---

toc: true title: 《从1到100深入学习Flink》—— JobManager 处理 SubmitJob 的过程 date: 2019-02-27 tags:

- Flink
- 大数据
- 流式计算

在获取到JobGraph后，客户端会封装一个SubmitJob消息，并将其提交给JobManager，本文就接着分析，JobManager在收到SubmitJob消息后，对其处理逻辑。JobManager是一个Actor，其对接受到的各种消息的处理入口是handleMessage这个方法，其中对SubmitJob的处理入口如下：

```
override def handleMessage: Receive = {
    ...

    case SubmitJob(jobGraph, listeningBehaviour) =>
        val client = sender()
        val jobInfo = new JobInfo(client, listeningBehaviour,
System.currentTimeMillis(), jobGraph.getSessionTimeout)

        submitJob(jobGraph, jobInfo)

    ...
}
```

这里构造了一个JobInfo实例，其是用来保存job的相关信息的，如提交job的客户端、客户端监听模式、任务提交的开始时间、会话超时时间、以及结束时间、耗时等信息。 其中监听模式有三种，三种模型下关心的消息内容依次增加，解释如下：

> a、DETACHED —— 只关心job提交的确认消息 b、EXECUTION_RESULT —— 还关心job的执行结果 c、EXECUTION_RESULT_AND_STATE_CHANGES —— 还关心job的状态变化

然后就进入了真正的处理逻辑subminJob()方法中了，这个方法的代码稍微有点长，这里就分段进行分析，另外submitJob这个方法除了上述的jobGraph和jobInfo两个入参外，还有一个isRecovery的布尔变量，默认值是false，用来标识当前处理的是否是一个job的恢复操作。这个逻辑根据jobGraph是否为null分为两个大的分支，先看下jobGraph为null的情况，处理逻辑就是构造一个job提交异常的消息，然后通知客户端，告诉客户端jobGraph不能为null。

```
jobInfo.notifyClients(
  decorateMessage(JobResultFailure(
    new SerializedThrowable(
      new JobSubmissionException(null, "JobGraph must not be
null.")))))
```

重点还是分析jobGraph不为null的情况下的处理逻辑，这部分的逻辑也可以分为两大部分。

> a、根据jobGraph构建ExecutionGraph b、对构建好的ExecutionGraph进行调度执行

在构建ExecutionGraph这部分，会进行一些初始化的工作，如果在这过程中，发生异常，会将初始化过程做的操作进行回滚操作。

1、构建ExecutionGraph

1.1、总览

在开始ExecutionGraph的构建之前，会先获取构建所需的参数，如下：

```
/** 将job所需jar相关信息注册到library管理器中，如果注册失败，则抛出异常 */
try {
  libraryCacheManager.registerJob(
    jobGraph.getJobID, jobGraph.getUserJarBlobKeys,
jobGraph.getClasspaths)
}
catch {
  case t: Throwable =>
    throw new JobSubmissionException(jobId,
      "Cannot set up the user code libraries: " + t.getMessage, t)
}
```

```
/** 获取用户类加载器，如果获取的类加载器为null，则抛出异常 */
val userCodeLoader =
libraryCacheManager.getClassLoader(jobGraph.getJobID)
if (userCodeLoader == null) {
  throw new JobSubmissionException(jobId,
    "The user code class loader could not be initialized.")
}

/** 判断{@code JobGraph}中的{@code StreamNode}的个数，如果为0，则说明是个
空任务，抛出异常 */
if (jobGraph.getNumberOfVertices == 0) {
  throw new JobSubmissionException(jobId, "The given job is empty")
}

/** 优先采用JobGraph配置的重启策略，如果没有配置，则采用JobManager中配置的重
启策略 */
val restartStrategy =
  Option(jobGraph.getSerializedExecutionConfig()
    .deserializeValue(userCodeLoader)
    .getRestartStrategy())
    .map(RestartStrategyFactory.createRestartStrategy)
    .filter(p => p != null) match {
    case Some(strategy) => strategy
    case None => restartStrategyFactory.createRestartStrategy()
  }

log.info(s"Using restart strategy $restartStrategy for $jobId.")

val jobMetrics = jobManagerMetricGroup.addJob(jobGraph)

/** 获取注册在调度器上的所有TaskManager实例的总的slot数量 */
val numSlots = scheduler.getTotalNumberOfSlots()

/** 针对jobID，看是否已经存在 ExecutionGraph，如果有，则直接获取已有的，并
将registerNewGraph标识为false */
val registerNewGraph = currentJobs.get(jobGraph.getJobID) match {
  case Some((graph, currentJobInfo)) =>
    executionGraph = graph
    currentJobInfo.setLastActive()
    false
  case None =>
    true
}
```

上面这段逻辑主要做一些准备工作，如jar包注册，类加载器，重启策略等，这些准备好之后，就可以开始ExecutionGraph的构建，调用如下：

```
/** 通过{@link JobGraph}构建出{@link ExecutionGraph} */
executionGraph = ExecutionGraphBuilder.buildGraph(
  executionGraph,
  jobGraph,
  flinkConfiguration,
  futureExecutor,
  ioExecutor,
  scheduler,
  userCodeLoader,
  checkpointRecoveryFactory,
  Time.of(timeout.length, timeout.unit),
  restartStrategy,
  jobMetrics,
  numSlots,
  blobServer,
  log.logger)

/** 如果还没有注册过，则进行注册 */
if (registerNewGraph) {
  currentJobs.put(jobGraph.getJobID, (executionGraph, jobInfo))
}

/** 注册job状态变化监听器 */
executionGraph.registerJobStatusListener(
  new StatusListenerMessenger(self, leaderSessionID.orNull))

jobInfo.clients foreach {
  /** 如果客户端关心执行结果和状态变化，则为客户端在executiongraph中注册相应
的监听器 */
  case (client,
ListeningBehaviour.EXECUTION_RESULT_AND_STATE_CHANGES) =>
    val listener  = new StatusListenerMessenger(client,
leaderSessionID.orNull)
    executionGraph.registerExecutionListener(listener)
    executionGraph.registerJobStatusListener(listener)
  case _ => // 如果不关心，则什么都不做
}
```

在ExecutionGraph构建好只有，就会设置相应的监听器，用来监听其后续的调度执行情况。 另外这段代码的执行会被整个的进行了try...catch，看下catch中的逻辑。

```
/** 如果异常，则进行回收操作 */
case t: Throwable =>
  log.error(s"Failed to submit job $jobId ($jobName)", t)
  /** 进行jar包的注册回滚 */
  libraryCacheManager.unregisterJob(jobId)
  blobServer.cleanupJob(jobId)
  /** 移除上面注册的graph */
  currentJobs.remove(jobId)
  /** 如果executionGraph不为null，还需要执行failGlobal操作 */
  if (executionGraph != null) {
    executionGraph.failGlobal(t)
  }
  /** 构建JobExecutionException移除 */
  val rt: Throwable = if (t.isInstanceOf[JobExecutionException]) {
    t
  } else {
    new JobExecutionException(jobId, s"Failed to submit job $jobId
($jobName)", t)
  }
  /** 通知客户端，job失败了 */
  jobInfo.notifyClients(
    decorateMessage(JobResultFailure(new SerializedThrowable(rt))))
  /** 退出submitJob方法 */
  return
```

可见catch中，主要进行一些回滚操作，这样可以确保在出现异常的情况下，可以让已经上传的jar等被删除掉。

1.2、ExecutionGraph

ExecutionGraph是JobGraph的并行模式，是基于JobGraph构建出来的，主要构建逻辑都在ExecutionGraphBuilder这个类中，而且该方法的构造函数是private的，且该类只有两个static方法，buildGraph()和idToVertex()，而ExecutionGraph的构造逻辑都在buildGraph()方法中。 在buildGraph()方法中，先是对executionGraph进行一些基础的设置，如果有需要，则对各个JobVertex进行初始化操作，然后就是将JobVertex转化成ExecutionGraph中的组件，转化成功后，则开始设置checkpoint相关的配置。 这里主要JobVertex转化的逻辑，代码如下：

```
/** 1、构建有序拓扑列表 */
List<JobVertex> sortedTopology =
jobGraph.getVerticesSortedTopologicallyFromSources();
if (log.isDebugEnabled()) {
   log.debug("Adding {} vertices from job graph {} ({}).",
sortedTopology.size(), jobName, jobId);
}
/** 2、转化JobVertex */
executionGraph.attachJobGraph(sortedTopology);
```

主要的转换代码就两行，先是将jobGraph中的所有的JobVertex，从数据源开始的有序拓扑节点列表，然后就是将这个有序集合转化到executionGraph中。

1.2.1 构建有序拓扑列表

有序拓扑列表的构建逻辑在JobGraph类中，如下：

```
public List<JobVertex> getVerticesSortedTopologicallyFromSources()
throws InvalidProgramException {
   /** 节点集合为空时，可以快速退出 */
   if (this.taskVertices.isEmpty()) {
      return Collections.emptyList();
   }

   /** 从source开始的，排好序的JobVertex列表 */
   List<JobVertex> sorted = new ArrayList<JobVertex>
(this.taskVertices.size());
   /** 还没有进入sorted集合，等待排序的JobVertex集合，初始值就是JobGraph中
所有JobVertex的集合 */
   Set<JobVertex> remaining = new LinkedHashSet<JobVertex>
(this.taskVertices.values());

   /** 找出数据源节点，也就是那些没有输入的JobVertex，以及指向独立数据集的
JobVertex */
   {
      Iterator<JobVertex> iter = remaining.iterator();
      while (iter.hasNext()) {
         JobVertex vertex = iter.next();
         /** 如果该节点没有任何输入，则表示该节点是数据源，添加到sorted集合，
同时从remaining集合中移除 */
         if (vertex.hasNoConnectedInputs()) {
            sorted.add(vertex);
            iter.remove();
         }
```

```
        }
    }
    /** sorted集合中开始遍历的起始位置，也就是从第一个元素开始遍历 */
    int startNodePos = 0;

    /** 遍历已经添加的节点，直到找出所有元素 */
    while (!remaining.isEmpty()) {
        /**
         * 处理一个节点后，startNodePos就会加1，
         * 如果startNodePos大于sorted的集合中元素个数，
         * 则说明经过一次处理，并没有找到新的JobVertex添加到sorted集合中，这表
明在graph中存在了循环，这是不允许的
         */
        if (startNodePos >= sorted.size()) {
            throw new InvalidProgramException("The job graph is
cyclic.");
        }
        /** 获取当前要处理的JobVertex */
        JobVertex current = sorted.get(startNodePos++);
        /** 遍历当前JobVertex的下游节点 */
        addNodesThatHaveNoNewPredecessors(current, sorted,
remaining);
    }
    return sorted;
}
```

上述逻辑就是首先从JobGraph的所有JobVertex集合中，找出所有的source节点，然后在从这些source节点开始，依次遍历其下游节点，当一个节点的所有输入都已经被添加到sorted集合中时，它自身就可以添加到sorted集合中了，同时从remining集合中移除。

```
private void addNodesThatHaveNoNewPredecessors(JobVertex start,
List<JobVertex> target, Set<JobVertex> remaining) {
    /** 遍历start节点的所有输出中间数据集合 */
    for (IntermediateDataSet dataSet : start.getProducedDataSets())
{
        /** 对于每个中间数据集合，遍历其所有的输出JobEdge */
        for (JobEdge edge : dataSet.getConsumers()) {
            /** 如果一个节点的所有输入节点都不在"remaining"集合中，则将这个节点
添加到target集合中 */

            /** 如果目标节点已经不在remaining集合中，则continue */
            JobVertex v = edge.getTarget();
            if (!remaining.contains(v)) {
                continue;
```

```
            }
            /** 一个JobVertex是否还有输入节点在remaining集合中的标识 */
            boolean hasNewPredecessors = false;
            /**
             * 如果节点v，其所有输入节点都已经不在remaining集合中，
             * 则说明其输入节点都已经被添加到sorted列表，则hasNewPredecessors
为false，
             * 否则hasNewPredecessors的值为true，表示节点v还有输入节点
在remaining集合中。
             */
            for (JobEdge e : v.getInputs()) {
               /** 跳过上层循环中遍历到的JobEdge，也就是edge变量 */
               if (e == edge) {
                  continue;
               }
               /** 只要有一个输入还在remaining集合中，说明当前它还不能添加
到target集合，直接结束这层内循环 */
               IntermediateDataSet source = e.getSource();
               if (remaining.contains(source.getProducer())) {
                  hasNewPredecessors = true;
                  break;
               }
            }
            /**
             * 如果节点v已经没有输入节点还在remaining集合中，则将节点v添加
到sorted列表中，
             * 同时从remaining集合中删除，
             * 然后开始递归遍历节点v的下游节点。
             */
            if (!hasNewPredecessors) {
               target.add(v);
               remaining.remove(v);
               addNodesThatHaveNoNewPredecessors(v, target,
remaining);
            }
         }
      }
   }
```

对于具体的某个JobVertex的遍历逻辑如上，详见注释。

1.2.2 JobVertex的转化

在获取了排序后的拓扑的JobVertex集合后，就可以开始将其转换成ExecutionGraph中的ExecutionJobVertex。

```
public void attachJobGraph(List<JobVertex> topologiallySorted)

throws JobException {
   LOG.debug("Attaching {} topologically sorted vertices to
existing job graph with {} " +
         "vertices and {} intermediate results.",
         topologiallySorted.size(), tasks.size(),
intermediateResults.size());

   final ArrayList<ExecutionJobVertex> newExecJobVertices = new
ArrayList<>(topologiallySorted.size());
   final long createTimestamp = System.currentTimeMillis();

   /** 依次顺序遍历排好序的JobVertex集合 */
   for (JobVertex jobVertex : topologiallySorted) {
      /** 对于ExecutionGraph来说，只要有一个不能停止的输入源JobVertex，那
ExecutionGraph就是不可停止的 */
      if (jobVertex.isInputVertex() && !jobVertex.isStoppable()) {
         this.isStoppable = false;
      }
      /** 创建jobVertex对应的ExecutionJobVertex，其中的第三个构造参数1，就
是默认的并行度 */
      ExecutionJobVertex ejv = new ExecutionJobVertex(
         this,
         jobVertex,
         1,
         rpcCallTimeout,
         globalModVersion,
         createTimestamp);

      /** 将新建的ExecutionJobVertex实例，与其前置处理器建立连接 */
      ejv.connectToPredecessors(this.intermediateResults);

      /** 将构建好的ejv，记录下来，如果发现对一个的jobVertexID已经存在一
个ExecutionJobVertex，则需要抛异常 */
      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));
      }

      /** 将这个ExecutionGraph中所有临时结果IntermediateResult，都保存
到intermediateResults这个map */
      for (IntermediateResult res : ejv.getProducedDataSets()) {
         IntermediateResult previousDataSet =
this.intermediateResults.putIfAbsent(res.getId(), res);
```

```
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));
         }
      }
      /** 将ejv按创建顺序记录下来 */
      this.verticesInCreationOrder.add(ejv);
      /** 统计所有ejv的并行度 */
      this.numVerticesTotal += ejv.getParallelism();
      newExecJobVertices.add(ejv);
   }
   terminationFuture = new CompletableFuture<>();
   failoverStrategy.notifyNewVertices(newExecJobVertices);
}
```

上述的逻辑是比较清晰的，就是依次遍历排好序的JobVertex集合，并构建相应的ExecutionJobVertex实例，并设置ExecutionGraph中的部分属性。 在ExecutionJobVertex的构造函数中，会根据并行度，构造相应的ExecutionVertex数组，该数组的索引就是子任务的索引号；而在ExecutionVertex的构造函数中，会构造出一个Execution实例。

2、ExecutionGraph的调度执行

在前面的准备工作都完成，ExecutionGraph也构建好之后，接下来就可以对ExecutionGraph进行调度执行。这部分的操作是比较耗时的，所以整个被包在一个futrue中进行异步执行。

> a、如果isRecovery为true，则先进行恢复操作； b、如果isRecovery为false，则进行checkpoint设置，并将jobGraph的相关信息进备份操作。 上述两步完成之后，则会通知客户端，job已经提交成功了。

```
jobInfo.notifyClients(decorateMessage(JobSubmitSuccess(jobGraph.get
JobID)))
```

接下来就是判断当前JobManager是否是leader，如果是，则开始对executionGraph进行调度执行，如果不是leader，则告诉JobManager自身，去进行remove操作，逻辑如下：

```
/** 根据当前JobManager是否是leader，执行不同的操作 */
if (leaderElectionService.hasLeadership) {
  log.info(s"Scheduling job $jobId ($jobName).")
  /** executionGraph进行调度执行 */
  executionGraph.scheduleForExecution()
} else {
  /** 移除这个job */
  self ! decorateMessage(RemoveJob(jobId, removeJobFromStateBackend
= false))
  log.warn(s"Submitted job $jobId, but not leader. The other leader
needs to recover " +
    "this. I am not scheduling the job for execution.")
}
```

接下里就看下executionGraph的调度执行逻辑。

```
public void scheduleForExecution() throws JobException {
   /** 将状态从'CREATED'转换为'RUNNING' */
   if (transitionState(JobStatus.CREATED, JobStatus.RUNNING)) {
      /** 根据调度模式，执行不同的调度策略 */
      switch (scheduleMode) {
         case LAZY_FROM_SOURCES:
            scheduleLazy(slotProvider);
            break;
         case EAGER:
            scheduleEager(slotProvider, scheduleAllocationTimeout);
            break;
         default:
            throw new JobException("Schedule mode is invalid.");
      }
   }
   else {
      throw new IllegalStateException("Job may only be scheduled
from state " + JobStatus.CREATED);
   }
}
```

上述逻辑就是先将ExecutionGraph的状态从'CREATED'转换为'RUNNING'，状态转换成功，会给状态监听者发送状态变化的消息，然后就根据调度的不同模式，进行不同的调度。调度模式分为两种：

> a、LAZY_FROM_SORUCES —— 该模式下，从source节点开始部署执行，成功后，在部署其下游节点，以此类推； b、EAGER —— 该模式下，所有节点同时部署执行；

这里继续分析'EAGER'模式下的调度。

```
private void scheduleEager(SlotProvider slotProvider, final Time
timeout) {
   /** 走到这里了，需要再次确认下当前的状态是否是'RUNNING' */
   checkState(state == JobStatus.RUNNING, "job is not running
currently");
   /** 标识在无法立即获取部署资源时，是否可以将部署任务入队列 */
   final boolean queued = allowQueuedScheduling;

   /** 用来维护所有槽位申请的future */
   final ArrayList<CompletableFuture<Execution>>
allAllocationFutures = new ArrayList<>
(getNumberOfExecutionJobVertices());

   /** 获取每个ExecutionJobGraph申请槽位的future */
   for (ExecutionJobVertex ejv : getVerticesTopologically()) {
      Collection<CompletableFuture<Execution>> allocationFutures =
ejv.allocateResourcesForAll(
         slotProvider,
         queued,
         LocationPreferenceConstraint.ALL);
      allAllocationFutures.addAll(allocationFutures);
   }

   /** 将上面的所有future连接成一个future，只有所有的future都成功，才算成
功，否则就是失败的 */
   final ConjunctFuture<Collection<Execution>>
allAllocationsComplete =
FutureUtils.combineAll(allAllocationFutures);

   /** 构建一个定时任务，用来检查槽位分配是否超时 */
   final ScheduledFuture<?> timeoutCancelHandle =
futureExecutor.schedule(new Runnable() {
      @Override
      public void run() {
         int numTotal =
allAllocationsComplete.getNumFuturesTotal();
         int numComplete =
allAllocationsComplete.getNumFuturesCompleted();
         String message = "Could not allocate all requires slots
within timeout of " +
               timeout + ". Slots required: " + numTotal + ", slots
```

```
                timeout + " : Slots required: " + numTotal + ", slots
allocated: " + numComplete;

            /** 如果超时，则以异常的方式结束分配 */
            allAllocationsComplete.completeExceptionally(new
NoResourceAvailableException(message));
         }
      }, timeout.getSize(), timeout.getUnit());

      /** 根据槽位分配，进行异步调用执行 */
      allAllocationsComplete.handleAsync(
         (Collection<Execution> executions, Throwable throwable) -> {
            try {
               /** 取消上面的超时检查任务 */
               timeoutCancelHandle.cancel(false);
               if (throwable == null) {
                  /** 成功后去所需槽位，现在开始部署 */
                  for (Execution execution : executions) {
                     execution.deploy();
                  }
               }
               else {
                  /** 抛出异常，让异常句柄处理这个 */
                  throw throwable;
               }
            }
            catch (Throwable t) {
               failGlobal(ExceptionUtils.stripCompletionException(t));
            }
            return null;
         },
         futureExecutor);
}
```

整个处理逻辑分为两大步骤：

> a、先进行槽位的分配，获取分配的future； b、成功获取槽位之后，进行部署，这步也是异步的；

另外，在槽位分配上，加上了超时机制，如果达到设定时间，槽位还没有分配好，则进行fail操作。

2.1、槽位的申请分配

槽位的申请分配逻辑如下：

```
public Collection<CompletableFuture<Execution>>
allocateResourcesForAll(
      SlotProvider resourceProvider,
      boolean queued,
      LocationPreferenceConstraint locationPreferenceConstraint) {
   final ExecutionVertex[] vertices = this.taskVertices;
   final CompletableFuture<Execution>[] slots = new
CompletableFuture[vertices.length];
   /** 为ExecutionJobVertex中的每个Execution尝试申请一个slot，并返回
future */
   for (int i = 0; i < vertices.length; i++) {
      final Execution exec =
vertices[i].getCurrentExecutionAttempt();
      final CompletableFuture<Execution> allocationFuture =
exec.allocateAndAssignSlotForExecution(
         resourceProvider,
         queued,
         locationPreferenceConstraint);
      slots[i] = allocationFuture;
   }
   /** 很好，我们请求到了所有的slots */
   return Arrays.asList(slots);
}
```

上述逻辑就是为ExecutionJobVertex中的每个Execution申请一个slot，然后具体的申请逻辑，是放在Execution中的，继续向下看。

```
public CompletableFuture<Execution>
allocateAndAssignSlotForExecution(
      SlotProvider slotProvider,
      boolean queued,
      LocationPreferenceConstraint locationPreferenceConstraint)
throws IllegalExecutionStateException {
   checkNotNull(slotProvider);
   /** 获取在构建JobVertex时已经赋值好的SlotSharingGroup实例和
CoLocationConstraint实例，如果有的话 */
   final SlotSharingGroup sharingGroup =
vertex.getJobVertex().getSlotSharingGroup();
   final CoLocationConstraint locationConstraint =
vertex.getLocationConstraint();

   /** 位置约束不为null，而共享组为null，这种情况是不可能出现的，出现了肯定就
是异常了 */
   if (locationConstraint != null && sharingGroup == null) {
      throw new IllegalStateException(
```

```
            "Trying to schedule with co-location constraint but
without slot sharing allowed.");

   }

   /** 只有状态是 'CREATED' 时，这个方法才能正常工作 */
   if (transitionState(CREATED, SCHEDULED)) {
      /** ScheduleUnit 实例就是在这里构造出来的 */
      ScheduledUnit toSchedule = locationConstraint == null ?
            new ScheduledUnit(this, sharingGroup) :
            new ScheduledUnit(this, sharingGroup,
locationConstraint);

      /** 获取当前任务分配槽位所在节点的"偏好位置集合"，也就是分配时，优先考虑
分配在这些节点上 */
      final CompletableFuture<Collection<TaskManagerLocation>>
preferredLocationsFuture =
calculatePreferredLocations(locationPreferenceConstraint);

      return preferredLocationsFuture
         .thenCompose(
            (Collection<TaskManagerLocation> preferredLocations) ->
               /** 在获取输入节点的位置之后，将其作为偏好位置集合，基于这些偏
好位置，申请分配一个slot */
               slotProvider.allocateSlot(
                  toSchedule,
                  queued,
                  preferredLocations))
         .thenApply(
            (SimpleSlot slot) -> {
               if (tryAssignResource(slot)) {
                  /** 如果slot分配成功，则返回这个future */
                  return this;
               } else {
                  /** 释放slot */
                  slot.releaseSlot();

                  throw new CompletionException(new
FlinkException("Could not assign slot " + slot + " to execution " +
this + " because it has already been assigned "));
               }
            });
   }
   else {
      throw new IllegalExecutionStateException(this, CREATED,
state);
   }
}
```

上述的逻辑还是很清晰的，

> a、将状态从’CREATED’成功转换成’SCHEDULED’； b、根据LocationPreferenceConstraint的设置，为这个Execution指定优先分配槽位所在的TaskManager； c、基于上述步骤获取的偏好位置，进行slot分配； d、在slot分配成功后，将slot设定给当前Execution，如果设定成功，则返回相应的slot，否则是否slot，然后抛出异常。

其中LocationPreferenceConstraint有两种取值：

> a、ALL —— 需要确认其所有的输入都已经分配好slot，然后基于其输入所在的TaskManager，作为其偏好位置集合； b、ANY —— 只考虑那些slot已经分配好的输入所在的TaskManager，作为偏好位置集合；

某个Execution的偏好位置的计算逻辑，是先由其对应的ExecutionVertex基于所有输入，获取偏好位置集合，然后根据LocationPreferenceConstraint的策略不同，删选出一个子集，作为这个Execution的偏好位置集合。 这里就只看下ExecutionVertex基于输入获取偏好集合的逻辑。

```
public Collection<CompletableFuture<TaskManagerLocation>>
getPreferredLocationsBasedOnInputs() {
   // 如果没有输入，则返回空集合，否则，基于输入连接确定偏好位置
   if (inputEdges == null) {
      return Collections.emptySet();
   }
   else {
      Set<CompletableFuture<TaskManagerLocation>> locations = new
HashSet<>(getTotalNumberOfParallelSubtasks());
      Set<CompletableFuture<TaskManagerLocation>> inputLocations =
new HashSet<>(getTotalNumberOfParallelSubtasks());

      // 遍历所有inputs
      for (int i = 0; i < inputEdges.length; i++) {
         inputLocations.clear();
         ExecutionEdge[] sources = inputEdges[i];
         if (sources != null) {
            // 遍历所有输入源
            for (int k = 0; k < sources.length; k++) {
               // 查询输入源的分配slot
               CompletableFuture<TaskManagerLocation>
locationFuture =
sources[k].getSource().getProducer().getCurrentTaskManagerLocationF
uture();
               inputLocations.add(locationFuture);
               // 如果其个输入源有太多的节点分布，则不考虑这个输入源的节点位置
```

```
                // 如果某个输入源有太多的节点分布，则不考虑这个输入源的节点位置
了

                if (inputLocations.size() >
MAX_DISTINCT_LOCATIONS_TO_CONSIDER) {
                    inputLocations.clear();
                    break;
                }
            }
         }
         // 保留具有最少分布位置的输入的位置
         if (locations.isEmpty() || // 当前还没有分配的位置
               (!inputLocations.isEmpty() && inputLocations.size()
< locations.size())) {
            // 当前的输入具有更少的偏好位置
            locations.clear();
            locations.addAll(inputLocations);
         }
      }

      return locations.isEmpty() ? Collections.emptyList() :
locations;
   }
}
```

逻辑拆分如下：

> a、如果没有输入源，则返回空集合，对于数据源节点来说，就是返回空集合；
> b、如果有输入源，则对每个输入源，都找出其所有分区所在的TaskManager的位置，如果某个输入源的分区所在位置超过MAX_DISTINCT_LOCATIONS_TO_CONSIDER(默认值为8)，则不考虑这个输入源，直接跳过，然后将满足条件的输入源中，分区位置分布做少的那个数据源对应的TaskManager的位置集合，作为计算结果返回。

2.2、部署

在槽位分配成功后，就开始各个Execution的部署操作。

```
public void deploy() throws JobException {
   final SimpleSlot slot  = assignedResource;
   checkNotNull(slot, "In order to deploy the execution we first
have to assign a resource via tryAssignResource.");
   /** 检查slot是否alive */
   if (!slot.isAlive()) {
      throw new JobException("Target slot (TaskManager) for
```

```
deployment is no longer alive.");
   }

   /**
    * 确保在正确的状态的情况下进行部署调用
    * 注意：从 CREATED to DEPLOYING 只是用来测试的
    */
   ExecutionState previous = this.state;
   if (previous == SCHEDULED || previous == CREATED) {
      if (!transitionState(previous, DEPLOYING)) {
         /**
          * 竞态条件，有人在部署调用上击中我们了(其实就是冲突了)
          * 这个在真实情况下不该发生，如果发生，则说明有地方发生冲突了
          */
         throw new IllegalStateException("Cannot deploy task:
Concurrent deployment call race.");
      }
   }
   else {
      // vertex 可能已经被取消了，或者已经被调度了
      throw new IllegalStateException("The vertex must be in
CREATED or SCHEDULED state to be deployed. Found state " +
previous);
   }

   try {
      // 很好，走到这里，说明我们被允许部署了
      if (!slot.setExecutedVertex(this)) {
         throw new JobException("Could not assign the
ExecutionVertex to the slot " + slot);
      }

      // 双重校验，是我们 失败/取消 ？ 我们需要释放这个slot?
      if (this.state != DEPLOYING) {
         slot.releaseSlot();
         return;
      }

      if (LOG.isInfoEnabled()) {
         LOG.info(String.format("Deploying %s (attempt #%d) to %s",
vertex.getTaskNameWithSubtaskIndex(),
               attemptNumber,
getAssignedResourceLocation().getHostname()));
      }

      final TaskDeploymentDescriptor deployment =
vertex.createDeploymentDescriptor(
         attemptId,
```

```
            attemptId,
            slot,
            taskState,
            attemptNumber);

      final TaskManagerGateway taskManagerGateway =
slot.getTaskManagerGateway();

      /** 这里就是将task提交到{@code TaskManager}的地方 */
      final CompletableFuture<Acknowledge> submitResultFuture =
taskManagerGateway.submitTask(deployment, timeout);

      /** 根据提交结果进行处理，如果提交失败，则进行fail处理 */
      submitResultFuture.whenCompleteAsync(
         (ack, failure) -> {
            // 只处理失败响应
            if (failure != null) {
               if (failure instanceof TimeoutException) {
                  String taskname =
vertex.getTaskNameWithSubtaskIndex() + " (" + attemptId + ')';

                  markFailed(new Exception(
                     "Cannot deploy task " + taskname + " -
TaskManager (" + getAssignedResourceLocation()
                           + ") not responding after a timeout of " +
timeout, failure));
               } else {
                  markFailed(failure);
               }
            }
         },
         executor);
   }
   catch (Throwable t) {
      markFailed(t);
      ExceptionUtils.rethrow(t);
   }
}
```

上述代码虽然很长，但是逻辑很简明，先是做一系列的校验工作，然后将状态转换为’DEPLOYING’，然后就是TaskDeploymentDescriptor实例，然后提交给相应的TaskManager实例，这里是异步的，如果执行失败，则进行fail处理。 其中提交到TaskManager的消息结构如下：JobManagerMessages.LeaderSessionMessage[TaskMessages.SubmitTask[TaskDeploymentDescriptor]]。