kafka入门

目录

1 kfk概述

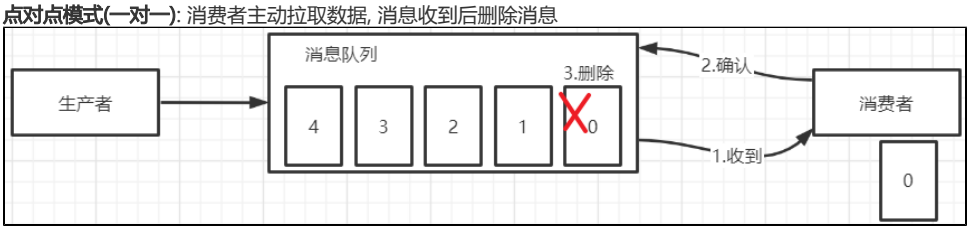
1.1 什么是kfk

- kfk是一个的基于的. 主要应用于大数据实时处理领域.

1.2 消息队列的两种模式

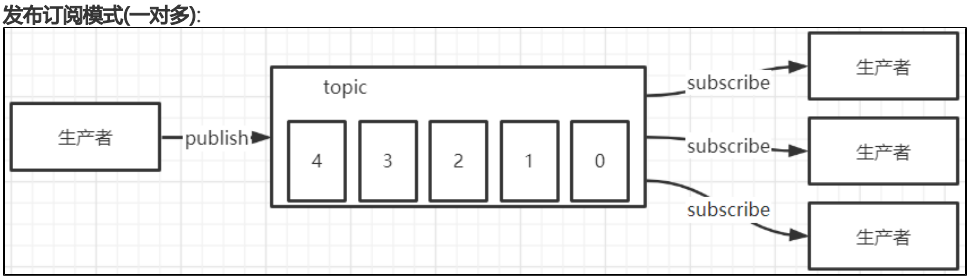
- 消息队列有两种模式: 点对点模式 和 发布订阅模式. kfk是基于发布订阅模式的消息队列.

- **点对点模式(一对一)**: 消费者主动拉取数据, 消息收到后删除消息



  生产者生产消息发送到queen中, 然后消费者从queen中数据并消费.
  queen, 所以消费者不可能消费到已经被消费的消息.
  queen支持存在多个消费者, 但是对一个消息而言, 只能被一个消费者消费.

- **发布订阅模式(一对多)**:



  生产者将消息发布到topic, 多个消费者订阅该topic. 发布到topic的消息会被所有的的订阅者消费.
  topic作用是进行消息分类, 例如订单相关消息放到订单topic, 用户相关消息放到用户topic.
  订阅topic类似订阅微信公众号.

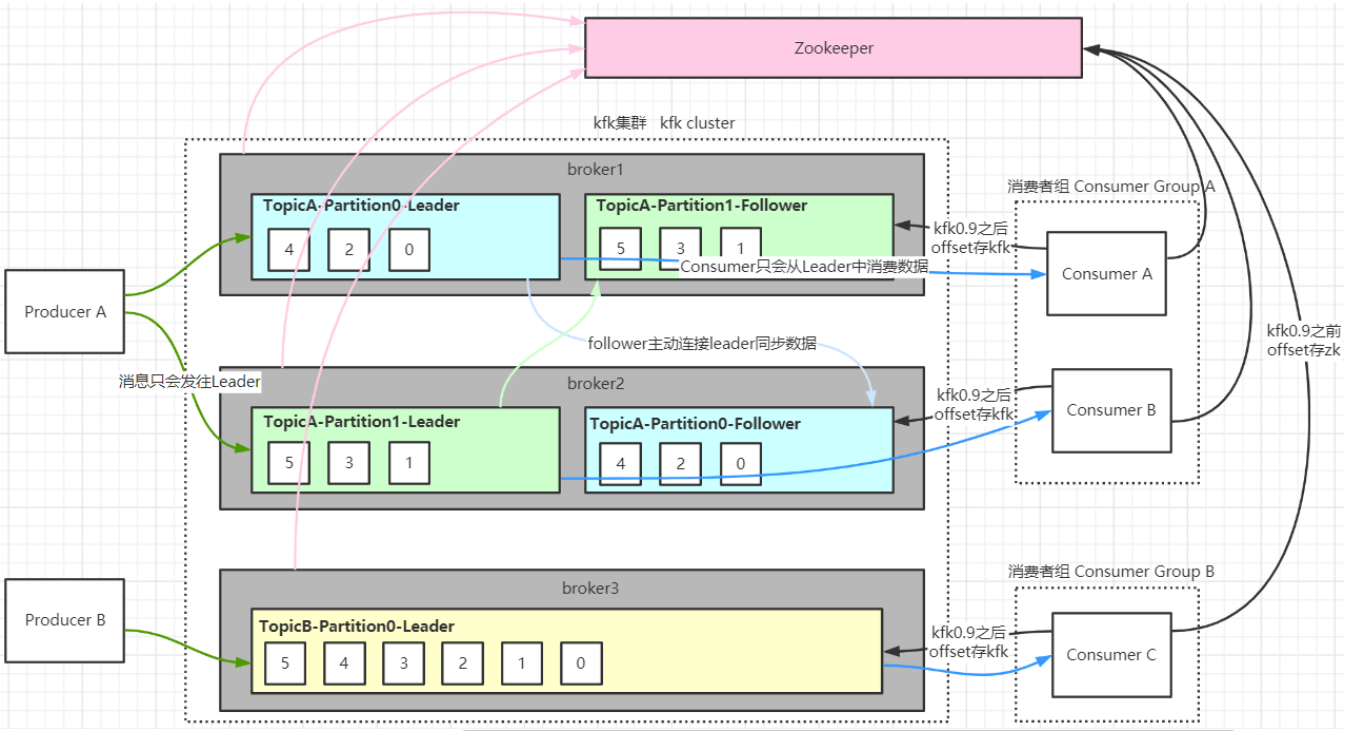
- 发布订阅模式下又分为两种模式, 和 . kafka.

- 发布订阅模式中消费者主动拉取数据的模式的**优点**
  - 1 同样的消息可以发送给多个消费者(这是发布订阅模式的优点).
  - 2 (如果是队列主动推送消息到消费者, 假如队列 每秒推送50M数据, 消费者每秒消费10M数据, 那么消费者就崩了, 每个消费者的消费能力是不 同的).

- 发布订阅模式中消费者主动拉取数据的模式的**缺点**
  - 1 (消费者主动拉取消息, 那么消费者必须一直轮询队列判断是否有新消息, 即便是队列中没有新消息时也要寻轮询问队列是否有消息. 即维护一个长轮询, 不断询问是否 有新消息).

- 注意: 根据上面两点可以知道点对点模式的优缺点.
- 注意: "发布订阅模式中消费者主动拉取数据的模式"中, 缺点是需要消费者维护一个长轮询, 不断询问mq是否有新消息. 那么能不能mq中收到消息之后通知消费者, 这时让消费者再主动拉取呢?
  可以,但是这样也是有缺点的. 优点就是消费者不用维护和mq的长轮询了. 缺点就是mq需要维护一个列表, 这个列表中放的是所有要订阅这个topic的消费者. 还有一个缺点是如果消费者挂了, mq就通知不到了.
- 注意: 消息队列并不是文件存储系统, , 不能无限期保存数据, 期限可配置. kfk中数据默认保存168小时, 即7天. kafka存消息存到磁盘中, 即能持久化.

1.3 kfk的架构



- 每个broker就是一个kfk服务, 就是一个kfk进程. 多个broker构成一个kfk集群. 一个broker可以容纳多个topic.
- topic作用是消息分类. 例如订单相关消息存到订单topic, 用户相关消息存到用户topic.
- 一个topic可以分为多个`partition`, 一个topic可以分布到多个broker中. **分区的作用是提高topic的负载能力**, 可以说是提高读写并发度.
- `replica`. 为了保证broker发生故障时, 该节点中的partition数据不会丢失且kfk能继续工作, kfk提供了副本机制, 一个topic的每个partition都可以有多个replica. 一个leader和若干个follower, 说一个partition有3个replica, 说明这个partition有1个leader和2个follower.
  分区的副本的作用是做高可用. lader挂掉会将follower提升为leader. 同一个partition的leader和follower肯定不能在同一个broker(kafka服务器)中, 否则不能高可用. 无论是生产者还是消费者, 无论是发送消息还是接受消息, 只会找leader, follower只是起到备份作用.
- `consumer group(CG)`. 由多个consumer组成. 消费者组中每个消费者负责消费不同partition的数据, 一个partition只能由一个组内的消费者消费. 消费者组之间互不影响. 消费者肯定属于某个消费者组. 消费者组是一个逻辑上的订阅者.

  这里面有一个消费者组中的分区分配策略问题. 生产者也有一个消息发送到分区的分配策略问题.

  某一个partition分区中的数据只能被同一个消费者组中的某一台消费者消费. 但是可以有多个消费者组中的一台消费者消费同一个partition分区中的数据. 我们可以将同一个组的所有消费者当做一个大的消费者.

  消费者组的作用是提高消费者的消费能力. 假如一个topic两个partition, 每个partition中各有50条数据, 只有一台消费者, 那么一台消费者需要消费50条数据. 假如一个topic两个partition, 每个partition中有50条数据, 两台消费者, 那么每台消费者可以消费50条数据, 速度快了一倍. 两台消费者必须同一个组(消费者集群中的所有节点作为一个mq消费者组), 因为两个partition属于同一个topic. 消费者组中消费者的个数多于topic的partition分区数没有意义, 最好消费者组中的消费者个数和topic中的partition分区数相同.
- zk在kfk中的作用: 帮kafka集群存信息, 帮消费者存储消费数据的位置偏移量offset(看kfk版本).

  zk帮助我们管理kafka集群. 如果想使多台kafka作为一个集群, 只需要让这多台kafka使用同一个zk即可, 当然zk可以是集群.

  消费者也要向zk中存储一些数据. 比如一个partition里面10条消息, 一台消费者消费到5条的时候挂了, 那么消费者重启需要从第6条消息开始消费, 消费者消费数据的位置信息就需要存到zk中. 肯定不能只 保存在消费者的内存中啊, 消费者挂了就操蛋了. 消费者内存中也是有一份消费数据的位置信息(offset偏移量)的.

  注意, kafka0.9版本前, 消费者消费消息的offset存在zk中; kafka0.9版本及之后, 消费者消费消息的offset存在kafka中, 存在kafka内置的topic中; 无论消费者消费消息的offset存在哪里, 作用都是记录消费位置, 以便当消费者挂掉后重启后可以接着消费消息.

  为什么要改呢, 肯定是存在zk中不好. 消费者和kfk之间维持连接, 还要和zk之间维持连接, 消费者拉取消息是很快的, 那么就需要消费者频繁和zk交互, 为了避免这种情况就改了.

2 kfk入门

2.1 安装部署kfk


```
#
[root@king kafka]# tar -axvf kafka_2.11-0.11.0.0.tgz
#
[root@king kafka]# mv kafka_2.11-0.11.0.0 kafka

[root@king kafka]# pwd
/opt/kafka/kafka
[root@king kafka]# ll
total 52
drwxr-xr-x 3 root root  4096 Jun 23  2017 bin
drwxr-xr-x 2 root root  4096 Jun 23  2017 config
drwxr-xr-x 2 root root  4096 Nov 22 23:18 libs
-rw-r--r-- 1 root root 28824 Jun 23  2017 LICENSE
-rw-r--r-- 1 root root   336 Jun 23  2017 NOTICE
drwxr-xr-x 2 root root  4096 Jun 23  2017 site-docs

[root@king config]# pwd
/opt/kafka/kafka/config
[root@king config]# ll
total 64
-rw-r--r-- 1 root root  906 Jun 23  2017 connect-console-sink.properties
-rw-r--r-- 1 root root  909 Jun 23  2017 connect-console-source.properties
-rw-r--r-- 1 root root 5807 Jun 23  2017 connect-distributed.properties
-rw-r--r-- 1 root root  883 Jun 23  2017 connect-file-sink.properties
-rw-r--r-- 1 root root  881 Jun 23  2017 connect-file-source.properties
-rw-r--r-- 1 root root 1111 Jun 23  2017 connect-log4j.properties
-rw-r--r-- 1 root root 2730 Jun 23  2017 connect-standalone.properties
-rw-r--r-- 1 root root 1199 Jun 23  2017 consumer.properties
-rw-r--r-- 1 root root 4696 Jun 23  2017 log4j.properties
-rw-r--r-- 1 root root 1900 Jun 23  2017 producer.properties
-rw-r--r-- 1 root root 6954 Jun 23  2017 server.properties
-rw-r--r-- 1 root root 1032 Jun 23  2017 tools-log4j.properties
-rw-r--r-- 1 root root 1023 Jun 23  2017 zookeeper.properties

# :
server.properties
zookeeper.properties
```

- /opt/kafka/kafka/config/server.properties

```
############################# Server Basics #############################
# kfkid, kfk
broker.id=0
# topic, , false. true, .
delete.topic.enable=true
############################# Socket Server Settings #############################
# kfk9092, 9091
listeners=PLAINTEXT://:9091
# kfk. PLAINTEXT://kfkip:kfk
advertised.listeners=PLAINTEXT://47.111.177.219:9091
############################# Log Basics #############################
# kfk. kfk.
# kfk.log, kfk.log
# , .
log.dirs=/tmp/kafka-logs
############################# Log Retention Policy #############################
# kfk()7
```

```
log.retention.hours=168
# kfk, 1G
log.segment.bytes=1073741824
############################# Zookeeper #############################
# zk, zk"127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
zookeeper.connect=localhost:2181
```

完成上述配置, 先启动zk, 再自动kfk.

- kfk的启动命令 /opt/kafka/kafka/bin

```
-rwxr-xr-x 1 root root 1335 Jun 23  2017 connect-distributed.sh
-rwxr-xr-x 1 root root 1332 Jun 23  2017 connect-standalone.sh
-rwxr-xr-x 1 root root  861 Jun 23  2017 kafka-acls.sh
-rwxr-xr-x 1 root root  873 Jun 23  2017 kafka-broker-api-versions.sh
-rwxr-xr-x 1 root root  864 Jun 23  2017 kafka-configs.sh
# 下面两个是控制台的生产者和消费者. 一般做测试用. 因为控制台消费数据也就是打印出来字符串而已.
-rwxr-xr-x 1 root root  945 Jun 23  2017 kafka-console-consumer.sh
-rwxr-xr-x 1 root root  944 Jun 23  2017 kafka-console-producer.sh
-rwxr-xr-x 1 root root  871 Jun 23  2017 kafka-consumer-groups.sh
-rwxr-xr-x 1 root root  872 Jun 23  2017 kafka-consumer-offset-checker.sh
# 消费者的测试 压力测试时候用
-rwxr-xr-x 1 root root  948 Jun 23  2017 kafka-consumer-perf-test.sh
-rwxr-xr-x 1 root root  869 Jun 23  2017 kafka-delete-records.sh
-rwxr-xr-x 1 root root  862 Jun 23  2017 kafka-mirror-maker.sh
-rwxr-xr-x 1 root root  886 Jun 23  2017 kafka-preferred-replica-
election.sh
# 生产者的测试 压力测试时候用
-rwxr-xr-x 1 root root  959 Jun 23  2017 kafka-producer-perf-test.sh
-rwxr-xr-x 1 root root  874 Jun 23  2017 kafka-reassign-partitions.sh
-rwxr-xr-x 1 root root  868 Jun 23  2017 kafka-replay-log-producer.sh
-rwxr-xr-x 1 root root  874 Jun 23  2017 kafka-replica-verification.sh
-rwxr-xr-x 1 root root 7027 Jun 23  2017 kafka-run-class.sh
# 下面两个是 启动/关闭 kfk服务的命令
-rwxr-xr-x 1 root root 1376 Jun 23  2017 kafka-server-start.sh
-rwxr-xr-x 1 root root  975 Jun 23  2017 kafka-server-stop.sh
-rwxr-xr-x 1 root root  870 Jun 23  2017 kafka-simple-consumer-shell.sh
-rwxr-xr-x 1 root root  945 Jun 23  2017 kafka-streams-application-reset.sh
# 关于topic的crud都使用该命令
-rwxr-xr-x 1 root root  863 Jun 23  2017 kafka-topics.sh
-rwxr-xr-x 1 root root  958 Jun 23  2017 kafka-verifiable-consumer.sh
-rwxr-xr-x 1 root root  958 Jun 23  2017 kafka-verifiable-producer.sh
drwxr-xr-x 2 root root 4096 Jun 23  2017 windows
-rwxr-xr-x 1 root root  867 Jun 23  2017 zookeeper-security-migration.sh
-rwxr-xr-x 1 root root 1393 Jun 23  2017 zookeeper-server-start.sh
-rwxr-xr-x 1 root root  978 Jun 23  2017 zookeeper-server-stop.sh
-rwxr-xr-x 1 root root  968 Jun 23  2017 zookeeper-shell.sh
```

- kfk启动/停止 单台

```
# kfk. -daemon
/opt/kafka/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/kafka/config/server-1.properties
/opt/kafka/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/kafka/config/server-2.properties
/opt/kafka/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/kafka/config/server-3.properties
# kfk
/opt/kafka/kfk/bin/kafka-server-stop.sh
```

可以编写shell脚本群启kfk.

三台broker只要连接同一个zk集群, 就是同一个kfk集群中的节点, 注意三台broker中的`broker.id`不能相同.

2.2 kfk的命令行操作

2.2.1 命令行操作topic

- 创建topic并指定分区和副本

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic topic01
--partitions 2 --replication-factor 2
```

创建topic. 使用--create参数. 需要指定zk地址.

--topic topic01 # 主题的名字. 主题的作用是帮我们分类消息.
--partitions 2 # 两个分区.
--replication-factor 2 # 每个分区有两个副本.

创建同名topic将报错;

- 查看topic

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
```

查看topic将打印topic的名字.

- kfk集群和消费者需要向zk中存数据, 因此需要指定zk地址. 如果操作生产者则无需指定zk地址;
kfk0.9前消费者将消息偏移量存入zk, kfk0.9开始消费者将offset存入broker.

- 创建topic后, 存储生产者发送过来的数据的文件
  - 我在/opt/kafka/kafka/config/中创建了三个kfk配置文件部署了伪集群: server-1.properties和server-2.properties和server-3.properties.
  - 在server-*.properties中配置了存放kfk日志和数据的目录(在kfk中数据也叫日志): log.dirs=/tmp/kafka-logs-1和log.dirs=/tmp/kafka-logs-2和log.dirs=/tmp/kafka-logs-3.
  - /tmp/kafka-logs-1目录中有topic01-0和topic01-1; /tmp/kafka-logs-2目录中有topic01-1; /tmp/kafka-logs-3目录中有topic01-0;
  - 可见我们创建了一个topic叫topic01, 这个topic有2个分区, 每个分区有两个副本; 两个分区就是topic01-0和topic01-1. topic01-0和topic01-1各有两个副本, 一个是leader一个是follower; 0和1是分区号;

- kfk的日志(非数据)文件
  - kfk日志文件目录是/opt/kafka/kafka/logs/server.log; 如果使用jps看不到kfk进程, 查看该文件排查问题;
  - 找不到日志文件就使用命令find / -name server.log获取日志文件路径;

- 删除名字为first的topic

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-topics.sh --delete --zookeeper localhost:2181 --topic first
```

执行上述命令可能提示下面两行:
Topic first is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
意思是名字为first的topic已被标记删除;

只有当delete.topic.enable设置为true时topic才会被物理删除;

经过测试, 实际上无论delete.topic.enable设置为t还是f, 都会有该提示;

- 查看一个topic详情

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-topics.sh --describe --topic topic01 --zookeeper localhost:2181
```

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-topics.sh --describe --topic topic01 --zookeeper localhost:2181
Topic:topic01   PartitionCount:2        ReplicationFactor:2     Configs:
        Topic: topic01  Partition: 0    Leader: 3       Replicas: 3,1   Isr: 3,1
        Topic: topic01  Partition: 1    Leader: 1       Replicas: 1,2   Isr: 1,2
[root@king ~]#
```

因为有两个副本, 所以Replicas后面跟了两个数字.

- 注意kfk副本数不能超过集群节点数(broker), 否则创建失败报错; 一个分区的多个副本不能同时存在同一个broker中, 否则不能高可用;
kfk的分区数(partition)可以超过集群节点数(broker), 这样做时某个broker中会有多个分区;

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic topic02 --partitions 2 --replication-factor 4
Error while executing topic command : replication factor: 4 larger than available brokers: 3
[2020-11-28 16:42:29,274] ERROR org.apache.kafka.common.errors.InvalidReplicationFactorException: replication factor: 4 larger than available brokers: 3
 (kafka.admin.TopicCommand$)
[root@king ~]#
```

一共3个broker我创建4个副本, topic创建失败报错

2.2.2 命令行测试生产者消费者

- 启动控制台的生产者

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-console-producer.sh --topic topic01 --broker-list localhost:
9091
```

启动生产者时, 需要往指定的topic中发送数据, 因此指定了topic; 需要指定broker; 生产者不和zk交互, 无需指定zk;

- 启动控制台的消费者 -- kfk0.9前

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-console-consumer.sh --topic topic01 --zookeeper localhost:2181
Using the ConsoleConsumer with old consumer is deprecated and will be removed in a future major release.
Consider using the new consumer by passing [bootstrap-server] instead of [zookeeper].
```

启动消费者时, 需要指定从哪个topic消费数据, 因此指定了topic; kfk0.9之后消费者不将offset存到zk, 因此我们指定zk会提示已过时;

- `--from-beginning`参数
  - 指定该参数可以让新启动的消费者从头开始消费数据, 否则只有当生产者再次生产数据时新启动的消费者才能消费数据; kfk中数据默认存放7天;

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-console-consumer.sh --topic topic01 --zookeeper
localhost:2181 --from-beginning
```

- 启动控制台的消费者 -- kfk0.9开始

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-console-consumer.sh --topic topic01 --bootstrap-server
localhost:9091 --from-beginning
```

kfk0.9开始消费者将offset存到broker而不是zk, 因此不指定zk而使用`--bootstrap-server`参数指定kfk.

- 消费指定topic指定partition中的数据，显示数据的key和value

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-console-consumer.sh --topic topicName --bootstrap-server
localhost:9092 --property print.key=true --partition 0 --from-beginning
```

- 查看`/tmp/kafka-logs-1`和`/tmp/kafka-logs-2`和`/tmp/kafka-logs-2`

```
[root@king tmp]# cd kafka-logs-1/
[root@king kafka-logs-1]# ll
total 116
-rw-r--r-- 1 root root    0 Nov 28 15:39 cleaner-offset-checkpoi
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-0
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-12
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-15
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-18
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-21
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-24
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-27
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-3
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-30
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-33
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-36
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-39
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-42
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-45
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-48
drwxr-xr-x 2 root root 4096 Nov 28 15:39 __consumer_offsets-6
```

  - 可见多了很多`__consumer_offsets-`, 实际上`__consumer_offsets`是kfk内置的topic, 数字代表partition分区编号, 实际该topic默认分区数50, 每个分区只有1个副本;
  - 我搭建的是3个broker的伪集群. 这50个partition均匀分布在3台broker中;
  - 小结: kfk中存数据的地方叫topic, 消费者消费消息的位置(消费消息的偏移量offset)在kfk0.9后存储在kfk的内置topic中, 这个topic叫做__consumer_offsets, 默认50个partition, 每个分区1个副本.

- 注意
  - 在kfk存储数据的目录中有两个文件: recovery-point-offset-checkpoint 和 replication-offset-checkpoint
    - recovery-point-offset-checkpoint: 已经被确认写入磁盘的offset

- replication-offset-checkpoint: 已经确认复制给其他replica的offset。也就是HW。(用来存储每一个replica的HighWatermark。由ReplicaManager负责写。)

- 注意
  - 如果往一个不存在的topic发送消息, 也能发送成功. kfk会自动创建该topic, 该topic默认有1个分区1个副本; 可在server.properties中配置kfk自动创建topic的分区数和副本数;
  - 如果指定每个分区2个副本, 指的是leader+follower一共2个, 并不是1个leader和2个follower;
  - 同一个partition的leader和follower不能在同一个broker中, 同一个partition的多个follower也不能在同一个broker中;

2.3 kfk的日志和数据分目录存放

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

# A comma seperated list of directories under which to store log files
log.dirs=/opt/module/kafka/data
```

指定kfk的数据存放在此

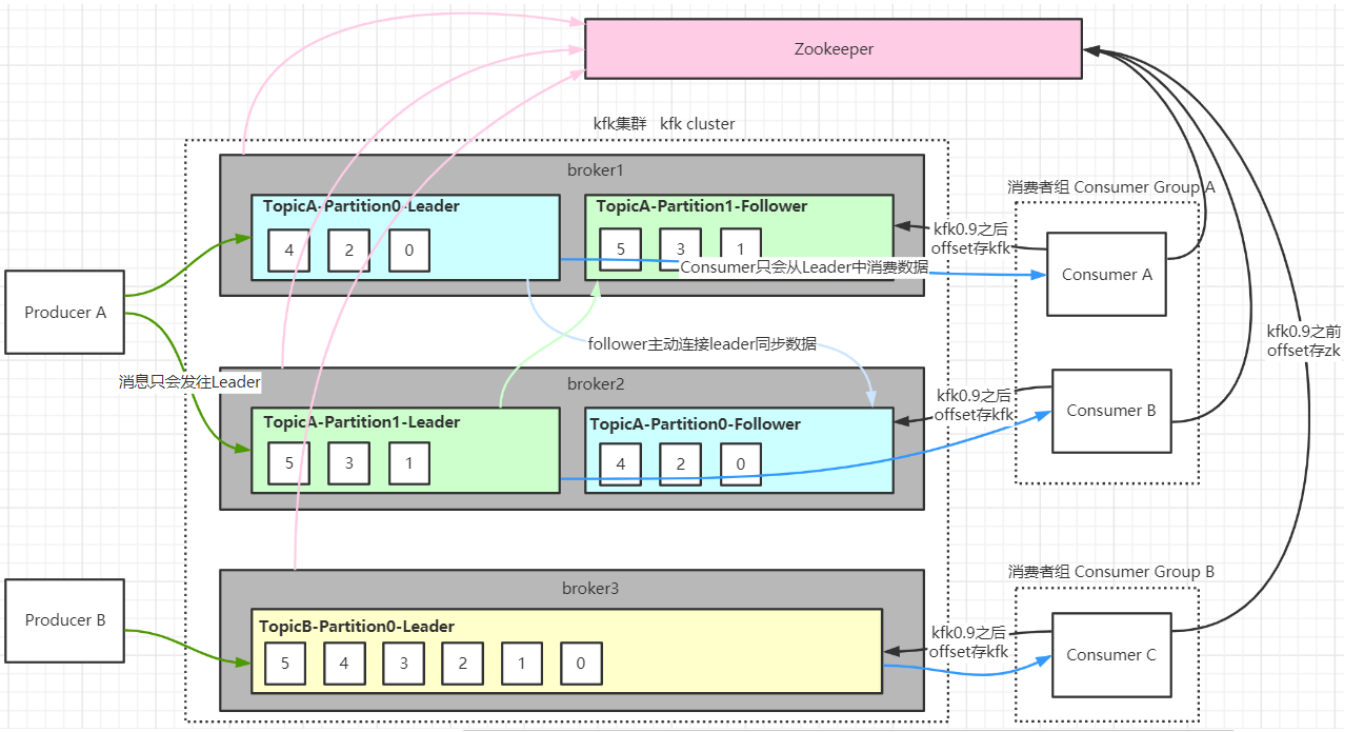
注意data目录需要手动创建一下; logs目录不需要手动创建;

```
[root@king topic01-0]# pwd
/tmp/kafka-logs-1/topic01-0
[root@king topic01-0]# ll
total 8
-rw-r--r-- 1 root root 10485760 Nov 28 15:48 00000000000000000000.index
-rw-r--r-- 1 root root       74 Nov 28 17:05 00000000000000000000.log
-rw-r--r-- 1 root root 10485756 Nov 28 15:48 00000000000000000000.timeindex
-rw-r--r-- 1 root root        8 Nov 28 17:05 leader-epoch-checkpoint
```

log中存的是生产者发来的数据

3 kfk架构深入

3.1 kfk工作流程



- 上面每个分区中的数字代表消息的偏移量(offset), 所有的分区没有全局的偏移量, 每个分区维护以及的消息偏移量;
  kfk0.9开始消费者中存储offset, kfk0.9之前zk中存储offset;

- follower主动连接leader同步数据. 可能出现数据丢失;

- kfk中消息按topic分类, 生产者需指定往哪个topic发消息, 消费者需指定从哪个topic消费消息;
  topic是逻辑上的概念, partition是物理上的概念. 因此磁盘中找不到topic对应的目录, 只能找到partition对应的目录.
  partition对应目录的名字是"主题名-分区编号";

- 每个partition对应一个.log文件, 该文件中存储生产者发送的消息. 生产者生产的消息会被不断追加到该文件末尾, 并且每条消息都有自己的offset(每条消息自己的偏移量, 可以理解为这条消息占几行). 消费者组中的每个消费者都会实时记录自己消费到了哪个offset(消息的偏移量, 可以理解为第几条消息), 以便消费出错恢复时继续从上次的位置继续消费(从上条消息开始继续消费).

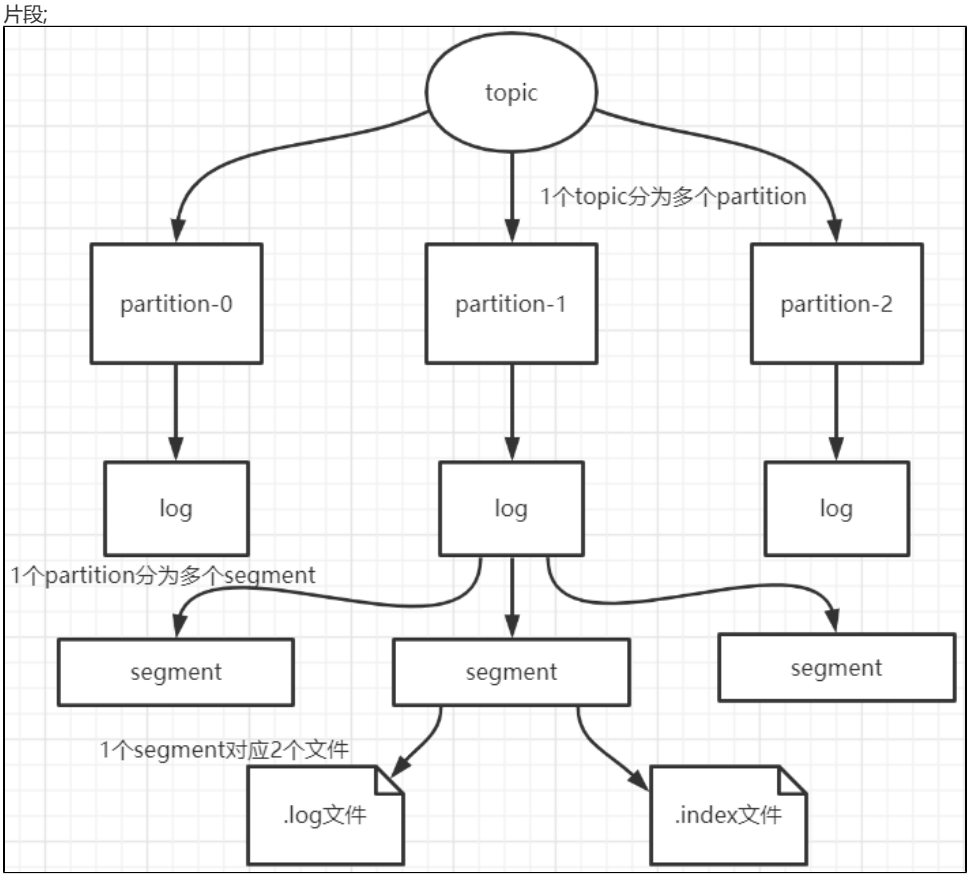
3.2 kfk文件存储机制

```
[root@king topic01-0]# pwd
/tmp/kafka-logs-1/topic01-0  topic为topic01中第0个partition 中存放数据的目录
[root@king topic01-0]# ll
total 8
-rw-r--r-- 1 root root 10485760 Nov 28 15:48 00000000000000000000.index
-rw-r--r-- 1 root root       74 Nov 28 17:05 00000000000000000000.log
-rw-r--r-- 1 root root 10485756 Nov 28 15:48 00000000000000000000.timeindex
-rw-r--r-- 1 root root        8 Nov 28 17:05 leader-epoch-checkpoint
```

- 00000000000000000000.log用来存放生产者发送的消息. 默认存7天. 该文件最大1G, 超过1G将生成一个新的.log文件.
- .log文件名是存储的第一条消息的offset(第一条消息的行数). 第一个.log文件的名字是0000, 可见消息的偏移量是从0开始计算而不是1;

```
######### Log Retention Policy #########
log.retention.hours=168 # 7
log.segment.bytes=1073741824 # 1G
```

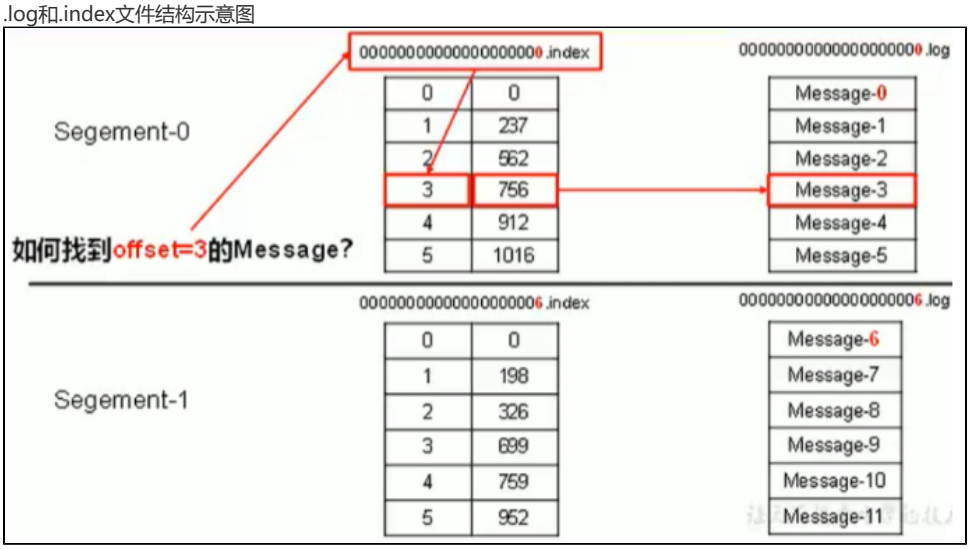
- `.log`和`.index`合起来叫做1个`segment`. 指1个partition中的1个片段的数据. 由于.log和.index可以有多个, 所以说1个partition分为多个segment片段;



- 由于生产者生产的消息会不断追加到.log文件末尾, 为防止.log文件过大导致数据定位效率低下, kfk采取和机制, 将每个partition分为多个segment. 每个segment对应两个文件, .log和.index. 这些文件位于同一个文件夹下, 文件命名规则为`topic+`. 例如topic01这个topic有3个partition, 则其对应的文件夹命名为topic01-0, topic01-1, topic01-2.

- .log和.index命名规则相同. 都是当前文件中存储的消息的最小偏移量. 例如第一个.log文件存了10000条消息就到达了1G, 则新生成的.log文件的命名为10001.log.

- .log和.index文件结构示意图



3指第三条消息, 就是生产者发送的消息的偏移量是3, 理解为生产者发送来的消息的序号;

756指第三条消息的物理偏移量, 理解为第三条消息在.log文件中第756行开始;

查找一条消息时, 拿着消息偏移量查询.log文件的名字, 通过定位这条消息在哪个.log文件中, 再从.log文件中通过(.log)就找到这条消息了.

.index文件中, 即()和(.log). 所以.index文件中每个"键值对(实际上不是键值对)"的大小都是相同的, 这样有利于快速定位索引. 索引的值中不仅存这条消息的物理偏移量, 也保存这条消息的大小(可以理解为这条消息的总行数). 例如值中存的是756,也存了这条消息一共1000行, 那么到log中查询756-1756行就能取出这条消息了.

3.3 kfk的生产者

3.3.1 topic分区的原因

1. . partition可以存在不同的机器(broker)中, 这样就能存储任意大小的数据了.
2. . 能以partition为单位进行读写. 即多个生产者可以同时将消息发往多个partition, 多个消费者能同时从多个partition读取数据.

3.3.2 生产者的分区策略

- 讨论一个topic中有多个partition, 那么生产者发送数据发送到哪个partition的问题.
- api中将生产者发送的消息封装到ProducerRecord中, 该对象的构造器如下:

```
ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value, Iterable<Header> headers)
ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)
ProducerRecord(String topic, Integer partition, K key, V value,  Iterable<Header> headers)
ProducerRecord(String topic, Integer partition, K key, V value)
ProducerRecord(String topic, K key, V value)
ProducerRecord(String topic, V value)
```

- `String topic`: 上述所有的构造器都有topic参数, 可见生产者发送消息, `topic`;
- `Integer partition`: 消息发送到哪个partition分区;
- `K key, V value`: kfk的消息是k/v形式的;
- `Iterable<Header> headers`: 发送消息时指定头信息, 非必须;
- `ProducerRecord(String topic, K key, V value)`中, 不需要指定分区号, kfk会通过`hash(key)%partition`来决定消息发往哪个partition;
- `ProducerRecord(String topic, V value)`中, 不指定k, 那么k=null, null不能hash()不能%, 所以kfk会在所有的partition分区中随机选择一个发送消息. 如果一个topic一共4个partition, 分区编号分别为0123, 随机指定往2号分区发, 那么下次发送时将会往3号分区发, 因此该api的分区策略是轮询, 即round-robin算法;

3.3.3 生产者发送数据的可靠性

- 讨论生产者发送消息后如何确认kfk接收到消息的问题; 涉及到ACK和ISR;

- 什么是ack确认机制
  - 为了保证producer发送的消息能可靠地到达指定的topic(能被topic接收到), topic的每个partition收到消息后都要向producer发送ack(acknowledgement 确认收到), 如果producer收到ack就会进行下一轮发送, 否则重发;

- 何时发送ack
  - leader和follower同步完成后leader再发送ack, 这样能保证leader挂掉后能在follower中选举出新的leader. 即保证在数据不丢失的情况下让leader发送ack.
  - follower可能有多个, 多少个follower同步完成后发送ack呢? 现有两个方案:
    - 方案1: 半数以上的follower同步完成后让leader发送ack;
    - 方案2: 所有的follower同步完成后让leader发送ack; -- kfk选择该方案.
  - 何时发送ack的方案比较

方案	优点	缺点
半数以上的follower同步完成后让leader发送ack;	延迟低;	选举新leader时, 容忍n台节点故障, 需要n+(n+1)个副本;
所有的follower同步完成后让leader发送ack;	选举新leader时, 容忍n台节点故障, 需要n+1个副本;	延迟高;

- kfk选择了方案2, 但是该方案仍然有问题. "同步所有的follower才会发送ack", 假如有一台follower同步的时候挂掉了, 那么永远不会发送ack了. 因此kfk引入了 `ISR` 的概念.

- 查看ISR

```
[root@king ~]# /opt/kafka/kafka/bin/kafka-topics.sh --describe --topic topic01 --zookeeper localhost:2181
Topic:topic01   PartitionCount:2        ReplicationFactor:2     Configs:
        Topic: topic01  Partition: 0    Leader: 3       Replicas: 3,1   Isr: 3,1
        Topic: topic01  Partition: 1    Leader: 1       Replicas: 1,2   Isr: 1,2
[root@king ~]#
```

- 什么是ISR (in-sync replica set 同步副本)
  - ISR的作用就是为了解决leader发送ack的问题.
  - 假如有10个副本, 1个leader和9个follower, 那么kfk必须等9个follower同步完leader的数据才会让leader向producer发送ack, 那么如果同步数据的时候有1个follower挂掉了, 那么就不可能发送ack了. 所以我们可以将9个follower中的部分follower放到ISR中, 假如将4个follower放到ISR中, 那么只需要等ISR中的这4个follower同步完数据, leader就会向producer发送ack, 如果leader挂了, 那么选新leader的时候从这4个follower中选即可. 除非这4个follower全挂了, 才会考虑选没加入ISR的其他follower作为leader.

- 小结: 为了保证生产者发送数据的数据可靠性, 就引入了ack, 那么什么时候发送ack呢? 为了不丢失数据, 有两种策略, 一是半数以上的follower同步数据完成才会发送ack, 二是全部的follower同步完数据完成才会发送ack. kfk选择了方案二, 方案二存在问题, 就是万一同步数据的时候有一台follower挂了, 那么永远不会发送ack了, 问了解决这个问题, 引入了ISR(同步副本). ISR的作用就是当leader挂掉后, 从ISR中选择一台follower作为新的leader.

- 当ISR中的follower完成数据同步之后, leader就会发送ack. 如果follower长时间没有从leader同步数据, 则该follower会被踢出ISR. 这个时间阈值由replica.lag.time.max.ms指定.

- follower满足什么条件才会被加入ISR
  - 条件1: `followerISR`. producer和follower发心跳, follower能够快速响应. 这样丢数据的可能性更小. 这里说的通信速度,通过上述的`replica.lag.time.max.ms`参数设定
  - 条件2: `followerISR`. producer向leader发送数据, 所有的follower异步同步数据, 有的follower同步了8条, 有 的follower同步了6条, 那么同步数据多个follower才能被选入ISR. 这样丢数据的可能性小. 这里说的同步的消息条数, 通过 `replica.lag.time.max.message`参数设定. 这个参数实际上是leader中数据和follower中数据的差值.

- kfk0.9开始, 条件2被去掉了.
  - 生产者是批量(batch)向leader发送数据的. ISR队列维护在kfk内存中, zk中也维护一份ISR. 如果`replica.lag.time.max.message`设置为10, 但是生产者一次batch向leader发送12条数据, 生产者不断向leader发送batch, 那么会有follower频繁加入ISR踢出ISR, 那么就需要频繁操作kfk内存和zk.

replica.lag.time.max.ms	10000	If a follower hasn't sent any fetch requests for this window of time, the leader will remove the follower from ISR (in-sync replicas) and treat it as dead.
replica.lag.max.messages	4000	If a replica falls more than this many messages behind the leader, the leader will remove the follower from ISR and treat it as dead.

- 综上, kfk0.9开始, leader向producer发送ack的步骤是
  - producer发送消息到leader, leader向ISR中的follower同步数据, ISR中follower同步完数据后向leader做出响应, leader接收到响应后向producer发送ack.

- 这种发送ack的步骤还有有问题

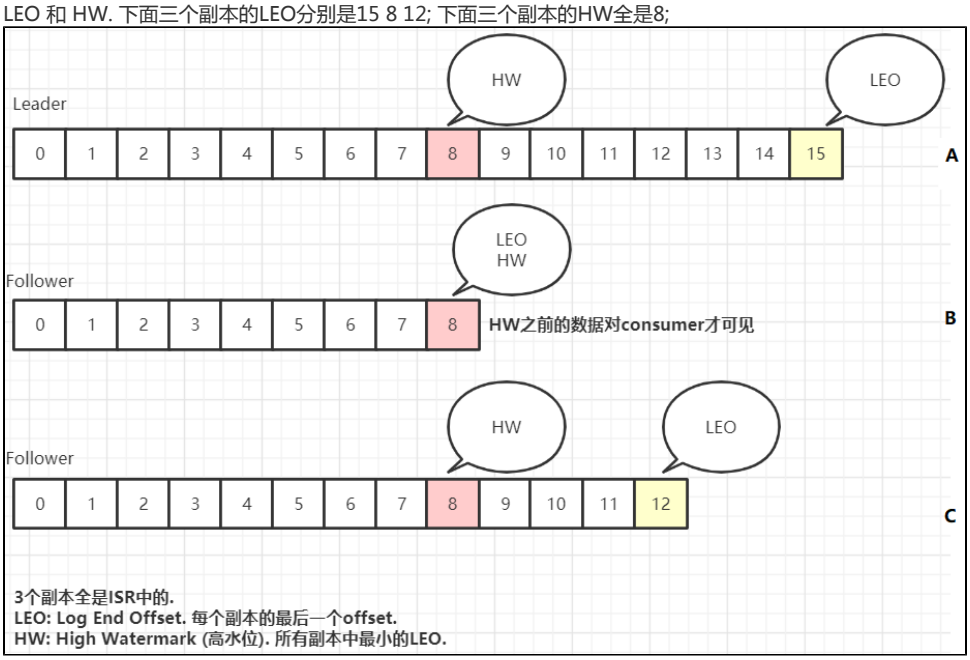
    - leader必须收到ISR中所有follower(同步完数据后)的响应之后才会发送ack, 这明显比较慢.

- 为了解决上述问题, ack的发送有很多机制. 对于不太重要的数据, 对于数据可靠性要求不高(消息丢失率), 能容忍少量数据丢失, 所以没必要等到ISR中所有的follower都同步完数据并都向leader响应数据同步成功才让leader发送ack. 所以kfk中有3种数据可靠性级别, 能让用户对可靠性和延迟要求进行权衡.
    - 可靠性级别通过producer的`acks`参数配置. 取值`0`, `1`, `-1`.
    - `acks:0 =>` producer不等待broker的ack. 即broker一接收到生产者消息, 还没写入磁盘就返回成功. 延迟最低, 最容易丢数据, leader挂掉的话刚才发送的数据就丢了, 不会重发消息.
    - `acks:1 =>` leader将数据落盘成功后就发送ack. 如果leader延迟过大或者leader挂掉, 即如果producer没收到ack, 那么重发消息. 有可能丢数据, 因为leader收到数据后可能挂掉了, 没来得及向ISR中的follower同步完数据.
    - `acks:-1`或 `acks:all =>` leader和ISR中的follower全部写入同步数据完成之后由leader向producer发送ack.
        - 这种情况下会造成 数据重复 .这种情况也会丢失数据, 就是ISR中没有一个副本的时候, 那么只有一个leader, 这个leader挂了就会丢失数据. 我们重点讨论这种情况中数据重复的问题.
        - 为什么会有重复数据? 当producer向leader中写完数据后, ISR中的follower也同步完数据, leader向producer发送ack之前挂掉了. 那么此时将选ISR中的一个follower作为leader, 同时producer由于长时间没收到ack会重发数据.
    - acks=0和acks=1会丢失数据, acks=-1时我们更多的讨论重复数据的问题.

3.3.4 数据一致性问题

- 场景
    - 假设ISR中有3个副本, 一个leader两个follower. 如果leader写了15条数据, 两个follower异步从leader同步数据, 同步数据过程中leader挂了, 假设此时follower_a同步了8条数据, follower_b同步了12条数据. 假设follower_b被选为leader. follower_b被选为leader之后挂掉的leader活了. leader如果多挂几次, 那么3个副本中存储的数据就会不同, 消费时就会出现问题.
    - `.` , `.`

- LEO 和 HW. 下面三个副本的LEO分别是15 8 12; 下面三个副本的HW全是8;



- 假设消费者消费Leader中第13条数据, 此时Leader挂掉了, B被选为Leader, 此时B才同步到第8条数据, 消费者接下来应该消费第14条数据, B根本没有这条数据, 报错. 引入HW后, 消费者最多只能看到第8条数据, 此时即便是Leader挂了, 无论哪个Follower被选为Leader, 消费者都不会报错. 所以说`HW`.

- 如果B被选为Leader. 此时生产者又发了一条msg来, 那么msg将是B的第9条数据, msg是A的第16条数据. 于是kfk使用了: 如果B被选为Leader, 那么A中和C中大于8的消息将会被删除, 如果C被选为Leader, 那么B中会补上9 10 11 12, A中会删除8后的数据然后补上9 10 11 12. 即截取高水位. 这样就保证了副本数据存储在log文件中的.

- HW不能保证数据丢失, 数据不丢失不重复是通过`acks`来保证的. 如果Leader挂了, C被选为Leader, 由于producer没有收到ack, 那么会重发消息, 这就导致了数据重复.

- 小结: `HW: ; ACK;`

3.3.5 Exactly Once 语义

- 解决重复消费问题. kfk0.11才引入了消息幂等性. kfk0.11之前通过Redis搞日志表进行消息去重. kfk0.11之后引入的精准一次性, 在broker中解决了幂等性问题.

- acks=-1的时候, 消息不会丢失(实际上可能会丢失), 但是可能会有重复数据, 消息可能会重发, 即at leastonce(至少一次).
  acks=0的时候, 可以保证消息只发送一次, 消息会丢失, 消息不会重发, 即at mostonce(至多一次). 消息即不重复也不丢失, 这就叫Exactly Once, 即精准一次性.
- `at least once + = Exactly Once`. 即 +=.
- kfk开启幂等性: 只需要将producer的参数中的 enable.idompotence 设置为 true 即可. 并且设置为true后, acks自动被设置为-1.
- kfk的幂等性实现实际上就是将消费者要做的事情放到了broker中做. (消费者通过Redis搞日志表进行消息去重).
- 开启幂等性的producer在初始化的时候会被分配一个PID, 发往同一个Partition的消息会附带Sequence Number. 而Broker端会对`<PID, Partition, Sequence>` 做缓存, 当具有相同主键的消息提交时, Broker只会持久化一条.
- 但是pid重启就会发生变化, 而且不同的partition有不同的pid, 因此幂等性无法保证跨分区(partition)跨会话的精确一次性.
- 跨会话指的是producer重启, 那么pid会重新生成. 生产者不挂掉不重启就能保证精准一次性. pid是生产者producer的id. sequence number即序列化号, 就是每条消息的id.

3.4 kfk的消费者

- 跨会话指的是producer重启, 那么pid会重新生成. 生产者不挂掉不重启就能保证精准一次性. pid是生产者producer的id. sequence number即序列化号, 就是每条消息的id.

3.4.1 消费者的消费方式

- consumer采用pull拉取的方式从broker中拉取数据.
  push, broker推送数据的方式很难适应不同的消费者, 因为不同消费者的消费速率不同.

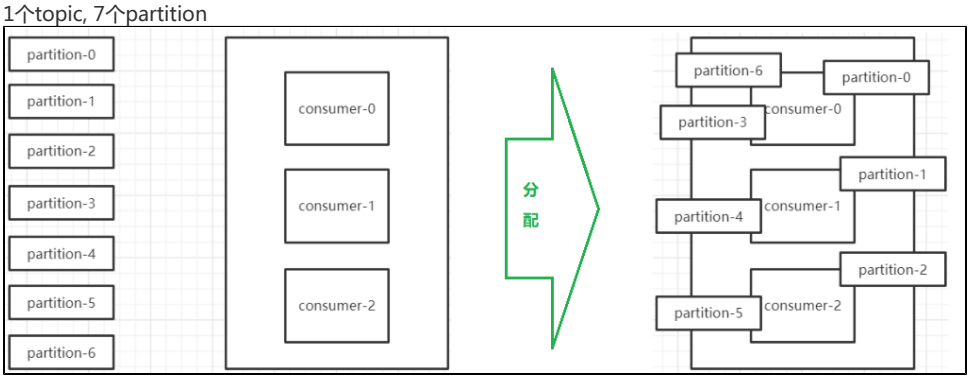
- pull模式的缺点是如果kfk中没有数据, 消费者可能陷入循环, 一直返回空数据.
  - 针对这一点, kfk的消费者在消费数据时会传入一个时长参数`timeout`, 如果当前没有数据可供消费, consumer会等待一段时间之后再返回, 这段时长即为`timeout`.

3.4.2 分区分配策略

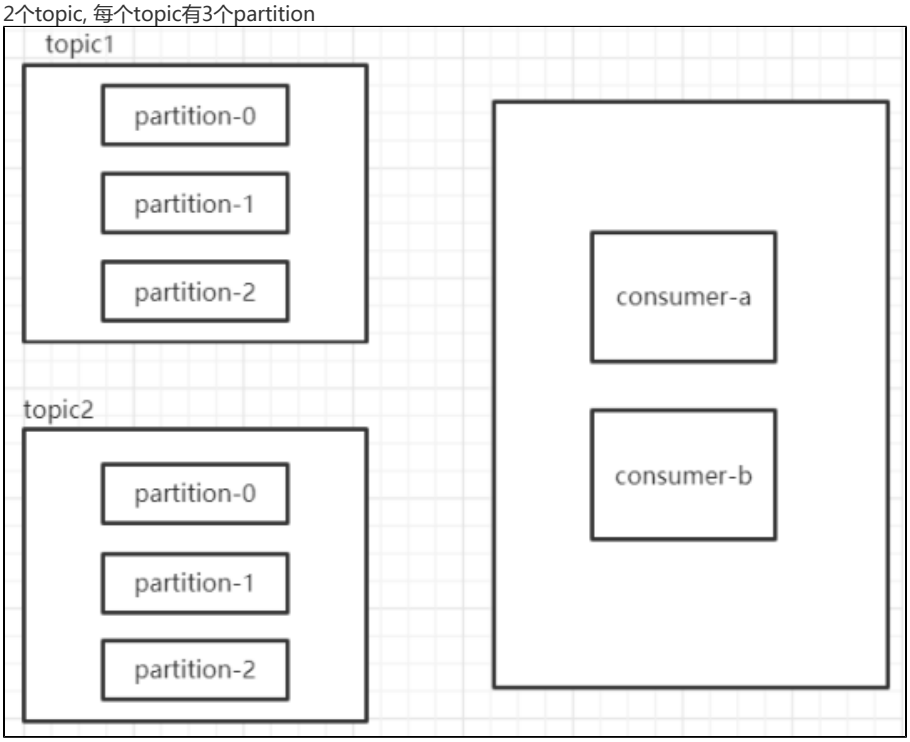
- 一个consumer group中有多个consumer, 一个topic有多个partition, 所以必然会涉及到partition的分配问题, 即确定哪个partition由哪个consumer来消费.
- 同一个消费者组中的不同消费者, 能同时消费同一个topic, 不能同时消费同一个partition.
- kfk有两种消费者分区分配策略: RoundRobin 和 Range(默认).
- RoundRobin是按照组来划分的, Range是按照主题来划分的.

3.4.2.1 RoundRobin轮询

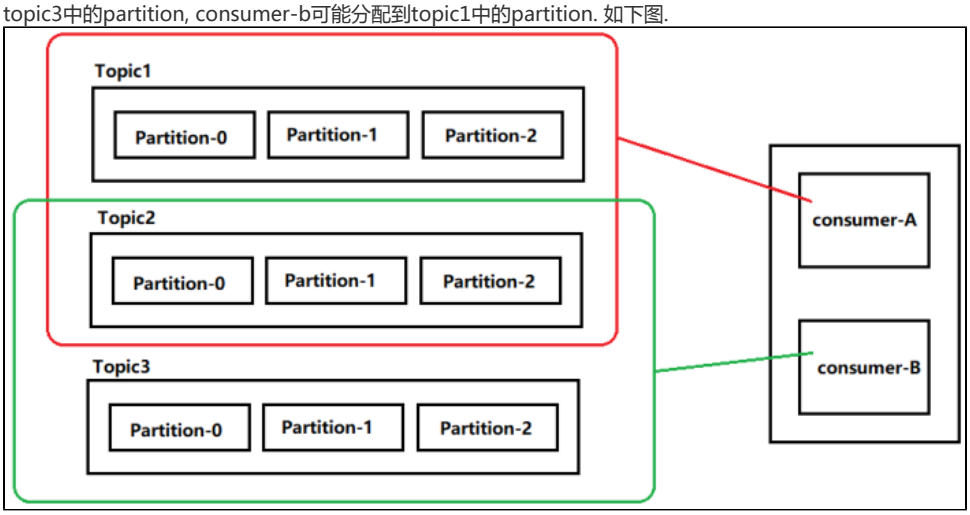
- 1个topic, 7个partition



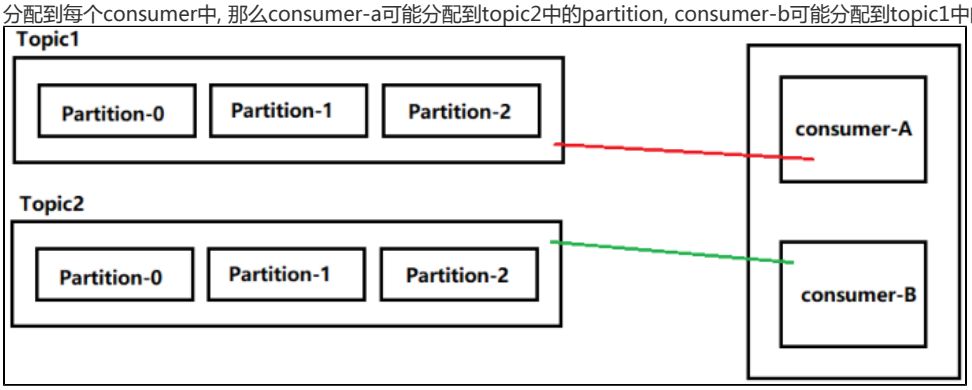
- 2个topic, 每个topic有3个partition



  - 假设消费者组同时订阅了两个topic(如上图). 那么在kfk轮询的策略中, 会把topic1和topic2中所有的partition看做一个整体, 根据每个partition的哈希值, 将这6个partition排序, 然后再轮询分配给consumer group中的两个consumer.
  - 在API中, 每个partition都会被封装成一个TopicAndPartition对象, 使用的是这个对象的哈希值.
  - 这样做的优点是consumer group中每个consumer 分配的partition的数量的差值最多差1个, 因为是轮询嘛. 即保证了每个consumer分配到的partition数量尽可能相同.
  - 这样做的是假如同一个consumer group中的consumer-a订阅了topic1和topic2, consumer-b订阅了topic2和topic3, 那么轮询策略会把topic1 topic2 topic3中所有的partition看做一个整体, 轮询分配到consumer-a和consumer-b中, 这样的话consumer-a可能分配到topic3中的partition, consumer-b可能分配到topic1中的partition. 如下图.



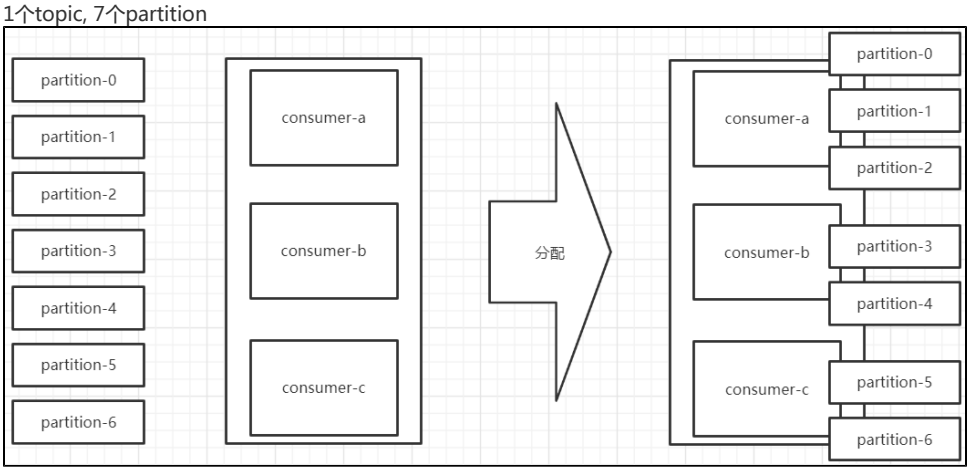
- 如果consumer-a消费topic1, consumer-b消费topic2, 那么按照轮询的分配策略, 会将topic1 topic2中的所有partition看做一个整体轮询分配到每个consumer中, 那么consumer-a可能分配到topic2中的partition, consumer-b可能分配到topic1中的partition. 如下图.



- 所以使用轮询的分区分配策略的前提是: 必须保证consumer group中consumer订阅的topic是相同的(可以是多个).
- 正是由于有上述的问题, 因此kfk默认的分区分配策略是Range.

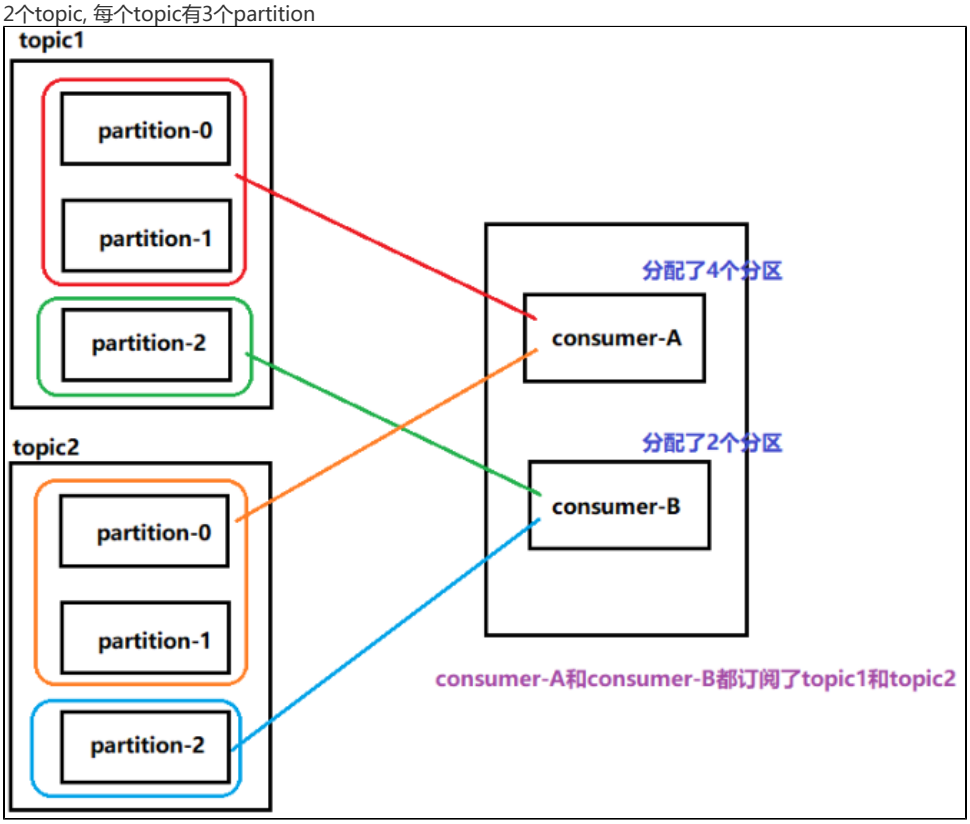
3.4.2.2 Range范围(默认)

- 这种策略按照topic分, 而不是按照partition分.
- 1个topic, 7个partition



- 按照 7/3 将partition分配到每个consumer中.
- 这种策略的缺点是有的consumer分配的partition太多, 有的consumer分配的partition太少. 即consumer group中每个consumer分配的partition数量不均匀.

- 2个topic, 每个topic有3个partition



  - 随着订阅的topic越多, 不同consumer中分配的partition数差距将越大.

- 结论
  - RoundRobin策略优点是能让consumer group中每个consumer尽量分配到相同数量的partition, 缺点是当consumer group中consumer订阅的topic不同时会出问题, 因为所有的topic会被看做一个整体.
  - Range策略的优点是RoundRobin策略的缺点, Range策略的缺点是RoundRobin策略的优点.
  - 两种策略按照实际业务场景选择, 默认Range.

3.4.2.3 什么时候会用到分区分配策略

- 分区策略在API中就是一个方法. 这个方法何时被调用? 即何时会触发策略的执行? 即策略的触发时机是什么时候?
  - 当consumer group中消费者个数变化时就会重新触发分区策略. 即使消费者数量增加到比分区数量还多时, 仍然会触发分区策略.

3.4.3 offset的维护

- 由于consumer在消费过程中可能会出现断电宕机等故障, consumer恢复后, 需要从故障前的位置继续消费, 所以需要consumer实时记录自己消费到了哪个offset, 以便故障恢复后继续消费.
- kfk0.9前偏移量保存在zk, kfk0.9开始偏移量保存在kfk.
- 从kfk0.9开始, consumer默认将offset保存在kfk一个内置的topic中, 该topic为_consumer_offsets.

- offset由3个部分唯一确定: consumer group + topic + partition. offset是按照consumer group保存的而不是按照consumer保存的, 这样的好处就是如果一个consumer挂了, offset不至于丢失. 如果consumer-a保存了10条msg, 这是consumer group中新加consumer-b(此时触发分区策略), 这就能保证consumer-b能从第11条msg接着消费.

- 查看zk中维护的offset
  - 创建一个topic名为topic100, 2个partition, 每个partition都有2个副本

```
/opt/kafka/kafka/bin/kafka-topics.sh --create --topic topic100 --zookeeper localhost:2181 --
partitions 2 --replication-factor 2
```

  - 启动一个生产者

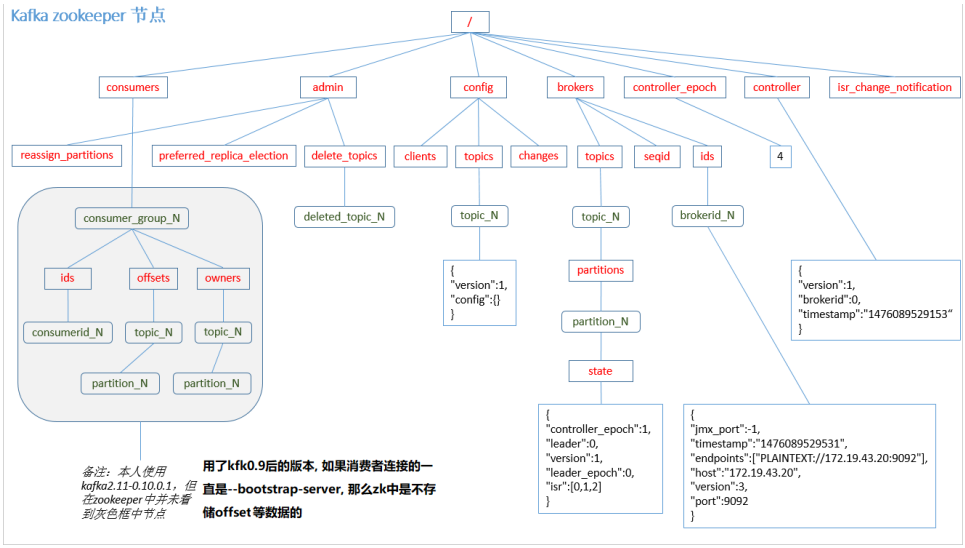
```
/opt/kafka/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic topic100
```

  - 启动一个消费者. 使用zk存储offset.

```
/opt/kafka/kafka/bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic topic100
```

- 生产者输入 "hello"
- 进入zk

```
# zk
/opt/zookeeper/zookeeper/bin/zkCli.sh

# .

# , zookeeper, kfk. controllerkfk, broker, brokercontroller, brokercontroller. controllerbroker,
controllerzk.

# brokersids,topics,seqid. idskfkbroker-id. ids, kfk. kfk, zkkfk; /brokers/topicskfktopic.

# /consumerszk. "console-consumer-68063", , , , "console-consumer-". API, , .

# offset, offsettopic, topic, offset.

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

[zk: localhost:2181(CONNECTED) 3] ls /consumers
[console-consumer-68063, console-consumer-39547, console-consumer-47083]

[zk: localhost:2181(CONNECTED) 4] ls /consumers/console-consumer-68063
[ids, owners, offsets]

[zk: localhost:2181(CONNECTED) 8] ls /consumers/console-consumer-39547/offsets
[topic100]

[zk: localhost:2181(CONNECTED) 9] ls /consumers/console-consumer-39547/offsets/topic100
[0, 1]

[zk: localhost:2181(CONNECTED) 10] get /consumers/console-consumer-39547/offsets/topic100/0
0 #
cZxid = 0x649
ctime = Sat Dec 05 16:39:13 CST 2020
mZxid = 0x649
mtime = Sat Dec 05 16:39:13 CST 2020
pZxid = 0x649
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 1
numChildren = 0
```



- 查看kfk本地维护的offset
  - 由于kfk中存储offset的主题是kfk内置的主题_consumer_offsets, 因此我们需要改一些配置才能消费这个topic中的数据.

```
# consumer.properties. topics.
exclude.internal.topis=false
```

  - 读取offset
    - kfk0.11之前的版本
      - 略
    - kfk0.11开始的版本
      - 略

3.4.4 消费者组案例

- 测试同一个消费者组中的消费者, 同一时刻只能有一个消费者消费.
- 使用控制台的consumer时, 每次启动一个consumer都会分配到一个新的consumer group中, 组名随机分配. 我们可以通过修改consumer.properties文件, 将group.id修改为任意, 那么启动任意个控制台的consumer就都会在同一个consumer group中了. group.id就是组名.

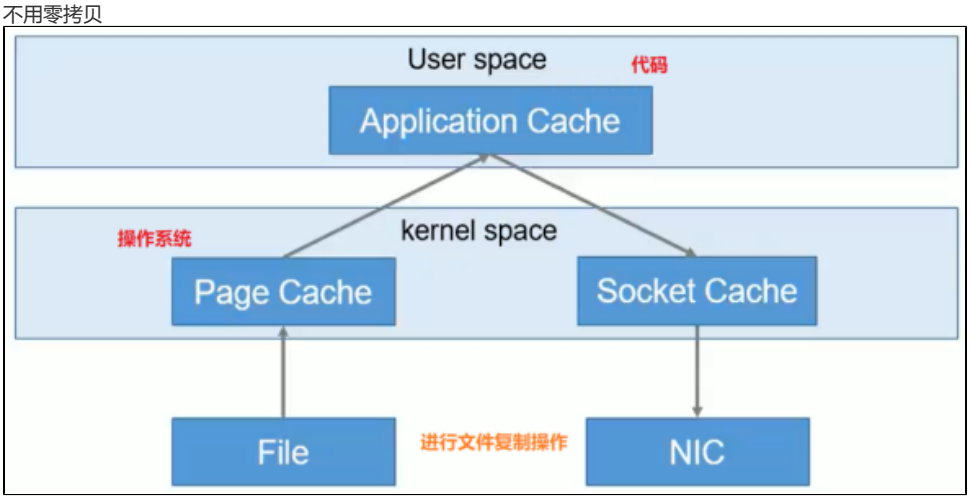
3.5 kfk高效读写数据的原因

- 研究集群版kfk中单个节点的读写效率为什么比其他mq效率高; 研究单台kfk的读写效率为什么比单台其他mq的效率高.

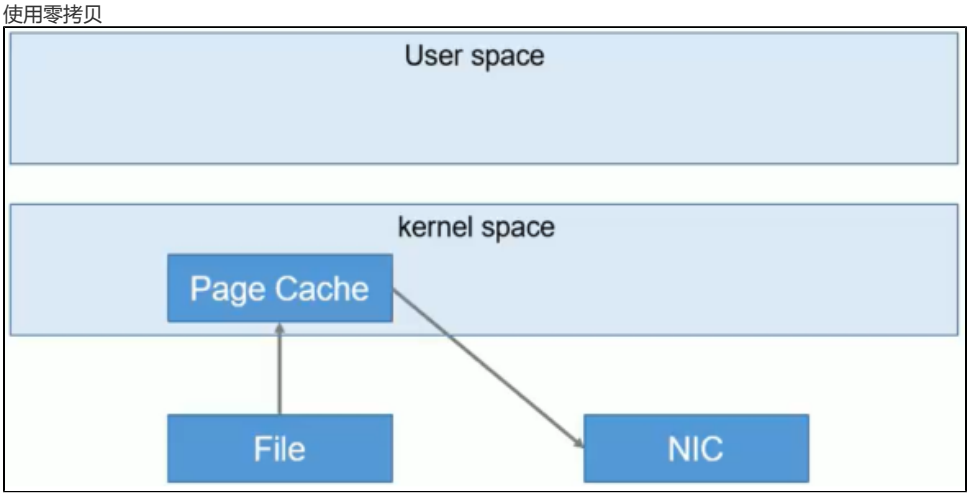
- 集群版原因
  - kfk是分布式的, 能并发读写.
  - 顺序写磁盘. 顺序写磁盘省去了大量磁头寻址的时间.
  - 零拷贝技术.

- 单节点原因
  - 顺序写磁盘. 顺序写磁盘省去了大量磁头寻址的时间.
  - 零拷贝技术.

- 不用零拷贝

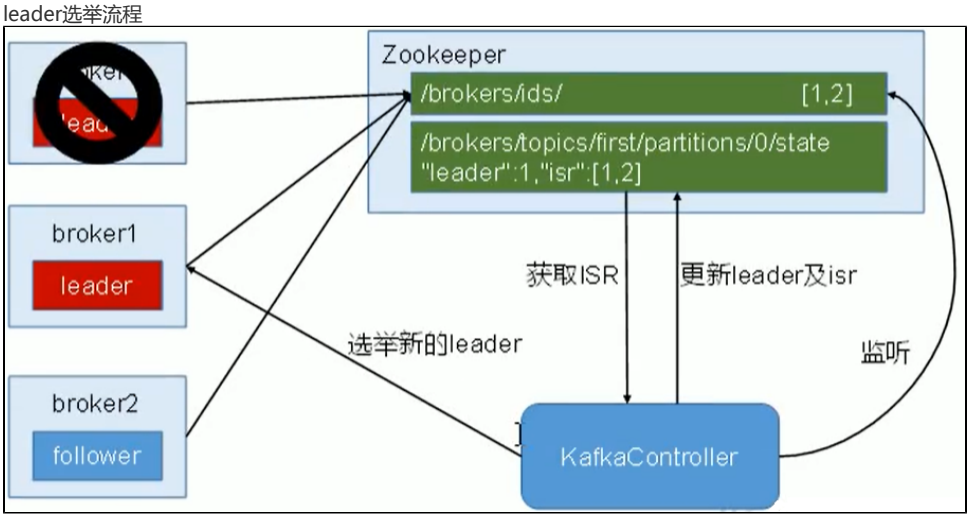


- 使用零拷贝

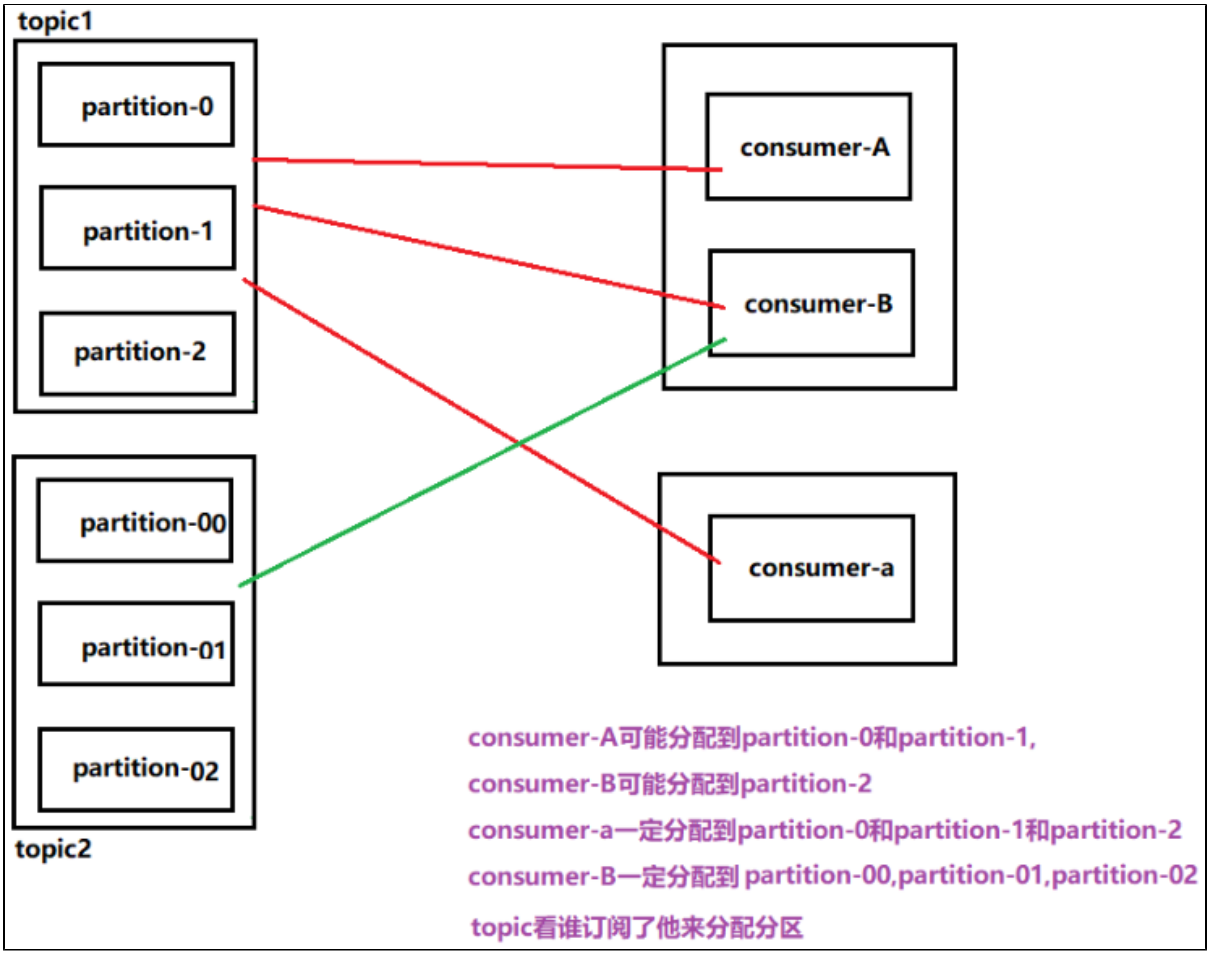


3.6 zk在kfk中的作用

- kfk集群中有一个broker会被选举为controller, 负责broker的上下线, 所有topic的和leader选举等工作. broker被选为controller的策略是抢占资源, 谁先抢到谁就是controller, 一般最先启动的broker将成为controller.
- controller的管理工作都是依赖于zk的.
- leader选举流程



3.7 Range策略再分析



- 上述使用的策略是Range
- 使用轮询的话, 看消费者组, 看整个组订阅了哪些topic, 然后将这些topic中的所有partition作为一个整体进行轮询. 所以上述的情况不能使用轮询策略, 否则consumer-A可能会分配到topic2的partition.

3.8 kfk事务

3.8.1 producer事务

- 保证跨分区跨会话级别的精准一次性.

3.8.2 consumer事务

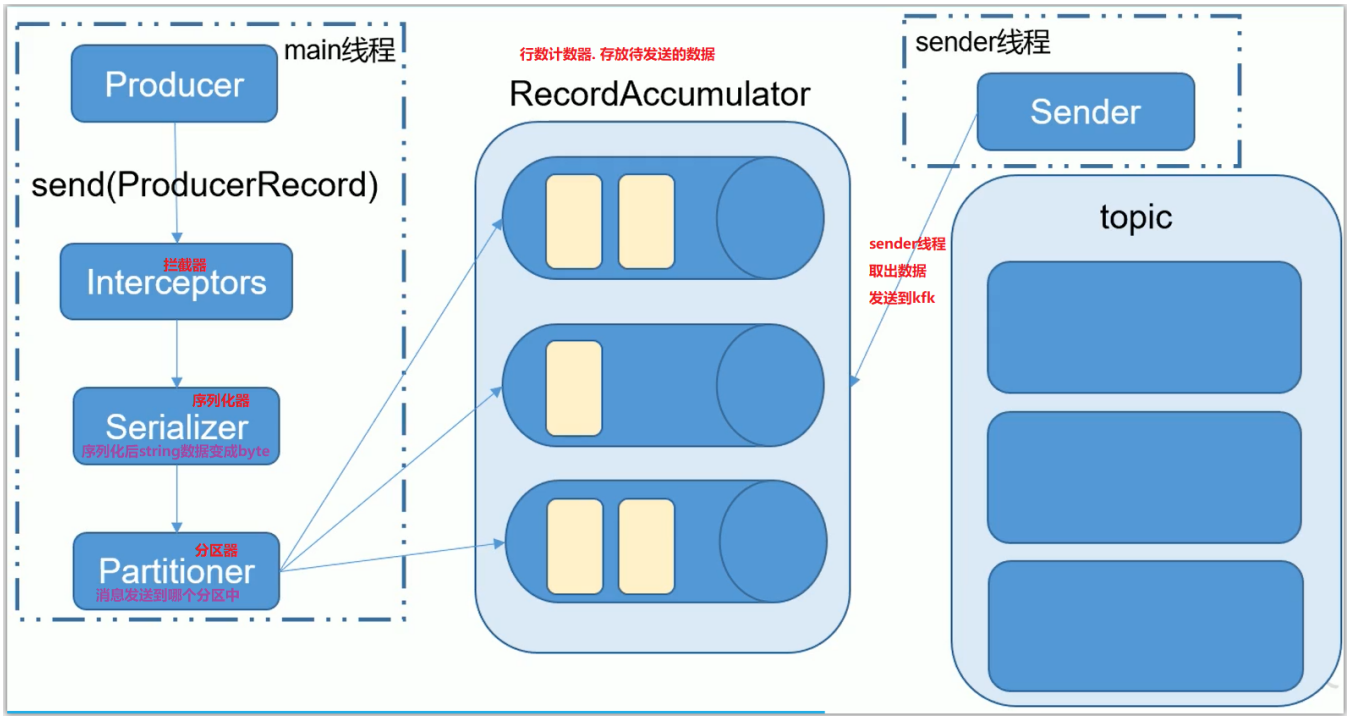
- 保证消费者消费的数据是只消费一次的. (这一块很少聊, 关键还是生产者那边的事务)

4 KafKa API

4.1 producer API

4.1.1 消息发送流程

- kfk的consumer发送消息是的, 一个main线程, 调用send()方法时启动一个sender线程发送 消息. 所以并不是发送一批消息等待ack后发送下一批消息. 有另外的线程接收ack, 如果ack没接收到 就重发.
- 消息发送过程中涉及到两个线程和一个(/), 这个共享变量中放的就 是待发送的数据.



- 注意上述顺序: send() -> 拦截器 -> 序列化器 -> 分区器. 我们将会自定义拦截器和分区器.

4.1.2 普通生产者

- 发送数据没有回调: cn.king.kfk01.producer.AProducer
- 发送数据有回调: cn.king.kfk01.producer.BProducer

4.1.3 API指定生产者的分区分配策略

- 发送数据指定分区: cn.king.kfk01.producer.CProducer

4.1.4 自定义分区器

- 自定义分区器: cn.king.kfk01.partitioner.MyPartitioner
- 使用自定义分区器的生产者: cn.king.kfk01.producer.DProducer

4.1.5 同步发送消息的API

- 如果在调用send()方法时将main线程阻塞, 那么这就相当于是同步发送消息了.
- 原理是用到了Future类的get()方法.
- 同步发送了解即可. 在此不写demo.

- 注意
  - 如果想保证全局消息有序, 光靠发送到同一个partition(分区)是不够的. 因为如果往partition-a发送 1 2 3 消息, 由于是异步发送, 那么无论partition-a是否收到消息, 都会立即发送 4 5 6 到partition-a中, 如果partition-a没收到 1 2 3, 那么生产者将消息 1 2 3重试发送到partition-a. 此时 1 2 3 消息在 4 5 6 消息之后.
  - 所以说如果想用kfk实现消息全局有序, 必须 `partition + .`

4.1.6 异步发送消息的API

- 除了`4.1.5` 中介绍的, 其余全是异步发送. kfk默认就是异步发消息.

4.2 consumer API

- kfk consumer有低级api和高级api.

- 高级api只需要引入

```
<!--api. kfk. api, -->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

- 低级api需要引入

```
<!--kfk. api, -->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
<!--kfk-->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.12</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

- 高级API
  - 优点
    - 书写简单.
    - offset, zk.
    - , , .
    - 消费者断线会自动根据上一次记录在zk中的offset去接着获取数据. (默认设置1分钟更新一下zookeeper中存的offset)
    - 可以使用group来区分对同一个topic 的不同程序访问分离开来. (不同的group记录不同的offset，这样不同程序读取同一个topic才不会因为offset互相影响)
  - 缺点
    - offset. (对于某些特殊需求来说)
    - 不能细化控制如分区、副本、zk等.
- 低级API
  - 优点
    - offset, .
    - 自行控制连接分区, 对分区自定义进行负载均衡.
    - 对zk的依赖性降低. (如: offset不一定非要靠zk存储, 自行存储offset即可, 比如存在文件或者内存中)
  - 缺点
    - . 需要自行控制offset, 连接哪个分区, 找到分区leader 等.
- 本文代码全部使用高级API.

4.2.1 普通消费者

- cn.king.kfk01.consumer.AConsumer

4.2.2 重置offset

- 即API实现控制台的 --first-beginning 的效果.
- 在 org.apache.kafka.clients.consumer.ConsumerConfig 类中有一个成员变量 public static final String AUTO_OFFSET_RESET_CONFIG = "auto.offset.reset"; 这个配置项并不是写上就会生效的. 必须在两个特定场景参会生效:
  - 当前消费者组第一次消费. (消费者换了个组, 即换了个组名, 这种情况下会生效)
  - 没换组名. 消费者组消费过数据, 但是消费的这个offset在集群中已经不存在了. (最常见的情况就是默认过了7天后数据已经被删除了)
- 这个属性有两个值可以选择: earliest() 和 latest(.). 即从头开始消费还是从最大的开始消费.

```
properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

- 最早的偏移量不一定是0, 因为如果偏移量是0-1000并且7天之后数据删掉了, 那么最早的偏移量将会是1001.
- 面试题: 如何重新消费某一个主题的数据? 答案: 换一个组, 同时将 auto.offset.reset 参数值设置为earliest.

4.2.3 自动提交offset

```
// offset
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
//
enable.auto.commit: offset.
auto.commit.interval.ms: offset.
```

4.2.4 手动提交offset

```
// offset. cn.king.kfk01.consumer.AConsumer.
properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
// . 1. , , .
properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG,"1000");
```

- 按理说如果consumer消费了90-100条数据, 如果不配置自动提交, 那么consumer不会将最新的offset(也就是100)写入kfk, 下次再消费也会从90开始消费, 消费的还是第90-100条数据.
- 但是实际上, consumer内存中维护了一份offset, 即便是没有将offset提交到kfk, 那么consumer再次消费数据的时候, 会先去消费者内存中获取最新消费的offset. 即下次消费会从101开始消费. 但是offset始终没有写入到kfk, 那么kill掉消费者重启, 还是会从第90条消息开始消费.

- 可见保存在kfk或者zk的offset只会在consumer启动的时候访问一次, consumer启动后访问的是consumer内存中维护的offset.

- 自动提交offset的缺点
  - 自动提交是基于时间提交的(例如配置1s), 开发人员难以把握offset提交的时机. 因此可以使用手动提交offset; 例如配置了自动提交时间是1s, producer生产100条数据, consumer在没消费完100条数据之前就挂了, 此时offset提交了, 那么consumer重启之后offset=100开始消费, 这就产生了consumer丢数据. 所以自动提交不好, 推荐使用手动提交offset.

- consumer手动提交offset有两种方式: 同步提交(commitSync) 异步提交(commitAsync)
  - 两者相同点: 都会将本次poll的一批数据的最高偏移量提交;
  - 两者不同点: commitSync阻塞当前线程, 一直到提交成功, 并且会自动失败重试(由不可控因素导致, 也会出现提交失败); commitAsync没有失败重试机制, 因此有可能提交失败.
  - 同步提交有失败重试机制, 因此更加可靠, 但是由于她会阻塞线程, 直到提交成功, 因此吞吐量会受到很大影响. 因此更多的情况下会选用异步提交offset的方式.
  - 同步提交: cn.king.kfk01.consumer.BConsumer
  - 异步提交: cn.king.kfk01.consumer.CConsumer

- 无论是同步提交还是异步提交, 都可能出现数据漏消费或重复消费的问题.
  - 自动提交缺点
    - 如果设置的延迟时间过短会容易丢失数据. 上述已说明原因.
    - 如果设置的延迟时间过长会产生重复消费问题. 例如发送100条msg, 设置自动提交延迟为10s, 假设消费者5秒处理完了100条msg并且第6秒的时候消费者挂了, 那么等到10s的时候消费者才会提交offset, 那么第6秒时消费者还没提交offset, 那么重启消费者后, 因为offset还未提交, kfk中的offset还是消费这100条msg之前的offset, 那么从这100条mag之前开始消费, 这就是重复消费了.
  - 手动提交时, 使用异步提交也可能出现重复消费. cn.king.kfk01.consumer.CConsumer中的代码, 如果在调用commitAsync()方法时挂了, 那么offset没提交, 就会出现重复消费.
  - 由于无论是手动提交还是自动提交都可能出现重复消费, 所以kfk提供了一个`offset`功能.

4.2.5 自定义存储offset

- 提交offset提交到了kfk或者zk. 自定义存储offset的意思是我们可以将offset维护在mysql或者本地文件等地方.
- 最好是将offset存储到mysql, 这样我们可以将 和 `offset` 搞一个事务.

- offset的维护是很麻烦的, 因为涉及到 `consumerRebalance()`.

  - 当有新的consumer加入consumer group 或者 已有的consumer退出consumer group 或者 订阅的topic的partition发生变化, 就会触发partition的重新分配 , 重新分配的过程叫做Rebalance.
  - consumer发生Rebalance之后, 每个consumer消费的partition就会发生变化. 因此consumer首先要获取到自己被重新分配到的partition, 并且定位到每个分区最近提交的offset位置继续消费.

        - 例如三个分区ABC的offset分别被3个consumer维护着, 每个分区的offset都写到mysql中, 如果维护A分区offset的消费者挂了, 那么就需要将A分区维护的offset交给B分区或者C分区维护. 这就叫再平衡.

- 要实现自定义存储offset, 需要使用的类是 `ConsumerRebalanceListener`.
- 如果要将offset存储到mysql, 那么这张表的列必须包含: consumer_group topic partition offset.
- 希望consumer完全不丢失数据的时候需要使用自定义存储offset, 就是将消息的消费和偏移量的存储做一个事务.

- 示例代码: cn.king.kfk01.consumer.DConsumer

4.3 自定义Interceptor

4.3.1 拦截器原理

- 作用: msg发送前做一些定制化的需求, 比如修改msg等.
- 一个producer可以指定多个拦截器形成一条拦截器链作用同一条消息.
- 需要实现的接口: `org.apache.kafka.clients.producer.ProducerInterceptor`.
- 拦截器是 `kfk0.10` 才有的.
- 条消息都会经过一次拦截器链.
- 拦截器中方法如下:

```
//
public void configure(Map<String, ?> configs);
// . .
public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record);
// ack. .
// metadatanull,
// exceptionnull,
public void onAcknowledgement(RecordMetadata metadata, Exception exception);
//
public void close();
```

4.3.2 拦截器案例

- cn.king.kfk01.interceptor.AInterceptor: 在发送的消息前面加上时间戳.
- cn.king.kfk01.interceptor.BInterceptor: 消息发送成功后会更新发送成功的消息数 或 发送失败的消息数.
- cn.king.kfk01.producer.EProducer: 带拦截器的生产者.

注意

本文所有代码均已上传本人github. 为遵守公司规章制度, 不贴github链接. 需要代码请私信我.