Flink整体执行流程

以Flink源码中自带的WordCount为例，执行的入口从用户程序的execute()函数入手，execute()的源码如下：

```
public JobExecutionResult execute(String jobName) throws Exception {
    StreamGraph streamGraph = getStreamGraph();
    streamGraph.setJobName(jobName);
    JobGraph jobGraph = streamGraph.getJobGraph();
    . . . . . .
        LocalFlinkMiniCluster exec = new LocalFlinkMiniCluster(configuration, true);
    try {
        exec.start();
        return exec.submitJobAndWait(jobGraph, getConfig().isSysoutLoggingEnabled())
    }
    finally {
        transformations.clear();
        exec.stop();
    }
}
```

函数内部主要有getStreamGraph()、getJobGraph()、exec.start()、exec.submitJobAndWait()等。getStreamGraph()的作用是生成StreamGraph图，getJobGraph()的作用是生成JobGraph的图，exec.start()的作用是建立Client、JobManager、TaskManager三者之间通信初始化，exec.submitJobAndWait()的作用提交job并且等待job执行后的结果，该函数提供了任务执行调度执行的入口，进入Client类中，首先执行createUserCodeClassLoader()函数，创建用户代码的加载器，然后执行jobClient.SubmitJobAndWait()，进入JobClient类，在函数内部会执行submit函数，从该函数开始进入AKKA通信阶段，首先会进入JobClientActor，会创建一个jobclientActor来对JobManager和client进行通信，当通信对象创建之后，会执行akka机制的ask函数，该函数的作用是发出一个消息，然后要求收到方给予回复。当消息发出之后，OnReceive()函数会收到actor发出的消息请求，然后调用handleMessage()方法来处理消息请求，该函数内部有connectToJobManager()方法，此方法内部的tryToSubmitJob()函数是正式提交任务的操作，主要做的工作就是uploadUserJars()上传用户程序的jar文件，接着会jobManager.tell()向JobManager发出一个submit消息请求。

当JobManager收到Client发送的消息之后，会执行JobManager内部的submitJob方法，

```
case SubmitJob(jobGraph, listeningBehaviour) =>
    val client = sender()

    val jobInfo = new JobInfo(client, listeningBehaviour, System.currentTimeMillis(),
                              jobGraph.getSessionTimeout)
    log.info("liuzf---开始执行JobManager的submitJob()")
    submitJob(jobGraph, jobInfo)
```

首先会把由client收到的job信息封装在jobinfo中，然后把jobinfo以及job的任务图jobGraph一起发送给submit()去执行，在JobManager的submit函数中处理的函数逻辑比较复杂，比较重要的函数执行过程如下：

```
private def submitJob(jobGraph: JobGraph, jobInfo: JobInfo, isRecovery: Boolean = fa
    try {
        libraryCacheManager.registerJob(jobGraph.getJobID, jobGraph.getUserJarBlobKe
                                        jobGraph.getClasspaths)
    }
    val userCodeLoader = libraryCacheManager.getClassLoader(jobGraph.getJobID)

}
executionGraph = ExecutionGraphBuilder.buildGraph()
    try {
        submittedJobGraphs.putJobGraph(new SubmittedJobGraph(jobGraph, jobInfo))
            jobInfo.notifyClients(
            decorateMessage(JobSubmitSuccess(jobGraph.getJobID)))
            log.info(s"开始调度 job $jobId ($jobName).")
            executionGraph.scheduleForExecution()
```

首先执行libraryCacheManager.registerJob()，向CacheManager进行注册，请求缓存，然后执行getClassLoader()来加载用户的代码加载器，接下来会调用ExecutionGraph中的buildGraph()构造ExecutionGraph的并行化版本的执行图，当逻辑执行图构造完毕之后，这时候可以通知Client任务已经成功提交，并且提交过程结束。接下来会调用sheduleForExecution()来会整体的资源进行调度分配，主要是每个TaskManager中的slot的分配，并且当slot分配完成之后，所有的task的任务状态发生改变，由CREATEDàSCHEDULED。接下分配完之后，接下来执行depolyToSlot()函数，就要进入部署状态，同样会执行transitionState()函数，将SCHEDULED状态变为DEPOLYING状态，接着的重要函数是shumitTask()函数，该函数会通过AKKA机制，向TaskManager发出一个submitTask的消息请求，TaskManager收到消息请求后，会执行submitTask()方法，该函数的重要执行过程如下：

```
public submitTask(){
    val task = new Task(. . . .)
        log.info(s"Received task ${task.getTaskInfo.getTaskNameWithSubtasks()}")
        val execId = tdd.getExecutionAttemptId
        val prevTask = runningTasks.put(execId, task)
        if (prevTask != null) {
            runningTasks.put(execId, prevTask)
                throw new IllegalStateException("TaskM}anager already contains a tas
        }
    task.startTaskThread()
        sender ! decorateMessage(Acknowledge.get())
}
```

首先执行Task的构造函数，生成具体物理执行的相关组件，比如ResultPartition等，最后创建执行Task的线程，然后调用startTaskThread()来启动具体的执行线程，Task线程内部的run()方法承载了被执行的核心逻辑，该方法具体的内容为：

```
public void run() {
    while (true) {
        ExecutionState current = this.executionState;
        if (current == ExecutionState.CREATED) {
            if (transitionState(ExecutionState.CREATED, ExecutionState.DEPLOYING)) {
                break;
```

```
            }
        }
        invokable = loadAndInstantiateInvokable(userCodeClassLoader, nameOfInvokable
        network.registerTask(this);
        Environment env = new RuntimeEnvironment(. . . . );
        invokable.setEnvironment(env);
        // ----------------------------------------------------------------
        //  actual task core work
        if (!transitionState(ExecutionState.DEPLOYING, ExecutionState.RUNNING)) {
        }
        // notify everyone that we switched to running
        notifyObservers(ExecutionState.RUNNING, null);
        executingThread.setContextClassLoader(userCodeClassLoader);
        // run the invokable
        invokable.invoke();

        if (transitionState(ExecutionState.RUNNING, ExecutionState.FINISHED)) {
            notifyObservers(ExecutionState.FINISHED, null);
        }
        Finally{
            // free the network resources
            network.unregisterTask(this);
            // free memory resources
            if (invokable != null) {
                memoryManager.releaseAll(invokable);
            }
            libraryCache.unregisterTask(jobId, executionId);
            removeCachedFiles(distributedCacheEntries, fileCache);
```

首先执行transitionState()函数将TaskManager的状态由CREATED转变为DEPOLYING状态，然后调用loadAndTrantiateInvokable()对用户代码打包成jar包，并且生成用户代码加载器，然后执行network.registerTask()，执行该函数之前，会执行NetworkEnvironment的构造函数，该类是TaskManager通信的主对象，主要用于跟踪中间结果并负责所有的数据交换，在该类中会创建协助通信的关键部件，比如网络缓冲池，连接管理器，结果分区管理器，结果分区可消费通知器等。当网络对象准备完成后，创建一个运行环境，然后执行invoke.setEnvironment(env)，将各种配置打包到运行环境中。

当运行环境准备之后，接下来到了具体分析任务执行的时候，首先会调用transitionState()函数将任务状态由DEPOLYING改为RUNNING状态，然后会调用notifyObservers()通知所有的task观察者也改变状态，然后执行setContextClassLoader()将执行的类加载器设置为用户执行的加载器，然后执行invokable.invoke()，该函数是分界点，执行前用户逻辑没有被触发，执行之后说明用户逻辑已完成。当执行完成之后，调用transitionState()函数执行的RUNNING状态改成FINISHED状态。同样调用notifyObservers()来通知其他观察者改变状态，最后，释放资源。

总体的函数执行图如下：因图片太大——»»>

链接: https://pan.baidu.com/s/149pxDTNtDX5kAG3ocJsNGg 提取码: bps8