

```
12.    <bean:bean id="sysoutAnnotationListener"
13.                  class="com.juxtapose.example.ch05.listener.SystemOut" />
```

其中，7行：定义 step 的拦截器，sysoutListener 的实现参见示例项目代码 com.juxtapose.example.ch04.listener.SystemOutJobExecutionListener。

8行：定义 step 的拦截器，sysoutAnnotationListener 的实现参见示例项目代码 com.juxtapose.example.ch04.listener.SystemOut。

系统实现

Spring Batch 框架默认提供了 StepExecutionListener 的实现，分别完成不同功能，具体参见表 5-3 描述。

表 5-3 StepExecutionListener 默认实现

StepExecutionListener 默认实现	功能说明
CompositeStepExecutionListener	拦截器组合模式，支持一组拦截器调用
StepExecutionListenerSupport	StepExecutionListener 空实现，可以直接继承，仅复写关心的操作
StepListenerSupport	同时实现 StepExecutionListener、ChunkListener、ItemReadListener、ItemProcessListener、ItemWriteListener、SkipListener 接口的空实现，可以直接继承，仅复写关心的操作
MulticasterBatchListener	同时实现 StepExecutionListener、ChunkListener、ItemReadListener、ItemProcessListener、ItemWriteListener、SkipListener 接口的组合模式，支持一组拦截器调用

拦截器异常

拦截器方法如果抛出异常会影响 Step 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 Step 执行的状态为"FAILED"；作业的状态同样为"FAILED"。

执行顺序

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

- (1) sysoutListener 拦截器的 before 方法；
- (2) sysoutAnnotationListener 拦截器的 before 方法；
- (3) sysoutAnnotationListener 拦截器的 after 方法；
- (4) sysoutListener 拦截器的 after 方法。

Annotation

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

- `@BeforeStep`
- `@AfterStep`

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

通过 Annotation 声明的拦截器代码，参见代码清单 5-6。

代码清单 5-6 Annotation 声明 Step 拦截器

```
1. public class SystemOut {
2.     @BeforeStep
3.     public void beforeStep(StepExecution stepExecution) {
4.         .....
5.     }
6.
7.     @AfterStep
8.     public ExitStatus afterStep(StepExecution stepExecution) {
9.         .....
10.    }
11. }
```

merge 属性

假设有这样一个场景，所有的 Step 都希望拦截器 sysoutListener 能够执行，而拦截器 sysoutAnnotationListener 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 5-7。

代码清单 5-7 merge 属性示例

```
1. <step id="abstractParentStep" abstract="true">
2.     <tasklet>.....</tasklet>
3.     <listeners>
4.         <listener ref="sysoutListener"></listener>
5.     </listeners>
6. </step>
7. <job id="subChunkJob">
8.     <step id="subChunkStep" parent="abstractParentStep">
9.         <tasklet>.....</tasklet>
10.        <listeners merge="true">
11.            <listener ref="sysoutAnnotationListener"></listener>
12.        </listeners>
13.    </step>
14. </job>
```

其中,10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。

5.2 配置 Tasklet

tasklet 元素定义任务的具体执行逻辑,执行逻辑可以自定义实现,也可以使用 Spring Batch 的 Chunk 操作,提供了标准的读、处理、写三步操作。通过 tasklet 元素同样可以定义事务、处理线程、启动控制、回滚控制、拦截器等功能。

tasklet 元素的属性 Schema 定义,参见图 5-6。

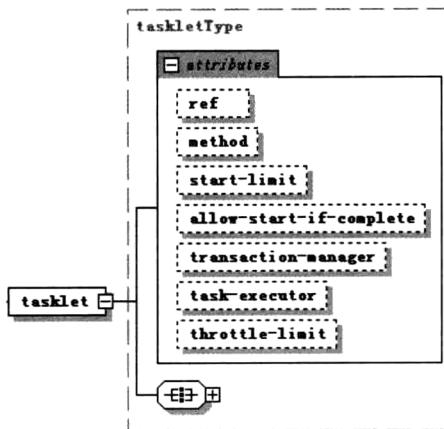


图 5-6 tasklet 属性 Schema 的定义

tasklet 属性说明,参见表 5-4。

表 5-4 tasklet 属性说明

属性	说 明	默 认 值
ref	引用自定义实现 Tasklet 接口的 Bean 当使用自定义的 Tasklet 业务时候,必须使用 ref 来引用,如果使用 Spring Batch 提供的 Chunk 组件,请使用 chunk 元素来定义业务	
method	引用自定义的业务 Bean,需要调用的操作,对操作的要求: 参数需要和 Tasklet.execute 具有相同的参数。 返回值: 可以是 void、Boolean 或者 RepeatStatus	execute
start-limit	Step 能够启动的最大次数,超过最大次数后会抛出异常	Integer.MAX_VALUE
allow-start-if-complete	是否允许完成(状态为"COMPLETED")的 Step 重新启动	false
transaction-manager	tasklet 配置的事务管理器,控制业务的处理操作	

续表

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

tasklet 元素的子元素 Schema 定义，参见图 5-7。

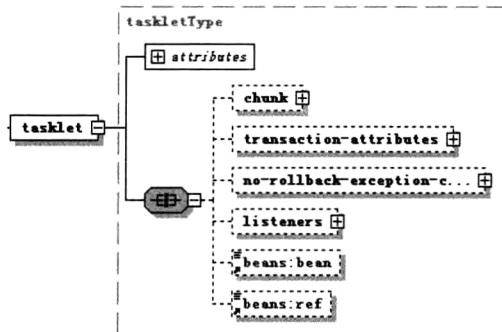


图 5-7 tasklet 子元素 Schema 的定义

tasklet 子元素说明，参见表 5-5。

表 5-5 tasklet 子元素说明

属性	说 明	默 认 值
chunk	定义面向批的业务操作，使用 chunk 可以复用 Spring Batch 提供的基础设施	
transaction-attributes	定义任务具体的事务属性，例如隔离级别、事务传播方式、超时时间等	
no-rollback-exception-classes	定义不会触发事务回滚的异常。 通常情况下，作业处理过程中发生的任何异常都会导致作业事务的回顾，Spring Batch 框架提供指定异常不影响事务的能力	
listeners	定义具体任务级别的拦截器	

5.2.1 重启 Step

批处理作业框架需要支持任务重新启动，批处理作业处理数据发生错误时，在数据修复后需要能够将当前的任务实例执行完毕。在基本概念章节我们已经介绍了 Job Instance、JobParameters、ExecutionContext 之间的关系。

Spring Batch 框架支持状态为非"COMPLETED"的 Job 实例重新启动，Job 实例重启的时候，会从当前失败的 Step 重新开始执行，同时可以通过 start-limit 属性控制任务启动的次数。

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。代码清单 5-8 展示了作业步 startLimitStep 只能启动一次。

代码清单 5-8 配置 Step 启动次数

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

定义 startLimitStep 仅能启动一次，第二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。

重启已完成的任务

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。仅在某些特殊场景下，已经完成的 Step 在任务重启的时候需要再次执行，这可以通过属性 allow-start-if-complete 来设置。代码清单 5-9 展示了作业步重启的配置。

代码清单 5-9 配置 Step 支持重启

```
1. <step id="allowStartStep">
2.   <tasklet allow-start-if-complete="true">
3.     <chunk reader="reader" processor="processor" writer="writer"/>
4.   </tasklet>
5. </step>
```

其中，2 行：通过设置属性 allow-start-if-complete 值为 true，表示已经完成的 Step 可以被再次启动。

5.2.2 事务

Spring Batch 框架提供了事务能力保障 Job 可靠地执行，能够将 Job 的 read、process 和 write 三者有效地控制在一起，保证操作的完整性；Job 执行期间的元数据状态的持久化同样依赖事务的保证。在 Job 执行期间，可以配置事务管理器、事务的基本属性（包括隔离级别、传播方式、事务超时等信息）。

事务管理器

为了使用事务管理器，需要在配置文件中声明标准 Spring 方式的事务管理器，参见代码清单 5-10 中配置的事务管理器。

代码清单 5-10 配置事务管理器

```
1. <bean:bean id="transactionManager"
2.     class="org.springframework.jdbc.datasource.
    DataSourceTransactionManager">
3.     <bean:property name="dataSource" ref="dataSource" />
4. </bean:bean>
```

其中，2行：指定使用的事务管理器类。

3行：指定使用的数据源。

事务管理器需要提供指定的数据源对象，在对此数据源的任何操作将会得到事务的保障。

事务管理器确定以后，需要为 tasklet 定义使用的事务管理器，参见代码清单 5-11。

代码清单 5-11 为 tasklet 配置事务管理器

```
1. <step id="chunkStep">
2.     <tasklet transaction-manager="transactionManager" start-limit="1">
3.         <chunk reader="reader" processor="processor" writer="writer"
        commit-interval="5" />
4.     </tasklet>
5. </step>
```

示例配置中使用 transactionManager 对象，保障 chunkStep 作业的操作在一个事务内完成。

事务属性

在事务管理器中可以指定事务的属性，例如事务的隔离级别、事务的传播方式、事务超时时间等，这些事务属性同样可以在 tasklet 中配置，详细定义任务的事务属性定义。

事务的隔离级别（**isolation**）参见表 5-6 描述。

表 5-6 事务的隔离级别（**isolation**）

隔离级别	说 明
SERIALIZABLE	最严格的级别，事务串行执行，资源消耗最大
REPEATABLE_READ	保证了一个事务不会修改已经由另一个事务读取但未提交（回滚）的数据。避免了“脏读取”和“不可重复读取”的情况，但是带来了更多的性能损失
READ_COMMITTED	默认事务等级，保证了一个事务不会读取另一个并行事务已修改但未提交的数据，避免了“脏读取”。该级别适用于大多数系统
READ_UNCOMMITTED	保证了读取过程中不会读取非法数据

事务传播方式（**propagation**）参见表 5-7 描述。

表 5-7 事务传播方式（**propagation**）

传播方式	说 明
REQUIRED	支持当前事务，如果当前没有事务，就新建一个事务
SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行

续表

传播方式	说 明
MANDATORY	支持当前事务，如果当前没有事务，就抛出异常
REQUIRES_NEW	新建事务，如果当前存在事务，则把当前事务挂起
NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
NEVER	以非事务方式执行，如果当前存在事务，则抛出异常

代码清单 5-12 给出了如何配置事务的基本属性。

代码清单 5-12 配置事务基本属性

```

1. <step id="chunkStep">
2.   <tasklet transaction-manager="transactionManager" start-limit="1">
3.     <chunk reader="reader" processor="processor" writer="writer"
4.       commit-interval="5" />
5.     <transaction-attributes isolation="DEFAULT"
6.       propagation="REQUIRED" timeout="30"/>
5.   </tasklet>
6. </step>

```

其中，4 行：定义事务的隔离级别为 DEFAULT，事务传播方式为 REQUIRED（表示需要事务，如果外面有事务则加入外部事务，如果外部没有事务则新启一个事务），事务超时时间为 30 秒。

5.2.3 事务回滚

通过事务控制可以较好地保证事务任务的执行。在业务处理过程中，包括读、写、处理数据如果发生了异常会导致事务回滚，Spring Batch 框架提供了发生特定异常不触发事务回滚的能力，可以在 tasklet 中通过子元素 no-rollback-exception-classes 来定义特定异常。

配置不触发回滚操作异常的配置，参见代码清单 5-13。

代码清单 5-13 配置不触发回滚操作异常

```

1. <step id="chunkStep">
2.   <tasklet transaction-manager="transactionManager" start-limit="1">
3.     <chunk reader="reader" processor="processor" writer="writer"
4.       commit-interval="5" />
5.     <transaction-attributes isolation="DEFAULT"
6.       propagation="REQUIRED" timeout="30"/>
5.     <no-rollback-exception-classes>
6.       <include class="org.springframework.batch.item.validator.
7.                 ValidationException"/>
7.     </no-rollback-exception-classes>
8.   </tasklet>
9. </step>

```

其中，5~7 行：配置发生 ValidationException 异常或者其子类异常时，不会触发事务的回滚操作；其他任何类型异常都会导致事务的回滚操作。

5.2.4 多线程 Step

Job 执行默认情况使用单个线程完成任务的执行。Spring Batch 框架支持为 Step 配置多个线程，即可以使用多个线程并行执行一个 Step，可以提高 Step 的处理速度。使用 tasklet 的属性 task-executor 为 Step 定义多线程。示例代码参见代码清单 5-14。

代码清单 5-14 配置多线程 Step

```
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="5"/>
6.      </tasklet>
7.    </step>
8.  </job>
9.
10. <bean:bean id="taskExecutor"
11.   class="org.springframework.scheduling.concurrent.
12.     ThreadPoolTaskExecutor">
13.   <bean:property name="corePoolSize" value="5"/>
14.   <bean:property name="maxPoolSize" value="15"/>
15. </bean:bean>
```

其中，3 行：通过属性 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。

9~13 行：定义线程池，默认有 5 个线程，最大为 15 个线程。

5.2.5 自定义 Tasklet

元素 tasklet 中可以配置面向 Chunk 的任务，或者是开发者实现自定义的 Tasklet。两种都可以完成业务逻辑的配置。面向 Chunk 的操作在下一节详细描述。自定义的 Tasklet 需要接口 org.springframework.batch.core.step.tasklet.Tasklet 来实现。

接口 Tasklet 声明，参见代码清单 5-15。

代码清单 5-15 接口 Tasklet 声明

```
1.  public interface Tasklet {
2.    RepeatStatus execute(StepContribution contribution, ChunkContext
3.      chunkContext) throws Exception;
4. }
```

`execute()`是接口 Tasklet 唯一必须实现的方法，该方法完成对业务的处理。通过自定义的 Tasklet 可以复用开发者已有的企业服务。

可以通过 tasklet 的属性 `ref` 和属性 `method` 来使用自定义的 Tasklet。配置示例参见代码清单 5-16。

代码清单 5-16 配置 tasklet 的 ref

```
1.      <job id="custTaskletJob">
2.          <step id="custTaskletStep">
3.              <tasklet ref="helloWorldTasklet">
4.                  </tasklet>
5.              </step>
6.          </job>
7.      <bean:bean id="helloWorldTasklet" class="com.juxtapose.example.ch05.
HelloWorldTasklet" />
```

其中，3 行：通过 `ref` 属性引用自定义的业务操作。

7 行：定义自己实现 Tasklet 接口的业务 `HelloWorldTasklet`。

Tasklet 系统实现

Spring Batch 框架默认提供了 Tasklet 的实现，分别完成不同功能，在实际应用开发过程中可以直接使用系统默认的实现，也可以重新实现自己的 Tasklet。

系统提供的默认 Tasklet 实现，参见表 5-8。

表 5-8 Tasklet 默认实现

Tasklet 默认实现	功能说明
CallableTaskletAdapter	Callable 接口适配器 <code>org.springframework.batch.core.step.tasklet.callableTaskletAdapter</code>
ChunkOrientedTasklet	面向批的任务处理，用于 Chunk 的处理操作 <code>org.springframework.batch.core.step.item.ChunkOrientedTasklet<I></code>
MethodInvokingTaskletAdapter	用于适配已有服务，通过代理的方式调用已经存在的服务 <code>org.springframework.batch.core.step.tasklet.MethodInvokingTaskletAdapter</code>
SystemCommandTasklet	系统命令任务类，可以调用执行的命令 <code>org.springframework.batch.core.step.tasklet.SystemCommandTasklet</code>

接下来我们以 `MethodInvokingTaskletAdapter` 为例子，展示如何使用系统默认提供的 Tasklet。框架提供的其他默认 Tasklet，读者可以自行深入研究。

使用 `MethodInvokingTaskletAdapter`，通过代理的方式调用已经存在的服务，代码清单 5-17 展示了在 Job 中调用查询 `jobRegistry` 对象的所有作业名的操作。

代码清单 5-17 配置代理服务

```
1.      <job id="taskletAdapterJob">
2.          <step id="taskletAdapterStep">
3.              <tasklet ref="adapter">
4.                  </tasklet>
5.              </step>
6.          </job>
7.
8.          <bean:bean id="adapter"
9.             class="org.springframework.batch.core.step.tasklet.
MethodInvokingTaskletAdapter">
10.             <bean:property name="targetObject" ref="jobRegistry" />
11.             <bean:property name="targetMethod" value="getJobNames" />
12.         </bean:bean>
```

其中，3 行：自定义的 Tasklet 使用 MethodInvokingTaskletAdapter 对象。

8~12 行：定义 adapter，通过 MethodInvokingTaskletAdapter 调用对象 jobRegistry 的 getJobNames 操作。

说明：引用自定义的业务 Bean，需要调用的操作，对操作有如下的要求：

参数需要和 Tasklet.execute 具有相同的参数；

返回值可以是 void、Boolean 或者 RepeatStatus。

5.3 配置 Chunk

Chunk 元素定义面向批的处理操作，Chunk 典型地提供了标准的 read、process、write 三种操作，同时 Spring Batch 框架提供了丰富的读、写的组件，包括对格式化文件的读/写、xml 文件的读/写、数据库的读/写、JMS 消息的读/写等技术组件，可以在配置文件中直接使用。

Chunk 元素提供了功能性以外的能力，包括异常处理、批处理的可靠性、稳定性、异常重入的能力。具体包括事务提交间隔、跳过策略、重试策略、读事务队列、批处理完成策略等。

Chunk 元素的属性 Schema 定义参见图 5-8。

Chunk 属性说明，参见表 5-9。

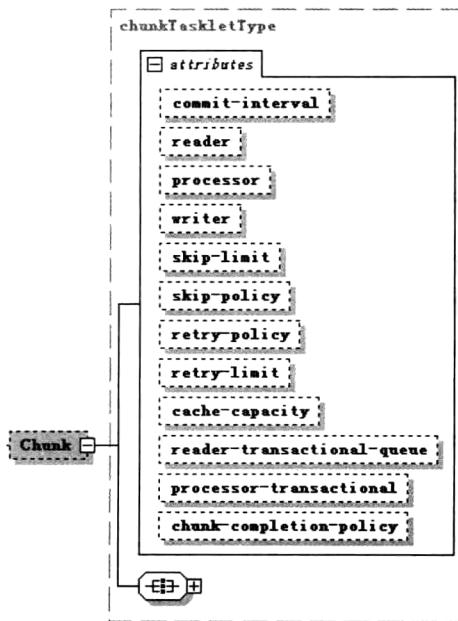


图 5-8 Chunk 属性 Schema 定义

表 5-9 Chunk 属性说明

属性	说 明	默 认 值
commit-interval	提交间隔，读出、处理指定的数据后，通过 writer 批量写入，并提交事务	
reader	读取数据的 Bean 的名字，需要实现接口： org.springframework.batch.item.ItemReader<T>	
processor	批处理中的处理逻辑，需要实现接口： org.springframework.batch.item.ItemProcessor<I, O>	
writer	写数据的 Bean 的名字，需要实现接口： org.springframework.batch.item.ItemWriter<T>	
skip-limit	异常发生时，允许跳过的最大数。 当跳过数目达到后，数据处理（包括读数据、处理数据、写数据）发生的异常会抛出，导致任务 Step 失败	
skip-policy	跳过策略 Bean，需要实现接口： org.springframework.batch.core.step.skip.SkipPolicy	
retry-policy	重试策略 Bean，需要实现接口： org.springframework.batch.retry.RetryPolicy	
retry-limit	任务执行重试的最大次数	
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.RetryCacheCapacityExceededException	4096
reader-transactional-queue	是否从一个事务性的队列读取数据： true 表示从一个事务性的队列中读取数据，一旦发生异常会导致事务回滚，从队列中读取的数据同样会被重新放回到队列中； false 表示从一个没有事务的队列获取数据，一旦发生异常会导致事务回滚，消费掉的数据不会重新放置在队列中	false
processor-transactional	处理数据是否在事务中，true 表示将 processor 处理的结果放在缓存中，当执行重试或者跳过策略时可以看到缓存中处理的数据； false 表示不会将 processor 处理的数据放在缓存中，即 processor 在 chunk 的每一条记录仅会执行一次。 需要注意：如果将 reader-transactional-queue 设置为 true，则 processor-transactional 必须设置为 true	true
chunk-completion-policy	批处理完成策略，需要实现接口： org.springframework.batch.repeat.CompletionPolicy	

Chunk 的子元素 Schema 定义，参见图 5-9。

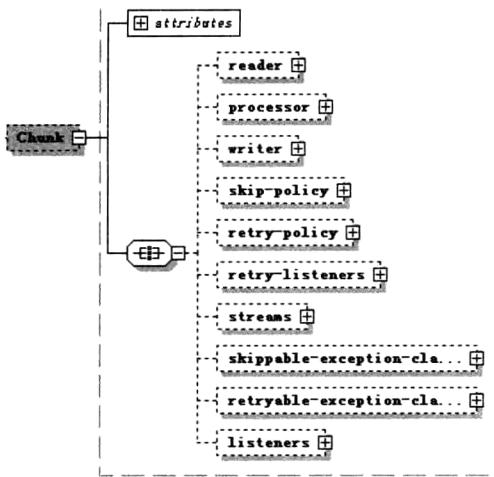


图 5-9 Chunk 子元素 Schema 定义

Chunk 子元素说明，参见表 5-10。

表 5-10 Chunk 子元素说明

属性	说 明	默 认 值
reader	读取数据的 Bean 的定义，需要实现接口： org.springframework.batch.item.ItemReader<T>	
processor	批处理中的处理逻辑的定义，需要实现接口： org.springframework.batch.item.ItemProcessor<I, O>	
writer	写数据的 Bean 的定义，需要实现接口： org.springframework.batch.item.ItemWriter<T>	
skip-policy	跳过策略 Bean 定义，需要实现接口： org.springframework.batch.core.step.skip.SkipPolicy	
retry-policy	重试策略 Bean 定义，需要实现接口： org.springframework.batch.retry.RetryPolicy	
retry-listeners	重试操作监听器，需要实现接口： org.springframework.batch.retry.RetryListener	
streams	定义一组实现 ItemStream 的对象，需要实现接口： org.springframework.batch.item.ItemStream。 Step 执行期间需要知道 reader、processor、writer 定义的实例哪些是实现接口 ItemStream 的（显现接口 ItemStream 的 reader、processor、writer 的对象能够在任务重启的时候从正确的点恢复）。 Spring Batch 框架自动注册实现了接口 ItemStream 的对象，如果在 reader、processor、writer 中用到的对象没有直接实现接口 ItemStream，需要在此处显式注册	

续表

属性	说 明	默 认 值
skippable-exception-classes	定义一组触发跳过的异常	
retryable-exception-classes	定义一组触发重试的异常	
listeners	定义一组 Chunk 的拦截器，需要实现接口： org.springframework.batch.core.ChunkListener	

5.3.1 提交间隔

批处理作业通常针对大数据量进行处理，同时框架需要将作业处理的状态实时地持久化到数据库中，如果读取一条记录就进行写操作或者状态数据的提交，会大量消耗系统资源，导致批处理框架性能度差。在面向批处理 Chunk 的操作中，可以通过属性 commit-interval 设置 read 多少条记录后进行一次提交。通过设置 commit-interval 的间隔值，减少提交频次，降低资源使用率。

面向 Chunk 的操作序列图，参见图 5-10。

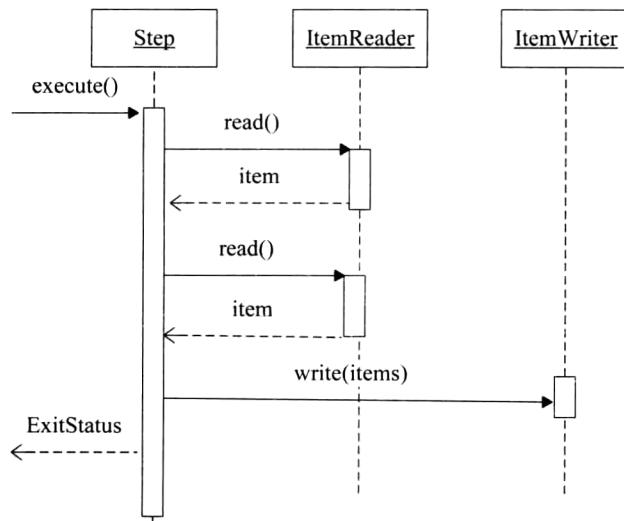


图 5-10 面向 Chunk 的操作序列图

通过 commit-interval 设置 Chunk 的提交频次，示例代码参见代码清单 5-18。

代码清单 5-18 设置 chunk 提交频次

```

1.   <job id="chunkJob">
2.     <step id="chunkStep">
3.       <tasklet transaction-manager="transactionManager" start-limit="1">
4.         <chunk reader="reader" processor="processor" writer="writer"
  
```

```
5.           commit-interval="5"/>
6.       </tasklet>
7.     </step>
8.   </job>
```

其中，5行：设置提交间隔属性 commit-interval 为 5，表示每读取 5 条记录后，进行一次写操作，同时将执行的状态数据通过 JobRepository 持久化到数据库中。可以将提交间隔设置的值作为作业中处理的最基本的单元，通过事务管理器保证了操作的一致性。

按照面向 Chunk 的操作，如果提交间隔是 5 次，那么读操作被调用 5 次，写操作被调用 1 次。读 Item 被汇总到列表中，最终被统一写出。读/写操作被封装在同一个事务块中，当前批次的作业步执行完毕后框架通过 JobRepository 将状态数据保存。

使用伪代码展示执行动作，参见代码清单 5-19。

代码清单 5-19 伪代码展示提交间隔

```
1. TransactionManager tm = ...;
2. JobRepository jobRepository=...;
3. try{
4.     tm.begin();
5.     List items = new ArrayList();
6.     for(int i = 0; i < commitInterval; i++){
7.         items.add(itemReader.read());
8.     }
9.     itemWriter.write(items);
10.    tm.commit();
11. }catch(Throwable t){
12.     tm.rollback();
13. }finally{
14.     jobRepository.addOrUpdate(batchMetaData); //状态数据保存
15. }
```

其中，4 行：事务开始。

5~9 行：根据 commit-interval 设置的值 5，读取 5 条数据，然后执行写。

10 行：事务提交。

12 行：发生异常回滚事务。

14 行：无论执行成功还是失败，都要将 Chunk 执行的状态数据持久化。

5.3.2 异常跳过

设想前面的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。

对账单格式错误文件，参见代码清单 5-20。

代码清单 5-20 格式错误对账单

```
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 Batch 框架提供了最大跳过记录数的限制，当跳过的记录数大于设定的值后，Job 作业将会失败。

Spring Batch 框架通过属性 skip-limit、skippable-exception-classes、skip-policy 来完成跳过能力。

skippable-exception-classes: 定义可以对记录跳过的异常，还可以定义跳过一组异常，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。

配置 Skip

配置 Skip 的配置文件参见代码清单 5-21。

代码清单 5-21 配置作业 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.ch05.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.ch05.RadomExceptionItemProcessor实现代码，参见代码清单5-22。

代码清单5-22 RadomExceptionItemProcessor实现类

```
1.  public class RadomExceptionItemProcessor implements ItemProcessor<String,
2.      String> {
3.
4.      Random ra = new Random();
5.
6.      public String process(String item) throws Exception {
7.          int i = ra.nextInt(10);
8.          System.out.println("Process " + item + "; Random i=" + i);
9.          if(i%2 == 0){
10.              throw new RuntimeException("make error!");
11.          }else{
12.              return item;
13.          }
14.      }
15.  }
```

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。

异常跳过的完整使用请参见第10章健壮Job，包括异常策略配置，异常拦截器的使用。

5.3.3 Step重试

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）和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，也不期望跳过处理，而是希望通过几次重试操作，尽可能地让 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 retryable-exception-classes、retry-limit、retry-policy、cache-capacity、retry-listeners 来实现。

retryable-exception-classes：定义可以重试的异常，可以定义一组重试的异常，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。

retry-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。

配置 Retry

配置 Retry 的配置文件，参见代码清单 5-23。

代码清单 5-23 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.ch05.listener.SystemOutRetry
19.               Listener" />
20.     <bean:bean id="alwaysExceptionItemProcessor"
```

```
20.           class="com.juxtapose.example.ch05.AlwaysExceptionItem  
Processor" />
```

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，可以定义多个重试的异常，`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.ch05.AlwaysExceptionItemProcessor` 实现，参见代码清单 5-24。

代码清单 5-24 AlwaysExceptionItemProcessor 实现类

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

5.3.4 Chunk 完成策略

面向 Chunk 的操作执行期间，根据设置的提交间隔 `commit-interval` 值，当读数据达到提交间隔后，执行一次提交操作，然后重复执行 Chunk 的读操作，直到再次达到间隔值。Spring Batch 框架除了提供 `commit-interval` 能力外，该框架还提供了 Chunk 完成策略能力，通过完成策略可以配置任务的提交时机，Chunk 完成策略的定义接口为 `org.springframework.batch.repeat.CompletionPolicy`。通过属性 `chunk-completion-policy` 定义批处理的完成策略。

面向 Chunk 的完成策略顺序参见图 5-11。

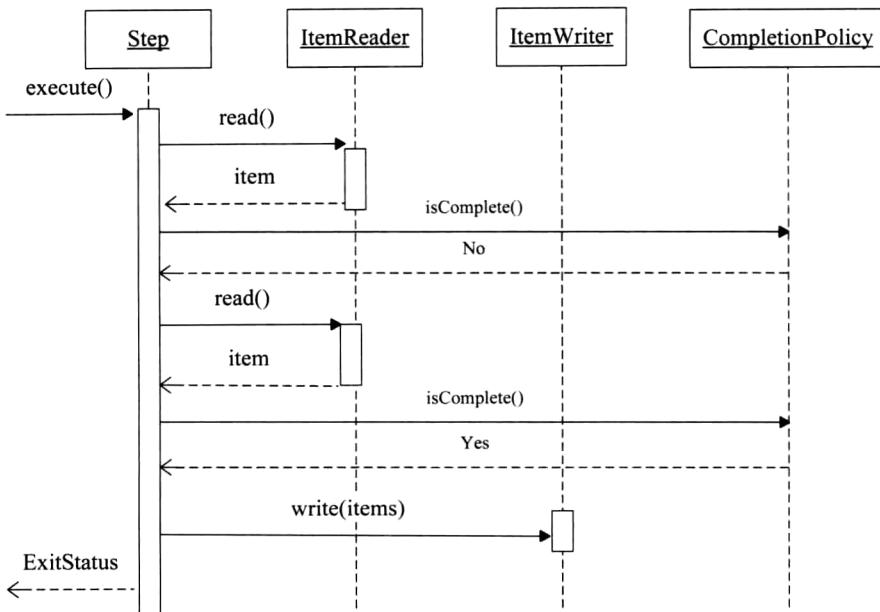


图 5-11 面向 Chunk 的完成策略顺序图

说明：chunk-completion-policy 定义批处理完成策略，不是表示任务的完成策略，Chunk 执行期间是按照 Chunk 完成策略执行批量提交的，批量提交会执行一次写操作，同时将批处理的状态数据通过 JobRepository 持久化。

说明：属性 chunk-completion-policy 和属性 commit-interval 不能同时存在；在 Chunk 中至少定义这两个其中的一个。即要么定义属性 chunk-completion-policy，要么定义属性 commit-interval。

CompletionPolicy 接口定义，参见代码清单 5-25。

代码清单 5-25 CompletionPolicy 接口定义

```

1. public interface CompletionPolicy {
2.     boolean isComplete(RepeatContext context, RepeatStatus result);
3.     boolean isComplete(RepeatContext context);
4.     RepeatContext start(RepeatContext parent);
5.     void update(RepeatContext context);
6. }

```

isComplete() 定义当前重复操作是否完成，start() 表示批处理操作开始，update() 完成更新当前批处理的状态数据。

系统提供的默认 CompletionPolicy 实现列表参见表 5-11。

接下来以 SimpleCompletionPolicy 为例展示如何使用 CompletionPolicy。读者可以自行研究其他完成策略的使用。

表 5-11 完成策略 CompletionPolicy 默认实现

CompletionPolicy 默认实现	功能说明
CompletionPolicySupport	完成策略实现类，可以基于此实现类继承实现自定义的完成策略。通常在 Spring Batch 中以 Support 为结尾的实现类，仅仅提供了接口的默认实现，方便继承类，仅覆盖感兴趣的方法，不用实现接口中的所有方法 org.springframework.batch.repeat.policy.CompletionPolicySupport
DefaultResultCompletionPolicy	完成策略默认实现类，根据 RepeatStatus 是否为 null 或者 RepeatStatus 允许继续判断是否完成 org.springframework.batch.repeat.policy.DefaultResultCompletionPolicy
SimpleCompletionPolicy	根据 chunkSize 值的大小，决定是否完成。最终 commit-interval 是通过 SimpleCompletionPolicy 来实现的 org.springframework.batch.repeat.policy.SimpleCompletionPolicy
TimeoutTerminationPolicy	根据 Chunk 执行的时间来判断是否完成，在给定的时间内 chunk 会重复执行，直到超过给定的时间 org.springframework.batch.repeat.policy.TimeoutTerminationPolicy
CompositeCompletionPolicy	组合完成策略，可以将多个完成策略组合在一起使用，按照顺序判断是否应该完成；多个组合策略中只要有一个满足完成条件，表示完成则组合完成策略 org.springframework.batch.repeat.policy.CompositeCompletionPolicy

配置 SimpleCompletionPolicy，参见代码清单 5-26。

代码清单 5-26 配置简单完成策略 SimpleCompletionPolicy

```

1.      <job id="completionPolicyJob">
2.          <step id="completionPolicyStep">
3.              <tasklet>
4.                  <chunk reader="reader" processor="processor" writer="writer"
5.                      chunk-completion-policy="completionPolicy"

```

其中，5 行：利用属性 chunk-completion-policy 定义 Chunk 的完成策略为 completionPolicy，completionPolicy 默认使用 SimpleCompletionPolicy。

10~12 行：定义 SimpleCompletionPolicy，配置属性 chunkSize 为 5，表示每处理 5 次数据后进行一次事务提交；然后重复执行 chunk 操作。

14~15 行：定义 chunk 的处理器为 PassThroughItemProcessor，该处理器不做任何业务处理，直接将 read 读取的数据返回。

5.3.5 读、处理事务

读事务队列

`reader-transactional-queue`：是否从一个事务性的队列读取数据，当 reader 从 JMS 的消息队列获取数据时候，此属性生效，默认值为 false。

true 表示从一个事务性的队列中读取数据，一旦发生异常会导致事务回滚，从队列中读取的数据同样会被重新放回到队列中；false 表示从一个没有事务的队列获取数据，一旦发生异常导致事务回滚，消费掉的数据不会重新放置在队列中。

读 JMS 消息的 ItemReader 的具体使用，请参见 6.6 章节。

处理事务

`processor-transactional`：处理数据是否在事务中，true 表示在一次 Chunk 处理期间将 processor 处理的结果放在缓存中，当执行重试或者跳过策略时可以看到缓存中处理的数据，在写操作完成前可以重新执行 processor；false 表示在一次 Chunk 处理期间不会将 processor 处理的数据放在缓存中，即 processor 在 chunk 执行期间每一条记录仅会执行一次。

事务性 Processor 处理过程图参见图 5-12。

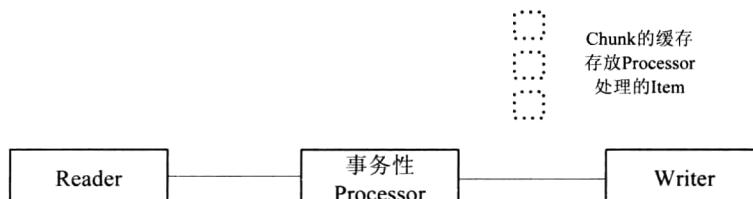


图 5-12 事务性 Processor 处理过程图

非事务性 Processor 处理过程图参见图 5-13。

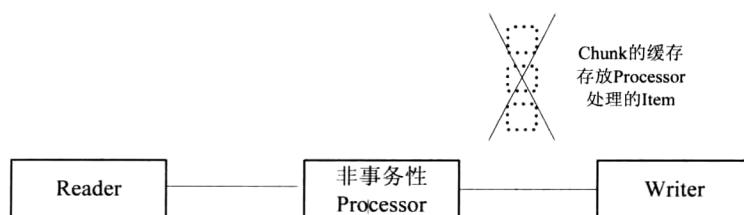


图 5-13 非事务性 Processor 处理过程图

说明：如果将 reader-transactional-queue 设置为 true，则 processor-transactional 必须设置为 true。

下面的示例，参见代码清单 5-27，它说明属性 processor-transactional 的具体使用方法和含义。作业 transactionPolicyJob 从整数中递增读取数据；processor 打印当前执行的数据，并根据属性 errorCount 的大小决定何时抛出 RuntimeException 异常；当异常 RuntimeException 抛出时，允许最大重试次数为 3 次；writer 负责将消息打印在控制台上。通过设置属性 processor-transactional 为不同的值，观察 processor 在控台上打印的消息：processor-transactional="true" 时，参见代码清单 5-29；processor-transactional="false" 时，参见代码清单 5-30。

代码清单 5-27 配置 processor-transactional

```
1.      <job id="transactionPolicyJob">
2.          <step id="transactionPolicyStep">
3.              <tasklet>
4.                  <chunk reader="reader" processor="processor" writer="writer"
5.                      commit-interval="5" processor-transactional="false"
6.                      retry-limit="3">
7.                      <retryable-exception-classes>
8.                          <include class="java.lang.RuntimeException" />
9.                      </retryable-exception-classes>
10.                 </chunk>
11.             </tasklet>
12.         </step>
13.     </job>
14.     <bean:bean id="processor" class="com.juxtapose.example.ch05.
    TransactionItemProcessor">
15.         <bean:property name="errorCount" value="3"></bean:property>
16.     </bean:bean>
```

其中，5 行：设置提交间隔为 5，重试次数为 3 次，processor 不在事务中。

6~8 行：设置重试异常为 java.lang.RuntimeException。

13~15 行：定义任务处理器，并设置 errorCount 为 3 时抛出 RuntimeException 异常 TransactionItemProcessor 类声明，参见代码清单 5-28。

代码清单 5-28 类 TransactionItemProcessor 定义

```
1. public class TransactionItemProcessor implements ItemProcessor<String,
   String> {
2.     private String errorCount = "3";
3.     public String process(String item) throws Exception {
4.         System.out.println("TransactionItemProcessor.process() item is:"
+ item);
5.         if(errorCount.equals(item)){
6.             throw new RuntimeException("make error!");
7.         }
}
```

```
8.         return item;
9.     }
10.    .....
11. }
```

其中，4行：控制台输出当前正在处理的item。

5~7行：根据设定的errorCount抛出异常，errorCount默认值为3。

代码清单 5-29 processor-transactional 为 true 执行结果

```
1. TransactionItemProcessor.process() item is:1
2. TransactionItemProcessor.process() item is:2
3. TransactionItemProcessor.process() item is:3
4. TransactionItemProcessor.process() item is:1
5. TransactionItemProcessor.process() item is:2
6. TransactionItemProcessor.process() item is:3
7. TransactionItemProcessor.process() item is:1
8. TransactionItemProcessor.process() item is:2
9. TransactionItemProcessor.process() item is:3
10. TransactionItemProcessor.process() item is:1
11. TransactionItemProcessor.process() item is:2
```

当processor-transactional为true时，表示每次Chunk执行过程中，在处理的数据没有达到提交间隔之前，经过processor处理的item会放在缓存中；当发生了重试时候，会从缓存中第一条开始执行。本例中执行Item分别为1、2、3，当Item为3的时候发生了异常导致进行重试操作，当processor-transactional为true的情况下进行重试的时候，会从Item值为1的开始重新执行，直到最后达到重试次数抛出异常终止Job为止。

代码清单 5-30 processor-transactional 为 false 执行结果

```
1. TransactionItemProcessor.process() item is:1
2. TransactionItemProcessor.process() item is:2
3. TransactionItemProcessor.process() item is:3
4. TransactionItemProcessor.process() item is:3
5. TransactionItemProcessor.process() item is:3
```

当processor-transactional为false时，表示每次Chunk执行过程中，经过processor处理的数据不会存放在缓存中，当发生了异常导致重试的时候，已经处理过的1、2两条记录不会再被执行。请注意控制台输出，当第一次执行到记录3发生异常后，后面的重试操作均直接从记录3开始。

5.4 拦截器

Chunk操作中提供了丰富的拦截器机制，通过拦截器可以实现额外的控制能力，例如日志记录、任务跟踪、状态报告、数据传递等能力；在使用Spring Batch的过程中，尽可能地保

持业务的简单性，任何额外的处理需要在拦截器中进行功能实现。

Chunk 操作中提供的拦截器，参见表 5-12。

表 5-12 Chunk 拦截器接口

Chunk 拦截器接口	功能说明
ChunkListener	批操作拦截器，主要操作：chunk 执行前，chunk 执行后 org.springframework.batch.core.ChunkListener
ItemProcessListener	处理器拦截器，主要操作：处理执行前，处理执行后，处理发生异常 org.springframework.batch.core.ItemProcessListener<T, S>
ItemReadListener	读操作拦截器，主要操作：读之前、读之后、读发生异常 org.springframework.batch.core.ItemReadListener<T>
ItemWriteListener	写操作拦截器，主要操作：写之前、写之后、写发生异常 org.springframework.batch.core.ItemWriteListener<S>
SkipListener	记录跳过拦截器，主要操作：读错误导致跳过时、写错误导致跳过时、处理错误导致跳过时 org.springframework.batch.core.SkipListener<T, S>
RetryListener	重试拦截器，主要操作：重试开始、重试结束、重试异常 org.springframework.batch.retry.RetryListener

Chunk 拦截器作用域参见图 5-14，按照作用域的大小分别执行对应的 before 和 after 操作，其中读、处理、写三个拦截器没有嵌套关系，三者按照顺序先后执行。Chunk 拦截器先后执行顺序参见代码清单 5-31。

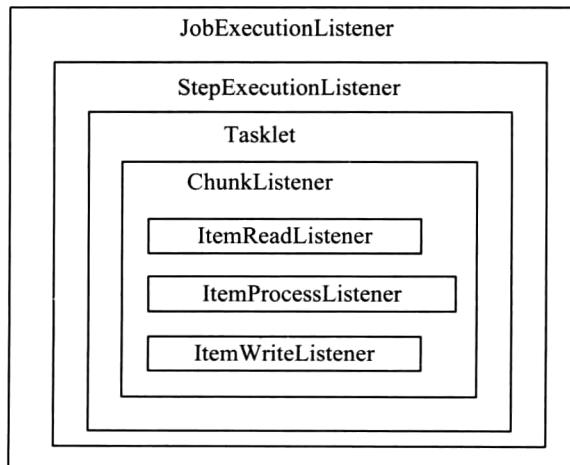


图 5-14 Chunk 拦截器作用域

代码清单 5-31 Chunk 拦截器先后执行顺序

```
1. JobExecutionListener.beforeJob()
2. StepExecutionListener.beforeStep()
3. ChunkListener.beforeChunk()
4. ItemReadListener.beforeRead()
5. ItemReadListener.afterRead()
6. ItemProcessListener.beforeProcess()
7. ItemProcessListener.afterProcess()
8. ItemWriteListener.beforeWrite()
9. ItemWriteListener.afterWrite()
10. ChunkListener.afterChunk()
11. StepExecutionListener.afterStep()
12. JobExecutionListener.afterJob()
```

典型的拦截器配置格式，参见代码清单 5-32。

代码清单 5-32 配置 Chunk 拦截器

```
1. <job id="listenerJob">
2.   <step id="listenerStep">
3.     <tasklet>
4.       <chunk reader="reader20" processor="processor" writer="writer"
5.             commit-interval="1" retry-limit="1">
6.             <retry-listeners>
7.               <listener ref="retryListener"></listener>
8.             </retry-listeners>
9.             <retryable-exception-classes>
10.               <include class="java.lang.RuntimeException" />
11.             </retryable-exception-classes>
12.             <listeners>
13.               <listener ref="chunkListener"></listener>
14.               <listener ref="itemReadListener"></listener>
15.               <listener ref="itemProcessListener"></listener>
16.               <listener ref="itemWriteListener"></listener>
17.               <listener ref="skipListener"></listener>
18.             </listeners>
19.           </chunk>
20.         </tasklet>
21.         <listeners>
22.           <listener ref="stepExecutionListener"></listener>
23.         </listeners>
24.       </step>
25.       <listeners>
26.         <listener ref="jobExecutionListener"></listener>
27.       </listeners>
28.     </job>
```

其中，7行：配置重试拦截器 retryListener，当发生指定的异常时会执行该拦截器操作。
13行：配置批拦截器 chunkListener，在批处理操作执行时执行该拦截器。
14行：配置读拦截器 itemReadListener，在批处理中的读操作执行时执行该拦截器。
15行：配置处理拦截器 itemProcessListener，在批处理中的处理操作执行时执行该拦截器。
16行：配置写拦截器 itemWriteListener，在批处理中的写操作执行时执行该拦截器。
17行：配置跳过拦截器 skipListener，在批处理中当有记录被跳过时执行该拦截器。
22行：配置 Step 执行的拦截器 stepExecutionListener，在批处理的每步执行时执行该拦截器。
26行：配置 Job 执行的拦截器 jobExecutionListener，在批处理 Job 执行时执行该拦截器。
接下来本节会详细描述 ChunkListener、SkipListener、RetryListener 的具体使用，另外的读、处理、写阶段的拦截器在后面读、写、处理章节详细描述。

5.4.1 ChunkListener

批操作拦截器，主要操作：Chunk 执行前，Chunk 执行后。接口 ChunkListener 声明参见代码清单 5-33。

代码清单 5-33 ChunkListener 接口定义

```
1. public interface ChunkListener extends StepListener {  
2.     void beforeChunk();  
3.     void afterChunk();  
4. }
```

ChunkListener 操作说明与 Annotation 定义，参见表 5-13。

表 5-13 ChunkListener 操作说明与 Annotation 定义

操作	操作说明	Annotation
beforeChunk()	在 Chunk 执行前出发，在 Step 事务中	@ BeforeChunk
afterChunk()	在 Chunk 执行后出发，不在 Step 事务中	@ AfterChunk

ChunkListener 系统默认实现，参见表 5-14。

表 5-14 ChunkListener 默认实现

ChunkListener 默认实现	功能说明
ChunkListenerSupport	Chunk 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.ChunkListenerSupport
CompositeChunkListener	组合 Chunk 拦截器实现，可以定义一组的 Chunk 拦截器，依照顺序执行 org.springframework.batch.core.listener.CompositeChunkListener

5.4.2 ItemReadListener

ItemReadListener 在 Chunk 读阶段的拦截器，可以在读之前、读之后、读发生异常时候触发该拦截器。接口 ItemReadListener 声明参见代码清单 5-34。

代码清单 5-34 ItemReadListener 接口声明

```
1. public interface ItemReadListener<T> extends StepListener {  
2.     void beforeRead();  
3.     void afterRead(T item);  
4.     void onReadError(Exception ex);  
5. }
```

ItemReadListener 操作说明与 Annotation 定义，参见表 5-15。

表 5-15 ItemReadListener 操作说明与 Annotation 定义

操作	操作说明	Annotation
beforeRead()	在 ItemReader#read()之前执行	@ BeforeRead
afterRead(T item)	在 ItemReader#read()之后执行	@ AfterRead
onReadError(Exception ex)	当 ItemReader#read()抛出异常时候触发该操作	@ OnReadError

ItemReadListener 的具体使用，请参见 6.9 章节。

5.4.3 ItemProcessListener

ItemProcessListener 在 Chunk 处理阶段的拦截器，可以在处理之前、处理之后、处理发生异常时触发该拦截器。接口 ItemProcessListener 声明参见代码清单 5-35。

代码清单 5-35 ItemProcessListener 接口声明

```
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. }
```

ItemProcessListener 操作说明与 Annotation 定义，参见表 5-16。

表 5-16 ItemProcessListener 操作说明与 Annotation 定义

操作	操作说明	Annotation
beforeProcess(T item)	在 ItemProcessor.process(Object)之前执行	@ BeforeProcess
afterProcess(T item, S result)	在 ItemProcessor.process(Object)之后执行，即使 process 方法返回 null，仍然会触发拦截器操作	@ AfterProcess
onProcessError(T item, Exception e)	当 ItemProcessor.process(Object)抛出异常时触发该操作	@ OnProcessError

ItemProcessListener 的具体使用，请参见 8.7 章节。

5.4.4 ItemWriteListener

ItemWriteListener 在 Chunk 写阶段的拦截器，可以在写之前、写之后、写发生异常时触发该拦截器。接口 ItemWriteListener 声明参见代码清单 5-36。

代码清单 5-36 ItemWriteListener 接口声明

```
1. public interface ItemWriteListener<S> extends StepListener {  
2.     void beforeWrite(List<? extends S> items);  
3.     void afterWrite(List<? extends S> items);  
4.     void onWriteError(Exception exception, List<? extends S> items);  
5. }
```

ItemWriteListener 操作说明与 Annotation 定义参见表 5-17。

表 5-17 ItemWriteListener 操作说明与 Annotation 定义

操作	操作说明	Annotation
beforeWrite(List<? extends S> items)	在 ItemWriter#write(java.util.List)之前执行	@ BeforeWrite
afterWrite(List<? extends S> items)	在 ItemWriter#write(java.util.List)之后执行 该操作会在事务提交之前、ChunkListener #afterChunk()之前执行	@ AfterWrite
onWriteError(Exception exception, List<? extends S> items)	当 ItemWriter#write(java.util.List)抛出异常时触发该操作	@ OnWriteError

ItemWriteListener 的具体使用，请参见 7.12 章节。

5.4.5 SkipListener

SkipListener 在 Chunk 处理阶段抛出跳过定义的异常时触发，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 声明参见代码清单 5-37。

代码清单 5-37 SkipListener 接口声明

```
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 5-18。

表 5-18 SkipListener 操作说明与 Annotation 定义

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触发该操作	@ OnSkipInProcess

SkipListener 系统默认实现类参见表 5-19。

表 5-19 SkipListener 默认实现

SkipListener 默认实现	功能说明
CompositeSkipListener	组合跳过拦截器实现，可以定义一组的跳过拦截器，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>

SkipListener 的具体使用请参见 10 章节。

5.4.6 RetryListener

RetryListener 在 Chunk 处理阶段抛出重试定义的异常时触发，通过拦截器可以在重试动作发生时进行日志记录、收集重试信息等。可以直接实现接口 RetryListener，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。接口 RetryListener 声明参见代码清单 5-38。

代码清单 5-38 RetryListener 接口声明

```

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 操作说明参见表 5-20。

表 5-20 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
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 系统默认实现参见表 5-21。

表 5-21 RetryListener 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

RetryListener 的具体使用请参见 10 章节。

读数据 ItemReader

批处理通过 Tasklet 完成具体的任务，Chunk 类型的 Tasklet 定义了标准的读、处理、写的执行步骤。ItemReader 是实现读的重要组件，Spring Batch 框架提供了丰富的读基础设施来完成各种数据来源的读取功能，数据来源包括文本文件、Json 格式文件、XML 文件、DB、JMS 消息等。

Spring Batch 框架默认提供了丰富的 Reader 实现；如果不能满足需求可以快速方便地实现自定义的数据读取；对于已经存在的读服务，框架提供了复用现有服务的能力，避免重复开发。

Spring Batch 框架为保证 Job 的可靠性、稳定性，在读数据阶段提供了另外一个重要的特性是对读数据状态的保存，在 Spring Batch 框架内置的 Read 实现中通过与执行上下文进行读数据的状态交互，可以高效地保证 Job 的重启和错误处理。

6.1 ItemReader

ItemReader 是 Step 中对资源的读处理，Spring Batch 框架已经提供了各种类型的读实现，包括对文本文件、XML 文件、数据库、JMS 消息等读的处理。读操作与 Chunk Tasklet 的关系参见图 6-1。

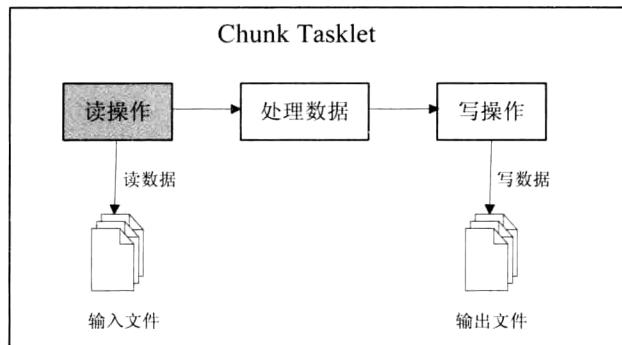


图 6-1 读操作与 Chunk Tasklet 关系图

6.1.1 ItemReader

所有的读操作需要实现 `org.springframework.batch.item.ItemReader<T>` 接口。ItemReader 接口定义参见代码清单 6-1。

代码清单 6-1 ItemReader 接口定义

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

其中，2 行：ItemReader 接口定义了核心作业方法 read() 操作，负责从给定的资源中读取可用的数据。

Job 中典型的配置 ItemReader 参见代码清单 6-2。

代码清单 6-2 配置 ItemReader

```
1. <job id="dbReadJob">  
2.     <step id="dbReadStep">  
3.         <tasklet transaction-manager="transactionManager">  
4.             <chunk reader="jdbcParameterItemReader"  
5.                 processor="creditBillProcessor"  
6.                 writer="creditItemWriter" commit-interval="2"></chunk>  
7.         </tasklet>  
8.     </step>  
9. </job>
```

6.1.2 ItemStream

Spring Batch 框架同时提供了另外一个重要的接口 org.springframework.batch.item.ItemStream。ItemStream 接口定义参见代码清单 6-3。ItemStream 接口定义了读操作与执行上下文 ExecutionContext 交互的能力。可以将已经读的条数通过该接口存放在执行上下文 ExecutionContext 中（ExecutionContext 中的数据在批处理 commit 的时候会通过 JobRepository 持久化到数据库中），这样到 Job 发生异常重新启动 Job 的时候，读操作可以跳过已经成功读过的数据，继续从上次出错的地点（可以从执行上下文中获取上次成功读的位置）开始读。

代码清单 6-3 ItemStream 接口定义

```
1. public interface ItemStream {  
2.     void open(ExecutionContext executionContext) throws  
3.             ItemStreamException;  
4.     void update(ExecutionContext executionContext) throws  
5.             ItemStreamException;  
6.     void close() throws ItemStreamException;  
7. }
```

其中，2 行：open() 操作根据参数 executionContext 打开需要读取资源的 stream；可以根据持久化在执行上下文 executionContext 中的数据重新定位需要读取记录的位置。

3 行：update() 操作将需要持久化的数据存放在执行上下文 executionContext 中。

4 行：close() 操作关闭读取的资源。

说明：Spring Batch 框架提供的读组件均实现了 ItemStream 接口。

6.1.3 系统读组件

Spring Batch 框架提供了丰富的的读组件，包括对文件、关系数据库、NoSQL 数据库、消息队列等，具体参见表 6-1。

表 6-1 Spring Batch 框架提供的读组件

ItemReader	说 明
ListItemReader	读取 List 类型数据，只能读一次
ItemReaderAdapter	ItemReader 适配器，可以复用现有的读操作
FlatFileItemReader	读 Flat 类型文件
StaxEventItemReader	读 XML 类型文件
JdbcCursorItemReader	基于 JDBC 游标方式读数据库
HibernateCursorItemReader	基于 Hibernate 游标方式读数据库
StoredProcedureItemReader	基于存储过程读数据库
IbatisPagingItemReader	基于 Ibatis 分页读数据库
JpaPagingItemReader	基于 Jpa 方式分页读数据库
JdbcPagingItemReader	基于 JDBC 方式分页读数据库
HibernatePagingItemReader	基于 Hibernate 方式分页读取数据库
JmsItemReader	读取 JMS 队列
IteratorItemReader	迭代方式的读组件
MultiResourceItemReader	多文件读组件
MongoItemReader	基于分布式文件存储的数据库 MongoDB 读组件
Neo4jItemReader	面向网络的数据库 Neo4j 的读组件
ResourcesItemReader	基于批量资源的读组件，每次读取返回资源对象
AmqpItemReader	读取 AMQP 队列组件
RepositoryItemReader	基于 Spring Data 的读组件

6.2 Flat 格式文件

Flat 类型文件是一种包含没有相对关系结构的记录的文件。在批处理应用中经常需要处理的文件是简单文本格式文件，这类文件通常没有复杂的关系结构，通常经过分隔符分割，或者定长字段来描述数据格式；稍复杂的文件通过 JSON 的方式定义数据格式。

Spring Batch 框架提供的 ItemReader 本质上是从 Flat 文件中读取记录，并将记录转换为 Java 中的对象。在进行读取 Flat 文件之前，我们先学习一下 Flat 文件格式的几种不同类型。

6.2.1 Flat 文件格式

本章仍然采用信用卡对账单的例子来介绍 Flat 文件的格式。通过框架提供的对文件的 ItemReader，可以支持定长类型、分隔符类型、JSON 格式、跨多行分隔符、交替记录分隔符文件等。图 6-2 给出了 ItemReader 支持的多种文件类型。

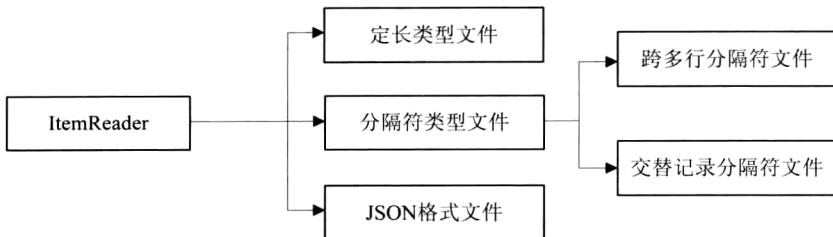


图 6-2 ItemReader 支持多种文件类型

6.2.1.1 分隔符类型文件

分隔符类型文件通常是根据设定的分隔符来区分一条记录中不同字段的，在每个字段中不能出现内容为分隔符的字符，否则会视为分隔符号。代码清单 6-4 给出了逗号分隔符的示例。

代码清单 6-4 逗号分隔符示例

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,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,Longyang road
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
```

第一段表示信用卡号，第二段表示用户名，第三段表示消费金额，第四段表示消费日期，第五段表示消费地点。

6.2.1.2 定长类型文件

每一个字段的长度是确定的，定长类型的文件没有任何分隔符。代码清单 6-5 给出了定长类型文件的示例。

代码清单 6-5 定长类型文件示例

```
1. 4041390012345678tom      00100.002013-02-02 12:00:08   Lu Jia Zui road
2. 4042390012345678tom      00320.002013-02-03 10:35:21   Lu Jia Zui road
3. 4043390012345678jerry    00674.702013-02-06 16:26:49   South Linyi road
4. 4044390012345678rose     00793.202013-02-09 15:15:37   Longyang road
5. 4045390012345678bruce    00360.002013-02-11 11:12:38   Longyang road
6. 4046390012345678rachle   00893.002013-02-28 20:34:19   Hunan road
```

前 16 个字符表示信用卡卡号，接下来 12 个字符表示用户名，接下来 8 个字符表示消费金额，接下来 19 个字符表示消费日期，最后 16 个字符表示消费地点。

6.2.1.3 JSON 格式文件

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。它是基于 JavaScript (Standard ECMA-262 3rd Edition - December 1999) 的一个子集。

JSON 有两种结构：json 简单说就是 JavaScript 中的对象和数组，所以这两种结构就是对象和数组，通过这两种结构可以表示各种复杂的结构。

(1) 对象：对象在 js 中表示为“{}”扩起来的内容，数据结构为 {key: value, key: value,...} 的键值对的结构，在面向对象的语言中，key 为对象的属性，value 为对应的属性值，所以很容易理解，取值方法为对象.key 获取属性值，这个属性值的类型可以是数字、字符串、数组和对象。

(2) 数组：数组在 js 中是用中括号“[]”扩起来的内容，数据结构为 ["java", "javascript", "vb", ...]，取值方式和所有语言中一样，使用索引获取，字段值的类型可以是数字、字符串、数组和对象。

经过对象、数组两种结构就可以组合成复杂的数据结构了。

代码清单 6-6 展示了信用卡数据是通过 JSON 格式进行存储的。

代码清单 6-6 JSON 格式数据示例

```
1. { "accountID": "4047390012345678",
2.   "name": "tom",
3.   "amount": 100.00,
4.   "date": "2013-2-2 12:00:08",
5.   "address": "Lu Jia Zui road"}
6. { "accountID": "4047390012345678",
7.   "name": "tom",
8.   "amount": 320.00,
9.   "date": "2013-2-3 10:35",
10.  "address": "Lu Jia Zui road"}
```

6.2.1.4 记录跨多行格式

通常情况下一条记录占用一行，在某些特殊情况下可能出现一条记录占用多行的场景，下面展示了一条记录占用两行。第一行描述了信用卡编号、姓名、消费金额、消费时间，第二行是消费地址。代码清单 6-7 给出了记录跨多行格式示例。

代码清单 6-7 记录跨多行格式示例

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08
2. ,Lu Jia Zui road
3. 4047390012345678,tom,320.00,2013-2-3 10:35:21
4. ,Lu Jia Zui road
5. 4047390012345678,tom,674.70,2013-2-6 16:26:49
6. ,South Linyi road
7. 4047390012345678,tom,793.20,2013-2-9 15:15:37
8. ,Longyang road
9. 4047390012345678,tom,360.00,2013-2-11 11:12:38
```

```
10. ,Longyang road  
11. 4047390012345678,tom,893.00,2013-2-28 20:34:19  
12. ,Hunan road
```

6.2.1.5 交替记录

另外一种更复杂的文件格式是在同一个文件中有两种或者更多类型的记录类型，不同的记录类型通过不同的前缀方式表示，下面展示了一个文件中两种不同类型的记录。代码清单 6-8 给出了交替记录的文件示例。

代码清单 6-8 交替记录的文件示例

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road  
2. 3047390012345671,249.10,rose,2013/4/1 11:34:56  
3. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road  
4. 3047390012345671,249.10,rose,2013/4/1 11:34:56  
5. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road  
6. 3047390012345673,840.20,marry,2013/4/3 3:21:43  
7. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
```

以 40 开头的为信用卡消费记录，以 30 开头的为借记卡消费记录情况。

6.2.2 FlatFileItemReader

FlatFileItemReader 实现 ItemReader 接口，核心作用将 Flat 文件中的记录转换为 Java 对象。FlatFileItemReader 通过引用 RecordSeparatorPolicy、LineMapper、LineCallbackHandler 关键接口实现上面的目的；在 FlatFileItemReader 将记录转换为 Java 对象时主要有两步工作，首先根据 RecordSeparatorPolicy 从文件中确定一条记录，其次使用 LineMapper 将一条记录转换为 Java 对象，具体步骤参见图 6-3。

FlatFileItemReader 结构及关键属性

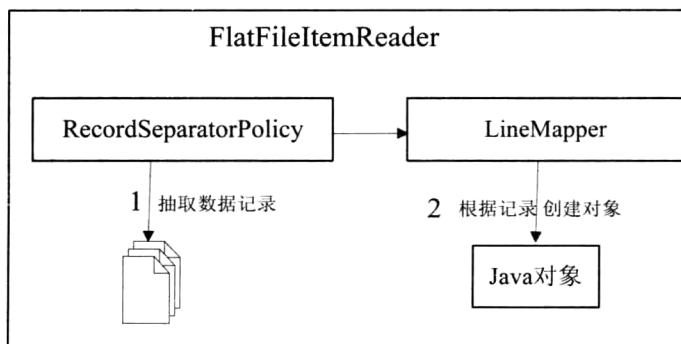


图 6-3 FlatFileItemReader 核心操作步骤

FlatFileItemReader 与关键接口 RecordSeparatorPolicy、LineMapper、LineCallbackHandler 之间的类图参见图 6-4，关键接口、类说明参见表 6-2。

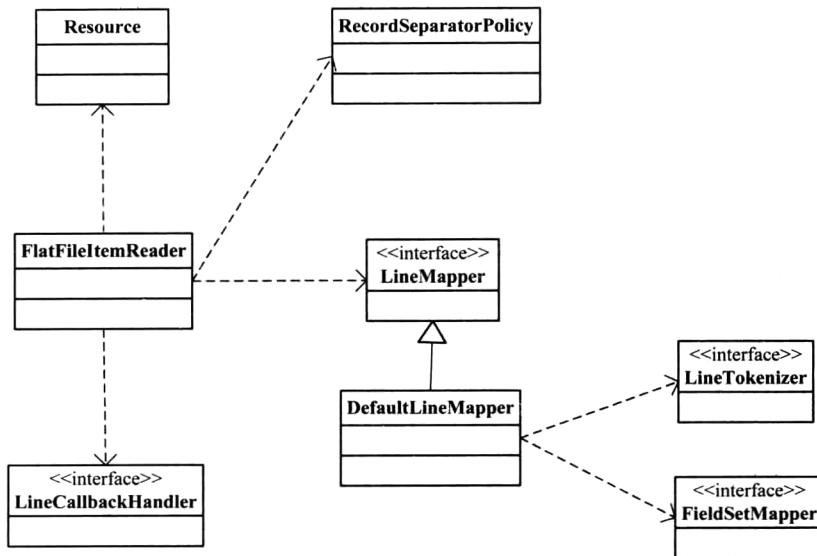


图 6-4 FlatFileItemReader 类关系图

FlatFileItemReader 引用 org.springframework.core.io.Resource，Resource 负责提供读取的资源信息，可以是文件，也可以是其他类型的资源；FlatFileItemReader 通过 RecordSeparatorPolicy 对文件进行分割，RecordSeparatorPolicy 定义了如何区分文件中的记录能力；LineMapper 对象提供最为核心的能力，将文件中的记录通过 LineMapper 对象转换为 Java 对象；LineCallbackHandler 在文件中行跳过时触发，通常与属性 linesToSkip 配合使用。

表 6-2 FlatFileItemReader 中关键接口、类说明

关键类	说 明
Resource	定义读取的文件资源
RecordSeparatorPolicy	从文件中确定一条记录的策略，记录可能是一行，可能是跨多行
LineMapper	将一条记录转化为 Java 数据对象，通常由 LineTokenizer 和 FieldSetMapper 组合来实现该功能
LineTokenizer	将一条记录分割为多个字段，在 LineMapper 的默认实现 DefaultLineMapper 中使用
FieldSetMapper	将多个字段值转化为 Java 对象，在 LineMapper 的默认实现 DefaultLineMapper 中使用
LineCallbackHandler	处理文件中记录回调处理操作

FlatFileItemReader 关键属性参见表 6-3。

表 6-3 FlatFileItemReader 关键属性

FlatFileItemReader 属性	类 型	说 明
bufferedReaderFactory	BufferedReaderFactory	根据给定的 resource 创建 BufferedReader 实例， 默认使用 DefaultBufferedReaderFactory 创建文本类型的 BufferedReader 实例
comments	String[]	定义注释行的前缀，当某行以这些字符串中的一个开头时候，此行记录将会被 Spring Batch 框架忽略
encoding	String	读取文件的编码类型， 默认值为从环境变量 file.encoding 获取，如果没有设置则默认为 UTF-8
lineMapper	LineMapper<T>	将一行文件记录转换为 Java 对象
linesToSkip	int	读取文件时候，定义跳过文件的行数；跳过的行记录将会被传递给 skippedLinesCallback，执行跳过行的回调操作
recordSeparatorPolicy	RecordSeparatorPolicy	定义文件如何区分记录，可以按照单行、也可以按照多行区分记录
resource	Resource	需要读取的资源文件
skippedLinesCallback	LineCallbackHandler	定义文件中记录跳过时执行的回调操作，通常与 linesToSkip 一起使用
strict	boolean	定义读取文件不存在时候的策略，如果为 true 则抛出异常，如果为 false 表示不抛出异常，默认值为 true

配置 FlatFileItemReader

在给出详细配置文件之前，我们首先给出一个更复杂的 csv 文件的复杂示例，具体参见代码清单 6-9，文件中增加了注释行，注释以"##"或者"\$\$"开头；同时在头部有文件的结构性说明。

完整 CSV 格式文件参见：ch06/data/flat/credit-card-bill-201303-complex.csv。

代码清单 6-9 复杂 CSV 格式文件

```

1. accountID,name,amount,date,address
2. ## this line is first comment
3. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
4. ## maybe address is not right
5. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
6. $$ this line is comment, begin with &&
7. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
8. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
9. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road
10. ## maybe time is not right
11. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road

```

其中，1 行：给出了文件的结构性说明，该文件每条记录共有 5 个字段，分别是信用卡账户 ID、姓名、消费金额、消费日期、消费地点；在进行记录处理的时候需要跳过该行。

2、4、6、10 行：是注释信息，在进行信用卡消费对账处理的时候不希望处理这些记录。了解了我们需要处理的 CSV 文件后，我们可利用现有的 FlatFileItemReader 组件，配置读复杂 CSV 格式文件，具体配置参见代码清单 6-10。

完整配置参见文件：ch06/job/job-flatfile.xml。

代码清单 6-10 配置 FlatFileItemReader

```
1.      <bean:bean id="flatFileItemReader" scope="step"
2.          class="org.springframework.batch.item.file.FlatFileItemReader">
3.              <bean:property name="resource"
4.                  value="classpath:ch06/data/flat/credit-card-bill-201303-
complex.csv"/>
5.              <bean:property name="lineMapper" ref="lineMapper" />
6.              <bean:property name="encoding" value="UTF-8" />
7.              <bean:property name="linesToSkip" value="1"/>
8.              <bean:property name="strict" value="true"/>
9.              <bean:property name="skippedLinesCallback" ref="lineCallbackHandler"/>
10.             <bean:property name="comments">
11.                 <bean:list>
12.                     <bean:value>##</bean:value>
13.                     <bean:value>$$</bean:value>
14.                 </bean:list>
15.             </bean:property>
16.             <bean:property name="recordSeparatorPolicy" ref="simpleRecord
SeparatorPolicy" />
17.         </bean:bean>
18.         <bean:bean id="lineCallbackHandler"
19.             class="com.juxtapose.example.ch06.flat.DefaultLineCallback
Handler">
20.             </bean:bean>
```

其中，3~4 行：定义读取的文件。

5 行：定义 lineMapper 属性，负责将单条记录转化为 Java 对象。

6 行：定义文件编码属性，使用 UTF-8 编码读取文件。

7 行：定义忽略文件的第一行，即文件第一行不处理，此处忽略的记录会触发属性 skippedLinesCallback 中定义的回调操作。

8 行：定义严格的文件存在检查策略，当 resource 中定义的文件不存在时会导致 Job 失败。

9 行：忽略文件前几行属性 linesToSkip 定义的情况下，这些忽略的记录被传递给此处定义的回调操作。

10~15 行：定义文件中的注释行，所有以"##"或者"\$\$"开头的行将会被忽略掉。

16 行：根据定义的策略 simpleRecordSeparatorPolicy 分割单条记录。

18~20 行：跳过行的回调操作，本实现中仅打印跳过的记录内容。

说明：在对 Flat 类型文件处理的时候，一行和一条记录是有区别的：一行内容不一定是一条记录，因为一条记录可能跨多行。

6.2.3 RecordSeparatorPolicy

RecordSeparatorPolicy 负责区分文件中不同记录的能力，即 RecordSeparatorPolicy 定义了如何从 Flat 文件中将不同的行记录区分开来作为一个完整的记录。通过该接口，读者可以实现任何复杂的文件记录区分的处理。

接口 org.springframework.batch.item.file.separator.RecordSeparatorPolicy 定义，参见代码清单 6-11。

代码清单 6-11 RecordSeparatorPolicy 接口定义

```
1. public interface RecordSeparatorPolicy {  
2.     boolean isEndOfRecord(String line);  
3.     String postProcess(String record);  
4.     String preProcess(String record);  
5. }
```

其中，isEndOfRecord()操作定义记录是否结束。

preProcess()操作在一个新行加入到记录前触发。

postProcess()操作在一个完整记录返回前触发。

RecordSeparatorPolicy 系统实现

Spring Batch 框架中提供了 4 类默认的实现，参见表 6-4，每种实现完成特定的记录分割能力，如果默认的 4 类实现不能满足要求，读者可以自行实现 RecordSeparatorPolicy 接口进行分割记录。

表 6-4 RecordSeparatorPolicy 系统默认实现类

RecordSeparatorPolicy	说 明
SimpleRecordSeparatorPolicy	每一行作为一条记录 org.springframework.batch.item.file.separator.SimpleRecordSeparatorPolicy
DefaultRecordSeparatorPolicy	如果行没有不匹配的引号，或者续行标识符“\\”，则每行作为一条记录，否则不作为一条记录；注意：判断行最后是否续行的时候，行后面的空格会被忽略掉 org.springframework.batch.item.file.separator.DefaultRecordSeparatorPolicy
SuffixRecordSeparatorPolicy	判断行是否以特定的后缀结束，如果是则认为是一条记录，否则不作为一条记录 org.springframework.batch.item.file.separator.SuffixRecordSeparatorPolicy
JsonRecordSeparatorPolicy	JSON 格式文件的行分割判断策略， org.springframework.batch.item.file.separator.JsonRecordSeparatorPolicy

说明：FlatFileItemReader 默认使用 SimpleRecordSeparatorPolicy 作为记录分割策略。

6.2.4 LineMapper

LineMapper 接口定义了如何将 Flat 文件中的一条记录转化为领域对象（通常为 Java 对象）。LineMapper 接口仅有一个方法 mapLine()，传入的参数是行的内容和行号，返回值为转换后的领域对象。LineMapper 接口定义参见代码清单 6-12。

代码清单 6-12 LineMapper 接口定义

```
1. public interface LineMapper<T> {  
2.     T mapLine(String line, int lineNumber) throws Exception;  
3. }
```

Spring Batch 框架中提供了 4 类默认的实现，参见表 6-5，每种实现完成特定的数据转换，如果默认的 4 类实现不能满足要求，读者可以自行定义 LineMapper 接口进行数据转换。

表 6-5 LineMapper 系统默认实现类

LineMapper	说 明
DefaultLineMapper<T>	默认的行转换类，引用接口 LineTokenizer 和 FieldSetMapper<T>完成数据转换；LineTokenizer 负责将一条记录转换为对象 FieldSet（可以看作是一个 key-value 对的组合），FieldSetMapper<T>负责将 FieldSet 转换为领域对象 org.springframework.batch.item.file.mapping.DefaultLineMapper<T>
JsonLineMapper	负责将文件中 JSON 格式的文本数据转换为领域对象，转换后的领域对象格式为 Map<String, Object> org.springframework.batch.item.file.mapping.JsonLineMapper
PassThroughLineMapper	最简化的数据转换实现类，将一条记录直接返回，可以认为返回的领域对象为 String 类型的格式 org.springframework.batch.item.file.mapping.PassThroughLineMapper
PatternMatchingCompositeLineMapper<T>	复杂数据转换类，可以为不同的记录定义不同的 LineTokenizer 和 FieldSetMapper<T>来实现数据转换；在执行过程中，根据每条记录的内容根据设置的条件找到匹配的 LineTokenizer 和 FieldSetMapper<T>进行数据转换；多用于处理同一文件中有不同类型记录的场景 可以认为 PatternMatchingCompositeLineMapper<T>组合了多个 DefaultLineMapper<T> org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper<T>

接下来的章节详细介绍 DefaultLineMapper 的功能实现。

6.2.5 DefaultLineMapper

org.springframework.batch.item.file.mapping.DefaultLineMapper<T>默认使用 LineTokenizer 和 FieldSetMapper<T>完成数据转换功能。DefaultLineMapper 通过组合的方式将任务委托给两个接口来完成，简化了 DefaultLineMapper 的代码结构，同时使得每个对象完成的任务职责非常简单，保持了代码结构清晰。DefaultLineMapper、LineTokenizer、FieldSetMapper 三者之间的关系参见图 6-5。

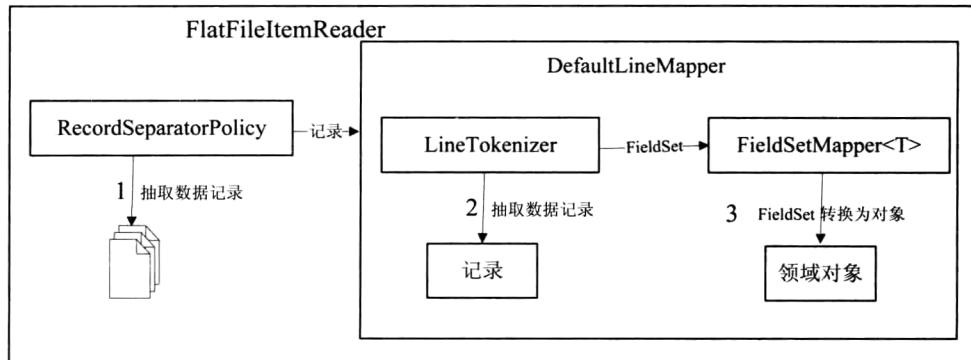


图 6-5 DefaultLineMapper、LineTokenizer、FieldSetMapper 三者之间的关系

其中，LineTokenizer 负责将一条记录转换为 FieldSet 对象；

FieldSetMapper 负责将 FieldSet 对象转换为领域对象。

代码清单 6-13 给出了配置 DefaultLineMapper 示例代码。

代码清单 6-13 配置 DefaultLineMapper

```
1. <bean:bean id="lineMapper"
2.   class="org.springframework.batch.item.file.mapping.
   DefaultLineMapper" >
3.   <bean:property name="lineTokenizer" ref=" delimitedLineTokenizer " />
4.   <bean:property name="fieldSetMapper"
   ref="creditBillBeanWrapperFieldSetMapper"/>
5. </bean:bean>
```

其中，3 行：属性 lineTokenizer，使用 DelimitedLineTokenizer 将行记录转换为 FieldSet 对象。

4 行：属性 fieldSetMapper，使用 BeanWrapperFieldSetMapper 将 FieldSet 对象转换为领域对象 CreditBill。

完成的配置 DefaultLineMapper

代码清单 6-14 给出了完成的配置 LineMapper，使用了 DelimitedLineTokenizer 和 BeanWrapperFieldSetMapper 进行数据转换。

代码清单 6-14 配置 LineMapper

```
1.      <bean:bean id="lineMapper"
2.          class="org.springframework.batch.item.file.mapping.
3.              DefaultLineMapper" >
4.              <bean:property name="lineTokenizer" ref="delimitedLineTokenizer" />
5.              <bean:property name="fieldSetMapper"
6.                  ref="creditBillBeanWrapperFieldSetMapper"/>
7.      </bean:bean>
8.
9.      <bean:bean id="delimitedLineTokenizer"
10.         class="org.springframework.batch.item.file.transform.
11.             DelimitedLineTokenizer">
12.             <bean:property name="delimiter" value=","/>
13.             <bean:property name="names" value="accountID, name, amount, date,
14.                 address" >
15.         </bean:bean>
16.     <bean:bean id="creditBillBeanWrapperFieldSetMapper"
17.         class="org.springframework.batch.item.file.mapping.
18.             BeanWrapperFieldSetMapper" >
19.             <bean:property name="prototypeBeanName" value="creditBill" />
20.         </bean:bean>
```

接下来详细介绍接口 LineTokenizer 与 FieldSetMapper<T>。

6.2.5.1 LineTokenizer

LineTokenizer 接口负责将一条记录转换 FieldSet 对象。接口仅有一个方法 tokenize(), 传入的参数是行的内容, 返回值为 FieldSet 对象。FieldSet 对象中可以认为是一组 key-value 的组合, 负责存放每条记录分割后的数据条目, 条目以 key-value (key 可以认为是字段的名字 name) 的方式存在。FieldSet 接口中定义了读取 value 值的基本类型操作, 以及操作 name 相关的方法。LineTokenizer 接口定义参见代码清单 6-15。FieldSet 接口定义参见代码清单 6-16。

代码清单 6-15 LineTokenizer 接口定义

```
1.  public interface LineTokenizer {
2.      FieldSet tokenize(String line);
3. }
```

代码清单 6-16 FieldSet 接口定义

```
1.  public interface FieldSet {
2.      String[] getNames();
3.      boolean hasNames();
4.      String[] getValues();
5.      String readString(int index);
6.      String readString(String name);
```

```

7.     String readRawString(int index);
8.     String readRawString(String name);
9.     boolean readBoolean ...;
10.    char readChar ...;
11.    byte readByte ...;
12.    short readShort ...;
13.    int readInt ...;
14.    long readLong ...;
15.    float readFloat ...;
16.    double readDouble ...;
17.    BigDecimal readBigDecimal ...;
18.    Date readDate ..;
19.    int getFieldCount();
20.    Properties getProperties();
21. }

```

在 FieldSet 中读取 value 时，支持直接转换为类型 String、Boolean、Char、Byte、Short、Int、Long、Float、Double、BigDecimal、Date。

LineTokenizer 系统实现

Spring Batch 框架中提供了 4 类默认的实现，参见表 6-6。每种实现完成特定的记录转换，如果默认的 4 类实现不能满足要求，读者可以自行定义 LineTokenizer 接口进行数据转换。

表 6-6 LineTokenizer 系统默认实现

LineTokenizer	说 明
DelimitedLineTokenizer	基于分隔符的行转换，根据给定的分隔符将一条记录转换为 FieldSet 对象 org.springframework.batch.item.file.transform.DelimitedLineTokenizer
FixedLengthTokenizer	基于定长数据的行转换，根据给定的数据长度将一条记录转换为 FieldSet 对象 org.springframework.batch.item.file.transform.FixedLengthTokenizer
RegexLineTokenizer	根据正则表达式的条件将一条记录转换为 FieldSet 对象 org.springframework.batch.item.file.transform.RegexLineTokenizer
PatternMatchingCompositeLineTokenizer	可以为不同的记录定义不同的 LineTokenizer；在执行过程中根据给定的标识与每条记录对比，如果满足则用指定的 LineTokenizer；多用于处理同一文件中有不同类型记录的场景 org.springframework.batch.item.file.transform.PatternMatchingCompositeLineTokenizer

LineTokenizer 配置文件

接下来给读者介绍分隔符和定长类型的文件处理。DelimitedLineTokenizer 用来处理分隔符文件，FixedLengthTokenizer 用来处理定长文件。

分隔符文件处理

代码清单 6-17 给出了需要处理的分隔符文件，每行记录用逗号分隔，通过 DelimitedLineTokenizer 可以快速地将每个字段和名称进行映射，代码清单 6-18 给出了如何配置 DelimitedLineTokenizer。

代码清单 6-17 待处理分隔符文件

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,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,Longyang road
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
```

分隔符 DelimitedLineTokenizer 的配置信息如代码清单 6-18 所示。

代码清单 6-18 配置分隔符 DelimitedLineTokenizer

```
1. <bean:bean id="delimitedLineTokenizer"
2.   class="org.springframework.batch.item.file.transform.
3.   DelimitedLineTokenizer">
4.   <bean:property name="delimiter" value=","/>
5.   <bean:property name="names" >
6.     <bean:list>
7.       <bean:value>accountID</bean:value>
8.       <bean:value>name</bean:value>
9.       <bean:value>amount</bean:value>
10.      <bean:value>date</bean:value>
11.      <bean:value>address</bean:value>
12.    </bean:list>
13.  </bean:property>
14. </bean:bean>
```

其中，3 行：属性 delimiter 定义分隔符标识为逗号","分割的字段内容按照顺序和 names 定义的名字匹配。

4~12 行：属性 names 为每个字段定义名字，根据分隔符","获取的每个字段值分别对应 names 中定义的名字。

以第一行数据为例子，描述分隔符字段与 name 之间的匹配关系，参见表 6-7。

表 6-7 分隔符字段与 name 的配置关系

name	行中的字段值
accountID	4047390012345678
name	tom
amount	100.00
date	2013-2-2 12:00:08
address	Lu Jia Zui road

定长类型文件处理

代码清单 6-19 给出了需要处理的定长文件，每行记录长度一致，字段间不足的使用空格补齐，每个字段的固定长度为 1-16,17-26,27-34,35-53,54-72。通过 FixedLengthTokenizer 可以快速地将每个字段和名称进行映射，代码清单 6-20 给出了如何配置 FixedLengthTokenizer。

代码清单 6-19 待处理定长类型文件

1.	4041390012345678tom	00100.002013-02-02 12:00:08	Lu Jia Zui road
2.	4042390012345678tom	00320.002013-02-03 10:35:21	Lu Jia Zui road
3.	4043390012345678jerry	00674.702013-02-06 16:26:49	South Linyi road
4.	4044390012345678rose	00793.202013-02-09 15:15:37	Longyang road
5.	4045390012345678bruce	00360.002013-02-11 11:12:38	Longyang road
6.	4046390012345678rachle	00893.002013-02-28 20:34:19	Hunan road

代码清单 6-20 配置定长文件 FixedLengthTokenizer

```
1. <bean:bean id="fixedLengthLineTokenizer"
2.   class="org.springframework.batch.item.file.transform.
3.   FixedLengthTokenizer">
4.   <bean:property name="columns" value="1-16,17-26,27-34,35-53,
5.   54-72"/>
6.   <bean:property name="names" value="accountID,name,amount,date,
7.   address"/>
5. </bean:bean>
```

其中，3 行：属性 columns 定义字段长度，字段顺序和 names 定义的名字匹配。

4 行：属性 names 为每个字段定义名字。

以第一行数据为例子，描述定长字段与 name 之间的匹配关系，参见表 6-8。

表 6-8 定长字段与 name 的配置关系

name	行中的字段值	长度范围
accountID	4047390012345678	1-16
name	tom	17-26
amount	100.00	27-34
date	2013-2-2 12:00:08	35-53
address	Lu Jia Zui road	54-72

说明：定义字段长度时，从 1 开始，而不是从 0 开始。

6.2.5.2 FieldSetMapper

FieldSetMapper 接口定义了如何将 FieldSet 对象转化为领域对象（通常为 Java 对象）。

FieldSetMapper 接口仅有一个方法 mapFieldSet()，传入的参数是 FieldSet 对象，返回值为转换后的领域对象。FieldSetMapper 接口定义参见代码清单 6-21。

代码清单 6-21 FieldSetMapper 接口定义

```
1. public interface FieldSetMapper<T> {  
2.     T mapFieldSet(FieldSet fieldSet) throws BindException;  
3. }
```

FieldSetMapper 系统实现

Spring Batch 框架中提供了 3 类默认的实现，参见表 6-9，每种实现完成特定的数据转换，如果默认的 3 类实现不能满足要求，读者可以自行实现 FieldSetMapper 接口进行数据转换。

表 6-9 FieldSetMapper 系统默认实现

FieldSetMapper	说 明
ArrayFieldSetMapper	将 FieldSet 对象转换为 String[] org.springframework.batch.item.file.mapping.ArrayFieldSetMapper
BeanWrapperFieldSetMapper<T>	将 FieldSet 对象根据名字映射到给定的 Bean 中，需要保证 FieldSet 中的 name 和 Bean 中属性的名称一致 org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper<T>
PassThroughFieldSetMapper	直接返回 FieldSet 对象 org.springframework.batch.item.file.mapping.PassThroughFieldSetMapper

接下来，我们向读者介绍如何使用 BeanWrapperFieldSetMapper 和自定义的 FieldSetMapper。

BeanWrapperFieldSetMapper

BeanWrapperFieldSetMapper 提供了一种方便的方式，将 FieldSet 对象根据名字直接映射成领域对象。如果声明使用 BeanWrapperFieldSetMapper，只需要在属性 prototypeBeanName 中指明需要映射的领域对象即可，在运行期自动将 FieldSet 中 name 与领域对象同名的属性进行赋值操作。

BeanWrapperFieldSetMapper 的配置信息，参见代码清单 6-22。

代码清单 6-22 配置 BeanWrapperFieldSetMapper

```
1. <bean:bean id="creditBillBeanWrapperFieldSetMapper"  
2.     class="org.springframework.batch.item.file.mapping.  
        BeanWrapperFieldSetMapper" >  
3.     <bean:property name="prototypeBeanName" value="creditBill" />  
4. </bean:bean>  
5. <bean:bean id="creditBill" scope="prototype"  
6.     class="com.juxtapose.example.ch06.CreditBill">  
7. </bean:bean>
```

其中，1 行：定义 BeanWrapperFieldSetMapper，将 FieldSet 映射到 creditBill 对象中。

3 行：属性 prototypeBeanName，指定需要映射的领域对象。

5~7 行：定义信用卡对账单类 creditBill，creditBill 对象中的属性和 FieldSet 中的 name 保持一致，BeanWrapperFieldSetMapper 能够自动根据 name 属性将 FieldSet 中的 value 值映射到 creditBill 对象中。

类 com.juxtapose.example.ch06.CreditBill 的定义参见代码清单 6-23。

代码清单 6-23 类 CreditBill 定义

```
1. public class CreditBill {  
2.     private String accountID = "";      /** 银行卡账户 ID */  
3.     private String name = "";          /** 持卡人姓名 */  
4.     private double amount = 0;         /** 消费金额 */  
5.     private String date;             /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */  
6.     private String address;          /** 消费场所 **/  
7. }
```

在实际的任务开发过程中，可以实现自定义的 FieldSetMapper，将 FieldSet 对象转换成实际需要的领域对象。接下来我们向读者介绍如何使用自定义的 FieldSetMapper。

自定义 FieldSetMapper

直接实现接口 FieldSetMapper，可以实现自定义的转换器，下面的代码将 FieldSet 对象直接转换为领域对象 CreditBill。

自定义转换器 CreditBillFieldSetMapper 的代码参见代码清单 6-24。

代码清单 6-24 自定义转换器 CreditBillFieldSetMapper 类

```
1. public class CreditBillFieldSetMapper implements  
2.     FieldSetMapper <CreditBill> {  
3.         public CreditBill mapFieldSet(FieldSet fieldSet) throws BindException {  
4.             CreditBill result = new CreditBill();  
5.             result.setAccountID(fieldSet.readString("accountID"));  
6.             result.setName(fieldSet.readString("name"));  
7.             result.setAmount(fieldSet.readDouble("amount"));  
8.             result.setDate(fieldSet.readString("date"));  
9.             result.setAddress(fieldSet.readString("address"));  
10.        }  
11.    }
```

唯一的实现方法 mapFieldSet()，在操作中将 fieldSet 对象内的值转换为领域对象 CreditBill。代码中使用了 FieldSet 中的 readXXX() 操作，在 readXXX() 操作中传入 name 参数获取基本类型数据。

开发完毕自定义转化器 CreditBillFieldSetMapper 后，配置该类非常简单，只需要声明该 Bean 即可。配置文件参见代码清单 6-25。

代码清单 6-25 配置自定义转换器 CreditBillFieldSetMapper

```
1.     <bean:bean id="creditBillFieldSetMapper"
2.         class="com.juxtapose.example.ch06.flat.CreditBillFieldSetMapper">
3.     </bean:bean>
```

6.2.6 LineCallbackHandler

在 FlatFileItemReader 中定义属性 linesToSkip，表示从文件头开始跳过指定的行；当记录被跳过时，会触发接口 LineCallbackHandler 中的 handleLine() 操作，该操作中能够获取跳过的行内容。

经常使用的场景是将 Flat 文件中的文件头跳过后，直接写到目标文件的文件头中，这可以通过实现 LineCallbackHandler 接口来完成。LineCallbackHandler 接口定义参见代码清单 6-26。

代码清单 6-26 LineCallbackHandler 接口定义

```
1. public interface LineCallbackHandler {
2.     void handleLine(String line);
3. }
```

说明：操作 handleLine() 在每次跳过行时均会被触发。

自定义实现 LineCallbackHandler

实现接口 LineCallbackHandler，将跳过的文件内容写入到目标文件中。

com.juxtapose.example.ch06.flat.CopyHeaderLineCallbackHandler 的实现，参见代码清单 6-27。

代码清单 6-27 类 CopyHeaderLineCallbackHandler 定义

```
1. public class CopyHeaderLineCallbackHandler implements LineCallbackHandler,
2.     FlatFileHeaderCallback {
3.     private String header = "";
4.
5.     public void handleLine(String line) {
6.         this.header = line;
7.     }
8.
9.     public void writeHeader(Writer writer) throws IOException {
10.         writer.write(header);
11.     }
12. }
```

其中，CopyHeaderLineCallbackHandler 同时实现了接口 LineCallbackHandler 和 FlatFileHeaderCallback。接口 FlatFileHeaderCallback 用于定义在 ItemWrite 之前写入特定的文件头，主要的操作是 writeHeader()。

handleLine() 操作负责收集读文件跳过的头记录，writeHeader() 操作负责写入到目标文件中。

配置文件 ch06/job/job-flatfile-copyheader.xml 展示了如何配置回调操作，参见代码清单 6-28。

代码清单 6-28 job-flatfile-copyheader.xml

```
1.   <bean:bean id="copyHeaderItemReader" scope="step"
2.     class="org.springframework.batch.item.file.FlatFileItemReader">
3.       <bean:property name="linesToSkip" value="1"/>
4.       <bean:property name="skippedLinesCallback"
5.         ref="copyHeaderLineCallbackHandler"/>
6.       .....
7.   </bean:bean>
8.   <bean:bean id="copyHeaderLineCallbackHandler"
9.     class="com.juxtapose.example.ch06.flat.
10.      CopyHeaderLineCallbackHandler">
11.   </bean:bean>
12.   <bean:bean id="csvItemWriter"
13.     class="org.springframework.batch.item.file.FlatFileItemWriter" >
14.       <bean:property name="headerCallback"
15.         ref="copyHeaderLineCallbackHandler"/>
16.       .....
17.   </bean:bean>
```

其中，3 行：定义跳过的行数为 1 行。

4 行：定义跳过的记录的触发器为 copyHeaderLineCallbackHandler。

12 行：定义写文件之前，先写入特定的文件头信息。

查看执行后的文件（target/ch06/copyheader/outputFile.csv），参见代码清单 6-29，头文件已经成功写入。

代码清单 6-29 写入后的文件

```
1. accountID,name,amount,date,address
2. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road
3. .....
```

截至本节，读 Flat 文件所用到的核心类和接口基本介绍完毕，后面我们将给出读取不同类型文件的实际使用场景，包括读分隔符文件、读定长文件、读 JSON 文件、读记录跨多行文件、读混合记录文件，在每节将会有完整的示例展示如何使用上面的核心类和接口。

6.2.7 读分隔符文件

分隔符文件使用 Spring Batch 框架提供的默认组件即可轻松地配置完成，使用到的核心组件包括 DelimitedLineTokenizer、CreditBillFieldSetMapper（自定义实现）；其他没有在图 6-6 中体现的均使用了 FlatFileItemReader 的默认属性。

读分隔符文件使用的核心组件类参见图 6-6。

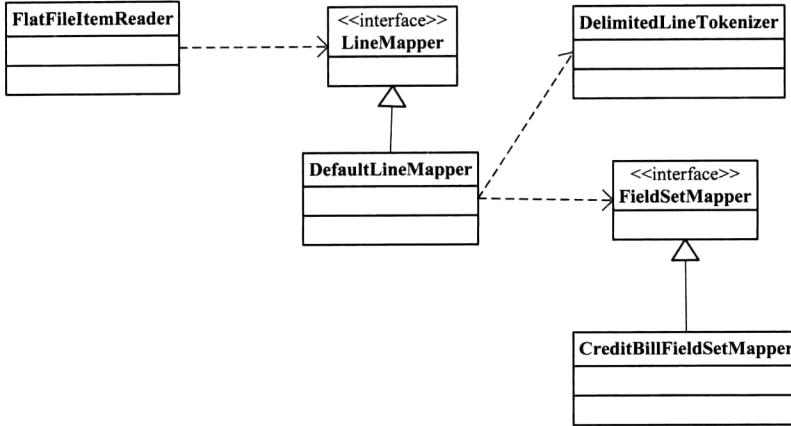


图 6-6 读分隔符文件使用的核心组件类

DelimitedLineTokenizer: 根据分隔符将一条记录转换为 FieldSet 对象。

CreditBillFieldSetMapper: 自定义的 FieldSetMapper，将 FieldSet 对象直接映射为 CreditBill。

自定义转换器 CreditBillFieldSetMapper 的代码，参见代码清单 6-30。

代码清单 6-30 自定义转换器 CreditBillFieldSetMapper 类定义

```

1.  public class CreditBillFieldSetMapper implements
2.      FieldSetMapper <CreditBill> {
3.          public CreditBill mapFieldSet(FieldSet fieldSet) throws BindException {
4.              CreditBill result = new CreditBill();
5.              result.setAccountID(fieldSet.readString("accountID"));
6.              result.setName(fieldSet.readString("name"));
7.              result.setAmount(fieldSet.readDouble("amount"));
8.              result.setDate(fieldSet.readString("date"));
9.              result.setAddress(fieldSet.readString("address"));
10.             return result;
11.         }

```

分隔符格式文件读取配置，参见代码清单 6-31。

代码清单 6-31 配置读分隔符格式文件

```

1.      <bean:bean id="flatFileItemReader" scope="step"
2.          class="org.springframework.batch.item.file.FlatFileItemReader">
3.              <bean:property name="resource"
4.                  value="classpath:ch06/data/flat/credit-card-bill-201303.csv"/>
5.              <bean:property name="lineMapper" ref="lineMapper" />
6.          </bean:bean>
7.

```

```

8.      <bean:bean id="lineMapper"
9.          class="org.springframework.batch.item.file.mapping.
   DefaultLineMapper" >
10.         <bean:property name="lineTokenizer" ref="delimitedLineTokenizer" />
11.         <bean:property name="fieldSetMapper" ref="creditBillFieldSetMapper"/>
12.     </bean:bean>
13.
14.     <bean:bean id="delimitedLineTokenizer"
15.         class="org.springframework.batch.item.file.transform.
   DelimitedLineTokenizer">
16.         <bean:property name="delimiter" value=","/>
17.         <bean:property name="names" value="accountID,name,amount,date,
   address" />
18.     </bean:bean>
19.
20.     <bean:bean id="creditBillFieldSetMapper"
21.         class="com.juxtapose.example.ch06.flat.
   CreditBillFieldSetMapper">
22.     </bean:bean>

```

其中，1~6 行：定义 ItemReader 类，主要包括需要读取的资源和行数据转换类。

8~12 行：定义行数据转换类 DefaultLineMapper，使用分隔符类和自定义的 FieldSetMapper。

14~18 行：定义 DelimitedLineTokenizer，属性 delimiter 字段区分标志，属性 names 定义字段映射的名字。

20~22 行：声明自定义的 CreditBillFieldSetMapper。

6.2.8 读定长文件

定长格式文件使用 Spring Batch 框架提供的默认组件即可轻松地配置完成，使用的核心组件包括 FixedLengthTokenizer、BeanWrapperFieldSetMapper；其他没有在图 6-7 中体现的均使用了 FlatFileItemReader 的默认属性。

读定长文件使用的组件类图参见图 6-7。

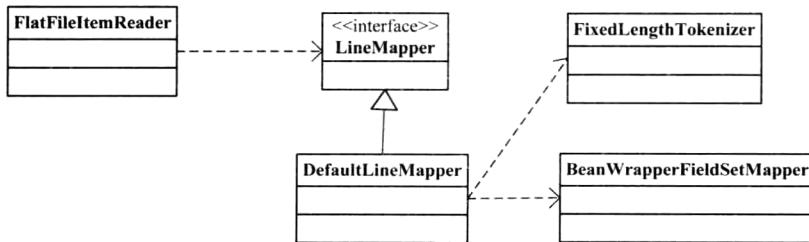


图 6-7 读定长文件使用的组件类

其中，`FixedLengthTokenizer`: 根据字段长度将一条记录转换为 `FieldSet` 对象。

`BeanWrapperFieldSetMapper`: 负责将 `FieldSet` 对象根据 name 映射到指定的 Java Bean 中。定长格式文件读取配置，参见代码清单 6-32。

代码清单 6-32 配置读定长格式文件

```
1. <bean:bean id="fixedLengthItemReader"
2.     class="org.springframework.batch.item.file.FlatFileItemReader">
3.     <bean:property name="resource"
4.         value="classpath:ch06/data/flat/credit-card-bill-fixed-length-
5.             201303.csv"/>
6.     </bean:bean>
7.
8.     <bean:bean id="fixedLengthLineMapper"
9.         class="org.springframework.batch.item.file.mapping.DefaultLine
10.            Mapper">
11.            <bean:property name="lineTokenizer" ref="fixedLengthLineTokenizer"/>
12.            <bean:property name="fieldSetMapper" ref="creditBillBeanWrapper
13.                FieldSetMapper"/>
12.     </bean:bean>
13.
14.     <bean:bean id="fixedLengthLineTokenizer"
15.         class="org.springframework.batch.item.file.transform.
16.             FixedLengthTokenizer">
16.         <bean:property name="columns" value="1-16,17-26,27-34,35-53,54-72"/>
17.         <bean:property name="names" value="accountID,name,amount,date,
18.             address"/>
18.     </bean:bean>
19.
20.     <bean:bean id="creditBillBeanWrapperFieldSetMapper"
21.         class="org.springframework.batch.item.file.mapping.
22.             BeanWrapperFieldSetMapper" >
22.         <bean:property name="prototypeBeanName" value="creditBill" />
23.     </bean:bean>
```

其中，1~6 行：定义 `ItemReader` 类，主要包括需要读取的资源和行数据转换类。

8~12 行：定义行数据转换类 `DefaultLineMapper`，使用定长的分隔符类和 Bean 自动装配的 `FieldSetMapper`。

14~18 行：定义 `FixedLengthTokenizer`，属性 `columns` 主要定义每个字段的长度，属性 `names` 定义字段映射的名字。

20~23 行：属性 `prototypeBeanName` 指定了映射的 Java Bean 对象，通过 `name` 将 `FieldSet` 中的 `value` 自动映射到 `creditBill` 对象上同名的属性上。

6.2.9 读 JSON 文件

JSON 格式数据在 Java 领域使用非常广泛，使用 Spring Batch 框架提供的默认实现可以方便地实现对 JSON 格式文件的读取。读取 JSON 文件使用到的核心组件类参见图 6-8。

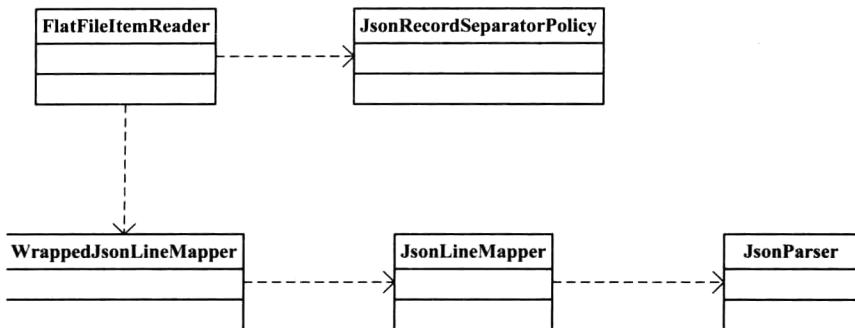


图 6-8 读取 JSON 文件使用到的核心组件类

JsonRecordSeparatorPolicy: Spring Batch 框架提供，根据匹配的 "{}" 分割 JSON 格式文件，每个完整且匹配的部分被认为是一条记录。代码清单 6-33 中的 JSON 文件经过 JsonRecordSeparatorPolicy 拆分后，1~5 行被组装为一条记录，6~10 行被组装为另外一条记录。

代码清单 6-33 JSON 格式文件

```
1. { "accountID": "4047390012345678",
2.   "name": "tom",
3.   "amount": 100.00,
4.   "date": "2013-2-2 12:00:08",
5.   "address": "Lu Jia Zui road"}
6. { "accountID": "4047390012345678",
7.   "name": "tom",
8.   "amount": 320.00,
9.   "date": "2013-2-3 10:35",
10.  "address": "Lu Jia Zui road")
```

其中，**JsonLineMapper:** Spring Batch 框架提供将一条完成的 JSON 记录转换为 Map<String, Object> 对象。在实际使用 JsonLineMapper 时候，可以继续封装 JsonLineMapper，在默认返回 Map<String, Object> 的基础上封装返回特定的领域对象。

WrappedJsonLineMapper: 扩展实现，用于代理 JsonLineMapper，负责将一条记录直接转换为领域对象 CreditBill。WrappedJsonLineMapper 将转换工作委托给 JsonLineMapper，并将 Map 结构转换为 CreditBill。

代码清单 6-34 展示了 WrappedJsonLineMapper 的实现。

代码清单 6-34 WrappedJsonLineMapper 类定义

```
1.  public class WrappedJsonLineMapper implements LineMapper<CreditBill> {
2.      private JsonLineMapper delegate;
3.
4.      public CreditBill mapLine(String line, int lineNumber) throws Exception {
5.          CreditBill result = new CreditBill();
6.          Map<String, Object> creditBillMap = delegate.mapLine(line,
7.              lineNumber);
8.          result.setAccountID((String)creditBillMap.get("accountID"));
9.          result.setName((String)creditBillMap.get("name"));
10.         result.setAmount((Double)creditBillMap.get("amount"));
11.         result.setDate((String)creditBillMap.get("date"));
12.         result.setAddress((String)creditBillMap.get("address"));
13.         return result;
14.     }
15. }
```

其中，2 行：引用 JsonLineMapper，负责完成数据转换。

6 行：通过 JsonLineMapper 获取 Map<String, Object>类型数据。

7~11 行：将 Map<String, Object>结构数据转换为 CreditBill 对象。

JsonParser：Spring Batch 框架中依赖的外部组件，负责实现 JSON 格式数据与 Java 类型数据之间的转换。

JSON 格式文件读取配置，参见代码清单 6-35。

代码清单 6-35 配置读 JSON 格式文件

```
1.      <bean:bean id="jsonItemReader"
2.          class="org.springframework.batch.item.file.FlatFileItemReader">
3.              <bean:property name="resource"
4.                  value="classpath:ch06/data/flat/credit-card-bill-201303.
5.                      json"/>
6.              <bean:property name="recordSeparatorPolicy"
7.                  ref="jsonRecordSeparatorPolicy"/>
8.              <bean:property name="lineMapper" ref="wrappedJsonLineMapper"/>
9.          </bean:bean>
10.
11.         <bean:bean id="jsonRecordSeparatorPolicy"
12.             class="org.springframework.batch.item.file.separator.
13.                 JsonRecordSeparatorPolicy"/>
14.         <bean:bean id="wrappedJsonLineMapper"
15.             class="com.juxtapose.example.ch06.flat.WrappedJsonLineMapper">
16.                 <bean:property name="delegate" ref="jsonLineMapper"/>
17.             </bean:bean>
```

```
15.      <bean:bean id="jsonLineMapper"
16.          class="org.springframework.batch.item.file.mapping.
           JsonLineMapper"/>
```

其中，5 行：声明使用 JSON 的记录分隔策略，使用 Bean 对象 jsonRecordSeparatorPolicy。

6 行：声明包装后的 JSON 数据转换器 WrappedJsonLineMapper，负责将 JSON 格式数据转换为 BillCredit 对象。

9~10 行：定义 JSON 格式的记录分割策略，使用类 JsonRecordSeparatorPolicy。

11~14 行：声明包装的行匹配策略 WrappedJsonLineMapper，数据转换由给 Spring Batch 框架提供的 jsonLineMapper 来表现。

15~16 行：声明 Spring Batch 框架自带的 JsonLineMapper，将 JSON 格式数据转换为 Map<String, Object>。

6.2.10 读记录跨多行文件

当 Flat 文件格式非标准时，通过实现记录分隔策略接口 RecordSeparatorPolicy 来实现非标准 Flat 格式文件。非标准的 Flat 文件格式有多种情况，例如记录跨多行、以特定的字符开头、以特定的字符结尾等。在代码清单 6-36 示例中每两行表示一条记录。

代码清单 6-36 记录跨多行（两行表示一条记录）

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08
2. ,Lu Jia Zui road
3. 4047390012345678,tom,320.00,2013-2-3 10:35:21
4. ,Lu Jia Zui road
```

默认的记录分隔策略 SimpleRecordSeparatorPolicy 或者 DefaultRecordSeparatorPolicy 已经不能处理此类文件。我们可以通过实现接口 RecordSeparatorPolicy，来自定义记录分隔策略 MultiLineRecordSeparatorPolicy。

读记录跨多行文件时，使用到的核心组件类图参见图 6-9。

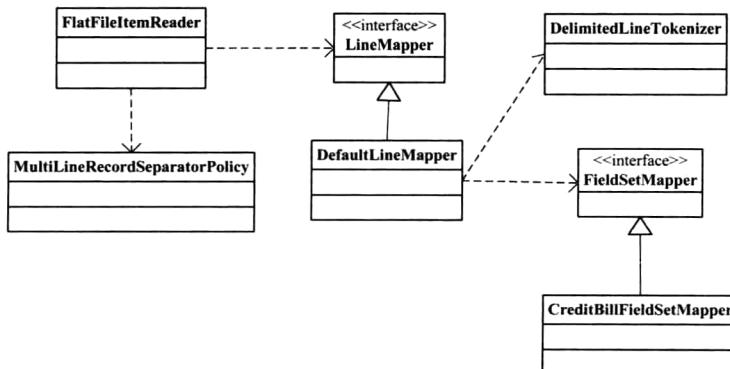


图 6-9 读记录跨多行文件时，使用到的核心组件类

在本类图中，除了 MultiLineRecordSeparatorPolicy 是自定义实现的，其他组件在前面章节已经详细介绍了，如有不明白的请参见前面的章节，此节不再赘述。

MultiLineRecordSeparatorPolicy：负责从文件中确认一条完整记录，在本实现中每读到指定的 4 个逗号分隔符，则认为是一条完整记录。

MultiLineRecordSeparatorPolicy 类定义参见代码清单 6-37。

代码清单 6-37 MultiLineRecordSeparatorPolicy 类定义

```
1. public class MultiLineRecordSeparatorPolicy implements
2.     RecordSeparatorPolicy {
3.     private String delimiter = ",";
4.     private int count = 0;
5.
6.     public boolean isEndOfRecord(String line) {
7.         return countDelimiter(line) == count;
8.     }
9.
10.    public String postProcess(String record) {
11.        return record;
12.    }
13.
14.    public String preProcess(String record) {
15.        return record;
16.    }
17.    private int countDelimiter(String s) {
18.        String tmp = s;
19.        int index = -1;
20.        int count = 0;
21.        while ((index=tmp.indexOf(","))!=-1) {
22.            tmp = tmp.substring(index+1);
23.            count++;
24.        }
25.        return count;
26.    }
27. }
```

其中，2 行：定义读取的分隔符号。

3 行：分隔符总数，给定的字符串包含的分隔符个数等于此值，则认为是一条完整记录。

5~6 行：定义一条记录的完整规则。

17~25 行：统计给定内容包含分隔符的个数。

记录跨多行文件读取的配置，参见代码清单 6-38。

代码清单 6-38 配置读记录跨多行文件

```
1. <bean:bean id="multiLineItemReader"
2.     class="org.springframework.batch.item.file.FlatFileItemReader">
3.     <bean:property name="resource"
```

```

4.           value="classpath:ch06/data/flat/credit-card-bill-multiline-
5.           201303.csv"/>
6.       <bean:property name="recordSeparatorPolicy"
7.           ref="multiLineRecordSeparatorPolicy"/>
8.       <bean:property name="lineMapper" ref="lineMapper"/>
9.   </bean:bean>
10.
11.  <bean:bean id="multiLineRecordSeparatorPolicy"
12.      class="com.juxtapose.example.ch06.flat.
13.          MultiLineRecordSeparatorPolicy">
14.      <bean:property name="delimiter" value=","/>
15.      <bean:property name="count" value="4"/>
16.  </bean:bean>
17.
18.  <bean:bean id="lineMapper"
19.      class="org.springframework.batch.item.file.mapping.
20.          DefaultLineMapper" >
21.      <bean:property name="lineTokenizer" ref="lineTokenizer" />
22.      <bean:property name="fieldSetMapper" ref="creditBillFieldSetMapper"/>
23.  </bean:bean>
24.
25.  <bean:bean id="lineTokenizer"
26.      class="org.springframework.batch.item.file.transform.
27.          DelimitedLineTokenizer">
28.      <bean:property name="delimiter" value=","/>
29.      <bean:property name="names" value="accountID,name,amount,date,address"/>
30.  </bean:bean>
31.
32.  <bean:bean id="creditBillFieldSetMapper"
33.      class="com.juxtapose.example.ch06.flat.CreditBillFieldSetMapper">
34.  </bean:bean>

```

其中，1~7 行：声明 FlatItemReader，定义读取文件资源 credit-card-bill-multiline-201303.csv、行分隔策略、行转换策略。

9~13 行：声明多行的分隔策略，分隔符为逗号，4 个分隔符为一条完整记录。

15~19 行：声明行转换策略，将 2 行记录转换为 CreditBill 对象。

6.2.11 读混合记录文件

通常 Flat 文件中的记录格式是一致的，在特殊情况下一个 Flat 文件中存在多种不同的记录格式，通过特定的开头可以区分不同的记录。例如下面的文件，以 40 开头的为信用卡消费记录，以 30 开头的为借记卡消费记录情况。

混合记录文件格式参见代码清单 6-39。

代码清单 6-39 混合记录文件格式

```
1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 3047390012345671,249.10,rose,2013/4/1 11:34:56
3. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road
4. 3047390012345671,249.10,rose,2013/4/1 11:34:56
5. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road
6. 3047390012345673,840.20,marry,2013/4/3 3:21:43
7. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
```

Spring Batch 框架对文件中存在不同的记录格式同样有友好的支持,通过复杂数据转换类 PatternMatchingCompositeLineMapper, 可以为不同的记录定义不同的 LineTokenizer 和 FieldSetMapper<T>来实现数据转换; 在执行过程中根据每条记录的内容找到匹配的 LineTokenizer 和 FieldSetMapper<T>进行数据转换。

混合记录读的架构参见图 6-10。

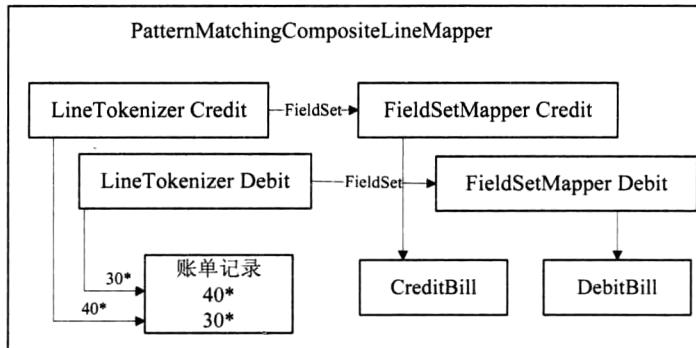


图 6-10 混合记录读的架构图

混合记录读主要类图参见图 6-11。

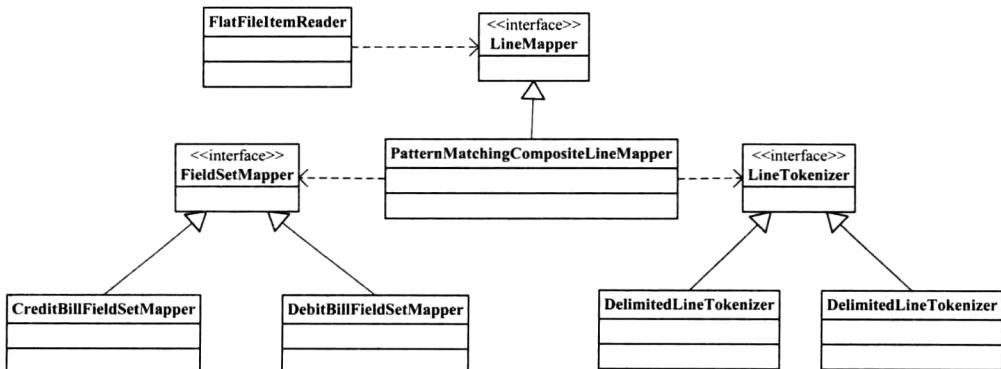


图 6-11 混合记录读主要类图

在图 6-11 中，`PatternMatchingCompositeLineMapper`：为不同的记录准备不同的`LineTokenizer` 和 `FieldSetMapper<T>`来实现数据转换。

`CreditBillFieldSetMapper`：信用卡数据转换类，将 40* 格式的记录转换为领域对象 `CreditBill`。

`DebitBillFieldSetMapper`：借货卡数据转换类，将 30* 格式的记录转换为领域对象 `DebitBill`。

`DelimitedLineTokenizer`：所有的记录是分隔符的，使用分隔符的字段分隔器来完成。

读混合记录文件配置参见代码清单 6-40。

代码清单 6-40 配置读混合记录文件

```
1. <bean:bean id="heterogeneousItemReader"
2.   class="org.springframework.batch.item.file.FlatFileItemReader">
3.     <bean:property name="resource"
4.       value="classpath:ch06/data/flat/credit-card-bill-
5.         heterogeneous-201303.csv"/>
6.     </bean:bean>
7.
8.     <bean:bean id="lineMapper" class="org.springframework.batch.item.
9.       file.mapping.PatternMatchingCompositeLineMapper">
10.       <bean:property name="tokenizers">
11.         <bean:map>
12.           <bean:entry key="40*" value-ref="creditBillRecordTokenizer"/>
13.           <bean:entry key="30*" value-ref="debitBillRecordTokenizer"/>
14.         </bean:map>
15.       </bean:property>
16.       <bean:property name="fieldSetMappers">
17.         <bean:map>
18.           <bean:entry key="40*" value-ref="creditBillFieldSetMapper"/>
19.           <bean:entry key="30*" value-ref="debitBillFieldSetMapper"/>
20.         </bean:map>
21.       </bean:property>
22.     </bean:bean>
```

其中，8~9 行：声明行转换策略 `PatternMatchingCompositeLineMapper`，有两个关键属性 `tokenizers` 和 `fieldSetMappers`；两个属性对应的类型分别是 `Map<String, LineTokenizer>` 和 `Map<String, FieldSetMapper<T>>`。

10~15 行：声明属性 `tokenizers`，以 40 开头的通过 `creditBillRecordTokenizer` 处理记录，以 30 开头的通过 `debitBillRecordTokenizer` 处理记录。

16~21 行：声明属性 `fieldSetMappers`，以 40 开头的通过 `creditBillFieldSetMapper` 将记录转换为 `CreditBill` 对象，以 30 开头的通过 `debitBillFieldSetMapper` 将记录转换为 `DebitBill` 对象。

下面给出了不同记录的转换配置参见代码清单 6-41。

代码清单 6-41 配置不同记录的格式转换

```
1.      <bean:bean id="creditBillRecordTokenizer" parent="parentLineTokenizer">
2.          <bean:property name="names" value="accountID,name,amount,date,
   address" />
3.      </bean:bean>
4.      <bean:bean id="debitBillRecordTokenizer" parent="parentLineTokenizer">
5.          <bean:property name="names" value="accountID,amount,name,date" />
6.      </bean:bean>
7.      <bean:bean id="parentLineTokenizer" abstract="true"
8.          class="org.springframework.batch.item.file.transform.
   DelimitedLineTokenizer">
9.          <bean:property name="delimiter" value=","/>
10.     </bean:bean>
11.
12.     <bean:bean id="creditBillFieldSetMapper"
13.         class="com.juxtapose.example.ch06.flat.CreditBillFieldSetMapper">
14.     </bean:bean>
15.     <bean:bean id="debitBillFieldSetMapper"
16.         class="com.juxtapose.example.ch06.flat.DebitBillFieldSetMapper">
17.     </bean:bean>
```

其中，1~10：定义不同记录转换为 FieldSet 对象。

12~18 行：定义不同记录将 FieldSet 对象转换为 CreditBill 或者 DebitBill 对象。

说明：混合记录文件之后可以将信用卡记录、借贷卡记录写入同一个文件也可以写入不同的文件中，上面例子中将两种类型的记录写入同一文件，如何写入不同的文件请参见写数据章节。

6.3 XML 格式文件

6.3.1 XML 解析

XM 是一种通用的数据交换格式，它的平台无关性、语言无关性、系统无关性，给数据集成与交互带来了极大的方便。XML 在 Java 领域的解析方式有两种，一种叫 SAX，另一种叫 DOM。SAX 是基于事件流的解析，DOM 是基于 XML 文档树结构的解析。

DOM 解析

DOM 解析器读入整个 XML 文档后构建一个驻留内存的树结构，然后代码就可以使用 DOM 接口来操作这个树结构。

优点：整个文档树在内存中，便于操作；支持删除、修改、重新排列等多种功能。

缺点：将整个文档调入内存（包括无用的节点），浪费时间和空间。

使用场合：一旦解析了文档还需多次访问这些数据。

SAX 解析

为解决 DOM 占用内存过大的问题，SAX 采用事件驱动的方式解析 XML 文档。当解析器发现元素开始、元素结束、文本、文档的开始或结束等时发送事件，开发者可以编写响应这些事件的代码，保存数据。

优点：不用事先调入整个文档，占用资源少。

缺点：不是持久的；事件过后，若没保存数据，那么数据就丢了；无状态性；从事件中只能得到文本，但不知该文本属于哪个元素。

使用场合：只需 XML 文档的少量内容，很少重复多次访问。

说明：DOM 解析和 SAX 解析都不适用于批处理的 XML 中的解析，因为两种处理方式都没有支持 Stream。下面我们将介绍一种支持 Stream 方式的 XML 解析方式。

StAX 解析

StAX（The Streaming API for XML），是一种利用拉模式解析（pull-parsing）XML 文档的 API。StAX 把重点放在流上，它提供了两套处理 XML 的 API：一种基于指针的 API，把 XML 文档当作一个标记（或事件）流来处理；允许应用程序检查解析器的状态，获得解析的上一个标记的信息，然后再处理下一个标记，依此类推；另一种较为高级的是基于迭代器的 API，把 XML 作为一系列事件对象来处理，每个对象和应用程序交换 XML 结构的一部分。应用程序根据需要定制解析事件的类型，然后将其转换成对应的具体类型，再然后利用定制事件提供的方法获得属于该事件的信息。

StAX 工作原理是通过一种基于事件迭代器（Iterator）的 API 让程序员去控制 xml 文档解析过程，程序遍历这个事件迭代器去处理每一个解析事件，解析事件可以看作是程序拉出来的，也就是程序促使解析器产生一个解析事件然后处理该事件，之后又促使解析器产生下一个解析事件，如此循环直到碰到文档结束符为止。

StAX 和 SAX 的区别如下。

SAX 也是基于事件处理的 XML 文档，但却是用推模式解析，解析器解析完整个 XML 文档后才产生解析事件，然后推给程序去处理这些事件。SAX 中解析器是工作主体，而事件处理器是由解析器驱动的，如果解析文档过程中产生问题，则剩余的所有文档就无法处理。

而 StAX 使用拉模式，解析器首先将 XML 文档所有的事件全部取出，然后通过处理程序处理这些事件。StAX 中处理器是工作主体，如果解析文档过程中产生问题，只会影响到出问题的部分，而其余部分处理不受影响。

6.3.2 Spring OXM

OMX 是 Object-to-XML-Mapping 的缩写，它是一个 O/X-mapper，负责将 Java 对象映射

为 XML 数据，或者将 XML 数据映射为 java 对象。

Spring framework 在 3.0 版本中引入了该特性，Spring O/X Mapper 仅定义由流行的第三方框架实现的统一的切面。要使用 Spring 的 O/X Mapper 功能，需提供一个在能在 Java 对象和 XML 之间转换的组件。通常这样的组件有 Castor、XMLBeans、Java Architecture for XML Binding (JAXB)、JiBX 和 XStream 等。

Spring OXM 框架参见图 6-12。

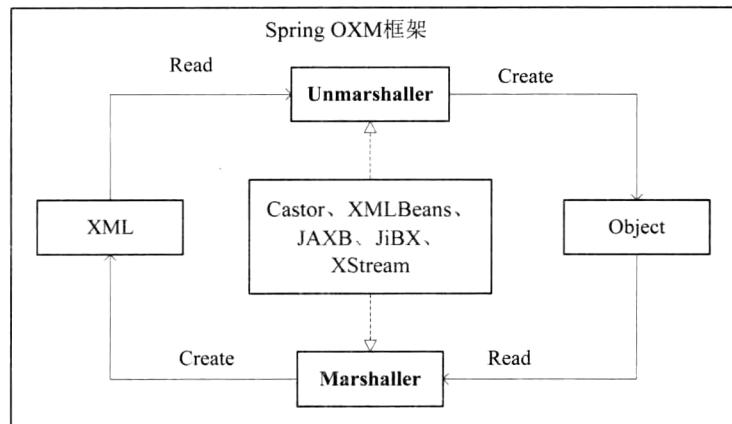


图 6-12 Spring OXM 框架图

图 6-12 中 OXM 框架通过 Unmarshaller 接口将 XML 文本反序列化为 Object 对象，通过接口 Marshaller 将 Object 对象序列化为 XML 文本。同时 Spring OXM 框架提供了内置的序列化和反序列化组件，包括 CastorMarshaller、JibxMarshaller、XmlBeansMarshaller、XStreamMarshaller、Jaxb2Marshaller 等。

编组、解组

编组或者称之为序列化 (serializing)，是指将 Java Bean 转换成 XML 文档的过程，这意味着将 Java Bean 中的所有字段和字段值都将作为 XML 元素或属性填充到 XML 文件中。

解组或者称之为反序列化 (deserializing)，是与编组完全相反的过程，即将 XML 文档转换为 Java Bean，这意味着 XML 文档的所有元素或属性都作为 Java 字段填充到 Java Bean 中。

Spring O/X Mapper 优点

- 易于配置

Spring 的 O/X Mapper 采用标准 Spring 框架的配置方式简化开发，Spring 的 Bean 库支持将实例化的 O/X 编组器注入那些使用编组器的对象。

- 一致的接口

Spring 的 O/X Mapper 框架统一提供两个接口：Marshaller 和 Unmarshaller，用于执行 O/X

功能。这些接口的实现完全对开发人员开放，开发人员可以轻松地切换它们而无须修改任何代码。例如，如果你一开始使用 Castor 进行 O/X 转换，但后来发现它缺乏你需要的某个功能，这时可以切换到 XStream 而无须任何代码更改（只需要做的是修改 Spring 配置文件以使用新的 O/X 框架）。

- 一致的异常层次结构

Spring 的 O/X Mapper 的另一个好处是统一的异常层次结构。Spring 框架将原始异常对象包装到 Spring 自身专为 O/X Mapper 建立的运行时异常（XMLMappingException）中。由于第三方提供商抛出的原始异常被包装到 Spring 运行时异常中，所以能够查明出现异常的根本原因。Spring 的 O/M Mapper 框架提供的异常包括 GenericMarshallingFailureException、ValidationFailureException、MarshallingFailureException、UnmarshallingFailureException

6.3.3 StaxEventItemReader

StaxEventItemReader 实现 ItemReader 接口，核心作用是将 XML 文件中的记录转换为 Java 对象。StaxEventItemReader 通过引用 OXM 组件完成对 XML 的读操作，负责将 XML 文件转换为 Java 对象，并交给处理或者写阶段。

StaxEventItemReader 结构关键属性

图 6-13 展示了 XML 文件读取的逻辑架构图，XMLEventReader 负责将文件按照 StaX 的模式读取指定的节点数据，然后交给反序列化组件 Unmarshaller 完成 Java 对象的生成。

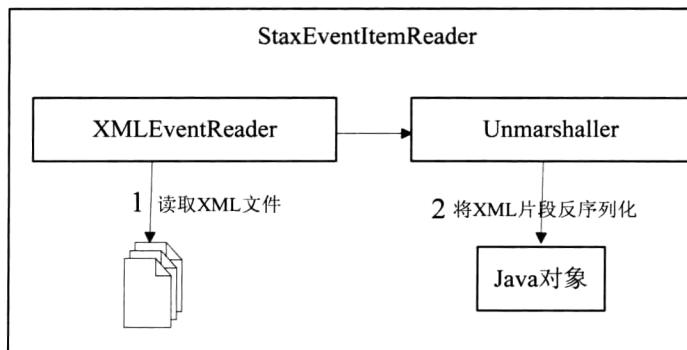


图 6-13 XML 文件读取的逻辑架构图

StaxEventItemReader 核心类架构图参见图 6-14。

StaxEventItemReader 中的核心类或者接口包括 Resource、Unmarshaller、XMLEventReader、FragmentEventReader、DefaultFragmentEventReader。Resource 表示需要处理的文件资源；Unmarshaller 负责将 XML 片段转换为 Java 对象，即进行反序列化；XMLEventReader 接口负责提供 StAX 方式对 XML 的解析；FragmentEventReader 接口继承接口 XMLEventReader，在

XMLEventReader 能力的基础上增加了将 XML 片段转换为 Document 能力，即增加了 StartDocument、EndDocument 两个事件；DefaultFragmentEventReader 是接口 FragmentEvent Reader 的默认实现，同时引用 XMLEventReader 对象，负责具体的 XML 处理工作。

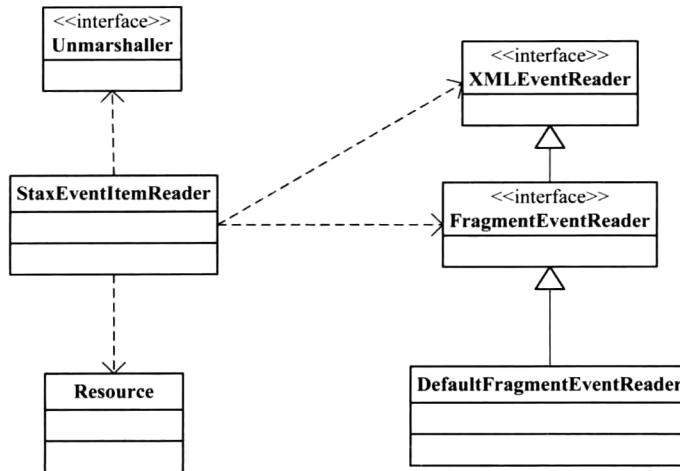


图 6-14 StaxEventItemReader 核心类图

在实际配置 StaxEventItemReader 时，只需要配置 Unmarshaller、Resource 两个属性即可。StaxEventItemReader 关键属性参见表 6-10。

表 6-10 StaxEventItemReader 关键属性

StaxEventItemReader 属性	类 型	说 明
fragmentRootElementName	String	需要转换为 Java 对象的根节点的名字
fragmentRootElementNameSpace	String	需要转换为 Java 对象的根节点的命名空间
resource	Resource	需要读取的资源文件
maxItemCount	String	能读取的最大条目数 默认值: Integer.MAX_VALUE
strict	Boolean	定义读取文件不存在时候的策略, 如果为 true 则抛出异常, 如果为 false 表示不抛出异常。 默认值: true
unmarshaller	Int	Spring OXM 实现类, 负责将 XML 内容转换为 Java 对象

配置 StaxEventItemReader

在给出详细配置文件之前，我们首先给出 XML 文件示例，参见代码清单 6-42，它通过 XML 的方式描述来信用卡的消费清单。

完整文件路径：ch06/data/xml/credit-card-bill-201303.xml。

代码清单 6-42 XML 示例文件

```
1.  <credits>
2.    <credit>
3.      <accountID>4047390012345678</accountID>
4.      <name>tom</name>
5.      <amount>100.00</amount>
6.      <date>2013-2-2 12:00:08</date>
7.      <address>Lu Jia Zui road</address>
8.    </credit>
9.    <credit>
10.      <accountID>4047390012345678</accountID>
11.      <name>tom</name>
12.      <amount>320.00</amount>
13.      <date>2013-2-3 10:35:21</date>
14.      <address>Lu Jia Zui road</address>
15.    </credit>
16.    .....
17. </credits>
```

信用卡账单文件的根节点为 credits，下面为具体的信用卡消费记录节点 credit，在根节点 credits 下面可以有多个信用卡消费记录 credit。

配置 StaxEventItemReader 参见代码清单 6-43。

完整的配置文件参见：ch06/job/job-xml.xml。

代码清单 6-43 配置 StaxEventItemReader

```
1.  <!-- XML 文件读取 -->
2.  <bean:bean id="xmlReader" scope="step"
3.    class="org.springframework.batch.item.xml.StaxEventItemReader">
4.    <bean:property name="fragmentRootElementName" value="credit"/>
5.    <bean:property name="unmarshaller" ref="creditMarshaller"/>
6.    <bean:property name="resource"
7.      value="classpath:ch06/data/xml/credit-card-bill-201303.xml"/>
8.  </bean:bean>
9.  <bean:bean id="creditMarshaller"
10.    class="org.springframework.oxm.xstream.XStreamMarshaller">
11.    <bean:property name="aliases">
12.      <util:map id="aliases">
13.        <bean:entry key="credit"
14.          value="com.juxtapose.example.ch06.CreditBill"/>
15.      </util:map>
16.    </bean:property>
17.  </bean:bean>
18.  <bean:bean id="creditBill" scope="prototype"
19.    class="com.juxtapose.example.ch06.CreditBill"/>
```

其中，2~8 行：定义 XML 的读取类，主要属性包括 fragmentRootElementName、unmarshaller、resource；fragmentRootElementName 属性值为“credit”，表示读取 XML 文件中的节点为 credit 的元素转换为 Java 对象；unmarshaller 属性定义序列化组件；resource 属性定义读取的文件资源。

9~19 行：定义 XML 序列化的组件，此处使用 XStream 的方式进行序列化，使用 Spring OXM 提供的组件 XStreamMarshaller 进行序列化；只需要指定类别名 aliases 属性就可以将 XML 文件转换为特定的 Java 对象，此处使用 com.juxtapose.example.ch06.CreditBill，即将 credits/credit 路径下的 XML 文件内容转换为 CreditBill 对象。

6.4 读多文件

前面章节完整地介绍了对 Flat 格式、XML 格式文件的读取操作，细心的读者会发现上面所有的文件读取基本上是对单文件执行的。在实际的应用中，我们需要经常操作批量的文件，本节将为读者介绍如何对多文件的读取。以信用卡账单为例，在处理信用卡账单时，每月都会产生账单记录，在应用中期望一次处理多个账单文件：3、4、5 月份的信用卡账单，参见图 6-15。

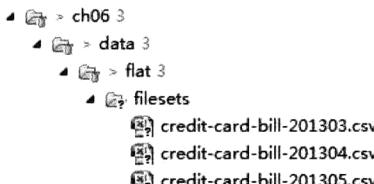


图 6-15 多文件处理示例

Spring Batch 框架提供了现有的组件 MultiResourceItemReader 支持对多文件的读取，通过 MultiResourceItemReader 读取批量文件非常简单。MultiResourceItemReader 通过代理的 ItemReader 来读取文件。

MultiResourceItemReader 结构关键属性

MultiResourceItemReader 组件实现 ItemReader、ItemStream 接口，并引用实现了接口 ResourceAwareItemReaderItemStream 的读组件，例如 FlatFileItemReader、StaxEventItemReader，通过这些具体的组件完成文件的读取。

MultiResourceItemReader 在处理文件的时候在一个线程中按照读取的文件顺序执行，在某些场景直接使用 MultiResourceItemReader 可能无法满足性能要求；如果需要高性能地处理海量数据可以通过 Spring Batch 框架提供的分区能力，并行计算来处理海量文件，分区请参照并行处理章节的介绍。

MultiResourceItemReader 核心类结构参见图 6-16。

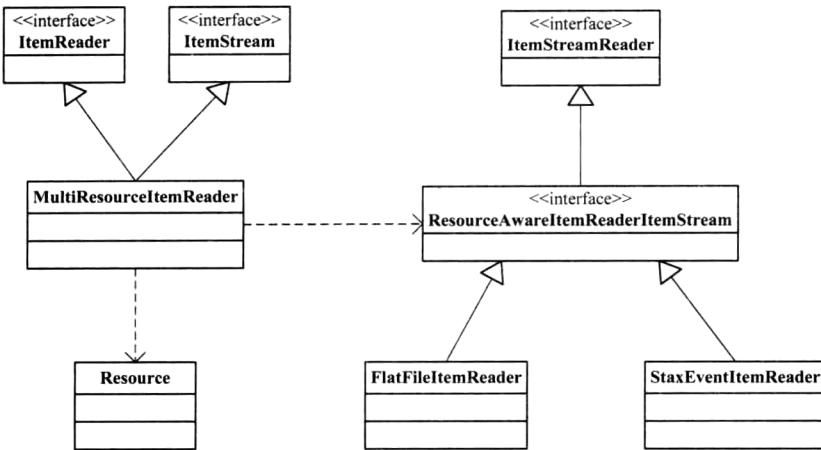


图 6-16 MultiResourceItemReader 核心类

MultiResourceItemReader 关键属性参见表 6-11。

表 6-11 MultiResourceItemReader 关键属性

MultiResourceItemReader 属性	类 型	说 明
delegate	ResourceAwareItemReaderItemStream	ItemReader 的代理, 将 resources 中定义的文件代理给当前指定的 ItemReader 进行处理
resources	Resource[]	需要读取的资源文件列表
strict	boolean	定义读取文件不存在时候的策略, 如果为 true 则抛出异常, 如果为 false 表示不抛出异常。 默认值为 true
saveState	boolean	保存状态标识, 读取资源时候是否保存当前读取的文件及当前文件是否读取条目记录的状态。 默认值为 true

配置 MultiResourceItemReader

在给出详细配置文件之前, 我们首先给出多文件示例 (参见代码清单 6-44、代码清单 6-45、代码清单 6-46), 本例中期望将目录 ch06/data/flat/filesets/下面所有以“credit-card-bill-”开头, 以 “.csv” 结尾的文件全部读取进行处理。下面给出了文件 credit-card-bill-201303.csv、credit-card-bill-201304.csv, credit-card-bill-201305.csv 的内容, 每个文件均有 2 条消费记录; 经过处理后, 将记录汇集在一个目标文件中。

完整的文件路径: ch06/data/flat/filesets/credit-card-bill-201303.csv。

代码清单 6-44 示例文件 credit-card-bill-201303.csv

1. 4047390012345678,tom,100.00,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.00,2013-2-3 10:35:21,Lu Jia Zui road

完整文件路径：ch06/data/flat/filesets/credit-card-bill-201304.csv。

代码清单 6-45 示例文件 credit-card-bill-201304.csv

```
1. 4047390012345678,tom,674.70,2013-2-6 16:26:49,South Linyi road  
2. 4047390012345678,tom,793.20,2013-2-9 15:15:37,Longyang road
```

完整文件路径：ch06/data/flat/filesets/credit-card-bill-201305.csv。

代码清单 6-46 示例文件 credit-card-bill-201305.csv

```
1. 4047390012345678,tom,360.00,2013-2-11 11:12:38,Longyang road  
2. 4047390012345678,tom,893.00,2013-2-28 20:34:19,Hunan road
```

配置 MultiResourceItemReader 参见代码清单 6-47。

完整配置文件参见：ch06/job/job-flatfile.xml。

代码清单 6-47 配置 MultiResourceItemReader

```
1. <job id="fileSetsJob">  
2.   <step id="fileSetsStep">  
3.     <tasklet transaction-manager="transactionManager">  
4.       <chunk reader="multiResourceReader" writer="csvItemWriter"  
5.         commit-interval="2">  
6.       </chunk>  
7.     </tasklet>  
8.   </step>  
9. </job>  
10.  
11. <bean:bean id="multiResourceReader"  
12.   class="org.springframework.batch.item.file.  
13.     MultiResourceItemReader">  
14.       <bean:property name="resources"  
15.         value="classpath:/ch06/data/flat/filesets/  
16.           credit-card-bill-*.*.csv"/>  
17.       <bean:property name="delegate" ref="flatFileItemReader"/>  
18.     </bean:bean>  
19.     <bean:bean id="flatFileItemReader" scope="step"  
20.       class="org.springframework.batch.item.file.FlatFileItemReader">  
21.         <bean:property name="lineMapper" ref="lineMapper" />  
22.         <bean:property name="strict" value="true"/>
```

其中，1~9 行：定义多文件处理的作业，读取多文件使用 multiResourceReader。

11~16 行：定义多文件读取 multiResourceReader，两个关键属性：resources 和 delegate；resources 属性表示需要读取的文件集合，配置文件的前缀为“credit-card-bill-”，后缀为“.csv”的文件；delegate 属性用于配置具体的文件读取 ItemReader，本例中使用了 FlatFileItemReader

读取 Flat 格式的文件。

18~22 行：定义 flatFileItemReader，读取配置的 CSV 文件。

说明：MultiResourceItemReader 自身的实现不是线程安全的，不能在多线程中并发使用该组件。在一些场景中，文件数量多且文件内容较大，单线程处理不能满足时间或者性能上的要求，使用后面 11 章节介绍的分区功能，将数据分区后，可以使用多个线程甚至多个物理节点来并行处理数据。

6.5 读数据库

众多企业将数据存放在数据库中，友好的批处理框架需要友好的对数据库的读/写支持。Spring Batch 框架对读数据库提供了非常好的支持，包括基于 JDBC 和 ORM（Object-Relational Mapping）的读取方式；基于游标和分页的读取数据的 ItemReader 组件。

Spring Batch 框架提供的读数据库组件列表参见表 6-12。

表 6-12 Spring Batch 框架提供的读数据库组件

JdbcCursorItemReader	基于 JDBC 游标方式读数据库
HibernateCursorItemReader	基于 Hibernate 游标方式读数据库
StoredProcedureItemReader	基于存储过程读数据库
IbatisPagingItemReader	基于 Ibatis 分页读数据库
JpaPagingItemReader	基于 Jpa 方式分页读数据库
JdbcPagingItemReader	基于 JDBC 方式分页读数据库
HibernatePagingItemReader	基于 Hibernate 方式分页读取数据库

基于游标的读

在数据库中，游标是一个十分重要的概念。游标提供了一种对从表中检索出的数据进行操作的灵活手段，就本质而言，游标实际上是一种能从包括多条数据记录的结果集中每次提取一条记录的机制。

Spring Batch 框架提供了三种基于游标的读取方式，分别是：

- 基于 JDBC 的 JdbcCursorItemReader；
- 基于 Hibernate 实现的 HibernateCursorItemReader；
- 基于存储过程的 StoredProcedureItemReader。

基于分页的读

基于游标的数据库读取避免了一次查询大批量的数据导致消耗应用大量的内存，这是由于数据库提供了另外一种能力：基于分页的读，即通过分页读每次获取指定页大小的数据。

Spring Batch 框架对分页读提供了默认的系统读组件。

Spring Batch 框架提供了四种基于分页的读取方式，既包括基于 JDBC 的分页读，也包含基于 ORM 的分页实现；四种分页读分别是：

- 基于 JDBC 的 `JdbcPagingItemReader`；
- 基于 Hibernate 实现的 `HibernatePagingItemReader`；
- 基于 Ibatis 实现的 `IbatisPagingItemReader`；
- 基于 Jpa 实现的 `JdbcPagingItemReader`。

6.5.1 JdbcCursorItemReader

Spring Batch 框架提供了对 JDBC 读取支持的组件 `JdbcCursorItemReader`。`JdbcCursorItemReader` 实现 `ItemReader` 接口，核心作用是将数据库中的记录转换为 Java 对象。`JdbcCursorItemReader` 通过引用 `PreparedStatement`、`RowMapper`、`PreparedStatementSetter` 关键接口实现上面的目的；在 `JdbcCursorItemReader` 将数据库记录转换为 Java 对象时主要有两步工作：首先根据 `PreparedStatement` 从数据库中获取结果集 `ResultSet`；其次使用 `RowMapper` 将结果集 `ResultSet` 转换为 Java 对象，具体步骤见图 6-17。

`JdbcCursorItemReader` 结构及关键属性

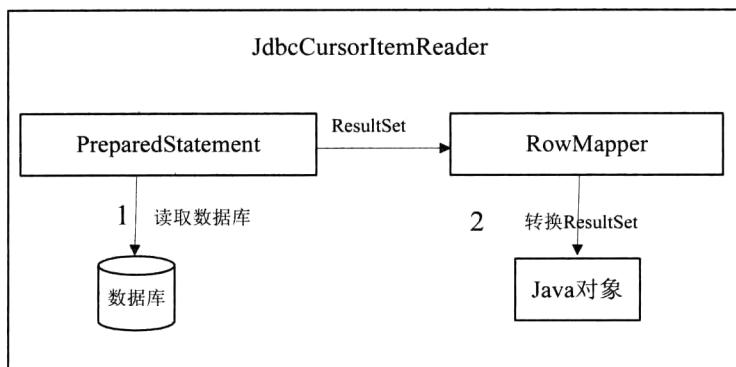


图 6-17 `JdbcCursorItemReader` 核心操作步骤

`JdbcCursorItemReader` 与关键接口 `PreparedStatement`、`PreparedStatementSetter`、`RowMapper` 之间的类图参见图 6-18。

`JdbcCursorItemReader` 引用 `javax.sql.DataSource`，`DataSource` 提供读取的数据库信息；`JdbcCursorItemReader` 通过 `PreparedStatement` 对数据库进行读/写，返回结果集 `ResultSet`；对象 `PreparedStatementSetter` 为读取的 SQL 提供参数设置能力；`RowMapper` 负责将 `ResultSet` 对象转换为 Java 对象。

关键接口、类说明参见表 6-13。

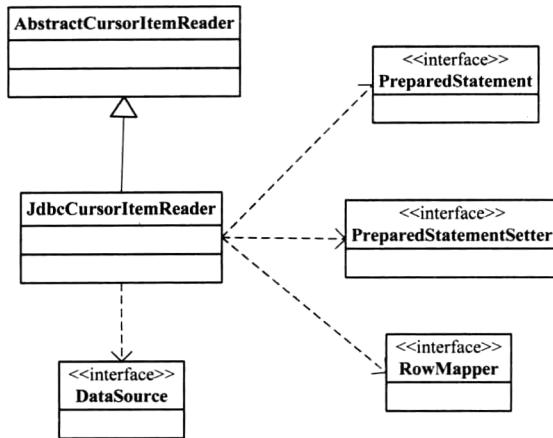


图 6-18 JdbcCursorItemReader 类关系图

表 6-13 关键接口、类说明

关键类	说 明
DataSource	提供读取数据库的数据源信息
PreparedStatement	PreparedStatement 实例包含已编译的 SQL 语句，并提供执行 SQL 语句的能力，返回结果集 ResultSet
PreparedStatementSetter	负责为 PreparedStatement 中的 SQL 提供参数支持
RowMapper	负责将结果集 ResultSet 转换为 Java 对象

JdbcCursorItemReader 关键属性参见表 6-14。

表 6-14 JdbcCursorItemReader 关键属性

JdbcCursorItemReader 属性	类 型	说 明
dataSource	DataSource	数据源，通过该属性指定使用的数据库信息
driverSupportsAbsolute	Boolean	数据库驱动是否支持结果集的绝对定位。 默认值： false
fetchSize	int	设置 ResultSet 每次向数据库取的行数； setFetchSize 的意思是当调用 rs.next 时，ResultSet 会一次性从服务器上取多少行数据回来，这样在下次 rs.next 时，可以直接从内存中获取数据而不需要网络交互，提高了效率。 默认值： -1
ignoreWarnings	Boolean	是否忽略 SQL 执行期间的警告信息， true 表示忽略警告， false 表示会抛出异常。 默认值： true

续表

JdbcCursorItemReader 属性	类 型	说 明
maxRows	int	设置结果集做大行数。 默认值: -1 (表示不限制)
preparedStatementSetter	PreparedStatementSetter	SQL 语句参数准备, 可以使用批处理框架提供的 ListPreparedStatementSetter, 也可以自定义实现该接口
queryTimeout	int	查询超时时间。如果超时, 将抛出超时异常。 默认值: -1 (表示永不超时)
rowMapper	RowMapper	将结果集 ResultSet 转换为指定的 Pojo 对象类; 需要实现 RowMapper 接口; 默认可以使用 BeanPropertyRowMapper
saveState	Boolean	是否将当前 Reader 的状态保存到 job repository 中, 即当前读取到数据库的行数; 在操作 ItemStream#update()中被持久化。 默认值: true
sql	String	需要执行的 SQL
useSharedExtendedConnection	Boolean	不同游标间是否共享数据库连接; 如果共享则必须在同一个事务当中, 否则使用不同的事务。 默认值: false
verifyCursorPosition	Boolean	处理完当前行后, 是否校验游标位置。 默认值: true

配置 JdbcCursorItemReader

使用 JdbcCursorItemReader 至少需要配置 dataSource、sql、rowMapper 三个属性; dataSource 指定访问的数据源, sql 用于指定查询的 SQL 语句, rowMapper 用于将结果集 ResultSet 转换为 Java 业务对象。

在给出详细的配置文件之前, 我们首先准备 SQL (使用的为 MySQL 数据库) 脚本, 示例中使用信用卡账单表, 存放信用卡消费记录情况, 主要字段包括 ID、ACCOUNTID、NAME、AMOUNT、DATE、ADDRESS; 建表脚本参见代码清单 6-48。

完整内容参见文件: ch06/db/create-tables-mysql.sql。

代码清单 6-48 建表脚本 create-tables-mysql.sql

```

1. CREATE TABLE t_credit
2.   (ID VARCHAR(10),
3.    ACCOUNTID VARCHAR(20),
4.    NAME VARCHAR(10),
5.    AMOUNT NUMERIC(10,2),

```

```
6.      DATE VARCHAR(20),  
7.      ADDRESS VARCHAR(128),  
8.      primary key (ID)  
9.    )  
10.   ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

配置 JdbcCursorItemReader 参见代码清单 6-49。

完整配置文件参见：ch06/job/job-db-jdbc.xml。

代码清单 6-49 配置 JdbcCursorItemReader

```
1.  <bean:bean id="jdbcItemReader" scope="step"  
2.    class="org.springframework.batch.item.database.  
JdbcCursorItemReader" >  
3.    <bean:property name="dataSource" ref="dataSource"/>  
4.    <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,  
5.    DATE,ADDRESS from t_credit where id between 1 and 5 "/>  
6.    <bean:property name="rowMapper">  
7.      <bean:bean class="org.springframework.jdbc.core.  
BeanPropertyRowMapper">  
8.        <bean:property name="mappedClass"  
9.          value="com.juxtapose.example.ch06.CreditBill"/>  
10.       </bean:bean>  
11.     </bean:property>  
12.   </bean:bean>
```

其中，3 行：配置访问的数据源。

4 行：定义需要查询的 SQL 语句，查询 t_credit 表，且 id 在 1~5 范围内的记录。

6~10 行：指定 rowMapper 属性，将结果集 ResultSet 转换为 CreditBill 对象；此时使用 Spring 提供的 BeanPropertyRowMapper 完成，通过设置属性 mappedClass，自动将结果集 ResultSet 和 CreditBill 对象映射（即将表中的字段名称映射到 CreditBill 对象的属性中）。

配置数据源

数据库访问需要数据源定义，在 Spring Batch 框架中数据源的配置是标准的 Spring 配置。具体的配置信息参见代码清单 6-50。

代码清单 6-50 配置数据源

```
1.  <context:placeholder location="classpath:/ch06/properties/  
batch-mysql.properties" />  
2.  <bean:bean id="dataSource"  
3.    class="org.springframework.jdbc.datasource.  
DriverManagerDataSource">  
4.    <bean:property name="driverClassName">  
5.      <bean:value>${datasource.driver}</bean:value>
```

```
6.      </bean:property>
7.      <bean:property name="url">
8.          <bean:value>${datasource.url}</bean:value>
9.      </bean:property>
10.     <bean:property name="username" value="${datasource.username}">
11.         </bean:property>
12.     <bean:property name="password" value="${datasource.password}">
13.         </bean:property>
14.     </bean:bean>
```

其中，1行：指定加载的属性配置文件，方便动态指定数据源的属性。

2~12行：配置数据源的具体实现，分别指定属性 driverClassName、url、username、password。

自定义 RowMapper

使用 RowMapper 进行数据映射时，可以使用 Spring 框架提供的 BeanPropertyRowMapper 来自动将数据库的字段映射到指定的 Java Bean 上；另外通过实现接口 org.springframework.jdbc.core.RowMapper<T>，可以自定义 RowMapper 的实现类。RowMapper 接口定义参见代码清单 6-51，根据需要完成自定义的数据转换功能。

代码清单 6-51 RowMapper 接口定义

```
1. public interface RowMapper<T> {
2.     T mapRow(ResultSet rs, int rowNum) throws SQLException;
3. }
```

其中，2行：核心转换方法，参数为结果集和当前游标所在的位置。

本例中使用自定义的信用卡账单转换类 com.juxtapose.example.ch06.db.CreditBillRowMapper。CreditBillRowMapper 的代码实现参见代码清单 6-52。

代码清单 6-52 自定义 RowMapper 实现 CreditBillRowMapper

```
1. public class CreditBillRowMapper implements RowMapper<CreditBill> {
2.
3.     public CreditBill mapRow(ResultSet rs, int rowNum) throws SQLException {
4.         CreditBill bill = new CreditBill();
5.         bill.setAccountID(rs.getString("ACCOUNTID"));
6.         bill.setAddress(rs.getString("ADDRESS"));
7.         bill.setAmount(rs.getDouble("AMOUNT"));
8.         bill.setDate(rs.getString("DATE"));
9.         bill.setName(rs.getString("NAME"));
10.        return bill;
11.    }
12. }
```

其中，3~11行：业务实现代码，将给定的结果集 ResultSet 转化为 CreditBill 对象。

接下来我们使用自定义的 CreditBillRowMapper 配置 JdbcCursorItemReader，参见代码清

单 6-53。CreditBillRowMapper 将数据库中的行记录转换为领域对象。

代码清单 6-53 配置 JdbcCursorItemReader

```
1. <bean:bean id="jdbcItemReader" scope="step"
2.   class="org.springframework.batch.item.database.
JdbcCursorItemReader" >
3.   <bean:property name="dataSource" ref="dataSource"/>
4.   <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
5.   DATE,ADDRESS from t_credit where id between 1 and 5 "/>
6.   <bean:property name="rowMapper" ref="custCreditRowMapper" />
7. </bean:bean>
8. <bean:bean id="custCreditRowMapper"
9.   class="com.juxtapose.example.ch06.db.CreditBillRowMapper"/>
```

其中，6 行：使用自定义的 custCreditRowMapper 来指定 rowMapper 属性，负责数据转换，将结果集 ResultSet 转换为 CreditBill 对象。

8~9 行：声明实现 RowMapper 接口的类 CreditBillRowMapper。

SQL 参数绑定

当 SQL 语句执行过程中需要动态指定参数时，可以使用属性 preparedStatementSetter 来设置，该属性是类型为 PreparedStatementSetter 的对象。

Spring Batch 框架提供了 PreparedStatementSetter 的默认实现 ListPreparedStatementSetter（org.springframework.batch.core.resource.ListPreparedStatementSetter），根据给定的参数顺序设置 SQL 语句的参数。

使用 preparedStatementSetter 属性配置 JdbcCursorItemReader，参见代码清单 6-54。

代码清单 6-54 使用 preparedStatementSetter 配置 JdbcCursorItemReader

```
1. <bean:bean id="jdbcParameterItemReader" scope="step"
2.   class="org.springframework.batch.item.database.
JdbcCursorItemReader" >
3.   <bean:property name="dataSource" ref="dataSource"/>
4.   <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
5.   DATE,ADDRESS from t_credit where id between 1 and ? "/>
6.   <bean:property name="rowMapper" ref="custCreditRowMapper" />
7.   <bean:property name="preparedStatementSetter" ref="paramStatement
Setter"/>
8. </bean:bean>
9.
10. <bean:bean id="paramStatementSetter" scope="step"
11.   class="org.springframework.batch.core.resource.
ListPreparedStatementSetter">
12.   <bean:property name="parameters">
```

```
13.          <bean:list>
14.              <bean:value>#${jobParameters['id']}</bean:value>
15.          </bean:list>
16.      </bean:property>
17.  </bean:bean>
```

其中，4~5 行：注意 SQL 语句变化，此处查找范围是 1 到待定的范围，待定的范围通过后续的属性 `PreparedStatementSetter` 设置。

10~16 行：使用 `ListPreparedStatementSetter` 来设置 SQL 语句需要的参数。

14 行：使用参数后绑定技术来实现，此处使用执行 Job 时候传入的参数 “`id`” 作为 SQL 语句查询的范围。

使用代码清单 6-55 执行定义的 `dbReadJob`。

完整代码参见：test.com/juxtapose/example/ch06/JobLaunchJDBC。

代码清单 6-55 执行 `dbReadJob`

```
1. executeJob("ch06/job/job-db-jdbc.xml", "dbReadJob",
2.             new JobParametersBuilder().addDate("date", new Date())
3.                         .addString("id", "5"));
```

其中，2 行：传入参数 `id=5`，最终 SQL 执行查询的脚本为“`select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS from t_credit where id between 1 and 5`”。

自定义 `PreparedStatementSetter`

上面使用了 Spring 框架自带的 `ListPreparedStatementSetter`，开发者可以根据自己业务需求实现自定义的 `PreparedStatementSetter`，进行参数设置。`PreparedStatementSetter` 接口定义参见代码清单 6-56。

代码清单 6-56 `PreparedStatementSetter` 接口定义

```
1. public interface PreparedStatementSetter {
2.     void setValues(PreparedStatement ps) throws SQLException;
3. }
```

其中，2 行：核心方法，为参数 `PreparedStatement` 设置 SQL 需要的参数。

自定义 `CreditBillPreparedStatementSetter` 类的实现参见代码清单 6-57。

完整代码参见：com.juxtapose.example.ch06.db.CreditBillPreparedStatementSetter。

代码清单 6-57 `CreditBillPreparedStatementSetter` 类定义

```
1. public class CreditBillPreparedStatementSetter implements
2.     PreparedStatementSetter {
3.         public void setValues(PreparedStatement ps) throws SQLException {
4.             ps.setString(1, "5");
5.         }
6.     }
```

其中 3~5 行：完成参数设置，此处设置第一个参数为“5”。

使用自定义的 CreditBillPreparedStatementSetter 配置 JdbcCursorItemReader，参见代码清单 6-58。

代码清单 6-58 自定义参数配置 JdbcCursorItemReader

```
1. <bean:bean id="jdbcParameterItemReader" scope="step"
2.           class="org.springframework.batch.item.database.
3.           JdbcCursorItemReader" >
4.           <bean:property name="dataSource" ref="dataSource"/>
5.           <bean:property name="sql" value="select ID,ACCOUNTID,NAME,AMOUNT,
6.           DATE,ADDRESS from t_credit where id between 1 and ? "/>
7.           <bean:property name="rowMapper" ref="custCreditRowMapper" />
8.           <bean:property name="preparedStatementSetter"
9.                         ref=" custPreparedStatementSetter " />
10.          </bean:bean>
11.
12.          <bean:bean id="custPreparedStatementSetter"
13.                      class="com.juxtapose.example.ch06.db.
14.          CreditBillPreparedStatementSetter"/>
```

其中，7~8 行：使用自定义的 custPreparedStatementSetter 设置属性 preparedStatement Setter。

11~13 行：声明 custPreparedStatementSetter，实现类为 com.juxtapose.example.ch06.db.CreditBillPreparedStatementSetter。

6.5.2 HibernateCursorItemReader

对象关系映射（Object Relational Mapping，ORM）是一种为解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，使得 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。Spring Batch 框架对 ORM 类型的 Hibernate 提供了基于游标的读 ItemReader。

HibernateCursorItemReader 实现 ItemReader 接口，核心作用是将数据库中的记录通过 ORM 的方式转换为 Java Bean 对象。学会使用基于 JDBC 的读数据库后，使用 HibernateCursorItemReader 非常简单。下面我们先看 HibernateCursorItemReader 的关键支持的属性，参见表 6-15。

HibernateCursorItemReader 结构及关键属性

表 6-15 HibernateCursorItemReader 关键属性

HibernateCursorItemReader 属性	类 型	说 明
fetchSize	int	设置 ResultSet 每次向数据库取的行数；setFetchSize 的意思是当调用 rs.next 时，ResultSet 会一次性从服务器上取多少行数据回来，这样在下次 rs.next 时，可以直接从内存中获取数据而不需要网络交互，提高了效率。 默认值： -1
maxItemCount	int	设置结果集做多行数。 默认值： Integer.MAX_VALUE
parameterValues	String	Statement 的参数值
queryProvider	HibernateQueryProvider	生成 HQL 的查询类
queryString	String	HQL 查询语句
sessionFactory	SessionFactory	Hibernate 的 SessionFactory，负责与数据库进行交互
useStatelessSession	Boolean	是否使用无状态的会话。 默认值： true

配置 HibernateCursorItemReader

使用 HibernateCursorItemReader 至少需要配置 sessionFactory 和 queryString 两个属性；sessionFactory 是 Hibernate 的会话工厂类，用于与数据库进行交互访问；queryString 用于指定查询的 HQL 语句。

在给出详细配置文件之前，我们首先准备实体对象（参见代码清单 6-59）、配置 Hibernate 的.cfg.xml（参见代码清单 6-60）。本节示例仍然使用 6.5.1 章节使用的数据库脚本。

配置数据实体对象

完整内容参见类 com.juxtapose.example.ch06.hibernate.CreditBill。

代码清单 6-59 数据实体对象 CreditBill

```
1. @Entity
2. @Table(name="t_credit")
3. public class CreditBill {
4.     @Id
5.     @Column(name="ID")
6.     private String id;
7.
8.     @Column(name="ACCOUNTID")
```

```
9.     private String accountID = "";      /** 银行卡账户 ID */
10.
11.    @Column(name="NAME")
12.    private String name = "";           /** 持卡人姓名 */
13.
14.    @Column(name="AMOUNT")
15.    private double amount = 0;          /** 消费金额 */
16.
17.    @Column(name="DATE")
18.    private String date;              /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */
19.
20.    @Column(name="ADDRESS")
21.    private String address;           /** 消费场所 */
22.
23.    .....
24. }
```

其中，1 行：@Entity 注释声明该类为持久类，将一个 JavaBean 类声明为一个实体的数据表映射类，建议实现序列化。默认情况下，所有的类属性都为映射到数据表的持久性字段。若在类中，添加另外属性，而非映射自数据库的，要用 Transient 来注解。

2 行：@Table(name="t_credit") 持久性映射的表，表名为“t_credit”。@Table 是类一级的注解，定义在@Entity 之下，为实体 Bean 映射表、目录和 schema 的名字，默认为实体 Bean 的类名，不包含包名。

4 行：@Id，用于标识数据表的主键。

5 行：@Column(name="ID") 表示属性对应的数据库列的字段名。

配置 hibernate 的 cfg 文件

数据实体配置完成后，需要配置 hibernate 的配置文件，用于声明上面的数据实体。

完整文件参见：ch06/cfg/hibernate.cfg.xml。

代码清单 6-60 配置 hibernate 的 cfg 文件

```
1.  <hibernate-configuration>
2.    <session-factory>
3.      <mapping class="com.juxtapose.example.ch06.hibernate.CreditBill"/>
4.    </session-factory>
5.  </hibernate-configuration>
```

其中，3 行：声明 Hibernate 中使用的数据实体类 CreditBill。

配置 HibernateCursorItemReader

完整配置文件参见：ch06/job/job-db-hibernate.xml。

代码清单 6-61 展示了从数据库表 t_credit 中读取数据。

代码清单 6-61 配置 HibernateCursorItemReader

```
1.      <bean:bean id="hibernateItemReader" scope="step"
2.          class="org.springframework.batch.item.database.Hibernate
3.          CursorItemReader" >
4.          <bean:property name="sessionFactory" ref="sessionFactory"/>
5.          <bean:property name="queryString"
6.              value="from CreditBill where id between :begin and :end "/>
7.          <bean:property name="parameterValues">
8.              <bean:map>
9.                  <bean:entry key="begin" value="#{jobParameters['begin']}
```

其中，3 行：属性 sessionFactory，定义 Hibernate 的会话访问工厂类，用于 ORM 映射，访问数据库表信息。

4~5 行：属性 queryString 定义需要查询的 HQL 语句，“from CreditBill where id between :begin and :end”，该 HQL 语句有两个查询参数，分别是：begin 和 end；需要通过属性 parameterValues 指定参数值。

6~11 行：属性 parameterValues 用于指定属性 queryString 中的变量：begin 和 end；此处使用了参数后绑定技术，在 Job 执行时候可以通过指定 JobParameter 给 begin 和 end 参数赋值。

14~27 行：声明 hibernate 使用的会话工厂，使用 hibernate 提供的 LocalSessionFactoryBean，需要为下面的属性赋值 dataSource（数据源）、configurationClass（通过注解的方式获取）、configLocation（配置文件地址）和 hibernateProperties（基本属性信息）。

使用代码清单 6-62 执行定义的 hibernateReadJob。完整代码参见：test.com.juxtapose.example.ch06.JobLaunchHibernate。

代码清单 6-62 执行 hibernateReadJob

```
1. executeJob("ch06/job/job-db-hibernate.xml", "hibernateReadJob",
2.             new JobParametersBuilder().addDate("date", new Date())
3.             .addString("begin", "1").addString("end", "4"));
```

其中，3 行：传入参数 begin=1, end=4；最终 SQL 执行查询的脚本为“from CreditBill where id between 1 and 4”。

6.5.3 StoredProcedureItemReader

Spring Batch 框架对存储过程提供了支持，StoredProcedureItemReader 提供了对存储过程的支持，其运行和 JdbcCursorItemReader 类似，均是获取游标对象，然后转换为 JavaBean 对象。存储过程获取游标通常有如下几种方式：

- (1) 执行存储过程直接返回结果集 ResultSet，数据库 SQL Server、Sybase、DB2、Derby、MySQL 中的存储过程支持直接返回结果集 ResultSet；
- (2) 通过返回 Out 类型的参数，获取引用类型的游标，数据库 Oracle、PostgreSQL 提供此类型的支持能力；
- (3) 通过 function 调用返回结果集 ResultSet。

StoredProcedureItemReader 关键属性

StoredProcedureItemReader 中可以使用 JdbcCursorItemReader 中定义的所有属性，同时 StoredProcedureItemReader 支持下面特有的属性，参见表 6-16 StoredProcedureItemReader 支持的特有属性。

表 6-16 StoredProcedureItemReader 支持的特有属性

StoredProcedureItemReader 属性	类 型	说 明
function	Boolean	是否调用存储过程的 function。 默认值：false
parameters	SqlParameter	存储过程的参数类型
procedureName	String	调用的存储过程名称
refCursorPosition	int	使用 OUT 类型参数时候，指定 OUT 类型参数在参数列表中的位置，index 的列表从 0 开始。 默认值：0

配置 StoredProcedureItemReader

使用 StoredProcedureItemReader 与如何从存储过程获取游标有较大的关系，下面的示例

使用上面的第一种，即执行存储过程直接返回结果集 ResultSet，本示例使用 MySQL 数据库。

在给出详细配置文件之前，我们首先准备存储过程（代码清单 6-63）；本节示例仍然使用 6.5.1 章节使用的数据库表 t_credit。

创建存储过程完整内容参见：ch06/db/create-stored-procedure-mysql.sql。

代码清单 6-63 创建存储过程

```
1.  DROP PROCEDURE IF EXISTS query_credit;
2.  CREATE PROCEDURE query_credit() SELECT * FROM t_credit;
```

其中，1 行：删除已经存在的存储过程。

2 行：创建存储过程，本存储过程查询 t_credit 中所有的信息。

1. 存储过程直接返回 ResultSet

完整配置文件参见：ch06/job/job-db-stored-procedure.xml。

代码清单 6-64 展示了从存储过程 “query_credit” 中读取数据。

代码清单 6-64 配置 StoredProcedureItemReader

```
1.  <bean:bean id="storedProcedureItemReader" scope="step"
2.      class="org.springframework.batch.item.database.
   StoredProcedureItemReader" >
3.      <bean:property name="dataSource" ref="dataSource"/>
4.      <bean:property name="procedureName" value="query_credit"/>
5.      <bean:property name="rowMapper">
6.          <bean:bean class="org.springframework.jdbc.core.
   BeanPropertyRowMapper">
7.              <bean:property name="mappedClass"
8.                  value="com.juxtapose.example.ch06.CreditBill"/>
9.          </bean:bean>
10.     </bean:property>
11. </bean:bean>
```

其中，3 行：属性 dataSource 指定访问的数据源。

4 行：属性 procedureName 声明需要执行的存储过程的名称。

5~10 行：属性 rowMapper 定义如何将结果集 ResultSet 转换为 JavaBean 对象，本示例使用 BeanPropertyRowMapper 将 ResultSet 根据属性名自动映射到 CreditBill 对象。

读者可能注意到，该存储过程的调用和基于 JDBC 的调用类似，唯一不同的是在存储过程中使用属性 procedureName 指定需要调用的存储过程；而 JDBC 中使用属性 sql 属性指定需要调用的 SQL 语句。

2. 使用 function

如果使用 function，需要将属性“function”设置为 true，代码清单 6-65 的示例代码给出了

如何使用数据库的 function 功能。

代码清单 6-65 使用 function 配置 StoredProcedureItemReader

```
1. <bean:bean id="storedProcedureItemReader" scope="step"
2.   class="org.springframework.batch.item.database.
   StoredProcedureItemReader" >
3.   <bean:property name="dataSource" ref="dataSource"/>
4.   <bean:property name="procedureName" value="query_credit"/>
5.   <bean:property name="function" value="true"/>
6.   <bean:property name="rowMapper">
7.     .....
8.   </bean:property>
9. </bean:bean>
```

其中，5 行：属性 function 设置为 true，表示使用数据库的 function 获取数据。

3. 通过 Out 参数返回 ResultSet

使用 Out 参数返回结果集，需要额外使用 2 个属性：parameters 与 refCursorPosition。属性 refCursorPosition 用于指定 Out 参数在参数列表中的位置；属性 parameters 指定使用的参数。代码清单 6-66 给出了如何使用 Out 参数返回结果集。

代码清单 6-66 通过 Out 参数获取结果集

```
1. <bean:bean id="storedProcedureItemReader" scope="step"
2.   class="org.springframework.batch.item.database.
   StoredProcedureItemReader" >
3.   <bean:property name="dataSource" ref="dataSource"/>
4.   <bean:property name="procedureName" value="query_credit"/>
5.   <bean:property name="refCursorPosition" value="0"/>
6.   <bean:property name="parameters">
7.     <bean:list>
8.       <bean:bean class="org.springframework.jdbc.core.
   SqlOutParameter">
9.         <constructor-arg index="0" value="products"/>
10.        <constructor-arg index="1">
11.          <util:constant static-field="oracle.jdbc.
   OracleTypes.CURSOR"/>
12.        </constructor-arg>
13.      </bean:bean>
14.    </bean:list>
15.  </bean:property>
16.  <bean:property name="rowMapper">
17.    .....
18.  </bean:property>
19. </bean:bean>
```

其中，5行：属性 refCursorPosition 指定 Out 参数在参数列表中的位置。
6~15行：属性 parameters 定义存储过程的参数信息。

6.5.4 JdbcPagingItemReader

Spring Batch 框架提供了对 JDBC 分页读取支持的组件 JdbcPagingItemReader。JdbcPagingItemReader 实现 ItemReader 接口，核心作用是将数据库中的记录通过分页的方式转换为 Java 对象。JdbcPagingItemReader 通过引用 PagingQueryProvider、SimpleJdbcTemplate、RowMapper 关键接口实现上面功能；在 JdbcPagingItemReader 将数据库记录转换为 Java 对象时主要有两步工作：首先根据 SimpleJdbcTemplate 与 PagingQueryProvider 从数据库中根据分页的大小获取结果集 ResultSet；其次使用 RowMapper 将结果集 ResultSet 转换为 Java 对象，具体步骤见图 6-19。

JdbcPagingItemReader 结构及关键属性

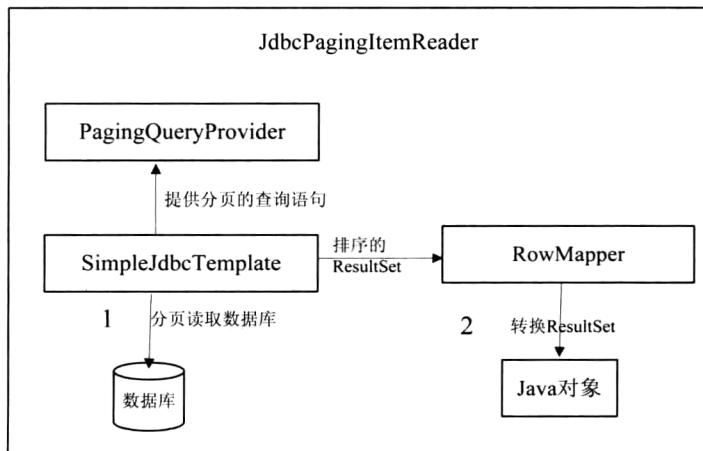


图 6-19 JdbcPagingItemReader 核心操作步骤

JdbcPagingItemReader 与关键接口 SimpleJdbcTemplate、PagingQueryProvider、RowMapper 之间的类图参见图 6-20。

JdbcPagingItemReader 引用 javax.sql.DataSource，DataSource 提供读取的数据库信息；JdbcCursorItemReader 通过 SimpleJdbcTemplate 对数据库进行读、写，使用 PagingQueryProvider 提供每次分页的查询语句，返回排序的结果集 ResultSet；如果需要为查询的 SQL 设置参数，可以通过属性 parameterValues 来设置参数值；RowMapper 负责将 ResultSet 对象转换为 Java 对象。JdbcPagingItemReader 关键接口、类说明参见表 6-17，JdbcPagingItemReader 关键属性参见表 6-18。

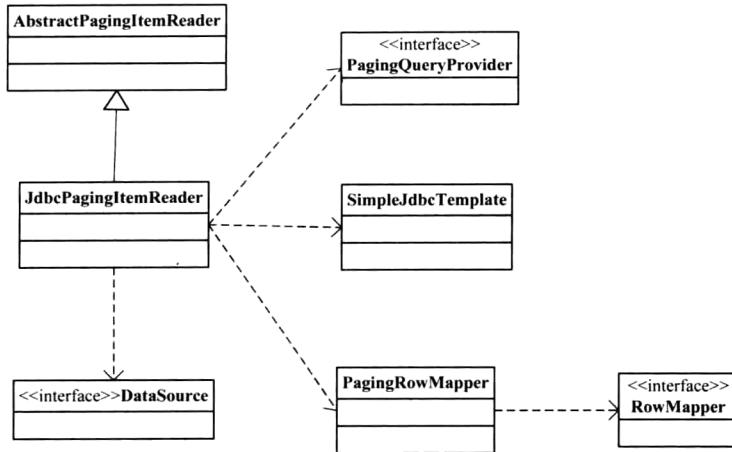


图 6-20 JdbcPagingItemReader 类关系图

表 6-17 JdbcPagingItemReader 关键接口、类说明

关键类	说 明
DataSource	提供读取数据库的数据源信息
SimpleJdbcTemplate	提供标准的 Spring 的 jdbc 模板，根据分页信息查询数据库，返回排序后的结果集 ResultSet
PagingQueryProvider	根据分页信息生成每次需要查询的 SQL 语句
RowMapper	负责将结果集 ResultSet 转换为 Java 对象

表 6-18 JdbcPagingItemReader 关键属性

JdbcPagingItemReader 属性	类 型	说 明
dataSource	DataSource	数据源，通过该属性指定使用的数据库信息
fetchSize	int	设置 ResultSet 每次向数据库取的行数；setFetchSize 的意思是当调用 rs.next 时，ResultSet 会一次性从服务器上取多少行数据回来，这样在下次 rs.next 时，可以直接从内存中取出数据而不需要网络交互，提高了效率。 默认值： -1
queryProvider	PagingQueryProvider	分页查询 SQL 语句生成器，负责根据分页信息生成每次需要执行的 SQL 语句
parameterValues	Map<String, Object>	设置定义的 SQL 语句中的参数
rowMapper	RowMapper	将结果集 ResultSet 转换为指定的 Pojo 对象类；需要实现 RowMapper 接口，默认，可以使用 BeanPropertyRowMapper；
pageSize	int	分页大小。 默认值： 10

Spring Batch 框架为了支持 PagingQueryProvider，根据不同的数据库类型提供多种实现，为了便于开发者屏蔽不同的数据库类型，Spring Batch 框架提供了友好的工厂类 SqlPagingQueryProviderFactoryBean 为不同的数据库类型提供 PagingQueryProvider 的实现类。下面列出工厂类需要的关键属性，参见表 6-19。

表 6-19 SqlPagingQueryProviderFactoryBean 关键属性

SqlPagingQueryProviderFactoryBean 属性	类 型	说 明
dataSource	DataSource	数据源，通过该属性指定使用的数据库信息
databaseType	String	指定数据库的类型，如果不显示指定该类型，则自动通过 dataSource 属性获取数据库的信息
ascending	Boolean	查询语句是否是升序。 默认值：true
fromClause	String	定义查询语句的 from 部分
selectClause	String	定义查询语句的 select 部分
sortKey	String	定义查询语句排序的关键字段
whereClause	String	定义查询语句的 where 字段

配置 JdbcPagingItemReader

使用 JdbcPagingItemReader 至少需要配置 dataSource、queryProvider、rowMapper 三个属性；dataSource 指定访问的数据源，queryProvider 用于定义分页查询的 SQL 语句，rowMapper 用于将结果集 ResultSet 转换为 Java 业务对象。

在给出详细配置文件之前，我们首先准备 SQL（使用的为 MySQL 数据库）脚本，示例中使用信用卡账单表，存放信用卡消费记录情况，主要字段包括 ID、ACCOUNTID、NAME、AMOUNT、DATE、ADDRESS；建表脚本参见代码清单 6-67。

完整内容参见文件 ch06/db/create-tables-mysql.sql。

代码清单 6-67 建表脚本

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

接下来我们使用分页度配置批处理任务，参见代码清单 6-68。

完整配置文件参见：ch06/job/job-db-paging-jdbc.xml。

代码清单 6-68 配置 JdbcPagingItemReader

```
1. <bean:bean id="jdbcItemPageReader" scope="step"
2.             class="org.springframework.batch.item.database.
JdbcPagingItemReader">
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="queryProvider" ref="refQueryProvider" />
5.     <bean:property name="parameterValues">
6.         <bean:map>
7.             <bean:entry key="begin" value="#{jobParameters['id_begin']}"/>
8.             <bean:entry key="end" value="#{jobParameters['id_end']}"/>
9.         </bean:map>
10.    </bean:property>
11.    <bean:property name="pageSize" value="2"/>
12.    <bean:property name="rowMapper" ref="custCreditRowMapper"/>
13. </bean:bean>
14. <bean:bean id="refQueryProvider" class="org.springframework.batch.item.
15. database.support.SqlPagingQueryProviderFactoryBean">
16.     <bean:property name="dataSource" ref="dataSource"/>
17.     <bean:property name="selectClause"
18.                   value="select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS"/>
19.     <bean:property name="fromClause" value="from t_credit"/>
20.     <bean:property name="whereClause" value="where ID between :begin
and :end"/>
21.     <bean:property name="sortKey" value="ID"/>
22. </bean:bean>
```

其中，3 行：属性 dataSource 定义需要访问的数据源。

4 行：属性 queryProvider 定义需要分页查询的语句，对象 “refQueryProvider” 的具体配置参见 14~22 行，本例中根据 ID 字段所在的范围区间查询表 t_credit 中的所有字段信息。

5~10 行：属性 parameterValues 定义需要执行 SQL 中的参数

```
<bean:entry key="begin" value="#{jobParameters['id_begin']}"/>
```

key 中的值 “begin” 对应 queryProvider 中的变量 “**:begin**”；value 中的 “#{jobParameters['id_begin']}

” 表示使用作业参数 “id_begin” 作为 begin 的值。

11 行：属性 pageSize 用于指定分页的大小，本例指定为 2；如果不指定使用默认值 10。

12 行：属性 rowMapper 将结果集 ResultSet 转换为 CreditBill 对象，使用自定义的 Mapper 实现 custCreditRowMapper。

配置数据源

配置数据源参见代码清单 6-69。

代码清单 6-69 配置数据源

```
1. <context:placeholder location="classpath:/ch06/properties/
batch-mysql.properties" />
```

```
2.  <bean:bean id="dataSource"
3.      class="org.springframework.jdbc.datasource.
   DriverManagerDataSource">
4.      <bean:property name="driverClassName">
5.          <bean:value>${datasource.driver}</bean:value>
6.      </bean:property>
7.      <bean:property name="url">
8.          <bean:value>${datasource.url}</bean:value>
9.      </bean:property>
10.     <bean:property name="username" value="${datasource.username}">
11.     </bean:property>
12.     <bean:property name="password" value="${datasource.password}">
13.     </bean:property>
14. </bean:bean>
```

其中，1行：指定加载的属性配置文件，方便动态指定数据源的属性。

2~12行：配置数据源的具体实现，分别指定属性 `driverClassName`、`url`、`username`、`password`。

自定义 RowMapper

使用 `RowMapper` 进行数据映射时，可以使用 Spring 框架提供的 `BeanPropertyRowMapper` 来自动将数据库的字段映射到指定的 Java Bean 上；另外只需要实现接口 `org.springframework.jdbc.core.RowMapper<T>`，根据需要完成自定义的数据转换功能，可以自定义 `RowMapper` 的实现类。`RowMapper` 接口定义参见代码清单 6-70。

代码清单 6-70 RowMapper 接口定义

```
1.  public interface RowMapper<T> {
2.      T mapRow(ResultSet rs, int rowNum) throws SQLException;
3. }
```

其中，2行：核心转换方法，参数为结果集和当前游标所在的位置。

本例中使用自定义的信用卡账单转换类 `com.juxtapose.example.ch06.db.CreditBillRowMapper`。`CreditBillRowMapper` 的代码实现参见代码清单 6-71。

代码清单 6-71 CreditBillRowMapper 类定义

```
1.  public class CreditBillRowMapper implements RowMapper<CreditBill> {
2.
3.      public CreditBill mapRow(ResultSet rs, int rowNum) throws SQLException {
4.          CreditBill bill = new CreditBill();
5.          bill.setAccountID(rs.getString("ACCOUNTID"));
6.          bill.setAddress(rs.getString("ADDRESS"));
7.          bill.setAmount(rs.getDouble("AMOUNT"));
8.          bill.setDate(rs.getString("DATE"));
9.          bill.setName(rs.getString("NAME"));}
```

```
10.         return bill;
11.     }
12. }
```

其中，3~11 行：业务实现代码，将给定的结果集 ResultSet 转化为 CreditBill 对象。

使用代码清单 6-72 执行定义的 dbPagingReadJob。

完整代码参见：test.com.juxtapose.example.ch06.JobLaunchJDBCPaging。

代码清单 6-72 执行 dbPagingReadJob

```
1. executeJob("ch06/job/job-db-paging-jdbc.xml", "dbPagingReadJob",
2.             new JobParametersBuilder().addDate("date", new Date())
3.             .addString("id_begin", "1").addString("id_end", "4"));
```

其中，2 行：传入参数 id_begin=1, id_end=4，最终 SQL 执行查询的脚本为“select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS from t_credit where id between 1 and 4”。

6.5.5 HibernatePagingItemReader

对象关系映射（Object Relational Mapping, ORM）是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，使得 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。Spring Batch 框架对 ORM 类型的 Hibernate 提供了基于分页的读 ItemReader。

HibernatePagingItemReader 实现 ItemReader 接口，核心作用是将数据库中的记录通过 ORM 的分页方式转换为 Java Bean 对象。学会使用基于 HibernateCursorItemReader 的读数据库后，使用 HibernatePagingItemReader 非常简单。

HibernatePagingItemReader 结构及关键属性

HibernatePagingItemReader 的关键支持的属性，参见表 6-20。

表 6-20 HibernatePagingItemReader 关键属性

HibernatePagingItemReader 属性	类 型	说 明
fetchSize	int	设置 ResultSet 每次向数据库取的行数； setFetchSize 的意思是当调用 rs.next 时， ResultSet 会一次性从服务器上取多少行数据 回来，这样在下次 rs.next 时，可以直接从内 存中取出数据而不需要网络交互，提高了效 率。 默认值： -1

续表

HibernatePagingItemReader 属性	类 型	说 明
maxItemCount	int	设置结果集做大行数。 默认值: Integer.MAX_VALUE
parameterValues	String	Statement 的参数值
queryProvider	HibernateQueryProvider	生成 HQL 的查询类
queryString	String	HQL 查询语句
sessionFactory	SessionFactory	Hibernate 的 SessionFactory, 负责与数据库进行交互
useStatelessSession	Boolean	是否使用无状态的会话。 默认值: true
pageSize	int	分页大小。 默认值: 10

配置 HibernatePagingItemReader

使用 HibernatePagingItemReader 至少需要配置 sessionFactory、queryString 两个属性; sessionFactory 是 Hibernate 的会话工厂类, 用于与数据库进行交互访问; queryString 用于指定查询的 HQL 语句。

在给出详细配置文件之前, 我们首先准备实体对象(参见代码清单 6-73)、配置 Hibernate 的.cfg.xml(参见代码清单 6-74)。本节示例仍然使用 6.5.1 章节使用的数据库脚本。

配置数据实体对象

完整内容参见类: com.juxtapose.example.ch06.hibernate.CreditBill。

代码清单 6-73 数据实体对象 CreditBill

```

1.  @Entity
2.  @Table(name="t_credit")
3.  public class CreditBill {
4.      @Id
5.      @Column(name="ID")
6.      private String id;
7.
8.      @Column(name="ACCOUNTID")
9.      private String accountID = "";    /** 银行卡账户 ID */
10.
11.     @Column(name="NAME")
12.     private String name = "";        /** 持卡人姓名 */
13.
14.     @Column(name="AMOUNT")
15.     private double amount = 0;       /** 消费金额 */

```

```
16.      @Column(name="DATE")
17.      private String date;           /** 消费日期，格式 YYYY-MM-DD HH:MM:SS*/
18.
19.
20.      @Column(name="ADDRESS")
21.      private String address;        /** 消费场所 */
22.
23.      .....
24. }
```

其中，1行：@Entity注释声明该类为持久类，将一个JavaBean类声明为一个实体的数据库表映射类，建议实现序列化。默认情况下，所有的类属性都为映射到数据表的持久性字段。若在类中，添加另外属性，而非映射来数据库的，要用Transient来注解。

2行：@Table(name="t_credit")持久性映射的表，表名为“t_credit”。@Table是类一级的注解，定义在@Entity之下，为实体Bean映射表，目录和schema的名字，默认为实体Bean的类名，不包含包名。

4行：@Id，用于标识数据表的主键。

5行：@Column(name="ID")表示属性对应的数据库列的字段名。

配置 hibernate 的 cfg 文件

数据实体配置完成后，需要配置hibernate的配置文件，用于声明上面的数据实体。

完整文件参见：ch06/cfg/hibernate.cfg.xml。

代码清单 6-74 配置 hibernate.cfg.xml

```
1. <hibernate-configuration>
2.   <session-factory>
3.     <mapping class="com.juxtapose.example.ch06.hibernate.CreditBill"/>
4.   </session-factory>
5. </hibernate-configuration>
```

其中，3行：声明Hibernate中使用的数据实体类CreditBill。

配置 HibernatePagingItemReader

完整配置文件参见：ch06/job/job-db-paging-hibernate.xml。

代码清单 6-75 展示了从数据库表 t_credit 中读取数据。

代码清单 6-75 配置 HibernatePagingItemReader

```
1.   <bean:bean id="hibernatePagingItemReader" scope="step"
2.     class="org.springframework.batch.item.database.
3.       HibernatePagingItemReader">
4.       <bean:property name="sessionFactory" ref="sessionFactory"/>
5.       <bean:property name="queryString" value="from CreditBill
           where id between :begin and :end"/>
```

```

6.      <bean:property name="parameterValues">
7.          <bean:map>
8.              <bean:entry key="begin" value="#{jobParameters['id_begin']}" />
9.              <bean:entry key="end" value="#{jobParameters['id_end']}" />
10.         </bean:map>
11.     </bean:property>
12.     <bean:property name="pageSize" value="2"/>
13. </bean:bean>
14.
15. <bean:bean id="sessionFactory"
16.     class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
17.     <bean:property name="dataSource" ref="dataSource"/>
18.     <bean:property name="configurationClass"
19.         value="org.hibernate.cfg.AnnotationConfiguration"/>
20.     <bean:property name="configLocation"
21.         value="classpath:/ch06/cfg/hibernate.cfg.xml"/>
22.     <bean:property name="hibernateProperties">
23.         <bean:value>
24.             hibernate.dialect=org.hibernate.dialect.MySQLDialect
25.             hibernate.show_sql=true
26.         </bean:value>
27.     </bean:property>
28. </bean:bean>

```

其中，3 行：属性 sessionFactory，定义 Hibernate 的会话访问工厂类，用于 ORM 映射，访问数据库表信息。

4~5 行：属性 queryString 定义需要查询的 HQL 语句，“from CreditBill where id between :begin and :end”，该 HQL 语句有两个查询参数，分别是:begin 和:end；需要通过属性 parameterValues 指定参数值。

6~11 行：属性 parameterValues 用于指定属性 queryString 中的变量:begin 和:end；此处使用了参数后绑定技术，在 Job 执行时候可以通过指定 JobParameter 给 begin 和 end 参数赋值。

12 行：属性 pageSize 定义分页的大小。

15~28 行：声明 hibernate 使用的会话工厂，使用 hibernate 提供的 LocalSessionFactoryBean，需要为下面的属性赋值 dataSource（数据源），configurationClass（通过注解的方式获取），configLocation（配置文件地址），hibernateProperties（基本属性信息）。

使用代码清单 6-76 中的代码执行定义的 hibernatePagingReadJob。

完整代码参见：test.com.juxtapose.example.ch06.JobLaunchHibernatePaging。

代码清单 6-76 执行 hibernatePagingReadJob

```

1. executeJob("ch06/job/job-db-paging-hibernate.xml",
"hibernatePagingReadJob",

```

```
2.             new JobParametersBuilder().addDate("date", new Date())
3.             .addString("id_begin", "1").addString("id_end", "4"));

```

其中，3行：传入参数 id_begin=1, id_end=4，最终 SQL 执行查询的脚本为“select ID, ACCOUNTID,NAME,AMOUNT,DATE, ADDRESS from t_credit where id between 1 and 4”。

6.5.6 JpaPagingItemReader

对象关系映射（Object Relational Mapping, ORM）是一种为解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

JPA（Java Persistence API）是 Sun 官方提出的 Java 持久化规范。JPA 通过 JDK 5.0 注解或 XML 描述对象—关系表的映射关系，并将运行期的实体对象持久化到数据库中；它为 Java 开发人员提供了一种对象/关系映射工具来管理 Java 应用中的关系数据。Spring Batch 框架对 ORM 类型的 JPA 提供了基于分页的读 ItemReader。

JpaPagingItemReader 实现 ItemReader 接口，核心作用将数据库中的记录通过 ORM 的分页方式转换为 Java Bean 对象。学会使用基于 HibernatePagingItemReader 的读数据库后，使用 JpaPagingItemReader 非常简单。下面我们先看 JpaPagingItemReader 的关键支持的属性，参见表 6-21。

JpaPagingItemReader 结构及关键属性

表 6-21 JpaPagingItemReader 关键属性

JpaPagingItemReader 属性	类 型	说 明
maxItemCount	int	设置结果集做大行数。 默认值： Integer.MAX_VALUE
parameterValues	String	执行 SQL 的参数值
queryProvider	JpaQueryProvider	生成 JPQL 的查询类
queryString	String	JPQL 查询语句
entityManagerFactory	EntityManagerFactory	用于创建实体管理器
pageSize	int	分页大小 默认值： 10

配置 JpaPagingItemReader

使用 JpaPagingItemReader 至少需要配置 entityManagerFactory、queryString 两个属性；entityManagerFactory 负责创建 EntityManager，后者负责完成对实体的增删改查等操作；queryString 用于指定查询的 JPQL 语句。

在给出详细配置文件之前，我们首先准备实体对象（参见代码清单 6-77）、配置 JPA 的

实体映射文件（参见代码清单 6-78）。本节示例仍然使用 6.5.1 章节使用的数据库脚本。

配置数据实体对象

完整内容参见类：com.juxtapose.example.ch06.jpa.CreditBill。

代码清单 6-77 数据实体对象 CreditBill

```
1.  @Entity
2.  @Table(name="t_credit")
3.  public class CreditBill {
4.      @Id
5.      @Column(name="ID")
6.      private String id;
7.
8.      @Column(name="ACCOUNTID")
9.      private String accountID = "";      /** 银行卡账户 ID */
10.
11.     @Column(name="NAME")
12.     private String name = "";      /** 持卡人姓名 */
13.
14.     @Column(name="AMOUNT")
15.     private double amount = 0;      /** 消费金额 */
16.
17.     @Column(name="DATE")
18.     private String date;          /** 消费日期，格式 YYYY-MM-DD HH:MM:SS */
19.
20.     @Column(name="ADDRESS")
21.     private String address;       /** 消费场所 */
22.
23.     .....
24. }
```

其中，1 行：@Entity 注释声明该类为持久类，将一个 JavaBean 类声明为一个实体的数据库表映射类，建议实现序列化。默认情况下，所有的类属性都为映射到数据表的持久性字段。若在类中，添加另外属性，而非映射自数据库的，要用 Transient 来注解。

2 行：@Table(name="t_credit") 持久性映射的表，表名为“t_credit”。@Table 是类一级的注解，定义在@Entity 之下，为实体 Bean 映射表、目录和 schema 的名字，默认为实体 Bean 的类名，不包含包名。

4 行：@Id，用于标识数据表的主键。

5 行：@Column(name="ID") 表示属性对应的数据库列的字段名。

配置 JPA 的持久化文件

数据实体配置完成后，需要配置 JPA 的实体持久化配置文件，用于声明上面的数据实体。

完整文件参见：ch06/jpa/persistence.xml。

代码清单 6-78 JPA 持久化配置 persistence.xml

```
1. <persistence>
2.   <persistence-unit name="creditBill" transaction-type="RESOURCE_LOCAL">
3.     <class>com.juxtapose.example.ch06.jpa.CreditBill</class>
4.     <exclude-unlisted-classes>true</exclude-unlisted-classes>
5.   </persistence-unit>
6. </persistence>
```

其中，3 行：声明 JPA 中使用的数据实体类 CreditBill。

4 行：声明排除所有未在此声明的实体类。

配置 JpaPagingItemReader

完整配置文件参见：ch06/job/job-db-paging-jpa.xml。

代码清单 6-79 展示了从数据库表 t_credit 中读取数据。

代码清单 6-79 配置 JpaPagingItemReader

```
1. <bean:bean id="jpaPagingItemReader" scope="step"
2.   class="org.springframework.batch.item.database.
JpaPagingItemReader">
3.   <bean:property name="entityManagerFactory"
ref="entityManagerFactory" />
4.   <bean:property name="queryString"
5.     value="select c from CreditBill c where id between :begin
and :end" />
6.   <bean:property name="parameterValues">
7.     <bean:map>
8.       <bean:entry key="begin" value="#{jobParameters['id_begin']}'" />
9.       <bean:entry key="end" value="#{jobParameters['id_end']}'" />
10.      </bean:map>
11.    </bean:property>
12.    <bean:property name="pageSize" value="2"/>
13. </bean:bean>
```

其中，3 行：属性 entityManagerFactory，定义 JPA 的实体管理器对象（参见代码清单 6-80），提供访问数据库表的操作。

4~5 行：属性 queryString 定义需要查询的 JPQL 语句，“select c from CreditBill c where id between :begin and :end”，该 JPQL 语句有两个查询参数，分别是：begin 和 end；需要通过属性 parameterValues 指定参数值。

6~11 行：属性 parameterValues 用于指定属性 queryString 中的变量：begin 和 end；此处使用了参数后绑定技术，在 Job 执行时候可以通过指定 JobParameter 给 begin 和 end 参数赋值。

12 行：属性 pageSize 定义分页的大小。

配置实体管理器对象

代码清单 6-80 配置实体管理器对象

```
1. <bean:bean id="entityManagerFactory"
2.           class="org.springframework.orm.jpa.
   LocalContainerEntityManagerFactoryBean">
3.   <bean:property name="dataSource" ref="dataSource" />
4.   <bean:property name="persistenceUnitName" value="creditBill" />
5.   <bean:property name="persistenceXmlLocation"
6.                 value="classpath:/ch06/jpa/persistence.xml" />
7.   <bean:property name="jpaVendorAdapter">
8.     bean:bean class="org.springframework.orm.jpa.vendor.
9.                 HibernateJpaVendorAdapter">
10.    <bean:property name="showSql" value="true" />
11.  </bean:bean>
12. </bean:property>
13. <bean:property name="jpaDialect">
14.   <bean:bean class="org.springframework.orm.jpa.vendor.
   HibernateJpaDialect" />
15. </bean:property>
16. </bean:bean>
```

其中，5行：属性 persistenceXmlLocation 指定需要加载的持久化配置文件。

7~11行：属性 jpaVendorAdapter 指定具体的 Provider，此处使用 Hibernate 的实现。

使用代码清单 6-81 执行定义的 jpaPagingReadJob。

完整代码参见：test.com.juxtapose.example.ch06.JobLaunchJpaPaging。

代码清单 6-81 执行 jpaPagingReadJob

```
1. executeJob("ch06/job/job-db-paging-jpa.xml", "jpaPagingReadJob",
2.             new JobParametersBuilder().addDate("date", new Date())
3.             .addString("id_begin", "1").addString("id_end", "4"));
```

其中，3行：传入参数 id_begin=1, id_end=4，最终 SQL 执行查询的脚本为“select ID, ACCOUNTID,NAME,AMOUNT,DATE, ADDRESS from t_credit where id between 1 and 4”。

6.5.7 IbatisPagingItemReader

对象关系映射（Object Relational Mapping, ORM）是一种为解决面向对象与关系数据库存在的互不匹配的现象的技术。简单地说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。

Ibatis 是一个开放源代码的对象关系映射框架，相对于 Hibernate，Ibatis 对 JDBC 进行了更轻量级的对象封装，由于 Ibatis 支持编写 SQL，使得 Ibatis 框架具有更高的灵活性和性能。Spring Batch 框架对 Ibatis 提供了基于分页的读 ItemReader。

IbatisPagingItemReader 实现 ItemReader 接口，其核心作用将数据库中的记录通过 ORM 的分页方式转换为 Java Bean 对象。学会使用基于 HibernatePagingItemReader 的读数据库后，使用 IbatisPagingItemReader 非常简单。下面我们先看 IbatisPagingItemReader 的关键支持的属性，参见表 6-22。

IbatisPagingItemReader 结构及关键属性

表 6-22 IbatisPagingItemReader 关键属性

IbatisPagingItemReader 属性	类 型	说 明
sqlMapClient	SqlMapClient	用于指定执行的命名 SQL 的配置文件和数据源
queryId	String	命名 SQL 定义的 ID
sqlMapClientTemplate	SqlMapClientTemplate	命名 SQL 执行的默认模板
parameterValues	Map<String, Object>	设置定义的 SQL 语句中的参数
pageSize	int	分页大小 默认值：10

配置 IbatisPagingItemReader

使用 IbatisPagingItemReader 至少需要配置 sqlMapClient、queryId 两个属性；sqlMapClient 用于指定配置的命名 SQL 的文件；queryId 用于指定命名 SQL 文件中定义的 SQL 语句。

在给出详细配置文件之前，我们首先准备配置命名 SQL 的配置文件和命名 SQL 文件。本节示例仍然使用 6.5.1 章节使用的数据库脚本。

定义命名 SQL 文件

命名 SQL 框架提供了标准的格式用于定义命名 SQL 文件。代码清单 6-82 的命名 SQL 对表 t_credit 进行操作：提供 ID 为“getAllCredits”的命名 SQL 用于查询所有的信用卡账单对象；提供 ID 为“getCreditsById”的命名 SQL 用于根据 ID 所在的区间查询信用卡账单对象。

完整文件参见：ch06/ibatis/ibatis-credit.xml。

代码清单 6-82 命名 SQL 定义

```
1.  <sqlMap namespace="Credit">
2.    <resultMap id="result" class="com.juxtapose.example.ch06.ibatis.CreditBill">
3.      <result property="id" column="ID" />
4.      <result property="accountID" column="ACCOUNTID" />
5.      <result property="name" column="NAME" />
6.      <result property="amount" column="AMOUNT" />
7.      <result property="date" column="DATE" />
8.      <result property="address" column="ADDRESS" />
9.    </resultMap>
10.
11.   <!-- 查询所有的信用卡账单对象 -->
```

```
12.   <select id="getAllCredits" resultMap="result">
13.       select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS from t_credit
14.   </select>
15.
16.   <!-- 根据 ID 所在区间查询信用卡账单对象 -->
17.   <select id="getCreditsById" parameterClass="java.util.HashMap"
      resultMap="result">
18.       select ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS
19.           from t_credit where ID between #begin# and #end#
20.   </select>
21. </sqlMap>
```

其中，1行：声明当前命名SQL文件的命名空间为“Credit”。

2~9行：定义命名SQL的返回值对象，将数据库的列于对象com.juxtapose.example.ch06.ibatis.CreditBill的属性进行映射，命名SQL执行后会自动将一行转换为指定的对象；以<result property="id" column="ID" />举例：命名SQL执行后，会将表t_credit中的字段ID映射到对象com.juxtapose.example.ch06.ibatis.CreditBill的ID属性上。

12~14行：提供ID为“getAllCredits”的命名SQL用于查询所有的信用卡账单对象。

16~20行：提供ID为“getCreditsById”的命名SQL用于根据ID所在的区间查询信用卡账单对象，此处定义了参数#begin#和#end#，通过属性parameterClass定义参数的类型为“java.util.HashMap”。

配置命名SQL的配置文件

命名SQL文件定义完成后，需要在Ibatis的配置文件中声明参见代码清单6-83；然后可以在IbatisPagingItemReader中使用命名SQL执行数据库的查询，参见代码清单6-84。

完整文件参见：ch06/ibatis/ibatis-config.xml。

代码清单6-83 声明命名SQL配置

```
1.  <sqlMapConfig>
2.      <sqlMap resource="ch06/ibatis/ibatis-credit.xml"/>
3.  </sqlMapConfig>
```

其中，2行：声明具体的Ibatis的命名SQL文件。

配置IbatisPagingItemReader

完整配置文件参见：ch06/job/job-db-paging-ibatis.xml。

代码清单6-84展示了通过命名SQL的方式从数据库表t_credit中读取数据。

代码清单6-84 配置IbatisPagingItemReader

```
1.  <bean:bean id="ibatisPagingItemReader" scope="step"
2.      class="org.springframework.batch.item.database.
      IbatisPagingItemReader">
```

```

3.      <!-- <bean:property name="queryId" value="getAllCredits" /> -->
4.      <bean:property name="queryId" value="getCreditsById" />
5.      <bean:property name="sqlMapClient" ref="sqlMapClient" />
6.      <bean:property name="parameterValues">
7.          <bean:map>
8.              <bean:entry key="begin"
9.                  value="#{jobParameters['id_begin']}" />
10.             <bean:entry key="end" value="#{jobParameters['id_end']}" />
11.         </bean:map>
12.     </bean:property>
13. </bean:bean>
14. <bean:bean id="sqlMapClient"
15.             class="org.springframework.orm.ibatis.
16.                 SqlMapClientFactoryBean">
17.     <bean:property name="dataSource" ref="dataSource" />
18.     <bean:property name="configLocation"
19.                 value="classpath:/ch06/ibatis/ibatis-config.xml" />
</bean:bean>
```

其中，4 行：属性 queryId，定义访问的命名 SQL 的 ID，此处访问 ID 为 getCreditById 的命名 SQL 语句。

5 行：属性 sqlMapClient 定义命名 SQL 的客户端配置文件。

6~11 行：属性 parameterValues 用于指定属性命名 SQL 中的变量#begin#和#end#；此处使用了参数后绑定技术，在 Job 执行时候可以通过指定 JobParameter 给 id_begin 和 id_end 参数赋值。

14~19 行：给出了 sqlMapClient 对象的具体定义，需要为其指定 2 个属性分别是 dataSource 和 configLocation； dataSource 指定使用的数据源， configLocation 指定命名 SQL 的配置文件加载地址。

使用代码清单 6-85 执行定义的 ibatisPagingReadJob。

完整代码参见：test.com.juxtapose.example.ch06.JobLaunchIbatisPaging。

代码清单 6-85 执行 ibatisPagingReadJob

```

1. executeJob("ch06/job/job-db-paging-ibatis.xml", "ibatisPagingReadJob",
2.             new JobParametersBuilder().addDate("date", new Date())
3.             .addString("id_begin", "1").addString("id_end", "4"));
```

其中，3 行：传入参数 id_begin=1, id_end=4，最终 SQL 执行查询的脚本为“select ID,ACCOUNTID,NAME,AMOUNT,DATE, ADDRESS from t_credit where id between 1 and 4”。