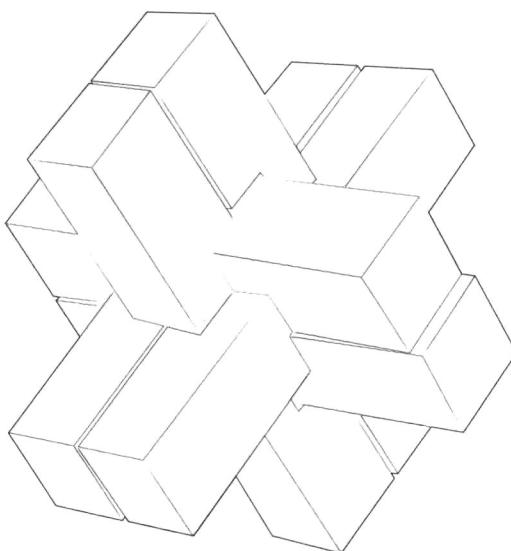


大数据时代数据批处理利器
首度原创解析Spring Batch框架

Broadview®
www.broadview.com.cn



Spring Batch 批处理框架

全
刘

深入的架构设计 底层的源码剖析

详细解读设计原理

刘相 编著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Spring Batch 批处理框架



内容概况

大数据时代，数据是现代企业最宝贵的核心资产，是企业运用科学管理、决策分析的基础。

企业如何通过各种技术手段，并把数据转换为信息、知识和商机已经成为提高其核心竞争力的主要手段。而数据批处理则是达成上述目标的一个主要的技术手段，通过数据批量处理，可以完成数据的加载、抽取、转换、清洗等功能，进而支撑企业的各种数据分析。Spring Batch批处理框架在金融、电信、大型制造业等使用非常广泛。

本书从基本的入门篇讲起，通过介绍批处理、Spring Batch基本特性、新特性，快速入门的Hello World等内容引领读者入门，进入批处理的世界。

之后的基本篇，重点讲述了批处理的核心概念、典型的作业配置、作业步配置，以及Spring Batch框架中经典的三步走策略：数据读，数据处理，数据写。

高级篇提供了高性能、高可靠性、并行处理的能力，分别向读者展示如何实现作业流的控制，包括顺序流、条件流、并行流；如何实现健壮的作业，包括跳过、重试、重启等；如何实现扩展作业及并行作业，包括多线程作业、并行作业、远程作业、分区作业等。

专家力荐

作为Spring认证培训师，本书是我读过的唯一一本关于Spring Batch的书，除了赞叹作者的与时俱进之外，也被本书之内容翔实，面面俱到所折服。

字里行间不难看出作者是经验丰富的实干派程序员。如果说缺点的话，唯一的一点就是遣词造句不如专业作者那样优雅。但本书作为一本技术手册已经非常棒了！

ThoughtWorks 架构师

李小波

这是第一本直接使用中文写作的Spring Batch图书，内容深入浅出，非常适合Java程序员理解批处理的编程模式。

微软中国 架构师

李博

这是一本关于Spring Batch的实战书籍，作者从具体的业务需求角度出发，逐渐展开到Spring Batch本身架构原理。通读本书，受益匪浅！

普元信息技术股份有限公司 软件产品部 总工程师

王葱权

图书配套代码下载链接：<https://github.com/jxtaliu/SpringBatchSample.git>



博文视点Broadview



@博文视点Broadview



责任编辑: 孙学瑛

责任编辑: 徐津平

封面设计: 李玲

Spring Batch

批处理框架

刘相 编著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书全面、系统地介绍了批处理框架 Spring Batch，通过详尽的实战示例向读者展示了 Spring Batch 框架对大数据批处理的基本开发能力，并对框架的架构设计、源码做了特定的剖析；在帮助读者掌握 Spring Batch 框架基本功能、高级功能的同时，深入剖析了 Spring Batch 框架的设计原理，帮助读者可以游刃有余地掌握 Spring Batch 框架。

本书分为入门篇、基本篇和高级篇三部分。入门篇介绍了批处理、Spring Batch 的基本特性和新特性，快速入门的 Hello World 等内容引领读者入门，从而进入数据批处理的世界。基本篇重点讲述了数据批处理的核心概念、典型的作业配置、作业步配置，以及 Spring Batch 框架中经典的三步走策略：数据读、数据处理和数据写，详尽地介绍了如何对 CSV 格式文件、JSON 格式文件、XML 文件、数据库和 JMS 消息队列中的数据进行读操作、处理和写操作，对于数据库的操作详细介绍了使用 JDBC、Hibernate、存储过程、JPA、Ibatis 等处理。高级篇提供了高性能、高可靠性、并行处理的能力，分别向读者展示了如何实现作业流的控制，包括顺序流、条件流、并行流，如何实现健壮的作业，包括跳过、重试和重启等，如何实现扩展作业及并行作业，包括多线程作业、并行作业、远程作业和分区作业等，从而实现分布式、高性能、高扩展性的数据批处理作业。

本书适合需要具体使用批处理作业、大数据处理的开发人员，设计人员和架构师，对于企业中存在大量作业的运维人员亦有一定的参考价值。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Spring Batch 批处理框架 / 刘相编著. —北京：电子工业出版社，2015.2

ISBN 978-7-121-25241-9

I . ①S… II . ①刘… III. ①数据处理 IV. ①TP274

中国版本图书馆 CIP 数据核字（2014）第 302744 号

策划编辑：孙学瑛

责任编辑：徐津平

特约编辑：顾慧芳

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：25.25 字数：582 千字

版 次：2015 年 2 月第 1 版

印 次：2015 年 2 月第 1 次印刷

印 数：3000 册 定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

批处理编程之美

这是一部论述批处理程序编程的书。在信息系统中，联机和批处理是计算机处理的两种基本模式，前者快速响应、超时中断、密集并发，后者处理时间长、异常需要支持重做、通常以顺序执行。早期的计算机系统多采用批处理的处理模式，客户机/服务器架构的产生让联机模式越来越多地被采用，但批处理模式一直在信息系统中起着重要作用，随着 OLAP、大数据等新技术的应用，批处理的处理模式又成为热点，例如在传统银行 IT 系统中每日动辄运行上万个批处理作业，在互联网应用中，腾讯、阿里每日的批处理作业可达百万量级的水平。

编程之美，美在架构，架构之美，美在抽象，只有具备充分理解复杂业务场景的格局，才能进行将复杂问题做简单化的抽象。同联机模式汗牛充栋的著作、框架相比，批处理模式的抽象不多，著名的 MapReduce 就是其中之一，MapReduce 将大批数据的处理过程进行了抽象，而 Spring Batch 则是对编写批处理程序本身的特性进行了抽象。通过将批处理程序分解为 Job 和 Job Step 两个部分，将处理环节定义为数据读、数据处理和数据写三个步骤，将异常处理机制归结为跳过、重试、重启三种类型，将作业方式区分为多线程、并行、远程、分区四大特征，正所谓增一分则肥，减一分则瘦。类似之美，Spring 系列项目还有很多，例如 Spring Framework 对依赖注入的抽象，Spring integration 中利用消息、队列、处理器三个概念的组合对集成模式的抽象，都让我叹为观止。

当相相^①递给我他的新作时，我吃了一惊，惊在他“悄无声息”地完成了这样一个大部头作品，迫不及待地有一种先睹为快的冲动。我发现，书中通过对 Spring Batch 本身的论述，让我体会到了 Spring Batch 的精髓，也更加深刻地理解了批处理编程模式，还看到了相相对信息系统中如何使用这一框架的见解，毕竟信息系统中的批处理程序，不仅仅是一个框架，还需要包含更多的管理、运维方面的流程、制度与平台。近年来，我看到很多企业都在构建集中的批处理平台，管理大量出现的批处理作业，也期待相相能有更多这方面的分享。

普元 CTO 焦烈焱

2015 年 1 月

① 相相为本书作者刘相的简称。

前 言

信息时代，数据是现代企业最宝贵的核心资产，是企业运用科学管理、决策分析的基础。截至目前，国内大多数企业已经完成了 OLTP（联机事务处理）的业务系统和办公自动化系统，用来记录事务处理的各种相关数据。据统计，企业的数据每年都在成倍增长，企业如能充分利用这些数据会带来巨大的商机。但目前企业通常所关注的数据仅占总数据量的 5% 左右，企业没有最大化地利用已经存在的数据资源，导致浪费了更多的时间和资金，同时也失去了制定关键商业决策的最佳契机。于是，企业如何通过各种技术手段，并把数据转换为信息、知识和商机已经成为提高其核心竞争力的主要手段。而数据批处理则是达成上述目标的一个主要技术手段，通过数据批量处理，可以完成数据的加载、抽取、转换、清洗等功能，进而支撑企业的各种数据分析。

2012 年底，笔者有幸接手某银行批处理项目，首次接触 Spring Batch 批处理框架，深入学习 Spring Batch 框架后发现：Spring Batch 框架的架构设计清晰和优雅，其功能完备，具有无所不在的扩展能力、丰富的业务组件，并且采用业务与技术分离的设计思想。笔者通过近半年多的学习，通读了官网提供的所有文档，深入学习了原版书籍 *Spring Batch In Action*。在大数据时代，批处理框架在金融、电信、大型制造业等应用非常广泛。在学习的过程中，笔者了解到同事、网络上有不少朋友苦于没有中文版的 Spring Batch 框架介绍资料，于是萌生出写一本 Spring Batch 框架中文版介绍图书的想法，希望能够帮助国内的读者快速地掌握该框架。

自 2013 年年中起笔者开始构思本书的大纲，在繁忙工作之余坚持每周完成部分章节内容，前后历时 1 年多时间完成书稿编写工作。在此感谢妻子 Phyllis 给予我充足的时间、感谢可爱女儿 Rachel 给我带来的巨大欢乐动力。本书主要的目的是全面、系统地介绍批处理框架 Spring Batch，通过详尽的实战示例向读者展示了 Spring Batch 框架的基本开发能力，并对框架的架构设计、源码做了特定的剖析；在帮助读者掌握该框架基本功能、高级功能使用的同时，深入剖析 Spring Batch 框架的设计原理，帮助读者可以游刃有余地掌握 Spring Batch 框架。

本书分为入门篇、基本篇、高级篇三个部分，从基本的入门篇讲起，通过介绍批处理、Spring Batch 基本特性、新特性、快速入门的 Hello World 等内容引领读者入门，进入批处理的世界。之后的基本篇，重点讲述了批处理的核心概念、典型的作业配置、作业步配置以及 Spring Batch 框架中经典的三步走策略：数据读、数据处理、数据写，详细介绍了如何对分隔符类型文件、定长类型文件、JSON 格式文件、复杂类型格式文件、XML 文件、数据库、JMS

消息队列中的数据进行读、处理、写操作，对于数据库的操作详细介绍了使用 JDBC、Hibernate、存储过程、JPA，Ibatis 等处理。为了能够让读者更深入地了解 Spring Batch 框架，高级篇提供了高性能、高可靠性、并行处理的能力，分别向读者展示如何实现作业流的控制包括顺序流、条件流、并行流，如何实现健壮的作业包括跳过、重试、重启等，如何实现扩展作业及并行作业包括多线程作业、并行作业、远程作业、分区作业等。

本书适合需要具体使用批处理框架 Spring Batch 的开发人员、设计人员、架构师，对于企业中存在大量作业的运维人员亦有一定的参考价值。

编 者

2014.12.23 于上海浦东

目 录

第 1 篇 入门篇

第 1 章 Spring Batch 简介	2
1.1 什么是批处理	2
1.2 Spring Batch	3
1.2.1 典型场景	3
1.2.2 Spring Batch 架构	4
1.3 Spring Batch 优势	4
1.3.1 丰富的开箱即用组件	5
1.3.2 面向 Chunk 的处理	5
1.3.3 事务管理能力	5
1.3.4 元数据管理	5
1.3.5 易监控的批处理应用	5
1.3.6 丰富的流程定义	5
1.3.7 健壮的批处理应用	6
1.3.8 易扩展的批处理应用	6
1.3.9 复用企业现有 IT 资产	6
1.4 Spring Batch 2.0 新特性	6
1.4.1 支持 Java 5	7
1.4.2 支持非顺序的 Step	7
1.4.3 面向 Chunk 处理	7
1.4.4 元数据访问	11
1.4.5 扩展性	11
1.4.6 可配置性	12
1.5 Spring Batch 2.2 新特性	13
1.5.1 Spring Data 集成	13
1.5.2 支持 Java 配置	13
1.5.3 Spring Retry	14
1.5.4 Job Parameters	14
1.6 开发环境搭建	15
第 2 章 Spring Batch 之 Hello World	16
2.1 场景说明	16
2.2 项目准备	16
2.2.1 项目结构	16
2.2.2 准备对账单文件	17
2.2.3 定义领域对象	18
2.3 定义 job 基础设施	18
2.4 定义对账 Job	19
2.4.1 配置 ItemReader	19
2.4.2 配置 ItemProcessor	21
2.4.3 配置 ItemWriter	22
2.5 执行 Job	23
2.5.1 Java 调用	23
2.5.2 JUnit 单元测试	24
2.6 概念预览	26

第 2 篇 基本篇

第 3 章 Spring Batch 基本概念	28
3.1 命名空间	29
3.2 Job	30
3.2.1 Job Instance	31
3.2.2 Job Parameters	33
3.2.3 Job Execution	34
3.3 Step	35
3.3.1 Step Execution	37
3.4 Execution Context	38
3.5 Job Repository	39
3.5.1 Job Repository Schema	39

3.5.2 配置 Memory Job	5.3.5 读、处理事务	110
Repository	5.4 拦截器	112
3.5.3 配置 DB Job Repository	5.4.1 ChunkListener	115
3.5.4 数据库 Schema	5.4.2 ItemReadListener	116
3.6 Job Launcher	5.4.3 ItemProcessListener	116
3.7 ItemReader	5.4.4 ItemWriteListener	117
3.8 ItemProcessor	5.4.5 SkipListener	117
3.9 ItemWriter	5.4.6 RetryListener	118
第 4 章 配置作业 Job	第 6 章 读数据 ItemReader	120
4.1 基本配置	6.1 ItemReader	120
4.1.1 重启 Job	6.1.1 ItemReader	120
4.1.2 Job 拦截器	6.1.2 ItemStream	121
4.1.3 Job Parameters 校验	6.1.3 系统读组件	122
4.1.4 Job 抽象与继承	6.2 Flat 格式文件	122
4.2 高级特性	6.2.1 Flat 文件格式	123
4.2.1 Step Scope	6.2.2 FlatFileItemReader	125
4.2.2 属性 Late Binding	6.2.3 RecordSeparatorPolicy	129
4.3 运行 Job	6.2.4 LineMapper	130
4.3.1 调度作业	6.2.5 DefaultLineMapper	131
4.3.2 命令行执行	6.2.6 LineCallbackHandler	138
4.3.3 与定时任务集成	6.2.7 读分隔符文件	139
4.3.4 与 Web 应用集成	6.2.8 读定长文件	141
4.3.5 停止 Job	6.2.9 读 JSON 文件	143
第 5 章 配置作业步 Step	6.2.10 读记录跨多行文件	145
5.1 配置 Step	6.2.11 读混合记录文件	147
5.1.1 Step 抽象与继承	6.3 XML 格式文件	150
5.1.2 Step 执行拦截器	6.3.1 XML 解析	150
5.2 配置 Tasklet	6.3.2 Spring OXM	151
5.2.1 重启 Step	6.3.3 StaxEventItemReader	153
5.2.2 事务	6.4 读多文件	156
5.2.3 事务回滚	6.5 读数据库	159
5.2.4 多线程 Step	6.5.1 JdbcCursorItemReader	160
5.2.5 自定义 Tasklet	6.5.2 HibernateCursorItem	167
5.3 配置 Chunk	Reader	
5.3.1 提交间隔	6.5.3 StoredProcedureItem	
5.3.2 异常跳过	Reader	171
5.3.3 Step 重试	6.5.4 JdbcPagingItemReader	174
5.3.4 Chunk 完成策略	6.5.5 HibernatePagingItem	
	Reader	179

6.5.6 JpaPagingItemReader	183	7.8 Item 路由 Writer	254
6.5.7 IbatisPagingItemReader	186	7.9 发送邮件	258
6.6 读 JMS 队列	190	7.9.1 SimpleMailMessageItem Writer	258
6.6.1 JmsItemReader	190	7.10 服务复用	262
6.7 服务复用	194	7.10.1 ItemWriterAdapter	262
6.8 自定义 ItemReader	197	7.10.2 PropertyExtracting DelegatingItemWriter	264
6.8.1 不可重启 ItemReader	197	7.11 自定义 ItemWrite	267
6.8.2 可重启 ItemReader	199	7.11.1 不可重启 ItemWriter	267
6.9 拦截器	202	7.11.2 可重启 ItemWriter	268
6.9.1 拦截器接口	202	7.12 拦截器	271
6.9.2 拦截器异常	203	7.12.1 拦截器接口	271
6.9.3 执行顺序	204	7.12.2 拦截器异常	273
6.9.4 Annotation	204	7.12.3 执行顺序	274
6.9.5 属性 Merge	205	7.12.4 Annotation	274
第 7 章 写数据 ItemWriter	207	7.12.5 属性 Merge	275
7.1 ItemWrite	207	第 8 章 处理数据 ItemProcessor	277
7.1.1 ItemWriter	208	8.1 ItemProcessor	277
7.1.2 ItemStream	208	8.1.1 ItemProcessor	277
7.1.3 系统写组件	209	8.1.2 系统处理组件	278
7.2 Flat 格式文件	210	8.2 数据转换	279
7.2.1 FlatFileItemWriter	210	8.2.1 部分数据转换	279
7.2.2 LineAggregator	214	8.2.2 数据类型转换	281
7.2.3 FieldExtractor	217	8.3 数据过滤	282
7.2.4 回调操作	219	8.3.1 数据 Filter	282
7.3 XML 格式文件	222	8.3.2 数据过滤统计	283
7.3.1 StaxEventItemWriter	222	8.4 数据校验	285
7.3.2 回调操作	226	8.4.1 Validator	285
7.4 写多文件	230	8.4.2 ValidatingItemProcessor	286
7.4.1 MultiResourceItemWriter	230	8.5 组合处理器	288
7.4.2 扩展 MultiResourceItem Writer	233	8.6 服务复用	291
7.5 写数据库	234	8.6.1 ItemProcessorAdapter	291
7.5.1 JdbcBatchItemWriter	235	8.7 拦截器	293
7.5.2 HibernateItemWriter	239	8.7.1 拦截器接口	293
7.5.3 IbatisBatchItemWriter	242	8.7.2 拦截器异常	295
7.5.4 JpaItemWriter	245	8.7.3 执行顺序	295
7.6 写 JMS 队列	248	8.7.4 Annotation	296
7.6.1 JmsItemWriter	248	8.7.5 属性 Merge	297
7.7 组合写	252		

第3篇 高级篇

第 9 章 作业流 Step Flow.....	300	10.2.3 重试拦截器	343
9.1 顺序 Flow.....	300	10.2.4 重试模板	345
9.2 条件 Flow.....	302	10.3 重启 Restart	353
9.2.1 next	303	10.3.1 重启 Job.....	353
9.2.2 ExitStatus VS BatchStatus.....	306	10.3.2 启动次数限制	355
9.2.3 decision 条件	308	10.3.3 重启已完成的任务	355
9.3 并行 Flow.....	311	第 11 章 扩展 Job、并行处理.....	357
9.4 外部 Flow 定义.....	314	11.1 可扩展性	357
9.4.1 Flow	314	11.2 多线程 Step	358
9.4.2 FlowStep	317	11.2.1 配置多线程 Step	359
9.4.3 JobStep.....	319	11.2.2 线程安全性	360
9.5 Step 数据共享	321	11.2.3 线程安全 Step	361
9.6 终止 Job	323	11.2.4 可重启的线程安全 Step	363
9.6.1 end	324	11.3 并行 Step	365
9.6.2 stop.....	326	11.4 远程 Step	366
9.6.3 fail.....	327	11.4.1 远程 Step 框架	366
第 10 章 健壮 Job	330	11.4.2 基于 SI 实现远程 Step	368
10.1 跳过 Skip.....	331	11.5 分区 Step	373
10.1.1 配置 Skip	331	11.5.1 关键接口	374
10.1.2 跳过策略 SkipPolicy	333	11.5.2 基本配置	376
10.1.3 跳过拦截器	335	11.5.3 文件分区	378
10.2 重试 Retry	338	11.5.4 数据库分区	382
10.2.1 配置 Retry	339	11.5.5 远程分区 Step	387
10.2.2 重试策略 RetryPolicy	341	后记	392

第 1 篇 入门篇

本篇从基本的入门讲起，通过介绍批处理、Spring Batch 基本特性、新特性，快速入门的 Hello World 等内容引领读者入门，进入批处理的世界。本篇包含两个章节。

第 1 章：向读者介绍什么是批处理，Spring Batch 框架适用的业务场景、技术场景，Spring Batch 的核心三层架构、Spring Batch 2.0、2.2 新特性，最后带领读者一起搭建 Spring Batch 的开发环境。

第 2 章：通过经典的 Hello World 示例向读者全面展示了 Spring Batch 的入门示例。

Spring Batch 简介

1.1 什么是批处理

现代互联网企业、金融业、电信业甚至传统行业通过 OLTP（联机事务处理）的业务系统积累了海量企业数据，需要企业应用能够在关键任务中进行批量处理来操作业务逻辑。通常情况下，此类业务并不需要人工参与就能够自动高效地进行复杂数据处理与分析。例如，定期对大批量数据进行业务处理（如银行对账和利率调整、或者跨系统的数据同步），或者是把从内部和外部系统中获取到的数据进行处理后集成到其他系统中去，这类工作被称之为“批处理”。

“批处理”工作在面对复杂的业务以及海量的数据处理时，无需人工干预，仅需定期读入批量数据，然后完成相应业务处理并进行归档操作。

典型的批处理应用有以下几个特点：

- (1) 自动执行，根据系统设定的工作步骤自动完成；
- (2) 数据量大，少则百万，多则千万甚至上亿；
- (3) 定时执行，如每天执行、每周或每月执行。

从上面的描述中可以看出，批处理的整个流程可以明显地分为 3 个阶段：

- (1) 读数据，数据可能来自文件、数据库或消息队列等；
- (2) 处理数据，处理读取的数据并形成输出结果，如银行对账系统的资金对账处理；
- (3) 写数据，将输出结果写入文件、数据库或消息队列等。

一个典型的批处理应用场景：系统 A 从数据库获取数据，经过业务处理后，导出系统 B 需要的数据到文件中，系统 B 读取该文件，经过业务处理后，最后存放在数据库中。通常情况下该动作在每天夜间 12:00~2:00 之间进行，此时对系统的性能影响最小。图 1-1 给出了典型批处理应用的场景。

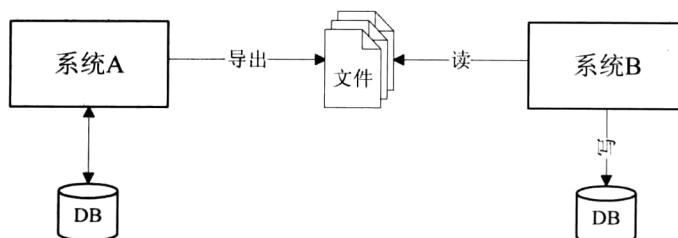


图 1-1 典型批处理应用场景

1.2 Spring Batch

Spring Batch 是一个轻量级的、完善的批处理框架，旨在帮助企业建立健壮、高效的批处理应用。Spring Batch 是 Spring 的一个子项目，使用 Java 语言并基于 Spring 框架为基础开发，使得已经使用 Spring 框架的开发者或者企业更容易访问和利用企业原有服务。

Spring Batch 提供了大量可重用的组件，包括日志、追踪、事务、任务作业统计、任务重启、跳过、重复、资源管理。对于大数据量和高性能的批处理任务，Spring Batch 同样提供了高级功能和特性来支持，比如分区功能、远程功能。总之，通过 Spring Batch 能够支持简单的、复杂的和大数据量的批处理作业。

Spring Batch 是一个批处理应用框架，不是调度框架，但需要和调度框架合作来构建完成批处理任务。它只关注批处理任务相关的问题，如事务、并发、监控、执行等，并不提供相应的调度功能。如果需要使用调用框架，在商业软件和开源软件中已经有很多优秀的企业级调度框架（如 Quartz、Tivoli、Control-M、Cron 等）可以使用。

1.2.1 典型场景

典型的批处理应用通常从数据库、文件或队列中读取数据，之后使用一些方法处理数据（抽取、分析、处理、过滤等），最终使用修改过的格式将数据写回目标系统。通常在一个无需用户交互的离线环境下，Spring Batch 能够自动进行基本的批处理迭代，也能够为一个数据集提供事务保证。批处理任务是一个大多数 IT 项目的组成部分，而 Spring Source 是唯一能够提供健壮的、企业级扩展的开源批处理框架。

Spring Batch 批处理框架支撑的业务场景：

- (1) 定期提交批处理任务；
- (2) 并行批处理，即并行处理任务；
- (3) 企业消息驱动处理；
- (4) 大规模的并行处理；
- (5) 手动或定时重启；
- (6) 按顺序处理依赖的任务（可扩展为工作流驱动的批处理）；
- (7) 部分处理，如在回滚时忽略记录；
- (8) 完整的批处理事务。

Spring Batch 批处理框架支撑的技术目标：

- (1) 利用 Spring 编程模型，使程序员专注于业务处理，让 Spring 框架管理流程；
- (2) 明确分离批处理的执行环境和应用；
- (3) 将通用核心的服务以接口形式提供；
- (4) 提供“开箱即用”的简单的默认的核心执行接口；
- (5) 提供 Spring 框架中配置、自定义和扩展服务；

- (6) 所有默认实现的核心服务能够容易地被扩展与替换，不会影响基础层；
- (7) 提供一个简单的部署模式，使用 Maven 进行编译。

1.2.2 Spring Batch 架构

Spring Batch 核心架构分为三层：应用层、核心层、基础架构层。具体参见图 1-2。

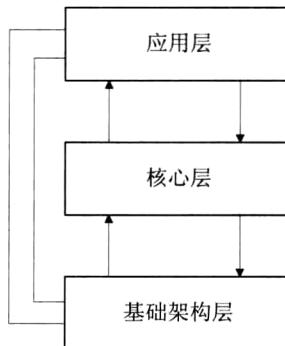


图 1-2 Spring Batch 三层核心架构

应用层包含所有的批处理作业，通过 Spring 框架管理程序员自定义的代码。核心层包含 Spring Batch 启动和控制所需要的核心类，如：JobLauncher、Job 和 step 等。应用层和核心层建立在基础构架层之上，基础构架层提供通用的读（ItemReader）、写（ItemWriter）和服务处理（如：RetryTemplate：重试模板；RepeatTemplate：重复模板，可以被应用层和核心层使用）。Spring Batch 的三层体系架构使得 Spring Batch 框架在不同的层级进行扩展，避免不同层级间的影响。

1.3 Spring Batch 优势

Spring Batch 是由 SpringSource 和 Accenture（埃森哲）合作开发的。Accenture 在批处理架构上有着丰富的工业级别的经验，贡献了之前专用的批处理体系框架（这些框架历经数十年研发和使用，为 Spring Batch 提供了大量的参考经验）；SpringSource 则有着深刻的技术认知和 Spring 框架编程模型。

Spring Batch 框架通过提供丰富的即开即用的组件和高可靠性、高扩展性的能力，使得开发批处理应用的人员专注于业务的处理，提升批处理应用的开发效率，通过 Spring Batch 可以快速地构建出轻量级的健壮的并行处理应用。

使用 Spring Batch 框架，你可以获得如下几小节所述的优势。

1.3.1 丰富的开箱即用组件

开箱即用组件包括对各种类型资源的读、写。

- 读：支持文本文件读、XML 文件读、数据库读，JMS 队列读。
- 写：支持写文本文件、XML 文件、数据库、JMS 队列。

该组件还提供作业仓库、作业调度器等基础设施，大大简化了批处理应用开发的复杂度。

1.3.2 面向 Chunk 的处理

面向 Chunk 的处理，支持多次读、一次写，避免了多次对资源的写入，大幅提升了批处理应用的处理效率。

1.3.3 事务管理能力

Spring Batch 框架默认采用 Spring 提供的声明式事务管理模型，面向 Chunk 的操作支持事务管理，同时支持为每个 tasklet 操作设置细粒度的事务配置：隔离级别、传播行为、超时设置等。

1.3.4 元数据管理

Spring Batch 框架自动记录 Job 的执行情况，包括 Job 的执行成功、失败、失败的异常信息，Step 的执行成功、失败、失败的异常信息，执行次数，重试次数，跳过次数，执行时间等，方便后期的维护和查看。

1.3.5 易监控的批处理应用

Spring Batch 框架提供多种监控技术，支持对批处理操作的信息进行查看和管理，通过 Spring Batch 框架为批处理应用提供了灵活的监控模式，包括：

- 直接查看数据库；
- 通过 Spring Batch 提供的 API 查看，基于 API，你可以打造自己的管理监控台；
- 通过 Spring Batch Admin 进行查看，Spring Batch Admin 是 Spring 的另外一个项目，通过该项目你可以通过 Web 控制台监控和操作 Job；
- 通过 JMX 控制台查看。

1.3.6 丰富的流程定义

Spring Batch 框架支持顺序任务、条件分支任务，基于顺序和条件任务可以组织复杂的任务流程。同时 Spring Batch 支持复用已经定义的 Job 或者 Step，同时提供 Job 和 Step 的继承

能力，方便任务的抽象。

1.3.7 健壮的批处理应用

Spring Batch 框架支持作业的跳过、重试、重启能力，避免因错误导致批处理作业的异常中断，例如如下操作。

- 跳过 (Skip): 通常在发生非致命异常的情况下，应该不中断批处理应用；
- 重试 (Retry): 发生瞬态异常情况下，应该能够通过重试操作避免该类异常，保证批处理应用的连续性和稳定性；
- 重启 (Restart): 当批处理应用因错误发生错误后，应该能够在最后执行失败的地方重新启动 Job 实例。

1.3.8 易扩展的批处理应用

Spring Batch 框架通过并发和并行技术实现应用的横向、纵向扩展机制，满足数据处理性能的需要。

扩展机制包括：

- 多线程执行一个 Step (Multithreaded step);
- 多线程并行执行多个 Step (Parallelizing step);
- 远程执行作业 (Remote chunking);
- 分区执行 (Partitioning Step)。

1.3.9 复用企业现有 IT 资产

Spring Batch 框架提供多种 Adapter 能力，使得企业现有的服务可以方便地集成到批处理应用中，避免了重新开发，达到复用企业遗留的服务资产。

1.4 Spring Batch 2.0 新特性

相对于 Spring Batch 1.X 系列，Spring Batch 2.X 系列提供了如下新的特性：

- (1) 支持 Java 5；
- (2) 非顺序的 Step 支持；
- (3) 面向 Chunk 处理；
- (4) 强化元数据访问；
- (5) 增强扩展性；
- (6) 可配置。

1.4.1 支持 Java 5

从 Spring Batch 2.X 版本开始，使用 Java 5 进行开发，支持 Java 5 提供的增强特性，如泛型、参数化类型等。

1.4.2 支持非顺序的 Step

Spring Batch 2.0 支持条件判断执行 Step 的方式。在 2.0 版本之前，仅支持顺序执行 Step。顺序执行 Step 参见图 1-3。

新的 Step 执行方式增加了条件判断功能，参见图 1-4。

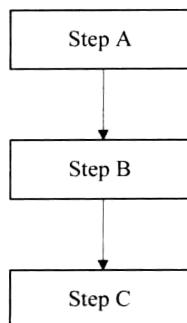


图 1-3 Spring Batch 1.X 版本仅支持顺序 Step

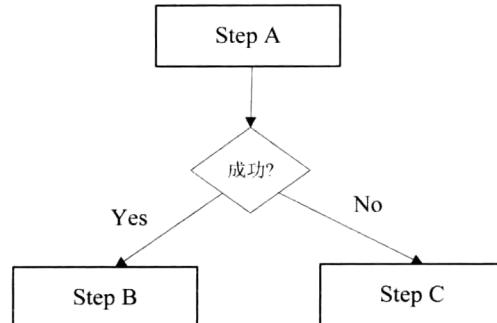


图 1-4 Spring Batch 2.X 版本支持条件 Step

代码清单 1-1 展示了如何配置图 1-4 中的条件流程。

代码清单 1-1 配置条件流程示例

```
1.  <job id="job">
2.    <step id="stepA">
3.      <next on="FAILED" to="stepB" />
4.      <next on="*" to="stepC" />
5.    </step>
6.    <step id="stepB" next="stepC" />
7.    <step id="stepC" />
8.  </job>
```

1.4.3 面向 Chunk 处理

Spring Batch 1.X 版本对数据处理默认提供的策略是面向 Item 处理。其序列图参见图 1-5。

在面向 Item 处理中，ItemReader 会返回一个对象（即 Item）给 ItemWriter 进行处理，Item 的数量为提交间隔的要求时提交计算结果。例如，如果提交所要求的 Item 数量为 3 时，ItemReader 和 ItemWriter 分别会被调用 3 次。使用代码清单 1-2 表示图 1-5 的执行。

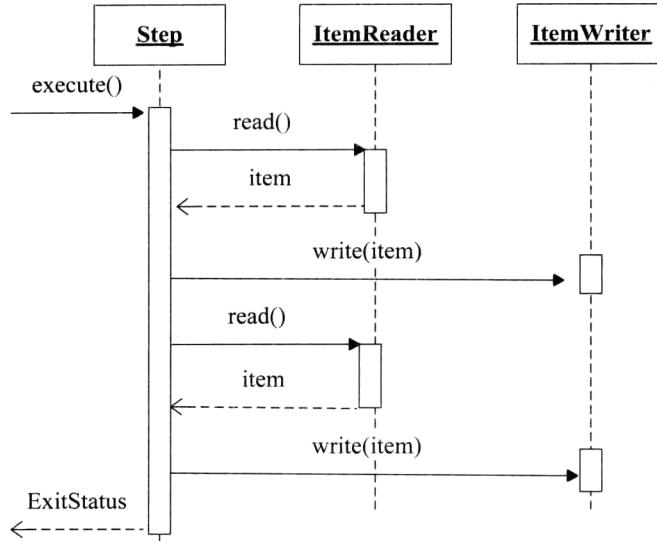


图 1-5 Spring Batch 1.X 版本支持面向 Item 的处理之序列图

代码清单 1-2 展示面向 Item 的处理伪代码

```

1.   for(int i = 0; i < commitInterval; i++) {
2.       Object item = itemReader.read();
3.       itemWriter.write(item);
4.   }
  
```

对应的 ItemReader 和 ItemWriter 为了实现回滚的场景，需要在内部定义复杂的方法（如 mark 标记方法、reset 恢复方法、clear 清除方法等）。接口 ItemReader 和 ItemWriter 在 Spring Batch 1.X 版本的实现参见代码清单 1-3。

代码清单 1-3 Spring Batch 1.X 版本中 ItemReader、ItemWriter 的接口声明

```

1. public interface ItemReader {
2.     Object read() throws Exception;
3.     void mark() throws MarkFailedException;
4.     void reset() throws ResetFailedException;
5. }
6. public interface ItemWriter {
7.     void write(Object item) throws Exception;
8.     void flush() throws FlushFailedException;
9.     void clear() throws ClearFailedException;
10. }
  
```

由于处理的范围是一个 Item，如果要支持回滚场景就需要额外的方法，此时 mark、reset、flush 和 clear 就派上了用场。例如，在成功读/写了 2 个 item 之后，在写第三个 item 时发生了错误，整个事务就需要回滚，writer 中的 clear 方法会被调用，用于清空缓存，Itemreader 中的

reset 被调用，用于把 mark 方法所指向的数据游标复原。

Spring Batch 2.0 中支持面向 Chunk 的操作，简化了 ItemReader 和 ItemWriter 接口的复杂度。面向 Chunk 的操作序列图参见图 1-6。

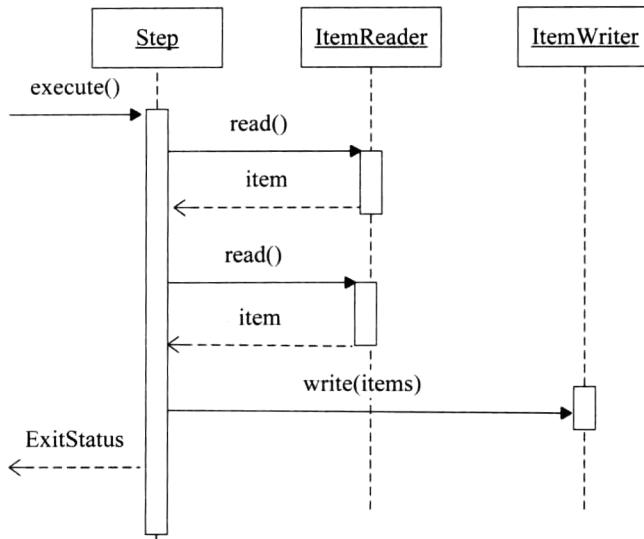


图 1-6 Spring Batch 2.X 版本支持面向 Chunk 的操作之序列图

按照面向 Chunk 的操作，如果提交间隔是 3 次，那么读操作被调用 3 次，写操作被调用 1 次。读 Item 被汇总到列表中，最终被统一写出。使用代码清单 1-4 来表示图 1-6 的执行：

代码清单 1-4 展示面向 Chunk 的处理伪代码

```
1. List items = new ArrayList();
2. for(int i = 0; i < commitInterval; i++){
3.     items.add(itemReader.read());
4. }
5. itemWriter.write(items);
```

面向 Chunk 的方案不仅更加简单更有扩展性，同时也让 ItemReader 和 ItemWriter 接口更加简洁。接口 ItemReader 和 ItemWriter 在 Spring Batch 2.X 版本的实现参见代码清单 1-5。

代码清单 1-5 Spring Batch 2.X 版本中 ItemReader、ItemWriter 的接口声明

```
1. public interface ItemReader<T> {
2.     T read() throws Exception, UnexpectedInputException, ParseException,
3.             NonTransientResourceException;
4. }
5. public interface ItemWriter<T> {
6.     void write(List<? extends T> items) throws Exception;
7. }
```

如代码所示，ItemReader 和 ItemWriter 接口不再包含 mark, reset, flush 和 clear 方法，使得读和写对象的创建更加直接。ItemReader 例子中，接口非常简单，框架会为开发者把读取的 item 缓存起来，以防 rollback 情况的发生。ItemWriter 也很简单，不再一次一个 item 地拿取，而是一次把整个 item 块都拿到，在把控制权交还给 step 前决定资源（如文件、数据库或者消息队列）的写入。

1.4.3.1 增加 ItemProcessor

在 Spring Batch 2.0 之前，Step 只依赖 ItemReader 和 ItemWriter，在 Spring Batch 2.0 中引入了 ItemProcessor（负责业务数据的处理）。

Spring Batch 1.X 的 Step 结构参见图 1-7。

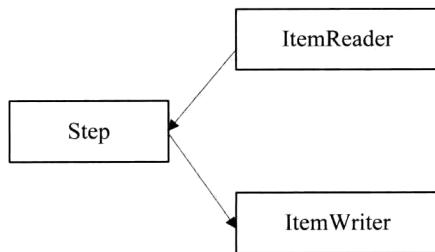


图 1-7 Spring Batch 1.X 的 Step 结构

通常的业务场景需要在数据写入之前，对数据进行处理，在 Spring Batch 1.X 版本中，可以使用组合模式，通过在读/写之间加入 ItemTransformer 这一层来实现。增加 ItemTransformer 之后的架构图参见图 1-8。

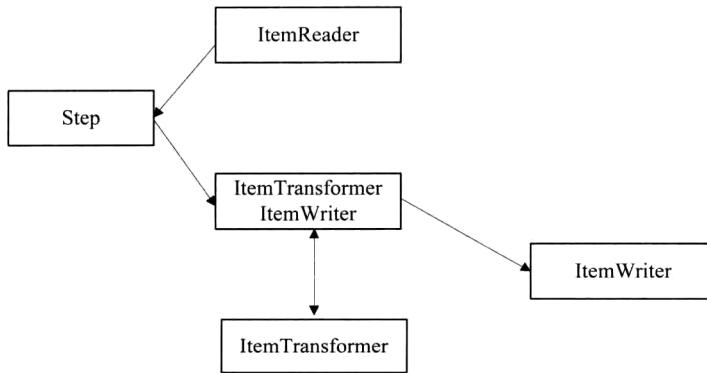


图 1-8 Spring Batch 1.X 通过 ItemTransformer 实现数据处理

Spring Batch 2.0 版本之后的 Step，将 ItemTransformer 重新命名为 ItemProcessor，和 ItemReader 与 ItemWriter 提升为相同的层级。Spring Batch 2.0 中增加 ItemProcessor 的架构图参见图 1-9。

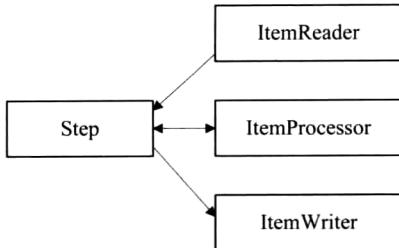


图 1-9 Spring Batch 2.0 通过 ItemProcessor 实现数据处理

1.4.4 元数据访问

Spring Batch 1.X 提供了元数据的增删查改接口 JobRepository。在 Spring Batch 2.X 版本中，新增元数据访问接口 JobExplorer 和 JobOperator。元数据访问接口的关系参见图 1-10。

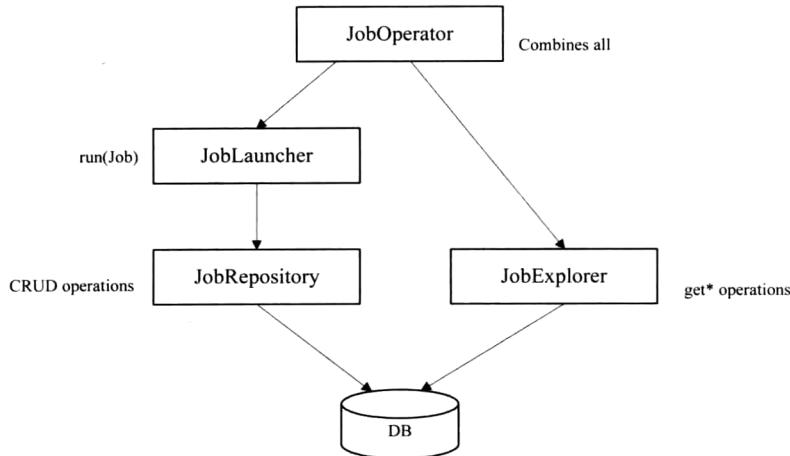


图 1-10 Spring Batch 2.X 中元数据管理接口关系图

1.4.5 扩展性

Spring Batch 1.X 只能在单个 JVM 中运行，支持任务在单个 JVM 多个线程并发执行。在 Spring Batch 2.0 增加支持多进程执行任务，包括远程分块和分区两个新功能。

远程分块

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

块的特性。

分区

分区是另一个可选方案，它需要对数据的结构有一定的了解，如主键的范围、待处理的文件的名字等。这种模式的优点在于分区中每一个元素的处理器都能够像一个普通 Spring Batch 任务的单步一样运行，也不必去实现任何特殊的或是新的模式，来让它们能够更容易配置与测试。分区理论上比远程分块更有扩展性，因为分区并不存在从一个地方读取所有输入数据并进行序列化的瓶颈。

在 Spring Batch 2.0 中有两个接口支持分区：PartitionHandler 和 StepExecutionSplitter。PartitionHandler 知道执行结构——它需要将请求发送到远程步骤，并使用任何可以使用的网格或是远程技术收集计算结果。PartitionHandler 是一个 SPI，它和 Spring Batch 通过 TaskExecutor 为本地执行提供了一个外部实现方式。在需要进行有大量 I/O 操作的并发处理时，这个功能是很有用的。

1.4.6 可配置性

Spring Batch 1.X 没有独立的命名空间，所有批处理的配置需要作为 Spring 的配置项；Spring Batch 2.X 版本中增加了批处理的命名空间，简化了配置操作，对应的 XSD 可以通过 <http://www.springframework.org/schema/batch/spring-batch-2.2.xsd> 访问获取。

Spring Batch 1.X 中批处理配置文件参见代码清单 1-6。

代码清单 1-6 Spring Batch 1.X 中批处理配置文件示例代码

```
1. <bean id="footballJob"
2.       class="org.springframework.batch.core.job.SimpleJob">
3.         <property name="steps">
4.           <list>
5.             <!-- Step bean details omitted for clarity -->
6.             <bean id="playerload"/>
7.             <bean id="gameLoad"/>
8.             <bean id="playerSummarization"/>
9.           </list>
10.        </property>
11.        <property name="jobRepository" ref="jobRepository" />
12.      </bean>
```

Spring Batch 2.X 中批处理配置文件参见代码清单 1-7。

代码清单 1-7 Spring Batch 2.X 中批处理配置文件示例代码

```
1. <job id="footballJob">
2.   <!-- Step bean details omitted for clarity -->
3.   <step id="playerload" next="gameLoad"/>
```

```
4.      <step id="gameLoad" next="playerSummarization"/>
5.      <step id="playerSummarization"/>
6.  </job>
```

1.5 Spring Batch 2.2 新特性

相对于 Spring Batch 2.0 系列，Spring Batch 2.2.X 系列提供了如下新的特性：

- (1) 支持 Spring Data 集成；
- (2) 支持 Java 配置；
- (3) 重试模块（Spring Retry）重构；
- (4) 作业参数变化（Job Parameters）。

1.5.1 Spring Data 集成

Spring Batch 2.2 版本提供了对 Spring Data 的支持。Spring Data 是一个用于简化数据库访问，并支持云服务的开源框架。其主要目标是使得数据库的访问变得方便快捷，并支持 map-reduce 框架和云计算数据服务。此外，它还支持基于关系型数据库的数据服务，如 Oracle RAC 等。对于拥有海量数据的项目，可以用 Spring Data 来简化项目的开发，就如 Spring Framework 对 JDBC、ORM 的支持一样，Spring Data 会让数据的访问变得更加方便。

Spring Data 提供了对 NoSQL 类型的数据库提供了支持，Spring Batch 框架新增对 NoSQL 类型的数据库 MongoDB、Neo4j、Gemfire 的支持。

1.5.2 支持 Java 配置

Spring Batch 2.2 之前的版本提供了基于 XML 的方式配置 Job 作业，Spring Batch 2.2 之后的版本新增了 Java 方式的配置。为 Java 配置框架提供一组 annotation (@EnableBatchProcessing) 和作业、作业步的工厂类 (JobBuilderFactory、StepBuilderFactory)。

Spring Batch 2.2 之前版本提供的基于 XML 方式配置参见代码清单 1-8。

代码清单 1-8 Spring Batch 2.2 之前提供的基于 XML 方式定义作业

```
1.  <batch>
2.    <job-repository/>
3.    <job id="myJob">
4.      <step id="step1" .../>
5.      <step id="step2" .../>
6.    </job>
7.    <beans:bean id="transactionManager" .../>
8.    <beans:bean id="jobLauncher"
9.      class="org.springframework.batch.core.launch.support.
```

```
        SimpleJobLauncher">
10.      <beans:property name="jobRepository" ref="jobRepository"/>
11.    </beans:bean>
12.  </batch>
```

Spring Batch 2.2 之后版本提供的基于 Java 配置方式参见代码清单 1-9。

代码清单 1-9 Spring Batch 2.2 之后提供的基于 Java 配置方式定义作业

```
1.  @Configuration
2.  @EnableBatchProcessing
3.  @Import(DataSourceCnfiguration.class)
4.  public class AppConfig {
5.      @Autowired
6.      private JobBuilderFactory jobs;
7.
8.      @Bean
9.      public Job job() {
10.          return jobs.get("myJob").start(step1()).next(step2()).build();
11.      }
12.
13.      @Bean
14.      protected Step step1() {
15.          ...
16.      }
17.
18.      @Bean
19.      protected Step step2() {
20.          ...
21.      }
22.  }
```

@EnableBatchProcessing 会依赖默认的 Spring Bean 对象，包括 JobRepository（作业仓库）、JobLauncher（作业调度器）、JobRegistry（作业注册器）、PlatformTransactionManager（事务管理器）、JobBuilderFactory（作业构建工厂）、StepBuilderFactory（作业步构建工厂）。

1.5.3 Spring Retry

Spring Batch 提供的重试功能已经被抽取出来作为 Spring 一个独立的组件 Spring Retry 发布。重试组件在 Spring Batch 2.2 之前的包为：org.springframework.batch.retry；在 Spring Batch 2.2 的版本重构为 org.springframework.retry。

1.5.4 Job Parameters

在 Spring Batch 2.2 之前的版本，作业参数（Job Parameters）被强制用来标识作业实例（Job

Instance)；在 Spring Batch2.2 版本之后作业参数是否标识作业实例用户可以自由选择。

Spring Batch 2.2 之前的版本强制作业参数作为作业的实例，导致在作业重启的情况下无法修改作业参数；Spring Batch 2.2 之后的版本修改了这个缺陷。为此 Spring Batch 修改了自己的数据模型，在 Spring Batch 2.2 之前的版本作业参数与作业实例 ID 进行关联；Spring Batch 2.2 之后的版本将作业参数与作业执行器的 ID 关联。

1.6 开发环境搭建

Spring 批处理框架 1.0.0 版本于 2008 年 3 月份推出，推出版本非常快，截止到目前 2014 年 10 月最新的 2.X 系列 Release 的 GA 版本是 2.2.7 版本；3.X 系列已经推出 3.0.1 的 GA 版本，3.1.0 的 SNAPSHOT 已经推出。本书写作时使用的是 2.2.1 的 GA 版本，本书的所有源代码都是在 2.2.1 版本上调试通过的。本书配套源代码可以在 <https://github.com/jxtaliu/SpringBatchSample.git> 免费下载。

如果你是一个新的开发者，请到 Spring Batch 官网 <http://projects.spring.io/spring-batch/> 去了解一下 Spring Batch 框架的内容。具体的下载地址为：<http://static.springsource.org/downloads/nightly/release-download.php?project=BATCH>；请下载最新的 RELEASE 2.2.1 版本（**BATCH/spring-batch-2.2.1.RELEASE-no-dependencies.zip**）。

下载后的文件为 `spring-batch-2.2.1.RELEASE-no-dependencies.zip`，请解压缩到目录 `spring-batch-2.2.1.RELEASE` 中，图 1-11 展示了解压缩后的目录结构。

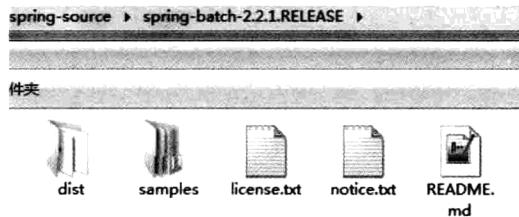


图 1-11 Spring Batch 2.2.1 解压缩后的目录结构

其中，

- `dist`: 编译好的 jar 和源文件压缩包。
- `samples`: 示例工程，默认是 maven 构件项目。
- `license.txt`: 授权的 license 文件，采用 Apache License。
- `notice.txt`: 使用 Spring Batch 的注意事项。
- `README.md`: 说明文档，包含如何获取源代码、编译及开发工具。

注意：为了编译 `samples` 项目，你需要安装 Maven，如果你还没有使用过 Maven 构建项目，请通过 Google 学习。Maven 的官方网站地址为：<http://maven.apache.org/>。

接下来的章节，我们通过“Hello World”示例展示如何快速使用 Spring Batch 框架。

Spring Batch 之 Hello World

本章通过一个实际信用卡账单对账例子，详细描述 Spring 批处理框架的使用方法，通过该例子，读者可以学会使用 Spring Batch 已经提供好的功能组件和基础设施，快速搭建批处理应用。

2.1 场景说明

个人使用信用卡消费，银行定期发送银行卡消费账单，本例模拟银行处理个人信用卡消费对账单对账，银行需要定期地把个人消费的记录导出成 CSV 格式文件，然后交给对账系统处理，本例子模拟银行读入 CSV 文件，经过处理后，生成新的对账单。示例场景架构参见图 2-1。

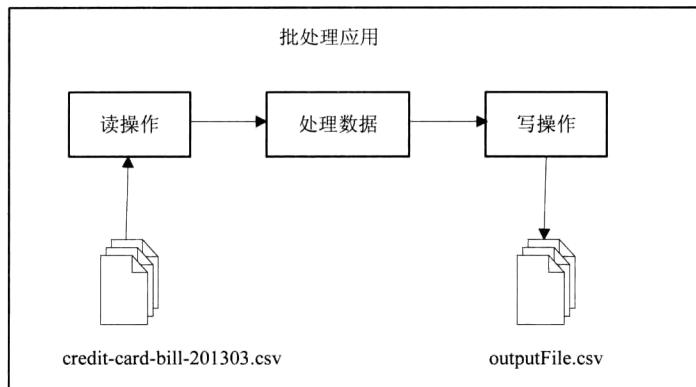


图 2-1 Hello World 信用卡消费对账单

2.2 项目准备

2.2.1 项目结构

项目的开发完成后的结构参见图 2-2。

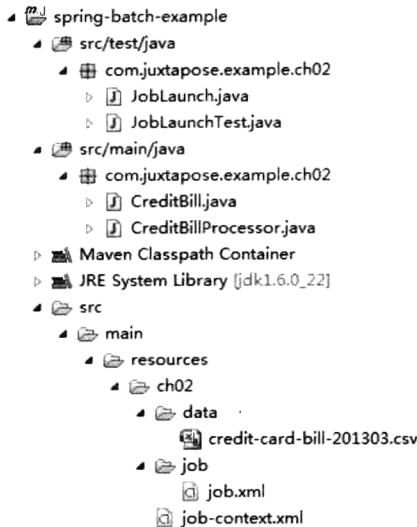


图 2-2 项目结构清单

文件说明如下。

- CreditBill.java：表示信用卡消费记录领域对象。
- CreditBillProcessor.java：记录处理类，本场景中没有做任何业务操作，仅打印账单信息。
- JobLaunch.java：调用批处理作业类。
- JobLaunchTest.java：JUnit 单元测试类，使用 Spring 提供的测试框架类。
- credit-card-bill-201303.csv：原始账单文件，存放账单消费条目。
- job.xml：作业定义文件。
- job-context.xml：Spring Batch 批处理任务需要的基础信息。

2.2.2 准备对账单文件

在介绍代码之前，我们一起先看一下信用卡消费清单文件，该文件是 CSV（Comma Separated Value 的简写，通常都是纯文本文件）格式文件。credit-card-bill-201303.csv 的文件内容参见代码清单 2-1。

代码清单 2-1 credit-card-bill-201303.csv

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

该对账单文件，每行表示一次信用卡的消费记录，中间用逗号分隔，分别表示银行卡账户、持卡人姓名、消费金额、消费日期、消费场所。

2.2.3 定义领域对象

为了映射上面的对账文件，需要开发信用卡账单领域对象 CreditBill，主要包括银行卡账户、持卡人姓名、消费金额、消费日期、消费场所等信息。该类的用途主要用于在 ItemReader 读取 CSV 文件中的数据后，转换为领域对象 CreditBill，供 ItemProcessoe 和 ItemWriter 使用。

领域对象 CreditBill 代码（此处省略了 get*，set*方法，以节省篇幅）参见代码清单 2-2。

代码清单 2-2 领域对象 CreditBill

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

2.3 定义 job 基础设施

接下来介绍如何在 Spring 配置文件中定义批处理任务，首先介绍 job-context.xml 内容，job-context.xml 中定义了批处理任务中需要的基础设施，主要配置任务仓库、任务调度器、任务执行中用到的事务管理器。job-context.xml 内容参见代码清单 2-3。

代码清单 2-3 配置文件 job-context.xml

```
1. <bean id="jobRepository"  
2.       class="org.springframework.batch.core.repository.support.  
3.               MapJobRepositoryFactoryBean">  
4. </bean>  
5. <bean id="jobLauncher"  
6.       class="org.springframework.batch.core.launch.support.  
7.               SimpleJobLauncher">  
8.     <property name="jobRepository" ref="jobRepository"/>  
9. </bean>  
10. <bean id="transactionManager"  
11.      class="org.springframework.batch.support.transaction.  
               ResourcelessTransactionManager"/>
```

该程序说明如下。

1~4 行：定义作业仓库，在 Spring Batch 框架中，任何任务的操作都会被记录在作业仓库中，Spring Batch 提供两种仓库，一种是内存的，另一种是数据库的。本例子中采用了内存

方式记录 Job 执行期产生的状态信息。

5~8 行：定义作业调度器，用来启动 Job；引用了 1~4 行定义的作业仓库。

9~11 行：定义了事务管理器，用于 Spring Batch 框架在对数据操作过程中提供事务能力。

2.4 定义对账 Job

job.xml 中定义批处理作业 billJob，billJob 共有一个 step，命名为 billStep，包含读、处理、写三个操作。job 配置内容参见代码清单 2-4。

代码清单 2-4 配置文件 job.xml

```
1.      <bean:import resource="classpath:ch02/job-context.xml"/>
2.      <job id="billJob">
3.          <step id="billStep">
4.              <tasklet transaction-manager="transactionManager">
5.                  <chunk reader="csvItemReader" writer="csvItemWriter"
6.                      processor="creditBillProcessor" commit-interval="2">
7.                  </chunk>
8.              </tasklet>
9.          </step>
10.     </job>
```

该程序说明如下。

1 行：导入 job-context.xml 文件，是 Spring 提供的功能，可以在不同的文件中定义 Spring 的配置信息，然后通过 import 方式组织导入。

2 行：定义名字为 billJob 的作业，该作业由一个名为 billStep 的 step 组成。

3~9 行：定义名字为 billStep 的作业步，billStep 作业步由一个面向批的操作组成。

4 行：定义批处理操作采用 job-context.xml 中定义的事务管理器，负责批处理中事务管理操作。

5~6 行：定义了面向批的操作，定义了读操作 csvItemReader、处理操作 creditBillProcessor、写操作 csvItemWriter； csvItemReader 负责从文件 credit-card-bill-201303.csv 中读取数据，creditBillProcessor 负责处理文件中的每一行数据，csvItemWriter 负责将 creditBillProcessor 处理的数据写到文件 outputFile.csv 中； commit-interval=2 表示提交间隔的大小，即每处理 2 条数据，进行一次写入操作，这样可以提高写的效率，在面向大数据量的批处理操作中，可以将 commit-interval 设置 1000~10000 的值。

2.4.1 配置 ItemReader

批处理操作的第一步是读取文本文件中的数据。csvItemReader 负责从文件中读取数据，并转换为信用卡对账单对象 CreditBill。 csvItemReader 的配置内容参见代码清单 2-5。

代码清单 2-5 配置读数据 csvItemReader

```
1.      <!-- 读取信用卡账单文件,CSV 格式 -->
2.      <bean:bean id="csvItemReader"
3.          class="org.springframework.batch.item.file.FlatFileItemReader"
4.          scope="step">
5.          <bean:property name="resource"
6.              value="classpath:ch02/data/credit-card-bill-201303.csv"/>
7.          <bean:property name="lineMapper">
8.              <bean:bean
9.                  class="org.springframework.batch.item.file.mapping.
10.                     DefaultLineMapper">
11.                     <bean:property name="lineTokenizer" ref="lineTokenizer"/>
12.                     <bean:property name="fieldSetMapper">
13.                         <bean:bean class="org.springframework.batch.item.file.
14.                            mapping.BeanWrapperFieldSetMapper">
15.                             <bean:property name="prototypeBeanName" value=
16.                               "creditBill">
17.                             </bean:property>
18.                         </bean:bean>
19.                     </bean:property>
20.                 </bean:bean>
21.                 <!-- lineTokenizer -->
22.                 <bean:bean id="lineTokenizer"
23.                     class="org.springframework.batch.item.file.transform.
24.                        DelimitedLineTokenizer">
25.                     <bean:property name="delimiter" value=","/>
26.                     <bean:property name="names">
27.                         <bean:list>
28.                             <bean:value>accountID</bean:value>
29.                             <bean:value>name</bean:value>
30.                             <bean:value>amount</bean:value>
31.                             <bean:value>date</bean:value>
32.                             <bean:value>address</bean:value>
33.                         </bean:list>
34.                     </bean:property>
35.                 </bean:bean>
```

该程序说明如下。

2~3 行: csvItemReader 由 Spring Batch 提供的基础设施定义,此例使用 FlatFileItemReader 读文本文件; FlatFileItemReader 有两个属性: 一个是 resource, 表示需要读取的文件资源; 另

一个是 lineMapper，通过 lineMapper 可以把文本中的一行转换为领域对象 Creditbill。

5~6 行：读操作的文件，本例中读文件为 classpath:ch02/data/credit-card-bill-201303.csv。

7 行：定义 lineMapper 属性，本例中使用 Spring Batch 提供的基础设施 org.springframework.batch.item.file.mapping.DefaultLineMapper。DefaultLineMapper 有两个基本属性 lineTokenizer 和 fieldSetMapper。

10 行：定义 lineTokenizer 属性，lineTokenizer 定义文本中每行的分隔符号，以及每行映射成 FieldSet 对象后的 name 列表。具体定义参见 22~33 行。

11~17 行：定义 fieldSetMapper 属性，本例中使用 Spring Batch 提供的基础设施 org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper。BeanWrapperFieldSetMapper 根据 lineTokenizer 中定义的 names 属性映射到 creditBill 中，最终组装为信用卡账单对象 CreditBill。

22~33 行：定义 lineTokenizer，用于指定分隔符策略为","，并将分隔后的字段映射到属性 names 中定义的字段上，最终将字段映射成 FieldSet 对象中的 name 列表。

2.4.2 配置 ItemProcessor

批处理操作第二步是处理 ItemReader 中读取的数据。creditBillProcessor 负责处理 2.4.1 节中 csvItemReader 读入的数据，本例中的处理操作功能简单，仅仅打印了对账单信息，没有进行任何业务上的处理。creditBillProcessor 的配置内容参见代码清单 2-6。

代码清单 2-6 配置处理数据 creditBillProcessor

```
1.      <bean:bean id="creditBillProcessor" scope="step"
2.          class="com.juxtapose.example.ch02.CreditBillProcessor">
3.      </bean:bean>
```

com.juxtapose.example.ch02.CreditBillProcessor 处理类定义参见代码清单 2-7。CreditBillProcessor 实现了接口 ItemProcessor，方法 process 用于处理 csvItemReader 读取的数据（参数 bill），并将处理后的数据返回，返回的数据会被写操作 ItemWriter 处理。

代码清单 2-7 数据处理类 CreditBillProcessor

```
1.  public class CreditBillProcessor implements
2.      ItemProcessor<CreditBill, CreditBill> {
3.
4.      public CreditBill process(CreditBill bill) throws Exception {
5.          System.out.println(bill.toString());
6.          return bill;
7.      }
8.  }
```

2.4.3 配置 ItemWriter

批处理操作的第三步是将 ItemProcessor 中处理的数据写入给定的目标存储中。 csvItemWriter 负责将 ItemProcessor 处理后的信用卡账单对象 CreditBill 写入到文本文件 outputFile.csv 中。

csvItemWriter 的配置信息参见代码清单 2-8。

代码清单 2-8 配置写数据 csvItemWriter

```
1.      <!-- 写信用卡账单文件，CSV 格式 -->
2.      <bean:bean id="csvItemWriter"
3.          class="org.springframework.batch.item.file.FlatFileItemWriter"
4.          scope="step">
5.          <bean:property name="resource" value="file:target/ch02/outputFile.
6.          csv"/>
7.          <bean:property name="lineAggregator">
8.              <bean:bean
9.                  class="org.springframework.batch.item.file.transform.
10.                     DelimitedLineAggregator">
11.                     <bean:property name="delimiter" value=","/></bean:property>
12.                     <bean:property name="fieldExtractor">
13.                         <bean:bean
14.                             class="org.springframework.batch.item.file.
15.                             transform.BeanWrapperFieldExtractor">
16.                             <bean:property name="names"
17.                                 value="accountID, name, amount, date, address">
18.                                 </bean:property>
19.                             </bean:bean>
20.                         </bean:property>
21.                     </bean:bean>
22.                 </bean:property>
23.             </bean:bean>
```

该程序说明如下。

2~3 行： csvItemWriter 由 Spring Batch 提供的基础设施定义，此例使用 FlatFileItemWriter 写入文本文件； FlatFileItemWriter 有两个属性：一个是 resource，表示需要写入的文件资源；另一个是 lineAggregator， lineAggregator 负责将信用卡账单对象根据定义的规则转换为一个文本。

5 行： 定义 resource 属性，本例中写入的文件是： file:target/ch02/outputFile.csv。

6~21 行： 定义 lineAggregator 属性， lineAggregator 默认使用了 Spring Batch 提供的基础设施类 org.springframework.batch.item.file.transform.DelimitedLineAggregator，DelimitedLineAggregator 共有两个属性：一个是 delimiter，另一个是 fieldExtractor。

10 行： 表示写入的文本以逗号分隔。

11~19 行：表示将信用卡账单对象 Creditbill 根据提供的 names 属性定义的名字转换为 Object 数组，最终由 DelimitedLineAggregator 转化为一行以逗号分隔的文本。

2.5 执行 Job

本节提供两种方式执行批处理作业：Java 调用和 Junit 单元测试。

2.5.1 Java 调用

到此为止，Spring Batch 配置文件完成，ItemProcessor 和信用卡账单对象 CreditBill 也已经完成，下面描述如何执行定义任务 billJob。还记得上面我们提到的 jobLauncher 对象吗，我们可以利用 jobLauncher 对作业进行调度。通过 Java 调用批处理作业的代码参见代码清单 2-9。

代码清单 2-9 Java 调用批处理作业

```
1. ApplicationContext context = new ClassPathXmlApplicationContext(  
    "ch02/job/job.xml");  
2. JobLauncher launcher = (JobLauncher) context.getBean("jobLauncher");  
3. Job job = (Job) context.getBean("billJob");  
4. try {  
5.     JobExecution result = launcher.run(job, new JobParameters());  
6.     System.out.println(result.toString());  
7. } catch (Exception e) {  
8.     e.printStackTrace();  
9. }
```

该程序说明如下。

1 行：初始化 Spring 上下文，传入任务定义文件 job.xml。

2 行：获取作业调度器，根据名字为"jobLauncher"的 Bean 从 Spring 上下文获取。

3 行：获取名字为"billJob"的任务对象。

5 行：通过 jobLauncher 的 run 方法执行 billJob 任务。

执行上面后，读者可以查看输出文件 outputFile.csv 内容，所有的信用卡消费记录已经全部写入到 outputFile.csv 文件中，具体内容参见代码清单 2-10。

代码清单 2-10 Java 调用写入后的 outputFile.csv 文件

```
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
```

读者仔细观察控制台，会有如下的信息在控制台输出，具体内容参见代码清单 2-11。

代码清单 2-11 Java 调用的控台输出

```
accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
12:00:08;address=Lu Jia Zui road
accountID=4047390012345678;name=tom;amount=320.0;date=2013-2-3
10:35:21;address=Lu Jia Zui road
accountID=4047390012345678;name=tom;amount=674.7;date=2013-2-6
16:26:49;address=South Linyi road
accountID=4047390012345678;name=tom;amount=793.2;date=2013-2-9
15:15:37;address=Longyang road
accountID=4047390012345678;name=tom;amount=360.0;date=2013-2-11
11:12:38;address=Longyang road
accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28
20:34:19;address=Hunan road
JobExecution: id=0, version=2, startTime=Thu Feb 28 21:31:34 CST 2013,
endTime=Thu Feb 28 21:31:34 CST 2013, lastUpdated=Thu Feb 28 21:31:34 CST 2013,
status=COMPLETED, exitStatus=exitCode=COMPLETED;exitDescription=,
job=[JobInstance: id=0, version=0, JobParameters=[{}], Job=[billJob]]
```

2.5.2 JUnit 单元测试

JUnit 单元测试的一些代码如下。

代码清单 2-12 JUnit 调用批处理作业

```
1.  @RunWith(SpringJUnit4ClassRunner.class)
2.  @ContextConfiguration(locations={"/ch02/job/job.xml"})
3.  public class JobLaunchTest {
4.      @Autowired
5.      private JobLauncher jobLauncher;
6.      @Autowired@Qualifier("billJob")
7.      private Job job;
8.
9.      @Before
10.     public void setUp() throws Exception {
11.     }
12.
13.      @After
14.     public void tearDown() throws Exception {
15.     }
16.
17.      @Test
18.     public void importProducts() throws Exception {
19.         JobExecution result = jobLauncher.run(job, new JobParameters());
20.         System.out.println(result.toString());
21.     }
22. }
```

该程序说明如下。

1 行: @RunWith(SpringJUnit4ClassRunner.class) 表示该测试用例是运用 junit4 进行测试

2 行: @ContextConfiguration(locations={"ch02/job/job.xml"}) 表示设置 Spring 上下文加载的文件路径, 这里加载"ch01/job/job.xml"文件, 也可以写成 classpath:xxx.xml 的格式。

4~5 行: 使用@Autowired 自动装配 jobLauncher 实例, Spring 上下文自动使用配置文件中定义的 bean id 为"jobLauncher"的对象。

6~7 行: 使用@Qualifier("billJob")手工指定装配对象的 bean id 值。

9~11 行: @Before 表示在单元测试用例执行前调用的方法。

13~15 行: @After 表示在单元测试用例执行后调用的方法。

17~20 行: @Test 表示该方法是单元测试用例, 本测试用例中使用 jobLauncher 调用批处理作业任务。

执行单元测试, 执行成功后的效果参见图 2-3。

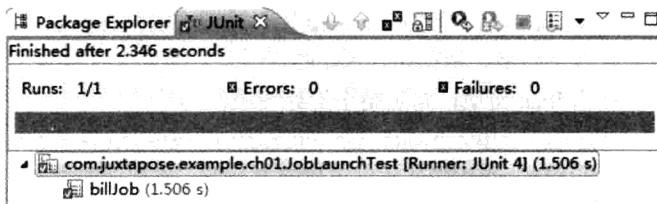


图 2-3 JUnit 执行批处理任务效果图

执行单元测试后, 读者可以查看输出文件 outputFile.csv 内容, 所有的信用卡消费记录已经全部写入到 outputFile.csv 文件中, 具体内容参见代码清单 2-13。

代码清单 2-13 JUnit 调用写入后的 outputFile.csv 文件

```
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
```

读者仔细观察控制台, 会有如下的信息在控制台输出, 具体内容参见代码清单 2-14。

代码清单 2-14 JUnit 调用的控台输出

```
accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
12:00:08;address=Lu Jia Zui road
accountID=4047390012345678;name=tom;amount=320.0;date=2013-2-3
10:35:21;address=Lu Jia Zui road
accountID=4047390012345678;name=tom;amount=674.7;date=2013-2-6
16:26:49;address=South Linyi road
accountID=4047390012345678;name=tom;amount=793.2;date=2013-2-9
```

```
15:15:37;address=Longyang road
accountID=4047390012345678;name=tom;amount=360.0;date=2013-2-11
11:12:38;address=Longyang road
accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28
20:34:19;address=Hunan road
JobExecution: id=0, version=2, startTime=Thu Feb 28 21:31:34 CST 2013,
endTime=Thu Feb 28 21:31:34 CST 2013, lastUpdated=Thu Feb 28 21:31:34 CST 2013,
status=COMPLETED, exitStatus=exitCode=COMPLETED;exitDescription=,
job=[JobInstance: id=0, version=0, JobParameters=[{}], Job=[billJob]]
```

2.6 概念预览

基于 Spring Batch 框架可以快速配置出复杂的批处理应用。我们和读者一起总结一下截止目前我们提到的 Spring Batch 中的一些基本概念。

Job Repository: 作业仓库，负责 Job、Step 执行过程中的状态保存。

Job launcher: 作业调度器，提供执行 Job 的入口。

Job: 作业，由多个 Step 组成，封装整个批处理操作。

Step: 作业步，Job 的一个执行环节，由多个或者一个 Step 组装成 Job。

Tasklet: Step 中具体执行逻辑的操作，可以重复执行，可以设置具体的同步、异步操作等。

Chunk: 给定数量 Item 的集合，可以定义对 Chunk 的读操作、处理操作、写操作，提交间隔等，这是 Spring Batch 框架的一个重要特性。

Item: 一条数据记录。

ItemReader: 从数据源（文件系统、数据库、队列等）中读取 Item。

ItemProcessor: 在 Item 写入数据源之前，对数据进行处理（如：数据清洗，数据转换，数据过滤，数据校验等）。

ItemWriter: 将 Item 批量写入数据源（文件系统、数据库、队列等）。

本节我们对这些概念有个基本了解，下面的章节我们将详细描述这些概念。

第 2 篇 基本篇

本篇重点讲述了批处理的核心概念、典型的作业配置、作业步配置，以及 Spring Batch 框架中经典的三步走策略：数据读、数据处理、数据写。详细地介绍了如何对分隔符类型文件、定长类型文件、JSON 格式文件、复杂类型格式文件、XML 文件、数据库、JMS 消息队列中的数据进行读操作、处理、写操作，对于数据库的操作详细介绍了使用 JDBC、Hibernate、存储过程、JPA、Ibatis 等处理。本篇包含六个章节。

第 3 章：介绍 Spring Batch 的基本概念，主要包括命名空间、作业 Job、作业步 Step、执行上下文 Execution Context、作业仓库 Job Repository、作业调度器 Job Launcher、条目读 Item Reader、条目处理 Item Processor、条目写 Item Writer 等。

第 4 章：带领读者配置作业 Job，包括作业的重启、作业拦截器、作业参数校验、抽象作业、作业继承、作业步 Scope、属性后绑定技术，最后向读者介绍了如何在定时任务、命令行、Web 应用中执行作业 Job、停止作业 Job。

第 5 章：带领读者配置作业步 Step，包括作业步抽象、继承、拦截器、重启、提交间隔、异常跳过、重试、完成策略，最后向读者介绍了作业步的事物和拦截器。

第 6 章：介绍读数据 ItemReader，主要包括对 CSV 格式文件、XML 格式文件、JSON 格式文件、多文件、DB、JMS 队列等各种资源的读取组件，最后向读者介绍了服务复用、自定义 ItemReader、读拦截器等功能。

第 7 章：介绍写数据 ItemWriter，主要包括对 CSV 格式文件、XML 格式文件、JSON 格式文件、多文件、DB、JMS 队列、发送邮件等各种资源的写组件，最后向读者介绍了服务复用、自定义 ItemWriter、写拦截器等功能。

第 8 章：介绍数据处理 ItemProcessor，主要包括数据转换、数据过滤、数据校验、组合数据处理，最后向读者介绍了服务复用、处理拦截器。

Spring Batch 基本概念

Spring Batch 框架采用了埃森哲提供的经典的批处理框架模型，该框架模型在包括 COBOL、C++、C#、Java 等经典平台得到十多年的验证。图 3-1 是该经典批处理框架的架构图。

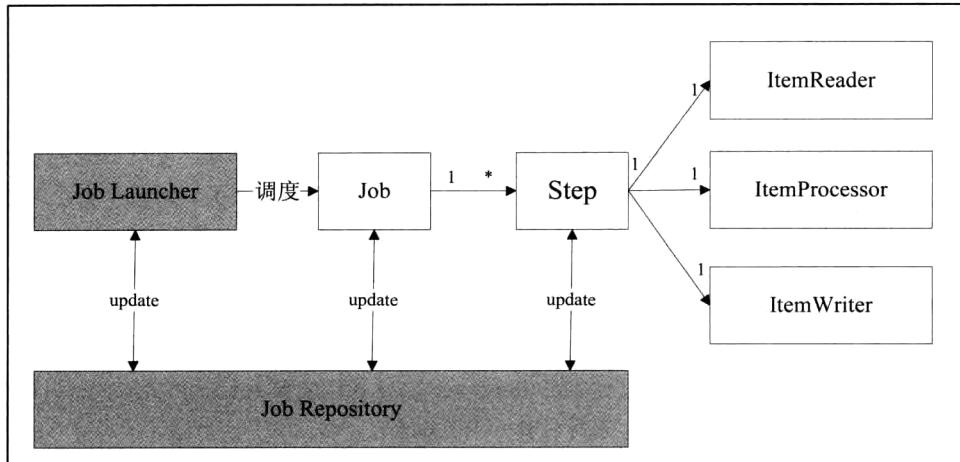


图 3-1 Spring Batch 批处理框架架构图

每个作业 Job 有 1 个或者多个作业步 Step; 每个 Step 对应一个 ItemReader、ItemProcessor、ItemWriter; 通过 Job Launcher 可以启动 Job, 启动 Job 时需要从 JobRepository 获取存在的 Job Execution; 当前运行的 Job 及 Step 的结果及状态会保存在 JobRepository 中。

表 3-1 列出了 Spring Batch 中用到的主要领域对象。

表 3-1 Spring Batch 主要领域对象

领域对象	描述
Job	作业。批处理中的核心概念，是 Batch 操作的基础单元
Job Instance	作业实例。每个作业执行时，都会生成一个实例，实例会被存放在 JobRepository 中，如果作业失败，下次重新执行该作业的时候，会使用同一个作业实例；对于 Job 和 Job Instance 的关系，大家可以想象为 Java 类定义和 Java 对象实例的关系
Job Parameters	作业参数。它是一组用来启动批处理任务的参数，在启动 Job 时候，可以设置任何需要的作业参数，需要注意作业参数会用来标识作业实例，即不同的 Job 实例是通过 Job 参数来区分的

续表

领域对象	描述
Job Execution	作业执行器。其负责具体 Job 的执行，每次运行 Job 都会启动一个新的 Job 执行器
Job Repository	作业仓库。其负责存储作业执行过程中的状态数据及结果，为 JobLauncher、Job、Step 提供标准的 CRUD 实现
Job Launcher	作业调度器。它根据给定的 JobParameters 执行作业
Step	作业步。Job 的一个执行环节，多个或者一个 Step 组装成 Job，封装了批处理任务中的一个独立的连续阶段
Step Execution	作业步执行器。它负责具体 Step 的执行，每次运行 Step 都会启动一个新的执行器
Tasklet	Tasklet。Step 中具体执行逻辑的操作，可以重复执行，可以设置具体的同步、异步操作等
Execution Context	执行上下文。它是一组框架持久化与控制的 key/value 对，能够让开发者在 Step Execution 或 Job Execution 范畴保存需要进行持久化的状态
Item	条目。一条数据记录
Chunk	Item 集合。它给定数量 Item 的集合，可以定义对 Chunk 的读操作、处理操作、写操作，提交间隔等
Item Reader	条目读。其表示 step 读取数据，一次读取一条
Item Processor	条目处理。用于表示 item 的业务处理
Item Writer	条目写。用于表示 step 输出数据，一次输出一批

本章为了解释清楚概念，会继续使用 HelloWorld 章节的示例。在原有示例的基础上，做了部分调整，具体代码参见第 3 章中的内容。

3.1 命名空间

Spring Batch 2.X 版本中增加了批处理的命名空间，简化了配置操作，对应的 XSD 可以通过 <http://www.springframework.org/schema/batch/spring-batch-2.2.xsd> 访问获取。通过引用命名空间，在进行 Job 配置的时候简化了配置文件的复杂度。

代码清单 3-1 给出了命名空间的配置信息。

代码清单 3-1 Spring Batch 2.X 提供的命名空间

```

1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <bean:beans xmlns="http://www.springframework.org/schema/batch"
3.      xmlns:bean="http://www.springframework.org/schema/beans"
4.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.      xmlns:p="http://www.springframework.org/schema/p"
6.      xmlns:tx="http://www.springframework.org/schema/tx"
7.      xmlns:aop="http://www.springframework.org/schema/aop"
```

```

8.      xmlns:context="http://www.springframework.org/schema/context"
9.      xsi:schemaLocation="http://www.springframework.org/schema/beans
10.     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
11.     http://www.springframework.org/schema/tx
12.     http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
13.     http://www.springframework.org/schema/aop
14.     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
15.     http://www.springframework.org/schema/context
16.     http://www.springframework.org/schema/context/spring-context-2.5.xsd
17.     http://www.springframework.org/schema/batch
18.     http://www.springframework.org/schema/batch/spring-batch-2.2.xsd">

```

具体代码参见源代码文件: /spring-batch-example/src/main/resources/ch03/job/job.xml。

3.2 Job

批处理作业 Job 由一组 Step 组成, 同时是作业配置文件的顶层元素。每个作业有自己的名字、可以定义 Step 执行的顺序, 以及定义作业是否可以重启。Job 的主要属性参见图 3-2。

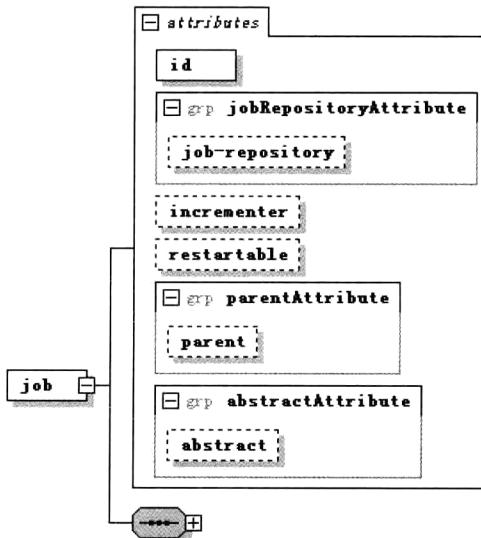


图 3-2 Job 元素的 Schema 定义

具体的属性定义参见第 4 章进行的描述。

Job 执行的时候会生成一个 Job Instance (作业实例), Job Instance 包含执行 Job 期间产生的数据以及 Job 执行的状态信息等; Job Instance 通过 Job Name(作业名称)和 Job Parameter(作业参数)来区分; 每次 Job 执行的时候都有一个 Job Execution(作业执行器), Job Execution 负责具体 job 的执行。Job、Job Instance 和 Job Execution 三者的关系参见图 3-3。

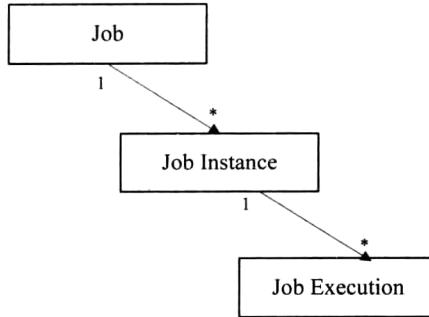


图 3-3 Job、Job Instance 和 Job Execution 三者关系图

一个 Job 可能有一到多个 Job Instance。

一个 Job Instance 可能有一到多个 Job Execution。

3.2.1 Job Instance

Job Instance(作业实例)是一个运行期的概念, Job 每执行一次都会涉及一个 Job Instance。Job Instance 来源可能有两种: 一种是根据设置的 Job Parameters 从 Job Repository(作业仓库)中获取一个; 如果根据 Job Parameters 从 Job Repository 没有获取 Job Instance, 则新创建一个新的 Job Instance。

为了更好地了解 Job Instance, 我们仍然使用第 2 章中用到的对账单的 Job。这里和第 2 章的变化在于将 Job Repository 由内存转到存放在数据库中, 具体的配置信息请参见 3.5.3 章节的配置。

说明: 在执行下面的示例前, 需要根据 Spring Batch 框架提供的数据库脚本完成数据库的初始化, 数据库脚本位置存放在 spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core 目录下, 针对不同的数据库提供了不同的脚本, 在运行本书的示例代码前, 需要初始化 schema-mysql.sql 文件。

在执行对账单 Job 的时候, 增加了参数配置信息, 具体参见代码清单 3-2。

代码清单 3-2 以参数方式执行 Job

```

1. public static void executeJob(String jobPath, String jobName,
2.                               JobParametersBuilder builder) {
3.     ApplicationContext context = new ClassPathXmlApplicationContext(
4.         jobPath);
5.     JobLauncher launcher = (JobLauncher) context.getBean("jobLauncher");
6.     Job job = (Job) context.getBean(jobName);
7.     try {
8.         JobExecution result = launcher.run(job, builder.toJobParameters());
9.         System.out.println(result.toString());
10.    } catch (Exception e) {
11.    }
12. }

```

```

10.           e.printStackTrace();
11.       }
12.   }
13.
14.   /**
15.    * @param args
16.    */
17.   public static void main(String[] args) {
18.       executeJob("ch03/job/job.xml", "billJob",
19.                   new JobParametersBuilder().addString("date", "20130308"));
20.   }

```

该程序说明如下。

7 行：调用 Job 的时候增加了 JobParameters，根据配置的 JobParameters 可以唯一地区分不同的 Job Instance。

19 行：设置调用 Job 时候的参数名称为“date”，参数值为“20130308”，参数的数据类型为“String”类型

为了展示 Job Instance 和 Job Execution 及 Job 的关系，请读者按照下面的操作步骤执行示例代码：

- (1) 执行类 com.juxtapose.example.ch03.JobLaunch，第一次确保参数为“20130308”；
- (2) 将文件 ch03\data\credit-card-bill-201303-bad.csv 中的内容替换到文件 ch03\data\credit-card-bill-201303.csv 中，以参数“20130309”执行类 com.juxtapose.example.ch03.JobLaunch；
- (3) 恢复文件 ch03\data\credit-card-bill-201303.csv 内容，以参数“20130309”再次执行类 com.juxtapose.example.ch03.JobLaunch。

经过三次执行后，可以到数据库中看出，一共有两个 Job Instance 和三个 Job Execution。Job Instance 对应的数据库表为 BATCH_JOB_INSTANCE，Job Execution 对应的数据库表为 BATCH_JOB_EXECUTION。

图 3-4 展示了两个 Job Instance，编号分别是 1 和 2。

	JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
1		1	0 billJob	2de7ecd6c3ef4d75cf6ec6013f576c80
2		2	0 billJob	ee1c173b88cdb2f97654eac50bacc51d

图 3-4 Job Instance 所在表 BATCH_JOB_INSTANCE 的数据

图 3-5 展示了三个 Job Execution，编号分别是 1、2、3，对应的 Job Instance 分别是 1、2、2。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	STATUS	CREATE_TIME
1		1	2	1 COMPLETED	2013-03-07 21:46:18
2		2	2	2 FAILED	2013-03-07 21:47:01
3		3	2	2 COMPLETED	2013-03-07 21:47:29

图 3-5 Job Execution 所在表 BATCH_JOB_EXECUTION 的数据

执行三次产生的 Job Instance 和 Job Execution，参见图 3-6。

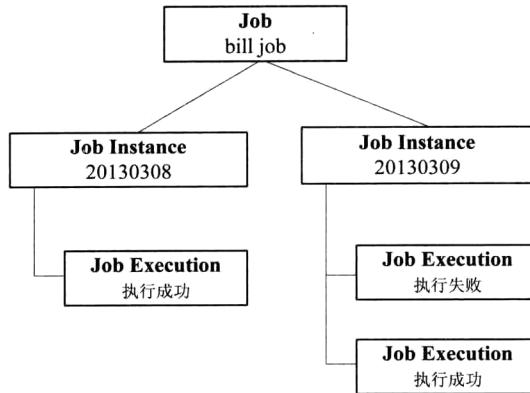


图 3-6 Job、Job Instance 和 Job Execution 三者示例图

以参数 date=20130308 执行了一次 billJob，执行成功；以参数 date=20130309 第一次执行 billJob 失败，重新以参数 date=20130309 第二次执行作业 billJob，执行成功。

Job Instance 对应的数据库表是：BATCH_JOB_INSTANCE。

具体表的字段信息参见 3.5.4.1 节。

Job Instance 对应的 Java 类是 org.springframework.batch.core.JobInstance。

小结：

- 第一次执行 Job 时候，会创建一个新的 Job Instance 和新的 Job Execution；
- 每次执行 Job 的时候，都会创建一个新的 Job Execution；
- 已经完成的 Job Instance，不能被重新执行；
- 在同一时刻，只能有一个 Job Execution 可以执行同一个 Job Instance。

3.2.2 Job Parameters

Job 通过 Job Parameters 来区分不同的 Job Instance。简单地说 Job Name + Job Parameters 来唯一地确定一个 Job Instance。如果 Job Name 一样，则 Job Parameters 肯定不一样；但是对于不同的 Job 来说，允许有相同的 Job Parameters。Job Instance、Job Name 和 Job Parameters 三个关系参见图 3-7。

$$\text{Job Instance} = \text{Job Name} + \text{Job Parameters}$$

图 3-7 Job Instance、Job Name 和 Job Parameters 三个关系

3.2.1 节执行了 3 次 Job，共使用了两个不同的 Job Parameters，具体参见图 3-8。

Job Parameters 共支持四种类型的参数，数据类型分别是 String、Date、Long、Double。同时 Spring Batch 框架提供了 JobParametersBuilder 来构建参数，JobParametersBuilder 提供的构造参数参见图 3-9。

	JOB_EXECUTION_ID	TYPE_CD	KEY_NAME	STRING_VAL
1		1 STRING	date	20130308
2		2 STRING	date	20130309
3		3 STRING	date	20130309

图 3-8 Job Parameter 所在表 BATCH_JOB_EXECUTION_PARAMS 的数据

-  JobParametersBuilder
 - addString(String, String) : JobParametersBuilder
 - addDate(String, Date) : JobParametersBuilder
 - addLong(String, Long) : JobParametersBuilder
 - addDouble(String, Double) : JobParametersBuilder
 - addParameter(String, JobParameter) : JobParametersBuilder

图 3-9 JobParametersBuilder 提供构建 Job Parameters 的操作

JobParameters 对应的数据库表是：BATCH_JOB_EXECUTION_PARAMS，表信息的具体描述参见 3.5.4.2 节。

JobParameters 对应的 Java 类是：org.springframework.batch.core.JobParameters。

Job Name 和相同的参数一样的情况下执行 Job，会发生错误。3.2.1 节中以参数为“20130308”的 billJob 已经执行过一次且成功，下面以参数“20130308”再次执行 billJob。操作步骤如下：

- (1) 修改类 com.juxtapose.example.ch03.JobLaunch 中的参数值为“20130308”；
- (2) 运行类 com.juxtapose.example.ch03.JobLaunch。

控制台会出现代码清单 3-3 的错误。

代码清单 3-3 相同 Job Name、Job Parameters 执行 Job 错误信息

```
org.springframework.batch.core.repository.JobInstanceAlreadyCompleteException: A job instance already exists and is complete for parameters={date=20130308}. If you want to run this job again, change the parameters.
  at org.springframework.batch.core.repository.support.SimpleJobRepository.createJobExecution(SimpleJobRepository.java:122)
```

该错误信息表示已经完成的 Job Instance 不能被再次执行，因为传入的参数相同，第二次执行 billJob 没有创建新的 Job Instance，而是根据参数“20130308”从数据库中读出上次已经完成的 Job Instance 来执行，导致出现上面的错误。

3.2.3 Job Execution

Job Execution 表示 Job 执行的句柄，根据上面的描述可知，一次 Job 的执行可能成功也可能失败。只有 Job Execution 执行成功后，对应的 Job Instance 才会被完成。根据上面例子对 billJob 的执行，参数为“20130308”的任务一次执行成功；参数为“20130309”的任务第一次执行失败，第二次执行成功。参数为“20130309”的任务共执行两次，共有两个 Job Execution 被创建，但是它们都对应同一个 Job Instance。

通过章节 3.2.1 中执行后，对应的 Job Execution 信息参见图 3-10。

表 BATCH_JOB_EXECUTION 中的数据如下。

	JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	STATUS	CREATE_TIME
1		1	2	1 COMPLETED	2013-03-07 21:46:18
2		2	2	2 FAILED	2013-03-07 21:47:01
3		3	2	2 COMPLETED	2013-03-07 21:47:29

图 3-10 Job Execution 对应的数据表 BATCH_JOB_EXECUTION 的数据

Job Execution 对应的数据库表是：BATCH_JOB_EXECUTION，具体表的字段信息参见 3.5.4.3 节。

Job Execution 对应的 Java 类是 org.springframework.batch.core.JobExecution。

Job Execution 主要的属性描述信息参见表 3-2。

表 3-2 Job Execution 关键属性说明

属性	说 明
status	BatchStatus 对象表示执行状态。BatchStatus.STARTED 表示运行时，BatchStatus.FAILED 表示执行失败，BatchStatus.COMPLETED 表示任务成功结束
startTime	表示任务开始时的系统时间；类型 java.util.Date
endTime	表示任务结束时的系统时间；类型 java.util.Date
exitStatus	ExitStatus 表示任务的运行结果，包含返回给调用者的退出代码
createTime	表示 Job Execution 第一次持久化时的系统时间。一个任务可能还没有启动（也就没有 startTime），但总是会有 createTime；类型 java.util.Date
lastUpdated	表示最近一次 Job Execution 被持久化的系统时间；类型 java.util.Date
executionContext	包含运行过程中所有需要被持久化的用户数据
failureException	在任务执行过程中例外的列表

3.3 Step

Step 表示作业中的一个完整步骤，一个 Job 可以由一个或者多个 Step 组成。Step 包含一个实际运行的批处理任务中的所有必需的信息，Step 可以是非常简单的业务实现，比如打印 HelloWorld 信息，也可以是非常复杂的业务处理，Step 的复杂程度通常是由业务决定的。

Step 与 Job 的关系参见图 3-11。

一个 Job 可以拥有一到多个 Step；一个 Step 可以有一到多个 Step Execution（当一个 Step 执行失败，下次重新执行该任务的时候，会为该 Step 重新生成一个 Step Execution）；一个 Job Execution 可以有一到多个 Step Execution（当一个 Job 由多个 Step 组成时，每个 Step 执行都会生成一个新的 Step Execution，则一个 Job Execution 会拥有多个 Step Execution）。

Step 中可以配置 tasklet、partition、job、flow，具体每个的作用后面章节会描述，这里仅罗列一下，给读者一个初步的印象。具体参见图 3-12。

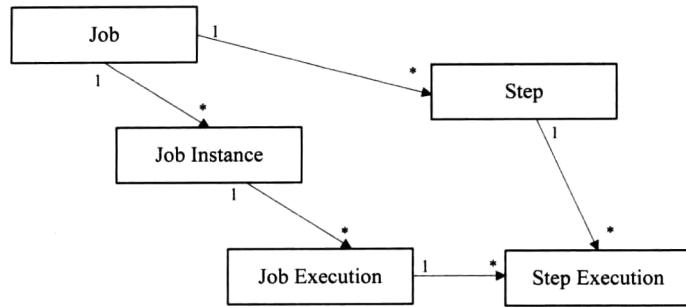


图 3-11 Job 与 Step 的关系

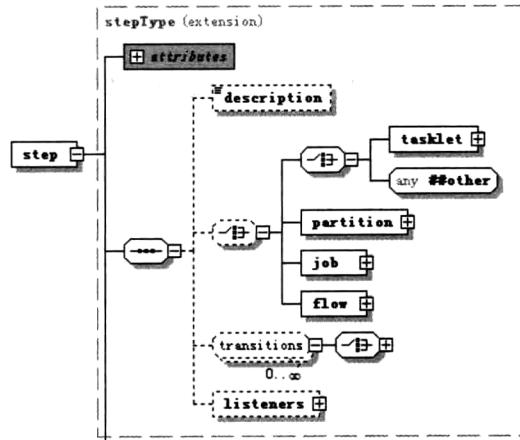


图 3-12 Step 元素的 Schema 定义

Step 的主要属性参见图 3-13。

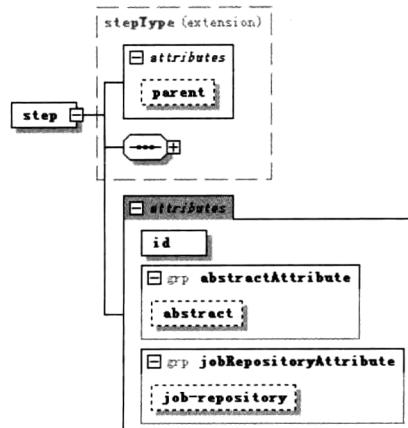


图 3-13 Step 元素的主要属性

具体的属性定义参见第 4 章的描述。

3.3.1 Step Execution

Step Execution 是 Step 执行的句柄。一次 Step 执行可能成功也可能失败。还记得 3.2.1 节我们执行的三次任务吗？我们的 billJob 只有一个 Step，但作业步执行器对应的表 BATCH_STEP_EXECUTION 却有三条记录。

表 BATCH_STEP_EXECUTION 中的记录参见图 3-14 和图 3-15。

	STEP_EXECUTION_ID	VERSION	STEP_NAME	JOB_EXECUTION_ID	STATUS
1		1	6 billStep		1 COMPLETED
2		2	3 billStep		2 FAILED
3		3	5 billStep		3 COMPLETED

图 3-14 Step Execution 对应的数据库表记录（一）

图 3-14 中的表数据如下。

START_TIME	END_TIME	COMMIT_COUNT	READ_COUNT	FILTER_COUNT
2013-03-07 21:46:18	2013-03-07 21:46:18	4	6	0
2013-03-07 21:47:01	2013-03-07 21:47:01	1	3	0
2013-03-07 21:47:29	2013-03-07 21:47:30	3	4	0

图 3-15 Step Execution 对应的数据库表记录（二）

具体描述如下：

第一条记录是执行参数为“20130308”的 billJob，唯一的 billStep 执行状态为完成；

第二条记录是执行参数为“20130309”的 billJob 且输入文件格式错误，导致本次的 billStep 执行状态为失败；

第三条记录是执行参数为“20130309”的 billJob 且输入文件格式正确，本次执行后，billStep 执行状态为成功。

Step Execution 对应的数据库表是：BATCH_STEP_EXECUTION。

具体表的字段信息参见 3.5.4.4 节。

Step Execution 对应的 Java 类是 org.springframework.batch.core.StepExecution。

Step 的执行过程是由 StepExecution 类的对象所表示的，包括每次执行所对应的 step、Job Execution、相关的事务操作（例如提交与回滚）、开始时间结束时间等。此外每次执行 step 时还包含一个 ExecutionContext，用来存放开发者在批处理运行过程中所需要的任何信息，例如用来重启的静态数据与状态数据。

表 3-3 列出了 StepExecution 的属性。

表 3-3 作业步执行器 StepExecution 属性说明

属性	说 明
status	BatchStatus 对象表示了执行状态。BatchStatus.STARTED 表示运行时，BatchStatus.FAILED 表示执行失败，BatchStatus.COMPLETED 表示任务成功结束

续表

属性	说 明
startTime	表示任务步开始时的系统时间；类型为 java.util.Date
endTime	表示任务步结束时的系统时间；类型为 java.util.Date
exitStatus	ExitStatus 表示任务步的运行结果，包含返回给调用者的退出代码
executionContext	在执行过程中任何需要进行持久化的用户数据
readCount	成功读取的记录数
writeCount	成功写入的记录数
commitCount	执行过程的事务中成功提交次数
rollbackCount	执行过程的事务中回滚次数
readSkipCount	读取失败而略过的记录数
processSkipCount	处理失败而略过的记录数
filterCount	被 ItemProcessor 过滤的记录数
writerSkipCount	写入失败而略过的记录数

3.4 Execution Context

Execution Context 是 Spring Batch 框架提供的持久化与控制的 key/value 对，能够让开发者在 Step Execution 或 Job Execution 中保存需要进行持久化的状态。我们在 3.2.1 节执行的三个作业，当以参数“20130309”且使用错误输入文件时，框架会每次 commit 后记录当前的提交记录数以及读的记录数，这样当 Step 发生错误时，下次重启 Job 会根据 Execution Context 中的数据恢复状态，保证继续从上次失败的点重新执行。

Step 的执行上下文数据存放在表 BATCH_STEP_EXECUTION_CONTEXT 中。

表 BATCH_STEP_EXECUTION_CONTEXT 中的记录参见图 3-16 和图 3-17。

STEP_EXECUTION_ID
1
2
3

```

1 {"map":{"entry":[{"string":"FlatFileItemWriter.current.count","long":366},
2 {"map":{"entry":[{"string":"FlatFileItemWriter.current.count","long":124},
3 {"map":{"entry":[{"string":"FlatFileItemWriter.current.count","long":366},

```

图 3-16 Execution Context 对应的数据库表记录（一）

SHORT_CONTEXT
{"string":"FlatFileItemWriter.written","long":6}, {"string":"FlatFileItemReader.read.count","int":7}])}
{"string":"FlatFileItemWriter.written","long":2}, {"string":"FlatFileItemReader.read.count","int":2}])}
{"string":"FlatFileItemWriter.written","long":4}, {"string":"FlatFileItemReader.read.count","int":7}])}

图 3-17 Execution Context 对应的数据库表记录（二）

图 3-16 的数据如下。

第一条记录：成功写了 6 条记录；

第二条记录：成功写了 2 条记录；读者可以看一下我们输入错误文件 ch03\data\credit-card-bill-201303-bad.csv 是第四行有错误，同时我们在 billJob 中定义的提交间隔属性 commit-interval="2"，当第四行读错误时候，只有前两条写成功；

第三条记录：成功写了 4 条记录；修复输入文件第四行错误后，重新执行该 Job，会从上次失败的点重新读取数据和写入数据（上次失败的地方是第 3 行，所以此次 Job 的执行从第三行开始），最终处理了 4 行数据。

Execution Context 分为两类：一类是 Job Execution 的上下文（对应表：BATCH_JOB_EXECUTION_CONTEXT）；另一类是 Step Execution 的上下文（对应表：BATCH_STEP_EXECUTION_CONTEXT）。两类上下文之间的关系：一个 Job Execution 对应一个 Job Execution 的上下文；每个 Step Execution 对应一个 Step Execution 上下文；同一个 Job 中的 Step Execution 共用 Job Execution 的上下文。因此如果同一个 Job 的不同 Step 间需要共享数据时，则可以通过 Job Execution 的上下文来共享数据。

3.5 Job Repository

Spring Batch 框架提供 Job Repository 来存储 Job 执行期的元数据（这里的元数据是指 Job Instance、Job Execution、Job Parameters、Step Execution、Execution Context 等数据），并提供两种默认实现。一种是存放在内存中（参见 2.3 节中 Job Repository 的定义）；另一种是将元数据存放在数据库中。通过将元数据存放在数据库中，可以随时监控批处理 Job 的执行状态，查看 Job 执行结果是成功还是失败，并且使得在 Job 失败的情况下重新启动 Job 成为可能。

Spring Batch 框架提供了可执行的数据库脚本和 JobRepository 的 CRUD 实现类。

(1) 数据库脚本位置

spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core

(2) 默认实现对元数据 CRUD 的操作类为：org.springframework.batch.core.repository.support.SimpleJobRepository。

本文中所有的例子使用 MySQL 数据库作为例子进行描述，读者根据需要可以使用其他的数据库。

说明：

Spring Batch 框架的 JobRepository 支持如下的数据库：DB2，Derby，H2，HSQLDB，MySQL，Oracle，PostgreSQL，SQLServer，Sybase。

3.5.1 Job Repository Schema

Job Repository 的 Schema 定义参见图 3-18。

Job Repository Schema 具体属性说明参见表 3-4。

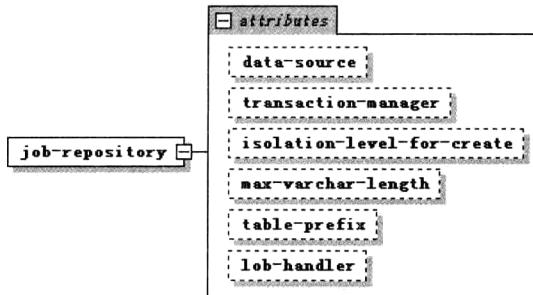


图 3-18 Job Repository 元素的 Schema 定义

表 3-4 作业仓库 Job Repository 属性说明

属性	说 明
data-source	数据源 默认引用 dataSource
transaction-manager	事务管理器，如果使用了 namespace，repository 会被自动加上事务控制，这是为了确保批处理操作元数据以及失败后重启的状态能够被准确地持久化，如果 repository 的方法不是事务控制的，那么框架的行为就不能够被准确地定义。 默认引用 transactionManager
isolation-level-for-create	创建 job execution 实体时候隔离级别，默认使用 SERIALIZABLE，通常使用 REPEATABLE_READ 也可以很好地工作。 默认值：SERIALIZABLE
max-varchar-length	数据库字段类型为 VARCHAR 时候允许的最大长度。 默认值：2500
table-prefix	表名前缀，JobRepository 可以修改元数据表的表前缀，但是表名和列名不能修改。 默认值：BATCH_
lob-handler	大字段列的处理方式，默认只有 Oracle 数据库，或者 Spring Batch 框架不支持的数据库类型需要配置

3.5.2 配置 Memory Job Repository

Job Repository 来存储 Job 执行期的元数据，并提供两种默认实现。一种是存放在内存中，默认实现类为 MapJobRepositoryFactoryBean，另一种是存入在数据库中（见 3.5.3 节）。

Job Repository 内存配置在 HelloWord2.3 节中已经给出，具体配置定义参见代码清单 3-4。

代码清单 3-4 Job Repository 内存配置定义

```

1.      <bean id="jobRepository"
2.          class="org.springframework.batch.core.repository.support.
3.          MapJobRepositoryFactoryBean">

```

通常在测试阶段可以使用基于内存的方式。

3.5.3 配置 DB Job Repository

Job Repository 用来存储 Job 执行期的元数据，另一种默认实现是存放在数据库中，通过将元数据存放在数据库中，可以随时监控批处理 Job 的执行状态，查看 Job 执行结果是成功还是失败，并且使得在 Job 失败的情况下重新启动 Job 成为可能。

说明：使用数据库的仓库时，需要首先根据 Spring Batch 框架提供的数据库脚本完成数据库的初始化，数据库脚本位置存放在 `spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core` 目录下，针对不同的数据库提供了不同的脚本，本书需要初始化文件：`schema-mysql.sql`。

Job Repository 数据库配置方式参见代码清单 3-5。

详细参见文件：`main/resources/ch03/job-context.xml`。

代码清单 3-5 Job Repository 数据库配置定义

```
1.      <job-repository id="jobRepository" data-source="dataSource"
2.          transaction-manager="transactionManager" isolation-level-for-
3.          create="SERIALIZABLE"
4.          table-prefix="BATCH_" max-varchar-length="1000"
5.      />
6.      <!-- 事务管理器 -->
7.      <bean:bean id="transactionManager"
8.          class="org.springframework.jdbc.datasource.DataSource
9.          TransactionManager">
10.         <bean:property name="dataSource" ref="dataSource" />
11.     </bean:bean>
12.     <!-- 数据源 -->
13.     <bean:bean id="dataSource"
14.         class="org.springframework.jdbc.datasource.DriverManager
15.         DataSource">
16.         <bean:property name="driverClassName">
17.             <bean:value>com.mysql.jdbc.Driver</bean:value>
18.         </bean:property>
19.         <bean:property name="url">
20.             <bean:value>jdbc:mysql://127.0.0.1:3306/test</bean:value>
21.         </bean:property>
22.         <bean:property name="username" value="root"></bean:property>
23.         <bean:property name="password" value="000000"></bean:property>
24.     </bean:bean>
```

该程序说明如下。

1~4 行：定义 JobRepository，属性 `data-source` 定义使用的数据源为 `dataSource`，属性

`transaction-manager` 定义使用的事务管理器，属性 `isolation-level-for-create` 定义创建 Job Execution 时候的事务隔离级别，避免多个 Job Execution 执行一个 Job Instance，属性 `table-prefix` 定义使用的数据库表的前缀为 `BATCH_`，属性 `max-varchar-length` 定义 varchar 的最大长度为 1000。

5~9 行：定义数据管理器。

11~22 行：定义数据源，本例中定义 MySQL 类型的数据源。

3.5.4 数据库 Schema

Spring Batch 框架进行元数据管理共有九张表，其中有三张表（后缀为 SEQ）是用来分配主键的。九张表分别是：

`BATCH_JOB_INSTANCE`、
`BATCH_JOB_EXECUTION`、
`BATCH_JOB_EXECUTION_PARAMS`、
`BATCH_STEP_EXECUTION`、
`BATCH_JOB_EXECUTION_CONTEXT`、
`BATCH_STEP_EXECUTION_CONTEXT`、
`BATCH_JOB_EXECUTION_SEQ`、
`BATCH_STEP_EXECUTION_SEQ`、
`BATCH_JOB_SEQ`

读者可以从 Spring Batch 发布物找到具体的数据库脚本。例如，如果需要查看 MySQL 数据库的创建脚本可以参考：`spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core\schema-mysql.sql`。对应的删除数据库脚本在如下位置：`spring-batch-core-2.2.1.RELEASE.jar\org\springframework\batch\core\schema-drop-mysql.sql`。

在本书中，如果没有特别说明，使用的均是 MySQL 数据库。

表之间的关系参见图 3-19。

Spring Batch 具体的描述参见表 3-5。

表 3-5 Spring Batch 框架核心表说明

表 名	说 明
<code>BATCH_JOB_INSTANCE</code>	作业实例表，用于存放 Job 的实例信息
<code>BATCH_JOB_EXECUTION_PARAMS</code>	作业参数表，用于存放每个 Job 执行时候的参数信息，该参数实际上是对应 Job 实例的
<code>BATCH_JOB_EXECUTION</code>	作业执行器表，用于存放当前作业的执行信息，比如创建时间，执行开始时间，执行结束时间，执行的那个 Job 实例，执行状态等
<code>BATCH_JOB_EXECUTION_CONTEXT</code>	作业执行上下文表，用于存放作业执行器上下文的信息。

续表

表 名	说 明
BATCH_STEP_EXECUTION	作业步执行器表，用于存放每个 Step 执行器的信息，比如作业步开始执行时间，执行完成时间，执行状态，读/写次数，跳过次数等信息
BATCH_STEP_EXECUTION_CONTEXT	作业步执行上下文表，用于存放每个作业步上下文的信息
BATCH_JOB_SEQ	作业序列表，用于给表 BATCH_JOB_INSTANCE 和 BATCH_JOB_EXECUTION_PARAMS 提供主键
BATCH_JOB_EXECUTION_SEQ	作业执行器序列表，用于给表 BATCH_JOB_EXECUTION 和 BATCH_JOB_EXECUTION_CONTEXT 提供主键
BATCH_STEP_EXECUTION_SEQ	作业步序列表，用于给表 BATCH_STEP_EXECUTION 和 BATCH_STEP_EXECUTION_CONTEXT 提供主键

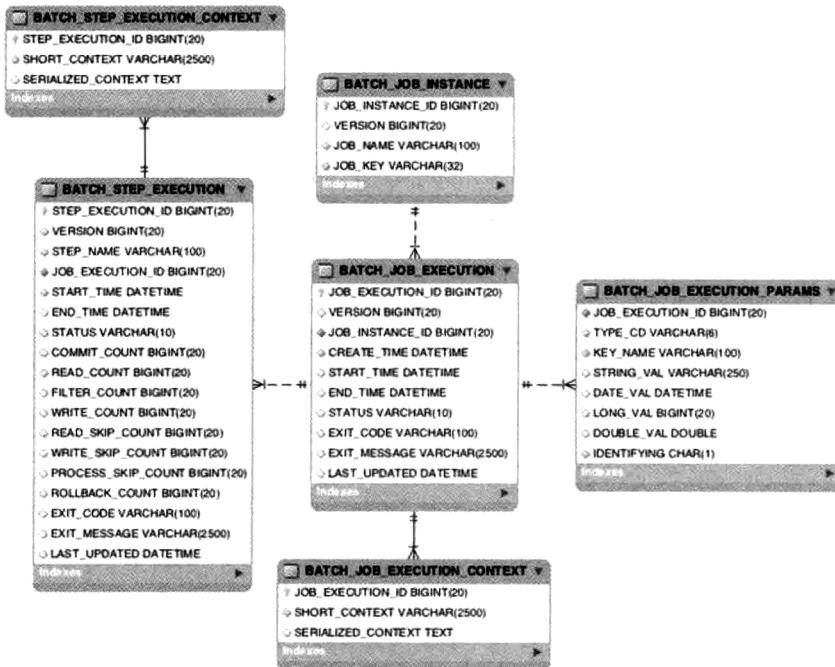


图 3-19 Spring Batch 框架核心数据库表关系图

接下来具体描述每张表的结构信息，以及字段的描述信息。

3.5.4.1 BATCH_JOB_INSTANCE

建表脚本参见代码清单 3-6。

代码清单 3-6 BATCH_JOB_INSTANCE 建表脚本

```
1. CREATE TABLE BATCH_JOB_INSTANCE (
2.     JOB_INSTANCE_ID BIGINT NOT NULL PRIMARY KEY ,
3.     VERSION BIGINT ,
4.     JOB_NAME VARCHAR(100) NOT NULL,
5.     JOB_KEY VARCHAR(32) NOT NULL,
6.     constraint JOB_INST_UN unique (JOB_NAME, JOB_KEY)
7. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_INSTANCE 字段信息描述参见表 3-6。

表 3-6 数据库表 BATCH_JOB_INSTANCE 字段说明

字 段	说 明
JOB_INSTANCE_ID	主键，作业实例 ID 编号，根据 BATCH_JOB_SEQ 自动生成
VERSION	版本号
JOB_NAME	作业名称，即在配置文件中定义的 Job id 字段内容
JOB_KEY	作业标识，根据作业参数序列化生成的标识；需要注意通过 JOB_NAME+ JOB_KEY 能够唯一区分一个作业实例。 如果是同一个 Job，则 JOB_KEY 一定不能相同，即作业参数不能相同；如果不是同一个 Job，JOB_KEY 则可以相同，即作业参数可以相同

3.5.4.2 BATCH_JOB_EXECUTION_PARAMS

建表脚本参见代码清单 3-7。

代码清单 3-7 BATCH_JOB_EXECUTION_PARAMS 建表脚本

```
1. CREATE TABLE BATCH_JOB_EXECUTION_PARAMS (
2.     JOB_EXECUTION_ID BIGINT NOT NULL ,
3.     TYPE_CD VARCHAR(6) NOT NULL ,
4.     KEY_NAME VARCHAR(100) NOT NULL ,
5.     STRING_VAL VARCHAR(250) ,
6.     DATE_VAL DATETIME DEFAULT NULL ,
7.     LONG_VAL BIGINT ,
8.     DOUBLE_VAL DOUBLE PRECISION ,
9.     IDENTIFYING CHAR(1) NOT NULL ,
10.    constraint JOB_EXEC_PARAMS_FK foreign key (JOB_EXECUTION_ID)
11.      references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
12. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_EXECUTION_PARAMS 字段信息描述参见表 3-7。

表 3-7 数据库表 BATCH_JOB_EXECUTION_PARAMS 字段说明

字 段	说 明
JOB_EXECUTION_ID	外键，作业执行器 ID 编号，一个作业实例可能会有多行参数记录，主要根据参数的个数决定

续表

字 段	说 明
TYPE_CD	参数的类型，可能是如下四种当中的一种：date、string、long、double
KEY_NAME	参数名字
STRING_VAL	参数如果是 string，此列存放 string 类型参数值
DATE_VAL	参数如果是 date，此列存放 date 类型参数值
LONG_VAL	参数如果是 long，此列存放 long 类型参数值
DOUBLE_VAL	参数如果是 double，此列存放 double 类型参数值
IDENTIFYING	用于标识作业参数是否标识作业实例

3.5.4.3 BATCH_JOB_EXECUTION

建表脚本参见代码清单 3-8。

代码清单 3-8 BATCH_JOB_EXECUTION 建表脚本

```
1. CREATE TABLE BATCH_JOB_EXECUTION (
2.     JOB_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY ,
3.     VERSION BIGINT ,
4.     JOB_INSTANCE_ID BIGINT NOT NULL,
5.     CREATE_TIME DATETIME NOT NULL,
6.     START_TIME DATETIME DEFAULT NULL ,
7.     END_TIME DATETIME DEFAULT NULL ,
8.     STATUS VARCHAR(10) ,
9.     EXIT_CODE VARCHAR(100) ,
10.    EXIT_MESSAGE VARCHAR(2500) ,
11.    LAST_UPDATED DATETIME,
12.    constraint JOB_INST_EXEC_FK foreign key (JOB_INSTANCE_ID)
13.      references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
14. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_EXECUTION 字段信息描述参见表 3-8。

表 3-8 数据库表 BATCH_JOB_EXECUTION 字段说明

字 段	说 明
JOB_EXECUTION_ID	主键，作业执行器 ID 编号
VERSION	版本号
JOB_INSTANCE_ID	作业实例 ID 编号
CREATE_TIME	作业执行器创建时间
START_TIME	作业执行器开始执行时间
END_TIME	作业执行器结束时间

续表

字 段	说 明
STATUS	作业执行器执行的状态, 如: COMPLETED, STARTING, STARTED, STOPPING, STOPPED, FAILED, ABANDONED, UNKNOWN。 这些状态在类 org.springframework.batch.core.BatchStatus 中定义
EXIT_CODE	作业执行器退出编码, 如: UNKNOWN, EXECUTING, COMPLETED, NOOP, FAILED, STOPPED。 这些状态在类 org.springframework.batch.core.ExitStatus 中定义
EXIT_MESSAGE	作业执行器退出描述, 详细描述退出的信息, 如果发生了异常, 通常包含异常的堆栈信息
LAST_UPDATED	本条记录上次更新时间

3.5.4.4 BATCH_STEP_EXECUTION

建表脚本参见代码清单 3-9。

代码清单 3-9 BATCH_STEP_EXECUTION 建表脚本

```

1. CREATE TABLE BATCH_STEP_EXECUTION (
2.     STEP_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY ,
3.     VERSION BIGINT NOT NULL,
4.     STEP_NAME VARCHAR(100) NOT NULL,
5.     JOB_EXECUTION_ID BIGINT NOT NULL,
6.     START_TIME DATETIME NOT NULL ,
7.     END_TIME DATETIME DEFAULT NULL ,
8.     STATUS VARCHAR(10) ,
9.     COMMIT_COUNT BIGINT ,
10.    READ_COUNT BIGINT ,
11.    FILTER_COUNT BIGINT ,
12.    WRITE_COUNT BIGINT ,
13.    READ_SKIP_COUNT BIGINT ,
14.    WRITE_SKIP_COUNT BIGINT ,
15.    PROCESS_SKIP_COUNT BIGINT ,
16.    ROLLBACK_COUNT BIGINT ,
17.    EXIT_CODE VARCHAR(100) ,
18.    EXIT_MESSAGE VARCHAR(2500) ,
19.    LAST_UPDATED DATETIME,
20.    constraint JOB_EXEC_STEP_FK foreign key (JOB_EXECUTION_ID)
21.        references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
22. ) ENGINE=InnoDB;

```

数据库表 BATCH_JOB_EXECUTION 字段信息描述参见表 3-9。

表 3-9 数据库表 BATCH_STEP_EXECUTION 字段说明

字段	说 明
STEP_EXECUTION_ID	主键，作业步实例 ID 编号
VERSION	版本号
STEP_NAME	作业步名字
JOB_EXECUTION_ID	外键，作业执行器 ID
START_TIME	作业步执行器开始执行时间
END_TIME	作业步执行器结束时间
STATUS	作业步执行器执行的状态，如：COMPLETED, STARTING, STARTED, STOPPING, STOPPED, FAILED, ABANDONED, UNKNOWN。 这些状态在类 org.springframework.batch.core.BatchStatus 中定义
COMMIT_COUNT	事务提交次数
READ_COUNT	读数据的次数
FILTER_COUNT	过滤掉的数据次数
WRITE_COUNT	写数据的次数
READ_SKIP_COUNT	读数据跳过的次数
WRITE_SKIP_COUNT	写数据跳过的次数
PROCESS_SKIP_COUNT	处理数据跳过的次数
ROLLBACK_COUNT	事务回滚次数
EXIT_CODE	作业步执行器退出编码，如：UNKNOWN, EXECUTING, COMPLETED, NOOP, FAILED, STOPPED。 这些状态在类 org.springframework.batch.core.ExitStatus 中定义
EXIT_MESSAGE	作业步执行器退出描述，详细描述退出的信息，如果发生了异常，通常包含异常的堆栈信息
LAST_UPDATED	本条记录上次更新时间

3.5.4.5 BATCH_JOB_EXECUTION_CONTEXT

建表脚本参见代码清单 3-10。

代码清单 3-10 BATCH_JOB_EXECUTION_CONTEXT 建表脚本

```

1. CREATE TABLE BATCH_JOB_EXECUTION_CONTEXT (
2.     JOB_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
3.     SHORT_CONTEXT VARCHAR(2500) NOT NULL,
4.     SERIALIZED_CONTEXT TEXT ,
5.     constraint JOB_EXEC_CTX_FK foreign key (JOB_EXECUTION_ID)
6.         references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
7. ) ENGINE=InnoDB;

```

数据库表 BATCH_JOB_EXECUTION_CONTEXT 字段信息描述参见表 3-10。

表 3-10 数据库表 BATCH_JOB_EXECUTION_CONTEXT 字段说明

字 段	说 明
JOB_EXECUTION_ID	外键, 作业执行器 ID 编号
SHORT_CONTEXT	作业执行器上下文字符串格式
SERIALIZED_CONTEXT	序列化的作业执行器上下文

3.5.4.6 BATCH_STEP_EXECUTION_CONTEXT

建表脚本参见代码清单 3-11。

代码清单 3-11 BATCH_STEP_EXECUTION_CONTEXT 建表脚本

```
1. CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT (
2.     STEP_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY,
3.     SHORT_CONTEXT VARCHAR(2500) NOT NULL,
4.     SERIALIZED_CONTEXT TEXT ,
5.     constraint STEP_EXEC_CTX_FK foreign key (STEP_EXECUTION_ID)
6.       references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
7. ) ENGINE=InnoDB;
```

数据库表 BATCH_JOB_EXECUTION_CONTEXT 字段信息描述参见表 3-11。

表 3-11 数据库表 BATCH_STEP_EXECUTION_CONTEXT 字段说明

字 段	说 明
STEP_EXECUTION_ID	外键, 作业步执行器 ID 编号
SHORT_CONTEXT	作业执行器上下文字符串格式
SERIALIZED_CONTEXT	序列化的作业执行器上下文

3.6 Job Launcher

Job Launcher（作业调度器）是 Spring Batch 框架基础设施层提供的运行 Job 的能力。通过给定的 Job 名称和作业参数 Job Parameters，可以通过 Job Launcher 执行 Job。通过 Job Launcher 可以在 Java 程序中调用批处理任务，也可以在通过命令行或者其他框架（如定时调度框架 Quartz）中调用批处理任务。Spring Batch 框架提供了 Job Launcher 的简单实现 SimpleJobLauncher。

JobLauncher 接口定义参见代码清单 3-12。

代码清单 3-12 JobLauncher 接口定义

```
1. public interface JobLauncher {
2.     public JobExecution run(Job job, JobParameters jobParameters) throws
3.             JobExecutionAlreadyRunningException,
4.             JobRestartException,
5.             JobInstanceAlreadyCompleteException,
```

```
6.                               JobParametersInvalidException;
7. }
```

该接口只有一个 `run` 方法，两个参数 `job` 和 `job Parameters`。批处理应用可以通过 `Job Launcher` 和外部系统交互，通常情况下外部系统可以同步也可以异步调用批处理应用；批处理应用本身可以访问外部的资源，例如数据库、文件、消息队列等。

3.7 ItemReader

`ItemReader` 是 `Step` 中对资源的读处理，`Spring Batch` 框架已经提供了多种类型的读实现，包括对文本文件、`XML` 文件、数据库、`JMS` 消息等读的处理。直接使用 `Spring Batch` 框架提供的读组件可以快速地完成批处理应用的开发和搭建。同时 `Spring Batch` 框架提供了较好的复用和扩展机制，可以复用现有的读服务或者快速实现自定义的读组件。

`Spring Batch` 框架提供的读组件列表参见表 3-12。

表 3-12 `Spring Batch` 框架提供的 `ItemReader` 组件

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

上面所有的组件均实现 `ItemReader` 接口，接口定义参见代码清单 3-13。

代码清单 3-13 ItemReader 接口定义

```
1. public interface ItemReader<T> {
2.     T read() throws Exception,
3.             UnexpectedInputException,
4.             ParseException,
5.             NonTransientResourceException;
6. }
```

read 方法负责从给定的资源中读取数据。

3.8 ItemProcessor

ItemProcessor 阶段表示对读取的数据进行处理，开发者可以实现自己的业务操作来对数据进行处理。

Spring Batch 框架提供的处理组件列表参见表 3-13。

表 3-13 Spring Batch 框架提供的 ItemProcessor 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

业务操作需要实现 ItemProcessor 接口，接口定义参见代码清单 3-14 ItemProcessor 接口定义。

代码清单 3-14 ItemProcessor 接口定义

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

process 方法中，参数 item 是 ItemReader 读取的数据，返回值 O 是交给 ItemWriter 写的数据，在 process 方法中可以修改读到的数据的值。如果返回值是 null，表示忽略这次的数据，具体描述参见 Item Processor 对应章节的描述。

3.9 ItemWriter

ItemWriter 是 Step 中对资源的写处理，Spring Batch 框架已经提供了多种类型的写实现，包括对文本文件、XML 文件、DB 等写的处理。直接使用 Spring Batch 框架提供的写组件可以快速地完成应用的开发和搭建。同时 Spring Batch 框架提供了较好的复用和扩展机制，可以

复用现有的写服务或者快速实现自定义的写组件。

Spring Batch 框架提供的写组件列表参见表 3-14。

表 3-14 Spring Batch 框架提供的 ItemWriter 组件

ItemWriter	说 明
FlatFileItemWriter	写 Flat 类型文件
MultiResourceItemWriter	多文件写组件
StaxEventItemWriter	写 XML 类型文件
AmqpItemWriter	写 AMQP 类型消息
ClassifierCompositeItemWriter	根据 Classifier 路由不同的 Item 到特定的 ItemWriter 处理
HibernateItemWriter	基于 Hibernate 方式写数据库
IbatisBatchItemWriter	基于 Ibatis 方式写数据库
ItemWriterAdapter	ItemWriter 适配器，可以复用现有的写服务
JdbcBatchItemWriter	基于 JDBC 方式写数据库
JmsItemWriter	写 JMS 队列
JpaItemWriter	基于 Jpa 方式写数据库
GemfireItemWriter	基于分布式数据库 Gemfire 的写组件
SpELMappingGemfireItemWriter	基于 Spring 表达式语言写分布式数据库 Gemfire 的组件
MimeMessageItemWriter	发送邮件的写组件
MongoItemWriter	基于分布式文件存储的数据库 MongoDB 写组件
Neo4jItemWriter	面向网络的数据库 Neo4j 的读组件
PropertyExtractingDelegatingItemWriter	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
RepositoryItemWriter	基于 Spring Data 的写组件
SimpleMailMessageItemWriter	发送邮件的写组件
CompositeItemWriter	条目写的组合模式，支持组装多个 ItemWriter

上面的组件均实现了 ItemWriter 接口，接口定义参见代码清单 3-15。

代码清单 3-15 ItemWriter 接口定义

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

write 方法负责将数据写入到给定的资源中。

第 4 章我们将详细描述如何配置 Job 和运行 Job。

配置作业 Job

本章详细描述如何配置 Job 及如何运行 Job。在配置 Job 部分，我们将配置重启特性、抽象特性、作业仓库、监听器、参数校验等功能。在运行 Job 部分我们将描述如何运行已经配置好的 Job。

第 3 章中我们描述了批处理的整体架构，在进入具体配置 Job 前，我们重新复习一下批处理的整体架构，参见图 4-1。一个 Job 由 1 个或者多个 Step 组成，Step 有读、处理、写三部分操作组成；Job 运行期所有的数据通过 Job Repository 进行持久化，同时通过 JobLauncher 负责调度 Job 作业。

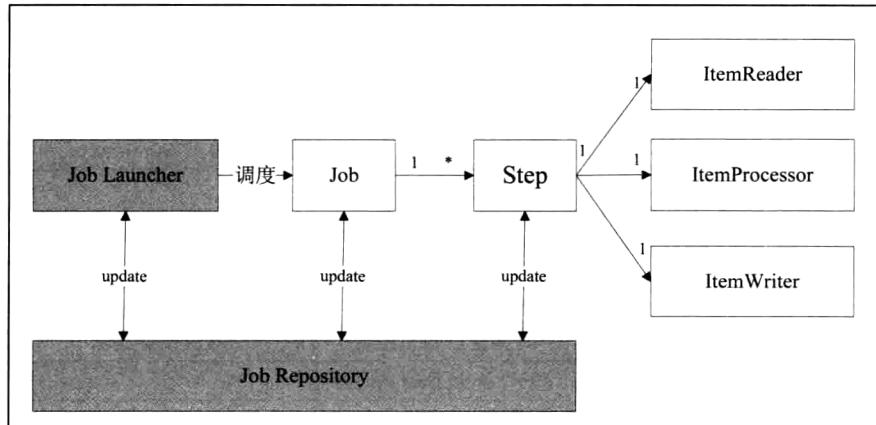


图 4-1 Spring Batch 批处理框架架构图

为了简化 Spring Batch 的配置，Spring Batch 框架提供了独立的 XML 配置，开发者无需再以 Spring Bean 的方式进行配置，提供了独立的 job、step、tasklet、chunk 标签。图 4-2 展示了 Spring Batch 框架提供的 XML 元素和属性，基于这些元素，开发者可以快速、高效地进行批处理应用的开发。

4.1 基本配置

在配置 Job 之前，我们一起看一下 Job 的主要属性定义和元素定义。

Job 主要属性包括 id（作业唯一标识）、job-repository（定义作业仓库）、incrementer（作业参数递增器）、restartable（作业是否可以重启）、parent（指定该作业的父作业）、abstract

(定义作业是否抽象的)。图 4-3 展示了 Job 属性的 Schema 的定义, 表 4-1 给出了 Job 属性的详细说明。

element	job
element	step
element	flow
element	job-listener
element	step-listener
element	job-repository
group	flowGroup
complexType	stepType
complexType	partitionType
complexType	taskletType
complexType	transactionAttributesType
simpleType	isolationType
group	beanElementGroup
complexType	chunkTaskletType
attributeGroup	nextAttribute
attributeGroup	exceptionClassAttribute
group	includeElementGroup
group	includeExcludeElementGroup
complexType	listenerType
complexType	jobExecutionListenerType
complexType	stepListenerType
complexType	stepListenersType
group	transitions
attributeGroup	jobRepositoryAttribute
attributeGroup	parentAttribute
attributeGroup	abstractAttribute
attributeGroup	mergeAttribute
attributeGroup	adapterMethodAttribute
simpleType	description

图 4-2 Spring Batch 框架提供的 XML 元素和属性

Job 主要子元素包括 step (定义作业步)、split (定义并行作业步)、flow (引用独立定义的作业流)、decision (定义作业步执行的条件判断器, 用于判断后续执行的作业步)、listeners (定义作业拦截器)、validator (定义作业参数校验器)。图 4-4 展示了 Job 子元素的 Schema 的定义, 表 4-2 给出了 Job 子元素的详细说明。

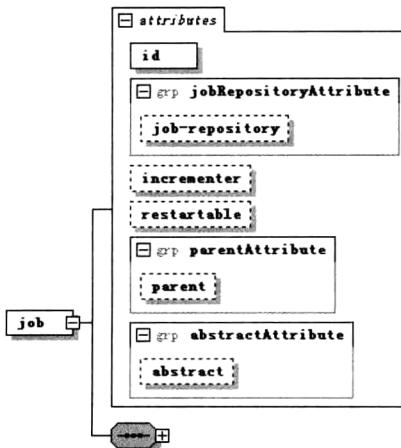


图 4-3 Job 属性 Schema 的定义

表 4-1 Job 属性说明

属性	说 明	默 认 值
id	Job 的唯一标识，在整个运行上下文中不允许重复	
job-repository	定义该 Job 运行期间使用的 Job 仓库，默认使用名字为 jobRepository 的 Bean	jobRepository
incrementer	作业参数递增器，只有在 org.springframework.batch.core.launch.JobOperator 的 startNextInstance 方法中使用	
restartable	定义当前作业是否支持重启，默认值是 true，表示支持重启，如果不需要重启，需要显示设置为 false	true
parent	定义当前 Job 的父 Job。Job 可以从其他 Job 继承。通常在父 Job 中定义共有的属性；在子 Job 中定义特有的属性	
abstract	定义当前 Job 是否是抽象的。True 表示当前 Job 是抽象的，不能被实例化	

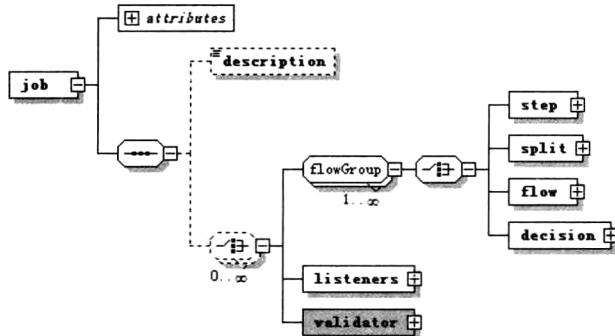


图 4-4 Job 子元素 Schema 的定义

表 4-2 Job 子元素说明

属性	说 明
step	定义 Job 的作业步
split	定义并行的 Step
flow	引用独立配置的作业步流程
decision	Step 执行的条件判断器，根据 decision 可以动态地决定后续执行的 Step
listeners	定义 Job 执行时的拦截器
validator	定义 JobParameters 的验证器

4.1.1 重启 Job

通过属性 restartable 可以定义 Job 是否可以重启。默认情况下 Job 是可以重启的，需要注意的是，即使配置了 Job 可以重新启动，仍需要保证 Job Instance 的状态一定不能为“COMPLETED”状态。代码清单 4-1 展示了如何配置 Job 不能重启：

代码清单 4-1 配置 Job 不能重启

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

该程序说明如下。

1 行：属性 `restartable` 的默认值为 `true`，表示支持重新启动；如果不需要支持重启，需要显示设该值为 `false`。

执行示例项目中的类 `test.com.juxtapose.example.ch04.JobLaunchRestart` 两次，第一次执行因为对应的输入文件格式错误，对应的 `Job Instance` 的状态是“FAILED”；第二次重新执行类 `test.com.juxtapose.example.ch04.JobLaunchRestart`，虽然对应的 `Job Instance` 的状态是“FAILED”，正常情况下可以重启 `Job`，但因为该 `Job` 配置为不可以重启，所以重新启动该 `Job` 会抛出异常 `org.springframework.batch.core.repository.JobRestartException`。

重新启动不支持重启的 `Job`，会得到代码清单 4-2 中的异常信息。

代码清单 4-2 重新启动无法重启 `Job` 异常信息

```
1. org.springframework.batch.core.repository.JobRestartException:
2.                               JobInstance already exists and is not restartable
3.   at org.springframework.batch.core.launch.support.SimpleJobLauncher.
4.         run(SimpleJobLauncher.java:97)
5.   at test.com.juxtapose.example.ch04.JobLaunchRestart.executeJob
6.     (JobLaunch.java:31)
6.   at test.com.juxtapose.example.ch04.JobLaunchRestart.main(JobLaunch.
java:42)
```

4.1.2 Job 拦截器

Spring Batch 框架在 `Job` 执行阶段提供了拦截器，使得在 `Job` 执行前后能够加入自定义的业务逻辑处理。`Job` 执行阶段拦截器需要实现接口：`org.springframework.batch.core.JobExecutionListener`。`JobExecutionListener` 接口参见代码清单 4-3。

代码清单 4-3 `JobExecutionListener` 接口定义

```
1. public interface JobExecutionListener {
2.   void beforeJob(JobExecution jobExecution);
3.   void afterJob(JobExecution jobExecution);
4. }
```

该程序说明如下。

2 行：表示在 Job 执行之前调用该方法。

3 行：表示在 Job 执行之后调用该方法。

通过属性 `listeners` 可以为 Job 指定拦截器列表，拦截器可以一个或者多个，如有多个，则拦截器执行顺序为配置文件中定义的顺序。代码清单 4-4 给出了作业 `billJob` 的拦截器配置。

代码清单 4-4 Job 拦截器配置

```
1.      <job id="billJob">
2.          <step id="billStep">
3.              .....
4.          </step>
5.          <listeners>
6.              <listener ref="sysoutListener"></listener>
7.          </listeners>
8.      </job>
9.      <bean:bean id="sysoutListener"
10.         class="com.juxtapose.example.ch04.listener.SystemOutJobExecution
11.             Listener">
12.         </bean:bean>
```

`billJob` 执行时，`listeners` 中定义的拦截器 `SystemOutJobExecutionListener` 会被执行，该拦截器仅做了打印输出。读者可以根据具体的业务需要实现自定义的作业拦截器。`SystemOutJobExecutionListener` 的实现参见示例项目代码 `com.juxtapose.example.ch04.listener.SystemOutJobExecutionListener`。

Spring Batch 框架默认提供 `JobExecutionListener` 的实现，`CompositeJobExecutionListener`（拦截器组合器，支持配置一组拦截器）、`JobExecutionListenerSupport`（拦截器空实现，让开发者快速实现关心的业务操作）分别完成不同的功能，表 4-3 给出了框架的默认实现。

表 4-3 `JobExecutionListener` 默认实现。

<code>JobExecutionListener</code> 默认实现	功能说明
<code>CompositeJobExecutionListener</code>	拦截器组合模式，支持一组拦截器调用
<code>JobExecutionListenerSupport</code>	<code>JobExecutionListener</code> 空实现，可以直接继承，仅覆写关心的方法

拦截器异常

拦截器方法如果抛出异常会影响 Job 的正常执行，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"。

执行顺序

在配置文件中可以配置多个 `listener`，拦截器之间的执行顺序按照 `listener` 定义的顺序执

行。before 方法按照 listener 定义的顺序执行，after 方法按照相反的顺序执行。代码清单 4-5 中执行顺序如下：

1. sysoutListener 拦截器的 before 方法；
2. sysoutAnnotationListener 拦截器的 before 方法；
3. sysoutAnnotationListener 拦截器的 after 方法；
4. sysoutListener 拦截器的 after 方法。

代码清单 4-5 Job 多拦截器配置

```
1.      <job id="billJob" restartable="false">
2.          <step id="billStep">
3.              .....
4.          </step>
5.          <listeners>
6.              <listener ref="sysoutListener"></listener>
7.              <listener ref="sysoutAnnotationListener"></listener>
8.          </listeners>
9.      </job>
```

Annotation

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

- @BeforeJob
- @AfterJob

使用 Annotation 声明的拦截器和实现接口 JobExecutionListener 的拦截器，对应的配置格式一致，只需要在 listeners 节点中声明即可。

代码清单 4-6 展示了通过 Annotation 声明的拦截器代码。

代码清单 4-6 通过 Annotation 定义 Job 拦截器

```
1. public class SystemOut {
2.     @BeforeJob
3.     public void beforeJob(JobExecution jobExecution) {
4.         System.out.println("Annotation: JobExecution creat time:" +
5.                             jobExecution.getCreateTime());
6.         //throw new RuntimeException("listener make error!");
7.     }
8.     @AfterJob
9.     public void afterJob(JobExecution jobExecution) {
10.        System.out.println("Annotation: Job execute state:" +
11.                            jobExecution.getStatus().toString());
12.    }
13. }
```

该程序的说明如下。

2 行：@BeforeJob 声明作业执行前的操作。

8 行：@AfterJob 声明作业执行后的操作。

4.1.3 Job Parameters 校验

Spring Batch 框架提供了 Job 作业参数的校验功能，在 3.2.2 节中给出了 Job Parameters 详细介绍，读者如有遗忘，可以重新阅读该节的内容。开发者可以自定义实现参数校验器（需要实现接口：org.springframework.batch.core.JobParametersValidator）或者使用框架已经提供的校验组件类。通过参数校验，可以保证 Job 执行可以按照给定的参数执行，避免出现不必要的逻辑错误。

Spring Batch 框架默认提供了 JobParametersValidator 的实现，CompositeJobParameters Validator（参数校验组合器，支持配置一组参数校验类）、DefaultJobParametersValidator（参数校验默认实现类，支持必输校验和可选校验）分别完成不同功能，表 4-4 给出了 Spring Batch 框架提供的参数校验组件。

表 4-4 JobParametersValidator 默认实现

JobParametersValidator 默认实现	功能说明
CompositeJobParametersValidator	参数校验组合模式，支持一组参数校验
DefaultJobParametersValidator	参数校验默认实现，支持必须输入的参数和可以选择输入的参数

通过属性 validator 定义参数校验实现类，代码清单 4-7 展示了如何为参数配置参数校验。

代码清单 4-7 Job Parameters 配置

```
1.      <job id="billJob">
2.          <step id="billStep">
3.              .....
4.          </step>
5.          <listeners>
6.              <listener ref="sysoutListener"></listener>
7.              <listener ref="sysoutAnnotationListener"></listener>
8.          </listeners>
9.          <validator ref="validator"></validator>
10.     </job>
11.     <!-- 参数校验 -->
12.     <bean:bean id="validator"
13.                 class="org.springframework.batch.core.job.DefaultJob
14.                               ParametersValidator" >
15.         <bean:property name="requiredKeys" >
```

```
15.      <bean:set>
16.          <bean:value>date</bean:value>
17.      </bean:set>
18.  </bean:property>
19.  <bean:property name="optionalKeys" >
20.      <bean:set>
21.          <bean:value>name</bean:value>
22.      </bean:set>
23.  </bean:property>
24. </bean:bean>
```

该程序的说明如下。

9 行：通过属性 validator 定义参数校验使用的类对象。

14~18 行：通过关键字 requiredKeys 定义必须输入的参数 date。

19~23 行：通过关键字 optionalKeys 定义可以选择输入的参数 name。

因此执行该 Job 时候，必须输入 date 参数，最多输入 date、name 两个参数。任何其他的参数名字都会导致参数校验类不通过。

在执行 Job 时候，可以通过 JobParametersBuilder 构造作业参数。代码清单 4-8 展示了输入不同参数执行 billJob 的示例。

代码清单 4-8 输入不同参数执行 Job

```
1. executeJob("ch04/job/job-validator.xml", "billJob",
2.             new JobParametersBuilder().addDate("date", new Date()));
3.
4. executeJob("ch04/job/job-validator.xml", "billJob",
5.             new JobParametersBuilder().addDate("date", new Date())
6.                         .addString("test", "test parameter not allowed."));
```

该程序的说明如下。

1~2 行：billJob 可以正确执行，因为只输入了参数“date”

4~6 行：执行 billJob 会发生错误，会抛出异常 org.springframework.batch.core.JobParametersInvalidException，这是因为输入了非法的参数“test”。

4.1.4 Job 抽象与继承

Spring Batch 框架支持抽象的 Job 定义和 Job 的继承特性。通过定义抽象的 Job 可以将 Job 的共性进行抽取，形成父类的 Job 定义，父 Job 通常具有较多的共性；然后各个具体的 Job 可以继承父类 Job 的特性，并定义自己的属性。通过 Job 的属性 abstract 可以定义抽象的 Job，通过属性 parent 可以指定当前 Job 的父 Job。

Job 的抽象、继承与 Java 中的抽象、继承概念比较类似。父不能被实例化、子可以继承父的所有特性，甚至可以覆写掉父中的特性。

抽象 Job

通过 `abstract` 属性可以指定 Job 为抽象的 Job。抽象的 Job 不能被实例化，只能作为其他 Job 的父。通常可以在抽象 Job 中定义全局性的配置，供子 Job 使用。代码清单 4-9 给出了抽象 Job 配置。

代码清单 4-9 抽象 Job 配置

```
1.      <job id="baseJob" abstract="true">
2.          <listeners>
3.              <listener ref="sysoutListener"></listener>
4.          </listeners>
5.      </job>
```

其中，1 行：通过 `abstract` 属性定义 `baseJob` 为抽象的 Job，注意抽象的 Job 不能被实例化，任何试图直接调用 `baseJob` 的用例都会发生错误。

继承 Job

通过 `parent` 属性可以指定当前 Job 的父类，类似于 Java 世界中的继承一样，子类 Job 具有父类中定义的所有属性能力。子类继承时候，可以从抽象 Job 继承，也可以从普通的 Job 中继承。图 4-5 展示了 Job 间的继承关系，`billJob` 继承了抽象的 `baseJob`，多个 Job 可以继承自同一个 Job。

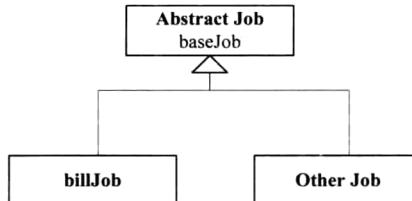


图 4-5 Job 继承

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

在抽象 `baseJob` 中定义 `sysoutListener`，在具体的 `billJob` 中定义 `sysAnnotationListener`，代码清单 4-10 给出了 `billJob` 的配置。

代码清单 4-10 继承 Job 配置

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

```
7.           </tasklet>
8.       </step>
9.       <listeners merge="true">
10.         <listener ref="sysoutAnnotationListener"></listener>
11.       </listeners>
12.     </job>
```

该程序说明如下。

1 行：通过属性 parent 指定当前 billJob 的父，通过集成 baseJob，billJob 拥有父类 baseJob 的所有特性，因此在 billJob 执行的时候也会调用 baseJob 中定义的拦截器 sysoutListener。

9 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 billJob 中有两个拦截器会同时工作

merge 列表

如果父子中均定义了拦截器，则可以通过设置 merge 属性为 true 对拦截器列表合并；如果设置 merge 属性为 false，则子类中定义的拦截器直接覆盖掉父类中定义的拦截器。

通过抽象和继承的特性可以方便地对 Job 进行共性抽取和 Job 的复用。

4.2 高级特性

4.2.1 Step Scope

什么是 Scope（作用域、范围、生命周期）？

Scope 用来声明 IOC 容器中对象的存活空间，即在 IOC 容器在对象进入相应的 Scope 之前，生成并装配这些对象，在该对象不再处于这些 Scope 的限定范围之后，容器通常会销毁这些对象。

Step Scope 是 Spring Batch 框架提供了自定义的 Scope，将 Spring Bean 定义为 Step Scope，支持 Spring Bean 在 Step 开始的时候初始化，在 Step 结束的时候销毁 Spring Bean，将 Spring Bean 的生命周期与 Step 绑定。

在 Spring Batch 框架中，Step Scope 会自动被注册到 Spring 上下文中，如果没有使用 Spring 的配置文件，需要显示的声明 Step Scope。显示声明 Step Scope 参见代码清单 4-11。

代码清单 4-11 显示声明 Step Scope

```
1.  <bean class="org.springframework.batch.core.scope.StepScope"/>
```

使用 Step Scope 的示例代码参见代码清单 4-12。

代码清单 4-12 使用 Scope 属性声明 Bean

```
1.  <bean:bean id="csvItemReader"
2.    class="org.springframework.batch.item.file.FlatFileItemReader"
3.    scope="step">
```

```
4.      .....
5.    </bean:bean>
```

其中，3行：通过属性 Scope="Step"来定义 csvItemReader 的生命周期和 Step 绑定。

通过使用 Step Scope，可以支持属性的 Late Binding（属性后绑定）能力。

4.2.2 属性 Late Binding

前面对账单的示例代码中对于读取文件的配置，都是将读取文件名字在开发期直接配置在 Spring 配置文件中。考虑一下实际的对账单场景，对账单的文件通常都是根据日期自动生成的，也就是说在开发期无法知道配置文件的名字，只有在运行期才能知道配置文件的名字。幸运的是 Spring Batch 框架可以通过属性后绑定的技术，支持在运行期获取属性的值。

Spring Batch 框架通过特定的表达式支持为 Job 或者 Step 关联的实体使用后绑定技术。在 Step Scope 中 Spring Batch 框架提供的可以使用的实体包括 jobParameters、jobExecutionContext、stepExecutionContext，具体参见表 4-5。

表 4-5 Late Binding 支持的实体

实 体	说 明
jobParameters	作业参数
jobExecutionContext	当前 Job 的执行器上下文
stepExecutionContext	当前 Step 的执行器上下文

下面的例子通过使用 jobParameters 为输入的对账单文件使用后绑定技术。

前面所有的例子中，需要读取的对账单文件名全部是在配置文件中硬编码的，在实际的使用场景中是不可能确定每期账单文件名字，而只有在调用期间才知道具体的文件名，下面的例子实现了在运行期指定需要执行的文件名的功能。

在配置文件硬编码文件名的示例代码参见代码清单 4-13。

代码清单 4-13 硬编码输入文件

```
1.      <bean:bean id="csvItemReader"
2.          class="org.springframework.batch.item.file.FlatFileItemReader"
3.          scope="step">
4.              <bean:property name="resource"
5.                  value="classpath:ch04/data/credit-card-bill-201303.csv"/>
6.              .....
7.      </bean:bean>
```

其中，5行：开发期直接使用指定的文件名，不友好。

使用后绑定技术，根据输入的作业参数指定代执行的文件名。示例代码参见代码清单 4-14。

代码清单 4-14 使用 Late Binding 指定输入文件

```
1.      <bean:bean id="csvItemReader"
2.          class="org.springframework.batch.item.file.FlatFileItemReader"
3.          scope="step">
4.              <bean:property name="resource"
5.                  value="#{jobParameters['inputResource']}"/>
6.              .....
7.      </bean:bean>
```

其中，5 行：定义输入文件使用运行 Job 时候输入的参数"inputResource"。

调用 Job 的示例代码参见代码清单 4-15。

代码清单 4-15 执行后绑定 Job

```
1.      executeJob("ch04/job/job-stepscope.xml", "billJob",
2.                  new JobParametersBuilder().addDate("date", new Date())
3.                  .addString("inputResource",
4.                  "classpath:ch04/data/credit-card-bill-201303.csv"));
```

其中，3~4 行：设置参数 inputResource 的值，可以在 Scope 为 Step 的 csvItemReader 中使用。

通过使用后绑定技术，避免在配置文件中使用硬编码的方式指定读取的配置文件，可以直接在运行期使用 Job 传入的参数。图 4-6 展示了 Job 配置与 Job Parameters 之间的关系。

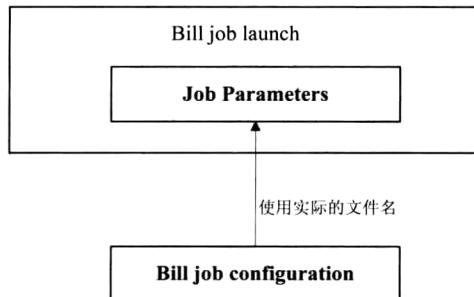


图 4-6 Job 配置与 Job Parameters 关系

4.3 运行 Job

Spring Batch 框架提供一组执行 Job 的接口 API。包括 JobLauncher、JobExplorer 和 JobOperator 三个操作 Job 的接口，三者关系参见图 4-7。JobLauncher 是最常用的作业调度器，通过给定的 Job Name 和 Job Parameters 可以执行 Job；JobExplorer 主要负责从 JobRepository 中获取执行的信息，包括获取作业实例、获取作业执行器、获取作业步执行器、获取正在运行的作业执行器、获取作业列表等操作；JobOperator 包含了 JobLauncher 和 JobExplorer 中的大部分操作。JobLauncher、JobExplorer 和 JobOperator 三者的详细描述参见表 4-6。

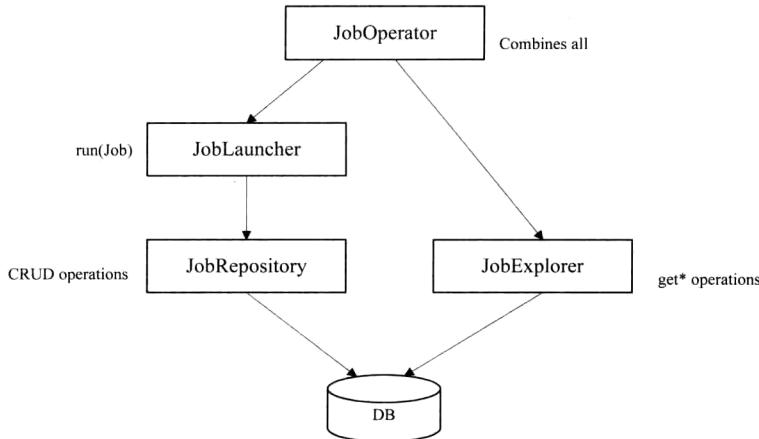


图 4-7 作业执行类 JobOperator、JobLauncher 和 JobExplorer 的关系

表 4-6 JobLauncher、JobExplorer 和 JobOperator 的详细描述

操作接口	说 明
org.springframework.batch.core.launch.JobLauncher	执行作业类
org.springframework.batch.core.explore.JobExplorer	作业状态查询类 返回作业状态的复杂对象，如 Job Execution、Job Instance、Step Execution 等
org.springframework.batch.core.launch.JobOperator	作业状态查询、作业执行类 返回作业状态的简单类型，如 Job Execution 的 id, Job Instance 的 id, Step Execution 的 id 等

JobLauncher

JobLauncher 是 Spring Batch 框架基础设施层提供的运行 Job 的能力。通过给定的 Job 名称和作业参数 JobParameters，可以通过 JobLauncher 执行 Job。通过 JobLauncher 执行 Job 时，Job 执行的状态信息通过 JobRepository 持久到数据库中。JobLauncher 接口主要操作参见图 4-8。



图 4-8 JobLauncher 接口操作的定义

JobExplorer

JobExplorer 主要负责从 JobRepository 中获取执行的信息，包括获取作业实例、获取作业执行器、获取作业步执行器、获取正在运行的作业执行器、获取作业列表等操作。JobExplorer 接口主要操作参见图 4-9。

```
JobExplorer
    ● getJobInstances(String, int, int) : List<JobInstance>
    ● getJobExecution(Long) : JobExecution
    ● getStepExecution(Long, Long) : StepExecution
    ● getJobInstance(Long) : JobInstance
    ● getJobExecutions(JobInstance) : List<JobExecution>
    ● findRunningJobExecutions(String) : Set<JobExecution>
    ● getJobNames() : List<String>
```

图 4-9 JobExplorer 接口操作的定义

JobOperator

JobOperator 包含了 JobLauncher 和 JobExplorer 中的大部分操作, 即包括从 JobRepository 中查询作业状态信息, 同时包含执行 Job 的操作。JobOperator 接口主要操作参见图 4-10。

```
JobOperator
    ● getExecutions(long) : List<Long>
    ● getJobInstances(String, int, int) : List<Long>
    ● getRunningExecutions(String) : Set<Long>
    ● getParameters(long) : String
    ● start(String, String) : Long
    ● restart(long) : Long
    ● startNextInstance(String) : Long
    ● stop(long) : boolean
    ● getSummary(long) : String
    ● getStepExecutionSummaries(long) : Map<Long, String>
    ● getJobNames() : Set<String>
```

图 4-10 JobOperator 接口操作的定义

4.3.1 调度作业

前面章节我们已经介绍了如何配置 Job, 一旦我们配置好 Job 后, 我们需要执行 Job 作业, 接下来我们描述外部系统如何通过 JobLauncher 来调度作业。代码清单 4-16 片段描述了如何通过 JobLauncher 来调度作业。

代码清单 4-16 JobLauncher 调度 Job

```
1. ApplicationContext context = new ClassPathXmlApplicationContext(
2.     "ch02/job/job.xml");
3. JobLauncher launcher = (JobLauncher) context.getBean("jobLauncher");
4. Job job = (Job) context.getBean("billJob");
5. try {
6.     JobExecution result = launcher.run(job, new JobParameters());
7.     System.out.println(result.toString());
8. } catch (Exception e) {
9.     e.printStackTrace();
}
```

JobLauncher 支持对作业的同步、异步两种调用模式。除了我们上面通过 Java Main 操作调用 Job 外，我们还可以在命令行、Web 应用、定时任务等入口调用 Job。接下来的章节我们向读者介绍 Job 的同步异步调用，以及 Job 与外界系统的调用关系。

4.3.1.1 同步异步

默认情况下，JobLauncher 的 run 操作通过同步方式调用 Job，任何调用 Job 的客户端需要等待 Job 的执行结果返回后才能结束。

同步执行 Job 作业的序列图参见图 4-11。

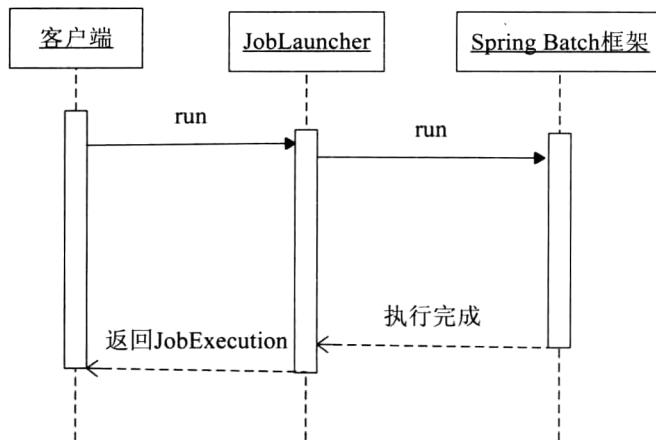


图 4-11 Job 同步执行序列图

同步操作的优势在于作业一旦执行完毕，调用客户端能够立刻收到返回值。但在实际的 Web 应用中进行 Job 调度时，因为 Job 的作业时间太长，客户端不能一直等待，用户的忍耐时间有限。同时如果 Job 作业执行时间太长，会导致 Web 容器中的大量线程被批处理作业 hold 住，无法为其他类型的请求服务，严重影响服务器的性能和吞吐量。为了解决此问题，需要通过异步的方式调用作业。如果客户端需要知道作业的执行结果，建议经过一段时间后，主动查询当前作业的执行情况。JobLauncher 提供了异步执行 Job 的能力。

异步执行 Job 作业的序列图参见图 4-12。

配置异步调用的 JobLauncher 只需要增加属性 taskExecutor，该属性表示当前执行的线程池，配置代码参见代码清单 4-17。

代码清单 4-17 配置异步 JobLauncher

```
1.      <task:executor id="executor" pool-size="5" />
2.      <!-- 异步作业调度器 -->
3.      <bean:bean id="jobLauncherAsyn"
4.          class="org.springframework.batch.core.launch.support.
SimpleJobLauncher">
```

```

5.      <bean:property name="jobRepository" ref="jobRepository"/>
6.      <bean:property name="taskExecutor" ref="executor" />
7.    </bean:bean>

```

其中，1 行：配置大小为 5 的线程池。

6 行：为作业调度器配置执行的线程池，作业执行期间会从线程池中选择一个线程执行 Job。

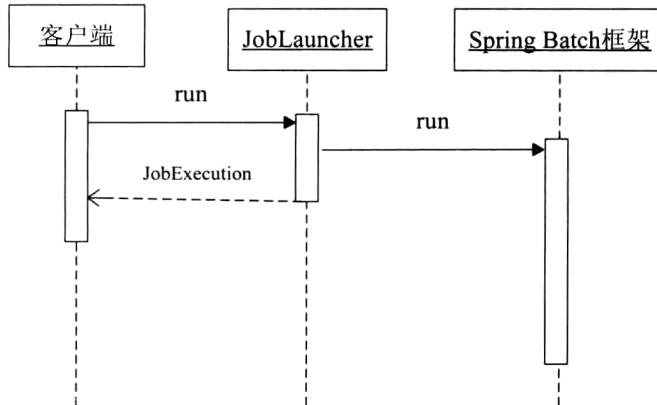


图 4-12 Job 异步执行序列图

运行代码工程中的 `test.com.juxtapose.example.ch04.JobLaunchAsyn`，仔细观察控制台输出：首先会输出 `Job Execution` 的信息，然后输出 `Job` 执行的具体信息。即在 `JobLauncher` 的 `run` 方法被调用后，客户端立刻返回；同时后台使用其他线程执行 `Job` 作业。

4.3.1.2 Job 与外界系统

在实际的 Job 使用场景中，标准 Web 应用、定时任务调度器、命令行等都可能触发不同的 Job 操作。图 4-13 展示了批处理作业和外界系统的关系图。

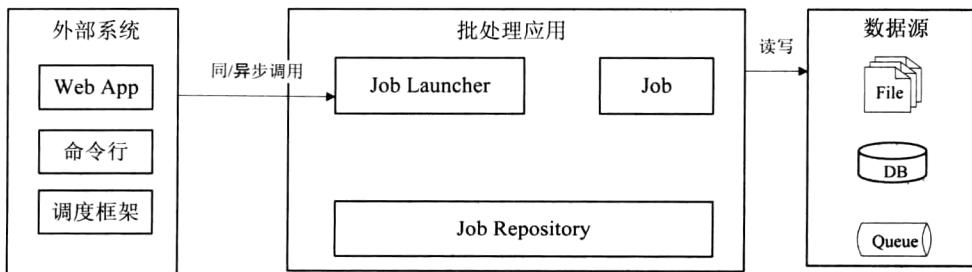


图 4-13 批处理作业和外界系统的关系图

接下来的三节我们将学习如何通过命令行、调度框架和 Web 应用程序执行 Job 作业。

4.3.2 命令行执行

Spring Batch 框架提供通过命令行方式执行 Job 的入口，通过命令行的方式可以在一个单独的 JVM 中执行批处理作业。通过命令行的方式可以手动触发，也可以定义自动任务通过脚本的方式执行批处理作业。Spring Batch 框架提供的命令行执行类：org.springframework.batch.core.launch.support.CommandLineJobRunner。

通过命令行方式执行 Job 的关系参见图 4-14。

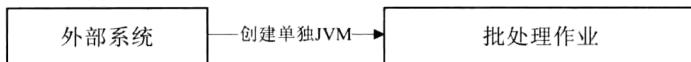


图 4-14 通过命令行执行批处理作业

示例

按照下面的步骤通过命令行执行 Hello World 章中的批处理作业。

(1) 构建示例项目 spring-batch-example。

可以在 Eclipse 中通过 Maven 插件构建项目，在 target 目录中生成 jar 文件。

(2) 调出命令行，进入目录：/spring-batch-example/target。

(3) 执行如下的命令，命令参见代码清单 4-18。

代码清单 4-18 命令行执行 Job 命令

```
java -classpath "./dependency/*;spring-batch-example-1.0.jar"
    org.springframework.batch.core.launch.support.CommandLineJobRunner
    ch02/job/job.xml
    billJob
```

命令参数说明：

-classpath "./dependency/*;spring-batch-example-1.0.jar" 表示当前执行的 classpath。

ch02/job/job.xml 表示执行的 Spring Batch 的配置文件。

billJob 表示需要执行的批处理作业。

(4) 查看控制台，会成功打印出执行的文件信息，控制台内容输出参见代码清单 4-19。

代码清单 4-19 命令行执行 Job 的控制台输出

```
1. 2013-03-18 20:29:52,397 INFO [org.springframework.batch.core.job.SimpleStepHandler] - Executing step: [billStep]
2. accountID=4047390012345678;name=tom;amount=100.0;date=2013-2-2
   12:00:08;address=Lu Jia Zui road
3. accountID=4047390012345678;name=tom;amount=320.0;date=2013-2-3
   10:35:21;address=Lu Jia Zui road
4. accountID=4047390012345678;name=tom;amount=674.7;date=2013-2-6
   16:26:49;address=South Linyi road
5. accountID=4047390012345678;name=tom;amount=793.2;date=2013-2-9
   15:15:37;address=Longyang road
```

```
6. accountID=4047390012345678;name=tom;amount=360.0;date=2013-2-11  
11:12:38;address=Longyang road  
7. accountID=4047390012345678;name=tom;amount=893.0;date=2013-2-28  
20:34:19;address=Hunan road
```

(5) 查看文件/spring-batch-example/target/target/ch02/outputFile.csv, 内容会正确地输出。

其中, CommandLineJobRunner 参数说明如下。

```
java -classpath "xxx" CommandLineJobRunner jobPath <options> jobIdentifier  
(jobParameters)
```

jobPath: CommandLineJobRunner 默认使用 ClassPathXmlApplicationContext 从当前的 classpath 中加载配置文件, 可以通过写前缀的方式更改 CommandLineJobRunner 的加载器, 例如使用 file:./job.xml 方式, 表示使用当前目录下面的 job.xml 文件。

options: 可选的参数, 支持的参数包括-restart、-stop、-abandon、-next。详细说明参见表 4-7。

表 4-7 CommandLineJobRunner 命令 options 参数说明

可选参数	说 明
-restart	根据 jobIdentifier 重启最后一次失败的作业
-stop	根据 jobIdentifier 停止正在执行的作业
-abandon	根据 jobIdentifier 废弃 stopped 的作业
-next	根据 JobParametersIncrementer 执行下一个作业

jobIdentifier: 作业的名字 (正常执行一个 Job 的时候, 需要传入 Job 的名字, 即 Spring 配置文件中配置 Job 的 Bean ID), 或者是 job execution 的 id (当使用-restart、-stop、-abandon 时候, 需要传入 job execution 的 id)。

job Parameters: 作业参数可以有 0 到多个, 其支持的类型为 String、Date、Long、Double。 Job Parameters 支持的参数及示例参见表 4-8。

表 4-8 Job Parameters 支持的参数及示例

参数类型	示 例
String	inputResource(string)=ch02/job.xml
Date	createTime(date)=2013/03/18
Long	timeout(long)=5000
Double	account(double)=100.23

获取作业退出状态

通过 CommandLineJobRunner 在命令行中执行 Job 同样可以获取 Job 执行的退出状态 ExitStatus, 默认情况 Job 的退出状态 ExitStatus 和 Job 的批处理状态 BatchStatus 一致。 CommandLineJobRunner 的退出状态值由类 org.springframework.batch.core.launch.support.SimpleJvmExitCodeMapper(实现接口: org.springframework.batch.core.launch.support.ExitCodeMapper)

定义，具体返回的退出状态值参见表 4-9。

表 4-9 命令行执行 Job 的退出状态值

状态值	说 明
0	作业正常完成，状态为 COMPLETED
1	作业完成失败，状态为 FAILED
2	作业失败，例如没有此作业

可以通过重新实现接口 org.springframework.batch.core.launch.support.ExitCodeMapper，来自定义作业的退出状态。例如通过自定义退出状态类，可以实现正常完成状态退出为 1，失败为 2，停止为 3，其他为 4。代码清单 4-20 为自定义退出状态实现类代码。

代码清单 4-20 自定义退出状态实现类

```
1. public class CustomerExitCodeMapper implements ExitCodeMapper {  
2.     public int intValue(String exitCode) {  
3.         if (ExitStatus.COMPLETED.getExitCode().equals(exitCode)) {  
4.             return 1;  
5.         } else if (ExitStatus.FAILED.getExitCode().equals(exitCode)) {  
6.             return 2;  
7.         } else if (ExitStatus.STOPPED.getExitCode().equals(exitCode)) {  
8.             return 3;  
9.         } else {  
10.            return 4;  
11.        }  
12.    }  
13. }
```

为了能在命令行中使用上面开发的退出码匹配类，需要在配置文件中定义上面的实现，配置内容参见代码清单 4-21。

代码清单 4-21 配置自定义退出码实现类

```
1. <bean:bean class="com.juxtapose.example.ch04.exitstatus.Customer  
ExitCodeMapper">  
2. </bean:bean>
```

可以通过代码清单 4-22 中的命令行，执行 job-exitstatus.xml 中定义的 billJob 作业。

代码清单 4-22 命令行执行 Job，使用自定义退出码实现类

```
java -classpath "./dependency/*;spring-batch-example-1.0.jar"  
      org.springframework.batch.core.launch.support.CommandLineJobRunner  
      ch04/job/job-exitstatus.xml  
      billJob  
      inputResource(string)=ch04/data/credit-card-bill-201303.csv  
      outputResource(string)=file:target/ch04/exitstatus/outputFile.csv  
      date(string)=2013/03/18
```

说明：在使用 `CommandLineJobRunner` 进行命令行执行过程中，Spring 对 `CommandLineJobRunner` 中的依赖全部使用基于类型的自动装配。例如 `CommandLineJobRunner` 中的属性 `launcher`，Spring 会自动基于类型 `JobLauncher` 进行装配，如果没有 `JobLauncher` 的定义或者有多个定义都会导致错误。

代码清单 4-23 展示当有 2 个 `JobLauncher` 定义时候的错误信息。

代码清单 4-23 命令行执行 Job 存在多个 `JobLauncher` 时，错误输出

```
: No unique bean of type [org.springframework.batch.core.launch.JobLauncher]
is defined: expected single matching bean but found 2: [jobLauncher,
jobLauncherAsyn]
    at org.springframework.beans.factory.support.AbstractAutowireCapable
BeanFactory.autowireByType(AbstractAutowireCapableBeanFactory.java:1167)
    ....
```

4.3.3 与定时任务集成

Spring Batch 提供了 Job 的执行能力，其本身不是一个定时的调度框架，因此可以将定时调度框架和 Spring Batch 结合起来完成定时作业。Spring 本身提供了一个轻量级的调度框架 `Spring scheduler`。本节展示如何通过 `Spring scheduler` 定时调度 Spring Batch 中的 Job。

图 4-15 展示了 Spring Batch 框架和 Spring scheduler 之间的关系。

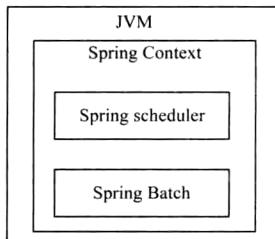


图 4-15 Spring Batch 框架和 Spring scheduler 关系图

`Spring scheduler` 使用非常简单，只需要完成 2 步即可完成简单的定时调度功能。

- (1) 定义一个 `scheduler`，该 `scheduler` 提供执行定时任务的线程。
- (2) 定义需要定时操作的方法和调度周期。

Spring Scheduler 和 Spring Batch Job 的配置参见代码清单 4-24。完整代码参见文件 `ch04/job/job-spring-scheduler.xml`。

代码清单 4-24 配置 Spring Scheduler 和 Spring Batch Job

```
1.      <task:scheduler id="scheduler" pool-size="10" />
2.
3.      <!-- 每一秒钟，执行对象 schedulerLauncher 的 launch 方法一次 -->
4.      <task:scheduled-tasks scheduler="scheduler">
```

```

5.          <task:scheduled ref="schedulerLauncher" method="launch" fixed-
6.          rate="1000" />
7.
8.      </task:scheduled-tasks>
9.
10.     <bean:bean id="schedulerLauncher"
11.         class="com.juxtapose.example.ch04.scheduler.SchedulerLauncher">
12.         <bean:property name="job" ref="helloworldJob" />
13.         <bean:property name="jobLauncher" ref="jobLauncher" />
14.     </bean:bean>
15.
16.     <!-- HelloWorld Job -->
17.     <job id="helloworldJob">
18.         <step id="helloworldStep">
19.             <tasklet>
20.                 <bean:bean class="com.juxtapose.example.ch04.HelloWorld
21.                               Tasklet">
22.                     </bean:bean>
23.                 </tasklet>
24.             </step>
25.         </job>

```

该程序说明如下。

1 行：定义执行定时任务的线程池大小。

4~6 行：fixed-rate="1000" 表示每秒执行一次对象 schedulerLauncher 的 launch 方法。

8~12 行：定时任务执行的对象声明，在类 com.juxtapose.example.ch04.scheduler.Scheduler Launcher 中的 launch 方法负责具体 Job 的调用。

15~22 行：定义了自定义实现 Tasklet 的任务。

com.juxtapose.example.ch04.scheduler.SchedulerLauncher 类的实现参见代码清单 4-25。

代码清单 4-25 定时调度类 SchedulerLauncher 定义

```

1.  public class SchedulerLauncher {
2.      private Job job;
3.      private JobLauncher jobLauncher;
4.
5.      public void launch() throws Exception {
6.          JobParameters jobParams = createJobParameters();
7.          jobLauncher.run(job, jobParams);
8.      }
9.
10.     private JobParameters createJobParameters() {
11.         JobParameters jobParams = new JobParametersBuilder().
12.             addDate("executeDate", new Date()).toJobParameters();
13.         return jobParams;
14.     }

```

```
15. ....省略get* set*方法  
16. }
```

其中，5~8 行：执行具体的 job 调度。

执行测试类 test.com.juxtapose.example.ch04.JobLaunchScheduler，控制台每一秒钟调度一次任务，控制台输出信息参见代码清单 4-26。

代码清单 4-26 定时调度 Job 的控制台输出

```
Execute job :helloworldJob.  
JobParameters:executeDate = Tue Mar 19 20:20:53 CST 2013 (DATE);  
.....  
Execute job :helloworldJob.  
JobParameters:executeDate = Tue Mar 19 20:20:54 CST 2013 (DATE);  
.....  
Execute job :helloworldJob.  
JobParameters:executeDate = Tue Mar 19 20:20:55 CST 2013 (DATE);  
.....  
Execute job :helloworldJob.  
JobParameters:executeDate = Tue Mar 19 20:20:56 CST 2013 (DATE);
```

注意：……处表示省略了其他日志信息。

通过本节学习，可以将 Spring Batch 框架和其他的调度框架进行集成使用，如 Quartz 框架、Cron（Linux 环境下的定时任务工具）。

4.3.4 与 Web 应用集成

Spring Batch 框架基于 Spring 开发，可以方便地内嵌在 Web 应用中使用，这样批处理作业可以通过 HTTP 协议进行远程的访问。同样可以在 Web 应用中内嵌定时任务处理框架，方便在 Web 应用内部通过定时框架调用 Spring Batch 中定义的 Job。

Web 应用、Spring Batch 框架和 Spring scheduler 的关系参见图 4-16。

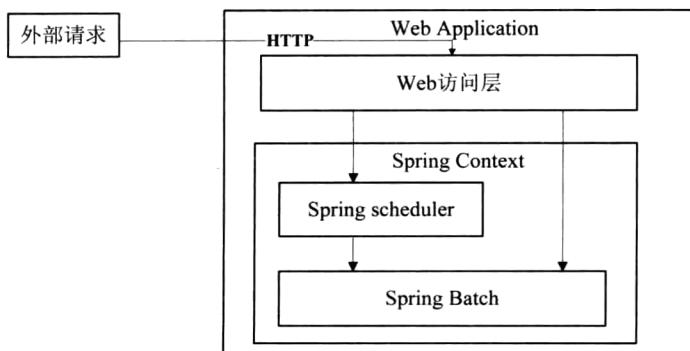


图 4-16 Web 应用、Spring Batch 框架和 Spring scheduler 关系图

外部请求可以直接通过 Web 访问层执行批处理作业，也可以由 Web 访问层访问批处理框架，然后由批处理框架调度批处理作业。

使用 Spring MVC 框架，调用指定的 Spring Batch 的 Job。Spring MVC 框架通过定义一个 controller 来描述具体的 HTTP 服务，controller 可以通过简单的 annotation（@Controller、@RequestMapping、@RequestParam 等）来描述。

配置 Controller

com.juxtapose.example.ch04.web.JobLauncherController 类定义代码，参见代码清单 4-27。

代码清单 4-27 JobLauncherController 类定义

```
1.  @Controller
2.  public class JobLauncherController {
3.      private static final String JOB_NAME = "jobName";
4.      private JobLauncher jobLauncher;
5.      private JobRegistry jobRegistry;
6.
7.      public JobLauncherController(JobLauncher jobLauncher, JobRegistry
8.          jobRegistry) {
9.          this.jobLauncher = jobLauncher;
10.         this.jobRegistry = jobRegistry;
11.     }
12.
13.     @RequestMapping(value="executeJob",method=RequestMethod.GET)
14.     public void launch(@RequestParam String jobName,HttpServletRequest
15.         request) throws Exception
16.     {
17.         JobParameters jobParameters = bulidParameters(request);
18.         jobLauncher.run( jobRegistry.getJob(jobName),jobParameters);
19.     }
20.
21.     private JobParameters bulidParameters(HttpServletRequest request) {
22.         JobParametersBuilder builder = new JobParametersBuilder();
23.         @SuppressWarnings("unchecked")
24.         Enumeration<String> paramNames = request.getParameterNames();
25.         while(paramNames.hasMoreElements()) {
26.             String paramName = paramNames.nextElement();
27.             if(!JOB_NAME.equals(paramName)) {
28.                 builder.addString(paramName,request.getParameter
29.                     (paramName));
30.             }
31.         }
32.         return builder.toJobParameters();
33.     }
34. }
```

该程序说明如下。

1 行：@Controller 表示 MVC 框架中的控制器。

12 行：@RequestMapping (value="executeJob",method=RequestMethod.GET) 表示当前操作对应请求为 executeJob，通过 get 的方式访问。

13 行：@RequestParam String jobName 表示 jobName 为请求的参数，表示需要执行 Job 的名字。

16 行：执行具体的 Job，Job 具体的对象通过 jobRegistry 获取（jobRegistry 中存放 Spring 配置文件中定义的所有 Job 信息，具体配置参见后面的作业配置文件）。

19~30 行：将 HTTP 请求的参数转换为作业参数。

配置 MVC 框架

在 web.xml 中配置加载 Spring 容器的 listener 和对应的 servlet 配置。web.xml 配置定义参见代码清单 4-28。

代码清单 4-28 Web 应用调用 Job 的 web.xml 配置

```
1.      <display-name>SpringBatch</display-name>
2.      <listener>
3.          <listener-class>org.springframework.web.context.Context
4.              LoaderListener
5.          </listener-class>
6.      </listener>
7.      <servlet>
8.          <servlet-name>batchapp</servlet-name>
9.          <servlet-class>org.springframework.web.servlet.DispatcherServlet
10.         </servlet-class>
11.     </servlet>
12.     <servlet-mapping>
13.         <servlet-name>batchapp</servlet-name>
14.         <url-pattern>/*</url-pattern>
15.     </servlet-mapping>
```

其中，3 行：定义加载 Spring 容器，默认加载 WEB-INF/applicationContext.xml 来初始化 Spring 容器。

6~10 行：定义匹配的 url，MVC 框架默认使用（**servlet-name**）-servlet.xml 文件来加载控制器，本例子中加载文件 batchapp-servlet.xml。配置控制器文件 batchapp-servlet.xml 参见代码清单 4-29。

代码清单 4-29 配置控制器文件 batchapp-servlet.xml

```
1.      <bean class="com.juxtapose.example.ch04.web.JobLauncherController">
2.          <constructor-arg ref="jobLauncher" />
3.          <constructor-arg ref="jobRegistry" />
4.      </bean>
```

其中，1~4 行：定义了对应的 web controller。

需要注意 JobLauncherController 中使用了两个对象：jobLauncher 和 jobRegistry，前者负责调用 Job 执行；后者定义了 Job 的注册器，jobRegistry 存放了所有的 Job 对象，可以根据 Job 的名字获取对应的 Job 对象。

JobLauncherController 中引用的 jobLauncher 和 jobRegistry 从 ContextLoaderListener 中加载的 Spring 容器获取对象。ContextLoaderListener 默认加载文件 applicationContext.xml 来初始化 Spring 容器。

配置 Spring 容器初始化文件

applicationContext.xml 负责引用所有的 Spring 配置文件，可以看作加载 Spring 容器的入口。本例中 applicationContext.xml 引入了 job-context.xml 和 job-sample.xml 文件。applicationContext.xml 文件的定义参见代码清单 4-30。

代码清单 4-30 applicationContext.xml 文件定义

```
1.      <import resource="job-context.xml"/>
2.      <import resource="job-sample.xml"/>
```

job-context.xml 中定义基本的批处理框架定义。

job-sample.xml 文件定义了一个示例的 Job。

JobLauncherController 用到的 jobLauncher 和 jobRegistry 的定义参见代码清单 4-31。

代码清单 4-31 JobLauncherController 类定义

```
1.      <bean:bean class="org.springframework.batch.core.configuration.support.
2.                           JobRegistryBeanPostProcessor">
3.          <bean:property name="jobRegistry" ref="jobRegistry" />
4.      </bean:bean>
5.
6.      <bean:bean id="jobRegistry"
7.          class="org.springframework.batch.core.configuration.support.
     MapJobRegistry" />
```

其中，1~4 行：通过类 JobRegistryBeanPostProcessor 的声明，可以在 Job 定义加载后，自动注册到 jobRegistry 中。

部署应用

将示例工程目录 spring-batch-example/src/main/resources/ch04/webapp/目录下的内容复制到 tomcat 目录\webapps 下面，应用名为 batchapp。

应用 batchapp 目录结构参见图 4-17。

应用部署后，启动 tomcat 应用，

如下的 url 访问，通过 HTTP 请求执行作业 chunkJob，使用的作业参数为 date。

<http://localhost:8080/batchapp/executeJob?jobName=chunkJob&date=20130320>

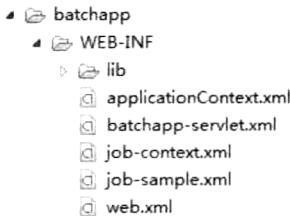


图 4-17 batchapp 目录结构

4.3.5 停止 Job

批处理作业执行过程中，如果发生了意外需要终止当前执行的 Job，Spring Batch 框架提供了停止正在运行 Job 的能力，通过接口 `org.springframework.batch.core.launch.JobOperator` 中的 `stop` 方法来实现；另外在作业执行过程中，开发人员可以根据业务的需要终止正在运行的 Job，可以在 Job 的业务代码中定义何时终止 Job。接下来我们学习这两种停止 Job 的能力。

4.3.5.1 JobOperator 停止 Job

Job 执行期间通过 `org.springframework.batch.core.launch.JobOperator` 的 `stop` 方法停止正在运行的 Job。`stop` 方法返回 `boolean` 值，表示当前终止消息是否成功发送，至于什么时候终止消息，则由 Spring Batch 框架决定。

图 4-18 展示了 `JobOperator` 的操作定义。

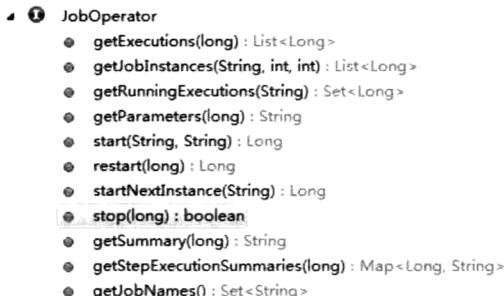


图 4-18 JobOperator 定义

可通过 `JobOperator` 的 `stop` 方法停止正在运行的 Job，代码参见代码清单 4-32。

代码清单 4-32 通过 JobOperator 的 stop 操作停止 Job

```

1. Set<Long> runningExecutions = jobOperator.getRunningExecutions(jobName);
2. Iterator<Long> iterator = runningExecutions.iterator();
3. while(iterator.hasNext()) {
4.     boolean sendMessage = jobOperator.stop(iterator.next());
5.     System.out.println("sendMessage:" + sendMessage);
6. }
  
```

其中，1行：根据 jobName 查找正在运行的所有的作业执行器的 ID。

3~6行：迭代执行，停止所有正在运行的 Job。

4行：根据具体的作业执行器的 ID 停止 Job。

查看类 org.springframework.batch.core.launch.JobOperator 的源代码，stop 方法调用后并没有立刻终止 Job，而是向 Spring Batch 框架发送了终止请求，对应的返回值为 true 表示当前终止作业的消息已经成功发送，如果对应的返回值为 false 表示终止作业的消息发送失败。只有当停止消息发送成功后，并且作业处理进入到 Spring Batch 框架时，才会真正停止正在运行的 Job。如果业务操作一直运行，会导致停止 Job 会需要很多时间。

接下来通过一个完整的示例，来展示如何通过 JobOperator 的 stop 方法来终止 Job。

配置文件

上面代码完成后，需要在配置文件中定义 JobOperator 及对应的属性定义，配置内容参见代码清单 4-33。

完整配置文件参见 ch04/job/job-stop.xml。

代码清单 4-33 配置 JobOperator 及属性

```
1.  <job id="chunkJob">
2.    <step id="chunkStep">
3.      <tasklet>
4.        <chunk reader="reader" writer="writer" commit-interval="10" />
5.      </tasklet>
6.    </step>
7.  </job>
8. <bean:bean
9.   class="org.springframework.batch.core.configuration.support.
10.          JobRegistryBeanPostProcessor">
11.    <bean:property name="jobRegistry" ref="jobRegistry" />
12.  </bean:bean>
13.
14. <bean:bean id="jobRegistry"
15.   class="org.springframework.batch.core.configuration.support.
16.          MapJobRegistry" />
17. <bean:bean id="jobExplorer"
18.   class="org.springframework.batch.core.explore.support.
19.          JobExplorerFactoryBean">
20.    <bean:property name="dataSource" ref="dataSource" />
21.  </bean:bean>
22. <bean:bean id="jobOperator"
23.   class="org.springframework.batch.core.launch.support.
```

```
    SimpleJobOperator">
24.      <bean:property name="jobRepository" ref="jobRepository" />
25.      <bean:property name="jobLauncher" ref="jobLauncherAsyn" />
26.      <bean:property name="jobRegistry" ref="jobRegistry" />
27.      <bean:property name="jobExplorer" ref="jobExplorer" />
28.  </bean:bean>
```

该程序说明如下。

1~7 行：定义作业，作业的具体 reader 和 writer 不在此贴出，可以到示例工程代码中查看，reader 功能是从 0 开始读取数据，直到 Integer 的最大值；writer 将读到的数据打印在控制台上。

8~12 行：定义 Job 自动注册功能。

14~15 行：定义作业注册器。

17~20 行：定义作业的浏览接口，可以通过数据库查询 Job 执行的元数据信息。

22~28 行：定义我们需要的 jobOperator，可以对作业实例进行 CRUD 和控制处理。

18 行：使用了异步的作业调度器：jobLauncherAsyn，目的是后面测试代码中可以在同一个线程中方便终止 Job。

示例代码

在本示例代码中，首先通过异步的作业调度器来执行一个 Chunk 类型的作业，然后通过 JobOperator 终止 Job。示例代码参见代码清单 4-34。

完整的示例代码参见工程中的类：test.com.juxtapose.example.ch04.JobLaunchStop。

代码清单 4-34 异步作业调度器执行 Job, JobOperator 终止 Job 示例代码

```
1.  public static void executeJobAndStop(String jobPath, String jobName,
2.      JobParametersBuilder builder) {
3.      ApplicationContext context = new ClassPathXmlApplicationContext
4.          (jobPath);
5.      JobLauncher launcher = (JobLauncher) context.getBean("jobLauncherAsyn");
6.      JobOperator jobOperator = (JobOperator) context.getBean
7.          ("jobOperator");
8.      Job job = (Job) context.getBean(jobName);
9.      try {
10.          launcher.run(job, builder.toJobParameters());
11.          Set<Long> runningExecutions = jobOperator.getRunningExecutions
12.              (jobName);
13.          Iterator<Long> iterator = runningExecutions.iterator();
14.          while(iterator.hasNext()){
15.              boolean sendMessage = jobOperator.stop(iterator.next());
16.              System.out.println("sendMessage:" + sendMessage);
17.          }
18.      } catch (Exception e) {
19.          e.printStackTrace();
20.      }
21.  }
```

```

17.      }
18.  }
19.
20. /**
21.  * @param args
22. */
23. public static void main(String[] args) {
24.     executeJobAndStop("ch04/job/job-stop.xml", "chunkJob",
25.                         new JobParametersBuilder().addDate("date", new Date()));
26. }

```

其中，9 行：通过 jobOpeartor 查找作业名“chunkJob”的所有正在运行的作业执行器 ID。
12 行：根据作业执行器 ID，发送停止作业的消息。
24~25 行：方法入口，使用配置文件 ch04/job/job-stop.xml 中的定义，启动并终止 chunkJob。

运行用例

执行用例 test.com.juxtapose.example.ch04.JobLaunchStop，控制台信息可以看到仅执行了一次循环操作。

代码清单 4-35 展示了控制台输出的信息。

代码清单 4-35 Job 执行控台输出

```

2013-03-20 07:57:28,147 INFO [org.springframework.batch.core.job.SimpleStepHandler] - Executing step: [chunkStep]
sendMessage:true
Write begin:
1,2,3,4,5,6,7,8,9,10,Write end!

```

通过控制台信息可以看出，作业步仅执行了一次循环就停止了。为了验证我们的猜测，查看作业步执行器表，可以看出读的次数仅有 10 次，正好验证了我们根据控台信息的猜测。

表 batch_step_execution 中的数据参见图 4-19。读操作一共有 10 次。

STEP_EXECUTION_ID	STEP_NAME	READ_COUNT	JOB_EXECUTION_ID	VERSION
114	chunkStep	10	115	3

图 4-19 作业步执行器记录

到对应的数据库中查看作业的执行状态，首先查看作业实例表，chunkJob 新产生作业实例编号为 114，再查看作业执行器表，对应作业实例 ID 为 114 的作业执行器的状态为 STOPPED 状态。

作业实例表 batch_job_instance 中的数据，参见图 4-20。

JOB_INSTANCE_ID	VERSION	JOB_NAME	JOB_KEY
114	0	chunkJob	70c3852431377e55179947a7dbb6e251

图 4-20 作业实例记录

作业执行器表 batch_job_execution 中的数据，参见图 4-21。

JOB_EXECUTION_ID	STATUS	EXIT_CODE	JOB_INSTANCE_ID	VERSION
115	STOPPED	STOPPED	114	3

图 4-21 作业执行器记录

通过 JMX 方式操作 JobOperator

上面的示例演示了如何在 Java 代码中执行停止作业，Spring 框架同样提供了其他的操作方式来友好地终止正在执行的作业。Spring 框架同样提供友好的方式将 Bean 暴露为 JMX 服务，仅仅需要在 Spring 配置文件中简单地定义。JConsole 提供了简单的执行 JMX 服务的入口，一旦 JobOperator 定义为 JMX 服务后，就可以通过 JConsole 的方式操作对应的 stop 操作。

将 JobOperator 配置为 JMX 服务参见代码清单 4-36。

代码清单 4-36 JobOperator 配置为 JMX 服务

```
1. <bean:bean class="org.springframework.jmx.export.MBeanExporter">
2.   <bean:property name="beans">
3.     <bean:map>
4.       <bean:entry key="com.juxtapose.example:name=jobOperator"
5.         value-ref="jobOperator" />
6.     </bean:map>
7.   </bean:property>
8. </bean:bean>
```

其中，4~5 行：将对象 jobOperator 暴露为 JMX 服务，定义服务名称为：“com.juxtapose.example:name=jobOperator”。

按照下面的操作步骤执行，展示了如何通过 JConsole 方式停止正在执行的 Job。

- (1) 执行类 test.com.juxtapose.example.ch04.JobLaunchStopJMX。
- (2) 通过 DB 查看当前作业的执行器 ID，本例操作的时候为 119。
- (3) 在命令行输入 jconsole 命令，调出 JMX 控制台，参见图 4-22。
- (4) 选择名字为 test.com.juxtapose.example.ch04.JobLaunchStopJMX 的进程，单击连接后进入步骤 5 的监控页面，参见图 4-23。

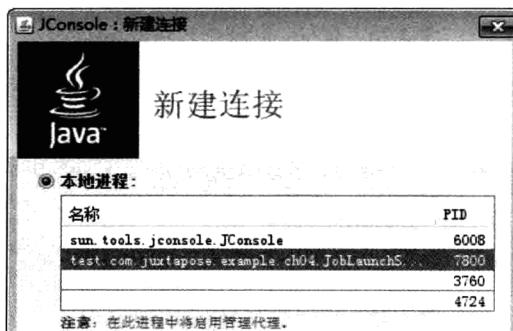


图 4-22 JConsole 登录界面

(5) 进入控台后，选择 MBean 页，找到对应的 stop 操作

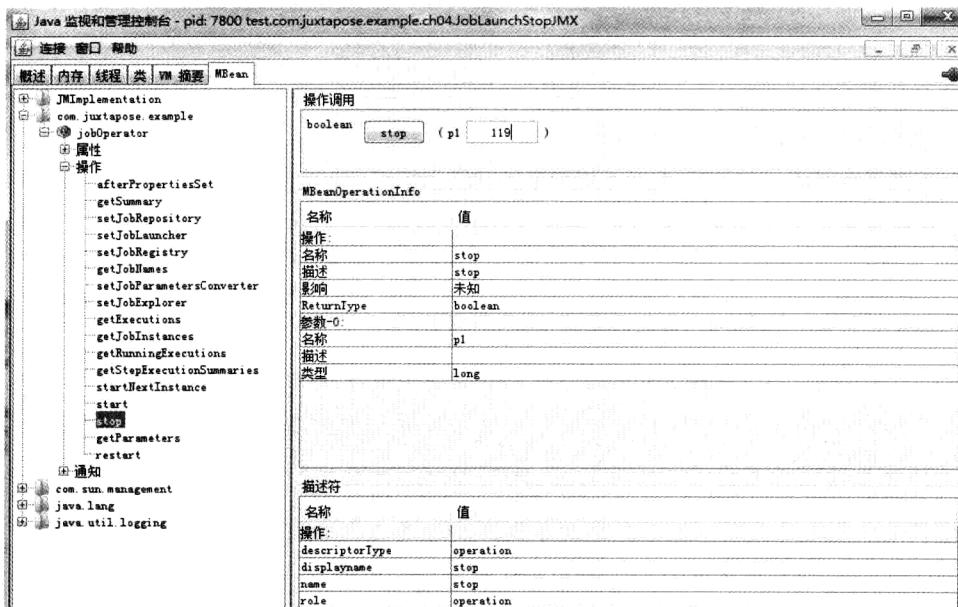


图 4-23 JConsole 控制台界面

(6) 选择 stop 操作后，输入参数 119，单击 stop 按钮，即可以停止当前的任务操作。

执行 stop 操作的返回值为 true，表示当前停止任务的请求已经被成功发送；如果要看任务是否被正确停止，需要参照步骤 7 的方式到数据库查看数据。

stop 操作的返回值参见图 4-24。

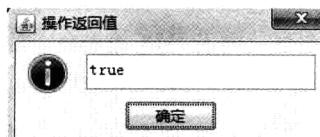


图 4-24 JConsole 操作返回值界面

(7) 查看数据库表 batch_job_execution，确认执行器为 119 的 Job 已经被停止。

图 4-25 展示了执行器为 119 的作业状态为 STOPPED。

	JOB_EXECUTION_ID	STATUS	EXIT_CODE	JOB_INSTANCE_ID	VERSION	CREATE_TIME
1	116	STOPPED	STOPPED	115	3	2013-03-20 08:23:05
2	117	STOPPED	STOPPED	116	3	2013-03-20 08:30:22
3	118	STARTED	UNKNOWN	117	1	2013-03-20 08:38:32
4	119	STOPPED	STOPPED	118	3	2013-03-20 08:38:55

图 4-25 执行器为 119 的作业状态

4.3.5.2 业务停止 Job

JobOperator 提供停止 Job 能力，通过 JobOperator 可以在作业外部，如 JMX 控制台，或者外部独立的进程中终止 Job。Spring Batch 框架同样提供了在作业步执行期间终止任务的能力。通过 StepExecution 中的 setTerminateOnly 操作可以终止正在运行的任务。

StepExecution.setTerminateOnly()会发送一个停止消息给框架，一旦 Spring Batch 框架接收到停止消息，并且框架获取作业的控制权，Spring Batch 框架会自动终止作业。

在通过业务操作终止任务操作时，不要在读、处理、写的业务逻辑中终止任务，以尽量保证业务操作的完整性。Spring Batch 框架提供了丰富的拦截器机制，可以在拦截器中执行任务的终止，例如在 ItemReadListener 中的 beforeRead()中终止任务。

通过拦截器（com.juxtapose.example.ch04.stop.StopStepListener<T>）终止任务示例代码参见代码清单 4-37。

代码清单 4-37 作业终止拦截器 StopStepListener

```
1. public class StopStepListener<T> implements StepExecutionListener,
2.     ItemReadListener<T> {
3.     private StepExecution stepExecution;
4.     private Boolean stop = Boolean.FALSE;
5.
6.     public void beforeStep(StepExecution stepExecution) {
7.         this.stepExecution = stepExecution;
8.     }
9.
10.    public void beforeRead() {
11.        if(isStop()) {
12.            this.stepExecution.setTerminateOnly();
13.        }
14.        .....
15.    }
}
```

其中，6 行：获取 stepExecution，并保存在拦截器的实例中。

11 行：判断是否终止任务，如果终止调用操作 stepExecution.setTerminateOnly()实现配置定义的拦截器，配置文件参见代码清单 4-38。

代码清单 4-38 作业步中配置业务停止拦截器

```
1. <job id="chunkBusinessJob">
2.   <step id="chunkBusinessStep">
3.     <tasklet>
4.       <chunk reader="reader" writer="writer" commit-interval="10" />
5.       <listeners>
6.         <listener ref="stopListener"></listener>
7.       </listeners>
```

```
8.          </tasklet>
9.      </step>
10. </job>
11. <bean:bean id="stopListener" class="com.juxtapose.example.ch04.stop.
StopStepListener" />
```

其中，6行：定义作业步 chunkBusinessStep 的拦截器，引用 **stopListener**。

11行：定义拦截器对象。

执行测试类 `test.com.juxtapose.example.ch04.JobLaunchStopBusiness`，可以在控制台看出任务被友好的方式停止掉。

基于拦截器的机制终止任务，可以保证业务代码专注于完成业务操作，保证了代码的优雅性。可以在拦截器 `org.springframework.batch.core.StepListener` 及子类的拦截器中进行作业的停止控制。

配置作业步 Step

Step 表示作业中的一个完整步骤，一个 Job 可以由一个或者多个 Step 组成。Step 包含了一个实际运行的批处理任务中所有必需的信息。如果忘记了 Job 和 Step 的关系，可以参考 3.3 节。Step 的声明使用 step 元素声明，每个 Step 由 tasklet 元素描述具体的作业，tasklet 可以是一个自定义的业务处理（需要实现接口 org.springframework.batch.core.step.tasklet.Tasklet），也可以使用 chunk 元素描述面向块的业务处理，一个典型的 chunk 包含 read、process 和 write 三个操作。

一个典型的 Step 声明参见代码清单 5-1。

代码清单 5-1 典型的 Step 声明

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

step、tasklet、chunk、read、processor、write 之间的关系如图 5-1 所示。Step 的作用域最大，然后是 Tasklet，接下来为 Chunk，在每个 Chunk 中可以定义 read、process、write。

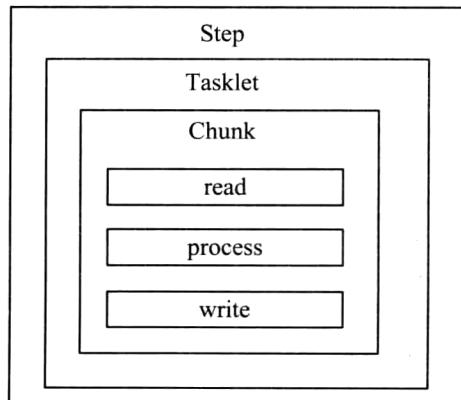


图 5-1 step、tasklet、chunk、read、processor、write 关系图

5.1 配置 Step

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

Step 主要属性包括 id（作业步唯一标识）、next（下一个执行的作业步）、parent（指定该作业步的父作业步）、job-repository（定义作业仓库）、abstract（定义作业步是否是抽象的）。图 5-2 展示了 Job 中 Step 属性的 Schema 的定义，图 5-3 展示了配置文件中顶层 Step 属性的 Schema 的定义，Step 属性说明参见表 5-1。

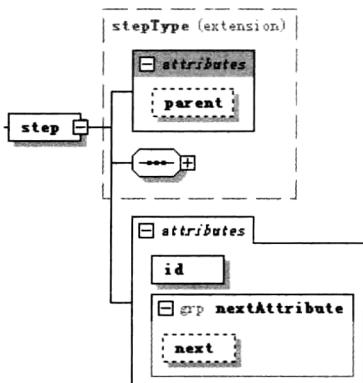


图 5-2 Step 属性 Schema 的定义

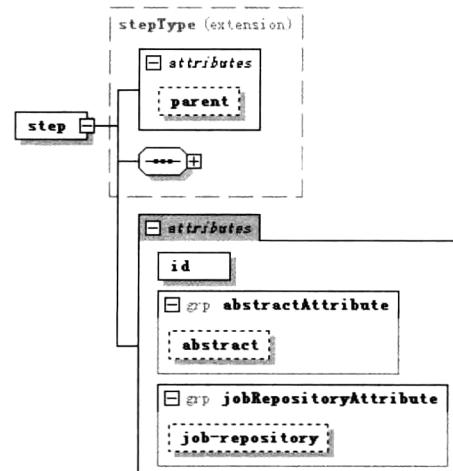


图 5-3 Step 顶层属性 Schema 的定义

表 5-1 Step 属性说明

属性	说 明	默 认 值
id	Step 的唯一标识，在整个运行上下文中不允许重复	
next	当前 Step 执行完成后，next 元素指定下一个需要执行的 Step	
parent	定义当前 Step 的父 Step。Step 可以从其他 Step 继承。通常在父 Step 中定义共有的属性，在子 Step 中定义特有的属性。 如果父子都有的属性以子类中定义的为准，即子类属性会覆盖父类属性	
job-repository	定义该 Step 运行期间使用的 Step 仓库，默认使用名字为 jobRepository 的 Bean。 该属性只有在 Step 为顶层元素时候生效	jobRepository
abstract	定义当前 Step 是否是抽象的。True 表示当前 Step 是抽象的，不能被实例化。 该属性只有在 Step 为顶层元素时候生效	false

Step 主要子元素包括 tasklet（定义作业步的执行逻辑）、partition（任务分区）、job（引用独立 Job 作为作业步）、flow（引用 Flow 作为作业步）、next（下一个执行的作业步）、

`stop`（退出当前任务）、`end`（结束当前任务）、`fail`（使当前任务失败）、`listeners`（定义作业步拦截器）。图 5-4 展示了 Step 子元素的 Schema 的定义，Step 元素说明参见表 5-2。

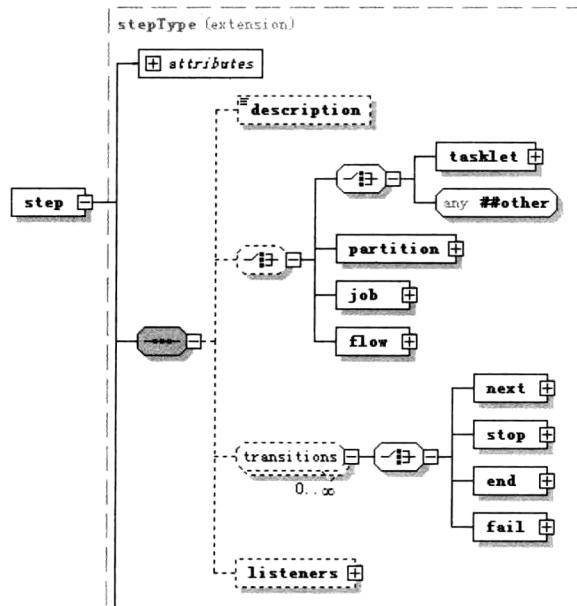


图 5-4 Step 子元素 Schema 的定义

表 5-2 Step 子元素说明

属性	说明
<code>tasklet</code>	定义具体作业步的执行逻辑
<code>partition</code>	定义当前任务是分区执行的，需要使用 <code>partition</code> 元素来声明 Step
<code>job</code>	引用独立配置的 Job 作为任务
<code>flow</code>	引用独立配置的 Flow 作为任务
<code>next</code>	根据退出状态定义下一步需要执行的 Step
<code>stop</code>	根据退出状态决定是否退出当前的任务，同时 Job 也会停止，作业状态为“STOPPED”
<code>end</code>	根据退出状态决定是否结束当前的任务，同时 Job 也会停止，作业状态为“COMPLETED”
<code>fail</code>	根据退出状态决定是否当前的任务失败，同时 Job 也会停止，作业状态为“FAILED”
<code>listeners</code>	定义作业步的拦截器

5.1.1 Step 抽象与继承

Spring Batch 框架支持抽象的 Step 定义和 Step 的继承特性。通过定义抽象的 Step 可以将 Step 的共性进行抽取，形成父类的 Step 定义；然后各个具体的 Step 可以继承父类 Step 的特

性，并定义自己的属性。通过 Step 的属性 abstract 可以定义抽象的 Step（抽象的 Step 不能被实例化执行），通过属性 parent 可以指定当前 Step 的父 Step。

抽象 Step

通过 abstract 属性可以指定 Step 为抽象的 Step。抽象的 Step 不能被实例化，只能作为其他 Step 的父。通常可以在抽象 Step 中定义全局性的配置，供子类使用。

代码清单 5-2 展示了典型的抽象 Step 的定义。

代码清单 5-2 抽象 Step 定义

```
1.      <step id="abstractParentStep" abstract="true">
2.          <tasklet>
3.              <chunk commit-interval="5" />
4.          </tasklet>
5.      </step>
```

其中，1 行：通过 abstract 属性，指定当前的 Step 为抽象的。

继承 Step

通过 parent 属性可以指定当前 Step 的父类，类似于 Java 世界中的继承一样，子类 Step 具有父类中定义的所有属性能力。子类继承时候，可以从抽象 Step 继承，也可以从普通的 Step 中继承。图 5-5 展示了 Step 间继承关系。

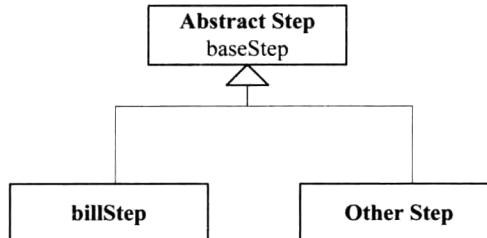


图 5-5 Step 继承

如果子类和父类都定义了相同的属性值，以子类中定义的为准。

代码清单 5-3 展示了典型的继承 Step 的定义。

代码清单 5-3 抽象 Step、继承 Step 配置

```
1.      <step id="parentStep" parent="abstractParentStep">
2.          <tasklet transaction-manager="transactionManager" allow-start-if-
   complete="true">
3.              <chunk reader="csvItemReader" writer="csvItemWriter" commit-
   interval="10" />
4.          </tasklet>
5.      </step>
```

```
7.      <job id="billJob">
8.          <step id="billStep" parent="parentStep">
9.              <tasklet>
10.                  <chunk processor="creditBillProcessor" commit-interval="2">
11.                      </chunk>
12.                  </tasklet>
13.          </step>
14.      </job>
```

其中，1行：定义 parentStep 的父为 abstractParentStep，继承抽象的 Step。

3行：定义了 reader、writer、commit-interval 三个属性，这样子类 billStep 不用重复定义这三个属性，可直接使用父类 parentStep 中的定义。

10行：子类 billStep 定义属性 processor、commit-interval 两个属性，因为属性 commit-interval 在父类和子类都有定义，因此使用子类 billStep 中定义的属性 commit-interval。

5.1.2 Step 执行拦截器

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

StepExecutionListener 接口声明参见代码清单 5-4。

代码清单 5-4 StepExecutionListener 接口声明

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

其中，2行：表示在 Step 执行之前调用该方法。

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

为 chunkStep 配置拦截器，参见代码清单 5-5。

代码清单 5-5 chunkStep 拦截器配置

```
1.      <job id="chunkJob">
2.          <step id="chunkStep">
3.              <tasklet>
4.                  <chunk reader="reader" writer="writer" commit-interval="10" />
5.              </tasklet>
6.              <listeners>
7.                  <listener ref="sysoutListener"></listener>
8.                  <listener ref="sysoutAnnotationListener"></listener>
9.              </listeners>
10.         </step>
11.     </job>
```