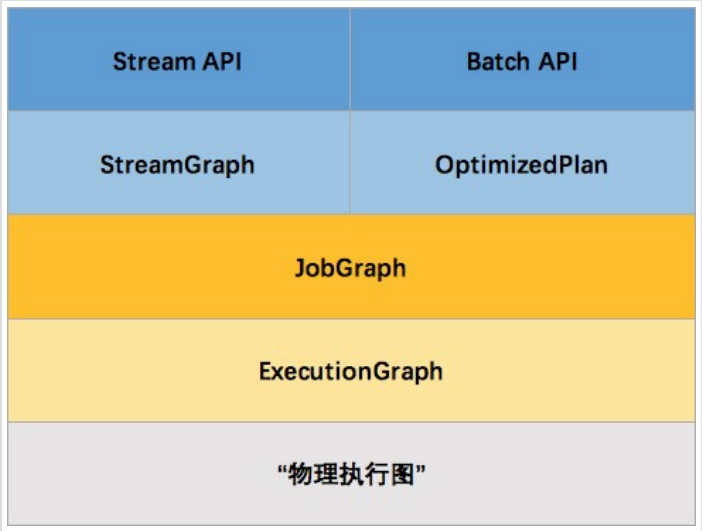
Flink数据流图的生成----StreamGraph的生成



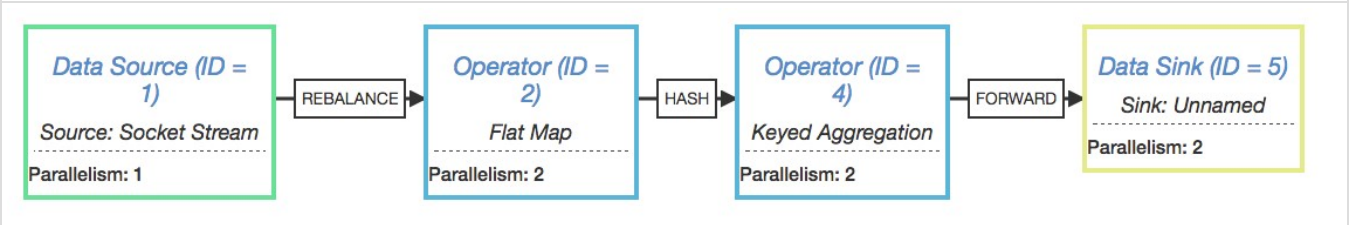
上图是完整的Flink层次图，首先我们看到，JobGraph 之上除了 StreamGraph 还有 OptimizedPlan。OptimizedPlan 是由 Batch API 转换而来的。StreamGraph 是由 Stream API 转换而来的。JobGraph 的责任就是统一 Batch 和 Stream 的图，用来描述清楚一个拓扑图的结构，并且做了 chaining 的优化。ExecutionGraph 的责任是方便调度和各个 tasks 状态的监控和跟踪，所以 ExecutionGraph 是并行化的 JobGraph。而“物理执行图”就是最终分布式在各个机器上运行着的tasks。前边介绍Flink数据流图可以分为简单执行计划一>**StreamGraph的生成**一>JobGraph的生成一>ExecutionGraph的生成一>物理执行图。

StreamGraph是在用户客户端形成的图结构，**StreamGraph**的DAG图结构由**StreamNode**和**StreamEdge**组成。

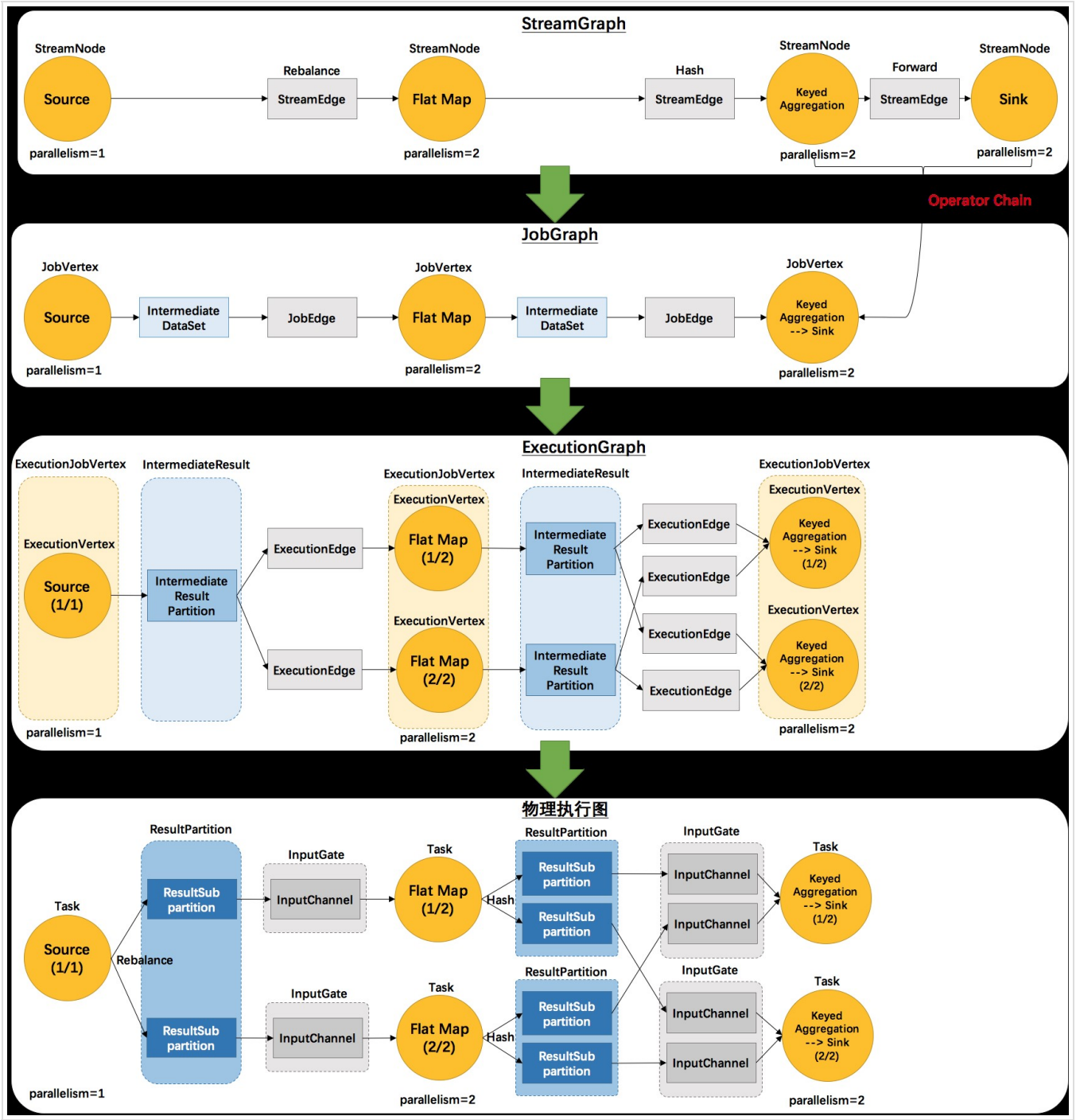
```
/**
        StreamNode
*/
private final int id;
private Integer parallelism = null;
private Long bufferTimeout = null;
private final String operatorName;
private transient StreamOperator<?> operator;
private List<OutputSelector<?>> outputSelectors;
private List<StreamEdge> inEdges = new ArrayList<StreamEdge>();
private List<StreamEdge> outEdges = new ArrayList<StreamEdge>();
addInEdge(StreamEdge inEdge)
addOutEdge(StreamEdge outEdge)
setParallelism(Integer parallelism)

/**
        StreamEdge
*/
private final String edgeId;
private final StreamNode sourceVertex;
```

```
21  private final StreamNode targetVertex;
22  private final List<String> selectedNames;
23  private StreamPartitioner<?> outputPartitioner
24  setPartitioner(StreamPartitioner<?> partitioner)
```

接下来，会以Flink 自带的 examples 包中的 SocketTextStreamWordCount文件为例分析数据图的形成过程，这是一个从 socket 流中统计单词出现次数的例子。通过flink官网给出的逻辑执行计划的可视化页面可以看到执行的StreamGraph：



为了更清晰整个数据流图的演变过程，见下图：



原理分析

整个的StreamGraph生成过程如下所示，以之前的wordcount为例，在开始生成StreamGraph之前，会得到一个Transformations的集合，该集合包括flatmap、keyed、sink。一个StreamGraph图是由StreamNode和StreamEdge组成，下面开始分析整个图的形成过程。

1. 首先得到Transformations的第一个元素flatmap，首先获取到其输入source，然后执行source对应的Transformation的类型函数，该内部通过addSource函数生成StreamNode结点，source部分就结束。

2. 然后开始回到flatmap，同样执行flatmap对应的Transformation的类型函数，接着会执行addOperator函数，然后会创建StreamNode，这样operator和StreamNode是对应的。接下来会新建一个StreamEdge()，该边会连接上下游的StreamNode。

3. 接着得到Transformations集合中的第二个元素keyed，同样先获取到其输入，即得到一个transformation(partition)，由于该Transformation的类型是PartitionTransformation，所以会生成一个虚结点，不会参与具体的物理调度。

4. 对于其他的StreamNode和StreamEdge的形成过程很类似。

源码分析

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。

```
/ 构造 StreamGraph 入口函数
public static StreamGraph generate(StreamExecutionEnvironment env, List<StreamTransf
    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);
  }
  transform.getOutputType();
  Collection<Integer> transformedIds;
  if (transform instanceof OneInputTransformation<?, ?>) {
    transformedIds = transformOnInputTransform((OneInputTransformation<?, ?>) transf
  }else if (transform instanceof TwoInputTransformation<?, ?, ?>) {
    transformedIds = transformTwoInputTransform((TwoInputTransformation<?, ?, ?>) tr
  }
```

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(OneInputTransformati
  // 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(
  // 添加 StreamNode
streamGraph.addOperator(transform.getId(),slotSharingGroup,transform.getOperator(),t
  if (transform.getStateKeySelector() != null) {
    TypeSerializer<?> keySerializer = transform.getStateKeyType().createSerializer(e
    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，通过调用addOperator()函数构造出StreamNode。但是有一些transform是不需要生成streamNode的，比如说union、split/select、partition等，这时候它们会调用transformPartition()：

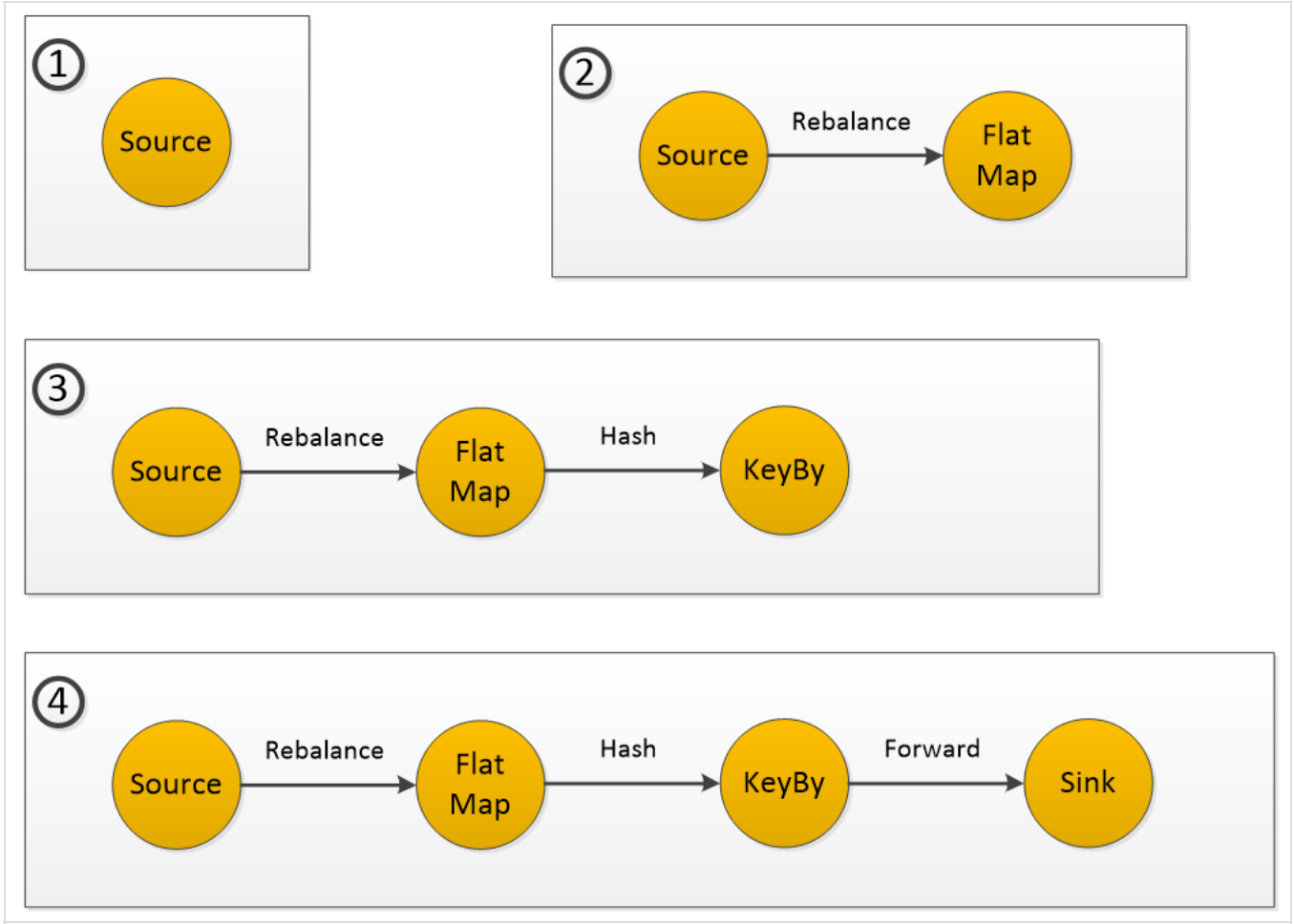
```
private <T> Collection<Integer> transformPartition(PartitionTransformation<T> partit
  StreamTransformation<T> input = partition.getInput();
  List<Integer> resultIds = new ArrayList<>();
  // 直接上游的id
  Collection<Integer> transformedIds = transform(input);
  for (Integer transformedId: transformedIds) {
    // 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.getParti
    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。接下来当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，具体的代码如下：

```
public void addEdge(Integer upStreamVertexID, Integer downStreamVertexID, int typeNu
  addEdgeInternal(upStreamVertexID, downStreamVertexID, typeNumber, null, new ArrayL
}
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
```

```
11      upStreamVertexID = virtualSelectNodes.get(virtualId).f0;
12      if (outputNames.isEmpty()) {
13        // selections that happen downstream override earlier selections
14        outputNames = virtualSelectNodes.get(virtualId).f1;
15      }
16      addEdgeInternal(upStreamVertexID, downStreamVertexID, typeNumber, partitioner, c
17    }
18    // 当上游是partition时，递归调用，并传入partitioner信息
19    else . . . . . .
20    else {
21      // 真正构建StreamEdge
22      StreamNode upstreamNode = getStreamNode(upStreamVertexID);
23      StreamNode downstreamNode = getStreamNode(downStreamVertexID);
24      // 未指定partitioner的话，会为其选择 forward 或 rebalance 分区。
25      if (partitioner == null && upstreamNode.getParallelism() == downstreamNode.getPa
26        partitioner = new ForwardPartitioner<Object>();
27      } else if (partitioner == null) {
28        partitioner = new RebalancePartitioner<Object>();
29      }
30      // 健康检查， forward 分区必须要上下游的并发度一致
31      if (partitioner instanceof ForwardPartitioner) {
32        if (upstreamNode.getParallelism() != downstreamNode.getParallelism()) {
33          . . . . .. .
34        }
35      }
36      // 创建 StreamEdge
37      StreamEdge edge = new StreamEdge(upstreamNode, downstreamNode, typeNumber, outpu
38      // 将该 StreamEdge 添加到上游的输出，下游的输入
39      getStreamNode(edge.getSourceId()).addOutEdge(edge);
40      getStreamNode(edge.getTargetId()).addInEdge(edge);
41    }
42  }
```

总结



- 首先处理的Source，生成了Source的StreamNode。
- 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。
- 处理的keyBy，生成了KeyBy的StreamNode，并且生成StreamEdge连接上游FlatMap和KeyBy。由于该操作是根据key进行hash计算，然后分组，所以此处是Hash分区。
- 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。

进行转换操作后，利用流分区器来精确控制数据流向.

Stream Graph生成原理： 链接: https://pan.baidu.com/s/1jgNDpF-boi3TAeTwp5EsgA 提取码: kk7s

Stream Graph源码分析：链接: https://pan.baidu.com/s/1aLAt-TzrBYk-x4mcLeyWxw 提取码: gfxz