

完整配置文件参见：ch08/job/job-composite.xml。

代码清单 8-24 配置 CompositeItemProcessor

```
1. <bean:bean id="compositeItemProcessor"
2.   class="org.springframework.batch.item.support.
3.   CompositeItemProcessor">
4.   <bean:property name="delegates">
5.     <bean:list>
6.       <bean:ref bean="partTranslateItemProcessor" />
7.       <bean:ref bean="validatorProcessor" />
8.     </bean:list>
9.   </bean:property>
10.  </bean:bean>
```

其中，3 行：属性 delegates 定义需要组合的处理器，本示例使用数据转换处理器与数据校验处理器，两个处理器的具体配置可以参见前面的章节。

5 行：引用部分数据转换处理器，修改 Item 的地址信息。

6 行：引用数据校验处理器，保证 Item 的消费金额要大于 0。

配置 CompositeItemProcessor 的 Job，参见代码清单 8-25。

完整配置文件参见：ch08/job/job-composite.xml。

代码清单 8-25 配置 compositeJob

```
1. <job id="compositeJob">
2.   <step id="compositeStep">
3.     <tasklet transaction-manager="transactionManager">
4.       <chunk reader="flatFileItemReader" writer="flatFileItemWriter"
5.         processor="compositeItemProcessor" commit-interval="2"
6.         skip-limit="5">
7.         <skippable-exception-classes>
8.           <include class="org.springframework.batch.item.
9.                         validator.ValidationException"/>
10.        </skippable-exception-classes>
11.        <listeners>
12.          <listener ref="filterCountStepExecutionListener" />
13.          <listener ref="skipCountStepExecutionListener" />
14.        </listeners>
15.      </chunk>
16.    </tasklet>
17.  </step>
18. </job>
```

使用代码清单 8-26 执行定义的 compositeJob。

完整代码参见：test.com.juxtapose.example.ch08.JobLaunchComposite。

代码清单 8-26 执行 compositeJob

```
1. JobLaunchBase.executeJob("ch08/job/job-composite.xml", "compositeJob",
2.                         new JobParametersBuilder().addDate("date", new Date())
3.                         .addString("filter", "true"));
```

经过部分数据处理后写入的文件内容参见代码清单 8-27。

代码清单 8-27 处理后的数据清单

```
1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road,tom
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road,tom
3. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road,tom
```

所有为消费金额为负数的行被过滤掉，每条记录的地址后面增加了名字信息。

8.6 服务复用

8.6.1 ItemProcessorAdapter

ItemProcessorAdapter 结构关键属性

ItemProcessorAdapter 持有服务对象，并调用指定的操作来完成 ItemProcessor 中定义的 process 功能。需要注意的是：已经存在的服务需要能够直接处理读提供的 Item 对象，即参数必须是读输出的 Item 的具体类型；返回值类型必须是 ItemWrite 阶段输入的参数类型。

ItemProcessorAdapter 和现有服务之间的关系参见图 8-9。

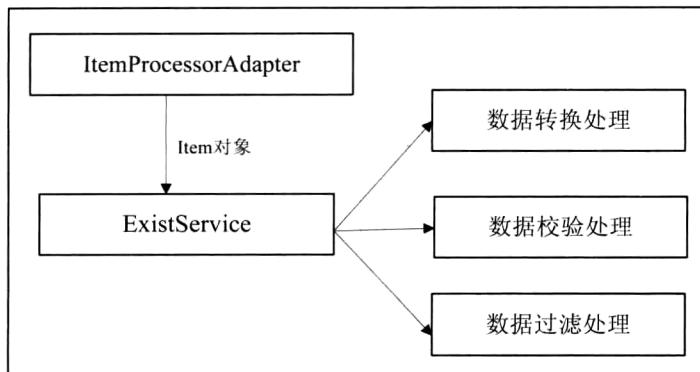


图 8-9 ItemProcessorAdapter 和现有服务之间的关系

ItemProcessorAdapter 类关系图参见图 8-10。

ItemProcessorAdapter 实现接口 ItemProcessor 并继承 AbstractMethodInvokingDelegator，后者提供了调用代理服务的一系列方法；ExistService 表示现存的服务。

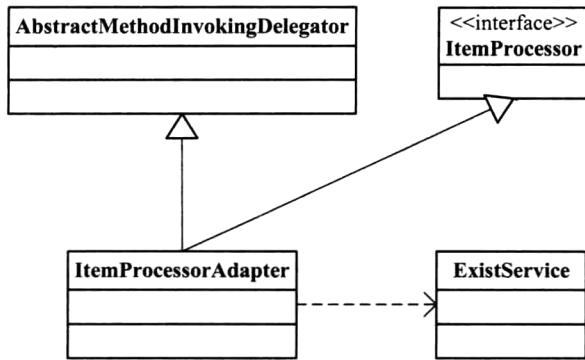


图 8-10 ItemProcessorAdapter 类关系图

ItemProcessorAdapter 关键属性

表 8-4 ItemProcessorAdapter 关键属性

ItemProcessorAdapter 属性	类 型	说 明
targetObject	Object	需要调用的目标服务对象
targetMethod	String	需要调用的目标操作名称
arguments	Object[]	需要调用的操作参数。 默认不需要传递该参数， 默认情况下将每次处理的 Item 对象作为参数传入

表 8-4 给出了 ItemProcessorAdapter 的关键属性。在配置 ItemProcessorAdapter 时候只需要指定上面的前两个属性即可，参数 arguments 默认情况下将每次处理的 Item 对象作为参数传入。

配置 ItemProcessorAdapter

目前已经有服务（com.juxtapose.example.ch08.ExistService）负责对信用卡账单信息进行验证，接下来我们使用 ExistService 作为示例来演示如何使用 ItemProcessorAdapter。

已经存在的服务 ExistService 的示例代码参见代码清单 8-28。

完整代码参见：com.juxtapose.example.ch08.ExistService。

代码清单 8-28 已存在服务 ExistService 示例代码

```

1.  public class ExistService {
2.      public CreditBill validate(CreditBill creditBill) throws Validation
Exception {
3.          if(Double.compare(0, creditBill.getAmount()) >0) {
4.              throw new ValidationException("Credit bill cannot be negative!");
5.          }
}
  
```

```
6.         return creditBill;
7.     }
8. }
```

本服务模拟已经存在的服务，validate 操作负责对账单对象进行验证，不通过会抛出验证异常；通过会直接返回信用卡账单对象。

接下来我们学习如何配置 ItemProcessorAdapter，具体配置代码参见代码清单 8-29。

完整配置参见文件：ch08/job/job-reuse-service.xml。

代码清单 8-29 配置 ItemProcessorAdapter

```
1. <bean:bean id="reuseServiceProcessor"
2.   class="org.springframework.batch.item.adapter.
3.   ItemProcessorAdapter">
4.   <bean:property name="targetObject" ref="existService"/>
5.   <bean:property name="targetMethod" value="validate"/>
6. </bean:bean>
7.
8. <bean:bean id="existService"
  class="com.juxtapose.example.ch08.ExistService"/>
```

其中，1~5 行：复用现有的服务完成 ItemProcessorAdapter 的配置，属性 targetObject 使用定义的已存服务 existService；属性 targetMethod 指定调用 validate 操作（参数为 CreditBill 的）。

7~8 行：声明现存的服务。

截止到目前我们配置完了如何使用现有的服务，通过 ItemProcessorAdapter 可以轻松方便地使用现有的服务功能，避免重复发明新的轮子。

8.7 拦截器

Spring Batch 框架在 ItemProcessor 执行阶段提供了拦截器，使得在 ItemProcessor 执行前后能够加入自定义的业务逻辑。ItemProcessor 执行阶段拦截器接口为：org.springframework.batch.core.ItemProcessListener<T, S>。

8.7.1 拦截器接口

接口 ItemProcessListener 的定义参见代码清单 8-30。

代码清单 8-30 ItemProcessListener 接口定义

```
1. public interface ItemProcessListener<T, S> extends StepListener {
2.     void beforeProcess(T item);
3.     void afterProcess(T item, S result);
4.     void onProcessError(T item, Exception e);
5. }
```

其中，2行：beforeProcess 在处理操作前触发该操作。

3行：afterProcess 在处理操作后触发该操作。

4行：onProcessError 在处理操作发生异常的时候触发该操作。

为 ItemProcessor 配置拦截器，参见代码清单 8-31。

完整配置参见文件：/ch08/job/job-listener.xml。

代码清单 8-31 配置 ItemProcessor 拦截器

```
1.      <job id="translateJob">
2.          <step id="translateStep">
3.              <tasklet transaction-manager="transactionManager">
4.                  <chunk reader="flatFileItemReader" writer="partTranslate
FlatFileItemWriter"
5.                      processor="partTranslateItemProcessor"
6.                      commit-interval="2">
7.                      <listeners>
8.                          <listener ref="sysoutItemProcessListener"></listener>
9.                          <listener ref="sysoutAnnotationItemProcessListener">
10.                         </listener>
11.                     </listeners>
12.                 </chunk>
13.             </tasklet>
14.         </step>
15.     </job>
16.     <bean:bean id="sysoutItemProcessListener"
17.                 class="com.juxtapose.example.ch08.listener.
SystemOutItemProcessListener">
18.     </bean:bean>
19.     <bean:bean id="sysoutAnnotationItemProcessListener"
20.                 class="com.juxtapose.example.ch08.listener.SystemOutAnnotation">
21.     </bean:bean>
```

其中，6~9行：为作业的读配置2个拦截器。

14~16行：定义拦截器 sysoutItemProcessListener，该拦截器实现接口 ItemProcessListener。

18~20行：定义拦截器 sysoutAnnotationItemProcessListener，该拦截器通过 Annotation 方式定义。

SystemOutItemProcessListener 的代码参见代码清单 8-32。

完整代码参见：com.juxtapose.example.ch08.listener.SystemOutItemProcessListener。

代码清单 8-32 SystemOutItemProcessListener 类定义

```
1.  public class SystemOutItemProcessListener implements
2.                      ItemProcessListener<CreditBill, CreditBill> {
3.      public void beforeProcess(CreditBill item) {
```

```
4.         System.out.println("SystemOutItemProcessListener.beforeProcess()");
5.     }
6.     public void afterProcess(CreditBill item, CreditBill result) {
7.         System.out.println("SystemOutItemProcessListener.afterProcess()");
8.     }
9.     public void onProcessError(CreditBill item, Exception e) {
10.        System.out.println("SystemOutItemProcessListener.onProcessError()");
11.    }
12. }
```

8.7.2 拦截器异常

拦截器方法如果抛出异常会影响 Job 的执行，所以在执行自定义的拦截器时候，需要考虑对拦截器发生的异常做处理，避免影响业务。

如果拦截器发生异常，会导致 Job 执行的状态为"FAILED"。

配置了错误拦截器的作业配置参见参见代码清单 8-33。

完整配置参见文件：/ch08/job/job-listener.xml。

代码清单 8-33 配置了错误拦截器的作业 errorTranslateJob

```
1. <job id="errorTranslateJob">
2.   <step id="errorTranslateStep">
3.     <tasklet transaction-manager="transactionManager">
4.       <chunk reader="flatFileItemReader"
5.             writer="partTranslateFlatFileItemWriter"
6.             processor="partTranslateItemProcessor"
7.             commit-interval="2">
7.       <listeners>
8.         <listener ref="errorItemProcessListener"></listener>
9.       </listeners>
10.      </chunk>
11.    </tasklet>
12.  </step>
12. </job>
```

其中，7 行：errorItemProcessListener 在 beforeProcess 操作中抛出异常，会导致整个作业的失败，当前的 Job 实例状态被标记为" FAILED "。

8.7.3 执行顺序

在配置文件中可以配置多个 ItemProcessListener，拦截器之间的执行顺序按照 listeners 定义的顺序执行。beforeProcess 方法按照 listener 定义的顺序执行，afterProcess 方法按照相反的顺序执行。上面示例代码中执行顺序如下：

1. sysoutItemProcessListener 拦截器的 beforeProcess 方法;
2. sysoutAnnotationItemProcessListener 拦截器的 beforeProcess 方法;
3. sysoutAnnotationItemProcessListener 拦截器的 afterProcess 方法;
4. sysoutItemProcessListener 拦截器的 afterProcess 方法。

8.7.4 Annotation

Spring Batch 框架提供了 Annotation 机制，可以不实现接口 ItemProcessListener，直接通过 Annotation 的机制定义拦截器。为 ItemProcessListener 提供的 Annotation 有：

- @BeforeProcess;
- @AfterProcess;
- @OnProcessError。

ItemProcessListener 操作说明与 Annotation 定义参见表 8-5。

表 8-5 ItemProcessListener 操作说明与 Annotation 定义

操作	操作说明	Annotation
beforeProcess(T item)	在 ItemProcessor#process()之前执行	@ BeforeProcess
afterProcess(T item, S result)	在 ItemProcessor#process()之后执行	@ AfterProcess
onProcessError(T item, Exception e)	当 ItemProcessor#process()抛出异常时候触发该操作	@ OnProcessError

使用 Annotation 声明的拦截器的 Spring 配置文件和实现接口 ItemProcessListener 的拦截器配置一样，只需要在 listeners 节点中声明即可。

SystemOutAnnotation 的代码参见代码清单 8-34。

完整代码参见：com.juxtapose.example.ch08.listener.SystemOutAnnotation。

代码清单 8-34 SystemOutAnnotation 类定义

```

1. public class SystemOutAnnotation {
2.     @BeforeProcess
3.     public void beforeProcess(CreditBill item) {
4.         System.out.println("SystemOutAnnotation.beforeProcess()");
5.     }
6.
7.     @AfterProcess
8.     public void afterProcess(CreditBill item, CreditBill result) {
9.         System.out.println("SystemOutAnnotation.afterProcess()");
10.    }
11.
12.    @OnProcessError

```

```
13.     public void onProcessError(CreditBill item, Exception e) {  
14.         System.out.println("SystemOutAnnotation.onProcessError()");  
15.     }  
16. }
```

8.7.5 属性 Merge

Spring Batch 框架提供了多处配置拦截器执行，可以在 chunk 元素节点配置，可以在 tasklet 中配置；框架同样提供了 step 的抽象和继承的能力，可以在父 Step 中定义通用的属性，在子 step 中定义个性化的属性，通过 merge 属性可以定义是覆盖父中的设置、还是和父中的定义合并；chunk 元素中的 listeners 支持 merge 属性。

假设有这样一个场景，所有的 Step 都希望拦截器 sysoutItemProcessListener 能够执行，而拦截器 sysoutAnnotationItemProcessListener 则由每个具体的 Step 定义是否执行，通过抽象和继承属性可以完成上面的场景。

merge 属性配置代码参见代码清单 8-35。

代码清单 8-35 merge 属性配置

```
1.      <job id="mergeTranslateJob" >  
2.          <step id="subChunkStep" parent="abstractParentStep">  
3.              <tasklet transaction-manager="transactionManager">  
4.                  <chunk reader="flatFileItemReader" writer="partTranslate  
FlatFileItemWriter" processor="partTranslateItemProcessor"  
commit-interval="2">  
5.                      <listeners merge="true">  
6.                          <listener ref="sysoutAnnotationItemProcessListener">  
7.                          </listener>  
8.                      </listeners>  
9.                  </chunk>  
10.             </tasklet>  
11.         </step>  
12.  
13.         <step id="abstractParentStep" abstract="true">  
14.             <tasklet>  
15.                 <chunk commit-interval="2" >  
16.                     <listeners>  
17.                         <listener ref="sysoutItemProcessListener"></listener>  
18.                     </listeners>  
19.                 </chunk>  
20.             </tasklet>  
21.         </step>
```

其中，17行：定义抽象作业步 `abstractParentStep` 中的拦截器。

5~7行：通过 `merge` 属性，可以与父类中的拦截器配置进行合并，表示在 `subChunkStep` 中有两个拦截器会同时工作；属性 `merge` 的值为 `true`，表示父子合并即两处定义的都生效；实行 `merge` 的值为 `false`，表示父类的不再生效，被子类的定义覆盖掉。

通过 `merge` 属性合并的拦截器的执行顺序如下：首先执行父 `Step` 中定义的拦截器，然后执行子 `Step` 中定义的拦截器。

第 3 篇 高级篇

本篇重点讲述了批处理的高性能、高可靠性和并行处理的能力，分别向读者展示：如何实现作业流的控制包括顺序流、条件流、并行流，如何实现健壮的作业包括跳过、重试、重启等，如何实现扩展作业及并行作业包括多线程作业、并行作业、远程作业、分区作业等。本篇包含三个个章节。

第 9 章：介绍作业流，包括顺序 Flow、条件 Flow、并行 Flow、外部 Flow，最后向读者介绍了作业步 Step 之间如何数据共享和多种终止 Job 的方式，包括 end、stop、fail 三种方式。

第 10 章：介绍如何设计健壮的作业 Job，主要包括跳过 Skip、重试 Retry、重启 Restart 三种可靠的策略。

第 11 章：介绍如何设计高可靠、高性能、支持大并发处理的作业，重点向读者介绍了 Spring Batch 框架提供的四种扩展策略：多线程 Step、并行 Step、远程 Step 和分区 Step。

作业流 Step Flow

前面章节所有的示例中，每个 Job 中只有一个 Step。Spring Batch 框架支持一个 Job 中配置多个 Step，不同的 Step 之间可以顺序执行，也可以按照不同的条件有选择地执行（条件通常使用 Step 的退出状态决定），通过 next 元素或者 decision 元素来定义跳转规则；为了提高多个 Step 的执行效率，框架提供了 Step 并行执行的能力（通常该情况下需要 Step 之间没有任何的依赖关系，否则容易引起业务上的错误）。

批处理任务中有些任务有先后的执行顺序，还有些 Step 没有先后执行顺序的要求，可以在同一时刻并行作业。批处理框架提供了并行处理 Step 的能力，通过 Split 元素可以定义并行的作业流，提高 Job 的执行效率。

除了正常完成 Step 后终止作业的执行，Spring Batch 框架提供了其他终止 Job 的能力，通过 End、Stop、Fail 等元素可以在任意需要终止 Job 的 Step 中终止作业的执行。

为了提高 Step 的复用，Spring Batch 框架提供了 flow（作业流）、FlowStep、JobStep 的支持，三者类似 Java 领域中的继承和复用的功能，使得 Job、Flow、Step 可以复用。

不同的 Step 之间如果需要共享数据，通过执行上下文 ExecutionContext，可以在不同的作业步 Step 之间达到共享数据的目的。

9.1 顺序 Flow

顺序 Flow 是指在 Job 中定义多个 Step，每个 Step 之间按照定义好的顺序执行，任何一个 Step 的失败都会导致 Job 的失败。元素 step 提供了 next 属性，可以定义当前 Step 执行完毕后下一个需要执行的 Step。顺序 Flow 执行的示意图参见图 9-1，表示作业步 StepA、StepB、StepC 三者顺序执行。

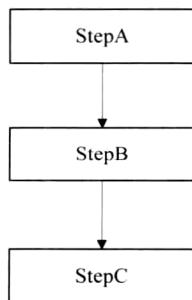


图 9-1 顺序 Flow 示意图

Step 元素 Schema 的定义参见图 9-2。

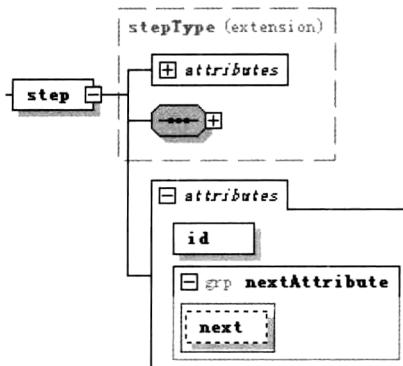


图 9-2 Step 元素 Schema 定义

next 属性定义当前 Step 执行完毕后接下来需要执行的 Step。

接下来使用信用卡账单处理的示例展示如何开发顺序的 Job。前面所有的章节直接读入 csv 格式的文件进行处理，在实际的企业应用中，账单文件通常以压缩的方式存放传输，批处理应用接收到的是压缩的文件，首选需要进行解压缩；解压缩后需要对文件是否存在，文件名称是否正确等进行校验；校验通过后才真正执行文件的解析，读取，数据保存等工作；最后需要将处理过程遗留下的临时文件清除，恢复工作环境。账单处理步骤参见图 9-3。

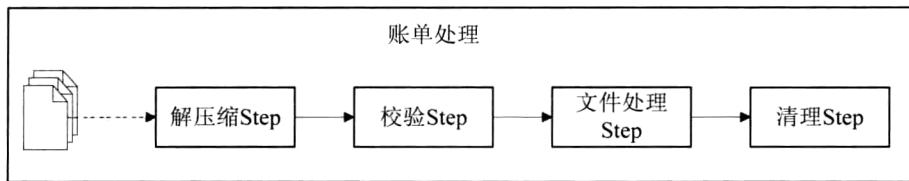


图 9-3 账单处理步骤

账单处理作业共有 4 个作业步，解压缩 Step、校验 Step、文件处理 Step 和清理 Step。配置顺序的作业步参见代码清单 9-1，在 Step 中通过 next 属性指定后续需要处理的 Step。

完整配置参见文件：ch09/job/job-sequential.xml。

代码清单 9-1 配置顺序的作业步

```
1.   <job id="sequentialJob" >
2.     <step id="decompressStep" parent="abstractDecompressStep"
3.       next="verifyStep" >
4.       <tasklet ref="decompressTasklet" />
5.     </step>
6.     <step id="verifyStep" next="readWriteStep">
7.       <tasklet ref="verifyTasklet" />
```

```

8.          </step>
9.          <step id="readWriteStep" parent="parentReadWriteStep"
10.             next="cleanStep" />
11.          <step id="cleanStep">
12.              <tasklet ref="cleanTasklet" />
13.          </step>
14.      </job>

```

其中，2~5 行：定义 decompressStep（解压缩 Step），完成文件的解压缩功能，next 属性定义了下一个作业步为校验 Step。

5~8 行：定义 verifyStep（校验 Step），完成对解压缩后文件的校验功能（文件是否存在，是否可读等），next 属性定义了下一个作业步为真正的文件读/写 Step。

9~10 行：定义 readWriteStep（文件读/写 Step），完成信用卡账单文件的读、处理、写的功能，next 属性定义了下一个作业步为清理 Step。

11~13 行：定义 cleanStep（清理 Step），完成临时目录的清理。

说明：顺序定义的 Step 中，执行顺序按照 Job 中定义的顺序执行，因此上面第一个执行的是 decompressStep。

9.2 条件 Flow

更多的业务场景需要根据作业步的执行结果决定后续调用哪个作业步，而不是像上节预先就定义好了作业的执行顺序。Spring Batch 框架提供了条件 Flow 来满足有选择的执行作业步的功能。

条件 Flow 执行示意图参见图 9-4，作业步 StepA 执行完毕后根据条件判断可能执行 StepB、也可能执行 StepC。

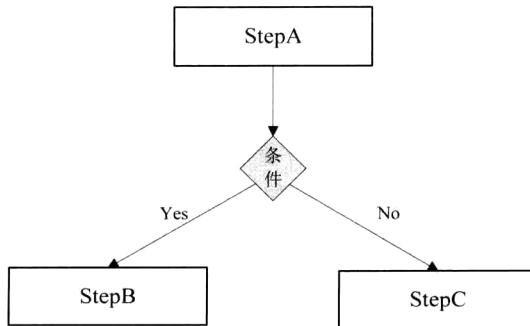


图 9-4 条件 Flow 示意图

条件 Flow 是指在 Job 中定义多个 Step，每个 Step 之间按照条件定义的规则执行。可以使用 Step 元素中的 next 元素定义条件；或者使用 Job 的子元素 decision 来定义跳转条件。

9.2.1 next

Step 中 next 元素的 Schema 定义参见图 9-5。

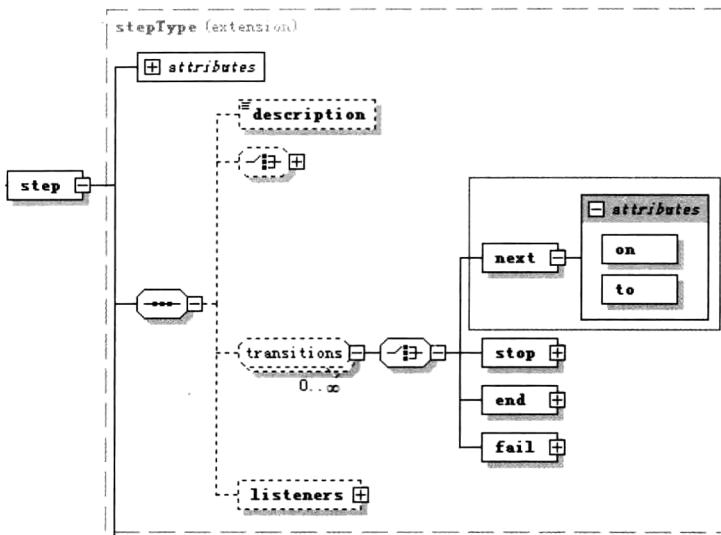


图 9-5 Step 中 next 元素 Schema 定义

next 属性说明参见表 9-1。

表 9-1 next 属性说明

属性	说 明	默 认 值
on	定义作业步的 ExitStatus (退出状态) 和 on 属性指定的值匹配的时候，则执行 to 指定的作业步。 on 属性的值可以是任意的字符串，同时支持通配符“*”、“?” “*”：表示退出状态为任何值都满足。 “？”：表示匹配一个字符，如 c?t，当作业的退出状态为 cat 的时候满足，如果是 cabt 则不满足	
to	当前 Step 执行完成后，to 属性元素指定下个需要执行的 Step	

使用上节中的示例，在校验 Step 环节增加条件判断，如果输入的文件满足条件则进入文件处理的 Step，否则跳过文件处理 Step，直接进入清理 Step 中。

条件账单处理步骤参见图 9-6。

配置条件作业步，参见代码清单 9-2，校验 Step 中增加了条件判断，通过 on 属性匹配作业步 Step 的退出状态，当 on 属性定义的值与 Step 退出状态匹配时，执行属性 to 对应的 Step。

完整配置参见文件：ch09/job/job-conditional.xml。

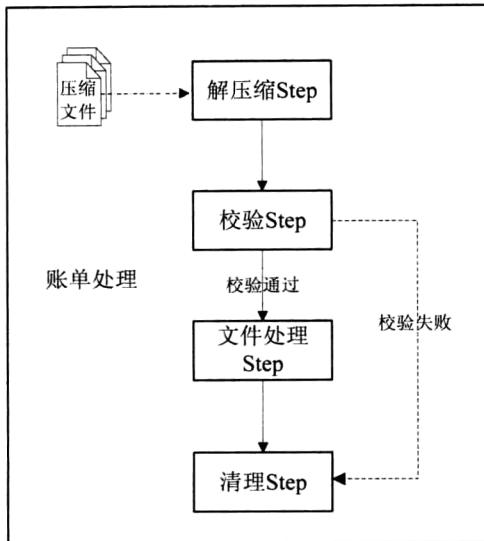


图 9-6 条件账单处理步骤

代码清单 9-2 配置条件作业步

```

1.      <job id="conditionalJob" >
2.          <step id="decompressStep" parent="abstractDecompressStep"
3.              next="verifyStep" >
4.                  <tasklet ref="decompressTasklet" />
5.              </step>
6.          <step id="verifyStep" >
7.              <tasklet ref="verifyTasklet" />
8.              <next on="*" to="readWriteStep" />
9.              <next on="SKIPTOCLEAN" to="cleanStep" />
10.             <listeners>
11.                 <listener ref="verifyStepExecutionListener" />
12.             </listeners>
13.         </step>
14.         <step id="readWriteStep" parent="parentReadWriteStep"
15.             next="cleanStep" />
16.         <step id="cleanStep">
17.             <tasklet ref="cleanTasklet" />
18.         </step>
19.     </job>

```

其中, 8 行: 属性 on 定义为 "*", 表示当前作业步 verifyStep 的退出状态非" SKIPTOCLEAN "的情况下都会跳转到 readWriteStep 中。

9 行: 属性 on 定义为" SKIPTOCLEAN ", 表示步 verifyStep 的返回值为" SKIPTOCLEAN "时, 会跳转到作业步 cleanStep 中。

10~12 行：声明作业步执行器，通过该拦截器修改 Step 的退出状态。

说明：属性 on 中定义的值，是 Step 的 ExitStatus 退出状态，可以通过拦截器 StepExecutionListener 修改 Step 的退出状态的值。

通过 Step 拦截器修改作业步 verifyStep 的退出状态，通常在拦截器 StepExecutionListener 的 afterStep()操作中修改退出状态的值。

拦截器 VerifyStepExecutionListener 的实现参见代码清单 9-3，在该拦截器中显式修改了 Step 的退出状态。

完整代码参见：com.juxtapose.example.ch09.listener.VerifyStepExecutionListener。

代码清单 9-3 VerifyStepExecutionListener 类定义

```
1. public class VerifyStepExecutionListener implements StepExecutionListener {  
2.     public void beforeStep(StepExecution stepExecution) { }  
3.     public ExitStatus afterStep(StepExecution stepExecution) {  
4.         String status = stepExecution.getExecutionContext()  
5.                         .getString(Constant.VERITY_STATUS);  
6.         if(null != status){  
7.             return new ExitStatus(status);  
8.         }  
9.         return stepExecution.getExitStatus();  
10.    }  
11. }
```

其中，4~5 行：从作业步上下文中获取退出状态的值，如果该值存在则返回；作业步上下文在整个作业执行的声明周期中生效，本例在作业步 verifyStep 的实现 verifyTasklet（参见代码清单 9-4）中将作业步的状态放入上下文中，然后在拦截器中修改。

9 行：如果作业步上下文中没有获取到，则直接返回当前作业步的退出状态值。

作业步 verifyStep 的实现代码参见代码清单 9-4。

完整代码参见：com.juxtapose.example.ch09.tasklet.VerifyTasklet。

代码清单 9-4 VerifyTasklet 类定义

```
1. public class VerifyTasklet implements Tasklet {  
2.     public RepeatStatus execute(StepContribution contribution,  
3.                                 ChunkContext chunkContext) throws Exception {  
4.         String status = creditService.verify(outputDirectory, readFileName);  
5.         if (null != status) {  
6.             chunkContext.getStepContext().getStepExecution()  
7.                 .getExecutionContext().put(Constant.VERITY_STATUS,status);  
8.         }  
9.         return RepeatStatus.FINISHED;  
10.    }  
11. }
```

其中，4行：调用服务获取校验的状态信息。

5~8行：将状态值放入当前的作业步上下文中。

通过next元素可以方便地设置条件Flow，on属性指定退出状态的匹配值，to属性指定当前Step完成后下一个执行的Step。

9.2.2 ExitStatus VS BatchStatus

Spring Batch框架有两个重要的状态，一个是9.2.1节提到的退出状态（ExitStatus），另一个是批处理状态（BatchStatus）。本节向读者分别介绍这两个状态的差异。

9.2.2.1 BatchStatus

批处理状态通常由批处理框架使用，用来记录Job或者Step的执行情况，在Job或者Step的重启中，批处理的状态起到关键作用（Job或者Step的重启状态判断使用的就是BatchStatus）。可以通过Job Execution和Step Execution获取当前Job、Step的批处理状态。

JobExecution.getStatus()操作可以获取作业Job的批处理状态。

StepExecution.getStatus()操作可以获取作业步Step的批处理状态。

批处理的状态是枚举类型，目前框架定义了8种类型，分别是完成COMPLETED、启动中STARTING、已启动STARTED、停止中STOPPING、已停止STOPPED、失败FAILED、废弃ABANDONED和未知UNKNOWN。

批处理状态属性列表参见表9-2。

表9-2 批处理状态属性

状态	说明
COMPLETED	表示完成状态，所有的Step都标记为COMPLETED后，Job会处于此状态
STARTING	表示作业正在启动状态，还没有启动完毕
STARTED	表示作业已经成功启动
STOPPING	表示作业正在停止中
STOPPED	表示作业停止完成
FAILED	表示作业执行失败
ABANDONED	表示当前下次重启Job时候需要废弃掉的Step，即不会被再次执行
UNKNOWN	表示未知的状态，该状态下重启Job会抛出错误

批处理状态在Job或者Step执行期间通过Job上下文或者Step上下文持久化到DB中，可以在表batch_job_execution查看Job的批处理状态（字段STATUS表示该状态），在表batch_step_execution查看Step的批处理状态（字段STATUS表示该状态）。

Job的批处理状态在表batch_job_execution中的示例，参见图9-7。

	JOB_EXECUTION_ID	STATUS	VERSION	JOB_INSTANCE_ID
1		152 STOPPED	2	151
2		153 COMPLETED	2	151
3		154 COMPLETED	2	152

图 9-7 Job 批处理状态示例

Step 的批处理状态在表 batch_step_execution 中的示例，参见图 9-8。

	STEP_EXECUTION_ID	STATUS	VERSION	STEP_NAME	JOB_EXECUTION_ID
1		221 COMPLETED	3	decompressStep	152
2		222 COMPLETED	3	verifyStep	152
3		223 COMPLETED	6	readWriteStep	153
4		224 COMPLETED	3	cleanStep	153
5		225 COMPLETED	3	decompressStep	154
6		226 COMPLETED	3	verifyStep	154
7		227 COMPLETED	6	readWriteStep	154
8		228 COMPLETED	3	cleanStep	154

图 9-8 Step 批处理状态示例

9.2.2.2 ExitStatus

退出状态表示 Step 执行后或者 Job 执行后的状态，该状态值可以被修改，通常用于条件 Flow 中。可以通过拦截器 StepExecutionListener 的 afterStep() 操作来修改退出状态的值。

StepExecutionListener 接口声明参见代码清单 9-5。

代码清单 9-5 StepExecutionListener 接口定义

```
1. public interface StepExecutionListener extends StepListener {
2.     void beforeStep(StepExecution stepExecution);
3.     ExitStatus afterStep(StepExecution stepExecution);
4. }
```

其中，2 行：表示在 Step 执行之前调用该方法。

3 行：表示在 Step 执行之后调用该方法，读者需要注意，该操作的返回值是 ExitStatus，表示当次作业步执行后的退出状态；可以实现该接口修改 ExitStatus 的值。

如何通过拦截器设置退出状态的值可以参见 9.2.1 节的示例。

退出状态在 Job 或者 Step 执行期间通过 Job 上下文或者 Step 上下文持久化到 DB 中，可以在表 batch_job_execution 查看 Job 的退出状态（字段 EXIT_CODE 表示该状态），在表 batch_step_execution 查看 Step 的退出状态（字段 EXIT_CODE 表示该状态）。

Job 的退出状态在表 batch_job_execution 中的示例参见图 9-9。

	JOB_EXECUTION_ID	STATUS	EXIT_CODE	VERSION
1		152 STOPPED	STOPPED	2
2		153 COMPLETED	COMPLETED	2
3		154 COMPLETED	COMPLETED	2

图 9-9 Job 退出状态示例

Step 的退出状态在表 batch_step_execution 中的示例参见图 9-10。

	STEP_EXECUTION_ID	STATUS	EXIT_CODE	VERSION	STEP_NAME
1		221	COMPLETED	COMPLETED	3 decompressStep
2		222	COMPLETED	COMPLETED	3 verifyStep
3		223	COMPLETED	COMPLETED	6 readWriteStep
4		224	COMPLETED	COMPLETED	3 cleanStep
5		225	COMPLETED	COMPLETED	3 decompressStep
6		226	COMPLETED	COMPLETED	3 verifyStep
7		227	COMPLETED	COMPLETED	6 readWriteStep
8		228	COMPLETED	COMPLETED	3 cleanStep

图 9-10 Step 退出状态示例

9.2.3 decision 条件

Spring Batch 框架提供了 decision 支持条件 Flow。decision 是 Job 的子元素，所有 decision 的实现必须实现接口 JobExecutionDecider。

元素 decision 的 Schema 定义参见图 9-11。

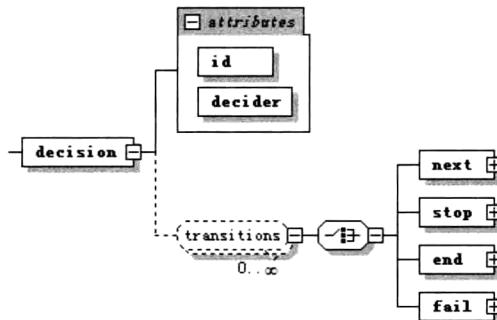


图 9-11 decision 的 Schema 定义

decision 属性说明参见表 9-3，核心属性为 id、decider。

表 9-3 decision 属性说明

属性	说 明	默 认 值
id	Job 定义中的唯一 ID	
decider	条件决定器的实现，需要实现接口 JobExecutionDecider	

Decision 中可以定义子元素 next, stop, end, fail 等，用于进行跳转的具体定义工作；如何使用 stop, end, fail 请参见 9.6 节的描述，next 元素参见 9.2.1 节描述。

JobExecutionDecider 接口

JobExecutionDecider 接口声明参见代码清单 9-6。

接口全称：org.springframework.batch.core.job.flow.JobExecutionDecider。

代码清单 9-6 JobExecutionDecider 接口定义

```
1. public interface JobExecutionDecider {  
2.     FlowExecutionStatus decide(JobExecution jobExecution, StepExecution  
        stepExecution);  
3. }
```

其中，2 行：接口唯一方法，用来决定如何跳转，参数是作业执行器和作业步执行器；返回值为 FlowExecutionStatus，可以使用自带的枚举值，也可以自定义自己的 FlowExecutionStatus。

本节仍然使用信用卡账单处理的场景，整个作业需要四个作业步，分别是：解压缩 Step、校验 Step、文件处理 Step、清理 Step。为了提高 Job 处理的效率，需要在解压缩 Step 之后判断解压缩的文件是否存在，不存在直接跳转到清理 Step，存在则跳转到校验 Step、文件处理 Step。

账单处理场景示意图参见图 9-12。

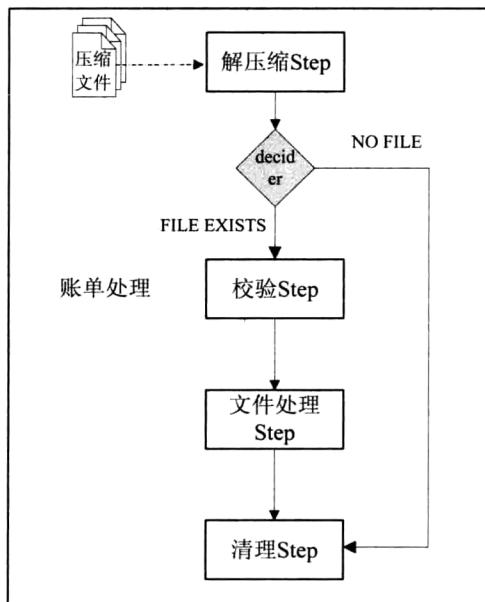


图 9-12 条件账单处理步骤

接下来我们首先实现文件是否存在 FileExistsDecider 类（代码实现参见代码清单 9-7），FileExistsDecider 实现接口 JobExecutionDecider，根据文件是否存在返回特定的退出状态标识，如果文件存在返回“FILE EXISTS”，不存在返回“NO FILE”。

完整代码参见：com.juxtapose.example.ch09.decider.FileExistsDecider。

代码清单 9-7 FileExistsDecider 类定义

```
1. public class FileExistsDecider implements JobExecutionDecider {  
2.     private CreditService creditService;  
3.     public FlowExecutionStatus decide(JobExecution jobExecution,
```

```

4.             StepExecution stepExecution) {
5.                 String readFileName = stepExecution.getJobParameters()
6.                         .getString(Constant.READ_FILE_NAME);
7.                 String workDirectory = stepExecution.getJobParameters()
8.                         .getString(Constant.WORK_DIRECTORY);
9.                 if(creditService.exists(workDirectory+readFileName)) {
10.                     return new FlowExecutionStatus("FILE EXISTS");
11.                 } else {
12.                     return new FlowExecutionStatus("NO FILE");
13.                 }
14.             }
15.             .....
16.         }

```

其中，10 行：文件存在返回退出状态标识为“FILE EXISTS”。

12 行：文件不存在返回退出状态标识为“NO FILE”。

最后我们来配置 decision 的 Job，参见代码清单 9-8。根据类 FileExistsDecider 的返回退出状态标识决定 Step 的流转，如果返回标识为“FILE EXISTS”则执行校验 Step，如果返回标识为“NO FILE”则执行清理 Step。

完整配置参见文件：ch09/job/job-conditional-decider.xml。

代码清单 9-8 配置 decision 的 Job

```

1.      <job id="conditionalDeciderJob" >
2.          <step id="decompressStep" parent="abstractDecompressStep"
3.              next="decision" >
4.                  <tasklet ref="decompressTasklet" />
5.              </step>
6.
7.              <decision id="decision" decider="fileExistsDecider">
8.                  <next on="FILE EXISTS" to="verifyStep" />
9.                  <next on="NO FILE" to="cleanStep" />
10.             </decision>
11.
12.             <step id="verifyStep" >
13.                 <tasklet ref="verifyTasklet" />
14.                 <next on="*" to="readWriteStep" />
15.                 <next on="SKIPTOCLEAN" to="cleanStep" />
16.             </step>
17.
18.             <step id="readWriteStep" parent="parentReadWriteStep"
19.                 next="cleanStep" />
20.
21.             <step id="cleanStep">
22.                 <tasklet ref="cleanTasklet" />

```

```

22.      </step>
23.    </job>
24.
25.    <bean:bean id="fileExistsDecider"
26.      class="com.juxtapose.example.ch09.decider.FileExistsDecider">
27.        <bean:property name="creditService" ref="creditService" />
28.      </bean:bean>

```

其中，1~3 行：作业步 decompressStep 属性 next 直接指向 fileExistsDecider，表示作业步 decompressStep 执行成功后，会跳转到 fileExistsDecider，由 fileExistsDecider 决定后续的作业步。

7~10 行：定义 decision，属性 decider 定义条件决定器的实现为 fileExistsDecider。

25~28 行：定义条件决定器 fileExistsDecider 的具体实现。

9.3 并行 Flow

批处理任务中有些任务有先后的执行顺序，还有些 Step 没有先后执行顺序的要求，可以在同一时刻并行作业，批处理框架提供了并行处理 Step 的能力，通过 Split 元素可以定义并行的作业流，为 split 定义执行的线程池，从而提高 Job 的执行效率。

Spring Batch 框架提供了 split 元素来执行并行作业的能力。

元素 split 的 Schema 定义参见图 9-13。

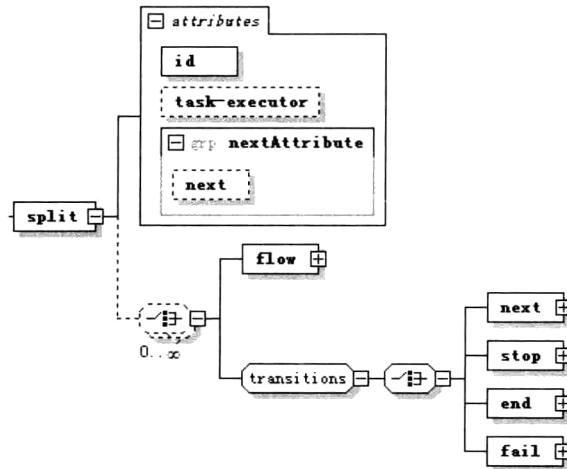


图 9-13 元素 split 的 Schema 定义

split 元素的属性说明参见表 9-4。

split 元素说明参见表 9-5。split 可以定义子元素 next, stop, end, fail 等，用于进行跳转的具体定义工作；如何使用 stop, end, fail 请参见 9.6 节的描述；split 的子元素 flow 用来定义并行处理的作业，并列的 flow 表示可以并行处理的任务。

表 9-4 split 元素的属性说明

属性	说 明	默 认 值
id	定义 split 的唯一 ID, 全局需要保证 id 唯一	
task-executor	任务执行处理器, 定义后表示采用多线程执行任务, 需要考虑多线程执行任务时候的安全性	如果不定义的话, 默认使用同步线程执行器: SyncTaskExecutor
next	当前 split 中所有的 flow 执行完毕后, 接下来执行的 Step	

表 9-5 split 元素说明

属性	说 明
flow	用来定义并行处理的作业, 并列的 flow 表示可以并行处理的任务; split 元素下面可以定义多个 flow 节点
next	根据退出状态定义下一步需要执行的 Step
stop	根据退出状态决定是否退出当前的任务, 同时 Job 也会停止, 作业状态为"STOPPED"
fail	根据退出状态决定是否失败当前的任务, 同时 Job 也会停止, 作业状态为"FAILED"
end	根据退出状态决定是否结束当前的任务, 同时 Job 也会停止, 作业状态为"COMPLETED"

本节仍然使用信用卡账单处理的场景, 现在需要同时处理 10、11 月份的账单, 处理两个月份的账单任务没有依赖关系因此可以并行处理。接下来我们并行完成 10、11 月份的账单的处理, 两个任务都完成后继续执行清理的 Step, 具体参见图 9-14。

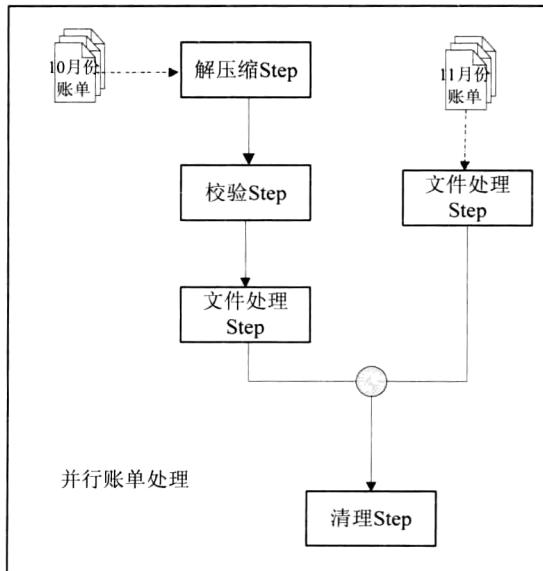


图 9-14 并行场景示意图

配置 split 的 Job 参见代码清单 9-9。

完整配置参见文件：ch09/job/job-split.xml。

代码清单 9-9 配置 split 的 Job

```
1.      <job id="splitJob" >
2.          <split id="split" task-executor="taskExecutor" next="cleanStep">
3.              <flow>
4.                  <step id="decompressStep" parent="abstractDecompressStep"
5.                      next="verifyStep" >
6.                      <tasklet ref="decompressTasklet" />
7.                  </step>
8.                  <step id="verifyStep" next="readWrite_10Step">
9.                      <tasklet ref="verifyTasklet" />
10.                 </step>
11.                 <step id="readWrite_10Step" parent="parentReadWriteStep"/>
12.             </flow>
13.             <flow>
14.                 <step id="readWrite_11Step" parent="parentReadWriteStep">
15.                     <listeners>
16.                         <listener ref="splitStepExecutionListener"></listener>
17.                     </listeners>
18.                 </step>
19.             </flow>
20.         </split>
21.         <step id="cleanStep">
22.             <tasklet ref="cleanTasklet" />
23.         </step>
24.     </job>
```

其中，2 行：使用 split 元素定义并行处理的作业步，属性 task-executor 用来定义执行多个 flow 时候使用的线程池，属性 next 定义当前的 split 执行完毕后接下来执行的 step 任务。

3~12 行：定义处理 10 月份账单的作业 flow，该 flow 定义了 3 个作业步，分别是 decompressStep、verifyStep、readWrite_10Step。

13~19 行：定义处理 11 月份账单的作业 flow，在 Job 执行期间和上面的 flow 可以并行执行。

21~23 行：定义 split 执行完毕后接下来执行的 cleanStep。

说明：split 内部的 flow 之间是并行执行的，只有当 split 内部所有的 flow 执行完毕后，才会继续执行 next 属性指定的 Step。可以为 split 指定执行的线程池。

接下来我们定义 split 使用的线程池 task-executor，具体参见代码清单 9-10。

代码清单 9-10 配置 split 使用的线程池

```
1.      <bean:bean id="taskExecutor"
2.          class="org.springframework.scheduling.concurrent.
```

```

    ThreadPoolTaskExecutor">
3.     <bean:property name="corePoolSize" value="5"/>
4.     <bean:property name="maxPoolSize" value="15"/>
5.   </bean:bean>

```

其中，3行：属性 corePoolSize 定义线程池的初始大小为 5。

4行：属性 maxPoolSize 定义线程池的最大值为 15。

通过属性 task-executor 指定对应的线程池，并行处理的作业步之间使用线程池中线程进行作业。

9.4 外部 Flow 定义

Spring Batch 框架提供了外部定义 Flow 的能力，通过在 Job 外部定义 Flow 可以做到复用。框架本身提供了三种方式的 Flow 复用，分别是：

- (1) 作业引用外部 Flow；
- (2) 作业步引用外部 Flow，称之为 FlowStep；
- (3) 作业步引用外部定义的 Job，称之为 JobStep。

接下来我们学习这三种外部 Flow 的使用方式。

9.4.1 Flow

Spring Batch 框架提供了 Flow 元素用于声明一组 Step，使用 Flow 元素声明的作业流可以被 Job 直接使用。就相当于在 Job 内部定义了一组 Step。

元素 flow 的 Schema 定义参见图 9-15。

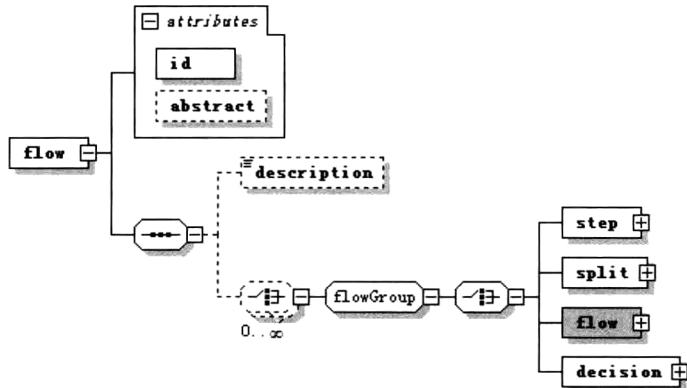


图 9-15 元素 flow 的 Schema 定义

flow 元素的属性说明参见表 9-6。

表 9-6 flow 元素的属性说明

属性	说 明	默 认 值
id	定义 flow 的唯一 ID, 全局需要保证 id 唯一	
abstrace	当前 flow 是否是抽象的	false

flow 的子元素可以是 step、split、flow、decision 四种。step 用于直接定义一个作业步；split 用于定义并行的作业步；flow 用于定义一批作业流；decision 用于进行作业的条件判断。

一个典型的 Flow 的定义参见代码清单 9-11。

完整配置文件参见：ch09/job/job-external-flow.xml。

代码清单 9-11 典型的 Flow 定义示例

```

1.   <flow id="flow1">
2.     <step id="decompressStep" parent="abstractDecompressStep"
3.       next="verifyStep" >
4.       <tasklet ref="decompressTasklet" />
5.     </step>
6.     <step id="verifyStep" next="readWriteStep">
7.       <tasklet ref="verifyTasklet" />
8.     </step>
9.     <step id="readWriteStep" parent="parentReadWriteStep"/>
  </flow>

```

定义当前的 flow 的 id 为 “flow1”，内部定义了三个作业步，分别是“decompressStep”、“verifyStep”、“readWriteStep”。

接下来描述 Job 中 flow 元素的 Schema 定义，参见图 9-16。

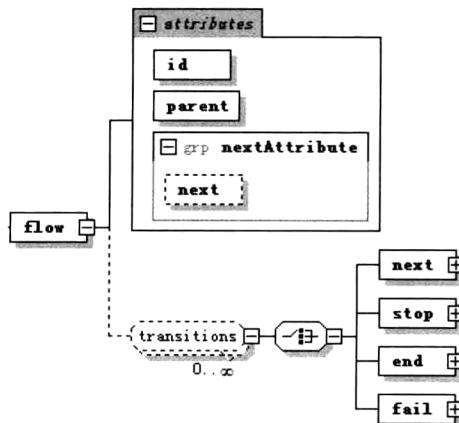


图 9-16 flow 元素的 Schema 定义

job 中 flow 元素的属性说明，参见表 9-7。

表 9-7 job 中 flow 元素的属性说明

属性	说 明	默 认 值
id	定义 flow 的唯一 ID, 全局需要保证 id 唯一	
parent	用于指定外部定义的 flow 的 ID	
next	当前 flow 中所有的 step 执行完毕后, 接下来执行的 step	

接下来向读者给出如何在 Job 中使用 flow 来配置作业, 具体参见代码清单 9-12。

完整配置文件参见: ch09/job/job-external-flow.xml。

代码清单 9-12 Job 引用外部 Flow

```

1.      <job id="externalFlowJob" >
2.          <flow id="externalFlowJob.flow1" parent="flow1" next="cleanStep"/>
3.          <step id="cleanStep">
4.              <tasklet ref="cleanTasklet" />
5.          </step>
6.      </job>

```

其中, 2 行: 使用 flow 定义 Job 的作业流, 使用 parent 属性指定使用 id 为"flow1"的外部定义。

本处使用 flow 元素, 在执行过程中相当于将 flow 的定义在此处复制了一份; 上面的配置和代码清单 9-13 完成的功能完全一致。

代码清单 9-13 Job 引用 flow 的等价配置

```

1.      <job id="externalFlowJob" >
2.          <step id="decompressStep" parent="abstractDecompressStep"
3.              next="verifyStep" >
4.                  <tasklet ref="decompressTasklet" />
5.              </step>
6.          <step id="verifyStep" next="readWriteStep">
7.              <tasklet ref="verifyTasklet" />
8.          </step>
9.          <step id="readWriteStep" parent="parentReadWriteStep"/>
10.         <step id="cleanStep">
11.             <tasklet ref="cleanTasklet" />
12.         </step>
13.     </job>

```

本处例子直接使用 step 定义, 和在外部使用 flow 定义没有任何区别, 唯一的好处在于, 通过 flow 可以进行 Step 的组装, 提供较好的服务复用功能。

执行使用 flow 定义的 Job, 查看数据库的源信息可以看到, 作业 externalFlowJob 先后执行了 4 个 Step。(执行代码入口: test.com.juxtapose.example.ch09.JobLaunchExternalFlow)

作业实例表信息, 参见图 9-17。

	JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
1	154	0	externalFlowJob	55b94c2720e510eaa4e9bd9435c85dc

图 9-17 作业实例表信息

当前的 Job Instance 的 ID 为 154，作业名字为 externalFlowJob。

作业执行器表信息，参见图 9-18。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	CREATE_TIME
1	156	2	154	2013-10-14 22:09:21

图 9-18 作业执行器表信息

当前作业执行器的 ID 为 156，对应的 Job Instance 为 154；

作业步执行器表信息，参见图 9-19。

	STEP_EXECUTION_ID	VERSION	STEP_NAME	JOB_EXECUTION_ID
1	234	3	decompressStep	156
2	235	3	verifyStep	156
3	236	6	readWriteStep	156
4	237	3	cleanStep	156

图 9-19 作业步执行器表信息

分别执行了四个作业步，对应的作业执行器的 ID 为 156。

9.4.2 FlowStep

FlowStep 是指 Step 元素使用外部定义的 Flow，FlowStep 和直接使用 Flow 的区别在于前者将 Flow 作为一个单独的 Step 在 Job 执行期间出现（该单独的 Step 是将 Flow 中定义的所有 Step 包装在一个大的 Step 中）；后者没有单独的 Step，Flow 中定义了几个 Step，则执行期间有多少 Step。

接下来我们给出 FlowStep 的示例参见代码清单 9-14。

完整配置参见文件：ch09/job/job-external-flow-step.xml。

代码清单 9-14 FlowStep 示例

```

1.   <job id="externalFlowStepJob" >
2.     <step id="externalFlowStepJob.flow1" next="cleanStep">
3.       <flow parent="flow1" />
4.     </step>
5.     <step id="cleanStep">
6.       <tasklet ref="cleanTasklet" />
7.     </step>
8.   </job>
9.
10.  <flow id="flow1">
11.    <step id="decompressStep" parent="abstractDecompressStep" />

```

```

    next="verifyStep" >
12.          <tasklet ref="decompressTasklet" />
13.      </step>
14.      <step id="verifyStep" next="readWriteStep">
15.          <tasklet ref="verifyTasklet" />
16.      </step>
17.      <step id="readWriteStep" parent="parentReadWriteStep"/>
18.  </flow>

```

Step Flow 的配置是使用 Step, Step 的实现内嵌了外部定义的 flow。

使用 FlowStep 的好处在于, 在 Job 中将外部定义的 Flow 作为一个完整的 Step; 同时 Flow 中定义的多个 Step 在 Job 执行期间作为完整 Step 的一个子活动, 当 Flow 中所有的 Step 执行完毕后 FlowStep 才会执行完成。

执行使用 FlowStep 定义的 Job, 查看数据库的源信息可以看到, 作业 externalFlowStepJob 先后执行了 4 个 Step。(执行代码入口: test.com.juxtapose.example.ch09.JobLaunchExternalFlowStep)

作业实例表信息参见图 9-20。

	JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
1	158	0	externalFlowStepJob	6f840e2318b821074478b28849181636

图 9-20 作业实例表信息

当前的 Job Instance 的 ID 为 158, 作业名字为 externalFlowStepJob。

作业执行器表信息参见图 9-21。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	CREATE_TIME
1	160	2	158	2013-10-14 22:59:36

图 9-21 作业执行器表信息

当前作业执行器的 ID 为 160, 对应的 Job Instance 为 158。

作业步执行器表信息参见图 9-22。

	STEP_EXECUTION_ID	STEP_NAME	JOB_EXECUTION_ID	START_TIME	END_TIME
1	248	externalFlowStepJob.flow1	160	2013-10-14 22:59:36	2013-10-14 22:59:37
2	249	decompressStep	160	2013-10-14 22:59:36	2013-10-14 22:59:36
3	250	verifyStep	160	2013-10-14 22:59:36	2013-10-14 22:59:37
4	251	readWriteStep	160	2013-10-14 22:59:37	2013-10-14 22:59:37
5	252	cleanStep	160	2013-10-14 22:59:37	2013-10-14 22:59:37

图 9-22 作业步执行器表信息

此处大家注意, 作业步 "externalFlowStepJob.flow1" 是 flow 中定义的三个 Step 的父, 当 flow1 中定义的三个作业步执行完毕后, 作业步 "externalFlowStepJob.flow1" 才执行完成, 请注意看开始时间和结束时间。

"externalFlowStepJob.flow1" 和 "flow1" 中 step 的生命周期参见图 9-23。

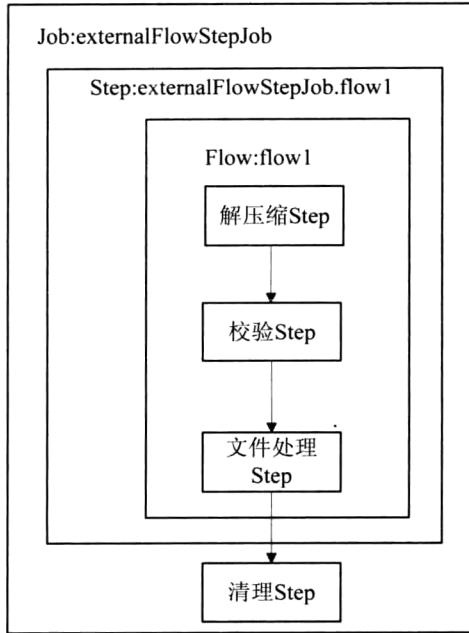


图 9-23 "externalFlowStepJob.flow1"和"flow1"中 step 的生命周期

9.4.3 JobStep

JobStep 是指 Step 元素使用外部定义的 Job， JobStep 和 FlowStep 的区别在于前者将 Job 作为一个单独的 Step 在 Job 执行期间出现（该单独的 Step 将 Job 中定义的所有 Step 包装在一个大的 Step 中；同时内部的 Job 在运行期间是一个完整的 Job 执行，同时又作为 Step 的子元素出现）；而后者将 Flow 作为一个单独的 Step 在 Job 执行期间出现（该单独的 Step 将 Flow 中定义的所有 Step 包装在一个大的 Step 中）。

接下来我们给出 JobStep 的示例参见代码清单 9-15。

完整配置参见文件：ch09/job/job-external-job.xml。

代码清单 9-15 JobStep 示例

```

1.      <job id="externalJobJobStep" >
2.          <step id="externalJobJob.flow1" next="cleanStep">
3.              <job ref="baseJob" />
4.          </step>
5.          <step id="cleanStep">
6.              <tasklet ref="cleanTasklet" />
7.          </step>
8.      </job>
9.
  
```

```

10.    <job id="baseJob">
11.        <step id="decompressStep" parent="abstractDecompressStep"
12.            next="verifyStep" >
13.                <tasklet ref="decompressTasklet" />
14.            </step>
15.        <step id="verifyStep" next="readWriteStep">
16.            <tasklet ref="verifyTasklet" />
17.        </step>
18.        <step id="readWriteStep" parent="parentReadWriteStep"/>
19.    </job>

```

本示例中定义两个 Job，分别是 baseJob、externalJobJobStep。baseJob 定义了三个作业步，分别是压缩、校验、读/写；在 externalJobJobStep 中定义了两个作业步，externalJobJob.flow1 和清理作业步，前者引用了 baseJob 作业作业步的实现。

因此在 externalJobJobStep 执行时候，会有两个 Job 执行，分别是 baseJob、externalJobJobStep。

执行使用 JobStep 定义的 Job，查看数据库的源信息可以看到，作业 externalJobJobStep 先后执行了 4 个 Step。（执行代码入口：test.com.juxtapose.example.ch09.JobLaunchExternalJobStep）

作业实例表信息参见图 9-24。

	JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
1	161	0	externalJobJobStep	604a64af02993207746dc96cdba7a5c0
2	162	0	baseJob	604a64af02993207746dc96cdba7a5c0

图 9-24 作业实例表信息

当前的 Job Instance 有两个，对应的 ID 分别为 161,162，作业名字为 externalJobJobStep、baseJob。

作业执行器表信息参见图 9-25。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	CREATE_TIME
1	163	2	161	2013-10-14 23:28:45
2	164	2	162	2013-10-14 23:28:45

图 9-25 作业执行器表信息

当前作业执行器的 ID 为 163，对应的 Job Instance 为 161。

当前作业执行器的 ID 为 164，对应的 Job Instance 为 162。

作业步执行器表信息参见图 9-26。

	STEP_EXECUTION_ID	STEP_NAME	JOB_EXECUTION_ID	START_TIME	END_TIME
1	258	externalJobJob.flow1	163	2013-10-14 23:28:45	2013-10-14 23:28:46
2	259	decompressStep	164	2013-10-14 23:28:45	2013-10-14 23:28:45
3	260	verifyStep	164	2013-10-14 23:28:46	2013-10-14 23:28:46
4	261	readWriteStep	164	2013-10-14 23:28:46	2013-10-14 23:28:46
5	262	cleanStep	163	2013-10-14 23:28:47	2013-10-14 23:28:47

图 9-26 作业步执行器表信息

此处大家注意，作业步"externalJobJob.flow1"是 baseJob 中定义的三个 Step 的父，当 baseJob 中定义的三个作业步执行完毕后，作业步" externalJobJob.flow1"才执行完成，请注意看开始时间和结束时间。

" externalJobJob.flow1"和"baseJob"中 step 的生命周期参见图 9-27。

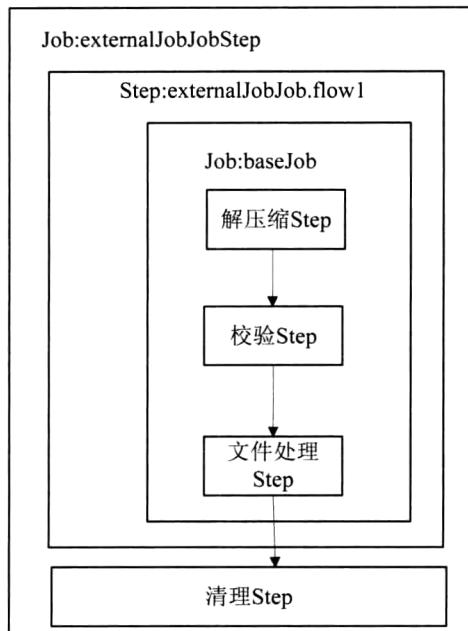


图 9-27 "externalJobJob.flow1"和"baseJob"中 step 的生命周期

9.5 Step 数据共享

Execution Context 是 Spring Batch 框架提供的持久化与控制的 key/value 对，能够让开发者在 Step Execution 或 Job Execution 中保存需要进行持久化的状态。

Execution Context 分为两类：一类是 Job Execution 的上下文（对应表：BATCH_JOB_EXECUTION_CONTEXT）；另一类是 Step Execution 的上下文（对应表：BATCH_STEP_EXECUTION_CONTEXT）。两类上下文之间的关系：一个 Job Execution 对应一个 Job Execution 的上下文；每个 Step Execution 对应一个 Step Execution 上下文；同一个 Job 中的 Step Execution 公用 Job Execution 的上下文。

因此如果同一个 Job 的不同 Step 间需要共享数据，则可以通过 Job Execution 的上下文来共享数据。利用 Execution Context 中的 key/value 对可以重新启动 Job；也可以利用 Execution Context 在不同的作业步 Step 之间进行数据功能。

Spring Batch 中可以通过 tasklet、reader、write、processor、listener 中访问 Execution Context 对象，在不同的 Step 中可以将数据写入 Context 或者从 Context 读取。

Job 上下文、Step 上下文之间的关系图参见图 9-28。

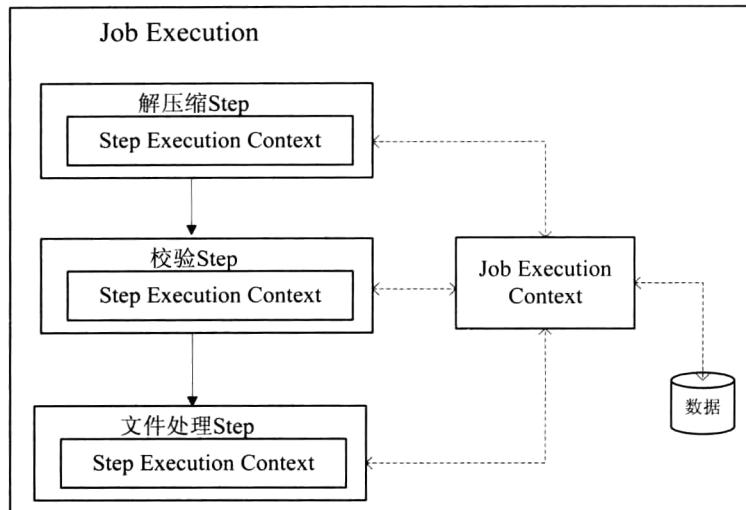


图 9-28 Job 上下文、Step 上下文之间关系

Job Execution Context 在整个 Job 的执行期间存在，不同的 Step 可以将数据存入 Job 上下文中，Job Execution Context 在执行期间会将数据保存到 DB 中，在 Job 重启的时候能够恢复 Job 的状态。

在 Execution Context 中写入、读取数据的典型用法参见代码清单 9-16。

代码清单 9-16 ExecutionContext 中写入、读取数据的典型用法

```
1. executionContext.putString(Constant.READ_FILE, workDirectory + readFileName);
2. String readfile = executionContext.getString(Constant.VERIFY_STATUS);
```

其中，1 行：将数据放入执行上下文中。

2 行：从数据上下文中获取数据。

接下来的示例展示如何在 Tasklet 中将共享数据放入 Job Execution Context 中；然后使用 StepExecutionListener 从 Execution Context 中访问数据。

VerifyTasklet 是校验作业步的实现，参见代码清单 9-17，在该 Step 中将校验的信息存入 Job 上下文。

完整代码参见：com.juxtapose.example.ch09.tasklet.VerifyTasklet。

代码清单 9-17 VerifyTasklet 类实现

```
1. public class VerifyTasklet implements Tasklet {
2.     @Override
```

```
3.     public RepeatStatus execute(StepContribution contribution,
4.         ChunkContext chunkContext) throws Exception {
5.         String status = creditService.verify(outputDirectory, readFileName);
6.         if (null != status) {
7.             chunkContext.getStepContext().getStepExecution()
8.                 .getExecutionContext().put(Constant.VERITY_STATUS, status);
9.         }
10.        return RepeatStatus.FINISHED;
11.    }
12.    .....
13. }
```

其中，7~8 行：将校验的状态存入 Job ExecutionContext 中，方便后面的拦截器能够获取校验的状态，根据校验状态设置 Step 的退出值。

VerifyStepExecutionListener 拦截器从 Job Execution Context 中获取校验信息。VerifyStepExecutionListener 的类实现参见代码清单 9-18。

完整代码参见：com.juxtapose.example.ch09.listener.VerifyStepExecutionListener。

代码清单 9-18 VerifyStepExecutionListener 类实现

```
1. public class VerifyStepExecutionListener implements StepExecutionListener {
2.     @Override
3.     public void beforeStep(StepExecution stepExecution) { }
4.
5.     @Override
6.     public ExitStatus afterStep(StepExecution stepExecution) {
7.         String status =
8.             stepExecution.getExecutionContext().getString(Constant.
9.                 VERITY_STATUS);
10.        if(null != status){
11.            return new ExitStatus(status);
12.        }
13.        return stepExecution.getExitStatus();
14.    }
15. }
```

其中，7~8 行：从 Job ExecutionContext 中获取校验 Step 中的校验结果信息。

9.6 终止 Job

Spring Batch 框架中支持在 Job 的某一个作业步终止作业。截止到目前，前面所有的 Job 均是在 Job 的最后一个 Step 完成 Job 的执行。本节我们展示如何使用 end、stop、fail 元素来根据 ExitStatus 完成 Job 的终止。

end 用来根据 ExitStatus 来正确的完成 Job，使用 end 结束后的 Job 的 BatchStatus 是 COMPLETED，不能重新启动。

stop 用来根据 ExitStatus 来停止 Job，使用 stop 结束后的 Job 的 BatchStatus 是 STOPPED，可以重新启动。

fail 用来根据 ExitStatus 来让 Job 失败，使用 fail 结束后的 Job 的 BatchStatus 是 FAILED，可以重新启动。

终止 end、stop、fail 三者比较参见表 9-8。

表 9-8 终止 end、stop、fail 三者区别

元素	Batch Status	Exit Status	Exit Status 可否更改	可否重启	说 明
end	COMPLETED	COMPLETED	是	否	完成当前的 Step 后，完成当前 Job
stop	STOPPED	STOPPED	否	是	完成当前 Step 后，停止当前 Job
fail	FAILED	FAILED	是	是	完成当前的 Step 后，Job 失败

9.6.1 end

使用 end 元素，可以根据给定的退出状态将 Job 正常的终止掉；通常可以根据业务状态的值将 Job 终止，默认情况下 Job 的退出状态为 COMPLETED，批处理状态为 COMPLETED；可以根据属性 exit-code 来指定 Job 的批处理退出状态值。

元素 end 的 schema 的定义参见图 9-29。

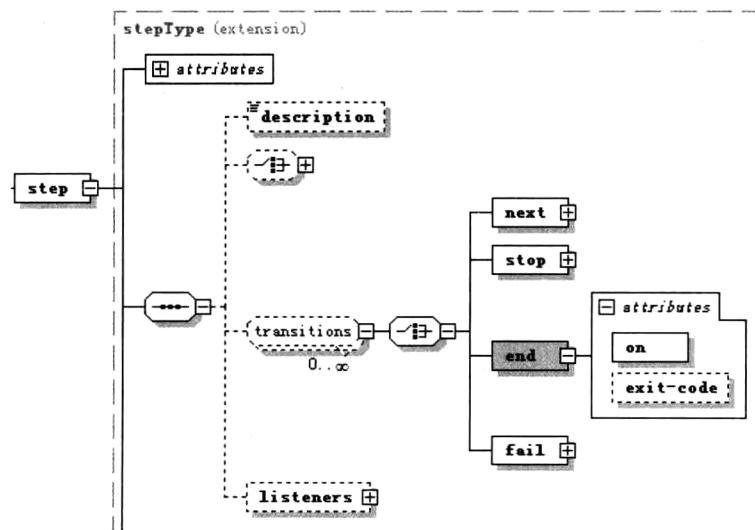


图 9-29 元素 end 的 schema 定义

end 元素的属性说明参见表 9-9。

表 9-9 end 元素的属性说明

属 性	说 明	默 认 值
on	定义作业步的 ExitStatus（退出状态）和 on 属性指定的值匹配的时候，则正常的完成 Job。 on 属性的值可以是任意的字符串，同时支持通配符“*”、“?” “*”：表示退出状态为任何值都满足。 “？”：表示匹配一个字符，如 c?t，当作业的退出状态为 cat 的时候满足，如果是 cabt 则不满足。	
exit-code	设置 Job 的退出状态	默认值为： COMPLETED

假设在信用卡账单处理过程中对文件进行校验的作业步进行特殊的处理，如果校验的 ExitStatus 为“FAILED”，则直接使用 end 完成当前的 Job。

代码清单 9-19 展示了如何使用 end 元素终止 Job。

完整配置参见文件：ch09/job/job-conditional-end.xml。

代码清单 9-19 Job 中使用 end 元素

```

1.  <job id="conditionalEndJob" >
2.      <step id="decompressStep" parent="abstractDecompressStep"
   next="verifyStep" >
3.          <tasklet ref="decompressTasklet" />
4.      </step>
5.      <step id="verifyStep" >
6.          <tasklet ref="verifyTasklet" />
7.          <end on="FAILED" exit-code="endExitCode"/>
8.          <next on="SKIPTOCLEAN" to="cleanStep" />
9.          <next on="*" to="readWriteStep" />
10.         <listeners>
11.             <listener ref="verifyStepExecutionListener" />
12.         </listeners>
13.     </step>
14.     <step id="readWriteStep" parent="parentReadWriteStep"
   next="cleanStep" />
15.     <step id="cleanStep">
16.         <tasklet ref="cleanTasklet" />
17.     </step>
18. </job>

```

其中，7 行：使用 end 元素终止 Job；属性 on 表示当作业步 verifyStep 的退出状态为“FAILED”的时候将会终止 Job 的执行；属性 exit-code 可以指定 Job 的退出状态。

执行 Job 后，可以查看作业执行器的信息，具体参见图 9-30。Job 的 BatchStatus 为“COMPLETED”，退出状态 EXIT_CODE 为“endExitCode”。

JOB_EXECUTION_ID	VERSION	STATUS	EXIT_CODE	JOB_INSTANCE_ID
1	168	2 COMPLETED	endExitCode	166

图 9-30 作业执行器的信息

9.6.2 stop

使用 stop 元素，可以根据给定的退出状态将 Job 停止掉；通常可以根据业务状态的值将 Job 停止，默认情况下 Job 的退出状态为 STOPPED，批处理状态也为 STOPPED；可以根据属性 restart 来指定 Job 重启时候从哪个 Step 开始执行。

元素 stop 的 schema 的定义参见图 9-31。

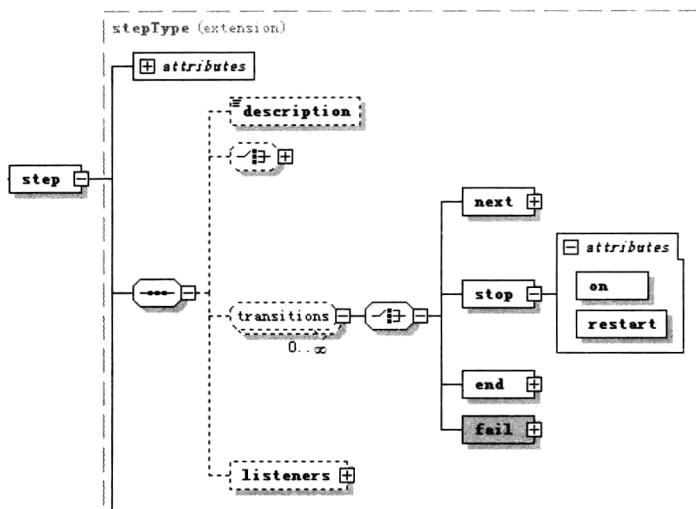


图 9-31 元素 stop 的 schema 定义

stop 元素的属性说明参见表 9-10。

表 9-10 stop 元素的属性说明

属性	说 明	默 认 值
on	定义作业步的 ExitStatus（退出状态）和 on 属性指定的值匹配的时候，停止当前的 Job。 on 属性的值可以是任意的字符串，同时支持通配符“*”、“?” “*”：表示退出状态为任何值都满足。 “？”：表示匹配一个字符，如 c?t，当作业的退出状态为 cat 的时候满足，如果是 cabt 则不满足。	
restart	指定 Job 重启的时候，从哪个 Step 开始执行	

假设在信用卡账单处理过程中对文件进行校验的作业步进行特殊的处理，如果校验的

ExitStatus 为"FAILED"，则直接使用 stop 停止当前的 Job，并使用属性 restart 来制定重启时候需要执行的 Step。

代码清单 9-20 展示了如何使用 stop 元素停止 Job。

完整配置参见文件：ch09/job/job-conditional-stop.xml。

代码清单 9-20 使用 stop 元素停止 Job

```
1.  <job id="conditionalStopJob" >
2.      <step id="decompressStep" parent="abstractDecompressStep"
3.          next="verifyStep" >
4.              <tasklet ref="decompressTasklet" />
5.          </step>
6.          <step id="verifyStep" >
7.              <tasklet ref="verifyTasklet" />
8.              <stop on="COMPLETED" restart="readWriteStep"/>
9.              <next on="SKIPTOCLEAN" to="cleanStep" />
10.             <next on="*" to="readWriteStep" />
11.             <listeners>
12.                 <listener ref="verifyStepExecutionListener" />
13.             </listeners>
14.         </step>
15.         <step id="readWriteStep" parent="parentReadWriteStep"
16.             next="cleanStep" />
17.             <step id="cleanStep">
18.                 <tasklet ref="cleanTasklet" />
19.             </step>
20.         </job>
```

其中，7 行：属性 on 定义满足的 ExitStatus 为" COMPLETED "的时候停止 Job；属性 restart 定义 Job 重启时候执行的 Step。

执行 Job 后，可以查看作业执行器的信息，具体参见图 9-32。Job 的 BatchStatus 为“STOPPED”，退出状态 EXIT_CODE 为“STOPPED”。

	JOB_EXECUTION_ID	VERSION	STATUS	EXIT_CODE	JOB_INSTANCE_ID
1	169	2	STOPPED	STOPPED	167

图 9-32 作业执行器的信息

9.6.3 fail

使用 fail 元素，可以根据给定的退出状态将 Job 正确地终止掉；通常可以根据业务状态的值将 Job 终止，默认情况下 Job 的退出状态为 FAILED，批处理状态为 FAILED；可以根据属性 exit-code 来指定 Job 的批处理退出状态值。

元素 fail 的 schema 的定义参见图 9-33。

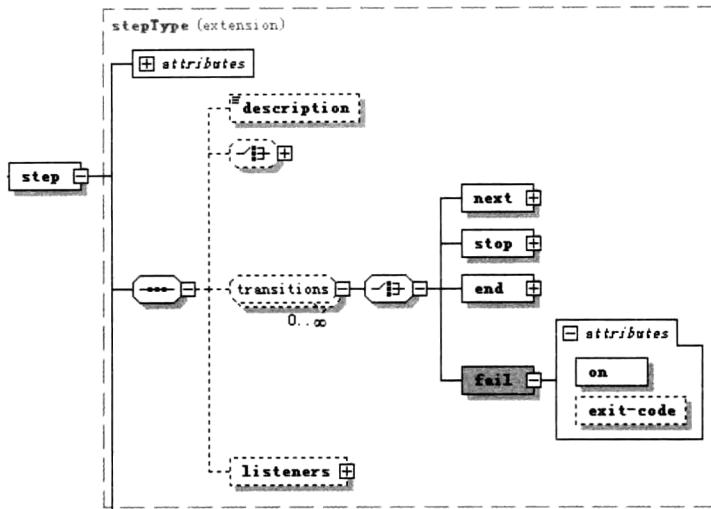


图 9-33 元素 fail 的 schema 的定义

fail 元素的属性说明参见表 9-11。

表 9-11 fail 元素的属性说明

属性	说 明	默认值
on	定义作业步的 ExitStatus（退出状态）和 on 属性指定的值匹配的时候，以失败的方式完成 Job。 on 属性的值可以是任意的字符串，同时支持通配符 "*"、"?" "*": 表示退出状态为任何值都满足。 "?": 表示匹配一个字符，如 c?t，当作业的退出状态为 cat 的时候满足，如果是 cabt 则不满足	
exit-code	设置 Job 的退出状态	默认值为： FAILED

假设在信用卡账单处理过程中对文件进行校验的作业步进行特殊的处理，如果校验的 ExitStatus 为"FAILED"，则直接使用 fail 终止当前的 Job。

代码清单 9-21 展示了如何使用 fail 元素终止 Job。

完整配置参见文件：ch09/job/job-conditional-fail.xml。

代码清单 9-21 使用 fail 元素终止 Job

```

1. <job id="conditionalFailJob" >
2.   <step id="decompressStep" parent="abstractDecompressStep"
      next="verifyStep" >
3.     <tasklet ref="decompressTasklet" />
4.   </step>

```

```
5.          <step id="verifyStep" >
6.              <tasklet ref="verifyTasklet" />
7.              <fail on="FAILED" exit-code="EARLY TERMINATION"/>
8.              <next on="SKIPTOCLEAN" to="cleanStep" />
9.              <next on="*" to="readWriteStep" />
10.             <listeners>
11.                 <listener ref="verifyStepExecutionListener" />
12.             </listeners>
13.         </step>
14.         <step id="readWriteStep" parent="parentReadWriteStep"
15.             next="cleanStep" />
16.         <step id="cleanStep">
17.             <tasklet ref="cleanTasklet" />
18.         </step>
19.     </job>
```

其中，7行：属性 on 定义满足的 ExitStatus 为" FAILED "的时候停止 Job；属性 exit-code 定义 Job 终止时候的退出状态为“EARLY TERMINATION”。

执行 Job 后，可以查看作业执行器的信息，具体参见图 9-34。Job 的 BatchStatus 为“FAILED”，退出状态 EXIT_CODE 为“EARLY TERMINATION”。

#	JOB_EXECUTION_ID	VERSION	STATUS	EXIT_CODE	JOB_INSTANCE_ID
1	174	2	FAILED	EARLY TERMINATION	172

图 9-34 作业执行器的信息

健壮 Job

批处理要求 Job 必须有较强的健壮性，通常 Job 是批量处理数据、无人值守的，这要求在 Job 执行期间能够应对各种发生的异常、错误，并对 Job 执行进行有效的跟踪。一个健壮的 Job 通常需要具备如下的几个特性。

(1) 容错性

在 Job 执行期间非致命的异常，Job 执行框架应能够进行有效的容错处理，而不是让整个 Job 执行失败；通常只有致命的、导致业务不正确的异常才可以终止 Job 的执行。

(2) 可追踪性

Job 执行期间任何发生错误的地方都需要进行有效的记录，方便后期对错误点进行有效的处理。例如在 Job 执行期间任何被忽略处理的记录行需要被有效地记录下来，应用程序维护人员可以针对被忽略的记录后续做有效的处理。

(3) 可重启性

Job 执行期间如果因为异常导致失败，应该能够在失败的点重新启动 Job；而不是从头开始重新执行 Job。

Spring Batch 框架提供了支持上面所有能力的特性，包括 Skip（跳过记录处理）、Retry（重试给定的操作）、Restart（从错误点开始重新启动失败的 Job），具体描述参见表 10-1。

表 10-1 健壮 Job 特性

特 性	功 能	适 用 时 机	适 用 场 景
Skip	跳过错误的记录行，保证 Job 能够正确地执行	适用于非致命的异常	面向 Chunk 的 Step
Retry	重试给定的操作，比如短暂的网络异常、并发异常等	适用于短暂的异常，经过重试之后该异常可能会不再重现	面向 Chunk 的 Step 或者应用代码
Restart	Job 执行失败后，重新启动 Job 实例	因异常、错误导致 Job 失败后	Job 执行重新启动

Skip，在对 Flat 类型的文件处理期间，如果文件中某行的格式不能满足要求，可以通过 Skip 跳过该行记录的处理，让 Processor 能够顺利地处理其余的记录行。

Retry，将给定的操作进行多次重试，在某些情况下操作因为短暂的异常导致执行失败，如网络连接异常、并发处理异常等，可以通过重试的方式避免单次的失败，下次执行操作的时候网络恢复正常，不再有并发的异常，这样通过重试的能力可以有效地避免这类短暂的异常。

Restart，在 Job 执行失败后，可以通过重启功能来继续完成 Job 的执行。在重启时候，

批处理框架允许在上次执行失败的点重新启动 Job，而不是从头开始执行，这样可以大幅提高 Job 执行的效率。

10.1 跳过 Skip

Step 执行期间 read、process、write 发生的任何异常都会导致 Step 执行失败，进而导致作业的失败。批处理作业的自动化、定时触发，有特定的执行时间窗口特性，决定了尽可能地减少 Job 的失败。设想信用卡对账单的处理的业务场景，银行每天需要处理海量的对账文件，如果对账文件中有少量的一行或者几行错误格式的记录，在真正进行作业处理的时候，不希望因为几行错误的记录而导致整个作业的失败；而是希望将这几行没有处理的记录跳过去，让整个 Job 正确执行，对于错误的记录则通过日志的方式记录下来后续进行单独的处理。

Spring Batch 框架通过属性 skip-limit、skippable-exception-classes、skip-policy 来完成异常跳过的能力，具体属性参见表 10-2。

表 10-2 Skip 属性描述

属性/元素	功能说明
skippable-exception-classes	定义允许跳过的异常，碰到该类型异常时候，不会导致 Job 失败；而是跳过当前记录的处理，保证 Job 继续正确的执行
include	skippable-exception-classes 的子元素，用以表示包括在内的异常
exclude	skippable-exception-classes 的子元素，用以表示排除在内的异常，通常用来定义某一类型的子异常
skip-limit	跳过限制次数，当超过该次数后再发生异常会导致 Job 失败
skip-policy	Job 的跳过策略，根据该策略判断是否允许跳过异常

skippable-exception-classes: 定义记录跳过的异常，可以定义一组异常，如果发生了定义的异常或者子类异常都不会导致作业失败。

skip-limmit: 任务处理发生异常时，允许跳过的最大次数。

skip-policy: 当默认的按照次数跳过策略不能满足需求时，可以配置自定义跳过策略，需要实现接口：org.springframework.batch.core.step.skip.SkipPolicy。

10.1.1 配置 Skip

何种类型的异常能够跳过处理，通常由开发人员根据具体的业务场景确定，比如单条记录不完整，或者完整但是格式不正确导致无法正确解析。可以使用属性 skippable-exception-classes 来定义当发生那些类型的异常时，避免 Job 失败。

代码清单 10-1 给出了对账单格式错误文件示例。

代码清单 10-1 错误格式对账单示例

```
1. 4047390012345678,tom,100.00,xxx-yyy-zzz,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.20,2013-2-9 15:15:37
5. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road
```

其中，1行：交易日期格式有误。

4行：记录不完整，缺少交易的地址列信息。

如果记录行数错误较多，即使成功执行完毕 Job，也没有任何意义，因为错误记录数太多导致后续的手工修复可能更复杂。此时最好的处理方式是让 Job 执行失败，然后修复记录文件重新执行 Job 作业。Spring 框架提供了最大跳过记录数的属性 skip-limmit，当跳过的记录数大于设定的值后，Job 作业将会失败。

配置 Skip

配置 Skip 的示例参见代码清单 10-2。设置允许跳过的最大记录为 20 条，在发生 java.lang.RuntimeException 但不包括 java.io.FileNotFoundException 的时候跳过处理。

完整配置参见文件：/ch10/job/job-step-skip.xml。

代码清单 10-2 配置 Skip

```
1. <job id="skipJob">
2.   <step id="skipStep">
3.     <tasklet>
4.       <chunk reader="reader" processor="processor" writer="writer"
5.             commit-interval="1" skip-limit="20">
6.         <skippable-exception-classes>
7.           <include class="java.lang.RuntimeException" />
8.           <exclude class="java.io.FileNotFoundException" />
9.         </skippable-exception-classes>
10.        </chunk>
11.      </tasklet>
12.    </step>
13.  </job>
14.
15.  <bean:bean id="processor"
16.    class="com.juxtapose.example.ch10.skip.
      RadomExceptionItemProcessor" />
```

其中，5行：异常发生时，允许跳过最大的异常次数为 20 次，如果超过 20 次后，再次发生的异常会导致 Job 的失败。

6~9 行：include 定义支持跳过的异常，exclude 定义排出在外的异常；当发生了任何 RuntimeException 类型异常（同时 FileNotFoundException 排出在外）时会跳过处理，但是当

已经跳过 20 次后，第 21 次发生异常的时候，会导致 Job 的失败，因为超过了 skip-limit="20" 的限制。

15~16 行：自定义一个 ItemProcessor 实现类 RadomExceptionItemProcessor，该类会随机发生 RuntimeException 类型异常。

com.juxtapose.example.ch10.skip.RadomExceptionItemProcessor 实现代码参见代码清单 10-3。

代码清单 10-3 RadomExceptionItemProcessor 类实现

```
1. public class RadomExceptionItemProcessor implements
   ItemProcessor<String, String> {
2.     Random ra = new Random();
3.
4.     public String process(String item) throws Exception {
5.         int i = ra.nextInt(10);
6.         System.out.println("Process " + item + "; Random i=" + i);
7.         if(i%2 == 0){
8.             throw new RuntimeException("make error!");
9.         }else{
10.             return item;
11.         }
12.     }
13. }
```

其中，5 行：定义随机数。

7~10 行：根据对 2 取模，抛出异常或者正确返回记录。

说明：发生的异常可能来自 Read、Process、Write 执行阶段，即 Read、Process、Write 任何操作发生的异常都可能导致 Skip 发生。

10.1.2 跳过策略 SkipPolicy

根据 skip-limit、skippable-exception-classes 可以配置简单的跳过逻辑处理，通常情况下可能需要复杂的跳过逻辑配置，Spring Batch 框架提供了友好的扩展策略，通过实现接口 SkipPolicy 可以自定义跳过策略。

skip-limit 设置限制跳过次数，默认使用的跳过策略为 LimitCheckingItemSkipPolicy。

接口 org.springframework.batch.core.step.skip.SkipPolicy 定义参见代码清单 10-4。

代码清单 10-4 SkipPolicy 接口定义

```
1. public interface SkipPolicy {
2.     boolean shouldSkip(Throwable t, int skipCount) throws SkipLimitExceeded
   Exception;
3. }
```

其中，2 行：接口核心方法，根据给定的异常和已经跳过的记录数，判断当前发生的异

常是否被跳过处理； t 表示发生的异常， $skipCount$ 表示已经跳过的记录数。

系统提供的默认 SkipPolicy 实现列表参见表 10-3。

表 10-3 SkipPolicy 默认实现

SkipPolicy 默认实现	功能说明
AlwaysSkipItemSkipPolicy	发生任何异常都会导致跳过记录处理 org.springframework.batch.core.step.skip.AlwaysSkipItemSkipPolicy
CompositeSkipPolicy	组合跳过策略，可以将多个跳过策略组合在一起使用，按照顺序判断是否应该跳过该记录；多个组合策略中只要有一个允许跳过，则组合策略允许跳过该记录的处理 org.springframework.batch.core.step.skip.CompositeSkipPolicy
ExceptionClassifierSkipPolicy	为不同的异常定义不同的跳过策略 org.springframework.batch.core.step.skip.ExceptionClassifierSkipPolicy
LimitCheckingItemSkipPolicy	根据设置的次数决定异常是否能被跳过处理 org.springframework.batch.core.step.skip.LimitCheckingItemSkipPolicy
NeverSkipItemSkipPolicy	发生任何异常都不会被跳过 org.springframework.batch.core.step.skip.NeverSkipItemSkipPolicy

接下来，以 AlwaysSkipItemSkipPolicy 为例展示如何使用 SkipPolicy，详细参见代码清单 10-5。

完整配置文件参见：/ch10/job/job-step-skip.xml。

代码清单 10-5 配置 SkipPolicy

```
1.      <job id="skipPolicyJob">
2.          <step id="skipPolicyStep">
3.              <tasklet>
4.                  <chunk reader="reader" processor="processor" writer="writer"
5.                      commit-interval="2" skip-policy="alwaysSkipPolicy">
6.                  </chunk>
7.              </tasklet>
8.          </step>
9.      </job>
10.
11.      <bean:bean id="alwaysSkipPolicy"
12.          class="org.springframework.batch.core.step.skip.
AlwaysSkipItemSkipPolicy"/>
```

其中，5 行：使用 alwaysSkipPolicy 对象作为跳过处理策略。

11~12：定义 alwaysSkipPolicy，默认实现为 AlwaysSkipItemSkipPolicy，表示发生的任何异常都不会中断 Step、Job 的处理。

执行 skipPolicyJob 作业期间，任何发生的异常都会被忽略掉，保证 Job 的正确执行，但

是对于跳过的记录需要通过日志或者其他的方式记录下来，方便后期的对账处理或者人工干预。在 10.1.3 章节我们将展示如果通过拦截器记录跳过的处理。

10.1.3 跳过拦截器

SkipListener 在 Chunk 处理阶段抛出跳过定义的异常时候触发，在 Chunk 的读、处理、写阶段发生的异常都会触发该拦截器。接口 SkipListener 定义参见代码清单 10-6。

代码清单 10-6 SkipListener 接口定义

```
1. public interface SkipListener<T,S> extends StepListener {  
2.     void onSkipInRead(Throwable t);  
3.     void onSkipInWrite(S item, Throwable t);  
4.     void onSkipInProcess(T item, Throwable t);  
5. }
```

SkipListener 操作说明与 Annotation 定义参见表 10-4。

表 10-4 SkipListener 操作说明与 Annotation

操作	操作说明	Annotation
onSkipInRead(Throwable t)	在读阶段发生异常并且配置了异常可以跳过时候触发该操作	@ OnSkipInRead
onSkipInWrite(S item, Throwable t)	在写阶段发生异常并且配置了异常可以跳过时候触发该操作	@ OnSkipInWrite
onSkipInProcess(T item, Throwable t)	在处理阶段发生异常并且配置了异常可以跳过时候触发该操作	@ OnSkipInProcess

SkipListener 系统实现参见表 10-5。

表 10-5 SkipListener 默认实现

SkipListener 默认实现	功能说明
CompositeSkipListener	组合跳过拦截器实现，可以定义一组跳过拦截器，依照顺序执行 org.springframework.batch.core.listener.CompositeSkipListener<T, S>
MulticasterBatchListener	同时实现 StepExecutionListener, ChunkListener, ItemReadListener<T>, ItemProcessListener<T, S>, ItemWriteListener<S>, SkipListener<T, S>六个接口组合策略的拦截器 org.springframework.batch.core.listener.MulticasterBatchListener<T, S>
SkipListenerSupport	跳过拦截器的默认执行，可以继承该类仅实现关心的方法 org.springframework.batch.core.listener.SkipListenerSupport<T, S>

跳过拦截器执行时机

Skip 拦截器并非在读、处理、写阶段发生异常后立刻执行；而是在批操作事务提交正确

之前才执行。下面我们分析一下 Spring Batch 框架如此实现的原因：Spring Batch 框架是面向批的操作，每一批的处理被隐性地封装在同一个事务中，当发生跳过的异常时候，在批后面的操作过程中可能出现其他类型的错误导致发生回滚，即作业失败，在这种场景下我们不希望 Skip 拦截器在前面已经执行；如果 Skip 拦截器已经执行了导致的结果是，整个作业失败但有部分跳过的记录做了 Skip 拦截器的动作，导致状态不一致。因此 Skip 拦截器在 Chunk 能正确提交事务之前被触发，而不是读、处理、写阶段发生异常的时候就触发。

说明：跳过记录的处理，并不仅仅在读阶段发生跳过异常，在处理、写阶段发生跳过异常时候同样会触发 Skip 拦截器。

Read 阶段发生跳过异常：此时 Spring Batch 框架继续调用 read 操作获取下一个条目。

Process 阶段发生跳过异常：Spring Batch 框架进行回滚当前批操作，并重新提交读到的数据（发生异常的那条数据除外）。

Write 阶段发生跳过异常：因为写数据是批量提交的，不知道哪条记录发生了跳过异常，Spring Batch 框架将为每一条数据启用一个单独的事务进行数据的重新处理。写阶段发生跳过异常的处理参见图 10-1。

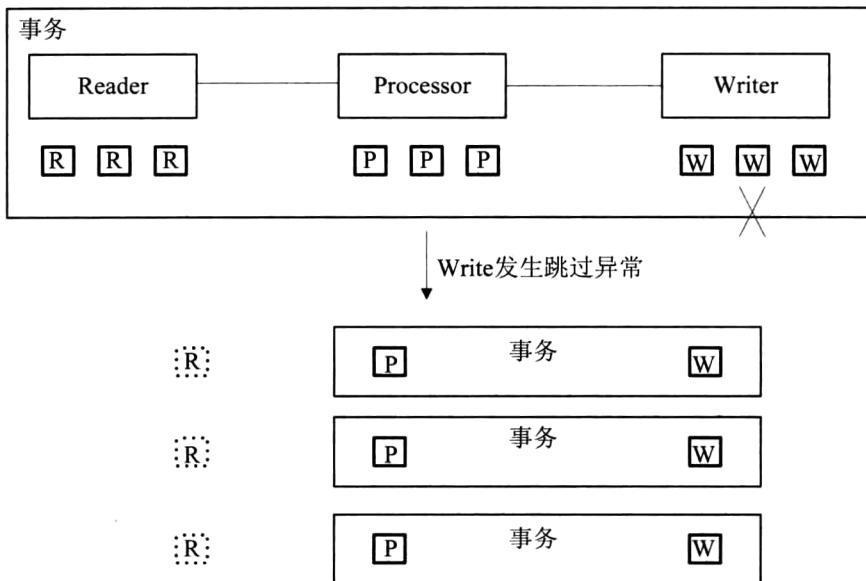


图 10-1 写阶段发生跳过异常的处理

在实际批处理作业中，对于跳过的记录需要记录下来，在 Job 完成后需要进行后续的跟踪处理或者人工干预。在信用卡对帐单作业处理工程中需要将跳过的记录存放在数据库中。

实现 SkipListener 记录跳过的记录存放在数据库。

com.juxtapose.example.ch10.skip.DBSkipListener 类定义参见代码清单 10-7。

代码清单 10-7 DBSkipListener 类实现

```
1. public class DBSkipListener implements SkipListener<String, String> {
2.     private JdbcTemplate jdbcTemplate;
3.
4.     public void onSkipInRead(Throwable t) {
5.         if (t instanceof FlatFileParseException) {
6.             jdbcTemplate.update("insert into skipbills "
7.                 + "(line,content) values (?,?)",
8.                 ((FlatFileParseException) t).getLineNumber(),
9.                 ((FlatFileParseException) t).getInput());
10.        }
11.    }
12.
13.    public void onSkipInWrite(String item, Throwable t) {}
14.    public void onSkipInProcess(String item, Throwable t) {}
15.    .....
16. }
```

其中，6~10 行：将读记录跳过的行信息记录数据库。

13~14 行：对于写和处理阶段的跳过操作忽略处理。

跳过拦截器声明配置参见代码清单 10-8。

完整配置文件参见：/ch10/job/job-step-skip-listener.xml。

代码清单 10-8 配置跳过拦截器

```
1. <job id="dbSkipJob">
2.     <step id="dbSkipStep">
3.         <tasklet>
4.             <chunk reader="randomExceptionAutoReader" processor="processor"
5.                 writer="writer" commit-interval="5" skip-limit="20">
6.                 <skippable-exception-classes>
7.                     <include class="org.springframework.batch.item.file.
8.                         FlatFileParseException" />
9.                 </skippable-exception-classes>
10.                <listeners>
11.                    <listener ref="dbSkipListener"></listener>
12.                </listeners>
13.            </chunk>
14.        </tasklet>
15.    </step>
16. </job>
17. <bean:bean id="randomExceptionAutoReader"
18.             class="com.juxtapose.example.ch10.
19.             RandomExceptionAutoReader" />
20. <bean:bean id="dbSkipListener"
```

```
20.          class="com.juxtapose.example.ch10.step.listener.  
           DBSkipListener" >  
21.      <bean:property name="jdbcTemplate" ref="jdbcTemplate">  
        </bean:property>  
22.    </bean:bean>
```

其中，7~8 行：定义允许跳过的异常为 `org.springframework.batch.item.file.FlatFileParseException`。

11 行：定义跳过拦截器为 `dbSkipListener`，负责将跳过的行记录到数据库中。

17~18 行：定义读 `radomExceptionAutoReader`，随机产生类型 `FlatFileParseException` 的异常抛出。

19~22 行：声明跳过拦截器，使用 Spring 中的 `jdbcTemplate` 负责将记录持久化到数据库中。

持久化信息配置声明参见代码清单 10-9。

完整配置文件参见：/ch10/job-context-db.xml。

代码清单 10-9 配置持久化信息

```
1.   <jdbc:initialize-database data-source="dataSource">  
2.     <jdbc:script location="classpath:ch10/db/create-table-skipbills.  
      sql" />  
3.   </jdbc:initialize-database>  
4.  
5.   <bean:bean id="jdbcTemplate" class="org.springframework.jdbc.core.  
      JdbcTemplate">  
6.     <bean:constructor-arg ref="dataSource" />  
7.   </bean:bean>
```

其中，3 行：初始化数据库文件 `create-table-skipbills.sql`，负责将拦截器用到的表 `skipbills` 初始化。

5~7 行：定义 Spring 的 `jdbc` 模板，提供执行 `sql` 语句的基本功能。

数据库表定义具体参见文件：/ch10/db/create-table-skipbills.sql。

10.2 重试 Retry

Step 执行期间 `read`、`process`、`write` 发生的任何异常都会导致 Step 执行失败，进而导致作业的失败。批处理作业的自动化、定时触发，有特定的执行时间窗口特性，决定了尽可能地减少 Job 的失败。处理任务阶段发生的异常可以让业务失败，也可以通过 `Skip` 的设置，跳过部分异常；但是另外还有部分异常，例如并发对数据库的操作导致的数据库锁的异常（`DeadlockLoserDataAccessException`）、网络不稳定导致的网络连接异常（`java.net.ConnectException`）。这类异常的出现可能在下次重新操作的时候消失，数据库锁的异常在下次操作可能正确恢复，网络不能连接的异常可能在重试几次后恢复正常。因此，这些异常出现的时候，不期望作业

发生异常，而是希望通过几次重试操作，尽可能让 Job 成功执行。

Spring Batch 框架提供了任务重试功能，重试次数限制功能、自定义重试策略以及重试拦截器能力。分别通过属性 `retryable-exception-classes`、`retry-limit`、`retry-policy`、`cache-capacity`、`retry-listeners` 来实现。具体属性描述参见表 10-6。

表 10-6 Retry 属性描述

属性/元素	功能说明
<code>retryable-exception-classes</code>	定义允许重试的异常，碰到该类型异常时候，不会导致 Job 失败；而是重试当前的操作，保证 Job 继续正确地执行
<code>include</code>	<code>retryable-exception-classes</code> 的子元素，用以表示包括在内的异常
<code>exclude</code>	<code>retryable-exception-classes</code> 的子元素，用以表示排除在内的异常，通常用来定义某一类型的子异常
<code>retry-limit</code>	任务最大重试次数，当超过该次数后再发生异常会导致 Job 失败
<code>retry-policy</code>	Job 的重试策略，根据该策略判断是否允许重试该次失败的操作
<code>cache-capacity</code>	存放 <code>RetryContext</code> 的缓存大小，当超过该值时，会发生异常
<code>retry-listeners</code>	定义重试拦截器

`retryable-exception-classes`: 定义可以重试的异常，可以定义一组异常，如果发生了定义的异常或者子类异常都会导致重试。

`retry-limit`: 任务执行重试的最大次数。

`skip-policy`: 定义自定义的重试策略，需要实现接口：`org.springframework.batch.retry.RetryPolicy`。

`cache-capacity`: `retry-policy` 缓存的大小，缓存用于存放重试上下文 `RetryContext`，如果超过配置最大值，会发生异常：`org.springframework.batch.retry.policy.RetryCacheCapacityExceeded Exception`。

`retry-listeners`: 配置重试监听器，监听器需要实现接口：`org.springframework.batch.retry.RetryListener`。

10.2.1 配置 Retry

为了展示重试功能，在 Job 的处理阶段模拟发生异常，为了避免无限次地重试该操作，设置限制重试次数为 3 次。

配置 Retry 的示例参见代码清单 10-10。

完整配置文件参见：[/ch10/job/job-step-retry.xml](#)。

代码清单 10-10 配置 Retry

```
1.      <job id="retryJob">
2.          <step id="retryStep">
```

```

3.          <tasklet>
4.              <chunk reader="reader" processor="alwaysExceptionItemProcessor"
5.                  writer="writer" commit-interval="1" retry-limit="3">
6.                  <retry-listeners>
7.                      <listener ref="sysoutRetryListener"></listener>
8.                  </retry-listeners>
9.                  <retryable-exception-classes>
10.                      <include class="java.lang.RuntimeException" />
11.                      <exclude class="java.io.FileNotFoundException" />
12.                  </retryable-exception-classes>
13.              </chunk>
14.          </tasklet>
15.      </step>
16.  </job>
17.  <bean:bean id="sysoutRetryListener"
18.      class="com.juxtapose.example.ch10.retry.SystemOutRetryListener"/>
19.  <bean:bean id="alwaysExceptionItemProcessor"
20.      class="com.juxtapose.example.ch10.retry.
AlwaysExceptionItemProcessor" />
```

其中，5行：属性 retry-limit 定义最大重试次数为 3 次，当重试超过 3 次后发生的异常会导致 Step 失败。

6~8 行：定义重试操作的拦截器，在 chunk 发生重试操作时候，会触发拦截器 sysoutRetryListener 操作。

9~12 行：属性 retryable-exception-classes 定义重试的异常，可以定义多个异常，include 表示能够触发重试的异常，exclude 表示不会触发重试的异常。

17~18 行：定义重试拦截器 sysoutRetryListener，拦截器仅把信息打印在控制台上。

19~20 行：定义自定义处理器 alwaysExceptionItemProcessor，每次操作均会发生异常，用于模拟 chunk 发生异常。

com.juxtapose.example.ch10.retry.AlwaysExceptionItemProcessor 实现代码参见代码清单 10-11。

代码清单 10-11 AlwaysExceptionItemProcessor 类实现

```

1.  public class AlwaysExceptionItemProcessor implements ItemProcessor
2.      <String, String> {
3.          Random ra = new Random();
4.          public String process(String item) throws Exception {
5.              int i = ra.nextInt(10);
6.              if(i%2 == 0){
7.                  System.out.println("Process " + item + "; Random i=" + i +"; ...");
8.                  throw new MockARuntimeException("make error!");
9.              }else{
10.                  System.out.println("Process " + item + "; Random i=" + i +"; ...");
```

```
10.         throw new MockBRuntimeException("make error!");
11.     }
12. }
13. }
```

其中，4行：定义随机数。

5~10行：根据对2取模，抛出不同类型的异常。

10.2.2 重试策略 RetryPolicy

根据 retryable-exception-classes、retry-limit 可以配置简单的重试逻辑处理，通常情况下可能需要复杂的重试逻辑配置，Spring Batch 框架提供了友好的扩展策略，通过实现接口 `RetryPolicy` 可以自定义重试策略。

接口 `org.springframework.batch.retry.RetryPolicy` 定义参见代码清单 10-12。

代码清单 10-12 `RetryPolicy` 接口定义

```
1. public interface RetryPolicy {
2.     boolean canRetry(RetryContext context);
3.     RetryContext open(RetryContext parent);
4.     void close(RetryContext context);
5.     void registerThrowable(RetryContext context, Throwable throwable);
6. }
```

其中，`canRetry()`判断是否应该重试，`open()`在重试开始时执行，`close()`在重试结束时执行，`registerThrowable()`操作注册异常和重试上下文。

2行：接口核心方法 `canRetry()`，根据给定的异常和已经跳过的记录数，判断当前发生的异常是否被跳过处理；`t`表示发生的异常，`skipCount`表示已经跳过的记录数。

系统提供的默认 `RetryPolicy` 实现列表参见表 10-7。

表 10-7 `RetryPolicy` 系统默认实现

RetryPolicy 默认实现	功能说明
AlwaysRetryPolicy	发生任何异常都会导致重试操作 <code>org.springframework.batch.retry.policy.AlwaysRetryPolicy</code>
NeverRetryPolicy	发生任何异常都会导致重试操作 <code>org.springframework.batch.retry.policy.NeverRetryPolicy</code>
CompositeRetryPolicy	组合重试策略，可以将多个重试策略组合在一起使用，按照顺序判断是否应该重试操作；多个组合策略中只要有一个不允许重试，则组合策略不允许重试该操作 <code>org.springframework.batch.retry.policy.CompositeRetryPolicy</code>
ExceptionClassifierRetryPolicy	为不同的异常定义不同的重试策略 <code>org.springframework.batch.retry.policy.ExceptionClassifierRetryPolicy</code>

续表

RetryPolicy 默认实现	功能说明
SimpleRetryPolicy	根据设置的次数决定是否能进行重试 org.springframework.batch.retry.policy.SimpleRetryPolicy
TimeoutRetryPolicy	在给定的时间内可以进行重试，超过给定的时间将不会进行重试操作 org.springframework.batch.retry.policy.TimeoutRetryPolicy

接下来以 ExceptionClassifierRetryPolicy（为不同的异常定义不同的重试策略）为例展示如何使用 SkipPolicy，具体参见代码清单 10-13。

完整配置文件参见：/ch10/job/job-step-retry.xml。

代码清单 10-13 配置 retryPolicy

```

1.      <job id="retryPolicyJob">
2.          <step id="retryPolicyStep">
3.              <tasklet>
4.                  <chunk reader="reader"
5.                         processor="alwaysExceptionItemProcessor"
6.                         writer="writer" commit-interval="1"
7.                         retry-policy="exceptionClassifierRetryPolicy">
8.                  </chunk>
9.              </tasklet>
10.             </step>
11.         </job>

```

其中，6 行：属性 retry-policy 定义使用的跳过策略 exceptionClassifierRetryPolicy，该策略根据不同的异常可以配置不同的重试次数，具体参见代码清单 10-14。

配置重试策略 exceptionClassifierRetryPolicy。

代码清单 10-14 配置重试策略 exceptionClassifierRetryPolicy

```

1.  <bean:bean id="exceptionClassifierRetryPolicy"
2.      class="org.springframework.retry.policy.
   ExceptionClassifierRetryPolicy">
3.      <bean:property name="policyMap">
4.          <bean:map>
5.              <bean:entry key="com.juxtapose.example.ch10.retry.
   MockARuntimeException">
6.                  <bean:bean class="org.springframework.retry.policy.
   SimpleRetryPolicy">
7.                      <bean:property name="maxAttempts" value="3" />
8.                  </bean:bean>
9.              </bean:entry>
10.             <bean:entry key="com.juxtapose.example.ch10.retry.
   MockBRuntimeException">

```

```

12.          <bean:bean class="org.springframework.retry.policy.
13.                                SimpleRetryPolicy">
14.              <bean:property name="maxAttempts" value="5" />
15.          </bean:bean>
16.      </bean:entry>
17.  </bean:map>
18. </bean:property>
19. </bean:bean>

```

其中，5~10 行：定义异常 MockARuntimeException 的重试策略为 SimpleRetryPolicy，最大重试次数为 3 次。

11~16：定义异常 MockBRuntimeException 的重试策略为 SimpleRetryPolicy，最大重试次数为 5 次。

通过对 Chunk 的重试支持，在发生瞬态异常情况下通过重试操作保证 Job 执行的稳定性，尽可能避免 Job 作业失败的可能。如果 Step 的实现不是 Chunk 类型操作，而是自定义的 Tasklet 操作，可以使用 Spring Batch 框架提供的重试模板 RetryTemplate 实现重试功能，RetryTemplate 的具体使用方式 10.2.4 节会详细介绍。

10.2.3 重试拦截器

RetryListener 在 Chunk 处理阶段抛出重试定义的异常时候触发，通过拦截器方便在重试动作发生时候进行日志记录、收集重试信息等。可以直接实现接口 RetryListener（参见代码清单 10-15），也可以直接继承 RetryListenerSupport，通常只需要实现 onError 操作，在重试发生错误时触发该操作。

代码清单 10-15 RetryListener 接口定义

```

1. public interface RetryListener {
2.     <T> boolean open(RetryContext context, RetryCallback<T> callback);
3.     <T> void close(RetryContext context, RetryCallback<T> callback,
4.                     Throwable throwable);
5.     <T> void onError(RetryContext context, RetryCallback<T> callback,
6.                      Throwable throwable);
7. }

```

RetryListener 核心操作说明参见表 10-8。

表 10-8 RetryListener 操作说明

操作	操作说明	Annotation
open(RetryContext context, RetryCallback<T> callback)	在进入 retry 之前执行该操作，可以在该操作中准备重试需要的资源；如果该操作返回 false，将会终止本次重试操作，且会抛出异常：org.springframework.batch.retry.TerminatedRetryException	无

续表

操作	操作说明	Annotation
close(RetryContext context, RetryCallback<T> callback, Throwable throwable)	在 retry 结束之前执行该操作，可以在该方法中关闭在 open 操作中打开的资源	无
onError(RetryContext context, RetryCallback<T> callback, Throwable throwable)	重试发生错误时触发该操作	无

RetryListener 系统实现参见表 10-9。

表 10-9 RetryListener 系统默认实现

RetryListener 默认实现	功能说明
RetryListenerSupport	重试拦截器的默认执行，可以继承该类仅实现关心的方法 org.springframework.batch.retry.listener.RetryListenerSupport

可以使用系统提供的重试拦截器的支持类，简化重试拦截器的实现，仅需要实现业务关心的操作；Spring 整个框架提供了丰富的模板类和支持类来简化业务开发能力。

说明：RetryListener 的 open、close 操作在 Chunk 的 process、write 中即使不发生重试异常也会执行。因为 Chunk 中的 process 和 write 操作默认使用 org.springframework.batch.retry.support.RetryTemplate 进行执行；所以会导致 open、close 操作会每次都执行。

接下来我们实现自定义的重试拦截器用于记录日志信息，并配置重试拦截器，具体参见代码清单 10-16。

完整配置文件参见：/ch10/job/job-step-retry-listener.xml。

代码清单 10-16 配置重试拦截器

```

1.      <job id="retryListenerJob">
2.          <step id="retryStep">
3.              <tasklet>
4.                  <chunk reader="reader" processor="alwaysExceptionItemProcessor"
5.                        writer="writer" commit-interval="1" retry-limit="3">
6.                      <retry-listeners>
7.                          <listener ref="sysoutRetryListener"></listener>
8.                      </retry-listeners>
9.                      .....
10.                     </chunk>
11.             </tasklet>
12.         </step>
13.     </job>
14.     <bean:bean id="sysoutRetryListener">

```

```
15.         class="com.juxtapose.example.ch10.retry.SystemOutRetryListener"/>
```

其中，6~8行：定义重试拦截器，在重试操作发生时候会触发该拦截器的操作。

SystemOutRetryListener的实现代码参见代码清单10-17。

完整代码参见：com.juxtapose.example.ch10.retry.SystemOutRetryListener。

代码清单10-17 SystemOutRetryListener类定义

```
1.  public class SystemOutRetryListener implements RetryListener {  
2.      public<T>boolean open(RetryContext context,RetryCallback<T> callback) {  
3.          System.out.println("SystemOutRetryListener.open()");  
4.          return true;  
5.      }  
6.  
7.      public <T> void close(RetryContext context, RetryCallback<T> callback,  
8.          Throwable throwable) {  
9.          System.out.println("SystemOutRetryListener.close()");  
10.     }  
11.  
12.     public <T> void onError(RetryContext context, RetryCallback<T> callback,  
13.         Throwable throwable) {  
14.         System.out.println("SystemOutRetryListener.onError()");  
15.     }  
16. }
```

本拦截器的实现仅打印了当前执行操作的日志，没有具体的业务信息，开发人员可以根据需要完成自己的业务逻辑功能。

10.2.4 重试模板

Spring Batch框架为面向批的操作提供了自动重试的能力，如果作业步的实现是自定义的Tasklet，Spring Batch框架提供了一组方便易用的重试模板RetryTemplate，使用重试模板可以方便地完成重试功能。目前框架中面向Chunk的重试功能同样是使用RetryTemplate来完成的。

通过重试模板可以完成无状态的重试、有状态的重试操作。RetryTemplte类关系图参见图10-2。

RetryOperations接口定义了重试操作的基本方法，重试模板实现该接口；RetryTemplte提供标准的重试操作；RetryCallback接口定义了具体的需要重试的逻辑，当具体的重试逻辑发生错误时候，会导致该回调实现的操作按照给定的重试策略进行重试；RetryPolicy接口定义重试策略，可以使用简单的重试策略或者超时策略；BackOffPolicy接口定义了补偿策略，每次重试发生的时候可以都会调用该业务补偿；RecoveryCallback接口定义有状态的业务补偿策略，在所有的重试完成之后会调用该接口完成业务恢复功能；RetryState接口表示重试状态，用来完成有状态的重试。

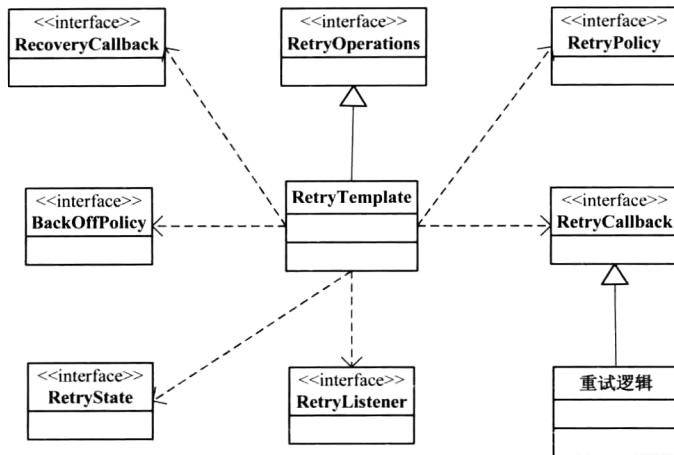


图 10-2 RetryTemplate 类关系图

关键接口、类说明参见表 10-10。

表 10-10 关键接口、类说明

关键类	说 明
RetryOperations	重试操作的接口类，定义了如何调用重试的操作，包括无状态的重试、有状态的重试操作; org.springframework.retry.RetryOperations
RetryTemplate	重试模板类，实现 RetryOperations 接口并组装其他接口完成重试功能； org.springframework.retry.support.RetryTemplate
RetryCallback	重试回调接口，当发生重试的时候会多次调用该回调操作，用户可以实现该接口，完成需要重试的业务逻辑； org.springframework.retry.RetryCallback<T>
RetryPolicy	重试策略，可以使用 Spring Batch 框架提供的简单次数重试策略、超时重试策略、或者自定义的重试策略； org.springframework.retry.RetryPolicy
BackOffPolicy	业务补偿操作，每次重试都会触发该接口的 backOff 操作； org.springframework.retry.backoff.BackOffPolicy
RetryListener	重试拦截器，重试发生期间会触发该拦截器的执行，可以定义多个拦截器； org.springframework.retry.RetryListener
RecoveryCallback	重试执行完毕后，会触发恢复回调操作，通常用在有状态的重试中； org.springframework.retry.RecoveryCallback<T>
RetryState	重试状态，用在有状态重试中，可以根据提供的 key 获取重试上下文； org.springframework.retry.RetryState

关键接口说明

接口 RetryOperations 定义参见代码清单 10-18。

代码清单 10-18 RetryOperations 接口定义

```
1. public interface RetryOperations {  
2.     <T> T execute(RetryCallback<T> retryCallback) throws Exception;  
3.     <T> T execute(RetryCallback<T> retryCallback,  
4.                     RecoveryCallback<T> recoveryCallback) throws Exception;  
5.     <T> T execute(RetryCallback<T> retryCallback,  
6.                     RetryState retryState) throws Exception, ExhaustedRetry  
    Exception;  
7.     <T> T execute(RetryCallback<T> retryCallback,  
8.                     RecoveryCallback<T> recoveryCallback, RetryState retryState)  
    throws Exception;  
9. }
```

RetryOperations 接口定义了重试的基本操作，前两个操作是无状态的重试；后两个操作表示有状态的重试。

接口 RetryCallback 定义参见代码清单 10-19。

代码清单 10-19 RetryCallback 接口定义

```
1. public interface RetryCallback<T> {  
2.     T doWithRetry(RetryContext context) throws Exception;  
3. }
```

其中，2 行：重试发生时候会多次调用 doWithRetry 操作，可以实现该接口在方法 doWithRetry 中实现需要重试的业务逻辑。

接口 BackOffPolicy 定义参见代码清单 10-20。

代码清单 10-20 BackOffPolicy 接口定义

```
1. public interface BackOffPolicy {  
2.     BackOffContext start(RetryContext context);  
3.     void backOff(BackOffContext backOffContext) throws BackOffInterruptedException;  
4. }
```

接口 BackOffPolicy 定义业务补偿操作，start 操作在重试过程中仅执行一次，backOff 操作在每次重试发生后都会触发该补偿操作。

接口 RecoveryCallback 定义参见代码清单 10-21。

代码清单 10-21 RecoveryCallback 接口定义

```
1. public interface RecoveryCallback<T> {  
2.     T recover(RetryContext context) throws Exception;  
3. }
```

recover 操作在整个重试操作完成后会被触发。

接口 RetryState 定义参见代码清单 10-22。

代码清单 10-22 RetryState 接口定义

```
1. public interface RetryState {  
2.     Object getKey();  
3.     boolean isForceRefresh();  
4.     boolean rollbackFor(Throwable exception);  
5. }
```

RetryState 提供了有状态重试的能力, getKey 操作提供了重试逻辑的唯一标识, 使用该 key 从 RetryContext 缓存中获取 RetryContext; isForceRefresh 操作定义是否每次重新刷新, 如果是重新刷新则下次获取到的是一个新的 RetryContext; rollbackFor 定义哪些类型的异常需要进行回滚操作, 如果发生回滚则不会进行重试操作。

具有重试功能的 Tasklet

接下来我们使用重试模板提供的 API 来实现自定义的 Tasklet, 该 Tasklet 具有重试的功能。我们实现信用卡账单 CreditBillTasklet, 内部使用使用重试模板, 使得 CreditBillTasklet 具有错误重试的功能。

CreditBillTasklet 的实现类参见代码清单 10-23。

完整代码参见: com.juxtapose.example.ch10.retry.template.CreditBillTasklet。

代码清单 10-23 CreditBillTasklet 类实现

```
1. public class CreditBillTasklet implements Tasklet {  
2.     public RepeatStatus execute(StepContribution contribution,  
3.         ChunkContext chunkContext) throws Exception {  
4.         RetryCallback<String> retryCallback = new DefaultRetryCallback();  
5.         RetryListener[] listeners = new RetryListener[]{new CountRetry  
Listener()};  
6.         SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();  
7.         retryPolicy.setMaxAttempts(3);  
8.         RetryTemplate template = new RetryTemplate();  
9.         template.setRetryPolicy(retryPolicy);  
10.        template.setListeners(listeners);  
11.        template.execute(retryCallback);  
12.        return RepeatStatus.FINISHED;  
13.    }  
14. }
```

其中, 4 行: 定义重试的逻辑操作, DefaultRetryCallback 定义了具体的业务实现, 当错误发生的时候, DefaultRetryCallback 会被执行重试操作。

5 行: 定义重试的拦截器, 使用 template.setListeners (listeners) 注入到重试模板中。

6~7 行: 定义重试策略, 最大重试次数设置为 3 次。

11 行: 使用重试模板执行重试回调操作。

`DefaultRetryCallback` 实现接口 `RetryCallback`，实现类中定义了需要重试的逻辑，`doWithRetry` 中的逻辑发生错误时候，根据重试模板定义的重试策略会重复执行 `doWithRetry` 操作，直到不再满足重试策略。

`DefaultRetryCallback` 的实现参见代码清单 10-24。

完整代码参见：com.juxtapose.example.ch10.retry.template.DefaultRetryCallback。

代码清单 10-24 `DefaultRetryCallback` 类定义

```
1. public class DefaultRetryCallback implements RetryCallback<String> {
2.     .....
3.
4.     public String doWithRetry(RetryContext context) throws Exception {
5.         Integer count = (Integer)context.getAttribute("count");
6.         if(count == null){
7.             count = new Integer(0);
8.         }
9.         count++;
10.        context.setAttribute("count", count);
11.        Thread.sleep(sleepTime);
12.        throw new RuntimeException("Mock make exception on business logic.");
13.    }
14. }
```

其中，4~13 行：执行实际的业务逻辑操作，最后模拟每次调用都发生 `RuntimeException` 异常；注意本处使用了重试上下文 `RetryContext`，在有状态的重试操作中，可以使用重试上下文在多次重试期间进行数据的共享。

接下来在 Job 文件中配置 `CreditBillTasklet`，具体参见代码清单 10-25。

完整配置文件参见：/ch10/job/job-step-retry-tasklet.xml。

代码清单 10-25 配置 `CreditBillTasklet`

```
1. <job id="retryTaskletJob">
2.     <step id="retryTaskletStep">
3.         <tasklet ref="retryTasklet"></tasklet>
4.     </step>
5. </job>
6.
7. <bean:bean id="retryTasklet"
8.     class="com.juxtapose.example.ch10.retry.template.
CreditBillTasklet" />
```

2 行：声明自定义的 Tasklet，本身具有重试错误的能力，最大重试次数为 3 次。

补偿策略 BackOffPolicy

重试模板提供了补偿策略，当执行重试的时候，可以为每次重试执行补偿动作，框架提

供了接口 BackOffPolicy 来实现该功能。

我们使用如下的示例来验证补偿操作。在重试发生的时候使用计数器来验证功能，CountRetryListener 在发生重试异常的时候计数器加 1（参见代码清单 10-26），在补偿策略实现 DefaultBackoffPolicy 中计数器减 1（参见代码清单 10-27）。

CountRetryListener 拦截器实现参见代码清单 10-26。

完整代码参见：com.juxtapose.example.ch10.retry.template.CountRetryListener。

代码清单 10-26 CountRetryListener 类定义

```
1. public class CountRetryListener implements RetryListener {  
2.  
3.     public<T>boolean open(RetryContext context,RetryCallback<T> callback) {  
4.         return true;  
5.     }  
6.  
7.     public <T> void close(RetryContext context, RetryCallback<T> callback,  
8.                           Throwable throwable) {  
9.  
10.    public <T> void onError(RetryContext context, RetryCallback<T> callback,  
11.                           Throwable throwable) {  
12.        CountHelper.increment();  
13.        System.out.println("CountRetryListener.onError() .");  
14.    }  
15. }
```

其中，12 行：拦截器中每次发生重试的时候，计数器加 1；

DefaultBackoffPolicy 的实现参见代码清单 10-27。

完整代码参见：com.juxtapose.example.ch10.retry.template.DefaultBackoffPolicy。

代码清单 10-27 DefaultBackoffPolicy 类定义

```
1. public class DefaultBackoffPolicy implements BackOffPolicy {  
2.     public BackOffContext start(RetryContext context) {  
3.         BackOffContextImpl backOffContext = new BackOffContextImpl  
4.             (context);  
5.         return backOffContext;  
6.  
7.     public void backOff(BackOffContext backOffContext)  
8.         throws BackOffInterruptedException {  
9.         Assert.assertNotNull(((BackOffContextImpl)backOffContext).  
10.                         getRetryContext().getAttribute("count"));  
11.         CountHelper.decrement();  
12.     }  
13. }
```

其中，2~5 行：实现 start 操作，这里根据重试上下文 RetryContext 生成补偿上下文，并持有 RetryContext 的句柄。

7~12 行：每次执行业务的补偿后，计数器减 1。

补偿上下文 BackOffContextImpl 实现接口 BackOffContext，具体实现参见代码清单 10-28。

完整代码参见：com.juxtapose.example.ch10.retry.template.BackOffContextImpl。

代码清单 10-28 BackOffContextImpl 类定义

```
1. public class BackOffContextImpl implements BackOffContext {
2.     private RetryContext retryContext;
3.
4.     public BackOffContextImpl(){}
5.     public BackOffContextImpl(RetryContext retryContext){
6.         this.retryContext = retryContext;
7.     }
8.
9.     public RetryContext getRetryContext() {
10.         return retryContext;
11.     }
12.     public void setRetryContext(RetryContext retryContext) {
13.         this.retryContext = retryContext;
14.     }
15. }
```

补偿上下文持有重试上下文对象，在有状态的重试中可以通过这种方式在每次补偿动作发生的时候获取重试上下文 RetryContext。

使用 JUnit 单元测试来验证补偿策略，参见代码清单 10-29。

完整代码参见：test.com.juxtapose.example.ch10.RetryTemplateTestCase。

代码清单 10-29 JUnit 单元测试来验证补偿策略

```
1. @Test
2. public void testBackoffPolicy(){
3.     RetryCallback<String> retryCallback = new DefaultRetryCallback();
4.     SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
5.     retryPolicy.setMaxAttempts(3);
6.     RetryListener[] listeners=new RetryListener[]{new CountRetryListener()};
7.     BackOffPolicy backOffPolicy = new DefaultBackoffPolicy();
8.     RetryTemplate template = new RetryTemplate();
9.     template.setRetryPolicy(retryPolicy);
10.    template.setListeners(listeners);
11.    template.setBackOffPolicy(backOffPolicy);
12.    try {
13.        template.execute(retryCallback);
14.        Assert.assertFalse(true);
15.    }
```

```
15.     } catch (Exception e) {
16.         Assert.assertTrue(true);
17.         Assert.assertEquals(1, CountHelper.getCount());
18.     }
19. }
```

其中，7行：定义补偿策略。

11行：将补偿策略设置到重试模板中，在执行重试操作时候会触发补偿策略。

17行：断言执行补偿策略后，计数器的值保持为1；这里为什么不是0，这是因为补偿策略只有在异常被抛出之前执行，最后一次的重试导致了失败，不会触发补偿策略的执行。

有状态重试

重试模板提供了有状态重试的功能，需要使用 `RetryState` 作为参数传入到重试模板。`RetryState` 的关键作用是提供缓存 `RetryContext` 的 key，定义那些异常不需要重试而是执行回滚操作。通过筛选器 `Classifier` 根据异常类型返回 `true` 或者 `false` 来实现后者的功能。

代码清单 10-30 展示了如何使用有状态的重试操作。

完整代码参见：[test.com.juxtapose.example.ch10.RetryTemplateTestCase](http://test.com/juxtapose/example/ch10/RetryTemplateTestCase)。

代码清单 10-30 `RetryTemplateTestCase` 类定义

```
1.  @Test
2.  public void testSimpleRetryPolicyStatefull(){
3.      RetryCallback<String> retryCallback = new DefaultRetryCallback();
4.      SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
5.      retryPolicy.setMaxAttempts(3);
6.      RetryListener[] listeners=new RetryListener[]{new CountRetryListener()};
7.      BackOffPolicy backOffPolicy = new DefaultBackoffPolicy();
8.      RetryTemplate template = new RetryTemplate();
9.      template.setRetryPolicy(retryPolicy);
10.     template.setListeners(listeners);
11.     template.setBackOffPolicy(backOffPolicy);
12.     @SuppressWarnings({"unchecked", "rawtypes"})
13.     Classifier<? super Throwable, Boolean> classifier =
14.             new ClassifierSupport(Boolean.FALSE);
15.     RetryState retryState = new DefaultRetryState("key", false,
16.             classifier);
16.     try {
17.         template.execute(retryCallback,retryState);
18.         Assert.assertFalse(true);
19.     } catch (Exception e) {
20.         Assert.assertTrue(true);
21.         Assert.assertEquals(1, CountHelper.getCount());
22.     }
23. }
```

其中，13~14 行：定义异常筛选类，本处使用框架提供的 `Support` 类，对所有类型的异常都会返回 `false`，`false` 表示重试逻辑发生任何异常的时候都不会导致 `rollback` 操作，而是执行重试操作。

15 行：定义 `RetryState`，本节使用“key”来作为唯一的关键字，将 `RetryContext` 通过关键字“key”缓存到 `Map` 中，后续每次重试的时候都能通过该关键字获取 `RetryContext`。

10.3 重启 Restart

即便再健壮的 Job，解决了 `Skip`、`Retry` 的问题，也有可能最终执行 Job 失败。在 Job 失败的场景下是让用户重头再次执行 Job 还是能够从上次 Job 失败的地点重新执行 Job？Spring Batch 框架提供了重启 Job 的功能，包括重启 Job、Step 支持重启、重启已经完整的 Step、禁止 Step 重启、限制重启次数等功能。

Job 的重启是通过多个 Job Execution 来完成的，每个 Job Instance 可以有多个 Job Execution，Job 执行器负责完成 Job 实例。

一个典型的任务重启的流程参见图 10-3。

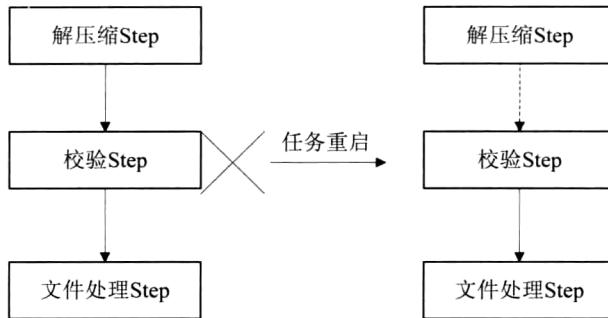


图 10-3 典型的任务重启的流程

第一次 Job 执行过程中校验 Step 出错，接下来重启 Job 会重上次失败的点进行重新启动 Job。Job 的两次执行过程中，对应同一个 Job 的实例，但是执行器是不同的两个执行器。

Job 定义、Job 实例、Job 执行器三者的关系参见图 10-4。

10.3.1 重启 Job

Spring Batch 框架对重启 Job 有如下的规则可以遵守。

- (1) 只能重启状态为失败的 Job 实例。
- (2) 任何 Job 失败的实例都可以被重新执行。
- (3) 重新执行 Job 的时候，会从上次执行失败的点重新开始执行，而不是从头开始执行。
- (4) 已经完成的 Step，通过特殊的标识也可以被重新执行。

(5) 一个失败的 Job 可以被不断的执行，取费有重启次数的限制。

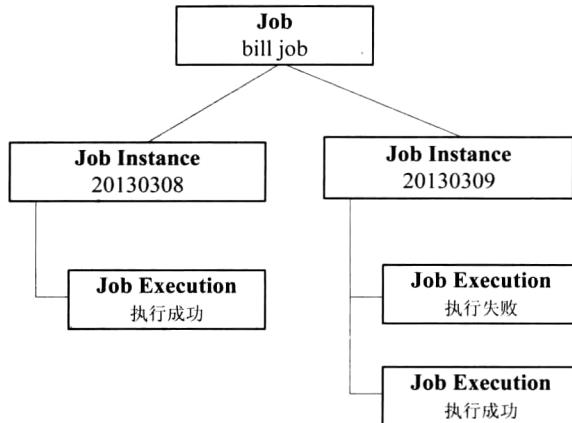


图 10-4 Job 定义、Job 实例、Job 执行器三者的关系

重启 Job 的几个关键属性包括 restartable、allow-start-ifcomplete、start-limit，具体描述参见表 10-11。

表 10-11 重启属性说明

属性	所属节点	说 明	默认值
restartable	Job	定义当前作业是否支持重启，默认值是 true，表示支持重启，如果不需要重启，需要显示设置为 false	True
allow-start-ifcomplete	tasklet	是否允许完成(状态为"COMPLETED")的 Step 重新启动	false
start-limit	tasklet	Sep 能够启动的最大次数 超过最大次数后会抛出异常	Integer.MAX_VALUE

代码清单 10-31 展示了如何设置 Job 是可以重新启动的。

代码清单 10-31 设置 Job 可以重新启动

```
1.      <job id="billJob" restartable="true">
2.          <step id="billStep">
3.              <tasklet transaction-manager="transactionManager">
4.                  <chunk reader="csvItemReader" writer="csvItemWriter"
5.                      processor="creditBillProcessor" commit-interval="2">
6.                  </chunk>
7.              </tasklet>
8.          </step>
9.      </job>
```

其中，1 行：设置当前的 Job 可以重新启动，如果设置属性 `restartable` 的值为 `false`，则该 Job 不支持重新启动。

10.3.2 启动次数限制

默认情况下，作业实例可以无限次地重复启动。在有些场景下需要限制作业实例的启动次数，例如在执行任务分区（particular）的 Step 中，需要对分区的 Step 限制启动次数为 1 次，因为在数据错误的情况下，多次重启 Step 没有任何意义。当然，读者可以找到更多的场景来使用限制任务重启的次数。限制启动次数示例参见代码清单 10-32。

代码清单 10-32 配置启动次数限制

```
1. <step id="startLimitStep">
2.   <tasklet start-limit="2">
3.     <chunk reader="reader" processor="processor" writer="writer"/>
4.   </tasklet>
5. </step>
```

定义 `startLimitStep` 仅能启动二次，第三次启动的时候会抛出异常。`start-limit` 默认值是 `Integer.MAX_VALUE`，表示任务可以无限次地启动。

10.3.3 重启已完成的任务

通常情况下已经完成的 Step 默认是不会自动重新启动的，可以通过属性 `allow-start-if-complete` 来设置已经完成的 Step 在重启的时候可以再次被执行，具体的效果图参见图 10-5。

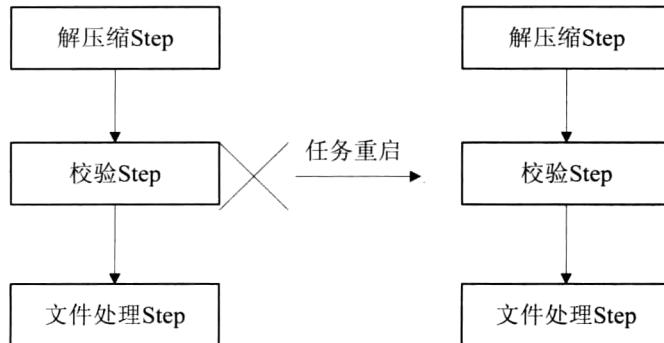


图 10-5 重启已经完成的 Step

默认情况下，Job Instance 重新启动的时候，已经完成的任务不会再次被执行。但在某些特殊场景下，已经完成的 Step 在任务重启的时候需要再次执行，可以通过属性 `allow-start-if-complete` 来设置。示例配置文件参见代码清单 10-33。

代码清单 10-33 配置可重启的完成任务

```
1.      <job id="conditionalJob" >
2.          <step id="decompressStep" parent="abstractDecompressStep"
3.              next="verifyStep" >
4.                  <tasklet ref="decompressTasklet" allow-start-if-complete="true"/>
5.              </step>
6.              .....
7.              <step id="cleanStep">
8.                  <tasklet ref="cleanTasklet" />
9.              </step>
10.         </job>
```

其中，4 行：定义解压缩 Step 即时在状态为完成的情况下，仍然可以再次被执行。

扩展 Job、并行处理

前面我们基本掌握了如何使用 Spring Batch 开发批处理任务，面向批处理的作业通常涉及大数据的处理、高的资源消耗。如何实现性能可靠的、可扩展的作业是一个比较高的挑战工作。在介绍如何开发高扩展性的 Job 之前，我们首先了解下软件系统的可扩展性。

11.1 可扩展性

可扩展性（可伸缩性）是一种对软件系统计算处理能力的设计指标，高可扩展性代表一种弹性，在系统扩展成长过程中，软件能够保证旺盛的生命力，通过很少的改动甚至只是硬件设备的添置，就能实现整个系统处理能力的线性增长，实现高吞吐量和低延迟高性能。

软件系统的扩展通常可以通过如下的两种方式实现，垂直扩展（参见图 11-1）、水平扩展（参见图 11-2）。

垂直扩展是通过升级原有的服务器或者为当前的应用更换更强大的硬件来实现系统处理能力的增强。比如为当前的服务器增加更大的内存、存储、处理器资源等。垂直扩展通常要求提供更强大的服务器资源来达到增加软件处理的能力；通过更换更强的服务器可以方便地实现软件系统的可扩展性，但是这种便利性同时也有较大的局限性，毕竟这种处理能力的增加是有限的，因为单个服务器的处理能力毕竟最终是有限的。

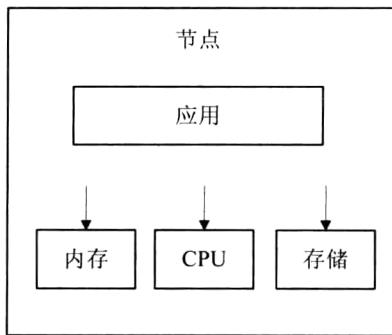


图 11-1 垂直扩展

水平扩展指的是通过增加更多的服务器来分散负载，可以将多个服务器从逻辑上看成一个实体，从而实现存储能力和计算能力的扩展。比如，可以简单地通过聚类或负载平衡策略，通过增加多个服务器来加快整个逻辑实体的运行速度及性能。

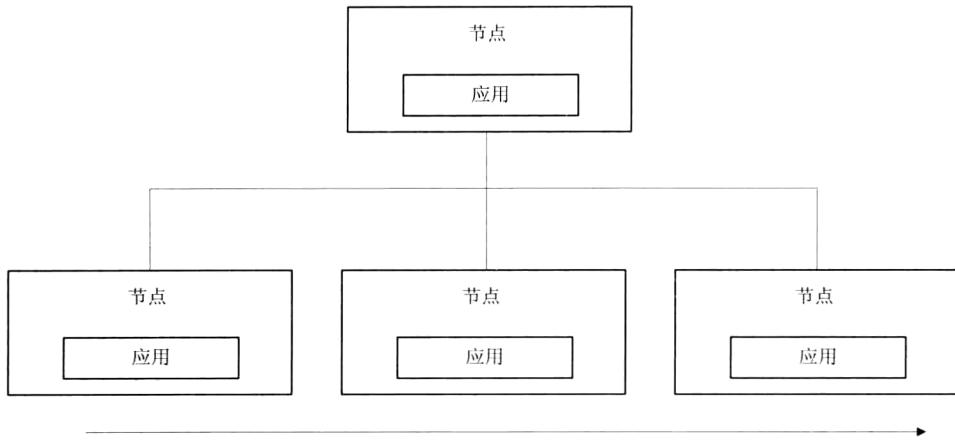


图 11-2 水平扩展

Spring Batch 框架提供了多种提高 Job 并行处理、扩展性的方式。通常情况下只需要调整 Job 的配置就可以达到扩展处理 Job 的目的。需要注意的是框架提供了在 Step 级别进行任务的扩展能力。

框架提供的扩展能力包括如下的四种模式，具体参见表 11-1。

表 11-1 Spring Batch 框架提供扩展能力的四种模式

扩展模式	Local/Remote	说 明
Multithreaded step 多线程作业步	Local	Step 可以使用多线程执行（通常一个 Step 是由一个线程执行的）
Parallel step 并行作业步	Local	Job 执行期间，不同的 Step 并行处理，由不同的线程执行（通常 Job 的 Step 都是顺序执行，且由同一个线程执行的）
Partitioning step 分区作业步	Local/Remote	通过将任务进行分区，不同的 Step 处理不同的任务数据达到提高 Job 效率的功能
Remote chunking 远程任务	Remote	将任务分发到远程不同的节点进行并行处理，提高 Job 的处理速度和效率

接下来我们将学习上面的四种扩展模式。

11.2 多线程 Step

批处理框架在 Job 执行时默认使用单个线程完成任务的执行，同时框架提供了线程池的支持，可以在 Step 执行时候进行并行处理，这里的并行是指同一个 Step 使用线程池进行执行，同一个 Step 被并行地执行。使用 tasklet 的属性 task-executor 可以非常容易地将普通的 Step 变成多线程 Step。配置多线程 Step 属性具体说明参见表 11-2。

表 11-2 配置多线程 Step 属性说明

属性	说 明	默 认 值
task-executor	任务执行处理器，定义后表示采用多线程执行任务，需要考虑多线程执行任务时候的安全性	
throttle-limit	最大使用线程池的数目	6

11.2.1 配置多线程 Step

代码清单 11-1 给出了配置多线程 Step 的示例。

完整配置参见文件：/ch11/job/job-multithreade.xml。

代码清单 11-1 配置多线程 Step 示例

```
1. <job id="multiThreadJob">
2.   <step id="multiThreadStep">
3.     <tasklet task-executor="taskExecutor" throttle-limit="6">
4.       <chunk reader="reader" processor="processor" writer="writer"
5.         commit-interval="2"/>
6.     </tasklet>
7.   </step>
8. </job>
```

其中，3 行：属性 task-executor 定义执行 Step 需要的线程池，属性 throttle-limit 用于限制能使用的最大的线程数目；线程池的具体配置参见代码清单 11-2。

线程池的配置

代码清单 11-2 线程池的配置

```
1. <bean:bean id="taskExecutor"
2.   class="org.springframework.scheduling.concurrent.
3.   ThreadPoolTaskExecutor">
4.   <bean:property name="corePoolSize" value="5"/>
5.   <bean:property name="maxPoolSize" value="15"/>
6. </bean:bean>
```

为了便于展示不同的线程执行上面的 Step，我们在 Read、Write 的实现中增加了打印当前线程的代码，执行上面的 Job，在控制台能够打印出代码清单 11-3 的片段。

代码清单 11-3 多线程 Step 执行控台输出结果

```
1. Read:0;Job Read Thread name: taskExecutor-1
2. Read:1;Job Read Thread name: taskExecutor-1
3. Write begin:0,1,Write end!!Job Write Thread name: taskExecutor-1
4. Read:2;Job Read Thread name: taskExecutor-3
5. Read:3;Job Read Thread name: taskExecutor-3
6. Write begin:2,3,Write end!!Job Write Thread name: taskExecutor-3
```

```
7. Read:4;Job Read Thread name: taskExecutor-2
8. Read:5;Job Read Thread name: taskExecutor-2
9. Write begin:4,5,Write end!!Job Write Thread name: taskExecutor-2
10. Read:6;Job Read Thread name: taskExecutor-4
11. Read:7;Job Read Thread name: taskExecutor-4
12. Write begin:6,7,Write end!!Job Write Thread name: taskExecutor-4
13. Read:8;Job Read Thread name: taskExecutor-5
14. Read:9;Job Read Thread name: taskExecutor-5
15. Write begin:8,9,Write end!!Job Write Thread name: taskExecutor-5
16. Read:10;Job Read Thread name: taskExecutor-1
17. Read:11;Job Read Thread name: taskExecutor-1
18. Write begin:10,11,Write end!!Job Write Thread name: taskExecutor-1
19. Read:12;Job Read Thread name: taskExecutor-3
20. Read:13;Job Read Thread name: taskExecutor-3
21. Write begin:12,13,Write end!!Job Write Thread name: taskExecutor-3
22. .....
```

可以看出在 Step 执行期间共有 5 个不同的线程进行了作业 Step 处理；另外大家需要注意在 Step 的读、处理、写执行时，是在一个完整的线程中进行处理的；另外一个需要注意的是同一个线程在进行任务处理的时候，其处理的数据是不连续的，例如线程 taskExecutor-1 在处理完数据 0、1 之后，接下来处理的数据为 10,11。

说明：

在多线程 Step 中为了保证代码处理的正确性，要求所有在多线程 Step 中处理的对象和操作必须是线程安全的；简单期间任何无状态的处理操作都是线程安全的，对于有状态的操作可以通过特殊的处理变成线程安全的操作。但是 Spring Batch 框架提供的大部分的 ItemReader、ItemWriter 等操作都是线程不安全的，最主要的原因在于 ItemReader、ItemWriter 提供了可重启特性的支持，在运行期间保存了大量的运行期状态导致了 ItemReader、ItemWriter 操作均是有状态的，不能直接运用在多线程步中。

11.2.2 线程安全性

我们首先学习一下什么是线程安全的，如果代码所在的进程中多个线程在同时运行，而这些线程可能会同时运行这段代码；如果每次运行结果和单线程运行的结果是一样的，而且其他变量的值也和预期的一样的，我们称之为代码是线程安全的。

在 Java 领域线程安全问题通常是全局变量或者静态变量引起的，若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有个线程同时执行写操作，则需要考虑线程同步，否则的话就可能影响线程安全。

我们以 Java 中的 ArrayList 举例，向 List 中增加元素（假设目前 List 中有 0 个元素），来描述线程的安全性。

单线程：在单线程运行的情况下，向 List 中增加一个元素，当前元素的位置为 0，共有 1

个元素；继续增加一个元素，因为是串行执行，结果是当前元素的位置为 1，共有 2 个元素。

多线程：假设有 2 个线程（分别是线程 1、线程 2）执行，线程 1 先将元素存放在位置 0，此时 CPU 调度线程 1 暂停，线程 2 得到运行的机会，线程 2 也向此 ArrayList 添加元素，因为此时 Size 仍然等于 0，所以线程 2 也将元素存放在位置 0；然后线程 1 和线程 2 都继续运行，都增加 Size 的值；最终的结果是 List 的 Size 为 2，但实际上只有 1 个元素，这就导致了 List 是线程不安全的。

Spring Batch 框架中提供的大量 ItemReader、ItemWriter 通常是线程不安全的，因为大多数的 ItemReader、ItemWriter 是有状态的，导致了无法直接在多线程 Step 中直接使用这些基础设施。通常可以通过 synchronized 来实现并发控制，或者通过业务规则来实现线程安全的 ItemReader、ItemWriter 等组件。

接下来我们将学习如何实现线程安全的 ItemReader。

11.2.3 线程安全 Step

ItemReader、ItemWriter 在运行过程中保存了部分状态信息，用于支撑 Step 的重启操作。这些状态导致是非线程安全的组件。接下来我们介绍如何实现线程安全的 ItemReader。本节我们以 JdbcCursorItemReader 为例来实现线程安全的组件。

JdbcCursorItemReader 不是线程安全的，不能直接使用于多线程 Step 中。一个简单的方法是对 read 操作使用关键字 synchronized 进行同步执行，这样可以保证在执行过程中保证线程安全性。使用 synchronized 会造成多线程读取时候的阻塞，但我们考虑下载批处理的作业中读操作通常是非常轻量级的，更多的处理器资源都消耗在处理操作、写操作上。因此对读操作的同步等待通常不会造成性能瓶颈。

如果在实际的业务开发中同步读导致性能问题，我们可以采取数据分区的功能来解决。具体分区 Step 的实现请参见 11.4 节分区 Step。

为了使 JdbcCursorItemReader 保证线程安全的，我们实现一个同步的 SynchronizedItem Reader，参见代码清单 11-4，对 read 操作使用 synchronized 关键字。

完整代码参见：com.juxtapose.example.ch11.multithread.SynchronizedItemReader。

代码清单 11-4 SynchronizedItemReader 类定义

```
1. public class SynchronizedItemReader implements ItemReader<CreditBill>,
   ItemStream{
2.
3.     private ItemReader<CreditBill> delegate;
4.
5.     public synchronized CreditBill read() throws Exception {
6.         CreditBill creditBill = delegate.read();
7.         return creditBill;
8.     }
9. }
```

```

10.    public void close() throws ItemStreamException {
11.        if (this.delegate instanceof ItemStream) {
12.            ((ItemStream)this.delegate).close();
13.        }
14.    }
15.
16.    public void open(ExecutionContext context) throws ItemStreamException {
17.        if (this.delegate instanceof ItemStream) {
18.            ((ItemStream)this.delegate).open(context);
19.        }
20.    }
21.
22.    public void update(ExecutionContext context) throws
23.        ItemStreamException {
24.        if (this.delegate instanceof ItemStream) {
25.            ((ItemStream)this.delegate).update(context);
26.        }
27.    .....
28. }

```

其中，3行：代理给定的 ItemReader，此处我们使用 JdbcCursorItemReader。

5行：给定的 read 操作使用 synchronized，保证多线程并发时刻按照处理器分配的顺序执行代理类的 read 操作，保证读操作的线程安全特性。

10~26行：实现 ItemStream 接口的 open、close、update 操作。

配置线程安全的 ItemReader 的 Job，配置代码参见代码清单 11-5。

完整配置文件参见：/ch11/job/job-multithreade-db.xml。

代码清单 11-5 配置线程安全的 ItemReader 的 Job

```

1. <job id="dbSynchronizedJob">
2.     <step id="dbSynchronizedStep">
3.         <tasklet task-executor="taskExecutor" throttle-limit="6">
4.             <chunk reader="creditBillItemReader"
5.                 processor="creditBillProcessor"
6.                 writer="jdbcSetterItemWriter" commit-interval="2">
7.                 </chunk>
8.             </tasklet>
9.         </step>
10.    </job>
11.
12.    <bean:bean id="creditBillItemReader"
13.        class="com.juxtapose.example.ch11.multithread.
14.        SynchronizedItemReader">
15.        <bean:property name="delegate" ref="jdbcItemReader" />
16.    </bean:bean>

```

```

15.
16. <bean:bean id="jdbcItemReader" scope="step"
17.   class="org.springframework.batch.item.database.
18.   JdbcCursorItemReader" >
19.   <bean:property name="dataSource" ref="dataSource"/>
20.   <bean:property name="sql"
21.     value="select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS from
22.     t_credit"/>
23.   <bean:property name="verifyCursorPosition" value="false"></bean:
property>
24.   <bean:property name="saveState" value="false"></bean:property>
25. </bean:bean>

```

其中，1~9 行：定义 Job，读组件使用线程安全的读，使用上面实现的 SynchronizedItem Reader。

11~14 行：定义线程安全的读组件，代理 JDBC 的数据库读组件。

16~23 行：定义 JDBC 提供的标准的数据库读，需要注意两个属性 saveState 和 verifyCursorPosition 都设置为 false。

11.2.4 可重启的线程安全 Step

上面我们实现了线程安全的读组件，但是上面的实现并不支持重启操作，当执行失败的时候因为没有保存当前读取的状态数据导致无法知道哪些数据已经读取成功，哪些是未读取的。接下来我们学习如何实现可重启的线程安全的读组件。

数据库读取组件 `JdbcCursorItemReader`，我们可以友好地设计数据库表，在读取的表中增加一个字段 Flag，用于标识当前的记录是否已经读取并处理成功，如果处理成功则标识 `Flag=true`，等下次重新读取的时候，对于已经成功读取且处理成功的记录直接跳过处理。

可重启的线程安全的 Step 处理逻辑图参见图 11-3。

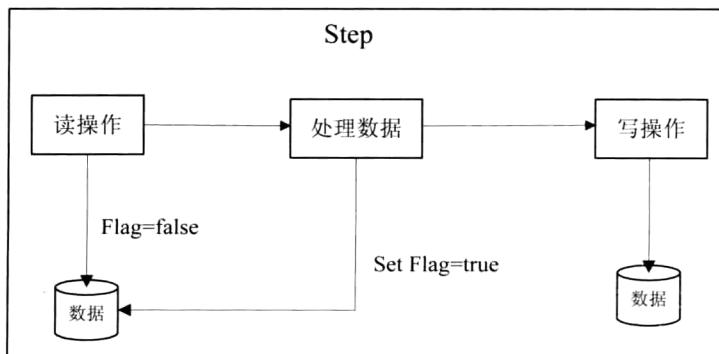


图 11-3 可重启的线程安全的 Step 处理逻辑

需要在数据处理成功后，通过 JDBCTemplate 模板将 Flag 的标识值写入到对应的数据库记录中。

设计新的数据库表 t_credit，增加 Flag 字段用于标识该行记录是否正确的处理过，代码清单 11-6 给出了完整的数据库创建脚本。

完整数据库表参见文件：/ch11/db/create-tables-mysql.sql。

代码清单 11-6 数据库创建脚本

```
1. CREATE TABLE t_credit
2.     (ID VARCHAR(64),
3.      ACCOUNTID VARCHAR(20),
4.      NAME VARCHAR(10),
5.      AMOUNT NUMERIC(10, 2),
6.      DATE VARCHAR(20),
7.      ADDRESS VARCHAR(128),
8.      FLAG VARCHAR(10),
9.      primary key (ID)
10.    )
11.    ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

其中，8 行：字段 FLAG 用于标识当前记录是否执行成功。

可重启的线程安全 Step 的 Job 配置参见代码清单 11-7。

完整配置参见文件：/ch11/job/job-multithreade-db.xml。

代码清单 11-7 配置可重启的线程安全 Step 的 Job

```
1. <job id="dbRestartSynchronizedJob">
2.     <step id="dbRestartSynchronizedStep">
3.         <tasklet task-executor="taskExecutor" throttle-limit="6">
4.             <chunk reader="creditBillRestartItemReader"
5.                   processor="creditBillRestartProcessor"
6.                   writer="jdbcSetterItemWriter" commit-interval="2">
7.                 </chunk>
8.             </tasklet>
9.         </step>
10.    </job>
11.
12.    <bean:bean id="creditBillRestartItemReader"
13.        class="com.juxtapose.example.ch11.multithread.
SynchronizedItemReader">
14.        <bean:property name="delegate" ref="jdbcRestartItemReader" />
15.    </bean:bean>
16.    <bean:bean id="jdbcRestartItemReader" scope="step"
17.        class="org.springframework.batch.item.database.
JdbcCursorItemReader" >
18.        <bean:property name="dataSource" ref="dataSource"/>
```

```
19.      <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
20.          DATE,ADDRESS from t_credit where flag=false"/>
21.      .....
22.  </bean:bean>
```

其中，1~10 行：定义可重启的线程安全的 Job。

5 行：特殊的数据处理，对应的实现类参见代码清单 11-8。

12~14 行：定义线程安全的 Step；此处使用上面用到的 synchronized 的线程安全组件。

16~22 行：定义 JDBC 的读组件，需要注意在读取的 SQL 中需要根据字段 flag 进行过滤，已经处理成功的记录不会被再次被处理。

接下来我们学习如何配置处理器，在处理阶段进行数据库记录的修改，需要重新实现处理类 CreditBillProcessor，实现代码参见代码清单 11-8。

完成类实现参见文件：com.juxtapose.example.ch11.multithread.CreditBillProcessor。

代码清单 11-8 CreditBillProcessor 类定义

```
1.  public class CreditBillProcessor implements
2.      ItemProcessor<CreditBill, DestinationCreditBill> {
3.      JdbcTemplate jdbcTemplate = null;
4.
5.      public DestinationCreditBill process(CreditBill bill) throws Exception {
6.          .....
7.          if(null != jdbcTemplate){
8.              jdbcTemplate.update("update t_credit set flag=? where id=?",
9.                  "true", bill.getId());
10.         }
11.         return destCreditBill;
12.     }
13. }
```

其中，8 行：使用 jdbcTemplate 更新记录的 Flag 标识位为 true。

用户可以基于上面类似的方案实现自定义的可重启的且线程安全的实现。通常情况下实现上述方案比较复杂，对于大数据处理而言，可以将需要处理的数据进行恰当地分区，交给不同的 Job 或者不同的 Step 进行处理，这些 Job 或者 Step 可以使本地的也可以是远程的；通过恰当的分区可以避免实现复杂的线程安全的读/写组件。

11.3 并行 Step

多线程步提供了多个线程执行一个 Step 的能力，但这种场景在实际的业务中使用的并不是非常多。更多的业务场景是 Job 中不同的 Step 没有明确的先后顺序，可以在执行期间并行地执行。Spring Batch 框架提供了并行 Step 的能力。可以通过 Split 元素来定义并行的作业流，

并制定使用的线程池。

并行 Step 的执行关系参见图 11-4。

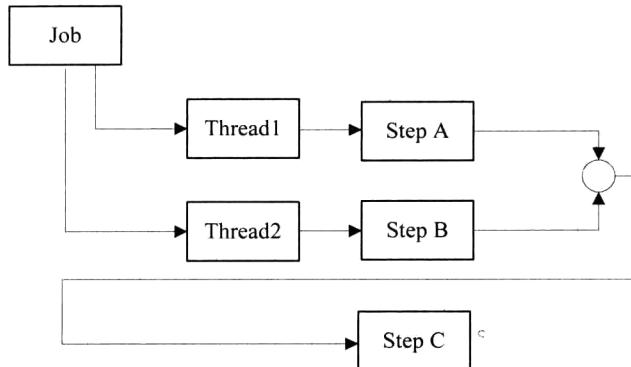


图 11-4 并行 Step 的执行关系

Step A、Step B 两个作业步由不同的线程执行，两者均执行完毕后，Step C 才会被执行。

配置并行执行的 Step 非常的简单，只需要使用 Spring Batch 框架提供的 Split 元素就可以完成，无须额外地编写任何的代码，将复杂度和业务代码的实现进行了有效的隔离，简化了开发难度。具体的如何使用 Split 请参见 9.3 章节的并行 Flow。

并行 Step 提供了在一个节点上横向处理，随着作业处理量的增加，有可能一台节点无法满足 Job 的处理，此时我们可以采用远程 Step 的方式将多个机器节点组合起来完成一个 Job 的处理。我们将在接下来的章节介绍远程 Step。

11.4 远程 Step

远程分块是一个把 step 进行技术分割的工作，不需要对处理数据的结构有明确了解。任何输入源都能够使用单进程读取并在动态分割后作为“块”发送给远程的工作进程。远程进程实现了监听者模式，反馈请求、处理数据，最终将处理结果异步返回。请求和返回之间的传输会被确保在发送者和单个消费者之间。Spring Batch 在 Spring Integration 顶部实现了远程分块的特性。接下来我们将向读者展示如何使用 Spring Integration 的技术实现远程 Step 的功能。

11.4.1 远程 Step 框架

在 Spring Batch 中对远程 Step 没有默认的实现，但是提供了远程 Step 的框架，通过框架可以方便地扩展出远程 Step 的实现。

远程 Step 技术本质上是将对 Item 读、写的处理逻辑进行分离；通常情况下读的逻辑放在一个节点进行操作，将写操作分发到另外的节点执行。

远程 Step 的作业情况参见图 11-5。

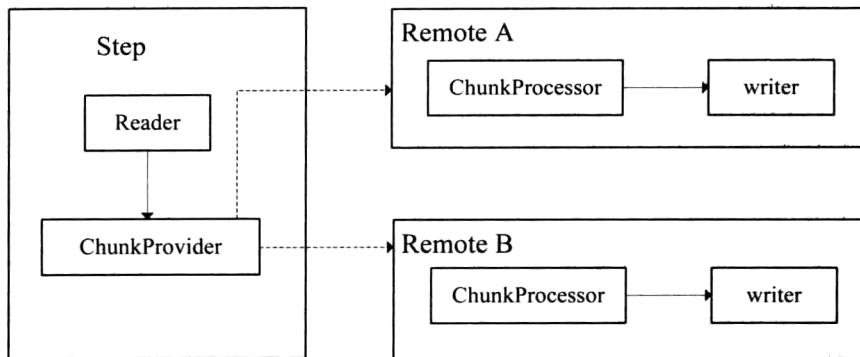


图 11-5 远程 Step 作业逻辑

在 Master 节点，作业步负责读取数据，并将读取的数据通过远程技术发送到指定的远端节点上，进行处理，处理完毕后 Master 负责收回 Remote 端执行的情况。在 Spring Batch 框架中通过两个核心的接口来完成远程 Step 的任务，分别是 ChunkProvider 与 ChunkProcessor。

ChunkProvider：根据给定的 ItemReader 操作产生批量的 Chunk 操作；接口定义参见代码清单 11-9。

代码清单 11-9 ChunkProvider 接口定义

```
1. public interface ChunkProvider<T> {  
2.     Chunk<T> provide(StepContribution contribution) throws Exception;  
3.     void postProcess(StepContribution contribution, Chunk<T> chunk);  
4. }
```

其中，2 行：操作 provider 产生 Chunk 操作，该 Chunk 操作通过各种远程的技术发送到远端进行执行。

ChunkProcessor：负责获取 ChunkProvider 产生的 Chunk 操作，执行具体的写逻辑；接口定义参见代码清单 11-10。

代码清单 11-10 ChunkProcessor 接口定义

```
1. public interface ChunkProcessor<I> {  
2.     void process(StepContribution contribution, Chunk<I> chunk) throws  
Exception;  
3. }
```

其中，2 行：操作 process 负责处理远端获取到的 Chunk。

在 Spring Batch 中对远程 Step 没有默认地实现，但 Spring 中提供了另外个项目，Spring Batch Integration 项目，将 Spring Batch 框架和 Spring Integration 做了集成，可以通过 Spring Integration 提供的远程能力实现远程 Step。接下来我们将介绍如何通过 Spring Integration (SI) 实现远程 Step。

11.4.2 基于 SI 实现远程 Step

Spring Batch Integration 项目中提供了远程 Step 的能力，关键是利用了 SI 提供的远程队列的能力。

Spring Batch Integration 中实现的远程 Step 的结构图参见图 11-6。

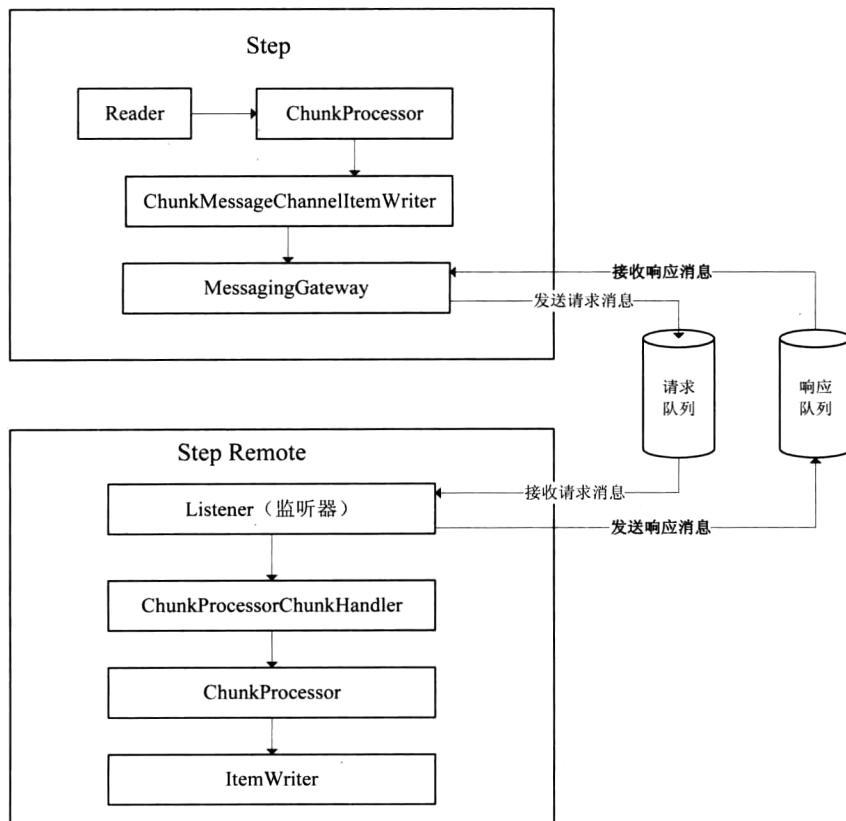


图 11-6 Spring Batch Integration 中实现的远程 Step 的结构图

Step 本地节点负责读取数据，并通过 MessagingGateway 将请求发送到远程 Step 上；远程 Step 提供了队列的监听器，当请求队列中有消息时候获取请求信息并交给 ChunkHander 负责处理。

接下来我们展示如何配置基于 SI 的远程 Step；如果读者对 SI 不熟悉，可以通过 Spring 官方网站学习该技术，本章节不会具体讲解 SI 的技术。

接下来的章节我们采用的示例是从数据库表 t_credit 中读取数据，然后通过远程的方式将 Step 远端执行，将数据写入表 t_destcredit 中。远程 Step 数据处理的示意图参见图 11-7。

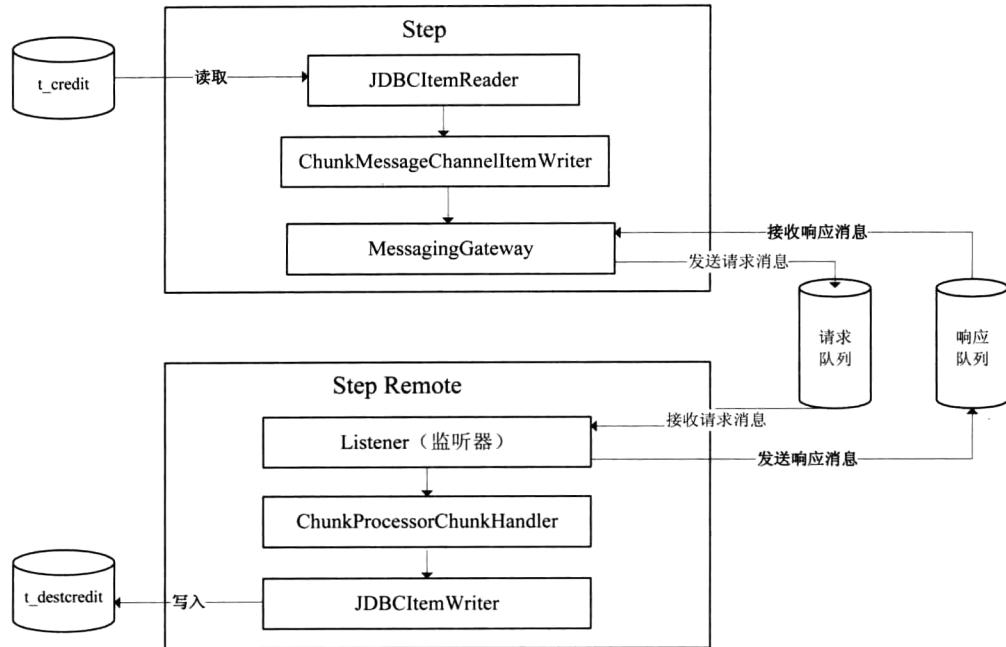


图 11-7 远程 Step 数据处理的示意图

11.4.2.1 配置消息队列

下面定义了需要使用的请求队列、响应队列，以及本地 Step 节点需要的 MessagingGateway，通过 MessagingGateway 可以将请求消息发送到 requests 队列中，并从 replies 队列中获取响应的消息。具体消息队列配置参见代码清单 11-11。

完整配置参见文件：/ch11/job/job-chunk-remote.xml。

代码清单 11-11 配置消息队列

```

1.  <bean:bean id="messagingGateway"
              class="org.springframework.integration.core.MessagingTemplate">
2.      <bean:property name="defaultChannel" ref="requests" />
3.      <bean:property name="receiveTimeout" value="1000" />
4.  </bean:bean>
5.  <int-jms:outbound-channel-adapter connection-factory="connectionFactory"
              channel="requests" destination-name="requests" />
6.  <int:channel id="requests" />
7.  <int:channel id="incoming" />
8.  <int:transformer input-channel="incoming" output-channel="replies" /
9.      ref="headerExtractor" method="extract" />
10. <int:channel id="replies" scope="thread">
11.   <int:queue />

```

```

12.      <int:interceptors>
13.          <bean:bean id="pollerInterceptor"
14.              class="org.springframework.batch.integration.
15.                  chunk.MessageSourcePollerInterceptor">
16.              <bean:property name="messageSource">
17.                  <bean:bean class="org.springframework.integration.
18.                      jms.JmsDestinationPollingSource">
19.                      <bean:constructor-arg>
20.                          <bean:bean class="org.springframework.jms.
21.                              core.JmsTemplate">
22.                              <bean:property name="connectionFactory"
23.                                  ref="connectionFactory" />
24.                              <bean:property name="defaultDestinationName"
25.                                  value="replies" />
26.                              <bean:property name="receiveTimeout"
27.                                  value="100" />
28.                      </bean:bean>
29.                  </bean:constructor-arg>
30.              </bean:bean>
31.          </bean:property>
32.          <bean:property name="channel" ref="incoming"/>
33.      </bean:bean>
34.  </int:interceptors>
35. </int:channel>

```

其中，1~4 行：定义消息网关，负责向 requests 队列发送消息，从 replies 队列收取消息。

5 行：定义请求队列 requests 的 adapter。

6~7 行：定义使用的 channel。

8~9 行：定义消息转换器。

10~29 行：定义响应 channel。

11.4.2.2 配置 AMQ 服务器

接下来的配置定义了 AMQ 的服务器定义，在配置 AMQ 服务器定义时需要在 XML 中指定对应的命名空间 `xmlns:amq="http://activemq.apache.org/schema/core"`。配置 AMQ 服务器参见代码清单 11-12。

完整配置参见文件：`/ch11/job/job-chunk-remote.xml`。

代码清单 11-12 配置 AMQ 服务器

```

1.  <bean:bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
2.      <bean:property name="connectionFactory" ref="connectionFactory" />
3.      <bean:property name="receiveTimeout" value="100" />

```

```
4.      <bean:property name="sessionTransacted" value="true" />
5.  </bean:bean>
6.
7.  <amq:broker useJmx="false" persistent="false" schedulerSupport="false">
8.    <amq:transportConnectors>
9.      <amq:transportConnector uri="tcp://localhost:61616"/>
10.   </amq:transportConnectors>
11. </amq:broker>
12. <amq:connectionFactory id="connectionFactory" brokerURL="tcp://localhost:
61616"/>
```

其中，1~5 行：定义使用的 JMS 消息模板 jmsTemplate。

7~12 行：定义了使用的 AMQ 服务器，默认使用的端口号为 61616。

11.4.2.3 配置本地 Step

本地 Step 负责读取消息，并将对应的 Chunk 通过 JMS 队列的方式发送到远端 Step 进行执行。本地 Step 配置参见代码清单 11-13。

完整配置参见文件：/ch11/job/job-chunk-remote.xml。

代码清单 11-13 配置本地 Step

```
1.  <bean:bean id="chunkWriter" scope="step"
2.    class="org.springframework.batch.integration.
           chunk.ChunkMessageChannelItemWriter" >
3.      <bean:property name="messagingOperations" ref="messagingGateway" />
4.      <bean:property name="replyChannel" ref="replies" />
5.      <bean:property name="maxWaitTimeouts" value="10" />
6.  </bean:bean>
7.
8.  <bean:bean id="chunkHandler"
9.    class="org.springframework.batch.integration.
           chunk.RemoteChunkHandlerFactoryBean">
10.     <bean:property name="chunkWriter" ref="chunkWriter" />
11.     <bean:property name="step" ref="stepRemoteChunk" />
12. </bean:bean>
```

其中，1~6 行：定义 chunkWriter，chunkWriter 负责将本地 Step 中的读取数据通过 messagingGateway 发送端远程 Step 中；ChunkMessageChannelItemWriter 是一个 StepExecution Listener 类型的拦截器。

8~12 行：RemoteChunkHandlerFactoryBean 在创建 chunkHandler 过程中，默认注册了 ChunkProcessor，与 chunkWriter 完成了工作的传递。

11 行：定义远程拦截器调用使用的作业步为 stepRemoteChunk。

除了使用 RemoteChunkHandlerFactoryBean 创建 chunkHandler 外，还可以直接通过

org.springframework.batch.integration.chunk.ChunkProcessorChunkHandler 声明。下面给出直接使用 ChunkProcessorChunkHandler 而不是 RemoteChunkHandlerFactoryBean 的方式创建 chunkHandler，具体参见代码清单 11-14。

完整的配置文件参见：ch11/job/job-chunk-remote-other.xml。

代码清单 11-14 配置本地 Step（使用 ChunkProcessorChunkHandler）

```
1. <bean:bean id="chunkHandler"
2.   class="org.springframework.batch.integration.
      chunk.ChunkProcessorChunkHandler">
3.   <bean:property name="chunkProcessor">
4.     <bean:bean class="org.springframework.batch.core.
       step.item.SimpleChunkProcessor">
5.       <bean:property name="itemWriter" ref="jdbcItemWriter"/>
6.       <bean:property name="itemProcessor">
7.         <bean:bean class="org.springframework.batch.item.
           support.PassThroughItemProcessor"/>
8.       </bean:property>
9.     </bean:bean>
10.    </bean:property>
11.  </bean:bean>
```

其中，1~2 行：通过 ChunkProcessorChunkHandler 定义实现类。

3~10 行：定义面向 Chunk 的处理类 SimpleChunkProcessor。

5 行：面向 Chunk 的处理类 SimpleChunkProcessor 定义使用的 itemWriter。

7 行：面向 Chunk 的处理类 SimpleChunkProcessor 定义使用的 itemProcessor。

11.4.2.4 配置远程 Step

上面的配置完成了队列、服务器、本地 Step 的配置后，接下来只需要完成远程 Step 的配置即可；远程 Step 本质上是一个队列的监听器与收到消息后的处理逻辑。配置远程 Step 参见代码清单 11-15。

完整配置参见文件：/ch11/job/job-chunk-remote.xml。

代码清单 11-15 配置远程 Step

```
1. <jms:listener-container connection-factory="connectionFactory"
2.   transaction-manager="transactionManager"
3.   acknowledge="transacted" concurrency="1">
4.   <jms:listener destination="requests" response-destination="replies"
5.     ref="chunkHandler" method="handleChunk" />
6. </jms:listener-container>
```

其中，1~6 行：配置请求队列的监听器与处理逻辑，处理逻辑调用 chunkHandler 的 handleChunk 操作；该操作接受 chunk 请求，并负责处理逻辑。

5 行：定义监听的队列为 requests，监听到消息后处理逻辑为 chunkHandler 的 handleChunk

操作，处理后的消息发送到 replies 队列。

11.4.2.5 Job 配置及运行

接下来我们定义远程 Job，参见代码清单 11-16。

完整配置参见文件：/ch11/job/job-chunk-remote.xml。

代码清单 11-16 定义远程 Job

```
1. <job id="remoteChunkJob">
2.   <step id="stepRemoteChunk">
3.     <tasklet>
4.       <chunk reader="jdbcItemPageReader" writer="jdbcItemWriter"
           commit-interval="10" />
5.     </tasklet>
6.   </step>
7. </job>
```

本节的 Job 定义和普通的 Job 没有任何区别；通过数据库读、然后再通过数据库写入。jdbcItemPageReader 是从表 t_cretid 读取数据，jdbcItemWriter 将数据写入 t_destcredit 中。为了节省篇幅，jdbcItemPageReader 与 jdbcItemWriter 的具体配置请参见文件/ch11/job/job-chunk-remote.xml。

使用代码清单 11-17 执行定义的 remoteChunkJob。

完整代码参见：test.com.juxtapose.example.ch11.JobLaunchChunkRemote。

代码清单 11-17 执行 remoteChunkJob

```
1. JobLaunchBase.executeJob("ch11/job/job-chunk-remote.xml",
  "remoteChunkJob",
2.   new JobParametersBuilder().addDate("date", new Date()));
```

测试用例执行完毕后，可以看到库表 t_destcredit 中的数据被远程写入。

至此，完成了远程 Step 的开发示例。读者可以基于 Spring Batch 提供的远程框架自己实现远程 Step 工作，从而提升作业的处理效率。

11.5 分区 Step

通过将任务进行分区，不同的 Step 处理不同的任务数据达到提高 Job 效率的功能。分区模式需要对数据的结构有一定的了解，如主键的范围、待处理的文件的名字等。这种模式的优点在于分区中每一个元素的处理器都能够像一个普通 Spring Batch 任务的单步一样运行，也不必去实现任何特殊的或是新的模式，来让他们能够更容易配置与测试。分区理论上比远程更有扩展性，因为分区并不存在从一个地方读取所有输入数据并进行序列化的瓶颈。

分区作业典型的可以分成两个处理阶段，数据分区、分区处理；分区作业逻辑结构参见图 11-8。

数据分区：根据特殊的规则（例如：根据文件名称，数据的唯一性标识，或者哈希算法）将数据进行合理地切片，为不同的切片生成数据执行上下文 Execution Context、作业步执行器 Step Execution。可以通过接口 Partitioner 生成自定义的分区逻辑，Spring Batch 批处理框架默认对多文件实现 org.springframework.batch.core.partition.support.MultiResourcePartitioner；读者可以自行扩展接口 Partitioner 来实现自定义的分区逻辑。

分区处理：通过数据分区后，不同的数据已经被分配到不同的作业步执行器中，接下来需要交给分区处理器进行作业，分区处理器可以在本地执行也可以在远程执行被划分的作业。接口 PartitionHandler 定义了分区处理的逻辑，Spring Batch 批处理框架默认实现了本地多线程的分区处理 org.springframework.batch.core.partition.support.TaskExecutorPartitionHandler；读者可以自行扩展接口 PartitionHandler 来实现自定义的分区处理逻辑。

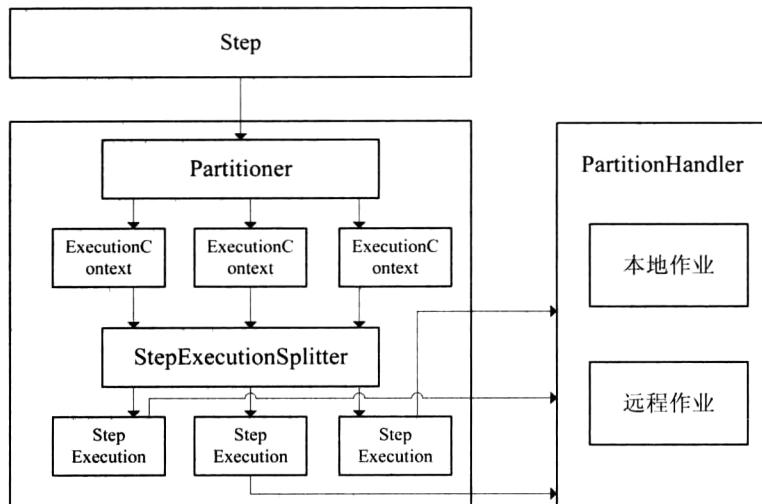


图 11-8 分区作业逻辑结构

11.5.1 关键接口

在 Spring Batch 中有如下接口支持分区：PartitionHandler、StepExecutionSplitter、Partitioner。核心接口关键类图参见图 11-9。

PartitionHandler 知道执行结构-它需要将请求发送到远程步骤并使用任何可以使用的远程技术收集计算结果，PartitionHandler 是一个 SPI，Spring Batch 通过 TaskExecutor 为本地执行提供了一个默认实现，在需要进行有大量 I/O 操作的并发处理时，这个功能是很有用的；Partitioner 接口定义了根据数据结构将作业进行分区，生成执行上下文 Execution Context；StepExecutionSplitter 根据给定 Partitioner 产生的执行上下文生成作业步执行器，然后交给 PartitionHandler 来进行处理。

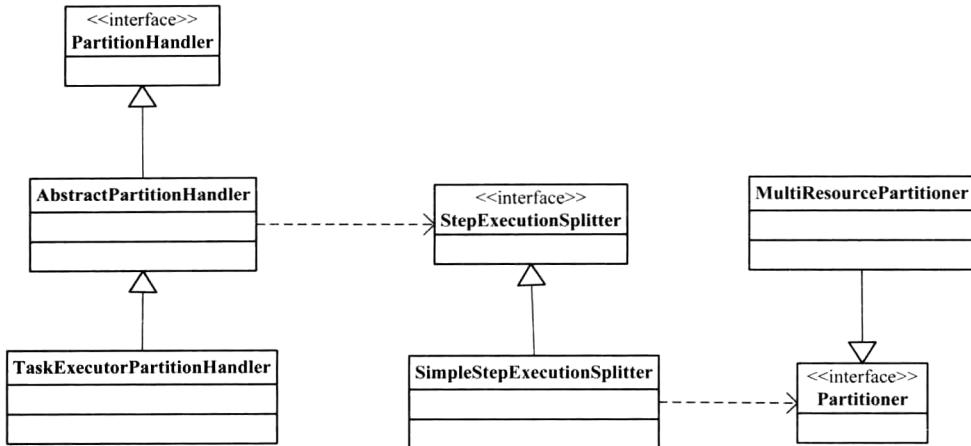


图 11-9 分区关键类图

11.5.1.1 Partitioner

Partitioner 接口定义了如何根据给定的分区规则进行创建作业步执行分区的上下文。每个分区的上下文需要根据对应的分区规则来计算当前分区的处理情况。Partitioner 接口定义参见代码清单 11-18。

代码清单 11-18 Partitioner 接口定义

```

1. public interface Partitioner {
2.     Map<String, ExecutionContext> partition(int gridSize);
3. }

```

其中，2 行：操作 partition 根据给定的 gridSize 大小进行执行上下文的划分。

11.5.1.2 StepExecutionSplitter

StepExecutionSplitter 接口定义了如何根据给定的分区规则进行创建作业步执行分区的执行器。StepExecutionSplitter 接口定义参见代码清单 11-19。

代码清单 11-19 StepExecutionSplitter 接口定义

```

1. public interface StepExecutionSplitter {
2.     String getStepName();
3.     Set<StepExecution> split(StepExecution stepExecution, int gridSize)
4.         throws JobExecutionException;
4. }

```

其中，2 行：操作 getStepName() 获取当前定义的分区作业步的名称。

3 行：操作 split() 根据给定的分区规则为每个分区生成对应的分区执行器。

11.5.1.3 PartitionHandler

PartitionHandler 接口定义了分区处理的逻辑;根据给定的 StepExecutionSplitter 进行分区，并执行，最后将执行的结果进行收集，最终反馈到前端。PartitionHandler 接口定义参见代码清单 11-20。

代码清单 11-20 PartitionHandler 接口定义

```
1. public interface PartitionHandler {  
2.     Collection<StepExecution> handle(StepExecutionSplitter stepSplitter,  
3.                                         StepExecution stepExecution) throws Exception;  
4. }
```

其中，2 行：操作 handle() 根据给定的 StepExecutionSplitter 进行分区并执行，最后将执行的结果进行收集，最终反馈到前端。

11.5.2 基本配置

一个典型的分区 Job 配置声明参见代码清单 11-21。

代码清单 11-21 配置典型的分区 Job 示例

```
1. <job id="partitionJob">  
2.     <step id="partitionStep">  
3.         <partition step="partitionReadWriteStep" partitioner="partitioner">  
4.             <handler grid-size="2" task-executor="taskExecutor"/>  
5.         </partition>  
6.     </step>  
7. </job>  
8.  
9. <step id="partitionReadWriteStep">  
10.    <tasklet>  
11.        <chunk reader="flatFileItemReader" writer="jdbcItemWriter"  
12.            processor="creditBillProcessor" commit-interval="2" />  
13.    </tasklet>  
14. </step>  
15.  
16. <bean:bean id="partitioner"  
17.     class="org.springframework.batch.core.partition.support.  
           MultiResourcePartitioner">  
18.     <bean:property name="keyName" value="fileName"/>  
19.     <bean:property name="resources" value="classpath:/ch11/data/*.csv"/>  
20. </bean:bean>
```

在配置分区 Step 之前，我们一起看一下分区 Step 的主要属性定义和元素定义。

图 11-10 展示了分区 Step 属性的 Schema 的定义。

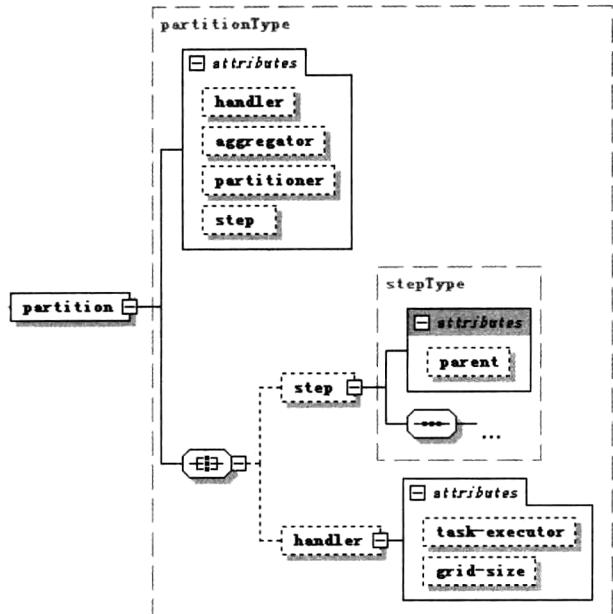


图 11-10 属性 partition 的 Schema 定义

分区 Step 属性说明参见表 11-3。

表 11-3 分区 Step 属性说明

属性	说 明	默 认 值
step	属性 step 用于指定分区 Step 的名字	
partitioner	属性 partitioner 用于指定当前使用的分区逻辑，需要实现接口 Partitioner	
aggregator	属性 aggregator 用于指定需要使用的聚合器，该聚合器的作用是将各个分区执行器执行的结果汇总到主执行器中，用于统计最终的计算结果，该聚合器需要实现接口 StepExecutionAggregator。	默认使用实现类： DefaultStepExecutionAggregator
handler（属性）	属性 handler 用于指定分区执行器，需要实现接口 PartitionHandler。	
handler（子元素）	用于定义默认的实现： TaskExecutorPartitionHandler	
task-executor	声明使用的线程池	
grid-size	声明分区的 HashMap 的初始值大小	6

接下来的章节，我们给出对文件进行分区和对数据库进行分区的示例，读者可以根据这两个例子扩展出其他可能的分区业务处理。

11.5.3 文件分区

Spring Batch 框架提供了对文件分区的支持，实现类 org.springframework.batch.core.partition.support.MultiResourcePartitioner 提供了对文件分区的默认支持，根据文件名将不同的文件处理进行分区，提升处理的速度和效率，适合有大量小文件需要处理的场景。图 11-11 给出了一个多文件的示例，信用卡每月的账单，本章节按照此例子给出如何配置多文件的分区实现。



图 11-11 多文件的示例

由于文件数目较多，我们针对文件进行分区，然后将文件的内容写入 DB 中。本节示例的处理逻辑关系参见图 11-12。

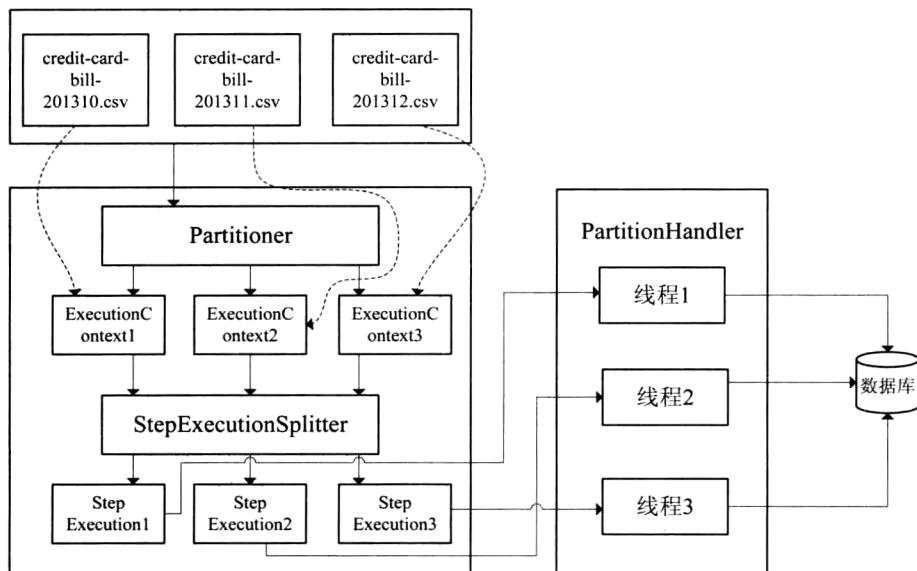


图 11-12 多文件处理逻辑关系图

接下来我们展示如何配置基于文件的分区处理。

配置分区

首先我们定义分区 Step，具体参见代码清单 11-22 配置。

完整配置文件参见：/ch11/job/job-partition-file.xml。

代码清单 11-22 配置分区 Step

```
1. <job id="partitionJob">
2.   <step id="partitionStep">
3.     <partition step="partitionReadWriteStep"
4.       partitioner="filePartitioner">
5.       <handler grid-size="2" task-executor="taskExecutor"/>
6.     </partition>
7.   </step>
8. </job>
9. <step id="partitionReadWriteStep">
10.  <tasklet>
11.    <chunk reader="flatFileItemReader" writer="jdbcItemWriter"
12.      processor="creditBillProcessor" commit-interval="2" />
13.    <listeners>
14.      <listener ref="partitionItemReadListener"></listener>
15.    </listeners>
16.  </tasklet>
17. </step>
```

其中，3~5 行：定义了分区作业，属性 step 声明了引用的作业步的名称，属性 partitioner 引用指定的分区规则，子元素 handler 用于指定使用的本地处理的线程池。

9~17 行：具体的作业操作步定义，本例中使用基于文件的读，然后通过 jdbc 的方式写入数据库中。

定义分区文件

接下来，我们定义文件分区，将上面的不同文件分配到不同的作业步中，使用 MultiResourcePartitioner 进行分区，意味着每个文件会被分配到一个不同的分区中。如果读者有其他的分区规则，可以通过实现接口 Partitioner 来进行自定义的扩展。我们展示使用 MultiResourcePartitioner 进行默认的分区配置，具体参见代码清单 11-23。

代码清单 11-23 配置分区文件

```
1. <bean:bean id="filePartitioner"
2.   class="org.springframework.batch.core.partition.support.
  MultiResourcePartitioner">
3.   <bean:property name="keyName" value="fileName"/>
4.   <bean:property name="resources" value="classpath:/ch11/data/*.csv"/>
5. </bean:bean>
```

其中，3 行：属性 keyName 用于指定作业步上下文中属性的名字，作用是在不同的作业上下文中可以获取设置的对应属性值，对于 MultiResourcePartitioner，对应的值是文件的全路径的名字，可以在对应的读、写等阶段通过#{stepExecutionContext[fileName]} 的方式获取。

4 行：定义需要分区的文件集，使用 MultiResourcePartitioner 时候，要求根据统一的后缀名。

定义文件读

配置好分区实现后，我们需要在每个分区的作业步中读入不同的文件，进而提高文件处理的效率。具体参见代码清单 11-24。

代码清单 11-24 配置文件读

```
1. <bean:bean id="flatFileItemReader" scope="step"
2.   class="org.springframework.batch.item.file.FlatFileItemReader">
3.   <bean:property name="resource"
4.     value="#{stepExecutionContext[fileName]}"/>
5.   <bean:property name="lineMapper" ref="lineMapper" />
6. </bean:bean>
```

其中，4 行：使用 **filePartitioner** 中定义的属性 `keyName` 获取对应的文件路径作为每个分区作业的文件输入。

定义线程池

分区执行处理器使用本地的线程池来处理不同的分区任务，代码清单 11-25 展示了分区执行器使用的线程池的定义。

代码清单 11-25 配置线程池

```
1. <bean:bean id="taskExecutor"
2.   class="org.springframework.scheduling.concurrent.
   ThreadPoolTaskExecutor">
3.   <bean:property name="corePoolSize" value="5"/>
4.   <bean:property name="maxPoolSize" value="15"/>
5. </bean:bean>
```

其中，3 行：属性 `corePoolSize` 定义线程池的始终处于激活状态线程的数量。

4 行：属性 `maxPoolSize` 定义线程池的最大值。

验证分区任务使用不同线程处理

为了更好地给读者验证不同的任务使用了不同的线程处理，本节使用作业步执行拦截器进行验证。在处理分区的作业步之前通过拦截器打印当前的线程名称。

`PartitionStepExecutionListener` 实现接口 `StepExecutionListener`，在对应的 `beforeStep` 操作中打印当前线程名称。具体的实现代码参见代码清单 11-26。

完成代码参见 `com.juxtapose.example.ch11.partition.PartitionStepExecutionListener`。

代码清单 11-26 PartitionStepExecutionListener 类定义

```
1. public class PartitionStepExecutionListener implements
   StepExecutionListener {
2.   @Override
3.   public void beforeStep(StepExecution stepExecution) {
4.     System.out.println("ThreadName=" + Thread.currentThread().
```

```

        getName() + " "
5.          + "StepName=" + stepExecution.getStepName() + " "
6.          + "FileName="
7.          + stepExecution.getExecutionContext().getString("fileName"));
8.      }
9.
10.     @Override
11.     public ExitStatus afterStep(StepExecution stepExecution) {
12.         return null;
13.     }
14. }

```

其中，4~7 行：打印出当前作业步的线程名，作业步的名称，对应处理的文件名。

使用代码清单 11-27 执行定义的 partitionJob。

完整代码参见：test.com.juxtapose.example.ch11.JobLaunchPartitionFile。

代码清单 11-27 执行 partitionJob

```

1. public static void main(String[] args) {
2.     JobLaunchBase.executeJob("ch11/job/job-partition-file.xml",
3.                             "partitionJob",
4.                             new JobParametersBuilder().addDate("date", new Date()));

```

代码清单 11-28 给出了执行的结果，可以看出不同的文件由不同的线程来处理，并且被分配到不同的分区的作业步中执行。

代码清单 11-28 执行 partitionJob 控台输出

```

1. ThreadName=taskExecutor-2; StepName=partitionReadWriteStep:partition0;
   FileName=file:/...../credit-card-bill-201310.csv
2. ThreadName=taskExecutor-3; StepName=partitionReadWriteStep:partition2;
   FileName=file:/...../credit-card-bill-201312.csv
3. ThreadName=taskExecutor-1; StepName=partitionReadWriteStep:partition1;
   FileName=file:/...../credit-card-bill-201311.csv

```

从控台的输出，我们可以抽取下面的关键信息：

taskExecutor-2--> partitionReadWriteStep:partition0--> credit-card-bill-201310.csv

taskExecutor-3--> partitionReadWriteStep:partition2--> credit-card-bill-201312.csv

taskExecutor-1--> partitionReadWriteStep:partition1--> credit-card-bill-201311.csv

每个不同的线程对文件进行了分区处理。

至此，我们完成了文件分区的处理。有兴趣的读者可以自行阅读文件分区默认实现类 org.springframework.batch.core.partition.support.MultiResourcePartitioner，实现非常简单；根据此类的实现，我们可以轻松地模拟出其他的分区实现方式。下一节我们将带领读者通过 Partitioner 来实现数据库的分区处理。

11.5.4 数据库分区

本节通过实现接口 Partitioner 进行数据分区，我们将对读取数据库表进行分区处理，交给不同的作业步进行处理。本节对数据表 t_credit 读取，然后写入表 t_destcredit 中。数据表 t_credit 中数据非常庞大，正常的单线程的读/写会消耗大量时间处理操作，我们接下来使用分区的技术将任务进行切分，交由不同的分区 Step 进行处理。

未分区的批处理作业调度情况参见图 11-13。

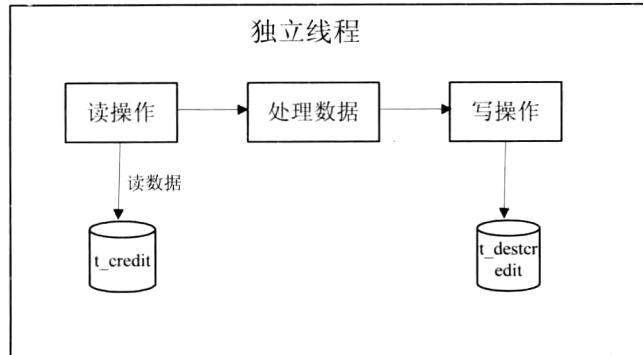


图 11-13 未分区的批处理作业调度

分区后的作业调度情况参见图 11-14，DBPartitioner 将表 t_credit 分割为 3 个作业步处理，每个作业步由独立的线程进行处理，提高了处理的效率。

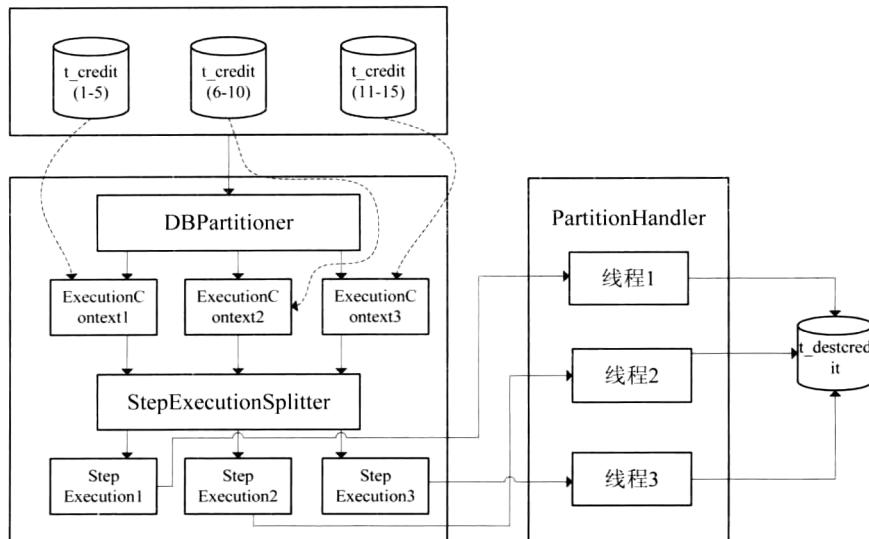


图 11-14 分区后批处理作业调度

实现 DBPartitioner

DBpartition 实现接口 Partitioner(参见代码清单 11-29)，其主要操作 partition(int gridSize) 需要完成的任务非常简单，只需要根据给定的分区 gridSize 的大小，对数据进行切片，并为每个分区的切片生成对应的执行上下文 ExecutionContext。在本例中，分区上下文 ExecutionContext 中存放数据片段的开始值_minRecord 和数据片段的结束值_maxRecord。在后续每个分区片段执行的过程中可以从对应的分区上下文中获取数据分段的开始和结束值 _minRecord 与 _maxRecord。

可以通过 #{stepExecutionContext[_minRecord]} 获取数据片段的开始值，通过 #{stepExecutionContext[_maxRecord]} 获取数据片段的最大值。

完整代码参见：com.juxtapose.example.ch11.partition.db.DBpartition。

代码清单 11-29 DBpartition 类定义

```
1. public class DBpartition implements Partitioner {  
2.     private static final String _MINRECORD = "_minRecord";  
3.     private static final String _MAXRECORD = "_maxRecord";  
4.     private static final String MIN_SELECT_PATTERN = "select min({0}) from {1}";  
5.     private static final String MAX_SELECT_PATTERN = "select max({0}) from {1}";  
6.     private JdbcTemplate jdbcTemplate ;  
7.     private DataSource dataSource;  
8.     private String table ;  
9.     private String column;  
10.  
11.    public Map<String, ExecutionContext> partition(int gridSize) {  
12.        validateAndInit();  
13.        Map<String, ExecutionContext> resultMap =  
14.            new HashMap<String, ExecutionContext>();  
15.        int min = jdbcTemplate.queryForInt(MessageFormat.  
16.            format(MIN_SELECT_PATTERN, new Object[]{column,table}));  
17.        int max = jdbcTemplate.queryForInt(MessageFormat.  
18.            format(MAX_SELECT_PATTERN, new Object[]{column,table}));  
19.        int targetSize = (max-min)/gridSize +1;  
20.        int number=0;  
21.        int start =min;  
22.        int end = start+targetSize-1;  
23.        while(start <= max){  
24.            ExecutionContext context = new ExecutionContext();  
25.            if(end>=max){ end=max;}  
26.            context.putInt(_MINRECORD, start);  
27.            context.putInt(_MAXRECORD, end);  
28.            start+=targetSize;  
29.            end+=targetSize;  
30.            resultMap.put("partition"+(number++), context);  
31.        }  
32.    }  
33.}
```

```
28.         }
29.     return resultMap;
30.   }
31. }
```

其中，14 行：根据主键计算最小值。

15 行：根据主键计算最大值。

20~28 行：根据给定的 gridSize 大小将数据进行分区，根据对应的主键进行平均划分。

23~24 行：将 _minRecord 与 _maxRecord 放入作业步上下文中。

配置分区

首先我们定义分区 Step，具体参见代码清单 11-30 配置。

完整配置文件参见：/ch11/job/job-partition-db.xml。

代码清单 11-30 配置分区

```
1. <job id="partitionJob" restartable="true">
2.   <step id="partitionStep">
3.     <partition step="partitionReadWriteDB" partitioner="partitionerDB">
4.       <handler grid-size="3" task-executor="taskExecutor"/>
5.     </partition>
6.   </step>
7. </job>
8.
9. <step id="partitionReadWriteDB">
10.   <tasklet>
11.     <chunk reader="jdbcItemPageReader" writer="jdbcItemWriter"
12.       processor="creditBillProcessor" commit-interval="2"/>
13.     <listeners>
14.       <listener ref="partitionItemReadListener"></listener>
15.     </listeners>
16.   </tasklet>
17. </step>
```

其中，3~5 行：定义了分区作业，属性 step 声明了引用的作业步的名称，属性 partitioner 引用指定的分区规则，子元素 handler 用于指定使用的本地处理的线程池。

9~17 行：具体的作业操作步定义，本例中使用基于数据库的读，然后通过 jdbc 的方式写入数据库中。

定义分区数据文件

接下来，我们定义文件分区，将上面的数据片段分配到不同的作业步中，使用 DBPartitioner 进行分区，意味着每个数据片段会被分配到一个不同的分区中。具体配置参见代码清单 11-31。

代码清单 11-31 配置分区数据文件

```
1. <!-- db 数据切分 -->
2. <bean:bean id="partitionerDB"
3.   class="com.juxtapose.example.ch11.partition.db.DBpartition">
4.   <bean:property name="table" value="t_credit"/>
5.   <bean:property name="column" value="ID"/>
6.   <bean:property name="dataSource" ref="dataSource"/>
7. </bean:bean>
```

其中，4 行：属性 table 指定分区的表，本例使用 t_credit。

5 行：属性 column 指定分区使用的主键字段，本例使用 ID 字段。

6 行：属性 dataSource 指定使用的数据源。

定义数据库读

配置好分区实现后，我们需要在每个分区的作业步中读入不同的数据片段，进而提高数据处理的效率。具体配置数据库参见代码清单 11-32。

代码清单 11-32 配置数据库读

```
1. <!-- 从 db 分页读数据 -->
2. <bean:bean id="jdbcItemPageReader" scope="step"
3.   class="org.springframework.batch.item.database.JdbcPaging
        ItemReader">
4.   <bean:property name="dataSource" ref="dataSource"/>
5.   <bean:property name="queryProvider" ref="refQueryProvider" />
6.   <bean:property name="pageSize" value="2"/>
7.   <bean:property name="rowMapper" ref="custCreditRowMapper"/>
8. </bean:bean>
9.
10. <bean:bean id="refQueryProvider" scope="step"
11.   class="org.springframework.batch.item.database.support
        .SqlPagingQueryProviderFactoryBean">
12.   <bean:property name="dataSource" ref="dataSource"/>
13.   <bean:property name="selectClause"
14.     value="select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS"/>
15.   <bean:property name="fromClause" value="from t_credit"/>
16.   <bean:property name="whereClause"
17.     value="where ID between #{stepExecutionContext[_minRecord]}
        and #{stepExecutionContext[_maxRecord]}"/>
18.   <bean:property name="sortKey" value="ID"/>
19. </bean:bean>
```

其中，15 行：为读操作指定分区的开始标志 **_minRecord** 和结束标志 **_maxRecord**。

定义线程池

分区执行处理器使用本地的线程池来处理不同的分区任务，代码清单 11-33 展示了分区执行器使用的线程池的定义。

代码清单 11-33 配置线程池

```
1. <bean:bean id="taskExecutor"
2.   class="org.springframework.scheduling.concurrent.
3.   ThreadPoolTaskExecutor">
4.   <bean:property name="corePoolSize" value="5"/>
5.   <bean:property name="maxPoolSize" value="15"/>
6. </bean:bean>
```

其中，3 行：属性 corePoolSize 定义线程池的始终处于激活状态线程的数量。

4 行：属性 maxPoolSize 定义线程池的最大值。

验证分区任务使用不同线程处理

为了更好地给读者验证不同的任务使用了不同的线程处理，本节使用作业步执行拦截器进行验证。在处理分区的作业步之前通过拦截器打印当前的线程名称。

PartitionStepExecutionListener 实现接口 StepExecutionListener，在对应的 beforeStep 操作中打印当前线程名称。具体的实现代码参见代码清单 11-34。

完成代码参见 com.juxtapose.example.ch11.partition.db.PartitionStepExecutionListener。

代码清单 11-34 PartitionStepExecutionListener 类定义

```
1. public class PartitionStepExecutionListener implements
2. StepExecutionListener {
3.   @Override
4.   public void beforeStep(StepExecution stepExecution) {
5.     System.out.println("ThreadName=" + Thread.currentThread().
6.     getName() + ";" +
7.     "StepName=" + stepExecution.getStepName() + ";");
8.   }
9.   @Override
10.  public ExitStatus afterStep(StepExecution stepExecution) {
11.    return null;
12.  }
}
```

其中，4~7 行：打印出当前作业步的线程名，作业步的名称，对应处理的文件名。

使用代码清单 11-35 执行定义的 partitionJob。

完整代码参见：test.com.juxtapose.example.ch11.JobLaunchPartitionDB。

代码清单 11-35 执行 partitionJob

```
1. public static void main(String[] args) {  
2.     JobLaunchBase.executeJob("ch11/job/job-partition-db.xml",  
3.         "partitionJob",  
4.         new JobParametersBuilder().addDate("date", new Date()));  
}
```

代码清单 11-36 给出了执行的结果，可以看出表中不同的数据由不同的线程来处理，并且被分配到不同的分区的作业步中执行。

代码清单 11-36 执行 partitionJob 控台输出

```
1. ThreadName=taskExecutor-3; StepName=partitionReadWriteDB:partition1;  
2. ThreadName=taskExecutor-2; StepName=partitionReadWriteDB:partition2;  
3. ThreadName=taskExecutor-1; StepName=partitionReadWriteDB:partition0;
```

从控制台的输出，我们可以抽取下面的关键信息：

taskExecutor-1--> partitionReadWriteDB:partition0
taskExecutor-2--> partitionReadWriteDB:partition2
taskExecutor-3--> partitionReadWriteDB:partition1

每个不同的线程对数据文件进行了分区处理。

11.5.5 远程分区 Step

前面章节我们介绍了远程 Step 和分区 Step，本节我们将提供一个远程分区 Step 的例子供读者参考。

远程分区 Job 对本地的多文件进行分区读取，然后通过远程的方式发送到远端执行，通过 JDBC 的方式写入数据库中。远程分区的处理示意图参见图 11-15。

通过分区 Partitioner 将多文件进行切分，通过 StepExecutionSplitter 将资源进行分割成不同的作业步执行上下文；MessageChannelPartitionHandler 将作业步执行上下文通过消息的方式发送到请求队列，远程机器的拦截器监听队列消息的到达，收取消息后交给 ChunkProcessor ChunkHandler 处理，最终通过 JDBC 的方式写入数据库的 t_destcredit 表中；处理完成后发送响应消息到响应队列，MessageChannelPartitionHandler 负责收取响应队列的消息，将远端的执行情况进行汇总。

11.5.5.1 配置远程分区 Job

代码清单 11-37 定义了远程分区 Job。

完整配置文件参见：ch11/job/job-partition-remote.xml。

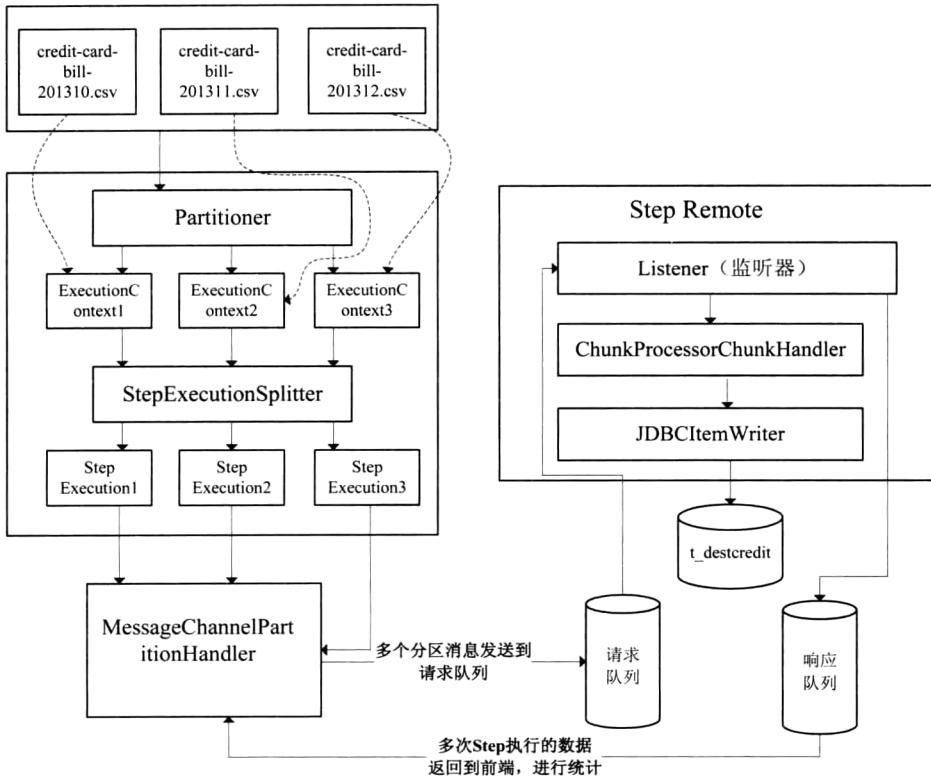


图 11-15 远程分区处理示意图

代码清单 11-37 配置远程分区 partitionRemoteJob

```

1. <job id="partitionRemoteJob">
2.   <step id="partitionRemoteStep">
3.     <partition partitioner="partitioner" handler="partitionHandler" />
4.   </step>
5. </job>

```

其中，3 行：属性 partitioner 定义分区定义，属性 partitionHandler 定义分区后的处理逻辑。

11.5.5.2 配置本地 Step

本地 Step 配置参见代码清单 11-38。

完整配置文件参见：ch11/job/job-partition-remote.xml。

代码清单 11-38 配置本地 Step

```

1. <bean:bean id="partitioner"
2.   class="org.springframework.batch.core.partition.
      support.MultiResourcePartitioner">

```

```

3.      <bean:property name="keyName" value="fileName"/>
4.      <bean:property name="resources" value="classpath:/ch11/data/*.csv"/>
5.  </bean:bean>
6.
7.  <bean:bean id="partitionHandler"
8.    class="org.springframework.batch.integration.
       partition.MessageChannelPartitionHandler">
9.    <bean:property name="messagingOperations">
10.     <bean:bean class="org.springframework.integration.
        core.MessagingTemplate">
11.       <bean:property name="defaultChannel" ref="requests" />
12.       <bean:property name="receiveTimeout" value="30000" />
13.     </bean:bean>
14.   </bean:property>
15.   <bean:property name="replyChannel" ref="replies"/>
16.   <bean:property name="stepName" value="remoteStep" />
17.   <bean:property name="gridSize" value="2" />
18. </bean:bean>
19.
20. <step id="remoteStep">
21.   <tasklet>
22.     <chunk reader="flatFileItemReader" writer="jdbcItemWriter"
           commit-interval="10"/>
23.     <listeners>
24.       <listener ref="partitionItemReadListener"></listener>
25.     </listeners>
26.   </tasklet>
27. </step>

```

其中，1~5 行：定义了分区策略，基于文件的分区，/ch11/data/*.csv 下面的所有文件单独划分为一个分区。

7~18 行：定义分区后的处理逻辑，messagingOperations 定义了消息发送接口，默认的发送消息到 requests 队列。

15 行：属性 replyChannel 定义消息返回的队列为 replies。

16 行：定义远程执行的作业步，定义真正的对文件读/写的逻辑。

20~27 行：定义远端执行的作业步 remoteStep，基于文件的读，基于 JDBC 的数据库写。

23~25 行：定义作业步 remoteStep 的拦截器，主要用于打印执行的线程名与对应处理的分区文件；完整的拦截器代码参见：com.juxtapose.example.ch11.partition.PartitionStepExecution Listener。

11.5.3 配置远程 Step

远程 Step 本质上是一个队列的监听器与收到消息后的处理逻辑。具体配置参见代码

清单 11-39。

完整配置参见文件：/ch11/job/job-partition-remote.xml。

代码清单 11-39 配置远程 Step

```
1.  <jms:listener-container connection-factory="connectionFactory"
2.    transaction-manager="transactionManager"
3.    acknowledge="transacted" concurrency="10">
4.      <jms:listener destination="requests" response-destination="replies"
5.        ref="stepExecutionRequestHandler" method="handle" />
6.    </jms:listener-container>
7.
8.    <bean:bean id="stepExecutionRequestHandler"
9.      class="org.springframework.batch.integration.
           partition.StepExecutionRequestHandler">
10.      <bean:property name="jobExplorer" ref="jobExplorer"/>
11.      <bean:property name="stepLocator" ref="stepLocator"/>
12.    </bean:bean>
13.    <bean:bean id="stepLocator"
14.      class="org.springframework.batch.integration.
           partition.BeanFactoryStepLocator" />
```

其中，1~6 行：配置请求队列的监听器与处理逻辑，处理逻辑调用 stepExecutionRequest Handler 的 handle 操作；该操作接受 chunk 请求，并负责处理逻辑。

4 行：定义监听的队列为 requests，监听到消息后处理逻辑为 stepExecutionRequestHandler 的 handle 操作，处理后的消息发送到 replies 队列。

8~12 行：定义接受消息后的处理逻辑，通过给定的属性 jobExplorer 和 stepLocator 在 jobRepository 查找到指定的作业步；通过 jobExplorer 接口，可以进行作业状态查询类，返回返回作业状态的复杂对象，如 Job Execution、Job Instance、StepExecution 等。

13~14 行：stepLocator 定义了作业步查找类，通过 stepName 可以从上下文中找到对应的 Step 对象。

11.5.5.4 运行远程分区 Job

在能够顺利执行远程分区 Job 之前，需要定义消息队列及配置 AMQ 服务器，读者可以参考 11.4.2 节中的配置消息队列、配置 AMQ 服务器章节。完整的配置文件参见：/ch11/job/job-partition-remote.xml。

使用代码清单 11-40 执行定义的 partitionRemoteJob。

完整代码参见：test.com.juxtapose.example.ch11.JobLaunchPartitionRemote。

代码清单 11-40 执行 partitionRemoteJob

```
1.  JobLaunchBase.executeJob("ch11/job/job-partition-remote.xml",
2.    "partitionRemoteJob",
2.    new JobParametersBuilder().addDate("date", new Date()));
```

验证点 1：数据被正确地写入数据库。

测试用例执行完毕后，可以看到库表 t_destcredit 中的数据被远程写入。

验证点 2：文件分区及被不同的线程处理。

代码清单 11-41 给出了执行的结果，可以看出不同的文件由不同的线程来处理，并且被分配到不同的分区的作业步中执行。

代码清单 11-41 执行 partitionRemoteJob 控台输出

```
1. ThreadName=org.springframework.jms.listener.DefaultMessage
   ListenerContainer#0-2; StepName=partitionRemoteStep:partition0;
   FileName= file:/...../data/credit-card-bill-201310.csv
2. ThreadName=org.springframework.jms.listener.DefaultMessageListener
   Container#0-1; StepName=partitionRemoteStep:partition2; FileName=file:
   /...../ch11/data/credit-card-bill-201312.csv
3. ThreadName=org.springframework.jms.listener.DefaultMessageListener
   Container#0-3; StepName=partitionRemoteStep:partition1; FileName=file:
   /...../ch11/data/credit-card-bill-201311.csv
```

验证点 3：查看数据库表 batch_step_execution 中的作业步信息，参见图 11-16、图 11-17。

	STEP_EXECUTION_ID	VERSION	STEP_NAME	STATUS
1		811	2 partitionRemoteStep	COMPLETED
2		812	3 partitionRemoteStep:partition2	COMPLETED
3		813	3 partitionRemoteStep:partition1	COMPLETED
4		814	3 partitionRemoteStep:partition0	COMPLETED

图 11-16 数据库表 batch_step_execution 中的作业步信息（一）

COMMIT_COUNT	READ_COUNT	WRITE_COUNT	START_TIME	END_TIME
3	18	18	2014-03-29 21:00:56	2014-03-29 21:00:59
1	6	6	2014-03-29 21:00:57	2014-03-29 21:00:57
1	6	6	2014-03-29 21:00:57	2014-03-29 21:00:57
1	6	6	2014-03-29 21:00:57	2014-03-29 21:00:57

图 11-17 数据库表 batch_step_execution 中的作业步信息（二）

通过上面的数据库信息可以看出，作业步 partitionRemoteStep 被划分为了三个分区，分别是 partitionRemoteStep:partition0、partitionRemoteStep:partition1、partitionRemoteStep:partition2；每个分区的作业步各处理 6 条消息，最后作业步 partitionRemoteStep 成功读/写了 18 条消息，正好是每个分区作业处理消息条数的总和。

后记

随着近年来互联网应用、云计算、移动互联网的发展，大数据的处理已成为当前企业迫切需要面对的问题。数据批量操作、数据分析、数据处理、数据抽取、元数据、数据质量、数据仓库、数据集成等一系列数据的概念如雨后春笋般地涌现，各种批处理的工具、数据集成框架、ETL 工具在开源社区也纷纷涌现。**Spring Batch** 就是在这样背景下出现的一款批处理框架。

Spring Batch 是一款基于 Spring 的企业批处理框架。通过它可以构建出健壮的企业批处理应用。**Spring Batch** 不仅提供了统一的读/写接口、丰富的任务处理方式、灵活的事务管理、并发处理，同时还支持日志、监控、任务重启与跳过等特性，大大简化了批处理应用开发，将开发人员从复杂的任务配置管理过程中解放出来，使他们可以更多地去关注核心的业务处理过程。

2007 年，**Spring Batch** 项目进入开发阶段；2008 年 3 月，**Spring Batch** 发布第一个正式版本 1.0.0；2014 年 10 月推出 **Spring Batch** 2.2.7 版本。2013 年 4 月发布的 JSR-352 标准定义了 Java 平台上的批处理应用程序，为应用开发人员提供了一个开发健壮批处理系统的模型，该模型的核心便借鉴了 **Spring Batch** 开发模式。Reader-Processor-Writer 模式，在这个模型中鼓励开发人员遵循面向块的处理标准。能够在 JSR-352 标准中看到 **Spring Batch** 框架的身影，表明 **Spring Batch** 框架在批处理的架构设计上得到了充分的认可。

但在企业级应用中面对批量数据处理，提供批处理框架仅能满足批处理作业的快速开发、执行能力。企业需要统一的批处理平台来处理复杂的企业批处理应用，批处理平台需要解决作业的统一调度、批处理作业的集中管理和管控和批处理作业的统一监控。试想一下，目前国内一般的银行每日处理的批处理作业达到万量级的水平，国内互联网每日的作业量在百万量级的水平，如果没有友好的统一作业调度平台进行支撑是一件多么可怕的事情。企业级的批处理平台需要在 **Spring Batch** 批处理框架的基础上集成调度框架，通过调度框架可以将任务按照企业的需求进行任务的定期执行；丰富目前 **Spring Batch Admin**（**Spring Batch** 的管理监控平台，目前能力还比较薄弱）框架，提供对 Job 的统一管理功能，增强 Job 作业的监控、预警等能力；通过与企业的组织机构、权限管理、认证系统进行合理的集成，增强平台对 Job 作业的权限控制、安全管理能力。

Spring Batch 框架针对批处理应用的支持在框架级别已经做到了完美，故本书就是将 **Spring Batch** 框架带给国内的读者，希望有更多的开源产品或者企业批处理平台能够基于 **Spring Batch** 进行发展，进一步带动 **Spring Batch** 的生态圈。

编者于上海
2014 年 12 月