toc: true title: 《从1到100深入学习Flink》—— TaskManager 处理 SubmitJob 的过程 date: 2019-02-28 tags:

- Flink
- 大数据
- 流式计算

在从JobManager中，将SubmitTask提交到TaskManager后，继续分析TaskManager的处理逻辑。 TaskManager是个Actor，混入了LeaderSessionMessageFilter这个trait，所以在从JobManager接收到JobManagerMessages.LeaderSessionMessage[TaskMessages.SubmitTask[TaskDeploymentDescriptor]]这样的一个封装消息后，会先在LeaderSessionMessageFilter这个trait的receive方法中，进行消息的过滤，过滤逻辑如下：

```
abstract override def receive: Receive = {
  case leaderMessage @ LeaderSessionMessage(msgID, msg) =>
    leaderSessionID match {
      case Some(leaderId) =>
        if (leaderId.equals(msgID)) {
          super.receive(msg)
        } else {
          handleDiscardedMessage(leaderId, leaderMessage)
        }
      case None =>
        handleNoLeaderId(leaderMessage)
    }
  case msg: RequiresLeaderSessionID =>
    throw new Exception(s"Received a message $msg without a leader
session ID, even though" +
      s" the message requires a leader session ID.")
  case msg =>
    super.receive(msg)
}
```

逻辑拆分如下：

1、接收到的是一个LeaderSessionMessage消息

1.1、当前TaskManager中有leaderSessionID

1.1.1、TaskManager所属的JobManager的sessionID和消息中的sessionID相同，则调用父类的receive方法 1.1.2、两个sessionID不同，则说明是一个过期消息，忽视该消息

1.2、当前TaskManager没有leaderSessionID，则打印个日志，不做任何处理

2、接收到的是一个RequiresLeaderSessionID消息，说明消息需要leaderSessionID，但其又没有封装在LeaderSessionMessage中，属于异常情况，抛出异常

3、其他消息，调用父类的receive方法

对于从JobManager接收到的上述消息，经过上述处理逻辑后，就变成TaskMessages.SubmitTask[TaskDeploymentDescriptor]，并作为handleMessage方法的入参，SubmitTask是TaskMessage的子类，所以在handleMessage中的处理逻辑如下：

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

  case message: TaskMessage => handleTaskMessage(message)

  ...
}
```

然后会就进入handleTaskMessage方法，如下：

```
private def handleTaskMessage(message: TaskMessage): Unit = {
    ...

    case SubmitTask(tdd) => submitTask(tdd)

    ...
}
```

经过上述两步转化后，就会进入submitTask方法中，且入参就是TaskDeploymentDescriptor。

submitTask()方法的代码很长，但是逻辑不复杂，分块说明如下：

```
/** 获取当前JobManager的actor */
val jobManagerActor = currentJobManager match {
  case Some(jm) => jm
  case None =>
    throw new IllegalStateException("TaskManager is not associated
with a JobManager.")
}

/** 获取library缓存管理器 */
val libCache = libraryCacheManager match {
  case Some(manager) => manager
  case None => throw new IllegalStateException("There is no valid
library cache manager.")
}

/** 获取blobCache */
val blobCache = this.blobCache match {
  case Some(manager) => manager
  case None => throw new IllegalStateException("There is no valid
BLOB cache.")
}

/** 槽位编号校验 */
val slot = tdd.getTargetSlotNumber
if (slot < 0 || slot >= numberOfSlots) {
  throw new IllegalArgumentException(s"Target slot $slot does not
exist on TaskManager.")
}

/** 获取一些链接相关 */
val (checkpointResponder,
  partitionStateChecker,
  resultPartitionConsumableNotifier,
  taskManagerConnection) = connectionUtils match {
  case Some(x) => x
  case None => throw new IllegalStateException("The connection
utils have not been " +
                                                 "initialized.")
}
```

这部分逻辑就是获取一些处理句柄，如果获取不到，则抛出异常，并校验当前任务的槽位编号是否在有效范围，以及一些链接信息。

```scala
/** 构建JobManager的gateway */
val jobManagerGateway = new AkkaActorGateway(jobManagerActor,
leaderSessionID.orNull)

/** 部分数据可能由于量较大，不方便通过rpc传输，会先持久化，然后在这里再加载回来
*/
try {
  tdd.loadBigData(blobCache.getPermanentBlobService);
} catch {
  case e @ (_: IOException | _: ClassNotFoundException) =>
    throw new IOException("Could not deserialize the job
information.", e)
}

/** 获取jobInformation */
val jobInformation = try {

tdd.getSerializedJobInformation.deserializeValue(getClass.getClassL
oader)
} catch {
  case e @ (_: IOException | _: ClassNotFoundException) =>
    throw new IOException("Could not deserialize the job
information.", e)
}

/** 校验jobID信息 */
if (tdd.getJobId != jobInformation.getJobId) {
  throw new IOException(
    "Inconsistent job ID information inside
TaskDeploymentDescriptor (" +
    tdd.getJobId + " vs. " + jobInformation.getJobId + ")")
}

/** 获取taskInformation */
val taskInformation = try {

tdd.getSerializedTaskInformation.deserializeValue(getClass.getClass
Loader)
} catch {
  case e@(_: IOException | _: ClassNotFoundException) =>
    throw new IOException("Could not deserialize the job vertex
information.", e)
}

/** 统计相关 */
val taskMetricGroup = taskManagerMetricGroup.addTaskForJob(
  jobInformation.getJobId,
  jobInformation.getJobName,
```

```
    taskInformation.getJobVertexId,
    tdd.getExecutionAttemptId,
    taskInformation.getTaskName,
    tdd.getSubtaskIndex,
    tdd.getAttemptNumber)

  val inputSplitProvider = new TaskInputSplitProvider(
    jobManagerGateway,
    jobInformation.getJobId,
    taskInformation.getJobVertexId,
    tdd.getExecutionAttemptId,
    new FiniteDuration(
      config.getTimeout().getSize(),
      config.getTimeout().getUnit()))

  /** 构建task */
  val task = new Task(
    jobInformation,
    taskInformation,
    tdd.getExecutionAttemptId,
    tdd.getAllocationId,
    tdd.getSubtaskIndex,
    tdd.getAttemptNumber,
    tdd.getProducedPartitions,
    tdd.getInputGates,
    tdd.getTargetSlotNumber,
    tdd.getTaskStateHandles,
    memoryManager,
    ioManager,
    network,
    bcVarManager,
    taskManagerConnection,
    inputSplitProvider,
    checkpointResponder,
    blobCache,
    libCache,
    fileCache,
    config,
    taskMetricGroup,
    resultPartitionConsumableNotifier,
    partitionStateChecker,
    context.dispatcher)

  log.info(s"Received task
  ${task.getTaskInfo.getTaskNameWithSubtasks()}")
```

上述逻辑还是在获取各种数据，主要的目的根据以上获取的变量，构建一个Task实例。

```
val execId = tdd.getExecutionAttemptId
// 将task添加到map
val prevTask = runningTasks.put(execId, task)
if (prevTask != null) {
  // 对于ID已经存在一个task，则恢复回来，并报告一个错误
  runningTasks.put(execId, prevTask)
  throw new IllegalStateException("TaskManager already contains a task for id " + execId)
}

// 一切都好，我们启动task，让它开始自己的初始化
task.startTaskThread()

sender ! decorateMessage(Acknowledge.get())
```

这里的逻辑就是将新建的task加入到runningTasks这个map中，如果发现相同execID，已经存在执行的task，则先回滚，然后抛出异常。 一切都执行顺利的话，则启动task，并给sender发送一个ack消息。

task的启动，就是执行Task实例中的executingThread这个变量表示的线程。

```
public void startTaskThread() {
   executingThread.start();
}
```

而executingThread这个变量的初始化是在Task的构造函数的最后进行的。

```
executingThread = new Thread(TASK_THREADS_GROUP, this, taskNameWithSubtask);
```

并且将Task实例自身作为其执行对象，而Task实现了Runnable接口，所以最后就是执行Task中的run()方法。 run方法的逻辑，先是进行状态的初始化，就是进入一个while循环，根据当前状态，执行不同的操作，有可能正常退出循环，进行向下执行，有可能直接reture，有可能抛出异常，逻辑如下：

```java
while (true) {
    ExecutionState current = this.executionState;
    if (current == ExecutionState.CREATED) {
        /** 如果是CREATED状态，则先将状态转换为DEPLOYING，然后退出循环 */
        if (transitionState(ExecutionState.CREATED,
ExecutionState.DEPLOYING)) {
            /** 如果成功，则说明我们可以开始启动我们的work了 */
            break;
        }
    }
    else if (current == ExecutionState.FAILED) {
        /** 如果当前状态是FAILED，则立即执行失败操作，告诉TaskManager，我们已
经到达最终状态了，然后直接返回 */
        notifyFinalState();
        if (metrics != null) {
            metrics.close();
        }
        return;
    }
    else if (current == ExecutionState.CANCELING) {
        if (transitionState(ExecutionState.CANCELING,
ExecutionState.CANCELED)) {
            /** 如果是CANCELING状态，则告诉TaskManager，我们到达最终状态了，
然后直接返回 */
            notifyFinalState();
            if (metrics != null) {
                metrics.close();
            }
            return;
        }
    }
    else {
        /** 如果是其他状态，则抛出异常 */
        if (metrics != null) {
            metrics.close();
        }
        throw new IllegalStateException("Invalid state for beginning
of operation of task " + this + '.');
    }
}
```

当从这个while循环正常退出后，继续向下执行，就是一个try-catch-finally的结构。

这里主要分析一下try块中的逻辑。

1、任务引导

```
// activate safety net for task thread
LOG.info("Creating FileSystem stream leak safety net for task {}",
this);
FileSystemSafetyNet.initializeSafetyNetForThread();

blobService.getPermanentBlobService().registerJob(jobId);

/**
 * 首先, 获取一个 user-code 类加载器
 * 这可能涉及下载作业的JAR文件和/或类。
 */
LOG.info("Loading JAR files for task {}.", this);

userCodeClassLoader = createUserCodeClassloader();
final ExecutionConfig executionConfig =
serializedExecutionConfig.deserializeValue(userCodeClassLoader);

if (executionConfig.getTaskCancellationInterval() >= 0) {
   /** 尝试取消task时, 两次尝试之间的时间间隔, 单位毫秒 */
   taskCancellationInterval =
executionConfig.getTaskCancellationInterval();
}

if (executionConfig.getTaskCancellationTimeout() >= 0) {
   /** 取消任务的超时时间, 可以在flink的配置中覆盖 */
   taskCancellationTimeout =
executionConfig.getTaskCancellationTimeout();
}

/**
 * 实例化AbstractInvokable的具体子类
 * {@see StreamGraph#addOperator}
 * {@see StoppableSourceStreamTask}
 * {@see SourceStreamTask}
 * {@see OneInputStreamTask}
 */
invokable = loadAndInstantiateInvokable(userCodeClassLoader,
nameOfInvokableClass);

/** 如果当前状态'CANCELING'、'CANCELED'、'FAILED', 则抛出异常 */
if (isCanceledOrFailed()) {
   throw new CancelTaskException();
}
```

这部分就是加载jar包，超时时间等获取，然后实例化AbstractInvokable的一个具体子类，目前主要是StoppableSourceStreamTask、SourceStreamTask、OneInputStreamTask 这三个子类。 并且会对状态进行检查，如果处于’CANCELING’、’CANCELED’、’FAILED’其中的一个状态，则抛出CancelTaskException异常。

2、相关注册

```
LOG.info("Registering task at network: {}.", this);

network.registerTask(this);

// add metrics for buffers
this.metrics.getIOMetricGroup().initializeBufferMetrics(this);

// register detailed network metrics, if configured
if
(taskManagerConfig.getConfiguration().getBoolean(TaskManagerOptions
.NETWORK_DETAILED_METRICS)) {
   // similar to MetricUtils.instantiateNetworkMetrics() but inside
this IOMetricGroup
   MetricGroup networkGroup =
this.metrics.getIOMetricGroup().addGroup("Network");
   MetricGroup outputGroup = networkGroup.addGroup("Output");
   MetricGroup inputGroup = networkGroup.addGroup("Input");

   // output metrics
   for (int i = 0; i < producedPartitions.length; i++) {
      ResultPartitionMetrics.registerQueueLengthMetrics(
         outputGroup.addGroup(i), producedPartitions[i]);
   }

   for (int i = 0; i < inputGates.length; i++) {
      InputGateMetrics.registerQueueLengthMetrics(
         inputGroup.addGroup(i), inputGates[i]);
   }
}

/** 接下来，启动为分布式缓存进行文件的后台拷贝 */
try {
   for (Map.Entry<String, DistributedCache.DistributedCacheEntry>
entry :
         DistributedCache.readFileInfoFromConfig(jobConfiguration))
   {
      LOG.info("Obtaining local cache file for '{}'.",
entry.getKey());
```

```
entry.getKey());
            Future<Path> cp = fileCache.createTmpFile(entry.getKey(),
entry.getValue(), jobId);
            distributedCacheEntries.put(entry.getKey(), cp);
        }
    }
catch (Exception e) {
    throw new Exception(
        String.format("Exception while adding files to distributed
cache of task %s (%s).", taskNameWithSubtask, executionId),
        e);
}

/** 再次校验状态 */
if (isCanceledOrFailed()) {
    throw new CancelTaskException();
}
```

这里最后，也会进行状态校验，以便可以快速执行取消操作。

3、用户代码初始化

```java
TaskKvStateRegistry kvStateRegistry = network
      .createKvStateTaskRegistry(jobId, getJobVertexId());

Environment env = new RuntimeEnvironment(
   jobId, vertexId, executionId, executionConfig, taskInfo,
   jobConfiguration, taskConfiguration, userCodeClassLoader,
   memoryManager, ioManager, broadcastVariableManager,
   accumulatorRegistry, kvStateRegistry, inputSplitProvider,
   distributedCacheEntries, writers, inputGates,
   checkpointResponder, taskManagerConfig, metrics, this);

/** 让task代码创建它的readers和writers */
invokable.setEnvironment(env);

// the very last thing before the actual execution starts running
is to inject
// the state into the task. the state is non-empty if this is an
execution
// of a task that failed but had backuped state from a checkpoint

if (null != taskStateHandles) {
   if (invokable instanceof StatefulTask) {
      StatefulTask op = (StatefulTask) invokable;
      op.setInitialState(taskStateHandles);
   } else {
      throw new IllegalStateException("Found operator state for a
non-stateful task invokable");
   }
   // be memory and GC friendly - since the code stays in invoke()
for a potentially long time,
   // we clear the reference to the state handle
   //noinspection UnusedAssignment
   taskStateHandles = null;
}
```

4、真正执行

```java
/** 在我们将状态切换到'RUNNING'状态时，我们可以方法cancel方法 */
this.invokable = invokable;

/** 将状态从'DEPLOYING'切换到'RUNNING'，如果失败，已经是在同一时间，发生了canceled/failed 操作。 */
if (!transitionState(ExecutionState.DEPLOYING,
ExecutionState.RUNNING)) {
   throw new CancelTaskException();
}

/** 告诉每个人，我们切换到'RUNNING'状态了 */
notifyObservers(ExecutionState.RUNNING, null);
taskManagerActions.updateTaskExecutionState(new
TaskExecutionState(jobId, executionId, ExecutionState.RUNNING));

/** 设置线程上下文类加载器 */
executingThread.setContextClassLoader(userCodeClassLoader);

/** run，这里就是真正开始执行处理逻辑的地方 */
invokable.invoke();

/** 确保，如果task由于被取消而退出了invoke()方法，我们可以进入catch逻辑块 */
if (isCanceledOrFailed()) {
   throw new CancelTaskException();
}
```

其中的 invokable.invoke() 这句代码就是真正逻辑开始执行的地方，且一般会阻塞在这里，直至任务执行完成，或者被取消，发生异常等。

5、结尾

```java
/** 完成生产数据分区。如果这里失败，我们也任务执行失败 */
for (ResultPartition partition : producedPartitions) {
   if (partition != null) {
      partition.finish();
   }
}

/**
 * 尝试将状态从'RUNNING'修改为'FINISHED'
 * 如果失败，那么task是同一时间被执行了 canceled/failed 操作
 */
if (transitionState(ExecutionState.RUNNING,
ExecutionState.FINISHED)) {
   notifyObservers(ExecutionState.FINISHED, null);
}
else {
   throw new CancelTaskException();
}
```

这里就是做收尾操作，以及把状态从'RUNNING'转换为'FINISHED'，并通知相关观察者。