

6.6 读 JMS 队列

JMS (Java Messaging Service) 是 Java 平台上有关面向消息中间件 (MOM) 的技术规范，它便于消息系统中的 Java 应用程序进行消息交换，并且通过提供标准的产生、发送、接收消息的接口。JMS 是一种与厂商无关的 API，用来访问消息收发系统消息。它类似于 JDBC (Java Database Connectivity)，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。

Spring 的 JMS 抽象框架简化了 JMS API 的使用，并与 JMS 提供者(比如 IBM 的 WebSphere MQ、开源的 ActiveMQ 等) 平滑地集成。Spring JMS 框架提供了 JMS 访问的模板类 JmsTemplate，模板类处理资源的创建和释放，简化了 JMS 的使用。Spring Batch 框架基于 Spring JMS 框架提供了对 JMS 队列读取的 ItemReader。

6.6.1 JmsItemReader

JmsItemReader 实现 ItemReader 接口，核心作用将 JMS 队列中的消息转换为 Java 对象。JmsItemReader 引用 JmsOperations，后者负责对 JMS 队列消息的进行读取，并转化为指定的类型。

JmsItemReader 结构关键属性

图 6-21 展示了 JMS 队列读取的逻辑架构图，JmsOperations 负责将消息从队列中读取，并按照指定的消息类型格式转换为 Java 对象。

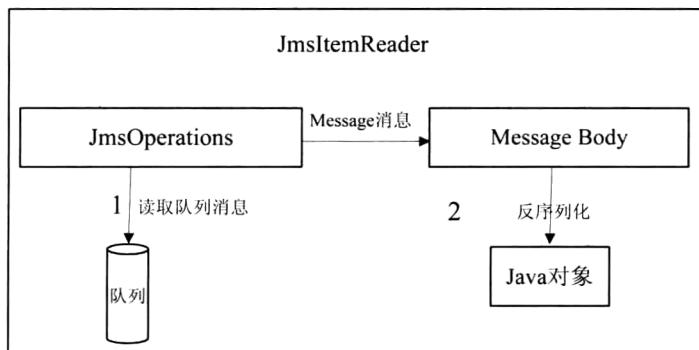


图 6-21 JMS 队列读取的逻辑架构

JmsItemReader 核心类架构图参见图 6-22。

JmsItemReader 将队列读取全部代理给 JmsOperations，JmsOperations 读取队列消息后根据指定的消息类型将消息的 Body 部分转换为属性 itemType 指定的对象。

JmsItemReader 关键属性参见表 6-23。

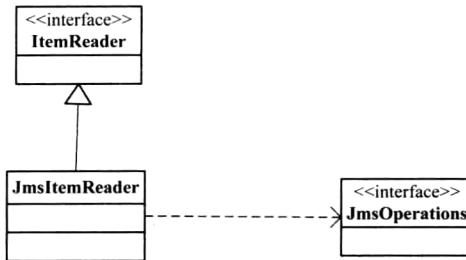


图 6-22 JmsItemReader 核心类

表 6-23 JmsItemReader 关键属性

JmsItemReader 属性	类 型	说 明
itemType	Class	Item 对象的类型
jmsTemplate	JmsOperations	消息发送模板

配置 JmsItemReader

现在假设所有的信用卡账单都是通过 JMS 消息的方式从外部系统发送过来的，需要使用 Spring Batch 框架从 JMS 队列中读取信用卡消费列表。配置 JmsItemReader 非常简单，只需要指定两个属性 itemType 和 jmsTemplate 即可，参见代码清单 6-86。本示例使用 Apache 的 ActiveMQ 作为消息中间件。

完整配置参见文件 ch06/job/job-jms.xml。

代码清单 6-86 配置 JmsItemReader

```

1.  <!-- 读取 jms -->
2.  <bean:bean id="jmsItemReader"
3.      class="org.springframework.batch.item.jms.JmsItemReader">
4.      <bean:property name="itemType" value="com.juxtapose.example.ch06.
   CreditBill"/>
5.      <bean:property name="jmsTemplate" ref="jmsTemplate"/>
6.  </bean:bean>

```

其中，4 行：属性 itemType 执行消息队列中存放的对象类型。

5 行：属性 jmsTemplate 指定读取 JMS 消息的模板，用于读取指定队列中的消息；消息模板配置参见代码清单 6-87。

JMS 消息模板的配置

代码清单 6-87 配置 JMS 消息模板

```

1.  <bean:bean id="jmsTemplate" class="org.springframework.jms.core.
   JmsTemplate">
2.      <bean:property name="connectionFactory" ref="jmsFactory"/>
3.      <bean:property name="defaultDestination" ref="creditDestination"/>

```

```

4.      <bean:property name="receiveTimeout" value="500"/>
5.    </bean:bean>
6.
7.    <amq:broker useJmx="false" persistent="false" schedulerSupport="false">
8.      <amq:transportConnectors>
9.        <amq:transportConnector uri="tcp://localhost:61616" />
10.     </amq:transportConnectors>
11.   </amq:broker>
12.
13.   <amq:connectionFactory id="jmsFactory" brokerURL="tcp://localhost:
14.     61616"/>
14.   <amq:queue id="creditDestination" physicalName="destination.
      creditBill" />
```

其中，2行：属性 connectionFactory 用于配置 JMS 的连接工厂。

3行：属性 defaultDestination 指定需要读取的目标消息队列。

4行：属性 receiveTimeout 指定读取消息的超时时间。

7~11行：定义 AMQ 的 broker，提供 JMS 的服务器端，指定对应的 ip 与 port；属性 persistent 表示不让消息持久化；属性 schedulerSupport 设置为 false，禁止掉 AMQ 的延迟发送的功能，可避免因为 AMQ 异常终止后导致无法启动。

13行：定义 JMS 的连接工厂。

14行：定义消息存储的队列，AMQ 启动时会自动创建名字为"destination.creditBill"的队列。

由于引用了新的命名空间 amq，需要在头文件中定义命名空间，参见代码清单 6-88。

代码清单 6-88 引入 amq 命名空间

```

1.  <bean:beans xmlns="
2.    xmlns:amq="http://activemq.apache.org/schema/core"
3.    xsi:schemaLocation="
4.      http://activemq.apache.org/schema/core
5.      http://activemq.apache.org/schema/core/activemq-core.xsd">
6.      .....
7.  </bean:beans>
```

作业 Job 配置完成后，为了能从指定的消息队列读取数据，需要将消息首先发送到队列中；使用 JMSTemplate 可以方便地读、写消息。代码清单 6-89 给出将消息发送到队列的代码片段。

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

代码清单 6-89 将消息发送到队列

```

1.  public static ApplicationContext getContext(String jobPath) {
2.      return new ClassPathXmlApplicationContext(jobPath);
3.  }
```

```

4.
5. public static JmsTemplate getJmsTemplate(ApplicationContext context){
6.     JmsTemplate jmsTemplate = (JmsTemplate)context.getBean
7.         ("jmsTemplate");
8.     return jmsTemplate;
9.
10. public static void sendMessage(JmsTemplate jmsTemplate, final CreditBill
11.     creditBill){
12.     jmsTemplate.send(new MessageCreator() {
13.         public Message createMessage(Session session)
14.             throws JMSException {
15.                 ObjectMessage message = session.createObjectMessage();
16.                 message.setObject(creditBill);
17.                 return message;
18.             }
19.     });
20.
21. //示例发送消息代码
22. //将消息发送到定义的队列"destination.creditBill"中
23. ApplicationContext context = getContex("ch06/job/job-jms.xml");
24. JmsTemplate jmsTemplate = getJmsTemplate(context);
25. sendMessage(jmsTemplate,
26.     new CreditBill("4047390012345678","tom",100.00,"2013-2-2 12:00:08",
27.         "Lu Jia Zui road"));
28. sendMessage(jmsTemplate,
29.     new CreditBill("4047390012345678","tom",320,"2013-2-3 10:35:21","Lu
30.         Jia Zui road"));
31. sendMessage(jmsTemplate,
32.     new CreditBill("4047390012345678","tom",360.00,"2013-2-11 11:12:38",
33.         "Longyang road"));

```

其中，1~3 行：定义获取 Spring Context 上下文方法。

5~8 行：定义从 Spring Context 获取 JMS 模板的方法。

10~19 行：使用给定的 JmsTemplate 发送给定的 CreditBill 消息到队列中；发送的消息类型为 Object 类型消息。

22~30 行：使用上面提供的方法发送 3 条消息到队列"destination.creditBill"中。

使用代码清单 6-90 执行定义的 jmsReadJob。

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

代码清单 6-90 执行 jmsReadJob

```

1. ApplicationContext context = getContex("ch06/job/job-jms.xml");
2. JmsTemplate jmsTemplate = getJmsTemplate(context);

```

```

3. sendMessage(jmsTemplate,
4.     new CreditBill("4047390012345678", "tom", 100.00, "2013-2-2 12:00:08",
5.         "Lu Jia Zui road"));
6. sendMessage(jmsTemplate,
7.     new CreditBill("4047390012345678", "tom", 320, "2013-2-3 10:35:21", "Lu
8.         Jia Zui road"));
9. executeJob(context, "jmsReadJob",
10.        new JobParametersBuilder().addDate("date", new Date())));

```

其中，9~10 行：执行“jmsReadJob”的作业，将消息从队列“destination.creditBill”中读出。

6.7 服务复用

复用现有的企业资产和服务是提高企业应用开发的快捷手段，Spring Batch 框架的读组件提供了复用现有服务的能力，利用 Spring Batch 框架提供的 ItemReaderAdapter 可以方便地复用业务服务、Spring Bean、EJB 或者其他远程服务。

ItemReaderAdapter 结构关键属性

ItemReaderAdapter 持有服务对象，并调用指定的操作来完成 ItemReader 中定义的 read 功能。需要注意的是：ItemReader 的 read 操作需要每次返回一条对象，当没有数据可以读取时需要返回 null；而现有的服务通常返回一个对象的数组或者 List 列表；因此现有的服务通常不能直接被 ItemReaderAdapter 使用，这需要我们在 ItemReaderAdapter 和现存的服务之间再增加一个 ServiceAdapter（服务适配器）来完成适配工作。

ItemReaderReader、服务适配（ServiceAdapter）和现有服务之间的关系参见图 6-23。

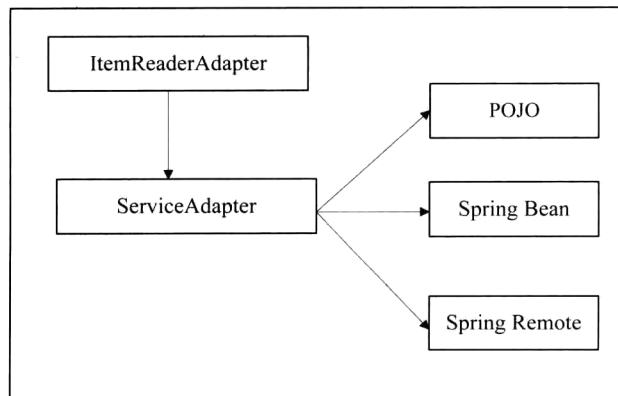


图 6-23 ItemReaderReader、服务适配（ServiceAdapter）和现有服务之间的关系

ItemReaderAdapter 通过 ServiceAdapter 来完成对现有服务的复用，ServiceAdapter 通常封装现存的服务例如 POJO、Spring Bean、Spring Remote 服务等满足 ItemReader 中定义的 read 操作要求。表 6-24 给出了 ItemReaderAdapter 的关键属性。

ItemReaderAdapter 关键属性

表 6-24 ItemReaderAdapter 关键属性

ItemReaderAdapter 属性	类 型	说 明
targetObject	Object	需要调用的目标服务对象
targetMethod	String	需要调用的目标操作名称
Arguments	Object[]	需要调用的操作参数

在配置 ItemReaderAdapter 时候只需要指定上面的三个属性即可，其中前两个参数 targetObject 和 targetMethod 必须填写，参数 arguments 根据操作是否有参数进行选择。

配置 ItemReaderAdapter

目前已经有服务（com.juxtapose.example.ch06.reuse.ExistService）能够查询所有的信用卡账单信息，接下来我们使用 ExistService 作为示例来演示如何使用 ItemReaderAdapter。

已经存在的服务 ExistService 的示例代码参见代码清单 6-91。

完整代码参见：com.juxtapose.example.ch06.reuse.ExistService。

代码清单 6-91 已存在服务 ExistService

```
1. public class ExistService {  
2.     public List<CreditBill> queryAllCreditBill(){  
3.         List<CreditBill> list = new ArrayList<CreditBill>();  
4.         //business service fill list  
5.         return list;  
6.     }  
7. }
```

操作 queryAllCreditBill 获取所有的账单信息；我们前面提到已有的服务其对应的返回值需要是 ItemReader 中 read 操作返回的类型；而 queryAllCreditBill 操作返回的是 List 类型，两者之间不匹配。因此我们需要在 ItemReaderAdapter 和现有的服务 ExistService 之间增加一个 Adapter 来完成转换。

新增类 CreditBillServiceAdapter，负责完成 ExistService 和 ItemReader 之间的适配功能。

CreditBillServiceAdapter 的示例代码参见代码清单 6-92。

完整代码参见：com.juxtapose.example.ch06.reuse.CreditBillServiceAdapter。

代码清单 6-92 CreditBillServiceAdapter 类定义

```
1. public class CreditBillServiceAdapter implements InitializingBean{  
2.     private ExistService existService;
```

```

3.     List<CreditBill> creditBillList;
4.
5.     @Override
6.     public void afterPropertiesSet() throws Exception {
7.         this.creditBillList = existService.queryAllCreditBill();
8.     }
9.
10.    public CreditBill nextCreditBill() {
11.        if (creditBillList.size() > 0) {
12.            return creditBillList.remove(0);
13.        } else {
14.            return null;
15.        }
16.    }
17.
18.    .....
19. }

```

CreditBillServiceAdapter 通过持有 ExistService 对象，并提供新的操作 nextCreditBill 来完成服务 ExistService 和 ItemReader 之间的适配。操作 nextCreditBill 每次返回一条信用卡账单信息，直到查询完毕返回 null 为止。

操作 afterPropertiesSet 在 Spring 组装对象之后，调用 existService 查询所有的账单信息。

接下来我们学习如何配置 ItemReaderAdapter，参见代码清单 6-93。

完整配置参见文件：ch06/job/job-reuse-service.xml。

代码清单 6-93 配置 ItemReaderAdapter

```

1.      <bean:bean id="reuseServiceRead"
2.          class="org.springframework.batch.item.adapter.
3.          ItemReaderAdapter">
4.          <bean:property name="targetObject" ref="existServiceAdapter"/>
5.          <bean:property name="targetMethod" value="nextCreditBill"/>
6.      </bean:bean>
7.
8.      <bean:bean id="existServiceAdapter"
9.          class="com.juxtapose.example.ch06.reuse.
10.         CreditBillServiceAdapter">
11.          <bean:property name="existService" ref="existService" />
12.      </bean:bean>
13.
14.      <bean:bean id="existService"
15.          class="com.juxtapose.example.ch06.reuse.ExistService" >
16.      </bean:bean>

```

其中，1~5 行：复用现有的服务完成 ItemReaderAdapter 的配置，属性 targetObject 使用

定义的服务适配器 `existServiceAdapter`; 属性 `targetMethod` 指定调用 `existServiceAdapter` 的操作 `nextCreditBill`。

7~10 行: 定义服务适配器 `existServiceAdapter`, 该对象完成了 `ItemReaderAdapter` 和 `existService` 之间的适配。

12~14 行: 声明现存的服务。

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

使用代码清单 6-94 执行定义的 `reuseServiceReadJob`。

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

代码清单 6-94 执行 `reuseServiceReadJob`

```
1. executeJob("ch06/job/job-reuse-service.xml", "reuseServiceReadJob",
2.               new JobParametersBuilder().addDate("date", new Date()));
```

6.8 自定义 ItemReader

Spring Batch 框架提供丰富的 `ItemReader` 组件, 当这些默认的系统组件不能满足需求时, 我们可以自己实现 `ItemReader` 接口, 完成需要的业务操作。自定义实现 `ItemRead` 非常容易, 只需要实现接口 `ItemReader`; 通常只实现接口 `ItemReader` 的读不支持重启, 为了支持可重启的自定义 `ItemReader` 需要实现接口 `ItemStream`。接下来我们展示如何自定义 `ItemReader`。

6.8.1 不可重启 `ItemReader`

接口 `ItemReader` 的定义参见代码清单 6-95。

代码清单 6-95 `ItemReader` 接口定义

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

在 `read` 操作中实现对指定资源的读取, 需要注意的是 `read` 每次返回一条记录, 直到没有数据记录返回 `null` 为止。代码清单 6-96 展示了自定义 `CustomCreditBillItemReader` 的实现。

完整代码参见: `com.juxtapose.example.ch06.cust.itemreader.CustomCreditBillItemReader`。

代码清单 6-96 `CustomCreditBillItemReader` 类定义

```
1. public class CustomCreditBillItemReader implements ItemReader
<CreditBill> {
2.     private List<CreditBill> list = new ArrayList<CreditBill>();
3. }
```

```

4.     public CustomCreditBillItemReader(List<CreditBill> list) {
5.         this.list = list;
6.     }
7.
8.     @Override
9.     public CreditBill read() throws Exception, UnexpectedInputException,
ParseException,
10.        NonTransientResourceException {
11.         if (!list.isEmpty()) {
12.             return list.remove(0);
13.         }
14.         return null;
15.     }
16. }

```

其中，9~15 行：每次执行 `read` 操作返回 `list` 中的一条记录，并将该记录从 `list` 中移除；如果 `list` 中没有记录可以返回，则返回 `null` 对象。

配置自定义的 `ItemReader` 非常简单，只需要简单的声明 `bean` 就可以。配置自定义 `ItemReader` 的声明参见代码清单 6-97。

完整 Job 配置参见文件：ch06/job/job-custom-itemreader.xml。

代码清单 6-97 配置自定义 ItemReader

```

1. <bean:bean id="customItemRead"
2.             class="com.juxtapose.example.ch06.cust.itemreader.
CustomCreditBillItemReader">
3. </bean:bean>

```

其中，1~3 行：声明自定义的 `ItemReader`。

接下来运行自定义 `ItemReader`，参见代码清单 6-98。

完整代码参见类：test.com.juxtapose.example.ch06.JobLaunchCustomItemReaderTest

代码清单 6-98 执行不可重启的自定义 ItemReader

```

1. List<CreditBill> list = new ArrayList<CreditBill>();
2. list.add(new CreditBill("1", "tom", 100.00, "2013-2-2 12:00:08", "Lu Jia Zui
road"));
3. list.add(new CreditBill("2", "tom", 320, "2013-2-3 10:35:21", "Lu Jia Zui
road"));
4. list.add(new CreditBill("3", "tom", 360.00, "2013-2-11 11:12:38", "Longyang
road"));
5. CustomCreditBillItemReader reader = new CustomCreditBillItemReader(list);
6. Assert.assertEquals("1", reader.read().getAccountID());
7. Assert.assertEquals("2", reader.read().getAccountID());
8. Assert.assertEquals("3", reader.read().getAccountID());
9. Assert.assertNull(reader.read());

```

其中，1~4 行：准备示例数据，完成 list 的初始化对象。

5 行：完成 CustomCreditBillItemReader 的初始化，指定从 list 中读取对象。

6~8 行：连续执行自定义 ItemReader 的 read 操作，获取 list 中的信用卡对账单对象；通过 JUnit 中的断言比较每次读取的对象是否正确。

9 行：第四次执行 read 操作应该返回 null，利用断言 assertNull 来验证读取的内容为 null。

本节实现了自定义的 ItemReader，但本节实现的自定义的 ItemReader 不支持重新启动的能力，导致 Job 作业失败的情况下不能从失败的执行点重新开始读取。6.8.2 章节我们将实现可重启的自定义 ItemReader。

6.8.2 可重启 ItemReader

Spring Batch 框架对 Job 提供了可重启的能力，所有 Spring Batch 框架中提供的 ItemReader 组件均支持可重启的能力。为了支持 ItemReader 的可重启能力，框架定义了接口 ItemStream，所有实现接口 ItemStream 的组件均支持可重启的能力。

ItemStream 接口定义参见代码清单 6-99。

代码清单 6-99 ItemStream 接口定义

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

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。

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。

接下来我们改造 6.8.1 章节的自定义的 ItemReader，新增实现接口 ItemStream，使其支持可重启的能力。

RestartableCustomCreditBillItemReader 的实现参见代码清单 6-100。

完整代码参见：com.juxtapose.example.ch06.cust.itemreader.RestartableCustomCreditBillItem Reader。

代码清单 6-100 RestartableCustomCreditBillItemReader 类定义

```
1.  public class RestartableCustomCreditBillItemReader implements
2.          ItemReader<CreditBill>, ItemStream{
3.      private List<CreditBill> list = new ArrayList<CreditBill>();
4.      private int currentLocation = 0;
5.      private static final String CURRENT_LOCATION = "current.location";
6.
7.      public RestartableCustomCreditBillItemReader(List<CreditBill> list) {
8.          this.list = list;
9.      }
10.
11.     public CreditBill read() throws Exception, UnexpectedInputException,
12.             ParseException, NonTransientResourceException {
13.         if (!list.isEmpty()) {
14.             return list.get(currentLocation++);
15.         }
16.         return null;
17.     }
18.
19.     public void open(ExecutionContext executionContext)
20.             throws ItemStreamException {
21.         if(executionContext.containsKey(CURRENT_LOCATION)){
22.             currentLocation = new Long(executionContext.getLong(CURRENT_
23.                                         LOCATION)).intValue();
24.         }
25.         else{
26.             currentLocation = 0;
27.         }
28.     }
29.
30.     public void update(ExecutionContext executionContext)
31.             throws ItemStreamException {
32.         executionContext.putLong(CURRENT_LOCATION, new Long(currentLocation)
33.                               .longValue());
34.     }
35.
36.     public void close() throws ItemStreamException {}
37. }
```

其中，11~17 行：read 操作，根据当前的位置 currentLocation 从 list 中读取数据，当前的位置标识在执行上下文中获取参见 open 操作中的代码实现。

19~29 行：open 操作，从执行上下文中获取当前读取的位置。

30~34 行：update 操作，将当前已经读过的数据位置存放在执行上下文中，通常 update

操作在 chunk 的事务提交后会执行一次。

36 行： close 操作，通常在此处关闭不再需要的资源。

配置自定义的可重启 ItemReader 非常简单，只需要简单的声明 bean 就可以。配置自定义 ItemReader 的声明参见代码清单 6-101。

完整 Job 配置参见文件：ch06/job/job-custom-itemreader.xml。

代码清单 6-101 配置自定义 ItemReader

```
1. <bean:bean id="restartableCustomItemRead"
2.     class="com.juxtapose.example.ch06.cust.itemreader.
3.             RestartableCustomCreditBillItemReader">
4. </bean:bean>
```

其中，1~4 行： 声明自定义的可重启的 ItemReader。

接下来运行 RestartableCustomCreditBillItemReader，参见代码清单 6-102。

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

代码清单 6-102 执行可重启的自定义 ItemReader

```
1. List<CreditBill> list = new ArrayList<CreditBill>();
2. list.add(new CreditBill("1", "tom", 100.00, "2013-2-2 12:00:08", "Lu Jia Zui
   road"));
3. list.add(new CreditBill("2", "tom", 320, "2013-2-3 10:35:21", "Lu Jia Zui
   road"));
4. list.add(new CreditBill("3", "tom", 360.00, "2013-2-11 11:12:38", "Longyang
   road"));
5.
6. RestartableCustomCreditBillItemReader reader =
7.         new RestartableCustomCreditBillItemReader(list);
8. ExecutionContext executionContext = new ExecutionContext();
9. ((ItemStream) reader).open(executionContext);
10. Assert.assertEquals("1", reader.read().getAccountID());
11. ((ItemStream) reader).update(executionContext);
12.
13. reader = new RestartableCustomCreditBillItemReader(list);
14. ((ItemStream) reader).open(executionContext);
15. Assert.assertEquals("2", reader.read().getAccountID());
```

其中，1~4 行：准备示例数据，完成 list 的初始化对象。

6~7 行：完成 RestartableCustomCreditBillItemReader 的初始化，指定从 list 中读取对象。

8 行：模拟新建执行上下文对象 executionContext。

9 行：执行自定义 ItemReader 的 open 操作，从执行上下文中获取当前已读记录的位置。

10 行：执行一次读操作。

11 行：执行自定义 ItemReader 的 update 操作，将当前记录的位置存放在执行上下文中。

13 行：重新构造一个新的 RestartableCustomCreditBillItemReader 对象，模拟重启该

Reader。

14 行：使用同一个执行上下文执行 open 操作。

15 行：再次执行读操作，这次会从上次执行的位置来读取数据，因此本次读取的数据是 ID 为 2 的记录。

本示例代码模拟了重新启动读操作的场景，其本质是运行时将游标的位置存放在执行上下文中，执行上下文中的数据在每次事务提交的时候会保存到数据库中；当作业 Job 重新启动的时候从执行上下文中重新获取上次读操作的位置，从正确的位置开始读操作；从而完成了支持重启的功能。

6.9 拦截器

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

6.9.1 拦截器接口

接口 ItemReadListener 定义参见代码清单 6-103。

代码清单 6-103 ItemReadListener 接口定义

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

为 ItemReader 配置拦截器参见代码清单 6-104。

完整配置参见文件：/ch06/job/job-listener.xml。

代码清单 6-104 配置 ItemReader 拦截器

```
1. <job id="itemReadJob">  
2.     <step id="itemReadStep">  
3.         <tasklet transaction-manager="transactionManager">  
4.             <chunk reader="creditItemRead" processor="creditBillProcessor"  
5.                 writer="creditItemWriter" commit-interval="2">  
6.                 <listeners>  
7.                     <listener ref="sysoutItemReadListener"></listener>  
8.                     <listener ref="sysoutAnnotationItemReadListener">  
9.                         </listener>  
10.                    </listeners>  
11.                </chunk>
```

```
11.      </tasklet>
12.      </step>
13.  </job>
14.  <bean:bean id="sysoutItemReadListener"
15.      class="com.juxtapose.example.ch06.listener.
16.      SystemOutItemReadlistener">
17.  </bean:bean>
18.  <bean:bean id="sysoutAnnotationItemReadListener"
19.      class="com.juxtapose.example.ch06.listener.SystemOutAnnotation">
20.  </bean:bean>
```

其中，6~9 行：为作业的读配置 2 个拦截器。

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

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

SystemOutItemReadlistener 的代码参见代码清单 6-105。

类 SystemOutItemReadlistener 完整代码参见：com.juxtapose.example.ch06.listener.SystemOutItemReadlistener。

代码清单 6-105 SystemOutItemReadlistener 类定义

```
1.  public class SystemOutItemReadlistener implements ItemReadListener
2.      <CreditBill> {
3.          public void beforeRead() {
4.              System.out.println("SystemOutItemReadlistener.beforeRead()");
5.          }
6.          public void afterRead(CreditBill item) {
7.              System.out.println("SystemOutItemReadlistener.afterRead()");
8.          }
9.          public void onReadError(Exception ex) {
10.              System.out.println("SystemOutItemReadlistener.onReadError()");
11.          }
11. }
```

6.9.2 拦截器异常

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

如果拦截器发生异常，会导致 Job 执行的状态为"FAILED"。

配置了错误拦截器的作业配置参见代码清单 6-106。

完整配置参见文件：/ch06/job/job-listener.xml。

代码清单 6-106 配置错误 ItemReader 拦截器

```
1. <job id="errorItemReadJob">
2.     <step id="errorItemReadStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="creditItemRead"
processor="creditBillProcessor"
5.                 writer="creditItemWriter" commit-interval="2">
6.                 <listeners>
7.                     <listener ref="errorItemReadListener"></listener>
8.                 </listeners>
9.             </chunk>
10.            </tasklet>
11.        </step>
12.    </job>
```

其中，7 行：errorItemReadListener 在 beforeRead 操作中抛出异常，会导致整个作业的失败。

6.9.3 执行顺序

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

1. sysoutItemReadListener 拦截器的 before 方法；
2. sysoutAnnotationItemReadListener 拦截器的 before 方法；
3. sysoutAnnotationItemReadListener 拦截器的 after 方法；
4. sysoutItemReadListener 拦截器的 after 方法。

6.9.4 Annotation

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

- @BeforeRead；
- @AfterRead；
- @OnReadError。

ItemReadListener 操作说明与 Annotation 定义参见表 6-25。

表 6-25 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

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

SystemOutAnnotation 的代码参见代码清单 6-107。

类 SystemOutAnnotation 完整代码参见：com.juxtapose.example.ch06.listener.SystemOutAnnotation。

代码清单 6-107 SystemOutAnnotation 类定义

```
1. public class SystemOutAnnotation {
2.     @BeforeRead
3.     public void beforeRead() {
4.         System.out.println("SystemOutAnnotation.beforeRead()");
5.     }
6.
7.     @AfterRead
8.     public void afterRead(CreditBill item) {
9.         System.out.println("SystemOutAnnotation.afterRead()");
10.    }
11.
12.    @OnReadError
13.    public void onReadError(Exception ex) {
14.        System.out.println("SystemOutAnnotation.onReadError()");
15.    }
16. }
```

6.9.5 属性 Merge

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

假设有这样一个场景，所有的 Step 都希望拦截器 sysoutItemReadListener 能够执行，而拦截器 sysoutAnnotationItemReadListener 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 6-108。

代码清单 6-108 merge 属性配置

```
1. <job id="mergeChunkJob">
2.     <step id="subChunkStep" parent="abstractParentStep">
3.         <tasklet>
4.             <chunk reader="creditItemRead" processor="creditBillProcessor"
5.                 writer="creditItemWriter" >
```

```
6.      <listeners merge="true">
7.          <listener ref="sysoutAnnotationItemReadListener">
8.              </listener>
9.          </listeners>
10.     </chunk>
11.   </tasklet>
12. </step>
13.
14. <step id="abstractParentStep" abstract="true">
15.     <tasklet>
16.         <chunk commit-interval="2" >
17.             <listeners>
18.                 <listener ref="sysoutItemReadListener"></listener>
19.             </listeners>
20.         </chunk>
21.     </tasklet>
22. </step>
```

其中，17~18 行：定义抽象作业步 abstractParentStep 中的拦截器。

6~8 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。

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

写数据 ItemWriter

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。ItemWriter 是实现写的重要组件，Spring Batch 框架提供了丰富的写基础设施来完成各种数据来源的写入功能，数据来源包括文本文件、Json 格式文件、XML 文件、DB、JMS 消息、写邮件等。

Spring Batch 框架默认提供了丰富的 Writer 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写入；对于已经存在的持久化服务，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较差。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。属性 commit-interval 的具体使用方式参照 5.3.1 章节。

7.1 ItemWrite

ItemWriter 是 Step 中对资源的写处理阶段，Spring Batch 框架已经提供了各种类型的写实现，包括对文本文件、XML 文件、数据库、JMS 消息、发送邮件等写的处理。写操作与 Chunk Tasklet 的关系参见图 7-1。

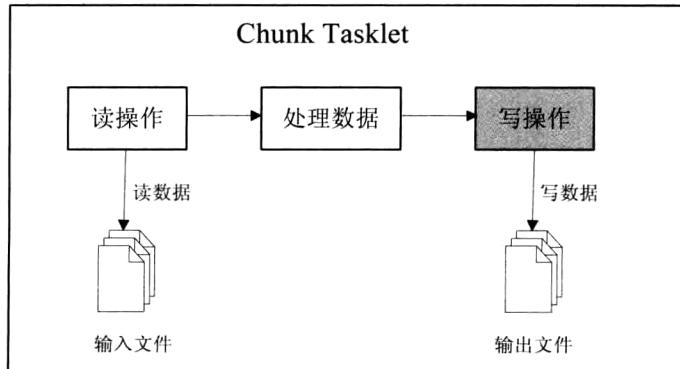


图 7-1 写操作与 Chunk Tasklet 的关系

7.1.1 ItemWriter

所有的写操作需要实现 org.springframework.batch.item.ItemWriter<T>接口。ItemWriter 接口定义参见代码清单 7-1。

代码清单 7-1 ItemWriter 接口定义

```
1. public interface ItemWriter<T> {  
2.     void write(List<? extends T> items) throws Exception;  
3. }
```

其中，2 行：ItemWriter 接口定义了核心作业方法 write() 操作，负责将 ItemReader 读入的数据写入指定的资源中；注意这里 write 操作的参数是 List<? extends T> items 类型的，表示写操作通常会进行批量的写入；每次写入 List 的大小由属性 commit-interval 决定。

Job 中典型的配置 ItemWriter 参见代码清单 7-2。

代码清单 7-2 配置 ItemWriter 示例

```
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>
```

7.1.2 ItemStream

框架同时提供了另外一个重要的接口 org.springframework.batch.item.ItemStream。ItemStream 接口定义了写操作与执行上下文 ExecutionContext 交互的能力。可以将已经写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，写操作可以跳过已经成功写过的数据，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写的位置）继续写。

ItemStream 接口定义参见代码清单 7-3。

代码清单 7-3 ItemStream 接口定义

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

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写入记录的位置。

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 框架提供的写组件 `FlatFileItemWriter`、`MultiResourceItemWriter`、`StaxEventItemWriter` 均实现了 `ItemStream` 接口。

7.1.3 系统写组件

Spring Batch 框架提供的写组件列表参见表 7-1。本章其他章节将重点介绍各种写组件的使用。

表 7-1 Spring Batch 框架提供的写组件

ItemWriter	说 明
<code>FlatFileItemWriter</code>	写 Flat 类型文件
<code>MultiResourceItemWriter</code>	多文件写组件
<code>StaxEventItemWriter</code>	写 XML 类型文件
<code>AmqpItemWriter</code>	写 AMQP 类型消息
<code>ClassifierCompositeItemWriter</code>	根据 Classifier 路由不同的 Item 到特定的 ItemWriter 处理
<code>HibernateItemWriter</code>	基于 Hibernate 方式写数据库
<code>IbatisBatchItemWriter</code>	基于 Ibatis 方式写数据库
<code>ItemWriterAdapter</code>	ItemWriter 适配器，可以复用现有的写服务
<code>JdbcBatchItemWriter</code>	基于 JDBC 方式写数据库
<code>JmsItemWriter</code>	写 JMS 队列
<code>JpaItemWriter</code>	基于 Jpa 方式写数据库
<code>GemfireItemWriter</code>	基于分布式数据库 Gemfire 的写组件
<code>SpELMappingGemfireItemWriter</code>	基于 Spring 表达式语言写分布式数据库 Gemfire 的组件
<code>MimeMessageItemWriter</code>	发送邮件的写组件
<code>MongoItemWriter</code>	基于分布式文件存储的数据库 MongoDB 写组件
<code>Neo4jItemWriter</code>	面向网络的数据库 Neo4j 的读组件
<code>PropertyExtractingDelegatingItemWriter</code>	属性抽取代理写组件：通过调用给定的 Spring Bean 方法执行写入，参数由 Item 中指定的属性字段获取作为参数
<code>RepositoryItemWriter</code>	基于 Spring Data 的写组件
<code>SimpleMailMessageItemWriter</code>	发送邮件的写组件
<code>CompositeItemWriter</code>	条目写的组合模式，支持组装多个 ItemWriter

7.2 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 ItemWriter 本质是将 Java 对象转换为 Flat 文件的记录。

7.2.1 FlatFileItemWriter

FlatFileItemWriter 实现 ItemWriter 接口，核心作用是将 ItemReader 或者 ItemProcessor 处理的 Java 对象转换为 Flat 文件中的一行记录，写入到指定的文件中。FlatFileItemWriter 通过引用 LineAggregator、FieldExtractor、FlatFileHeaderCallback、FlatFileFooterCallback 关键接口实现上面的目的；在 FlatFileItemWriter 将 Java 对象转换为文本记录时主要有两步工作，首先根据 FieldExtractor 将 Java 对象根据属性抽取出 Object 数组，其次使用 LineAggregator 将 Object 数组转换为 Flat 文本的一行记录，具体参步骤见图 7-2。

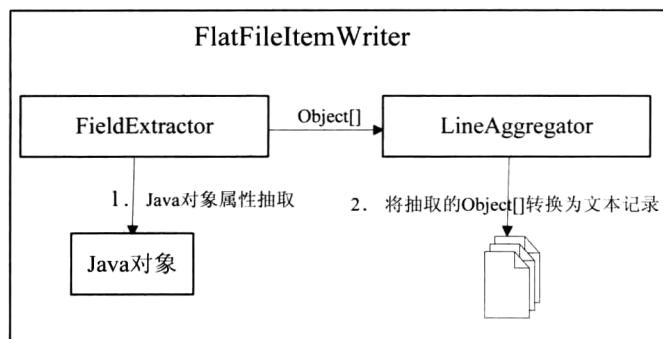


图 7-2 FlatFileItemWriter 核心操作步骤

FlatFileItemWriter 结构及关键属性

FlatFileItemWriter 与关键接口 LineAggregator、FieldExtractor、FlatFileHeaderCallback、FlatFileFooterCallback 之间的类图参见图 7-3。

FlatFileItemWriter 引用 org.springframework.core.io.Resource，Resource 负责提供写入的资源信息，可以是文件，也可以是其他类型的资源；FlatFileItemWriter 通过 LineAggregator 将 Java 对象转换为文本的一行记录，LineAggregator 引用 FieldExtractor 后，则负责将 Java 对象根据定义的属性将字段抽取为 Object 类型的数组，前者将 Object 类型数组转换为文本的一行记录；FlatFileHeaderCallback 是写文件头的回调类，在写记录之前会先调用该回调操作，通常在 open() 操作中调用该回调操作；FlatFileFooterCallback 是写文件尾的回调类，在写记录完成后会调用该回调操作，通常在 close() 操作中调用该回调操作。

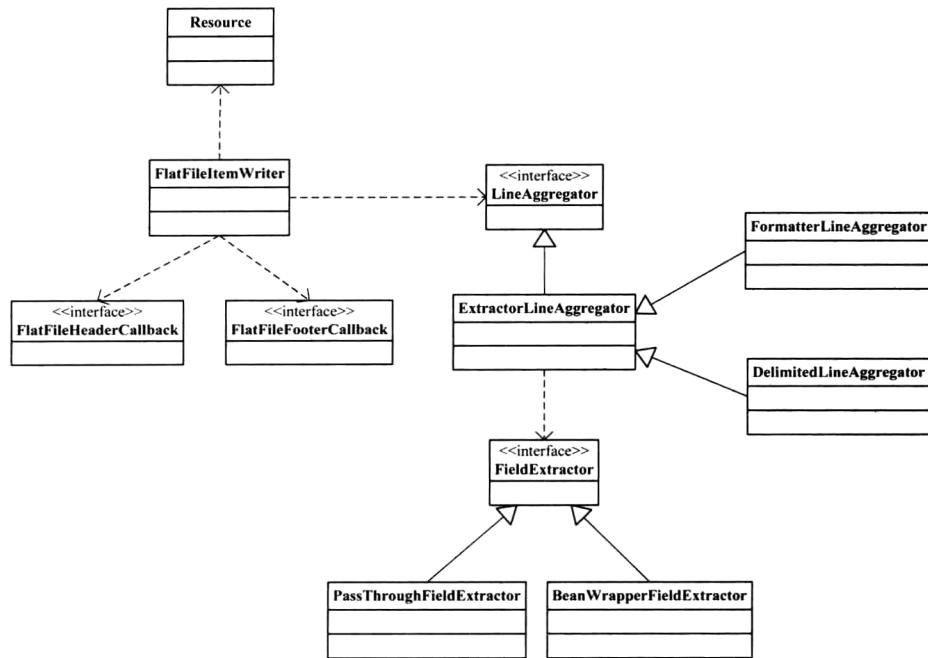


图 7-3 FlatFileItemWriter 类关系

关键接口、类说明参见表 7-2。

表 7-2 FlatFileItemWriter 关键接口、类说明

关键类	说 明
Resource	定义写入的文件资源
LineAggregator	将 Item 对象转换为一条文本记录，引用 FieldExtractor 来完成部分工作
FieldExtractor	字段抽取器，负责将 Item 对象根据定义的属性转换为 Object 数组；然后 LineAggregator 将 Object 数组转换为文本的一条记录
FlatFileHeaderCallback	文件头回调类，在 open() 操作中回调该接口，可以在此接口中完成文件头的写入
FlatFileFooterCallback	文件尾回调类，在 close() 操作中回调该接口，可以在此接口中完成文件尾的写入

FlatFileItemWriter 关键属性参见表 7-3。

表 7-3 FlatFileItemWriter 关键属性

FlatFileItemWriter 属性	类 型	说 明
appendAllowed	boolean	写入的文件如果存在，是否采用追加写的方式；如果此属性设置为 true，则属性 shoudDeleteIfExists 自动被设置为 false。 默认值：false

续表

FlatFileItemWriter 属性	类 型	说 明
forceSync	Boolean	是否强制同步写入。 默认值: false
Encoding	String	写入文件的编码类型, 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。 默认值: UTF-8
headerCallback	FlatFileHeaderCallback	文件头回调类, 在 open() 操作中回调该接口, 可以在此接口中完成文件头的写入
footerCallback	FlatFileFooterCallback	文件尾回调类, 在 close() 操作中回调该接口, 可以在此接口中完成文件尾的写入
lineAggregator	LineAggregator<T>	将条目对象转为一行文本
lineSeparator	String	行分隔符。 默认值: 从系统属性 line.separator 中获取
resource	Resource	需要写入的资源文件
saveState	boolean	是否保存写的状态。 默认值: true
shouldDeleteIfEmpty	boolean	没有记录写入文件情况下, 是否删除此文件。 默认值: false
shoudDeleteIfExists	boolean	文件已经存在情况下是否先删除此文件。 默认值: true
transactional	boolean	写是否在事务当中。 默认值: true

配置 FlatFileItemWriter

配置 Flat 格式的文件写相对读取较为简单, 只需要填写必要属性 resource 和 lineAggregator 即可; 其他属性可以单独设置, 如果不设置则使用上面属性列表中给出的默认值。本例中使用 FlatFileItemReader 读取文件 classpath:ch07/data/flat/credit-card-bill-201310.csv, 然后写入到文件 file:target/ch07/flat/outputFile.csv 中。FlatFileItemWriter 的配置参见代码清单 7-4。

配置 FlatFileItemWriter

完整配置参见文件: ch07/job/job-flatfile.xml。

代码清单 7-4 配置 FlatFileItemWriter

```

1.      <bean:bean id="flatFileItemWriter"
2.          class="org.springframework.batch.item.file.FlatFileItemWriter">
3.          <bean:property name="resource" value="file:target/ch07/flat/
outputFile.csv"/>

```

```

4.      <bean:property name="lineAggregator" ref="lineAggregator"/>
5.    </bean:bean>
6.
7.    <bean:bean id="lineAggregator"
8.      class="org.springframework.batch.item.file.transform.
9.      DelimitedLineAggregator">
10.      <bean:property name="delimiter" value=","></bean:property>
11.      <bean:property name="fieldExtractor">
12.        <bean:bean class="org.springframework.batch.item.
13.          file.transform.BeanWrapperFieldExtractor">
14.            <bean:property name="names"
15.              value="accountID,name,amount,date,address">
16.            </bean:property>
17.        </bean:bean>
18.      </bean:property>
19.    </bean:bean>

```

其中，3行：属性 resource 定义写入的文件，本例写入文件 target/ch07/flat/outputFile.csv 中。

4行：属性 lineAggregator 定义行聚合器对象，lineAggregator 负责将 Item 对象转换为文本的一行记录，本例使用分隔符的行聚合器 DelimitedLineAggregator 来实现该功能。

7~18行：定义分隔符行聚合器的实现，需要设置两个关键属性 delimiter 与 fieldExtractor；前者属性 delimiter 指定字段间的分隔符，后者属性 fieldExtractor 负责根据指定的属性将 Item 对象转换为 Object 数组。

9行：定义 Flat 格式文件每行中不同字段的分隔符号，本例使用","作为分隔符。

10行：定义属性字段抽取器，本例使用 BeanWrapperFieldExtractor 作为实现，根据 names 属性指定的值将 Item 对象转换为 Object 数组。

使用代码清单 7-5 执行定义的 flatFileJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchFlatFile。

代码清单 7-5 执行 flatFileJob

```

1. JobLaunchBase.executeJob("ch07/job/job-flatfile.xml", "flatFileJob",
2.                           new JobParametersBuilder().addDate("date", new Date()));

```

写入的文件（target/ch07/flat/outputFile.csv）内容，参见代码清单 7-6。

代码清单 7-6 写入文件 outputFile.csv 文件内容

```

1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road
3. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road
4. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road
5. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road
6. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road

```

7.2.2 LineAggregator

LineAggregator 负责将 Item 对象转换为一条文本记录，引用 FieldExtractor 来协助完成上面的任务。两者的分工如下：FieldExtractor 负责将 Item 对象转换为 Object 数组；LineAggregator 再将 Object 数组转换为字符串。

接口 org.springframework.batch.item.file.transform.LineAggregator<T> 定义参见代码清单 7-7。

代码清单 7-7 LineAggregator 接口定义

```
1. public interface LineAggregator<T> {  
2.     String aggregate(T item);  
3. }
```

aggregate() 操作负责将 Item 对象转换为一条文本记录。

LineAggregator 与 FieldExtractor 的关系参见图 7-4。

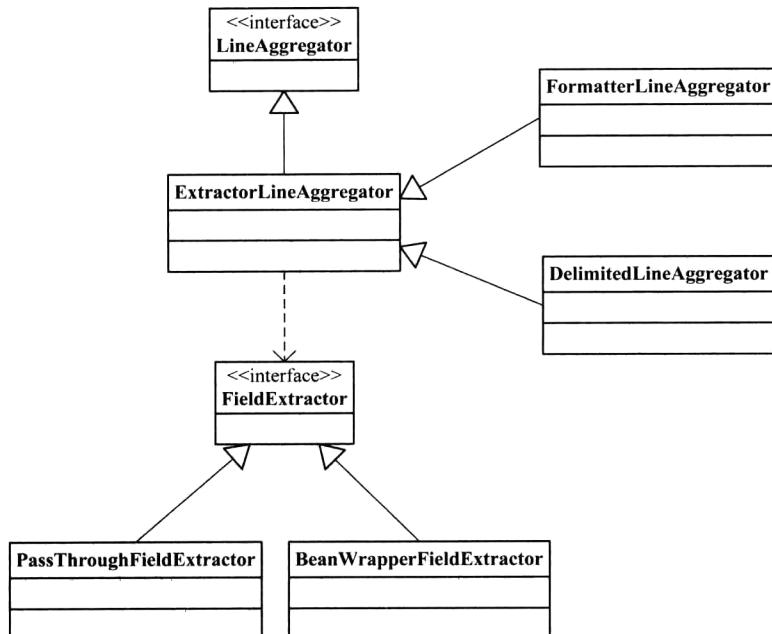


图 7-4 LineAggregator 与 FieldExtractor 类图

LineAggregator 系统实现

Spring Batch 框架中提供了 5 类默认的实现，参见表 7-4，每种实现完成特定的对象转换为一行文本记录的功能，如果默认的 5 类实现不能满足要求，读者可以自行实现 LineAggregator 接口进行转换。

表 7-4 LineAggregator 默认实现

LineAggregator	说 明
ExtractorLineAggregator	抽象类，提供抽象方法 String doAggregate(Object[] fields)，子类只需要实现将 Object 数组转换为 String 字符串即可。 org.springframework.batch.item.file.transform.ExtractorLineAggregator<T>
DelimitedLineAggregator	基于分隔符的行聚合器，将 Item 对象转换为分隔符的一行文本记录；默认的分隔符为","。 org.springframework.batch.item.file.transform.DelimitedLineAggregator<T>
FormatterLineAggregator	格式化行聚合器，将 Item 中指定的属性当做参数格式化一个字符串。 org.springframework.batch.item.file.transform.FormatterLineAggregator<T>
PassThroughLineAggregator<T>	直接使用 item 的 toString() 操作转换为 String 对象。 org.springframework.batch.item.file.transform.PassThroughLineAggregator<T>
RecursiveCollectionLineAggregator<T>	递归集合转换器，将 Collection 对象的每条记录转换为一行，采用默认的换行符号。 org.springframework.batch.item.file.transform.RecursiveCollectionLineAggregator<T>

自定义 LineAggregator

接下来我们介绍如何自定义实现 LineAggregator。假设账单文件需要每行记录有如下的要求，采用类似分隔符的方式存放属性，且每个属性需要有属性的名字和属性的值，两者之间用等号连接。举例如下，参见代码清单 7-8。

代码清单 7-8 写目标记录格式

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

然后我们创建自定义的类聚合器 CustomLineAggregator，将 Item 对象转换为上面格式的一行文本。为类 CustomLineAggregator 增加两个属性 delimiter 和 names，前者表示分割的符号，后者 names 表示每个字段对应的属性名称。为了实现简单，自定义的类 CustomLineAggregator 可以直接继承 ExtractorLineAggregator，只需要实现操作 doAggregate (Object[] fields) 即可（doAggregate 负责将 Object 数组转换为 String 对象）。

CustomLineAggregator 的实现参见代码清单 7-9。

类 CustomLineAggregator 完整代码参见：com.juxtapose.example.ch07.flat.CustomLineAggregator<T>。

代码清单 7-9 CustomLineAggregator 类定义

```
1. public class CustomLineAggregator<T> extends ExtractorLineAggregator<T> {
2.     private String delimiter = ";";
3.     private String[] names;
```

```

4.
5.     public void setDelimiter(String delimiter) {
6.         this.delimiter = delimiter;
7.     }
8.
9.     public void setNames(String[] names) {
10.         Assert.notNull(names, "Names must be non-null");
11.         this.names = Arrays.asList(names).toArray(new String[names.length]);
12.     }
13.
14.     @Override
15.     protected String doAggregate(Object[] fields) {
16.         List<String> fieldList = new ArrayList<String>();
17.         for (int i = 0; i < names.length; i++) {
18.             fieldList.add(names[i] + "=" + fields[i]);
19.         }
20.         return StringUtils
21.             .arrayToDelimitedString(
22.                 fieldList.toArray(new String[fieldList.size()]),
23.                 this.delimiter);
24.     }
25. }

```

其中，2 行：定义分隔符属性。

3 行：定义属性名称数组的属性，用于和 Object 中的元素值对应。

15~24 行：将 Object 数组转换为 String 格式的对象。

配置自定义的 CustomLineAggregator，具体配置参见代码清单 7-10。

完整配置参见文件：ch07/job/job-flatfile-custom-aggregator.xml。

代码清单 7-10 配置 CustomLineAggregator

```

1.      <bean:bean id="flatFileItemWriter"
2.          class="org.springframework.batch.item.file.FlatFileItemWriter">
3.          <bean:property name="resource"
4.              value="file:target/ch07/flat/custom-aggregator/
5.                  outputFile.csv"/>
6.          <bean:property name="lineAggregator" ref="customLineAggregator"/>
7.      </bean:bean>
8.
9.      <bean:bean id="customLineAggregator"
10.          class="com.juxtapose.example.ch07.flat.CustomLineAggregator">
11.          <bean:property name="names"
12.              value="accountID,name,amount,date,address"/>

```

```
12.      <bean:property name="fieldExtractor">
13.          <bean:bean class="org.springframework.batch.item.file.
14.                      transform.BeanWrapperFieldExtractor">
15.              <bean:property name="names"
16.                  value="accountID,name,amount,date,address">
17.              </bean:property>
18.          </bean:bean>
19.      </bean:property>
20.  </bean:bean>
```

其中，5行：使用自定义的行聚合器 customLineAggregator 定义 flatFileItemWriter。

8~20行：定义 customLineAggregator，实现类为 com.juxtapose.example.ch07.flat.CustomLineAggregator。属性 names 定义字段的名称，属性 fieldExtractor 定义使用的字段抽取器。

使用代码清单 7-11 执行定义的 flatFileJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchFlatFileCustomLineAggregator。

代码清单 7-11 执行 flatFileJob

```
1. JobLaunchBase.executeJob("ch07/job/job-flatfile-custom-aggregator",
2. "flatFileJob", new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容参见代码清单 7-12。

代码清单 7-12 写入后的目标文件 outputFile.csv

```
1. accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
   12:00:08;address=Lu Jia Zui road
2. accountID=4047390012345678;name=tom;amount=320.0;date=2013-2-3
   10:35:21;address=Lu Jia Zui road
3. accountID=4047390012345678;name=tom;amount=674.7;date=2013-2-6
   16:26:49;address=South Linyi road
4. accountID=4047390012345678;name=tom;amount=793.2;date=2013-2-9
   15:15:37;address=Longyang road
5. accountID=4047390012345678;name=tom;amount=360.0;date=2013-2-11
   11:12:38;address=Longyang road
6. accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28
   20:34:19;address=Hunan road
```

7.2.3 FieldExtractor

FieldExtractor 负责将 Item 对象转换为 Object 数组，LineAggregator 再将 Object 数组转换为字符串。

接口 org.springframework.batch.item.file.transform.FieldExtractor<T> 定义参见代码清单 7-13。

代码清单 7-13 FieldExtractor 接口定义

```
1. public interface FieldExtractor<T> {  
2.     Object[] extract(T item);  
3. }
```

extract()操作负责将 Item 对象转换为 Object 类型的数组。

FieldExtractor 与 LineAggregator 的关系参见图 7-5 中的类图。

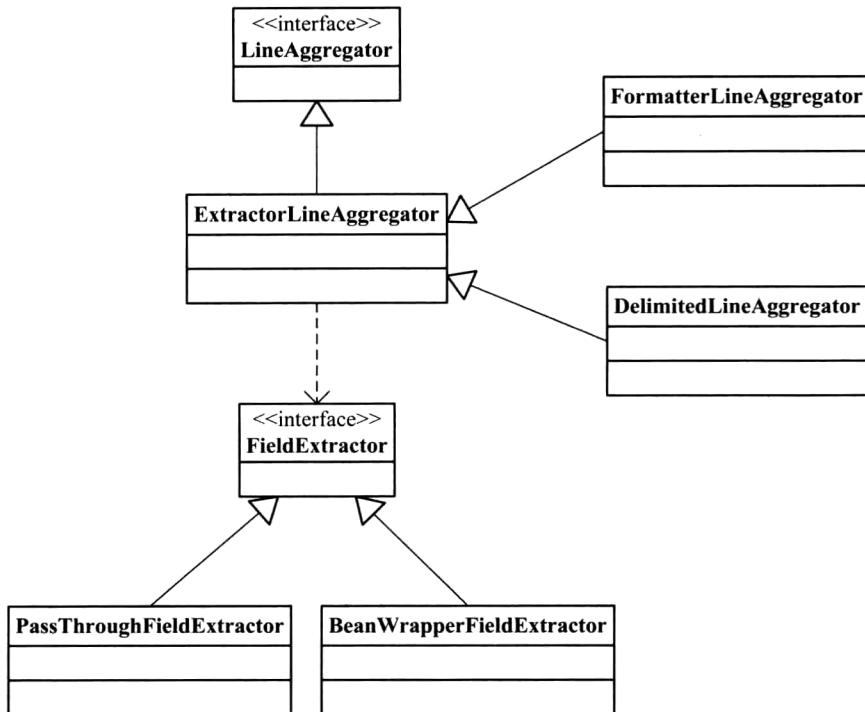


图 7-5 FieldExtractor 与 LineAggregator 的类图

FieldExtractor 系统实现

Spring Batch 框架中提供了 2 类默认的实现，参见表 7-5，每种实现完成特定的对象转换为 Object 类型数组的功能，如果默认的 2 类实现不能满足要求，读者可以自行实现 FieldExtractor 接口进行转换。

表 7-5 FieldExtractor 默认实现

FieldExtractor	说 明
BeanWrapperFieldExtractor	根据给定的属性名，分别调用指定属性的 getter 操作获取属性的值，并组装为 Object 数组直接返回； <code>org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor<T></code>

续表

FieldExtractor	说 明
PassThroughFieldExtractor	如果 Item 对象是数组类型直接返回; 如果 Item 对象是 Collection 类型, 直接使用 toArray 操作返回; 如果 Item 对象是 Map 类型, 使用 values 返回; 如果 Item 对象是 FieldSet 类型, 使用 getValues 返回; 如果 Item 对象是非上面的类型, 则返回 new Object[] { item }; org.springframework.batch.item.file.transform.PassThroughFieldExtractor<T>

7.2.4 回调操作

在 Flat 类型的写入操作中, 通常在写入文件之前和写入文件之后需要在文本文件中写入额外的内容, 例如注释、写入开始时间、写入完成时间等信息。Spring Batch 框架对 FlatFileItemWriter 增加了写入之前和写入之后的回调操作 FlatFileHeaderCallback 与 FlatFileFooterCallback。

FlatFileHeaderCallback 是写文件头的回调类, 在写记录之前会先调用该回调操作, 通常在 open() 操作中调用该回调操作。

FlatFileFooterCallback 是写文件尾的回调类, 在写记录完成后会调用该回调操作, 通常在 close() 操作中调用该回调操作。

接口声明

接口 FlatFileHeaderCallback 的定义, 参见代码清单 7-14。

接口完整的代码路径: 由 org.springframework.batch.item.file.FlatFileHeaderCallback 定义。

代码清单 7-14 FlatFileHeaderCallback 接口定义

```
1. public interface FlatFileHeaderCallback {
2.     void writeHeader(Writer writer) throws IOException;
3. }
```

writeHeader() 操作在执行真正的文件写入之前调用。

接口 FlatFileFooterCallback 的定义参见代码清单 7-15。

接口完整的代码路径: 由 org.springframework.batch.item.file.FlatFileFooterCallback 定义。

代码清单 7-15 FlatFileFooterCallback 接口定义

```
1. public interface FlatFileFooterCallback {
2.     void writeFooter(Writer writer) throws IOException;
3. }
```

writeFooter() 操作在执行真正的文件写入完成后调用。

FlatFileItemWriter、FlatFileHeaderCallback、FlatFileFooterCallback 三者的序列图参见图 7-6。

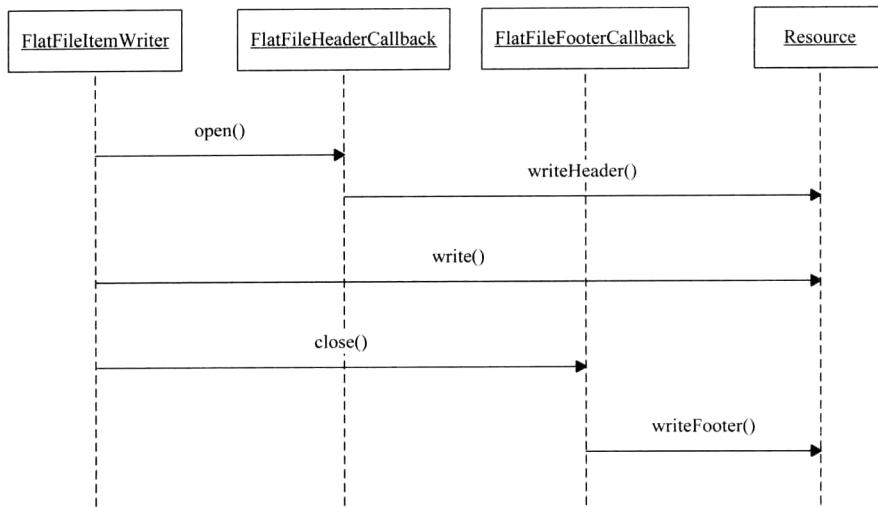


图 7-6 FlatFileItemWriter、FlatFileHeaderCallback、FlatFileFooterCallback 序列图

接口实现

接下来我们实现回调接口，在写入的文件头和文件末尾处写入的开始、结束日期及注释信息。

DefaultFlatFileHeaderCallback 的实现参见代码清单 7-16。

类 DefaultFlatFileHeaderCallback 的完整代码参见：com.juxtapose.example.ch07.flat.DefaultFlatFileHeaderCallback。

代码清单 7-16 DefaultFlatFileHeaderCallback 类定义

```

1. public class DefaultFlatFileHeaderCallback implements
2.   FlatFileHeaderCallback {
3.     public void writeHeader(Writer writer) throws IOException {
4.       writer.write("##credit 201310 begin.");
5.     }
  
```

其中，3 行：在文件的头部写入注释"##credit 201310 begin."。

DefaultFlatFileFooterCallback 的实现参见代码清单 7-17。

类 DefaultFlatFileFooterCallback 完整代码参见：com.juxtapose.example.ch07.flat.DefaultFlatFileFooterCallback。

代码清单 7-17 DefaultFlatFileFooterCallback 类定义

```

1. public class DefaultFlatFileFooterCallback implements
2.   FlatFileFooterCallback {
3.     public void writeFooter(Writer writer) throws IOException {
  
```

```
4.      }
5.  }
```

其中，3行：在文件的尾部写入注释“##credit 201310 end.”。

配置回调

配置 FlatFileItemWriter 的回调操作，具体配置参见代码清单 7-18。

完整配置参见文件：ch07/job/job-flatfile-callback.xml。

代码清单 7-18 配置 FlatFileItemWriter 回调操作

```
1.  <bean:bean id="flatFileComplexItemWriter"
2.    class="org.springframework.batch.item.file.FlatFileItemWriter">
3.      <bean:property name="resource"
4.        value="file:target/ch07/flat/callback/outputFile.csv"/>
5.      <bean:property name="lineAggregator" ref="lineAggregator"/>
6.      <bean:property name="headerCallback" ref="headerCallback"/>
7.      <bean:property name="footerCallback" ref="footerCallback"/>
8.  </bean:bean>
9.  <bean:bean id="headerCallback"
10.   class="com.juxtapose.example.ch07.flat.
11.     DefaultFlatFileHeaderCallback" />
12.  <bean:bean id="footerCallback"
13.   class="com.juxtapose.example.ch07.flat.
14.     DefaultFlatFileFooterCallback" />
```

其中，6行：属性 headerCallback 定义文件写入之间的回调操作。

7行：属性 footerCallback 定义文件写入完成之后的回调操作。

使用代码清单 7-19 执行定义的 flatFileComplexJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchFlatFileCallback。

代码清单 7-19 执行 flatFileComplexJob

```
1. JobLaunchBase.executeJob("ch07/job/job-flatfile-callback.xml",
2.   "flatFileComplexJob",
3.   new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容参见代码清单 7-20。

代码清单 7-20 写入后的 outputFile.csv 文件内容

```
1. ##credit 201310 begin.
2. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road
3. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road
4. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road
5. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road
6. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road
7. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road
8. ##credit 201310 end.
```

其中，1行：为 DefaultFlatFileHeaderCallback 写入的头信息。

8行：为 DefaultFlatFileFooterCallback 写入的尾信息。

7.3 XML 格式文件

Spring Batch 框架提供了方便的 XML 文件的写组件 StaxEventItemWriter。StaxEventItemWriter 采用 StAX（The Streaming API for XML）方式对 XML 文件进行写入。

7.3.1 StaxEventItemWriter

StaxEventItemWriter 实现 ItemWriter 接口，其核心作用是将 Item 对象转换为 XML 文件中的记录。StaxEventItemWriter 通过引用 OXM 组件完成对 XML 的写操作，负责将 Item 对象转换为 XML。

StaxEventItemWriter 结构关键属性

图 7-7 展示了 XML 文件写入的逻辑架构图，Marshaller 组件负责将 Item 对象序列化为 XML 格式数据，然后交给 XMLEventWriter 组件采用 StAX 的模式写入到指定的 XML 文件中。

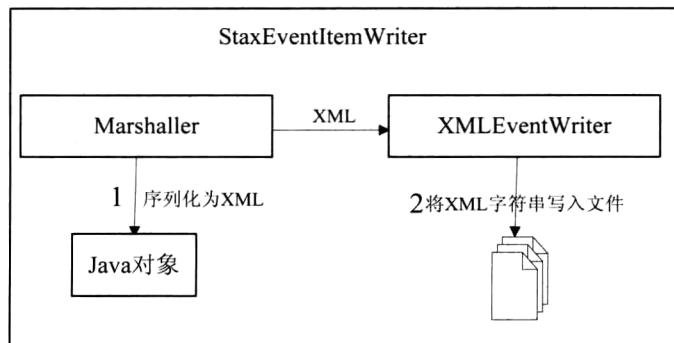


图 7-7 XML 文件写入的逻辑架构图

StaxEventItemWriter 核心类架构图参见图 7-8。

StaxEventItemWriter 中的核心类或者接口包括：Resource、Marshaller、XMLEvenWriter、StaxWriterCallback。Resource 表示需要写入的文件资源；Marshaller 负责将 Java 对象转换为 XML 片段，即进行序列化操作；XMLEventWriter 接口负责提供 StAX 方式对 XML 的写入；StaxWriterCallback 提供写入文件之前、之后的回调操作。

在实际配置 StaxEventItemWriter 时，只需要配置 Marshaller、Resource 两个属性即可。

关键接口、类说明参见表 7-6。

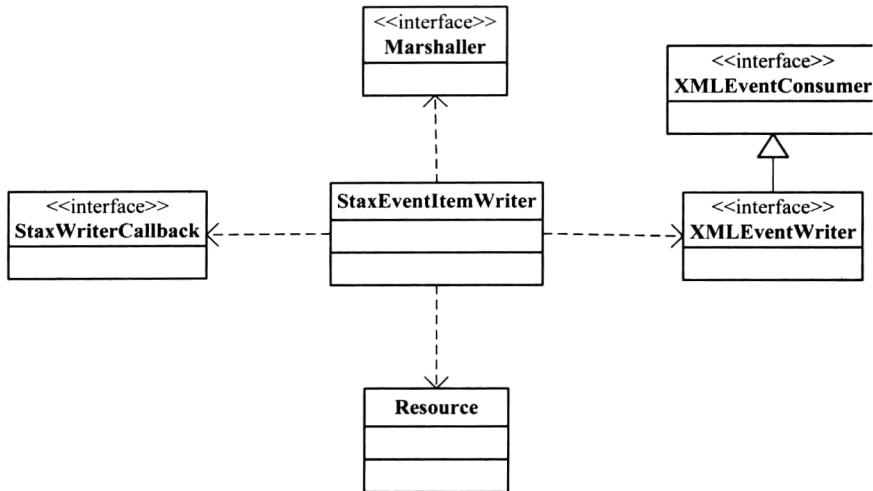


图 7-8 StaxEventItemWriter 核心类架构图

表 7-6 StaxEventItemWriter 关键接口、类说明

关键类	说 明
Resource	定义写入的文件资源
Marshaller	序列化类，负责将 Java 对象转换为 XML 片段
XMLEventWriter	负责提供 StAX 方式对 XML 的写入，基于事件驱动的 XML 处理方式
StaxWriterCallback	写入文件之前、之后的回调类

StaxEventItemWriter 关键属性参见表 7-7。

表 7-7 StaxEventItemWriter 关键属性

StaxEventItemWriter 属性	类 型	说 明
encoding	String	写入 XML 时候的文件编码。 默认值：UTF-8
footerCallback	StaxWriterCallback	写文件尾的回调类，在 close() 操作中回调该接口，可以在接口中完成文件尾的写入
headerCallback	StaxWriterCallback	写文件头的回调类，在 open() 操作中回调该接口，可以在接口中完成文件头的写入
marshaller	Marshaller	Spring OXM 实现类，负责将 Java 对象转换为 XML 内容
overwriteOutput	Boolean	如果文件存在是否覆盖原有的文件。 默认值：true
resource	Resource	需要写入的资源文件

续表

StaxEventItemWriter 属性	类 型	说 明
rootElementAttributes	Map<String, String>	根元素的属性，如果 key 的名字以“xmlns”为前缀表示命名空间。
rootTagName	String	根元素的名字。 默认值：root
version	String	指定 XML 的版本号。 默认值：1.0
saveState	Boolean	是否保存写的状态。 默认值：true
transactional	Boolean	写是否在事务当中。 默认值：true
forceSync	Boolean	是否强制同步写入。 默认值：false
shouldDeleteIfEmpty	Boolean	文件已经存在情况下是否先删除此文件。 默认值：true

配置 StaxEventItemWriter

在给出详细配置文件之前，我们首先给出读入的 XML 文件示例（参见代码清单 7-21），通过 XML 的方式描述了信用卡的消费清单。

文件 ch07/data/xml/credit-card-bill-201310.xml。

代码清单 7-21 读入的 XML 文件示例

```

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.    .....
10.   </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 StaxEventItemWriter 代码，参见代码清单 7-22。

完整配置文件参见：ch07/job/job-xml.xml。

代码清单 7-22 配置 StaxEventItemWriter

```
1. <bean:bean id="xmlWriter"
2.     class="org.springframework.batch.item.xml.StaxEventItemWriter">
3.     <bean:property name="rootTagName" value="juxtapose"/>
4.     <bean:property name="marshaller" ref="creditMarshaller"/>
5.     <bean:property name="resource" value="file:target/ch07/xml/
       credit-card-bill.xml"/>
6. </bean:bean>
7. <bean:bean id="creditMarshaller"
8.     class="org.springframework.oxm.xstream.XStreamMarshaller">
9.     <bean:property name="aliases">
10.        <util:map id="aliases">
11.            <bean:entry key="credit"
12.                value="com.juxtapose.example.ch07.CreditBill"/>
13.        </util:map>
14.    </bean:property>
15. </bean:bean>
```

其中，3 行：属性 rootTagName 指定写入文件的根节点名称。

4 行：属性 marshaller 指定序列化的组件。

5 行：属性 resource 指定写入的文件路径，为 target/ch07/xml/credit-card-bill.xml。

7~15 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 属性就可以将特定的 Java 对象转换为 XML 文件，此处使用 com.juxtapose.example.ch07.CreditBill，即将 CreditBill 对象转换为 juxtapose/credit 路径下的 XML 文件片段。

使用代码清单 7-23 执行定义的 xmlFileReadAndWriteJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchXml。

代码清单 7-23 执行 xmlFileReadAndWriteJob

```
1. JobLaunchBase.executeJob("ch07/job/job-xml.xml",
2. "xmlFileReadAndWriterJob",
2.     new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容参见代码清单 7-24。

代码清单 7-24 写入后的 credit-card-bill.xml 文件内容

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <juxtapose>
3.     <credit>
4.         <accountID>4047390012345678</accountID>
5.         <name>tom</name>
6.         <amount>100.0</amount>
7.         <date>2013-2-2 12:00:08</date>
```

```
8.          <address>Lu Jia Zui road</address>
9.      </credit>
10.     .....
11. </juxtapose>
```

读者注意，写入后的 XML 的根节点为 juxtapose，是通过属性 rootTagName 进行指定的。

7.3.2 回调操作

在 XML 类型的写入操作中，通常在写入文件之前和写入文件之后需要写入额外的内容，例如注释，写入开始时间，写入完成时间等信息。Spring Batch 框架对 StaxEventItemWriter 增加了写入之前和写入之后的回调操作 StaxWriterCallback，可以通过属性 headerCallback 和 footerCallback 来分别指定。

属性 headerCallback 定义写文件头的回调类，在写记录之前会先调用该回调操作，通常在 open() 操作中调用该回调操作。

属性 footerCallback 定义写文件尾的回调类，在写记录完成后会调用该回调操作，通常在 close() 操作中调用该回调操作。

接口声明

接口 org.springframework.batch.item.xml.StaxWriterCallback 定义，参见代码清单 7-25。

代码清单 7-25 StaxWriterCallback 接口定义

```
1. public interface StaxWriterCallback {
2.     void write(XMLEventWriter writer) throws IOException;
3. }
```

write() 操作通常在 ItemWriter 的 open() 或者 close() 操作时被触发，负责向指定的 XML 文件写入内容。

StaxEventItemWriter、StaxWriterCallback（可以用属性 headerCallback 和 footerCallback 指定写文件头或者写文件尾）的序列图参见图 7-9。

接口实现

接下来我们实现回调接口，在写入的文件头和文件末尾处写入开始、结束日期及注释信息；本例中在文件头部增加一条 XML 注释信息“<!--credit 201310 begin.-->”，在文件末尾处增加一条 XML 注释信息“<!--Total write count = 3;credit 201310 end.-->”，包含成功写入的记录总条数的信息。</p>

HeaderStaxWriterCallback 的实现参见代码清单 7-26。

类 HeaderStaxWriterCallback 完整代码，参见：com.juxtapose.example.ch07.xml.HeaderStaxWriterCallback。

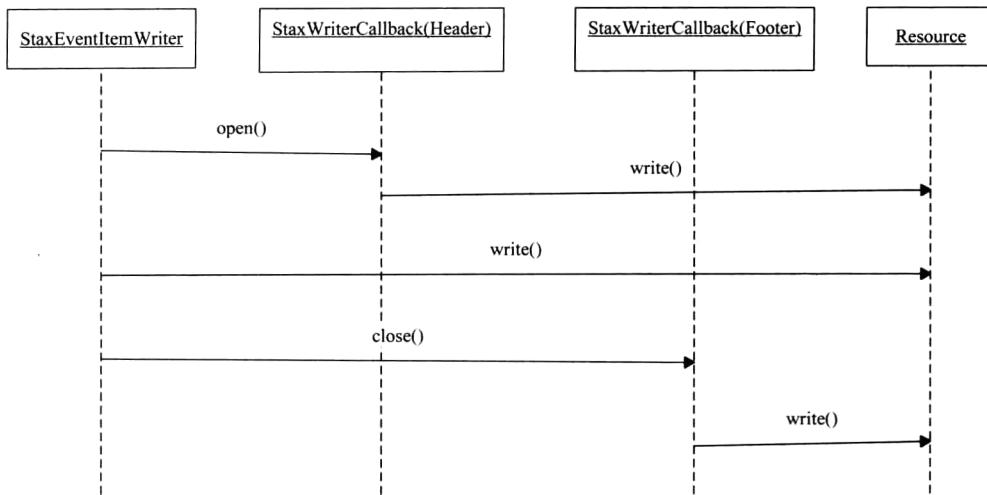


图 7-9 StaxEventItemWriter、StaxWriterCallback 的序列图

代码清单 7-26 HeaderStaxWriterCallback 类定义

```

1. public class HeaderStaxWriterCallback implements StaxWriterCallback {
2.     public void write(XMLEventWriter writer) throws IOException {
3.         XMLEventFactory factory = XMLEventFactory.newInstance();
4.         try {
5.             writer.add(factory.createComment("credit 201310 begin."));
6.         } catch (XMLStreamException e) {
7.             e.printStackTrace();
8.         }
9.     }
10. }

```

其中，5 行：在文件的头部写入注释“credit 201310 begin.”。

为了实现获取成功写入的总条目，我们需要在尾部的回调操作中获取作业步的执行上下文，因此我们让 FooterStaxWriterCallback 实现接口 StepExecutionListener，以方便获取作业步的执行上下文。

FooterStaxWriterCallback 的实现参见代码清单 7-27。

类 FooterStaxWriterCallback 完整代码参见：com.juxtapose.example.ch07.xml.FooterStaxWriterCallback。

代码清单 7-27 FooterStaxWriterCallback 类定义

```

1. public class FooterStaxWriterCallback extends StepExecutionListenerSupport
2.     implements StaxWriterCallback {
3.     public void write(XMLEventWriter writer) throws IOException {
4.         XMLEventFactory factory = XMLEventFactory.newInstance();

```

```

5.         try {
6.             writer.add(factory.createComment("Total write count = "
7.                 + stepExecution.getWriteCount() + ";credit 201310 end."));
8.         } catch (XMLStreamException e) {
9.             e.printStackTrace();
10.        }
11.    }
12.
13.    public void beforeStep(StepExecution stepExecution) {
14.        this.stepExecution = stepExecution;
15.    }
16. }

```

其中，1行：继承 StepExecutionListenerSupport 类，实现了 StepExecutionListener 接口，在实现类中获取作业步执行上下文 StepExecution。

6~7行：在文件的尾部写入注释信息，注释包括写入的总行数和写入的日期及结束标志 "end."。

13~15行：实现 beforeStep() 操作，方便在回调实现类 FooterStaxWriterCallback 中获取作业步执行上下文 Step Execution，通过作业步执行上下文 Step Execution 可以方便地获取写入的条目总数等信息。

配置回调

配置 StaxEventItemWriter 的回调操作，配置代码参见代码清单 7-28。

完整配置参见文件：ch07/job/job-flatfile-callback.xml。

代码清单 7-28 配置 StaxEventItemWriter 的回调操作

```

1.      <job id="xmlFileReadAndWriterJob">
2.          <step id="xmlFileReadAndWriterStep">
3.              <tasklet>
4.                  <chunk reader="xmlReader" writer="xmlWriter" commit-
5.                      interval="2"/>
6.                  <listeners>
7.                      <listener ref="footerCallback"/>
8.                  </listeners>
9.              </tasklet>
10.             </step>
11.         </job>
12.         <bean:bean id="xmlWriter"
13.             class="org.springframework.batch.item.xml.StaxEventItemWriter">
14.             <bean:property name="rootTagName" value="juxtapose"/>
15.             <bean:property name="marshaller" ref="creditMarshaller"/>
16.             <bean:property name="resource" />

```

```

16.           value="file:target/ch07/xml/callback/credit-card-bill.xml"/>
17.           <bean:property name="headerCallback" ref="headerCallback" />
18.           <bean:property name="footerCallback" ref="footerCallback" />
19.       </bean:bean>
20.       <bean:bean id="headerCallback"
21.           class="com.juxtapose.example.ch07.xml.
22.           HeaderStaxWriterCallback"/>
23.       <bean:bean id="footerCallback"
24.           class="com.juxtapose.example.ch07.xml.
25.           FooterStaxWriterCallback"/>

```

其中,6行:定义作业步执行拦截器 footerCallback,可以方便地在 FooterStaxWriterCallback 中获取作业步执行上下文。

17行:属性 headerCallback 定义文件写入之间的回调操作。

18行:属性 footerCallback 定义文件写入完成之后的回调操作。

20~21行:定义文件写入之间的回调类 com.juxtapose.example.ch07.xml.HeaderStaxWriter Callback。

22~23行:定义文件写入之后的回调类 com.juxtapose.example.ch07.xml.FooterStaxWriter Callback。

使用代码清单 7-29 执行定义的 xmlFileReadAndWriterJob。

完整代码参见: test.com.juxtapose.example.ch07.JobLaunchXmlCallback。

代码清单 7-29 执行 xmlFileReadAndWriterJob

```

1. JobLaunchBase.executeJob("ch07/job/job-xml-callback.xml",
2. "xmlFileReadAndWriterJob",
3. new JobParametersBuilder().addDate("date", new Date()));

```

写入的文件内容,参见代码清单 7-30。

代码清单 7-30 写入后的 credit-card-bill.xml 文件内容

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <juxtapose>
3.   <!-- credit 201310 begin. -->
4.   <credit>
5.     <accountID> 4047390012345678</accountID>
6.     <name> tom</name>
7.     <amount> 100.0</amount>
8.     <date> 2013-2-2 12:00:08</date>
9.     <address> Lu Jia Zui road</address>
10.    </credit>
11.    .....
12.    <!--Total write count = 3;credit 201310 end. -->
13. </juxtapose>

```

其中，3行：回调实现 HeaderStaxWriterCallback 写入的头信息。

12行：回调实现 FooterStaxWriterCallback 写入的尾信息，包含写入记录的总条数。

7.4 写多文件

前面章节完整地介绍了对 Flat 格式、XML 格式文件的写操作，细心的读者会发现上面所有的文件读取基本上是写入单文件。在实际的应用中，批量处理的文件数量非常大，如果全部写入一个文件会导致文件非常大；本节将为读者介绍如何写入多个文件。以信用卡账单为例，在处理信用卡账单时，每个文件存放 100 条记录，超过 100 条之后自动写入一个新的文件，避免每个文件非常大，类似于于 Log4j 中的分文件记录日志的能力，参见图 7-10。

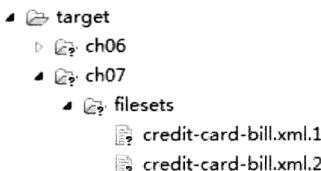


图 7-10 写多文件示例

7.4.1 MultiResourceItemWriter

Spring Batch 框架提供了现有的组件 MultiResourceItemWriter 支持对多文件的写入，通过 MultiResourceItemWriter 写入批量文件非常简单。MultiResourceItemWriter 通过代理的 ItemWriter 来读取文件。

MultiResourceItemWriter 结构关键属性

MultiResourceItemWriter 组件实现 ItemWriter、ItemStream 接口，并引用实现了接口 ResourceAwareItemWriterItemStream 的读组件，例如 FlatFileItemWriter、StaxEventItemWriter，通过这些具体的组件完成文件的读取；ResourceSuffixCreator 为文件后缀生成器接口，根据给定的 Resource 生成不同的文件后缀名，默认情况下使用 SimpleResourceSuffixCreator 作为默认实现。

ResourceSuffixCreator 接口定义参见代码清单 7-31。

代码清单 7-31 ResourceSuffixCreator 接口定义

```
1. public interface ResourceSuffixCreator {  
2.     String getSuffix(int index);  
3. }
```

其中，2行：根据当前序列 index 生成文件后缀名称。

SimpleResourceSuffixCreator 的实现非常简单，参见代码清单 7-32。

代码清单 7-32 SimpleResourceSuffixCreator 类定义

```
1. public class SimpleResourceSuffixCreator implements ResourceSuffixCreator {  
2.     public String getSuffix(int index) {  
3.         return "." + index;  
4.     }  
5. }
```

其中，3 行：文件名为.+index，其中 index 表示当前的文件序列。

基于 SimpleResourceSuffixCreator 生成的文件名为：

credit-card-bill.xml.1

credit-card-bill.xml.2

.....

MultiResourceItemWriter 核心类结构图，参见图 7-11。

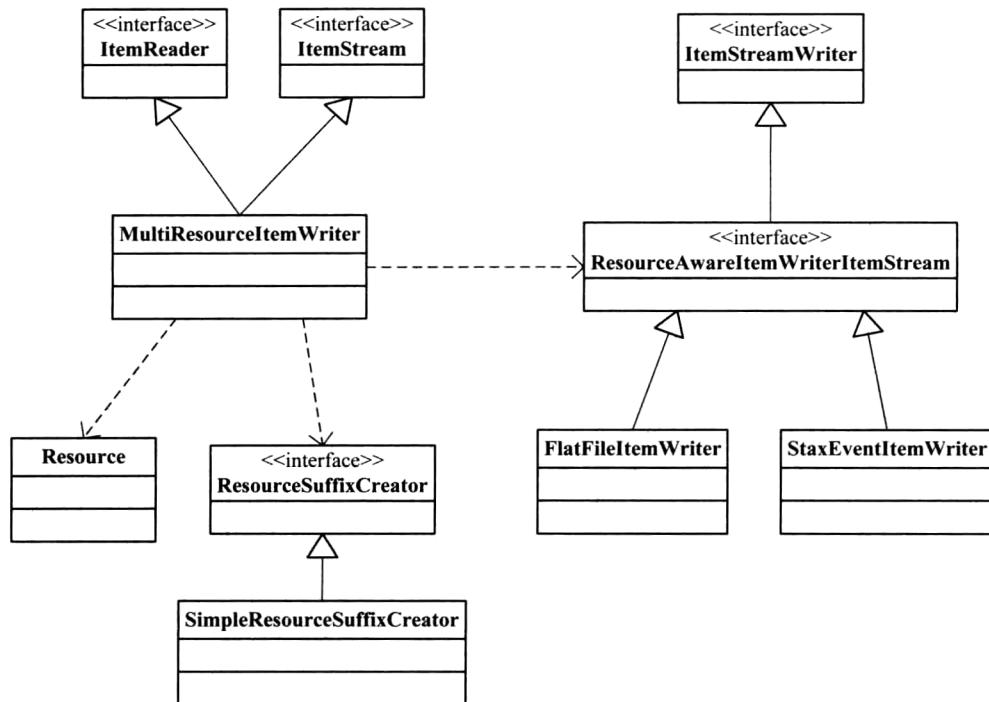


图 7-11 MultiResourceItemWriter 核心类结构图

MultiResourceItemWriter 关键属性参见表 7-8。

配置 MultiResourceItemWriter

本章例子仍然使用上节中的 XML 写入示例，每 2 条记录写入一个新的文件中。配置 MultiResourceItemWriter 的示例参见代码清单 7-33。

完整的配置文件参见：ch07/job/job-filesets.xml。

表 7-8 MultiResourceItemWriter 关键属性

StaxEventItemReader 属性	类 型	说 明
delegate	ResourceAwareItemWriterItemStream	ItemWriter 的代理，将 resources 中定义的文件代理给当前指定的 ItemWriter 进行处理
resource	Resource	需要写入的资源文件名称
suffixCreator	ResourceSuffixCreator	文件后缀生成器，根据给定的文件序列生成文件的后缀名；需实现接口 ResourceSuffixCreator，默认的实现 SimpleResourceSuffixCreator 使用的后缀名为“.index”，index 表示当前的文件序号。 默认值：SimpleResourceSuffixCreator
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读取到条目记录的状态。 默认值:true
itemCountLimitPerResource	int	每个文件可以写入的最大条目，当超过该数值后，会自动写入下一个文件。 默认值：Integer.MAX_VALUE

代码清单 7-33 配置 MultiResourceItemWriter

```
1.      <bean:bean id="multiItemWriter"
2.          class="org.springframework.batch.item.file.
3.              MultiResourceItemWriter" >
4.      <bean:property name="resource"
5.          value="file:target/ch07/credit-card-bill.xml" />
6.      <bean:property name="itemCountLimitPerResource" value="2" />
7.      <bean:property name="delegate" ref="xmlWriter" />
8.  </bean:bean>
9.  <bean:bean id="xmlWriter"
10.     class="org.springframework.batch.item.xml.StaxEventItemWriter">
11.     <bean:property name="rootTagName" value="juxtapose"/>
12.     <bean:property name="marshaller" ref="creditMarshaller"/>
    </bean:bean>
```

其中，4 行：属性 resource 指定需要写入的文件资源，其父目录必须存在。

5 行：属性 itemCountLimitPerResource 定义每个文件能写入的最大条目数，本例定义为 2，表示每个文件只能写入 2 条记录。

6 行：属性 delegate 用于指定代理的真正写入的 ItemWriter 的实现类。

8~12 行：定义 xmlWriter 的实现类。

使用代码清单 7-34 执行定义的 filesetsWriteJob。

完整的代码参见：test.com.juxtapose.example.ch07.JobLaunchFileSets。

代码清单 7-34 执行 filesetsWriteJob

```
1. JobLaunchBase.executeJob("ch07/job/job-filesets.xml",
   "filesetsWriterJob",
2.           new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容如图 7-12 所示。

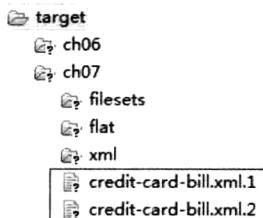


图 7-12 写入的文件内容

说明： MultiResourceItemWriter 中指定的 Resource 其父目录是必须存在的，否则运行期会报如下的错误。修改 multiItemWriter 中的 resource 属性值为："file:target/ch07/notexist/credit-card-bill.xml"，重新执行 JobLaunchFileSets，控制台会出现如下的错误，参见代码清单 7-35。

代码清单 7-35 控制台错误输出

```
java.io.IOException: 系统找不到指定的路径。
  at java.io.WinNTFileSystem.createFileExclusively(Native Method)
  at java.io.File.createNewFile(File.java:883)
```

7.4.2 扩展 MultiResourceItemWriter

MultiResourceItemWriter 要求给定的 Resource 的父目录必须存在在，实际使用中可能存在的问题较多，本节我们扩展 MultiResourceItemWriter 实现一个新的支持给定的 Resource，其父目录允许不存在。MultiResourceItemWriter 本身将所有的实现都变成 private 的方式，导致无法通过继承 MultiResourceItemWriter 的方式来扩展，本章节直接使用源代码的方式实现一个新的多文件写的实现类： com.juxtapose.example.ch07.multiresource.ext.ExtMultiResourceItemWriter<T>。

读者可以自行比较 ExtMultiResourceItemWriter 与 MultiResourceItemWriter 的差别，最主要的差别是在 write 方法中创建文件的时候增加了空目录的判断。新的 ExtMultiResourceItemWriter 在 write() 操作中增加了如下的代码片段，具体参见代码清单 7-36。

代码清单 7-36 新的 ExtMultiResourceItemWriter 类定义

```
1. //modify 20130921 begin
2. if (file.getParent() != null) {
3.     new File(file.getParent()).mkdirs();
4. }
5. //modify 20130921 end
```

配置 ExtMultiResourceItemWriter

本章的例子仍然使用上节中的 XML 写入示例，每 2 条记录写入一个新的文件中。配置 ExtMultiResourceItemWriter 代码参见代码清单 7-37。

完整的配置文件参见：ch07/job/job-filesets.xml。

代码清单 7-37 配置 ExtMultiResourceItemWriter

```
1.      <bean:bean id="extMultiItemWriter"
2.          class="com.juxtapose.example.ch07.multiresource.ext.
3.              ExtMultiResourceItemWriter" >
4.      <bean:property name="resource"
5.          value="file:target/ch07/filesets/credit-card-bill.xml" />
6.      <bean:property name="itemCountLimitPerResource" value="2" />
7.      <bean:property name="delegate" ref="xmlWriter" />
8.  </bean:bean>
```

其中，3 行：属性 resource 指定需要写入的文件资源，其父目录可以不存在。

5 行：属性 itemCountLimitPerResource 定义每个文件能写入的最大条目数，本例子定义为 2，表示每个文件只能写入 2 条记录。

6 行：属性 delegate 用于指定代理的真正写入的 ItemWriter 的实现类。

使用代码清单 7-38 执行定义的 extFilesetsWriterJob。

完整的代码参见：test.com.juxtapose.example.ch07.JobLaunchFileSetsExt。

代码清单 7-38 执行 extFilesetsWriterJob

```
1. JobLaunchBase.executeJob("ch07/job/job-filesets.xml",
2.     "extFilesetsWriterJob",
3.     new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容如图 7-13 所示。

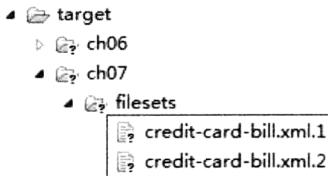


图 7-13 写入后的多文件示例

7.5 写数据库

Spring Batch 框架对写数据库提供了较好的支持，包括基于 JDBC 和 ORM (Object-Relational Mapping) 的写入方式。

Spring Batch 框架提供的写数据库组件列表参见表 7-9。

表 7-9 Spring Batch 框架提供的写数据库组件

JdbcBatchItemWriter	基于 JDBC 写数据库组件
HibernateItemWriter	基于 Hibernate 写数据库组件
IbatisBatchItemWriter	基于 Ibatis 写读数据库组件
JpaItemWriter	基于 Jpa 方式写读数据组件

7.5.1 JdbcBatchItemWriter

Spring Batch 框架提供了对 JDBC 写支持的组件 JdbcBatchItemWriter。JdbcBatchItemWriter 实现 Itemwriter 接口，其核心作用是将 Item 对象转换为数据库中的记录。JdbcBatchItemWriter 通过引用 ItemPreparedStatementSetter、ItemSqlParameterSourceProvider、DataSource 关键接口实现上面的目的。

JdbcBatchItemWriter 对用户屏蔽了数据库访问的操作细节，且提供了批处理的特性，JdbcBatchItemWriter 会批量执行一组 SQL 语句来提高性能，而不是逐条执行 SQL 语句；每次批量提交的语句数和 Chunk 中定义的提交间隔是一致的。

JdbcBatchItemWriter 结构及关键属性

JdbcBatchItemWriter 核心操作步骤，参见图 7-14。

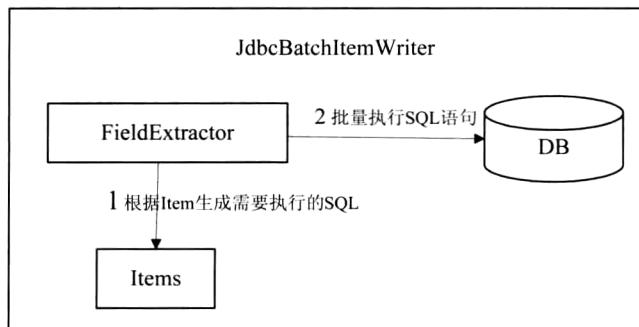


图 7-14 JdbcBatchItemWriter 核心操作步骤

JdbcBatchItemWriter 与关键接口 ItemPreparedStatementSetter、ItemSqlParameterSource Provider、DataSource 之间的类图参见图 7-15。

JdbcBatchItemWriter 引用 javax.sql.DataSource，DataSource 提供数据库信息；JdbcBatchItemWriter 通过 NamedParameterJdbcTemplate 对数据库进行操作；执行的 SQL 语句如果有"?"的参数，可以通过接口 ItemPreparedStatementSetter 进行指定；执行的 SQL 语句如果使用有名字的参数，需要通过接口 ItemSqlParameterSourceProvider 进行声明。

JdbcBatchItemWriter 关键接口、类说明参见表 7-10。

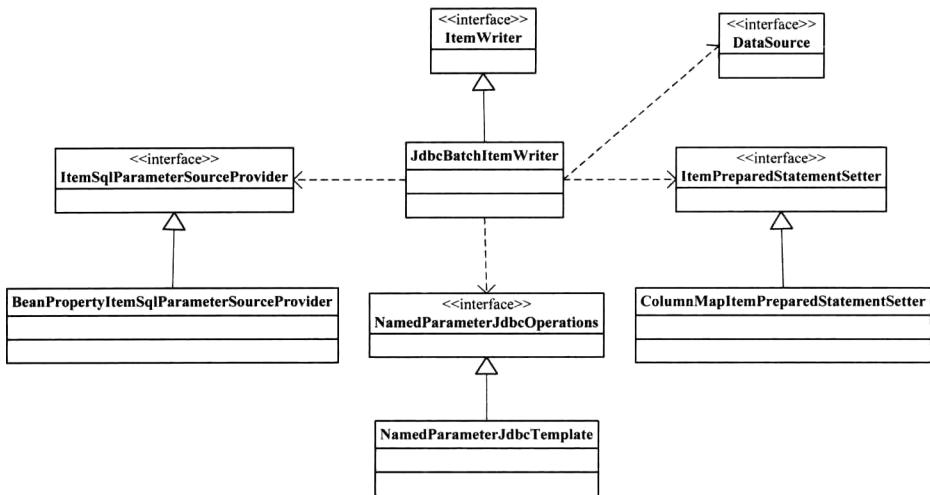


图 7-15 JdbcBatchItemWriter 类关系图

表 7-10 JdbcBatchItemWriter 关键接口、类说明

关键类	说 明
DataSource	提供写入数据库的数据源信息
ItemPreparedStatementSetter	为 SQL 语句中有"?"的参数提供赋值接口
ColumnMapItemPreparedStatementSetter	接口 ItemPreparedStatementSetter 的实现类, 提供基于列的参数设置
ItemSqlParameterSourceProvider	为 SQL 语句中有命名的参数提供赋值接口
BeanPropertyItemSqlParameterSourceProvider	从给定的 Item 中根据参数名称获取 Item 对应的属性值作为参数
NamedParameterJdbcOperations	JDBCTemplate 操作, 提供执行 SQL 的能力

JdbcBatchItemWriter 关键属性参见表 7-11。

表 7-11 JdbcBatchItemWriter 关键属性

JdbcBatchItemWriter 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
sql	String	执行的 SQL 语句
itemSqlParameterSourceProvider	ItemSqlParameterSourceProvider	为 SQL 语句中有命名的参数提供赋值
itemPreparedStatementSetter	ItemPreparedStatementSetter	为 SQL 语句中有"?"的参数提供赋值
assertUpdates	Boolean	当没有修改、删除一条记录时候, 是否抛出异常。 默认值: true

配置 JdbcBatchItemWriter

使用 JdbcBatchItemWriter 至少需要配置 dataSource、sql 两个属性；dataSource 指定访问的数据源，sql 用于指定查询的 SQL 语句。

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；示例从表 t_credit 读取所有的记录，然后通过 JdbcBatchItemWriter 写入到表 t_destcredit 中。

建表脚本参见代码清单 7-39。

完整内容参见文件 ch07/db/create-tables-mysql.sql。

代码清单 7-39 数据库建表脚本

```
1. CREATE TABLE t_destcredit
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 JdbcBatchItemWriter 参见代码清单 7-40。

完整的配置文件参见：ch07/job/job-db-jdbc.xml。

代码清单 7-40 配置 JdbcBatchItemWriter

```
1. <bean:bean id="jdbcItemWriter"
2.     class="org.springframework.batch.item.database.
JdbcBatchItemWriter">
3.     <bean:property name="dataSource" ref="dataSource"/>
4.     <bean:property name="sql"
5.         value="insert into t_destcredit (ID,ACCOUNTID,NAME,AMOUNT,
6.             DATE,ADDRESS)
7.                 values (:id,:accountID,:name,:amount,:date,:address)"/>
8.     <bean:property name="itemSqlParameterSourceProvider">
9.         <bean:bean class="org.springframework.batch.item.database.
BeanPropertyItemSqlParameterSourceProvider"/>
10.    </bean:property>
11. </bean:bean>
```

其中，3 行：属性 dataSource 用于配置访问的数据源。

4~6 行：属性 sql 定义需要执行的 SQL 语句，插入 t_destcredit 表，SQL 语句有命名的参数，需要从输入的 Item 中获取参数值赋值给 SQL 语句中的参数，本节使用属性 itemSqlParameterSourceProvider 为 SQL 语句的参数赋值。

7~10 行：属性 itemSqlParameterSourceProvider 指定 SQL 语句使用的参数，根据参数的名字到 Item 对象中获取对应的属性值。

使用代码清单 7-41 执行定义的 dbWriteJob。

完整的代码参见：test.com.juxtapose.example.ch07.JobLaunchJDBC。

代码清单 7-41 执行 dbWriteJob

```
1. JobLaunchBase.executeJob("ch07/job/job-db-jdbc.xml", "dbWriteJob",
2.     new JobParametersBuilder().addDate("date", new Date()));
```

查看数据库表 t_destcredit，有 5 条记录表名 Job 执行成功。

为?方式参数赋值

上面我们为命名的参数进行了赋值，接下来我们学习如何为带有"?"的 SQL 语句进行赋值；需要使用接口 ItemPreparedStatementSetter。接口 ItemPreparedStatementSetter 定义参见代码清单 7-42。

代码清单 7-42 ItemPreparedStatementSetter 接口定义

```
1. public interface ItemPreparedStatementSetter<T> {
2.     void setValues(T item, PreparedStatement ps) throws SQLException;
3. }
```

接口 ItemPreparedStatementSetter 仅包含一个操作，根据给定的 Item 为 PreparedStatement 进行设置 SQL 语句中的"?"参数。接下来我们实现该接口 com.juxtapose.example.ch07.jdbc.DestCreditBillItemPreparedStatementSetter。

DestCreditBillItemPreparedStatementSetter 实现，参见代码清单 7-43。

代码清单 7-43 DestCreditBillItemPreparedStatementSetter 类定义

```
1. public class DestCreditBillItemPreparedStatementSetter implements
2.     ItemPreparedStatementSetter<DestinationCreditBill> {
3.
4.     public void setValues(DestinationCreditBill item, PreparedStatement ps)
5.         throws SQLException {
6.         ps.setString(1, item.getId());
7.         ps.setString(2, item.getAccountID());
8.         ps.setString(3, item.getName());
9.         ps.setDouble(4, item.getAmount());
10.        ps.setString(5, item.getDate());
11.        ps.setString(6, item.getAddress());
12.    }
13. }
```

重新配置 JdbcBatchItemWriter，具体配置代码参见代码清单 7-44。

完整的配置文件参见：ch07/job/job-db-jdbc.xml。

代码清单 7-44 配置带参数的 JdbcBatchItemWriter

```
1. <bean:bean id="jdbcSetterItemWriter"
2.   class="org.springframework.batch.item.database.
3.   JdbcBatchItemWriter">
4.   <bean:property name="dataSource" ref="dataSource"/>
5.   <bean:property name="sql" value="insert into t_destcredit
6.     (ID,ACCOUNTID,NAME,AMOUNT,DATE,ADDRESS) values (?,?,?,?,?,?)"/>
7.   <bean:property name="itemPreparedStatementSetter">
8.     <bean:bean class="com.juxtapose.example.ch07.jdbc.
9.     DestCreditBillItemPreparedStatementSetter"/>
10.    </bean:property>
11.  </bean:bean>
```

其中，4~5 行：使用带有“?”的 SQL 语句。

6 行：使用自定义的参数设置类 DestCreditBillItemPreparedStatementSetter 为 SQL 语句的参数赋值。

7.5.2 HibernateItemWriter

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 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 提供了写的 ItemWriter。

HibernateItemWriter 实现 ItemWriter 接口，核心作用是将 Item 对象持久化到数据库中。学会使用基于 JDBC 方式的写数据库后，使用 HibernateItemWriter 非常简单。下面我们先看 HibernateItemWriter 的关键支持的属性，参见表 7-12。

HibernateItemWriter 结构及关键属性

表 7-12 HibernateItemWriter 关键属性

HibernateItemWriter 属性	类 型	说 明
clearSession	boolean	在写结束阶段是否将 session 清理掉或者 flush。 默认值： true
hibernateTemplate	HibernateOperations	Hibernate 的模板类，提供基础的操作
sessionFactory	SessionFactory	Hibernate 的 SessionFactory，负责与数据库进行交互

配置 HibernateItemWriter

使用 HibernateItemWriter 仅需要配置属性 hibernateTemplate 属性； hibernateTemplate 属性

性提供了对数据库的持久化访问。

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 7-45）、配置 Hibernate 的.cfg.xml（参见代码清单 7-46）。本节示例仍然使用 JDBC 写章节使用的数据库脚本。示例从表 t_credit 读取所有的记录，然后通过 JdbcBatchItemWriter 写入到表 t_destcredit 中。

配置数据实体对象

完整的内容参见类：com.juxtapose.example.ch07.hibernate.DestinationCreditBill。

代码清单 7-45 配置数据实体对象

```
1.  @Entity
2.  @Table(name="t_destcredit")
3.  public class DestinationCreditBill {
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，而是非映射自数据库的，要用 Transient 来注解。

2 行：@Table(name="t_destcredit")持久性映射的表，表名为“t_destcredit”。@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，hibernate 的 cfg 文件配置参见代码清单 7-46。

完整的文件参见：ch07/cfg/hibernate.cfg.xml。

代码清单 7-46 配置 hibernate 的 cfg 文件

```
1. <hibernate-configuration>
2.   <session-factory>
3.     <mapping class="com.juxtapose.example.ch07.hibernate.CreditBill"/>
4.     <mapping class="com.juxtapose.example.ch07.hibernate.
   DestinationCreditBill"/>
5.   </session-factory>
6. </hibernate-configuration>
```

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

4 行：声明 Hibernate 中使用的数据实体类 DestinationCreditBill。

配置 HibernateItemWriter

完整的配置文件参见：ch07/job/job-db-hibernate.xml。

下面示例展示了将 Item 数据写入数据库表 t_destcredit 中，具体参见代码清单 7-47。

代码清单 7-47 配置 HibernateItemWriter

```
1. <bean:bean id="hibernateItemWriter"
2.           class="org.springframework.batch.item.database.
   HibernateItemWriter">
3.   <bean:property name="hibernateTemplate" ref="hibernateTemplate" />
4. </bean:bean>
5.
6. <bean:bean id="hibernateTemplate"
7.           class="org.springframework.orm.hibernate3.HibernateTemplate">
8.   <bean:property name="sessionFactory" ref="sessionFactory" />
9. </bean:bean>
10.
11. <bean:bean id="sessionFactory"
12.           class="org.springframework.orm.hibernate3.LocalSession
   FactoryBean">
13.   <bean:property name="dataSource" ref="dataSource"/>
14.   <bean:property name="configurationClass"
15.                 value="org.hibernate.cfg.AnnotationConfiguration"/>
16.   <bean:property name="configLocation"
17.                 value="classpath:/ch07/cfg/hibernate.cfg.xml"/>
18.   <bean:property name="hibernateProperties">
```

```

19.          <bean:value>
20.              hibernate.dialect=org.hibernate.dialect.MySQLDialect
21.              hibernate.show_sql=true
22.          </bean:value>
23.      </bean:property>
24.  </bean:bean>

```

其中，3行：属性 hibernateTemplate，定义 Hibernate 的访问模板，提供持久化操作。

6~9行：定义 Hibernate 的模板操作，此处使用 Spring 提供的 org.springframework.orm.hibernate3.HibernateTemplate 提供访问功能。

11~24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 7-48 执行定义的 hibernateWriteJob。

完整的代码参见：test.com.juxtapose.example.ch07.JobLaunchHibernate。

代码清单 7-48 执行 hibernateWriteJob

```

1. JobLaunchBase.executeJob("ch07/job/job-db-hibernate.xml",
    "hibernateWriteJob",
2.         new JobParametersBuilder().addDate("date", new Date()));

```

执行完毕，查看数据库表 t_destcredit，记录被成功写入表 t_destcredit。

7.5.3 IbatisBatchItemWriter

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中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 提供了写的 ItemWriter。

IbatisBatchItemWriter 实现 ItemWriter 接口，核心作用是将 Item 对象通过 Ibatis 的方式写入到数据库中。学会使用基于 HibernatePagingItemReader 的读数据库后，使用 IbatisBatchItemWriter 非常简单。下面我们先看 IbatisBatchItemWriter 的关键支持的属性，具体内容参见表 7-13。

IbatisBatchItemWriter 结构及关键属性

表 7-13 IbatisBatchItemWriter 关键属性

IbatisBatchItemWriter 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
statementId	String	命名 SQL 定义的 ID

续表

IbatisBatchItemWriter 属性	类 型	说 明
assertUpdates	boolean	是否断言执行的 SQL 对数据库有记录更新，如果没有更新会抛出异常 EmptyResultDataAccessException。 默认值： true

配置 IbatisBatchItemWriter

使用 IbatisBatchItemWriter 至少需要配置 sqlMapClient、statementId 两个属性；sqlMapClient 用于指定配置的命名 SQL 的文件；statementId 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 JDBC 写章节使用的数据库脚本。示例从表 t_credit 读取所有的记录，然后通过 JdbcBatchItemWriter 写入到表 t_destcredit 中。

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。本例中的命名 SQL 对表 t_destcredit 进行操作：提供 id 为 “insertDestCredits” 的命名 SQL 用于新增一条信用卡记录。命名 SQL 具体配置参见代码清单 7-49。

完整的文件参见：ch07/ibatis/ibatis-destcredit.xml。

代码清单 7-49 配置命名 SQL

```

1.  <sqlMap namespace="DestCredit">
2.    <resultMap id="result" class="com.juxtapose.example.ch07.db.
   DestinationCreditBill">
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.
12.   <insert id="insertDestCredits"
13.         parameterClass="com.juxtapose.example.ch07.db.
   DestinationCreditBill">
14.     insert into t_destcredit values(#id#, #accountID#, #name#, #amount#,
   #date#, #address#)
15.   </insert>
16. </sqlMap>
```

其中，1 行：声明当前命名 SQL 文件的命名空间为 “DestCredit”。

2~9 行：定义命名 SQL 的返回值对象，将数据库的列与对象 com.juxtapose.example.ch07.db.DestinationCreditBill 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_destcredit 中的字段 ID 映射到对象 com.juxtapose.example.ch07.db.DestinationCreditBill 的 id 属性上。

12~15 行：提供 id 为“insertDestCredits”的命名 SQL，用于向表 t_destcredit 中插入一条记录。

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，具体参见代码清单 7-50；然后可以在 IbatisBatchItemWriter 中使用命名 SQL 执行数据库的查询。

完整的文件参见：ch07/ibatis/ibatis-config.xml。

代码清单 7-50 配置 ibatis-config.xml

```
1. <sqlMapConfig>
2.   <sqlMap resource="ch07/ibatis/ibatis-credit.xml"/>
3.   <sqlMap resource="ch07/ibatis/ibatis-destcredit.xml"/>
4. </sqlMapConfig>
```

其中，2~3 行：声明具体的 Ibatis 的命名 SQL 文件。

配置 IbatisBatchItemWriter

完整的配置文件参见：ch07/job/job-db-ibatis.xml。

代码清单 7-51 展示了通过命名 SQL 的方式向数据库表 t_destcredit 中插入记录。

代码清单 7-51 配置 IbatisBatchItemWriter

```
1.   <!-- Ibatis 写数据库 -->
2.   <bean:bean id="ibatisItemWriter"
3.     class="org.springframework.batch.item.database.
IbatisBatchItemWriter">
4.     <bean:property name="statementId" value="insertDestCredits" />
5.     <bean:property name="sqlMapClient" ref="sqlMapClient" />
6.   </bean:bean>
7.   <bean:bean id="sqlMapClient"
8.     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
9.     <bean:property name="dataSource" ref="dataSource" />
10.    <bean:property name="configLocation"
11.      value="classpath:/ch07/ibatis/ibatis-config.xml" />
12.    </bean:bean>
```

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

5 行：属性 sqlMapClient 定义命名 SQL 的客户端配置文件，7~12 行给出了该对象的具

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。

7~12 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 7-52，执行定义的 ibatisWriteJob。

完整的代码参见：test.com/juxtapose/example/ch07/JobLaunchIbatis。

代码清单 7-52 执行 ibatisWriteJob

```
1. JobLaunchBase.executeJob("ch07/job/job-db-ibatis.xml",
    "ibatisWriteJob",
2.         new JobParametersBuilder().addDate("date", new Date()));
```

执行完毕，查看数据库表 t_destcredit，记录被成功写入表 t_destcredit。

7.5.4 JpaItemWriter

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 提供了基于写的 ItemWriter。

JpaItemWriter 实现 ItemWriter 接口，核心作用是将 Item 对象转换为数据库中的记录。学会使用基于 JdbcBatchItemWriter 的数据库写后，使用 JpaItemWriter 非常简单。下面我们先看 JpaItemWriter 的关键支持的属性，参见表 7-14。

JpaItemWriter 结构及关键属性

表 7-14 JpaItemWriter 关键属性

JpaItemWriter 属性	类 型	说 明
entityManagerFactory	EntityManagerFactory	JPA 提供的实体管理器的工厂类，用于生成 EntityManager 对象

配置 JpaItemWriter

使用 JpaItemWriter 仅需要配置属性 entityManagerFactory；entityManagerFactory 负责创建 EntityManager，后者负责完成对实体的增删改查等。

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 7-53）、配置 JPA 的

实体映射文件（参见代码清单 7-54）。本节示例仍然使用 7.5.1 章节使用的数据库脚本。

配置数据实体对象

完整的内容参见类 com.juxtapose.example.ch07.jpa.DestinationCreditBill。

代码清单 7-53 配置数据实体对象

```
1.  @Entity
2.  @Table(name="t_destcredit")
3.  public class DestinationCreditBill {
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. }
```

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_destcredit")持久性映射的表，表名为“t_destcredit”。@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。

完整的文件参见：ch07/jpa/persistence.xml。

代码清单 7-54 配置 JPA 的实体映射文件

```
1. <persistence>
2.   <persistence-unit name="creditBill" transaction-type="RESOURCE_LOCAL">
3.     <class>com.juxtapose.example.ch07.jpa.CreditBill</class>
4.     <class>com.juxtapose.example.ch07.jpa.DestinationCreditBill
5.     </class>
6.     <exclude-unlisted-classes>true</exclude-unlisted-classes>
7.   </persistence-unit>
8. </persistence>
```

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

4 行：声明 JPA 中使用的数据实体类 DestinationCreditBill，用于 ItemWriter 中使用。

5 行：声明排除所有未在此声明的实体类。

配置 JpaItemWriter

完整配置文件参见：ch07/job/job-db-jpa.xml。

代码清单 7-55 展示了向数据库表 t_destcredit 中插入数据。

代码清单 7-55 配置 JpaItemWriter

```
1. <!-- jpa 写数据库 -->
2. <bean:bean id="jpaItemWriter"
3.   class="org.springframework.batch.item.database.JpaItemWriter">
4.   <bean:property name="entityManagerFactory" ref="entityManager
   Factory" />
5. </bean:bean>
```

其中，4 行：属性 entityManagerFactory，定义 JPA 的实体管理器对象，提供访问数据库表的操作；JPA 的实体管理器对象配置参见代码清单 7-56。

代码清单 7-56 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="persistenceXmlLocation"
5.     value="classpath:/ch07/jpa/persistence.xml" />
6.   <bean:property name="jpaVendorAdapter">
7.     <bean:bean class="org.springframework.orm.jpa.vendor.
   HibernateJpaVendorAdapter">
8.       <bean:property name="showSql" value="true" />
9.     </bean:bean>
10.   </bean:property>
11.   <bean:property name="jpaDialect">
```

```
12.      <bean:bean class="org.springframework.orm.jpa.vendor.  
    HibernateJpaDialect" />  
13.    </bean:property>  
14. </bean:bean>
```

其中，4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 7-57 执行定义的 jpaWriteJob。

完整的代码参见：test.com.juxtapose.example.ch07.JobLaunchJpa。

代码清单 7-57 执行 jpaWriteJob

```
1. JobLaunchBase.executeJob("ch07/job/job-db-jpa.xml", "jpaWriteJob",  
2.                     new JobParametersBuilder().addDate("date", new Date())  
3.                     .addString("id_begin", "1").addString("id_end", "5"));
```

执行完毕，查看数据库表 t_destcredit，记录被成功写入表 t_destcredit。

7.6 写 JMS 队列

JMS (Java Messaging Service) 是 Java 平台上有关面向消息中间件 (MOM) 的技术规范，它便于消息系统中的 Java 应用程序进行消息交换，并且通过提供标准的产生、发送、接收消息的接口。JMS 是一种与厂商无关的 API，用来访问消息收发系统消息。它类似于 JDBC (Java Database Connectivity)，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。

Spring 的 JMS 抽象框架简化了 JMS API 的使用，并与 JMS 提供者(比如 IBM 的 WebSphere MQ、开源的 ActiveMQ 等)平滑地集成。Spring JMS 框架提供了 JMS 访问的模板类 JmsTemplate，模板类处理资源的创建和释放，简化了 JMS 的使用。Spring Batch 框架基于 Spring JMS 框架提供了对 JMS 队列写入的 ItemWriter。

7.6.1 JmsItemWriter

JmsItemWriter 实现 ItemWriter 接口，它的核心作用是将 Item 对象转换为 Message 后发送到 JMS 队列。

JmsItemWriter 结构关键属性

图 7-16 展示了写 JMS 队列的逻辑架构图，JmsOperations 负责将 Item 对象转换为 Message 消息，并发送到指定的队列中。

JmsItemWriter 核心类架构图参见图 7-17。

JmsItemWriter 将发送消息全部代理给 JmsOperations，JmsOperations 将 Item 对象转换为 Message 对象后发送到 JmsOperations 默认指定的队列中。JmsItemWriter 关键属性参见表 7-15。

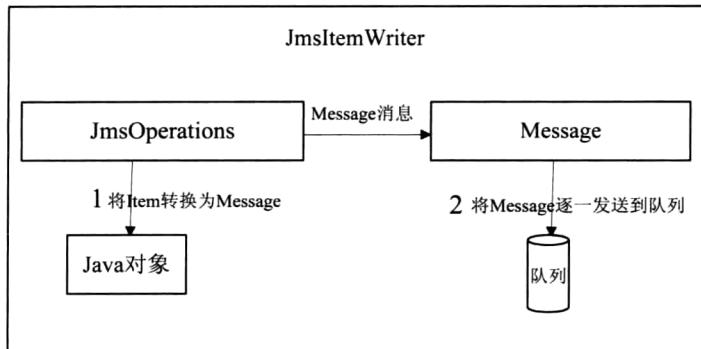


图 7-16 写 JMS 队列的逻辑架构图

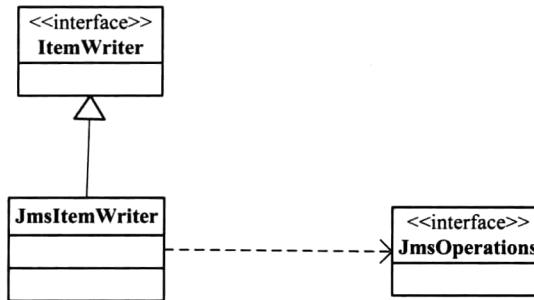


图 7-17 JmsItemWriter 核心类架构图

表 7-15 JmsItemWriter 关键属性

JmsItemWriter 属性	类 型	说 明
jmsTemplate	JmsOperations	消息发送模板

配置 JmsItemWriter

现在假设所有的信用卡账单需要通过 JMS 消息的方式发送到指定的消息队列，需要使用 Spring Batch 框架提供的 JmsItemWriter 来实现该功能。配置 JmsItemWriter 非常简单，只需要指定属性 jmsTemplate 即可，配置代码参见代码清单 7-58。本示例使用 Apache 的 ActiveMQ 作为消息中间件。

完整的配置参见文件 ch07/job/job-jms.xml。

代码清单 7-58 配置 JmsItemWriter

```

1.      <bean:bean id="jmsItemWriter"
2.          class="org.springframework.batch.item.jms.JmsItemWriter">
3.              <bean:property name="jmsTemplate" ref="jmsTemplate" />
4.      </bean:bean>

```

其中，4 行：属性 jmsTemplate 指定读取 JMS 消息的模板，用于读取指定队列中的消息；

JMS 消息模板的配置参见代码清单 7-59。

代码清单 7-59 JMS 消息模板的配置

```
1.      <bean:bean id="jmsTemplate" class="org.springframework.jms.core.  
JmsTemplate">  
2.          <bean:property name="connectionFactory" ref="jmsFactory"/>  
3.          <bean:property name="defaultDestination" ref="creditDestination"/>  
4.          <bean:property name="receiveTimeout" value="500"/>  
5.      </bean:bean>  
6.  
7.      <amq:broker useJmx="false" persistent="false">  
8.          <amq:transportConnectors>  
9.              <amq:transportConnector uri="tcp://localhost:61616" />  
10.         </amq:transportConnectors>  
11.     </amq:broker>  
12.  
13.     <amq:connectionFactory id="jmsFactory" brokerURL="tcp://localhost:  
61616"/>  
14.     <amq:queue id="creditDestination" physicalName="destination.  
creditBill" />
```

其中，2 行：属性 connectionFactory 用于配置 JMS 的连接工厂。

3 行：属性 defaultDestination 指定需要读取的目标消息队列。

4 行：属性 receiveTimeout 指定读取消息的超时时间。

7~11 行：定义 AMQ 的 broker，提供 JMS 的服务器端，指定对应的 ip 与 port；属性 persistent 表示不让消息持久化；属性 schedulerSupport 设置为 false，禁止掉 AMQ 的延迟发送的功能，可避免因为 AMQ 异常终止后导致无法启动。

13 行：定义 JMS 的连接工厂。

14 行：定义消息存储的队列，AMQ 启动时会自动创建名字为"destination.creditBill"的队列。

增加拦截器，验证发送消息成功。

增加作业步执行拦截器 StepExecutionListener，在 afterStep 中使用 JMSTemplate 读取消息。新实现的拦截器为 JMSDetectItemWriteListener，核心代码参见代码清单 7-60。

完整代码参见：com.juxtapose.example.ch07.jms.JMSDetectItemWriteListener。

代码清单 7-60 JMSDetectItemWriteListener 类定义

```
1.  public class JMSDetectItemWriteListener implements StepExecutionListener{  
2.      private JmsTemplate jmsTemplate;  
3.  
4.      public void beforeStep(StepExecution stepExecution) {}  
5.  
6.      public ExitStatus afterStep(StepExecution stepExecution) {
```

```

7.         int writeCount = 0;
8.         Object obj = jmsTemplate.receiveAndConvert();
9.         while(null != obj){
10.             writeCount++;
11.             CreditBill result = (CreditBill) obj;
12.             System.out.println("Receive from jms queue:"+result);
13.             obj = jmsTemplate.receiveAndConvert();
14.         }
15.         Assert.assertEquals(stepExecution.getWriteCount(), writeCount);
16.         return stepExecution.getExitStatus();
17.     }
18.
19.     public void setJmsTemplate(JmsTemplate jmsTemplate) {
20.         this.jmsTemplate = jmsTemplate;
21.     }
22. }
```

其中，6~17 行：使用 jmsTemplate 从消息队列中读取消息，打印在控制台，并使用断言确认从队列中读取消息个数与从 stepExecution 中获取写入队列的消息个数是一致的。

由于引用了新的命名空间 amq，需要在头文件中定义命名空间，参见代码清单 7-61。

代码清单 7-61 声明 amq 命名空间

```

1. <bean:beans xmlns="http://www.springframework.org/schema/beans"
2.   xmlns:amq="http://activemq.apache.org/schema/core"
3.   xsi:schemaLocation="
4.     http://activemq.apache.org/schema/core
5.     http://activemq.apache.org/schema/core/activemq-core.xsd">
6.   .....
7. </bean:beans>
```

Job 及拦截器的配置参见代码清单 7-62。

完整的配置参见文件 ch07/job/job-jms.xml。

代码清单 7-62 配置 Job 及拦截器

```

1. <job id="jmsWriteJob">
2.   <step id="jmsWriteStep">
3.     <tasklet transaction-manager="transactionManager">
4.       <chunk reader="flatFileItemReader"
5.             processor="creditBillProcessor"
6.             writer="jmsItemWriter" commit-interval="2">
7.         <listeners>
8.           <listener ref="jmsDetectItemWriteListener"/>
9.         </listeners>
10.        </chunk>
11.    </tasklet>
```

```

11.      </step>
12.    </job>
13.    <bean:bean id="jmsDetectItemWriteListener"
14.      class="com.juxtapose.example.ch07.jms.
JMSDetectItemWriteListener">
15.      <bean:property name="jmsTemplate" ref="jmsTemplate" />
16.    </bean:bean>

```

其中，7行：为作业 jmsWriteJob 增加作业步拦截器，用于验证消息成功写入队列。

13~16行：声明作业步拦截器的实现 JMSDetectItemWriteListener。

使用代码清单 7-63 执行定义的 jmsWriteJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchJMS。

代码清单 7-63 执行 jmsWriteJob

```

1. JobLaunchBase.executeJob("ch07/job/job-jms.xml", "jmsWriteJob",
2.                           new JobParametersBuilder().addDate("date", new Date()));

```

如果没有任何错误产生，表示消息成功写入到指定的队列 creditDestination 中，并经过拦截器 JMSDetectItemWriteListener 断言后，确认写入消息数与读出的消息个数一致。至此我们学完了 JMS 消息的写入。

7.7 组合写

在 Spring Batch 框架中对于 Chunk 只能配置一个 ItemWriter，但在有些业务场景中需要将一个 Item 同时写到多个不同的资源文件中，即需要写入到多个 ItemWriter 中。Spring Batch 框架提供了组合 ItemWriter（CompositeItemWriter）的模式满足上面的需求。组合 ItemWriter 完成的功能参见图 7-18。

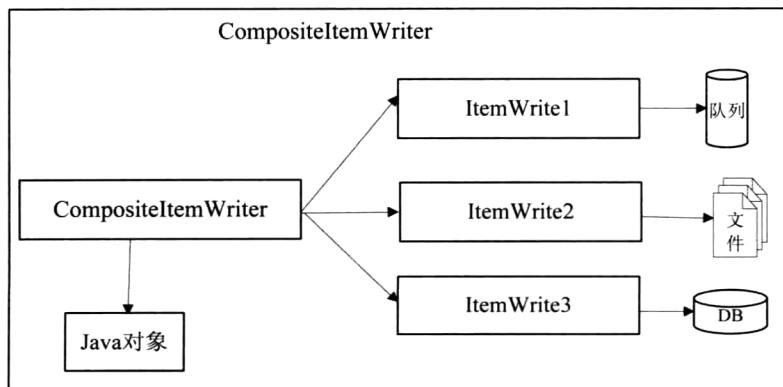


图 7-18 组合 ItemWriter 功能示意图

需要注意的是，不同的 ItemWriter 写入的记录数是完全相同的。

CompositeItemWriter 结构关键属性

CompositeItemWriter 组件实现 ItemWriter、ItemStream 接口，并引用一组 ItemWriter 实现类，CompositeItemWriter 在进行写入的时候循环调用 ItemWriter 中的实现，将单个 Item 对象写入到不同的 ItemWriter 中。

CompositeItemWriter 核心类结构图参见图 7-19。

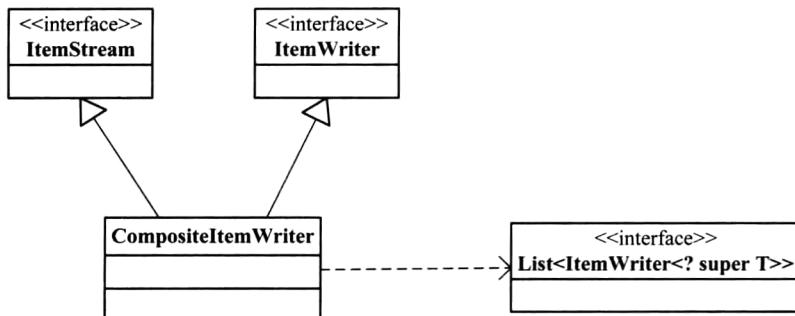


图 7-19 CompositeItemWriter 核心类结构图

CompositeItemWriter 的关键属性参见表 7-16。

表 7-16 CompositeItemWriter 关键属性

CompositeItemWriter 属性	类 型	说 明
delegates	List<ItemWriter<? super T>>	被代理的 ItemWriter 的组合，对于每一个 Item 对象会循环写入到被代理的 ItemWriter 中
ignoreItemStream	boolean	是否忽略被代理的 ItemWriter 实现的 ItemStream 接口的能力。 默认值： false

配置 CompositeItemWriter

在信用卡处理的对象中，需要将信用卡账单写入到不同的 Flat 格式文件中，接下来展示如何使用 CompositeItemWriter 进行组合文件的写入功能。配置 CompositeItemWriter 参见代码清单 7-64。

完整的配置文件参见：ch07/job/job-composite.xml。

代码清单 7-64 配置 CompositeItemWriter

```
1.   <bean:bean id="compositeWriter"
2.     class="org.springframework.batch.item.support.
3.       CompositeItemWriter">
4.     <bean:property name="delegates">
5.       <bean:list>
6.         <bean:ref bean="flatFileItemWriter1" />
```

```
6.           <bean:ref bean="flatFileItemWriter2" />
7.       </bean:list>
8.   </bean:property>
9. </bean:bean>
```

其中，3行：属性 delegates 用于定义需要写入的 ItemWriter，本例中将 Item 对象写入到两个不同 Flat 格式的文件中。

5行：属性 flatFileItemWriter1 将 Item 写入到文件 file:target/ch07/composite/outputFile1.csv 中。

6行：属性 flatFileItemWriter2 将 Item 写入到文件 file:target/ch07/composite/outputFile2.csv 中。

使用代码清单 7-65 执行定义的 compositeWriteJob。

完整的代码参见：test.com.juxtapose.example.ch07.JobLaunchComposite。

代码清单 7-65 执行 compositeWriteJob

```
1. JobLaunchBase.executeJob("ch07/job/job-composite.xml",
   "compositeWriteJob",
2.         new JobParametersBuilder().addDate("date", new Date()));
```

写入的文件内容如图 7-20 所示。

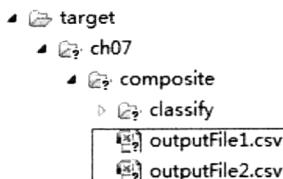


图 7-20 写入后的文件列表

需要注意，outputFile1.csv 和 outputFile2.csv 的内容是完全一样的，这是组合写模式的特点所在，即将 Item 对象写入所有的 ItemWriter 中。

7.8 Item 路由 Writer

上节我们学习了如何将一个 Item 对象写入到一组 ItemWriter 中。另外一些业务场景需要将不同的 Item 写入到不同的 ItemWriter 中，举例在信用卡账单对象处理过程中需要根据消费的金额将记录写入到不同的文件中：消费金额大于 500 的写入单独的文件，其他小于等于 500 的需要写入另外一个文件中。Spring Batch 框架提供了支持 Item 路由写的组件 ClassifierCompositeItemWriter。路由 ItemWriter 完成的功能如图 7-21 所示。

需要注意的是，不同的 ItemWriter 写入的记录数是不同的，根据前面路由的条件来判断写入哪个文件中。

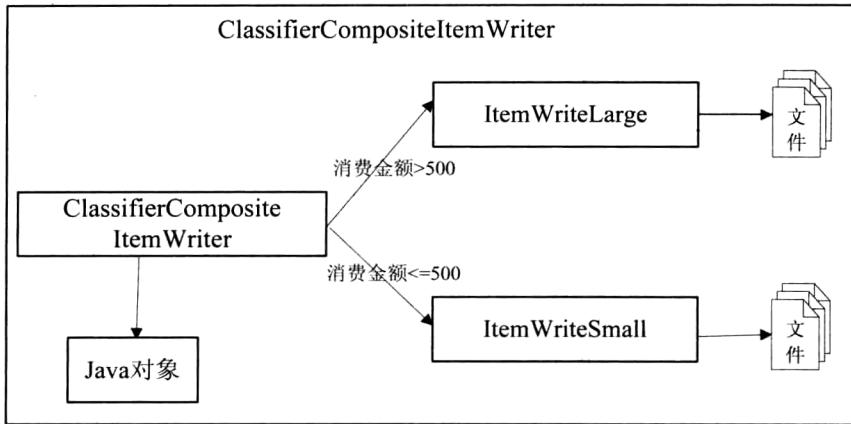


图 7-21 路由 ItemWriter 完成的功能示意图

ClassifierCompositeItemWriter 结构关键属性

ClassifierCompositeItemWriter 组件实现 ItemWriter 接口，Classifier 接口提供路由功能，根据给定的对象返回另外的一个对象；通常可以使用 Classifier 的实现 BackToBackPatternClassifier 来进行路由，BackToBackPatternClassifier 提供根据 Map 的方式支持路由选择。

ClassifierCompositeItemWriter 核心类结构图参见图 7-22。

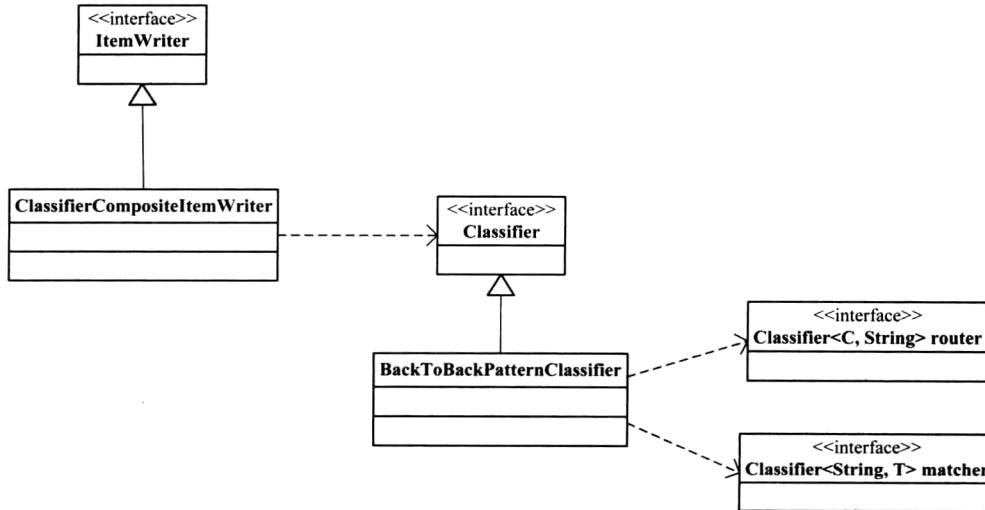


图 7-22 ClassifierCompositeItemWriter 核心类结构图

接口 Classifier 定义参见代码清单 7-66。

接口 Classifier 的完整类路径：org.springframework.classify.Classifier<C, T>。

代码清单 7-66 Classifier 接口定义

```
1. public interface Classifier<C, T> {  
2.     T classify(C classifiable);  
3. }
```

其中，2 行：方法 classify() 提供路由功能，输入参数 C classifiable，返回 T 类型对象。

类 BackToBackPatternClassifier 提供了友好的基于 Map 路由能力，属性 router 根据 Item 返回一个字符串，matcher 属性提供一个 Map 对象，根据 router 返回的字符串作为 key，从 Map 中获取 ItemWriter 对象。

接下来我们基于接口 Classifier 实现 router 的对象 CreditBillRouterClassifier，根据信用卡账单对象 CreditBill 的金额决定返回的值，如果信用卡账单对象大于 500，则返回"large"；如果信用卡账单没有超过 500，则返回"small"。

CreditBillRouterClassifier 实现参见代码清单 7-67。

完整代码参见：com.juxtapose.example.ch07.classifiercomposite.CreditBillRouterClassifier。

代码清单 7-67 CreditBillRouterClassifier 类定义

```
1. public class CreditBillRouterClassifier{  
2.     @Classifier  
3.     public String classify(CreditBill classifiable) {  
4.         if(classifiable.getAmount() > 500){  
5.             return "large";  
6.         }else{  
7.             return "small";  
8.         }  
9.     }  
10. }
```

接下来我们使用 BackToBackPatternClassifier 展示如何根据 CreditBill 对象来路由到不同的 ItemWrite 中去。具体配置代码参见代码清单 7-68。

配置 BackToBackPatternClassifier

代码清单 7-68 配置 BackToBackPatternClassifier

```
1.     <bean:bean id="backToBackClassifier"  
2.         class="org.springframework.classify.  
            BackToBackPatternClassifier" >  
3.         <bean:property name="routerDelegate"  
4.             ref="creditBillRouterClassifier" />  
5.         <bean:property name="matcherMap">  
6.             <bean:map>  
7.                 <bean:entry key="large" value-ref="flatFileItemWriterLarge"/>  
8.                 <bean:entry key="small" value-ref="flatFileItemWriterSmall"/>  
9.             </bean:map>
```

```

9.           </bean:property>
10.      </bean:bean>
11.      <bean:bean id="creditBillRouterClassifier"
12.          class="com.juxtapose.example.ch07.classifiercomposite.
             CreditBillRouterClassifier">
13.      </bean:bean>

```

其中，3 行：属性 routerDelegate 定义路由代理类，负责根据给定的 Item 对象返回路由规则，本处返回 large 或者 small。

4~9 行：属性 matcherMap 用于声明 Map 对象，key 通常用来定义路由规则，它是属性 routerDelegate 的返回值；value 通常定义 ItemWriter 的实例；通过 routerDelegate 与 matcherMap 的配合，可以方便地将 Item 路由到不同的 ItemWrite 进行处理。

ClassifierCompositeItemWriter 关键属性参见表 7-17。

表 7-17 ClassifierCompositeItemWriter 关键属性

ClassifierCompositeItemWriter 属性	类 型	说 明
classifier	Classifier<T, ItemWriter<? super T>>	根据给定的 Item 对象，选择不同的 ItemWrite 进行处理

配置 ClassifierCompositeItemWriter

接下来我们展示使用 ClassifierCompositeItemWriter 将不同的信用卡账单记录存入到不同的文件中。具体配置参见代码清单 7-69。

配置 ClassifierCompositeItemWriter

完整配置文件参见：ch07/job/job-composite-classify.xml。

代码清单 7-69 配置 ClassifierCompositeItemWriter

```

1.      <bean:bean id="classifierCompositeWriter"
2.          class="org.springframework.batch.item.support.
             ClassifierCompositeItemWriter">
3.          <bean:property name="classifier" ref="backToBackClassifier">
4.          </bean:property>
5.      </bean:bean>
6.      <bean:bean id="backToBackClassifier"
7.          class="org.springframework.classify.BackToBackPatternClassifier">
8.          <bean:property name="routerDelegate"
9.              ref="creditBillRouterClassifier" />
10.         <bean:property name="matcherMap">
11.             <bean:map>
12.                 <bean:entry key="large" value-ref="flatFileItemWriterLarge"/>
13.                 <bean:entry key="small" value-ref="flatFileItemWriterSmall"/>

```

```
13.          </bean:map>
14.      </bean:property>
15.  </bean:bean>
```

其中,3行:属性 classifier 用于定义路由规则的实现,将 Item 对象路由到不同的 ItemWrite 中处理。

11行:属性 flatFileItemWriterLarge 将 Item 写入到文件 file:target/ch07/composite/classify/outputFile_Large.csv。

12行:属性 flatFileItemWriter2 将 Item 写入到文件 file:target/ch07/composite/classify/outputFile_Small.csv。

使用代码清单 7-70 执行定义的 classifierCompositeWriteJob。

完整代码参见: test.com.juxtapose.example.ch07.JobLaunchClassifierComposite。

代码清单 7-70 执行 classifierCompositeWriteJob

```
1. JobLaunchBase.executeJob("ch07/job/job-composite-classify.xml",
2. "classifierCompositeWriteJob", new JobParametersBuilder().addDate
   ("date", new Date()));
```

文件 outputFile_Large.csv 写入信用卡账单记录如下(金额全部大于 500), 具体参见代码清单 7-71。

代码清单 7-71 文件 outputFile_Large.csv 内容清单

```
1. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road
2. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road
3. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road
```

文件 outputFile_Small.csv 写入信用卡账单记录如下(金额全部小于 500), 具体参见代码清单 7-72。

代码清单 7-72 文件 outputFile_Small.csv 内容清单

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

7.9 发送邮件

Spring Batch 框架提供了发送邮件的功能, 使用 org.springframework.batch.item.mail.SimpleMailMessageItemWriter 可以轻松地完成邮件发送的功能。

7.9.1 SimpleMailMessageItemWriter

本节我们使用的信用卡账单需要每期都发送到用户指定的邮箱中, 我们使用 SimpleMailMessageItemWriter 展示如何发送邮件功能。

SimpleMailMessageItemWriter 结构关键属性

图 7-23 展示了发送邮件的逻辑架构图，ItemProcessor 负责将 Item 对象转换为 SimpleMailMessage 消息；MailSender 将生成的消息发送到具体的邮箱中。

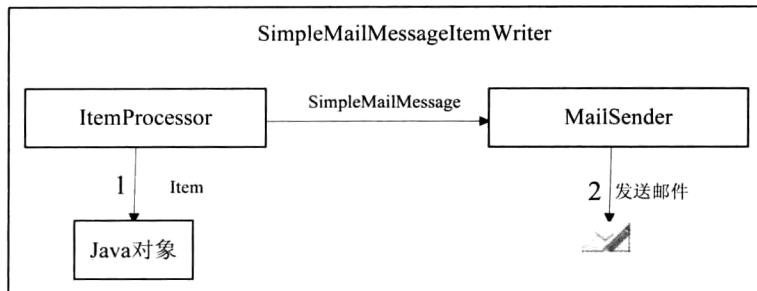


图 7-23 发送邮件的逻辑架构图

SimpleMailMessageItemWriter 关键属性参见表 7-17。

表 7-18 SimpleMailMessageItemWriter 关键属性

SimpleMailMessageItemWriter 属性	类 型	说 明
mailSender	MailSender	邮件发送工具类，负责具体的邮件发送
mailErrorHandler	MailErrorHandler	邮件发送失败情况下的消息处理类。 默认值： org.springframework.batch.item.mail.DefaultMailErrorHandler

通常情况下只需要设置属性 mailSender 即可。需要注意 SimpleMailMessageItemWriter 中 write 操作处理的参数类型为 SimpleMailMessage，因此在调用 SimpleMailMessageItemWriter 之前需要将数据准备正确。

配置 SimpleMailMessageItemWriter

本例中将从 Flat 文件中读取的账单信息，发送到指定的信箱中。首先准备 SimpleMailMessage，我们新实现一个 ItemProcessor，实现类为 MailItemProcessor，实现代码参见代码清单 7-73，用于处理从 Flat 文件中读取的 CreditBill 信息，转换为邮件发送需要的 SimpleMailMessage 对象。

实现 MailItemProcessor

完整代码参见：com.juxtapose.example.ch07.mail.MailItemProcessor。

代码清单 7-73 MailItemProcessor 类定义

```
1. public class MailItemProcessor implements ItemProcessor<CreditBill,  
    SimpleMailMessage> {  
2.     @Override
```

```

3.     public SimpleMailMessage process(CreditBill item) throws Exception {
4.         SimpleMailMessage msg = new SimpleMailMessage();
5.         msg.setFrom("springbatchexample@163.com");
6.         msg.setTo("springbatchexample@163.com");
7.         msg.setSubject("Credit detail " +
8.             new SimpleDateFormat("yyyy 年 MM 月 dd 日 hh 时 mm 分 ss 秒"));
9.             format(Calendar.getInstance().getTime()));
10.        msg.setText("Credit details: " + item.toString());
11.        return msg;
12.    }
13. }

```

其中，3~12 行：处理传入的信用卡账单对象，生成需要发送的邮件信息 SimpleMailMessage。

5 行：指定邮件的发件人为“springbatchexample@163.com”。

6 行：指定邮件的收件人为“springbatchexample@163.com”。

7~9 行：指定邮件的主题信息。

10 行：定义邮件的正文。

配置 SimpleMailMessageItemWriter

配置代码参见代码清单 7-74。

完整配置文件参见：ch07/job/job-java-mail.xml。

代码清单 7-74 配置 SimpleMailMessageItemWriter

```

1.   <context:property-placeholder
2.       location="classpath:/ch07/properties/batch-mail.properties"
3.       ignore-unresolvable="true"/>
4.
5.   <bean:bean id="mailItemWriter"
6.       class="org.springframework.batch.item.mail.
7.           SimpleMailMessageItemWriter">
8.       <bean:property name="mailSender" ref="javaMailSender" />
9.   </bean:bean>
10.  <bean:bean id="javaMailSender"
11.      class="org.springframework.mail.javamail.JavaMailSenderImpl">
12.      <bean:property name="host" value="${mail.smtp.host}" />
13.      <bean:property name="username" value="${mail.smtp.username}" />
14.      <bean:property name="password" value="${mail.smtp.password}" />
15.      <bean:property name="javaMailProperties">
16.          <bean:props>
17.              <bean:prop key="mail.smtp.auth">${mail.smtp.auth}</bean:prop>

```

```

18.          <bean:prop key="mail.smtp.timeout">${mail.smtp.timeout}
19.          </bean:prop>
20.      </bean:props>
21.  </bean:property>

```

其中，1~3 行：指定发送邮件需要的属性配置文件，该文件定义了后续需要用到的变量，包括 12、13、14、17、18 行中的 \${} 中的内容。

4~8 行：定义邮件发送类的实现，只需要定义属性 mailSender。

10~21 行：定义邮件发送的具体实现类 org.springframework.mail.javamail.JavaMailSenderImpl。

12 行：属性 host 定义发送邮件的主机地址，本例使用 163 邮箱，设置为 "smtp.163.com"。

13 行：属性 username 声明发件人信息，账户为 "springbatchexample"。

14 行：属性 password 声明发件人的密码，密码为 "springbatch"。

17 行：属性 mail.smtp.auth 定义是否采用安全策略，true 表示需要输入用户名口令信息。

18 行：属性 mail.smtp.timeout 定义超时时间，超时时间为 25000 毫秒。

batch-mail.properties 的详细信息参见代码清单 7-75。

代码清单 7-75 batch-mail.properties 的详细信息

```

1. mail.smtp.host=smtp.163.com
2. mail.smtp.username=springbatchexample
3. mail.smtp.password=springbatch
4. mail.smtp.auth=true
5. mail.smtp.timeout=25000

```

使用代码清单 7-76 执行定义的 javaMailJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchJavaMail。

代码清单 7-76 执行 javaMailJob

```

1. JobLaunchBase.executeJob("ch07/job/job-java-mail.xml", "javaMailJob",
2.     new JobParametersBuilder().addDate("date", new Date()));

```

执行成功后，读者可以登录到 163 的邮箱（账户：springbatchexample，密码：springbatch），可以到成功发送的邮件信息如图 7-24。



图 7-24 发送邮件的截图

7.10 服务复用

复用现有的企业资产和服务是提高企业应用开发的快捷手段，Spring Batch 框架的写组件提供了复用现有服务的能力，利用 Spring Batch 框架提供的 ItemWriterAdapter、PropertyExtractingDelegatingItemWriter 可以方便地复用业务服务、Spring Bean、EJB 或者其他远程服务。ItemWriterAdapter 代理的现有服务需要能够处理 Item 对象；PropertyExtractingDelegatingItemWriter 代理的服务支持更复杂的参数，参数可以根据指定的属性从 Item 中抽取，接下来我们介绍如何使用已经存在的服务。

7.10.1 ItemWriterAdapter

ItemWriterAdapter 结构关键属性

ItemWriterAdapter 持有服务对象，并调用指定的操作来完成 ItemWriter 中定义的 write 功能。需要注意的是：已经存在的服务需要能够直接处理 Item 对象，即参数必须是 Item 的具体类型。如果是更复杂的参数可以使用 PropertyExtractingDelegatingItemWriter 或者自己实现新的 ItemWriter 来完成从 Item 对象到现有服务参数的转变功能。

ItemWriterAdapter 和现有服务之间的关系参见图 7-25。

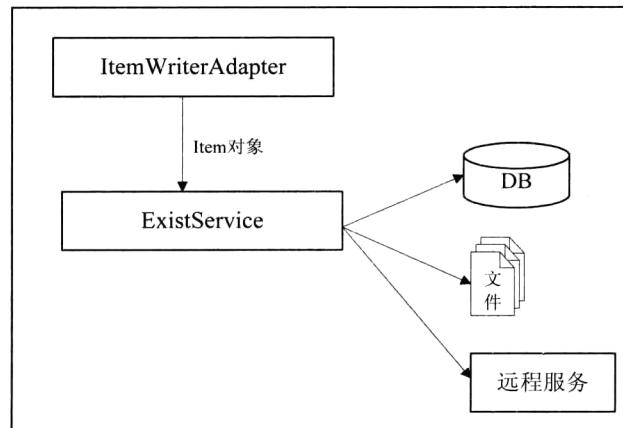


图 7-25 ItemWriterAdapter 和现有服务之间的关系

ItemWriterAdapter 类关系图参见图 7-26。

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

ItemWriterAdapter 关键属性

表 7-19 给出了 ItemWriterAdapter 的关键属性列表。在配置 ItemWriterAdapter 时候只需

要指定上面的前两个属性即可，参数 arguments 默认情况下将每次处理的 Item 对象作为参数传入。

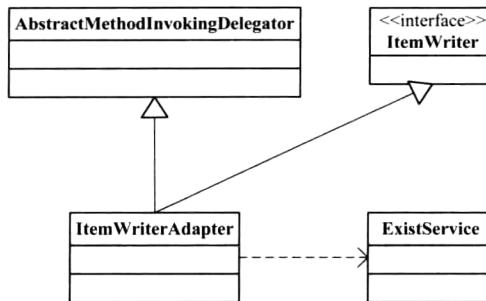


图 7-26 ItemWriterAdapter 类关系图

表 7-19 ItemWriterAdapter 关键属性

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

配置 ItemWriterAdapter

目前已经有服务（com.juxtapose.example.ch07.reuse.ExistService）可以将信用卡账单信息持久化，接下来我们使用 ExistService 作为示例来演示如何使用 ItemWriterAdapter。

已经存在的服务 ExistService 的示例代码，参见代码清单 7-77。

完整代码参见：com.juxtapose.example.ch07.reuse.ExistService。

代码清单 7-77 已存在服务 ExistService 示例代码

```
1. public class ExistService {
2.     List<CreditBill> billList = new ArrayList<CreditBill>();
3.
4.     public void insert(CreditBill creditBill) {
5.         billList.add(creditBill);
6.         System.out.println("ExistService insert:" + creditBill.toString());
7.     }
8.
9.     public void insert(String accountID, String name, double amount,
10.                      String date, String address) {
11.         CreditBill creditBill = new CreditBill(accountID, name, amount, date,
12.                                              address);
13.     }
14. }
```

```
12.         billList.add(creditBill);
13.         System.out.println("ExistService insert:" + creditBill.toString());
14.     }
15. }
```

本服务模拟已经存在的服务，共有两个不同的 insert 操作，4~7 行使用 CreditBill 作为参数，9~14 行使用多个基本类型作为参数；本例中将介绍如何使用 CreditBill 作为参数的 insert 操作。

接下来我们学习如何配置 ItemWriterAdapter。配置代码参见代码清单 7-78。

完整配置参见文件：ch07/job/job-reuse-service.xml。

代码清单 7-78 配置 ItemWriterAdapter

```
1. <bean:bean id="reuseServiceWriter"
2.   class="org.springframework.batch.item.adapter.
   ItemWriterAdapter">
3.   <bean:property name="targetObject" ref="existService"/>
4.   <bean:property name="targetMethod" value="insert"/>
5. </bean:bean>
6. <bean:bean id="existService"
7.   class="com.juxtapose.example.ch07.reuse.ExistService"/>
```

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

6~7 行：声明现存的服务。

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

使用代码清单 7-79 执行定义的 reuseServiceJob。

完整代码参见：test.com.juxtapose.example.ch07.JobLaunchReuseServiceWrite。

代码清单 7-79 执行 reuseServiceJob

```
1. executeJob("ch07/job/job-reuse-service.xml", "reuseServiceJob",
2.             new JobParametersBuilder().addDate("date", new Date()));
```

7.10.2 PropertyExtractingDelegatingItemWriter

PropertyExtractingDelegatingItemWriter 代理的服务支持更复杂的参数，参数可以根据指定的属性值从 Item 中抽取（ItemWriterAdapter 仅支持参数类型为具体的 Item 对象）。我们仍然使用上节中提供的服务，这次使用多个基本类型作为参数的 insert 操作。

PropertyExtractingDelegatingItemWriter 结构关键属性

PropertyExtractingDelegatingItemWriter 持有服务对象，并调用指定的操作来完成 ItemWriter 中定义的 write 功能。需要注意的是：使用该代理类支持已经存在的服务是复杂类

型的参数，使用接口 BeanWrapper 将给定的 Item 对象进行值的抽取，抽取的值作为存在服务的参数。

PropertyExtractingDelegatingItemWriter 和现有服务之间的关系参见图 7-27。

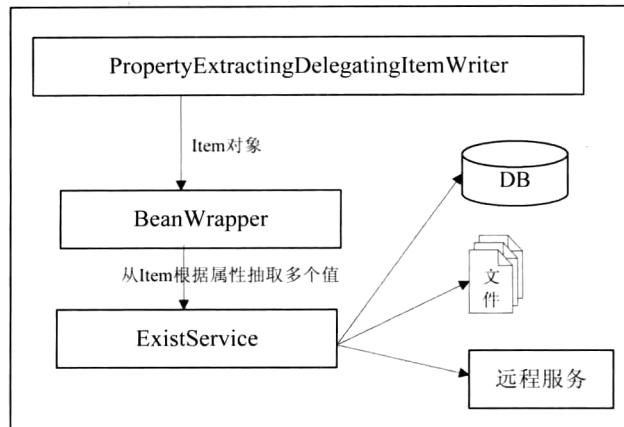


图 7-27 PropertyExtractingDelegatingItemWriter 和现有服务之间的关系

PropertyExtractingDelegatingItemWriter 类关系图参见图 7-28。

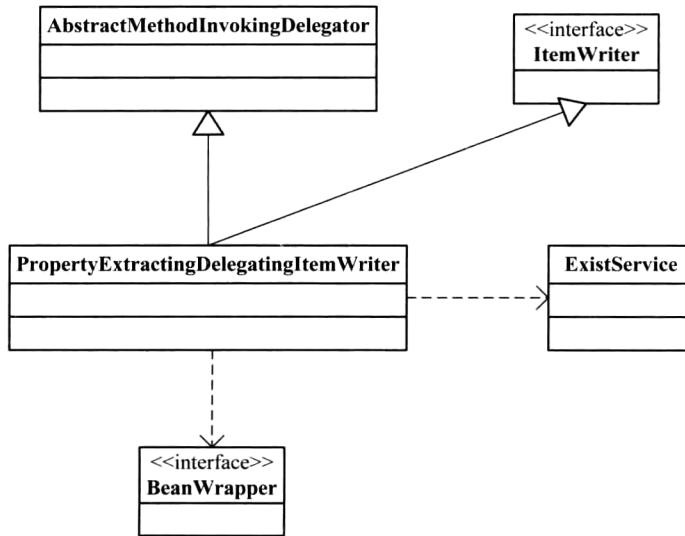


图 7-28 PropertyExtractingDelegatingItemWriter 类关系图

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

BeanWrapper 负责将给定的 Item 对象进行值的抽取，抽取的值作为 ExistService 的参数。PropertyExtractingDelegatingItemWriter 关键属性参见表 7-20。

表 7-20 PropertyExtractingDelegatingItemWriter 关键属性

PropertyExtractingDelegatingItemWriter 属性	类 型	说 明
targetObject	Object	需要调用的目标服务对象
targetMethod	String	需要调用的目标操作名称
fieldsUsedAsTargetMethodArguments	String[]	指定的 Item 中的属性名列表，根据这些指定的属性名从 Item 中抽取属性值，这些属性值被当做调用目标方法的参数值

配置 PropertyExtractingDelegatingItemWriter

接下来我们学习如何配置 PropertyExtractingDelegatingItemWriter。具体配置参见代码清单 7-80。

完整配置参见文件：ch07/job/job-reuse-service.xml。

代码清单 7-80 配置 PropertyExtractingDelegatingItemWriter

```
1.      <bean:bean id="reuseServicePropertyExtractWriter"
2.          class="org.springframework.batch.item.adapter.
3.                  PropertyExtractingDelegatingItemWriter">
4.          <bean:property name="targetObject" ref="existService"/>
5.          <bean:property name="targetMethod" value="insert"/>
6.          <bean:property name="fieldsUsedAsTargetMethodArguments">
7.              <bean:list>
8.                  <bean:value>accountID</bean:value>
9.                  <bean:value>name</bean:value>
10.                 <bean:value>amount</bean:value>
11.                 <bean:value>date</bean:value>
12.                 <bean:value>address</bean:value>
13.             </bean:list>
14.         </bean:property>
15.     </bean:bean>
16.     <bean:bean id="existService"
17.         class="com.juxtapose.example.ch07.reuse.ExistService"/>
```

其中，1~15 行：复用现有的服务完成 ItemWriterAdapter 的配置，属性 targetObject 使用定义的已存服务 existService；属性 targetMethod 指定调用 insert 操作（参数为多个基本类型的）。

6 行：定义需要从 Item 中抽取的属性，将抽取出来的属性值对应到目标操作的参数上。

16~17 行：声明现存的服务。

7.11 自定义 ItemWrite

Spring Batch 框架提供了丰富的 ItemWriter 组件，当这些默认的系统组件不能满足需求时，我们可以自己实现 ItemWriter 接口来完成需要的业务操作。自定义实现 ItemWriter 非常容易，只需要实现接口 ItemWriter；通常只实现接口 ItemWriter 的写不支持重启，为了支持可重启的自定义 ItemWriter 需要新增实现接口 ItemStream。接下来我们展示如何实现自定义的 ItemWriter。

7.11.1 不可重启 ItemWriter

接口 ItemWriter 的定义，参见代码清单 7-81。

代码清单 7-81 ItemWriter 接口定义

```
1. public interface ItemWriter<T> {  
2.     void write(List<? extends T> items) throws Exception;  
3. }
```

write 操作批量接受 Item 对象，一次将所有给定的 Item 持久化到给定的资源中，每次接受的 items 的大小为在 Chunk 中定义的提交间隔（commit-interval）的值。下面的示例展示了自定义 CustomCreditBillItemReader 的实现。

CustomCreditBillItemWriter 的实现参见代码清单 7-82。

完整代码参见：com.juxtapose.example.ch07.cust.itemwriter.CustomCreditBillItemWriter。

代码清单 7-82 CustomCreditBillItemWriter 类定义

```
1. public class CustomCreditBillItemWriter implements ItemWriter<CreditBill> {  
2.     private List<CreditBill> result = TransactionAwareProxyFactory.  
3.         createTransactionalList();  
4.     public void write(List<? extends CreditBill> items) throws Exception {  
5.         for(CreditBill item : items){  
6.             result.add(item);  
7.         }  
8.     }  
9.     .....  
10. }
```

其中，4~7 行：write 操作将传入的 Item 列表持久化到模拟的事务数组中。

配置自定义的 ItemWriter 非常简单，只需要简单地声明 bean 就可以了。

配置自定义 ItemWriter 的声明，参见代码清单 7-83。

完整 Job 配置参见文件：ch07/job/job-custom-itemwriter.xml。

代码清单 7-83 配置自定义的 ItemWriter

```
1. <bean:bean id="customItemWriter"  
2.             class="com.juxtapose.example.ch07.cust.itemwriter.
```

```
        CustomCreditBillItemWriter">
3.     </bean:bean>
```

其中，1~3 行：声明自定义的 ItemWriter。

接下来运行自定义 ItemWriter，执行代码参见代码清单 7-84。

完整代码参见类：test.com.juxtapose.example.ch07.JobLaunchCustomItemWriterTest。

代码清单 7-84 执行自定义 ItemWriter

```
1. List<CreditBill> list = new ArrayList<CreditBill>();
2. list.add(new CreditBill("1","tom",100.00,"2013-2-2 12:00:08","Lu Jia Zui
   road"));
3. list.add(new CreditBill("2","tom",320,"2013-2-3 10:35:21","Lu Jia Zui
   road"));
4. list.add(new CreditBill("3","tom",360.00,"2013-2-11 11:12:38","Longyang
   road"));
5. CustomCreditBillItemWriter writer = new CustomCreditBillItemWriter();
6. writer.write(list);
7. Assert.assertEquals(3, writer.getResult().size());
```

其中，1~4 行：准备示例数据，完成 list 的初始化对象。

5 行：完成 CustomCreditBillItemWriter 的初始化。

6 行：执行 write 操作，将 Items 列表持久化。

7 行：断言最终写入的记录条数为 3 条。

本节实现了自定义的 ItemWriter，但本节实现的自定义的 ItemWriter 不支持重新启动的特性，导致 Job 作业失败的情况下不能从失败的执行点重新写入。接下来的章节我们将实现可重启的自定义 ItemWriter。

7.11.2 可重启 ItemWriter

Spring Batch 框架对 Job 提供了可重启的能力，Spring Batch 框架提供的写组件 FlatFileItemWriter、MultiResourceItemWriter、StaxEventItemWriter 均实现了 ItemStream 接口。

和 ItemReader 不一样的是，ItemReader 的系统读组件基本都实现了 ItemStream 接口；而 ItemWriter 仅有部分的系统组件实现了 ItemStream 接口。因为通常情况下如果写的资源本身是事务性操作的，比如数据库的写，对 JMS 消息的写等，如果发生错误之后，因为在事务上下文中会导致整个批量写入都不成功，下次 Job 重启的时候，会从失败点继续写入。因此本身具有事务性的写操作不需要实现 ItemStream 就支持可重启的特性，例如 JdbcBatchItemWriter、HibernateItemWriter 等。

如果写操作本身是有状态的，为了支持可重启的特性必须实现 ItemStream，例如 FlatFileItemWriter、StaxEventItemWriter，写入的文件不具有事务的特性。接下来我们学习如何实现可重启的自定义的 ItemWriter 实现。

ItemStream 接口定义参见代码清单 7-85。

代码清单 7-85 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 打开需要写入的资源，可以根据持久化在执行上下文 executionContext 中的坐标信息重新定位需要写入记录的位置。

3 行：update()操作将需要持久化的数据存放在执行上下文 executionContext 中，通常将当前的状态数据保存在上下文中。

4 行：close()操作关闭写入的资源。

ItemStream 接口定义了写操作与执行上下文 ExecutionContext 交互的能力。可以将已经写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，写操作可以跳过已经成功写的数据，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写的位置）开始写。

接下来我们改造上节的自定义的 ItemWriter，新增实现接口 ItemStream，使其支持可重启的能力。RestartableCustomCreditBillItemWriter 的实现参见代码清单 7-86。

完整代码 参见：com.juxtapose.example.ch07.cust.itemwriter.RestartableCustomCreditBillItemWriter。

代码清单 7-86 RestartableCustomCreditBillItemWriter 类定义

```
1. public class RestartableCustomCreditBillItemWriter implements  
2.             ItemWriter<CreditBill>, ItemStream {  
3.     private List<CreditBill> result = new ArrayList<CreditBill>();  
4.     private int currentLocation = 0;  
5.     private static final String CURRENT_LOCATION = "current.location";  
6.  
7.     public void write(List<? extends CreditBill> items) throws Exception {  
8.         for(;currentLocation < items.size();){  
9.             result.add(items.get(currentLocation++));  
10.        }  
11.    }  
12.  
13.    public void open(ExecutionContext executionContext)  
14.        throws ItemStreamException {  
15.        if(executionContext.containsKey(CURRENT_LOCATION)){  
16.            currentLocation = new Long(executionContext.  
17.                getLong(CURRENT_LOCATION)).intValue();  
18.        }  
19.    }  
20.    public void update(ExecutionContext executionContext)  
21.        throws ItemStreamException {  
22.        if(executionContext.containsKey(CURRENT_LOCATION)){  
23.            currentLocation = new Long(executionContext.  
24.                getLong(CURRENT_LOCATION)).intValue();  
25.        }  
26.    }  
27.    public void close() throws ItemStreamException {  
28.        //  
29.    }  
30. }
```

```

18.     }else{
19.         currentLocation = 0;
20.     }
21. }
22.
23. public void update(ExecutionContext executionContext)
24.     throws ItemStreamException {
25.     executionContext.putLong(CURRENT_LOCATION,
26.                             new Long(currentLocation).longValue());
27. }
28.
29. public void close() throws ItemStreamException {}
30. }

```

其中，7~11 行： write 操作，根据当前的位置 currentLocation 从 list 中读取数据并写入持久化资源中，当前的位置标识在执行上下文中获取，具体参见 open 操作中的代码实现。

13~21 行： open 操作，从执行上下文中获取当前写入的位置。

23~27 行： update 操作，将已经写过的数据位置存放在执行上下文中，通常 update 操作在 chunk 的事务提交后会执行一次。

29 行： close 操作，通常在此处关闭不再需要的资源。

配置自定义的可重启 ItemWriter 非常简单，只需要简单的声明 bean 就可以。

配置自定义 ItemWriter 的声明，参见代码清单 7-87。

完整 Job 配置参见文件： ch07/job/job-custom-itemwriter.xml。

代码清单 7-87 配置自定义 ItemWriter

```

1. <bean:bean id="restartableCustomItemWriter"
2.             class="com.juxtapose.example.ch07.cust.itemwriter.
3.                     RestartableCustomCreditBillItemWriter">
4. </bean:bean>

```

其中，1~4 行： 声明自定义的可重启的 ItemWriter。

接下来运行 RestartableCustomCreditBillItemWriter，执行代码参见代码清单 7-88。

完整代码参见类： test.com.juxtapose.example.ch07.JobLaunchCustomItemWriterTest。

代码清单 7-88 执行自定义 ItemWriter

```

1. List<CreditBill> list = new ArrayList<CreditBill>();
2. list.add(new CreditBill("1","tom",100.00,"2013-2-2 12:00:08","Lu Jia Zui
   road"));
3. RestartableCustomCreditBillItemWriter writer = new RestartableCustom
   CreditBillItemWriter();
4. ExecutionContext executionContext = new ExecutionContext();
5. ((ItemStream)writer).open(executionContext);
6. writer.write(list);
7. Assert.assertEquals(1, writer.getResult().size());

```

```
8. ((ItemStream)writer).update(executionContext);
9.
10. list.add(new CreditBill("2","tom",320,"2013-2-3 10:35:21","Lu Jia Zui
    road"));
11. list.add(new CreditBill("3","tom",360.00,"2013-2-11 11:12:38","Longyang
    road"));
12. writer = new RestartableCustomCreditBillItemWriter();
13. ((ItemStream)writer).open(executionContext);
14. writer.write(list);
15. Assert.assertEquals(2, writer.getResult().size());
16. ((ItemStream)writer).update(executionContext);
```

其中，1~2 行：准备示例数据，完成 list 的初始化对象。

3 行：完成 RestartableCustomCreditBillItemWriter 的初始化。

4 行：模拟新建执行上下文对象 executionContext。

5 行：执行自定义 ItemWriter 的 open 操作，从执行上下文中获取当前已写入记录的位置。

6 行：执行写操作。

7 行：断言写入后只有一条记录。

8 行：执行自定义 ItemWriter 的 update 操作，将当前记录的位置存放在执行上下文中。

10~11 行：为 list 增加两条记录。

12 行：重新构造一个新的 RestartableCustomCreditBillItemWriter 对象，模拟重启该 Writer。

13 行：使用同一个执行上下文执行 open 操作。

14 行：再次执行写操作，这次会从上次执行的位置来写数据，因此本次写入的总记录数应该为 2 条。

15 行：断言本次写入的行数为 2 行。

本示例代码模拟了重新启动写操作的场景，其本质是运行时将游标的位置存放在执行上下文中，执行上下文中的数据在每次事务提交的时候会保存到数据库中；当作业 Job 重新启动的时候从执行上下文中重新获取上次写操作的位置，从正确的位置开始写操作，从而完成了支持重启的能力。

7.12 拦截器

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

7.12.1 拦截器接口

接口 ItemWriteListener 的定义参见代码清单 7-89。

代码清单 7-89 ItemWriteListener 接口定义

```
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. }
```

其中，2 行：beforeWrite 在写入资源前触发该操作。

3 行：afterWrite 在写入资源后触发该操作。

4 行：onWriteError 在写入资源发生异常的时候触发该操作。

为 ItemWriter 配置拦截器，具体配置参见代码清单 7-90。

完整配置参见文件：/ch07/job/job-listener.xml。

代码清单 7-90 配置 ItemWriter 拦截器

```
1. <job id="itemReadJob">  
2.     <step id="itemReadStep">  
3.         <tasklet transaction-manager="transactionManager">  
4.             <chunk reader="creditItemRead" processor="creditBillProcessor"  
5.                 writer="creditItemWriter" commit-interval="2">  
6.                 <listeners>  
7.                     <listener ref="sysoutItemWriteListener"></listener>  
8.                     <listener ref="sysoutAnnotationItemWriteListener">  
9.                     </listener>  
10.                </listeners>  
11.            </chunk>  
12.            <listeners>  
13.            </listeners>  
14.        </tasklet>  
15.    </step>  
16. </job>  
17. <bean:bean id="sysoutItemWriteListener"  
18.     class="com.juxtapose.example.ch07.listener.  
19.         SystemOutItemWriteListener">  
20.     </bean:bean>  
21. <bean:bean id="sysoutAnnotationItemWriteListener"  
22.     class="com.juxtapose.example.ch07.listener.SystemOutAnnotation">  
23.     </bean:bean>
```

其中，6~9 行：为作业的读配置 2 个拦截器。

16~18 行：定义拦截器 sysoutItemWriteListener，该拦截器实现接口 ItemWriteListener。

20~22 行：定义拦截器 sysoutAnnotationItemWriteListener，该拦截器通过 Annotation 方式定义。

SystemOutItemWriteListener 的代码参见代码清单 7-91。

类 SystemOutItemWriteListener 完整代码参见： com.juxtapose.example.ch07.listener.SystemOutItemWriteListener。

代码清单 7-91 SystemOutItemWriteListener 类定义

```
1. public class SystemOutItemWriteListener implements
   ItemWriteListener <CreditBill> {
2.     public void beforeWrite(List<? extends CreditBill> items) {
3.         System.out.println("SystemOutItemWriteListener.beforeWrite()");
4.     }
5.
6.     public void afterWrite(List<? extends CreditBill> items) {
7.         System.out.println("SystemOutItemWriteListener.afterWrite()");
8.     }
9.
10.    public void onWriteError(Exception exception,
11.        List<? extends CreditBill> items) {
12.        System.out.println("SystemOutItemWriteListener.onWriteError()");
13.    }
14. }
```

7.12.2 拦截器异常

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

如果拦截器发生异常，会导致 Job 执行的状态为"FAILED"。

配置了错误拦截器的作业配置参见代码清单 7-92。

完整配置参见文件：/ch07/job/job-listener.xml。

代码清单 7-92 配置了错误拦截器的作业

```
1. <job id="errorItemReadJob">
2.     <step id="errorItemReadStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="creditItemRead" processor="creditBillProcessor"
5.                 writer="creditItemWriter" commit-interval="2">
6.                 <listeners>
7.                     <listener ref="errorItemWriteListener"></listener>
8.                 </listeners>
9.             </chunk>
10.            </tasklet>
11.        </step>
12.    </job>
```

其中，7 行：errorItemWriteListener 在 beforeWrite 操作中抛出异常，会导致整个作业的

失败，当前的 Job 实例状态被标记为" FAILED "。

7.12.3 执行顺序

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

- (1) sysoutItemWriteListener 拦截器的 beforeWrite 方法；
- (2) sysoutAnnotationItemWriteListener 拦截器的 beforeWrite 方法；
- (3) sysoutAnnotationItemWriteListener 拦截器的 afterWrite 方法；
- (4) sysoutItemWriteListener 拦截器的 afterWrite 方法。

7.12.4 Annotation

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

- @BeforeWrite；
- @AfterWrite；
- @OnWriteError。

ItemWriteListener 操作说明与 Annotation 定义参见表 7-21。

表 7-21 ItemWriteListener 操作说明与 Annotation 定义

操作	操作说明	Annotation
beforeWrite(List<? extends S> items)	在 ItemWriter#write()之前执行	@ BeforeWrite
afterWrite(List<? extends S> items)	在 ItemWriter# write ()之后执行	@ AfterWrite
onWriteError(Exception exception, List<? extends S> items)	当 ItemWriter# write ()抛出异常时候触发该操作	@ OnWriteError

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

SystemOutAnnotation 的代码参见代码清单 7-93。

类 SystemOutAnnotation 完整代码参见：com.juxtapose.example.ch07.listener.SystemOutAnnotation。

代码清单 7-93 SystemOutAnnotation 类定义

```
1.  public class SystemOutAnnotation {  
2.      @BeforeWrite  
3.      public void beforeWrite(List<? extends CreditBill> items) {  
4.          System.out.println("SystemOutAnnotation.beforeWrite()");  
5.      }  
6.  }
```

```

5.      }
6.
7.      @AfterWrite
8.      public void afterWrite(List<? extends CreditBill> items) {
9.          System.out.println("SystemOutAnnotation.afterWrite()");
10.     }
11.
12.      @OnWriteError
13.      public void onWriteError(Exception exception,
14.          List<? extends CreditBill> items) {
15.          System.out.println("SystemOutAnnotation.onWriteError()");
16.      }
17.  }

```

7.12.5 属性 Merge

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

假设有这样一个场景，所有的 Step 都希望拦截器 sysoutItemWriteListener 能够执行，而拦截器 sysoutAnnotationItemWriteListener 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 7-94。

代码清单 7-94 merge 属性配置

```

1.  <job id="mergeChunkJob">
2.      <step id="subChunkStep" parent="abstractParentStep">
3.          <tasklet>
4.              <chunk reader="creditItemRead" processor="creditBillProcessor"
5.                  writer="creditItemWriter" >
6.                  <listeners merge="true">
7.                      <listener ref="sysoutAnnotationItemWriteListener">
8.                      </listener>
9.                  </listeners>
10.             </chunk>
11.         </tasklet>
12.     </step>
13.  </job>
14.  <step id="abstractParentStep" abstract="true">
15.      <tasklet>

```

```
16.          <chunk commit-interval="2" >
17.              <listeners>
18.                  <listener ref="sysoutItemWriteListener"></listener>
19.              </listeners>
20.          </chunk>
21.      </tasklet>
22.  </step>
```

其中，17~18 行：定义抽象作业步 abstractParentStep 中的拦截器。

6~8 行：通过 merge 属性，可以与父类中的拦截器配置进行合并，表示在 subChunkStep 中有两个拦截器会同时工作；属性 merge 的值为 true，表示父子合并即两处定义的都生效；实行 merge 的值为 false，表示父类的不再生效，被子类的定义覆盖掉了。

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

处理数据 ItemProcessor

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。批处理在读取数据后，写入数据之前，希望能够提供一个处理数据的阶段，ItemProcessor 是实现处理阶段的重要组件，Spring Batch 框架提供了丰富的处理组件，包括数据转换、组合处理、数据过滤、数据校验等能力，和前面的数据读取和写入一样，在处理数据阶段框架同样提供了复用现有业务逻辑的能力。

8.1 ItemProcessor

ItemProcessor 是 Step 中对资源的处理阶段，Spring Batch 框架已经提供了各种类型的处理实现，包括包括数据转换、组合处理、数据过滤、数据校验等。

处理操作与 Chunk Tasklet 的关系图参见图 8-1。

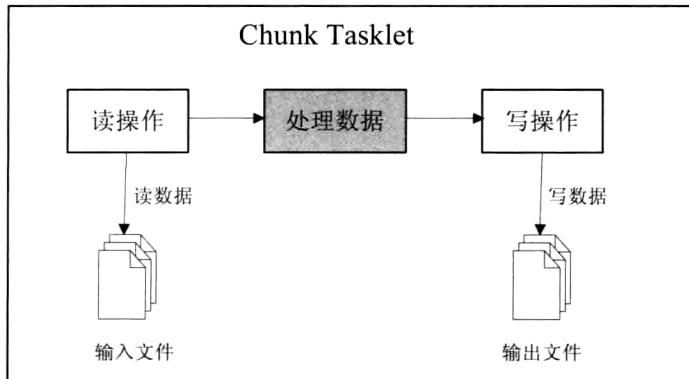


图 8-1 处理操作与 Chunk Tasklet 的关系图

需要注意的是数据处理阶段是可选的，也就是说可以只有读、写操作而没有中间的处理数据的阶段；这种情况下读的数据会直接交给写阶段来处理。

8.1.1 ItemProcessor

所有的处理操作都需要实现 `org.springframework.batch.item.ItemProcessor<I, O>` 接口。`ItemProcessor` 接口定义参见代码清单 8-1。

代码清单 8-1 ItemProcessor 接口定义

```
1. public interface ItemProcessor<I, O> {  
2.     O process(I item) throws Exception;  
3. }
```

其中，2 行：ItemProcessor 接口定义了核心作业方法 process () 操作，参数 I item 是读阶段获取的对象；返回值 O 会提供给写阶段，作为写阶段的输入参数。

读者可能会注意到 ItemWrite 接口定义的写操作的参数是 List<? extends T> items，Spring Batch 框架会将 ItemProcessor 阶段处理的数据收集起来，直到满足在 chunk 中定义的提交间隔（commit-interval）的大小才将处理的数据批量提交给 ItemWrite 进行处理。

读、处理、写操作三者间的数据转换关系图参见图 8-2。

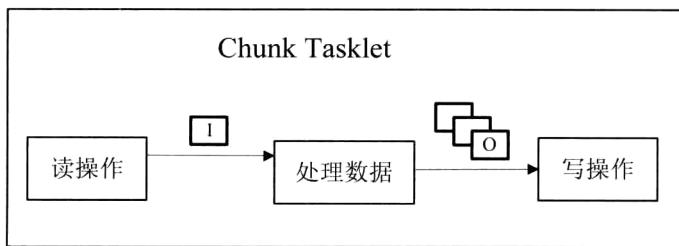


图 8-2 读、处理、写操作三者间的数据转换关系图

负责将 ItemReader 读入的数据进行处理后写入指定的资源中；注意这里 write 操作的参数是 List<? extends T> items 类型的，表示写操作通常会进行批量的写入；每次写入 List 的大小由属性 commit-interval 决定。

Job 中典型的配置 ItemProcessor 参见代码清单 8-2。

代码清单 8-2 典型的 ItemProcessor 配置

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

8.1.2 系统处理组件

Spring Batch 框架提供的处理组件列表参见表 8-1。

表 8-1 Spring Batch 框架提供的处理组件

ItemProcessor	说 明
CompositeItemProcessor	组合处理器，可以封装多个业务处理服务
ItemProcessorAdapter	ItemProcessor 适配器，可以复用现有的业务处理服务
PassThroughItemProcessor	不做任何业务处理，直接返回读到的数据
ValidatingItemProcessor	数据校验处理器，支持对数据的校验，如果校验不通过可以进行过滤掉或者通过 skip 的方式跳过对记录的处理

8.2 数据转换

ItemProcessor 的一个核心作用是对读阶段的数据进行转换，其中包括对部分数据进行更改，还可以根据读的数据完全返回一个不同类型的数据给处理阶段。

8.2.1 部分数据转换

部分数据转换效果图参见图 8-3。

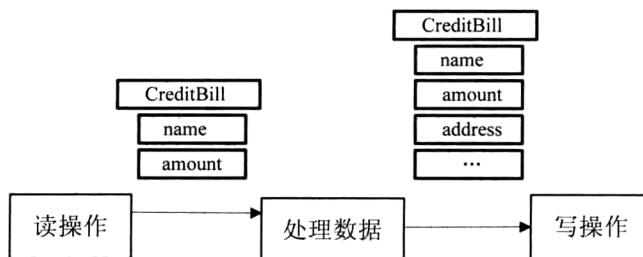


图 8-3 部分数据转换效果图

在部分转换的情况下，不会更改读入阶段的数据类型，可以针对读出的数据进行属性值的重新修订或者重新计算；在数据库的典型操作中甚至可以根据主键到数据库中进行查询，重新生成一个相同类型的数据对象。

需要处理的记录参见代码清单 8-3。

代码清单 8-3 需要处理的数据清单

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

接下来我们展示如何使用 ItemProcessor 进行数据转换操作，PartTranslateItemProcessor

实现 ItemProcessor 接口，负责将读入的对象 CreditBill 进行部分属性的变更。PartTranslateItemProcessor 的实现代码参见代码清单 8-4。

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

代码清单 8-4 PartTranslateItemProcessor 类定义

```
1. public class PartTranslateItemProcessor implements
2.     ItemProcessor<CreditBill, CreditBill> {
3.
4.     public CreditBill process(CreditBill bill) throws Exception {
5.         bill.setAddress(bill.getAddress() + "," + bill.getName());
6.         .....
7.         return bill;
8.     }
9. }
```

配置文件声明参见代码清单 8-5。

完整配置参见：ch08/job/job-translate.xml。

代码清单 8-5 配置 partTranslateJob

```
1. <job id="partTranslateJob">
2.     <step id="partTranslateStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="flatFileItemReader"
5.                 writer="part TranslateFlatFileItemWriter"
6.                 processor="partTranslateItemProcessor"
7.                 commit-interval="2">
8.             </chunk>
9.         </tasklet>
10.    </step>
11. </job>
12. <bean:bean id="partTranslateItemProcessor"
13.     class="com.juxtapose.example.ch08.PartTranslateItemProcessor">
14. </bean:bean>
```

其中，10~12 行：声明数据转换的定义。

经过部分数据处理后写入的文件内容参见代码清单 8-6。

代码清单 8-6 转换后的数据清单

```
1. 4047390012345678,tom,100.0,2013-2-2 12:00:08,Lu Jia Zui road,tom
2. 4047390012345678,tom,320.0,2013-2-3 10:35:21,Lu Jia Zui road,tom
3. 4047390012345678,tom,674.7,2013-2-6 16:26:49,South Linyi road,tom
4. 4047390012345678,tom,793.2,2013-2-9 15:15:37,Longyang road,tom
5. 4047390012345678,tom,360.0,2013-2-11 11:12:38,Longyang road,tom
6. 4047390012345678,tom,893.0,2013-2-28 20:34:19,Hunan road,tom
```

每行的记录最后多了"tom"。

8.2.2 数据类型转换

数据类型转换效果图参见图 8-4。

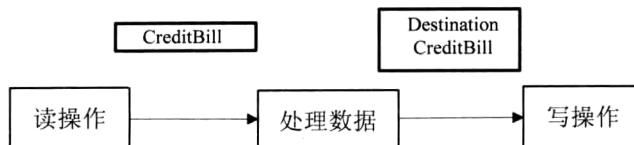


图 8-4 数据类型转换效果图

在数据类型变换的情况下，通常是读、写阶段需要处理的数据类型不一致，此时可以通过 ItemProcessor 进行数据的适配，例如图 8-4 中读阶段获取的对象为 CreditBill 类型，通过数据处理后返回给写操作的类型为 DestinationCreditBill 的对象。

接下来我们展示如何使用 ItemProcessor 进行数据转换操作，TranslateItemProcessor 实现 ItemProcessor 接口，负责将读入的对象 CreditBill 转换为写阶段需要处理的对象 Destination CreditBill。TranslateItemProcessor 类实现参见代码清单 8-7。

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

代码清单 8-7 TranslateItemProcessor 类定义

```
1. public class TranslateItemProcessor implements
2.     ItemProcessor<CreditBill, DestinationCreditBill> {
3.
4.     public DestinationCreditBill process(CreditBill bill) throws Exception {
5.         DestinationCreditBill destCreditBill = new DestinationCreditBill();
6.         destCreditBill.setAccountID(bill.getAccountID());
7.         destCreditBill.setAddress(bill.getAddress());
8.         ....
9.         return destCreditBill;
10.    }
11. }
```

配置文件声明参见代码清单 8-8。

完整配置参见：ch08/job/job-translate.xml。

代码清单 8-8 配置 translateJob

```
1. <job id="translateJob">
2.     <step id="translateStep">
3.         <tasklet transaction-manager="transactionManager">
4.             <chunk reader="flatFileItemReader"
5.                   writer="translateFlatFileItemWriter"
6.                   processor="translateItemProcessor"
7.                   commit-interval="2">
8.             </chunk>
```

```
7.           </tasklet>
8.       </step>
9.   </job>
10.  <bean:bean id="translateItemProcessor"
11.      class="com.juxtapose.example.ch08.TranslateItemProcessor">
12.  </bean:bean>
```

其中，10~12 行：声明数据转换的定义。

8.3 数据过滤

数据处理除了支持上节的数据转换功能外，同样对数据提供了过滤的能力。过滤是指如果对读入阶段的数据不期望在写入阶段被写入，可以通过返回 null 来阻止该 Item 数据被写入。

8.3.1 数据 Filter

数据过滤的功能参见图 8-5。



图 8-5 数据过滤示意图

仍然使用处理信用卡账单的例子，所有金额大于 500 的数据需要过滤掉，不能被提交到 write 阶段写入到文件中。接下来我们实现数据过滤的 FilterItemProcessor。

需要处理的记录参见代码清单 8-9。

代码清单 8-9 需要处理的数据清单

```
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
```

FilterItemProcessor 的实现参见代码清单 8-10。

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

代码清单 8-10 FilterItemProcessor 类定义

```
1. public class FilterItemProcessor implements ItemProcessor<CreditBill,
   CreditBill> {
2.
```

```
3.     @Override
4.     public CreditBill process(CreditBill item) throws Exception {
5.         if(item.getAmount() > 500) {
6.             return null;
7.         }else{
8.             return item;
9.         }
10.    }
11. }
```

配置文件声明参见代码清单 8-11。

完整配置参见：ch08/job/job-filter.xml。

代码清单 8-11 配置 filterJob

```
1.   <job id="filterJob">
2.     <step id="filterStep">
3.       <tasklet transaction-manager="transactionManager">
4.         <chunk reader="flatFileItemReader" writer="flatFileItemWriter"
5.               processor="filterItemProcessor" commit-interval="2">
6.           </chunk>
7.         </tasklet>
8.       </step>
9.     </job>
10.    <bean:bean id="filterItemProcessor"
11.      class="com.juxtapose.example.ch08.FilterItemProcessor">
12.    </bean:bean>
```

其中，1~9 行：定义 Job，使用数据过滤功能。

10~12 行：声明数据过滤的定义。

经过数据过滤后写入的文件内容参见代码清单 8-12。

代码清单 8-12 过滤后的数据清单

```
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,360.00,2013-2-11 11:12:38,Longyang road
```

数据过滤与异常跳过的区别

(1) 数据过滤是指 ItemProcessor 中的处理操作如果返回值是 Null，则该条数据被过滤掉，不会进入到 ItemWrite 阶段。

(2) 异常跳过是指在 Item 的处理阶段如果发生了指定类型的异常，则该条记录被忽略掉。

8.3.2 数据过滤统计

无论是数据过滤还是异常跳过处理，在 Job 执行期间都会把源信息存入到 JobRepository

中；可以通过作业步执行器获取被过滤的记录总数、异常跳过的记录总数等信息。

`StepExecution.getFilterCount()`可以获取被过滤的记录总数。

`StepExecution.getSkipCount()`可以获取异常跳过的记录总数。

接下来我们展示通过拦截器 `FilterCountStepExecutionListener`（实现作业步拦截器接口）获取被过滤的记录条数。

`FilterCountStepExecutionListener` 的实现参见代码清单 8-13。

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

代码清单 8-13 FilterCountStepExecutionListener 类定义

```
1. public class FilterCountStepExecutionListener extends  
2.     StepExecutionListenerSupport {  
3.     @Override  
4.     public ExitStatus afterStep(StepExecution stepExecution) {  
5.         int filterCount = stepExecution.getFilterCount();  
6.         System.out.println("Filter count=" + filterCount);  
7.         return stepExecution.getExitStatus();  
8.     }  
9. }
```

其中，4 行：通过作业步执行器 `stepExecution` 获取过滤的记录总数。

配置文件声明参见代码清单 8-14。

完整配置参见：`ch08/job/job-filter.xml`。

代码清单 8-14 配置 filterJob

```
1.     <job id="filterJob">  
2.         <step id="filterStep">  
3.             <tasklet transaction-manager="transactionManager">  
4.                 <chunk reader="flatFileItemReader" writer="flatFileItemWriter"  
5.                     processor="filterItemProcessor" commit-interval="2">  
6.                     <listeners>  
7.                         <listener ref="filterCountStepExecutionListener" />  
8.                     </listeners>  
9.                 </chunk>  
10.            </tasklet>  
11.        </step>  
12.    </job>  
13.    <bean:bean id="filterCountStepExecutionListener"  
14.        class="com.juxtapose.example.ch08.  
15.        FilterCountStepExecutionListener">  
16.    </bean:bean>
```

其中，1~12 行：定义 Job，使用数据过滤功能，同时增加了作业步执行器的拦截器。

13~15 行：声明数据过滤的拦截器，负责收集被过滤数据的总条数。

执行该 Job 后，统计到被过滤的数据信息参见代码清单 8-15。

代码清单 8-15 统计过滤后的数据清单

```
1. Filter count=3
```

该 Job 中所有被过滤的记录条目为 3 条。

8.4 数据校验

在业务数据处理过程中经常需要对输入的数据进行有效性校验，例如对于信用卡交易记录显然账号不能为空，需要还款的交易金额不能小于 0 等业务规则。Spring 框架提供了对数据校验的接口，如果校验不通过可以抛出给定类型的异常。

Spring Batch 框架提供了数据校验处理类 `ValidatingItemProcessor`，可以在处理的阶段进行数据校验，`ValidatingItemProcessor` 本身支持过滤的功能和跳过两种能力。

8.4.1 Validator

Spring 框架提供的校验接口为 `org.springframework.batch.item.validator.Validator<T>`，仅有一个操作 `validate`，对输入的参数进行数据校验，如果校验不通过可以抛出类型为 `ValidationException` 的异常。

接口 `Validator` 定义参见代码清单 8-16。

代码清单 8-16 Validator 接口定义

```
1. public interface Validator<T> {  
2.     void validate(T value) throws ValidationException;  
3. }
```

对于信用卡交易，在统计账单的时候仅需要统计消费的部分，而对于客户直接存入的交易部分则不需要统计（通常用负数表示）。因此针对信用卡消费单的数据作如下的校验，如果消费金额小于 0 则不进行数据的处理。

`CreditBillValidator` 的实现参见代码清单 8-17。

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

代码清单 8-17 CreditBillValidator 类定义

```
1. public class CreditBillValidator implements Validator<CreditBill> {  
2.  
3.     @Override  
4.     public void validate(CreditBill creditBill) throws ValidationException {  
5.         if(Double.compare(0, creditBill.getAmount()) > 0) {  
6.             throw new ValidationException("Credit bill cannot be negative!");  
7.         }  
8.     }  
9. }
```

如果消费金额小于 0，则抛出类型为 `ValidationException` 的异常。

代码清单 8-18 是我们本节示例用到的数据记录。

代码清单 8-18 待校验的数据记录清单

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

负数的记录行需要通过校验的方式跳过处理。

8.4.2 ValidatingItemProcessor

ValidatingItemProcessor 结构关键属性

ValidatingItemProcessor 实现接口 ItemProcessor，通过引用接口 Validator 进行数据校验功能，根据业务需要自定义实现符合业务需求的校验器。ValidatingItemProcessor 支持过滤的功能和跳过两种能力，通过属性 filter 进行标识，true 表示校验不通过的时候直接返回 null，跳出该条记录的处理；false 表示使用异常跳过的能力，可以通过配置 skipable-exception-classes 的方式忽略校验异常。ValidatingItemProcessor 核心类结构图参见图 8-6。

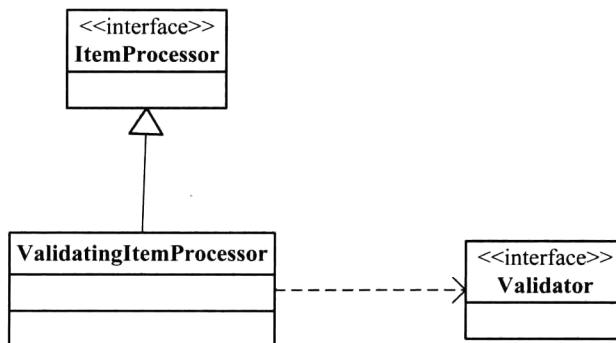


图 8-6 ValidatingItemProcessor 核心类结构图

ValidatingItemProcessor 关键属性参见表 8-2。

表 8-2 ValidatingItemProcessor 关键属性

ValidatingItemProcessor 属性	类 型	说 明
validator	Validator<? super T>	数据校验器，校验给定的 Item 数据是否合法
Filter	Boolean	是否使用过滤功能。 默认值： false

配置 ValidatingItemProcessor

首先定义 ValidatingItemProcessor，参见代码清单 8-19 中的代码。

完整配置文件参见：ch08/job/job-validate.xml。

代码清单 8-19 配置 ValidatingItemProcessor

```
1.      <bean:bean id="validatorProcessor" scope="step"
2.          class="org.springframework.batch.item.validator.
   ValidatingItemProcessor">
3.          <bean:property name="filter" value="#{jobParameters['filter']}
```

其中，3 行：属性 filter 用于声明是否采用过滤的功能，value 值采用了参数后绑定的技术，在执行 Job 的时候需要指定参数"filter"的值。

4~6 行：属性 validator 定义参数校验的具体实现类，使用上节定义的 com.juxtapose.example.ch08.CreditBillValidator。

Job 的配置参见代码清单 8-20。

完整配置文件参见：ch08/job/job-validate.xml。

代码清单 8-20 配置 validateJob

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

其中，6~9 行：skippable-exception-classes 定义处理阶段发生异常的会被忽略掉，ValidationException 类型的异常不会引起 Job 作业失败。

11 行：声明过滤统计作业步执行器，实现参见 8.3.2 章节。

12 行：声明跳过统计作业步执行器，skipCountStepExecutionListener 实现参见下面的

SkipCountStepExecutionListener。

SkipCountStepExecutionListener 的实现参见代码清单 8-21。

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

代码清单 8-21 SkipCountStepExecutionListener 类定义

```
1. public class SkipCountStepExecutionListener extends StepExecutionListener
2.     implements StepExecutionListenerSupport {
3.     @Override
4.     public ExitStatus afterStep(StepExecution stepExecution) {
5.         int skipCount = stepExecution.getSkipCount();
6.         System.out.println("Skip count=" + skipCount);
7.         return stepExecution.getExitStatus();
8.     }
}
```

其中，4 行：通过作业步执行器 stepExecution 获取跳过的记录总数。

执行 Job，经过数据校验后的文件内容参见代码清单 8-22。

代码清单 8-22 数据校验后的文件内容清单

```
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,360.00,2013-2-11 11:12:38,Longyang road
```

8.5 组合处理器

在 Spring Batch 框架中对于 Chunk 只能配置一个 ItemProcessor，但在有些业务场景中需要将一个 Item 同时执行多个不同处理器，例如首先进行 Item 的转换，然后再进行数据的校验工作；Spring Batch 框架提供了组合 ItemProcessor（CompositeItemProcessor）的模式满足上面的需求。组合 ItemProcessor 完成的功能参见图 8-7。

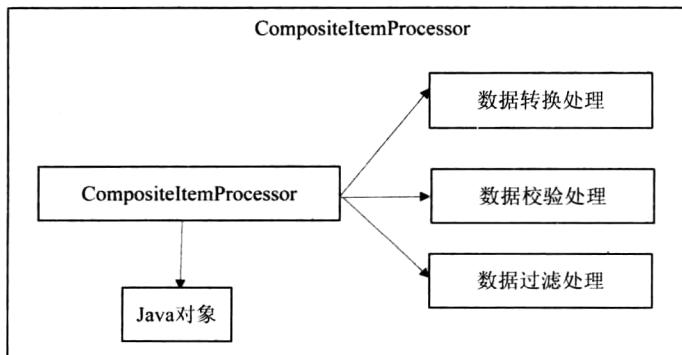


图 8-7 组合 ItemProcessor 完成的功能示意图

组合处理器代理不同的数据处理类，按照定义的顺序先后执行处理器，完成多个处理器的功能。

CompositeItemProcessor 结构关键属性

CompositeItemProcessor 组件实现 ItemProcessor 接口，并引用一组 ItemProcessor 实现类，CompositeItemProcessor 在进行处理阶段循环调用 ItemProcessor 中的实现，实现执行多个处理器的能力。CompositeItemProcessor 核心类结构图参见图 8-8。

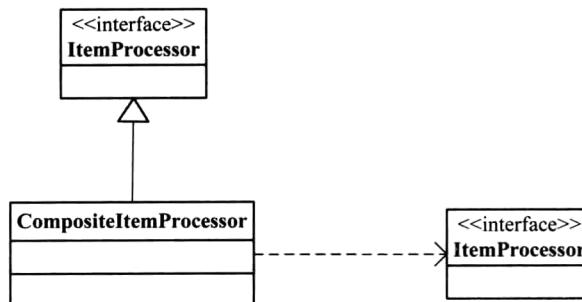


图 8-8 CompositeItemProcessor 核心类结构图

CompositeItemProcessor 关键属性参见表 8-3。

表 8-3 CompositeItemProcessor 关键属性

CompositeItemProcessor 属性	类 型	说 明
delegates	List<? extends ItemProcessor<?, ?>>	被代理的 ItemProcessor 的组合，对于每一个 Item 对象会经过每个处理器进行加工

配置 CompositeItemProcessor

接下来我们使用前面章节使用的部分数据转换和数据校验的功能，首先使用数据转换的功能，修改每条记录的地址信息；然后使用数据校验功能需要保证每条记录的消费金额大于 0。通过 CompositeItemProcessor 来实现上面的处理组合。

代码清单 8-23 是我们本节示例用到的数据记录。

代码清单 8-23 需要处理的数据清单

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

配置 CompositeItemProcessor，配置代码参见代码清单 8-24。