96_源码剖析：Spring Batch 底层工作原理

1、开篇

上节课首先介绍了 Spring Batch 的实现框架，包括：JobRepositroy、JobLuancher、Job 和 Step。然后又说了如何让 Spring Batch 落地到当前项目。

在利用 Spring Batch 进行编码之前，先通过对 Spring Batch 源码分析了解其底层的工作原理，这也是本节课的教学内容。今天课程的内容包括以下几个部分：

- JobSynchronizationManager
- FlowJob
- SimpleFlow
- StepState
- SimpleStephandler
- AbstractStep

2、JobSynchronizationManager

启动 batch 任务时，会调用 job.execute(jobExecution)： job 为 FlowJob 类型，jobExecution 调用 jobRepository 的 createJobExecution 方法生成。

FlowJob 继承 org.springframework.batch.core.job.AbstractJob，调用 AbstractJob#execute 方法执行 job，这个方法负责执行 job、处理所有的 listeners 和 repository 调用、将实际的处理委托给子类的 doExecute 方法。

如下代码所示，通过 JobSynchronizationManager 为当前线程注册 step context。

在 JobSynchronizationManager#register 方法中，实际调用 JobSynchronizationManager 的 manager 属性的 register 方法来完成。

JobSynchronizationManager.register(execution);

SynchronizationManagerSupport 用来存储当前线程的 execution，execution 与 context（new JobContext(execution)）的 map。

new SynchronizationManagerSupport<JobExecution, JobContext>()

然后调用调用 doExecute 方法执行 job。

3、FlowJob 的 doExecute 方法

如下代码所示，创建 JobFlowExecutor，JobFlowExecutor 用在需要执行与 JobExecution 有关的 flow 的组件中

JobFlowExecutor executor = new JobFlowExecutor(getJobRepository(), new SimpleStepHandler(getJobRepository()), execution);

SimpleStepHandler 负责管理 repository，重启业务，在 new SimpleStepHandler 时，会新建 ExecutionContext。这个类封装了一个 ConcurrentHashMap，能够提供类型安全的 read 操作。

实际负责 job 的是 flow.start(executor)，如下代码，flow 是根据配置文件中定义的 job 生成的 SimpleFlow。

executor.updateJobExecutionStatus(flow.start(executor).getStatus());

SimpleFlow 的属性 startState 为 StepState 类型，值为 job 的第一个 step 定义，name 为 job 的 id.第一个 step 的 id。

start 方法中，会定义一个 state 并且将 startState 赋值给他，然后取得 stateName（job 的 id.第一个 step 的 id）,最后调用 resume 方法

4、SimpleFlow 的 resume 方法

resume 方法的核心是调用 state 的 handle 方法，如下代码所示，在 resume 方法体内会通过 state 中的 handle 获取执行器状态信息，通过 executor 中的 getStepExecution 获取每一步的执行信息 stepExecution。

```
while (isFlowContinued(state, status, stepExecution)) {
  stateName = state.getName();
  try {
     status = state.handle(executor);
     stepExecution = executor.getStepExecution();
  }
  state = nextState(stateName, status, stepExecution);
}
FlowExecution result = new FlowExecution(stateName, status);
```

5、StepState 的 handle 方法

如下代码所示 在启动新的 step 时，要更新上一次 execution，确保他执行失败后在这次启动时能够被放弃。

```
executor.abandonStepExecution();
return new FlowExecutionStatus(executor.executeStep(step));
```

在启动新的 step 时，要更新上一次 execution，确保他执行失败后在这次启动时能够被放弃。JobFlowExecutor 继承 FlowExecutor 接口，这个接口为 FlowJob 提供 step by step 执行的 context 和执行策略。JobFlowExecutor#executeStep 方法的核心如下代码所示。

```
StepExecution stepExecution = stepHandler.handleStep(step, execution);
```

6、SimpleStepHandler 的 handleStep 方法

接着上面的 stepHandler 的部分，实际上是调用了 SimpleStepHandler 中的 handleStep 方法，下面是代码节选，大家大致了解起运作过程。通过 execution 的 createStepExecution 创建 currentStepExecution，然后设置执行的上下文，方法是 setExecutionContext。然后通过 step 的 execute 方法执行 currentStepExecution。这里的 execute 方法是来自于 AbstractStep 类中的 doExecute 方法。

```
currentStepExecution = execution.createStepExecution(step.getName());
currentStepExecution.setExecutionContext(new ExecutionContext(executionContext));
step.execute(currentStepExecution);
doExecute(stepExecution);
```

7、AbstractStep 的 doExecute 方法

这个方法中会创建一个 Semaphore，这个信号量是为了 step 能够在不使用锁的情况下并发执行。同时还执行 doInChunkContext 的操作，如下代码，其中会建立 TransactionTemplate 完成事务的执行。

```
stepOperations.iterate(new StepContextRepeatCallback(stepExecution) {
  @Override
  public RepeatStatus doInChunkContext(RepeatContext repeatContext, ChunkContext
chunkContext)
        throws Exception {
    StepExecution stepExecution = chunkContext.getStepContext().getStepExecution();
    // Before starting a new transaction, check for interruption.
    interruptionPolicy.checkInterrupted(stepExecution);
    RepeatStatus result;
    try {
        result = new TransactionTemplate(transactionManager, transactionAttribute)
        .execute(new ChunkTransactionCallback(chunkContext, semaphore));
    }
    catch (UncheckedTransactionException e) {
        // Allow checked exceptions to be thrown inside callback
        throw (Exception) e.getCause();
    }
    chunkListener.afterChunk(chunkContext);
```

```
        // Check for interruption after transaction as well, so that the interrupted exception is。
//correctly propagated up to
        // caller
        interruptionPolicy.checkInterrupted(stepExecution);
        return result;
    }
});
```

8、总结

本节课对 Spring Batch 的部分源码进行了讲解，其中包括：JobSynchronizationManager、FlowJob、SimpleFlow、StepState、SimpleStephandler、AbstractStep 等等。下节课会基于 Spring Batch，编写代码实现教师教学天数的批量更新。下期见，拜拜。