工程师/胡顺强 2013-01-23

目录

，引言

1，什么是批处理

在现代企业应用当中，面对复杂的业务以及海量的数据，除了通过庞杂的人机交互界面进行各种处理外，还有一类工作，不需要人工干预，只需要定期读入大批量数据，然后完成相应业务处理并进行归档。这类工作即为“批处理”。

从上面的描述可以看出，批处理应用有如下几个特点：

- 数据量大，少则百万，多则上亿的数量级。
- 不需要人工干预，由系统根据配置自动完成。
- 与时间相关，如每天执行一次或每月执行一次。

同时，批处理应用又明显分为三个环节：

- 读数据，数据可能来自文件、数据库或消息队列等
- 数据处理，如电信支撑系统的计费处理
- 写数据，将输出结果写入文件、数据库或消息队列等

因此，从系统架构上，应重点考虑批处理应用的事务粒度、日志监控、执行、资源管理（尤其存在并发的情况下）。从系统设计上，应重点考虑数据读写与业务处理的解耦，提高复用性以及可测试性。

2，什么是 Spring Batch

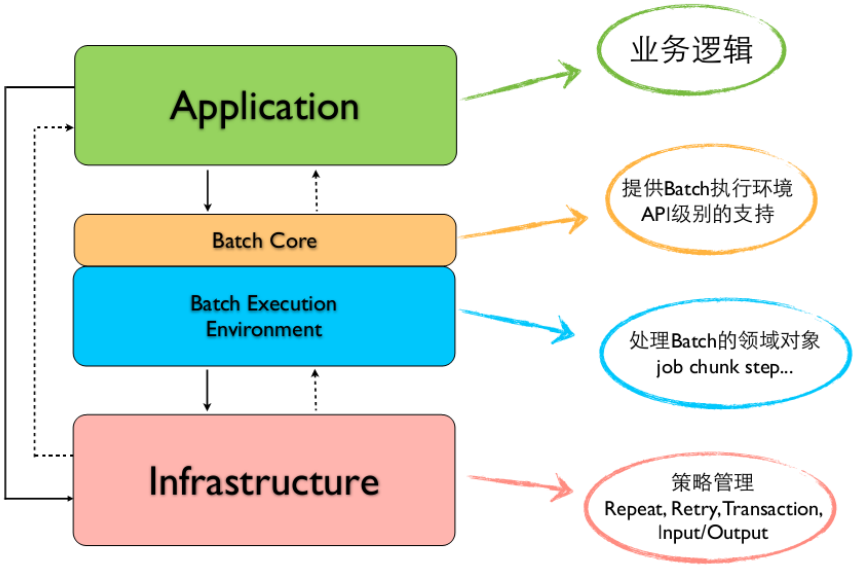
- SpringSource 与 Accenture 合作开发了 Spring Batch
- Accenture 在批处理架构上有着丰富的工业级别的经验，SpringSource 则有着深刻的技术认知和 Spring 框架编程模型
- Accenture 贡献了之前专用的批处理体系框架，这些框架历经数十年研发和使用，为 Spring Batch 提供了大量的参考经验
- **Spring Batch 是一款优秀的、开源的大数据量并行处理框架。通过 Spring Batch 可以构建出轻量级的健壮的并行处理应用，支持事务、并发、流程、监控，提供统一的接口管理和任务管理。**
- 另外 Spring Batch 是一款批处理应用框架，不是调度框架。它只关注批处理任务相关的问题，如事务、并发、监控、执行等，并不提供相应的调度功能。因此，如果我们希望批处理任务定期执行，

可结合 Quartz 等成熟的调度框架实现

上图寓意：火车通行处理，在很多火车都要通过该站台的时候，我们无需耗费大量的人力资源，人工协调处理。

二，Spring Batch 结构

1，Spring Batch 体系结构



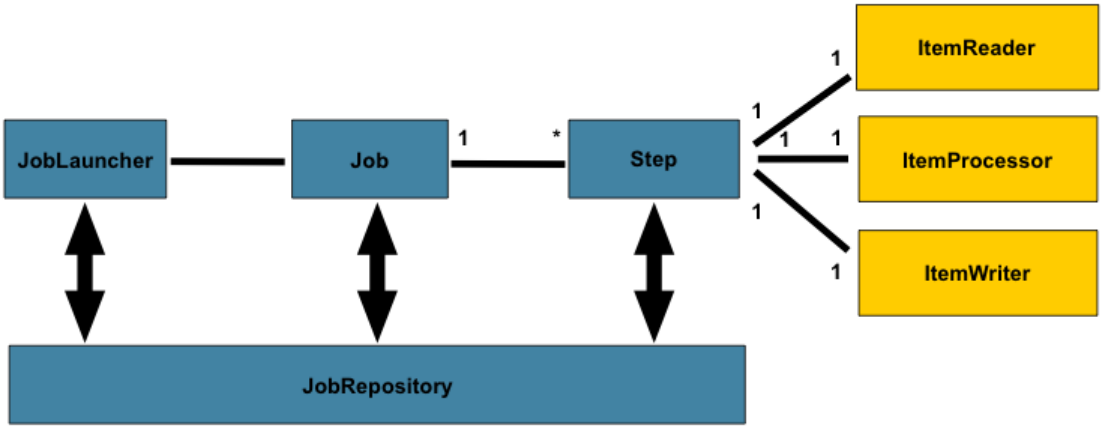
12年12月6日星期四

这种分层结构有三个重要的组成部分：应用层、核心层、基础架构层。应用层包含所有的批处理作业，通过 Spring 框架管理程序员自定义的代码。核心层包含了 Batch 启动和控制所需要的核心类，如：JobLauncher、Job 和 step 等。应用层和核心层建立在基础构架层之上，基础构架层提供共通的读（ItemReader）、写（ItemWriter）、和服务（如 RetryTemplate：重试模块。可以被应用层和核心层使用）。

2，Spring Batch 主要对象

领域对象	描述
Job repository	基础组件，用来持久化Job的元数据，默认使用内存
Job launcher	基础组件，用来启动Job
Job	应用组件，是Batch操作的基础执行单元
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的一条数据记录
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写入数据源之前进行数据清洗（转换校验过滤...）
Item Writer	把Chunk中包含的Item写入数据源

三，Spring Batch 流程介绍



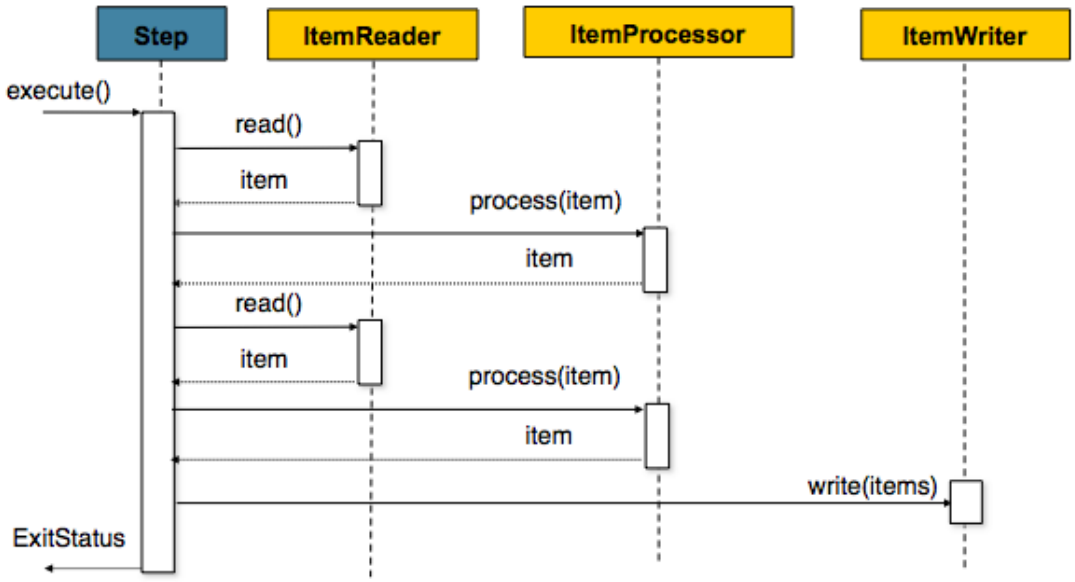
上图描绘了 Spring Batch 的执行过程。说明如下：

每个 Batch 都会包含一个 Job。Job 就像一个容器，这个容器里装了若干 Step，Batch 中实际干活的也就是这些 Step，至于 Step 干什么活，无外乎读取数据，处理数据，然后将这些数据存储起来

(ItemReader 用来读取数据，ItemProcessor 用来处理数据，ItemWriter 用来写数据）。JobLauncher 用来启动 Job，JobRepository 是上述处理提供的一种持久化机制，它为 JobLauncher，Job，和 Step 实例提供 CRUD 操作。

外部控制器调用 JobLauncher 启动一个 Job，Job 调用自己的 Step 去实现对数据的操作，Step 处理完成后，再将处理结果一步步返回给上一层，这就是 Batch 处理实现的一个简单流程。

四，Spring Batch 之 Step 执行过程介绍



从 DB 或是文件中取出数据的时候，read()操作每次只读取一条记录，之后将读取的这条数据传递给 processor(item)处理，框架将重复做这两步操作，直到读取记录的件数达到 batch 配置信息中”commin-interval”设定值的时候，就会调用一次 write 操作。然后再重复上图的处理，直到处理完所有的数据。当这个 Step 的工作完成以后，或是跳到其他 Step，或是结束处理。

这就是一个 SpringBatch 的基本工作流程。

五，Spring Batch 应用

1，简单应用

➢ 构建应用

如“引言”中所述 Spring Batch 按照关注点的不同，将整个批处理过程分为三部分：读、处理、写，从而将批处理应用进行合理解耦。同时，Spring Batch 还针对读、写操作提供了多种实现，如消息、文件、数据库。对于数据库，还提供了 Hibernate、iBatis、JPA 等常见 ORM 框架的读、写接口支持。

➢ 对象定义

首先我们需要编写用户以及消息类，比较简单，如 清单 1 和 清单 2 所示：

清单 1. User 类

```
package org.springframework.batch.sample;

public class User {
        private String name;
        private Integer age;
        public String getName() {return name;}
        public void setName(String name) {this.name = name;}
        public Integer getAge() {return age;}
        public void setAge(Integer age) {this.age = age;}
}
```

清单 2. Message 类

```
package org.springframework.batch.sample;
```

```
public class Message {
		private String content;
		public String getContent() {return content;}
		public void setContent(String content) {this.content = content;}
}
```

清单 3. 用户信息

```
User1,20
User2,21
User3,22
User4,23
User5,24
User6,25
User7,26
User8,27
User9,28
User10,29
```

➢ 读写及处理接口

首先，所有 Spring Batch 的读操作均需要实现 ItemReader 接口，而且 Spring Batch 为我们提供了多种默认实现，如：JdbcPagingItemReader、FlatFileItemReader 等。因此，多数情况下我们并不需要手动编写 ItemReader 类，而是直接使用相应实现类即可。

在该示例中，我们使用 org.springframework.batch.item.file.FlatFileItemReader 类从文件中进行信息读入，用户信息格式定义如 清单 3 所示。

该类封装了文件读操作，仅仅需要我们手动设置 LineMapper 与访问文件路径即可。Spring Batch 通过 LineMapper 可以将文件中的一行映射为一个对象。我们不难发现，Spring Batch 将文件操作封装为类似 Spring JDBC 风格的接口，这也与 Spring 一贯倡导的接口统一是一致的。此处我们使用 org.springframework.batch.item.file.mapping.DefaultLineMapper 进行行映射。读操作的配置信息如 清单 4 所示：

清单 4. message_job.xml

```
<beans:bean id="messageReader"
	class="org.springframework.batch.item.file.FlatFileItemReader">
		<beans:property name="lineMapper" ref="lineMapper">
		</beans:property>
```

```
            <beans:property name="resource"
            value="classpath:/users.txt"></beans:property>
</beans:bean>
<beans:bean id="lineMapper"
            class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <beans:property name="lineTokenizer">
                        <beans:bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
        </beans:bean>
            </beans:property>
            <beans:property name="fieldSetMapper">
                        <beans:bean class="org.springframework.batch.sample.UserMapper">
        </beans:bean>
            </beans:property>
</beans:bean>
```

从清单我们可以知道，DefaultLineMapper 需要设置 lineTokenizer 和 fieldSetMapper 两个属性，首先通过 lineTokenizer 完成文件行拆分，并封装为一个属性结果集，因为我们使用“,”分隔用户属性，所以需要将 lineTokenizer 设置为 DelimitedLineTokenizer。最后通过 fieldSetMapper 完成将结果集封装为一个 POJO 对象。具体实现如 清单 5 所示：

清单 5. UserMapper 类

```
 package org.springframework.batch.sample;

import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.FieldSet;
import org.springframework.validation.BindException;

public class UserMapper implements FieldSetMapper<User> {
            public User mapFieldSet(FieldSet fs) throws BindException {
                        User u = new User();
                        u.setName(fs.readString(0));
                        u.setAge(fs.readInt(1));
                        return u;
            }
}
```

该接口的实现方式与 Spring JDBC 的 RowMapper 极其相似。接下来，再让我们看一下如何实现写操作。Spring Batch 所有写操作均需要实现 ItemWriter 接口。该接口只有一个方法 void write(List<? extends T> items)，参数是输出结果的列表。之所以如此定义，是为了便于我们进行批量操作，以提高性能。每次传入的列表由事务提交粒度确定，也就是说 Spring Batch 每次将提交的结果集传入写操作接口。因为我们要做的仅仅是将通知输出到控制台，所以，写操作实现如 清单 6 所示：

清单 6. MessagesItemWriter 类

```
package org.springframework.batch.sample;

import java.util.List;
import org.springframework.batch.item.ItemWriter;

public class MessagesItemWriter implements ItemWriter<Message>{
        public void write(List<? extends Message> messages) throws Exception {
                System.out.println("write results");
                for (Message m : messages) {
                        System.out.println(m.getContent());
                }
        }
}
```

同 ItemReader 一样，Spring Batch 也为我们提供了多样的写操作支持。

最后，再看一下如何实现业务处理。Spring Batch 提供了 ItemProcessor 接口用于完成相应业务处理。在本示例中，即为根据用户信息生成一条缴费通知信息，如 清单 7 所示：

清单 7. MessagesItemProcessor 类

```
package org.springframework.batch.sample;

import org.springframework.batch.item.ItemProcessor;

public class MessagesItemProcessor implements ItemProcessor<User, Message> {

        public Message process(User user) throws Exception {
                Message m = new Message();
                m.setContent("Hello " + user.getName()
                                + ",please pay promptly at the end of this month.");
                return m;
        }

}
```

➢ 任务定义

通过上面部分，我们已经完成了批处理任务的读数据、处理过程、写数据三个过程。那么，我们如何将这三部分结合在一起完成批处理任务呢？

Spring Batch 将批处理任务称为一个 Job，同时，Job 下分为多个 Step。Step 是一个独立的、顺序的处理步骤，包含该步骤批处理中需要的所有信息。多个批处理 Step 按照一定的流程组成一个 Job。通过这样的设计方式，我们可以灵活配置 Job 的处理过程。

接下来，让我们看一下如何配置缴费通知的 Job，如 清单 8 所示：

清单 8. message_job.xml

```
<job id="messageJob">
  <step id="messageStep">
            <tasklet>  <!— 事务级别 -->
              <chunk reader="messageReader" processor="messageProcessor"  <!— 数据级别 -->

                writer="messageWriter" commit-interval="5"
                chunk-completion-policy="">
                        </chunk>
            </tasklet>
   </step>
</job>
```

如上，我们定义了一个名为“messageJob”的 Job，该 Job 仅包含一个 Step。在配置 Step 的过程中，我们不仅要指定读数据、处理、写数据相关的 bean，还要指定 commit-interval 和 chunk-completion-policy 属性。前者指定了该 Step 中事务提交的粒度，取值为 5 即表明每当处理完毕读入的 5 条数据时，提交一次事务。后者指定了 Step 的完成策略，即当什么情况发生时表明该 Step 已经完成，可以转入后续处理。由于没有明确指定相应的类，Spring Batch 使用默认策略，即当读入数据为空时认为 Step 结束。

最后，我们还需要配置一个 JobRepository 并为其指定一个事务管理器，该类用于对 Job 进行管理，如清单 9 所示：

清单 9. message_job.xml

```
<beans:bean id="jobRepository"
class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
                        <beans:property name="transactionManager" ref="transactionManager" />
</beans:bean>

<beans:bean id="transactionManager"
```

```
class="org.springframework.batch.support.transaction.ResourcelessTransactionManager"/>
```

因为我们整个该示例不需要数据库操作，所以选择了使用 MapJobRepositoryFactoryBean 和 ResourcelessTransactionManager，如果为数据库操作则其应对应使用数据库 Spring batch 提供的相应插件，如 JobRepositoryFactoryBean 和 DataSourceTransactionManager。所有配置完成以后，进入最后一步——任务执行。

➢ 任务执行

那么如何运行一个 Job 呢？ Spring Batch 提供了 JobLauncher 接口用于运行 Job，并提供了一个默认实现 SimpleJobLauncher。先让我们看一下具体执行代码，如 清单 10 所示：

清单 10. Main 类

```
public class Main {
        public static void main(String[] args) {
    ClassPathXmlApplicationContext c =
            new ClassPathXmlApplicationContext("message_job.xml");
    SimpleJobLauncher launcher = new SimpleJobLauncher();
    launcher.setJobRepository((JobRepository) c.getBean("jobRepository"));
    launcher.setTaskExecutor(new SimpleAsyncTaskExecutor());
    try {
        launcher.run((Job) c.getBean("messageJob"), new JobParameters());
    } catch (Exception e) {
    e.printStackTrace();
    }
        }
}
```

首先，我们需要为 JobLauncher 指定一个 JobRepository，该类负责创建一个 JobExecution 对象来执行 Job，此处直接从上下文获取即可。其次，需要指定一个任务执行器，我们使用 Spring Batch 提供的 SimpleAsyncTaskExecutor。最后，通过 run 方法来执行指定的 Job，该方法包含两个参数，需要执行的 Job 以及执行参数。运行结果如下：

```
write results
Hello User1,please pay promptly at end of this month.
Hello User2,please pay promptly at end of this month.
Hello User3,please pay promptly at end of this month.
Hello User4,please pay promptly at end of this month.
Hello User5,please pay promptly at end of this month.
```

```
write results
Hello User6,please pay promptly at end of this month.
Hello User7,please pay promptly at end of this month.
Hello User8,please pay promptly at end of this month.
Hello User9,please pay promptly at end of this month.
Hello User10,please pay promptly at end of this month.
```

从业务功能上考虑，同一任务应该尽量避免重复执行（即相同条件下的任务只能成功运行一次），试想如果本示例中发送缴费通知过多只能导致用户不满，那么电信计费批处理任务重复执行则将导致重复计费，从而使用户遭受损失。幸运的是，Spring Batch 已经为我们考虑好了这些。
对于 Spring Batch 来说，JobParameters 相同的任务只能成功运行一次。您如果在示例 Main 类中连续运行同一 Job，将会得到如下异常（见 清单 11 ）：

清单 11. 异常信息

```
org.springframework.batch.core.repository.JobInstanceAlreadyCompleteException:
A job instance already exists and is complete for parameters={}.
If you want to run this job again, change the parameters.
```

因此，如果我们希望该任务是周期执行的（如每月执行一次），那么必须保证周期内参数是唯一。假如该客户要求我们每月为用户发送一次缴费通知。我们的任务执行可以如 清单 12 所示：

清单 12. Main 类

```
Map<String,JobParameter> parameters = new HashMap<String,JobParameter>();
parameters.put(RUN_MONTH_KEY,new JobParameter("2011-10"));
launcher.run((Job) c.getBean("messageJob"),new JobParameters(parameters));
parameters.put(RUN_MONTH_KEY,new JobParameter("2011-11"));
launcher.run((Job) c.getBean("messageJob"),new JobParameters(parameters));
```

在示例中，我将执行月份作为 Job 的参数传入，分别执行了 10、11 月两个月的任务。

➢ 任务重试

既然相同参数的任务只能成功执行一次，那么，如果任务失败该如何处理？此时，需要考虑的是，既然任务步骤有事务提交粒度，那么可能任务已经提交了部分处理结果，这部分不应该被重复处理。也就是说，此时应该有重试操作。

在 Spring Batch 中，通过配置可以实现步骤 Step 的重试，如 清单 13 所示：

清单 13. message_job.xml

```
<job id="messageJob" restartable="true">
        <step id="messageStep">
                <tasklet>
                        <chunk reader="messageReader" processor="messageProcessor"
                                writer="messageWriter"
            commit-interval="5" chunk-completion-policy="" retry-limit="2">
                                <retryable-exception-classes>
                                        <include
class="java.lang.RuntimeException" />
                                </retryable-exception-classes>
                        </chunk>
                </tasklet>
        </step>
</job>
```

我们可以看到，主要分两部分：首先，需要设置重试次数，其次是当执行过程中捕获到哪些异常时需要重试。如果在执行过程中捕获到重试异常列表中的异常信息，则进行重试操作。如果重试操作达到最大次数仍提示异常，则认为任务执行失败。对于异常信息的配置，除了通过 include 配置包含列表外，也可以通过 exclude 配置排除列表。

由于通过配置进行的 Step 重试是自动的，因此较难控制（多用于网络访问异常等不需要人工干预的情况）。可以考虑一下本示例，如果有一个用户的信息有问题，名字为空，不能发送缴费通知，步骤重试便不合适了，此时我们可以对 Job 进行重试操作。

Spring Batch 允许重复执行未成功的 Job，而每次执行即为一次重试操作。示例代码如 清单 14 所示：

清单 14. Main 类

```
Map<String,JobParameter> parameters = new HashMap<String,JobParameter>();
parameters.put(RUN_MONTH_KEY,new JobParameter("2011-10"));
launcher.run((Job) c.getBean("messageJob"),new JobParameters(parameters));
Thread.sleep(10000);
launcher.run((Job) c.getBean("messageJob"),new JobParameters(parameters));
```

可以通过如下步骤查看运行结果：首先，将 users.txt 文件中的第 7 行（之所以指定该行，便于验证事务提交以及重复执行的起始位置）的用户名修改为空。其次，运行示例。最后，在程序出现异常提示时，更新第 7 行的用户名（为了便于演示，程序在两次任务执行过程中等待 10 秒钟）。

您可以在控制台中很明显的看到，任务先打印了 5 条记录（第一次事务提交），然后出现异常信息，待我们将错误更正后，又打印了 5 条记录，任务最终成功完成。

从输出结果，我们可以知道 Spring Batch 是从出错的事务边界内第一条记录重复执行的，这样便确保了数据完整性，而且所有这一切对于用户均是透明的。
那么 Spring Batch 是如何做到这一步的呢？这与 Spring Batch 的运行时管理是分不开的。

➢ 运行时管理

Spring Batch 提供了如 表 1 所示的类用于记录每个 Job 的运行信息：

表 1. 运行时类信息

类名	描述
JobInstance	该类记录了 Job 的运行实例。举例：10 月和 11 月分别执行同一 Job，将生成两个 JobInstance。主要信息有：标识、版本、Job 名称、Job 参数
JobExecution	该类记录了 Job 的运行记录。如上面的示例，Job 第一次运行失败，第二次运行成功，那么将形成两条运行记录，但是对应的是同一个运行实例。主要信息有：Job 的运行时间、运行状态等。
JobParameters	该类记录了 Job 的运行参数
ExecutionContext	该类主要用于开发人员存储任务运行过程中的相关信息（以键值对形式），主要分为 Job 和 Step 两个范围
StepExecution	该类与 JobExecution 类似，主要记录了 Step 的运行记录。包括此次运行读取记录条数、输出记录条数、提交次数、回滚次数、读跳过条数、处理跳过条数、写跳过条数等信息

Spring Batch 通过 JobRepository 接口维护所有 Job 的运行信息，此外 JobLauncher 的 run 方法也返回一个 JobExecution 对象，通过该对象可以方便的获得 Job 其他的运行信息，代码如 清单 15 所示：

清单 15. Main 类

```
Map<String,JobParameter> parameters = new HashMap<String,JobParameter>();
parameters.put(RUN_MONTH_KEY,new JobParameter("2011-10"));
JobExecution je =
        launcher.run((Job) c.getBean("messageJob"),new JobParameters(parameters));
System.out.println(je);
System.out.println(je.getJobInstance());
System.out.println(je.getStepExecutions());
```

输出信息如 清单 16 所示：

清单 16. 输出结果

```
JobExecution: id=0, version=2, startTime=Tue Nov 15 21:00:09 CST 2011,
endTime=Tue Nov 15 21:00:09 CST 2011, lastUpdated=Tue Nov 15 21:00:09 CST 2011,
status=COMPLETED, exitStatus=exitCode=COMPLETED;exitDescription=,
job=[JobInstance: id=0, version=0, JobParameters=[{run.month=2011-10}], Job=[messageJob]]

 JobInstance: id=0, version=0, JobParameters=[{run.month=2011-10}], Job=[messageJob]

 [StepExecution: id=1, version=5, name=messageStep, status=COMPLETED,
 exitStatus=COMPLETED, readCount=10, filterCount=0, writeCount=10 readSkipCount=0,
   writeSkipCount=0, processSkipCount=0, commitCount=3 , rollbackCount=0,
   exitDescription=]
```

从日志您可以发现事务一共提交了 3 次，这与前面的说明是不一致的。之所以会如此是因为当事务提交粒度恰好可以被记录数整除时，事务会有一次空提交。

关于 Spring Batch 运行时信息管理，将在讲解 Job 监控时再详细介绍，此处不再赘述，你也可以查看 Spring Batch 参考资料了解相关信息。

2，高级应用

➢ Step Flow 介绍

通过前文我们已经知道，Step 是一个独立的、顺序的处理步骤，包含完整的输入、处理以及输出。但是在企业应用中，我们面对的更多情况是多个步骤按照一定的顺序进行处理。因此如何维护步骤之间的执行顺序是我们需要考虑的。Spring Batch 提供了 Step Flow 来解决这个问题。

➢ 批量操作数据库介绍

让我们回到用户缴费通知的 Job。客户提出了进一步的需求：计费、扣费、缴费通知要确保顺序执行。

1, 为每个用户生成账单

2,缴费，客户余额减少，账单修改

3，发送缴费通知

清单 1. billing_job.xml

```
<beans:bean id="jobRepository"
    class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">
    <beans:property name="dataSource" ref="dataSource" />
    <beans:property name="transactionManager" ref="transactionManager" />
</beans:bean>
<beans:bean id="userDbReader"
                    class="org.springframework.batch.item.database.JdbcPagingItemReader">
                    <beans:property name="dataSource" ref="dataSource" />
                    <beans:property name="rowMapper" ref="userDbMapper" />
                    <beans:property name="queryProvider" ref="userQueryProvider" />
</beans:bean>
<beans:bean id="userDbMapper"
                    class="org.springframework.batch.sample.UserDbMapper" />
<beans:bean id="userQueryProvider"
                    class="org.springframework.batch.item.database.support.OraclePagingQueryProvider">
                    <beans:property name="selectClause" value="u.id,u.name,u.age,u.balance" />
                    <beans:property name="fromClause" value="users u" />
                    <beans:property name="sortKey" value="u.id" />
</beans:bean>
<beans:bean id="messageDbWriter"
                    class="org.springframework.batch.item.database.JdbcBatchItemWriter">
                    <beans:property name="dataSource" ref="dataSource" />
                    <beans:property name="sql"
     value="insert into messages(id,user_id,content) values(:id,:user.id,:content)" />
                    <beans:property name="itemSqlParameterSourceProvider"
                                        ref="itemSqlParameterSourceProvider" />
</beans:bean>
<beans:bean id="itemSqlParameterSourceProvider"
class="org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider"
 />
```

我们分别使用 Spring Batch 提供的 JdbcPagingItemReader 和 JdbcBatchItemWriter 进行读写。同时，我将 jobRepository 修改为 JobRepositoryFactoryBean，因此，运行示例前，还需要建立 User、PayRecord、Bill 对应的表。

- Job 多个 Step 的执行

在配置 Step 时，我们可以指定其 next 属性，该属性指向另一个 Step。通过配置 Step 的 next 属性，我们便可以轻易实现上述流程。具体如清单 2 所示

清单 2. billing_job.xml

```
<job id="billingJob" restartable="true">
        <step id="billingStep" next="payStep">
            <tasklet>
                <chunk reader="userDbReader" processor="billingProcessor"
    writer="billDbWriter" commit-interval="5" chunk-completion-policy="">
                </chunk>
            </tasklet>
        </step>
        <step id="payStep" next="messageStep">
            <tasklet>
  <chunk reader="billDbReader" processor="payProcessor" writer="payDbWriter"
                commit-interval="5" chunk-completion-policy="" skip-limit="100" >
<skippable-exception-classes>
 <include class="org.springframework.batch.sample.MoneyNotEnoughException" />
                        </skippable-exception-classes>
                </chunk>
            </tasklet>
        </step>
        <step id="messageStep">
            <tasklet>
                <chunk reader="billArrearsDbReader" processor="messageProcessor"
                        writer="messageDbWriter" commit-interval="5"
                        chunk-completion-policy="">
                </chunk>
            </tasklet>
        </step>
</job>
```

我们将 billStep 的 next 设置为 payStep，将 payStep 的 next 设置为 messageStep，同时分别指定了读、处理、写接口。Spring Batch 在运行 billingJob 时，首先执行 billingStep，查找用户信息生成账单费用，然后执行 payStep，查找账单信息生成扣费记录，如果用户余额不足则跳过。最后，查找欠费账单，生成缴费通知。只有当上一步执行成功后，才会执行下一步。

billStep 和 payStep 的 ItemProcessor 实现分别如清单 3 和清单 4 所示：

清单 3.BillingItemProcessor 类

```
public class BillingItemProcessor implements ItemProcessor#<User, Bill> {

    public Bill process(User item) throws Exception {
        Bill b = new Bill();
```

```
                b.setUser(item);
                b.setFees(70.00);
                b.setPaidFees(0.0);
                b.setUnpaidFees(70.00);
                b.setPayStatus(0);/*unpaid*/
                return b;
        }

}
```

清单 4.PaymentItemProcessor 类

```
public class PaymentItemProcessor implements ItemProcessor<Bill, PayRecord> {
public PayRecord process(Bill item) throws Exception {
                if (item.getUser().getBalance() <= 0) {
                        return null;
                }
                if (item.getUser().getBalance() >= item.getUnpaidFees()) {
                        // create payrecord
                        PayRecord pr = new PayRecord();
                        pr.setBill(item);
                        pr.setPaidFees(item.getUnpaidFees());
                        // update balance
                        item.getUser().setBalance(item.getUser().getBalance() -
                            item.getUnpaidFees());
                        // update bill
                        item.setPaidFees(item.getUnpaidFees());
                        item.setUnpaidFees(0.0);
                        item.setPayStatus(1);/* paid */
                        return pr;
                } else {
                        throw new MoneyNotEnoughException();
                }
        }
}
```

在清单 3 中，我们为每个用户生成一条 70 元的账单，已缴费用为 0，未缴费用为 70。在清单 4 中，将账单金额从用户余额中扣除，并更新账单已缴和未缴费用，如果余额不足，提示异常（通过清单 2 可知，我们对于此类异常进行了跳过处理）。

此外，我们现在的缴费通知需要基于欠费账单生成，因此，我们需要新提供一个缴费通知的 ItemProcessor，具体如清单 5 所示：

清单 5.ArrearsMessagesItemProcessor 类

```
public class ArrearsMessagesItemProcessor implements
                        ItemProcessor<Bill, Message> {

    public Message process(Bill item) throws Exception {
                        if (item.getPayStatus() == 0) {/*unpaid*/
                                Message m = new Message();
                                m.setUser(item.getUser());
                                m.setContent("Hello " + item.getUser().getName()
                                                + ",please pay promptly at end of this month.");
                                return m;
                        }
                        return null;
    }

}
```

每个 Step 的读写接口可参照 billing_job.xml，均使用 Spring Batch 提供的实现类，此处不再赘述（此处需要特别注意 payDbWriter，由于扣费时，我们需要同时生成扣费记录，并更新用户和账单，因此我们使用了 CompositeItemWriter）。至此，我们已经完成了第一步，实现了基本的多步骤顺序处理，您可以运行 Main2，并通过数据库查看运行结果（bills、payrecords、messages）。

- 条件流程和流程决策

通过上面的 Step Flow，我们已经满足了客户的初步需求，但是客户又提出进一步要求：能否当所有用户费用均足够的情况下，不再执行缴费通知处理。因为查询一遍欠费账单在一定程度上还是降低了处理性能。Spring Batch 提供了条件流程和流程决策来支持类似应用场景。
首先，让我们看一下如何使用条件流程来实现该需求。Step 通过在 next 元素上设置 on 属性来支持条件流程，on 属性取值为 Step 的结束状态，如 COMPLETED、FAILED 等，同时还支持 * 以及 ？通配符。

由于我们希望当存在余额不足的情况时，也就是 payStep 的跳过条数大于 0 时，再执行缴费通知 Step，因此，我们需要特殊指定一种结束状态。此处，我们可以为 Step 添加一个监听器，以返回指定的结束状态。修改后的 payStep 如清单 6 所示，监听器实现如清单 7 所示：

清单 6. billing_job.xml

```
<step id="payStep">
        <tasklet>
                <chunk reader="billDbReader" processor="payProcessor" writer="payDbWriter"
                        commit-interval="5" chunk-completion-policy="" skip-limit="100">
```

```
                                        <skippable-exception-classes>
                                                <include
        class="org.springframework.batch.sample.MoneyNotEnoughException" />
                                        </skippable-exception-classes>
                        </chunk>
                </tasklet>
                <next on="COMPLETED WITH SKIPS" to="messageStep"/>
                <listeners>
                        <listener ref="payStepCheckingListener"></listener>
                </listeners>
</step>
```

清单 7. PayStepCheckingListener 类

```
public class PayStepCheckingListener extends StepExecutionListenerSupport {

        @Override
        public ExitStatus afterStep(StepExecution stepExecution) {
                String exitCode = stepExecution.getExitStatus().getExitCode();
                if (!exitCode.equals(ExitStatus.FAILED.getExitCode())
                                && stepExecution.getSkipCount() > 0) {
                        return new ExitStatus("COMPLETED WITH SKIPS");
                } else {
                        return null;
                }
        }

}
```

接下来，再让我们看一下如何使用流程决策来实现该功能。多数情况下，Step 的结束状态并不能够满足较为复杂的条件流程，此时便用到了流程决策器。通过它，我们可以根据 Job 和 Step 的各种执行情况返回相应的执行状态来控制流程。

首先，我们需要定义一个流程决策器，代码如清单 8 所示：

清单 8. MessagesDecider 类

```
public class MessagesDecider implements JobExecutionDecider {

        public FlowExecutionStatus decide(JobExecution jobExecution,
                        StepExecution stepExecution) {
                String exitCode = stepExecution.getExitStatus().getExitCode();
```

```
                        if (!exitCode.equals(ExitStatus.FAILED.getExitCode())
                                        && stepExecution.getSkipCount() > 0) {
                                return new FlowExecutionStatus("COMPLETED WITH SKIPS");
                        } else {
                                return FlowExecutionStatus.COMPLETED;
                        }
                }

}
```

与 StepExecutionListener 不同，该类的 decide 方法返回一个 FlowExecutionStatus 对象。与之对应，Job 配置修改为如清单 9 所示：

清单 9. billing_job2.xml

```
<job id="billingJob" restartable="true">
                <step id="billingStep" next="payStep">
                </step>
                <step id="payStep" next="decider">
                </step>
                <decision id="decider" decider="messagesDecider">
                        <next on="COMPLETED WITH SKIPS" to="messageStep" />
                        <end on="COMPLETED" />
                </decision>
                <step id="messageStep">
                </step>
</job>
```

可以看到 payStep 的 next 变成了 decider，在 decider 中根据返回结果确定执行路径：如果存在跳过的情况，执行 messageStep，否则直接结束 Job（注意：此处我们用到了 end 元素）。
通过上面的讲述，我们大体了解了 Spring Batch 对于条件流程的支持（此外，我们可以通过设置 Step 的 next 属性为先前执行的 Step，从而实现支持循环的 Job，但是大家并不认为这是实现循环任务的一个好方案），接下来再让我们看一下批处理中另一项重要特征——并发。

➢ 并发处理

如果我们的批处理任务足够简单，硬件配置及网络环境也足够好，那么我们完全可以将批处理任务设计为单线程，但现实是企业应用对于硬件的要求要比硬件自身发展快的多，更何况还有那么多的企业要在较差的硬件环境中运行自己的企业应用并希望拥有一个可以接受的性能。因此在企业应用中，尤其是涉及到大批量数据处理，并发是不可避免的。那么，Spring Batch 在并发方面又提供了哪些功能支持呢？

首先，Spring Batch 提供了 Step 内的并发，这也是最简单的一种并发处理支持。通过为 Step 设置 task-executor 属性，我们便可以使当前 Step 以并发方式执行。同时，还可以通过 throttle-limit 设置并发线程数（默认为 4）。也就是说您不必修改任何业务处理逻辑，仅仅通过修改配置即可以实现同步到异步的切换。

如果我们希望示例中的 billingStep 以并发方式执行，且并发任务数为 5，那么只需要做如下配置即可，见清单 10：

清单 10. billing_job3.xml

```
<step id="billingStep" next="payStep">
        <tasklet task-executor="taskExecutor" throttle-limit="5">
                <chunk reader="userDbReader" processor="billingProcessor"
                writer="billDbWriter" commit-interval="5" chunk-completion-policy="">
                </chunk>
        </tasklet>
</step>
<beans:bean id="taskExecutor"
        class="org.springframework.core.task.SimpleAsyncTaskExecutor">
</beans:bean>
```

从清单可以看出，我们为 billingStep 指定了一个异步任务执行器 SimpleAsyncTaskExecutor，该执行器将会按照配置创建指定数目的线程来进行数据处理。通过这种方式，避免了我们手动创建并管理线程的工作，使我们只需要关注业务处理本身。

需要补充说明的是，Spring Core 为我们提供了多种执行器实现（包括多种异步执行器），我们可以根据实际情况灵活选择使用。当然，像我们此处需要并发处理时，必须使用异步执行器。几种主要实现如表 1 所示：

表 1. 任务执行器列表

类名	描述	是否异步
SyncTaskExecutor	简单同步执行器	否
ThrottledTaskExecutor	该执行器为其他任意执行器的装饰类，并完成提供执行次数限制的功能	视被装饰的执行器而定
SimpleAsyncTaskExecutor	简单异步执行器，提供了一种最基本的异步执行实现	是
WorkManagerTaskExecutor	该类作为通过 JCA 规范进行任务执行的实现，其包含 JBossWorkManagerTaskExecutor 和 GlassFishWorkManagerTaskExecutor 两个子类	是
ThreadPoolTaskExecutor	线程池任务执行器	是

其次，Spring Batch 还支持 Step 间的并发，这是通过 Split Flow 实现的。让我们看看 Split Flow 是如何使用的。在此之前，让我们设想一下，假如客户基于上面的示例提出进一步需求：每月为用户生成扣费通知，并生成账单、扣费、缴费通知（对于费用不足的情况）。
当然，要实现上述需求有很多种方式，比如，按照生成账单、扣费通知、扣费、缴费通知的顺序串行执行，然而，此种处理方式势必会降低性能，即使我们可以使用 Step 多线程处理来提高性能，可仍不是最优方式。那么我们该如何改进呢？显然，我们可以将生成扣费通知和扣费并行执行，因为这两步是完全独立的。修改后的 billing_job 如清单 11 所示：

清单 11. billing_job3.xml

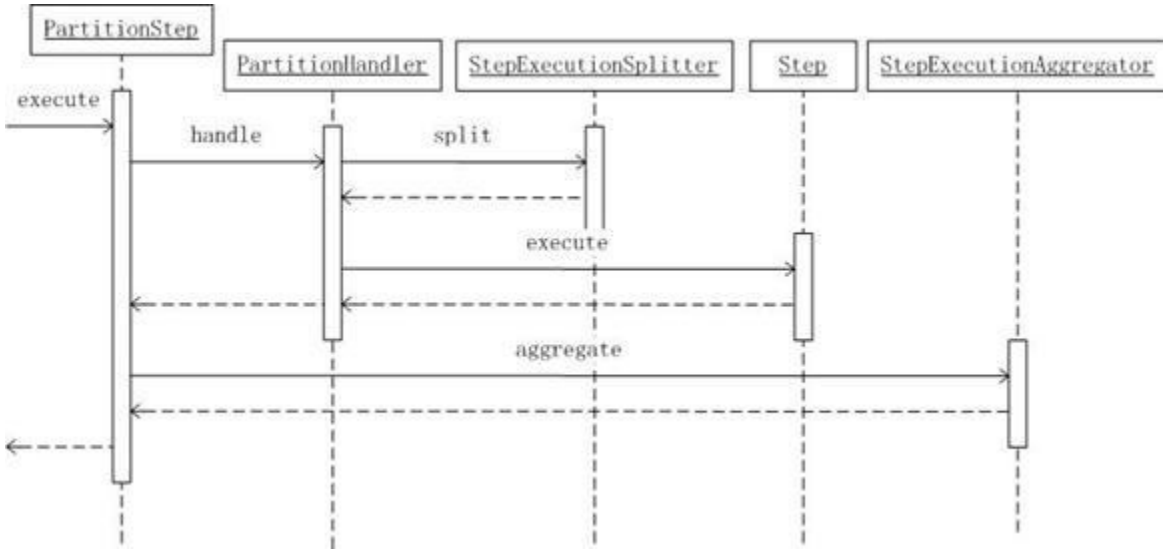
```
<job id="billingJob" restartable="true">
        <step id="billingStep" next="splitStep">
                <tasklet task-executor="taskExecutor" throttle-limit="5">
                        <chunk reader="userDbReader" processor="billingProcessor"
writer="billDbWriter" commit-interval="5" chunk-completion-policy="">
                        </chunk>
                </tasklet>
        </step>
        <split id="splitStep" task-executor="taskExecutor"  next="decider">
                <flow>
                        <step id="billingMessageStep">
                                <tasklet>
 <chunk reader="billDbReader" processor="billMessageItemProcessor"
                                                writer="messageDbWriter" commit-interval="5"
                                                        chunk-completion-policy="">
                                                </chunk>
                                </tasklet>
                        </step>
                </flow>
                <flow>
                        <step id="payStep">
                                <tasklet>
  <chunk reader="billDbReader" processor="payProcessor"
  writer="payDbWriter" commit-interval="5" chunk-completion-policy=""
  skip-limit="100">
   <skippable-exception-classes>
     <include
 class="org.springframework.batch.sample.MoneyNotEnoughException" />
     </skippable-exception-classes>
                                                </chunk>
                                </tasklet>
                        </step>
```

```
                        </flow>
                </split>
                <decision id="decider" decider="messagesDecider">
                        <next on="COMPLETED WITH SKIPS" to="paymentMessageStep" />
                        <end on="COMPLETED" />
                </decision>
                <step id="paymentMessageStep">
                        <tasklet>
                                <chunk reader="billArrearsDbReader" processor="messageProcessor"
                                                                writer="messageDbWriter" commit-interval="5"
                                                                chunk-completion-policy="">
                                </chunk>
                        </tasklet>
                </step>
</job>
```

从清单 10 可以看出，billingStep 的下一步变成了一个 split 元素，该元素下包含两个 flow。“flow”顾名思义包含一系列可执行的 step，示例中两个 flow 分别包含 billingMessageStep（生成扣费通知）和 payStep 两个 step。Spring Batch 执行 split 时，将会并行执行其下所有 flow，而且只有当所有 step 均执行完毕之后，才会执行 split 元素的下一步，当然，前提是您为 split 元素指定的"task-executor"为 SimpleAsyncTaskExecutor，该属性默认为 SyncTaskExecutor，即串行执行。

通过上述两种方式，我们可以实现 Job 的并发处理，但显然该方式有其局限性，即仅限于单机。让我们设想一下如下场景：在上述缴费任务中，用户生成账单非常慢（也许是因为业务处理过于复杂，也许因为生成账单的过程中同时处理了好多关联信息）。这种场景我们该怎么优化呢？显然，即便我们将该步骤配置为并行，那么它的优化空间也是有限的，因为线程并发到一定数量之后必定受限于系统硬件配置。这个时候，我们自然会想到修改部署方式，将耗时操作分配到多个机器上并行执行。那么基于 Spring Batch 我们该如何实现呢？此处便用到了 PartitionStep。让我们看一下它是如何执行的，其时序图如图 1 所示：

图 1. PartitionStep 序列图



从图中我们可以看到，PartitionStep 并不负责读、写数据，它只是根据配置的策略（PartitionHandler）将 StepExecution 进行分解，并委派到指定的 Step 上并行执行（该 Step 可能是本地，也可能是远程），执行完毕后，将所有执行结果进行合并（由 StepExecutionAggregator 完成）作为自身的执行结果。利用 PartitionStep，在委派 Step 为远程调用的情况下，我们可以很容易通过增加从机数目的方式来提高任务运行效率，大大提高了系统的可伸缩性。而且此种方式并不会影响 PartitionStep 所在 Job 的执行顺序，因为 PartitionStep 只有当所有委派 Step 完成之后，才会继续往下执行。

不过使用 PartitionStep 需要注意以下几点：

- 由于数据的读写以及处理均在从机上进行，因此需要确保并发的从机之间不会重复读取数据（当然，这个问题是所有批处理应用采用主从和集群架构时所必须考虑的问题，而并非只有 Spring Batch 才会有）。
- 确保分解到各个从机上的 StepExecution 是不同的。在 StepExecutionSplitter 的默认实现 SimpleStepExecutionSplitter 中，首先通过一个 Partitioner 得到分解后的 ExecutionContext，然后针对每个 ExecutionContext，创建 StepExecution（当然，如果 Step 为重复执行，那么将会得到上次运行的 ExecutionContext 和 StepExecution，而非重新创建）。

从第二点可以看出，通过在 ExecutionContext 设置唯一的信息，我们便可以保证每个从机读取的数据是不同的。

主从方式的具体配置如清单 12 所示：

清单 12. partition.xml

```
<beans:bean name="step"
                class="org.springframework.batch.core.partition.support.PartitionStep">
                <beans:property name="partitionHandler">
                        <beans:bean
class="org.springframework.batch.core.partition.support.TaskExecutorPartitionHandler">
                                <beans:property name="step" ref="remoteStep" />
                                <beans:property name="gridSize" value="10" />
```

```
                                        <beans:property name="taskExecutor" ref="taskExecutor" />
                                </beans:bean>
                </beans:property>
                <beans:property name="stepExecutionSplitter">
        <beans:bean
class="org.springframework.batch.core.partition.support.SimpleStepExecutionSplitter">
                                        <beans:constructor-arg ref="jobRepository" />
                                        <beans:constructor-arg ref="messageStep" />
                                        <beans:constructor-arg ref="simplePartitioner" />
                                </beans:bean>
                </beans:property>
                <beans:property name="jobRepository" ref="jobRepository" />
</beans:bean>

<step id="messageStep">
                <tasklet task-executor="taskExecutor">
                                <chunk reader="messageReader" processor="messageProcessor"
                                writer="messageWriter" commit-interval="5" chunk-completion-policy=""
                                                                retry-limit="2">
                                        <retryable-exception-classes>
                                                                <include class="java.lang.RuntimeException" />
                                        </retryable-exception-classes>
                                </chunk>
                </tasklet>
</step>
<beans:bean id="remoteStep"
                class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
                <beans:property name="serviceInterface"
                   value="org.springframework.batch.core.Step" />
                <beans:property name="serviceUrl"
                    value="${batch.remote.base.url}/steps/messageStep" />
</beans:bean>
```

此处只采用 Spring Batch 的默认实现，将 Step 发送到一台从机上执行，当然，您完全可以基于 Spring Batch 当前接口，轻易扩展出分发到 N 台从机上执行的实现。

此外，在耗时的 Step 比较独立的情况下（如发送扣费通知的 Step，后续 Step 不会依赖扣费通知 Step 的任何输出结果），我们还可以采用另一种主从架构。在主机上配置一个标准的 Step，其 ItemWriter 负责将读取的记录以 Message 的形式发送给消息中间件（当然，该方案并未充分利用 Spring Batch 的特性，而是由消息中间件完成并发处理）。

3，监控

Spring Batch 提供了 4 种监控方式：

- 直接查看 Job repository 的数据库信息，所有的 Batch 元数据都会持久化到数据库中
- 使用Spring Batch 提供的 API自己构建监控数据
- 使用Spring Batch Admin，通过 web 控制台监控和操作 Job
- 使用JMX

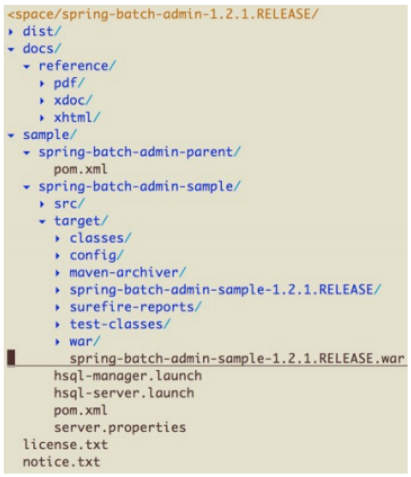
其中，Spring Batch Admin 是 Spring Source 开源的基于 Web方式监控 Batch Job 的应用框架

- 既可以独立运行，也可以非常方便的集成到现有应用中
- 可以启动和监控 Job 的执行情况，提供 Json 数据
- 前端基于 FreeMarker 模板引擎，非常易与定制开发
- 当前版本：1.2.1

安装步骤

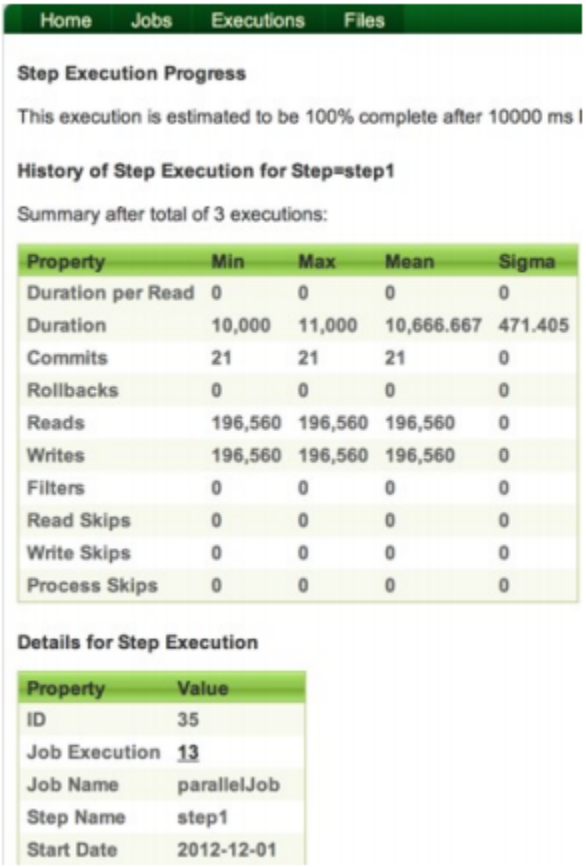
- 下载spring-batch-admin-1.2.1.RELEASE.zip
- 解压缩进入spring-batch-admin-1.2.1.RELEASE的sample目录
- cd spring-batch-admin-parent/
  mvn install
- cd spring-batch-admin-sample/
  mvn install

- mvn会在spring-batch-admin-sample/target下构建出spring-batch-admin-sample-1.2.1.RELEASE.war，根据该war包可以容易搭建出batch admin的web应用



Batch Admin的配置

- Batch Admin的配置文件和资源文件默认都打包到了jar中
- 定义dataSource、transactionManager
  META-INF/spring/batch/bootstrap/manager/data-source-context.xml
- 定义jobRepository、jobExplorer、jobLauncher
  META-INF/spring/batch/bootstrap/manager/execution-context.xml
- 定义配置文件加载路径
  /org/springframework/batch/admin/web/resources/webapp-config.xml
- 支持多种数据库，启动Server时增加虚拟机参数识别数据库类型
  -DENVIRONMENT=mysql
- 是否需要初始化数据
  batch.data.source.init=false
- 新增的job配置文件放置到META-INF/spring/batch/jobs/下，自动识别



六，总结

本文通过一个简单示例演示了如何构建 Spring Batch 应用，同时介绍了 Spring Batch 的相关核心概念。希望大家通过本文可以掌握 Spring Batch 的基本功能。