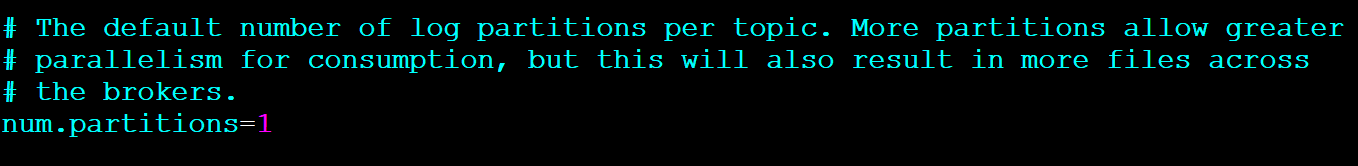
# **Kafka分区(partition)如何分配的**

# **1.  前言**

我们知道，生产者发送消息到主题，消费者订阅主题（以消费者组的名义订阅），而主题下是分区，消息是存储在分区中的，所以事实上生产者发送消息到分区，消费者则从分区读取消息，那么，这里问题来了，**生产者将消息投递到哪个分区**？**消费者组中的消费者实例之间是怎么分配分区的呢**？接下来，就围绕着这两个问题一探究竟。

# **2.  主题的分区数设置**

在server.properties配置文件中可以指定一个全局的分区数设置，这是对每个主题下的分区数的默认设置，默认是1。



当然每个主题也可以自己设置分区数量，如果创建主题的时候没有指定分区数量，则会使用server.properties中的设置。

bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic my-topic --partitions 2 --replication-factor 1

在创建主题的时候，可以使用****--partitions****选项指定主题的分区数量

[root@localhost kafka\_2.11-2.0.0]*# bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic abc*

Topic:abc PartitionCount:2 ReplicationFactor:1 Configs:

Topic: abc Partition: 0 Leader: 0 Replicas: 0 Isr: 0

Topic: abc Partition: 1 Leader: 0 Replicas: 0 Isr: 0

# **3.  生产者与分区**

首先提出一个问题：生产者将消息投递到分区有没有规律？如果有，那么它是如何决定一条消息该投递到哪个分区的呢？

## **3.1.  默认的分区策略**

The default partitioning strategy:

* If a partition is specified in the record, use it
* If no partition is specified but a key is present choose a partition based on a hash of the key
* If no partition or key is present choose a partition in a round-robin fashion

****org.apache.kafka.clients.producer.internals.DefaultPartitioner****

默认的分区策略是：

* 如果在发消息的时候指定了分区，则消息投递到指定的分区
* 如果没有指定分区，但是消息的key不为空，则基于key的哈希值来选择一个分区
* 如果既没有指定分区，且消息的key也是空，则用轮询的方式选择一个分区
* 生产者发布消息至partition的具体文章(包含实现自定义Partitioner，覆写其partition()方法和close()方法)

[https://yanbin.blog/how-kafka-select-partition/#more-8179](https://yanbin.blog/how-kafka-select-partition/" \l "more-8179)

# **4.  分区与消费者**

消费者以组的名义订阅主题，主题有多个分区，消费者组中有多个消费者实例，那么消费者实例和分区之前的对应关系是怎样的呢？

换句话说，就是组中的每一个消费者负责那些分区，这个分配关系是如何确定的呢？



同一时刻，一条消息只能被组中的一个消费者实例消费

消费者组订阅这个主题，意味着主题下的所有分区都会被组中的消费者消费到，如果按照从属关系来说的话就是，主题下的每个分区只从属于组中的一个消费者，不可能出现组中的两个消费者负责同一个分区。

那么，问题来了。如果分区数大于或者等于组中的消费者实例数，那自然没有什么问题，无非一个消费者会负责多个分区，（PS：当然，最理想的情况是二者数量相等，这样就相当于一个消费者负责一个分区）；但是，如果消费者实例的数量大于分区数，那么按照默认的策略（PS：之所以强调默认策略是因为你也可以自定义策略），有一些消费者是多余的，一直接不到消息而处于空闲状态。

话又说回来，假设多个消费者负责同一个分区，那么会有什么问题呢？

我们知道，Kafka它在设计的时候就是要保证分区下消息的顺序，也就是说消息在一个分区中的顺序是怎样的，那么消费者在消费的时候看到的就是什么样的顺序，那么要做到这一点就首先要保证消息是由消费者主动拉取的（pull），其次还要保证一个分区只能由一个消费者负责。倘若，两个消费者负责同一个分区，那么就意味着两个消费者同时读取分区的消息，由于消费者自己可以控制读取消息的offset，就有可能C1才读到2，而C1读到1，C1还没处理完，C2已经读到3了，则会造成很多浪费，因为这就相当于多线程读取同一个消息，会造成消息处理的重复，且不能保证消息的顺序，这就跟主动推送（push）无异。

## **4.1.  消费者分区分配策略**

org.apache.kafka.clients.consumer.internals.AbstractPartitionAssignor

如果是自定义分配策略的话可以继承AbstractPartitionAssignor这个类，它默认有3个实现。

Range 、RoundRobin和Sticky.

### **4.1.1.  Range策略**

range策略对应的实现类是org.apache.kafka.clients.consumer.RangeAssignor

这是默认的分配策略

可以通过消费者配置中****partition.assignment.strategy****参数来指定分配策略，它的值是类的全路径，是一个数组

/\*\*

\* The range assignor works on a per-topic basis. For each topic, we lay out the available partitions in numeric order

\* and the consumers in lexicographic order. We then divide the number of partitions by the total number of

\* consumers to determine the number of partitions to assign to each consumer. If it does not evenly

\* divide, then the first few consumers will have one extra partition.

\*

\* For example, suppose there are two consumers C0 and C1, two topics t0 and t1, and each topic has 3 partitions,

\* resulting in partitions t0p0, t0p1, t0p2, t1p0, t1p1, and t1p2.

\*

\* The assignment will be:

\* C0: [t0p0, t0p1, t1p0, t1p1]

\* C1: [t0p2, t1p2]

\*/

range策略是基于每个主题的

对于每个主题，我们以数字顺序排列可用分区，以字典顺序排列消费者。然后，将分区数量除以消费者总数，以确定分配给每个消费者的分区数量。如果没有平均划分（PS：除不尽），那么最初的几个消费者将有一个额外的分区。

简而言之，就是，

1. range分配策略针对的是主题（PS：也就是说，这里所说的分区指的某个主题的分区，消费者值的是订阅这个主题的消费者组中的消费者实例）
2. 首先，将分区按数字顺序排行序，消费者按消费者名称的字典序排好序
3. 然后，用分区总数除以消费者总数。如果能够除尽，则皆大欢喜，平均分配；若除不尽，则位于排序前面的消费者将多负责一个分区

例如，假设有两个消费者C0和C1，两个主题t0和t1，并且每个主题有3个分区，分区的情况是这样的：t0p0，t0p1，t0p2，t1p0，t1p1，t1p2

那么，基于以上信息，最终消费者分配分区的情况是这样的：

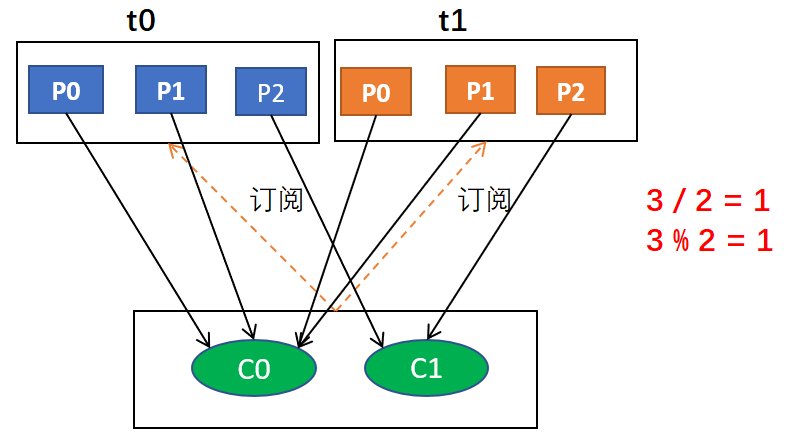
C0: [t0p0, t0p1, t1p0, t1p1]

C1: [t0p2, t1p2]

为什么是这样的结果呢？

因为，对于主题t0，分配的结果是C0负责P0和P1，C1负责P2；对于主题t2，也是如此，综合起来就是这个结果

上面的过程用图形表示的话大概是这样的：



### **4.1.2.  RoundRobin（轮询）策略**

roundronbin分配策略的具体实现是org.apache.kafka.clients.consumer.RoundRobinAssignor

*/\*\**

*\* The round robin assignor lays out all the available partitions and all the available consumers. It*

*\* then proceeds to do a round robin assignment from partition to consumer. If the subscriptions of all consumer*

*\* instances are identical, then the partitions will be uniformly distributed. (i.e., the partition ownership counts*

*\* will be within a delta of exactly one across all consumers.)*

*\**

*\* For example, suppose there are two consumers C0 and C1, two topics t0 and t1, and each topic has 3 partitions,*

*\* resulting in partitions t0p0, t0p1, t0p2, t1p0, t1p1, and t1p2.*

*\**

*\* The assignment will be:*

*\* C0: [t0p0, t0p2, t1p1]*

*\* C1: [t0p1, t1p0, t1p2]*

*\**

*\* When subscriptions differ across consumer instances, the assignment process still considers each*

*\* consumer instance in round robin fashion but skips over an instance if it is not subscribed to*

*\* the topic. Unlike the case when subscriptions are identical, this can result in imbalanced*

*\* assignments. For example, we have three consumers C0, C1, C2, and three topics t0, t1, t2,*

*\* with 1, 2, and 3 partitions, respectively. Therefore, the partitions are t0p0, t1p0, t1p1, t2p0,*

*\* t2p1, t2p2. C0 is subscribed to t0; C1 is subscribed to t0, t1; and C2 is subscribed to t0, t1, t2.*

*\**

*\* Tha assignment will be:*

*\* C0: [t0p0]*

*\* C1: [t1p0]*

*\* C2: [t1p1, t2p0, t2p1, t2p2]*

*\*/*

轮询分配策略是基于所有可用的消费者和所有可用的分区的

与前面的range策略最大的不同就是它不再局限于某个主题

如果所有的消费者实例的订阅都是相同的，那么这样最好了，可用统一分配，均衡分配

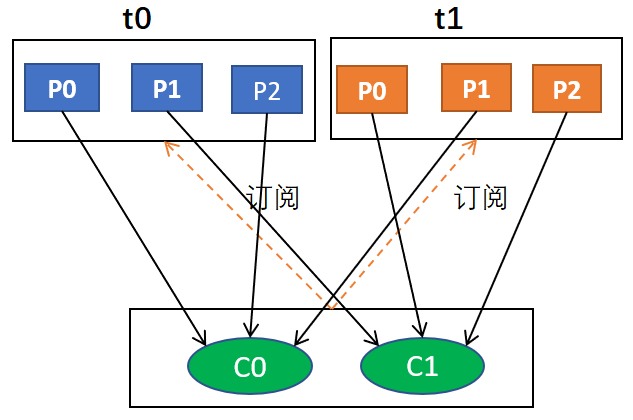
例如，假设有两个消费者C0和C1，两个主题t0和t1，每个主题有3个分区，分别是t0p0，t0p1，t0p2，t1p0，t1p1，t1p2

那么，最终分配的结果是这样的：

C0: [t0p0, t0p2, t1p1]

C1: [t0p1, t1p0, t1p2]

用图形表示大概是这样的：



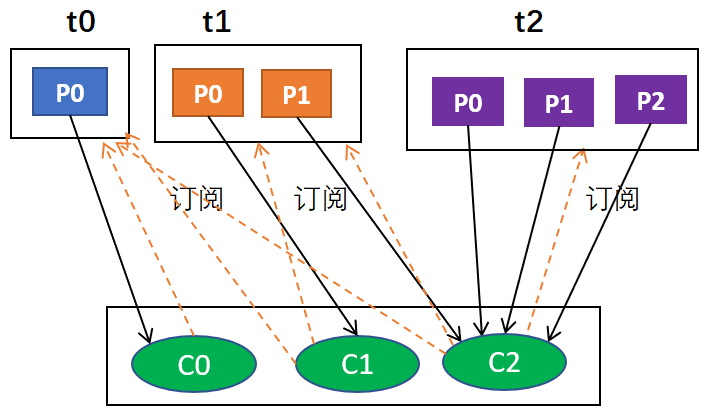
假设，组中每个消费者订阅的主题不一样，分配过程仍然以轮询的方式考虑每个消费者实例，但是如果没有订阅主题，则跳过实例。当然，这样的话分配肯定不均衡。

什么意思呢？也就是说，消费者组是一个逻辑概念，同组意味着同一时刻分区只能被一个消费者实例消费，换句话说，同组意味着一个分区只能分配给组中的一个消费者。事实上，同组也可以不同订阅，这就是说虽然属于同一个组，但是它们订阅的主题可以是不一样的。

另一种分区-消费者分布示例

比如说有主题t0，t1，t2，它们分别有1，2，3个分区，也就是t0有1个分区，t1有2个分区，t2有3个分区；有3个消费者分别从属于3个组，C0订阅t0，C1订阅t0和t1，C2订阅t0，t1，t2；那么，按照轮询分配的话，C0应该负责t0p0，C1应该负责t1p0，其余均由C2负责。

上述过程用图形表示大概是这样的：



为什么最后的结果是

C0: [t0p0]

C1: [t1p0]

C2: [t1p1, t2p0, t2p1, t2p2]

这是因为，按照轮询t0p1由C0负责，t1p0由C1负责，由于同组，C2只能负责t1p1，由于只有C2订阅了t2，所以t2所有分区由C2负责，综合起来就是这个结果

细想一下可以发现，这种情况下跟range分配的结果是一样的

### **4.1.3 StickyAssignor策略**

我们再来看一下StickyAssignor策略，“sticky”这个单词可以翻译为“粘性的”，Kafka从0.11.x版本开始引入这种分配策略，它主要有两个目的：

分区的分配要尽可能的均匀；  
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。我们举例来看一下StickyAssignor策略的实际效果。  
假设消费组内有3个消费者：C0、C1和C2，它们都订阅了4个主题：t0、t1、t2、t3，并且每个主题有2个分区，也就是说整个消费组订阅了t0p0、t0p1、t1p0、t1p1、t2p0、t2p1、t3p0、t3p1这8个分区。最终的分配结果如下：

消费者C0：t0p0、t1p1、t3p0

消费者C1：t0p1、t2p0、t3p1

消费者C2：t1p0、t2p1

这样初看上去似乎与采用RoundRobinAssignor策略所分配的结果相同，但事实是否真的如此呢？再假设此时消费者C1脱离了消费组，那么消费组就会执行再平衡操作，进而消费分区会重新分配。如果采用RoundRobinAssignor策略，那么此时的分配结果如下：

消费者C0：t0p0、t1p0、t2p0、t3p0

消费者C2：t0p1、t1p1、t2p1、t3p1

如分配结果所示，RoundRobinAssignor策略会按照消费者C0和C2进行重新轮询分配。而如果此时使用的是StickyAssignor策略，那么分配结果为：

消费者C0：t0p0、t1p1、t3p0、t2p0

消费者C2：t1p0、t2p1、t0p1、t3p1

可以看到分配结果中保留了上一次分配中对于消费者C0和C2的所有分配结果，并将原来消费者C1的“负担”分配给了剩余的两个消费者C0和C2，最终C0和C2的分配还保持了均衡。

如果发生分区重分配，那么对于同一个分区而言有可能之前的消费者和新指派的消费者不是同一个，对于之前消费者进行到一半的处理还要在新指派的消费者中再次复现一遍，这显然很浪费系统资源。StickyAssignor策略如同其名称中的“sticky”一样，让分配策略具备一定的“粘性”，尽可能地让前后两次分配相同，进而减少系统资源的损耗以及其它异常情况的发生。

到目前为止所分析的都是消费者的订阅信息都是相同的情况，我们来看一下订阅信息不同的情况下的处理。

举例，同样消费组内有3个消费者：C0、C1和C2，集群中有3个主题：t0、t1和t2，这3个主题分别有1、2、3个分区，也就是说集群中有t0p0、t1p0、t1p1、t2p0、t2p1、t2p2这6个分区。消费者C0订阅了主题t0，消费者C1订阅了主题t0和t1，消费者C2订阅了主题t0、t1和t2。

如果此时采用RoundRobinAssignor策略，那么最终的分配结果如下所示（和讲述RoundRobinAssignor策略时的一样，这样不妨赘述一下）：

【分配结果集1】

消费者C0：t0p0

消费者C1：t1p0

消费者C2：t1p1、t2p0、t2p1、t2p2

如果此时采用的是StickyAssignor策略，那么最终的分配结果为：

【分配结果集2】

消费者C0：t0p0

消费者C1：t1p0、t1p1

消费者C2：t2p0、t2p1、t2p2

可以看到这是一个最优解（消费者C0没有订阅主题t1和t2，所以不能分配主题t1和t2中的任何分区给它，对于消费者C1也可同理推断）。  
假如此时消费者C0脱离了消费组，那么RoundRobinAssignor策略的分配结果为：

消费者C1：t0p0、t1p1

消费者C2：t1p0、t2p0、t2p1、t2p2

可以看到RoundRobinAssignor策略保留了消费者C1和C2中原有的3个分区的分配：t2p0、t2p1和t2p2（针对结果集1）。而如果采用的是StickyAssignor策略，那么分配结果为：

消费者C1：t1p0、t1p1、t0p0

消费者C2：t2p0、t2p1、t2p2

可以看到StickyAssignor策略保留了消费者C1和C2中原有的5个分区的分配：t1p0、t1p1、t2p0、t2p1、t2p2。

从结果上看StickyAssignor策略比另外两者分配策略而言显得更加的优异，这个策略的代码实现也是异常复杂，如果读者没有接触过这种分配策略，不妨使用一下来尝尝鲜。