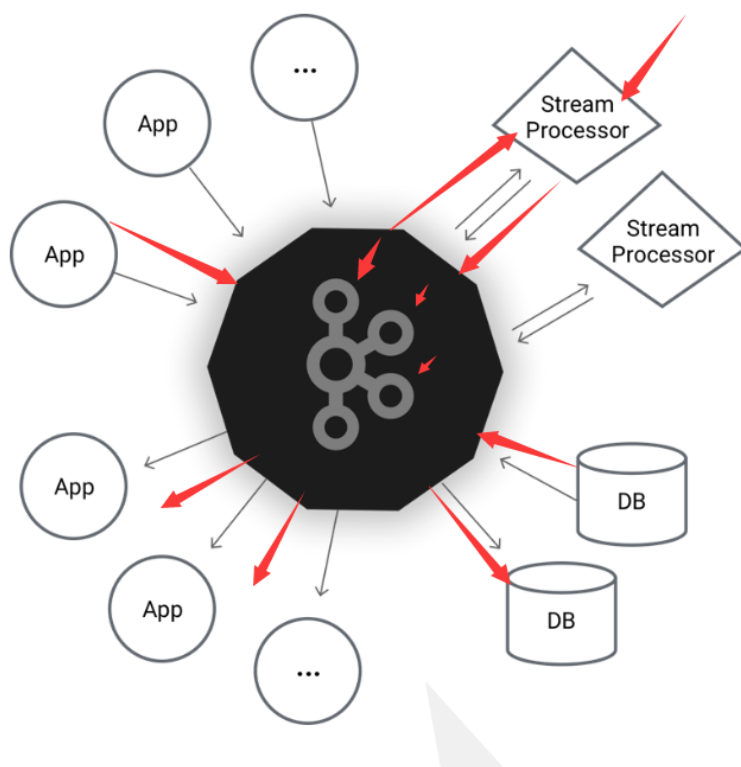


# Kafka上课记录

## 1 Kafka 入门

目标：掌握Kafka是什么，主要用途是什么，了解Kafka的特性

### 1.1 简介



### kafka是什么

- 一个分布式的流式数据处理平台。
- 可以用它来发布和订阅流式的记录。这一方面与消息队列或者企业消息系统类似
- 它将流式的数据安全地存储在分布式、有副本备份、容错的集群上

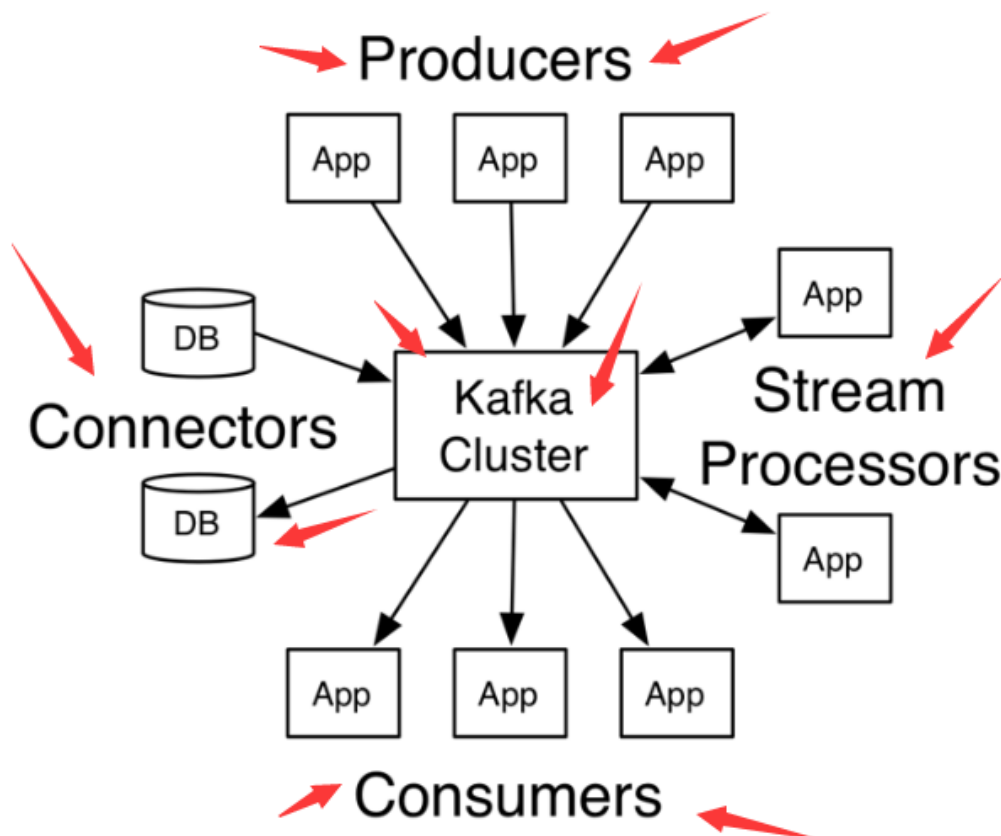
- 可以用来做流式计算

它可以用于两大类的应用:

1. 构造实时流数据管道，它可以在系统或应用之间可靠地获取数据。(相当于message queue)
2. 构建实时流式应用程序，对这些流数据进行转换或者影响。(就是流处理，通过kafka stream topic和topic之间内部进行变化)

## Kafka的体系结构

5大组件



## 1.2 安装

### 环境要求

- 生产环境linux 学习可以windows
- java1.8 及以上

### 安装

1. 下载安装包: [https://www.apache.org/dyn/closer.cgi?path=/kafka/2.3.0/kafka\\_2.12-2.3.0.tgz](https://www.apache.org/dyn/closer.cgi?path=/kafka/2.3.0/kafka_2.12-2.3.0.tgz)  
windows 和 linux都是同一个安装包，window命令在 bin/windows/ 下。
2. 安装

```
[root@node4 ~]# mkdir /usr/kafka
[root@node4 ~]#
[root@node4 ~]# tar -xzf kafka_2.12-2.3.0.tgz -C /usr/kafka/
[root@node4 ~]# ln -s /usr/kafka/kafka_2.12-2.3.0 /usr/kafka/latest
```

### 3. 了解目录结构

```
[root@node4 ~]# ll /usr/kafka/latest
drwxr-xr-x. 3 root root 4096 6月 20 04:44 bin
drwxr-xr-x. 2 root root 4096 6月 20 04:44 config
drwxr-xr-x. 2 root root 4096 8月 24 17:32 libs
-rw-r--r--. 1 root root 32216 6月 20 04:43 LICENSE
-rw-r--r--. 1 root root 337 6月 20 04:43 NOTICE
drwxr-xr-x. 2 root root 44 6月 20 04:44 site-docs
```

了解 bin、config、libs 下都有些什么。

## 启动

1、Kafka是用zookeeper来存储元数据，生产环境请一定要部署一个独立的zookeeper集群（至少3个节点）。

学习用，用Kafka里带的zookeeper。

首先要启动zookeeper。

```
[root@node4 ~]# cd /usr/kafka/latest/
[root@node4 latest]# bin/zookeeper-server-start.sh config/zookeeper.properties &
```

### 2 启动Kafka

```
[root@node4 latest]# bin/kafka-server-start.sh config/server.properties &
```

## 集群搭建

【生产集群】在其他机器上同样安装kafka，配置它们连接到同一个zookeeper集群、它们的唯一id，数据目录，启动Broker实例即加入集群。

【学习用集群】在同一台机器上启动多个broker实例，按如下步骤操作来搭建一个3节点的集群：

### 学习用集群搭建

#### 1. 拷贝配置文件

```
[root@node4 latest]# cp config/server.properties config/server-1.properties
[root@node4 latest]# cp config/server.properties config/server-2.properties
```

## 2. 修改配置文件:

config/server-1.properties:

```
broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=/var/kafka-logs-1
```

config/server-2.properties:

```
broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=/var/kafka-logs-2
```

## 3. 启动这两个broker实例

```
[root@node4 latest]# bin/kafka-server-start.sh config/server-1.properties &
[root@node4 latest]# bin/kafka-server-start.sh config/server-2.properties &
```

**【启动失败说明】** 如果启动第二个或第三个broker时提示内存不够用, 可以做如下调整:

- 1、调大你的虚拟机的内存 (1G 或更多)
- 2、调小Kafka的堆大小, 默认是1G, 生产用时可以调大。这里学习用可以调为256M (不能太小了, 启动时会heap OOM)

```
[root@node4 latest]# vi bin/kafka-server-start.sh
```

export KAFKA\_HEAP\_OPTS="-Xmx256M -Xms256M" 最大最小设为同一值 (减少内存的申请和回收)

```
[root@node4 latest]# bin/kafka-topics.sh --create --bootstrap-server
192.168.100.12:9092 --replication-factor 3 --partitions 1 --topic my-13-topic
[root@node4 latest]#
[root@node4 latest]#
[root@node4 latest]# bin/kafka-topics.sh --describe --bootstrap-server
192.168.100.12:9092 --topic my-13-topic
Topic:my-13-topic PartitionCount:1 ReplicationFactor:3
Configs:segment.bytes=1073741824
Topic: my-13-topic Partition: 0 Leader: 0 Replicas: 0,2,1 Isr: 0,2,1
```

--replication-factor 3 备份因子 副本数 3

--partitions 1 分片数 这个主题有几个分片

创建了一个有一个分片, 每个分片有3个副本的主题 my-13-topic。

分片是对主题Topic 的数据的分布式存储, 它是对数据的物理分割。但它本身是一个逻辑概念。

分片的存储实体是副本 3个副本，就是这个分片的数据会存3份。

Topic: my-13-topic Partition: 0 Leader: 0 Replicas: 0,2,1 Isr: 0,2,1

Partition: 0 分片号

Leader: 0 leader 副本在 0号Broker上

Replicas: 0,2,1 三个副本位于 0 2 1 号broker上

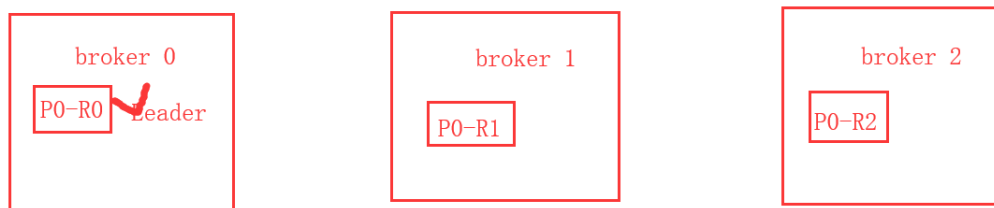
Isr: 0,2,1 in sync 处于同步状态的broker

分片和broker的关系

broker: 一般是一台计算机上一个Broker

分片: 分片的一个副本存放在一个broker上。

创建一个3个分片，3个副本的主题



【强调】 分片数 是在创建 主题时 根据你的估算数据量来选择的。

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning  
--topic my-replicated-topic
```

## 1.3 监控管理工具

Kafka自身未提供图形化的监控管理工具，市面上有很多开源的监控管理工具，但都不怎么成熟可靠。这里给介绍一款稍可靠的工具。

**Kafka Offset Monitor**

<https://github.com/quantifind/KafkaOffsetMonitor>

可以实时监控：

- Kafka集群状态
- Topic、Consumer Group列表

- 图形化展示topic和consumer之间的关系
- 图形化展示consumer的Offset、Lag等信息

它是一个jar 包，使用很简单

```
java -cp KafkaOffsetMonitor-assembly-0.2.1.jar \
    com.quantifind.kafka.offsetapp.OffsetGetterWeb \
    --offsetStorage kafka
    --zk zk-server1,zk-server2 \
    --port 8080 \
    --refresh 10.seconds \
    --retain 2.days
```

### 0.2.0 版本启动命令

```
java -cp KafkaOffsetMonitor-assembly-0.2.0.jar \
    com.quantifind.kafka.offsetapp.OffsetGetterWeb \
    --zk zk-server1,zk-server2 \
    --port 8088 \
    --refresh 10.seconds \
    --retain 2.days
```

The arguments are:

- **offsetStorage** valid options are "zookeeper", "kafka" or "storm". Anything else falls back to "zookeeper" 【说明】0.2.1版本才有这个参数
- **zk** the ZooKeeper hosts
- **port** on what port will the app be available
- **refresh** how often should the app refresh and store a point in the DB
- **retain** how long should points be kept in the DB
- **dbName** where to store the history (default 'offsetapp')
- **kafkaOffsetForceFromStart** only applies to "kafka" format. Force KafkaOffsetMonitor to scan the commit messages from start (see notes below)
- **stormZKOffsetBase** only applies to "storm" format. Change the offset storage base in zookeeper, default to "/stormconsumers" (see notes below)
- **pluginsArgs** additional arguments used by extensions (see below)

启动后就可以在浏览器中访问了: <http://localhost:8088>

## 1.4 spring中使用

发生的时候 type

```
foo:com.study.kafka.sample_01_pub_sub.common.Foo1,bar:com.study.kafka.sample_02_multi_m
ethod_listener.common.Bar1
```

接收消息的时候 type=foo:com.study.kafka.sample\_01\_pub\_sub.common.Foo1 type=bar

同学想了解的MessageConverter的调用

```
Thread [multiGroup-0-C-1] (Suspended (breakpoint at line 105 in MessagingMessageConverter))
  StringJsonMessageConverter(MessagingMessageConverter).toMessage(ConsumerRecord<?,?>, Acknowledgment, Consumer<?,?>, Type) line: 105
  RecordMessagingMessageListenerAdapter<K,V>(MessagingMessageListenerAdapter<K,V>).toMessagingMessage(ConsumerRecord<K,V>, Acknowledgment, Consumer<?,?>, Type) line: 50
  RecordMessagingMessageListenerAdapter<K,V>.onMessage(ConsumerRecord<K,V>, Acknowledgment, Consumer<?,?>) line: 74
  RecordMessagingMessageListenerAdapter<K,V>.onMessage(Object, Acknowledgment, Consumer) line: 50
  KafkaMessageListenerContainer$ListenerConsumer.doInvokeOnMessage(ConsumerRecord<K,V>) line: 1275
  KafkaMessageListenerContainer$ListenerConsumer.invokeOnMessage(ConsumerRecord<K,V>, Producer) line: 1258
  KafkaMessageListenerContainer$ListenerConsumer.doInvokeWithRecords(ConsumerRecords<K,V>) line: 1200
  KafkaMessageListenerContainer$ListenerConsumer.doInvokeRecordListener(ConsumerRecord<K,V>, Producer, Iterator<ConsumerRecord<K,V>>) line: 1120
  KafkaMessageListenerContainer$ListenerConsumer.invokeRecordListener(ConsumerRecords<K,V>) line: 935
  KafkaMessageListenerContainer$ListenerConsumer.invokeListener(ConsumerRecords<K,V>) line: 935
  KafkaMessageListenerContainer$ListenerConsumer.pollAndInvoke() line: 751
  KafkaMessageListenerContainer$ListenerConsumer.run() line: 700
  Executors$RunnableAdapter<T>.call() line: 511
  ListenableFutureTask<T>(FutureTask<V>).run() line: 266
  Thread.run() line: 745
```

## 1.5 java客户端

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.3.0</version>
</dependency>
```

### 核心 API

#### Topic及管理

- AdminClient
- NewTopic

#### Client

- CommonClientConfigs

#### Producer

- KafkaProducer
- ProducerRecord
- ProducerConfig
- Serializer

- 
- ✓ ⓘ Serializer<T> - org.apache.kafka.common.serialization
    - 🕒 ByteArraySerializer - org.apache.kafka.common.serialization
    - 🕒 ByteBufferSerializer - org.apache.kafka.common.serialization
    - 🕒 BytesSerializer - org.apache.kafka.common.serialization
    - 🕒 DoubleSerializer - org.apache.kafka.common.serialization
    - 🕒 FloatSerializer - org.apache.kafka.common.serialization
    - 🕒 IntegerSerializer - org.apache.kafka.common.serialization
    - 🕒 LongSerializer - org.apache.kafka.common.serialization
    - 🕒 ShortSerializer - org.apache.kafka.common.serialization
    - 🕒 StringSerializer - org.apache.kafka.common.serialization
  - ✓ ⓘ ExtendedSerializer<T> - org.apache.kafka.common.serialization
    - 🕒 JsonSerializer<T> - org.springframework.kafka.support.serializer
    - 🕒<sup>s</sup> Wrapper<T> - org.apache.kafka.common.serialization.ExtendedSerializer

## Consumer

- KafkaConsumer
- ConsumerConfig
- ConsumerRecord
- Deserializer

- 
- ✓ ⓘ Deserializer<T> - org.apache.kafka.common.serialization
    - 🕒 ByteArrayDeserializer - org.apache.kafka.common.serialization
    - 🕒 ByteBufferDeserializer - org.apache.kafka.common.serialization
    - 🕒 BytesDeserializer - org.apache.kafka.common.serialization
    - 🕒 DoubleDeserializer - org.apache.kafka.common.serialization
    - 🕒 FloatDeserializer - org.apache.kafka.common.serialization
    - 🕒 IntegerDeserializer - org.apache.kafka.common.serialization
    - 🕒 LongDeserializer - org.apache.kafka.common.serialization
    - 🕒 ShortDeserializer - org.apache.kafka.common.serialization
    - 🕒 StringDeserializer - org.apache.kafka.common.serialization
  - ✓ ⓘ ExtendedDeserializer<T> - org.apache.kafka.common.serialization
    - 🕒 ErrorHandlingDeserializer<T> - org.springframework.kafka.support.serializer
    - 🕒 ErrorHandlingDeserializer2<T> - org.springframework.kafka.support.serializer
    - 🕒 JsonSerializer<T> - org.springframework.kafka.support.serializer
    - 🕒<sup>s</sup> Wrapper<T> - org.apache.kafka.common.serialization.ExtendedDeserializer

## 2 核心概念

### 2.1 Topic

#### 2.1.1 Topic 说明



```
[root@node4 latest]# bin/kafka-topics.sh --create --bootstrap-server
localhost:9092 --replication-factor 3 --partitions 3 --topic my-33-topic
```

```
[root@node4 latest]# bin/kafka-topics.sh --describe --bootstrap-server
localhost:9092 --topic my-33-topic
Topic:my-33-topic PartitionCount:3 ReplicationFactor:3
Configs:segment.bytes=1073741824
  Topic: my-33-topic Partition: 0 Leader: 0 Replicas: 0,2,1 Isr: 0,2,1
  Topic: my-33-topic Partition: 1 Leader: 2 Replicas: 2,1,0 Isr: 2,1,0
  Topic: my-33-topic Partition: 2 Leader: 1 Replicas: 1,0,2 Isr: 1,0,2
```

- **Topic** : 主题，一类消息（数据）
- **partition** : 一个Topic可以分成多个分片来分布式存放数据，分片以顺序号来编号。
- **Replicas** : 为保证数据存储的可靠性，一个分片可以存储多个副本（一般3个），副本被自动均衡分布在集群节点上
- **Leader** : 一个分片的多个副本中自动选举一个作为Leader，通过Leader操作数据，Leader同步给其他副本，以此来保证一致性。当Leader挂了时，自动选择一个做Leader。
- Topic的这些元信息存储在Zookeeper上。
- Replicas: 0,2,1 副本在哪些broker上
- Isr: 0,2,1 副本 存活且同步的broker

## 2.1.2 Leader选举

【问题】每个分片的Leader如何产生？

怎么进行Leader选举？

zk leader 选举的原理是什么？

临时节点 + watch

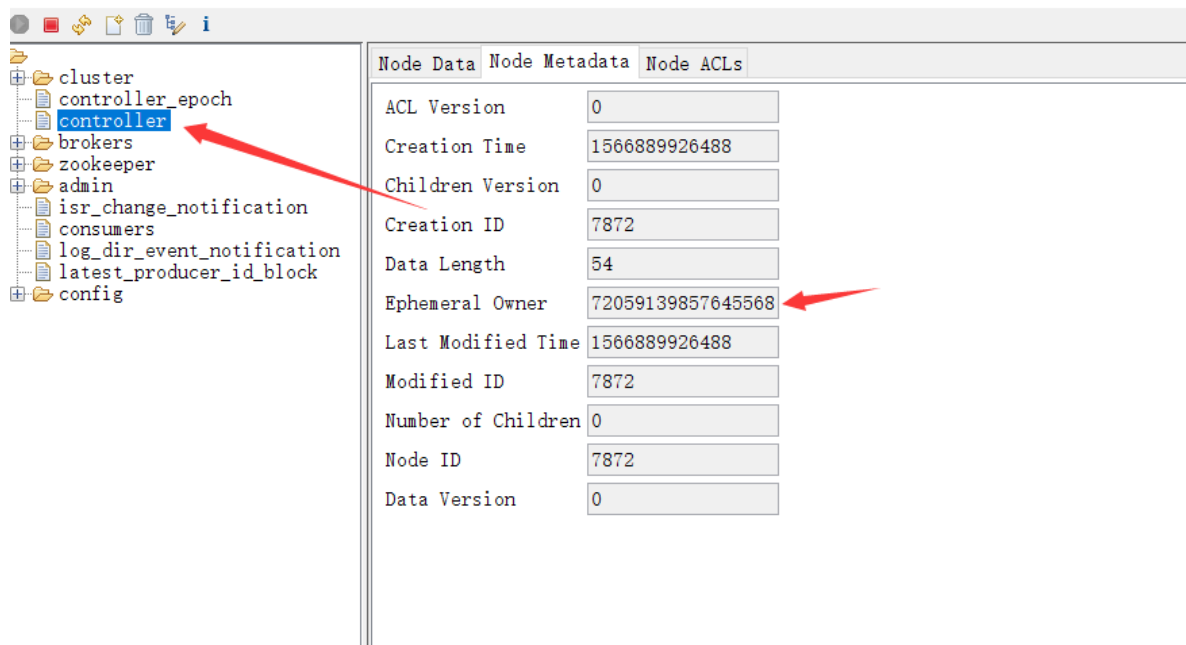
因为惊群效应，Kafka没有直接使用zk来进行分片的Leader选举

Kafka中增加一个角色：Controller，由集群的一个broker来担任这个角色

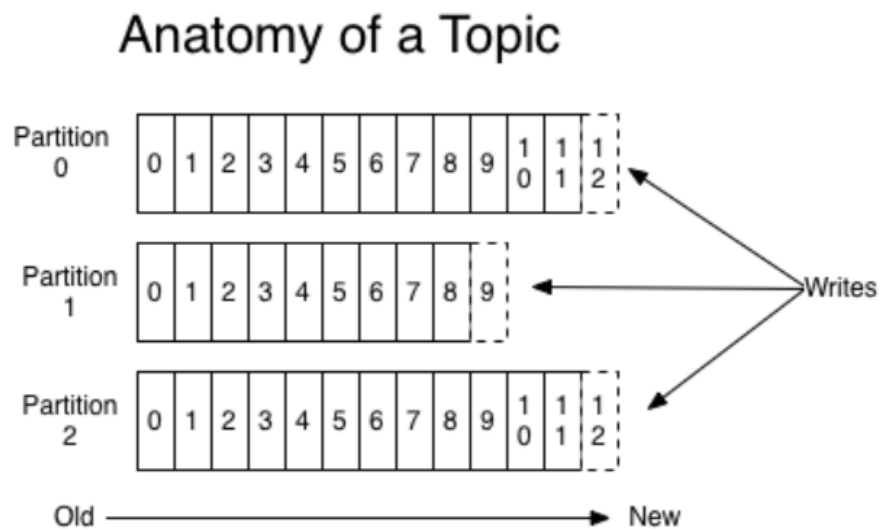
谁来当Controller 怎么定？挂了怎么办？

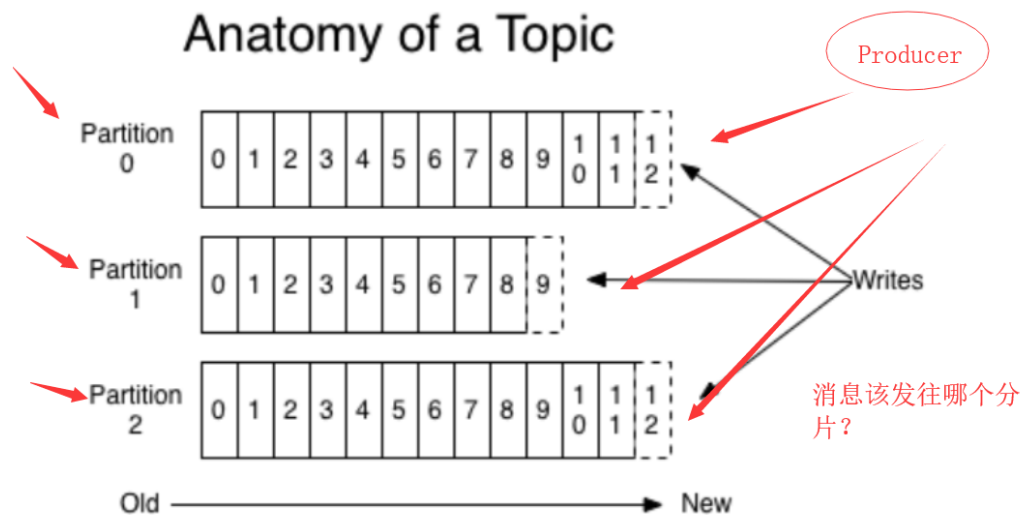
这里就是真正用的zk 来进行 Controller的选举。

然后所有的 主题的分片的副本的分布、leader的选定都由Controller来完成。



### 2.1.3 消息的分片选择





## Producer客户端负责消息的分发

- kafka集群中的任何一个broker都可以向producer提供metadata信息,这些metadata中包含“集群中存活的servers列表”/“partitions leader列表”等信息;
- 当producer获取到metadata信息之后, producer将会和Topic下所有partition leader保持socket连接;
- 消息由producer直接通过socket发送到broker, 中间不会经过任何“路由层”, 事实上, 消息被路由到哪个partition上由producer客户端决定; 比如可以采用“random”“key-hash”“轮询”等,如果一个topic中有多个partitions, 那么在producer端实现“消息均衡分发”是必要的。

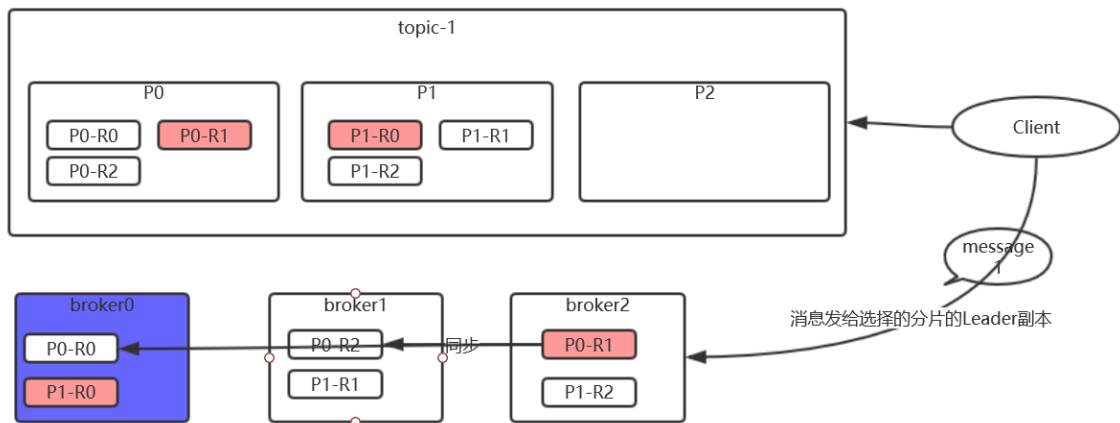
### 消息的分片选择规则:

- 用户给定了分片号且正确有效，则发到给定分片；
- 未指定分片，指定了Key，则对Key取Hash 求余决定目标分片
- 未指定分片，也未提供key，则采用轮询

```
ProducerRecord<K, V>
```

## ▼ KafkaTemplate<K, V>

- `sendDefault(V) : ListenableFuture<SendResult<K, V>>`
- `sendDefault(K, V) : ListenableFuture<SendResult<K, V>>`
- `sendDefault(Integer, K, V) : ListenableFuture<SendResult<K, V>>`
- `sendDefault(Integer, Long, K, V) : ListenableFuture<SendResult<K, V>>`
- `send(String, V) : ListenableFuture<SendResult<K, V>>`
- `send(String, K, V) : ListenableFuture<SendResult<K, V>>`
- `send(String, Integer, K, V) : ListenableFuture<SendResult<K, V>>`
- `send(String, Integer, Long, K, V) : ListenableFuture<SendResult<K, V>>`
- `send(ProducerRecord<K, V>) : ListenableFuture<SendResult<K, V>>`
- `send(Message<?>) : ListenableFuture<SendResult<K, V>>`
- `sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata>) : void`
- `sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata>, String) : void`



### 2.1.4 分片数据持久化存储原理

【问题】副本怎么存储消息数据？

Kafka 是采用文件来存储数据

数据量大

#### 磁盘文件组织方式

Kafka是一个分布式的流数据存储平台，它将流数据以日志的方式顺序存储在磁盘文件中。数据文件的数据存放目录下的组织方式为：

```
drwxr-xr-x. 2 root root 141 8月 27 15:12 my-33-topic-0
drwxr-xr-x. 2 root root 141 8月 27 15:12 my-33-topic-1
drwxr-xr-x. 2 root root 141 8月 27 15:12 my-33-topic-2
```

```
[root@node4 kafka-logs]# ll my-33-topic-0
总用量 4
-rw-r--r--. 1 root root 10485760 8月 25 15:41 00000000000000000000.index
-rw-r--r--. 1 root root          0 8月 25 15:41 00000000000000000000.log
-rw-r--r--. 1 root root 10485756 8月 25 15:41 00000000000000000000.timeindex
-rw-r--r--. 1 root root          8 8月 25 15:41 leader-epoch-checkpoint
[root@node4 kafka-logs]#
[root@node4 kafka-logs]# ll my-33-topic-1
总用量 0
-rw-r--r--. 1 root root 10485760 8月 25 15:41 00000000000000000000.index
-rw-r--r--. 1 root root          0 8月 25 15:41 00000000000000000000.log
-rw-r--r--. 1 root root 10485756 8月 25 15:41 00000000000000000000.timeindex
-rw-r--r--. 1 root root          0 8月 25 15:41 leader-epoch-checkpoint
```

**【说明】消息数据是顺序追加到 .log 文件中，这用写入速度非常快。**

像写日志一样，追加流数据。

文件的顺序写 远快于 随机写

**【问题1】**日志文件名 这串 00000000000000000000 表示什么意思？为什么这么命名？

**【问题2】**在这个日志文件中怎么存储数据？怎么知道一条消息的结尾。

存得都是字节

offset 偏移量 记录

4个字节（消息内容的长度） + 消息内容（字节序列）

## 日志文件数据存储格式

每个日志文件都是“log entries”序列，每一个log entry包含一个4字节整型数（值为N），其后跟N个字节的消息体。每条消息都有一个当前partition下唯一的64字节的offset，它指明了这条消息的起始位置。磁盘上存储的消息格式如下：

**消息长度: 4 bytes (value: 1 + 4 + n)**

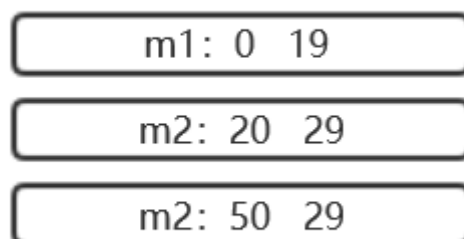
**版本号: 1 byte**

**CRC 校验码: 4 bytes**

**具体的消息: n bytes**

**【问题】**每条消息都有一个当前partition下唯一的64字节的offset，它指明了这条消息的起始位置。那这个offset值存哪里？

下面是一个消息的偏移量图示

[illegible]

这个信息存储到.index索引文件中

## 索引

【思考】索引中存储的数据的结构是怎样的？

{消息序号, 存储偏移地址}

**【思考】**有必要在索引中存储每一条消息的偏移地址吗？

m1: 0 19

m3: 50 29

kafka用的稀疏索引。

【思考】 kafka作为一个分布式的流数据存储平台，它能存储海量的消息数据，那一个分片的数据可能会很大吗？

一个很大的分片，也即一个很大的文件，操作方便吗？

### Segment 段

分片分成多个段（一个.log文件）来存储，段的大小固定（可以指定）



每个partion(目录)相当于一个巨型文件被平均分配到多个大小相等segment(段)数据文件中，每个segment文件名为该segment第一条消息的offset和“.log”组成。但每个段segment file消息数量不一定相等，这种特性方便old segment file快速被删除。（默认情况下每个文件大小为1G）

每个partiton只需要支持顺序读写就行了，segment文件生命周期由服务端配置参数决定。

这样做的好处就是能快速删除无用文件，有效提高磁盘利用率。

### 消息什么时候删除

- 通过在server.properties文件中配置全局默认的日志保留策略来控制：

支持两种策略：时间 和 大小 。可多策略，哪个达到即哪个生效。

```
##### Log Retention Policy #####
```

```
# The following configurations control the disposal of log segments. The policy
can
# be set to delete segments after a period of time, or after a given size has
accumulated.
# A segment will be deleted whenever *either* of these criteria are met.
Deletion always happens
# from the end of the log.

# The minimum age of a log file to be eligible for deletion due to age
# 策略1 留存多长时间
log.retention.hours=168

# A size-based retention policy for logs. Segments are pruned from the log
unless the remaining
# segments drop below log.retention.bytes. Functions independently of
log.retention.hours.
# 策略2 留存多少字节
#log.retention.bytes=1073741824

# The maximum size of a log segment file. When this size is reached a new log
segment will be created.
log.segment.bytes=1073741824

# The interval at which log segments are checked to see if they can be deleted
according
# to the retention policies
log.retention.check.interval.ms=300000
```

- 在定义Topic时指定对应参数

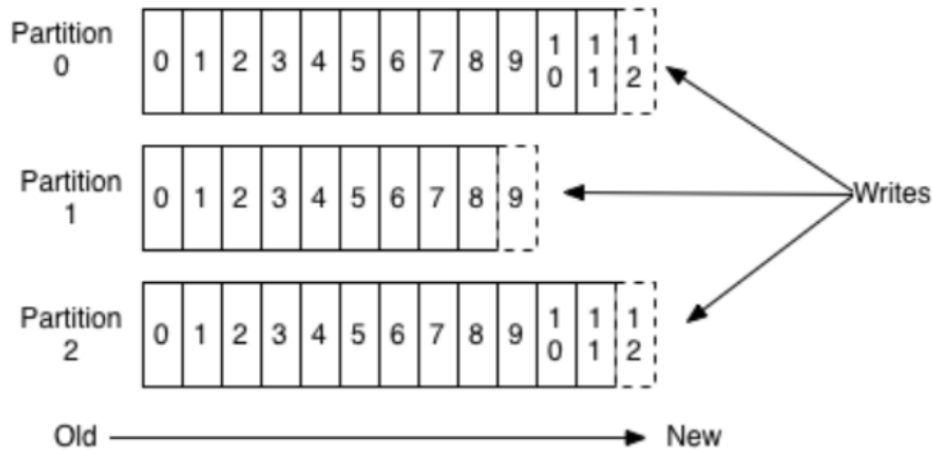
retention.bytes	如果使用“delete”保留策略，此配置控制分区(由日志段组成)在放弃旧日志段以释放空间之前的最大大小。默认情况下，没有大小限制，只有时间限制。由于此限制是在分区级别强制执行的，因此，将其乘以分区数，计算出topic保留值，以字节为单位。	long	-1		log.retention.bytes	medium
retention.ms	如果使用“delete”保留策略，此配置控制保留日志的最长时间，然后将旧日志段丢弃以释放空间。这代表了用户读取数据的速度的SLA。	long	604800000		log.retention.ms	medium
segment.bytes	此配置控制日志的段文件大小。保留和清理总是一次完成一个文件，所以更大的段大小意味着更少的文件，但对保留的粒度控制更少。	int	1073741824	[14,...]	log.segment.bytes	medium

## 消息的序号

在每个分片中会给每条消息一个**递增的序号**



# Anatomy of a Topic



【重点】消费者的offset（序号偏移量）

## TimeIndex 时间索引

【问题】如果我们想要消费从某时刻开始的消息，该怎么办？

- 消息要有时间戳
- 建立时间索引 .timeindex 数据结构 {时间戳，序号}

当要获取某时刻开始的消息时，根据时间戳到时间索引中获得  $\geq$  该时刻的第一个消息序号；然后从该序号开始拉取数据。

## Broker中关于Timestamp的全局默认配置参数：

名称	描述	类型	默认值	有效值	重要性
log.message.timestamp.difference.max.ms	broker收到消息时的时间戳和消息中指定的时间戳之间允许的最大差异。当log.message.timestamp.type=CreateTime,如果时间差超过这个阈值，消息将被拒绝。如果log.message.timestamp.type = logappendtime，则该配置将被忽略。允许的最大时间戳差值，不应大于log.retention.ms，以避免不必要的频繁日志滚动。	long	9223372036854775807		中
log.message.timestamp.type	定义消息中的时间戳是消息创建时间还是日志追加时间。该值应该是“createtime”或“logappendtime”。	string	CreateTime	[CreateTime, LogAppendTime]	中

## Topic中关于Timestamp的配置参数：

名称	描述	类型	默认值	有效值	服务器默认属	重要性
message.timestamp.difference.max.ms	broker接收消息时所允许的时间戳与消息中指定的时间戳之间的最大差异。如果message.timestamp.type=CreateTime，则如果时间戳的差异超过此阈值，则将拒绝消息。如果message.timestamp.type=LogAppendTime，则忽略此配置。	long	9223372036854775807	[0,...]	log.message.timestamp.difference.max.ms	medium
message.timestamp.type	定义消息中的时间戳是消息创建时间还是日志追加时间。值应该是“CreateTime”或“LogAppendTime”	string				

## 2.1.5 Topic的配置参数【了解】

### 参数列表

<http://kafka.apachecn.org/documentation.html#configuration>

## 参数修改

与Topic相关的配置既包含服务器默认值，也包含可选的每个Topic覆盖值。如果没有给出每个Topic的配置，那么服务器默认值就会被使用。通过提供一个或多个 `--config my-topic`

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic my-topic --
partitions 1 --replication-factor 1 --config max.message.bytes=64000 --config
flush.messages=1
```

也可以在使用alter configs命令稍后更改或设置覆盖值. 本示例重置my-topic

```
> bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-
name my-topic --alter --add-config max.message.bytes=128000
```

您可以执行如下操作来检查topic设置的覆盖值

```
bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --entity-
name my-topic --describe
```

您可以执行如下操作来删除一个覆盖值

```
> bin/kafka-configs.sh --zookeeper localhost:2181 --entity-type topics --
entity-name my-topic --alter --delete-config max.message.bytes
```

## 2.1.6 删除Topic

```
[root@node4 latest]# bin/kafka-topics.sh --create --bootstrap-server
localhost:9092 --replication-factor 1 --partitions 1 --topic test-1
```

```
# 发送一下消息
[root@node4 latest]# bin/kafka-console-producer.sh --broker-list localhost:9092
--topic test-1
```

删除Topic

```
[root@node4 latest]# bin/kafka-topics.sh --delete --bootstrap-server
localhost:9092 --topic test-1
```

**说明：**

当Topic是新创建的空Topic时，元信息和Topic的存储目录都会删除。

当Topic已有数据时，元信息在zookeeper上被删除，数据存储目录被标识为delete

```
drwxr-xr-x. 2 root root 141 8月 26 20:49 test-1-
0.20bb2388c6f04c34b79419411bc1bda6-delete
```

## 2.2 Producer

生产者可以将数据发布到所选择的topic（主题）中。生产者负责将记录分配到topic的哪一个partition（分片）中。

### 2.2.1 示例代码

### 2.2.2 生产者消息发布确认机制

设置发送数据是否需要服务端的反馈,有三个值 [all,0,1,-1] , 默认 1

- 0: producer不会等待broker发送ack
- 1: 当leader接收到消息之后发送ack
- all: leader接收到消息后, 等待所有in-sync的副本同步完成之后 发送ack。这样可以提供最好的可靠性。This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee. This is equivalent to the acks=-1 setting.
- -1: 等价于 all


#### Callback异步处理的确认结果


在发送时, 你可以完全非阻塞, 通过指定回调来处理发送结果, 但当使用事务时, 则没必要使用Callback, 因为结果在事务方法中处理。

```
public Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback  
callback)
```

发送到同一分片的结果处理将按顺序执行回调。

---

 **Callback**

```
 onCompletion(RecordMetadata, Exception) : void
```

请详细阅读它的注释。

#### spring中的处理方式

```
/**  
 * Set a {@link ProducerListener} which will be invoked when Kafka acknowledges  
 * a send operation. By default a {@link LoggingProducerListener} is configured  
 * which logs errors only.  
 * @param producerListener the listener; may be {@code null}.  
 */  
public void setProducerListener(@Nullable ProducerListener<K, V>  
producerListener) {  
    this.producerListener = producerListener;  
}
```

#### ProducerInterceptor

如果想对消息的发送过程增加额外的统一处理逻辑可以提供 ProducerInterceptor 实现

```
▼ ⓘ ProducerInterceptor<K, V>
  ● A onSend(ProducerRecord<K, V>) : ProducerRecord<K, V>
  ● A onAcknowledgement(RecordMetadata, Exception) : void
  ● A close() : void
```

请详细了解它的注释说明。

**使用方式**，通过`interceptor.classes`指定实现类名：

```
//配置 ProducerInterceptor
props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
"com.study.xxx.MyProducerInterceptor1,com.study.xxx.MyProducerInterceptor2");
```

下面是这个参数的说明

```
/** <code>interceptor.classes</code> */
    public static final String INTERCEPTOR_CLASSES_CONFIG =
"interceptor.classes";
    public static final String INTERCEPTOR_CLASSES_DOC = "A list of classes to
use as interceptors. "
                                                                + "Implementing the
<code>org.apache.kafka.clients.producer.ProducerInterceptor</code> interface
allows you to intercept (and possibly mutate) the records "
                                                                + "received by the
producer before they are published to the kafka cluster. By default, there are
no interceptors.";
```

### 2.2.3 幂等模式 【了解】

从Kafka 0.11开始，KafkaProducer支持另外两种模式:幂等生成器和事务生成器。幂等生成器将Kafka的传递语义从至少一次增强到恰好一次。特别是生产者重试将不再引入重复。事务生成器允许应用程序以原子方式向多个分区(和主题!)发送消息。

配置生产者参数`enable.idempotence`为`true`。 `retries` `acks` 都不要配置了，因为会有自动的默认值：`Integer.MAX_VALUE` `all`。 不可以业务上重发相同的业务数据。

### 2.2.4 事务模式

事务模式是为了让发往多个分片、多个Topic的多条消息具有原子性。

**事务模式要求：**

- 要使用事务模式和对应的api，必须设置 `transactional.id` 属性。 `transactional.id` 配置后，将自动启用幂等性，同时启用幂等性所依赖的生成器配置。`transactional.id` 是事务标识，用于跨单个生产者实例的多个会话启用事务恢复。对于分区应用程序中运行的每个生成器实例，它应该是惟一的。
- Kafka集群需要有至少3个节点
- 为了从端到端实现事务保证，还必须将消费者配置为只读取提交的消息。

所有事务API是同步阻塞的。

Producer是线程安全。

## 2.3 Consumer

Kafka中消费者是采用 poll 拉模式来获取消息。

### 2.3.1 Consumer示例

【注意】Consumer是非线程安全的。多线程中的情况下，一个线程一个Consumer。

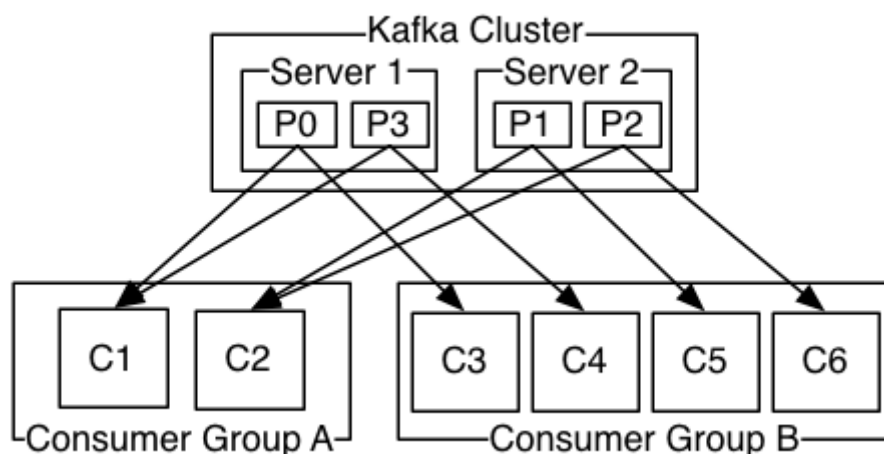
### 2.3.2 消费组

#### 消费组的说明

消费者使用一个 *消费组* 名称来进行标识，发布到topic中的每条记录被分配给订阅消费组中的一个消费者实例。消费者实例可以分布在多个进程中或者多个机器上。

如果所有的消费者实例在同一消费组中，消息记录会负载平衡到每一个消费者实例。

如果所有的消费者实例在不同的消费组中，每条消息记录会广播到所有的消费者进程。



通常情况下，每个 topic 都会有一些消费组，一个消费组对应一个"逻辑订阅者"。一个消费组由许多消费者实例组成，便于扩展和容错。这就是发布和订阅的概念，只不过订阅者是一组消费者而不是单个的进程。

在Kafka中实现消费的方式是将日志中的分片划分到每一个消费者实例上，以便在任何时间，每个实例都是分片唯一的消费者。维护消费组中的消费关系由Kafka协议动态处理。如果新的实例加入组，他们将从组中其他成员处接管一些 partition 分区;如果一个实例消失，拥有的分区将被分发到剩余的实例。

Kafka 只保证分区内的记录是有序的，而不保证主题中不同分区的顺序。如果你希望所有消息都有序消费，可使用仅有一个分区的主题来实现，这意味着每个消费者组只有一个消费者进程。

#### group rebalance

【注意】消费组的重平衡延时参数设置

```
##### Group Coordinator Settings
#####

# The following configuration specifies the time, in milliseconds, that the
GroupCoordinator will delay the initial consumer rebalance.
# The rebalance will be further delayed by the value of
group.initial.rebalance.delay.ms as new members join the group, up to a maximum
of max.poll.interval.ms.
# The default value for this is 3 seconds.
# We override this to 0 here as it makes for a better out-of-the-box experience
for development and testing.
# However, in production environments the default value of 3 seconds is more
suitable as this will help to avoid unnecessary, and potentially expensive,
rebalances during application startup.
# 消费组的重平衡延时（单位毫秒），【注意】生产环境请恢复为默认值3秒，或根据实际需要增大
group.initial.rebalance.delay.ms=0
```

触发重平衡的事件：

- 消费者数量变化
- 分片的增减

【思考】Kafka怎么知道消费者离线了，需要rebalance了？

## Beatheart Session

Name	Description	Type	Default	Valid Values	Importance
heartbeat.interval.ms	The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than <code>session.timeout.ms</code> , but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	int	3000		high
session.timeout.ms	The timeout used to detect consumer failures when using Kafka's group management facility. The consumer sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove this consumer from the group and initiate a rebalance. Note that the value must be in the allowable range as configured in the broker configuration by <code>group.min.session.timeout.ms</code> and <code>group.max.session.timeout.ms</code> .	int	10000		high
max.poll.interval.ms	The maximum delay between invocations of <code>poll()</code> when using consumer group management. This places an upper bound on the amount of time that the consumer can be idle before fetching more records. If <code>poll()</code> is not called before expiration of this timeout, then the consumer is considered failed and the group will rebalance in order to reassign the partitions to another member.	int	300000	[1,...]	medium

### 2.3.3 消费offset

【思考】消费者启动时从哪里开始消费消息？

## auto.offset.reset

如果zookeeper上没有消费者的offset，或保存的消费者offset被删除了，消费者启动时从哪里开始消费？

auto.offset.reset	largest	What to do when there is no initial offset in ZooKeeper or if an offset is out of range ;smallest : automatically reset the offset to the smallest offset; largest : automatically reset the offset to the largest offset;anything else: throw exception to the consumer

```
//设置创建的消费者从何处开始消费消息
props.put("auto.offset.reset", "smallest");
```

【思考】每次启动都从头开始消费吗？如何从上次结束的位置开始？

## 自动提交

```
// 开启自动消费offset提交
// 如果此值设置为true，consumer会周期性的把当前消费的offset值保存到zookeeper。当
consumer失败重启之后将会使用此值作为新开始消费的值。
props.put("enable.auto.commit", "true");
// 自动消费offset提交的间隔时间
props.put("auto.commit.interval.ms", "1000");
```

存在问题：

- 重复消费
- 丢数据（丢消息）

## 手动提交

### 代码示例

```
//设置手动提交
props.put("enable.auto.commit", "false");
...
//手动同步提交消费者offset到zookeeper
consumer.commitSync();
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Properties;
```

```

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

public class ManualCommitConsumerDemo {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("group.id", "test");
        //设置手动提交消费offset
        props.put("enable.auto.commit", "false");
        props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
        @SuppressWarnings("resource")
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("foo", "bar"));
        final int minBatchSize = 200;
        List<ConsumerRecord<String, String>> buffer = new ArrayList<>();
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);
            for (ConsumerRecord<String, String> record : records) {
                buffer.add(record);
            }
            if (buffer.size() >= minBatchSize) {
                insertIntoDb(buffer);
                //手动同步提交消费者offset到zookeeper
                consumer.commitSync();
                buffer.clear();
            }
        }

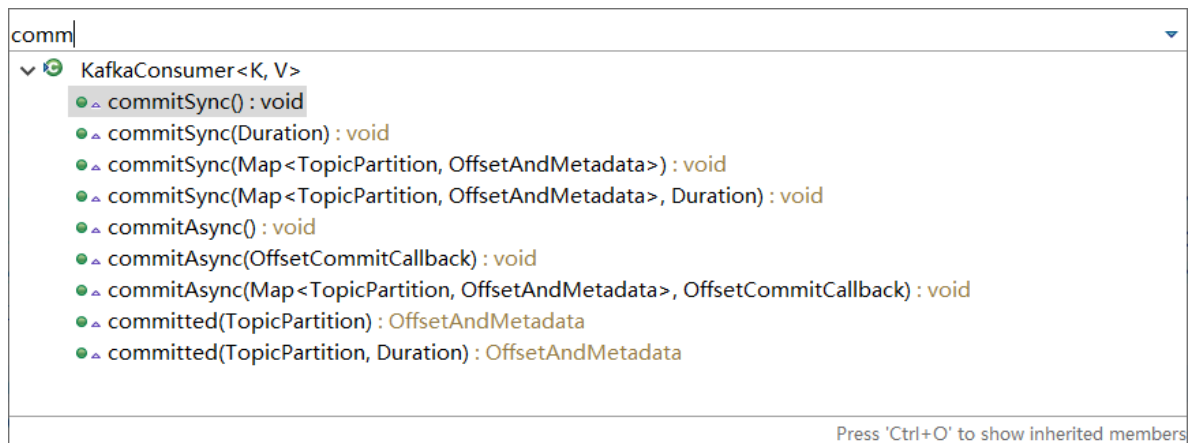
        private static void insertIntoDb(List<ConsumerRecord<String, String>>
buffer) {
            // Insert into db
        }
    }
}

```

## Consumer 的提交方法

有同步、异步的方法，还有获取上次提交的offset数据的方法





### 灵活按分片提交消费offset值

当你订阅了多个topic的消息，一次拉取可能会返回多个主题多个分片的消息集，上面的手动提交是所有的分片一起提交，如有需要，我们可以更细粒度地控制提交，下面的代码示例展示了按分片处理消息，没处理完一个则提交该分片的消费offset值。【注意】offset值是下次拉取的起始值（lastOffset + 1）

```
try {
    while(running) {
        ConsumerRecords<String, String> records =
consumer.poll(Long.MAX_VALUE);
        for (TopicPartition partition : records.partitions()) {
            List<ConsumerRecord<String, String>> partitionRecords =
records.records(partition);
            for (ConsumerRecord<String, String> record : partitionRecords)
            {
                System.out.println(record.offset() + ": " +
record.value());
            }
            long lastOffset = partitionRecords.get(partitionRecords.size()
- 1).offset();
            consumer.commitSync(Collections.singletonMap(partition, new
OffsetAndMetadata(lastOffset + 1)));
        }
    }
} finally {
    consumer.close();
}
```

### 控制消费的位置


有两种情况需要控制消费的位置：

- 消费者程序在某个时刻停止了运行，重启后继续消费从那个时刻以来的消息。
- 消费者在本地可能存储了消费的消息，但这份本地存储坏了，想重新取一份到本地。

控制消费位置的情况的下一般使用 指定消费的分片方式来进行消费

```
String topic = "foo";
TopicPartition partition0 = new TopicPartition(topic, 0);
TopicPartition partition1 = new TopicPartition(topic, 1);
consumer.assign(Arrays.asList(partition0, partition1));
```

---

▼  KafkaConsumer<K, V>

- ▲ seek(TopicPartition, long) : void
- ▲ seekToBeginning(Collection<TopicPartition>) : void
- ▲ seekToEnd(Collection<TopicPartition>) : void

### 2.3.4 subscribe vs assign

```
// 不是订阅Topic
// consumer.subscribe(Arrays.asList("test", "test-group"));
// 而是直接分配该消费者读取某些分片 subscribe 和 assign 只能用其一，assign 时不受
// rebalance影响。
TopicPartition partition = new TopicPartition("test", 0);
consumer.assign(Arrays.asList(partition));
```

- subscribe：该消费者消费的是哪些分片是由Kafka根据消费组动态指定，并可以动态rebalance
- assign：则是由用户指定要消费哪些分片，不受rebalance影响。

了解 ConsumerRebalanceListener

### 2.3.5 poll设置

【思考】一次拉取，返回多少数据？

可以通过下面的消费者参数来指定。

Name	Description	Type	Default	Valid Values	Importance
fetch.min.bytes	The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request. The default setting of 1 byte means that fetch requests are answered as soon as a single byte of data is available or the fetch request times out waiting for data to arrive. Setting this to something greater than 1 will cause the server to wait for larger amounts of data to accumulate which can improve server throughput a bit at the cost of some additional latency.	int	1	[0,...]	high
max.partition.fetch.bytes	The maximum amount of data per-partition the server will return. Records are fetched in batches by the consumer. If the first record batch in the first non-empty partition of the fetch is larger than this limit, the batch will still be returned to ensure that the consumer can make progress. The maximum record batch size accepted by the broker is defined via <code>message.max.bytes</code> (broker config) or <code>max.message.bytes</code> (topic config). See <code>fetch.max.bytes</code> for limiting the consumer request size.	int	1048576	[0,...]	high
fetch.max.bytes	The maximum amount of data the server should return for a fetch request. Records are fetched in batches by the consumer, and if the first record batch in the first non-empty partition of the fetch is larger than this value, the record batch will still be returned to ensure that the consumer can make progress. As such, this is not a absolute maximum. The maximum record batch size accepted by the broker is defined via <code>message.max.bytes</code> (broker config) or <code>max.message.bytes</code> (topic config). Note that the consumer performs multiple fetches in parallel.	int	52428800	[0,...]	medium
max.poll.records	The maximum number of records returned in a single call to <code>poll()</code> .	int	500	[1,...]	medium

### 2.3.6 消费的流速控制 Flow Control

当我们给消费者指定从多个分片取数据时，一次poll它会同时从所有指定的分片中拉取数据。但在有些情况下我们可能需要先全速消费指定分片的一个子集，当这个子集只有少量或没有数据时再开始消费其他的分片。

这样的场景如：

- 在流式处理中，程序要对两个Topic中的流数据执行join操作，而一个Topic的生产速度快与另一个，此时就需要降低快的topic的消费速度来匹配慢的。
- 另一个场景：启动消费者时，已经有大量消息堆积在这些指定的Topic中，程序需要优先处理包含最新数据的topic，再处理老旧数据的topic。

Kafka的Consumer支持通过 `pause(Collection)` 和 `resume(Collection)` 来动态控制消费流速。

- `pause(Collection partitions)` 暂停一个子集的拉取
- `resume(Collection partitions)` 恢复子集的拉取

下次调用 `poll(Duration)` 时它们生效。

### 2.3.7 事务

和生产者使用事务相关。重点就是隔离级别。

### 2.3.8 消费者的配置参数

<http://kafka.apachecn.org/documentation.html#newconsumerconfigs>

建议看英文最新版的。

### 2.3.9 Spring API 【了解】

- @KafkaListener
- @TopicPartition
- @PartitionOffset
- KafkaListenerErrorHandler

### 2.3.10 ConsumerInterceptor 【知道】

配置参数: interceptor.classes

## 3 Kafka Streams process

### 3.1 流式计算说明

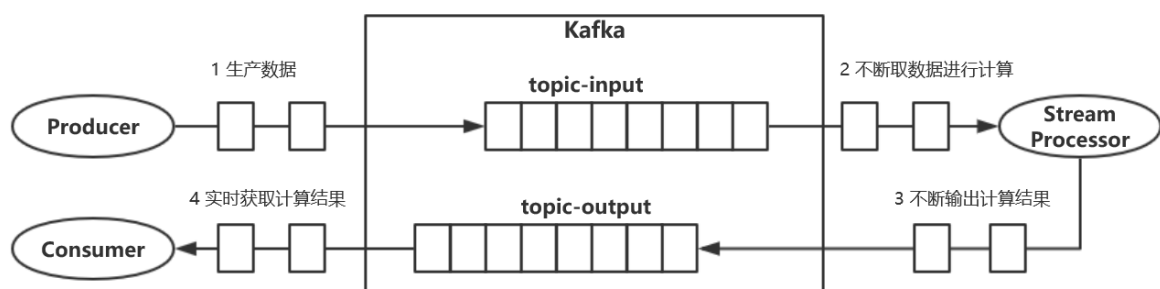
示例:

- 双十一时实时滚动的订单量、成交总金额。
- 每十分钟的成交额
- 股票交易看板

流式数据 --> 流式计算

流式计算的特点:

- 数据是随时间不断产生的，没有界限，数据是不能变更的。
- 计算也是不断进行的，是近实时的计算。
- 计算的结果是不断更新的，每次计算产生最新的结果



Kafka中提供了 **Kafka-Streams** 客户端库，让我们可以非常轻松地编写流式计算程序，来对Kafka集群中存储的流数据进行实时计算、分析处理。

#### Kafka-Streams的特点：

- 作为一个简单的轻量级客户端库设计，它可以很容易地嵌入到任何Java应用程序中。
- 除Apache Kafka本身作为内部消息层外，对系统没有外部依赖;值得注意的是，它使用Kafka的分片模型进行水平伸缩处理，同时保持了强大的顺序保证。
- 支持容错的本地状态，这支持非常快速和高效的有状态操作，比如窗口连接和聚合。
- 支持精确的一次处理语义，以确保每条记录将被处理一次，且仅被处理一次，即使流客户机或Kafka代理在处理过程中出现故障也是如此。
- 使用一次一个记录的处理来实现毫秒级的处理延迟，并支持基于事件时间的窗口操作和记录的延迟到达。
- 提供必要的流处理原语，以及高级流DSL和低级处理器API。

## 3.2 一个流式计算程序示例

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams</artifactId>
  <version>2.3.0</version>
</dependency>
```

【需求】编写一个统计单词在消息总出现次数统计的流计算程序

【思考】需要做哪些事情来完成单词统计。

- 不断消费消息
- 分割消息为单词
- 累计单词的出现次数
- 每隔一分钟输出统计结果

## 3.3 Kafka Streams Low-level processor API 和 核心概念

### 3.3.1 Processor 处理器

我们实现Processor接口提供我们的数据处理逻辑。**要掌握的知识点：**

1. 掌握Processor的三个方法的用途
2. 掌握ProcessorContext的用途
3. 重新认识Kafka中的数据为什么是Key、Value 对结构的。

### 3.3.2 Processor Topology 处理器拓扑结构

处理器拓扑结构，即流计算的流程。Kafka-streams API中提供了Topology这个API来让我们组合多个流处理步骤来构成一个复杂的流计算流程。看下面的Topology组合示例代码：

```

Topology topology = new Topology();

topology.addSource("SOURCE", "src-topic")
// add "PROCESS1" node which takes the source processor "SOURCE" as its upstream
processor
.addProcessor("PROCESS1", () -> new MyProcessor1(), "SOURCE")

// add "PROCESS2" node which takes "PROCESS1" as its upstream processor
.addProcessor("PROCESS2", () -> new MyProcessor2(), "PROCESS1")

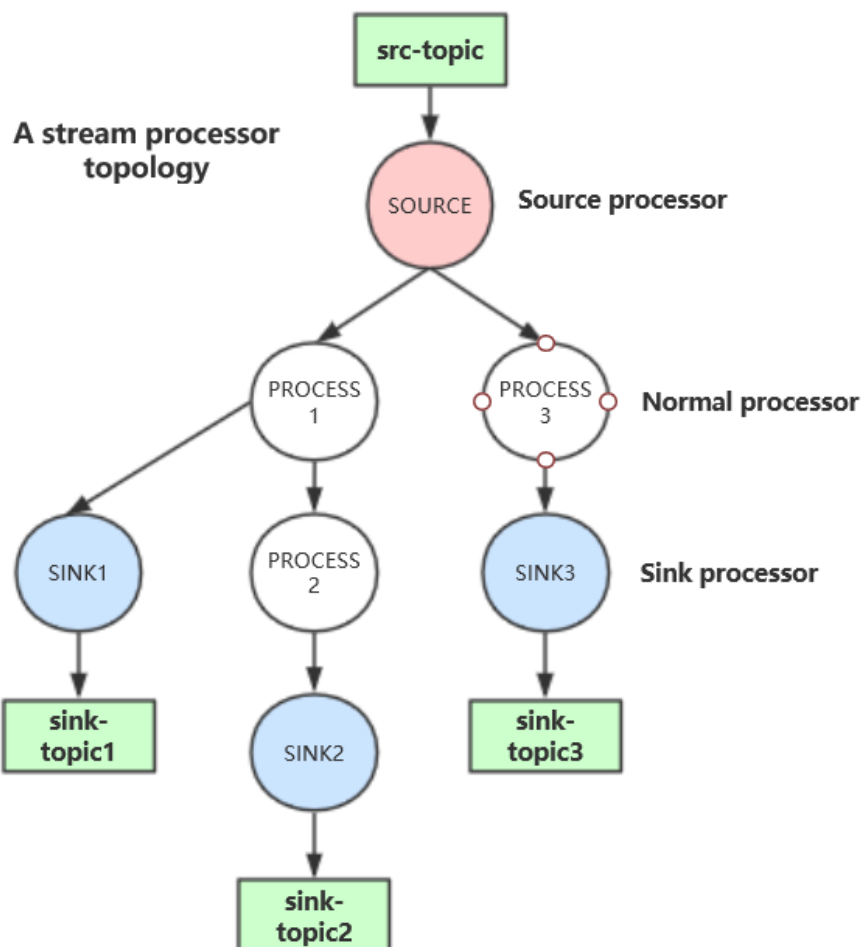
// add "PROCESS3" node which takes "PROCESS1" as its upstream processor
.addProcessor("PROCESS3", () -> new MyProcessor3(), "PROCESS1")

// add the sink processor node "SINK1" that takes Kafka topic "sink-topic1"
// as output and the "PROCESS1" node as its upstream processor
.addSink("SINK1", "sink-topic1", "PROCESS1")

// add the sink processor node "SINK2" that takes Kafka topic "sink-topic2"
// as output and the "PROCESS2" node as its upstream processor
.addSink("SINK2", "sink-topic2", "PROCESS2")

// add the sink processor node "SINK3" that takes Kafka topic "sink-topic3"
// as output and the "PROCESS3" node as its upstream processor
.addSink("SINK3", "sink-topic3", "PROCESS3");

```



拓扑结构中有两种特殊的处理器：

- **Source Processor** : Source Processor 是一种没有前置节点的特殊流处理器。它从一个或者多个 Kafka Topic 消费数据并产生一个输入流给到拓扑结构的后续处理节点。
- **Sink Processor** : sink processor 是一种特殊的流处理器，没有处理器需要依赖于它。它从前置流处理器接收数据并传输给指定的 Kafka Topic

### 3.3.3 State Store

【思考】流处理程序启动后能停吗？

重启时怎么恢复到停前的计算状态？

得要能保存计算过程中的状态（结算的结果，特别是中间环节的计算结果）

【思考】为了使流计算过程能容错，我们需要存储计算状态，那可以存储到哪里呢？

内存、磁盘、db

存储到本机可靠吗？

如果机器故障了，为了容错，需要能将计算迁移到其他机器上继续，存储到本机就不合适了。

那存到哪里合适？

Topic

```
KeyValueBytesStoreSupplier countStoreSupplier =  
Stores.inMemoryKeyValueStore("Counts");  
StoreBuilder<KeyValueStore<String, Long>> builder =  
Stores.keyValueStoreBuilder(countStoreSupplier, Serdes.String(), Serdes.Long());
```

```
// add the count store associated with the wordCountProcessor processor  
// 在topoly中关联Processor要使用的state store  
topology.addStateStore(builder, "Process")
```

```
import org.apache.kafka.streams.state.StoreBuilder;  
import org.apache.kafka.streams.state.Stores;  
  
StoreBuilder<KeyValueStore<String, Long>> countStoreSupplier =  
Stores.keyValueStoreBuilder(  
    Stores.persistentKeyValueStore("Counts"),  
    Serdes.String(),  
    Serdes.Long()  
).withLoggingDisabled(); // disable backing up the store to a changelog topic
```

```

import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;

// 存储 state changelog 的topic的配置
Map<String, String> changelogConfig = new HashMap();
// override min.insync.replicas
changelogConfig.put("min.insync.replicas", "1")

StoreBuilder<KeyValueStore<String, Long>> countStoreSupplier =
Stores.keyValueStoreBuilder(
    Stores.persistentKeyValueStore("Counts"),
    Serdes.String(),
    Serdes.Long())
    .withLoggingEnabled(changelogConfig); // enable changelogging, with custom
changelog settings

```

容错而记录变更日志 (state store 变更日志topic), 默认是开启的。

存储变更日志的topic名称为 `流计算应用名-存储名-changelog`, 如 `my-stream-processing-application-Counts-changelog`

## 3.4 DSL High-Level API

【思考】流计算一般都计算些什么结果? 或做些什么计算?

聚合计算 数据转换

kafka-streams 提供了一种更高级的简便DSL, 为我们定义好了很多聚合函数, 方便我们快速开发流计算程序。

详细的API

KStream KTable

<http://kafka.apache.org/23/documentation/streams/developer-guide/dsl-api.html>

## 4 Connect

<http://kafka.apachecn.org/documentation.html#connect>