Flink数据流图的生成----JobGraph的生成

该部分的内容是StreamGraph怎么生成JobGraph. JobGraph是StreamGraph经过优化后生成的并且提交给JobManager 的数据结构。其具体包括如下几种属性：

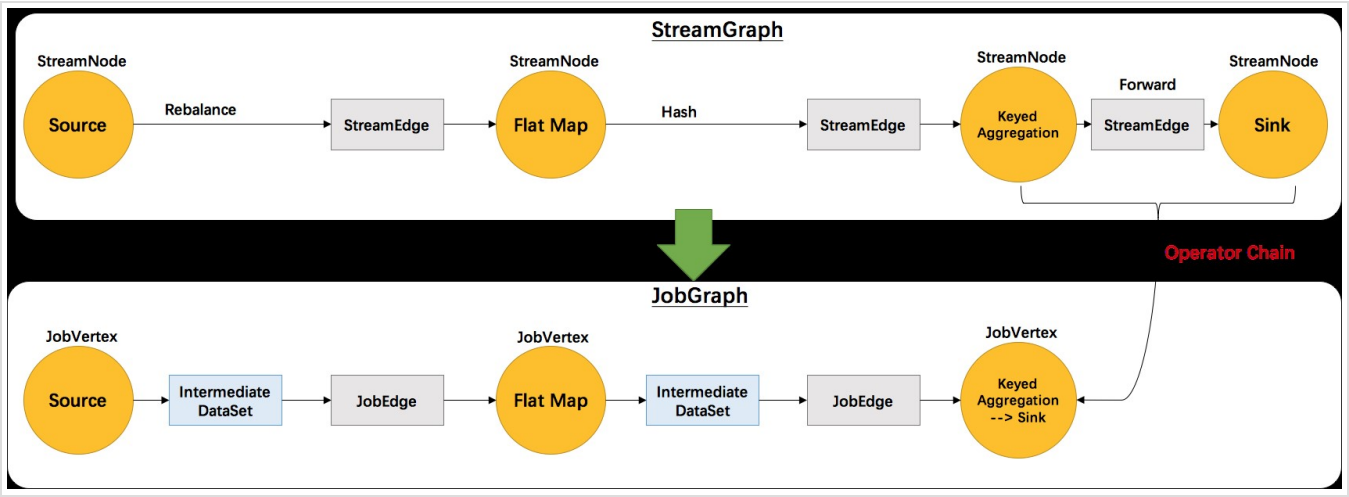
- JobVertex：经过优化后符合条件的多个StreamNode可能会chain在一起生成一个JobVertex，即一个JobVertex包含一个或多个operator，JobVertex的输入是JobEdge，输出是IntermediateDataSet。
- IntermediateDataSet：表示JobVertex的输出，即经过operator处理产生的数据集。producer是JobVertex，consumer是JobEdge。
- JobEdge：代表了job graph中的一条数据传输通道。source是IntermediateDataSet，target是JobVertex。即数据通过JobEdge由IntermediateDataSet传递给目标JobVertex。

```
/**
        JobVertex
*/
private final JobVertexID id;
private final ArrayList<OperatorID> operatorIDs = new ArrayList<>();
private final ArrayList<IntermediateDataSet> results = new ArrayList<IntermediateDat
private final ArrayList<JobEdge> inputs = new ArrayList<JobEdge>();
private int parallelism = ExecutionConfig.PARALLELISM_DEFAULT;
private Configuration configuration;
private String invokableClassName;
private String name;
private String operatorName;
private String operatorDescription;

/**
        JobEdge
*/
private final JobVertex target;
private final DistributionPattern distributionPattern;
private IntermediateDataSet source;
private IntermediateDataSetID sourceId;
private String preProcessingOperationName;

/**
        IntermediateDataSet
*/
private final IntermediateDataSetID id;                 // the identifier
private final JobVertex producer;                       // the operation that produc
private final List<JobEdge> consumers = new ArrayList<JobEdge>();
// The type of partition to use at runtime
private final ResultPartitionType resultType;
addConsumer(JobEdge edge)
```

对于SocketWindowWordCount.java而言，由StreamGraph生成JobGraph的过程如下：



原理分析

上图展示了JobGraph的生成过程，JobGraph是由JobVertex、IntermediateDataSet和JobEdge三部分组成。整个JobGraph的形成过程是首先会根据生成的StreamGraph来获取到所有的StreamNode，然后倒序对每一个StreamNode进行遍历操作，从而形成整个图。该图是一个StreamGraph的优化，优化的部分就是尽可能的将operator的subtask链接在一起，形成一个task，每个task在一个线程中执行。这是一个非常有效的优化，它能够减少线程之间的切换，减少消息的序列化和反序列化，减少数据在缓冲区中的交换，减少了延迟同时提高了系统的吞吐量。

构建JobGraph图时采用倒序遍历的方式，首先判断sink是否是可以链接的，该判断是一个重要的过程，满足的链接的条件大致为以下几条：1、上下游的并行度一致。2、下游的入度为1。3、下游的chain策略是ALWAYS。4、上游的chain策略是ALWAYS或HEAD。5、两个节点之间的数据分区策略是Forward等。该operator正好满足可链接的条件，然后就会把该StreamNode的信息，包括名称，id等序列化到StreamConfig中，

源码分析

JobGraph 的相关数据结构主要在`org.apache.flink.runtime.jobgraph`包中。构造 JobGraph 的代码主要集中在`StreamingJobGraphGenerator`类中，入口函数是`StreamingJobGraphGenerator.createJobGraph()`。源码如下：

```
public class StreamingJobGraphGenerator {
  // 根据 StreamGraph，生成 JobGraph
  public JobGraph createJobGraph() {
    Map<Integer, byte[]> hashes = traverseStreamGraphAndGenerateHashes();
    // 最重要的函数，生成JobVertex，JobEdge等，并尽可能地将多个节点chain在一起
    setChaining(hashes);
    // 将每个JobVertex的入边集合也序列化到该JobVertex的StreamConfig中
    // （出边集合已经在setChaining的时候写入了）
    setPhysicalEdges();
    // 根据group name，为每个 JobVertex 指定所属的 SlotSharingGroup
    // 以及针对 Iteration的头尾设置  CoLocationGroup
    return jobGraph;
  }
```

```
  ...
}
```

该函数就是生成的JobGraph的函数，其内部有几个重要的方法：

> traverseStreamGraphAndGenerateHashes()：给StreamGraph的每个StreamNode生成一个唯一的hash值，该hash值在节点不发生改变的情况下多次生成始终是一致的，可用来判断节点在多次提交时是否产生了变化并且该值也将作为JobVertex的ID。

> setChaining()：基于StreamGraph从所有的source开始构建task chain

> setPhysicalEdges()：建立目标节点的入边的连接

对于traverseStreamGraphAndGenerateHashes()，源码如下：

```
public Map<Integer, byte[]> traverseStreamGraphAndGenerateHashes(StreamGraph streamG
    HashMap<Integer, byte[]> hashResult = new HashMap<>();
    for (StreamNode streamNode : streamGraph.getStreamNodes()) {
        String userHash = streamNode.getUserHash();
        if (null != userHash) {
            hashResult.put(streamNode.getId(), StringUtils.hexStringToByte(userHash)
        }
    }
    return hashResult;
}
```

traverseStreamGraphAndGenerateHashes()该函数会为每一个StreamNode生成对应的hash值。下面是setChaining()，该函数是对SteamGraph的一个优化，把符合相关条件的operator链接在一起，组成operator chain，在调度时这些被链接到一起的operator会被视为一个任务(Task)。SetChaining()的源码如下：

```
private void setChaining(Map<Integer, byte[]> hashes, List<Map<Integer, byte[]>> lega
    for (Integer sourceNodeId : streamGraph.getSourceIDs()) {
        createChain(sourceNodeId, sourceNodeId, hashes, legacyHashes, 0, chainedOpera
    }
}
```

setChaining会遍历StreamGraph中的sourceID集合，然后每个source会调用createChain方法，该方法以当前source为起点向后遍历并创建operator chain。首先createChain会分析当前节点的出边，调用isChainable()函数并且根据Operator Chain中的chainable条件，将出边分成chainalbe和noChainable两类。具体的条件为：

```
return downStreamVertex.getInEdges().size() == 1          //如果边的下游流节点的入边数目为
        && outOperator != null                            //边的下游节点对应的算子不为nul
        && headOperator != null                           //边的上游节点对应的算子不为nul
        && upStreamVertex.isSameSlotSharingGroup(downStreamVertex)      //边两端节点有
        && outOperator.getChainingStrategy() == ChainingStrategy.ALWAYS //边下游算子的
        && (headOperator.getChainingStrategy() == ChainingStrategy.HEAD ||
        headOperator.getChainingStrategy() == ChainingStrategy.ALWAYS)//上游算子的链接
```

```
        && (edge.getPartitioner() instanceof ForwardPartitioner)      //边的分区器类型
        && upStreamVertex.getParallelism() == downStreamVertex.getParallelism()   //
        && streamGraph.isChainingEnabled();        //当前的streamGraph允许链接的
```

然后递归调用createChain方法，从而构建出node chains，

```
for (StreamEdge chainable : chainableOutputs) {
transitiveOutEdges.addAll()
createChain(startNodeId, chainable.getTargetId(), hashes, legacyHashes, chainIndex +
        for (StreamEdge nonChainable : nonChainableOutputs) {
                transitiveOutEdges.add(nonChainable);
createChain(nonChainable.getTargetId(), nonChainable.getTargetId(), hashes, legacyHas

```

这两种情况下都会调用createChain()函数，该函数内的createJobVertex为链接头节点或者无法链接的节点创建JobVertex对象，就是对应着StreamGraph中的StreamNode，创建完成之后会返回一个空的StreamConfig。

```
chainedNames.put(currentNodeId, createChainedName(currentNodeId, chainableOutputs));
```

接着会判断当前节点是不是chain中的头节点，如果当前是chain的头节点，则会调用createJobVertex 函数来根据StreamNode 创建对应的 JobVertex, 并返回空的 StreamConfig。如果当前不是chain的头节点，则会将StreamConfig添加到该chain的config集合中。

```
StreamConfig config = currentNodeId.equals(startNodeId)
                                        ? createJobVertex(startNodeId, hashes, legac
                                        : new StreamConfig(new Configuration());
if (currentNodeId.equals(startNodeId)) {
config.setChainStart();config.setChainIndex(0);
    config.setOperatorName(streamGraph.getStreamNode(currentNodeId).getOperatorName(
config.setOutEdgesInOrder(transitiveOutEdges);
        config.setOutEdges(streamGraph.getStreamNode(currentNodeId).getOutEdges());
        for (StreamEdge edge : transitiveOutEdges) {
                connect(startNodeId, edge);
        }
        config.setTransitiveChainedTaskConfigs(chainedConfigs.get(startNodeId));
} else {
        Map<Integer, StreamConfig> chainedConfs = chainedConfigs.get(startNodeId);
if (chainedConfs == null) {
                chainedConfigs.put(startNodeId, new HashMap<Integer, StreamConfig>()
        }
        config.setChainIndex(chainIndex);
}
```

这里对于一个node chains，除了chain的头节点会生成对应的 JobVertex，其余的nodes都是以序列化的形式写入到StreamConfig中，并保存到chain 头结点的 CHAINED_TASK_CONFIG 配置项中。直到部署时，才会取出并生成对应的ChainOperators。当前节点是chain的头节点的话，会调用connect方法，根据StreamEdge创建JobEdge与IntermediateDataSet，然后根据出边找到下游的JobVertex，这样就会建立起当前JobVertex、IntermediateDataSet、JobEdge三者之间的关系。

```
private void connect(Integer headOfChain, StreamEdge edge) {

                StreamPartitioner<?> partitioner = edge.getPartitioner();
                JobEdge jobEdge;
                if (partitioner instanceof ForwardPartitioner) {
                        jobEdge = downStreamVertex.connectNewDataSetAsInput(
                                headVertex,
                                DistributionPattern.POINTWISE,
                                ResultPartitionType.PIPELINED_BOUNDED);
                } else if (partitioner instanceof RescalePartitioner){
                . . . . . .
        }
public JobEdge connectNewDataSetAsInput(. . . . .) {
                IntermediateDataSet dataSet = input.createAndAddResultDataSet(partit
                JobEdge edge = new JobEdge(dataSet, this, distPattern);
                this.inputs.add(edge);
                dataSet.addConsumer(edge);
                return edge;
        }
```

这样setChaining()方法执行完毕，回到createJobGraph()方法，通过setChaining()方法，完成了JobVertex->IntermediateDataSet的连接，下面会调用setPhysicalEdges()方法来建立入边的连接，即`IntermediateDataSet-->JobVertex`。该函数会通过以下命令将每个JobVertex的入边集合也序列化到该JobVertex的StreamConfig中.

```
vertexConfigs.get(vertex).setInPhysicalEdges(edgeList);
```

然后会调用createJobGraph()方法里的一些其他配置信息，到此JobGraph就形成完毕了

总结

由StreamGraph生成JobGraph做的最大的优化就是operator chain，尽可能让满足条件的多个operator形成一个operator chain。

JobGraph生成原理 链接: https://pan.baidu.com/s/1WW2RThBM_05YeNWXi9NTrg 提取码: qqdn

JobGraph源码分析 链接: https://pan.baidu.com/s/15F4IyGmGm5WlqxIWcmz7Uw 提取码: 19fk