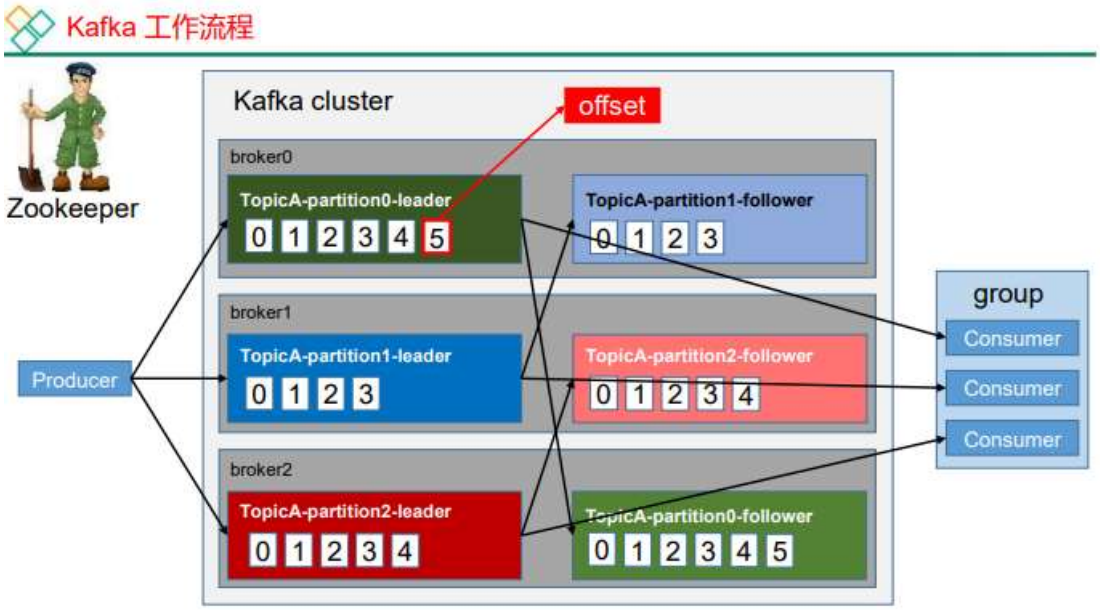
Kafka的深入

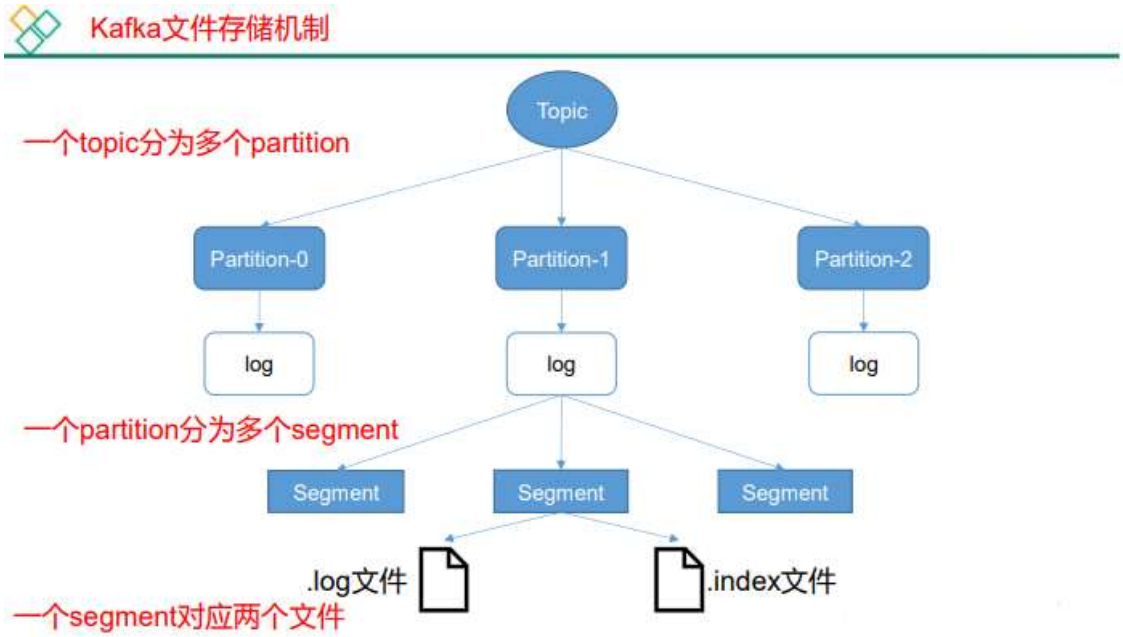
1）Kafka的结构

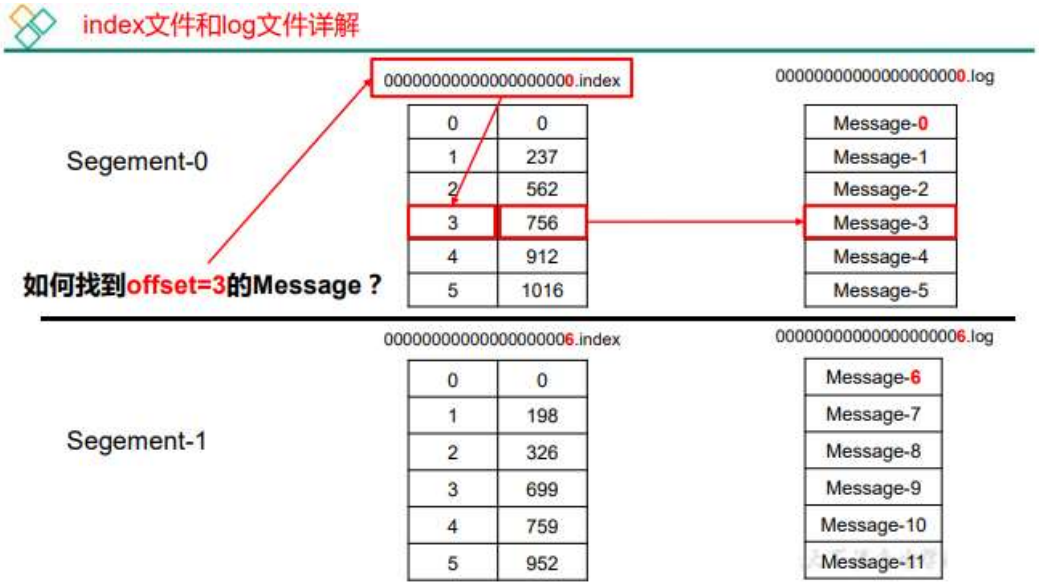
1.1）Kafka的工作流程



- topic是逻辑上的概念，而partition是物理上的概念，每个partition对应一个log文件，producer的每条数据会被写入log，且每条数据拥有自己的offset，消费者组的消费者会记录自己消费到了哪个offset，以便出错后可以恢复，从上次位置继续。

1.2）Kafka的文件存储





- 未防止log过大以及提高数据定位效率，Kafka采取分片和索引机制，将每个partition分成多个segment，每个segment有一个.log和.index文件，分别存储数据和索引，且.log和.index的文件命名为当前segment的第一条数据的offset。index文件存放每条数据的开始偏移，再去log文件定位数据

2）Kafka的生产者

2.1）分区的目的与原则

```
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、目的:分区可以通过调整以适配不同的机器，便于在集群中扩展，且以分区为单位可以提高并发。
> 2、原则:producer发送的消息会被封装成一个ProducerRecord对象，它的几种构造方式如图
> 2.1、指明partition
> 2.2、未指明partition，但有key。将key的hash与topic的partition取余得到parition值
> 2.3、key与parition均未指明，第一次取随机整数(此后每次均自增)，将此值与topic的partition取余得partition值。相当于轮询。

2.2）如何保证数据可靠性

> 1、为了保证producer的数据能准确的发到topic中，topic的每个partition收到数据后都会向producer发送acknowledgement(简称ack)确认收到，否则producer将重新发送
> 2、而由于parition通常是有副本的，副本的数据同步策略有两种：一是半数以上完成同步，leader就向producer发送ack；另一种是全部完成同步才发送ack。前者延迟低，但若leader挂了，在容忍n台故障的时候需要2n+1个副本，后者延迟高，但相同情况下只需要n+1个副本。Kafka不在乎延迟，选择第二种。
> 3、由于存在某个follower故障，导致leader无法收到全部同步的信号，持续等待的情况。Kafka引

入了ISR(in-sync replica set)机制，维护当前与leader保持同步的follower集合，若follower长时间未向leader确认同步，则被踢出ISR，超时由replica.lag.time.max.ms设定。新leader的选取也从ISR中选择。

4、Kafka允许设置不同的数据可靠性要求。通过修改acks的参数实现三种可靠性。

4.1、ack=0,producer不等broker回复ack，就直接发送下一条信息

4.2、ack=1,producer等待broker的leader收到后，不等follower同步就返回ack。可能造成返回后leader故障，follower还未同步，数据丢失

4.3、ack=-1,完全同步后才发送ack。可能在同步完成了，发出ack前leader故障了，producer又重发了一次，造成数据重复。有可能ISR里只有一个leader，也会存在数据丢失

2.3）数据一致性

1、为了避免partition每个副本由于宕机和同步造成的数据条数不一致，导致消费者获取数据不同。以所有副本中最小的LEO作为HW，HW前的数据才对消费者可见。

2、当leader故障时，选择ISR中一个follower做新leader，其余的截取高于HW的部分，再从新leader同步数据；当follower故障时，被踢出ISR。恢复后将高于HW的部分截取，从HW开始向leader同步，等LEO大于HW时，重新加入ISR

2.4）Exactly Once

1、ack=0时，实现at most once。ack=-1时，实现at least once。而为了实现Exactly Once，Kafka0.11之前由下游应用单独去重，后引入幂等性。即At Least Once + 幂等性 = Exactly Once。

2、producer初始化时被分配PID，发往同一个Partition时消息附带Sequence Number，而Broker端对<PID,Partition,Sequence Number>放入数据，只放一条。但producer重启PID会不同，且不同Partition具有不同主键。因此无法做到跨分区跨会话。

3、kafka0.11之后引入事务，保证生产和消费跨分区跨会话，要么全部成功，要么全部失败。引入一个全局唯一的TransactionID，将PID与TransactionID绑定，当Producer重启后，通过TransactionID获得原来的PID

3）Kafka的消费者

3.1）消费方式

> Kafka消费者采用拉取的方式从broker读取数据。若Kafka没有数据，则消费者可能陷入循环，一直返回空数据。针对这一点，Kafka消费者消费数据时传入timeout参数，当前无数据等待一段时间后再返回

3.2）分区分配策略

> 一个消费者组有多个消费者，一个topic有多个partition，如何确定由哪个消费者消费哪个partition，有两种分配策略:RoundRobin(轮询)和Range
> RoundRobin将所有主题的分区作为一个整体，进行轮询。使用前提是消费者组里每个消费者订阅的topic一样，不然有可能消费到不是自己订阅的topic信息
> Kafka默认策略Range，以单独topic的分区来划分。则可能导致每个消费者消费的数据量存在差距

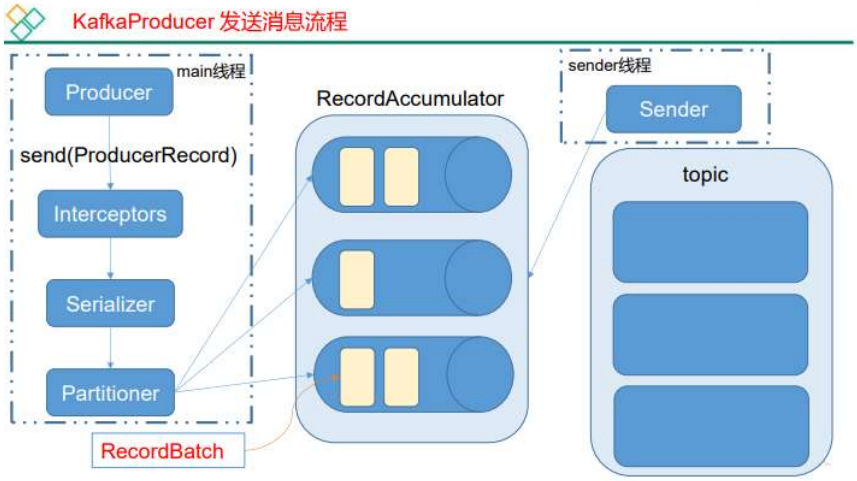
3.3）消费者offset

> 消费者组+主题+分区能唯一确定一个特定主题的特定分区，该组内的新消费者应该从哪里开始读取。Kafka0.9之前offset默认保存在zookeeper中，0.9之后保存到本地的__consunmer_offsets中。

4）Kafka的API(传统依赖)

4.1）消息发送流程

> 采用异步发送的方式，消息发送的过程中涉及main线程和sender线程，以及一个线程共享变量RecordAccumulator。main线程将消息发送到变量中，sender从变量中取数发送到broker
> 相关参数: `batch.size` ，数据积累到该量后，sender才会发送数据。
> `linger.ms` :若数据一直未达到batch.size，则等待设定时间后sender也会发送数据



4.2）生产者异步发送API

- 引入依赖

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0</version>
</dependency>
```

- 带回调/不带回调函数的API

```
    // 需要用的类KafkaProducer, ProducerConfig, ProducerRecord
    public static void main(String[] args){
        Properties props = new Properties();
        // 比起props.put("bootstrap.servers", "Master:9092");
        // 下面这种更方便且不容易出错
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "Master:9092");
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 1);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 1);
        // RecordAccumulator缓冲区大小
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33554432);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);
        for (int i = 0;i < 100;i++){
            // 不带回调
            producer.send(new ProducerRecord<String, String>("test",i));
            // 带回调,通过实现CallBack接口，参数如下
            // RecordMetadata:包含分区，offset等属性
            // exception:若发送成功为null
            producer.send(
                new ProducerRecord<>("test", "HOME" + i),
                (metadata, exception) -> {
                    if (exception == null){
                        // 处理回传的metadata
                    }
                });

        }
        // 如果不关闭，若信息不足batch.size或者时间不足linger.ms，则缓存清空，发不出去
        producer.close();
    }
```

- 自定义分区器

```
public class MyPartitioner implements Partitioner{
    public int partition(String topic, Object key,
        byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
        // 定义分区规则
    }
    public void close() {}
    public void configure(Map<String, ?> configs) {}
}

// 通过生产者的属性定义分区器
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "类全名");
```

4.3）生产者同步发送API

```
// 由于send方法返回Future对象，通过get()方法阻塞可实现同步发送
Future<RecordMetadata> future = producer.send(new ProducerRecord<>("test", i));
try{
    future.get();
} catch (Exception e) {}
```

4.4）消费者相关API

> 消费者需要考虑宕机后从哪重新开始取数据，因此重点在于对offset的维护

- 自动提交offset

```
    // 需要用的类KafkaConsumer，ConsumerConfig，ConsumerRecords，ConsumerRecord
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "Master:9092");
    // 允许offset自动提交
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
    // 自动提交offset的时间间隔
    props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
    // 消费者组
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "data");
    // 重置offset，可选earliest/latest，
    // 仅当该组第一次消费此主题，或者当前offset已失效，如数据已被删除等情况
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");
```

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
// 订阅的主题
consumer.subscribe(Collections.singletonList("test"));
// 轮询拉取数据
while (true){
    // 拉取延时
    ConsumerRecords<String, String> consumerRecords = consumer.poll(1000);
    consumerRecords.forEach(consumerRecord -> {
        System.out.println(consumerRecord.key() + "," + consumerRecord.value());
    });
}
```

- 手动提交offset

> 虽然自动提交简便，但难以把握自动提交时间，存在时间短可能提交了offset，但消费者还未处理完就故障了，下次从新的offset取数据，造成了消费者丢失数据；时间长可能消费者处理完了故障，但offset未提交，下次取到重复数据

```
// 使用手动提交前需要关闭自动提交
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
// 同步提交commitSync()
while (true){
    // 拉取延时
    ConsumerRecords<String, String> consumerRecords = consumer.poll(1000);
    consumerRecords.forEach(consumerRecord -> {
        // 处理数据
    });
    // 阻塞直到offset提交成功
    consumer.commitSync();
}
// 异步提交commitAsync(),实现OffsetCommitCallback接口
while (true){
    // 拉取延时
    ConsumerRecords<String, String> consumerRecords = consumer.poll(1000);
    consumerRecords.forEach(consumerRecord -> {
        // 处理数据
    });
    consumer.commitAsync((offsets, exception) -> {
        if (exception != null){
            System.err.println("commit offset failed," + offsets);
        }
    });
}
```

- 自定义存储offset

> 当有新的消费者加入消费者组、已有的消费者退出消费者组或者订阅的主题的分区发生变化，就会触发到分区的重新分配，重新分配的过程就叫做Rebalance

```
// 在订阅时配置自定义存储offset，实现该接口两个方法
consumer.subscribe(Collections.singletonList("test"),
    new ConsumerRebalanceListener() {
    @Override
    // Rebalance前调用，发送当前offset
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        commitOffset(currentOffset);
    }

    @Override
    // Rebalance后调用，将当前消费者定位到某分区最新的offset中继续消费
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions){
            consumer.seek(partition, getOffset(partition));
        }
    }
});
while (true){
    // 拉取延时
    ConsumerRecords<String, String> consumerRecords = consumer.poll(1000);
    consumerRecords.forEach(r -> {
        // 处理数据...
        // 写入offset
        currentOffset.put(new TopicPatition(r.topic(),r.partition()), r.offset());
    });
    // 异步方法
    commitOffset(currentOffset);
}

// commitOffset和getOffset为自己实现,例如在MySQL里存取offset
private static long getOffset(TopicPartition partition){}
private static void commitOffset(Map<TopicPartition, Long> currentOffset);
```

4.5）自定义拦截器Interceptor

```
public class MyInterceptor implements ProducerInterceptor {
    @Override
    public void configure(Map<String, ?> configs) {
        // 获取配置信息和初始化数据时调用
    }

    @Override
    public ProducerRecord onSend(ProducerRecord record) {
        // 该方法封装进KafkaProducer.send方法中，可在此对消息进行操作
        // 但最好不要修改消息的topic和分区，否在影响目标分区计算
    }
```

```
    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        // 该方法运行在RecordAccumulator发送到Kafka Broker之后
        // 或者运行在发送失败时，此方法运行在producer的IO线程中
        // 因此最好不要放入中逻辑，以免影响发送效率
    }

    @Override
    public void close() {
        // 用于执行一些资源清理工作
    }
}

// 生产者配置自定义的拦截器
props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
    Arrays.asList("拦截器类全名1", "拦截器类全名1"));
```

5）Kafka监控

5.1）Kafka Eagle

- 修改Kafka开启脚本

```
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"
fi
改成:
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-server -Xmx2G -Xms2G -XX:PermSize=128M
    -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:ParallelGCThreads=8
    -XX:ConcGCThreads=5 -XX:InitiatingHeapOccupancyPercent=70"
    export JMX_PORT="9999"
fi
```

- 安装并配置eagle

> 操作目录:kafka-eagle-bin压缩包里的kafka-eagle-web压缩包

```
# 修改名称
mv kafka-eagle-web-1.3.7/ eagle

# 配置环境变量
export KE_HOME=/opt/module/eagle
export PATH=$PATH:$KE_HOME/bin

# 授予启动权限
cd eagle/bin/
chmod 777 ke.sh
```

```
# 修改配置(增加)
cluster1.zk.list=Master:2181,Worker1:2181,Worker2:2181
cluster1.kafka.eagle.offset.storage=kafka
kafka.eagle.metrics.charts=true
kafka.eagle.driver=com.mysql.jdbc.Driver
kafka.eagle.url=jdbc:mysql://hadoop102:3306/ke?useUnicode=true&ch
aracterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
kafka.eagle.username=root
kafka.eagle.password=000000
```