バッチ・フレームワーク（SpringBatch/Maven, MyBatis）

内容

バッチ・フレームワーク（SpringBatch/Maven，MyBatis）

はじめに

Java EE 7 の jBatch 導入で Java によるバッチ処理が規格化されました。

jBatch が参考にしたという Spring Batch と Spring Batch 開発のツール／ライブラリの使い方について、関連ドキュメントが少なくて分かりづらい部分を初心者向けに纏めました。

1. 使用する資材

本資料では、以下の資材・バージョンで確認をしています。

1.1. 資材と取得先

#	名前/バージョン*1	説明・用途	取得先
1	JDK11	Maven build で使用 （Java 8 以降）	https://adoptopenjdk.net/
2	Maven 3.8.1	プロジェクトのビルド等	https://maven.apache.org/
3	spring-tool-suite(STS) 4-4.10	Eclipse を土台とした Spring 開発用の IDE	https://spring.io/tools
4	PostgreSQL 13.3-1	データ格納	https://www.postgresql.org/download/
5	H2 Database 1.4.200 (2019-10-14)	データ格納	https://www.h2database.com/

*1：使用したバージョン。メジャーバージョン番号が変わると動作が変わる可能性が有ります

1.2. セットアップ

1.2.1. インストールパス

各資材のインストール先はデフォルトでも構いませんが、Windows の場合は“Program△Files”のようにパスに空白を含んで扱いづらくなるためツール用のフォルダを作り以下のようにします。

[例] C:¥Tools¥*資材名*

※インストーラが表示しているインストール先パスの一部を“Program△Files”⇒“Tools”と書き換えてください。

zip 形式の資材は解凍して、Tools フォルダに格納します。

[例] C:¥Tools¥apache-maven-3.8.1

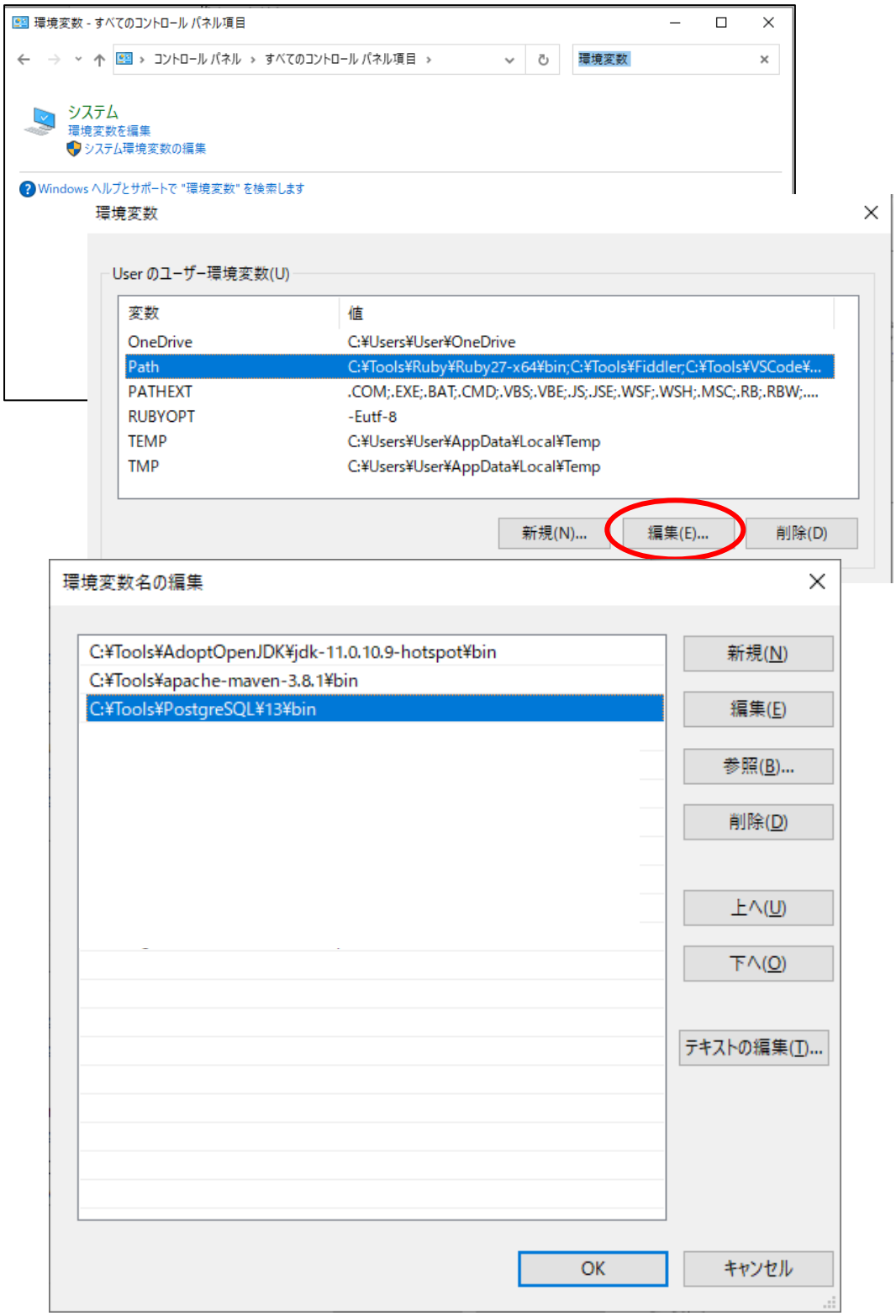
1.2.2. PostgreSQL のサービス

PostgreSQL をインストーラで入れるとサービスが常駐します。使わないときは、Windows 管理ツール ＞ コンピュータの管理 ＞ サービス から postgresql～ を停止し、「スタートアップの種類」を“手動”に変えてください。

1.2.3. 環境変数（path）

環境変数 Path に JDK、Maven、PostgreSQL の bin フォルダのパスを加えます。

インストーラを使った場合は設定済の場合があるので、先にコマンドプロンプトから set コマンドを実行して Path の内容を確認してください。設定済であればこの作業は不要です。



1.2.4. spring-tool-suite(STS)の設定

STS を起動するとワークスペースの場所を問い合わせてきます。プロジェクト等の指定がなければ、利用者のホームディレクトリに workspace の名前で作成します。既存の Eclipse があり名前が重複する場合は、“sts-workspace”等に変更しても正常に動作します。

（1）Maven プラグイン

項番 1.1 の STS には Maven プラグインが同梱されており、STS の Packatge Explorer や Project Explorer の”Run As >”等から mvn コマンドを実行することができますが、コマンドプロンプトから mvn コマンドを実行するために STS の外に Maven をインストールします。

それぞれ環境設定は以下から行います。

① STS-Maven プラグイン

STS の Window メニュー ＞Preferences＞Maven から

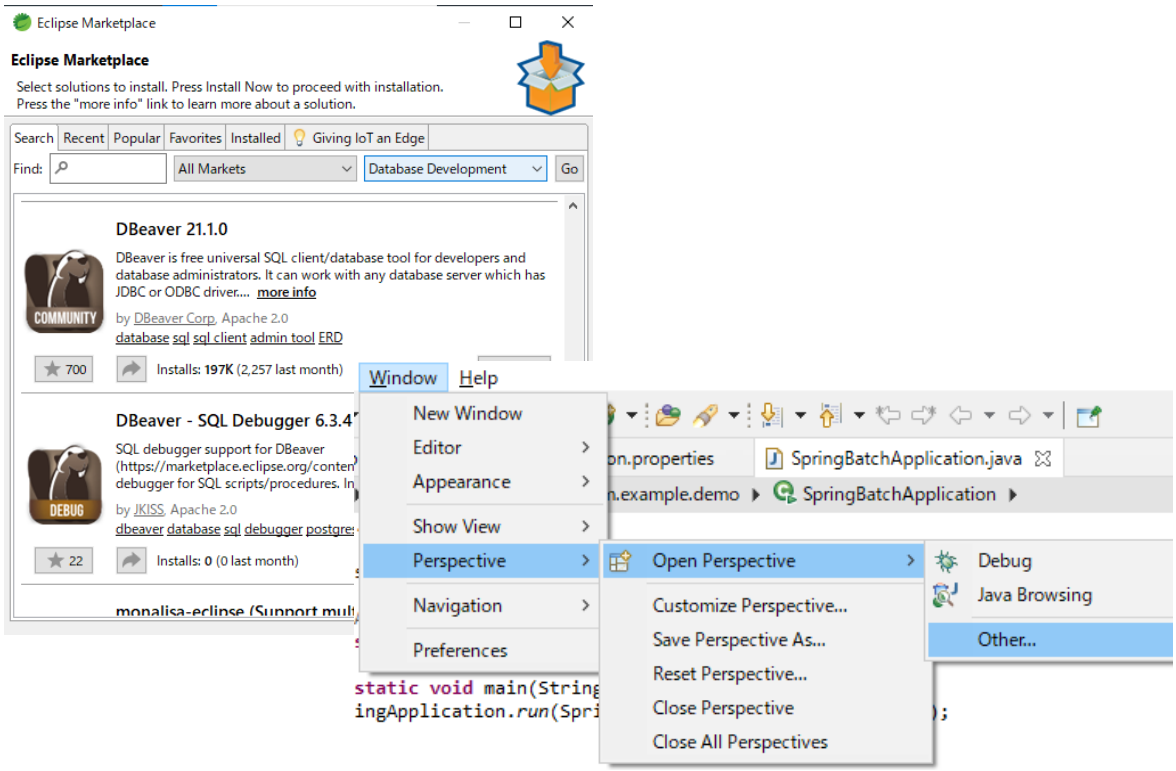
② 個別インストール

インストール先フォルダ（C:¥Tools¥apache-maven-3.8.1）¥conf¥settings.xml です。

（2）データベース開発プラグイン（DBeaver）

STS の中で使える RDB アクセス用のプラグインがいくつかあります。

インストール:Help メニュー ＞Eclipse Merketplace... より DBeaver 等がインストールできます。



画面表示：Window メニュー ＞Perspective＞Open Perspective＞Other...＞**DBeaver** を選択

※DBever はデスクトップ版もあります。https://dbeaver.io/download/

〔コミュニティ版 It is free and open source (license:Apache 2.0).〕

※STS は Eclipse を母体にしているので Pleiades プラグインによる日本語化も紹介[1]されています。

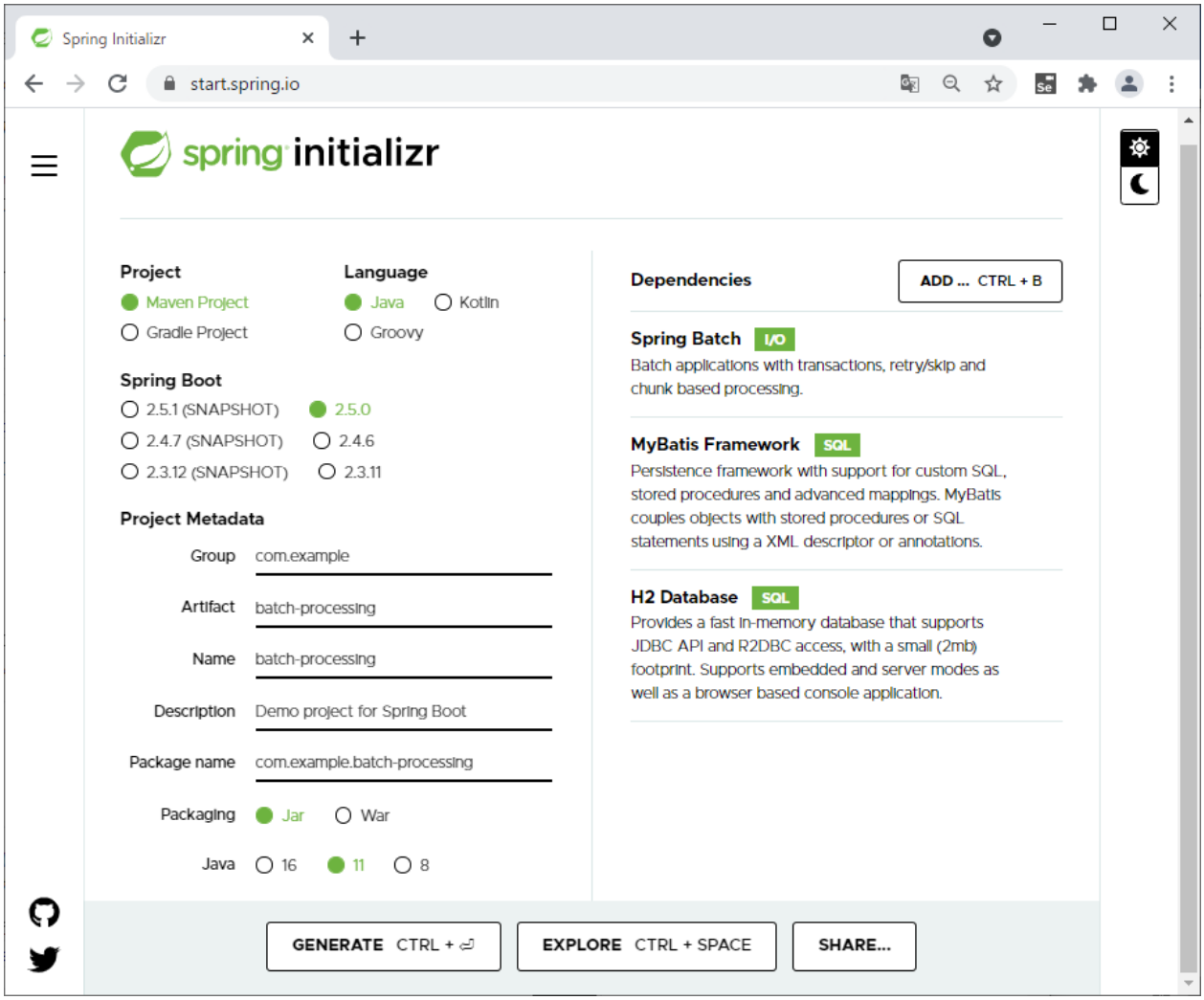
[1] https://spring.pleiades.io/guides/gs/sts/

2. プロジェクトの作成

Spring Batch の開発に着手するにあたり、関連するフレームワークやライブラリ、ドライバを用意する必要があります。方法は Spring initializr よりライブラリの組み合わせを選んで新規に作るか、Github や mvnrepository に登録されているテンプレートを取得します。

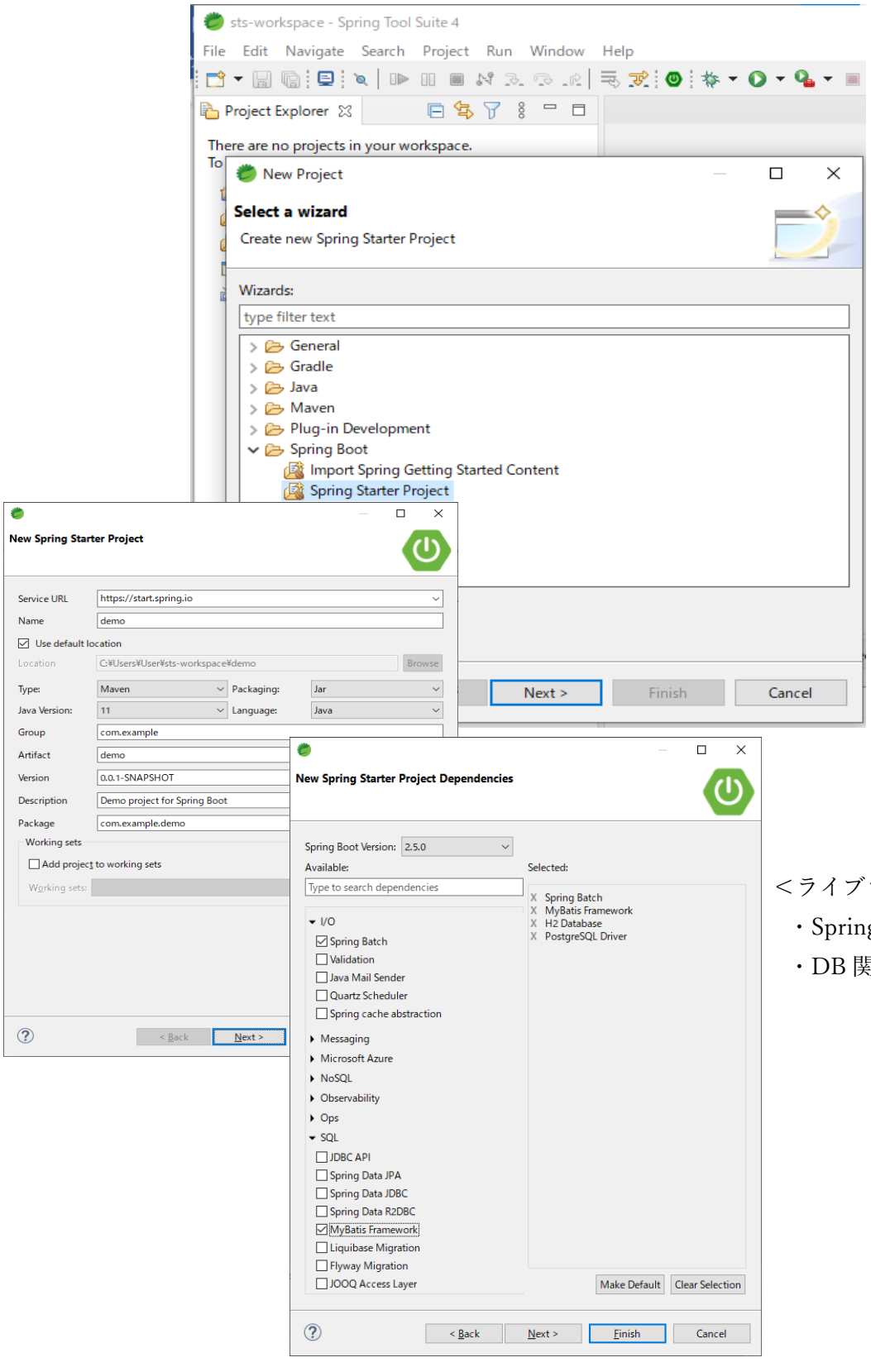
2.1. Spring initializr

https://start.spring.io/ から必要な構成と、依存するフレームワークやライブラリを選択し〔GENERATE〕または〔EXPLORE〕を押下します。ダウンロード後に解凍して STS に既存プロジェクトとしてインポートします（Import... > Existing Maven Projects）。



2.2. Spring Starter Project

STS の Project Explorer を右クリックし、New > Spring Starter Project を実行します。または、Project Explorer が初期状態で、Create a project... が表示されている場合はこのリンクをクリックすると、Spring initializr と同様の選択肢を使いローカルにプロジェクトが作成されます。



＜ライブラリ等の分類＞

・Spring Batch：I/O

・DB 関連：SQL

2.3. Maven によるテンプレートプロジェクトの取得

mvn archetype:generate ^
-DarchetypeGroupId=グループ ID ^
-DarchetypeArtifactId=アーティファクト ID

※プロジェクトが、以下のリモート・リポジトリからダウンロードされます。
（デフォルト）リポジトリ https://mvnrepository.com/
「グループ ID」 >> 「アーティファクト ID」で登録されている前提です。

2.4. 依存ライブラリの追加等

Maven プロジェクトに依存ライブラリを追加したい場合は、プロジェクトに入っている pom.xml の<dependencies> ～ </dependencies>を編集して、mvn dependency:copy-dependencies コマンドを実行します。

＜pom.xml＞

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.5.0</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.example</groupId>
	<artifactId>spring-batch</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>spring-batch</name>
	<description>Demo project for Spring Boot</description>
	<properties>
		<java.version>11</java.version>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-batch</artifactId>
		</dependency>
		<dependency>
			<groupId>org.mybatis.spring.boot</groupId>
			<artifactId>mybatis-spring-boot-starter</artifactId>
			<version>2.2.0</version>
		</dependency>

		<dependency>
			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.postgresql</groupId>
			<artifactId>postgresql</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.batch</groupId>
			<artifactId>spring-batch-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```


3. Spring Batch の開発

「コマンドラインからジョブを開始するための基本的なランチャー」の CommandLineJobRunner クラスを使う場合、Spring Batch で作成したジョブの起動は以下のように行います。

“java org.springframework.batch.core.launch.support.CommandLineJobRunner **jobPath** jobIdentifier”

※コマンドラインオプションは次のとおりです

jobPath: Job を含む xml アプリケーションコンテキスト（以下、**ジョブ定義ファイル**）

-restart: （オプション）最後に失敗した実行を再開する

-stop: （オプション）実行中の実行を停止する

-abandon: （オプション）停止した実行を放棄する

-next: （オプション）Job の JobParametersIncrementer に従ってシーケンスの次を開始する

jobIdentifier: ジョブの名前またはジョブ実行の ID（-stop、-abandon、または -restart の場合）。

※ジョブ定義ファイルの<batch:job id= jobIdentifier >と一致していること

jobParameters: 0 から key=value ペアの形式で指定されたジョブを起動するために使用される多くのパラメーター。

この実行形態の場合、CommandLineJobRunner に Spring Framework の DI により業務クラスを渡し、実行させます。どのクラスを実行させるかはジョブ定義ファイルで指定します。

3.1. ジョブ定義ファイル

以下の例は、一般ファイルの入出力を行う**チャンクモデル**のジョブと Bean の定義を行っています。

自作クラス（ItemProcessor）は component-scan(行 22)で bean 定義を生成（id はクラス名の先頭を小文字に変換）し、ジョブ共通の Bean 定義は外部ファイルから import(行 20)しています。

（1）ジョブ定義ファイル例

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!–Spring Batch 名前空間(Namespace＞context:, batch:, c:, p:)宣言 始め -->
3  <beans xmlns="http://www.springframework.org/schema/beans"
4         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5         xmlns:context="http://www.springframework.org/schema/context"
6         xmlns:batch="http://www.springframework.org/schema/batch"
7         xmlns:c="http://www.springframework.org/schema/c"
8         xmlns:p="http://www.springframework.org/schema/p"
9         xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
10        xsi:schemaLocation="http://www.springframework.org/schema/beans
11              https://www.springframework.org/schema/beans/spring-beans.xsd
12              http://www.springframework.org/schema/context
13              https://www.springframework.org/schema/context/spring-context.xsd
14              http://www.springframework.org/schema/batch
15              https://www.springframework.org/schema/batch/spring-batch.xsd
16              http://mybatis.org/schema/mybatis-spring
17              http://mybatis.org/schema/mybatis-spring.xsd">
18 <!–Spring Batch 名前空間(Namespace:context, batch, c, p)宣言 終わり -->
19     <!-- "import"で Bean定義をresourceで指定したファイルからロードします -->
20     <import resource="classpath:META-INF/spring/batch-context.xml"/>
21     <!-- "context:component-scan"は base-package 中の@Componentから
              bean(このケースではItemProcessor>"myProcessor")を抽出します。-->
22     <context:component-scan base-package="開発パッケージ"/>
23     <!–ItemReader -->
24     <bean id="reader"
25       class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
26       p:resource="file:#{jobParameters['inputFile']}"
       （入力ファイルプロパティ…略 ジョブ起動パラメータに inputfile=ファイル名が必要）
27     </bean>
28     <!–ItemWriter -->
29     <bean id="writer"
30       class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
31       p:resource="file:#{jobParameters['outputFile']}"
       （出力ファイルプロパティ…略 ジョブ起動パラメータに outputfile=ファイル名が必要）
32     </bean>
33     <!-- ジョブ構成 -->
34     <batch:job id="jobIdentifier" job-repository="jobRepository">
35         <batch:step id=" jobIdentifier.step01">
36             <batch:tasklet transaction-manager="jobTransactionManager">
37                 <batch:chunk reader="reader"
38                     processor="myProcessor" …開発パッケージ.MyProcessorを実行
39                     writer="writer" commit-interval="10"/>
40             </batch:tasklet>
41         </batch:step>
42     </batch:job>
43 </beans>
```


（2）ジョブ共通 Bean 定義

ジョブ定義ファイルに<import resource="classpath:META-INF/spring/batch-context.xml"/>の指定で取り込まれるファイルです。（*記述量の関係からbean idに関連する部分だけ切り出し*）

・Spring、RDB 等組込みクラスはここで個別に bean 定義しています。

・${ }で囲んだ文字列は、4 行目<context:property-placeholder/>で取り込んだ.properties ファイルの値で置き換わります。

```
1<?xml version="1.0" encoding="UTF-8"?>
2<beans xmlns="http://www.springframework.org/schema/beans"
   名前空間(Namespace＞context:, batch:, c:, p:, util:, jdbc:)宣言
 3>
 4    <context:property-placeholder location="classpath:batch-application.properties"
 5           system-properties-mode="OVERRIDE"
 6           ignore-resource-not-found="false"
 7           ignore-unresolvable="true"
 8           order="1"
 9    />
10    <!-- framework definitions -->
11    <bean id="adminDataSource" p:driverClassName="${admin.jdbc.driver}" p:url="${admin.jdbc.url}"
12    <bean id="adminTransactionManager"
13    <batch:job-repository id="jobRepository"
14    <bean id="jobRegistry"
15    <bean id="jobExplorer"
16    <bean id="jobParametersConverter"
17    <bean id="jobLauncher"
18    <bean id="jobOperator"
19    <bean class="org.springframework.batch.core.configuration.support.JobRegistryBeanPostProcessor"
20    <bean id="exitCodeMapper"
21    <jdbc:initialize-database data-source="adminDataSource"
22       enabled="${data-source.initialize.enabled:false}" ignore-failures="ALL">
23       <jdbc:script location="${spring-batch.schema.script}" />
24    </jdbc:initialize-database>
25    <!-- ### ADD ### database initialize definition -->
26    <jdbc:initialize-database data-source="jobDataSource"
27    <!-- Job-common definitions -->
28    <bean id="jobDataSource" p:driverClassName="${jdbc.driver}" p:url="${jdbc.url}"
29    <bean id="jobTransactionManager" p:dataSource-ref="jobDataSource"
30    <bean id="messageSource" p:basenames="i18n/application-messages"
31    <bean id="validator" p:validator-ref="beanValidator"
32    <bean id="beanValidator"
33    <bean id="jobSqlSessionFactory" p:dataSource-ref="jobDataSource"
34    <bean id="batchModeSqlSessionTemplate"
35         c:sqlSessionFactory-ref="jobSqlSessionFactory"
36         c:executorType="BATCH" />
37    <bean id="jobResourcelessTransactionManager"
38</beans>
```

（3）bean定義で参照している.propertiesファイルの内容

以下が<context:property-placeholder/>で参照しているファイルの一部です。

```
## Application settings.
# Admin DataSource settings.
admin.jdbc.driver=org.h2.Driver
admin.jdbc.url=jdbc:h2:~/batch-admin;AUTO_SERVER=TRUE
```


3.2. Framework の bean 定義／記法

以下は、Spring Batch と RDB データソース関連の bean 定義の全体です。

```
<!-- framework definitions -->
<bean id="adminDataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"
      p:driverClassName="${admin.jdbc.driver}"
      p:url="${admin.jdbc.url}"
      p:username="${admin.jdbc.username}"
      p:password="${admin.jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false"/>
<bean id="adminTransactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
      p:dataSource-ref="adminDataSource"
      p:rollbackOnCommitFailure="true"/>
<batch:job-repository id="jobRepository"
       data-source="adminDataSource"
       transaction-manager="adminTransactionManager"
       isolation-level-for-create="READ_COMMITTED"/>
<bean id="jobRegistry" class="org.springframework.batch.core.configuration.support.MapJobRegistry"/>
<bean id="jobExplorer" class="org.springframework.batch.core.explore.support.JobExplorerFactoryBean"
      p:dataSource-ref="adminDataSource"/>
<bean id="jobParametersConverter" class="org.springframework.batch.core.jsr.JsrJobParametersConverter"
      c:dataSource-ref="adminDataSource" />
<bean id="jobLauncher" class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
      p:jobRepository-ref="jobRepository" />
<bean id="jobOperator" class="org.springframework.batch.core.launch.support.SimpleJobOperator"
      p:jobRepository-ref="jobRepository"
      p:jobRegistry-ref="jobRegistry"
      p:jobExplorer-ref="jobExplorer"
      p:jobParametersConverter-ref="jobParametersConverter"
      p:jobLauncher-ref="jobLauncher" />
<bean class="org.springframework.batch.core.configuration.support.JobRegistryBeanPostProcessor"
      p:jobRegistry-ref="jobRegistry"/>
```

・P-名前空間(namespace)[2]

```
<bean id="adminTransactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
       p:dataSource-ref="adminDataSource"
       p:rollbackOnCommitFailure="true"/>
```

p:xxxx はプロパティ名、xxxx-ref は”ref=”と等価で、bean の定義内容は以下記述と同一です。

```
<bean id="adminTransactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
      <property name="dataSource" ref="adminDataSource"/>
      <property name="rollbackOnCommitFailure” value="true"/>
</bean>
```

※P-名前空間を使うとプロパティの名前や参照先を bean 要素の属性として書くことができます。

・id の記述がない bean

クラス名の先頭一文字を小文字にしたものが bean の id として作られます。

[2] 詳細は https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-p-namespace

3.3. RDB を使用する場合

RDB を使用する場合は以下の設定が必要です（MyBatis3/mybatis-spring-チャンクモデルで例示）。

3.3.1. ジョブ共通 bean 定義

Spring、MyBatis、DB 接続の Bean は複数ジョブ共通になります。ジョブ定義ファイルに import されているファイルに以下の bean 定義を追加します。

（1）データソース

jobRepositoryのデータソース(adminDataSource)とは別の業務処理用のデータソースです。

```
<bean id="jobDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${jdbc.driver}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false" />
<!-- トランザクション管理 -->
<bean id="jobTransactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
      p:dataSource-ref="jobDataSource"
      p:rollbackOnCommitFailure="true" />
```

（2）SqlSessionFactory/SqlSessionTemplate (MyBatis-Spring)

MyBatisのセッション管理をSpringのセッション管理に置き換えます。Springのトランザクション設定に従うSqlSessionが注入されます。

MyBatisの環境設定にmybatis-config.xmlを使わずconfigurationプロパティで行っています。

```
<bean id="jobSqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="jobDataSource" >
    <property name="configuration">
        <bean class="org.apache.ibatis.session.Configuration"
            p:localCacheScope="STATEMENT"
            p:lazyLoadingEnabled="true"
            p:aggressiveLazyLoading="false"
            p:defaultFetchSize="1000"
            p:defaultExecutorType="REUSE" />
    </property>
</bean>
<!-- SqlSessionTemplateは、MyBatisのSqlSessionをSpring管理のトランザクション内で実行するため
      のSqlSession実装。MyBatisのデフォルト実装である DefaultSqlSessionは使用しません。-->
<bean id="batchModeSqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate"
      c:sqlSessionFactory-ref="jobSqlSessionFactory"
      c:executorType="BATCH" />
```


3.3.2. プロパティファイルの項目追加

プロパティファイル(batch-application.properties)にデータソースで参照しているDB接続用の情報（以下、H2 embeddedを想定した例）を追加します。

```
jdbc.driver=org.h2.Driver
jdbc.url=jdbc:h2:~/batch-admin;AUTO_SERVER=TRUE
jdbc.username=sa
jdbc.password=
```

※例では、admin.jdbc.urlとjdbc.urlが両方“~/batch-admin”になっているため、ユーザのホームディレクトリに“batch-admin…”でDBが作られJobRepositoryとジョブのデータが同居します。

H2 embeddedの場合、インストール先の ¥H2¥bin¥h2.bat を実行するとサービスが立ち上がりブラウザからRDBの内容を確認／更新することができます。

3.3.3. ジョブ定義ファイル

ItemReader／ItemWriter、MyBatis で使用する Mapper 等はジョブ毎に異なることが多いため、ジョブ起動時に指定するジョブ定義ファイルに記載します。

Mapper はエンティティ毎に Java interface と Mapper xml のペアを作成し、格納先のパス構成を同一（xml の方は resources/...：以下、*Mapper 開発パッケージ*と記載）にしておきます。

（１）Mapper の自動検出

context:component-scanがBeanを検索するのと良く似た方法でMapperを検出します。

```
<mybatis:scan base-package="Mapper開発パッケージ"
      factory-ref="jobSqlSessionFactory"/>
```

（２）ItemReader

setQueryIdプロパティとして指定されたクエリを実行し、selectCursor()メソッドを使用して要求されたデータを取得します。read()メソッドが呼び出されるたびに、要素がなくなるまでカーソルの次の要素が返されます。

```
<bean id="reader"
  class="org.mybatis.spring.batch.MyBatisCursorItemReader"
  p:queryId="Mapper開発パッケージ.Mapper.cursor"
  p:sqlSessionFactory-ref="jobSqlSessionFactory"/>
```

※MyBatis-Springにはもう一つ、MyBatisPagingItemReadeというページング方式のItemReaderがあります。

（３）ItemWriter

SqlSessionTemplateのバッチ機能を使用します。SqlSessionFactoryは、BATCHエグゼキュータで構成する必要があります。write()が呼び出されるとプロパティstatementIdにマップされたステートメントを実行します。write()はトランザクション内で呼び出されることが期待されます。

```
<bean id="writer" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"
  p:statementId="Mapper開発パッケージ.Mapper.ステートメントid"
  p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>
```


3.3.4. DTO

エンティティの1レコード分のデータを保持しているJavaBeanを用意します。無駄な走査が入らないように、Mapperの自動検出(mybatis:scan)対象のパッケージである*Mapper開発パッケージ*とは分けておきます。

3.3.5. Mapper インタフェース

MyBatisでDBアクセスを行うためのJava interface

```
package Mapper開発パッケージ;
import DTO;
import org.apache.ibatis.cursor.Cursor;
public interface Mapper{
  Cursor<Dtoクラス> cursor();
  int ステートメントid (Dtoクラス stmtPara);
```

3.3.6. Mapper xml

Mapper インタフェースと対になる xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="Mapper開発パッケージ.Mapper">
    <select id="cursor" resultType="DTO ">
        SELECT
            id,
            type,
            status,
            point
        FROM
            member_info
        ORDER BY
            id ASC
    </select>
    <update id="ステートメントid " parameterType="DTO ">
        UPDATE
            member_info
        SET
            status = #{status},
            point = #{point}
        WHERE
            id = #{id}
    </update>
</mapper>
```

#{}の部分は MyBatis が PreparedStatement のパラメーターを DTO から作成します。


```
<update id="ステートメントid" parameterType="DTO">
    UPDATE
        member_info
    SET
        status = #{status},
        point = #{point}
    WHERE
        id = #{id}
</update>
```

は、以下のように変換されます。

```
PreparedStatement ps = conn.prepareStatement(
    "UPDATE member_info SET status = ? point = ? WHERE id = ?"
)
ps.setObject(1, DTO.getStatus);
ps.setObject(2, DTO.getPoint);
ps.setObject(3, DTO.get Id);
```

※DTO のメソッド get*Xxxx*, set*Xxxx* の *Xxxx* はテーブルのカラム ID と一致している場合は、自動的に設定されます。ID が一致していない場合は as 句で一致させるか、resultMap による割付が必要です。（以下、resultMap による result⇔property 割付の例）

```
<resultMap id="mapDenpyo" type="jp.co.focs.dto.DenpyoDto">
  <!-- 配列はTypeHandlerの設定が必要 -->
  <result column="karikamokucd" property="karikamokucd" typeHandler="org.apache.ibatis.type.ArrayTypeHandler"
    jdbcType="CHAR"  javaType="java.lang.String" />
  <result column="karikingaku"   property="karikingaku" typeHandler="org.apache.ibatis.type.ArrayTypeHandler"
    jdbcType="VARCHAR"  javaType="java.lang.String" />
  <result column="kasikamokucd" property="kasikamokucd" typeHandler="org.apache.ibatis.type.ArrayTypeHandler"
    jdbcType="CHAR"  javaType="java.lang.String" />
  <result column="kasikingaku"   property="kasikingaku" typeHandler="org.apache.ibatis.type.ArrayTypeHandler"
    jdbcType="VARCHAR"  javaType="java.lang.String" />
</resultMap>

<!-- select id="cursor" resultType="jp.co.focs.dto.DenpyoDto" -->
<select id="cursor" resultMap="mapDenpyo">
   SELECT
        err
      , trdate
      , denpyono
      , tantou
      , karikamokucd
      , karikingaku
      , kasikamokucd
      , kasikingaku
      , tekiyo
      , sysdate
      , errmessage
   FROM
       denpyo
   ORDER BY
       denpyono ASC
</select>
```

※Mapper はテーブル単位で作る必要はなく、同一トランザクションとして扱う複数テーブル単位で問題ありません。ItemReader、ItemWriter で共用可能です。**プロジェクトの方針を確認してください。**

3.3.7. チャンクモデルでの複数テーブル更新とトランザクション

チャンクモデル（ItemReader/ItemProcessor(任意)/ItemWriter）はコミットのタイミングをジョブ定義ファイルで設定できるため大量データを扱うシステムでは便利ですが、MyBatis 3が提供するItemWriterのMyBatisBatchItemWriterには以下の制約があります。

① テーブルの更新が１SQLコマンドに限定される

② SQLの実行を「バッチ更新」[3]（executorType="BATCH"）に統一しなければならない

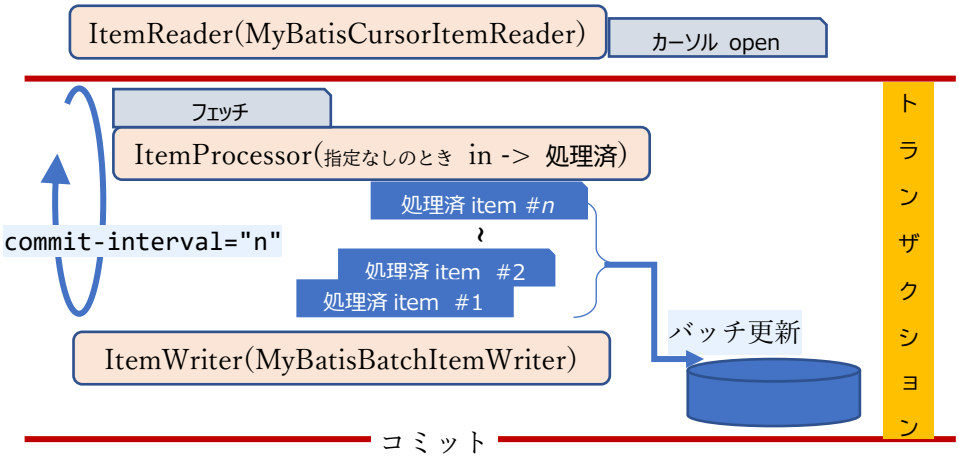
これ等に反すると、以下のメッセージと共にエラーになります。

⇒`"Batch execution returned invalid results. Expected 1 but number of BatchResult objects returned was n"`

（１）チャンクモデルのトランザクション

チャンクモデルの実行サイクルとトランザクションは、入力(ItemReader)や出力(ItemWriter)にMyBatis 3を使った場合以下のように行われます。

※入力が一般ファイルの場合でも、トランザクションのサイクルは同様になります



※ItemReaderは<ItemProcessor～ItemWriter>のトランザクション外で動作するため、入力にテーブルを使う場合トランザクション中のデータ一貫性は保障されません。**別途 排他の制御／運用が必要**です。

※図にはステップの中で使われるトランザクションを表していますが、その他にJob管理DB更新用のトランザクションが作られます。

[3] JDBC のバッチ更新機能⇒addBatch()メソッドで更新する SQL を溜め込み、executeBatch()メソッドで溜め込んだＳＱＬをＤＢへ送信して複数の SQL を一度に実行します。

（２）MyBatisBatchItemWriterを使わないチャンクモデル

MyBatisBatchItemWriterは単一SQLによるバッチ更新に特化しており、入力：出力が１：１の大量データ更新のみに適用できる[4]形態です。また、タスクレットモデルを使えばMyBatisBatchItemWriterの制約は関係なくなりますが、データ量によっては中間コミットを自前で行う必要があるかもしれません。

Spring Batchのトランザクション制御（中間コミット）を使って複数テーブルを更新する例（MyBatisBatchItemWriterを使わないチャンクモデル）を以下にあげます。

【課題例】

貸方／借方 各々複数行の明細を持つ伝票イメージのテーブルを入力とし、共通項目と明細を切り出して、1件の共通レコードとｎ件の明細レコードを作成します。

＜入力テーブル＞

```
CREATE TABLE denpyo(
       err bool
     , trdate CHAR(8) NOT NULL
     , denpyono CHAR(5)
     , tantou CHAR(4)
     , karikamokucd CHAR(6)[]
     , karikingaku VARCHAR(13)[]
     , kasikamokucd CHAR(6)[]
     , kasikingaku VARCHAR(13)[]
     , tekiyo VARCHAR(50)
     , sysdate CHAR(8) NOT NULL
     , errmessage VARCHAR(100)
 , CONSTRAINT pk_denpyo PRIMARY KEY (denpyono, tantou)
);
```

＜出力テーブル①…共通＞

```
CREATE TABLE t_siwake_kyotu(
       trdate CHAR(8) NOT NULL
     , denpyono CHAR(5)
     , tantou CHAR(4)
     , tekiyo VARCHAR(50)
     , sysdate CHAR(8) NOT NULL
 , CONSTRAINT pk_t_siwake_kyotu PRIMARY KEY (denpyono)
);
```

＜出力テーブル②…明細＞

```
CREATE TABLE t_siwake_meisai(
       denpyono CHAR(5) NOT NULL
     , taisyakukb CHAR(1) NOT NULL
     , gyo NUMERIC(1) NOT NULL
     , kamokucd CHAR(6) NOT NULL
     , kingaku NUMERIC(13,0) NOT NULL
     , aitekamokucd CHAR(6) NOT NULL
 , CONSTRAINT pk_t_siwake_meisai PRIMARY KEY (denpyono, gyo, taisyakukb)
 , CONSTRAINT fk_denpyono FOREIGN KEY (denpyono) REFERENCES t_siwake_kyotu(denpyono)
);
```

[4] データ加工が少なければSQLを実行ツールで走行した方が効率的です

バッチ・フレームワーク（SpringBatch/Maven，MyBatis）

① ジョブ定義ファイル

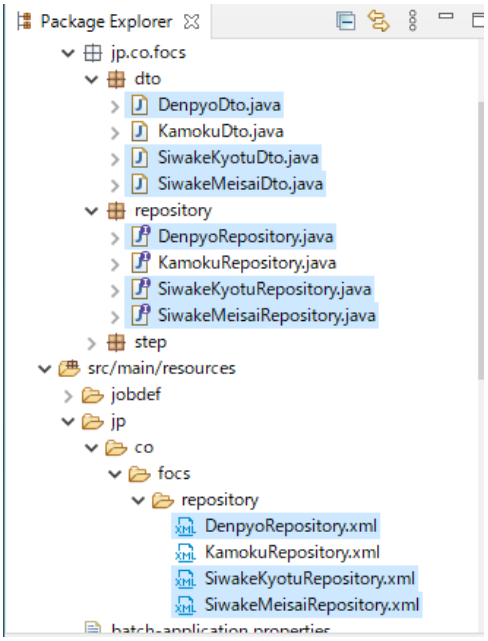
ItemWriterは省略できないので（ItemProcessorの戻り値を全てnullにすることでItemWriterの呼び出しを回避することはできます)、下記の例ではWindowsのデスクトップに“jobtest-out.txt”というファイル名でItemProcessorの処理済item(DTO)の内容を出力しています。

ItemReader、ItemWriterのbean定義(id=”filewriter”)とbatch定義です。

```
<bean id="reader"
      class="org.mybatis.spring.batch.MyBatisCursorItemReader"
      p:queryId="jp.co.focs.repository.DenpyoRepository.cursor"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>
<bean id="filewriter" class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
       p:resource="file:${HOMEPATH}/Desktop/jobtest-out.txt"
       p:encoding="MS932"
       p:lineSeparator="&#x0A;"
       p:appendAllowed="false"
       p:shouldDeleteIfExists="true"
       p:shouldDeleteIfEmpty="false"
       p:transactional="true">
   <property name="lineAggregator">
     <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator"
     <bean class="org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregator"
           p:delimiter=",">
       <property name="fieldExtractor">
         <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
           p:names=出力対象のDTOの項目名…例⇒"err,trdate,denpyono,tantou, …以下略"/>
       </property>
     </bean>
   </property>
 </bean>

<batch:job id="job001" job-repository="jobRepository">
    <batch:step id="job001.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                         processor="denpyoValidateItemProcessor"
                         writer="filewri
                         commit-interval="100"/>
        </batch:tasklet>
    </batch:step>
 </batch:job>
```

② DTO、Mapperインタフェース／Mapper XML



使用する各テーブルの下記資材を作ります。

・Dto（テーブル全カラムのgetter/setter）

・Mapperインタフェース／xml のペア

＜denpyo テーブル＞

〔Mapperインタフェース〕

```
package jp.co.focs.repository;

import org.apache.ibatis.cursor.Cursor;
import jp.co.focs.dto.DenpyoDto;

public interface DenpyoRepository {
  Cursor<DenpyoDto> cursor();

  int updateDenpyo(DenpyoDto denpyo);
}
```

〔Mapper xml〕

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="jp.co.focs.repository.DenpyoRepository">
    <resultMap id="mapDenpyo" type="jp.co.focs.dto.DenpyoDto">
      <!-- 配列はTypeHandlerの設定が必要 -->
      <result column="karikamokucd" property="karikamokucd" typeHandler="org.apache.ibatis.type.ArrayTypeHandler"
        jdbcType="CHAR"  javaType="java.lang.String" />
      <result column="karikingaku"   property="karikingaku" typeHandler="org.apache.ibatis.type.ArrayTypeHandler"
        jdbcType="VARCHAR"  javaType="java.lang.String" />
      <result column="kasikamokucd" property="kasikamokucd" typeHandler="org.apache.ibatis.type.ArrayTypeHandler"
        jdbcType="CHAR"  javaType="java.lang.String" />
      <result column="kasikingaku"   property="kasikingaku" typeHandler="org.apache.ibatis.type.ArrayTypeHandler"
        jdbcType="VARCHAR"  javaType="java.lang.String" />
    </resultMap>
    <select id="cursor" resultMap="mapDenpyo">
        SELECT
            err
          , trdate
          , denpyono
          , tantou
          , karikamokucd
          , karikingaku
          , kasikamokucd
          , kasikingaku
          , tekiyo
          , sysdate
          , errmessage
        FROM
            denpyo
        ORDER BY
            denpyono ASC
    </select>

    <update id="updateDenpyo" parameterType="jp.co.focs.dto.DenpyoDto">
        UPDATE
            denpyo
        SET
            err        = #{err}
          , errmessage = #{errmessage}
        WHERE
            denpyono   = #{denpyono}
    </update>
</mapper>
```


＜t_siwake_kyotu テーブル＞

〔Mapperインタフェース〕

```
package jp.co.focs.repository;
import jp.co.focs.dto.SiwakeKyotuDto;
public interface SiwakeKyotuRepository {
  int insKyotu(SiwakeKyotuDto siwake);
}
```

〔Mapper xml〕

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="jp.co.focs.repository.SiwakeKyotuRepository">
    <insert id="insKyotu" parameterType="jp.co.focs.dto.SiwakeKyotuDto">
        INSERT INTO t_siwake_kyotu(
           trdate
         , denpyono
         , tantou
         , tekiyo
         , sysdate
         )
        VALUES(
           #{trdate}
         , #{denpyono}
         , #{tantou}
         , #{tekiyo}
         , #{sysdate}
         )
    </insert>
</mapper>
```

＜t_meisai_kyotu テーブル＞

〔Mapperインタフェース〕

```
package jp.co.focs.repository;
import jp.co.focs.dto.SiwakeMeisaiDto;
public interface SiwakeMeisaiRepository {
  int insMeisai(SiwakeMeisaiDto siwake);
}
```

〔Mapper xml〕

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="jp.co.focs.repository.SiwakeMeisaiRepository">
    <insert id="insMeisai" parameterType="jp.co.focs.dto.SiwakeMeisaiDto">
        INSERT INTO t_siwake_meisai(
           denpyono
         , gyo
         , kamokucd
         , taisyakukb
         , kingaku
         , aitekamokucd
         )
        VALUES(
           #{denpyono}
         , #{gyo}
         , #{kamokucd}
         , #{taisyakukb}
         , #{kingaku}
         , #{aitekamokucd}
         )
    </insert>
</mapper>
```


③ ItemProcessor

```
package jp.co.focs.step;

（略）

/**
 * 伝票の内容をチェックし、問題が無ければ仕訳共通テーブルと仕訳明細テーブルに登録します。
 * @author User
 *
 */
@Component
public class DenpyoValidateItemProcessor implements ItemProcessor<DenpyoDto, DenpyoDto> {
    Logger logger = LoggerFactory.getLogger(this.getClass());

    @Inject
    KamokuRepository kamokuMst;
    @Inject
    SiwakeKyotuRepository siwakeKyotu;
    @Inject
    SiwakeMeisaiRepository siwakeMeisai;

    @Override
    public DenpyoDto process(DenpyoDto item) throws Exception {
        logger.debug("item denpyono:" + item.getDenpyono());
        //項目チェック
        if (validateDenpyo(item) > 0) {
            return item;
        };

        //共通レコード作成
        SiwakeKyotuDto kyotuDto = new SiwakeKyotuDto();
        kyotuDto.setDenpyono(item.getDenpyono());
        kyotuDto.setTantou(item.getTantou());
        kyotuDto.setTekiyo(item.getTekiyo());
        kyotuDto.setTrdate(item.getTrdate());
        kyotuDto.setSysdate(sysDate);
        siwakeKyotu.insKyotu(kyotuDto);
        //借方明細レコード作成
        String aiteKamoku = item.getKasikamokucd().length > 1 ? /*諸口*/ "000000":item.getKasikamokucd()[0];
        insMeisai(item.getDenpyono()
                , "1"
                , item.getKarikamokucd()
                , item.getKarikingaku()
                , aiteKamoku
                );
        //貸方明細レコード作成
        aiteKamoku = item.getKarikamokucd().length > 1 ? /*諸口*/ "000000":item.getKarikamokucd()[0];
        insMeisai(item.getDenpyono()
                , "2"
                , item.getKasikamokucd()
                , item.getKasikingaku()
                , aiteKamoku
                );
        item.setErrmessage("仕訳テーブルに登録しました。");
        return item;
    }
    private int validateDenpyo(DenpyoDto item) { 略 }
    private void insMeisai(String denpyono, String taisyakukb, String[] kamokucdArr,
                           String[] kingakuArr, String aiteKamokucd) { 略 }

}
```


3.3.8. DB 初期化スクリプト

```
① <jdbc:initialize-database data-source="adminDataSource" enabled="${data-
  source.initialize.enabled:false}" ignore-failures="ALL">
②     <jdbc:script location="${spring-batch.schema.script}" />
③     <jdbc:script location="${terasoluna-batch.commit.script}" />
④ </jdbc:initialize-database>
⑤
⑥ <!-- ### ADD ### database initialize definition ${変数:デフォルト値} -->
⑦ <jdbc:initialize-database data-source="jobDataSource" enabled="${data-
  source.initialize.enabled:false}" ignore-failures="ALL">
⑧     <jdbc:script location="${tutorial.create-table.script}" />
⑨     <jdbc:script location="${tutorial.insert-data.script}" />
⑩ </jdbc:initialize-database>
```

<jdbc:initialize-database ~>をジョブ定義ファイルやジョブ共通 bean 定義ファイルに書いておくとジョブ起動時にスクリプト（<jdbc:script location=*スクリプトファイル*>）が実行されて RDB の初期設定等できます。

enabled="${*変数*:false}"と書くと、プロパティファイルに *変数*=true の記述がなければ実行されません。

3.4. クラスとリソースの追加／フォルダパス

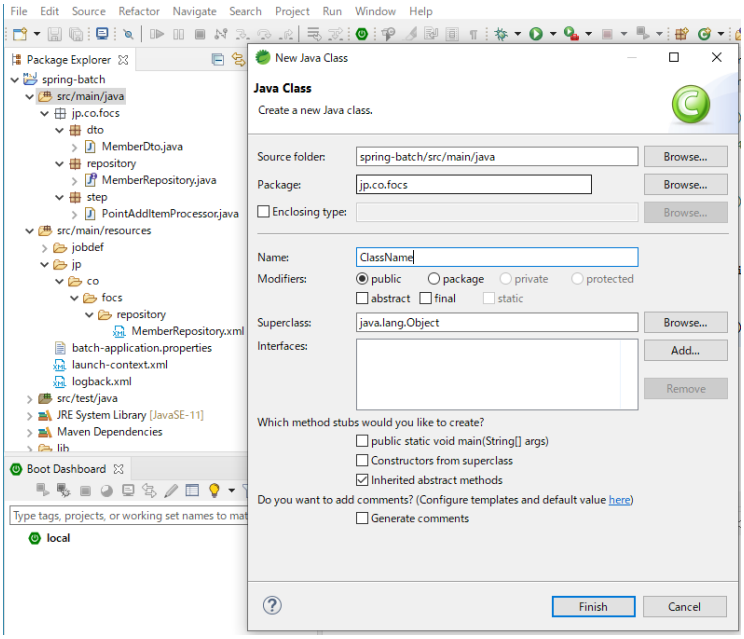
資材の格納先は、ジョブ定義ファイルから参照されます。

bean 定義の抽出…　　`<context:component-scan base-package="開発パッケージ"/>`

MyBatisのMapper抽出…　`<mybatis:scan base-package="Mapper開発パッケージ"`

（1）Java の開発パッケージ(package)

開発パッケージ は、クラス作成時（STS上でプロジェクトを右クリック ＞New＞Class を選択し）[Package:]欄に指定したものか、その上位階層*1がbase-packageになります。

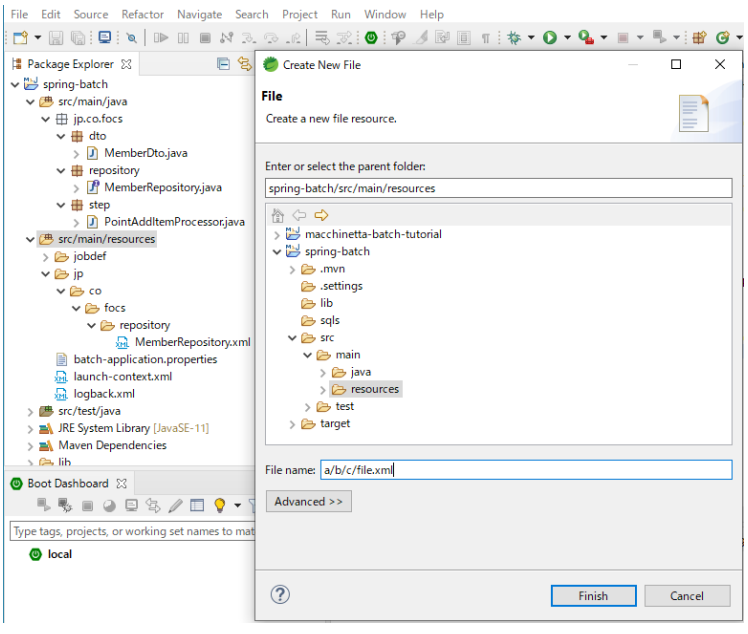


*1: package は格納先のフォルダ階層に一致しており、この階層を使って component-scan の対象を広げたり絞ったりができます。

（2）リソースフォルダ

MyBatisのMapper xmlは resources 配下に格納しますが、この時のパスはMapper interfaceのパッケージと一致させます。src/main/resourcesを右クリックし、

New＞File　でJava interfaceのpackageの場所を選択します。フォルダ階層が無い場合は、[File name:]に a/b/c/Mapper.xml のようにフォルダを指定して中間パスも作成できます。



4. デバッグ

Spring Batch のジョブを実行して、試験対象のパスを通過している確証が取れなかったり、不具合の解析が難しい場合があります。以下に開発工程でのデバッグ方法を紹介します。

4.1. STS によるデバッグ実行

デバッグモードで実行し、ブレークポイントで止めることができます。手順は、以下の通りです。

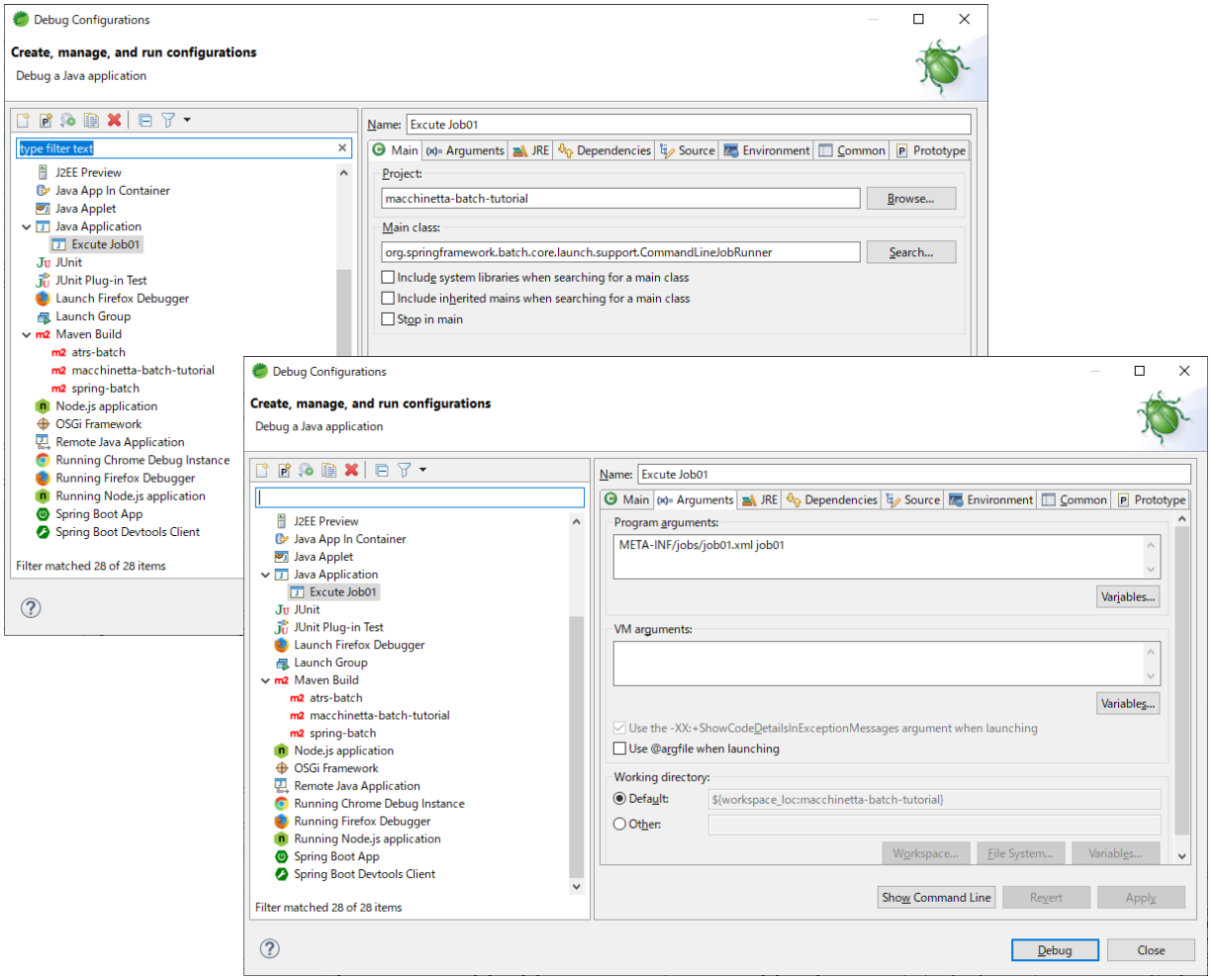
（1）デバッグ構成 の作成

プロジェクトを右クリックし、Debug As ＞Debug Configurations... と進み
以下の情報を設定します。

① Main タグ-Main class:
org.springframework.batch.core.launch.support.CommandLineJobRunner

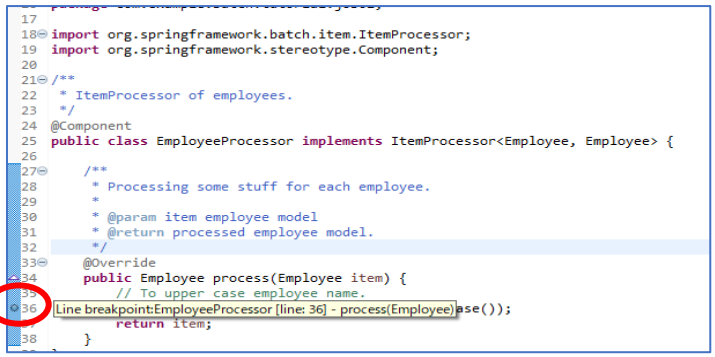
② (x)= Arguments タグ -Program arguments:
ジョブ定義ファイルのパス、ファイル名

※この情報は記録され再利用可能です。『Run Configuration（実行構成）の作成』で作られた構成情報があればそのまま使うことができます。



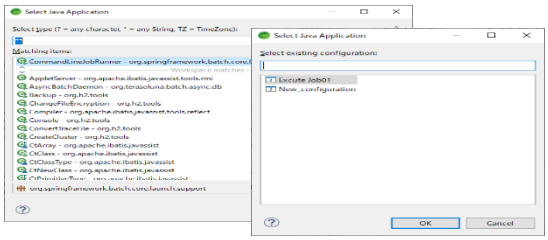
（2）ブレークポイントの設定

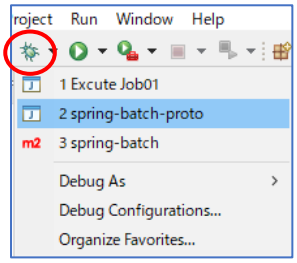
行番号左枠のところをダブルクリックすると青い中点（ブレークポイント）が付きます。処理がこの行まで進んだ時に中断して操作待ちになります。



（3）デバッグの実行

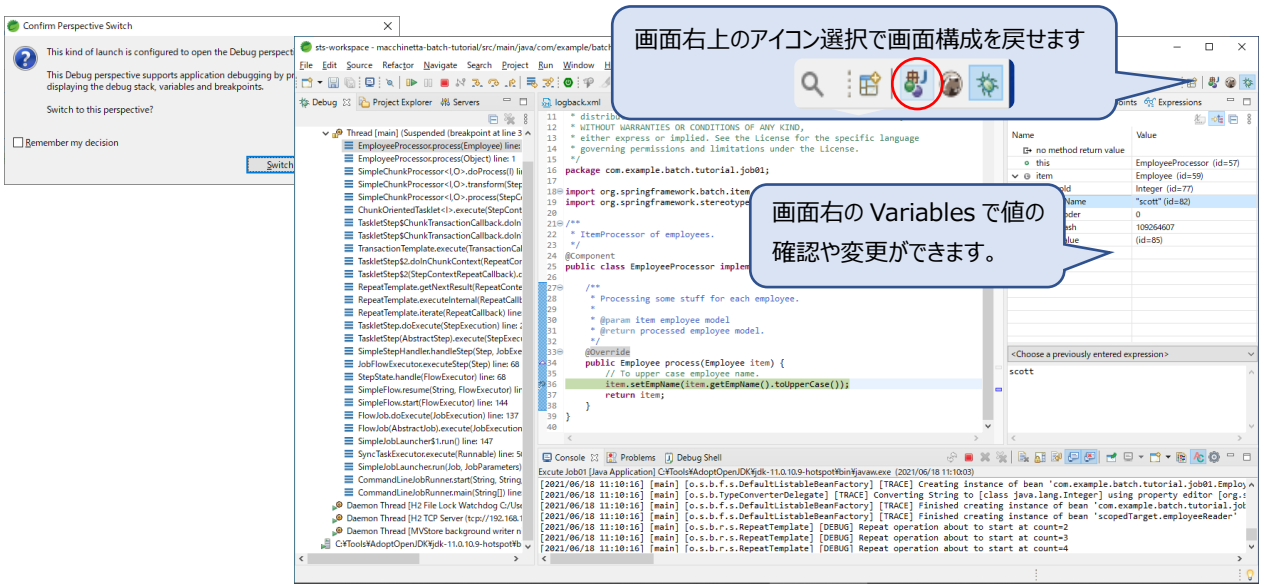
① プロジェクトを右クリック、Debug As ＞1 Java Application ＞
org.springframework.batch.core.launch.support.CommandLineJobRunner を選択します。
構成情報が複数ある場合は、ツールバーのデバッグをプルダウンして選択することができます。





複数の構成情報から選択

② 処理が始まり、ブレークポイントまで処理が進むとデバッグ画面(Debug perspective)への切替え確認がでるので、[Switch]を押下すると画面がデバッグ用構成に切り替わります。



主な操作キー　F8：処理続行（次のブレークポイント迄進みます）
F6：ステップ実行（次のステップに進む）
F5：メソッド呼び出しに入る、F7：メソッドから抜ける

4.2. ログ出力

想定外の結果となり原因が分からない場合は最初に問題の切り分けが必要です。問題を絞り込むためにはログが有用です。

（1）ログ出力レベル

Spring Bootはlogbackをログ出力のデフォルトで使っています。プロジェクトのクラスパス上にあるlogback.xmlの出力レベルを下げることで色々な情報が出力されます。

logback.xml

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

    <appender name="consoleLog" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>[%d{yyyy/MM/dd HH:mm:ss}] [%thread] [%-22logger{22}] [%-5level] %msg%n</pattern>
        </encoder>
    </appender>

<!--
-->

    <root level="TRACE">
        <appender-ref ref="consoleLog" />
    </root>
</configuration>
```

※level=”TRACE”にすると、SQL の実行時に関連の情報を見ることができます。

Problems Javadoc Declaration Search Console Debug

```
<terminated> Excute Job01 [Java Application] C:¥Tools¥AdoptOpenJDK¥jdk-11.0.10.9-hotspot¥bin¥javaw.exe (2021/06/18 11:50:34 – 11:53:40)
.TransactionInterceptor] [TRACE] Getting transaction for [org.springframework.batch.core.repository.support.SimpleJobRepository.update]
.JdbcTemplate  ] [DEBUG] Executing prepared SQL update
.JdbcTemplate  ] [DEBUG] Executing prepared SQL statement [UPDATE BATCH_STEP_EXECUTION set START_TIME = ?, END_TIME = ?, STATUS = ?, COMMIT_COUNT = ?, READ_COUNT = ?, FILTER_COUNT = ?,
.TransactionSynchronizationManager] [TRACE] Retrieved value [org.springframework.jdbc.datasource.ConnectionHolder@6b1b19cf] for key [org.apache.commons.dbcp2.BasicDataSource@1f2f0109]
.TransactionSynchronizationManager] [TRACE] Retrieved value [org.springframework.jdbc.datasource.ConnectionHolder@6b1b19cf] for key [org.apache.commons.dbcp2.BasicDataSource@1f2f0109]
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 1, parameter value [Fri Jun 18 11:50:48 JST 2021], value class [java.util.Date], SQL type 93
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 2, parameter value [Fri Jun 18 11:53:39 JST 2021], value class [java.util.Date], SQL type 93
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 3, parameter value [COMPLETED], value class [java.lang.String], SQL type 12
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 4, parameter value [1], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 5, parameter value [5], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 6, parameter value [0], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 7, parameter value [5], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 8, parameter value [COMPLETED], value class [java.lang.String], SQL type 12
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 9, parameter value [], value class [java.lang.String], SQL type 12
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 10, parameter value [3], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 11, parameter value [0], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 12, parameter value [0], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 13, parameter value [0], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 14, parameter value [0], value class [java.lang.Integer], SQL type 4
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 15, parameter value [Fri Jun 18 11:53:39 JST 2021], value class [java.util.Date], SQL type 93
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 16, parameter value [10], value class [java.lang.Long], SQL type -5
.StatementCreatorUtils] [TRACE] Setting SQL statement parameter value: column index 17, parameter value [2], value class [java.lang.Integer], SQL type 4
.JdbcTemplate  ] [TRACE] SQL update affected 1 rows
```

※ファイルにログを出力する場合は、<appender> ~ </appender>の中を以下のようにします。

```
<appender name="fileLog" class="ch.qos.logback.core.FileAppender">
  <!-- Windows の場合 c:/temp/test.log のよう書くか、以下のようにパス区切りをエスケープ「¥¥」します -->
  <file>c:¥¥temp¥¥test.log</file>
  <append>true</append>
  <!-- encoders are assigned the type
       ch.qos.logback.classic.encoder.PatternLayoutEncoder by default -->
  <encoder>
    <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n</pattern>
  </encoder>
</appender>
  (中略)
  <appender-ref ref="fileLog" />
```


（2）デバッグ／内容確認用ログの作成

ログを出力したい場合は以下のようにします。

① 以下ライブラリの追加

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

② ログのテキスト編集／出力

```
Logger logger = LoggerFactory.getLogger(this.getClass());//ロガー
logger.debug("item empname bef:" + item.getEmpName()); //ログ出力
```

※レベル=DEBUGで出力する例。logback.xmlのレベルをINFOに設定すると出力されなくなるため、このままコードを消さずに運用に移すことができる

```
17
18 import org.slf4j.Logger;
19 import org.slf4j.LoggerFactory;
20 import org.springframework.batch.item.ItemProcessor;
21 import org.springframework.stereotype.Component;
22
24  * ItemProcessor of employees.
26 @Component
27 public class EmployeeProcessor implements ItemProcessor<Employee, Employee> {
28
30      * Processing some stuff for each employee.
35     @Override
36     public Employee process(Employee item) {
37         // To upper case employee name.
38         Logger logger = LoggerFactory.getLogger(this.getClass());
39         logger.debug("item empname bef:" + item.getEmpName());
40
41         item.setEmpName(item.getEmpName().toUpperCase());
42
43         logger.debug("item empname aft:" + item.getEmpName());
44         return item;
45     }
46 }
```

出力例

```
Console  Problems  Debug Shell
<terminated> Excute Job01 [Java Application] C:¥Tools¥AdoptOpenJDK¥jdk-11.0.10.9-hotspot¥bin¥javaw.exe (2021/06/18 1
11:53:25] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname aft:SCOTT
11:53:25] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname bef:virgil
11:53:38] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname aft:VIRGIL
11:53:38] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname bef:gordon
11:53:39] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname aft:GORDON
11:53:39] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname bef:john
11:53:39] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname aft:JOHN
11:53:39] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname bef:alan
11:53:39] [main] [c.e.b.t.j.EmployeeProcessor] [DEBUG] item empname aft:ALAN
11:53:39] [main] [o.s.b.i.s.AbstractFileItemWriter] [DEBUG] Writing to file with 5 items.
```


5. Maven ビルド

開発中は STS に設定されているクラスパス／JRE を使ってコンパイルが行われていますが、実行環境向けにビルドした資材は STS の環境と切り離されるため動作しなくなる場合があります。また、Spring では実行するまで bean 定義がインスタンス化及び DI が行われなず不具合に気が付けないためビルド時には注意が必要です。

5.1. JRE のバージョン

JRE のバージョンは、以下が関係します。

① STS 自体が動作している JRE(JDK) …STS 内臓です
② STS の Window メニュー ＞ Preferences ＞ Java – Installed JREs
③ プロジェクト固有指定 …プロジェクトを右クリック ＞ Properties for *プロジェクト* ＞ Java Build Path ＞ Libraries タグ ＞ Modulepath
④ プロジェクト内の pom.xml <java-version> … Java 8⇒ 1.8 ｜ Java 11 ⇒ 11

※②～④のバージョンは一致させておいてください

5.2. Dependencies

プロジェクトが依存しているライブラリは、pom.xml に<dependency>として定義してダウンロードするか、共有ライブラリ（ローカルリポジトリ）に入っていないと使えません。

Maven を使うと、ユーザのホームディレクトリ配下に「.m2¥repository」というローカルリポジトリが作られます(デフォルトのまま変更していなければ)。STS のプロジェクトに表示される「Maven Dependancies」にはローカルリポジトリの jar ファイルが入っていますが、単にビルドをしただけでは target/lib に入らず、実行時に class not found が発生します。

5.3. ビルドの実行

（1）コマンドプロンプトから実行する場合

① cd コマンドでカレントディレクトリをプロジェクトの場所(pom.xml が入っている)に移す
② 以下の Maven のコマンドを実行する

```
mvn clean dependency:copy-dependencies -DoutputDirectory=lib package
```

※次のことが起こります

・target ディレクトリの削除
・lib ディレクトリに依存ライブラリ(jar)をコピー
・target フォルダ内に実行用の資材を収納
・target フォルダを固めた jar ファイルの作成

（2）STS から実行する場合

① プロジェクトを右クリック ＞ Run As ＞ Maven Build... ＞ [Goals:]に以下を入力し [Run]
“clean dependency:copy-dependencies -DoutputDirectory=lib package” …「“”」は不要です

