Flink数据流图的生成----简单执行计划的生成

Flink的数据流图的生成主要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。其中前三个(ExecutionGraph的之前都是在client上生成的)。ExectuionGraph是JobGraph的并行版本，是在JobManager(master)端生成的。而物理执行图只是一个抽象的概念，其具体的执行是在多个slave上并行执行的。

> 简单执行计划：根据用户程序加载配置参数，为形成数据DAG图准备条件

> StreamGraph：是根据用户通过 Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。

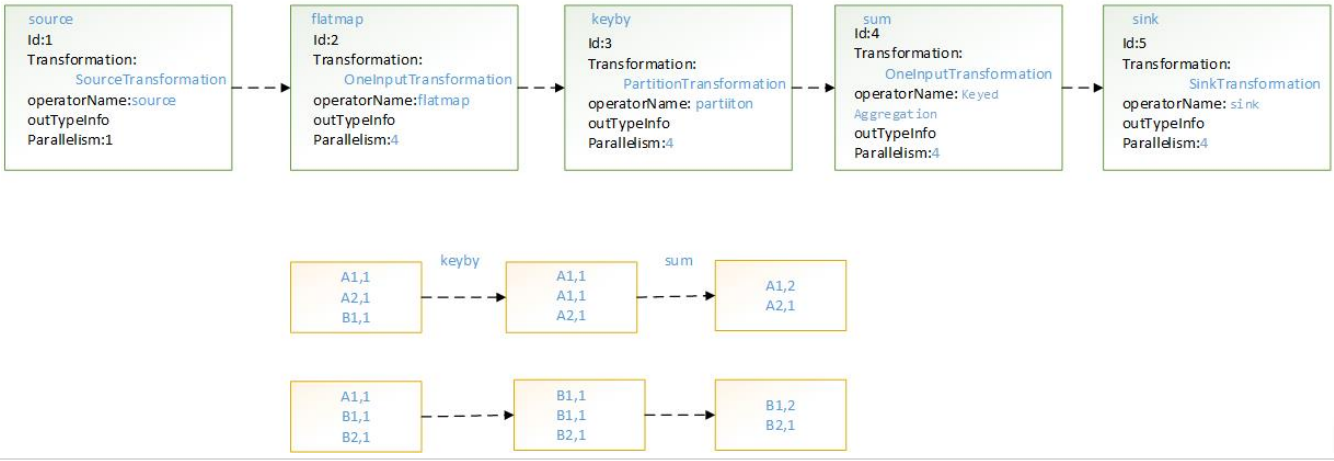
> JobGraph：StreamGraph经过优化后生成了 JobGraph，提交给 JobManager 的数据结构。主要的优化的部分是：将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化/反序列化/传输消耗。

> ExecutionGraph：JobManager 根据 JobGraph 生成ExecutionGraph。ExecutionGraph是JobGraph的并行化版本，是调度层最核心的数据结构。

> 物理执行图：JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个TaskManager 上部署 Task 后形成的“图”，并不是一个具体的数据结构。

原理分析

```
DataStream<String> text = env.readTextFile("D:\\test.txt")
DataStream<Tuple2<String, Integer>> counts =text.flatMap(new Tokenizer()).keyBy(0).sum(1);
counts.print();
env.execute("Streaming WordCount");
public static final class Tokenizer implements FlatMapFunction<String, Tuple2<String, Integer>>
{
      @Override
      public void flatMap(String value, Collector<Tuple2<String, Integer>> out)
                  throws Exception {
            String[] tokens = value.toLowerCase().split("\\W+");
            for (String token : tokens) {
                  if (token.length() > 0) {
                        out.collect(new Tuple2<String, Integer>(token, 1));
                  }
            }
      }
}
```



Flink效仿了传统的关系型数据库在运行SQL时生成运行计划并对其进行优化的思路。在具体生成数据流图之前会生成一个运行计划，当程序执行execute方法时，才具体生成数据流图运行任务。

首先Flink会加载数据源，读取配置文件，获取配置参数parallelism等，为source 的transformation对应的类型是SourceTransformation,opertorName是source，然后进入flatmap，用户重写了内置的flatmap内核函数，按照空格进行划分单词，获取到其各种配制参数，parallelism以及输出的类型封装Tuple2<String,Integer>，以及operatorName是Flat Map，其对应的Transformation类型是OneInputTransformation。然后开始keyby(0)，其中0指的是Tuple2<String, Integer>中的String，其意义是按照word进行重分区，其对应的parallelism是4，operatorName 是 partition ， Transformation 的 类 型 是 PartitionTransformation ， 输 出 类 型 的 封 装 是 Tuple2<String, Integer>。接着sum(1)，该函数的作用是把相同的key对应的值进行加1操作。其对应的parallelism是4，operatorName是keyed Aggregation，对应的输出类型封装是Tuple2<String, Integer>，Transformation的类型是OneInputTransformation。最后是进行结果输出处理sink，对应的parallelism是4，输出类型的封装是Tuple2<String, Integer>，对应的operatorName是sink，对应的Transformation类型是SinkTransformation。

源码

以**WordCount.java**为例：

```java
public class WordCount {
    private static  Logger LOG = LoggerFactory.getLogger(WordCount.class);
    private static SimpleDateFormat df=new SimpleDateFormat("yyyy/MM/dd HH:mm:ss:SSS
    public static long time=0;
    public static void main(String[] args) throws Exception {
        // Checking input parameters
        LOG.info("set up the execution environment: start= "+df.format(System.currer
        final ParameterTool params = ParameterTool.fromArgs(args);
```

```
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecuti
        env.getConfig().setGlobalJobParameters(params);
        DataStream<String> text;
        if (params.has("input")) {
            text = env.readTextFile(params.get("input"));
        } else {
            text = env.fromElements(WordCountData.WORDS);
        }
        DataStream<Tuple2<String, Integer>> counts =
            text.flatMap(new Tokenizer()).keyBy(0).sum(1);
        if (params.has("output")) {
            counts.writeAsText(params.get("output"));
        } else {
            System.out.println("Printing result to stdout. Use --output to specify o
            counts.print();
        }
        env.execute("Streaming WordCount");
    }
    public static final class Tokenizer implements FlatMapFunction<String, Tuple2<St
        private static final long serialVersionUID = 1L;
        @Override
        public void flatMap(String value, Collector<Tuple2<String, Integer>> out)
                throws Exception {
            String[] tokens = value.toLowerCase().split("\W+");
            for (String token : tokens) {
                if (token.length() > 0) {
                    out.collect(new Tuple2<String, Integer>(token, 1));
                }
            }
        }
    }
}
```

Flink在程序执行时，首先会获取程序需要的执行计划，类似数据的惰性加载，当具体执行execute()函数时，程序才会具体真正执行。首先执行

```
text = env.readTextFile(params.get("input"));
```

该函数的作用是加载数据文件，获取数据源，形成source的属性信息，包括source的Transformation类型、并行度、输出类型等。源码如下：

```
public final <OUT> DataStreamSource<OUT> readTextFile(OUT... data) {
        TypeInformation<OUT> typeInfo;
        try {
            typeInfo = TypeExtractor.getForObject(data[0]);
        }
        return fromCollection(Arrays.asList(data), typeInfo);
}

public <OUT> DataStreamSource<OUT> fromCollection(Collection<OUT> data, TypeInformat
        FromElementsFunction.checkCollection(data, typeInfo.getTypeClass());
```

```
        SourceFunction<OUT> function;
        try {
            function = new FromElementsFunction<>(typeInfo.createSerializer(getConfi
        }
        catch (IOException e) {
            throw new RuntimeException(e.getMessage(), e);
        }
        return addSource(function, "Collection Source", typeInfo).setParallelism(1);
    }

    public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, Strir
        boolean isParallel = function instanceof ParallelSourceFunction;
        clean(function);
        StreamSource<OUT, ?> sourceOperator;
        if (function instanceof StoppableFunction) {
            sourceOperator = new StoppableStreamSource<>(cast2StoppableSourceFunctic
        } else {
            sourceOperator = new StreamSource<>(function);
        }
        return new DataStreamSource<>(this, typeInfo, sourceOperator, isParallel, sc
    }

public DataStreamSource(StreamExecutionEnvironment environment,
            TypeInformation<T> outTypeInfo, StreamSource<T, ?> operator,
            boolean isParallel, String sourceName) {
        super(environment, new SourceTransformation<>(sourceName, operator, outTypeI
        this.isParallel = isParallel;
        if (!isParallel) {
            setParallelism(1);
        }
    }
```

从上述代码可知，这部分会执行addSource()函数，通过new StreamSource，**生成source的operator**，然后通过new DataStreamSource生成SourceTransformation，获取并行度等。然后就是执行flatmap函数text.flatMap(new Tokenizer())，该函数内和source类似，也是获取Transformation类型、并行度、输出类型等。

```
public <R> SingleOutputStreamOperator<R> flatMap(FlatMapFunction<T, R> flatMapper) {
    TypeInformation<R> outType = TypeExtractor.getFlatMapReturnTypes(clean(flatMappe
                                                                     getType(), Util
    SingleOutputStreamOperator result = transform("Flat Map", outType, new StreamFla
    return result;
}

public <R> SingleOutputStreamOperator<R> transform(String operatorName, TypeInformat
    transformation.getOutputType();
    OneInputTransformation<T, R> resultTransform = new OneInputTransformation<>(
        this.transformation,
        operatorName,
        operator,
        outTypeInfo,
        environment.getParallelism());
    SingleOutputStreamOperator<R> returnStream = new SingleOutputStreamOperator(envi
```

```
    getExecutionEnvironment().addOperator(resultTransform);
    return returnStream;
}
```

对应该operator，其Transformation的类型是OneInputTransformation类型，对应着属性信息有该operator的名称，输出类型，执行的并行度等，然后会执行addOperator函数将该operator (flatmap)加入到执行环境中，以便后续执行。 接下来执行.keyBy(0)，该函数的作用就是重分区，把word的单词作为key，然后按照key相同的放在一个分区内，方便执行。该函数的内部是形成其transformation类型(PartitionTransformation)，以及相关的属性信息等。

```
private KeyedStream<T, Tuple> keyBy(Keys<T> keys) {
    return new KeyedStream<>(this, clean(KeySelectorUtil.getSelectorForKeys(keys,get
}
public KeyedStream(DataStream<T> dataStream, KeySelector<T, KEY> keySelector, TypeIr
    super(
        dataStream.getExecutionEnvironment(),
        new PartitionTransformation<>(
            dataStream.getTransformation(),
            new KeyGroupStreamPartitioner<>(keySelector, StreamGraphGenerator.DEFAUL
    this.keySelector = keySelector;
    this.keyType = validateKeyType(keyType);
    LOG.info("part of keyBy(partition): end= "+df.format(System.currentTimeMillis())
}
```

在上述代码中，keyby会创建一个PartitionTransformation，作为其Transformation的类型，该在类中会得到input(输入数据)，以及partioner分区器。同样会得到执行的并行度、输出类型等信息。 接下来是sum(1)，该函数的作用是按照keyby的word作为key，进行加1操作。源码如下：

```
protected SingleOutputStreamOperator<T> aggregate(AggregationFunction<T> aggregate) {
    StreamGroupedReduce<T> operator = new StreamGroupedReduce<T>( clean(aggregate), g
    return transform("Keyed Aggregation", getType(), operator);
}
```

在上述的代码中，可以看到，该operator的所属的类型是StreamGroupedReduce，对着着核心方法reduce()，通过new该对象，会获取到其operator的名称等属性信息，然后执行transform()函数，该函数的代码之前已经给出，主要的作用是创建一个该operator的Transformation类型，即OneInputTransformtion，会得到并行度、输出类型等属性信息，然后执行addOperator()函数，将operator加入执行环境，让能起能够具体执行任务。 接下来会对结果进行输出，将执行counts.print()，该函数内部对应着一个operator，即sink(具体的逻辑就是结果输出)，源码如下：

```
public DataStreamSink<T> print() {
    PrintSinkFunction<T> printFunction = new PrintSinkFunction<>();
    return addSink(printFunction);
}
public DataStreamSink<T> addSink(SinkFunction<T> sinkFunction) {
    transformation.getOutputType();
    if (sinkFunction instanceof InputTypeConfigurable) {
        ((InputTypeConfigurable) sinkFunction).setInputType(getType(), getExecutionC
```

```
9       }
10      StreamSink<T> sinkOperator = new StreamSink<>(clean(sinkFunction));
11      DataStreamSink<T> sink = new DataStreamSink<>(this, sinkOperator);
12      getExecutionEnvironment().addOperator(sink.getTransformation());
13      return sink;
14  }
15  protected DataStreamSink(DataStream<T> inputStream, StreamSink<T> operator) {
16      this.transformation = new SinkTransformation<T>(inputStream.getTransformation(),
17  }
```

print()函数内部只有一个方法：addSink()，其功能和addSource()一样，首先会创建一个StreamSink，生成一个operator对象，然后创建DataStreamSink，该类中会创建一个该operator的Transformation类型即SinkTransformtion，会得到该operator的名称，并行度，输出类型等属性信息。同样，会执行addOperator()函数，该函数的作用将该operator加入到env执行环境中，用来进行具体操作。

该部分只是生成简单的执行计划，并没有生成具体的图结构。