1. 一台服务器可能同时是一个分区的leader，另一个分区的follower。 这样可以平衡负载，避免所有的请求都只让一台或者某几台服务器处理。
2. Topic分区中消息只能由消费者组中的唯一一个消费者处理，所以消息肯定是按照先后顺序进行处理的。但是它也仅仅是保证Topic的一个分区顺序处理，不能保证跨分区的消息先后处理顺序。 所以，如果你想要顺序的处理Topic的所有消息，那就只提供一个分区。
3. kafka中消费者组有两个概念：队列：消费者组（consumer group）允许同名的消费者组成员瓜分处理。发布订阅：允许你广播消息给多个消费者组（不同名）。
4. 如果消费者过多，出现了消费者的个数大于分区个数的情况，就会有消费者分配不到任何分区

初识Kafka

多分区、多副本，基于ZK协调的分布式消息系统

* 基本概念

LEO(Log End Offset)：标识当前日志文件中下一条待写入消息的offset，相当于当前日志分区中最后一条消息的offset值加1。

HW(High Watermark)：标识了一个特定的消息偏移量，消费者只能拉取到这个offset之前的消息，分区ISR集合中最小的LEO即为该分区的HW(本质是多个副本的同步效率可能不一样导致的每个副本的LEO在同一时刻可能不同，因此就取最小的LEO避免数据不一致)

* 服务器端参数配置

listeners：

该参数指明broker监听客户端连接的地址列表，即为客户端要连接broker的入口地址列表。advertised.listeners作用类似

生产者

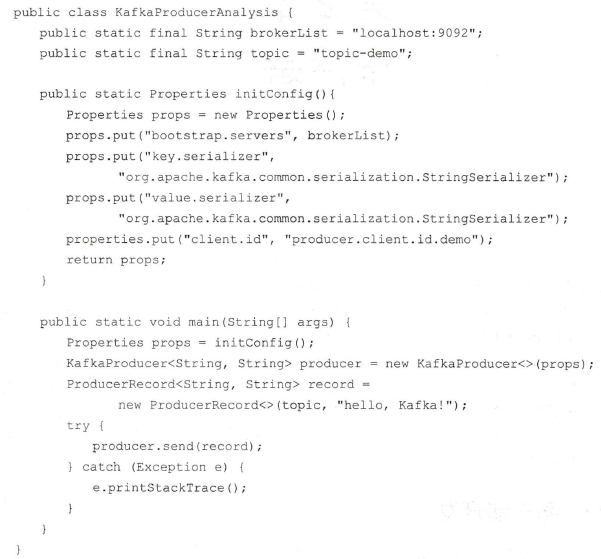
* 客户端开发

步骤：1.配置生产者客户端以及创建相应的生产者实例

2.构建待发送的消息

3.发送消息

4.关闭生产者实例



* 必要的配置参数：

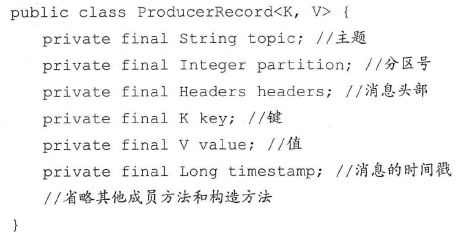
1. bootstrap.servers：并非需要所有的地址都写上，因为生产者会从给定的broker里找到其他broker，不过建议至少设置2个以上防止宕机的情况。
2. key.serializer和value.serializer序列化，broker端接收的消息必须以字节数组(byte[])存在

更多参数可在ProducerConfig里查看

KafkaProducer是线程安全的，可以在多个线程中共享单个KafkaProducer实例

* 消息的发送

构建待发送的消息对象是ProducerRecord



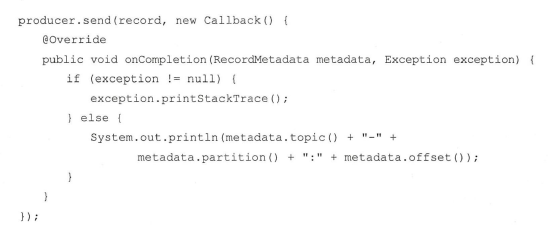
其中topic和value是必填项

发送消息3种方式：1.发后即忘 producer.send() 效率最高，可靠性差

2.同步 producer.send().get() 效率最低，可靠性高

返回的对象是RecordMetadata，包含了分区中的偏移量和时间戳等

3.异步



kafka有响应时就会回调，要么成功要么抛出异常，即onCompletion中metadata和exception两者是互斥的

消息重试配置：ProducerConfig. RETRIES\_CONFIG

* 序列化

生产者需要用序列化器Serializer把对象转换成字节数组才能通过网络发送给Kafka，注意 生产者使用的序列化器要与消费者使用的发序列化器相对应，否则无法解析出来

* 分区器

需实现Partitioner接口

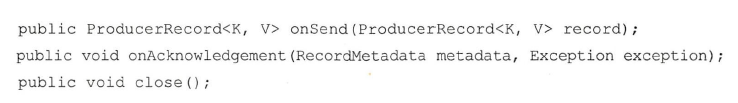
为发送的消息分配分区。如果ProducrRecord中指定了partition字段，那么就不需要分区器的作用

默认分区器：key为null采用轮询方式(且仅分配给可用分区)，key不为null采用哈希方式(所有分区，包含不可用分区)

* 拦截器

实现ProducerInterceptor接口

生产者拦截器既可以用来在消息发送前做一些准备工作， 比如按照某个规则过滤不符合要求的消息、修改消息的内容等，也可以用来在发送回调逻辑前做一些定制化的需求，比如统计类工作。



onSend：消息序列化和计算分区之前

onAcknowledgement：消息被应答之前或失败时被调用，优先于callback

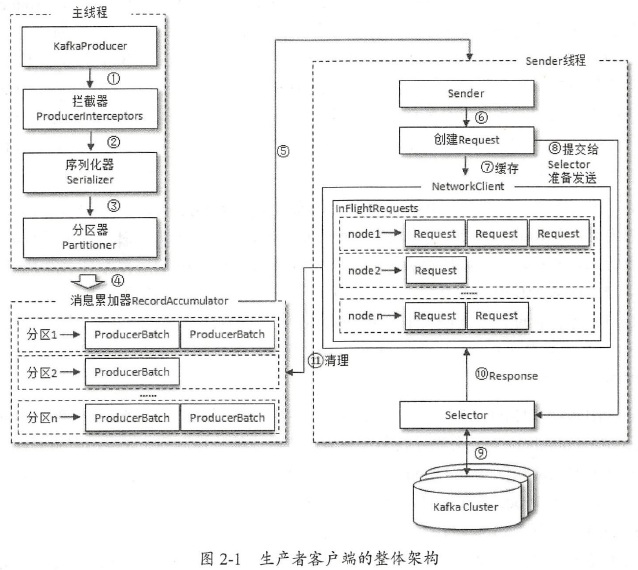
close:用于在关闭拦截器之前执行一些资源的清理工作

以上三个方法抛出的异常均会被捕获到日志中，但不会在向上传递

KafkaProducer中不仅可以指定一个拦截器，还可以指定多个拦截器已形成拦截链，按照拦截器配置的顺序来执行。

注意：拦截器链中某个拦截器由于异常而执行失败，那么下一个拦截器会接着从上一个执行成功的拦截器继续执行

* 原理分析



▲元数据

元数据是指Kafka 集群的元数据，这些元数据具体记录了集群中有哪些主题，这些主题有哪些分区，每个分区的lead巳r 副本分配在哪个节点上， follower 副本分配在哪些节点上，哪些副本在AR 、ISR 等集合中，集群中有哪些节点，控制器节点又是哪一个等信息。

消费者

▲消费者组

1.如果消费者都隶属于同一个消费者组，那么所有的消息都会被均衡地投递给每一个消费者，即每条消息只会被一个消费者消费，相当于点对点模式

2.如果所有的消费者都隶属于不同的消费组，那么所有的消息都会被广播给所有的消费者，即每条消息会被所有的消费者处理，这就相当于发布／订阅模式的应用

消费者组只是一个逻辑上的概念，消费者在消费前需要指定其所属的消费者组名称，消费者并非逻辑上的概念，它是实际的应用实例，可以是一个线程，也可以是一个进程，既可以在同一台机器上，也可以在不同机器上

客户端开发

* 客户端开发

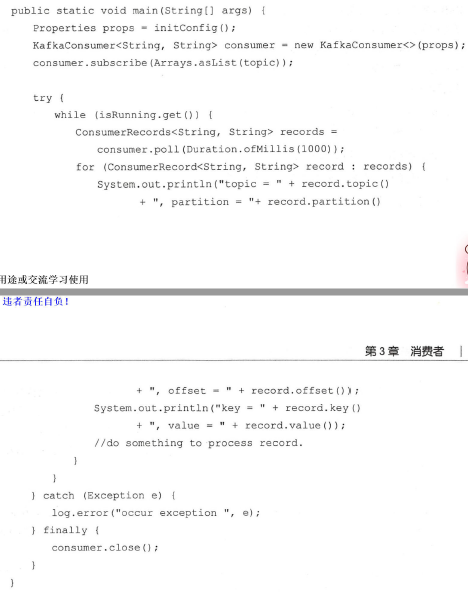
步骤：1.配置消费者客户端参数以及创建相应的消费者实例

2.订阅主题

3.拉取消息并消费

4.提交消费位移

5.关闭消费者实例



* 消费者客户端

旧消费者客户端是用Scala编写的

新消费者客户端是用java编写的

必要的参数：

bootstrap.servers:可以指定多个连接地址，但其实只要指定一个，消费者便会从现有的配置中查找到全部的Kafka集群成员

group.id：消费者组名称，一般设置为跟业务名称挂钩

key .deserializer、value.deserializer：消费者从broker端获取的消息格式都是字节数组byte[]类型，需要通过反序列化操作才能还原成原有的对象格式

所有的配置参数可参考ConsumerConfig这个类

* 订阅主题与分区

如果前后两次订阅了不同的主题(即执行了2次subscribe)，那么消费者以最后一次的为准

例如：

最终只消费topic2

三种订阅方式:

1. subscribe(Collection)订阅主题集合
2. subscribe(pattern)订阅主题正则匹配
3. assign(Collection<TopicPartition>)订阅指定分区
   1. TopicPartition只有2个属性：topic和partition
   2. 分区的信息PartitionInfo可以通过consumer.partitionsFor(topic)获取

其中除了assign订阅方式其他方式都具有消费者自动再均衡的功能，即当消费组内的消费者增加或减少时，分区分配关系会自动调整，以实现消费负载均衡及故障自动转移。

取消订阅可以用unsubscribe()方法

* 反序列化

需要实现Deserializer这个接口

其中具体的反序列化过程在

deserialize(String topic, byte [] data)这个方法中实现

要注意上游的序列化和下游的反序列化要一致

* 消息消费

kafka中的消费是基于拉模式的

调用poll()方法

public ConsumerRecords<K, V> poll(final Duration timeout)

poll方法类似命令行kafka-console.consumer.sh 就那样阻塞在窗口中等待消息的到来，返回的是还没有被消费过的消息集。timeout则是这个阻塞要持续多久。

timeout 的设置取决于应用程序对响应速度的要求，比如需要在多长时间内将控制权移交给执行轮询的应用线程。可以直接将timeout 设置为0 ， 这样poll （）方法会立刻返回，而不管是否己经拉取到了消息。如果应用线程唯一的工作就是从Kafka 中拉取并消费消息，则可以将这个参数设置为最大值Lon g.MAX\_VALUE 。

消费者消费到的每条消息类型为ConsumerRecord

可以在拉取到消息后再指定topic或者分区

public Iterable<ConsumerRecord<K , V> records(String topic)

public List<ConsumerRecord<K, V >records(TopicPartition partition)

* 位移提交

消息在分区中的位置称为offset(偏移量)

旧消费者客户端中，消费位移是存储在Zookeeper中的，而在新的消费者客户端中，消费位移存储在kafka内部的主题\_\_consumer\_offsets中。把将消费位移存储起来(持久化)的动作称为”提交”，消费者在消费完消息之后需要执行消费位移的提交

注意消费者提交的位移是下一条需要拉取的消息的位置

log-end-offset 总的记录数

位移提交的时机不当可能会造成重复消费和消息丢失的现象。

消息丢失：拉取到消息后就进行位移提交，那么当消费消息的时候出现了异常，下一次重新拉取的时候就会丢失掉出现消费异常的消息及其之后消息。

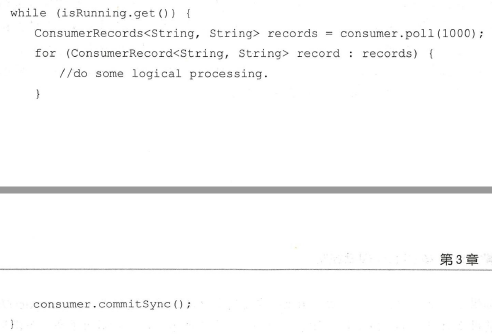
重复消费：消费完在提交消息，如果消费过程中出现异常，导致没有提交位移，那么下一次还是从之前提交的位移处拉取消息，那么出现消费异常的消费及其之前的消息就又重新消费了一遍

kafak默认的提交方式是自动提交，有客户端参数enable.auto.commit配置，默认值为true。默认提交是定期提交，由auto.commit.interval.ms配置，默认5秒

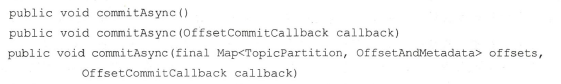
手动提交：

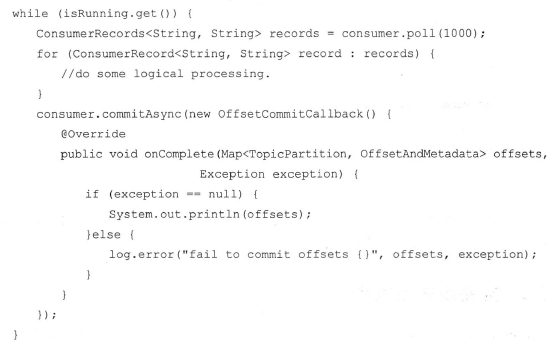
首先将enable.auto.commit设置为false。

commitSync() 同步提交



commitAsync() 异步提交





位移提交失败的情况，可采取重试，重试时为了避免重复消费可引入版本号机制。

* 控制或关闭消费

在有些应用场景下我们可能需要暂停某些分区的消费而先消费其他分区，当达到一定条件时再恢复这些分区的消费。

pause()暂停某些分区在拉取操作时返回数据给客户端的操作

resume()恢复某些分区向客户端返回数据的操作

wakeup()是KafkaConsumer中唯一可以从其他线程里安全调用的方法(KafkaConsumer 是非线程安全的)。可以退出poll()的逻辑，并抛出WakeupException异常，这个异常不需要处理，只是一种跳出循环的方式

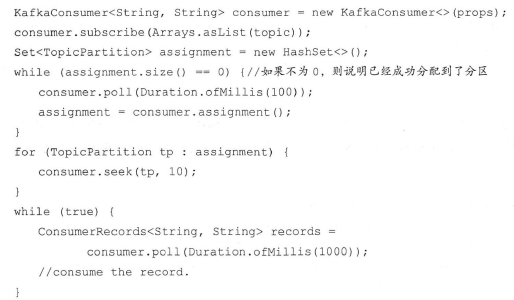
* 指定位移消费

在Kafka中每当消费者查找不到所处记录的消费位移时，就会根据消费者客户端参数auto.offset.reset的配置来决定从何处开始进行消费，默认值”latest”(下一条要写入消息的位置)，如果设置为”earliest”则会从起始处也就是0开始消费。

位移越界也会触发auto.offset.reset参数的执行

指定位移消费：seek()

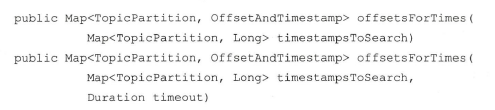
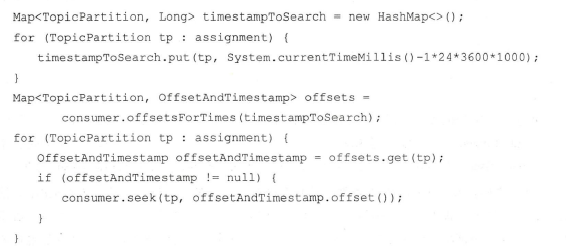
seek()方法只能重置消费者分配到的分区的消费位置，而分区的分配是在poll()方法的调用过程中实现的。也就是说，在执行seek()方法之前需要先执行一次poll()方法，等到分配到分区之后才可以重置消费位置。



从分区开头开始消费seekToBeginning()

从分区末尾开始消费seekToEnd()

从某个时间点消费 offsetsForTimes()

消费位移可以保存在任意的存储介质中，下次消费时可以读取消费位移并结合seek()方法来进行相应的操作

* 再均衡

再均衡是指分区的所属权从一个消费者转移到另一消费者的行为，它为消费组具备高可用性和伸缩性提供保障，使我们可以既方便又安全地删除消费组内的消费者或往消费组内添加消费者。

再均衡期间，消费者组会变得不可用且消费者当前的状态也会丢失(可能造成重复消费)

* 再均衡监听器 ConsumerRebalanceListener

void onPartitionsRevoked(Collection<TopicPartition> partitions)

再均衡开始之前和消费者停止读取消息之后被调用。可以通过该方法来处理消费位移的提交。partitions 再均衡前所分配到的分区

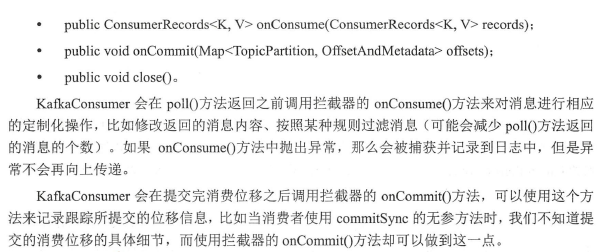
void onPartitionsAssigned(Collection<TopicPartition> partitions)

重新分配分区之后和消费者开始读取消息之前被调用partitions 再均衡后所分配到的分区

* 消费者拦截器

主要在消费到消息或者再提交消费位移时进行一些定制化的操作

实现ConsumerInterceptor接口



通过interceptor.classes配置参数指定

拦截链同生产者

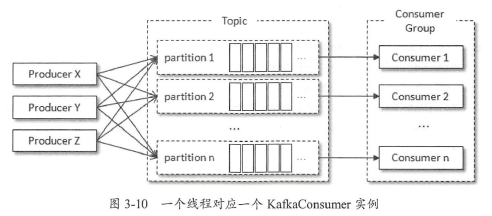
* 多线程实现

KafkaProducer是线程安全的，KafkaConsumer是非线程安全的。

KafkaConsumer中定义了一个acquire()用来检测当前是否只有一个线程在操作，否则抛出异常

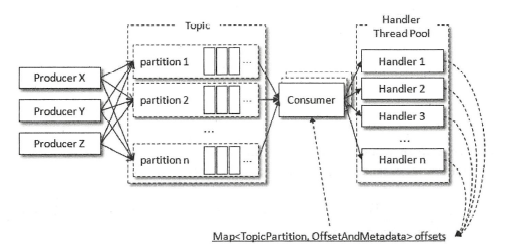


1. 为每个线程实例化一个KafkaConsumer对象

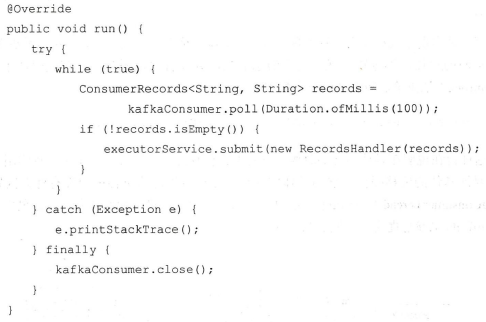


线程数设置为不大于分区数

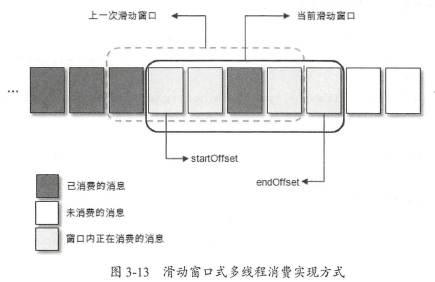
1. 引入线程池



将拉取后的消息提交给线程池中的线程，然后并发处理这些消息，同时维护一个Map来存储各个线程中offset的情况。



1. 滑动窗口式的实现方式



一个方格对应一个消费线程，当startOffset所指的方格中的对应消息被标记消费完成，整个滑动窗口就往前前进一格

如果startOffset指向的消息处理有异常导致无法被标记成消费完成，可以设定一个阈值，对该部分消息进行重试消费，开2个队列 重试队列和死信队列(重试后依旧失败的消息)，最后再单独对这一部分消息进行分析

* 重要的消费者参数

fetch.min.bytes:

该参数用来配置Consumer 在一次拉取请求（调用poll（）方法）中能从Kafka 中拉取的最小数据量，默认值为1 (B）

fetch.max.bytes:

该参数与fetch . max . bytes 参数对应，它用来配置Consumer 在一次拉取请求中从Kafka中拉取的最大数据量，默认值50MB 。

fetch.max.wait.ms:

拉取消息时阻塞等待的时间

主题与分区

主题的管理

* 创建主题

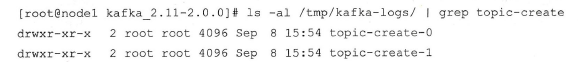
broker端配置参数auto.create.topics.enable(默认值为true)，

生产者或消费者向一个尚未创建的主题进行发送或消费时,都会自动创建一个相应的主题，建议设置为false。

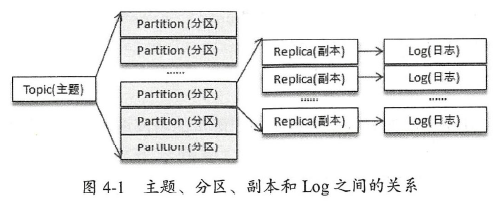
创建主题脚本：kafka-topics.sh



执行完脚本后，会在log.dir或log.dirs参数所配置的目录下创建相应的主题分区。



文件的命名方式为<topic>-<partition>



主题、分区、副本和Log （日志）的关系如图4-1 所示，主题和分区都是提供给上层用户的抽象， 而在副本层面或更加确切地说是Log 层面才有实际物理上的存在。

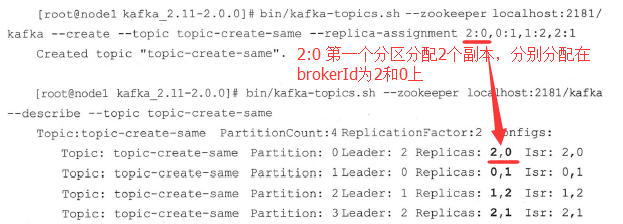
在zookeeper中对应的目录/brokers/topics/对应的主题名：



示例数据中的”2”：［ 1 , 2 ］表示分区2 分配了2 个副本，分别在brokerld 为1 和2 的broker

节点中。

--replica-assignment手动指定分区副本的分配方案：



注：同一个分区内的副本不能有重复，分区之间所指定的副本数也必须相同。

--config 设置创建主题的相关参数



对应zookeeper /config/topics/主题名

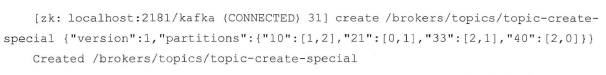
注：kafka内部做埋点时会根据主题名来命名metrics的名称，并且会将“.”改成“\_”。

* 分区副本的分配

默认都是从编号为0的分区开始一次轮询分配，但是副本分配的位置是随机的

创建主题时无论通过kafka-topics.sh脚本还是其他方式，本质都是在zookeeper中的/brokers/topics节点下创建与该主题对应的子节点并写入分区副本分配方案。

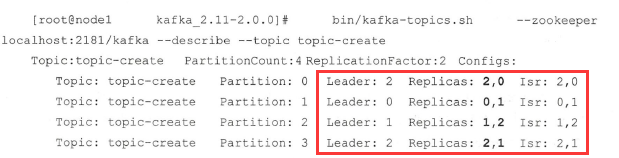
因此可以直接使用zookeeper创建对应的主题



如上，kafka-topics.sh创建副本时默认是从分区0开始创建，但使用zookeper就可以没有这个限制

* 查看主题

describe：



红框中数字代表的都是brokerId

--topics-with-overrides 参数可以找出所有包含覆盖配置的主题

--under-replicated-partitions 参数可以找出所有包含失效副本的分区

--unavailable-partitions 参数可以查看主题中没有leader副本的分区

* 修改主题

--alter

注:当主题中的消息包含key时, 在进行增加分区数操作时根据key计算分区的行为就会受到影响()。

kafka只支持增加分区数而不支持减少分区数

设置配置本质上是覆盖原本的默认配置，删除配置也即恢复成原来的默认配置。

* 配置管理

kafka-configs.sh

entity-type指定操作配置的类型，entity-name指定操作配置的名称



add-config 参数用来实现配置的增、改，即覆盖原有的配置； delete-config 参数用来实现配置的删， 即删除被覆盖的配置以恢复默认值。



zookeeper中存在的节点命名形式为/config/<entity-type>/<entity-name>

* 主题端参数

与主题相关的所有配置参数在broker层面都有对应参数，如果没有修改过主题的任何配置参数，那么就会使用broker端的对应参数(server.properties)作为其默认值

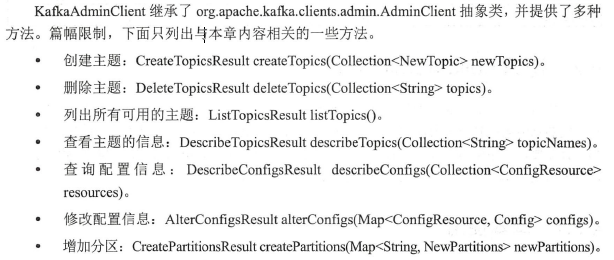
* 删除主题

必须将delete.topic.enable参数配置为true才能删除主题，该参数默认值就是true

使用kafka-topics .sh 脚本删除主题的行为本质上只是在ZooKeeper 中的／admin /delete \_topics 路径下创建一个与待删除主题同名的节点，以此标记该主题为待删除的状态。与创建主题相同的是，真正删除主题的动作也是由Kafka 的控制器负责完成的。

KafkaAdminClient

* 基本使用



* 主题合法性验证

Kafka broker端参数：create.topic.policy.class.name，默认值为null，它提供了一个入口用来验证主题创建的合法性。只需要自定义实现org.apache.kafka.policy.CreateTopicPolicy接口。

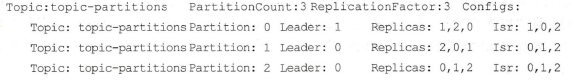
分区的管理

* 优先副本的选举(针对leader的)

分区使用多副本机制来提升可靠性，但只有leader副本对外提供读写服务，而follower副本只负责在内部进行消息的同步。如果一个分区的副本变得不可用，那么就意味着整个分区变得不可用。也就是说broker节点中leader副本个数的多少决定了这个节点负载的高低。

优先副本就是该分区的leader副本，也可以称之为preferred leader。

例如：



分区0和分区2中leader都是其replicas中的第一个，所以都是优先副本

优先副本的选举是指通过一定的方式促使优先副本选举为leader副本，即”分区平衡”，分区平衡并不意味着kafka集群的负载均衡

kafka中可以提供分区自动平衡的功能，于此对应的broker端参数是auto.leader.rebalance.enale默认值为true，不过在生产环境中不建议开启这个参数,可能会引起负面的性能问题。

* 手动操作：

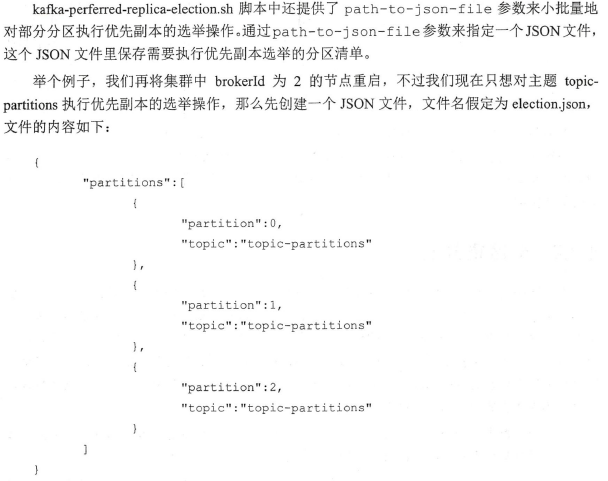
kafka-preferred-replia-election.sh脚本提供了对分区leader副本进行重新平衡的功能

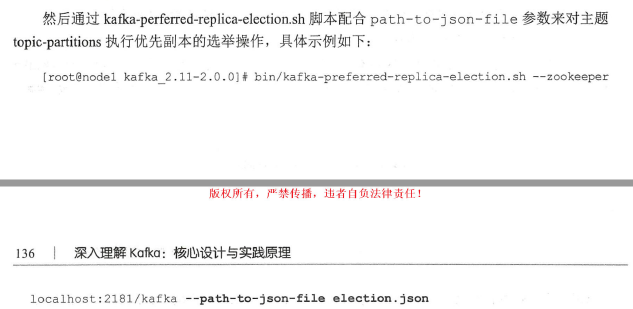


该方法会将集群上所有的分区都执行一遍优先副本的选举操作，leader副本转移是一项高成本的工作，如果要执行的分区数很多，那么会对客户端造成一定影响。

在优先副本的选举过程中，具体的元数据信息会被存入ZooKeeper的／admin/preferred\_replica\_election 节点，如果这些数据超过了ZooKeeper 节点所允许的大小，那么选举就会失败。默认情况下ZooKeeper所允许的节点数据大小为lMB 。

更好的方法是指定需要执行的分区清单：



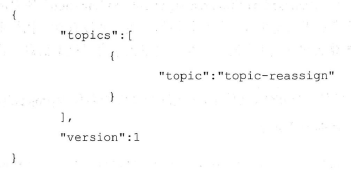


* 分区重分配（针对partition的）

当集群中新增broker 节点时，只有新创建的主题分区才有可能被分配到这个节点上，而之前的主题分区并不会自动分配到新加入的节点中，因为在它们被创建时还没有这个新节点，这样新节点的负载和原先节点的负载之间严重不均衡。

kafka-reassign-partition.sh使用步骤：

1. 创建一个包含主题清单的json文件(reassign.json)



1. 根据主题清单和broker节点生成一份重新分配方案

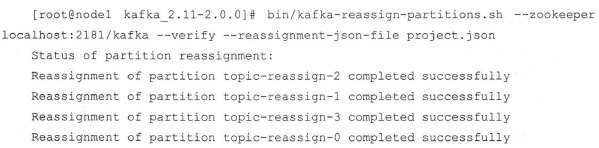


执行完后打印出两个json内容：current partition replica assignment是当前的，可以用来回滚。proposed partition reassignment configuration是重新分配后的方案(此时还没真正执行)，将其保存在一个json文件(project.json)里。

1. 最后根据这份方案执行具体的重分配动作



4.查看分区重分配的进度



* 原理：

分区重分配的基本原理是先通过控制器为每个分区添加新副本（增加副本因子）， 新的副本将从分区的leader副本那里复制所有的数据。根据分区的大小不同， 复制过程可能需要花一些时间， 因为数据是通过网络复制到新副本上的。在复制完成之后，控制器将旧副本从副本清单里移除（恢复为原先的副本因子数）。注意在重分配的过程中要确保有足够的空间。

如果要将某个broker 下线，那么在执行分区重分配动作之前最好先关闭或重启broker。这样这个broker 就不再是任何分区的leader 节点了，它的分区就可以被分配给集群中的其他broker 。这样可以减少broker 间的流量复制，以此提升重分配的性能，以及减少对集群的影响。

* 复制限流

对副本间的复制流量加以限制来保证重分配期间整体服务不会受太大的影响

两种方式：1.kafka-config.sh 2.kafka-reassign-partition.sh

两个与限流相关的参数：

1.leader.relication.throttled.rate：设置leader副本传输的速度 B/s

2.follower.replication.throttled.rate：设置副本复制的速度 B/s

通常情况下两者值是相同的，可在broker级别或者主题级别进行配置

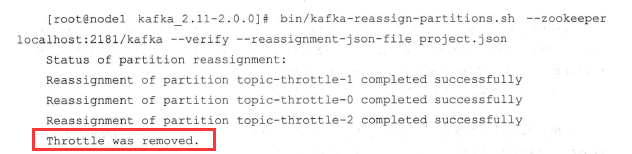
1. leader.replication.throttled.replicas
2. follower.replication.throttled.replicas

用来分别配置被限制速度的主题所对应的leader副本和follow副本列表，该方法比较烦琐容易出错

kafka-reassign-partition.sh 执行execute时可以直接指定 --throttle



为了不影响Kafka本身的性能，对临时设置的—throttle在使用完后要即使删除，可通过kafka-reassign-partition.sh 执行—verify



注：需要周期性的执行查看进度的命令直到重分配完成，这样可以确保限流设置被移除。

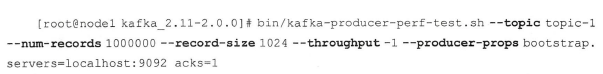
* 修改副本因子

使用kafka-reassign-partition.sh脚本 自己手动配置reassign.json文件 然后执行execute

如何选择合适的分区数

* 性能测试工具

生产者性能测试：kafka-producer-perf-test.sh



num-records用来指定发送消息的总条数

record-size用来设置每条消息的字节数

producer-props用来指定生产者的配置

throughput用来进行限流控制，当设定的值小于0时不限流，当设定的值大于0时，当发送的吞吐量大于该值时就会被阻塞一段时间，该吞吐量是按照MB/sec来算的

消费者性能测试：kafka-consumer-perf-test.sh



* 分区数越多吞吐量

分区数越多吞吐量越高，但是有一个临界值，超过该临界值就不一样了

* 分区数的上限

分区数的增加有一个上限，可通过linux命令 ulimit -n查看

查看当前kafka所占用的fd





当超过当前系统配置的fd后kafka就会崩溃，可在logs/server.log查看日志报”too many files”的错误

可通过修改/etc/security/limits.conf文件设置fd的大小

* 考量因素

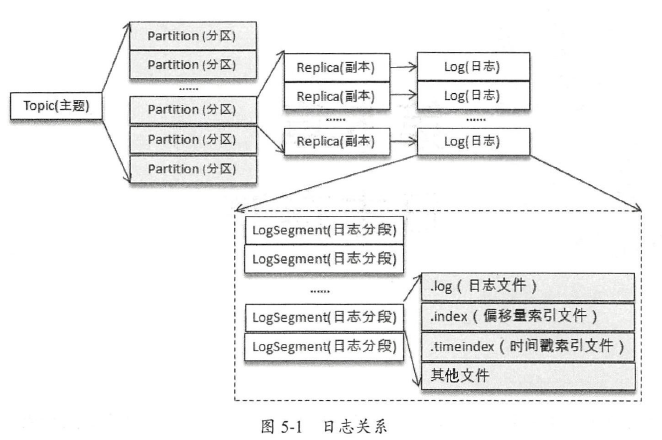
增加分区数时需要考虑的因素，不能盲目增加

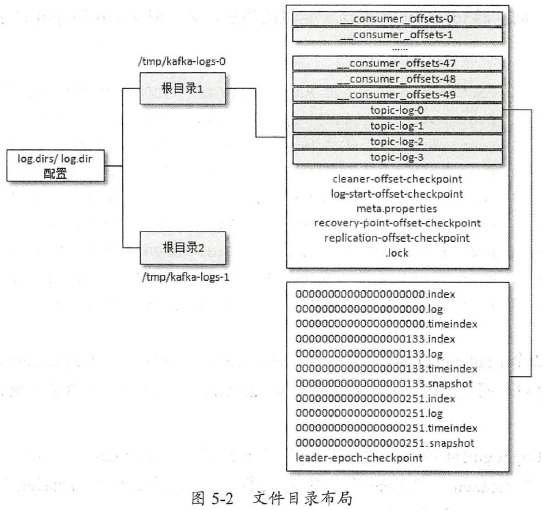
1. 基于key计算的主题
2. 要求消息保证顺序性的主题
3. 当前系统kafka已使用的fd个数
4. 系统可用性要求(分区数越多需要进行的同步操作，leader切换等机制也会相应的更加耗时)

分区数可以设为broker节点数的倍数，该准则不适用于大几十，上百以上的broker节点数

日志存储

* 文件目录布局

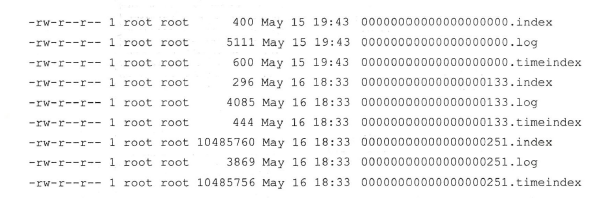




log文件命名<topic>-<partition>

具体的消息存储在.log文件中

每个logSegment都有对应的两个索引文件：偏移量索引文件(.index为后缀)和时间戳索引文件(.timeindex为后缀)，每个LogSegment都有一个基准偏移量baseOffset，用来表示当前logSegment中的第一条消息的offset。



* 日志格式的演变

▲ v0版本

0.10.0之前的版本

▲ v1版本

0.10.0到0.11.0之前版本 比起v0就多了一个timestamp字段

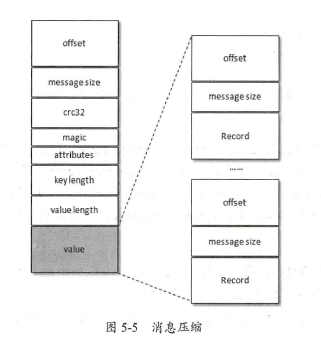
▲ v2版本

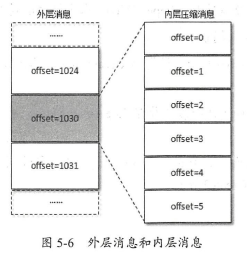
0.11.0开始为v2版本 消息集称为Record Batch 而不是先前的Message Set

* 消息压缩

Kafka实现的压缩方式是将多条消息一起进行压缩

消息压缩是将整个消息集进行压缩作为内层消息，内层消息整体作为外层的value





* 日志索引

日志分段文件对应了两个索引文件(.index和.timestamp)，主要用来提高查找消息的效率。

.log文件是日志分段文件

偏移量索引文件用来建立消息偏移量（ offset ）到物理地址之间的映射关系，方便快速定位消息所在的物理文件位置；时间戳索引文件则根据指定的时间戳（ timestamp ）来查找对应的偏移量信息。

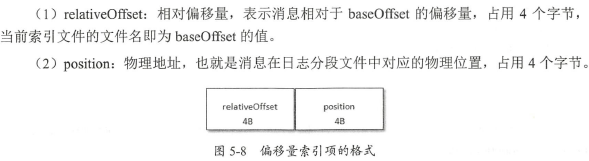
Kafka中的索引文件以稀疏索引的方式构造消息的索引，即不保证每个消息在索引文件都有对应的索引项。每当写入一定量的消息后，才会分别增加对应的偏移量索引项和时间戳索引项(broker端参数log.index.interval.bytes指定)

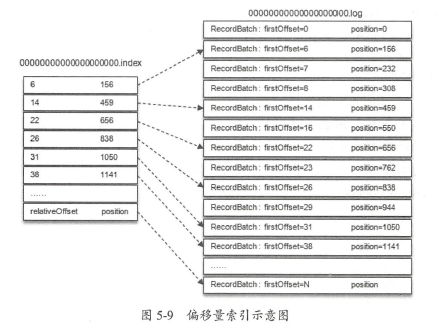
▲ 查找方式

1.偏移量索引：

首先快速定位日志分段文件：采用跳跃表，即把各个日志分段的baseOffset(偏移量索引文件名)作为Key存入一个map，进而来快速定位。

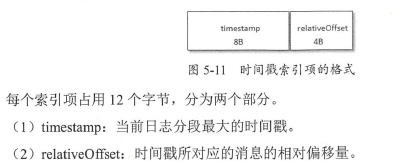
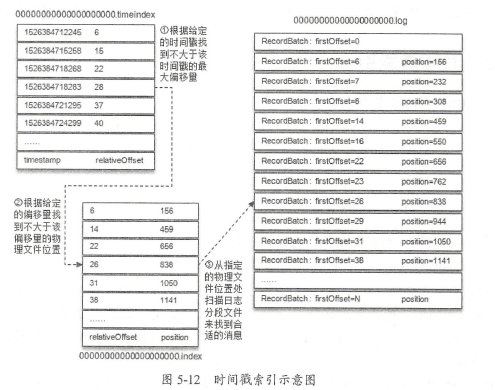
然后在定位到的索引文件中采用二分法查找不大于指定偏移量(已转换成相对偏移量了)的最大偏移量，然后再根据position(日志分段文件物理位置)去日志分段文件查找对应的消息。





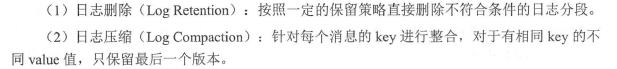
2.时间戳索引：没有使用跳跃表来定位日志分段，而是将指定的时间戳先跟各个日志分段的最大时间戳进行对比，直到找到不小于指定时间戳的largestTimeStamp。

然后在时间戳索引文件中使用二分法查找不大于指定时间戳的最大索引项，拿到后再去偏移量索引文件查找，接下来就跟偏移量索引文件查找方式一样了。

* 日志清理

两种日志清理策略：



log.cleanup.policy=”delete”

log.cleanup.policy=”compact”还可以同时支持两种即”delete,compact”

1. 日志删除

在kafka的日志管理器中会有一个专门的日志删除任务来周期性地检测和删除不符合保留条件的日志分段文件。

1. 基于时间：

log . retention . hours 、log.retention.minutes 和log.retention . ms

默认7天

1. 基于日志大小：

默认无穷大

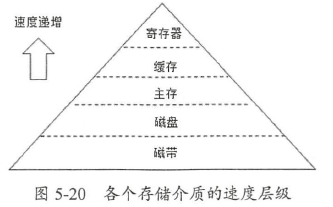
1. 基于日志起始偏移量

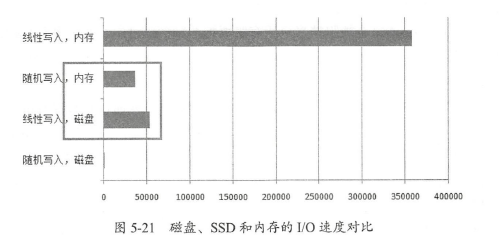
2. 日志压缩

这里的压缩指的是在默认的日志删除规则之提供一种清理过时数据的方式，即对于有相同key的不同value值，只保留最后一个版本。如果应用只关心key对应的最新value的值，则可以开启Kafka的日志清理功能，Kafka会定期将相同的key消息合并，只保留最新的value值。

注意区别：这里指的是日志压缩而不是消息压缩(消息可以采用gzip、Snappy、LZ4等压缩方式进行压缩)

* 磁盘存储





Kafka在设计时采用了文件追加的方式来写入消息，即只能在日志文件的尾部追加新的消息，并且也不允许修改已写入的消息，属于典型的顺序写盘操作。

1. 页缓存

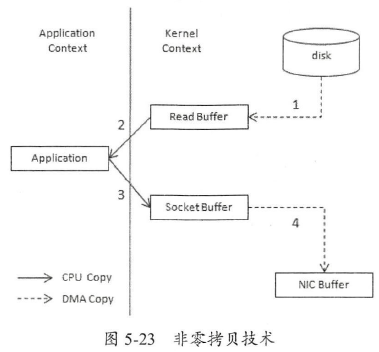
把磁盘中的数据缓存到内存中，把对磁盘的访问变为对内存的访问。

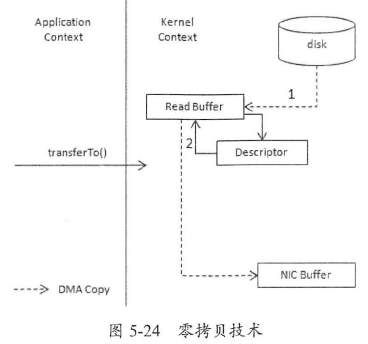
被修改过的也称为脏页，操作系统会在合适的时间把脏页中的数据写入磁盘，以保持数据的一致性。

Kafka中大量使用了页缓存，这是Kafka实现高吞吐的重要因素之一

1. 零拷贝

Kafka使用零拷贝技术将数据直接从磁盘文件复制到网卡设备中，而不需要经由应用程序之手，减少了内核和用户模式之间的上下文切换





深入服务端