ch 02. 스프링 배치 아키텍처

(1) 스프링 배치 기본 구조

* Job은 JobLauncher에 의해 실행
* Job은 배치의 실행 단위를 의미
* Job은 N개의 Step을 실행할 수 있으며, 흐름(Flow)을 관리할 수 있다.
    - 예를 들면, A step 실행 후 조건에 따라 B step 또는 C step을 실행 설정

* Step은 Job의 세부 실행 단위이며, N개가 등록돼 실행된다.
* Step의 실행 단위는 크게 2가지로 나눌 수 있다.
    - Chunk 기반 : 하나의 큰 덩어리를 n개씩 나눠서 실행
    - Task 기반 : 하나의 작업 기반으로 실행
* Chunk 기반 Step은 ItemReader, ItemProcessor, ItemWriter가 있다.
    - 여기서 item은 배치 처리 대상 객체를 의미한다.

* ItemReader는 배치 처리 대상 객체를 읽어 ItemProcessor 또는 ItemWriter에 전달한다.
    - 예를 들면, 파일 또는 DB에서 데이터를 읽는다.
* ItemProcessor는 input 객체를 output 객체로 filtering 또는 ItemWriter에게 전달한다.
    - ItemReader에서 읽은 데이터를 수정 또는 ItemWriter 대상인지 filtering 한다.
    - ItemProcessor는 optional 하다.
    - ItemProcessor가 하는 일을 ItemReader 또는 ItemWriter가 대신할 수 있다.
* ItemWriter는 배치 처리 대상 객체를 처리한다.
    - DB update를 하거나, 처리 대상 사용자에게 알림을 보낸다.

(2) 스프링 배치 테이블 구조와 이해

* 배치 실행을 위한 메타 데이터가 저장되는 테이블

* BATCH_JOB_INSTANCE
    - Job이 실해되며 생성되는 최상위 계층의 테이블
    - job_name과 job_key를 기준으로 하나의 row가 생성
    - 같은 job_name과 job_key가 저장될 수 없다.
    - job_key는 BATCH_JOB_EXECUTION_PARAM에 저장되는 파라미터를 나열해 암호화해 저장한다.

* BATCH_JOB_EXECUTION
    - Job이 샐행되는 동안 시작/종료 시각, job 상태등을 관리

* BATCH_JOB_EXECUTION_CONTEXT
    - Job이 실행되며 공유해야할 데이터를 직렬화해 저장
    - 이는 하위 Step들이 공유하여 사용

* BATCH_JOB_EXECUTION_PARAMS
    - Job을 실행하기 위해 주입된 parameter 정보 저장

* BATCH_STEP_EXECUTION
    - Step이 실행되는 동안 필요한 데이터 또는 실행된 결과 저장

* BATCH_STEP_EXECUTION_CONTEXT
    - Step이 실행되며 공유해야할 데이터를 직렬화해 저장(다른 Step과 공유x)
* spring-bath-core/org.springframework/batch/core/*
* 스프링 배치를 실행하고 관리하기 위한 테이블 생성 sql이 존재

* schema.sql 설정
    - schema-**.sql 의 실행 구분은 DB종류별로 script가 구분
    - spring.batch.initialize-schema config로 구분한다.
    - ALWAYS, EMBEDDED, NEVER로 구분한다.
        ALWAYS : 항상 실행
        EMBEDDED : 내장 DB일 때만 실행
        NEVER : 항상 실행 안함
    - 기본값은 EMBEDDED다.

(3) 객체 관리 with JPA

* JobInstance : BATCH_JOB_INSTANCE 테이블과 매핑
* JobExecution : BATCH_JOB_EXECUTION 테이블과 매핑
* JobParameters : BATCH_JOB_EXECUTION_PARAMS 테이블과 매핑
* ExecutionContext : BATCH_JOB_EXECUTION_CONTEXT 테이블과 매핑

* JobInstance의 생성 기준은 JobParameters 중복 여부에 따라 생성
    - 다른 parameter로 Job이 실행되면, JobInstance가 생성
    - 같은 parameter로 Job이 실행되면, 이미 생성된 JobInstance가 실행
    - 이때 Job이 재실행 대상이 아닌 경우 에러가 발생

* 예를 들어
    - 처음 Job실행 시 data parameter가 A로 실행 되었으면, 1번 JobInstance 생성
    - 다음 Job실행 시 data parameter가 B로 실행 되었으면, 2번 JobInstance 생성
    - 다음 Job실행 시 data parameter가 B로 실행 되었으면, 2번 JobInstance 재실행
* Job을 항상 새로운 JobInstance가 실행 될 수 있도록 RunIdIncrementer 제공
    - RunIdIncrementer는 항상 다른 run.id를 Parameter로 설정

* StepExecution : BATCH_STEP_EXECUTION 테이블과 매핑
* ExecutionContext : BATCH_STEP_EXECUTION_CONTEXT 테이블과 매핑

* Job 내에서 공유할 수 있는 BATCH_JOB_EXECUTION_CONTEXT
    - 하위 Step들이 공유하여 사용
* 하나의 Step에서 공유할 수 있는 BATCH_STEP_EXECUTION_CONTEXT
    - 다른 Step과 공유를 하지 못하고 독자적으로 사용

ch03. 스프링 배치 기초 이해하기

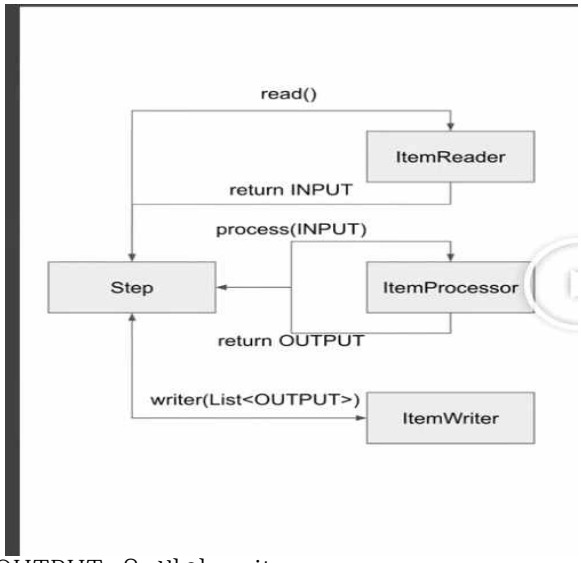
(1) Task 기반 배치와 Chunk 기반 배치

* 배치를 처리할 수 잇는 방법은 크게 2가지

* Tasklet을 사용한 Task 기반 처리
    - 배치 처리 과정이 비교적 쉬운 경우 쉽게 사용
    - 대량 처리를 하는 경우 더 복잡
    - 하나의 큰 덩어리를 여러 덩어리로 나누어 처리하기 부적합

* Chunk를 사용한 chunk(덩어리) 기반 처리
    - ItemReadr, ItemProcessor, ItemWriter의 관계 이해 필요
    - 대량 처리를 하는 경우 Tasklet보다 비교적 쉽게 구현
    - 1000개의 데이터 중 100개씩 10개의 덩어리로 수행
    - 이를 Tasklet으로 처리하면 1000개를 한번에 처리하거나, 수동으로 100개씩 분할

* reader에서 null을 return 할 때 까지 Step은 반복
    - 읽을 데이터가 없을 때 까지
* <INPUT, OUTPUT>chunk(int)
    - reader에서 INPUT을 return
    - processor에서 INPUT을 받아 processing 후 OUTPUT을 return
    - INPUT, OUTPUT은 같은 타입일 수 있음



* writer에서 List<OUTPUT>을 받아 write
* 모든 과정은 하나씩 처리 되며 writer는 chunk.size 만큼 write 또는 processor가 처리후 실행
* 예제 참고 : part3 – ChunkProcessingConfiguration

(2) JobParameters 이해

* 배치를 실행에 필요한 값을 parameter를 통해 외부에서 주입
  - JVM 환경 변수를 통해서 일단 주입한다.
* JobParameters는 외부에서 주입된 parameter를 관리하는 객체
* parameter를 JobParameters와 Spring EL(Expression Language)로 접근
  - String parameter = jobParameters.getString(key,defaultValue);
  - @Value("#{jobParameters[key]}")
* 예제 참고 : part3 – ChunkProcessingConfiguration

● Job에 validator()를 추가한다

(3) @JobScope와 @StepScope 이해

* @Scope는 어떤 시점에 bean을 생성/소멸 시킬지 bean의 lifecycle을 설정
* @JobScope는 job 실행 시점에 생성/소멸
  - Step에 선언
  - @Bean으로 선언되어야 한다.(public)
* @StepScope는 step 실행 시점에 생성/소멸
  - Tasklet, Chunk(ItemReader, ItemProcessor, ItemWriter)에 선언
  - @Bean으로 선언되어야 한다.(public)

* Spring의 @Scope과 같은 것
  - @Scope("job") == @JobScope

- @Scope("step") == @StepScope

* Job과 Step 라이프사이클에 의해 생성되기 때문에 Thread safe하게 작동
    - 여러개의 Job과 Step이 동시에 사용하게 될 경우
    - Bean을 소멸 / 생성하기 때문이다.

* @Value("#{jobParameters[key]}")를 사용하기 위해 @JobScope, @StepScope 필수
* 예제 참고 : part3 - ChunkProcessingConfiguration

(4) ItemReader Interface

* 배치 대상 데이터를 읽이 위한 설정
    - 파일, DB, 네트워크 등에서 읽기 위함
* Step에 ItemReader는 필수
* 기본 제공되는 ItemReader 구현체
    - file, jdbc, jpa, hibernate, kafka, etc,...
* ItemReader 구현체가 없으면 직접 개발
* ItemStream은 ExecutionContext로 read, write 정보를 저장
* 예제 참고 : part3 - CustomItemReader

(5) csv 파일 데이터 읽기

* FlatFileItemReader 클래스로 파일에 저장된 데이터를 읽어 객체에 매핑
* 예제참고 : part3 - ItemReaderConfiguration - csvFileItemReader

(6) JDBC 데이터 읽기

* Cursor 기반 조회
    - 배치 처리가 완료될 때 까지 DB Connection이 연결
    - DB Connection 빈도가 낮아 성능이 좋은 반면, 긴 Connection 유지 시간 필요
    - 하나의 Connection에서 처리되기 때문에, Thread Safe 하지 않음
    - 모든 결과를 메모리에 할당하기 때문에, 더 많은 메모리를 사용
* Paging 기반 조회
    - 페이징 단위로 DB Connection을 연결
    - DB Connection 빈도가 높아 성능이 낮은 반면, 짧은Connection 유지 시간 필요
    - 매번 Connection을 하기 때문에 Thread Safe
    - 페이징 단위의 결과만 메모리에 할당하기 때문에, 비교적 더 적은 메모리를 사용
* JdbcCursorItemReader 예제 참고
    - fetchSize : 한번에 가져올 데이터의 크기

method/종류	JdbcCursorItemReader	JdbcPagingItemReader
datasource	JDBC를 실행하기 위한 Datasource	
beanMapper | rowMapper	조회된 데이터 row를 클래스와 매핑하기 위한 설정	
sql	조회 쿼리 설정	x
selectClause, fromClause whereClause, queryProvider	x	조회 쿼리 설정
fetchSize	cursor에서 fetch될 size	JdbcTemplate.fetchSize
pageSize	x	paging에 사용될 page 크기 (offset / limit)

* 참고 예시
    part3 - ItemReaderConfiguration - jdbcCursorItemReader()
    part4 - UserConfiguration - orderStatisticsItemReader()

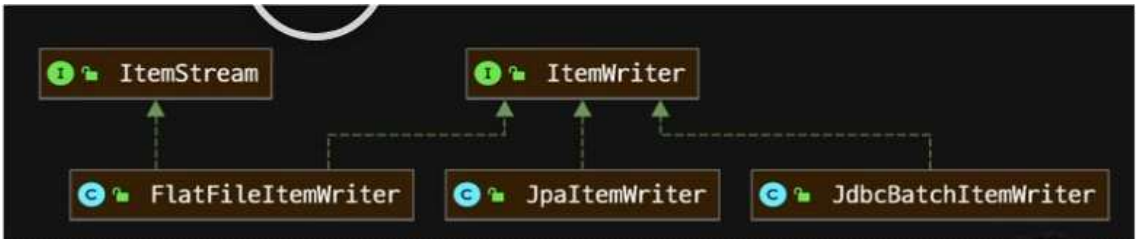
(7) JPA 데이터 읽기

* Spring 4.3+에서 JPA 기반 Cursor ItemReader가 제공됨
* 기존 JPA는 Paging 기반의 ItemReader만 제공됨

Method / 종류	JpaCursorItemReader	JpaPagingItemReader
entityManagerFactory	JPA를 실행하기 위해 EntityManager를 생성하기 위함	
	조회된 데이터 row를 클래스와 매핑하기 위한 설정	
queryString	조회 쿼리	x
selectClause, fromClause whereClause	x	조회 쿼리
pageSizse	x	paging에 사용될 page 크기 (offset / limit)

* 참고 예시
    part3 - ItemReaderConfiguration - jpaCursorItemReader()
    part4 - UserConfiguration - itemReader()

(8) ItemWriter Interface 구조 이해



* ItemWriter는 마지막으로 배치 처리 대상 데이터를 어떻게 처리할지 결정
* Step에서 ItemWriter는 필수
* 예를 들면 ItemReader에서 읽은 데이터를
  - DB에 저장, API로 서버에 요청, 파일에 데이터를 write
* 항상 write가 아님
  - 데이터를 최종 마무리 하는 것이 ItemWriter

(9) CSV 파일 데이터 쓰기

* FlatFileItemWriter는 데이터가 매핑된 객체를 파일로 write
* 데이터 생성은 chunk가 반복될때마다 생성되는 것이 아닌 step이 끝날 때 메모리에 저장된 데이터로 파일을 생성
* 예제 참고 : part3 - ItemWriterConfiguration - csvFileItemReader

(10) JDBC 데이터 쓰기

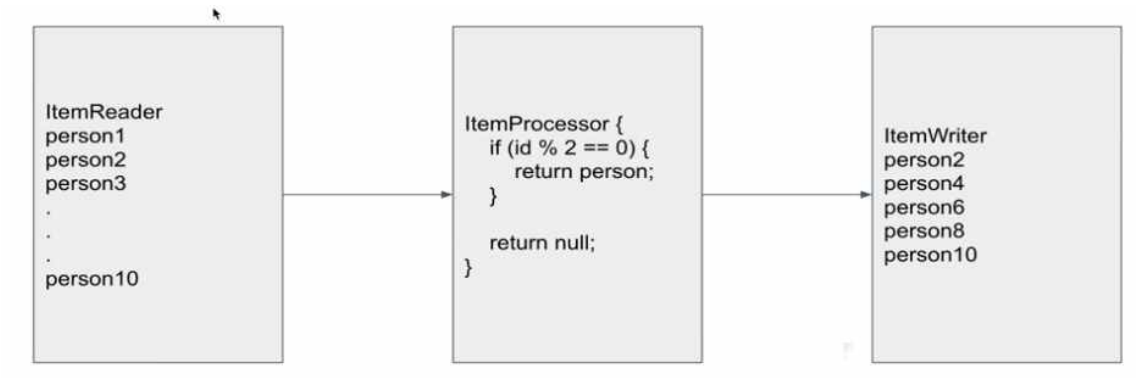
* JdbcBatchItemWriter는 jdbc를 사용해 DB에 write
* JdbcBatchItemWriter는 bulk insert/update/delete 처리
  - insert into person (name,age,address) values (1,2,3), (4,5,6), (7,8,9)
* 단건 처리가 아니기 때문에 비교적 높은 성능
* 예제 참고 : part3 - ItemWriterConfiguration - jdbcBatchItemReader

(11) JPA 데이터 쓰기

* JpaItemWriter는 JPA Entity 기반으로 데이터를 DB에 write
* Entity를 하나씩 EntityManager.persist 또는 EntityManager.merge로 insert
* 예제 참고
  part3 - ItemWriterConfiguration - jpaItemWriter
  part4 - UserConfiguration - itemReader()
* EntityManager Life Cycle : JPA강의 - Transaction - EntityManager.txt 참고

(12) ItemProcessor Interface

* ItemReader에서 읽은 데이터를 가공 또는 Filtering
* Step의 ItemProcessor는 Optional
* ItemProcessor는 필수는 아니지만, 책임 분리를 분리하기 위해 사용
* ItemProcessor는 Input을 Output으로 변환하거나
* ItemWriter의 실행 여부를 판단 할 수 있도록 filtering 역할을 한다
    - ItemWriter는 not null만 처리한다.



* 예를 들어 persion.id가 짝수인 person만 return 하는 경우
* ItemWriter는 5개의 person만 받아 처리
* 예제 참고 : part3 - ItemConfiguration - itemProcessor

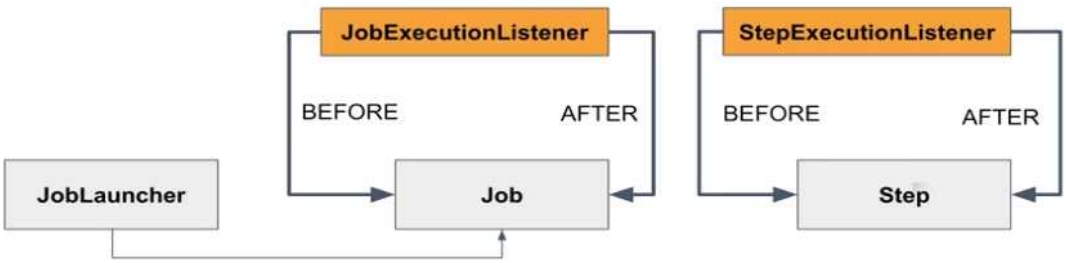
(13) 과제 요구 사항

* CSV 파일 데이터를 읽어 H2 DB에 데이터 저장하는 배치 개발
* Reader
    - 100개의 person data를 csv 파일에서 읽는다
* Processor
    - allow_duplicate 파라미터로 person.name의 중복 여부 조건을 판단한다.
    - 'allow_duplicate=true' 인 경우 모든 person을 return 한다.
    - 'allow_duplicate=false' 인 경우 person.name이 중복된 데이터는 return null
    - 힌트 : 증복 체크는 'java.util.Map' 사용
* Writer
    - 2개의 ItemWriter를 사용해서 Person H2 DB에 저장 후 몇 건 저장인지 log
    - Person 저장 ItemWriter와 log 출력 ItemWriter
    - 힌트 : 'CompositeItemWriter' 사용 --> ItemWriter를 묶어서 사용

(14) 테스트 코드 작성하기

* JobLauncher는 Job을 실행
* JobLauncherTestUtils은 테스트 코드에서 Job과 Step 실행
* 예제 참고 : test - part3 - SubjectConfigurationTest

(15) JobExecutionListener, StepExecutionListener



* 스프링 배치에서 전 처리, 후 처지를 하는 다양한 종류의 Listener 존재
    - Interface 구현
    - @Annotation 정의
* Job 실행 전과 후의 실행할 수 있는 JobExecutionListener
    - 2개 이상의 listener 등록 시 순서대로 1,2를 등록하면
    - before 순서 : 1,2
    - after 순서 : 2,1
* Step 실행 전과 후의 실행할 수 있는 StepExecutionListener
    - before 순서 : JobListener, StepListenr
    - after 순서 : StepListenr, JobListener
* 예제 참고 :
    part3 - subjectConfiguration, subjectConfigurationTest, SavePersonListener

(16) StepListener 이해

* Step에 관련된 모든 Listener는 StepListener를 상속

* StepExecutionListener
    - 위의 (15) 참고

* SkipListener
    - onSkipInRead : @OnSkipInRead
        ItemReader에서 Skip이 발생한 경우 호출
    - onSkipInWrite : @OnSkipInWrite
        ItemWriter에서 Skip이 발생한 경우 호출
    - onSkipInProcess : @OnSkipInProcess
        ItemProcessor에서 Skip이 발생한 경우 호출

* ItemReadListener
    - beforeRead : @BeforeRead
        ItemReader.read() 메소드 호출 전 호출
    - afterRead : @AfterRead
        ItemReader.read() 메소드 호출 후 호출
    - onReadError : @OnReadError
        ItemReader.read() 메소드에서 에러 발생 시 호출

* ItemProcessListener
    - beforeProcess : @BeforeProcess
        ItemProcess.process() 메소드 호출 전 호출
    - afterProcess : @AfterProcess
        ItemProcess.process() 메소드 호출 후 호출
    - onProcessError : @OnProcessError
        ItemProcess.process() 메소드에서 에러 발생 시 호출

* ItemWriteListener
    - beforeWrite : @BeforeWrite
        ItemWriter.write() 메소드 호출 전 호출
    - afterWrite : @AfterWrite
        ItemWriter.write() 메소드 호출 후 호출
    - onWriteError : @OnWriteError
        ItemWriter.write() 메소드에서 에러 발생 시 호출

* ChunkListener
    - chunk size = 10, itemSize = 100 ---> chunk는 10번 실행
    - beforeChunk : @BeforeChunk
        chunk 실행 전 호출
    - afterChunk : @AfterChunk
        chunk 실행 후 호출
    - afterChunkError : @BfterChunkError
        chunk 실행 중 에러 발생시 호출

(17) skip 예외 처리

* step 수행 중 발생한 특정 Exception과 에러 횟수 설정으로 예외 처리 설정

* skip(NotFoundNameException.class), skipLimit(3) 으로 설정된 경우
    - NotFoundNameException Step 내에서 발생 3번까지는 에러를 skip 한다.
    - NotFoundNameExcetion 발생 4번째는 Job과 Step의 상태는 실패로 끝나며 배치가 중지된다.
    - 단, 에러가 발생하기 전까지 데이터는 모두 처리된 상태로 남는다.
    - 항상 faultTolerant() 메소드 후에 선언

* Step은 chunk 1개 기준으로 Transaction 동작
    - items = 100, chunk.size = 10, 총 chunk 동작 횟수 = 10
    - chunk 1-9는 정상 처리, chunk 10에서 Exception이 발생한 경우
    - chunk 1-9에서 처리된 데이터는 정상 저장되고, Job과 Step의 상태는 FAILED

* 배치 재 실행 시 chunk 10부터 처리할 수 있도록 배치를 만든다.

* SubjectConfiguration 추가 요구사항
    - Person.name이 empty String인 경우 NotFoundNameException 발생
    - NotFoundNameException이 3번 이상 발생한 경우 step 실패 처리

* SkipListener가 실행 되는 조건
    - 에러 발생 횟수가 skip Limit이하인 경우
    - skipLimit(2), throw Exception이 3번 발생하면 실행되지 않는다.
    - skipLimit(3), throw Exception이 3번 발생하면 실행된다.
    - skip 설정 조건에 해당하는 경우에만 실행된다.
    - SkipListener는 항상 faultTolerant() 메소드 바로 다음 줄에 선언

* 예제 참고 : part3 - SubjectConfiguration - subjectStep(), itemProcessor()

(18-1) retry 예외처리

* Step 수행 중 간헐적으로 Exception 발생 시 재시도(retry) 설정
  - DB Deadlock, Network timeout등

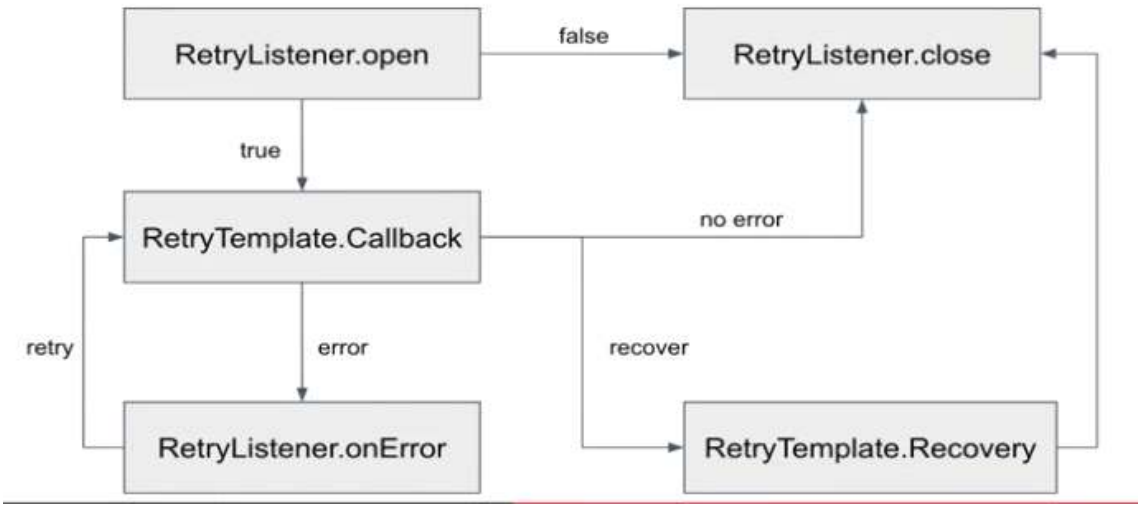
* retry(NullPointerException.class, retryLimit(3) 으로 설정된 경우
  - NotFoundNameException이 발생한 경우 한 대상에 대해 3번까지 재시도
  - 성공할 가능성이 있는 경우 사용
  - 항상 faultTolerant() 메소드 후에 선언

* 더 구체적으로 retry를 정의하려면 RetryTemplate 이용

* SubjectConfiguration 추가 요구사항
  - NotFoundNameException이 발생하면, 3번 재 시도 후
  - Person.name을 "UNKNOWN" 으로 변경

* 예제 참고
  part3 - SubjectConfiguration - subjectStep(), itemProcessor()
  part3 - PersonValidationRetryProcessor

(18-2) retry Listener



1. RetryListener.open() : return false인 경우 retry를 시도 하지 않음
2. RetryTemplate.RetryCallback
3. RetryListener.onError() : maxAttempts 설정값 만큼 반복
4 RetryTemplate.RecoveryCallback : maxAttempts 반복 후에도 에러가 발생한 경우 실행
5. RetryListener.close

ch04. 회원 등급 프로젝트

* User 등급을 4개로 구분
    - 일반(NORMAL), 실버(SILVER), 골드(GOLD), VIP

* User 등급 상향 조건은 총 주문 금액 기준으로 등급 상향
    - 200000원 이상인 경우 실버로 상향
    - 300000원 이상인 경우 골드로 상향
    - 500000원 이상인 경우 VIP로 상향
    - 등급 하향은 없음

* 총 2개의 Step으로 회원 등급 Job 생성
    - saveUserStep : User 데이터 저장
    - userLevelUpStep : User 등급 상향

* JobExecutionListener.afterJob 메소드에서
    - “총 데이터 처리 {}건, 처리 시간 : {}millis” 와 같은 로그 출력

ch05. 주문금액 집계 프로젝트

(1) 요구사항 이해

* User의 totalAmout를 OrdersEntity로 변경
    - 하나의 User는 Nrodml Orders를 포함
* 주문 총 금액은 Orders Entity를 기준으로 합산
* ‘-date=2020-11’ JobParameters 사용
    - 주문 음액 집계는 orderStatisticsStep으로 생성
    - 2020년.csv 파일은 2020.11.1.~2020.11.30. 주문 통계 내역
* ‘date’ 파라미터가 없는 경우, orderStatisticsStep은 실행하지 않는다.

* part4 project와 연계에서 작성

(4) JobExcutionDecider

* Batch 상태에서 Job의 실행여부를 결정
* 예제 참고
    part5 - JobParametersDecide
    part4 - UserConfiguration - userJob()

ch06. 성능 개선과 성능 비교

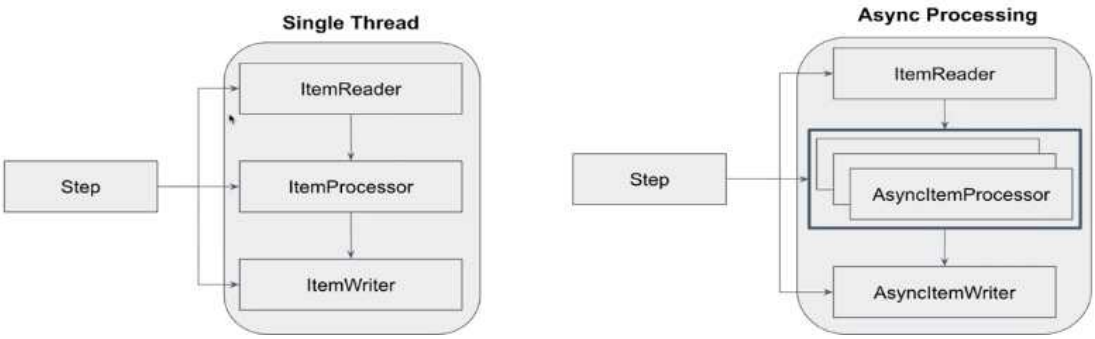
(1) 성능 개선 계획 이해

* part4 - part5 연계에서 작성한 프로젝트의 성능 개선
* SaveUserTasklet에서 User 40,000건 저장, Chunk Size는 1,000
* 아래 표 순서대로 실행
    - 예제를 만들고 성능 측정 후 비교, 3번씩 실행, 환경에 따라 성능이 다를 수 있음

* Step마다 역할이 모두 다르다

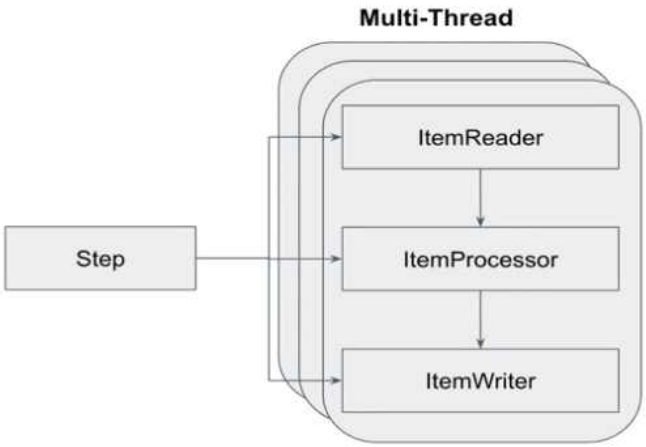
단위 : millis	1회	2회	3회
Simple Step(기존)	12365	11890	12037
Async Step(비동기)	12136	12220	12004
Multi-Thread Step	7380	7743	7489
Partition Step	8967	8225	8038
Async + Partation Step	7938	8599	8517
Parallel Step	12190	11432	12006
Partition + Parallel Step	8123	8844	7969

(2) Async Step 적용하기



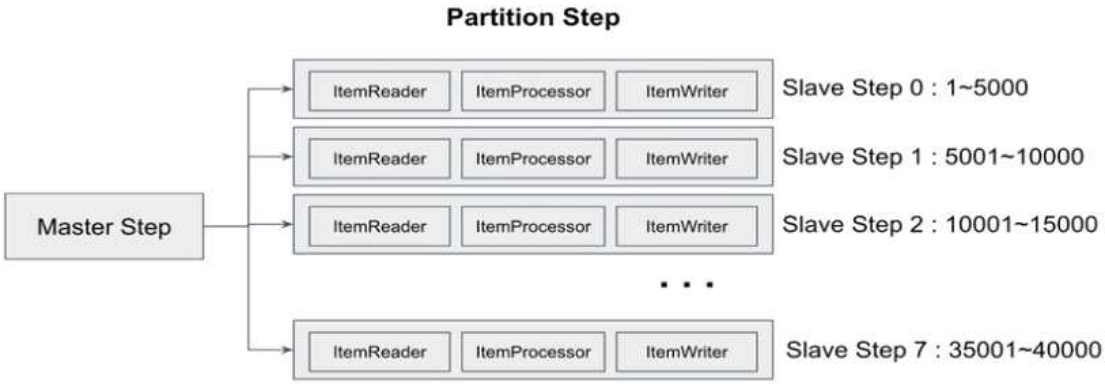
* ItemProcessor와 ItemWritter를 Async로 실행
* Simple : Writer는 Processor의 작업 종료를 기다려야 하고 새로운 chunk를 실행하기 위해서는 Writer의 작업 종료를 기다려야 한다.
* Async : 각각 작업종료를 기다리지 않고 개인의 작업이 종료되면 새로운 작업을 실행한다.
* java.util.concurrent에서 제공하는 Future 기반 asynchronous
* Async를 사용하기 위해 spring-batch-integration 필요
* 예제 참고 :
  SpringBatchExampleApplication
  part6 - AsyncUserConfiguration
  itemWriter() & itemProcesr() & userLevelUpStep()
  Part4 - User : Fetch Type 변경(LAZY -> EAGER)

(3) Multi-Thread Step



* Async Step은 ItemProcessor와 ItemWriter 기준으로 비동기 처리
* Multi-Thread Step은 Chunk 단위로 멀티 스레딩 처리
* Thread-Safe 한 ItemReader 필수
* 예제 참고 :.part6 - MultiThreadUserConfiguration - userLevelUpStep()

(4) Partition Step



* 하나의 Master 기준으로 여러 Slave Step 기준으로 Multi-Thread 처리
* 예를 들어
  - item이 40,000개, Slave Step이 8개면
  - 40000 / 8 = 5000 이므로 하나의 Slave Step 당 5,000건 씩 나눠서 처리

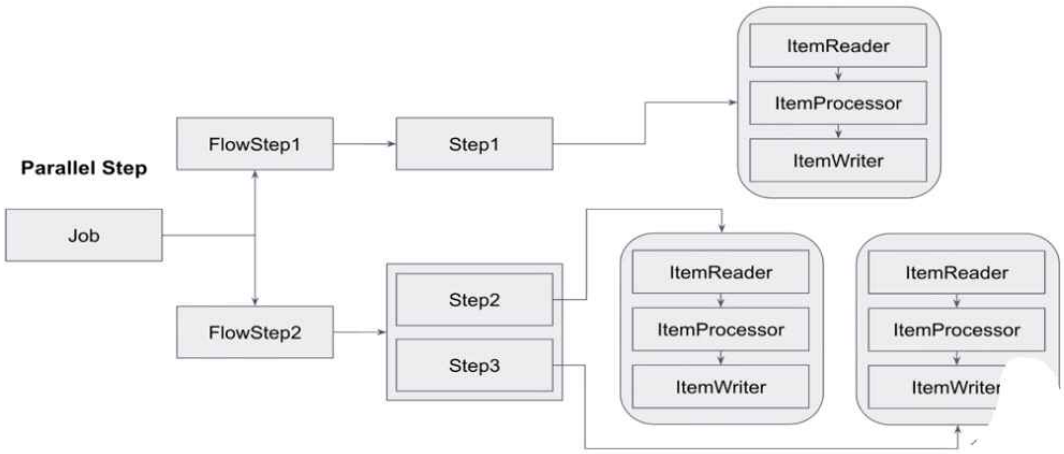
* Slave Step은 각각 하나의 Step으로 동작
  - 순차적 실행이 아닌 동시에 실행이 된다.

* 예제 참고

part6 - UserLevelUpPartitioner : 생성
part4 - UserRepository : 쿼리 추가

part6 - PartitioneUserCongiguration
userLevelUpStep() : itemReader() -> itemReader(null, null)
itemReader() : 어노테이션 추가, @StepScope사용 시 return 되는 클래스 타입을 정확히 명시해야 한다. proxy때문에
userLevelUpManagerStep() : 생성
taskExecutorPartitionHandler() : 생성
userJob() : userLevelUpStep -> userLevelUpManagerStep

(5) Parallel Step



* n개의 Thread가 Step 단위로 동시 실행(Flow내의 step이 동시실행)
* Muli-Thread Step은 chunk 단위로 동시 실행, Parallel Step은 step단위로 동시 실행
* 위의 그림을 예로 들면
    - Job은 FlowStep1과 FlowStep2를 순차 실행
    - FlowStep2는 Step2와 Step3을 동시 실행
    - 설정에 따라 FlowStep1과 FlowStep2를 동시 실행도 가능
* Flow로 합치려면 각각의 step이 flow야 합칠 수 있다.

* 예제 참고

part6 - ParallelUserConfiguration
saveUserFlow() : saveUserStep() 옮김,
saveUserStep() : 삭제
userJob() : step 정보 변경
orderStatisticsStep() : 어노테이션 제거
ordersStatisticsFlow() : 생성
splitFlow() : 생성

ch07. 스프링 배치 설정과 실행

(1) jar 생성과 실행

* 정확히 어떤 위치에 파일을 생성해야 하는지 명확해야 한다.
    UserConfiguration - orderStatisticsStep() : 파라미터 추가
    UserConfiguration - orderStatisticsItemWriter()

* 프로젝트 내에 gradlew 로 jar 생성
    - ./gradlew clean jar build -x test

* build/libs 내에 jar 생성
    java -jar sample.jar —job.name=userJob -date=2020-11 -path=/Users/

(2) Jenkins Scheduler를 이용한 스프링 배치 실행

* Docker에 Jenkins를 설치힌다.(Web Browser로 사용 가능)
* GitHub Repository를 만든 후 프로젝트를 등록한다.
* Jenkins -> 새로운 item -> Jenkins Job 생성 -> userJobTest -> Freestyle Project
    - project에 git repository 정보도 기입한다.
    - build에 jar 생성 실행 명령어 입력
        이때 path는 Docker의 path를 입력한다.