Kafka

一、概述

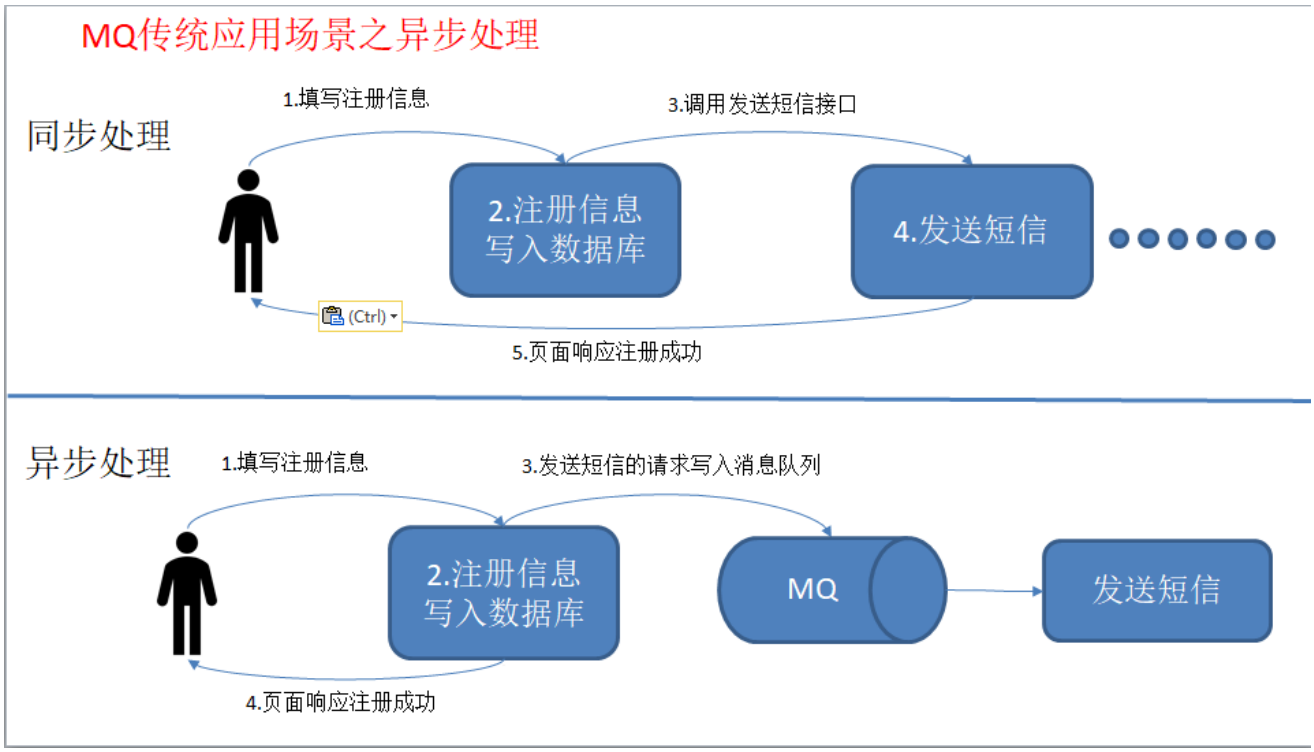
1.定义

> Kafka是一个分布式的基于发布/订阅模式的**消息队列**，主要应用于大数据实时处理领域。
>
> 优势：kafka可以做到，使用非常普通的硬件，也可以支持每秒数百万的消息读写。

2.消息队列 MQ

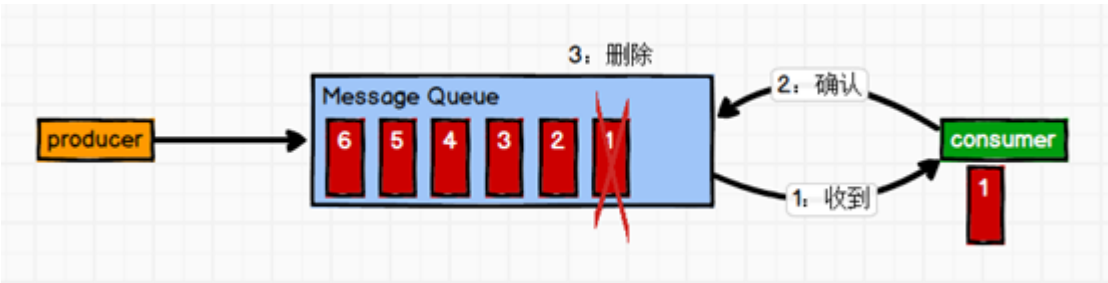
> MQ (message Queue) 在传统开发中的应用场景 : 发短信、秒杀等等
>
> rocketMQ/activeMQ/rabbitMQ kafka(软件功能并不多，但是吞吐量比较高)

> 特点：削峰 解耦 异步

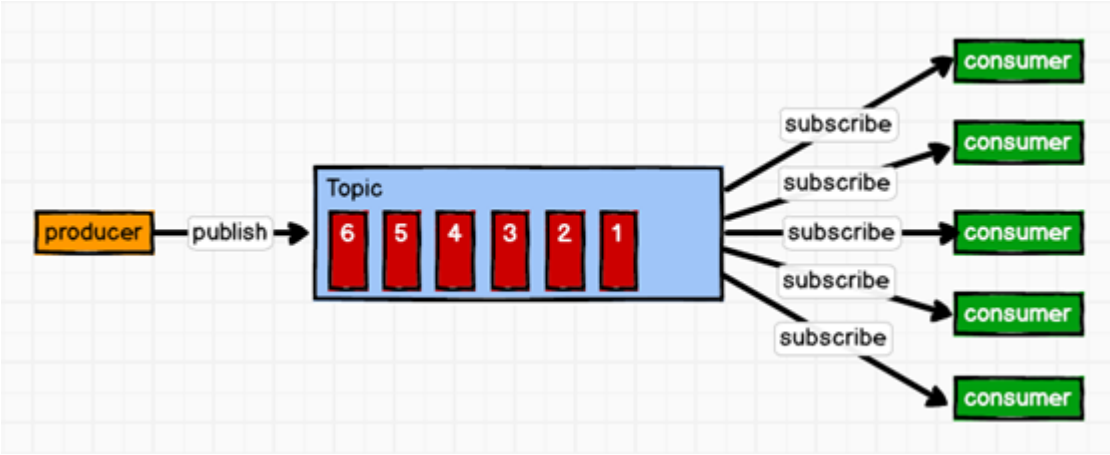


3.消息队列的两种模式

（1）点对点模式（一对一，消费者主动拉取数据，消息收到后消息清除） 消息生产者生产消息发送到Queue中，然后消息消费者从Queue中取出并且消费消息。 消息被消费以后，queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。



（2）发布/订阅模式（一对多，消费者消费数据之后不会清除消息） 消息生产者（发布）将消息发布到topic中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到topic的消息会被所有订阅者消费。

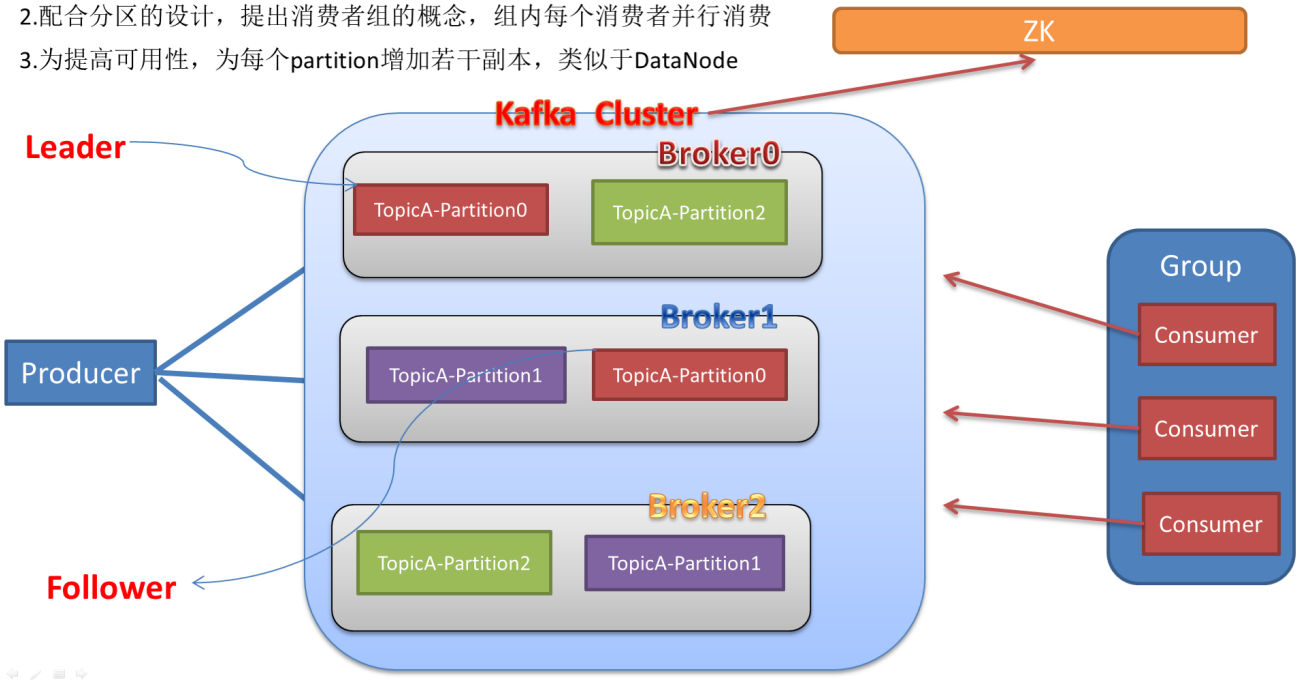


4.Kafka基础架构

1.为了提高吞吐量，一个topic分为多个partition

2.配合分区的设计，提出消费者组的概念，组内每个消费者并行消费

3.为提高可用性，为每个partition增加若干副本，类似于DataNode



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

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

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

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

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

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

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

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

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

二、安装

集群规划

hadoop11	hadoop12	hadoop13
zk	zk	zk
kafka	kafka	kafka

http://kafka.apache.org/downloads.html

安装步骤

0.准备工作

```
安装JDK,搭建zookeeper集群环境
```

1.解压kafka安装包

```
[root@hadoop11 modules]# tar -zxvf kafka_2.11-0.11.0.0.tgz -C /opt/installs
```

2.修改解压后的文件名 （配置环境变量）

```
[root@hadoop11 installs]# mv kafka_2.11-0.11.0.0  kafka0.11
```

3.在/opt/installs/kafka0.11目录下创建logs文件夹

```
[root@hadoop11 installs]# cd kafka0.11
[root@hadoop11 kafka0.11]# mkdir logs
```

4.修改配置文件

```
[root@hadoop11 kafka0.11]# cd config
[root@hadoop11 config]# vi server.properties
# 每个kafka节点,broker.id必须不一样
broker.id=11
# 允许topic可以删除
delete.topic.enable=true
# kafka运行日志存放的路径
log.dirs=/opt/installs/kafka0.11/logs
# 配置连接Zookeeper集群地址
zookeeper.connect=hadoop11:2181,hadoop12:2181,hadoop13:2181
```

5.分发安装包

```
[root@hadoop11 installs]# scp -r /opt/kafka0.11 root@hadoop12:/opt/installs
[root@hadoop11 installs]# scp -r /opt/kafka0.11 root@hadoop13:/opt/installs
```

6.分别在hadoop12和hadoop13上修改配置文件/opt/installs/kafka/config/server.properties中的broker.id

```
broker.id=12
broker.id=13
注：broker.id不得重复
```

7.启动集群

```
在三台节点分别执行命令启动kafka
[root@hadoop11 kafka0.11]# bin/kafka-server-start.sh -daemon config/server.properties
[root@hadoop12 kafka0.11]# bin/kafka-server-start.sh -daemon config/server.properties
[root@hadoop13 kafka0.11]# bin/kafka-server-start.sh -daemon config/server.properties
```

8.验证集群是否启动成功

```
[root@hadoop11 kafka]# jps
1571 Kafka
1622 Jps
1215 QuorumPeerMain
```

8.关闭集群

```
[root@hadoop11  kafka0.11]$ bin/kafka-server-stop.sh stop
[root@hadoop12  kafka0.11]$ bin/kafka-server-stop.sh stop
[root@hadoop13  kafka0.11]$ bin/kafka-server-stop.sh stop
```

三、常用命令

1. 创建topic

> 注意：一般在系统设计的时候，先把topic规划好，业务含义相同的数据，放在一个topic中。

```
[root@hadoop11 kafka0.11]# bin/kafka-topics.sh  --zookeeper hadoop11:2181  --create   --
topic topica --partitions 3 --replication-factor 2

## ---------------参数说明------------------ ##
--zookeeper 连接的zookeeper
--create 表示要创建一个topic
--topic 指定topic的名字(不同业务的数据，放在不同的topic中)
--partitions 指定分区数
--replication-factor 指定副本数
```

2. 查看所有topic

```
[root@hadoop11 kafka0.11]# bin/kafka-topics.sh  --zookeeper hadoop11:2181  --list
```

3. 查看某个topic详情

```
[root@hadoop11 kafka0.11]# bin/kafka-topics.sh  --zookeeper hadoop11:2181  --describe --
topic topica
```

4. 删除topic

```
[root@hadoop11  kafka0.11]# bin/kafka-topics.sh  --zookeeper hadoop11:2181  --delete   --
topic topica
```

5. 接收消息

```
[root@hadoop11  kafka0.11]# bin/kafka-console-consumer.sh --bootstrap-server hadoop11:9092
--topic topica
```

6. 发送消息

```
[root@hadoop11  kafka0.11]# bin/kafka-console-producer.sh --broker-list   hadoop11:9092 --
topic topica
```

四、Kafka Java API

1.Producer API

① 添加依赖

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

② 相关API

KafkaProducer：需要创建一个生产者对象，用来发送数据

ProducerConfig：获取所需的一系列配置参数

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

③ 异步发送 不带回调函数的Producer

```
public class CustomProducer {


   public static void main(String[] args) throws Exception {
        //1. 初始化参数信息
        Map<String, Object> configs = new HashMap<>();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"hadoop11:9092");
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        //2. 创建生产者
        KafkaProducer<String,String> producer = new KafkaProducer<>(configs);
        //3. 发送数据

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

```
            producer.send(new ProducerRecord<>("topica","hello"+i));
        }

        producer.close();
    }

}
```

④ 异步发送 带回调函数的Producer

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

注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试。

```
public class CustomProducer_CallBack {


    public static void main(String[] args) throws Exception {
        Map<String, Object> configs = new HashMap<>();
        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"hadoop11:9092");
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);

        KafkaProducer<String,String> producer = new KafkaProducer<>(configs);
        for (int i=0;i<10;i++){
            producer.send(new ProducerRecord<>("topica", "hello" + i), new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception exception) {
                    if(exception == null){
                         System.out.println("发送成功: "+metadata.partition());//数据所在分区
                         System.out.println("发送成功: "+metadata.topic());//数据所对应的topic
                         System.out.println("发送成功: "+metadata.offset());//数据的offset
                    }
                }
            });
        }

        producer.close();
    }

}
```

2.Consumer API

① 相关API

KafkaConsumer：需要创建一个消费者对象，用来消费数据

ConsumerConfig：获取所需的一系列配置参数

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

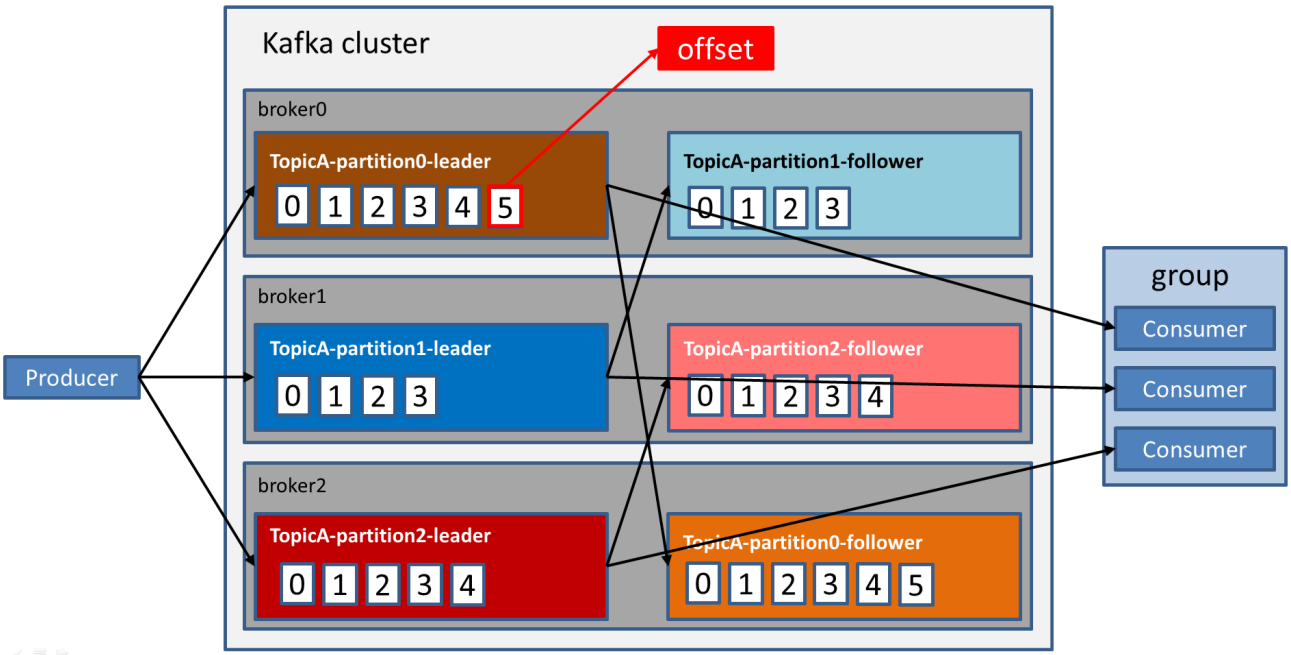
② Consumer接收数据

```
public class CustomConsumer {

    public static void main(String[] args) {
        //1. 初始化配置信息
        Map<String,Object> map = new HashMap<>();
        map.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"hadoop11:9092");
        map.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        map.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,StringDeserializer.class);
        map.put(ConsumerConfig.GROUP_ID_CONFIG,"g1");
        //2. 创建Consumer
        KafkaConsumer<String,String> kafkaConsumer = new KafkaConsumer(map);
        //订阅 topic-user的数据
        kafkaConsumer.subscribe(Arrays.asList("topica"));

        while (true){
            //3. 消费数据
            ConsumerRecords<String, String> consumerRecords = kafkaConsumer.poll(100);
            Iterator<ConsumerRecord<String, String>> iterator = consumerRecords.iterator();
            for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
                System.out.println(consumerRecord);
            }
        }

    }

}
```

五、Kafka架构深入

1. Kafka工作流程及文件存储机制

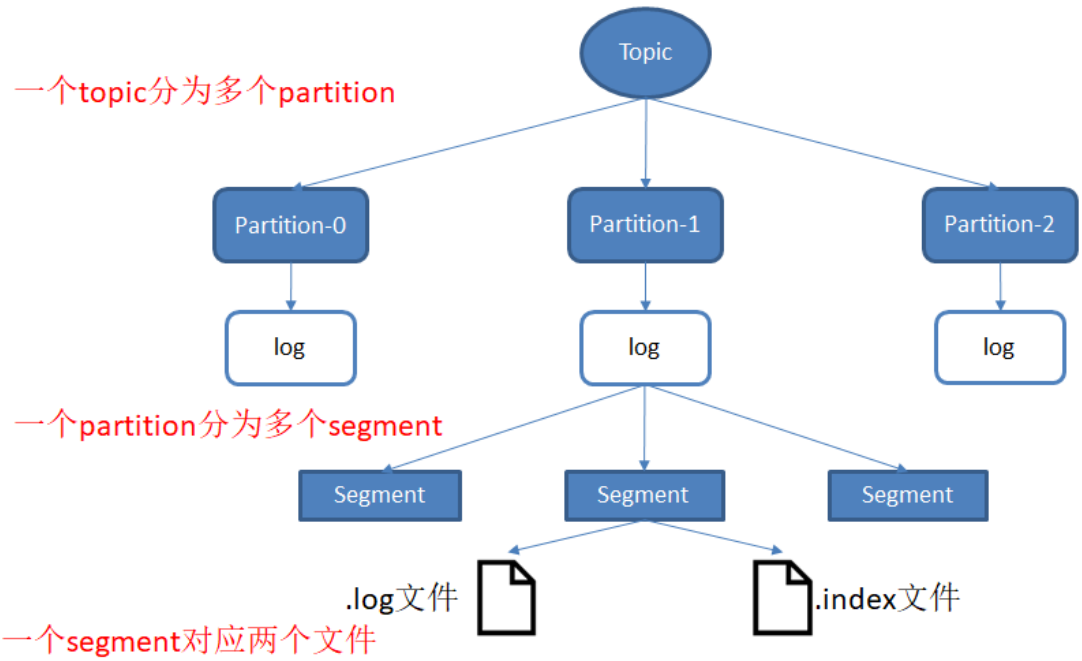
Kafka 工作流程



Kafka中消息是以**topic**进行分类的，生产者生产消息，消费者消费消息，都是面向topic的。

topic是逻辑上的概念，而partition是物理上的概念，每个partition对应于一个log文件，该log文件中存储的就是producer生产的数据。Producer生产的数据会被不断追加到该log文件末端，且每条数据都有自己的offset。消费者组中的每个消费者，都会实时记录自己消费到了哪个offset，以便出错恢复时，从上次的位置继续消费。

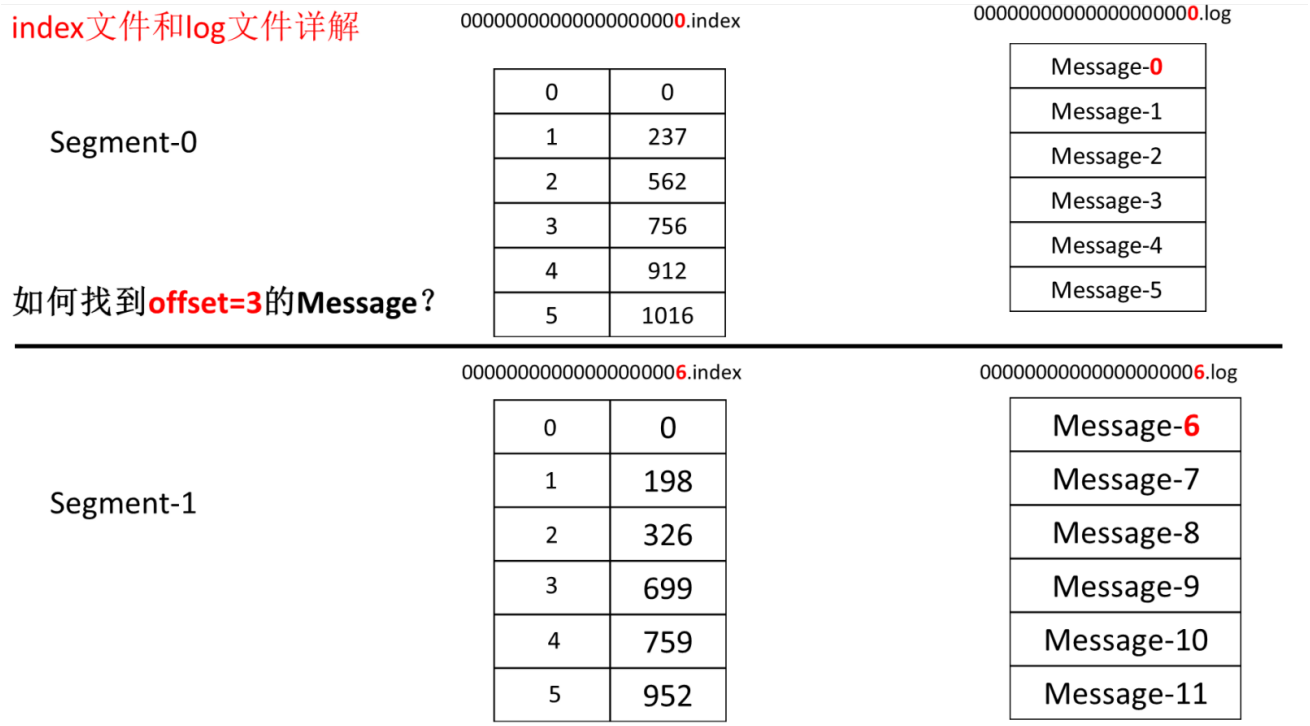
Kafka文件存储机制



由于生产者生产的消息会不断追加到log文件末尾，为防止log文件过大导致数据定位效率低下，Kafka采取了**分片**和**索引**机制，将每个partition分为多个segment。每个segment对应两个文件——“.index”文件和“.log”文件。这些文件位于一个文件夹下，该文件夹的命名规则为：topic名称+分区序号。例如，first这个topic有三个分区，则其对应的文件夹为first-0,first-1,first-2。

```
00000000000000000000.index
00000000000000000000.log
00000000000028532124.index
00000000000028532124.log
```

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



“.index”文件存储大量的索引信息，“.log”文件存储大量的数据，索引文件中的元数据指向对应数据文件中message的物理偏移地址。

2. Kafka生产者 之 分区策略

1）分区的原因

（1）方便在集群中扩展，每个Partition可以通过调整以适应它所在的机器，而一个topic又可以有多个Partition组成，因此整个集群就可以适应任意大小的数据了；

（2）可以提高并发，因为可以以Partition为单位读写了。

2）分区的原则

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

```
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value)
ProducerRecord(@NotNull String topic, String key, String value)
ProducerRecord(@NotNull String topic, String value)
```

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

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

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

3) 自定义生产者分区策略

```
① 自定义类 implements Partitioner
② props.put("partitioner.class", "自定义类");
```

3. Kafka生产者 之 数据可靠性保证

为保证producer发送的数据，能可靠的发送到指定的topic，topic的每个partition收到producer发送的数据后，都需要向producer发送ack（acknowledgement确认收到），如果producer收到ack，就会进行下一轮的发送，否则重新发送数据。

ack应答机制

对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失，所以没必要等ISR中的follower全部接收成功。

所以Kafka为用户提供了三种可靠性级别，用户根据对可靠性和延迟的要求进行权衡，选择以下的配置。

acks参数配置：

acks：

0：producer不等待broker的ack，这一操作提供了一个最低的延迟，broker一接收到还没有写入磁盘就已经返回，当broker故障时有可能**丢失数据**；

1：producer等待broker的ack，partition的leader落盘成功后返回ack，如果在follower同步成功之前leader故障，那么将会**丢失数据**；

-1（all）：producer等待broker的ack，partition的leader和follower全部落盘成功后才返回ack。但是如果在follower同步完成后，broker发送ack之前，leader发生故障，那么会造成**数据重复。**

4. Kafka生产者 之 幂等性

> 幂等性
>
> 当重复提交一个请求时，对服务器的影响只会产生一次，这就可以称之为幂等性
>
> 查询一定是幂等性，增删改可能是幂等性，也可能不是

对于某些比较重要的消息（业务消息），我们需要保证exactly once语义，即保证每条消息被发送且仅被发送一次。

在0.11版本之后，Kafka引入了幂等性机制（idempotent），配合acks = -1时的at least once语义，实现了producer到broker的exactly once语义。

idempotent + at least once = exactly once

使用时，只需将enable.idempotence属性设置为true，kafka自动将acks属性设为-1。

5. Kafka消费者 之 分区分配策略

> 一个`Topic`中的数据，由一个`Consumer group`进行消费
>
> 一个Topic中的一个partition，由对应Consumer group中的1个consumer进行消费。
>
> 注意：1个Partition分区中的数据，只能被1个Consumer分区消费。(1个partition分区不能同时被一个`Consumer group`中的多个Consumer消费)

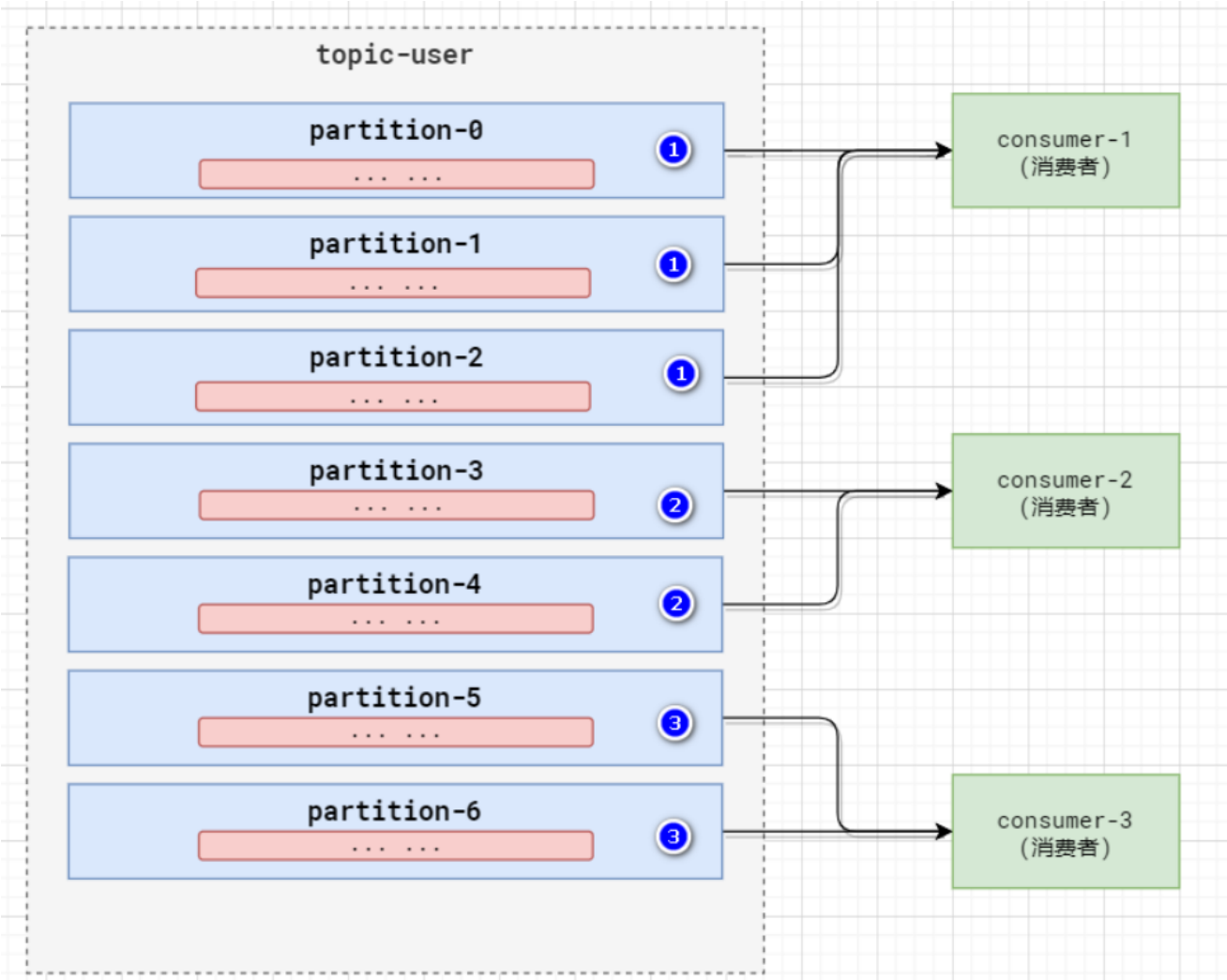
> 问题：一旦partition个数和consumer个数不一致，如何分配?
>
> 说明：一般开发中，生产者的代码的数据写入由一个团队开发，消费者消费代码，和数据读取可能是由另一个业务团队读取。有时候无法做到统一。

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。

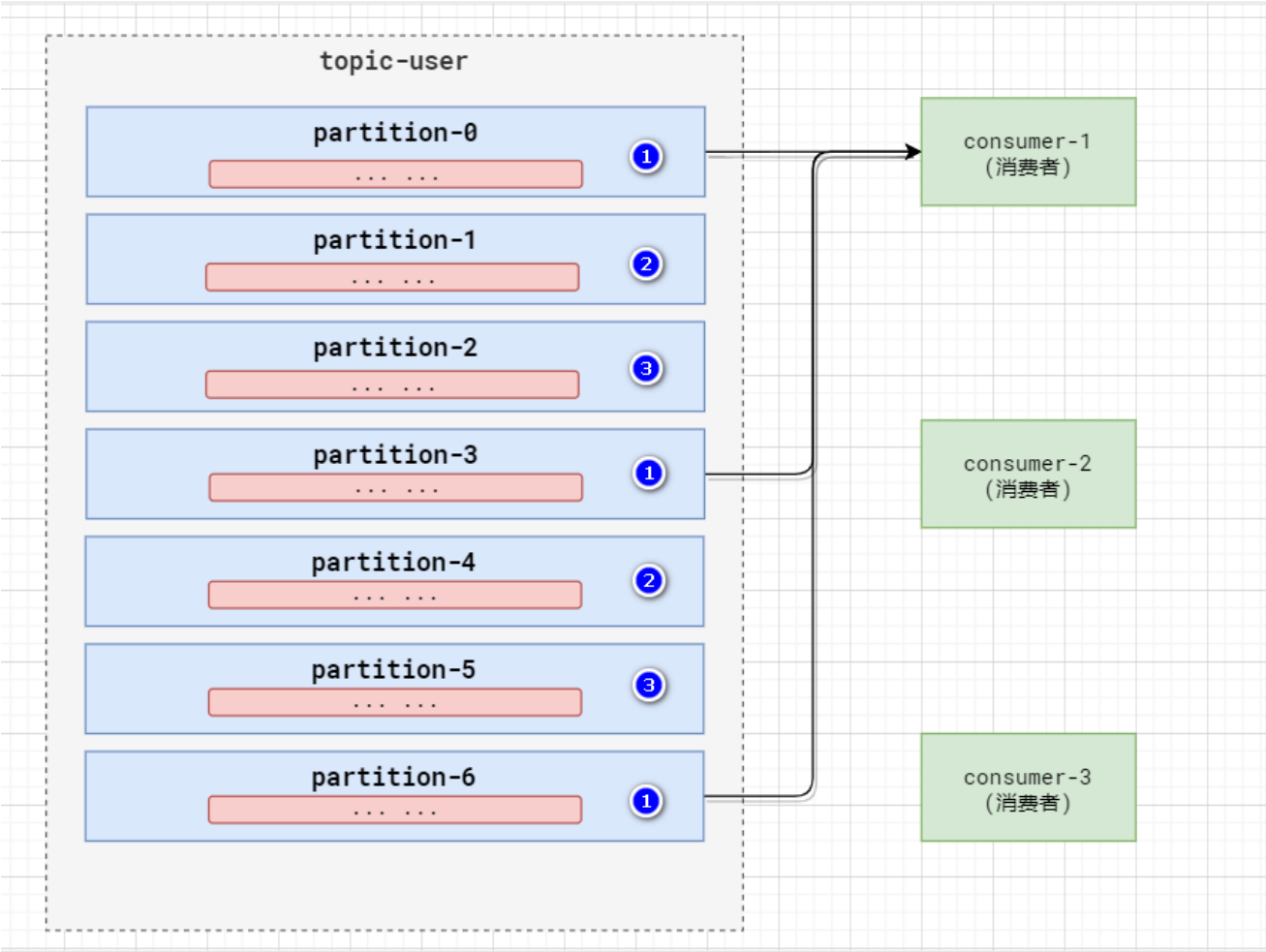
Kafka有两种分配策略，一是roundrobin，一是range。

1）range (区间)

> 默认



2）roundrobin (轮训)



6. Kafka消费者 之 offset的维护

> 由于consumer在消费过程中可能会出现断电宕机等故障，consumer恢复后，需要从故障前的位置的继续消费，所以consumer需要实时记录自己消费到了哪个offset，以便故障恢复后继续消费。
>
> `Offset：kafka会保存每个topic数据消费的记录offset，以便记录consumer消费到哪个数据了`

> Kafka 0.9版本之前，consumer默认将offset保存在Zookeeper中，从0.9版本开始，consumer默认将offset保存在Kafka一个内置的topic中，该topic为**__consumer_offsets**。

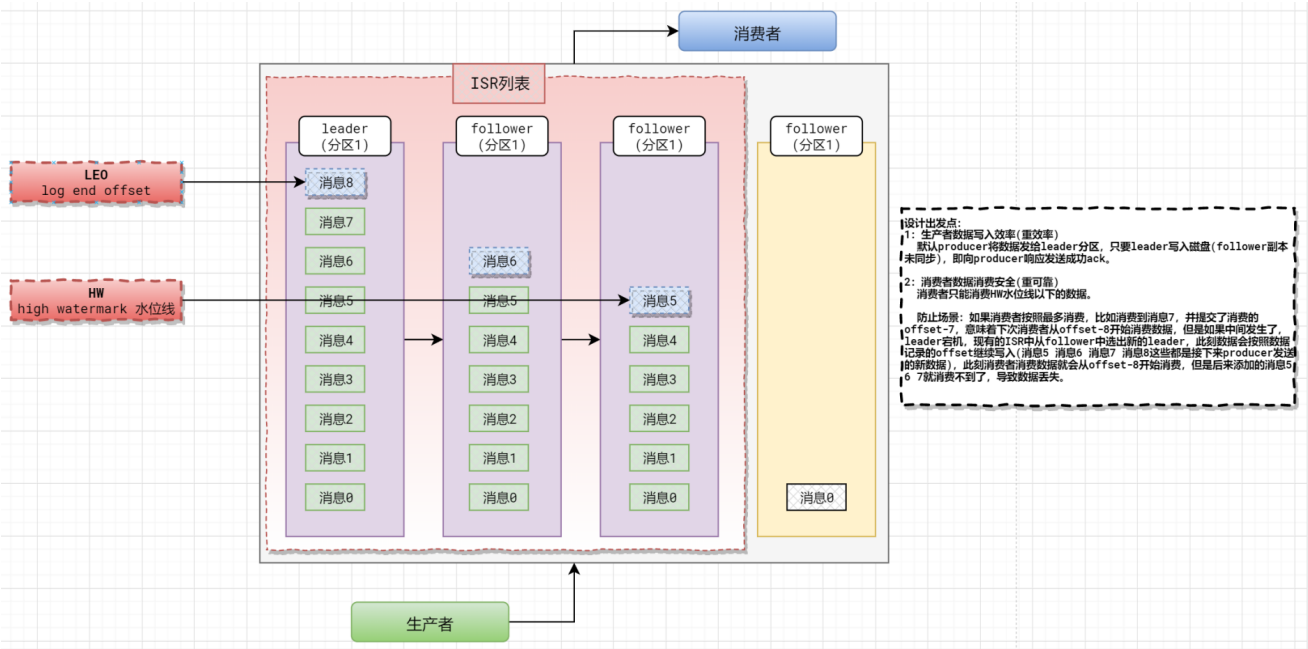
 # HW水位线和offset的关系

HW概念：

全称 high watermarker 水位线。

ISR（partition分区的主从副本）列表中，每个写入对应分区中leader副本的数据，follower会拉取数据期望与leader数据进行同步数据和offset，此刻因为网络延迟，会导致不同的follower拉取的速度不一样，在高并发场景下follower通常会滞后于leader，那么ISR内部offset最低的那个值就是HW。

作用：消费者只能消费HW这个offset以下的数据。



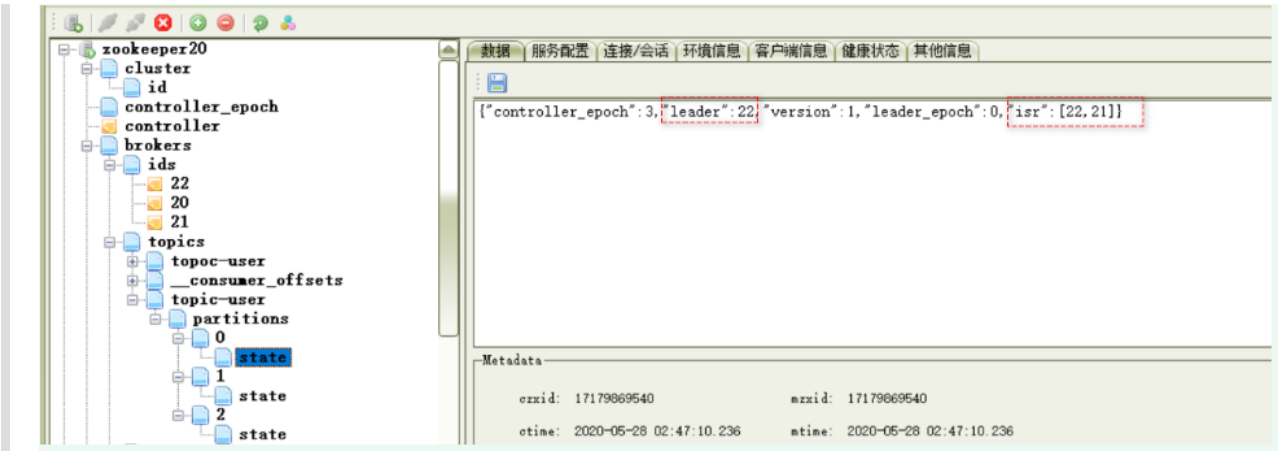
7. Zookeeper在Kafka中的作用

> kafka对zookeeper是强依赖，最新版的kafka2.8版本，已经不需要依赖zk了

Kafka集群中有一个broker会被选举为Controller(启动broker向zk注册，先到先得)，负责管理集群broker的上下线，所有topic的分区副本分配和leader选举等工作。

Controller的管理工作都是依赖于Zookeeper的。

> 1. 注册所有的broker(kafka节点)
>
> `临时节点`
>
> 2. 从broker中选出一个承担KafkaController的职责
>
> `1：防止单点故障，也是临时节点`
>
> `2：负责在需要的时候对ISR列表中选出leader`
>
> 3. 保存Topic的元数据信息(描述信息)
>
> `zookeeper保存topic的分区和leader信息，并协助KafkaController在需要的时候(启动或者broker故障)选出新的分区的leader`
>
> `kafkaController会监听 zookeeper中的borkers下的ids节点的子节点变化`



六、Flume整合Kafka

①. 在flume的job目录下新建taildir-memory-kafka.conf文件

```
a1.sources = r1
a1.channels = c1
a1.sinks = k1

a1.sources.r1.type = TAILDIR
a1.sources.r1.filegroups = f1
a1.sources.r1.filegroups.f1 = /opt/data/ceshi.log

a1.channels.c1.type = memory

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = topica
a1.sinks.k1.kafka.bootstrap.servers = hadoop11:9092,hadoop12:9092,hadoop13:9092

a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

②. 启动kafka的consumer消费者

```
[root@hadoop11 kafka]# bin/kafka-console-consumer.sh --bootstrap-server hadoop11:9092 --topic
topica
```

③.启动flume

```
[root@hadoop11 apache-flume-1.9.0-bin]# bin/flume-ng agent --conf conf --name a1 --conf-file
job/exec-memory-kafka.conf -Dflume.root.logger=INFO,console
```

④.向a.log文件中追加数据

```
[root@hadoop11 data]# echo hello >> ceshi.log
```

⑤.查看kafka的consumer消费者的消费情况

```
[root@hadoop11 kafka]# bin/kafka-console-consumer.sh --bootstrap-server kafka1:9092 --topic
topica
hello
```

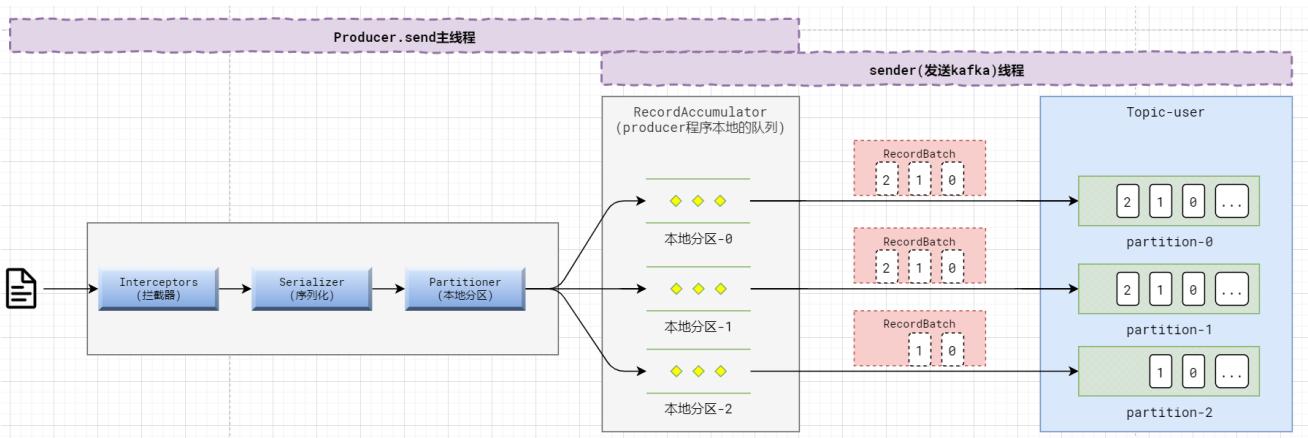
七、kafka执行流程

1. producer发送数据流程

```
# 流程说明
1. Producer.send的主线程。
    ① 数据先经过拦截器。
    ② 然后进行网络传输前的序列化
    ③ 计算消息所属的分区
    ④ 按照将消息存入对应的本地分区。(本地缓存，为了批量发送)
2. sender线程
    ① 当某个分区内的消息数量达到一定值：`batch.size`之后，才会发送数据。(默认值：16384 (16kb))
       config.put(ProducerConfig.BATCH_SIZE_CONFIG,16384);
    ② 如果某个分区内消息数量未达到batch.size，sender等待linger.ms之后也会批量发送。(默认值：0ms)
       config.put(ProducerConfig.LINGER_MS_CONFIG,200);
```

```
# 流程细节解释
1. 主线程一条条向本地分区中存入数据。
2. Sender线程批量将本地分区的数据，发送到kafka的topic中的对应分区。--提高效率
3. 拦截器：可以对producer发送的数据，做一些通用功能的处理。
4. 序列化：为了保证数据在网络中传输和kafka的broker之间同步，需要数据执行序列化。
5. Partitioner：数据在写入到本地的内存队列(缓冲区)之前，会先计算分区再存放数据。
```

```
# 图示
```



2. Consumer消费数据流程

offset相关

```
# offset
```

topic-user-0分区中的文件

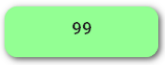
000000000010.index(索引文件)

offset(偏移量第几条数据)	msg数据的物理位置
0	1677
1	1808
2	1940
3	2072
4	2203
5	2335
6	2466
7	2598
8	2730
9	2861

000000000010.log(数据文件)

数据内容
message-10(数据-10)
message-11(数据-11)
message-12(数据-12)
message-13(数据-13)
message-14(数据-14)
message-15(数据-15)
message-16(数据-16)
message-17(数据-17)
message-18(数据-18)
message-19(数据-19)

_comsumer_offsets文件



> Consumer从kafka的磁盘中消费数据，所以不用担心数据丢失问题。
>
> 但是，Consumer作为一个消费者，是有可能出现宕机等问题的，也就意味着会出现重启后，继续消费的问题，那么就必须要消费者偏移量，消费到哪条数据了。
>
> 结论：`offset是用来记录Consumer的消费位置的，由Consumer自己负责维护(提交)，保存在kafka的broker的内置topic中`
>
> - 相关配置
>
> ```
> # consumer重启offset机制，三个可选值，过早的offset记录会被删除。
> auto.offset.reset=latest # 默认值，从最新的offset继续消费数据。
> ```

```
# 自动提交offset
1. 默认情况下Consumer的offset自动提交。
```

```
# ------------------配置参数-----------------------
# 自动提交开启
enable.auto.commit=true # 默认值
# 自动提交的时间间隔
auto.commit.interval.ms=5000 # 默认值5000 单位毫秒。
```

```
// java配置
config.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,"true");
```

 # 手动提交offset
通过代码的方式手动明确offset提交的方式。
`config.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,"false");`

提交方式	优点	缺点
同步提交	有失败重试机制，可以确保每次offset提交成功。	会影响消费者的消费速度。
异步提交	异步提交offset，不会阻挡继续消费，消费速度快。	可能会导致最新的offset没有提交成功，重启consumer之后，消费已经消费过的数据

```
// 同步提交：consumer提交完毕offset之后，才会继续消费数据。
//3. 消费数据
while (true){
    //JDK1.8 的API 毫秒数,
    ConsumerRecords<String, String> crs = kafkaConsumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> cr : crs) {
        System.out.println("cr = " + cr);
    }
    kafkaConsumer.commitAsync();
}
```

```
// 异步提交：consumer只需要发出提交offset的指令之后，就可以继续消费数据,不需要等待本地offset是否提交成功。
while (true){
    //JDK1.8 的API 毫秒数,
    ConsumerRecords<String, String> crs = kafkaConsumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> cr : crs) {
        System.out.println("cr = " + cr);
    }
    kafkaConsumer.commitSync();
}
```

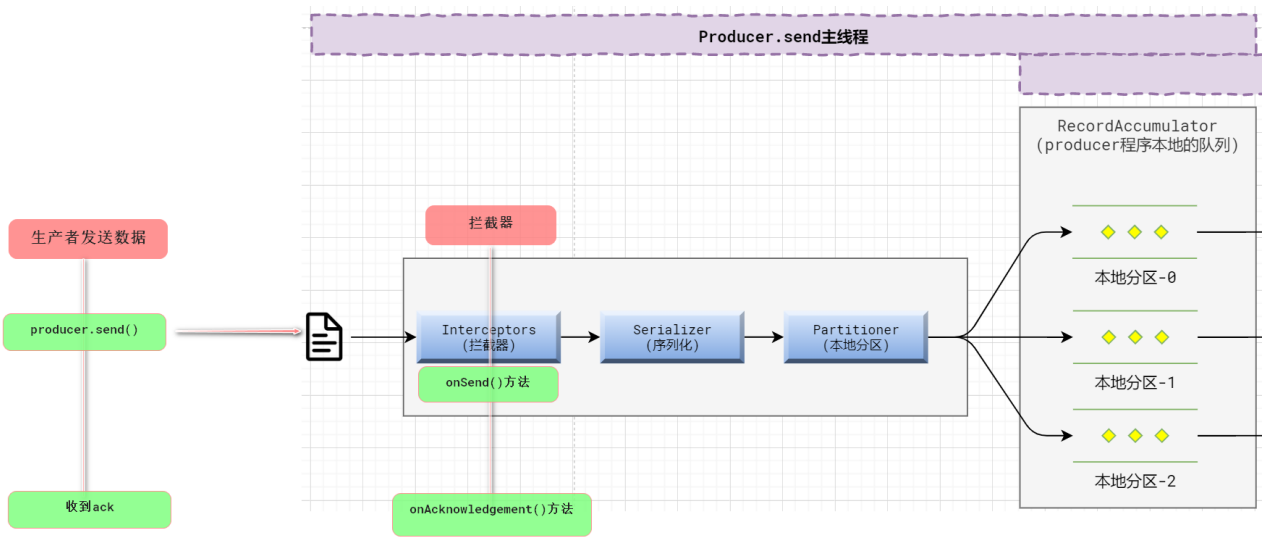
3. 拦截器

> 对Producer发送到Kafka的数据，进行前置处理和后置处理。
>
> 接口：`org.apache.kafka.clients.producer.ProducerInterceptor`
>
> 方法说明：

```
public class TimeInterceptor implements ProducerInterceptor<String, String> {
/**
*

    * @param record 生产者发送出来的ProducerRecord数据
```

```
     * @return 将处理后的数据封装成ProducerRecord继续处理
     */
    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {
        // 数据离开producer到达broker之前。准确的是，进入Serializer之前。
        return null;
    }

    /**
     *
     * @param metadata 响应的ack携带的数据描述信息(topic、分区、offset等)
     * @param exception 如果ack响应成功，则异常为null，否则就是真正的异常对象。
     */
    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        //响应的ack返回给producer之前执行
        String topic = metadata.topic();
        int partition = metadata.partition();
        long offset = metadata.offset();
        System.out.println(topic+" : "+partition+" : "+offset);

    }

    @Override
    public void close() {
        // Producer调用close或者Producer对象回收，调用该close方法。
    }

    @Override
    public void configure(Map<String, ?> configs) {
        //初始化拦截器时候，调用该方法。
    }
}
```

执行时机



编码

> 需求如下：
>
> 1. 对Producer发出的消息，添加一个时间戳：`124313782314+message`
> 2. 对kafka接收到的消息反回ack的时候，统计个数，以统计生产者发送数据的成功数据和失败数，并且打印失败的消息的offset。

 # 定义拦截器

```
public class TimeInterceptor implements ProducerInterceptor<String, String> {
    private Long successNum = 0L;
    private Long errorNum = 0L;
    /**
     *
     * @param record 生产者发送出来的ProducerRecord数据
     * @return 将处理后的数据封装成ProducerRecord继续处理
     */
    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {
        // 数据离开producer到达broker之前。准确的是，进入Serializer之前。
        ProducerRecord<String, String> records = new ProducerRecord<String, String>
(record.topic(),record.partition(),record.timestamp(),record.key(),System.currentTimeMillis()+re
cord.value(),record.headers());
        //System.out.println("records = " + records);
        return records;
    }
    /**
     *
     * @param metadata 响应的ack携带的数据描述信息(topic、分区、offset等)
     * @param exception 如果ack响应成功，则异常为null，否则就是真正的异常对象。
     */
    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        //响应的ack返回给producer之前执行
        String topic = metadata.topic();
        int partition = metadata.partition();
        long offset = metadata.offset();
        System.out.println(topic+" : "+partition+" : "+offset);

        if (exception == null){
            successNum++;
        }else{
            errorNum++;
        }
    }
    @Override
    public void close() {
        // Producer调用close或者Producer对象回收，调用该close方法。
        System.out.println("successNum = " + successNum);
        System.out.println("errorNum = " + errorNum);
        System.out.println("----------close---------");
```

```
    }
    @Override
    public void configure(Map<String, ?> configs) {
        //初始化拦截器时候，调用该方法。
        System.out.println("拦截器初始化：configs = " + configs);
    }
}
```

 # 使用拦截器

```
// 将拦截器的全类名注册到config中，多个拦截器的类名，使用逗号隔开。
config.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, "demo3.TimeInterceptor");
```