toc: true title: 《从1到100深入学习Flink》——如何获取 StreamGraph？ date: 2019-02-19 tags:

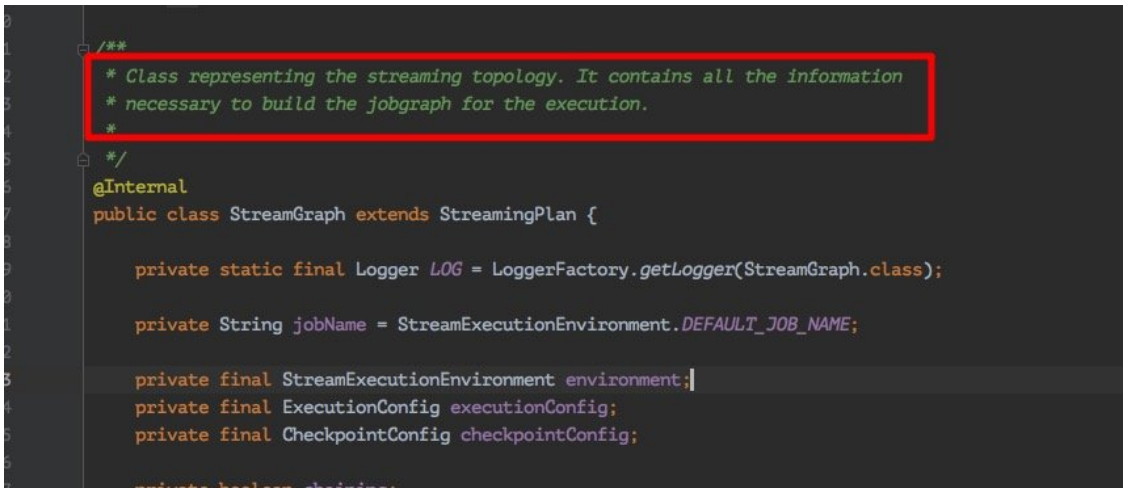
- Flink
- 大数据
- 流式计算

前言

前两篇文章分析了批和流的 wordcount 程序的启动流程，在上篇流程序的文章中，我们可以留个两个问题就是 StreamGraph 和 JobGraph 是如何获取的?

在了解获取之前我们先看一下 StreamGraph 是啥?

StreamGraph



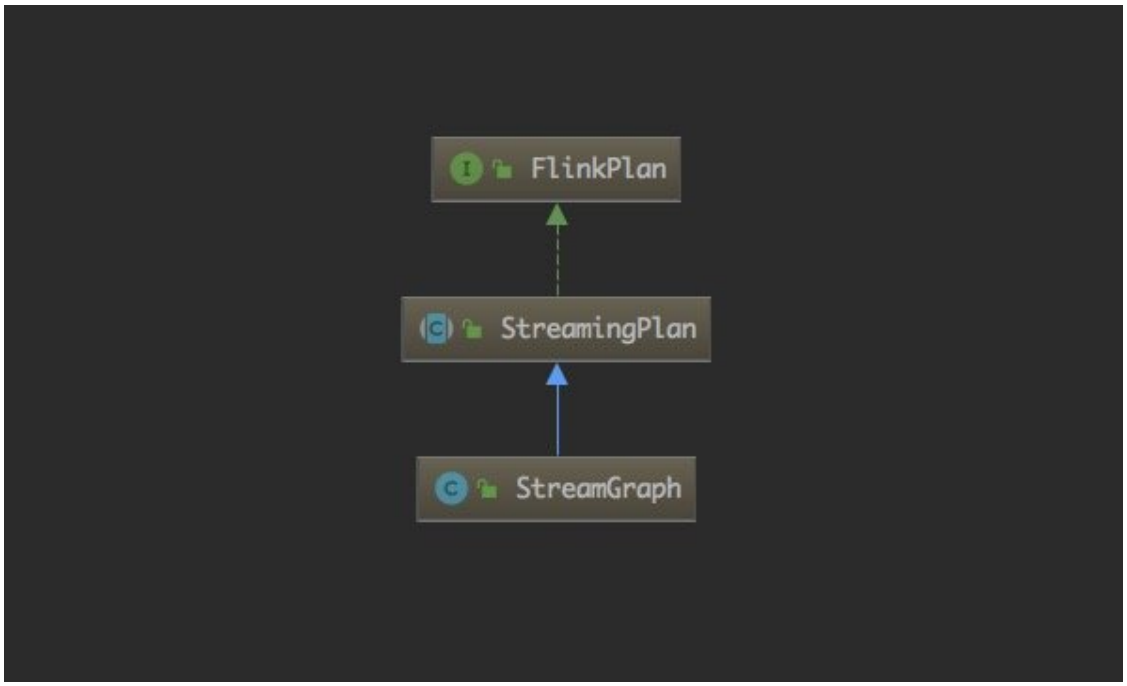
```
/**
 * Class representing the streaming topology. It contains all the information
 * necessary to build the jobgraph for the execution.
 *
 */
@Internal
public class StreamGraph extends StreamingPlan {

    private static final Logger LOG = LoggerFactory.getLogger(StreamGraph.class);

    private String jobName = StreamExecutionEnvironment.DEFAULT_JOB_NAME;

    private final StreamExecutionEnvironment environment;
    private final ExecutionConfig executionConfig;
    private final CheckpointConfig checkpointConfig;
```

它表示流拓扑, 包含了为执行构建作业图所需的所有信息。



它继承了抽象类 StreamingPlan（代表 Flink 流计划）类，而 FlinkPlan 是一个接口，是代表了批和流程序的执行计划。

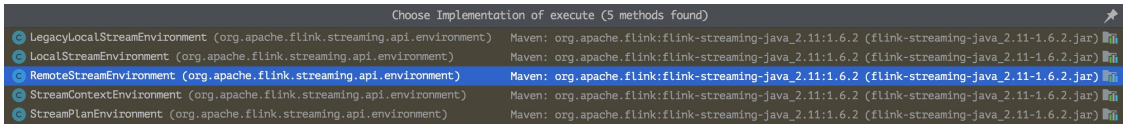
下面再来讲解一下 StreamGraph 是如何获取的？下篇文章我们再来分析一下 如何获取 JobGraph?

execute 方法

在流程序 word count 程序中有如下这行：

```
env.execute("zhisheng ——— word count streaming demo");
```

跟进去发现这个 execute 方法是在 StreamExecutionEnvironment 类中的，但是这是一个抽象方法，具体的实现在其子类中都有实现。



但是不管其中哪个子类，他在 execute 方法中都有调用 `getStreamGraph()` 方法，而这个方法依旧是在 StreamExecutionEnvironment 类中：

```
     * Getter of the {@link
org.apache.flink.streaming.api.graph.StreamGraph} of the streaming
job.
     *
     * @return The streamgraph representing the transformations
     */
@Internal
public StreamGraph getStreamGraph() {
    if (transformations.size() <= 0) {
        throw new IllegalStateException("No operators defined in
streaming topology. Cannot execute.");
    }
    return StreamGraphGenerator.generate(this, transformations);
}
```

这里的 transformations 这个列表就是在对数据流的处理过程中，会将 flatMap、reduce 这些转换操作对应的 StreamTransformation 保存下来的列表，根据对数据流做的转换操作，这个列表中，当前保存的对象有：

- 表示 flatMap 操作的 OneInputTransformation 对象，其 input 属性指向的是数据源的转换 SourceTransformation。
- 表示 reduce 操作的 OneInputTransformation 对象，其 input 属性指向的是表示 keyBy 的转换 PartitionTransformation，而 PartitionTransformation 的 input 属性指向的是 flatMap 的转换 OneInputTransformation；
- sink 操作对应的 SinkTransformation 对象，其 input 属性指向的是 reduce 转化的 OneInputTransformation 对象。

接着调用 StreamGraphGenerator 的 generate 方法来获取 transformations 列表对应的 StreamGraph。从这里可以看出，数据流在经过各种转换操作之后，各种转换的相关信息都已经保存在了 transformations 这个列表里的元素中了。继续向下看代码：

```
/**
 * Generates a {@code StreamGraph} by traversing the graph of
{@code StreamTransformations}
 * starting from the given transformations.
 *
 * @param env The {@code StreamExecutionEnvironment} that is used
to set some parameters of the
 *            job
 * @param transformations The transformations starting from which
to transform the graph
 *
 * @return The generated {@code StreamGraph}
 */
public static StreamGraph generate(StreamExecutionEnvironment env,
List<StreamTransformation<?>> transformations) {
    return new
StreamGraphGenerator(env).generateInternal(transformations);
}
```

可以看到，先以 env 为构造函数的入参，构建了一个 StreamGraphGenerator 实例，看下构造函数的实现：

```
/**
 * Private constructor. The generator should only be invoked using
{@link #generate}.
 */
private StreamGraphGenerator(StreamExecutionEnvironment env) {
    this.streamGraph = new StreamGraph(env);
    this.streamGraph.setChaining(env.isChainingEnabled());
    this.streamGraph.setStateBackend(env.getStateBackend());
    this.env = env;
    this.alreadyTransformed = new HashMap<>();
}
```

这个构造函数是 private的，所以 StreamGraphGenerator 的实例构造只能通过其静态的 generate 方法。另外在构造函数中，初始化了一个 StreamGraph 实例，并设置了一些属性值，然后给 env 赋值，并初始化 alreadyTransformed 为一个空 map。再看下 StreamGraph 的构造函数实现。

```
public StreamGraph(StreamExecutionEnvironment environment) {
    this.environment = environment;
    this.executionConfig = environment.getConfig();
    this.checkpointConfig = environment.getCheckpointConfig();

    // create an empty new stream graph.
    clear();
}

/**
 * Remove all registered nodes etc.
 */
public void clear() {
    streamNodes = new HashMap<>();
    virtualSelectNodes = new HashMap<>();
    virtualSideOutputNodes = new HashMap<>();
    virtualPartitionNodes = new HashMap<>();
    vertexIDtoBrokerID = new HashMap<>();
    vertexIDtoLoopTimeout  = new HashMap<>();
    iterationSourceSinkPairs = new HashSet<>();
    sources = new HashSet<>();
    sinks = new HashSet<>();
}
```

可以看出其构造函数中就是做了一些初始化的操作，给 StreamGraph 的各个属性设置初始值，都是一些空集合。 在获取到 StreamGraphGenerator 的实例后，继续看其 generatorInternal 方法的逻辑：

```
/**
 * This starts the actual transformation, beginning from the sinks.
 */
private StreamGraph generateInternal(List<StreamTransformation<?>>
transformations) {
    for (StreamTransformation<?> transformation: transformations) {
        transform(transformation);
    }
    return streamGraph;
}
```

逻辑很明了，就是顺序遍历 transformations 列表中的元素，然后依次转换，最后返回转化好的 StreamGraph 的实例。

1、flatMap对应StreamTransformation子类实例的转化

接下来就看下，对于 transformations 列表中的每个元素是如何转化的。

```
/**
 * Transforms one {@code StreamTransformation}.
 *
 * <p>This checks whether we already transformed it and exits early
in that case. If not it
 * delegates to one of the transformation specific methods.
 */
private Collection<Integer> transform(StreamTransformation<?>
transform) {

    //判断传入的transform是否已经被转化过，如果已经转化过，则直接返回转化后对
应的结果
    if (alreadyTransformed.containsKey(transform)) {
        return alreadyTransformed.get(transform);
    }

    LOG.debug("Transforming " + transform);
   /** 对于该transform，如果没有设置最大并行度，则尝试获取job的最大并行度，
并设置给它 */
    if (transform.getMaxParallelism() <= 0) {

      /** 如果最大并行度没有设置，如果job设置了最大并行度，则获取job最大并行度
*/
        int globalMaxParallelismFromConfig =
env.getConfig().getMaxParallelism();
        if (globalMaxParallelismFromConfig > 0) {

transform.setMaxParallelism(globalMaxParallelismFromConfig);
        }
    }

   /** 这里调用该方法的目的就是为了尽早发现，输出数据类型信息是否丢失，抛出异常
*/
    transform.getOutputType();

   /** 针对不同的StreamTransformation的子类实现，委托不同的方法进行转化 */
    Collection<Integer> transformedIds;
    if (transform instanceof OneInputTransformation<?, ?>) {
        transformedIds =
transformOneInputTransform((OneInputTransformation<?, ?>)
transform);
    } else if (transform instanceof TwoInputTransformation<?, ?, ?
>) {
```

```
        transformedIds =
transformTwoInputTransform((TwoInputTransformation<?, ?, ?>)
transform);
    } else if (transform instanceof SourceTransformation<?>) {
        transformedIds = transformSource((SourceTransformation<?>)
transform);
    } else if (transform instanceof SinkTransformation<?>) {
        transformedIds = transformSink((SinkTransformation<?>)
transform);
    } else if (transform instanceof UnionTransformation<?>) {
        transformedIds = transformUnion((UnionTransformation<?>)
transform);
    } else if (transform instanceof SplitTransformation<?>) {
        transformedIds = transformSplit((SplitTransformation<?>)
transform);
    } else if (transform instanceof SelectTransformation<?>) {
        transformedIds = transformSelect((SelectTransformation<?>)
transform);
    } else if (transform instanceof FeedbackTransformation<?>) {
        transformedIds =
transformFeedback((FeedbackTransformation<?>) transform);
    } else if (transform instanceof CoFeedbackTransformation<?>) {
        transformedIds =
transformCoFeedback((CoFeedbackTransformation<?>) transform);
    } else if (transform instanceof PartitionTransformation<?>) {
        transformedIds =
transformPartition((PartitionTransformation<?>) transform);
    } else if (transform instanceof SideOutputTransformation<?>) {
        transformedIds =
transformSideOutput((SideOutputTransformation<?>) transform);
    } else {
        throw new IllegalStateException("Unknown transformation: "
+ transform);
    }

   /** 将转化结果记录到alreadyTransformed这个map中，这里做条件判断，是考虑到
迭代转换下，transform会先添加自身，再转化反馈边界 */
    if (!alreadyTransformed.containsKey(transform)) {
        alreadyTransformed.put(transform, transformedIds);
    }

   /** 将transform的相应属性设置到其在streamGraph中对应的节点上 */
    if (transform.getBufferTimeout() >= 0) {
        streamGraph.setBufferTimeout(transform.getId(),
transform.getBufferTimeout());
    }
    if (transform.getUid() != null) {
        streamGraph.setTransformationUID(transform.getId(),
```

```
transform.getUid());
    }

    if (transform.getUserProvidedNodeHash() != null) {
        streamGraph.setTransformationUserHash(transform.getId(),
transform.getUserProvidedNodeHash());
    }

    if (transform.getMinResources() != null &&
transform.getPreferredResources() != null) {
        streamGraph.setResources(transform.getId(),
transform.getMinResources(), transform.getPreferredResources());
    }
  /** 返回转化结果 */
    return transformedIds;
}
```

在进行一些校验和设置后，会根据 StreamTransformation 的不同实现子类，调用不同的方法进行转化，转化完成后，记录到 alreadyTransformed 这个 map 中，再将一些属性设置到 streamGraph 中对应的节点上，并返回转化结果。

先来看 transformations 这个列表中的第一个元素，flatMap 对应的 OneInputTransformation 的转化过程。通过上述代码，对于 OneInputTransformation，执行如下：

```
if (transform instanceof OneInputTransformation<?, ?>) {
   transformedIds =
transformOneInputTransform((OneInputTransformation<?, ?>)
transform);
}



/**
 * Transforms a {@code OneInputTransformation}.
 *
 * <p>This recursively transforms the inputs, creates a new {@code
StreamNode} in the graph and
 * wired the inputs to this new node.
 */
private <IN, OUT> Collection<Integer>
transformOneInputTransform(OneInputTransformation<IN, OUT>
transform) {
   /** 先递归转化对应的input属性 */
```

```
    Collection<Integer> inputIds = transform(transform.getInput());


    /** 在递归调用过程中，有可能已经被转化过 */
    if (alreadyTransformed.containsKey(transform)) {
        return alreadyTransformed.get(transform);
    }

    /** 判断transform的槽共享组的名称 */
    String slotSharingGroup =
determineSlotSharingGroup(transform.getSlotSharingGroup(),
inputIds);
    /** 添加新的节点 */
    streamGraph.addOperator(transform.getId(),
            slotSharingGroup,
            transform.getCoLocationGroupKey(),
            transform.getOperator(),
            transform.getInputType(),
            transform.getOutputType(),
            transform.getName());

    /** 属性设置 */
    if (transform.getStateKeySelector() != null) {
        TypeSerializer<?> keySerializer =
transform.getStateKeyType().createSerializer(env.getConfig());
        streamGraph.setOneInputStateKey(transform.getId(),
transform.getStateKeySelector(), keySerializer);
    }

    streamGraph.setParallelism(transform.getId(),
transform.getParallelism());
    streamGraph.setMaxParallelism(transform.getId(),
transform.getMaxParallelism());

    /** 构建edge */
    for (Integer inputId: inputIds) {
        streamGraph.addEdge(inputId, transform.getId(), 0);
    }

    /** 将transform的id封装成一个集合返回 */
    return Collections.singleton(transform.getId());
}
```

转化过程，先递归转化其 input，由于递归转化中，可能自身已经被转化，如果转化，则返回转化好的结果，如果没有，则继续获取其槽共享组名称，在 streamGraph 中添加 operator 节点，然后进行属性设置，再进行 edge 的构建，最后返回自身 id 构成的集合。

对于这里的 flatMap 对应的 OneInputTransformation，其 input 属性的值是数据源的转换 SourceTransformation，所以先来看下递归调用转化 SourceTransformation 的逻辑。

```
/**
 * Transforms a {@code SourceTransformation}.
 */
private <T> Collection<Integer>
transformSource(SourceTransformation<T> source) {
   /** 对于数据流的源来说，如果用户没有指定slotSharingGroup，这里返回的就
是"default" */
    String slotSharingGroup =
determineSlotSharingGroup(source.getSlotSharingGroup(),
Collections.emptyList());

    streamGraph.addSource(source.getId(),
            slotSharingGroup,
            source.getCoLocationGroupKey(),
            source.getOperator(),
            null,
            source.getOutputType(),
            "Source: " + source.getName());
    if (source.getOperator().getUserFunction() instanceof
InputFormatSourceFunction) {
        InputFormatSourceFunction<T> fs =
(InputFormatSourceFunction<T>)
source.getOperator().getUserFunction();
        streamGraph.setInputFormat(source.getId(), fs.getFormat());
    }
    streamGraph.setParallelism(source.getId(),
source.getParallelism());
    streamGraph.setMaxParallelism(source.getId(),
source.getMaxParallelism());
    return Collections.singleton(source.getId());
}
```

对于每个 StreamTransformation 在转化的时候，都会需要决定其所属的槽共享组的名称，来看下决定的逻辑。

```
/**
 * Determines the slot sharing group for an operation based on the
slot sharing group set by
 * the user and the slot sharing groups of the inputs.
 *
 * <p>If the user specifies a group name, this is taken as is. If
nothing is specified and
 * the input operations all have the same group name then this name
is taken. Otherwise the
 * default group is chosen.
 *
 * @param specifiedGroup The group specified by the user.
 * @param inputIds The IDs of the input operations.
 */
private String determineSlotSharingGroup(String specifiedGroup,
Collection<Integer> inputIds) {
   /** 如果以及指定了，则直接返回指定值 */
    if (specifiedGroup != null) {
        return specifiedGroup;
    } else {
          /** 如果没有指定，则遍历输入节点对应的槽共享组名称 */
        String inputGroup = null;
        for (int id: inputIds) {
            String inputGroupCandidate =
streamGraph.getSlotSharingGroup(id);
            if (inputGroup == null) {
                inputGroup = inputGroupCandidate;
            } else if (!inputGroup.equals(inputGroupCandidate)) {
                return "default";
            }
        }
         /** 如果还没null，则返回"default" */
        return inputGroup == null ? "default" : inputGroup;
    }
}
```

上述逻辑分为以下几种情况：

- 如果指定了分组名，则直接返回指定的值；
- 如果没有指定分组名，则遍历输入的各个节点的分组名；
  - 如果所有输入的分组名都是一样的，则将这个一样的分组名作为当前节点的分组名；
  - 如果所有输入的分组名有不一样的，则返回默认分组名”default”；

对于这里，在转化 SourceTransformation 时，由于没有设定指定的槽共享分组名，同时它是数据源，没有输入，所有其 slotSharingGroup 的值为”default”。分组名确定后，就会向 streamGraph 中添加一个 source。对于 source 来说，输入数据类型为 null，操作符名称拼接够就是"Source: Socket Stream”，添加逻辑如下：

```
public <IN, OUT> void addSource(Integer vertexID,
    String slotSharingGroup,
    @Nullable String coLocationGroup,
    StreamOperator<OUT> operatorObject,
    TypeInformation<IN> inTypeInfo,
    TypeInformation<OUT> outTypeInfo,
    String operatorName) {
    addOperator(vertexID, slotSharingGroup, coLocationGroup,
operatorObject, inTypeInfo, outTypeInfo, operatorName);
    sources.add(vertexID);
}
```

先调用 addOperator 方法，添加一个新的节点，然后将节点的 id，其实就是 SourceTransformation 的 ID，这里就是 1，添加到 sources 这个集合中。看下添加操作符节点的逻辑。

```
public <IN, OUT> void addOperator(
        Integer vertexID,
        String slotSharingGroup,
        @Nullable String coLocationGroup,
        StreamOperator<OUT> operatorObject,
        TypeInformation<IN> inTypeInfo,
        TypeInformation<OUT> outTypeInfo,
        String operatorName) {
   /** 根据操作符的不同类型，决定不同的节点类，进行节点的构造 */
    if (operatorObject instanceof StoppableStreamSource) {
        addNode(vertexID, slotSharingGroup, coLocationGroup,
StoppableSourceStreamTask.class, operatorObject, operatorName);
    } else if (operatorObject instanceof StreamSource) {
        addNode(vertexID, slotSharingGroup, coLocationGroup,
SourceStreamTask.class, operatorObject, operatorName);
    } else {
        addNode(vertexID, slotSharingGroup, coLocationGroup,
OneInputStreamTask.class, operatorObject, operatorName);
    }
   /** 确定输入和输出的序列化器，类型不为null，且不是MissingTypeInfo，则构
造对应的序列化器，否则序列化器为null */
    TypeSerializer<IN> inSerializer = inTypeInfo != null && !
(inTypeInfo instanceof MissingTypeInfo) ?
```

```
inTypeInfo.createSerializer(executionConfig) : null;
   /** 设置序列化器 */

    TypeSerializer<OUT> outSerializer = outTypeInfo != null && !
(outTypeInfo instanceof MissingTypeInfo) ?
outTypeInfo.createSerializer(executionConfig) : null;
   /** 根据操作符类型, 进行输出数据类型设置 */
    setSerializers(vertexID, inSerializer, null, outSerializer);

    if (operatorObject instanceof OutputTypeConfigurable &&
outTypeInfo != null) {
        @SuppressWarnings("unchecked")
        OutputTypeConfigurable<OUT> outputTypeConfigurable =
(OutputTypeConfigurable<OUT>) operatorObject;
        // sets the output type which must be know at StreamGraph
creation time
        outputTypeConfigurable.setOutputType(outTypeInfo,
executionConfig);
    }
   /** 根据操作符类型, 进行输入数据类型判断 */
    if (operatorObject instanceof InputTypeConfigurable) {
        InputTypeConfigurable inputTypeConfigurable =
(InputTypeConfigurable) operatorObject;
        inputTypeConfigurable.setInputType(inTypeInfo,
executionConfig);
    }
   /** 根据配置, 打印调试日志 */
    if (LOG.isDebugEnabled()) {
        LOG.debug("Vertex: {}", vertexID);
    }
}
```

其中会根据 operatorObject 的不同类型，在添加节点，以及后续的节点配置中，做一些特定配置。逻辑比较清晰，详见注解。

addNode 方法的逻辑如下，就是新建了一个 StreamNode 实例作为新的节点，然后保存到 streamNodes 这个 map 中，key 是节点 id，value 就是节点 StreamNode。StreamNode 就是用来描述流处理中的一个节点，其包含了该节点对数据的操作符，并行度，输入输出数据类型，分区等属性，以及这个节点与上游节点链接的 StreamEdge 集合，与下游节点链接的 StreamEdge 集合。

```
protected StreamNode addNode(Integer vertexID,
    String slotSharingGroup,
    @Nullable String coLocationGroup,
    Class<? extends AbstractInvokable> vertexClass,
    StreamOperator<?> operatorObject,
    String operatorName) {
   /** 校验添加新节点的id, 与已添加的节点id, 是否有重复, 如果有, 则抛出异常*/
    if (streamNodes.containsKey(vertexID)) {
        throw new RuntimeException("Duplicate vertexID " +
vertexID);
    }

    StreamNode vertex = new StreamNode(environment,
        vertexID,
        slotSharingGroup,
        coLocationGroup,
        operatorObject,
        operatorName,
        new ArrayList<OutputSelector<?>>(),
        vertexClass);
   /** 将新构建的节点保存记录 */
    streamNodes.put(vertexID, vertex);

    return vertex;
}
```

到这里，数据源节点就添加到了 streamGraph 中，然后就可以回到 flatMap 对应的 OneInputTransformation 的转化操作中。

flatMap 对应的 OneInputTransformation 在递归转化完其输入后，也会决定其 slotSharingGroup 的值，这里决定的结果也是”default”。然后调用 streamGraph 的 addOperator 方法进行节点添加，对应的节点的id是2，并增加到 StreamGraph 的属性 streamNodes 这个 map 中，key 的值是2。

在节点这两个节点构建好之后，会进行 edge 的构建，如下：

```
for (Integer inputId: inputIds) {
   streamGraph.addEdge(inputId, transform.getId(), 0);
}
```

这里来说，inputId 这个集合只有一个值，就是 1，表示数据源节点。而这里就会构建一个节点 id 为 1 到节点 id 为 2 的 edge。

```
public void addEdge(Integer upStreamVertexID, Integer
downStreamVertexID, int typeNumber) {
    addEdgeInternal(upStreamVertexID,
            downStreamVertexID,
            typeNumber,
            null,
            new ArrayList<String>(),
            null);

}
```

真正的添加逻辑在 addEdgeInternal 方法中。该方法中，会构建一个 StreamEdge。StreamEdge 是用来描述流拓扑中的一个边界，其有对一个的源 StreamNode 和目标 StreamNode，以及数据在源到目标直接转发时，进行的分区与 select 等操作的逻辑。接下来看 StreamEdge 的新增逻辑。

```
private void addEdgeInternal(Integer upStreamVertexID,
        Integer downStreamVertexID,
        int typeNumber,StreamPartitioner<?> partitioner,
List<String> outputNames,
        OutputTag outputTag) {
   /** 根据源节点的id，判断进入不同的处理逻辑 */
    if (virtualSideOutputNodes.containsKey(upStreamVertexID)) {
        int virtualId = upStreamVertexID;
        upStreamVertexID =
virtualSideOutputNodes.get(virtualId).f0;
        if (outputTag == null) {
            outputTag = virtualSideOutputNodes.get(virtualId).f1;
        }
        addEdgeInternal(upStreamVertexID, downStreamVertexID,
typeNumber, partitioner, null, outputTag);
    } else if (virtualSelectNodes.containsKey(upStreamVertexID)) {
        int virtualId = upStreamVertexID;
        upStreamVertexID = virtualSelectNodes.get(virtualId).f0;
        if (outputNames.isEmpty()) {
            // selections that happen downstream override earlier
selections
            outputNames = virtualSelectNodes.get(virtualId).f1;
        }
        addEdgeInternal(upStreamVertexID, downStreamVertexID,
typeNumber, partitioner, outputNames, outputTag);
    } else if (virtualPartitionNodes.containsKey(upStreamVertexID))
{
        int virtualId = upStreamVertexID;
        upStreamVertexID = virtualPartitionNodes.get(virtualId).f0;
```

```
        if (partitioner == null) {
            partitioner = virtualPartitionNodes.get(virtualId).f1;
        }
        addEdgeInternal(upStreamVertexID, downStreamVertexID,
typeNumber, partitioner, outputNames, outputTag);
    } else {
          /** 根据上下节点的id，分别获取对应的StreamNode实例 */
        StreamNode upstreamNode = getStreamNode(upStreamVertexID);
        StreamNode downstreamNode =
getStreamNode(downStreamVertexID);

      /** 如果没有指定分区器，并且上游节点和下游节点的操作符的并行度也是一样的
话，就采用forward分区，否则，采用rebalance分区 */
        if (partitioner == null && upstreamNode.getParallelism() ==
downstreamNode.getParallelism()) {
            partitioner = new ForwardPartitioner<Object>();
        } else if (partitioner == null) {
            partitioner = new RebalancePartitioner<Object>();
        }
      /** 如果是Forward分区，而上下游的并行度不一致，则抛异常，这里是进行双重
校验 */
        if (partitioner instanceof ForwardPartitioner) {
            if (upstreamNode.getParallelism() !=
downstreamNode.getParallelism()) {
                throw new UnsupportedOperationException("Forward
partitioning does not allow " +
                        "change of parallelism. Upstream operation:
" + upstreamNode + " parallelism: " + upstreamNode.getParallelism()
+
                        ", downstream operation: " + downstreamNode
+ " parallelism: " + downstreamNode.getParallelism() +
                        " You must use another partitioning
strategy, such as broadcast, rebalance, shuffle or global.");
            }
        }
      /** 新建StreamEdge实例 */
        StreamEdge edge = new StreamEdge(upstreamNode,
downstreamNode, typeNumber, outputNames, partitioner, outputTag);
      /** 将新建的StreamEdge添加到源节点的输出边界集合中，目标节点的输入边界
集合中 */
        getStreamNode(edge.getSourceId()).addOutEdge(edge);
        getStreamNode(edge.getTargetId()).addInEdge(edge);
    }
}
```

对于这里的 flatMap 的 OneInputTransformation 的转化来说，这里在添加 StreamEdge 时，是直接进入 else 逻辑，由于数据源和 flatMap 都是并行度为 1，所以会得到 ForwardPartitioner 的实例作为分区器。

这样就处理完了 flatMap 对应的 OneInputTransformation 的转化操作，并将转化结果记录在 StreamGraph 中，source 和 faltMap 对应的 StreamNode 之间构建了一个 StreamEdge。

2、reduce对应StreamTransformation子类实例的转化

接下来再看继续分析transformations这个列表中的第二个元素，表示reduce操作的OneInputTransformation对象，其input属性指向的是表示keyBy的转换PartitionTransformation，而PartitionTransformation的input属性指向的是flatMap的转换OneInputTransformation。

由于flatMap以及source在前面已经转化完成，所以在递归调用的过程中，都是直接返回转化后的结果。然后就会轮到keyBy对应的PartitionTransformation的转化。

```
private <T> Collection<Integer>
transformPartition(PartitionTransformation<T> partition) {
   StreamTransformation<T> input = partition.getInput();
   List<Integer> resultIds = new ArrayList<>();
   /** 递归转化输入 */
   Collection<Integer> transformedIds = transform(input);
   for (Integer transformedId: transformedIds) {
      int virtualId = StreamTransformation.getNewNodeId();
      streamGraph.addVirtualPartitionNode(transformedId, virtualId,
partition.getPartitioner());
      resultIds.add(virtualId);
   }
   return resultIds;
}
```

由于其输入是flatMap转换，已经转化过，可以直接获取到transformedIds集合的元素就是2。因为分区转换是虚拟操作，所以产生一个虚拟节点的id为6，然后就是在streamGraph中添加一个虚拟分区节点。

```
public void addVirtualPartitionNode(Integer originalId, Integer
virtualId, StreamPartitioner<?> partitioner) {
   if (virtualPartitionNodes.containsKey(virtualId)) {
      throw new IllegalStateException("Already has virtual
partition node with id " + virtualId);
   }
   virtualPartitionNodes.put(virtualId,
         new Tuple2<Integer, StreamPartitioner<?>>(originalId,
partitioner));
}
```

逻辑很清晰，对于这里，就是originalId=2，virtualId=6，所以在virtualPartitionNodes这个map中就新增映射：6 ——> [2, partitioner]。

接下来开始转换reduce自身的StreamTransformation子类实例，因为也是OneInputTransformation实例，在flatMap中已经介绍过，这里就不再详细介绍，但是对于reduce又有一些不同的地方，这里就分析其不同的地方，而不同的地方就在于构建StreamEdge时，略有不同。

调用addEdge方法时的，upStreamVertexID为6，对应于keyBy产生的虚拟节点，downStreanVertexID为4，对应于reduce产生的节点。所在addEdgeInternal方法中，由于upStreamVertexID=6是被添加在virtualPartitionNodes这个map中的，所以调用时会进入对于分支，执行如下逻辑：

```
int virtualId = upStreamVertexID;
upStreamVertexID = virtualPartitionNodes.get(virtualId).f0;
if (partitioner == null) {
   partitioner = virtualPartitionNodes.get(virtualId).f1;
}
addEdgeInternal(upStreamVertexID, downStreamVertexID, typeNumber,
partitioner, outputNames, outputTag);
```

这里会将upStreamVertexID更新为2，并将keyBy对应的分区器设置给partitioner变量，然后递归调用addEdgeInternal方法后，会走到else逻辑中，这段逻辑前面已经介绍，这里就不赘述了。

这样就在flatMap和reduce对应的StreamNode之间构建了一个StreamEdge，且该StreamEdge中包含了keyBy转换中设置的分区器。

3、sink对应StreamTransformation子类实例的转化

sink操作对应的SinkTransformation对象，其input属性指向的是reduce转化的OneInputTransformation对象，根据递归转化输入的逻辑，reduce已经转化过，所以这里会进行sink自身的转化。

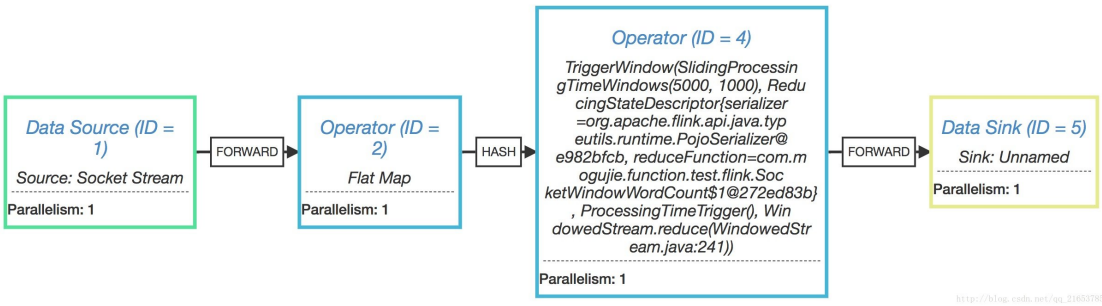
```
private <T> Collection<Integer> transformSink(SinkTransformation<T>
sink) {
   /** 递归转化输入 */
   Collection<Integer> inputIds = transform(sink.getInput());
   /** 决定槽共享分组名 */
   String slotSharingGroup =
determineSlotSharingGroup(sink.getSlotSharingGroup(), inputIds);
   /** 添加sink */
   streamGraph.addSink(sink.getId(),
         slotSharingGroup,
         sink.getOperator(),
         sink.getInput().getOutputType(),
         null,
         "Sink: " + sink.getName());
   /** 属性设置 */
   streamGraph.setParallelism(sink.getId(), sink.getParallelism());
   streamGraph.setMaxParallelism(sink.getId(),
sink.getMaxParallelism());
   /** 构建edge */
   for (Integer inputId: inputIds) {
      streamGraph.addEdge(inputId,
            sink.getId(),
            0
      );
   }
   if (sink.getStateKeySelector() != null) {
      TypeSerializer<?> keySerializer =
sink.getStateKeyType().createSerializer(env.getConfig());
      streamGraph.setOneInputStateKey(sink.getId(),
sink.getStateKeySelector(), keySerializer);
   }
   return Collections.emptyList();
}
```

逻辑与transformOneInputTransform方法的逻辑类似，不同的地方就是添加的是sink，会在StreamGraph的sinks这个属性中，加入sink节点的id，这里sink节点的id是5。

通过上述转化过程，就将对数据流的相关转换操作，都通过StreamGraph这个对象实例描述出来了。得到的StreamGraph实例的json字符串如下所示：

```
{"nodes":[{"id":1,"type":"Source: Socket Stream","pact":"Data
Source","contents":"Source: Socket Stream","parallelism":1},
{"id":2,"type":"Flat Map","pact":"Operator","contents":"Flat
Map","parallelism":1,"predecessors":
[{"id":1,"ship_strategy":"FORWARD","side":"second"}]},
{"id":4,"type":"TriggerWindow(SlidingProcessingTimeWindows(5000,
1000),
ReducingStateDescriptor{serializer=org.apache.flink.api.java.typeut
ils.runtime.PojoSerializer@e982bfcb,
reduceFunction=com.mogujie.function.test.flink.SocketWindowWordCoun
t$1@272ed83b}, ProcessingTimeTrigger(),
WindowedStream.reduce(WindowedStream.java:241))","pact":"Operator",
"contents":"TriggerWindow(SlidingProcessingTimeWindows(5000, 1000),
ReducingStateDescriptor{serializer=org.apache.flink.api.java.typeut
ils.runtime.PojoSerializer@e982bfcb,
reduceFunction=com.mogujie.function.test.flink.SocketWindowWordCoun
t$1@272ed83b}, ProcessingTimeTrigger(),
WindowedStream.reduce(WindowedStream.java:241))","parallelism":1,"p
redecessors":[{"id":2,"ship_strategy":"HASH","side":"second"}]},
{"id":5,"type":"Sink: Unnamed","pact":"Data Sink","contents":"Sink:
Unnamed","parallelism":1,"predecessors":
[{"id":4,"ship_strategy":"FORWARD","side":"second"}]}]}
```


将上述json字符串放如到 http://flink.apache.org/visualizer/ ，可以转化为图形表示，如下：



总结

本文主要通过流的 execute 方法入口，然后分析了 getStreamGraph 方法内部的具体实现。

本文参考 https://blog.csdn.net/qq_21653785/article/details/79499127