前言

已经分析过 StreamGraph, JobGraph 的生成过程，这两个执行图都是在 client 端生成的。接下来我们将把目光头投向 Flink Job 运行时调度层核心的执行图 - ExecutionGraph。

和 StreamGraph 以及 JobGraph 不同的是，ExecutionGraph 是在 JobManager 中生成的。 Client 向 JobManager 提交 JobGraph 后， JobManager 就会根据 JobGraph 来创建对应的 ExecutionGraph,并以此来调度任务。

核心概念

ExecutionJobVertex

在 ExecutionGraph 中，节点对应的类是 ExecutionJobVertex，与之对应的就是 JobGraph 中的 JobVertex。每一个 ExexutionJobVertex 都是由一个 JobVertex 生成的。

```
private final JobVertex jobVertex;

private final List<OperatorID> operatorIDs;
private final List<OperatorID> userDefinedOperatorIds;

//ExecutionVertex 对应一个并行的子任务
private final ExecutionVertex[] taskVertices;

private final IntermediateResult[] producedDataSets;

private final List<IntermediateResult> inputs;

private final int parallelism;

private final SlotSharingGroup slotSharingGroup;

private final CoLocationGroup coLocationGroup;

private final InputSplit[] inputSplits;

private int maxParallelism;
```

ExecutionVertex

ExexutionJobVertex 的成员变量中包含一个 ExecutionVertex 数组。我们知道，Flink Job 是可以指定任务的并行度的，在实际运行时，会有多个并行的任务同时在执行，对应到这里就是 ExecutionVertex。ExecutionVertex 是并行任务的一个子任务，算子的并行度是多少，那么就会有多少个 ExecutionVertex。

```
private final ExecutionJobVertex jobVertex;

private final Map<IntermediateResultPartitionID,
IntermediateResultPartition> resultPartitions;

private final ExecutionEdge[][] inputEdges;

private final int subTaskIndex;

private final EvictingBoundedList<ArchivedExecution>
priorExecutions;

private volatile CoLocationConstraint locationConstraint;

/** The current or latest execution attempt of this vertex's task.
*/
private volatile Execution currentExecution;    // this field must
never be null
```

Execution

Execution 是对 ExecutionVertex 的一次执行，通过 ExecutionAttemptId 来唯一标识。

IntermediateResult

在 JobGraph 中用 IntermediateDataSet 表示 JobVertex 的对外输出，一个 JobGraph 可能有 n(n >=0) 个输出。在 ExecutionGraph 中，与此对应的就是 IntermediateResult。

```
//对应的IntermediateDataSet的ID
private final IntermediateDataSetID id;
//生产者
private final ExecutionJobVertex producer;

//对应ExecutionJobVertex的并行度
private final int numParallelProducers;

private final IntermediateResultPartition[] partitions = new
IntermediateResultPartition[numParallelProducers];

private final ResultPartitionType resultType;
```

由于 ExecutionJobVertex 有 numParallelProducers 个并行的子任务，自然对应的每一个 IntermediateResult 就有 numParallelProducers 个生产者，每个生产者的在相应的 IntermediateResult 上的输出对应一个 IntermediateResultPartition。IntermediateResultPartition 表示的是 ExecutionVertex 的一个输出分区，即：

```
ExecutionJobVertex --->  IntermediateResult

ExecutionVertex --->  IntermediateResultPartition
```

一个 ExecutionJobVertex 可能包含多个（n） 个 IntermediateResult， 那实际上每一个并行的子任务 ExecutionVertex 可能会会包含（n） 个 IntermediateResultPartition。

IntermediateResultPartition 的生产者是 ExecutionVertex，消费者是一个或若干个 ExecutionEdge。

ExecutionEdge

ExecutionEdge 表示 ExecutionVertex 的输入，通过 ExecutionEdge 将 ExecutionVertex 和 IntermediateResultPartition 连接起来，进而在不同的 ExecutionVertex 之间建立联系。

```
private final IntermediateResultPartition source;

private final ExecutionVertex target;

private final int inputNum;
```

构建 ExecutionGraph 的流程

创建 ExecutionGraph 的入口在 ExecutionGraphBuilder#buildGraph() 中。

1. 创建 ExecutionGraph 对象并设置基本属性

设置 JobInformation, SlotProvider 等信息，下面罗列了一些比较重要的属性：

```
/** Job specific information like the job id, job name, job
configuration, etc. */
private final JobInformation jobInformation;

/** The slot provider to use for allocating slots for tasks as they
are needed. */
private final SlotProvider slotProvider;

/** The classloader for the user code. Needed for calls into user
code classes. */
private final ClassLoader userClassLoader;

/** All job vertices that are part of this graph. */
private final ConcurrentHashMap<JobVertexID, ExecutionJobVertex>
tasks;

/** All vertices, in the order in which they were created. **/
private final List<ExecutionJobVertex> verticesInCreationOrder;

/** All intermediate results that are part of this graph. */
private final ConcurrentHashMap<IntermediateDataSetID,
IntermediateResult> intermediateResults;

/** Current status of the job execution. */
private volatile JobStatus state = JobStatus.CREATED;

    /** Listeners that receive messages when the entire job
switches it status
 * (such as from RUNNING to FINISHED). */
private final List<JobStatusListener> jobStatusListeners;

/** Listeners that receive messages whenever a single task
execution changes its status. */
private final List<ExecutionStatusListener> executionListeners;
```

2. JobVertex 初始化

JobVertex 在 Master 上进行初始化，主要关注OutputFormatVertex 和 InputFormatVertex，其他类型的 vertex 在这里没有什么特殊操作。File output format 在这一步准备好输出目录, Input splits 在这一步创建对应的 splits。

```
for (JobVertex vertex : jobGraph.getVertices()) {
    ....
    try {
        vertex.initializeOnMaster(classLoader);
    }
    catch (Throwable t) {
            throw new JobExecutionException(jobId,
                    "Cannot initialize task '" + vertex.getName() +
"': " + t.getMessage(), t);
    }
}
```

3. 生成 ExecutionGraph 内部的节点和连接

对所有的 Jobvertext 进行拓扑排序，并生成 ExecutionGraph 内部的节点和连接

```
//topologically sort the job vertices and attach the graph to the
existing one
List<JobVertex> sortedTopology =
jobGraph.getVerticesSortedTopologicallyFromSources();
        if (log.isDebugEnabled()) {
            log.debug("Adding {} vertices from job graph {} ({}).",
sortedTopology.size(), jobName, jobId);
        }
        executionGraph.attachJobGraph(sortedTopology);
```

3.1 对 JobVertex 进行拓扑排序

所谓拓扑排序，即保证如果存在 A -> B 的有向边，那么在排序后的列表中 A 节点一定在 B 节点之前。具体的算法这里不再详细分析。

3.2 创建 ExecutionJobVertex

按照拓扑排序的结果依次为每个 JobVertex 创建对应的 ExecutionJobVertex。

```
for (JobVertex jobVertex : topologiallySorted) {


    if (jobVertex.isInputVertex() && !jobVertex.isStoppable()) {
        this.isStoppable = false;
    }

    // create the execution job vertex and attach it to the graph
    //创建 ExecutionJobVertex
    ExecutionJobVertex ejv = new ExecutionJobVertex(
        this,
        jobVertex,
        1,
        rpcTimeout,
        globalModVersion,
        createTimestamp);

    //连接上游节点
    ejv.connectToPredecessors(this.intermediateResults);

    ExecutionJobVertex previousTask =
this.tasks.putIfAbsent(jobVertex.getID(), ejv);
    if (previousTask != null) {
        throw new JobException(String.format("Encountered two job
vertices with ID %s : previous=[%s] / new=[%s]",
            jobVertex.getID(), ejv, previousTask));
    }

    for (IntermediateResult res : ejv.getProducedDataSets()) {
        IntermediateResult previousDataSet =
this.intermediateResults.putIfAbsent(res.getId(), res);
        if (previousDataSet != null) {
            throw new JobException(String.format("Encountered two
intermediate data set with ID %s : previous=[%s] / new=[%s]",
                res.getId(), res, previousDataSet));
        }
    }

    this.verticesInCreationOrder.add(ejv);
    this.numVerticesTotal += ejv.getParallelism();
    newExecJobVertices.add(ejv);
}
```

在创建 ExecutionJobVertex 的时候会创建对应的 ExecutionVertex，IntermediateResult，ExecutionEdge ， IntermediateResultPartition 等对象，这里涉及到的对象相对较多，概括起来大致是这样的：

- 每一个 JobVertex 对应一个 ExecutionJobVertex,
- 每一个 ExecutionJobVertex 有 parallelism 个 ExecutionVertex
- 每一个 JobVertex 可能有 n(n>=0) 个 IntermediateDataSet，在 ExecutionJobVertex 中，一个 IntermediateDataSet 对应一个 IntermediateResult,
- 每一个 IntermediateResult 都有 parallelism 个生产者, 对应 parallelism 个 IntermediateResultPartition
- 每一个 ExecutionJobVertex 都会和前向的 IntermediateResult 连接，实际上是 ExecutionVertex 和 IntermediateResult 建立连接，生成 ExecutionEdge

4. 配置 state checkpointing (忽略)

从 ExecutionGraph 到实际运行的任务 ExecutionGraph 是在创建 JobMaster 时就构建完成的，之后就可以被调度执行了。下面简单概括下调度执行的流程，具体分析见后续的文章。

ExecutionGraph.scheduleForExecution 按照拓扑顺序为所有的 ExecutionJobVertex 分配资源，其中每一个 ExecutionVertex 都需要分配一个 slot, ExecutionVertex 的一次执行对应一个 Execution，在分配资源的时候会依照 SlotSharingGroup 和 CoLocationConstraint 确定，分配的时候会考虑 slot 重用的情况。

在所有的节点资源都获取成功后，会逐一调用 Execution.deploy() 来部署 Execution, 使用 TaskDeploymentDescriptor 来描述 Execution，并提交到分配给该 Execution 的 slot 对应的 TaskManager, 最终被分配给对应的 TaskExecutor 执行。

总结

本文简单概括了 ExecutionGraph 涉及到的概念和其生成过程。

到目前为止，我们了解了 StreamGraph, JobGraph 和 ExecutionGraph 的生成过程，以及他们内部的节点和连接的对应关系。总的来说， streamGraph 是最原始的，更贴近用户逻辑的 DAG 执行图；JobGraph 是对 StreamGraph 的进一步优化，将能够合并的算子合并为一个节点以降低运行时数据传输的开销；ExecutionGraph 则是作业运行是用来调度的执行图，可以看作是并行化版本的 JobGraph,将 DAG 拆分到基本的调度单元。