Flink数据流图的生成----ExecutionGraph的生成

ExecutionGraph是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。

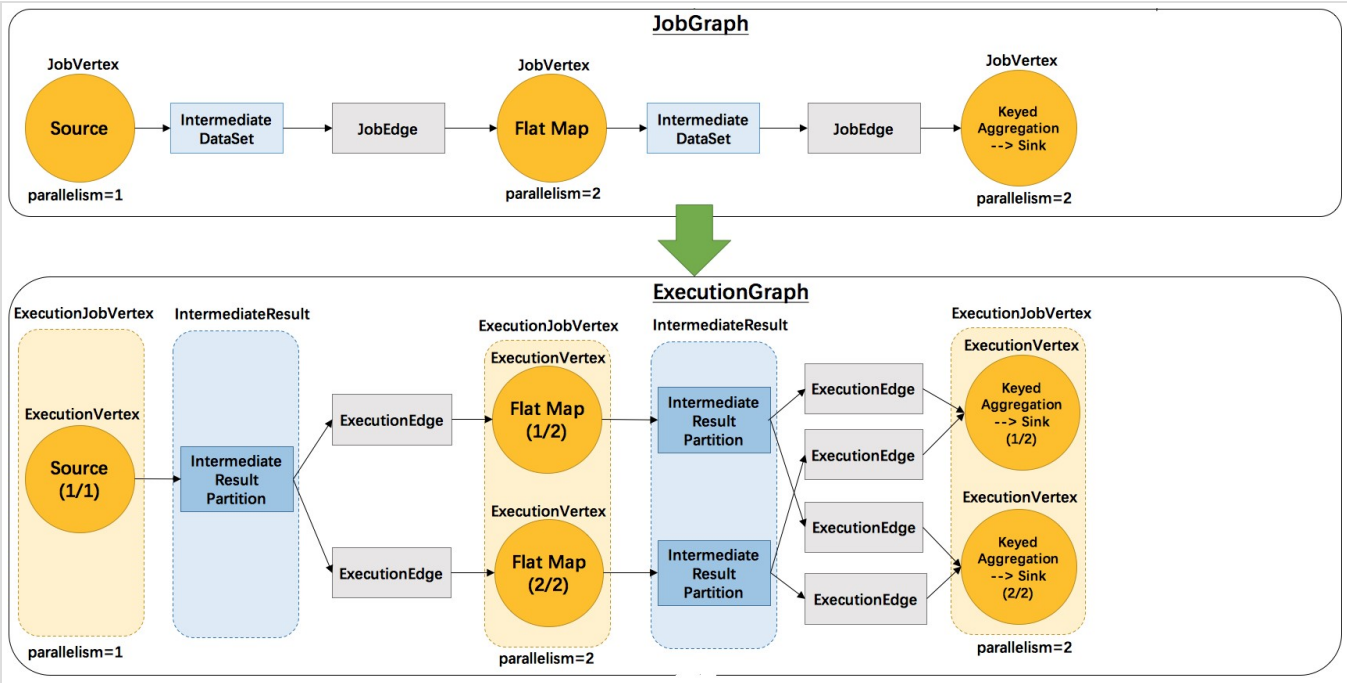
- ExecutionJobVertex：和JobGraph中的JobVertex一一对应。每一个ExecutionJobVertex都有和并发度一样多的 ExecutionVertex
- ExecutionVertex：表示ExecutionJobVertex的其中一个并发子任务，输入是ExecutionEdge，输出是IntermediateResultPartition
- IntermediateResult：和JobGraph中的IntermediateDataSet一一对应。一个IntermediateResult包含多个IntermediateResultPartition，其个数等于该operator的并发度
- IntermediateResultPartition：表示ExecutionVertex的一个输出分区，producer是ExecutionVertex，consumer是若干个ExecutionEdge
- ExecutionEdge：表示ExecutionVertex的输入，source是IntermediateResultPartition，target是ExecutionVertex。source和target都只能是一个
- Execution：是执行一个 ExecutionVertex 的一次尝试。当发生故障或者数据需要重算的情况下 ExecutionVertex 可能会有多个 ExecutionAttemptID。一个 Execution 通过 ExecutionAttemptID 来唯一标识。JM和TM之间关于 task 的部署和 task status 的更新都是通过 ExecutionAttemptID 来确定消息接受者

```
/**
        ExecutionJobVertex
*/
private final ExecutionGraph graph;
private final JobVertex jobVertex;
private final List<OperatorID> operatorIDs;
private final List<OperatorID> userDefinedOperatorIds;
private final ExecutionVertex[] taskVertices;
private final IntermediateResult[] producedDataSets;
private final List<IntermediateResult> inputs;
private final int parallelism;

/**
        ExecutionVertex
*/
private final ExecutionJobVertex jobVertex;
private final Map<IntermediateResultPartitionID, IntermediateResultPartition> result
private final ExecutionEdge[][] inputEdges;
private final int subTaskIndex;
private final Time timeout;
/** The name in the format "myTask (2/7)", cached to avoid frequent string concatena
private final String taskNameWithSubtask;
private volatile Execution currentExecution;    // this field must never be null
```

```
24
25  /**
26          IntermediateResult
27  */
28  private final IntermediateDataSetID id;
29  private final ExecutionJobVertex producer;
30  private final IntermediateResultPartition[] partitions
31          private final int numParallelProducers;
32  private final AtomicInteger numberOfRunningProducers;
33  private int partitionsAssigned;
34  private int numConsumers;
35  private final int connectionIndex;
36  private final ResultPartitionType resultType;
37
38  /**
39          IntermediateResultPartition
40  */
41  private final IntermediateResult totalResult;
42  private final ExecutionVertex producer;
43  private final int partitionNumber;
44  private final IntermediateResultPartitionID partitionId;
45  private List<List<ExecutionEdge>> consumers;
46  addConsumer(ExecutionEdge edge, int consumerNumber)
47
48  /**
49          Execution
50  */
51  private final Executor executor;
52  private final ExecutionVertex vertex;
53  private final ExecutionAttemptID attemptId;
54  private final int attemptNumber;
55  private final Time timeout;
56  private volatile ExecutionState state = CREATED;
57  private volatile SimpleSlot assignedResource;     // once assigned, never changes ur
58  /** The handle to the state that the task gets on restore */
59  private volatile TaskStateHandles taskState;
60  scheduleForExecution()
61  allocateSlotForExecution(SlotProvider slotProvider, boolean queued)
62  deployToSlot(final SimpleSlot slot)
```

对于SocketWindowWordCount.java而言，由JobGraph生成ExecutionGraph的过程如下



源码分析

ExecutionGraph 的生成是在 `org.apache.flink.runtime.executiongraph` 包下的 `ExecutionGraphBuilder` 类中实现的。构造ExcutionGraph的入口函数是buildGraph，该函数首先会判断是否之前存在ExecutionGraph，如果不存在，就新建一个ExecutionGraph。源码如下

```
public static ExecutionGraph buildGraph( . . . . )throws JobExecutionException, JobEx
        final ExecutionGraph executionGraph = (prior != null) ? prior :
                                new ExecutionGraph(
                                        . . . .
)
```

根据JobGraph，得到所有的JobVertex，

```
List<JobVertex> sortedTopology = jobGraph.getVerticesSortedTopologicallyFromSources()
```

getVerticesSortedTopologicallyFromSources()的作用就是生成拓扑排序的所有JobVertex。

```
public List<JobVertex> getVerticesSortedTopologicallyFromSources() throws InvalidPro
                // early out on empty lists
                if (this.taskVertices.isEmpty()) {
                        return Collections.emptyList();
                }
                List<JobVertex> sorted = new ArrayList<JobVertex>(this.taskVertices.
                Set<JobVertex> remaining = new LinkedHashSet<JobVertex>(this.taskVer
                // start by finding the vertices with no input edges
                // and the ones with disconnected inputs (that refer to some standal
                {
                        Iterator<JobVertex> iter = remaining.iterator();
                        while (iter.hasNext()) {
```

```
                                JobVertex vertex = iter.next();

                                if (vertex.hasNoConnectedInputs()) {
                                        sorted.add(vertex);
                                        iter.remove();
                                }
                        }
                }
        return sorted;
```

在该函数内，有一个taskVertices对象，其实一个Map数据，key是JobVertexID，value是JobVertex，通过遍历该对象的value，得到所有的JobVertex。回到buildGraph()函数内，开始正式构建ExecutionGraph，构建的入口就是attachJobGraph()函数，

```
executionGraph.attachJobGraph(sortedTopology);
public void attachJobGraph(List<JobVertex> topologiallySorted) throws JobException {
                final ArrayList<ExecutionJobVertex> newExecJobVertices = new ArrayLi
                final long createTimestamp = System.currentTimeMillis();
                for (JobVertex jobVertex : topologiallySorted) {
                        if (jobVertex.isInputVertex() && !jobVertex.isStoppable()) {
                                this.isStoppable = false;
                        }
                        // create the execution job vertex and attach it to the grap
                        ExecutionJobVertex ejv =
                                        new ExecutionJobVertex(this, jobVertex, 1, r
                        ejv.connectToPredecessors(this.intermediateResults);
```

在attachJobGraph()函数内，会遍历之前得到的具有拓扑顺序的JobVertex，为每一个JobVertex重新建立一个ExecutionJobVertex，会执行ExecutionJobVertex的构造函数，ExecutionGraph所有的结构都是在ExecutionGraph的构造函数中构建的，接下来看其构造函数：

```
. . . . . .//前边是一些初始化的工作
this.producedDataSets = new IntermediateResult[jobVertex.getNumberOfProducedIntermed
        for (int i = 0; i < jobVertex.getProducedDataSets().size(); i++) {
                        final IntermediateDataSet result = jobVertex.getProducedData
                        this.producedDataSets[i] = new IntermediateResult(
                                        result.getId(),
                                        this,
                                        numTaskVertices,
                                        result.getResultType());
        }
```

首先根据JobVertex的每个输出结果集，然后新建一个对应的IntermediateResult，并且执行该类的构造函数

```
public IntermediateResult(. . . . . .) {
    this.id = checkNotNull(id);
    this.producer = checkNotNull(producer);
    checkArgument(numParallelProducers >= 1);
    this.numParallelProducers = numParallelProducers;
```

```
    this.partitions = new IntermediateResultPartition[numParallelProducers];
    this.numberOfRunningProducers = new AtomicInteger(numParallelProducers);
```

在IntermediateResult构造函数中会根据其并行度的大小，(即JobVertex的并行度)，创建对应数量的IntermediateResultPartition。这时候IntermediateResult、IntermediateResultPartition都已经创建完成。回到ExecutionJobVertex的构造函数中，接下来开始根据JobVertex的并行度大小创建对应数量ExecutionVertex，

```
// create all task vertices
                for (int i = 0; i < numTaskVertices; i++) {
                        ExecutionVertex vertex = new ExecutionVertex(
                                        this,
                                        i,
                                        producedDataSets,
                                        timeout,
                                        initialGlobalModVersion,
                                        createTimestamp,
                                        maxPriorAttemptsHistoryLength);
                        this.taskVertices[i] = vertex;
                }
```

接下来会进入ExecutionVertex的构造函数，该构造函数首先进行一些初始化设置，然后建立与IntermediateResultPartition的连接，接着创建ExecutionEdge，再创建Excution，最后通过调用registerExecution()函数来注册该Execution，并且把ExecutionAttemptID作为一个唯一的key。代码如下：

```
for (IntermediateResult result : producedDataSets) {
                        IntermediateResultPartition irp = new IntermediateResultPart
                        result.setPartition(subTaskIndex, irp);
                        resultPartitions.put(irp.getPartitionId(), irp);
                }
this.inputEdges = new ExecutionEdge[jobVertex.getJobVertex().getInputs().size()][];
this.currentExecution = new Execution(
                        getExecutionGraph().getFutureExecutor(),
                        this,
                        0,
                        initialGlobalModVersion,
                        createTimestamp,
                        timeout);
getExecutionGraph().registerExecution(currentExecution);
void registerExecution(Execution exec) {
                Execution previous = currentExecutions.putIfAbsent(exec.getAttemptId
                if (previous != null) {
                        failGlobal(new Exception("Trying to register execution " + e
                }
        }
```

到此ExecutionJobVertex的构造函数就执行完毕，接着回到ExcutionGraph类中的attachJobGraph()函数，然后执行connectToPredecessors()函数来建立ExcutionEdge、IntermediateResultPartition、ExecutionVertex三者之间的连接。具体的连接函数是connectSource()函数(和connect()的函数类似)。

```
ejv.connectToPredecessors(this.intermediateResults);
public void connectToPredecessors(Map<IntermediateDataSetID, IntermediateResult> int
                List<JobEdge> inputs = jobVertex.getInputs();
                for (int num = 0; num < inputs.size(); num++) {
                        JobEdge edge = inputs.get(num);
                        IntermediateResult ires = intermediateDataSets.get(edge.getS
                        this.inputs.add(ires);
                        int consumerIndex = ires.registerConsumer();
                        for (int i = 0; i < parallelism; i++) {
                                ExecutionVertex ev = taskVertices[i];
                                ev.connectSource(num, ires, edge, consumerIndex);
                        }
                }
        }
```

总结

ExecutionGraph就是JobGraph的并行版本，不过ExecutionGraph是在JoBManager端生成的。

ExecutionGraph源码 链接: https://pan.baidu.com/s/18sxpFzPGCOYFZdDNTkDO6w 提取码: rij9