

目 录

致谢

Spring Batch参考文档中文版

Spring Batch介绍

背景

使用场景

Spring Batch架构

通用批处理的指导原则

批处理策略

非官方示例(CVS_MySQL)

Spring Batch 3.0新特性

JSR-352支持

改进的Spring Batch Integration模块

升级到支持Spring 4和Java 8

JobScope支持

SQLite支持

批处理专业术语

配置并运行Job

Configuring a Job

Java Config

Configuring a JobRepository

Configuring a JobLauncher

Running a Job

Meta-Data 高级用法

配置Step

面向块的流程

ItemReaders和ItemWriters

ItemReader

ItemWriter

ItemProcessor

ItemStream

代理模式与Step注册

纯文本文件

字段集合

FlatFileItemReader

FlatFileItemWriter

XML条目读写器

StaxEventItemReader

StaxEventItemWriter

多个输入文件

数据库Database

基于游标的ItemReaders

ItemReaders分页

数据库ItemWriters

重用已有服务

输入校验

不参与持久化的字段

自定义ItemReaders和ItemWriters

自定义ItemReader示例

自定义ItemWriter示例

扩展与并行处理

多线程 Step

并行 Steps

远程分块

分区

重复执行

重试处理

重试模板

重试策略

补偿策略

监听器

声明式重试

单元测试

创建一个单元测试

点对点的批处理任务测试

测试各个步骤

测试Step-Scoped组件

验证输出文件

模拟域对象

通用批处理模式

日志项处理和失败

业务原因手工停止任务

添加一个Footer记录

基于ItemReaders的driving query

多行记录

执行系统命令

当没有找到输入时Step处理完成

将数据传递给Future Steps

JSR352支持

General Notes

Setup

依赖注入

Batch Properties

处理模型

Running a job

Contexts

Step Flow

扩展 JSR-352 批处理作业

测试

Spring Batch Integration模块

附录A

附录B

附录C

术语表

致谢

当前文档《Spring Batch参考文档中文版》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建,生成于 2018-04-15。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能,以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理,书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候,发现文档内容有不恰当的地方,请向我们反馈,让我们共同携手,将知识准确、高效且有效地传递给每一个人。

同时,如果您在日常生活、工作和学习中遇到有价值有营养的知识文档,欢迎分享到 书栈(BookStack.CN),为知识的传承献上您的一份力量!

如果当前文档生成时间太久,请到 书栈(BookStack.CN) 获取最新的文档,以跟上知识更新换代的步伐。

文档地址: <http://www.bookstack.cn/books/SpringBatchReferenceCN>

书栈官网: <http://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享,让知识传承更久远! 感谢知识的创造者,感谢知识的分享者,也感谢每一位阅读到此处的读者,因为我们都将成为知识的传承者。

Spring Batch参考文档中文版

- [Spring Batch参考文档中文版](#)

Spring Batch参考文档中文版

原作者: Lucas Ward , Dave Syer , Thomas Risberg , Robert Kasanicky , Dan Garrette , Wayne Lund , Michael Minella , Chris Schaefer , Gunnar Hillert

版本号: 3.0.1.RELEASE 查看原文请[点击此处](#)。或者 [查看HTML单页面版](#)

在线预览版本地址: [在线预览](#)

原文版权属于 © 2009, 2010, 2011, 2012, 2013, 2014 GoPivotal公司, 保留所有权利。本文档的所有分发给其他人的拷贝, 都应该包括上述的版权信息, 而且不能收取任何费用, 无论分发的是电子版还是打印版。

本翻译文档由Spring Batch参考中文翻译组于2014年7月21日开始翻译, 版权由原文版权所有者和翻译组共同所有。参与翻译请发邮件到: kimmking@163.com。

翻译组成员列表如下:

- kimmking(<http://blog.csdn.net/kimmking>)
- 铁锚(<http://blog.csdn.net/renfufei>)
- stormhouse(<http://stormhouse.github.io/>)
- Sean(<http://caochengbo.iteye.com>)
- 会亮(<http://blog.csdn.net/ohalo>)

完成百分比: 14/17 ~ 82%

章节	页数	翻译者	状态	计划完成时间
01	-	铁锚	done	2014-08-10
02	-	stormhouse	done	2014-08-02
03	-	Sean	done	2014-08-03
04	-	Sean	done	2014-08-20
05	-	stormhouse	done	2014-08-20
06	-	铁锚	done	2015-04-15
07	-	铁锚	done	2014-09-10
08	-	翟伟	done	2015-04-15
09	-	赵辉亮	done	2015-04-16
10	-	赵辉亮	done	2015-07-16
11	-	翟伟	done	2015-11-11
12	-	铁锚	done	2015-06-01
13	-	-	-	-
14A	-	铁锚	done	2015-04-21
15B	-	-	-	-
16C	-	-	-	-
17G	-	铁锚	done	2015-04-25

Spring Batch介绍

- [Spring Batch介绍](#)

Spring Batch介绍

在企业领域,有很多应用和系统需要在生产环境中使用批处理来执行大量的业务操作.批处理业务需要自动地对海量数据信息进行各种复杂的业务逻辑处理,同时具备极高的效率,不需要人工干预.执行这种操作通常根据时间事件(如月末统计,通知或信件),或者定期处理那些业务规则超级复杂,数据量非常庞大的业务,(如保险赔款确定,利率调整),也可能是从内部/外部系统抓取到的各种数据,通常需要格式化、数据校验、并通过事务的方式处理到自己的数据库中.企业中每天通过批处理执行的事务多达数十亿.

Spring Batch是一个轻量级的综合性批处理框架,可用于开发企业信息系统中那些至关重要的数据批量处理业务.Spring Batch 基于POJO 和 Spring框架,相当容易上手使用,让开发者很容易地访问和利用企业级服务.Spring Batch不是调度(scheduling)框架.因为已经有很多非常好的企业级调度框架,包括商业性质的和开源的,例如Quartz, Tivoli, Control-M等.它是为了与调度程序一起协作完成任务而设计的,而不是用来取代调度框架的.

Spring Batch提供了大量的,可重用的功能,这些功能对大数据处理来说是必不可少的,包括 日志/跟踪(tracing),事务管理, 任务处理(processing)统计,任务重启, 忽略(skip),和资源管理等功能.此外还提供了许多高级服务和特性,使之能够通过优化(optimization) 和分片技术(partitioning techniques)来高效地执行超大型数据集的批处理任务.

Spring Batch是一个具有高可扩展性的框架,简单的批处理,或者复杂的大数据批处理作业都可以通过Spring Batch框架来实现.

背景

- 背景

背景

在开源项目及其相关社区把大部分注意力集中在基于web和SOA基于消息机制的框架中时，基于java的批处理框架却无人问津，尽管在企业IT环境中一直都有这种批处理的需求。但因为缺乏一个标准的、可重用的批处理框架导致在企业客户的IT系统中存在着很多一次编写，一次使用的版本，以及很多不同的内部解决方案。

SpringSource和Accenture (埃森哲)致力于通过合作来改善这种状况。埃森哲在实现批处理架构上有着丰富的产业实践经验，SpringSource有深入的技术开发积累，背靠Spring框架提供的编程模型，意味着两者能够结合成为默契且强大的合作伙伴，创造出高质量的、市场认可的企业级java解决方案，填补这一重要的行业空白。两家公司目前也正着力于开发基于spring的批处理解决方案，为许多客户解决类似的问题。这同时提供了一些有用的额外的细节和以及真实环境的约束，有助于确保解决方案能够被客户用于解决实际的问题。基于这些原因，SpringSource和埃森哲一起合作开发Spring Batch。

埃森哲已经贡献了先前自己的批处理体系结构框架，这个框架基于数十年宝贵的经验并基于最新的软件平台(如COBOL/Mainframe, C++/Unix 及现在非常流行的Java平台)来构建Spring Batch项目，Spring Batch未来将会由开源社区提交者来驱动项目的开发，增强，以及未来的路线图。

埃森哲咨询公司与SpringSource合作的目标是促进软件处理方法、框架和工具标准化改进，并在创建批处理应用时能够持续影响企业用户。企业和政府机构希望为他们提供标准的、经验验证过的解决方案，而他们的企业系统也将受益于Spring Batch。

使用场景

- 使用场景
 - 业务场景
 - 技术目标

使用场景

典型的批处理程序通常是从数据库、文件或队列中读取大量数据，然后通过某些方法处理数据，最后将处理好格式的数据写回库中。通常SpringBatch工作在离线模式下，不需要用户干预、就能自动进行基本的批处理迭代，进行类似事务方式的处理。批处理是大多数IT项目的一个组成部分，而Spring Batch是唯一能够提供健壮的企业级扩展性的批处理开源框架。

业务场景

- 定期提交批处理任务
- 并发批处理：并行执行任务
- 分阶段，企业消息驱动处理
- 高并发批处理任务
- 失败后手动或定时重启
- 按顺序处理任务依赖(使用工作流驱动的批处理插件)
- 局部处理：跳过记录(例如在回滚时)
- 完整的批处理事务：因为可能有小数据量的批处理或存在存储过程/脚本

技术目标

- 利用Spring编程模式：使开发者专注于业务逻辑，让框架解决基础功能
- 在基础架构、批处理执行环境、批处理应用之间有明确的划分
- 以接口形式提供通用的核心服务，以便所有项目都能使用
- 提供简单的默认实现，以实现核心执行接口的“开箱即用”
- 易于配置、定制和扩展服务，基于spring框架的各个层面
- 所有的核心服务都可以很容易地扩展与替换，却不会影响基础系统层。
- 提供一个简单的部署模型，通过Maven编译，将应用程序与框架的JAR包完全分离

Spring Batch架构

- [Spring Batch架构](#)

Spring Batch架构

Spring Batch 设计时充分考虑了可扩展性和各类终端用户。下图显示了Spring Batch的架构层次示意图, 这种架构层次为终端用户开发者提供了很好的扩展性与易用性。

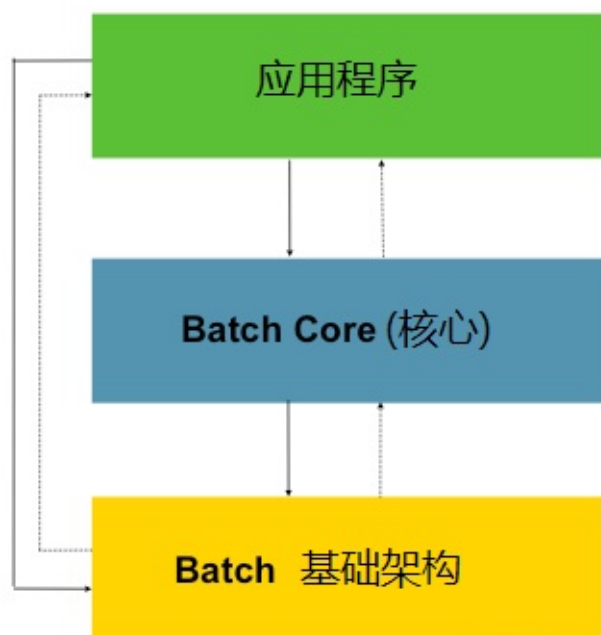


图1.1: Spring Batch 分层架构

Spring Batch 架构主要分为三类高级组件：应用层(Application)，核心层(Core) 和基础架构层(Infrastructure)。

应用层(Application)包括开发人员用Spring batch编写的所有批处理作业和自定义代码。

Batch核心(Batch Core) 包含加载和控制批处理作业所必需的核心类。包括 **JobLauncher**, **Job**, 和 **Step** 的实现。

应用层(Application) 与 核心等(Core)都构建在通用基础架构层之上。基础架构包括通用的 readers(**ItemReader**) 和 writers(**ItemWriter**), 以及 services (如重试模块 **RetryTemplate**), 可以被应用层和核心层所使用。

通用批处理的指导原则

- [通用批处理的指导原则](#)

通用批处理的指导原则

下面是一些关键的指导原则, 在构建批处理解决方案可以参考:

- 批处理架构通常会影响在线服务的架构, 反之亦然。设计架构和环境时请尽可能使用公共的模块。
- 尽可能的简化, 避免在单个批处理应用中构建复杂的逻辑结构。
- 尽可能在数据存放的地方处理这些数据, 反之亦然(即, 各自负责处理自己的数据)。
- 尽可能少的使用系统资源, 尤其是I/O。尽可能多地在内存中执行大部分操作。
- 审查应用程序I/O(分析SQL语句)以避免不必要的物理I/O。特别是以下四个常见的缺陷(flaws)需要避免:
 - 在每个事务中都读(所有并不需要的)数据, 并缓存起来;
 - 多次读取/查询同一事务中已经读取过的数据;
 - 引起不必要的表或索引扫描;
 - 在SQL语句的WHERE子句中不指定过滤条件。
- 在同一个批处理不要做两次一样的事。例如, 如果你需要报表的数据汇总, 请在处理每一条记录时使用增量来存储, 尽可能不要去遍历一次同样的数据。
- 在批处理程序开始时就分配足够的内存, 以避免运行过程中再执行耗时的内存分配。
- 总是将数据完整性假定为最坏情况。插入适当的检查和数据校验以保持数据完整性(integrity)。
- 如有可能, 请为内部校验实现checksum。例如, 平面文件应该有一条结尾记录, 说明文件中的总记录数和关键字段的集合(aggregate)。
- 尽可能早地在模拟生产环境下使用真实的数据量, 进行计划和执行压力测试。
- 在大型批处理系统中, 备份会是一个很大的挑战, 特别是 7x24小时不间断的在线服务系统。数据库备份通常在设计时就考虑好了, 但是文件备份也应该提升到同养的重要程度。如果系统依赖于文本文件, 文件备份程序不仅要正确设置和形成文档, 还要定期进行测试。

批处理策略

- [批处理策略](#)

批处理策略

为了辅助批处理系统的设计和实现、应该通过结构示意图和代码实例的形式为设计师和程序员提供基础的批处理程序构建模块和以及处理模式。

在设计批处理Job时,应该将业务逻辑分解成一系列的步骤,使每个步骤都可以利用以下的标准构建模块来实现:

- 转换程序(Conversion Applications): 由外部系统提供或需要写入到外部系统的各种类型的文件,我们都需要为其创建一个转换程序,用来将所提供的事务记录转换成符合要求的标准格式.这种类型的批处理程序可以部分或全部由转换工具模块组成(translation utility modules)(参见 Basic Batch Services,基本批处理服务)。
- 验证程序(Validation Applications): 验证程序确保所有输入/输出记录都是正确和一致的.验证通常基于文件头和结尾信息,校验和(checksums)以及记录级别的交叉验证算法。
- 提取程序(Extract Applications): 这种程序从数据库或输入文件读取一堆记录,根据预定义的规则选取记录,并将选取的记录写入到输出文件。
- 提取/更新程序(Extract/Update Applications): 这种程序从数据库或输入文件读取记录,并将输入的每条记录都更新到数据库,或记录到输出文件。
- 处理和更新程序(Processing and Updating Applications): 这种程序对从提取或验证程序传过来的输入事务记录进行处理.这些处理通常包括从数据库读取数据,有可能更新数据库,并创建输出记录。
- 输出/格式化程序(Output/Format Applications): 这种程序从输入文件中读取信息,将数据重组成为标准格式,并打印到输出文件,或者传输给另一个程序或系统。

因为业务逻辑不能用上面介绍的这些标准模块来完成,所以还需要另外提供一个基本的程序外壳(application shell)。

除了这些主要的模块,每个应用还可以使用一到多个标准的实用程序环节(standard utility steps),如:

- Sort 排序,排序程序从输入文件读取记录,并根据记录中的某个key字段重新排序,然后生成输出文件.排序通常由标准的系统实用程序来执行。
- Split 拆分,拆分程序从单个输入文件中读取记录,根据某个字段的值,将记录写入到不同的输出文件中.拆分可以自定义或者由参数驱动的(parameter-driven)系统实用程序来执行。
- Merge 合并,合并程序从多个输入文件读取记录,并将组合后的数据写入到单个输出文件中.合并可以自定义或者由参数驱动的(parameter-driven)系统实用程序来执行。

批处理程序也可以根据输入来源分类:

- 数据库驱动(Database-driven)的应用程序, 由从数据库中获取的行或值驱动.
- 文件驱动(File-driven)的应用程序, 是由从文件中获取的值或记录驱动的.
- 消息驱动(Message-driven)的应用程序由从消息队列中检索到的消息驱动.

所有批处理系统的基础都是处理策略. 影响策略选择的因素包括: 预估的批处理系统容量, 在线并发或与另一个批处理系统的并发量, 可用的批处理时间窗口(随着越来越多的企业想要全天候(7x24小时)运转, 所以基本上没有明确的批处理窗口).

典型的批处理选项包括:

- 在一个批处理窗口中执行常规离线批处理
- 并发批处理/在线处理
- 同一时刻有许多不同的批处理(runs or jobs)在并行执行
- 分区(即同一时刻, 有多个实例在处理同一个job)
- 上面这些的组合

上面列表中的顺序代表了批处理实现复杂性的排序, 在同一个批处理窗口的处理最简单, 而分区实现最复杂.

商业调度器可能支持上面的部分/或所有类型.

下面的部分将详细讨论这些处理选项. 需要特别注意的是, 批处理所采用的提交和锁定策略将依赖于处理执行的类型, 作为最佳实践, 在线锁策略应该使用相同的原则. 因此, 在设计批处理整体架构时不能简单地拍脑袋决定(译注: 即需要详细的论证和分析).

锁策略可以只使用普通的数据库锁, 也可以在架构中实现自定义的锁服务. 锁服务将跟踪数据库锁定(例如在一个专用的数据库表(db-table)中存储必要的信息), 然后在应用程序请求数据库操作时授予权限或拒绝. 重试逻辑也可以通过这种架构实现, 以避免批处理作业因为资源锁定的情况而失败.

1. 在一个批处理窗口中的常规处理 对于运行在一个单独批处理窗口中的简单批处理, 更新的数据对在线用户或其他批处理来说并没有实时性要求, 也没有并发问题, 在批处理运行完成后执行单次提交即可.

大多数情况下, 一种更健壮的方法会更合适. 要记住的一件事是, 批处理系统会随着时间的流逝而增长, 包括复杂度和需要处理的数据量. 如果没有合适的锁定策略, 系统仍然依赖于一个单一的提交点, 则修改批处理程序会是一件痛苦的事情. 因此, 即使是最简单的批处理系统, 也应该为重启-恢复(restart-recovery)选项考虑提交逻辑, 更不用说下面涉及到的那些更复杂情况下的信息.

2. 并发批处理/在线处理 批处理程序处理的数据如果会同时被在线用户更新, 就不应该锁定在线用户需要的所有任何数据(不管是数据库还是文件), 即使只需要锁定几秒钟的时间. 还应该每处理一批事务就提交一次数据库. 这减少了其他程序不可用的数据, 也压缩了数据不可用的时间.

减少物理锁的另一个选择是实现一个行级的逻辑锁, 通过使用乐观锁模式或悲观锁模式.

- 乐观锁假设记录争用的可能性很低. 这通常意味着并发批处理和在线处理所使用的每个数据表中都

有一个时间戳列。当程序读取一行进行处理时，同时也获得对应的时间戳。当程序处理完该行以后尝试更新时，在update操作的WHERE子句中使用原来的时间戳作为条件。如果时间戳相匹配，则数据和时间戳都更新成功。如果时间戳不匹配，这表明在本程序上次获取和此次更新这段时间内已经有另一个程序修改了同一条记录，因此更新不会被执行。

- 悲观锁定策略假设记录争用的可能性很高，因此在检索时需要获得一个物理锁或逻辑锁。有一种悲观逻辑锁在数据表中使用一个专用的lock-column列。当程序想要为更新目的而获取一行时，它在lock column上设置一个标志。如果为某一行设置了标志位，其他程序在试图获取同一行时将会逻辑上获取失败。当设置标志的程序更新该行时，它也同时清除标志位，允许其他程序获取该行。请注意，在初步获取和初次设置标志位这段时间内必须维护数据的完整性，比如使用数据库锁（eg., SELECT FOR UPDATE）。还请注意，这种方法和物理锁都有相同的缺点，除了它在构建一个超时机制时比较容易管理，比如记录而用户去吃午餐了，则超时时间到了以后锁会被自动释放。

这些模式并不一定适用于批处理，但他们可以被用在并发批处理和在线处理的情况下（例如，数据库不支持行级锁）。作为一般规则，乐观锁更适用于在线应用，而悲观锁更适用于批处理应用。只要使用了逻辑锁，那么所有访问逻辑锁保护的数据的程序都必须采用同样的方案。

请注意，这两种解决方案都只锁定(address locking)单条记录。但很多情况下我们需要锁定一组相关的记录。如果使用物理锁，你必须非常小心地管理这些以避免潜在的死锁。如果使用逻辑锁，通常最好的解决办法是创建一个逻辑锁管理器，使管理器能理解你想要保护的逻辑记录分组(groups)，并确保连贯和没有死锁(non-deadlocking)。这种逻辑锁管理器通常使用其私有的表来进行锁管理、争用报告、超时机制等等。

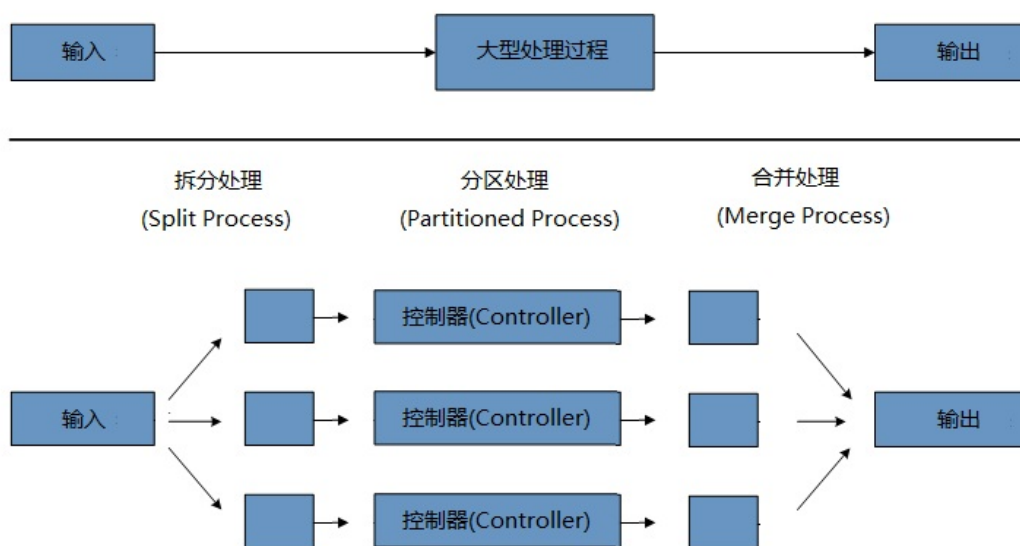
3. 并行处理 并行处理允许多个批处理运行(run, 名词, 大意为运行中的程序)/任务(job)同时并行地运行，以使批处理总运行时间降到最低。如果多个任务不使用同一个文件、数据表、索引空间时这并不算什么问题。如果确实存在共享和竞争，那么这个服务就应该使用分区数据来实现。另一种选择是使用控制表来构建一个架构模块以维护他们之间的相互依赖关系。控制表应该为每个共享资源分配一行记录，不管这些资源是否被某个程序所使用。执行并行作业的批处理架构或程序随后将查询这个控制表，以确定是否可以访问所需的资源。

如果解决了数据访问的问题，并行处理就可以通过使用额外的线程来并行实现。在传统的大型主机环境中，并行作业类上通常被用来确保所有进程都有充足的CPU时间。无论如何，解决方案必须足够强劲，以确保所有正在运行的进程都有足够的时间片。

并行处理的其他关键问题还包括负载平衡以及一般系统资源的可用性(如文件、数据库缓冲池等)。还要注意控制表自身也可能很容易变成一个至关重要的资源(即可能发生严重竞争?)。

4. 分区(Partitioning) 分区技术允许多版本的大型批处理程序并发地(concurrently)运行。这样做的目的是减少超长批处理作业过程所需的时间。可以成功分区的过程主要是那些可以拆分的输入文件 和/或 主要的数据库表被分区以允许程序使用不同的数据来运行。

此外，被分区的过程必须设计为只处理分配给他的数据集。分区架构与数据库设计和数据库分区策略是密切相关的。请注意，数据库分区并不一定指数据库需要在物理上实现分区，尽管在大多数情况下这是明智的。下面的图片展示了分区的方法：



系统架构应该足够灵活, 以允许动态配置分区数量。自动控制 and 用户配置都应该纳入考虑范围。自动配置可以根据参数来决定, 例如输入文件大小 和/或 输入记录的数量。

4.1 分区方法 下面列出了一些可能的分区方法。选择哪种分区方法要根据具体情况来决定。

1. 使用固定值来分解记录集

这涉及到将输入的记录集合分解成偶数个部分 (例如10份, 这样每部分是整个数据集的十分之一)。每个部分稍后由一个批处理/提取程序实例来处理。

为了使用这种方法, 需要在预处理时将记录集拆分。拆分的结果有一个最大值和最小值位置, 这两个值可以用作限制每个批处理/提取程序处理部分的输入。

预处理可能是一个很大的开销, 因为它必须计算并确定的每部分数据集的边界。

2. 根据关键列 (Key Column) 分解

这涉及到将输入记录按照某个关键列来分解, 比如定位码 (location code), 并将每个键分配给一个批处理实例。为了达到这个目标, 也可以使用列值。

3. 根据分区表决定分配给哪一个批处理实例 (详情见下文)。

4. 根据值的一部分决定分配给哪个批处理实例的值 (例如值 0000-0999、1000-1999 等)

在使用第1种方法时, 新值的添加将意味着需要手动重新配置批处理/提取程序, 以确保新值被添加到某个特定的实例。

在使用第2种方法时, 将确保所有的值都会被某个批处理作业实例处理到。然而, 一个实例处理的值的

数量依赖于列值的分布(即可能存在大量的值分布在0000-0999范围内,而在1000-1999范围内的值却很少).如果使用这种方法,设计时应该考虑到数据范围的切分.

在这两种方法中,并不能将指定给批处理实例的记录实现最佳均匀分布.批处理实例的数量并不能动态配置.

5. 根据视图来分解

这种方法基本上是根据键列来分解,但不同的是在数据库级进行分解.它涉及到将记录集分解成视图.这些视图将被批处理程序的各个实例在处理时使用.分解将通过数据分组来完成.

使用这个方法时,批处理的每个实例都必须为其配置一个特定的视图(而非主表).当然,对于新添加的数据,这个新的数据分组必须被包含在某个视图中.也没有自动配置功能,实例数量的变化将导致视图需要进行相应的改变.

6. 附加的处理指示器

这涉及到输入表一个附加的新列,它充当一个指示器.在预处理阶段,所有指示器都被标志为未处理.在批处理程序获取记录阶段,只会读取被标记为未处理的记录,一旦他们被读取(并加锁),它们就被标记为正在处理状态.当记录处理完成,指示器将被更新为完成或错误.批处理程序的多个实例不需要改变就可以开始,因为附加列确保每条纪录只被处理一次.

使用该选项时,表上的I/O会动态地增长.在批量更新的程序中,这种影响被降低了,因为写操作是必定要进行的.

7. 将表提取到平面文件

这包括将表中的数据提取到一个文件中.然后将这个文件拆分成多个部分,作为批处理实例的输入.

使用这个选项时,将数据提取到文件中,并将文件拆分的额外开销,有可能抵消多分区处理(multi-partitioning)的效果.可以通过改变文件分割脚本来实现动态配置.

8. 使用哈希列(*Hashing Column*)

这个计划需要在数据库表中增加一个哈希列(key/index)来检索驱动(driver)记录.这个哈希列将有一个指示器来确定将由批处理程序的哪个实例处理某个特定的行.例如,如果启动了三个批处理实例,那么“A”指示器将标记某行由实例1来处理,“B”将标记着将由实例2来处理,以此类推.

稍后用于检索记录的过程(procedure,程序)将有一个额外的WHERE子句来选择以一个特定指标标记的所有行.这个表的insert需要附加的标记字段,默认值将是其中的某一个实例(例如“A”).

一个简单的批处理程序将被用来更新不同实例之间的重新分配负载的指标.当添加足够多的新行时,这个批处理会被运行(在任何时间,除了在批处理窗口中)以将新行分配给其他实例.

批处理应用程序的其他实例只需要像上面这样的批处理程序运行着以重新分配指标,以决定新实例的数

量。

4.2 数据库和应用程序设计原则

如果一个支持多分区(multi-partitioned)的应用程序架构,基于数据库采用关键列(key column)分区方法拆成的多个表,则应该包含一个中心分区仓库来存储分区参数。这种方式提供了灵活性,并保证了可维护性。这个中心仓库通常只由单个表组成,叫做分区表。

存储在分区表中的信息应该是静态的,并且只能由DBA维护。每个多分区程序对应的单个分区有一行记录,组成这个表。这个表应该包含这些列: 程序ID编号,分区编号(分区的逻辑ID),一个分区对应的关键列(keycolumn)的最小值,分区对应的关键列的最大值。

在程序启动时,应用程序架构(Control Processing Tasklet,控制处理微线程)应该将程序id和分区号传递给该程序。这些变量被用于读取分区表,来确定应用程序应该处理的数据范围(如果使用关键列的话)。另外分区号必须在整个处理过程中用来:

- 为了使合并程序正常工作,需要将分区号添加到输出文件/数据库更新
- 向框架的错误处理程序报告正常处理批处理日志和执行期间发生的所有错误

4.3 尽可能杜绝死锁

当程序并行或分区运行时,会导致数据库资源的争用,还可能会发生死锁(Deadlocks)。其中的关键是数据库设计团队在进行数据库设计时必须考虑尽可能消除潜在的竞争情况。

还要确保设计数据库表的索引时考虑到性能以及死锁预防。

死锁或热点往往发生在管理或架构表上,如日志表、控制表,锁表(lock tables)。这些影响也应该纳入考虑。为了确定架构可能的瓶颈,一个真实的压力测试是至关重要的。

要最小化数据冲突的影响,架构应该提供一些服务,如附加到数据库或遇到死锁时的等待-重试(wait-and-retry)间隔时间。这意味着要有一个内置的机制来处理数据库返回码,而不是立即引发错误处理,需要等待一个预定的时间并重试执行数据库操作。

4.4 参数传递和校验

对程序开发人员来说,分区架构应该相对透明。框架以分区模式运行时应该执行的相关任务包括:

- 在程序启动之前获取分区参数
- 在程序启动之前验证分区参数
- 在启动时将参数传递给应用程序

验证(validation)要包含必要的检查,以确保:

- 应用程序已经足够涵盖整个数据的分区
- 在各个分区之间没有遗漏断代(gaps)

如果数据库是分区的,可能需要一些额外的验证来保证单个分区不会跨越数据库的片区。

体系架构应该考虑整合分区(partitions). 包括以下关键问题:

- 在进入下一个任务步骤之前是否所有的分区都必须完成?
- 如果一个分区Job中止了要怎么处理?

非官方示例(CVS_MySQL)

- [Spring Batch示例：读取CSV文件并写入MySQL数据库](#)
 - [作业与分块：Spring Batch 范例](#)
 - [读取并处理CSV文件](#)
 - [写入数据库](#)
 - [用 application context 将上下文组装起来](#)
 - [定义job](#)
 - [构建并运行项目](#)
 - [Spring Batch 连接数据库](#)
 - [使用 Spring Batch 执行批量处理](#)
 - [创建多个processors](#)
 - [Tasklets\(微任务\)](#)
 - [弹性\(Resiliency\)](#)
 - [Skipping Items\(跳过某项\)](#)
 - [重试 \(Retrying Items\)](#)
 - [重启 job](#)
 - [总结](#)

Spring Batch示例：读取CSV文件并写入MySQL数据库

原文链接：[Reading and writing CVS files with Spring Batch and MySQL](#)

原文作者：[Steven Haines](#) - 技术架构师

下载本教程的源代码：

[SpringBatch-CSV演示代码](#)

用批处理程序来操作动辄上GB的数据很可能会拖死整个系统,但现在我们可以通过Spring Batch将其拆解为多个小块(chunk)。Spring框架中的 Spring Batch 模块, 是专门设计了用来对各种类型文件进行批处理的工程。 本文先从一个简单的作业(Job)入手 — 将从CSV文件中读取产品列表, 并导入到MySQL数据库中; 然后我们一起研究 Spring Batch 的批处理特性: 如单/多处理单元(processors), 以及多个微线程(tasklets); 最后简单介绍一下 Spring Batch 提供的用来处理 忽略记录(skipping), 重试记录(retrying),以及批处理作业重启(restarting) 的弹性工具(resiliency tools)。

如果你曾在Java企业系统中用批处理来处理过十万级以上的数据, 那么你肯定知道工作负载是怎么回事。 批处理系统需要处理大量的数据, 内部消化单条记录失败的情况, 还要管理中断, 在重启后也不去重复执行已经处理过的部分。

对于初学者来说,下面是一些需要用到批处理的场景,在这些情况下如果使用Spring Batch,则会节省大量的时间:

- 接收的文件中缺少了一部分汇总信息,需要读取并解析整个文件,调用服务来获取缺少的那部分信息,然后写入输出文件,供另一个批处理程序使用。
- 如果在执行代码时发生错误,则将失败信息写入数据库中。有一个专门的程序每隔15分钟来遍历一次失败信息,如果标记为可以重试,则完成后再重复执行一次。
- 在工作流中,约定由其他系统在收到消息事件时,来调用某个特定的服务。但如果其他系统没有调用这个服务,那么一段时间后需要自动清理过期数据,以免影响正常的业务流程。
- 每天收到员工信息更新文件,需要为新员工建立相关档案和账号(artifacts)。
- 生成自定义订单的服务。每天晚上需要执行批处理程序来生成清单文件,并将它们发送给相应的供应商。

作业与分块: Spring Batch 范例

Spring Batch 有很多组成部分,我们先来看批量作业中的核心部分。可以将一个作业分成以下3个步骤:

1. 读取数据
2. 对数据进行各种处理
3. 对数据进行写操作

例如,打开一个CSV格式的数据文件,对文件中的数据执行某种处理,然后将数据写入数据库。在Spring Batch中,需要配置一个 **reader** 来读取文件中的数据(每次一行),然后将数据传递给 **processor** 进行处理,处理完成之后会将结果收集并分组为“块 chunks”,然后把这些记录发送给 **writer**,在这里是插入到数据库中。如图1所示。



图1 Spring Batch批处理的基本逻辑

Spring Batch 提供了常见输入源的 reader 实现,极大地简化了批处理过程。例如 CSV文件, XML文件、数据库、文件中的JSON记录,甚至是 JMS; 同样也实现了对应的 writer。如有需要,创建自定义的 reader 和 writer 也很简单。

下载本教程的源代码:

[SpringBatch-CSV演示代码](#)

首先,让我们配置一个 file reader 来读取 CSV文件,将其内容映射到一个对象中,并将生成的对象插入数据库中。

读取并处理CSV文件

Spring Batch 的内置 reader, `org.springframework.batch.item.file.FlatFileItemReader`, 用来将文件解析为多个独立的行。需要纯文本文件的引用,文件开头要忽略的行数(比如标题,表头等信息),以及将单行转换为对象的 `line mapper`。行映射器需要一个分割字符串的分词器(`line tokenizer`),用来将一行拆分成多个组成字段,还需要一个 `field set mapper`,根据字段值构建对象。`FlatFileItemReader` 的配置如下所示:

清单1 一个Spring Batch 配置文件

```

1. <bean id="productReader" class="org.springframework.batch.item.file.FlatFileItemReader"
   scope="step">
2.
3.     <!-- <property name="resource" value="file:./sample.csv" /> -->
4.     <property name="resource" value="file:${jobParameters['inputFile']}" />
5.
6.     <property name="linesToSkip" value="1" />
7.
8.     <property name="lineMapper">
9.         <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
10.
11.             <property name="lineTokenizer">
12.                 <bean
13.                     class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
14.                         <property name="names" value="id,name,description,quantity" />
15.                     </bean>
16.                 </property>
17.             <property name="fieldSetMapper">
18.                 <bean
19.                     class="com.geekcap.javaworld.springbatchexample.simple.reader.ProductFieldSetMapper" />
20.                 </property>
21.             </bean>
22.         </property>
23.     </bean>

```

下面来看看这些组件。图2概括了它们之间的关系:

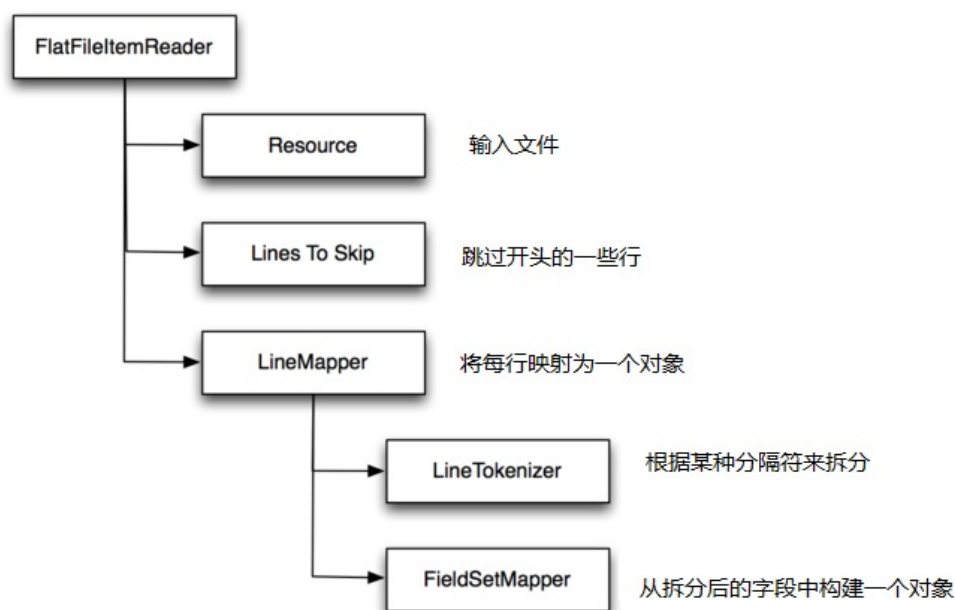


图2 FlatFileItemReader的组件

Resources: `resource` 属性指定了要读取的文件。注释部分的 `resource` 使用了文件的相对路径,也就是批处理作业工作目录下的 `sample.csv`。有趣的是使用了Job参数 `InputFile` : 使用 `job parameters` 则允许在运行时才根据需要决定相关参数。在使用 `import` 文件的情况下,在运行时才决定使用哪个参数比起在编译时就固定要灵活好用很多。(要一遍又一遍,五六七八遍导入同一个文件是相当无聊的!)

Lines to skip: 属性 `linesToSkip` 告诉 `file reader` 有多少标题行需要跳过。通常CSV文件的第一行包含标题信息,如列名称,所以本例中让 `reader` 跳过文件的第一行。

Line mapper: `lineMapper` 负责将一行记录转换为一个对象。依赖两个组件:

- `LineTokenizer` 指定了如何将一行拆分为多个字段。本例中列出了CSV文件中各列的列名。
- `fieldSetMapper` 根据字段值构造一个对象。示例中构建了一个 `Product` 对象,属性包括 `id`, `name`, `description`, 以及 `quantity`。

请注意,虽然Spring Batch提供了基础框架,但我们仍需要设置字段映射的逻辑。清单2显示了 `Product` 对象的源码,也就是我们准备构建的对象。

清单2 `Product.java`

```

1. package com.geekcap.javaworld.springbatchexample.simple.model;
2.
3. /**
4.  * 代表产品的简单值对象(POJO)
5.  */
  
```

```
6. public class Product
7. {
8.     private int id;
9.     private String name;
10.    private String description;
11.    private int quantity;
12.
13.    public Product() {
14.    }
15.
16.    public Product(int id, String name, String description, int quantity) {
17.        this.id = id;
18.        this.name = name;
19.        this.description = description;
20.        this.quantity = quantity;
21.    }
22.
23.    public int getId() {
24.        return id;
25.    }
26.
27.    public void setId(int id) {
28.        this.id = id;
29.    }
30.
31.    public String getName() {
32.        return name;
33.    }
34.
35.    public void setName(String name) {
36.        this.name = name;
37.    }
38.
39.    public String getDescription() {
40.        return description;
41.    }
42.
43.    public void setDescription(String description) {
44.        this.description = description;
45.    }
46.
47.    public int getQuantity() {
48.        return quantity;
49.    }
50.
51.    public void setQuantity(int quantity) {
52.        this.quantity = quantity;
53.    }
```

54. }

`Product` 类是一个简单的POJO, 包含4个字段。清单3显示了 `ProductFieldSetMapper` 类的源代码。

清单3 `ProductFieldSetMapper.java`

```

1. package com.geekcap.javaworld.springbatchexample.simple.reader;
2.
3. import com.geekcap.javaworld.springbatchexample.simple.model.Product;
4. import org.springframework.batch.item.file.mapping.FieldSetMapper;
5. import org.springframework.batch.item.file.transform.FieldSet;
6. import org.springframework.validation.BindException;
7.
8. /**
9.  * 根据 CSV 文件中的字段集合构建 Product 对象
10. */
11. public class ProductFieldSetMapper implements FieldSetMapper<Product>
12. {
13.     @Override
14.     public Product mapFieldSet(FieldSet fieldSet) throws BindException {
15.         Product product = new Product();
16.         product.setId( fieldSet.readInt( "id" ) );
17.         product.setName( fieldSet.readString( "name" ) );
18.         product.setDescription( fieldSet.readString( "description" ) );
19.         product.setQuantity( fieldSet.readInt( "quantity" ) );
20.         return product;
21.     }
22. }

```

`ProductFieldSetMapper` 类实现了 `FieldSetMapper` 接口, 该接口只定义了一个方法: `mapFieldSet()`。只要 `line mapper` 将一行数据解析为单独的字段, 就会构建一个 `FieldSet` (包含命名好的字段), 然后将这个 `FieldSet` 对象传递给 `mapFieldSet()` 方法。该方法负责创建对象来表示 CSV文件中的一行。在本例中, 我们通过 `FieldSet` 的各种 `read` 方法 构建一个 `Product` 实例。

写入数据库

在读取文件得到一组 `Product` 之后, 下一步就是将其写入到数据库。原则上我们可以组装一个 `processing step`, 用来对这些数据进行某些业务处理, 为简单起见, 我们直接将数据写到数据库中。清单4是 `ProductItemWriter` 类的源码。

清单4 `ProductItemWriter.java`

```

1. package com.geekcap.javaworld.springbatchexample.simple.writer;

```

```

2.
3. import com.geekcap.javaworld.springbatchexample.simple.model.Product;
4. import org.springframework.batch.item.ItemWriter;
5. import org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.jdbc.core.JdbcTemplate;
7. import org.springframework.jdbc.core.RowMapper;
8.
9. import java.sql.ResultSet;
10. import java.sql.SQLException;
11. import java.util.List;
12.
13. /**
14.  * Writes products to a database
15.  */
16. public class ProductItemWriter implements ItemWriter<Product>
17. {
18.     private static final String GET_PRODUCT = "select * from PRODUCT where id = ?";
19.     private static final String INSERT_PRODUCT = "insert into PRODUCT
20. (id,name,description,quantity) values (?, ?, ?, ?)";
21.     private static final String UPDATE_PRODUCT = "update PRODUCT set name = ?, description =
22. ?, quantity = ? where id = ?";
23.
24.     @Autowired
25.     private JdbcTemplate jdbcTemplate;
26.
27.     @Override
28.     public void write(List<? extends Product> products) throws Exception
29.     {
30.         for( Product product : products )
31.         {
32.             List<Product> productList = jdbcTemplate.query(GET_PRODUCT, new Object[]
33. {product.getId()}, new RowMapper<Product>() {
34.                 @Override
35.                 public Product mapRow( ResultSet resultSet, int rowNum ) throws SQLException {
36.                     Product p = new Product();
37.                     p.setId( resultSet.getInt( 1 ) );
38.                     p.setName( resultSet.getString( 2 ) );
39.                     p.setDescription( resultSet.getString( 3 ) );
40.                     p.setQuantity( resultSet.getInt( 4 ) );
41.                     return p;
42.                 }
43.             });
44.
45.             if( productList.size() > 0 )
46.             {
47.                 jdbcTemplate.update( UPDATE_PRODUCT, product.getName(),
48. product.getDescription(), product.getQuantity(), product.getId() );
49.             }
50.             else

```

```

47.         {
48.             jdbcTemplate.update( INSERT_PRODUCT, product.getId(), product.getName(),
product.getDescription(), product.getQuantity() );
49.         }
50.     }
51. }
52. }

```

`ProductItemWriter` 类实现了 `ItemWriter` 接口，该接口只有一个方法：`write()`。方法 `write()` 接受一个 `list`，这里是 `List<? extends Product> products`。Spring Batch 使用一种称为“chunking”的策略来实现 `writer`，`chunking` 的意思就是在读取时是一次读取一条数据，但写入时是将一组数据一起执行的。在 `job` 配置中，可以(通过 `commit-interval`)来控制每次想要一起写的 `item` 的数量。在上面的例子中，`write()` 方法做了这些事：

1. 执行一条 `SELECT` 语句来根据指定的 `id` 获取 `Product`。
2. 如果 `SELECT` 取得一条记录，则 `write()` 中更新数据库中对应记录的值。
3. 如果 `SELECT` 没有查询结果，则 `write()` 执行 `INSERT` 将产品信息添加到数据库中。

`ProductItemWriter` 类使用了 Spring 的 `JdbcTemplate` 类，这是在 `applicationContext.xml` 文件中定义的，通过自动装配机制注入到 `ProductItemWriter` 中。如果你没有用过 `JdbcTemplate`，可以把它理解为是对 `JDBC` 接口的一个封装。与数据库进行交互的 [模板设计模式](#) 的实现。代码应该很容易理解，如果想了解更多信息，请查看 [SpringJdbcTemplate](#) 的 [javadoc](#)。

用 application context 将上下文组装起来

到目前为止我们创建了 `Product` 领域对象类，一个映射器类 `ProductFieldSetMapper`，用来将 CSV 文件中的一行转换为一个对象，以及 `ProductItemWriter` 类，用来将对象写入数据库。下面我们需要配置 Spring Batch 来将这些组件组装起来。清单5 显示了 `applicationContext.xml` 文件的源代码，里面定义了我们需要的 bean。

清单 5. `applicationContext.xml`

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:context="http://www.springframework.org/schema/context"
5.     xmlns:batch="http://www.springframework.org/schema/batch"
6.     xmlns:jdbc="http://www.springframework.org/schema/jdbc"
7.     xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
8.         http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
9.         http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch.xsd
10.        http://www.springframework.org/schema/jdbc

```

```

    http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">
11.
12.
13.     <context:annotation-config />
14.
15.     <!-- Component scan to find all Spring components -->
16.     <context:component-scan base-package="com.geekcap.javaworld.springbatchexample" />
17.
18.
19.     <!-- Data source - connect to a MySQL instance running on the local machine -->
20.     <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
21.         <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
22.         <property name="url" value="jdbc:mysql://localhost/spring_batch_example"/>
23.         <property name="username" value="sbe"/>
24.         <property name="password" value="sbe"/>
25.     </bean>
26.
27.     <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
28.         <property name="dataSource" ref="dataSource" />
29.     </bean>
30.
31.     <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
32.         <property name="dataSource" ref="dataSource" />
33.     </bean>
34.
35.     <!-- Create job-meta tables automatically -->
36.     <jdbc:initialize-database data-source="dataSource">
37.         <jdbc:script location="org/springframework/batch/core/schema-drop-mysql.sql" />
38.         <jdbc:script location="org/springframework/batch/core/schema-mysql.sql" />
39.     </jdbc:initialize-database>
40.
41.
42.     <!-- Job Repository: used to persist the state of the batch job -->
43.     <bean id="jobRepository"
class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
44.         <property name="transactionManager" ref="transactionManager" />
45.     </bean>
46.
47.
48.     <!-- Job Launcher: creates the job and the job state before launching it -->
49.     <bean id="jobLauncher"
class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
50.         <property name="jobRepository" ref="jobRepository" />
51.     </bean>
52.
53.
54.     <!-- Reader bean for our simple CSV example -->
55.     <bean id="productReader" class="org.springframework.batch.item.file.FlatFileItemReader"

```

```

scope="step">
56.
57.     <!-- <property name="resource" value="file:./sample.csv" /> -->
58.     <property name="resource" value="file:${jobParameters['inputFile']}" />
59.
60.
61.     <!-- Skip the first line of the file because this is the header that defines the fields
-->
62.     <property name="linesToSkip" value="1" />
63.
64.     <!-- Defines how we map lines to objects -->
65.     <property name="lineMapper">
66.         <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
67.
68.             <!-- The lineTokenizer divides individual lines up into units of work -->
69.             <property name="lineTokenizer">
70.                 <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
71.
72.                     <!-- Names of the CSV columns -->
73.                     <property name="names" value="id,name,description,quantity" />
74.                 </bean>
75.             </property>
76.
77.             <!-- The fieldSetMapper maps a line in the file to a Product object -->
78.             <property name="fieldSetMapper">
79.                 <bean
class="com.geekcap.javaworld.springbatchexample.simple.reader.ProductFieldSetMapper" />
80.                 </property>
81.             </bean>
82.         </property>
83.     </bean>
84.
85.     <bean id="productWriter"
class="com.geekcap.javaworld.springbatchexample.simple.writer.ProductItemWriter" />
86.
87. </beans>

```

将 job 配置从 application/environment 中分离出来，使我们能够在环境迁移时不需要再重新定义一个 job。清单5中定义了下面这些bean：

- `dataSource` : 示例程序需要连接到MySQL，所以连接池也就配置为连接到一个名为 `spring_batch_example` 的MySQL数据库，地址为本机(localhost)，具体设置请参见下文。
- `transactionManager` : Spring事务管理器，用于管理MySQL事务。
- `jdbcTemplate` : 该类提供了与JDBC 连接交互的模板设计模式实现。这是一个用来简化数据库操作的辅助类，。在实际的项目中一般会使用某种ORM工具，例如Hibernate，上面再包装一

个服务层，但为了简单，那就先这样了。

- `jobRepository` : `MapJobRepositoryFactoryBean` 是 Spring Batch 用来管理 job 状态的组件。在这里它使用前面配置的 `jdbcTemplate` 将 job 信息储存到MySQL数据库中。
- `jobLauncher` : 该组件用来启动和管理 Spring Batch 作业的工作流，
- `productReader` : 在job中这个 bean 负责执行读操作。
- `productWriter` : 这个bean 负责将 `Product` 实例写入数据库。

请注意，`jdbc:initialize-database` 节点指向了两个用来创建所需数据库表的Spring Batch 脚本。这些脚本文件位于 Spring Batch core 的JAR文件中(由Maven自动引入了)对应的路径下。JAR文件中包含了对应多个数据库的脚本，比如MySQL、Oracle、SQL Server,等等。这些脚本负责在运行 job 时创建需要的schema。在本示例中，它删除(drop)表，然后再创建(create)表，你可以试着运行一下。如果在生产环境中，你应该将SQL文件提取出来，然后手动执行——毕竟生产环境一般创建了就不会删除。

Spring Batch 中的 Lazy scope

你可能注意到 `productReader` 的 `scope` 属性值为“`step`”。`step scope` 是Spring框架中的一种作用域，主要用于 Spring Batch。它本质上是一种 `lazy scope`，Spring在首次访问时才创建这种 `bean`。在本例中，我们需要使用 `step scope` 是因为使用了 `job` 参数中的“`InputFile`”值，这个值在应用启动时是不存在的。在 Spring Batch 中使用 `step scope` 使得在 `Bean`创建时能收到“`InputFile`”值。

定义job

清单6显示了 `file-import-job.xml` 文件，该文件定义了实际的 job

清单6 `file-import-job.xml`

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:context="http://www.springframework.org/schema/context"
5.     xmlns:batch="http://www.springframework.org/schema/batch"
6.     xsi:schemaLocation="http://www.springframework.org/schema/beans
7.         http://www.springframework.org/schema/beans/spring-beans.xsd
8.         http://www.springframework.org/schema/context
9.         http://www.springframework.org/schema/context/spring-context.xsd
10.         http://www.springframework.org/schema/batch
11.         http://www.springframework.org/schema/batch/spring-batch.xsd">
12.     <!-- Import our beans -->
13.     <import resource="classpath:/applicationContext.xml" />
14.     <job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">

```

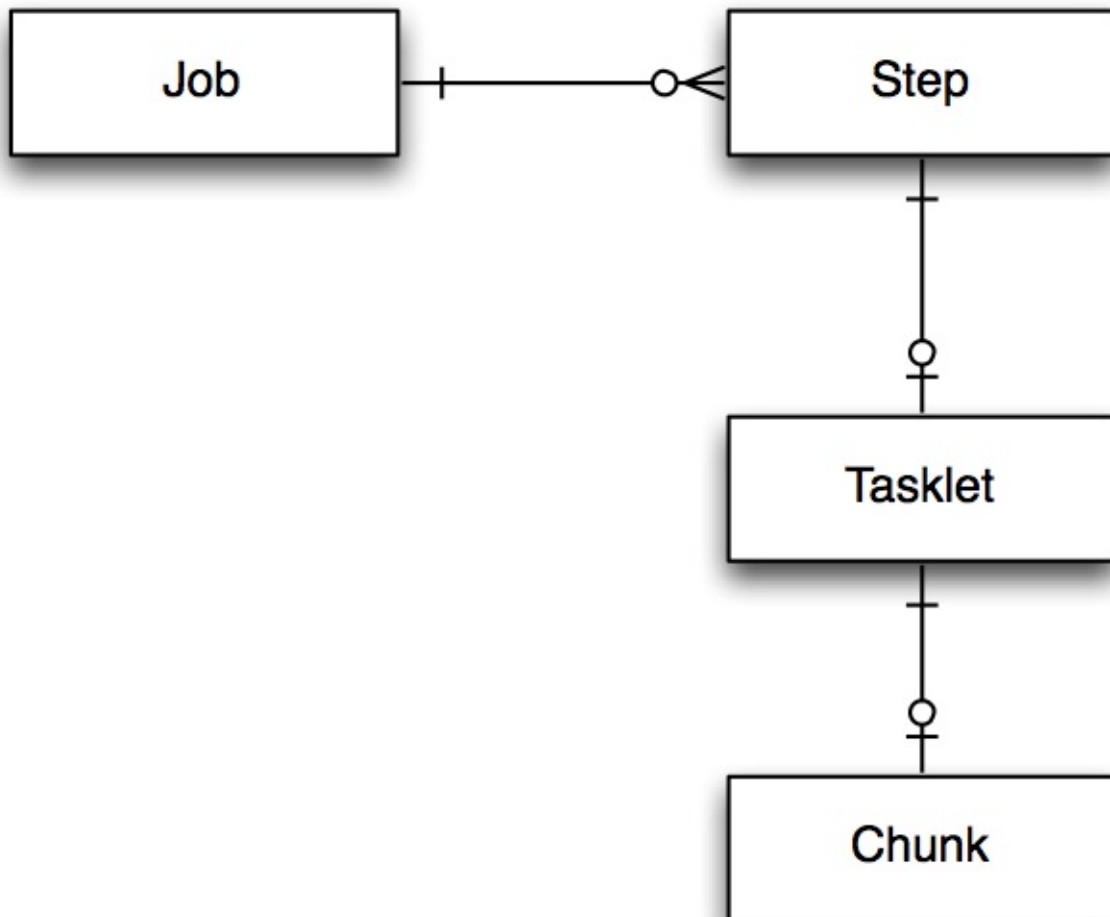
```

15.     <step id="importFileStep">
16.         <tasklet>
17.             <chunk reader="productReader" writer="productWriter" commit-interval="5" />
18.         </tasklet>
19.     </step>
20. </job>
21.
22. </beans>

```

请注意，一个 `job` 可以包含 0到多个 `step`；一个 `step` 可以有 0/1 个 `tasklet`；一个 `tasklet` 可以有 0/1 个 `chunk`，如图3所示。

图3 `job`、`step`、`tasklet` 和 `chunk` 关系



示例中，`simpleFileImportJob` 包含一个名为 `importFileStep` 的 `step`。`importFileStep` 包含一个未命名的 `tasklet`，而 `tasklet` 中有一个 `chunk`。`chunk` 引用了 `productReader` 和 `productWriter`。同时指定 `commit-interval` 值为 `5`。意思是每次最多传递给 `writer` 的记录数是5条。该 `step` 使用 `productReader` 读取5条产品记录，然后将这些记录传递给 `productWriter` 写出。`chunk` 一直重复执行，直到所有数据都处理完成。

清单6 还引入了 `applicationContext.xml` 文件,该文件包含所有需要的bean。而 Job 通常在单独的文件中定义;这是因为 job 加载器在执行时需要一个 job 文件以及对应的 job name。虽然可以把所有的东西揉进一个文件中,但很快会变得臃肿和难以维护,所以一般约定,一个 job 定义在一个文件中,同时引入其他依赖文件。

最后,你可能会注意到,job 节点 上定义了XML名称空间(`xmlns`)。这样做是为了不想在每个节点上加上前缀 “ `batch:` ”。在某个节点上定义的 namespace 会作用于该节点和所有子节点。

构建并运行项目

清单7显示了构建此项目的POM文件的内容

清单7 `pom.xml`

```

1. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance"
2.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
   4.0.0.xsd">
3.   <modelVersion>4.0.0</modelVersion>
4.
5.   <groupId>com.geekcap.javaworld</groupId>
6.   <artifactId>spring-batch-example</artifactId>
7.   <version>1.0-SNAPSHOT</version>
8.   <packaging>jar</packaging>
9.
10.  <name>spring-batch-example</name>
11.  <url>http://maven.apache.org</url>
12.
13.  <properties>
14.    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.    <spring.version>3.2.1.RELEASE</spring.version>
16.    <spring.batch.version>2.2.1.RELEASE</spring.batch.version>
17.    <java.version>1.6</java.version>
18.  </properties>
19.
20.  <dependencies>
21.    <!-- Spring Dependencies -->
22.    <dependency>
23.      <groupId>org.springframework</groupId>
24.      <artifactId>spring-context</artifactId>
25.      <version>${spring.version}</version>
26.    </dependency>
27.    <dependency>
28.      <groupId>org.springframework</groupId>
29.      <artifactId>spring-core</artifactId>
30.      <version>${spring.version}</version>
31.    </dependency>

```

```
32.     <dependency>
33.         <groupId>org.springframework</groupId>
34.         <artifactId>spring-beans</artifactId>
35.         <version>${spring.version}</version>
36.     </dependency>
37.     <dependency>
38.         <groupId>org.springframework</groupId>
39.         <artifactId>spring-jdbc</artifactId>
40.         <version>${spring.version}</version>
41.     </dependency>
42.     <dependency>
43.         <groupId>org.springframework</groupId>
44.         <artifactId>spring-tx</artifactId>
45.         <version>${spring.version}</version>
46.     </dependency>
47.     <dependency>
48.         <groupId>org.springframework.batch</groupId>
49.         <artifactId>spring-batch-core</artifactId>
50.         <version>${spring.batch.version}</version>
51.     </dependency>
52.     <dependency>
53.         <groupId>org.springframework.batch</groupId>
54.         <artifactId>spring-batch-infrastructure</artifactId>
55.         <version>${spring.batch.version}</version>
56.     </dependency>
57.
58.     <!-- Apache DBCP -->
59.     <dependency>
60.         <groupId>commons-dbcp</groupId>
61.         <artifactId>commons-dbcp</artifactId>
62.         <version>1.4</version>
63.     </dependency>
64.
65.     <!-- MySQL -->
66.     <dependency>
67.         <groupId>mysql</groupId>
68.         <artifactId>mysql-connector-java</artifactId>
69.         <version>5.1.27</version>
70.     </dependency>
71.
72.
73.     <!-- Testing -->
74.     <dependency>
75.         <groupId>junit</groupId>
76.         <artifactId>junit</artifactId>
77.         <version>4.11</version>
78.         <scope>test</scope>
79.     </dependency>
```

```

80. </dependencies>
81.
82. <build>
83.   <plugins>
84.     <plugin>
85.       <groupId>org.apache.maven.plugins</groupId>
86.       <artifactId>maven-compiler-plugin</artifactId>
87.       <configuration>
88.         <source>${java.version}</source>
89.         <target>${java.version}</target>
90.       </configuration>
91.     </plugin>
92.     <plugin>
93.       <groupId>org.apache.maven.plugins</groupId>
94.       <artifactId>maven-jar-plugin</artifactId>
95.       <configuration>
96.         <archive>
97.           <manifest>
98.             <addClasspath>>true</addClasspath>
99.             <classpathPrefix>lib/</classpathPrefix>
100.          </manifest>
101.        </archive>
102.      </configuration>
103.    </plugin>
104.    <plugin>
105.      <groupId>org.apache.maven.plugins</groupId>
106.      <artifactId>maven-dependency-plugin</artifactId>
107.      <executions>
108.        <execution>
109.          <id>copy</id>
110.          <phase>install</phase>
111.          <goals>
112.            <goal>copy-dependencies</goal>
113.          </goals>
114.          <configuration>
115.            <outputDirectory>${project.build.directory}/lib</outputDirectory>
116.          </configuration>
117.        </execution>
118.      </executions>
119.    </plugin>
120.  </plugins>
121.  <finalName>spring-batch-example</finalName>
122. </build>
123.
124.
125. </project>

```

上面的POM文件引入了 Spring context, core, beans,tx 和 JDBC 这些依赖库, 还引入了

Spring Batch core 以及 infrastructure 依赖(包)。这些依赖组合起来就是 Spring 和 Spring Batch 的基础。当然也引入了 Apache DBCP 数据库连接池和MySQL驱动。 `plug-in` 部分指定了使用Java 1.6进行编译,并在 build 时将所有依赖项复制到 lib 目录下。

可以使用下面的命令来构建项目:

```
1. mvn clean install
```

Spring Batch 连接数据库

现在 job 已经组装好了,如果想在生产环境中运行还需要将Spring Batch 连到数据库。Spring Batch 需要有一些表来记录 job 的当前状态和已经处理过的 record 表。这样,假若某个 job 确实需要重启,就可以从上次断开的地方继续执行。

Spring Batch 可以连接到多个数据库,但为了演示,所以使用MySQL。请 [下载MySQL](#) 并安装后再执行下面的脚本。社区版是免费的,而且能满足大多数需要。请根据你的操作系统选择合适的版本下载安装。然后可能需要手动启动MySQL(Windows 一般自动启动)。

安装好MySQL后还需要创建数据库以及相应的用户(并赋予权限)。启动命令行并进入MySQL的bin目录启动 mysql 客户端,连接服务器后执行以下SQL命令(请注意,在Linux下可能需要使用 `root` 用户执行 `mysql` 客户端程序,或者使用 `sudo` 进行权限切换。

```
1. create database spring_batch_example;
2. create user 'sbe'@'localhost' identified by 'sbe';
3. grant all on spring_batch_example.* to 'sbe'@'localhost';
```

第一行SQL创建名为 `spring_batch_example` 的数据库(database),这个库用来保存产品数据。第二行创建名为 `sbe` 的用户, Spring Batch Example的缩写,也可以使用其他名字,只要配置得一致就行),密码也指定为 `sbe`。最后一行将 `spring_batch_example` 数据库上的所有权限赋予 `sbe` 用户。

接下来,使用下面的命令创建 **PRODUCT** 表:

```
1. CREATE TABLE PRODUCT (
2.     ID INT NOT NULL,
3.     NAME VARCHAR(128) NOT NULL,
4.     DESCRIPTION VARCHAR(128),
5.     QUANTITY INT,
6.     PRIMARY KEY(ID)
7. );
```

接着,我们在项目的 target 目录下创建一个文件 `sample.csv`,并填充一些数据(用英文逗号分

隔):

```
1. id,name,description,quantity
2. 1,Product One,This is product 1, 10
3. 2,Product Two,This is product 2, 20
4. 3,Product Three,This is product 3, 30
5. 4,Product Four,This is product 4, 20
6. 5,Product Five,This is product 5, 10
7. 6,Product Six,This is product 6, 50
8. 7,Product Seven,This is product 7, 80
9. 8,Product Eight,This is product 8, 90
```

在Linux中可以使用下面的命令启动 batch job, 其中的CLASSPATH 分隔符是英文冒号:

```
1. java -cp spring-batch-example.jar:./lib/*
    org.springframework.batch.core.launch.support.CommandLineJobRunner classpath:/jobs/file-import-
    job.xml simpleFileImportJob inputFile=sample.csv
```

当然, 如果操作系统是windows, 因为使用英文分号作为 CLASSPATH 分隔符, 所以需要有一点改变:

```
1. java -cp spring-batch-example.jar;./lib/*
    org.springframework.batch.core.launch.support.CommandLineJobRunner classpath:/jobs/file-import-
    job.xml simpleFileImportJob inputFile=sample.csv
```

`CommandLineJobRunner` 是 Spring Batch 框架中执行 job 的类。它需要定义了 job的XML文件名, 以及需要执行的job名字, 还可以传入其他自定义参数。因为 `file-import-job.xml` 被打包在JAR文件中, 所以可以通过: `classpath:/jobs/file-import-job.xml` 来访问。我们需要执行的 Job 名称为: `simpleFileImportJob`, 还传入一个参数 `InputFile`, 值为 `sample.csv`。

如果执行不出错, 输出结果类似于下面这样:

```
1. Nov 12, 2013 4:09:17 PM org.springframework.context.support.AbstractApplicationContext
    prepareRefresh
2. INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@6b4da8f4:
    startup date [Tue Nov 12 16:09:17 EST 2013]; root of context hierarchy
3. Nov 12, 2013 4:09:17 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
    loadBeanDefinitions
4. INFO: Loading XML bean definitions from class path resource [jobs/file-import-job.xml]
5. Nov 12, 2013 4:09:18 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
    loadBeanDefinitions
6. INFO: Loading XML bean definitions from class path resource [applicationContext.xml]
7. Nov 12, 2013 4:09:19 PM org.springframework.beans.factory.support.DefaultListableBeanFactory
    registerBeanDefinition
8. INFO: Overriding bean definition for bean 'simpleFileImportJob': replacing [Generic bean: class
    [org.springframework.batch.core.configuration.xml.SimpleFlowFactoryBean]; scope=;
    abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true;
    primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null;
    destroyMethodName=null] with [Generic bean: class
```

```

[org.springframework.batch.core.configuration.xml.JobParserJobFactoryBean]; scope=;
abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=true;
primary=false; factoryBeanName=null; factoryMethodName=null; initMethodName=null;
destroyMethodName=null]
9. Nov 12, 2013 4:09:19 PM org.springframework.beans.factory.support.DefaultListableBeanFactory
registerBeanDefinition
10. INFO: Overriding bean definition for bean 'productReader': replacing [Generic bean: class
[org.springframework.batch.item.file.FlatFileItemReader]; scope=step; abstract=false;
lazyInit=false; autowireMode=0; dependencyCheck=0; autowireCandidate=false; primary=false;
factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null;
defined in class path resource [applicationContext.xml]] with [Root bean: class
[org.springframework.aop.scope.ScopedProxyFactoryBean]; scope=; abstract=false; lazyInit=false;
autowireMode=0; dependencyCheck=0; autowireCandidate=true; primary=false; factoryBeanName=null;
factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in BeanDefinition
defined in class path resource [applicationContext.xml]]
11. Nov 12, 2013 4:09:19 PM org.springframework.beans.factory.support.DefaultListableBeanFactory
preInstantiateSingletons
12. INFO: Pre-instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@6aba4211: defining beans
[org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework
root of factory hierarchy
13. Nov 12, 2013 4:09:19 PM org.springframework.batch.core.launch.support.SimpleJobLauncher
afterPropertiesSet
14. INFO: No TaskExecutor has been set, defaulting to synchronous executor.
15. Nov 12, 2013 4:09:22 PM org.springframework.batch.core.launch.support.SimpleJobLauncher$1 run
16. INFO: Job: [FlowJob: [name=simpleFileImportJob]] launched with the following parameters:
[{:inputFile=sample.csv}]
17. Nov 12, 2013 4:09:22 PM org.springframework.batch.core.job.SimpleStepHandler handleStep
18. INFO: Executing step: [importFileStep]
19. Nov 12, 2013 4:09:22 PM org.springframework.batch.core.launch.support.SimpleJobLauncher$1 run
20. INFO: Job: [FlowJob: [name=simpleFileImportJob]] completed with the following parameters:
[{:inputFile=sample.csv}] and the following status: [COMPLETED]
21. Nov 12, 2013 4:09:22 PM org.springframework.context.support.AbstractApplicationContext doClose
22. INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@6b4da8f4:
startup date [Tue Nov 12 16:09:17 EST 2013]; root of context hierarchy
23. Nov 12, 2013 4:09:22 PM org.springframework.beans.factory.support.DefaultSingletonBeanRegistry
destroySingletons
24. INFO: Destroying singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@6aba4211: defining beans
[org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework
root of factory hierarchy

```

然后到数据库中校对一下 **PRODUCT** 表中是否正确保存了我们在 csv 文件中指定的那几条记录(示例是8条)。

使用 Spring Batch 执行批量处理

到这一步，示例程序从CSV文件中读取数据，并将信息保存到了数据库中。虽然可以运行起来，但有时候想要对数据进行转换或者过滤某些数据，然后再插入到数据库中。在本节中，我们将创建一个简单的 processor，并不覆盖原有的 product 数量，而是先从数据库中查询现有记录，然后将CSV

文件中对应的数量增加到 product 中, 然后再写入数据库。

清单8显示了 **ProductItemProcessor** 类的源码。

清单8 `ProductItemProcessor.java`

```

1. package com.geekcap.javaworld.springbatchexample.simple.processor;
2.
3. import com.geekcap.javaworld.springbatchexample.simple.model.Product;
4. import org.springframework.batch.item.ItemProcessor;
5. import org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.jdbc.core.JdbcTemplate;
7. import org.springframework.jdbc.core.RowMapper;
8.
9. import java.sql.ResultSet;
10. import java.sql.SQLException;
11. import java.util.List;
12.
13. /**
14.  * Processor that finds existing products and updates a product quantity appropriately
15.  */
16. public class ProductItemProcessor implements ItemProcessor<Product,Product>
17. {
18.     private static final String GET_PRODUCT = "select * from PRODUCT where id = ?";
19.     @Autowired
20.     private JdbcTemplate jdbcTemplate;
21.
22.     @Override
23.     public Product process(Product product) throws Exception
24.     {
25.         // Retrieve the product from the database
26.         List<Product> productList = jdbcTemplate.query(GET_PRODUCT, new Object[]
{product.getId()}, new RowMapper<Product>() {
27.             @Override
28.             public Product mapRow( ResultSet resultSet, int rowNum ) throws SQLException {
29.                 Product p = new Product();
30.                 p.setId( resultSet.getInt( 1 ) );
31.                 p.setName( resultSet.getString( 2 ) );
32.                 p.setDescription( resultSet.getString( 3 ) );
33.                 p.setQuantity( resultSet.getInt( 4 ) );
34.                 return p;
35.             }
36.         });
37.
38.         if( productList.size() > 0 )
39.         {
40.             // Add the new quantity to the existing quantity
41.             Product existingProduct = productList.get( 0 );

```

```

42.         product.setQuantity( existingProduct.getQuantity() + product.getQuantity() );
43.     }
44.
45.     // Return the (possibly) update product
46.     return product;
47. }
48. }

```

`ProductItemProcessor` 实现了 `ItemProcessor<I,O>` 接口，其中类型 **I** 表示传递给处理器的对象类型，而 **O** 则表示处理器返回的对象类型。在本例中，我们传入的是 `Product` 对象，返回的也是 `Product` 对象。`ItemProcessor` 接口只定义了一个方法：`process()`，在里面我们根据给定的 `id` 执行一条 **SELECT** 语句从数据库中查询对应的 `Product`。如果找到 `Product` 对象，则将该对象的数量加上新的数量。

`processor` 没有做任何过滤，如果 `process()` 方法返回 `null`，则Spring Batch 将会忽略这个 `item`，不将其发送给 `writer`。

将 `processor` 组装到 `job` 中很简单。首先，添加一个新的bean 到 `applicationContext.xml` 中：

```

1. <bean id="productProcessor"
    class="com.geekcap.javaworld.springbatchexample.simple.processor.ProductItemProcessor" />

```

接下来，在 `chunk` 中通过 `processor` 属性来引用这个 bean：

```

1. <job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
2.     <step id="importFileStep">
3.         <tasklet>
4.             <chunk reader="productReader" processor="productProcessor" writer="productWriter"
    commit-interval="5" />
5.         </tasklet>
6.     </step>
7. </job>

```

编译并执行 `job`，如果不出错，每执行一次都可以在数据库中看到产品数量发生了变化。

创建多个processors

前面我们定义了单个处理器，但有时候可能想要以适当的粒度来创建多个 `item processor`，然后在同一个 `chunk` 中按顺序执行。例如，可能需要一个过滤器来忽略数据库中不存在的记录，还需要一个 `processor` 来正确地管理 `item` 数量。这时可以使用Spring Batch 的

`CompositeItemProcessor` 来大显身手。使用步骤如下：

1. 创建 processor 类

2. 在applicationContext.xml中配置各个 bean
3. 定义一个类型为 `org.springframework.batch.item.support.CompositeItemProcessor` 的 bean, 然后将其 `delegates` 设置为需要执行的处理器bean 的 list
4. 让 `chunk` 的 `processor` 属性指向 `CompositeItemProcessor`

假设有一个 `ProductFilterProcessor` , 则可以像下面这样指定 process :

```

1. <bean id="productFilterProcessor"
   class="com.geekcap.javaworld.springbatchexample.simple.processor.ProductFilterItemProcessor" />
2.
3. <bean id="productProcessor"
   class="com.geekcap.javaworld.springbatchexample.simple.processor.ProductItemProcessor" />
4.
5. <bean id="productCompositeProcessor"
   class="org.springframework.batch.item.support.CompositeItemProcessor">
6.   <property name="delegates">
7.     <list>
8.       <ref bean="productFilterProcessor" />
9.       <ref bean="productProcessor" />
10.    </list>
11.  </property>
12. </bean>

```

然后只需修改一下 job 配置即可, 如下所示:

```

1. <job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
2.   <step id="importFileStep">
3.     <tasklet>
4.       <chunk reader="productReader" processor="productCompositeProcessor"
5.         writer="productWriter" commit-interval="5" />
6.     </tasklet>
7.   </step>
8. </job>

```

Tasklets (微任务)

分块是一个非常好的作业拆分策略, 用来将作业拆分成多块: 依次读取每一个 item , 执行处理, 然后将其按块写出。但如果想执行某些只需要执行一次的线性操作该怎么办呢? 此时可以创建一个 `tasklet` 。 `tasklet` 可以执行各种操作/需求! 例如, 可以从FTP站点下载文件, 解压/解密文件, 或者调用web服务来判断文件处理是否已经执行。下面是创建 `tasklet` 的基本过程:

1. 定义一个实现 `org.springframework.batch.core.step.tasklet.Tasklet` 接口的类。
2. 实现 `execute()` 方法。
3. 返回恰当的 `org.springframework.batch.repeat.RepeatStatus` 值: `CONTINUABLE` 或者是 `FINISHED` .

4. 在 `applicationContext.xml` 文件中定义对应的 bean。
5. 创建一个 `step`，其中子元素 `tasklet` 引用第4步定义的bean。

清单9 显示了一个新的 `tasklet` 的源码，将我们的输入文件拷贝到存档目录中。

清单9 `ArchiveProductImportFileTasklet.java`

```

1. package com.geekcap.javaworld.springbatchexample.simple.tasklet;
2.
3. import org.apache.commons.io.FileUtils;
4. import org.springframework.batch.core.StepContribution;
5. import org.springframework.batch.core.scope.context.ChunkContext;
6. import org.springframework.batch.core.step.tasklet.Tasklet;
7. import org.springframework.batch.repeat.RepeatStatus;
8.
9. import java.io.File;
10.
11. /**
12.  * A tasklet that archives the input file
13.  */
14. public class ArchiveProductImportFileTasklet implements Tasklet
15. {
16.     private String inputFile;
17.
18.     @Override
19.     public RepeatStatus execute(StepContribution stepContribution, ChunkContext chunkContext)
20.     throws Exception
21.     {
22.         // Make our destination directory and copy our input file to it
23.         File archiveDir = new File( "archive" );
24.         FileUtils.forceMkdir( archiveDir );
25.         FileUtils.copyFileToDirectory( new File( inputFile ), archiveDir );
26.
27.         // We're done...
28.         return RepeatStatus.FINISHED;
29.     }
30.
31.     public String getInputFile() {
32.         return inputFile;
33.     }
34.
35.     public void setInputFile(String inputFile) {
36.         this.inputFile = inputFile;
37.     }

```

`ArchiveProductImportFileTasklet` 类实现了 `Tasklet` 接口的 `execute()` 方法。其中

使用Apache Commons I/O 库的工具类 `FileUtils` 来创建一个新的 `archive` 目录,然后将 `input file` 拷贝到里面。

将下面的 bean 定义添加到 `applicationContext.xml` 文件中:

```

1. <bean id="archiveFileTasklet"
   class="com.geekcap.javaworld.springbatchexample.simple.tasklet.ArchiveProductImportFileTasklet"
   scope="step">
2.     <property name="inputFile" value="#{jobParameters['inputFile']}" />
3. </bean>

```

请注意, 因为传入了一个名为 `inputFile` 的 job 参数,所以 这个bean 也设置了作用域范围 `scope="step"`, 以确保在 bean 对象创建时能取得需要的 job 参数。

清单10 显示了更新后的job.

清单10 `file-import-job.xml`

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:context="http://www.springframework.org/schema/context"
5.     xmlns:batch="http://www.springframework.org/schema/batch"
6.     xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context.xsd
   http://www.springframework.org/schema/batch
   http://www.springframework.org/schema/batch/spring-batch.xsd">
7.
8.
9.
10.
11.     <!-- Import our beans -->
12.     <import resource="classpath:/applicationContext.xml" />
13.
14.     <job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
15.         <step id="importFileStep" next="archiveFileStep">
16.             <tasklet>
17.                 <chunk reader="productReader" processor="productProcessor"
   writer="productWriter" commit-interval="5" />
18.             </tasklet>
19.         </step>
20.         <step id="archiveFileStep">
21.             <tasklet ref="archiveFileTasklet" />
22.         </step>
23.     </job>
24.
25. </beans>

```

清单10中添加了一个id为 `archiveFileStep` 的step, , 然后在 `importFileStep` 中将 `"next"` 指向他。 `"next"` 参数可以用来控制 job 中 step 的执行流程。虽然超出了本文所述的范围,但需要注意,可以根据某个任务的执行结果状态来决定下面执行哪个 step[也就是 job 分支,类似于 if,switch 什么的]。 `archiveFileStep` 只包含上面创建的这个 `tasklet`。

弹性(Resiliency)

Spring Batch job resiliency提供了以下三个工具:

1. **Skip** : 如果处理过程中某条记录是错误的,如CSV文件中格式不正确的行,那么可以直接跳过该对象,继续处理下一个。
2. **Retry** : 如果出现错误,而在几毫秒后再次执行就可能解决,那么可以让 Spring Batch 对该元素重试一次/(或多次)。例如,你可能想要更新数据库中的某条记录,但另一个查询把这条记录给锁了。根据业务设计,这个锁将会很快释放,那么重试可能就会成功。
3. **Restart** : 如果将 job 状态存储在数据库中,而一旦它执行失败,那么就可以选择重启 job 实例,并继续上次执行的位置。

我们这里不会详细讲述每个 Resiliency 特征,但我想总结一下可用的选项。

Skipping Items(跳过某项)

有时你可能想要跳过某些记录,比如 reader 读取的无效记录,或者处理/写入过程中出现异常的对象。要跳过记录有两种方式:

- 在 `chunk` 元素上定义 `skip-limit` 属性,告诉Spring 最多允许跳过多少个 items,超过则 job 失败(如果无效记录很少那可以接受,但如果无效记录太多,那可能输入数据就有问题了)。
- 定义一个 `skippable-exception-classes` 列表,用来判断当前记录是否可以跳过,可以指定 `include` 元素来决定发生哪些异常时会跳过当前记录,还可以指定 `exclude` 元素来决定哪些异常不会触发 skip(比如你想跳过某个异常层次父类,但排除一或多个子类异常时)。

示例如下:

```

1. <job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
2.     <step id="importFileStep">
3.         <tasklet>
4.             <chunk reader="productReader" processor="productProcessor" writer="productWriter"
5.                 commit-interval="5" skip-limit="10">
6.                 <skippable-exception-classes>
7.                     <include class="org.springframework.batch.item.file.FlatFileParseException" />
8.                 </skippable-exception-classes>
9.             </chunk>
10.        </tasklet>
11.    </step>
12. </job>

```

这里在处理某条记录时如果抛出 `FlatFileParseException` 异常，则这条记录将被跳过。如果超过 10次 skip，那么直接让 job 失败。

重试 (Retrying Items)

有时发生的异常是可以重试的，如由于数据库锁导致的失败。重试(Retry)的实现和跳过(Skip)非常相似：

- 在 `chunk` 元素上定义 `retry-limit` 属性，告诉Spring 每个 item 最多允许重试多少次，超过则认为该记录处理失败。如果只用重试，不指定跳过，则如果某条记录重试处理失败，则 job将被标记为失败。
- 定义一个 `retryable-exception-classes` 列表，用来判断当前记录是否可以重试；可以指定 `include` 元素来决定哪些异常发生时当前记录可以重试，还可以指定 `exclude` 元素来决定哪些异常不对当前记录重试执行。。

例如：

```

1. <job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
2.   <step id="importFileStep">
3.     <tasklet>
4.       <chunk reader="productReader" processor="productProcessor" writer="productWriter"
5.         commit-interval="5" retry-limit="5">
6.         <retryable-exception-classes>
7.           <include class="org.springframework.dao.OptimisticLockingFailureException" />
8.         </retryable-exception-classes>
9.       </chunk>
10.    </tasklet>
11.  </step>
12. </job>

```

还可以将重试和可跳过的异常通过 `skippable exception` 与 `retry exception` 对应起来。因此，如果某个异常触发了5次重试，5次重试之后还没搞定，恰好该异常也在 `skippable` 列表中，则这条记录将被跳过。如果 `exception` 不在 `skippable` 列表则会让整个 job 失败。

重启 job

最后，对于执行失败的 job作业，我们可以重新启动，并让他们从上次断开的地方继续执行。要做到这一点，只需要使用和上次一模一样的参数来启动 job，则 Spring Batch 会自动从数据库中找到这个实例然后继续执行。你也可以拒绝重启，或者通过参数控制 job 中一个 step 可以重启的次数(一般来说多次重试都失败了，则可能需要放弃处理了。)

总结

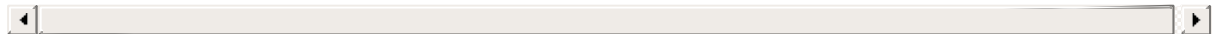
批处理是某些业务需求最好的解决方案，Spring batch 框架提供了实现批处理作业的整体架构。Spring Batch 将分块模式分为三个阶段：读取(read)、处理(process)、和写入(write)，并支持常规资源的读取和写入。本期的[Open source Java projects](#) 系列探讨了 Spring Batch 是干什么的以及如何使用它。

我们先创建了一个简单的 job，从CSV文件读取 Product 数据然后导入数据库，接着添加 processor 来对 job 进行扩展：用来管理 product 数量。最后我们写了一个单独的 tasklet 来归档输入文件。虽然示例中没有涉及，但Spring Batch 的弹性特征是非常重要的，所以简单介绍了Spring Batch提供的三大弹性工具：skipping records, retrying records, 和 restarting batch jobs。

本文只是简单介绍 Spring Batch 的皮毛，但希望能让你对 Spring Batch 执行批处理作业有一定的了解和认识。

下载本教程的源代码：

[SpringBatch-CSV演示代码](#)



Spring Batch 3.0新特性

- [Spring Batch 3.0新特性](#)

Spring Batch 3.0新特性

Spring Batch 3.0 release 主要有5个主题

- JSR-252 的支持
- 支持升级至 Spring4 和 java8
- 增强 Spring Batch 之间的整合
- 支持 JobScope
- 支持 SQLite

JSR-352支持

- [JSR-352支持](#)

JSR-352支持

JSR-352是java批处理的新规范。受Spring Batch的深度影响，该规范提供了Spring Batch已经存在的相关功能。Spring Batch 3.0已实现该规范，支持遵循标准定义批处理任务了。使用JSR-352 的任务规范语言（JSL）配置一个批处理任务，代码如下：

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <job id="myJob3" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
3.     <step id="step1" >
4.         <batchlet ref="testBatchlet" />
5.     </step>
6. </job>
```

详细请参见 [JSR-352 Support](#) 章节

改进的Spring Batch Integration模块

- [改进的Spring Batch Integration模块](#)

改进的Spring Batch Integration模块

Spring Batch Integration 曾经是 Spring Batch Admin 项目的子模块。Spring Batch 的Spring Integration提供了更好整合能力的功能。这些特殊功能包括：

- 通过消息启动任务
- 异步ItemProcessors
- 提供反馈信息的消息
- 通过远程分区和远程块执行外部化批处理

详细请参见 [Spring Batch Integration](#) 章节

升级到支持Spring 4和Java 8

- [升级到支持Spring 4和Java 8](#)

升级到支持Spring 4和Java 8

随着 Spring Batch Integration 成为 Spring Batch 项目的一个模块，它将更新为使用 Spring Integration 4。Spring Integration 4 给Spring引进了核心消息api。由此，Spring Batch 3 需要 Spring 4或更高版本的支持。

作为依赖此次版本更新的一部分，Spring Batch 可运行在java8上。但它仍可在java6或更高版本 (java6-java8)上运行。

JobScope支持

- [JobScope支持](#)

JobScope支持

在很长一段时间内，Spring Batch 的scope配置项“step”在批处理应用中起到关键作用，它提供了后期绑定功能。在Spring release 3.0版本，Spring Batch支持一个“job”的配置项。这个配置允许对象延迟创建，一般直到每个job将要执行时提供新的实例。你可在 [Section 5.4.2, “Job Scope”](#) 章节查看关于该配置的详细内容。

SQLite支持

- [SQLite支持](#)

SQLite支持

SQLite已作为新的数据库所支持，可为JobRepository添加job repository的DDL。这将提供了一个有用的、基于文件、数据存储的测试意图。

批处理专业术语

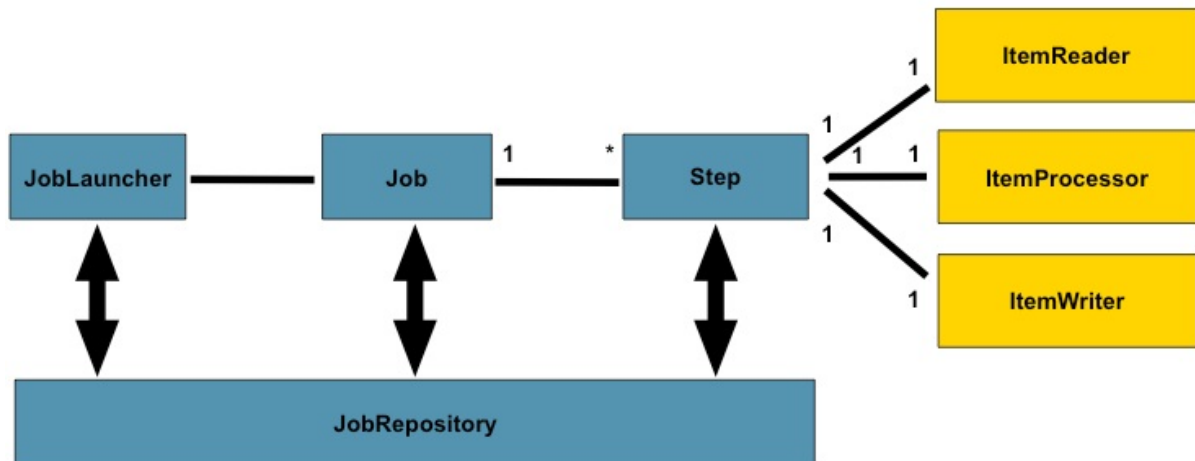
- 3. 批处理的域语言
 - 3.1 Job
 - 3.1.1 JobInstance
 - 3.1.2 JobParameters
 - 3.1.3 JobExecution
 - 3.2 Step
 - 3.2.1 StepExecution
 - 3.3 ExecutionContext
 - 3.4 JobRepository
 - 3.5 JobLauncher
 - 3.6 Item Reader
 - 3.7 Item Writer
 - 3.8 Item Processor
 - 3.9 Batch Namespace

3. 批处理的域语言

对于有任何批处理操作经验的架构师来说，在Spring Batch中所使用的批处理的整体概念都会感到熟悉与舒适。其中有“Jobs”，“Steps”以及开发者所提供的被称为“ItemReader”和“ItemWriter”的批处理单元。另外，基于Spring的模式、操作、模板、回调和术语，还有着以下的方便性：

- 在分离关注点方面的显著增强
- 轮廓清晰的架构层次与作为接口提供服务
- 简单与默认的实现能够快速的上手以及能够容易使用框架以外的技术
- 显著增强的可扩展性

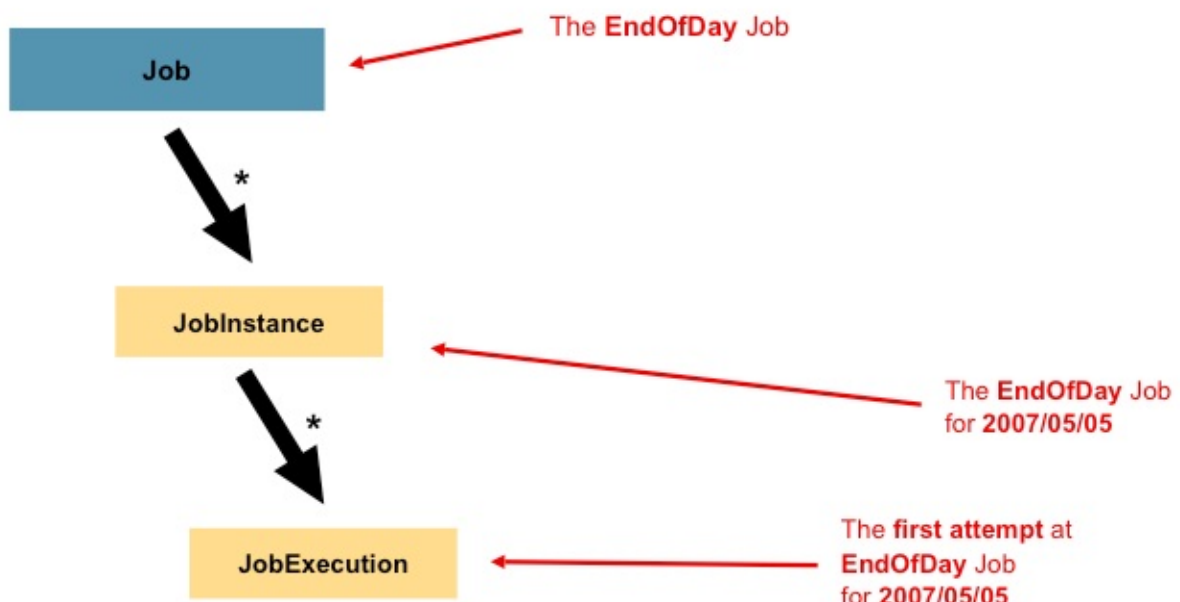
下面的关系图是一个已使用了数十年的批处理体系结构的简化版本，它提供了组成批处理操作域语言各组件的概述。这个框架体系作为一个蓝图在最近几代平台上提供了数十种实现（COBOL/Mainframe, C++/Unix, 以及现在的Java/其他平台）。JCL和COBOL开发者同C++、C#和Java开发者一样熟悉这个概念。Spring Batch为此提供的层次、组件与技术服务的物理实现被验证是健壮的，作为基础服务与扩展服务通常被用于建立从简单到复杂的批处理应用，用来解决十分复杂的处理需求。



在上面的关系图中，所包含的高亮部分就是组成批处理域语言的关键概念。一个Job有多个Step，一个Step精确对应一个ItemReader、ItemProcessor、ItemWriter。一个Job需要被启动 (JobLauncher)，当前运行中的流程需要被存储(JobRepository)。

3.1 Job

本节介绍了有关批处理任务的固有概念。一个 **Job** 作为一个实体，封装了整个批处理过程，同 Spring 其他的项目一样，**Job** 经过xml配置文件或基于java的配置关联起来。此配置可能会被称为“job configuration”。但是，**Job** 也只是整个层次结构的顶层：



在Spring Batch中，**Job** 只是**Step**的容器，组合了一个流程中属于一个逻辑下的多个step，也能够进行针对所有step的全局属性配置，例如可重启配置。job的配置包括：

- job的简称
- Step的定义及执行顺序
- job是否是可重启的

Spring Batch提供了**SimpleJob**这样一个默认的简单job接口的实现形式，在job的顶层建立了标准功能，然而有时批处理的名空间概念需要直接实例化，这时就可以使用 `<job>` 标签：

```

1. <job id="footballJob">
2.     <step id="playerload" next="gameLoad"/>
3.     <step id="gameLoad" next="playerSummarization"/>
4.     <step id="playerSummarization"/>
5. </job>

```

3.1.1 JobInstance

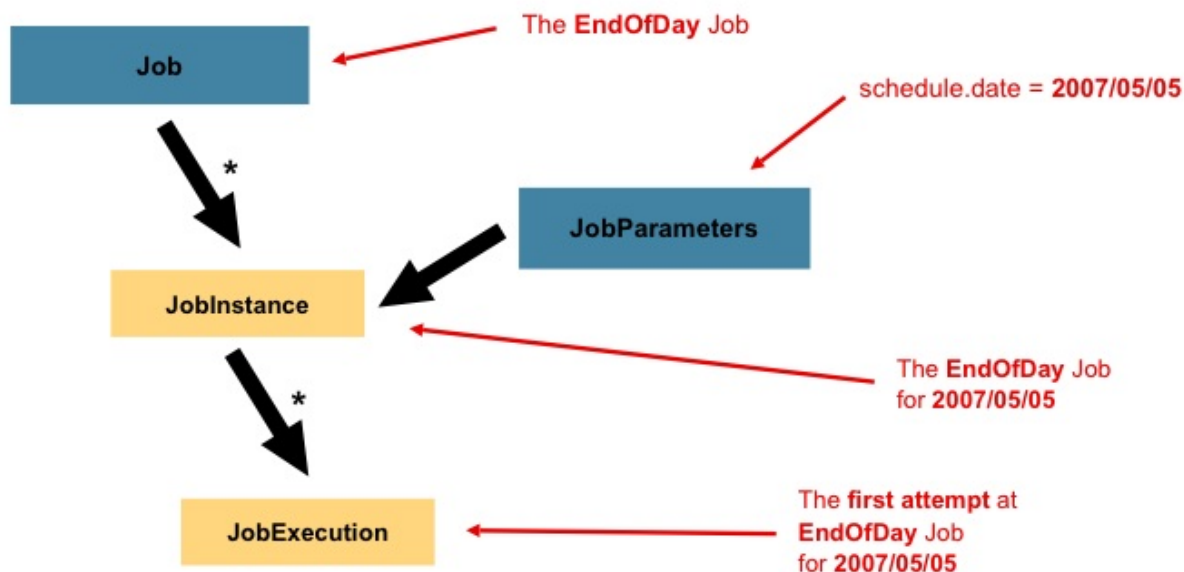
JobInstance涉及到一个运行逻辑job的概念。让我们考虑一下，在一天结束时运行一次批处理任务，

类似上述图表中的“EndOfDay”任务。有一个统一的“EndOfDay”任务，但是job的每个独立运行都必须分开跟踪监控。在这个例子中，每天会有一个逻辑的JobInstance。例如，在1月1日运行一次，在1月2日运行一次。如果1月1日第一次运行失败了，第二天再运行一次，这仍然是1月1日的运行（通常也和处理的数据对应起来，意味着处理的是1月1日的数据等等）。因此，每个JobInstance能有多次执行（JobExecution的更多详细信息将在下文讨论），而对应于在特定时间运行特定的Job和JobParameter只有一个JobInstance。

JobInstance的定义完全与加载的数据无关，数据的加载方式只与ItemReader的实现方式相关。例如在EndOfDay场景中，可能有一个名为“effective data”或是“schedule data”的数据列用来表示数据所属日期。因此，1月1日运行只会加载属于1日的数据，1月2日运行只会加载2日的数据。因为这更像是一个业务决定，所以留给ItemReader去处理。另外，使用相同的JobInstance可以决定是否使用前一次执行使用的状态（例如ExecutionContext，下文讨论）。使用新的JobInstance意味着“从头开始”，而使用存在的instance通常意味着“从离开的地方开始”。

3.1.2 JobParameters

讨论完JobInstance与Job之间的区别，自然要开始讨论“如何区分一个JobInstance与另一个JobInstance”。答案是：JobParameters。JobParameters是一组用来启动批处理任务的参数，他们被用于在运行过程中标记识别或是引用数据：



在上面的例子中一个job有两个实例，一个是1月1日以01-01-2008的参数启动运行，一个是1月2日以01-02-2008的参数启动运行。因此可以这样定义：JobInstance = Job + JobParameter。这让开发者有效地控制JobInstance，因为开发者可以有效控制输入给JobInstance的参数。

注意：并非所有的job需要设定参数来对JobInstance进行标识，默认情况下应该设定参数，在该框架中也允许带有参数提交的job不做对JobInstance进行标识的处理工作。

3.1.3 JobExecution

一个JobExecution的概念是对于运行一次Job。一次执行可能成功也可能失败，但是只有这次执行完全成功后对应的JobInstance才会被认为是完成了。以之前的EndOfDay任务为例，01-01-2008第一次运行生成的JobInstance失败后，以相同的参数(01-01-2008)再次运行，一个新的JobExecution会被创建，但是仍然是同一个JobInstance。

Job定义了任务是什么以及如何启动，JobInstance是纯粹的组织对象用来执行操作组织在一起，主要目的是开启正确的重启语义，而JobExecution是运行过程中状态的主要存储机制，有着多得多的属性需要控制与持久化：

表3.1 JobExecution属性

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
startTime	使用java.util.Date类表示任务开始时的系统时间
endTime	使用java.util.Date类表示任务结束时的系统时间
exitStatus	ExitStatus表示任务的运行结果。它是最重要的，因为它包含了返回给调用者的退出代码。更多详细信息参见第五章
createTime	使用java.util.Date类表示JobExecution第一次持久化时的系统时间。这是框架管理任务的ExecutionContext所要求的，一个任务可能还没有启动(也就没有startTime)，但总是会有createTime
lastUpdated	使用java.util.Date类表示最近一次JobExecution被持久化的系统时间

executionContext	'属性包'包含了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。在任务失败时有不止一个例外发生的情况下会很有用。

这些属性是很重要，将被持久化并用于判断任务执行的状态。例如，如果EndOfDay的任务在01-01的下午9点启动，在9:30时失败了，那么在批处理元数据表中会创建下面的记录：

表 3.2. BATCH_JOB_INSTANCE

JOB_INST_ID	JOB_NAME
1	EndOfDayJob

表 3.3. BATCH_JOB_EXECUTION_PARAMS

JOB_EXECUTION_ID	TYPE_CD	KEY_NAME	DATE_VAL	IDENTIFYING
1	DATE	schedule.Date	2008-01-01	TRUE

表 3.4. BATCH_JOB_EXECUTION

JOB_EXEC_ID	JOB_INST_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00	2008-01-01 21:30	FAILED

注意：针对清晰度与格式设置，列的名称可能已经缩写或简化。

现在任务失败了，花费了一整夜解决问题，但是批处理开放时间已经过去了。假定批处理开放时间是下午9:00开始，那么01-01的任务会重新开始，到9:30结束。但是由于现在已经是第二天了，01-02的任务也必须在之后的9:31开始运行，正常运行一个小时在10:30结束。除非两个job可能访问相同的数据，在数据库层面造成冲突锁，一般并不会要求一个JobInstance在另一个之后运行。一个Job什么时候能够运行完全是调度程序决定的，因此对于不同的JobInstance，Spring Batch并不会组织并发执行(如果一个JobInstance在运行时尝试同时运行相同的JobInstance则会抛出JobExecutionAlreadyRunningException例外)。此时JobInstance表和JobParameters表会增加一行数据，JobExecution表会增加两行数据：

表 3.5. BATCH_JOB_INSTANCE

JOB_INST_ID	JOB_NAME
1	EndOfDayJob
2	EndOfDayJob

表3.6. BATCH_JOB_EXECUTION_PARAMS

JOB_EXECUTION_ID	TYPE_CD	KEY_NAME	DATE_VAL	IDENTIFYING

1	DATE	schedule.Date	2008-01-01 00:00:00	TRUE
2	DATE	schedule.Date	2008-01-01 00:00:00	TRUE
3	DATE	schedule.Date	2008-01-02 00:00:00	TRUE

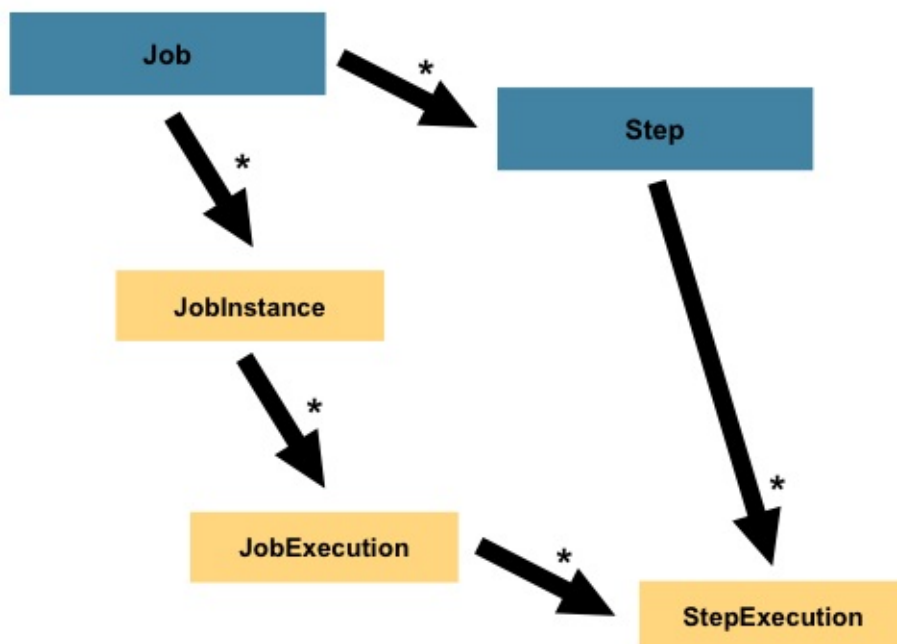
表3.7. BATCH_JOB_EXECUTION

JOB_EXEC_ID	JOB_INST_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00	2008-01-01 21:30	FAILED
2	1	2008-01-02 21:00	2008-01-02 21:30	COMPLETED
3	2	2008-01-02 21:31	2008-01-02 22:29	COMPLETED

注意: 针对清晰度与格式设置, 列的名称可能已经缩写或简化。

3.2 Step

Step是一个域对象, 它封装了批处理任务中的一个独立的连续阶段。因此每个job是由一个或是多个step组成的, step包含了定义以及控制一个实际运行中批处理任务所必须的所有信息。这个描述很含糊, 是因为step中的内容完全是编写job的开发者所赋予的, step的复杂度完全依赖于开发者。一个简单的step可能只是从文件中加载数据到数据库, 只需要几行代码(依赖于实现方式)。一个复杂的step可能作为整个业务处理的一部分而有着复杂的业务流程。像job那样, step有着自己的StepExecution并且对应于唯一一个JobExecution:



3.2.1 StepExecution

StepExecution表示需要执行一个step，和JobExecution类似，在每次运行step时会创建一个新的StepExecution。但是，如果一个Step之前的那个step执行失败导致这个step无法执行，则不会为这个step创建对应的StepExecution，因为StepExecution只会在step实际启动时创建。

Step的执行过程是由StepExecution类的对象所表示的，包括了每次执行所对应的step、JobExecution、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时还包含了一个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。下表列出了StepExecution的属性：

status	使用BatchStatus对象来表示执行状态。运行时，状态为BatchStatus.STARTED；运行失败状态为BatchStatus.FAILED；成功结束时状态为BatchStatus.COMPLETED
--------	---

startTime	执行开始时间, 使用java.util.Date类表示
endTime	执行结束时间(成功或失败), 使用java.util.Date类表示
exitStatus	执行结果, 使用ExitStatus表示。最重要的是它包含了返回给调用者的退出代码。更多详细信息参见第五章
executionContext	包含了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	因为读取失败而略过的记录数
processSkipCount	因为处理失败而略过的记录数
filterCount	被ItemProcessor过滤的记录数
writerSkipCount	因为写入失败而略过的记录数

3.3 ExecutionContext

ExecutionContext是一组框架持久化与控制的键/值对, 能够让开发者在StepExecution或JobExecution范畴保存需要进行持久化的状态, 它同Quartz的JobDataMap是相似的。任务重启就是最好的例子。以一个平面文件输入为例, 在处理单行时, 框架会在每次commit(提交)之后持久化ExecutionContext, 这样ItemReader万一在运行过程中遇到问题, 甚至是掉电, 也能够存储之前的状态。而完成这些只需要把当前读取的行数放入context中, 框架就会完成剩下的:

```
1. executionContext.putLong(getKey(LINES_READ_COUNT), reader.getPosition());
```


使用之前Job概念模块中的EndOfDay示例，假设有这么一个step：'loadData'（加载数据），将文件加载到数据库中。在第一次运行失败后，元数据表应该是这样：

表 3.9. BATCH_JOB_INSTANCE

JOB_INST_ID	JOB_NAME
1	EndOfDayJob

表3.10. BATCH_JOB_PARAMS

JOB_INST_ID	TYPE_CD	KEY_NAME	DATE_VAL
1	DATE	schedule.Date	2008-01-01

表3.11. BATCH_JOB_EXECUTION

JOB_EXEC_ID	JOB_INST_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00	2008-01-01 21:30	FAILED

表3.12. BATCH_STEP_EXECUTION

STEP_EXEC_ID	JOB_EXEC_ID	STEP_NAME	START_TIME	END_TIME	STATUS
1	1	loadDate	2008-01-01 21:00	2008-01-01 21:30	FAILED

表3.13. BATCH_STEP_EXECUTION_CONTEXT

STEP_EXEC_ID	SHORT_CONTEXT
1	{piece.count=40321}

在这个例子中，step运行30分钟处理了40321条记录，在该场景中将代表在文件中产生的行数。框架在每次提交前会更新这个数字，并且ExecutionContext可以包含多个相关条目。如果要在提交前进行通知，还需要一个StepListener变量或是一个ItemStream变量，这个在随后的章节中会有更详细的讨论。就之前的这个例子，假定job在第二天重启，在重启时会从数据库中读取前一次运行保存下来的数据重组成ExecutionContext，打开ItemReader的时候，ItemReader会检查context中保存的状态并使用这些状态进行初始化：

```

1. if (executionContext.containsKey(getKey(LINES_READ_COUNT))) {
2.     log.debug("Initializing for restart. Restart data is: " + executionContext);
3.
4.     long lineCount = executionContext.getLong(getKey(LINES_READ_COUNT));
5.
6.     LineReader reader = getReader();
7.

```

```

8.     Object record = "";
9.     while (reader.getPosition() < lineCount && record != null) {
10.         record = readLine();
11.     }
12. }

```

在上面的代码执行完成后，记录下当前行是40322，这样下次启动时就能从这一行开始继续执行。ExecutionContext也能用来统计需要保存的运行内容，例如，一个文件需要处理订单，而一条订单可能保存为多行记录，此时需要记录处理了多少条订单(订单数与文件行数不同)并在任务处理完成后把处理的订单总数用邮件发送出去。框架会为开发者把这些数据记录到当前运行的这个JobInstance的scope中。另外，判断ExecutionContext的使用时机是比较困难的。例如使用上面的'EndOfDay'例子，当01-01任务在01-02再次运行时，框架会认识到这是一个相同的JobInstance中一个不同的Step，于是从数据库中取出ExecutionContext作为Step的一部分StepExecution内容。相反对于01-02任务框架要认识到这是一个不同的instance，于是会给step一个空的context。框架需要为工程师做各种判断以保证在准确的时间有准确的状态。还需要重视的是在任何时间对于每个StepExecution只有一个ExecutionContext，因为创建的是一个共有的keyspace，ExecutionContext的客户端需要小心放置key-value数据以确保没有数据被覆盖。当然如果step没有在Context中保存数据，自然也不会被框架造成不利的影晌。

同样需要重点注意的是：一个JobExecution至少有一个ExecutionContext，所有StepExecution共用一个ExecutionContext。如下代码所示：

```

1. ExecutionContext ecStep = stepExecution.getExecutionContext();
2. ExecutionContext ecJob = jobExecution.getExecutionContext();
3. //ecStep does not equal ecJob

```

正如在注释中的标注，ecStep不等于ecJob，他们是两个不同的ExecutionContext。step范围的ExecutionContext在Step被提交时保存，job范围的ExecutionContext在两个step执行之间保存。

3.4 JobRepository

JobRepository是上面所有概念的持久化机制，为JobLauncher, Job和Step提供了CRUD实现。当一个Job第一次启动时，从仓库中获取一个JobExecution，之后在整个执行过程中StepExecution和JobExecution都被这样持久化到仓库中：

```

1. <job-repository id="jobRepository"/>

```

3.5 JobLauncher

JobLauncher是根据设定的参数JobParameters来启动Job的简单接口：

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

在此期望从JobRepository获取一个合法的JobExecution并执行Job。

3.6 Item Reader

ItemReader是一个抽象概念，用于表示step读取数据，一次读取一条。当ItemReader读完了所有数据，它会返回一个null值。关于 ItemReader接口的更多详细信息及其多种实现方式可以参见 [Chapter 6, ItemReaders and ItemWriters](#)。

3.7 Item Writer

ItemWriter是一个抽象概念，用于表示step输出数据，一次输出一批或大块数据。通常情况下，ItemWriter只有在数据传递到ItemWriter时，才知道输入的数据内容。关于ItemWriter 接口的更多详细信息及其多种实现方式可以参见 [Chapter 6, ItemReaders and ItemWriters](#)。

3.8 Item Processor

ItemProcessor是一个抽象概念，用于表示item的业务处理。ItemReader读数据，ItemWriter写数据，ItemProcessor能够转换数据或是处理业务逻辑。如果在处理过程中数据不合法，则会返回null值表示数据没有输出。关于ItemProcessor 接口的更多详细信可以参见 [Chapter 6, ItemReaders and ItemWriters](#)。

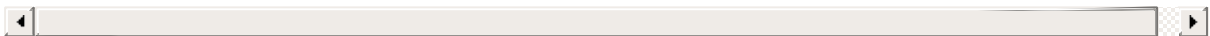
3.9 Batch Namespace

上面的许多域概念都需要在spring的ApplicationContext中配置。但是如果这些接口的实现使用标准的bean定义方式，那么namespace提供了更容易的配置方式：

```
1. <beans:beans xmlns="http://www.springframework.org/schema/batch"
2.     xmlns:beans="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="
```

```
5.         http://www.springframework.org/schema/beans
6.         http://www.springframework.org/schema/beans/spring-beans.xsd
7.         http://www.springframework.org/schema/batch
8.         http://www.springframework.org/schema/batch/spring-batch-2.2.xsd">
9.
10.        <job id="ioSampleJob">
11.            <step id="step1">
12.                <tasklet>
13.                    <chunk reader="itemReader" writer="itemWriter" commit-interval="2"/>
14.                </tasklet>
15.            </step>
16.        </job>
17.
18. </beans:beans>
```

只要在批处理的namespace申明过，那么这些元素就能够使用。关于配置Job的更多信息可以参见 [Chapter 4, Configuring and Running a Job](#)。关于配置Step 的更多信息可以参见 [Chapter 5, Configuring a Step](#)。

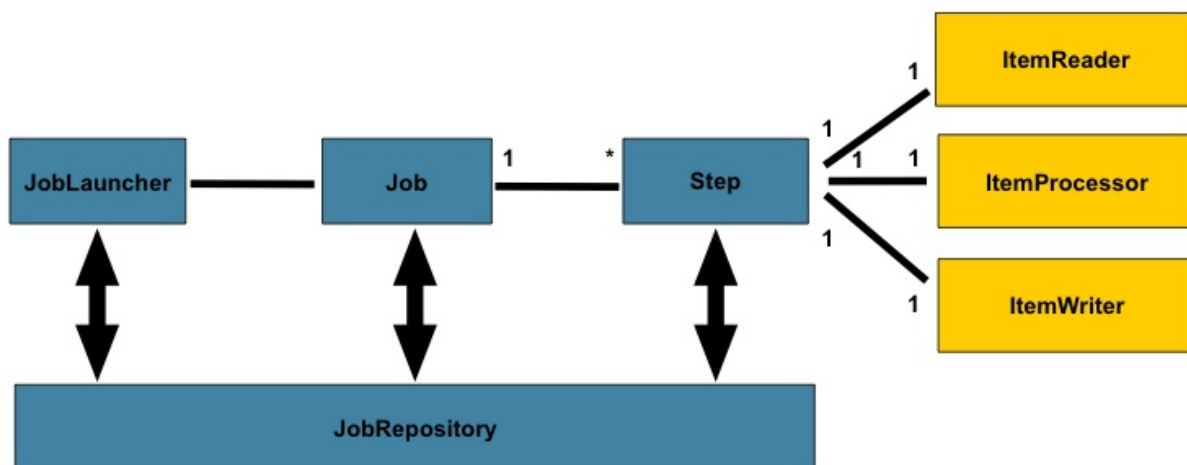


配置并运行Job

- 4. 配置并运行Job

4. 配置并运行Job

在上一章节([domain section](#)), 即批处理的域语言中, 讨论了整体的架构设计, 并使用如下关系图来进行表示:



虽然Job对象看上去像是对于多个Step的一个简单容器, 但是开发者必须要注意许多配置项。此外, Job的运行以及Job运行过程中元数据如何被保存也是需要考虑的。本章将会介绍Job在运行时所需要注意的各种配置项。

Configuring a Job

- [4.1 Configuring a Job](#)
 - [4.1.1 Restartability](#)
 - [4.1.2 Intercepting Job Execution](#)
 - [4.1.3 Inheriting from a parent Job](#)
 - [4.1.4 JobParametersValidator](#)

4.1 Configuring a Job

Job接口的实现有多个，但是在配置上命名空间存在着不同。必须依赖的只有三项：名称 **name**，**JobRepository** 和 **Step** 的列表：

```

1. <job id="footballJob">
2.     <step id="playerload"          parent="s1" next="gameLoad"/>
3.     <step id="gameLoad"          parent="s2" next="playerSummarization"/>
4.     <step id="playerSummarization" parent="s3"/>
5. </job>

```

在这个例子中使用了父类的bean定义来创建step，更多描述step配置的信息可以参考[step configuration](#)这一节。XML命名空间默认会使用id为'jobRepository'的引用来作为repository的定义。然而可以向如下显式的覆盖：

```

1. <job id="footballJob" job-repository="specialRepository">
2.     <step id="playerload"          parent="s1" next="gameLoad"/>
3.     <step id="gameLoad"          parent="s3" next="playerSummarization"/>
4.     <step id="playerSummarization" parent="s3"/>
5. </job>

```

此外，job配置的step还包含其他的元素，有并发处理(`parallel`)，显示的流程控制(`flowControl`)和外化的流程定义(`flowDefinition`)。

4.1.1 Restartability

执行批处理任务的一个关键问题是要考虑job被重启后的行为。如果一个 **JobExecution** 已经存在一个特定的 **JobInstance**，那么这个job启动时可以认为是“重启”。理想情况下，所有任务能够在他们中止的地方启动，但是有许多场景这是不可能的。在这种场景中就要有开发者来决定创建一个新的 **JobInstance**，Spring对此也提供了一些帮助。如果job不需要重启，而是总是作为新的 **JobInstance** 来运行，那么可重启属性可以设置为'false'：

```

1. <job id="footballJob" restartable="false">
2.     ...
3. </job>

```

设置重启属性restartable为'false'表示'这个job不支持再次启动'，重启一个不可重启的job会抛出JobRestartException的异常：

```

1. Job job = new SimpleJob();
2. job.setRestartable(false);
3.
4. JobParameters jobParameters = new JobParameters();
5.
6. JobExecution firstExecution = jobRepository.createJobExecution(job, jobParameters);
7. jobRepository.saveOrUpdate(firstExecution);
8.
9. try {
10.     jobRepository.createJobExecution(job, jobParameters);
11.     fail();
12. }
13. catch (JobRestartException e) {
14.     //预计抛出JobRestartException异常
15. }

```

这个JUnit代码展示了创建一个不可重启的Job后，第一次能够创建 **JobExecution** ，第二次再创建相同的JobExecution会抛出一个 **JobRestartException**。

4.1.2 Intercepting Job Execution

在job执行过程中，自定义代码能够在生命周期中通过事件通知执行会是很有用的。SimpleJob能够在适当的时机调用JobListener：

```

1. public interface JobExecutionListener {
2.
3.     void beforeJob(JobExecution jobExecution);
4.
5.     void afterJob(JobExecution jobExecution);
6.
7. }

```

JobListener能够添加到SimpleJob中去，作为job的listener元素：

```

1. <job id="footballJob">
2.     <step id="playerload"           parent="s1" next="gameLoad"/>

```

```

3.     <step id="gameLoad"          parent="s2" next="playerSummarization"/>
4.     <step id="playerSummarization" parent="s3"/>
5.     <listeners>
6.         <listener ref="sampleListener"/>
7.     </listeners>
8. </job>

```

无论job执行成功或是失败都会调用afterJob，都可以从 **JobExecution** 中获取运行结果后，根据结果来进行不同的处理：

```

1. public void afterJob(JobExecution jobExecution){
2.     if( jobExecution.getStatus() == BatchStatus.COMPLETED ){
3.         //job执行成功    }
4.     else if(jobExecution.getStatus() == BatchStatus.FAILED){
5.         //job执行失败    }
6. }

```

对应于这个interface的annotation为：

- @BeforeJob
- @AfterJob

4.1.3 Inheriting from a parent Job

如果一组job配置共有相似，但又不是完全相同，那么可以定义一个“父”job，让这些job去继承属性。同Java的类继承一样，子job会把父job的属性和元素合并进来。

下面的例子中，“baseJob”是一个抽象的job定义，只定义了一个监听器列表。名为“job1”的job是一个具体定义，它继承了“baseJob”的监听器，并且与自己的监听器合并，最终生成的job带有两个监听器，以及一个名为“step1”的step。

```

1. <job id="baseJob" abstract="true">
2.     <listeners>
3.         <listener ref="listenerOne"/>
4.     </listeners>
5. </job>
6.
7. <job id="job1" parent="baseJob">
8.     <step id="step1" parent="standaloneStep"/>
9.
10.    <listeners merge="true">
11.        <listener ref="listenerTwo"/>
12.    </listeners>
13. </job>

```


更多信息可参见 [Inheriting from a Parent Step](#)

4.1.4 JobParametersValidator

一个在xml命名空间描述的job或是使用任何抽象job子类的job，可以选择为运行时为job参数定义一个验证器。在job启动时需要保证所有必填参数都存在的场景下，这个功能是很有用的。有一个DefaultJobParametersValidator可以用来限制一些简单的必选和可选参数组合，你也可以实现接口用来处理更复杂的限制。验证器的配置支持使用xml命名空间来作为job的子元素，例如：

```
1. <job id="job1" parent="baseJob3">
2.     <step id="step1" parent="standaloneStep"/>
3.     <validator ref="paremetersValidator"/>
4. </job>
```

验证器可以作为一个引用(如上)来定义也可以直接内嵌定义在bean的命名空间中。

Java Config

- [4.2 Java Config](#)

4.2 Java Config

在Spring 3版本中可以采用java程序来配置应用程序，来替代XML配置的方式。正如在Spring Batch 2.2.0版本中，批处理任务中可以使用相同的java配置项来对其进行配置。关于Java的基础配置的两个组成部分分别是：`@EnableBatchConfiguration` 注释和两个builder。

在Spring的体系中 `@EnableBatchProcessing` 注释的工作原理与其它的带有 `@Enable *` 的注释类似。在这种情况下，`@EnableBatchProcessing` 提供了构建批处理任务的基本配置。在这个基本的配置中，除了创建了一个 **StepScope** 的实例，还可以将一系列可用的bean进行自动装配：

- **JobRepository** bean 名称 “jobRepository”
- **JobLauncher** bean名称“jobLauncher”
- **JobRegistry** bean名称“jobRegistry”
- **PlatformTransactionManager** bean名称 “transactionManager”
- **JobBuilderFactory** bean名称“jobBuilders”
- **StepBuilderFactory** bean名称“stepBuilders”

这种配置的核心接口是 **BatchConfigurer**。它为以上所述的bean提供了默认的实现方式，并要求在context中提供一个bean，即 **DataSource** 。数据库连接池由被 **JobRepository** 使用。

注意

只有一个配置类需要有@ enablebatchprocessing注释。只要有一个类添加了这个注释，则以上所有的bean都是可以使用的。

在基本配置中，用户可以使用所提供的builder factory来配置一个job。下面的例子是通过 **JobBuilderFactory** 和 **StepBuilderFactory** 配置的两个step job 。

```

1. @Configuration
2. @EnableBatchProcessing
3. @Import(DataSourceCnfiguration.class)
4. public class AppConfig {
5.
6.     @Autowired
7.     private JobBuilderFactory jobs;
8.
9.     @Autowired
10.    private StepBuilderFactory steps;
11.
12.    @Bean

```

```
13.     public Job job() {
14.         return jobs.get("myJob").start(step1()).next(step2()).build();
15.     }
16.
17.     @Bean
18.     protected Step step1(ItemReader<Person> reader, ItemProcessor<Person, Person> processor,
19. ItemWriter<Person> writer) {
20.         return steps.get("step1")
21.             .<Person, Person> chunk(10)
22.             .reader(reader)
23.             .processor(processor)
24.             .writer(writer)
25.             .build();
26.
27.     @Bean
28.     protected Step step2(Tasklet tasklet) {
29.         return steps.get("step2")
30.             .tasklet(tasklet)
31.             .build();
32.     }
33. }
```

Configuring a JobRepository

- 4.3 Configuring a JobRepository
 - 4.3.1 JobRepository 的事物配置
 - 4.3.2 修改 Table 前缀
 - 4.3.3 In-Memory Repository
 - 4.3.4 Non-standard Database Types in a Repository

4.3 Configuring a JobRepository

之前说过，**JobRepository** 是基本的CRUD操作，用于持久化Spring Batch的领域对象(如 JobExecution, StepExecution)。许多主要的框架组件(如 JobLauncher, Job, Step)都需要使用 JobRepository。batch的命名空间中已经抽象走许多 JobRepository 的实现细节，但是仍然需要一些配置：

```

1. <job-repository id="jobRepository"
2.     data-source="dataSource"
3.     transaction-manager="transactionManager"
4.     isolation-level-for-create="SERIALIZABLE"
5.     table-prefix="BATCH_"
6.     max-varchar-length="1000"/>

```

上面列出的配置除了id外都是可选的。如果没有进行参数配置，默认值就是上面展示的内容，之所以写出来是用于展示给读者。`max-varchar-length` 的默认值是2500，这表示varchar列的长度，在 [sample schema scripts](#) 中用于存储类似于 `exit code` 这些描述的字符。如果你不修改schema并且也不会使用多字节编码，那么就不用修改它。

4.3.1 JobRepository 的事物配置

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的持久化，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的定义。`create*` 方法的隔离级别会被单独指定，为了确保任务启动时，如果两个操作尝试在同时启动相同的任务，那么只有一个任务能够被成功启动。这种方法默认的隔离级别是 `SERIALIZABLE`，这是相当激进的做法：`READ_COMMITED` 能达到同样效果；如果两个操作不以这种方式冲突的话 `READ_UNCOMMITTED` 也能很好工作。但是，由于调用 `create*` 方法是相当短暂的，只要数据库支持，就不会对性能产生太大影响。它也能被这样覆盖：

```

1. <job-repository id="jobRepository"
2.     isolation-level-for-create="REPEATABLE_READ" />

```

如果factory的namespace没有被使用，那么可以使用AOP来配置repository的事务行为：

```

1. <aop:config>
2.     <aop:advisor
3.         pointcut="execution(* org.springframework.batch.core.*Repository+.*(..))"/>
4.     <advice-ref="txAdvice" />
5. </aop:config>
6.
7. <tx:advice id="txAdvice" transaction-manager="transactionManager">
8.     <tx:attributes>
9.         <tx:method name="*" />
10.    </tx:attributes>
11. </tx:advice>

```

这个配置片段基本上可以不做修改直接使用。记住加上适当的namespace描述去确保spring-tx和spring-aop(或是整个spring)都在classpath中。

4.3.2 修改 Table 前缀

JobRepository 可以修改的另一个属性是元数据表的表前缀。默认是以BATCH_开头，`BATCH_JOB_EXECUTION` 和 `BATCH_STEP_EXECUTION` 就是两个例子。但是，有一些潜在的原因可能需要修改这个前缀。例如schema的名字需要被预置到表名中，或是不止一组的元数据表需要放在同一个schema中，那么表前缀就需要改变：

```

1. <job-repository id="jobRepository"
2.     table-prefix="SYSTEM.TEST_" />

```

按照上面的修改配置，每一个元数据查询都会带上 `SYSTEM.TEST_` 的前缀，`BATCH_JOB_EXECUTION` 将会被更换为 `SYSTEM.TEST_JOB_EXECUTION`。

注意：表名前缀是可配置的，表名和列名是不可配置的。

4.3.3 In-Memory Repository

有的时候不想把你的领域对象持久化到数据库中，可能是为了运行的更快速，因为每次提交都要开销额外的时间；也可能并不需要为特定任务保存状态。那么Spring Batch还提供了内存Map版本的job仓库：

```

1. <bean id="jobRepository"
2.     class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
3.     <property name="transactionManager" ref="transactionManager"/>
4. </bean>

```

需要注意的是 内存 **Repository** 是轻量的并且不能在两个JVM实例间重启任务，也不能允许同时启动带有相同参数的任务，不适合在多线程的任务或是一个本地分片任务的场景下使用。而使用数据库版本的Repository则能够拥有这些特性。

但是也需要定义一个事务管理器，因为仓库需要回滚语义，也因为商业逻辑要求事务性（例如RDBMS访问）。经过测试许多人觉得 **ResourcelessTransactionManager** 是很有用的。

4.3.4 Non-standard Database Types in a Repository

如果使用的数据库平台不在支持的平台列表中，在SQL类型类似的情况下你可以使用近似的数据库类型。使用原生的 **JobRepositoryFactoryBean** 来取代命名空间缩写后设置一个相似的数据库类型：

```
1. <bean id="jobRepository" class="org...JobRepositoryFactoryBean">
2.     <property name="databaseType" value="db2"/>
3.     <property name="dataSource" ref="dataSource"/>
4. </bean>
```

(如果没有指定 `databaseType`，**JobRepositoryFactoryBean** 会通过DataSource自动检测数据库的类型)。平台之间的主要不同之处在于主键的计算策略，也可能需要覆盖 `incrementerFactory` (使用Spring Framework提供的标准实现)。

如果它还不能工作，或是你不使用RDBMS，那么唯一的选择是让 **SimpleJobRepository** 使用Spring方式依赖并且绑定在手工实现的各种Dao接口上。

Configuring a JobLauncher

- [4.4 Configuring a JobLauncher](#)

4.4 Configuring a JobLauncher

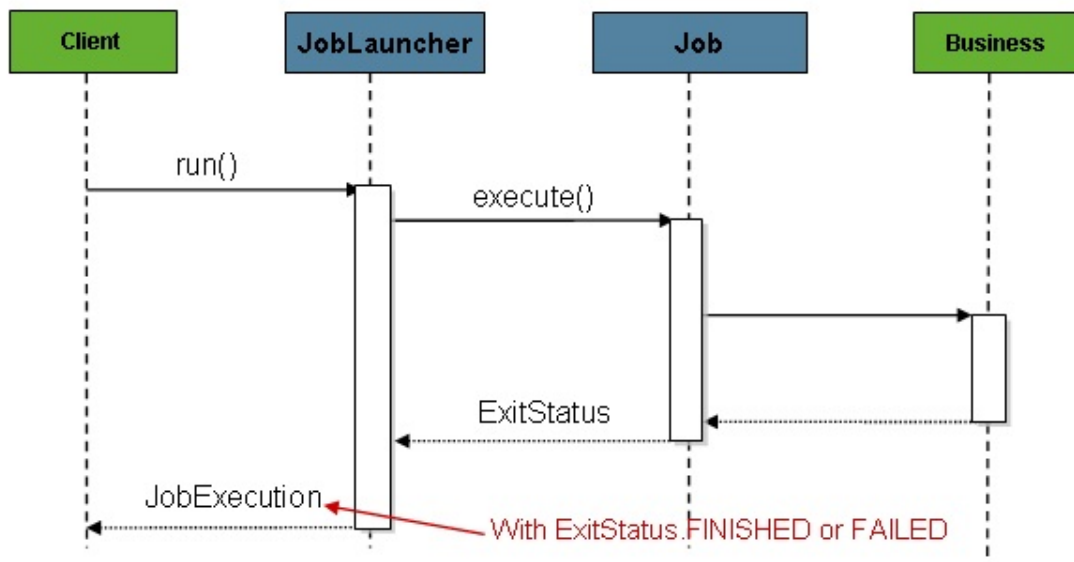
`JobLauncher` 最基本的实现是 `SimpleJobLauncher`，它唯一的依赖是通过 `JobRepository` 获取一个 `execution`：

```

1. <bean id="jobLauncher"
2.     class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
3.     <property name="jobRepository" ref="jobRepository" />
4. </bean>

```

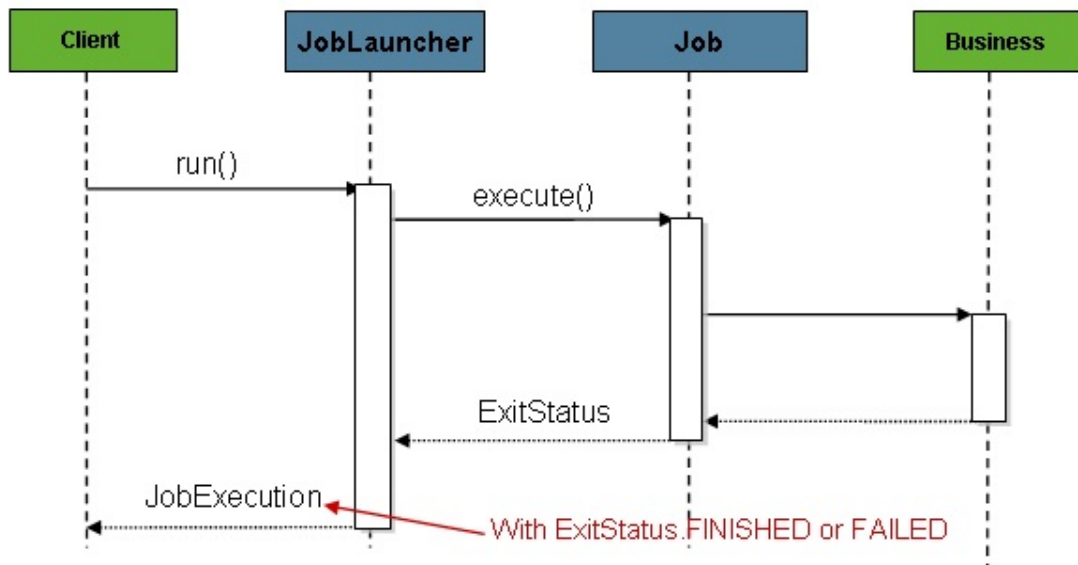
一旦获取到 `JobExecution`，那么可以通过执行 `Job` 的方法，最终将 `JobExecution` 返回给调用者：



28

从调度启动时，整个序列能够很好的直接工作，但是，从HTTP请求中启动则会出现一些问题。在这种

场景中，启动任务需要异步操作，让**SimpleJobLauncher**能够立刻返回结果给调用者，如果让HTTP请求一直等待很长时间知道批处理任务完成获取到执行结果，是很糟糕的操作体验。一个流程如下图所示：



28

通过配置 `TaskExecutor` 可以很容易的将 `SimpleJobLauncher` 配置成异步操作：

```

1. <bean id="jobLauncher"
2.     class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
3.     <property name="jobRepository" ref="jobRepository" />
4.     <property name="taskExecutor">
5.         <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
6.     </property>
7. </bean>
  
```

`TaskExecutor` 接口的任何实现都能够用来控制 `job` 的异步执行。

Running a Job

- 4.5 Running a Job
 - 4.5.1 从命令行启动 Jobs
 - 4.5.2 在 Web Container 内部运行 Jobs

4.5 Running a Job

运行一个批处理任务至少有两点要求：一个 `JobLauncher` 和一个用来运行的 `job`。它们都包含了相同或是不同的 `context`。举例来说，从命令行来启动job，会为每一个job初始化一个JVM，因此每个job会有一个自己的 **JobLauncher**；从web容器的HttpRequest来启动job，一般只是用一个 **JobLauncher** 来异步启动job，http请求会调用这个 **JobLauncher** 来启动它们需要的job。

4.5.1 从命令行启动 Jobs

对于使用企业级调度器来运行job的用户来说，命令行是主要的使用接口。这是因为大多数的调度器是之间工作在操作系统进程上(除了Quartz，否则使用 `NativeJob`)，使用shell脚本来启动。除了shell脚本还有许多脚本语言能启动java进程，如Perl和Ruby，甚至一些构建工具也可以，如ant和maven。但是大多数人都熟悉shell脚本，这个例子主要展示shell脚本。

The CommandLineJobRunner

由于脚本启动job需要开启java虚拟机，那么需要有一个类带有main方法作为操作的主入口点。Spring Batch针对这个需求提供了一个实现：**CommandLineJobRunner**。需要强调的是这只是引导你的应用程序的一种方法，有许多方法能够启动java进程，不应该把这个类视为最终方法。**CommandLineJobRunner**执行四个任务：

- 加载适当的 `ApplicationContext`
- 解析到 `JobParameters` 的命令行参数
- 根据参数确定合适的任务
- 使用在application context(应用上下文)中所提供的 `JobLauncher` 来启动job。

所有这些任务只要提供几个参数就可以完成。以下是要求的参数：

Table 4.1. CommandLineJobRunner arguments

- | | |
|------------|---|
| 1. jobPath | 用于创建 <code>ApplicationContext</code> 的xml文件地址。这个文件包含了完成任务的一切配置。 |
| 2. | |
| 3. jobName | 需要运行的job的名字。 |

参数中必须路径参数在前，任务名参数在后。被设置到JobParameter中的参数必须使

用“`name=value`”的格式：

```
1. bash$ java CommandLineJobRunner endOfDayJob.xml endOfDay schedule.date(date)=2007/05/05
```

大多数情况下在jar中放置一个manifest文件来描述main class，但是直接使用class会比较简洁。还是使用 `domain section`中的‘EndOfDay’例子，第一个参数是‘endOfDayJob.xml’，这是包含了Job和Spring ApplicationContext；第二个参数是‘endOfDay’，指定了Job的名字；最后一个参数‘schedule.date(date)=2007/05/05’会被转换成JobParameters。例子中的xml如下所示：

```
1. <job id="endOfDay">
2.     <step id="step1" parent="simpleStep" />
3. </job>
4.
5. <!--为清晰起见省略了Launcher的详细信息-->
6. <beans:bean id="jobLauncher"
7.     class="org.springframework.batch.core.launch.support.SimpleJobLauncher" />
```

例子很简单，在实际案例中Spring Batch运行一个Job通常有多得多的要求，但是这里展示了 `CommandLineJobRunner` 的两个主要要求：`Job` 和 `JobLauncher`。

ExitCodes

使用企业级调度器通过命令行启动一个批处理任务后，大多数调度器都是在进程级别沉默的工作。这意味着它们只知道一些操作系统进程信息(如它们执行的脚本)。在这种场景下，只能通过返回code来和调度器交流job执行成功还是失败的信息。返回code是返回给调度程序进程的一个数字，用于指示运行结果。最简单的一个例子：0表示成功，1表示失败。更复杂的场景如：job A返回4就启动job B，返回5就启动job C。这种类型的行为被配置在调度器层级，但重要的是像Spring Batch这种处理框架需要为特殊的批处理任务提供一个返回‘Exit Code’数字表达式。在SpringBatch中，退出代码被封装在 `ExitStatus`，具体细节会在Chapter 5中介绍。对于 `exit code`，只需要知道 `ExitStatus` 有一个 `exit code` 属性能够被框架或是开发者设置，作为 `JobLauncher` 返回的 `JobExecution` 的一部分。 `CommandLineJobRunner` 使用 `ExitCodeMapper` 接口将字符串的值转换为数值：

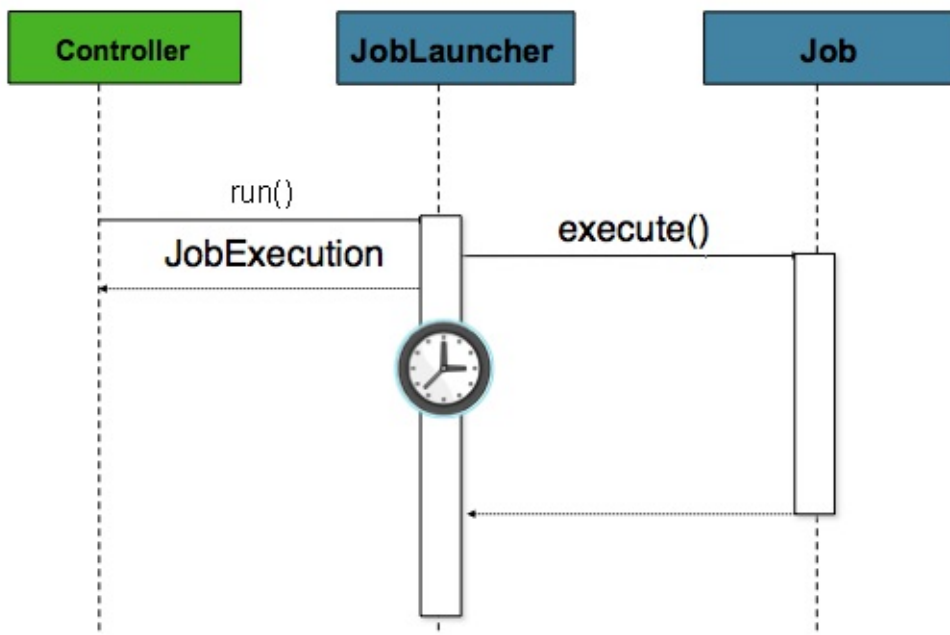
```
1. public interface ExitCodeMapper {
2.
3.     public int intValue(String exitCode);
4.
5. }
```

`ExitCodeMapper` 的基本协议是传入一个字符串，返回一个数字表达式。job运行器默认使用的是 `SimpleJvmExitCodeMapper`，完成返回0，一般错误返回1，上下文找不到job，这类job运行器启动级别的错误则返回2。如果需要比上面三个值更复杂的返回值，就提供自定义 `ExitCodeMapper`

的实现。由于 `CommandLineJobRunner` 是创建 `ApplicationContext` 的类，不能够使用绑定功能，所以所有的值都需要覆盖后使用自动绑定，因此 `ExitCodeMapper` 在 `BeanFactory` 中加载，就会在上下文被创建后注入到job运行器中。而所有需要做的就是提供自己的 `ExitCodeMapper` 描述为 `ApplicationContext` 的一部分，使之能够被运行器加载。

4.5.2 在 Web Container 内部运行 Jobs

过去，像批处理任务这样的离线计算都需要从命令行启动。但是，许多例子(包括报表、点对点任务和web支持)都表明，从HttpRequest启动是一个更好的选择。另外，批处理任务一般都是需要长时间运行，异步启动时最为重要的：



这个例子中的Controller就是spring MVC中的Controller(Spring MVC的信息可以在<http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html> 中查看)。Controller通过使用配置为异步的 (`asynchronously`) JobLauncher启动job后立即返回了JobExecution。job保持运行，这个非阻塞的行为能够让controller在持有HttpRequest时立刻返回。示例如下：

```

1. @Controller
2. public class JobLauncherController {
3.
4.     @Autowired
5.     JobLauncher jobLauncher;
6.
7.     @Autowired
8.     Job job;
9.
  
```

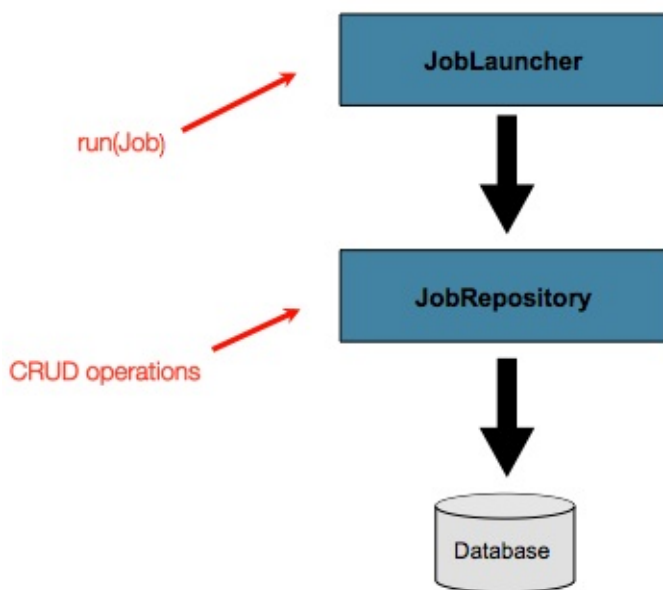
```
10.     @RequestMapping("/jobLauncher.html")
11.     public void handle() throws Exception{
12.         jobLauncher.run(job, new JobParameters());
13.     }
14. }
```

Meta-Data 高级用法

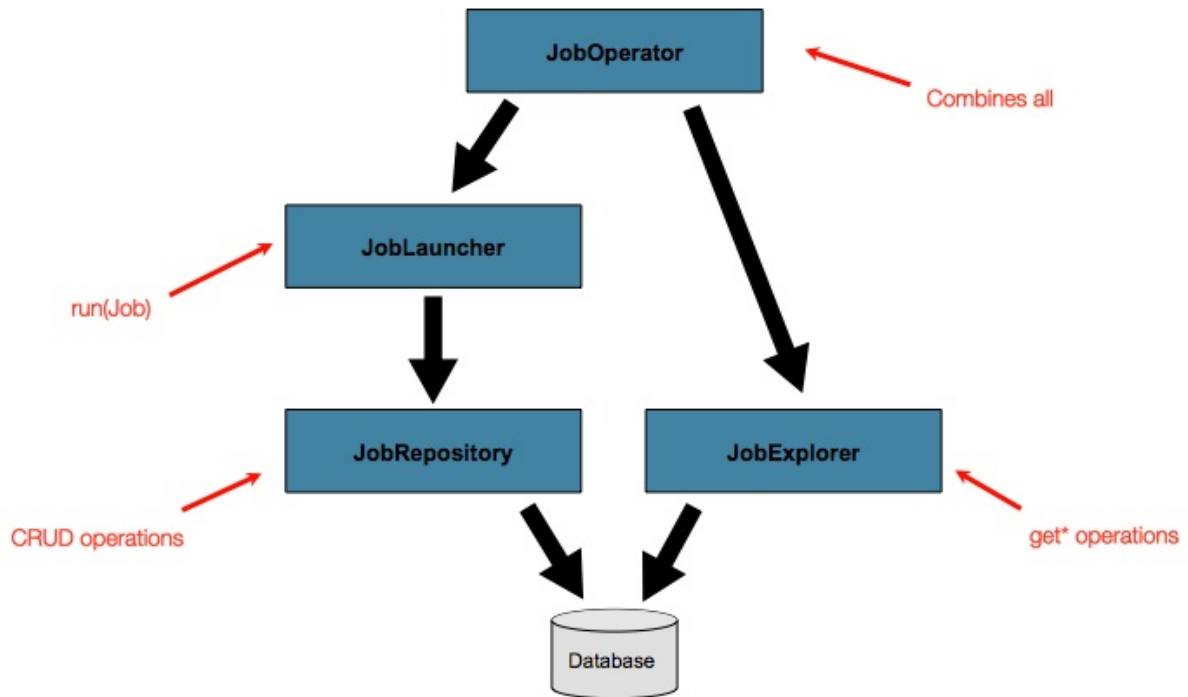
- 4.6 Meta-Data 高级用法
 - 4.6.1 Querying the Repository
 - 4.6.2 JobRegistry
 - 4.6.3 JobOperator
 - 4.6.4 JobParametersIncrementer
 - 4.6.5 Stopping a Job
 - 4.6.6 Aborting a Job

4.6 Meta-Data 高级用法

到目前为止，已经讨论了 **JobLauncher** 和 **JobRepository** 接口，它们展示了简单启动任务，以及批处理领域对象的基本CRUD操作：



一个JobLauncher使用一个JobRepository创建并运行新的JobExecution对象，Job和Step实现随后使用相同的JobRepository在job运行期间去更新相同的JobExecution对象。这些基本的操作能够满足简单场景的需要，但是对于有着数百个任务和复杂定时流程的大型批处理情况来说，就需要使用更高级的方式访问元数据：



接下去会讨论 **JobExplorer** 和 **JobOperator** 两个接口，能够使用更多的功能去查询和修改元数据。

4.6.1 Querying the Repository

在使用高级功能之前，需要最基本的方法来查询repository去获取已经存在的 **JobExecution**。 **JobExplorer** 接口提供了这些功能：

```

1. public interface JobExplorer {
2.
3.     List<JobInstance> getJobInstances(String jobName, int start, int count);
4.
5.     JobExecution getJobExecution(Long executionId);
6.
7.     StepExecution getStepExecution(Long jobExecutionId, Long stepExecutionId);
8.
9.     JobInstance getJobInstance(Long instanceId);
10.
11.     List<JobExecution> getJobExecutions(JobInstance jobInstance);
12.
13.     Set<JobExecution> findRunningJobExecutions(String jobName);
14. }
  
```

上面的代码表示的很明显，**JobExplorer**是一个只读版的**JobRepository**，同**JobRepository**一

样，它也能够很容易配置一个工厂类：

```
1. <bean id="jobExplorer" class="org.spr...JobExplorerFactoryBean"
2.     p:dataSource-ref="dataSource" />
```

([Earlier in this chapter](#)) 之前有提到过，**JobRepository** 能够配置不同的表前缀用来支持不同的版本或是schema。**JobExplorer** 也支持同样的特性：

```
1. <bean id="jobExplorer" class="org.spr...JobExplorerFactoryBean"
2.     p:dataSource-ref="dataSource" p:tablePrefix="BATCH_" />
```

4.6.2 JobRegistry

JobRegistry (父接口为 **JobLocator**) 并非强制使用，它能够协助用户在上下文中追踪job是否可用，也能够应用上下文收集在其他地方(子上下文)创建的job信息。自定义的**JobRegistry**实现常被用于操作job的名称或是其他属性。框架提供了一个基于map的默认实现，能够从job的名称映射到job的实例：

```
1. <bean id="jobRegistry" class="org.spr...MapJobRegistry" />
```

有两种方法自动注册job进**JobRegistry**：使用bean的post处理器或是使用注册生命周期组件。这两种机制在下面描述。

JobRegistryBeanPostProcessor

这是post处理器，能够将job在创建时自动注册进**JobRegistry**：

```
1. <bean id="jobRegistryBeanPostProcessor" class="org.spr...JobRegistryBeanPostProcessor">
2.     <property name="jobRegistry" ref="jobRegistry"/>
3. </bean>
```

并不一定要像例子中给post处理器一个id，但是使用id可以在子context中(比如作为父 bean 定义)也使用post处理器，这样所有的job在创建时都会自动注册进**JobRegistry**。

AutomaticJobRegistrar

这是生命周期组件，用于创建子context以及注册这些子context中的job。这种做法有一个好处，虽然job的名字仍然要求全局唯一，但是job的依赖项可以不用全局唯一，它可以有一个“自然”的名字。例如，创建了一组xml配置文件，每个文件有一个job，每个job的ItemReader都有一个相同的名字(如“reader”)，如果这些文件被导入到一个上下文中，reader的定义会冲突并且互相覆盖。如果使用了自动注册机就能避免这一切发生。这样集成几个不同的应用模块就变得更加容易了：

```

1. <bean class="org.spr...AutomaticJobRegistrar">
2.   <property name="applicationContextFactories">
3.     <bean class="org.spr...ClasspathXmlApplicationContextsFactoryBean">
4.       <property name="resources" value="classpath*/config/job*.xml" />
5.     </bean>
6.   </property>
7.   <property name="jobLoader">
8.     <bean class="org.spr...DefaultJobLoader">
9.       <property name="jobRegistry" ref="jobRegistry" />
10.    </bean>
11.  </property>
12. </bean>

```

注册机有两个主要的属性，一个是ApplicationContextFactory数组(这儿创建了一个简单的factory bean)，另一个是 `jobLoader`。**JobLoader** 负责管理子context的生命周期以及注册任务到JobRegistry。

ApplicationContextFactory 负责创建子 **Context**，大多数情况下像上面那样使用 **ClasspathXmlApplicationContextFactory**。这个工厂类的一个特性是默认情况下他会复制父上下文的一些配置到子上下文。因此如果不变的情况下不需要重新定义子上下文中的 **PropertyPlaceholderConfigurer** 和AOP配置。

在必要情况下，**AutomaticJobRegistrar** 可以和 **JobRegistryBeanPostProcessor** 一起使用。例如，job有可能既定义在父上下文中也定义在子上下文中的情况。

4.6.3 JobOperator

正如前面所讨论的，JobRepository 提供了对元数据的 CRUD 操作，JobExplorer 提供了对元数据的只读操作。然而，这些操作最常用于联合使用诸多的批量操作类，来对任务进行监测，并完成相当多的任务控制功能，比如停止、重启或对任务进行汇总。在Spring Batch 中JobOperator 接口提供了这些操作类型：

```

1. public interface JobOperator {
2.
3.   List<Long> getExecutions(long instanceId) throws NoSuchJobInstanceException;
4.
5.   List<Long> getJobInstances(String jobName, int start, int count)
6.     throws NoSuchJobException;
7.
8.   Set<Long> getRunningExecutions(String jobName) throws NoSuchJobException;
9.
10.  String getParameters(long executionId) throws NoSuchJobExecutionException;
11.
12.  Long start(String jobName, String parameters)

```



```

13.         throws NoSuchJobException, JobInstanceAlreadyExistsException;
14.
15.     Long restart(long executionId)
16.         throws JobInstanceAlreadyCompleteException, NoSuchJobExecutionException,
17.             NoSuchJobException, JobRestartException;
18.
19.     Long startNextInstance(String jobName)
20.         throws NoSuchJobException, JobParametersNotFoundException, JobRestartException,
21.             JobExecutionAlreadyRunningException, JobInstanceAlreadyCompleteException;
22.
23.     boolean stop(long executionId)
24.         throws NoSuchJobExecutionException, JobExecutionNotRunningException;
25.
26.     String getSummary(long executionId) throws NoSuchJobExecutionException;
27.
28.     Map<Long, String> getStepExecutionSummaries(long executionId)
29.         throws NoSuchJobExecutionException;
30.
31.     Set<String> getJobNames();
32.
33. }

```

上图中展示的操作重现了来自其它接口提供的方法，比如JobLauncher, JobRepository, JobExplorer, 以及 JobRegistry。因为这个原因，所提供的JobOperator的实现SimpleJobOperator的依赖项有很多：

```

1. <bean id="jobOperator" class="org.spr...SimpleJobOperator">
2.     <property name="jobExplorer">
3.         <bean class="org.spr...JobExplorerFactoryBean">
4.             <property name="dataSource" ref="dataSource" />
5.         </bean>
6.     </property>
7.     <property name="jobRepository" ref="jobRepository" />
8.     <property name="jobRegistry" ref="jobRegistry" />
9.     <property name="jobLauncher" ref="jobLauncher" />
10. </bean>

```

注意

如果你在JobRepository中设置了表前缀，那么不要忘记在JobExplorer中也做同样设置。

4.6.4 JobParametersIncrementer

JobOperator 的多数方法都是不言自明的，更多详细的说明可以参见该接口的javadoc ([javadoc of the interface](#))。然而startNextInstance方法却有些无所是处。这个方法通常用于启动

Job的一个新的实例。但如果 JobExecution 存在若干严重的问题，同时该Job 需要从头重新启动，那么这时候这个方法就相当有用了。不像JobLauncher，启动新的任务时如果参数不同于任何以往的参数集，这就要求一个新的 JobParameters 对象来触发新的 JobInstance，startNextInstance 方法将使用当前的JobParametersIncrementer绑定到这个任务，并强制其生成新的实例：

```
1. public interface JobParametersIncrementer {
2.     JobParameters getNext(JobParameters parameters);
3. }
```

JobParametersIncrementer 的协议是这样的，当给定一个 **JobParameters** 对象，它将返回填充了所有可能需要的值“下一个” JobParameters 对象。这个策略非常有用，因为框架无需知晓变成“下一个”的JobParameters 做了哪些更改。例如，如果任务参数中只包含一个日期参数，那么当创建下一个实例时，这个值就应该是不是该自增一天？或者一周（如果任务是以周为单位运行的话）？任何包含数值类参数的任务，如果需要对其进行区分，都涉及这个问题，如下：

```
1. public class SampleIncrementer implements JobParametersIncrementer {
2.
3.     public JobParameters getNext(JobParameters parameters) {
4.         if (parameters==null || parameters.isEmpty()) {
5.             return new JobParametersBuilder().addLong("run.id", 1L).toJobParameters();
6.         }
7.         long id = parameters.getLong("run.id",1L) + 1;
8.         return new JobParametersBuilder().addLong("run.id", id).toJobParameters();
9.     }
10. }
```

在该示例中，键值“run.id”用以区分各个JobInstance。如果当前的JobParameters为空（null），它将被视为该Job从未运行过，并同时为其初始化，然后返回。反之，非空的时候自增一个数值，再返回。自增的数值可以在命名空间描述中通过Job的“incrementer”属性进行设置：

```
1. <job id="footballJob" incrementer="sampleIncrementer">
2.     ...
3. </job>
```

4.6.5 Stopping a Job

JobOperator 最常见的作用莫过于停止某个Job：

```
1. Set<Long> executions = jobOperator.getRunningExecutions("sampleJob");
2. jobOperator.stop(executions.iterator().next());
```

关闭不是立即发生的，因为没有办法将一个任务立刻强制停掉，尤其是当任务进行到开发人员自己的代码段时，框架在此刻是无能为力的，比如某个业务逻辑处理。而一旦控制权还给了框架，它会立刻设置当前 `StepExecution` 为 `BachStatus.STOPPED`，意为停止，然后保存，最后在完成前对 `JobExecution`进行相同的操作。

4.6.6 Aborting a Job

一个job的执行过程当执行到FAILED状态之后，如果它是可重启的，它将会被重启。如果任务的执行过程状态是ABANDONED，那么框架就不会重启它。ABANDONED状态也适用于执行步骤，使得它们可以被跳过，即便是在一个可重启的任务执行之中：如果任务执行过程中碰到在上一次执行失败后标记为ABANDONED的步骤，将会跳过该步骤直接到下一步(这是由任务流定义和执行步骤的退出码决定的)。

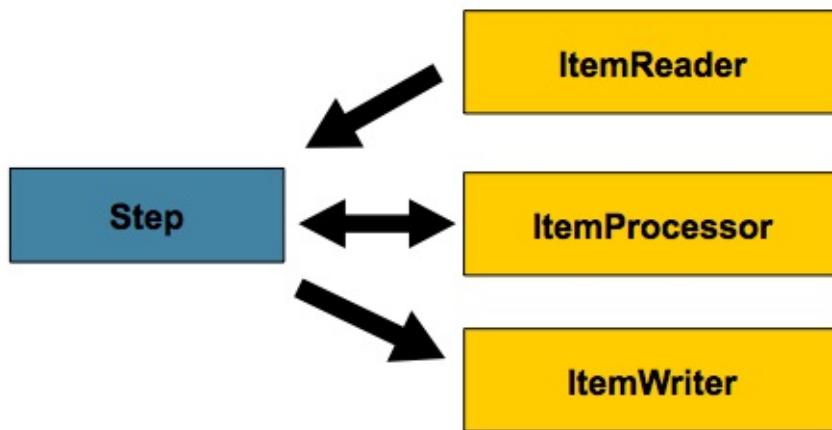
如果当前的系统进程死掉了(“kill -9”或系统错误)，job自然也不会运行，但JobRepository是无法侦测到这个错误的，因为进程死掉之前没有对它进行任何通知。你必须手动的告诉它，你知道任务已经失败了还是说考虑放弃这个任务(设置它的状态为FAILED或ABANDONED)-这是业务逻辑层的事情，无法做到自动决策。只有在不可重启的任务中才需要设置为FAILED状态，或者你知道重启后数据还是有效的。Spring Batch Admin中有一系列工具JobService，用以取消正在进行执行的任务。

配置Step

- [配置Step](#)

配置Step

正如在[Batch Domain Language](#)中叙述的，Step是一个独立封装域对象，包含了所有定义和控制实际处理信息批任务的序列。这是一个比较抽象的描述，因为任意一个Step的内容都是开发者自己编写的Job。一个Step的简单或复杂取决于开发者的意愿。一个简单的Step也许是从本地文件读取数据存入数据库，写很少或基本无需写代码。一个复杂的Step也许有复杂的业务规则（取决于所实现的方式），并作为整个流程的一部分。

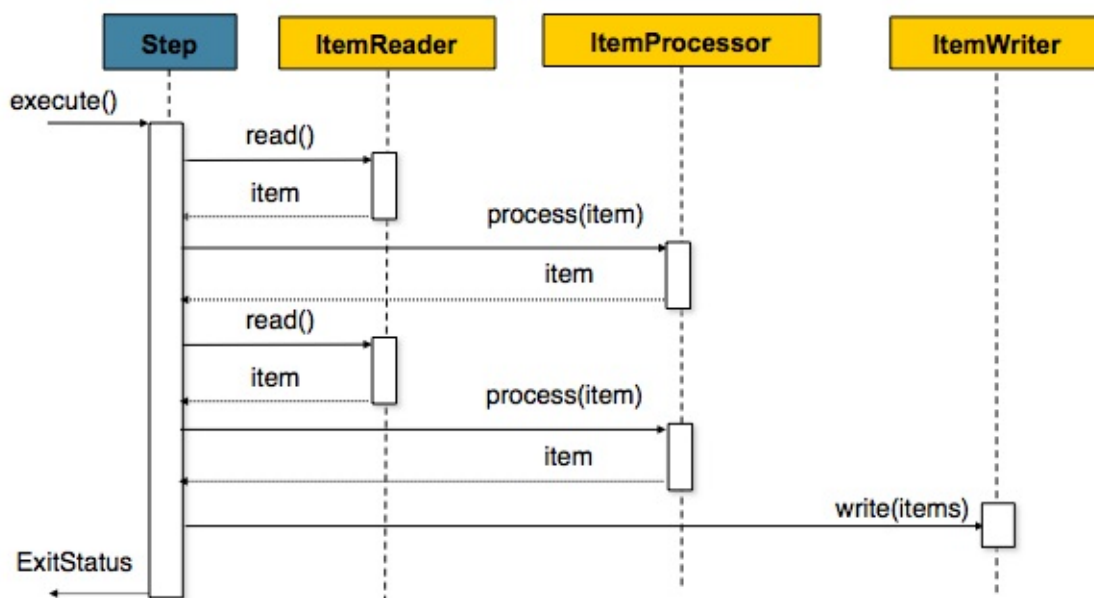


面向块的流程

- 面向块的流程
 - 配置Step
 - 继承自一个父Step

面向块的流程

Spring Batch最通用的实现方式是使用“面向块”的处理风格。面向块处理是指在一个事务范围内，一次性读取数据，创建被输出的“块”。ItemReader读取一条项目，通过ItemProcessor处理，并整合。当处理完所有项目后，整个块将由ItemWriter输出，然后提交该事务。



下面一段代码展示上面提示的内容：

```

1. List items = new ArrayList();
2. for(int i = 0; i < commitInterval; i++){
3.     Object item = itemReader.read()
4.     Object processedItem = itemProcessor.process(item);
5.     items.add(processedItem);
6. }
7. itemWriter.write(items);
  
```

配置Step

尽管配置一个Step的的依赖列表比较简短，但它是一个包含许多协作者的极其复杂的类。为了简化Spring Batch配置，如下：

上面配置表示创建一个面向条目step，必要的依赖：

- reader - ItemReader提供待处理的条目。
- writer - 由ItemReader提供并被ItemWriter处理的条目。
- transaction-manager - Spring中的PlatformTransactionManager，将用管理事务的开始和提交。
- job-repository - JobRepository用于在事务中存储StepExecution和ExecutionContext。在一行，它作为元素的一个属性；对于一个独立的step，定义一个作为它的属性。
- commit-interval - 在事务提交之前，将被处理的条目个数。

应当注意的是，job-repository这里默认设置为“jobRepository”，transaction-manager默认设置为“transactionManger”。此外，这里ItemProcessor是可选的，不是必须，因为该条目可能从reader读取并直接传给writer。

继承自一个父Step

ItemReaders和ItemWriters

- [ItemReaders和ItemWriters](#)

ItemReaders和ItemWriters

所有的批处理都可以描述为最简单的形式：读取大量的数据，执行某种类型的计算/转换，以及写出执行结果。Spring Batch 提供了三个主要接口来辅助执行大量的读取与写出：**ItemReader**，**ItemProcessor** 和 **ItemWriter**。

ItemReader

- [6.1 ItemReader](#)

6.1 ItemReader

最简单的概念，**ItemReader** 就是一种从各个输入源读取数据，然后提供给后续步骤的方式。最常见的例子包括：

- **Flat File** Flat File Item Readers 从纯文本文件中读取一行行的数据，存储数据的纯文本文件通常具有固定的格式，并且使用某种特殊字符来分隔每条记录中的各个字段(例如逗号, Comma)。
- **XML** XML ItemReaders 独立地处理XML，包括用于解析、映射和验证对象的技术。还可以对输入数据的XML文件执行XSD schema验证。
- **Database** 数据库就是对请求返回结果集的资源，结果集可以被映射转换为需要处理的对象。默认的SQL ItemReaders调用一个 `RowMapper` 来返回对象，并跟踪记录当前行，以备有重启的情况，存储基本统计信息，并提供一些事务增强特性，关于事物将在稍后解释。

虽然有各种各样的数据输入方式，但本章我们只关注最基本的部分。关于详细的可用 ItemReaders 列表可以参照 [附录A](#)

ItemReader 是一个通用输入操作的基本接口：

```

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

`read` 是**ItemReader**中最根本的方法；每次调用它都会返回一个 Item 或 null(如果没有更多 item)。每个 item 条目，一般对应文件中的一行(line)，或者对应数据库中的一行(row)，也可以是XML文件中的一个元素(element)。一般来说，这些item都可以被映射为一个可用的domain 对象(如 Trade, User 等等)，但也不是强制要求(最偷懒的方式，返回一个Map)。

一般约定 **ItemReader** 接口的实现都是向前型的(forward only)。但如果底层资源是事务性质的(如JMS队列)，并且发生回滚(rollback)，那么下一次调用 `read` 方法有可能会返回和前次逻辑上相等的结果(对象)。值得一提的是，处理过程中如果没有items，**ItemReader** 不应该抛出异常。例如，数据库 **ItemReader** 配置了一条查询语句，返回结果数为0，则第一次调用read方法将返回null。

ItemWriter

- [6.2 ItemWriter](#)

6.2 ItemWriter

ItemWriter 在功能上类似于 **ItemReader**, 但属于相反的操作。资源仍然需要定位, 打开和关闭, 区别就在于在于 **ItemWriter** 执行的是写入操作(write out), 而不是读取。在使用数据库或队列的情况下, 写入操作对应的是插入(`insert`), 更新(`update`), 或发送(`send`)。序列化输出的格式依赖于每个批处理作业自己的定义。

和 **ItemReader** 接口类似, **ItemWriter** 也是个相当通用的接口:

```
1. public interface ItemWriter<T> {  
2.  
3.     void write(List<? extends T> items) throws Exception;  
4.  
5. }
```

类比于 **ItemReader** 中的 `read`, `write` 方法是 **ItemWriter** 接口的根本方法; 只要传入的 `items` 列表是打开的, 那么它就会尝试着将其写入(write out)。因为一般来说, `items` 将要被批量写入到一起, 然后再输出, 所以 `write` 方法接受一个 `List` 参数, 而不是单个对象(`item`)。 `list` 被输出后, 在 `write` 方法返回(return)之前, 对缓冲执行刷出(`flush`)操作是很必要的。例如, 如果使用 `Hibernate DAO` 时, 对每个对象要调用一次 `DAO` 写操作, 操作完成之后, 方法 `return` 之前, `writer` 就应该关闭 `hibernate` 的 `Session` 会话。

ItemProcessor

- 6.3 ItemProcessor
 - 6.3.1 Chaining ItemProcessors
 - 6.3.2 Filtering Records
 - 6.3.3 容错(Fault Tolerance)

6.3 ItemProcessor

ItemReader 和 **ItemWriter** 接口对于每个任务来说都是非常必要的，但如果想要在写出数据之前执行某些业务逻辑操作时要怎么办呢？

一个选择是对读取(reading)和写入(writing)使用组合模式(composite pattern)：创建一个 **ItemWriter** 的子类实现，内部包含另一个 **ItemWriter** 对象的引用(对于 **ItemReader** 也是类似的)。示例如下：

```

1. public class CompositeItemWriter<T> implements ItemWriter<T> {
2.
3.     ItemWriter<T> itemWriter;
4.
5.     public CompositeItemWriter(ItemWriter<T> itemWriter) {
6.         this.itemWriter = itemWriter;
7.     }
8.
9.     public void write(List<? extends T> items) throws Exception {
10.        // ... 此处可以执行某些业务逻辑
11.        itemWriter.write(item);
12.    }
13.
14.    public void setDelegate(ItemWriter<T> itemWriter){
15.        this.itemWriter = itemWriter;
16.    }
17. }
```

上面的类中包含了另一个**ItemWriter**引用,通过代理它来实现某些业务逻辑。这种模式对于**ItemReader**也是一样的道理,但也可能持有内部 **ItemReader** 所拥有的多个数据输入对象的引用。在**ItemWriter**中如果我们想要自己控制 `write` 的调用也可能需要持有其他引用。

但假如我们只想在对象实际被写入之前“改造”一下传入的item,就没必要实现**ItemWriter**和执行 `write` 操作:我们只需要这个将被修改的item对象而已。对于这种情况, Spring Batch提供了 **ItemProcessor** 接口:

```

1. public interface ItemProcessor<I, O> {
2.
```

```

3.     0 process(I item) throws Exception;
4. }

```

ItemProcessor 非常简单；传入一个对象，对其进行某些处理/转换，然后返回另一个对象（也可以是同一个）。传入的对象和返回的对象类型可以一样，也可以不一致。关键点在于处理过程中可以执行一些业务逻辑操作，当然这完全取决于开发者怎么实现它。一个 **ItemProcessor** 可以被直接关联到某个 Step(步骤)，例如，假设 **ItemReader** 的返回类型是 **Foo** 【译者注：Foo, Bar 一类的词就和 BalaBala 一样，没什么实际意义】，而在写出之前需要将其转换成类型 **Bar** 的对象。就可以编写一个 **ItemProcessor** 来执行这种转换：

```

1. public class Foo {}
2.
3. public class Bar {
4.     public Bar(Foo foo) {}
5. }
6.
7. public class FooProcessor implements ItemProcessor<Foo,Bar>{
8.     public Bar process(Foo foo) throws Exception {
9.         //执行某些操作,将 Foo 转换为 Bar对象
10.        return new Bar(foo);
11.    }
12. }
13.
14. public class BarWriter implements ItemWriter<Bar>{
15.     public void write(List<? extends Bar> bars) throws Exception {
16.         //write bars
17.     }
18. }

```

在上面的简单示例中，有两个类：**Foo** 和 **Bar**，以及实现了 **ItemProcessor** 接口的 **FooProcessor** 类。因为是 demo，所以转换很简单，在实际使用中可能执行转换为任何类型，响应的操作请读者根据需要自己编写。**BarWriter** 将被用于写出 **Bar** 对象，如果传入其他类型的对象可能会抛出异常。同样，如果 **FooProcessor** 传入的参数不是 **Foo** 也会抛出异常。**FooProcessor** 可以注入到某个 Step 中：

```

1. <job id="ioSampleJob">
2.     <step name="step1">
3.         <tasklet>
4.             <chunk reader="fooReader" processor="fooProcessor" writer="barWriter"
5.                 commit-interval="2"/>
6.         </tasklet>
7.     </step>
8. </job>

```

6.3.1 Chaining ItemProcessors

在很多情况下执行单个转换就可以了，但假如想要将多个 **ItemProcessors** “串联(chain)” 在一起要怎么实现呢？我们可以使用前面提到的组合模式(composite pattern)来完成。接着前面单一转换的示例，我们将**Foo**转换为**Bar**，然后再转换为**FooBar**类型，并执行写出：

```

1. public class Foo {}
2.
3. public class Bar {
4.     public Bar(Foo foo) {}
5. }
6.
7. public class FooBar{
8.     public FooBar(Bar bar) {}
9. }
10.
11. public class FooProcessor implements ItemProcessor<Foo,Bar>{
12.     public Bar process(Foo foo) throws Exception {
13.         //Perform simple transformation, convert a Foo to a Bar
14.         return new Bar(foo);
15.     }
16. }
17.
18. public class BarProcessor implements ItemProcessor<Bar,FooBar>{
19.     public FooBar process(Bar bar) throws Exception {
20.         return new FooBar(bar);
21.     }
22. }
23.
24. public class FooBarWriter implements ItemWriter<FooBar>{
25.     public void write(List<? extends FooBar> items) throws Exception {
26.         //write items
27.     }
28. }

```

可以将 **FooProcessor** 和 **BarProcessor** “串联”在一起来生成 **FooBar** 对象，如果用 Java代码表示，那就像下面这样：

```

1. CompositeItemProcessor<Foo,FooBar> compositeProcessor = new CompositeItemProcessor<Foo,FooBar>
   ();
2. List itemProcessors = new ArrayList();
3. itemProcessors.add(new FooTransformer());
4. itemProcessors.add(new BarTransformer());
5. compositeProcessor.setDelegates(itemProcessors);

```

就和前面的示例类似, 复合处理器也可以配置到Step中:

```

1. <job id="ioSampleJob">
2.     <step name="step1">
3.         <tasklet>
4.             <chunk reader="fooReader" processor="compositeProcessor" writer="foobarWriter"
5.                 commit-interval="2"/>
6.         </tasklet>
7.     </step>
8. </job>
9.
10. <bean id="compositeItemProcessor"
11.     class="org.springframework.batch.item.support.CompositeItemProcessor">
12.     <property name="delegates">
13.         <list>
14.             <bean class="..FooProcessor" />
15.             <bean class="..BarProcessor" />
16.         </list>
17.     </property>
18. </bean>

```

6.3.2 Filtering Records

item processor 的典型应用就是在数据传给ItemWriter之前进行过滤(filter out)。过滤(Filtering)是一种有别于跳过(skipping)的行为; skipping表明某几行记录是无效的,而filtering 则只是表明某条记录不应该写入(written)。

例如, 某个批处理作业, 从一个文件中读取三种不同类型的记录: 准备 insert 的记录、准备 update 的记录, 需要 delete 的记录。如果系统中不允许删除记录, 那么我们肯定不希望将“delete”类型的记录传递给 **ItemWriter**。但因为这些记录又不是损坏的信息(bad records), 我们只想将其过滤掉, 而不是跳过。因此, ItemWriter只会收到“insert”和“update”的记录。

要过滤某条记录, 只需要 **ItemProcessor** 返回“`null`”即可。框架将自动检测结果为“`null`”的情况, 不会将该item 添加到传给**ItemWriter**的list中。像往常一样, 在 **ItemProcessor** 中抛出异常将会导致跳过(skip)。

6.3.3 容错(Fault Tolerance)

当某一个分块回滚时, 读取后已被缓存的那些item可能会被重新处理。如果一个step被配置为支持容错(通常使用 skip跳过 或 retry重试处理), 使用的所有 `ItemProcessor` 都应该实现为幂等的(idempotent)。通常ItemProcessor对已经处理过的输入数据不执行任何修改, 而只更新需要处

理的实例。

ItemStream

- [6.4 ItemStream](#)

6.4 ItemStream

ItemReader 和 **ItemWriter** 都为各自的目的服务，但他们之间有一个共同点，就是都需要与另一个接口配合。一般来说，作为批处理作业作用域范围的一部分，**readers** 和 **writers** 都需要打开(**open**)，关闭(**close**)，并需要某种机制来持久化自身的状态：

```
1. public interface ItemStream {
2.
3.     void open(ExecutionContext executionContext) throws ItemStreamException;
4.
5.     void update(ExecutionContext executionContext) throws ItemStreamException;
6.
7.     void close() throws ItemStreamException;
8. }
```

在描述每种方法之前，我们应该提到**ExecutionContext**。**ItemReader**的客户端也应该实现**ItemStream**，在任何 `read` 之前调用 `open` 以打开需要的文件或数据库连接等资源。实现**ItemWriter**也有类似的限制/约束，即需要同时实现**ItemStream**。如第2章所述，如果将数据存放在**ExecutionContext**中，那么它可以在某个时刻用来启动 **ItemReader** 或 **ItemWriter**，而不是在初始状态时。对应的，应该确保在调用 `open` 之后的适当位置调用 `close` 来安全地释放所有分配的资源。调用 `update` 主要是为了确保当前持有的所有状态都被加载到所提供的**ExecutionContext**中。`update` 一般在提交之前调用，以确保当前状态被持久化到数据库之中。

在特殊情况下，**ItemStream** 的客户端是一个 **Step**(由 Spring Batch Core 决定)，会为每个 **StepExecution** 创建一个 **ExecutionContext**，以允许用户存储特定部分的执行状态，一般来说如果同一个**JobInstance**重启了，则预期它将会在重启后被返回。对于熟悉 Quartz的人来说，逻辑上非常像是 Quartz 的 **JobDataMap**。

代理模式与Step注册

- 6.5 委托模式(Delegate Pattern)与注册Step

6.5 委托模式(Delegate Pattern)与注册Step

请注意, **CompositeItemWriter**是委托模式的一个示例,这在Spring Batch中很常见的。委托自身可以实现回调接口 **StepListener**。如果实现了,那么他们就会被当作Job中Step的一部分与Spring Batch Core 结合使用,然后他们基本上必定需要手动注册到 **Step** 中。

一个 reader, writer, 或 processor,如果实现了 **ItemStream** / **StepListener**接口,就会被自动组装到 Step 中。但因为 delegates 并不为 **Step** 所知,因此需要被注入(作为 listeners监听器或streams流,或两者都可):

```
1. <job id="ioSampleJob">
2.   <step name="step1">
3.     <tasklet>
4.       <chunk reader="fooReader" processor="fooProcessor" writer="compositeItemWriter"
5.         commit-interval="2">
6.         <streams>
7.           <stream ref="barWriter" />
8.         </streams>
9.       </chunk>
10.    </tasklet>
11.  </step>
12. </job>
13.
14. <bean id="compositeItemWriter" class="...CustomCompositeItemWriter">
15.   <property name="delegate" ref="barWriter" />
16. </bean>
17.
18. <bean id="barWriter" class="...BarWriter" />
```

纯文本文件

- [6.6 纯文本平面文件\(Flat Files\)](#)

6.6 纯文本平面文件(Flat Files)

最常见的批量数据交换机制是使用纯文本平面文件(flat file)。XML由统一约定好的标准来定义文件结构(即XSD),与XML等格式不同,想要阅读纯文本平面文件必须先了解其组成结构。一般来说,纯文本平面文件分两种类型:有分隔的类型(Delimited)与固定长度类型(Fixed Length)。有分隔的文件中各个字段由分隔符进行间隔,比如英文逗号(,)。而固定长度类型的文件每个字段都有固定的长度。

字段集合

- [6.6.1 The FieldSet\(字段集\)](#)

6.6.1 The FieldSet(字段集)

当在Spring Batch中使用纯文本文件时，不管是将其作为输入还是输出，最重要的一个类就是 **FieldSet**。许多架构和类库会抽象出一些方法/类来辅助你从文件读取数据，但是这些方法通常返回 `String` 或者 `String[]` 数组，很多时候这确实是些半成品。而 **FieldSet** 是Spring Batch中专门用来将文件绑定到字段的抽象。它允许开发者和使用数据库差不多的方式来使用数据输入文件。 `FieldSet` 在概念上非常类似于Jdbc的 `ResultSet`。 `FieldSet` 只需要一个参数：即 token数组 `String[]`。另外，您还可以配置字段的名称，然后就可以像使用 `ResultSet` 一样，使用 `index` 或者 `name` 都可以取得对应的值：

```
1. String[] tokens = new String[]{"foo", "1", "true"};
2. FieldSet fs = new DefaultFieldSet(tokens);
3. String name = fs.readString(0);
4. int value = fs.readInt(1);
5. boolean booleanValue = fs.readBoolean(2);
```

在 **FieldSet** 接口可以返回很多类型的对象/数据，如 `Date` , `long` , `BigDecimal` 等。`FieldSet` 最大的优势在于,它对文本输入文件 提供了统一的解析。不是每个批处理作业采用不同的方式进行解析,而一直是一致的,不论是在处理格式异常引起的错误,还是在进行简单的数据转换。

FlatFileItemReader

- 6.6.2 FlatFileItemReader

6.6.2 FlatFileItemReader

译注:

本文中 将 *Flat File* 翻译为“平面文件”，这是一种没有特殊格式的非二进制的文件，里面的内容没有相对关系结构的记录。

平面文件(flat file)是最多包含二维(表格)数据的任意类型的文件。在 Spring Batch 框架中 **FlatFileItemReader** 类负责读取平面文件，该类提供了用于读取和解析平面文件的基本功能。FlatFileItemReader 主要依赖两个东西：**Resource** 和 **LineMapper**。LineMapper接口将在下一节详细讨论。 `resource` 属性代表一个 Spring Core Resource(Spring核心资源)。关于如何创建这一类 bean 的文档可以参考 [Spring框架, Chapter 5.Resources](#)。所以本文档就不再深入讲解创建 Resource 对象的细节。但可以找到一个文件系统资源的简单示例，如下所示：

```
1. Resource resource = new FileSystemResource("resources/trades.csv");
```

在复杂的批处理环境中，目录结构通常由EAI基础设施管理，并且会建立放置区(drop zones)，让外部接口将文件从ftp移动到批处理位置，反之亦然。文件移动工具(File moving utilities)超出了spring batch架构的范畴，但在批处理作业中包括文件移动步骤这种事情那也是很常见的。批处理架构只需要知道如何定位需要处理的文件就足够了。Spring Batch 将会从这个起始点开始，将数据传输给数据管道。当然，Spring Integration也提供了很多这一类的服务。

FlatFileItemReader 中的其他属性让你可以进一步指定数据如何解析：

Table 6.1. FlatFileItemReader 的属性(Properties)

属性(Property)	类型(Type)	说明(Description)
comments	String[]	指定行前缀，用来表明哪些是注释行
encoding	String	指定使用哪种文本编码 - 默认值为 "ISO-8859-1"
lineMapper	LineMapper	将一个 <code>String</code> 转换为相应的 <code>Object</code> .
linesToSkip	int	在文件顶部有多少行需要跳过/忽略
recordSeparatorPolicy	RecordSeparatorPolicy	记录分拆策略，用于确定行尾，以及在引号之中时，如何处理跨行的内容。
resource	Resource	从哪个资源读取数据。
skippedLinesCallback	LineCallbackHandler	忽略输入文件中某些行时，会将忽略行的原始内容传递给这个回调接口。如果 <code>linesToSkip</code> 设置为 2 ，那么这个接口就会被调用 2 次。

strict	boolean	如果处于严格模式(strict mode), reader 在 ExecutionContext 中执行时, 如果输入资源不存在, 则抛出异常.
--------	---------	--

LineMapper

就如同 **RowMapper** 在底层根据 `ResultSet` 构造一个 `Object` 并返回, 平面文件处理过程中也需要将一行 `String` 转换并构造成为 `Object` :

```

1. public interface LineMapper<T> {
2.
3.     T mapLine(String line, int lineNumber) throws Exception;
4.
5. }
```

基本的约定是, 给定当前行以及和它关联的行号(line number), mapper 应该能够返回一个领域对象。这类似于在 `RowMapper` 中每一行也有一个 line number 相关联, 正如 `ResultSet` 中的每一行(Row)都有其绑定的 row number。这允许行号能被绑定到生成的领域对象以方便比较(identity comparison)或者更方便进行日志记录。

但与 `RowMapper` 不同的是, `LineMapper` 只能取得原始行的String值, 正如上面所说, 给你的是一个半成品。这行文本值必须先被解析为 `FieldSet`, 然后才可以映射为一个对象, 如下所述。

LineTokenizer

对将每一行输入转换为 `FieldSet` 这种操作的抽象是很有必要的, 因为可能会有各种平面文件格式需要转换为 `FieldSet`。在Spring Batch中, 对应的接口是 **LineTokenizer** :

```

1. public interface LineTokenizer {
2.
3.     FieldSet tokenize(String line);
4.
5. }
```

使用 **LineTokenizer** 的约定是, 给定一行输入内容(理论上 `String` 可以包含多行内容), 返回一个表示该行的 `FieldSet` 对象。这个 `FieldSet` 接着会传递给 **FieldSetMapper**。Spring Batch 包括以下LineTokenizer实现:

- `DelimitedLineTokenizer` 适用于处理使用分隔符(delimiter)来分隔一条数据中各个字段的文件。最常见的分隔符是逗号(comma), 但管道或分号也经常使用。
- `FixedLengthTokenizer` 适用于记录中的字段都是“固定宽度(fixed width)”的文件。每种记录类型中, 每个字段的宽度必须先定义。
- `PatternMatchingCompositeLineTokenizer` 通过使用正则模式匹配, 来决定对特定的某一行应该使用

LineTokenizers 列表中的哪一个来执行字段拆分。

FieldSetMapper

FieldSetMapper 接口只定义了一个方法，`mapFieldSet`，这个方法接收一个 **FieldSet** 对象，并将其内容映射到一个 **object** 中。根据作业需要，这个对象可以是自定义的 **DTO**，领域对象，或者是简单数组。**FieldSetMapper** 与 **LineTokenizer** 结合使用以将资源文件中的一行数据转化为所需类型的对象：

```
1. public interface FieldSetMapper<T> {
2.
3.     T mapFieldSet(FieldSet fieldSet);
4.
5. }
```

这和 **JdbcTemplate** 中的 **RowMapper** 是一样的道理。

DefaultLineMapper

既然读取平面文件的接口已经定义好了，那很明显我们需要执行以下三个步骤：

1. 从文件中读取一行。
2. 将读取的字符串传给 `LineTokenizer#tokenize()` 方法，以获取一个 **FieldSet**。
3. 将解析后的 **FieldSet** 传给 **FieldSetMapper**，然后将 `ItemReader#read()` 方法执行的结果返回给调用者。

上面的两个接口代表了两个不同的任务：将一行文本转换为 **FieldSet**，以及把 **FieldSet** 映射为一个领域对象。因为 **LineTokenizer** 的输入对应着 **LineMapper** 的输入（一行），并且 **FieldSetMapper** 的输出对应着 **LineMapper** 的输出，所以 **SpringBatch** 提供了一个使用 **LineTokenizer** 和 **FieldSetMapper** 的默认实现。**DefaultLineMapper** 就是大多数情况下用户所需要的：

```
1. public class DefaultLineMapper<T> implements LineMapper<T>, InitializingBean {
2.
3.     private LineTokenizer tokenizer;
4.
5.     private FieldSetMapper<T> fieldSetMapper;
6.
7.     public T mapLine(String line, int lineNumber) throws Exception {
8.         return fieldSetMapper.mapFieldSet(tokenizer.tokenize(line));
9.     }
10.
11.     public void setLineTokenizer(LineTokenizer tokenizer) {
12.         this.tokenizer = tokenizer;
13.     }
14. }
```

```

15.     public void setFieldSetMapper(FieldSetMapper<T> fieldSetMapper) {
16.         this.fieldSetMapper = fieldSetMapper;
17.     }
18. }

```

上面的功能由一个默认实现类来提供,而不是 reader 本身内置的(以前版本的框架这样干),让用户可以更灵活地控制解析过程,特别是需要访问原始行的时候。

文件分隔符读取简单示例

下面的例子用来说明一个实际的领域情景。这个批处理作业将从如下文件中读取 football player(足球运动员) 信息:

```

1. ID,lastName,firstName,position,birthYear,debutYear
2. "AbduKa00,Abdul-Jabbar,Karim,rb,1974,1996",
3. "AbduRa00,Abdullah,Rabih,rb,1975,1999",
4. "AberWa00,Abercrombie,Walter,rb,1959,1982",
5. "AbraDa00,Abramowicz,Danny,wr,1945,1967",
6. "AdamBo00,Adams,Bob,te,1946,1969",
7. "AdamCh00,Adams,Charlie,wr,1979,2003"

```

该文件的内容将被映射为领域对象 **Player**:

```

1. public class Player implements Serializable {
2.
3.     private String ID;
4.     private String lastName;
5.     private String firstName;
6.     private String position;
7.     private int birthYear;
8.     private int debutYear;
9.
10.    public String toString() {
11.        return "PLAYER:ID=" + ID + ",Last Name=" + lastName +
12.            ",First Name=" + firstName + ",Position=" + position +
13.            ",Birth Year=" + birthYear + ",DebutYear=" +
14.            debutYear;
15.    }
16.
17.    // setters and getters...
18. }

```

为了将 FieldSet 映射为 Player 对象,需要定义一个 `FieldSetMapper`, 返回 player 对象:


```

1. protected static class PlayerFieldSetMapper implements FieldSetMapper<Player> {
2.     public Player mapFieldSet(FieldSet fieldSet) {
3.         Player player = new Player();
4.
5.         player.setID(fieldSet.readString(0));
6.         player.setLastName(fieldSet.readString(1));
7.         player.setFirstName(fieldSet.readString(2));
8.         player.setPosition(fieldSet.readString(3));
9.         player.setBirthYear(fieldSet.readInt(4));
10.        player.setDebutYear(fieldSet.readInt(5));
11.
12.        return player;
13.    }
14. }

```

然后就可以通过正确构建一个 `FlatFileItemReader` ，调用 `read` 方法来读取文件：

```

1. FlatFileItemReader<Player> itemReader = new FlatFileItemReader<Player>();
2. itemReader.setResource(new FileSystemResource("resources/players.csv"));
3. //DelimitedLineTokenizer defaults to comma as its delimiter
4. LineMapper<Player> lineMapper = new DefaultLineMapper<Player>();
5. lineMapper.setLineTokenizer(new DelimitedLineTokenizer());
6. lineMapper.setFieldSetMapper(new PlayerFieldSetMapper());
7. itemReader.setLineMapper(lineMapper);
8. itemReader.open(new ExecutionContext());
9. Player player = itemReader.read();

```

每调用一次 `read` 方法，都会读取文件中的一行，并返回一个新的 `Player` 对象。如果到达文件结尾，则会返回 `null` 。

根据 Name 映射 Fields

有一个额外的功能，`DelimitedLineTokenizer` 和 `FixedLengthTokenizer` 都支持，在功能上类似于 Jdbc 的 `ResultSet`。字段的名称可以注入到这些 `LineTokenizer` 实现以提高映射函数的读取能力。首先，平面文件中所有字段的列名会注入给 `tokenizer`：

```

1. tokenizer.setNames(new String[] {"ID",
    "lastName", "firstName", "position", "birthYear", "debutYear"});

```

`FieldSetMapper` 可以像下面这样使用此信息：

```

1. public class PlayerMapper implements FieldSetMapper<Player> {
2.     public Player mapFieldSet(FieldSet fs) {
3.
4.         if(fs == null){

```

```

5.         return null;
6.     }
7.
8.     Player player = new Player();
9.     player.setID(fs.readString("ID"));
10.    player.setLastName(fs.readString("lastName"));
11.    player.setFirstName(fs.readString("firstName"));
12.    player.setPosition(fs.readString("position"));
13.    player.setDebutYear(fs.readInt("debutYear"));
14.    player.setBirthYear(fs.readInt("birthYear"));
15.
16.    return player;
17. }
18. }

```

将 FieldSet 字段映射为 Domain Object

很多时候，创建一个 FieldSetMapper 就跟 JdbcTemplate 里编写 RowMapper 一样繁琐。Spring Batch通过使用JavaBean规范，提供了一个 FieldSetMapper 来自动将字段映射到对应 setter的属性域。还是使用足球的例子，**BeanWrapperFieldSetMapper** 的配置如下所示：

```

1. <bean id="fieldSetMapper"
2.     class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
3.     <property name="prototypeBeanName" value="player" />
4. </bean>
5.
6. <bean id="player"
7.     class="org.springframework.batch.sample.domain.Player"
8.     scope="prototype" />

```

对于 FieldSet 中的每个条目(entry)，mapper都会在Player对象的新实例中查找相应的 setter (因此,需要指定 prototype scope)，和 Spring容器 查找 setter匹配属性名是一样的方式。FieldSet 中每个可用的字段都会被映射，然后返回组装好的 Player 对象,不需要再手写代码。

Fixed Length File Formats

到这一步,我们讨论了带分隔符的文件，但实际应用中可能只有一半左右是这种文件。还有很多机构使用固定长度形式的平面文件。固定长度文件的示例如下：

```

1. UK21341EAH4121131.11customer1
2. UK21341EAH4221232.11customer2
3. UK21341EAH4321333.11customer3
4. UK21341EAH4421434.11customer4
5. UK21341EAH4521535.11customer5

```

虽然看起来像是一个很长的字段,但实际上代表了4个分开的字段:

1. ISIN : 唯一标识符,订购的商品编码 - 占12字符。
2. Quantity : 订购的商品数量 - 占3字符。
3. Price : 商品的价格 - 占5字符。
4. Customer : 订购商品的顾客Id - 占9字符。

配置好 `FixedLengthLineTokenizer` 以后, 每个字段的长度必须用范围(range)的形式指定:

```
1. <bean id="fixedLengthLineTokenizer"
2.     class="org.springframework.batch.io.file.transform.FixedLengthTokenizer">
3.     <property name="names" value="ISIN,Quantity,Price,Customer" />
4.     <property name="columns" value="1-12, 13-15, 16-20, 21-29" />
5. </bean>
```

因为 `FixedLengthLineTokenizer` 使用的也是 `LineTokenizer` 接口, 所以返回值同样是 `FieldSet`, 和使用分隔符基本上是一样的。这也就可以使用同样的方式来处理其输出, 例如使用 `BeanWrapperFieldSetMapper`。

注意

要支持上面这种范围式的语法需要使用专门的属性编辑器: `RangeArrayPropertyEditor`, 可以在 `ApplicationContext` 中配置。当然, 这个 `bean` 在批处理命名空间中的 `ApplicationContext` 里已经自动声明了。

单文件中含有多种类型数据的处理

前面所有的文件读取示例, 为简单起见都做了一个关键性假设: 在同一个文件中的所有记录都具有相同的格式。但情况有时候并非如此。其实在一个文件包含不同的格式的记录是很常见的, 需要使用不同的拆分方式, 映射到不同的对象中。下面是一个文件中的片段, 仅作演示:

```
1. USER;Smith;Peter;;T;20014539;F
2. LINEA;1044391041ABC037.49G201XX1383.12H
3. LINEB;2134776319DEF422.99M005LI
```

这个文件中有三种类型的记录, “USER”, “LINEA”, 以及 “LINEB”。一行 “USER” 对应一个 `User` 对象。“LINEA” 和 “LINEB” 对应的都是 `Line` 对象, 只是 “LINEA” 包含的信息比 “LINEB” 要多。

`ItemReader` 分别读取每一行, 当然我们必须指定不同的 `LineTokenizer` 和 `FieldSetMapper` 以便 `ItemWriter` 能获得到正确的 `item`。 `PatternMatchingCompositeLineMapper` 就是专门拿来干这个事的, 可以通过模式映射到对应的 `LineTokenizer` 和 `FieldSetMapper` :

```
1. <bean id="orderFileLineMapper"
2.     class="org.spr...PatternMatchingCompositeLineMapper">
3.     <property name="tokenizers">
```

```

4.     <map>
5.         <entry key="USER*" value-ref="userTokenizer" />
6.         <entry key="LINEA*" value-ref="lineATokenizer" />
7.         <entry key="LINEB*" value-ref="lineBTokenizer" />
8.     </map>
9. </property>
10. <property name="fieldSetMappers">
11.     <map>
12.         <entry key="USER*" value-ref="userFieldSetMapper" />
13.         <entry key="LINE*" value-ref="lineFieldSetMapper" />
14.     </map>
15. </property>
16. </bean>

```

在这个示例中，“LINEA”和“LINEB”使用独立的 LineTokenizer，但使用同一个 FieldSetMapper。

PatternMatchingCompositeLineMapper 使用 `PatternMatcher` 的 `match` 方法来为每一行选择正确的代理(delegate)。PatternMatcher 支持两个有特殊的意义通配符(wildcard)：问号(“?”，question mark)将匹配 1 个字符(注意不是0-1次)，而星号(“*”，asterisk)将匹配 0 到多个 字符。

请注意,在上面的配置中,所有以星号结尾的 pattern，使他们变成了行的有效前缀。

PatternMatcher 总是匹配最具体的可能模式，而不是按配置的顺序从上往下来。所以如果“LINE*”和“LINEA*”都配置为 pattern，那么“LINEA”将会匹配到“LINEA*”，而“LINEB”将匹配到“LINE*”。此外,单个星号(“*”)可以作为默认匹配所有行的模式，如果该行不匹配其他任何模式的话。

```
1. <entry key="*" value-ref="defaultLineTokenizer" />
```

还有一个 PatternMatchingCompositeLineTokenizer 可用来单独解析。

在平面文件中，也常常有单条记录跨越多行的情况。要处理这种情况，就需要一种更复杂的策略。这种模式的示例可以参考 第 11.5 节“跨域多行的记录”。

Flat File 的异常处理

在解析一行时，可能有很多情况会导致异常被抛出。很多平面文件不是很完整，或者里面的某些记录格式不正确。许多用户会选择忽略这些错误的行，只将这个问题记录到日志，比如原始行,行号。稍后可以人工审查这些日志，也可以由另一个批处理作业来检查。出于这个原因, Spring Batch提供了一系列的异常类：`FlatFileParseException`，和 `FlatFileFormatException`。

`FlatFileParseException` 是由 `FlatFileItemReader` 在读取文件时解析错误而抛出的。

`FlatFileFormatException` 是由实现了 `LineTokenizer` 接口的类抛出的，表明在拆分字段时发生了一个更具体的错误。

IncorrectTokenCountException

`DelimitedLineTokenizer` 和 `FixedLengthLineTokenizer` 都可以指定列名(column name), 用来创建一个 `FieldSet`。但如果 column name 的数量和 拆分时找到的列数目, 则不会创建 `FieldSet`, 只会抛出 `IncorrectTokenCountException` 异常, 里面包含了 字段的实际数量, 还有预期的数量:

```
1. tokenizer.setNames(new String[] {"A", "B", "C", "D"});
2.
3. try {
4.     tokenizer.tokenize("a,b,c");
5. }
6. catch(IncorrectTokenCountException e){
7.     assertEquals(4, e.getExpectedCount());
8.     assertEquals(3, e.getActualCount());
9. }
```

因为 tokenizer 配置了4列的名称, 但在这个文件中只找到 3 个字段, 所以会抛出 `IncorrectTokenCountException` 异常。

IncorrectLineLengthException

固定长度格式的文件在解析时有额外的要求, 因为每一列都必须严格遵守其预定义的宽度。如果一行的总长度不等于所有字段宽度之和, 就会抛出一个异常:

```
1. tokenizer.setColumns(new Range[] { new Range(1, 5),
2.                                     new Range(6, 10),
3.                                     new Range(11, 15) });
4. try {
5.     tokenizer.tokenize("12345");
6.     fail("Expected IncorrectLineLengthException");
7. }
8. catch (IncorrectLineLengthException ex) {
9.     assertEquals(15, ex.getExpectedLength());
10.    assertEquals(5, ex.getActualLength());
11. }
```

上面配置的范围是: `1-5`, `6-10`, 以及 `11-15`, 因此预期的总长度是15。但在这里传入的行的长度是 `5`, 所以会导致 `IncorrectLineLengthException` 异常。之所以直接抛出异常, 而不是先去映射第一个字段的原因是为了更早发现处理失败, 而不再调用 `FieldSetMapper` 来读取第2列。但是呢, 有些情况下, 行的长度并不总是固定的。出于这个原因, 可以通过设置 'strict' 属性的值, 不验证行的宽度:

```
1. tokenizer.setColumns(new Range[] { new Range(1, 5), new Range(6, 10) });
2. tokenizer.setStrict(false);
```

```
3. FieldSet tokens = tokenizer.tokenize("12345");
4. assertEquals("12345", tokens.readString(0));
5. assertEquals("", tokens.readString(1));
```

上面示例和前一个几乎完全相同，只是调用了 `tokenizer.setStrict(false)`。这个设置告诉 `tokenizer` 在对一行进行解析(tokenizing)时不要去管(enforce)行的长度。然后就正确地创建了一个 `FieldSet`并返回。当然,剩下的值就只会包含空的token值。

FlatFileItemWriter

- [6.6.3 FlatFileItemWriter](#)

6.6.3 FlatFileItemWriter

将数据写入到纯文本文件也必须解决和读取文件时一样的问题。在事务中,一个 step 必须通过分隔符或采用固定长度的格式将数据写出去。

LineAggregator

与 `LineTokenizer` 接口的处理方式类似,写入文件时也需要有某种方式将一条记录的多个字段组织拼接成单个 String,然后再将string写入文件。Spring Batch 对应的接口是

`LineAggregator` :

```
1. public interface LineAggregator<T> {
2.
3.     public String aggregate(T item);
4.
5. }
```

接口 `LineAggregator` 与 `LineTokenizer` 相互对应。`LineTokenizer` 接收 String,处理后返回一个 `FieldSet` 对象,而 `LineAggregator` 则是接收一条记录,返回对应的 String。

PassThroughLineAggregator

`LineAggregator` 接口最基础的实现类是 `PassThroughLineAggregator`,这个简单实现仅仅是将接收到的对象调用 `toString()` 方法的值返回:

```
1. public class PassThroughLineAggregator<T> implements LineAggregator<T> {
2.
3.     public String aggregate(T item) {
4.         return item.toString();
5.     }
6. }
```

上面的实现对于需要直接转换为string的时候是很管用的,但是 `FlatFileItemWriter` 的一些优势也是很有必要的,比如 事务,以及 支持重启特性等。

简单的文件写入示例

既然已经有了 `LineAggregator` 接口以及其最基础的实现, `PassThroughLineAggregator`,那就可以解释基础的写出流程了:

1. 将要写出的对象传递给 `LineAggregator` 以获取一个字符串(String).
2. 将返回的 `String` 写入配置指定的文件中.

下面是 `FlatFileItemWriter` 中对应的代码:

```
1. public void write(T item) throws Exception {
2.     write(lineAggregator.aggregate(item) + LINE_SEPARATOR);
3. }
```

简单的配置如下所示:

```
1. <bean id="itemWriter" class="org.spr...FlatFileItemWriter">
2.     <property name="resource" value="file:target/test-outputs/output.txt" />
3.     <property name="lineAggregator">
4.         <bean class="org.spr...PassThroughLineAggregator"/>
5.     </property>
6. </bean>
```

属性提取器 `FieldExtractor`

上面的示例可以应对最基本的文件写入情景。但使用 `FlatFileItemWriter` 时可能更多地是需要将某个领域对象写到文件,因此必须转换到单行之中。在读取文件时,有以下步骤:

1. 从文件中读取一行.
2. 将这一行字符串传递给 `LineTokenizer#tokenize()` 方法,以获取 `FieldSet` 对象
3. 将分词器返回的 `FieldSet` 传给一个 `FieldSetMapper` 映射器,然后将 `ItemReader#read()` 方法得到的结果 return。

文件的写入也很类似,但步骤正好相反:

1. 将要写入的对象传递给 `writer`
2. 将领域对象的属性域转换为数组
3. 将结果数组合并(aggregate)为一行字符串

因为框架没办法知道需要将领域对象的哪些字段写入到文件中,所以就需要有一个 `FieldExtractor` 来将对象转换为数组:

```
1. public interface FieldExtractor<T> {
2.
3.     Object[] extract(T item);
4.
5. }
```

`FieldExtractor` 的实现类应该根据传入对象的属性创建一个数组,稍后使用分隔符将各个元素写

入文件，或者作为 field-width line 的一部分。

PassThroughFieldExtractor

在很多时候需要将一个集合(如 array、Collection、FieldSet等)写出到文件。从集合中“提取”一个数组那真的是非常简单：直接进行简单转换即可。因此在这种场合 PassThroughFieldExtractor 就派上用场了。应该注意，如果传入的对象不是集合类型的，那么 PassThroughFieldExtractor 将返回一个数组，其中只包含提取的单个对象。

BeanWrapperFieldExtractor

与文件读取一节中所描述的 `BeanWrapperFieldSetMapper` 一样，通常使用配置来指定如何将领域对象转换为一个对象数组是比较好的办法，而不用自己写个方法来进行转换。

`BeanWrapperFieldExtractor` 就提供了这类功能：

```

1. BeanWrapperFieldExtractor<Name> extractor = new BeanWrapperFieldExtractor<Name>();
2. extractor.setNames(new String[] { "first", "last", "born" });
3.
4. String first = "Alan";
5. String last = "Turing";
6. int born = 1912;
7.
8. Name n = new Name(first, last, born);
9. Object[] values = extractor.extract(n);
10.
11. assertEquals(first, values[0]);
12. assertEquals(last, values[1]);
13. assertEquals(born, values[2]);

```

这个 extractor 实现只有一个必需的属性，就是 `names`，里面用来存放要映射字段的名称。就像 `BeanWrapperFieldSetMapper` 需要字段名称来将 FieldSet 中的 field 映射到对象的 setter 方法一样，`BeanWrapperFieldExtractor` 需要 names 映射 getter 方法来创建一个对象数组。值得注意的是，names的顺序决定了field在数组中的顺序。

分隔符文件(Delimited File)写入示例

最基础的平面文件格式是将所有字段用分隔符(delimiter)来进行分隔(separated)。这可以通过 `DelimitedLineAggregator` 来完成。下面的例子把一个表示客户信用额度的领域对象写出：

```

1. public class CustomerCredit {
2.
3.     private int id;
4.     private String name;
5.     private BigDecimal credit;
6.
7.     //getters and setters removed for clarity

```

```
8. }
```

因为使用到了领域对象,所以必须提供 `FieldExtractor` 接口的实现,当然也少不了要使用的分隔符:

```
1. <bean id="itemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
2.     <property name="resource" ref="outputResource" />
3.     <property name="lineAggregator">
4.         <bean class="org.springframework.batch.item.file.separator.DelimitedLineAggregator">
5.             <property name="delimiter" value="," />
6.             <property name="fieldExtractor">
7.                 <bean class="org.springframework.batch.item.file.separator.BeanWrapperFieldExtractor">
8.                     <property name="names" value="name,credit" />
9.                 </bean>
10.            </property>
11.        </bean>
12.    </property>
13. </bean>
```

在这种情况下,本章前面提到过的 `BeanWrapperFieldExtractor` 被用来将 `CustomerCredit` 中的 `name` 和 `credit` 字段转换为一个对象数组,然后在各个字段之间用逗号分隔写入文件。

固定宽度的(Fixed Width)文件写入示例

平面文件的格式并不是只有采用分隔符这种类型。许多人喜欢对每个字段设置一定的宽度,这样就能区分各个字段了,这种做法通常被称为“固定宽度, fixed width”。Spring Batch 通过 `FormatterLineAggregator` 支持这种文件的写入。使用上面描述的 `CustomerCredit` 领域对象,则可以对它进行如下配置:

```
1. <bean id="itemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
2.     <property name="resource" ref="outputResource" />
3.     <property name="lineAggregator">
4.         <bean class="org.springframework.batch.item.file.separator.FormatterLineAggregator">
5.             <property name="fieldExtractor">
6.                 <bean class="org.springframework.batch.item.file.separator.BeanWrapperFieldExtractor">
7.                     <property name="names" value="name,credit" />
8.                 </bean>
9.             </property>
10.            <property name="format" value="%-9s%-2.0f" />
11.        </bean>
12.    </property>
13. </bean>
```

上面的示例大部分看起来是一样的,只有 `format` 属性的值不同:

```
1. <property name="format" value="%-9s%-2.0f" />
```

底层实现采用 Java 5 提供的 `Formatter` 。Java的 `Formatter` (格式化) 基于C语言的 `printf` 函数功能。关于如何配置 `formatter` 请参考 `Formatter` 的javadoc.

处理文件创建(Handling File Creation)

`FlatFileItemReader` 与文件资源的关系很简单。在初始化 `reader` 时,如果文件存在则打开,如果文件不存在那就抛出一个异常(exception)。

但是文件的写入就没那么简单了。乍一看可能会觉得跟 `FlatFileItemWriter` 一样简单直接粗暴:如果文件存在则抛出异常, 如果不存在则创建文件并开始写入。

但是, 作业的重启有可能会有BUG。 在正常的重启情景中, 约定与前面所想的恰恰相反: 如果文件存在, 则从已知的最后一个正确位置开始写入, 如果不存在, 则抛出异常。

如果此作业(Job)的文件名每次都一样的那怎么办? 这时候可能需要删除已存在的文件(重启则不删除)。 因为有这些可能性, `FlatFileItemWriter` 有一个属性 `shouldDeleteIfExists` 。将这个属性设置为 `true` , 打开 `writer` 时会将已有的同名文件删除。

XML条目读写器

- 6.7 XML Item Readers and Writers

6.7 XML Item Readers and Writers

Spring Batch为读取XML映射为Java对象以及将Java对象写为XML记录提供了事务基础。

[注意]XML流的限制

StAX API 被用在其他XML解析引擎不适合批处理请求 I/O 时的情况(DOM方式把整个输入文件加载到内存中, 而SAX方式在解析过程中需要用户提供回调)。

让我们仔细看看在Spring Batch中 XML输入和输出是如何运行的。首先,有一些不同于文件读取和写入的概念,但在Spring Batch XML处理中是很常见的。在处理XML时,并不像读取文本文件(FieldSets)时采取分隔符标记逐行读取的方式,而是假定XML资源是对应于单条记录的文档片段(' fragments ')的集合:

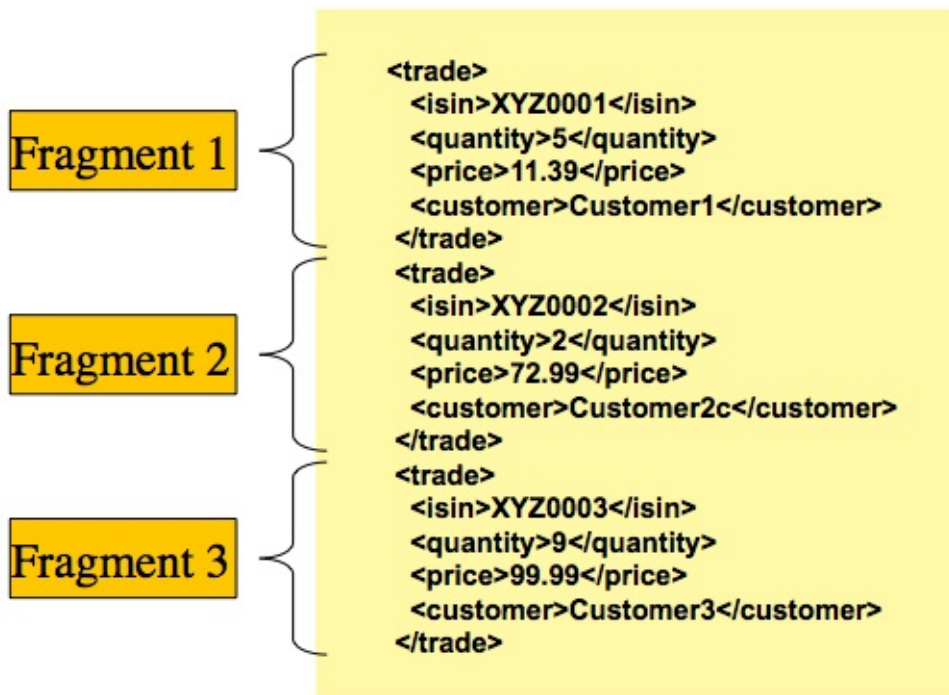


图 3.1: XML 输入文件

“ trade ”标签在上面的场景中是根元素“root element”。在 ‘ <trade> ’和‘ </trade> ’之间的一切都被认为是一个 文档片段 ‘ fragment ’。Spring Batch使用 Object/XML映射(OXM)将 fragments 绑定到对象。但 Spring Batch 并不依赖某个特定的XML绑定技术。Spring OXM 委托是最典型的用途, 其为常见的OXM技术提供了统一的抽象。Spring OXM 依赖是可选的, 如有

必要，你也可以自己实现 Spring Batch 的某些接口。 OXM支持的技术间的关系如下图所示：

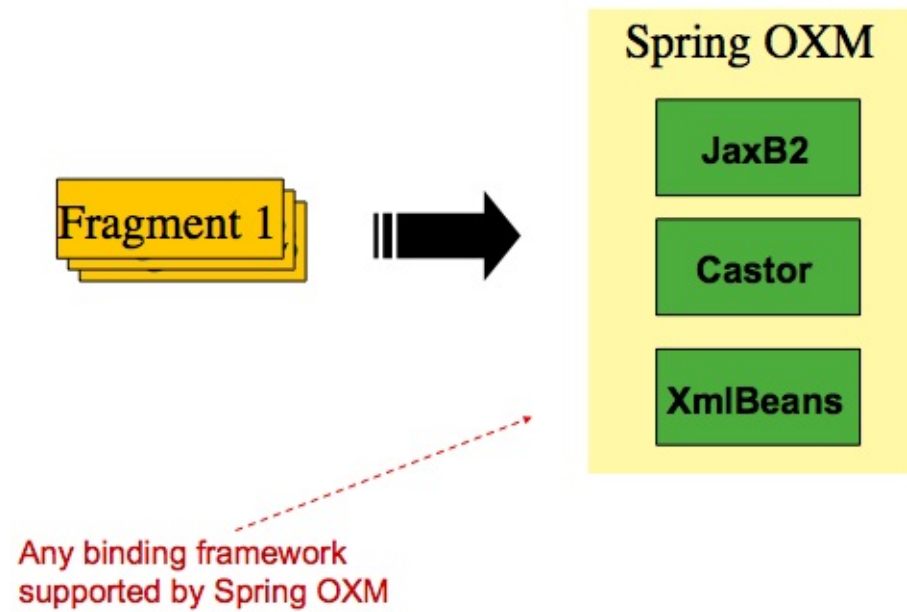


图 3.2: OXM Binding

上面介绍了OXM以及如何使用XML片段来表示记录，接着让我们仔细了解下 readers 和 writers 。

StaxEventItemReader

- [6.7.1 StaxEventItemReader](#)
- #

6.7.1 StaxEventItemReader

StaxEventItemReader 提供了从XML输入流进行记录处理的典型设置。 首先,我们来看一下 **StaxEventItemReader**能处理的一组XML记录。

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <records>
3.   <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
4.     <isin>XYZ0001</isin>
5.     <quantity>5</quantity>
6.     <price>11.39</price>
7.     <customer>Customer1</customer>
8.   </trade>
9.   <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
10.    <isin>XYZ0002</isin>
11.    <quantity>2</quantity>
12.    <price>72.99</price>
13.    <customer>Customer2c</customer>
14.  </trade>
15.  <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
16.    <isin>XYZ0003</isin>
17.    <quantity>9</quantity>
18.    <price>99.99</price>
19.    <customer>Customer3</customer>
20.  </trade>
21. </records>

```

能被处理的XML记录需要满足下列条件：

- **Root Element Name** 片段根元素的名称就是要映射的对象。上面的示例代表的是 `trade` 的值。
- **Resource** Spring Resource 代表了需要读取的文件。
- **Unmarshaller** Spring OXM提供的Unmarshalling 用于将 XML片段映射为对象。

#

```
1. <bean id="itemReader" class="org.springframework.batch.item.xml.StaxEventItemReader">
```

```

2. <property name="fragmentRootElementName" value="trade" />
3. <property name="resource" value="data/iosample/input/input.xml" />
4. <property name="unmarshaller" ref="tradeMarshaller" />
5. </bean>

```

请注意, 在上面的例子中, 我们选用一个 XStreamMarshaller, 里面接受一个id为 `aliases` 的 map, 将首个entry的 `key` 值作为文档片段的name(即根元素), 将 `value` 作为绑定的对象类型。类似于FieldSet, 后面的其他元素映射为对象内部的字段名/值对。在配置文件中, 我们可以像下面这样使用Spring配置工具来描述所需的alias:

```

1. <bean id="tradeMarshaller"
2.     class="org.springframework.xml.xstream.XStreamMarshaller">
3.     <property name="aliases">
4.         <util:map id="aliases">
5.             <entry key="trade"
6.                 value="org.springframework.batch.sample.domain.Trade" />
7.             <entry key="price" value="java.math.BigDecimal" />
8.             <entry key="name" value="java.lang.String" />
9.         </util:map>
10.    </property>
11. </bean>

```

当 reader 读取到XML资源的一个新片段时(匹配默认的标签名称)。reader 根据这个片段构建一个独立的XML(或至少看起来是这样), 并将 document 传给反序列化器(通常是一个Spring OXM Unmarshaller 的包装类)将XML映射为一个Java对象。

总之, 这个过程类似于下面的Java代码, 其中配置了 Spring的注入功能:

```

1. StaxEventItemReader xmlStaxEventItemReader = new StaxEventItemReader()
2. Resource resource = new ByteArrayResource(xmlResource.getBytes())
3.
4. Map aliases = new HashMap();
5. aliases.put("trade", "org.springframework.batch.sample.domain.Trade");
6. aliases.put("price", "java.math.BigDecimal");
7. aliases.put("customer", "java.lang.String");
8. Marshaller marshaller = new XStreamMarshaller();
9. marshaller.setAliases(aliases);
10. xmlStaxEventItemReader.setUnmarshaller(marshaller);
11. xmlStaxEventItemReader.setResource(resource);
12. xmlStaxEventItemReader.setFragmentRootElementName("trade");
13. xmlStaxEventItemReader.open(new ExecutionContext());
14.
15. boolean hasNext = true
16.
17. CustomerCredit credit = null;
18.

```

```
19. while (hasNext) {
20.     credit = xmlStaxEventItemReader.read();
21.     if (credit == null) {
22.         hasNext = false;
23.     }
24.     else {
25.         System.out.println(credit);
26.     }
27. }
```


StaxEventItemWriter

- 6.7.2 StaxEventItemWriter

6.7.2 StaxEventItemWriter

输出与输入相对应。StaxEventItemWriter 需要 1个 Resource，1个 marshaller 以及 1个 rootTagName。Java对象传递给marshaller(通常是标准的Spring OXM marshaller)，marshaller 使用自定义的事件writer写入Resource，并过滤由OXM工具为每条 fragment 产生的 StartDocument 和 EndDocument事件。我们用 MarshallingEventWriterSerializer 示例来显示这一点。Spring配置如下所示：

```

1. <bean id="itemWriter" class="org.springframework.batch.item.xml.StaxEventItemWriter">
2.     <property name="resource" ref="outputResource" />
3.     <property name="marshaller" ref="customerCreditMarshaller" />
4.     <property name="rootTagName" value="customers" />
5.     <property name="overwriteOutput" value="true" />
6. </bean>

```

上面配置了3个必需的属性，以及1个可选属性 overwriteOutput = true，（本章前面提到过）用来指定一个已存在的文件是否可以被覆盖。应该注意的是，writer 使用的 marshaller 和前面讲的 reading 示例中是完全相同的：

```

1. <bean id="customerCreditMarshaller"
2.     class="org.springframework.oxm.xstream.XStreamMarshaller">
3.     <property name="aliases">
4.         <util:map id="aliases">
5.             <entry key="customer"
6.                 value="org.springframework.batch.sample.domain.CustomerCredit" />
7.             <entry key="credit" value="java.math.BigDecimal" />
8.             <entry key="name" value="java.lang.String" />
9.         </util:map>
10.    </property>
11. </bean>

```

我们用一段Java代码来总结所讨论的知识点，并演示如何通过代码手动设置所需的属性：

```

1. StaxEventItemWriter staxItemWriter = new StaxEventItemWriter()
2. FileSystemResource resource = new FileSystemResource("data/outputFile.xml")
3.
4. Map aliases = new HashMap();
5. aliases.put("customer", "org.springframework.batch.sample.domain.CustomerCredit");

```

```
6. aliases.put("credit", "java.math.BigDecimal");
7. aliases.put("name", "java.lang.String");
8. Marshaller marshaller = new XStreamMarshaller();
9. marshaller.setAliases(aliases);
10.
11. staxItemWriter.setResource(resource);
12. staxItemWriter.setMarshaller(marshaller);
13. staxItemWriter.setRootTagName("trades");
14. staxItemWriter.setOverwriteOutput(true);
15.
16. ExecutionContext executionContext = new ExecutionContext();
17. staxItemWriter.open(executionContext);
18. CustomerCredit Credit = new CustomerCredit();
19. trade.setPrice(11.39);
20. credit.setName("Customer1");
21. staxItemWriter.write(trade);
```

多个输入文件

- 6.8 多个数据输入文件

6.8 多个数据输入文件

在单个 **Step** 中处理多个输入文件是很常见的需求。如果这些文件都有相同的格式，则可以使用 **MultiResourceItemReader** 来进行处理(支持 XML/或 纯文本文件)。假如某个目录下有如下3个文件：

```
1. file-1.txt
2. file-2.txt
3. ignored.txt
```

`file-1.txt` 和 `file-2.txt` 具有相同的格式，根据业务需求需要一起处理。可以通过 **MultiResourceItemReader** 使用 通配符 的形式来读取这两个文件：

```
1. <bean id="multiResourceReader" class="org.springframework.MultiResourceItemReader">
2.     <property name="resources" value="classpath:data/input/file-*.txt" />
3.     <property name="delegate" ref="flatFileItemReader" />
4. </bean>
```

`delegate` 引用的是一个简单的 **FlatFileItemReader**。上面的配置将会从两个输入文件中读取数据,处理回滚以及重启场景。应该注意的是,所有 **ItemReader** 在添加额外的输入文件后(如本示例),如果重新启动则可能会导致某些潜在的问题。官方建议是每个批作业处理独立的目录,一直到成功完成为止。

数据库Database

- [6.9 数据库\(Database\)](#)

6.9 数据库(Database)

和大部分企业应用一样,数据库也是批处理系统存储数据的核心机制。但批处理与其他应用的不同之处在于,批处理系统一般都运行于大规模数据集基础上。如果一条SQL语句返回100万行,则结果集可能全部存放在内存中直到所有行全部读完。Spring Batch提供了两种类型的解决方案来处理这个问题:游标(Cursor)和可分页的数据库ItemReaders。

基于游标的ItemReaders

- [6.9.1 基于Cursor的ItemReaders](#)
 - [译注](#)
- [附加属性" level="4">附加属性](#)
- [需要整理](#)

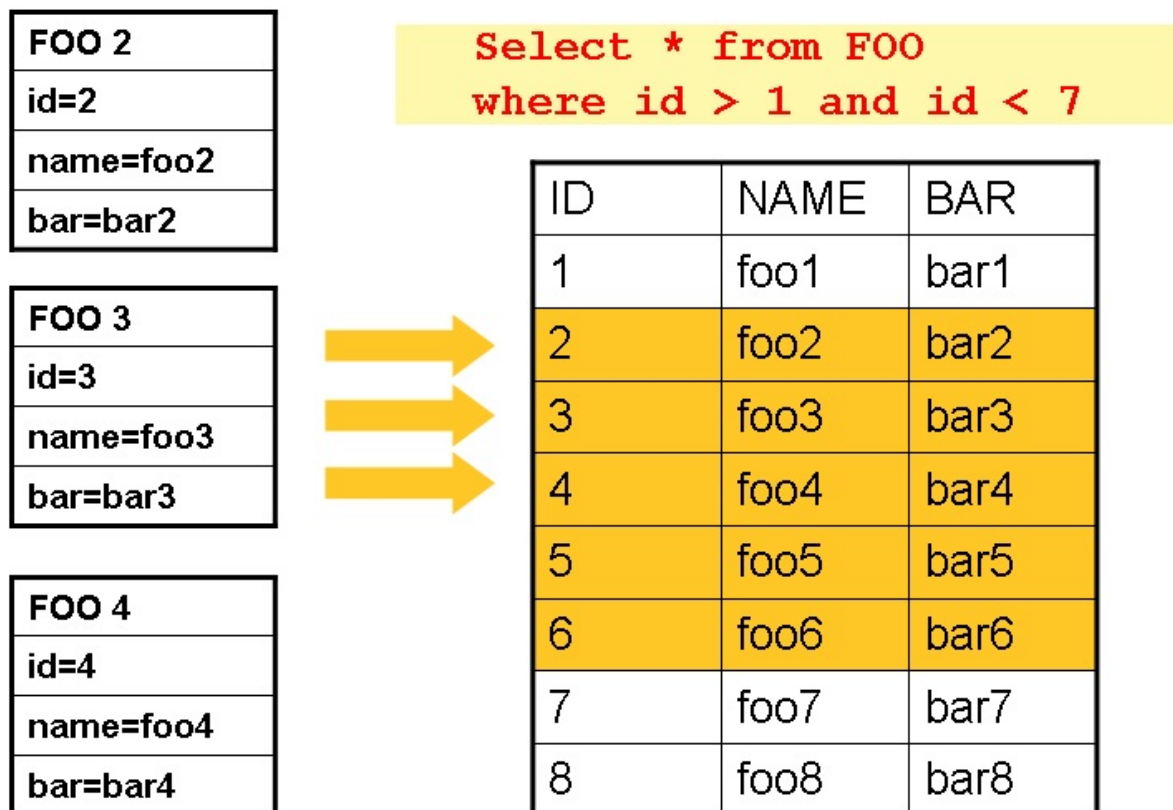
6.9.1 基于Cursor的ItemReaders

使用游标(cursor)是大多数批处理开发人员默认采用的方法, 因为它是处理有关系的数据“流”在数据库级别的解决方案。Java 的 `ResultSet` 类其本质就是用面向对象的游标处理机制。`ResultSet` 维护着一个指向当前数据行的 cursor。调用 `ResultSet` 的 `next` 方法则将游标移到下一行。

Spring Batch 基于 cursor 的 `ItemReaders` 在初始化时打开游标, 每次调用 `read` 时则将游标向前移动一行, 返回一个可用于进行处理的映射对象。最好将会调用 `close` 方法, 以确保所有资源都被释放。

Spring 的 `JdbcTemplate` 的解决办法, 是通过回调模式将 `ResultSet` 中所有行映射之后, 在返回调用方法前关闭结果集来处理的。

但是, 在批处理的时候就不一样了, 必须得等 `step` 执行完成才能调用`close`。下图描绘了基于游标的ItemReader是如何处理的, 使用的SQL语句非常简单, 而且都是类似的实现方式:



这个例子演示了基本的处理模式。数据库中有一个 “FOO” 表,它有三个字段: ID, NAME, 以及 BAR, select 查询所有ID大于1但小于7的行。这样的话光标起始于 ID 为 2的行(第1行)。这一行的结果会被映射为一个Foo对象。再次调用read()则将光标移动到下一行,也就是ID为3的Foo。在所有行读取完毕之后这些结果将会被写出去,然后这些对象就会被垃圾回收(假设没有其他引用指向他们)。

译注

Foo、Bar 都是英文中的任意代词,没有什么具体意义,就如我们说的 张三,李四 一样

JdbcCursorItemReader

JdbcCursorItemReader 是基于 cursor 的Jdbc实现。它直接使用ResultSet, 需要从数据库连接池中获取连接来执行SQL语句。我们的示例使用下面的数据库表:

```

1. CREATE TABLE CUSTOMER (
2.     ID BIGINT IDENTITY PRIMARY KEY,
3.     NAME VARCHAR(45),
4.     CREDIT FLOAT
5. );

```

我们一般使用领域对象来对应到每一行, 所以用 RowMapper 接口的实现来映射

CustomerCredit 对象:

```

1. public class CustomerCreditRowMapper implements RowMapper {
2.
3.     public static final String ID_COLUMN = "id";
4.     public static final String NAME_COLUMN = "name";
5.     public static final String CREDIT_COLUMN = "credit";
6.
7.     public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
8.         CustomerCredit customerCredit = new CustomerCredit();
9.
10.        customerCredit.setId(rs.getInt(ID_COLUMN));
11.        customerCredit.setName(rs.getString(NAME_COLUMN));
12.        customerCredit.setCredit(rs.getBigDecimal(CREDIT_COLUMN));
13.
14.        return customerCredit;
15.    }
16. }

```

一般来说Spring的用户对 `JdbcTemplate` 都不陌生，而 `JdbcCursorItemReader` 使用其作为关键API接口，我们一起来学习如何通过 `JdbcTemplate` 读取这一数据，看看它与 `ItemReader` 有何区别。为了演示方便，我们假设CUSTOMER表有1000行数据。第一个例子将使用

`JdbcTemplate`：

```

1. //For simplicity sake, assume a dataSource has already been obtained
2. JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
3. List customerCredits = jdbcTemplate.query("SELECT ID, NAME, CREDIT from CUSTOMER",
4.                                         new CustomerCreditRowMapper());

```

当执行完上面的代码，`customerCredits` 这个 List 中将包含 1000 个 `CustomerCredit` 对象。在 `query` 方法中，先从 `DataSource` 获取一个连接，然后用来执行给定的SQL，获取结果后对 `ResultSet` 中的每一行调用一次 `mapRow` 方法。让我们来对比一下

`JdbcCursorItemReader` 的实现：

```

1. JdbcCursorItemReader itemReader = new JdbcCursorItemReader();
2. itemReader.setDataSource(dataSource);
3. itemReader.setSql("SELECT ID, NAME, CREDIT from CUSTOMER");
4. itemReader.setRowMapper(new CustomerCreditRowMapper());
5. int counter = 0;
6. ExecutionContext executionContext = new ExecutionContext();
7. itemReader.open(executionContext);
8. Object customerCredit = new Object();
9. while(customerCredit != null){
10.    customerCredit = itemReader.read();
11.    counter++;
12. }
13. itemReader.close(executionContext);

```

运行这段代码后 counter 的值将变成 1000。如果上面的代码将返回的 customerCredit 放入 List, 则结果将和使用 `JdbcTemplate` 的例子完全一致。但是呢, 使用 `ItemReader` 的强大优势在于, 它允许数据项变成“流式(streamed)”。调用一次 `read` 方法, 通过 `ItemWriter` 写出数据对象, 然后再通过 `read` 获取下一项。这使得 item 读取和写出可以进行“分块(chunks)”, 并且周期性地提交, 这才是高性能批处理的本质。此外, 它可以很容易地通过配置注入到某个 Spring Batch `Step` 中:

```

1. <bean id="itemReader" class="org.springframework.jdbc.cursor.ItemReader">
2.     <property name="dataSource" ref="dataSource"/>
3.     <property name="sql" value="select ID, NAME, CREDIT from CUSTOMER"/>
4.     <property name="rowMapper">
5.         <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>
6.     </property>
7. </bean>

```

附加属性" `class="reference-link">附加属性`

因为在Java中有很多种不同的方式来打开游标, 所以 `JdbcCursorItemReader` 有许多可以设置的属性:

需要整理

Table 6.2. `JdbcCursorItemReader` 的属性(Properties)

ignoreWarnings	决定 SQL警告(SQLWarnings)是否被日志记录, 还是导致异常 - 默认值为 true
fetchSize	给 Jdbc driver 一个提示, 当 ItemReader 对象需要从 ResultSet 中获取更多记录时, 每次应该取多少行数据. 默认没有给定 hint 值.
maxRows	设置底层的 ResultSet 最多可以持有多少行记录
queryTimeout	设置 driver 在执行 Statement 对象时应该在给定的时间(单位: 秒)内完成. 如果超过这个时间限制, 就抛出一个 DataAccessException 异常. (详细信息请参考/咨询具体数据库驱动的相关文档).
verifyCursorPosition	因为 ItemReader 持有的同一个 ResultSet 会被传递给 RowMapper, 所以用户有可能会自己调用 ResultSet.next(), 这就有可能会影响到 reader 内部的计数状态. 将这个值设置为 true 时, 如果在调用 RowMapper 前后游标位置(cursor position)不一致, 就会抛出一个异常.
saveState	明确指定 reader 的状态是否应该保存在 ItemStream#update (ExecutionContext) 提供的 ExecutionContext 中, 默认值为 true.
driverSupportsAbsolute	默认值为 false. 指明 Jdbc 驱动是否支持在 ResultSet 中设置绝对行(absolute row). 官方建议, 对于支持 ResultSet.absolute() 的 Jdbc drivers, 应该设置为 true, 一般能提高效率和性能, 特别是在某个 step 处理很大的数据集失败时.
setUseSharedExtendedConnection	默认值为 false. 指明此 cursor 使用的数据库连接是否和其他处理过程共享连接, 以便处于同一个事务中. 如果设置为 false, 也就是默认值, 那么游标会打开自己的数据库连接, 也就不会参与到 step 处理中的其他事务. 如果要将标志位设置为 true, 则必须将 DataSource 包装在一个 ExtendedConnectionDataSourceProxy 中, 以阻止每次提交之后关闭/释放连接. 如果此选项设置为 true, 则打开cursor的语句将会自动带上

'READ_ONLY' 和 'HOLD_CUSORS_OVER_COMMIT' 选项。这样就允许在 step 处理过程中保持 cursor 跨越多个事务。要使用这个特性,需要数据库服务器的支持,以及JDBC驱动符合 Jdbc 3.0 版本规范。

HibernateCursorItemReader

使用 Spring 的程序员需要作出一个重要的决策,即是否使用ORM解决方案,这决定了是否使用 **JdbcTemplate** 或 **HibernateTemplate**, Spring Batch开发者也面临同样的选择。**HibernateCursorItemReader** 是 Hibernate 的游标实现。其实在批处理中使用 Hibernate 那是相当有争议。这很大程度上是因为 Hibernate 最初就是设计了用来开发在线程序的。

但也不是说Hibernate就不能用来进行批处理。最简单的解决办法就是使用一个 **StatelessSession** (无状态会话),而不使用标准 **session**。这样就去掉了在批处理场景中 Hibernate 那些恼人的缓存、脏检查等等。

更多无状态会话与正常hibernate会话之间的差异,请参考你使用的 hibernate 版本对应的文档。**HibernateCursorItemReader** 允许您声明一个HQL语句,并传入 **SessionFactory**,然后每次调用 **read** 时就会返回一个对象,和 **JdbcCursorItemReader** 一样。下面的示例配置也使用 and JDBC reader 相同的数据库表:

```

1. HibernateCursorItemReader itemReader = new HibernateCursorItemReader();
2. itemReader.setQueryString("from CustomerCredit");
3. //For simplicity sake, assume sessionFactory already obtained.
4. itemReader.setSessionFactory(sessionFactory);
5. itemReader.setUseStatelessSession(true);
6. int counter = 0;
7. ExecutionContext executionContext = new ExecutionContext();
8. itemReader.open(executionContext);
9. Object customerCredit = new Object();
10. while(customerCredit != null){
11.     customerCredit = itemReader.read();
12.     counter++;
13. }
14. itemReader.close(executionContext);

```

这里配置的 **ItemReader** 将以完全相同的方式返回**CustomerCredit**对象,和 **JdbcCursorItemReader** 没有区别,如果 hibernate 映射文件正确的话。`useStatelessSession` 属性的默认值为 `true`,这里明确设置的目的是为了引起你的注意,我们可以通过他来进行切换。还值得注意的是可以通过 `setFetchSize` 设置底层 **cursor** 的 `fetchSize` 属性。与**JdbcCursorItemReader**一样,配置很简单:

```

1. <bean id="itemReader"
2.     class="org.springframework.batch.item.database.HibernateCursorItemReader">
3.     <property name="sessionFactory" ref="sessionFactory" />

```

```

4.     <property name="queryString" value="from CustomerCredit" />
5. </bean>

```

StoredProcedureItemReader

有时候使用存储过程来获取游标数据是很有必要的。 **StoredProcedureItemReader** 和 **JdbcCursorItemReader** 其实差不多，只是不再执行一个查询来获取游标，而是执行一个存储过程，由存储过程返回一个游标。存储过程有三种返回游标的方式：

1. 作为一个 `ResultSet` 返回(SQL Server, Sybase, DB2, Derby 以及 MySQL支持)
2. 作为一个 `out` 参数返回 `ref-cursor` (Oracle和PostgreSQL使用这种方式)
3. 作为存储函数(stored function)的返回值

下面是一个基本的配置示例，还是使用上面“客户信用”的例子：

```

1. <bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
2.     <property name="dataSource" ref="dataSource"/>
3.     <property name="procedureName" value="sp_customer_credit"/>
4.     <property name="rowMapper">
5.         <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>
6.     </property>
7. </bean>

```

这个例子依赖于存储过程提供一个 `ResultSet` 作为返回结果(方式1)。

如果存储过程返回一个`ref-cursor`(方式2),那么我们就需要提供返回的`ref-cursor`(`out` 参数)的位置。下面的示例中,第一个参数是返回的`ref-cursor`：

```

1. <bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
2.     <property name="dataSource" ref="dataSource"/>
3.     <property name="procedureName" value="sp_customer_credit"/>
4.     <property name="refCursorPosition" value="1"/>
5.     <property name="rowMapper">
6.         <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>
7.     </property>
8. </bean>

```

如果存储函数的返回值是一个游标(方式 3),则需要将 `function` 属性设置为 `true` , 默认为 `false` 。如下面所示：

```

1. <bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
2.     <property name="dataSource" ref="dataSource"/>
3.     <property name="procedureName" value="sp_customer_credit"/>
4.     <property name="function" value="true"/>
5.     <property name="rowMapper">

```

```

6.     <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>
7.     </property>
8. </bean>

```

在所有情况下,我们都需要定义 **RowMapper** 以及 **DataSource**, 还有存储过程的名字。

如果存储过程/函数需要传入参数, 那么必须声明并通过 `parameters` 属性来设置值。下面是一个关于 Oracle 的示例, 其中声明了三个参数。第一个是 `out` 参数, 用来返回 `ref-cursor`, 第二第三个参数是 `in` 型参数, 类型都是 **INTEGER** :

```

1. <bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
2.     <property name="dataSource" ref="dataSource"/>
3.     <property name="procedureName" value="spring.cursor_func"/>
4.     <property name="parameters">
5.         <list>
6.             <bean class="org.springframework.jdbc.core.SqlOutParameter">
7.                 <constructor-arg index="0" value="newid"/>
8.                 <constructor-arg index="1">
9.                     <util:constant static-field="oracle.jdbc.OracleTypes.CURSOR"/>
10.                </constructor-arg>
11.            </bean>
12.            <bean class="org.springframework.jdbc.core.SqlParameter">
13.                <constructor-arg index="0" value="amount"/>
14.                <constructor-arg index="1">
15.                    <util:constant static-field="java.sql.Types.INTEGER"/>
16.                </constructor-arg>
17.            </bean>
18.            <bean class="org.springframework.jdbc.core.SqlParameter">
19.                <constructor-arg index="0" value="custid"/>
20.                <constructor-arg index="1">
21.                    <util:constant static-field="java.sql.Types.INTEGER"/>
22.                </constructor-arg>
23.            </bean>
24.        </list>
25.    </property>
26.    <property name="refCursorPosition" value="1"/>
27.    <property name="rowMapper" ref="rowMapper"/>
28.    <property name="preparedStatementSetter" ref="parameterSetter"/>
29. </bean>

```

除了参数声明, 我们还需要指定一个 `PreparedStatementSetter` 实现来设置参数值。这和上面的 `JdbcCursorItemReader` 一样。查看全部附加属性请查看 [附加属性](#), `StoredProcedureItemReader` 的附加属性也一样。

ItemReaders分页

- 6.9.2 可分页的 `ItemReader`
 - `JdbcPagingItemReader`
 - `JpaPagingItemReader`
 - `IbatisPagingItemReader`

6.9.2 可分页的 `ItemReader`

另一种是使用数据库游标执行多次查询,每次查询只返回一部分结果。我们将这一部分称为一页(a page)。分页时每次查询必须指定想要这一页的起始行号和想要返回的行数。

`JdbcPagingItemReader`

分页 `ItemReader` 的一个实现是 `JdbcPagingItemReader`。`JdbcPagingItemReader` 需要一个 `PagingQueryProvider` 来负责提供获取每一页所需的查询SQL。由于每个数据库都有不同的分页策略,所以我们需要为各种数据库使用对应的 `PagingQueryProvider`。也有自动检测所使用数据库类型的 `SqlPagingQueryProviderFactoryBean`,会根据数据库类型选用适当的 `PagingQueryProvider` 实现。这简化了配置,同时也是推荐的最佳实践。

`SqlPagingQueryProviderFactoryBean` 需要指定一个 `select` 子句以及一个 `from` 子句(`clause`)。当然还可以选择提供 `where` 子句。这些子句加上所需的排序列 `sortKey` 被组合成为一个 SQL 语句(statement)。

在 `reader` 被打开以后,每次调用 `read` 方法则返回一个 `item`,和其他的 `ItemReader`一样。使用分页是因为可能需要额外的行。

下面是一个类似 'customer credit' 示例的例子,使用上面提到的基于 `cursor`的 `ItemReaders`:

```

1. <bean id="itemReader" class="org.spr...JdbcPagingItemReader">
2.     <property name="dataSource" ref="dataSource"/>
3.     <property name="queryProvider">
4.         <bean class="org.spr...SqlPagingQueryProviderFactoryBean">
5.             <property name="selectClause" value="select id, name, credit"/>
6.             <property name="fromClause" value="from customer"/>
7.             <property name="whereClause" value="where status=:status"/>
8.             <property name="sortKey" value="id"/>
9.         </bean>
10.    </property>
11.    <property name="parameterValues">
12.        <map>
13.            <entry key="status" value="NEW"/>

```

```

14.     </map>
15.   </property>
16.   <property name="pageSize" value="1000"/>
17.   <property name="rowMapper" ref="customerMapper"/>
18. </bean>

```

这里配置的ItemReader将返回CustomerCredit对象，必须指定使用的RowMapper。

‘`pageSize`’属性决定了每次数据库查询返回的实体数量。

‘`parameterValues`’属性可用来为查询指定参数映射map。如果在where子句中使用了命名参数，那么这些entry的key应该和命名参数一一对应。如果使用传统的‘?’占位符，则每个entry的key就应该是占位符的数字编号，和JDBC占位符一样索引都是从1开始。

JpaPagingItemReader

另一个分页ItemReader的实现是 `JpaPagingItemReader`。JPA没有 Hibernate 中 `StatelessSession` 之类的概念，所以必须使用JPA规范提供的其他功能。因为JPA支持分页，所以在使用JPA来处理分页时这是一种很自然的选择。读取每页后，实体将会分离而且持久化上下文将会被清除，以允许在页面处理完成后实体会被垃圾回收。

JpaPagingItemReader 允许您声明一个JPQL语句，并传入一个 **EntityManagerFactory**。然后就和其他的 ItemReader 一样，每次调用它的 `read` 方法都会返回一个 `item`。当需要更多实体，则内部就会自动发生分页。下面是一个示例配置，和上面的JDBC reader一样，都是 ‘customer credit’：

```

1. <bean id="itemReader" class="org.spr...JpaPagingItemReader">
2.   <property name="entityManagerFactory" ref="entityManagerFactory"/>
3.   <property name="queryString" value="select c from CustomerCredit c"/>
4.   <property name="pageSize" value="1000"/>
5. </bean>

```

这里配置的ItemReader和前面所说的 `JdbcPagingItemReader` 返回一样的 `CustomerCredit` 对象，假设 `Customer` 对象有正确的JPA注解或者ORM映射文件。‘`pageSize`’属性决定了每次查询时读取的实体数量。

IbatisPagingItemReader

[Note] 注意事项

这个 `reader` 在 `Spring Batch 3.0`中已经被废弃(*deprecated*)。

如果使用 IBATIS/MyBatis，则可以使用 `IbatisPagingItemReader`，顾名思义，也是一种实现分页的ItemReader。IBATIS不对分页提供直接支持，但通过提供一些标准变量就可以为IBATIS查询提供分页支持。

下面是和上面的示例同样功能的配置,使用IbatisPagingItemReader来读取CustomerCredits:

```

1. <bean id="itemReader" class="org.spr...IbatisPagingItemReader">
2.     <property name="sqlMapClient" ref="sqlMapClient"/>
3.     <property name="queryId" value="getPagedCustomerCredits"/>
4.     <property name="pageSize" value="1000"/>
5. </bean>

```

上述 **IbatisPagingItemReader** 配置引用了一个IBATIS查询,名为“getPagedCustomerCredits”。如果使用MySQL,那么查询XML应该类似于下面这样。

```

1. <select id="getPagedCustomerCredits" resultMap="customerCreditResult">
2.     select id, name, credit from customer order by id asc LIMIT #_skiprows#, #_pagesize#
3. </select>

```

`_skiprows` 和 `_pagesize` 变量都是 **IbatisPagingItemReader** 提供的,还有一个 `_page` 变量,需要时也可以使用。分页查询的语法根据数据库不同使用。下面是使用Oracle的一个例子(但我们需要使用CDATA来包装某些特殊符号,因为是放在XML文档中嘛):

```

1. <select id="getPagedCustomerCredits" resultMap="customerCreditResult">
2.     select * from (
3.         select * from (
4.             select t.id, t.name, t.credit, ROWNUM ROWNUM_ from customer t order by id
5.             ) where ROWNUM_ <![CDATA[ > ]]> ( #_page# * #_pagesize# )
6.         ) where ROWNUM <![CDATA[ <= ]]> #_pagesize#
7. </select>

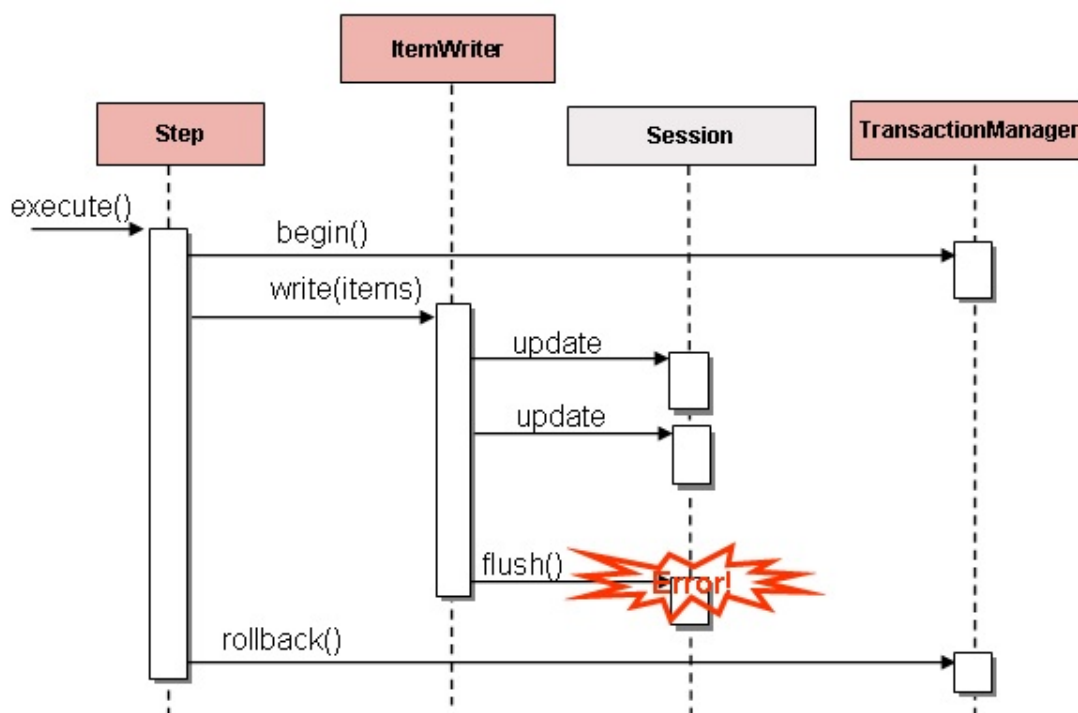
```


数据库ItemWriters

- 6.9.3 Database ItemWriters

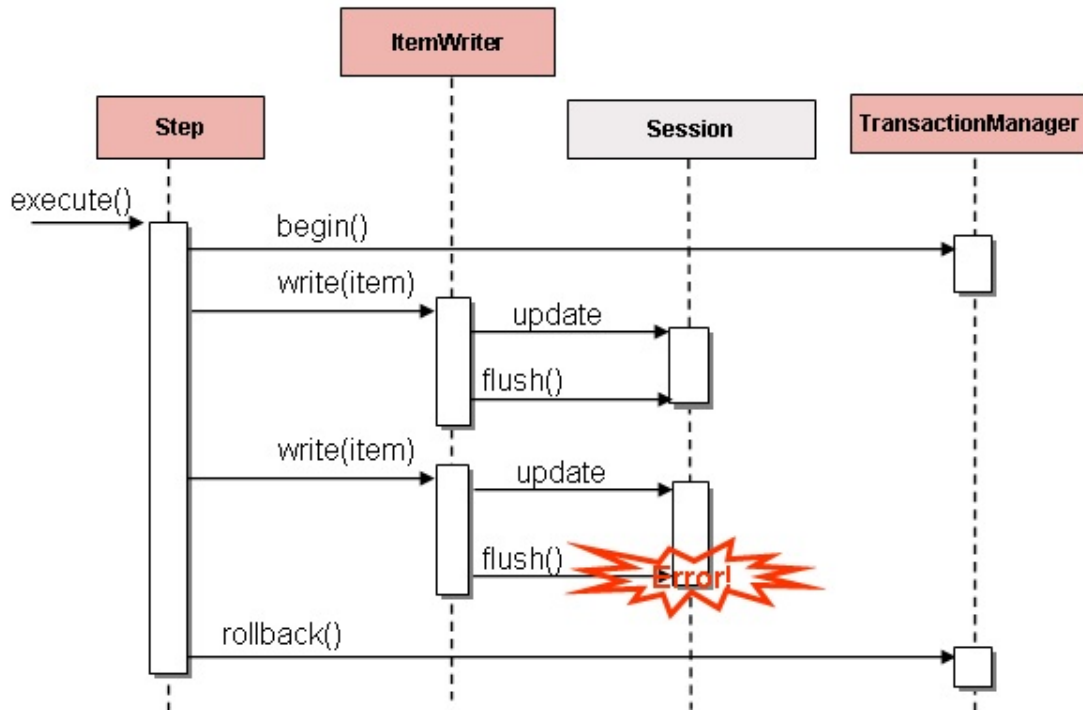
6.9.3 Database ItemWriters

虽然文本文件和XML都有自己特定的 ItemWriter，但数据库和他们并不一样。这是因为事务提供了所需的全部功能。对于文件来说 ItemWriters 是必要的，因为如果需要事务特性，他们必须充当这种角色，跟踪输出的 item，并在适当的时间 flushing/clearing。使用数据库时不需要这个功能，因为写已经包含在事务之中。用户可以自己创建实现ItemWriter接口的 DAO，或使用一个处理常见问题的自定义ItemWriter，无论哪种方式，都不会有任何问题。需要注意的一件事是批量输出时的性能和错误处理能力。在使用hibernate作为ItemWriter 时是最常见的，但在使用Jdbc batch 模式时可能也会存在同样的问题。批处理数据库输出没有任何固有的缺陷，如果我们注意 flush 并且数据没有错误的话。但是，在写出时如果发生了什么错误，就可能引起混乱，因为没有办法知道是哪个item引起的异常，甚至是否某个单独的 item 负有责任，如下图所示：



如果 items 在输出之前有缓冲，则遇到任何错误将不会立刻抛出，直到缓冲区刷新之后，提交之前才会抛出。例如，我们假设每一块写出20个item，第15个 item 会抛出

`DataIntegrityViolationException`。如果与 Step 有关, 则20项数据都会写入成功, 因为没有办法知道会出现错误, 直到全部写入完成。一旦调用 `Session#flush()`, 就会清空缓冲区buffer, 而异常也将被放出来。在这一点上, Step无能为力, 事务也必须回滚。通常, 异常会导致 item 被跳过(取决于 skip/retry 策略), 然后该item就不会被输出。然而, 在批处理的情况下, 是没有办法知道到底是哪一项引起的问题, 在错误发生时整个缓冲区都将被写出。解决这个问题的唯一方法就是在每一个 item之后 flush 一下:



这种用法是很常见的, 尤其是在使用Hibernate时, ItemWriter的简单实现建议, 在每次调用 `write()` 之后执行 `flush`。这样做可以让跳过 items 变得可靠, 而Spring Batch 在错误发生后会在内部关注适当粒度的ItemWriter调用。

重用已有服务

- 6.10 重用已存在的 Service

6.10 重用已存在的 Service

批处理系统通常是与其他应用程序相结合的方式使用。最常见的是与一个在线应用系统结合，但也支持与瘦客户端集成，通过移动每个程序所使用的批量数据。由于这些原因，所以很多用户想要在批处理作业中重用现有的DAO或其他服务。Spring容器通过注入一些必要的类就可以实现这些重用。但可能需要现有的服务作为 **ItemReader** 或者 **ItemWriter**，也可以适配另一个Spring Batch类，或其本身就是一个 step 主要的**ItemReader**。为每个需要包装的服务编写一个适配器类是很简单的，而因为这是很普遍的需求，所以 Spring Batch 提供了实现：`ItemReaderAdapter` 和 `ItemWriterAdapter`。两个类都实现了标准的Spring方法委托模式调用，设置也相当简单。下面是一个reader的示例：

```
1. <bean id="itemReader" class="org.springframework.batch.item.adapter.ItemReaderAdapter">
2.     <property name="targetObject" ref="fooService" />
3.     <property name="targetMethod" value="generateFoo" />
4. </bean>
5.
6. <bean id="fooService" class="org.springframework.batch.item.sample.FooService" />
```

特别需要注意的是，`targetMethod` 必须和 **read** 方法行为对等：如果不存在则返回null，否则返回一个 **Object**。其他的值会使框架不知道何时该结束处理，或者引起无限循环或不正确的失败，这取决于 **ItemWriter** 的实现。 **ItemWriter** 的实现同样简单：

```
1. <bean id="itemWriter" class="org.springframework.batch.item.adapter.ItemWriterAdapter">
2.     <property name="targetObject" ref="fooService" />
3.     <property name="targetMethod" value="processFoo" />
4. </bean>
5.
6. <bean id="fooService" class="org.springframework.batch.item.sample.FooService" />
```

输入校验

- 6.11 输入校验

6.11 输入校验

在本章中，已经讨论了很多种用来解析 input 的方法。如果格式不对，那这些基本的实现都是抛出异常。如果数据丢失一部分，`FixedLengthTokenizer` 也会抛出异常。同样，使用 `FieldSetMapper` 时，如果读取超出 `RowMapper` 索引范围的值，又或者返回值类型不匹配，都会抛出异常。所有的异常都会在 `read` 返回之前抛出。然而，他们不能确定返回的 item 是否是合法的。例如，如果其中一个字段是 `age`，很显然不能是负数。解析为数字是没问题的，因为确实存在这个数，所以就不会抛出异常。因为当下已经有大量的第三方验证框架，所以 Spring Batch 并不提供另一个验证框架，而是提供了一个非常简单的接口，其他框架可以实现这个接口来提供兼容：

```
1. public interface Validator {
2.
3.     void validate(Object value) throws ValidationException;
4. }
```

The contract is that the `validate` method will throw an exception if the object is invalid, and return normally if it is valid. Spring Batch provides an out of the box `ItemProcessor`:

约定是如果对象无效则 `validate` 方法抛出一个异常，如果对象合法那就正常返回。Spring Batch 提供了开箱即用的 `ItemProcessor`：

```
1. <bean class="org.springframework.batch.item.validator.ValidatingItemProcessor">
2.     <property name="validator" ref="validator" />
3. </bean>
4.
5. <bean id="validator"
6.     class="org.springframework.batch.item.validator.SpringValidator">
7.     <property name="validator">
8.         <bean id="orderValidator"
9.             class="org.springframework.validation.valang.ValangValidator">
10.            <property name="valang">
11.                <value>
12.                    <![CDATA[
13.                { orderId : ? > 0 AND ? <= 9999999999 : 'Incorrect order ID' : 'error.order.id' }
14.                { totalLines : ? = size(lineItems) : 'Bad count of order lines'
15.                    : 'error.order.lines.badcount'}
16.                { customer.registered : customer.businessCustomer = FALSE OR ? = TRUE
```

```
17.         : 'Business customer must be registered'
18.         : 'error.customer.registration'}
19.     { customer.companyName : customer.businessCustomer = FALSE OR ? HAS TEXT
20.         : 'Company name for business customer is mandatory'
21.         : 'error.customer.companyname'}
22.     ]]>
23.     </value>
24. </property>
25. </bean>
26. </property>
27. </bean>
```

这个示例展示了一个简单的 **ValangValidator**，用来校验 `order` 对象。这样写目的是为了尽可能多地演示如何使用 Valang 来添加校验程序。

不参与持久化的字段

- 6.12 不保存执行状态

6.12 不保存执行状态

默认情况下,所有 **ItemReader** 和 **ItemWriter** 在提交之前都会把当前状态信息保存到 **ExecutionContext** 中。但有时我们又不希望保存这些信息。例如,许多开发者使用处理指示器 (process indicator) 让数据库读取程序 ‘可重复运行 (rerunnable)’。在数据表中添加一个附加列来标识该记录是否已被处理。当某条记录被读取/写入时,就将标志位从 `false` 变为 `true`, 然后只要在SQL语句的where子句中包含一个附加条件, 如 “ `where PROCESSED_IND = false` ”, 就可确保在任务重启后只查询到未处理过的记录。这种情况下,就不需要保存任何状态信息, 比如当前 row number 什么的, 因为在重启后这些信息都没用了。基于这种考虑, 所有的 readers 和 writers 都含有一个 `saveState` 属性:

```
1. <bean id="playerSummarizationSource" class="org.spr...JdbcCursorItemReader">
2.     <property name="dataSource" ref="dataSource" />
3.     <property name="rowMapper">
4.         <bean class="org.springframework.batch.sample.PlayerSummaryMapper" />
5.     </property>
6.     <property name="saveState" value="false" />
7.     <property name="sql">
8.         <value>
9.             SELECT games.player_id, games.year_no, SUM(COMPLETES),
10.             SUM(ATTEMPTS), SUM(PASSING_YARDS), SUM(PASSING_TD),
11.             SUM(INTERCEPTIONS), SUM(RUSHES), SUM(RUSH_YARDS),
12.             SUM(RECEPTIONS), SUM(RECEPTIONS_YARDS), SUM(TOTAL_TD)
13.             from games, players where players.player_id =
14.             games.player_id group by games.player_id, games.year_no
15.         </value>
16.     </property>
17. </bean>
```

上面配置的这个 **ItemReader** 在任何情况下都不会将 entries (状态信息) 存放到 **ExecutionContext** 中。

自定义ItemReaders和ItemWriters

- [6.13 创建自定义 ItemReaders 与 ItemWriters](#)

6.13 创建自定义 ItemReaders 与 ItemWriters

到目前为止,本章已将 Spring Batch 中基本的读取(reading)和写入(writing)概念讲完,还对一些常用的实现进行了讨论。然而,这些都是相当普通的,还有很多潜在的场景可能没有现成的实现。本节将通过一个简单的例子,来演示如何创建自定义的 `ItemReader` 和 `ItemWriter`,并且如何正确地实现和使用。`ItemReader` 同时也将 `ItemStream`,以说明如何让reader(读取器)或writer(写入器)支持重启(restartable)。

自定义ItemReader示例

- 6.13.1 自定义 ItemReader 示例

6.13.1 自定义 ItemReader 示例

为了实现这个目的,我们实现一个简单的 `ItemReader`, 从给定的list中读取数据。我们将实现最基本的 `ItemReader` 功能, `read`:

```
1. public class CustomItemReader<T> implements ItemReader<T>{
2.
3.     List<T> items;
4.
5.     public CustomItemReader(List<T> items) {
6.         this.items = items;
7.     }
8.
9.     public T read() throws Exception, UnexpectedInputException,
10.        NoWorkFoundException, ParseException {
11.
12.         if (!items.isEmpty()) {
13.             return items.remove(0);
14.         }
15.         return null;
16.     }
17. }
```

这是一个简单的类,传入一个 `items` list,每次读取时删除其中的一条并返回。如果list里面没有内容,则将返回null,从而满足 `ItemReader` 的基本要求,测试代码如下所示:

```
1. List<String> items = new ArrayList<String>();
2. items.add("1");
3. items.add("2");
4. items.add("3");
5.
6. ItemReader itemReader = new CustomItemReader<String>(items);
7. assertEquals("1", itemReader.read());
8. assertEquals("2", itemReader.read());
9. assertEquals("3", itemReader.read());
10. assertNull(itemReader.read());
```

使 `ItemReader` 支持重启

现在剩下的问题就是让 *ItemReader* 变为可重启的。到目前这一步,如果发生掉电之类情况,那么必须重新启动 *ItemReader*,而且是从头开始。在很多时候这是允许的,但有时候更好的处理办法是让批处理作业在上次中断的地方重新开始。判断的关键是根据 reader 是有状态的还是无状态的。无状态的 reader 不需要考虑重启的情况,但有状态的则需要根据其最后一个已知的状态来重新启动。出于这些原因,官方建议尽可能地让 reader 成为无状态的,使开发者不需要考虑重新启动的情况。

如果需要保存状态信息,那应该使用 `ItemStream` 接口:

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

每次调用 *ItemStream* 的 `update` 方法时, *ItemReader* 的当前 `index` 都会被保存到给定

的 `ExecutionContext` 中, key 为 `'current.index'`。当调用 `ItemStream` 的 `open` 方法时, `ExecutionContext` 会检查是否包含该 key 对应的条目。如果找到 key, 那么当前索引 index 就好移动到该位置。这是一个相当简单的例子, 但它仍然符合通用原则:

```
1. ExecutionContext executionContext = new ExecutionContext();
2. ((ItemStream)itemReader).open(executionContext);
3. assertEquals("1", itemReader.read());
4. ((ItemStream)itemReader).update(executionContext);
5.
6. List<String> items = new ArrayList<String>();
7. items.add("1");
8. items.add("2");
9. items.add("3");
10. itemReader = new CustomItemReader<String>(items);
11.
12. ((ItemStream)itemReader).open(executionContext);
13. assertEquals("2", itemReader.read());
```

大多数 `ItemReaders` 具有更加复杂的重启逻辑。例如 `JdbcCursorItemReader`, 存储了游标 (`Cursor`) 中最后所处理的行的 row id。

还值得注意的是 `ExecutionContext` 中使用的 key 不应该过于简单。这是因为 `ExecutionContext` 被一个 `Step` 中的所有 `ItemStreams` 共用。在大多数情况下, 使用类名加上 key 的方式应该就足以保证唯一性。然而, 在极端情况下, 同一个类的多个 `ItemStream` 被用在同一个 `Step` 中时(如需要输出两个文件的情况), 就需要更加具备唯一性的 name 标识。出于这个原因, Spring Batch 的许多 `ItemReader` 和 `ItemWriter` 实现都有一个 `setName()` 方法, 允许覆盖默认的 key name。

自定义ItemWriter示例

- 6.13.2 自定义 ItemWriter 示例
 - 让 `ItemWriter` 支持重新启动

6.13.2 自定义 ItemWriter 示例

自定义实现 `ItemWriter` 和上一小节所讲的 `ItemReader` 有很多方面是类似，但也有足够多的不同之处。但增加可重启特性在本质上是一样的，所以本节的示例就不再讨论这一点。和

`ItemReader` 示例一样，为了简单我们使用的参数也是 `List`：

```

1. public class CustomItemWriter<T> implements ItemWriter<T> {
2.
3.     List<T> output = TransactionAwareProxyFactory.createTransactionalList();
4.
5.     public void write(List<? extends T> items) throws Exception {
6.         output.addAll(items);
7.     }
8.
9.     public List<T> getOutput() {
10.        return output;
11.    }
12. }
```

让 `ItemWriter` 支持重新启动

要让 `ItemWriter` 支持重新启动，我们将会使用和 `ItemReader` 相同的过程，实现并添加 `ItemStream` 接口来同步 execution context。在例子中我们可能要记录处理过的items数量,并添加为到 footer 记录。我们可以在 `ItemWriter` 的实现类中同时实现 `ItemStream`，以便在 stream 重新打开时从执行上下文中取回原来的数据重建计数器。

实际开发中，如果自定义 `ItemWriter` restartable(支持重启),则会委托另一个 writer(例如，在写入文件时)，否则会写入到关系型数据库(支持事务的资源)中，此时 `ItemWriter` 不需要 restartable特性,因为自身是无状态的。如果你的 writer 有状态，则应该实现2个接口：

`ItemStream` 和 `ItemWriter`。请记住，writer客户端需要知道 `ItemStream` 的存在，所以需要在 xml 配置文件中将其注册为 stream。

扩展与并行处理

- [7. 扩展与并行处理](#)

7. 扩展与并行处理

很多批处理问题都可以通过单进程、单线程的工作模式来完成，所以在想要做一个复杂设计和实现之前，请审查你是否真的需要那些超级复杂的实现。

衡量实际作业(job)的性能，看看最简单的实现是否能满足需求：即便是最普通的硬件，也可以在一分钟内读写上百MB数据文件。

当你准备使用并行处理技术来实现批处理作业时，Spring Batch提供一系列选择，本章将对他们进行讲述，虽然某些功能不在本章中涵盖。从高层次的抽象角度看，并行处理有两种模式：单进程，多线程模式；或者多进程模式。还可以将他分成下面这些种类：

- 多线程Step(单个进程)
- 并行Steps(单个进程)
- 远程分块Step(多个进程)
- 对Step分区（单/多个进程）

下面我们先回顾一下单进程方式，然后再看多进程方式。

多线程 Step

- 7.1 多线程 Step

7.1 多线程 Step

启动并行处理最简单的方式就是在 Step 配置中加上一个 `TaskExecutor`，比如，作为 `tasklet` 的一个属性：

```
1. <step id="loading">
2.     <tasklet task-executor="taskExecutor">...</tasklet>
3. </step>
```

上面的示例中，`taskExecutor`指向了另一个实现 `TaskExecutor` 接口的Bean。`TaskExecutor` 是一个标准的Spring接口，具体有哪些可用的实现类，请参考 [Spring用户指南](#)。最简单的多线程 `TaskExecutor` 实现是 `SimpleAsyncTaskExecutor`。

以上配置的结果就是在 Step 在(每次提交的块)记录的读取，处理，写入时都会在单独的线程中执行。请注意，这段话的意思就是在要处理的数据项之间没有了固定的顺序，并且一个非连续块可能包含项目相比，单线程的例子。此外`executor`还有一些限制(例如，如果它是由一个线程池在后台执行的)，有一个`tasklet`的配置项可以调整，`throttle-limit`默认为4。你可能根据需要增加这个值以确保线程池被充分利用，如：

```
1. <step id="loading"> <tasklet
2.     task-executor="taskExecutor"
3.     throttle-limit="20">...</tasklet>
4. </step>
```

还需要注意在step中并发使用连接池资源时可能会有一些限制，例如数据库连接池 `DataSource`。请确保连接池中的资源数量大于或等于并发线程的数量。

在一些常见的批处理情景中，对使用多线程Step有一些实际的限制。Step中的许多部分(如`readers`和`writers`)是有状态的，如果某些状态没有进行线程隔离，那么这些组件在多线程Step中就是不可用的。特别是大多数Spring Batch提供的`readers`和`writers`不是为多线程而设计的。但是，我们也可以使用无状态或线程安全的`readers`和`writers`，可以参考Spring Batch Samples中(`parallelJob`)的这个示例(点击进入[Section 6.12, "Preventing State Persistence"](#))，示例中展示了通过指示器来跟踪数据库input表中的哪些项目已经被处理过，而哪些还没有被处理。

Spring Batch 提供了 `ItemWriter` 和 `ItemReader` 的一些实现。通常在javadoc中会指明是否是线程安全的，或者指出在并发环境中需要注意哪些问题。假若文档中没有明确说明，你只能通过查

看源代码来看看是否有什么线程不安全的共享状态。一个并非线程安全的 `reader`，也可以在你自己处理了同步的代理对象中高效地使用。

如果你的step中写操作和处理操作所消耗的时间更多，那么即使你对`read()`操作加锁进行同步，也会比你在单线程环境中执行要快很多。

并行 Steps

- [7.2 并行 Steps](#)

7.2 并行 Steps

只要需要并行的程序逻辑可以划分为不同的职责,并分配给各个独立的step,那么就可以在单个进程中并行执行。并行Step执行很容易配置和使用,例如,将执行步骤(**step1, step2**)和步骤3**step3**并行执行,则可以向下面这样配置一个流程:

```
1. <job id="job1">
2.     <split id="split1" task-executor="taskExecutor" next="step4">
3.         <flow>
4.             <step id="step1" parent="s1" next="step2"/>
5.             <step id="step2" parent="s2"/>
6.         </flow>
7.         <flow>
8.             <step id="step3" parent="s3"/>
9.         </flow>
10.    </split>
11.    <step id="step4" parent="s4"/>
12. </job>
13.
14. <beans:bean id="taskExecutor" class="org.spr...SimpleAsyncTaskExecutor"/>
```

可配置的“task-executor”属性是用来指明应该用哪个TaskExecutor实现来执行独立的流程。默认是SyncTaskExecutor,但有时需要使用异步的TaskExecutor来并行运行某些步骤。请注意,这项工作将确保每一个流程在聚合之前完成,并进行过渡。

更详细的信息请参考 [Section 5.3.5, “Split Flows”](#)。

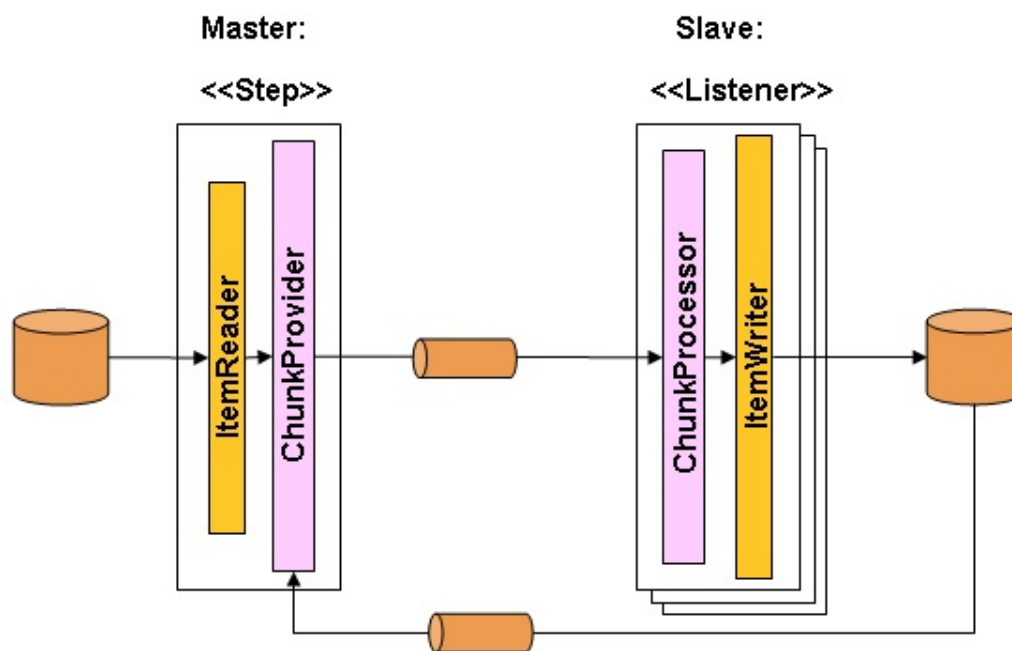
远程分块

- 7.3 远程分块(Remote Chunking)

7.3 远程分块(Remote Chunking)

使用远程分块的Step被拆分成多个进程进行处理, 多个进程间通过中间件实现通信。下面是一幅模型示意图:

Remote Chunking



Master组件是单个进程,从属组件(Slaves)一般是多个远程进程。如果Master进程不是瓶颈的话,那么这种模式的效果几乎是最好的,因此应该在处理数据比读取数据消耗更多时间的情况下使用(实际应用中常常是这种情形)。

Master组件只是Spring Batch **Step** 的一个实现,只是将ItemWriter替换为一个通用的版本,这个通用版本“知道”如何将数据项的分块作为消息(messages)发送给中间件。从属组件(Slaves)是标准的监听器(listeners),不论使用哪种中间件(如使用JMS时就是 **MessageListeners**),Slaves的作用都是处理数据项的分块(chunks),可以使用标准的 **ItemWriter** 或者是 **ItemProcessor**加上一个 **ItemWriter**,使用的接口是 **ChunkProcessor interface**。使用此模式的一个优点是: reader, processor和 writer 组件都是现成的(就和

在本机执行的step一样)。数据项被动态地划分,工作是通过中间件共享的,因此,如果监听器都是饥饿模式的消费者,那么就自动实现了负载均衡。

中间件必须持久可靠,能保证每个消息都会被分发,且只分发给单个消费者。JMS是很受欢迎的解决方案,但在网格计算和共享内存产品空间里还有其他可选的方式(如 Java Spaces服务; 为Java对象提供分布式的共享存储器)。

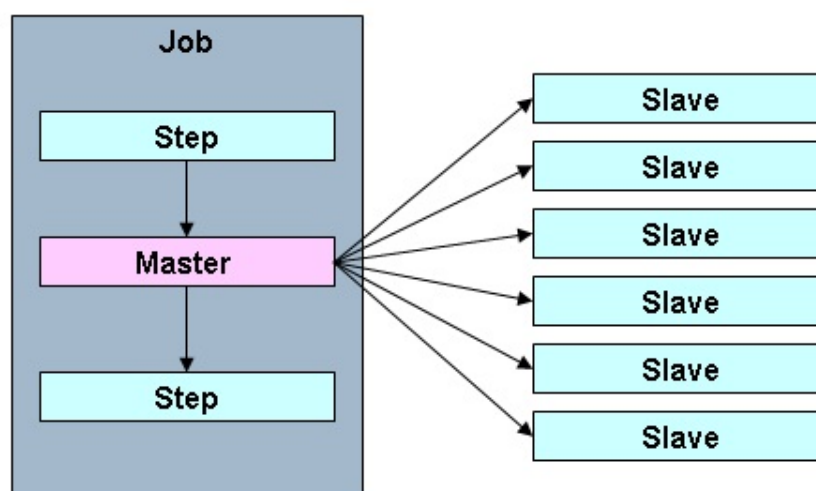
分区

- 7.4 分区
 - 7.4.1 分区处理器(PartitionHandler)
 - 7.4.2 分割器(Partitioner)
 - 7.4.3 将输入数据绑定到 Steps

7.4 分区

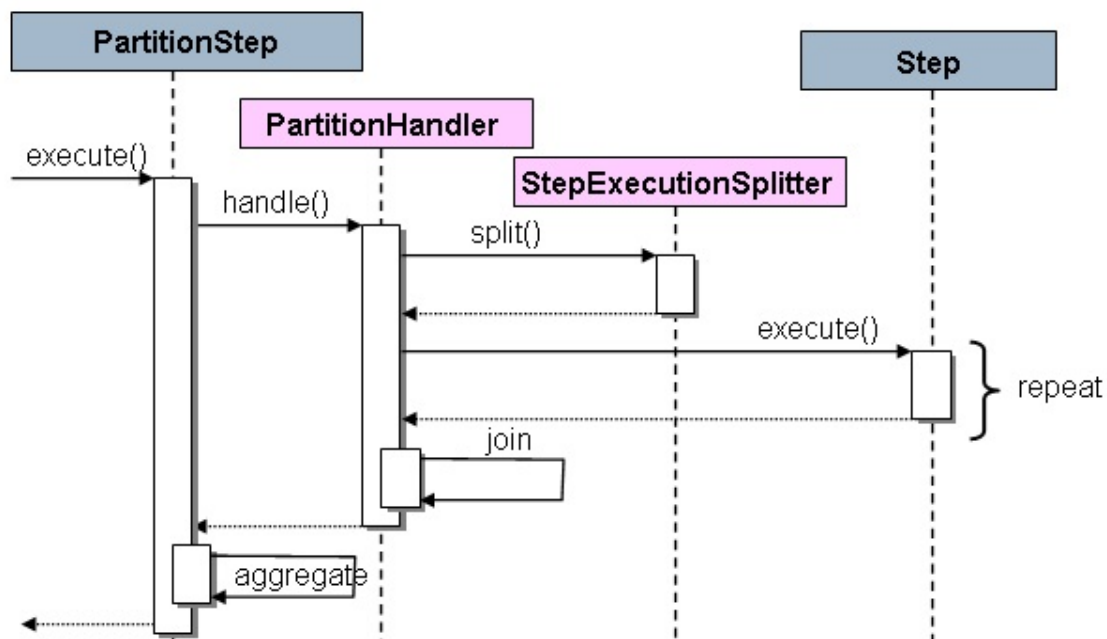
Spring Batch也为Step的分区执行和远程执行提供了一个SPI(服务提供者接口)。在这种情况下,远端的执行程序只是一些简单的Step实例,配置和使用方式都和本机处理一样容易。下面是一幅实际的模型示意图:

Partitioning Overview



在左侧执行的作业(Job)是串行的Steps,而中间的那一个Step被标记为 Master。图中的 Slave 都是一个Step的相同实例,对于作业来说,这些Slave的执行结果实际上等价于就是Master的结果。Slaves通常是远程服务,但也有可能是本地执行的其他线程。在此模式中,Master发送给Slave的消息不需要持久化(durable),也不要求保证交付:对每个作业执行步骤来说,保存在 **JobRepository** 中的Spring Batch元信息将确保每个Slave都会且仅会被执行一次。

Spring Batch的SPI由Step的一个专门的实现(**PartitionStep**),以及需要由特定环境实现的两个策略接口组成。这两个策略接口分别是 **PartitionHandler** 和 **StepExecutionSplitter**,他们的角色如下面的序列图所示:



此时在右边的Step就是“远程”Slave,所以可能会有多个对象 和/或 进程在扮演这一角色,而图中的PartitionStep 在驱动(/控制)整个执行过程。PartitionStep的配置如下所示:

```

1. <step id="step1.master">
2.     <partition step="step1" partitioner="partitioner">
3.         <handler grid-size="10" task-executor="taskExecutor"/>
4.     </partition>
5. </step>
  
```

类似于多线程step的 `throttle-limit` 属性, `grid-size`属性防止单个Step的任务执行器过载。

在Spring Batch Samples示例程序中有一个简单的例子在单元测试中可以拷贝/扩展(详情请参考 ***PartitionJob.xml** 配置文件)。

Spring Batch 为分区创建执行步骤,名如“step1:partition0”,等等,所以我们经常把Master step叫做“step1:master”。在Spring 3.0中也可以为Step指定别名(通过指定 `name` 属性,而

不是 `id` 属性)。

7.4.1 分区处理器(PartitionHandler)

PartitionHandler组件知道远程网格环境的组织结构。它可以发送**StepExecution**请求给远端Steps,采用某种具体的数据格式,例如DTO.它不需要知道如何分割输入数据,或者如何聚合多个步骤执行的结果。一般来说它可能也不需要了解弹性或故障转移,因为在许多情况下这些都是结构的特性,无论如何Spring Batch总是提供了独立于结构的可重启能力:一个失败的作业总是会被重新启动,并且只会重新执行失败的步骤。

PartitionHandler接口可以有各种结构的实现类:如简单RMI远程方法调用,EJB远程调用,自定义web服务、JMS、Java Spaces,共享内存网格(如Terracotta或Coherence)、网格执行结构(如GridGain)。Spring Batch自身不包含任何专有网格或远程结构的实现。

但是Spring Batch也提供了一个有用的**PartitionHandler**实现,在本地分开的线程中执行Steps,该实现类名为 **TaskExecutorPartitionHandler**,并且他是上面的XML配置中的默认处理器。还可以像下面这样明确地指定:

```

1. <step id="step1.master">
2.     <partition step="step1" handler="handler"/>
3. </step>
4.
5. <bean class="org.spr...TaskExecutorPartitionHandler">
6.     <property name="taskExecutor" ref="taskExecutor"/>
7.     <property name="step" ref="step1" />
8.     <property name="gridSize" value="10" />
9. </bean>

```

gridSize决定要创建的独立的step执行的数量,所以它可以配置为**TaskExecutor**中线程池的大小,或者也可以设置得比可用的线程数稍大一点,在这种情况下,执行块变得更小一些。

TaskExecutorPartitionHandler 对于IO密集型步骤非常给力,比如要拷贝大量的文件,或复制文件系统到内容管理系统时。它还可用于远程执行的实现,通过为远程调用提供一个代理的步骤实现(例如使用Spring Remoting)。

7.4.2 分割器(Partitioner)

分割器有一个简单的职责:仅为新的step实例生成执行环境(contexts),作为输入参数(这样重启时就不需要考虑)。该接口只有一个方法:

```

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

```

```
3. }
```

这个方法的返回值是一个Map对象,将每个Step执行分配的唯一名称(Map泛型中的 **String**),和与其相关的输入参数以**ExecutionContext** 的形式做一个映射。

这个名称随后在批处理 meta data 中作为分区 **StepExecutions** 的Step名字显示。

ExecutionContext仅仅只是一些 名-值对的集合,所以它可以包含一系列的主键,或行号,或者是输入文件的位置。然后远程Step 通常使用 `#{...}`占位符来绑定到上下文输入(在 step作用域内的后期绑定),详情请参见下一节。

step执行的名称(**Partitioner**接口返回的 **Map** 中的 **key**)在整个作业的执行过程中需要保持唯一,除此之外没有其他具体要求。要做到这一点,并且需要一个对用户有意义的名称,最简单的方法是使用 前缀+后缀 的命名约定,前缀可以是被执行的Step的名称(这本身在作业**Job**中就是唯一的),后缀可以是一个计数器。在框架中有一个使用此约定的 **SimplePartitioner**。

有一个可选接口 **PartitionNameProvider** 可用于和分区本身独立的提供分区名称。如果一个 **Partitioner** 实现了这个接口,那么重启时只有names会被查询。如果分区是重量级的,那么这可能是一个很有用的优化。很显然,**PartitionNameProvider**提供的名称必须和**Partitioner**提供的名称一致。

7.4.3 将输入数据绑定到 Steps

因为step的输入参数在运行时绑定到**ExecutionContext**中,所以由相同配置的 **PartitionHandler**执行的steps是非常高效的。通过 **Spring Batch**的**StepScope**特性这很容易实现(详情请参考 [后期绑定](#))。例如,如果 **Partitioner** 创建 **ExecutionContext** 实例,每个step执行都以**fileName**为key 指向另一个不同的文件(或目录),则 **Partitioner** 的输出看起来可能像下面这样:

表 7.1. 由执行目的目录处理**Partitioner**提供的的step执行上下文名称示例

1.	Step Execution Name (key)	ExecutionContext (value)
2.	filecopy:partition0	fileName=/home/data/one
3.	filecopy:partition1	fileName=/home/data/two
4.	filecopy:partition2	fileName=/home/data/three

然后就可以将文件名绑定到 step 中, step使用了执行上下文的后期绑定:

```
1. <bean id="itemReader" scope="step"
2.     class="org.spr...MultiResourceItemReader">
3.     <property name="resource" value="#{stepExecutionContext[fileName]}"/>
4. </bean>
```


重复执行

- 8. 重复执行
- 8.1 RepeatTemplate
- 8.1.1 RepeatContext
- 8.1.2 RepeatStatus
- 8.2 结束策略
- 8.3 异常处理
- 8.4 监听
- 8.5 并行处理
- 8.6 声明式迭代

8. 重复执行

8.1 RepeatTemplate

批处理是重复的动作-无论是作为一个简单的优化，或作为工作的一部分。策划和归纳重复以及提供一个相当于迭代器的框架，Spring Batch提供RepeatOperations接口，这个RepeatOperations接口看起来像是这样：

```
1. public interface RepeatOperations {
2.
3.     RepeatStatus iterate(RepeatCallback callback) throws RepeatException;
4.
5. }
```

上面代码中的callback是一个简单的接口, 允许您插入一些重复的业务逻辑：

```
1. public interface RepeatCallback {
2.
3.     RepeatStatus doInIteration(RepeatContext context) throws Exception;
4.
5. }
```

上面代码中的callback是反复执行的，直到执行决定的迭代应该结束。这些接口的返回值是一个枚举，可以是repeatstatus.continuable或repeatstatus.finished二者其中任何一个。

一个RepeatStatus传达信息给重复操作的调用者，是否有更多的工作要做。

一般来说，实现repeatoperations应检查repeatstatus并使用它作为决定结束迭代的一部分。

任何callback都希望发出信号给调用者，如果没有更多的工作要做，可以返回repeatstatus.finished

最简单的repeatoperations通用实现repeattemplate。它可以这样使用：

```
1. RepeatTemplate template = new RepeatTemplate();
2.
3. template.setCompletionPolicy(new FixedChunkSizeCompletionPolicy(2));
4.
5. template.iterate(new RepeatCallback() {
6.
7.     public ExitStatus doInIteration(RepeatContext context) {
8.         // Do stuff in batch...
9.         return ExitStatus.CONTINUABLE;
10.    }
11.
12. });
```

在这个例子中我们返回repeatstatus.continuable表明这儿还有更多的工作要做。该callback也可以返回exitstatus.finished信号给调用者，如果这儿没有更多的工作要做。

8.1.1 RepeatContext

上文提到的repeatcallback方法的参数是一个repeatcontext。许多callback会简单地忽略上下文，但必要时它可以作为属性包，存储临时数据迭代的持续时间。在迭代方法返回后，上下文将不再存在。

如果在一个嵌套迭代过程，RepeatContext将有一个父上下文。该父上下文是偶尔用于存储数据，需要调用迭代之间共享。

例如这种情况：如果你想计算一个事件发生迭代的次数和记住它后续调用。

8.1.2 RepeatStatus

RepeatStatus是一个枚举，Spring Batch中用表明是否处理完成。这些都是RepeatStatus可能的值：

Table 8.1. ExitStatus Properties

值	描述
CONTINUABLE	这儿还有很多工作要做。
FINISHED	没有更多的重复发生。

RepeatStatus值也可以调用RepeatStatus中的and()方法来做逻辑与操作。效果就是和一个可持续的标志位做逻辑与操作。换句话说,如果状态是FINISHED,那么结果必须是FINISHED的

8.2 结束策略

CompletionPolicy可以中止RepeatTemplate内部的循环迭代,CompletionPolicy同样也是一个RepeatContext工厂。RepeatTemplate有责任使用当前的策略来创建一个RepeatContext并且在迭代时的每一个阶段传递RepeatCallback。如果迭代已经完成,在doInIteration的回调完成之后,RepeatTemplate必须通知CompletionPolicy来更新自己的状态(状态存储在RepeatContext中)。

Spring Batch的CompletionPolicy提供了一些简单的通用实现。SimpleCompletionPolicy只允许执行固定的次数(任何时候RepeatStatus.FINISHED强制的提早完成)。

用户可能需要自己完成实现更复杂的决定策略。例如,一个批处理窗口,在线系统需要使用一个定制的策略阻止批任务执行一次。

8.3 异常处理

如果RepeatCallback内部抛出一个异常。RepeatCallback通过ExceptionHandler决定是否再抛出这个异常。

```

1. public interface ExceptionHandler {
2.
3.     void handleException(RepeatContext context, Throwable throwable)
4.     throws RuntimeException;
5.
6. }
```

一个常见的用例是计算指定类型异常的数量,并到达限制,为此Spring Batch提供SimpleLimitExceptionHandler和稍微更灵活的RethrowOnThresholdExceptionHandler。RethrowOnThresholdExceptionHandler有limit属性和异常类型,包括当前异常类型及其所有子类提供的类型。其他类型正常被抛出,指定的异常类型会被忽略,直到达到数量限制时抛出。SimpleLimitExceptionHandler一个重要可选择的属性useparent该属性为boolean类型。它默认为false,仅限制当前的RepeatContext。设置为true时,限制保持在兄弟上下文的嵌套迭代中(例如 一个块内的一步)

8.4 监听

不同的迭代往往能够在横切点获得有用的回调。为了这个目的，Spring Batch提供repeatlistener接口。RepeatTemplate允许用户注册RepeatListeners, 监听器会回调RepeatContext and RepeatStatus, 这两个参数在整个迭代过程中都可用。

这个interface看起来像是这样：

```

1. public interface RepeatListener {
2.     void before(RepeatContext context);
3.     void after(RepeatContext context, RepeatStatus result);
4.     void open(RepeatContext context);
5.     void onError(RepeatContext context, Throwable e);
6.     void close(RepeatContext context);
7. }
```

打开和关闭回调之前和之后整个迭代。之前, 之后和onError适用于单独RepeatCallback调用。

值得注意的是，当有一个以上的监听，他们是在一个列表，所以有一个顺序。

8.5 并行处理

RepeatOperations的实现不限于顺序执行的回调。一些实现能够并行执行的回调，这是很重要的。为此，Spring Batch提供TaskExecutorRepeatTemplate, 它使用Spring TaskExecutor 策略运行RepeatCallback。默认使用SynchronousTaskExecutor, 整个迭代执行的影响是在同一线程中（和一个正常的repeattemplate相同）

8.6 声明式迭代

S有时会有一些业务处理, 你想知道每次重复产生的结果。典型的例子就是消息的优化管道—这是更有效地处理一批消息, 如果他们频繁到来, 超过每条消息单独的事务的成本。为了这个目的, Spring Batch AOP提供了一个拦截器, 封装RepeatOperations方法的调用。

RepeatOperationsInterceptor执行拦截方法和重复的RepeatTemplate.CompletionPolicy。

下面是一个示例使用Spring AOP命名空间的声明式迭代重复服务调用一个名为processMessage的方法（如何配置AOP拦截器更多细节见Spring用户指南）：

```

1. <aop:config>
2.     <aop:pointcut id="transactional"
3.         expression="execution(* com.*Service.processMessage(..))" />
4.     <aop:advisor pointcut-ref="transactional"
```

```
5.         advice-ref="retryAdvice" order="-1"/>
6.     </aop:config>
7.
8. <bean id="retryAdvice" class="org.spr...RepeatOperationsInterceptor"/>
```

上面的例子使用了默认的repeattemplate在拦截。要改变政策，监听器等。你只需要为拦截器注入repeattemplate实例。

如果拦截方法返回void那么拦截器总是返回ExitStatus.CONTINUABLE（因此这儿有个危险如果CompletionPolicy不含有一个有限的结束点会形成一个无限循环）。否则返回ExitStatus.CONTINUABLE直到拦截方法的返回值为空，此时它返回ExitStatus.FINISHED。所以目标方法内的业务逻辑可以发出信号，没有更多的工作要做返回null，或RepeatTemplate.ExceptionHandler重新抛出异常。

重试处理

- [重试处理](#)

重试处理

重试模板

- 重试模板" level="1">9.1 重试模板
- 重试上下文" level="1">9.1.1 重试上下文
- 恢复回调" level="1">9.1.2 恢复回调
- 无状态的重试" level="1">9.1.3 无状态的重试
- 状态性重试" level="1">9.1.4 状态性重试

重试模板" class="reference-link">9.1 重试模板



请注意 这个重试功能在Spring Batch 2.2.0里面退出，现在它是 `Spring Retry` 的一部分。

为了让这个进程更稳定，更小的失败性。有时它帮助自动重试一个失败的操作以防止它可能在后续的尝试成功。本质上，这种处理会导致误差。例如，远程调用网络服务或RMI服务失败是由于在短暂的数据更新后，网络故障或冻结异常 `aDeadLockLoserException`。例如重试这种自动化操作，`Spring Batch` 有重试操作的策略。

重试操作接口如下：

```

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

```

基本回调是一个简单的接口，允许您插入一些业务逻辑重试：

```

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

```

执行回调如果失败了（抛出异常），它会重试直至成功或者执行决定终止。在重试界面，有一些超载的操作方法，专门处理恢复各种用户实例。也可以在重试阶段，允许客户端和实现信息存储之间的调用（后面详细讨论）

最简单通用的实施 `RetryOperations` 就是 `Retry Template`。就像这样

```

1. RetryTemplate template = new RetryTemplate();
2.
3. TimeoutRetryPolicy policy = new TimeoutRetryPolicy();
4. policy.setTimeout(30000L);
5.
6. template.setRetryPolicy(policy);
7.
8. Foo result = template.execute(new RetryCallback<Foo>() {
9.
10.     public Foo doWithRetry(RetryContext context) {
11.         // Do stuff that might fail, e.g. webservice operation
12.         return result;
13.     }
14.
15. });

```

在这个例子中我们执行web服务调用和返回结果给用户。如果调用失败然后重试,直到超时。

重试上下文" class="reference-link">9.1.1 重试上下文

`RetryCallback` 的方法参数是 `RetryContext` 。许多回调会简单地忽略上下文,但必要时它可以作为一个属性包为迭代的持续时间存储数据。

在同一线程中,如果一个嵌套在重试过程中, `RetryContext` 会有一个母本。这个母本偶尔在存储数据用于调用和执行共享很有帮助

恢复回调" class="reference-link">9.1.2 恢复回调

当一个重试耗尽这个 `RetryOperations` 可以传递控制一个不同的回调,是恢复回调。要使用该功能的客户只是通过同样的方法回调在一起,例如:

```

1. Foo foo = template.execute(new RetryCallback<Foo>() {
2.     public Foo doWithRetry(RetryContext context) {
3.         // business logic here
4.     },
5.     new RecoveryCallback<Foo>() {
6.         Foo recover(RetryContext context) throws Exception {
7.             // recover logic here
8.         }
9. });

```

在这个模板决定终止之前如果业务逻辑没有执行成功，然后客户端有机会做一些交替处理通过恢复回调处理

无状态的重试" class="reference-link">9.1.3 无状态的重试

在一个简单的实例中，重试只是一个while循环，`RetryTemplate` 可以不断尝试，直到成功或失败，`RetryContext` 包含一些状态来确定是否重试或中止，但这种状态在堆栈上没有必要在任何时候地方存储，所以我们称之为无状态的重试，无状态和有状态重试之间的区别是包含在实施 `RetryPolicy` 中(`RetryTemplate` 可以同时处理)，在一个无状态的重试，执行回调总是在同一线程上耗尽而失败

状态性重试" class="reference-link">9.1.4 状态性重试

有一些需要特别考虑的是，当失败引起事务资源无效，这并不适用于一个简单的远程调用，因为没有事务资源(通常)，但它有时适用于数据库更新，尤其是使用 `Hibernate`，这个案例中，我们重新抛出这个异常，我们称之为立即失效，这样的话，事物资源可以回滚，我们可以启用一个新的。

在这些情况下无状态重试还不够好，因为 `re-throw` 和回滚必然涉及离开 `RetryOperations.execute()` 方法和潜在损失的上下文堆栈。为了避免失去它我们必须引入存储策略提升了堆栈并把它(至少)放在堆存储中，为此 `Spring Batch` 提供了一个存储策略 `RetryContextCache` 可以注入 `RetryTemplate`。内存中有 `RetryContextCache` 默认的实现，使用一个简单的 `Map`。多个进程的高级用法在集群环境中也会考虑实现集群缓存的 `RetryContextCache`(不过，在集群环境中这可能是过度)。

`RetryOperations` 的责任之一就是在执行新任务时候识别错误操作(通常是包裹在一个新的事务)。为了使它更方便，`Spring Batch` 提供抽象的 `RetryState`。`RetryOperations` 结合特殊执行操作方法

识别错误的操作方法是识别跨多个调用的重试。辨别出这个状态，用户可以提供 `RetryState` 对象返回一个唯一键识别项。标识符用作 `RetryContextCache` 的一个关键。



警告

实现Object的equals方法和hashCode方法要非常小心，关键是在返回重试状态。最好的建议是使用一个业务主键来标识这个项目，在JMS 消息的 `message Id` 可以使用的情况下。

当重试停止也有选择以不同的方式处理失败的项，而不是调用`RetryCallback`(假定现在可能失败)。就像在无状态的情况下，这个选项是 `RecoveryCallback` 提供的，也可以用越

过 `RetryOperations` 的执行方法提供。

重试与否实际上是委托给一个普通的 `RetryPolicy` ,所以通常的使自己关心的限制和超时是可以注入的(见下文)。

重试策略

- 重试策略" level="1">9.2 重试策略

重试策略" class="reference-link">9.2 重试策略

在 `RetryTemplate` 里面, 执行 `excuter` 方法是重试还是失败是由 `RetryPolicy` 决定的, 这也是一个 `RetryContext` 工厂. 这个 `RetryTemplate` 有责任使用当前的策略创建一个 `RetryContext` 并且把它注入到 `RetryCallback` 在每一次尝试中. 回调失败后 `RetryTemplate` 必须由 `RetryPolicy` 决定使其更新状态(存储在 `RetryContext` 中), 然后它询问策略是否会作出其他决定. 如果不能作出其他决定, (例如达到极限或超时), 政策有责任处理这个状态. 简单的操作将会抛出异常它会引起封闭事务的回滚. 更多更复杂的操作可能会采取一些回复操作, 在这种情况下, 事务可以保持不变.



提示

失败是固有的, 无论重试与否, 如果在业务逻辑中总是抛出相同的异常. 这不能帮助重试它. 所以不要重试所有异常类型, 试着只关注那些你期望可能重试的异常, 它虽然对业务逻辑重试变得更具侵略性, 但是它没有危害性. 如果你事先知道它会失败还去花些时间来证明, 只是浪费时间.

`Spring Batch` 提供给无状态 `RetryPolicy` 一些简单通用的实现, 例如 `SimpleRetryPolicy`, 和在上面的示例中使用的 `TimeoutRetryPolicy`。

`SimpleRetryPolicy` 只允许重试一些异常类型的命名列表, 它也有一个 `"fatal"` 的异常列表, 不应重试. 并且这个列表重写了 `retryable` 的列表, 以便它可以用来很好地控制重试行为:

```

1. SimpleRetryPolicy policy = new SimpleRetryPolicy();
2. // Set the max retry attempts
3. policy.setMaxAttempts(5);
4. // Retry on all exceptions (this is the default)
5. policy.setRetryableExceptions(new Class[] {Exception.class});
6. // ... but never retry IllegalStateException
7. policy.setFatalExceptions(new Class[] {IllegalStateException.class});
8.
9. // Use the policy...
10. RetryTemplate template = new RetryTemplate();
11. template.setRetryPolicy(policy);
12. template.execute(new RetryCallback<Foo>() {
13.     public Foo doWithRetry(RetryContext context) {
14.         // business logic here
15.     }
16. });

```

还有一个更灵活的实现叫 `ExceptionClassifierRetryPolicy`。它允许用户配置不同的重试行为为了一组固定的异常类型集合，虽然 `ExceptionClassifier` 抽象类型，分类器上的政策是通过调用异常转换为一个委托 `RetryPolicy`，举个例子，一个异常类型能比映射不通的策略重试更多的次数。

用户需要为更多的客户定制的决定实施他们自己重试策略。例如，如果有一个清楚的详细的解决方案，是归类到可以重试还是不可以重试

补偿策略

- 补偿策略" level="1">9.3 补偿策略

补偿策略" class="reference-link">9.3 补偿策略

在瞬时失效之后做一个尝试它常常有助于在再试一次之前等待一会。因为通常失败是有一些问题引起的它只能通过等待来解决这些问题，如果 `RetryCallback` 失败，`RetryTemplate` 可以暂停执行取决于 `BackoffPolicy` 在适当的位置

```
1. public interface BackoffPolicy {
2.
3.     BackOffContext start(RetryContext context);
4.
5.     void backOff(BackOffContext backOffContext)
6.         throws BackOffInterruptedException;
7.
8. }
```

`backoffPolicy` 可以不受约束的以任何选择的方式实现 `backoff`。 `spring batch` 这个政策创造性的使用了 `object.wait()`。通常情况下等待期间以指数方式上升。避免两个重试锁定步骤和失败。这是网上学到的，为了这个目的 `spring batch` 提供了 `ExponentialBackoffPolicy`。

监听器

- [监听器](#) `level="1">9.4 监听器`

监听器" `class="reference-link">9.4 监听器`

常常它是有用的能够接受附加的回调为了切割关注点穿过一些不同的重试

为了这个目的 `Spring Batch` 提供了 `RetryListene` 接口, `RetryTemplate` 允许使用者注册 `RetryListene`, 并且他们将发送回调随 `RetryContext` 和 `Throwable`, 在迭代期间可用。

这个接口看起来像这样:

```
1. public interface RetryListener {
2.
3.     void open(RetryContext context, RetryCallback<T> callback);
4.
5.     void onError(RetryContext context, RetryCallback<T> callback, Throwable e);
6.
7.     void close(RetryContext context, RetryCallback<T> callback, Throwable e);
8. }
```

`open` 和 `close` 回调在整个的重试前后调用在简单的实例和 `onError` 适用于个体的 `RetryCallback` 调用, `close`方法可能也需要接收一个 `Throwable`. 如果已经有一个错误它是最后一个在 `RetryCallback` 抛出。

注意当有多余一个监听的时候, 它们是在一个列表。所以它们是顺序的, 在这种情况下, `open`方法将被顺序访问执行, `onError` 和 `close`方法被倒序执行。

声明式重试

- 声明式重试" level="1">9.5 声明式重试

声明式重试" class="reference-link">9.5 声明式重试

有时一些业务逻辑的发生你每次都想重试它。最经典的例子就是远程调用，为了这个目的 `Spring Batch` 提供了一个Aop拦截器用来封装一个方法调用 `RetryOperations`。`RetryOperationsInterceptor` 依据 `RetryPolicy` 提供的 `RepeatTemplate` 执行拦截方法和重试失败。

下面是一个示例使用Spring AOP命名空间的声明式迭代重复一个服务调用的一个方法称为 `remoteCall` (AOP如何配置拦截器,更多细节见Spring用户指南):

```

1. <aop:config>
2.     <aop:pointcut id="transactional"
3.         expression="execution(* com.*Service.remoteCall(..))" />
4.     <aop:advisor pointcut-ref="transactional"
5.         advice-ref="retryAdvice" order="-1"/>
6. </aop:config>
7.
8. <bean id="retryAdvice"
9.     class="org.springframework.batch.retry.interceptor.RetryOperationsInterceptor"/>

```

在上面的示例中在拦截器内使用了一个默认的 `RetryTemplate`。改变策略和监听，你只需要在拦截器中注入一个 `RetryTemplate` 实例。

单元测试

- [单元测试](#) level="1">10 单元测试

单元测试" [class="reference-link">10 单元测试](#)

与其他应用程序一样，作为批处理任务的一部分去编写单元测试用例是非常重要的。Spring核心文档极其详尽的描述了如何使用单元测试和集成测试，所以这里就不再重复。然而，思考一下点对点的批处理任务是很重要的，这是本章的重点。 `spring-batch-test` 项目包含的类，将有助于使点到点的测试方法更容易。

创建一个单元测试

- 创建一个单元测试" level="1">10.1 创建一个单元测试

创建一个单元测试" class="reference-link">10.1 创建一个单元测试

为了让单元测试跑一个批处理的任务，这个框架必须加载这个任务的 `ApplicationContext`，两个注解的触发方式：

- `@RunWith(SpringJUnit4ClassRunner.class)`:表明这个类需要使用 `Spring` 的测试工具
- `@ContextConfiguration(locations = {...})`:表明哪些xml文件包含 `ApplicationContext`

```
1. @RunWith(SpringJUnit4ClassRunner.class)
2. @ContextConfiguration(locations = { "/simple-job-launcher-context.xml",
3.                                     "/jobs/skipSampleJob.xml" })
4. public class SkipSampleFunctionalTests { ... }
```

点对点的批处理任务测试

- 点对点的批处理任务测试" level="1">10.2 点对点的批处理任务测试

点对点的批处理任务测试" class="reference-link">10.2 点对点的批处理任务测试

点对点的测试被定义为测试从开始到结束完整的运行一个批处理任务。它允许一个测试设置一个测试条件，执行任务，并且验证最后的结果。

在下面的这个例子中，这个批处理任务从数据库读取数据并且写入一个平面文件中。这个测试方法首先要建立数据库与测试数据。它清空 `CUSTOMER` 表然后往里面插入10条新的记录。测试然后启动任务使用 `launchJob()` 方法，这个 `launchJob()` 方法是由 `JobLauncherTestUtils` 类提供的。还提供工具类

`launchJob(JobParameters)`，允许测试给特定的参数。`launchJob()`方法返回`JobExecution`对象有助于任务运行声明特定的信息，在这个实例中，这个测试证实任务的结束与状态“COMPLETED”。

```

1. @RunWith(SpringJUnit4ClassRunner.class)
2. @ContextConfiguration(locations = { "/simple-job-launcher-context.xml",
3.                                     "/jobs/skipSampleJob.xml" })
4. public class SkipSampleFunctionalTests {
5.
6.     @Autowired
7.     private JobLauncherTestUtils jobLauncherTestUtils;
8.
9.     private SimpleJdbcTemplate simpleJdbcTemplate;
10.
11.    @Autowired
12.    public void setDataSource(DataSource dataSource) {
13.        this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);
14.    }
15.
16.    @Test
17.    public void testJob() throws Exception {
18.        simpleJdbcTemplate.update("delete from CUSTOMER");
19.        for (int i = 1; i <= 10; i++) {
20.            simpleJdbcTemplate.update("insert into CUSTOMER values (?, 0, ?, 100000)",
21.                                     i, "customer" + i);
22.        }
23.
24.        JobExecution jobExecution = jobLauncherTestUtils.launchJob().getStatus();
25.

```



```
26.  
27.     Assert.assertEquals("COMPLETED", jobExecution.getExitStatus());  
28.     }  
29. }
```

测试各个步骤

- 测试各个步骤" level="1">10.3 测试各个步骤

测试各个步骤" class="reference-link">10.3 测试各个步骤

对于复杂的批处理任务，测试用例在点对点的测试方法中可能变成难于管理的。这些情况下，凭你们自己的力量用测试用例测试各个步骤更加有用。`AbstractJobTests` 类包含一个方法`launchStep`需要一个步骤名称和运行特定的步骤。这个方法允许更有针对性的测试通过这个步骤允许测试设置数据并验证其结果。

```
1. JobExecution jobExecution = jobLauncherTestUtils.launchStep("loadFileStep");
```

测试Step-Scoped组件

- 测试Step-Scoped组件" level="1">10.4 测试Step-Scoped组件

测试Step-Scoped组件" class="reference-link">10.4 测试Step-Scoped组件

时常组件在运行的时候需要配置你的步骤使用步骤并且迟绑定注入上下文从步骤或者是任务执行。这些是机警的测试像单独的组件除非你有一个办法设置上下文就像他们在一个步骤里执行。那是两个组件的目标在spring batch中: StepScopeTestExecutionListener 和 StepScopeTestUtils

这个监听是公开的在类级别中,它的工作是创建一个步骤为每个测试方法执行上下文。例如:

```

1. @ContextConfiguration
2. @TestExecutionListeners( { DependencyInjectionTestExecutionListener.class,
3.     StepScopeTestExecutionListener.class })
4. @RunWith(SpringJUnit4ClassRunner.class)
5. public class StepScopeTestExecutionListenerIntegrationTests {
6.
7.     // This component is defined step-scoped, so it cannot be injected unless
8.     // a step is active...
9.     @Autowired
10.    private ItemReader<String> reader;
11.
12.    public StepExecution getStepExecution() {
13.        StepExecution execution = MetaDataInstanceFactory.createStepExecution();
14.        execution.getExecutionContext().putString("input.data", "foo,bar,spam");
15.        return execution;
16.    }
17.
18.    @Test
19.    public void testReader() {
20.        // The reader is initialized and bound to the input data
21.        assertNotNull(reader.read());
22.    }
23.
24. }
```

有两个TestExecutionListeners, 一个来自普通的Spring测试框架和处理配置应用上下文的依赖注入, 注入reader和其他的Spring batch StepScopeTestExecutionListener. 它以一个stepExecution在测试用例中寻找一个工厂方法为动力, 并且使用它如同测试方法的上下文。在运行时在一个步骤内如果它运行是活动的。通过工厂方法的签名检测到工厂方法 (它仅仅有返回一个

StepExecution)。如果一个工厂方法不被提供，会有一个默认的StepExecution被创建

这个监听方法是方便的，如果你想在测试方法中执行持续的step scope。为了更灵活，但是更侵入性的方法你能使用StepScopeTestUtils。例如，去计算可用产品的数量在reader上面：

```
1. int count = StepScopeTestUtils.doInStepScope(stepExecution,
2.     new Callable<Integer>() {
3.         public Integer call() throws Exception {
4.
5.             int count = 0;
6.
7.             while (reader.read() != null) {
8.                 count++;
9.             }
10.            return count;
11.        }
12.    });
```

验证输出文件

- 验证输出文件" level="1">10.5 验证输出文件

验证输出文件" class="reference-link">10.5 验证输出文件

当一个批处理任务写入数据库的时候,很容易去查询数据去验证结果是否如预期一样。然而,如果批处理任务写入一个文件,验证输出量同样重要。Spring Batch 提供了一个类AssertFile使输出文件的验证变得容易。assertFileEquals方法带了两个文件对象(或者是两个资源对象)和断言,一行一行的,两个文件有相同的上下文。因此,它可能创建了一个文件,有预期的输出和对比之后返回的真实结果:

```
1. private static final String EXPECTED_FILE = "src/main/resources/data/input.txt";
2. private static final String OUTPUT_FILE = "target/test-outputs/output.txt";
3.
4. AssertFile.assertFileEquals(new FileSystemResource(EXPECTED_FILE),
5.                             new FileSystemResource(OUTPUT_FILE));
```

模拟域对象

- 模拟域对象" level="1">10.6 模拟域对象

模拟域对象" class="reference-link">10.6 模拟域对象

遇到了另一个常见的问题,同时为Spring Batch编写单元测试和集成测试组件是如何模拟域对象。一个很好的例子是StepExecutionListener,如下所示:

```
1. public class NoWorkFoundStepExecutionListener extends StepExecutionListenerSupport {
2.
3.     public ExitStatus afterStep(StepExecution stepExecution) {
4.         if (stepExecution.getReadCount() == 0) {
5.             throw new NoWorkFoundException("Step has not processed any items");
6.         }
7.         return stepExecution.getExitStatus();
8.     }
9. }
```

上面的监听是框架提供的,并且它检测到stepExecution的read count是为空的,因此它表示没有工作要做。虽然这个例子相当简单,当试图单元测试类的时候它解释了可能遇到的问题类型,实现接口验证Spring Batch 域对象。考虑到上面的侦听器的单元测试:

```
1. private NoWorkFoundStepExecutionListener tested = new NoWorkFoundStepExecutionListener();
2.
3. @Test
4. public void testAfterStep() {
5.     StepExecution stepExecution = new StepExecution("NoProcessingStep",
6.         new JobExecution(new JobInstance(1L, new JobParameters(),
7.             "NoProcessingJob")));
8.
9.     stepExecution.setReadCount(0);
10.
11.     try {
12.         tested.afterStep(stepExecution);
13.         fail();
14.     } catch (NoWorkFoundException e) {
15.         assertEquals("Step has not processed any items", e.getMessage());
16.     }
17. }
```

因为Spring Batch域模型遵循良好的面向对象原则，StepExecution需要一个JobExecution，JobExecution需要一个JobInstance和JobParameters为了创建一个有效的StepExecution。虽然这是很好的可靠的域模型，为了冗长的单元测试创建存根对象。为了解决这个问题，Spring Batch测试模块引入一个工厂来创建域对象:MetadataInstanceFactory，把这个给到工厂，更新后的单元测试可以更简洁：

```
1. private NoWorkFoundStepExecutionListener tested = new NoWorkFoundStepExecutionListener();
2.
3. @Test
4. public void testAfterStep() {
5.     StepExecution stepExecution = MetadataInstanceFactory.createStepExecution();
6.
7.     stepExecution.setReadCount(0);
8.
9.     try {
10.         tested.afterStep(stepExecution);
11.         fail();
12.     } catch (NoWorkFoundException e) {
13.         assertEquals("Step has not processed any items", e.getMessage());
14.     }
15. }
```

上面的方法为了创建一个简单的StepExecution，它仅仅是便利的方法中可用的工厂。完整的方法清单可以在Javadoc找到

通用批处理模式

- [11. 通用批处理模型](#)

11. 通用批处理模型

一些批处理任务可以使用spring batch现成的组件完全的组装。例如ItemReader和ItemWriter实现可配置覆盖范围广泛的场景，然而，对于大多数情况下，必须编写自定义代码。应用程序开发人员的主要API入口点是Tasklet, ItemReader ItemWriter和各种各样的监听器接口。最简单的批处理任务能够使用Spring BatchItemReader现成的输出，但通常情况下，自定义问题的处理和写作，需要开发人员实现一个ItemWriter或ItemProcessor。

在这里，我们提供了一个通用模式自定义业务逻辑的一些例子。这些例子的主要特性是监听器接口，应该注意的是，如果合适，一个ItemReader或ItemWriter也可以实现一个监听器接口。

日志项处理和失败

- 11.1 日志项处理和失败

11.1 日志项处理和失败

一个常见的用例是需要在一个步骤中特殊处理错误, chunk-oriented步骤(从创建工厂bean的这个步骤)允许用户实现一个简单的ItemReadListener用例,用来监听读入错误, 和一个ItemWriteListener,用来监听写出错误. 下面的代码片段说明一个监听器监听失败日志的读写:

```
1. >public class ItemFailureLoggerListener extends ItemListenerSupport {
2.
3.     private static Log logger = LoggerFactory.getLog("item.error");
4.
5.     public void onReadError(Exception ex) {
6.         logger.error("Encountered error on read", e);
7.     }
8.
9.     public void onWriteError(Exception ex, Object item) {
10.        logger.error("Encountered error on write", ex);
11.    }
12.
13. }
```

在实现此监听器必须注册步骤:

```
1. <step id="simpleStep">
2.     ...
3.     <listeners>
4.         <listener>
5.             &lt;bean class="org.example...ItemFailureLoggerListener"/>
6.         </listener>
7.     </listeners>
8. </step>
```

记住, 如果你的监听器在任何一个onError()方法中, 它将在一个事务中回滚. 如果在一个onError()方法中需要使用事务性的资源(比如数据库), 可以考虑添加一个声明式事务方法(有关详细信息, 请参阅Spring核心参考指南), 并给予其传播属性REQUIRES_NEW值。

业务原因手工停止任务

- 11.2 业务原因手工停止任务

11.2 业务原因手工停止任务

Spring Batch通过JobLauncher接口提供一个stop()方法,但是这实际上是给维护人员用的,而不是程序员.有时有更方便和更有意义的阻止任务中的业务逻辑执行.

最简单的做法是抛出一个RuntimeException(不是无限的重试或跳过).例如,可以使用一个自定义异常类型,如下示例:

```
public class PoisonPillItemWriter implements ItemWriter {
```

```
1. public void write(T item) throws Exception {
2.     if (isPoisonPill(item)) {
3.         throw new PoisonPillException("Posion pill detected: " + item);
4.     }
5. }
6.
7. }
```

ItemReader中阻止一个步骤执行的另一种简单的方法是简单地返回null:

```
public class EarlyCompletionItemReader implements ItemReader {
```

```
1. private ItemReader<T> delegate;
2.
3. public void setDelegate(ItemReader<T> delegate) { ... }
4.
5. public T read() throws Exception {
6.     T item = delegate.read();
7.     if (isEndItem(item)) {
8.         return null; // end the step here
9.     }
10.    return item;
11. }
12.
13. }
```

前面的例子实际上依赖于这样一个事实,当item处理是null时,有一个默认的实现CompletionPolicy的策略来实现批处理,更复杂完善是策略可以通过SimpleStepFactoryBean实现和注入Step:

```

1. <step id="simpleStep">
2. <tasklet>
3.     <chunk reader="reader" writer="writer" commit-interval="10"
4.         chunk-completion-policy="completionPolicy"/>
5. </tasklet>
6. </step>
7.
8. <bean id="completionPolicy" class="org.example...SpecialCompletionPolicy"/>

```

另一种方法是在框架启动处理item的时候检查step设置一个标志StepExecution.实现这个替代方案,我们需要使用当前的StepExecution,这可以通过实现一个StepListener 和注册step:

```

1. public class CustomItemWriter extends ItemListenerSupport implements StepListener    {
2.
3.     private StepExecution stepExecution;
4.
5.     public void beforeStep(StepExecution stepExecution) {
6.         this.stepExecution = stepExecution;
7.     }
8.
9.     public void afterRead(Object item) {
10.        if (isPoisonPill(item)) {
11.            stepExecution.setTerminateOnly(true);
12.        }
13.    }
14.
15. }

```

在这里flag设置默认的行为是step抛出一个JobInterruptedException.这可以通过StepInterruptionPolicy控制,但唯一的选择是抛出一个exception,所以这个job总是一个异常的结果.

添加一个Footer记录

- 11.3 添加一个Footer记录
 - 11.3.1 写一个简单Footer

11.3 添加一个Footer记录

经常写文本文件时,在所有处理都已经完成,一个“footer”记录必须附加到文件的末尾.这也可以通过使用由Spring Batch提供的FlatFileFooterCallback接口,FlatFileItemWriter的FlatFileFooterCallback(和与之对应的FlatFileHeaderCallback)是可选的属性.

```
1. <bean id="itemWriter" class="org.spr...FlatFileItemWriter">
2.     <property name="resource" ref="outputResource" />
3.     <property name="lineAggregator" ref="lineAggregator"/>
4.     <property name="headerCallback" ref="headerCallback" />
5.     <property name="footerCallback" ref="footerCallback" />
6. </bean>
```

footer回调接口非常简单,它只有一个方法调用.

```
1. public interface FlatFileFooterCallback {
2.
3.     void writeFooter(Writer writer) throws IOException;
4.
5. }
```

11.3.1 写一个简单Footer

是一个非常常见的需求涉及到footer记录,在输出过程中总计信息然后把这信息附加到文件末尾.这footer作为文件的总结或提供了一个校验和.

例如,如果一个批处理作业是flat文件写贸易记录,所有交易的totalAmount需要放置在footer,然后可以使用followingItemWriter实现:

```
1. public class TradeItemWriter implements ItemWriter<Trade>,
2.                                     FlatFileFooterCallback {
3.
4.     private ItemWriter<Trade> delegate;
5.
6.     private BigDecimal totalAmount = BigDecimal.ZERO;
7.
8.     public void write(List<? extends Trade> items) {
```

```

9.     BigDecimal chunkTotal = BigDecimal.ZERO;
10.    for (Trade trade : items) {
11.        chunkTotal = chunkTotal.add(trade.getAmount());
12.    }
13.
14.    delegate.write(items);
15.
16.    // After successfully writing all items
17.    totalAmount = totalAmount.add(chunkTotal);
18. }
19.
20. public void writeFooter(Writer writer) throws IOException {
21.     writer.write("Total Amount Processed: " + totalAmount);
22. }
23.
24. public void setDelegate(ItemWriter delegate) {...}
25.
26. }

```

TradeItemWriter存储的totalAmount值随着每笔交易Amount写出而增长,在交易处理完成后,框架将调用writeFooter,将总金额加入到文件中.注意,写方法使用一个临时变量,chunkTotalAmount,存储交易总数到chunk.这样做是为了确保如果发生跳过写出的方法,totalAmount将保持不变.只有在写出方法结束时,不抛出异常,我们将更新totalAmount.

为了writeFooter被调用,TradeItemWriter(因为实现了FlatFileFooterCallback接口)必须以footerCallback为属性名注入到FlatFileItemWriter 中

```

1. <bean id="tradeItemWriter" class="..TradeItemWriter">
2.     <property name="delegate" ref="flatFileItemWriter" />
3. </bean>
4.
5. <bean id="flatFileItemWriter" class="org.spr...FlatFileItemWriter">
6.     <property name="resource" ref="outputResource" />
7.     <property name="lineAggregator" ref="lineAggregator"/>
8.     <property name="footerCallback" ref="tradeItemWriter" />
9. </bean>

```

如果step是不可重新开始的,TradeItemWriter只会正常运行.这是因为类是有状态(因为它存储了totalAmount),但totalAmount不是持久化到数据库中,因此,它不能被重启.为了使这个类可重新开始,ItemStream接口应该实现的open和update方法.

```

1. public void open(ExecutionContext executionContext) {
2.     if (executionContext.containsKey("total.amount") {
3.         totalAmount = (BigDecimal) executionContext.get("total.amount");
4.     }
5. }

```

```
6.  
7. public void update(ExecutionContext executionContext) {  
8.     executionContext.put("total.amount", totalAmount);  
9. }
```

ExecutionContext持久化到数据库之前, update方法将存储totalAmount最新的值.open方法根据ExecutionContext中记录的处理起始点恢复totalAmount.在Step 中断执行之前,允许TradeItemWriter 重新启动

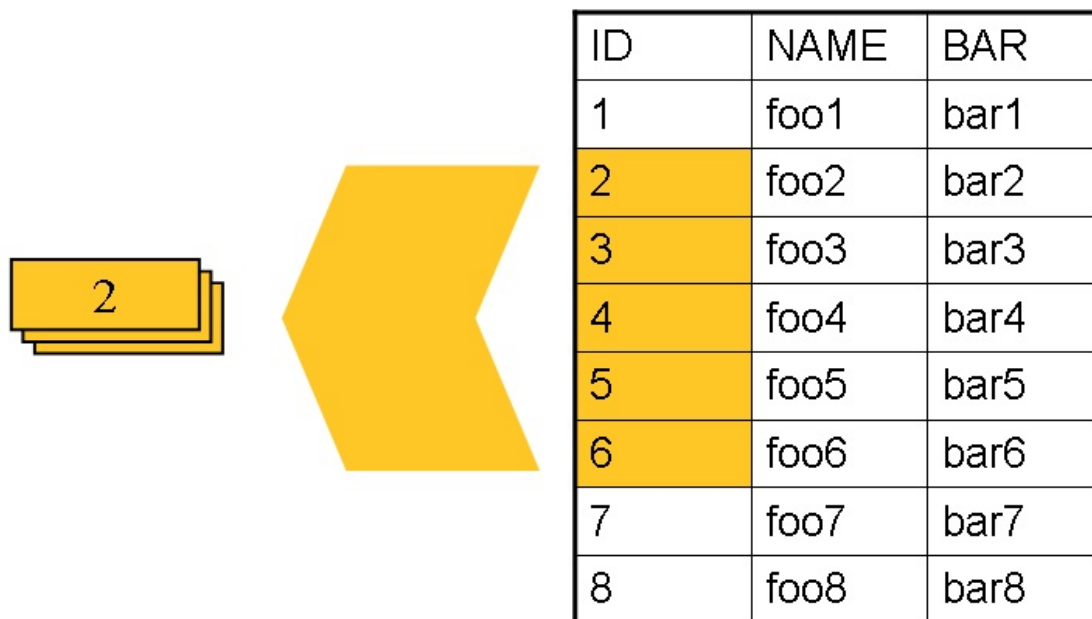
基于ItemReaders的driving query

- 11.4 基于ItemReaders的driving query

11.4 基于ItemReaders的driving query

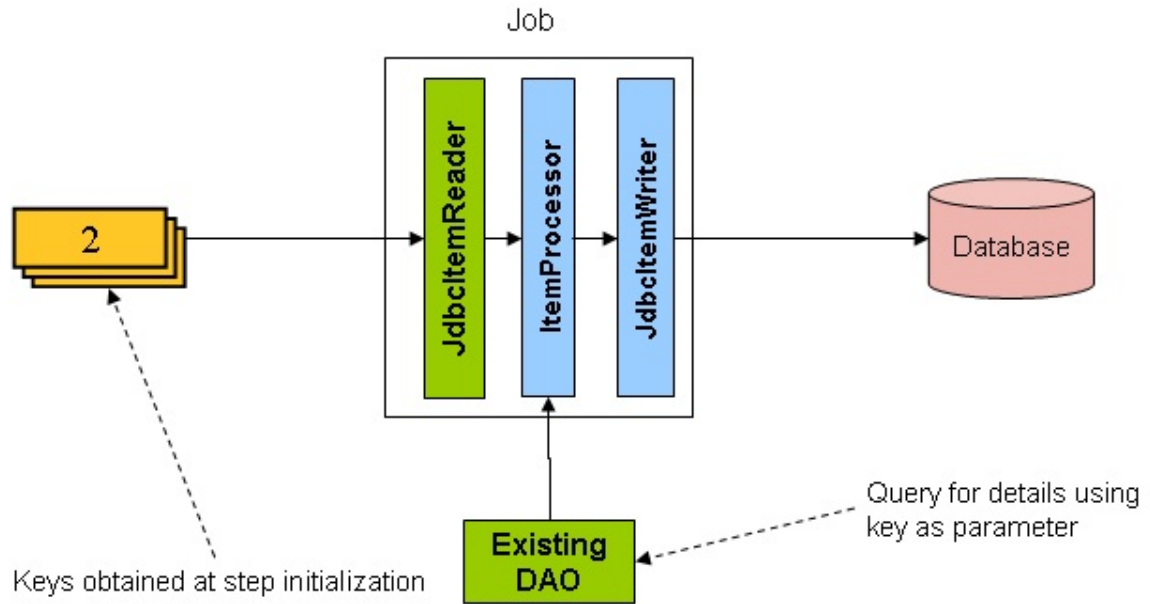
在readers 和writers章节中对数据库分页进行了讨论,很多数据库厂商,比如DB2,如果读表也需要使用的在线应用程序的其他部分,悲观锁策略,可能会导致问题.此外,打开游标在超大数据集可能导致某些供应商的问题.因此,许多项目更喜欢使用一个'Driving Query'的方式读入数据.这种方法是
通过遍历keys,而不是整个对象,因此需要返回对象,如下示例所示:

```
Select ID from F00  
where id > 1 and id < 7
```



ID	NAME	BAR
1	foo1	bar1
2	foo2	bar2
3	foo3	bar3
4	foo4	bar4
5	foo5	bar5
6	foo6	bar6
7	foo7	bar7
8	foo8	bar8

如您所见,这个例子使用一样的“F00”表中使用基于指针的例子.然而,不是选择了整个行,只选择了ID的SQL语句.因此,返回一个整数,而不是返回F00对象.这个数字可以用来查询的'details',这是一个完整的Foo对象:



ItemProcessor应该用转换的key从driving query得到一个完整的'Foo'对象, 现有的Dao可以用查询完整的基于key的对象

多行记录

- 11.5 多行记录

11.5 多行记录

虽然通常的flat文件, 每个记录限制是一行, 但是一个文件可能与多种格式多行记录, 这是很常见的. 以下摘录文件说明了这一点:

```
1. HEA;0013100345;2007-02-15
2. NCU;Smith;Peter;;T;20014539;F
3. BAD;;;Oak Street 31/A;;Small Town;00235;IL;US
4. FOT;2;2;267.34
```

之间的所有行从'HEA'开始和从'FOT'开始被认为是一个记录. 有一些注意事项, 必须正确地处理这种情况:

不是一次读取一条记录, 而是ItemReader必须读取多行的每一行记录作为一个分组. 这样它就可以被完整的传递到ItemWriter.

1. 每一行可能需要标记不同的类型.

因为单行记录跨越多行, 我们可能不知道有多少行, 所以ItemReader必须总是小心的读一个完整的记录. 为了做到这一点, 一个自定义ItemReader, 应该实现一个FlatFileItemReader的包装器

```
1. <bean id="itemReader" class="org.spr...MultiLineTradeItemReader">
2.   <property name="delegate">
3.     <bean class="org.springframework.batch.item.file.FlatFileItemReader">
4.       <property name="resource" value="data/iosample/input/multiLine.txt" />
5.       <property name="lineMapper">
6.         <bean class="org.spr...DefaultLineMapper">
7.           <property name="lineTokenizer" ref="orderFileTokenizer"/>
8.           <property name="fieldSetMapper">
9.             <bean class="org.spr...PassThroughFieldSetMapper" />
10.          </property>
11.        </bean>
12.      </property>
13.    </bean>
14.  </property>
15. </bean>
```

确保正确标记的每一行, 特别是重要的固定长度的输入, 可以将PatternMatchingCompositeLineTokenizer委托于FlatFileItemReader. 见章节

“Multiple Record Types within a Single File”中的更多的细节.这个委托reader,会将FieldSet传递到PassThroughFieldSetMapper再返回每一行到ItemReader 包装器中.

```

1. <bean id="orderFileTokenizer" class="org.spr...PatternMatchingCompositeLineTokenizer">
2.     <property name="tokenizers">
3.         <map>
4.             <entry key="HEA*" value-ref="headerRecordTokenizer" />
5.             <entry key="FOT*" value-ref="footerRecordTokenizer" />
6.             <entry key="NCU*" value-ref="customerLineTokenizer" />
7.             <entry key="BAD*" value-ref="billingAddressLineTokenizer" />
8.         </map>
9.     </property>
10. </bean>

```

这个包装器必须能够识别结束的记录,所以它可以不断调用delegate的read()方法直到结束.对于读取的每一行,该wrapper应该绑定返回的item,一旦读到footer结束,item应该传递给ItemProcessor和ItemWriter返回.

```
private FlatFileItemReader
```

```
delegate;
```

```

1. public Trade read() throws Exception {
2.     Trade t = null;
3.
4.     for (FieldSet line = null; (line = this.delegate.read()) != null;) {
5.         String prefix = line.readString(0);
6.         if (prefix.equals("HEA")) {
7.             t = new Trade(); // Record must start with header
8.         }
9.         else if (prefix.equals("NCU")) {
10.            Assert.notNull(t, "No header was found.");
11.            t.setLast(line.readString(1));
12.            t.setFirst(line.readString(2));
13.            ...
14.        }
15.        else if (prefix.equals("BAD")) {
16.            Assert.notNull(t, "No header was found.");
17.            t.setCity(line.readString(4));
18.            t.setState(line.readString(6));
19.            ...
20.        }
21.        else if (prefix.equals("FOT")) {
22.            return t; // Record must end with footer
23.        }
24.    }
25.    Assert.isNull(t, "No 'END' was found.");

```

```
26.     return null;  
27. }
```

执行系统命令

- 11.6 执行系统命令

11.6 执行系统命令

许多批处理作业可能需要一个外部命令调用内部的批处理作业. 这样一个过程可以分开调度, 但常见的元数据对运行的优势将会丢失. 此外, multi-step 作业需要分割成多个作业.

因此通常的 spring batch提供一个tasklet实现调用系统命令:

```
1. <bean class="org.springframework.batch.core.step.tasklet.SystemCommandTasklet">
2.     <property name="command" value="echo hello" />
3.     <!-- 5 second timeout for the command to complete -->
4.     <property name="timeout" value="5000" />
5. </bean>
```

当没有找到输入时Step处理完成

- 11.7 当没有找到输入时Step处理完成

11.7 当没有找到输入时Step处理完成

在许多批处理场景中,发现数据库或者文件中没有对应的行.Step只是认为没有找到工作和读入0行items.所有的ItemReader实现提供了开箱即用默认的Spring Batch方法.如果没有输出,可能会导致一些混乱,即使输入是当前的(如果一个文件通常是错误的,等等),为此,元数据本身应该检查以确定多少工作被框架发现和处理.然而,如果发现没有输入被认为只是例外呢?在这种情况下,以编程方式检查元数据有多少items没有被处理及其导致失败的原因,是最好的解决方案.这是一个常见的用例,一个监听器提供此功能:

```
public class NoWorkFoundStepExecutionListener extends
StepExecutionListenerSupport {
```

```
1.     public ExitStatus afterStep(StepExecution stepExecution) {
2.         if (stepExecution.getReadCount() == 0) {
3.             return ExitStatus.FAILED;
4.         }
5.         return null;
6.     }
7.
8.
9. }
```

以上所述StepExecutionListener检查readCount属性,StepExecution 在'afterStep'阶段确定items是否被读取到,如果这样的话,一个退出代码返回FAILED,表明这个Step失败.否则,返回null,这不会影响step状态

将数据传递给Future Steps

- 11.8 将数据传递给Future Steps

11.8 将数据传递给Future Steps

将一个步骤传递给另外一个步骤,这通常很有用.这可以通过使用ExecutionContext实现.值得注意的是,有两个ExecutionContexts:一个step层面,一个在job级别.Step级别ExecutionContext生命周期和一个step一样长,然而 job级别ExecutionContext贯穿整个job.另一方面,每次更新Step级别ExecutionContext,该Step提交一个chunk,然而Job级别ExecutionContext更新只会存在最后一个 Step.

这种分离的结果是,step的执行过程中所有的数据必须放置在step级别ExecutionContext的执行.当step是on-going状态时,将确保数据存储是正确的.如果数据存储(job级别的ExecutionContext中,那么它将不是持久化状态的,在Step执行期间如果Step失败,则数据会丢失.

```

1. public class SavingItemWriter implements ItemWriter<Object> {
2.     private StepExecution stepExecution;
3.
4.     public void write(List<? extends Object> items) throws Exception {
5.         // ...
6.
7.         ExecutionContext stepContext = this.stepExecution.getExecutionContext();
8.         stepContext.put("someKey", someObject);
9.     }
10.
11.    @BeforeStep
12.    public void saveStepExecution(StepExecution stepExecution) {
13.        this.stepExecution = stepExecution;
14.    }
15. }

```

使数据可用于future Steps,step完成之后Job级别ExecutionContext将有'promoted',Spring Batch为此提供了ExecutionContextPromotionListener.该监听器必须配置和数据相关的keys到ExecutionContext,它必须被提升.可选地,也可以配置的退出代码模式("COMPLETED" 是默认的),与所有监听一样,它必须在step中注册

```

1. <job id="job1">
2.     <step id="step1">
3.         <tasklet>
4.             <chunk reader="reader" writer="savingWriter" commit-interval="10"/>
5.         </tasklet>

```

```
6.     <listeners>
7.         <listener ref="promotionListener"/>
8.     </listeners>
9. </step>
10.
11. <step id="step2">
12.     ...
13. </step>
14. </job>
15.
16. <beans:bean id="promotionListener" class="org.spr....ExecutionContextPromotionListener">
17.     <beans:property name="keys" value="someKey"/>
18. </beans:bean>
```

最后,保存的值必须从job ExecutionContext重新获得:

```
1.     public class RetrievingItemWriter implements ItemWriter<Object> {
2.         private Object someObject;
3.
4.         public void write(List<? extends Object> items) throws Exception {
5.             // ...
6.         }
7.
8.         @BeforeStep
9.         public void retrieveInterstepData(StepExecution stepExecution) {
10.             JobExecution jobExecution = stepExecution.getJobExecution();
11.             ExecutionContext jobContext = jobExecution.getExecutionContext();
12.             this.someObject = jobContext.get("someKey");
13.         }
14.     }
```


JSR352支持

- [12. JSR-352 支持](#)

12. JSR-352 支持

Spring Batch 3.0 对 JSR-352 提供完整的支持。本节并不讲述这个规范，而是讲解如何将 JSR-352 的相关概念应用于Spring Batch。关于JSR-352 的更多信息可以参考 JCP 网站：<https://jcp.org/en/jsr/detail?id=352>

General Notes

- [12.1 General Notes Spring Batch and JSR-352](#)

12.1 General Notes Spring Batch and JSR-352

Spring Batch 和 JSR-352 在结构上是一致的。

两者的作业(jobs) 都由步骤(step)构成。

都有这些组件: reader, processor, writer, 以及 listener。

但两者的交互略有不同。例如: Spring Batch 中的

`org.springframework.batch.core.SkipListener#onSkipInWrite(S item, Throwable t)` 接收2个参数: 忽略的 item 以及 引起 skip的 `Exception`。而 JSR-352 规范中同样的方法, (`javax.batch.api.chunk.listener.SkipWriteListener#onSkipWriteItem(List<Object> items, Exception ex)`) 虽然也接受2个参数。但第一个参数是 `List`, 是当前 chunk 中的所有 item, 第二个参数是引起 skip的 `Exception`。因为有一些差别,所以需要注意在 Spring Batch 中执行 job 有两种途径: 一是常规的 Spring Batch job, 二是基于 JSR-352 的 job。使用JSR-352's JSL 配置的 Spring Batch 组件 (readers, writers, etc), 将通过 `JsrJobOperator` 来执行, 他们会按照 JSR-352 规定的行为来执行。还需要注意的是采用 JSR-352 规范开发的 batch 组件, 在常规的 Spring Batch job 中不能正常运行。

Setup

- [12.2 Setup](#)
 - [提示](#)

12.2 Setup

JSR-352 要用到一个很简单的 `path` 来执行批处理作业。下面的代码就是执行一个批处理作业所需要的一切：

```
1. JobOperator operator = BatchRuntime.getJobOperator();
2. jobOperator.start("myJob", new Properties());
```

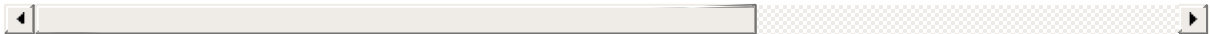
虽然对于开发人员来说很方便，但其中的坑可能埋藏在细节中。Spring Batch 在启动时会加载一些基础类，开发者可能需要重写其中的某些部分。在首次调用 “`BatchRuntime.getJobOperator()`” 时会加载下面这些对象：

Bean Name	Default Configuration
<code>dataSource</code>	配置的 Apache DBCP <code>BasicDataSource</code> 数据库连接池对象。
<code>transactionManager</code>	<code>org.springframework.jdbc.datasource.DataSourceTransactionManager</code>
A Datasource initializer	
<code>jobRepository</code>	基于 JDBC 的 <code>SimpleJobRepository</code> 。
<code>jobLauncher</code>	<code>org.springframework.batch.core.launch.support.SimpleJobLauncher</code>
<code>batchJobOperator</code>	<code>org.springframework.batch.core.launch.support.SimpleJobOperator</code>

jobExplorer	<code>org.springframework.batch.core.explore.support.JobExplorerFa</code>
jobParametersConverter	<code>org.springframework.batch.core.jsr.JsrJobParametersConverter</code>
jobRegistry	<code>org.springframework.batch.core.configuration.support.MapJobF</code>
placeholderProperties	<code>org.springframework.beans.factory.config.PropertyPlaceholder</code>

提示

对基于 JSR-352的 *job*来说,上面的这些 *bean* 都是必须具备的。当然,开发者可以根据需要重载某些类,以提供自定义的功能。



依赖注入

- [12.3 依赖注入](#)

12.3 依赖注入

JSR-352主要基于Spring Batch的编程模型。因此,尽管没有明确需要正式的依赖注入实现,但还是推荐使用 DI 的方式。Spring Batch支持 JSR-352 中定义的三种加载组件方式:

- 实现专用加载器 - Spring Batch 构建于 Spring 之上, 支持在 JSR-352 批处理作业中使用依赖注入。
- 归档文件加载器 - JSR-352 定义了一个现有的 `batch.xml` 文件, 对逻辑name和实际的类名之间做了映射。这个文件必须放在 `/META-INF/` 目录中才会生效。
- 线程上下文类加载器(Thread Context Class Loader) - JSR-352 允许在 JSL 中通过内联的方式指定全限定类名来配置批处理组件。Spring Batch 也支持 JSR-352 的这种配置作业的方式。

要在 JSR-352的作业中使用Spring依赖性注入, 需要在 Spring application context 中将组件配置为 bean。bean 定义以后, 作业就可以像 `batch.xml` 中定义的部分 一样引用他们。

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.       xsi:schemaLocation="http://www.springframework.org/schema/beans
5.                           http://www.springframework.org/schema/beans/spring-beans.xsd
6.                           http://xmlns.jcp.org/xml/ns/javaee
7.                           http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd">
8.
9.   <!-- javax.batch.api.Batchlet implementation -->
10.  <bean id="fooBatchlet" class="io.spring.FooBatchlet">
11.    <property name="prop" value="bar"/>
12.  </bean>
13.
14.  <!-- Job is defined using the JSL schema provided in JSR-352 -->
15.  <job id="fooJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
16.    <step id="step1">
17.      <batchlet ref="fooBatchlet"/>
18.    </step>
19.  </job>
20. </beans>

```

Spring 的 contexts (imports, etc) 组装在 JSR-352 中和其他地方的使用基本一样。唯一

的区别是上下文定义的入口点在 `/META-INF/batch-jobs/` 。

要使用线程上下文类加载器的方式，只需要提供全限定类名作为 `ref` 即可。要注意的是，在使用这种方式 或者 `batch.xml` 时，引用的类需要有一个无参构造函数来创建bean。

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <job id="fooJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
3.     <step id="step1" >
4.         <batchlet ref="io.spring.FooBatchlet" />
5.     </step>
6. </job>
```

Batch Properties

- [12.4 Batch Properties](#)
 - [12.4.1 Property 支持](#)
 - [12.4.2 `@BatchProperty` 注解](#)
 - [12.4.3 Property 替换](#)
- `{jobParameters['unresolving.prop']}??:#{systemProperties['file.separator']}`

12.4 Batch Properties

12.4.1 Property 支持

JSR-352 允许在 Job, Step 和其他 batch 组件级别配置 properties 信息, 在 JSL 文件中通过如下的方式配置 Batch properties:

```

1. <properties>
2.     <property name="propertyName1" value="propertyValue1"/>
3.     <property name="propertyName2" value="propertyValue2"/>
4. </properties>

```

Properties 也可以配置在所有批处理组件中.

12.4.2 `@BatchProperty` 注解

在批处理组件中 可以通过组件的方式来引用 Properties, 分别是 `@BatchProperty` 和 `@Inject` (两个注解需同时要指定). 根据 JSR-352 的定义, property 属性域必须是 String 类型. 类型转换取决于代码实现(Any type conversion is up to the implementing developer to perform).

可以像前面所描述的那样定义属性, 然后通过类似下面 `javax.batch.api.chunk.ItemReader` 组件的这种方式来使用:

```

1. public class MyItemReader extends AbstractItemReader {
2.     @Inject
3.     @BatchProperty
4.     private String propertyName1;
5.
6.     ...
7. }

```

属性域 “propertyName1” 的值会被设置为 “propertyValue1”

12.4.3 Property 替换

Property 替换是由操作符和简单条件表达式实现的。常用的方式是 `{operator['key']}`。

支持以下操作符：

- `jobParameters` - 启动/重启 Job 时的参数。
- `jobProperties` - JSL中配置在 job 级别的属性。
- `systemProperties` - 命名的系统属性。
- `partitionPlan` - 在分区 step 中访问分区计划的命名属性。

```
{jobParameters['unresolving.prop']}
?:#
{systemProperties['file.separator']}
}
```

左边是希望设置的值, 右边是默认值。在这里, 因为解析不到 `#` `{jobParameters['unresolving.prop']}` 的值, 所以生效的会是 system property `file.separator`。如果一个都没解析到, 则会返回空字符串(empty String)。可以使用多个条件, 使用英文分号 ‘;’ 分隔。

处理模型

- 12.5 处理模型(Processing Models)
 - 12.5.1 基于 Item 的处理
 - 12.5.2 自定义检查点(checkpointing)

12.5 处理模型(Processing Models)

JSR-352 和 Spring Batch 一样支持两种基本处理模型:

- 基于 Item 的处理 - 通过 `javax.batch.api.chunk.ItemReader`, 以及可选的 `javax.batch.api.chunk.ItemProcessor`, 加上 `javax.batch.api.chunk.ItemWriter` .
- 基于 Task 的处理 - 使用 `javax.batch.api.Batchlet` 的实现. 当前此模型和基于 `org.springframework.batch.core.step.tasklet.Tasklet` 的处理是一样的.

12.5.1 基于 Item 的处理

Item based processing in this context is a chunk size being set by the number of items read by an ItemReader. To configure a step this way, specify the item-count (which defaults to 10) and optionally configure the checkpoint-policy as item (this is the default).

这里说的 基于 Item 的处理是指定由ItemReader读取的 item 数量。如果以这种方式配置step, 指定item-count(默认值为10),还可以配置可选参数 checkpoint-policy 为 `item` (也是默认值)。

```

1. ...
2. <step id="step1">
3.     <chunk checkpoint-policy="item" item-count="3">
4.         <reader ref="fooReader"/>
5.         <processor ref="fooProcessor"/>
6.         <writer ref="fooWriter"/>
7.     </chunk>
8. </step>
9. ...

```

If item based checkpointing is chosen, an additional attribute time-limit is supported. This sets a time limit for how long the number of items specified has to be processed. If the timeout is reached, the chunk will complete with however many items have been read by then regardless of what the item-count is configured to be.

如果选择了基于item的检查点，则可以使用附加属性 `time-limit`。时限设置一定时间内要处理的 `item` 数量。如果到达时限，不管完成多少 `item`，`chunk`都会进入完成状态，然后不管 `item-count` 配置。

12.5.2 自定义检查点(checkpointing)

JSR-352 calls the process around the commit interval within a step “checkpointing”. Item based checkpointing is one approach as mentioned above. However, this will not be robust enough in many cases. Because of this, the spec allows for the implementation of a custom checkpointing algorithm by implementing the `javax.batch.api.chunk.CheckpointAlgorithm` interface. This functionality is functionally the same as Spring Batch’s custom completion policy. To use an implementation of `CheckpointAlgorithm`, configure your step with the custom `checkpoint-policy` as shown below where `fooCheckpointner` refers to an implementation of `CheckpointAlgorithm`.

JSR-352 在一个step的“检查点”中调用提交间隔流程。基于 `Item` 的检查点是上面提到的一种方法。然而，在许多时候这不够健壮。因此，规范允许自定义的

`javax.batch.api.chunk.CheckpointAlgorithm` 实现。`CheckpointAlgorithm`接口。这也是 Spring Batch 的定制完成策略。使用 `CheckpointAlgorithm` 的实现，配置自定义的 `checkpoint-policy`，如下所示，其中 `fooCheckpointner` 指向 `CheckpointAlgorithm` 的实现。

```
1. ...
2. <step id="step1">
3.     <chunk checkpoint-policy="custom">
4.         <checkpoint-algorithm ref="fooCheckpointner"/>
5.         <reader ref="fooReader"/>
6.         <processor ref="fooProcessor"/>
7.         <writer ref="fooWriter"/>
8.     </chunk>
9. </step>
10. ...
```

Running a job

- [12.6 Running a job](#)
 - [\[Note\] Note](#)

12.6 Running a job

执行基于 JSR-352 作业的入口是 `javax.batch.operations.JobOperator`。Spring Batch 提供了这个接口的实现（`org.springframework.batch.core.jsr.launch.JsrJobOperator`）。通过 `javax.batch.runtime.BatchRuntime` 来实现加载。使用方法如下所示：

```
1. JobOperator jobOperator = BatchRuntime.getJobOperator();
2. long jobExecutionId = jobOperator.start("fooJob", new Properties());
```

这段代码的含义如下：

- 引导基本的 `ApplicationContext` - 为了提供批处理功能，框架需要用到一些基础组件。在每个 JVM 中都会执行一次。这些组件的引导和 `@EnableBatchProcessing` 很相似。详细的信息可以参考 `JsrJobOperator` 的 javadoc 文档。
- 加载所需的 `ApplicationContext` - 前面的例子中，框架回去查找 `/META-INF/batch-jobs` 中一个叫做 `fooJob.xml` 的文件，并加载前面提到的 `shared context` 的一个子 `context`。
- 运行 `job` - 在上下文 `context` 中定义的 `job` 会被异步执行。但会先返回 `JobExecution` 的 `id`。

[Note] Note

所有基于 JSR-352 的批处理作业都是异步执行的。

调用 `SimpleJobOperator` 的 `JobOperator#start` 方法时，Spring Batch 会判断这次调用是初始运行还是重试运行。使用基于 JSR-352 的 `JobOperator#start(String jobXMLName, Properties jobParameters)`，框架每次都会创建一个新的 `JobInstance`（JSR-352 的作业参数不是唯一的 `[non-identifying]`）。若重启作业，需要调用 `JobOperator#restart(long executionId, Properties restartParameters)`。

Contexts

- [12.7 Contexts](#)
 - [\[提示\] @Autowired for JSR-352 contexts](#)

12.7 Contexts

JSR-352 定义了两个上下文对象：`javax.batch.runtime.context.JobContext` 和 `javax.batch.runtime.context.StepContext`，分别用来处理 job/step 的元数据(meta-data)信息。JobContext 和 StepContext对象在所有的 step 级别的组件中都是可用的(如 Batchlet, ItemReader, 等)；当然，JobContext对象在 job 级别的组件中也是可见的(比如 JobListener)。

要获取当前 scope的 JobContext 和 StepContext 对象引用,请使用 `@Inject` 注解:

1. `@Inject`
2. `JobContext jobContext;`

[提示] @Autowired for JSR-352 contexts

在这一类的上下文注入中, 不支持Spring的 `@Autowired` 注解.

在 Spring Batch 中, JobContext 和 StepContext 封装了相应的执行对象 (分别是 JobExecution 以及 StepExecution). 在 Spring Batch 的 StepExecution#executionContext 中通过 `StepContext#persistent#setPersistentUserData` (序列化)保存 Data.

Step Flow

- [12.8 Step Flow](#)

12.8 Step Flow

基于JSR-352的作业，step的流程和Spring Batch基本上是一样的。但也有一些细微的差别：

- Decision 也是 step — 在常规 Spring Batch 作业中，decision是一个状态，在整个 step 中没有独立的 StepExecution 或任何其他权利/职责，。然而，JSR-352，decision 就是一个 step，所以表现得也和其他step一样。（在事务中时，它会得到 StepExecution 等）。这意味着他们在重启后也会当成一个普通的 step来处理。
- next 属性与 step 切换 — 在常规作业中，允许两者在同一个step中一起出现。JSR-352 允许在同一个 step 中使用他们。根据 next 属性来决定使用哪一个。
- Transition 元素排序 — 在标准的Spring Batch作业中，transition 元素排序是从最具体的到最模糊的，计算和评估也是采用这种顺序。但是，JSR-352 的 job 顺序是根据他们在XML中定义的顺序来决定的。

扩展 JSR-352 批处理作业

- [12.9 扩展 JSR-352 批处理作业](#)
 - [12.9.1 分区\(Partitioning\)](#)
 - [\[Note\] 提示](#)

12.9 扩展 JSR-352 批处理作业

Spring Batch 的 job 有4种扩展方式（后两种方式支持使用多个JVM来执行）：

- Split - 并行执行多个 step.
- Multiple threads - 通过多线程执行单个 step.
- Partitioning - 将数据切分后并行处理（主从, master/slave）.
- Remote Chunking - 远程执行逻辑处理块.

JSR-352 提供了两种扩展批处理的方式。这两种方式都只支持在单个 JVM 中运行：

- Split - 和 Spring Batch 一样
- Partitioning - 在概念上与Spring Batch一样,但实现略有不同.

12.9.1 分区(Partitioning)

从概念上讲，JSR-352的分区是和Spring Batch一样的。传递 元数据(meta-data)给每个 slave 以确定需要处理的输入，在执行完成后返回结果信息给 master。但他们之间有一些重要的区别：

- Partitioned Batchlet - 这会在多个线程中运行多个配置的 Batchlet 实例。每个实例都有自己的 properties，一般通过 JSL 或者 PartitionPlan 设置
- PartitionPlan - 在 Spring Batch 中，每个分区都有一个 ExecutionContext。但在 JSR-352 中只提供单个 `javax.batch.api.partition.PartitionPlan`，附带一个 Properties 数组，存放每个 partition 的 meta-data 信息。
- PartitionMapper - JSR-352 提供2种方式来生成分区的 meta-data。一是通过 JSL (partition 属性)。二是通过 `javax.batch.api.partition.PartitionMapper` 接口的一个实例。在功能上，这个接口类似于 Spring Batch 的 `org.springframework.batch.core.partition.support.Partitioner` 接口(在其中允许手工生成分区的 meta-data 信息)。
- StepExecutions - 在 Spring Batch 中，分区 step 使用主从的方式运行。在 JSR-352 中,也使用相同的配置。但是 slave steps 不能获得正式的 StepExecutions。因此，调用 `JsrJobOperator#getStepExecutions(long jobExecutionId)` 方法智慧返回 master 的 StepExecution。

[Note] 提示

子 `StepExecution` 仍然存在于 `job repository` 中,可以通过 `JobExplorer` 和 `Spring Batch Admin` 来查看/使用.

- 补偿逻辑(Compensating logic) - 因为Spring Batch使用 `step` 实现了主/从 分区逻辑, 如果出现错误则可以用 `StepExecutionListener` 来处理补偿逻辑。 JSR-352 提供了一个collection, 用来在其他组件发生错误时提供补偿逻辑, 并动态设置退出状态。这些组件包括:

构件接口(Artifact Interface)	说明(Description)
<code>javax.batch.api.partition.PartitionCollector</code>	slave step 可以通过该对象发送消息给 master. 每个 slave 线程都有一个实例.
<code>javax.batch.api.partition.PartitionAnalyzer</code>	End point, 接收由 <code>PartitionCollector</code> 收集的信息, 以及已完成分区的结果状态.
<code>javax.batch.api.partition.PartitionReducer</code>	为分区 step 提供补偿逻辑..

测试

- [12.10 测试](#)

12.10 测试

因为所有基于JSR-352的作业都是异步执行的,所以很难确定一项作业什么时候完成。为了辅助测试, Spring Batch提供了 `org.springframework.batch.core.jsr.JsrTestUtils`。这个工具类可以启动一个job,或者重新启动一个job,以及等待作业完成。当作业完成后,就返回相关联的JobExecution。

Spring Batch Integration模块

- 13. 集成 Spring Batch
 - 13.1. Spring Batch Integration Introduction
 - 13.1.1. Namespace Support
 - 13.1.2. Launching Batch Jobs through Messages
 - 13.1.3. Providing Feedback with Informational Messages
 - 13.1.4. Asynchronous Processors

13. 集成 Spring Batch

13.1. Spring Batch Integration Introduction

Many users of Spring Batch may encounter requirements that are outside the scope of Spring Batch, yet may be efficiently and concisely implemented using Spring Integration. Conversely, Spring Batch users may encounter Spring Batch requirements and need a way to efficiently integrate both frameworks. In this context several patterns and use-cases emerge and Spring Batch Integration will address those requirements.

The line between Spring Batch and Spring Integration is not always clear, but there are guidelines that one can follow. Principally, these are: think about granularity, and apply common patterns. Some of those common patterns are described in this reference manual section.

Adding messaging to a batch process enables automation of operations, and also separation and strategizing of key concerns. For example a message might trigger a job to execute, and then the sending of the message can be exposed in a variety of ways. Or when a job completes or fails that might trigger a message to be sent, and the consumers of those messages might have operational concerns that have nothing to do with the application itself. Messaging can also be embedded in a job, for example reading or writing items for processing via channels. Remote partitioning and remote chunking provide methods to distribute workloads over an number of workers.

Some key concepts that we will cover are:

- Namespace Support

- Launching Batch Jobs through Messages
- Providing Feedback with Informational Messages
- Asynchronous Processors
- Externalizing Batch Process Execution

13.1.1. Namespace Support

Since Spring Batch Integration 1.3, dedicated XML Namespace support was added, with the aim to provide an easier configuration experience. In order to activate the namespace, add the following namespace declarations to your Spring XML Application Context file:

```

1. <beans xmlns="http://www.springframework.org/schema/beans"
2.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xmlns:batch-int="http://www.springframework.org/schema/batch-integration"
4.     xsi:schemaLocation="
5.         http://www.springframework.org/schema/batch-integration
6.         http://www.springframework.org/schema/batch-integration/spring-batch-integration.xsd">
7.
8.     ...
9.
10. </beans>

```

A fully configured Spring XML Application Context file for Spring Batch Integration may look like the following:

```

1. <beans xmlns="http://www.springframework.org/schema/beans"
2.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xmlns:int="http://www.springframework.org/schema/integration"
4.     xmlns:batch="http://www.springframework.org/schema/batch"
5.     xmlns:batch-int="http://www.springframework.org/schema/batch-integration"
6.     xsi:schemaLocation="
7.         http://www.springframework.org/schema/batch-integration
8.         http://www.springframework.org/schema/batch-integration/spring-batch-integration.xsd
9.         http://www.springframework.org/schema/batch
10.        http://www.springframework.org/schema/batch/spring-batch.xsd
11.        http://www.springframework.org/schema/beans
12.        http://www.springframework.org/schema/beans/spring-beans.xsd
13.        http://www.springframework.org/schema/integration
14.        http://www.springframework.org/schema/integration/spring-integration.xsd">
15.
16.     ...
17.

```

```
18. </beans>
```

Appending version numbers to the referenced XSD file is also allowed but, as a version-less declaration will always use the latest schema, we generally don't recommend appending the version number to the XSD name. Adding a version number, for instance, would create possibly issues when updating the Spring Batch Integration dependencies as they may require more recent versions of the XML schema.

13.1.2. Launching Batch Jobs through Messages

When starting batch jobs using the core Spring Batch API you basically have 2 options:

- Command line via the `CommandLineJobRunner`
- Programmatically via either `JobOperator.start()` or `JobLauncher.run()`.

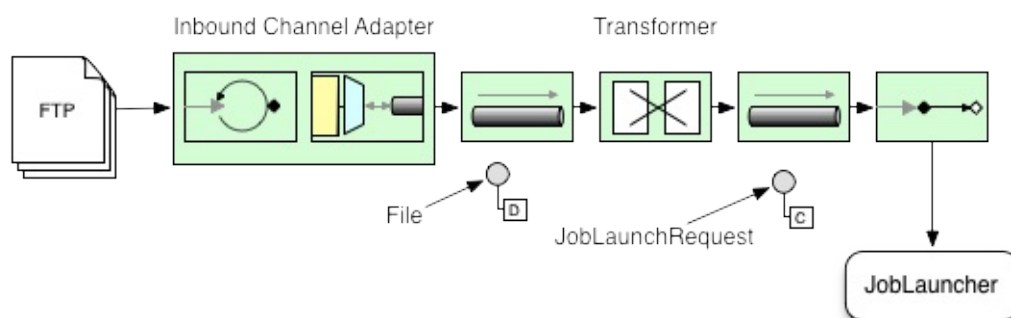
For example, you may want to use the `CommandLineJobRunner` when invoking Batch Jobs using a shell script. Alternatively, you may use the `JobOperator` directly, for example when using Spring Batch as part of a web application. However, what about more complex use-cases? Maybe you need to poll a remote (S)FTP server to retrieve the data for the Batch Job. Or your application has to support multiple different data sources simultaneously. For example, you may receive data files not only via the web, but also FTP etc. Maybe additional transformation of the input files is needed before invoking Spring Batch.

Therefore, it would be much more powerful to execute the batch job using Spring Integration and its numerous adapters. For example, you can use a `File Inbound Channel Adapter` to monitor a directory in the file-system and start the Batch Job as soon as the input file arrives. Additionally you can create Spring Integration flows that use multiple different adapters to easily ingest data for your Batch Jobs from multiple sources simultaneously using configuration only. Implementing all these scenarios with Spring Integration is easy as it allow for an decoupled event-driven execution of the `JobLauncher`.

Spring Batch Integration provides the `JobLaunchingMessageHandler` class that you can use to launch batch jobs. The input for the `JobLaunchingMessageHandler` is provided by a Spring Integration message, which payload is of type `JobLaunchRequest`. This class is a wrapper around

the Job that needs to be launched as well as the JobParameters necessary to launch the Batch job.

The following image illustrates the typical Spring Integration message flow in order to start a Batch job. The EIP (Enterprise IntegrationPatterns) website provides a full overview of messaging icons and their descriptions.



Transforming a file into a JobLaunchRequest

```

1. package io.spring.sbi;
2.
3. import org.springframework.batch.core.Job;
4. import org.springframework.batch.core.JobParametersBuilder;
5. import org.springframework.batch.integration.launch.JobLaunchRequest;
6. import org.springframework.integration.annotation.Transformer;
7. import org.springframework.messaging.Message;
8.
9. import java.io.File;
10.
11. public class FileMessageToJobRequest {
12.     private Job job;
13.     private String fileParameterName;
14.
15.     public void setFileParameterName(String fileParameterName) {
16.         this.fileParameterName = fileParameterName;
17.     }
18.
19.     public void setJob(Job job) {
20.         this.job = job;
21.     }
22.
23.     @Transformer
24.     public JobLaunchRequest toRequest(Message<File> message) {
25.         JobParametersBuilder jobParametersBuilder =
26.             new JobParametersBuilder();

```

```

27.
28.     jobParametersBuilder.addString(fileParameterName,
29.         message.getPayload().getAbsolutePath());
30.
31.     return new JobLaunchRequest(job, jobParametersBuilder.toJobParameters());
32. }
33. }

```

The JobExecution Response

When a Batch Job is being executed, a `JobExecution` instance is returned. This instance can be used to determine the status of an execution. If a `JobExecution` was able to be created successfully, it will always be returned, regardless of whether or not the actual execution was successful.

The exact behavior on how the `JobExecution` instance is returned depends on the provided `TaskExecutor`. If a synchronous (single-threaded) `TaskExecutor` implementation is used, the `JobExecution` response is only returned after the job completes. When using an asynchronous `TaskExecutor`, the `JobExecution` instance is returned immediately. Users can then take the id of `JobExecution` instance (`JobExecution.getJobId()`) and query the `JobRepository` for the job's updated status using the `JobExplorer`. For more information, please refer to the Spring Batch reference documentation on [Querying the Repository](#).

The following configuration will create a file inbound-channel-adapter to listen for CSV files in the provided directory, hand them off to our transformer (`FileMessageToJobRequest`), launch the job via the Job Launching Gateway then simply log the output of the `JobExecution` via the logging-channel-adapter.

Spring Batch Integration Configuration

```

1. <int:channel id="inboundFileChannel"/>
2. <int:channel id="outboundJobRequestChannel"/>
3. <int:channel id="jobLaunchReplyChannel"/>
4.
5. <int-file:inbound-channel-adapter id="filePoller"
6.     channel="inboundFileChannel"
7.     directory="file:/tmp/myfiles/"
8.     filename-pattern="*.csv">
9.     <int:poller fixed-rate="1000"/>
10. </int-file:inbound-channel-adapter>
11.

```

```

12. <int:transformer input-channel="inboundFileChannel"
13.     output-channel="outboundJobRequestChannel">
14.   <bean class="io.spring.sbi.FileMessageToJobRequest">
15.     <property name="job" ref="personJob"/>
16.     <property name="fileParameterName" value="input.file.name"/>
17.   </bean>
18. </int:transformer>
19.
20. <batch-int:job-launching-gateway request-channel="outboundJobRequestChannel"
21.     reply-channel="jobLaunchReplyChannel"/>
22.
23. <int:logging-channel-adapter channel="jobLaunchReplyChannel"/>

```

Now that we are polling for files and launching jobs, we need to configure for example our Spring Batch ItemReader to utilize found file represented by the job parameter "input.file.name":

Example ItemReader Configuration

```

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

```

The main points of interest here are injecting the value of `#{jobParameters['input.file.name']}` as the Resource property value and setting the ItemReader bean to be of Step scope to take advantage of the late binding support which allows access to the jobParameters variable.

Available Attributes of the Job-Launching Gateway

- id Identifies the underlying Spring bean definition, which is an instance of either:
 - EventDrivenConsumer
 - PollingConsumer

The exact implementation depends on whether the component's input channel is a:

- SubscribableChannel or
- PollableChannel

- `auto-startup` Boolean flag to indicate that the endpoint should start automatically on startup. The default is `true`.
- `request-channel` The input `MessageChannel` of this endpoint.
- `reply-channel` Message Channel to which the resulting `JobExecution` payload will be sent.
- `reply-timeout` Allows you to specify how long this gateway will wait for the reply message to be sent successfully to the reply channel before throwing an exception. This attribute only applies when the channel might block, for example when using a bounded queue channel that is currently full. Also, keep in mind that when sending to a `DirectChannel`, the invocation will occur in the sender's thread. Therefore, the failing of the send operation may be caused by other components further downstream. The `reply-timeout` attribute maps to the `sendTimeout` property of the underlying `MessagingTemplate` instance. The attribute will default, if not specified, to `-1`, meaning that by default, the Gateway will wait indefinitely. The value is specified in milliseconds.
- `job-launcher` Pass in a custom `JobLauncher` bean reference. This attribute is optional. If not specified the adapter will re-use the instance that is registered under the id `jobLauncher`. If no default instance exists an exception is thrown.
- `order` Specifies the order for invocation when this endpoint is connected as a subscriber to a `SubscribableChannel`.

Sub-Elements

When this Gateway is receiving messages from a `PollableChannel`, you must either provide a global default `Poller` or provide a `Poller` sub-element to the Job Launching Gateway:

```

1. <batch-int:job-launching-gateway request-channel="queueChannel"
2.     reply-channel="replyChannel" job-launcher="jobLauncher">
3.     <int:poller fixed-rate="1000"/>
4. </batch-int:job-launching-gateway>

```

13.1.3. Providing Feedback with Informational Messages

As Spring Batch jobs can run for long times, providing progress information will be critical. For example, stake-holders may want to be notified if a some or all parts of a Batch Job has failed. Spring Batch provides support for this information being gathered through:

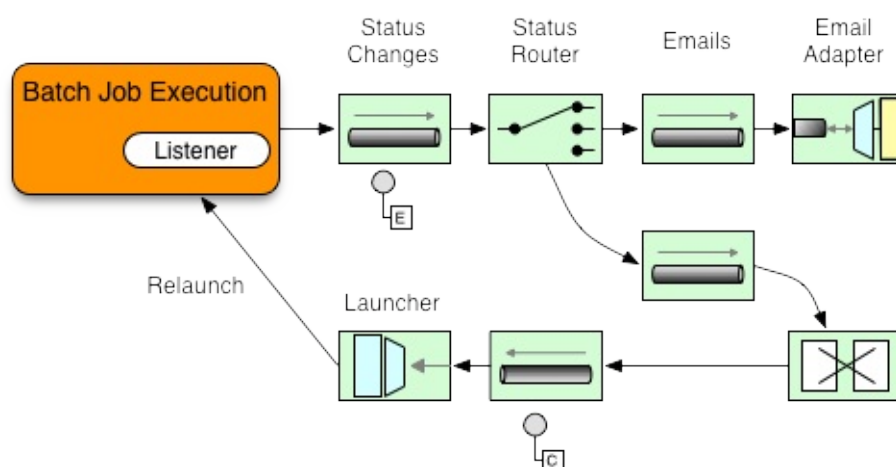
- Active polling or
- Event-driven, using listeners.

When starting a Spring Batch job asynchronously, e.g. by using the Job Launching Gateway, a JobExecution instance is returned. Thus, JobExecution.getJobId() can be used to continuously poll for status updates by retrieving updated instances of the JobExecution from the JobRepository using the JobExplorer. However, this is considered sub-optimal and an event-driven approach should be preferred.

Therefore, Spring Batch provides listeners such as:

- StepListener
- ChunkListener
- JobExecutionListener

In the following example, a Spring Batch job was configured with a StepExecutionListener. Thus, Spring Integration will receive and process any step before/after step events. For example, the received StepExecution can be inspected using a Router. Based on the results of that inspection, various things can occur for example routing a message to a Mail Outbound Channel Adapter, so that an Email notification can be send out based on some condition.



Below is an example of how a listener is configured to send a message to a Gateway for StepExecution events and log its output to a logging-channel-

adapter:

First create the notifications integration beans:

```

1. <int:channel id="stepExecutionsChannel"/>
2.
3. <int:gateway id="notificationExecutionsListener"
4.     service-interface="org.springframework.batch.core.StepExecutionListener"
5.     default-request-channel="stepExecutionsChannel"/>
6.
7. <int:logging-channel-adapter channel="stepExecutionsChannel"/>

```

Then modify your job to add a step level listener:

```

1. <job id="importPayments">
2.     <step id="step1">
3.         <tasklet ../>
4.             <chunk ../>
5.                 <listeners>
6.                     <listener ref="notificationExecutionsListener"/>
7.                 </listeners>
8.             </tasklet>
9.             ...
10.        </step>
11. </job>

```

13.1.4. Asynchronous Processors

Asynchronous Processors help you to scale the processing of items. In the asynchronous processor use-case, an AsyncItemProcessor serves as a dispatcher, executing the ItemProcessor's logic for an item on a new thread. The Future is passed to the AsyncItemWriter to be written once the processor completes.

Therefore, you can increase performance by using asynchronous item processing, basically allowing you to implement fork-join scenarios. The AsyncItemWriter will gather the results and write back the chunk as soon as all the results become available.

Configuration of both the AsyncItemProcessor and AsyncItemWriter are simple, first the AsyncItemProcessor:

```

1. <bean id="processor"
2.     class="org.springframework.batch.integration.async.AsyncItemProcessor">

```

```

3. <property name="delegate">
4.     <bean class="your.ItemProcessor"/>
5. </property>
6. <property name="taskExecutor">
7.     <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>
8. </property>
9. </bean>

```

The property “delegate” is actually a reference to your ItemProcessor bean and the “taskExecutor” property is a reference to the TaskExecutor of your choice.

Then we configure the AsyncItemWriter:

```

1. <bean id="itemWriter"
2.     class="org.springframework.batch.integration.async.AsyncItemWriter">
3. <property name="delegate">
4.     <bean id="itemWriter" class="your.ItemWriter"/>
5. </property>
6. </bean>

```

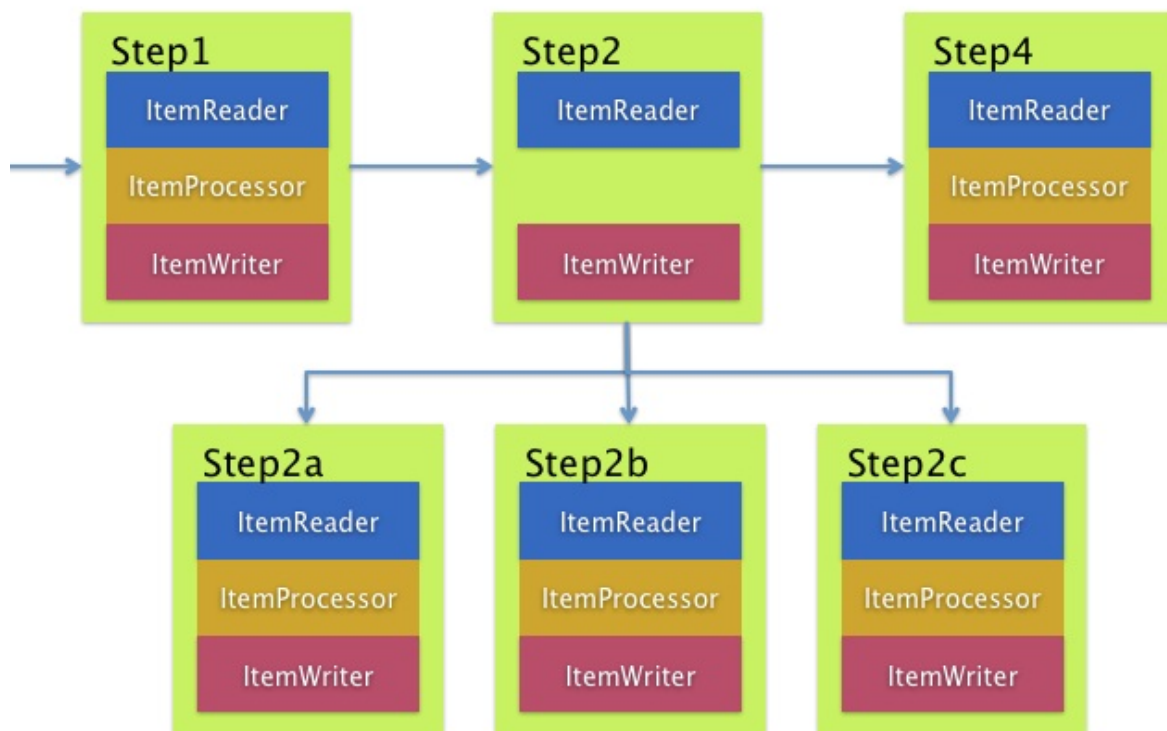
Again, the property “delegate” is actually a reference to your ItemWriter bean.

13.1.5. Externalizing Batch Process Execution

The integration approaches discussed so far suggest use-cases where Spring Integration wraps Spring Batch like an outer-shell. However, Spring Batch can also use Spring Integration internally. Using this approach, Spring Batch users can delegate the processing of items or even chunks to outside processes. This allows you to offload complex processing. Spring Batch Integration provides dedicated support for:

- Remote Chunking
- Remote Partitioning

Remote Chunking



Taking things one step further, one can also externalize the chunk processing using the `ChunkMessageChannelItemWriter` which is provided by Spring Batch Integration which will send items out and collect the result. Once sent, Spring Batch will continue the process of reading and grouping items, without waiting for the results. Rather it is the responsibility of the `ChunkMessageChannelItemWriter` to gather the results and integrate them back into the Spring Batch process.

Using Spring Integration you have full control over the concurrency of your processes, for instance by using a `QueueChannel` instead of a `DirectChannel`. Furthermore, by relying on Spring Integration's rich collection of Channel Adapters (E.g. JMS or AMQP), you can distribute chunks of a Batch job to external systems for processing.

A simple job with a step to be remotely chunked would have a configuration similar to the following:

```

1. <job id="personJob">
2.   <step id="step1">
3.     <tasklet>
4.       <chunk reader="itemReader" writer="itemWriter" commit-interval="200"/>
5.     </tasklet>
6.     ...
7.   </step>

```

```
8. </job>
```

The ItemReader reference would point to the bean you would like to use for reading data on the master. The ItemWriter reference points to a special ItemWriter “ChunkMessageChannelItemWriter” as described above. The processor (if any) is left off the master configuration as it is configured on the slave. The following configuration provides a basic master setup. It’s advised to check any additional component properties such as throttle limits and so on when implementing your use case.

```
1. <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
2.   <property name="brokerURL" value="tcp://localhost:61616"/>
3. </bean>
4.
5. <int-jms:outbound-channel-adapter id="requests" destination-name="requests"/>
6.
7. <bean id="messagingTemplate"
8.   class="org.springframework.integration.core.MessagingTemplate">
9.   <property name="defaultChannel" ref="requests"/>
10.  <property name="receiveTimeout" value="2000"/>
11. </bean>
12.
13. <bean id="itemWriter"
14.   class="org.springframework.batch.integration.chunk.ChunkMessageChannelItemWriter"
15.   scope="step">
16.  <property name="messagingOperations" ref="messagingTemplate"/>
17.  <property name="replyChannel" ref="replies"/>
18. </bean>
19.
20. <bean id="chunkHandler"
21.   class="org.springframework.batch.integration.chunk.RemoteChunkHandlerFactoryBean">
22.  <property name="chunkWriter" ref="itemWriter"/>
23.  <property name="step" ref="step1"/>
24. </bean>
25.
26. <int:channel id="replies">
27.  <int:queue/>
28. </int:channel>
29.
30. <int-jms:message-driven-channel-adapter id="jmsReplies"
31.   destination-name="replies"
32.   channel="replies"/>
```

This configuration provides us with a number of beans. We configure our messaging middleware using ActiveMQ and inbound/outbound JMS adapters provided by Spring Integration. As shown, our itemWriter bean which is

referenced by our job step utilizes the `ChunkMessageChannelItemWriter` for writing chunks over the configured middleware.

Now lets move on to the slave configuration:

```

1. <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
2.   <property name="brokerURL" value="tcp://localhost:61616"/>
3. </bean>
4.
5. <int:channel id="requests"/>
6. <int:channel id="replies"/>
7.
8. <int-jms:message-driven-channel-adapter id="jmsIn"
9.   destination-name="requests"
10.  channel="requests"/>
11.
12. <int-jms:outbound-channel-adapter id="outgoingReplies"
13.  destination-name="replies"
14.  channel="replies">
15. </int-jms:outbound-channel-adapter>
16.
17. <int:service-activator id="serviceActivator"
18.  input-channel="requests"
19.  output-channel="replies"
20.  ref="chunkProcessorChunkHandler"
21.  method="handleChunk"/>
22.
23. <bean id="chunkProcessorChunkHandler"
24.  class="org.springframework.batch.integration.chunk.ChunkProcessorChunkHandler">
25.  <property name="chunkProcessor">
26.    <bean class="org.springframework.batch.core.step.item.SimpleChunkProcessor">
27.      <property name="itemWriter">
28.        <bean class="io.spring.sbi.PersonItemWriter"/>
29.      </property>
30.      <property name="itemProcessor">
31.        <bean class="io.spring.sbi.PersonItemProcessor"/>
32.      </property>
33.    </bean>
34.  </property>
35. </bean>

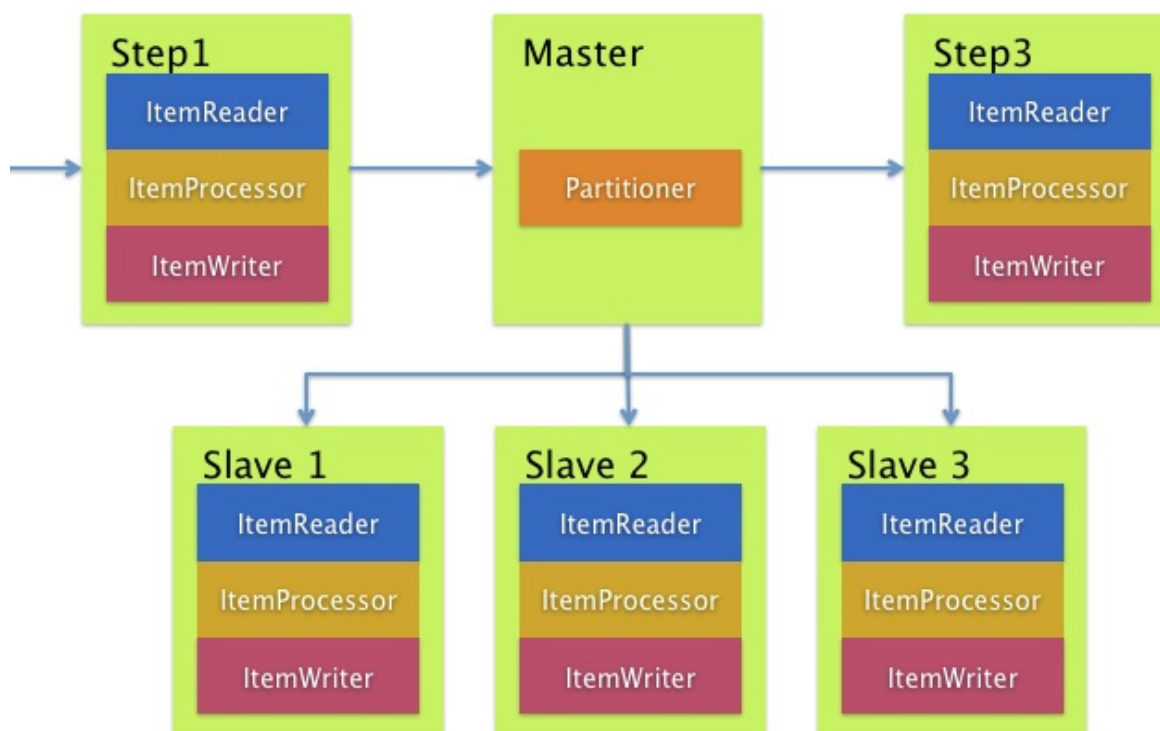
```

Most of these configuration items should look familiar from the master configuration. Slaves do not need access to things like the Spring Batch `JobRepository` nor access to the actual job configuration file. The main bean of interest is the `chunkProcessorChunkHandler`. The `chunkProcessor` property of `ChunkProcessorChunkHandler` takes a configured

SimpleChunkProcessor which is where you would provide a reference to your ItemWriter and optionally your ItemProcessor that will run on the slave when it receives chunks from the master.

For more information, please also consult the Spring Batch manual, specifically the chapter on Remote Chunking.

Remote Partitioning



Remote Partitioning, on the other hand, is useful when the problem is not the processing of items, but the associated I/O represents the bottleneck. Using Remote Partitioning, work can be farmed out to slaves that execute complete Spring Batch steps. Thus, each slave has its own ItemReader, ItemProcessor and ItemWriter. For this purpose, Spring Batch Integration provides the MessageChannelPartitionHandler.

This implementation of the PartitionHandler interface uses MessageChannel instances to send instructions to remote workers and receive their responses. This provides a nice abstraction from the transports (E.g. JMS or AMQP) being used to communicate with the remote workers.

The reference manual section Remote Partitioning provides an overview of the concepts and components needed to configure Remote Partitioning and shows an example of using the default TaskExecutorPartitionHandler to

partition in separate local threads of execution. For Remote Partitioning to multiple JVM's, two additional components are required:

- Remoting fabric or grid environment
- A PartitionHandler implementation that supports the desired remoting fabric or grid environment

Similar to Remote Chunking JMS can be used as the "remoting fabric" and the PartitionHandler implementation to be used as described above is the MessageChannelPartitionHandler. The example shown below assumes an existing partitioned job and focuses on the MessageChannelPartitionHandler and JMS configuration:

```

1. <bean id="partitionHandler"
2.     class="org.springframework.batch.integration.partition.MessageChannelPartitionHandler">
3.     <property name="stepName" value="step1"/>
4.     <property name="gridSize" value="3"/>
5.     <property name="replyChannel" ref="outbound-replies"/>
6.     <property name="messagingOperations">
7.         <bean class="org.springframework.integration.core.MessagingTemplate">
8.             <property name="defaultChannel" ref="outbound-requests"/>
9.             <property name="receiveTimeout" value="100000"/>
10.        </bean>
11.    </property>
12. </bean>
13.
14. <int:channel id="outbound-requests"/>
15. <int-jms:outbound-channel-adapter destination="requestsQueue"
16.     channel="outbound-requests"/>
17.
18. <int:channel id="inbound-requests"/>
19. <int-jms:message-driven-channel-adapter destination="requestsQueue"
20.     channel="inbound-requests"/>
21.
22. <bean id="stepExecutionRequestHandler"
23.     class="org.springframework.batch.integration.partition.StepExecutionRequestHandler">
24.     <property name="jobExplorer" ref="jobExplorer"/>
25.     <property name="stepLocator" ref="stepLocator"/>
26. </bean>
27.
28. <int:service-activator ref="stepExecutionRequestHandler" input-channel="inbound-requests"
29.     output-channel="outbound-staging"/>
30.
31. <int:channel id="outbound-staging"/>
32. <int-jms:outbound-channel-adapter destination="stagingQueue"
33.     channel="outbound-staging"/>
34.

```

```
35. <int:channel id="inbound-staging"/>
36. <int-jms:message-driven-channel-adapter destination="stagingQueue"
37.     channel="inbound-staging"/>
38.
39. <int:aggregator ref="partitionHandler" input-channel="inbound-staging"
40.     output-channel="outbound-replies"/>
41.
42. <int:channel id="outbound-replies">
43.     <int:queue/>
44. </int:channel>
45.
46. <bean id="stepLocator"
47.     class="org.springframework.batch.integration.partition.BeanFactoryStepLocator" />
```

Also ensure the partition handler attribute maps to the partitionHandler bean:

```
1. <job id="personJob">
2.     <step id="step1.master">
3.         <partition partitioner="partitioner" handler="partitionHandler"/>
4.         ...
5.     </step>
6. </job>
```


附录A

- [附录A ItemReader 与 ItemWriter 列表](#)
 - [A.1 Item Readers](#)
 - [A.2 Item Writers](#)

附录A ItemReader 与 ItemWriter 列表

A.1 Item Readers

Table A.1. 所有可用的Item Reader列表

Item Reader	说明
AbstractItemCountingItemStreamItemReader	抽象基类，支持重启，通过统计 (counting) 从 <code>ItemReader</code> 返回对象数量来实现。

AggregateItemReader	此 ItemReader 提供一个 list , 用存储 ItemReader 读取的对象, 直到他们已准备装配为一个集合。此 ItemReader 通过 FieldSetMapper 的常量值 AggregateItemReader#BEGIN_RECORD 以及 AggregateItemReader#END_RECORD 来标记记录的开始与结束。
AmqpItemReader	给定一个提供同步获取方法(synchronous receive methods)的 Spring AmqpTemplate. 使用 receiveAndConvert() 方法可以得到 POJO 对象.
FlatFileItemReader	从平面文件(flat file)中读取数据, 支持 ItemStream 以及 Skippable 特性. 请参考 Read from a File 一节
HibernateCursorItemReader	从基于 HQL 查询的 cursor 中读取数据. 请参考 Reading from a Database 一节。
HibernatePagingItemReader	从分页(paginated)的HQL查询中读取数据
IbatisPagingItemReader	通过 iBATIS 的分页查询读取数据, 对大型数据集, 分页能避免内存不足/溢出的问题. 请参考: HOWTO - Read from a Database. 这个 ItemReader 在 Spring Batch 3.0 中已废弃.
ItemReaderAdapter	将任意类适配到 ItemReader 接口.
JdbcCursorItemReader	通过 JDBC 从一个 database cursor 中读取数据. 请参考: HOWTO - Read from a Database
JdbcPagingItemReader	给定一个 SQL statement, 通过分页查询读取数据, 避免读取大型数据集时内存不足/溢出的问题
JmsItemReader	给一个 Spring JmsOperations 对象一个 JMS Destination 对象/也可以是用来发送错误的 destination name , 调用注入 JmsOperations 里面的 receive() 方法来获取对象
JpaPagingItemReader	给定一个 JPQL statement, 通过分页查询读取数据, 避免读取大型数据集时内存不足/溢出的问题
ListItemReader	从 list 中读取数据, 一次返回一条
MongoItemReader	给定一个 MongoOperations 对象, 从 MongoDB 中查询数据所使用的JSON, 通过

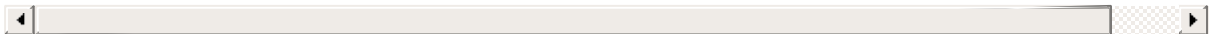
	MongoOperations 的 find 方法来获取数据
Neo4jItemReader	给定一个 Neo4jOperations 对象, 以一个 Cypher query 所需的 components, 将 Neo4jOperations.query 方法的结果返回
RepositoryItemReader	给定一个 Spring Data PagingAndSortingRepository 对象, 一个 Sort 对象, 以及要执行的方法名, 返回 Spring Data repository 实现提供的数据
StoredProcedureItemReader	执行存储过程(database stored procedure), 从返回的 database cursor 中读取数据. 请参考: HOWTO Read from a Database
StaxEventItemReader	通过 StAX 读取. 请参考 HOWTO - Read from a File

A.2 Item Writers

Table A.2. 所有可用的 *Item Writer* 列表

Item Writer	说明
AbstractItemStreamItemWriter	抽象基类, 组合了 <code>ItemStream</code> 和 <code>ItemWriter</code> 接口。
AmqpItemWriter	给定一个提供同步发送方法的 <code>Spring AmqpTemplate</code> . 使用 <code>convertAndSend(Object)</code> 方法可以输出 POJO 对象.
CompositeItemWriter	将注入的 <code>List</code> 里面每一个元素都传给 <code>ItemWriter</code> 的处理方法
FlatFileItemWriter	写入平面文件(Flat file). 支持 <code>ItemStream</code> 以及 <code>Skippable</code> 特性. 请参考 <code>Writing to a File</code> 一节
GemfireItemWriter	使用 <code>GemfireOperations</code> 对象, 根据配置的 <code>delete</code> 标志, 对 items 进行写入或者删除
HibernateItemWriter	这个 item writer 是 <code>Hibernate</code> 会话相关的(<code>hibernate session aware</code>), 用来处理非 <code>hibernate</code> 相关的组件 (<code>non-“hibernate aware”</code>) 不需要关心的事务性工作, 并且委托另一个 item writer 来执行实际的写入工作.
IbatisBatchItemWriter	在批处理中直接使用 <code>iBatis</code> 的 API. 这个 <code>ItemWriter</code> 在 <code>Spring Batch 3.0</code> 中已废弃.
ItemWriterAdapter	将任意类适配到 <code>ItemWriter</code> 接口.
JdbcBatchItemWriter	尽可能地利用 <code>PreparedStatement</code> 的批处理功能 (<code>batching features</code>), 还可以采取基本的步骤来定位 <code>flush</code> 失败等问题.
JmsItemWriter	利用 <code>JmsOperations</code> 对象, 通过 <code>JmsOperations.convertAndSend()</code> 方法将 items 写入到默认队列(<code>default queue</code>)
JpaItemWriter	这个 item writer 是 <code>JPA EntityManager aware</code> 的, 用来处理非 <code>jpa</code> 相关的 (<code>non-“jpa aware”</code>) <code>ItemWriter</code> 不需要关心的事务性工作, 并且委托另一个 item writer 来执行实际的写入工作.
MimeMessageItemWriter	通过 <code>Spring</code> 的 <code>JavaMailSender</code> 对象, 类型为 <code>MimeMessage</code> 的 item 可以作为 <code>mail messages</code> 发送出去
	给定一个 <code>MongoOperations</code> 对象, 数据通过

MongoItemWriter	MongoOperations.save(Object) 方法写入。实际的写操作会推迟到事务提交时才执行。
Neo4jItemWriter	给定一个 Neo4jOperations 对象, item 通过 save(Object) 方法完成持久化, 或者通过 delete(Object) 方法来删除, 取决于 ItemWriter 的配置
PropertyExtractingDelegatingItemWriter	扩展 AbstractMethodInvokingDelegator 创建动态参数。动态参数是通过注入的 field name 数组, 从 (SpringBeanWrapper) 处理的 item 中获取的
RepositoryItemWriter	给定一个 Spring Data CrudRepository 实现, 则使用配置文件指定的方法保存 item。
StaxEventItemWriter	通过 ObjectToXmlSerializer 对象将每个 item 转换为 XML, 然后用 StAX 将这些内容写入 XML 文件。



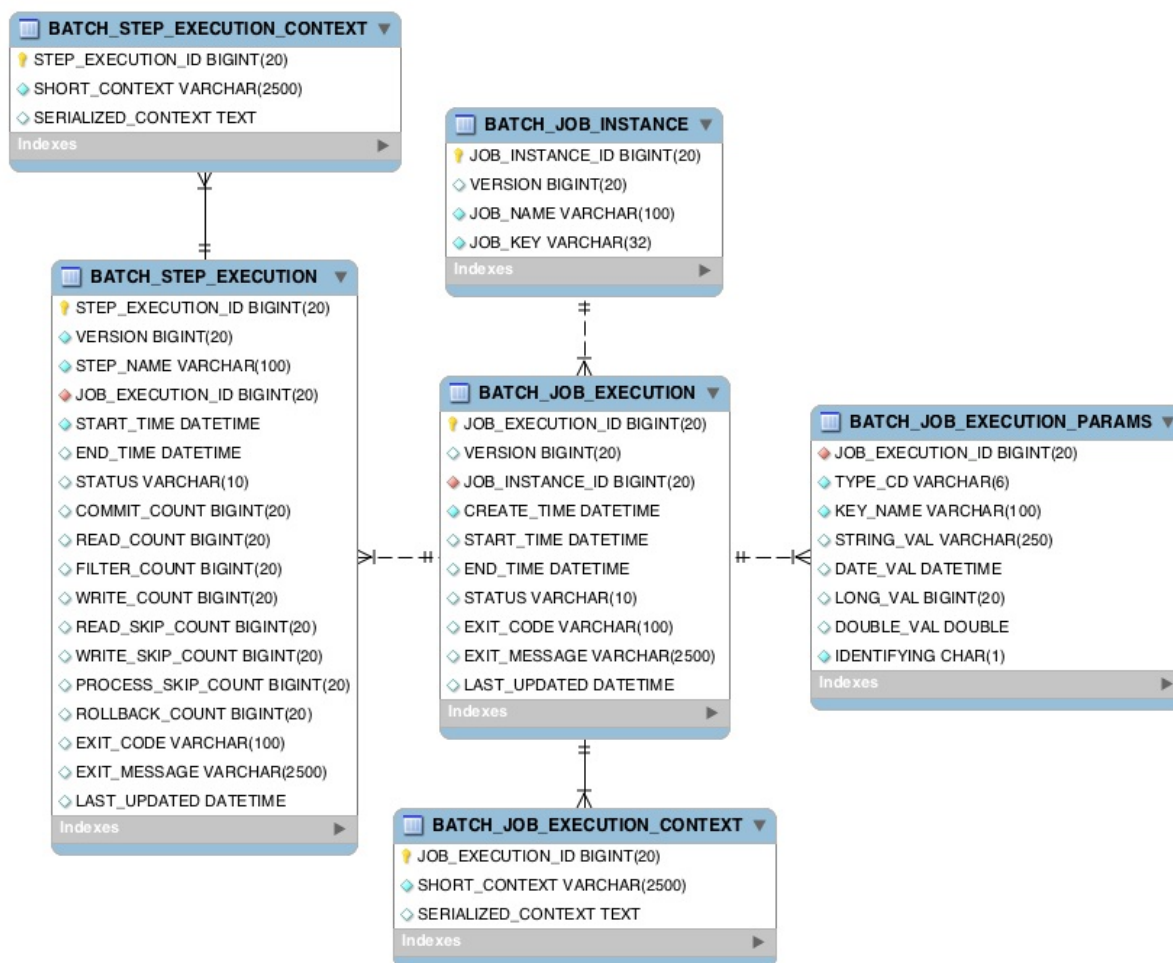
附录B

- 15. 附录B Meta-Data Schema
 - B.1 Overview
 - B.1.1 Example DDL Scripts
 - B.1.2 Version
 - B.1.3 Identity
 - B.2 BATCH_JOB_INSTANCE
 - B.3 BATCH_JOB_EXECUTION_PARAMS
 - B.4 BATCH_JOB_EXECUTION
 - B.5 BATCH_STEP_EXECUTION
 - B.6 BATCH_JOB_EXECUTION_CONTEXT
 - B.7 BATCH_STEP_EXECUTION_CONTEXT
 - B.8 Archiving
 - B.9 International and Multi-byte Characters
 - B.10 Recommendations for Indexing Meta Data Tables

15. 附录B Meta-Data Schema

B.1 Overview

The Spring Batch Meta-Data tables very closely match the Domain objects that represent them in Java. For example, `JobInstance`, `JobExecution`, `JobParameters`, and `StepExecution` map to `BATCH_JOB_INSTANCE`, `BATCH_JOB_EXECUTION`, `BATCH_JOB_EXECUTION_PARAMS`, and `BATCH_STEP_EXECUTION`, respectively. `ExecutionContext` maps to both `BATCH_JOB_EXECUTION_CONTEXT` and `BATCH_STEP_EXECUTION_CONTEXT`. The `JobRepository` is responsible for saving and storing each Java object into its correct table. The following appendix describes the meta-data tables in detail, along with many of the design decisions that were made when creating them. When viewing the various table creation statements below, it is important to realize that the data types used are as generic as possible. Spring Batch provides many schemas as examples, which all have varying data types due to variations in individual database vendors' handling of data types. Below is an ERD model of all 6 tables and their relationships to one another:



B.1.1 Example DDL Scripts

The Spring Batch Core JAR file contains example scripts to create the relational tables for a number of database platforms (which are in turn auto-detected by the job repository factory bean or namespace equivalent). These scripts can be used as is, or modified with additional indexes and constraints as desired. The file names are in the form `schema-.sql`, where `""` is the short name of the target database platform. The scripts are in the package `org.springframework.batch.core`.

B.1.2 Version

Many of the database tables discussed in this appendix contain a version column. This column is important because Spring Batch employs an optimistic locking strategy when dealing with updates to the database. This means that each time a record is 'touched' (updated) the value in the version column is incremented by one. When the repository goes back to try and save the value, if the version number has change it will throw

OptimisticLockingFailureException, indicating there has been an error with concurrent access. This check is necessary since, even though different batch jobs may be running in different machines, they are all using the same database tables.

B.1.3 Identity

BATCH_JOB_INSTANCE, BATCH_JOB_EXECUTION, and BATCH_STEP_EXECUTION each contain columns ending in _ID. These fields act as primary keys for their respective tables. However, they are not database generated keys, but rather they are generated by separate sequences. This is necessary because after inserting one of the domain objects into the database, the key it is given needs to be set on the actual object so that they can be uniquely identified in Java. Newer database drivers (Jdbc 3.0 and up) support this feature with database generated keys, but rather than requiring it, sequences were used. Each variation of the schema will contain some form of the following:

1. CREATE SEQUENCE BATCH_STEP_EXECUTION_SEQ;
2. CREATE SEQUENCE BATCH_JOB_EXECUTION_SEQ;
3. CREATE SEQUENCE BATCH_JOB_SEQ;

Many database vendors don't support sequences. In these cases, work-arounds are used, such as the following for MySQL:

1. CREATE TABLE BATCH_STEP_EXECUTION_SEQ (ID BIGINT NOT NULL) type=InnoDB;
2. INSERT INTO BATCH_STEP_EXECUTION_SEQ values(0);
3. CREATE TABLE BATCH_JOB_EXECUTION_SEQ (ID BIGINT NOT NULL) type=InnoDB;
4. INSERT INTO BATCH_JOB_EXECUTION_SEQ values(0);
5. CREATE TABLE BATCH_JOB_SEQ (ID BIGINT NOT NULL) type=InnoDB;
6. INSERT INTO BATCH_JOB_SEQ values(0);

In the above case, a table is used in place of each sequence. The Spring core class MySQLMaxValueIncrementer will then increment the one column in this sequence in order to give similar functionality.

B.2 BATCH_JOB_INSTANCE

The BATCH_JOB_INSTANCE table holds all information relevant to a JobInstance, and serves as the top of the overall hierarchy. The following generic DDL statement is used to create it:

```

1. CREATE TABLE BATCH_JOB_INSTANCE (
2.     JOB_INSTANCE_ID BIGINT PRIMARY KEY ,
3.     VERSION BIGINT,
4.     JOB_NAME VARCHAR(100) NOT NULL ,
5.     JOB_KEY VARCHAR(2500)
6. );

```

Below are descriptions of each column in the table:

- **JOB_INSTANCE_ID**: The unique id that will identify the instance, which is also the primary key. The value of this column should be obtainable by calling the `getId` method on `JobInstance`.
- **VERSION**: See above section.
- **JOB_NAME**: Name of the job obtained from the `Job` object. Because it is required to identify the instance, it must not be null.
- **JOB_KEY**: A serialization of the `JobParameters` that uniquely identifies separate instances of the same job from one another. (`JobInstances` with the same job name must have different `JobParameters`, and thus, different `JOB_KEY` values).

B.3 BATCH_JOB_EXECUTION_PARAMS

The `BATCH_JOB_EXECUTION_PARAMS` table holds all information relevant to the `JobParameters` object. It contains 0 or more key/value pairs passed to a `Job` and serve as a record of the parameters a job was run with. For each parameter that contributes to the generation of a job's identity, the `IDENTIFYING` flag is set to true. It should be noted that the table has been denormalized. Rather than creating a separate table for each type, there is one table with a column indicating the type:

```

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. );

```

Below are descriptions for each column:

- `JOB_EXECUTION_ID`: Foreign Key from the `BATCH_JOB_EXECUTION` table that indicates the job execution the parameter entry belongs to. It should be noted that multiple rows (i.e key/value pairs) may exist for each execution.
- `TYPE_CD`: String representation of the type of value stored, which can be either a string, date, long, or double. Because the type must be known, it cannot be null.
- `KEY_NAME`: The parameter key.
- `STRING_VAL`: Parameter value, if the type is string.
- `DATE_VAL`: Parameter value, if the type is date.
- `LONG_VAL`: Parameter value, if the type is a long.
- `DOUBLE_VAL`: Parameter value, if the type is double.
- `IDENTIFYING`: Flag indicating if the parameter contributed to the identity of the related obInstance.

It is worth noting that there is no primary key for this table. This is simply because the framework has no use for one, and thus doesn't require it. If a user so chooses, one may be added with a database generated key, without causing any issues to the framework itself.

B.4 BATCH_JOB_EXECUTION

The `BATCH_JOB_EXECUTION` table holds all information relevant to the `JobExecution` object. Every time a Job is run there will always be a new `JobExecution`, and a new row in this table:

```

1. CREATE TABLE BATCH_JOB_EXECUTION (
2.   JOB_EXECUTION_ID BIGINT PRIMARY KEY ,
3.   VERSION BIGINT,
4.   JOB_INSTANCE_ID BIGINT NOT NULL,
5.   CREATE_TIME TIMESTAMP NOT NULL,
6.   START_TIME TIMESTAMP DEFAULT NULL,
7.   END_TIME TIMESTAMP DEFAULT NULL,
8.   STATUS VARCHAR(10),
9.   EXIT_CODE VARCHAR(20),
10.  EXIT_MESSAGE VARCHAR(2500),
11.  LAST_UPDATED TIMESTAMP,
12.  constraint JOB_INSTANCE_EXECUTION_FK foreign key (JOB_INSTANCE_ID)
13.  references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
14. ) ;

```

Below are descriptions for each column:

- **JOB_EXECUTION_ID**: Primary key that uniquely identifies this execution. The value of this column is obtainable by calling the `getId` method of the `JobExecution` object.
- **VERSION**: See above section.
- **JOB_INSTANCE_ID**: Foreign key from the `BATCH_JOB_INSTANCE` table indicating the instance to which this execution belongs. There may be more than one execution per instance.
- **CREATE_TIME**: Timestamp representing the time that the execution was created.
- **START_TIME**: Timestamp representing the time the execution was started.
- **END_TIME**: Timestamp representing the time the execution was finished, regardless of success or failure. An empty value in this column even though the job is not currently running indicates that there has been some type of error and the framework was unable to perform a last save before failing.
- **STATUS**: Character string representing the status of the execution. This may be `COMPLETED`, `STARTED`, etc. The object representation of this column is the `BatchStatus` enumeration.
- **EXIT_CODE**: Character string representing the exit code of the execution. In the case of a command line job, this may be converted into a number.
- **EXIT_MESSAGE**: Character string representing a more detailed description of how the job exited. In the case of failure, this might include as much of the stack trace as is possible.
- **LAST_UPDATED**: Timestamp representing the last time this execution was persisted.

B.5 BATCH_STEP_EXECUTION

The `BATCH_STEP_EXECUTION` table holds all information relevant to the `StepExecution` object. This table is very similar in many ways to the `BATCH_JOB_EXECUTION` table and there will always be at least one entry per `Step` for each `JobExecution` created:

```

1. CREATE TABLE BATCH_STEP_EXECUTION (
2.   STEP_EXECUTION_ID BIGINT PRIMARY KEY ,
3.   VERSION BIGINT NOT NULL,
4.   STEP_NAME VARCHAR(100) NOT NULL,
5.   JOB_EXECUTION_ID BIGINT NOT NULL,
6.   START_TIME TIMESTAMP NOT NULL ,

```

```

7.   END_TIME TIMESTAMP 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(20) ,
18.  EXIT_MESSAGE VARCHAR(2500) ,
19.  LAST_UPDATED TIMESTAMP,
20.  constraint JOB_EXECUTION_STEP_FK foreign key (JOB_EXECUTION_ID)
21.  references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
22. ) ;

```

Below are descriptions for each column:

- **STEP_EXECUTION_ID**: Primary key that uniquely identifies this execution. The value of this column should be obtainable by calling the `getId` method of the `StepExecution` object.
- **VERSION**: See above section.
- **STEP_NAME**: The name of the step to which this execution belongs.
- **JOB_EXECUTION_ID**: Foreign key from the `BATCH_JOB_EXECUTION` table indicating the `JobExecution` to which this `StepExecution` belongs. There may be only one `StepExecution` for a given `JobExecution` for a given `Step` name.
- **START_TIME**: Timestamp representing the time the execution was started.
- **END_TIME**: Timestamp representing the time the execution was finished, regardless of success or failure. An empty value in this column even though the job is not currently running indicates that there has been some type of error and the framework was unable to perform a last save before failing.
- **STATUS**: Character string representing the status of the execution. This may be `COMPLETED`, `STARTED`, etc. The object representation of this column is the `BatchStatus` enumeration.
- **COMMIT_COUNT**: The number of times in which the step has committed a transaction during this execution.
- **READ_COUNT**: The number of items read during this execution.
- **FILTER_COUNT**: The number of items filtered out of this execution.
- **WRITE_COUNT**: The number of items written and committed during this execution.
- **READ_SKIP_COUNT**: The number of items skipped on read during this

execution.

- `WRITE_SKIP_COUNT`: The number of items skipped on write during this execution.
- `PROCESS_SKIP_COUNT`: The number of items skipped during processing during this execution.
- `ROLLBACK_COUNT`: The number of rollbacks during this execution. Note that this count includes each time rollback occurs, including rollbacks for retry and those in the skip recovery procedure.
- `EXIT_CODE`: Character string representing the exit code of the execution. In the case of a command line job, this may be converted into a number.
- `EXIT_MESSAGE`: Character string representing a more detailed description of how the job exited. In the case of failure, this might include as much of the stack trace as is possible.
- `LAST_UPDATED`: Timestamp representing the last time this execution was persisted.

B.6 BATCH_JOB_EXECUTION_CONTEXT

The `BATCH_JOB_EXECUTION_CONTEXT` table holds all information relevant to an Job's ExecutionContext. There is exactly one Job ExecutionContext per JobExecution, and it contains all of the job-level data that is needed for a particular job execution. This data typically represents the state that must be retrieved after a failure so that a JobInstance can 'start from where it left off'.

```

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

```

Below are descriptions for each column:

- `JOB_EXECUTION_ID`: Foreign key representing the JobExecution to which the context belongs. There may be more than one row associated to a given execution.
- `SHORT_CONTEXT`: A string version of the `SERIALIZED_CONTEXT`.
- `SERIALIZED_CONTEXT`: The entire context, serialized.

B.7 BATCH_STEP_EXECUTION_CONTEXT

The `BATCH_STEP_EXECUTION_CONTEXT` table holds all information relevant to an Step's `ExecutionContext`. There is exactly one `ExecutionContext` per `StepExecution`, and it contains all of the data that needs to be persisted for a particular step execution. This data typically represents the state that must be retrieved after a failure so that a `JobInstance` can 'start from where it left off'.

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

Below are descriptions for each column:

`STEP_EXECUTION_ID`: Foreign key representing the `StepExecution` to which the context belongs. There may be more than one row associated to a given execution.

`SHORT_CONTEXT`: A string version of the `SERIALIZED_CONTEXT`.

`SERIALIZED_CONTEXT`: The entire context, serialized.

B.8 Archiving

Because there are entries in multiple tables every time a batch job is run, it is common to create an archive strategy for the meta-data tables. The tables themselves are designed to show a record of what happened in the past, and generally won't affect the run of any job, with a couple of notable exceptions pertaining to restart:

- The framework will use the meta-data tables to determine if a particular `JobInstance` has been run before. If it has been run, and the job is not restartable, then an exception will be thrown.
- If an entry for a `JobInstance` is removed without having completed successfully, the framework will think that the job is new, rather than a restart.
- If a job is restarted, the framework will use any data that has been persisted to the `ExecutionContext` to restore the Job's state. Therefore, removing any entries from this table for jobs that haven't

completed successfully will prevent them from starting at the correct point if run again.

B.9 International and Multi-byte Characters

If you are using multi-byte character sets (e.g. Chines or Cyrillic) in your business processing, then those characters might need to be persisted in the Spring Batch schema. Many users find that simply changing the schema to double the length of the VARCHAR columns is enough. Others prefer to configure the JobRepository with max-varchar-length half the value of the VARCHAR column length is enough. Some users have also reported that they use NVARCHAR in place of VARCHAR in their schema definitions. The best result will depend on the database platform and the way the database server has been configured locally.

B.10 Recommendations for Indexing Meta Data Tables

Spring Batch provides DDL samples for the meta-data tables in the Core jar file for several common database platforms. Index declarations are not included in that DDL because there are too many variations in how users may want to index depending on their precise platform, local conventions and also the business requirements of how the jobs will be operated. The table below provides some indication as to which columns are going to be used in a WHERE clause by the Dao implementations provided by Spring Batch, and how frequently they might be used, so that individual projects can make up their own minds about indexing.

Table B.1. Where clauses in SQL statements (excluding primary keys) and their approximate frequency of use.

Default Table Name	Where Clause	Frequency
BATCH_JOB_INSTANCE	JOB_NAME = ? and JOB_KEY = ?	Every time a job is launched
BATCH_JOB_EXECUTION	JOB_INSTANCE_ID = ?	Every time a job is restarted
BATCH_EXECUTION_CONTEXT	EXECUTION_ID = ? and KEY_NAME = ?	On commit interval, a.k.a. chunk
		On commit interval,

BATCH_STEP_EXECUTION	VERSION = ?	a.k.a. chunk (and at start and end of step)
BATCH_STEP_EXECUTION	STEP_NAME = ? and JOB_EXECUTION_ID = ?	Before each step execution

附录C

- [附录C](#)
- [Appendix C. Batch Processing and Transactions](#)
 - [C.1 Simple Batching with No Retry](#)
 - [C.2 Simple Stateless Retry](#)
 - [C.3 Typical Repeat-Retry Pattern](#)

附录C

Appendix C. Batch Processing and Transactions

C.1 Simple Batching with No Retry

Consider the following simple example of a nested batch with no retries. This is a very common scenario for batch processing, where an input source is processed until exhausted, but we commit periodically at the end of a “chunk” of processing.

```
1. 1 | REPEAT(until=exhausted) {  
2. |  
3. 2 | TX {  
4. 3 | REPEAT(size=5) {  
5. 3.1 | input;  
6. 3.2 | output;  
7. | }  
8. | }  
9. |  
10. | }
```

The input operation (3.1) could be a message-based receive (e.g. JMS), or a file-based read, but to recover and continue processing with a chance of completing the whole job, it must be transactional. The same applies to the operation at (3.2) - it must be either transactional or idempotent.

If the chunk at REPEAT(3) fails because of a database exception at (3.2), then TX(2) will roll back the whole chunk.

C.2 Simple Stateless Retry

It is also useful to use a retry for an operation which is not transactional, like a call to a web-service or other remote resource. For example:

```

1. 0 | TX {
2. 1 |   input;
3. 1.1 |   output;
4. 2 |   RETRY {
5. 2.1 |     remote access;
6. |   }
7. | }
```

This is actually one of the most useful applications of a retry, since a remote call is much more likely to fail and be retryable than a database update. As long as the remote access (2.1) eventually succeeds, the transaction TX(0) will commit. If the remote access (2.1) eventually fails, then the transaction TX(0) is guaranteed to roll back.

C.3 Typical Repeat-Retry Pattern

The most typical batch processing pattern is to add a retry to the inner block of the chunk in the Simple Batching example. Consider this:

```

1. 1 | REPEAT(until=exhausted, exception=not critical) {
2. |
3. 2 |   TX {
4. 3 |     REPEAT(size=5) {
5. |
6. 4 |       RETRY(stateful, exception=deadlock loser) {
7. 4.1 |         input;
8. 5 |       } PROCESS {
9. 5.1 |         output;
10. 6 |     } SKIP and RECOVER {
11. |       notify;
12. |     }
13. |
14. |   }
15. | }
16. |
17. | }
```

The inner `RETRY(4)` block is marked as “stateful” - see the typical use case for a description of a stateful retry. This means that if the the retry `PROCESS(5)` block fails, the behaviour of the `RETRY(4)` is as follows.

- Throw an exception, rolling back the transaction `TX(2)` at the chunk level, and allowing the item to be re-presented to the input queue.
- When the item re-appears, it might be retried depending on the retry policy in place, executing `PROCESS(5)` again. The second and subsequent attempts might fail again and rethrow the exception.
- Eventually the item re-appears for the final time: the retry policy disallows another attempt, so `PROCESS(5)` is never executed. In this case we follow a `RECOVER(6)` path, effectively “skipping” the item that was received and is being processed.

Notice that the notation used for the `RETRY(4)` in the plan above shows explicitly that the the input step (4.1) is part of the retry. It also makes clear that there are two alternate paths for processing: the normal case is denoted by `PROCESS(5)`, and the recovery path is a separate block, `RECOVER(6)`. The two alternate paths are completely distinct: only one is ever taken in normal circumstances.

In special cases (e.g. a special `TransactionValidException` type), the retry policy might be able to determine that the `RECOVER(6)` path can be taken on the last attempt after `PROCESS(5)` has just failed, instead of waiting for the item to be re-presented. This is not the default behavior because it requires detailed knowledge of what has happened inside the `PROCESS(5)` block, which is not usually available - e.g. if the output included write access before the failure, then the exception should be rethrown to ensure transactional integrity.

The completion policy in the outer, `REPEAT(1)` is crucial to the success of the above plan. If the output(5.1) fails it may throw an exception (it usually does, as described), in which case the transaction `TX(2)` fails and the exception could propagate up through the outer batch `REPEAT(1)`. We do not want the whole batch to stop because the `RETRY(4)` might still be successful if we try again, so we add the `exception=not critical` to the outer `REPEAT(1)`.

Note, however, that if the `TX(2)` fails and we do try again, by virtue of the outer completion policy, the item that is next processed in the inner `REPEAT(3)` is not guaranteed to be the one that just failed. It might well be, but it depends on the implementation of the input(4.1). Thus the output(5.1) might fail again, on a new item, or on the old one. The client

of the batch should not assume that each `RETRY(4)` attempt is going to process the same items as the last one that failed. E.g. if the termination policy for `REPEAT(1)` is to fail after 10 attempts, it will fail after 10 consecutive attempts, but not necessarily at the same item. This is consistent with the overall retry strategy: it is the inner `RETRY(4)` that is aware of the history of each item, and can decide whether or not to have another attempt at it.

C.4 Asynchronous Chunk Processing

The inner batches or chunks in the typical example above can be executed concurrently by configuring the outer batch to use an `AsyncTaskExecutor`. The outer batch waits for all the chunks to complete before completing.

```

1. 1 | REPEAT(until=exhausted, concurrent, exception=not critical) {
2. |
3. 2 |   TX {
4. 3 |     REPEAT(size=5) {
5. |
6. 4 |       RETRY(stateful, exception=deadlock loser) {
7. 4.1 |         input;
8. 5 |       } PROCESS {
9. |         output;
10. 6 |       } RECOVER {
11. |         recover;
12. |     }
13. |
14. |   }
15. | }
16. |
17. | }

```

C.5 Asynchronous Item Processing

The individual items in chunks in the typical can also in principle be processed concurrently. In this case the transaction boundary has to move to the level of the individual item, so that each transaction is on a single thread:

```

1. 1 | REPEAT(until=exhausted, exception=not critical) {
2. |
3. 2 |   REPEAT(size=5, concurrent) {
4. |
5. 3 |     TX {
6. 4 |       RETRY(stateful, exception=deadlock loser) {
7. 4.1 |         input;
8. 5 |       } PROCESS {

```

```

9. |         output;
10. 6 |         } RECOVER {
11. |         recover;
12. |     }
13. | }
14. |
15. | }
16. |
17. | }

```

This plan sacrifices the optimisation benefit, that the simple plan had, of having all the transactional resources chunked together. It is only useful if the cost of the processing (5) is much higher than the cost of transaction management (3).

C.6 Interactions Between Batching and Transaction Propagation

There is a tighter coupling between batch-retry and TX management than we would ideally like. In particular a stateless retry cannot be used to retry database operations with a transaction manager that doesn't support NESTED propagation.

For a simple example using retry without repeat, consider this:

```

1. 1 | TX {
2. |
3. 1.1 | input;
4. 2.2 | database access;
5. 2 | RETRY {
6. 3 | TX {
7. 3.1 | database access;
8. | }
9. | }
10. |
11. | }

```

Again, and for the same reason, the inner transaction TX(3) can cause the outer transaction TX(1) to fail, even if the RETRY(2) is eventually successful.

Unfortunately the same effect percolates from the retry block up to the surrounding repeat batch if there is one:

```

1. 1 | TX {
2. |
3. 2 | REPEAT(size=5) {

```

```

4. 2.1 |     input;
5. 2.2 |     database access;
6. 3   |     RETRY {
7. 4   |         TX {
8. 4.1 |             database access;
9. |         }
10. |     }
11. | }
12. |
13. | }

```

Now if TX(3) rolls back it can pollute the whole batch at TX(1) and force it to roll back at the end.

What about non-default propagation?

- In the last example `PROPAGATION_REQUIRES_NEW` at TX(3) will prevent the outer TX(1) from being polluted if both transactions are eventually successful. But if TX(3) commits and TX(1) rolls back, then TX(3) stays committed, so we violate the transaction contract for TX(1). If TX(3) rolls back, TX(1) does not necessarily (but it probably will in practice because the retry will throw a roll back exception).
- `PROPAGATION_NESTED` at TX(3) works as we require in the retry case (and for a batch with skips): TX(3) can commit, but subsequently be rolled back by the outer transaction TX(1). If TX(3) rolls back, again TX(1) will roll back in practice. This option is only available on some platforms, e.g. not Hibernate or JTA, but it is the only one that works consistently.

So `NESTED` is best if the retry block contains any database access.

C.7 Special Case: Transactions with Orthogonal Resources

Default propagation is always OK for simple cases where there are no nested database transactions. Consider this (where the `SESSION` and `TX` are not global XA resources, so their resources are orthogonal):

```

1. 0   | SESSION {
2. 1   |     input;
3. 2   |     RETRY {
4. 3   |         TX {
5. 3.1 |             database access;
6. |         }
7. |     }
8. | }

```


Here there is a transactional message SESSION(0), but it doesn't participate in other transactions with PlatformTransactionManager, so doesn't propagate when TX(3) starts. There is no database access outside the RETRY(2) block. If TX(3) fails and then eventually succeeds on a retry, SESSION(0) can commit (it can do this independent of a TX block). This is similar to the vanilla "best-efforts-one-phase-commit" scenario - the worst that can happen is a duplicate message when the RETRY(2) succeeds and the SESSION(0) cannot commit, e.g. because the message system is unavailable.

C.8 Stateless Retry Cannot Recover

The distinction between a stateless and a stateful retry in the typical example above is important. It is actually ultimately a transactional constraint that forces the distinction, and this constraint also makes it obvious why the distinction exists.

We start with the observation that there is no way to skip an item that failed and successfully commit the rest of the chunk unless we wrap the item processing in a transaction. So we simplify the typical batch execution plan to look like this:

```

1. 0 | REPEAT(until=exhausted) {
2. |
3. 1 | TX {
4. 2 | REPEAT(size=5) {
5. |
6. 3 | RETRY(stateless) {
7. 4 | TX {
8. 4.1 | input;
9. 4.2 | database access;
10. | }
11. 5 | } RECOVER {
12. 5.1 | skip;
13. | }
14. |
15. | }
16. | }
17. |
18. | }

```

Here we have a stateless RETRY(3) with a RECOVER(5) path that kicks in after the final attempt fails. The "stateless" label just means that the block will be repeated without rethrowing any exception up to some limit. This will only work if the transaction TX(4) has propagation NESTED.

If the TX(3) has default propagation properties and it rolls back, it will pollute the outer TX(1). The inner transaction is assumed by the transaction manager to have corrupted the transactional resource, and so it cannot be used again.

Support for NESTED propagation is sufficiently rare that we choose not to support recovery with stateless retries in current versions of Spring Batch. The same effect can always be achieved (at the expense of repeating more processing) using the typical pattern above.

术语表

- 术语表(Glossary)
 - Spring Batch术语表
 - Batch 批
 - Batch Application Style (批处理程序风格)
 - Batch Processing (批处理任务)
 - Batch Window (批处理窗口)
 - Step (步骤)
 - Tasklet (小任务)
 - Batch Job Type (批处理作业类型)
 - Driving Query (驱动查询)
 - Item (数据项)
 - Logical Unit of Work (LUW, 逻辑工作单元)
 - Commit Interval (提交区间)
 - Partitioning (分块, 分区)
 - Staging Table (分段表, 阶段表)
 - Restartable (可再次启动的)
 - Rerunnable (可再次运行)
 - Repeat (重复)
 - Retry (重试)
 - Recover (恢复)
 - Skip (跳过)

术语表(Glossary)

Spring Batch术语表

Batch 批

An accumulation of business transactions over time.

随着时间累积而形成的一批业务事务。

Batch Application Style (批处理程序风格)

Term used to designate batch as an application style in its own right similar to online, Web or SOA. It has standard elements of input, validation, transformation of information to business model, business

processing and output. In addition, it requires monitoring at a macro level.

用来称呼批处理自身的程序风格的术语，类似于 online, web 或者 SOA。 其具有的标准元素包括： 输入、验证、将信息转换为业务模型、业务处理 以及 输出。此外,还需要在宏观层面上进行监控。

Batch Processing (批处理任务)

The handling of a batch of many business transactions that have accumulated over a period of time (e.g. an hour, day, week, month, or year). It is the application of a process, or set of processes, to many data entities or objects in a repetitive and predictable fashion with either no manual element, or a separate manual element for error processing.

积累了一定时间周期(比如小时、天、周、月或年)的业务事务归到一批进行处理。这种程序可以有一个进程,或者一组进程; 以重复可预测的方式处理很多数据实体/对象,要么没有人工干预,或者有些错误需要人工单独进行处理。

Batch Window (批处理窗口)

The time frame within which a batch job must complete. This can be constrained by other systems coming online, other dependent jobs needing to execute or other factors specific to the batch environment.

批处理作业必须在这个时间范围内完成。 可能受到的制约包括： 其他系统要上线, 相关作业要执行, 或者是特定于批处理环境的其他因素 。

Step (步骤)

It is the main batch task or unit of work controller. It initializes the business logic, and controls the transaction environment based on commit interval setting, etc.

这是主要的批处理任务, 或者工作控制器的组成单元。在其中进行业务逻辑初始化, 基于提交间隔控制事务环境, 等等。

Tasklet (小任务)

A component created by application developer to process the business logic for a Step.

由程序员创建的组件，用来处理某个 step 中的业务逻辑。

Batch Job Type (批处理作业类型)

Job Types describe application of jobs for particular type of processing. Common areas are interface processing (typically flat files), forms processing (either for online pdf generation or print formats), report processing.

作业类型描述特定类型的作业处理程序。共同领域包括 接口处理(通常是平面文件)，格式处理(如在线生成pdf或打印格式)，报表处理等。

Driving Query (驱动查询)

A driving query identifies the set of work for a job to do; the job then breaks that work into individual units of work. For instance, identify all financial transactions that have a status of "pending transmission" and send them to our partner system. The driving query returns a set of record IDs to process; each record ID then becomes a unit of work. A driving query may involve a join (if the criteria for selection falls across two or more tables) or it may work with a single table.

一次驱动查询用来标识一个作业要做的工作组；然后工作被打散为单个的工作单元。例如，找出所有状态为“等待传输”的金融交易 并发送给合作伙伴系统。驱动查询返回要处理的记录的ID集合；每个记录ID 稍后都会成为一个工作单元。一次驱动查询可能涉及 join连接(如果条件遇到两个或多个表)，也可能只使用单个表。

Item (数据项)

An item represents the smallest amount of complete data for processing. In the simplest terms, this might mean a line in a file, a row in a database table, or a particular element in an XML file.

一个 item 代表要处理的最小的完整的数据。最简单的理解，可以是文件中的一行(line)，数据表中的一行(row)，或者XML文件中一个特定的元素(element)。

Logical Unit of Work (LUW, 逻辑工作单元)

原文可能错了, `Logicial`

A batch job iterates through a driving query (or another input source such as a file) to perform the set of work that the job must accomplish. Each iteration of work performed is a unit of work.

批处理作业通过驱动查询(或者是文件之类的输入源)来迭代执行必须完成的工作。工作执行中的每次迭代就是一个工作单元。

Commit Interval (提交区间)

A set of LUWs processed within a single transaction.

在单个事务中处理的 逻辑工作单元集合。

Partitioning (分块, 分区)

Splitting a job into multiple threads where each thread is responsible for a subset of the overall data to be processed. The threads of execution may be within the same JVM or they may span JVMs in a clustered environment that supports workload balancing.

将一个作业拆分给多个线程来执行, 每个线程只负责处理整个数据中的一部分。这些线程可能在同一个JVM中执行, 也可能跨越JVM在支持工作负载平衡的集群环境中运行。

Staging Table (分段表, 阶段表)

A table that holds temporary data while it is being processed.

一个存储临时数据的表, 里面的数据即将被处理。

Restartable (可再次启动的)

A job that can be executed again and will assume the same identity as when run initially. In otherwords, it is has the same job instance id.

可再次执行的作业, 而且再次执行时, 和初次运行具有同样的身份。换言之, 两者具有相同的作业实例 id。

Rerunnable (可再次运行)

A job that is restartable and manages its own state in terms of previous run's record processing. An example of a rerunnable step is one based on a driving query. If the driving query can be formed so that it will limit the processed rows when the job is restarted than it is re-runnable. This is managed by the application logic. Often times a condition is added to the where statement to limit the rows returned by the driving query with something like "and processedFlag != true".

可再次启动的作业，并且可根据之前的处理记录来合理调整自身的状态。可再次运行 Step 的一个例子是基于 driving query 的部分。如果是 re-runnable 的，那么当 restarted 后，driving query 就会排除已经处理过的那些行。当然这由应用程序逻辑决定。通常是在 where 子句中添加条件来限制查询返回的结果，例如 “processedFlag != true”。

Repeat (重复)

One of the most basic units of batch processing, that defines repeatability calling a portion of code until it is finished, and while there is no error. Typically a batch process would be repeatable as long as there is input.

批处理最基本的单元之一，定义了可重复调用的一部分代码，直到完成某个任务为止，如果不出错的话。通常来说，只要还有输入数据，批处理过程就会一直重复。

Retry (重试)

Simplifies the execution of operations with retry semantics most frequently associated with handling transactional output exceptions. Retry is slightly different from repeat, rather than continually calling a block of code, retry is stateful, and continually calls the same block of code with the same input, until it either succeeds, or some type of retry limit has been exceeded. It is only generally useful if a subsequent invocation of the operation might succeed because something in the environment has improved.

简化了的重试语义通常和事务输出异常处理有关。重试(Retry)和重复(Repeat)略有不同，不仅仅是持续不断地调用某个代码块，因为重试是有状态的，所以每次都是使用相同的输入，直到成功为止，或者是已经超过了某种类型的重试限制。一般只有在依赖某种外部环境的情况下，如果外部环境得到改善，就会使得后续操作会成功的情况下就会很有用。

Recover (恢复)

Recover operations handle an exception in such a way that a repeat process is able to continue.

恢复操作用来对付异常，通过这种方式使重复过程得以继续下去。

Skip (跳过)

Skip is a recovery strategy often used on file input sources as the strategy for ignoring bad input records that failed validation.

跳过是一种容错策略，通常在读取文件输入时，用来忽略验证失败的脏数据。