Flink 原理与实现：如何生成 StreamGraph

继上文Flink 原理与实现：架构和拓扑概览中介绍了Flink的四层执行图模型，本文将主要介绍 Flink 是如何根据用户用Stream API编写的程序，构造出一个代表拓扑结构的StreamGraph的。

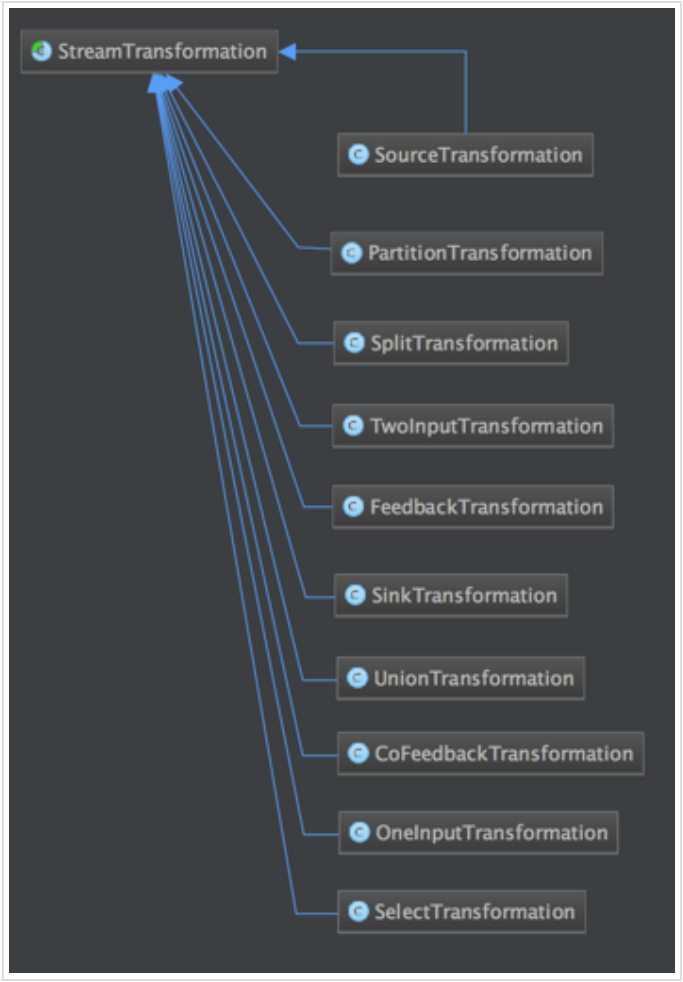
注：本文比较偏源码分析，所有代码都是基于 flink-1.0.x 版本，建议在阅读本文前先对Stream API有个了解，详见官方文档。

StreamGraph 相关的代码主要在 `org.apache.flink.streaming.api.graph` 包中。构造StreamGraph的入口函数是 `StreamGraphGenerator.generate(env, transformations)`。该函数会由触发程序执行的方法`StreamExecutionEnvironment.execute()`调用到。也就是说 StreamGraph 是在 Client 端构造的，这也意味着我们可以在本地通过调试观察 StreamGraph 的构造过程。

Transformation

`StreamGraphGenerator.generate` 的一个关键的参数是 `List<StreamTransformation<?>>`。`StreamTransformation`代表了从一个或多个`DataStream`生成新`DataStream`的操作。`DataStream`的底层其实就是一个 `StreamTransformation`，描述了这个`DataStream`是怎么来的。

StreamTransformation的类图如下图所示：

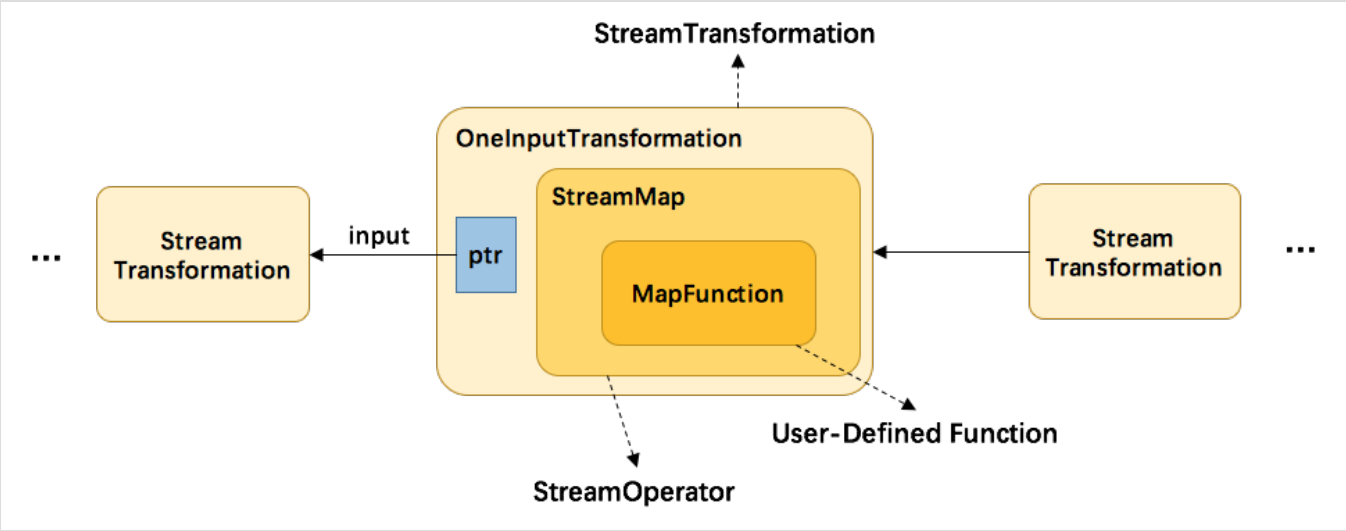


DataStream 上常见的 transformation 有 map、flatmap、filter等（见DataStream Transformation了解更多）。这些transformation会构造出一棵 StreamTransformation 树，通过这棵树转换成 StreamGraph。比如 `DataStream.map`源码如下，其中`SingleOutputStreamOperator`为DataStream的子类：

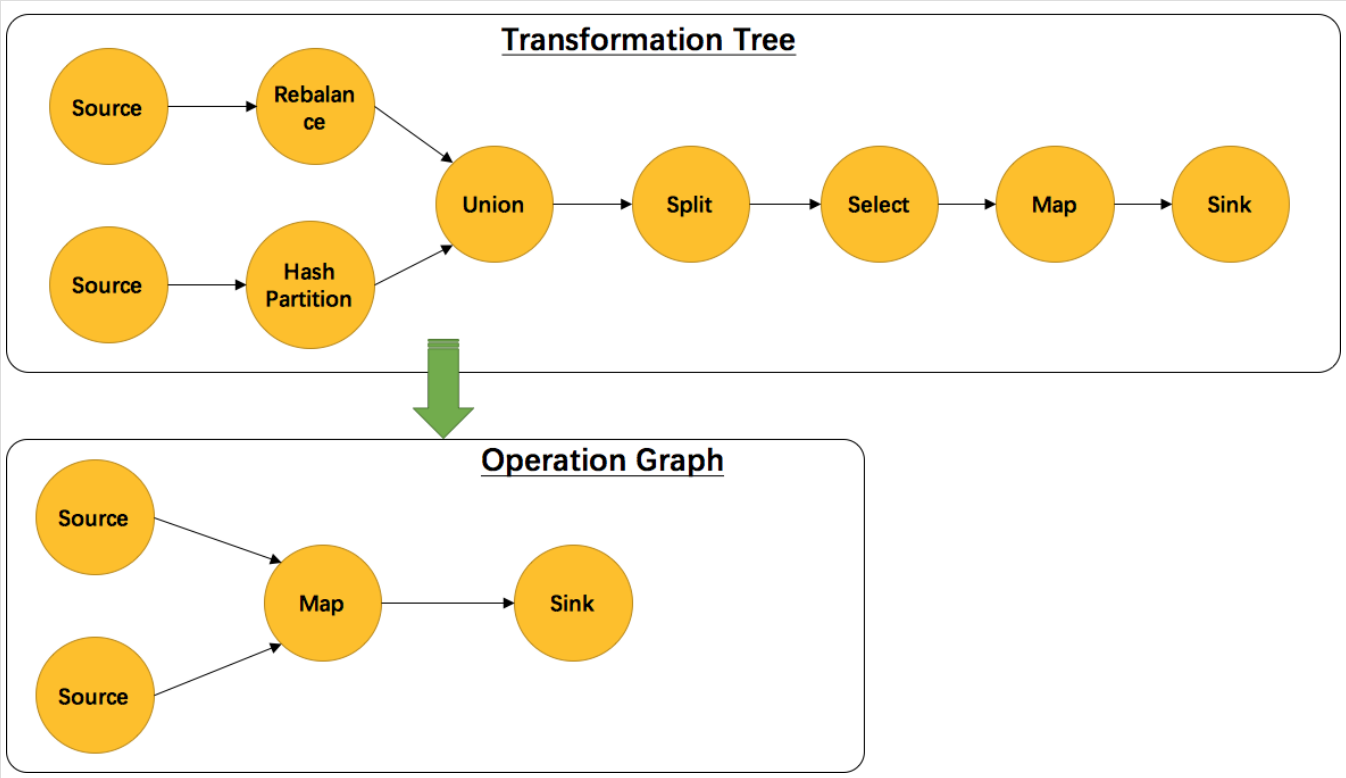
```
public <R> SingleOutputStreamOperator<R> map(MapFunction<T, R> mapper) {
  // 通过java reflection抽出mapper的返回值类型
  TypeInformation<R> outType = TypeExtractor.getMapReturnTypes(clean(mapper), getType(),
      Utils.getCallLocationName(), true);

  // 返回一个新的DataStream，SteramMap 为 StreamOperator 的实现类
  return transform("Map", outType, new StreamMap<>(clean(mapper)));
}

public <R> SingleOutputStreamOperator<R> transform(String operatorName, TypeInformation<
  // read the output type of the input Transform to coax out errors about MissingTypeInf
  transformation.getOutputType();

  // 新的transformation会连接上当前DataStream中的transformation，从而构建成一棵树
  OneInputTransformation<T, R> resultTransform = new OneInputTransformation<>(
      this.transformation,
      operatorName,
      operator,
      outTypeInfo,
      environment.getParallelism());

  @SuppressWarnings({ "unchecked", "rawtypes" })
```

```
    SingleOutputStreamOperator<R> returnStream = new SingleOutputStreamOperator(environmer

    // 所有的transformation都会存到 env 中，调用execute时遍历该list生成StreamGraph
    getExecutionEnvironment().addOperator(resultTransform);

    return returnStream;
}
```

从上方代码可以了解到，map转换将用户自定义的函数`MapFunction`包装到`StreamMap`这个Operator中，再将`StreamMap`包装到`OneInputTransformation`，最后该transformation存到env中，当调用`env.execute`时，遍历其中的transformation集合构造出StreamGraph。其分层实现如下图所示：



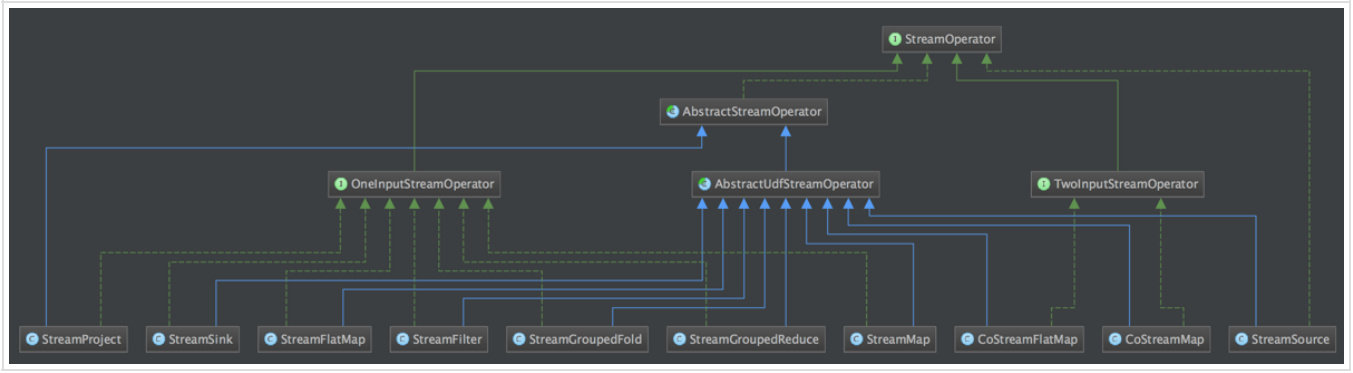
另外，并不是每一个 StreamTransformation 都会转换成 runtime 层中物理操作。有一些只是逻辑概念，比如union、split/select、partition等。如下图所示的转换树，在运行时会优化成下方的操作图。



union、split/select、partition中的信息会被写入到 Source —> Map 的边中。通过源码也可以发现，`UnionTransformation`,`SplitTransformation`,`SelectTransformation`,`PartitionTransformation`由于不包含具体的操作所以都没有StreamOperator成员变量，而其他StreamTransformation的子类基本上都有。

StreamOperator

DataStream 上的每一个 Transformation 都对应了一个 StreamOperator，StreamOperator是运行时的具体实现，会决定UDF(User-Defined Funtion)的调用方式。下图所示为 StreamOperator 的类图（点击查看大图）：



可以发现，所有实现类都继承了`AbstractStreamOperator`。另外除了 project 操作，其他所有可以执行UDF代码的实现类都继承自`AbstractUdfStreamOperator`，该类是封装了UDF的StreamOperator。UDF就是实现了`Function`接口的类，如`MapFunction`,`FilterFunction`。

生成 StreamGraph 的源码分析

我们通过在DataStream上做了一系列的转换（map、filter等）得到了StreamTransformation集合，然后通过`StreamGraphGenerator.generate`获得StreamGraph，该方法的源码如下：

```
// 构造 StreamGraph 入口函数
public static StreamGraph generate(StreamExecutionEnvironment env, List<StreamTransforma
    return new StreamGraphGenerator(env).generateInternal(transformations);
}

// 自底向上（sink->source）对转换树的每个transformation进行转换。
private StreamGraph generateInternal(List<StreamTransformation<?>> transformations) {
  for (StreamTransformation<?> transformation: transformations) {
    transform(transformation);
  }
  return streamGraph;
}

// 对具体的一个transformation进行转换，转换成 StreamGraph 中的 StreamNode 和 StreamEdge
// 返回值为该transform的id集合，通常大小为1个（除FeedbackTransformation）
private Collection<Integer> transform(StreamTransformation<?> transform) {
  // 跳过已经转换过的transformation
  if (alreadyTransformed.containsKey(transform)) {
    return alreadyTransformed.get(transform);
  }
```

```
    LOG.debug("Transforming " + transform);

    // 为了触发 MissingTypeInfo 的异常
    transform.getOutputType();

    Collection<Integer> transformedIds;
    if (transform instanceof OneInputTransformation<?, ?>) {
      transformedIds = transformOnInputTransform((OneInputTransformation<?, ?>) transform)
    } else if (transform instanceof TwoInputTransformation<?, ?, ?>) {
      transformedIds = transformTwoInputTransform((TwoInputTransformation<?, ?, ?>) transf
    } 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 {
      throw new IllegalStateException("Unknown transformation: " + transform);
    }

    // need this check because the iterate transformation adds itself before
    // transforming the feedback edges
    if (!alreadyTransformed.containsKey(transform)) {
      alreadyTransformed.put(transform, transformedIds);
    }

    if (transform.getBufferTimeout() > 0) {
      streamGraph.setBufferTimeout(transform.getId(), transform.getBufferTimeout());
    }
    if (transform.getUid() != null) {
      streamGraph.setTransformationId(transform.getId(), transform.getUid());
    }

    return transformedIds;
  }
```

最终都会调用 `transformXXX` 来对具体的StreamTransformation进行转换。我们可以看下`transformOnInputTransform(transform)`的实现：

```
  private <IN, OUT> Collection<Integer> transformOnInputTransform(OneInputTransformation<I
    // 递归对该transform的直接上游transform进行转换，获得直接上游id集合
```

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

  // 递归调用可能已经处理过该transform了
  if (alreadyTransformed.containsKey(transform)) {
    return alreadyTransformed.get(transform);
  }

  String slotSharingGroup = determineSlotSharingGroup(transform.getSlotSharingGroup(), i

  // 添加 StreamNode
  streamGraph.addOperator(transform.getId(),
      slotSharingGroup,
      transform.getOperator(),
      transform.getInputType(),
      transform.getOutputType(),
      transform.getName());

  if (transform.getStateKeySelector() != null) {
    TypeSerializer<?> keySerializer = transform.getStateKeyType().createSerializer(env.g
    streamGraph.setOneInputStateKey(transform.getId(), transform.getStateKeySelector(),
  }

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

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

  return Collections.singleton(transform.getId());
}
```

该函数首先会对该transform的上游transform进行递归转换，确保上游的都已经完成了转化。然后通过transform构造出StreamNode，最后与上游的transform进行连接，构造出StreamNode。

最后再来看下对逻辑转换（partition、union等）的处理，如下是`transformPartition`函数的源码：

```
private <T> Collection<Integer> transformPartition(PartitionTransformation<T> partition)
  StreamTransformation<T> input = partition.getInput();
  List<Integer> resultIds = new ArrayList<>();

  // 直接上游的id
  Collection<Integer> transformedIds = transform(input);
  for (Integer transformedId: transformedIds) {
    // 生成一个新的虚拟id
    int virtualId = StreamTransformation.getNewNodeId();
    // 添加一个虚拟分区节点，不会生成 StreamNode
    streamGraph.addVirtualPartitionNode(transformedId, virtualId, partition.getPartitior
    resultIds.add(virtualId);
  }
```

```
    return resultIds;
}
```

对partition的转换没有生成具体的StreamNode和StreamEdge，而是添加一个虚节点。当partition的下游transform（如map）添加edge时（调用`StreamGraph.addEdge`），会把partition信息写入到edge中。如`StreamGraph.addEdgeInternal`所示：

```
public void addEdge(Integer upStreamVertexID, Integer downStreamVertexID, int typeNumber
    addEdgeInternal(upStreamVertexID, downStreamVertexID, typeNumber, null, new ArrayList<
}
private void addEdgeInternal(Integer upStreamVertexID,
        Integer downStreamVertexID,
        int typeNumber,
        StreamPartitioner<?> partitioner,
        List<String> outputNames) {

    // 当上游是select时，递归调用，并传入select信息
    if (virtualSelectNodes.containsKey(upStreamVertexID)) {
        int virtualId = upStreamVertexID;
        // select上游的节点id
        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, outpu
    }
    // 当上游是partition时，递归调用，并传入partitioner信息
    else if (virtuaPartitionNodes.containsKey(upStreamVertexID)) {
        int virtualId = upStreamVertexID;
        // partition上游的节点id
        upStreamVertexID = virtuaPartitionNodes.get(virtualId).f0;
        if (partitioner == null) {
            partitioner = virtuaPartitionNodes.get(virtualId).f1;
        }
        addEdgeInternal(upStreamVertexID, downStreamVertexID, typeNumber, partitioner, outpu
    } else {
        // 真正构建StreamEdge
        StreamNode upstreamNode = getStreamNode(upStreamVertexID);
        StreamNode downstreamNode = getStreamNode(downStreamVertexID);

        // 未指定partitioner的话，会为其选择 forward 或 rebalance 分区。
        if (partitioner == null && upstreamNode.getParallelism() == downstreamNode.getParall
            partitioner = new ForwardPartitioner<Object>();
        } else if (partitioner == null) {
            partitioner = new RebalancePartitioner<Object>();
        }

        // 健康检查， forward 分区必须要上下游的并发度一致
        if (partitioner instanceof ForwardPartitioner) {
            if (upstreamNode.getParallelism() != downstreamNode.getParallelism()) {
```

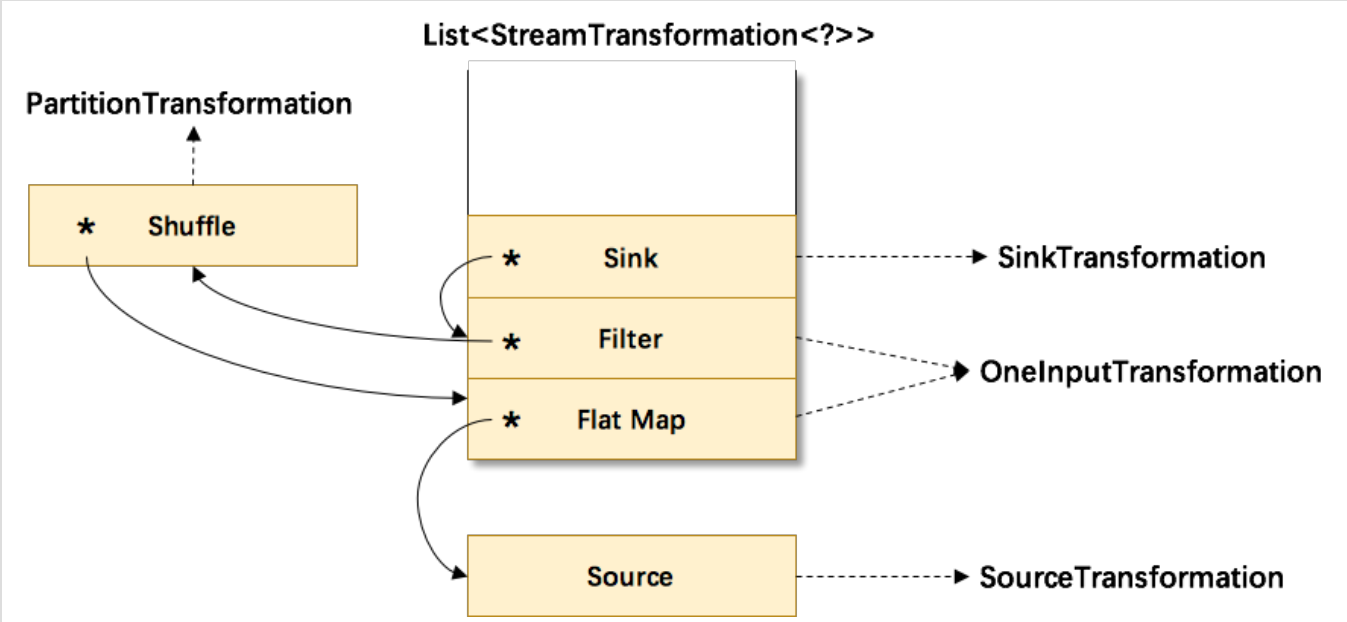
```
            throw new UnsupportedOperationException("Forward partitioning does not allow " +
                "change of parallelism. Upstream operation: " + upstreamNode + " parallelism
                ", downstream operation: " + downstreamNode + " parallelism: " + downstreamN
                " You must use another partitioning strategy, such as broadcast, rebalance,
        }
    }
    // 创建 StreamEdge
    StreamEdge edge = new StreamEdge(upstreamNode, downstreamNode, typeNumber, outputNam
    // 将该 StreamEdge 添加到上游的输出，下游的输入
    getStreamNode(edge.getSourceId()).addOutEdge(edge);
    getStreamNode(edge.getTargetId()).addInEdge(edge);
  }
}
```

实例讲解

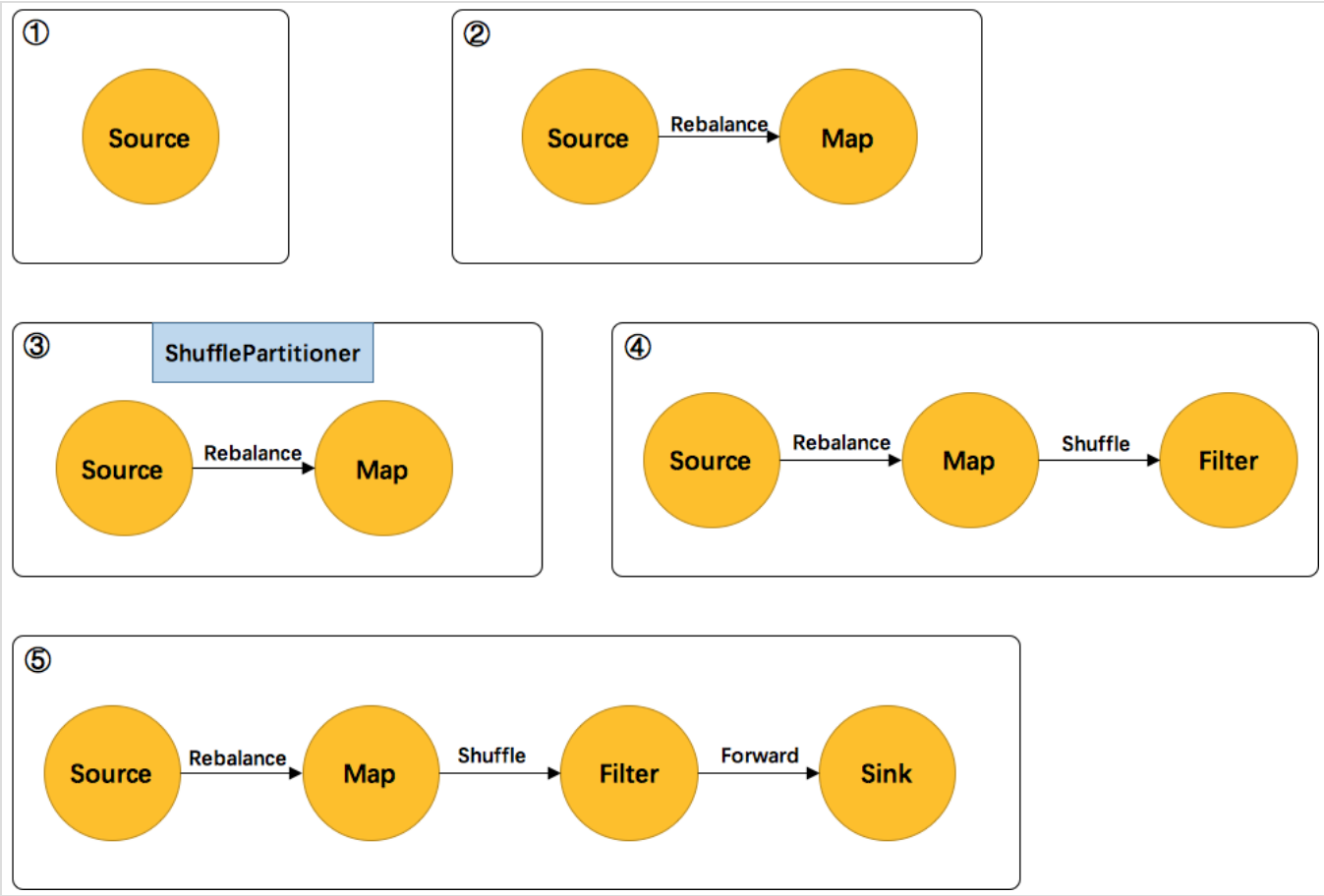
如下程序，是一个从 Source 中按行切分成单词并过滤输出的简单流程序，其中包含了逻辑转换：随机分区shuffle。我们会分析该程序是如何生成StreamGraph的。

```
DataStream<String> text = env.socketTextStream(hostName, port);
text.flatMap(new LineSplitter()).shuffle().filter(new HelloFilter()).print();
```

首先会在env中生成一棵transformation树，用`List<StreamTransformation<?>>`保存。其结构图如下：



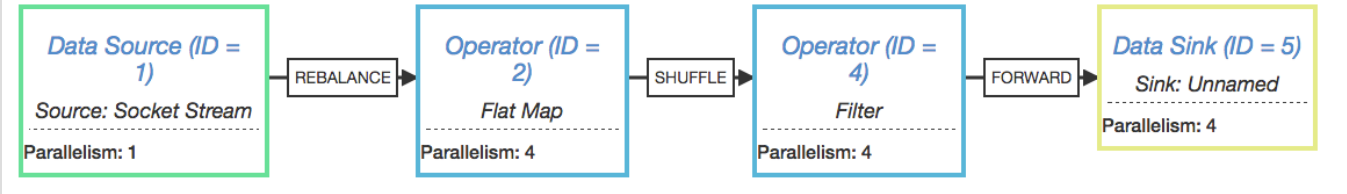
其中符号`*`为input指针，指向上游的transformation，从而形成了一棵transformation树。然后，通过调用`StreamGraphGenerator.generate(env, transformations)`来生成StreamGraph。自底向上递归调用每一个transformation，也就是说处理顺序是Source->FlatMap->Shuffle->Filter->Sink。



如上图所示：

1. 首先处理的Source，生成了Source的StreamNode。
2. 然后处理的FlatMap，生成了FlatMap的StreamNode，并生成StreamEdge连接上游Source和FlatMap。由于上下游的并发度不一样（1:4），所以此处是Rebalance分区。
3. 然后处理的Shuffle，由于是逻辑转换，并不会生成实际的节点。将partitioner信息暂存在`virtuaPartitionNodes`中。
4. 在处理Filter时，生成了Filter的StreamNode。发现上游是shuffle，找到shuffle的上游FlatMap，创建StreamEdge与Filter相连。并把ShufflePartitioner的信息写到StreamEdge中。
5. 最后处理Sink，创建Sink的StreamNode，并生成StreamEdge与上游Filter相连。由于上下游并发度一样（4:4），所以此处选择 Forward 分区。

最后可以通过 UI可视化 来观察得到的 StreamGraph。



总结

本文主要介绍了 Stream API 中 Transformation 和 Operator 的概念，以及如何根据Stream API编写的程序，构造出一个代表拓扑结构的StreamGraph的。本文的源码分析涉及到较多代码，如果有兴趣建议结合完整源码进行学习。下一篇文章将介绍 StreamGraph 如何转换成 JobGraph 的，其中设计到了图优化的技巧。