
toc: true title: 《从1到100深入学习Flink》——如何获取 JobGraph? date: 2019-02-23 tags:

- Flink
 - 大数据
 - 流式计算
-

前言

在上一篇文章中 《从1到100深入学习Flink》——如何获取 StreamGraph? 中获取到 StreamGraph 后, 这篇文章我们分析如何获取 JobGraph?

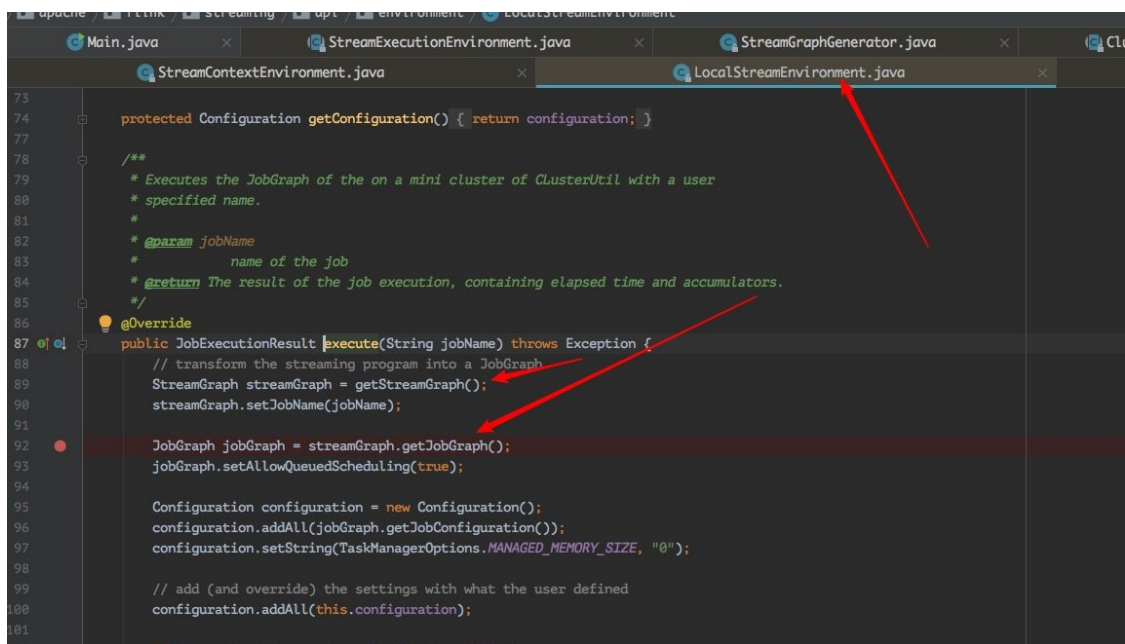
在了解如何获取 JobGraph 之前我们还是老规矩, 先来了解一下 JobGraph 是啥?

JobGraph

```
/**
 * The JobGraph represents a Flink dataflow program, at the low level that the JobManager accepts.
 * All programs from higher level APIs are transformed into JobGraphs.
 *
 * <p>The JobGraph is a graph of vertices and intermediate results that are connected together to
 * form a DAG. Note that iterations (feedback edges) are currently not encoded inside the JobGraph
 * but inside certain special vertices that establish the feedback channel amongst themselves.
 *
 * <p>The JobGraph defines the job-wide configuration settings, while each vertex and intermediate result
 * define the characteristics of the concrete operation and intermediate data.
 */
public class JobGraph implements Serializable {
    private static final long serialVersionUID = 1L;

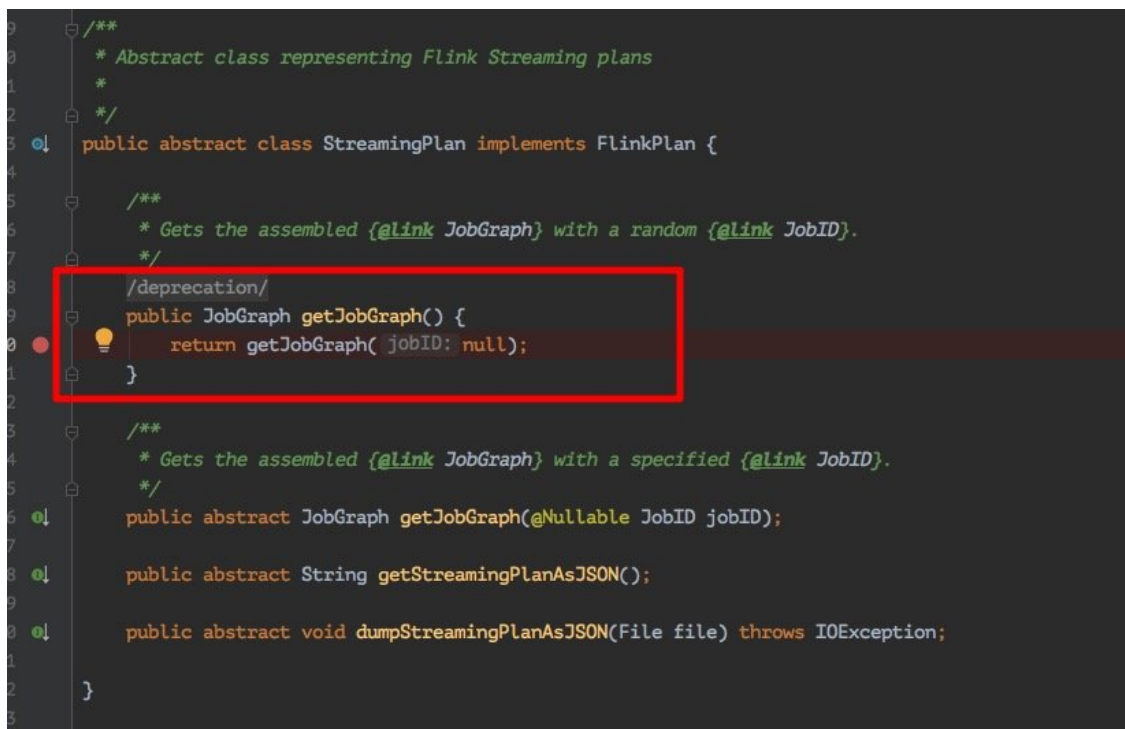
    // ... job and configuration ...
}
```

StreamGraph 如何转换成 JobGraph?



```
73 protected Configuration getConfiguration() { return configuration; }
74
77
78 /**
79  * Executes the JobGraph of the on a mini cluster of ClusterUtil with a user
80  * specified name.
81  *
82  * @param jobName
83  *       name of the job
84  * @return The result of the job execution, containing elapsed time and accumulators.
85  */
86
87 @Override
88 public JobExecutionResult execute(String jobName) throws Exception {
89     // transform the streaming program into a JobGraph
90     StreamGraph streamGraph = getStreamGraph();
91     streamGraph.setJobName(jobName);
92
93     JobGraph jobGraph = streamGraph.getJobGraph();
94     jobGraph.setAllowQueuedScheduling(true);
95
96     Configuration configuration = new Configuration();
97     configuration.addAll(jobGraph.getJobConfiguration());
98     configuration.setString(TaskManagerOptions.MANAGED_MEMORY_SIZE, "0");
99
100     // add (and override) the settings with what the user defined
101     configuration.addAll(this.configuration);
102 }
```

从这里我们看到的是直接 `streamGraph.getJobGraph()` 就获取到了 `JobGraph` 了，但是里面的细节却还很多，我们可以深入探究一下。



```
9 /**
10  * Abstract class representing Flink Streaming plans
11  *
12  */
13 public abstract class StreamingPlan implements FlinkPlan {
14
15     /**
16      * Gets the assembled {@link JobGraph} with a random {@link JobID}.
17      */
18     /**deprecation/
19     public JobGraph getJobGraph() {
20         return getJobGraph(jobID: null);
21     }
22
23     /**
24      * Gets the assembled {@link JobGraph} with a specified {@link JobID}.
25      */
26     public abstract JobGraph getJobGraph(@Nullable JobID jobID);
27
28     public abstract String getStreamingPlanAsJSON();
29
30     public abstract void dumpStreamingPlanAsJSON(File file) throws IOException;
31
32 }
33 }
```

点进来会发现是一个 deprecation 的方法，并且这个类还是抽象的，那么我们找下他的子类是否有该方法的具体实现呢？

```

public JobGraph getJobGraph(@Nullable JobID jobID) {
    // temporarily forbid checkpointing for iterative jobs
    if (isIterative() && checkpointConfig.isCheckpointingEnabled()
        && !checkpointConfig.isForceCheckpointing()) {
        throw new UnsupportedOperationException(
            "Checkpointing is currently not supported by default
            for iterative jobs, as we cannot guarantee exactly once semantics.
            "
            + "State checkpoints happen normally, but records
            in-transit during the snapshot will be lost upon failure. "
            + "\nThe user can force enable state checkpoints
            with the reduced guarantees by calling:
            env.enableCheckpointing(interval,true)");
    }

    return StreamingJobGraphGenerator.createJobGraph(this, jobID);
}

```

入口是 `getJobGraph` 方法，该方法先检查 job 是不是 iterative 类型，禁止 iterative job 的 checkpoint，然后调用 `StreamingJobGraphGenerator` 的静态方法 `createJobGraph` 创建 `JobGraph`。

在 `createJobGraph` 方法内部实现如下：

```

public static JobGraph createJobGraph(StreamGraph streamGraph,
    @Nullable JobID jobID) {
    return new StreamingJobGraphGenerator(streamGraph,
        jobID).createJobGraph();
}

```

这个方法里面先是初始化了一个 `StreamingJobGraphGenerator` 的实例，`StreamingJobGraphGenerator` 构造函数是私有的，只能通过这里进行实例构造，构造函数中就是做了一些基本的初始化的工作，并初始化了一个 `JobGraph` 实例，然后调用内部的私有方法 `createJobGraph()`。

```

private StreamingJobGraphGenerator(StreamGraph streamGraph,
@Nullable JobID jobID) {
    this.streamGraph = streamGraph;
    this.defaultStreamGraphHasher = new StreamGraphHasherV2();
    this.legacyStreamGraphHashers = Arrays.asList(new
StreamGraphUserHashHasher());

    this.jobVertices = new HashMap<>();
    this.builtVertices = new HashSet<>();
    this.chainedConfigs = new HashMap<>();
    this.vertexConfigs = new HashMap<>();
    this.chainedNames = new HashMap<>();
    this.chainedMinResources = new HashMap<>();
    this.chainedPreferredResources = new HashMap<>();
    this.physicalEdgesInOrder = new ArrayList<>();
    // 初始化了一个 JobGraph 实例
    jobGraph = new JobGraph(jobID, streamGraph.getJobName());
}

```

createJobGraph() 方法就是 JobGraph 进行配置的主要逻辑，如下：

```

private JobGraph createJobGraph() {

    // 设置调度模式，采用的EAGER模式，既所有节点都是立即启动的
    jobGraph.setScheduleMode(ScheduleMode.EAGER);

    /** 第一步 */
    /**
     * 1.1
     * 广度优先遍历StreamGraph，并且为每个StreamNode生成散列值，这里的散列值产生算法，可以保证如果提交的拓扑没有改变，则每次生成的散列值都是一样的。
     * 一个StreamNode的ID对应一个散列值。
     */
    Map<Integer, byte[]> hashes =
defaultStreamGraphHasher.traverseStreamGraphAndGenerateHashes(stream
mGraph);

    // 为向后兼容性生成遗留版本散列
    List<Map<Integer, byte[]>> legacyHashes = new ArrayList<>
(legacyStreamGraphHashers.size());
    for (StreamGraphHasher hasher : legacyStreamGraphHashers) {

        legacyHashes.add(hasher.traverseStreamGraphAndGenerateHashes(stream
Graph));
    }
}

```

```

// 相连接的操作符的散列值对
    Map<Integer, List<Tuple2<byte[], byte[]>>>
    chainedOperatorHashes = new HashMap<>();

//2. 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, legacyHashes, chainedOperatorHashes);

//3. 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();

//4. 据group name, 为每个 JobVertex 指定所属的 SlotSharingGroup 以及针对
    Iteration的头尾设置 CoLocationGroup
    setSlotSharingAndCoLocation();

//5. 配置checkpoint
    configureCheckpointing();

JobGraphGenerator.addUserArtifactEntries(streamGraph.getEnvironment
().getCachedFiles(), jobGraph);

    // 设置ExecutionConfig
    try {

jobGraph.setExecutionConfig(streamGraph.getExecutionConfig());
    }
    catch (IOException e) {
        throw new IllegalConfigurationException("Could not
serialize the ExecutionConfig." +
        "This indicates that non-serializable types (like
custom serializers) were registered");
    }

    return jobGraph;
}

```

jobGraph的整个产生过程就如上所示, 接下来针对其中的主要步骤进行简单分析。

第一步: 为每个节点产生散列值

这里就是根据StreamGraph的配置，给StreamGraph中的每个StreamNode产生一个长度为16的字节数组的散列值，这个散列值是用来后续生成JobGraph中对应的JobVertex的ID。在Flink中，任务存在从checkpoint中进行状态恢复的场景，而在恢复时，是以JobVertexID为依据的，所有就需要任务在重启的过程中，对于相同的任务，其各JobVertexID能够保持不变，而StreamGraph中各个StreamNode的ID，就是其包含的StreamTransformation的ID，而StreamTransformation的ID是在对数据流中的数据进行转换的过程中，通过一个静态的累加器生成的，比如有多个数据源时，每个数据源添加的顺序不一致，则有可能导致相同数据处理逻辑的任务，就会对应于不同的ID，所以为了得到确定的ID，在进行JobVertexID的产生时，需要以一种确定的方式来确定其值，要么是通过用户为每个ID直接指定对应的一个散列值，要么参考StreamGraph中的一些特征，为每个JobVertex产生一个确定的ID。

defaultStreamGraphHasher是在StreamingJobGraphGenerator构造函数中初始化的，其对应StreamGraphHasherV2的实例，这个类就是负责给StreamGraph中的每个StreamNode产生一个确定的散列值，这里主要介绍下其在产生一个StreamNode时，主要考虑的因素。（最好是结合着具体的代码看这段逻辑，会更清晰）

如果用户对节点指定了一个散列值，则基于用户指定的值，产生一个长度为16的字节数组；如果用户没有指定，则根据当前节点所处的位置，产生一个散列值，考虑的因素有：

- a、在当前StreamNode之前已经处理过的节点的个数，作为当前StreamNode的id，添加到hasher中；
- b、遍历当前StreamNode输出的每个StreamEdge，并判断当前StreamNode与这个StreamEdge的目标StreamNode是否可以链接，如果可以，则将目标StreamNode的id也放入hasher中，且这个目标StreamNode的id与当前StreamNode的id取相同的值；
- c、将上述步骤后产生的字节数据，与当前StreamNode的所有输入StreamNode对应的字节数据，进行相应的位操作，最终得到的字节数据，就是当前StreamNode对应的长度为16的字节数组。

另外在StreamingJobGraphGenerator的构造函数中，legacyStreamGraphHashers这个数组中，默认添加一个StreamGraphHasher的子类实现StreamGraphUserHashHasher。所以在上述的代码中，1.2步骤就是执行StreamGraphUserHashHasher这个类的逻辑。这个类的逻辑很简单，就是判断用户是否设置了散列值，如果设置了，就为对应的StreamNode产生一个散列值数组。

这里涉及到两个用户设置的散列值，StreamingJobGraphGenerator中使用的是StreamTransformation的uid属性，StreamGraphUserHashHasher使用的是StreamTransformation的userProvidedNodeHash属性。这两个属性解析如下：

uid —— 这个字段是用户设置的，用来在任务重启时，保障JobVertexID一致，一般是从之前的任务日志中，找出对应的值而设置的；

userProvidedNodeHash —— 这个字段也是用户设置的，设置的用户自己产生的散列值。

第二步、设置执行链

执行链的设置，就是从数据源StreamNode，依次遍历，如下：

```
private void setChaining(Map<Integer, byte[]> hashes,
    List<Map<Integer, byte[]>> legacyHashes, Map<Integer,
    List<Tuple2<byte[], byte[]>>> chainedOperatorHashes) {
    for (Integer sourceNodeId : streamGraph.getSourceIDs()) {
        createChain(sourceNodeId, sourceNodeId, hashes,
            legacyHashes, 0, chainedOperatorHashes);
    }
}
```

三个入参：

1、hashes和legacyHashes就是上面产生的每个StreamNode的ID对应的散列字节数组。

chainedOperatorHashes是一个map:

其key是顺序链接在一起的StreamNode的起始那个StreamNode的ID，比如source->flatMap这个两个对应的StreamNode，在这个例子中，key的值就是source对应的id，为1；

value是一个列表，包含了这个链上的所有操作符的散列值；

这个列表中的每个元素是一个二元组，这个列表的值就是[{source的主hash, source的备用hash_1}, {source的主hash, source的备用hash_2}, {flatMap的主hash, flatMap的备用hash_1}, ...}，对于这里的例子，列表中只有二个元素，为[{source的主hash, null}, {flatMap的主hash, null}]

在 setChaining 方法内部循环的调用 createChain 方法，我们先看一下该方法中的 isChainable 方法，这个方里面的逻辑是判断两个 StreamNode 是否可以链接到一起执行，如下：

```
public static boolean isChainable(StreamEdge edge, StreamGraph
streamGraph) {
    // 获取StreamEdge的源和目标StreamNode
    StreamNode upStreamVertex = edge.getSourceVertex();
    StreamNode downStreamVertex = edge.getTargetVertex();

    // 获取源和目标StreamNode中的StreamOperator
    StreamOperator<?> headOperator = upStreamVertex.getOperator();
    StreamOperator<?> outOperator = downStreamVertex.getOperator();

    /**
     * 1、下游节点只有一个输入
     * 2、下游节点的操作符不为null
     * 3、上游节点的操作符不为null
     * 4、上下游节点在一个槽位共享组内
     * 5、下游节点的连接策略是 ALWAYS
     * 6、上游节点的连接策略是 HEAD 或者 ALWAYS
     * 7、edge 的分区函数是 ForwardPartitioner 的实例
     * 8、上下游节点的并行度相等
     * 9、可以进行节点连接操作
     */
    return downStreamVertex.getInEdges().size() == 1
        && outOperator != null
        && headOperator != null
        &&
        upStreamVertex.isSameSlotSharingGroup(downStreamVertex)
        && outOperator.getChainingStrategy() ==
        ChainingStrategy.ALWAYS
        && (headOperator.getChainingStrategy() ==
        ChainingStrategy.HEAD ||
            headOperator.getChainingStrategy() ==
        ChainingStrategy.ALWAYS)
        && (edge.getPartitioner() instanceof
        ForwardPartitioner)
        && upStreamVertex.getParallelism() ==
        downStreamVertex.getParallelism()
        && streamGraph.isChainingEnabled();
}
```

只有上述的9个条件都同时满足时，才能说明两个StreamEdge的源和目标StreamNode是可以链接在一起执行的。


```

private List<StreamEdge> createChain(Integer startNodeId,Integer
currentNodeId,

        Map<Integer, byte[]> hashes,List<Map<Integer, byte[]>>
legacyHashes,
        int chainIndex,
        Map<Integer, List<Tuple2<byte[], byte[]>>>
chainedOperatorHashes) {
// builtVertices这个集合是用来存放已经构建好的StreamNode的id
    if (!builtVertices.contains(startNodeId)) {

        /**
         * 过渡用的出边集合，用来生成最终的 JobEdge，
         * 注意：存在某些StreamNode会连接到一起，比如source->map->flatMap，
         * 如果这几个StreamNode连接到一起，则transitiveOutEdges是不包括
         chain 内部的边，既不包含source->map的StreamEdge的 */
        List<StreamEdge> transitiveOutEdges = new
ArrayList<StreamEdge>();

        /** 可以与当前节点链接的StreamEdge */
        List<StreamEdge> chainableOutputs = new
ArrayList<StreamEdge>();
        /** 不可以与当前节点链接的StreamEdge */
        List<StreamEdge> nonChainableOutputs = new
ArrayList<StreamEdge>();

        for (StreamEdge outEdge :
streamGraph.getStreamNode(currentNodeId).getOutEdges()) {
            if (isChainable(outEdge, streamGraph)) {
                chainableOutputs.add(outEdge);
            } else {
                nonChainableOutputs.add(outEdge);
            }
        }

        /** 对于每个可连接的StreamEdge，递归调用其目
标StreamNode，startNodeId保持不变，但是chainIndex会加1 */
        for (StreamEdge chainable : chainableOutputs) {
            transitiveOutEdges.addAll(
                createChain(startNodeId,
chainable.getTargetId(), hashes, legacyHashes, chainIndex + 1,
chainedOperatorHashes));
        }

        /**
         * 对于每个不可连接的StreamEdge，则将对于的StreamEdge就是当前链的一个
输出StreamEdge，所以会添加到transitiveOutEdges这个集合中
         * 然后递归调用其目标节点，注意，startNodeID变成了nonChainable这

```

```

一个StreamEdge的输出节点id, chainIndex也赋值为0, 说明重新开始一条链的建立
*/

    for (StreamEdge nonChainable : nonChainableOutputs) {
        transitiveOutEdges.add(nonChainable);
        createChain(nonChainable.getTargetId(),
nonChainable.getTargetId(), hashes, legacyHashes, 0,
chainedOperatorHashes);
    }

    /** 获取链起始节点对应的操作符散列值列表, 如果没有, 则是空列表 */
    List<Tuple2<byte[], byte[]>> operatorHashes =
        chainedOperatorHashes.computeIfAbsent(startNodeId, k ->
new ArrayList<>());

    /** 当前 StreamNode 对应的主散列值 */
    byte[] primaryHashBytes = hashes.get(currentNodeId);

    /** 遍历每个备用散列值, 并与主散列值, 组成一个二元组, 添加到列表中 */
    for (Map<Integer, byte[]> legacyHash : legacyHashes) {
        operatorHashes.add(new Tuple2<>(primaryHashBytes,
legacyHash.get(currentNodeId)));
    }

    /** 生成当前节点的显示名, 如: "Keyed Aggregation -> Sink: Unnamed"
*/
        chainedNames.put(currentNodeId,
createChainedName(currentNodeId, chainableOutputs));
        chainedMinResources.put(currentNodeId,
createChainedMinResources(currentNodeId, chainableOutputs));
        chainedPreferredResources.put(currentNodeId,
createChainedPreferredResources(currentNodeId, chainableOutputs));

    /**
     * 1、如果当前节点是起始节点, 则直接创建 JobVertex 并返回
     StreamConfig,
     * 2、否则先创建一个空的 StreamConfig
     *
     * createJobVertex 函数就是根据 StreamNode 创建对应的 JobVertex,
     并返回了空的 StreamConfig
     */
        StreamConfig config = currentNodeId.equals(startNodeId)
            ? createJobVertex(startNodeId, hashes,
legacyHashes, chainedOperatorHashes) : new StreamConfig(new
Configuration());

    /**
     * 对Flink StreamConfig就是对Flink Configuration的封装

```

```

    * {@link StreamConfig}就是对{@link Configuration}的封装,
    * 所以通过{@code StreamConfig}设置的配置, 最终都是保存在{@code
Configuration}中的。
    */

    /**
     * 设置 JobVertex 的 StreamConfig, 基本上是序列化 StreamNode 中的
     配置到 StreamConfig 中。
     * 其中包括 序列化器, StreamOperator, Checkpoint 等相关配置
     * 经过这一步操作后, StreamNode的相关配置会通过对{@code StreamNode}
     的设置接口, 将配置保存在{@code Configuration}中,
     * 而{@code Configuration}是是{@link JobVertex}的属性, 也就是说经
     过这步操作, 相关配置已经被保存到了{@code JobVertex}中。
    */
    setVertexConfig(currentNodeId, config, chainableOutputs,
nonChainableOutputs);

    if (currentNodeId.equals(startNodeId)) {
        /** 如果是chain的起始节点。(不是chain的中间节点, 会被标记成 chain
start) */
        config.setChainStart();
        config.setChainIndex(0);

        config.setOperatorName(streamGraph.getStreamNode(currentNodeId).get
OperatorName());
        /** 我们也会把物理出边写入配置, 部署时会用到 */
        config.setOutEdgesInOrder(transitiveOutEdges);

        config.setOutEdges(streamGraph.getStreamNode(currentNodeId).getOutE
dges());

        /** 将当前节点(headOfChain)与所有出边相连 */
        for (StreamEdge edge : transitiveOutEdges) {
            /** 通过StreamEdge构建出JobEdge, 创建 IntermediateDataSet
, 用来将JobVertex和JobEdge相连 */
            connect(startNodeId, edge);
        }
        /** 将chain中所有子节点的StreamConfig写入到 headOfChain 节点的
CHAINED_TASK_CONFIG 配置中 */

        config.setTransitiveChainedTaskConfigs(chainedConfigs.get(startNode
Id));

    } else {
        /** 如果是 chain 中的子节点 */
        Map<Integer, StreamConfig> chainedConfs =

```

```

chainedConfigs.get(startNodeId);

        if (chainedConfs == null) {
            chainedConfigs.put(startNodeId, new
HashMap<Integer, StreamConfig>());
        }
        config.setChainIndex(chainIndex);
        StreamNode node =
streamGraph.getStreamNode(currentNodeId);
        config.setOperatorName(node.getOperatorName());
        /** 将当前节点的StreamConfig添加到该chain的config集合中 */
        chainedConfigs.get(startNodeId).put(currentNodeId,
config);
    }

    config.setOperatorID(new OperatorID(primaryHashBytes));
    /** 如果节点的输出StreamEdge已经为空, 则说明是链的结尾 */
    if (chainableOutputs.isEmpty()) {
        config.setChainEnd();
    }
    return transitiveOutEdges;

} else {
    /** startNodeId 如果已经构建过, 则直接返回 */
    return new ArrayList<>();
}
}

```

创建过程详解代码中的注解。

如果startNodeId已经被构建完成，则直接返回一个空集合；如果还没有构建，则开始新的构建，

- 显示递归构建链的下游节点，在下游节点都递归构建完成后，再构建当前节点；
- 如果当前节点是一个链的起始节点，则新建一个JobVertex，并将相关配置都通过StreamConfig提供的接口，配置到JobVertex的configuration属性中；
- 如果是链的中间节点，则将相关配置添加到其对应的StreamConfig对象中。

在对head节点设置时，会在head节点与每个输出StreamEdge的目标节点之间建立连接，代码如下：

```

private void connect(Integer headOfChain, StreamEdge edge) {

```

```

    /** 将当前edge记录物理边界顺序集合中 */
    physicalEdgesInOrder.add(edge);

    /** 获取StreamEdge的输出节点的id */
    Integer downStreamVertexID = edge.getTargetId();

    /** 通过节点id获取到要进行连接的上下游JobVertex节点 */
    JobVertex headVertex = jobVertices.get(headOfChain);
    JobVertex downStreamVertex =
    jobVertices.get(downStreamVertexID);

    /** 获取下游JobVertex的配置属性 */
    StreamConfig downStreamConfig = new
    StreamConfig(downStreamVertex.getConfiguration());

    /** 下游JobVertex的输入源加1 */

    downStreamConfig.setNumberOfInputs(downStreamConfig.getNumberOfInputs() + 1);

    /** 获取StreamEdge中的分区器 */
    StreamPartitioner<?> partitioner = edge.getPartitioner();
    JobEdge jobEdge;
    /** 根据分区器的不同子类, 创建相应的JobEdge */
    if (partitioner instanceof ForwardPartitioner) {
        /** 向前传递分区 */
        jobEdge = downStreamVertex.connectNewDataSetAsInput(
            headVertex,
            DistributionPattern.POINTWISE,
            ResultPartitionType.PIPELINED_BOUNDED);
    } else if (partitioner instanceof RescalePartitioner){
        /** 可扩展分区 */
        jobEdge = downStreamVertex.connectNewDataSetAsInput(
            headVertex,
            DistributionPattern.POINTWISE,
            ResultPartitionType.PIPELINED_BOUNDED);
    } else {
        /** 其他分区 */
        jobEdge = downStreamVertex.connectNewDataSetAsInput(
            headVertex,
            DistributionPattern.ALL_TO_ALL,
            ResultPartitionType.PIPELINED_BOUNDED);
    }
    /** 设置数据传输策略,以便在web上显示 */
    jobEdge.setShipStrategyName(partitioner.toString());

    /** 打印调试日志 */

```

```

    if (LOG.isDebugEnabled()) {
        LOG.debug("CONNECTED: {} - {} -> {}",
            partitioner.getClass().getSimpleName(),
            headOfChain, downstreamVertexID);
    }
}

```

其中JobEdge是通过下游JobVertex的connectNewDataSetAsInput方法来创建的，在创建JobEdge的前，会先用上游JobVertex创建一个IntermediateDataSet实例，用来作为上游JobVertex的结果输出，然后作为JobEdge的输入，构建JobEdge实例，具体实现如下：

```

public JobEdge connectNewDataSetAsInput(
    JobVertex input,
    DistributionPattern distPattern,
    ResultPartitionType partitionType) {
    /** 创建输入JobVertex的输出数据集合 */
    IntermediateDataSet dataSet =
input.createAndAddResultDataSet(partitionType);
    /** 构建JobEdge实例 */
    JobEdge edge = new JobEdge(dataSet, this, distPattern);
    /** 将JobEdge实例，作为当前JobVertex的输入 */
    this.inputs.add(edge);
    /** 设置中间结果集合dataSet的消费者是上面创建的JobEdge */
    dataSet.addConsumer(edge);
    return edge;
}

```

通过上述的构建过程，就可以实现上下游JobVertex的连接，上游JobVertex ——> 中间结果集合IntermediateDataSet ——> JobEdge ——> 下游JobVertex。其中IntermediateDataSet和JobEdge是用来建立上下游JobVertex之间连接的配置；一个IntermediateDataSet有一个消息producer，可以有多个消息消费者JobEdge；一个JobEdge则有一个数据源IntermediateDataSet，一个目标JobVertex；一个JobVertex可以产生多个输出IntermediateDataSet，也可以接受来自多个JobEdge的数据。

通过上述的构建过程，对于这里的例子，source -> flatMap 组成一个链，构建成一个JobVertex，reduce -> sink 组成一个链，构建成一个JobVertex。

总结

JobGraph是在StreamGraph的基础之上，对StreamNode进行了关联合并的操作，比如对于source -> flatMap -> reduce -> sink 这样一个数据处理链，当source和flatMap满足链接的条件时，可以可以将两个操作符的操作放到一个线程并行执行，这样可以减少网络中的数据传输，由于在source和flatMap之间的传输的数据也不用序列化和反序列化，所以也提高了程序的执行效率。

本文参考：https://blog.csdn.net/qq_21653785/article/details/79510140