分区副本的分配

上一节中多处提及了分区副本的分配，读者对此或许有点迷惑，在生产者和消费者中也都有分区分配的概念。生产者的分区分配是指为每条消息指定其所要发往的分区，消费者中的分区分配是指为消费者指定其可以消费消息的分区，而这里的分区分配是指为集群制定创建主题时的分区副本分配方案，即在哪个 broker 中创建哪些分区的副本。

在创建主题时，如果使用了 replica-assignment 参数，那么就按照指定的方案来进行分区副本的创建；如果没有使用 replica-assignment 参数，那么就需要按照内部的逻辑来计算分配方案了。使用 kafka-topics.sh 脚本创建主题时的内部分配逻辑按照机架信息划分成两种策略：未指定机架信息和指定机架信息。如果集群中所有的 broker 节点都没有配置 broker.rack 参数，或者使用 disable-rack-aware 参数来创建主题，那么采用的就是未指定机架信息的分配策略，否则采用的就是指定机架信息的分配策略。

首先看一下未指定机架信息的分配策略，具体的实现涉及代码的逻辑细节，未指定机架信息的分配策略比较容易理解，这里通过源码来逐一进行分析。所对应的具体实现为 kafka.admin.AdminUtils.scala 文件中的 assignReplicasToBrokersRackUnaware() 方法，该方法的内容如下：

```
private def assignReplicasToBrokersRackUnaware(
    nPartitions: Int,                  //分区数
    replicationFactor: Int,   //副本因子
    brokerList: Seq[Int],     //集群中broker列表
    fixedStartIndex: Int,     //起始索引，即第一个副本分配的位置，默认值为-1
    startPartitionId: Int):   //起始分区编号，默认值
```

```
为-1
Map[Int, Seq[Int]] = {
  val ret = mutable.Map[Int, Seq[Int]]() //保存分配
结果的集合
  val brokerArray = brokerList.toArray
//brokerId的列表
//如果起始索引fixedStartIndex小于0，则根据broker列表长
度随机生成一个，以此来保证是
//有效的brokerId
  val startIndex = if (fixedStartIndex >= 0)
fixedStartIndex
    else rand.nextInt(brokerArray.length)
  //确保起始分区号不小于0
  var currentPartitionId = math.max(0,
startPartitionId)
  //指定了副本的间隔，目的是为了更均匀地将副本分配到不同的
broker上
  var nextReplicaShift = if (fixedStartIndex >=
0) fixedStartIndex
    else rand.nextInt(brokerArray.length)
  //轮询所有分区，将每个分区的副本分配到不同的broker上
  for (_ <- 0 until nPartitions) {
    if (currentPartitionId > 0 &&
(currentPartitionId % brokerArray.length == 0))
      nextReplicaShift += 1
    val firstReplicaIndex = (currentPartitionId +
startIndex) % brokerArray.length
    val replicaBuffer =
mutable.ArrayBuffer(brokerArray(firstReplicaIndex
))
    //保存该分区所有副本分配的broker集合
    for (j <- 0 until replicationFactor - 1)
      replicaBuffer += brokerArray(
```

```
        replicaIndex(firstReplicaIndex,
nextReplicaShift,
          j, brokerArray.length)) //为其余的副本分配
broker
    //保存该分区所有副本的分配信息
    ret.put(currentPartitionId, replicaBuffer)
    //继续为下一个分区分配副本
    currentPartitionId += 1
  }
  ret
}
```

该方法参数列表中的 fixedStartIndex 和 startPartitionId 值是从上游的方法中调用传下来的，都是-1，分别表示第一个副本分配的位置和起始分区编号。assignReplicasToBrokersRackUnaware () 方法的核心是遍历每个分区 partition，然后从 brokerArray（brokerId的列表）中选取 replicationFactor 个 brokerId 分配给这个 partition。

该方法首先创建一个可变的 Map用来存放该方法将要返回的结果，即分区 partition 和分配副本的映射关系。由于 fixedStartIndex 为-1，所以 startIndex 是一个随机数，用来计算一个起始分配的 brokerId，同时又因为 startPartitionId 为-1，所以 currentPartitionId 的值为0，可见默认情况下创建主题时总是从编号为0的分区依次轮询进行分配。

nextReplicaShift 表示下一次副本分配相对于前一次分配的位移量，从字面上理解有点绕口。举个例子：假设集群中有3个 broker 节点，对应于代码中的 brokerArray，创建的某个主题中有3个副本和6个分区，那么首先从 partitionId（partition的编号）为0的分区开始进行分配，假设第一次计算（由 rand.nextInt(brokerArray.length)随机产生）得到的 nextReplicaShift 值为1，第一次随机产生的 startIndex 值为2，那

么 partitionId 为0的第一个副本的位置（这里指的是 brokerArray 的数组下标）firstReplicaIndex = (currentPartitionId + startIndex) % brokerArray.length= (0+2)%3=2，第二个副本的位置为 replicaIndex(firstReplicaIndex, nextReplicaShift, j, brokerArray. length) = replicaIndex(2, nextReplicaShift+1,0, 3)=?，这里引入了一个新的方法 replicaIndex()，不过这个方法很简单，具体如下：

```
private def replicaIndex(firstReplicaIndex: Int,
secondReplicaShift: Int,
                         replicaIndex: Int,
nBrokers: Int): Int = {
  val shift = 1 + (secondReplicaShift +
replicaIndex) % (nBrokers - 1)
  (firstReplicaIndex + shift) % nBrokers
}
```

继续计算 replicaIndex(2,nextReplicaShift+1,0,3) = replicaIndex(2,2,0,3) = (2+(1+(2+0) % (3-1))) % 3 = 0。继续计算下一个副本的位置 replicaIndex(2,2,1,3) = (2+(1+(2+1)%(3-1)))%3 = 1。所以 partitionId 为0的副本分配位置列表为[2,0,1]，如果 brokerArray 正好是从0开始编号的，也正好是顺序不间断的，即 brokerArray 为[0,1,2]，那么当前 partitionId 为0的副本分配策略为[2,0,1]。如果 brokerId 不是从0开始的，也不是顺序的（有可能之前集群的其中几个 broker 下线了），最终的 brokerArray 为[2,5,8]，那么 partitionId 为0的分区的副本分配策略为[8,2,5]。为了便于说明问题，可以简单假设 brokerArray 就是[0,1,2]。

同样计算下一个分区，即 partitionId 为1的副本分配策略。此时 nextReplicaShift 的值还是2，没有满足自增的条件。这个分区的 firstReplicaIndex=(1+2)%3=0。第二个副本的位置 replicaIndex(0,2,0,3) = (0+(1+(2+0)%(3-1)))%3 = 1，第三个副

本的位置 replicaIndex(0,2,1,3) = 2，最终 partitionId 为2的分区分配策略为[0,1,2]。
依次类推，更多的分配细节可以参考下面的示例，topic-test2 的分区分配策略和上面陈述的一致：

```
[root@node1 kafka_2.11-2.0.0]# bin/kafka-
topics.sh --zookeeper localhost:2181/ kafka --
create --topic topic-test2 --replication-factor 3
--partitions 6
Created topic "topic-test2".

[root@node1 kafka_2.11-2.0.0]# bin/kafka-
topics.sh --zookeeper localhost:2181/ kafka --
describe --topic topic-test2
Topic:topic-test2   PartitionCount:6
ReplicationFactor:3 Configs:
    Topic: topic-test2  Partition: 0    Leader: 2
Replicas: 2,0,1 Isr: 2,0,1
    Topic: topic-test2  Partition: 1    Leader: 0
Replicas: 0,1,2 Isr: 0,1,2
    Topic: topic-test2  Partition: 2    Leader: 1
Replicas: 1,2,0 Isr: 1,2,0
    Topic: topic-test2  Partition: 3    Leader: 2
Replicas: 2,1,0 Isr: 2,1,0
    Topic: topic-test2  Partition: 4    Leader: 0
Replicas: 0,2,1 Isr: 0,2,1
    Topic: topic-test2  Partition: 5    Leader: 1
Replicas: 1,0,2 Isr: 1,0,2
```

我们无法预先获知 startIndex 和 nextReplicaShift 的值，因为都是随机产生的。startIndex 和 nextReplicaShift 的值可以通过最终的分区分配方案来反推，比如上面的 topic-test2，第一个分区（即 partitionId=0 的分区）的第一个副本为2，那么可由2 =

(0+startIndex)%3推断出 startIndex 为2。之所以 startIndex 选择随机产生，是因为这样可以在多个主题的情况下尽可能地均匀分布分区副本，如果这里固定为一个特定值，那么每次的第一个副本都是在这个 broker 上，进而导致少数几个 broker 所分配到的分区副本过多而其余 broker 分配到的分区副本过少，最终导致负载不均衡。尤其是某些主题的副本数和分区数都比较少，甚至都为1的情况下，所有的副本都落到了那个指定的 broker 上。与此同时，在分配时位移量 nextReplicaShift 也可以更好地使分区副本分配得更加均匀。

相比较而言，指定机架信息的分配策略比未指定机架信息的分配策略要稍微复杂一些，但主体思想并没相差很多，只是将机架信息作为附加的参考项。假设目前有3个机架 rack1、rack2 和 rack3，Kafka 集群中的9个 broker 点都部署在这3个机架之上，机架与 broker 节点的对照关系如下：

```
rack1: 0, 1, 2
rack2: 3, 4, 5
rack3: 6, 7, 8
```

如果不考虑机架信息，那么对照 assignReplicasToBrokersRackUnaware() 方法里的 brokerArray 变量的值为[0, 1, 2, 3, 4, 5 6, 7, 8]。指定基架信息的 assignReplicasToBrokersRackAware() 方法里的 brokerArray 的值在这里就会被转换为[0, 3, 6, 1, 4, 7, 2, 5, 8]，显而易见，这是轮询各个机架而产生的结果，如此新的 brokerArray（确切地说是 arrangedBrokerList）中包含了简单的机架分配信息。之后的步骤也和 assignReplicasToBrokersRackUnaware() 方法类似，同样包含 startIndex、currentPartiionId、nextReplicaShift 的概念，循环为每一个分区分配副本。分配副本时，除了处理第一个副本，其余的也调用 replicaIndex() 方法来获得一个 broker，但这里和 assignReplicasToBrokersRackUnaware() 不同的是，这里不是简

单地将这个 broker 添加到当前分区的副本列表之中，还要经过一层筛选，满足以下任意一个条件的 broker 不能被添加到当前分区的副本列表之中：

- 如果此 broker 所在的机架中已经存在一个 broker 拥有该分区的副本，并且还有其他的机架中没有任何一个 broker 拥有该分区的副本。
- 如果此 broker 中已经拥有该分区的副本，并且还有其他 broker 中没有该分区的副本。

当创建一个主题时，无论通过 kafka-topics.sh 脚本，还是通过其他方式（比如第30节中介绍的 KafkaAdminClient）创建主题时，实质上是在 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，并且在/config/topics/节点下创建与该主题对应的子节点并写入主题相关的配置信息（这个步骤可以省略不执行）。而 Kafka 创建主题的实质性动作是交由控制器异步去完成的。

知道了 kafka-topics.sh 脚本的实质之后，我们可以直接使用 ZooKeeper 的客户端在/brokers/topics 节点下创建相应的主题节点并写入预先设定好的分配方案，这样就可以创建一个新的主题了。这种创建主题的方式还可以绕过一些原本使用 kafka-topics.sh 脚本创建主题时的一些限制，比如分区的序号可以不用从0开始连续累加了。首先我们通过 ZooKeeper 客户端创建一个除了与主题 topic-create 名称不同其余都相同的主题 topic-create-zk，示例如下：

```
[zk: localhost:2181/kafka(CONNECTED) 29] create
/brokers/topics/topic-create-zk
{"version":1,"partitions":{"2":[1,2],"1":
[0,1],"3":[2,1],"0":[2,0]}}
Created /brokers/topics/topic-create-zk
```

通过查看主题 topic-create-zk 的分配情况，可以看到与主题 topic-create 的信息没有什么差别。

```
[root@node1 kafka_2.11-2.0.0]# bin/kafka-
topics.sh --zookeeper localhost:2181/ kafka --
describe --topic topic-create-zk
Topic:topic-create-zk   PartitionCount:4
ReplicationFactor:2 Configs:
Topic: topic-create-zk  Partition: 0    Leader: 2
Replicas: 2,0 Isr: 2,0
        Topic: topic-create-zk  Partition: 1
Leader: 0 Replicas: 0,1 Isr: 0,1
        Topic: topic-create-zk  Partition: 2
Leader: 1 Replicas: 1,2 Isr: 1,2
        Topic: topic-create-zk  Partition: 3
Leader: 2 Replicas: 2,1 Isr: 2,1
```

我们再创建一个另类的主题，分配情况和主题 topic-create 一样，唯独分区号已经与主题 topic-create-special 大相径庭，示例如下：

```
[zk: localhost:2181/kafka (CONNECTED) 31] create
/brokers/topics/topic-create- special
{"version":1,"partitions":{"10":[1,2],"21":
[0,1],"33":[2,1],"40":[2,0]}}
Created /brokers/topics/topic-create-special

[root@node1 kafka_2.11-2.0.0]# bin/kafka-
topics.sh --zookeeper localhost:2181/ kafka --
describe --topic topic-create-special
Topic:topic-create-special  PartitionCount:4
ReplicationFactor:2 Configs:
    Topic: topic-create-special Partition: 10
Leader: 1 Replicas: 1,2   Isr: 1,2
    Topic: topic-create-special Partition: 21
Leader: 0 Replicas: 0,1   Isr: 0,1
    Topic: topic-create-special Partition: 33
Leader: 2 Replicas: 2,1   Isr: 2,1
    Topic: topic-create-special Partition: 40
Leader: 2 Replicas: 2,0   Isr: 2,0
```

可以看到分区号为10、21、33和40，而通过单纯地使用 kafka-topics.sh 脚本是无法实现的。不过这种方式也只是一些实战方面上的技巧，笔者还是建议使用更加正统的 kafka-topics.sh 脚本或 KafkaAdminClient 来管理相应的主题。