Flink的八种分区策略源码解读

Flink包含8中分区策略，这8中分区策略(分区器)分别如下面所示，本文将从源码的角度一一解读每个分区器的实现方式。

- GlobalPartitioner
- ShufflePartitioner
- RebalancePartitioner
- RescalePartitioner
- BroadcastPartitioner
- ForwardPartitioner
- KeyGroupStreamPartitioner
- CustomPartitionerWrapper

继承关系图

接口

名称

ChannelSelector

实现

```
public interface ChannelSelector<T extends IOReadableWritable> {

        /**
         * 初始化channels数量，channel可以理解为下游Operator的某个实例(并行算子的某个subtask).
         */
        void setup(int numberOfChannels);

        /**
         *根据当前的record以及Channel总数，
         *决定应将record发送到下游哪个Channel。
         *不同的分区策略会实现不同的该方法。
         */
        int selectChannel(T record);

        /**
        *是否以广播的形式发送到下游所有的算子实例
         */
        boolean isBroadcast();
}
```

抽象类

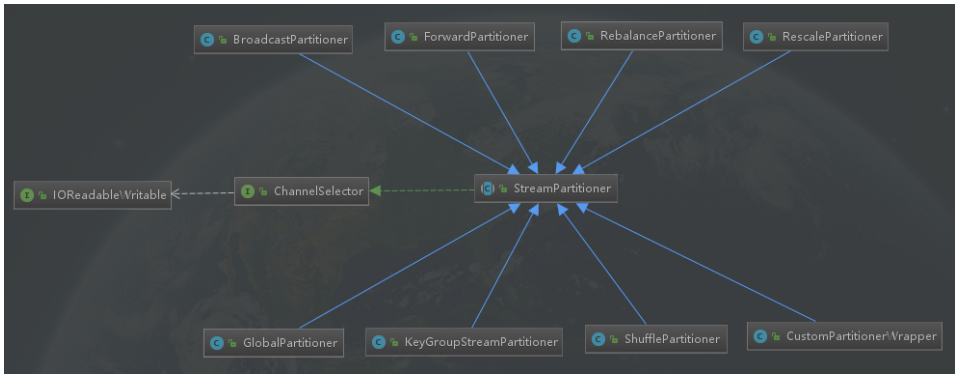
名称

StreamPartitioner

实现

```
public abstract class StreamPartitioner<T> implements
                ChannelSelector<SerializationDelegate<StreamRecord<T>>>, Serializable {
        private static final long serialVersionUID = 1L;

        protected int numberOfChannels;

        @Override
        public void setup(int numberOfChannels) {
                this.numberOfChannels = numberOfChannels;
        }

        @Override
        public boolean isBroadcast() {
                return false;
        }

        public abstract StreamPartitioner<T> copy();
}
```

继承关系图



GlobalPartitioner

简介

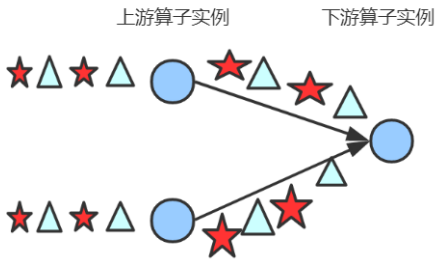
该分区器会将所有的数据都发送到下游的某个算子实例(subtask id = 0)

源码解读

```
/**
 * 发送所有的数据到下游算子的第一个task(ID = 0)
 * @param <T>
 */
```

```
@Internal
public class GlobalPartitioner<T> extends StreamPartitioner<T> {
        private static final long serialVersionUID = 1L;

        @Override
        public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
                //只返回0，即只发送给下游算子的第一个task
                return 0;
        }

        @Override
        public StreamPartitioner<T> copy() {
                return this;
        }

        @Override
        public String toString() {
                return "GLOBAL";
        }
}
```

图解



GlobalPartitioner分区器示意图

ShufflePartitioner

简介

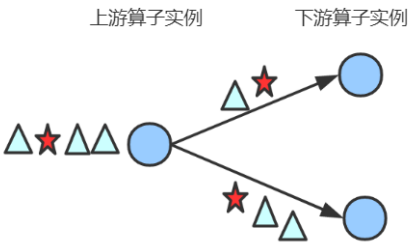
随机选择一个下游算子实例进行发送

源码解读

```
/**
 * 随机的选择一个channel进行发送
 * @param <T>
 */
@Internal
public class ShufflePartitioner<T> extends StreamPartitioner<T> {
        private static final long serialVersionUID = 1L;

        private Random random = new Random();

        @Override
        public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
                //产生[0,numberOfChannels)伪随机数，随机发送到下游的某个task
                return random.nextInt(numberOfChannels);
```

```
        }

        @Override
        public StreamPartitioner<T> copy() {
                return new ShufflePartitioner<T>();
        }

        @Override
        public String toString() {
                return "SHUFFLE";
        }
}
```

图解



ShufflePartitioner分区器示意图

BroadcastPartitioner

简介

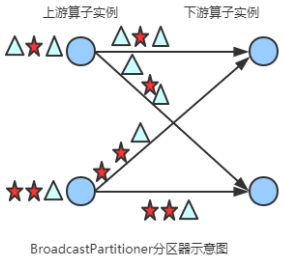
发送到下游所有的算子实例

源码解读

```
/**
 * 发送到所有的channel
 */
@Internal
public class BroadcastPartitioner<T> extends StreamPartitioner<T> {
        private static final long serialVersionUID = 1L;
        /**
         * Broadcast模式是直接发送到下游的所有task，所以不需要通过下面的方法选择发送的通道
         */
        @Override
        public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
                throw new UnsupportedOperationException("Broadcast partitioner does not
upport select channels.");
        }

        @Override
        public boolean isBroadcast() {
                return true;
        }

        @Override
        public StreamPartitioner<T> copy() {
```

```
                return this;
        }

        @Override
        public String toString() {
                return "BROADCAST";
        }
}
```

图解



BroadcastPartitioner分区器示意图

RebalancePartitioner

简介

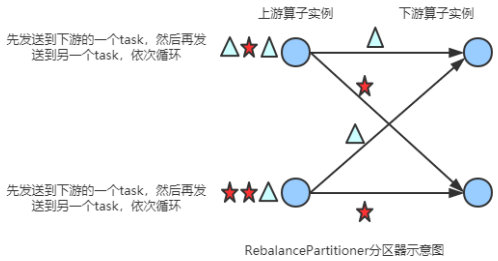
通过循环的方式依次发送到下游的task

源码解读

```
/**
 *通过循环的方式依次发送到下游的task
 * @param <T>
 */
@Internal
public class RebalancePartitioner<T> extends StreamPartitioner<T> {
        private static final long serialVersionUID = 1L;

        private int nextChannelToSendTo;

        @Override
        public void setup(int numberOfChannels) {
                super.setup(numberOfChannels);
                //初始化channel的id，返回[0,numberOfChannels)的伪随机数
                nextChannelToSendTo = ThreadLocalRandom.current().nextInt(numberOfChanne
s);
        }

        @Override
        public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
                //循环依次发送到下游的task，比如：nextChannelToSendTo初始值为0，numberOfChanne
s(下游算子的实例个数，并行度)值为2
                //则第一次发送到ID = 1的task，第二次发送到ID = 0的task，第三次发送到ID = 1的task
上...依次类推
                nextChannelToSendTo = (nextChannelToSendTo + 1) % numberOfChannels;
                return nextChannelToSendTo;
        }
```

```
        public StreamPartitioner<T> copy() {
                return this;
        }

        @Override
        public String toString() {
                return "REBALANCE";
        }
}
```

图解



RebalancePartitioner分区器示意图

RescalePartitioner

简介

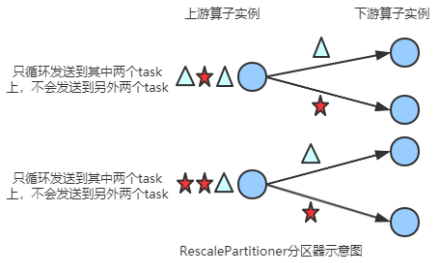
基于上下游Operator的并行度，将记录以循环的方式输出到下游Operator的每个实例。
举例: 上游并行度是2，下游是4，则上游一个并行度以循环的方式将记录输出到下游的两个并行度上;上游另一个并行度以循环的方式将记录输出到下游另两个并行度上。
若上游并行度是4，下游并行度是2，则上游两个并行度将记录输出到下游一个并行度上；上游另两个并行度将记录输出到下游另一个并行度上。

源码解读

```
@Internal
public class RescalePartitioner<T> extends StreamPartitioner<T> {
        private static final long serialVersionUID = 1L;

        private int nextChannelToSendTo = -1;

        @Override
        public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
                if (++nextChannelToSendTo >= numberOfChannels) {
                        nextChannelToSendTo = 0;
                }
                return nextChannelToSendTo;
        }

        public StreamPartitioner<T> copy() {
                return this;
        }

        @Override
```

```
        public String toString() {
                return "RESCALE";
        }
}
```

图解



尖叫提示

Flink 中的执行图可以分成四层：StreamGraph -> JobGraph -> ExecutionGraph -> 物理执行图。

StreamGraph：是根据用户通过 Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。

JobGraph：StreamGraph经过优化后生成了 JobGraph，提交给 JobManager 的数据结构。主要的优化为，将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化/反序列化/传输消耗。

ExecutionGraph：JobManager 根据 JobGraph 生成ExecutionGraph。ExecutionGraph是JobGraph的并行化版本，是调度层最核心的数据结构。

物理执行图：JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个TaskManager 上部署 Task 后形成的“图”，并不是一个具体的数据结构。

而 StreamingJobGraphGenerator 就是 StreamGraph 转换为 JobGraph。在这个类中，把ForwardPartitioner和RescalePartitioner列为POINTWISE分配模式，其他的为ALL_TO_ALL分配模式。代码如下：

```
if (partitioner instanceof ForwardPartitioner || partitioner instanceof RescalePartitior
r) {
                        jobEdge = downStreamVertex.connectNewDataSetAsInput(
                                headVertex,

                            // 上游算子(生产端)的实例(subtask)连接下游算子(消费端)的一个或者多个实
例(subtask)
                                DistributionPattern.POINTWISE,
                                resultPartitionType);
                } else {
                        jobEdge = downStreamVertex.connectNewDataSetAsInput(
                                headVertex,
                                // 上游算子(生产端)的实例(subtask)连接下游算子(消费端)的所有实例
subtask)
                                DistributionPattern.ALL_TO_ALL,
```

```
                                        resultPartitionType);
                }
```

ForwardPartitioner

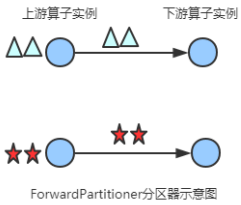
简介

发送到下游对应的第一个task，保证上下游算子并行度一致，即上有算子与下游算子是1:1的关系

源码解读

```
/**
 * 发送到下游对应的第一个task
 * @param <T>
 */
@Internal
public class ForwardPartitioner<T> extends StreamPartitioner<T> {
        private static final long serialVersionUID = 1L;

        @Override
        public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
                return 0;
        }

        public StreamPartitioner<T> copy() {
                return this;
        }

        @Override
        public String toString() {
                return "FORWARD";
        }
}
```

图解



ForwardPartitioner分区器示意图

尖叫提示

在上下游的算子没有指定分区器的情况下，如果上下游的算子并行度一致，则使用ForwardPartitioner，否则使用RebalancePartitioner，对于ForwardPartitioner，必须保证上下游算子并行度一致，否则会抛出异常

```
//在上下游的算子没有指定分区器的情况下，如果上下游的算子并行度一致，则使用ForwardPartitioner，否则使用
ebalancePartitioner
```

```
                        if (partitioner == null && upstreamNode.getParallelism() == down
treamNode.getParallelism()) {
                                partitioner = new ForwardPartitioner<Object>();
                        } else if (partitioner == null) {
                                partitioner = new RebalancePartitioner<Object>();
                        }

                        if (partitioner instanceof ForwardPartitioner) {
                                //如果上下游的并行度不一致，会抛出异常
                                if (upstreamNode.getParallelism() != downstreamNode.getP
rallelism()) {
                                        throw new UnsupportedOperationException("Forward
partitioning does not allow " +
                                                        "change of parallelism. Upstream operati
n: " + upstreamNode + " parallelism: " + upstreamNode.getParallelism() +
                                                        ", downstream operation: " + downstreamN
de + " parallelism: " + downstreamNode.getParallelism() +
                                                        " You must use another partitioning stra
egy, such as broadcast, rebalance, shuffle or global.");
                                }
                        }
```

KeyGroupStreamPartitioner

简介

根据key的分组索引选择发送到相对应的下游subtask

源码解读

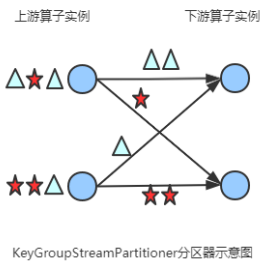
- org.apache.flink.streaming.runtime.partitioner.KeyGroupStreamPartitioner

```
/**
 * 根据key的分组索引选择发送到相对应的下游subtask
 * @param <T>
 * @param <K>
 */
@Internal
public class KeyGroupStreamPartitioner<T, K> extends StreamPartitioner<T> implements Con
igurableStreamPartitioner {
...

        @Override
        public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
                K key;
                try {
                        key = keySelector.getKey(record.getInstance().getValue());
                } catch (Exception e) {
                        throw new RuntimeException("Could not extract key from " + recor
.getInstance().getValue(), e);
                }
                //调用KeyGroupRangeAssignment类的assignKeyToParallelOperator方法,代码如下所
                return KeyGroupRangeAssignment.assignKeyToParallelOperator(key, maxParal
elism, numberOfChannels);
        }
...
}
```

- org.apache.flink.runtime.state.KeyGroupRangeAssignment

```
public final class KeyGroupRangeAssignment {
...

        /**
         * 根据key分配一个并行算子实例的索引，该索引即为该key要发送的下游算子实例的路由信息，
         * 即该key发送到哪一个task
         */
        public static int assignKeyToParallelOperator(Object key, int maxParallelism, in
 parallelism) {
                Preconditions.checkNotNull(key, "Assigned key must not be null!");
                return computeOperatorIndexForKeyGroup(maxParallelism, parallelism, assi
nToKeyGroup(key, maxParallelism));
        }

        /**
         *根据key分配一个分组id(keyGroupId)
         */
        public static int assignToKeyGroup(Object key, int maxParallelism) {
                Preconditions.checkNotNull(key, "Assigned key must not be null!");
                //获取key的hashcode
                return computeKeyGroupForKeyHash(key.hashCode(), maxParallelism);
        }

        /**
         * 根据key分配一个分组id(keyGroupId),
         */
        public static int computeKeyGroupForKeyHash(int keyHash, int maxParallelism) {

                //与maxParallelism取余，获取keyGroupId
                return MathUtils.murmurHash(keyHash) % maxParallelism;
        }

        //计算分区index，即该key group应该发送到下游的哪一个算子实例
        public static int computeOperatorIndexForKeyGroup(int maxParallelism, int parall
lism, int keyGroupId) {
                return keyGroupId * parallelism / maxParallelism;
        }
...
```

图解



KeyGroupStreamPartitioner分区器示意图

CustomPartitionerWrapper

简介

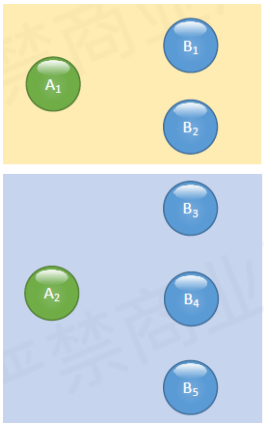
通过 `Partitioner` 实例的 `partition` 方法(自定义的)将记录输出到下游。

```
public class CustomPartitionerWrapper<K, T> extends StreamPartitioner<T> {
        private static final long serialVersionUID = 1L;

        Partitioner<K> partitioner;
        KeySelector<T, K> keySelector;

        public CustomPartitionerWrapper(Partitioner<K> partitioner, KeySelector<T, K> ke
Selector) {
                this.partitioner = partitioner;
                this.keySelector = keySelector;
        }

        @Override
        public int selectChannel(SerializationDelegate<StreamRecord<T>> record) {
                K key;
                try {
                        key = keySelector.getKey(record.getInstance().getValue());
                } catch (Exception e) {
                        throw new RuntimeException("Could not extract key from " + recor
.getInstance(), e);
                }
//实现Partitioner接口，重写partition方法
                return partitioner.partition(key, numberOfChannels);
        }

        @Override
        public StreamPartitioner<T> copy() {
                return this;
        }

        @Override
        public String toString() {
                return "CUSTOM";
        }
}
```

比如：

```
public class CustomPartitioner implements Partitioner<String> {
      // key: 根据key的值来分区
      // numPartitions: 下游算子并行度
          @Override
      public int partition(String key, int numPartitions) {
         return key.length() % numPartitions;//在此处定义分区策略
      }
  }
```

总结



类型	描述
dataStream.global();	全部发往第1个task
dataStream.broadcast();	广播
dataStream.forward();	上下游并发度一样时一对一发送
dataStream.shuffle();	随机均匀分配
dataStream.rebalance();	Round-Robin（轮流分配）
dataStream.recale();	Local Round-Robin（本地轮流分配）
dataStream.partitionCustom()	自定义单播