

Spring Batch Framework Reference **beta with Korean**



제 작: 박찬욱

메 일: Jajamen1367@gmail.com

블로그: <http://chanwook.tistory.com>

한국 스프링 사용자 모임: <http://ksug.org>

최초 배포 : 2008.08.05

Beta 2.0 배포 : 2008.08.31

Beta 3.0 배포 : 2008.11.27

스프링 배치 프레임워크 레퍼런스 번역(완역이 아닙니다.) 문서 베타 버전입니다.

2.0 M2 버전을 대상으로 편역한 문서입니다. 완역이 아닌 편역이기에 일부 누락되거나 의미 전달이 불충분한 부분도 있을 수 있습니다. 양해해주시고, 메일이나 블로그로 연락주시면 수정하도록 하겠습니다.

아직 스프링 배치에 대한 한글 자료가 많이 없습니다. 미약하지만 많은 도움 되셨으면 합니다. 그럼 좋은 하루 되시길 바랍니다.

수정, 배포는 자유입니다. 출처 정도만 남겨주세요.

감사합니다.



버전	변경 여부	변경 대상
1.0	추가	1, 2, 3, 4 장 전체 혹은 일부
2.0	추가	3.11. 아이템 변환하기
2.0	추가	5. 반복하기(전체)
2.0	추가	6. 재시도하기(전체)
2.0	수정	3.6 XML 아이템 reader와 writer
2.0	수정	3.9 데이터베이스
3.0	수정	5. 반복하기(전체)
3.0	수정	4. Job 구성하고 실행하기(일부)
3.0	수정	1. 스프링 배치 소개(+2.0M2에서 추가된 내용)
3.0	추가	2.10. Item Processor (+2.0M2)
3.0	추가	3.5.2.9. 플랫 파일에서 예외 처리하기(+2.0M2)
3.0	추가	3.9.2. 페이지 처리가 적용된 ItemReader (+2.0M2)
3.0	삭제	3.5.3.2. FieldSetCreator(-2.0M2)

1장 스프링 배치 소개

1.1. 소개

1.1.1. 배경

1.1.2. 쓰임새 시나리오

1. 비즈니스 시나리오

- 주기적인 커밋 배치 프로세스
- 동시 배치 처리: 잡 병행 처리
- Staged, 엔터프라이즈 메세지-지향 처리
- 규모가 큰 병행 배치 처리
- 실패 후에 수동 또는 스케줄을 적용한 재시작
- (작업흐름-지향 배치 확장과 함께) 독립적인 스템을 순차적으로 처리
- 부분적인 처리하기: 기록 건너뛰기(예를 들면, 롤백시처럼)
- 전체적으로 적용되는 배치 트랜잭션

1. 기술적인 목적

- 배치 개발시에 스프링 프로그래밍 모델 사용: 비즈니스 로직에 집중하고, 인프라스트럭처는 프레임워크이 관리하도록
- 인프라스트럭처, 배치 실행 환경, 배치 애플리케이션 간의 완전한 SoC
- 모든 프로젝트에서 구현할 수 있는 공통되며, 핵심이 되는 실행 서비스를 인터페이스로 제공
- 모든 레이어에서 스프링 프레임워크를 활용해서 쉽게 환경 구성을 하고, 커스터마이징도 해서, 서비스를 확장
- 기존 핵심 서비스를 모두 인프라스트럭처 레이어에 큰 영향을 주지 않고 쉽게 교체하거나 확장할 수 있다.
- Maven을 사용해 구축함으로써 애플리케이션과 아키텍처 JAR는 완전히 분리해서 단순한 배포 모델 제공

1.1.3. 스프링 배치 아키텍처

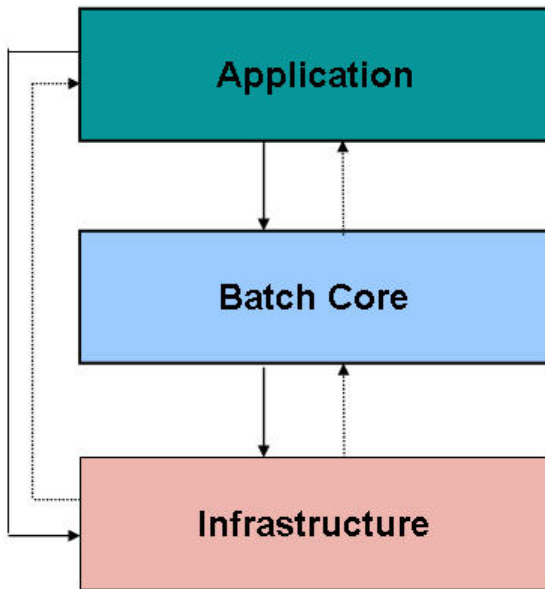
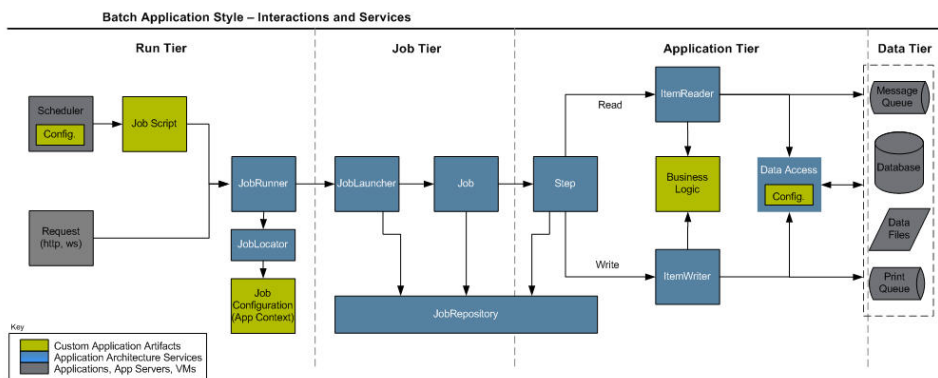


Figure 1.1: Spring Batch Layered Architecture

여기서 적용한 레이어 아키텍처는 세 개의 주요한 고수준 컴포넌트인 애플리케이션, 코어, 인프라스트럭처로 요약할 수 있다. 애플리케이션은 모든 배치 잡과 스프링 배치를 사용하는 개발자가 작성한 커스텀 코드를 모두 포함한다. 배치 코어는 배치 잡을 실행하고, 제어하는데 필요한 코어 런타임 클래스인 JobLauncher, Job, Step등을 포함한다. 애플리케이션과 코어 모두 공통이 되는 인프라스트럭처 위에 구축된다. 이 인프라스트럭처는 공통적으로 사용되는 reader, writer와 RetryTemplate처럼 애플리케이션 개발자나 코어 프레임워크 자체에서 사용되는 RetryTemplate과 같은 service 등을 포함한다.

2장 배치 도메인 언어

2.2. Batch Application Style Interactions and Services



구성

- 회색 박스
외부 애플리케이션을 표현한다. 스케줄링은 회색 박스로 표현되어 있는 것처럼 스프링 배치의 범위에 포함되지 않는다.
- 파란 박스
애플리케이션 아키텍처 서비스를 표현한다. 대부분 스프링 배치에 의해서 제공된다.
- 노란 박스

개발자에 의해서 구성되는 부분을 표현한다. Job schedule이나 Job 설정 파일 등을 예로 들수 있다.

논리적 티어

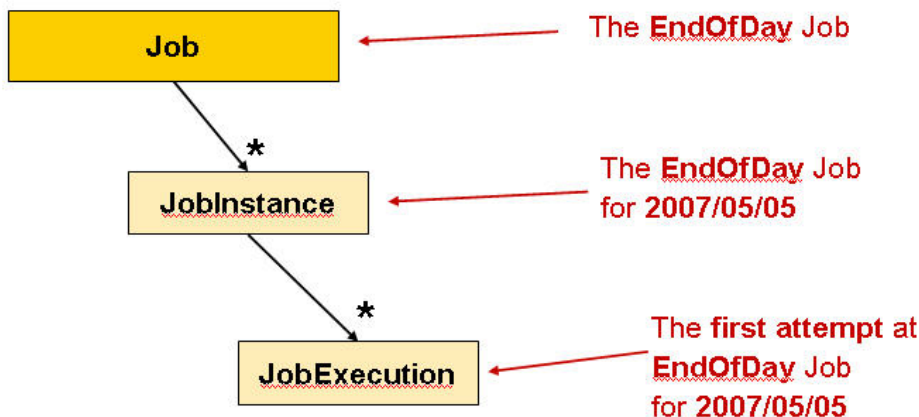
시스템 내의 SoC(Separation of Concern)을 위해 논리적으로 티어 구분한다. 개념적으로 구분했지만, 논리적인 컴포넌트 배포를 매핑할 때와 데이터 소스나 타겟과 통합하는데 효율적이다.

- Run 티어
스케줄링과 애플리케이션 실행에 관련.
- Job 티어
일반적으로 배치 잡의 실행을 책임. 연속적으로 배치 step을 실행시키고, 모든 step이 실행되서 정확한 상태에 있고, 모든 정책이 정확하게 적용됐는지를 보장
- Application 티어
프로그램을 실행하는데 필요한 컴포넌트를 포함. 배치 기능을 수행하는데 적용되는 특정 태스크와 정책을 적용.
- Data 티어
데이터베이스, 파일, 큐 등을 포함하는 물리적인 데이터 소스와 통합

2.3 Job 스테레오타입

잡(Job): 전체 배치 처리 과정을 캡슐화하는 엔티티

일반 스프링 프로젝트인 경우, Job은 XML 설정 파일을 통해서 의존성을 맺게 된다. 이 설정 파일은 "잡 환경 설정"으로 부른다. 그렇지만 잡은 단지 일반적인 상속 구조의 최상단에 있는 엔티티다.



2.3.1 Job

잡은 Job 인터페이스를 구현하는 스프링 빈으로 표현한다. 잡은 잡에 의해서 실행되는 오퍼레이션에 정의될 필요가 있는 모든 정보를 포함한다. 잡은 스프링 XML 설정 파일에 선언하고, 잡의 이름은 스프링 빈 id 속성으로 표현한다.

잡 설정은 다음 정보를 포함한다.

- 간단한 잡의 이름
- Step들의 정의와 순서
- 잡의 재시작 여부

Job 인터페이스의 기본적인 간단한 구현은 SimpleJob 클래스의 형식으로 스프링 배치에 의해서 제공된다. 이 클래스에는 다른 모든 잡에서 유용하게 사용할 수 있는 표준 실행 로직을 포함한다. 일반적으로 모든 잡은 SimpleJob 타입의 빈을 사용하도록 정의해야 한다.

```
<bean id="footballJob"
      class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
    </list>
  </property>
</bean>
```

```

    <bean id="gameLoad" parent="simpleStep" />
    <bean id="playerSummarization" parent="simpleStep" />
  </list>
</property>
<property name="restartable" value="true" />
</bean>

```

2.3.2. JobInstance

JobInstance는 논리적으로 잡 실행의 개념으로 이해하면 된다.

하루가 끝날 때 한 번씩 실행되는 'EndOfDay' 잡이 있다고 생각해보자. 'EndOfDay' Job은 하나지만, 매번 실행되는 Job은 개별적으로 추적되어야 한다. 이때 하루 당 하나의 논리적인 JobInstance가 있다고 보면 된다.

JobInstance의 정의는 로드되는 데이터와 절대적으로 관련이 있다. 데이터가 로드되는 방식을 결정하는 것은 전적으로 ItemReader 구현에 달려 있다. 이전 실행에서 사용한 'state'(예를 들어, ExecutionContext)가 사용되는지 마는지에 따라 동일한 JobInstance를 사용할지를 말지를 결정한다.

새로운 JobInstance를 생성한다는 것은 '시작에서부터 시작함'을 의미하고, 기존 인스턴스의 사용은 '이전에 그만둔데서 시작함'을 의미한다.

2.3.3. JobParameters

JobInstance를 어떻게 구분할까? 의 대답이 되는 것이 바로 JobParameters다. JobParameters는 배치 잡을 시작하는데 사용하는 파라미터의 집합으로, 잡이 실행되는 동안에 잡을 식별하거나 잡에서 참조하는 데이터로 사용된다.

앞서 예로 들었던 1월 1일에 시작하는 잡과 1월 2일에 시작하는 잡, 총 두개의 잡이 있었다. 이 두 잡을 구분하는 방법은 1월 1일, 1월 2일 이라는 날짜가 사용됐다. 결국 JobInstance를 구별할 수 있는 계약(contract)는 Job + JobParameters가 된다.

JobInstance = Job + JobParameters

개발자들은 전달하는 파라미터를 제어함으로써 JobInstance를 어떻게 정의할지를 효율적으로 제어할 수 있다.

2.3.4. JobExecution

JobExecution은 단 한 번 시도되는 Job 실행을 의미하는 기술적인 개념으로 이해하면 된다.

실행 자체는 성공이나 실패로 끝나지만, 실행에 대응되는 JobInstance는 실행이 성공적으로 종료된 경우에만 완료된 것으로 여긴다.

2008-01-01의 파라미터를 갖는 JobInstance가 처음 실행에서 실패했다고 가정해보자. 동일한 파라미터로 다시 실행하려는 경우에 새로운 JobExecution이 생성되지만, 여전히 JobInstance는 하나가 된다.

Job은 잡이 무엇을 하고, 어떻게 실행되는 지를 정의한다.

JobInstance는 함께 실행되는 그룹에 대한 순수한 의미의 유기적인 객체가 된다. 주로 JobInstance는 정확하게 재시작 의미를 가능하게 해주는 개념으로 생각하면 된다. 그렇지만 JobExecution은 주로 실행중에 실제로 어떤 일이 일어났는지에 대한 저장 메카니즘 역할을 하며, 제어되고, 영속화되는 좀 더 많은 프로퍼티들을 포함한다.

status	실행 상태를 가리키는 BatchStatus 객체. 실행하는 동안에는 BatchStatus.STARTED 가 되고, 실행이 실패한 경우에는 BatchStatus.FAILED, 성공적으로 종료됐을 때는 BatchStatus.COMPLETED 가 된다.
startTime	실행이 시작되는 당시 시스템 시간을 java.util.Date로 저장
endTime	실행의 성공 여부에 관계 없이, 실행이 끝나는 당시 시스템 시간을 java.util.Date로 저장
exitStatus	실행의 결과를 ExitStatus로 가리킨다. 호출자에게 반환될 종료 코드를 포함하기 때문에 매우 중요하다.(자세한 내용은 5장에..)
createTime	JobExecution이 처음으로 영속화 됐을 때 현재 시스템 시간을 java.util.Date로 표현. 이 때까지 잡이 시작되지 않을 수도 있지만, 이럴 때도 항상 createTime은 항상 가지고 있게 된다.

이 프로퍼티들은 영속화 되고, 실행 상태를 결정하는데 사용되기 때문에 매우 중요하다. 예를 들어, 01-01에 대한 EndOfDay 잡이 PM 09:00에 실행되고, 09:30에 실패하는 상황의 배치 메타 데이터 테이블에 다음 처럼 데이터가 들어간다.

Table 2.2. BATCH_JOB_INSTANCE

JOB_INSTANCE_ID	JOB_NAME
1	EndOfDayJob

Table 2.3. BATCH_JOB_PARAMS

JOB_INSTANCE_ID	TYPE_CD	KEY_NAME	DATE_VAL
1	DATE	schedule.Date	2008-01-01 00:00:00

Table 2.4. BATCH_JOB_EXECUTION

JOB_EXECUTION_ID	JOB_INSTANCE_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00:23.571	2008-01-01 21:30:17.132	FAILED

배치 잡은 실패했고, 원인을 찾아서 조치했다고 가정하자. 01-02에 실패했던 첫 번째 잡이 다시 09:00에 실행되고, 09:30에 정상적으로 종료하게 됐다. 그리고 01-02에 실행되어야 하는 잡은 이어서 09:31에 시작해서, 10:30에 성공적으로 끝났다고 해보자. 이 때 JobInstance, JobParameters, JobExecution에 데이터가 추가됐다.

Job이 실행되는 시점은 전적으로 스케줄러에 의해서 결정된다. 예에서 두 JobInstance는 서로 다른 개별적인 JobInstance기 때문에, 스프링 배치는 이 둘이 동시에 실행되는 것을 막으려고 하지 않는다.

Table 2.5. BATCH_JOB_INSTANCE

JOB_INSTANCE_ID	JOB_NAME
1	EndOfDayJob
2	EndOfDayJob

Table 2.6. BATCH_JOB_PARAMS

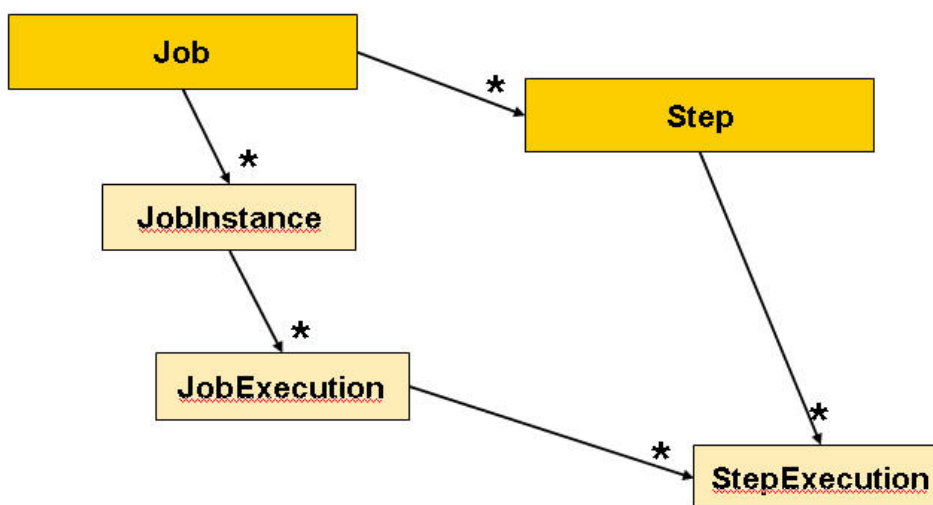
JOB_INSTANCE_ID	TYPE_CD	KEY_NAME	DATE_VAL
1	DATE	schedule.Date	2008-01-01 00:00:00
2	DATE	schedule.Date	2008-01-02 00:00:00

Table 2.7. BATCH_JOB_EXECUTION

JOB_EXECUTION_ID	JOB_INSTANCE_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

2.4. Step 스테레오타입

Step은 배치 잡의 독립적이고, 연속적인 단계를 캡슐화하는 도메인 객체다. 그러므로 모든 Job은 하나 이상의 step들로 구성되게 된다. 하나의 Step은 순차적으로 실행되는 유일한 처리과정의 흐름으로써 생각하면 되겠다. 각 step들은 다음 step으로 넘어가기 전에 완전하게 수행되어야 한다. Job처럼 Step도 유일한 JobExecution에 대응되는 개별 StepExecution을 가지고 있다.



2.4.1.Step

Step은 실제 배치 처리 과정을 정의하고, 제어하는데 필요한 모든 정보를 포함한다. 실제로 Step에 주어지는 내용은 전적으로 Job을 작성하는 개발자의 판단에 달려있기 때문에, 막연하게 설명할 수 밖에 없다.

Step은 Step 인터페이스의 구현체를 생성하는 것에 의해서 정의한다. 대부분의 경우에는 ItemOrientedStep 구현체로 충분하지만, 저장 프로시저 호출이나 기존 스크립트를 래핑하는 경우 등에는 TaskLetStep 이 더 좋은 선택이 된다.

2.4.2. StepExecution

StepExecution은 Step을 실행하는 한 번의 시도를 나타낸다. JobExecution에서 예를 든 내용에서, 한 번 실패한 JobInstance를 다시 실행할 때 새로운 StepExecution이 실행되게 된다. step 실행은 매번 배치 프레임웍에 의해서 별도로 호출되지만, 동일한 JobInstance에 대응된다. 즉, 이말은 하나의 JobInstance가 다수의 StepExecution을 갖을 수 있다는 말이다.

Step 실행은 StepExecution 클래스의 객체로 표현된다. 각 실행은 해당 step이 참조하는 Step과 JobExecution, 그리고 트랜잭션 관련 데이터(커밋, 롤백 카운트), 시작, 종료 시간 등을 포함한다. 또한 각 step 실행은 배치 실행에 걸쳐서 영속화될 필요가 있는 모든 데이터를 포함하는 ExecutionContext를 포함한다.

다음은 StepExecution의 프로퍼티 리스트다.

status	실행 상태를 가리키는 BatchStatus 객체. 실행하는 동안에는 BatchStatus.STARTED 가 되고, 실행이 실패한 경우에는 BatchStatus.FAILED, 성공적으로 종료됐을 때는 BatchStatus.COMPLETED 가 된다.
startTime	실행이 시작되는 당시 시스템 시간을 java.util.Date로 저장
endTime	실행의 성공 여부에 관계 없이, 실행이 끝나는 당시 시스템 시간을 java.util.Date로 저장
exitStatus	실행의 결과를 ExitStatus로 가리킨다. 호출자에게 반환될 종료 코드를 포함하기 때문에 매우 중요하다.(자세한 내용은 5 장에..)
executionContext	실행 사이에 영속화 되어 할 필요가 있는 모든 사용자 데이터를 포함하는 '프로퍼티 bag'
commitCount	이번 실행에서 커밋된 트랜잭션의 개수
itemCount	이번 실행에서 처리된 아이템의 개수
rollbackCount	롤백된 Step에 의해서 제어된 비즈니스 트랜잭션의 개수
readSkipCount	읽기가 실패해서, 결과적으로 지나친 아이템의 갯수
writeSkipCount	쓰기가 실패해서, 결과적으로 지나친 아이템의 갯수

2.4.3. ExecutionContext

ExecutionContext는 StepExecution이나 JobExecution 범위에서 영속화 상태를 저장하기 위해 프레임웍에 의해서 영속화되고, 제어되는 키/값 쌍의 컬렉션으로 표현된다. 이 개념은 Quartz의 JobDataMap과 비슷하다.

ExecutionContext를 가장 잘 사용하는 예는 'restart'다. 개별 라인을 처리하는 과정 동안 예제에서 플랫 파일 입력을 사용할 때 프레임웍은 주기적으로 커밋 지점에서 ExecutionContext를 영속화 한다. 이는 ItemReader가 실행 중에 fatal 에러가 발생한 경우에 상태를 저장하거나 심지어 중단(power go out)시킬 수 있도록 해준다. 현재 라인 수로 넣어 두어야 할 필요가 있는 모든 내용을 context에 읽어 오게 되고, 프레임웍이 나머지 일을 맡아서 해준다.

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

지금까지 예로 봐온 EndOfDay에서 'loadData' step 하나가 있다고 가정하자. 이 step은 파일에서 데이터베이스로 데이터를 로드한다. 첫 번째 실행이 실패하면 메타 데이터 테이블은 다음 처럼 될 것이다.

Table 2.9. BATCH_JOB_INSTANCE

JOB_INSTANCE_ID	JOB_NAME
1	EndOfDayJob

Table 2.10. BATCH_JOB_PARAMS

JOB_INSTANCE_ID	TYPE_CD	KEY_NAME	DATE_VAL
1	DATE	schedule.Date	2008-01-01 00:00:00

Table 2.11. BATCH_JOB_EXECUTION

JOB_EXECUTION_ID	JOB_INSTANCE_ID	START_TIME	END_TIME	STATUS
1	1	2008-01-01 21:00:23.571	2008-01-01 21:30:17.132	FAILED

Table 2.12. BATCH_STEP_EXECUTION

STEP_EXECUTION_ID	JOB_EXECUTION_ID	STEP_NAME	START_TIME	END_TIME	STATUS
1	1	loadDate	2008-01-01 21:00:23.571	2008-01-01 21:30:17.132	FAILED

Table 2.13. BATCH_EXECUTION_CONTEXT

EXECUTION_ID	TYPE_CD	KEY_NAME	LONG_VAL
1	LONG	piece.count	40321

이 테이블을 보면 이 Step은 30분 동안 실행됐고, 40,321 개(이번 예에서는 처리되는 파일의 라인을 나타냄)를 처리했다는 것을 알 수 있다. 이 값은 프레임워크에 의해서 매번 커밋되기 전에 갱신되게 되며, ExecutionContext의 엔트리에 대응되어 다수의 행을 포함할 수도 있다.

커밋되기 전에 통지 받도록 다양한 StepListener나 ItemStream가 필요하기도 한다. 이전 예에서 EndOfDay 잡은 실패했을 때 다음 날(day)에 다시 시작하도록 되었었다. 재시작되면, 마지막 실행의 ExecutionContext에 있는 값이 데이터베이스에서 재구성되고, ItemReader가 열리면, context에 저장된 상태를 갖고 있는지 없는지를 확인해서, 이에 따라 초기화되게 할 수 있다.

```
if (executionContext.containsKey(getKey(LINES_READ_COUNT))) {
    log.debug("Initializing for restart. Restart data is: " + executionContext);

    long lineCount = executionContext.getLong(getKey(LINES_READ_COUNT));

    LineReader reader = getReader();

    Object record = "";
    while (reader.getPosition() < lineCount && record != null) {
        record = readLine();
    }
}
```

이번 예에서는 위 코드가 실행된 다음의 현재 라인은 40,322가 되고, Step은 이전에 정지된 지점에서 다시 시작하도록 해준다.

ExecutionContext는 실행에 대해 영속화될 필요가 있는 내용에 대한 통계로 사용될 수도 있다.

//TODO

2.5. JobRepository

JobRepository는 위에서 언급한 모든 스테레오타입에서 사용하는 영속화 메카니즘이다. 잡이 처음으로 실행되면, JobExecution이 저장소(repository)의 createJobExecution() 메소드를 호출해서 받아온다. 실행 동안에는 StepExecution과 JobExecution은 이 저장소에 건네져서 영속화 된다.

```
public interface JobRepository {

    public JobExecution createJobExecution(Job job, JobParameters jobParameters)
        throws JobExecutionAlreadyRunningException, JobRestartException;

    void saveOrUpdate(JobExecution jobExecution);

    void saveOrUpdate(StepExecution stepExecution);

    void saveOrUpdateExecutionContext(StepExecution stepExecution);

    StepExecution getLastStepExecution(JobInstance jobInstance, Step step);

    int getStepExecutionCount(JobInstance jobInstance, Step step);

}
```

2.6 JobLauncher

JobLauncher는 주어진 JobParameters로 Job을 실행시키는 단순한 인터페이스다.

```
public interface JobLauncher {  
  
    public JobExecution run(Job job, JobParameters jobParameters) throws  
        JobExecutionAlreadyRunningException, JobRestartException;  
}
```

이 인터페이스 구현에서는 JobRepository에서 알맞은 JobExecution을 받아와서 Job을 실행하도록 구현됨을 기대하게 된다.

2.7. JobLocator

JobLocator는 Job을 찾아주는 인터페이스다.

```
public interface JobLocator {  
    Job getJob(String name) throws NoSuchJobException;  
}
```

이 인터페이스는 스프링 자체의 특성때문에 매우 필요하다. 하나의 ApplicationContext가 하나의 Job과 동일함을 보장할 수 없기 때문에, 주어진 이름으로 Job을 얻어오도록 추상화하는 작업이 필요하다. 특히 자바 EE 애플리케이션 서버에서 잡을 실행할 때 매우 유용하다.

2.8. Item Reader

ItemReader는 한 번에 하나의 아이템 별로 Step에서 입력을 받아오는 표현을 추상화한다. ItemReader가 제공할 수 있는 아이템을 다 소비했을 때는, null을 반환하도록 지정하면 된다.(자세한 내용은 3장 참조)

2.9. Item Writer

ItemWriter는 한 번에 하나의 아이템 별로 한 Step의 출력하는 표현을 추상화한다. 일반적으로 item writer는 다음 번에 받아오는 입력에 대해 전혀 알지 못한다. 단지 현재 호출에서 건내지는 item에 대해서만 알 뿐이다. (자세한 내용은 3장 참조)

2.10. Item Processor

ItemProcessor는 항목의 비즈니스 처리 과정 표현을 추상화한다. ItemReader가 하나의 항목을 읽어들이고, ItemWriter가 이를 쓰는 동안에, ItemProcessor는 변환을 위한 접근이나 다른 비즈니스 처리 적용을 제공한다. 항목 처리 중 처리 과정이 유효하지 못하다고 결정됐다면, 쓰기 작업이 되지 않았다는 것을 나타내는 null을 반환한다.

h2 2.11. Tasklet

Tasklet은 Tasklet 인터페이스로 제공되는 스프링 배치의 구현체에 의해서 정의되며, 논리적인 작업 단위의 실행을 표현한다. Tasklet은 저장 프로시저나 시스템 커맨드처럼 자연스럽게 읽기-(변환)-쓰기 단계로 분리할 수 없는 처리과정 로직을 캡슐화하는데 유용하다.

3장 ItemReaders와 ItemWriters

3.1. 소개

모든 배치 처리 과정은 대부분 대규모 데이터를 읽어 들이고, 특정 타입으로 계산하거나 변환되서, 이 결과를 쓰는 형태로 설명할 수 있다. 스프링 배치는 이러한 벌크 읽기, 쓰기를 도와주는 ItemReader와 ItemWriter 인터페이스를 제공한다.

3.2. ItemReader

간단한 개념이기는 하지만, ItemReader는 다양한 타입의 입력을 받을 수 있다. 가장 대표적인 예로 플랫(Flat) 파일, XML, 데이터베이스를 들 수 있다.

- 플랫 파일: 플랫 파일 Item reader는 파일에서 고정된 위치로 정의된 데이터 필드나 특수 문자에 의해서 구별되는 데이터로 기록된 플랫 파일의 각 행을 읽게 된다.
- XML: 파싱, 매핑, 유효성 검증을 XML에서 독립적으로 작업할 수 있게 처리해준다. 입력 받은 데이터는 XSD 스키마에 맞춰 유효성 검증이 가능하다.
- 데이터베이스: 데이터베이스 자원에는 처리과정 중에 객체로 매핑될 수 있는 resultset을 반환해서 접근하게 된다. 기본적인 SQL ItemReaders는 객체를 반환하는 RowMapper를 호출한다. 여기에 더해서 제시작 때 현재 가리키는 행을 유지하고, 기본적인 통계, 트랜잭션 강화 등의 기능을 제공한다.

기본적인 ItemReader 인터페이스를 살펴보자.

```
public interface ItemReader {

    Object read() throws Exception;

    void mark() throws MarkFailedException;

    void reset() throws ResetFailedException;

}
```

read() 메소드는 ItemReader의 중심이 되는 계약으로, 결과값으로 하나의 Item을 반환한다. 더 이상 반환할 Item이 없는 경우 null을 반환한다. 여기서 아이템은 플랫 파일에서는 한 행(line), 데이터베이스에서의 한 행(row), XML 파일에서의 엘리먼트(element)를 나타낸다.

mark()와 reset() 메소드는 배치 처리과정의 트랜잭션 처리에서 중요하게 사용된다.

mark()는 읽기 작업을 시작하기 전에 호출된다. reset()은 mark()가 마지막으로 호출된 위치로 ItemReader를 위치시키게 된다. 이런 논리적인 흐름은 java.io.Reader와 매우 비슷하다.

또한 ItemReader를 처리하는 과정에서 아이템이 부족하다고 예외를 던지지 않는다.

3.3. ItemWriter

ItemWriter는 ItemReader와 비슷한 기능을 갖지만, 정반대의 오퍼레이션을 제공한다.

```
public interface ItemWriter {

    void write(Object item) throws Exception;

    void flush() throws FlushFailedException;

    void clear() throws ClearFailedException;

}
```

write() 메소드는 ItemWriter의 가장 기본적인 계약이며, 인자로 건넨 객체가 열려 있는 동안 쓰기 작업을 시도한다.

flush()와 clear()는 배치 처리 작업에 대한 트랜잭션 처리에서 필요한 메소드다. flush()는 아이템의 쓰기 작업을 실행해서 버퍼를 비우는 반면에, clear()는 간단하게 내용을 제거해서 버퍼를 비운다. 일반적으로 Step 구현할 때는 커밋 전에 flush()를 호출하고, 롤백하기 전에 clear()를 호출한다.

ItemStream

ItemReader와 ItemWriter는 각자 목적이 있고, 서로 다른 일을 하지만, 공통적으로 영속화 상태에 대한 메커니즘이 필요하다.

```
public interface ItemStream {

    void open(ExecutionContext executionContext) throws StreamException;

    void update(ExecutionContext executionContext);

    void close(ExecutionContext executionContext) throws StreamException;

}
```

```
}
```

각 메소드를 기술하기 전에, 간단하게 ExecutionContext를 언급할 필요가 있다. ItemReader의 클라이언트들은 ItemStream을 구현해야 하며, read()를 호출하기 전에 파일이나 커넥션과 같은 자원을 얻어내기 위해서 open() 메소드를 호출해야 한다. 이 내용은 ItemWriter에도 동일하게 적용된다.

2장에서 살펴봤듯이, 필요한 데이터를 ExecutionContext에서 찾았다면, 최초 상태가 아닌 해당 위치에서 ItemReader나 ItemWriter를 시작하는데 사용될 수도 있다.

반대로 말해서 close() 메소드는 open()에서 할당된 모든 자원을 안전하게 릴리스 된 것을 보장하도록 호출된다.

update()는 현재 유지하고 있는 모든 상태를 ExecutionContext로 불러들여는 것을 보장하기 위해서 호출한다. 이 메소드는 커밋되기 전에 데이터베이스에 이 상태를 영속화 하도록 커밋되기 전에 호출된다.

ItemStream의 클라이언트가 특정 실행의 상태를 사용자가 저장할 수 있도록 각 StepExecution에서 생성된 ExecutionContext를 위한 Step인 이런 특별한 상황에서는 동일한 JobInstance가 다시 시작되는 경우에 이 상태가 반환되기를 기대하게 된다. Quartz와 매우 비슷하게도, 이 의미는 Quartz JobDataMap과 상당히 비슷하다.

3.5. 플랫(Flat) 파일

별크 데이터를 주고 받는 가장 일반적인 메카니즘 중 하나가 플랫 파일이다. 구조를 구성하는 방법이 XSD로 표준화되어 정의되는 XML과는 다르게, 플랫 파일은 그전에 반드시 파일이 구성되어 있는 방법에 대한 이해가 필요하다.

일반적으로 플랫 파일은 구분자(Delimited)와 고정된 길이(Fixed Length)의 두 가지 일반적인 타입으로 되어 있다.

3.5.1. FieldSet

입력이든 출력이든지에 관계 없이 스프링 배치에서 가장 중요한 클래스 중 하나가 FieldSet 이다.

다른 배치 프레임워크나 라이브러리는 파일에서 String이나 String의 배열을 받아들일 수 있도록 도와주는게 추상화를 제공하는게 일반적이다.

이에 반해 스프링 배치는 파일 자원에서 필드를 바인딩할 수 있게 해주는 추상화 레벨로 FieldSet을 제공한다. 그렇기 때문에 개발자들은 파일을 사용할 때도 데이터베이스를 대상으로 작업하듯이 할 수 있게 해준다.

FieldSet은 JDBC ResultSet과 상당히 비슷한 개념이다. FieldSet은 토큰의 String 배열만 인자로 받는다. ResultSet처럼 인덱스나 이름으로 필드에 접근하기 위해서 필드의 이름을 설정할 수도 있다.

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

FieldSet 인터페이스에는 Date, BigDecimal 등과 같은 추가적인 다양한 옵션을 제공한다. FieldSet이 제공해주는 가장 큰 이점은 입력 파일에 대한 일관적인 파싱 제공이다. 또한 일관적인 예외 제어와 데이터 변환 방법을 제공한다.

3.5.2. FlatFileItemReader

플랫 파일은 대부분 두 가지 영역의 데이터를 포함하는 모든 타입의 파일이다. 스프링 배치에서는 플랫 파일을 읽고, 파싱하는 기본적인 기능을 제공하는 FlatFileItemReader 클래스에 의해서 도움을 받게 된다.

FlatFileItemReader는 Resource, FieldSetMapper, LineTokenizer에 기본적으로 의존성을 갖는다. Resource는 스프링 코어 Resource 인터페이스를 사용한다. (4장 참조) 자세한 내용은 스프링 코어 레퍼런스 4장을 참조하고, 여기서는 간단하게 Resource 객체를 생성하는 방법만 살펴보자.

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

복잡한 배치 환경에서 디렉토리 구조는 EAI 인프라스트럭처에 의해서 관리되기도 한다. 파일 이동 유틸리티는 스프링 배치 아키텍처의 범위 아래 있지만, job stream의 step에 대해서 까지도 파일 이동 유틸리티를 포함하는 배치 잡 stream은 흔치 않다. 배치 아키텍처는 단지 처리되는 파일의 위치를 알기만 하면 된다. 스프링 배치는 이러한 시작 지점에서 데이터를 처리하는 과정을 시작한다.

FlatFileItemReader는 데이터를 해석하는 방법을 더 구체적으로 정할 수 있도록 다른 프로퍼티들을 제공한다.

프로퍼티	타입	설명
encoding	String	사용하는 텍스트 인코딩 설정 - 기본은 'ISO-8859-1'
comments	String[]	코멘트 행을 지정하는 줄 앞 첨저(line prefixes) 지정
lineToSkip	int	파일이 시작되는 최상단에서 부터 무시할 줄의 수
firstLineIsHeader	boolean	파일의 첫 번째 행이 필드의 이름을 포함하는 헤더인지를 지정. 아직 필드 이름이 설정되어 있지 않은 경우에는 AbstractLineTokenizer를 상속한 tokenizer를 사용하는 경우, 첫 번째 줄에 자동으로 필드 이름이 설정되게 된다.
recordSeparatorPolicy	RecordSeparatorPolicy	줄이 끝나는걸 결정하고, 따옴표(quoted) 문자열이 있는 경우 한 줄이 넘어가도 계속 같은 줄로 처리할 것인지 결정하는데 사용

3.5.2.1. LineMapper

As with RowMapper, which takes a low level construct such as ResultSet and returns an Object, flat file processing requires the same construct to convert a String line into an Object:
ResultSet을 처리해서 Object를 반환하는 저 수준 구성을 하는 RowMapper처럼, 플랫폼 파일 처리 과정에서는 String 한 줄을 Object로 변경해주기 위해 동일한 구성을 필요로 한다.

```
public interface LineMapper<T> {
    T mapLine(String line, int lineNumber) throws Exception;
}
```

3.5.2.2. FieldSetMapper

FieldSetMapper 인터페이스는 FieldSet 객체에서 매핑하는 객체를 생성해서 반환하는 mapLine 메소드를 정의하고 있다. 이 객체는 커스텀 DTO가 될 수도 있고, 도메인 객체나 간단한 배열이 될 수도 있다. FieldSetMapper는 리소스의 데이터 한 줄에서 원하는 타입의 객체로 변환하는데 LineTokenizer와 협력한다.

```
public interface FieldSetMapper<T> {
    T mapFieldSet(FieldSet fieldSet);
}
```

JdbcTemplate에서 사용하는 RowMapper와 같은 패턴을 사용한다.

3.5.2.3. LineTokenizer

플랫 파일 데이터의 포맷은 정말 많기 때문에, 입력의 한 줄을 FieldSet으로 변환하는 작업을 추상화할 필요가 있다. 이를 스프링 배치에서는 LinkTokenizer라 부른다.

```
public interface LineTokenizer {
    FieldSet tokenize(String line);
}
```

LineTokenizer는 입력의 한 줄(이론적으로 String은 한 줄이 아닐수도 있지만..)을 받아서 FieldSet을 반환한다. 스프링 배치는 몇 가지 LineTokenizer를 포함하고 있다.

- DelimitedLineTokenizer: 구분자에 의해서 각 레코드를 구분하는 파일에 사용. 일반적으로 , 를 사용하지만, | 나 ; 을 사용하기도 한다.
- FixedLenghTokenizer: 정해진 길이로 각 레코드를 구분하는 토큰나이저
- PrefixMatchingCompositeLineTokenizer: 앞 첨자(prefix)를 확인해서 레코드를 구분하는 토큰나이저

3.5.2.4. 간단한 구분자로 구분되는 파일 읽기 예제

기본적인 인터페이스를 살펴봤으니 간단한 예제를 살펴보자. 가장 간단한 형식은 다음과 같은 흐름을 갖게 된다.

1. 파일에서 한 줄 읽기
2. 한 줄 문자열을 LineTokenizer.tokenize() 메소드에 전달해서 FieldSet을 받아 옴
3. 반환 받은 FieldSet을 FieldSetMapper에게 전달하고, 그 결과 객체를 ItemReader.read()에서 반환한다.

이 흐름을 코드로 살펴보면,

```
String line = readLine();

if (line != null) {
    FieldSet tokenizedLine = tokenizer.tokenize(line);
    return fieldSetMapper.mapLine(tokenizedLine);
}

return null;
```

다음은 실제 도메인 시나리오를 사용한 예제다. 이 잡은 다음 파일에서 특정 축구 선수를 읽어 들인다.

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

이 파일의 내용은 다음 Player 도메인 객체로 매핑된다.

```
public class Player implements Serializable {

    private String ID;
    private String lastName;
    private String firstName;
    private String position;
    private int birthYear;
    private int debutYear;

    public String toString() {

        return "PLAYER:ID=" + ID + ",Last Name=" + lastName +
            ",First Name=" + firstName + ",Position=" + position +
            ",Birth Year=" + birthYear + ",DebutYear=" +
            debutYear;
    }

    // setter# getter...
}
```

FieldSet을 Player 객체로 매핑하려면, FieldSetMapper를 정의하면 된다.

```
protected static class PlayerFieldSetMapper implements FieldSetMapper {
    public Object mapLine(FieldSet fieldSet) {
        Player player = new Player();
```

```

        player.setID(fieldSet.readString(0));
        player.setLastName(fieldSet.readString(1));
        player.setFirstName(fieldSet.readString(2));
        player.setPosition(fieldSet.readString(3));
        player.setBirthYear(fieldSet.readInt(4));
        player.setDebutYear(fieldSet.readInt(5));

        return player;
    }
}

```

다음으로 파일은 구성된 FlatFileItemReader에 의해서 읽혀 들어지고, read()를 호출한다.

```

FlatFileItemReader itemReader = new FlatFileItemReader();
itemReader.setResource(new FileSystemResource("resources/players.csv"));
//DelimitedLineTokenizer defaults to comma as it's delimiter
itemReader.setLineTokenizer(new DelimitedLineTokenizer());
itemReader.setFieldSetMapper(new PlayerFieldSetMapper());
itemReader.open(new ExecutionContext());
Player player = (Player)itemReader.read();

```

read를 호출할 때마다 파일에 있는 각 라인에 매핑되는 새로운 Player 객체를 반환하게 된다. 파일이 끝나게 되면, null를 반환한다.

3.5.2.5. 이름으로 필드 매핑하기

LineTokenizer에는 JDBC ResultSet과 비슷한 기능을 가지고 있다. 필드의 이름을 LineTokenizer에 주입해서 매핑의 가독성을 높일 수 있다. 먼저 플랫폼 파일에 있는 모든 필드의 이름은 LineTokenizer에 주입해보자.

```

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

```

FieldSetMapper는 다음처럼 이 정보를 사용할 수 있다.

```

public class PlayerMapper implements FieldSetMapper {
    public Object mapLine(FieldSet fs) {

        if(fs == null){
            return null;
        }

        Player player = new Player();
        player.setID(fs.readString("ID"));

        player.setLastName(fs.readString("lastName"));
        player.setFirstName(fs.readString("firstName"));
        player.setPosition(fs.readString("position"));
        player.setDebutYear(fs.readInt("debutYear"));
        player.setBirthYear(fs.readInt("birthYear"));

        return player;
    }
}

```

3.5.2.6. FieldSet을 도메인 객체로 자동 매핑하기

매번 FieldSetMapper를 작성하기는 매우 성가신 일이다. 스프링 배치는 자반 빈즈 규약을 사용해서 객체의 필드 이름과 일치하는 setter 메소드를 통해서 자동으로 필드 매핑을 해준다. 다시 축구 예제를 살펴보자. FieldSetMapper는 다음처럼 설정한다.

```

<bean id="fieldSetMapper"

```

```

        class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
        <property name="prototypeBeanName" value="player" />
    </bean>

    <bean id="player"
        class="org.springframework.batch.sample.domain.Player"
        scope="prototype" />

```

FieldSet의 각 엔트리를 위해 매핑하는 Player 객체의 새로운 인스턴스에서 프로퍼티 이름에 일치하는 setter 메소드를 찾아준다. FieldSet에서 각 필드를 사용해서 매핑하고, Player 객체를 반환하는데 까지 코드가 전혀 필요 없다.

3.5.2.7. 고정된 길이(fixed length) 파일 포맷

많은 조직들은 고정된 길이를 갖는 플랫폼 파일은 사용한다.

```

UK21341EAH4121131.11customer1
UK21341EAH4221232.11customer2
UK21341EAH4321333.11customer3

UK21341EAH4421434.11customer4
UK21341EAH4521535.11customer5

```

길이가 긴 위 필드는 실제로 총 네 개의 필드로 구분된다.

- ISIN: 12 캐릭터
- Quality: 3 캐릭터
- Price: 4 캐릭터
- Customer: 8 캐릭터

FixedLengthLineTokenizer를 구성할 때, 각 길이는 범위 형식으로 제공되어야만 한다.

```

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

```

여기서도 구분자를 사용한 LineTokenizer를 사용할 때처럼 동일하게 FieldSet을 반환하고, BeanWrapperFieldMapper처럼 실제 라인이 어떻게 파싱되는지 관계 없이 동일한 접근 방법을 갖도록 해준다.

ApplicationContext에서 어디든 컬럼의 범위를 지정해주는데 사용하는 특화된 프로퍼티 에디터를 선언해줘야 하는 것을 기억하자.

```

<bean id="customEditorConfigurer"
    class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="org.springframework.batch.item.file.transform.Range[]">
                <bean class="org.springframework.batch.item.file.transform.RangeArrayPropertyEditor" />
            </entry>
        </map>
    </property>
</bean>

```

3.5.2.8. 하나의 파일 안에서 다양한 레코드 타입 사용하기

지금까지는 한 줄이니 하나의 레코드로 매핑됐지만, 실제로 항상 그렇지는 않다. 파일 내에 여러 줄에 걸쳐서, 여러 포맷으로 매핑되는 것은 매우 일반적이다. 예제가 되는 일부 파일에서 내용을 뽑아 보면,


```

HEA;0013100345;2007-02-15
NCU;Smith;Peter;;T;20014539;F
BAD;;Oak Street 31/A;;Small Town;00235;IL;US
SAD;Smith, Elizabeth;Elm Street 17;;Some City;30011;FL;United States
BIN;VISA;VISA-12345678903
LIT;1044391041;37.49;0;0;4.99;2.99;1;45.47
LIT;2134776319;221.99;5;0;7.99;2.99;1;221.87
SIN;UPS;EXP;DELIVER ONLY ON WEEKDAYS
FOT;2;2;267.34

```

'HEA'와 'FOT'로 시작하는 사이에 모든 내용은 하나의 레코드로 취급한다.

PrefixMatchingCompositeLineTokenizer는 특정 tokenizer에 있는 줄의 앞 첨자를 매핑해서 이를 더 쉽게 해준다.

```

<bean id="orderFileDescriptor"
      class="org.springframework.batch.io.file.transform.PrefixMatchingCompositeLineTokenizer">
  <property name="tokenizers">
    <map>
      <entry key="HEA" value-ref="headerRecordDescriptor" />
      <entry key="FOT" value-ref="footerRecordDescriptor" />
      <entry key="BCU" value-ref="businessCustomerLineDescriptor" />
      <entry key="NCU" value-ref="customerLineDescriptor" />
      <entry key="BAD" value-ref="billingAddressLineDescriptor" />
      <entry key="SAD" value-ref="shippingAddressLineDescriptor" />
      <entry key="BIN" value-ref="billingLineDescriptor" />
      <entry key="SIN" value-ref="shippingLineDescriptor" />
      <entry key="LIT" value-ref="itemLineDescriptor" />
      <entry key="" value-ref="defaultLineDescriptor" />
    </map>
  </property>
</bean>

```

이는 해당 줄이 정확하게 파싱되는 지를 보장한다. 이 시나리오에서 FlatFileItemReader의 모든 사용자는 레코드의 footer가 반환될 때까지 계속해서 read를 호출해서 하나의 '아이템'으로서 완벽한 주문을 반환하도록 해준다.

3.5.2.9. 플랫 파일에서 예외 처리하기

한 행을 토큰으로 나눌 때 예외가 발생할 수 있는 시나리오는 정말 많다. 대부분의 플랫 파일이 완벽하지 않으며, 포맷을 정확하게 지키지 않는 기록을 포함한다. 대다수의 사용자는 이런 행을 예외를 발생시켜, 로그로 남기는 방법을 선택한다. 이런 이유 때문에 스프링 배치는 예외를 파싱해서 제외하기 위한 예외 계층구조로 FlatFileParseException와 FlatFileFormatException를 제공한다. FlatFileParseException는 FlatFileItemReader가 플랫 파일을 읽어들이는 동안에 던지는 예외다. FlatFileFormatException는 LineTokenizer 인터페이스의 구현체가 던지는 예외며, 토큰 처리를 하는 동안에 발생하는 예외를 좀더 상세하게 지정해준다.

3.5.2.9.1. IncorrectTokenCountException

DelimitedLineTokenizer와 FixedLengthLineTokenizer는 FieldSet을 만들면서 사용하는 컬럼 이름을 지정해주는 기능을 제공한다. 그렇지만 컬럼 이름의 수가 토큰 처리를 하는 동안 식별해낸 컬럼의 수와 일치하지 않는다면 FieldSet을 만들 수 없고, 이 때 IncorrectTokenCountException을 던진다. 이 예외는 발견한 토큰의 수와 기대한 수를 포함한다.

```

tokenizer.setNames(new String[] { "A", "B", "C", "D" });

try{
    tokenizer.tokenize("a,b,c");
}
catch(IncorrectTokenCountException e){
    assertEquals(4, e.getExpectedCount());
    assertEquals(3, e.getActualCount());
}
}

```

3.5.2.9.2. IncorrectLineLengthException

고정된 길이를 사용하는 형식이 지정된 파일은 파싱할 때 추가 요구사항이 발생한다. 구분자를 사용하는 형식과는 다르게 각 컬럼은 이미 정해진 길이를 엄격하게 따라야만 한다. 전체 행 길이가 해당 컬럼의 가장 넓은 값까지 추가되지 않은 경우에는 예외가 발생한다.

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

tokenizer에 구성된 범위는 1-5, 6-10, 11-15므로 기대하는 전체 행의 길이는 15가 된다. 그러나 이 예의 경우에 한 줄의 길이가 5인 값을 건넴으로 IncorrectLineLengthException을 던지게 되는 원인이 된다. 일찍 실패가 되도록 행의 첫 번째 컬럼을 처리하기 보다는 예외를 바로 던지도록 했다. 토큰 처리가 실패한 경우, FieldSetMapper에서 두 번째 컬럼을 읽으려고 시도하는 동안 실패하도록 한 것 보다는 더 많은 정보를 가지고 있을 것이다. 그렇지만 이 시나리오에서는 행의 길이가 항상 동일하지 않다. 그렇기 때문에 행 길이에 대한 유효성 검증은 'strict' 프로퍼티를 통해서 기능을 끌 수 있다.

```
tokenizer.setColumns(new Range[] { new Range(1, 5), new Range(6, 10) });
tokenizer.setStrict(false);
FieldSet tokens = tokenizer.tokenize("12345");
assertEquals("12345", tokens.readString(0));
assertEquals("", tokens.readString(1));
```

위 예제는 tokenizer.setStrict(false)를 호출한 부분을 제외하고는 이전 예제와 거의 동일하다. 이 설정은 행을 토큰 처리할 때 행의 길이를 강제로 정하지 말라고 tokenizer에게 요청하게 된다. 그렇지만 남은 값에 대한 토큰은 빈 토큰을 갖게 된다.

3.5.3. FlatFileWriter

플랫 파일을 쓰기는 플랫 파일을 읽어 들일 때 극복해야 할 때와 동일한 문제와 이슈를 갖고 있다. 전통적인 방법으로 플랫 파일을 구분자나 고정된 길이의 포맷으로 쓸 수 있어야 한다.

3.5.3.1. LineAggregator

문자열을 토큰으로 나눌 필요가 있는 LineTokenizer와 동일하게, 파일 쓰기도 다수의 필드를 하나의 문자열로 모으는 방법을 가지고 있어야 한다. 스프링 배치에서는 LineAggregator를 제공한다.

```
public interface LineAggregator {
    public String aggregate(FieldSet fieldSet);
}
```

LineAggregator는 LineTokenizer에 반대에 위치한다. LineTokenizer는 String을 받고, FieldSet을 반환한다. 반면에 LineAggregator는 FieldSet을 받고, String을 반환한다. 스프링 배치에서는 두 가지를 LineAggregator를 제공한다.

3.5.3.2. 간단하게 구분자로 구분되는 파일 쓰기 예제

이제 LineAggregator와 FieldSetCreator 인터페이스를 정의해서, 쓰기 작업을 하는 기본 흐름을 살펴보자.

1. 쓰기 작업하는 객체는 FieldSet을 얻기 위해서 FieldSetCreator로 건네진다.
2. 반환되는 FieldSet은 LineAggregator로 건네진다.
3. 반환되는 String은 설정된 파일로 쓰여진다.

다음 코드는 FlatFileItemWriter에서 일부를 발췌한 내용이다.

```
public void write(Object data) throws Exception {
    FieldSet fieldSet = fieldSetCreator.mapItem(data);
```

```
getOutputState().write(lineAggregator.aggregate(fieldSet) + LINE_SEPARATOR);  
}
```

가장 작은 setter가 사용되도록 해주는 간단한 설정은 다음과 같다.

```
<bean id="itemWriter"  
    class="org.springframework.batch.io.file.FlatFileItemWriter">  
    <property name="resource"  
        value="file:target/test-outputs/20070122.testStream.multilineStep.txt" />  
    <property name="fieldSetCreator">  
        <bean class="org.springframework.batch.io.file.mapping.PassThroughFieldSetMapper"/>  
    </property>  
</bean>
```

3.5.3.3. 파일 생성 제어하기

FlatFileItemReader는 파일 자원과 매우 약한 연관 관계를 갖는다. 리더가 초기화됐을 때, 파일이 있으면 열고, 없으면 예외를 던진다. 파일 쓰기는 리더처럼 단순하지는 않다. 기존에 파일이 있는 경우에는 예외를 던지고, 없는 경우, 새로 생성해서 시작한다.

그렇지만 Job의 재시작이 문제가 될 수 있다. 일반적인 재시작 시나리오에서는 계약이 반대가 된다. 기존에 파일이 있으면 정상적으로 처리된 마지막 위치를 알아내서 계속 처리하면 되고, 파일이 없는 경우에는 예외를 던진다.

그렇지만 항상 파일 이름이 같을까? 이런 경우에 기존 파일이 있다면 파일을 삭제하고 싶을 수도 있다. 스프링 배치에서는 이런 시나리오를 처리하기 위해서 FlatFileItemReader에 shouldDeleteIfExists 프로퍼티를 포함하고 있다. 이 프로퍼티가 true로 설정되어 있다면, writer가 열릴 때 기존 파일을 삭제하게 된다.

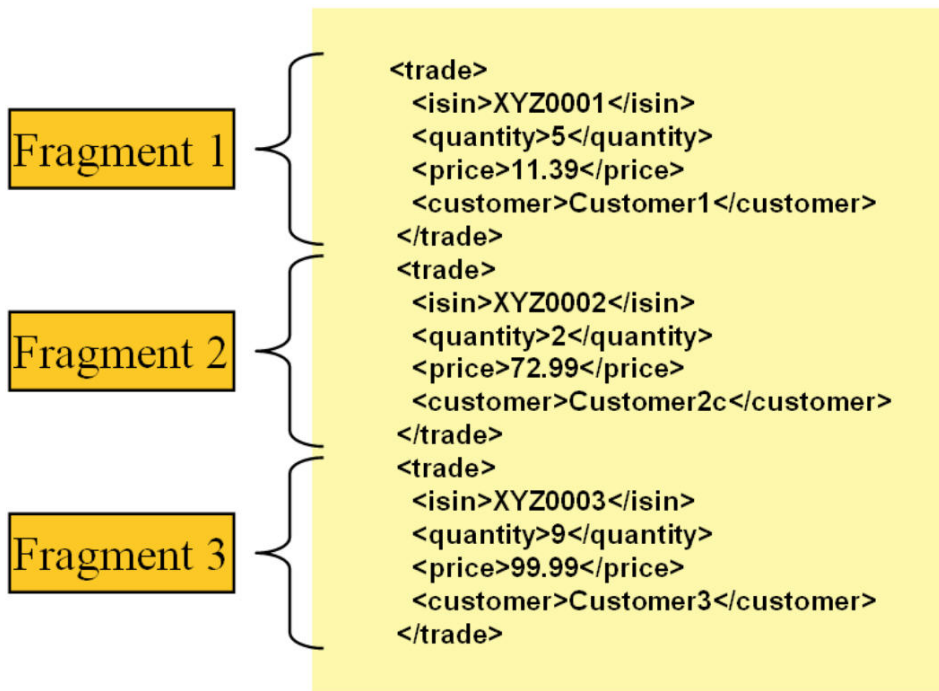
3.6. XML 아이템 reader와 Writer

스프링 배치는 자바 객체를 XML 레코드로 쓸 수 있을 뿐만 아니라 XML 레코드를 읽어서 자바 객체로 매핑하는 작업에 대해서 트랜잭션을 지원하는 인프라스트럭처를 제공한다.

스트리밍 XML에서의 제약

: StAX API는 배치 처리 요구사항에만 적합할 뿐만 아니라 다른 XML 파싱 API의 I/O로도 사용된다.(DOM은 한 번에 메모리로 모든 입력 데이터를 불러 오고, SAX는 사용자가 콜백을 제공해서 파싱 처리 과정을 제어한다.)

스프링 배치에서 XML 입력과 출력이 어떻게 작동되는지 더 자세히 살펴보자. 파일 읽기와 쓰기를 다양하게 해주는 몇 가지 개념이 있지만, 스프링 배치 XML 처리 과정에서는 처리 과정을 공통화 했다. XML 처리 과정에서 토큰라이징이 필요한 레코드의 행(FieldSet) 대신에, XML 자원을 개별 레코드에 대응되는 개념인 'fragments'의 컬렉션으로 가정하고 있다.



이 시나리오에서 'trade' 태그는 '루트 엘리먼트'로 정의 됐다. <trade>와 </trade> 사이에 모든 내용을 하나의 '프래그먼트(fragment)'로 취급한다. 스프링 배치는 프래그먼트를 객체로 바인드 하는데 Object/XML Mapping(OXM)을 사용한다. 그렇지만 스프링 배치는 특정 XML 바인딩 기술에 종속되지 않는다. 대표적인 사용방법은 가장 대중적인 OXM 기술에 대한 일관된 추상화를 제공하는 [Spring OXM](#)에 위임하는 방법이다.

Spring OXM에 대한 의존성은 선택이며, 필요하다면 스프링 배치에서 특정 인터페이스를 구현하도록 선택할 수 있다. OXM 지원과 관련된 기술에 대한 연관을 다음 그림을 통해서 살펴보자.

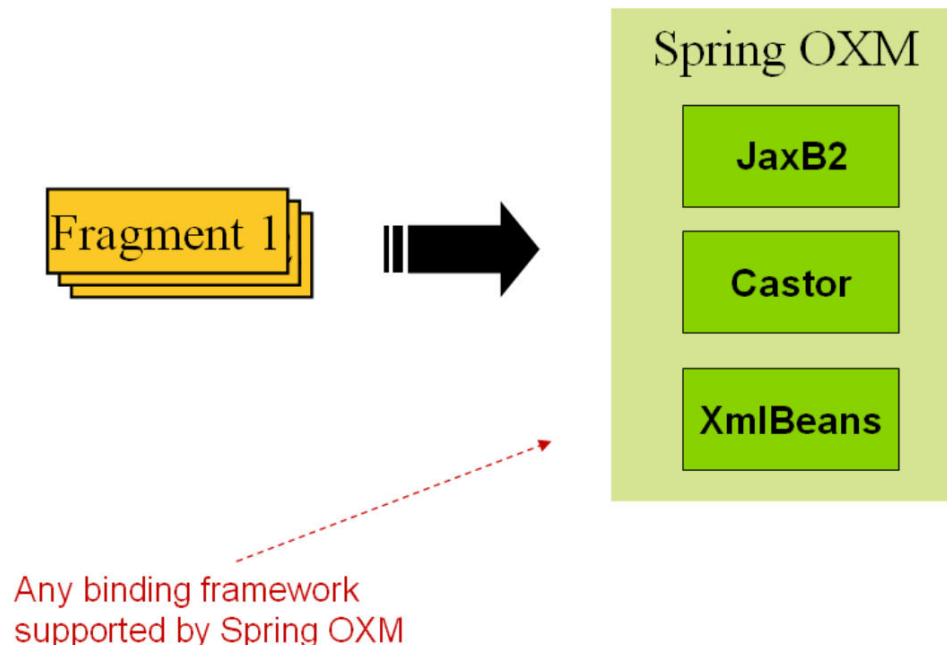


Figure 3.2: OXM Binding

3.6.1. StaxEventItemReader

StaxEventItemReader 설정은 XML 입력 스트림에서 레코드 처리 과정에 대한 전형적인 셋업을 제공한다. 먼저 StaxEventItemReader를 처리할 수 있는 XML 레코드의 집합을 검토해보자.

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
  <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
    <isin>XYZ0001</isin>
    <quantity>5</quantity>
    <price>11.39</price>
    <customer>Customer1</customer>
  </trade>
  <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
    <isin>XYZ0002</isin>
    <quantity>2</quantity>
    <price>72.99</price>
    <customer>Customer2c</customer>
  </trade>
  <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
    <isin>XYZ0003</isin>
    <quantity>9</quantity>
    <price>99.99</price>
    <customer>Customer3</customer>
  </trade>
</records>
```

XML 레코드 처리하려면 다음 사항이 필요하다.

- 루트 엘리먼트 이름: 매핑되는 객체를 구성하는 프래그먼트의 루트 엘리먼트 이름. 예제 환경 설정에서는 trade로 보면 된다.
- Resource: 읽어 들이는 파일을 나타내는 Spring Resource
- FragmentDeserializer: XML 프래그먼트를 객체로 매핑하는 스프링 OXM에 의해서 제공되는 언마샬링 기능

```
<property name="itemReader">
  <bean class="org.springframework.batch.io.xml.StaxEventItemReader">
    <property name="fragmentRootElementName" value="trade" />

    <property name="resource" value="data/staxJob/input/20070918.testStream.xmlFileStep.xml" />
  >

  <property name="fragmentDeserializer">
    <bean
      class="org.springframework.batch.io.xml.oxm.UnmarshallingEventReaderDeserializer">
      <constructor-arg>
        <bean class="org.springframework.oxm.xstream.XStreamMarshaller">
          <property name="aliases" ref="aliases" />
        </bean>
      </constructor-arg>
    </bean>
  </property>
</bean>
</property>
```

이 예제에서는 XStreamMarshaller를 사용한다. XStreamMarshaller에는 먼저 프래그먼트의 이름과 객체 타입을 바인드 해주기 위해 사용되는 별칭을 키와 값을 포함하는 맵으로 건내주도록 했다. 그 다음 FieldSet과 비슷하게 맵에 엘리먼트 이름과 타입이 키/값 쌍으로 들어가게 된다. 다음처럼 설정 파일에서 필요한 별칭을 기술하는데 스프링 설정 유틸리티를 사용할 수 있다.

```
<util:map id="aliases">
  <entry key="trade"
    value="org.springframework.batch.sample.domain.Trade" />
  <entry key="isin" value="java.lang.String" />
  <entry key="quantity" value="long" />
  <entry key="price" value="java.math.BigDecimal" />
  <entry key="customer" value="java.lang.String" />
```

```
</util:map>
```

입력 리더는 (기본적으로 태그 이름의 일치에 의해서) 새로운 프래그먼트가 시작하는 것을 인식할 때까지 XML 자원을 읽어 들인다. 리더는 프래그먼트에서 독립적으로 작동하는 XML 문서를 생성하고, XML을 자바 객체로 매핑하기 위해 `deserializer`에게 이 문서를 전달한다.

정리해보면,

```
StaxEventItemReader xmlStaxEventItemReader = new StaxEventItemReader()
Resource resource = new ByteArrayResource(xmlResource.getBytes())

Map aliases = new HashMap();
aliases.put("trade", "org.springframework.batch.sample.domain.Trade");
aliases.put("isin", "java.lang.String");
aliases.put("quantity", "long");
aliases.put("price", "java.math.BigDecimal");
aliases.put("customer", "java.lang.String");
Marshaller marshaller = new XStreamMarshaller();
marshaller.setAliases(aliases);
xmlStaxEventItemReader.setFragmentDeserializer(new
    UnmarshallingEventReaderDeserializer(marshaller));
xmlStaxEventItemReader.setResource(resource);
xmlStaxEventItemReader.setFragmentRootElementName("trade");
xmlStaxEventItemReader.open(new ExecutionContext());

boolean hasNext = true

while (hasNext) {
    trade = xmlStaxEventItemReader.read();
    if (trade == null) {
        hasNext = false;
    } else {
        println trade;
    }
}
```

3.6.2. StaxEventItemWriter

출력은 입력과 대칭적으로 작동한다. `StaxEventItemWriter`은 `Resource`, `serializer`, `rootTagName`을 필요로 한다. 자바 객체는 (일반적으로 스프링 OXM Marshaller를 래핑하는) `serializer`에게 전달되어 OXM 톨에 의해서 각 프래그먼트마다 생성되는 `StartDocument`와 `EndDocument` 이벤트를 걸러내고 나머지 커스텀 이벤트 `writer`를 사용해서 `Resource`를 작성하게 된다. 이번 예제에서는 `MarshallingEventWriterSerializer`를 사용한다.

스프링 설정을 살펴보면,

```
<bean class="org.springframework.batch.item.xml.StaxEventItemWriter" id="tradeStaxWriter">
    <property name="resource" value="file:target/test-
outputs/20070918.testStream.xmlFileStep.output.xml" />
    <property name="serializer" ref="tradeMarshallingSerializer" />
    <property name="rootTagName" value="trades" />
    <property name="overwriteOutput" value="true" />
</bean>
```

`serializer`가 의존성 참조를 하고 있는 `tradeMarshallingSerializer`는 다음처럼 설정하면 된다.

```
<bean class="org.springframework.batch.item.xml.oxm.MarshallingEventWriterSerializer"
id="tradeMarshallingSerializer">
    <constructor-arg>
        <bean class="org.springframework.oxm.xstream.XStreamMarshaller">
            <property name="aliases" ref="aliases" />
        </bean>
    </constructor-arg>
```

```
</bean>
```

역시 정리해보면, 다음처럼 프로그램적으로 필요한 프로퍼티를 준비해서 확인해볼 수 있다.

```
StaxEventItemWriter staxItemWriter = new StaxEventItemWriter()
FileSystemResource resource = new
    FileSystemResource(File.createTempFile("StaxEventWriterOutputSourceTests", "xml"))

Map aliases = new HashMap();
aliases.put("trade", "org.springframework.batch.sample.domain.Trade");
aliases.put("isin", "java.lang.String");
aliases.put("quantity", "long");
aliases.put("price", "java.math.BigDecimal");
aliases.put("customer", "java.lang.String");
XStreamMarshaller marshaller = new XStreamMarshaller()
marshaller.setAliases(aliases)

MarshallingEventWriterSerializer tradeMarshallingSerializer = new
    MarshallingEventWriterSerializer(marshaller);

staxItemWriter.setResource(resource);
staxItemWriter.setSerializer(tradeMarshallingSerializer);
staxItemWriter.setRootTagName("trades");
staxItemWriter.setOverwriteOutput(true);

ExecutionContext executionContext = new ExecutionContext();
staxItemWriter.open(executionContext);
Trade trade = new Trade();
trade.isin = "XYZ0001";
trade.quantity = 5;
trade.price = 11.39;
trade.customer = "Customer1";
println trade;
staxItemWriter.write(trade);
staxItemWriter.flush();
```

3.7 실시간에 파일 이름 만들기

```
//todo
```

3.8 다수의 파일 입력

한 Step에서 다수의 파일을 처리하는 건 매우 일반적이 요구사항이다. MultiResourceItemReader는 XML과 플랫 파일 처리과정에서 다양한 타입의 입력을 제공한다.

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

MultiResourceItemReader는 와일드카드로 다수의 파일을 읽어올 수 있다.

```
<bean id="multiResourceReader"
    class="org.springframework.batch.item.SortedMultiResourceItemReader">
    <property name="resources" value="classpath:data/multiResourceJob/input/file-*.txt" />
    <property name="delegate" ref="flatFileItemReader" />
</bean>
```

실제 읽기는 flatFileItemReader에게 위임한다. 배치 잡은 작업이 성공적으로 종료될 때까지 개별적으로 디렉토리를 유지하는게 좋다.

3.9. 데이터베이스

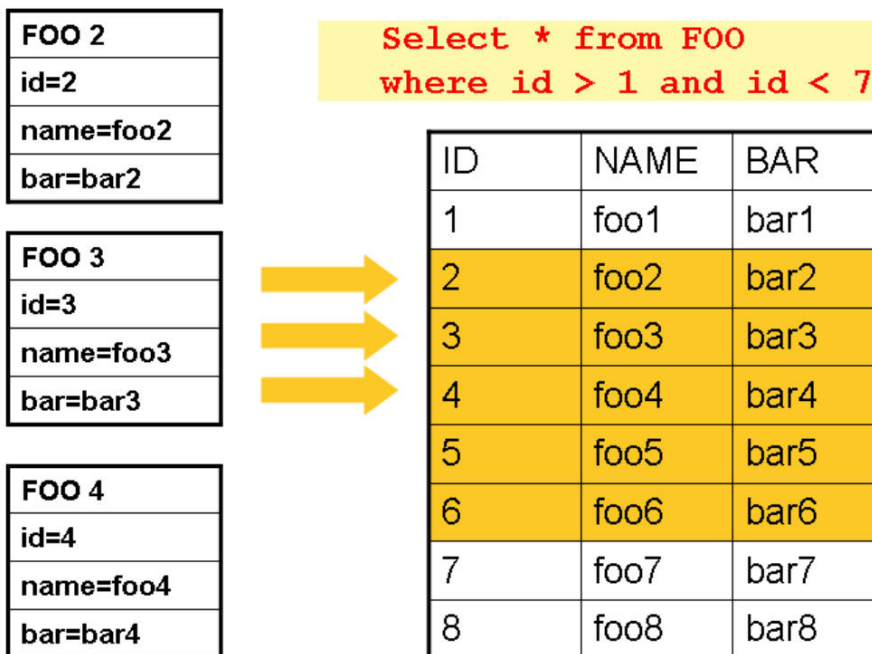
대부분 엔터프라이즈 애플리케이션 스타일처럼 데이터베이스는 배치 저장 메커니즘의 중심이 된다. 그렇지만 배치는 다른 애플리케이션 스타일과는 다르다. 스프링 코어 JdbcTemplate은 이러한 문제를 해결해준다.

RowMapper와 함께 JdbcTemplate를 사용하는 경우, RowMapper는 쿼리에서 반환되는 모든 결과(행)마다 호출되게 된다. 이는 작은 데이터 집합에서는 별 문제가 안되지만, 배치 처리처럼 매우 큰 규모의 데이터 집합을 처리 해야 하는 경우에는 JVM에 돌아버리는 원인이 되기도 한다. SQL 문이 백만 행을 반환하는 경우에, RowMapper는 백만번 호출되고, 모든 행이 읽혀질 때까지 메모리에 반환된 결과를 모두 유지하게 된다. 스프링 배치는 이 문제에 대해서 Cursor와 DrivingQuery ItemReader 두 가지 해결책을 제시한다.

3.9.1. 커서(Cursor) 기반 ItemReader

데이터베이스 커서 사용은 많은 배치 개발자들이 사용하는 기본적인 접근 방법이다. 커서는 'streaming' 관계형 데이터 문제의 해결책이 되기 때문이다. ResultSet 클래스는 커서를 조정하는 객체 지향 메커니즘의 필수적인 클래스다. ResultSet은 데이터의 현재 행에 대한 커서를 유지한다. ResultSet의 next를 호출하면 커서를 다음 행으로 이동한다. 스프링 배치 커서는 커서를 초기화 해서 열어 주는 ItemReader에 기반하며, read가 호출될 때마다 커서를 다음 행으로 이동시키며, 처리 과정 중에 사용되는 매핑된 객체를 반환한다. close 메소드는 모든 자원을 해제 됨을 보장해준다.

스프링 코어 JdbcTemplate은 ResultSet에서 모든 행을 완벽하게 매핑하고, 메소드 호출자에게 제어권을 반환하기 전에 자원을 종료하는데 콜백 패턴을 사용해서 문제를 극복한다. 그렇지만 배치에서는 step이 종료되기 전까지 대기한다. 아래 그림은 예제에서 사용되는 SQL문을 통해서 ItemReader에 기반을 둔 커서가 작동되는 방법을 보여주는 다이어그램이다.



이 예제는 기본 패턴을 보여준다. 'FOO' 테이블은 ID, NAME, BAR 세 개의 컬럼을 갖는다. 예제는 ID가 1보다 크고 7보다 작은 모든 행의 결과를 조회한다. 커서는 ID 2에서 시작한다. read()가 호출될 때마다 Foo 객체로 매핑되고, 커서는 다음 행으로 옮겨진다. 각 read 호출 후에 쓰여진 다음에는 가비지 컬렉션의 대상이 된다.(인스턴스 변수가 전혀 참조되지 않다고 가정한다면...)

3.9.1.1. JdbcCursorItemReader

JdbcCursorItemReader는 특정 기술 기반 커서 JDBC 구현체다. 이 클래스는 직접 ResultSet과 함께 작동되며, DataSource에서 얻은 커넥션으로 실행되는 SQL 문이 필요하다.

예제에서 사용한 데이터베이스 스키마를 살펴보면,

```
CREATE TABLE CUSTOMER (  
  ID BIGINT IDENTITY PRIMARY KEY,  
  NAME VARCHAR(45),  
  CREDIT FLOAT
```



```
);
```

많은 사람들은 각 행을 도메인 객체로 표현하는 것을 선호하므로, RowMapper 인터페이스를 구현해서 사용한다.

```
public class CustomerCreditRowMapper implements RowMapper {

    public static final String ID_COLUMN = "id";
    public static final String NAME_COLUMN = "name";
    public static final String CREDIT_COLUMN = "credit";

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        CustomerCredit customerCredit = new CustomerCredit();

        customerCredit.setId(rs.getInt(ID_COLUMN));
        customerCredit.setName(rs.getString(NAME_COLUMN));
        customerCredit.setCredit(rs.getBigDecimal(CREDIT_COLUMN));

        return customerCredit;
    }
}
```

JdbcTemplate은 스프링 사용자에게는 친숙하고, JdbcCursorItemReader는 키(key) 인터페이스를 공유하기 때문에, ItemReader와 대조하기 위해서 JdbcTemplate으로 데이터를 읽어들이는 방법을 예제에서 살펴보는게 매우 좋을 것 같다. 이런 목적을 가지고 CUSTOMER 데이터베이스에 1,000개의 행이 있다고 가정해보고, 먼저 JdbcTemplate을 사용하는 예제를 살펴보자.

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

이 코드를 실행한 다음에 customerCredits 리스트는 1,000개의 CustomerCredit 객체를 포함하게 된다. 쿼리 메소드에서 커넥션은 DataSource에서 얻게 되고, SQL이 실행되서, mapRow 메소드는 ResultSet에 각 행별로 호출된다.

```
JdbcCursorItemReader itemReader = new JdbcCursorItemReader();
itemReader.setDataSource(dataSource);
itemReader.setSql("SELECT ID, NAME, CREDIT from CUSTOMER");
itemReader.setMapper(new CustomerCreditRowMapper());
int counter = 0;
ExecutionContext executionContext = new ExecutionContext();

itemReader.open(executionContext);
Object customerCredit = new Object();
while(customerCredit != null){
    customerCredit = itemReader.read();
    counter++;
}
itemReader.close(executionContext);
```

이 코드가 실행된 후에도 동일하게 1,000개의 객체가 생성된다. 위 코드가 반환된 customerCredit을 list로 넣었다면, 결과는 JdbcTemplate을 사용한 예제와 결과와 동일하다. 그렇지만, ItemReader의 가장 큰 장점은 아이템이 '흐름 streamed' 을 갖도록 해준다는 점이다. read() 메소드는 일단 호출되면, ItemWriter를 통해서 쓰여지고, 다시 read()를 통해서 아이템을 받게 된다.

이는 높은 성능과 배치 처리에 필수적인 아이템 읽기와 쓰기가 '한 묶음 chunk'으로 처리되고, 순차적으로 커밋된다.

3.9.1.1. 추가적인 프로퍼티들

ignoreWarnings

SQL Warning을 로그로 남길지 말지를 결정 - 기본값은 true

fetchSize

ItemReader에서 사용되는 객체가 ResultSet에서 필요한 행보다 더 많을 때 데이터베이스에서 fetch되어야 하는 행의 개

maxRows	수를 JDBC 드라이버 힌트로 제공 - 기본값은 힌트를 주지 않는 것
queryTimeout	ResultSet에서 한 번에 유지할 수 있는 최대 행의 개수를 제한 Statement 객체가 응답을 줄 때까지 드라이버가 기다리는 초를 설정. 제한된 시간을 초과하면, DataAccessException을 발생
verifyCursorPosition	ItemReader에 의해서 유지되는 동일한 ResultSet 이 RowMapper로 전달되기 때문에, 사용자가 직접 ResultSet.next()를 호출할 수 있다. 이를 예방하려고 true로 설정하면, RowMapper를 호출하기 전과 후에 커서 위치가 같지 않은 경우에 예외를 던진다.
saveState	리더의 상태가 ItemStream.update(ExecutionContext)에 의해서 제공되는 ExecutionContext에 저장될지 말지를 정함

3.9.1.2. HibernateCursorItemReader

일반 스프링 사용자들은 ORM을 사용하나 마냐는 매우 중요한 결정이다. 이 결정에 따라 JdbcTemplate이나 HibernateTemplate을 사용하게 된다. 스프링 배치 사용자도 이와 동일한 선택을 하게 된다. HibernateCursorItemReader는 커서를 적용한 하이버네이트 구현이다. 배치에서 하이버네이트의 사용은 논쟁의 여지가 있다. 하이버네이트는 근본적으로 온라인 애플리케이션 스타일을 지원하지만, 그렇다고 배치 처리에서 사용할 수 없다는 건 아니다.

이 문제를 해결하는 가장 쉬운 방법은 표준 Session이 아니라 StatelessSession을 사용하는 것이다. 이 클래스에서는 배치 시나리오에서 하이버네이트를 사용하는데 이슈의 원인이 되는 캐시나 dirty checking을 제거했다. JdbcCursorItemReader와 동일하게 기본적인 방법으로 read를 호출할 때마다 하나의 아이템이 건네지도록, HibernateCursorItemReader는 HQL 문을 선언하고, SessionFactory를 전달할 수 있다. 아래 코드는 JDBC 리더와 동일한 'customer credit' 예제를 사용한 예제 설정이다.

```
HibernateCursorItemReader itemReader = new HibernateCursorItemReader();
itemReader.setQueryString("from CustomerCredit");
//For simplicity sake, assume sessionFactory already obtained.
itemReader.setSessionFactory(sessionFactory);
itemReader.setUseStatelessSession(true);
int counter = 0;
ExecutionContext executionContext = new ExecutionContext();
itemReader.open(executionContext);
Object customerCredit = new Object();
while(customerCredit != null){
    customerCredit = itemReader.read();
    counter++;
}
itemReader.close(executionContext);
```

설정된 ItemReader는 JdbcCursorItemReader와 정확하게 동일한 방법으로 CustomerCredit를 반환한다. 'useStatelessSession' 프로퍼티는 기본적으로 true지만, 이 기능을 켜다 끄다 할 수 있다는 능력을 보여주려고 추가했다. setFetchSize 프로퍼티를 통해서 커서 기반의 fetchSize를 설정하는 건 매우 중요하다.

3.9.2. 페이지 처리가 적용된 ItemReader

데이터베이스 커서를 사용하는 대신 여러번 쿼리를 실행할 수 있다. 실행되는 각 쿼리는 정해진 크기의 결과를 가져오게 된다. 여기서는 이 크기를 페이지로 부른다. 실행되는 각 쿼리는 시작하는 행의 수를 지정하고, 페이지 별로 반환하고자 하는 행의 수를 지정해야 한다.

3.9.2.1. JdbcPagingItemReader

페이지 처리를 적용한 ItemReader의 구현체 중 하나로 JdbcPagingItemReader가 있다. JdbcPagingItemReader는 페이지를 형성하는 행을 반환하는데 사용하는 SQL 쿼리를 제공할 책임을 지고 있는 PagingQueryProvider가 필요하다. 데이터베이스마다 페이지 처리를 지원하는 자체적인 전략을 가지고 있기 때문에, 데이터 베이스 유형 별로 지원하는 서로 다른 PagingQueryProvider를 사용해야 한다. 사용하고 있는 데이터베이스를 자동으로 식별해주고, 적절한 PagingQueryProvider 구현체를 적용해주는데 사용하는 SimpleDelegatingPagingQueryProvider도 있다. SimpleDelegatingPagingQueryProvider는 환경 설정을 간단히 해주며, 추천하는 베스트 프랙트스다.

SimpleDelegatingPagingQueryProvider는 select 절과 from 절을 지정해줘야 한다. where 절 제공은 선택이다. 이 세 절들은 필수인 sortKey와 조합해서 SQL 문을 구성하는데 사용된다.

리더가 열린 후에는 여타 다른 ItemReader처럼 동일한 기본 방식으로 read 호출 마다 하나의 항목을 되돌려 보낸다. 페이지 처리는 추가로 행이 필요할 때마다 보이지 않는 곳에서 일어난다.

아래는 커서 기반 ItemReader로 'customer credit' 예제와 비슷한 예제 환경 설정이다.

```
<bean id="itemReader"
      class="org.springframework.batch.item.database.JdbcPagingItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="queryProvider">
        <bean
            class="org.springframework.batch.item.database.support.SimpleDelegatingPagingQueryProvider">
            <property name="selectClause" value="select id, name, credit"/>
            <property name="fromClause" value="from customer"/>
            <property name="whereClause" value="where status=:status"/>
            <property name="sortKey" value="id"/>
        </bean>
    </property>
    <property name="parameterValues">
        <map>
            <entry key="status" value="NEW"/>
        </map>
    </property>
    <property name="pageSize" value="10"/>
    <property name="parameterizedRowMapper" ref="customerMapper"/>
</bean>
```

여기서 구성한 ItemReader는 필수로 지정해야 하는 ParameterizedRowMapper를 사용해 CustomerCredit 객체를 반환하게 된다. 'pageSize' 프로퍼티는 매번 쿼리 실행 시마다 데이터베이스에서 읽어오는 엔티티의 수를 결정해준다.

'parameterValues' 프로퍼티는 쿼리의 매개변수 값을 Map으로 지정하는데 사용한다. where 절에서 named 매개변수를 사용하는 경우, 각 entry의 key를 named 매개변수의 이름과 일치시키면 된다. 전통적인 '?' 개체들(placeholder)을 사용하면, 각 entry의 key로는 개체들의 순서를 지정하면 된다. 1부터 시작한다.

3.9.2.2. JpaPagingItemReader

페이지 처리가 적용된 ItemReader의 또 다른 구현체로 JpaPagingItemReader가 있다. JPA는 하이버네이트 StatelessSession과 비슷한 개념이 없으므로, JPA 명세에서 제공하는 다른 특징을 사용해야만 한다. JPA는 페이지 처리를 지원하기 때문에, 배치 처리에서 JPA를 사용함에 따라서 오는 자연스러운 선택이 된다. 각 페이지에서 읽어들이는 엔티티는 detached 상태가 되며, persistence context는 페이지가 처리되자마자 GC가 되기 위해서 제거된다.

JpaPagingItemReader에는 JPQL 문을 선언하고, EntityManagerFactory를 건네준다. 그렇게 되면 여타 다른 ItemReader처럼 동일한 기본 방식으로 read 호출마다 하나의 항목을 돌려보내게 된다. 페이지 처리는 추가로 엔티티가 필요할 때 보이지 않는 곳에서 실행된다.

아래는 커서 기반 ItemReader로 'customer credit' 예제와 비슷한 예제 환경 설정이다.

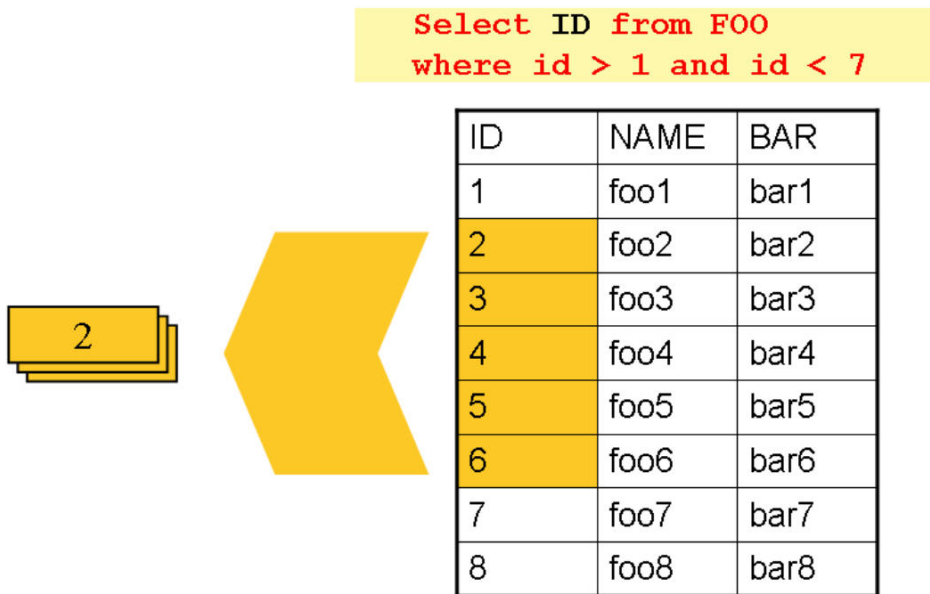
```
<bean id="itemReader"
      class="org.springframework.batch.item.database.JpaPagingItemReader">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
    <property name="queryString" value="select c from CustomerCredit c"/>
    <property name="pageSize" value="1000"/>
</bean>
```

Customer 객체는 정확하게 JPA 애노테이션이나 ORM 매핑 파일에 등록됐다는 가정하에, 구성된 ItemReader는 JdbcPagingItemReader에서 묘사한 것과 정확하게 동일한 방법으로 CustomerCredit 객체를 반환하게 된다. 'pageSize' 프로퍼티는 매번 쿼리 실행마다 데이터베이스에서 엔티티를 읽어들이는 수를 결정한다.

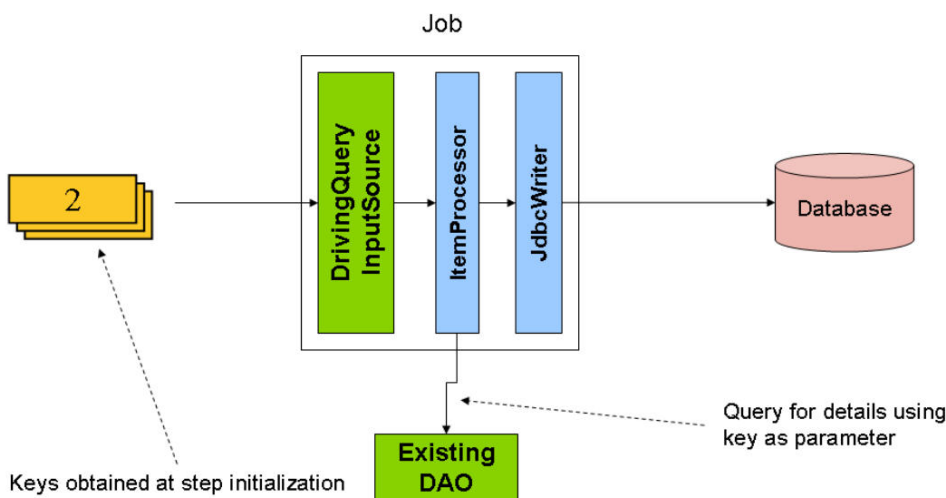
3.9.3. 쿼리 기반 ItemReader

많은 애플리케이션 벤더들은 매우 극단적인 pessimistic 락 전략을 가지고 있다. 이 전략은 또 다른 온라인 애플리케이션에서도 테이블의 읽기가 필요한 경우에 문제의 원인이 될 수 있다. 또한 극단적으로 양이 많은 데이터 집합에 대해서 커서를 여는건 특정 벤더에 따라서 이슈가 되기도 한다.

그러므로 많은 프로젝트는 데이터를 읽는데 'Driving Query' 접근 방법 사용을 선호한다. 이 접근 방법은 키를 통한 반복 접근으로 작동한다.



앞서 커서 기반 예제를 들때와 동일한 'FOO' 테이블이다. 그렇지만 전체 행을 선택하는 것이 아니라, SQL 문에서 ID만 선택하도록 한다. read 메소드에서 Foo 객체를 반환하는게 아니라, Integer를 반환한다. 이 값은 완전한 Foo 객체 받아오는데 사용한다.



이 그림에서 볼 수 있듯이, 기존 DAO는 driving 쿼리에서 받아온 키를 사용해서 완전한 'Foo' 객체를 얻어오는데 사용한다. 스프링 배치에서 driving 쿼리 스타일 입력은 DrivingQueryItemReader로 구현되었고, 이 클래스는 KeyCollector에 대해서만 의존성을 가지고 있다.

3.9.3.1. KeyCollector

앞서 예에서 볼 수 있듯이, DrivingQueryItemReader는 매우 간단한 구조를 가지고 있다. 간단히 키의 리스트를 받아서 반복잡업을 하게 된다.

그렇지만 실제로 복잡한 문제는 키를 얻어오는 방법이다. KeyCollector 인터페이스가 이 방법을 추상화했다.

```
public interface KeyCollector {

    List retrieveKeys(ExecutionContext executionContext);

    void updateContext(Object key, ExecutionContext executionContext);
}
```

```
}
```

이 인터페이스의 가장 주요한 메소드는 `retrieveKeys` 메소드다. 이 메소드는 해당 시나리오가 재시작 시나리오인지 아닌지에 관계 없이 처리되는 키를 반환하게 된다. 예를 들어, 잡이 1에서 1,000까지 키를 처리하도록 시작했고, 500번째를 처리하고 난 후 실패하게 되면, 재시작할 때는 500에서 1,000개의 키를 반환받게 된다. 이 기능은 `updateContext` 메소드에 의해서 가능해진다. `updateContext`는 `ExecutionContext`에 제공된 키(현재 처리된 키여야 함)를 저장한다. `retrieveKeys` 메소드는 최초 전체 키의 하위 집합을 반환하도록 사용할 수 있다.

```
ExecutionContext executionContext = new ExecutionContext();
List keys = keyStrategy.retrieveKeys(executionContext);
//Assume keys contains 1 through 1,000
keyStrategy.updateContext(new Long(500), executionContext);
keys = keyStrategy.retrieveKeys(executionContext);
//keys should now contains 500 through 1,000
```

이런 일반화는 `KeyCollector` 계약으로 표현되어 있다. 최초 초기화하려고 `retrieveKeys`를 호출하는 것이 귀찮다면, 500 키값으로 `updateContext`를 호출하면 동일한 `ExecutionContext`으로 500개의 키 값(501~1000)을 반환하도록 `retrieveKeys` 메소드를 호출해준다.

3.9.3.2. SingleColumnJdbcKeyCollector

가장 일반적인 driving 쿼리 시나리오는 키를 나타내는 컬럼이 하나만 있는 입력이 있는 경우다. 이 시나리오는 `SingleColumnJdbcKeyCollector` 클래스로 구현된다.

jdbcTemplate	데이터베이스에 쿼리를 실행하는데 사용하는 <code>JdbcTemplate</code>
sql	쿼리를 실행하는데 사용되는 sql 문. 하나의 값만 반환해야 한다.
restartSql	재시작 하는 경우에 사용하는 sql 문. 하나의 키만 사용하기 때문에, 이 쿼리는 단지 하나의 인자만을 필요로 한다.
keyMapper	키를 객체로 매핑하는데 사용하는 <code>RowMapper</code> 구현체. 기본적으로 스프링 코어 <code>SingleColumnRowMapper</code> 를 사용한다.

예를 살펴보자.

```
SingleColumnJdbcKeyCollector keyCollector = new SingleColumnJdbcKeyCollector(getJdbcTemplate(),
    "SELECT ID from T_FOOS order by ID");

keyCollector.setRestartSql("SELECT ID from T_FOOS where ID > ? order by ID");

ExecutionContext executionContext = new ExecutionContext();

List keys = keyStrategy.retrieveKeys(new ExecutionContext());

for (int i = 0; i < keys.size(); i++) {
    System.out.println(keys.get(i));
}
```

이 코드가 정상적으로 설정된 환경에서 실행되면, 다음과 같은 결과를 리턴하게 된다.

```
1
2
3
4
5
```

이제 재시작된 후에 다시 실행되는 코드를 보여주도록 예제 코드를 약간 수정해보자. 3번째 키까지 성공적으로 처리하고 실패한 경우의 예다.

```
SingleColumnJdbcKeyCollector keyCollector = new SingleColumnJdbcKeyCollector(getJdbcTemplate(),
    "SELECT ID from T_FOOS order by ID");
```

```
keyCollector.setRestartSql("SELECT ID from T_FOOS where ID > ? order by ID");

ExecutionContext executionContext = new ExecutionContext();

keyStrategy.updateContext(new Long(3), executionContext);

List keys = keyStrategy.retrieveKeys(executionContext);

for (int i = 0; i < keys.size(); i++) {
    System.out.println(keys.get(i));
}
```

이 코드를 실행하면 다음과 같은 결과를 받게 된다.

```
4
5
```

결과가 다르게 나온 이유는 다음 코드 때문이다.

```
keyStrategy.updateContext(new Long(3), executionContext);
```

이 코드는 키 콜렉터에게 세 번째 키까지 ExecutionContext에 제공됐다는 것을 말해주고 있다. 이 코드는 일반적으로 DrivingQueryItemReader에 의해서 자동으로 호출되지만, 간단히 직접 호출할 수도 있다. 3을 포함하도록 갱신된 ExecutionContext로 retrieveKeys를 호출하면 restartSql의 인자로 3이 전달되게 된다.

```
keyCollector.setRestartSql("SELECT ID from T_FOOS where ID > ? order by ID");
```

이렇게 3보다 큰 ID 값인 4, 5만 결과로 반환되게 된다.

3.9.3.3. 다수의 키 컬럼 매핑하기

SingleColumnJdbcKeyCollector는 자동 생성되는 키를 사용할 때나 유일한 식별자가 하나인 경우에 매우 유용하다. 식별자 컬럼이 여러 개 일 경우는 그렇게 많지 않지만 충분히 일어날 수 있다. 이런 경우에는 MultipleColumnJdbcKeyCollector를 사용하면 된다. 단순함은 조금 희생되지만 다수의 컬럼을 매핑할 수 있다. 다수의 컬럼 콜렉터에서 필요한 프로퍼티는 단일 컬럼일 때와 한 가지만 빼고 같다. 한 가지 다른점은 RowMapper 대신에 ExecutionContextRowMapper를 제공해야 한다는 점이다. 재시작 SQL 역시 필요하지만, 단일 컬럼일 때와 다른 점은 인자가 하나 이상 필요하기 때문에, execution context에서 키를 매핑하는 방법에서 좀더 복잡한 제어 방법이 필요하다.

```
public interface ExecutionContextRowMapper extends RowMapper {
    public void mapKeys(Object key, ExecutionContext executionContext);
    public PreparedStatementSetter createSetter(ExecutionContext executionContext);
}
```

ExecutionContextRowMapper 인터페이스는 표준 RowMapper 인터페이스는 상속해서 ExecutionContext에 다수의 키를 저장하도록 해준다. 그리고 재시작 SQL 문에서 인자를 설정하기 위해 PreparedStatementSetter를 생성한다.

기본적으로 ExecutionContextRowMapper는 Map을 사용해서 구현했다. 이 구현을 재정의 하지 않기를 바란다. 그렇지 만 키의 특정 타입을 반환해야 하는 경우에는 새로운 구현체를 제공할 수 있다.

3.9.3.4. iBatisKeyCollector

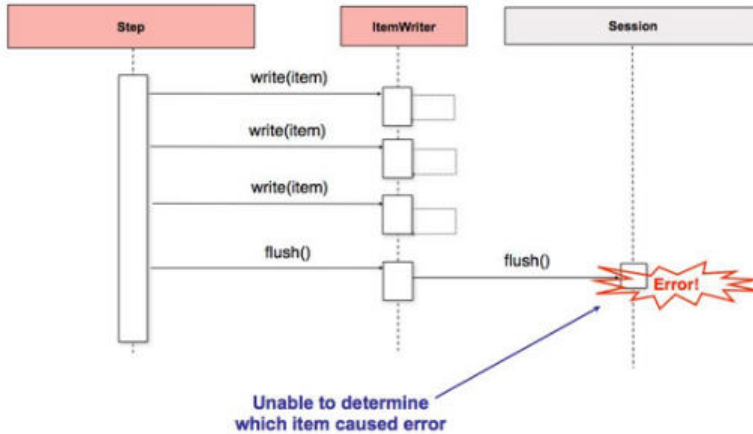
Jdbc 뿐만 아니라 iBatis에서도 사용할 수 있다. iBatis를 사용한다고 해서 KeyCollector의 기본적인 요구사항이 바뀌는 건 아니다. 그렇지만 iBatis를 사용하기 때문에, 쿼리 아이디와 SqlMapClient가 필요하다.

3.9.4. 데이터베이스 ItemWriter

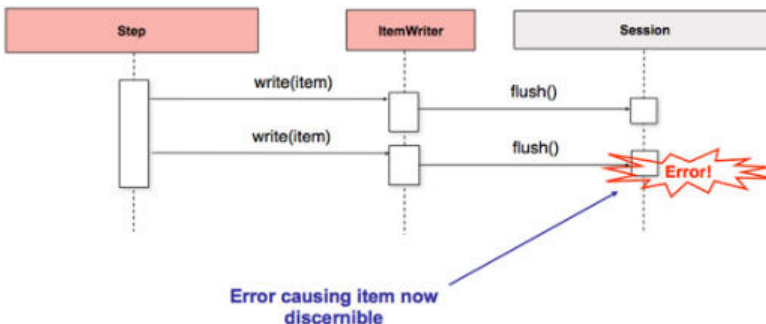
플랫 파일과 XML 둘 다 ItemWriter를 갖고 있지만, 이 둘은 완전히 동일하지 않다. ItemWriter에는 추가적으로 트랜잭션이 필요하다. ItemWriter는 적절한 시점에서 작성된 아이템을 추적하고, flush하거나 제거하는 작업을 트랜잭션으로써

처리해야 할 필요가 있다. 그렇지만 데이터베이스는 기본적으로 트랜잭션 지원을 포함하고 있기 때문에 별도로 기능을 추가할 필요가 없다. 사용자는 ItemWriter 인터페이스를 구현해서 DAO를 만들거나 일반적인 처리 과정 관점에서 작성된 커스텀 ItemWriter 중 하나를 사용하면 된다.

한 가지 예외는 버퍼 출력이다. ItemWriter로 하이버네이트를 사용할 때가 가장 일반적인 경우지만, Jdbc 배치 모드를 사용할 때도 같은 이유를 가지고 있게 된다. 버퍼링해둔 데이터베이스 출력이 본래부터 결함이 없다면, 데이터에는 에러가 없다고 가정한다. 그렇지만 그림에서처럼 쓰기 작업중에 발생한 모든 에러는 예외가 발생한 원인이 어느 아이템인지 알 길이 없기 때문에 문제의 원인이 될 수 있다.



쓰기 전에 버퍼된 아이템이라면, 발생한 모든 에러는 커밋 전에 버퍼가 flush될 때까지 예외를 던지지 않는다. 예를 들어, 한 묶음으로 20 개의 아이템을 쓴다고 가정했을 때, 15번째 아이템에서 DataIntegrityViolationException를 던졌다고 해보자. 실제로 쓰기 작업이 수행되기 전까지는 에러가 발생했는지 알 수 없기 때문에, 20개의 아이템이 모두 성공적으로 쓰기 작업(write) 메소드 호출을 마치게 된다. 일단 ItemWriter.flush()가 호출되면, 버퍼는 비워지면서 예외가 발견되게 된다. 이 시점에서 Step은 아무것도 하면 안되고, 트랜잭션은 반드시 롤백되어야 한다. 일반적으로 이 예외는 Item을 지나치게 될 수도 있는 원인이 되며, 다시 쓰기 작업을 하지 못한다. 그렇지만, 이 시나리오에서 버퍼가 한 번에 쓰기 작업을 하기 때문에, 이슈의 원인이 무엇인지를 알 수 있는 방법이 없다. 이 문제를 해결하는 유일한 방법은 각 아이템별로 flush를 하는 것이다.



이런 상황이 충분히 일반적이기 때문에, 특히 하이버네이트를 사용할 때는 스프링 배치가 이 문제를 도와주는 HibernateAwareItemWriter 구현을 제공한다. HibernateAwareItemWriter는 매우 직관적인 방법으로 이 문제를 해결한다. 첫 번째로 한 처리 단위가 실패하면, 차후에는 매번 각 아이템 별로 flush를 하게 된다. 이는 처리 묶음(chunk)의 길이 중 하나인 commit interval을 효율적으로 낮춰준다. 다음 예제는 HibernateAwareItemWriter를 구성하는 방법을 보여준다.

```

<bean id="hibernateItemWriter"
    class="org.springframework.batch.item.database.HibernateAwareItemWriter">
    <property name="sessionFactory" ref="sessionFactory" />
    <property name="delegate" ref="customerCreditWriter" />
</bean>

<bean id="customerCreditWriter"
    class="org.springframework.batch.sample.dao.HibernateCreditDao">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
  
```

3.10. 기존 서비스 재사용

배치 시스템은 종종 다른 애플리케이션 스타일과 함께 사용되기도 한다. 가장 일반적인 경우는 온라인 시스템이며, 이 뿐만 아니라 사용하는 각 애플리케이션 스타일에서 벌크 데이터 처리가 필요하는 thick 클라이언트 애플리케이션도 지원한다.

이런 상황에서 일반적으로 많은 클라이언트는 기존 DAO나 다른 서비스를 배치 잡에서 사용하길 원한다. 스프링은 DI를 활용해서 필요한 클래스를 주입하기에 최적화 되어 있지만, 배치의 경우에 이런 기존 서비스를 `ItemReader` 나 `ItemWriter`로 사용해야 한다는 점이다. 스프링 배치는 이에 대한 래핑 클래스 구현인 `ItemReaderAdapter` 와 `ItemWriterAdapter` 를 제공한다. 이 두 클래스는 표준 스프링 메소드 호출 Delegator 패턴을 구현한 클래스로, 간단하게 설정할 수 있다.

```
<bean id="itemReader" class="org.springframework.batch.item.adapter.ItemReaderAdapter">
  <property name="targetObject" ref="fooService" />
  <property name="targetMethod" value="generateFoo" />
</bean>

<bean id="fooService" class="org.springframework.batch.item.sample.FooService" />
```

한 가지 중요한 포인트는 `targetMethod`의 계약은 반드시 `read()` 메소드의 계약과 동일해야 한다는 점이다.

```
<bean id="itemWriter" class="org.springframework.batch.item.adapter.ItemWriterAdapter">
  <property name="targetObject" ref="fooService" />
  <property name="targetMethod" value="processFoo" />
</bean>

<bean id="fooService" class="org.springframework.batch.item.sample.FooService" />
```

3.11. 아이템 변환하기

대상 아이템을 쓰기 전에 비즈니스 로직을 넣고 싶을 때가 있지 않을까? 읽기와 쓰기 작업할 때 한 가지 옵션으로 `ItemReader`가 또 다른 `ItemReader`를 포함하고, `ItemWriter`도 또 다른 `ItemWriter`를 포함하는 `composite` 패턴을 사용할 수 있다.

예를 들어,

```
public class CompositeItemWriter implements ItemWriter {

    ItemWriter itemWriter;

    public CompositeItemWriter(ItemWriter itemWriter) {
        this.itemWriter = itemWriter;
    }

    public void write(Object item) throws Exception {

        //Add business logic here

        itemWriter.write(item);
    }

    public void clear() throws ClearFailedException {
        itemWriter.clear();
    }

    public void flush() throws FlushFailedException {
        itemWriter.flush();
    }
}
```

위 클래스는 제공된 일부 비즈니스 로직을 실행한 후에 쓰기 작업을 또 다른 `ItemWriter`에게 위임한다. 위임하는 `ItemWriter` 뿐만 아니라 `clear`와 `flush`도 정의해야 한다. 이 패턴은 `ItemReader`에 대해서 쉽게 사용할 수 있을 뿐 아니라,

메인 ItemReader(위의 예에서 CompositeItemWriter)에 의해서 제공되는 입력 값에 기반해서 더 많은 참조 데이터를 얻어올 수도 있다. 이 패턴은 우리 자체적으로 write() 메소드 호출을 제어할 필요가 있는 경우에 매우 유용하다. 그렇지만 실제 쓰기 작업을 하기 전에 쓰는 대상이 되는 아이템을 전달하기 전에 아이템을 '변환'만 하고 싶은 경우에는, write()를 우리가 직접 호출할 필요 없이 간단히 아이템을 수정하고만 싶을 것이다.

이 시나리오에 맞게 스프링 배치는 ItemTransformer 인터페이스를 제공한다.

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

ItemTransformer는 매우 간단한 인터페이스로, 객체를 받아서, 변환한 다음, 변환한 또 다른 객체를 반환한다. 제공되는 객체 타입은 반환할 객체와 같은 타입일 수도 있고, 다른 타입일 수도 있다. ItemTransformer는 ItemTransformerWriter의 일부로 사용되며, 아이템을 쓰기 전에, 먼저 아이템을 변환한다. 예를 들어, ItemReader가 Foo 타입의 클래스를 제공하고, 쓰기 전에 Bar 타입으로 변환될 필요가 있다고 가정해보자. ItemTransformer는 변환을 수행하도록 작성될 수 있다.

```
public class Foo {}  
  
public class Bar {  
    public Bar(Foo foo) {}  
}  
  
public class FooTransformer implements ItemProcessor{  
  
    //Perform simple transformation, convert a Foo to a Bar  
    public Object transform(Object item) throws Exception {  
        assertTrue(item instanceof Foo);  
        Foo foo = (Foo)item;  
        return new Bar(foo);  
    }  
}  
  
public class BarWriter implements ItemWriter{  
  
    public void write(Object item) throws Exception {  
        assertTrue(item instanceof Bar);  
    }  
  
    //rest of class omitted for clarity  
}
```

변환은 간단하지만, 모든 타입을 다 변환할 수 있다. BarWriter는 'Bar' 타입을 쓰는데 사용되며, 다른 타입이 들어오면 예외를 던진다. 이와 비슷하게 FooTransformer는 Foo가 아닌 다른 타입이 들어오면 예외를 던진다. ItemTransformerWriter는 일반적인 ItemWriter로 사용될 수 있다. ItemTransformerWriter에 Foo 객체를 넘기면, 이 객체가 변환되어, 실제로는 Bar가 건네진다. 결과적으로는 Bar가 쓰여지게 된다.

```
ItemTransformerItemWriter itemTransformerItemWriter = new ItemTransformerItemWriter();  
itemTransformerItemWriter.setItemTransformer(new FooTransformer());  
itemTransformerItemWriter.setDelegate(new BarWriter());  
itemTransformerItemWriter.write(new Foo());
```

3.11.1 Delegate 패턴과 Step과 함께 등록하기

//TODO

3.11.2. 연쇄적으로 실행되는 ItemTransformer

단일 변환 수행이 많은 시나리오에서 유용하지만, 다수의 ItemTransformer를 함께 '연쇄적으로' 실행하고 싶을 수 있다. CompositeItemTransformer를 사용해서 이런 목적을 달성할 수 있다. 이전 예제를 개선해서 보면, 변환이 한 번 발생해서, Foo가 Bar로 변환되고, 다시 FooBar로 변환되어 쓰여진다.

```
public class Foo {}
```

```

public class Bar {
    public Bar(Foo foo) {}
}

public class Foobar{
    public Foobar(Bar bar){}
}

public class FooTransformer implements ItemTransformer{

    //Perform simple transformation, convert a Foo to a Barr
    public Object transform(Object item) throws Exception {
        assertTrue(item instanceof Foo);
        Foo foo = (Foo)item;
        return new Bar(foo);
    }
}

public class BarTransformer implements ItemTransformer{

    public Object transform(Object item) throws Exception {
        assertTrue(item instanceof Bar);
        return new Foobar((Bar)item);
    }
}

public class FoobarWriter implements ItemWriter{

    public void write(Object item) throws Exception {
        assertTrue(item instanceof Foobar);
    }

    //rest of class omitted for clarity
}

```

FooTransformer와 BarTransformer는 결과적으로 Foobar를 주도록 연속해서 실행된다.

```

CompositeItemTransformer compositeTransformer = new CompositeItemTransformer();
List itemTransformers = new ArrayList();
itemTransformers.add(new FooTransformer());
itemTransformers.add(new BarTransformer());
compositeTransformer.setItemTransformers(itemTransformers);

```

compositeTransformer는 Foo 타입을 받아서, Foobar 타입을 반환한다. 복합 변환의 고객은 실제로 개별 변환이 어떻게 이뤄지는지 전혀 알 필요가 없다.

이 내용을 반영해서 예제를 다시 수정해보면,

```

ItemTransformerItemWriter itemTransformerItemWriter = new ItemTransformerItemWriter();
itemTransformerItemWriter.setItemTransformer(compositeTransformer);
itemTransformerItemWriter.setDelegate(new FoobarWriter());
itemTransformerItemWriter.write(new Foo());

```

3.12. 입력 유효성 검증

//todo

3.13. 상태 영속화 예방하기

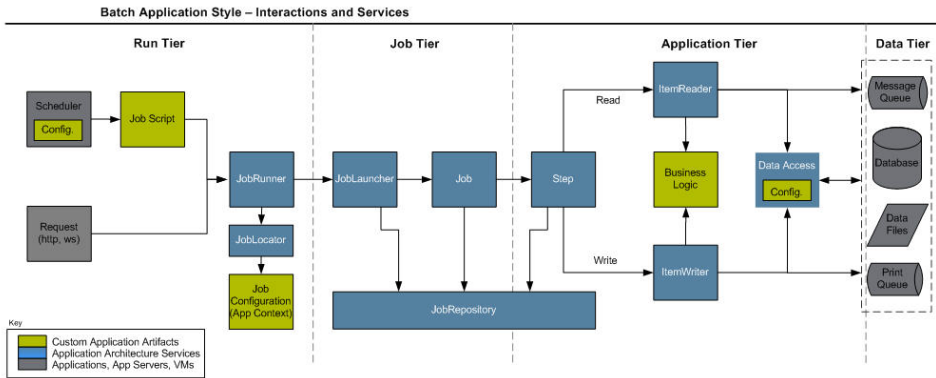
//todo

3.14. 커스텀 ItemReaders와 ItemWriters 만들기

//todo

4장 Job 구성하고 실행하기

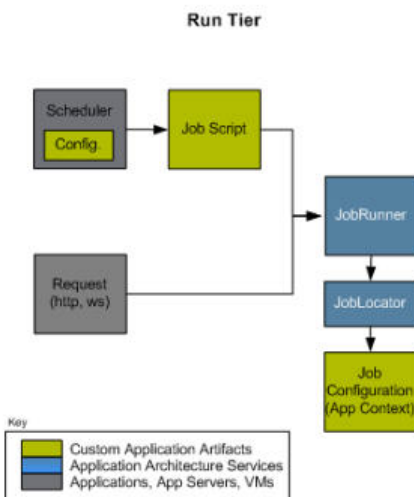
4.1. 소개



왼쪽에서 오른쪽으로 살펴보면, 다이어그램은 배치 잡 실행의 기본 흐름을 설명하고 있다.

1. 제일 처음으로 Scheduler 가 잡 스크립트를 시작시킨다!
2. 스크립트는 적절한 클래스패스를 구성하고, 자바 프로세스를 시작시킨다. 대부분 경우 진입 지점으로 CommandLineJobRunner 를 사용한다.
3. JobRunner 는 JobLocator 를 사용해서 Job 을 찾아내고, JobParameter 와 함께 Job 을 시작한다(launch).
4. JobLauncher 는 JobRepository 에서 JobExecution 을 받아오고, Job 을 실행한다(execute).
5. Job 은 순서에 따라 Step 을 실행한다.
6. Step 은 ItemReader 로 읽기 작업을 호출하고, 반환되는 값이 null이 될 때까지 ItemWriter 로 아이템의 결과값을 제어해서, 주기적으로 JobRepository 에 상태를 커밋하거나 저장한다.

4.2. Run 티어



이름에서 알 수 있듯이, Run 티어는 전체적인 관점에서 실질적인 잡의 운영과 관련이 있다. 그 시작 발생지가 Scheduler 나 HTTP 요청이냐에 관계 없이, Job을 반드시 얻어야 하고, 파라미터는 파싱되어야 하며, 결국에는 JobLauncher가 호출되게 된다.

4.2.1. 커맨드 라인에서 Job 실행하기

엔터프라이즈 스케줄러에 의해서 잡을 실행하고자 하는 사용자에게 커맨드 라인은 매우 중요한 인터페이스가 된다. 대부분의 스케줄러가 OS 프로세스로 바로 작동되기 때문이다.(셸 스크립트처럼) 자바 프로세스를 시작할 수 있는 방법은 셸 스크립트, 펄, 루비, '빌드 툴'인 앤트나 메이븐처럼 그 종류가 매우 많다. (레퍼런스에서는 셸 스크립트를 사용해서 예제를 진행한다.)

4.2.1.1. CommandLineJobRunner

잡을 시작시키는 스크립트는 반드시 JVM을 띄워야 때문에, 진입 지점으로 사용되는 메인 메소드가 필요하다. 스프링 배치는 이러한 목적으로 CommandLineJobRunner를 제공한다. 이 방법은 애플리케이션의 부트스트랩의 한 가지 방법이라는 점이 중요하긴 하지만, 이외에도 자바 프로세스를 시작시키는 방법이 많고, 이 클래스는 결정적으로 눈으로 확인할 수 있는 방법이 전혀 없다는 것이 문제다.

CommandLineJobRunner 클래스는 총 네 가지 작업을 수행한다.

- 적절한 Application Context 불러오기
- 커맨드 라인 인자를 JobParameters로 변환
- 여러 인자를 기반으로 적절한 잡 정하기
- 잡을 시작하기 위해서 application context에서 제공되는 JobLauncher 사용

이 모든 작업은 건네지는 인자에 따라서 완료하게 된다. 필수 인자 값은 다음 두 가지다.

jobPath	ApplicationContext를 생성하는데 사용하는 XML 파일의 경로. 이 파일에 Job 실행을 완료하는데 필요한 모든 정보를 포함해야 한다.
jobName	실행되는 잡 이름

반드시 경로가 제일 먼저, 그 다음 이름 순으로 인자를 건네야 한다. 다른 JobParameters에서 사용할 인자는 '이름-값' 쌍으로 뒤에 추가하면 된다.

```
bash$ java CommandLineJobRunner endOfDayJob.xml endOfDay schedule.date(date)=2008/01/01
```

'endOfDayJob.xml' 이 application context의 경로가 되고, endOfDay가 잡 이름이 되겠다.(2장에서 살펴본 예제) 그리고 마지막에 있는 schedule.date(date)=2008/01/01 은 JobParameters로 변환 된다.

```
<bean id="endOfDay"
    class="org.springframework.batch.core.job.SimpleJob">
    <property name="steps">
        <bean id="step1" parent="simpleStep" />

        <!-- Step details removed for clarity -->
    </property>
</bean>

<!-- Launcher details removed for clarity -->
<bean id="jobLauncher"

    class="org.springframework.batch.core.launch.support.SimpleJobLauncher" />
```

이 코드는 매우 단순하지만, 스프링 배치에서 기본적으로 요구하는 Job과 JobLauncher를 포함하고 있다.

4.2.1.2. ExitCodes

커맨드 라인에서 실행되는 대부분의 배치 잡은 엔터프라이즈 스케줄러에 의해서 시작되게 된다. 대부분 스케줄러는 단독으로 데이터를 처리할 수 없고, OS 프로세스 수준으로만 작동하게 된다. 즉, 스케줄러는 호출하는 셸 스크립트와 같은 OS 프로세스에 대한 정보만 알고 있다.

이번 시나리오에서 스케줄러는 단지 반환되는 코드 값에 따라서 잡의 성공 여부를 판단할 수 밖에 없다. 이렇게 반환되는 코드로 성공 여부를 판단할 수 있는데, 가장 쉽게 하자면 0이 성공, 1이 실패 이렇게 이해하면 된다. 그러나 좀더 상세히 정해서 잡 B가 시작되면 4를 반환하고, 잡 C가 시작되면 5를 반환하게 할수도 있다. 이는 스케줄러 레벨에서 걱정이야 할 문제지만, 스프링 배치와 같은 프레임워크가 'Exit Code'를 상수로 표현하는 방법을 제공하는 것도 매우 중요하다.

스프링 배치에서는 이를 `ExitStatus`로 캡슐화해서 했다.(자세한 내용은 5장 참조) 여기서 이 얘기를 하는 이유는 `ExitStatus`가 프레임워크(또는 개발자)에 의해서 설정되는 프로퍼티를 갖고 있고, `JobLauncher`에 의해서 반환되는 `JobExecution`의 일부로서 반환된다는 점을 강조하고 싶기 때문이다.

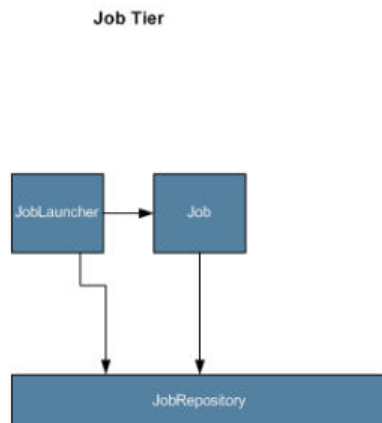
`CommandLineJobRunner`는 `ExitCodeMapper` 인터페이스를 사용해서 문자열 값을 숫자로 변환해준다.

```
public interface ExitCodeMapper {  
  
    public int intValue(String exitCode);  
  
}
```

이 인터페이스의 기본 구현으로 `SimpleJvmExitCodeMapper`가 있다. 이 매퍼는 완료는 0, 일반적인 에러는 1, `Job`을 찾을 수 없는 등의 잡 실행중 에러는 2를 반환한다. 좀 더 복잡한 상황이 필요할 때는 `ExitCodeMapper` 인터페이스를 확장하면 된다.

`ExitCodeMapper`가 `BeanFactory`에서 찾아지게 되려면, `context`가 생성된 후에 `runner`로 주입되어야 하기 때문에, `ExitCodeMapper`를 구현하는 모든 클래스들은 루트 레벨 빈으로 선언해서, `runner`에 의해서 생성되는 `ApplicationContext`의 일부로 포함되도록 해야 한다.

4.3. Job 티어



`Job` 티어는 배치 잡의 전반적인 실행을 책임진다. `Job` 티어는 배치 `step`을 순서대로 실행하고, 모든 `step` 정확한 상태를 갖고, 모든 정책이 적절하게 실시 됐는지를 보장한다.

`Job` 티어는 세 가지 잡 스테레오 타입인 `Job`, `JobInstance`, `JobExecution`를 유지하는 책임을 가지고 있다.

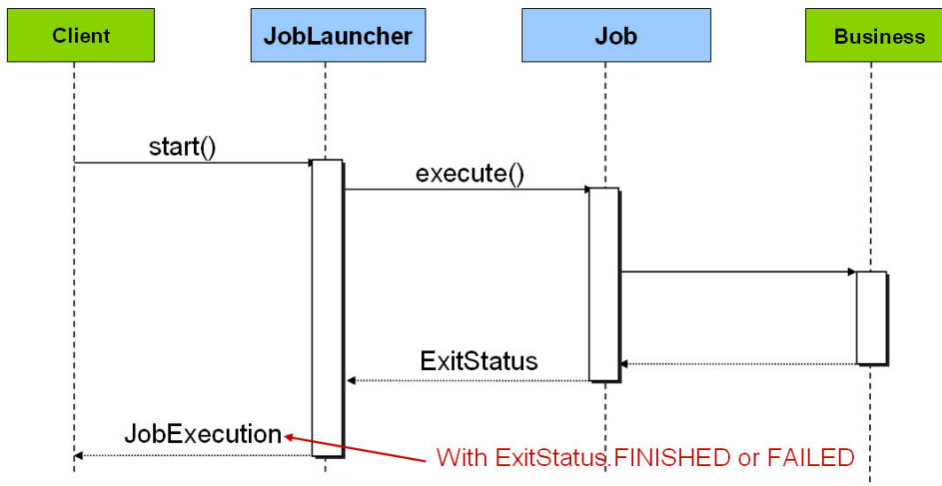
`JobLauncher`는 `JobExecution`을 생성하기 위해서 `JobRepository`와 상호작용을 한다. `Job`은 저장소를 사용해서 `JobExecution`을 저장한다.

4.3.1. SimpleJobLauncher

`SimpleJobLauncher`는 `JobLauncher` 인터페이스의 가장 기본적인 구현이다. 단지 `JobRepository`에 대한 의존성만 필요하다.

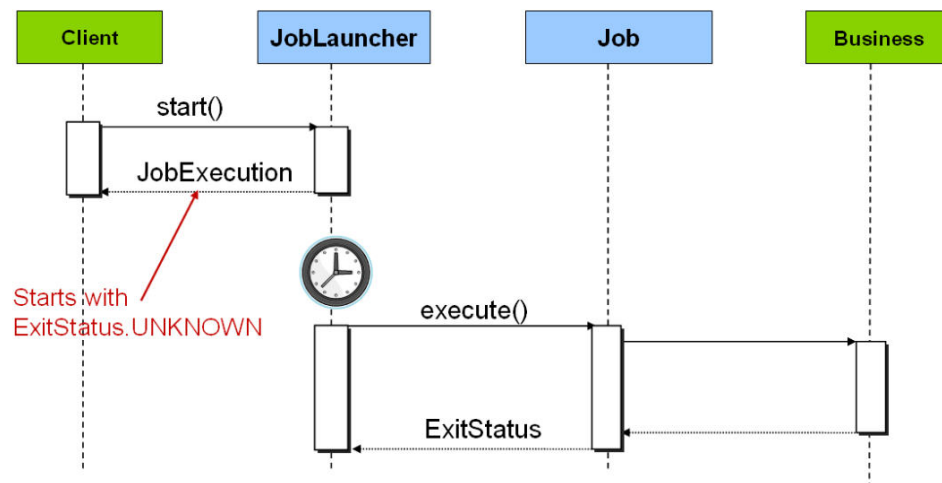
```
<bean id="jobLauncher"  
      class="org.springframework.batch.execution.launch.SimpleJobLauncher">  
    <property name="jobRepository" ref="jobRepository" />  
</bean>
```

일단 `JobExecution`을 얻었다면, `Job`의 `execute()` 메소드로 건네지고, 궁극적으로 호출자에게 `JobExecution`을 반환해준다.



이 흐름은 매우 직관적이다. 스케줄러에 의해서 시작될 때는 매우 잘 작동하지만, HTTP 요청으로 시작을 시도할 때는 이슈의 원인이 된다.

SimpleJobLauncher가 호출자에게 즉각 결과를 반환하려면, 잡 시작을 비동기화 할 필요가 있다. 배치처럼 오랫동안 실행되는 처리과정에서는 필요한 대로 시간동안 HTTP 요청을 계속 유지하는 건 좋은 방법이 아니다. HTTP 요청의 경우 아래와 같은 처리 흐름을 거치게 된다.



TaskExecutor 구성해서 이 시나리오를 가능하게 해주는 SimpleJobLauncher를 쉽게 구성할 수 있다.

```

<bean id="jobLauncher"
    class="org.springframework.batch.execution.launch.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository" />
    <property name="taskExecutor">
        <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
    </property>
</bean>

```

스프링 TaskExecutor 인터페이스의 모든 구현은 비동기화된 실행하는 방법에 대한 제어의 목적으로 사용된다.

4.3.1.1. Job 정지하기

잡을 비동기로 시작하고자 하는 가장 공통적인 이유 중 하나는 잡을 적절하게 종료할 수 있도록 하기 위함이다. 이는 JobLauncher에서 반환되는 JobExecution을 통해서 해결된다.

```

JobExecution jobExecution = launcher.run(getJob(), jobParameters);

//give job adequate time to start

```

```

Thread.sleep(1000);

assertEquals(BatchStatus.STARTED, jobExecution.getStatus());
assertTrue(jobExecution.isRunning());

jobExecution.stop();

//give job time to stop
Thread.sleep(1000);

assertEquals(BatchStatus.STOPPED, jobExecution.getStatus());
assertFalse(jobExecution.isRunning());

```

즉각적으로 종료(shutdown)를 강제할 수 있는 방법이 없기 때문에, 즉각적으로 모든 잡이 종료되지는 않는다. 특히 비즈니스 서비스처럼 현재 실행중인 코드가 프레임워크의 제어하에 없는 경우에는 더욱더 바로 종료할 수 없다.

실행 제어권이 프레임워크로 돌아오자마자, StepExecution의 현재 상태를 BatchStatus.STOPPED로 설정하고, 저장한 다음, 끝나기 전에 JobExecution에 대해서도 동일한 처리를 하게 된다.

4.3.2. SimpleJobRepository

SimpleJobRepository는 JobRepository 인터페이스의 유일한 구현체다. SimpleJobRepository는 다양한 배치 도메인 객체를 관리하고, 생성하고, 영속화하는 것을 보장한다.

SimpleJobRepository는 세 개의 주요한 도메인 타입을 저장하기 위해 세 개의 서로 다른 DAO 인터페이스인 JobInstanceDao, JobExecutionDao, StepExecutionDao를 사용한다. repository는 다양한 도메인 객체를 영속화하고, 초기화 동안에 하는 쿼리를 이 DAO들에게 위임한다.

```

<bean id="jobRepository"
class="org.springframework.batch.core.repository.support.SimpleJobRepository">
    <constructor-arg ref="jobInstanceDao" />
    <constructor-arg ref="jobExecutionDao" />
    <constructor-arg ref="stepExecutionDao" />
</bean>

<bean id="jobInstanceDao"
class="org.springframework.batch.core.repository.support.dao.JdbcJobInstanceDao" >
    <property name="jdbcTemplate" ref="jdbcTemplate" />
    <property name="jobIncrementer" ref="jobIncrementer" />
</bean>

<bean id="jobExecutionDao"
class="org.springframework.batch.core.repository.support.dao.JdbcJobExecutionDao" >
    <property name="jdbcTemplate" ref="jdbcTemplate" />
    <property name="jobExecutionIncrementer" ref="jobExecutionIncrementer" />
</bean>

<bean id="stepExecutionDao"
class="org.springframework.batch.core.repository.support.dao.JdbcStepExecutionDao" >
    <property name="jdbcTemplate" ref="jdbcTemplate" />
    <property name="stepExecutionIncrementer" ref="stepExecutionIncrementer" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate" >
    <property name="dataSource" ref="dataSource" />
</bean>

```

이 설정은 완전하지 않은데, 각 DAO 구현은 스프링의 DataFieldMaxValueIncrementer를 참조한다. JobInstance, JobExecution, StepExecution은 각각 유일한 ID를 갖게 되며, incrementer가 이 ID를 생성하는데 사용된다.

4.3.2.1. JobRepositoryFactoryBean

incrementer를 포함해서 특정 데이터베이스를 지정해야 하는환경 설정은 매우 성가신 일이다. 좀더 쉽게 관리할 수 있도록 편리한 FactoryBean인 JobRepositoryFactoryBean 을 제공한다.

```
<bean id="jobRepository"
      class="org.springframework.batch.execution.repository.JobRepositoryFactoryBean"
      <property name="databaseType" value="hsql" />
      <property name="dataSource" ref="dataSource" />
      <property name="transactionManager" ref="transactionManager" />
    />
```

databaseType 프로퍼티는 사용하게 될 incrementer의 타입을 지정한다.

선택 가능한 옵션

"db2", "derby", "hsql", "mysql", "oracle", "postgres"

4.3.2.2. In-Memory Repository

도메인 객체를 데이터베이스에 영속화하고 싶지 않은 시나리오가 있을 수도 있다. 빠르게 개발하거나, 특정 잡의 상태를 영속화 하고 싶지 않을 수도 있다.

```
<bean id="simpleJobRepository"
      class="org.springframework.batch.core.repository.support.SimpleJobRepository">
  <constructor-arg ref="mapJobInstanceDao" />
  <constructor-arg ref="mapJobExecutionDao" />
  <constructor-arg ref="mapStepExecutionDao" />
</bean>

<bean id="mapJobInstanceDao"
      class="org.springframework.batch.core.repository.dao.MapJobInstanceDao" />

<bean id="mapJobExecutionDao"
      class="org.springframework.batch.core.repository.dao.MapJobExecutionDao" />

<bean id="mapStepExecutionDao"
      class="org.springframework.batch.core.repository.dao.MapStepExecutionDao" />
```

Map*DAO 구현은 트랜잭션 처리가 된 맵에 배치 아티팩트를 저장한다. 그렇기 때문에, repository와 DAO는 여전히 정상적으로 사용할 수 있고, 트랜잭션 처리도 되지만, 클래스가 파괴되면 내용은 손실된다.

설정을 줄이는 별도의 in-memory JobRepository FactoryBean을 제공한다.

```
<bean id="jobRepository"
      class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean" />
```

4.3.2.2.1. JobRepository에 대한 트랜잭션 환경 설정

어떤 JobRepository 팩토리 빈을 사용하던지, transactional advice는 자동으로 repository를 둘러싸서 생성된다. 이는 실패 시에 재시작하는데 필요한 상태를 포함하는 배치 메타 데이터를 정확하게 영속화 되는 것을 보장한다.

repository 메소드에 트랜잭션이 적용되지 않은 경우에 대한 프레임워크의 행동은 제대로 정의되어 있지 않다. create* 메소드 속성의 isolation 레벨은 두 개의 프로세스에서 동시에 동일한 잡을 시작하는 것을 시도한 다면 답이 시작될 때 개별적으로 정확하게 실행됨을 보장하도록 설정되어 있다. 이 경우 단 하나만 성공하게 된다.

메소드의 기본 isolation level은 SERIALIZABLE이다. SERIALIZABLE은 매우 적극적인 방법이다.

이 경우 READ_COMMITTED를 사용하는게 더 좋다. READ_UNCOMMITTED는 두 개의 프로세스가 이런 상황에서 충돌이 발생할 일이 없는 경우라면 적절하다.

그렇지만 create* 메소드 호출이 매우 짧기 때문에, 데이터베이스 플랫폼이 지원하는 동안에는 SERIALIZED가 문제의 원인이 되는 일은 거의 없다. 그렇지만 팩토리 빈에서 재정의 할 수 있다.


```
<bean id="jobRepository"
      class="org.springframework.batch.execution.repository.JobRepositoryFactoryBean"
      <property name="databaseType" value="hsql" />
      <property name="dataSource" ref="dataSource" />
      <property name="transactionManager" ref="transactionManager" />

      <property name="IsolationLevelForCreate" value="ISOLATION_REPEATABLE_READ" />
</bean>
```

팩토리 빈을 사용하지 않은 경우에는 반드시 AOP를 사용해서 repository의 트랜잭션 처리 행동을 설정해줘야 한다.

```
<aop:config>
  <aop:advisor
    pointcut="execution(* org.springframework.batch.core.*Repository+.*(..))"
    <advice-ref="txAdvice" />
  </aop:advisor>

  <tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
      <tx:method name="*" />
    </tx:attributes>
  </tx:advice>
</aop:config>
```

이 코드는 별다른 수정 없이도 사용될 수 있다. 적절한 네이스페이스 선언을 포함하고, 클래스패스에 사용하고 있는 spring-tx와 spring-aop를 확인해야 함을 기억하자.

4.3.2.2.2 (추천)메타 데이터 테이블 인덱싱

스프링 배치는 Core jar 파일에 일반적으로 사용하는 여러 데이터베이스 플랫폼에 대한 메타 데이터 테이블을 위한 예제 DDL을 제공하고 있다. 사용자가 사용하는 플랫폼과 로컬 규약과 잡이 실행될 방법의 비즈니스 요구사항에 따라 의존적으로 많은 변수가 발생하기 때문에 인덱스 선언은 DDL에 포함되어 있지 않다. 아래 테이블은 스프링 배치에서 제공되는 Dao 구현의 WHERE 절에서 사용하는 컬럼과 얼마나 자주 사용되는지를 제시하고 있다. 이 테이블을 참조해서 각 프로젝트별로 인덱싱을 프로젝트 별로 구성할 수 있다.

기본 테이블 이름	where 절	얼마나 자주~
BATCH_JOB_INSTANCE	JOB_NAME = ? and JOB_KEY = ?	잡이 실행될 때마다
BATCH_JOB_EXECUTION	JOB_INSTANCE_ID = ?	잡이 재시작 될 때마다
BATCH_EXECUTION_CONTEXT	EXECUTION_ID = ? and KEY_NAME = ?	커밋 주기마다(묶음-chunk)
BATCH_STEP_EXECUTION	VERSION = ?	커밋 주기마다(묶음-chunk)(그리고 step 시작, 종료 시점마다)
BATCH_STEP_EXECUTION	STEP_NAME = ? and JOB_EXECUTION_ID = ?	각 step이 실행되기 전

4.3.3. SimpleJob

Job 인터페이스의 현재 구현은 SimpleJob이 유일하다. Job은 단지 Step의 리스트를 통해서 반복하기만 하면 때문에, 이 구현만으로도 필요한 주요 기능을 충족해준다. Job은 잡 이름, JobRepository, Step의 리스트를 필수 의존성을 가지고 있다.

```
<bean id="footballJob"
      class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
      <bean id="gameLoad" parent="simpleStep" />
      <bean id="playerSummarization" parent="simpleStep" />
    </list>
  </property>
  <property name="jobRepository" ref="jobRepository" />
</bean>
```

```
</bean>
```

각 Step은 모두 성공적으로 완료될 때까지 순차적으로 실행된다. 실패한 모든 Step은 전체 잡 실패의 원인이 된다.

4.3.3.1. 재시작 능력(Restartability)

배치 잡을 실행할 때 한 가지 핵심 관점은 실패된 잡을 재시작할 때 어떤 일일 일어나는가이다. Job은 한 JobExecution 보다 많은 동일한 JobInstance를 갖게되는 경우에 '재시작'을 고려한다.

이상적으로 모든 잡은 그만둔 장소에서 바로 시작할 수 있지만, 특정 시나리오에서는 불가능할 수도 있다. 이런 시나리오에서는 항상 새로운 인스턴스가 생성되는 것을 보장하는 것은 전적으로 개발자들에게 달려 있다.

그렇지만 스프링 배치는 이 상황에서 일부 도움을 준다. Job이 절대로 재시작 되면 안되지만, 항상 새로운 JobInstance의 일부로서 실행되어야 하는 경우에 restartable 프로퍼티를 'false'로 설정할 수 있다.

```
<bean id="footballJob"
      class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
      <bean id="gameLoad" parent="simpleStep" />
      <bean id="playerSummarization" parent="simpleStep" />
    </list>
  </property>
  <property name="jobRepository" ref="jobRepository" />

  <property name="restartable" value="false" />
</bean>
```

restartable이 false의 의미는 "이 Job은 다시 시작됨을 제공하지 않는다"라는 의미다. 재시작할 수 없는 Job을 재시작하려고 하면 JobRestartException을 던지게 된다.

```
Job job = new SimpleJob();
job.setRestartable(false);

JobParameters jobParameters = new JobParameters();

JobExecution firstExecution = jobRepository.createJobExecution(job, jobParameters);
jobRepository.saveOrUpdate(firstExecution);

try {
  jobRepository.createJobExecution(job, jobParameters);
  fail();
}
catch (JobRestartException e) {
  // expected
}
```

JUnit을 사용한 위 코드는 재시작을 할 수 없는 JobExecution을 만드는 장면을 보여준다. 첫 번째 JobExecution은 정상적으로 생성되지만, 두 번째 JobExecution 생성할 때는 JobRestartException이 발생한다.

4.3.3.3. Job 실행 가로채기

Job의 실행 과정 중에 커스텀 코드를 실행할 수 있도록 생명 주기에서 다양한 이벤트의 발생을 통보 받는 것은 매우 유용할 수 있다. SimpleJob은 적절한 시기에 JobListener를 호출해서 이를 가능하게 한다.

```
public interface JobListener {

  void beforeJob(JobExecution jobExecution);

  void afterJob(JobExecution jobExecution);
}
```

```

void onError(JobExecution jobExecution, Throwable e);

void onInterrupt(JobExecution jobExecution);
}

```

JobListener는 setJobListeners 프로퍼티로 SimpleJob에 추가할 수 있다.

```

<bean id="footballJob"
      class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">
    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
      <bean id="gameLoad" parent="simpleStep" />
      <bean id="playerSummarization" parent="simpleStep" />
    </list>
  </property>
  <property name="jobRepository" ref="jobRepository" />
  <property name="jobListeners">
    <bean class="org.springframework.batch.core.listener.JobListenerSupport" />
  </property>
</bean>

```

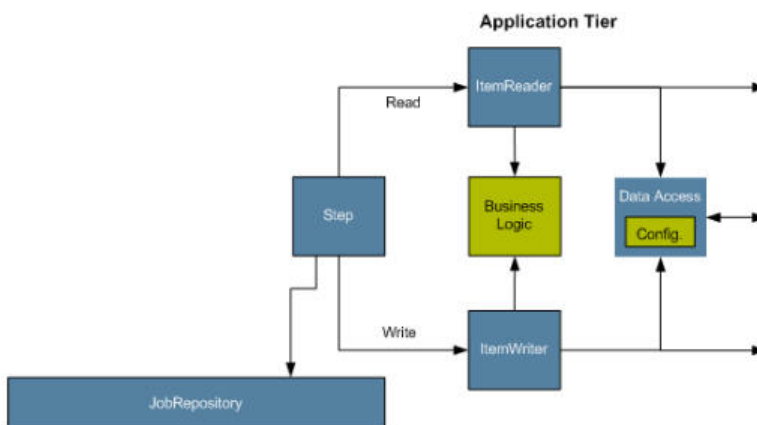
4.3.4. JobFactory와 Step에서 상태 유지 컴포넌트

전통적인 스프링 애플리케이션과 다르게 배치 애플리케이션의 모든 컴포넌트는 상태를 유지한다. 파일 reader와 writer를 예로 들 수 있다. 각 잡 실행마다 새로운 ApplicationContext 생성하도록 다루는 방법을 추천 한다. CommandLineJobRunner로 커맨드 라인에서 시작한 경우에는 별로 문제가 안 된다. 더 복잡한 시나리오는 동일한 프로세스에서 잡이 병렬이나 직렬화 되서 실행될 때, 일부 추가적인 step들은 ApplicationContext가 새로 생성되는 것을 보장 받아야 한다. 사용이 끝난 시점에 컨테이너에서 생명주기 콜백을 받지 못하기 때문에, 상태 유지 빈에서 프로토타입 범위를 사용하는게 더 좋다.

이 시나리오를 다루는 스프링 배치의 전략은 JobFactory을 제공하는 것이다. ApplicationContext를 불러오고, 잡이 끝날 때 적절하게 종료할 수 있도록 특화된 구현의 예제를 제공한다. 관련된 예제는 Samples 프로젝트에서 ClassPathXmlApplicationContextJobFactory과 adhoc-job-launcher-context.xml과 quartz-job-launcher-context.xml을 보자.

4.4. Application 티어

Application 티어는 전체적으로 실제 입력을 처리하는 과정을 책임진다.



4.4.1. ItemOrientedStep

위 그림은 간단한 '아이템-지향' 실행 흐름을 보여준다. ItemReader에서 아이템을 읽어 들이고, 아이템이 하나도 남지 않았을 때까지 ItemWriter에게 아이템을 전달한다. 일단 처리 과정이 시작되면, 트랜잭션도 시작되고, Step이 완료될 때까지 순차적으로 커밋한다. ItemOrientedStep은 다음과 같은 의존성을 기본적으로 가지고 있다.

- ItemReader: 처리할 아이템을 제공
- ItemWriter: ItemReader에 의해서 제공받은 아이템을 처리
- PlatformTransactionManager: 처리과정 동안 트랜잭션을 시작하고, 커밋하는데 사용하는 스프링 트랜잭션 관리자
- JobRepository: 처리 과정 동안 StepExecution과 ExecutionContext를 순차적으로 저장하는데 사용

4.4.1.1. SimpleStepFactoryBean

ItemOrientedStep에 필요한 의존성이 상대적으로 간단하지만, 많은 협력을 잠재적으로 포함해서 클래스가 극단적으로 복잡해질 수 있다. 설정을 쉽게 하려면 SimpleStepFactoryBean을 사용하자.

```
<bean id="simpleStep"
      class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
  <property name="transactionManager" ref="transactionManager" />
  <property name="jobRepository" ref="jobRepository" />
  <property name="itemReader" ref="itemReader" />
  <property name="itemWriter" ref="itemWriter" />
</bean>
```

이 설정은 단지 팩토리 빈이 필요한 의존성만 표현한 것이다. 이 최소한의 의존성도 설정하지 않으면 스프링 컨테이너에서 예외를 발생시킨다.

4.4.1.2. commitInterval 환경 설정

ItemOrientedStep은 아이템을 읽고, 쓰고, PlatformTransactionManager을 사용해서 순차적으로 커밋을 실행한다. 기본적으로 각 아이템들의 쓰기 작업을 한 후에 커밋을 하게 된다. 이상적으로 각 트랜잭션에서 가능한 많은 아이템을 처리하는게 더 좋다. 처리되는 데이터의 타입과 상호작용하는 자원에 완전하게 의존한다. 이런 이유로 한 번 커밋하는데서 처리되는 아이템의 갯수를 commit interval로 설정할 수 있다.

```
<bean id="simpleStep"
      class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
  <property name="transactionManager" ref="transactionManager" />
  <property name="jobRepository" ref="jobRepository" />
  <property name="itemReader" ref="itemReader" />
  <property name="itemWriter" ref="itemWriter" />
  <property name="commitInterval" value="10" />
</bean>
```

이번 예에서는 각 트랜잭션에서 10개의 아이템을 처리한다. 처리 과정이 시작하면 트랜잭션이 시작되고, ItemReader에서 매번 아이템을 읽어 들인 다음, 를 증가시킨다.

4.4.1.3. Restart에 대한 Step 설정

4.4.1.3.1. StartLimit 설정

많은 시나리오에서 Step의 실행을 제어하길 원할 수 있다. 일단 한 번 시작된 후에, 다시 시작하기 전에 잘못된 부분을 직접 수정해야할 수 있다. 다른 step에서는 다른 요구사항을 갖을 수 있기 때문에, step 레벨에서 설정할 수 있다.

```
<bean id="simpleStep"
      class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
  <property name="transactionManager" ref="transactionManager" />
  <property name="jobRepository" ref="jobRepository" />
  <property name="itemReader" ref="itemReader" />
  <property name="itemWriter" ref="itemWriter" />
  <property name="commitInterval" value="10" />
  <property name="startLimit" value="1" />
</bean>
```

이 step은 단 한 번만 실행된다. startLimit의 기본값은 Integer.MAX_VALUE 다.

4.4.1.3.2. 완료된 step 재시작하기

재시작이 가능한 잡의 경우에 첫 번째 시도의 성공 여부에 관계 없이 하나 이상의 step이 언제나 실행되도록 할 수 있다. 유효한 step이나 처리하기 전에 자원을 정리한 step을 예로 들 수 있다. 재시작되는 잡의 일반적인 처리 과정 동안에 이미 성공적으로 완료했다는 것을 의미하는 'COMPLETED' 상태인 모든 step은 건너뛰게 된다.

```
<bean id="simpleStep"
      class="org.springframework.batch.core.step.item.SimpleStepFactoryBean" >
  <property name="transactionManager" ref="transactionManager" />
  <property name="jobRepository" ref="jobRepository" />
  <property name="itemReader" ref="itemReader" />
  <property name="itemWriter" ref="itemWriter" />
  <property name="commitInterval" value="10" />
  <property name="startLimit" value="1" />
  <property name="allowStartIfComplete" value="true" />
</bean>
```

4.4.1.3.3. Step 재시작 설정 예제

```
<bean id="footballJob"
      class="org.springframework.batch.core.job.SimpleJob">
  <property name="steps">

    <list>
      <!-- Step Bean details omitted for clarity -->
      <bean id="playerload" parent="simpleStep" />
      <bean id="gameLoad" parent="simpleStep" >
        <property name="allowStartIfComplete" value="true" />
      </bean>
      <bean id="playerSummarization" parent="simpleStep" >
        <property name="startLimit" value="2" />
      </bean>
    </list>
  </property>
  <property name="jobRepository" ref="jobRepository" />
  <property name="restartable" value="true" />
</bean>
```

첫 번째 실행 :

- playerLoad: 성공적으로 실행. 'PLAYERS' 테이블에 400명 선수 추가
- gameLoad: 11 파일을 처리하고, 'GAMES' 테이블에서 컨텐츠 로딩
- playerSummarization: 처리 과정을 시작했지만, 5분 뒤에 실패

두 번째 실행 :

- playerLoad: 이미 성공했기 때문에, allowStartIfComplete가 false로 설정되어 있기 때문에 실행되지 않아요..
- gameLoad: 실행되서 또 다른 2 개 파일을 처리. 'GAMES' 테이블에서 컨텐츠 로딩
- playerSummarization: 모든 나머지 게임 데이터의 처리를 시작했지만, 30분 후에 다시 실패 됨

세 번째 실행 :

- playerLoad: 이미 성공했기 때문에, allowStartIfComplete가 false로 설정되어 있기 때문에 실행되지 않아요..
- gameLoad: 실행되서 또 다른 2 개 파일을 처리. 'GAMES' 테이블에서 컨텐츠 로딩
- playerSummarization: 시작 제한이 2로 설정되어 있는데, 이번 실행이 세 번째이므로, 잡이 바로 종료된다.

4.4.1.4. Skip 로직 설정하기

Step이 실패하지 않지만, 대신 그냥 지나쳐야 하는 상황이 많이 발생한다. 이런 의사 결정은 데이터의 의미에 대해서 정확하게 이해하는 누군가가 해야 한다.

SkipLimitStepFactoryBean 팩토리 빈을 사용해서 지니가야 하는 상황에 대한 요구 사항을 설정할 수 있다.

```
<bean id="skipSample"
```

```

        class="org.springframework.batch.core.step.item.SkipLimitStepFactoryBean">
        <property name="skipLimit" value="10" />
        <property name="itemReader" ref="flatFileItemReader" />
        <property name="itemWriter" ref="itemWriter" />
        <property name="skippableExceptionClasses"
            value="org.springframework.batch.item.file.FlatFileParseException">
        </property>
    </bean>

```

이 예에서 FlatFileParseException이 던져지면, 지나가게 되고(skip), skip limit의 횟수를 1 증가 한다.(여기서는 10이 제한..) 이 횟수는 절대로 commit interval로 세지 않는다. commit interval은 쓰기 작업을 했을때만 증가시킨다.(성공이나 실패나에 관계없이)

이 예에서 한 가지 문제점은 FlatFileParseException을 제외한 다른 모든 예외는 Job 실패의 원인이 된다는 것이다. 실패하는 잡과 지나가는 잡의 원인이 되는 예외를 설정할 수 있다.

```

<bean id="skipSample"
    class="org.springframework.batch.core.step.item.SkipLimitStepFactoryBean">
    <property name="skipLimit" value="10" />
    <property name="itemReader" ref="flatFileItemReader" />
    <property name="itemWriter" ref="itemWriter" />
    <property name="skippableExceptionClasses"
        value="java.lang.Exception">
    <property name="fatalExceptionClasses"
        value="java.io.FileNotFoundException">
    </property>
</bean>

```

java.lang.Exception으로 설정되어 있으므로, 어떤 예외든지 던져지만, 바로 지나치게 된다. 'fatalExceptionClasses'에 발생 시 심각한 상황이 되는 예외를 지정하게 된다.

4.4.1.6. Retry 로직 설정하기

```

<bean id="step1"
    class="org.springframework.batch.core.step.item.SkipLimitStepFactoryBean">
    <property name="itemReader" ref="itemGenerator" />
    <property name="itemWriter" ref="itemWriter" />
    <property name="retryLimit" value="3" />
    <property name="retryableExceptionClasses"
        value="org.springframework.dao.DeadlockLoserDataAccessException" />
</bean>

```

4.4.1.7. 롤백 제어하기

SkipLimitStepFactoryBean에는 롤백의 원인이 되지 않는 예외의 리스트를 설정할 수 있다.

```

<bean id="step2"
    class="org.springframework.batch.core.step.item.SkipLimitStepFactoryBean">
    <property name="commitInterval" value="2" />
    <property name="skipLimit" value="1" />
    <!-- No rollback for exceptions that are marked with "+" in the tx attributes -->
    <property name="transactionAttribute"
        value="+org.springframework.batch.item.validator.ValidationException" />
    <property name="itemReader"
        ref="tradeSqlItemReader" />
    <property name="itemWriter"
        ref="itemTrackingWriter" />
</bean>

```

transactionAttribute 프로퍼티는 isolation 레벨이나 propagation 행동과 같은 다양한 트랜잭션 관련 설정을 하는데 사용된다.

4.4.1.8. Step으로 ItemStream 등록하기

//todo

4.4.1.9. Step 실행 가로채기

//todo

4.4.2. TaskletStep

//todo

4.5. 커스터마이징된 비즈니스 로직 예제

//TODO

5. 반복하기

5.1. RepeatTemplate

배치 처리 과정은 단순한 최적화된 잡의 일부로든 모두 반복적인 행동이 된다. 반복을 전략적이면서 일반화하고, iterator 프레임워크를 제공하기 위해, 스프링 배치는 RepeatOperations 인터페이스를 가지고 있다.

```
public interface RepeatOperations {  
    ExitStatus iterate(RepeatCallback callback) throws RepeatException;  
}
```

콜백은 반복되는 비즈니스 로직을 추가하도록 해주는 간단한 인터페이스다.

```
public interface RepeatCallback {  
    ExitStatus doInIteration(RepeatContext context) throws Exception;  
}
```

콜백은 반복이 끝났다고 결정할 때까지 반복적으로 실행된다. 이 인터페이스에서 반환하는 값은 확장 가능하게 전문화된 enumeration이다.(사용자가 새로운 값을 만드는게 자유롭기 때문에 진정한 enumeration은 아니다) ExitStatus는 불변이며, 어떤 일을 더 해야하는지 마는지에 대한 정보를 연산 호출자에게 전달한다. 대체로 RepeatOperations의 구현은 ExitStatus를 확인하고, 반복이 끝나야 하는지를 결정하는 기준의 일부로 사용한다. 호출자에게 더 이상 할 일이 없다는 신호를 보내고자 하는 모든 콜백은 ExitStatus.FINISHED를 반환하면 된다.

RepeatOperations의 가장 간단하면서 일반적인 목적을 갖는 구현은 RepeatTemplate이다. 다음처럼 사용한다.

```
RepeatTemplate template = new RepeatTemplate();  
  
template.setCompletionPolicy(new FixedChunkSizeCompletionPolicy(2));  
  
template.iterate(new RepeatCallback() {  
  
    public ExitStatus doInIteration(RepeatContext context) {  
        // ## ## #####...  
        return ExitStatus.CONTINUABLE;  
    }  
  
});
```

이 예에서는 계속 할 일이 있다는 것을 보여주는 ExitStatus.CONTINUABLE를 반환한다. 콜백에서 완료되어야 하는 본질에 대한 고려에 의해서 반복(iterations)을 중단될 수 있다. 그 밖에 콜백이 관심 있을 때까지 효과적으로 무한 반복(loop)을 돌 수도 있고, 위 예의 경우에서 처럼 이러한 결정이 외부 정책에 위임되기도 한다.

5.1.1. RepeatContext

RepeatContext는 RepeatCallback의 메소드 인자다. 많은 콜백들은 단순히 context를 무시하지만, 반복 하는 동안에 일시적으로 사용할 필요가 있는 데이터를 저장하는 속성 가방(attribute bag)으로서 사용될 수 있다. iterate 메소드가 결과를 반환한 후에, context는 더 이상 존재하지 않게 된다.

RepeatContext는 처리 과정에서 내제된 반복이 필요한 경우 부모 context를 갖게 된다. 종종 부모 context는 반복되는 호출 사이에 공유할 필요가 있는 데이터를 저장하는데 유용하다.

5.1.2. ExitStatus

ExitStatus는 스프링 배치에서 처리 과정이 끝났고, 처리 과정이 성공인지 아닌지를 지정하는 목적으로 사용한다. 또는 배치나 반복의 종료 상태에 대한 원문 자체의 정보(textual information)을 운반하도록 사용되기도 한다. 이 정보는 종료 코드의 형태와 자유 형식의 문자의 상태에 대한 설명이 된다.

프로퍼티 이름	타입	설명
continuable	boolean	좀더 할 일이 있다면 true
exitCode	String	CONTINUABLE, FINISHED, FAILED 처럼 종료 상태를 설명하는 짧은 코드
exitDescription	String	종료 상태에 대한 긴 설명(예를 들어, 스택 트레이스)

ExitStatus 값은 유연하게 디자인 되기 때문에, 사용자가 필요로 하는 코드나 설명을 사용해서 만들 수 있다. 스프링 배치는 공통적인 경우를 지원하는 표준화된 값을 갖고 있지만, 기존의 지속가능한 프로퍼티(continuable property)의 의미를 인정하는 한 사용자들은 자체적으로 값을 자유롭게 만들 수 있다.

ExitStatus 값은 메소드로 클래스에 내장된 다양한 연산자(operator)와 함께 조합될 수 있다. 종료 코드나 설명, ExitStatus에 있는 메소드를 사용해서 논리적인 AND와 함께 지속가능한 값을 조합해서 추가할 수 있다. 인자로 ExitStatus를 받는 and() 메소드를 사용해서 두 개의 ExitStatus 값을 조합할 수도 있다. 이 방법의 효과는 결과가 지속 가능하거나 입력이 지속가능하지 않은 동안 지속가능한 표시(continuable flag)에 대해 논리적인 AND를 수행하고, 설명을 연결하고, 종료 코드를 새로운 값으로 바꾼다. 이 효과는 지속가능한 표시의 의미를 유지해주는 효과를 갖지만, 종료 코드에 "놀랄만한" 변경을 만들지는 못한다.(예를 들어, 지속 가능하지 못한 값을 전달하는 것처럼 고의적으로 어떤 행동을 하지 않는 이상, 이미 FINISHED가 됐을 때는, 절대로 CONTINUABLE이 될 수 없다.)

5.2. 완료 정책

RepeatTemplate 내에서 iterate 메소드에 있는 루프의 종료는 CompletionPolicy에 의해서 결정된다. CompletionPolicy는 RepeatContext에 대한 팩토리도 된다. RepeatTemplate은 RepeatContext를 생성하는 현재 정책을 사용해서 반복 중 모든 단계에서 RepeatCallback에게 전달해야 하는 책임을 가지고 있다. 콜백이 완료된 후에 RepeatTemplate의 doInIteration는 상태를 갱신해야 하는지(RepeatContext에 저장될 것인지) 여부를 CompletionPolicy에게 물어보는 호출을 하게 된다. 그 다음으로 반복이 완료된 경우에 정책을 요청하게 된다.

스프링 배치는 간단하게 일반적인 목적으로 사용되는 CompletionPolicy 구현체를 제공한다. 위 예에서 사용한 SimpleCompletionPolicy을 예로 들 수 있다. SimpleCompletionPolicy은 고정된 시간만큼만 실행을 허용한다. (ExitStatus.FINISHED로 정해진 시간보다 강제로 일찍 완료할 수 있다.)

좀더 복잡한 의사결정에서는 사용자가 자체적으로 완벽하게 맞는 정책을 구현할 필요가 있을 수도 있다.


5.3. 예외 다루기

RepeatCallback 내에서 예외가 던져지는 경우, RepeatTemplate은 예외를 다시 던져야 하는지를 결정하는데 ExceptionHandler에게 의견을 묻게 된다.

```
public interface ExceptionHandler {  
    void handleException(RepeatContext context, Throwable throwable) throws RuntimeException;  
}
```

공통적으로 사용되는 경우는 주어진 타입의 예외 발생 횟수를 세는 것과 한계에 도달했을 때 실패를 일으키는 것이다. 이러한 목적에 맞게 스프링 배치는 SimpleLimitExceptionHandler와 비슷하지만 좀더 유연한 RethrowOnThresholdExceptionHandler를 제공한다.

SimpleLimitExceptionHandler는 limit 프로퍼티와 현재 예외와 비교하는 예외 타입을 가지고 있다. 이 때 제공된 타입의 모든 하위 클래스들도 계산한다. 주어진 타입의 예외는 한계에 도달할 때까지 무시된다. 그리고 나서는 다시 던져지게 된다. 다른 예외 타입은 항상 다시 던진다.

SimpleLimitExceptionHandler의 선택 가능한 중요한 프로퍼티는 useParent boolean 표시다. 기본값은 false기 때문에, 한계는 현재 RepeatContext에서만 설명된다.  true로 설정됐을 때 한계는 내제된 반복(nested iteration)에서 형제 context에 걸쳐서 유지된다.

5.4 리스너(Listeners)

종종 서로 다른 여러 반복에서 cross-cutting concern(다수의 반복에서 공통으로 관심을 갖고 있는) 하는 추가적인 콜백을 받아올 수 있도록 하는게 유용할 때가 있다. 이러한 목적으로 스프링 배치는 RepeatListener 인터페이스를 제공한다. RepeatTemplate은 사용자가 RepeatListener를 등록할 수 있게 해준다. 그리고 콜백 반복 중에 이용할 수 있도록 RepeatContext와 ExitStatus를 전달한다.

```
public interface RepeatListener {
    /*
     * # ##### ##
     */
    void before(RepeatContext context);

    /*
     * ### ##### ## ##, # ##### ## ## ##.
     */
    void after(RepeatContext context, ExitStatus result);

    /*
     *
     */
    void open(RepeatContext context);

    void onError(RepeatContext context, Throwable e);

    void close(RepeatContext context);
}
```

open()과 close() 콜백은 전체 반복 전, 후에 호출되고, before(), after(), onError()는 개별적인 RepeatCallback 호출에 적용된다.

하나 이상의 리스너가 있을 때, 리스트에 들어가 있게 된다. 그러므로 리스너는 순서를 갖게 된다. open()과 before()는 동일한 순서로 호출되며, after(), onError(), close()는 반대 순서로 호출된다.

5.5 병렬 처리하기(parallel processing)

RepeatOperations의 구현은 순차적으로 콜백을 실행하도록 제한하지 못한다. 구현은 동시에 콜백이 실행할 수 있도록 하는 건 제일 중요하다. 이 때문에 스프링 배치는 RepeatCallback을 실행하는데 TaskExecutor 전략을 사용하는 TaskExecutorRepeatTemplate을 제공한다. 기본 구현으로는 (일반 RepeatTemplate과 같은) 동일한 쓰레드에 있는 전체 반복을 실행하는데 영향을 미치는 SynchronousTaskExecutor를 사용한다.

5.6 선언적 반복

때때로 발생할 때마다 반복하고 싶은 비즈니스 처리과정이 있다. 고전적인 예제로 메세지 파이프라인의 최적화가 있다. 메세지를 자주 받게 되는 경우, 매세지 마다 개별적인 트랜잭션으로 처리하는 비용을 참기 보다는 메세지를 배치로 처리하는게 좀더 효율적이다. 스프링 배치는 딱 이목적에 맞게 RepeatOperations에서 메소드 호출을 감싸는 AOP 인터셉터를 제공한다. RepeatOperationsInterceptor는 가로챈 메소드를 실행하고, 제공된 RepeatTemplate에 CompetitionPolicy에 따라서 반복하게 된다.

여기서는 스프링 AOP 네임스페이스를 사용해서 호출되는 processMessage 메소드를 호출하는 서비스를 반복하는데 선언적인 반복의 예를 보자.

```
<aop:config>
  <aop:pointcut id="transactional"
```

```

        expression="execution(* com...*Service.processMessage(..))" />
        <aop:advisor pointcut-ref="transactional"
            advice-ref="retryAdvice" order="-1"/>
    </aop:config>

    <bean id="retryAdvice"
        class="org.springframework.batch.repeat.interceptor.RepeatOperationsInterceptor"/>

```

위 예에서는 인터셉터 내부에서 기본 RepeatTemplate을 사용한다. 정책, 리스너 등을 변경하려면 인터셉터에 RepeatTemplate을 주입할 필요가 있다.

가로챌 메소드가 void 반환 타입이라면, 인터셉터는 언제나 ExitStatus.CONTINUABLE을 반환한다. (그렇기 때문에 CompletionPolicy가 한정된 종료 지점이 없는 경우라면 무한 반복의 위험이 있다.) 만약 그렇지 않다면 가로챌 메소드에서 ExitStatus.FINISHED를 반환하는 지점이 되는 null을 반환할 때까지 ExitStatus.CONTINUABLE을 반환한다. 그래서 대상 메소드 내에 있는 비즈니스 로직은 null을 반환하거나 RepeatTemplate에서 제공하는 ExceptionHandler에 의해서 다시 던진 예외를 던져서 더 할 일이 없다는 신호를 보낼 수 있거나

6. 재시도하기

6.2 RetryTemplate

좀더 견고하고, 덜 성가시게 실패를 처리하고 싶다면, 바로 이어서 시도해서 성공할 수 있다고 생각되는 경우, 자동으로 실패한 연산을 재시도하는 게 도움이 된다. 이러한 종류의 예러는 사실상 일시적으로 발생한 예러로 볼 수 있다. 네트워크 문제로 실패한 웹 서비스나 ROM 서비스나 데이터베이스 갱신에서 발생한 DeadLockLoserException을 예로 들 수 있다. 스프링 배치에서는 이러한 연산을 자동으로 재시도 하려고 위한 RestryOperations 전략을 갖고 있다.

```

public interface RetryOperations {

    Object execute(RetryCallback retryCallback) throws Exception;

}

```

콜백은 재시도하는 비즈니스 로직을 넣을 수 있는 간단한 인터페이스다.

```

public interface RetryCallback {

    Object doWithRetry(RetryContext context) throws Throwable;

}

```

콜백이 실행되고, 예외가 발생해서 실패하는 경우, 성공할때까지 재시도를 하게 된다. 또는 구현 여부에 따라 취소 여부를 결정한다.

가장 간단한 일반적으로 목적의 RetryOperations의 구현은 RetryTemplate이다.

```

RetryTemplate template = new RetryTemplate();

template.setRetryPolicy(new TimeoutRetryPolicy(30000L));

Object result = template.execute(new RetryCallback() {

    public Object doWithRetry(RetryContext context) {
        // Do stuff that might fail, e.g. webservice operation
        return result;
    }

});

```

이 예제에서는 웹 서비스 호출을 실행하고, 결과를 사용자에게 반환한다. 호출이 실패하면, 타임아웃이 될때까지 재시도한다.

6.1.1. RetryContext

RetryCallback의 메소드 매개변수로 RetryContext가 있다. 대부분의 콜백에서는 단순히 이 context를 무시하지만, 필요하다면 반복 동안에 데이터를 저장하는 속성 가방(attribute bag)으로 사용할 수 있다.

6.2 RetryPolicy

RetryTemplate에서 execute() 메소드에서 재시도나 실패를 결정하는건 RetryPolicy에 의해서 결정된다. RetryPolicy는 RetryContext의 팩토리가 되기도 한다. RetryTemplate은 RetryContext를 만들기 위해서 현재 정책을 사용해야 할 책임을 갖고, 시도마다 RetryCallback에 이를 건네야 한다. 콜백이 실패한 후에 RetryTemplate은 상태를 갱신하려고 RetryPolicy를 호출하며(RetryContext에 저장된다), 그 다음으로 또 다른 시도를 할 수 있는 경우에 RetryPolicy를 호출해서 정책을 문의하게 된다. (예를 들어, 제한에 걸렸거나 타입아웃이 되버린 것처럼) 또 다른 시도를 하지 못하게 되면, 정책은 다 사용된 상태를 관리하는 책임을 진다.

단순히 구현하자면 RetryExhaustedException을 던지게 되고, 관련된 트랜잭션은 롤백된다. 좀 더 정교하게 구현하자면, 트랜잭션을 손상하지 않고 유지할 수 있는 경우에 복구 행동을 시도할 수 있겠다.

tip

실패는 태생적으로 재시도를 할 수 있는지 없는지가 결정된다. 일부 예외는 언제나 비즈니스 로직 문제로 던져지므로 재시작 하는데 전혀 도움이 되지 못한다. 그러므로 모든 예외 타입에 대해서 재시도를 하면 안된다. 오로지 재시도를 할 수 있는 예외에만 집중해야 한다. 좀더 공격적으로 재시도를 처리하는건 일반적으로 비즈니스 로직에 해가 되지는 않지만, 루프에 매우 관련이 많은 재시도를 하는 실패가 확실하다면 비경제적이 된다.

6.2.1. 무상태 재시도

가장 간단한 retry 예는 RetryTemplate이 성공이나 실패를 할때까지 시도하는 행위를 유지할 수 있는 루프(loop)이 있다. RetryContext는 재시도 할건지 취소할 건지를 결정하는데 사용하는 상태를 포함하지만, 이 상태는 스택에 저장되고, 어디서나 접근하도록 글로벌하게 저장할 필요는 없다. 그러므로 우리는 이 방법을 무상태 재시도(stateless retry)로 부른다. 무상태 재시도와 상태 유지 재시도 사이의 차이는 RetryPolicy의 구현을 포함하는가 이다. (RetryTemplate은 둘 다 제어할 수 있다) 무상태 재시도에서 콜백은 실패가 될때마다 재시도가 항상 동일한 스레드에서 실행된다.

스프링 배치는 간단하며 일반적인 무상태 RetryPolicy의 구현을 제공한다. 위 예에서 사용한 SimpleRepositoryPolicy나 TimeoutRetryPolicy를 예로 들 수 있다.

SimpleRetryPolicy는 예외 타입의 목록에 이름이 있는 경우에만 정해진 횟수만큼 재시도를 허용한다. 또한 절대로 재시도 하면 안되는 "치명적인(fatal)" 예외의 목록을 갖는다. 재시도할 수 있는 리스트를 재정의해서, 재시도 행동을 좀더 미묘하게 제어하는데 사용할 수 있다.

예를 들면,

```
SimpleRetryPolicy policy = new SimpleRetryPolicy(5);
// Retry on all exceptions (this is the default)
policy.setRetryableExceptions(new Class[] {Exception.class});
// ... but never retry IllegalStateException
policy.setFatalExceptions(new Class[] {IllegalStateException.class});


// Use the policy...
RetryTemplate template = new RetryTemplate();
template.setRetryPolicy(policy);
template.execute(new RetryCallback() {
    public Object doWithRetry(RetryContext context) {
        // business logic here
    }
});
```

ExceptionClassifier 추상화를 통해 예외 타입의 임의 집합에 대한 재시도 행위를 사용자가 구성할 수 있게 해주는 ExceptionClassifierRetryPolicy처럼 좀더 유연한 구현도 있다. 정책은 예외를 RetryPolicy에 위임해서 변환해주는 classifier의 호출에 의해 작동된다. 예를 들어, 한 예외 타입은 다른 정책으로 매핑되어 또 다른 실패 보다 이전에 좀더 많이 재시도될 수 있다.

좀더 커스터마이징된 결정을 위해서 자체적인 재시도 정책을 구현할 필요가 있는 사용자도 있을 수 있다.


6.2.2 상태유지 재시도

트랜잭션 처리를 하는 자원이 무효화되는 것이 원인이 되는 실패는 특별히 몇 가지를 고려해봐야 한다. (일반적으로) 트랜잭션을 처리가 없는 자원인 간단한 원격 호출에 적용될 뿐만 아니라, 때로는 특히 하이버네이트를 사용처럼, 데이터베이스 갱신에도 적용된다. 이럴 경우 트랜잭션이 롤백되어서, 다시 유용한 트랜잭션으로 시작할 수 있도록, 실패가 되자마자 예외를 다시 던지는게 상황에 맞다.

예외를 다시 던지고(re-throw), 롤백하는 건 필수적으로 남은 `RetryOperations.execute()` 메소드를 포함해서 , 잠재적으로 스택에 있는 context를 손실하게 되기 때문에 이런 경우에 재시도로 충분치 못하다. 이 손실을 피하기 위해서는 스택에서 context를 빼내서, 힙 저장 영역에 넣어두는 저장 전략을 도입해야 한다. 이러한 목적으로 스프링 배치는 `RetryContextCache`를 제공한다. `RetryContextCache`의 기본 구현은 단순하게 Map을 사용해서 메모리에 저장한다. 클러스터 환경에서 다수의 프로세스로 처리하는 고급 사용법은 여러 종류이 클러스터 캐시를 사용하는 `RetryContextCache` 구현을 고려해보자.(여기서 클러스터 환경은 지나친 상상일 수 있다.)

6.2.2.1. Item 처리과정과 상태유지 재시도

상태유지 재시도 정책의 책임의 일부 중 새로운 트랜잭션이 돌아 왔을 때 실패한 연산을 알아차리는 것이다. 객체가 처리되는 가장 일반화된 경우에서 이를 처리하기 위해서 스프링 배치는 `ItemWriterRetryPolicy`를 제공한다. 이 클래스에 전문화된 `RetryCallback`의 구현인 `ItemWriterRetryCallback`과 협력해서 작동한다. `ItemWriterRetryCallback`는 사용자가 제공하는 `ItemWriter`에 대한 의존성을 갖는다. 이 콜백은 아이템을 writer에게 넘기는 일반화된 패턴을 구현했다.

이 구현에서 실패된 연산을 알아차리는 방법은 여러 재시도의 호출에 걸쳐 있는 아이템을 식별하는 방법이다. 아이템을 식별하려면, 사용자는 `ItemKeyGenerator` 전략을 제공할 수 있고, 이 클래스는 아이템을 식별하는 유일한 키를 반환하는 책임을 갖고 있다. 이 식별자는 `RetryCacheContext`에서 키로써 사용된다. `ItemKeyGenerator`는 `ItemWriterRetryCallback`에 직접 주입되거나 `ItemWriter`에서 인터페이스를 구현하거나, 단순히 아이템을 자체적인 키로써 사용하는 기본적인 접근으로 제공할 수 있다. 



주의

기본 아이템 생성 전략을 사용하는 경우라면, 아이템 클래스에서 `Object.equals()`와 `Object.hashCode()` 구현은 매우 조심해야 한다. 특히 `ItemWriter`가 아이템을 데이터베이스에 삽입하고, 주기를 갱신할 생각이라면, `ItemWriter` 호출 전, 후에 이 값이 변경될 것이기 때문에, `equals()`와 `hashCode()`에서 주기를 사용하는 건 좋은 생각이 아니다. 제일 권하고 싶은 방법은 아이템을 식별하는데 비즈니스 키를 사용하는 것이다.

상태유지 재시도는 항상 새로운 트랜잭션에서 시작되기 때문에, 재시도를 다 해버렸을 때, `RetryCallback`를 호출하는 대신에 다른 방법으로 실패한 아이템을 제어할 수 있는 옵션이 있다. 이 옵션은 `ItemRecoverer` 전략(strategy)에 의해서 제공된다. 키 생성기(key generator)처럼 이 클래스는 `ItemWriter`에서 인터페이스를 구현함으로써 직접 주입되거나 제공될 수 있다.

재시도 여부의 결정은 실제로 평범한 무상태 재시도 정책에 위임할 수 있어서, 프로퍼티 위임을 통해서 대개 제한이나 타임아웃에 대한 관심이 `ItemWriterRetryPolicy`로 주입된다.

Backoff 정책

일시적인 실패 후에 재시도할 때, 실패의 원인이 되는 일부 문제들은 단지 잠시 기다리기만 해도 해결되는 경우가 있기 때문에, 많은 경우 다시 시도해보기 전에 잠시 기다리는게 도움이 되기도 한다. `RetryCallback`이 실패한 경우, `RetryTemplate`은 적절하게 `BackoffPolicy`에 따라서 실행을 잠시 멈춘다.

```
public interface BackoffPolicy {

    BackOffContext start(RetryContext context);

    void backOff(BackOffContext backOffContext)
        throws BackOffInterruptedException;

}
```

`BackoffPolicy`는 자유롭게 방법을 선택해서 구현하면 된다. 스프링 배치에서 제공되는 정책은 특별히 `Object.wait()`를 사용한다. 공통적인 쓰임새는 두 개의 재시도가 락에 들어가 버리고, 두 시도 모두 실패해버리는 걸 피하기 위해, 전형적으로 기다리는 시간 범위를 증가하는 backoff가 있다. 이러한 목적으로 스프링 배치는 `ExponentialBackoffPolicy`를 제공한다.

6.4. 리스너

종종 서로 다른 다수의 반복에서 cross-cutting concern(다수의 반복에서 공통으로 관심을 갖고 있는) 하는 추가적인 콜백을 받아올 수 있도록 하는게 유용할 때가 있다. 이러한 목적으로 스프링 배치는 RetryListener 인터페이스를 제공한다. RetryTemplate은 사용자가 RetryListener를 등록하도록 해주며, 반복 동안에 이용할 수 있는 RetryContext와 Throwable와 함께 콜백에 전해진다.

```
public interface RetryListener {

    void open(RetryContext context, RetryCallback callback);

    void onError(RetryContext context, RetryCallback callback, Throwable e);

    void close(RetryContext context, RetryCallback callback, Throwable e);

}
```

단순한 경우에 open()과 close() 콜백은 전체 재시도 전, 후에 호출되며, onError()는 개별적인 RetryCallback 호출에 적용된다. 또한 close() 메소드는 RetryCallback에 의해서 마지막에 던져진 예러가 있는 경우에 Throwable(인스턴스겠죠..)을 받아오게 된다.

다수의 리스너를 리스트로 줄 수 있고, 순서를 갖게 된다는 점을 기억하자. open() 메소드의 경우 같은 순서로 호출되며, onError()와 close()는 역순으로 호출된다.

선언적인 재시도

때때로 발생할 때마다 재시작하고 싶은 비즈니스 처리과정이 있다. 고전적인 예로 월격 서비스 호출이 있다. 스프링 배치는 이러한 목적에 딱 맞는 RetryOperations에서 호출되는 메소드를 감싸는 AOP 인터셉터를 제공한다. RetryOperationsInterceptor는 가로챈 메소드를 실행하고, 제공된 RetryTemplate에 있는 RetryPolicy에 따라서 실패를 재시도 한다.

아래는 remoteCall()을 호출하는 메소드 서비스 호출을 재시도 하는데 스프링 AOP 네임스페이스를 사용하는 선언적인 재시도의 예제다.

```
<aop:config>
  <aop:pointcut id="transactional"
    expression="execution(* com...*Service.remoteCall(..))" />
  <aop:advisor pointcut-ref="transactional"
    advice-ref="retryAdvice" order="-1"/>
</aop:config>

<bean id="retryAdvice"
  class="org.springframework.batch.retry.interceptor.RetryOperationsInterceptor"/>
```

이 예에서는 인터셉터 내에 있는 기본 RetryTemplate을 사용한다.