今天和大家聊一下，kafka对于消息的可靠性保证。作为消息引擎组件，保证消息不丢失，是非常重要的。

那么kafka是如何保证消息不丢失的呢？

**前提条件**

任何消息组件不丢数据都是在特定场景下一定条件的，kafka要保证消息不丢，有两个核心条件。

第一，必须是已提交的消息，即committed message。kafka对于committed message的定义是，生产者提交消息到broker，并等到多个broker确认并返回给生产者已提交的确认信息。而这多个broker是由我们自己来定义的，可以选择只要有一个broker成功保存该消息就算是已提交，也可以是令所有broker都成功保存该消息才算是已提交。不论哪种情况，kafka只对已提交的消息做持久化保证。

第二，也就是最基本的条件，虽然kafka集群是分布式的，但也必须保证有足够broker正常工作，才能对消息做持久化做保证。也就是说 kafka不丢消息是有前提条件的，假如你的消息保存在 N 个kafka broker上，那么这个前提条件就是这 N 个broker中至少有 1 个存活。只要这个条件成立，kafka就能保证你的这条消息永远不会丢失。

**如何保证消息不丢**

一条消息从产生，到发送到kafka保存，到被取出消费，会有多个场景和流程阶段，可能会出现丢失情况，我们聊一下kafka通过哪些手段来保障消息不丢。

**生产端**

Producer端可能会丢失消息。目前Kafka Producer是异步发送消息的，也就是说如果你调用的是producer.send(msg)这个API，那么它通常会立即返回，但此时你不保证消息发送已成功完成。可能会出现：网络抖动，导致消息压根就没有发送到Broker端；或者消息本身不合规导致Broker拒绝接收（比如消息太大了，超过了Broker的限制）。

实际上，使用producer.send(msg, callback)接口就能避免这个问题，根据回调，一旦出现消息提交失败的情况，就可以有针对性地进行处理。如果是因为那些瞬时错误，Producer重试就可以了；如果是消息不合规造成的，那么调整消息格式后再次发送。总之，处理发送失败的责任在Producer端而非Broker端。当然，如果此时broker宕机，那就另当别论，需要及时处理broker异常问题。

**消费端**

Consumer端丢数据的情况，稍微复杂点。Consumer有个”位移“(offset)的概念，表示Consumer当前消费到topic分区的哪个位置。如图：

kafka通过先消费消息，后更新offset，来保证消息不丢失。但是这样可能会出现消息重复的情况，具体如何保证only-once，后续再单独分享。

当我们consumer端开启多线程异步去消费时，情况又会变得复杂一些。此时consumer自动地向前更新offset，假如其中某个线程运行失败了，它负责的消息没有被成功处理，但位移已经被更新了，因此这条消息对于consumer而言实际上是丢失了。这里的关键就在自动提交offset，如何真正地确认消息是否真的被消费，再进行更新offset。

这个问题的解决起来也简单：如果是多线程异步处理消费消息，consumer不要开启自动提交offset，consumer端程序自己来处理offset的提交更新。提醒你一下，单个consumer程序使用多线程来消费消息说起来容易，写成代码还是有点麻烦的，因为你很难正确地处理offset的更新，也就是说避免无消费消息丢失很简单，但极易出现消息被消费了多次的情况。

**实践配置**

最后分享下kafka无消息丢失配置：

1. producer端使用producer.send(msg, callback)带有回调的send方法。
2. 设置acks = all。acks是Producer的一个参数，代表“已提交”消息的定义。如果设置成all，则表明所有Broker都要接收到消息，该消息才算是“已提交”。
3. 设置retries为一个较大的值。同样是Producer的参数。当出现网络抖动时，消息发送可能会失败，此时配置了retries的Producer能够自动重试发送消息，尽量避免消息丢失。
4. 设置unclean.leader.election.enable = false。这是Broker端的参数，在kafka版本迭代中社区也多次反复修改过他的默认值，之前比较具有争议。它控制哪些Broker有资格竞选分区的Leader。如果一个Broker落后原先的Leader太多，那么它一旦成为新的Leader，将会导致消息丢失。故一般都要将该参数设置成false。
5. 设置replication.factor >= 3。这也是Broker端的参数。保存多份消息冗余，不多解释了。
6. 设置min.insync.replicas > 1。Broker端参数，控制消息至少要被写入到多少个副本才算是“已提交”。设置成大于 1 可以提升消息持久性。在生产环境中不要使用默认值 1。确保replication.factor > min.insync.replicas。如果两者相等，那么只要有一个副本离线，整个分区就无法正常工作了。推荐设置成replication.factor = min.insync.replicas + 1。
7. 确保消息消费完成再提交。Consumer端有个参数enable.auto.commit，最好设置成false，并自己来处理offset的提交更新。

分区策略，consume 端：

众所周知，Apache Kafka是基于生产者和消费者模型作为开源的分布式发布订阅消息系统（当然，目前Kafka定位于an open-source distributed event streaming platform），由Scala和Java编写。

Kafka提供了类似于JMS的特性，但设计上又有很大区别，它不是JMS规范的实现，如Kafka允许多个消费者主动拉取数据，而在JMS中只有点对点模式消费者才会主动拉取数据。

Kafka producer在向Kafka集群发送消息时，需要指定topic，Kafka根据topic对消息进行归类（逻辑划分），而一个topic通常会有多个partition分区，落到磁盘上就是多个partition目录。

Kafka consumer为了及时消费消息，会以Consumer Group（消费组）的形式，启动多个consumer消费消息。不同的消费组在消费消息时彼此互不影响，同一个消费组的consumer协调在一起消费订阅的topic所有分区消息。这就引申一个问题：消费组中的consumer是如何确定自己该消费哪些分区的数据的？

Kafka提供了多种分区策略如RoundRobin（轮询）、Range（按范围），可通过参数partition.assignment.strategy进行配置。

一般情况下，在topic和消费组不发生变化，Kafka会根据topic分区、消费组情况等确定分区策略，但是当发生以下情况时，会触发Kafka的分区重分配：

1. Consumer Group中的consumer发生了新增或者减少

1. 同一个Consumer Group新增consumer
2. Consumer Group订阅的topic分区发生变化如新增分区

2. Consumer Group订阅的topic分区发生变化如新增分区

本文通过下面的场景，来分别阐述Kafka主要的分配策略RoundRobin和Range：

**Range Strategy**

Range策略是针对topic而言的，在进行分区分配时，为了尽可能保证所有consumer均匀的消费分区，会对同一个topic中的partition按照序号排序，并对consumer按照字典顺序排序。

然后为每个consumer划分固定的分区范围，如果不够平均分配，那么排序靠前的消费者会被多分配分区。具体就是将partition的个数除于consumer线程数来决定每个consumer线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多分配分区。

通过下面公式更直观：

假设n = 分区数 / 消费者数量，m = 分区数 % 消费者线程数量，那么前m个消费者每个分配n+1个分区，后面的（消费者线程数量 - m）个消费者每个分配n个分区。

举个例子：

一个消费组CG1中有C0和C1两个consumer，消费Kafka中的主题t1。t1的分区数为10，并且C1的num.streams为1，C2的num.streams为2。

经过排序后，分区为：0, 1, 2, 3, 4, 5, 6, 7, 8, 9；CG1中消费者线程为C0-0、C1-0、C1-1。然后因为 10除3除不尽，那么消费者线程C0-0将会多分配分区，所以分区分配之后结果如下：

C0-0 将消费0、1、2、3分区

C1-0 将消费4、5、6分区

C1-1 将消费7、8、9分区

当存在有2个Kafka topic(t1和t2)，它们都有有10个partition，那么最后分区结果为：

C0-0 将消费t1主题的0、1、2、3分区以及t2主题的0、1、2、3分区

C1-0 将消费t1主题的4、5、6分区以及t2主题的4、5、6分区

C2-1 将消费t1主题的7、8、9分区以及t2主题的7、8、9分区

如上场景，随着topic的增多，那么针对每个topic，消费者C0-0都将多消费1个分区，topic越多比如为N个，C0-0消费的分区会比其他消费者明显多消费N个分区。

可以明显的看到这样的分配并不均匀，如果将类似的情形扩大，有可能会出现部分消费者过载的情况，这就是Range分区策略的一个很明显的弊端。

**RoundRobin Strategy**

RoundRobin策略的工作原理：将所有topic的partition组成TopicAndPartition列表，然后对TopicAndPartition列表按照hashCode进行排序：

val allTopicPartitions = ctx.partitionsForTopic.flatMap { case(topic, partitions) =>

info("Consumer %s rebalancing the following partitions for topic %s: %s"

.format(ctx.consumerId, topic, partitions))

partitions.map(partition => {

TopicAndPartition(topic, partition)

})

}.toSeq.sortWith((topicPartition1, topicPartition2) => {

/\*

\* Randomize the order by taking the hashcode to reduce the likelihood of all partitions of a given topic ending

\* up on one consumer (if it has a high enough stream count).

\*/

topicPartition1.toString.hashCode < topicPartition2.toString.hashCode

})

最后按照RoundRobin风格将分区分别分配给不同的消费者。

使用RoundRobin策略必须满足以下条件：

1.同一个Consumer Group里面的所有consumer的num.streams必须相等

2.每个consumer订阅的topic必须相同

假设消费组CG1中有C0和C1两个consumer的num.streams都为2。按照hashCode排序完的topic-partition组依次为t1-5, t1-3, t1-0, t1-8, t1-2, t1-1, t1-4, t1-7, t1-6, t1-9，我们的消费者排序为C0-0, C0-1, C1-0, C1-1，最后分区分配的结果为：

C0-0将消费t1-5、t1-2、t1-6分区

C0-1将消费t1-3、t1-1、t1-9分区

C1-0将消费t1-0、t1-4分区

C1-1将消费t1-8、t1-7分区

多个主题的分区分配和单个主题类似，这里就不在介绍了。

上面RoundRobin要求每个consumer订阅的topic必须相同，当订阅的topic不同时，那么在执行分区分配的时候就不是完全的轮询分配，有可能会导致分区分配的不均匀。比如，某个consumer没有订阅消费组内的某个topic，那么在分配分区的时候，这个consumer将分配不到这个topic的分区。

除了上述的介绍的RoundRobin和Range分配策略，Kafka还有Sticky分配策略，它主要有两个目的：

1. 分区的分配要尽可能的均匀
2. 分区的分配尽可能的与上次分配的保持相同

当两者发生冲突时，第一个目标优先于第二个目标。鉴于这两个目标，StickyAssignor策略的具体实现要比RangeAssignor和RoundRobinAssignor这两种分配策略要复杂很多。

<https://cloud.tencent.com/developer/article/1708388>

Producer 端：

# Kafka分区分配策略

[川行](https://developer.aliyun.com/profile/3ti5keb2ril6g) https://img.alicdn.com/tfs/TB10fDzdND1gK0jSZFsXXbldVXa-36-32.png 2020-07-21 262浏览量

**简介：** 我们知道每个Topic会分配为很多partitions，Producers会将数据分配到每个partitions中，然后消费者Consumers从partitions中获取数据消费，那么Producers是如何将数据分到partitions中？Consumers又怎么知道从哪个partitions中消费数据？

我们知道每个Topic会分配为很多partitions，Producers会将数据分配到每个partitions中，然后消费者Consumers从partitions中获取数据消费，那么Producers是如何将数据分到partitions中？Consumers又怎么知道从哪个partitions中消费数据？

# 生产者往Topic写数据

我们从product.send方法入手，看看里面的具体实现，可以看到在调用send方法时，其内部是调用了doSend方法，在doSend方法中有一个获取partitions的方法

int partition = partition(record, serializedKey, serializedValue, cluster);

private int partition(ProducerRecord<K, V> record, byte[] serializedKey, byte[] serializedValue, Cluster cluster) {

Integer partition = record.partition();

return partition != null ?

partition :

partitioner.partition(

record.topic(), record.key(), serializedKey, record.value(), serializedValue, cluster);

}

从上面代码中，首先先选择配置的分区，如果没有配置则使用默认的分区，即使用了DefaultPartitioner中的partition方法

public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {

List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);

int numPartitions = partitions.size();

if (keyBytes == null) {

int nextValue = nextValue(topic);

List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);

if (availablePartitions.size() > 0) {

int part = Utils.toPositive(nextValue) % availablePartitions.size();

return availablePartitions.get(part).partition();

} else {

*// 没有可用的分区，则给一个不可用分区*

return Utils.toPositive(nextValue) % numPartitions;

}

} else {

*// 根据key的hash值和分区数取模*

return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;

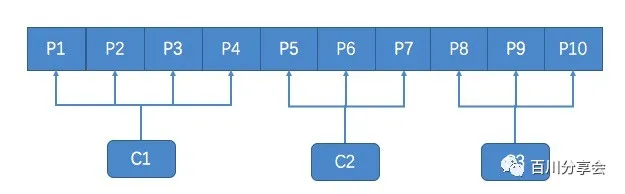
}

}

上面代码先会根据topic获取所有的分区  
1，如果key为null，则通过先产生随机数，之后在该数上自增的方式产生一个数nextValue，如果存在可用分区，将nextValue转为正数之后对可用分区进行取模操作，如果不存在可用分区，则将nextValue对总分区数进行取模操作

2，如果key不为空，就先获取key的hash值，然后和分区数进行取模操作

消费者从Topic读数据  
kafka默认对消费分区指定了两种策略，分别为Range策略（org.apache.kafka.clients.consumer.RangeAssignor)和RoundRobin策略(org.apache.kafka.clients.consumer.RoundRobinAssignor)，它们都实现了PartitionAssignor接口

Range策略  
比如有10个分区，分别为P1、P2、P3、P4、P5、P6、P7、P8、P9、P10，三个消费者C1、C2、C3，消费如下图：  
  
我们来看看源代码：

@Override

public Map<String, List<TopicPartition>> assign(Map<String, Integer> partitionsPerTopic,

Map<String, Subscription> subscriptions) {

*// 得到topic和订阅的消费者集合信息，例如{t1:[c1,c2,c3], t2:[c1,c2,c3,c4]}*

Map<String, List<String>> consumersPerTopic = consumersPerTopic(subscriptions);

Map<String, List<TopicPartition>> assignment = new HashMap<>();

*// 将consumersPerTopic信息转换为assignment，memberId就是消费者client.id+uuid(kafka在client.id上追加的)*

for (String memberId : subscriptions.keySet())

assignment.put(memberId, new ArrayList<TopicPartition>());

*// 遍历每个Topic，获取所有的订阅消费者*

for (Map.Entry<String, List<String>> topicEntry : consumersPerTopic.entrySet()) {

String topic = topicEntry.getKey();

List<String> consumersForTopic = topicEntry.getValue();

Integer numPartitionsForTopic = partitionsPerTopic.get(topic);

*// 如果Topic没有分区，则调过*

if (numPartitionsForTopic == null)

continue;

*// 将Topic的订阅者根据字典排序*

Collections.sort(consumersForTopic);

*// 总分区数/订阅者的数量 得到每个订阅者应该分配分区数*

int numPartitionsPerConsumer = numPartitionsForTopic / consumersForTopic.size();

*// 无法整除的剩余分区数量*

int consumersWithExtraPartition = numPartitionsForTopic % consumersForTopic.size();

List<TopicPartition> partitions = AbstractPartitionAssignor.partitions(topic, numPartitionsForTopic);

*//遍历所有的消费者*

for (int i = 0, n = consumersForTopic.size(); i < n; i++) {

*//分配到的分区的开始位置*

int start = numPartitionsPerConsumer \* i + Math.min(i, consumersWithExtraPartition);

*// 分配到的分区数量（整除分配到的分区数量，加上1个无法整除分配到的分区--如果有资格分配到这个分区的话。判断是否有资格分配到这个分区：如果整除后余数为m，那么排序后的消费者集合中前m个消费者都能分配到一个额外的分区）*

int length = numPartitionsPerConsumer + (i + 1 > consumersWithExtraPartition ? 0 : 1);

*//给消费者分配分区*

assignment.get(consumersForTopic.get(i)).addAll(partitions.subList(start, start + length));

}

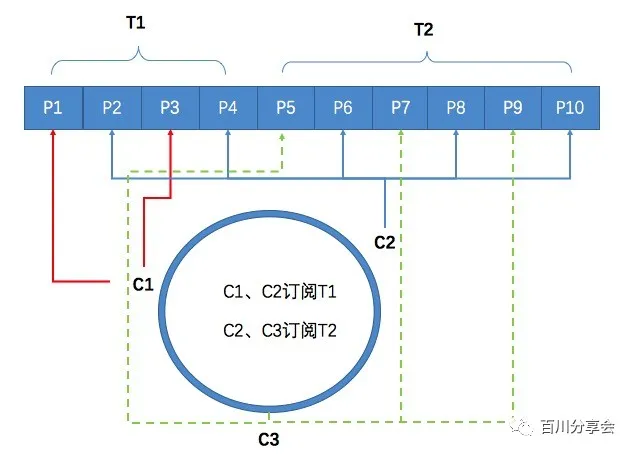
}

return assignment;

}

上面的代码添加了注释很清楚的展现了range的实现，对应上面的例子，如果有4个消费者C1、C2、C3、C4，那么根据上面的算法：

C1 -> [P1,P2,P3] ，C2 -> [P4,P5,P6] ，C3 -> [P7,P8] C4 -> [P9,P10] 。取余多出来的两个分区，由最前n个消费者来消费

RoundRobin策略  
将主题的所有分区依次分配给消费者，比如有两个Topic：T1[P1,P2,P3,P4]，T2[P5,P6,P7,P8,P9,P10]，若C1、C2订阅了T1，C2、C3订阅了T2,那么C1将消费T1[P1,P3]，C2将消费T1[P2,P4,P6,P8,P10]，C3将消费T2[P5,P7,P9]，如下图：  


@Override

public Map<String, List<TopicPartition>> assign(Map<String, Integer> partitionsPerTopic,

Map<String, Subscription> subscriptions) {

Map<String, List<TopicPartition>> assignment = new HashMap<>();

for (String memberId : subscriptions.keySet())

assignment.put(memberId, new ArrayList<TopicPartition>());

*// 将消费集合先按字典排序，构建成一个环形迭代器*

CircularIterator<String> assigner = new CircularIterator<>(Utils.sorted(subscriptions.keySet()));

*// 按Topic的名称排序，得到Topic下的所有分区*

for (TopicPartition partition : allPartitionsSorted(partitionsPerTopic, subscriptions)) {

final String topic = partition.topic();

while (!subscriptions.get(assigner.peek()).topics().contains(topic))

assigner.next();

*// 给消费者分配分区，并轮询到下一个消费者*

assignment.get(assigner.next()).add(partition);

}

return assignment;

}

*/\*\**

*\* 根据消费者得到订阅的Topic下的所有分区*

*\* Topic按名称字典排序*

*\*/*

public List<TopicPartition> allPartitionsSorted(Map<String, Integer> partitionsPerTopic,

Map<String, Subscription> subscriptions) {

SortedSet<String> topics = new TreeSet<>();

for (Subscription subscription : subscriptions.values())

topics.addAll(subscription.topics());

List<TopicPartition> allPartitions = new ArrayList<>();

for (String topic : topics) {

Integer numPartitionsForTopic = partitionsPerTopic.get(topic);

if (numPartitionsForTopic != null)

allPartitions.addAll(AbstractPartitionAssignor.partitions(topic, numPartitionsForTopic));

}

return allPartitions;

}

<https://developer.aliyun.com/article/768415>

# kafka幂等性和事务使用及实现原理

## 开篇

在开始这篇之前，先抛出问题，这章解决如下问题：

1. 如何开启幂等性？
2. 如何使用事务？
3. 幂等性的原理
4. 事务实现原理

## 正文

### Producer 幂等性

Producer 的幂等性指的是当发送同一条消息时，数据在 Server 端只会被持久化一次，数据不丟不重，但是这里的幂等性是有条件的：

* 只能保证 Producer 在单个会话内不丟不重，如果 Producer 出现意外挂掉再重启是无法保证的（幂等性情况下，是无法获取之前的状态信息，因此是无法做到跨会话级别的不丢不重）;
* 幂等性不能跨多个 Topic-Partition，只能保证单个 partition 内的幂等性，当涉及多个 Topic-Partition 时，这中间的状态并没有同步。

如果需要跨会话、跨多个 topic-partition 的情况，需要使用 Kafka 的事务性来实现。

使用方式：props.put(ProducerConfig.ENABLE\_IDEMPOTENCE\_CONFIG, "true");

当幂等性开启的时候acks即为all。如果显性的将acks设置为0，-1，那么将会报错Must set acks to all in order to use the idempotent producer. Otherwise we cannot guarantee idempotence.

示例：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | Properties props = new Properties(); props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "127.0.0.1:9092"); props.put(ProducerConfig.ACKS\_CONFIG, "all"); props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName()); props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName()); props.put(ProducerConfig.ENABLE\_IDEMPOTENCE\_CONFIG, "true");  KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(props); kafkaProducer.send(new ProducerRecord<String, String>("truman\_kafka\_center", "1", "hello world.")).get(); kafkaProducer.close(); |

### 幂等性原理

幂等性是通过两个关键信息保证的，PID(Producer ID)和sequence numbers。

* PID 用来标识每个producer client
* sequence numbers 客户端发送的每条消息都会带相应的 sequence number，Server 端就是根据这个值来判断数据是否重复

producer初始化会由server端生成一个PID,然后发送每条信息都包含该PID和sequence number，在server端，是按照partition同样存放一个sequence numbers 信息，通过判断客户端发送过来的sequence number与server端number+1差值来决定数据是否重复或者漏掉。

通常情况下为了保证数据顺序性，我们可以通过max.in.flight.requests.per.connection=1来保证，这个也只是针对单实例。在kafka2.0+版本上，只要开启幂等性，不用设置这个参数也能保证发送数据的顺序性。

### 为什么要求 MAX\_IN\_FLIGHT\_REQUESTS\_PER\_CONNECTION 小于等于5

其实这里，要求 MAX\_IN\_FLIGHT\_REQUESTS\_PER\_CONNECTION 小于等于 5 的主要原因是：Server 端的 ProducerStateManager 实例会缓存每个 PID 在每个 Topic-Partition 上发送的最近 5 个batch 数据（这个 5 是写死的，至于为什么是 5，可能跟经验有关，当不设置幂等性时，当这个设置为 5 时，性能相对来说较高，社区是有一个相关测试文档），如果超过 5，ProducerStateManager 就会将最旧的 batch 数据清除。

假设应用将 MAX\_IN\_FLIGHT\_REQUESTS\_PER\_CONNECTION 设置为 6，假设发送的请求顺序是 1、2、3、4、5、6，这时候 server 端只能缓存 2、3、4、5、6 请求对应的 batch 数据，这时候假设请求 1 发送失败，需要重试，当重试的请求发送过来后，首先先检查是否为重复的 batch，这时候检查的结果是否，之后会开始 check 其 sequence number 值，这时候只会返回一个 OutOfOrderSequenceException 异常，client 在收到这个异常后，会再次进行重试，直到超过最大重试次数或者超时，这样不但会影响 Producer 性能，还可能给 Server 带来压力（**相当于client 狂发错误请求**）。

### Kafka 事务性

#### 示例

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 | //Producer Properties props = new Properties(); props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "127.0.0.1:9092"); props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName()); props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName()); props.put(ProducerConfig.ENABLE\_IDEMPOTENCE\_CONFIG, "true"); props.put(ProducerConfig.TRANSACTIONAL\_ID\_CONFIG, "transactional\_id-0"); KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(props); kafkaProducer.initTransactions(); kafkaProducer.beginTransaction(); for (int i = 0; i < 10; i++) {  kafkaProducer.send(new ProducerRecord<String, String>("truman\_kafka\_center", "key"+i, "hello world.")).get(); } kafkaProducer.commitTransaction(); kafkaProducer.close(); //Consumer Properties config = new Properties(); config.put("group.id", "test11"); config.put("bootstrap.servers", "127.0.0.1:9092"); config.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); config.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); config.put(ConsumerConfig.ISOLATION\_LEVEL\_CONFIG, "read\_committed");  KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(config); consumer.subscribe(Arrays.asList(TOPIC)); boolean isConsumer = true; while (isConsumer) {  ConsumerRecords<String, String> records = (ConsumerRecords<String, String>) consumer  .poll(Duration.ofMillis(100));  for (ConsumerRecord<String, String> record : records) {  System.out.println("consumer message: key =" + record.key() + " value:" + record.value());  } } consumer.close(); } |

#### 事务实现原理

###### (1)查找TransactionCoordinator

通过transaction\_id 找到TransactionCoordinator，具体算法是Utils.abs(transaction\_id.hashCode %transactionTopicPartitionCount )，获取到partition，再找到该partition的leader,即为TransactionCoordinator。

###### (2)获取PID

凡是开启幂等性都是需要生成PID(Producer ID),只不过未开启事务的PID可以在任意broker生成，而开启事务只能在TransactionCoordinator节点生成。这里只讲开启事务的情况，Producer Client的initTransactions()方法会向TransactionCoordinator发起InitPidRequest ，这样就能获取PID。这里面还有一些细节问题，这里不探讨，例如transaction\_id 之前的事务状态什么的。但需要说明的一点是这里**会将 transaction\_id 与相应的 TransactionMetadata 持久化到事务日志**（\_transaction\_state）中。

###### (3)开启事务

Producer调用beginTransaction开始一个事务状态，这里只是在客户端将本地事务状态转移成 IN\_TRANSACTION，只有在发送第一条信息后，TransactionCoordinator才会认为该事务已经开启。

###### (4)Consume-Porcess-Produce Loop

这里说的是一个典型的consume-process-produce场景：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | while (true) {  ConsumerRecords records = consumer.poll(Duration.ofMillis(1000));  producer.beginTransaction();  //start  for (ConsumerRecord record : records){  producer.send(producerRecord(“outputTopic1”, record));  producer.send(producerRecord(“outputTopic2”, record));  }  producer.sendOffsetsToTransaction(currentOffsets(consumer), group);  //end  producer.commitTransaction(); } |

该阶段主要经历以下几个步骤：

1. AddPartitionsToTxnRequest
2. ProduceRequest
3. AddOffsetsToTxnRequest
4. TxnOffsetsCommitRequest

关于这里的详细介绍可以查看参考链接，或者直接查看官网文档！

###### (5)提交或者中断事务

Producer 调用 commitTransaction() 或者 abortTransaction() 方法来 commit 或者 abort 这个事务操作。

基本上经历以下三个步骤，才真正结束事务。

1. EndTxnRequest
2. WriteTxnMarkerRquest
3. Writing the Final Commit or Abort Message

其中EndTxnRequest是在Producer发起的请求，其他阶段都是在TransactionCoordinator端发起完成的。WriteTxnMarkerRquest是发送请求到partition的leader上写入事务结果信息（ControlBatch）,第三步主要是在\_transaction\_state中标记事务的结束。

<http://trumandu.github.io/2019/07/08/kafka%E5%B9%82%E7%AD%89%E6%80%A7%E5%92%8C%E4%BA%8B%E5%8A%A1%E4%BD%BF%E7%94%A8%E5%8F%8A%E5%AE%9E%E7%8E%B0%E5%8E%9F%E7%90%86/>

### （1）数据倾斜的介绍

#### 1）数据分区的策略：

     - **随机分区**：每一个数据分配的任意一个分区的概率是均等的  
     - **Hash分区**：使用数据的Hash分区值，%分区数。（导致数据倾斜的原因）  
     - **范围分区**：将数据范围划分，数据分配到不同的范围中（分布式的全局排序）

#### 2）数据倾斜的原因：

  Shuffle数据之后导致数据分布不均匀，但是所有节点的机器的性能都是一样的，程序也是一样的，就是数据量不一致，所以决定了task的执行时长就被数据量决定了。

#### 3）定位数据倾斜的代码：

  数据倾斜发生在shuffle过程，有shuffle过程的算子有：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。或者查看哪一个task执行缓慢、内存溢出...

#### 4）查看数据倾斜的key的分布情况：

*//使用spark中的抽样算子sample，查看相应的key的分布*

val sampledPairs = pairs.sample(false, 0.1) *//抽样*

val sampledWordCounts = sampledPairs.countByKey()

sampledWordCounts.**foreach**(println(\_))

### （2）数据倾斜的解决方案

#### 1）过滤掉少数数据倾斜的key：

  如果发现导致数据倾斜的key是极少数，并且对计算本身影响不大，那么这种方案比较适用。  
   实现原理：通过spark的sample算子，定位到数据倾斜的key，然后使用filter算子将其过滤即可。

#### 2）提高shuffle的并行度：

   这是一种尝试性策略：就是提高增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。

#### 3）两阶段的聚合（局部聚合和全局聚合）：

**适用场景**：对RDD执行reduceByKey等这类有聚合操作的shuffle算子或者spark SQL使用使用group by语句进行分组聚合，比较适用。  
  
原理：将原本相同的key通过附加随机前缀的方式，变成多个不同key，就可以让原本被一个task处理的数据分散到多个task上做局部聚合，进行解决单个task处理数据量过多的问题。接着去除随机前缀，再次进行全局的聚合，就可以得到最终的结果。  
代码实现：

object \_01SparkDataSkewTwoStageOps {

def main(args: **Array**[String]): Unit = {

Logger.getLogger("org.apache.hadoop").setLevel(Level.WARN)

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.project-spark").setLevel(Level.WARN)

val sc = SparkUtil.sparkContext("local[2]", "\_01SparkDataSkewTwoStageOps")

val **list** = **List**(

"hello you hello hello me",

"hello you hello hello shit",

"oh hello she study"

)

val listRDD = sc.parallelize(**list**)

val pairsRDD = listRDD.flatMap(line => line.split("\\s+")).map((\_, 1))

*//step 1 找到发生数据倾斜key*

val sampleRDD = pairsRDD.sample(**false**, 0.6)

val cbk= sampleRDD.countByKey()

*// cbkRDD.foreach(println)*

val sortedInfo = cbk.toBuffer.sortWith((t1, t2) => t1.\_2 > t2.\_2)

val dataSkewKey = sortedInfo.head.\_1

*// sortedInfo.foreach(println)*

println("发生了数据倾斜的Key：" + dataSkewKey)

*//step 2 给对应的key打上N以内的随机前缀*

val prefixPairsRDD = pairsRDD.map{**case** (word, count) => {

**if**(word.equals(dataSkewKey)) {

val random = **new** Random()

val prefix = random.nextInt(2)*//0 1*

(s"${prefix}\_${word}", count)

} **else** {

(word, count)

}

}}

prefixPairsRDD.**foreach**(println)

*//step 3 局部聚合*

val partAggrInfo = prefixPairsRDD.reduceByKey(\_+\_)

println("===============>局部聚合之后的结果：")

partAggrInfo.**foreach**(println)

*//step 4 全局聚合*

*//step 4.1 去掉前缀*

val unPrefixPairRDD = partAggrInfo.map{**case** (word, count) => {

**if**(word.contains("\_")) {

(word.substring(word.indexOf("\_") + 1), count)

} **else** {

(word, count)

}

}}

println("================>去掉随机前缀之后的结果：")

unPrefixPairRDD.**foreach**(println)

*// step 4.2 全局聚合*

val fullAggrInfo = unPrefixPairRDD.reduceByKey(\_+\_)

println("===============>全局聚合之后的结果：")

fullAggrInfo.**foreach**(println)

sc.stop()

}

}

#### 4）将reduce join 转换为map join（大小表）：

   适用场景：在对RDD使用join操作，或者是在sparksql 中使用join语句的时候，而且join操作中的一个RDD或者表的数据量比较小，此方法适用  
   实现原理：有reduce join的过程一定有shuffle，有shuffle就可能出现数据的倾斜，所以将reduce join使用map join 代替。如果一个RDD是比较小的，那么可以使用广播变量的方式，将小RDD发送到各个worker的executor中，实现本地的连接  
代码实现：

object \_02SparkRDDBroadcastOps {

def main(args: Array[String]): Unit = {

Logger.getLogger("org.apache.hadoop").setLevel(Level.WARN)

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.project-spark").setLevel(Level.WARN)

val conf = **new** SparkConf()

.setMaster("local[2]")

.setAppName(s"${\_02SparkRDDBroadcastOps.getClass.getSimpleName}")

val sc = **new** SparkContext(conf)

val stu = List(

"1 郑祥楷 1",

"2 王佳豪 1",

"3 刘鹰 2",

"4 宋志华 3",

"5 刘帆 4",

"6 OLDLi 5"

)

val cls = List(

"1 1807bd-bj",

"2 1807bd-sz",

"3 1807bd-wh",

"4 1807bd-xa",

"7 1805bd-bj"

)

*/\**

*使用广播变量来完成上述操作*

*一般用户表都比较大，而班级表相对很小，符合我们在共享变量中提出的第一个假设*

*所以我们可以尝试使用广播变量来进行解决*

*\*/*

val stuRDD = sc.parallelize(stu)

*//cls-->map---->*

val map = cls.map{**case** line => {

(line.substring(0, line.indexOf(" ")), line.substring(line.indexOf(" ")).trim)

}}.toMap

*//map--->broadcast*

val clsMapBC:Broadcast[Map[String, String]] = sc.broadcast(map)

stuRDD.map{**case** line => {

val map = clsMapBC.value

val fields = line.split("\\s+")

val cid = fields(2)

*// map.get(cid)*

val className = map.getOrElse(cid, "UnKnown")

s"${fields(0)}\t${fields(1)}\t${className}"*//在mr中学习到的map join*

}}.foreach(println)

sc.stop()

}

#### 5）采样倾斜的key并拆分join操作（大大表）：

  适用场景：在hive两张表进行join的时候，如果两张表的数据都很大，并且，一张表的数据很均匀，但是另一张表的数据有少量的key数据量过大，此时使用这个解决方案  
  实现原理：对于join导致的数据倾斜，如果只是某几个key导致了倾斜，可以将少数几个key分拆成独立RDD，并附加随机前缀打散成n份去进行join，此时这几个key对应的数据就不会集中在少数几个task上，而是分散到多个task进行join了。  
代码实现：

一张表中：

Id num

1 100W

2 10

3 10

4 10

5 10

可以使用union的方式来启动多个job并行执行：

//通过分离数据量大key来解决数据倾斜

**select** **count**(\*) **from** t\_test **where** **id** !=1 **group** **by** **id**

**union**

**select** **count**(\*) **from** t\_test **where** **id** ==1 **group** **by** **id**

#### 6）使用随机前缀和扩容RDD进行join（大量key的数据倾斜）：

  适用场景：如果进行join操作时，RDD中有大量的key导致数据倾斜，那么进行拆分可以也没有意义，此时使用这种方法  
  
  实现原理：这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。  
代码实现：  
  
左表的连接条件的值，可以在某个范围内进行随机，并且这个随机值有多少个，那么右表的数据就要复制多少份。

object \_03SparkJoinDataSkewOps {

def main(args: Array[String]): Unit = {

Logger.getLogger("org.apache.hadoop").setLevel(Level.WARN)

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.project-spark").setLevel(Level.WARN)

val sc = SparkUtil.sparkContext("local[2]", "\_03SparkJoinDataSkewOps")

val list1 = List(

"hello 1",

"hello 2",

"hello 3",

"hello 4",

"you 1",

"me 2"

)

val list2 = List(

"hello zhangsan dfsadfasdfsa",

"hello lisi adfasdfasd",

"you wangwu adfasdfs",

"me zhouqi adfadfa"

)

*//<key, value>*

val listRDD1 = sc.parallelize(list1).map(line => {

val fields = line.split("\\s+")

(fields(0), fields(1))

})

*//<key, value>*

val listRDD2 = sc.parallelize(list2).map(line => {

val fields = line.split("\\s+")

(fields(0), fields(1))

})

val joinRDD: RDD[(String, (String, String))] = dataSkewRDDJoin(sc, listRDD1, listRDD2)

println("最后进行join的结果：")

joinRDD.foreach(println)

sc.stop()

}

private def dataSkewRDDJoin(sc: SparkContext, listRDD1: RDD[(String, String)], listRDD2: RDD[(String, String)]) = {

*//假设listRDD1中的部分key有数据倾斜，所以我在进行join操作的时候，需要进行拆分计算*

*//step 1 找到发生数据倾斜的key*

val dataSkewKeys = listRDD1.sample(false, 0.6).countByKey().toList.sortWith((t1, t2) => t1.\_2 > t2.\_2).take(1).map(t => t.\_1)

println("通过sample算子得到的可能发生数据倾斜的key：" + dataSkewKeys)

*//step 2 对listRDD1和listRDD2中的数据按照dataSkewKeys各拆分成两个部分*

*//step 2.1 讲dataSkewKeys进行广播*

val dskBC = sc.broadcast(dataSkewKeys)

*// step 2.2 进行拆分*

val dataSkewRDD1 = listRDD1.filter { **case** (word, value) => {

*//有数据倾斜的rdd--->dataskewRDD1*

val dsks = dskBC.value

dsks.contains(word)

}

}

val commonRDD1 = listRDD1.filter { **case** (word, value) => {

*//没有数据倾斜的rdd--->commonRDD1*

val dsks = dskBC.value

!dsks.contains(word)

}

}

val dataSkewRDD2 = listRDD2.filter { **case** (word, value) => {

*//有数据倾斜的rdd--->dataskewRDD1*

val dsks = dskBC.value

dsks.contains(word)

}

}

val commonRDD2 = listRDD2.filter { **case** (word, value) => {

*//没有数据倾斜的rdd--->commonRDD1*

val dsks = dskBC.value

!dsks.contains(word)

}

}

}

*//step 3 对dataskewRDD进行添加N以内随机前缀*

*// step 3.1 添加随机前缀*

val prefixDSRDD1:RDD[(String, String)] = dataSkewRDD1.map { case (word, value) => {

val random = **new** Random()

val prefix = random.nextInt(2)

(s"${prefix}\_${word}", value)

}

}

*// step 3.2 另一个rdd进行扩容*

val prefixDSRDD2:RDD[(String, String)] = dataSkewRDD2.flatMap { case (word, value) => {

val ab = ArrayBuffer[(String, String)]()

**for** (i <- 0 until 2) {

ab.append((s"${i}\_${word}", value))

}

ab

}

}

println("---->有数据倾斜RDD1添加前缀成prefixDSRDD1的结果：" + prefixDSRDD1.collect().mkString(","))

println("---->有数据倾斜RDD2扩容之后成prefixDSRDD2的结果：" + prefixDSRDD2.collect().mkString(","))

*// step 4 分步进行join操作*

*// step 4.1 有数据倾斜的prefixDSRDD1和prefixDSRDD2进行join*

val prefixJoinDSRDD = prefixDSRDD1.join(prefixDSRDD2)

*//ste 4.2 无数据倾斜的commonRDD1和commonRDD2进行join*

val commonJoinRDD = commonRDD1.join(commonRDD2)

*// step 4.3 将随机前缀去除*

val dsJionRDD = prefixJoinDSRDD.map { **case** (word, (value1, value2)) => {

(word.substring(2), (value1, value2))

}

}

*//step 5 将拆分进行join之后的结果进行union连接，得到最后的结果 sql union all*

val joinRDD = dsJionRDD.union(commonJoinRDD)

joinRDD

}

}

<https://blog.51cto.com/14048416/2338651>