Spring Bacth 框架介绍

在大型企业中，由于业务复杂、数据量大、数据格式不同、数据交互格式繁杂，并非所有的操作都能通过交互界面进行处理。而有一些操作需要定期读取大批量的数据，然后进行一系列的后续处理。这样的过程就是“批处理”。

批处理应用通常有以下特点：

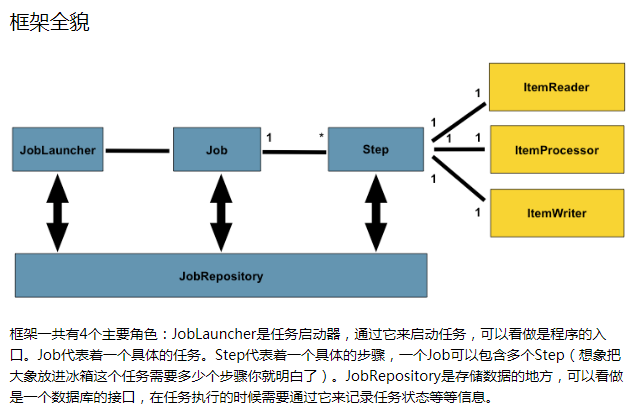
* 数据量大，从数万到数百万甚至上亿不等；
* 整个过程全部自动化，并预留一定接口进行自定义配置；
* 这样的应用通常是周期性运行，比如按日、周、月运行；
* 对数据处理的准确性要求高，并且需要容错机制、回滚机制、完善的日志监控等。

### **什么是Spring batch**

Spring batch是一个轻量级的全面的批处理框架，它专为大型企业而设计，帮助开发健壮的批处理应用。Spring batch为处理大批量数据提供了很多必要的可重用功能，比如日志追踪、事务管理、job执行统计、重启job和资源管理等。同时它也提供了优化和分片技术用于实现高性能的批处理任务。

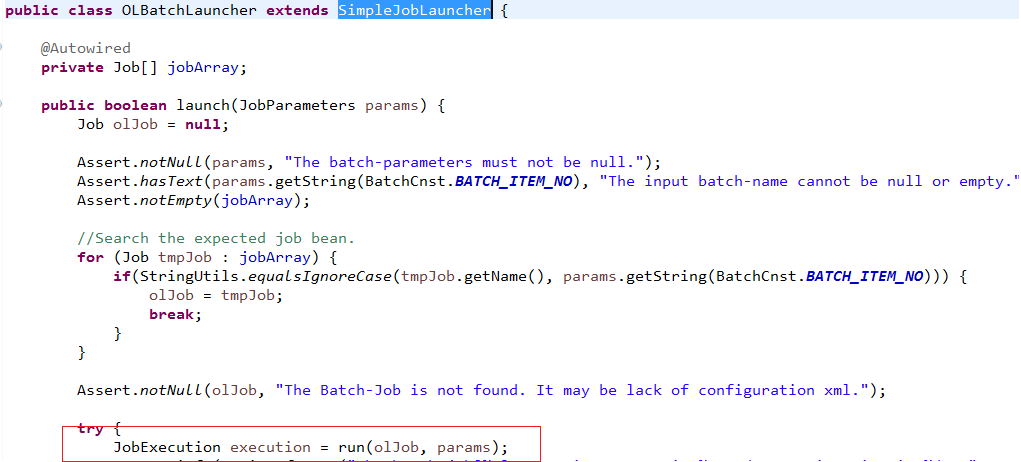
它的核心功能包括：

* 事务管理
* 基于块的处理过程
* 声明式的输入/输出操作
* 启动、终止、重启任务
* 重试/跳过任务
* 基于Web的管理员接口

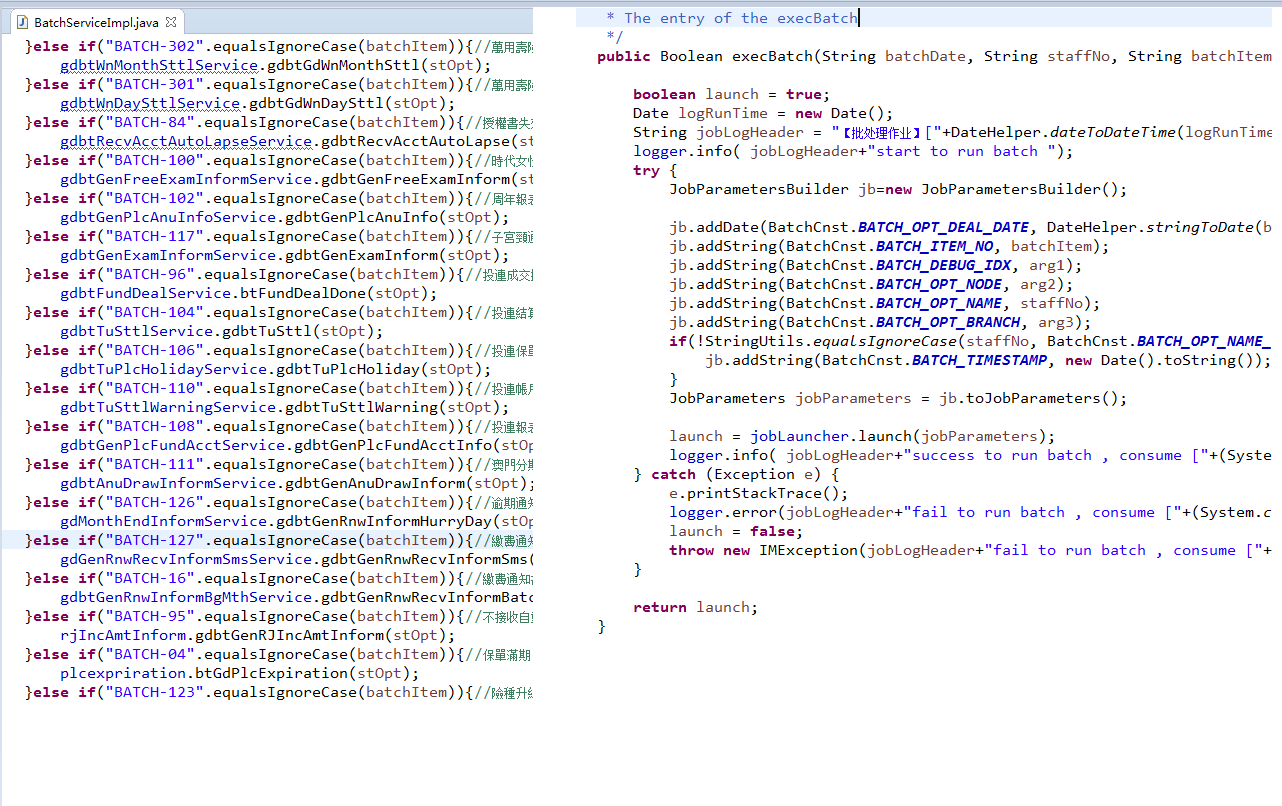


### JobLauncher

JobLauncher是任务启动器，该接口只有一个run方法：如OlBatchLauncher.java（下图所示）除了传入Job对象之外，还需要传入JobParameters对象，通过JobLauncher可以在Java程序中调用批处理任务，也可以通过命令行或者其他框架（如定时调度框架Quartz、Web后台框架Spring MVC）中调用批处理任务。Spring Batch框架提供了一个JobLauncher的实现类SimpleJobLauncher。

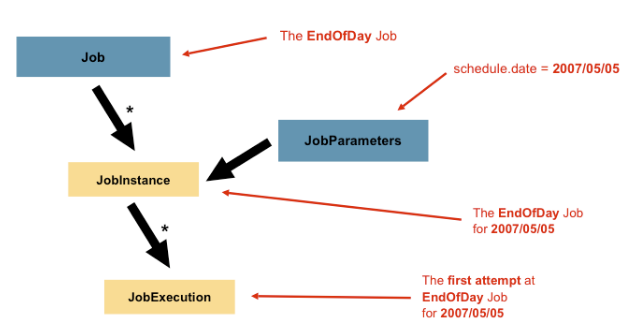


使用Spring Bacth 框架后，批处理调用入口比之前要简洁很多（如下图）：Spring Batch框架不需要像传统的Spring 框架一个一个将所有批处理Service 注入，然后手动调用服务，只需要通过任务加载器JobLauncher 直接加载配置文件Spring-Bacth中对应Job启动即可；（BatchServiceImpl.java前后对比）



### Job

Job代表着一个任务，一个Job与一个或者多个JobInstance相关联，而一个JobInstance又与一个或者多个JobExecution相关联：



考虑到任务可能不是只执行一次就再也不执行了，更多的情况可能是定时任务，如每天执行一次，每个星期执行一次等等，那么为了区分每次执行的任务，框架使用了JobInstance。如上图所示，Job是一个EndOfDay（每天最后时刻执行的任务），那么其中一个JobInstance就代表着2007年5月5日那天执行的任务实例。框架通过在执行JobLauncher.run(Job, JobParameters)方法时传入的JobParameters来区分是哪一天的任务。

由于2007年5月5日那天执行的任务可能不会一次就执行完成，比如中途被停止，或者出现异常导致中断，需要多执行几次才能完成，所以框架使用了JobExecution来表示每次执行的任务。

### Step

一个Job任务可以分为几个Step步骤，与JobExection相同，每次执行Step的时候使用StepExecution来表示执行的步骤。每一个Step还包含着一个ItemReader、ItemProcessor、ItemWriter，下面分别介绍这三者。

****ItemReader****

ItemReader代表着读操作，其接口如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **public** **interface** ItemReader<T> {        T read() **throws** Exception, UnexpectedInputException, ParseException, NonTransientResourceException;    } |

框架已经提供了多种ItemReader接口的实现类，包括对文本文件、XML文件、数据库、JMS消息等读的处理，当然我们也可以自己实现该接口。

****ItemProcessor****

ItemProcess代表着处理操作，其接口如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **public** **interface** ItemProcessor<I, O> {        O process(I item) **throws** Exception;    } |

process方法的形参传入I类型的对象，通过处理后返回O型的对象。开发者可以实现自己的业务代码来对数据进行处理。

****ItemWriter****

ItemReader代表着写操作，其接口如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | public abstract class OLItemWriter<T extends OutItem> implements ItemWriter<T> {  protected static final Log logger = LogFactory.getLog(OLItemWriter.class);  }  @Service("bt02Writer")  public class BT02Writer extends OLItemWriter<OutItem> {  @Override  public void write(List<? extends OutItem> items) throws Exception {  System.out.println("do nothing");  }  }    } |

框架已经提供了多种ItemWriter接口的实现类，包括对文本文件、XML文件、数据库、JMS消息等写的处理，当然我们也可以自己实现该接口。

### JobRepository

JobRepository用于存储任务执行的状态信息，比如什么时间点执行了什么任务、任务执行结果如何等等。框架提供了2种实现，一种是通过Map形式保存在内存中，当Java程序重启后任务信息也就丢失了，并且在分布式下无法获取其他节点的任务执行情况；另一种是保存在数据库中，并且将数据保存在下面6张表里：

* BATCH\_JOB\_INSTANCE
* BATCH\_JOB\_EXECUTION\_PARAMS
* BATCH\_JOB\_EXECUTION
* BATCH\_STEP\_EXECUTION
* BATCH\_JOB\_EXECUTION\_CONTEXT
* BATCH\_STEP\_EXECUTION\_CONTEXT

### JobParameters

从上面的代码片段可以看到，运行 Job 时，我们可以给 Job 传递参数，参数会保存到 BATCH\_JOB\_EXECUTION\_PARAMS 表中

### JobInstance

当我们运行一个 Job 时，Spring 会根据 JobParameters 查询 BATCH\_JOB\_INSTANCE 表，如果不存在则插入一条数据，如果存在则说明是失败后重新运行，Spring 会根据表中的状态信息在失败的地方重新运行。如果我们重新运行一个已经成功的 Job 会怎么样呢？就向上个例子演示的那样，会抛异常。

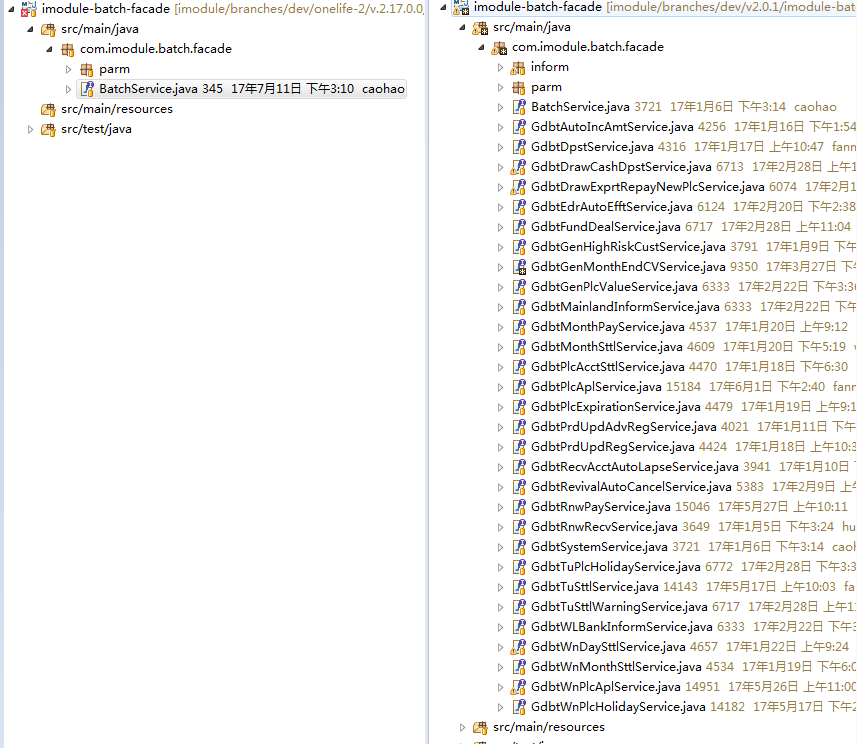
### JobExecution 和 ExecutionContext

每次运行 Job 时，Spring 都会往 BATCH\_JOB\_EXECUTION 表插入一个数据，保存 Job 开始，结束时间，状态等信息。   
有时候我们需要在 Job 的执行过程中初始化并访问一些变量，因此每个 JobExecution 都有一个 ExecutionContext 对象，它其实就是一个 Map，存储在 BATCH\_JOB\_EXECUTION\_CONTEXT 表中

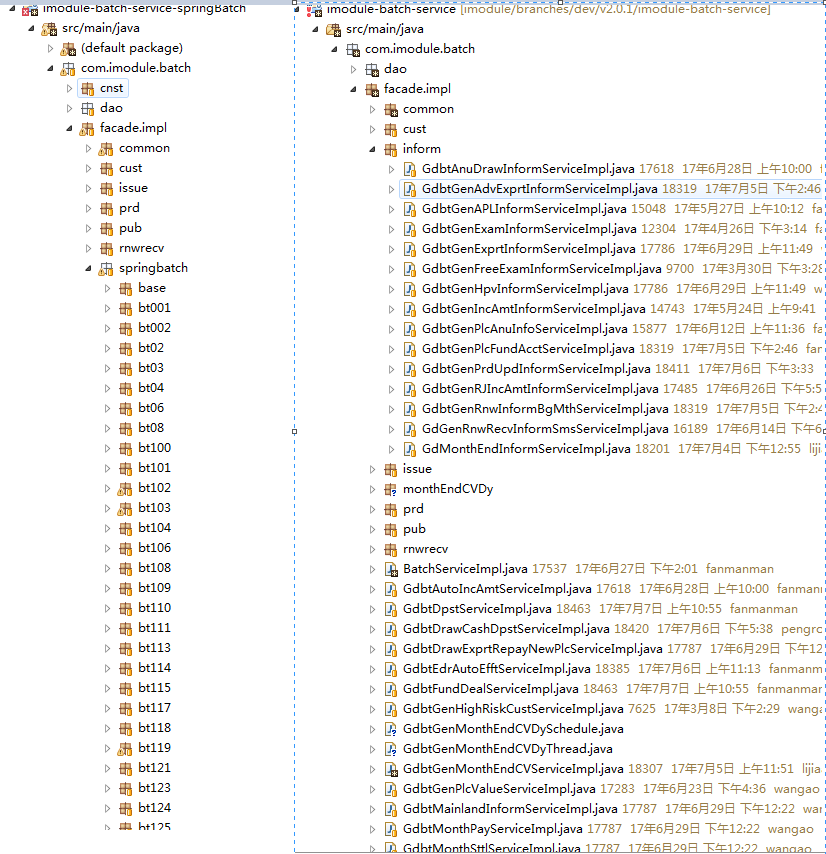
### StepExecution 和 ExecutionContext

每次运行 Step 时，Spring 都会往 BATCH\_STEP\_EXECUTION 表插入一个数据，保存 Step 开始，结束时间，状态等信息。   
有时候我们需要在 Step 的执行过程中初始化并访问一些变量，因此每个 StepExecution 都有一个 ExecutionContext 对象，它其实就是一个 Map，存储在 BATCH\_STEP\_EXECUTION\_CONTEXT 表中。

使用Spring Batch 后接口变得更加简洁，imodule-batch-facede 只剩下一个批处理入口服务一个接口；（如下图）



imodule-batch-service 结构也更加集中：



### 配置job和step：（可以使用xml的配置方式或Java Config，我们使用的是xml配置的方式）

|  |  |
| --- | --- |
| 1  2  3  4  5 | <!-- 123 config -->  <batch:job id=*"BATCH-123"* parent=*"lifeproBatchJob"*>  <batch:step id=*"bt123-step"* next=*"bt123hl-step"*>  <batch:tasklet transaction-manager=*"transactionManager"* task-executor=*"taskExecutor"* throttle-limit=*"12"*>  <batch:chunk reader=*"bt123Reader"* processor=*"bt123Processer"* writer=*"bt123Writer"* commit-interval=*"1"*  chunk-completion-policy=*""* skip-policy=*"btskipPolicy"*>  </batch:chunk>  <batch:transaction-attributes isolation=*"DEFAULT"* propagation=*"REQUIRED"*/>  </batch:tasklet>  <batch:listeners>  <batch:listener ref=*"onelifeFailLogListener"* />  </batch:listeners>  </batch:step>  <batch:step id=*"bt123hl-step"*>  <batch:tasklet transaction-manager=*"transactionManager"* task-executor=*"taskExecutor"* throttle-limit=*"12"*>  <batch:chunk reader=*"bt123ReaderHL"* processor=*"bt123ProcesserHL"* writer=*"bt123Writer"* commit-interval=*"1"*  chunk-completion-policy=*""* skip-policy=*"btskipPolicy"*>  </batch:chunk>  <batch:transaction-attributes isolation=*"DEFAULT"* propagation=*"REQUIRED"*/>  </batch:tasklet>  <batch:listeners>  <batch:listener ref=*"onelifeFailLogListener"* />  </batch:listeners>  </batch:step>  </batch:job>    } |

如上xml配置，一个Job任务可以有一个或多个Step步骤，通过tasklet属性throttle-limit配置多线程启用个数，根据实际情况配置多线程开启个数，可以有效的利用资源明显增加批处理的运行速度；transaction-manager 配置事务管理器，面向chunk的处理，支持一次读多次写，大幅提高批处理应用的处理效率，Step在开始处理的时候启动一个事务，通过指定的Spring PlatformTransactionManager周期性地提交Item的写操作，通过commit-interval来指定批次大小，但是我们在chunk配置时commit-interval = 1，为防止一次提交多条数据时数据异常回滚，暂时没有使用重跑机制，并指定对应的reader process writer Service;

### **使用Listener来监视job执行情况并及时做相应的处理**

|  |  |  |
| --- | --- | --- |
| **监听器** | | **说明** | | --- | |
| JobExecutionListener | 在 Job 开始之前(beforeJob)和之后(afterJob)触发 |
| StepExecutionListener | 在 Step 开始之前(beforeStep)和之后(afterStep)触发 |
| ChunkListener | 在 Chunk 开始之前(beforeChunk)，之后(afterChunk)和错误后(afterChunkError)触发 |
| ItemReadListener | 在 Read 开始之前(beforeRead)，之后(afterRead)和错误后(onReadError)触发 |
| ItemProcessListener | 在 Read 开始之前(beforeProcess)，之后(afterProcess)和错误后(onProcessError)触发 |
| ItemWriteListener | 在 Read 开始之前(beforeWrite)，之后(afterWrite)和错误后(onWriteError)触发 |
| SkipListener | 在 Read 开始之前(beforeWrite)，之后(afterWrite)和错误后(onWriteError)触发 |

Spring batch提供了大量的Listener来对job的各个执行环节进行全面的监控,实现Listener有两种方式，一种是继承自相应的接口，比如继承JobExecutionListener接口，另一种是使用annoation（注解）的方式。

<batch:job id=*"lifeproBatchJob"* abstract=*"true"*>

<batch:listeners>

<batch:listener ref=*"lifeproContextListener"* />

<batch:listener ref=*"onelifeLogListener"*/>

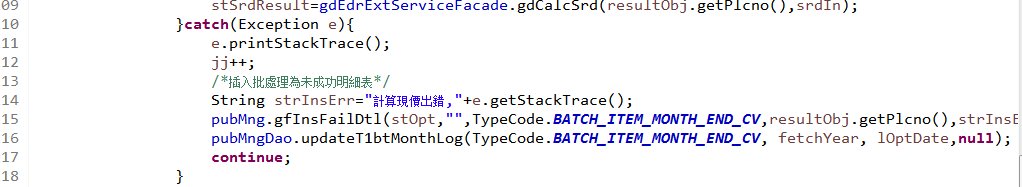
</batch:listeners>

<batch:validator ref=*"olJobParametersValidator"* />

</batch:job>

如上配置所示：通过配置上下文监听，参数校验监听，日志监听等大大简化了代码量，无需手动去校验参数的准确及一些必须的参数如批处理批次号，运行时间，操作信息等通过上下文监听设置到ExecutionContext中，而使用时直接通过stepExecution.getJobParameters()获取到对应的参数，减少数据之间的传递；

*onelifeLogListener* 极大的减少了业务处理时对日志的操作代码量及代码可读性增加，只专注于业务的处理：



*onelifeLogListener* 通过@OnSkipInProcess/@OnSkipInWrite注解的方式对异常进行监听，即通过手动抛出throw new OLBatchBizException 业务异常时，监听异常，回滚事务并插入失败日志Batch\_Log\_Faildtl；

最后*onelifeLogListener*通过@AfterJob注解在JOB完成后监听 自动插入汇总日志Batch\_Log\_Summary表；

使用SPRING BATCH框架后，运行速度明显提高：

|  |  |  |  |
| --- | --- | --- | --- |
| 批处理 | 总条数 | 耗时（SpringBatch） | 耗时 |
| BATCH-126 | 3000左右 | 8分钟 |  |
| BATCH-103 | 55W左右 | 一个半小时 | 跑不动 |
| BATCH-16 | 5W | 两个小时 |  |
| BATCH-58 | 1千以内 | 一到两分钟 |  |
| BATCH-130 | 2千以内 | 两分钟 |  |