toc: true title: 《从1到100深入学习Flink》——分析 Streaming WordCount 程序的执行过程 date: 2019-02-17 tags:

- Flink
- 大数据
- 流式计算

前言

前一篇文章写了 《从1到100深入学习Flink》——分析 Batch WordCount 程序的执行过程，主要是批处理的情况，当我们的 Job 类型是 STREAMING 时，执行情况会有一点区别，所以我们再来讲解一下流程序的 WordCount 的执行流程。

https://github.com/zhisheng17/flink-learning/blob/master/flink-learning-examples/src/main/java/com/zhisheng/examples/streaming/wordcount/Main.java

再来看下程序：

```
package com.zhisheng.examples.wordcount.streaming.wordcount;

/**
 * blog：http://www.54tianzhisheng.cn/
 * 微信公众号：zhisheng
 */
public class Main {
    public static void main(String[] args) throws Exception {
        //创建流运行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

env.getConfig().setGlobalJobParameters(ParameterTool.fromArgs(args)
);

        env.fromElements(WORDS)
                .flatMap(new FlatMapFunction<String, Tuple2<String,
Integer>>() {
                    @Override
                    public void flatMap(String value,
```

```
Collector<Tuple2<String, Integer>> out) throws Exception {
                        String[] splits =
value.toLowerCase().split("\\W+");

                        for (String split : splits) {
                            if (split.length() > 0) {
                                out.collect(new Tuple2<>(split,
1));
                            }
                        }
                    }
                })
                .keyBy(0)
                .reduce(new ReduceFunction<Tuple2<String, Integer>>
() {
                    @Override
                    public Tuple2<String, Integer>
reduce(Tuple2<String, Integer> value1, Tuple2<String, Integer>
value2) throws Exception {
                        return new Tuple2<>(value1.f0, value1.f1 +
value1.f1);
                    }
                })
                .print();

        //Streaming 程序必须加这个才能启动程序，否则不会有结果
        env.execute("zhisheng —— word count streaming demo");
    }

    private static final String[] WORDS = new String[]{"具体的 words
看上面 GitHub 源码"};
    };
}
```

分析

不知道你有没有发现和前一篇文章的代码有什么区别呢？

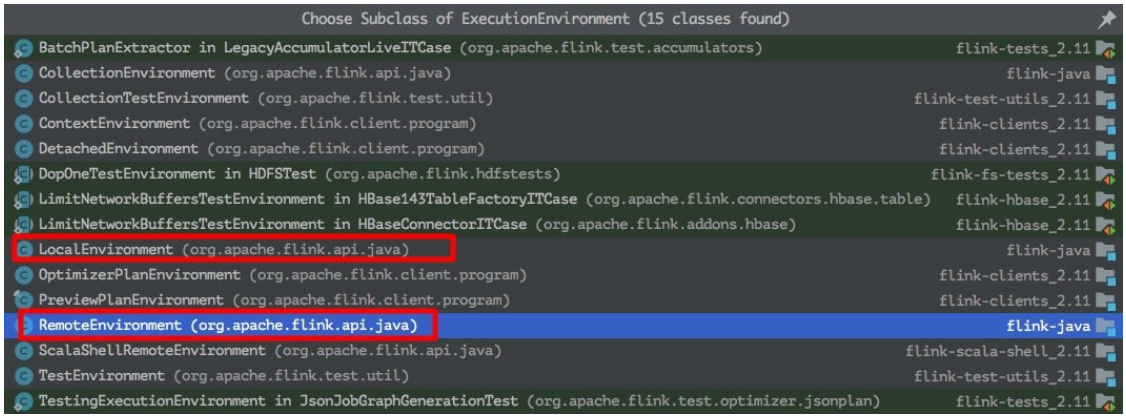
- 使用了 StreamExecutionEnvironment，创建流程序运行环境
- 使用了 keyBy，而不是 groupBy
- 使用了 env.execute()，而之前是没加这个的

看到我们这段代码如果发现了这些区别的话，那么我们就逐一的来分析一波：

1、为什么使用 StreamExecutionEnvironment？和 ExecutionEnvironment 有什么区别呢？

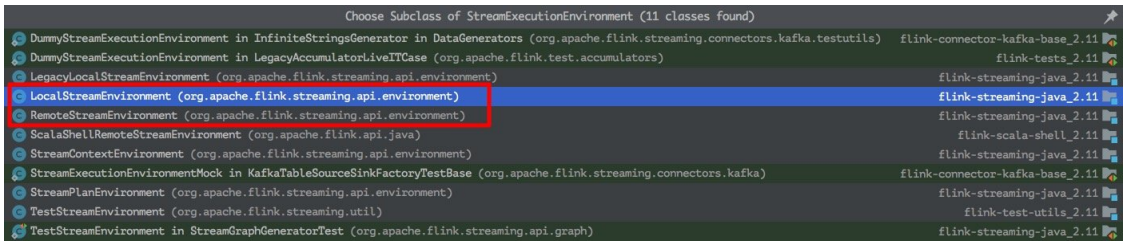
要回答这个问题，我们先分别来了解下这两个类：

ExecutionEnvironment：这是一个抽象的类，代表程序执行的上下文，它的实现类有如下图这些



一般用的比较多的有：LocalEnvironment、RemoteEnvironment，其中 LocalEnvironment 是在当前的 JVM 中运行（本地运行的都是走 LocalEnvironment），RemoteEnvironment 的话就是利用远程安装好的 Flink 环境（可以本地连远程的 Flink 地址）。

StreamExecutionEnvironment：StreamExecutionEnvironment 是执行流程序的上下文，它也是一个抽象的类，实现类有：



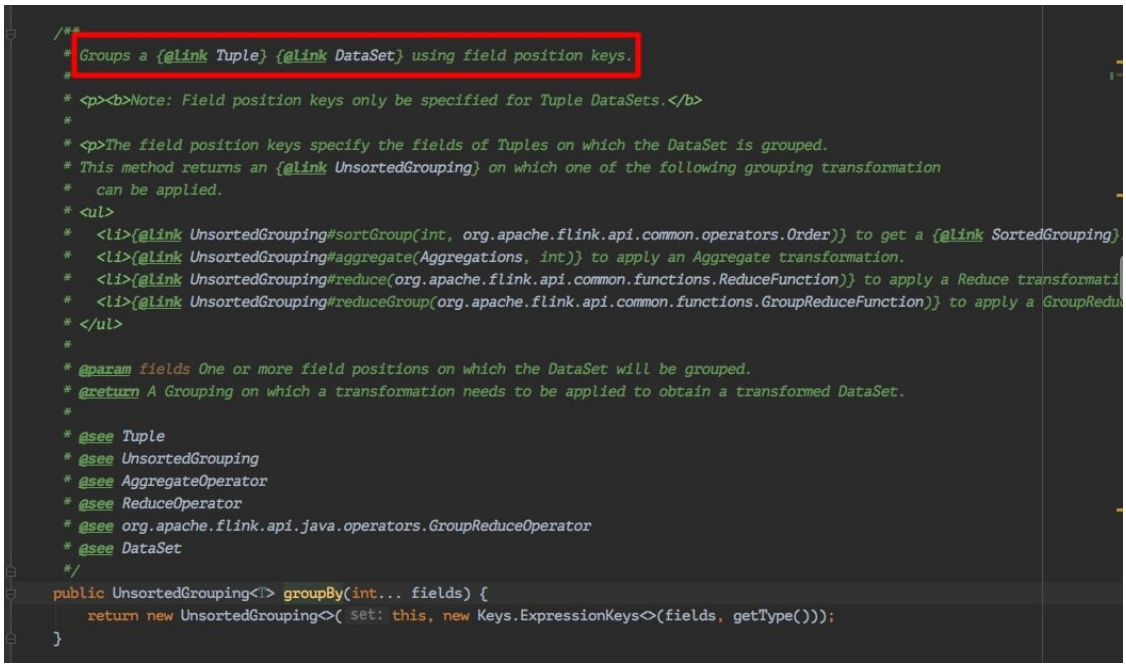
同样，LocalStreamEnvironment、RemoteStreamEnvironment 也是和上面的作用类似！

它们两的区别最大的就是一个是负责批程序的执行上下文，一个是负责流程序的执行上下文。

另外 getExecutionEnvironment 方法会根据你的执行环境自动创建一个 ExecutionEnvironment（StreamExecutionEnvironment），如果你是在 IDE 中运行你的程序，getExecutionEnvironment 方法方法会返回一个 LocalEnvironment（LocalStreamEnvironment），如果你对程序打成 Jar 包之后使用命令行提交到集群运行的话，getExecutionEnvironment 方法将会返回一个 RemoteEnvironment（RemoteStreamEnvironment）。

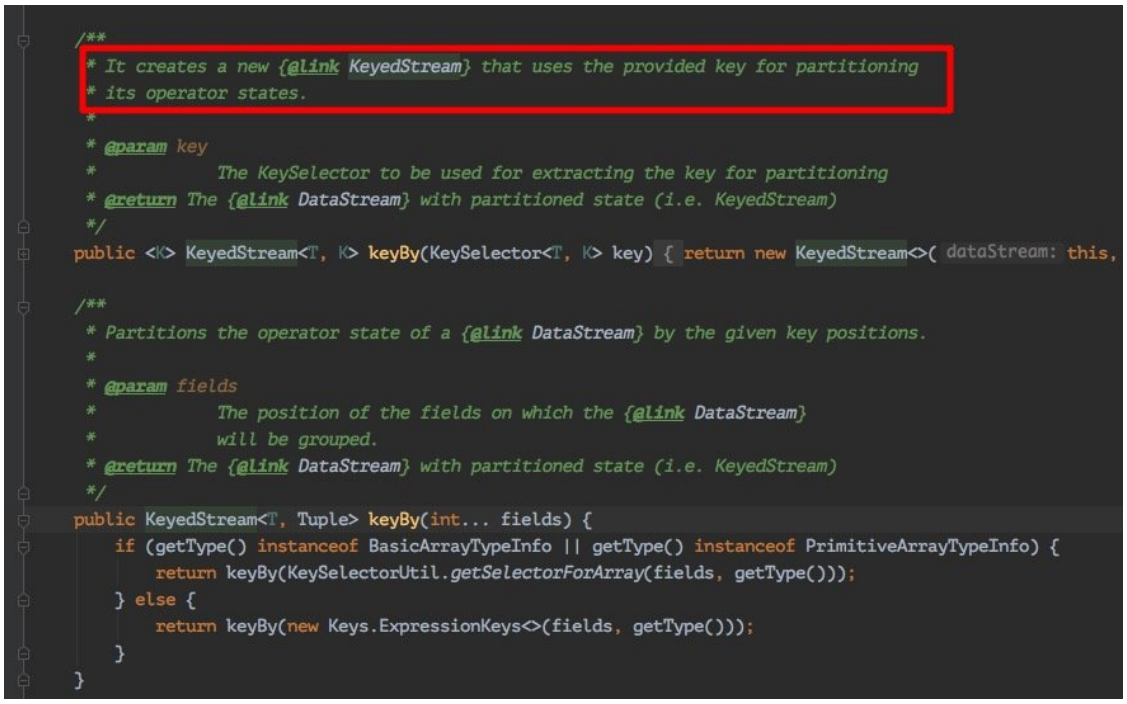
2、为什么使用 keyBy，而不是 groupBy?

其实你可以在 ide 中将批程序 groupBy 后的结果返回，你会发现返回的是一个 DataSet 类型的数据，而在流程序中 keyBy 后返回的结果数据类型是 KeyedStream（继承自 DataStream）。



```
/**
 * Groups a {@link Tuple} {@link DataSet} using field position keys.
 *
 * <p><b>Note: Field position keys only be specified for Tuple DataSets.</b>
 *
 * <p>The field position keys specify the fields of Tuples on which the DataSet is grouped.
 * This method returns an {@link UnsortedGrouping} on which one of the following grouping transformation
 *   can be applied.
 * <ul>
 *   <li>{@link UnsortedGrouping#sortGroup(int, org.apache.flink.api.common.operators.Order)} to get a {@link SortedGrouping}
 *   <li>{@link UnsortedGrouping#aggregate(Aggregations, int)} to apply an Aggregate transformation.
 *   <li>{@link UnsortedGrouping#reduce(org.apache.flink.api.common.functions.ReduceFunction)} to apply a Reduce transformati
 *   <li>{@link UnsortedGrouping#reduceGroup(org.apache.flink.api.common.functions.GroupReduceFunction)} to apply a GroupRedu
 * </ul>
 *
 * @param fields One or more field positions on which the DataSet will be grouped.
 * @return A Grouping on which a transformation needs to be applied to obtain a transformed DataSet.
 *
 * @see Tuple
 * @see UnsortedGrouping
 * @see AggregateOperator
 * @see ReduceOperator
 * @see org.apache.flink.api.java.operators.GroupReduceOperator
 * @see DataSet
 */
public UnsortedGrouping<T> groupBy(int... fields) {
    return new UnsortedGrouping<>( set: this, new Keys.ExpressionKeys<>(fields, getType()));
}
```

这就说明了 groupBy 就是 DataSet 中的 API，只适合批程序，



```java
/**
 * It creates a new {@link KeyedStream} that uses the provided key for partitioning
 * its operator states.
 *
 * @param key
 *            The KeySelector to be used for extracting the key for partitioning
 * @return The {@link DataStream} with partitioned state (i.e. KeyedStream)
 */
public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key) { return new KeyedStream<>( dataStream: this,

/**
 * Partitions the operator state of a {@link DataStream} by the given key positions.
 *
 * @param fields
 *            The position of the fields on which the {@link DataStream}
 *            will be grouped.
 * @return The {@link DataStream} with partitioned state (i.e. KeyedStream)
 */
public KeyedStream<T, Tuple> keyBy(int... fields) {
    if (getType() instanceof BasicArrayTypeInfo || getType() instanceof PrimitiveArrayTypeInfo) {
        return keyBy(KeySelectorUtil.getSelectorForArray(fields, getType()));
    } else {
        return keyBy(new Keys.ExpressionKeys<>(fields, getType()));
    }
}
```

keyBy 是 DataStream 中的 API，只适合流程序。

3、为什么使用 env.execute()，而之前是没使用?

其实在上一篇文章中我们分析了批程序中的 print 方法内部其实是有调用 env.execute() 方法，但是在外部显示的调用 env.execute() 其实会报错的，但是在流程序中是一定要有 env.execute() 这个方法的，代表开始执行整个程序。

注意：流程序的 print 方法和批程序的 print 方法也还是有点区别的，上一篇文章中我们已经分析过了批程序的 print 方法内部实现，这里我们先看下流程序的 print 方法：

```java
public DataStreamSink<T> print() {
    PrintSinkFunction<T> printFunction = new PrintSinkFunction<>();
    return addSink(printFunction).name("Print to Std. Out");
}
```

可以发现它先创建一个 PrintSinkFunction 实例，然后将实例对象直接传进方法 addSink 了，其中 PrintSinkFunction 是继承了 RichSinkFunction，addSink 方法如下：

```
public DataStreamSink<T> addSink(SinkFunction<T> sinkFunction) {

    // read the output type of the input Transform to coax out
errors about MissingTypeInfo
    transformation.getOutputType();

    // configure the type if needed
    if (sinkFunction instanceof InputTypeConfigurable) {
        ((InputTypeConfigurable)
sinkFunction).setInputType(getType(), getExecutionConfig());
    }

    StreamSink<T> sinkOperator = new StreamSink<>
(clean(sinkFunction));

    DataStreamSink<T> sink = new DataStreamSink<>(this,
sinkOperator);


getExecutionEnvironment().addOperator(sink.getTransformation());
    return sink;
}
```

最后再执行 env.execute() 方法，下面我们好好分析下这个方法里面干了啥?

流程序中的 env.execute() 方法

```
public JobExecutionResult execute(String jobName) throws Exception
{
    return executeInternal(jobName, false,
SavepointRestoreSettings.none()).getJobExecutionResult();
}

protected abstract JobSubmissionResult executeInternal(String
jobName, boolean detached, SavepointRestoreSettings
savepointRestoreSettings) throws Exception;
```

这个 executeInternal 方法是一个抽象方法，它在 LocalStreamEnvironment 中有实现如下

```
protected JobSubmissionResult executeInternal(String jobName,

boolean detached, SavepointRestoreSettings
savepointRestoreSettings) throws Exception {
    //将流程序转换成 StreamGraph
    StreamGraph streamGraph = getStreamGraph();

    streamGraph.setJobName(jobName);
    //将 StreamGraph 转换成 JobGraph
    JobGraph jobGraph = streamGraph.getJobGraph();
    jobGraph.setAllowQueuedScheduling(true);
    //创建 MiniCluster 并启动
    MiniCluster miniCluster = prepareMiniCluster(jobGraph);

    try {
        //运行 job
        return miniCluster.executeJob(jobGraph, detached);
    }
    finally {
        //关闭资源
        transformations.clear();
        if (!detached) {
            miniCluster.close();
        } else {
            this.submitMapping.put(jobGraph.getJobID(),
miniCluster);
        }
    }
}
```

这里主要有五步：

- 将流程序转换成 StreamGraph
- 将 StreamGraph 转换成 JobGraph
- 创建 MiniCluster 并启动
- 运行 job
- 关闭资源

下面分别讲解下这几个：

将流程序转换成 StreamGraph

```
public StreamGraph getStreamGraph() {

    if (transformations.size() <= 0) {
        throw new IllegalStateException("No operators defined in
streaming topology. Cannot execute.");
    }

    if (JobType.STREAMING.equals(jobType)) {//流程序
        //通过给定的 StreamTransformation 遍历生成 StreamGraph
        return
StreamGraphGenerator.generate(StreamGraphGenerator.Context.buildStr
eamProperties(this), transformations);
    } else if (JobType.BATCH.equals(jobType)) {//批程序
        return
StreamGraphGenerator.generate(StreamGraphGenerator.Context.buildBat
chProperties(this), transformations);
    } else {
        throw new UnsupportedOperationException("Not support the "
+ jobType + " job type");
    }
}
```

上面中先会判断 transformations 的个数是否大于 0，如果不大于 0 就会抛出异常，然后判断 Job 的类型是 STREAMING（流程序）还是 BATCH（批程序），他们的区别就是一个使用 buildStreamProperties 方法来构建流程序的配置，另一个使用 buildBatchProperties 来构建批程序的配置，接着都是利用 StreamGraphGenerator 来生成 StreamGraph。

```
public static StreamGraph generate(Context context,
List<StreamTransformation<?>> transformations) {
    return new
StreamGraphGenerator(context).generateInternal(transformations);
}

/**
 * This starts the actual transformation, beginning from the sinks.
 * //todo：讲解如何转换成 StreamGraph
 */
private StreamGraph generateInternal(List<StreamTransformation<?>>
transformations) {
    //循环遍历 transformations
    for (StreamTransformation<?> transformation: transformations) {
        transform(transformation);
    }

    // 为算子设置默认的资源
```

```
    // 为算子设置默认的资源
    boolean needToSetDefaultResources = false;

    if (context.getDefaultResources() == null ||
ResourceSpec.DEFAULT.equals(context.getDefaultResources())) {
        for (StreamNode node : streamGraph.getStreamNodes()) {
            ResourceSpec resources = node.getMinResources();
            if (resources != null &&
!ResourceSpec.DEFAULT.equals(resources)) {
                needToSetDefaultResources = true;
                break;
            }
        }
    } else {
        needToSetDefaultResources = true;
    }

    if (needToSetDefaultResources) {
        ResourceSpec defaultResource =
context.getDefaultResources();
        if (defaultResource == null ||
ResourceSpec.DEFAULT.equals(defaultResource)) {
            defaultResource = context.getGlobalDefaultResources();
        }

        for (StreamNode node : streamGraph.getStreamNodes()) {
            ResourceSpec resources = node.getMinResources();
            if (resources == null ||
ResourceSpec.DEFAULT.equals(resources)) {
                node.setResources(defaultResource,
defaultResource);
            }
        }
    }

    return streamGraph;
}
```

上面的代码先将给定的 StreamTransformation 转换，然后给算子设置默认的资源，如何解析转换的这不是本篇文章的主题，后面会写一篇的，这里我们知道它会将这些转换成 streamGraph 就行。

将 StreamGraph 转换成 JobGraph

```
JobGraph jobGraph = streamGraph.getJobGraph();

public JobGraph getJobGraph() {
```

```
public JobGraph getJobGraph() {
    ...


    return StreamingJobGraphGenerator.createJobGraph(this);
}

public static JobGraph createJobGraph(StreamGraph streamGraph) {
    return new
StreamingJobGraphGenerator(streamGraph).createJobGraph();
}

private JobGraph createJobGraph() {
    //向 jobGraph 添加 streamGraph 中的配置

jobGraph.addCustomConfiguration(streamGraph.getCustomConfiguration(
));

    // Generate deterministic hashes for the nodes in order to
identify them across
    // submission iff they didn't change.
    Map<Integer, byte[]> hashes =
defaultStreamGraphHasher.traverseStreamGraphAndGenerateHashes(strea
mGraph);

    // 为向后兼容性
    List<Map<Integer, byte[]>> legacyHashes = new ArrayList<>
(legacyStreamGraphHashers.size());
    for (StreamGraphHasher hasher : legacyStreamGraphHashers) {

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

    setChaining(hashes, legacyHashes);

    connectEdges();

    setSlotSharing();

    configureCheckpointing();

    setSchedulerConfiguration();

    // 将已注册的缓存文件添加到作业配置中
    for (Tuple2<String, DistributedCache.DistributedCacheEntry> e :
streamGraph.getCachedFiles()) {
        jobGraph.addUserArtifact(e.f0, e.f1);
    }
```

```
    // 最后设置 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;
}
```

上面的 createJobGraph 方法里面其实也是很复杂，后面单独拿一篇文章来写。

创建 MiniCluster 并启动

```
MiniCluster miniCluster = prepareMiniCluster(jobGraph);

private MiniCluster prepareMiniCluster(JobGraph jobGraph) throws
Exception {
    //准备配置
    Configuration configuration = new Configuration();
    configuration.addAll(jobGraph.getJobConfiguration());
    //将 miniCluster 的资源设置为 无限

configuration.setInteger(TaskManagerOptions.TASK_MANAGER_HEAP_MEMOR
Y, Integer.MAX_VALUE / 4);
    configuration.setDouble(TaskManagerOptions.TASK_MANAGER_CORE,
Integer.MAX_VALUE / 4);
    configuration.setLong(TaskManagerOptions.MANAGED_MEMORY_SIZE,
(Integer.MAX_VALUE / 4) >> 10);

    //引入用户自定义的配置（可能会覆盖上面的配置）
    configuration.addAll(this.configuration);

    if (!configuration.contains(RestOptions.PORT)) {
        configuration.setInteger(RestOptions.PORT, 0);
    }

    int numSlotsPerTaskManager =
configuration.getInteger(TaskManagerOptions.NUM_TASK_SLOTS,
jobGraph.getMaximumParallelism() * jobGraph.getNumberOfVertices());
```

```
    //构建 MiniClusterConfiguration 对象
    MiniClusterConfiguration cfg = new
MiniClusterConfiguration.Builder()
            .setConfiguration(configuration)
            .setNumSlotsPerTaskManager(numSlotsPerTaskManager)
            .build();

    if (LOG.isInfoEnabled()) {
        LOG.info("Running job on local embedded Flink mini
cluster");
    }
    //构建 MiniCluster
    MiniCluster miniCluster = new MiniCluster(cfg);
    //启动 MiniCluster
    miniCluster.start();

    configuration.setInteger(RestOptions.PORT,
miniCluster.getRestAddress().getPort());
    return miniCluster;
}
```

上面代码准备好创建 MiniCluster 的配置，然后创建 MiniCluster 并启动，在 start 方法里面就是我们前一篇文章讲的内容了，这里我就不再重复了。

运行 job

启动 MiniCluster 后，那么接下来就是运行 job 了：

```
miniCluster.executeJob(jobGraph, detached);
```

里面包括 job 的提交及运行，同样也在上一篇文章讲过了

关闭资源

```
//关闭资源
transformations.clear();
if (!detached) {
    miniCluster.close();
} else {
    this.submitMapping.put(jobGraph.getJobID(), miniCluster);
}
```

总结

本文讲解了下流程序的 wordcount 和批程序的 wordcount 执行流程有什么区别