- 事务组的事务协调器
- Leader brokers（事务数据所在分区的Broker）
- 事务的消费者

六、事务组

事务组用于映射到特定的事务协调器（基于日志分区数字的哈希）。该组中的生产者需要配置为该组事务生产者。由于来自这些生产者的所有事务都通过此协调器进行，因此我们可以在这些事务生产者之间实现严格的有序。

七、生产者ID和事务组状态

事务生产者需要两个新参数：生产者ID和生产组。

需要将生产者的输入状态与上一个已提交的事务相关联。这使事务生产者能够重试事务（通过为该事务重新创建输入状态；在我们的用例中通常是偏移量的向量）。

可以使用消费者偏移量管理机制来管理这些状态。消费者偏移量管理器将每个键（`consumergroup-topic-partition`）与该分区的最后一个检查点偏移量和元数据相关联。在事务生产者中，我们保存消费者的偏移量，该偏移量与事务的提交点关联。此偏移提交记录（在`__consumer_offsets`主题中）应作为事务的一部分写入。即，存储消费组偏移量的`__consumer_offsets`主题分区将需要参与事务。因此，假定生产者在事务中间失败（事务协调器随后到期）；当生产者恢复时，它可以发出偏移量获取请求，以恢复与最后提交的事务相关联的输入偏移量，并从该点恢复事务处理。

为了支持此功能，我们需要对偏移量管理器和压缩的`__consumer_offsets`主题进行一些增强。

首先，压缩的主题现在还将包含事务控制记录。我们将需要为这些控制记录提出剔除策略。

其次，偏移量管理器需要具有事务意识；特别是，如果组与待处理的事务相关联，则偏移量提取请求应返回错误。

八、事务协调器

- 事务协调器是`__transaction_state`主题特定分区的Leader分区所在的Broker。它负责初始化、提交以及回滚事务。事务协调器在内存管理如下的状态：

- 对应正在处理的事务的第一个消息的HW。事务协调器周期性地将HW写到ZK。
- 事务控制日志中存储对应于日志HW的所有正在处理的事务：
- 事务消息主题分区的列表。
    - 事务的超时时间。
    - 与事务关联的Producer ID。

需要确保无论是什么样的保留策略（日志分区的删除还是压缩），都不能删除包含事务HW的日志分段。

九、事务流程

初始阶段

(图中步骤A)

1. Producer：计算哪个Broker作为事务协调器。
2. Producer：向事务协调器发送BeginTransaction(producerId, generation, partitions...)请求，当然也可以发送另一个包含事务过期时间的。如果生产者需要将消费者状态作为事务的一部分提交事务，则需要在BeginTransaction中包含对应的`__consumer_offsets`主题分区信息。
3. Broker：生成事务ID
4. Coordinator：向事务协调主题追加BEGIN(TxId, producerId, generation, partitions...)消息，然后发送响应给生产者。
5. Producer：读取响应（包含了事务ID：TxId）
6. Coordinator (and followers)：在内存更新当前事务的待确认事务状态和数据分区信息。

发送阶段

(图中步骤2)

Producer：发送事务消息给主题Leader分区所在的Broker。每个消息需要包含TxId和TxCtl字段。TxCtl仅用于标记事务的最终状态（提交还是中止）。生产者请求也封装了生产者ID，但是不追加到日志中。

结束阶段 (生产者准备提交事务)

(图中步骤3、4、5。)

1. Producer：发送OffsetCommitRequest请求提交与事务结束状态关联的输入状态（如下一个事务输入从哪儿开始）
2. Producer：发送CommitTransaction(TxId, producerId, generation)请求给事务协调器并等待响应。(如果响应中没有错误信息，表示将提交事务)
3. Coordinator：向事务控制主题追加PREPARE_COMMIT(TxId)请求并向生产者发送响应。
4. Coordinator：向事务涉及到的每个Leader分区（事务的业务数据的目标主题）的Broker发送一个CommitTransaction(TxId, partitions...)请求。
5. 事务业务数据的目标主题相关Leader分区Broker：
    1. 如果是非`__consumer_offsets`主题的Leader分区：一收到CommitTransaction(TxId, partition1, partition2, ...)请求就会向对应的分区Broker发送空(null)消息(没有key/value)并给该消息设置TxId和TxCtl（设置为COMMITTED）字段。Leader分区的Broker给协调器发送响应。

2. 如果是`__consumer_offsets`主题的Leader分区：追加消息，该消息的key是`G-LAST-COMMIT`，value就是`TxId`的值。同时也应该给该消息设置TxId和TxCtl字段。Broker向协调器发送响应。

6. Coordinator：向事务控制主题发送COMMITTED(TxId)请求。`__transaction_state`
7. Coordinator (and followers)：尝试更新HW。

十、事务的中止

当事务生产者发送业务消息的时候如果发生异常，可以中止该事务。如果事务提交超时，事务协调器也会中止当前事务。

- Producer：向事务协调器发送AbortTransaction(TxId)请求并等待响应。(一个没有异常的响应表示事务将会中止)
- Coordinator：向事务控制主题追加PREPARE_ABORT(TxId)消息，然后向生产者发送响应。
- Coordinator：向事务业务数据的目标主题的每个涉及到的Leader分区Broker发送AbortTransaction(TxId, partitions...)请求。(收到Leader分区Broker响应后，事务协调器中止动作跟上面的提交类似。)

十一、基本事务流程的失败

- 生产者发送BeginTransaction(TxId):的时候超时或响应中包含异常，生产者使用相同的TxId重试。
- 生产者发送数据时的Broker错误：生产者应中止（然后重做）事务（使用新的TxId）。如果生产者没有中止事务，则协调器将在事务超时后中止事务。仅在可能已将请求数据附加并复制到Follower的错误的情况下才需要重做事务。例如，生产者请求超时将需要重做，而NotLeaderForPartitionException不需要重做。
- 生产者发送CommitTransaction(TxId)请求超时或响应中包含异常，生产者使用相同的TxId重试事务。此时需要幂等性。

十二、主题的压缩

压缩主题在压缩过程中会丢弃具有相同键的早期记录。如果这些记录是事务的一部分，这合法吗？ 这可能有点怪异，但可能不会太有害，因为在主题中使用压缩策略的理由是保留关键数据的最新更新。

如果该应用程序正在（例如）更新某些表，并且事务中的消息对应于不同的键，则这种情况可能导致数据库视图不一致。

十三、事务相关配置

1、Broker configs

配置项	说明
`transactional.id.timeout.ms`	在ms中，事务协调器在生产者TransactionalId提前过期之前等待的最长时间，并且没有从该生产者TransactionalId接收到任何事务状态更新。默认是604800000(7天)。这允许每周一次的生产者作业维护它们的id
`max.transaction.timeout.ms`	事务允许的最大超时。如果客户端请求的事务时间超过此时间，broke将在InitPidRequest中返回InvalidTransactionTimeout错误。这可以防止客户机超时过大，从而导致用户无法从事务中包含的主题读取内容。 默认值为900000(15分钟)。这是消息事务需要发送的时间的保守上限。
`transaction.state.log.replication.factor`	事务状态topic的副本数量。默认值:3
`transaction.state.log.num.partitions`	事务状态主题的分区数。默认值:50
`transaction.state.log.min.isr`	事务状态主题的每个分区ISR最小数量。默认值:2
`transaction.state.log.segment.bytes`	事务状态主题的segment大小。默认值:104857600字节

2、Producer configs

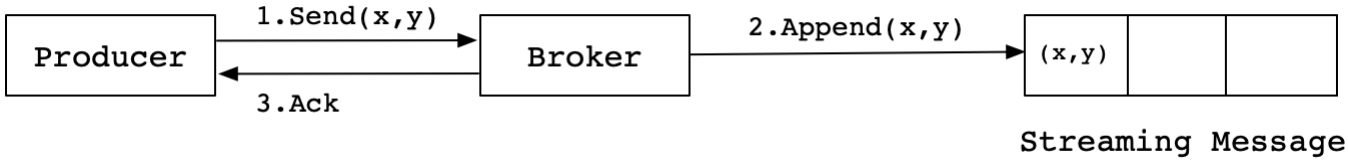
配置项	说明
`enable.idempotence`	开启幂等
`transaction.timeout.ms`	事务超时时间 事务协调器在主动中止正在进行的事务之前等待生产者更新事务状态的最长时间。这个配置值将与InitPidRequest一起发送到事务协调器。如果该值大于max.transaction.timeout。在broke中设置ms时，请求将失败，并出现InvalidTransactionTimeout错误。 默认是60000。这使得交易不会阻塞下游消费超过一分钟，这在实时应用程序中通常是允许的。
`transactional.id`	用于事务性交付的TransactionalId。这支持跨多个生产者会话的可靠性语义，因为它允许客户端确保使用相同TransactionalId的事务在启动任何新事务之前已经完成。如果没有提供TransactionalId，则生产者仅限于幂等交付。

3、Consumer configs

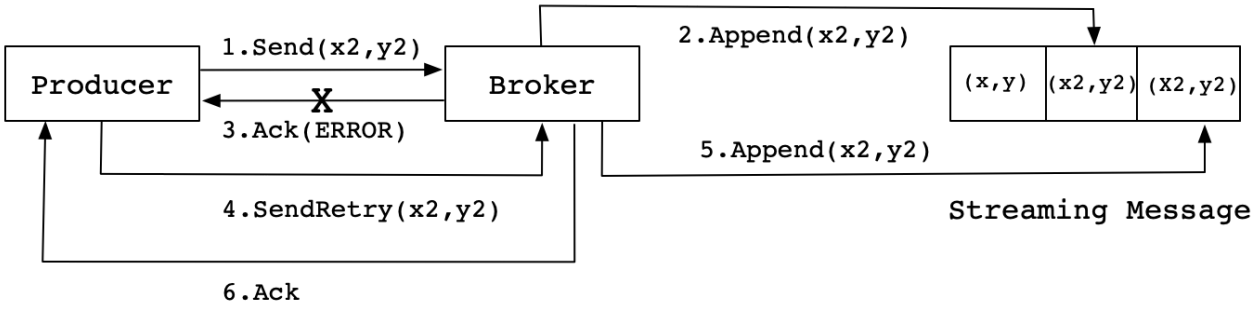
配置项	说明
`isolation.level`	- read_uncommitted:以偏移顺序使用已提交和未提交的消息。 - read_committed:仅以偏移量顺序使用非事务性消息或已提交事务性消息。为了维护偏移排序，这个设置意味着我们必须在使用者中缓冲消息，直到看到给定事务中的所有消息。

2.6.1.1 幂等性

Kafka在引入幂等性之前，Producer向Broker发送消息，然后Broker将消息追加到消息流中后给Producer返回Ack信号值。实现流程如下：



生产中，会出现各种不确定的因素，比如在Producer在发送给Broker的时候出现网络异常。比如以下这种异常情况的出现：



上图这种情况，当Producer第一次发送消息给Broker时，Broker将消息(x2,y2)追加到了消息流中，但是在返回Ack信号给Producer时失败了（比如网络异常）。此时，Producer端触发重试机制，将消息(x2,y2)重新发送给Broker，Broker接收到消息后，再次将该消息追加到消息流中，然后成功返回Ack信号给Producer。这样下来，消息流中就被重复追加了两条相同的(x2,y2)的消息。

幂等性

保证在消息重发的时候，消费者不会重复处理。即使在消费者收到重复消息的时候，重复处理，也要保证最终结果的一致性。

所谓幂等性，数学概念就是：`f(f(x)) = f(x)`。f函数表示对消息的处理。

比如，银行转账，如果失败，需要重试。不管重试多少次，都要保证最终结果一定是一致的。

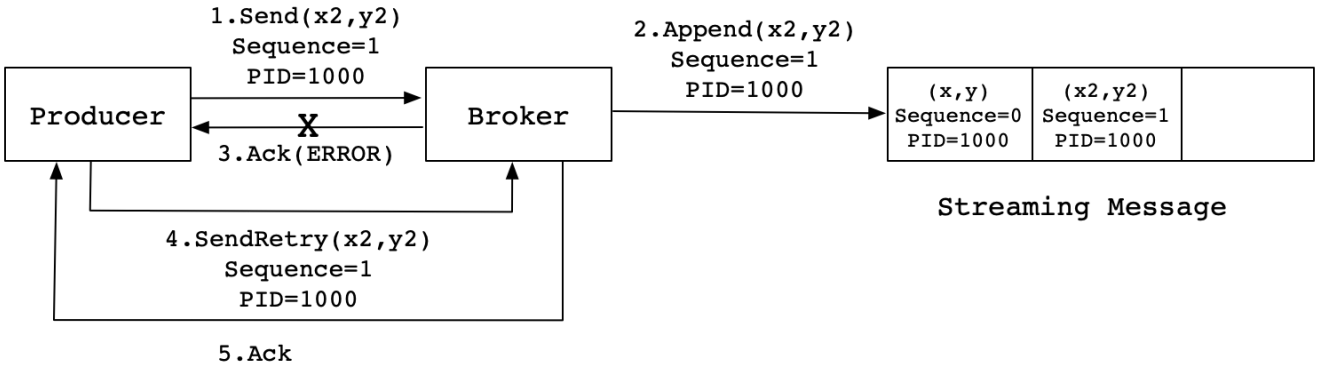
幂等性实现

添加唯一ID，类似于数据库的主键，用于唯一标记一个消息。

Kafka为了实现幂等性，它在底层设计架构中引入了ProducerID和SequenceNumber。

- ProducerID：在每个新的Producer初始化时，会被分配一个唯一的ProducerID，这个ProducerID对客户端使用者是不可见的。
- SequenceNumber：对于每个ProducerID，Producer发送数据的每个Topic和Partition都对应一个从0开始单调递增的SequenceNumber值。

同样，这是一种理想状态下的发送流程。实际情况下，会有很多不确定的因素，比如Broker在发送Ack信号给Producer时出现网络异常，导致发送失败。异常情况如下图所示：



当Producer发送消息(x2,y2)给Broker时，Broker接收到消息并将其追加到消息流中。此时，Broker返回Ack信号给Producer时，发生异常导致Producer接收Ack信号失败。对于Producer来说，会触发重试机制，将消息(x2,y2)再次发送，但是，由于引入了幂等性，在每条消息中附带了PID（ProducerID）和SequenceNumber。相同的PID和SequenceNumber发送给Broker，而之前Broker缓存过之前发送的相同的消息，那么在消息流中的消息就只有一条(x2,y2)，不会出现重复发送的情况。

客户端在生成Producer时，会实例化如下代码：

```
// 实例化一个Producer对象
Producer<String, String> producer = new KafkaProducer<>(props);
```

在org.apache.kafka.clients.producer.internals.Sender类中，在run()中有一个maybeWaitForPid()方法，用来生成一个ProducerID，实现代码如下：

```
private void maybeWaitForPid() {
        if (transactionState == null)
            return;

        while (!transactionState.hasPid()) {
            try {
                Node node = awaitLeastLoadedNodeReady(requestTimeout);
                if (node != null) {
                    ClientResponse response = sendAndAwaitInitPidRequest(node);
                    if (response.hasResponse() && (response.responseBody() instanceof InitPidResponse)) {
                        InitPidResponse initPidResponse = (InitPidResponse) response.responseBody();
                        transactionState.setPidAndEpoch(initPidResponse.producerId(), initPidResponse.epoch());
```

```
                    } else {
                        log.error("Received an unexpected response type for an InitPidRequest from {}. " +
                                  "We will back off and try again.", node);
                    }
                } else {
                    log.debug("Could not find an available broker to send InitPidRequest to. " +
                              "We will back off and try again.");
                }
            } catch (Exception e) {
                log.warn("Received an exception while trying to get a pid. Will back off and retry.", e);
            }
            log.trace("Retry InitPidRequest in {}ms.", retryBackoffMs);
            time.sleep(retryBackoffMs);
            metadata.requestUpdate();
        }
    }
```

2.6.1.2 事务操作

在Kafka事务中，一个原子性操作，根据操作类型可以分为3种情况。情况如下：

- 只有Producer生产消息，这种场景需要事务的介入；
- 消费消息和生产消息并存，比如Consumer&Producer模式，这种场景是一般Kafka项目中比较常见的模式，需要事务介入；
- 只有Consumer消费消息，这种操作在实际项目中意义不大，和手动Commit Offsets的结果一样，而且这种场景不是事务的引入目的。

```
// 初始化事务，需要注意确保transation.id属性被分配
void initTransactions();

// 开启事务
void beginTransaction() throws ProducerFencedException;

// 为Consumer提供的在事务内Commit Offsets的操作
void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets,
                              String consumerGroupId) throws ProducerFencedException;

// 提交事务
void commitTransaction() throws ProducerFencedException;

// 放弃事务，类似于回滚事务的操作
```

```
void abortTransaction() throws ProducerFencedException;
```

案例1：单个Producer，使用事务保证消息的仅一次发送：

```
package com.lagou.kafka.demo.producer;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.HashMap;
import java.util.Map;

public class MyTransactionalProducer {

    public static void main(String[] args) {

        Map<String, Object> configs = new HashMap<>();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        // 提供客户端ID
        configs.put(ProducerConfig.CLIENT_ID_CONFIG, "tx_producer");
        // 事务ID
        configs.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "my_tx_id");
        // 要求ISR都确认
        configs.put(ProducerConfig.ACKS_CONFIG, "all");

        KafkaProducer<String, String> producer = new KafkaProducer<String, String>
(configs);
        // 初始化事务
        producer.initTransactions();

        // 开启事务
        producer.beginTransaction();

        try {
//            producer.send(new ProducerRecord<>("tp_tx_01", "tx_msg_01"));
            producer.send(new ProducerRecord<>("tp_tx_01", "tx_msg_02"));
//            int i = 1 / 0;
            // 提交事务
            producer.commitTransaction();
        } catch (Exception ex) {
            // 中止事务
            producer.abortTransaction();
        } finally {
```

```
            // 关闭生产者
            producer.close();
        }
    }
}
```

案例2：在`消费-转换-生产`模式，使用事务保证仅一次发送。

```
package com.lagou.kafka.demo;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class MyTransactional {

    public static KafkaProducer<String, String> getProducer() {
        Map<String, Object> configs = new HashMap<>();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

        // 设置client.id
        configs.put(ProducerConfig.CLIENT_ID_CONFIG, "tx_producer_01");

        // 设置事务id
        configs.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "tx_id_02");

        // 需要所有的ISR副本确认
        configs.put(ProducerConfig.ACKS_CONFIG, "all");

        // 启用幂等性
        configs.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);

        KafkaProducer<String, String> producer = new KafkaProducer<String, String>
(configs);

        return producer;
    }
```

```
39
40      public static KafkaConsumer<String, String> getConsumer(String consumerGroupId)
{
41          Map<String, Object> configs = new HashMap<>();
42          configs.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
43          configs.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
44          configs.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
45
46          // 设置消费组ID
47          configs.put(ConsumerConfig.GROUP_ID_CONFIG, "consumer_grp_02");
48
49          // 不启用消费者偏移量的自动确认，也不要手动确认
50          configs.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
51
52          configs.put(ConsumerConfig.CLIENT_ID_CONFIG, "consumer_client_02");
53
54          configs.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
55
56          // 只读取已提交的消息
57  //          configs.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");
58
59          KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>
(configs);
60
61          return consumer;
62      }
63
64
65      public static void main(String[] args) {
66
67          String consumerGroupId = "consumer_grp_id_101";
68
69          KafkaProducer<String, String> producer = getProducer();
70          KafkaConsumer<String, String> consumer = getConsumer(consumerGroupId);
71
72          // 事务的初始化
73          producer.initTransactions();
74          //订阅主题
75          consumer.subscribe(Collections.singleton("tp_tx_01"));
76
77          final ConsumerRecords<String, String> records = consumer.poll(1_000);
78
79          // 开启事务
80          producer.beginTransaction();
81          try {
82              Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();
83
84              for (ConsumerRecord<String, String> record : records) {
85
```

```
                System.out.println(record);

                producer.send(new ProducerRecord<String, String>("tp_tx_out_01",
record.key(), record.value()));

                offsets.put(
                        new TopicPartition(record.topic(), record.partition()),
                        new OffsetAndMetadata(record.offset() + 1)); // 偏移量表示下一
条要消费的消息
            }

            // 将该消息的偏移量提交作为事务的一部分，随事务提交和回滚（不提交消费偏移量）
            producer.sendOffsetsToTransaction(offsets, consumerGroupId);

//              int i = 1 / 0;

            // 提交事务
            producer.commitTransaction();

        } catch (Exception e) {
            e.printStackTrace();
            // 回滚事务
            producer.abortTransaction();
        } finally {
            // 关闭资源
            producer.close();
            consumer.close();
        }


    }
}

```

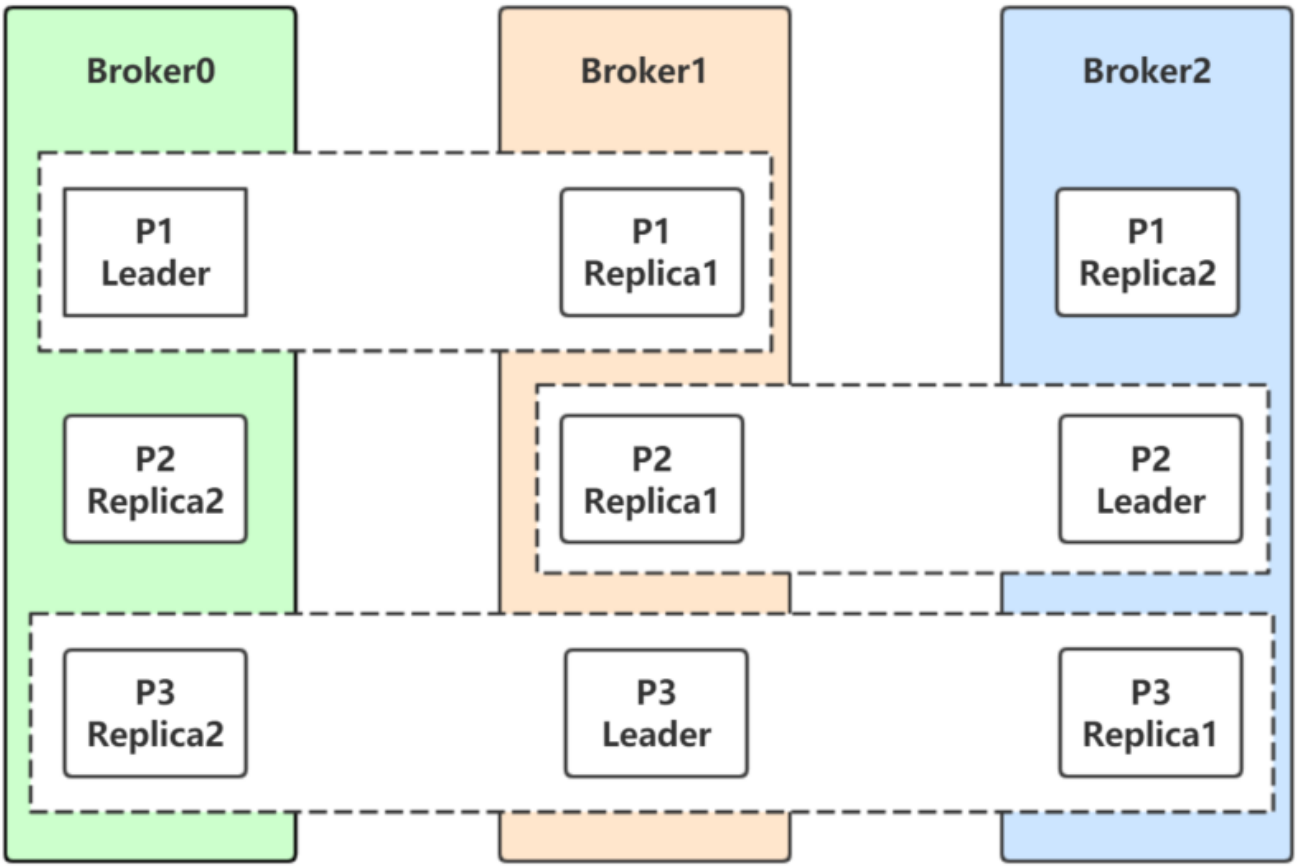
2.6.2 控制器

Kafka集群包含若干个broker，broker.id指定broker的编号，编号不要重复。

Kafka集群上创建的主题，包含若干个分区。

每个分区包含若干个副本，副本因子包括了Follower副本和Leader副本。

副本又分为ISR（同步副本分区）和OSR（非同步副本分区）。



控制器就是一个broker。

控制器除了一般broker的功能，还负责Leader分区的选举。

2.6.2.1 broker选举

集群里第一个启动的broker在Zookeeper中创建临时节点`<KafkaZkChroot>/controller`。

其他broker在该控制器节点创建Zookeeper watch对象，使用Zookeeper的监听机制接收该节点的变更。

即：Kafka通过Zookeeper的分布式锁特性选举**集群控制器**。

下图中，节点`/myKafka/controller`是一个zookeeper临时节点，其中`"brokerid":0`，表示当前控制器是broker.id为0的broker。

```
[zk: localhost:2181(CONNECTED) 3] get /myKafka/controller
{"version":1,"brokerid":0,"timestamp":"1596122550321"}
cZxid = 0x39b
ctime = Thu Jul 30 23:22:30 CST 2020
mZxid = 0x39b
mtime = Thu Jul 30 23:22:30 CST 2020
pZxid = 0x39b
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x100000034460001
dataLength = 54
numChildren = 0
```

每个新选出的控制器通过 Zookeeper 的条件递增操作获得一个全新的、数值更大的 controller epoch。其他 broker 在知道当前 controller epoch 后，如果收到由控制器发出的包含较旧epoch 的消息，就会忽略它们，以防止“**脑裂**”。

比如当一个Leader副本分区所在的broker宕机，需要选举新的Leader副本分区，有可能两个具有不同纪元数字的控制器都选举了新的Leader副本分区，如果选举出来的Leader副本分区不一样，听谁的？脑裂了。有了纪元数字，直接使用纪元数字最新的控制器结果。

```
[zk: localhost:2181(CONNECTED) 7] get /myKafka/controller_epoch
9
cZxid = 0x1b
ctime = Sun Jul 26 11:50:06 CST 2020
mZxid = 0x39c
mtime = Thu Jul 30 23:22:30 CST 2020
pZxid = 0x1b
cversion = 0
dataVersion = 8
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 1
numChildren = 0
```

当控制器发现一个 broker 已经离开集群，那些失去Leader副本分区的Follower分区需要一个新Leader（这些分区的首领刚好是在这个 broker 上）。

1. 控制器需要知道哪个broker宕机了?
2. 控制器需要知道宕机的broker上负责的时候哪些分区的Leader副本分区?

下图中，`<KafkaChroot>/brokers/ids/0`保存该broker的信息，此节点为临时节点，如果broker节点宕机，该节点丢失。

集群控制器负责监听`ids`节点，一旦节点子节点发送变化，集群控制器得到通知。

```
[zk: localhost:2181(CONNECTED) 10] get /myKafka/brokers/ids/0
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://node1:9092"],"jmx_port":
-1,"host":"node1","timestamp":"1596122550833","port":9092,"version":4}
cZxid = 0x3a0
ctime = Thu Jul 30 23:22:30 CST 2020
mZxid = 0x3a0
mtime = Thu Jul 30 23:22:30 CST 2020
pZxid = 0x3a0
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x100000034460001
dataLength = 180
numChildren = 0
```

控制器遍历这些Follower副本分区，并确定谁应该成为新Leader分区，然后向所有包含新Leader分区和现有Follower的 broker 发送请求。该请求消息包含了谁是新Leader副本分区以及谁是Follower副本分区的信息。随后，新Leader分区开始处理来自生产者和消费者的请求，而跟随者开始从新Leader副本分区消费消息。

当控制器发现一个 broker 加入集群时，它会使用 broker ID 来检查新加入的 broker 是否包含现有分区的副本。如果有，控制器就把变更通知发送给新加入的 broker 和其他 broker，新 broker上的副本分区开始从Leader分区那里消费消息，与Leader分区保持同步。

结论:

1. Kafka使用 Zookeeper 的分布式锁选举控制器，并在节点加入集群或退出集群时通知控制器。
2. 控制器负责在节点加入或离开集群时进行分区Leader选举。
3. 控制器使用epoch来避免“脑裂”。“脑裂”是指两个节点同时认为自己是当前的控制器。

2.6.3 可靠性保证

概念

1. 创建Topic的时候可以指定`--replication-factor 3`，表示分区的副本数，不要超过broker的数量。
2. Leader是负责读写的节点，而其他副本则是Follower。Producer只把消息发送到Leader，Follower定期地到Leader上Pull数据。
3. ISR是Leader负责维护的与其保持同步的Replica列表，即当前活跃的副本列表。如果一个Follow落后太多，Leader会将它从ISR中移除。落后太多意思是该Follow复制的消息Follow长时间没有向Leader发送fetch请求(参数：`replica.lag.time.max.ms`默认值：10000)。
4. 为了保证可靠性，可以设置`acks=all`。Follower收到消息后，会像Leader发送ACK。一旦Leader收到了ISR中所有Replica的ACK，Leader就commit，那么Leader就向Producer发送ACK。

副本的分配：

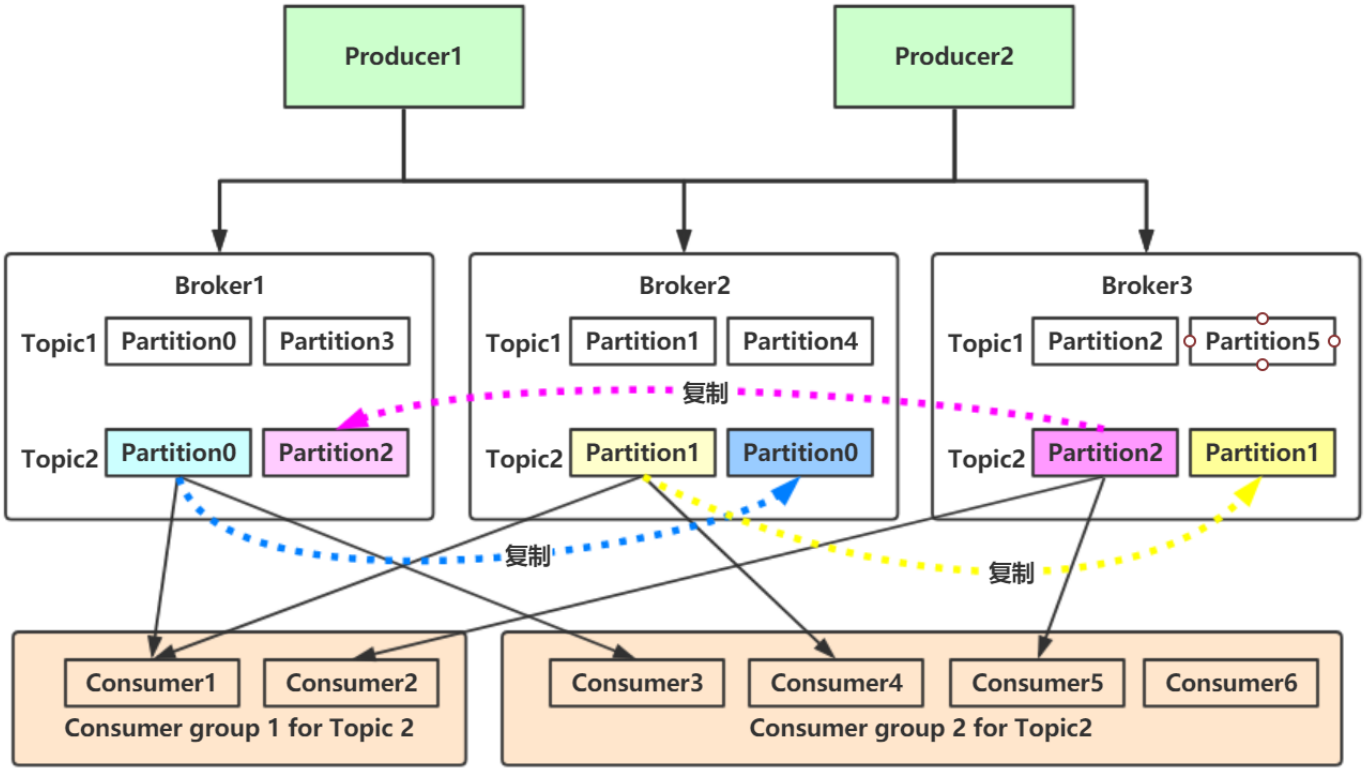
当某个topic的`--replication-factor`为N(N>1)时，每个Partition都有N个副本，称作replica。原则上是将replica均匀的分配到整个集群上。不仅如此，partition的分配也同样需要均匀分配，为了更好的负载均衡。

副本分配的三个目标：

1. 均衡地将副本分散于各个broker上
2. 对于某个broker上分配的分区，它的其他副本在其他broker上
3. 如果所有的broker都有机架信息，尽量将分区的各个副本分配到不同机架上的broker。

在不考虑机架信息的情况下：

1. 第一个副本分区通过轮询的方式挑选一个broker，进行分配。该轮询从broker列表的随机位置进行轮询。
2. 其余副本通过增加偏移进行分配。



2.6.3.1 失效副本

失效副本的判定

`replica.lag.time.max.ms`默认大小为10000。

当ISR中的一个Follower副本滞后Leader副本的时间超过参数`replica.lag.time.max.ms`指定的值时即判定为副本失效，需要将此Follower副本剔出除ISR。

具体实现原理：当Follower副本将Leader副本的LEO之前的日志全部同步时，则认为该Follower副本已经追赶上Leader副本，此时更新该副本的lastCaughtUpTimeMs标识。

Kafka的副本管理器（ReplicaManager）启动时会启动一个副本过期检测的定时任务，而这个定时任务会定时检查当前时间与副本的lastCaughtUpTimeMs差值是否大于参数`replica.lag.time.max.ms`指定的值。

Kafka源码注释中说明了一般有两种情况会导致副本失效：

1. Follower副本进程卡住，在一段时间内没有向Leader副本发起同步请求，比如频繁的Full GC。
2. Follower副本进程同步过慢，在一段时间内都无法追赶上Leader副本，比如IO开销过大。

如果通过工具增加了副本因子，那么新增加的副本在赶上Leader副本之前也都是处于失效状态的。

如果一个Follower副本由于某些原因（比如宕机）而下线，之后又上线，在追赶上Leader副本之前也是出于失效状态。

失效副本的分区个数是用于衡量Kafka性能指标的重要部分。Kafka本身提供了一个相关的指标，即UnderReplicatedPartitions，这个可以通过JMX访问:

```
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
```

取值范围是大于等于0的整数。注意：如果Kafka集群正在做分区迁移（kafka-reassign-partitions.sh）的时候，这个值也会大于0。

2.6.3.2 副本复制

日志复制算法（log replication algorithm）必须提供的基本保证是，如果它告诉客户端消息已被提交，而当前Leader出现故障，新选出的Leader也必须具有该消息。在出现故障时，Kafka会从挂掉Leader的ISR里面选择一个Follower作为这个分区新的Leader。

每个分区的 leader 会维护一个in-sync replica（同步副本列表，又称 ISR）。当Producer向broker发送消息，消息先写入到对应Leader分区，然后复制到这个分区的所有副本中。ACKS=ALL时，只有将消息成功复制到所有同步副本（ISR）后，这条消息才算被提交。

什么情况下会导致一个副本与 leader 失去同步

一个副本与 leader 失去同步的原因有很多，主要包括：

- **慢副本（Slow replica）**：follower replica 在一段时间内一直无法赶上 leader 的写进度。造成这种情况的最常见原因之一是 follower replica 上的 I/O瓶颈，导致它持久化日志的时间比它从 leader 消费消息的时间要长；
- **卡住副本（Stuck replica）**：follower replica 在很长一段时间内停止从 leader 获取消息。这可能是以为 GC 停顿，或者副本出现故障；
- **刚启动副本（Bootstrapping replica）**：当用户给某个主题增加副本因子时，新的 follower replicas 是不同步的，直到它跟上 leader 的日志。

当副本落后于 leader 分区时，这个副本被认为是不同步或滞后的。在 Kafka中，副本的滞后于Leader是根据`replica.lag.time.max.ms`来衡量。

如何确认某个副本处于滞后状态

通过`replica.lag.time.max.ms`来检测卡住副本（Stuck replica）在所有情况下都能很好地工作。它跟踪 follower 副本没有向 leader 发送获取请求的时间，通过这个可以推断 follower 是否正常。另一方面，使用消息数量检测不同步慢副本（Slow replica）的模型只有在为单个主题或具有同类流量模式的多个主题设置这些参数时才能很好地工作，但我们发现它不能扩展到生产集群中所有主题。

2.6.4 一致性保证

一、概念

1. 水位标记

水位或水印（watermark）一词，表示位置信息，即位移（offset）。Kafka源码中使用的名字是高水位，HW（high watermark）。

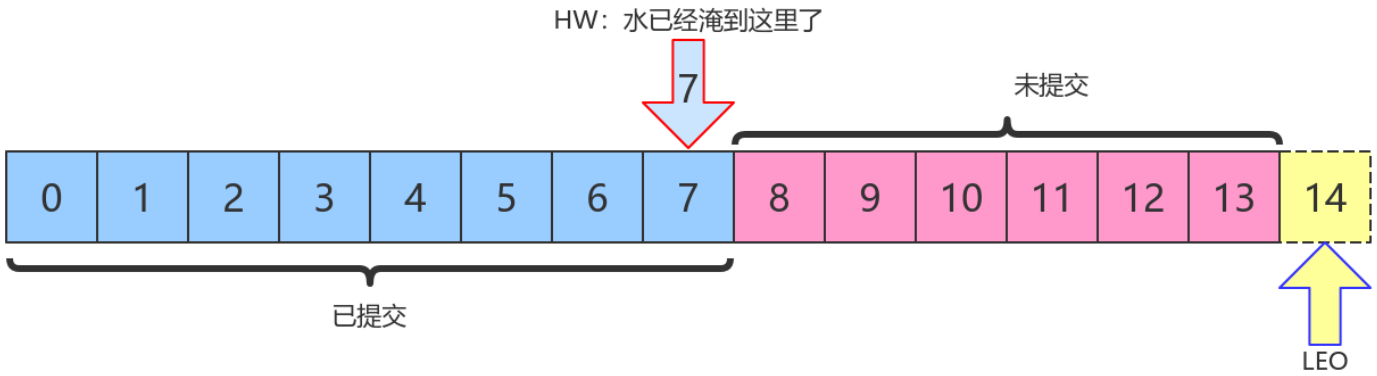
2. 副本角色

Kafka分区使用多个副本(replica)提供高可用。

3. LEO和HW

每个分区副本对象都有两个重要的属性：LEO和HW。

- LEO：即日志末端位移(log end offset)，记录了该副本日志中下一条消息的位移值。如果LEO=10，那么表示该副本保存了10条消息，位移值范围是[0, 9]。另外，Leader LEO和Follower LEO的更新是有区别的。
- HW：即上面提到的水位值。对于同一个副本对象而言，其HW值不会大于LEO值。小于等于HW值的所有消息都被认为是“已备份”的（replicated）。Leader副本和Follower副本的HW更新不同。



上图中，HW值是7，表示位移是`0~7`的所有消息都已经处于“已提交状态”（committed），而LEO值是14，8~13的消息就是未完全备份（fully replicated）——为什么没有14？LEO指向的是下一条消息到来时的位移。

消费者无法消费分区下Leader副本中位移大于分区HW的消息。

二、Follower副本何时更新LEO

Follower副本不停地向Leader副本所在的broker发送FETCH请求，一旦获取消息后写入自己的日志中进行备份。那么Follower副本的LEO是何时更新的呢？首先我必须言明，Kafka有两套Follower副本LEO：

1. 一套LEO保存在Follower副本所在Broker的**副本管理机**中；
2. 另一套LEO保存在Leader副本所在Broker的副本管理机中。Leader副本机器上保存了所有的follower副本的LEO。

Kafka使用前者帮助Follower副本更新其HW值；利用后者帮助Leader副本更新其HW。

1. Follower副本的本地LEO何时更新?

Follower副本的LEO值就是日志的LEO值，每当新写入一条消息，LEO值就会被更新。当Follower发送FETCH请求后，Leader将数据返回给Follower，此时Follower开始Log写数据，从而自动更新LEO值。

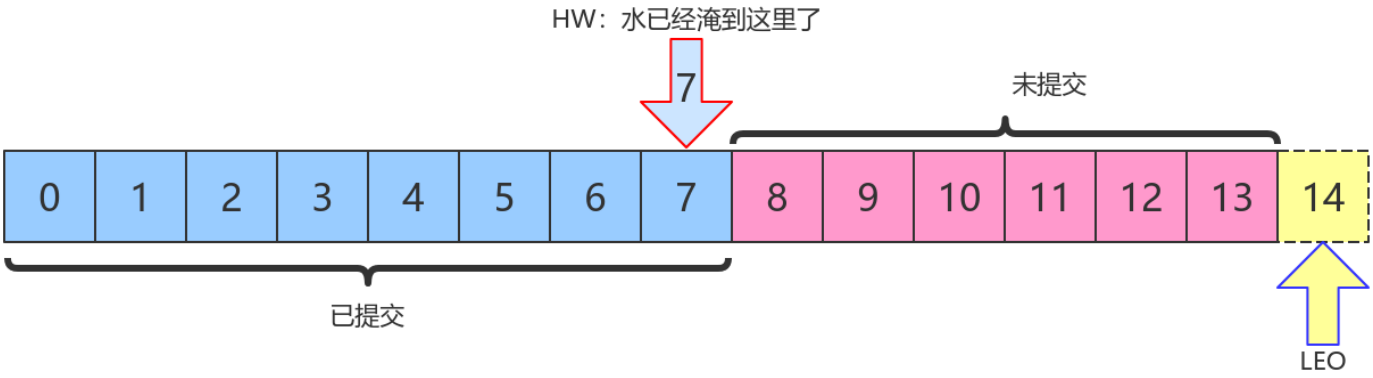
2. Leader端Follower的LEO何时更新?
   Leader端的Follower的LEO更新发生在Leader在处理Follower FETCH请求时。一旦Leader接收到Follower发送的FETCH请求，它先从Log中读取相应的数据，给Follower返回数据前，先更新Follower的LEO。

三、Follower副本何时更新HW

Follower更新HW发生在其更新LEO之后，一旦Follower向Log写完数据，尝试更新自己的HW值。

比较当前LEO值与FETCH响应中Leader的HW值，取两者的小者作为新的HW值。

即：如果Follower的LEO大于Leader的HW，Follower HW值不会大于Leader的HW值。



四、Leader副本何时更新LEO

和Follower更新LEO相同，Leader写Log时自动更新自己的LEO值。

五、Leader副本何时更新HW值

Leader的HW值就是分区HW值，直接影响分区数据对消费者的可见性 。

Leader会**尝试**去更新分区HW的四种情况：

1. Follower副本成为Leader副本时：Kafka会尝试去更新分区HW。
2. Broker崩溃导致副本被踢出ISR时：检查下分区HW值是否需要更新是有必要的。
3. 生产者向Leader副本写消息时：因为写入消息会更新Leader的LEO，有必要检查HW值是否需要更新
4. Leader处理Follower FETCH请求时：首先从Log读取数据，之后尝试更新分区HW值

结论：

当Kafka broker都正常工作时，分区HW值的更新时机有两个：

1. Leader处理PRODUCE请求时
2. Leader处理FETCH请求时。

Leader如何更新自己的HW值？Leader broker上保存了一套Follower副本的LEO以及自己的LEO。当尝试确定分区HW时，它会选出所有**满足条件的副本**，比较它们的LEO(包括Leader的LEO)，并**选择最小的LEO值作为HW值**。

需要满足的条件，（二选一）：

1. 处于ISR中
2. 副本LEO落后于Leader LEO的时长不大于`replica.lag.time.max.ms`参数值(默认是10s)

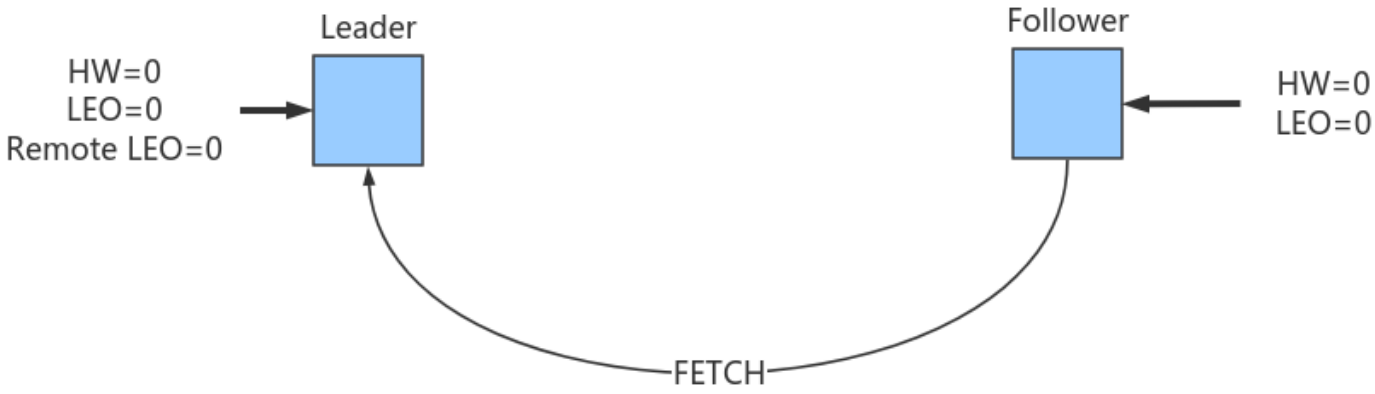
如果Kafka只判断第一个条件的话，确定分区HW值时就不会考虑这些未在ISR中的副本，但这些副本已经具备了“立刻进入ISR”的资格，因此就可能出现分区HW值越过ISR中副本LEO的情况——不允许。因为分区HW定义就是ISR中所有副本LEO的最小值。

六、HW和LEO正常更新案例

我们假设有一个topic，单分区，副本因子是2，即一个Leader副本和一个Follower副本。我们看下当producer发送一条消息时，broker端的副本到底会发生什么事情以及分区HW是如何被更新的。

1. **初始状态**

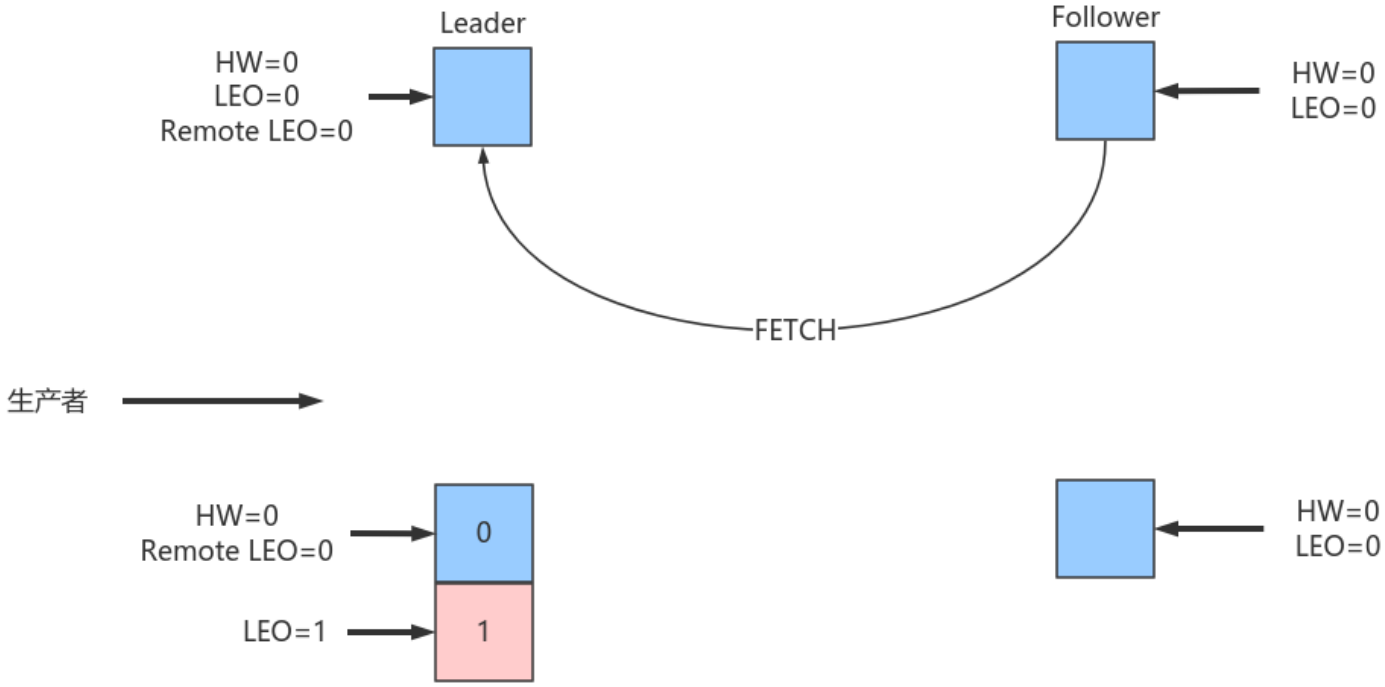
初始时Leader和Follower的HW和LEO都是0(严格来说源代码会初始化LEO为-1，不过这不影响之后的讨论)。Leader中的Remote LEO指的就是Leader端保存的Follower LEO，也被初始化成0。此时，生产者没有发送任何消息给Leader，而Follower已经开始不断地给Leader发送FETCH请求了，但因为没有数据因此什么都不会发生。值得一提的是，Follower发送过来的FETCH请求因为无数据而暂时会被寄存到Leader端的purgatory中，待500ms(`replica.fetch.wait.max.ms`参数)超时后会强制完成。倘若在寄存期间生产者发来数据，则Kafka会自动唤醒该FETCH请求，让Leader继续处理。



2. **Follower发送FETCH请求在Leader处理完PRODUCE请求之后**

producer给该topic分区发送了一条消息

此时的状态如下图所示：



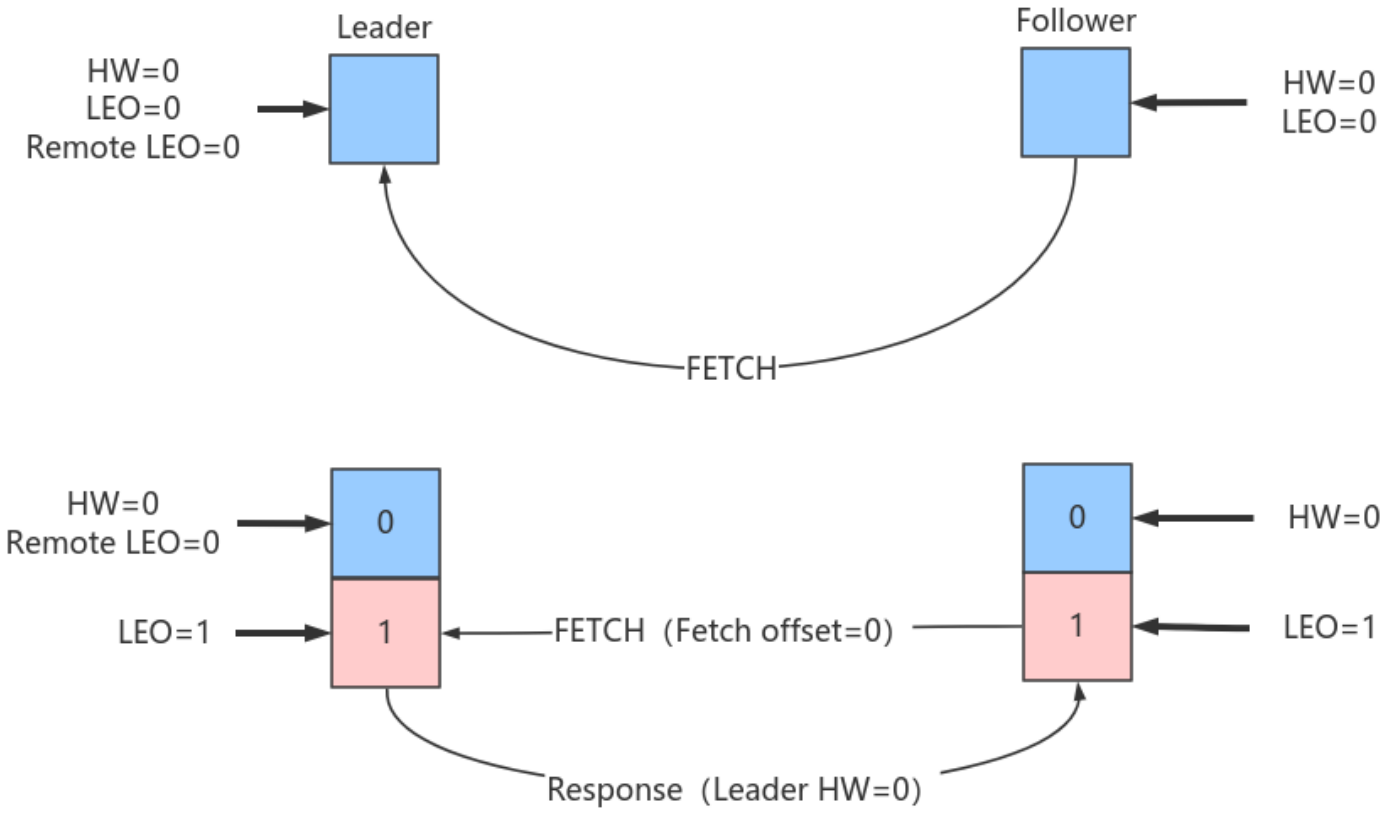
如上图所示，Leader接收到PRODUCE请求主要做两件事情：

1. 把消息写入Log，同时自动更新Leader自己的LEO
2. 尝试更新Leader HW值。假设此时Follower尚未发送FETCH请求，Leader端保存的Remote LEO依然是0，因此Leader会比较它自己的LEO值和Remote LEO值，发现最小值是0，与当前HW值相同，故不会更新分区HW值（仍为0）

PRODUCE请求处理完成后各值如下，Leader端的HW值依然是0，而LEO是1，Remote LEO也是0。

属性	阶段	旧值	新值	备注
Leader LEO	PRODUCE处理完成	0	1	写入了一条数据
Remote LEO	PRODUCE处理完成	0	0	还未Fetch
Leader HW	PRODUCE处理完成	0	0	min(LeaderLEO=1, RemoteLEO=0)=0
Follower LEO	PRODUCE处理完成	0	0	还未Fetch
Follower HW	PRODUCE处理完成	0	0	min(LeaderHW=0, FollowerLEO=0)=0

假设此时follower发送了FETCH请求，则状态变更如下：



本例中当follower发送FETCH请求时，Leader端的处理依次是：

1. 读取Log数据
2. 更新remote LEO = 0（为什么是0？ 因为此时Follower还没有写入这条消息。Leader如何确认Follower还未写入呢？这是通过Follower发来的FETCH请求中的Fetch offset来确定的）
3. 尝试更新分区HW：此时Leader LEO = 1，Remote LEO = 0，故分区HW值= min(Leader LEO, Follower Remote LEO) = 0
4. 把数据和当前分区HW值（依然是0）发送给Follower副本

而Follower副本接收到FETCH Response后依次执行下列操作：

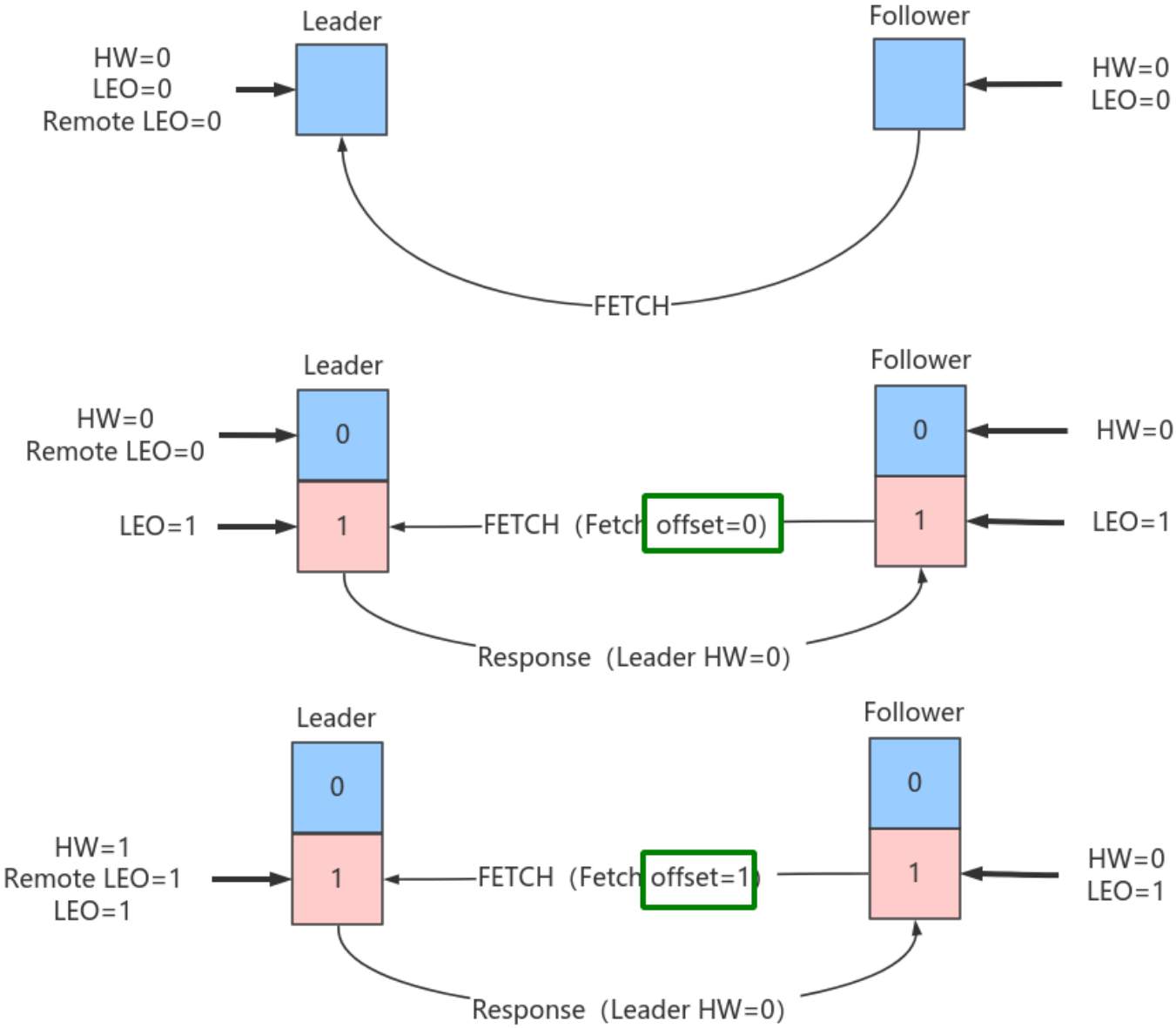
1. 写入本地Log，同时更新Follower自己管理的 LEO为1
2. 更新Follower HW：比较本地LEO和 FETCH Response 中的当前Leader HW值，取较小者，Follower HW = 0

此时，第一轮FETCH RPC结束，我们会发现虽然Leader和Follower都已经在Log中保存了这条消息，但分区HW值尚未被更新，仍为0。

属性	阶段	旧值	新值	备注
Leader LEO	PRODUCE和Follower FETCH处理完成	0	1	写入了一条数据
Remote LEO	PRODUCE和Follower FETCH处理完成	0	0	第一次fetch中offset为0
Leader HW	PRODUCE和Follower FETCH处理完成	0	0	min(LeaderLEO=1, RemoteLEO=0)=0
Follower LEO	PRODUCE和Follower FETCH处理完成	0	1	同步了一条数据
Follower HW	PRODUCE和Follower FETCH处理完成	0	0	min(LeaderHW=0, FollowerLEO=1)=0

Follower第二轮FETCH

分区HW是在第二轮FETCH RPC中被更新的，如下图所示：



Follower发来了第二轮FETCH请求，Leader端接收到后仍然会依次执行下列操作：

1. 读取Log数据
2. 更新Remote LEO = 1（这次为什么是1了？ 因为这轮FETCH RPC携带的fetch offset是1，那么为什么这轮携带的就是1了呢，因为上一轮结束后Follower LEO被更新为1了）
3. 尝试更新分区HW：此时leader LEO = 1，Remote LEO = 1，故分区HW值= min(Leader LEO, Follower Remote LEO) = **1**。
4. 把数据（实际上没有数据）和当前分区HW值（已更新为1）发送给Follower副本作为Response

同样地，Follower副本接收到FETCH response后依次执行下列操作：

1. 写入本地Log，当然没东西可写，Follower LEO也不会变化，依然是1。
2. 更新Follower HW：比较本地LEO和当前LeaderHW取小者。由于都是1，故更新follower HW = 1 。

属性	阶段	旧值	新值	备注
Leader LEO	第二次Follower FETCH处理完成	1	1	未写入新数据
Remote LEO	第二次Follower FETCH处理完成	0	1	第2次fetch中offset为1
Leader HW	第二次Follower FETCH处理完成	0	1	min(RemoteLEO,LeaderLEO)=1
Follower LEO	第二次Follower FETCH处理完成	1	1	未写入新数据
Follower HW	第二次Follower FETCH处理完成	0	1	第2次fetch resp中的LeaderHW和本地FollowerLEO都是1

此时消息已经成功地被复制到Leader和Follower的Log中且分区HW是1，表明消费者能够消费offset = 0的消息。

3. **FETCH请求保存在purgatory中，PRODUCE请求到来。**

当Leader无法立即满足FECTH返回要求的时候(比如没有数据)，那么该FETCH请求被暂存到Leader端的purgatory中（炼狱），待时机成熟尝试再次处理。Kafka不会无限期缓存，默认有个超时时间（500ms），一旦超时时间已过，则这个请求会被强制完成。当寄存期间还没超时，生产者发送PRODUCE请求从而使之满足了条件以致被唤醒。此时，Leader端处理流程如下：

1. Leader写Log（自动更新Leader LEO）
2. 尝试唤醒在purgatory中寄存的FETCH请求
3. 尝试更新分区HW

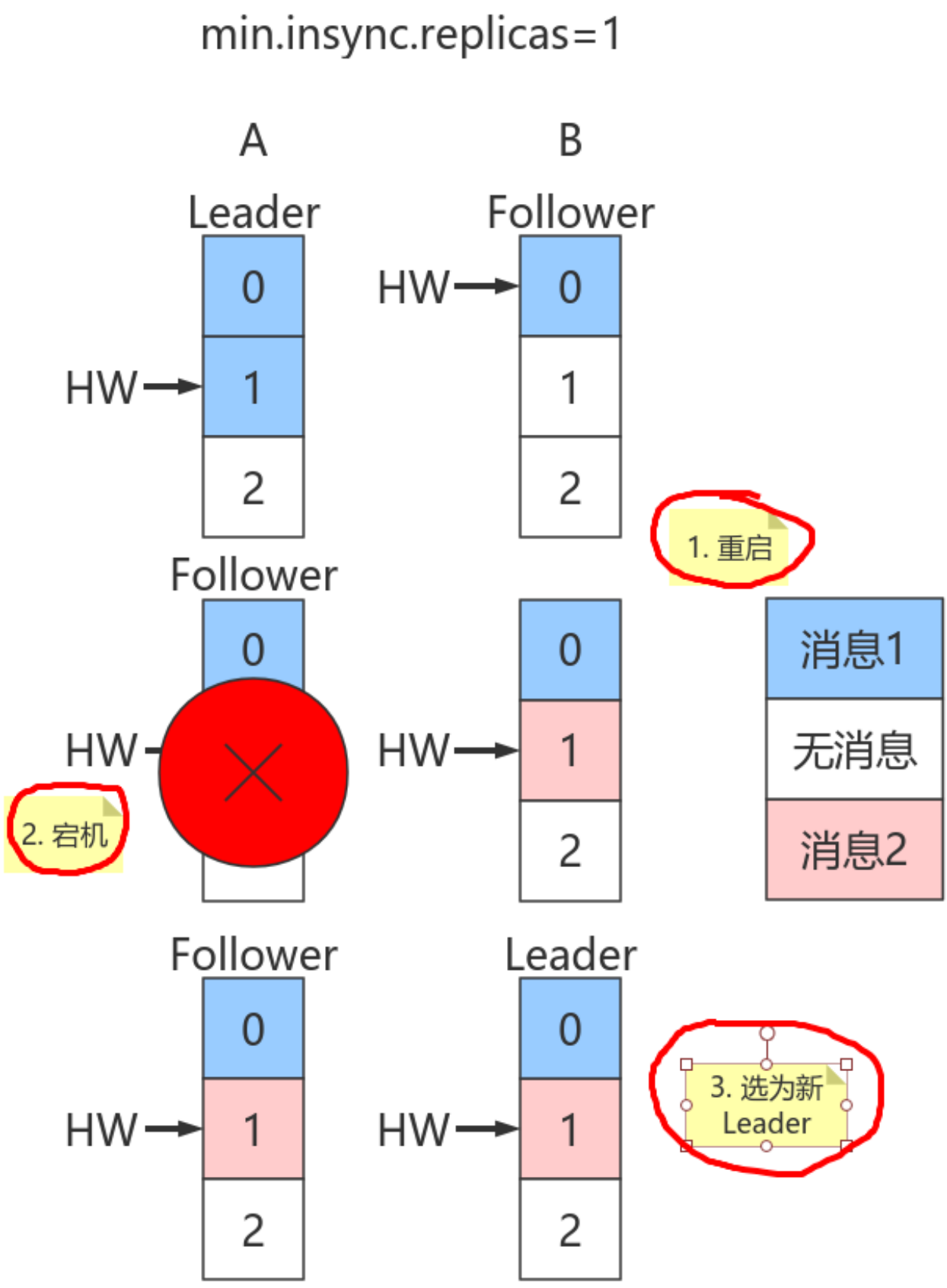
七、HW和LEO异常案例

Kafka使用HW值来决定副本备份的进度，而HW值的更新通常需要额外一轮FETCH RPC才能完成。但这种设计是有问题的，可能引起的问题包括：

1. 备份数据丢失
2. 备份数据不一致

1. **数据丢失**

使用HW值来确定备份进度时其值的更新是在**下一轮**RPC中完成的。如果Follower副本在标记上方的的第一步与第二步之间发生崩溃，那么就有可能造成数据的丢失。



上图中有两个副本：A和B。开始状态是A是Leader。

假设生产者`min.insync.replicas`为1，那么当生产者发送两条消息给A后，A写入Log，此时Kafka会通知生产者这两条消息写入成功。

代	属性	阶段	旧值	新值	备注
1	Leader LEO	PRODUCE和Follower FETCH处理完成	0	1	写入了一条数据
1	Remote LEO	PRODUCE和Follower FETCH处理完成	0	0	第一次fetch中offset为0
1	Leader HW	PRODUCE和Follower FETCH处理完成	0	0	min(LeaderLEO=1,FollowerLEO=0)=0
1	Follower LEO	PRODUCE和Follower FETCH处理完成	0	1	同步了一条数据
1	Follower HW	PRODUCE和Follower FETCH处理完成	0	0	min(LeaderHW=0, FollowerLEO=1)=0
2	Leader LEO	第二次Follower FETCH处理完成	1	2	写入了第二条数据
2	Remote LEO	第二次Follower FETCH处理完成	0	1	第2次fetch中offset为1
2	Leader HW	第二次Follower FETCH处理完成	0	1	min(RemoteLEO=1,LeaderLEO=2)=1
2	Follower LEO	第二次Follower FETCH处理完成	1	2	写入了第二条数据
2	Follower HW	第二次Follower FETCH处理完成	0	1	min(LeaderHW=1,FollowerLEO=2)=1
3	Leader LEO	第三次Follower FETCH处理完成	2	2	未写入新数据
3	Remote LEO	第三次Follower FETCH处理完成	1	2	第3次fetch中offset为2
3	Leader HW	第三次Follower FETCH处理完成	1	2	min(RemoteLEO=2,LeaderLEO)=2
3	Follower LEO	第三次Follower FETCH处理完成	2	2	未写入新数据
3	Follower HW	第三次Follower FETCH处理完成	1	2	第3次fetch resp中的LeaderHW和本地FollowerLEO都是2

但是在broker端，Leader和Follower的Log虽都写入了2条消息且分区HW已经被更新到2，但Follower HW尚未被更新还是1，也就是上面标记的第二步尚未执行，表中最后一条未执行。

倘若此时副本B所在的broker宕机，那么重启后B会自动把LEO调整到之前的HW值1，故副本B会做日志截断(log truncation)，将offset = 1的那条消息从log中删除，并调整LEO = 1。此时follower副本底层log中就只有一条消息，即offset = 0的消息！

B重启之后需要给A发FETCH请求，但若A所在broker机器在此时宕机，那么Kafka会令B成为新的Leader，而当A重启回来后也会执行日志截断，将HW调整回1。这样，offset=1的消息就从两个副本的log中被删除，也就是说这条已经被生产者认为发送成功的数据丢失。

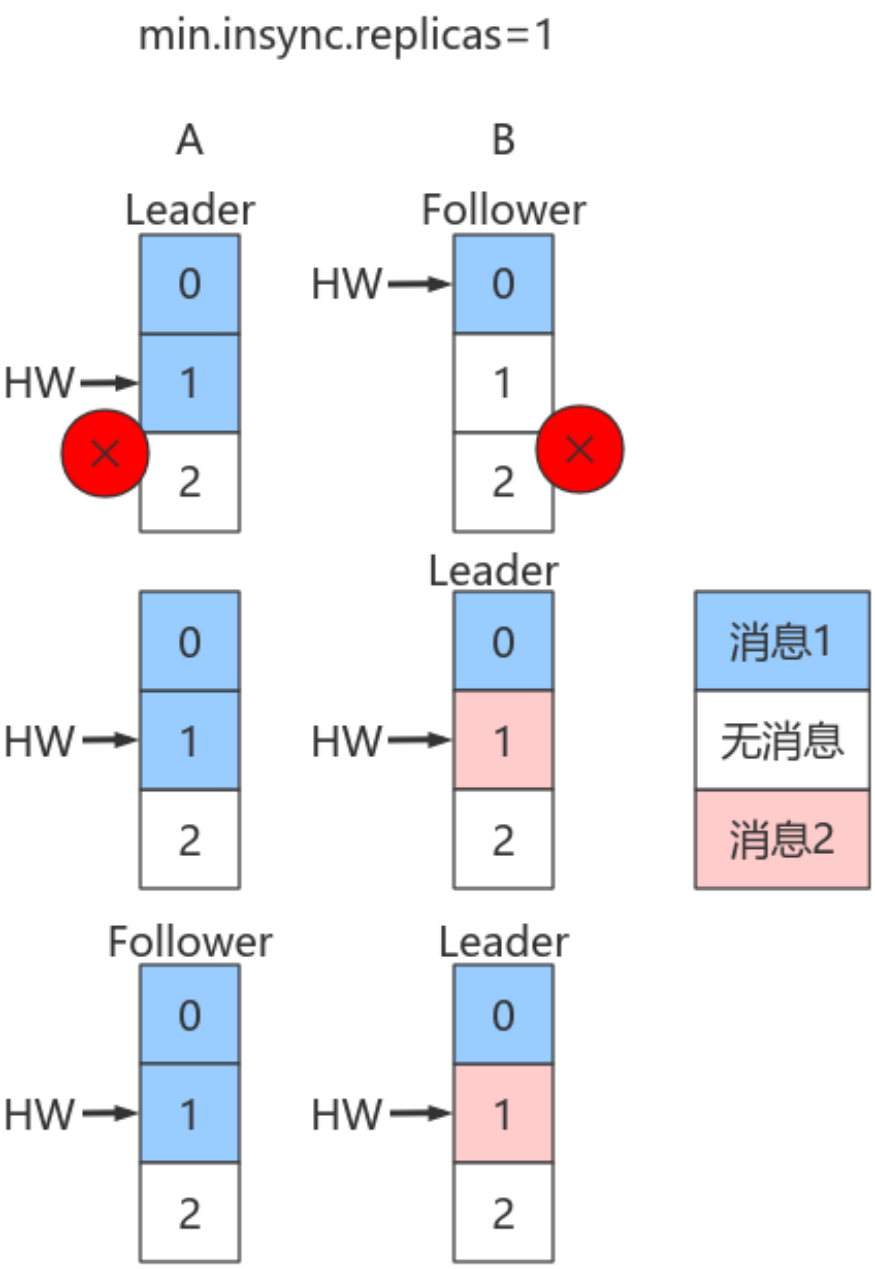
丢失数据的前提是`min.insync.replicas=1`时，一旦消息被写入Leader端Log即被认为是`committed`。**延迟一轮`FETCH RPC`更新HW值的设计使follower HW值是异步延迟更新**，若在这个过程中Leader发生变更，那么成为新Leader的Follower的HW值就有可能是过期的，导致生产者本是成功提交的消息被删除。

2. **Leader和Follower数据离散**

除了可能造成的数据丢失以外，该设计还会造成Leader的Log和Follower的Log数据不一致。

如Leader端记录序列：m1,m2,m3,m4,m5,...；Follower端序列可能是m1,m3,m4,m5,...。

看图：



假设：A是Leader，A的Log写入了2条消息，但B的Log只写了1条消息。分区HW更新到2，但B的HW还是1，同时生产者`min.insync.replicas`仍然为1。

假设A和B所在Broker同时宕机，B先重启回来，因此B成为Leader，分区HW = 1。假设此时生产者发送了第3条消息(红色表示)给B，于是B的log中offset = 1的消息变成了红框表示的消息，同时分区HW更新到2（A还没有回来，就B一个副本，故可以直接更新HW而不用理会A）之后A重启回来，需要**执行日志截断**，但发现此时分区HW=2而A之前的HW值也是2，故**不做任何调整**。此后A和B将以这种状态继续正常工作。

显然，这种场景下，A和B的Log中保存在offset = 1的消息是不同的记录，从而引发不一致的情形出现。

八、Leader Epoch使用

0. **Kafka解决方案**

造成上述两个问题的根本原因在于

1. HW值被用于衡量副本备份的成功与否。
2. 在出现失败重启时作为日志截断的依据。

但HW值的更新是异步延迟的，特别是需要额外的FETCH请求处理流程才能更新，故这中间发生的任何崩溃都可能导致HW值的过期。

Kafka从0.11引入了`leader epoch`来取代HW值。Leader端使用**内存**保存Leader的epoch信息，即使出现上面的两个场景也能规避这些问题。

所谓Leader epoch实际上是一对值：<epoch, offset>：

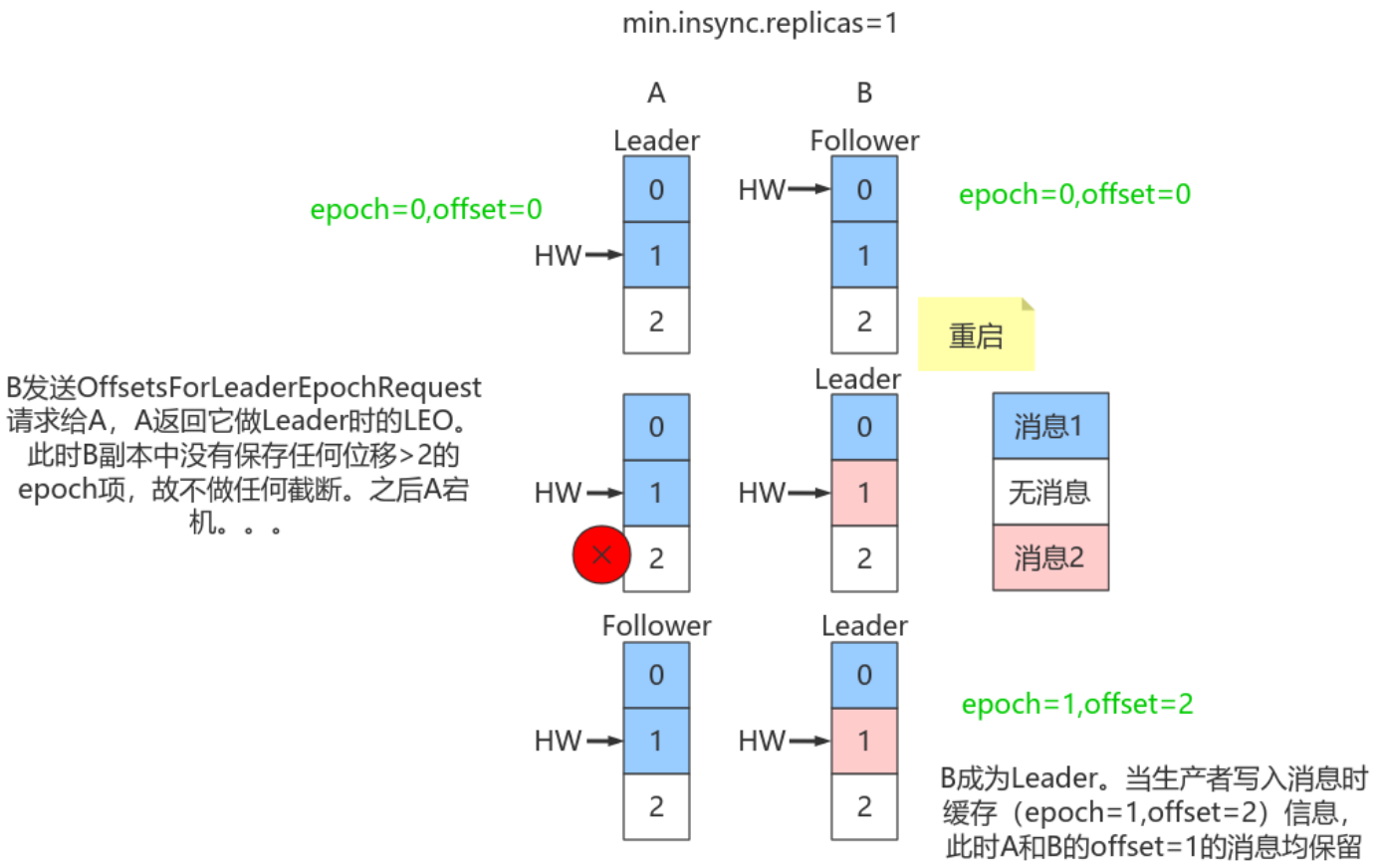
1. epoch表示Leader的版本号，从0开始，Leader变更过1次，epoch+1
2. offset对应于该epoch版本的Leader写入第一条消息的offset。因此假设有两对值：

```
<0, 0>
<1, 120>
```

则表示第一个Leader从位移0开始写入消息；共写了120条[0, 119]；而第二个Leader版本号是1，从位移120处开始写入消息。

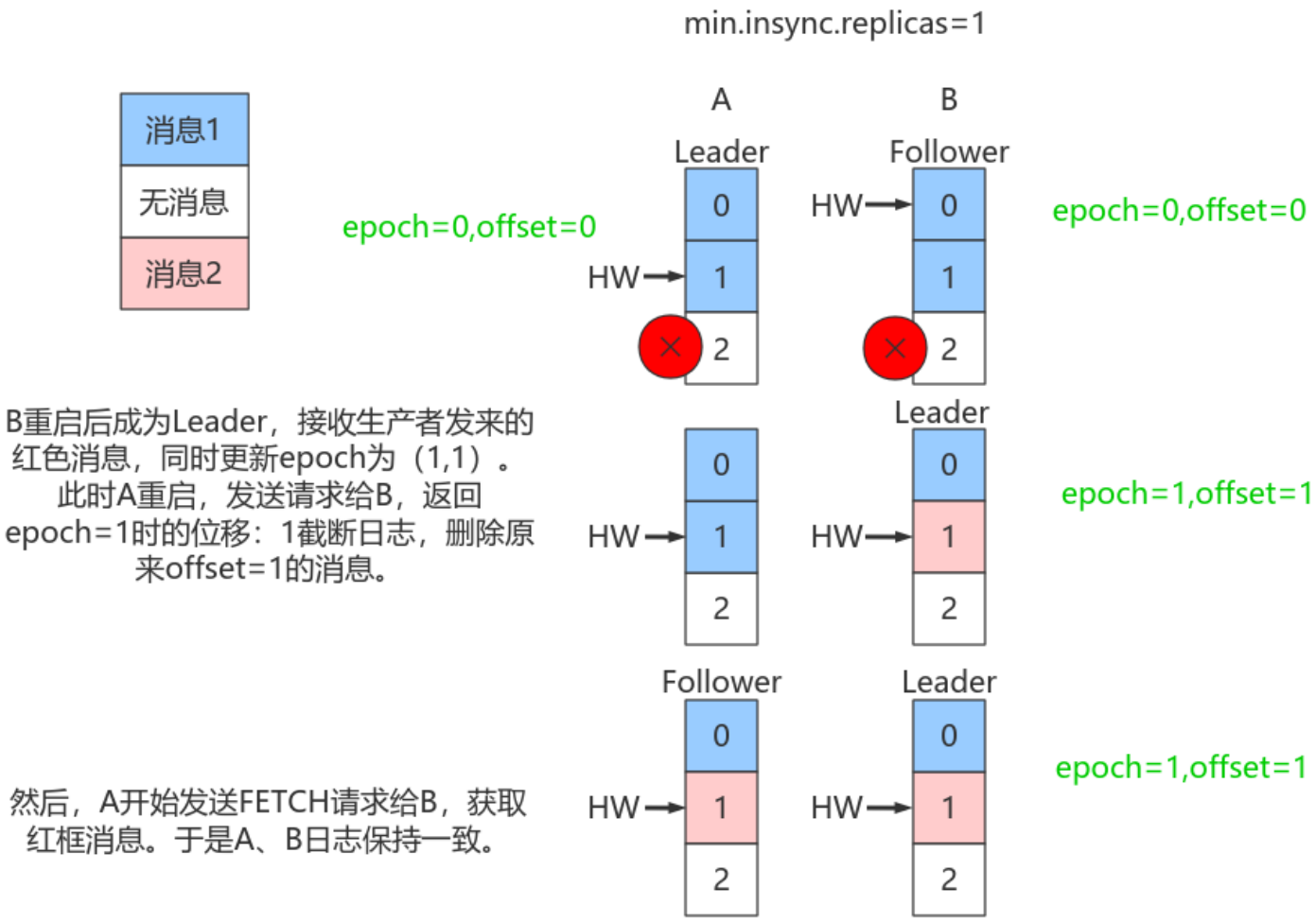
1. Leader broker中会保存这样的一个缓存，并定期地写入到一个`checkpoint`文件中。
2. 当Leader写Log时它会尝试更新整个缓存：如果这个Leader首次写消息，则会在缓存中增加一个条目；否则就不做更新。
3. 每次副本变为Leader时会查询这部分缓存，获取出对应Leader版本的位移，则不会发生数据不一致和丢失的情况。

1. **规避数据丢失**



只需要知道每个副本都引入了新的状态来保存自己当leader时开始写入的第一条消息的offset以及leader版本。这样在恢复的时候完全使用这些信息而非HW来判断是否需要截断日志。

2. **规避数据不一致**



依靠Leader epoch的信息可以有效地规避数据不一致的问题。

2.6.5 消息重复的场景及解决方案

消息重复和丢失是kafka中很常见的问题，主要发生在以下三个阶段：

1. 生产者阶段
2. broke阶段
3. 消费者阶段

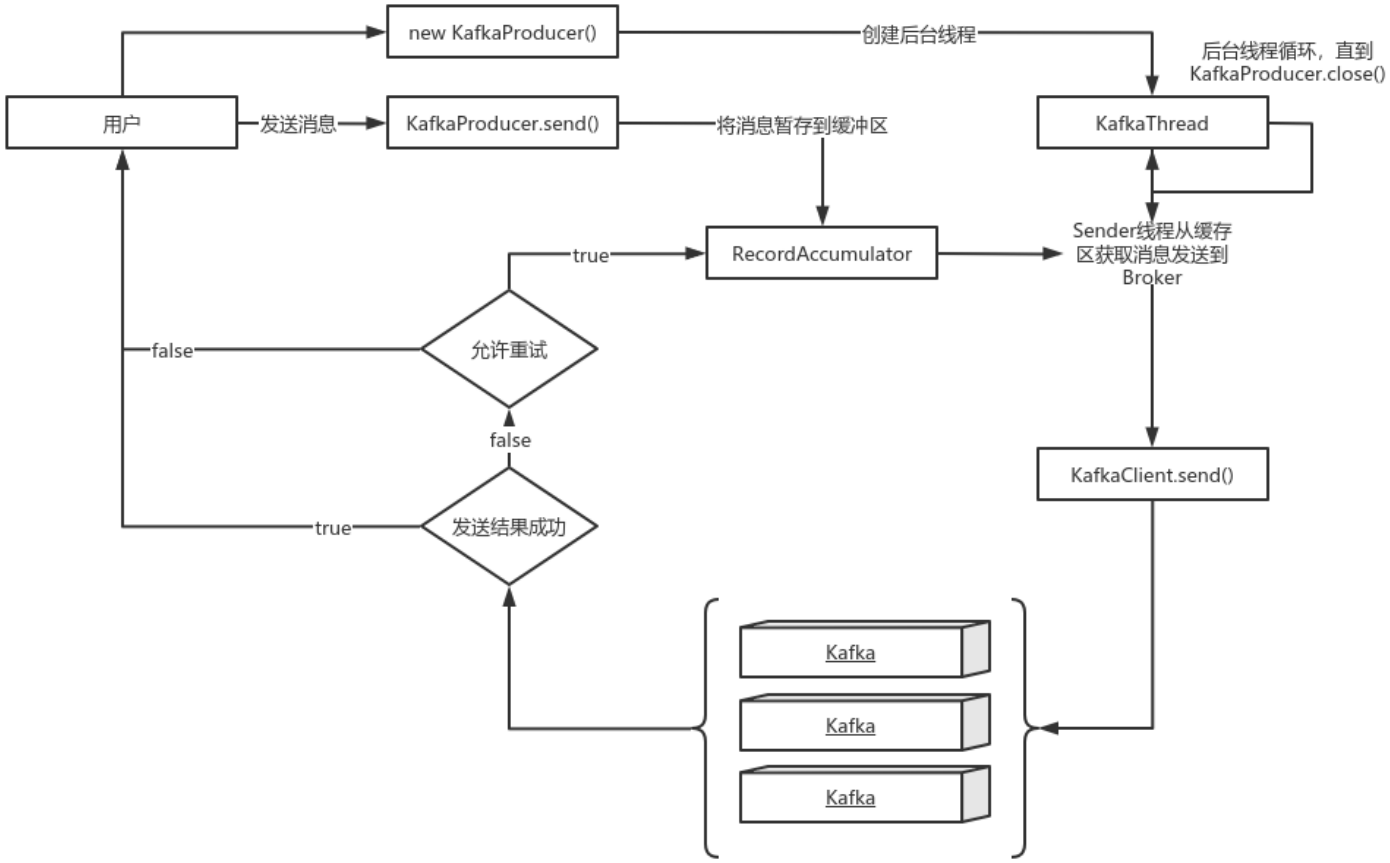
2.6.5.1 生产者阶段重复场景

2.6.5.1.1 根本原因

生产发送的消息没有收到正确的broke响应，导致生产者重试。

生产者发出一条消息，broke落盘以后因为网络等种种原因发送端得到一个发送失败的响应或者网络中断，然后生产者收到一个可恢复的Exception重试消息导致消息重复。

2.6.5.1.2 重试过程

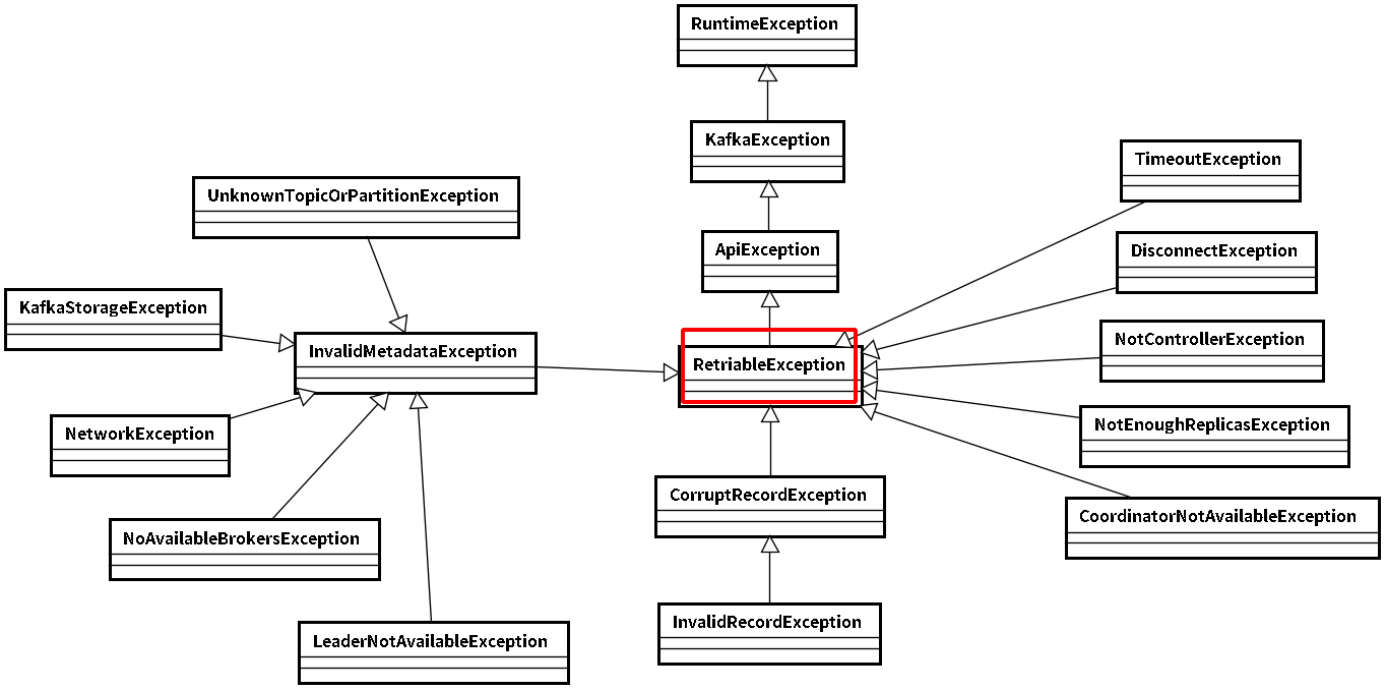


说明：

1. new KafkaProducer()后创建一个后台线程KafkaThread扫描RecordAccumulator中是否有消息；
2. 调用KafkaProducer.send()发送消息，实际上只是把消息保存到RecordAccumulator中；
3. 后台线程KafkaThread扫描到RecordAccumulator中有消息后，将消息发送到kafka集群；
4. 如果发送成功，那么返回成功；
5. 如果发送失败，那么判断是否允许重试。如果不允许重试，那么返回失败的结果；如果允许重试，把消息再保存到RecordAccumulator中，等待后台线程KafkaThread扫描再次发送；

2.6.5.1.3 可恢复异常说明

异常是RetriableException类型或者TransactionManager允许重试；RetriableException类继承关系如下：



2.6.5.1.4 记录顺序问题

如果设置`max.in.flight.requests.per.connection`大于1（默认5，单个连接上发送的未确认请求的最大数量，表示上一个发出的请求没有确认下一个请求又发出了）。大于1可能会改变记录的顺序，因为如果将两个batch发送到单个分区，第一个batch处理失败并重试，但是第二个batch处理成功，那么第二个batch处理中的记录可能先出现被消费。

设置`max.in.flight.requests.per.connection`为1，可能会影响吞吐量，可以解决单个生产者发送顺序问题。如果多个生产者，生产者1先发送一个请求，生产者2后发送请求，此时生产者1返回可恢复异常，重试一定次数成功了。虽然生产者1先发送消息，但生产者2发送的消息会被先消费。

2.6.5.2 生产者发送重复解决方案

2.6.5.2.1 启动kafka的幂等性

要启动kafka的幂等性，设置：`enable.idempotence=true`，以及`ack=all`以及`retries > 1`。

2.6.5.2.2 ack=0，不重试。

可能会丢消息，适用于吞吐量指标重要性高于数据丢失，例如：日志收集。

2.6.5.3 生产者和broker阶段消息丢失场景

2.6.5.3.1 ack=0，不重试

生产者发送消息完，不管结果了，如果发送失败也就丢失了。

2.6.5.3.2 ack=1，leader crash

生产者发送消息完，只等待Leader写入成功就返回了，Leader分区丢失了，此时Follower没来及同步，消息丢失。

2.6.5.3.3 `unclean.leader.election.enable` 配置true

允许选举ISR以外的副本作为leader,会导致数据丢失，默认为false。生产者发送异步消息，只等待Lead写入成功就返回，Leader分区丢失，此时ISR中没有Follower，Leader从OSR中选举，因为OSR中本来落后于Leader造成消息丢失。

2.6.5.4 解决生产者和broker阶段消息丢失

2.6.5.4.1 禁用unclean选举，ack=all

`ack=all / -1,tries > 1,unclean.leader.election.enable : false`

生产者发完消息，等待Follower同步完再返回，如果异常则重试。副本的数量可能影响吞吐量，不超过5个，一般三个。

不允许unclean Leader选举。

2.6.5.4.2 配置：min.insync.replicas > 1

当生产者将`acks`设置为`all`(或`-1`)时，`min.insync.replicas>1`。指定确认消息写成功需要的最小副本数量。达不到这个最小值，生产者将引发一个异常(要么是NotEnoughReplicas，要么是NotEnoughReplicasAfterAppend)。

当一起使用时，`min.insync.replicas`和`ack`允许执行更大的持久性保证。一个典型的场景是创建一个复制因子为3的主题，设置min.insync复制到2个，用`all`配置发送。将确保如果大多数副本没有收到写操作，则生产者将引发异常。

2.6.5.4.3 失败的offset单独记录

生产者发送消息，会自动重试，遇到不可恢复异常会抛出，这时可以捕获异常记录到数据库或缓存，进行单独处理。

2.6.5.5 消费者数据重复场景及解决方案

2.6.5.5.1 根本原因

数据消费完没有及时提交offset到broker。

2.6.5.5.2 场景

消息消费端在消费过程中挂掉没有及时提交offset到broke，另一个消费端启动拿之前记录的offset开始消费，由于offset的滞后性可能会导致新启动的客户端有少量重复消费。

2.6.5.6 解决方案

2.6.5.6.1 取消自动提交

每次消费完或者程序退出时手动提交。这可能也没法保证一条重复。

2.6.5.6.2 下游做幂等

一般是让下游做幂等或者尽量每消费一条消息都记录offset，对于少数严格的场景可能需要把offset或唯一ID（例如订单ID）和下游状态更新放在同一个数据库里面做事务来保证精确的一次更新或者在下游数据表里面同时记录消费offset，然后更新下游数据的时候用消费位移做乐观锁拒绝旧位移的数据更新。

2.6.6 __consumer_offsets

Zookeeper不适合大批量的频繁写入操作。

Kafka 1.0.2将consumer的位移信息保存在Kafka内部的topic中，即__consumer_offsets主题，并且默认提供了kafka_consumer_groups.sh脚本供用户查看consumer信息。

1. 创建topic “tp_test_01”

```
[root@node1 ~]# kafka-topics.sh --zookeeper node1:2181/myKafka --create --topic tp_test_01 --partitions 5 --replication-factor 1
```

2. 使用kafka-console-producer.sh脚本生产消息

```
[root@node1 ~]# for i in `seq 100`; do echo "hello lagou $i" >> messages.txt; done
[root@node1 ~]# kafka-console-producer.sh --broker-list node1:9092 --topic tp_test_01 < messages.txt
```

由于默认没有指定key，所以根据round-robin方式，消息分布到不同的分区上。(**本例中生产了100条消息**)

3. 验证消息生产成功

```
[root@node1 ~]#  kafka-console-producer.sh --broker-list node1:9092 --topic tp_test_01 < messages.txt
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>[root@node1 ~]# kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list node1:9092 --topic tp_test_01 --time -1
tp_test_01:2:20
tp_test_01:4:20
tp_test_01:1:20
tp_test_01:3:20
tp_test_01:0:20
[root@node1 ~]#
```

结果输出表明100条消息全部生产成功!

4. 创建一个console consumer group

```
[root@node1 ~]#kafka-console-consumer.sh --bootstrap-server node1:9092 --topic tp_test_01 --from-beginning
```

5. 获取该consumer group的group id(后面需要根据该id查询它的位移信息)

```
[root@node1 ~]# kafka-consumer-groups.sh --bootstrap-server node1:9092 --list
```

输出： console-consumer-49366（记住这个id！）

6. 查询__consumer_offsets topic所有内容

注意：运行下面命令前先要在consumer.properties中设置exclude.internal.topics=false

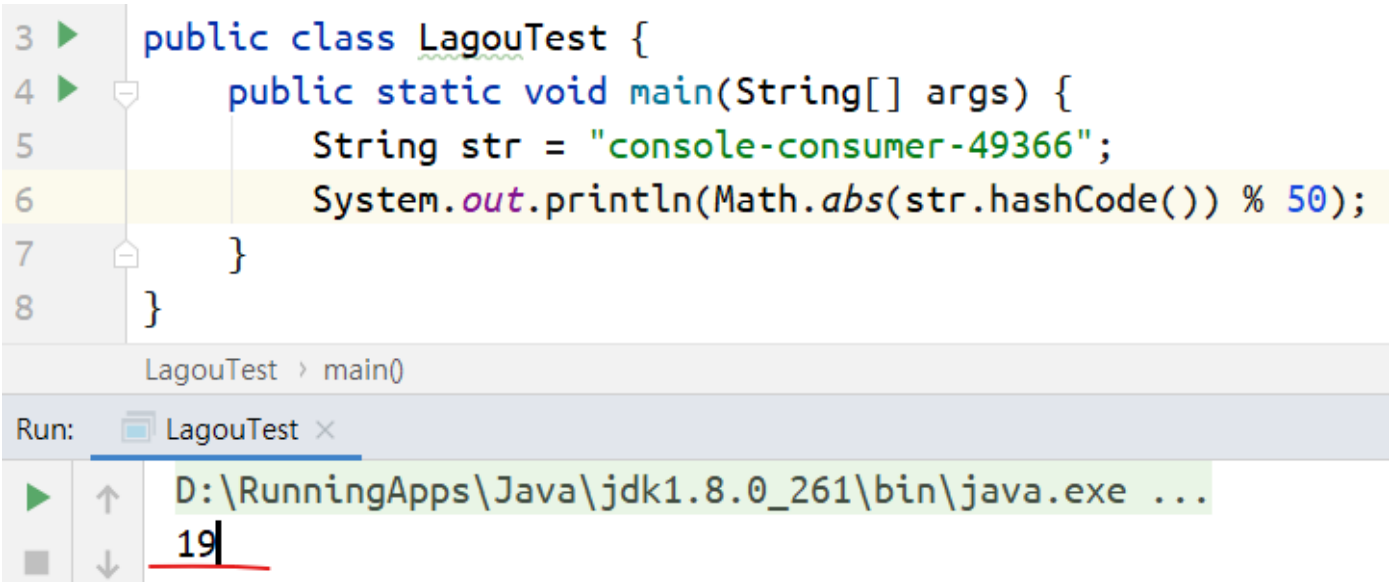
```
[root@node1 ~]# kafka-console-consumer.sh --topic __consumer_offsets --bootstrap-server node1:9092 --formatter "kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter" --consumer.config config/consumer.properties --from-beginning
```

默认情况下__consumer_offsets有50个分区，如果你的系统中consumer group也很多的话，那么这个命令的输出结果会很多。

7. 计算指定consumer group在__consumer_offsets topic中分区信息

这时候就用到了第5步获取的group.id(本例中是console-consumer-49366)。Kafka会使用下面公式计算该group位移保存在__consumer_offsets的哪个分区上：

```
Math.abs(groupID.hashCode()) % numPartitions
```



对应的分区=Math.abs("console-consumer-49366".hashCode()) % 50 = 19，即__consumer_offsets的分区19保存了这个consumer group的位移信息。

8. 获取指定consumer group的位移信息

```
[root@node1 ~]# kafka-simple-consumer-shell.sh --topic __consumer_offsets --partition 19 --broker-list node1:9092 --formatter "kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter"
```

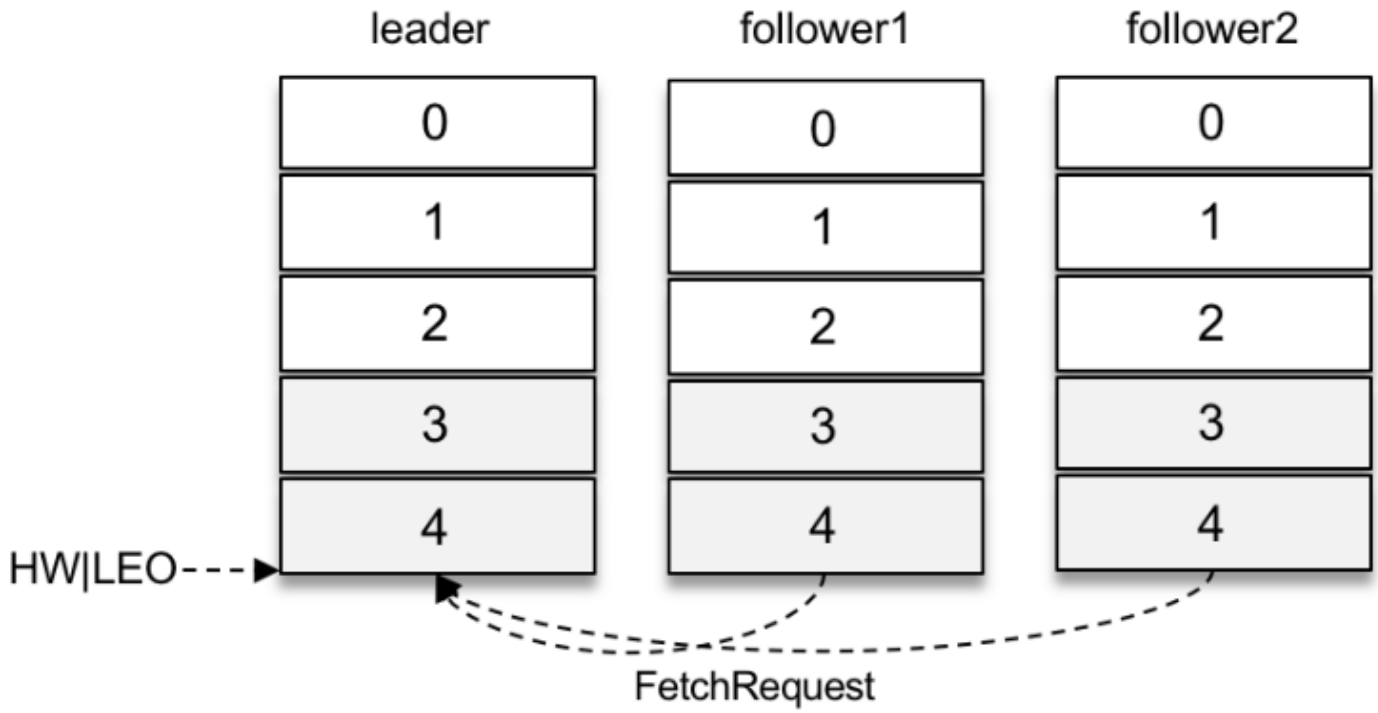
下面是输出结果：

```
...
[console-consumer-49366,tp_test_01,3]::[OffsetMetadata[20,NO_METADATA],CommitTime 1596424702212,ExpirationTime 1596511102212]
[console-consumer-49366,tp_test_01,4]::[OffsetMetadata[20,NO_METADATA],CommitTime 1596424702212,ExpirationTime 1596511102212]
[console-consumer-49366,tp_test_01,0]::[OffsetMetadata[20,NO_METADATA],CommitTime 1596424702212,ExpirationTime 1596511102212]
[console-consumer-49366,tp_test_01,1]::[OffsetMetadata[20,NO_METADATA],CommitTime 1596424702212,ExpirationTime 1596511102212]
[console-consumer-49366,tp_test_01,2]::[OffsetMetadata[20,NO_METADATA],CommitTime 1596424702212,ExpirationTime 1596511102212]
[console-consumer-49366,tp_test_01,3]::[OffsetMetadata[20,NO_METADATA],CommitTime 1596424707212,ExpirationTime 1596511107212]
[console-consumer-49366,tp_test_01,4]::[OffsetMetadata[20,NO_METADATA],CommitTime 1596424707212,ExpirationTime 1596511107212]
[console-consumer-49366,tp_test_01,0]::[OffsetMetadata[20,NO_METADATA],CommitTime 1596424707212,ExpirationTime 1596511107212]
 ...
```

上图可见，该consumer group果然保存在分区11上，且位移信息都是对的(这里的位移信息是已消费的位移，严格来说不是第3步中的位移。由于我的consumer已经消费完了所有的消息，所以这里的位移与第3步中的位移相同)。另外，可以看到__consumer_offsets topic的每一日志项的格式都是：[Group, Topic, Partition]::[OffsetMetadata[Offset, Metadata], CommitTime, ExpirationTime]。

2.7 延时队列

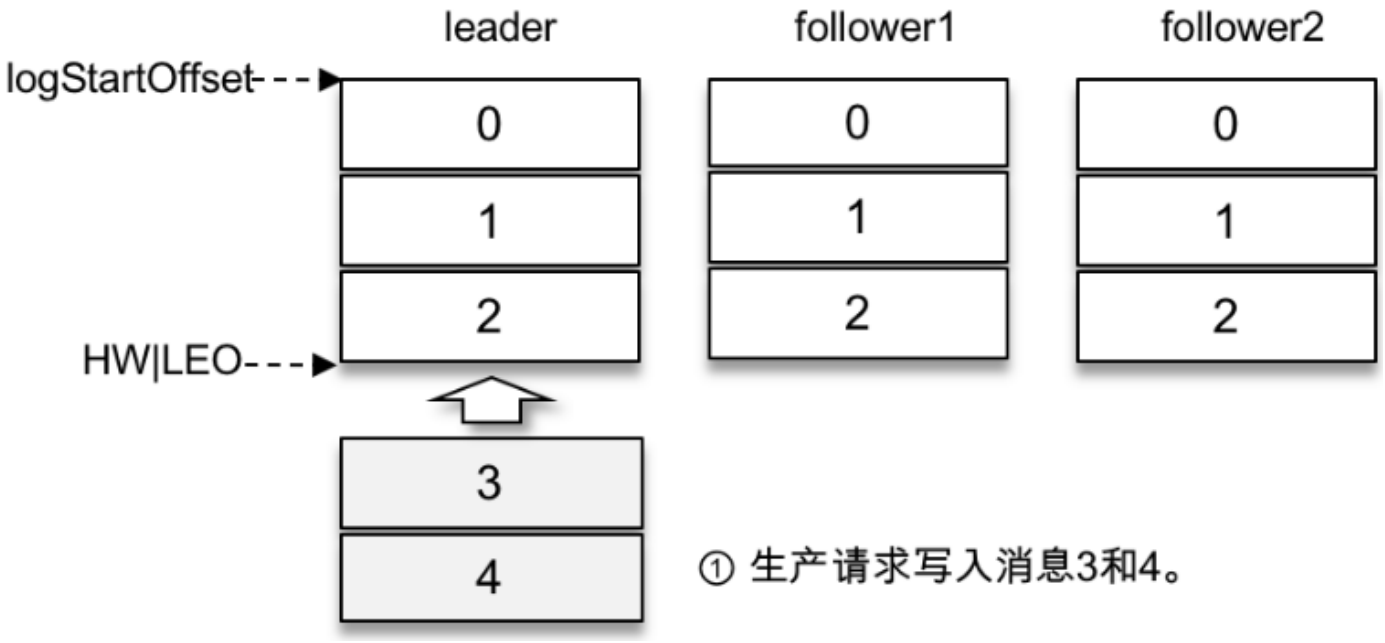
两个follower副本都已经拉取到了leader副本的最新位置，此时又向leader副本发送拉取请求，而leader副本并没有新的消息写入，那么此时leader副本该如何处理呢？可以直接返回空的拉取结果给follower副本，不过在leader副本一直没有新消息写入的情况下，follower副本会一直发送拉取请求，并且总收到空的拉取结果，消耗资源。



Kafka在处理拉取请求时，会先读取一次日志文件，如果收集不到足够多（fetchMinBytes，由参数fetch.min.bytes配置，默认值为1）的消息，那么就会创建一个延时拉取操作（DelayedFetch）以等待拉取到足够数量的消息。当延时拉取操作执行时，会再读取一次日志文件，然后将拉取结果返回给follower副本。

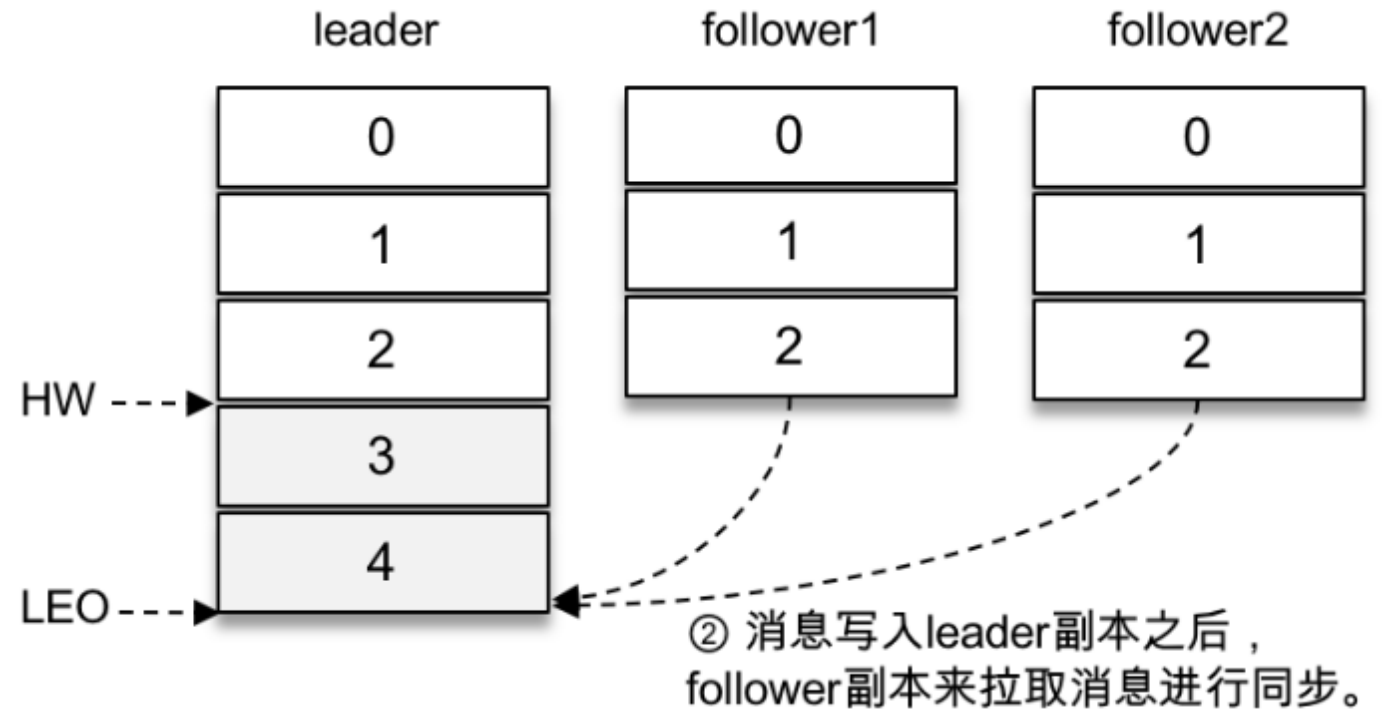
延迟操作不只是拉取消息时的特有操作，在Kafka中有多种延时操作，比如延时数据删除、延时生产等。

对于延时生产（消息）而言，如果在使用生产者客户端发送消息的时候将acks参数设置为-1，那么就意味着需要等待ISR集合中的所有副本都确认收到消息之后才能正确地收到响应的结果，或者捕获超时异常。



假设某个分区有3个副本：leader、follower1和follower2，它们都在分区的ISR集合中。不考虑ISR变动的情况，Kafka在收到客户端的生产请求后，将消息3和消息4写入leader副本的本地日志文件。

由于客户端设置了acks为-1，那么需要等到follower1和follower2两个副本都收到消息3和消息4后才能告知客户端正确地接收了所发送的消息。如果在一定的时间内，follower1副本或follower2副本没能够完全拉取到消息3和消息4，那么就需要返回超时异常给客户端。生产请求的超时时间由参数request.timeout.ms配置，默认值为30000，即30s。



那么这里等待消息3和消息4写入follower1副本和follower2副本，并返回相应的响应结果给客户端的动作是由谁来执行的呢？在将消息写入leader副本的本地日志文件之后，Kafka会创建一个延时的生产操作（DelayedProduce），用来处理消息正常写入所有副本或超时的情况，以返回相应的响应结果给客户端。

延时操作需要延时返回响应的结果，首先它必须有一个超时时间（delayMs），如果在这个超时时间内没有完成既定的任务，那么就需要强制完成以返回响应结果给客户端。其次，延时操作不同于定时操作，定时操作是指在特定时间之后执行的操作，而延时操作可以在所设定的超时时间之前完成，所以延时操作能够支持外部事件的触发。

就延时生产操作而言，它的外部事件是所要写入消息的某个分区的HW（高水位）发生增长。也就是说，随着follower副本不断地与leader副本进行消息同步，进而促使HW进一步增长，HW每增长一次都会检测是否能够完成此次延时生产操作，如果可以就执行以此返回响应结果给客户端；如果在超时时间内始终无法完成，则强制执行。

延时拉取操作，是由超时触发或外部事件触发而被执行的。超时触发很好理解，就是等到超时时间之后触发第二次读取日志文件的操作。外部事件触发就稍复杂了一些，因为拉取请求不单单由follower副本发起，也可以由消费者客户端发起，两种情况所对应的外部事件也是不同的。如果是follower副本的延时拉取，它的外部事件就是消息追加到了leader副本的本地日志文件中；如果是消费者客户端的延时拉取，它的外部事件可以简单地理解为HW的增长。

时间轮实现延时队列。

TimeWheel。size，每个单元格的时间，每个单元格都代表一个时间，size*每个单元格的时间就是一个周期。

2.8 重试队列

kafka没有重试机制不支持消息重试，也没有死信队列，因此使用kafka做消息队列时，需要自己实现消息重试的功能。

实现

创建新的kafka主题作为重试队列：

1. 创建一个topic作为重试topic，用于接收等待重试的消息。
2. 普通topic消费者设置待重试消息的下一个重试topic。
3. 从重试topic获取待重试消息储存到redis的zset中，并以下一次消费时间排序
4. 定时任务从redis获取到达消费事件的消息，并把消息发送到对应的topic
5. 同一个消息重试次数过多则不再重试

代码实现

1. 新建springboot项目

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.8.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.lagou.kafka.demo</groupId>
    <artifactId>demo-retryqueue</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>demo-retryqueue</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>

        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
            <version>1.2.73</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>

```

2. 添加application.properties

```
# bootstrap.servers
spring.kafka.bootstrap-servers=node1:9092
# key序列化器
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
# value序列化器
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer

# 消费组id：group.id
spring.kafka.consumer.group-id=retryGroup
# key反序列化器
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
# value反序列化器
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer

# redis数据库编号
spring.redis.database=0
# redis主机地址
spring.redis.host=node1
# redis端口
spring.redis.port=6379
# Redis服务器连接密码（默认为空）

```

```
spring.redis.password=
# 连接池最大连接数（使用负值表示没有限制）
spring.redis.jedis.pool.max-active=20
# 连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.jedis.pool.max-wait=-1
# 连接池中的最大空闲连接
spring.redis.jedis.pool.max-idle=10
# 连接池中的最小空闲连接
spring.redis.jedis.pool.min-idle=0
# 连接超时时间（毫秒）
spring.redis.timeout=1000

# Kafka主题名称
spring.kafka.topics.test=tp_demo_retry_01
# 重试队列
spring.kafka.topics.retry=tp_demo_retry_02
```

3. RetryqueueApplication.java

```
package com.lagou.kafka.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RetryqueueApplication {

    public static void main(String[] args) {
        SpringApplication.run(RetryqueueApplication.class, args);
    }

}

```

4. AppConfig.java

```
package com.lagou.kafka.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@Configuration
public class AppConfig {
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
factory) {

        RedisTemplate<String, Object> template = new RedisTemplate<>();
```

```
        // 配置连接工厂
        template.setConnectionFactory(factory);

        return template;
    }

}

```

5. KafkaController.java

```
package com.lagou.kafka.demo.controller;

import com.lagou.kafka.demo.service.KafkaService;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.ExecutionException;

@RestController
public class KafkaController {

    @Autowired
    private KafkaService kafkaService;

    @Value("${spring.kafka.topics.test}")
    private String topic;

    @RequestMapping("/send/{message}")
    public String sendMessage(@PathVariable String message) throws
ExecutionException, InterruptedException {

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

        String result = kafkaService.sendMessage(record);

        return result;
    }

}

```

6. KafkaService.java

```java
package com.lagou.kafka.demo.service;

import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;

import java.util.concurrent.ExecutionException;

@Service
public class KafkaService {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public String sendMessage(ProducerRecord<String, String> record) throws ExecutionException, InterruptedException {

        SendResult<String, String> result = this.kafkaTemplate.send(record).get();
        RecordMetadata metadata = result.getRecordMetadata();
        String returnResult = metadata.topic() + "\t" + metadata.partition() + "\t" + metadata.offset();
        System.out.println("发送消息成功: " + returnResult);

        return returnResult;
    }

}

```

7. ConsumerListener.java

```java
package com.lagou.kafka.demo.listener;

import com.lagou.kafka.demo.service.KafkaRetryService;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Component
public class ConsumerListener {

    private static final Logger log = LoggerFactory.getLogger(ConsumerListener.class);
```

```
15
16      @Autowired
17      private KafkaRetryService kafkaRetryService;
18
19      private static int index = 0;
20
21      @KafkaListener(topics = "${spring.kafka.topics.test}", groupId =
    "${spring.kafka.consumer.group-id}")
22      public void consume(ConsumerRecord<String, String> record) {
23          try {
24              // 业务处理
25              log.info("消费的消息：" + record);
26              index++;
27              if (index % 2 == 0) {
28                  throw new Exception("该重发了");
29              }
30          } catch (Exception e) {
31              log.error(e.getMessage());
32              // 消息重试
33              kafkaRetryService.consumerLater(record);
34          }
35      }
36
37  }
```

8. KafkaRetryService.java

```
1   package com.lagou.kafka.demo.service;
2
3   import com.alibaba.fastjson.JSON;
4   import com.lagou.kafka.demo.entity.RetryRecord;
5   import org.apache.kafka.clients.consumer.ConsumerRecord;
6   import org.apache.kafka.common.header.Header;
7   import org.slf4j.Logger;
8   import org.slf4j.LoggerFactory;
9   import org.springframework.beans.factory.annotation.Autowired;
10  import org.springframework.beans.factory.annotation.Value;
11  import org.springframework.kafka.core.KafkaTemplate;
12  import org.springframework.stereotype.Service;
13
14  import java.nio.ByteBuffer;
15  import java.util.Calendar;
16  import java.util.Date;
17
18  @Service
19  public class KafkaRetryService {
20      private static final Logger log =
    LoggerFactory.getLogger(KafkaRetryService.class);
21
22      /**
```

```
    * 消息消费失败后下一次消费的延迟时间(秒)
    * 第一次重试延迟10秒;第  二次延迟30秒,第三次延迟1分钟...
    */
    private static final int[] RETRY_INTERVAL_SECONDS = {10, 30, 1*60, 2*60, 5*60, 10*60, 30*60, 1*60*60, 2*60*60};

    /**
    * 重试topic
    */
    @Value("${spring.kafka.topics.retry}")
    private String retryTopic;

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void consumerLater(ConsumerRecord<String, String> record){
        // 获取消息的已重试次数
        int retryTimes = getRetryTimes(record);
        Date nextConsumerTime = getNextConsumerTime(retryTimes);
        // 如果达到重试次数，则不再重试
        if(nextConsumerTime == null) {
            return;
        }

        // 组织消息
        RetryRecord retryRecord = new RetryRecord();
        retryRecord.setNextTime(nextConsumerTime.getTime());
        retryRecord.setTopic(record.topic());
        retryRecord.setRetryTimes(retryTimes);
        retryRecord.setKey(record.key());
        retryRecord.setValue(record.value());

        // 转换为字符串
        String value = JSON.toJSONString(retryRecord);
        // 发送到重试队列
        kafkaTemplate.send(retryTopic, null, value);
    }

    /**
    * 获取消息的已重试次数
    */
    private int getRetryTimes(ConsumerRecord record){
        int retryTimes = -1;
        for(Header header : record.headers()){
            if(RetryRecord.KEY_RETRY_TIMES.equals(header.key())){
                ByteBuffer buffer = ByteBuffer.wrap(header.value());
                retryTimes = buffer.getInt();
            }
        }
        retryTimes++;
        return retryTimes;
```

```
    }

    /**
    * 获取待重试消息的下一次消费时间
    */
    private Date getNextConsumerTime(int retryTimes){
        // 重试次数超过上限,不再重试
        if(RETRY_INTERVAL_SECONDS.length < retryTimes) {
            return null;
        }

        Calendar calendar = Calendar.getInstance();
        calendar.add(Calendar.SECOND, RETRY_INTERVAL_SECONDS[retryTimes]);
        return calendar.getTime();
    }
}

```

9. RetryListener.java

```
package com.lagou.kafka.demo.listener;

import com.alibaba.fastjson.JSON;
import com.lagou.kafka.demo.entity.RetryRecord;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.ZSetOperations;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.util.Set;
import java.util.UUID;

@Component
@EnableScheduling
public class RetryListener {

    private Logger log = LoggerFactory.getLogger(RetryListener.class);

    private static final String RETRY_KEY_ZSET = "_retry_key";
    private static final String RETRY_VALUE_MAP = "_retry_value";
    @Autowired

```

```
    private RedisTemplate<String,Object> redisTemplate;
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @Value("${spring.kafka.topics.test}")
    private String bizTopic;

    @KafkaListener(topics = "${spring.kafka.topics.retry}")
//    public void consume(List<ConsumerRecord<String, String>> list) {
//        for(ConsumerRecord<String, String> record : list){
    public void consume(ConsumerRecord<String, String> record) {

        System.out.println("需要重试的消息: " + record);
                    RetryRecord retryRecord = JSON.parseObject(record.value(), RetryRecord.class);

            /**
            * 防止待重试消息太多撑爆redis,可以将待重试消息按下一次重试时间分开存储放到不同介质
            * 例如下一次重试时间在半小时以后的消息储存到mysql,并定时从mysql读取即将重试的消息储储存到redis
            */

            // 通过redis的zset进行时间排序
            String key = UUID.randomUUID().toString();
            redisTemplate.opsForHash().put(RETRY_VALUE_MAP, key, record.value());
            redisTemplate.opsForZSet().add(RETRY_KEY_ZSET, key, retryRecord.getNextTime());
        }
//    }

    /**
    * 定时任务从redis读取到达重试时间的消息,发送到对应的topic
    */
//    @Scheduled(cron="2 * * * * *")
    @Scheduled(fixedDelay = 2000)
    public void retryFromRedis() {
        log.warn("retryFromRedis----begin");
        long currentTime = System.currentTimeMillis();
        // 根据时间倒序获取
        Set<ZSetOperations.TypedTuple<Object>> typedTuples =

 redisTemplate.opsForZSet().reverseRangeByScoreWithScores(RETRY_KEY_ZSET, 0, currentTime);
        // 移除取出的消息
        redisTemplate.opsForZSet().removeRangeByScore(RETRY_KEY_ZSET, 0, currentTime);
        for(ZSetOperations.TypedTuple<Object> tuple : typedTuples){
            String key = tuple.getValue().toString();
```

```
            String value = redisTemplate.opsForHash().get(RETRY_VALUE_MAP,
key).toString();
            redisTemplate.opsForHash().delete(RETRY_VALUE_MAP, key);
            RetryRecord retryRecord = JSON.parseObject(value,
RetryRecord.class);
            ProducerRecord record = retryRecord.parse();

            ProducerRecord recordReal = new ProducerRecord(
                    bizTopic,
                    record.partition(),
                    record.timestamp(),
                    record.key(),
                    record.value(),
                    record.headers()
            );

            kafkaTemplate.send(recordReal);
        }
        // todo 发生异常将发送失败的消息重新发送到redis
    }
}
```

10. RetryRecord.java

```
package com.lagou.kafka.demo.entity;

import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.header.Header;
import org.apache.kafka.common.header.internals.RecordHeader;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;

public class RetryRecord {

    public static final String KEY_RETRY_TIMES = "retryTimes";

    private String key;
    private String value;

    private Integer retryTimes;
    private String topic;
    private Long nextTime;

    public RetryRecord() {
    }

    public String getKey() {
```

```
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public Integer getRetryTimes() {
        return retryTimes;
    }

    public void setRetryTimes(Integer retryTimes) {
        this.retryTimes = retryTimes;
    }

    public String getTopic() {
        return topic;
    }

    public void setTopic(String topic) {
        this.topic = topic;
    }

    public Long getNextTime() {
        return nextTime;
    }

    public void setNextTime(Long nextTime) {
        this.nextTime = nextTime;
    }

    public ProducerRecord parse() {
        Integer partition = null;
        Long timestamp = System.currentTimeMillis();
        List<Header> headers = new ArrayList<>();
        ByteBuffer retryTimesBuffer = ByteBuffer.allocate(4);
        retryTimesBuffer.putInt(retryTimes);
        retryTimesBuffer.flip();
        headers.add(new RecordHeader(RetryRecord.KEY_RETRY_TIMES,
retryTimesBuffer));

        ProducerRecord sendRecord = new ProducerRecord(
                topic, partition, timestamp, key, value, headers);
```

```
76          return sendRecord;
77      }
78  }
```

第三节 Kafka集群与运维

3.1 集群应用场景

第1节 消息传递

Kafka可以很好地替代传统邮件代理。消息代理的使用有多种原因（将处理与数据生产者分离，缓冲未处理的消息等）。与大多数邮件系统相比，Kafka具有更好的吞吐量，内置的分区，复制和容错功能，这使其成为大规模邮件处理应用程序的理想解决方案。
根据我们的经验，消息传递的使用通常吞吐量较低，但是可能需要较低的端到端延迟，并且通常取决于Kafka提供的强大的持久性保证。

在这个领域，Kafka与ActiveMQ或 RabbitMQ等传统消息传递系统相当。

第2节 网站活动路由

Kafka最初的用例是能够将用户活动跟踪管道重建为一组实时的发布-订阅。这意味着将网站活动（页面浏览，搜索或用户可能采取的其他操作）发布到中心主题，每种活动类型只有一个主题。这些提要可用于一系列用例的订阅，包括实时处理，实时监控，以及加载到Hadoop或脱机数据仓库系统中以进行脱机处理和报告。

活动跟踪通常量很大，因为每个用户页面视图都会生成许多活动消息。

第3节 监控指标

Kafka通常用于操作监控数据。这涉及汇总来自分布式应用程序的统计信息，以生成操作数据的集中。

第4节 日志汇总

许多人使用Kafka代替日志聚合解决方案。日志聚合通常从服务器收集物理日志文件，并将它们放在中央位置（也许是文件服务器或HDFS）以进行处理。Kafka提取文件的详细信息，并以日志流的形式更清晰地抽象日志或事件数据。这允许较低延迟的处理，并更容易支持多个数据源和分布式数据消耗。与以日志为中心的系统（例如Scribe或Flume）相比，Kafka具有同样出色的性能，由于复制而提供的更强的耐用性保证以及更低的端到端延迟。

第5节 流处理

Kafka的许多用户在由多个阶段组成的处理管道中处理数据，其中原始输入数据从Kafka主题中使用，然后进行汇总，充实或以其他方式转换为新主题，以供进一步使用或后续处理。例如，用于推荐新闻文章的处理管道可能会从RSS提要中检索文章内容，并将其发布到“文章”主题中。进一步的处理可能会使该内容规范化或重复数据删除，并将清洗后的文章内容发布到新主题中；最后的处理阶段可能会尝试向用户推荐此内容。这样的处理管道基于各个主题创建实时数据流的图形。从0.10.0.0开始，一个轻量但功能强大的流处理库称为Kafka Streams 可以在Apache Kafka中使用来执行上述数据处理。除了Kafka Streams以外，其他开源流处理工具还包括Apache Storm和 Apache Samza。

第6节 活动采集

事件源是一种应用程序，其中状态更改以时间顺序记录记录。Kafka对大量存储的日志数据的支持使其成为以这种样式构建的应用程序的绝佳后端。

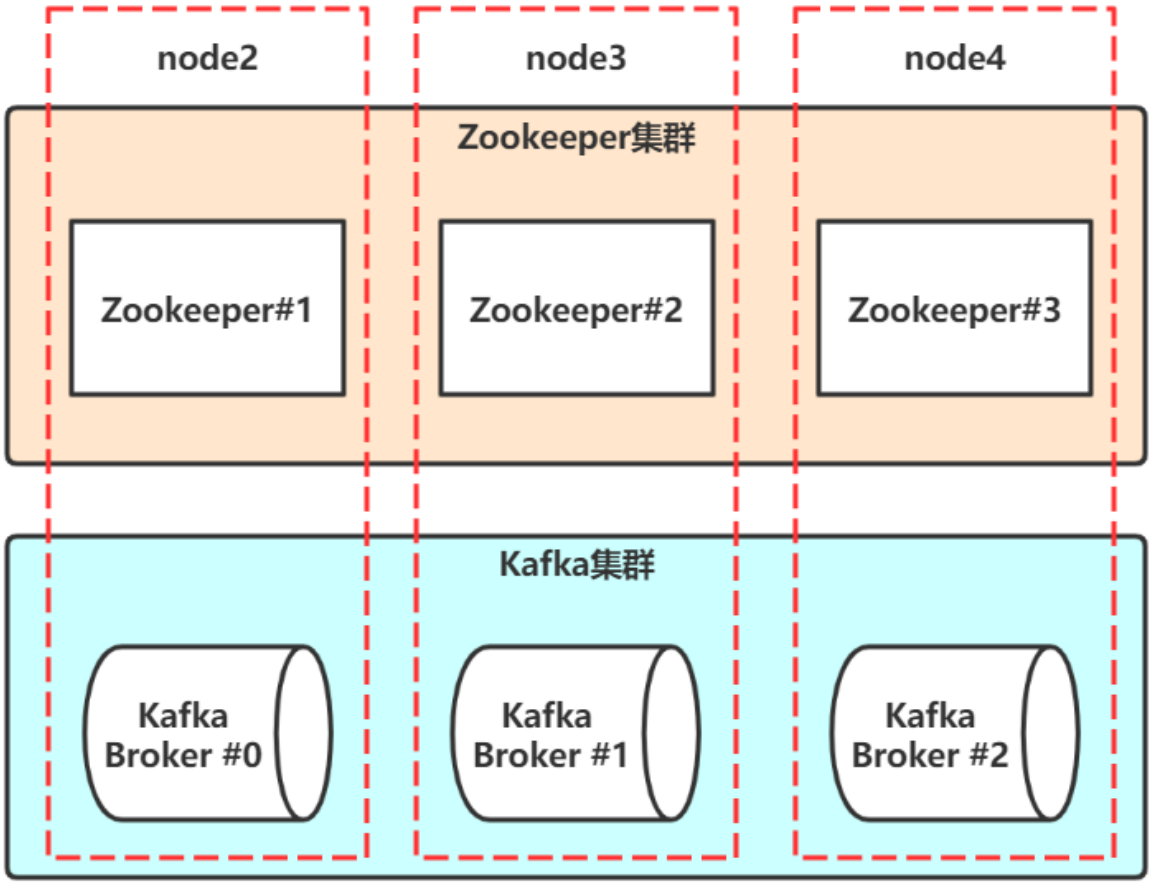
第7节 提交日志

Kafka可以用作分布式系统的一种外部提交日志。该日志有助于在节点之间复制数据，并充当故障节点恢复其数据的重新同步机制。Kafka中的日志压缩功能有助于支持此用法。在这种用法中，Kafka类似于Apache BookKeeper项目。

1. 横向扩展，提高Kafka的处理能力
2. 镜像，副本，提供高可用。

3.2 集群搭建

1. 搭建设计



2. 分配三台Linux，用于安装拥有三个节点的Kafka集群。

- node2(192.168.100.102)

- node3(192.168.100.103)
- node4(192.168.100.104)

以上三台主机的/etc/hosts配置：

```
192.168.100.101 node1
192.168.100.102 node2
192.168.100.103 node3
192.168.100.104 node4
```

3.2.1 Zookeeper集群搭建

1. Linux安装JDK，三台Linux都安装。
    - 上传JDK到linux
    - 安装并配置JDK

```
# 使用rpm安装JDK
rpm -ivh jdk-8u261-linux-x64.rpm
# 默认的安装路径是/usr/java/jdk1.8.0_261-amd64
# 配置JAVA_HOME
vim /etc/profile
# 文件最后添加两行
export JAVA_HOME=/usr/java/jdk1.8.0_261-amd64
export PATH=$PATH:$JAVA_HOME/bin
# 退出vim，使配置生效
source /etc/profile
```

    - 查看JDK是否正确安装

```
java -version
```

```
[root@node2 ~]# java -version
java version "1.8.0_261"
Java(TM) SE Runtime Environment (build 1.8.0_261-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.261-b12, mixed mode)
[root@node2 ~]#
```

2. Linux 安装Zookeeper，三台Linux都安装，以搭建Zookeeper集群
    - 上传zookeeper-3.4.14.tar.gz到Linux
    - 解压并配置zookeeper

```
# node2操作
# 解压到/opt目录
```

```
tar -zxf zookeeper-3.4.14.tar.gz -C /opt
# 配置
cd /opt/zookeeper-3.4.14/conf
cp zoo_sample.cfg zoo.cfg
vim zoo.cfg
# 设置
dataDir=/var/lagou/zookeeper/data
# 添加
server.1=node2:2881:3881
server.2=node3:2881:3881
server.3=node4:2881:3881

# 退出vim
mkdir -p /var/lagou/zookeeper/data
echo 1 > /var/lagou/zookeeper/data/myid

# 配置环境变量
vim /etc/profile
# 添加
export ZOOKEEPER_PREFIX=/opt/zookeeper-3.4.14
export PATH=$PATH:$ZOOKEEPER_PREFIX/bin
export ZOO_LOG_DIR=/var/lagou/zookeeper/log

# 退出vim，让配置生效
source /etc/profile

# 将/opt/zookeeper-3.4.14拷贝到node3，node4
scp -r /opt/zookeeper-3.4.14/ node3:/opt
scp -r /opt/zookeeper-3.4.14/ node4:/opt
```

- node3配置

```
# 配置环境变量
vim /etc/profile
# 在配置JDK环境变量基础上，添加内容
export ZOOKEEPER_PREFIX=/opt/zookeeper-3.4.14
export PATH=$PATH:$ZOOKEEPER_PREFIX/bin
export ZOO_LOG_DIR=/var/lagou/zookeeper/log

# 退出vim，让配置生效
source /etc/profile

mkdir -p /var/lagou/zookeeper/data
echo 2 > /var/lagou/zookeeper/data/myid
```

- node4配置

```
# 配置环境变量
vim /etc/profile
# 在配置JDK环境变量基础上，添加内容
export ZOOKEEPER_PREFIX=/opt/zookeeper-3.4.14
export PATH=$PATH:$ZOOKEEPER_PREFIX/bin
export ZOO_LOG_DIR=/var/lagou/zookeeper/log

# 退出vim，让配置生效
source /etc/profile

mkdir -p /var/lagou/zookeeper/data
echo 3 > /var/lagou/zookeeper/data/myid
```

- 启动zookeeper

```
# 在三台Linux上启动Zookeeper
[root@node2 ~]# zkServer.sh start
[root@node3 ~]# zkServer.sh start
[root@node4 ~]# zkServer.sh start

# 在三台Linux上查看Zookeeper的状态
[root@node2 ~]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.14/bin/../conf/zoo.cfg
Mode: follower

[root@node3 ~]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.14/bin/../conf/zoo.cfg
Mode: leader

[root@node4 ~]# zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.14/bin/../conf/zoo.cfg
Mode: follower
```

3.2.2 Kafka集群搭建

1. 安装Kafka
    - 上传并解压Kafka到/opt

```
# 解压到/opt
tar -zxf kafka_2.12-1.0.2.tgz -C /opt

# 拷贝到node3和node4
scp -r /opt/kafka_2.12-1.0.2/ node3:/opt
scp -r /opt/kafka_2.12-1.0.2/ node4:/opt
```

- 配置Kafka

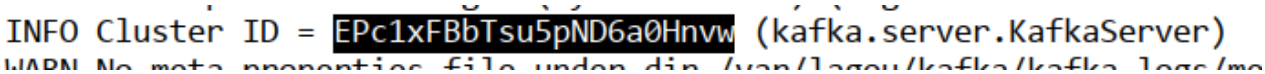
```
# 配置环境变量，三台Linux都要配置
vim /etc/profile
# 添加以下内容：
export KAFKA_HOME=/opt/kafka_2.12-1.0.2
export PATH=$PATH:$KAFKA_HOME/bin

# 让配置生效
source /etc/profile

# node2配置
vim /opt/kafka_2.12-1.0.2/config/server.properties

broker.id=0
listeners=PLAINTEXT://:9092
advertised.listeners=PLAINTEXT://node2:9092
log.dirs=/var/lagou/kafka/kafka-logs
zookeeper.connect=node2:2181,node3:2181,node4:2181/myKafka
# 其他使用默认配置


# node3配置
vim /opt/kafka_2.12-1.0.2/config/server.properties

broker.id=1
listeners=PLAINTEXT://:9092
advertised.listeners=PLAINTEXT://node3:9092
log.dirs=/var/lagou/kafka/kafka-logs
zookeeper.connect=node2:2181,node3:2181,node4:2181/myKafka
# 其他使用默认配置

# node4配置
vim /opt/kafka_2.12-1.0.2/config/server.properties

broker.id=2
listeners=PLAINTEXT://:9092
advertised.listeners=PLAINTEXT://node4:9092
log.dirs=/var/lagou/kafka/kafka-logs
zookeeper.connect=node2:2181,node3:2181,node4:2181/myKafka
# 其他使用默认配置
```

- 启动Kafka

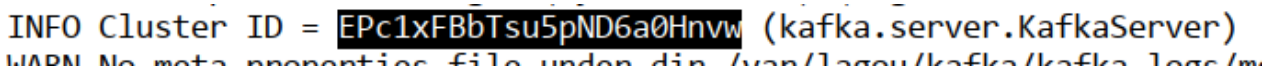
```
[root@node2 ~]# kafka-server-start.sh /opt/kafka_2.12-1.0.2/config/server.properties
[root@node3 ~]# kafka-server-start.sh /opt/kafka_2.12-1.0.2/config/server.properties
[root@node4 ~]# kafka-server-start.sh /opt/kafka_2.12-1.0.2/config/server.properties
```

- 验证Kafka

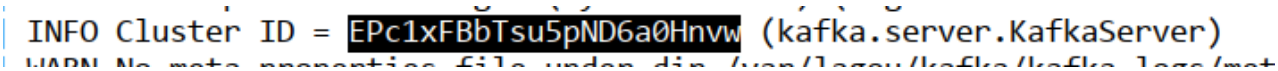
  node2节点的Cluster Id：



  node3节点的Cluster Id：



  node4节点的Cluster Id：



1. Cluster Id是一个唯一的不可变的标志符，用于唯一标志一个Kafka集群。
2. 该Id最多可以有22个字符组成，字符对应于URL-safe Base64。
3. Kafka 0.10.1版本及之后的版本中，在集群第一次启动的时候，Broker从Zookeeper的<Kafka_ROOT>/cluster/id节点获取。如果该Id不存在，就自动生成一个新的。

```
zkCli.sh
# 查看每个Broker的信息
get /brokers/ids/0
get /brokers/ids/1
get /brokers/ids/2
```

node2节点在Zookeeper上的信息：

```
[zk: localhost:2181(CONNECTED) 5] get /myKafka/brokers/ids/0
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://node2:9092"],"jmx_port":-1,"host":"node2","timestamp":"1595997319960","port":9092,"version":4}
cZxid = 0x100000020
```

node3节点在Zookeeper上的信息：

```
[zk: localhost:2181(CONNECTED) 6] get /myKafka/brokers/ids/1
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://node3:9092"],"jmx_port":-1,"host":"node3","timestamp":"1595997333687","port":9092,"version":4}
cZxid = 0x100000027
```

node4节点在Zookeeper上的信息：

```
[zk: localhost:2181(CONNECTED) 7] get /myKafka/brokers/ids/2
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://node4:9092"],"jmx_port":-1,"host":"node4","timestamp":"1595997347123","port":9092,"version":4}
```

3.3 集群监控

3.3.1 监控度量指标

Kafka使用Yammer Metrics在服务器和Scala客户端中报告指标。Java客户端使用Kafka Metrics，它是一个内置的度量标准注册表，可最大程度地减少拉入客户端应用程序的传递依赖项。两者都通过JMX公开指标，并且可以配置为使用可插拔的统计报告器报告统计信息，以连接到您的监视系统。

具体的监控指标可以查看官方文档。

3.3.1.1 JMX

3.3.1.1.1 Kafka开启Jmx端口

```
[root@node4 bin]# vim /opt/kafka_2.12-1.0.2/bin/kafka-server-start.sh
```

```
#!/bin/bash
# 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.

export JMX_PORT=9581

if [ $# -lt 1 ];
then
```

所有kafka机器添加一个`JMX_PORT`，并重启kafka

3.3.1.1.2 验证JMX开启

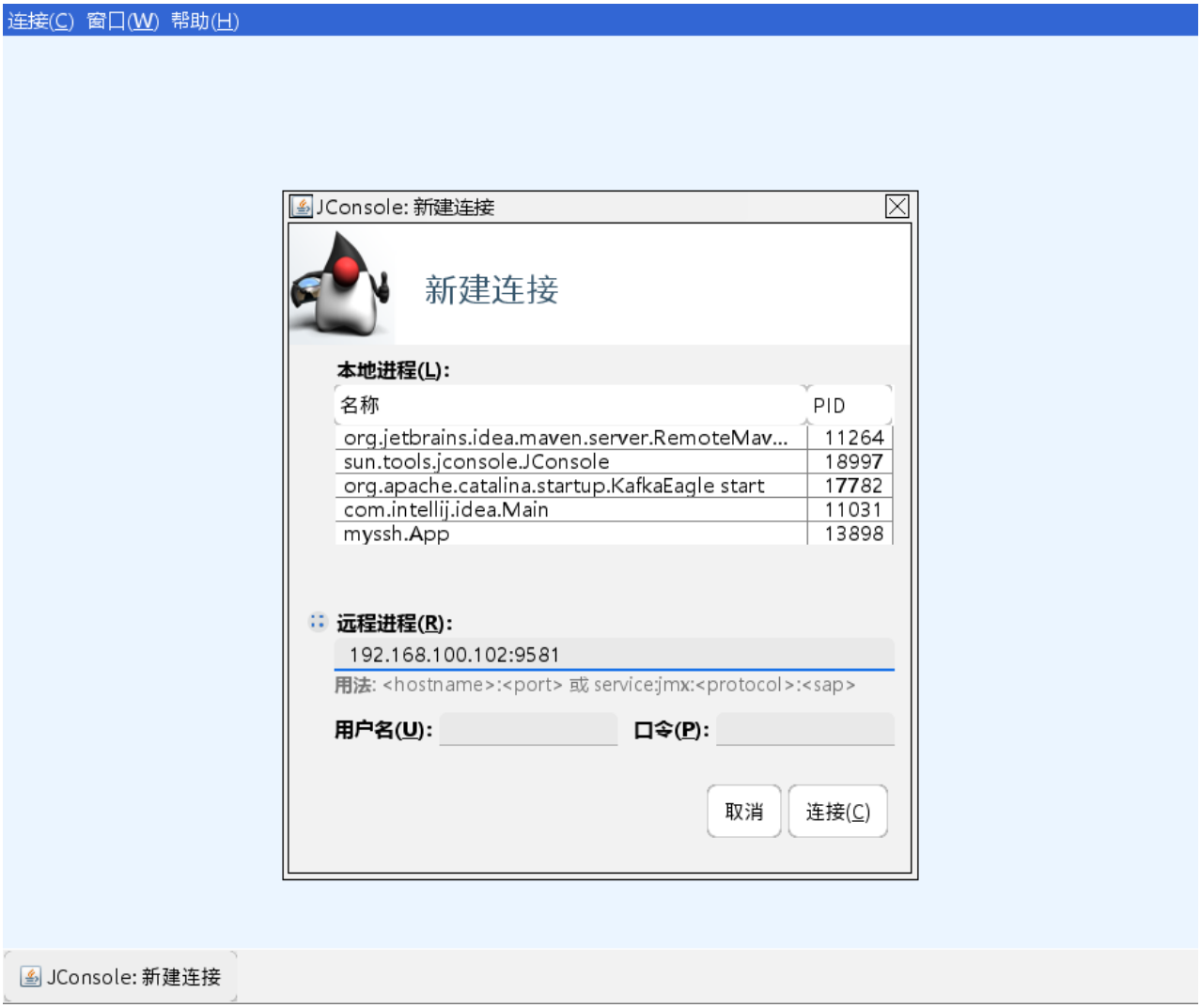
首先打印9581端口占用的进程信息，然后使用进程编号对应到Kafka的进程号，搞定。

```
[root@node4 bin]# ss -nelp | grep 9581
tcp    LISTEN     0      50     [::]:9581               [::]:*                   users:(("java",pid=18369,fd=79)) ino:27176 sk:ffff8865391139c0 v6only:0 <->
[root@node4 bin]# jps
18369 Kafka
1554 QuorumPeerMain
18989 Jps
[root@node4 bin]#
```

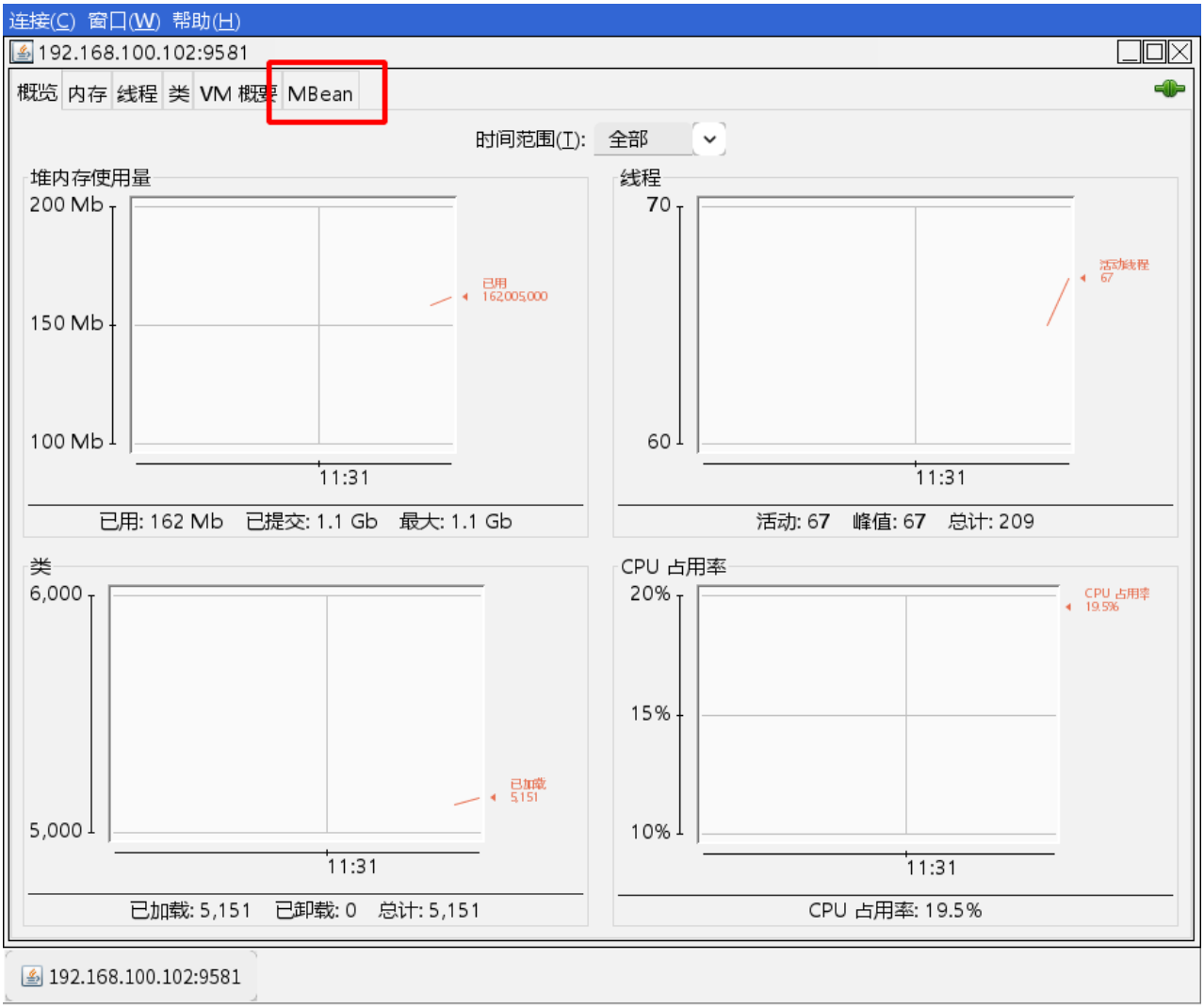
> 也可以查看Kafka启动日志，确定启动参数 `-Dcom.sun.management.jmxremote.port=9581` 存在即可

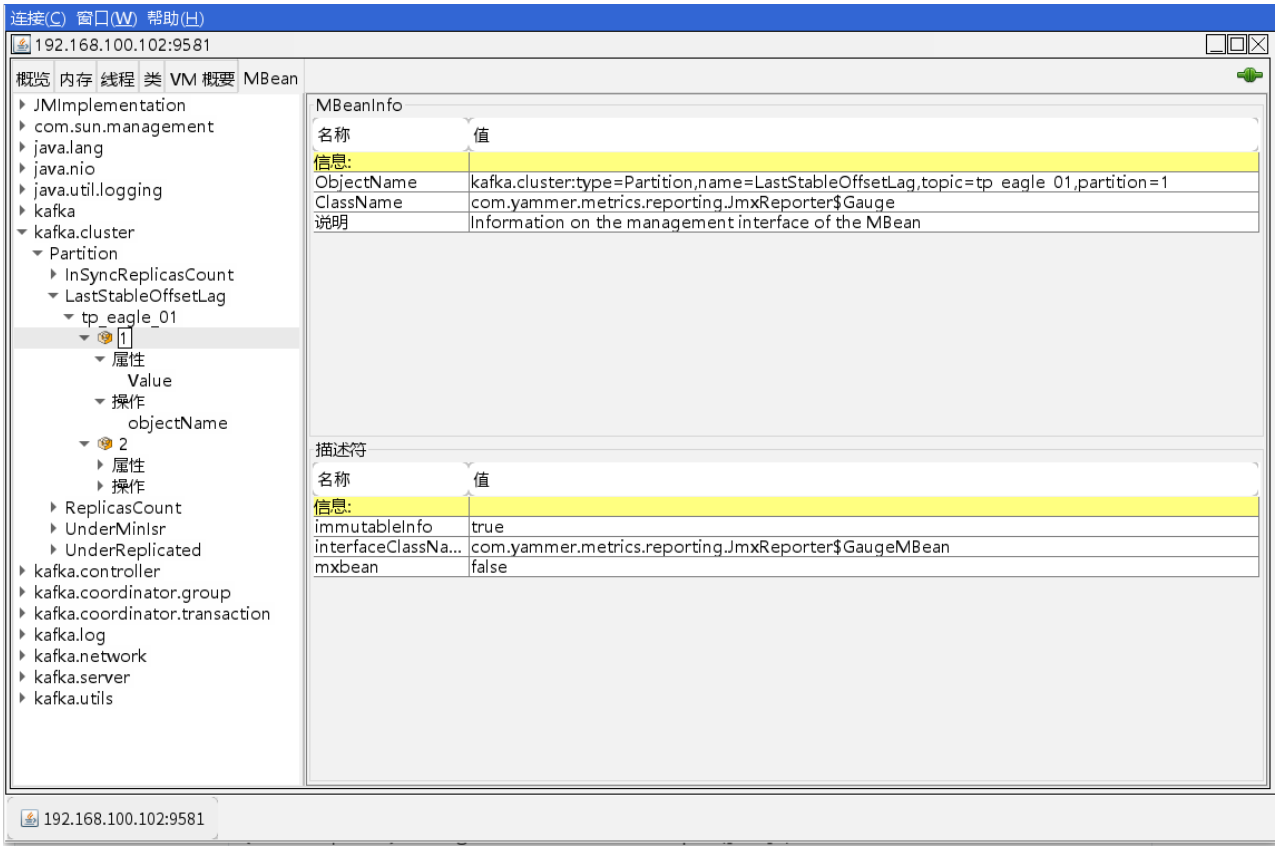
3.3.1.2 使用JConsole链接JMX端口

1. win/mac，找到jconsole工具并打开， 在 `${JAVA_HOEM}/bin/`
   Mac电脑可以直接命令行输入 `jconsole`




连接(C) 窗口(W) 帮助(H)
192.168.100.102:9581 (已断开连接)
概览 内存 线程 类 VM 概要 MBean
安全连接失败。是否以不安全的方式重试?
无法使用 SSL 连接到192.168.100.102:9581。
是否在不使用 SSL 的情况下进行尝试?
(用户名和口令将以纯文本格式发送。)
取消
不安全的连接
192.168.100.102:9581 (已断开连接)


连接(C) 窗口(W) 帮助(H)
192.168.100.102:9581
概览 内存 线程 类 VM 概要 MBean
时间范围(T): 全部
堆内存使用量
200 Mb
150 Mb
100 Mb
11:31
已用 162,005,000
已用: 162 Mb 已提交: 1.1 Gb 最大: 1.1 Gb
线程
70
60
11:31
活动线程 67
活动: 67 峰值: 67 总计: 209
类
6,000
5,000
11:31
已加载 5,151
已加载: 5,151 已卸载: 0 总计: 5,151
CPU 占用率
20%
15%
10%
11:31
CPU 占用率 19.5%
CPU 占用率: 19.5%
192.168.100.102:9581


连接(C) 窗口(W) 帮助(H)
192.168.100.102:9581
概览 内存 线程 类 VM 概要 MBean
JMImplementation
com.sun.management
java.lang
java.nio
java.util.logging
kafka
kafka.cluster
Partition
InSyncReplicasCount
LastStableOffsetLag
tp_eagle_01
1
属性
Value
操作
objectName
2
属性
操作
ReplicasCount
UnderMinIsr
UnderReplicated
kafka.controller
kafka.coordinator.group
kafka.coordinator.transaction
kafka.log
kafka.network
kafka.server
kafka.utils
MBeanInfo
名称 值
信息:
ObjectName kafka.cluster:type=Partition,name=LastStableOffsetLag,topic=tp_eagle_01,partition=1
ClassName com.yammer.metrics.reporting.JmxReporter$Gauge
说明 Information on the management interface of the MBean
描述符
名称 值
信息:
immutableInfo true
interfaceClassNa... com.yammer.metrics.reporting.JmxReporter$GaugeMBean
mxbean false
192.168.100.102:9581

详细的监控指标

相见官方文档：http://kafka.apache.org/10/documentation.html#monitoring

这里列出常用的：

OS监控项

objectName	指标项	说明
java.lang:type=OperatingSystem	FreePhysicalMemorySize	空闲物理内存
java.lang:type=OperatingSystem	SystemCpuLoad	系统CPU利用率
java.lang:type=OperatingSystem	ProcessCpuLoad	进程CPU利用率
java.lang:type=GarbageCollector, name=G1 Young Generation	CollectionCount	GC次数

broker指标

objectName	指标项	说明
kafka.server:type=BrokerTopicMetrics, name=BytesInPerSec	Count	每秒输入的流量
kafka.server:type=BrokerTopicMetrics, name=BytesOutPerSec	Count	每秒输出的流量
kafka.server:type=BrokerTopicMetrics, name=BytesRejectedPerSec	Count	每秒扔掉的流量
kafka.server:type=BrokerTopicMetrics, name=MessagesInPerSec	Count	每秒的消息写入总量
kafka.server:type=BrokerTopicMetrics, name=FailedFetchRequestsPerSec	Count	当前机器每秒fetch请求失败的数量
kafka.server:type=BrokerTopicMetrics, name=FailedProduceRequestsPerSec	Count	当前机器每秒produce请求失败的数量
kafka.server:type=ReplicaManager, name=PartitionCount	Value	该broker上的partition的数量
kafka.server:type=ReplicaManager, name=LeaderCount	Value	Leader的replica的数量
kafka.network:type=RequestMetrics, name=TotalTimeMs,request=FetchConsumer	Count	一个请求FetchConsumer耗费的所有时间
kafka.network:type=RequestMetrics, name=TotalTimeMs,request=FetchFollower	Count	一个请求FetchFollower耗费的所有时间
kafka.network:type=RequestMetrics, name=TotalTimeMs,request=Produce	Count	一个请求Produce耗费的所有时间

producer以及topic指标

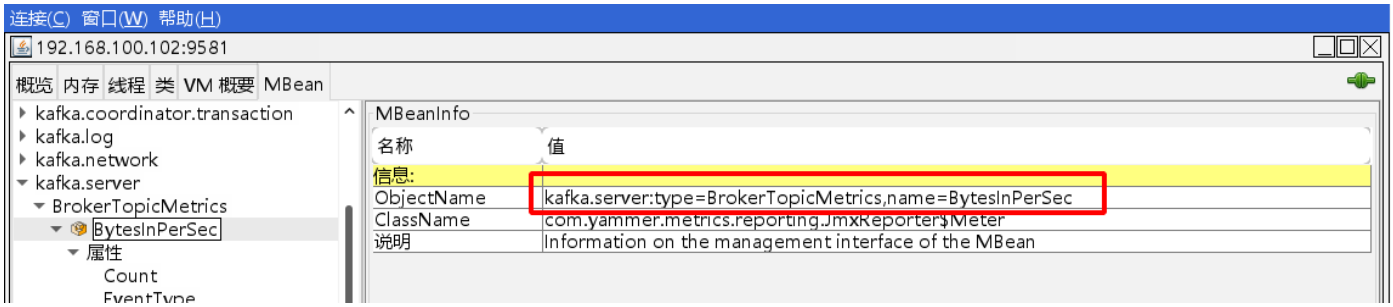
objectName	指标项	官网说明	译文说明
kafka.producer:type=producer-metrics,client-id=console-producer(client-id会变化)	incoming-byte-rate	The average number of incoming bytes received per second from all servers.	producer每秒的平均写入流量
kafka.producer:type=producer-metrics,client-id=console-producer(client-id会变化)	outgoing-byte-rate	The average number of outgoing bytes sent per second to all servers.	producer每秒的输出流量
kafka.producer:type=producer-metrics,client-id=console-producer(client-id会变化)	request-rate	The average number of requests sent per second to the broker.	producer每秒发给broker的平均request次数
kafka.producer:type=producer-metrics,client-id=console-producer(client-id会变化)	response-rate	The average number of responses received per second from the broker.	producer每秒发给broker的平均response次数
kafka.producer:type=producer-metrics,client-id=console-producer(client-id会变化)	request-latency-avg	The average time taken for a fetch request.	一个fetch请求的平均时间
kafka.producer:type=producer-topic-metrics,client-id=console-producer,topic=testjmx(client-id和topic名称会变化)	record-send-rate	The average number of records sent per second for a topic.	每秒从topic发送的平均记录数
kafka.producer:type=producer-topic-metrics,client-id=console-producer,topic=testjmx(client-id和topic名称会变化)	record-retry-total	The total number of retried record sends	重试发送的消息总数量
kafka.producer:type=producer-topic-metrics,client-id=console-producer,topic=testjmx(client-id和topic名称会变化)	record-error-total	The total number of record sends that resulted in errors	发送错误的消息总数量

consumer指标

objectName	指标项	官网说明	说明
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=consumer-1(client-id会变化)	records-lag-max	Number of messages the consumer lags behind the producer by. Published by the consumer, not broker.	由consumer提交的消息消费lag
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=consumer-1(client-id会变化)	records-consumed-rate	The average number of records consumed per second	每秒平均消费的消息数量

3.3.1.3 编程手段来获取监控指标

查看要监控哪个指标：



代码实现：

```
package com.lagou.kafka.demo.monitor;

import javax.management.*;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import java.io.IOException;
import java.util.Iterator;
import java.util.Set;

public class JMXMonitorDemo {

    public static void main(String[] args) throws IOException,
MalformedObjectNameException, AttributeNotFoundException, MBeanException,
ReflectionException, InstanceNotFoundException {

        String jmxServiceURL =
"service:jmx:rmi:///jndi/rmi://192.168.100.103:9581/jmxrmi";

        JMXServiceURL jmxURL = null;
        JMXConnector jmxc = null;
        MBeanServerConnection jmxs = null;
        ObjectName mbeanObjName = null;
        Iterator sampleIter = null;
        Set sampleSet = null;

        // 创建JMXServiceURL对象，参数是
        jmxURL = new JMXServiceURL(jmxServiceURL);
        // 建立到指定URL服务器的连接
        jmxc = JMXConnectorFactory.connect(jmxURL);
        // 返回代表远程MBean服务器的MBeanServerConnection对象
        jmxs = jmxc.getMBeanServerConnection();
        // 根据传入的字符串，创建ObjectName对象
//        mbeanObjName = new
ObjectName("kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec");
        mbeanObjName = new
ObjectName("kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=tp_eagle_01
");
        // 获取指定ObjectName对应的MBeans
```

```
        sampleSet = jmxs.queryMBeans(null, mbeanObjName);
        // 迭代器
        sampleIter = sampleSet.iterator();

        if (sampleSet.isEmpty()) {

        } else {
            // 如果返回了，则打印信息
            while (sampleIter.hasNext()) {
                // Used to represent the object name of an MBean and its class name.
                // If the MBean is a Dynamic MBean the class name should be retrieved
from the MBeanInfo it provides.
                // 用于表示MBean的ObjectName和ClassName
                ObjectInstance sampleObj = (ObjectInstance) sampleIter.next();
                ObjectName objectName = sampleObj.getObjectName();
                // 查看指定MBean指定属性的值
                String count = jmxs.getAttribute(objectName, "Count").toString();
                System.out.println(count);
            }
        }

        // 关闭
        jmxc.close();
    }

}
```

3.3.2 监控工具 Kafka Eagle

我们可以使用Kafka-eagle管理Kafka集群

核心模块：

- 面板可视化
- 主题管理，包含创建主题、删除主题、主题列举、主题配置、主题查询等
- 消费者应用：对不同消费者应用进行监控，包含Kafka API、Flink API、Spark API、Storm API、Flume API、LogStash API等
- 集群管理：包含对Kafka集群和Zookeeper集群的详情展示，其内容包含Kafka启动时间、Kafka端口号、Zookeeper Leader角色等。同时，还有多集群切换管理，Zookeeper Client操作入口
- 集群监控：包含对Broker、Kafka核心指标、Zookeeper核心指标进行监控，并绘制历史趋势图
- 告警功能：对消费者应用数据积压情况进行告警，以及对Kafka和Zookeeper监控度进行告警。同时，支持邮件、微信、钉钉告警通知
- 系统管理：包含用户创建、用户角色分配、资源访问进行管理

架构：

- 可视化：负责展示主题列表、集群健康、消费者应用等
- 采集器：数据采集的来源包含Zookeeper、Kafka JMX & 内部Topic、Kafka API（Kafka 2.x以后版本）
- 数据存储：目前Kafka Eagle存储采用MySQL或SQLite，数据库和表的创建均是自动完成的，按照官方文档进行配置好，启动Kafka Eagle就会自动创建，用来存储元数据和监控数据
- 监控：负责见消费者应用消费情况、集群健康状态
- 告警：对监控到的异常进行告警通知，支持邮件、微信、钉钉等方式
- 权限管理：对访问用户进行权限管理，对于管理员、开发者、访问者等不同角色的用户，分配不用的访问权限

需要Kafka节点开启**JMX**。前面讲过了。

```
# 下载编译好的包
wget http://pkgs-linux.cvimer.com/kafka-eagle.zip

# 配置kafka-eagle
unzip kafka-eagle.zip
cd kafka-eagle/kafka-eagle-web/target
mkdir -p test
cp kafka-eagle-web-2.0.1-bin.tar.gz test/
tar xf kafka-eagle-web-2.0.1-bin.tar.gz
cd kafka-eagle-web-2.0.1
```

需要配置环境变量：

KE_HOME=

PATH=

conf下的配置文件：system-config.properties

```
######################################
# multi zookeeper & kafka cluster list
######################################
# 集群的别名，用于在kafka-eagle中进行区分。
# 可以配置监控多个集群，别名用逗号隔开
# kafka.eagle.zk.cluster.alias=cluster1,cluster2,cluster3
kafka.eagle.zk.cluster.alias=cluster1
# cluster1.zk.list=10.1.201.17:2181,10.1.201.22:2181,10.1.201.23:2181
# 配置当前集群的zookeeper地址，此处的值要与Kafka的server.properties中的zookeeper.connect的值一致
# 此处的前缀就是集群的别名
cluster1.zk.list=node2:2181,node3:2181,node4:2181/myKafka
```

```
#cluster2.zk.list=xdn10:2181,xdn11:2181,xdn12:2181

######################################
# zookeeper enable acl
######################################
cluster1.zk.acl.enable=false
cluster1.zk.acl.schema=digest
cluster1.zk.acl.username=test
cluster1.zk.acl.password=test123

######################################
# broker size online list
######################################
cluster1.kafka.eagle.broker.size=20

######################################
# zookeeper客户端连接数限制
######################################
kafka.zk.limit.size=25

######################################
# kafka eagle网页端口号
######################################
kafka.eagle.webui.port=8048

######################################
# kafka 消费信息存储位置，用来兼容kafka低版本
######################################
cluster1.kafka.eagle.offset.storage=kafka
cluster2.kafka.eagle.offset.storage=zk

######################################
# kafka metrics, 15 days by default
######################################
kafka.eagle.metrics.charts=true
kafka.eagle.metrics.retain=15


######################################
# kafka sql topic records max
######################################
kafka.eagle.sql.topic.records.max=5000
kafka.eagle.sql.fix.error=true

######################################
# 管理员删除kafka中topic的口令
######################################
kafka.eagle.topic.token=keadmin

######################################
#  kafka 集群是否开启了认证模式，此处是cluster1集群的配置，禁用
```

```
######################################
cluster1.kafka.eagle.sasl.enable=false
cluster1.kafka.eagle.sasl.protocol=SASL_PLAINTEXT
cluster1.kafka.eagle.sasl.mechanism=SCRAM-SHA-256
cluster1.kafka.eagle.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required username="kafka" password="kafka-eagle";
cluster1.kafka.eagle.sasl.client.id=
cluster1.kafka.eagle.sasl.cgroup.enable=false
cluster1.kafka.eagle.sasl.cgroup.topics=

######################################
# kafka ssl authenticate, 示例配置
######################################
cluster2.kafka.eagle.sasl.enable=false
cluster2.kafka.eagle.sasl.protocol=SASL_PLAINTEXT
cluster2.kafka.eagle.sasl.mechanism=PLAIN
cluster2.kafka.eagle.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="kafka" password="kafka-eagle";
cluster2.kafka.eagle.sasl.client.id=
cluster2.kafka.eagle.sasl.cgroup.enable=false
cluster2.kafka.eagle.sasl.cgroup.topics=

######################################
# kafka ssl authenticate, 示例配置
######################################
cluster3.kafka.eagle.ssl.enable=false
cluster3.kafka.eagle.ssl.protocol=SSL
cluster3.kafka.eagle.ssl.truststore.location=
cluster3.kafka.eagle.ssl.truststore.password=
cluster3.kafka.eagle.ssl.keystore.location=
cluster3.kafka.eagle.ssl.keystore.password=
cluster3.kafka.eagle.ssl.key.password=
cluster3.kafka.eagle.ssl.cgroup.enable=false
cluster3.kafka.eagle.ssl.cgroup.topics=

######################################
# 存储监控数据的数据库地址
# kafka默认使用sqlite存储，需要指定和创建sqlite的目录
# 如 /home/lagou/hadoop/kafka-eagle/db
######################################
kafka.eagle.driver=org.sqlite.JDBC
kafka.eagle.url=jdbc:sqlite:/home/lagou/hadoop/kafka-eagle/db/ke.db
kafka.eagle.username=root
kafka.eagle.password=www.kafka-eagle.org

######################################
# 还可以使用MySLQ存储监控数据
######################################
#kafka.eagle.driver=com.mysql.jdbc.Driver
#kafka.eagle.url=jdbc:mysql://127.0.0.1:3306/ke?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
```

```
#kafka.eagle.username=root
#kafka.eagle.password=123456

######################################
# kafka eagle 设置告警邮件服务器
######################################
kafka.eagle.mail.enable=true
kafka.eagle.mail.sa=kafka_lagou_alert
kafka.eagle.mail.username=kafka_lagou_alert@163.com
kafka.eagle.mail.password=Pas2W0rd
kafka.eagle.mail.server.host=smtp.163.com
kafka.eagle.mail.server.port=25
```

> 也可以自行编译，`https://github.com/smartloli/kafka-eagle`
> 创建Eagel的存储目录：`mkdir -p /hadoop/kafka-eagle`

```
# 启动kafka-eagle
./bin/ke.sh start
```

> 会提示我们登陆地址和账号密码

第四节 Kafka源码剖析

4.1 Kafka源码剖析之源码阅读环境搭建

首先下载源码：http://archive.apache.org/dist/kafka/1.0.2/kafka-1.0.2-src.tgz

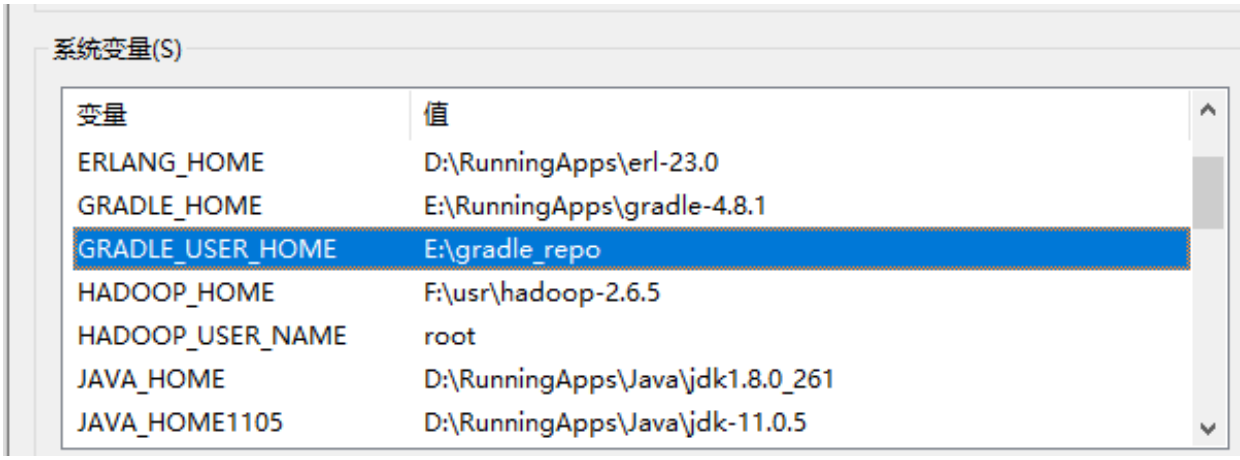
gradle-4.8.1下载地址：https://services.gradle.org/distributions/gradle-4.8.1-bin.zip

Scala-2.12.12下载地址：https://downloads.lightbend.com/scala/2.12.12/scala-2.12.12.msi

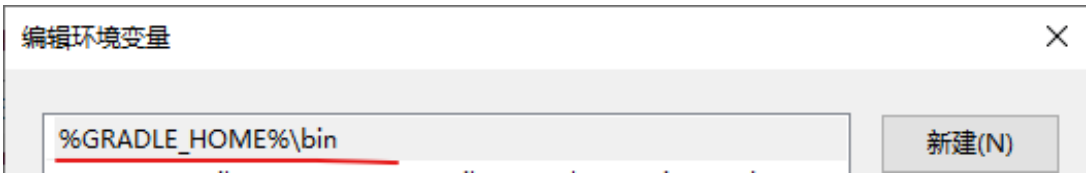
4.1.1 安装配置Gradle

解压gradle4.8.-bin.zip到一个目录

配置环境变量，其中GRADLE_HOME指向gradle解压到的根目录，GRADLE_USER_HOME指向gradle的本地仓库位置。



PATH环境变量：



进入GRADLE_USER_HOME目录，添加init.gradle，配置gradle的源：

init.gradle内容：

```
allprojects {
    repositories {
        maven { url 'https://maven.aliyun.com/repository/public/' }
        maven { url 'https://maven.aliyun.com/nexus/content/repositories/google' }
        maven { url 'https://maven.aliyun.com/nexus/content/groups/public/' }
        maven { url 'https://maven.aliyun.com/nexus/content/repositories/jcenter'}

        all { ArtifactRepository repo ->
            if (repo instanceof MavenArtifactRepository) {
                def url = repo.url.toString()

                if (url.startsWith('https://repo.maven.apache.org/maven2/') ||
url.startsWith('https://repo.maven.org/maven2') ||
url.startsWith('https://repo1.maven.org/maven2') ||
url.startsWith('https://jcenter.bintray.com/')) {
                    //project.logger.lifecycle "Repository ${repo.url} replaced by
$REPOSITORY_URL."
                    remove repo
                }
            }
        }
    }

    buildscript {

        repositories {

```

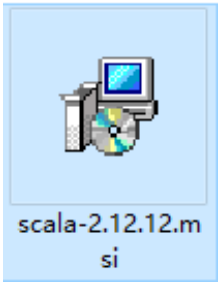
```
            maven { url 'https://maven.aliyun.com/repository/public/'}
            maven { url 'https://maven.aliyun.com/nexus/content/repositories/google'
}
            maven { url 'https://maven.aliyun.com/nexus/content/groups/public/' }
            maven { url
'https://maven.aliyun.com/nexus/content/repositories/jcenter'}
            all { ArtifactRepository repo ->
                if (repo instanceof MavenArtifactRepository) {
                    def url = repo.url.toString()
                    if (url.startsWith('https://repo1.maven.org/maven2') ||
url.startsWith('https://jcenter.bintray.com/')) {
                        //project.logger.lifecycle "Repository ${repo.url} replaced
by $REPOSITORY_URL."
                        remove repo
                    }
                }
            }
        }
    }
}
```

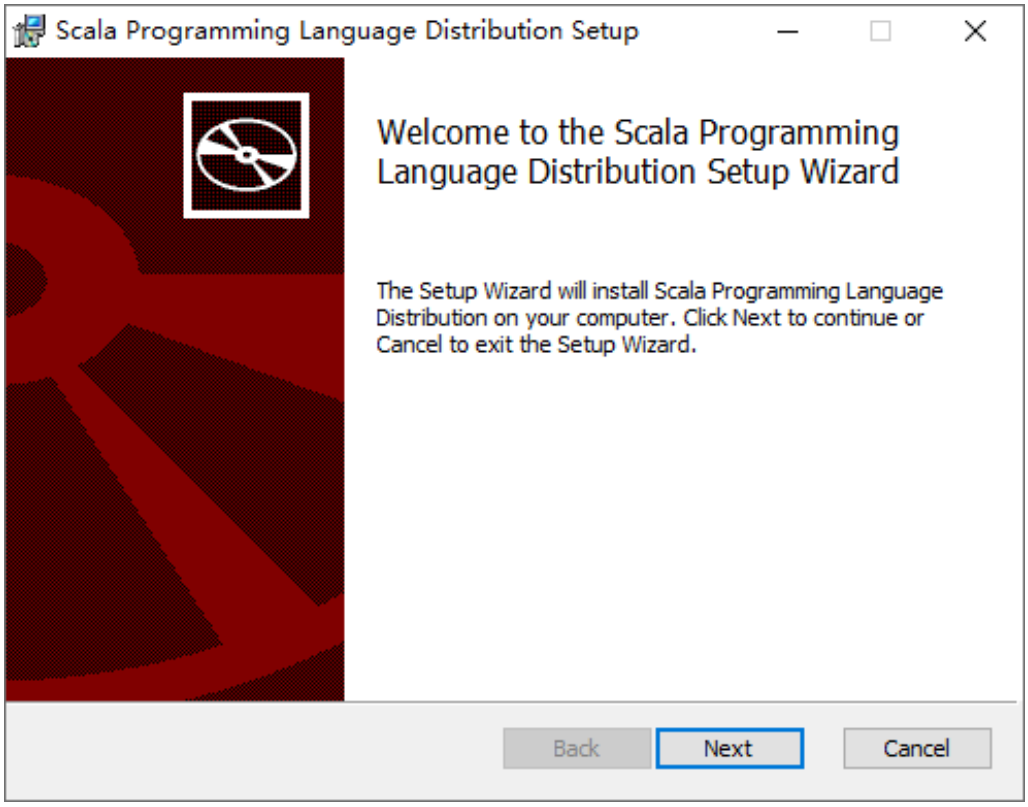
保存并退出，打开cmd，运行：

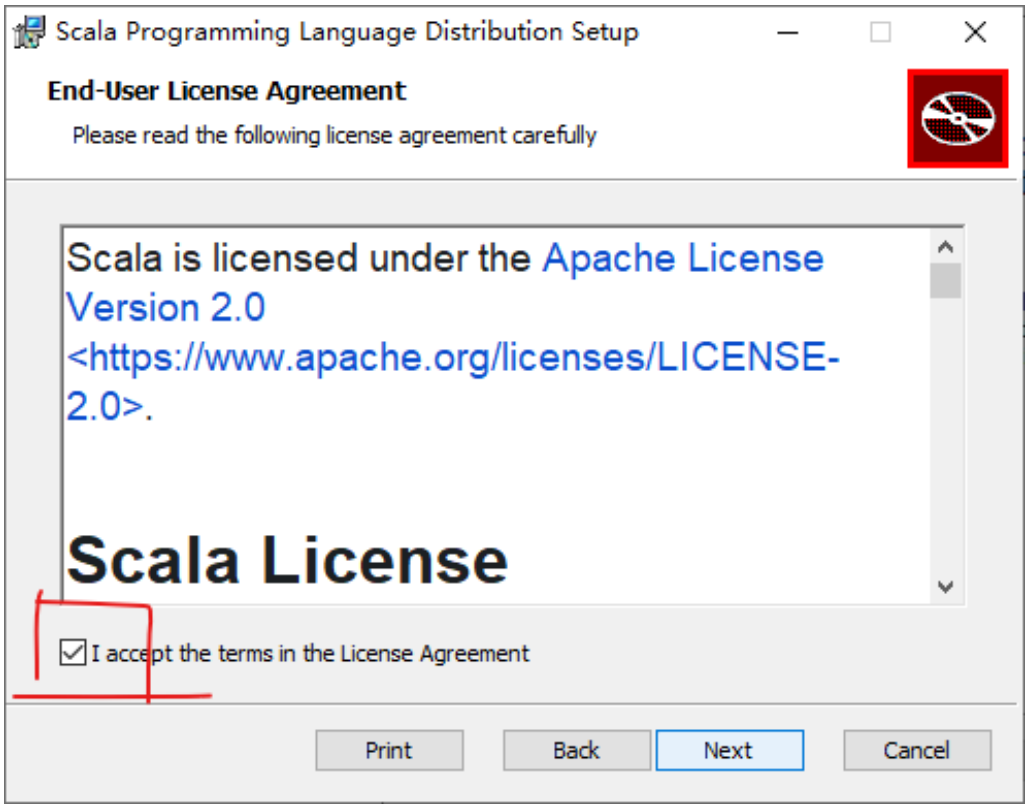
设置成功。

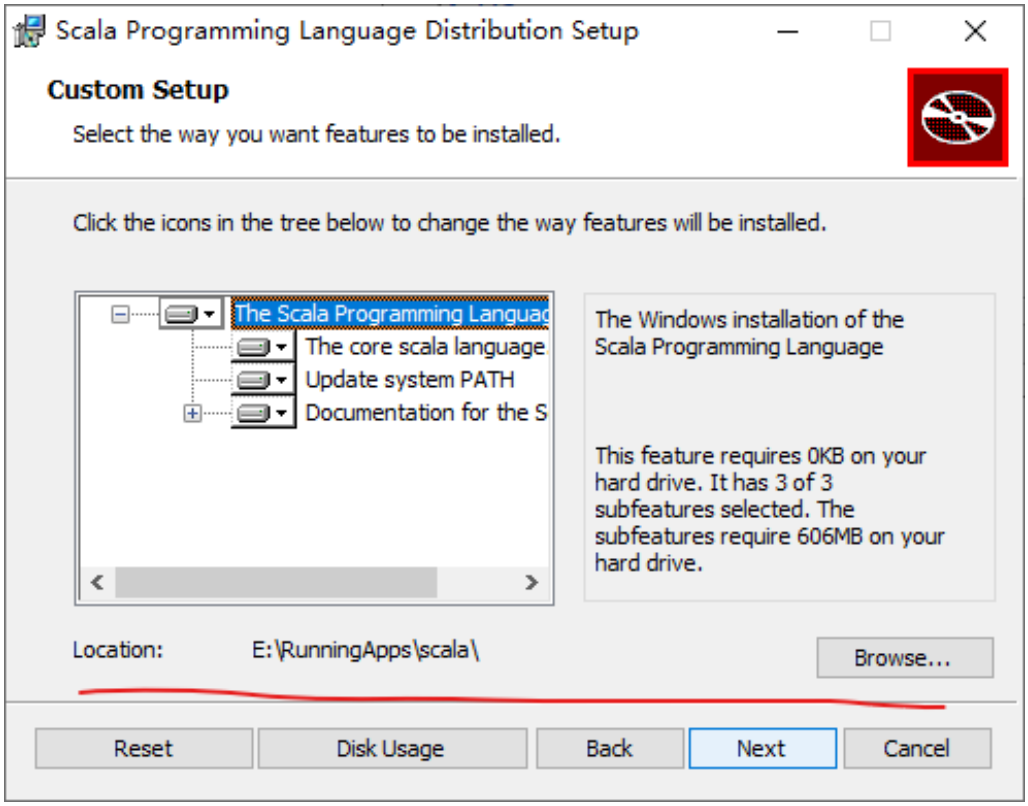
4.1.2 Scala的安装和配置

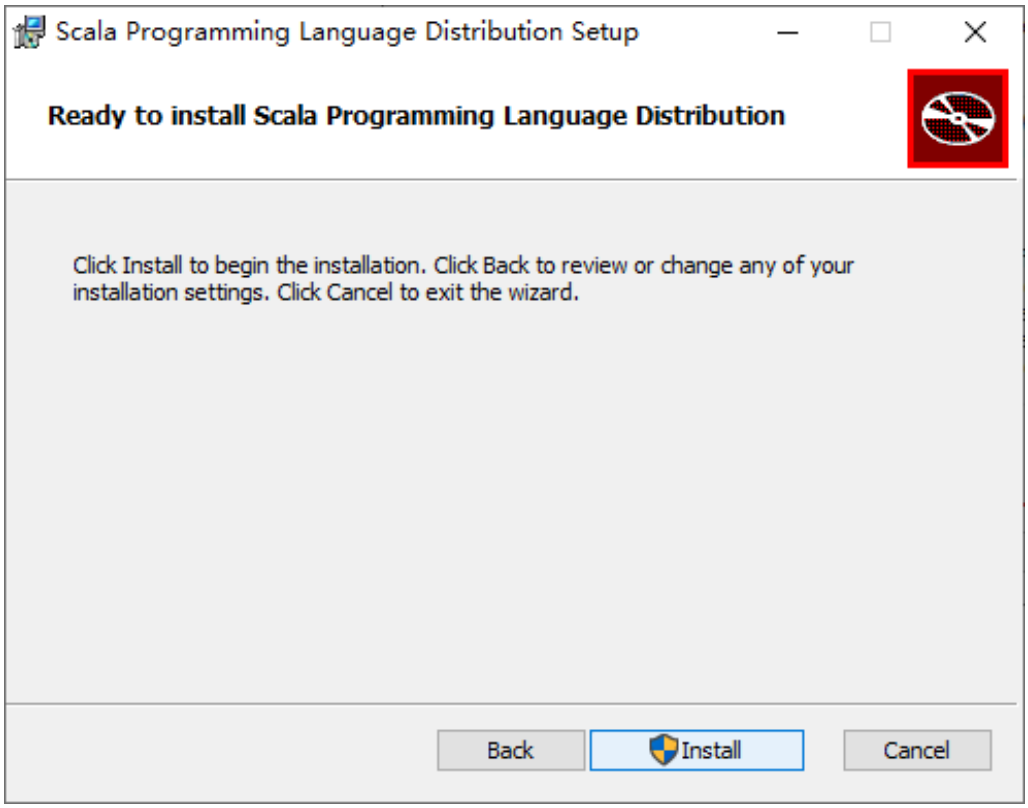
双击安装

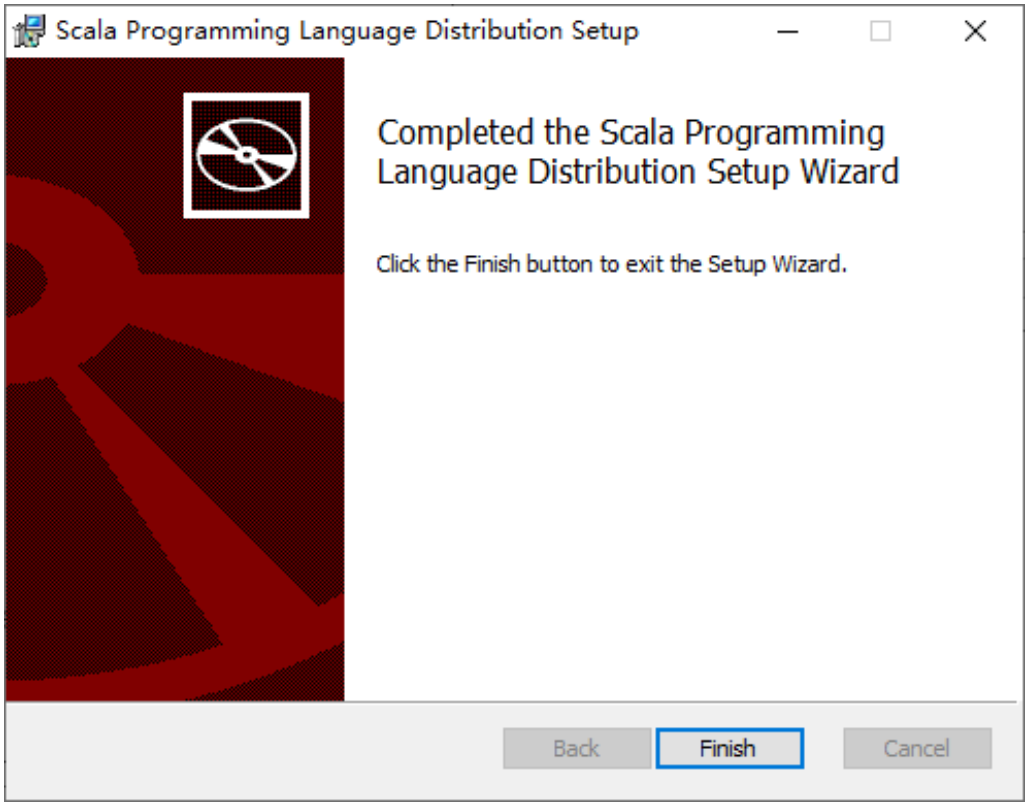


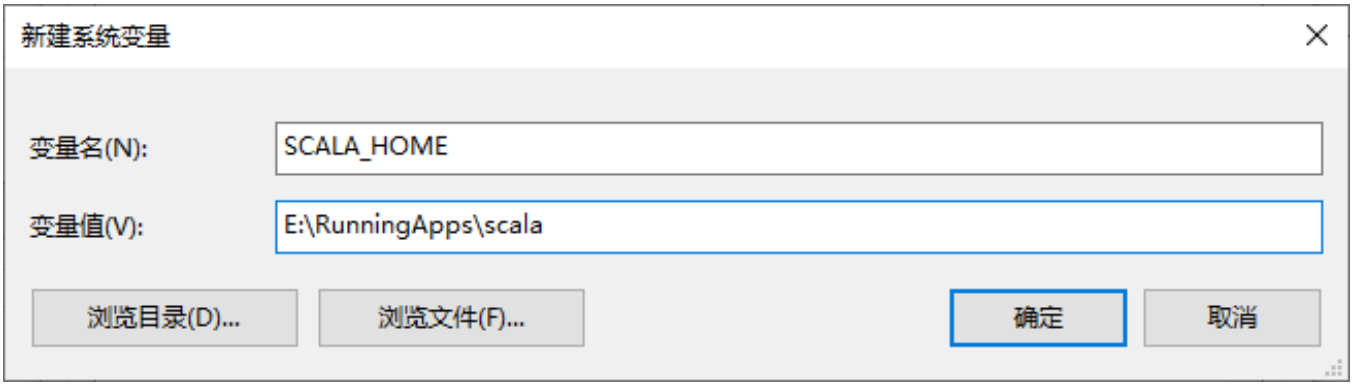

Scala Programming Language Distribution Setup
Welcome to the Scala Programming Language Distribution Setup Wizard
The Setup Wizard will install Scala Programming Language Distribution on your computer. Click Next to continue or Cancel to exit the Setup Wizard.
Back
Next
Cancel


Scala Programming Language Distribution Setup
End-User License Agreement
Please read the following license agreement carefully
Scala is licensed under the Apache License Version 2.0 <https://www.apache.org/licenses/LICENSE-2.0>.
Scala License
I accept the terms in the License Agreement
Print
Back
Next
Cancel

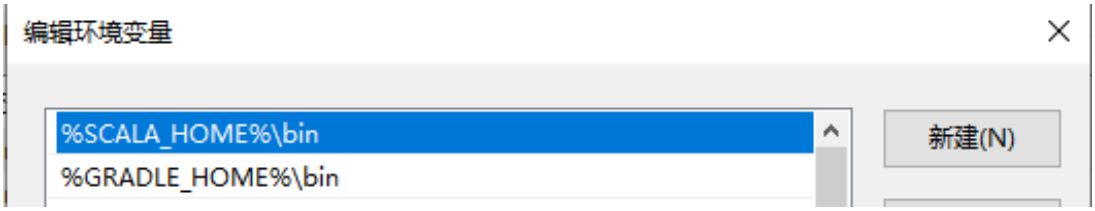

Scala Programming Language Distribution Setup
Custom Setup
Select the way you want features to be installed.
Click the icons in the tree below to change the way features will be installed.
The Scala Programming Languac
The core scala language
Update system PATH
Documentation for the S
The Windows installation of the Scala Programming Language
This feature requires 0KB on your hard drive. It has 3 of 3 subfeatures selected. The subfeatures require 606MB on your hard drive.
Location:
E:\RunningApps\scala\
Browse...
Reset
Disk Usage
Back
Next
Cancel


Scala Programming Language Distribution Setup
Ready to install Scala Programming Language Distribution
Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.
Back
Install
Cancel

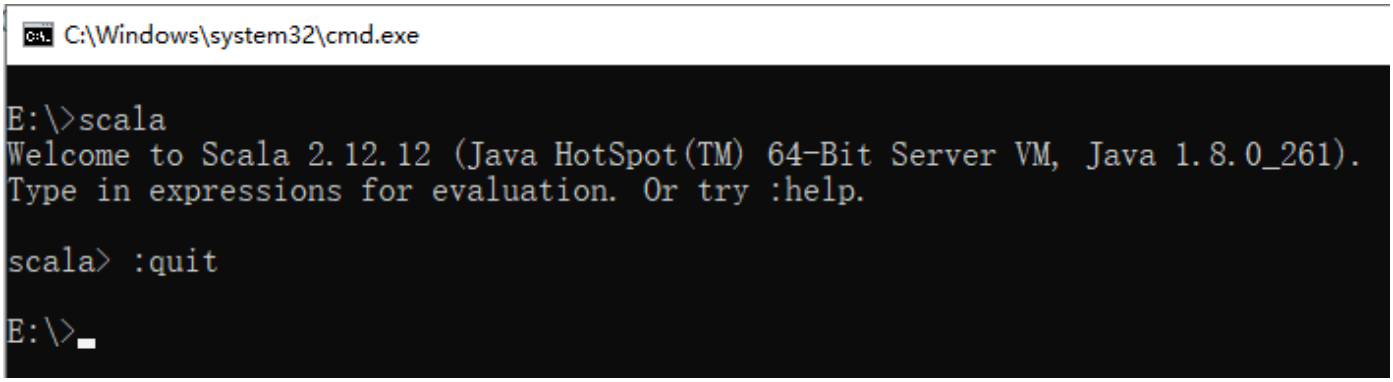




添加gradle的bin目录到PATH中。



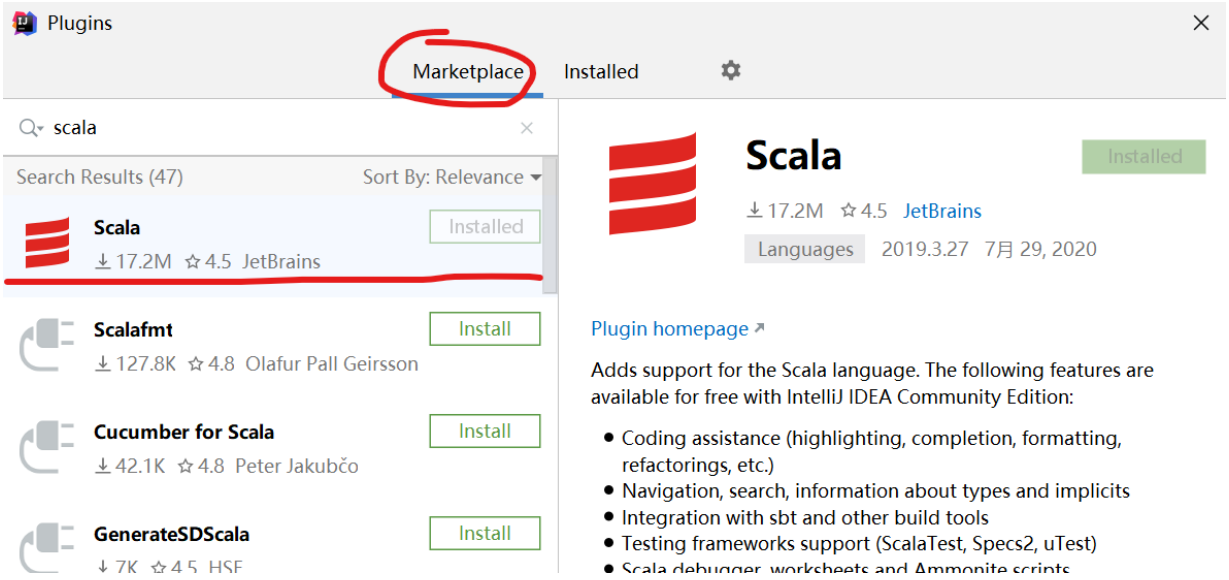
打开cmd，输入`scala`验证：



输入:quit退出Scala的交互式环境。

4.1.3 Idea配置

idea安装Scala插件：



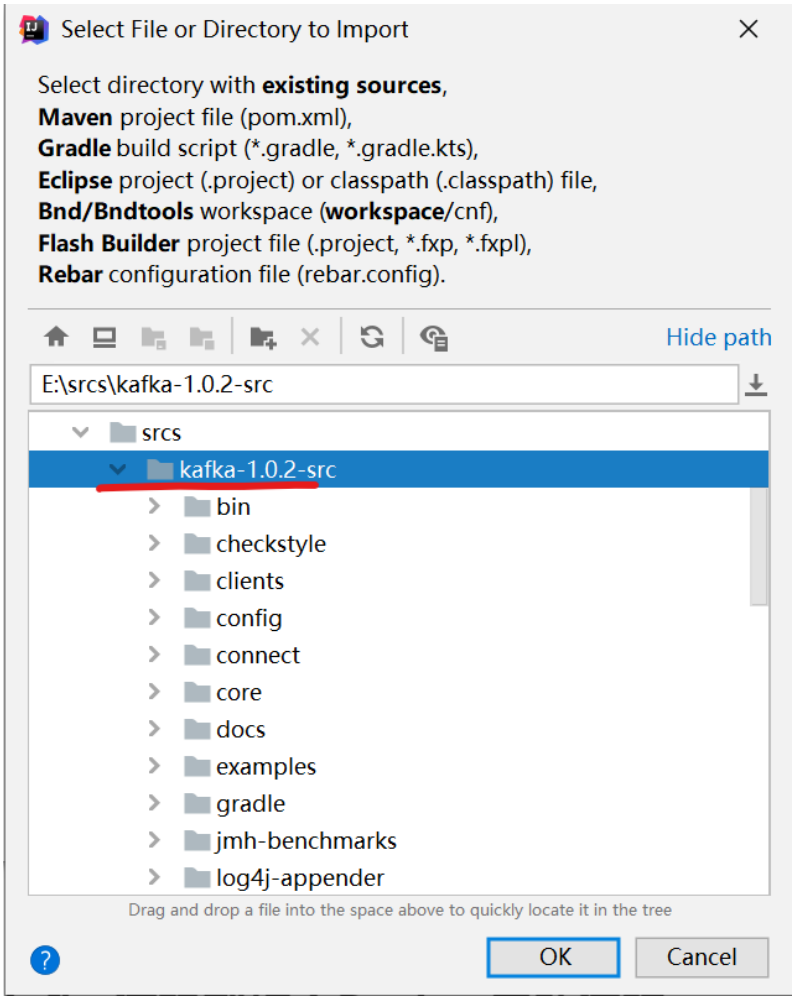
4.1.4 源码操作

解压源码：

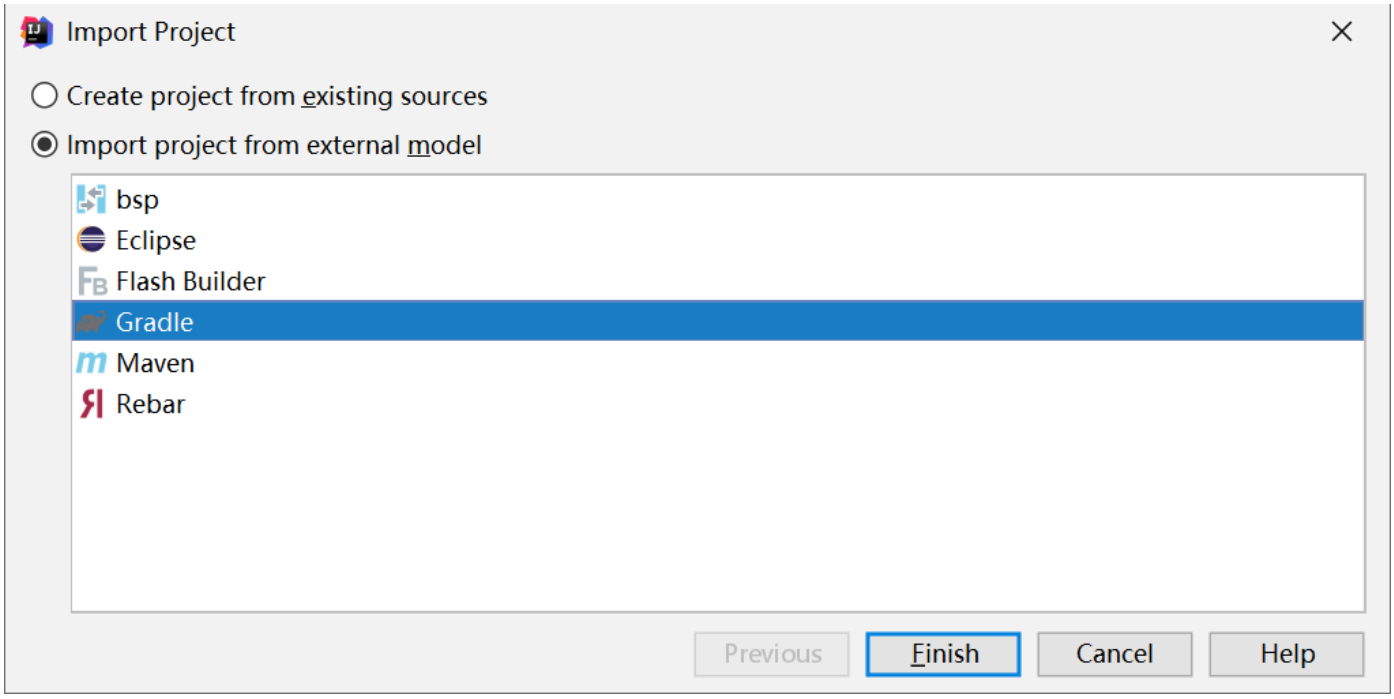
打开CMD，进入kafka-1.0.2-src目录，执行：gradle

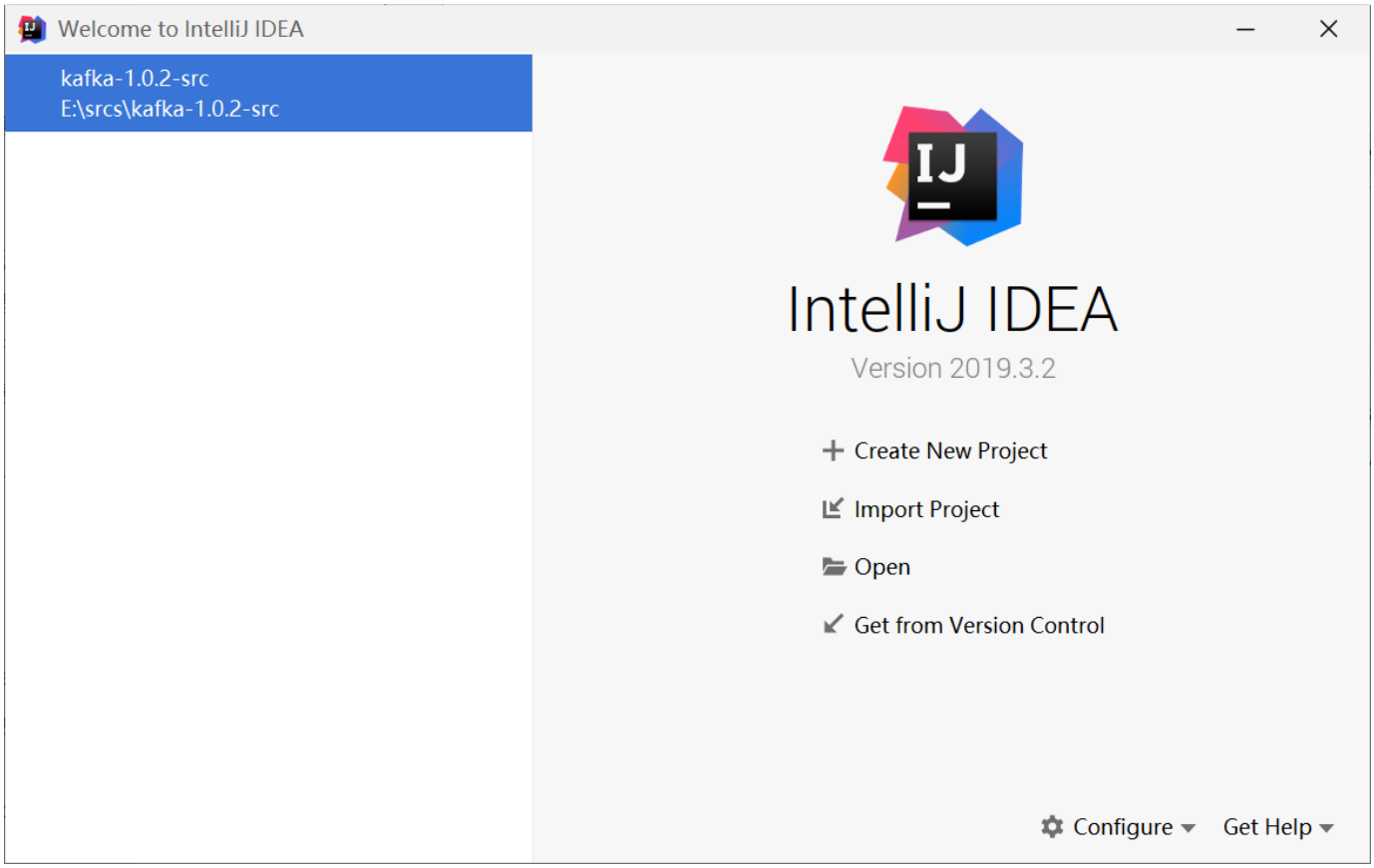
结束后，执行gradle idea（注意不要使用生成的gradlew.bat执行操作）

idea导入源码：


Select File or Directory to Import
Select directory with existing sources,
Maven project file (pom.xml),
Gradle build script (*.gradle, *.gradle.kts),
Eclipse project (.project) or classpath (.classpath) file,
Bnd/Bndtools workspace (workspace/cnf),
Flash Builder project file (.project, *.fxp, *.fxpl),
Rebar configuration file (rebar.config).
Hide path
E:\srcs\kafka-1.0.2-src
srcs
kafka-1.0.2-src
bin
checkstyle
clients
config
connect
core
docs
examples
gradle
jmh-benchmarks
log4j-appender
Drag and drop a file into the space above to quickly locate it in the tree
OK
Cancel

选择Gradle


Import Project
Create project from existing sources
Import project from external model
bsp
Eclipse
Flash Builder
Gradle
Maven
Rebar
Previous
Finish
Cancel
Help



4.2 Kafka源码剖析之Broker启动流程

4.2.1 启动kafka

命令如下：`kafka-server-start.sh /opt/kafka_2.12-1.0.2/config/server.properties` 。

kafka-server-start.sh内容如下：

```
if [ $# -lt 1 ];
then
    echo "USAGE: $0 [-daemon] server.properties [--override property=value]*"
    exit 1
fi
base_dir=$(dirname $0)

if [ "x$KAFKA_LOG4J_OPTS" = "x" ]; then
    export KAFKA_LOG4J_OPTS="-
Dlog4j.configuration=file:$base_dir/../config/log4j.properties"
fi

if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"
```

```
fi

EXTRA_ARGS=${EXTRA_ARGS-'-name kafkaServer -loggc'}

COMMAND=$1
case $COMMAND in
  -daemon)
    EXTRA_ARGS="-daemon "$EXTRA_ARGS
    shift
    ;;
  *)
    ;;
esac

exec $base_dir/kafka-run-class.sh $EXTRA_ARGS kafka.Kafka "$@"
```

4.2.2 查看Kafka.Kafka源码

```
def main(args: Array[String]): Unit = {
    try {
      // 读取启动配置
      val serverProps = getPropsFromArgs(args)
      // 封装KafkaServer
      val kafkaServerStartable = KafkaServerStartable.fromProps(serverProps)

      // register signal handler to log termination due to SIGTERM, SIGHUP and SIGINT (control-c)
      registerLoggingSignalHandler()

      // attach shutdown handler to catch terminating signals as well as normal termination
      // 增加回调监听
      Runtime.getRuntime().addShutdownHook(new Thread("kafka-shutdown-hook") {
        override def run(): Unit = kafkaServerStartable.shutdown()
      })

      // 启动服务
      kafkaServerStartable.startup()
      // 等待
      kafkaServerStartable.awaitShutdown()
    }
    catch {
      case e: Throwable =>
        fatal(e)
        Exit.exit(1)
    }
    Exit.exit(0)
  }
```

上面的`kafkaServerStartabl`封装了`KafkaServer`，最终执行`startup`的是KafkaServer

```
class KafkaServerStartable(val serverConfig: KafkaConfig, reporters: Seq[KafkaMetricsReporter]) extends Logging {
  private val server = new KafkaServer(serverConfig, kafkaMetricsReporters = reporters)

  def this(serverConfig: KafkaConfig) = this(serverConfig, Seq.empty)

  // 启动
  def startup() {
    try server.startup()
    catch {
      case _: Throwable =>
        // KafkaServer.startup() calls shutdown() in case of exceptions, so we invoke `exit` to set the status code
        fatal("Exiting Kafka.")
        Exit.exit(1)
    }
  }

  // 关闭
  def shutdown() {
    try server.shutdown()
    catch {
      case _: Throwable =>
        fatal("Halting Kafka.")
        Exit.halt(1)
    }
  }

  def setServerState(newState: Byte) {
    server.brokerState.newState(newState)
  }
  def awaitShutdown(): Unit = server.awaitShutdown()
}


```

下面来看一下`KafkaServe`r的`startup`方法，启动了很多东西，后面都会用到，代码中也加入了注释

```


def startup() {
    try {
      info("starting")

      // 是否关闭
      if (isShuttingDown.get)
```

```
9           throw new IllegalStateException("Kafka server is still shutting down, cannot re-start!")
10
11        // 是否已启动完成
12        if (startupComplete.get)
13          return
14
15        // 开始启动，并设置已启动变量
16        val canStartup = isStartingUp.compareAndSet(false, true)
17        if (canStartup) {
18          // 设置broker为启动状态
19          brokerState.newState(Starting)
20
21          /* start scheduler */
22          // 启动定时器
23          kafkaScheduler.startup()
24
25          /* setup zookeeper */
26          // 初始化zookeeper配置
27          zkUtils = initZk()
28
29          /* Get or create cluster_id */
30          // 在zookeeper上生成集群Id
31          _clusterId = getOrGenerateClusterId(zkUtils)
32          info(s"Cluster ID = $clusterId")
33
34          /* generate brokerId */
35          // 从配置文件获取brokerId
36          val (brokerId, initialOfflineDirs) = getBrokerIdAndOfflineDirs
37          config.brokerId = brokerId
38          // 日志上下文
39          logContext = new LogContext(s"[KafkaServer id=${config.brokerId}] ")
40          this.logIdent = logContext.logPrefix
41
42          /* create and configure metrics */
43          // 通过配置文件中的MetricsReporter的实现类来创建实例
44          val reporters = config.getConfiguredInstances(KafkaConfig.MetricReporterClassesProp, classOf[MetricsReporter],
45            Map[String, AnyRef](KafkaConfig.BrokerIdProp -> (config.brokerId.toString)).asJava)
46          // 默认监控会增加jmx
47          reporters.add(new JmxReporter(jmxPrefix))
48          val metricConfig = KafkaServer.metricConfig(config)
49          // 创建metric对象
50          metrics = new Metrics(metricConfig, reporters, time, true)
51
52          /* register broker metrics */
53          _brokerTopicStats = new BrokerTopicStats
54
```

```
55          // 初始化配额管理服务,对于每个producer或者consumer,可以对他们produce或者consum的速度上限作出限制
56          quotaManagers = QuotaFactory.instantiate(config, metrics, time, threadNamePrefix.getOrElse(""))
57          // 增加监听器
58          notifyClusterListeners(kafkaMetricsReporters ++ reporters.asScala)
59
60          logDirFailureChannel = new LogDirFailureChannel(config.logDirs.size)
61
62          // 创建日志管理组件，创建时会检查log目录下是否有.kafka_cleanshutdown文件，如果没有的话，broker进入RecoveringFrom UncleanShutdown 状态
63          /* start log manager */
64          logManager = LogManager(config, initialOfflineDirs, zkUtils, brokerState, kafkaScheduler, time, brokerTopicStats, logDirFailureChannel)
65          logManager.startup()
66
67          // 创建元数据管理组件
68          metadataCache = new MetadataCache(config.brokerId)
69          // 创建凭证提供者组件
70          credentialProvider = new CredentialProvider(config.saslEnabledMechanisms)
71
72          // Create and start the socket server acceptor threads so that the bound port is known.
73          // Delay starting processors until the end of the initialization sequence to ensure
74          // that credentials have been loaded before processing authentications.
75          // 创建一个sockerServer组件，并启动。该组件启动后，就会开始接收请求
76          socketServer = new SocketServer(config, metrics, time, credentialProvider)
77          socketServer.startup(startupProcessors = false)
78
79          // 创建一个副本管理组件，并启动该组件
80          /* start replica manager */
81          replicaManager = createReplicaManager(isShuttingDown)
82          replicaManager.startup()
83
84          // 创建kafka控制器,并启动。该控制器启动后broker会尝试去zk创建节点竞争成为controller
85          /* start kafka controller */
86          kafkaController = new KafkaController(config, zkUtils, time, metrics, threadNamePrefix)
87          kafkaController.startup()
88
89          // 创建一个集群管理组件
90          adminManager = new AdminManager(config, metrics, metadataCache, zkUtils)
91
92          // 创建群组协调器,并且启动
93          /* start group coordinator */
94          // Hardcode Time.SYSTEM for now as some Streams tests fail otherwise, it would be good to fix the underlying issue
95          groupCoordinator = GroupCoordinator(config, zkUtils, replicaManager, Time.SYSTEM)
96          groupCoordinator.startup()
```

```
97
98          // 启动事务协调器，带有单独的后台线程调度程序，用于事务到期和日志加载
99          /* start transaction coordinator, with a separate background thread
    scheduler for transaction expiration and log loading */
100         // Hardcode Time.SYSTEM for now as some Streams tests fail otherwise, it
    would be good to fix the underlying issue
101         transactionCoordinator = TransactionCoordinator(config, replicaManager, new
    KafkaScheduler(threads = 1, threadNamePrefix = "transaction-log-manager-"), zkUtils,
    metrics, metadataCache, Time.SYSTEM)
102         transactionCoordinator.startup()
103
104         // 构造授权器
105         /* Get the authorizer and initialize it if one is specified.*/
106         authorizer = Option(config.authorizerClassName).filter(_.nonEmpty).map {
    authorizerClassName =>
107           val authZ = CoreUtils.createObject[Authorizer](authorizerClassName)
108           authZ.configure(config.originals())
109           authZ
110         }
111
112         // 构造api组件，针对各个接口会处理不同的业务
113         /* start processing requests */
114         apis = new KafkaApis(socketServer.requestChannel, replicaManager,
    adminManager, groupCoordinator, transactionCoordinator,
115           kafkaController, zkUtils, config.brokerId, config, metadataCache, metrics,
    authorizer, quotaManagers,
116           brokerTopicStats, clusterId, time)
117
118         // 请求处理池
119         requestHandlerPool = new KafkaRequestHandlerPool(config.brokerId,
    socketServer.requestChannel, apis, time,
120           config.numIoThreads)
121
122         Mx4jLoader.maybeLoad()
123
124         // 动态配置处理器的相关配置
125         /* start dynamic config manager */
126         dynamicConfigHandlers = Map[String, ConfigHandler](ConfigType.Topic -> new
    TopicConfigHandler(logManager, config, quotaManagers),
127           ConfigType.Client -> new ClientIdConfigHandler(quotaManagers),
128           ConfigType.User -> new UserConfigHandler(quotaManagers,
    credentialProvider),
129           ConfigType.Broker -> new BrokerConfigHandler(config, quotaManagers))
130
131         // 初始化动态配置管理器,并启动
132         // Create the config manager. start listening to notifications
133         dynamicConfigManager = new DynamicConfigManager(zkUtils,
    dynamicConfigHandlers)
134         dynamicConfigManager.startup()
135
136         // 通知监听者
```

```
        /* tell everyone we are alive */
        val listeners = config.advertisedListeners.map { endpoint =>
          if (endpoint.port == 0)
            endpoint.copy(port = socketServer.boundPort(endpoint.listenerName))
          else
            endpoint
        }
        // kafka健康检查组件
        kafkaHealthcheck = new KafkaHealthcheck(config.brokerId, listeners, zkUtils,
config.rack,
          config.interBrokerProtocolVersion)
        kafkaHealthcheck.startup()

        // 记录一下恢复点
        // Now that the broker id is successfully registered via KafkaHealthcheck,
checkpoint it
        checkpointBrokerId(config.brokerId)

        // 修改broker状态
        socketServer.startProcessors()
        brokerState.newState(RunningAsBroker)
        shutdownLatch = new CountDownLatch(1)
        startupComplete.set(true)
        isStartingUp.set(false)
        AppInfoParser.registerAppInfo(jmxPrefix, config.brokerId.toString, metrics)
        info("started")
      }
    }
    catch {
      case e: Throwable =>
        fatal("Fatal error during KafkaServer startup. Prepare to shutdown", e)
        isStartingUp.set(false)
        shutdown()
        throw e
    }
  }
```

4.3 Kafka源码剖析之Topic创建流程

4.3.1 Topic创建

有两种创建方式：自动创建、手动创建。在server.properties中配置`auto.create.topics.enable=true`时，kafka在发现该topic不存在的时候会按照默认配置自动创建topic,触发自动创建topic有以下两种情况：

1. Producer向某个不存在的Topic写入消息
2. Consumer从某个不存在的Topic读取消息

4.3.2 手动创建

当`auto.create.topics.enable=false`时，需要手动创建topic，否则消息会发送失败。手动创建topic的方式如下：

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --
partitions 10 --topic kafka_test
```

--replication-factor: 副本数目

--partitions: 分区数据

--topic: topic名字

4.3.3 查看Topic入口

查看脚本文件`kafka-topics.sh`

```
exec $(dirname $0)/kafka-run-class.sh kafka.admin.TopicCommand "$@"
```

最终还是调用的`TopicCommand`类：首先判断参数是否为空，并且create、list、alter、descibe、delete只允许存在一个，进行参数验证，创建`zookeeper`链接，如果参数中包含`create`则开始创建topic，其他情况类似。

```
def main(args: Array[String]): Unit = {

    val opts = new TopicCommandOptions(args)

    // 判断参数长度
    if(args.length == 0)
      CommandLineUtils.printUsageAndDie(opts.parser, "Create, delete, describe, or
change a topic.")

    // create、list、alter、descibe、delete只允许存在一个
    // should have exactly one action
    val actions = Seq(opts.createOpt, opts.listOpt, opts.alterOpt, opts.describeOpt,
opts.deleteOpt).count(opts.options.has _)
    if(actions != 1)
      CommandLineUtils.printUsageAndDie(opts.parser, "Command must include exactly
one action: --list, --describe, --create, --alter or --delete")

    // 参数验证
    opts.checkArgs()

    // 初始化zookeeper链接
    val zkUtils = ZkUtils(opts.options.valueOf(opts.zkConnectOpt),
                          30000,
                          30000,
                          JaasUtils.isZkSecurityEnabled())
    var exitCode = 0
    try {
      if(opts.options.has(opts.createOpt))
```

```
        // 创建topic
        createTopic(zkUtils, opts)
      else if(opts.options.has(opts.alterOpt))
        // 修改topic
        alterTopic(zkUtils, opts)
      else if(opts.options.has(opts.listOpt))
        // 列出所有的topic，bin/kafka-topics.sh --list --zookeeper localhost:2181
        listTopics(zkUtils, opts)
      else if(opts.options.has(opts.describeOpt))
        // 查看topic描述，bin/kafka-topics.sh --describe --zookeeper localhost:2181
        describeTopic(zkUtils, opts)
      else if(opts.options.has(opts.deleteOpt))
        // 删除topic
        deleteTopic(zkUtils, opts)
    } catch {
      case e: Throwable =>
        println("Error while executing topic command : " + e.getMessage)
        error(Utils.stackTrace(e))
        exitCode = 1
    } finally {
      zkUtils.close()
      Exit.exit(exitCode)
    }

  }
```

4.3.4 创建Topic

下面我们主要来看一下`createTopic`的执行过程：

```
def createTopic(zkUtils: ZkUtils, opts: TopicCommandOptions) {
    // 获取topic名称
    val topic = opts.options.valueOf(opts.topicOpt)
    val configs = parseTopicConfigsToBeAdded(opts)
    val ifNotExists = opts.options.has(opts.ifNotExistsOpt)
    if (Topic.hasCollisionChars(topic))
      println("WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is best to use either, but not both.")
    try {
      //如果客户端指定了topic的partition的replicas分配情况，则直接把所有topic的元数据信息持久化写入到zk，
      // topic的properties写入到/config/topics/{topic}目录，
      // topic的PartitionAssignment写入到/brokers/topics/{topic}目录
      if (opts.options.has(opts.replicaAssignmentOpt)) {
        val assignment = parseReplicaAssignment(opts.options.valueOf(opts.replicaAssignmentOpt))
        AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK(zkUtils, topic, assignment, configs, update = false)
      } else {
```

```
        // 否则需要自动生成topic的PartitionAssignment
        CommandLineUtils.checkRequiredArgs(opts.parser, opts.options,
opts.partitionsOpt, opts.replicationFactorOpt)
        // 分区
        val partitions = opts.options.valueOf(opts.partitionsOpt).intValue
        // 副本集
        val replicas = opts.options.valueOf(opts.replicationFactorOpt).intValue
        // 从0.10.x版本开始，kafka可以支持指定broker的机架信息，如果指定了机架信息则在副本分配时
会尽可能地让分区的副本分不到不同的机架上。
        // 指定机架信息是通过kafka的配置文件config/server.properties中的broker.rack参数来配
置的
        val rackAwareMode = if (opts.options.has(opts.disableRackAware))
RackAwareMode.Disabled
        else RackAwareMode.Enforced
        AdminUtils.createTopic(zkUtils, topic, partitions, replicas, configs,
rackAwareMode)
      }
      println("Created topic \"%s\".".format(topic))
    } catch {
      case e: TopicExistsException => if (!ifNotExists) throw e
    }
  }
```

1. 如果客户端指定了topic的partition的replicas分配情况，则直接把所有topic的元数据信息持久化写入到zk，topic的properties写入到/config/topics/{topic}目录， topic的PartitionAssignment写入到/brokers/topics/{topic}目录
2. 根据分区数量、副本集、是否指定机架来自动生成topic的分区数据
3. 下面继续来看`AdminUtils.createTopic`方法

```
def createTopic(zkUtils: ZkUtils,
                  topic: String,
                  partitions: Int,
                  replicationFactor: Int,
                  topicConfig: Properties = new Properties,
                  rackAwareMode: RackAwareMode = RackAwareMode.Enforced) {
    // 获取集群中每个broker的brokerId和机架信息信息的列表，为下面的
    val brokerMetadatas = getBrokerMetadatas(zkUtils, rackAwareMode)
    // 根据是否禁用指定机架策略来生成分配策略
    val replicaAssignment =
AdminUtils.assignReplicasToBrokers(brokerMetadatas, partitions,
replicationFactor)
    // 在zookeeper中创建或更新主题分区分配路径
    AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK(zkUtils, topic,
replicaAssignment, topicConfig)
  }
```

4. 下面继续来看`AdminUtils.assignReplicasToBrokers`方法

```
def assignReplicasToBrokers(brokerMetadatas: Seq[BrokerMetadata],
```

```
                                nPartitions: Int,
                                replicationFactor: Int,
                                fixedStartIndex: Int = -1,
                                startPartitionId: Int = -1): Map[Int, Seq[Int]]
= {
    if (nPartitions <= 0)
      // 分区个数partitions不能小于等于0
      throw new InvalidPartitionsException("Number of partitions must be
larger than 0.")
    if (replicationFactor <= 0)
      // 副本个数replicationFactor不能小于等于0
      throw new InvalidReplicationFactorException("Replication factor must be
larger than 0.")
    if (replicationFactor > brokerMetadatas.size)
      // 副本个数replicationFactor不能大于broker的节点个数
      throw new InvalidReplicationFactorException(s"Replication factor:
$replicationFactor larger than available brokers: ${brokerMetadatas.size}.")
    if (brokerMetadatas.forall(_.rack.isEmpty))
      // 没有指定机架信息的情况
      assignReplicasToBrokersRackUnaware(nPartitions, replicationFactor,
brokerMetadatas.map(_.id), fixedStartIndex,
        startPartitionId)
    else {
      // 针对指定机架信息的情况，更加复杂一点
      if (brokerMetadatas.exists(_.rack.isEmpty))
        throw new AdminOperationException("Not all brokers have rack
information for replica rack aware assignment.")
      assignReplicasToBrokersRackAware(nPartitions, replicationFactor,
brokerMetadatas, fixedStartIndex,
        startPartitionId)
    }
  }
```

1. 未指定机架策略

```
private def assignReplicasToBrokersRackUnaware(nPartitions: Int,
                                               replicationFactor:
Int,
                                               brokerList: Seq[Int],
                                               fixedStartIndex: Int,
                                               startPartitionId:
Int): Map[Int, Seq[Int]] = {
    val ret = mutable.Map[Int, Seq[Int]]()
    val brokerArray = brokerList.toArray
    val startIndex = if (fixedStartIndex >= 0) fixedStartIndex else
rand.nextInt(brokerArray.length)
    var currentPartitionId = math.max(0, startPartitionId)
    var nextReplicaShift = if (fixedStartIndex >= 0) fixedStartIndex
else rand.nextInt(brokerArray.length)
    for (_ <- 0 until nPartitions) {
```

```
      if (currentPartitionId > 0 && (currentPartitionId % brokerArray.length == 0))
        nextReplicaShift += 1
      val firstReplicaIndex = (currentPartitionId + startIndex) % brokerArray.length
      val replicaBuffer = mutable.ArrayBuffer(brokerArray(firstReplicaIndex))
      for (j <- 0 until replicationFactor - 1)
        replicaBuffer += brokerArray(replicaIndex(firstReplicaIndex, nextReplicaShift, j, brokerArray.length))
      ret.put(currentPartitionId, replicaBuffer)
      currentPartitionId += 1
    }
    ret
  }
```

遍历每个分区partition然后从brokerArray（brokerId的列表）中选取replicationFactor个brokerId分配给这个partition.

创建一个可变的Map用来存放本方法将要返回的结果，即分区partition和分配副本的映射关系。由于fixedStartIndex为-1，所以startIndex是一个随机数，用来计算一个起始分配的brokerId，同时由于startPartitionId为-1，所以currentPartitionId的值为0，可见默认创建topic时总是从编号为0的分区依次轮询进行分配。nextReplicaShift表示下一次副本分配相对于前一次分配的位移量，这个字面上理解有点绕，不如举个例子：假设集群中有3个broker节点，即代码中的brokerArray，创建某topic有3个副本和6个分区，那么首先从partitionId（partition的编号）为0的分区开始进行分配，假设第一次计算（由rand.nextInt(brokerArray.length)随机）到nextReplicaShift为1，第一次随机到的startIndex为2，那么partitionId为0的第一个副本的位置（这里指的是brokerArray的数组下标）firstReplicaIndex = (currentPartitionId + startIndex) % brokerArray.length =（0+2）%3 = 2，第二个副本的位置为replicaIndex(firstReplicaIndex, nextReplicaShift, j, brokerArray.length) = replicaIndex(2, nextReplicaShift+1,0, 3)=?。

继续计算 replicaIndex(2, nextReplicaShift+1,0, 3) = replicaIndex(2, 2,0, 3)= (2+(1+(2+0)%(3-1)))%3=0。继续计算下一个副本的位置replicaIndex(2, 2,1, 3) = (2+(1+(2+1)%(3-1)))%3=1。所以partitionId为0的副本分配位置列表为[2,0,1]，如果brokerArray正好是从0开始编号，也正好是顺序不间断的，即brokerArray为[0,1,2]的话，那么当前partitionId为0的副本分配策略为[2,0,1]。如果brokerId不是从零开始，也不是顺序的（有可能之前集群的其中broker几个下线了），最终的brokerArray为[2,5,8]，那么partitionId为0的分区的副本分配策略为[8,2,5]。为了便于说明问题，可以简单的假设brokerArray就是[0,1,2]。

同样计算下一个分区，即partitionId为1的副本分配策略。此时nextReplicaShift还是为2，没有满足自增的条件。这个分区的firstReplicaIndex = (1+2)%3=0。第二个副本的位置replicaIndex(0,2,0,3) = (0+(1+(2+0)%(3-1)))%3 = 1，第三个副本的位置replicaIndex(0,2,1,3) = 2，最终partitionId为2的分区分配策略为[0,1,2]

2. 指定机架策略

```
private def assignReplicasToBrokersRackAware(nPartitions: Int,
                                             replicationFactor: Int,
                                             brokerMetadatas: Seq[BrokerMetadata],
                                             fixedStartIndex: Int,
```

```
                                                startPartitionId: Int):
Map[Int, Seq[Int]] = {
    val brokerRackMap = brokerMetadatas.collect { case
BrokerMetadata(id, Some(rack)) =>
      id -> rack
    }.toMap
    val numRacks = brokerRackMap.values.toSet.size
    val arrangedBrokerList =
getRackAlternatedBrokerList(brokerRackMap)
    val numBrokers = arrangedBrokerList.size
    val ret = mutable.Map[Int, Seq[Int]]()
    val startIndex = if (fixedStartIndex >= 0) fixedStartIndex else
rand.nextInt(arrangedBrokerList.size)
    var currentPartitionId = math.max(0, startPartitionId)
    var nextReplicaShift = if (fixedStartIndex >= 0) fixedStartIndex
else rand.nextInt(arrangedBrokerList.size)
    for (_ <- 0 until nPartitions) {
      if (currentPartitionId > 0 && (currentPartitionId %
arrangedBrokerList.size == 0))
        nextReplicaShift += 1
      val firstReplicaIndex = (currentPartitionId + startIndex) %
arrangedBrokerList.size
      val leader = arrangedBrokerList(firstReplicaIndex)
      val replicaBuffer = mutable.ArrayBuffer(leader)
      val racksWithReplicas = mutable.Set(brokerRackMap(leader))
      val brokersWithReplicas = mutable.Set(leader)
      var k = 0
      for (_ <- 0 until replicationFactor - 1) {
        var done = false
        while (!done) {
          val broker =
arrangedBrokerList(replicaIndex(firstReplicaIndex, nextReplicaShift *
numRacks, k, arrangedBrokerList.size))
          val rack = brokerRackMap(broker)
          // Skip this broker if
          // 1. there is already a broker in the same rack that has
assigned a replica AND there is one or more racks
          //    that do not have any replica, or
          // 2. the broker has already assigned a replica AND there is
one or more brokers that do not have replica assigned
          if ((!racksWithReplicas.contains(rack) ||
racksWithReplicas.size == numRacks)
              && (!brokersWithReplicas.contains(broker) ||
brokersWithReplicas.size == numBrokers)) {
            replicaBuffer += broker
            racksWithReplicas += rack
            brokersWithReplicas += broker
            done = true
          }
          k += 1
        }
```

```
      }
      ret.put(currentPartitionId, replicaBuffer)
      currentPartitionId += 1
    }
    ret
  }
```

1. assignReplicasToBrokersRackUnaware的执行前提是所有的broker都没有配置机架信息，而assignReplicasToBrokersRackAware的执行前提是所有的broker都配置了机架信息，如果出现部分broker配置了机架信息而另一部分没有配置的话，则会抛出AdminOperationException的异常，如果还想要顺利创建topic的话，此时需加上"--disable-rack-aware"
2. 第一步获得brokerId和rack信息的映射关系列表brokerRackMap ，之后调用getRackAlternatedBrokerList()方法对brokerRackMap做进一步的处理生成一个brokerId的列表。举例：假设目前有3个机架rack1、rack2和rack3，以及9个broker，分别对应关系如下：

```
rack1: 0, 1, 2
rack2: 3, 4, 5
rack3: 6, 7, 8
```

那么经过getRackAlternatedBrokerList()方法处理过后就变成了[0, 3, 6, 1, 4, 7, 2, 5, 8]这样一个列表，显而易见的这是轮询各个机架上的broker而产生的，之后你可以简单的将这个列表看成是brokerId的列表，对应assignReplicasToBrokersRackUnaware()方法中的brokerArray，但是其中包含了简单的机架分配信息。之后的步骤也和未指定机架信息的算法类似，同样包含startIndex、currentPartiionId, nextReplicaShift的概念，循环为每一个分区分配副本。分配副本时处理第一个副本之外，其余的也调用replicaIndex方法来获得一个broker，但是这里和assignReplicasToBrokersRackUnaware()不同的是，这里不是简单的将这个broker添加到当前分区的副本列表之中，还要经过一层的筛选，满足以下任意一个条件的broker不能被添加到当前分区的副本列表之中：

```
1. 如果此broker所在的机架中已经存在一个broker拥有该分区的副本，并且还有其他的机架中没有任何一个broker拥有该分区的副本。对应代码中的(!racksWithReplicas.contains(rack) || racksWithReplicas.size == numRacks)

2. 如果此broker中已经拥有该分区的副本，并且还有其他broker中没有该分区的副本。对应代码中的(!brokersWithReplicas.contains(broker) || brokersWithReplicas.size == numBrokers))
```

5. 无论是带机架信息的策略还是不带机架信息的策略，上层调用方法AdminUtils.assignReplicasToBrokers()最后都是获得一个[Int, Seq[Int]]类型的副本分配列表，其最后作为kafka zookeeper节点/brokers/topics/{topic-name}节点数据。至此kafka的topic创建就讲解完了，有些同学会感到很疑问，全文通篇（包括上一篇）都是在讲述如何分配副本，最后得到的也不过是个分配的方案，并没有真正创建这些副本的环节，其实这个观点没有任何问题，对于通过kafka提供的kafka-topics.sh脚本创建topic的方法来说，它只是提供一个副本的分配方案，并在kafka zookeeper中创建相应的节点而已。kafka broker的服务会注册监听/brokers/topics/目录下是否有节点变化，如果有新节点创建就会监听到，然后根据其节点中的

数据（即topic的分区副本分配方案）来创建对应的副本。

4.4 Kafka源码剖析之Producer生产者流程

4.4.1 Producer示例

首先我们先通过一段代码来展示`KafkaProducer`的使用方法。在下面的示例中，我们使用`KafkaProducer`实现向kafka发送消息的功能。在示例程序中，首先将`KafkaProduce`使用的配置写入到`Properties`中，每项配置的具体含义在注释中进行解释。之后以此`Properties`对象为参数构造`KafkaProducer`对象，最后通过`send`方法完成发送，代码中包含同步发送、异步发送两种情况。

```
public static void main(String[] args) throws ExecutionException,
InterruptedException {
        Properties props = new Properties();
        // 客户端id
        props.put("client.id", "KafkaProducerDemo");
        // kafka地址,列表格式为host1:port1,host2:port2,…，无需添加所有的集群地址，kafka会根据提供的地址发现其他的地址（建议多提供几个，以防提供的服务器关闭）
        props.put("bootstrap.servers", "localhost:9092");
        // 发送返回应答方式
        // 0:Producer 往集群发送数据不需要等到集群的返回，不确保消息发送成功。安全性最低但是效率最高。
        // 1:Producer 往集群发送数据只要 Leader 应答就可以发送下一条，只确保Leader接收成功。
        // -1或者all：Producer 往集群发送数据需要所有的ISR Follower都完成从Leader的同步才会发送下一条，确保Leader发送成功和所有的副本都成功接收。安全性最高，但是效率最低。
        props.put("acks", "all");
        // 重试次数
        props.put("retries", 0);
        // 重试间隔时间
        props.put("retries.backoff.ms", 100);
        // 批量发送的大小
        props.put("batch.size", 16384);
        // 一个Batch被创建之后，最多过多久，不管这个Batch有没有写满，都必须发送出去
        props.put("linger.ms", 10);
        // 缓冲区大小
        props.put("buffer.memory", 33554432);
        // key序列化方式
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        // value序列化方式
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        // topic
        String topic = "lagou_edu";

        Producer<String, String> producer = new KafkaProducer<>(props);
        AtomicInteger count = new AtomicInteger();
        while (true) {
```

```
            int num = count.get();
            String key = Integer.toString(num);
            String value = Integer.toString(num);
            ProducerRecord<String, String> record = new ProducerRecord<>(topic, key,
value);
            if (num % 2 == 0) {
                // 偶数异步发送
                // 第一个参数record封装了topic、key、value
                // 第二个参数是一个callback对象，当生产者接收到kafka发来的ACK确认消息时，会调用
此CallBack对象的onComplete方法
                producer.send(record, (recordMetadata, e) -> {
                    System.out.println("num:" + num + " topic:" +
recordMetadata.topic() + " offset:" + recordMetadata.offset());
                });
            } else {
                // 同步发送
                // KafkaProducer.send方法返回的类型是Future<RecordMetadata>，通过get方法阻
塞当前线程，等待kafka服务端ACK响应
                producer.send(record).get();
            }
            count.incrementAndGet();
            TimeUnit.MILLISECONDS.sleep(100);
        }
    }
```

4.4.1.1 同步发送

1. KafkaProducer.send方法返回的类型是Future<RecordMetadata>，通过get方法阻塞当前线程，等待kafka服务端ACK响应

```
producer.send(record).get()
```

4.4.1.2 异步发送

1. 第一个参数record封装了topic、key、value
2. 第二个参数是一个callback对象，当生产者接收到kafka发来的ACK确认消息时，会调用此CallBack对象的onComplete方法

```
producer.send(record, (recordMetadata, e) -> {
                    System.out.println("num:" + num + " topic:" +
recordMetadata.topic() + " offset:" + recordMetadata.offset());
                });
```

4.4.2 KafkaProducer实例化

了解了`KafkaProducer`的基本使用，开始深入了解的KafkaProducer原理和实现，先看一下构造方法核心逻辑

```
private KafkaProducer(ProducerConfig config, Serializer<K> keySerializer,
Serializer<V> valueSerializer) {
        try {
            // 获取用户的配置
            Map<String, Object> userProvidedConfigs = config.originals();
            this.producerConfig = config;
            // 系统时间
            this.time = Time.SYSTEM;
            // 获取client.id配置
            String clientId = config.getString(ProducerConfig.CLIENT_ID_CONFIG);
            // 如果client.id为空，设置默认值:producer-1
            if (clientId.length() <= 0)
                clientId = "producer-" +
PRODUCER_CLIENT_ID_SEQUENCE.getAndIncrement();
            this.clientId = clientId;

            // 获取事务id,如果没有配置则为null
            String transactionalId =
userProvidedConfigs.containsKey(ProducerConfig.TRANSACTIONAL_ID_CONFIG) ?
                    (String)
userProvidedConfigs.get(ProducerConfig.TRANSACTIONAL_ID_CONFIG) : null;
            LogContext logContext;
            if (transactionalId == null)
                logContext = new LogContext(String.format("[Producer clientId=%s] ",
clientId));
            else
                logContext = new LogContext(String.format("[Producer clientId=%s,
transactionalId=%s] ", clientId, transactionalId));
            log = logContext.logger(KafkaProducer.class);
            log.trace("Starting the Kafka producer");

            // 创建client-id的监控map
            Map<String, String> metricTags = Collections.singletonMap("client-id",
clientId);

            // 设置监控配置，包含样本量、取样时间窗口、记录级别
            MetricConfig metricConfig = new
MetricConfig().samples(config.getInt(ProducerConfig.METRICS_NUM_SAMPLES_CONFIG))

.timeWindow(config.getLong(ProducerConfig.METRICS_SAMPLE_WINDOW_MS_CONFIG),
TimeUnit.MILLISECONDS)

.recordLevel(Sensor.RecordingLevel.forName(config.getString(ProducerConfig.METRICS_R
ECORDING_LEVEL_CONFIG)))
                    .tags(metricTags);

            // 监控数据上报类
```

```
            List<MetricsReporter> reporters =
config.getConfiguredInstances(ProducerConfig.METRIC_REPORTER_CLASSES_CONFIG,
                    MetricsReporter.class);
            reporters.add(new JmxReporter(JMX_PREFIX));
            this.metrics = new Metrics(metricConfig, reporters, time);

            // 生成生产者监控
            ProducerMetrics metricsRegistry = new ProducerMetrics(this.metrics);

            // 分区类
            this.partitioner =
config.getConfiguredInstance(ProducerConfig.PARTITIONER_CLASS_CONFIG,
Partitioner.class);

            // 重试时间 retry.backoff.ms 默认100ms
            long retryBackoffMs =
config.getLong(ProducerConfig.RETRY_BACKOFF_MS_CONFIG);
            if (keySerializer == null) {
                // 反射生成key序列化方式
                this.keySerializer =
ensureExtended(config.getConfiguredInstance(ProducerConfig.KEY_SERIALIZER_CLASS_CONF
IG,
                        Serializer.class));
                this.keySerializer.configure(config.originals(), true);
            } else {
                config.ignore(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG);
                this.keySerializer = ensureExtended(keySerializer);
            }
            if (valueSerializer == null) {
                // 反射生成key序列化方式
                this.valueSerializer =
ensureExtended(config.getConfiguredInstance(ProducerConfig.VALUE_SERIALIZER_CLASS_CO
NFIG,
                        Serializer.class));
                this.valueSerializer.configure(config.originals(), false);
            } else {
                config.ignore(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG);
                this.valueSerializer = ensureExtended(valueSerializer);
            }

            // load interceptors and make sure they get clientId
            // 确认client.id添加到用户的配置里面
            userProvidedConfigs.put(ProducerConfig.CLIENT_ID_CONFIG, clientId);

            // 获取多个拦截器,为空则不处理
            List<ProducerInterceptor<K, V>> interceptorList = (List) (new
ProducerConfig(userProvidedConfigs,
false)).getConfiguredInstances(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
                    ProducerInterceptor.class);
            this.interceptors = interceptorList.isEmpty() ? null : new
ProducerInterceptors<>(interceptorList);
```

```
76
77              // 集群资源监听器,在元数据变更时会有通知
78              ClusterResourceListeners clusterResourceListeners =
    configureClusterResourceListeners(keySerializer, valueSerializer, interceptorList,
    reporters);
79
80              // 生产者每隔一段时间都要去更新一下集群的元数据,默认5分钟
81              this.metadata = new Metadata(retryBackoffMs,
    config.getLong(ProducerConfig.METADATA_MAX_AGE_CONFIG),
82                      true, true, clusterResourceListeners);
83
84              // 生产者往服务端发送消息的时候，规定一条消息最大多大?
85              // 如果你超过了这个规定消息的大小，你的消息就不能发送过去。
86              // 默认是1M，这个值偏小，在生产环境中，我们需要修改这个值。
87              // 经验值是10M。但是大家也可以根据自己公司的情况来。
88              this.maxRequestSize =
    config.getInt(ProducerConfig.MAX_REQUEST_SIZE_CONFIG);
89              //指的是缓存大小
90              //默认值是32M，这个值一般是够用，如果有特殊情况的时候，我们可以去修改这个值。
91              this.totalMemorySize =
    config.getLong(ProducerConfig.BUFFER_MEMORY_CONFIG);
92              // kafka是支持压缩数据的，可以设置压缩格式,默认是不压缩，支持gzip、snappy、lz4
93              // 一次发送出去的消息就更多。生产者这儿会消耗更多的cpu.
94              this.compressionType =
    CompressionType.forName(config.getString(ProducerConfig.COMPRESSION_TYPE_CONFIG));
95
96              // 配置控制了KafkaProducer.send()并将KafkaProducer.partitionsFor()被阻塞多长
    时间,由于缓冲区已满或元数据不可用，这些方法可能会被阻塞止
97              this.maxBlockTimeMs =
    config.getLong(ProducerConfig.MAX_BLOCK_MS_CONFIG);
98
99              // 控制客户端等待请求响应的最长时间。如果在超时过去之前未收到响应，客户端将在必要时重
    新发送请求，或者如果重试耗尽，请求失败
100             this.requestTimeoutMs =
    config.getInt(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG);
101
102             // 事务管理器
103             this.transactionManager = configureTransactionState(config, logContext,
    log);
104
105             // 重试次数
106             int retries = configureRetries(config, transactionManager != null, log);
107
108             // 使用幂等性，需要将 enable.idempotence 配置项设置为true。并且它对单个分区的发
    送，一次性最多发送5条
109             int maxInflightRequests = configureInflightRequests(config,
    transactionManager != null);
110
111             // 如果开启了幂等性，但是用户指定的ack不为 -1，则会抛出异常
112             short acks = configureAcks(config, transactionManager != null, log);
113
```

```
114             this.apiVersions = new ApiVersions();
115 
116             // 创建核心组件：记录累加器
117             this.accumulator = new RecordAccumulator(logContext,
118                     config.getInt(ProducerConfig.BATCH_SIZE_CONFIG),
119                     this.totalMemorySize,
120                     this.compressionType,
121                     config.getLong(ProducerConfig.LINGER_MS_CONFIG),
122                     retryBackoffMs,
123                     metrics,
124                     time,
125                     apiVersions,
126                     transactionManager);
127 
128             // 获取broker地址列表
129             List<InetSocketAddress> addresses =
ClientUtils.parseAndValidateAddresses(config.getList(ProducerConfig.BOOTSTRAP_SERVER
S_CONFIG));
130 
131             // 更新元数据
132             this.metadata.update(Cluster.bootstrap(addresses), Collections.
<String>emptySet(), time.milliseconds());
133 
134             // 创建通道，是否需要加密
135             ChannelBuilder channelBuilder =
ClientUtils.createChannelBuilder(config);
136             Sensor throttleTimeSensor =
Sender.throttleTimeSensor(metricsRegistry.senderMetrics);
137 
138             // 初始化了一个重要的管理网路的组件
139             // connections.max.idle.ms：默认值是9分钟，一个网络连接最多空闲多久，超过这个空闲
时间，就关闭这个网络连接。
140             // max.in.flight.requests.per.connection：默认是5，producer向broker发送数据
的时候，其实是有多个网络连接。每个网络连接可以忍受 producer端发送给broker 消息然后消息没有响应的个
数
141             NetworkClient client = new NetworkClient(
142                     new
Selector(config.getLong(ProducerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG),
143                             this.metrics, time, "producer", channelBuilder,
logContext),
144                     this.metadata,
145                     clientId,
146                     maxInflightRequests,
147                     config.getLong(ProducerConfig.RECONNECT_BACKOFF_MS_CONFIG),
148                     config.getLong(ProducerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG),
149                     config.getInt(ProducerConfig.SEND_BUFFER_CONFIG),
150                     config.getInt(ProducerConfig.RECEIVE_BUFFER_CONFIG),
151                     this.requestTimeoutMs,
152                     time,
153                     true,
154                     apiVersions,
```

```
                    throttleTimeSensor,
                    logContext);

            // 发送线程
            this.sender = new Sender(logContext,
                    client,
                    this.metadata,
                    this.accumulator,
                    maxInflightRequests == 1,
                    config.getInt(ProducerConfig.MAX_REQUEST_SIZE_CONFIG),
                    acks,
                    retries,
                    metricsRegistry.senderMetrics,
                    Time.SYSTEM,
                    this.requestTimeoutMs,
                    config.getLong(ProducerConfig.RETRY_BACKOFF_MS_CONFIG),
                    this.transactionManager,
                    apiVersions);

            // 线程名称
            String ioThreadName = NETWORK_THREAD_PREFIX + " | " + clientId;

            // 启动守护线程
            this.ioThread = new KafkaThread(ioThreadName, this.sender, true);
            this.ioThread.start();
            this.errors = this.metrics.sensor("errors");

            // 把用户配置的参数，但是没有用到的打印出来
            config.logUnused();
            AppInfoParser.registerAppInfo(JMX_PREFIX, clientId, metrics);
            log.debug("Kafka producer started");
        } catch (Throwable t) {
            // call close methods if internal objects are already constructed this is to prevent resource leak. see KAFKA-2121
            close(0, TimeUnit.MILLISECONDS, true);
            // now propagate the exception
            throw new KafkaException("Failed to construct kafka producer", t);
        }
    }
```

4.4.2 消息发送过程

Kafka消息实际发送以`send`方法为入口：

```
@Override
    public Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback)
{
        // intercept the record, which can be potentially modified; this method does
not throw exceptions
        ProducerRecord<K, V> interceptedRecord = this.interceptors == null ? record :
this.interceptors.onSend(record);
        return doSend(interceptedRecord, callback);
    }
```

4.4.2.1 拦截器

首先方法会先进入拦截器集合`ProducerInterceptors`，`onSend`方法是遍历拦截器`onSend`方法，拦截器的目的是将数据处理加工，`kafka`本身并没有给出默认的拦截器的实现。如果需要使用拦截器功能，必须自己实现**ProducerInterceptor** 接口。

```
public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record) {
        ProducerRecord<K, V> interceptRecord = record;
        // 遍历所有拦截器，顺序执行，如果有异常只打印日志，不会向上抛出
        for (ProducerInterceptor<K, V> interceptor : this.interceptors) {
            try {
                interceptRecord = interceptor.onSend(interceptRecord);
            } catch (Exception e) {
                // do not propagate interceptor exception, log and continue calling
other interceptors
                // be careful not to throw exception from here
                if (record != null)
                    log.warn("Error executing interceptor onSend callback for topic:
{}, partition: {}", record.topic(), record.partition(), e);
                else
                    log.warn("Error executing interceptor onSend callback", e);
            }
        }
        return interceptRecord;
    }
```

4.4.2.2 拦截器核心逻辑

`ProducerInterceptor`接口包括三个方法：

1. onSend(ProducerRecord)：该方法封装进KafkaProducer.send方法中，即它运行在用户主线程中的。Producer确保在消息被序列化以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
2. onAcknowledgement(RecordMetadata, Exception)：该方法会在消息被应答之前或消息发送失败时调用，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
3. close：关闭interceptor，主要用于执行一些资源清理工作
4. 拦截器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。

4.4.2.3 发送五步骤

下面仔细来看一下`doSend`方法的运行过程：

```
private Future<RecordMetadata> doSend(ProducerRecord<K, V> record, Callback callback)
{
        // 首先创建一个主题分区类
        TopicPartition tp = null;
        try {
            // first make sure the metadata for the topic is available
            // 首先确保该topic的元数据可用
            ClusterAndWaitTime clusterAndWaitTime = waitOnMetadata(record.topic(),
record.partition(), maxBlockTimeMs);
            long remainingWaitMs = Math.max(0, maxBlockTimeMs -
clusterAndWaitTime.waitedOnMetadataMs);
            Cluster cluster = clusterAndWaitTime.cluster;
            // 序列化 record 的 key 和 value
            byte[] serializedKey;
            try {
                serializedKey = keySerializer.serialize(record.topic(),
record.headers(), record.key());
            } catch (ClassCastException cce) {
                throw new SerializationException("Can't convert key of class " +
record.key().getClass().getName() +
                        " to class " +
producerConfig.getClass(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG).getName() +
                        " specified in key.serializer", cce);
            }
            byte[] serializedValue;
            try {
                serializedValue = valueSerializer.serialize(record.topic(),
record.headers(), record.value());
            } catch (ClassCastException cce) {
                throw new SerializationException("Can't convert value of class " +
record.value().getClass().getName() +
                        " to class " +
producerConfig.getClass(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG).getName() +
                        " specified in value.serializer", cce);
            }
            // 获取该 record 要发送到的 partition
            int partition = partition(record, serializedKey, serializedValue,
cluster);
            tp = new TopicPartition(record.topic(), partition);

            // 给header设置只读
            setReadOnly(record.headers());
            Header[] headers = record.headers().toArray();

```

```
            int serializedSize = AbstractRecords.estimateSizeInBytesUpperBound(apiVersions.maxUsableProduceMagic(),
                    compressionType, serializedKey, serializedValue, headers);
            ensureValidRecordSize(serializedSize);
            long timestamp = record.timestamp() == null ? time.milliseconds() : record.timestamp();
            log.trace("Sending record {} with callback {} to topic {} partition {}", record, callback, record.topic(), partition);
            // producer callback will make sure to call both 'callback' and interceptor callback
            Callback interceptCallback = this.interceptors == null ? callback : new InterceptorCallback<>(callback, this.interceptors, tp);

            if (transactionManager != null && transactionManager.isTransactional())
                transactionManager.maybeAddPartitionToTransaction(tp);

            // 向 accumulator 中追加 record 数据，数据会先进行缓存
            RecordAccumulator.RecordAppendResult result = accumulator.append(tp, timestamp, serializedKey,
                    serializedValue, headers, interceptCallback, remainingWaitMs);

            // 如果追加完数据后，对应的 RecordBatch 已经达到了 batch.size 的大小（或者batch 的剩余空间不足以添加下一条 Record），则唤醒 sender 线程发送数据。
            if (result.batchIsFull || result.newBatchCreated) {
                log.trace("Waking up the sender since topic {} partition {} is either full or getting a new batch", record.topic(), partition);
                this.sender.wakeup();
            }
            return result.future;
            // handling exceptions and record the errors;
            // for API exceptions return them in the future,
            // for other exceptions throw directly
        } catch (ApiException e) {
            log.debug("Exception occurred during message send:", e);
            if (callback != null)
                callback.onCompletion(null, e);
            this.errors.record();
            if (this.interceptors != null)
                this.interceptors.onSendError(record, tp, e);
            return new FutureFailure(e);
        } catch (InterruptedException e) {
            this.errors.record();
            if (this.interceptors != null)
                this.interceptors.onSendError(record, tp, e);
            throw new InterruptException(e);
        } catch (BufferExhaustedException e) {
            this.errors.record();
            this.metrics.sensor("buffer-exhausted-records").record();
            if (this.interceptors != null)
                this.interceptors.onSendError(record, tp, e);
            throw e;
```

```
        } catch (KafkaException e) {
            this.errors.record();
            if (this.interceptors != null)
                this.interceptors.onSendError(record, tp, e);
            throw e;
        } catch (Exception e) {
            // we notify interceptor about all exceptions, since onSend is called before anything else in this method
            if (this.interceptors != null)
                this.interceptors.onSendError(record, tp, e);
            throw e;
        }
    }
```

1. Producer 通过 `waitOnMetadata()` 方法来获取对应 topic 的 metadata 信息，需要先该 topic 是可用的
2. Producer 端对 record 的 key 和 value 值进行序列化操作，在 Consumer 端再进行相应的反序列化
3. 获取partition值，具体分为下面三种情况：
    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
    4. Producer 默认使用的 partitioner 是 `org.apache.kafka.clients.producer.internals.DefaultPartitioner`
4. 向 accumulator 写数据，先将 record 写入到 buffer 中，当达到一个 batch.size 的大小时，再唤起 sender 线程去发送 RecordBatch，这里仔细分析一下Producer是如何向buffer写入数据的
    1. 获取该 topic-partition 对应的 queue，没有的话会创建一个空的 queue
    2. 向 queue 中追加数据，先获取 queue 中最新加入的那个 RecordBatch，如果不存在或者存在但剩余空余不足以添加本条 record 则返回 null，成功写入的话直接返回结果，写入成功
    3. 创建一个新的 RecordBatch，初始化内存大小根据 max(batch.size, Records.LOG_OVERHEAD + Record.recordSize(key, value)) 来确定（防止单条 record 过大的情况）
    4. 向新建的 RecordBatch 写入 record，并将 RecordBatch 添加到 queue 中，返回结果，写入成功
5. 发送 RecordBatch，当 record 写入成功后，如果发现 RecordBatch 已满足发送的条件（通常是 queue 中有多个 batch，那么最先添加的那些 batch 肯定是可以发送了），那么就会唤醒 sender 线程，发送 `RecordBatch`。sender 线程对`RecordBatch`的处理是在 run() 方法中进行的，该方法具体实现如下：
    1. 获取那些已经可以发送的 RecordBatch 对应的 nodes
    2. 如果与node 没有连接（如果可以连接,同时初始化该连接）,就证明该 node 暂时不能发送数据,暂时移除该 node
    3. 返回该 node 对应的所有可以发送的 RecordBatch 组成的 batches（key 是 node.id）,并将 RecordBatch 从对应的 queue 中移除
    4. 将由于元数据不可用而导致发送超时的 RecordBatch 移除
    5. 发送 RecordBatch

4.4.2.4 MetaData更新机制

1. metadata.requestUpdate() 将 metadata 的 needUpdate 变量设置为 true（强制更新），并返回当前的版本号（version），通过版本号来判断 metadata 是否完成更新
2. sender.wakeup() 唤醒 sender 线程，sender 线程又会去唤醒NetworkClient线程去更新
3. metadata.awaitUpdate(version, remainingWaitMs) 等待 metadata 的更新
4. 所以，每次 Producer 请求更新 metadata 时，会有以下几种情况:
    1. 如果 node 可以发送请求，则直接发送请求
    2. 如果该 node 正在建立连接，则直接返回
    3. 如果该 node 还没建立连接，则向 broker 初始化链接
5. NetworkClient的poll方法中判断是否需要更新meta数据，`handleCompletedReceives`处理`metadata`的更新，最终是调用的`DefaultMetadataUpdater`中的`handleCompletedMetadataResponse`方法处理

4.5 Kafka源码剖析之Consumer消费者流程

4.5.1 Consumer示例

KafkaConsumer

消费者的根本目的是从Kafka服务端拉取消息，并交给业务逻辑进行处理。

开发人员不必关心与Kafka服务端之间网络连接的管理、心跳检测、请求超时重试等底层操作

也**不必关心**订阅Topic的分区数量、分区Leader副本的网络拓扑以及消费组的Rebalance等细节，

另外还提供了自动提交offset的功能。

案例：

```
public static void main(String[] args) throws InterruptedException {
        // 是否自动提交
        Boolean autoCommit = false;
        // 是否异步提交
        Boolean isSync = true;

        Properties props = new Properties();
        // kafka地址,列表格式为host1:port1,host2:port2,…，无需添加所有的集群地址，kafka会根据提供的地址发现其他的地址（建议多提供几个，以防提供的服务器关闭）
        props.put("bootstrap.servers", "localhost:9092");
        // 消费组
        props.put("group.id", "test");
        // 开启自动提交offset
        props.put("enable.auto.commit", autoCommit.toString());
        // 1s自动提交
        props.put("auto.commit.interval.ms", "1000");
        // 消费者和群组协调器的最大心跳时间，如果超过该时间则认为该消费者已经死亡或者故障，需要踢出消费者组
        props.put("session.timeout.ms", "60000");
```

```
        // 一次poll间隔最大时间
        props.put("max.poll.interval.ms", "1000");
        // 当消费者读取偏移量无效的情况下，需要重置消费起始位置，默认为latest（从消费者启动后生成的记录），另外一个选项值是 earliest，将从有效的最小位移位置开始消费
        props.put("auto.offset.reset", "latest");
        // consumer端一次拉取数据的最大字节数
        props.put("fetch.max.bytes", "1024000");
        // key序列化方式
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        // value序列化方式
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

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

        String topic = "lagou_edu";
        // 订阅topic列表
        consumer.subscribe(Arrays.asList(topic));
        while (true) {
            // 消息拉取
            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());
            }
            if (!autoCommit) {
                if (isSync) {
                    // 处理完成单次消息以后，提交当前的offset，如果失败会一直重试直至成功
                    consumer.commitSync();
                } else {
                    // 异步提交
                    consumer.commitAsync((offsets, exception) -> {
                        exception.printStackTrace();
                        System.out.println(offsets.size());
                    });
                }
            }
            TimeUnit.SECONDS.sleep(3);
        }
    }
```

Kafka服务端并不会记录消费者的消费位置，而是由消费者自己决定如何保存如何记录其消费的offset。在Kafka服务端中添加了一个名为“__consumer_offsets"的内部topic来保存消费者提交的offset，当出现消费者上、下线时会触发Consumer Group进行Rebalance操作，对分区进行重新分配，待Rebalance操作完成后。消费者就可以读取该topic中记录的offset，并从此offset位置继续消费。当然，使用该topic记录消费者的offset只是默认选项，开发人员可以根据业务需求将offset记录在别的存储中。

在消费者消费消息的过程中，提交offset的时机非常重要，因为它决定了消费者故障重启后的消费位置。在上面的示例中，我们通过将`enable.auto.commit`选项设置为true可以起到自动提交offset的功能，`auto.commit.interval.ms`选项则设置了自动提交的时间间隔。每次在调用`KafkaConsumer.poll()`方法时都会检测是否需要自动提交，并提交上次`poll()`方法返回的最后一个消息的offset。为了避免消息丢失，建议poll()方法之前要处理完上次poll()方法拉取的全部消息。KafkaConsumer中还提供了两个手动提交offset的方法，分别是`commitSync()`和`commitAsync()`，它们都可以指定提交的offset值，区别在于前者是同步提交，后者是异步提交。

4.5.2 KafkaConsumer实例化

了解了`KafkaConsumer`的基本使用，开始深入了解`KafkaConsumer`原理和实现，先看一下构造方法核心逻辑

```
1  private KafkaConsumer(ConsumerConfig config,
2                        Deserializer<K> keyDeserializer,
3                        Deserializer<V> valueDeserializer) {
4          try {
5              // 获取client.id，如果为空则默认生成一个，默认：consumer-1
6              String clientId = config.getString(ConsumerConfig.CLIENT_ID_CONFIG);
7              if (clientId.isEmpty())
8                  clientId = "consumer-" +
   CONSUMER_CLIENT_ID_SEQUENCE.getAndIncrement();
9              this.clientId = clientId;
10             // 获取消费组名
11             String groupId = config.getString(ConsumerConfig.GROUP_ID_CONFIG);
12
13             LogContext logContext = new LogContext("[Consumer clientId=" + clientId
   + ", groupId=" + groupId + "] ");
14             this.log = logContext.logger(getClass());
15
16             log.debug("Initializing the Kafka consumer");
17             this.requestTimeoutMs =
   config.getInt(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG);
18             int sessionTimeOutMs =
   config.getInt(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG);
19             int fetchMaxWaitMs =
   config.getInt(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG);
20             if (this.requestTimeoutMs <= sessionTimeOutMs || this.requestTimeoutMs
   <= fetchMaxWaitMs)
21                 throw new ConfigException(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG +
   " should be greater than " + ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG + " and " +
   ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG);
22             this.time = Time.SYSTEM;
23
24             // 与生产者逻辑相同
25             Map<String, String> metricsTags = Collections.singletonMap("client-id",
   clientId);
26             MetricConfig metricConfig = new
   MetricConfig().samples(config.getInt(ConsumerConfig.METRICS_NUM_SAMPLES_CONFIG))
```

```
27
.timeWindow(config.getLong(ConsumerConfig.METRICS_SAMPLE_WINDOW_MS_CONFIG),
TimeUnit.MILLISECONDS)
28
.recordLevel(Sensor.RecordingLevel.forName(config.getString(ConsumerConfig.METRICS_R
ECORDING_LEVEL_CONFIG)))
29                      .tags(metricsTags);
30              List<MetricsReporter> reporters =
config.getConfiguredInstances(ConsumerConfig.METRIC_REPORTER_CLASSES_CONFIG,
31                      MetricsReporter.class);
32              reporters.add(new JmxReporter(JMX_PREFIX));
33              this.metrics = new Metrics(metricConfig, reporters, time);
34              this.retryBackoffMs =
config.getLong(ConsumerConfig.RETRY_BACKOFF_MS_CONFIG);
35
36              // 消费者拦截器
37              // load interceptors and make sure they get clientId
38              Map<String, Object> userProvidedConfigs = config.originals();
39              userProvidedConfigs.put(ConsumerConfig.CLIENT_ID_CONFIG, clientId);
40              List<ConsumerInterceptor<K, V>> interceptorList = (List) (new
ConsumerConfig(userProvidedConfigs,
false)).getConfiguredInstances(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
41                      ConsumerInterceptor.class);
42              this.interceptors = interceptorList.isEmpty() ? null : new
ConsumerInterceptors<>(interceptorList);
43
44              // key反序列化
45              if (keyDeserializer == null) {
46                  this.keyDeserializer =
config.getConfiguredInstance(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
47                          Deserializer.class);
48                  this.keyDeserializer.configure(config.originals(), true);
49              } else {
50                  config.ignore(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG);
51                  this.keyDeserializer = keyDeserializer;
52              }
53
54              // value反序列化
55              if (valueDeserializer == null) {
56                  this.valueDeserializer =
config.getConfiguredInstance(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
57                          Deserializer.class);
58                  this.valueDeserializer.configure(config.originals(), false);
59              } else {
60                  config.ignore(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG);
61                  this.valueDeserializer = valueDeserializer;
62              }
63              ClusterResourceListeners clusterResourceListeners =
configureClusterResourceListeners(keyDeserializer, valueDeserializer, reporters,
interceptorList);
```

```
            this.metadata = new Metadata(retryBackoffMs,
config.getLong(ConsumerConfig.METADATA_MAX_AGE_CONFIG),
                    true, false, clusterResourceListeners);
            List<InetSocketAddress> addresses =
ClientUtils.parseAndValidateAddresses(config.getList(ConsumerConfig.BOOTSTRAP_SERVER
S_CONFIG));
            this.metadata.update(Cluster.bootstrap(addresses), Collections.
<String>emptySet(), 0);
            String metricGrpPrefix = "consumer";
            ConsumerMetrics metricsRegistry = new
ConsumerMetrics(metricsTags.keySet(), "consumer");
            ChannelBuilder channelBuilder =
ClientUtils.createChannelBuilder(config);

            // 事务隔离级别
            IsolationLevel isolationLevel = IsolationLevel.valueOf(

 config.getString(ConsumerConfig.ISOLATION_LEVEL_CONFIG).toUpperCase(Locale.ROOT));
            Sensor throttleTimeSensor = Fetcher.throttleTimeSensor(metrics,
metricsRegistry.fetcherMetrics);

            // 网络组件
            NetworkClient netClient = new NetworkClient(
                    new
Selector(config.getLong(ConsumerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG), metrics,
time, metricGrpPrefix, channelBuilder, logContext),
                    this.metadata,
                    clientId,
                    100, // a fixed large enough value will suffice for max in-
flight requests
                    config.getLong(ConsumerConfig.RECONNECT_BACKOFF_MS_CONFIG),
                    config.getLong(ConsumerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG),
                    config.getInt(ConsumerConfig.SEND_BUFFER_CONFIG),
                    config.getInt(ConsumerConfig.RECEIVE_BUFFER_CONFIG),
                    config.getInt(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG),
                    time,
                    true,
                    new ApiVersions(),
                    throttleTimeSensor,
                    logContext);

            // 客户端
            this.client = new ConsumerNetworkClient(
                    logContext,
                    netClient,
                    metadata,
                    time,
                    retryBackoffMs,
                    config.getInt(ConsumerConfig.REQUEST_TIMEOUT_MS_CONFIG));

            // offset重置策略，默认是自动提交
```

```
            OffsetResetStrategy offsetResetStrategy =
OffsetResetStrategy.valueOf(config.getString(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG
).toUpperCase(Locale.ROOT));
            this.subscriptions = new SubscriptionState(offsetResetStrategy);
            this.assignors = config.getConfiguredInstances(
                    ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
                    PartitionAssignor.class);

            // offset协调者
            this.coordinator = new ConsumerCoordinator(logContext,
                    this.client,
                    groupId,
                    config.getInt(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG),
                    config.getInt(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG),
                    config.getInt(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG),
                    assignors,
                    this.metadata,
                    this.subscriptions,
                    metrics,
                    metricGrpPrefix,
                    this.time,
                    retryBackoffMs,
                    config.getBoolean(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG),
                    config.getInt(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG),
                    this.interceptors,

 config.getBoolean(ConsumerConfig.EXCLUDE_INTERNAL_TOPICS_CONFIG),
                    config.getBoolean(ConsumerConfig.LEAVE_GROUP_ON_CLOSE_CONFIG));

            // 拉取器
            this.fetcher = new Fetcher<>(
                    logContext,
                    this.client,
                    config.getInt(ConsumerConfig.FETCH_MIN_BYTES_CONFIG),
                    config.getInt(ConsumerConfig.FETCH_MAX_BYTES_CONFIG),
                    config.getInt(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG),
                    config.getInt(ConsumerConfig.MAX_PARTITION_FETCH_BYTES_CONFIG),
                    config.getInt(ConsumerConfig.MAX_POLL_RECORDS_CONFIG),
                    config.getBoolean(ConsumerConfig.CHECK_CRCS_CONFIG),
                    this.keyDeserializer,
                    this.valueDeserializer,
                    this.metadata,
                    this.subscriptions,
                    metrics,
                    metricsRegistry.fetcherMetrics,
                    this.time,
                    this.retryBackoffMs,
                    isolationLevel);

            // 打印用户设置，但是没有使用的配置项
            config.logUnused();
```

```
            AppInfoParser.registerAppInfo(JMX_PREFIX, clientId, metrics);

            log.debug("Kafka consumer initialized");
        } catch (Throwable t) {
            // call close methods if internal objects are already constructed
            // this is to prevent resource leak. see KAFKA-2121
            close(0, true);
            // now propagate the exception
            throw new KafkaException("Failed to construct kafka consumer", t);
        }
    }
```

1. 初始化参数配置
    1. client.id、group.id、消费者拦截器、key/value序列化、事务隔离级别
2. 初始化网络客户端`NetworkClient`
3. 初始化消费者网络客户端`ConsumerNetworkClient`
4. 初始化offset提交策略，默认自动提交
5. 初始化消费者协调器`ConsumerCoordinator`
6. 初始化拉取器`Fetcher`

4.5.3 订阅Topic

下面我们先来看一下subscribe方法都有哪些逻辑：

```
public void subscribe(Collection<String> topics, ConsumerRebalanceListener listener)
{
  // 轻量级锁
  acquireAndEnsureOpen();
  try {
    if (topics == null) {
      throw new IllegalArgumentException("Topic collection to subscribe to cannot be
null");
    } else if (topics.isEmpty()) {
      // topics为空，则开始取消订阅的逻辑
      this.unsubscribe();
    } else {
      // topic合法性判断,包含null或者空字符串直接抛异常
      for (String topic : topics) {
        if (topic == null || topic.trim().isEmpty())
          throw new IllegalArgumentException("Topic collection to subscribe to cannot
contain null or empty topic");
      }
      // 如果没有消费协调者直接抛异常
      throwIfNoAssignorsConfigured();
      log.debug("Subscribed to topic(s): {}", Utils.join(topics, ", "));
      // 开始订阅
      this.subscriptions.subscribe(new HashSet<>(topics), listener);
      // 更新元数据,如果metadata当前不包括所有的topics则标记强制更新
```

```
      metadata.setTopics(subscriptions.groupSubscription());
    }
  } finally {
    release();
  }
}

public void subscribe(Set<String> topics, ConsumerRebalanceListener listener) {
  if (listener == null)
    throw new IllegalArgumentException("RebalanceListener cannot be null");

  // 按照指定的Topic名字进行订阅，自动分配分区
  setSubscriptionType(SubscriptionType.AUTO_TOPICS);
  // 监听
  this.listener = listener;
  // 修改订阅信息
  changeSubscription(topics);
}

private void changeSubscription(Set<String> topicsToSubscribe) {
  if (!this.subscription.equals(topicsToSubscribe)) {
    // 如果使用AUTO_TOPICS或AUTO_PARTITION模式，则使用此集合记录所有订阅的Topic
    this.subscription = topicsToSubscribe;
    // Consumer Group中会选一个Leader,Leader会使用这个集合记录Consumer Group中所有消费者订阅的Topic,而其他的Follower的这个集合只会保存自身订阅的Topic
    this.groupSubscription.addAll(topicsToSubscribe);
  }
}
```

1. KafkaConsumer不是线程安全类，开启轻量级锁，topics为空抛异常，topics是空集合开始取消订阅，再次判断topics集合中是否有非法数据，判断消费者协调者是否为空。开始订阅对应topic。listener默认为`NoOpConsumerRebalanceListener`，一个空操作

   > 轻量级锁：分别记录了当前使用KafkaConsumer的线程id和重入次数，KafkaConsumer的acquire()和release()方法实现了一个"轻量级锁"，它并非真正的锁，仅是检测是否有多线程并发操作KafkaConsumer而已

2. 每一个KafkaConsumer实例内部都拥有一个SubscriptionState对象，subscribe内部调用了subscribe方法，subscribe方法订阅信息记录到`SubscriptionState`，多次订阅会覆盖旧数据。
3. 更新metadata，判断如果metadata中不包含当前groupSubscription，开始标记更新（后面会有更新的逻辑），并且消费者侧的topic不会过期

4.5.4 消息消费过程

下面KafkaConsumer的核心方法poll是如何拉取消息的，先来看一下下面的代码：

4.5.4.1 poll

```
public ConsumerRecords<K, V> poll(long timeout) {
        // 使用轻量级锁检测kafkaConsumer是否被其他线程使用
        acquireAndEnsureOpen();
        try {
            // 超时时间小于0抛异常
            if (timeout < 0)
                throw new IllegalArgumentException("Timeout must not be negative");

            // 订阅类型为NONE抛异常,表示当前消费者没有订阅任何topic或者没有分配分区
            if (this.subscriptions.hasNoSubscriptionOrUserAssignment())
                throw new IllegalStateException("Consumer is not subscribed to any
topics or assigned any partitions");

            // poll for new data until the timeout expires
            long start = time.milliseconds();
            long remaining = timeout;
            do {
                // 核心方法，拉取消息
                Map<TopicPartition, List<ConsumerRecord<K, V>>> records =
pollOnce(remaining);
                if (!records.isEmpty()) {
                    // before returning the fetched records, we can send off the next
round of fetches
                    // and avoid block waiting for their responses to enable
pipelining while the user
                    // is handling the fetched records.
                    //
                    // NOTE: since the consumed position has already been updated, we
must not allow
                    // wakeups or any other errors to be triggered prior to returning
the fetched records.
                    // 如果拉取到了消息，发送一次消息拉取的请求，不会阻塞不会被中断
                    // 在返回数据之前，发送下次的 fetch 请求，避免用户在下次获取数据时线程
block
                    if (fetcher.sendFetches() > 0 || client.hasPendingRequests())
                        client.pollNoWakeup();

                    // 经过拦截器处理后返回
                    if (this.interceptors == null)
                        return new ConsumerRecords<>(records);
                    else
                        return this.interceptors.onConsume(new ConsumerRecords<>
(records));
                }

                long elapsed = time.milliseconds() - start;
                // 拉取超时就结束
                remaining = timeout - elapsed;
            } while (remaining > 0);
```

```
            return ConsumerRecords.empty();
        } finally {
            release();
        }
    }
```

1. 使用轻量级锁检测kafkaConsumer是否被其他线程使用
2. 检查超时时间是否小于0，小于0抛出异常，停止消费
3. 检查这个 consumer 是否订阅的相应的 topic-partition
4. 调用 pollOnce() 方法获取相应的 records
5. 在返回获取的 records 前，发送下一次的 fetch 请求，避免用户在下次请求时线程 block 在 pollOnce() 方法中
6. 如果在给定的时间（timeout）内获取不到可用的 records，返回空数据

这里可以看出，poll 方法的真正实现是在 pollOnce 方法中，poll 方法通过 pollOnce 方法获取可用的数据

4.5.4.2 pollOnce

```
// 除了获取新数据外，还会做一些必要的 offset-commit和reset-offset的操作
private Map<TopicPartition, List<ConsumerRecord<K, V>>> pollOnce(long timeout) {
        client.maybeTriggerWakeup();

        // 1. 获取 GroupCoordinator 地址并连接、加入 Group、sync Group、自动 commit, join 及 sync 期间 group 会进行 rebalance
        coordinator.poll(time.milliseconds(), timeout);

        // 2. 更新订阅的 topic-partition 的 offset（如果订阅的 topic-partition list 没有有效的 offset 的情况下）
        if (!subscriptions.hasAllFetchPositions())
            updateFetchPositions(this.subscriptions.missingFetchPositions());

        // 3. 获取 fetcher 已经拉取到的数据
        Map<TopicPartition, List<ConsumerRecord<K, V>>> records = fetcher.fetchedRecords();
        if (!records.isEmpty())
            return records;

        // 4. 发送 fetch 请求,会从多个 topic-partition 拉取数据（只要对应的 topic-partition 没有未完成的请求）
        fetcher.sendFetches();

        long now = time.milliseconds();
        long pollTimeout = Math.min(coordinator.timeToNextPoll(now), timeout);

        // 5. 调用 poll 方法发送请求（底层发送请求的接口）
        client.poll(pollTimeout, now, new PollCondition() {
            @Override
            public boolean shouldBlock() {
                // since a fetch might be completed by the background thread, we need this poll condition
```

```
                // to ensure that we do not block unnecessarily in poll()
                return !fetcher.hasCompletedFetches();
            }
        });

        // 6. 如果 group 需要 rebalance,直接返回空数据,这样更快地让 group 进行稳定状态
        if (coordinator.needRejoin())
            return Collections.emptyMap();

        // 获取到请求的结果
        return fetcher.fetchedRecords();
    }
```

pollOnce 可以简单分为6步来看，其作用分别如下:

4.5.4.2.1 coordinator.poll()

获取 GroupCoordinator 的地址，并建立相应 tcp 连接，发送 join-group、sync-group，之后才真正加入到了一个 group 中，这时会获取其要消费的 topic-partition 列表，如果设置了自动 commit，也会在这一步进行 commit。总之，对于一个新建的 group，group 状态将会从 Empty –> PreparingRebalance –> AwaiSync –> Stable；

1. 获取 GroupCoordinator 的地址，并建立相应 tcp 连接；
2. 发送 join-group 请求，然后 group 将会进行 rebalance；
3. 发送 sync-group 请求，之后才正在加入到了一个 group 中，这时会通过请求获取其要消费的 topic-partition 列表；
4. 如果设置了自动 commit，也会在这一步进行 commit offset

4.5.4.2.2 updateFetchPositions()

这个方法主要是用来更新这个 consumer 实例订阅的 topic-partition 列表的 fetch-offset 信息。目的就是为了获取其订阅的每个 topic-partition 对应的 position，这样 Fetcher 才知道从哪个 offset 开始去拉取这个 topic-partition 的数据

```
private void updateFetchPositions(Set<TopicPartition> partitions) {
        // 先重置那些调用 seekToBegin 和 seekToEnd 的 offset 的 tp,设置其  the fetch position 的 offset
        fetcher.resetOffsetsIfNeeded(partitions);

        if (!subscriptions.hasAllFetchPositions(partitions)) {
            // 获取所有分配 tp 的 offset, 即 committed offset, 更新到 TopicPartitionState 中的 committed offset 中
            coordinator.refreshCommittedOffsetsIfNeeded();

            // 如果 the fetch position 值无效,则将上步获取的 committed offset 设置为 the fetch position
            fetcher.updateFetchPositions(partitions);
        }
    }
```

在 Fetcher 中，这个 consumer 实例订阅的每个 topic-partition 都会有一个对应的 TopicPartitionState 对象，在这个对象中会记录以下这些内容：

```
private static class TopicPartitionState {
        // Fetcher 下次去拉取时的 offset，Fecher 在拉取时需要知道这个值
        private Long position; // last consumed position
        // 最后一次获取的高水位标记
        private Long highWatermark; // the high watermark from last fetch
        private Long lastStableOffset;
        // consumer 已经处理完的最新一条消息的 offset，consumer 主动调用 offset-commit 时会更新这个值；
        private OffsetAndMetadata committed;  // last committed position
        // 是否暂停
        private boolean paused;  // whether this partition has been paused by the user
        // 这 topic-partition offset 重置的策略，重置之后，这个策略就会改为 null，防止再次操作
        private OffsetResetStrategy resetStrategy;  // the strategy to use if the offset needs resetting
}
```

4.5.4.2.3 fetcher.fetchedRecords()

返回其 fetched records，并更新其 fetch-position offset，只有在 offset-commit 时（自动 commit 时，是在第一步实现的），才会更新其 committed offset；

```
public Map<TopicPartition, List<ConsumerRecord<K, V>>> fetchedRecords() {
        Map<TopicPartition, List<ConsumerRecord<K, V>>> fetched = new HashMap<>();
        // 在 max.poll.records 中设置单词最大的拉取条数
        int recordsRemaining = maxPollRecords;

        try {
            while (recordsRemaining > 0) {
                if (nextInLineRecords == null || nextInLineRecords.isFetched) {
                    // 从队列中获取但不移除此队列的头；如果此队列为空，返回null
                    CompletedFetch completedFetch = completedFetches.peek();
                    if (completedFetch == null) break;

                    // 获取下一个要处理的 nextInLineRecords
                    nextInLineRecords = parseCompletedFetch(completedFetch);
                    completedFetches.poll();
                } else {
                    // 拉取records,更新 position
                    List<ConsumerRecord<K, V>> records = fetchRecords(nextInLineRecords, recordsRemaining);
                    TopicPartition partition = nextInLineRecords.partition;
                    if (!records.isEmpty()) {
                        List<ConsumerRecord<K, V>> currentRecords = fetched.get(partition);
                        if (currentRecords == null) {
                            fetched.put(partition, records);
                        } else {
                            List<ConsumerRecord<K, V>> newRecords = new ArrayList<>(records.size() + currentRecords.size());
```

```
                            newRecords.addAll(currentRecords);
                            newRecords.addAll(records);
                            fetched.put(partition, newRecords);
                        }
                        recordsRemaining -= records.size();
                    }
                }
            }
        } catch (KafkaException e) {
            if (fetched.isEmpty())
                throw e;
        }
        return fetched;
    }

    private List<ConsumerRecord<K, V>> fetchRecords(PartitionRecords
partitionRecords, int maxRecords) {
        if (!subscriptions.isAssigned(partitionRecords.partition)) {
            log.debug("Not returning fetched records for partition {} since it is no
longer assigned",
                    partitionRecords.partition);
        } else {
            long position = subscriptions.position(partitionRecords.partition);
            // 这个 tp 不能来消费了,比如调用 pause方法暂停消费
            if (!subscriptions.isFetchable(partitionRecords.partition)) {
                log.debug("Not returning fetched records for assigned partition {}
since it is no longer fetchable",
                        partitionRecords.partition);
            } else if (partitionRecords.nextFetchOffset == position) {
                // 获取该 tp 对应的records,并更新 partitionRecords 的 fetchOffset (用于判
断是否顺序)
                List<ConsumerRecord<K, V>> partRecords =
partitionRecords.fetchRecords(maxRecords);

                long nextOffset = partitionRecords.nextFetchOffset;
                log.trace("Returning fetched records at offset {} for assigned
partition {} and update " +
                        "position to {}", position, partitionRecords.partition,
nextOffset);
                // 更新消费的到 offset ( the fetch position)
                subscriptions.position(partitionRecords.partition, nextOffset);

                // 获取 Lag (即 position与 hw 之间差值) ,hw 为 null 时,才返回 null
                Long partitionLag =
subscriptions.partitionLag(partitionRecords.partition, isolationLevel);
                if (partitionLag != null)
                    this.sensors.recordPartitionLag(partitionRecords.partition,
partitionLag);

                return partRecords;
            } else {
```

```
                log.debug("Ignoring fetched records for {} at offset {} since the
current position is {}",
                        partitionRecords.partition, partitionRecords.nextFetchOffset,
position);
            }
        }

        partitionRecords.drain();
        return emptyList();
    }
```

4.5.4.2.4 fetcher.sendFetches()

只要订阅的 topic-partition list 没有未处理的 fetch 请求，就发送对这个 topic-partition 的 fetch 请求，在真正发送时，还是会按 node 级别去发送，leader 是同一个 node 的 topic-partition 会合成一个请求去发送；

```
// 向订阅的所有 partition （只要该 leader 暂时没有拉取请求）所在 leader 发送 fetch 请求
    public int sendFetches() {
        // 1. 创建 Fetch Request
        Map<Node, FetchRequest.Builder> fetchRequestMap = createFetchRequests();
        for (Map.Entry<Node, FetchRequest.Builder> fetchEntry :
fetchRequestMap.entrySet()) {
            final FetchRequest.Builder request = fetchEntry.getValue();
            final Node fetchTarget = fetchEntry.getKey();

            log.debug("Sending {} fetch for partitions {} to broker {}",
isolationLevel, request.fetchData().keySet(),
                    fetchTarget);
            // 2 发送 Fetch Request
            client.send(fetchTarget, request)
                    .addListener(new RequestFutureListener<ClientResponse>() {
                        @Override
                        public void onSuccess(ClientResponse resp) {
                            FetchResponse response = (FetchResponse)
resp.responseBody();
                            if (!matchesRequestedPartitions(request, response)) {
                                log.warn("Ignoring fetch response containing
partitions {} since it does not match " +
                                                "the requested partitions {}",
response.responseData().keySet(),
                                                request.fetchData().keySet());
                                return;
                            }

                            Set<TopicPartition> partitions = new HashSet<>
(response.responseData().keySet());
                            FetchResponseMetricAggregator metricAggregator = new
FetchResponseMetricAggregator(sensors, partitions);

                            for (Map.Entry<TopicPartition,
FetchResponse.PartitionData> entry : response.responseData().entrySet()) {
```

```
                                TopicPartition partition = entry.getKey();
                                long fetchOffset =
request.fetchData().get(partition).fetchOffset;
                                FetchResponse.PartitionData fetchData =
entry.getValue();

                                log.debug("Fetch {} at offset {} for partition {}
returned fetch data {}",
                                        isolationLevel, fetchOffset, partition,
fetchData);
                                completedFetches.add(new CompletedFetch(partition,
fetchOffset, fetchData, metricAggregator,
                                        resp.requestHeader().apiVersion()));
                            }

                            sensors.fetchLatency.record(resp.requestLatencyMs());
                        }

                        @Override
                        public void onFailure(RuntimeException e) {
                            log.debug("Fetch request {} to {} failed",
request.fetchData(), fetchTarget, e);
                        }
                    });
        }
        return fetchRequestMap.size();
    }
```

1. createFetchRequests()：为订阅的所有 topic-partition list 创建 fetch 请求（只要该topic-partition 没有还在处理的请求），创建的 fetch 请求依然是按照 node 级别创建的；
2. client.send()：发送 fetch 请求，并设置相应的 Listener，请求处理成功的话，就加入到 completedFetches 中，在加入这个 completedFetches 集合时，是按照 topic-partition 级别去加入，这样也就方便了后续的处理。

从这里可以看出，在每次发送 fetch 请求时，都会向所有可发送的 topic-partition 发送 fetch 请求，调用一次 fetcher.sendFetches，拉取到的数据，可需要多次 pollOnce 循环才能处理完，因为 Fetcher 线程是在后台运行，这也保证了尽可能少地阻塞用户的处理线程，因为如果 Fetcher 中没有可处理的数据，用户的线程是会阻塞在 poll 方法中的

4.5.4.2.5 client.poll()

调用底层 NetworkClient 提供的接口去发送相应的请求；

4.5.4.2.6 coordinator.needRejoin()

如果当前实例分配的 topic-partition 列表发送了变化，那么这个 consumer group 就需要进行 rebalance

4.5.5 自动提交

最简单的提交方式是让悄费者自动提交偏移量。如果enable.auto.commit被设为 true，消费者会自动把从 poll() 方法接收到的最大偏移量提交上去。提交时间间隔由 auto.commit.interval.ms 控制，默认值是 5s。与消费者里的其他东西 一样，自动提交也是在轮询(poll())里进行的。消费者每次在进行轮询时会检查是否该提交偏移量了，如果是，那 么就会提交从上一次轮询返回的偏移量。

不过，这种简便的方式也会带来一些问题，来看一下下面的例子：
假设我们仍然使用默认的 5s提交时间间隔，在最近一次提交之后的 3s发生了再均衡，再 均衡之后，消费者从最后一次提交的偏移量位置开始读取消息。这个时候偏移量已经落后 了 3s，所以在这 3s 内到达的消息会被重复处理。可以通过修改提交时间间隔来更频繁地提交偏移量，减小可能出现重复消息的时间窗，不过这种情况是无也完全避免的

4.5.6 手动提交

4.5.6.1 同步提交

取消自动提交，把 auto.commit.offset 设为 false，让应用程序决定何时提交 偏 移量。使用 commitSync() 提交偏移量最简单也最可靠。这个 API会提交由 poll() 方法返回 的最新偏移量，提交成 功后马上返回，如果提交失败就抛出异常

```
while (true) {
  // 消息拉取
  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());
  }
  // 处理完成单次消息以后，提交当前的offset，如果提交失败就抛出异常
  consumer.commitSync();
}
```

4.5.6.2 异步提交

同步提交有一个不足之处，在 broker对提交请求作出回应之前，应用程序会一直阻塞，这样会限制应用程序的吞吐量。我们可以通过降低提交频率来提升吞吐量，但如果发生了再均衡， 会增加重复消息的数量。这个时候可以使用异步提交 API。我们只管发送提交请求，无需等待 broker的响应。

```java
while (true) {
  // 消息拉取
  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());
  }
  // 异步提交
  consumer.commitAsync((offsets, exception) -> {
    exception.printStackTrace();
    System.out.println(offsets.size());
  });
}
```

4.6 Kafka源码剖析之消息存储机制

log.dirs/<topic-name>-<partitionno>/{.index，.timeindex，.log}

首先查看Kafka如何处理生产的消息：

Log.scala × KafkaApis.scala ×

```scala
  */
def handle(request: RequestChannel.Request) {
  try {
    trace( msg = s"Handling request:${request.requestDesc( details = true)} f
      s"securityProtocol:${request.context.securityProtocol},principal:$
    request.header.apiKey match {
      // 消息生产请求的处理逻辑
      case ApiKeys.PRODUCE => handleProduceRequest(request)
      // 处理
```

调用副本管理器，将记录追加到分区的副本中。

Log.scala × KafkaApis.scala ×

```scala

    // 调用副本管理器将消息追加到副本中
    replicaManager.appendRecords(
      timeout = produceRequest.timeout.toLong, // 请求的超时时间
      requiredAcks = produceRequest.acks,      // acks
      internalTopicsAllowed = internalTopicsAllowed, // 如果是管理客户端，允许操作内部主题
      isFromClient = true, // 请求来源是客户端
      entriesPerPartition = authorizedRequestInfo, // 请求的授权信息
      responseCallback = sendResponseCallback,
      processingStatsCallback = processingStatsCallback)
```

将数据追加到本地的Log日志中：

```
Log.scala  KafkaApis.scala  ReplicaManager.scala
 * @param processingStatsCallback 统计指标信息的回调。
 */
def appendRecords(timeout: Long,
                  requiredAcks: Short,
                  internalTopicsAllowed: Boolean,
                  isFromClient: Boolean,
                  entriesPerPartition: Map[TopicPartition, MemoryRecords],
                  responseCallback: Map[TopicPartition, PartitionResponse] => Unit,
                  delayedProduceLock: Option[Lock] = None,
                  processingStatsCallback: Map[TopicPartition, RecordsProcessingStats] => Unit = _ => ()) {
  // 判断acks设置正确与否
  if (isValidRequiredAcks(requiredAcks)) {
    val sTime = time.milliseconds
    // 追加消息到日志，并等待返回结果
    val localProduceResults = appendToLocalLog(internalTopicsAllowed = internalTopicsAllowed,
      isFromClient = isFromClient, entriesPerPartition, requiredAcks)
    debug("Produce to local log in %d ms".format(time.milliseconds - sTime))
```

追加消息的实现：

```
Log.scala  KafkaApis.scala  ReplicaManager.scala  MemoryRecords.java
 * 将消息追加到本地日志副本
 * @param internalTopicsAllowed 是否允许操作内部主题
 * @param isFromClient  是否来自客户端
 * @param entriesPerPartition 每个分区的条目
 * @param requiredAcks acks
 * @return
 */
private def appendToLocalLog(internalTopicsAllowed: Boolean,
                             isFromClient: Boolean,
                             entriesPerPartition: Map[TopicPartition, MemoryRecords],
                             requiredAcks: Short): Map[TopicPartition, LogAppendResult] = {
  trace( msg = s"Append [$entriesPerPartition] to local log")
  // 遍历要追加的每个主题分区的记录
```

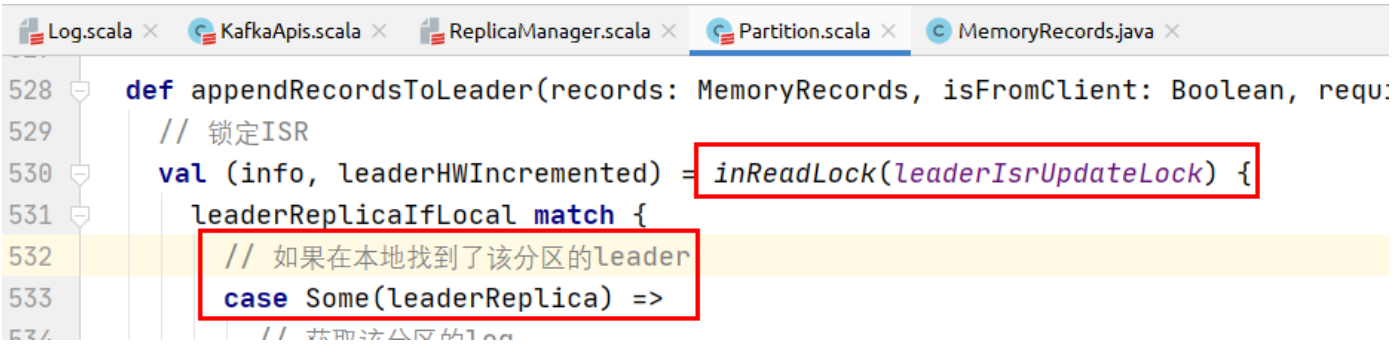
遍历需要追加的每个主题分区的消息：

```
Log.scala  KafkaApis.scala  ReplicaManager.scala  MemoryRecords.java
736     trace( msg = s"Append [$entriesPerPartition] to local log")
737     // 遍历要追加的每个主题分区的记录
738     entriesPerPartition.map {
739       case (topicPartition, records) =>
```

调用partition的方法将记录追加到该分区的leader分区中：

```
Log.scala  KafkaApis.scala  ReplicaManager.scala  MemoryRecords.java
755             if (partition eq ReplicaManager.OfflinePartition)
756               throw new KafkaStorageException(s"Partition $topicPartition is in an of
757             // 调用partition的方法将记录追加到该分区的leader分区中
758             partition.appendRecordsToLeader(records, isFromClient, requiredAcks)
759           // 如果没有找到目标分区
```

如果在本地找到了该分区的leader：



执行下述逻辑将消息追加到leader分区：

```
// 获取该分区的log
val log = leaderReplica.log.get
// 获取最小ISR副本数
val minIsr = log.config.minInSyncReplicas
// 计算同步副本的个数
val inSyncSize = inSyncReplicas.size

// 如果同步副本的个数小于要求的最小副本数，并且acks设置的是-1，则不追加消息
if (inSyncSize < minIsr && requiredAcks == -1) {
  throw new NotEnoughReplicasException("Number of insync replicas for partition %s is [%d], below required minimum [%d]"
    .format(topicPartition, inSyncSize, minIsr))
}
// 追加消息到leader
val info = log.appendAsLeader(records, leaderEpoch = this.leaderEpoch, isFromClient)
// 尝试锁定follower获取消息的请求，因为此时leader正在更新LEO。
replicaManager.tryCompleteDelayedFetch(TopicPartitionOperationKey(this.topic, this.partitionId))
// 如果ISR只有一个元素的话，需要HW+1
(info, maybeIncrementLeaderHW(leaderReplica))
```

`log.appendAsLeader(records, leaderEpoch = this.leaderEpoch, isFromClient)`的实现：

具体代码实现：

```
/**
     * 验证如下信息：
     * 每条消息与其CRC是否匹配
     * 每条消息的字节数是否匹配
     * 传入记录批的序列号与现有状态以及彼此之间是否一致。
     * 同时计算如下值：
     * 消息批中第一个偏移量
     * 消息批中最后一个偏移量
     * 消息个数
     * 正确字节的个数
     * 偏移量是否单调递增
     * 是否使用了压缩编码解码器（如果使用了压缩编解码器，则给出最后一个）
     */
  val appendInfo = analyzeAndValidateRecords(records, isFromClient = isFromClient)

  // 如果没有消息需要追加或该消息集合与上一个消息集合重复，则返回
  if (appendInfo.shallowCount == 0)
    return appendInfo

  // 在向磁盘日志追加之前剔除不正确的字节或剔除不完整的消息
  var validRecords = trimInvalidBytes(records, appendInfo)

  // 消息集合剩余的正确部分，插入到日志中
  lock synchronized {
    // 检查日志的MMap是否关闭了，如果关闭无法进行写操作，抛异常
    checkIfMemoryMappedBufferClosed()
    if (assignOffsets) {
      // 如果需要给消息添加偏移量
      val offset = new LongRef(nextOffsetMetadata.messageOffset)
```

```
      appendInfo.firstOffset = offset.value
      val now = time.milliseconds
      val validateAndOffsetAssignResult = try {
        // 校验消息和赋值给消息的偏移量是否正确无误
        LogValidator.validateMessagesAndAssignOffsets(validRecords,
          offset,
          time,
          now,
          appendInfo.sourceCodec,
          appendInfo.targetCodec,
          config.compact,
          config.messageFormatVersion.messageFormatVersion.value,
          config.messageTimestampType,
          config.messageTimestampDifferenceMaxMs,
          leaderEpoch,
          isFromClient)
      } catch {
        case e: IOException => throw new KafkaException("Error in validating
messages while appending to log '%s'".format(name), e)
      }
      // 正确的消息集合，此时处于内存中
      validRecords = validateAndOffsetAssignResult.validatedRecords
      // 要追加消息的最大时间戳
      appendInfo.maxTimestamp = validateAndOffsetAssignResult.maxTimestamp
      // 要追加的消息的最大时间戳对应的偏移量
      appendInfo.offsetOfMaxTimestamp =
validateAndOffsetAssignResult.shallowOffsetOfMaxTimestamp
      // 最后一个偏移量是偏移量的值-1
      appendInfo.lastOffset = offset.value - 1
      appendInfo.recordsProcessingStats =
validateAndOffsetAssignResult.recordsProcessingStats
      if (config.messageTimestampType == TimestampType.LOG_APPEND_TIME)
        // 如果消息时间戳的类型是日志追加的时间，则需要赋值当前系统时间
        appendInfo.logAppendTime = now
      // 需要重新验证消息的大小，以防消息发生改变，如重新压缩或者转换了消息格式
      if (validateAndOffsetAssignResult.messageSizeMaybeChanged) {
        for (batch <- validRecords.batches.asScala) {
          // 如果消息集合的字节数大于配置的消息最大字节数，抛异常
          if (batch.sizeInBytes > config.maxMessageSize) {
            // we record the original message set size instead of the trimmed size
            // to be consistent with pre-compression bytesRejectedRate recording

 brokerTopicStats.topicStats(topicPartition.topic).bytesRejectedRate.mark(records.si
zeInBytes)

 brokerTopicStats.allTopicsStats.bytesRejectedRate.mark(records.sizeInBytes)
            throw new RecordTooLargeException("Message batch size is %d bytes which
exceeds the maximum configured size of %d."
              .format(batch.sizeInBytes, config.maxMessageSize))
          }
        }
```

```
      }
    } else {
      // 如果不需要分配消息偏移量，则使用给定的消息偏移量
      if (!appendInfo.offsetsMonotonic)
        // 如果偏移量不是单调递增的，抛异常
        throw new OffsetsOutOfOrderException(s"Out of order offsets found in append to $topicPartition: " +
                                             records.records.asScala.map(_.offset))
      // 如果消息批的第一个偏移量小于分区leader日志中下一条记录的偏移量，抛异常。
      if (appendInfo.firstOffset < nextOffsetMetadata.messageOffset) {
        // we may still be able to recover if the log is empty
        // one example: fetching from log start offset on the leader which is not batch aligned,
        // which may happen as a result of AdminClient#deleteRecords()
        // appendInfo.firstOffset maybe either first offset or last offset of the first batch.
        // get the actual first offset, which may require decompressing the data
        val firstOffset = records.batches.asScala.head.baseOffset()
        throw new UnexpectedAppendOffsetException(
          s"Unexpected offset in append to $topicPartition. First offset or last offset of the first batch " +
          s"${appendInfo.firstOffset} is less than the next offset ${nextOffsetMetadata.messageOffset}. " +
          s"First 10 offsets in append: ${records.records.asScala.take(10).map(_.offset)}, last offset in" +
          s" append: ${appendInfo.lastOffset}. Log start offset = $logStartOffset",
          firstOffset, appendInfo.lastOffset)
      }
    }

    // 使用leader给消息赋值的epoch值更新缓存的epoch值。
    validRecords.batches.asScala.foreach { batch =>
      if (batch.magic >= RecordBatch.MAGIC_VALUE_V2)
        // 需要在epoch中记录leader的epoch值和消息集合的起始偏移量
        leaderEpochCache.assign(batch.partitionLeaderEpoch, batch.baseOffset)
    }
    // 检查消息批的字节大小是否大于日志分段的最大值，如果是，则抛异常
    if (validRecords.sizeInBytes > config.segmentSize) {
      throw new RecordBatchTooLargeException("Message batch size is %d bytes which exceeds the maximum configured segment size of %d."
        .format(validRecords.sizeInBytes, config.segmentSize))
    }
    // 消息批的消息都正确，偏移量也都赋值了，时间戳也更新了
    // 此时需要验证生产者的幂等性/事务状态，并收集一些元数据
    val (updatedProducers, completedTxns, maybeDuplicate) = analyzeAndValidateProducerState(validRecords, isFromClient)
    // 如果是重复的消息批，则直接返回被重复的消息批的appendInfo
    maybeDuplicate.foreach { duplicate =>
      appendInfo.firstOffset = duplicate.firstOffset
      appendInfo.lastOffset = duplicate.lastOffset
      appendInfo.logAppendTime = duplicate.timestamp
```

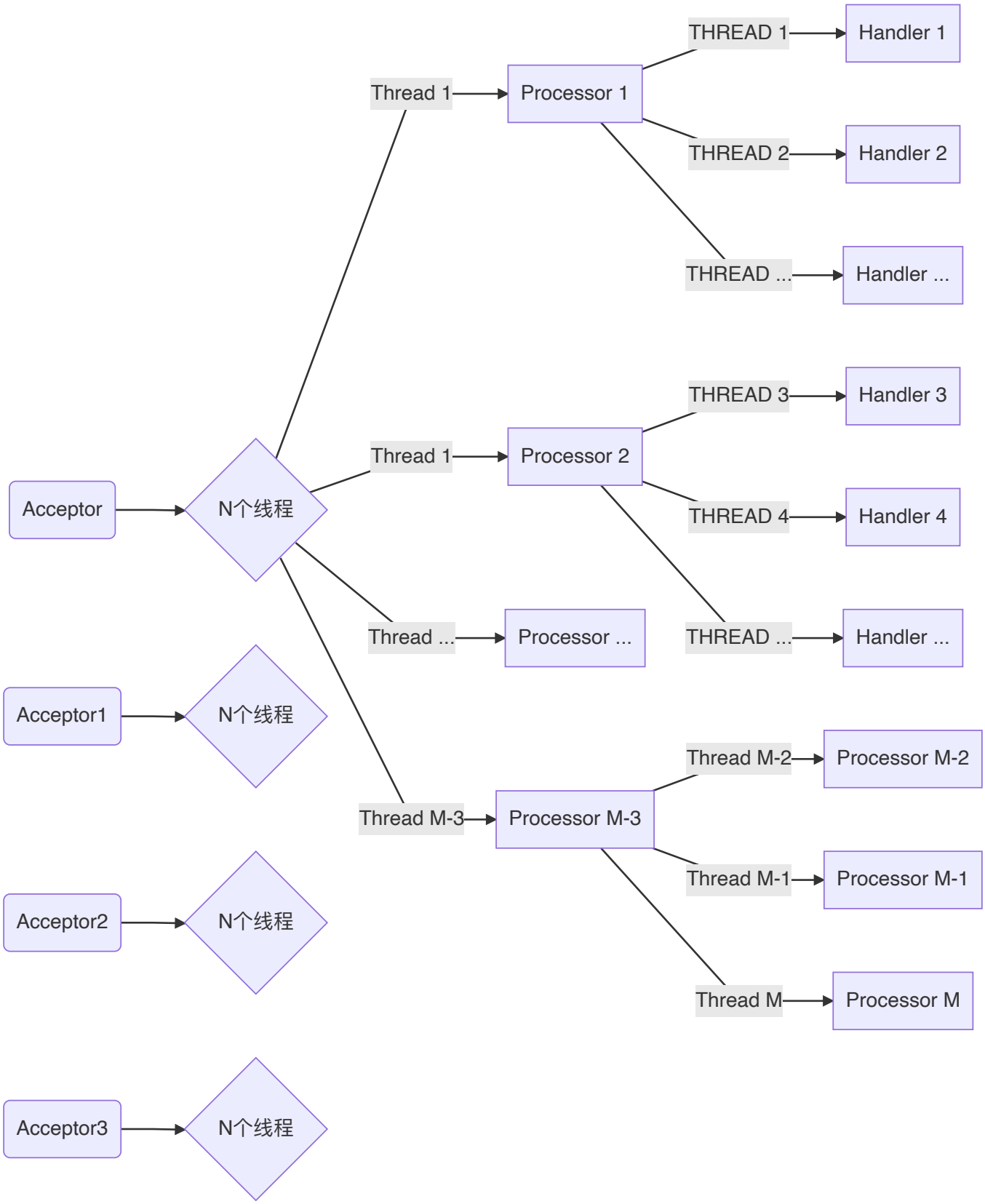
```
117       appendInfo.logStartOffset = logStartOffset
118       return appendInfo
119     }
120
121     // 如果当前日志分段写满了，则滚动日志分段
122     val segment = maybeRoll(messagesSize = validRecords.sizeInBytes,
123       maxTimestampInMessages = appendInfo.maxTimestamp,
124       maxOffsetInMessages = appendInfo.lastOffset)
125
126     val logOffsetMetadata = LogOffsetMetadata(
127       messageOffset = appendInfo.firstOffset,
128       segmentBaseOffset = segment.baseOffset,
129       relativePositionInSegment = segment.size)
130
131     // 日志片段中追加消息
132     segment.append(firstOffset = appendInfo.firstOffset,
133       largestOffset = appendInfo.lastOffset,
134       largestTimestamp = appendInfo.maxTimestamp,
135       shallowOffsetOfMaxTimestamp = appendInfo.offsetOfMaxTimestamp,
136       records = validRecords)
137     // 更新生产者状态
138     for ((producerId, producerAppendInfo) <- updatedProducers) {
139       producerAppendInfo.maybeCacheTxnFirstOffsetMetadata(logOffsetMetadata)
140       producerStateManager.update(producerAppendInfo)
141     }
142
143     // update the transaction index with the true last stable offset. The last
    offset visible
144     // to consumers using READ_COMMITTED will be limited by this value and the high
    watermark.
145     for (completedTxn <- completedTxns) {
146       val lastStableOffset = producerStateManager.completeTxn(completedTxn)
147       segment.updateTxnIndex(completedTxn, lastStableOffset)
148     }
149
150     // always update the last producer id map offset so that the snapshot reflects
    the current offset
151     // even if there isn't any idempotent data being written
152     producerStateManager.updateMapEndOffset(appendInfo.lastOffset + 1)
153     // leader的LEO+1
154     updateLogEndOffset(appendInfo.lastOffset + 1)
155
156     // update the first unstable offset (which is used to compute LSO)
157     updateFirstUnstableOffset()
158
159     trace(s"Appended message set to log with last offset ${appendInfo.lastOffset} "
    +
160       s"first offset: ${appendInfo.firstOffset}, " +
161       s"next offset: ${nextOffsetMetadata.messageOffset}, " +
162       s"and messages: $validRecords")
163
```

```
164     // 如果未刷盘的消息个数大于配置的消息个数，刷盘
165     if (unflushedMessages >= config.flushInterval)
166       // 刷盘
167       flush()
168 
169     appendInfo
170   }
171 }
```

4.7 Kafka源码剖析之SocketServer

线程模型：

1. 当前broker上配置了多少个listener，就有多少个Acceptor，用于新建连接。
2. 每个Acceptor对应N个线程的处理器（Processor），用于接收客户端请求。
3. 处理器们对应M个线程的处理程序（Handler），处理用户请求，并将响应发送给等待给客户写响应的处理器线程。



在启动KakfaServer的startup方法中启动SocketServer：

KafkaServer.scala | RequestChannel.scala | RequestOrResponseSend.scala | SocketServer.scala

```
251
252         // 创建并启动SocketServer的接收线程，并绑定到指定的端口。
253         // 在初始化序列的最后，才启动处理程序（延迟启动）
254         // 保证在处理认证的时候，证书已经加载了。
255         socketServer = new SocketServer(config, metrics, time, credentialProvider)
256         // 启动SocketServer
257         socketServer.startup(startupProcessors = false)
258
```

每个listener就是一个端点，每个端点创建多个处理程序。

KafkaServer.scala | RequestChannel.scala | RequestOrResponseSend.scala | SocketServer.scala | Gauge.class

```
114
115     var processorBeginIndex = 0
116     // 配置的一堆listener
117     // 对每个listener启动多个处理程序processor
118     config.listeners.foreach { endpoint =>
119       val listenerName = endpoint.listenerName
120       val securityProtocol = endpoint.securityProtocol
121       val processorEndIndex = processorBeginIndex + numProcessorThreads
122
123       for (i <- processorBeginIndex until processorEndIndex)
124         //
125         processors(i) = newProcessor(i, connectionQuotas, listenerName, securityProtocol, memoryPool)
126
127       val acceptor = new Acceptor(endpoint, sendBufferSize, recvBufferSize, brokerId,
128         processors.slice(processorBeginIndex, processorEndIndex), connectionQuotas)
129       acceptors.put(endpoint, acceptor)
130       KafkaThread.nonDaemon( name = s"kafka-socket-acceptor-$listenerName-$securityProtocol-${endpoint.port}",
          acceptor).start()
131       acceptor.awaitStartup()
132
133       processorBeginIndex = processorEndIndex
134     }
135     // 如果需要启动处理程序
136     if (startupProcessors) {
137       // 启动处理程序
138       startProcessors()
139     }
140   }
141
```

究竟启动多少个处理程序?

processor个数为numProcessorThreads个。上图中for循环为从processorBeginIndex到prodessorEndIndex（不包括）。

numProcessorThread为：

```scala
KafkaServer.scala | RequestChannel.scala | RequestOrResponseSend.scala | SocketServer.scala | KafkaCon
57  */
58  class SocketServer(
59                      val config: KafkaConfig, // 用于封装Kafka配置信息的对象
60                      val metrics: Metrics,    //
61                      val time: Time,          // 时间对象
62                      val credentialProvider: CredentialProvider
63                    ) extends Logging with KafkaMetricsGroup {
64
65    private val endpoints = config.listeners.map(l => l.listenerName -> l).toMap
66    private val numProcessorThreads = config.numNetworkThreads
67    private val maxQueuedRequests = config.queuedMaxRequests
```

```scala
KafkaServer.scala | RequestChannel.scala | SocketServer.scala | KafkaConfig.scala | Gauge.class
911   val maxReservedBrokerId: Int = getInt(KafkaConfig.MaxReservedBrokerIdProp
912   var brokerId: Int = getInt(KafkaConfig.BrokerIdProp)
913
914   val numNetworkThreads = getInt(KafkaConfig.NumNetworkThreadsProp)
915   val backgroundThreads = getInt(KafkaConfig.BackgroundThreadsProp)
916   val queuedMaxRequests = getInt(KafkaConfig.QueuedMaxRequestsProp)
```

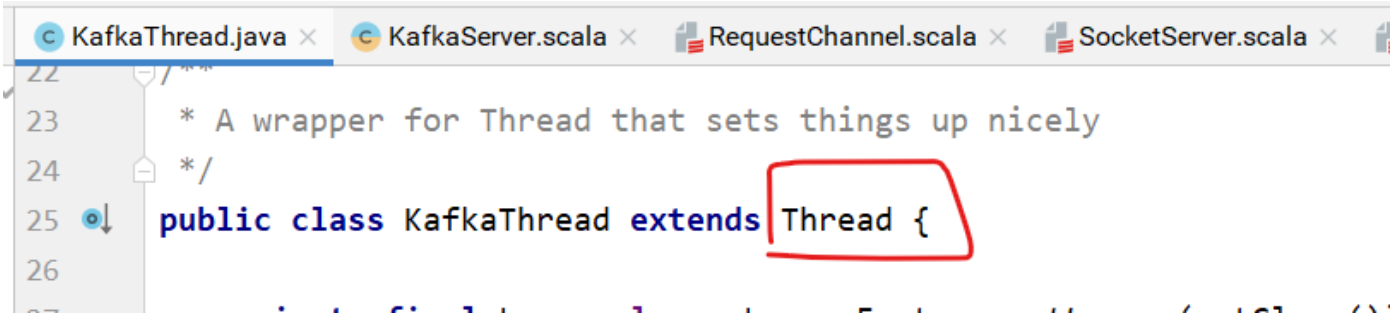
```scala
KafkaServer.scala | RequestChannel.scala | SocketServer.scala | KafkaConfig.scala | G
237   val NumNetworkThreadsProp = "num.network.threads"
238   val NumIoThreadsProp = "num.io.threads"
```

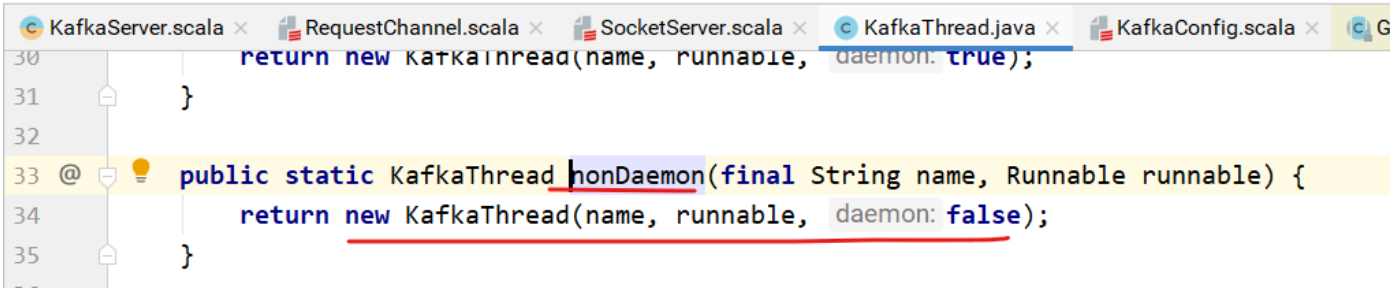
```
1 node1 | 2 node2 | 3 node3 | 4 node4 | +
# Maps listener names to security protocols, the default is for them to be the same. See the config documentation for more detai
#listener.security.protocol.map=PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL

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

acceptor的启动过程：

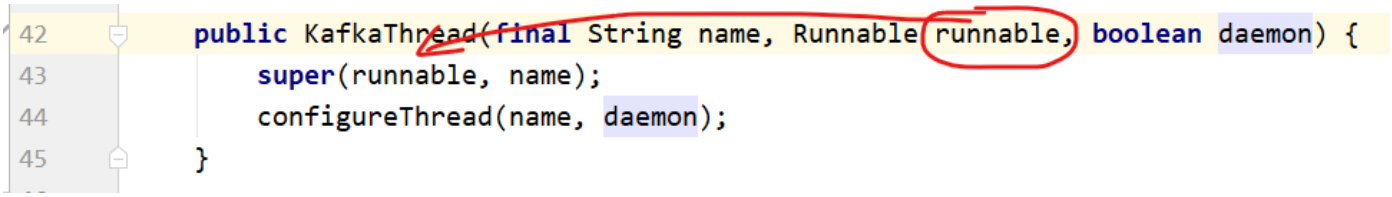
```scala
KafkaServer.scala | RequestChannel.scala | SocketServer.scala | KafkaThread.java | KafkaConfig.scala | Gauge.class
128     processors.slice(processorBeginIndex, processorEndIndex), connectionQuotas)
129   acceptors.put(endpoint, acceptor)
130   KafkaThread.nonDaemon( name = s"kafka-socket-acceptor-$listenerName-$securityProtocol-${endpoint.port}", acceptor).start()
131   acceptor.awaitStartup()
```

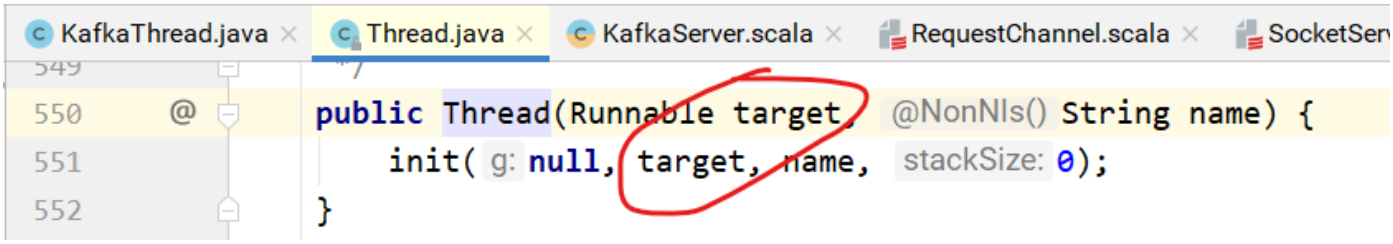
KafkaThread：





调用Thread的构造器：





KafkaThread的start方法即是Thread的start方法，此时调用的是acceptor的run方法：

```
/**
 * 使用Java的NIO
 * 循环检查是否有新的连接尝试
 * 轮询的方式将请求交给各个processor来处理。
 */
def run() {
  serverChannel.register(nioSelector, SelectionKey.OP_ACCEPT)
  startupComplete()
  try {
    var currentProcessor = 0
    while (isRunning) {
      try {
        val ready = nioSelector.select(500)
        if (ready > 0) {
          val keys = nioSelector.selectedKeys()
          val iter = keys.iterator()
          while (iter.hasNext && isRunning) {
```

```
              try {
                val key = iter.next
                iter.remove()
                if (key.isAcceptable)
                  // 指定一个processor处理请求
                  accept(key, processors(currentProcessor))
                else
                  throw new IllegalStateException("Unrecognized key state for acceptor thread.")

                // round robin to the next processor thread
                // 通过轮询的方式找到下一个processor线程
                currentProcessor = (currentProcessor + 1) % processors.length
              } catch {
                case e: Throwable => error("Error while accepting connection", e)
              }
            }
          }
        }
        catch {
          // We catch all the throwables to prevent the acceptor thread from exiting on exceptions due
          // to a select operation on a specific channel or a bad request. We don't want
          // the broker to stop responding to requests from other clients in these scenarios.
          case e: ControlThrowable => throw e
          case e: Throwable => error("Error occurred", e)
        }
      }
    } finally {
      debug("Closing server socket and selector.")
      swallowError(serverChannel.close())
      swallowError(nioSelector.close())
      shutdownComplete()
    }
  }
```

Acceptor建立连接，处理请求：

```
  /*
   * Accept a new connection
   * 建立一个新连接
   */
  def accept(key: SelectionKey, processor: Processor) {
    // 服务端
    val serverSocketChannel = key.channel().asInstanceOf[ServerSocketChannel]
    // 客户端
    val socketChannel = serverSocketChannel.accept()
```

```
    try {
      connectionQuotas.inc(socketChannel.socket().getInetAddress)
      // 非阻塞
      socketChannel.configureBlocking(false)
      socketChannel.socket().setTcpNoDelay(true)
      socketChannel.socket().setKeepAlive(true)
      // 设置发送缓冲大小
      if (sendBufferSize != Selectable.USE_DEFAULT_BUFFER_SIZE)
        socketChannel.socket().setSendBufferSize(sendBufferSize)

      debug("Accepted connection from %s on %s and assigned it to processor %d, sendBufferSize [actual|requested]: [%d|%d] recvBufferSize [actual|requested]: [%d|%d]"
            .format(socketChannel.socket.getRemoteSocketAddress, socketChannel.socket.getLocalSocketAddress, processor.id,
                    socketChannel.socket.getSendBufferSize, sendBufferSize,
                    socketChannel.socket.getReceiveBufferSize, recvBufferSize))
      // 调用Processor的accept方法，由processor处理请求
      processor.accept(socketChannel)
    } catch {
      case e: TooManyConnectionsException =>
        info("Rejected connection from %s, address already has the configured maximum of %d connections.".format(e.ip, e.count))
        close(socketChannel)
    }
  }
```

Processor将连接加入缓冲队列，同时唤醒处理线程：

Processor的run方法从newConnections中取出请求的channel，解析封装请求，交给handler处理：

RequestChannel.scala | SocketServer.scala | Selector.java | SelectorImpl.class | Wind

```scala
override def run() {
  // 启动完成
  startupComplete()
  try {
    while (isRunning) {
      try {
        // setup any new connections that have been queued up
        // 在选择器注册channel
        configureNewConnections()
        // register any new responses for writing
        // 注册新的响应对象
        processNewResponses()
        poll()
        // 接收
        processCompletedReceives()
        // 发送
        processCompletedSends()
        // 断开连接
        processDisconnected()
```

RequestChannel.scala | SocketServer.scala | Selector.java | SelectorImpl.class | WindowsSelectorImpl.class | Thread.java | Kafk

```scala
private def processCompletedReceives() {
  selector.completedReceives.asScala.foreach { receive =>
    try {
      openOrClosingChannel(receive.source) match {
        case Some(channel) =>
          // 获取请求消息头
          val header = RequestHeader.parse(receive.payload)
          // 封装请求上下文
          val context = new RequestContext(header, receive.source, channel.socketAddress,
            channel.principal, listenerName, securityProtocol)
          // 创建Request对象封装请求信息
          val req = new RequestChannel.Request(processor = id, context = context,
            startTimeNanos = time.nanoseconds, memoryPool, receive.payload, requestChannel.metrics)
          // 发送请求信息
          requestChannel.sendRequest(req)
          selector.mute(receive.source)
        case None =>
          // This should never happen since completed receives are processed immediately after `poll`
```

将请求信息放到请求队列中：

RequestChannel.scala | SocketServer.scala | Selector.java | SelectorImpl.class | Wi

```
271     * Send a request to be handled, potentially blocking until th
272     * 发送等待处理的请求，入队列，该方法有可能阻塞，等待队列中有空间
273     */
274    def sendRequest(request: RequestChannel.Request) {
275      requestQueue.put(request)
276    }
```

在KafkaServer的startup方法中实例化KafkaRequestHandlerPool，该类会立即初始化numIoThreads个线程用于执行KafkaRequestHandler处理请求的逻辑。

SocketServer.scala | RequestChannel.scala | KafkaRequestHandler.scala | KafkaServer.scala | Selector.java

```
289      }
290
291      // KafkaApi包含了各种请求的具体处理逻辑
292      apis = new KafkaApis(socketServer.requestChannel, replicaManager, adminManager, groupCoordinator, transactionCoordinator,
293        kafkaController, zkUtils, config.brokerId, config, metadataCache, metrics, authorizer, quotaManagers,
294        brokerTopicStats, clusterId, time)
295      // 将KafkaApi实例赋值给KafkaRequestHandlerPool，以执行各种请求的处理
296      requestHandlerPool = new KafkaRequestHandlerPool(config.brokerId, socketServer.requestChannel, apis, time,
297        config.numIoThreads)
298
```

KafkaRequestHandlerPool以多线程的方式启动多个KafkaRequestHandler：

RequestChannel.scala | KafkaRequestHandler.scala | SocketServer.scala | Selector.java | SelectorImpl.class | WindowsSelectorImpl.class | Thread.java | KafkaCo

```
86   class KafkaRequestHandlerPool(val brokerId: Int,
87                                 val requestChannel: RequestChannel,
88                                 val apis: KafkaApis,
89                                 time: Time,
90                                 numThreads: Int) extends Logging with KafkaMetricsGroup {
91
92     /* a meter to track the average free capacity of the request handlers */
93     private val aggregateIdleMeter = newMeter( name = "RequestHandlerAvgIdlePercent", eventType = "percent", TimeUnit.NANOSECONDS)
94
95     this.logIdent = "[Kafka Request Handler on Broker " + brokerId + "], "
96     val runnables = new Array[KafkaRequestHandler](numThreads)
97     for(i <- 0 until numThreads) {
98       runnables(i) = new KafkaRequestHandler(i, brokerId, aggregateIdleMeter, numThreads, requestChannel, apis, time)
99       // 启动这些KafkaRequestHandler线程用于请求的处理
100      KafkaThread.daemon( name = "kafka-request-handler-" + i, runnables(i)).start()
101    }
```

KafkaRequestHandler的run方法中，receiveRequest方法从请求队列获取请求：

```scala
def run() {
  while(true) {
    // We use a single meter for aggregate idle percentage for
    // Since meter is calculated as total_recorded_value / time
    // time_window is independent of the number of threads, eac
    // time should be discounted by # threads.
    val startSelectTime = time.nanoseconds
    // 获取请求
    val req = requestChannel.receiveRequest( timeout = 300)
    val endTime = time.nanoseconds
```

具体实现：

```scala
/**
 * Get the next request or block until specified time has elapsed
 * 获取下个请求，如果队列中没有请求，就等待指定的时间
 */
def receiveRequest(timeout: Long): RequestChannel.BaseRequest =
  requestQueue.poll(timeout, TimeUnit.MILLISECONDS)
```

KafkaRequestHandler的run方法中使用模式匹配：

```scala
case request: RequestChannel.Request =>
  try {
    request.requestDequeueTimeNanos = endTime
    trace( msg = s"Kafka request handler $id on broker $brokerId handling request $request")
    apis.handle(request)
  } catch {
```

上图中，apis的handle方法处理请求：

```scala
/**
 * 处理所有请求的顶级方法，使用模式匹配，交给具体的api来处理
 */
def handle(request: RequestChannel.Request) {
  try {
    trace(s"Handling request:${request.requestDesc(true)} from connection ${request.context.connectionId};" +
      s"securityProtocol:${request.context.securityProtocol},principal:${request.context.principal}")
    request.header.apiKey match {
      case ApiKeys.PRODUCE => handleProduceRequest(request)
```

```
        case ApiKeys.FETCH => handleFetchRequest(request)
        case ApiKeys.LIST_OFFSETS => handleListOffsetRequest(request)
        case ApiKeys.METADATA => handleTopicMetadataRequest(request)
        case ApiKeys.LEADER_AND_ISR => handleLeaderAndIsrRequest(request)
        case ApiKeys.STOP_REPLICA => handleStopReplicaRequest(request)
        case ApiKeys.UPDATE_METADATA => handleUpdateMetadataRequest(request)
        case ApiKeys.CONTROLLED_SHUTDOWN => handleControlledShutdownRequest(request)
        case ApiKeys.OFFSET_COMMIT => handleOffsetCommitRequest(request)
        case ApiKeys.OFFSET_FETCH => handleOffsetFetchRequest(request)
        case ApiKeys.FIND_COORDINATOR => handleFindCoordinatorRequest(request)
        case ApiKeys.JOIN_GROUP => handleJoinGroupRequest(request)
        case ApiKeys.HEARTBEAT => handleHeartbeatRequest(request)
        case ApiKeys.LEAVE_GROUP => handleLeaveGroupRequest(request)
        case ApiKeys.SYNC_GROUP => handleSyncGroupRequest(request)
        case ApiKeys.DESCRIBE_GROUPS => handleDescribeGroupRequest(request)
        case ApiKeys.LIST_GROUPS => handleListGroupsRequest(request)
        case ApiKeys.SASL_HANDSHAKE => handleSaslHandshakeRequest(request)
        case ApiKeys.API_VERSIONS => handleApiVersionsRequest(request)
        case ApiKeys.CREATE_TOPICS => handleCreateTopicsRequest(request)
        case ApiKeys.DELETE_TOPICS => handleDeleteTopicsRequest(request)
        case ApiKeys.DELETE_RECORDS => handleDeleteRecordsRequest(request)
        case ApiKeys.INIT_PRODUCER_ID => handleInitProducerIdRequest(request)
        case ApiKeys.OFFSET_FOR_LEADER_EPOCH =>
handleOffsetForLeaderEpochRequest(request)
        case ApiKeys.ADD_PARTITIONS_TO_TXN => handleAddPartitionToTxnRequest(request)
        case ApiKeys.ADD_OFFSETS_TO_TXN => handleAddOffsetsToTxnRequest(request)
        case ApiKeys.END_TXN => handleEndTxnRequest(request)
        case ApiKeys.WRITE_TXN_MARKERS => handleWriteTxnMarkersRequest(request)
        case ApiKeys.TXN_OFFSET_COMMIT => handleTxnOffsetCommitRequest(request)
        case ApiKeys.DESCRIBE_ACLS => handleDescribeAcls(request)
        case ApiKeys.CREATE_ACLS => handleCreateAcls(request)
        case ApiKeys.DELETE_ACLS => handleDeleteAcls(request)
        case ApiKeys.ALTER_CONFIGS => handleAlterConfigsRequest(request)
        case ApiKeys.DESCRIBE_CONFIGS => handleDescribeConfigsRequest(request)
        case ApiKeys.ALTER_REPLICA_LOG_DIRS =>
handleAlterReplicaLogDirsRequest(request)
        case ApiKeys.DESCRIBE_LOG_DIRS => handleDescribeLogDirsRequest(request)
        case ApiKeys.SASL_AUTHENTICATE => handleSaslAuthenticateRequest(request)
        case ApiKeys.CREATE_PARTITIONS => handleCreatePartitionsRequest(request)
      }
    } catch {
      case e: FatalExitError => throw e
      case e: Throwable => handleError(request, e)
    } finally {
      request.apiLocalCompleteTimeNanos = time.nanoseconds
    }
  }
```

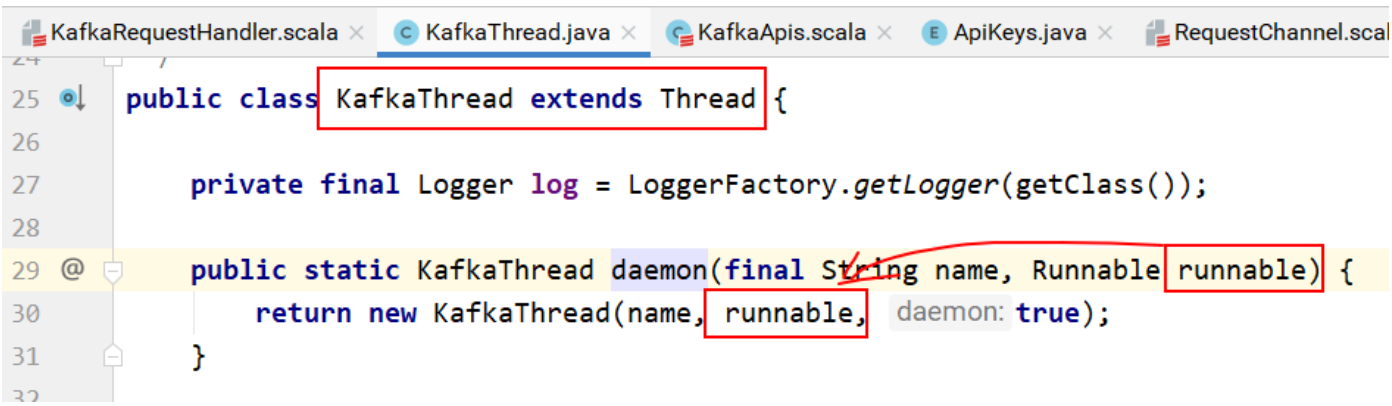
4.8 Kafka源码剖析之KafkaRequestHandlerPool

KafkaRequestHandlerPool的作用是创建numThreads个KafkaRequestHandler实例，使用numThreads个线程启动KafkaRequestHandler。

每个KafkaRequestHandler包含了id，brokerId，线程数，请求的channel，处理请求的api等信息。

只要该类进行实例化，就执行创建KafkaRequestHandler实例并启动的逻辑。

```
/**
 * @param brokerId
 * @param requestChannel
 * @param apis 处理具体请求和响应的api
 * @param time
 * @param numThreads 运行KafkaRequestHandler的线程数
 */
class KafkaRequestHandlerPool(val brokerId: Int,
                              val requestChannel: RequestChannel,
                              val apis: KafkaApis,
                              time: Time,
                              numThreads: Int) extends Logging with KafkaMetricsGroup
{

  // 创建包含numThreads个元素的数组
  val runnables = new Array[KafkaRequestHandler](numThreads)
  // 循环numThreads次，初始化KafkaRequestHandler实例numThreads个
  for(i <- 0 until numThreads) {
    // 赋值：每个KafkaRequestHandler中包含了KafkaRequestHandler的id，brokerId，线程数，请求的channel，处理请求的api等。
    runnables(i) = new KafkaRequestHandler(i, brokerId, aggregateIdleMeter, numThreads, requestChannel, apis, time)
    // 启动这些KafkaRequestHandler线程用于请求的处理
    KafkaThread.daemon("kafka-request-handler-" + i, runnables(i)).start()
  }
}
```



KafkaThread的start方法即是调用Thread的start方法，而start执行run方法，即此处执行的是KafkaThread的run方法：

```
  def run() {
    while(true) {
      // We use a single meter for aggregate idle percentage for the thread pool.
      // Since meter is calculated as total_recorded_value / time_window and
      // time_window is independent of the number of threads, each recorded idle
      // time should be discounted by # threads.
      val startSelectTime = time.nanoseconds
      // 获取请求
      val req = requestChannel.receiveRequest(300)
      val endTime = time.nanoseconds
      val idleTime = endTime - startSelectTime
      aggregateIdleMeter.mark(idleTime / totalHandlerThreads)

      req match {
        case RequestChannel.ShutdownRequest =>
          debug(s"Kafka request handler $id on broker $brokerId received shut down command")
          latch.countDown()
          return

        case request: RequestChannel.Request =>
          try {
            request.requestDequeueTimeNanos = endTime
            trace(s"Kafka request handler $id on broker $brokerId handling request $request")
            // 对于其他请求，直接交给apis来负责处理。
            apis.handle(request)
          } catch {
            case e: FatalExitError =>
              latch.countDown()
              Exit.exit(e.statusCode)
            case e: Throwable => error("Exception when handling request", e)
          } finally {
            request.releaseBuffer()
          }

        case null => // continue
      }
    }
  }
```

KafkaRequestHandler.scala | KafkaThread.java | KafkaApis.scala | ApiKeys.java | RequestChannel.scala

```
91    * Top-level method that handles all requests and multiplexes to the right api
92    * 处理所有请求的顶级方法，使用模式匹配，交给具体的api来处理
93    */
94   def handle(request: RequestChannel.Request) {
95     try {
96       trace( msg = s"Handling request:${request.requestDesc( details = true)} from connect
97         s"securityProtocol:${request.context.securityProtocol},principal:${request.con
98       request.header.apiKey match { 模式匹配
99         case ApiKeys.PRODUCE => handleProduceRequest(request)
100        case ApiKeys.FETCH => handleFetchRequest(request)
```

该类包含了关闭KafkaRequestHandler的方法：

```
114   /**
115    * 关闭KafkaRequestHandler线程
116    */
117   def shutdown() {
118     info( msg = "shutting down")
119     for (handler <- runnables)
120       handler.initiateShutdown()
121     for (handler <- runnables)
122       handler.awaitShutdown()
123     info( msg = "shut down completely")
124   }
```

具体的方法：

KafkaRequestHandler.scala

```
78   }
79
80   def initiateShutdown(): Unit = requestChannel.sendShutdownRequest()
81
82   def awaitShutdown(): Unit = latch.await()
```

首先发送停止的请求，等待用户请求处理的结束`latch.await()`。

优雅停机。

KafkaRequestHandler.scala | RequestChannel.scala

```
337    def sendShutdownRequest(): Unit = requestQueue.put(ShutdownRequest)
338  }
```

将请求直接放到requestQueue中。

其中处理ShutdownRequest的处理逻辑：

KafkaRequestHandler.scala | KafkaApis.scala | ApiKeys.java | RequestChannel.scala

```
42  def run() {
43    while(true) {
44      // We use a single meter for aggregate idle percentage for the thread pool.
45      // Since meter is calculated as total_recorded_value / time_window and
46      // time_window is independent of the number of threads, each recorded idle
47      // time should be discounted by # threads.
48      val startSelectTime = time.nanoseconds
49      // 获取请求
50      val req = requestChannel.receiveRequest( timeout = 300)
51      val endTime = time.nanoseconds
52      val idleTime = endTime - startSelectTime
53      aggregateIdleMeter.mark(idleTime / totalHandlerThreads)
54
55      req match {
56        case RequestChannel.ShutdownRequest =>
57          debug( msg = s"Kafka request handler $id on broker $brokerId received shut down command")
58          latch.countDown()
59          return
60
```

4.9 Kafka源码剖析之LogManager

1. kafka日志管理子系统的入口。日志管理器负责日志的创建、抽取、和清理。
2. 所有的读写操作都代理给具体的Log实例。
3. 日志管理器在一个或多个目录维护日志。新的日志创建到拥有最少log的目录中。
4. 分区不移动。
5. 通过一个后台线程通过定期截断多余的日志段来处理日志保留。

启动Kafka服务器的脚本：

kafka-server-start.sh | KafkaServer.scala | KafkaController.scala | ControllerEventManager.

```
44  exec $base_dir/kafka-run-class.sh $EXTRA_ARGS kafka.Kafka "$@"
```

main方法中创建KafkaServerStartable对象：

kafka-server-start.sh | Kafka.scala | KafkaServerStartable.scala | KafkaServer.scala | KafkaController.scala | ControllerEventManag

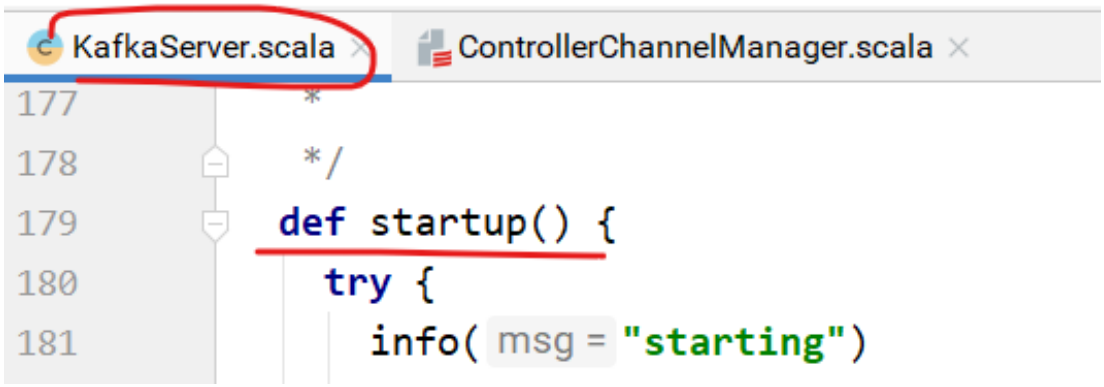
```scala
def main(args: Array[String]): Unit = {
  try {
    // 从命令参数解析kafka的代码
    val serverProps = getPropsFromArgs(args)
    val kafkaServerStartable = KafkaServerStartable.fromProps(serverProps)

    // register signal handler to log termination due to SIGTERM, SIGHUP and SIGINT (control-c)
    registerLoggingSignalHandler()

    // attach shutdown handler to catch terminating signals as well as normal termination
    Runtime.getRuntime().addShutdownHook(new Thread( name = "kafka-shutdown-hook") {
      override def run(): Unit = kafkaServerStartable.shutdown()
    })

    // 启动服务
    kafkaServerStartable.startup()
    kafkaServerStartable.awaitShutdown()
  }
  catch {
```

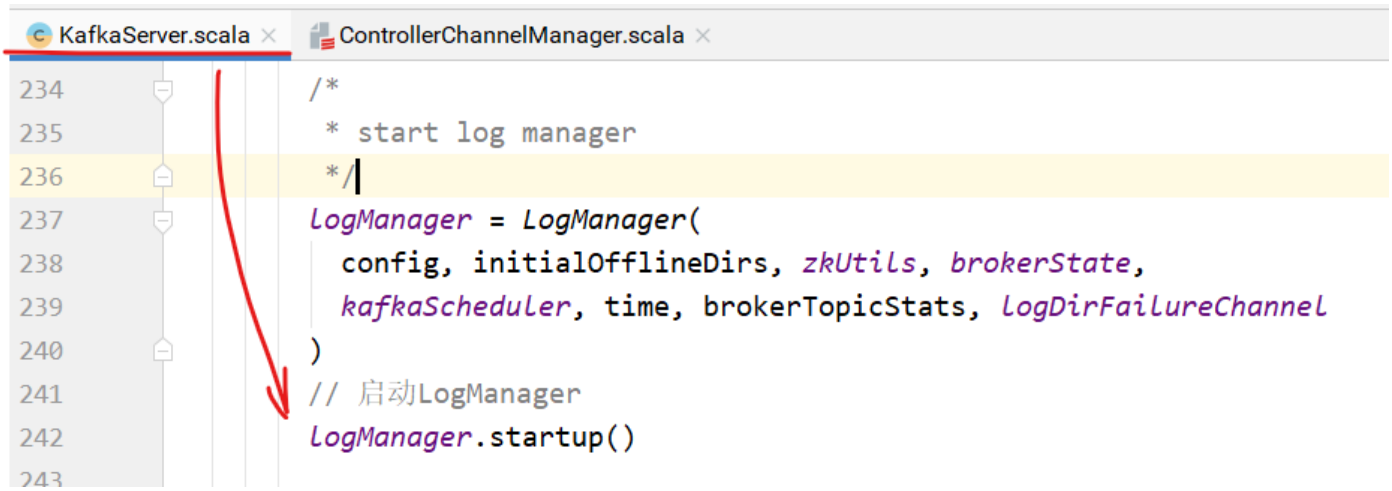
该类中包含KakfaServer对象，startup方法调用的是KafkaServer的startup方法：

kafka-server-start.sh | Kafka.scala | KafkaServerStartable.scala | KafkaServer.scala | KafkaController.scala | ControllerEven

```scala
object KafkaServerStartable {
  def fromProps(serverProps: Properties): KafkaServerStartable = {
    val reporters = KafkaMetricsReporter.startReporters(new VerifiableProperties(serverProps))
    new KafkaServerStartable(KafkaConfig.fromProps(serverProps), reporters)
  }
}

class KafkaServerStartable(val serverConfig: KafkaConfig, reporters: Seq[KafkaMetricsReporter]) {
  private val server = new KafkaServer(serverConfig, kafkaMetricsReporters = reporters)

  def this(serverConfig: KafkaConfig) = this(serverConfig, Seq.empty)

  def startup() {
    try server.startup()
    catch {
      case _: Throwable =>
        // KafkaServer.startup() calls shutdown() in case of exceptions, so we invoke `exit` to
        fatal( msg = "Exiting Kafka.")
        Exit.exit( statusCode = 1)
    }
  }

  def shutdown() {
```

KafkaServer的startup方法中，启动了LogManager：






```
/**
 * @param logDirs 主题分区目录的File对象
 * @param initialOfflineDirs
 * @param topicConfigs 主题配置
 * @param defaultConfig 主题的默认配置
 * @param cleanerConfig 日志清理器配置
 * @param ioThreads IO线程数
 * @param flushCheckMs
 * @param flushRecoveryOffsetCheckpointMs
 * @param flushStartOffsetCheckpointMs
 * @param retentionCheckMs 检查日志保留的时间
 * @param maxPidExpirationMs
 * @param scheduler
 * @param brokerState
 * @param brokerTopicStats
 * @param logDirFailureChannel
 * @param time 时间
 */
```

Log.scala | LogSegment.scala | LogManager.scala

```
@threadsafe
class LogManager(logDirs: Seq[File],
                 initialOfflineDirs: Seq[File],
                 val topicConfigs: Map[String, LogConfig], // note that this doesn't get updated after creation
                 val defaultConfig: LogConfig,
                 val cleanerConfig: CleanerConfig,
                 ioThreads: Int,
                 val flushCheckMs: Long,
                 val flushRecoveryOffsetCheckpointMs: Long,
                 val flushStartOffsetCheckpointMs: Long,
                 val retentionCheckMs: Long,
                 val maxPidExpirationMs: Int,
                 scheduler: Scheduler,
                 val brokerState: BrokerState,
                 brokerTopicStats: BrokerTopicStats,
                 logDirFailureChannel: LogDirFailureChannel,
                 time: Time) extends Logging with KafkaMetricsGroup {

  import LogManager
```

LogManager的startup方法：

```
  /**
   *  启动后台线程们用于将日志刷盘以及日志的清理
   */
  def startup() {

    if (scheduler != null) {
      info("Starting log cleanup with a period of %d ms.".format(retentionCheckMs))
      // 用于清除日志片段的调度任务，没有压缩，周期性执行
      scheduler.schedule("kafka-log-retention",
                         cleanupLogs _,
                         delay = InitialTaskDelayMs,
                         period = retentionCheckMs,
                         TimeUnit.MILLISECONDS)
      info("Starting log flusher with a default period of %d
ms.".format(flushCheckMs))
      // 用于日志片段刷盘的调度任务，周期性执行
      scheduler.schedule("kafka-log-flusher",
                         flushDirtyLogs _,
                         delay = InitialTaskDelayMs,
                         period = flushCheckMs,
                         TimeUnit.MILLISECONDS)
      // 用于将当前broker上各个分区的恢复点写到文本文件的调度任务，周期性执行
      scheduler.schedule("kafka-recovery-point-checkpoint",
                         checkpointLogRecoveryOffsets _,
                         delay = InitialTaskDelayMs,
                         period = flushRecoveryOffsetCheckpointMs,
                         TimeUnit.MILLISECONDS)
      // 用于将当前broker上各个分区起始偏移量写到文本文件的调度任务，周期性执行
      scheduler.schedule("kafka-log-start-offset-checkpoint",
                         checkpointLogStartOffsets _,
                         delay = InitialTaskDelayMs,
                         period = flushStartOffsetCheckpointMs,
```

```
                          TimeUnit.MILLISECONDS)
      scheduler.schedule("kafka-delete-logs",
                          deleteLogs _,
                          delay = InitialTaskDelayMs,
                          period = defaultConfig.fileDeleteDelayMs,
                          TimeUnit.MILLISECONDS)
    }
    // 如果配置了日志的清理，则启动清理任务
    if (cleanerConfig.enableCleaner)
      cleaner.startup()
  }
```

4.9.1 清除日志片段

LogManager.scala × KafkaServer.scala × Log.scala ×

```
437     if (scheduler != null) {
438       info("Starting log cleanup with a period of %d ms.".format(retentionCheckMs))
439       // 用于清除日志片段的调度任务，没有压缩，周期性执行
440       scheduler.schedule( name = "kafka-log-retention",
441                           cleanupLogs _,
442                           delay = InitialTaskDelayMs,
443                           period = retentionCheckMs,
444                           TimeUnit.MILLISECONDS)
```

cleanupLogs的具体实现：

LogManager.scala × KafkaServer.scala × Log.scala ×

```
821
822   /**
823    * 删除符合要求的日志。
824    * 此处只考虑日志不压缩的情况。
825    * @return 返回删除的日志片段的个数
826    */
827   def cleanupLogs() {
828     debug( msg = "Beginning log cleanup...")
829     var total = 0
830     val startMs = time.milliseconds
831
832     // 遍历所有主题分区日志，如果日志不压缩，则执行删除
833     for(log <- allLogs; if !log.config.compact) {
834       debug( msg = "Garbage collecting '" + log.name + "'")
835       // 对所有主题分区日志，执行删除，计算删除的日志片段的个数
836       total += log.deleteOldSegments()
837     }
838     debug( msg = "Log cleanup completed. " + total + " files deleted in " +
839                  (time.milliseconds - startMs) / 1000 + " seconds")
840   }
```

deleteOldSegments()的实现：

LogManager.scala | KafkaServer.scala | Log.scala

```
1324  /**
1325   * 对于超过保存策略时长的或超过保存策略大小(log size is > retentionSize)的日志片段，执行删除
1326   * @return 删除的日志分段个数
1327   */
1328  def deleteOldSegments(): Int = {
1329    // 如果配置的不是删除，比如说是压缩，则不删除
1330    if (!config.delete) return 0
1331    // 根据时间删除的日志片段数量+根据保留大小删除的日志片段的个数+根据偏移量删除日志片段的个数
1332    deleteRetentionMsBreachedSegments() + deleteRetentionSizeBreachedSegments() + deleteLogStartOffsetBreachedSegments()
1333  }
```

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```
1333  }
1334
1335  /**
1336   * 根据保留策略的超时时长，删除超时的旧的日志片段文件
1337   * @return
1338   */
1339  private def deleteRetentionMsBreachedSegments(): Int = {
1340    // retention.ms，默认7天
1341    if (config.retentionMs < 0) return 0
1342    // 获取当前时间
1343    val startMs = time.milliseconds
1344    // 删除超过保留时长的旧的日志片段
1345    deleteOldSegments((segment, _) => startMs - segment.largestTimestamp > config.retentionMs,
1346      reason = s"retention time ${config.retentionMs}ms breach")
1347  }
```

首先找到所有可以删除的日志片段

然后执行删除

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```
1251  /**
1252   * 删除任何入参函数值为true的日志片段。
1253   * 从最旧的开始删除，直到条件不满足为止
1254   *
1255   * @param predicate A function that takes in a candidate log segment and the next higher segment
1256   *                  (if there is one) and returns true iff it is deletable
1257   * @return 被删的日志片段个数
1258   */
1259  private def deleteOldSegments(predicate: (LogSegment, Option[LogSegment]) => Boolean, reason: String): Int = {
1260    lock synchronized {
1261      // 找到所有需要被删除的，判断条件通过函数传入
1262      val deletable = deletableSegments(predicate)
1263      if (deletable.nonEmpty)
1264        info( msg = s"Found deletable segments with base offsets [${deletable.map(_.baseOffset).mkString(",")}] du
1265      // 执行删除
1266      deleteSegments(deletable)
1267    }
1268  }
```

```scala
Log.scala × LogConfig.scala × TopicConfig.java × LogManager.scala ×
1269
1270  private def deleteSegments(deletable: Iterable[LogSegment]): Int = {
1271    maybeHandleIOException(s"Error while deleting segments for $topicPartition in dir ${dir.getParent}") {
1272      val numToDelete = deletable.size
1273      if (numToDelete > 0) {
1274        // 永远保留一个日志片段，如果所有的日志片段都要删除，则滚动出一个新的来
1275        if (segments.size == numToDelete)
1276          roll()
1277        lock synchronized {
1278          // 确认日志片段的MMap已经关闭，不可再读写
1279          checkIfMemoryMappedBufferClosed()
1280          // 迭代删除
1281          deletable.foreach(deleteSegment)
1282          // 删除后有可能需要更新LogStartOffset
1283          maybeIncrementLogStartOffset(segments.firstEntry.getValue.baseOffset)
1284        }
1285      }
1286      numToDelete
1287    }
1288  }
```

该方法执行日志片段的异步删除。步骤如下：

1. 将日志片段的信息从map集合移除，之后再也不读了
2. 在日志片段的索引和log文件名称后追加.deleted，加标记而已
3. 调度异步删除操作，执行.deleted文件的真正删除。

异步删除允许在读取文件的同时执行删除，而不需要进行同步，避免了在读取一个文件的同时物理删除引起的冲突。

该方法不需要将IOException转换为KafkaStorageException，因为该方法要么在所有日志加载之前调用，要么在使用中由调用者处理IOException。

```scala
Log.scala × LogConfig.scala × TopicConfig.java × LogManager.scala ×
1761
1762  /**
1763   * 该方法执行日志片段的异步删除，步骤如下：
1764   * <ol>
1765   *   <li>将日志片段从片段map集合移除，之后再也不读了。
1766   *   <li>在日志片段的索引和log文件名称后追加.deleted。加标记而已。
1767   *   <li>调度异步删除操作，执行.deleted文件的真正的删除。
1768   * </ol>
1769   * 异步删除允许在读取文件的同时执行删除任务，而不需要进行同步，也避免了在读取一个文件的同时物理删除该文件的冲突。
1770   * 该方法不需要将IOException转换为KafkaStorageException，因为该方法要么在所有日志加载之前调用，要么在中间调用者来捕获和处理IOException。
1771   *
1772   * @param segment 需要调度异步删除的日志片段
1773   */
1774  private def deleteSegment(segment: LogSegment) {
1775    info( msg = s"Scheduling log segment [baseOffset ${segment.baseOffset}, size ${segment.size}] for deletion.")
1776    lock synchronized {
1777      segments.remove(segment.baseOffset)
1778      asyncDeleteSegment(segment)
1779    }
1780  }
```

根据日志片段大小进行删除：

```scala
/**
 * 根据保留策略的日志片段文件大小删除
 * @return
 */
private def deleteRetentionSizeBreachedSegments(): Int = {
  // retention.bytes默认为-1。日志中所有片段大小为size
  if (config.retentionSize < 0 || size < config.retentionSize) return 0
  // 日志的大小减去需要保留的大小，剩下的都应该是删除的
  var diff = size - config.retentionSize
  // 判断日志片段是否需要删除
  def shouldDelete(segment: LogSegment, nextSegmentOpt: Option[LogSegment]): Boolean = {
    // 如果日志需要删除的大小减去当前日志片段大小还是大于0的，表示当前日志片段需要删除
    if (diff - segment.size >= 0) {
      // 同时计算还剩下多少字节是可以删除的。
      diff -= segment.size
      true
    } else {
      // 如果剩余的需要删除的字节个数减去当前日志片段大小小于零，则不能删除
      // 返回false
      false
    }
  }
  deleteOldSegments(shouldDelete, reason = s"retention size in bytes ${config.retentionSize} breach")
}
```

shouldDelete是一个函数，作为deleteOldSegments删除日志片段的判断条件。

根据偏移量删除日志片段：

对于当前日志片段是否需要删除，要看它的下一个日志片段的baseOffset是否小于等于日志对外暴露给消费者的日志偏移量，如果小，消费者不用读取，当前日志片段就可以删除。

```scala
/**
 * 根据偏移量计算日志片段是否需要删除
 * 根据是当前日志片段的下一个日志片段的最小偏移量是否小于等于日志对外暴露给消费者的起始偏移量
 * 如果小于等于，当前日志片段就可以删除
 * @return 删除日志片段的个数。
 */
private def deleteLogStartOffsetBreachedSegments(): Int = {
  def shouldDelete(segment: LogSegment, nextSegmentOpt: Option[LogSegment]): Boolean =
    // 下一个日志片段的最小偏移量的值是否小于日志起始偏移量
    // 如果小于起始偏移量，则当前日志片段的最大偏移量肯定小于日志起始偏移量
    // 小于，就可以删除当前日志片段
    nextSegmentOpt.exists(_.baseOffset <= logStartOffset)
  // 删除
  deleteOldSegments(shouldDelete, reason = s"log start offset $logStartOffset breach")
}
```

4.9.2 日志片段刷盘

在LogManager的startup中，启动了刷盘的线程：

调用flushDirtyLogs方法进行日志刷盘处理。

```
Log.scala × LogConfig.scala × TopicConfig.java × LogManager.scala ×
444                         TimeUnit.MILLISECONDS)
445       info("Starting log flusher with a default period of %d ms.".format(flushCheckMs))
446       // 用于日志片段刷盘的调度任务，周期性执行
447       scheduler.schedule( name = "kafka-log-flusher",
448                         flushDirtyLogs _,
449                         delay = InitialTaskDelayMs,
450                         period = flushCheckMs,
451                         TimeUnit.MILLISECONDS)
452       // 用于将当前broker上各个分区的恢复点写到文本文件的调度任务，周期性执行
```

Kafka推荐让操作系统后台进行刷盘，使用副本保证数据高可用，这样效率更高。

因此此种方式不推荐。

```
Log.scala × LogConfig.scala × TopicConfig.java × LogManager.scala ×
868
869   /**
870    * 将超过刷盘时间间隔还未曾刷盘的消息刷盘
871    */
872   private def flushDirtyLogs(): Unit = {
873     debug( msg = "Checking for dirty logs to flush...")
874     // 遍历当前broker上所有的主题分区和log的对应条目
875     for ((topicPartition, log) <- logs) {
876       try {
877         // 根据上此刷盘时间与当前时间的时间差判断是否需要将当前主题分区的日志刷盘
878         val timeSinceLastFlush = time.milliseconds - log.lastFlushTime
879         debug( msg = "Checking if flush is needed on " + topicPartition.topic + " flush interval  " + log.config.flushMs +
880               " last flushed " + log.lastFlushTime + " time since last flush: " + timeSinceLastFlush)
881         // 如果该时间差大于等于配置的主题分区日志刷盘时间间隔
882         // 执行刷盘     flush.ms默认为9223372036854775807，很显然不想让你用这个时间执行刷盘
883         if(timeSinceLastFlush >= log.config.flushMs)
884           log.flush
885       } catch {
886         case e: Throwable =>
887           error( msg = "Error flushing topic " + topicPartition.topic, e)
888       }
889     }
890   }
```

执行刷盘的方法：

```
1   /**
2    * 日志片段刷盘到offset-1的偏移量位置。
3    *
4    * @param offset 从上一个恢复点开始刷盘到该偏移量-1的位置。offset偏移量的不刷盘。
5    *               offset是新的恢复点值。
6    */
7   def flush(offset: Long) : Unit = {
```

```
    maybeHandleIOException(s"Error while flushing log for $topicPartition in dir ${dir.getParent} with offset $offset") {
      // 如果偏移量小于等于该日志的恢复点，则不需要刷盘
      if (offset <= this.recoveryPoint)
        return
      debug(s"Flushing log up to offset $offset, last flushed: $lastFlushTime, current time: ${time.milliseconds()}, " +
        s"unflushed: $unflushedMessages")
      // 遍历需要刷盘的日志片段
      for (segment <- logSegments(this.recoveryPoint, offset))
        // 执行刷盘
        segment.flush()

      lock synchronized {
        // 检查MMAP是否关闭
        checkIfMemoryMappedBufferClosed()
        // 如果偏移量大于恢复点
        if (offset > this.recoveryPoint) {
          // 设置新的恢复点，表示到达这个偏移量位置的消息都已经刷盘了
          this.recoveryPoint = offset
          // 设置当前时间为刷盘的时间
          lastflushedTime.set(time.milliseconds)
        }
      }
    }
  }
```

4.9.3 将当前broker上各个分区的恢复点写到文本文件

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```
452    // 用于将当前broker上各个分区的恢复点写到文本文件的调度任务，周期性执行
453    scheduler.schedule( name = "kafka-recovery-point-checkpoint",
454                        checkpointLogRecoveryOffsets _,
455                        delay = InitialTaskDelayMs,
456                        period = flushRecoveryOffsetCheckpointMs,
457                        TimeUnit.MILLISECONDS)
```

方法实现：

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```scala
/**
 * 将当前broker上所有日志的当前恢复点偏移量写到日志目录的文本文件中
 * 防止broker重启的时候恢复整个日志。只要从该偏移量恢复即可。
 */
def checkpointLogRecoveryOffsets() {
  liveLogDirs.foreach(checkpointLogRecoveryOffsetsInDir)
}
```

方法实现：

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```scala

/**
 * 为指定目录中的所有日志创建检查点
 * @param dir 指定的日志目录
 */
private def checkpointLogRecoveryOffsetsInDir(dir: File): Unit = {
  for {
    partitionToLog <- logsByDir.get(dir.getAbsolutePath)
    checkpoint <- recoveryPointCheckpoints.get(dir)
  } {
    try {
      // 将每个分区的恢复点写到文本文件中
      checkpoint.write(partitionToLog.mapValues(_.recoveryPoint))
      // 当恢复点写到文件中之后，删除快照文件。
      logs.values.foreach(_.deleteSnapshotsAfterRecoveryPointCheckpoint())
    } catch {
      case e: IOException =>
        logDirFailureChannel.maybeAddOfflineLogDir(dir.getAbsolutePath, msg = s
          s"file in directory $dir", e)
```

4.9.4 将当前broker上各个分区起始偏移量写到文本文件

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```scala
                        TimeUnit.MILLISECONDS)
    // 用于将当前broker上各个分区起始偏移量写到文本文件的调度任务，周期性执行
    scheduler.schedule( name = "kafka-log-start-offset-checkpoint",
                        checkpointLogStartOffsets _,
                        delay = InitialTaskDelayMs,
                        period = flushStartOffsetCheckpointMs,
                        TimeUnit.MILLISECONDS)
```

方法实现：

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```
605   /**
606    * 将当前日志的起始偏移量写到日志目录的文本文件中
607    * 避免将已经通过DeleteRecordsRequest请求删除的数据暴露出去。
608    */
609   def checkpointLogStartOffsets() {
610     liveLogDirs.foreach(checkpointLogStartOffsetsInDir)
611   }
612
```

写文本文件：

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```
636   * 对给定目录中所有的日志创建起始偏移量记录点
637   */
638  private def checkpointLogStartOffsetsInDir(dir: File): Unit = {
639    for {
640      partitionToLog <- logsByDir.get(dir.getAbsolutePath)
641      checkpoint <- logStartOffsetCheckpoints.get(dir)
642    } {
643      try {
644        val logStartOffsets = partitionToLog.filter { case (_, log) =>
645          // 日志起始偏移量大于日志片段的基础偏移量时需要写起始偏移量。
646          log.logStartOffset > log.logSegments.head.baseOffset
647        }.mapValues(_.logStartOffset)
648        checkpoint.write(logStartOffsets)
649      } catch {
650        case e: IOException =>
651          logDirFailureChannel.maybeAddOfflineLogDir(dir.getAbsolutePath, m
652      }
653    }
654  }
```

4.9.5 删除日志片段

```
Log.scala × LogConfig.scala × TopicConfig.java × LogManager.scala ×
463                         TimeUnit.MILLISECONDS)
464       scheduler.schedule( name = "kafka-delete-logs",
465                         deleteLogs _,
466                         delay = InitialTaskDelayMs,
467                         period = defaultConfig.fileDeleteDelayMs,
468                         TimeUnit.MILLISECONDS)
```

对标记为删除的日志执行删除的动作：

```
Log.scala × LogConfig.scala × TopicConfig.java × LogManager.scala ×
728   /**
729    *  删除标记为删除的日志
730    */
731   private def deleteLogs(): Unit = {
732     try {
733       while (!logsToBeDeleted.isEmpty) {
734         val removedLog = logsToBeDeleted.take()
735         if (removedLog != null) {
736           try {
737             removedLog.delete()
738             info( msg = s"Deleted log for partition ${removedLog.topicPa
739           } catch {
740             case e: KafkaStorageException =>
741               error( msg = s"Exception while deleting $removedLog in dir
742           }
743         }
```

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```
1600  /**
1601   * 立即将日志目录以及内容从文件系统删除。
1602   */
1603  private[log] def delete() {
1604    maybeHandleIOException(s"Error while deleting log for $top
1605      lock synchronized {
1606        checkIfMemoryMappedBufferClosed()
1607        // 迭代执行日志片段的删除
1608        logSegments.foreach(_.delete())
1609        // 清空日志片段元数据
1610        segments.clear()
1611        // 清空日志leader纪元元数据
1612        leaderEpochCache.clear()
1613        // 删除目录
1614        Utils.delete(dir)
1615        // MMap缓存标记为已关闭
1616        isMemoryMappedBufferClosed = true
1617      }
1618    }
1619  }
```

4.9.6 clearner

如果配置了日志清理，则启动清理任务：

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala

```
468                                TimeUnit.MILLISECONDS)
469    }
470    // 如果配置了日志的清理，则启动清理任务
471    if (cleanerConfig.enableCleaner)
472      cleaner.startup()
473  }
```

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala | LogCleaner.scala

```
129
130   /**
131    * 启动后台清理任务
132    */
133   def startup() {
134     info( msg = "Starting the log cleaner")
135     cleaners.foreach(_.start())
136   }
```

cleaners是多个CleanerThread集合：

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala | LogCleaner.scala

```
106
107   /* the threads */
108   private val cleaners = (0 until config.numThreads).map(new CleanerThread(_))
```

最终执行清理的是，压缩：

Log.scala | LogConfig.scala | TopicConfig.java | LogManager.scala | LogCleaner.scala

```
408
409   /**
410    * 将原来一组日志片段压缩后，腾到新建的日志片段
411    * 删除原来的一组日志片段
412    *
413    * @param log The log being cleaned
414    * @param segments The group of segments being cleaned
415    * @param map The offset map to use for cleaning segments
416    * @param deleteHorizonMs The time to retain delete tombstones
417    * @param stats Collector for cleaning statistics
418    */
419   private[log] def cleanSegments(log: Log,
420                                  segments: Seq[LogSegment],
```

4.10 Kafka源码剖析之ReplicaManager

4.10.1 副本管理器的启动和ISR的收缩和扩展

在启动KafkaServer的时候，运行KafkaServer的startup方法。在该方法中实例化ReplicaManager，并调用ReplicaManager的startup方法：

ReplicaManager.scala | KafkaServer.scala

```
258
259         /*
260          * start replica manager
261          */
262         replicaManager = createReplicaManager(isShuttingDown)
263         replicaManager.startup()
264
```

ReplicaManager的startup方法：

ReplicaManager.scala | LogManager.scala | ApiVersion.scala | Partition.scala | KafkaServer.scala | KafkaConfig.scala

```
321
322   def startup() {
323     // 启动ISR超时线程
324     // 如果ISR中的一个副本落后leader副本最多达到config.replicaLagTimeMaxMs x 1.5，就从ISR移除该副本
325     scheduler.schedule( name = "isr-expiration", maybeShrinkIsr _, period = config.replicaLagTimeMaxMs / 2, unit = TimeUnit.MILLISECONDS)
326     // 周期性地检查是否有ISR变动需要广播
327     scheduler.schedule( name = "isr-change-propagation", maybePropagateIsrChanges _, period = 2500L, unit = TimeUnit.MILLISECONDS)
328
329     val haltBrokerOnFailure = config.interBrokerProtocolVersion < KAFKA_1_0_IV0
330     // 处理日志目录异常的失败
331     logDirFailureHandler = new LogDirFailureHandler( name = "LogDirFailureHandler", haltBrokerOnFailure)
332     logDirFailureHandler.start()
333   }
```

处理ISR收缩的情况：

ReplicaManager.scala | Partition.scala | KafkaServer.scala

```
1326
1327   private def maybeShrinkIsr(): Unit = {
1328     trace( msg = "Evaluating ISR list of partitions to see which replicas can be removed from the ISR")
1329     // 使用非离线分区迭代器遍历，判断每个分区是否需要发生ISR收缩
1330     nonOfflinePartitionsIterator.foreach(_.maybeShrinkIsr(config.replicaLagTimeMaxMs))
1331   }
```

```
1   def maybeShrinkIsr(replicaMaxLagTimeMs: Long) {
2     val leaderHWIncremented = inWriteLock(leaderIsrUpdateLock) {
3       leaderReplicaIfLocal match {
4         case Some(leaderReplica) =>
5           // 获取ISR中的不同步副本
6           val outOfSyncReplicas = getOutOfSyncReplicas(leaderReplica,
replicaMaxLagTimeMs)
7           // 如果该集合不为空，则需要收缩ISR
8           if(outOfSyncReplicas.nonEmpty) {
9             // 从ISR中除去非同步副本
10            val newInSyncReplicas = inSyncReplicas -- outOfSyncReplicas
11            assert(newInSyncReplicas.nonEmpty)
```

```
            info("Shrinking ISR from %s to
%s".format(inSyncReplicas.map(_.brokerId).mkString(","),
              newInSyncReplicas.map(_.brokerId).mkString(",")))
            // 在缓存和zk中更新ISR
            updateIsr(newInSyncReplicas)
            // 标记ISR收缩事件
            replicaManager.isrShrinkRate.mark()
            // 由于ISR可能发生变化，变为1，如果ISR发生变化，需要增加HW
            maybeIncrementLeaderHW(leaderReplica)
          } else {
            false
          }
        // 如果leader不在当前broker，则什么都不做
        case None => false
      }
    }
    // 当更新HW之后，尝试完成因ISR收缩而阻塞的操作。
    if (leaderHWIncremented)
      tryCompleteDelayedRequests()
  }
```

对于在ISR集合中的副本，检查有没有需要从ISR中移除的：

两种情况需要从ISR中移除：

1. 卡主的Follower：如果副本的LEO经过maxLagMs毫秒还没有更新，则Follower卡主了，需要从ISR移除
2. 慢Follower：如果副本从maxLagMs毫秒之前到现在还没有读到leader的LEO，则Follower落后，需要从ISR移除。

ReplicaManager.scala | Partition.scala | KafkaServer.scala

```
/**
 * 此处处理两种情况：<br />
 * 1. 卡住的Follower：如果副本的LEO经过maxLagMs毫秒还没有更新，则Follower卡住了，需要从ISR移除<br />
 * 2. 慢Follower：如果副本从maxLagMs毫秒之前到现在都没有读到leader的LEO，则Follower落后，需要从ISR移除。<br />
 * 以上两种情况都需要检查lastCaughtUpTimeMs，即检查上次副本完全追上leader的时刻。<br />
 * 如果违反了上述任何一个条件，认为副本不同步，需要从ISR移除。
 * @param leaderReplica
 * @param maxLagMs
 * @return
 */
def getOutOfSyncReplicas(leaderReplica: Replica, maxLagMs: Long): Set[Replica] = {
  // candidateReplicas就是从ISR中排除leader副本的集合。
  val candidateReplicas = inSyncReplicas - leaderReplica
  // 从candidate副本中过滤出落后的副本：现在的时间减去上次同步的时间，大于masLagMs。
  val laggingReplicas = candidateReplicas.filter(r => (time.milliseconds - r.lastCaughtUpTimeMs) > maxLagMs)
  // 如果非同步副本存在，打印落后副本信息
  if (laggingReplicas.nonEmpty)
    debug("Lagging replicas are %s".format(laggingReplicas.map(_.brokerId).mkString(",")))
  // 返回落后副本集合
  laggingReplicas
}
```

处理ISR变动事件广播：

同时startup方法中周期性地调用maybePropagateIsrChanges()方法：

该函数周期性运行，检查ISR是否需要扩展。两种情况发生ISR的广播：

1. 有尚未广播的ISR变动
2. 最近5s没有发生ISR变动，或者上次ISR广播已经过去60s了。

该方法保证在ISR偶尔发生变动时，几秒之内即可将ISR变动广播出去。

避免了当发生大量ISR变更时压垮controller和其他broker。

ReplicaManager.scala | Partition.scala | KafkaServer.scala

```
264   /**
265    * 该函数周期性地运行，检查ISR是否需要扩展。下面两种情况会发生ISR的广播：<br />
266    * 1. 有尚未广播的ISR变动。<br />
267    * 2. 最近5s没有发生ISR变动，或者上次ISR广播已过去60s了。<br />
268    * 如此则保证了在ISR偶尔发生变动时，几秒之内即可将ISR变动广播出去。<br />
269    * 避免了当发生大量ISR变更时压垮controller和其他broker。
270    */
271   def maybePropagateIsrChanges() {
272     val now = System.currentTimeMillis()
273     isrChangeSet synchronized {
274       if (isrChangeSet.nonEmpty &&
275         //                                                  5000L
276         (lastIsrChangeMs.get() + ReplicaManager.IsrChangePropagationBlackOut < now ||
277           //                                                 60000L
278           lastIsrPropagationMs.get() + ReplicaManager.IsrChangePropagationInterval < now)) {
279         ReplicationUtils.propagateIsrChanges(zkUtils, isrChangeSet)
280         isrChangeSet.clear()
281         lastIsrPropagationMs.set(now)
282       }
283     }
284   }
```

处理日志目录异常的失败：

ReplicaManager.scala | LogManager.scala | ApiVersion.scala | Partition.scala | KafkaServer.scala | KafkaConfig.scala

```
195   private var logDirFailureHandler: LogDirFailureHandler = null
196
197   private class LogDirFailureHandler(name: String, haltBrokerOnDirFailure: Boolean) extends ShutdownableThread(name) {
198     override def doWork() {
199       val newOfflineLogDir = logDirFailureChannel.takeNextOfflineLogDir()
200       if (haltBrokerOnDirFailure) {
201         fatal( msg = s"Halting broker because dir $newOfflineLogDir is offline")
202         Exit.halt( statusCode = 1)
203       }
204       // 处理日志目录异常的失败
205       handleLogDirFailure(newOfflineLogDir)
206     }
207   }
```

4.10.2 follower副本如何与leader同步消息

副本管理器类：

ReplicaManager.scala | ReplicaFetcherManager.scala | ReplicaFetcherBlockingSend.scala | ReplicaFetcherThread.scala | AbstractFetcherThread.sca

```
132 }
133
134 class ReplicaManager(val config: KafkaConfig,
135                      metrics: Metrics,
136                      time: Time,
137                      val zkUtils: ZkUtils,
138                      scheduler: Scheduler,
139                      val logManager: LogManager,
140                      val isShuttingDown: AtomicBoolean,
141                      quotaManager: ReplicationQuotaManager,
142                      val brokerTopicStats: BrokerTopicStats,
143                      val metadataCache: MetadataCache,
144                      logDirFailureChannel: LogDirFailureChannel,
145                      val delayedProducePurgatory: DelayedOperationPurgatory[DelayedProduce],
146                      val delayedFetchPurgatory: DelayedOperationPurgatory[DelayedFetch],
147                      val delayedDeleteRecordsPurgatory: DelayedOperationPurgatory[DelayedDeleteRecords],
148                      threadNamePrefix: Option[String]) extends Logging with KafkaMetricsGroup {
```

副本管理器类在实例化的时候创建ReplicaFetcherManager对象，该对象是负责从leader拉取消息与leader保持同步的线程管理器：

ReplicaManager.scala | ReplicaFetcherManager.scala | ReplicaFetcherBlockingSend.scala | ReplicaFetcherThread.scala | AbstractFetcherThread.scala

```
181   private val replicaStateChangeLock = new Object
182   // 创建副本拉取管理器
183   val replicaFetcherManager = createReplicaFetcherManager(metrics, time, threadNamePrefix, quotaManager)
184   private val highWatermarkCheckPointThreadStarted = new AtomicBoolean( initialValue = false)
```

方法的具体实现：

创建负责从leader拉取消息与leader保持同步的线程管理器：

ReplicaManager.scala | ReplicaFetcherManager.scala | ReplicaFetcherBlockingSend.scala | ReplicaFetcherThread.scala | AbstractFetcherThread.scal

```
1470  /**
1471   * 创建负责从leader拉取消息与leader保持同步的线程管理器
1472   * ReplicaFetcherManager
1473   */
1474  protected def createReplicaFetcherManager(metrics: Metrics,
1475                                            time: Time,
1476                                            threadNamePrefix: Option[String],
1477                                            quotaManager: ReplicationQuotaManager) = {
1478    new ReplicaFetcherManager(config, replicaManager = this, metrics, time, threadNamePrefix, quotaManager)
1479  }
```

```scala
/**
 * 负责从leader拉取消息与leader保持同步的线程管理器
 */
class ReplicaFetcherManager(brokerConfig: KafkaConfig, protected val replicaManager: ReplicaManager, metrics: Metrics,
                            time: Time, threadNamePrefix: Option[String] = None, quotaManager: ReplicationQuotaManager)
      extends AbstractFetcherManager("ReplicaFetcherManager on broker " + brokerConfig.brokerId,
        "Replica", brokerConfig.numReplicaFetchers) {

```

副本拉取管理器中实现了createFetcherThread方法，该方法返回ReplicaFetcherThread对象：

```scala
  /**
   * 创建拉取消息的线程
   * @param fetcherId
   * @param sourceBroker
   * @return
   */
  override def createFetcherThread(fetcherId: Int, sourceBroker: BrokerEndPoint): AbstractFetcherThread = {
    val prefix = threadNamePrefix.map(tp => s"${tp}:").getOrElse("")
    val threadName = s"${prefix}ReplicaFetcherThread-$fetcherId-${sourceBroker.id}"
    new ReplicaFetcherThread(threadName, fetcherId, sourceBroker, brokerConfig, replicaManager, metrics, time, quotaManager)
  }
```

ReplicaFetcherThread线程负责从Leader副本拉取消息进行同步。

```scala
 * 负责从leader副本拉取消息进行同步的线程类
 */
class ReplicaFetcherThread(name: String,
                           fetcherId: Int,
                           sourceBroker: BrokerEndPoint,
                           brokerConfig: KafkaConfig,
                           replicaMgr: ReplicaManager,
                           metrics: Metrics,
                           time: Time,
                           quota: ReplicationQuotaManager,
                           leaderEndpointBlockingSend: Option[BlockingSend] = None)
  extends AbstractFetcherThread(name = name,
                                clientId = name,
                                sourceBroker = sourceBroker,
                                fetchBackOffMs = brokerConfig.replicaFetchBackoffMs,
                                isInterruptible = false,
                                includeLogTruncation = true) {

```

AbstractFetcherManager中的addFetcherForPartitions方法中的嵌套方法addAndStartFetcherThread创建并启动拉取线程：

而其中用到的createFetcherThread方法便是在AbstractFetcherManager的实现类ReplicaFetcherManager中实现的。

```scala
// to be defined in subclass to create a specific fetcher
def createFetcherThread(fetcherId: Int, sourceBroker: BrokerEndPoint): AbstractFetcherThread

def addFetcherForPartitions(partitionAndOffsets: Map[TopicPartition, BrokerAndInitialOffset]) {
  mapLock synchronized {
    val partitionsPerFetcher = partitionAndOffsets.groupBy { case(topicPartition, brokerAndInitialOffset) =>
      BrokerAndFetcherId(brokerAndInitialOffset.broker, getFetcherId(topicPartition.topic, topicPartition.partition))}

    def addAndStartFetcherThread(brokerAndFetcherId: BrokerAndFetcherId, brokerIdAndFetcherId: BrokerIdAndFetcherId) {
      // 创建拉取线程
      val fetcherThread = createFetcherThread(brokerAndFetcherId.fetcherId, brokerAndFetcherId.broker)
      fetcherThreadMap.put(brokerIdAndFetcherId, fetcherThread)
      // 启动拉取线程
      fetcherThread.start
    }
```

抽象类AbstractFetcherThread从同一个远程broker上为当前broker上的多个分区follower副本拉取消息。

即，在远程同一个broker上有多个leader副本的follower副本在当前broker上。

```scala
/**
 *  Abstract class for fetching data from multiple partitions from the same broker.
 */
abstract class AbstractFetcherThread(name: String,
                                     clientId: String,
                                     val sourceBroker: BrokerEndPoint,
                                     fetchBackOffMs: Int = 0,
                                     isInterruptible: Boolean = true,
                                     includeLogTruncation: Boolean
                                    )
  extends ShutdownableThread(name, isInterruptible) {
```

ReplicaFetcherThread的start方法实际上就是AbstractFetcherThread中的start方法。

在AbstractFetcherThread中没有start方法，在其父类ShutdownableThread也没有start方法：

```scala
import ...

abstract class ShutdownableThread(val name: String, val isInterruptible: Boolean = true)
        extends Thread(name) with Logging {
  this.setDaemon(false)
```

但是ShutdownableThread继承自Thread，Thread中有start方法，并且start方法要调用run方法，在ShutdownableThread中有run方法：

该run方法重复调用doWork方法进行数据的拉取。

doWork方法是抽象方法，没有实现。其实现在ShutdownableThread的实现类AbstractFetcherThread中：

```scala
// 线程的run方法调用该方法
override def doWork() {
  maybeTruncate()
  val fetchRequest = inLock(partitionMapLock) {
    // 创建fetchRequest
    val ResultWithPartitions(fetchRequest, partitionsWithError) = buildFetchRequest(states)
    if (fetchRequest.isEmpty) {
      trace( msg = s"There are no active partitions. Back off for $fetchBackOffMs ms before sending a fetch request")
      partitionMapCond.await(fetchBackOffMs, TimeUnit.MILLISECONDS)
    }
    handlePartitionsWithErrors(partitionsWithError)
    fetchRequest
  }
  if (!fetchRequest.isEmpty)
    // 处理fetchRequest请求的发送和结果处理
    processFetchRequest(fetchRequest)
}
```

上图中的doWork方法会反复调用，上图中的方法创建拉取请求对象，然后调用processFetchRequest方法进行请求的发送和结果的处理。

```scala
/**
 * 处理fetchRequest请求的发送和结果处理
 * @param fetchRequest 拉取数据的请求
 */
private def processFetchRequest(fetchRequest: REQ) {
  val partitionsWithError = mutable.Set[TopicPartition]()

  var responseData: Seq[(TopicPartition, PD)] = Seq.empty
```

```scala
try {
  trace( msg = s"Issuing fetch to broker ${sourceBroker.id}, request: $fetchRequest")
  // 从leader副本拉取消息
  responseData = fetch(fetchRequest)
} catch {
```

fetch方法的实现在AbstractFetcherThread的子类ReplicaFetcherThread中：

```scala
 * 从leader副本拉取消息
 * @param fetchRequest
 * @return
 */
protected def fetch(fetchRequest: FetchRequest): Seq[(TopicPartition, PartitionData)] = {
  // 发送请求到leader分区拉取消息，返回结果
  val clientResponse = leaderEndpoint.sendRequest(fetchRequest.underlying)
  // 从结果中获取拉取的结果
  val fetchResponse = clientResponse.responseBody.asInstanceOf[FetchResponse]
  // 遍历拉取的数据，转换为返回值的格式：
  // 二元组的Seq集合
  // Seq[(TopicPartition, PartitionData)]
  fetchResponse.responseData.asScala.toSeq.map { case (key, value) =>
    key -> new PartitionData(value)
  }
}
```

sendRequest方法在ReplicaFetcherBlockingSend中：

```scala
 * 从leader副本拉取消息
 * @param fetchRequest
 * @return
 */
protected def fetch(fetchRequest: FetchRequest): Seq[(TopicPartition, PartitionData)] = {
  // 发送请求到leader分区拉取消息，返回结果
  val clientResponse = leaderEndpoint.sendRequest(fetchRequest.underlying)
  // 从结果中获取拉取的结果
  val fetchResponse = clientResponse
  // 遍历拉取的数据，转换为返回值的格
  // 二元组的Seq集合
```

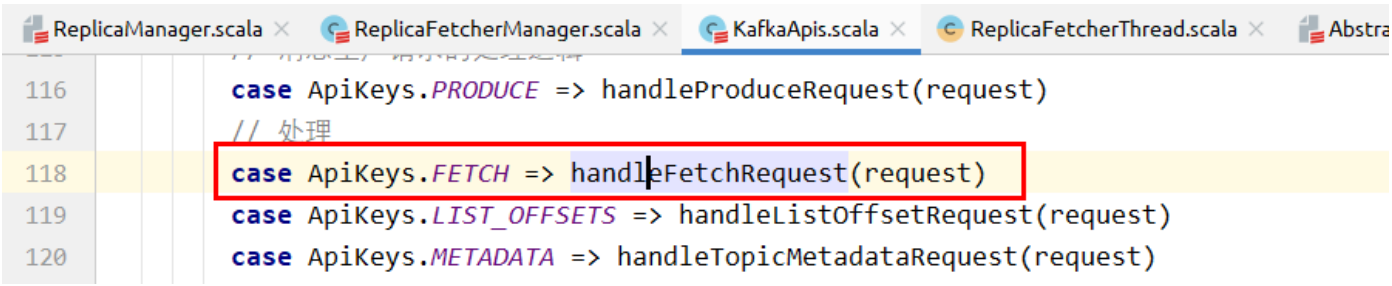
Choose Implementation of **BlockingSend.sendReques**

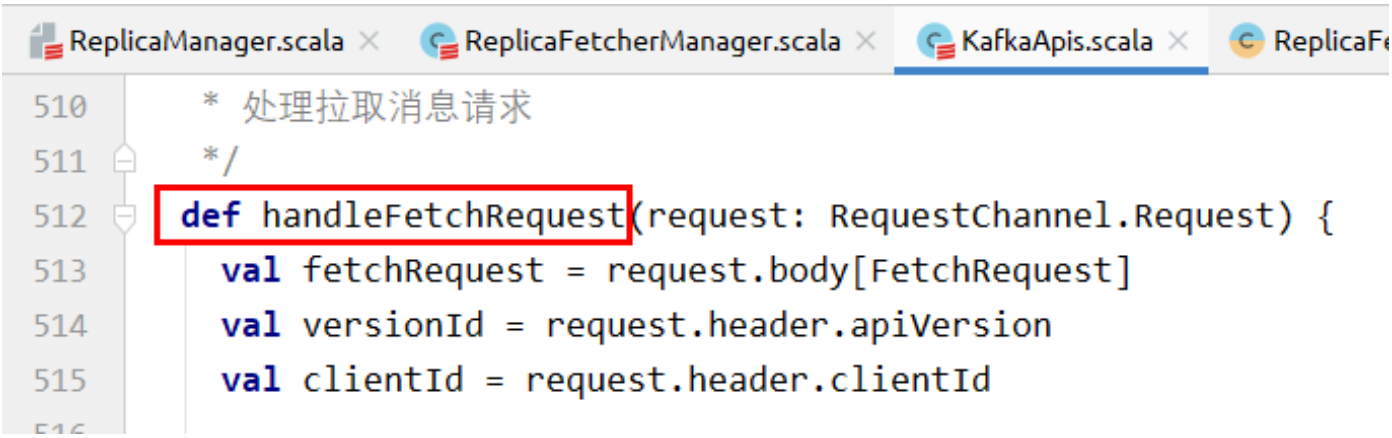
- BlockingSend (kafka.server)
- ReplicaFetcherBlockingSend (kafka.server)
- ReplicaFetcherMockBlockingSend (kafka.server.epoch.uti

通过NetworkClientUtils发送请求，并等待请求的响应：

```scala
}

override def sendRequest(requestBuilder: Builder[_ <: AbstractRequest]): ClientResponse =  {
  try {
    if (!NetworkClientUtils.awaitReady(networkClient, sourceNode, time, socketTimeout))
      throw new SocketTimeoutException(s"Failed to connect within $socketTimeout ms")
    else {
      val clientRequest = networkClient.newClientRequest(sourceBroker.id.toString, requestBuilder,
        time.milliseconds(), expectResponse = true)
      // 返回值就是ClientResponse，作为sendRequest方法的返回值
      NetworkClientUtils.sendAndReceive(networkClient, clientRequest, time)
    }
  }
```

KafkaApis对Fetch的处理：





该方法中，Leader从本地日志读取数据，返回：

总结：

当KafkaServer启动的时候，会实例化副本管理器

```
258
259         /*
260          * 启动副本管理器
261          */
262         replicaManager = createReplicaManager(isShuttingDown)
263         replicaManager.startup()
264
```

副本管理器实例化的时候会实例化副本拉取器管理器：

```
182   // 创建副本拉取管理器
183   val replicaFetcherManager = createReplicaFetcherManager(metrics, time, threadNamePrefix, quotaManager)
```

副本管理器中有实现createFetcherThread方法，创建副本拉取器对象

```
35    * @param sourceBroker
36    * @return
37    */
38   override def createFetcherThread(fetcherId: Int, sourceBroker: BrokerEndPoint): AbstractFetcherThread = {
39     val prefix = threadNamePrefix.map(tp => s"${tp}:").getOrElse("")
40     val threadName = s"${prefix}ReplicaFetcherThread-$fetcherId-${sourceBroker.id}"
41     new ReplicaFetcherThread(threadName, fetcherId, sourceBroker, brokerConfig, replicaManager, metrics, time, quotaManager)
42   }
43
```

拉取线程启动起来之后不断地从leader副本所在的broker拉取消息，以便Follower与leader保持消息的同步。

4.11 Kafka源码剖析之OffsetManager

消费者如何提交偏移量?

自动提交

手动提交

同步提交

异步提交

客户端提交偏移量，交给KafkaApis的handle方法，handle方法使用模式匹配，调用handleOffsetCommitRequest方法进行处理：

KafkaApis.scala

```
107     * Top-level method that handles all requests and multiplexes to the right api
108     * 处理所有请求的顶级方法，使用模式匹配，交给具体的api来处理
109     */
110    def handle(request: RequestChannel.Request) {
111      try {
112        trace( msg = s"Handling request:${request.requestDesc( details = true)} from connection ${
113          s"securityProtocol:${request.context.securityProtocol},principal:${request.context.p
114        request.header.apiKey match {
115          // 消息生产请求的处理逻辑
116          case ApiKeys.PRODUCE => handleProduceRequest(request)
117          // 处理
118          case ApiKeys.FETCH => handleFetchRequest(request)
119          case ApiKeys.LIST_OFFSETS => handleListOffsetRequest(request)
120          case ApiKeys.METADATA => handleTopicMetadataRequest(request)
121          case ApiKeys.LEADER_AND_ISR => handleLeaderAndIsrRequest(request)
122          case ApiKeys.STOP_REPLICA => handleStopReplicaRequest(request)
123          case ApiKeys.UPDATE_METADATA => handleUpdateMetadataRequest(request)
124          case ApiKeys.CONTROLLED_SHUTDOWN => handleControlledShutdownRequest(request)
125          // 如果是提交偏移量：
126          case ApiKeys.OFFSET_COMMIT => handleOffsetCommitRequest(request)
127          case ApiKeys.OFFSET_FETCH => handleOffsetFetchRequest(request)
128          case ApiKeys.FIND_COORDINATOR => handleFindCoordinatorRequest(request)
```

handleOffsetCommitRequest的实现：

KafkaApis.scala

```
277     * 处理提交偏移量的请求
278     */
279    def handleOffsetCommitRequest(request: RequestChannel.Request) {
280      val header = request.header
281      val offsetCommitRequest = request.body[OffsetCommitRequest]
282
```

如果apiVersion的值是0，则交给zookeeper保存偏移量信息：

KafkaApis.scala

```
322        sendResponseCallback(Map.empty)
323      else if (header.apiVersion == 0) {
324        // 如果apiVersion的值是0，则在zk中保存偏移量信息。
325        val responseInfo = authorizedTopicRequestInfo.map {
326          case (topicPartition, partitionData) =>
327            val topicDirs = new ZKGroupTopicDirs(offsetCommitRequest.groupId, topicPartition.topic)
328            try {
329              if (partitionData.metadata != null && partitionData.metadata.length > config.offsetMetada
330                (topicPartition, Errors.OFFSET_METADATA_TOO_LARGE)
331              else {
332                zkUtils.updatePersistentPath( path = s"${topicDirs.consumerOffsetDir}/${topicPartition.pa
333                (topicPartition, Errors.NONE)
334              }
335            } catch {
```

否则调用组协调器负责处理偏移量提交请求：

KafkaApis.scala

```
372            }
373
374            // 调用组协调器负责偏移量的提交
375            groupCoordinator.handleCommitOffsets(
376              offsetCommitRequest.groupId,
377              offsetCommitRequest.memberId,
378              offsetCommitRequest.generationId,
379              partitionData,
380              sendResponseCallback)
381          }
382        }
383      }
```

handleCommitOffsets的实现：

首先根据groupId查找消费组元数据。

如果没有找到消费组元数据，则要么该消费组不依赖Kafka进行消费组管理，允许提交；要么提交的偏移量信息是消费组再平衡之前的偏移量，旧请求，拒绝。

正常情况就是最后的分支：

找到了消费组元数据，调用doCommitOffsets处理。偏移量提交的请求。

KafkaApis.scala | GroupCoordinator.scala

```
484     * @param responseCallback
485     */
486    def handleCommitOffsets(groupId: String,
487                            memberId: String,
488                            generationId: Int,
489                            offsetMetadata: immutable.Map[TopicPartition, OffsetAndMetadata],
490                            responseCallback: immutable.Map[TopicPartition, Errors] => Unit) {
491      validateGroup(groupId) match {
492        case Some(error) => responseCallback(offsetMetadata.mapValues(_ => error))
493        case None =>
494          groupManager.getGroup(groupId) match {
495            case None =>
496              if (generationId < 0) {
497                // the group is not relying on Kafka for group management, so allow the commit
498                // 如果没有找到消费组元数据，表示当前消费组不依赖Kafka管理消费组，允许提交偏移量
499                val group = groupManager.addGroup(new GroupMetadata(groupId, initialState = Empty))
500                doCommitOffsets(group, memberId, generationId, NO_PRODUCER_ID, NO_PRODUCER_EPOCH,
501                  offsetMetadata, responseCallback)
502              } else {
503                // 提交的是再平衡之前消费组提交的偏移量，拒绝提交。
504                responseCallback(offsetMetadata.mapValues(_ => Errors.ILLEGAL_GENERATION))
505              }
506
507            case Some(group) =>
508              doCommitOffsets(group, memberId, generationId, NO_PRODUCER_ID, NO_PRODUCER_EPOCH,
509                offsetMetadata, responseCallback)
510          }
511      }
```

doCommitOffsets的实现：

该方法判断消费组的状态：

1. 如果是Dead，则响应错误信息。
2. 如果消费组还在等待消费者同步，则响应错误信息
3. 如果消费组中没有这个消费者，则响应错误信息
4. 如果请求中的纪元数字和消费组当前纪元数字不符，则响应错误信息
5. 如果仅使用Kafka存储偏移量，而不需要管理，则直接保存偏移量
6. 正常情况下，找到了消费组，消费组中有这个消费者，同时消费组工作正常，则保存偏移量信息

KafkaApis.scala | GroupCoordinator.scala

```scala
  */
private def doCommitOffsets(group: GroupMetadata,
                            memberId: String,
                            generationId: Int,
                            producerId: Long,
                            producerEpoch: Short,
                            offsetMetadata: immutable.Map[TopicPartition, OffsetAndMetadata],
                            responseCallback: immutable.Map[TopicPartition, Errors] => Unit) {
  group.inLock {
    if (group.is(Dead)) {
      responseCallback(offsetMetadata.mapValues(_ => Errors.UNKNOWN_MEMBER_ID))
    } else if ((generationId < 0 && group.is(Empty)) || (producerId != NO_PRODUCER_ID)) {
      // 要么当前消费组仅使用Kafka存储偏移量，而不用管理消费组
      // 要么是事务的偏移量提交，不需要验证消费组成员关系和消费组的纪元，直接存储即可。
      groupManager.storeOffsets(group, memberId, offsetMetadata, responseCallback, producerId, producerEpoch)
    } else if (group.is(AwaitingSync)) {
      // 消费组还在等待同步
      responseCallback(offsetMetadata.mapValues(_ => Errors.REBALANCE_IN_PROGRESS))
    } else if (!group.has(memberId)) {
      // 消费组中没有这个消费者
      responseCallback(offsetMetadata.mapValues(_ => Errors.UNKNOWN_MEMBER_ID))
    } else if (generationId != group.generationId) {
      // 消费组的纪元数字与现在的纪元数字不符
      responseCallback(offsetMetadata.mapValues(_ => Errors.ILLEGAL_GENERATION))
    } else {
      val member = group.get(memberId)
      completeAndScheduleNextHeartbeatExpiration(group, member)
      // 存偏移量信息
      groupManager.storeOffsets(group, memberId, offsetMetadata, responseCallback)
    }
```

storeOffsets方法的实现：

KafkaApis.scala | GroupCoordinator.scala | GroupMetadataManager.scala

```scala
  /**
   * 将偏移量信息追加到分区副本的日志中，然后将信息插入到缓存中
   */
  def storeOffsets(group: GroupMetadata,
                   consumerId: String,
                   offsetMetadata: immutable.Map[TopicPartition, OffsetAndMetadata],
                   responseCallback: immutable.Map[TopicPartition, Errors] => Unit,
                   producerId: Long = RecordBatch.NO_PRODUCER_ID,
                   producerEpoch: Short = RecordBatch.NO_PRODUCER_EPOCH): Unit = {
    // first filter out partitions with offset metadata size exceeding limit
```

需要先计算当前消费组的偏移量需要提交到`__consumer_offsets`主题的哪个分区中。

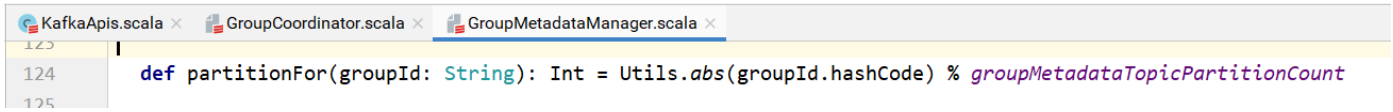
KafkaApis.scala | GroupCoordinator.scala | GroupMetadataManager.scala

计算当前消费组的偏移量应该保存在__consumer_offsets主题的哪个分区中

```scala
    }
    // 计算当前消费组的偏移量应该放到__consumer_offsets的哪个分区
    val offsetTopicPartition = new TopicPartition(Topic.GROUP_METADATA_TOPIC_NAME, partitionFor(group.groupId))
    // 分配缓冲区
    val buffer = ByteBuffer.allocate(AbstractRecords.estimateSizeInBytes(magicValue, compressionType, records.asJava))

    if (isTxnOffsetCommit && magicValue < RecordBatch.MAGIC_VALUE_V2)
      throw Errors.UNSUPPORTED_FOR_MESSAGE_FORMAT.exception("Attempting to make a transaction offset commit with an inval
```

将消息追加到`__consumer_offsets`主题的指定分区中：

其中计算`__consumer_offsets`分区的实现：

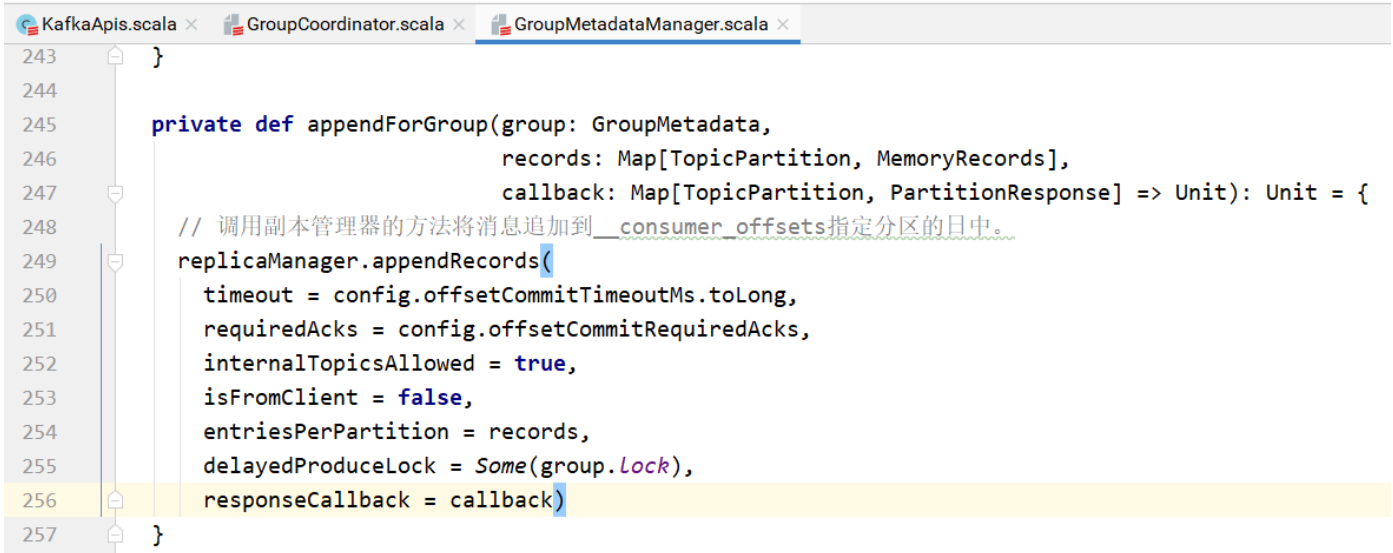


上图中的函数，计算方式如下：

获取消费组ID的散列值，取绝对值，然后将此绝对值对`__consumer_offsets`主题分区个数取模得到。

appendForGroup方法的实现：

调用副本管理器的方法将消息追加到`__consumer_offsets`主题的指定分区日志中。



如果偏移量消息追加成功，则调用callback响应客户端：

KafkaApis.scala | GroupCoordinator.scala | GroupMetadataManager.scala

```scala
312
313         // 当消息成功追加到日志中后，调用该函数，将偏移量插入缓冲区
314         def putCacheCallback(responseStatus: Map[TopicPartition, PartitionResponse]) {
315           // the append response should only contain the topics partition
316           if (responseStatus.size != 1 || !responseStatus.contains(offsetTopicPartition))
317             throw new IllegalStateException("Append status %s should only have one partition %s"
318               .format(responseStatus, offsetTopicPartition))
319
```

缓存偏移量信息：

KafkaApis.scala | GroupCoordinator.scala | GroupMetadataManager.scala | GroupMetadata.scala

```scala
323           val responseError = group.inLock {
324             if (status.error == Errors.NONE) {
325               if (!group.is(Dead)) {
326                 filteredOffsetMetadata.foreach { case (topicPartition, offsetAndMetadata) =>
327                   if (isTxnOffsetCommit)
328                     group.onTxnOffsetCommitAppend(producerId, topicPartition, CommitRecordMetadataAndOffset(Some(status.baseOffset), offsetAndMetadata))
329                   else
330                     group.onOffsetCommitAppend(topicPartition, CommitRecordMetadataAndOffset(Some(status.baseOffset), offsetAndMetadata))
331                 }
332               }
```

缓存偏移量信息

具体实现：

KafkaApis.scala | GroupCoordinator.scala | GroupMetadataManager.scala | GroupMetadata.scala

```scala
337
338   def onOffsetCommitAppend(topicPartition: TopicPartition, offsetWithCommitRecordMetadata: CommitRecordMetadataAndOffset) {
339     if (pendingOffsetCommits.contains(topicPartition)) {
340       if (offsetWithCommitRecordMetadata.appendedBatchOffset.isEmpty)
341         throw new IllegalStateException("Cannot complete offset commit write without providing the metadata of the record " +
342           "in the log.")
343
344       // offsets是消费组元数据的本地缓存偏移量map集合
345       // private val offsets = new mutable.HashMap[TopicPartition, CommitRecordMetadataAndOffset]
346       // 如果offsets中不包含该主题分区信息或者offsets中该主题分区的数据是旧的
347       if (!offsets.contains(topicPartition) || offsets(topicPartition).olderThan(offsetWithCommitRecordMetadata))
348         // 将主题分区和元数据映射put到offsets中
349         offsets.put(topicPartition, offsetWithCommitRecordMetadata)
350       }
351
352       pendingOffsetCommits.get(topicPartition) match {
353         case Some(stagedOffset) if offsetWithCommitRecordMetadata.offsetAndMetadata == stagedOffset =>
354           pendingOffsetCommits.remove(topicPartition)
355         case _ =>
356           // The pendingOffsetCommits for this partition could be empty if the topic was deleted, in which case
357           // its entries would be removed from the cache by the `removeOffsets` method.
```

将消费的偏移量缓存到offsets中
offsets是当前消费组的元数据对象

KafkaApis.scala | GroupCoordinator.scala | GroupMetadataManager.scala

```scala
375             else
376               (topicPartition, Errors.OFFSET_METADATA_TOO_LARGE)
377           }
378
379           // 触发回调逻辑
380           responseCallback(commitStatus)
381         }
```

responseCallback最终是KafkaApis中的308行（有可能不是，因为我加注释了，差不多这么多行）：

该函数将消费者提交的偏移量追加到日志中并添加到消费组缓存中之后，返回结果给消费者客户端。

KafkaApis.scala GroupCoordinator.scala GroupMetadataManager.scala

```
305     val authorizedTopicRequestInfo = authorizedTopicRequestInfoBldr.result()
306
307     // 负责发送偏移量提交请求响应的回调函数。内嵌函数
308     def sendResponseCallback(commitStatus: immutable.Map[TopicPartition, Errors]) {
309       val combinedCommitStatus = commitStatus ++ unauthorizedTopicErrors ++ nonExistingTopicErrors
310       if (isDebugEnabled)
311         combinedCommitStatus.foreach { case (topicPartition, error) =>
312           if (error != Errors.NONE) {
313             debug( msg = s"Offset commit request with correlation id ${header.correlationId} from client ${header.clientId} "
314               s"on partition $topicPartition failed due to ${error.exceptionName}")
315           }
316         }
317       sendResponseMaybeThrottle(request, requestThrottleMs =>
318         new OffsetCommitResponse(requestThrottleMs, combinedCommitStatus.asJava))
319     }
320
```

消费者提交偏移量：KafkaApis，KafkaApis->GroupCoordinator的方法-> GroupMetadata

不仅需要将消费组的偏移量提交到日志中，还需要在内存维护该偏移量信息。

其实对于消费者，获取结果后，也需要在消费者客户端解析该响应，将消费者的偏移量缓存到消费者客户端：

消费者客户端消费消息的方法：KafkaConsumer.poll(1_000);

调用poll方法拉取消息：

该方法调用pollOnce进行消息的拉取：

KafkaConsumer.java ConsumerCoordinator.java WorkerSinkTask.java AbstractCoordinator.java

```
1103     public ConsumerRecords<K, V> poll(long timeout) {
1104         acquireAndEnsureOpen();
1105         try {
1106             if (timeout < 0)
1107                 throw new IllegalArgumentException("Timeout must not be negative");
1108
1109             if (this.subscriptions.hasNoSubscriptionOrUserAssignment())
1110                 throw new IllegalStateException("Consumer is not subscribed to any topics or assigne
1111
1112             // poll for new data until the timeout expires
1113             long start = time.milliseconds();
1114             long remaining = timeout;
1115             do {
1116                 Map<TopicPartition, List<ConsumerRecord<K, V>>> records = pollOnce(remaining);
1117                 if (!records.isEmpty()) {
1118                     // before returning the fetched records, we can send off the next round of fetch
```

pollOnce方法会调用coordinator的poll方法周期性地提交偏移量：

KafkaConsumer.java | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
1149    private Map<TopicPartition, List<ConsumerRecord<K, V>>> pollOnce(long timeout) {
1150        client.maybeTriggerWakeup();
1151        // Poll for coordinator events.
1152        // This ensures that the coordinator is known
1153        // and that the consumer has joined the group (if it is using group management).
1154        // This also handles periodic offset commits if they are enabled.
1155        // 负责周期性的偏移量提交
1156        coordinator.poll(time.milliseconds(), timeout);
1157
```

其中poll方法的实现：

KafkaConsumer.java | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
273         * 轮询协调器事件。保证协调器已知，保证消费者加入消费组（如果使用消费组）。
274         * 同时，如果启用了自动提交偏移量，则负责周期性地提交偏移量
275         *
276         * @param now current time in milliseconds
277         */
278        public void poll(long now, long remainingMs) {
279            // 调用偏移量提交已完成的回调
280            invokeCompletedOffsetCommitCallbacks();
281
282            // 如果是订阅主题，自动分配的分区，则进行如下处理：
283            if (subscriptions.partitionsAutoAssigned()) {
284                if (coordinatorUnknown()) {
285                    ensureCoordinatorReady();
286                    now = time.milliseconds();
```

poll方法中，最后会判断是否需要自动提交偏移量：

KafkaConsumer.java | GroupMetadata.scala | ConsumerCoordinator.java

```
319
320                 maybeAutoCommitOffsetsAsync(now);
321             }
```

KafkaConsumer.java | GroupMetadata.scala | ConsumerCoordinator.java | WorkerSinkTask.java | Abst

```
637     /**
638      * 如果自动提交偏移量设置为true，则异步提交消费者偏移量
639      * @param now
640      */
641     private void maybeAutoCommitOffsetsAsync(long now) {
642         if (autoCommitEnabled) {
643             if (coordinatorUnknown()) {
644                 this.nextAutoCommitDeadline = now + retryBackoffMs;
645             } else if (now >= nextAutoCommitDeadline) {
646                 // 如果当前时间达到了提交偏移量的时间，则异步提交偏移量
647                 this.nextAutoCommitDeadline = now + autoCommitIntervalMs;
648                 // 异步自动提交
649                 doAutoCommitOffsetsAsync();
650             }
651         }
652     }
```

KafkaConsumer.java | GroupMetadata.scala | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
660     * 异步自动提交偏移量
661     */
662    private void doAutoCommitOffsetsAsync() {
663        Map<TopicPartition, OffsetAndMetadata> allConsumedOffsets = subscriptions.allConsumed();
664        log.debug("Sending asynchronous auto-commit of offsets {}", allConsumedOffsets);
665
666        // 异步提交偏移量
667        commitOffsetsAsync(allConsumedOffsets, (offsets, exception) -> {
670            if (exception != null) {
671                log.warn("Asynchronous auto-commit of offsets {} failed: {}", offsets, except
672                if (exception instanceof RetriableException)
673                    nextAutoCommitDeadline = Math.min(time.milliseconds() + retryBackoffMs, n
674            } else {
675                log.debug("Completed asynchronous auto-commit of offsets {}", offsets);
676            }
677        });
679    }
```

KafkaConsumer.java | GroupMetadata.scala | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoor

```
516
517    public void commitOffsetsAsync(final Map<TopicPartition, OffsetAndMetadata> o
518        invokeCompletedOffsetCommitCallbacks();
519
520        if (!coordinatorUnknown()) {
521            // 如果消费组已知，则异步提交
522            doCommitOffsetsAsync(offsets, callback);
523        } else {
```

KafkaConsumer.java | GroupMetadata.scala | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
554         * 真正异步提交offsets
555         * @param offsets
556         * @param callback
557         */
558        private void doCommitOffsetsAsync(final Map<TopicPartition, OffsetAndMetadata> offsets, final Offse
559            this.subscriptions.needRefreshCommits();
560            // 发送提交请求
561            RequestFuture<Void> future = sendOffsetCommitRequest(offsets);
562            final OffsetCommitCallback cb = callback == null ? defaultOffsetCommitCallback : callback;
563            future.addListener(new RequestFutureListener<Void>() {
564                @Override
565                public void onSuccess(Void value) {
566                    if (interceptors != null)
567                        interceptors.onCommit(offsets);
568
569                    completedOffsetCommits.add(new OffsetCommitCompletion(cb, offsets, exception: null));
570                }
```

invokeCompletedOffsetCommitCallbacks方法用于轮询偏移量提交后broker端的响应信息：

KafkaConsumer.java | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
506
507        // visible for testing
508        void invokeCompletedOffsetCommitCallbacks() {
509            while (true) {
510                OffsetCommitCompletion completion = completedOffsetCommits.poll();
511                if (completion == null)
512                    break;
513                completion.invoke();
514            }
515        }
```

该方法轮询，等待提交偏移量的响应结果
如果得到结果，就调用completion的invoke
方法进行处理

KafkaConsumer.java | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
950
951    private static class OffsetCommitCompletion {
952        private final OffsetCommitCallback callback;
953        private final Map<TopicPartition, OffsetAndMetadata> offsets;
954        private final Exception exception;
955
956        private OffsetCommitCompletion(OffsetCommitCallback callback, Map<TopicPartition, Offse
957            this.callback = callback;
958            this.offsets = offsets;
959            this.exception = exception;
960        }
961
962        public void invoke() {
963            if (callback != null)
964                callback.onComplete(offsets, exception);
965        }
966    }
```

KafkaConsumer.java | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
322
323     private void doCommitAsync(Map<TopicPartition, OffsetAndMetadata> offsets, final int seqno) {
324         log.info("{} Committing offsets asynchronously using sequence number {}: {}", this, seqno, offsets);
325         OffsetCommitCallback cb = new OffsetCommitCallback() {
326             @Override
327             public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception error) {
328                 onCommitCompleted(error, seqno, offsets);
329             }
330         };
331         consumer.commitAsync(offsets, cb);
332     }
```

回调，当提交偏移量收到响应之后，调用该方法进行处理

onCommitCompleted的实现：

KafkaConsumer.java | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
238     private void onCommitCompleted(Throwable error, long seqno, Map<TopicPartition, OffsetAndMetadata> committed
239         if (commitSeqno != seqno) {
240             log.debug("{} Received out of order commit callback for sequence number {}, but most recent sequence
241                     this, seqno, commitSeqno);
242             sinkTaskMetricsGroup.recordOffsetCommitSkip();
243         } else {
244             long durationMillis = time.milliseconds() - commitStarted;
245             if (error != null) {
246                 log.error("{} Commit of offsets threw an unexpected exception for sequence number {}: {}",
247                         this, seqno, committedOffsets, error);
248                 commitFailures++;
249                 recordCommitFailure(durationMillis, error);
250             } else {
251                 log.debug("{} Finished offset commit successfully in {} ms for sequence number {}: {}",
252                         this, durationMillis, seqno, committedOffsets);
253                 if (committedOffsets != null) {
254                     log.debug("{} Setting last committed offsets to {}", this, committedOffsets);
255                     lastCommittedOffsets = committedOffsets;
256                     sinkTaskMetricsGroup.recordCommittedOffsets(committedOffsets);
257                 }
258                 commitFailures = 0;
259                 recordCommitSuccess(durationMillis);
260             }
261             committing = false;
262         }
```

lastCommittedOffsets为：

KafkaConsumer.java | ConsumerCoordinator.java | WorkerSinkTask.java | AbstractCoordinator.java

```
74      private WorkerSinkTaskContext context;
75      private final List<SinkRecord> messageBatch;
76      private Map<TopicPartition, OffsetAndMetadata> lastCommittedOffsets;
77      private Map<TopicPartition, OffsetAndMetadata> currentOffsets;
78      private final Map<TopicPartition, OffsetAndMetadata> origOffsets;
```

KafkaConsumer -> Broker -> KafkaApis -handle-> GroupCoordinator -> GroupMetadataManager -> GroupMetadata -> ReplicaManager -> log-> KafkaConsumer -> lastCommittedOffsets集合。

在Kafka 1.0.2之前的版本中有一个OffsetManager负责偏移量的处理。

OffsetManager主要提供对offset的保存和读取，kafka管理topic的偏移量有2种方式：

1. zookeeper，即把偏移量提交至zk上；
2. kafka，即把偏移量提交至kafka内部，主要由offsets.storage参数决定。1.0.2版本中默认是kafka。也就是说如果配置offsets.storage= kafka，则kafka会把这种offsetcommit请求转变为一种Producer，保存至topic为`__consumer_offsets`的log里面。

```
class OffsetManager(val config: OffsetManagerConfig,
                    replicaManager: ReplicaManager,
                    zkClient: ZkClient,
                    scheduler: Scheduler) extends Logging with KafkaMetricsGroup {

  //通过offsetsCache提供对GroupTopicPartition的查询
  private val offsetsCache = new Pool[GroupTopicPartition, OffsetAndMetadata]
  //把过时的偏移量刷入磁盘，因为这些偏移量长时间没有被更新，意味着消费者可能不再消费了，也就不需要了，因此刷入到磁盘
  scheduler.schedule(name = "offsets-cache-compactor",
                     fun = compact,
                     period = config.offsetsRetentionCheckIntervalMs,
                     unit = TimeUnit.MILLISECONDS)
```

主要完成2件事情：

```
1. 提供对topic偏移量的查询
2. 将偏移量消息刷入`__consumer_offsets`主题的log中。
```

4.12 Kafka源码剖析之KafkaApis

当启动KafkaServer的时候，在其startup方法中实例化了KafkaApi，并赋值给KafkaRequestHandlerPool用于执行具体的请求处理逻辑：

`KafkaApi`主构造器参数：

KafkaApis.scala

```scala
*/
class KafkaApis(val requestChannel: RequestChannel,  // 请求的channel
                val replicaManager: ReplicaManager,  // 副本管理器
                val adminManager: AdminManager,      // 管理员管理器
                val groupCoordinator: GroupCoordinator,  // 组协调器
                val txnCoordinator: TransactionCoordinator,  // 事务协调器
                val controller: KafkaController,      // 集群控制器
                val zkUtils: ZkUtils,                 // zk操作对象
                val brokerId: Int,
                val config: KafkaConfig,
                val metadataCache: MetadataCache,
                val metrics: Metrics,
                val authorizer: Option[Authorizer],
                val quotas: QuotaManagers,
                brokerTopicStats: BrokerTopicStats,
                val clusterId: String,
                time: Time) extends Logging {
```

各种请求的处理逻辑入口：

KafkaApis.scala KafkaServer.scala

```scala

  /**
   * Top-level method that handles all requests and multiplexes to the right api
   * 处理所有请求的顶级方法，使用模式匹配，交给具体的api来处理
   */
  def handle(request: RequestChannel.Request) {
    try {
```

使用模式匹配：

```scala
        case ApiKeys.PRODUCE => handleProduceRequest(request)
        case ApiKeys.FETCH => handleFetchRequest(request)
        case ApiKeys.LIST_OFFSETS => handleListOffsetRequest(request)
        case ApiKeys.METADATA => handleTopicMetadataRequest(request)
        case ApiKeys.LEADER_AND_ISR => handleLeaderAndIsrRequest(request)
        case ApiKeys.STOP_REPLICA => handleStopReplicaRequest(request)
        case ApiKeys.UPDATE_METADATA => handleUpdateMetadataRequest(request)
        case ApiKeys.CONTROLLED_SHUTDOWN => handleControlledShutdownRequest(request)
        case ApiKeys.OFFSET_COMMIT => handleOffsetCommitRequest(request)
        case ApiKeys.OFFSET_FETCH => handleOffsetFetchRequest(request)
        case ApiKeys.FIND_COORDINATOR => handleFindCoordinatorRequest(request)
        case ApiKeys.JOIN_GROUP => handleJoinGroupRequest(request)
        case ApiKeys.HEARTBEAT => handleHeartbeatRequest(request)
        case ApiKeys.LEAVE_GROUP => handleLeaveGroupRequest(request)
```

```
        case ApiKeys.SYNC_GROUP => handleSyncGroupRequest(request)
        case ApiKeys.DESCRIBE_GROUPS => handleDescribeGroupRequest(request)
        case ApiKeys.LIST_GROUPS => handleListGroupsRequest(request)
        case ApiKeys.SASL_HANDSHAKE => handleSaslHandshakeRequest(request)
        case ApiKeys.API_VERSIONS => handleApiVersionsRequest(request)
        case ApiKeys.CREATE_TOPICS => handleCreateTopicsRequest(request)
        case ApiKeys.DELETE_TOPICS => handleDeleteTopicsRequest(request)
        case ApiKeys.DELETE_RECORDS => handleDeleteRecordsRequest(request)
        case ApiKeys.INIT_PRODUCER_ID => handleInitProducerIdRequest(request)
        case ApiKeys.OFFSET_FOR_LEADER_EPOCH =>
handleOffsetForLeaderEpochRequest(request)
        case ApiKeys.ADD_PARTITIONS_TO_TXN => handleAddPartitionToTxnRequest(request)
        case ApiKeys.ADD_OFFSETS_TO_TXN => handleAddOffsetsToTxnRequest(request)
        case ApiKeys.END_TXN => handleEndTxnRequest(request)
        case ApiKeys.WRITE_TXN_MARKERS => handleWriteTxnMarkersRequest(request)
        case ApiKeys.TXN_OFFSET_COMMIT => handleTxnOffsetCommitRequest(request)
        case ApiKeys.DESCRIBE_ACLS => handleDescribeAcls(request)
        case ApiKeys.CREATE_ACLS => handleCreateAcls(request)
        case ApiKeys.DELETE_ACLS => handleDeleteAcls(request)
        case ApiKeys.ALTER_CONFIGS => handleAlterConfigsRequest(request)
        case ApiKeys.DESCRIBE_CONFIGS => handleDescribeConfigsRequest(request)
        case ApiKeys.ALTER_REPLICA_LOG_DIRS =>
handleAlterReplicaLogDirsRequest(request)
        case ApiKeys.DESCRIBE_LOG_DIRS => handleDescribeLogDirsRequest(request)
        case ApiKeys.SASL_AUTHENTICATE => handleSaslAuthenticateRequest(request)
        case ApiKeys.CREATE_PARTITIONS => handleCreatePartitionsRequest(request)
```

4.13 Kafka源码剖析之KafkaController

当前broker被选为新的controller的时候，执行如下操作：

1. 注册controller epoch事件监听
2. controller epoch +1
3. 初始化controller上下文对象，该上下文对象缓存当前所有主题、活跃broker以及所有分区leader的信息
4. 启动controller channel manager
5. 启动副本状态机
6. 启动分区状态机

如果注册为controller的过程中发生了异常，重新注册当前broker为controller，如此则触发新一轮controller选举，以保证永远有一个活跃的controller。

启动Kafka服务器的脚本：

kafka-server-start.sh × KafkaServer.scala × KafkaController.scala × ControllerEventManager.

```
44    exec $base_dir/kafka-run-class.sh $EXTRA_ARGS kafka.Kafka "$@"
```

main方法中创建KafkaServerStartable对象：

kafka-server-start.sh × Kafka.scala × KafkaServerStartable.scala × KafkaServer.scala × KafkaController.scala × ControllerEventManag

```
80    def main(args: Array[String]): Unit = {
81      try {
82        // 从命令参数解析kafka的代码
83        val serverProps = getPropsFromArgs(args)
84        val kafkaServerStartable = KafkaServerStartable.fromProps(serverProps)
85
86        // register signal handler to log termination due to SIGTERM, SIGHUP and SIGINT (control-c)
87        registerLoggingSignalHandler()
88
89        // attach shutdown handler to catch terminating signals as well as normal termination
90        Runtime.getRuntime().addShutdownHook(new Thread( name = "kafka-shutdown-hook") {
91          override def run(): Unit = kafkaServerStartable.shutdown()
92        })
93
94        // 启动服务
95        kafkaServerStartable.startup()
96        kafkaServerStartable.awaitShutdown()
97      }
98      catch {
```

该类中包含KakfaServer对象，startup方法调用的是KafkaServer的startup方法：

kafka-server-start.sh × Kafka.scala × KafkaServerStartable.scala × KafkaServer.scala × KafkaController.scala × ControllerEven

```
25    object KafkaServerStartable {
26      def fromProps(serverProps: Properties): KafkaServerStartable = {
27        val reporters = KafkaMetricsReporter.startReporters(new VerifiableProperties(serverProps))
28        new KafkaServerStartable(KafkaConfig.fromProps(serverProps), reporters)
29      }
30    }
31
32    class KafkaServerStartable(val serverConfig: KafkaConfig, reporters: Seq[KafkaMetricsReporter])
33      private val server = new KafkaServer(serverConfig, kafkaMetricsReporters = reporters)
34
35      def this(serverConfig: KafkaConfig) = this(serverConfig, Seq.empty)
36
37      def startup() {
38        try server.startup()
39        catch {
40          case _: Throwable =>
41            // KafkaServer.startup() calls shutdown() in case of exceptions, so we invoke `exit` to
42            fatal( msg = "Exiting Kafka.")
43            Exit.exit( statusCode = 1)
44        }
45      }
46
47      def shutdown() {
```

KafkaServer中的startup方法调用了kafkaController的startup方法：

KafkaServer.scala | KafkaController.scala | ControllerEventManager.scala | LogDirUtils.scala | PartitionLeaderSelector.scala | Replic

```
255     /*
256      * 启动Kafka的Controller
257      */
258     kafkaController = new KafkaController(config, zkUtils, time, metrics, threadNamePrefix)
259     // 调用Controller的startup方法
260     kafkaController.startup()
```

KafkaController的startup方法中，将Startup样例类设置到eventManager中，然后调用eventManager的start方法：

KafkaServer.scala | KafkaController.scala

```
707   /**
708    * 当启动Kafka服务器的controller模块的时候调用该方法。
709    * 当前方法不假定当前broker为controller
710    * 仅仅注册会话过期监听器，并开始controller的选举过程。
711    */
712   def startup() = {
713     eventManager.put(Startup)
714     eventManager.start()
715   }
```

上图中的eventManager.put(Startup)方法实现：

KafkaServer.scala | KafkaController.scala | ControllerEventManager.scala

```
45    // eventManager.put(Startup)
46    // 将样例类Startup放到queue中
47    def put(event: ControllerEvent): Unit = queue.put(event)
```

上图中的方法将Startup样例类放到queue中。

queue的实现：

KafkaServer.scala | KafkaController.scala | ControllerEventManager.scala

```
35    private val queue = new LinkedBlockingQueue[ControllerEvent]
```

Startup样例类：

其中的process方法执行controller的选举：

KafkaServer.scala | KafkaController.scala | ControllerEventManager.scala

```scala
/**
 * 样例类
 * 其中process方法执行controller选举
 */
case object Startup extends ControllerEvent {

  def state: ControllerState.ControllerChange.type = ControllerState.ControllerChange

  override def process(): Unit = {
    registerSessionExpirationListener()  1
    registerControllerChangeListener() 2
    // controller选举
    elect()  3
  }
```

上图中1的代码表示当session超时的时候的处理逻辑，也就是controller到zk连接超时重连，触发该逻辑：

KafkaServer.scala | KafkaController.scala | ZkUtils.scala | ControllerEventManager.scala

```scala
private def registerSessionExpirationListener(): Unit = {
  zkUtils.subscribeStateChanges(new SessionExpirationListener( controller = this, eventManager))
}
```

方法的实现：

KafkaServer.scala | KafkaController.scala | ZkUtils.scala | ControllerEventManager.scala

```scala
def subscribeStateChanges(listener: IZkStateListener): Unit =
  // 此处的listener是：new SessionExpirationListener(this, eventManager)
  zkClientWrap(_.subscribeStateChanges(listener))
```

当Controller到zk的连接过期重连的时候，调用方法：

KafkaServer.scala | ShutdownableThread.scala | KafkaController.scala | ZkUtils.scala | ZkClient.java

```
1936    * 会话过期监听器，一旦会话过期，触发controller选举
1937    * @param controller
1938    * @param eventManager
1939    */
1940   class SessionExpirationListener(controller: KafkaController,
1941                                   eventManager: ControllerEventManager)
1942     extends IZkStateListener with Logging {
1943
1944     override def handleStateChanged(state: KeeperState) {}
1945
1946     /**
1947       * 当到zk的会话过期，创建一个到zk的新会话，调用该方法。
1948       * 需要重新创建临时节点
1949       * @throws Exception On any error.
1950       */
1951     @throws[Exception]
1952     override def handleNewSession(): Unit = {
1953       eventManager.put(controller.Reelect)
1954     }
1955
1956     override def handleSessionEstablishmentError(error: Throwable): Unit = {}
1957   }
```

样例类：Reelect

KafkaServer.scala | ShutdownableThread.scala | KafkaController.scala | ZkUtils.scala | ZkClient.java | ControllerEve

```
1691    * 样例类，其中process方法执行controller选举
1692    */
1693   case object Reelect extends ControllerEvent {
1694
1695     def state: ControllerState.ControllerChange.type = ControllerState.ControllerChange
1696
1697     override def process(): Unit = {
1698       val wasActiveBeforeChange = isActive
1699       activeControllerId = getControllerID()
1700       if (wasActiveBeforeChange && !isActive) {
1701         onControllerResignation()
1702       }
1703       elect()
1704     }
1705   }
```

KafkaServer.scala | KafkaController.scala | ControllerEventManager.scala | ShutdownableThread.scala

```scala
/**
 * 选举Controller。通过zk的共享锁将自己选为Controller
 * 如果已经存在controller，则监听controller节点
 * 一旦controller发送变化，则尝试选举自己为controller
 */
def elect(): Unit = {
  val timestamp = time.milliseconds
  val electString = ZkUtils.controllerZkData(config.brokerId, timestamp)

```

上图中2的代码，表示当controller发生变化的时候的处理逻辑：

KafkaServer.scala | KafkaController.scala | ZkUtils.scala | ControllerEventManager.scala

```scala
private def registerControllerChangeListener(): Unit = {
  // 其中ZkUtils.ControllerPath = "/controller"
  zkUtils.subscribeDataChanges(ZkUtils.ControllerPath,
    new ControllerChangeListener( controller = this, eventManager))
}
```

方法的实现：

KafkaServer.scala | KafkaController.scala | ZkUtils.scala | ControllerEventManager.scala

```scala
def subscribeDataChanges(path: String, listener: IZkDataListener): Unit =
  // 其中的listener是：new ControllerChangeListener(this, eventManager)
  zkClientWrap(_.subscribeDataChanges(path, listener))
```

当controller发生变化的时候的处理逻辑（subscribeDataChanges）：

KafkaServer.scala | ShutdownableThread.scala | KafkaController.scala | ZkUtils.scala | ZkClient.java

```scala
public void subscribeStateChanges(final IZkStateListener listener) {
    synchronized (_stateListener) {
        _stateListener.add(listener);
    }
}
```

调用zk客户端设置监听器

调用：

KafkaServer.scala | ShutdownableThread.scala | KafkaController.scala | ZkUtils.scala | ZkClient.java | ControllerEventManager.scala

```scala
 * controller变更监听器
 * @param controller
 * @param eventManager
 */
class ControllerChangeListener(controller: KafkaController, eventManager: ControllerEventManager) extends IZkDataListener {
  /**
   * 数据改变了，说明controller发生变化了，监听就是了。
   * @param dataPath
   * @param data
   */
  override def handleDataChange(dataPath: String, data: Any): Unit = {
    eventManager.put(controller.ControllerChange(KafkaController.parseControllerId(data.toString)))
  }

  /**
   * 数据删除了，需要重新选举controller
   * @param dataPath
   */
  override def handleDataDeleted(dataPath: String): Unit = {
    eventManager.put(controller.Reelect)
  }
}
```

上图中3处的代码表示执行controller的选举：

KafkaServerStartable.scala | KafkaServer.scala | KafkaController.scala | ControllerEventManager.scala

```scala
  /**
   * 选举Controller。通过zk的共享锁将自己选为Controller
   * 如果已经存在controller，则监听controller节点
   * 一旦controller发送变化，则尝试选举自己为controller
   */
  def elect(): Unit = {
    val timestamp = time.milliseconds
    val electString = ZkUtils.controllerZkData(config.brokerId, timestamp)

    // 获取当前controller id
    activeControllerId = getControllerID()
    /*
```

KafkaController的startup方法中，调用eventManager的start方法：

KafkaServer.scala | KafkaController.scala

```scala
/**
 * 当启动Kafka服务器的controller模块的时候调用该方法。
 * 当前方法不假定当前broker为controller
 * 仅仅注册会话过期监听器，并开始controller的选举过程。
 */
def startup() = {
  eventManager.put(Startup)
  eventManager.start()
}
```

实现：

KafkaServer.scala | KafkaController.scala | ZkUtils.scala | ControllerEventManager.scala

```scala
def state: ControllerState = _state
// 调用ControllerEventThread的start方法
// 实际上该方法调用的是ControllerEventThread父类ShutdownableThread的start方法
// 最终调用的是当前类中的doWork方法
def start(): Unit = thread.start()
```

thread是ControllerEventThread对象：

KafkaServer.scala | KafkaController.scala | ZkUtils.scala | ControllerEventManager.scala

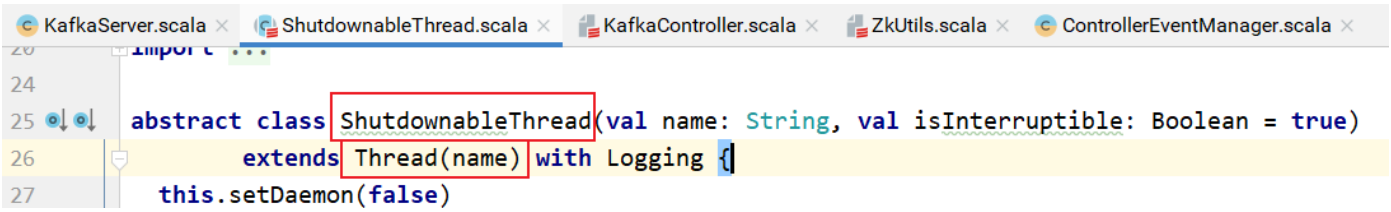
```scala
class ControllerEventThread(name: String) extends ShutdownableThread(name = name) {

  override def doWork(): Unit = {
    // 取出Startup
    val controllerEvent = queue.take()
    _state = controllerEvent.state

    try {
      rateAndTimeMetrics(state).time {
        // 调用样例类的process方法完成controller的选举过程
        controllerEvent.process()
      }
    } catch {
      case e: Throwable => error( msg = s"Error processing event $controllerEvent", e)
    }

    try eventProcessedListener(controllerEvent)
    catch {
      case e: Throwable => error( msg = s"Error while invoking listener for processed event $controllerEvent", e)
    }

    _state = ControllerState.Idle
  }
}
```

1. ControllerEventThread的父类是ShutdownableThread
ShutdownableThread的父类是Thread

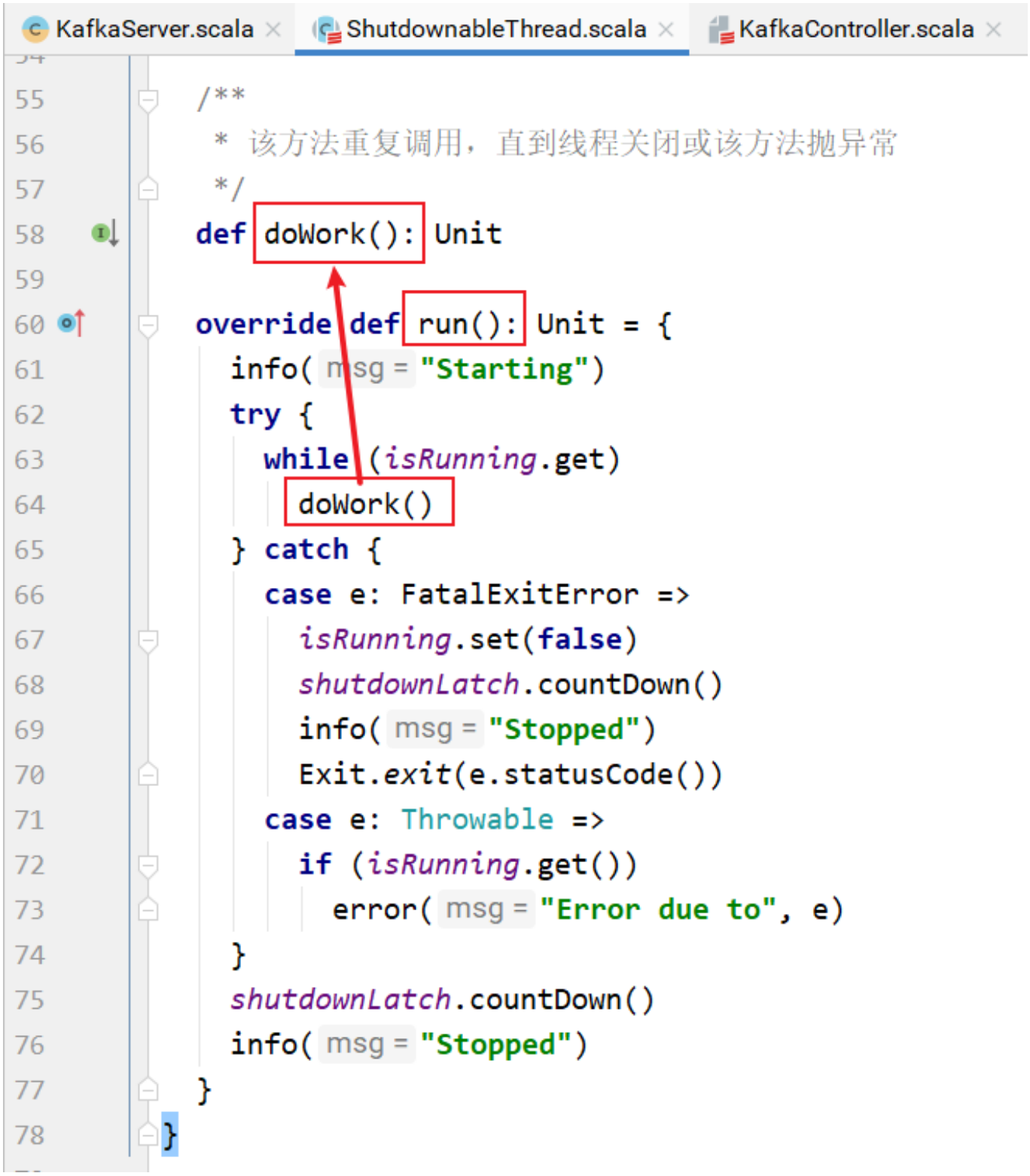
2. ControllerEventThread的start方法调用的是Thread的start方法
Thread的start方法调用run方法，而run方法实际上是调用
ShutdownableThread的run方法。

3. 该run方法调用ControllerEventThread的doWork方法。

ShutdownableThread的实现：



其中的run方法：



只要系统正常运行，就会不断调用doWork方法：

KafkaServer.scala | ControllerEventManager.scala | ShutdownableThread.scala | KafkaController.scala | ZkUtils.scala

```scala
class ControllerEventThread(name: String) extends ShutdownableThread(name = name) {

  override def doWork(): Unit = {
    // 取出Startup
    val controllerEvent = queue.take()
    _state = controllerEvent.state

    try {
      rateAndTimeMetrics(state).time {
        // 调用样例类的process方法完成controller的选举过程
        controllerEvent.process()
      }
    } catch {
      case e: Throwable => error( msg = s"Error processing event $controllerEvent", e)
    }

    try eventProcessedListener(controllerEvent)
    catch {
      case e: Throwable => error( msg = s"Error while invoking listener for processed event
    }

    _state = ControllerState.Idle
  }
}
```

样例类ControllerChange中：

KafkaServer.scala | KafkaController.scala | ControllerEventManager.scala | ShutdownableThread.scala | ZkUtils.scala

```scala
/**
 * controller发生变化之后的处理逻辑
 * @param newControllerId
 */
case class ControllerChange(newControllerId: Int) extends ControllerEvent {

  def state: ControllerState.ControllerChange.type = ControllerState.ControllerChange

  override def process(): Unit = {
    val wasActiveBeforeChange = isActive
    activeControllerId = newControllerId
    if (wasActiveBeforeChange && !isActive) {
      // 执行当前controller的卸任步骤
      onControllerResignation()
    }
  }

}
```

```
/**
  * This callback is invoked by the zookeeper leader elector when the current broker resigns as the controller. This is
  * required to clean up internal controller data structures
  */
def onControllerResignation() {
  debug("Resigning")
  // 取消注册ISR变化通知监听器
  deregisterIsrChangeNotificationListener()
  // 取消注册分区重新分配监听器
  deregisterPartitionReassignmentListener()
  // 取消注册带偏向的副本leader选举监听器
  deregisterPreferredReplicaElectionListener()
  // 取消注册log.dirs事件通知监听器
  deregisterLogDirEventNotificationListener()

  // 重置主题删除管理器
  topicDeletionManager.reset()

  // 关闭Kafka的leader再平衡调度器
  kafkaScheduler.shutdown()
  offlinePartitionCount = 0
  preferredReplicaImbalanceCount = 0
  globalTopicCount = 0
  globalPartitionCount = 0

  // 取消注册分区再平衡ISR变化监听器
  deregisterPartitionReassignmentIsrChangeListeners()
  // 关闭分区状态机
  partitionStateMachine.shutdown()
  // 取消注册主题变化监听器
  deregisterTopicChangeListener()
  // 取消注册一堆分区修改监听器
  partitionModificationsListeners.keys.foreach(deregisterPartitionModificationsListener)
  // 取消注册主题删除监听器
  deregisterTopicDeletionListener()
  // 关闭副本状态机
  replicaStateMachine.shutdown()
  // 取消注册broker变化监听器
  deregisterBrokerChangeListener()
  // 重置controller上下文
  resetControllerContext()

  // 日志：controller辞职不干了
  info("Resigned")
}
```

4.14 Kafka源码剖析之KafkaHealthcheck

健康检查的初始化和启动：

在启动KafkaServer的startup方法中，实例化并启动了健康检查：

KafkaHealthcheck.scala | KafkaServer.scala

```
321        }
322        // 创建健康检查对象
323        kafkaHealthcheck = new KafkaHealthcheck(config.brokerId, listeners, zkUtils, config.rack,
324          config.interBrokerProtocolVersion)
325        // 启动健康检查
326        kafkaHealthcheck.startup()
327
```

健康检查的startup方法的执行逻辑：

KafkaHealthcheck.scala | ZkUtils.scala | KafkaServer.scala

```
49    /**
50     * 健康检查的启动
51     */
52    def startup() {
53      // 注册状态监听器
54      zkUtils.subscribeStateChanges(sessionExpireListener)
55      // 将当前broker的信息注册到zk中
56      register()
57    }
```

注册状态监听器的具体实现：

KafkaHealthcheck.scala | EndPoint.scala | DynamicConfigManager.scala | ZkUtils.scala

```
677   def subscribeStateChanges(listener: IZkStateListener): Unit =
678     zkClientWrap(_.subscribeStateChanges(listener))
679
```

subscribeStateChanges(listener)具体实现：

调用zookeeper客户端的方法，该方法将监听器对象添加到_stateListener这个Set集合中：

```java
    }
209 }
210
211 public void subscribeStateChanges(final IZkStateListener listener) {
212     synchronized (_stateListener) {
213         _stateListener.add(listener);
214     }
215 }
```

zookeeper客户端的回调方法：

新建会话事件触发监听器：

```java
730 private void fireNewSessionEvents() {
731     for (final IZkStateListener stateListener : _stateListener) {
732         _eventThread.send(new ZkEvent( description: "New session event sent to " + stateListener) {
733
734             @Override
735             public void run() throws Exception {
736                 stateListener.handleNewSession();
737             }
738         });
739     }
740 }
```

如果发生了zk重连，则需要重新在zk中注册当前borker。

```scala
122 // zookeeper回调，用于处理建立到zk新会话事件
123 @throws[Exception]
124 override def handleNewSession() {
125   info( msg = "re-registering broker info in ZK for broker " + brokerId)
126   // 重新建立到zk的连接之后，重新注册broker到zk
127   register()
128   info( msg = "done re-registering broker")
129   info("Subscribing to %s path to watch for new topics".format(ZkUtils.BrokerTopicsPath))
130 }
```

会话建立异常，触发监听器：

```java
752 }
753
754 private void fireSessionEstablishmentError(final Throwable error) {
755     for (final IZkStateListener stateListener : _stateListener) {
756         _eventThread.send(new ZkEvent( description: "Session establishment error(" + error + ") sent to " + stateListener) {
757
758             @Override
759             public void run() throws Exception {
760                 stateListener.handleSessionEstablishmentError(error);
761             }
762         });
763     }
764 }
```

无法建立到zk的连接：

```scala
// zookeeper回调，用于处理到zk会话建立报错的事件
override def handleSessionEstablishmentError(error: Throwable) {
  fatal( msg = "Could not establish session with zookeeper", error)
}
```

状态改变，触发执行监听器方法：

```java
private void fireStateChangedEvent(final KeeperState state) {
    for (final IZkStateListener stateListener : _stateListener) {
        _eventThread.send(new ZkEvent( description: "State changed to " + state + " sent to " + stateListener) {

            @Override
            public void run() throws Exception {
                stateListener.handleStateChanged(state);
            }
        });
    }
}
```

只要状态发生改变，就标记当前事件的发生。用于监控。

```scala
// zookeeper回调，用于处理zk节点状态改变的事件
@throws[Exception]
override def handleStateChanged(state: KeeperState) {
  stateToMeterMap.get(state).foreach(_.mark())
}
```

其中register方法具体逻辑：

解决端点的主机名端口号，然后调用zkUtil的方法将当前broker的信息注册到zookeeper中：

KafkaHealthcheck.scala | EndPoint.scala | ZkUtils.scala | KafkaServer.scala

```scala
/**
 * Register this broker as "alive" in zookeeper
 * 在zk注册当前broker，保持broker状态：alive。
 */
def register() {
  val jmxPort = System.getProperty( key = "com.sun.management.jmxremote.port", def = "-1").toInt
  val updatedEndpoints = advertisedEndpoints.map(endpoint =>
    if (endpoint.host == null || endpoint.host.trim.isEmpty)
      endpoint.copy(host = InetAddress.getLocalHost.getCanonicalHostName)
    else
      endpoint
  )
  // 兼容仅支持PLAINTEXT的老客户端。如果提供了plaintext的端口，则使用它，否则注册一个空的端点，此时老客户端不能连接。
  val plaintextEndpoint = updatedEndpoints.find(_.securityProtocol == SecurityProtocol.PLAINTEXT).getOrElse(
    new EndPoint( host = null,  port = -1,  listenerName = null,  securityProtocol = null))
  // 在zookeeper中注册当前broker
  zkUtils.registerBrokerInZk(brokerId, plaintextEndpoint.host, plaintextEndpoint.port, updatedEndpoints, jmxPort, rack,
    interBrokerProtocolVersion)
}
```

registerBrokerInZk的具体逻辑：

```scala
  /**
   * 如果Kafka的apiVersion不低于0.10.0.X，则使用json v4格式（包含多个端点和机架）注册broker。
   * 否则使用json v2格式注册。
   *
   * json v4格式包含了默认的端点以兼容老客户端。
   *
   * @param id broker ID
   * @param host broker host name
   * @param port broker port
   * @param advertisedEndpoints broker对外提供服务的端点
   * @param jmxPort jmx port
   * @param rack broker所在机架
   * @param apiVersion Kafka version the broker is running as
   */
  def registerBrokerInZk(id: Int,
                         host: String,
                         port: Int,
                         advertisedEndpoints: Seq[EndPoint],
                         jmxPort: Int,
                         rack: Option[String],
                         apiVersion: ApiVersion) {
    // /brokers/ids/<broker.id>
    val brokerIdPath = BrokerIdsPath + "/" + id
    // see method documentation for reason why we do this
    val version = if (apiVersion >= KAFKA_0_10_0_IV1) 4 else 2
    val json = Broker.toJson(version, id, host, port, advertisedEndpoints, jmxPort,
rack)
    // 将broker信息注册到指定的路径。该znode的值就是json字符串
    // 默认znode节点是：/broker
    registerBrokerInZk(brokerIdPath, json)

    info("Registered broker %d at path %s with addresses: %s".format(id,
brokerIdPath, advertisedEndpoints.mkString(",")))
  }
```

在zk中注册broker的具体实现：

KafkaHealthcheck.scala | ZkUtils.scala | KafkaServer.scala

```
439   }
440
441   private def registerBrokerInZk(brokerIdPath: String, brokerInfo: String) {
442     try {
443       val zkCheckedEphemeral = new ZKCheckedEphemeral(brokerIdPath,
444                                                       brokerInfo,
445                                                       zkConnection.getZookeeper,
446                                                       isSecure)
447       zkClientWrap(_ => zkCheckedEphemeral.create())
448     } catch {
449       case _: ZkNodeExistsException =>
450         throw new RuntimeException("A broker is already registered on the path " + brokerIdPath
451                 + ". This probably " + "indicates that you either have configured a brokerid that is already in use, or "
452                 + "else you have shutdown this broker and restarted it faster than the zookeeper "
453                 + "timeout so it appears to be re-registering.")
454     }
455   }
```

创建临时节点，将信息写到zk中

主要是在zk的/brokers/[0...N] 路径上建立该Broker的信息，并且该节点是ZK中的Ephemeral Node，当此Broker离线的时候，zk上对应的节点也就消失了，那么其它Broker可以及时发现该Broker的异常。

```
1  class KafkaHealthcheck(private val brokerId: Int,
2                         private val advertisedHost: String,
3                         private val advertisedPort: Int,
4                         private val zkSessionTimeoutMs: Int,
5                         private val zkClient: ZkClient) extends Logging {
6    val brokerIdPath = ZkUtils.BrokerIdsPath + "/" + brokerId
7    val sessionExpireListener = new SessionExpireListener
8    def startup() {
9      zkClient.subscribeStateChanges(sessionExpireListener)
10     register()
11   }
12   def shutdown() {
13     zkClient.unsubscribeStateChanges(sessionExpireListener)
14     ZkUtils.deregisterBrokerInZk(zkClient, brokerId)
15   }
16   def register() {
17     val advertisedHostName =
18       if(advertisedHost == null || advertisedHost.trim.isEmpty)
19         InetAddress.getLocalHost.getCanonicalHostName
20       else
21         advertisedHost
22     val jmxPort = System.getProperty("com.sun.management.jmxremote.port", "-1").toInt
23 //在/brokers/ids/路径下存储broker的基本消息，例如端口号，ip地址，时间戳等，以上内容均在Ephemeral Node上，只要该broker和zk失去链接，则zk对应目录的内容被清空
24     ZkUtils.registerBrokerInZk(zkClient, brokerId, advertisedHostName, advertisedPort, zkSessionTimeoutMs, jmxPort)
25   }
```

```
//该SessionExpireListener的作用就是重建broker的节点，防止短暂的和zk失去链接之后，该broker对应的节点也全部丢失了
  class SessionExpireListener() extends IZkStateListener {
    @throws(classOf[Exception])
    def handleStateChanged(state: KeeperState) {
      // do nothing, since zkclient will do reconnect for us.
    }
    def handleNewSession() {
      info("re-registering broker info in ZK for broker " + brokerId)
      register()
      info("done re-registering broker")
      info("Subscribing to %s path to watch for new topics".format(ZkUtils.BrokerTopicsPath))
    }
  }
}
```

4.15 Kafka源码剖析之DynamicConfigManager

工作流程如下：

配置存储于/config/entityType/entityName，如/config/topics/<topic_name>以及/config/clients/<clientId>

默认配置存储与各自的<default>节点中，上述节点中保存的是覆盖默认配置的数据，以properties的格式。

可以使用分级路径同时指定多个实体的名称，如：/config/users/<user>/clients/<clientId>

设置通知路径/config/changes，避免对所有主题进行监控，有事通知。DynamicConfigManager监控该路径。

更新配置的第一步是更新配置的properties。

之后，在/config/changes/下创建一个新的序列znode，类似于/config/changes/config_change_12231，该节点保存了实体类型和实体名称。

序列znode包含的数据形式：{"version" : 1, "entity_type":"topic/client", "entity_name" : "topic_name/client_id"}

这只是一个通知，真正的配置数据存储于/config/entityType/entityName节点

版本2的通知格式：{"version" : 2, "entity_path":"entity_type/entity_name"}

可以使用分级路径指定多个实体：如，users/<user>/clients/<clientId>

该类对所有的broker设置监视器。监视器工作流程如下：

1. 监视器读取所有的配置更改通知。
2. 监视器跟踪它应用过的后缀数字最高的配置更新。
3. 监视器先前处理过的通知，15min之后监视器将其删除。

4. 对于新的更改，监视器读取新的配置，将新的配置和默认配置整合，然后更新现有的配置。

配置永远从zk配置路径读取，通知仅用于触发该动作。

如果一个broker宕机，错过了一个更新，没问题——当broker重启的时候，加载所有的配置。

注意：如果有两个连续的配置更新，可能只有最后一个会处理（因为在broker读取配置信息的时候，可能两个更新都处理过了）。

此时，broker不需要进行两次配置更新，虽然人畜无害。

DynamicConfigManager重启的时候，重新处理所有的通知。可能有点儿浪费资源，但是它避免了丢失配置更新。

但要避免在启动时出现任何竞争情况， 因为这些情况可能会丢失初始配置加载与注册更改通知之间的更改。

KafkaServer启动的时候，在startup方法中，配置动态配置管理器，并启动动态配置管理器：

KafkaApis.scala | ConfigHandler.scala | KafkaServer.scala

```
301     /* start dynamic config manager */
302     // 设置动态配置管理器，包含了主题配置管理，客户端ID配置管理，用户配置管理，broker配置管理
303     dynamicConfigHandlers = Map[String, ConfigHandler](
304       ConfigType.Topic -> new TopicConfigHandler(logManager, config, quotaManagers),
305       ConfigType.Client -> new ClientIdConfigHandler(quotaManagers),
306       ConfigType.User -> new UserConfigHandler(quotaManagers, credentialProvider),
307       ConfigType.Broker -> new BrokerConfigHandler(config, quotaManagers)
308     )
309
310     // 创建动态配置管理器
311     dynamicConfigManager = new DynamicConfigManager(zkUtils, dynamicConfigHandlers)
312     // 启动动态配置管理器，开始监听通知
313     dynamicConfigManager.startup()
```

DynamicConfigManager的startup方法的逻辑：

在动态配置管理器启动的时候，首先执行一遍配置更新。

DynamicConfigManager.scala | ConfigHandler.scala | LogManager.scala | KafkaServer.scala | AdminUtils.scala | ZkUtils.scala | ZkNodeChangeNotificationListen

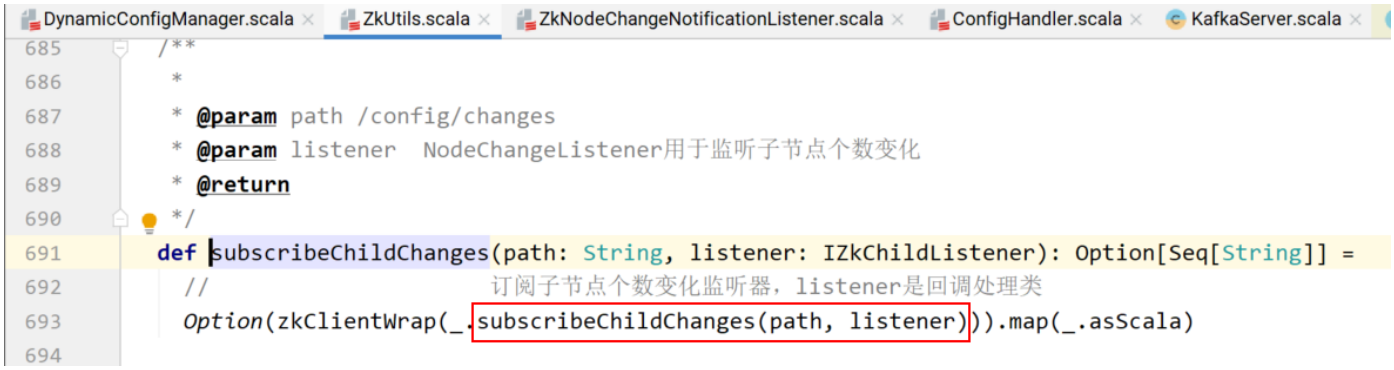
```scala
/**
 * 启动动态配置管理器，开始监控配置更改
 */
def startup(): Unit = {
  // 创建序列节点根节点（默认：/config/changes）并开始监控任何新增的子序列节点。
  configChangeListener.init()

  // Apply all existing client/user configs to the ClientIdConfigHandler/UserConfigHandler to bootstrap the overrides
  configHandlers.foreach {
    case (ConfigType.User, handler) =>
      // ConfigType.User -> new UserConfigHandler(quotaManagers, credentialProvider)
      AdminUtils.fetchAllEntityConfigs(zkUtils, ConfigType.User).foreach {
        case (sanitizedUser, properties) => handler.processConfigChanges(sanitizedUser, properties)
      }
      AdminUtils.fetchAllChildEntityConfigs(zkUtils, ConfigType.User, ConfigType.Client).foreach {
        case (sanitizedUserClientId, properties) => handler.processConfigChanges(sanitizedUserClientId, properties)
      }
    case (configType, handler) =>
      // ConfigType.Topic -> new TopicConfigHandler(logManager, config, quotaManagers)
      // ConfigType.Client -> new ClientIdConfigHandler(quotaManagers)
      // ConfigType.Broker -> new BrokerConfigHandler(config, quotaManagers)
      AdminUtils.fetchAllEntityConfigs(zkUtils, configType).foreach {
        case (entityName, properties) => handler.processConfigChanges(entityName, properties)
      }
  }
}
```

configChangeListener.init()方法的具体实现：

DynamicConfigManager.scala | ZkNodeChangeNotificationListener.scala | ConfigHandler.scala

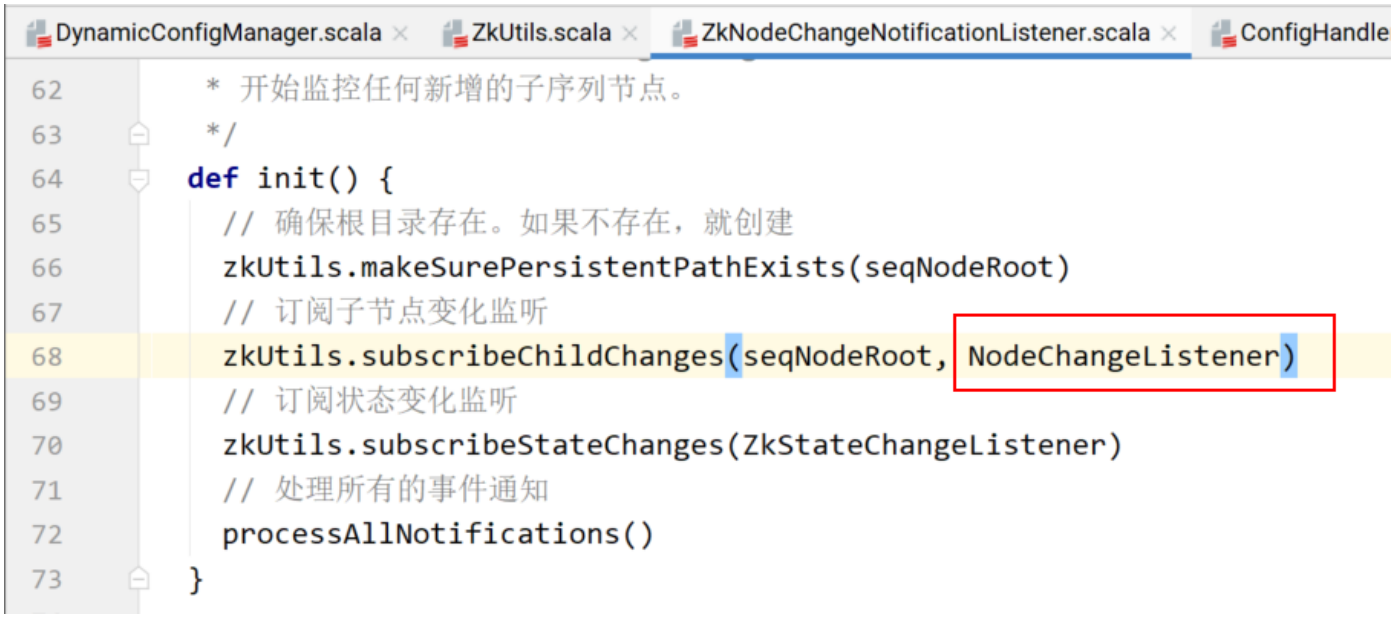
```scala
/**
 * 创建序列节点根节点 /config/changes
 * 开始监控任何新增的子序列节点。
 */
def init() {
  // 确保根目录存在。如果不存在，就创建
  zkUtils.makeSurePersistentPathExists(seqNodeRoot)
  // 订阅子节点变化监听
  zkUtils.subscribeChildChanges(seqNodeRoot, NodeChangeListener)
  // 订阅状态变化监听
  zkUtils.subscribeStateChanges(ZkStateChangeListener)
  // 处理所有的事件通知
  processAllNotifications()
}
```

上图中68行订阅子节点个数变化监听器，具体实现：



上图中标红框的是订阅子节点个数变化监听器，只要子节点个数发生变化，就回调listener。

listener是哪个？NodeChangeListener



NodeChangeListener的具体实现：

DynamicConfigManager.scala | ZkUtils.scala | ZkNodeChangeNotificationListener.scala | ConfigHandler.scala | KafkaServer.sc

```scala
/**
 * 当一个节点创建的时候调用该类中的handleChildChange方法
 */
object NodeChangeListener extends IZkChildListener {
  /**
   * 处理/config/changes子节点个数变化的回调方法
   * @param path
   * @param notifications
   */
  override def handleChildChange(path: String, notifications: java.util.List[String]) {
    try {
      import scala.collection.JavaConverters._
      if (notifications != null)
        // 处理通知
        processNotifications(notifications.asScala.sorted)
    } catch {
      case e: Exception => error( msg = s"Error processing notification change for path =
    }
  }
}
```

处理通知的实现：

```scala
/**
 * 处理给定的通知列表中的所有通知
 */
private def processNotifications(notifications: Seq[String]) {
  // 如果通知非空
  if (notifications.nonEmpty) {
    info(s"Processing notification(s) to $seqNodeRoot")
    try {
      val now = time.milliseconds
      // 遍历通知集合
      for (notification <- notifications) {
        // 获取通知的编号
        val changeId = changeNumber(notification)
        // 对比最后执行的修改通知编号，如果当前通知编号大于上次执行的，就执行配置更新
        if (changeId > lastExecutedChange) {
          // /config/changes/config_change_12121
          val changeZnode = seqNodeRoot + "/" + notification
          // 读取该通知节点的内容
          val data = zkUtils.readDataMaybeNull(changeZnode)._1.orNull
          if (data != null) {
            // 如果有需要更改的数据，则执行配置的更新
            notificationHandler.processNotification(data)
          } else {
            logger.warn(s"read null data from $changeZnode when processing
notification $notification")
          }
          // 修改上次已执行编号为当前节点编号
          lastExecutedChange = changeId
```

```
28            }
29          }
30          // 移除过期的通知
31          purgeObsoleteNotifications(now, notifications)
32        } catch {
33          case e: ZkInterruptedException =>
34            if (!isClosed.get)
35              throw e
36        }
37      }
38    }
```

上面代码中第22行的实现：

首先，notificationHandler是哪个？

DynamicConfigManager.scala | ZkUtils.scala | ZkNodeChangeNotificationListener.scala | ConfigHandler.scala | KafkaServer.scala | ZkClient.java

```
49   * @param time
50   */
51  class ZkNodeChangeNotificationListener(private val zkUtils: ZkUtils, // zk工具类
52                                         private val seqNodeRoot: String, // 序列节点根目录 /config/changes
53                                         private val seqNodePrefix: String,  // 序列节点名称前缀：config_changes_
54                                         private val notificationHandler: NotificationHandler, // 通知处理程序
55                                         private val changeExpirationMs: Long = 15 * 60 * 1000,  // 更新过期时间：15min
56                                         private val time: Time = Time.SYSTEM) extends Logging {
57    private var lastExecutedChange = -1L
58    private val isClosed = new AtomicBoolean( initialValue = false)
```

该类在哪里实例化？

DynamicConfigManager.scala | ZkUtils.scala | ZkNodeChangeNotificationListener.scala | ConfigHandler.scala

```
190   }
191
192   // 实例化配置更新监听器
193   private val configChangeListener = new ZkNodeChangeNotificationListener(
194     zkUtils,
195     ZkUtils.ConfigChangesPath, // "$ConfigPath/changes"
196     AdminUtils.EntityConfigChangeZnodePrefix, // "config_change_"
197     ConfigChangedNotificationHandler)
198
```

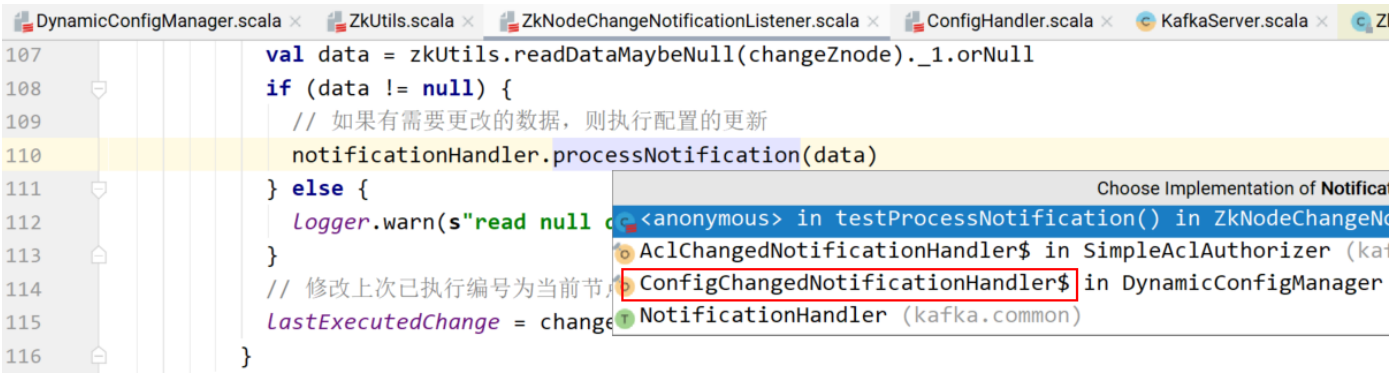
DynamicConfigManager.scala | ZkUtils.scala | ZkNodeChangeNotificationListener.scala | ConfigHandler.scala

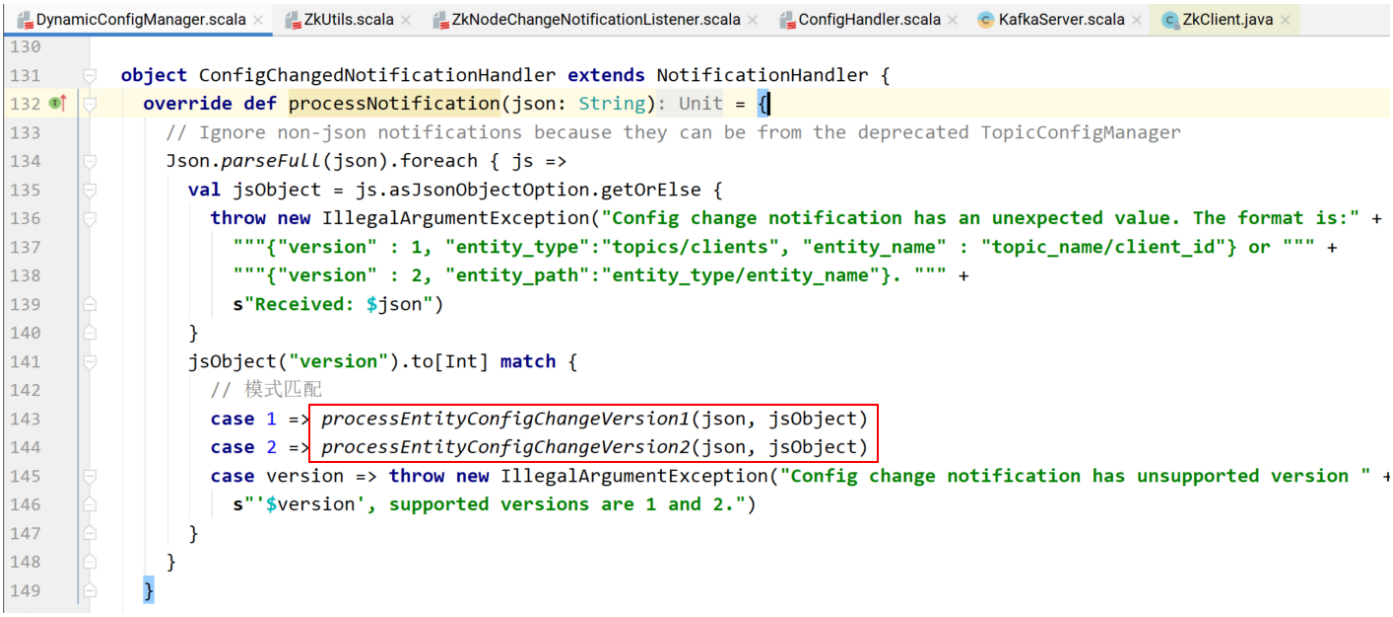
```
198
199   /**
200    * 启动动态配置管理器，开始监控配置更改
201    */
202   def startup(): Unit = {
203     // 创建序列节点根节点（默认：/config/changes）并开始监控任何新增的子序列节点。
204     configChangeListener.init()
205
```

即notificationHandler就是ConfigChangedNotificationHandler类。

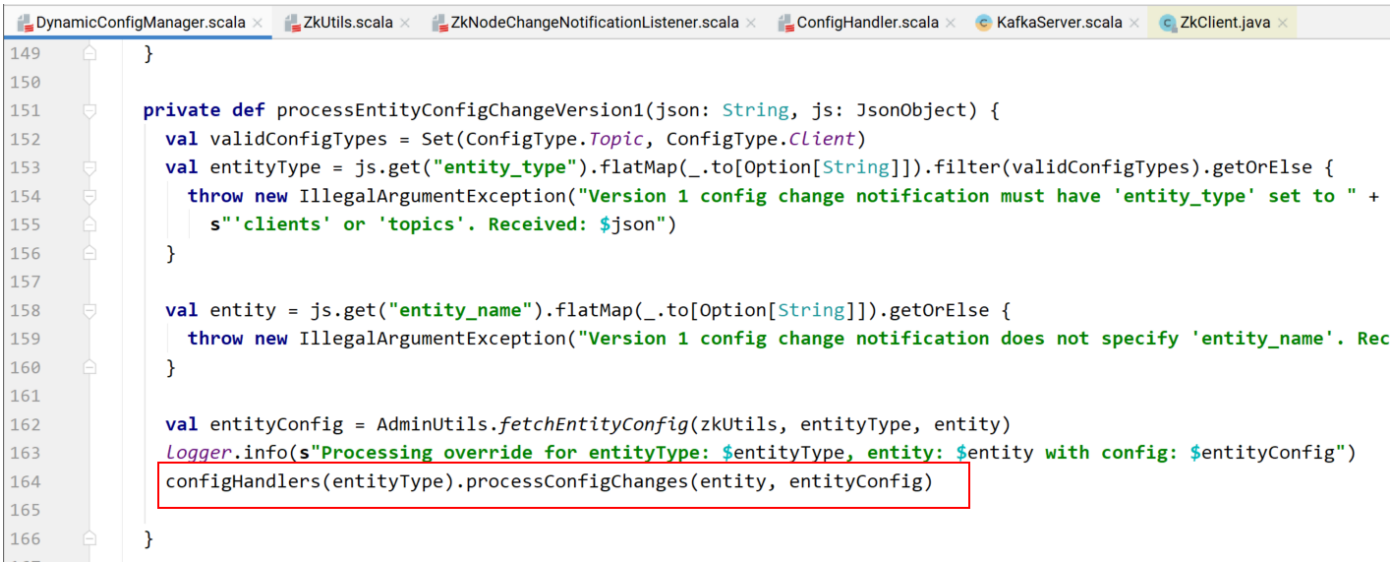
```
notificationHandler.processNotification(data)
```

上面代码的具体实现：





如果版本1，则：



如果版本2，则：

DynamicConfigManager.scala | ZkUtils.scala | ZkNodeChangeNotificationListener.scala | ConfigHandler.scala | KafkaServer.scala | ZkClient.java

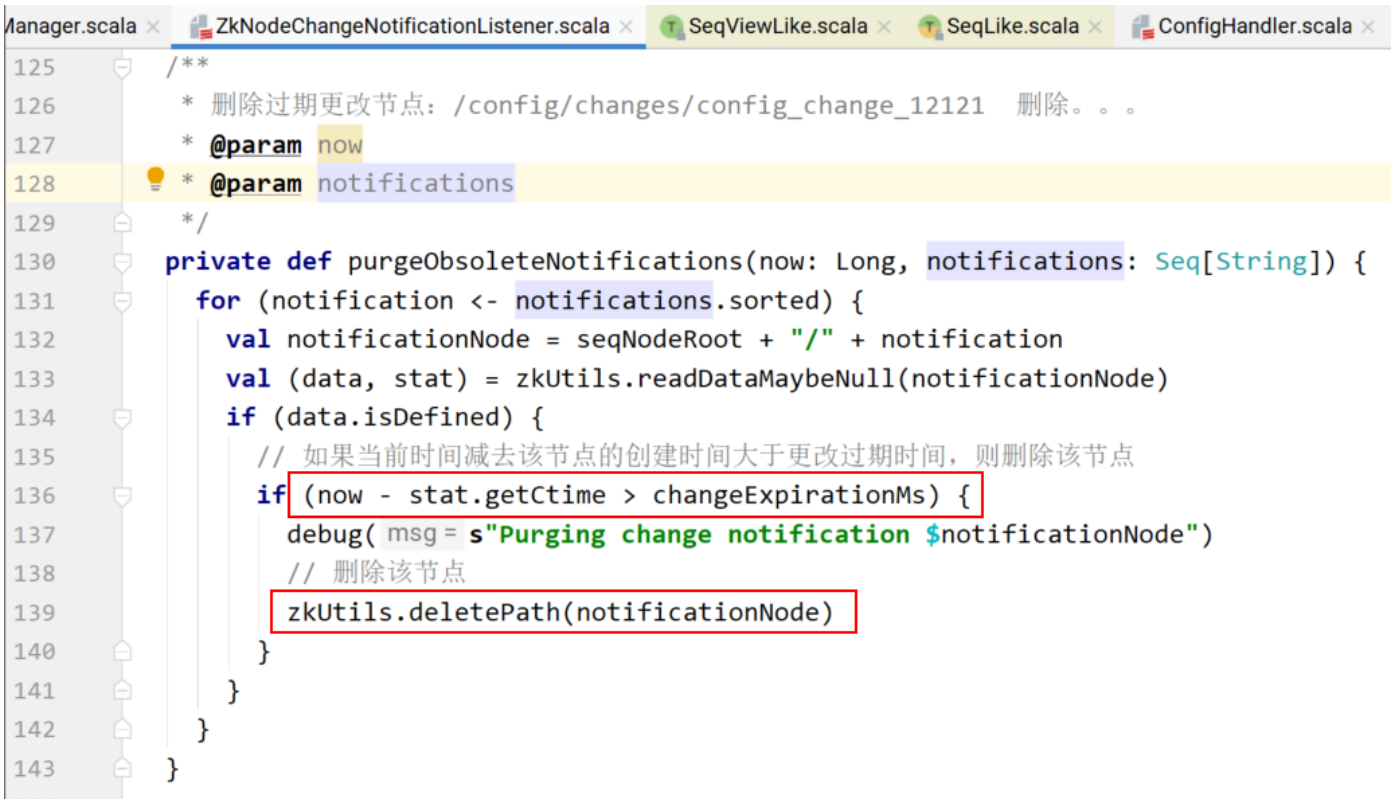
```scala
167
168   private def processEntityConfigChangeVersion2(json: String, js: JsonObject) {
169
170     val entityPath = js.get("entity_path").flatMap(_.to[Option[String]]).getOrElse {
171       throw new IllegalArgumentException(s"Version 2 config change notification must specify 'entity_path'. R
172     }
173
174     //
175     val index = entityPath.indexOf('/')
176     val rootEntityType = entityPath.substring(0, index)
177     if (index < 0 || !configHandlers.contains(rootEntityType)) {
178       val entityTypes = configHandlers.keys.map(entityType => s"'$entityType'/").mkString(", ")
179       throw new IllegalArgumentException("Version 2 config change notification must have 'entity_path' startin
180         s"one of $entityTypes. Received: $json")
181     }
182     val fullSanitizedEntityName = entityPath.substring(index + 1)
183
184     val entityConfig = AdminUtils.fetchEntityConfig(zkUtils, rootEntityType, fullSanitizedEntityName)
185     val loggableConfig = entityConfig.asScala.map {
186       case (k, v) => (k, if (ScramMechanism.isScram(k)) Password.HIDDEN else v)
187     }
188     logger.info(s"Processing override for entityPath: $entityPath with config: $loggableConfig")
189     // 比如是topics
190     configHandlers(rootEntityType).processConfigChanges(fullSanitizedEntityName, entityConfig)
191
192   }
```

具体实现：

```scala
  def processConfigChanges(topic: String, topicConfig: Properties) {
    // Validate the configurations.
    // 找出需要排除的配置条目
    val configNamesToExclude = excludedConfigs(topic, topicConfig)

    // 过滤出当前指定主题的所有分区日志
    val logs = logManager.logsByTopicPartition.filterKeys(_.topic ==
topic).values.toBuffer
    // 如果日志非空
    if (logs.nonEmpty) {
      // 整合默认配置和zk中覆盖默认的配置，创建新的Log配置信息
      val props = new Properties()
      // 添加默认配置
      props ++= logManager.defaultConfig.originals.asScala
      // 遍历覆盖默认配置的条目，如果该条目不在要排除的集合中，则直接put到props中
      // 该操作会覆盖默认相同key的配置
      topicConfig.asScala.foreach { case (key, value) =>
        if (!configNamesToExclude.contains(key)) props.put(key, value)
      }
      // 实例化新的logConfig
      val logConfig = LogConfig(props)
      if ((topicConfig.containsKey(LogConfig.RetentionMsProp)
        || topicConfig.containsKey(LogConfig.MessageTimestampDifferenceMaxMsProp))
        && logConfig.retentionMs < logConfig.messageTimestampDifferenceMaxMs)
        warn(s"${LogConfig.RetentionMsProp} for topic $topic is set to
${logConfig.retentionMs}. It is smaller than " +
```

```
25            s"${LogConfig.MessageTimestampDifferenceMaxMsProp}'s value ${logConfig.messageTimestampDifferenceMaxMs}. " +
26            s"This may result in frequent log rolling.")
27        // 更新当前主题所有分区日志的配置信息
28        logs.foreach(_.config = logConfig)
29      }
30
31      def updateThrottledList(prop: String, quotaManager: ReplicationQuotaManager) = {
32        if (topicConfig.containsKey(prop) && topicConfig.getProperty(prop).length > 0) {
33          val partitions = parseThrottledPartitions(topicConfig, kafkaConfig.brokerId, prop)
34          quotaManager.markThrottled(topic, partitions)
35          logger.debug(s"Setting $prop on broker ${kafkaConfig.brokerId} for topic: $topic and partitions $partitions")
36        } else {
37          quotaManager.removeThrottle(topic)
38          logger.debug(s"Removing $prop from broker ${kafkaConfig.brokerId} for topic $topic")
39        }
40      }
41      updateThrottledList(LogConfig.LeaderReplicationThrottledReplicasProp, quotas.leader)
42      updateThrottledList(LogConfig.FollowerReplicationThrottledReplicasProp, quotas.follower)
43    }
```

删除过期配置更新通知节点。通过时间对比，过期时间为：15min。



DynamicConfigManager.scala | ZkUtils.scala | ZkNodeChangeNotificationListener.scala | ConfigHandler.scala | KafkaServer.scala | ZkClient.java

```scala
/**
 * 启动动态配置管理器，开始监控配置更改
 */
def startup(): Unit = {
  // 创建序列节点根节点（默认：/config/changes）并开始监控任何新增的子序列节点。
  configChangeListener.init()

  // Apply all existing client/user configs to the ClientIdConfigHandler/UserConfigHandler to bootstrap the overrides
  configHandlers.foreach {
    case (ConfigType.User, handler) =>
      // ConfigType.User -> new UserConfigHandler(quotaManagers, credentialProvider)
      AdminUtils.fetchAllEntityConfigs(zkUtils, ConfigType.User).foreach {
        case (sanitizedUser, properties) => handler.processConfigChanges(sanitizedUser, properties)
      }
      AdminUtils.fetchAllChildEntityConfigs(zkUtils, ConfigType.User, ConfigType.Client).foreach {
        case (sanitizedUserClientId, properties) => handler.processConfigChanges(sanitizedUserClientId, properties)
      }
    case (configType, handler) =>
      // ConfigType.Topic -> new TopicConfigHandler(logManager, config, quotaManagers)
      // ConfigType.Client -> new ClientIdConfigHandler(quotaManagers)
      // ConfigType.Broker -> new BrokerConfigHandler(config, quotaManagers)
      AdminUtils.fetchAllEntityConfigs(zkUtils, configType).foreach {
        case (entityName, properties) => handler.processConfigChanges(entityName, properties)
      }
  }
}
```

动态配置管理器在启动的时候，初始化配置更新监听器
同时也会对所有的实体执行一次配置更新。

获取指定实体类型中各个实体的配置信息：

DynamicConfigManager.scala | KafkaServer.scala | AdminUtils.scala | ZkUtils.scala | ZkNodeChangeNotificationListener.scala | BaseProducer.scala

```scala
/**
 * 使用zk工具类，获取指定实体类型在zk中所有子节点的信息。<br />
 * 首先根据实体类型找到该实体类型在zk中对应的绝对节点路径<br />
 * 获取该绝对节点路径下所有子节点<br />
 * 从获取的所有子节点中找到各个子节点（实体名称）的配置信息。
 * @param zkUtils zk工具类
 * @param entityType 实体类型，如"users"
 * @return 返回指定实体类型下所有实体名称的配置信息。Map集合
 */
def fetchAllEntityConfigs(zkUtils: ZkUtils, entityType: String): Map[String, Properties] =
  //                          Seq集合          遍历，根据entity获取该entity的配置信息，以Map返回。
  zkUtils.getAllEntitiesWithConfig(entityType).map(entity => (entity, fetchEntityConfig(zkUtils, entityType, entity))).toMap
```

DynamicConfigManager.scala | KafkaServer.scala | AdminUtils.scala | ZkUtils.scala | ZkNodeChangeNotificationListene

```scala
/**
 * 根据指定的实体类型获取zk中该实体绝对路径下的所有子节点
 * @param entityType 实体类型，如"users"
 * @return 如果指定的实体类型不存在，返回空的Seq集合；<br />
 *         如果存在，则返回zk中该实体类型路径下的所有子节点集合
 */
def getAllEntitiesWithConfig(entityType: String): Seq[String] = {
  // 首先获取"users"节点的绝对路径，然后获取该节点下的子节点
  val entities = getChildrenParentMayNotExist(getEntityConfigRootPath(entityType))
  if(entities == null)
    // 如果不存在，就返回空Seq
    Seq.empty[String]
  else
    entities
}
```

getEntityConfigRootPath(entityType)的具体实现：

DynamicConfigManager.scala | KafkaServer.scala | AdminUtils.scala | ZkUtils.scala | Zk

```scala
/**
 * 返回指定实体类型在zk中的绝对路径
 * @param entityType 实体类型，如："users"
 * @return 返回指定实体类型在zk中的绝对路径 /config/users
 */
def getEntityConfigRootPath(entityType: String): String =
  ZkUtils.ConfigPath + "/" + entityType
```

其中，主题配置管理器TopicConfigHandler：

```scala
  def processConfigChanges(topic: String, topicConfig: Properties) {
    // Validate the configurations.
    // 找出需要排除的配置条目
    val configNamesToExclude = excludedConfigs(topic, topicConfig)

    // 过滤出当前指定主题的所有分区日志
    val logs = logManager.logsByTopicPartition.filterKeys(_.topic ==
topic).values.toBuffer
    // 如果日志非空
    if (logs.nonEmpty) {
      // 整合默认配置和zk中覆盖默认的配置，创建新的Log配置信息
      val props = new Properties()
      // 添加默认配置
      props ++= logManager.defaultConfig.originals.asScala
      // 遍历覆盖默认配置的条目，如果该条目不在要排除的集合中，则直接put到props中
      // 该操作会覆盖默认相同key的配置
      topicConfig.asScala.foreach { case (key, value) =>
```

```
        if (!configNamesToExclude.contains(key)) props.put(key, value)
      }
      // 实例化新的logConfig
      val logConfig = LogConfig(props)
      if ((topicConfig.containsKey(LogConfig.RetentionMsProp)
        || topicConfig.containsKey(LogConfig.MessageTimestampDifferenceMaxMsProp))
        && logConfig.retentionMs < logConfig.messageTimestampDifferenceMaxMs)
        warn(s"${LogConfig.RetentionMsProp} for topic $topic is set to ${logConfig.retentionMs}. It is smaller than " +
          s"${LogConfig.MessageTimestampDifferenceMaxMsProp}'s value ${logConfig.messageTimestampDifferenceMaxMs}. " +
          s"This may result in frequent log rolling.")
      // 更新当前主题所有分区日志的配置信息
      logs.foreach(_.config = logConfig)
    }

    def updateThrottledList(prop: String, quotaManager: ReplicationQuotaManager) = {
      if (topicConfig.containsKey(prop) && topicConfig.getProperty(prop).length > 0) {
        val partitions = parseThrottledPartitions(topicConfig, kafkaConfig.brokerId, prop)
        quotaManager.markThrottled(topic, partitions)
        logger.debug(s"Setting $prop on broker ${kafkaConfig.brokerId} for topic: $topic and partitions $partitions")
      } else {
        quotaManager.removeThrottle(topic)
        logger.debug(s"Removing $prop from broker ${kafkaConfig.brokerId} for topic $topic")
      }
    }
    updateThrottledList(LogConfig.LeaderReplicationThrottledReplicasProp, quotas.leader)
    updateThrottledList(LogConfig.FollowerReplicationThrottledReplicasProp, quotas.follower)
  }
```

4.16 Kafka源码剖析之分区消费模式

在分区消费模式，需要手动指定消费者要消费的主题和主题的分区信息。

可以设置从分区的哪个偏移量开始消费。

典型的分区消费：

```
Map<String, Object> configs = new HashMap<>();
configs.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
configs.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
```

```
configs.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
configs.put(ConsumerConfig.CLIENT_ID_CONFIG, "mycsmr" + System.currentTimeMillis());
configs.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
// 设置消费组id
// configs.put(ConsumerConfig.GROUP_ID_CONFIG, "csmr_grp_01");

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

TopicPartition tp0 = new TopicPartition("tp_demo_01", 0);
TopicPartition tp1 = new TopicPartition("tp_demo_01", 1);
TopicPartition tp2 = new TopicPartition("tp_demo_01", 2);

/*
  * 如果不设置消费组ID，则系统不会自动给消费者分配主题分区
  * 此时需要手动指定消费者消费哪些分区数据。
  */
consumer.assign(Arrays.asList(tp0, tp1, tp2));

consumer.seek(tp0, 0);
consumer.seek(tp1, 0);
consumer.seek(tp2, 0);

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

records.forEach(record -> {
    System.out.println(record.topic() + "\t"
    + record.partition() + "\t"
    + record.offset() + "\t"
    + record.key() + "\t"
    + record.value());
});

// 最后关闭消费者
consumer.close();
```

上面代码中的assign方法的实现：

KafkaConsumer.java | SubscriptionState.java | Fetcher.java

```
1037         */
1038        @Override
1039        public void assign(Collection<TopicPartition> partitions) {
1040            acquireAndEnsureOpen();
1041            try {
1042                if (partitions == null) {
1043                    throw new IllegalArgumentException("Topic partition collection to assign to cannot b
1044                } else if (partitions.isEmpty()) {
1045                    this.unsubscribe();
1046                } else {
1047                    Set<String> topics = new HashSet<>();
1048                    // 遍历手动赋值的主题分区
1049                    for (TopicPartition tp : partitions) {
1050                        String topic = (tp != null) ? tp.topic() : null;
1051                        if (topic == null || topic.trim().isEmpty())
1052                            throw new IllegalArgumentException("Topic partitions to assign to cannot hav
1053                        // 将手动指定的主题名称添加到集合中
1054                        topics.add(topic);
1055                    }
1056
1057                    // 对于取消订阅的主题分区，可能需要提交偏移量，因为手动赋值主题分区后就没有再平衡了。
1058                    this.coordinator.maybeAutoCommitOffsetsNow();
1059
1060                    log.debug("Subscribed to partition(s): {}", Utils.join(partitions, separator: ", "));
1061                    // 使用用户指定的主题分区集合重置subscriptions
1062                    this.subscriptions.assignFromUser(new HashSet<>(partitions));
1063                    metadata.setTopics(topics);
1064                }
```

如果该方法指定的集合为空，则清空原来的主题分区信息

assignFromUser的实现：

KafkaConsumer.java | SubscriptionState.java | Fetcher.java

```
158     /**
159      * 将赋值的主题分区信息按照用户指定的来修改。<br />
160      * 该方法与{@link #assignFromSubscribed(Collection)}不同，它是从已经订阅的主题分区中进行修
161      */
162     public void assignFromUser(Set<TopicPartition> partitions) {
163         // 设置订阅的类型为用户赋值的，不是再平衡的结果
164         setSubscriptionType(SubscriptionType.USER_ASSIGNED);
165
166         // 如果赋值信息与partitions不同
167         if (!this.assignment.partitionSet().equals(partitions)) {
168             fireOnAssignment(partitions);
169
170             Map<TopicPartition, TopicPartitionState> partitionToState = new HashMap<>();
171             for (TopicPartition partition : partitions) {
172                 TopicPartitionState state = assignment.stateValue(partition);
173                 if (state == null)
174                     state = new TopicPartitionState();
175                 partitionToState.put(partition, state);
176             }
177             // 重新设置赋值主题分区信息
178             this.assignment.set(partitionToState);
179             this.needsFetchCommittedOffsets = true;
180         }
181     }
```

调用seek方法指定各个主题分区从哪个偏移量开始消费：

KafkaConsumer.java | SubscriptionState.java | Fetcher.java

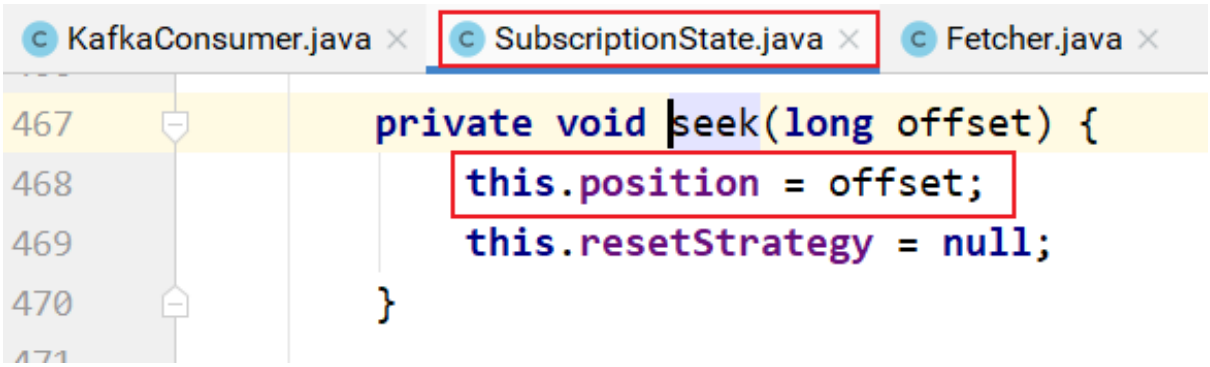
```
1338     @Override
1339     public void seek(TopicPartition partition, long offset) {
1340         acquireAndEnsureOpen();
1341         try {
1342             if (offset < 0)
1343                 throw new IllegalArgumentException("seek offset must not be a negative number
1344
1345             log.debug("Seeking to offset {} for partition {}", offset, partition);
1346             this.subscriptions.seek(partition, offset);
1347         } finally {
1348             release();
1349         }
```

subscriptions的seek方法实现：

KafkaConsumer.java | SubscriptionState.java | Fetcher.java

```
294
295     public void seek(TopicPartition tp, long offset) {
296         assignedState(tp).seek(offset);
297     }
```

上图中seek的实现：



此时poll方法的调用为：

pollOnce方法的实现：

发起请求：

该方法的实现：

创建需要发送的请求对象并发起请求：

```
KafkaConsumer.java | ConsumerCoordinator.java | SubscriptionState.java | Fetcher.java
193         * @return number of fetches sent
194         */
195        public int sendFetches() {
196            // 创建一个map集合，其中的信息是对应于节点的FetchRequest建造器对象
197            Map<Node, FetchRequest.Builder> fetchRequestMap = createFetchRequests();
198            for (Map.Entry<Node, FetchRequest.Builder> fetchEntry : fetchRequestMap.entrySet()) {
199                final FetchRequest.Builder request = fetchEntry.getValue();
200                final Node fetchTarget = fetchEntry.getKey();
201
202                log.debug("Sending {} fetch for partitions {} to broker {}", isolationLevel, request.fetchData().keySet(),
203                        fetchTarget);
204                client.send(fetchTarget, request)
205                        .addListener(new RequestFutureListener<ClientResponse>() {
206                            @Override
```

client.send方法添加监听器，等待broker端的响应：

```
KafkaConsumer.java | ConsumerCoordinator.java | SubscriptionState.java | Fetcher.java
203                         fetchTarget);
204                 client.send(fetchTarget, request)
205                         .addListener(new RequestFutureListener<ClientResponse>() {
206                             @Override
207                             public void onSuccess(ClientResponse resp) {
208                                 FetchResponse response = (FetchResponse) resp.responseBod
```

监听的逻辑：

```
KafkaConsumer.java | ConsumerCoordinator.java | SubscriptionState.java | Fetcher.java
216     }
217
218     Set<TopicPartition> partitions = new HashSet<>(response.responseData().keySet());
219     FetchResponseMetricAggregator metricAggregator = new FetchResponseMetricAggregator(sensors, partitions);
220
221     for (Map.Entry<TopicPartition, FetchResponse.PartitionData> entry : response.responseData().entrySet()) {
222         TopicPartition partition = entry.getKey();
223         long fetchOffset = request.fetchData().get(partition).fetchOffset;
224         FetchResponse.PartitionData fetchData = entry.getValue();
225
226         log.debug("Fetch {} at offset {} for partition {} returned fetch data {}",
227                 isolationLevel, fetchOffset, partition, fetchData);
228         completedFetches.add(new CompletedFetch(partition, fetchOffset, fetchData, metricAggregator,
229                 resp.requestHeader().apiVersion()));
230     }
231
232     sensors.fetchLatency.record(resp.requestLatencyMs());
```

上面方法中createFetchRequests用于创建需要发起的请求：

KafkaConsumer.java | ConsumerCoordinator.java | SubscriptionState.java | Fetcher.java

```
777 @   private Map<Node, FetchRequest.Builder> createFetchRequests() {
778         // create the fetch info
779         Cluster cluster = metadata.fetch();
780         Map<Node, LinkedHashMap<TopicPartition, FetchRequest.PartitionData>> fetchable = new LinkedHashMap<>();
781         for (TopicPartition partition : fetchablePartitions()) {
782             // 查找分区的leader所在的broker节点
783             Node node = cluster.leaderFor(partition);
784             if (node == null) {
785                 metadata.requestUpdate();
786             } else if (!this.client.hasPendingRequests(node)) {
787                 // if there is a leader and no in-flight requests, issue a new fetch
788                 LinkedHashMap<TopicPartition, FetchRequest.PartitionData> fetch = fetchable.get(node);
789                 if (fetch == null) {
790                     fetch = new LinkedHashMap<>();
791                     fetchable.put(node, fetch);
792                 }
793
794                 long position = this.subscriptions.position(partition);
795                 fetch.put(partition, new FetchRequest.PartitionData(position, FetchRequest.INVALID_LOG_START_OFFSET,
796                         this.fetchSize));
797                 log.debug("Added {} fetch request for partition {} at offset {} to node {}", isolationLevel,
798                         partition, position, node);
799             } else {
800                 log.trace("Skipping fetch for partition {} because there is an in-flight request to {}", partition, node);
```

KafkaConsumer.java | ConsumerCoordinator.java | SubscriptionState.java | Fetcher.java

```
803
804         // create the fetches
805         Map<Node, FetchRequest.Builder> requests = new HashMap<>();
806         for (Map.Entry<Node, LinkedHashMap<TopicPartition, FetchRequest.PartitionData>> entry : fetchable.entrySet()) {
807             Node node = entry.getKey();
808             FetchRequest.Builder fetch = FetchRequest.Builder.forConsumer(this.maxWaitMs, this.minBytes,
809                     entry.getValue(), isolationLevel)
810                     .setMaxBytes(this.maxBytes);
811             requests.put(node, fetch);
812         }
813         return requests;
814     }
```

fetchablePartitions方法的实现：

KafkaConsumer.java | ConsumerCoordinator.java | SubscriptionState.java | Fetcher.java

```
756     /**
757      * 获取需要拉取数据的分区
758      * @return
759      */
760 @   private List<TopicPartition> fetchablePartitions() {
761         Set<TopicPartition> exclude = new HashSet<>();
762         List<TopicPartition> fetchable = subscriptions.fetchablePartitions();
763         if (nextInLineRecords != null && !nextInLineRecords.isFetched) {
764             exclude.add(nextInLineRecords.partition);
765         }
766         for (CompletedFetch completedFetch : completedFetches) {
767             exclude.add(completedFetch.partition);
768         }
769         fetchable.removeAll(exclude);
770         return fetchable;
771     }
```

subscriptions.fetchablePartitions()方法的实现：

KafkaConsumer.java | ConsumerCoordinator.java | SubscriptionState.java | Fetcher.java

```
303     /**
304      * 获取可以拉取数据的主题分区
305      * @return
306      */
307     public List<TopicPartition> fetchablePartitions() {
308         List<TopicPartition> fetchable = new ArrayList<>(assignment.size());
309         for (PartitionStates.PartitionState<TopicPartitionState> state : assignment.partitionStates()) {
310             if (state.value().isFetchable())
311                 fetchable.add(state.topicPartition());
312         }
313         return fetchable;
314     }
```

最终，pollOnce方法返回拉取的结果：

KafkaConsumer.java | ConsumerCoordinator.java | SubscriptionState.java | Fetcher.java

```
1163
1164            // send any new fetches (won't resend pending fetches)
1165            fetcher.sendFetches();
1166
1167            long now = time.milliseconds();
1168            long pollTimeout = Math.min(coordinator.timeToNextPoll(now), timeout);
1169
1170            client.poll(pollTimeout, now, () -> {
1173                    // since a fetch might be completed by the background thread, we need t
1174                    // to ensure that we do not block unnecessarily in poll()
1175                    return !fetcher.hasCompletedFetches();
1176            });
1178
1179            // after the long poll, we should check whether the group needs to rebalance
1180            // prior to returning data so that the group can stabilize faster
1181            if (coordinator.needRejoin())
1182                return Collections.emptyMap();
1183
1184            return fetcher.fetchedRecords();
1185        }
```

4.17 Kafka源码剖析之组消费模式

组消费模式指的是在消费者消费消息的时候，使用组协调器的再平衡机制自动分配要消费的分区（们）。

此时需要在消费者的配置中指定消费组ID，同时如果需要，设置偏移量重置的策略。

然后消费者订阅主题，就可以消费消息了。

```java
Map<String, Object> configs = new HashMap<>();
configs.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092");
configs.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
configs.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
configs.put(ConsumerConfig.CLIENT_ID_CONFIG, "mycsmr" + System.currentTimeMillis());
configs.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
// 设置消费组id
configs.put(ConsumerConfig.GROUP_ID_CONFIG, "csmr_grp_01");

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

consumer.subscribe(Collections.singleton("tp_demo_01"));

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

records.forEach(record -> {
    System.out.println(record.topic() + "\t"
    + record.partition() + "\t"
    + record.offset() + "\t"
    + record.key() + "\t"
    + record.value());
});

// 最后关闭消费者
consumer.close();
```

`consumer.subscribe`方法的实现：

KafkaConsumer.java

```java
     */
    @Override
    public void subscribe(Collection<String> topics) {
        subscribe(topics, new NoOpConsumerRebalanceListener());
    }
```

上面方法中第一个参数是订阅的主题集合，第二个参数是一个监听器，当发送再平衡的时候消费者想要执行的操作。

默认是NoOpConsumerRebalanceListener，即什么都不做：

NoOpConsumerRebalanceListener的实现：

KafkaConsumer.java | NoOpConsumerRebalanceListener.java

```java
/.../
package org.apache.kafka.clients.consumer.internals;

import ...

public class NoOpConsumerRebalanceListener implements ConsumerRebalanceListener {

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {}

    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {}

}
```

订阅方法的实现：

KafkaConsumer.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```java
    @Override
    public void subscribe(Collection<String> topics, ConsumerRebalanceListener listener) {
        acquireAndEnsureOpen();
        try {
            if (topics == null) {
                throw new IllegalArgumentException("Topic collection to subscribe to cannot be null");
            } else if (topics.isEmpty()) {
                // 如果topics集合不是null，但是空集合，
                // 则取消该消费者的订阅信息。
                this.unsubscribe();
            } else {
                for (String topic : topics) {
                    if (topic == null || topic.trim().isEmpty())
                        throw new IllegalArgumentException("Topic collection to subscribe to cannot con
                }

                throwIfNoAssignorsConfigured();

                log.debug("Subscribed to topic(s): {}", Utils.join(topics, separator: ", "));
                // subscriptions的操作。
                this.subscriptions.subscribe(new HashSet<>(topics), listener);
                // 设置元数据的订阅的主题
                metadata.setTopics(subscriptions.groupSubscription());
            }
        } finally {
            release();
        }
    }
```

subscriptions的订阅操作实现：

KafkaConsumer.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```
113
114     public void subscribe(Set<String> topics, ConsumerRebalanceListener listener) {
115         if (listener == null)
116             throw new IllegalArgumentException("RebalanceListener cannot be null");
117         // 设置订阅类型为自动指定
118         setSubscriptionType(SubscriptionType.AUTO_TOPICS);
119
120         this.listener = listener;
121         // 修改subscriptions的订阅信息
122         changeSubscription(topics);
123     }
```

就是对SubscriptionState的操作：

KafkaConsumer.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```
133     private void changeSubscription(Set<String> topicsToSubscribe) {
134         if (!this.subscription.equals(topicsToSubscribe)) {
135             this.subscription = topicsToSubscribe;
136             this.groupSubscription.addAll(topicsToSubscribe);
137         }
138     }
```

用户的poll的操作调用pollOnce方法：

KafkaConsumer.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```
1102    @Override
1103    public ConsumerRecords<K, V> poll(long timeout) {
1104        acquireAndEnsureOpen();
1105        try {
1106            if (timeout < 0)
1107                throw new IllegalArgumentException("Timeout must not be negative");
1108
1109            if (this.subscriptions.hasNoSubscriptionOrUserAssignment())
1110                throw new IllegalStateException("Consumer is not subscribed to any topics or assig
1111
1112            // poll for new data until the timeout expires
1113            long start = time.milliseconds();
1114            long remaining = timeout;
1115            do {
1116                Map<TopicPartition, List<ConsumerRecord<K, V>>> records = pollOnce(remaining);
1117                if (!records.isEmpty()) {
1118                    // before returning the fetched records, we can send off the next round of fet
```

pollOnce的实现：

KafkaConsumer.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```
1149    private Map<TopicPartition, List<ConsumerRecord<K, V>>> pollOnce(long timeout) {
1150        client.maybeTriggerWakeup();
1151        // Poll for coordinator events.
1152        // This ensures that the coordinator is known
1153        // and that the consumer has joined the group (if it is using group management).
1154        // This also handles periodic offset commits if they are enabled.
1155        // 负责周期性的偏移量提交
1156        coordinator.poll(time.milliseconds(), timeout);
1157
1158        // 如果没有指定从哪里开始消费的偏移量，自动补全
1159        if (!subscriptions.hasAllFetchPositions())
1160            updateFetchPositions(this.subscriptions.missingFetchPositions());
1161
1162        // if data is available already, return it immediately
1163        Map<TopicPartition, List<ConsumerRecord<K, V>>> records = fetcher.fetchedRecords();
1164        if (!records.isEmpty())
1165            return records;
1166
1167        // send any new fetches (won't resend pending fetches)
1168        fetcher.sendFetches();
```

coordinator.poll负责周期性地向broker提交偏移量信息。

上面方法中updateFetchPositions方法表示：如果订阅的主题分区没有偏移量信息，则更新主题分区的偏移量信息，这样就知道消费的时候从哪里开始消费了：

KafkaConsumer.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```
1776     */
1777    private void updateFetchPositions(Set<TopicPartition> partitions) {
1778        // lookup any positions for partitions which are awaiting reset (which may be the
1779        // case if the user called seekToBeginning or seekToEnd. We do this check first to
1780        // avoid an unnecessary lookup of committed offsets (which typically occurs when
1781        // the user is manually assigning partitions and managing their own offsets).
1782        fetcher.resetOffsetsIfNeeded(partitions);
1783
1784        if (!subscriptions.hasAllFetchPositions(partitions)) {
1785            // if we still don't have offsets for the given partitions, then we should either
1786            // seek to the last committed position or reset using the auto reset policy
1787
1788            // first refresh commits for all assigned partitions
1789            coordinator.refreshCommittedOffsetsIfNeeded();
1790
1791            // then do any offset lookups in case some positions are not known
1792            fetcher.updateFetchPositions(partitions);
1793        }
1794    }
```

上图中的fetcher.resetOffsetsIfNeeded方法的实现：

KafkaConsumer.java | Fetcher.java | RequestFuture.java | Metadata.java | SubscriptionState.java | NoOpConsume

```java
    /**
     * 对于等待显式重置偏移量的主题分区，查找并设置偏移量。
     * @param partitions 需要重置偏移量的分区
     */
    public void resetOffsetsIfNeeded(Set<TopicPartition> partitions) {
        final Set<TopicPartition> needsOffsetReset = new HashSet<>();
        for (TopicPartition tp : partitions) {
            // 如果订阅了该主题分区，并且该主题分区的偏移量需要重置
            if (subscriptions.isAssigned(tp) && subscriptions.isOffsetResetNeeded(tp))
                // 将当前主题分区添加到set集合中
                needsOffsetReset.add(tp);
        }
        // 如果集合不空，则重置偏移量信息
        if (!needsOffsetReset.isEmpty()) {
            resetOffsets(needsOffsetReset);
        }
    }
```

resetOffsets的具体实现：

KafkaConsumer.java | Fetcher.java | RequestFuture.java | Metadata.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```java
    private void resetOffsets(final Set<TopicPartition> partitions) {
        final Map<TopicPartition, Long> offsetResets = new HashMap<>();
        final Set<TopicPartition> partitionsWithNoOffsets = new HashSet<>();
        for (final TopicPartition partition : partitions) {
            // 根据重置策略重置主题分区的偏移量：EARLIEST|LATEST|无法处理的
            // 无法处理的放到partitionsWithNoOffsets中
            offsetResetStrategyTimestamp(partition, offsetResets, partitionsWithNoOffsets);
        }
        final Map<TopicPartition, OffsetData> offsetsByTimes
                = retrieveOffsetsByTimes(offsetResets, Long.MAX_VALUE, requireTimestamps: false);
        for (final TopicPartition partition : partitions) {
            final OffsetData offsetData = offsetsByTimes.get(partition);
            if (offsetData == null) {
                partitionsWithNoOffsets.add(partition);
                continue;
            }
            // we might lose the assignment while fetching the offset, so check it is still active
            // 给各个主题分区设置偏移量。
            if (subscriptions.isAssigned(partition)) {
                log.debug("Resetting offset for partition {} to offset {}.", partition, offsetData.offset);
                this.subscriptions.seek(partition, offsetData.offset);
            }
        }
        if (!partitionsWithNoOffsets.isEmpty()) {
            throw new NoOffsetForPartitionException(partitionsWithNoOffsets);
        }
```

KafkaConsumer.java | Fetcher.java | RequestFuture.java | Metadata.java | SubscriptionState.java | NoOpCor

```
390     private void offsetResetStrategyTimestamp(
391             final TopicPartition partition,
392             final Map<TopicPartition, Long> output,
393             final Set<TopicPartition> partitionsWithNoOffsets) {
394         // 获取重置偏移量的策略
395         OffsetResetStrategy strategy = subscriptions.resetStrategy(partition);
396         // 如果是最早的
397         if (strategy == OffsetResetStrategy.EARLIEST)
398             output.put(partition, ListOffsetRequest.EARLIEST_TIMESTAMP);
399         // 如果是最晚的
400         else if (strategy == OffsetResetStrategy.LATEST)
401             output.put(partition, endTimestamp());
402         else
403             // 如果不是上述两种情况，则将该主题分区添加到集合partitionsWithNoOffsets中
404             partitionsWithNoOffsets.add(partition);
405     }
```

上述的实现表示：首先根据重置策略重置主题分区的偏移量请求类型，然后发送请求，真正从主题的分区中获取偏移量。

其中上图中的

```
421         final Map<TopicPartition, OffsetData> offsetsByTimes
422                 = retrieveOffsetsByTimes(offsetResets, Long.MAX_VALUE, requireTimestamps: false);
```

需要向broker发请求，获取主题分区的偏移量，更新偏移量的值：

KafkaConsumer.java | Fetcher.java | RequestFuture.java | Metadata.java | SubscriptionState.java | NoOpConsumerRebalanceListener.jav

```
465 @   private Map<TopicPartition, OffsetData> retrieveOffsetsByTimes(
466             Map<TopicPartition, Long> timestampsToSearch, long timeout, boolean requireTimestamps) {
467         if (timestampsToSearch.isEmpty())
468             return Collections.emptyMap();
469
470         long startMs = time.milliseconds();
471         long remaining = timeout;
472         do {
473             // 向broker发送请求，获取指定主题分区的消费偏移量
474             RequestFuture<Map<TopicPartition, OffsetData>> future =
475                     sendListOffsetRequests(requireTimestamps, timestampsToSearch);
476             client.poll(future, remaining);
477
478             if (!future.isDone())
```

发送请求的实现：

KafkaConsumer.java | Fetcher.java | RequestFuture.java | Metadata.java | SubscriptionState.java | NoOpConsumerRebalanceLi

```
621         * @return A response which can be polled to obtain the corresponding timestamps and offsets
622         */
623 @       private RequestFuture<Map<TopicPartition, OffsetData>> sendListOffsetRequests(
624                 final boolean requireTimestamps,
625                 final Map<TopicPartition, Long> timestampsToSearch) {
626             // Group the partitions by node.
627             final Map<Node, Map<TopicPartition, Long>> timestampsToSearchByNode = new HashMap<>();
628             for (Map.Entry<TopicPartition, Long> entry: timestampsToSearch.entrySet()) {
629                 TopicPartition tp = entry.getKey();
```

KafkaConsumer.java | Fetcher.java | RequestFuture.java | Metadata.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```
650         for (Map.Entry<Node, Map<TopicPartition, Long>> entry : timestampsToSearchByNode.entrySet()) {
651             // 发送请求，添加监听器，接收broker返回的结果
652             sendListOffsetRequest(entry.getKey(), entry.getValue(), requireTimestamps)
653                     .addListener(new RequestFutureListener<Map<TopicPartition, OffsetData>>() {
654                         @Override
655                         public void onSuccess(Map<TopicPartition, OffsetData> value) {
656                             synchronized (listOffsetRequestsFuture) {
657                                 fetchedTimestampOffsets.putAll(value);
658                                 if (remainingResponses.decrementAndGet() == 0 && !listOffsetRequestsFuture.isDone())
659                                     listOffsetRequestsFuture.complete(fetchedTimestampOffsets);
660                             }
661                         }
662
663                         @Override
664                         public void onFailure(RuntimeException e) {
665                             synchronized (listOffsetRequestsFuture) {
666                                 // This may cause all the requests to be retried, but should be rare.
667                                 if (!listOffsetRequestsFuture.isDone())
668                                     listOffsetRequestsFuture.raise(e);
669                             }
670                         }
671                     });
672         }
673         return listOffsetRequestsFuture;
674     }
```

发送的请求是ListOffsetRequest请求：

```java
/**
 * 发送ListOffsetRequest请求到指定的broker，获取指定主题分区及指定时间戳的信息
 * @param node 向哪个节点发送请求
 * @param timestampsToSearch 主题分区和目标时间戳的map集合
 * @param requireTimestamp 是否需要broker在响应中添加时间戳信息
 * @return 可以通过poll的方式来获取的对应的时间戳和偏移量的响应对象。
 */
private RequestFuture<Map<TopicPartition, OffsetData>> sendListOffsetRequest(
        final Node node,
        final Map<TopicPartition, Long> timestampsToSearch,
        boolean requireTimestamp) {
    ListOffsetRequest.Builder builder = ListOffsetRequest.Builder
            .forConsumer(requireTimestamp, isolationLevel)
            .setTargetTimes(timestampsToSearch);

    log.trace("Sending ListOffsetRequest {} to broker {}", builder, node);
    return client.send(node, builder)
            .compose((response, future) -> {
                ListOffsetResponse lor = (ListOffsetResponse) response.responseBody();
                log.trace("Received ListOffsetResponse {} from broker {}", lor, node);
                handleListOffsetResponse(timestampsToSearch, lor, future);
            });
}
```

该请求在Broker中的处理：

```scala
 */
def handle(request: RequestChannel.Request) {
  try {
    trace( msg = s"Handling request:${request.requestDesc( details = true)} fro
      s"securityProtocol:${request.context.securityProtocol},principal:${re
    request.header.apiKey match {
      // 消息生产请求的处理逻辑
      case ApiKeys.PRODUCE => handleProduceRequest(request)
      // 处理
      case ApiKeys.FETCH => handleFetchRequest(request)
      case ApiKeys.LIST_OFFSETS => handleListOffsetRequest(request)
      case ApiKeys.METADATA => handleTopicMetadataRequest(request)
      case ApiKeys.LEADER_AND_ISR => handleLeaderAndIsrRequest(request)
```

具体处理：

KafkaConsumer.java | Fetcher.java | KafkaApis.scala | RequestFuture.java | Metadata.ja

```scala
698  def handleListOffsetRequest(request: RequestChannel.Request) {
699    val version = request.header.apiVersion()
700
701    val mergedResponseMap = if (version == 0)
702      handleListOffsetRequestV0(request)
703    else
704      // 处理请求
705      handleListOffsetRequestV1AndAbove(request)
706
707    sendResponseMaybeThrottle(request, requestThrottleMs => new Lis
708  }
```

该方法的实现：

KafkaConsumer.java | Fetcher.java | KafkaApis.scala | Log.scala | LogSegment.scala | RequestFuture.java | Metadat

```scala
761  }
762
763  private def handleListOffsetRequestV1AndAbove(request : RequestChannel.Request): Map[TopicPa
764    val correlationId = request.header.correlationId
765    val clientId = request.header.clientId
766    val offsetRequest = request.body[ListOffsetRequest]
```

如果是最晚的，直接设置最晚的偏移量，如果不是最晚的，则需要根据主题分区以及时间戳查找：

KafkaConsumer.java | Fetcher.java | KafkaApis.scala | Log.scala | LogSegment.scala | RequestFuture.java | Metadata.java | SubscriptionState.ja

```scala
800
801        if (timestamp == ListOffsetRequest.LATEST_TIMESTAMP)
802          TimestampOffset(RecordBatch.NO_TIMESTAMP, lastFetchableOffset)
803        else {
804          def allowed(timestampOffset: TimestampOffset): Boolean =
805            timestamp == ListOffsetRequest.EARLIEST_TIMESTAMP || timestampOffset.offset < lastFetchableOffset
806
807          fetchOffsetForTimestamp(topicPartition, timestamp)
808            .filter(allowed).getOrElse(TimestampOffset.Unknown)
809        }
```

查找的逻辑：

KafkaConsumer.java | Fetcher.java | KafkaApis.scala | Log.scala | LogSegment.scala | RequestFuture.java | Metadata.java | Subscripti

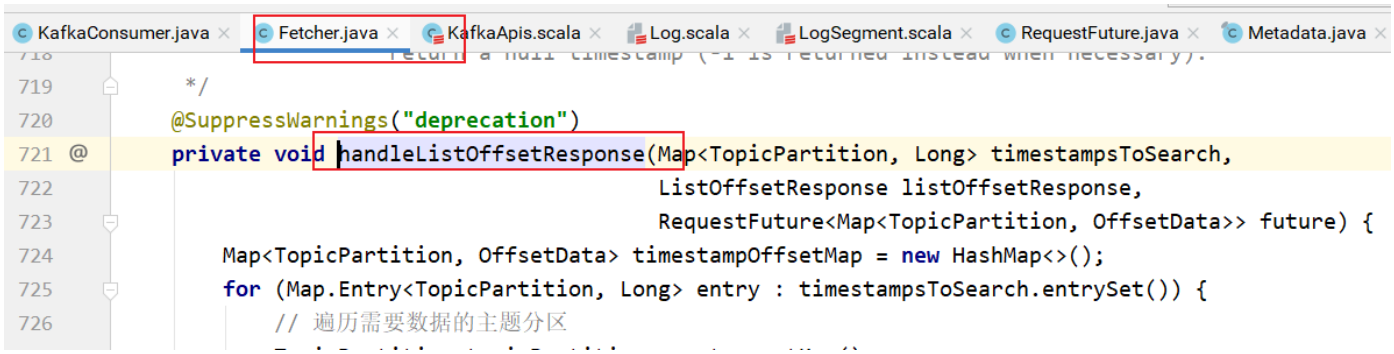
```scala
452
453   /**
454    * 根据消息的时间戳和偏移量查找消息的偏移量
455    *
456    * 返回Option(TimestampOffset)。返回值取决于：
457    *
458    * - 如果日志片段中的消息偏移量都比指定的小，则返回None
459    * - 如果日志片段中的消息时间戳都比指定的笑，返回None
460    * - 如果片段中所有的消息时间戳都比指定的大，或者消息都没有时间戳
461    *    返回的偏移量是max(片段的第一个消息偏移量，第一个对消费者可见的偏移量)
462    *    时间戳是：Message.NoTimestamp
463    * - 其他情况，返回Option(TimestampOffset)，可能有，也可能没有。
464    *    偏移量是比指定时间戳大的第一个消息的偏移量，以及偏移量是消息的偏移量大于等于startingOffset的偏移量。
465    *
466    * @param timestamp The timestamp to search for.
467    * @param startingOffset The starting offset to search.
468    * @return the timestamp and offset of the first message that meets the requirements. None will be returned
469    */
470   def findOffsetByTimestamp(timestamp: Long, startingOffset: Long = baseOffset): Option[TimestampOffset] = {
471     // Get the index entry with a timestamp less than or equal to the target timestamp
472     val timestampOffset = timeIndex.lookup(timestamp)
473     val position = index.lookup(math.max(timestampOffset.offset, startingOffset)).position
474
475     // Search the timestamp
476     Option(log.searchForTimestamp(timestamp, position, startingOffset)).map { timestampAndOffset =>
477       TimestampOffset(timestampAndOffset.timestamp, timestampAndOffset.offset)
478     }
479   }
```

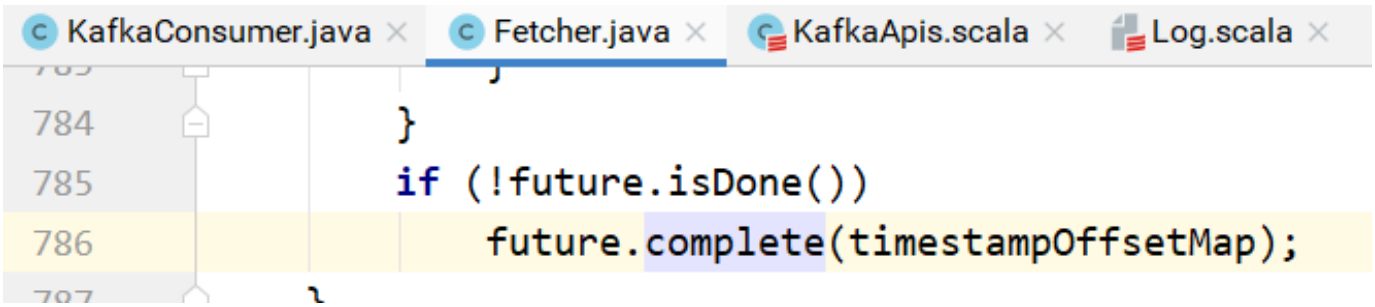
对于消费者，向指定的broker发送ListOffsetRequest请求，获取指定主题分区的偏移量和时间戳信息：

KafkaConsumer.java | Fetcher.java | Metadata.java | SubscriptionState.java | NoOpConsumerRebalanceListener.java

```java
676     /**
677      * 发送ListOffsetRequest请求到指定的broker，获取指定主题分区及指定时间戳的信息
678      * @param node 向哪个节点发送请求
679      * @param timestampsToSearch 主题分区和目标时间戳的map集合
680      * @param requireTimestamp 是否需要broker在响应中添加时间戳信息
681      * @return 可以通过poll的方式来获取的对应的时间戳和偏移量的响应对象。
682      */
683     private RequestFuture<Map<TopicPartition, OffsetData>> sendListOffsetRequest(
684             final Node node,
685             final Map<TopicPartition, Long> timestampsToSearch,
686             boolean requireTimestamp) {
687         ListOffsetRequest.Builder builder = ListOffsetRequest.Builder
688                 .forConsumer(requireTimestamp, isolationLevel)
689                 .setTargetTimes(timestampsToSearch);
690
691         log.trace("Sending ListOffsetRequest {} to broker {}", builder, node);
692         return client.send(node, builder)
693                 .compose((response, future) -> {
696                     ListOffsetResponse lor = (ListOffsetResponse) response.responseBody();
697                     log.trace("Received ListOffsetResponse {} from broker {}", lor, node);
698                     handleListOffsetResponse(timestampsToSearch, lor, future);
699                 });
701     }
```

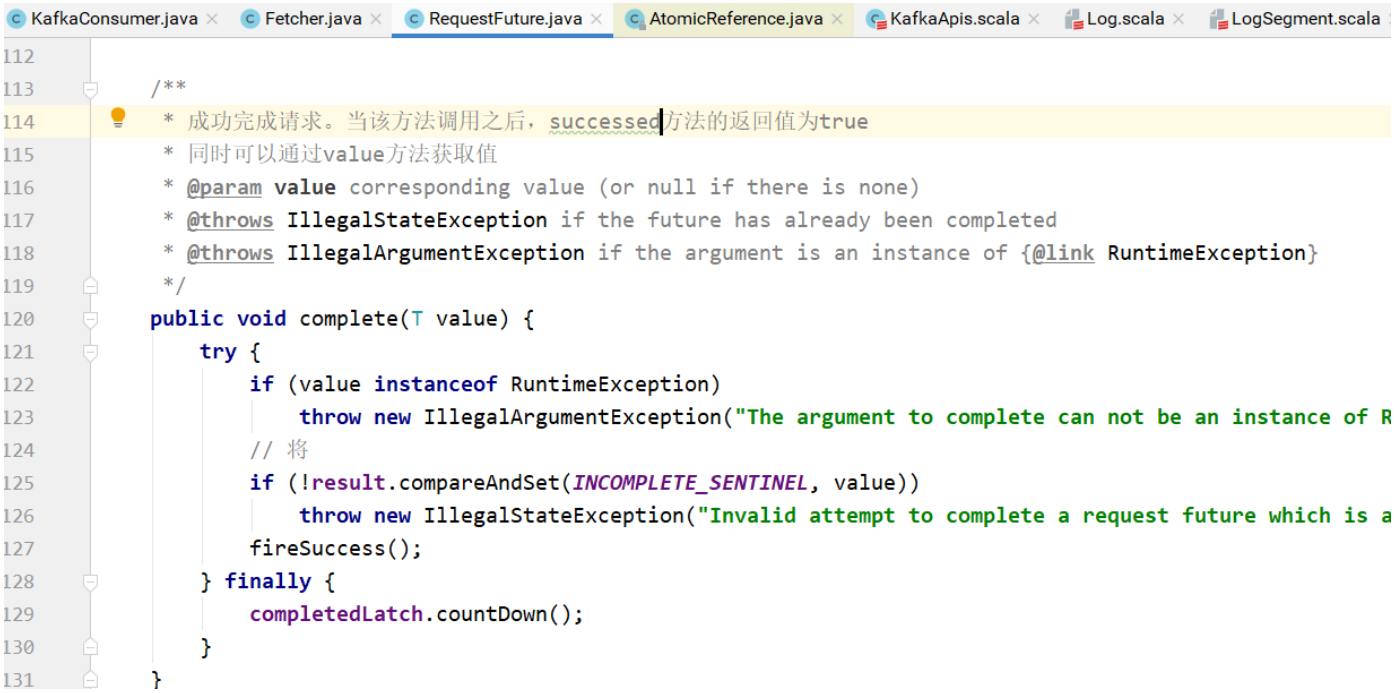
调用handleListOffsetResponse处理获取的偏移量信息：



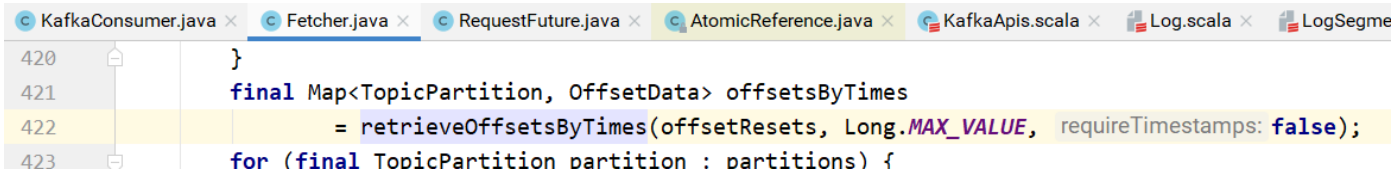


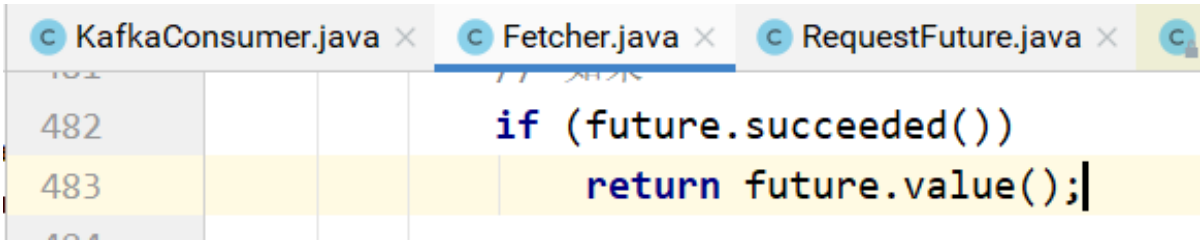
complete方法用于完成请求。当complete方法调用之后，successed方法返回true。

同时偏移量信息可以通过value方法获取：

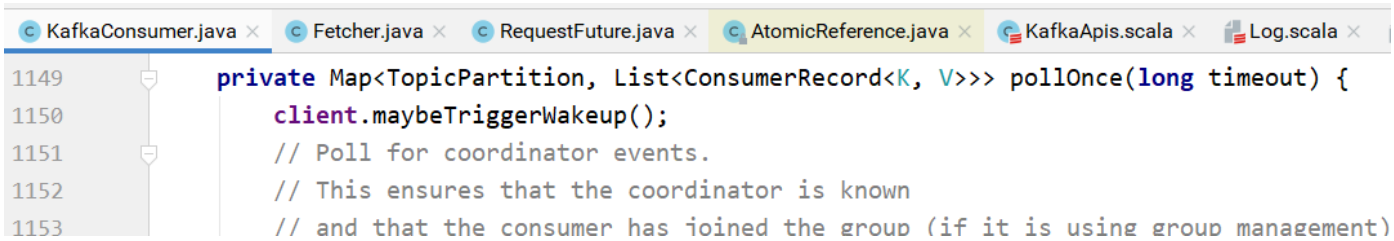


即：变量offsetsByTimes的值就是下图中future.value()的值。此时各个主题分区的偏移量已经设置好了：



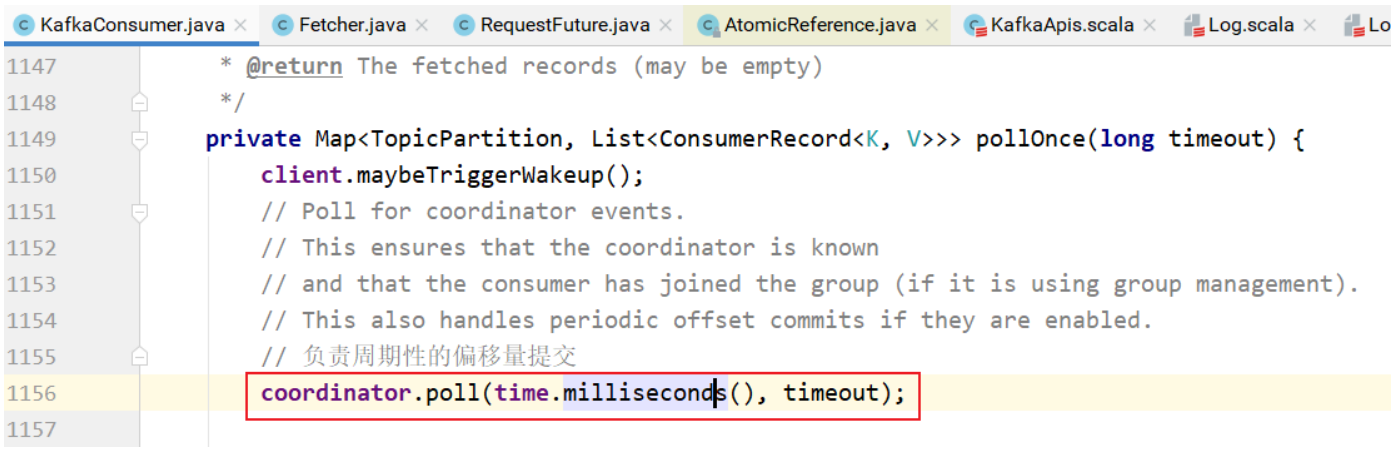


pollOnce方法：

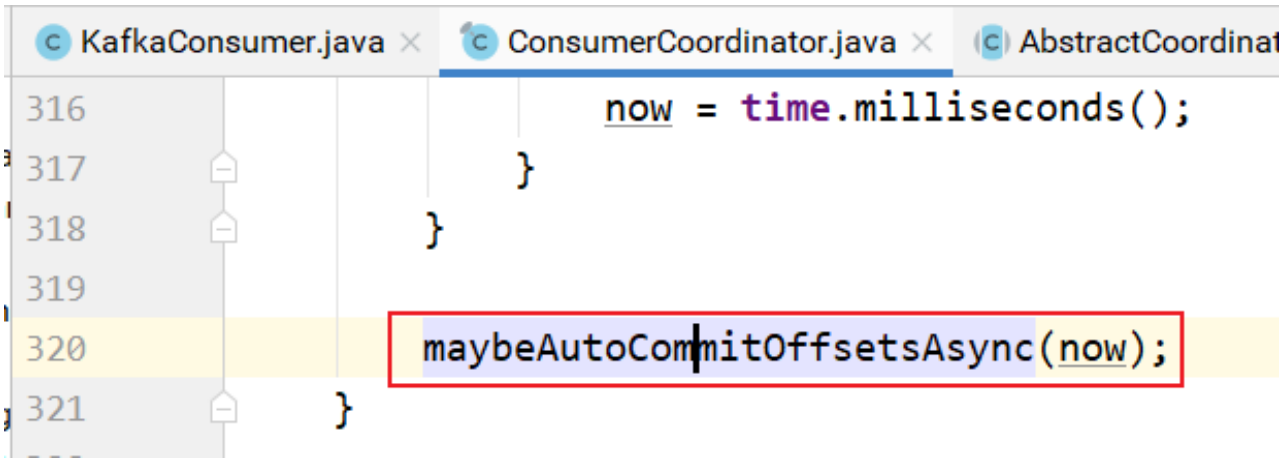


在更新主题分区的偏移量之后，就可以发送请求消费消息了：

对于组消费，还需要定期将偏移量提交到`__consumer_offsets`主题中：



poll方法的实现：



如果是自动提交消费者偏移量到broker的`__consumer_offsets`主题，则maybeAutoCommitOffsetsAsync的实现：

doAutoCommitOffsetsAsync的实现：

```java
KafkaConsumer.java | ConsumerCoordinator.java | AbstractCoordinator.java | Heartbeat.java | Fetcher.java | RequestFuture.java
660         * 异步自动提交偏移量
661         */
662        private void doAutoCommitOffsetsAsync() {
663            Map<TopicPartition, OffsetAndMetadata> allConsumedOffsets = subscriptions.allConsumed();
664            log.debug("Sending asynchronous auto-commit of offsets {}", allConsumedOffsets);
665
666            // 异步提交偏移量
667            commitOffsetsAsync(allConsumedOffsets, (offsets, exception) -> {
670                    if (exception != null) {
671                        log.warn("Asynchronous auto-commit of offsets {} failed: {}", offsets, excepti
672                        if (exception instanceof RetriableException)
673                            nextAutoCommitDeadline = Math.min(time.milliseconds() + retryBackoffMs, ne
674                    } else {
675                        log.debug("Completed asynchronous auto-commit of offsets {}", offsets);
676                    }
677            });
679        }
```

commitOffsetsAsync的实现：

```java
KafkaConsumer.java | ConsumerCoordinator.java | AbstractCoordinator.java | Heartbeat.java | Fetcher.java | RequestFuture.java | AtomicReference.java | KafkaApis.
517    public void commitOffsetsAsync(final Map<TopicPartition, OffsetAndMetadata> offsets, final OffsetCommitCallback callback) {
518        invokeCompletedOffsetCommitCallbacks();
519
520        if (!coordinatorUnknown()) {
521            // 如果消费组已知，则异步提交
522            doCommitOffsetsAsync(offsets, callback);
523        } else {
524            // we don't know the current coordinator, so try to find it and then send the commit
525            // or fail (we don't want recursive retries which can cause offset commits to arrive
526            // out of order). Note that there may be multiple offset commits chained to the same
527            // coordinator lookup request. This is fine because the listeners will be invoked in
528            // the same order that they were added. Note also that AbstractCoordinator prevents
529            // multiple concurrent coordinator lookup requests.
530            pendingAsyncCommits.incrementAndGet();
531            lookupCoordinator().addListener(new RequestFutureListener<Void>() {
532                @Override
533                public void onSuccess(Void value) {
534                    pendingAsyncCommits.decrementAndGet();
535                    doCommitOffsetsAsync(offsets, callback);
536                }
537
538                @Override
539                public void onFailure(RuntimeException e) {
540                    pendingAsyncCommits.decrementAndGet();
541                    completedOffsetCommits.add(new OffsetCommitCompletion(callback, offsets,
542                            new RetriableCommitFailedException(e)));
543                }
544            });
545        }
```

在异步提交消费者偏移量的时候，如果组协调器已知，直接发送

如果未知，则异步提交等待，查找组协调器，等找到之后，异步提交消费者偏移量：

```java
private void doCommitOffsetsAsync(final Map<TopicPartition, OffsetAndMetadata> offsets, final OffsetCommitCallback callback) {
    this.subscriptions.needRefreshCommits();
    // 发送提交请求
    RequestFuture<Void> future = sendOffsetCommitRequest(offsets);
    final OffsetCommitCallback cb = callback == null ? defaultOffsetCommitCallback : callback;
    future.addListener(new RequestFutureListener<Void>() {
        @Override
        public void onSuccess(Void value) {
            if (interceptors != null)
                interceptors.onCommit(offsets);

            completedOffsetCommits.add(new OffsetCommitCompletion(cb, offsets, exception: null));
        }

        @Override
        public void onFailure(RuntimeException e) {
            Exception commitException = e;

            if (e instanceof RetriableException)
                commitException = new RetriableCommitFailedException(e);

            completedOffsetCommits.add(new OffsetCommitCompletion(cb, offsets, commitException));
        }
    });
}
```

发送提交偏移量的请求到组协调器

如果设置了拦截器，则调用拦截器的方法

更新本地已提交偏移量的数据

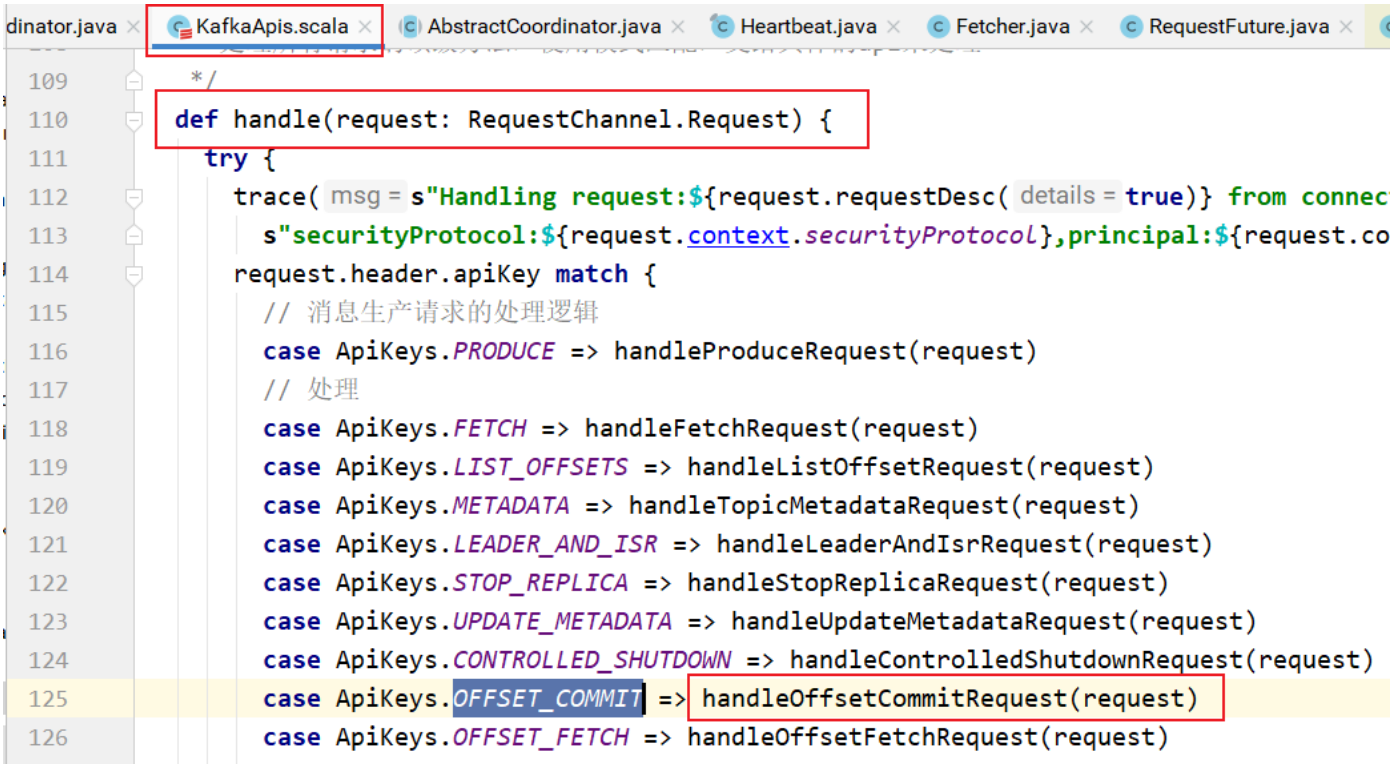
上图中sendOffsetCommitRequest的实现：

1. 首先查找消费组协调器
2. 然后创建偏移量提交请求对象
3. 发送请求

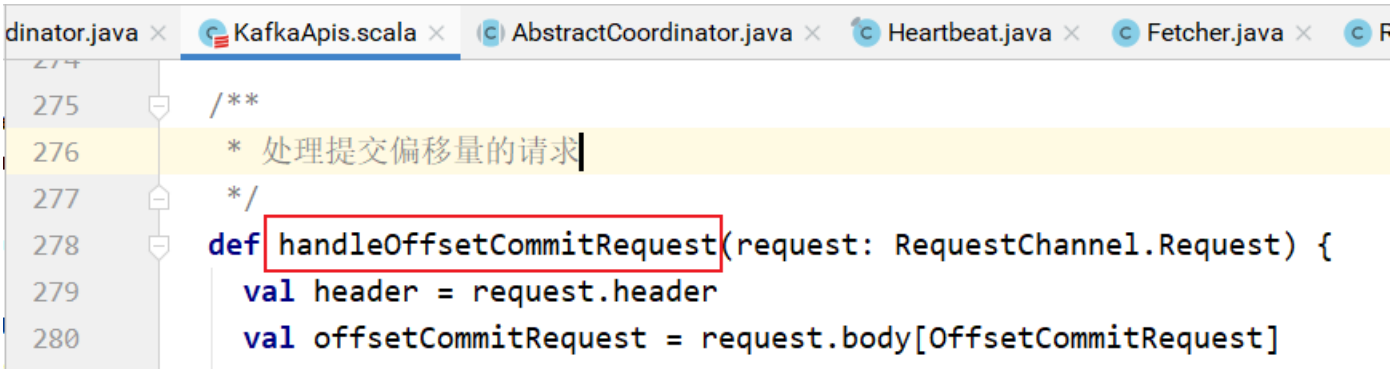
```java
private RequestFuture<Void> sendOffsetCommitRequest(final Map<TopicPartition, OffsetAndMetadata> offsets) {
    if (offsets.isEmpty())
        return RequestFuture.voidSuccess();
    // 查找组协调器
    Node coordinator = checkAndGetCoordinator();
    if (coordinator == null)
        return RequestFuture.coordinatorNotAvailable();

    // 创建偏移量提交请求对象
    Map<TopicPartition, OffsetCommitRequest.PartitionData> offsetData = new HashMap<>(offsets.size());
    for (Map.Entry<TopicPartition, OffsetAndMetadata> entry : offsets.entrySet()) {
        OffsetAndMetadata offsetAndMetadata = entry.getValue();
        if (offsetAndMetadata.offset() < 0) {
            return RequestFuture.failure(new IllegalArgumentException("Invalid offset: " + offsetAndMetadata.offset()));
        }
        offsetData.put(entry.getKey(), new OffsetCommitRequest.PartitionData(
                offsetAndMetadata.offset(), offsetAndMetadata.metadata()));
    }
```

```java
            return RequestFuture.failure(new CommitFailedException());

        OffsetCommitRequest.Builder builder = new OffsetCommitRequest.Builder(this.groupId, offsetData).
                setGenerationId(generation.generationId).
                setMemberId(generation.memberId).
                setRetentionTime(OffsetCommitRequest.DEFAULT_RETENTION_TIME);

        log.trace("Sending OffsetCommit request with {} to coordinator {}", offsets, coordinator);

        return client.send(coordinator, builder)
                .compose(new OffsetCommitResponseHandler(offsets));
    }
```

在KafkaServer处理的时候：



handleOffsetCommitRequest的实现：



```
dinator.java × KafkaApis.scala × AbstractCoordinator.java × Heartbeat.java ×
372
373            // call coordinator to handle commit offset
374            groupCoordinator.handleCommitOffsets(
375              offsetCommitRequest.groupId,
376              offsetCommitRequest.memberId,
377              offsetCommitRequest.generationId,
378              partitionData,
379              sendResponseCallback)
380          }
381        }
382      }
```

消费组协调器的处理：

```
dinator.java × KafkaApis.scala × GroupCoordinator.scala × AbstractCoordinator.java × Heartbeat.java × Fetcher.java × Reques
485    */
486   def handleCommitOffsets(groupId: String,
487                           memberId: String,
488                           generationId: Int,
489                           offsetMetadata: immutable.Map[TopicPartition, OffsetAndMetadata],
490                           responseCallback: immutable.Map[TopicPartition, Errors] => Unit) {
491     validateGroup(groupId) match {
492       case Some(error) => responseCallback(offsetMetadata.mapValues(  => error))
```

```
dinator.java × KafkaApis.scala × GroupCoordinator.scala × AbstractCoordinator.java × Heartbeat.java × Fetcher.java × Reques
506           case Some(group) =>
507             doCommitOffsets(group, memberId, generationId, NO_PRODUCER_ID, NO_PRODUCER_EPOCH,
508               offsetMetadata, responseCallback)
509           }
510       }
511   }
```

doCommitOffsets的实现：

```
dinator.java × KafkaApis.scala × GroupCoordinator.scala × AbstractCoordinator.java × Heartbeat.java × Fetcher.java × RequestFutur
536    */
537   private def doCommitOffsets(group: GroupMetadata,
538                               memberId: String,
539                               generationId: Int,
540                               producerId: Long,
541                               producerEpoch: Short,
542                               offsetMetadata: immutable.Map[TopicPartition, OffsetAndMetadata],
543                               responseCallback: immutable.Map[TopicPartition, Errors] => Unit) {
544     group.inLock {
```

```
556           responseCallback(offsetMetadata.mapValues(_ => Errors.ILLEGAL_GENERATION))
557         } else {
558           val member = group.get(memberId)
559           completeAndScheduleNextHeartbeatExpiration(group, member)
560           groupManager.storeOffsets(group, memberId, offsetMetadata, responseCallback)
561         }
562       }
563     }
```

storeOffsets的实现：

其中：

```
298       }
299       // 计算当前消费组的偏移量应该放到__consumer_offsets的哪个分区
300       val offsetTopicPartition = new TopicPartition(Topic.GROUP_METADATA_TOPIC_NAME, partitionFor(group.groupId))
```

```
307           producerId, producerEpoch, baseSequence = 0, isTxnOffsetCommit
308
309         records.foreach(builder.append)
310         val entries = Map(offsetTopicPartition -> builder.build())
```

```
391
392         appendForGroup(group, entries, putCacheCallback)
```

appendForGroup的实现如下，将当前消费组的偏移量消息追加到`__consumer_offsets`的指定分区中：

```
245   private def appendForGroup(group: GroupMetadata,
246                              records: Map[TopicPartition, MemoryRecords],
247                              callback: Map[TopicPartition, PartitionResponse] => Unit): Unit = {
248     // call replica manager to append the group message
249     replicaManager.appendRecords(
250       timeout = config.offsetCommitTimeoutMs.toLong,
251       requiredAcks = config.offsetCommitRequiredAcks,
252       internalTopicsAllowed = true,
253       isFromClient = false,
254       entriesPerPartition = records,
255       delayedProduceLock = Some(group.lock),
256       responseCallback = callback)
257   }
```

4.18 Kafka源码剖析之同步发送模式

消息同步发送的代码：

所谓同步，就是调用Future的get方法同步等待。

```
40          // 用于封装Producer的消息
41          ProducerRecord<Integer, String> record = new ProducerRecord<Integer, String>(
42                  topic: "topic_1",    // 主题名称
43                  partition: 0,        // 分区编号，现在只有一个分区，所以是0
44                  key: 0,              // 数字作为key
45                  value: "message 0"   // 字符串作为value
46          );
47
48          // 异步发送消息给指定主题
49          final Future<RecordMetadata> future = producer.send(record);
50          // 调用Fugure的get方法同步等待。
51          final RecordMetadata metadata = future.get();
52
```

send方法是异步的：

KafkaProducer.java

```
648     }
649
650     /**
651      * 异步给主题发送消息。等价于send(record, null)
652      * See {@link #send(ProducerRecord, Callback)} for details.
653      */
654     @Override
655     public Future<RecordMetadata> send(ProducerRecord<K, V> record) {
656         return send(record, callback: null);
657     }
658
```

send方法将消息发送给broker，当前线程同步等待broker返回的消息。

send发的实现：

KafkaProducer.java

```
764     @Override
765     public Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback) {
766         // 如果设置了拦截器，调用生产者拦截器们的方法，该方法不抛异常
767         ProducerRecord<K, V> interceptedRecord = this.interceptors == null ? record : this.interceptors.onSend(record);
768         // 调用发送的方法，send名字太多，加个do，即使加do，也不是真的发送
769         return doSend(interceptedRecord, callback);
770     }
```

看doSend：

KafkaProducer.java

```
772     /**
773      * 异步向主题发送记录的实现
774      */
775     private Future<RecordMetadata> doSend(ProducerRecord<K, V> record, Callback callback) {
776         TopicPartition tp = null;
777         try {
778             // first make sure the metadata for the topic is available
```

该方法首先将消息放到累加器中

判断是否需要发起请求，如果需要，则唤醒sender线程发送消息

该方法的返回值：RecordApendResult.future：

KafkaProducer.java

```
813             transactionManager.maybeAddPartitionToTransaction(tp);
814
815         // 将消息追加到累加器中
816         RecordAccumulator.RecordAppendResult result = accumulator.append(tp, timestamp, serializedKey,
817                 serializedValue, headers, interceptCallback, remainingWaitMs);
818         if (result.batchIsFull || result.newBatchCreated) {
819             log.trace("Waking up the sender since topic {} partition {} is either full or getting a new batch
820             // 如果批次写满或新建了消息批，唤醒sender线程发送消息
821             this.sender.wakeup();
822         }
823         return result.future;
```

RecordApendResult类：

KafkaProducer.java　RecordAccumulator.java

```
750     /*
751      * 只是将消息放到记录累加器，就返回该元数据对象
752      */
753     public final static class RecordAppendResult {
754         public final FutureRecordMetadata future;
755         public final boolean batchIsFull;
756         public final boolean newBatchCreated;
757
```

累加器的append方法将消息追加到累加器，并返回追加到累加器的结果：

```java
KafkaProducer.java × RecordAccumulator.java ×
180     */
181    public RecordAppendResult append(TopicPartition tp,
182                                     long timestamp,
183                                     byte[] key,
184                                     byte[] value,
185                                     Header[] headers,
186                                     Callback callback,
187                                     long maxTimeToBlock) throws InterruptedException {
```

其中主要实现：

```java
KafkaProducer.java × RecordAccumulator.java ×
197                 if (closed)
198                     throw new IllegalStateException("Cannot send after the producer is closed.");
199                 RecordAppendResult appendResult = tryAppend(timestamp, key, value, headers, callback, dq);
200                 if (appendResult != null)
201                     return appendResult;
202             }
```

tryAppend的实现：

```java
KafkaProducer.java × RecordAccumulator.java × ProducerBatch.java × FutureRecordMetadata.java ×
103     public FutureRecordMetadata tryAppend(long timestamp, byte[] key, byte[] value, Header[] headers, Callback callback,
104         if (!recordsBuilder.hasRoomFor(timestamp, key, value, headers)) {
105             return null;
106         } else {
107             Long checksum = this.recordsBuilder.append(timestamp, key, value, headers);
108             this.maxRecordSize = Math.max(this.maxRecordSize, AbstractRecords.estimateSizeInBytesUpperBound(magic(),
109                     recordsBuilder.compressionType(), key, value, headers));
110             this.lastAppendTime = now;
111             FutureRecordMetadata future = new FutureRecordMetadata(this.produceFuture, this.recordCount,
112                                                                    timestamp, checksum,
113                                                                    key == null ? -1 : key.length,
114                                                                    value == null ? -1 : value.length);
115             // we have to keep every future returned to the users in case the batch needs to be
116             // split to several new batches and resent.
117             thunks.add(new Thunk(callback, future));
118             this.recordCount++;
119             return future;
```

上述方法的返回值是FutureRecordMetadata，而该类的实现：

```java
KafkaProducer.java × RecordAccumulator.java × ProducerBatch.java × FutureRecordMetadata.java ×
59      @Override
60      public RecordMetadata get() throws InterruptedException, ExecutionException {
61          this.result.await();
62          if (nextRecordMetadata != null)
63              return nextRecordMetadata.get();
64          return valueOrError();
65      }
```

上述方法中，await方法等待broker端返回结果。

result实际上是tryAppend方法赋值的produceFuture对象：

```
111             FutureRecordMetadata future = new FutureRecordMetadata(this.produceFuture, this.recordCount,
112                                                                    timestamp, checksum,
113                                                                    key == null ? -1 : key.length,
114                                                                    value == null ? -1 : value.length);
115             // we have to keep every future returned to the users in case the batch needs to be
```

produceFuture对象是：

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | ProduceRequestResult.java | Futur

```
31  */
32 public final class ProduceRequestResult {
33
34     private final CountDownLatch latch = new CountDownLatch(1);
35     private final TopicPartition topicPartition;
36
37     private volatile Long baseOffset = null;
38     private volatile long logAppendTime = RecordBatch.NO_TIMESTAMP;
```

该类中有一个CountDownLatch，future的get方法中的等待实际上就是该CountDownLatch的等待。

最终我们的producer.send方法的返回值就是FutureRecordMetadata对象。

future.get就是在等待该CountDownLatch的countDown的触发：

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | ProduceRequestResult.java | F

```
66     public void done() {
67         if (baseOffset == null)
68             throw new IllegalStateException("The method `set` must be invoked b
69         this.latch.countDown();
70     }
```

该方法何时调用?

completeFutureAndFireCallbacks方法调用

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | ProduceRequestResult.java | MockProducer.java

```
191     private void completeFutureAndFireCallbacks(long baseOffset, long logAppendTime, RuntimeExce
192         // Set the future before invoking the callbacks as we rely on its state for the `onCompl
193         produceFuture.set(baseOffset, logAppendTime, exception);
194 
195         // execute callbacks
196         for (Thunk thunk : thunks) {
197             try {
198                 if (exception == null) {
199                     RecordMetadata metadata = thunk.future.value();
200                     if (thunk.callback != null)
201                         thunk.callback.onCompletion(metadata,  exception: null);
202                 } else {
203                     if (thunk.callback != null)
204                         thunk.callback.onCompletion( metadata: null, exception);
205                 }
206             } catch (Exception e) {
207                 log.error("Error executing user-provided callback on message for topic-partitio
208             }
209         }
210 
211         produceFuture.done();
212     }
```

（Alt+F7 查看元素的使用位置）

completeFutureAndFireCallbacks方法何时调用?

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | ProduceRequestResult.java | MockProducer.java | Fu

```
166      * @return true if the batch was completed successfully and false if the batch was previously
167      */
168     public boolean done(long baseOffset, long logAppendTime, RuntimeException exception) {
169         final FinalState finalState;
170         if (exception == null) {
171             log.trace("Successfully produced messages to {} with base offset {}.", topicPartition
172             finalState = FinalState.SUCCEEDED;
173         } else {
174             log.trace("Failed to produce messages to {}.", topicPartition, exception);
175             finalState = FinalState.FAILED;
176         }
177 
178         if (!this.finalState.compareAndSet( expect: null, finalState)) {
179             if (this.finalState.get() == FinalState.ABORTED) {
180                 log.debug("ProduceResponse returned for {} after batch had already been aborted."
181                 return false;
182             } else {
183                 throw new IllegalStateException("Batch has already been completed in final state
184             }
185         }
186 
187         completeFutureAndFireCallbacks(baseOffset, logAppendTime, exception);
188         return true;
189     }
```

done方法何时调用?

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | ProduceRequestResult.java | MockProducer.java

579
580     private void completeBatch(ProducerBatch batch, ProduceResponse.PartitionResponse response) {
581         if (transactionManager != null) {
582             if (transactionManager.hasProducerIdAndEpoch(batch.producerId(), batch.producerEpoch())) {
583                 transactionManager.maybeUpdateLastAckedSequence(batch.topicPartition, sequence: batch.base
584                 log.debug("ProducerId: {}; Set last ack'd sequence number for topic-partition {} to {}", b
585                         transactionManager.lastAckedSequence(batch.topicPartition));
586             }
587             transactionManager.updateLastAckedOffset(response, batch);
588             transactionManager.removeInFlightBatch(batch);
589         }
590
591         if (batch.done(response.baseOffset, response.logAppendTime, exception: null))
592             this.accumulator.deallocate(batch);
593     }
```

在completeBatch方法的最后，如果batch.done，则释放累加器的空间。

completeBatch方法何时调用?

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | ProduceRequestResult.java | MockProducer.java | FutureRecor

499      * @param now The current POSIX timestamp in milliseconds
500      */
501     private void completeBatch(ProducerBatch batch, ProduceResponse.PartitionResponse response, long correlationId,
502                                long now) {
503         Errors error = response.error;
504
```

在该方法中:

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | Produ

565
566             } else {
567                 completeBatch(batch, response);
568             }
569
```

completeBatch何时调用?

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | ProduceRequestResult.java

456     /**
457      * 处理生产的响应
458      */
459     private void handleProduceResponse(ClientResponse response, Map<TopicPartition, Produ
460         RequestHeader requestHeader = response.requestHeader();
461         int correlationId = requestHeader.correlationId();
```

在handleProduceResponse中如果有响应，则解析，并调用completeBatch方法

如果没有响应，表示是acks=0的情形，不需要解析响应，直接调用completeBatch方法即可。

```java
log.trace("Received produce response from node {} with correlation id {}", response.destination(), correlationId);
// 如果有响应，解析
if (response.hasResponse()) {
    ProduceResponse produceResponse = (ProduceResponse) response.responseBody();
    for (Map.Entry<TopicPartition, ProduceResponse.PartitionResponse> entry : produceResponse.responses().entrySet()) {
        TopicPartition tp = entry.getKey();
        ProduceResponse.PartitionResponse partResp = entry.getValue();
        ProducerBatch batch = batches.get(tp);
        completeBatch(batch, partResp, correlationId, now);
    }
    this.sensors.recordLatency(response.destination(), response.requestLatencyMs());
} else {
    // 如果没有响应，表示acks=0，直接完成所有的请求即可。
    for (ProducerBatch batch : batches.values()) {
        completeBatch(batch, new ProduceResponse.PartitionResponse(Errors.NONE), correlationId, now);
    }
}
```

handleProduceResponse何时调用?

Sender线程创建回调，回调中调用了handleProduceResponse方法，创建生产请求对象，该对象中封装了回调对象

发送请求，等待回调的触发。

```java
/**
 * 从给定的消息批创建一个生产请求，并发送
 */
private void sendProduceRequest(long now, int destination, short acks, int timeout, List<ProducerBatch> batches) {
    if (batches.isEmpty())
        return;
```

```java
}
// 生产请求的builder
ProduceRequest.Builder requestBuilder = ProduceRequest.Builder.forMagic(minUsedMagic, acks, timeout,
        produceRecordsByPartition, transactionalId);
// 创建回调
RequestCompletionHandler callback = new RequestCompletionHandler() {
    public void onComplete(ClientResponse response) {
        // 如果请求得到响应，则处理生产的响应
        handleProduceResponse(response, recordsByPartition, time.milliseconds());
    }
};

String nodeId = Integer.toString(destination);
// 创建ClientRequest对象，封装了请求数据和回调
ClientRequest clientRequest = client.newClientRequest(nodeId, requestBuilder, now, expectResponse: acks != 0, callback);
// 发送响应
client.send(clientRequest, now);
log.trace("Sent produce request to {}: {}", nodeId, requestBuilder);
}
```

sendProduceRequest的调用：

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | ProduceRequestResult.java | Moc

```
644     * 对每个节点要发送的消息批，封装为每个节点的生产请求对象并发送请求，等待响应
645     */
646    private void sendProduceRequests(Map<Integer, List<ProducerBatch>> collated, long now) {
647        for (Map.Entry<Integer, List<ProducerBatch>> entry : collated.entrySet())
648            sendProduceRequest(now, entry.getKey(), acks, requestTimeout, entry.getValue());
649    }
```

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | Produc

```
240     }
241
242     private long sendProducerData(long now) {
243         Cluster cluster = metadata.fetch();
```

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java

```
306                 // otherwise the select time will be the time differen
307                 pollTimeout = 0;
308             }
309             sendProduceRequests(batches, now);
310
311             return pollTimeout;
```

sendProducerData的调用：

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | Net

```
196     /**
197      * 发送线程的run方法
198      *
199      * @param now The current POSIX time in milliseconds
200      */
201     void run(long now) {
202         if (transactionManager != null) {
```

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | Net

```
236             }
237
238             long pollTimeout = sendProducerData(now);
239             client.poll(pollTimeout, now);
240     }
```

总结：

所谓同步调用，指的是生产者调用`producer.send(record).get()`方法。

该方法首先将要发送的消息发送到消息累加器

判断累加器中的消息批次是否达到，或者当前批次没写满，但是加入当前消息会让消息批大于消息批最大值，则创建新的批次。

如果需要发送消息批次，则唤醒sender线程，让sender线程发送消息。

sender线程会返回一个future对象给生产者客户端线程。

若生产者调用该future的get方法，则该方法使用CountDownLatch阻塞，直到收到broker响应，触发CountDownLatch的countDown方法

此时生产者线程的get方法返回，得到发送的结果。

4.19 Kafka源码剖析之异步发送模式

异步发送消息

在发送消息的时候设置回调函数：

```
27  //        使用回调异步等待消息的确认
28          producer.send(record, new Callback() {
29              @Override
30              public void onCompletion(RecordMetadata metadata, Exception exception) {
31                  if (exception == null) {
32                      System.out.println(
33                              "主题: " + metadata.topic() + "\n"
34                              + "分区: " + metadata.partition() + "\n"
35                              + "偏移量: " + metadata.offset() + "\n"
36                              + "序列化的key字节: " + metadata.serializedKeySize() + "\n"
37                              + "序列化的value字节: " + metadata.serializedValueSize() + "\n"
38                              + "时间戳: " + metadata.timestamp()
39                      );
40                  } else {
41                      System.out.println("有异常: " + exception.getMessage());
42                  }
43              }
44          });
```

调用KafkaProducer的send方法，该方法接收要发送的消息批，同时接收回调对象：

```
764     @Override
765     public Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback) {
766         // 如果设置了拦截器，调用生产者拦截器们的方法，该方法不抛异常
767         ProducerRecord<K, V> interceptedRecord = this.interceptors == null ? record : this.interceptors.onSend(record);
768         // 调用发送的方法，send名字太多，加个do，即使加do，也不是真的发送
769         return doSend(interceptedRecord, callback);
770     }
```

doSend的实现：

```java
log.trace("Sending record {} with callback {} to topic {} partition {}", record, callback, record.topic(), partition);
// 如果设置了拦截器，则回调之前需要先经过拦截器，否则直接调用用户设置的回调
Callback interceptCallback = this.interceptors == null ? callback : new InterceptorCallback<>(callback, this.interceptors, tp);

if (transactionManager != null && transactionManager.isTransactional())
    transactionManager.maybeAddPartitionToTransaction(tp);

// 将消息追加到累加器中
RecordAccumulator.RecordAppendResult result = accumulator.append(tp, timestamp, serializedKey,
        serializedValue, headers, interceptCallback, remainingWaitMs);
if (result.batchIsFull || result.newBatchCreated) {
    log.trace("Waking up the sender since topic {} partition {} is either full or getting a new batch", record.topic(), partitio
    // 如果批次写满或新建了消息批，唤醒sender线程发送消息
    this.sender.wakeup();
}
```

累加器append的实现：

```java
throw new IllegalStateException("Cannot send after the producer is closed.");

RecordAppendResult appendResult = tryAppend(timestamp, key, value, headers, callback, dq);
if (appendResult != null) {
    // Somebody else found us a batch, return the one we waited for! Hopefully this doesn't h
    return appendResult;
}
```

tryAppend的实现：

```java
private RecordAppendResult tryAppend(long timestamp, byte[] key, byte[] value, Header[] headers,
                                     Callback callback, Deque<ProducerBatch> deque) {
    ProducerBatch last = deque.peekLast();
    if (last != null) {
        FutureRecordMetadata future = last.tryAppend(timestamp, key, value, headers, callback, time.milliseconds());
        if (future == null)
            last.closeForRecordAppends();
        else
```

tryAppend的实现：

```java
     * @return The RecordSend corresponding to this record or null if there isn't sufficient room.
     */
public FutureRecordMetadata tryAppend(long timestamp, byte[] key, byte[] value, Header[] headers, Callback callback, long now) {
    if (!recordsBuilder.hasRoomFor(timestamp, key, value, headers)) {
        return null;
    } else {
        Long checksum = this.recordsBuilder.append(timestamp, key, value, headers);
        this.maxRecordSize = Math.max(this.maxRecordSize, AbstractRecords.estimateSizeInBytesUpperBound(magic(),
                recordsBuilder.compressionType(), key, value, headers));
        this.lastAppendTime = now;
        FutureRecordMetadata future = new FutureRecordMetadata(this.produceFuture, this.recordCount,
                                                               timestamp, checksum,
                                                               key == null ? -1 : key.length,
                                                               value == null ? -1 : value.length);
        // we have to keep every future returned to the users in case the batch needs to be
        // split to several new batches and resent.
        thunks.add(new Thunk(callback, future));
        this.recordCount++;
        return future;
```

Sender的run方法调用：

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | Netwo
200     */
201    void run(long now) {
202        if (transactionManager != null) {
203            try {
204                if (transactionManager.shouldResetProducerStateAfterResc
205                    // Check if the previous run expired batches which r
```

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java
238            long pollTimeout = sendProducerData(now);
239            client.poll(pollTimeout, now);
240        }
241
```

sendProducerData的实现：

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java
241
242    private long sendProducerData(long now) {
243        Cluster cluster = metadata.fetch();
244
```

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java
308            }
309            sendProduceRequests(batches, now);
310
311            return pollTimeout;
```

sendProducerRequests的实现：

```
KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | NetworkClient.java | ProduceReques
643     /**
644      * 对每个节点要发送的消息批，封装为每个节点的生产请求对象并发送请求，等待响应
645      */
646     private void sendProduceRequests(Map<Integer, List<ProducerBatch>> collated, long now) {
647         for (Map.Entry<Integer, List<ProducerBatch>> entry : collated.entrySet())
648             sendProduceRequest(now, entry.getKey(), acks, requestTimeout, entry.getValue());
649     }
650
```

sendProduceRequest的实现：

```
653     */
654    private void sendProduceRequest(long now, int destination, short acks, int timeout, List<ProducerBatch> batches) {
655        if (batches.isEmpty())
656            return;
657
```

```
689        // 生产请求的builder
690        ProduceRequest.Builder requestBuilder = ProduceRequest.Builder.forMagic(minUsedMagic, acks, timeout,
691                produceRecordsByPartition, transactionalId);
692        // 创建回调
693        RequestCompletionHandler callback = new RequestCompletionHandler() {
694            public void onComplete(ClientResponse response) {
695                // 如果请求得到响应，则处理生产的响应
696                handleProduceResponse(response, recordsByPartition, time.milliseconds());
697            }
698        };
699
700        String nodeId = Integer.toString(destination);
701        // 创建ClientRequest对象，封装了请求数据和回调
702        ClientRequest clientRequest = client.newClientRequest(nodeId, requestBuilder, now, expectResponse: acks != 0, callback);
703        // 发送响应
704        client.send(clientRequest, now);
```

上述方法如果得到broker的响应，就回调`handleProduceResponse`方法：

```
458     */
459    private void handleProduceResponse(ClientResponse response, Map<TopicPartition, ProducerBatch> batches, long now) {
460        RequestHeader requestHeader = response.requestHeader();
461        int correlationId = requestHeader.correlationId();
```

该方法对响应的处理：

```
473            log.trace("Received produce response from node {} with correlation id {}", response.destination(), correlationId);
474            // 如果有响应，解析
475            if (response.hasResponse()) {
476                ProduceResponse produceResponse = (ProduceResponse) response.responseBody();
477                for (Map.Entry<TopicPartition, ProduceResponse.PartitionResponse> entry : produceResponse.responses().entrySet()) {
478                    TopicPartition tp = entry.getKey();
479                    ProduceResponse.PartitionResponse partResp = entry.getValue();
480                    ProducerBatch batch = batches.get(tp);
481                    completeBatch(batch, partResp, correlationId, now);
482                }
483                this.sensors.recordLatency(response.destination(), response.requestLatencyMs());
484            } else {
485                // 如果没有响应，表示acks=0，直接完成所有的请求即可。
486                for (ProducerBatch batch : batches.values()) {
487                    completeBatch(batch, new ProduceResponse.PartitionResponse(Errors.NONE), correlationId, now);
488                }
489            }
```

completeBatch的实现：

```
500     */
501    private void completeBatch(ProducerBatch batch, ProduceResponse.PartitionResponse response, long correlationId,
502                               long now) {
503        Errors error = response.error;
504        // 消息太大并且批中消息个数>1，并且如果是V2版本并且经过压缩了
505        if (error == Errors.MESSAGE_TOO_LARGE && batch.recordCount > 1 &&
```

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java

```
567         } else {
568             completeBatch(batch, response);
569         }
570
```

completeBatch的实现：

KafkaProducer.java | RecordAccumulator.java | ProducerBatch.java | Sender.java | NetworkClient.java | Produce

```
580
581     private void completeBatch(ProducerBatch batch, ProduceResponse.PartitionResponse respo
582         if (transactionManager != null) {
583             if (transactionManager.hasProducerIdAndEpoch(batch.producerId(), batch.produce
584                 transactionManager.maybeUpdateLastAckedSequence(batch.topicPartition, sequ
585                 log.debug("ProducerId: {}; Set last ack'd sequence number for topic-partit
586                         transactionManager.lastAckedSequence(batch.topicPartition));
587             }
588             transactionManager.updateLastAckedOffset(response, batch);
589             transactionManager.removeInFlightBatch(batch);
590         }
591
592         if (batch.done(response.baseOffset, response.logAppendTime, exception: null))
593             this.accumulator.deallocate(batch);
594     }
```

batch的done方法：

```java
public boolean done(long baseOffset, long logAppendTime, RuntimeException exception) {
    final FinalState finalState;
    if (exception == null) {
        log.trace("Successfully produced messages to {} with base offset {}.", topicPartition, ba
        finalState = FinalState.SUCCEEDED;
    } else {
        log.trace("Failed to produce messages to {}.", topicPartition, exception);
        finalState = FinalState.FAILED;
    }

    if (!this.finalState.compareAndSet( expect: null, finalState)) {
        if (this.finalState.get() == FinalState.ABORTED) {
            log.debug("ProduceResponse returned for {} after batch had already been aborted.", to
            return false;
        } else {
            throw new IllegalStateException("Batch has already been completed in final state " +
        }
    }

    completeFutureAndFireCallbacks(baseOffset, logAppendTime, exception);
    return true;
}
```

触发回调函数的执行：

```java
private void completeFutureAndFireCallbacks(long baseOffset, long logAppendTime, RuntimeException exception) {
    // Set the future before invoking the callbacks as we rely on its state for the `onCompletion` call
    produceFuture.set(baseOffset, logAppendTime, exception);

    // execute callbacks
    for (Thunk thunk : thunks) {
        try {
            if (exception == null) {
                RecordMetadata metadata = thunk.future.value();
                if (thunk.callback != null)
                    thunk.callback.onCompletion(metadata, exception: null);
            } else {
                if (thunk.callback != null)
                    thunk.callback.onCompletion( metadata: null, exception);
            }
        } catch (Exception e) {
```

上图中执行用户设置的callback函数的onCompletion方法：

```java
//        使用回调异步等待消息的确认
        producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata metadata, Exception exception) {
                if (exception == null) {
                    System.out.println(
                                "主题: " + metadata.topic() + "\n"
                                + "分区: " + metadata.partition() + "\n"
                                + "偏移量: " + metadata.offset() + "\n"
                                + "序列化的key字节: " + metadata.serializedKeySize() + "\n"
                                + "序列化的value字节: " + metadata.serializedValueSize() + "\n"
                                + "时间戳: " + metadata.timestamp()
                    );
                } else {
                    System.out.println("有异常: " + exception.getMessage());
                }
            }
        });
```

由于上述方法都是在Sender线程中调用，因此回调的onCompletion方法的执行也是异步的，跟用户的producer.send方法不在同一个线程。

回调的异步执行即是生产的异步发送模式。