배치 개발가이드

Version 1.0

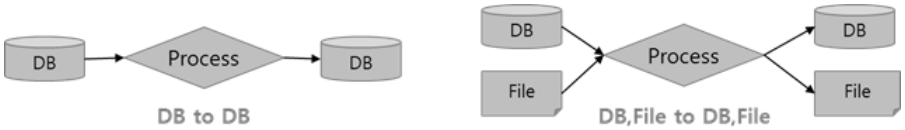
차례

1장. 개요

본 문서는 배치 어플리케이션 개발 시 개발 생산성 향상 및 운영의 효율화를 위해 반드시 준수되어야 하는 사항들을 정의해 놓은 문서이다.

1.1 배치 개요

배치 작업이란 DB 또는 파일에서 대량의 데이터를 읽어 일괄 처리하고 처리 결과를 다시 DB 또는 파일로 일괄 반영하는 작업을 의미한다.



일반적으로 온라인 어플리케이션은 사용자가 요청한 단 건 데이터를 즉시 처리하는데 최적화된 구조를 가진 사용자 중심의 처리 방식이라고 할 수 있다. 반면 배치 어플리케이션은 일정 기간 또는 일정량의 데이터를 축적했다가 시스템 또는 운영자에 의해 시작되어 대량의 데이터를 일괄해서 연속적으로 처리하는데 최적화된 구조를 가진다.

배치는 입출력 데이터 유형과 입출력 데이터의 개수에 따라 다음과 같이 구분한다.

Table 1.1. 입출력 데이터 유형에 따른 구분

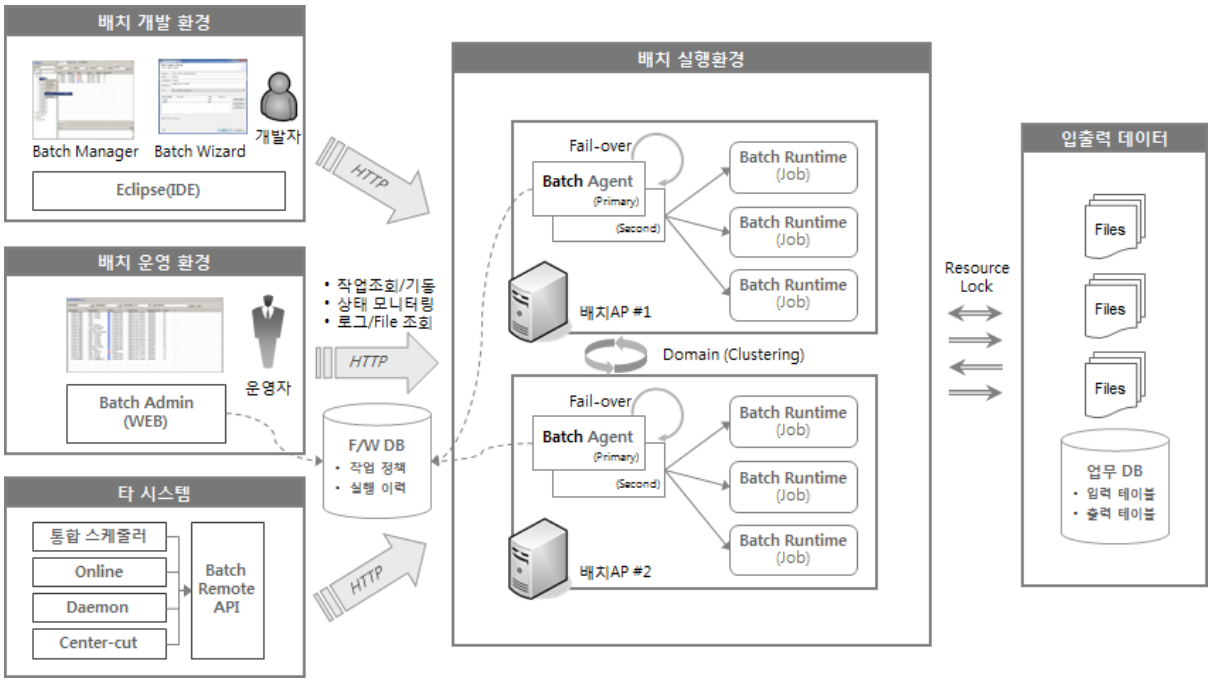
유형	입력	출력	설명	비고
DB2DB	DB	DB	DB에서 읽어서 결과를 DB에 기록	
DB2FILE	DB	FILE	DB에서 읽어서 결과를 FILE에 기록	Unload
FILE2DB	FILE	DB	FILE에서 읽어서 결과를 DB에 기록	Load
FILE2FILE	FILE	FILE	FILE에서 읽어서 결과를 FILE에 기록	Match, Sort, Filter

Table 1.2. 입출력 데이터 개수에 따른 구분

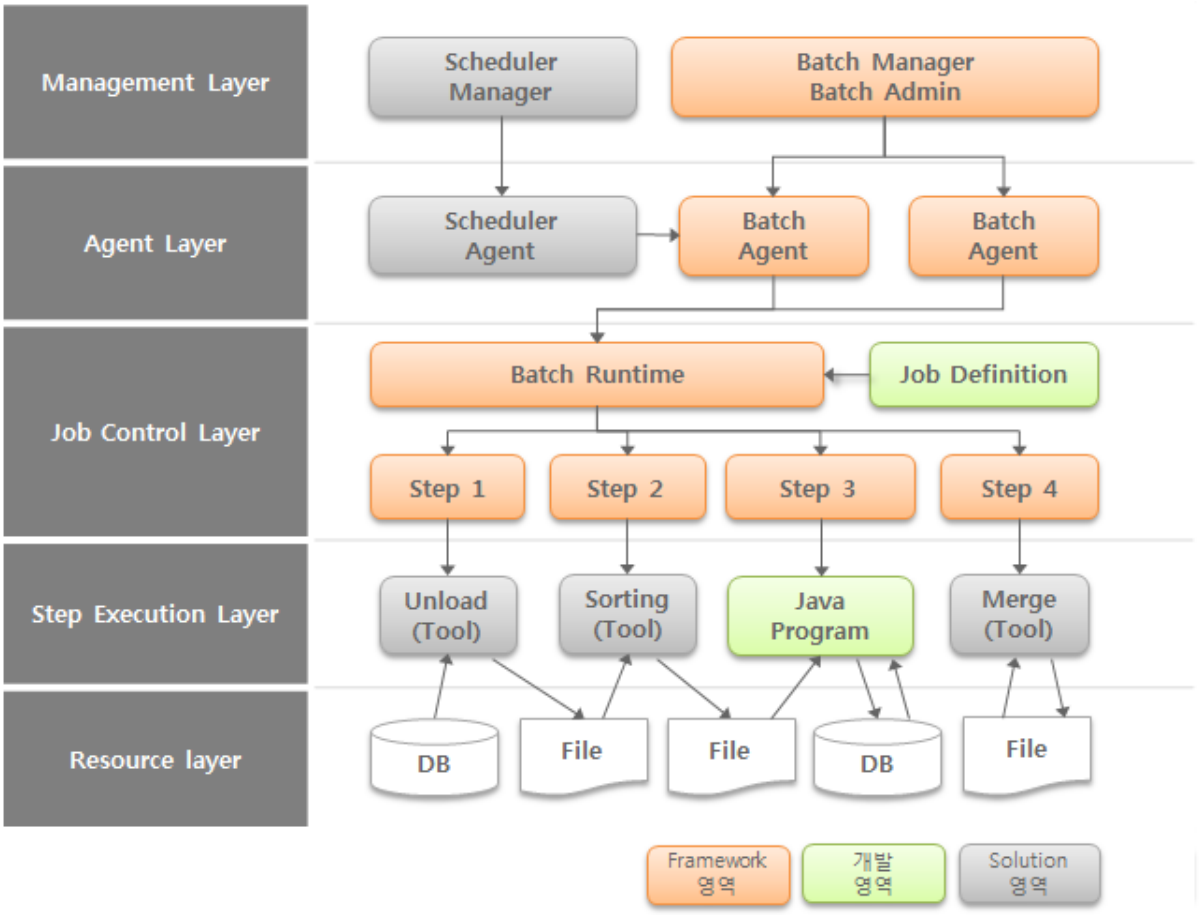
유형	입력 개수	출력 개수	설명	비고
1:1	1	1	하나의 입력 데이터를 읽어 하나의 출력 데이터로 기록	Transform
1:N	1	N	하나의 입력 데이터를 읽어 N개의 출력 데이터로 기록	Split
N:1	N	1	N개의 입력 데이터를 읽어 1개의 출력 데이터로 기록	Merge

1.2 아키텍처

배치 프레임워크는 크게 개발환경(IDE), 운영환경(Admin), 실행환경(Runtime)으로 구분한다. 고 가용성을 위한 클러스터링 기반의 아키텍처로 다수의 분산 서버에서 수행되는 작업을 중앙에서 관제할 수 있는 환경을 제공한다.



배치 아키텍처는 관리 제어에서 실행에 필요한 리소스까지의 각 영역을 5개의 Layer로 구성되며, 이들 레이어는 각기 중앙 관제, 원격 제어, 작업 실행, 스텝 실행, 입출력 역할을 수행한다.

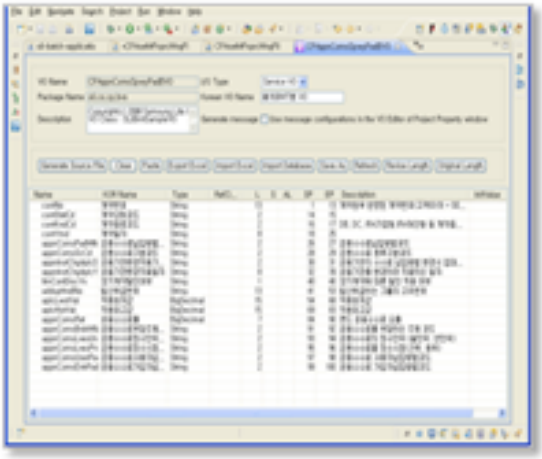


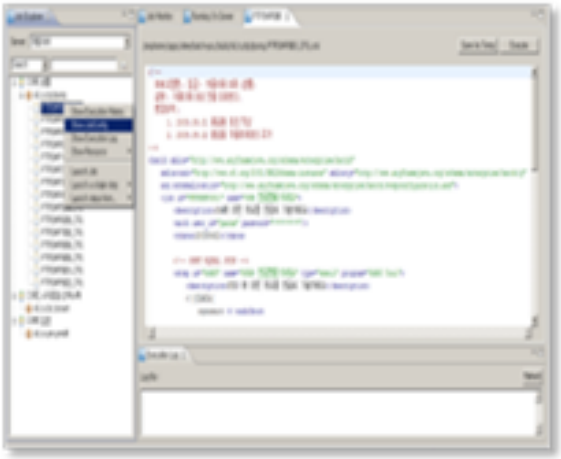
- **Management Layer**
  - 분산 환경에서의 중앙 통제 기능을 담당

    - 작업 정책 정의, 실행 통제, 상태 조회
- **Agent Layer**
    - 각 배치 실행 서버에서 작업 기동, 제어 및 상태 모니터링을 수행
    - 작업 정책에 따른 실행 제한을 처리
- **Job Control Layer**
    - 배치 작업정의(CFG)에 포함된 스텝을 수행 시키는 역할을 담당
    - 스텝의 순차/병렬처리 및 조건부 분기, 작업 재시작 등을 처리함
- **Step Execution Layer**
    - 비즈니스 로직을 구현한 프로그램 및 툴을 실행하여 단위 스텝을 실행하는 영역
    - IO 맵핑, 스텝 재시작 등을 처리함
- **Resource Layer**
    - 배치 작업에 필요한 입출력 데이터 영역
    - 파일(SAM, VSAM) 및 데이터베이스 등

1.2.1 개발환경

Java Eclipse 환경 하에서 배치 어플리케이션을 설계, 개발, 테스트 할 수 있도록 GUI 기반의 개발도구를 제공한다. 시각적인 비즈니스 로직 설계, 템플릿 기반의 코드 자동 생성 기능 등을 지원하여 개발 생산성 향상 및 향후 원활한 유지보수를 지원한다.





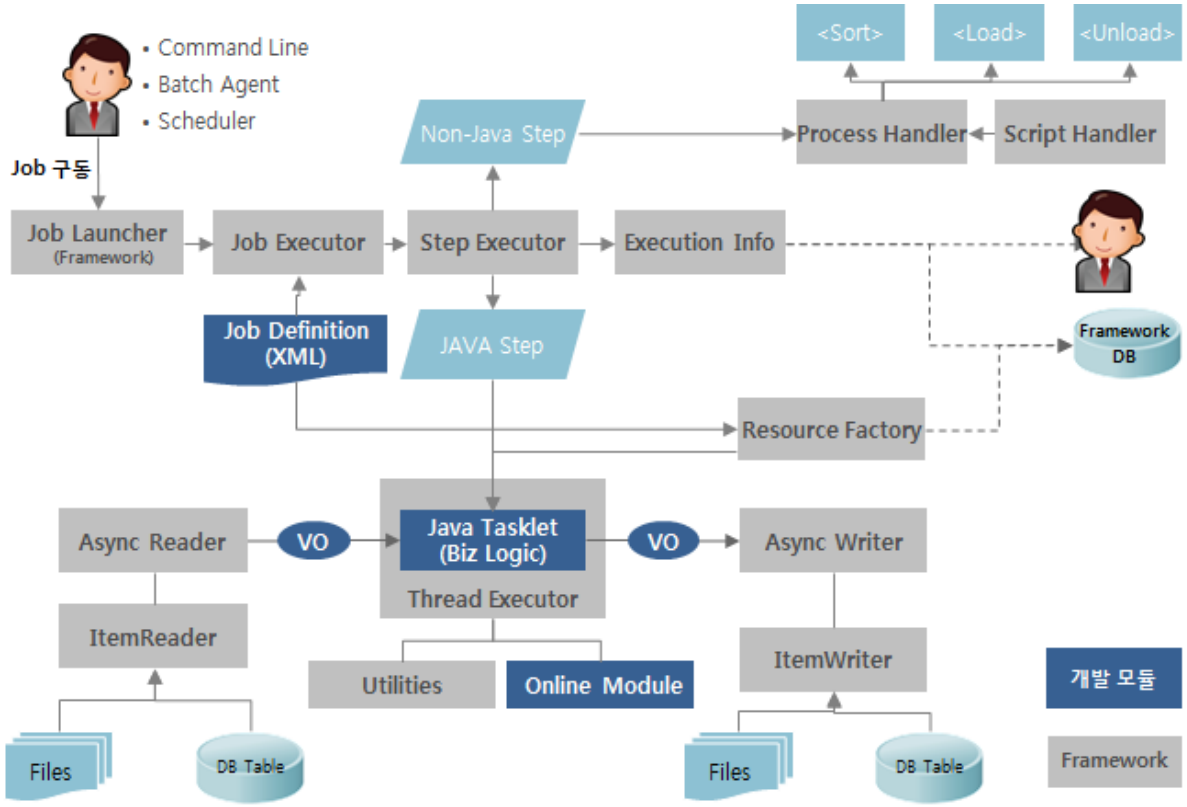
배치 개발환경(IDE)에서는 다음과 같은 기능을 제공한다.

- **VO Editor**
    - 입출력 데이터 맵핑 객체(VO) 생성 및 편집
    - 용어사전과 연계하여 용어 표준화
- **Batch Manager**
    - 개발 서버의 배치 작업 조회
    - 작업 실행 제어 및 실시간 모니터링

- 작업 결과 및 로그 조회
- 입출력 파일 데이터 조회

1.2.2 실행환경

배치 실행 환경(Runtime)은 배치 어플리케이션 개발에 필요한 공통 기능을 제공하고 배치 작업에 필요한 다양한 입출력 데이터에 대한 초기화 및 데이터 변환을 자동으로 수행한다. 또한 개발된 배치 어플리케이션 실행 시 최적화된 성능과 안정성을 보장한다.



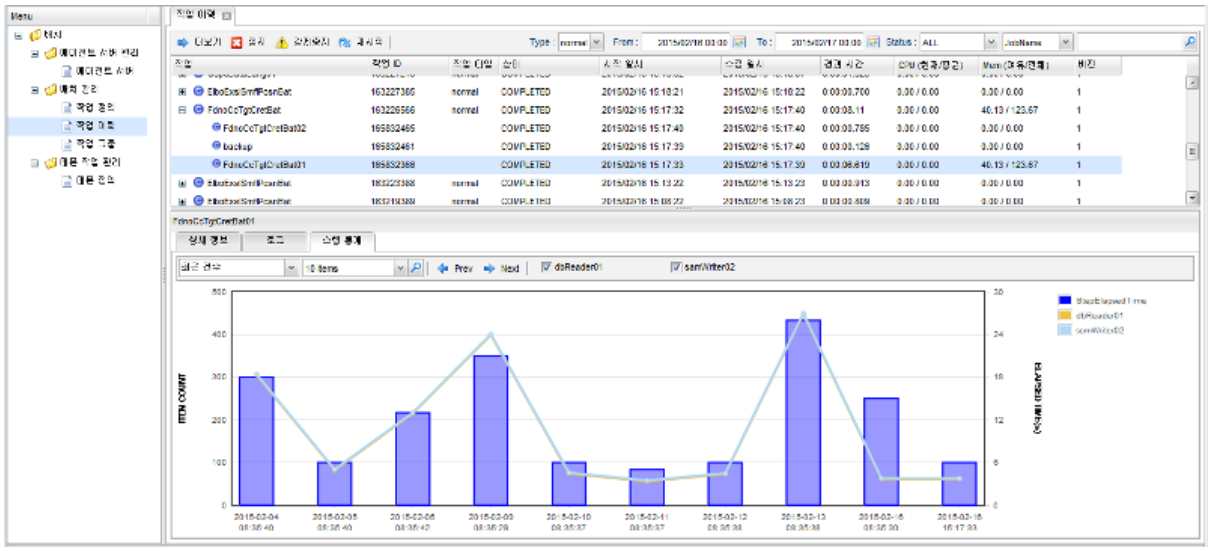
배치 실행환경(Runtime)은 다음과 같은 기능을 제공한다.

- XML 기반의 직관적인 작업 정의
- 일반 배치, 데몬 배치, 온라인 배치 등 다양한 실행 유형 제공
- 다양한 입출력 데이터를 처리할 수 있는 Reader/Writer를 기본 제공
- 데이터 ↔ Value 객체 간 자동 맵핑
- File/DB 초기화, 종료 처리 및 트랜잭션 자동 처리

1.2.3 운영환경

배치 운영 환경(Admin)은 웹 기반으로 중앙에서 작업 관제를 할 수 있는 관리 어플리케이션과 배치 서버에서 작업의 배치 작업 기동 및 실행 상태를 모니터링 하는 배치 에이전트(Agent)로 구성된다.

배치 운영환경은 안정적인 시스템 운영을 위한 작업 실행 정책과 용량 제어 기능을 제공하고 분산 환경에서의 고 가용성을 보장하기 위하여 Fail-over가 가능한 클러스터링 아키텍처로 구성되어 있다.



배치 운영 환경(Admin)은 다음과 같은 기능을 제공한다.

- **작업 관리**
  - 배치 작업 및 실행 파라메터 등록
  - 작업 실행 정책 등록 (동시 실행 여부, 실행 허용 시간 등)
- **작업 실행 제어 및 모니터링**
  - 원격 서버의 작업 실행, 중지 및 재시작
  - 작업 실행상태 실시간 모니터링 (처리건수, CPU, 메모리 사용량 등)
  - 시스템 및 작업 실행 오류 시 통지
- **실행 정책**
  - 서버 당 최대 실행 작업 수 제한 (초과 작업은 대기 후 실행)
  - 작업 실행 제한 및 접근 권한
  - 시스템 안정성 확보를 위한 파일크기, 메모리 사용량 제한

2장. 배치 작업 정의

2.1 작업 및 스텝 정의

배치 작업(Job)은 비즈니스 관점에서 처리해야 하는 단위 업무(Goal)로 하나 이상의 스텝(Step)으로 구성되며 지정된 시간(스케줄링) 또는 관리자에 의해 실행되는 단위가 된다. 각각의 배치 작업은 XML 형태의 작업정의 파일로 구성하며 다음과 같은 특성을 갖는다.

- 하위의 스텝(Step)을 순차적으로 실행하며, 스텝 실행 중에 에러가 발생하면 이후 스텝은 실행하지 않으며 작업은 실패(Fail)
- 모든 스텝이 정상 처리 되면 작업은 완료(Complete)
- 작업정의 파일의 파일명은 작업의 고유 식별자(ID)로 사용

배치 스텝(Step)은 배치 작업을 구성하는 요소로 배치 프로그램 실행, 쉘 명령어 실행 등 비즈니스 요건에 따라 구성된다. 배치 스텝은 다음과 같은 특성을 갖는다.

- 단위 비즈니스를 처리하는 단위로 기본적으로 트랜잭션 단위 (Step 시작 시 트랜잭션 시작, 완료 시 Commit, 에러 시 Roll-back 처리)
- 일반적인 배치 프로그램은 Loop 문 내에서 데이터 입력(Read)-> 가공(Process) -> 저장(Write) 형태로 구성됨

스텝은 다음과 같은 유형을 갖는다.

Table 2.1. 배치 스텝 유형

유형	설명
java	지정된 Java 프로그램(Class)를 실행하여 배치 업무를 수행 한다
parallel	하위의 스텝을 병렬로 실행시킨다 (하위 스텝 각각은 별도의 트랜잭션 단위로 처리됨)
delete	지정된 파일을 삭제한다
shell	shell script를 실행하여 작업을 처리한다

[작업정의(CFG) 예]

```
<job concurrent="true" restartable="true">

    <description>샘플 배치 작업</description>

    <step id="parallel-group" type="parallel">

        <step id="process-sam" type="java" class="sample.batch.SampleBat">
            <description>SampleBat를 실행</description>
            <resources>
                <reader id="reader" type="SAM" url="${BASE_DIR}/sample1.txt" />
                <writer id="writer" type="SAM" url="${BASE_DIR}/sample1.txt.out" />
            </resources>
        </step>

        <step id="process-vsam" type="java" class="sample.batch.SampleBat">
            <description>SampleBat를 실행</description>
            <resources>
                <reader id="reader" type="VSAM" url="${BASE_DIR}/sample2.txt" />
                <writer id="writer" type="VSAM" url="${BASE_DIR}/sample2.txt.out" />
            </resources>
        </step>
```

```
    </step>

    <step id="say-hello" type="shell" restartable="false">
        <script>
            <![CDATA[
            echo hello world
            exit 0
            ]]>
        </script>
    </step>

    <step id="clear-file" type="delete">
        <description>resource에 선언된 파일을 삭제</description>
        <resources>
            <resource url="${BASE_DIR}/sample*.txt" />
            <resource url="${BASE_DIR}/sample*.txt.out" />
        </resources>
    </step>
</job>
```

2.1.1 작업 및 스텝 속성

배치 작업 빛 스텝을 정의하는데 사용되는 XML 엘리먼트(Element)와 속성(Attribute)에 대한 상세 내역은 다음과 같다.

Table 2.2. 작업 및 스텝 속성

엘리먼트	속성	설명	기본값
<job>		작업 엘리먼트	
	@name	작업명	
	@concurrent	동시실행 허용 여부	false
	@restartable	작업 재시작 가능 여부	false
<step>		스텝 엘리먼트	
	@id	스텝 고유 ID	필수
	@type	스텝 유형	필수
	@class	배치 프로그램 클래스	
	@restartable	스텝 재시작 가능 여부	false
	@query-file	쿼리파일 위치	
	@on-ok	스텝 성공 이후 다음 스텝ID	
	@on-error	스텝 실패 이후 다음 스텝ID	
	@thread-size	병렬스텝(parallel)의 동시 실행 수	-1
<parameters>		parameter의 묶음	
<parameter>		작업/스텝 파라메터	
	@key	파라메터 키	
	@value	파라메터 값	
<description>		작업/스텝에 대한 부가 설명	
<script>		실행시킬 쉘 스크립트	

2.1.2 Java 스텝

지정된 Java 프로그램(Class)를 실행하여 배치 업무를 수행 한다. Java 프로그램은 비즈니스 로직을 구현하여 데이터를 가공한 후 Reader/Writer를 활용하여 입출력 데이터를 처리한다. (Section 3.3, “배치 프로그램(Task)” 참조)

- @type을 "java"로 지정
- @class에 Java 프로그램을 지정
- 프로그램 내에서 오류(Exception) 발생 시 해당 스텝은 실패(Fail)로 처리되며 오류 없이 정상적으로 Return 되면 완료(Complete)로 처리

[Java 스텝 예]

```
<job>

    <description>샘플 Java 스텝</description>

    <step id="step1" type="java" class="sample.batch.SampleBat">
        <description>SampleBat를 실행</description>
        <resources>
            <reader id="reader" type="SAM" url="${BASE_DIR}/sample1.txt" />
            <writer id="writer" type="SAM" url="${BASE_DIR}/sample1.txt.out" />
        </resources>
    </step>

    <step id="step2" type="java" class="sample.batch.SampleBat">
        <description>SampleBat를 실행</description>
        <resources>
            <reader id="reader" type="SAM" url="${BASE_DIR}/sample2.txt" />
            <writer id="writer" type="SAM" url="${BASE_DIR}/sample2.txt.out" />
        </resources>
    </step>

</job>
```

2.1.3 Parallel 스텝

여러 개의 스텝을 동시에 처리하기 위해 병렬스텝(Parallel)으로 묶어 하위에 선언된 스텝을 동시에 실행시킬 수 있다. 병렬 스텝으로 묶인 대상은 동시에 수행되기 때문에, 실행순서는 순차적이지 않으며 병렬 스텝으로 묶인 하위의 스텝이 모두 종료되어야 다음 스텝으로 진행된다.

- @type을 "parallel"로 지정
- parall step 하위에 동시 실행시킬 step을 정의
- @thread-size에 동시 실행 수를 지정 (지정하지 않으면 하위 스텝을 모두 동시 실행)

[Parallel 스텝 예]

```
<job>

    <description>샘플 Parallel 스텝</description>

    <step id="group1" type="parallel">
     <!-- 하위 step을 모두 동시에 실행시킨다 -->
        <step id="step1" type="java" class="sample.batch.Hello"/>
        <step id="step2" type="java" class="sample.batch.Hello"/>
    </step>

    <step id="group2" type="parallel" thread-size="2">
     <!-- @thread-size 수만큼 하위 step을 동시에 실행시킨다. -->
     <!-- thread1, thread2가 먼저 실행되고 완료되는대로 thread3, thread4가 실행된다 -->
        <step id="thread1" type="java" class="sample.batch.Hello"/>
```

```
        <step id="thread2" type="java" class="sample.batch.Hello"/>
        <step id="thread3" type="java" class="sample.batch.Hello"/>
        <step id="thread4" type="java" class="sample.batch.Hello"/>
    </step>

</job>
```

2.1.4 Delete 스텝

불필요한 파일을 삭제하고자 할 경우 다음과 같이 Delete 스텝을 사용한다.

- @type을 "delete"로 지정
- <resources> 태그 내에 삭제하고자 하는 파일을 지정 (wild card 사용 가능)
- 지정된 파일이 존재하지 않는 경우 무시함

[Delete 스텝 예]

```
<job>

    <description>샘플 Delete 스텝</description>

    <step id="delete" type="delete">
        <resources>
            <resource url="/sample/sample.001"/> <!-- sample.001 파일 삭제 -->
            <resource url="/sample/sample.002"/> <!-- sample.002 파일 삭제 -->
            <resource url="/sample/sample.out.00?"/> <!-- "?"는 문자 1개에 대응 -->
            <resource url="/sample/*.log"/> <!-- 모든 .log 파일 삭제 -->
        </resources>
    </step>

</job>
```

2.1.5 Shell 스텝

OS에서 지원하는 Shell Script를 실행하여 작업을 처리하고자 하는 경우 Shell 스텝을 사용한다. FTP 전송, Sort 프로그램 호출 등 타 솔루션을 기동하고자 하는 경우에 사용할 수 있다.

- @type을 "shell"로 지정
- 하위 <script>내에 실행시킬 스크립트 명령어를 작성
- 명령어 라인이 긴 경우 끝에 "₩₩"을 붙여 다음 라인과 병합 가능
- '0'이외의 리턴값은 실패로 처리되므로 스크립트 작성시 유의

[Shell 스텝 예]

```
<job>

    <description>샘플 Shell 스텝</description>

    <step id="hello" type="shell" restartable="false">
        <script>
            <![CDATA[
            echo "hello world"
            echo "hello world \\
            hello world again"
            exit 0
            ]]>
        </script>
```

```
    </step>

</job>
```

2.2 입출력 정의

배치 프로그램에서 데이터를 처리하기 위해서는 처리하고자 하는 데이터 유형에 맞는 Reader, Writer를 작업정의(CFG)에 지정해야 한다.

Reader, Writer는 데이터 유형에 따른 자동 맵핑을 처리하며 데이터 유형 및 포맷과 상관없이 비즈니스 로직을 개발 할 수 있도록 추상화된 API를 제공한다.

- Reader - File/DB 로부터 한 레코드씩 읽어서 VO 객체로 변환 (read)
- Writer - VO 객체를 데이터 레코드로 변환하여 File/DB 에 저장 (write)
- Updater - 동일한 파일에 대해 한 레코드씩 읽어서 다시 저장 (Updater = Reader + Writer)

배치에서 처리할 수 있는 데이터 유형과 Reader/Writer 사용 가능 여부는 다음과 같다

Table 2.3. 데이터 유형

유형	설명	Reader	Writer	Updater
SAM	Byte 길이로 칼럼이 구분되는 파일	○	○	○
VSAM	구분자(column separator)로 칼럼이 구분되는 파일	○	○	
DB	쿼리를 이용한 데이터베이스의 테이블	○	○	
DAO	Dao 클래스를 이용한 데이터베이스 테이블		○	

[입출력 정의 예]

```
<step id="SampleBat" type="java" class="sample.batch.SampleBat">
    <resources>
        <reader id="reader1" type="SAM" url="${base_dir}/sample.txt" encoding="euc-kr" />
        <reader id="reader2" type="VSAM" url="${base_dir}/sample*.csv" />
        <reader id="reader3" type="DB" url="default" query-id="select">
            <parameters>
                <parameter key="no" value="10" />
                <parameter key="name" value="kim" />
            </parameters>
        </reader>
        <updater id="updator1" type="SAM" url="${base_dir}/sample2.txt" />
        <writer id="writer1" type="SAM" url="${base_dir}/sample.out" delete-empty="true" />
        <writer id="writer2" type="VSAM" url="${base_dir}/sample.csv.out" colsep="|" />
        <writer id="writer3" type="DB" url="default" query-id="insert" />
        <writer id="writer4" type="DAO" url="sample.batch.dao.SampleDao.insert()" />
    </resources>
</step>
```

2.2.1 입출력 속성

입출력 데이터에 대한 Reader, Writer를 정의 시 사용되는 속성(Attribute)에 대한 상세 내역은 다음과 같다.

Table 2.4. 입출력 속성

구분	속성	설명	기본값
필수	@id	고유 ID(스탭 내에서 유일한 값)	
	@type	입출력 데이터 유형 (Table 2.3, “데이터 유형”)	
	@url	입출력 데이터의 위치를 지정 • SAM/VSAM - File Path • DB - Datasource 명 • DAO - DAO 클래스명 + 메써드명	
File	@encoding	입출력 파일의 문자셋을 지정	EUC_KR
	@linesep	파일의 라인 분리자(개행문자)를 지정 (대소문자 구분 안함) • cr, mac - Carrige Return Only • lf, unix - Line Feed Only • crlf, dos, windows - Carrige Return + Line Feed • null - 개행문자 없음(VO 크기 기준으로 처리)	OS Default
	@colsep	VSAM 파일의 칼럼 분리자를 지정	,
	@trim	내용 앞뒤에 포함된 공백문자(white-space)제거 여부	true
	@fixed	SAM 파일의 모든 레코드 길이가 맵핑 데이터와 동일한지 여부를 지정 • true - 모든 레코드 길이가 동일(성능↑) • false - 레코드 길이가 동일하지 않음(레코드를 개행문자 기준으로 구분하여 VO 크기로 강제 맞춤)	true
	@escape	VSAM 파일 esacpe 여부	false
	@append	파일 기록시 append 여부 • true - 파일이 존재할 경우 기존 파일에 이어서 쓴다 • false - 파일이 존재할 경우 삭제 후 새 파일을 만들어 쓴다	false
	@max-size	출력 파일의 최대 용량을 지정(단위 GB) 지정된 용량을 초과하는 경우 에러 발생함(-1 제한 없음)	-1
	@delete-empty	출력 파일의 크기가 0인 경우 자동 삭제	false
	@columns	칼럼 크기를 명시적으로 지정 ex) "3,3,5,5,7" - 5개 칼럼이며 각각의 크기는 3,3,5,5,7임	
	@align-numeric	SAM 파일 Write시 숫자형 컬럼의 정렬 • left - 좌측 정렬	right

구분	속성	설명	기본값
		• right - 우측 정렬	
	@align-string	SAM 파일 Write시 문자형 컬럼의 정렬 • left - 좌측 정렬 • right - 우측 정렬	left
	@pad-numeric	SAM 파일 Write시 숫자형 컬럼의 패딩문자 • zero - 숫자 영(0)으로 패딩 • space - 공백문자로 패딩 **Note** left align 시에는 space만 가능	zero
	@overflow	SAM 파일 write 시 데이터가 지정된 컬럼 크기를 초과했을 시 처리 방법 • error - 에러(exception) 발생 • ignore - 초과분 무시	error
	@malformed	파일 read/write 시 깨진 문자에 대한 처리 방법 • error - 에러(exception) 발생 • ignore - 오류 문자 무시 • replace - 공백(space) 문자로 대체	error
	@unmappable	문자셋 변환 시 변환되지 않는 문자에 대한 처리 방법 • error - 에러(exception) 발생 • ignore - 오류 문자 무시 • replace - 공백(space) 문자로 대체	replace
	@temp	파일 write 시 임시파일을 사용할지 여부 임시파일은 지정된 파일명 뒤에 @temp-suffix 문자열이 더해진 파일명으로 생성되어 스텝이 정상 완료 후 원래 파일명으로 변경됨	false
	@temp-suffix	임시파일 생성 시 사용될 접미사	.tmp
DB	@query	사용할 쿼리를 지정	
	@query-id	사용할 쿼리id 지정(쿼리 파일에서 읽어옴)	
	@query-file	사용할 쿼리 파일의 위치를 지정	
	@fetch-size	데이터의 패치 크기를 지정	1,000
	@batch-update	batch update 적용 여부를 지정	false
Etc	@commit-interval	자동 커밋 간격을 지정	
	@save	커밋 시 읽고 쓴 위치를 기록할지 여부 지정(재시작 시 해당 위치로 자동 이동함)	false

구분	속성	설명	기본값
	@show-log	입출력 raw 데이터를 로그로 출력할지 여부	false
	@mapping	맵핑할 VO 객체의 클래스를 지정 ex) "com.anyframe.sample.SampleVo"	
	@null	null 데이터를 읽고 쓸 때 처리 방법 • error - 에러(exception) 발생 • replace - default 값으로 대체 • allow - null 데이터로 처리	allow

입출력 데이터 유형에 따라 Reader, Writer가 사용할 수 있는 속성은 다음과 같다.

Table 2.5. 입출력 속성 적용

구분	속성	SAM		VSAM		DB		DAO
		Reader	Writer	Reader	Writer	Reader	Writer	Writer
필수	@id	○	○	○	○	○	○	○
	@type	○	○	○	○	○	○	○
	@url	○	○	○	○	○	○	○
File	@encoding	○	○	○	○			
	@linesep	○	○	○	○			
	@colsep			○	○			
	@trim	○		○		○		
	@fixed	○						
	@escape			○	○			
	@append		○		○			
	@max-size		○		○			
	@delete-empty		○		○			
	@columns	○	○	○	○			
	@align-numeric		○					
	@align-string		○					
	@pad-numeric		○					
	@overflow		○					
	@malformed	○		○				
	@unmappable	○	○	○	○			
	@temp		○		○			
	@temp-suffix		○		○			

구분	속성	SAM		VSAM		DB		DAO
		Reader	Writer	Reader	Writer	Reader	Writer	Writer
DB	@query					○	○	
	@query-id					○	○	
	@query-file					○	○	
	@fetch-size					○		
	@batch-update						○	
Etc	@commit-interval	○	○	○	○	○	○	○
	@save	○	○	○	○	○		
	@show-log	○	○	○	○	○	○	○
	@mapping	○		○		○		
	@null	○	○	○	○	○	○	

2.2.2 다중 파일 입력(@url)

파일을 읽어 처리하기 위해서는 @url 속성에 해당 파일 위치를 지정하게 된다. 일반적으로는 하나의 파일을 지정하여 사용하지만 여러 파일을 논리적으로 하나의 파일처럼 묶어서 처리할 수도 있다. (보통 파일들은 모두 같은 레이아웃을 갖는다)

다중파일 지정 방법

- 구분자(;)를 사용하여 다건의 파일을 지정함 - 지정된 순서대로 파일을 읽음
- 와일드카드(?,*)를 사용하여 다건의 파일을 지정함 - 매칭되는 파일을 파일명 순서대로 읽음

[다중파일 지정 예]

```
<job>

    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step1" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample1.txt" />
            <!-- 단건 파일을 지정: /sample/batch/sample1.txt 파일 -->
        </resources>
    </step>

    <step id="step2" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample1.txt; ${base}/sample2.txt" />
            <!-- 두개 파일을 지정: /sample/batch/sample1.txt, /sample/batch/sample2.txt 순서로 읽음 -->
        </resources>
    </step>

    <step id="step3" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/*.txt " />
            <!-- 다중 파일을 지정함: /sample/batch/ 폴더의 *.txt 파일을 파일명 순서로 읽음 -->
        </resources>
    </step>
```

```
</job>
```

2.2.3 파일 이어쓰기(@append)

Writer를 사용하여 파일에 데이터 기록 시 @url에 지정된 파일이 이미 존재하는 경우, 기본적으로 기존에 있는 파일 삭제 후 새로운 파일을 생성하여 데이터를 기록하게 된다. 만약 기존 파일에 이어서 데이터를 기록하고자 하면 @append 속성을 지정해 주어야 한다. (기존 파일이 존재하지 않는 경우는 새로운 파일을 생성하여 기록함)

[이어쓰기 @append]

```
<job>

    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step1" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample1.txt" append="true" />
            <!-- /sample/batch/sample1.txt 파일이 존재하는 경우 기존 파일에 이어서 기록함 -->
        </resources>
    </step>

</job>
```

2.2.4 Escape(@escape)

VSAM 파일은 데이터를 칼럼 분리자(column separator)를 기준으로 구분한다. 만약 본문 내용에 분리자(separator) 문자가 사용될 경우 데이터 처리가 올바르게 동작하지 않으며 이 경우에는 본문에 대해 Escape 처리가 필요하다.

다음과 같이 작업 정의(XML)에 Reader, Writer의 @escape 속성을 "true"로 지정하면 해당 Reader, Writer는 자동으로 escape 처리를 수행한다. (단 처리 성능은 약간 저하됨)

> **Note**
>
> Escape : separator가 포함된 컬럼은 따옴표("")로 묶어서 값을 저장하고 읽을 때에는 따옴표("")를 제외함

[Escape 예]

```
<job>

    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step1" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="VSAM" url="${base}/sample.txt" escape="true" />
            <writer id="writer" type="VSAM" url="${base}/sample.out" escape="true" />
        </resources>
    </step>

</job>
```

다음은 escape 처리가 된 VSAM 파일의 예이다. 구분자가 콤마(,)인 경우 취미 항목에 [독서,영화감상]과 같이 ','가 포함된 내용이 들어가는 경우 ["독서,영화감상"] 처럼 앞뒤로 따옴표가 붙어서 처리된다.

```
[이름],[나이],[주소],[취미]
----------------------------
```

```
홍길동,30,서울,"독서,영화감상"
이순신,31,부산,"수영,국궁"
일지매,32,울산,무예
... ...
... ...
```

2.2.5 쿼리 지정(@query-id)

데이터베이스 테이블을 읽고 쓰기 위해서는 Reader, Writer가 사용할 쿼리를 설정해야 한다. 쿼리는 작업 정의(XML)에서 @query-file, @query-id 속성을 통하여 지정할 수 있으며 프로그램 내에서 API를 사용하여 지정하는 것도 가능하다. (Section 3.4.2, “Reader, Writer API” 참조)

[쿼리 지정]

```
<job>

    <step id="step1" type="java" class="sample.batch.SimpleBat">
        <resources>
         <!-- query-id만 지정된 경우 기본 쿼리파일에서 해당 쿼리를 찾음 -->
         <!-- 기본 쿼리파일은 Class로부터 유추하여 "sample/batch/SimpleBat_sql.xml" -->
            <reader id="reader" type="DB" url="default" query-id="select" />

            <!-- query-file 속성으로 쿼리를 조회할 쿼리파일을 명시적으로 지정할 수 있음 -->
            <writer id="writer" type="DB" url="default" query-id="insert" query-file="sample/common/Common_sql.xml" />
        </resources>
    </step>

    <!-- step의 query-file 속성을 지정하여 하위 reader/writer가 사용할 쿼리파일을 명시적으로 지정할 수 있음 -->
    <step id="step2" type="java" class="sample.batch.SimpleBat" query-file="sample/common/Common_sql.xml">
        <resources>
            <reader id="reader" type="DB" url="default" query-id="select" />
            <writer id="writer" type="DB" url="default" query-id="insert" />
        </resources>
    </step>

</job>
```

2.2.6 쿼리 파라미터

데이터베이스 테이블을 읽기 위하여 Reader를 사용할 때, 실행하는 쿼리에 파라미터가 포함된 경우 해당 파라메터 값을 설정해주어야 한다. 쿼리에 포함된 파라메터는 작업정의(CFG)에서 <parameter> 테그를 사용하여 지정하거나 프로그램 내에서 API를 사용하여 지정할 수 있다. (Section 3.9, “배치 파라미터” 참조)

[쿼리 파라미터]

```
<job>

    <step id="step1" type="java" class="sample.batch.SimpleBat">
        <resources>
         <!-- SELECT * FROM SAMPLE WHERE NO=:no AND TYPE=:type -->
            <reader id="reader" type="DB" url="default" query-id="select" >
             <parameters>
              <parameter key="no" value="5"/>
              <parameter key="type" value="AA"/>
              <!-- 쿼리 파라미터(no, type)의 값을 parameter 태그로 지정함 -->
              <!-- value에 ${var} 형태의 배치 파라미터를 사용할 수 있음 -->
             </parameters>
            </reader>
        </resources>
    </step>
```

```
</job>
```

2.2.7 일괄 반영(@batch-update)

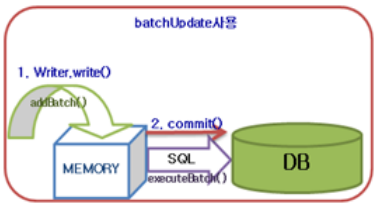
DB Writer의 경우 기본적으로 write()를 호출할 때마다 쿼리를 수행하여 결과를 DB 테이블에 반영한다.

다음과 같이 Writer의 @batch-update 속성을 true로 지정하면, write() 때마다 쿼리를 수행하지 않고, 커밋시점에 일괄적으로 쿼리를 수행하는 것이 가능하다.

[Batch Update 사용]

```
<job>

    <step id="step1" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="DB" url="default" query-id="select" />
            <writer id="writer" type="DB" url="default" query-id="insert" batch-update="true" />
        </resources>
    </step>

</job>
```

Batch update를 사용할 경우 성능은 향상되나 커밋이 수행될 때까지 쿼리를 메모리에 저장하기 때문에 필수적으로 일정 간격마다 커밋을 수행해야 한다. 또한 write() 시 쿼리가 수행되는 것이 아니므로 write() 결과에 따른 처리는 불가능하다. (변경된 row 수 확인, Exception 처리 등)



2.3 쿼리 정의

데이터베이스에 대한 입출력을 처리하기 위해서는 Reader/Writer가 사용할 쿼리를 정의해야 한다.

배치에서 사용되는 쿼리는 별도의 쿼리파일(XML)에 정의하며 @query-id 속성을 이용하여 사용할 쿼리를 지정한다.

@query-id로 지정된 쿼리는 기본적으로 배치 프로그램 Class와 동일한 위치에 있는 "[class명]_sql.xml"의 쿼리파일로부터 쿼리를 찾게 된다. 다른 쿼리파일을 사용하기 위해서는 @query-file 속성을 이용하여 명시적으로 쿼리파일을 지정해야 한다.

예) `sample.batch.SimpleBat`에서 사용할 쿼리는 `sample/batch/SimpleBat_sql.xml`에 정의한다

> **Note**
>
> '?' 바인드 변수형태와 NamedParameter 형태 지원

[쿼리 정의 예]

```
<sql>

    <!-- 쿼리에 "<", ">" 문자가 포함되어 있는 경우 CDATA 섹션으로 Escape 시킴 -->
    <query id="insert">
        <![CDATA[
```

```
            INSERT INTO SAMPLE ( NO, NAME, AGE )
            VALUES ( :no, :name, :age )
        ]]>
    </query>

    <query id="select">
        <![CDATA[
            SELECT NO, NAME, AGE
              FROM SAMPLE
         ]]>
    </query>

</sql>
```

2.4 파라미터(Parameter)

파라미터는 배치 프로그램 내에서 Key/Value 형태로 등록/사용할 수 있는 일종의 변수로써 일반 Java 변수와 달리 배치 파라미터는 Commit 시 변수 테이블에 자동 저장되고 배치 작업을 재 시작할 때 마지막에 저장된 값으로 복구된다. 또한 파라미터는 작업정의(CFG)에서 @url 및 script에서 변수 형태로 표현하여 사용할 수 있으며 다른 파라미터를 정의할 때도 사용할 수 있다.

파라미터에는 작업 파라미터(Job Parameter)와 스텝 파라미터(Step Parameter)가 있다.

작업 파라미터 (Job Parameter)

- 배치 작업내 모든 스텝에서 사용할 수 있음
- 스탭 간 결과값을 전달하기 위해서 사용할 수 있음(이전 스텝에서 추가/변경한 작업 파라미터는 이후 스텝에서 사용할 수 있다)
- 작업 실행 시 "key=value" 형태로 전달된 모든 외부 Argument는 작업 파라미터로 자동 등록됨
- 작업정의(CFG)의 <job> 하위에 <parameter>로 등록하거나 프로그램 내에서 API를 통해 등록/변경할 수 있음

[작업 파라미터 예]

```
<job>

    <!-- 작업 파라미터는 job 하위에 parameter 태그를 이용하여 등록함 -->
    <parameters>
        <parameter key="base" value="/sample/batch"/>
        <parameter key="today" value="20130605"/>
        <parameter key="file" value="${base}/${today}.txt"/>
        <!-- 사전에 정의한 파라메터를 사용하여 새로운 파라메터 등록 가능 -->
    </parameters>

    <step id="SimpleBat" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample.txt" />
            <writer id="writer" type="SAM" url="${base}/sample.out" />
        </resources>
    </step>
    ...
</job>
```

스텝 파라미터 (Step Parameter)

- 지정된 스텝 내에서만 사용할 수 있음
- 병렬스텝(Parallel)에서 각 스텝 별 입력값을 전달하고자 사용
- 재시작 시 이전 값을 복구하여 재처리하고자 할 때 사용

- 작업 정의(XML)의 <step> 하위에 <parameter>로 등록하거나 프로그램 내에서 API를 통해 등록/변경할 수 있음

[스텝 파라미터 예]

```
<job>

    <!-- 작업 파라미터는 job 하위에 parameter 태그를 이용하여 등록함 -->
    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="parallel" type="parallel">

        <step id="SimpleBat" type="java" class="sample.batch.SimpleBat">
            <!-- 스텝 파라미터는 step 하위에 parameter 태그를 이용하여 등록함 -->
            <parameters>
                <parameter key="seq" value="1"/>
            </parameters>
            <resources>
                <reader id="reader" type="SAM" url="${base}/sample${seq}.txt" />
                <writer id="writer" type="SAM" url="${base}/sample${seq}.out" />
            </resources>
        </step>

        <step id="SimpleBat" type="java" class="sample.batch.SimpleBat">
            <!-- 스텝 파라미터는 step 하위에 parameter 태그를 이용하여 등록함 -->
            <parameters>
                <parameter key="seq" value="2"/>
            </parameters>
            <resources>
                <reader id="reader" type="SAM" url="${base}/sample${seq}.txt" />
                <writer id="writer" type="SAM" url="${base}/sample${seq}.out" />
            </resources>
        </step>

    </step>
    ...
</job>
```

2.4.1 작업 파라미터 전달

작업 파라미터는 다음과 같은 방법으로 전달할 수 있다.

- 작업 실행 시 명령어 인자에 "key=value" 형태로 전달
- 배치 프로그램 내에서 API를 사용하여 전달 (Section 3.3.1, “Task API” 참조)

[작업 실행 시 파라미터 전달 (Local 테스트 실행)]

```
import com.anyframe.batch.test.BatchTestUtils

public class TestJob {

    @Test
    public void run() {
        // 두번째 인자로 key=value 형태의 파라미터를 전달
        BatchTestUtils.launchJob("sample/sample_cfg.xml", new String[]{"base=/batch/sample", "today=20130605" });
    }

}
```

[작업 실행 시 파라미터 전달 (서버 실행)]

```
# runjob.sh sample/sample_cfg.xml base=/batch/sample today=20130605
```

> **Note**
>
> 작업정의(CFG)에 등록된 파라미터와 동일한 KEY값의 파라미터가 실행 시 전달되면 실행 시 전달된 값이 적용된다.

2.4.2 파라미터 사용

작업 정의(XML) 시 변수 표현식을 활용하여 배치 파라미터 값을 사용할 수 있다.

파라미터는 작업 정의(XML)에 포함된 Reader, Writer의 @url과 Shell Step의 Script 내에서 사용할 수 있다.

- @url에 변수 사용 - 동적으로 입출력 파일을 변경하고자 할 때
- script에서 사용 - shell script에 값 전달

> **Important**
>
> - 작업정의(CFG)에 사용된 변수는 스텝이 실행되는 시점에 값으로 치환되어 적용된다.
> - 변수 값은 [스텝 파라미터] → [작업 파라미터] → [환경변수] 순으로 검색을 하며, 등록된 값이 없는 경우에는 치환되지 않는다.

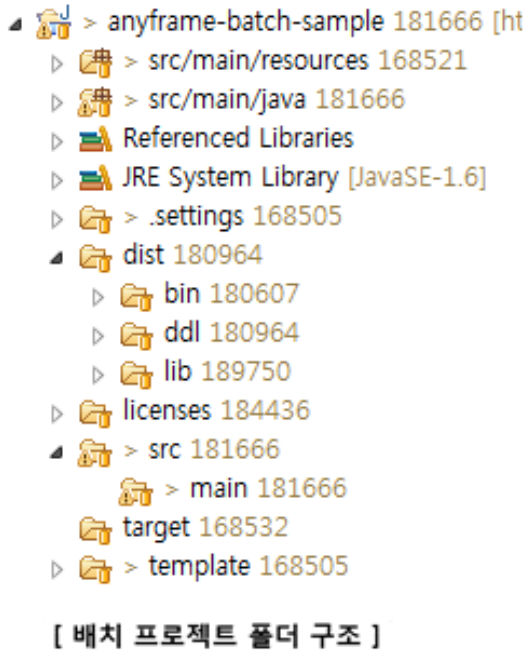
[파라미터 사용 예]

```
<job>

    <parameters>
        <parameter key="base" value="/sample/batch"/>
        <parameter key="greeting" value="Hello World~"/>
    </parameters>

    <step id="sample" type="java" class="sample.batch.SimpleBat">
        <resources>
            <!-- url에 포함된 ${base}이 치환되어 "/sample/batch/sample.txt" -->
            <reader id="reader" type="SAM" url="${base}/sample.txt" />
            <!-- url에 포함된 ${base}이 치환되어 "/sample/batch/sample.txt.out" -->
            <writer id="writer" type="SAM" url="${base}/sample.txt.out" />
        </resources>
    </step>

    <step id="sample_script" type="shell">
        <!-- script에 포함된 ${greeting}이 치환되어 "Hello World" 출력 -->
        <script>echo ${greeting}</script>
    </step>

</job>
```

3장. 배치 프로그램 개발

3.1 배치 프로젝트

배치 프로젝트는 일반적인 Java 프로젝트와 동일하며 다음과 같은 구조를 갖는다.



[배치 프로젝트 폴더 구조]

- src/main/java
  - 업무 패키지 별로 배치 JOB 구성 요소가 위치함 (task class, vo class, xml 등)
- src/main/resources
  - Local에서 배치를 구동하기 위한 설정파일
- src/main/test
  - 테스트 코드가 위치함
- dist
  - 서버에서 배치를 구동하기 위한 설정파일
- dist/lib
  - 배치를 실행하기 위한 라이브러리 파일
- template
  - 배치 개발도구가 사용하는 템플릿 파일

3.2 프로그램 구성 요소

프레임워크 기반으로 개발되는 배치 프로그램은 다음과 같은 요소들로 구성된다.

Table 3.1. 배치 프로그램 구성

구분	유형	설명
작업정의(CFG)	XML	• 배치 작업을 정의한 XML 파일. (2장. 배치 작업 정의 참조) • 하위 스텝과 스텝에서 사용되는 reader/writer를 명시적으로 선언함 • 하나의 작업은 하나의 작업 정의(XML)로 표현
쿼리 파일	XML	• 데이터베이스에 대하여 입출력을 처리할 경우 수행할 쿼리를 XML 형태로 모은 파일 • Section 2.3, “쿼리 정의” 참조
배치 프로그램(TASK)	Java Class	• 입출력 데이터를 비즈니스 로직에 맞게 처리할 Java 프로그램

구분	유형	설명
		• AbstractTask를 상속해야 하며, execute() 메써드를 재정의(Override)해야 함
데이터 맵핑(VO)	Java Class	• 입출력 데이터의 하나의 레코드에 대응하는 자바 데이터 객체 • 입출력 데이터와 자바 데이터 객체 사이의 변환은 reader/writer가 자동으로 처리함

3.3 배치 프로그램(Task)

배치 프로그램(Task)는 배치 작업의 스탭에서 구동시킬 Java 프로그램으로 다음과 같이 특성을 갖는다.

- AbstractTask를 상속, execute( ) 메써드를 재정의(Override)한다
- execute( )의 인자로 전달되는 ItemFactory를 통해 reader/writer 획득하여 입출력 데이터를 처리
- 선후처리가 필요한 경우 beforeStep( ), afterStep( ) 메써드를 재정의(Override)한다
- 에러 없이 프로그램 끝에 도달하거나 프로그램 중간에 return 하면 해당 프로그램을 호출한 스텝은 완료(Complete)
- 프로그램 실행 중 에러(Exception)이 발생하면 스텝은 실패(Fail)

[Task 예]

```
public class SimpleBat extends AbstractTask {

    // logger 선언
    private static final Logger logger = LoggerFactory.getLogger(SimpleBat.class);

    @Override
    public void execute(ItemFactory factory) throws Exception {

     // factory로부터 id를 가지고 reader/writer 획득
        ItemWriter writer = factory.getItemWriter("writer");
        ItemReader reader = factory.getItemReader("reader", SampleVo.class);

        while(reader.next()) {                // 입력데이터 끝까지
            SampleVo vo = reader.read();      // 차례대로 읽어서(read)
            if(vo.getNo() > 5) {              // 조건(no>5)에 맞는 데이터만
                writer.write(vo);             // 결과에 기록 (write)
            }
        }

        logger.debug("debug message");        // 개발 시 필요한 디버그 메시지 출력

        print("************** 실행 결과 **************");   // 처리 결과 로깅
        print("* READ  횟수 : " + reader.getItemCount());
        print("* WRITE 횟수 : " + writer.getItemCount());
        print("***************************************");

    }

    @Override
    public void beforeStep() {
        // 스텝 실행 전에 스텝 파라미터 등록
        // beforeStep에서 등록된 파라미터는 작업정의(CFG)에서 변수${var}로 사용할 수 있음
        putStepParameter("filename", "/batch/sample.txt");
    }

}
```

3.3.1 Task API

AbstractTask가 제공하는 API는 다음과 같다.

Table 3.2. API

구분	API	설명	비고
작업실행	execute(ItemFactory factory)	배치 프로그램의 메인 메써드 재정의하여 업무처리 로직을 구현	필수
	beforeStep()	스텝 시작전에 호출됨 재정의하여 선처리 로직을 추가할수 있으며 이곳에서 등록된 파라미터는 본처리에 적용됨	필요한 경우 재정의
작업정보	getJobName()	현재 작업의 이름을 반환한다. (String)	
	getJobId()	현재 작업의 고유 ID를 반환한다 (long)	
	getStepName()	현재 스텝의 이름을 반환한다 (String)	
	getStepId()	현재 스텝의 고유 ID를 반환한다 (long)	
파라미터	getJobParameter(String key)	key에 해당하는 작업 파라미터를 반환한다	값이 없는 경우 null
	getStepParameter(String key)	key에 해당하는 스텝 파라미터를 반환한다	
	putJobParameter(String key, Object value)	작업 파라미터를 등록한다	등록된 파라미터가 있는 경우 overwrite 됨
	putStepParameter(String key, Object value)	스텝 파라미터를 등록한다	
트랜잭션	commit()	현재까지 처리한 내용을 커밋한다	
	rollback()	현재까지 처리한 내용을 롤백한다	
진행상황	setTotalCount(int count)	스텝에서 처리할 전체 처리량을 지정한다	스텝의 진행경과로 활용
	setCurrentCount(int count)	현재 처리량을 갱신한다	
로그	print(String message)	처리결과를 로그에 기록한다	

3.4 Reader & Writer

프로그램 내에서 입출력 데이터를 처리하기 위해서는 작업정의(CFG)에서 데이터 유형 및 포맷에 맞게 Reader, Writer를 정의하고, 정의된 Reader, Writer를 execute() 메써드의 인자로 전달되는 ItemFactory를 통하여 획득하여 사용한다.

ItemReader

- ItemReader는 입력 File이나 DB로부터 데이터를 읽어서 프로그램 내에서 사용할 수 있는 데이터객체(VO)로 변환하는 역할을 수행한다. next()를 호출하여 데이터의 우뮤를 확인할 수 있으며 read()를 호출하여 현재 위치의 데이터를 데이터객체(VO)로 변환한다.

ItemWriter

- ItemWriter는 데이터 객체(VO)를 출력 파일이나 DB에 저장할 때 사용한다. write()를 호출하며 데이터객체(VO)를 전달하면 데이터객체(VO)를 변환하여 파일 또는 DB에 저장한다.

ItemUpdater

- ItemUpdater는 SAM 파일의 읽어서 해당 파일의 특정 데이터에 대해 수정이 필요할 시 사용된다. read()를 호출하여 한 라인을 읽고 write()를 호출하여 해당 라인을 업데이트한다.

> **Important**
>
> ItemUpdater는 SAM 파일에 대해서만 사용 가능하다.

다음은 작업정의(CFG)에 정의된 Reader, Writer를 프로그램 내에서 획득하여 사용하는 예이다.

[작업정의(CFG)]

```
<job>

    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step1" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample1.txt" />
            <writer id="writer" type="SAM" url="${base}/sample1.out" />
        </resources>
    </step>

</job>
```

[배치 프로그램]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        // XML에 정의한 id를 가지고 reader/writer 획득

        ItemWriter writer = factory.getItemWriter("writer");
        ItemReader reader = factory.getItemReader("reader", SampleVo.class);

        ... ...
        ... ...

    }

}
```

3.4.1 ItemFactory

ItemFactory는 작업정의(CFG)에 정의한 Reader, Writer를 프로그램 내에서 사용할 수 있도록 반환하는 기능을 제공하며 필요 시 신규 Reader, Writer를 생성하는 기능도 제공한다.

ItemFactory가 제공하는 API는 다음과 같다.

API	설명	비고
getItemReader(String id) getItemWriter(String id) getItemUpdater(String id)	작업정의에 정의한 id에 해당하는 reader,writer,updater를 반환한다.	

getItemReader(String id, Class mapping)
getItemUpdater(String id, Class mapping)
	작업정의에 정의한 id에 해당하는 reader, updater를 반환한다.
	반환된 reader, updater는 mapping 기준으로 데이터 맵핑을 수행한다.
	`다음과 동일함`
	`ItemReader reader = factory.getItemReader(id);`
	`reader.setMapping(mapping);`	

createItemReader(String id, String type, String url, Map<String, String> attributes) createItemWriter(String id, String type, String url, Map<String, String> attributes) createItemUpdater(String id, String type, String url, Map<String, String> attributes)	신규 reader,writer,updater를 생성하여 반환한다. • id - 고유 식별자 • type - 리소스 유형 (SAM, VSAM, DB) • url - 리소스 위치 (파일위치 또는 데이터 소스명) • attributes - 추가 속성 (Section 2.2.1, “입출력 속성” 참조)	생성기능은 제한될 수 있음

3.4.2 Reader, Writer API

ItemReader, ItemWriter, ItemUpdater가 제공하는 API는 다음과 같다.

구분	API	설명	비고
reader	boolean next()	데이터가 존재하는 경우 true를 반환하고 읽는 위치(cursor)를 다음 줄(row)로 이동한다. 없는 경우 false를 반환한다. 초기 위치는 첫 번째 줄(row) 앞에 위치하며, 결과가 false인 경우 읽는 위치는 마지막 줄 뒤에 위치한다.	
	boolean isLast()	현재 위치가 입력 데이터의 마지막 줄(row)인지 여부를 반환한다.	File Only
	Object read()	현재 위치의 데이터(row)를 읽어 VO 객체로 반환한다.	
	void reset()	입력 데이터를 초기화 한다. File인 경우 읽는 위치(Cursor)를 처음으로 이동하며 DB인 경우 쿼리를 재실행하여 ResultSet을 다시 구성한다.	
	void setMapping(Object mapping)	입력 데이터를 맵핑할 VO 클래스를 지정한다. mapping: VO 클래스 또는 VO 객체	@mapping
	void setParameters(Object param)	쿼리에 포함된 파라메터를 설정하고 reader를 초기화 시킨다. 이후 next(), read()는 새로운 ResultSet을 기준으로 처리됨 param: key(String)/value(Object) 형태의 맵 또는 VO객체	<parameters> DB Only

구분	API	설명	비고
writer	int write(Object vo)	Vo 객체를 출력 데이터에 기록한다. Return: 기록된 row 수 (insert/update 된 Row 수)	
	int write()	Setter Method를 통해 설정된 값을 출력 데이터에 기록한다. Return: 기록된 row 수 (insert/update 된 Row 수)	
공통	long getItemCount()	현재까지 처리한 item 수를 반환한다.	
	String getURL()	URL을 반환한다.	@url
	void setColumns(int[] columns)	VO를 사용하지 않는 경우 칼럼 정보를 셋팅한다 ex) setColumns(new int[]{3,3,5,5,}) 4개의 칼럼이며 각각의 길이는 3,3,5,5,임	@columns
	void setShowLog(boolean showLog)	입출력 데이터를 로그로 출력할지 지정한다.	@show-log
Get/ Set	getXXX(int index)	index 칼럼에 위치한 데이터를 반환한다.	
	setXXX(int index, Object value)	index 칼럼에 value 데이터를 셋팅한다.	

3.4.3 ItemUpdater

ItemUpdater는 하나의 SAM 파일의 읽어서 해당 파일을 직접 수정할 수 있다. 따라서 ItemUpdater는 Reader,Writer API를 모두 사용할 수 있으며 `read()`를 호출하여 라인을 읽고 읽은 데이터를 변경한 후 `write()`를 호출하여 해당 라인을 업데이트한다.

다음은 ItemUpdater를 사용하여 "MONEY" 칼럼의 데이터를 변경하는 프로그램 예이다.

[Updater 사용 예]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemUpdater updater = factory.getItemUpdater("updater", SampleVo.class);

        while(updater.next()) {
            SampleVo vo = updater.read();
            if(vo.getNo() == 5) {
                vo.setMoney(100);       // 조건(no=5)에 맞는 레코드의 칼럼값(money)을 변경하여
                updater.write(vo);      // 같은 파일에 기록
            }
        }

    }
}
```

3.4.4 쿼리 사용

데이터베이스 테이블을 읽고 쓰기 위해서는 Reader, Writer가 사용할 쿼리를 설정해야 한다. 쿼리는 작업 정의(XML)에서 @query-file, @query-id 속성을 통하여 지정할 수 있으며 프로그램 내에서 setQueryId(), setQueryPath()를 사용하여 설정할 수 있다.

[Section 2.2.1, “입출력 속성” 참조] , [Section 3.4.2, “Reader, Writer API” 참조]

> **Note**
>
> 쿼리 설정은 작업정의(CFG)에 설정하는 것을 권장함

> **Important**
>
> API를 이용한 쿼리 지정은 next(), read(), write() 전에 수행되어야 한다.

다음은 프로그램 내에서 API를 사용하여 쿼리를 지정하는 예이다

[쿼리 API 사용 예]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemWriter writer = factory.getItemWriter("writer");
        ItemReader reader1 = factory.getItemReader("reader1");
        ItemReader reader2 = factory.getItemReader("reader2");

        writer.setQueryId("insert");                    // 기본 쿼리파일에서 id 지정
        reader1.setQueryId("select");                   // (SimpleBat_sql.xml)

        reader2.setQueryPath("common/Common_sql.xml");  // 명시적으로 쿼리파일을 지정 (Common_sql.xml)
        reader2.setQueryId("select");                   // 새로 지정한 쿼리파일의 id 지정

        while(reader1.next()) {
            SampleVo vo = reader1.read();

            if(vo.getNo() == 5) {
                vo.setMoney(100);      // 조건(no>5)에 맞는 레코드의 칼럼값(money)을 변경하여
                writer.write(vo);      // 같은 파일에 기록
            }
        }

    }
}
```

만약 Reader 실행하는 SELECT 쿼리에 파라미터(조회조건)가 포함된 경우 파라메터 값을 사전에 설정해주어야 한다. 쿼리에 포함된 파라메터는 작업정의(CFG)에서 <parameter> 엘리먼트를 통하여 지정하거나 프로그램 내에서 setParameters()를 사용하여 설정할 수 있다. (Section 2.2.6, “쿼리 파라미터” 참조, Section 3.4.2, “Reader, Writer API” 참조)

다음은 프로그램 내에서 API를 사용하여 파라미터를 설정하는 예이다

[MAP을 사용한 파라미터 설정]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemReader reader = factory.getItemReader("reader");
```

```
        reader.setQueryId("select");  // SELECT * FROM SAMPLE WHERE NO=:no AND TYPE=:type

        Map<String, Object> param = new HashMap<String, Object>();
        param.put("no", 5);
        param.put("type", "AAA");

        reader.setParameters(param);  // Map으로 파라미터 전달. next() 전에 설정되어야 함

        while(reader.next()) {
            SampleVo vo = reader1.read();
            ... ...
            ... ...
        }

    }
}
```

[VO를 사용한 파라미터 설정]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemReader reader = factory.getItemReader("reader");
        reader.setQueryId("select");  // SELECT * FROM SAMPLE WHERE NO=:no AND TYPE=:type

        SampleVo param = new SampleVo();
        param.setNo(5);
        param.setType("AAA");

        reader.setParameters(param);  // VO로 파라미터 전달. next() 전에 설정되어야 함

        while(reader.next()) {
            SampleVo vo = reader1.read();
            ... ...
            ... ...
        }

    }
}
```

3.5 입출력 데이터 맵핑

데이터 맵핑은 파일 또는 DB에 있는 데이터를 Java 프로그램 내에서 사용할 수 있는 객체(VO)로 변환하거나 그 반대를 의미한다.

배치 프로그램에서 입출력 데이터에 대한 맵핑은 Reader, Writer에 의해 자동으로 처리된다. 맵핑에 필요한 정보(데이터유형, 구분자, 인코딩, 칼럼길이 등)를 작업정의(CFG)의 <reader>, <writer>의 속성으로 지정하면 Reader, Writer는 속성에 따라 데이터를 변환한다. 따라서 배치 프로그램 내에서 데이터 유형 및 포맷에 따른 개발이 필요 없으며 비즈니스 측면에서의 개발에만 집중할 수 있다.

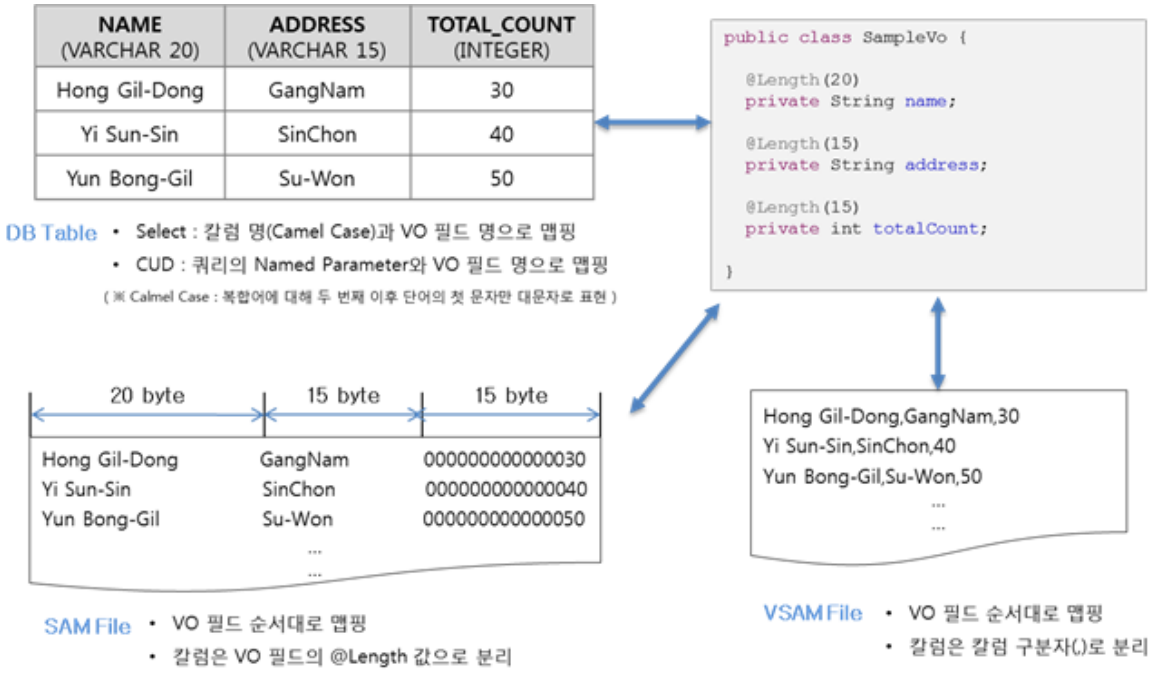
[데이터 맵핑 예]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {
```

```
        ItemWriter writer = factory.getItemWriter("writer");
        // reader 획득 시 맵핑할 VO 클래스를 지정
        ItemReader reader = factory.getItemReader("reader", SampleVo.class);

        while(reader.next()) {
            SampleVo vo = reader.read();  // 입력 데이터를 SampleVo로 변환하여 반환
            ... ...
            ... ...
            writer.write(vo);             // 전달된 SampleVo를 출력 데이터로 변환
        }

    }

}
```

3.5.1 유형 별 맵핑 기준

입출력 데이터 유형은 SAM, VSAM, DB가 있으며 입출력 데이터 유형에 따른 맵핑 기준은 다음과 같다.



3.5.2 Value Object(VO)

VO는 데이터(레코드)의 값을 담고 있는 Java 객체로써 데이터(레코드)를 구성하는 필드들을 VO Class의 멤버 변수로 하고 해당 변수에 접근할 수 있는 Getter/Setter 메쏘드의 조합으로 구성된다.

배치 프로그램에서 Reader, Writer를 사용한 데이터 맵핑은 VO를 사용하는 것을 기본으로 하나 VO를 사용하지 않고 직접 데이터를 접근하여 사용하는 것도 가능하다.

VO를 사용할 때 장단점은 다음과 같다.

장점

- 데이터 표준과 연계한 VO 클래스 자동 생성으로 개발 생산성 증가
- 명시적인 데이터 타입(Integer, String, Date 등) 사용으로 명시성 확보 및 오류 감소
- 비즈니스 로직과 맵핑 로직 분리로 코드품질 및 유지보수성 증가

단점

- 클래스 수 증가

VO 클래스를 사용하지 않고 입출력 데이터를 처리하기 위해서는 프로그램 내에서 setColumns()를 호출하여 데이터 레이아웃을 지정해야 하며 Reader, Writer가 제공하는 Getter, Setter 메써드를 사용하여 각 칼럼의 데이터 타입에 맞춰서 프로그램을 작성해야 한다.

다음은 같은 프로그램에 대해서 VO를 사용한 경우와 사용하지 않은 프로그램의 비교이다.

[VO 사용]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemWriter writer = factory.getItemWriter("writer");
        ItemReader reader = factory.getItemReader("reader", SampleVo.class);

        while(reader.next()) {
            SampleVo vo = reader.read();
            writer.write(vo);
        }

    }

}
```

[Get/Set 사용]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemWriter writer = factory.getItemWriter("writer");
        ItemReader reader = factory.getItemReader("reader");

        reader.setColumns(new int[]{5,5,10,10,5,20,20});  // 각 칼럼의 수와 길이를 배열로 지정함 (SAM)
        writer.setColumns(new int[]{5,5,10,10,5,20,20});  // 각 칼럼의 수와 길이를 배열로 지정함 (SAM)

        while(reader.next()) {
            reader.read();    // read() 시 VO를 반환하지 않음

            // 각 칼럼별 데이터 타입에 맞는 ItemReader의 Getter를 호출하여 값 조회
            int no = reader.getInt(0);
            String name = reader.getString(1);
            int age = reader.getInt(2);
            BigDecimal salary = reader.getBigDecimal(3);
            Timestamp birthday = reader.getTimestamp(4);
            String address = reader.getString(5);
            String description = reader.getString(6);

            // 각 칼럼별 데이터 타입에 맞는 ItemWriter의 Setter를 호출하여 값 설정
            writer.setInt(0, no);
            writer.setString(1, name);
            writer.setInt(2, age);
            writer.setBigDecimal(3, salary);
            writer.setTimestamp(4, birthday);
            writer.setString(5, address);
            writer.setString(6, description);

            writer.write();  // Setter로 호출된 칼럼 데이터를 저장

        }
```

```
    }

}
```

3.5.3 VO 클래스 생성

입출력 데이터 맵핑에 사용되는 VO 클래스는 FDD 에서 <Vo> 유형의 클래스로 생성하며 각 필드 구성 및 편집은 VO Editor를 사용하여 편집한다.

> **Note**
>
> VO Editor 사용법은 개발도구 매뉴얼 참조

3.6 트랜잭션

배치 프로그램 내에서 별도의 트래잭션 처리를 하지 않더라도 프레임워크에서 다음과 같은 기본적인 트랜잭션 처리를 수행한다.

- 스텝(프로그램) 시작 시 자동으로 트랜잭션 시작
- 에러 없이 정상적으로 처리되면 스텝(프로그램) 종료 시 자동으로 커밋(Commit)
- 에러(Exception) 발생 시 자동으로 롤백(Rollback)

> **Note**
>
> 병렬처리(Parallel)으로 실행되는 각 스텝은 별도의 트랜잭션으로 처리됨

그러나 다음과 같은 경우에는 프로그램 내에서 별도의 트랜잭션 제어가 필요하다.

처리하는 데이터가 큰 경우

- 스텝 종료 시 전체 데이터를 커밋/롤백 하는 것은 DB에 악영향을 주므로 중간중간 커밋 처리를 해주어야 함

프로그램 오류 시 오류 지점부터 재시작이 필요한 경우

- 처리 중간에 커밋을 하여 오류 발생 전까지 데이터 및 재시작에 필요한 파라메터를 저장해야 함 (Section 3.10, “재시작(재처리)” 참조)

3.6.1 Transaction API

배치 프로그램 내에서 명시적으로 트랜잭션을 제어하고자 하는 경우 AbstractTask가 제공하는 다음 API를 사하여 현재까지 처리한 데이터를 커밋 하거나 롤백 할 수 있다.

API	설명	비고
commit()	현재까지 처리한 내용을 커밋한다.	
rollback()	현재까지 처리한 내용을 롤백한다..	
setCommitInterval(int interval)	커밋 간격을 설정한다	
checkCommit()	누적 카운트를 증가시키고 카운트가 setCommitInterval()에 의해 설정된 간격을 초과하는 경우 자동으로 커밋을 수행한다.	

3.6.2 자동 커밋

처리하는 데이터 양이 크거나 오류 지점부터 재시작이 필요한 경우 처리 중간에 커밋을 수행해야 한다. 일정 간격으로 자동 커밋을 하고자 하는 경우 작업정의(CFG)의 reader, writer에 @commit-interval 속성을 지정하여 처리할 수 있으며 프로그램 내에서 트랜잭션 API를 호출하여 명시적으로 처리할 수도 있다.

다음은 1,000건 처리(write) 시 커밋되도록 하는 프로그램 예이다.

[방법1. @commit-interval 사용]

```
<job>

    <step id="step" type="java" class="sample.batch.SampleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample.txt" />
            <writer id="writer" type="DB"  url="default" commit-interval="1000" />
            <!-- @commit-interval 속성을 지정하면 프로그램 내에서 -->
            <!-- writer의 write 횟가 도달하면 자동으로 커밋됨(reader인 경우 read 횟수) -->
        </resources>
    </step>

</job>
```

[방법2. checkCommit() API 사용]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemWriter writer = factory.getItemWriter("writer");
        ItemReader reader = factory.getItemReader("reader", SampleVo.class);

        setCommitInterval(1000);  // 커밋 간격을 지정함

        while(reader.next()) {
```

```
            SampleVo vo = reader.read();
            writer.write(vo);
            checkCommit();        // main loop 내에서 커밋 횟수를 누적시킴
                                  // 커밋 간격에 도달하면 자동 커밋됨
        }

    }

}
```

[방법3. 직접 커밋]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemWriter writer = factory.getItemWriter("writer");
        ItemReader reader = factory.getItemReader("reader", SampleVo.class);

        long count = 0;
        while(reader.next()) {
            SampleVo vo = reader.read();
            writer.write(vo);

            if(count++ % 1000 == 0) {  // count를 누적시켜 1,000 간격으로 커밋시킴
                commit();
            }
        }

    }

}
```

3.7 예외처리(Exception)

배치 프로그램을 실행하는 도중 예외(Exception) 발생하면 해당 스텝은 에러가 발생한 것으로 간주하여 스텝과 작업은 실패(Fail) 처리된다.

따라서 프로그램 실행 중에 Exception이 발생해도 나머지 작업을 지속하거나 해당 에러 건을 별도 처리하고 싶다면, 프로그램 내에서 적절하게 try~catch를 활용하여 Exception을 재 처리해야 한다.

[write 오류가 발생하면 작업 종료 됨]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemReader reader = factory.getItemReader("reader", SampleVo.class);
        // INSERT INTO SAMPLE (NO, NAME) VALUES (:no, :name);
        ItemWriter writer = factory.getItemWriter("writer");

        while(reader.next()) {
            SampleVo vo = reader.read();

            writer.write(vo);
        }

    }

}
```

[Key 중복 오류를 체크하여 작업 지속함]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemReader reader = factory.getItemReader("reader", SampleVo.class);
        // INSERT INTO SAMPLE (NO, NAME) VALUES (:no, :name);
        ItemWriter writer1 = factory.getItemWriter("writer1");
        // UPDATE SAMPLE SET NAME = :name WHERE NO = :no;
        ItemWriter writer2 = factory.getItemWriter("writer2");

        while(reader.next()) {
            SampleVo vo = reader.read();

            try {
                writer1.write(vo);
            } catch (SQLException e) {     // SQL 오류 중
                if(e.getErrorCode() == 1)  // Key 중복 오류(에러코드1)인 경우
                    writer2.write(vo);     // UPDATE 쿼리를 수행하여 업데이트
                else                       // 그 외 오류는
                    throw e;               // rethrow하여 작업을 중지시킴
            }
        }

    }

}
```

3.8 로그(Log)

배치 프로그램에서 출력되는 로그는 로그 내용을 쉽게 확인하기 위하여 각각의 Step 단위로 로그파일이 분리되어 기록된다

로그는 목적에 따라 다음과 같이 구분된다.

개발로그

- 프로그램 개발 시 디버그 목적으로 출력하는 로그 메시지
- 운영 단계(개발/테스트/운영)에 따라 출력 여부가 결정됨
- Logger의 debug()를 사용하여 메시지 출력

작업로그

- 배치 작업 수행 결과로 출력하는 로그 메시지
- 운영 단계와 상관없이 로그로 출력됨
- AbstractTask의 print()를 사용하여 메시지 출력

[로그 출력 예]

```
public class SimpleBat extends AbstractTask {

    // logger 선언
    private static final Logger logger = LoggerFactory.getLogger(SampleBat.class);

    @Override
    public void execute(ItemFactory factory) throws Exception {
```

```
        ItemReader reader = factory.getItemReader("reader", SampleVo.class);
        ItemWriter writer = factory.getItemWriter("writer");

        while(reader.next()) {
            SampleVo vo = reader.read();
            writer.write(vo);

            logger.debug("no={} name={}", vo.getNo(), vo.getName());  // 개발로그 출력 (Debug Level)
        }

        print("************** 실행 결과 **************");             // 작업 로그 출력
        print("* READ  횟수 : " + reader.getItemCount());
        print("* WRITE 횟수 : " + writer.getItemCount());
        print("****************************************");


    }

}
```

3.8.1 입출력 데이터 로그

Reader, Writer를 통해 처리되는 데이터 내용을 확인하고 싶은 경우 @show-log 속성을 "true"로 지정하여 입출력 데이터 내용을 로그로 출력할 수 있다.

입출력 데이터 로그는 대상이 파일인 경우 파일 내용을 출력하고 DB인 경우 쿼리 파라미터가 포함된 실행 쿼리를 출력한다.

> **Note**
>
> 입출력 데이터 로그도 개발 로그의 일종으로 운영단계에 따라 출력 여부가 결정된다.

다음은 @show-log를 지정한 후 출력되는 로그 메시지 예이다.

```
<job>

    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample.txt" show-log="true" />
            <writer id="writer" type="DB"  url="default" query-id="insert" show-log="true" />
        </resources>
    </step>

</job>
```

```
DEBUG ItemLogger : reader - 00005name_5    00500000000052013-05-28 19:00:48.662  address_5                    description_5

DEBUG ItemLogger : writer - insert into SAMPLE (no, name, age, salary, birthday, address, description) values ('5',
 'name_5', '5', '5', '2013-05-28 19:00:48.662', 'address_5', 'description_5')
DEBUG ItemLogger : reader - 00006name_6    00600000000062013-05-28 19:00:48.662  address_6                    description_6

DEBUG ItemLogger : writer - insert into SAMPLE (no, name, age, salary, birthday, address, description) values ('6',
 'name_6', '6', '6', '2013-05-28 19:00:48.662', 'address_6', 'description_6')
DEBUG ItemLogger : reader - 00007name_7    00700000000072013-05-28 19:00:48.662  address_7                    description_7

DEBUG ItemLogger : writer - insert into SAMPLE (no, name, age, salary, birthday, address, description) values ('7',
 'name_7', '7', '7', '2013-05-28 19:00:48.662', 'address_7', 'description_7')
... ...
... ...
```

3.9 배치 파라미터

배치 프로그램 내에서 AbstractTask가 제공하는 API를 활용하여 배치 파라미터 값을 조회, 변경하거나 신규 값을 등록할 수 있다.

다음은 배치 프로그램 내에서 배치 파라메터를 사용하는 예이다.

```
<job>

    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="DB"  url="default" query-id="select" />
            <writer id="writer" type="SAM" url="${base}/sample.${today}" />
        </resources>
    </step>

</job>
```

```
public class SimpleBat extends AbstractTask {

    @Override
    public void beforeStep() {
        // 스텝 파라미터 신규 등록
        // beforeStep()에서 등록된 파라미터는 작업정의에서 변수로 사용할 수 있음
        putStepParameter("today", "20130402");
    }

    @Override
    public void execute(ItemFactory factory) throws Exception {

        String base = getJobParameter("base");    // 작업정의에 등록된 작업 파라미터 조회
        print("Job Param = " + base);

        putJobParameter("filename", base + "/sample.out");  // 작업 파라미터 신규 등록. 이후 스텝에서 사용 가능

        ... ...
        ... ...

    }

}
```

3.10 재시작(재처리)

작업 실행 중 에러에 의해 작업이 실패했거나 실행 중지된 작업은 재시작 할 수 있다. (처리 완료된 작업은 재시작 할 수 없음)

실패했거나 중지된 작업을 재시작 하면 기존 실행에서 완료된 스텝은 다시 수행하지 않으며 에러가 발생했거나 중지된 스텝부터 다시 시작한다.

일반적으로 작업이나 스텝이 재시작 하는 경우 문제가 발생할 수 있기 때문에 재시작 가능한 작업과 스텝에 대해서는 @restartable 속성을 "true"로 설정하여 명시적으로 재시작 가능 여부를 지정해야 한다.

@restartable 속성이 "false"로 지정되어 있는 경우 해당 작업이나 스텝은 재시작 할 수 없다. (재시작 시 오류 발생)

[재시작 가능여부 지정]

```
<job restartable="true">
```

```
    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step" type="java" class="sample.batch.SimpleBat" restartable="true">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample.txt" />
            <writer id="writer" type="DB"  url="default" query-id="insert" commit-interval="1000" />
        </resources>
    </step>

    ... ...
    ... ...

</job>
```

3.10.1 중복처리 방지

작업을 재시작 하는 경우 기존에 처리했던 데이터를 중복으로 처리하면 문제가 되는 경우가 많다. 이를 방지하기 위해서는 재시작 지점을 스텝 파라미터로 저장하여 재시작 시 해당 스텝 파라미터의 값을 확인하여 재처리를 위한 비즈니스 로직을 수행해야 한다.

> **Note**
>
> 작업/스텝 파라미터는 커밋 시에 자동으로 저장되고 재시작 시에 마지막 저장 값으로 복구됨

다음은 처리 위치를 스텝 파라미터로 등록하여 재시작 시 기존에 처리했던 데이터를 Skip 처리하는 프로그램 예이다.

```
<job restartable="true">

    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step" type="java" class="sample.batch.SimpleBat" restartable="true">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample.txt" />
            <!-- 1,000건 단위로 자동 커밋을 수행함 -->
            <writer id="writer" type="DB"  url="default" query-id="insert" commit-interval="1000" />
        </resources>
    </step>

    ... ...
    ... ...

</job>
```

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemWriter writer = factory.getItemWriter("writer");
        ItemReader reader = factory.getItemReader("reader", SampleVo.class);

        // 스텝 파라미터를 조회. 최초 실행인 경우 등록된 파라미터가 없으므로 null을 반환함
        Long commitPos = getStepParameter("commitPos");

        if(commitPos == null {    // 최초 실행된 경우
            commitPos = 0l;
        } else {                  // 재시작 된 경우 read 위치를 변경 (skip 처리)
            for(long i=0; i<commitPos; i++) {
                reader.next();
```

```
            }
        }

        while(reader.next()) {
            SampleVo vo = reader.read();

            ... ...
            ... ...    // business 로직 처리
            ... ...

            putStepParameter("commitPos", commitPos++);  // 현재 위치를 스텝 파라미터로 등록
            writer.write(vo);  // 1,000건 write 시 자동 커밋이 되며 스텝 파라미터도 같이 저장됨
        }

    }

}
```

3.10.2 입출력 데이터 복구

배치 프로그램이 재시작 하는 경우 기본적으로 Reader는 입력 데이터의 첫 번째 줄(Row)부터 다시 읽으며 Writer는 기존 파일을 삭제하고 새로운 파일에 데이터를 기록하게 된다.

그러나 @save 속성을 "true"로 지정한 경우 Reader, Writer는 현재 처리 위치를 커밋 시에 자동으로 저장하고, 재시작 시 Reader는 마지막 위치로 자동으로 이동하여 읽기를 수행하고 Writer는 새 파일을 생성하지 않고 기존 파일의 마지막 위치부터 이어쓰게 된다(append).

> **Important**
>
> 재시작 시 입출력 데이터가 변경되는 경우에는 자동 복구 기능이 정상적으로 적용되지 않으니 유의한다.

```
<job restartable="true">

    <parameters>
        <parameter key="base" value="/sample/batch"/>
    </parameters>

    <step id="step" type="java" class="sample.batch.SimpleBat" restartable="true">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample.txt" save="true" />
            <writer id="writer" type="SAM" url="${base}/sample.out" save="true" commit-interval="1000" />
        </resources>
    </step>

</job>
```

4장. 작업 실행

4.1 로컬 PC 실행

개발된 배치 작업을 로컬 PC에서 실행하기 위해서는 BatchTestUtils 클래스가 제공하는 API를 사용하여 지정된 작업 또는 스텝을 실행할 수 있다.

> **Note**
>
> 반복적인 실행 테스트를 위하여 JUnit 테스트 케이스로 작성하는 것을 권장함

[로컬 PC 작업실행 예]

```
import com.anyframe.batch.test.BatchTestUtils;

public class TestJob {

    @Test
    public void run() {
        BatchTestUtils.launchJob("sample/Simple_cfg.xml", new String[]{"today="20130402"});
    }

}
```

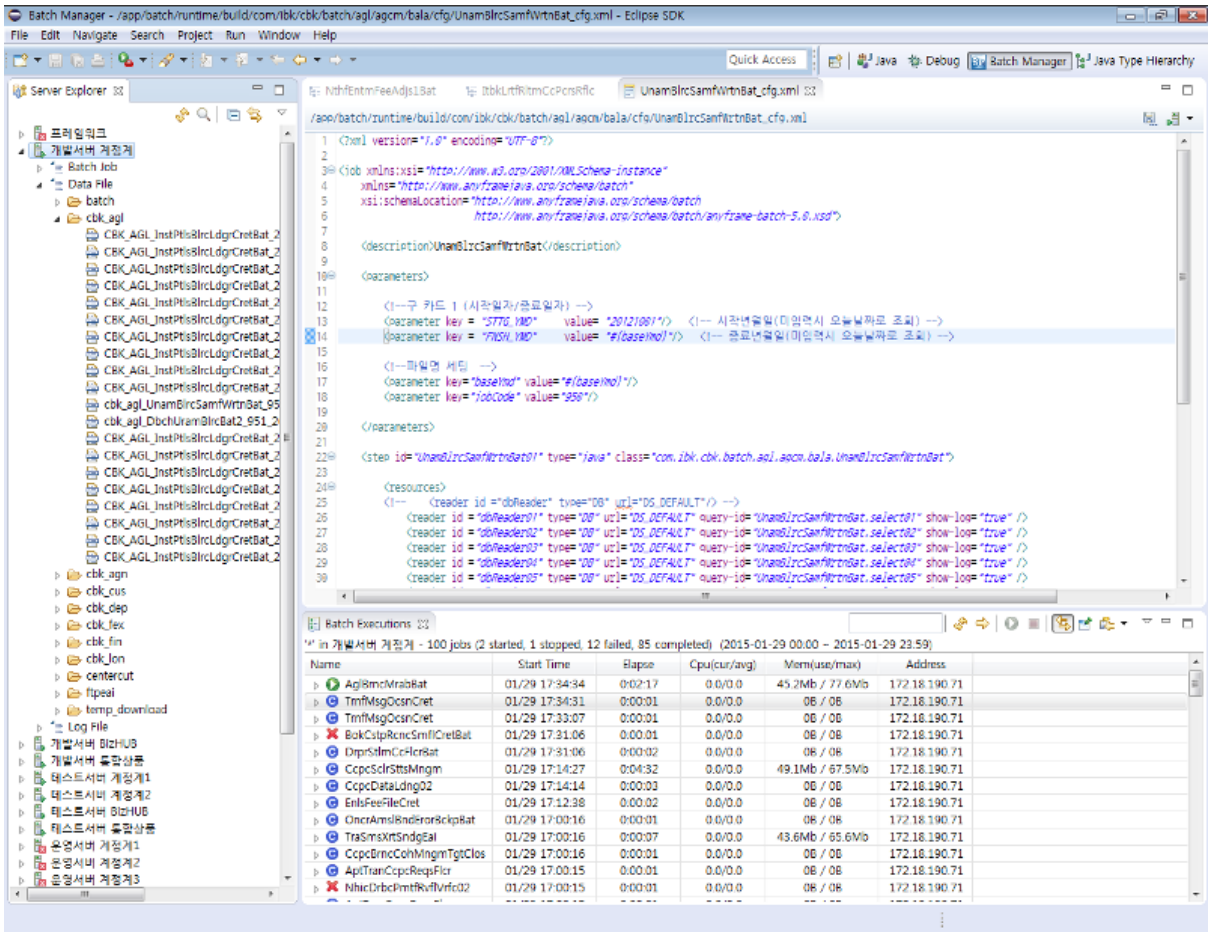
BatchTestUtils 클래스가 제공하는 API는 다음과 같다.

API	설명
launchJob(String location, String[] params)	배치 작업을 실행시킴 • location: 작업정의(CFG) 위치 (classpath 기준) • params: key=value 형태의 작업 파라메터
launchStep(String location, String stepName, String[] params)()	배치 작업의 특정 스텝만 실행시킴 • location: 작업정의(CFG) 위치 (classpath 기준) • stepName: 실행할 스텝 이름 • params: key=value 형태의 작업 파라메터

4.2 서버 실행

개발된 배치 작업을 서버에서 실행하기 위해서는 배치 개발도구인 배치 매니저를 사용한다. (개발된 배치 프로그램은 빌드되어 서버에 배포되어 있어야 함)

> **Note**
>
> 배치 매니저의 사용법은 배치 매니저 사용가이드를 참조



5장. 온라인 모듈 사용

5.1 개요

배치 프로그램 개발 시 온라인 프로그램에서 개발된 모듈을 배치에서 사용할 수 있다.

배치에서 사용하는 온라인 모듈의 유형은 다음과 같다.

분류	항목	설명
서비스 호출	Service	• 온라인의 서비스를 배치에서 실행 • 호출된 온라인 서비스는 건 별로 독립된 Global ID를 가지며 온라인에서 처리되는 것과 동일하게 처리 • 온라인 서비스의 업무 선후처리 및 서비스에서 축적한 업무로그도 배치 트랜잭션 기준으로 처리
모듈 사용	Biz BizUtil	• 온라인에서 개발된 비즈니스 로직을 배치에서 재사용하기 위하여 온라인 모듈을 배치에서 사용
	DAO DSO	• 쿼리를 수행하여 DB 테이블을 읽거나 쓰기 위해서 온라인 모듈에서 개발된 DAO/DSO 모듈을 사용

배치에서 호출되는 모든 온라인 모듈의 트랜잭션은 배치 트랜잭션에 동기화 된다. 즉, 온라인 모듈(Service, Biz, Dao)를 통해 수행된 쿼리는 배치에서 커밋을 수행하는 시점에 데이터베이스에 반영된다. 또한 트랜잭션 커밋 시에 수행되는 후처리(업무로그)도 배치 트랜잭션이 커밋 되는 시점에 처리된다.

일반적으로 온라인 모듈에서는 업무로그와 같이 공유메모리(SGC) 영역에 적재되는 데이터가 많으므로 커밋 간격을 크게 할 경우 메모리 부족 문제가 발생할 수 있으므로 일반적인 배치 프로그램보다는 커밋 간격을 작게 해야 한다.

5.2 공통정보 설정

기본적으로 온라인 모듈은 표준 전문을 기반으로 작동을 하도록 구성되어 있으며 표준전문 내에 포함된 공통정보를 참조하는 경우가 많다. 이런 공통정보는 단말 시스템 같은 최초 요청 시스템에서 정보가 설정되는 경우가 일반적이다.

따라서 배치에서 온라인 모듈을 호출할 경우에는 이런 공통정보를 설정해야 호출된 온라인 모듈이 정상적으로 실행될 수 있다.

배치에서 온라인 모듈을 호출할 때 필요한 정보는 배치 작업정의(CFG)에서 작업 파라메터로 지정할 수도 있으며 배치 프로그램 내에서 API를 통해서 지정할 수도 있다. 공통정보 관련 정보를 설정한 경우 해당 배치 작업에서 호출되는 모든 온라인 모듈은 설정된 공통 정보를 기준으로 동작하게 된다.

[작업정의(CFG)에 공통정보 설정]

```
<job>

    <parameters>
        <parameter key="com.anyframe.batch.brcd" value="0012"/>
        <parameter key="com.anyframe.batch.tln" value="BA000123"/>
        <parameter key="com.anyframe.batch.xcd" value="CBKCOM935410000"/>
```

```
        <parameter key="com.anyframe.batch.emn" value="000123"/>
        <parameter key="com.anyframe.batch.ymd" value="20131031"/>
        <parameter key="com.anyframe.batch.ip" value="127.0.0.1"/>
    </parameters>

    <step id="step1" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample1.txt" />
            <writer id="writer" type="SAM" url="${base}/sample1.out" />
        </resources>
    </step>

</job>
```

5.3 서비스 호출(SERVICE)

배치에서 온라인 서비스(Service) 호출은 온라인 클래스가 제공하는 메써드를 사용한다.

배치 프로그램에서 온라인 서비스를 호출하는 예는 다음과 같다.

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        SampleInSvo inVo = new SampleInSvo();

        SampleOutSvo outVo = OnlineSupport.executeService(ISampleSvc.class, "CBKCOM935410", "000", inVo, false);

        ... ...
        ... ...

    }

}
```

5.4 모듈 사용(BIZ)

배치 프로그램에서 Biz 모듈의 호출은 온라인 프로그램에서 사용하는 것과 동일하다. 업무 내에 포함된 Biz 클래스는 해당 클래스를 new 해서 사용한다.

[온라인 BIZ 모듈 사용]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        MySampleBiz mySampleBiz = new MySampleBiz();
        mySampleBiz.validate();

        ... ...
        ... ...

    }

}
```

5.5 모듈 사용(DAO)

배치 프로그램에서 DAO 모듈의 호출은 온라인 프로그램에서 사용하는 것과 동일하다. 사용하고자 하는 DAO 클래스의 getInstance()를 호출하여 DAO 객체를 획득한 후 사용한다.

[온라인 DAO 모듈 사용]

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        // DAO를 호출할 경우
        SampleDvo dvo = new SampleDvo();
        BizWorkLogDao dao = BizWorkLogDao.getInstance();
        int ret = dao.insertTbCbkFrw010m(dvo);

        ... ...
        ... ...

    }

}
```

DAO 클래스가 제공하는 API 중에 전체 조회(selectListXXX) API는 결과 데이터 전체를 List 형태로 반환하기 때문에 데이터 건수가 많은 경우에는 메모리 사용량이 증가하게 되어 에러가 발생할 수 있다. 이 경우에는 페이징 조회(selectPageXXX)를 사용하거나 배치의 DB Reader를 사용하여 데이터를 조회해야 한다.

5.6 모듈 사용(DAO Writer)

배치 프로그램 내에서 DAO 객체를 직접 사용하여 테이블 변경작업을 수행한 경우에는 데이터 처리 건수가 배치 실행 이력에 기록되지 않는다. DAO 유형의 Writer를 선언하여 사용하면 일반 Writer와 동일하게 방법으로 처리할 수 있으며 처리 건수도 실시간으로 확인할 수 있다.

[DAO Writer 정의]

```
<job>

    <step id="step" type="java" class="sample.batch.SimpleBat">
        <resources>
            <reader id="reader" type="SAM" url="${base}/sample.txt" />
            <writer id="writer" type="DAO" url="sample.dao.SampleDao.insert()" />
        </resources>
    </step>

</job>
```

```
public class SimpleBat extends AbstractTask {

    @Override
    public void execute(ItemFactory factory) throws Exception {

        ItemReader reader = factory.getItemReader("reader", SampleVo.class);
        ItemWriter writer = factory.getItemWriter("writer");

        while(reader.next()) {
            SampleVo vo = reader.read();
            writer.write(vo);  // SampleDao.insert()가 수행됨
        }
```

```
    }

}
```

5.7 원격작업 호출 (온라인 배치)

온라인 프로그램에서 원격으로 배치 작업을 호출하여 실행할 수 있다.

온라인 프로그램에서 RemoteJobLauncher을 사용하여 실행시킬 작업과 파라미터를 전달하여 실시간으로 배치작업을 기동시킬 수 있다.

> **Important**
>
> 과도한 배치작업 실행은 서버에 악영향을 초래하므로 사전에 협의가 필요함

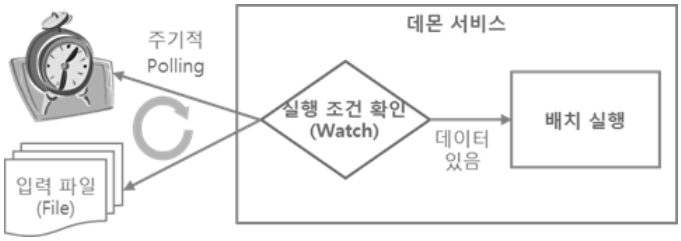
다음은 온라인에서 배치를 호출한 예이다.

```
public class SimpleBiz {

    public void run() {

        // 첫번째 인자로 작업정의(CFG) 위치 전달
        // 두번째 인자로 배치 파라미터 전달
        RemoteJobLauncher launcher = new RemoteJobLauncher("40.10.33.111:9001");
        launcher.launch("sample/batch/hello/cfg/BSDT_CFG.xml", new String[]{"bsdt=20140101","-type=online"});

    }

}
```

6장. 데몬 배치

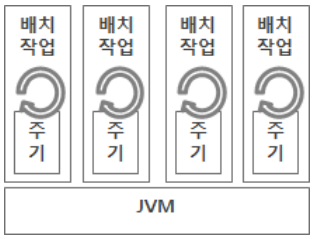
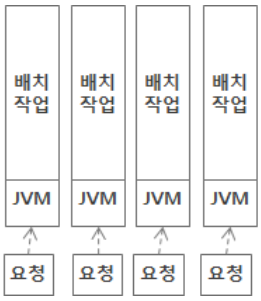
6.1 개요

데몬 배치는 일정 주기 또는 파일 생성 여부를 감지하여 지정된 배치 프로그램이 자동 실행되는 형태이다.



데몬으로 실행되는 배치는 별도의 데몬 프로세스 내에서 실행되며 여러개의 데몬 배치를 하나의 데몬 프로세스 내에서 처리할 수 있다. (배치 관리도구에서 등록)

일반 배치와 달리 데몬 프로세스는 상시적으로 구동되어 있으며 데몬 배치는 데몬 프로세스 내에서 Thread 형태로 기동되므로 배치 기동에 필요한 초기화 비용이 작다. 이런 특성으로 인하여 데몬 배치는 일반적으로 준 실시간처리(Deferred)를 위하여 주로 사용된다.

데몬 배치	일반 배치
	

6.2 데몬 배치 유형

데몬 배치는 실행 조건에 따라 다음과 같이 구분한다.

유형	설명	비고
파일 데몬	지정된 파일 생성 여부를 감지하여 배치 작업을 구동	EAI/FEP 파일 수신 후처리 로그 파일 후처리
타이머 데몬	지정된 시간 간격으로 배치 작업을 구동	준 실시간 후행 아키텍처 주기적 호출
스케줄 데몬	지정된 시간에 배치 작업을 구동	특정시간 호출

6.2.1 파일 데몬

파일 데몬은 입력 폴더에 지정된 파일이 생성되면 이를 감지하여 배치 작업을 구동시키는 형태이다.

처리할 파일은 Regular Expression 형태로 지정할 수 있으며 입력파일은 중복처리을 방지하기 위하여 파일명 뒤에 생성일자(TIMESTAMP)를 붙여 처리폴더로 이동되어 처리된다.

입력파일의 파일명(Absolute Path)은 "daemon.input" 파라미터를 통해서 실행되는 배치로 전달된다. 따라서 다음과 같이 작업정의(CFG)에서 파라미터를 사용하여 입력 파일을 사용할 수 있다.

> **Note**
>
> "daemon.input" 파라미터로 전달되는 파일명은 원본 파일이 처리폴더로 이동된 [파일명.생성일자] 형태로 변경된 파일명임

```
<job>

    <step id="step" type="java" class="sample.batch.SampleBat">
        <resources>
            <!-- 파일 데몬 실행 시 변경된 파일명이 전달됨 ex) input.txt.2013-11-25T190235.418 -->
            <reader id="reader" type="SAM" url="${daemon.input}" />
            <writer id="writer" type="DB"  url="default" />
        </resources>
    </step>

</job>
```

6.2.2 타이머 데몬

타이머 데몬은 지정된 시간 간격에 따라 배치 작업을 구동시킨다. 따라서 주기적으로 데이터를 확인해서(Polling) 처리해야 하는 후행처리 아키텍처로 주로 사용된다.

타이머 데몬의 시간 간격은 절대 간격과 상대 간격이 있다. 절대 간격은 배치 실행시간과 상관없이 계산되고 상대 간격은 배치 실행 종료 시점을 기준으로 계산된다.

> **Important**
>
> 절대 간격이라도 배치 실행시간이 절대 간격을 초과하는 경우, 중복 실행되는 것이 아니라 배치 종료 후 바로 다시 배치가 기동됨

6.2.3 스케줄 데몬

스케줄 데몬은 지정된 시간에 배치 작업을 구동한다.

시간은 Cron Expression으로 지정한다

6.3 데몬 배치 개발

데몬 배치도 일반 배치와 배치 프로그램은 동일하다. 따라서 데몬 배치용 프로그램도 일반 배치 프로그램 개발과 동일한 방법으로 개발한다.

> **Note**
>
> 개발 완료된 배치는 관리도구에서 데몬으로 등록하여 데몬으로 싱행됨

6.4 데몬 배치 실행

데몬을 테스트 목적으로 로컬 PC에서 실행하기 위해서는 DaemonTestUtils 클래스가 제공하는 API를 사용하여 데몬을 실행할 수 있다. 데몬은 독립된 프로세스로 실행되어야 하므로 main() 메써드 내에서 DaemonTestUtil을 통해 실행시켜야 한다.

> **Note**
>
> 서버 실행은 배치 관리도구를 통하여 데몬 등록 후 실행됨

[로컬 PC 실행]

```
import com.anyframe.daemon.test.DaemonTestUtils;

public class DaemonTest {

    public static void main(String[] args) {

        String location = "test/daemon/file/daemon_cfg.xml";
        String inputDir = "/daemon/input";
        String filename = "input[0-9]*\\.txt";
        String outputDir = /daemon/output";
        String interval = 10000;

        // daemon_cfg.xml을 파일 데몬으로 실행
        DaemonTestUtils.launchDaemonFile(location, inputDir, filename, outputDir, interval);

    }

}
```

BatchTestUtils 클래스가 제공하는 API는 다음과 같다.

API	설명
launchDaemonFile(String location, String inputDir, String filename, String outputDir, long interval)	파일 데몬을 실행시킴 • location - 배치 작업정의(CFG) 위치(classpath 기준) • inputDir - 입력 디렉토리 위치 • filename - 감지할 파일명(Regular Expression 사용 가능) • outputDir - 출력 디렉토리 위치(처리 완료된 파일 위치) • interval - 파일 조회 간격(ms)
launchDaemonTimer(String location, long interval)	타이머 데몬을 실행시킴 • location - 배치 작업정의(CFG) 위치(classpath 기준) • interval - 실행 간격(ms)

7장. Appendix 배치 예제

7.1 단순 유형

다음은 하나의 입력 데이터를 읽어 하나의 출력 데이터를 처리하는 배치 프로그램 샘플이다. 샘플용 데이터를 생성하는 프로그램과, 생성된 데이터를 간단한 로직을 적용하여 처리하는 프로그램은 공통으로 사용하며 작업 정의(XML)에서 입출력 데이터 유형에 따른 Reader, Writer 선언을 통해 다양한 유형의 데이터를 처리할 수 있다.

[입출력 맵핑(VO)]

```
@LocalName("샘플 Vo")
public class SampleVo extends AbstractVo {

 /**
  * 일련번호 정보
  */
 @LocalName("일련번호")
 @Length(10)
 private int srn;

 /**
  * 해당 거래의 고객명
  */
 @LocalName("성명")
 @Length(100)
 @Scale(0)
 private String srnmNm;

 /**
  * 연령 상각완료자 연령(법인 연령은 무시)
  */
 @LocalName("연령")
 @Length(3)
 @Scale(0)
 private int age;

 /**
  * 주소
  */
 @LocalName("주소")
 @Length(200)
 @Scale(0)
 private String adr;

 /**
  * 일련번호 Getter Method
  *
  * @return 일련번호
  */
 @LocalName("일련번호 Getter Method")
 public int getSrn() {
  this.srn = (Integer) super.getValue(0);
  return this.srn;
 }

 /**
  * 일련번호 Setter Method
  *
  * @param int 일련번호
  */
 @LocalName("일련번호 Setter Method")
 public void setSrn(int srn) {
```

```
 super.setValue(0, srn);
 this.srn = srn;
}

/**
 * 성명 Getter Method
 *
 * @return 성명
 */
@LocalName("성명 Getter Method")
public String getSrnmNm() {
 this.srnmNm = super.getValue(1);
 return this.srnmNm;
}

/**
 * 성명 Setter Method
 *
 * @param String
 *            성명
 */
@LocalName("성명 Setter Method")
public void setSrnmNm(String srnmNm) {
 super.setValue(1, srnmNm);
 this.srnmNm = srnmNm;
}

/**
 * 연령 Getter Method
 *
 * @return 연령
 */
@LocalName("연령 Getter Method")
public int getAge() {
 this.age = super.getValue(2);
 return this.age;
}

/**
 * 연령 Setter Method
 *
 * @param BigDecimal
 *            연령
 */
@LocalName("연령 Setter Method")
public void setAge(int age) {
 super.setValue(2, age);
 this.age = age;
}

/**
 * 주소 Getter Method
 *
 * @return 주소
 */
@LocalName("주소 Getter Method")
public String getAdr() {
 this.adr = super.getValue(3);
 return this.adr;
}

/**
 * 주소 Setter Method
 *
 * @param String
 *            주소
 */
@LocalName("주소 Setter Method")
public void setAdr(String adr) {
 super.setValue(3, adr);
 this.adr = adr;
```

```
 }

}
```

[배치 프로그램 - 데이터 생성]

```
/**
 * 샘플용 데이터를 생성하여 기록하는 배치 프로그램
 */
public class CreateBat extends AbstractTask {

 @Override
 public void execute(ItemFactory factory) throws Exception {

   ItemWriter writer = factory.getItemWriter("vsamWriter01");

   String param = this.getJobParameter("count");
   int count = Integer.valueOf(param);

   for(int index=0; index<count; index++) {
    SampleVo vo = new SampleVo();

    vo.setSrn(index);
    vo.setAge(index);
    vo.setSrnmNm("name");
    vo.setAdr("address");

   writer.write(vo);
   }

   print("************** 실행 결과 **************");
  print("* WRITE 횟수 : " + writer.getItemCount());
  print("************************************");
 }

}
```

[배치 프로그램 - 1:1 처리]

```
/**
 * 입력 데이터를 읽어 단순한 비즈니스 로직 처리(no 값 비교) 후 결과를 출력하는 배치 프로그램
 *
 */
public class ProcessBat extends AbstractTask {

 private static final Logger LOGGER = LoggerFactory.getLogger(ProcessBat.class);

 @Override
 public void execute(ItemFactory factory) throws Exception {

  ItemWriter writer = factory.getItemWriter("samWriter01");
  ItemReader reader = factory.getItemReader("samReader01", SampleVo.class);

  while(reader.next()) {

   SampleVo vo = reader.read();

   if(vo.getSrn() >= 5) {
    writer.write(vo);
   }

  }

   print("************** 실행 결과 **************");
   print("* READ  횟수 : " + reader.getItemCount());
  print("* WRITE 횟수 : " + writer.getItemCount());
  print("************************************");
 }
```

```
}
```

배치 쿼리

```
<?xml version="1.0" encoding="UTF-8"?>

<sql>

 <query id="sample.batch.simple.insert01">
  insert into SAMPLE (SRN, SRNM_NM, AGE, ADR) values  (:srn, :srnmNm, :age, :adr)
 </query>

 <query id="sample.batch.simple.selec01t">
  SELECT * FROM SAMPLE
 </query>
</sql>
```

[작업 정의 - SAM TO SAM]

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.anyframejava.org/schema/batch"
 xsi:schemaLocation="http://www.anyframejava.org/schema/batch
                     http://www.anyframejava.org/schema/batch/anyframe-batch-5.0.xsd">

 <description>유형 별 샘플 배치 작업(SAM to SAM)</description>

 <parameters>
  <parameter key="base" value="/sample/batch/simple/sam2sam"/>
  <parameter key="count" value="10"/>
 </parameters>

 <step id="CreateBat01" type="java" class="sample.batch.simple.CreateBat">
  <resources>
   <writer id="samWriter01" type="SAM" url="${base}/input.txt" show-log="true"/>
 </resources>
 </step>

 <step id="ProcessBat01" type="java" class="sample.batch.simple.ProcessBat">
    <resources>
   <reader id="samReade01" type="SAM" url="${base}/input.txt" show-log="true"/>
   <writer id="samWriter01" type="SAM" url="${base}/output.txt" show-log="true"/>
    </resources>
 </step>

</job>
```

[작업 정의 - VSAM TO VSAM]

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.anyframejava.org/schema/batch"
 xsi:schemaLocation="http://www.anyframejava.org/schema/batch
                     http://www.anyframejava.org/schema/batch/anyframe-batch-5.0.xsd">

 <description>유형 별 샘플 배치 작업(VSAM to VSAM)</description>

 <parameters>
  <parameter key="base" value="/sample/batch/simple/vsam2vsam"/>
  <parameter key="count" value="10"/>
 </parameters>

 <step id="CreateBat01" type="java" class="sample.batch.simple.CreateBat">
     <resources>
   <writer id="vsamWriter01" type="VSAM" url="${base}/input.txt" show-log="true"/>
```

```
    </resources>
 </step>

 <step id="ProcessBat01" type="java" class="sample.batch.simple.ProcessBat">
   <resources>
    <reader id="vsamReader01" type="VSAM" url="${base}/input.txt" show-log="true"/>
    <writer id="vsamWriter01" type="VSAM" url="${base}/output.txt" show-log="true"/>
 </resources>
 </step>

</job>
```

[작업 정의 - SAM TO DB]

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.anyframejava.org/schema/batch"
 xsi:schemaLocation="http://www.anyframejava.org/schema/batch
                     http://www.anyframejava.org/schema/batch/anyframe-batch-5.0.xsd">

 <description>유형 별 샘플 배치 작업(SAM to DB)</description>

 <parameters>
  <parameter key="base" value="/sample/batch/simple/sam2db"/>
  <parameter key="count" value="10"/>
 </parameters>

 <step id="CreateBat01" type="java" class="sample.batch.simple.CreateBat">
  <resources>
   <writer id="samWriter01" type="SAM" url="${base}/input.txt" show-log="true"/>
 </resources>
 </step>

 <step id="ProcessBat01" type="java" class="sample.batch.simple.ProcessBat">
  <resources>
   <reader id="samReader01" type="SAM" url="${base}/input.txt" show-log="true"/>
   <writer id="dbWriter01" type="DB" url="DS_DEFAULT" query-id="sample.batch.ProcessBat.insert01" show-log="true"/>
 </resources>
 </step>

</job>
```

[작업 정의 - DB TO SAM]

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.anyframejava.org/schema/batch"
 xsi:schemaLocation="http://www.anyframejava.org/schema/batch
                     http://www.anyframejava.org/schema/batch/anyframe-batch-5.0.xsd">

 <description>유형 별 샘플 배치 작업(DB to SAM)</description>

 <parameters>
  <parameter key="base" value="/sample/batch/simple/db2sam"/>
 </parameters>

 <step id="ProcessBat01" type="java" class="sample.batch.simple.ProcessBat">
  <resources>
   <reader id="dbReader01" type="DB" url="DS_DEFAULT" query-id="sample.batch.ProcessBat.select01" show-log="true"/>
   <writer id="samWriter01" type="SAM" url="${base}/output.txt" show-log="true"/>
 </resources>
 </step>

</job>
```

7.2 1:N 유형 (Split)

다음은 하나의 입력 파일을 읽어 조건에 따라 두 개의 출력 파일에 기록하는 배치 프로그램 샘플이다.

[배치 프로그램 - 1:N 처리]

```
/**
 * 입력 데이터를 읽어 홀수/짝수 열을 기준으로 두개의 결과파일을 생성하는 샘플 프로그램
 *
 */
public class ProcessBat extends AbstractTask {

 @SuppressWarnings("unused")
 private static final Logger LOGGER = LoggerFactory.getLogger(ProcessBat.class);

 @Override
 public void execute(ItemFactory factory) throws Exception {

  ItemReader reader = factory.getItemReader("samReader01", SampleVo.class);
  ItemWriter writer1 = factory.getItemWriter("samWriter01");
  ItemWriter writer2 = factory.getItemWriter("samWriter02");

  while(reader.next()) {

   SampleVo vo = reader.read();

   if(vo.getSrn() % 2 == 0) {
    writer1.write(vo);
   } else {
    writer2.write(vo);
   }

  }

  this.getJobName();

   print("************** 실행 결과 **************");
   print("* READ    횟수 : " + reader.getItemCount());
  print("* WRITE1 횟수 : " + writer1.getItemCount());
  print("* WRITE2 횟수 : " + writer2.getItemCount());
  print("**********************************");
 }
}
```

[작업 정의 - 1:N]

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.anyframejava.org/schema/batch"
 xsi:schemaLocation="http://www.anyframejava.org/schema/batch
                     http://www.anyframejava.org/schema/batch/anyframe-batch-5.0.xsd">

 <description>1:N(Split) 유형의 샘플</description>

 <parameters>
  <parameter key="base" value="/sample/batch/split"/>
  <parameter key="count" value="10"/>
 </parameters>

 <step id="CreateBat01" type="java" class="sample.batch.split.CreateBat">
  <resources>
   <writer id="samWriter01" type="SAM" url="${base}/input.txt"/>
  </resources>
 </step>

 <step id="ProcessBat01" type="java" class="sample.batch.split.ProcessBat">
  <resources>
```

```
    <reader id="samReader01" type="SAM" url="${base}/input.txt"/>
    <writer id="samWriter01" type="SAM" url="${base}/output1.txt"/>
    <writer id="samWriter02" type="SAM" url="${base}/output2.txt"/>
   </resources>
  </step>

</job>
```

7.3 Header & Tail 처리

다음은 하나의 입력 파일을 읽어 조건에 따라 두 개의 출력 파일에 기록하는 배치 프로그램 샘플이다.

[Header, Body, Tail 구조의 파일 생성 프로그램]

```
/**
 * 샘플 데이터를 생성하여 기록하는 배치 프로그램
 * Header, Body, Tail 형태로 구성된 파일을 생성
 */
public class CreateBat extends AbstractTask {

 @Override
 public void execute(ItemFactory factory) throws Exception {

   ItemWriter writer = factory.getItemWriter("samWriter01");


   // write header
   HeaderVo header = new HeaderVo();

   header.setName("header");
   header.setDescription("sample header description");

   writer.write(header);


   // write body (loop)
   int count = 0;
   BigDecimal totalAmount = new BigDecimal(0);

   for(int index=0; index<10; index++) {
    BodyVo body = new BodyVo();

    body.setNo(index);
    body.setName("name_" + index);
    body.setAge(index);
    body.setSalary(new BigDecimal(index));
    body.setBirthday(new Timestamp(System.currentTimeMillis()));
    body.setAddress("address_" + index);
    body.setDescription("description_" + index);

    count++;

    totalAmount = totalAmount.add(body.getSalary());
   writer.write(body);
   }

   // write tail
   TailVo tail = new TailVo();

   tail.setCount(count);
   tail.setTotalAmount(totalAmount);

   writer.write(tail);
```

```
   print("************** 실행 결과 **************");
  print("* WRITE 횟수 : " + writer.getItemCount());
  print("************************************");
 }

}
```

[Header, Body, Tail 구조의 파일 처리 프로그램]

```
/**
 * Header, Body, Tail 형태로 구성된 파일을 읽어 처리하는 배치 프로그램
 * setMapping() API를 통하여 맵핑을 변경할 수 있으며
 * isLast() API를 통하여 마지막 줄을 확인할 수 있다
*/
public class ProcessBat extends AbstractTask {

 @SuppressWarnings("unused")
 private static final Logger LOGGER = LoggerFactory.getLogger(ProcessData.class);

 @Override
 public void execute(ItemFactory factory) throws Exception {

  ItemWriter writer = factory.getItemWriter("samWriter01");
  ItemReader reader = factory.getItemReader("samReader01");

  //HeaderVo로 맵핑
  reader.setMapping(HeaderVo.class);

  reader.next();
  HeaderVo header = reader.read();

  print("HEADER: " + header);

  // BodyVo로 맵핑
  reader.setMapping(BodyVo.class);

  // 마지막 줄 전까지는 BodyVo로 read
  while(reader.next() && !reader.isLast()) {

   BodyVo body = reader.read();

   if(body.getNo() >= 5) {
    writer.write(body);
   }
  }

  // 마지막 줄은 TailVo로 맵핑
  reader.setMapping(TailVo.class);

  TailVo tail = reader.read();
  print("TAIL: " + tail);


   print("************** 실행 결과 **************");
   print("* READ  횟수 : " + reader.getItemCount());
  print("* WRITE 횟수 : " + writer.getItemCount());
  print("************************************");
 }
}
```