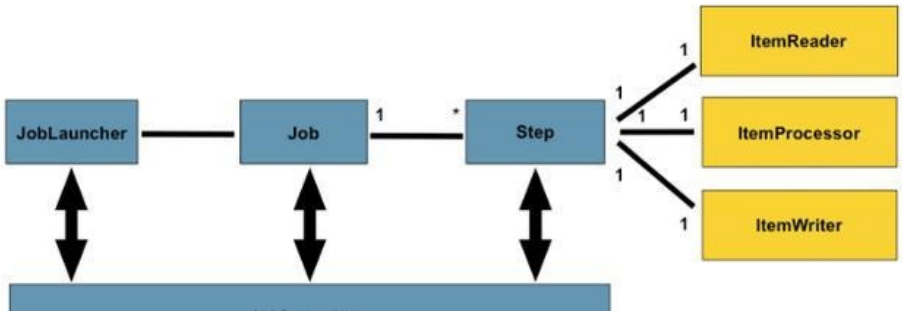
Springbatch批量处理框架最佳实践

spring batch精选，一文吃透spring batch批量处理框架



前言碎语

批处理是企业级业务系统不可或缺的一部分，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.SpringBatch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具有高可扩展性的框架,简单的批处理,复杂的大数据批处理作业都可以通过SpringBatch框架来实现。

下面援引《SpringBatch批处理框架》一书作者刘相的一篇文章，分四个步骤来阐述springbatch的方方面面

初识批处理典型场景

探秘领域模型及关键架构

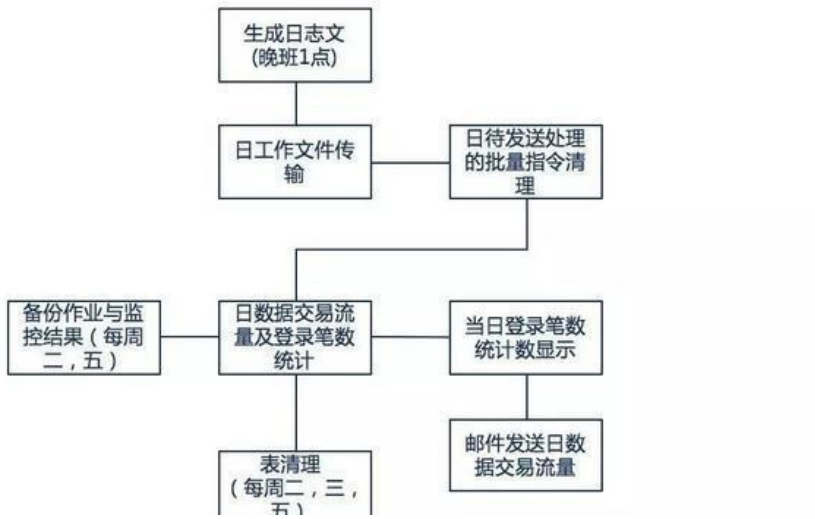
实现作业健壮性与扩展性

批处理框架的不足与增强

批处理典型业务场景

对账是典型的批处理业务处理场景，各个金融机构的往来业务和跨主机系统的业务都会涉及到对账的过程，如大小额支付、银联交易、人行往来、现金管理、POS业务、ATM业务、证券公司资金账户、证券公司与证券结算公司。

下面是某行网银的部分日终跑批实例场景需求。



涉及到的需求点包括：

批量的每个单元都需要错误处理和回退；

每个单元在不同平台中运行；

需要有分支选择；

每个单元需要监控和获取单元处理日志；

提供多种触发规则，按日期，日历，周期触发；

除此之外典型的批处理适用于如下的业务场景：

定期提交批处理任务（日终处理）

并行批处理：并行处理任务

企业消息驱动处理

大规模的并行处理

手动或定时重启

按顺序处理依赖的任务(可扩展为工作流驱动的批处理)

部分处理：忽略记录(例如在回滚时)

完整的批处理事务

与OLTP类型交易不同，批处理作业两个典型特征是批量执行与自动执行（需要无人值守）：前者能够处理大批量数据的导入、导出和业务逻辑计算；后者无需人工干预，能够自动化执行批量任务。

在关注其基本功能之外，还需要关注如下的几点：

健壮性：不会因为无效数据或错误数据导致程序崩溃；

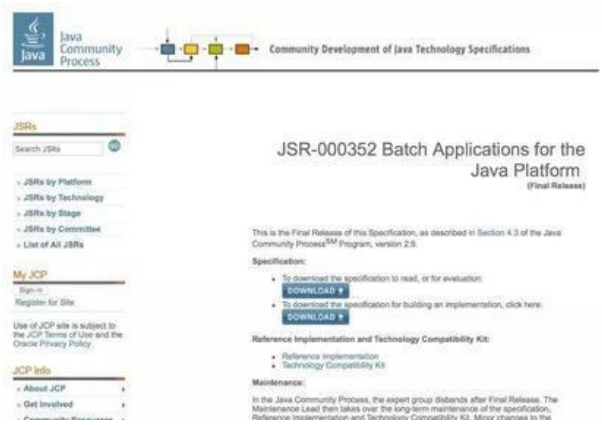
可靠性：通过跟踪、监控、日志及相关的处理策略（重试、跳过、重启）实现批作业的可靠执行；

扩展性：通过并发或者并行技术实现应用的纵向和横向扩展，满足海量数据处理的性能需求；

苦于业界真的缺少比较好的批处理框架，Spring Batch是业界目前为数不多的优秀批处理框架（Java语言开发），SpringSource和Accenture（埃森哲）共同贡献了智慧。

Accenture在批处理架构上有着丰富的工业级别的经验，贡献了之前专用的批处理体系框架（这些框架历经数十年研发和使用，为Spring Batch提供了大量的参考经验）。

SpringSource则有着深刻的技术认知和Spring框架编程模型，同时借鉴了JCL(Job Control Language)和COBOL的语言特性。2013年JSR-352将批处理纳入规范体系，并被包含在了JEE7之中。这意味着，所有的JEE7应用服务器都会有批处理的能力，目前第一个实现此规范的应用服务器是Glassfish 4。当然也可以在Java SE中使用。



Spring Batch	JSR-352
Job	Job
Step	Step
Chunk	Chunk
Item	Item
ItemReader	ItemReader
ItemProcessor	ItemProcessor
ItemWriter	ItemWriter
JobInstance	JobInstance
JobExecution	JobExecution
StepExecution	StepExecution
JobExecutionListeners	JobListeners
StepExecutionListeners	StepListeners

通过Spring Batch框架可以构建出轻量级的健壮的并行处理应用,支持事务、并发、流程、监控、纵向和横向扩展,提供统一的接口管理和任务管理。

框架提供了诸如以下的核心能力，让大家更关注在业务处理上。更是提供了如下的丰富能力：

明确分离批处理的执行环境和应用

将通用核心的服务以接口形式提供

提供“开箱即用” 的简单的默认的核心执行接口

提供Spring框架中配置、自定义、和扩展服务

所有默认实现的核心服务能够容易的被扩展与替换，不会影响基础层

提供一个简单的部署模式，使用Maven进行编译

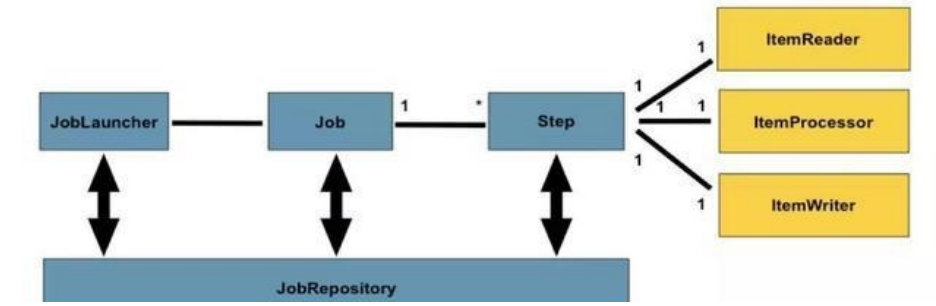
批处理关键领域模型及关键架构

先来个Hello World示例，一个典型的批处理作业。

```
<!-- 账单作业 -->
<job id="billJob">
    <step id="billStep">
        <tasklet transaction-manager="transactionManager">
            <chunk reader="csvItemReader" writer="csvItemWriter"
                processor="creditBillProcessor" commit-interval="2">
            </chunk>
        </tasklet>
    </step>
</job>
```

典型的一个作业分为3部分：作业读、作业处理、作业写，也是典型的三步式架构。整个批处理框架基本上围绕Read、Process、Writer来处理。除此之外，框架提供了作业调度器、作业仓库（用以存放Job的元数据信息，支持内存、DB两种模式）。

完整的领域概念模型参加下图：



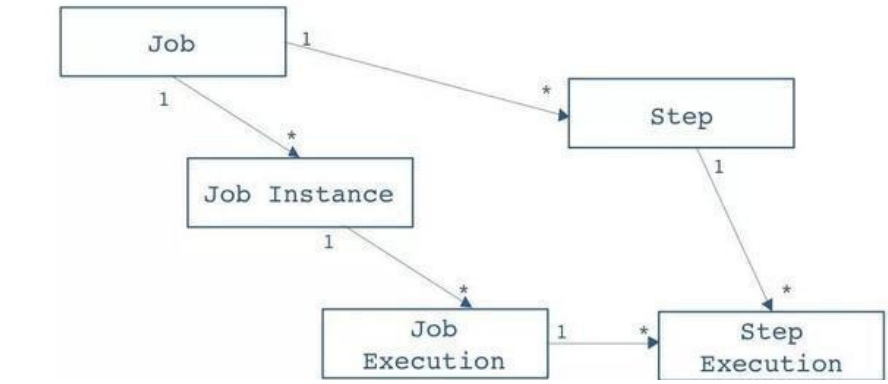
Job Launcher（作业调度器）是Spring Batch框架基础设施层提供的运行Job的能力。通过给定的Job名称和作Job Parameters，可以通过Job Launcher执行Job。

通过Job Launcher可以在Java程序中调用批处理任务，也可以在通过命令行或者其它框架（如定时调度框架Quartz）中调用批处理任务。

Job Repository来存储Job执行期的元数据（这里的元数据是指Job Instance、Job Execution、Job Parameters、Step Execution、Execution Context等数据），并提供两种默认实现。

一种是存放在内存中；另一种将元数据存放在数据库中。通过将元数据存放在数据库中，可以随时监控批处理Job的执行状态。Job执行结果是成功还是失败，并且使得在Job失败的情况下重新启动Job成为可能。Step表示作业中的一个完整步骤，一个Job可以有一个或者多个Step组成。

批处理框架运行期的模型也非常简单：



Job Instance（作业实例）是一个运行期的概念，Job每执行一次都会涉及到一个Job Instance。

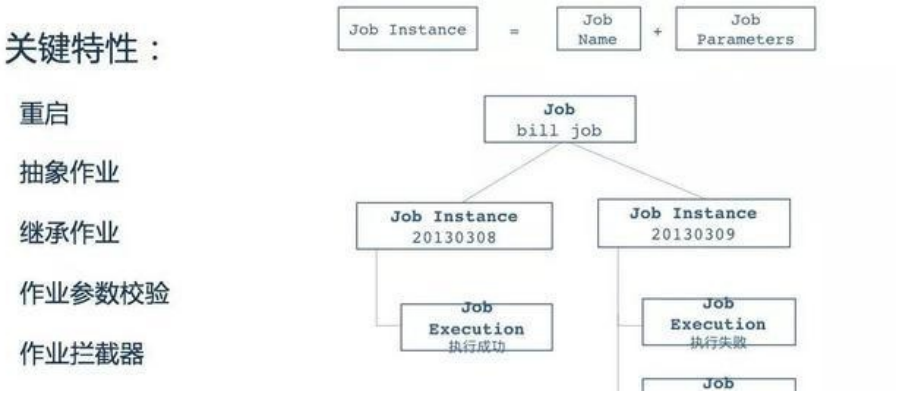
Job Instance来源可能有两种：一种是根据设置的Job Parameters从Job Repository（作业仓库）中获取一个；如果根据Job Parameters从Job Repository没有获取Job Instance，则新创建一个新的Job Instance。

Job Execution表示Job执行的句柄，一次Job的执行可能成功也可能失败。只有Job执行成功后，对应的Job Instance才会被完成。因此在Job执行失败的情况下，会有一个Job Instance对应多个Job Execution的场景发生。

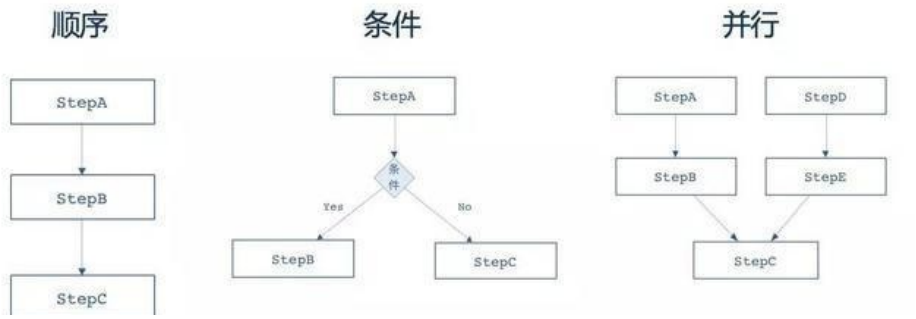
总结下批处理的典型概念模型，其设计非常精简的十个概念，完整支撑了整个框架。

关键词	描述
Job repository	基础组件，用用来持久化Job的元数据，默认使用用内存
Job launcher	基础组件，用用来启动Job
Job	应用用组件，是Batch操作的基础执行行单元
Step	Job的一个阶段，Job由一组Step构成
Tasklet	Step的一个事务过程，包含重复执行、同步、异步等策略
Item	从数据源读出或写入入的一一条数据记录
Chunk	给定数量的Item的集合
Item Reader	从给定的数据源读取Item集合
Item Processor	在Item写入入数据源之前进行行数据清洗（转换校验过滤...）

Job提供的核心能力包括作业的抽象与继承，类似面向对象中的概念。对于执行异常的作业，提供重启的能力。



框架在Job层面，同样提供了作业编排的概念，包括顺序、条件、并行作业编排。

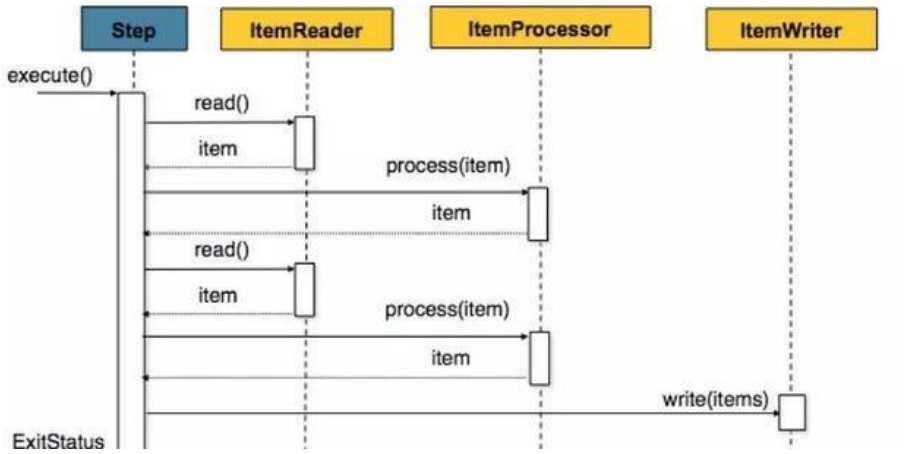


在一个Job中配置多个Step。不同的Step间可以顺序执行，也可以按照不同的条件有选择的执行（条件通常使用Step的退出状态决定），通过next元素或者decision元素来定义跳转规则；

为了提高多个Step的执行效率，框架提供了Step并行执行的能力（使用split进行声明，通常该情况下需要Step之间没有任何的依赖关系，否则容易引起业务上的错误）。Step包含了一个实际运行的批处理任务中的所有必需的信息，其实现可以是非常简单的业务实现，也可以是非常复杂的业务处理，Step的复杂程度通常是业务决定的。

每个Step由ItemReader、ItemProcessor、ItemWriter组成，当然根据不同的业务需求，ItemProcessor可以做适当的精简。同时框架提供了大量的ItemReader、ItemWriter的实现，提供了对FlatFile、XML、Json、DataBase、Message等多种数据类型的支持。

框架还为Step提供了重启、事务、重启次数、并发数；以及提交间隔、异常跳过、重试、完成策略等能力。基于Step的灵活配置，可以完成常见的业务功能需求。其中三步走（Read、Processor、Writer）是批处理中的经典抽象。



作为面向批的处理，在Step层提供了多次读、处理，一次提交的能力。

在Chunk的操作中，可以通过属性commit-interval设置read多少条记录后进行一次提交。通过设置commit-interval的间隔值，减少提交频次，降低资源使用率。Step的每一次提交作为一个完整的事务存在。默认采用Spring提供的声明式事务管理模式，事务编排非常方便。如下是一个声明事务的示例：

```
<job id="billJob">
    <step id="billStep">
        <tasklet transaction-manager="transactionManager" >
            <chunk reader="csvItemReader" writer="csvItemWriter"
                processor="creditBillProcessor" commit-interval="2" >
            </chunk>
            <transaction-attributes isolation="DEFAULT" timeout="200" propagation="REQUIRED" />
            <no-rollback-exception-classes >
                <include class="java.lang.NullPointerException"/>
            </no-rollback-exception-classes>
        </tasklet>
    </step>
</job>
    <!-- 事务管理器 -->
<bean:bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
```

框架对于事务的支持能力包括：

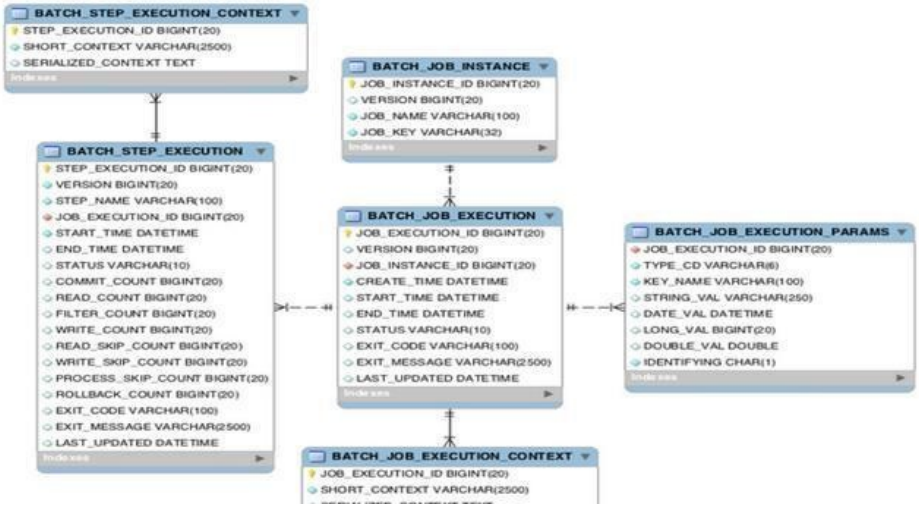
Chunk支持事务管理，通过commit-interval设置每次提交的记录数；

支持对每个Tasklet设置细粒度的事务配置：隔离界别、传播行为、超时；

支持rollback和no rollback，通过skippable-exception-classes和no-rollback-exception-classes进行支撑；

支持JMS Queue的事务级别配置；

另外，在框架资深的模型抽象方面，Spring Batch也做了极为精简的抽象。



仅仅使用六张业务表存储了所有的元数据信息（包括Job、Step的实例，上下文，执行器信息，为后续的监控、重启、重试、状态恢复等提供了可能）。

BATCH_JOB_INSTANCE：作业实例表，用于存放Job的实例信息

BATCH_JOB_EXECUTION_PARAMS：作业参数表，用于存放每个Job执行时候的参数信息，该参数实际对应Job实例的。

BATCH_JOB_EXECUTION：作业执行器表，用于存放当前作业的执行信息，比如创建时间，执行开始时间，执行结束时间，执行的那个Job实例，执行状态等。

BATCH_JOB_EXECUTION_CONTEXT：作业执行上下文表，用于存放作业执行器上下文的信息。

BATCH_STEP_EXECUTION：作业步执行器表，用于存放每个Step执行器的信息，比如作业步开始执行时间，执行完成时间，执行状态，读写次数，跳过次数等信息。

BATCH_STEP_EXECUTION_CONTEXT：作业步执行上下文表，用于存放每个作业步上下文的信息。

实现作业的健壮性与扩展性

批处理要求Job必须有较强的健壮性，通常Job是批量处理数据、无人值守的，这要求在Job执行期间能够应对各种发生的异常、错误，并对Job执行进行有效的跟踪。

一个健壮的Job通常需要具备如下的几个特性：

\1. 容错性

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

\2. 可追踪性

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

\3. 可重启性

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

✧Skip 在出现异常的情况下保证主体程序正常运行
✧Retry 当发生生瞬态失败的时候进行重试
✧Restart 在最后执行失败的地重启Job

特性	功能	适用时机	适用场景
Skip	跳过错误的记录行，保证Job能够正确的执行	适用于非致命的异常	面向Chunk的Step
Retry	重试给定的操作，比如短暂的网络异常、并发异常等	适用于短暂的异常，经过重试之后该异常可能会不再重现	面向Chunk的Step或者应用代码

框架提供了支持上面所有能力的特性，包括Skip（跳过记录处理）、Retry（重试给定的操作）、Restart（从错误点开始重新启动失败的Job）：

Skip，在对数据处理期间，如果数据的某几条的格式不能满足要求，可以通过Skip跳过该行记录的处理，让Processor能够顺利的处理其余的记录行。

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

Restart，在Job执行失败后，可以通过重启功能来继续完成Job的执行。在重启时候，批处理框架允许在上次执行失败的点重新启动Job，而不是从头开始执行，这样可以大幅提高Job执行的效率。

对于扩展性，框架提供的扩展能力包括如下的四种模式：

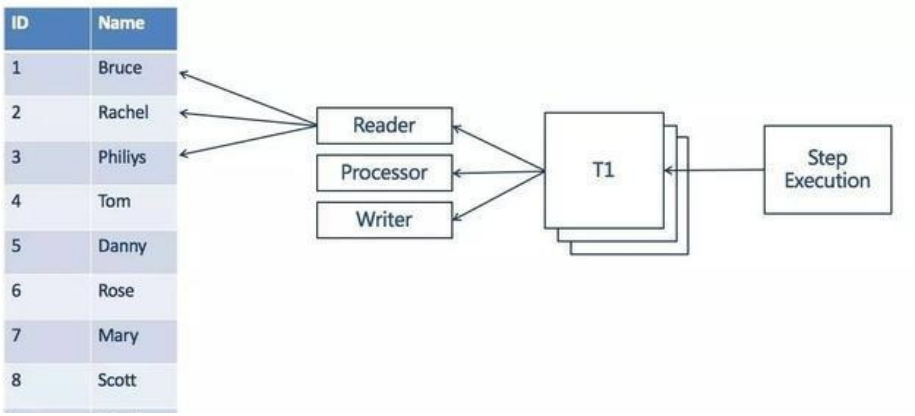
Multithreaded Step 多线程执行一个Step;

Parallel Step 通过多线程并行执行多个Step;

Remote Chunking 在远端节点上执行分布式Chunk操作;

Partitioning Step 对数据进行分区，并分开执行;

我们先来看第一种的实现Multithreaded Step:



批处理框架在Job执行时默认使用单个线程完成任务的执行，同时框架提供了线程池的支持（Multithreaded Step模式），可以在Step执行时候进行并行处理，这里的并行是指同一个Step使用线程池进行执行，同一个Step被并行的执行。使用tasklet的属性task-executor可以非常容易的将普通的Step变成多线程Step。

Multithreaded Step的实现示例:

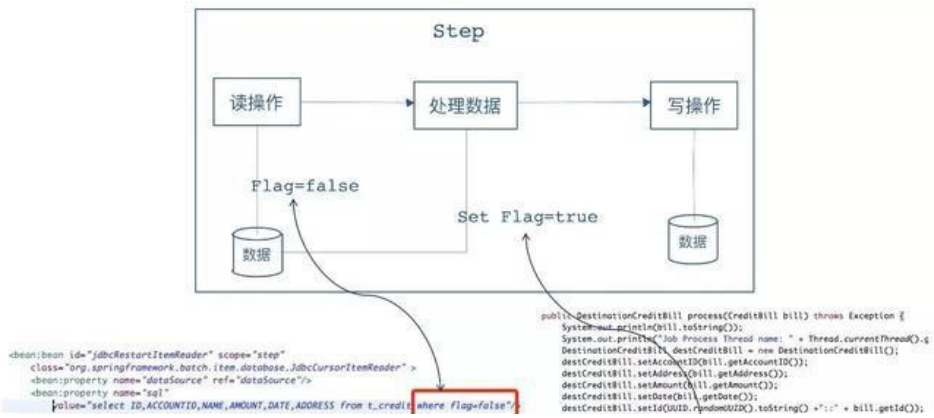
```
<job id="multiThreadJob">
    <step id="multiThreadStep">
      <tasklet task-executor="taskExecutor" throttle-limit="6">
            <chunk reader="reader" writer="writer" commit-interval="2"/>
      </tasklet>
    </step>
</job>

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

需要注意的是Spring Batch框架提供的大部分的ItemReader、ItemWriter等操作都是线程不安全的。

可以通过扩展的方式显现线程安全的Step。

下面为大家展示一个扩展的实现:



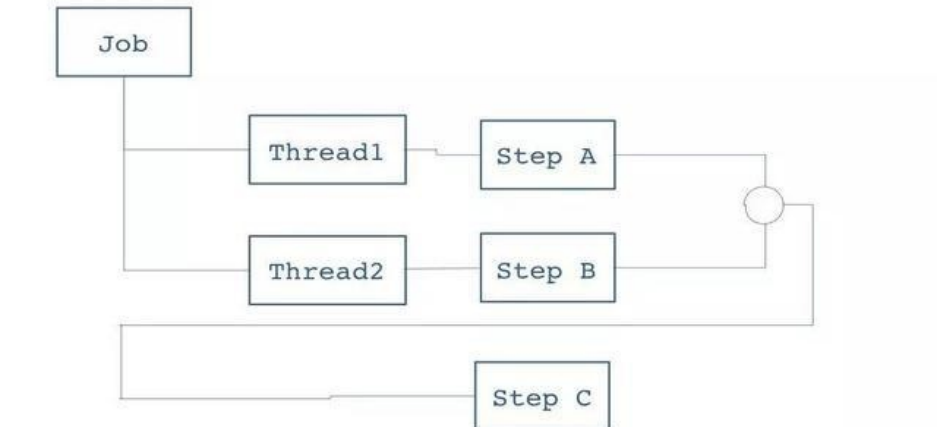
需求：针对数据表的批量处理，实现线程安全的Step，并且支持重启能力，即在执行失败点可以记录批处理的状态。

对于示例中的数据库读取组件JdbcCursorItemReader，在设计数据库表时，在表中增加一个字段Flag，用于标识当前的记录是否已经读取并处理成功，如果处理成功则标识Flag=true，等下次重新读取的时候，对于已经成功读取且处理成功的记录直接跳过处理。

Multithreaded Step（多线程步）提供了多个线程执行一个Step的能力，但这种场景在实际的业务中使用的并不是非常多。

更多的业务场景是Job中不同的Step没有明确的先后顺序，可以在执行期并行的执行。

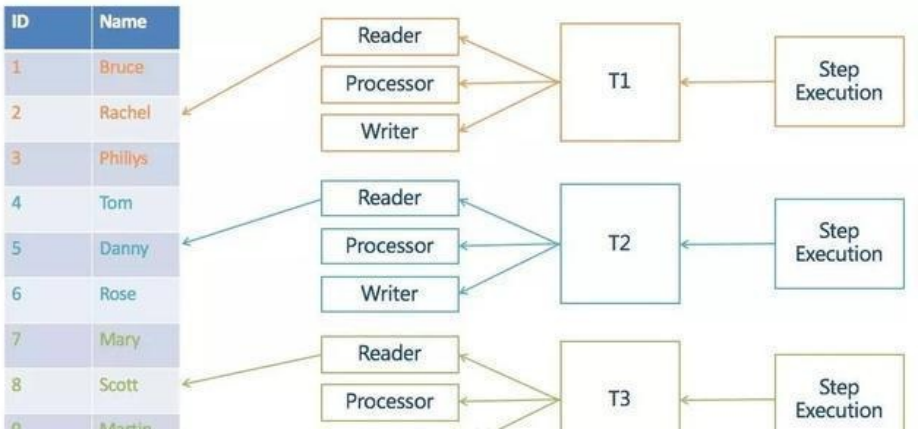
Parallel Step：提供单个节点横向扩展的能力



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

框架提供了并行Step的能力。可以通过Split元素来定义并行的作业流，并制定使用的线程池。

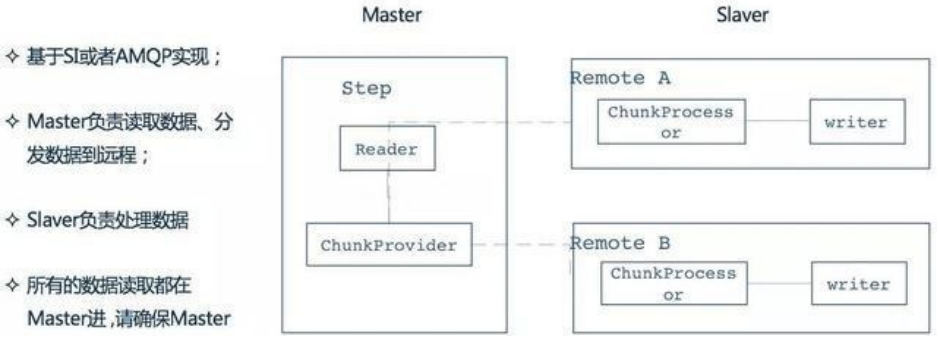
Parallel Step模式的执行效果如下：



每个作业步并行处理不同的记录，示例中三个作业步，处理同一张表中的不同数据。

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

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



远程分块是一个把step进行技术分割的工作，不需要对处理数据的结构有明确了解。

任何输入源能够使用单进程读取并在动态分割后作为"块"发送给远程的工作进程。

远程进程实现了监听者模式，反馈请求、处理数据最终将处理结果异步返回。请求和返回之间的传输会被确保在发送者和单个消费者之间。

在Master节点，作业步负责读取数据，并将读取的数据通过远程技术发送到指定的远端节点上，进行处理，处理完毕后Master负责回收Remote端执行的情况。

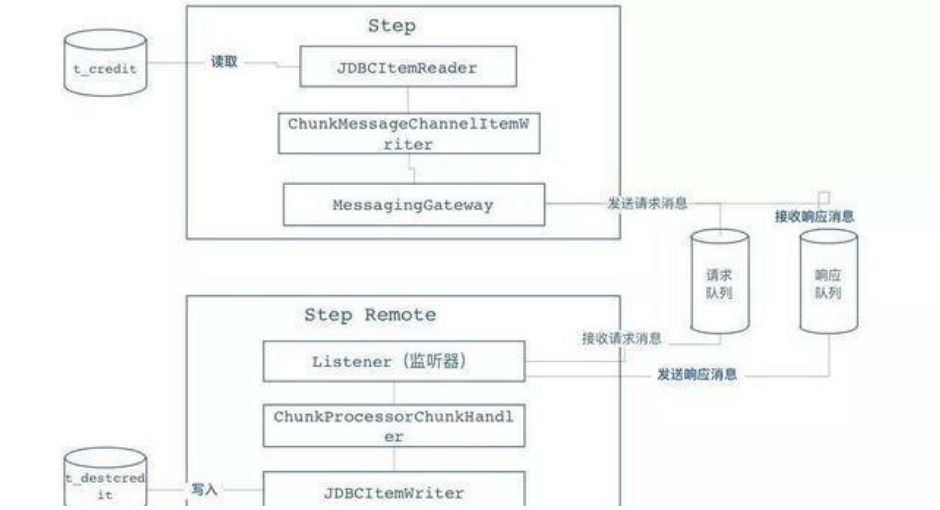
在Spring Batch框架中通过两个核心的接口来完成远程Step的任务，分别是ChunkProvider与ChunkProcessor。

ChunkProvider：根据给定的ItemReader操作产生批量的Chunk操作；

ChunkProcessor：负责获取ChunkProvider产生的Chunk操作，执行具体的写逻辑；

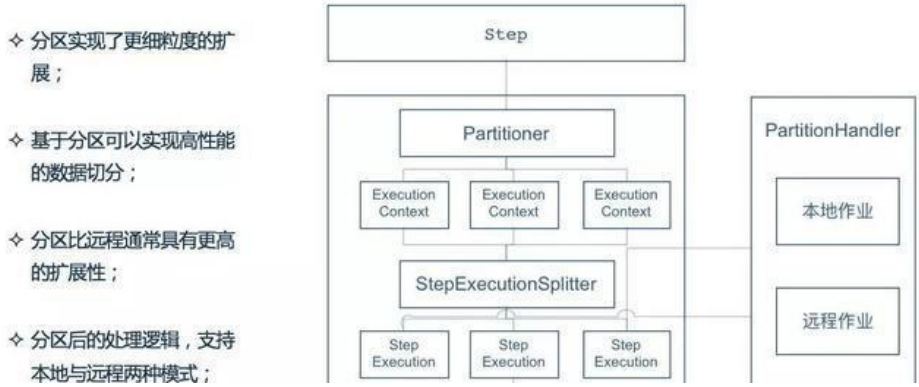
Spring Batch中对远程Step没有默认的实现，但我们可以借助SI或者AMQP实现来实现远程通讯能力。

基于SI实现Remote Chunking模式的示例：



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

接下来我们看下最后一种分区模式；Partitioning Step：分区模式需要对数据的结构有一定的了解，如主键的范围、待处理的文件的名字等。



这种模式的优点在于分区中每一个元素的处理器都能够像一个普通Spring Batch任务的单步一样运行，也不必去实现任何特殊的或是新的模式，来让他们能够更容易配置与测试。

通过分区可以实现以下的优点：

分区实现了更细粒度的扩展；

基于分区可以实现高性能的数据切分；

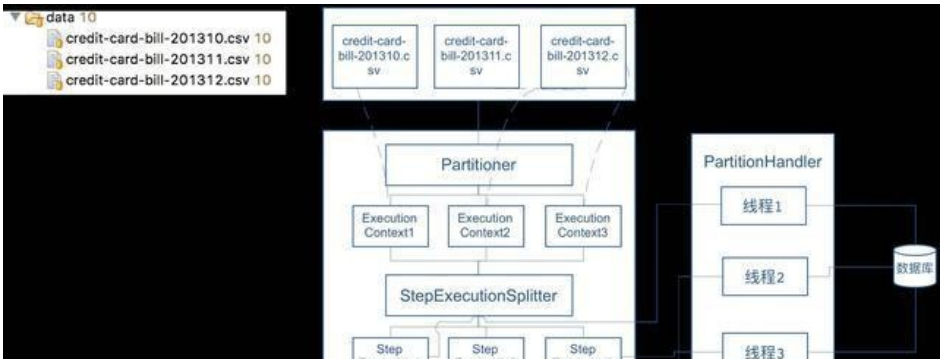
分区比远程通常具有更高的扩展性；

分区后的处理逻辑，支持本地与远程两种模式；

分区作业典型的可以分成两个处理阶段，数据分区、分区处理；

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

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



Spring Batch框架提供了对文件分区的支持，实现类org.springframework.batch.core.partition.support.MultiResourcePartitioner提供了对文件分区的默认支持，根据文件名将不同的文件处理进行分区，提升处理的速度和效率，适合有大量小文件需要处理的场景。

```
<job id="partitionJob">
    <step id="partitionStep">
        <partition step="partitionReadWriteStep" partitioner="partitioner">
            <handler grid-size="2" task-executor="taskExecutor"/>
        </partition>
    </step>
</job>

<step id="partitionReadWriteStep">
    <tasklet>
        <chunk reader="flatFileItemReader" writer="jdbcItemWriter"
            processor="creditBillProcessor" commit-interval="2" />
    </tasklet>
</step>

<bean:bean id="partitioner"
    class="org.springframework.batch.core.partition.support.MultiResourcePartitioner">
    <bean:property name="keyName" value="fileName"/>
    <bean:property name="resources" value="classpath:/ch11/data/*.csv"/>
</bean:bean>

<bean:bean id="flatFileItemReader" scope="step"
    class="org.springframework.batch.item.file.FlatFileItemReader">
    <bean:property name="resource"
        value="#{stepExecutionContext[fileName]}"/
```

示例展示了将不同文件分配到不同的作业步中，使用MultiResourcePartitioner进行分区，意味着每个文件会被分配到一个不同的分区中。如果有其它的分区规则，可以通过实现接口Partitioner来进行自定义的扩展。有兴趣的TX，可以自己实现基于数据库的分区能力哦。

总结一下，批处理框架在扩展性上提供了4中不同能力，每种都是各自的使用场景，我们可以根据实际的业务需要进行选择。

✧Multithreaded Step 多线程执行一个Step

✧Parallel Step 通过多线程并行执行多个Step

✧Remote Chunking 在远端节点上执行分布式Chunk操作

✧Partitioning Step 对数据进行分区，并分开执行

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

批处理框架的不足与增强

Spring Batch批处理框架虽然提供了4种不同的监控方式，但从目前的使用情况来看，都不是非常的友好。

通过DB直接查看，对于管理人员来讲，真的不忍直视；

通过API实现自定义的查询，这是程序员的天堂，确实运维人员的地狱；

提供了Web控制台，进行Job的监控和操作，目前提供的功能太裸露，无法直接用于生产；

提供JMX查询方式，对于非开发人员太不友好；

但在企业级应用中面对批量数据处理，仅仅提供批处理框架仅能满足批处理作业的快速开发、执行能力。

企业需要统一的批处理平台来处理复杂的企业批处理应用，批处理平台需要解决作业的统一调度、批处理作业的集中管理和管控、批处理作业的统一监控等能力。

那完美的解决方案是什么呢?

企业级批处理平台需要在Spring Batch批处理框架的基础上，集成调度框架，通过调度框架可以将任务按照企业的需求进行任务的定期执行；

丰富目前Spring Batch Admin（Spring Batch的管理监控平台，目前能力比较薄弱）框架，提供对Job的统一管理功能，增强Job作业的监控、预警等能力；

通过与企业的组织机构、权限管理、认证系统进行合理的集成，增强平台对Job作业的权限控制、安全管理能力。